HIGH LEVEL DESIGN NOTATIONS

Martin Rem

Rapporteurs: Dr. S. Jones
Mr. P.R.H. Place

Abstracts:

1. A Notation for Designing Restoring Logic Circuitry in CMOS

A program notation will be introduced in which every syntactically correct program specifies a restoring logic component, i.e. a component whose outputs are permanently connected via 'not too many' transistors to the power supply. It is shown how the components thus specified can be translated into transistor diagrams for CMOS integrated circuits. As these components are designed as strict hierarchies, it is hoped that the translation of the transistor diagrams into layouts for integrated circuits can be accomplished mechanically, i.e. without interference from or consultation of the programmer. (In this lecture the dynamic behaviour of components, i.e. how they react to transitions on their outputs, will not be addressed).

2. Some Observations on Partially Ordered Computations

In general, a partially ordered computation consists of a collection of subcomputations and relations between them. Some computations are not subdivided and are called atoms. A trace is a finite-length sequence of atoms. The role of sets of traces as a formalism for characterising partially ordered computations will be explored and a composition rule will be introduced, i.e. a method of constructing the set of traces characterising a computation from the sets of traces of the constituting subcomputations, together with a program in which every program corresponds to a partially ordered computation.

A NOTATION FOR DESIGNING RESTORING LOGIC CIRCUITRY IN CMOS

Martin Rem
Eindhoven University of Technology
and California Institute of Technology
and
Carver Mead
Professor of Computer Science, Electrical Engineering
and Applied Physics
California Institute of Technology

1. INTRODUCTION

As the underlying silicon fabrication technology has become capable of producing chips with transistor counts in excess of 1,000,000, problems associated with correct design are assuming ever greater importance. Exhaustive checking of mask artwork for errors becomes prohibitive. Technologies and design styles which obviate large classes of potential errors are enormously preferable to those that do not.

A modular, hierarchical design style can, with proper restriction, confine many types of checks to one level of the hierarchy within each module. A set of such restrictions is given in this paper, together with a mechanism for their enforcement. These restrictions capture a substantial fraction of the design style given in [1].

As feature sizes are scaled below one micron, ratio logic processes like nMOS and I^2L become progressively less attractive. Straightforward scaling to smaller sizes results in a linear increase in current per unit chip area. Technological tricks such as high resistivity polysilicon pullup devices or very small injector current can be used to decrease current drain, but the resulting devices become increasingly vulnerable to "soft error" problems from alpha particles, etc. Fully restored "static" logic using a complementary process is the natural choice for systems with submicron components. Present bulk CMOS processes have a number of very ugly analog rules associated with the 4-layer nature of the process. As a result, the designer must be aware of details of the technology to an alarming degree. CMOS on an insulating substrate is, on the other hand, a conceptually clean process: it requires no analog rules whatsoever if proper timing conventions are observed. There are recent signs that it may become reliably producible as well.

We introduce a programming notation in which every syntactically correct program specifies a restoring logic component, i.e., a component whose outputs are permanently connected, via "not too many" transistors, to the power supply. It is shown how the specified components can be translated into transistor diagrams for CMOS integrated circuits. As these components are designed as strict hierarchies, it is hoped that the translation of the transistor diagrams into layouts for integrated circuits can be accomplished mechanically.

In this paper we do not address the dynamic behavior of the logic components. The "proper timing conventions," alluded to above, are left for a subsequent paper.

2. SWITCHES IN CMOS

The CMOS technology uses two types of transistors: the N-channel enhancement transistor (1a) and the P-channel enhacement transistor (1b).

Fig. 1

Both of them act as switches but they are "on" and "off" for complementary values on their gates. Denoting a high voltage by "1" and a low voltage by "0", switch 1a is on if the gate is 1 and 1b is on if the gate is 0. When the switches are on, however, they do not convey a 1 and a 0 on their paths (in Fig. 1 the horizontal connections) equally well. Switch 1a conveys a 0 virtually perfectly, but it is not a perfect switch for a 1. Switch 1b, conversely, is a good conveyor for a 1 only.

Using these CMOS transistors we want to make two types of switches, a "normally-off" switch (2a) and a "normally-on" switch (2b).

Fig. 2

If the gate is 0 switch 2a is off (nonconveying) and 2b is on (conveying). Otherwise 2a is on and 2b is off. The points e1 and e2 are called the end points of the switch. We call the connection between the end points its path. If nothing is known about the values conveyed through its path, except that they are 0's and 1's, the realization of a switch requires two transistors: (the complement of g is denoted as g')

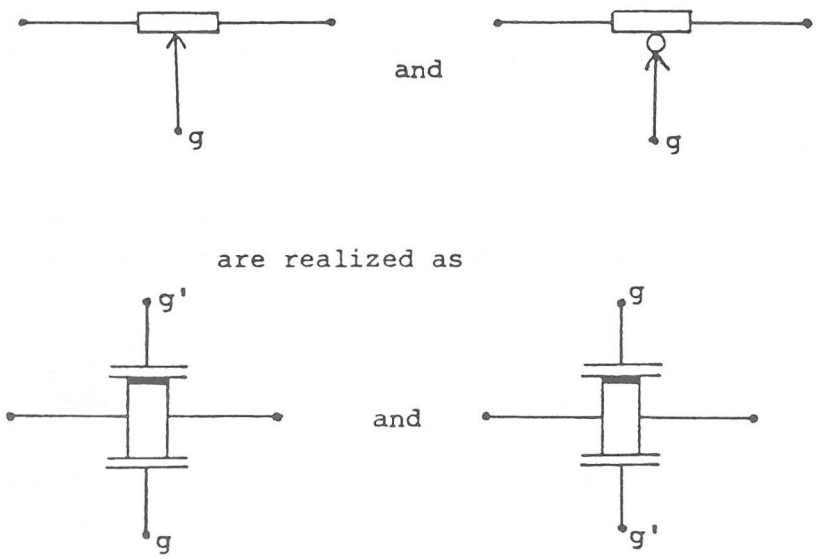


Fig. 3

These double transistors make our switches good conveyors for both 0's and 1's, which allows the use of longer strings of switches. These strings of switches, however, should not be too long: the distance to the "power supply" must not be excessive, otherwise the signal will become inaccurate and the circuit slow. To do justice to the nature of restoring logic we disallow the driving of external outputs by long strings of switches. This shall be reflected in the composition rules to be formulated in Section 3.

