TOWARDS A CALCULUS OF TOTAL CORRECTNESS

FOR THE DESIGN OF C-MOS SWITCHING CIRCUITS

C.A.R. HOARE, August 1987.

Summary. This note explores the possibility of extending some of the mathematical methods that have been recommended for the design of software to the design of hardware, in particular of synchronous switching circuits implemented in C-mos. A hardware design can in principle be translated into a simulation program, which with a certain accuracy models the behaviour of actual C-mos circuits. The program may be specified and proved correct with respect to a precondition and a postcondition. More simply, assertional proof techniques can be applied directly to the hardware design. The objective is to guide the design of networks which are known by construction to be correct.

The paper starts with a general philosophical introduction to the role of formal methods in design, which may be omitted. The next two sections describe a mathematical theory of hardware design which is delightfully simple, but seriously inaccurate to the actual behaviour of switching circuits implemented in C-mos. These inadequacies are discussed together with suggestions on how they may be remedied in future research.

0. Design philosophy.

A specification of an engineering product or component is a predicate describing the desired properties of its observable behaviour when put into service. Observable values are represented by free variables occurring in formulae, equations, inequations, or other predicates. The predicates may also include any definable concepts of the relevant branches of pure or applied mathematics. A combination of requirements is expressed as a logical conjunction of the individual predicates. Conjunction is the connective that permits a large and complex specification to be structured from its smaller and simpler components.

If correct operation of the product will depend upon proper conditions of use, then these too can be expressed as a predicate, and correct behaviour of the product is required only when this precondition is true. Thus if the environment or user fails to meet the specified precondition, the specification is vacuously satisfied, without placing any constraint on the

behaviour of the product. Thus the overall specification is expressible as an implication, in which preconditions are listed as the antecedent and the desired behaviour as the consequent.

When the specification has reached a sufficient degree of completeness, an engineering project moves into the design phase. The outcome of the design is also expressed in some formalised notation, often including scale drawings. The design notation is usually quite different and much more restricted and cumbersome than that of the specification. This is because it does not directly describe the behaviour of the product, but rather some technologically sound method for its manufacture. Nevertheless, if the technology is well understood, it is possible to re-interpret the design as an indirect description of the range of behaviours of any product made in accordance with the design. This predicate is the strongest specification satisfied by the design. The correctness of the design can then be proved before manufacture by showing that the design predicate logically implies the original specification; from the meaning of logical implication, it will follow that every observation of every product manufactured in accordance with the design will also accord with the specification. That is exactly what we mean by correctness of a design.

The approach expounded in the previous paragraph presupposed that the completed design is available before the proof starts. For a non-trivial project, this is much too late. Far greater value can be obtained from proofs conducted throughout all stages in the progress of the design. At each stage, the designer plans to build a component of the product out of several smaller sub-components. The method of assembly is decided, and each component is carefully specified. Before proceeding with the design of the subcomponents, a proof should be given that the eventual assembly of subcomponents (meeting their individual specifications) will meet the original specification of the complete component. This too is done by interpreting even the incomplete design as a predicate. Such proofs can be repeated at every stage, in the hope of eliminating one of the most serious problems in large system implementation, namely the diagnosis and elimination of errors detected after assembly of the manufactured components. This design philosophy is encapsulated in the slogan "design right - first time".

The specification and design of a single product to be implemented and remain unchanged throughout its working life is difficult enough. Far more difficult and important is the specification of the architecture of a range of compatible products, capable of adaptation and evolution over a period of many decades, This is the problem that faces the major manufacturers

of aeroplanes, automobiles and computers. It also faces the designer of every large application or systems program, which must be structured from the beginning as a member of the family of programs which it is most likely to evolve into.

Fortunately, the structure of a family of products can be conveniently explored and clearly formalised as a family of predicates. The most general and persistent features of the architecture are expressed in highly abstract terms by a general predicate; the more specific details of specialised subranges and individual products are expressed as separate predicates which can be proved to conform in an appropriate degree to the more general ones. Clarification of the structure of the design space is a serious intellectual challenge; but it provides an opportunity to plan for the multiple use, throughout the working life of the architecture, of the early design steps, the partial implementations, their interfaces, as well as the completed components.

But there is usually a price to be paid for splitting a design into modules with clear general specifications and reasonably simple narrow interfaces. The price is often exacted as an increase in the number of components or lines of code, and a reduction in execution efficiency. If the subassemblies are not intended for disassembly during use, the price may be reduced by subjecting the design at some suitable stage to a series of correctness-preserving optimisations, which disregard and over-ride the original modular structure of the design. Such optimisations may be applied automatically, as in many compilers for a high-level language, or under human guidance. In either case, the validity of the optimisations must be guaranteed by algebraic equations or inequations, which are proved sound for the mathematical theory and notations in which the design is expressed.

An example of our design philosophy is provided by modern techniques for the design of algorithms and programs for a general-purpose computer. A conventional sequential program is specified by a predicate, whose free variables denote initial and final values of the variables manipulated by the program. The precondition on the environment is permitted to mention only the variables denoting initial values.

The end product of program design is expressed in a very formal notation, usually a programming language. The products described by the program are rather intangible; they are the executions of the program. These executions correspond so closely to the structure and content of the program, that their elaboration is entrusted to a mindless computer, and require no further human intervention. In addition to this mechanical

interpretation, there are now available mathematical methods for deriving from the text of a program (expressed in certain restricted languages) the strongest specification which will be met by every execution of the program. The program can be proved correct before execution by showing that this predicate implies the original specification. The correct program can then be optimised if necessary by algebraic transformations which are known to be valid for the programming language in which it is expressed.

