University of Edinburgh

Department of Computer Science

A Model of Register Transfer Systems with Applications to Microcode and VLSI correctness

by

Mike Gordon

Internal Report

CSR- 82-81

James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

March, 1981
Revised May, 1982

Corrected version of Department of Computer Science Internal Report CSR-82-81, University of Edinburgh.

A MODEL OF REGISTER TRANSFER SYSTEMS WITH APPLICATIONS TO MICROCODE AND VLSI CORRECTNESS

Mike Gordon

Computer Laboratory, University of Cambridge
Corn Exchange Street, Cambridge CB2 3QG, UK

Abstract

In this paper we describe and illustrate a simple semantic model of register transfer systems - i.e. systems built by connecting together storage devices like registers and memories via combinational circuits like gates and arithmetic units. The goal is to develop an elegant and efficient framework in which to conduct correctness proofs. After explaining the model we illustrate our methods by presenting two case studies. In the first of these we completely specify a small general purpose computer, and then prove correct a microcoded implementation. This involves showing that the signals generated by the microprogrammed controller cause register transfers in the host which correctly fetch, decode and execute machine instructions; and also that the control unit correctly interprets and sequences microinstructions according to the microcode semantics. In the second case study we go down a level and verify nMOS implementations of devices like those used to build the computer. Starting from four primitives - gates, joins, pullups and ground - we first implement and verify *not* and *nor* elements. Using these we then specify, implement and verify a stackcell and controller taken from Mead and Conway's book "Introduction to VLSI Systems". In both case studies the proofs are highly structured. For example, in the nMOS study the stack controller is expressed as the composition of two subsystems and a clock, and its correctness follows from the correctness of the subsystems; the correctness of these, in turn, follows from the correctness of their immediate constituents (*not* and *nor* elements). It is not necessary to flatten down to the gate level and hence proofs do not explode in size.

N.B. I would be very grateful for any comments and advice on the work reported here. Please write to me at the Computer Laboratory, Cambridge.

Introductory Overview

A typical register transfer system is shown in Fig.I

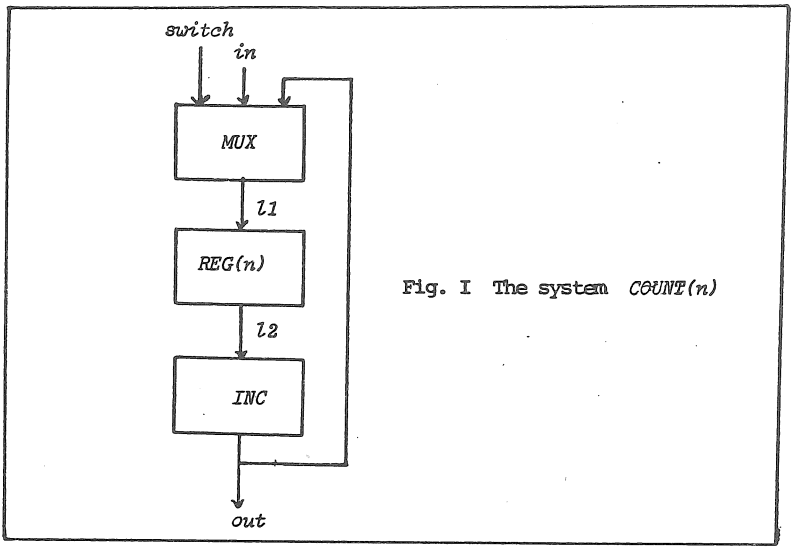


Fig. I The system *COUNT(n)*

The complete system *COUNT(n)* is composed out of three components *MUX*, *REG(n)* and *INC*. *MUX* and *INC* are combinational devices; the value on the output line is a function of the values on the input lines. We abstract away from the finite delay present in practice. The value on the output line *l1* of *MUX* is either the value on the input line *in*, if *true* is the value on the input line *switch*, or it is the value on the input line *out* if *false* is the value on *switch*. The value on the output line *out* of *INC* is one plus the value on its input line *l2*.

REG(n) is a sequential device. It behaves like a state machine which changes state when signalled to do so by the clock. In a state in which the device is storing the value *n* it outputs *n* on to the output line *l2*; when the system is clocked the value on the input line *l1* is stored, overwriting the previous value.