The gate inputs are run in two-rail logic to accommodate both the g and the g' signals. For switches that are known to convey always the same value there are two instances in which they can be realized by just one transistor:

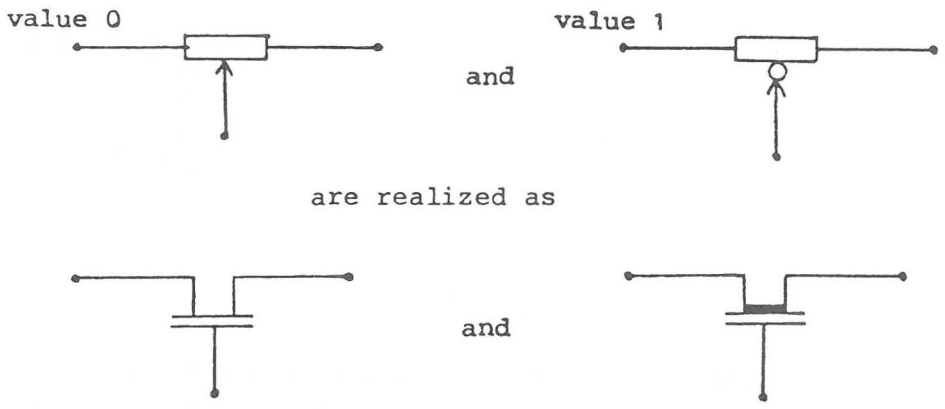


Fig. 4

In that case, the two-rail representation of the gate signal is not necessary. It is assumed that the compiler can recognize instances in which one transistor suffices. From now on we shall simply design in terms of switches and apply the above knowledge only if we wish to count the number of transistors a component requires.

3. RESTORING LOGIC COMPONENTS

A restoring logic component (RL) has external ports. The purpose of an RL is to establish a relation between the values it communicates via its external ports. We restrict ourselves to the values 0 and 1.

We design components in a hierarchical fashion. A typical RL is shown in Fig. 5.

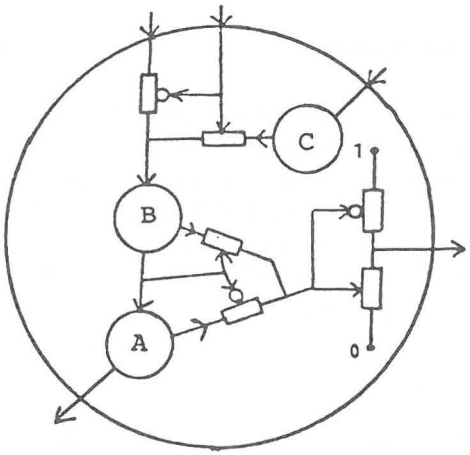


Fig. 5

It consists of subcomponents A, B, and C, which are also RL's, and a pattern of connections between them. We restrict the possible connection patterns to guarantee that the composite is again an RL. Such restrictions are only useful if they can be formulated in terms of the connection pattern, i.e., independent of the internal structures of the subcomponents thus connected. Before we can formulate these connection rules we have to give a few definitions. Each port is either an input port or an output port. The connection pattern of an RL specifies connections between its external ports and the external ports of the subRL's. We call the external ports of a subRL internal ports of the RL. An external output port of a subRL is an internal input port of the RL. Conversely every external input port of a subRL gives the RL an internal output port. The rules on connection patterns will be stated in terms of external and internal ports of the RL.

We assume that the distribution of power and ground to all components is taken care of by the compiler. Johannsen [1] has outlined a method for the distribution of power and ground over hierarchically defined components. In our nomenclature: each RL has two constant internal input ports, denoted by 0 and 1. These constants are the power supply rails which must be present in every component.

In Section 2 we have introduced the term path for the connection between the two end points of a switch. We now generalize that term. We say that there is a path between two ports p1 and p2 if either they are connected by a wire (a "wire path") or there is a switch such that there are paths between p1 and one end point of the switch and between p2 and the other end point. In the latter case we say that the switch is on the path. A path is called a conveying path if all switches on

the path are on. The values on the input ports (external or internal) determine which switches are on and which are off, and hence between which ports there are conveying paths. (Whenever we do not specify whether a port is external or internal, that is done intentionally.)

Two input ports are said to be fighting if there exists any assignment of values to all input ports such that there is a conveying path between the two input ports.

We introduce three rules the connection pattern must satisfy:

Rule 1. [no fighting]: No two input ports are fighting.
Rule 2. [restored external outputs]: Every external output port
(a) has a wire path to an internal port, or
(b) has a conveying path to 0 or 1 for every assignment of values to all input ports.
Rule 3. [nonfloating internal outputs]: For every internal output port p and for every assignment of values to all input ports there is a conveying path between p and an input port.

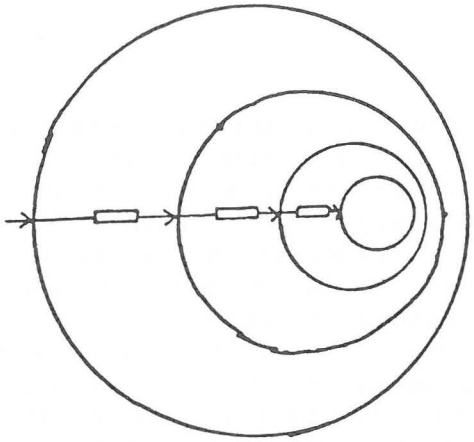
Notice that Rule 1 includes 0 and 1 (the two constant internal input ports). Remember that internal outputs are regarded as (external) inputs of the subcomponent and that the subcomponent's external outputs are internal inputs for the component.

The justification of Rule 1 is obvious. The result of Rule 2 is that all external outputs are driven by power or ground. They may be driven via a number of switches, but such a string of switches is confined to one component, viz. the component in which the actual connection to 0 or 1 is made.

The rules for internal outputs, i.e., outputs to subcomponents, are more liberal. We allow that inputs from subcomponents and inputs from the environment are directed through switches before they are output to subcomponents. For inputs from subcomponents this is reasonable: they are restored by the subcomponents. With inputs from the environment we have to be more careful. We have to allow that such a signal from an external input port goes through a switch to an internal output port. Otherwise we would be unable to make the flip-flop to be shown in Example 3. But it does allow long strings of switches "going into" the hierarchy, as sketched in Fig. 6.