In practice, these proofs should be conducted piecemeal during the design of the program. Suppose at some stage it is decided to implement a component with specification R by the sequential composition (X;Y), where X and Y are unknown, because they have not yet been designed. Instead, they are carefully specified by means of an intermediate predicate S , which is intended to be true on termination of X and on initiation of Y . The specification of X has S as its postcondition and the same precondition as R ; the specification of Y has S as its precondition and the same postcondition as R. Now it is obvious that if X and Y meet their respective specifications, then their assembly (X;Y) will meet the overall specification R . That is a general, trivial, but useful theorem of the theory of programming. The specification S (together with appropriate resource allocations for space and time) serves as a provably complete specification of the interface between X and Y ; if all goes well, no further communication will be needed between the implementors of these two components. Similar techniques are available for the other structuring features of a programming language.

Of course, nothing in these calculations will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgement, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalise and communicate design decisions, and ensure that they are correctly carried out.

1. A simple story.

The objective of this note is to show how the proof-driven design methods now available for computer programs may have a possible analogue in the design of a certain class of hardware circuit. In this section we tell a simple story about the behaviour of the hardware. In section 3 we describe its inadequacies, and in section 4 we will try to suggest extensions which may overcome the inadequacies without introducing too much extra complexity.

A switching circuit is designed to be used repeatedly. At the beginning of each cycle of operation, some of the wires (the input wires) are connected to power or to ground. This leads to a succession of switch changes within the network. When these have stabilised, this cycle ends, and certain other wires (the output wires) are found to be connected to power or ground. A switching circuit is fairly symmetric, and the choice of which wires are used for input and which for output may vary from one cycle of operation to the next. We will postpone consideration of the succession of cycles, and deal first with the combinational aspects, i.e., those which relate to just a single cycle of operation.

The specification of a combinational switching logic network can be expressed very simply as a formula in propositional logic. The free propositional variables have the same names as the wires connecting the network to its environment. Each variable takes the value true if the corresponding wire is connected to power when the circuit stabilises, and the value false if it is connected to ground on stabilisation. For example, we define

$$SNEG = (e \equiv \neg a)$$

as the specification of a network which stabilises only when e is the negation of a. A specification of an exclusive disjunction is

$$SEXOR = (e \equiv (a \not\equiv b))$$

These specifications are much simplified by treating input and output wires symmetrically; we will describe later the dangers of the simplification.

The design of a combinational network can be modelled formally as a set of transistors. Each transistor is described by four components

$$(t, g, \{s, d\})$$

where t is the transistor type (N or P)
g is the name of the wire connected to its gate
s and d are the names of the wires connected to its source and drain. Because of symmetry, it is convenient to group these in a set of just these two elements.

The first example of a circuit has four wires (a , b , c , and e) and two transistors, one of each type

$$NEG = \{(N, a, \{c, e\}), (P, a, \{b, e\})\}$$

The next example has the same wires, and two transistors connected in a different way

$$TMG = \{(N, c, \{a, e\}), (P, b, \{a, e\})\}$$

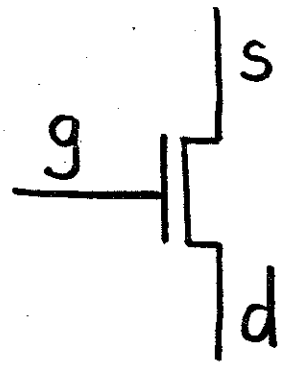
The third example is obtained by combining the two previous ones, and joining up wires with the same name; mathematically, this is achieved by the simple expedient of taking the union of the two sets of transistors

$$EXOR = NEG \cup TMG$$

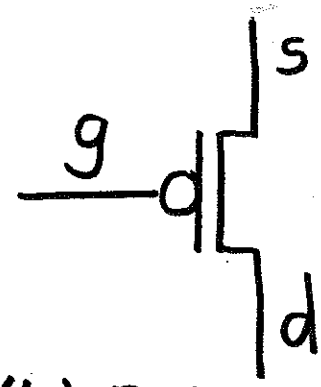
It is usual to display a picture of the network as follows. A wire is drawn as a possibly branching line, with branch-points indicated by a dot; its name is found nearby. A transistor of type N is drawn in any orientation as shown in figure 1(a), with three wires attached. A P-transistor is drawn as in figure 1(b). The other networks defined above are also drawn in figure 1. The geometrical configuration of transistors and wires may correspond to their relative positions when printed onto the surface of silicon.

But here we are not concerned with such implementation details. Rather, our task is to interpret the network design as a predicate describing the behaviour of any implementation of it. The behaviour of a single N-transistor