Suppose we put *true* on line *switch* and *0* on line *in*, then *0* will be on line *l1* and so if we clock the system *0* will be stored in the register. We

express this by saying *COUNT(n)* becomes *COUNT(0)*. If we now put *false* on line *switch* then the value on *l1* will be the value on *out* which is one plus the value on *l2* which is the value stored - i.e. we will have *0,1,1* on lines *l2,out* and *l1* respectively. Thus if we clock the system it becomes *COUNT(1)*. Clearly, successively clocking the system whilst keeping *false* on line *switch* will cause the system to become *COUNT(2),COUNT(3)* ...

In this paper we explain a notation for representing and verifying both specifications and implementations of register transfer systems. This notation is based on the λ-calculus and is interpreted within the theory of domains developed by Dana Scott [11]. However, to make what follows accessible to as wide an audience as possible, we do not assume knowledge of this theory. In the next section we explain everything that we need.

Technical Preliminaries

$\{x|...x...\}$ denote the set of all x such that $...x...$ is true; $\{\}$ denotes the empty set; $S \cup S'$, $S \cap S'$ and $S - S'$ denote respectively the union, intersection and difference of the sets S and S' ; $x \in S$ means x is a member of S.

If $f:S \to S'$ is a function and $x \in S$ then we write either fx or $f(x)$ for the result of applying f to x. If $f:S \to S'$ and $f':S' \to S''$ then $f' \circ f : S \to S''$ is the functional composition of f and f' defined by $(f' \circ f)(x) = f'(f(x))$.

If S and S' are sets then $S \times S'$ is the set of ordered pairs $\{(x,y)|x \in S \ , \ y \in S'\}$. The functions $fst:(S \times S') \to S$ and $snd:(S * S') \to S'$ are defined by:

$$fst(x,y) = x$$
$$snd(x,y) = y$$

The set of all functions from S to S' is denoted by $S \to S'$, thus $f:S \to S'$ is equivalent to $f \in (S \to S')$. We abbreviate $S_1 \to (S_2 \to ... \to (S_n \to S)...)$ by $S_1 \to S_2 \to ... \to S_n \to S$ - i.e. $\to$ associates to the right. If $f:S_1 \to S_2 \to ... \to S_n \to S$ and $x_i \in S_i$ (for $1 \le i \le n$) then $fx_1x_2...x_n$ abbreviates $((..((fx_1)x_2)....)x_n)$ - i.e. function application association to the left.

We denote the set of integers by *Int* and assume the usual operations over them. The set {*true*, *false*} of truth values, or booleans, is denoted by *Bool*. The unary operation $\neg$ and binary operations $\wedge$ and $\vee$ denote negation (not), conjunction (and) and disjunction (or) respectively. We shall sometimes use *1* instead of *true* and *0* instead of *false*; it should be clear from context when *0,1* denote integers and when they denote booleans.

If E is some expression which takes values in a set S' whenever x takes values in a set S.then $\lambda x.E$ denotes the function $f:S \to S'$ defined by $f(x)=E$. For example, $\lambda x.x^2+1$ denotes the function which maps a number to the successor of its square. Such function denoting expressions are called λ-expressions. In $\lambda x.E$ we call x the bound variable, and E the body. Note that renaming the bound variable of a λ-expression does not alter its meaning; for example, $\lambda x.x^2+1$ and $\lambda n.n^2+1$ both denote the same function.

We sometimes denote pairs (x,y) by just x,y - i.e. we omit the brackets. An expression of the form $\lambda x.E,E'$ means $\lambda x.(E,E')$ not $(\lambda x.E),E'$.

If E is an expression which takes values in S' whenever x_i takes a value in a set S_i (for $1 \le i \le n$), then $\lambda(x_1,\ldots,x_n).E$ denotes the function $f:(S_1 \times S_2 \times \ldots \times S_n) \to S'$ defined by $f(x_1,\ldots,x_n)= E$ and $\lambda x_1 x_2 \ldots x_n.E$ denotes the function $f':S_1 \to S_2 \to \ldots \to S_n \to S$ defined by $fx_1x_2\ldots x_n \equiv E$. For example, $\lambda(x,y)\ x+y$ denotes the addition function $add:(Int \times Int) \to Int$ defined by $add(x,y)=x+y$, and $\lambda xy.x+y$ denotes the 'curried' addition function $addc:Int \to (Int \to Int)$ defined by $addc(x) = \lambda y.x+y$ or equivalently $addc\ x\ y = x+y$. Note that the result of applying $addc$ to an integer is a function - for example, $addc\ 1$ is the successor function.