We do not consider this a serious drawback. One may expect a subcomponent to have (physically) shorter connections than the component itself. Restoring in the "inward" direction, therefore, seems less vital than in the "outward" direction. Still, if we wish to bound the lengths of such inward strings of switches we could have the compiler insert amplifiers into them to restore their signals.

The consequence of allowing the switches in the outputs to subcomponents is that Rule 2 has to be stronger than one might expect. In Rule 2 we could not allow wire paths between external input ports and external output ports. This may seem to disallow running through a



—▭— stands for a connection via one or more switches

Fig. 6.

component wire whose signals are not used by the component. In fact, it does not. Such a wire is just not part of the component. (On the chip a wire between two components may run through the "area" of another component, but that is a matter of chip layout. It is a physical property, not a functional one.) Allowing wire paths between external input ports and external output ports would have given rise to the possibility of ill-restored outputs. Fig. 7 sketches an RL that is allowed by Rules 2 and 3. Now assume that each S_i is just a wire path from its input to its output, which would be allowed if we weakened Rule 2. The output of the RL is then not restored. Imagine now that each S_i actually has the same structure as the whole RL. It is clear that this would violate our goal of having restored external outputs.

In one respect is Rule 3 stronger than necessary. It requires that all subcomponents receive well-defined inputs, even a subcomponent whose ouputs are not used. We could have restricted the rule to subcomponents whose outputs are actually used in the computation, but that would have made both the rule and the checking whether it is obeyed more complicated.

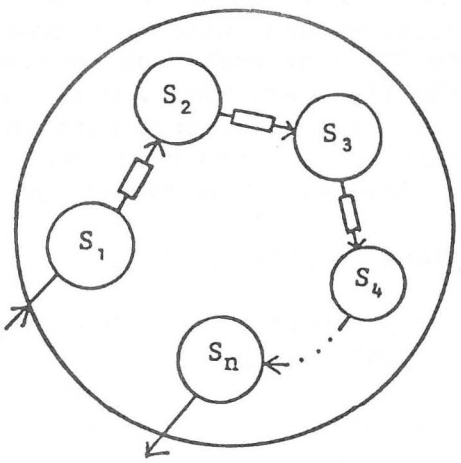


Fig. 7

4. THE PROGRAMMING NOTATION

In this section we introduce a programming notation in which connection patterns can be specified that satisfy the three rules of the preceding section. There are two properties a good notation should enjoy. First, it should be relatively simple for the compiler to check that a program is syntactically correct. If this mechanical check is simple, it will probably be simple for programmers to convince themselves that their designs satisfy the rules. We shall show how the syntactic checking can be performed. Second, it should be possible to give a formal definition of the semantics of our programs. We have not yet achieved the second goal, but ultimately we must be able to prove that a component performs a certain computation. That seems a much better technique than a demonstration of its effect with an a posteriori simulation. (Besides, how do we know that the simulation is correct if we do not have a rigorous definition of the meaning of our statements?) It will not be simple, but remember: a program of more than, say, 20 lines is probably too long, we then have not chosen the right subcomponents.

For the formulation of connection patterns we introduce the term node. Every port is a node, but the program may introduce additional (interior) nodes. For each node n we shall introduce a connection condition C(n) and a connected-to-constant condition CC(n). We shall, furthermore, distinguish a directly driven set D, which is a subset of the set of nodes. These concepts will be used in the syntax checking. A formal definition of how they depend on the connection pattern specified will be given later. Intuitively, C(n) will be the condition on the input values under which node n is connected to an input, and CC(n) will be the condition under which it is connected to a constant. The C(n)'s will be used to enforce the no-fighting rule. The set D will comprise all nodes that are connected by a wire path to an internal input port.

The program consists of a sequence of statements. Each statement introduces a number of connections and switches between nodes, and thereby affects the C(n) and CC(n) of each node involved and the set D. Initially, i.e., prior to the first statement, D is the set of all internal input ports, C(n) is 1 for each input port and CC(n) is 1 for the two constant internal input ports. The C(n) and CC(n) are 0 for all other nodes. ("1" should be interpreted as "true" and "0" as "false.")

The program is complete if finally we have:

for every external output port p : $p \in D \vee CC(p) = 1$
for every internal output port p : $C(p) = 1$

(These completeness conditions correspond to Rules 2 and 3. The observing of Rule 1 is discussed below.)

EXAMPLE 1

```
comp inverter (in?,out!):
begin in' → out = 1; in → out = 0 end
```

The above is a simple example of an RL, it does not have subRL's. The first line specifies the name of the component and its external ports. A question mark or an exclamation point indicates that the port is an input port or an output port, respectively. In the connection pattern two switches are specified, textually separated by a semicolon. The first statement expresses that the output port out is connected to the constant input port 1. The condition in front of the arrow specifies under which circumstances the switch in the connection should be on. In this case a normally-on switch whose gate is connected to the input port in (or a normally-off switch with its gate connected to in') is specified. The second statement specifies the second switch.

For the more pictorially inclined reader we observe the resemblance of the program and the following diagram.

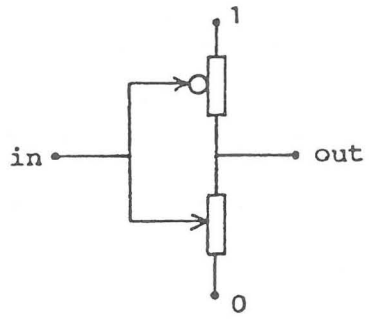


Fig. 8

Why is the program syntactically correct? In order to be able to show that the only output port out satisfies

$$out \in D \vee CC(out) = 1$$

we have to be more precise as to how a statement affects C(n), CC(n) and D.

In a program switches are introduced by statements

$$BE \rightarrow x = y$$

in which x and y are nodes, and BE is a boolean expression in terms of nodes, more precisely: BE is a production of the grammar

<boolean expression>::= <term> { ∨ <term> }

<term>::= <factor> { ∧ <factor> }

<factor>::= <primary> | <primary>'

<primary>::= <node> | (<boolean expression>)

Prior to the statement

$$BE \rightarrow x = y$$

we should have

$$\text{for all nodes } n \text{ in } BE : C(n) = 1, \text{ and}$$

$$(C(x) \wedge C(y) \wedge BE) = 0$$

The first requirement is introduced to permit the syntax checking to be done incrementally at each statement of the program. A consequence, however, is that not every order of the statements in the program is permissible. It is still an open question whether this serializability requirement is not too strong. If we succeed in designing our components under this regime it will certainly enhance both the readability and the checkability of our programs.