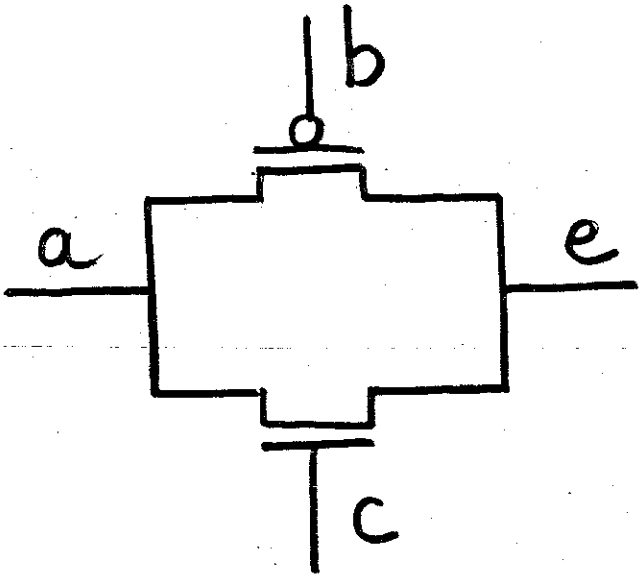
$$(N, g, \{s, d\})$$



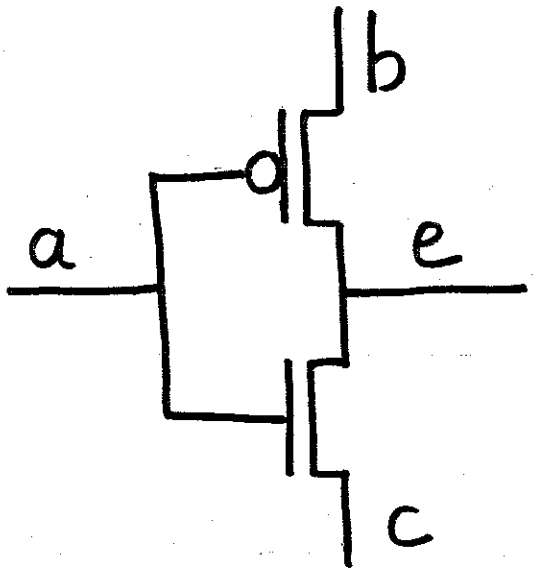
(a) N-transistor



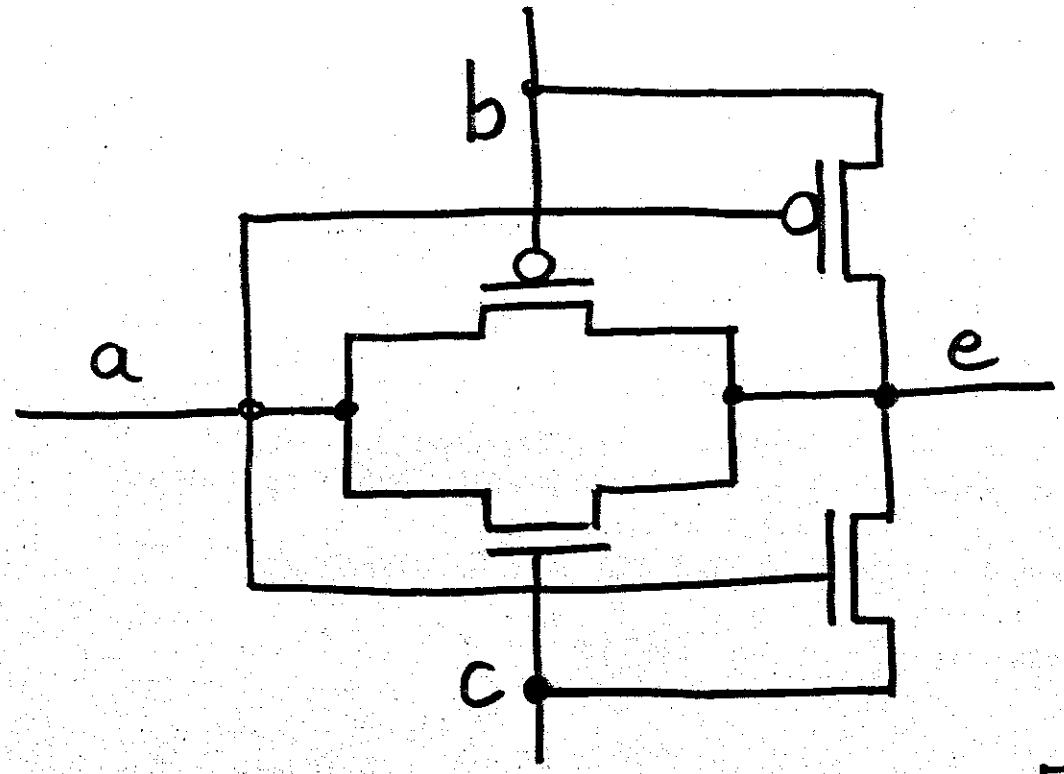
(b) P-transistor



(c) TMG



(d) NEG



(e) EXOR

FIG 1

is as follows. If on reaching stability the value of the gate g is true, a connection is made between s and d , as a result of which they must end with the same voltage. If the gate is false, the values of s and d are arbitrary — they are free to be determined by other transistors connected to the same wires. This behaviour can be expressed by defining the strongest (i.e., exact) specification of an N-transistor to be the proposition

$$g \Rightarrow (s \equiv d)$$

The behaviour of a single P-transistor is similar, except that it is the falsity of the gate that ensures connection and therefore equality of the source and drain

$$(\neg g) \Rightarrow (s \equiv d)$$

The behaviour of a network of transistors is nothing but the propositional conjunction (and) of the specifications of all its components. Thus we calculate

$$B(NEG) = (a \Rightarrow (c \equiv e)) \wedge (\neg a \Rightarrow (b \equiv e))$$

which simplifies to

$$= (e \equiv (\text{if } a \text{ then } c \text{ else } b))$$

$$B(TMG) = (c \Rightarrow (a \equiv e)) \wedge (\neg b \Rightarrow (a \equiv e))$$

$$= (c \vee \neg b) \Rightarrow (e \equiv a)$$

$$B(EXOR) = \text{if } b \equiv c \text{ then } (e \equiv a) \wedge (e \equiv c) \text{ else } e \equiv (a \not\equiv c)$$

We introduce two special wire names, true which is always connected to power, and false which is always connected to ground. These may be used as wire names in the description of transistors and networks. For example, substitute true for b in NEG and false for c to obtain

$$\{(N, a, \{\text{false}, e\}), (P, a, \{e, \text{true}\})\}$$

The behaviour of this network is described by making the same substitution in B(NEG). This then simplifies to

$$e \equiv \neg a$$

which is the specification SNEG of a negation circuit.