If $x_1,\ldots,x_n$ are distinct variables and $E,E_1,\ldots,E_n$ are expressions then

let $\{x_1 = E_1,\ldots,x_n = E_n\}$ in E

denotes the value of E when each x_i denotes the value of E_i. For example, "let $\{x=1,\ y=2\}$ in $x+y$" denotes *3*. The expression "let$\{x_1 = E_1,\ldots,x_n = E_n\}$ in E" is equivalent to (but often more readable than) $(\lambda(x_1,\ldots,x_n).E)(E_1,\ldots,E_n)$.

We shall often want to define functions recursively, for example.

factorial (n) = if $n=0$ then *1* else $n \times$ *factorial (n-1)*

such 'definitions' do not obviously make sense because the thing they purport to define (e.g. the factorial function) is assumed to exist by

the defining expression. We shall give meaning to these definitions via the theory of least fixed points: According to this theory, recursive definitions are equations (analogous to $x=\frac{1}{2}(x^2+1)$) which are taken to define their solutions (just as $x=\frac{1}{2}(x^2+1)$ defines x to be 1). Unfortunately solutions to recursive function equations may either not exist - for example, there is no $f:Int\to Int$ such that $f(x) = f(x)+1$ - or may exist, but not be unique; for example, every $f:Int\to Int$ satisfies $f(x)=f(x)$. The theory of least fixed points solves these problems by imposing an ordering on functions in such a way that every 'well formed' recursive definition has a least solution. To define this ordering it is required that the ranges of functions be domains - i.e. ordered sets containing a least member $\perp$ and for which every ascending chain has a least upper bound. We shall not assume the reader is familiar with the theory of domains and least fixed points. All that is needed to understand what follows is a willingness to accept recursive definitions as unproblematical. Such definitions can be thought of intuitively as defining the function that would be computed if they were expressed in a programming language such as LISP. The interpretation of the undefined element $\perp$ is as the 'result' returned by a function when applied to an argument on which it does not terminate. For example if we define $f(x) = f(x)+1$ then this defines $f(x)=\perp$ for all x. For this to work we must make Int into a domain by adding $\perp$ to it.

As well as recursively defined functions we will also need recursively defined domains. More specifically if D and D' are given domains we will use the domain B defined by $B = D\to(D'\times B)$ - much of the next section is concerned with motivating this. Technically such domain equations are solved using a beautiful theory developed by Dana Scott [11] - a theory that we do not assume the reader is familiar with. Intuitively domains can be thought of as data types, and then recursive domain equations can be thought of as recursive data type definitions. As a first glimpse of how we shall use the recursively defined domain B to model register transfer systems, note that if $f\in B$, then as $B = D\to(D'\times B)$, we have $f:D\to(D'\times B)$ and hence $(fst\circ f):D\to D'$ and $(snd\circ f):D\to B$. If we think of B as the set of states of a sequential machine with input alphabet D and output alphabet D' then $(fst\circ f)$ will turn out to be the output function and $(snd\circ f)$ the next-state function. This should become clear later, but first we must continue with the technical preliminaries.

Because of our policy of trying to make this paper accessible to readers not familiar with the theory of domains, we shall omit all proofs which require appeal to this theory. In fact there is absolutely nothing of intrinsic mathematical interest in what follows, and all the omitted proofs are routine applications of standard techniques, although some of them are rather tedious. We shall also sometimes avoid discussing technicalities which arise from the use of domains rather than sets. For instance we will gloss over continuity questions, and the handling of $\perp$ in our examples. Readers familiar with the necessary theory should easily be able to supply the missing details.

We assume that both *Int* and *Bool* are flat domains (and hence contain $\perp$), and that the standard operations ($=, +, -, \times, <, >, \neg, \wedge, \vee$ etc.) are extended to continuous counterparts.