The second requirement guarantees the observance of the no-fighting rule. The statement does not have an effect on the set D. The effect on C(n) and CC(n) is

$$Z(x) := (Z(x) \vee (Z(y) \wedge BE))$$

$$Z(y) := (Z(y) \vee (Z(x) \wedge BE))$$

in which Z stands for C or CC.

The set D is affected only by a statement that specifies a direct connection, i.e., one that does not go through a switch. We obtain such a statement by dropping the conditional part "BE→":

$$x = y$$

As for the effect on C(n) and CC(n) this statement is like a switch specification with "1" as its boolean expression. Prior to the statement the condition

$$(C(x) \wedge C(y)) = 0$$

should hold, and its effect is that Z(x) and Z(y) both become Z(x) ∨ Z(y) (Z still standing for C or CC). The effect on the set D is that if either node x or node y was a member of D then D is extended with the other node.

In the example of the inverter we initially have out $\notin$ D. As the program leaves the set D unchanged we have to show that it establishes CC(out) = 1. The first statement is legitimate as we initially have C(in) = 1 and

$$C(out) \wedge C(1) \wedge in' = 0 \wedge 1 \wedge in' = 0$$

The effect is that both C(out) and CC(out) become in'. The second statement is legitimate as well: C(in) is still 1 and

$$C(out) \wedge C(0) \wedge in = in' \wedge 1 \wedge in$$
$$= 0$$

It establishes CC(out) = in' ∨ in, which is 1. Hence, it is a complete program.

Notice that both switches in the inverter are of the type that can be implemented by one transistor. The inverter, consequently, requires only two transistors. We shall use this inverter as a subcomponent in our third example.

EXAMPLE 2.

```
comp nor(a?, b?, out!):
begin a ∨ b → out = 0; a' ∧ b' → out = 1 end
```

In the first statement the boolean expression is a disjunction of two nodes. This gives rise to a diagram in which two switches are placed in parallel. The boolean expression of the second statement specifies two switches that are placed in series. The whole component requires four transistors. The following picture shows a diagram of the component.

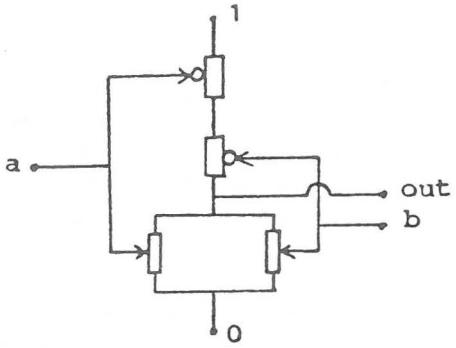


Fig. 9

A new node is introduced by mentioning it in the right-hand side (in the part to the right of the arrow) of a statement. There is no example of this in the paper.

EXAMPLE 3.

```
comp flip-flop(in?, ld?, q!, qbar!):
begin sub i1,i2: inverter;
      i2.in = i1.out;
      ld' → i1.in = i2.out; ld → i1.in = in;
      q = i2.out; qbar = i1.out
end
```

The second line of the program specifies that the component flip-flop has two subcomponents, named i1 and i2, of type inverter. As each inverter has two external ports, this declaration provides the component with four internal ports. An internal port that corresponds to the external port p of a subcomponent S is denoted as S.p. As both i1 and i2 have an external output port out, the component flip-flop has the internal input ports i1.out and i2.out. Likewise, it has the internal output ports i1.in and i2.in.

The reader is encouraged to check that the component satisfies the rules by formally deriving that all statements are legitimate and that the program establishes

$$q \in D,\ qbar \in D,\ C(i1.in) = 1,\ C(i2.in) = 1$$

A possible diagram or the component is

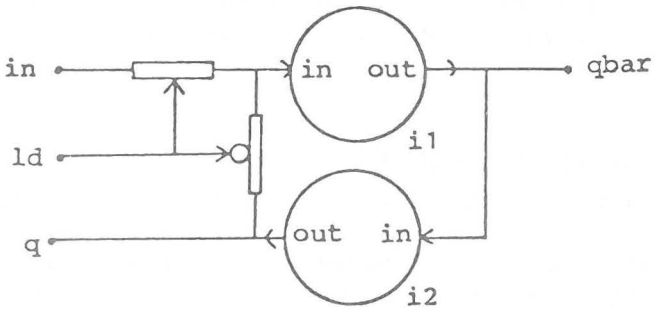


Fig. 10

5. BUSES

If we want to design a random access memory out of inverters, we must be able to connect their inputs and outputs via buses to the inputs and outputs of the memory. We want to connect the outputs of many subcomponents (inverters) to the same bus. Just connecting these outputs (internal inputs to the memory) to the bus would violate the no-fighting rule. We shall remedy this by putting switches in these connections.

To indicate when the memory cell has to drive the bus ("reading") and when it has to receive a value from the bus ("writing") two inputs, r and w, go into the cell:

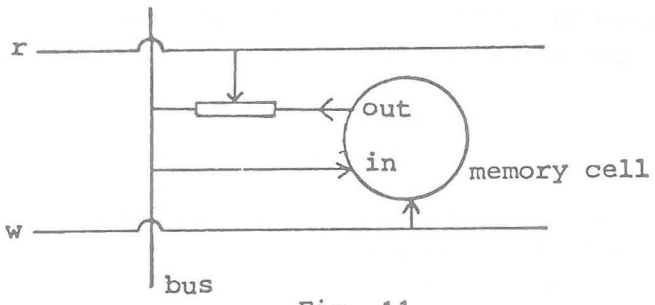


Fig. 11

We attach a number of cells to the same bus. Such a composition will only be an RL if we guarantee that, at most one of the cells can have its r equal to 1. The signals r come from another subcomponent of the memory, usually called the "decoder." The purpose of the decoder is to assure that at most one r equals 1. Given that the outputs of the decoder satisfy that requirement, we can show that the composition is again an RL. This is a new phenomenon: a condition on the values output

by a subcomponent has to be taken into account to prove that a connection pattern specifies an RL. We call such a check a semantic check.