The proof technique described above can be extended to bi-directional designs, which are capable of taking their input on occasion from the output wires, and then will present their output on the input wires. For example, a bi-directional exclusive or network can be constructed from EXOR by adding two more transistors,

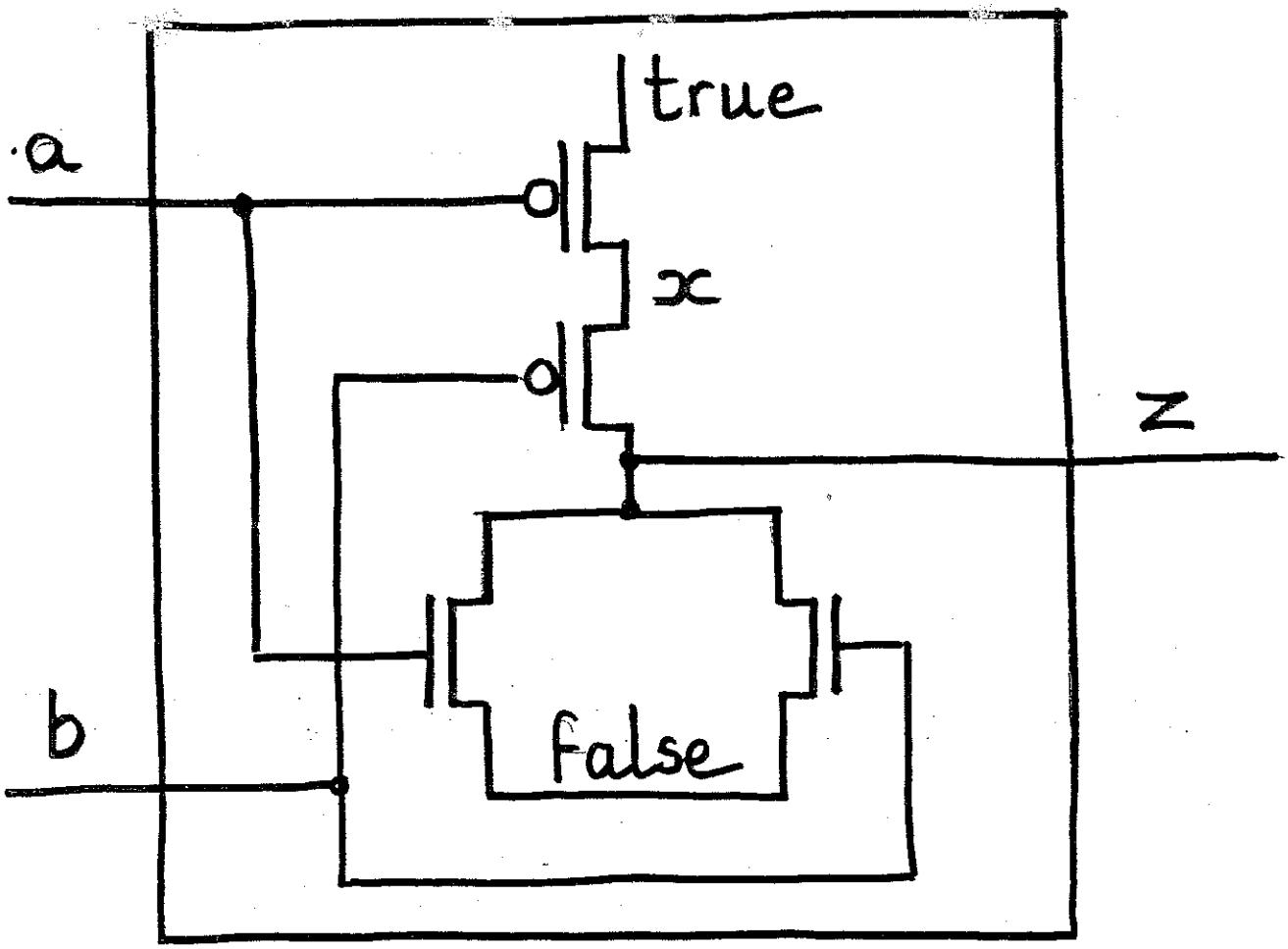
$$EXOR2 = EXOR \cup \{(P, e, \{b, a\}), (N, e, \{c, a\})\}.$$

The reason for the extra transistors is indicated in 3.1. The method may also be applied to sub-networks in which the external connections are designed to be used sometimes for input, and sometimes for output, and sometimes for neither.