If D is any domain then the conditional function $cond_D : Bool \to (D \times D) \to D$ is defined by:

$$cond_D\ t\ (x,y) = \begin{cases} x & \text{if } t = true \\ y & \text{if } t = false \\ \perp & \text{if } t = \perp \end{cases}$$

Because we use conditionals so much we introduce the special notation $(t \to x, y)$ (due to McCarthy) for $cond_D\ t\ (x,y)$. We also write:

$$(t_1 \to x_1, t_2 \to x_2, \ldots, t_n \to x_n)$$

to mean $(t_1 \to x_1, (t_2 \to x_2, (\ldots, (t_n \to x_n, \perp)\ldots)))$. The value of this expression is x_i if $t_i = true$ and for all $j < i, t_j = false$, and is $\perp$ otherwise.

If $x_1, \ldots, x_n$ are distinct variables and $E, E_1, \ldots, E_n$ are expressions then:

letrec $\{x_1 = E_1, \ldots, x_n = E_n\}$ in E

denotes the value of E when each $x_1, \ldots, x_n$ is defined mutually recursively by the equations $x_1 = E_1, \ldots, x_n = E_n$. The difference between this and

let $\{x_1 = E_1, \ldots, x_n = E_n\}$ in E

is that in the former any occurrence of x_i in an E_j (where j may, or may not, equal i) is interpreted recursively as referring to E_i, whereas in the latter such x_i's are assumed to be defined in the enclosing context, and are interpreted with respect to this. For example:

```
let {y=1}
in letrec {x=y+1, y=2}
in (x, y)
```

denotes the pair $(3,2)$, whereas:

```
let {y=1}
in let {x=y+1, y=2}
in (x,y)
```

denotes the pair $(2,2)$. As another example:

```
let {f=λx.0}
in letrec {f=λx.(x=0→1, x × f(x−1))}
in f(6)
```

denotes $6!=720$, whereas:

```
let {f=λx.0}
in let {f=λx.(x=0→1, x × f(x−1))}
in f(6)
```

denotes $6\times((\lambda x.0)(5))=6\times 0=0$. In the former case the occurrence of f in the right hand side of the equation is interpreted recursively. In the latter case it is interpreted in the enclosing context where it is defined to be $(\lambda x.0)$.

In expressions:

let $\{x_1=E_1,\ldots,x_n=E_n\}$ in E

and

letrec $\{x_1=E_1,\ldots,x_n=E_n\}$ in E

if any of the E_i have the form $\lambda x.E_i'$ then we may write $x_i(x)=E_i'$ instead of $x_i=\lambda x.E_i'$ For example:

```
let {f(x)=0}
in letrec {f(x)=(x=0→1, x × f(x−1))}
in f(6)
```

This concludes the necessary preliminaries; we can now proceed to our model.

The Model

The semantics of a combinational device will be represented by a function from signals to signals, where a signal is just a function from lines to values. Assume we are given a set *Line* of lines, and a domain *Val* of values; variables ranging over *Line* will be written in small italics (e.g. *in*, *switch*, *l2*). For simplicity we shall assume all lines carry values drawn from a single domain - namely *Val*. More generally lines could be typed and only carry values of their associated types. For example, in the system of Fig. 1 the line *switch* carries values from *Bool*, whilst all

other lines carry values from *Int*. We shall assume *Val* contains all the values we need (e.g. integers and truthvalues). The theory that follows routinely extends to encompass differently typed lines.

Definition 1

If X is a non-empty subset of *Line* then the domain $Sig[X]$ is the set $\{f \mid f: X \to Val\}$ with the pointwise ordering. $Sig[\{\}]$ is the one element domain, and $Sig = Sig[Line]$. Members of $Sig[X]$ are called signals. We use the notation $\{x_1 = E_1, \ldots, x_n = E_n\}$ to denote the signal $s \in Sig[\{x_1, \ldots, x_n\}]$ defined by $s(x_i) = E_i$ for $1 \leq i \leq n$.