The following program is a 1-of-2 decoder.

```
comp 1-of-2 decoder(in?, out1!, out2!):
begin in → out1 = 1; in → out2 = 0;
      in' → out1 = 0; in' → out2 = 1
end
```

By a syntactic check, as described in Section 4, we can show that this is a legitimate RL. In this case it is also simple to check that the output values satisfy (out1 ∧ out2) = 0, but that is a semantic check.

The moral is that we will design components that are only "conditional RL's," i.e., they are RL's under the condition that the output values of other components satisfy certain constraints. When such components are put together we will have to see to it that such semantic constraints are indeed satisfied.

6. A GLANCE INTO THE FUTURE OF COMPUTING

In this paper we have not addressed the dynamic behavior of components, i.e., how they react to transitions on their inputs. That is obviously the next step. By adopting proper timing and signaling conventions (cf. Chapter 7 of [2]) one should be able to address the dynamic behavior in an equally discrete fashion. The purpose of such conventions is to generate "data valid" inputs that signal that the input data are well-defined and may be inspected. Such a data valid signal may come from a clock or it may be an asynchronous acknowledge signal.

After that there are two roads we can follow. We can make a machine. That machine will accept programs and execute them. We then concentrate on the programs and if we wish to have a certain computation performed, we write a program for it. That is the traditional road.

We are led to the other, more promising, road if we observe that we are already designing programs, programs that can be compiled into transistor diagrams for CMOS. We make components out of subcomponents. Every time they will be more "powerful" or "sophisticated" than their subcomponents. We can inspect how a component is implemented by looking at its program text to see how it is composed out of subcomponents. Every component is again an implementation of a "higher level" concept. We can, e.g., introduce components that communicate other data types than just 0's and 1's. If we look at the implementation of that concept, we may notice that it is achieved by multiplexing or by the use of multiple ports. In that way the components we introduce will give us new modes of expression so that we can formulate our programs in terms of concepts that are more appropriate to our computations. After a while, we will have a mode of expression that one would customarily call a "higher level programming language."

Throughout all the levels of the hierarchy we have maintained that we program by composing components out of communicating sub-components. But by expressing a program in such a notation we have also specified an implementation for it, we have actually specified for the program a transistor diagram in CMOS. From there, the step to a complete silicon compiler is a (nontrivial) matter of generating the proper geometric representation of the transistor diagrams.

Of course, we do not have to translate all our programs into silicon to have them executed. We could also compile them into machine code, e.g., into code for a machine designed by taking the other aforementioned road. Our choice will depend on such external factors as the speed with which the computation has to be performed or the expected frequency of its use. It is also possible that we want to make a translation into machine code first in order to get some experience with the program and that we do not have it compiled into silicon until it is in a form that suits us.

POSTSCRIPT

Is this an article about machine design or about programming? The answer to that question is definitely "Yes!".

ACKNOWLEDGEMENTS

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order Number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

REFERENCES

[1] Johannsen, Dave, "Hierarchical Power Routing." Display file 2069, Computer Science Department, California Institute of Technology, Pasadena, CA, October 1978

[2] Mead, Carver & Lynn Conway, "Introduction to VLSI Systems." Addison-Wesley Publishing Company, Reading MA, 1980

Discussion

Dr. Rideout felt that the terms "normally-on" and "normally-off" were somewhat confusing. **Professor Rem** agreed that terms other than "normally-on" and "normally-off" should have been chosen. He then added that the compiler chooses N or P type transistors based on the analysis of the program.

Professor Dijkstra asked if the validity check leads to a combinatorial explosion depending upon the number of input parts. **Professor Rem** said that this could be so, but that he ordered the statements to prevent this. The ordering is such that any expression to the left of an arrow must have the connection condition true. All components produced so far have been designed this way, but it is unclear if any designs are excluded by this ordering.

Professor Michaelson asked about problems arising in the analysis of a component if any of the sub-components had memory, such as a flip-flop. He felt that the connected-to-constant condition would then depend upon the previous history of the system. **Professor Rem** replied that this was not so for instantaneous states and that timing would be considered later.

Professor Katzenelson said that the flip-flop example looked dynamic and asked if it was dependant upon the ordering of the statements. **Professor Rem** replied that the order of the statements was immaterial except as stated previously.

Some observations on partially ordered computations

Martin Rem

Jan L.A. van de Snepscheut

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513
5600 MB Eindhoven
The Netherlands

1. Introduction

A program specifies a relation between input and output values. As the number of possible input values may be infinite, that relation cannot be specified by listing all pairs of values satisfying the relation. In a program that relation is, therefore, defined by an effective procedure specifying how the output values can be derived from the input values. That procedure must be expressed in a notation that allows the derivation of the output values to be mechanized.

One form the effective procedure can take is that of an arithmetic expression. Its free variables then denote the inputs. This approach is known as functional programming. We adopt a different view: a program consists of subprograms (or statements) and relations between these subprograms. The meaning of a program depends solely on the meanings of the subprograms and the relations between them. Order, as expressed by the semicolon in many program notations, is a possible relation between subprograms. The more order a program expresses the more constrained the range of possible mechanizations is. Of course, one could try to localize irrelevant order expressed in the program, but such a "search for potential concurrency" is in general a difficult task to mechanize. We would rather have a way of programming that does not introduce irrelevant order in the first place. Ideally, any order expressed in the program should be necessary for the program's correctness. We may wish to introduce order to exclude inefficient mechanizations. But be careful: whether a program can give rise to inefficient mechanizations depends on properties of the media in which the mechanizations are realized, and we would like our programs to allow a wide range of possible mechanizations. Our notation should be sufficiently general to allow its usage as a design notation for VLSI circuits.

2. Programs denoting sets of traces

We have said that a program specifies a relation between the values it communicates with its environment. We call our programs components. Each component defines a set of traces, just like a grammar defines a set of sentences. With each set of traces an alphabet of symbols is associated. A trace is a finite-length sequence of symbols chosen from the alphabet. It may be interpreted as a possible sequence of communications. A component's set of traces then captures all possible communications with the environment.

Notation

(i) ε denotes the empty trace.