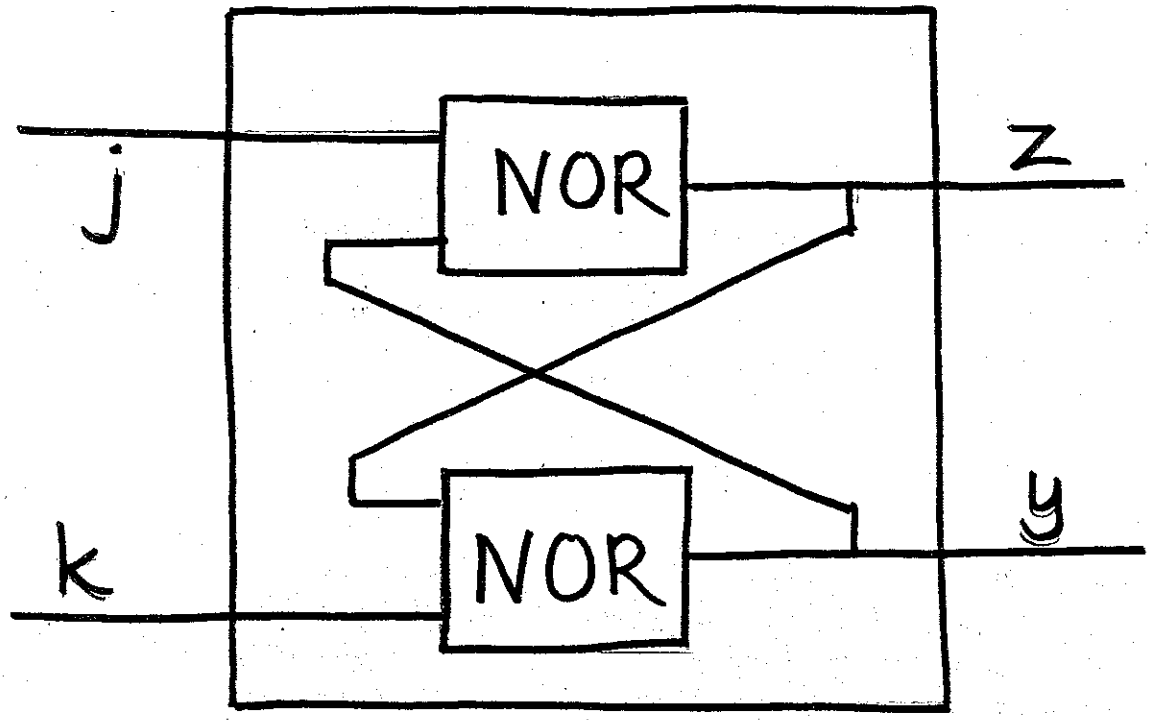
This method is very conducive to proof during design. A specification falls naturally into parts conjoined by $\wedge$. Each part is implemented as a separate network; and when these are joined together in the standard way, they will meet the whole of the original specification. But if the specification R is not conveniently partitioned, it will be necessary to introduce a planned split in the design. Suppose it is suggested that one of the components should meet the specification Q. Then the remaining part of the specification to be met by the other components can be simply calculated as $(Q \Rightarrow R)$. Similar but more complicated calculations, using the weakest prespecification, are available for the design of programs. Unfortunately, the simplicity of the hardware development method described above is delusory, as will be explained in section 3.

2. Abstraction

A large program is usually designed as a collection of somewhat independent parts. Some of these parts will be written as procedures, which will be used repeatedly throughout the program, possibly with different parameters at each call. A similar facility for declaration of sub-assemblies is very necessary in hardware design. For example, a general design for the familiar NOR circuit can be declared in the form of a procedure (see Figure 2)



(a) NOR



(b) SR

FIG 2

NOR (a , b , z) = **loc** x . { (P , a , { true , x }) , (P , b , { x , z }) ,

(N , a , { z , false }) , (N , b , { z , false }) }

where x is declared as a local wire (explained below)
a , b , and z are formal parameters, varied on each call.

To illustrate the calls of this procedure, a SR flipflop can be defined in terms of two NORs

SR = NOR (j , y , z) ∪ NOR (k , z , y)

In a conventional procedure body, it is usual to declare local variables which are used in the procedure body but nowhere else in the program. The effect is that each call of the procedure uses a fresh instance of such a variable, distinct from any variable used in the other calls, or elsewhere. The distinctness of the local variables of each procedure call makes the procedure re-entrant, so that it can be called recursively or even concurrently by simultaneously active processes.

In hardware, every call of a procedure is concurrent with every other call, so local wire names are an indispensible feature of hardware procedures. For example, the wire name x is declared as local to the NOR procedure, so that each of the two calls of NOR within the definition of SR uses a distinct and separate wire name in place of x . In the mathematical definition of a network, I have chosen to indicate locality of a wire name by the declaration

loc x .

Pictorially, it is indicated by enclosing the network in a box, whose boundaries are crossed only by wires in the parameter list, and not by local wires.

The name and value of a wire local to a procedure is inaccessible and irrelevant to the environment of the procedure call. For this reason, the name should not appear in any description or specification of the procedure. Avoidance of local wire names will usually simplify the specification, and so make it easier to use. So we need to find a method of defining the behaviour of a network with a localised wire name. As explained above, we cannot know, or even want to know, the actual value

of the local wire. All we know is that on each occasion of use there exists such a value. This is stated by quantifying over the two possible values of the wire, true and false

$$B(\textbf{loc}\ x\ .\ C) = \exists x\ .\ (x = \text{true} \vee x = \text{false}) \wedge B(C)$$

In future, we will assume without explicit mention that a quantified variable ranges over just the two values true and false. Thus we can calculate the behaviour of NOR

$$B(\text{NOR}(a, b, z)) = \exists x\ .\ (\neg a \Rightarrow (x \equiv \text{true})) \wedge (\neg b \Rightarrow (x \equiv z))$$

$$\wedge\ (a \Rightarrow (z \equiv \text{false})) \wedge (b \Rightarrow (z \equiv \text{false}))$$