We also use the notation $\lambda\{x_1 = v_1, \ldots, x_n = v_n\}.E$, where E is an expression taking values in D, to denote the function $f: Sig[\{x_1, \ldots, x_n\}] \to D$ defined by:

$$f(s) = \text{let } \{v_1 = s(x_1), \ldots, v_n = s(x_n)\} \text{ in } E$$

A very convenient abbreviation of this notation is to write $\lambda\{x_1, \ldots, x_n\}.E$ for $\lambda\{x_1 = x_1, \ldots, x_n = x_n\}.E$. In this case we are using the same name for a line and for the variable associated with it to which the incoming value is bound.

Example 1

The combinational behaviour of the device *INC* of Fig. 1 is:

$$\lambda\{l2 = v\}.\{out = v + 1\}$$

or, equivalently:

$$\lambda\{l2\}.\{out = l2 + 1\}$$

The combinational behaviour of *MUX* of Fig. 1 is:

$$\lambda\{switch, in, out\}.\{l1 = (switch \to in, out)\}$$

In general the combinational behaviour of a device whose set of input lines is X and set of output lines is Y is a function from $Sig[X]$ to $Sig[Y]$.

Definition 2

The domain $Com[X;Y]$ of combinational behaviours from X to Y is defined by:

$$Com[X;Y] = Sig[X] \to Sig\ [Y]$$

The semantics of a sequential device with set of input lines X and set of output lines Y is a member of $Seq[X;Y]$ where:

Definition 3

The domain $Seq[X;Y]$ of sequential behaviours from X to Y is defined to be the least solution of the domain equation:

$$Seq[X;Y] = (Sig[X] \to (Sig[Y] \times Seq[X;Y]))$$

The idea of sequential behaviours is that a device with semantics $f \in Seq[X;Y]$ acts like a combinational device with behaviour $(fst \circ f) \in Com[X;Y]$ until it is clocked, whereupon it changes behaviour to $(snd\,(f\,s)) \in Seq[X;Y]$, where $s \in Sig[X]$ is the input signal when the clocking occurs.

To further motivate sequential behaviours consider the sequential machine:

$$M = (S_M, out_M, next_M)$$

where S_M is the set of states

and $out_M : Sig[X] \times S_M \to Sig[Y]$ is the output function

and $next_M : Sig[X] \times S_M \to S_M$ is the next-state function.

In a state $x \in S_M$ this machine acts like a combinational device with behaviour $\lambda s.out_M(s,x)$. On being clocked with input signal s it moves to state $next_M(s,x)$ The sequential behaviour corresponding to machine M is given by the function $\mathcal{B}[M] : S_M \to Seq[X;Y]$ defined recursively by:

$$\mathcal{B}[M]x = \lambda s.(out_M(s,x), \mathcal{B}[M](next_M(s,x)))$$

The behaviour of M in state x is $\mathcal{B}[M]x$. This relationship between sequential machines and members of the recursively defined domain $Seq[X;Y]$ derives from Milner's early theory of processes [7]. For further details of the connection see [5].

Example 2

The register $REG(n)$ of Fig.I is modelled as a member of $Seq[\{l1\};\{l2\}]$ by the definition:

$$REG(n) = \lambda\{l1 = v\}.(\{l2 = n\}, REG(v))$$

which is equivalent to the more concise:

$$REG(n) = \lambda\{l1\}.\{l2 = n\}, REG(l1)$$

This corresponds to the machine $(Int, out, next)$ where:

$$out(\{l1 = n\}, n') = \{l2 = n'\}$$

$$next(\{l1 = n\}, n') = n$$

At this point the reader might wonder why we use sequential behaviours at all - why not just work with machines? There are two main reasons: (1), when writing specifications it is often natural just to give a desired behaviour, having to give a machine realising this behaviour would lead to overspecification, and (2), if we worked with machines we would have to define some notion of simulation to express what it means for an implementing machine to correctly meet a specification. Such a notion of simulation would be essentially equivalent to equality of behaviour, and we feel that working directly with behaviours (rather than equivalence classes of machines under simulation) leads to a cleaner, more algebraic theory. In fact, at various times during the development of our model, we have tried to eliminate behaviours in favour of machines, in order to avoid having to use the recursive domain equation which defines $Seq[X;Y]$. We have never succeeded; considerations of elegance and manipulative simplicity have always driven us back to the more abstract notion of behaviour. We hope the two case studies at the end of this paper will justify these remarks.