(ii) If V is a set of traces then V' denotes the associated alphabet.

(iii) If A is a set of symbols then A^* denotes the set of all traces consisting of symbols in A.

(iv) Whenever obvious from the context, the alphabets are omitted. Unless noted otherwise, small and capital letters near the beginning of the (Roman) alphabet stand for symbols and sets of symbols respectively, small and capital letters near the end of the alphabet stand for traces and sets of traces respectively.

(v) $V \div W$ denotes the symmetric set difference of the sets V and W.

Definition 2.1 We write $t{:}comp(t_0{:}A_0, t_1{:}A_1)$ to indicate that trace t can be composed from traces t_0 (with alphabet A_0) and t_1 (with alpnabet A_1). It is defined as follows.

$$
\begin{aligned}
&t{:}comp(t_0{:}A_0, t_1{:}A_1) = \\
&\quad t_0 = \varepsilon \wedge t_1 = \varepsilon \wedge t = \varepsilon \\
&\vee\ t_0 = au_0 \wedge t = au \wedge a \notin A_1 \wedge u{:}comp(u_0{:}A_0, t_1{:}A_1) \\
&\vee\ t_1 = au_1 \wedge t = au \wedge a \notin A_0 \wedge u{:}comp(u_1{:}A_1, t_0{:}A_0) \\
&\vee\ t_0 = au_0 \wedge t_1 = au_1 \wedge t{:}comp(u_0{:}A_0, u_1{:}A_1)
\end{aligned}
$$

The last alternative is called elimination of a. Notice that symbols in $A_0 \cap A_1$ do not occur in t. Obviously, composition is symmetric in t_0 and t_1. If A_0 and A_1 are disjoint $t{:}comp(t_0{:}A_0, t_1{:}A_1)$ expresses that t is an interleaving of t_0 and t_1.

Definition 2.2 The composition of two sets V and W of traces, notation $V + W$, is defined by

$$V + W = \{t: (\exists v \in V, w \in W: t{:}comp(v{:}V', w{:}W'))\}$$

The alphabet of $V + W$ is $V' \div W'$.

If V' and W' are disjoint then $V + W$ is the set of all interleavings of traces from V and W.

Example 2.2 Let $V = \{ab, cd\}$ and $W = \{be, df\}$. Then $V + W = \{ae, cf\}$.

To guide the reader's intuition we point out that the composition in Example 2.2 may be interpreted as follows. The environment of V chooses between the symbols a and c. V then communicates that choice to W. W answers by e or by f. Input of a into $V + W$ causes it to answer e, input of c gives output f. The fact that $V + W$ communicates the choice internally has been eliminated; $V + W$ has ae and cf as its only traces.

Example 2.3

$$(\{ab\} + \{ac\}) + \{ac\} = \{bc, cb\} + \{ac\} = \{ab, ba\}$$

$$\{ab\} + (\{ac\} + \{ac\}) = \{ab\} + \{\varepsilon\} = \{ab\}$$

Composition of sets of traces is symmetric. Example 2.3 shows that it is not associative. However, we have the following property.

Property 2.1 If each symbol occurs in at most two of the three alphabets V_0', V_1', and V_2' then

$$(V_0 + V_1) + V_2 = V_0 + (V_1 + V_2)$$

From now on we shall restrict ourselves to compositions in which each symbol occurs in at most two of the alphabets of the composing sets of traces.

We now introduce a program notation and the mechanism by which programs define sets of traces.

Notation If S is a program then $T(S)$ denotes the set of traces defined by S.

Definition 2.3

(i) A symbol a is a program. $T(a) = \{a\}$. Its alphabet contains symbol a only.

(ii) If S_0 and S_1 are programs then $S_0|S_1$ and $S_0;S_1$ are also programs. Both have $T(S_0)' \cup T(S_1)'$ as their alphabets.

$$T(S_0|S_1) = T(S_0) \cup T(S_1)$$
$$T(S_0;S_1) = \{t_0t_1: t_0 \in T(S_0) \wedge t_1 \in T(S_1)\}$$

(iii) If S_0 and S_1 are programs with disjoint alphabets then S_0,S_1 is also a program. It has $T(S_0)' \cup T(S_1)'$ as its alphabet.

$$T(S_0,S_1) = T(S_0) + T(S_1)$$

(So the comma denotes interleaving. For composition with elimination we shall introduce a separate notation, viz. that of a component.)

(iv) If S is a program then $\{S\}$ is also a program. It has $T(S)'$ as its alphabet.

$$T(\{S\}) = \{t_0 t_1 \ldots t_{n-1} : n \geq 0 \wedge (\forall i: 0 \leq i < n: t_i \in T(S))\}$$

Priority rules The comma has the highest priority, followed by the semicolon, and then the bar:

$$S_0 , S_1 \mid S_2 = (S_0 , S_1) \mid S_2$$

$$S_0 , S_1 ; S_2 = (S_0 , S_1) ; S_2$$

$$S_0 ; S_1 \mid S_2 = (S_0 ; S_1) \mid S_2$$

Properties 2.2

(i) $T(S_0|S_1) = T(S_1|S_0)$

(ii) $T(S_0,S_1) = T(S_1,S_0)$

(iii) $T((S_0|S_1)|S_2) = T(S_0|(S_1|S_2))$

(iv) $T((S_0;S_1);S_2) = T(S_0;(S_1;S_2))$

(v) $T((S_0,S_1),S_2) = T(S_0,(S_1,S_2))$

(vi) $T(S_0;(S_1|S_2)) = T(S_0;S_1|S_0;S_2)$

(vii) $T((S_0|S_1);S_2) = T(S_0;S_2|S_1;S_2)$

(viii) $T(S_0,(S_1|S_2)) = T(S_0,S_1|S_0,S_2)$

(ix) $T(\{\{S\}\}) = T(\{S\})$

(x) $T(\{S\};\{S\}) = T(\{S\})$

Notation

(i) A symbol is either simple or it is of the form $c.a$. Let V be a set of traces with an alphabet of simple symbols. Then $c.V$ denotes the set of traces obtained from V by replacing in each trace each symbol a by the symbol $c.a$. The alphabet of $c.V$ is obtained from V' by changing each symbol a in V' into the symbol $c.a$.

(ii) If t is a trace then t_0 is called a prefix of t when there exists a t_1 such that $t = t_0 t_1$.