By propositional logic, this reduces to

$$[\exists x\ .\ (\neg a \Rightarrow x) \wedge (\neg b \Rightarrow (x \equiv z))] \wedge ((a \vee b) \Rightarrow \neg z)$$

Expand the quantification by substitution of true and false for x

$$[((\neg a \Rightarrow \text{true}) \wedge (\neg b \Rightarrow (\text{true} \equiv z)))$$

$$\vee (\neg a \Rightarrow \text{false}) \wedge (\neg b \Rightarrow (\text{false} \equiv z)))]$$

$$\wedge ((a \vee b) \Rightarrow \neg z)$$

This reduces to

$$z \equiv (\neg a \wedge \neg b)$$

which is just the long-awaited specification of the logical NOR function.

The behaviour of each call of the procedure can be discovered by substituting the actual arguments of the call for the parameter names in this specification. Thus the behaviour of the SR flipflop is

$$B(SR) = (z \equiv (\neg j \wedge \neg y)) \wedge (y \equiv (\neg k \wedge \neg z))$$

which can be rewritten in propositional logic to

$$(j \wedge k \Rightarrow \neg y \wedge \neg z)$$

$$\wedge (j \wedge \neg k \Rightarrow y \wedge \neg z)$$

$$\wedge (\neg j \wedge k \Rightarrow \neg y \wedge z)$$

$$\wedge (\neg j \wedge \neg k \Rightarrow (y \not\equiv z))$$

In the fourth case when both j and k are false , this specification states merely that y and z take different values. However, the actual values taken are left undetermined. This can be an accurate description of the behaviour of the hardware: the C-mos circuit is also non-deterministic when the signals j and k go false at about the same time. More usually, if j goes false earlier z will remain true, and if k goes false earlier y will remain true. This fact is very important in the design of storage elements; but it cannot be described in the simple story. The point is discussed again in 3.4.

3. What goes wrong.

The theory outlined above appears to give a very simple and effective method for correct design of switching networks. Unfortunately, it does not apply to the design of networks for implementation in current C-mos technology. In practice, there are at least five things that can go wrong with designs produced in this way, and five other deficiencies in theory.

3.1 Asymmetry

The logical formula describing the behaviour of a transistor is completely neutral with respect to the direction of input and output. For example, if opposite values are supplied to the source and the drain, the formula $(g \Rightarrow (s \equiv d))$ leads to the conclusion that the gate will be false on stabilisation. But in fact a C-mos transistor cannot change the value on its gate in response to true or false connections made to its source and drain. That explains the need for the extra two transistors in the design of the bi-directional exclusive disjunction EXOR2.

3.2 Weak propagation

The whole theory is based on the presupposition that wires take only two values, true and false . Thus the N-transistor

(N , true , { true , d })

should ensure the truth of d . Unfortunately a C-mos N-transistor is not good at transmitting true between source and drain; in practice, an intermediate value is generated, which we will call "weakly true" .
Similarly a P-transistor

(P , false , { false , d })

will make d only "weakly false" . These weak signals are not capable of controlling the behaviour of other transistors.

3.3 Short circuits

The proof method given above does not protect against the danger of connecting power directly or indirectly to ground. Such a connection is known as a "short", short for short circuit. Consider for example the N-transistor

(N , true , { true , false })

The predicate describing its behaviour is

true $\Rightarrow$ (true $\equiv$ false)

which reduces just to the false proposition. Since in the propositional calculus false implies everything, this transistor will miraculously satisfy every possible specification. But of course there never could exist any product whose behaviour was described by the false proposition. Any theory which assumes the possibility of what is impossible must be at best unrealistic and at worst dangerous.

In practice, the transistor connected as shown above is an immediate short; not only is it useless, but it will render useless the entire chip on which it is printed. In more complicated networks, such shorts may depend on particular unexpected combinations of input values, which occur only after the chip has gone into service.

3.4 Charge

The gate of a C-mos transistor has significant electrical capacitance, which can store a charge from one cycle of operation of a circuit to the next. This property is essential to the the design of storage, registers, and certain kinds of high-speed combinational circuitry which take advantage of precharging. Such networks cannot be treated at all by the simple methods of the previous sections.

3.5 Simulation

A widely used technique for checking a network prior to production is to translate the design into a computer program, and simulate its behaviour on test cases by running the program on the corresponding input data. Apart from assisting in the discovery of errors, a simulation program can help in convincing an engineer of the validity of the logical theory on which a design is based; it can also serve as a "prototype", giving an early check on the appropriateness of the original specification. For small circuits, it can even establish correctness by testing on every possible combination of input values.

For simulation of C-mos switching circuit designs, it is most convenient to use a logic programming language like PROLOG. This is because the proposition

$$g \Rightarrow (s \equiv d)$$

which describes the behaviour of an N-transistor translates directly into four Horn clauses

s if d and g	d if s and g.
nots if notd and g	notd if nots and g.

The complete program is obtained by assembling these clauses for all the transistors.

Unfortunately, this technique does not work for the simple story, because PROLOG does not implement negation in the required manner.

3.6 Design rules

There are other reasons for the weakening of electrical signals passing along wires, which are forbidden by the design rules for silicon. For example, there is a limit on the number of transistors that can be safely

connected by a single wire, and on the number of transistors through which a signal may be passed without amplification. Furthermore, the propagation of signals through transistors takes time, and a circuit will fail unless an appropriate delay is interposed between input and an attempt to access the output of a network.

These concerns are best separated from concerns of purely logical correctness of circuit behaviour. They can be treated independently by syntactically checkable design rules, appropriate for current manufacturing processes. By ignoring such issues, a mathematical theory, and the designs which result from its use, are more immune to changes in technology.

3.7 Top-down design

The definitions given in this paper have shown how to translate a pre-existing circuit design into a predicate which describes the behaviour of any circuit manufactured in accordance with the design. The translated text is in general larger and more complex than the original; so this method is applicable only to small designs. In any case, as explained in section 0, the bottom-up approach is far less valuable than a topdown approach, which starts with a specification (usually quite simple) and transforms it methodically into a more complex design, whose correctness is assured by its history of construction. The real advantage of a formal method will be apparent only when it is used for rapid and routine production of complex designs, which are also more efficient than those which could have been produced by unaided intuition or invention.

In view of the other deficiencies of the simple story, its use as a basis for a top-down design methodology is likely to lead to wholly unrealistic results.

3.8 Continuous models

In the interests of tractability, calculations and reasoning about correctness of logic design are based on a model with relatively small number of values for quantities such as voltage, capacitance, and resistance (in the simple story just true and false). But of course, in a real circuit these quantities range over a complete continuum. In view of the dangers of oversimplification (amply illustrated above), it would be very desirable to check the validity of any proposed discrete model against some well established continuous scientific theory of electrodynamics, for example Maxwell's equations.

In practice, however, a crude discrete theory may be superior to a precise theory, because it is more tolerant of the wide variation in parameters inevitably arising in the mass production of engraved silicon. The main value of the precise theory is to calculate the permissible limits of such variation, within which the approximations made in the discrete model remain valid.

3.9 Algebra

An algebra of C-mos circuit design is most elegantly expressed by writing the name of the gate of a transistor as a subscript to the equivalence symbol; for example we define

$$(s \equiv_g d) = (g \Rightarrow (s \equiv_g d))$$

Like $\equiv$, the operator $\equiv_g$ is reflexive, symmetric, associative, admits

distribution by disjunction, etc.

$$(x \equiv_g x) = \text{true}$$

$$(x \equiv_g y) = (y \equiv_g x)$$

$$((x \equiv_g y) \equiv_g z) = (x \equiv_g (y \equiv_g z))$$

$$((x \equiv_g y) \vee z) = ((x \vee z) \equiv_g (y \vee z))$$

In the case of switching circuits, conventional algebraic laws like these do not promise much assistance in the transformation or optimisation of designs. The reason is that algebraic formulae use nesting as the main method of composition of complex formulae from simple ones, whereas circuit designs use only conjunction.

3.10 General

C-mos switching circuitry is only one among a number of techniques for design of general-purpose or custom-built computing devices. Standard logic gates, discretionary wired gate arrays and programmed logic arrays

raise few of the problems listed above, and often provide adequate efficiency in return for considerably less ingenuity. And when the higher switching speed, of ECL is required, the techniques described in this paper are not applicable at the circuit level. They are equally inapplicable to asynchronous circuit design.

The scope for switching circuit design is further restricted by practical limitations on the number of transistors through which a signal may pass before regeneration. So switching techniques are used mainly to implement simple logic functions like EXOR; and these functions are then composed using simpler and more standard techniques for logic design. The effort of developing a theory and installing a complicated methodology for design of such a restricted class of circuits may be hard to justify.

4. Towards a solution

A solution to some of the problems raised in the previous section may be sought by increasing the number of variables used to describe the behaviour of a circuit from one variable per wire to six or more. There is a corresponding increase in the complexity of the predicate describing each transistor.

4.1. Asymmetry

The first problem of asymmetry is solved by using two propositional variables instead of one to describe the state of each wire in the circuit. Let w be the name of the wire. Then the propositions are

Tw and Fw

where Tw is true if wire w is connected to power when the circuit stabilises (and false otherwise); whereas Fw is true if the wire is connected to ground on stabilisation. It is possible that both of these propositions will be false, in which case the wire is said to be floating. It is possible that both of them will be true, in which case the wire w is part of a connection between power and ground, i.e., a short. The introduction of two separate variables for each wire ensures that a short is no longer a logical contradiction; so a short across the source and drain of a transistor no longer has a miraculous effect on the value at its gate.

4.2. Weak propagation

The second problem is solved by introducing two more variables associated with each wire,

WTw and WFw

The first of these indicates that w is connected to power, but possibly one of the transistors on the connecting path is only weakly conducting. A similar interpretation is given to WFw. For convenience this definition has been formulated in such a way that a strongly connected wire is also weakly connected; so it is axiomatic that

$Tw \Rightarrow WTw$ and $Fw \Rightarrow WFw$ for all w

Since a weak signal is anyway unusable, we do not need to model the fact that weak conductance of a weak signal makes it weaker still.

The behaviour of transistors can now be more accurately described than in the simple story. An N-transistor strongly conducts a strong false signal when its gate is strongly true. When the gate is strongly false it does not conduct at all. In all other cases, the transistor weakly conducts any signal, either weak or strong.

$$B(N, g, \{s, d\}) = (Tg \Rightarrow (Fs \equiv Fd)) \wedge (\neg Fg \Rightarrow (WTs \equiv WTd)) \wedge (WFs \equiv WFd))$$

4.3. Short circuits

It is a reasonable simplification to treat weakly connected shorts as just as bad as strongly connected ones. Thus the possibility of a short is indicated by the existance of a wire w for which WTw and WFw are both true. Thus the absence of a short can be proved by showing that at least one of these remains false in every permissible combination of input values to the circuit. Proof of absence of shorts is the only reason for modelling weak propagation.