(iii) If V is a set of traces then $V^{\sim}$ denotes the set of all prefixes of traces in V . It has the same alphabet as V .

We now define the last form a program can have: the component. Syntactically, a component C is of the form

com C("alphabet"): "subcomponents" "equalities" S moc

"alphabet" must be an alphabet of simple symbols.

"subcomponents" is a list of zero or more components C_i , each with a name c_i : $c_0:C_0$, $c_1:C_1$, ... , $c_{n-1}:C_{n-1}$.

"equalities" is a list of equalities of the form $c_i.a = c_j.b$ in which $a \in T(C_i)'$ and $b \in T(C_j)'$ with $i \neq j$.

S is a program as defined in Definition 2.3. Its alphabet must be a subset of the union of the alphabet of C and $\bigcup_{i=0}^{n-1} c_i.T(C_i)'$.
Every symbol of the form $c_i.a$ $(0 \leq i < n,\ a \in T(C_i)')$ must occur either in $T(S)'$ or in exactly one equality.

Definition 2.4 Let component C be defined as above. Then

$$T(C) = T(S)^{\sim} + c_0.T(C_0) + c_1.T(C_1) + \ldots + c_{n-1}.T(C_{n-1})$$

In this composition symbols $c_i.a$ and $c_j.b$ are considered to be the same symbol when C contains the equality $c_i.a = c_j.b$.

Each symbol $c_i.a$ occurs in exactly two of the sets $T(S)^{\sim}$, $c_0.T(C_0)$, $c_1.T(C_1)$, ... , $c_{n-1}.T(C_{n-1})$. It, therefore, does not occur in the traces in T(C) ; All traces in T(C) consist of symbols from the alphabet of C .
Because of Property 2.1 the composition above is associative.

Example 2.4

```
com id(x0, x1, y0, y1):  x0; y0 | x1; y1  moc
```

$T(id) = \{\varepsilon,\ x0,\ x0\ y0,\ x1,\ x1\ y1\}$

Example 2.5

```
com binsem(p, v):  {p; v}  moc
```

Let N(p) stand for "the number of occurrences of p ". Then

$$T(binsem) = \{t \in \{p,v\}^{*} :\ 0 \leq N(p) - N(v) \leq 1 \text{ in every prefix of } t\}$$

We say that T(binsem) is characterized by $0 \leq N(p) - N(v) \leq 1$.

Example 2.6

```
com quinsem(p, v):
    sub b0,b1: binsem
    b0.v = b1.p
    {p; b0.p} , {b1.v; v}
moc
```

(The second line of the component's text expresses that the subcomponents are b0:binsem, b1:binsem .)

$T(\{p;\ b0.p\})^{\sim}$	is characterized by		$0 \leq N(p) - N(b0.p) \leq 1$	
$T(\{b1.v;\ v\})^{\sim}$	"	"	$0 \leq N(b1.v) - N(v) \leq 1$	
b0.T(binsem)	"	"	$0 \leq N(b0.p) - N(b0.v) \leq 1$	
b1.T(binsem)	"	"	$0 \leq N(b0.v) - N(b1.v) \leq 1$	(b0.v=b1.p
			———————————————	+
T(quinsem)	"	"	$0 \leq N(p) - N(v) \leq 4$	

The above is an application of the so-called adding rule.

3. States and input/output as derived properties

Given a set V of traces we call two prefixes s and t of traces in V V-equivalent when

$$\{u: su \in V\} = \{u: tu \in V\}$$

This is an equivalence relation. The equivalence classes are called states. We denote the equivalence class (state) of which prefix s is a member by $[s]_V$. Whenever V is obvious from the context it is omitted.

Example 3.1 The component

```
com buf0(x0, x1, y0, y1): {x0; y0 | x1; y1} moc
```

has three states. The empty trace and all prefixes ending on y0 or y1 are equivalent. All prefixes ending on x0 are equivalent and all prefixes ending on x1 are equivalent. The three states are thus $[\varepsilon]$, $[x0]$, and $[x1]$.

Component binsem has two states and component quinsem five states.

We write $[x]_V a$ for $(\exists t: xat \in V)$. Again, we omit V whenever it is obvious from the context. We call two symbols a and b V-related when

$$\{x: [x]_V a\} = \{x: [x]_V b\}$$

This is again an equivalence relation. Each equivalence class of this relation that contains at least two symbols is called an input. The singleton equivalence classes are the outputs. (This is actually a simplified definition of input/output, but it suffices for the examples to be discussed below.)

In component buf0 symbols x0 and x1 are related and thus constitute one input. We may interpret them as the reception of a value 0 or a value 1, respectively. The component responds to an input x0 or x1 with an output y0 or y1, respectively.

Example 3.2

```
com binvar(x0, x1, y0, y1):
    {x0; {y0} | x1; {y1}}
moc
```

The pair (x0,x1) forms again one input, y0 and y1 are the outputs. Notice that we have constructed this component in such a way that it must be initialized before it can be inspected.

Example 3.3

```
com buf1(x0, x1, y0, y1):
    sub b0,b1: buf0
    b0.y0 = b1.x0, b0.y1 = b1.x1
    {x0; b0.x0 | x1; b0.x1} , {b1.y0; y0 | b1.y1; y1}
moc
```

The pair (x0,x1) is the input. Component buf1 may be interpreted as a four-bit buffer.

Example 3.4

```
com buf2(x0, x1, y0, y1):
    sub b0,b1: buf0
    {(x0; b0.x0 | x1; b0.x1); (x0; b1.x0 | x1; b1.x1)} ,
    {(b0.y0; y0 | b0.y1; y1); (b1.y0; y0 | b1.y1; y1)}
moc
```

The components buf1 and buf2 define the same set of traces. Up to now the authors have only been able to demonstrate this by an elaborate case analysis. Internally the two components are very different. Only in buf1 is there communication between the two subcomponents. In buf2 the two subcomponents are "used" alternately.