To maintain validity of our proof method, it is necessary that a short is not hidden when abstraction is used to render invisible the name of the guilty wire. For this reason, we introduce a single further propositional variable "sh", which is true of a network just when it contains a short. Thus it is axiomatic that

$$WFw \wedge WTw \Rightarrow sh \qquad \text{for all } w$$

To maintain the realism of our theory, the formula for behavioural abstraction must be modified to

$$B(\textbf{loc}\ x\ .\ C) = \exists Fx, Tx, WFx, WTx.$$
$$(Fx \Rightarrow WFx) \wedge (Tx \Rightarrow WTx)$$
$$\wedge\ (WFx \wedge WTx \Rightarrow sh)$$
$$\wedge\ B(C)$$

Unfortunately, proof of absence of a short is still not simple. The formula describing the behavioural predicate for a circuit is defined in such a way that it it satisfied when **all** the propositional variables (including sh) take the value true. This means that it is **always** impossible to prove the desired implication

$$B(C) \Rightarrow \neg\ sh$$

What is required is more like a proof that the behavioural predicate is satisfiable by an interpretation which assigns the value false to sh. This complication is the price that must be paid for modelling composition as the conjunction of predicates. Further explanation of the details is beyond the scope of this paper.

3.4. Charge

To model the storage of charge, we introduce two more propositional variables for each wire w, namely

CTw and CFw

CTw is true when

both (a) Tw or CTw was true at the end of the previous cycle of operation
and (b) The charge on wire w has not been dissipated on the current cycle.

The meaning of CFw is analogous.

A charged wire behaves exactly like a driven wire in opening or closing the gate of a transistor. So the definition of an N-transistor needs to be extended

$$B(N, g, \{s, d\}) =$$

$$((Tg \vee CTg) \Rightarrow (Fs \equiv Fd))$$

$$\wedge ((\neg Fg \wedge \neg CFg) \Rightarrow (WTs \equiv WTd)$$

$$\wedge (WFs \equiv WFd))$$

The above formula correctly reflects the fact that a charge cannot propagate between the source and the drain of a transistor. Indeed, such propagation would dissipate the charge; this will make CTw and CFw both false, as described below.

Charge dissipation cannot occur on the current cycle if the relevant wire is isolated from the rest of the circuit. The perimeter of a wire is defined as the set of transistors to which it is connected as source or drain

$$perim(w) = \{ t \mid w = st \vee w = dt \}$$

A wire is isolated if all the transistors at its perimeter are switched off

$$isol(w) = (\forall t.\ t \in perim(w) \Rightarrow off(t))$$

where

$$off(N, g, \{s, d\}) = Fg \vee CFg$$

and $$off(P, g, \{s, d\}) = Tg \vee CTg$$

The reason for introducing charge into our model is to permit the design of sequential circuits, whose behaviour unfolds over many cycles of operation. On each cycle, enough time is allowed for the circuit to stabilise before a change to one or more input wires (often a clock) causes the next cycle of operation to start. A charged wire which is isolated on the current cycle will retain the charge it had at the end of the previous cycle.

The easiest way to extend the methods described in this paper to the more general case of sequential circuits is to reinterpret all the propositions Tw , Fw , WTw , WFw , CTw , CFw as infinite **sequences** of Boolean variables. We can therefore define a shift operator Δ on these sequences, such that ΔP takes the same value at time $t+1$ as P took at time t, and was false at time 0.

Now we can define the conditions under which wires are charged to power or to ground.

$$isol(w) \wedge (\Delta Tw \vee \Delta CTw) \Rightarrow CTw$$

$$isol(w) \wedge (\Delta Fw \vee \Delta CFw) \Rightarrow CFw$$

Recall that isol (w) stands just for a conjunction of normal propositions (Tg $\vee$ CTg) and (Fg $\vee$ CFg) for all the wires g connected to the gates of transistors whose sources or drains are connected to w.

The formulae given above seem to deal adequately with the conventional

techniques of precharging. However, they ignore the problem of charge sharing. Charge sharing occurs when a wire, which in the stable state will be isolated, loses too much of its charge before the transistors on its perimeter have fully switched off. This problem can sometimes be averted by careful timing of clock signals.

Another problem ignored in this model is that a charge on a wire will decay spontaneously after a certain number of cycles during which the wire has not been connected to power or to ground. The model could be extended to deal with this gradual decay, but there may be simpler ways of avoiding it.

The technique described above is also incapable of dealing with such circuits as the SR flipflop, in which each of a pair of charged wires is isolated with the aid of the charge on the other. A solution to this problem may be to allow the circuit designer to annotate the circuit by specifying the conditions under which each wire is expected to retain its charge. These annotations are like the assertions used in program verification. They have no significance on the operation of the circuit; they are needed only to establish correctness.

Of course, the validity of the assertion itself must also be established by a proof that each wire which has been asserted to retain charge will actually be found to be isolated at the end of the current cycle. And for this, a weaker definition of isolation would be allowable. But the details are beyond the scope of this paper.

Acknowledgements

For inspiration, encouragement, and helpful comments to Mike Gordon, Geraint Jones, Geoff Brown, Mani Chandy, Mohamed Gouda, Chris Lengauer, Roy Jenevein, David Wheeler and others yet to come.