We now look in more detail at another example.

```
com fulladder(a0, a1, b0, b1, c0, c1, d0, d1, s0, s1):
    {a0,b0; d0 , (c0; s0 | c1; s1)
    |a1,b1; d1 , (c0; s0 | c1; s1)
    |(a0,b1 | a1,b0); (c0; d0,s1 | c1; d1,s0)}
moc
```

The symbols a0 and a1 are related, b0 and b1 are related, and c0 and c1 are related. The component, consequently, has three binary inputs: (a0,a1) , (b0,b1) , and (c0,c1) . It may be interpreted as a full-adder element: (a0,a1) and (b0,b1) represent the two bits to be added, and (c0,c1) represents the carry-in. The carry-out is represented by the outputs d0 and d1 ; s0 and s1 represent the sum.

For each output symbol we list the input symbols that precede it, i.e., those that are separated from it by at least one semicolon:

d0: $a0 \wedge b0 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0$

d1: $a1 \wedge b1 \vee a0 \wedge b1 \wedge c1 \vee a1 \wedge b0 \wedge c1$

s0: $a0 \wedge b0 \wedge c0 \vee a1 \wedge b1 \wedge c0 \vee a0 \wedge b1 \wedge c1 \vee a1 \wedge b0 \wedge c1$

s1: $a0 \wedge b0 \wedge c1 \vee a1 \wedge b1 \wedge c1 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0$

Seitz discusses a PLA-like full-adder element on p. 251 of [1]. The 14 terms in the four lines above give exactly the 14 crossings in his PLA at which the horizontal and vertical wires are connected. Seitz's realization could thus be derived directly from our program text.

4. Conclusions

We have defined the meaning (the semantics) of a component in terms of traces. Each trace is a sequence with the symbols of the component as elements. Also the communication with a component has been expressed in terms of the symbols of that component. Actually, the meaning of a component constitutes exactly all possible ways in which it can communicate with its environment. All internal communication has disappeared from it. This "information hiding" was taken care of by our elimination rule.

Our way of defining the semantics of programs may be contrasted with approaches based on states and state transistions. In the latter approaches the meaning of a program is defined as a function from states to states or from sets of states to sets of states. As we do not consider the notion of state to be fundamental in partially ordered computations, we believe a formalization in terms of all possible communications to be more appropriate.

We are currently working on three extensions of the material presented. First of all, we need more theorems on sets of traces: a trace theory. We must, for example, be able to find a nice proof that $T(buf1) = T(buf2)$. Secondly, we want to find a method of deriving transistor diagrams (so-called schematics) from the definitions of components. Communication along wires introduces delays. Making no assumptions about these delays amounts to the construction of self-timed systems. In the long term we want to make a silicon compiler for programs that express partially ordered computations. But that requires the solving of the nontrivial problem of mapping schematics on the two-dimensional medium of silicon. This may be simpler for self-timed systems than for synchronous systems. Thirdly, we want to introduce values and ports. We have the intention of introducing these as abbreviations. For each type there will be an instantiation scheme defining how values and ports of that type are to be represented in terms of symbols.

5. Reference

[1] Mead, Carver & Lynn Conway. "Introduction to VLSI Systems". Addison-Wesley, Reading, Mass., 1980.

Page 251 of [1]:

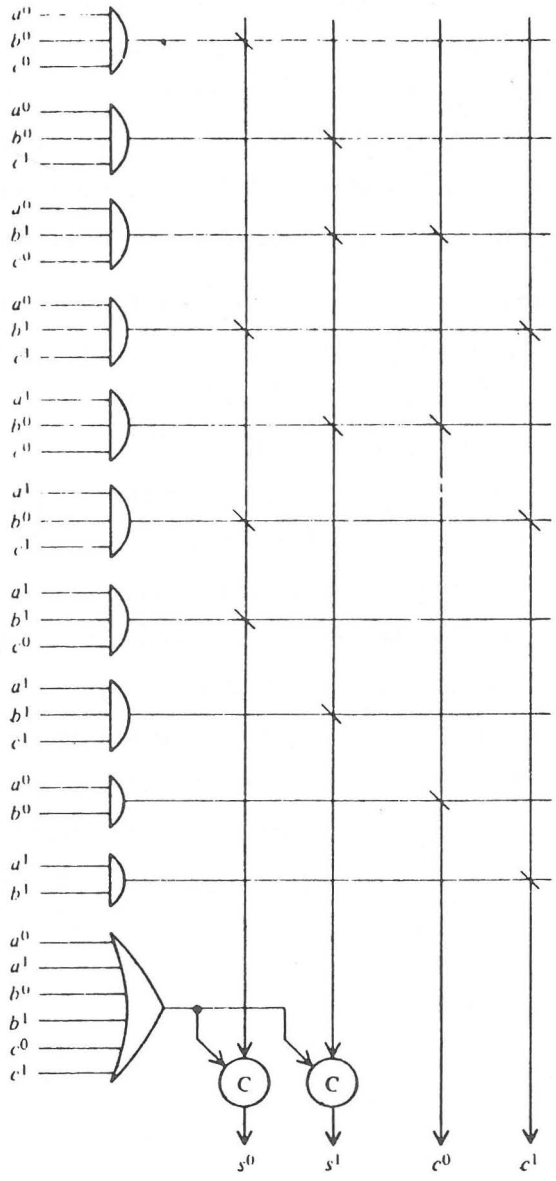


Fig. 7.15 Full-adder element.

Another thing to notice about this element is that the carry-out function c_{i+1} is generated as soon as the operands a_i and b_i become defined if they are in either the 00 (carry-kill) or 11 (carry-generate) conditions. The time required to perform an addition is generally limited by the worst-case carry propagation through the entire length of the adder. However, this case occurs only rarely. For operands

Discussion

During the lecture **Professor Wells** asked about the purpose of the notation which was being developed. **Professor Rem** replied that the purpose of the notation was to describe the communications of a component with its environment. He agreed with **Professor Katzenelson** who suggested that components were being characterised in terms of their input/output behaviours.

With regard to the full adder example **Professor Sequin** asked whether the solution showed full symmetry between the three inputs. **Professor Rem** replied that the inputs a and b were symmetrical, but not c; this was a deliberate decision made in order to enable generation of the carry-out signal as early as possible. **Professor Sequin** asked to be shown how a particular trace would be shown to satisfy the program text, or not, and **Professor Rem** did so for a randomly chosen trace.

Professor Katzenelson then asked whether the simultaneous arrival of inputs should not be excluded from the specification. **Professor Rem** explained that the order of arrival was immaterial, and that the concurrent behaviour was modelled by all possible interleavings of the input signals.