From the informal description we gave of $COUNT(n)$ of Fig.I we see that its behaviour is given by:

$$COUNT(n) = \lambda\{switch, in\}.\{out = n + 1\}, COUNT(switch \rightarrow in,\ n + 1\}$$

Fig. II. Behaviour of $COUNT(n)$

We now show how to write down an expression representing the combination of the components MUX, $REG(n)$ and INC corresponding to Fig. I. We can then prove that this expression denotes the behaviour $COUNT(n)$ defined directly above. This will illustrate (albeit on a completely trivial example) the way we intend to verify register transfer systems: we write down an expression corresponding to the structure of the system being verified, and then show that this has the desired behaviour.

For much of what follows it is convenient to regard combinational devices as degenerate sequential devices - namely sequential devices which never change state.

Definition

If $f \in Com[X;Y]$ define $seq(f) \in Seq[X;Y]$ recursively by:

$$seq(f) = \lambda s.(f(s), seq(f))$$

Thus the combinational component of the sequential behaviour $seq(f)$ is always the same - namely f.

Example 3

The combinational behaviour of the device *INC* of Fig.I is $\lambda\{l2\}.\{out = l2 + 1\}$. The sequential behaviour of it is $seq(\lambda\{l2\}.\{out = l2 + 1\})$ which is equal to $INC \in Seq[\{l2\};\{out\}]$ defined recursively by:

$$INC = \lambda\{l2\}.\{out = l2 + 1\}, INC$$

We can now define all three components of the device shown in Fig.I as sequential behaviours:

$$MUX = \lambda\{switch, in, out\}.\{l1 = (switch \rightarrow in, out)\}, MUX$$

$$REG(n) = \lambda\{l1\}, \{l2 = n\}, REG(l1)$$

$$INC = \lambda\{l2\}.\{out = l2 + 1\}, INC$$

Fig.III. Behaviour of the components of *COUNT(n)*

From these definitions we see that:

$$MUX \in Seq[\{switch, in, out\};\{l1\}]$$

$$REG(n) \in Seq[\{l1\};\{l2\}]$$

$$INC \in Seq[\{l2\};\{out\}]$$

In order to join these three components together to get the device portrayed in Fig.I we must do two things:

(i) connect together output lines to input lines with the same name,

(ii) hide the internal lines (*out* is an output line of the composite device, but neither *l1* nor *l2* are).

To model (i) we will define for behaviours $f_1,\ldots,f_n$ a composition $[\![f_1|\ldots|f_n]\!]$ which is the behaviour obtained by joining all output lines of $f_1,\ldots,f_n$ to input lines with the same name. For example if f and f' are the behaviours of the devices *DEV* and *DEV'* shown below

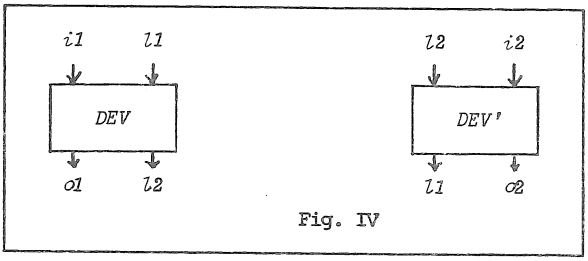
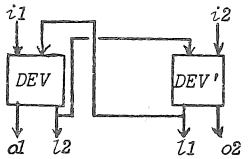


Fig. IV

Then $[\![f|f']\!]$ is the behaviour of



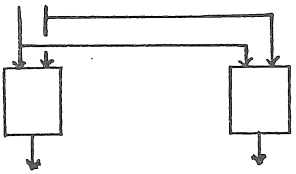
The output lines of $[\![f|f']\!]$ are just the output lines of f and f', whilst the input lines are those input lines which are not output lines as well.

If $f_i \in Seq[X_i;Y_i]$ then $[\![f_1|\ldots|f_n]\!]$ is only defined when $Y_1,\ldots,Y_n$ are pairwise disjoint. This is because the semantics of joining two lines depends on what one is modelling. For example, if lines represent wires carrying truthvalues then a join could either be disjunction or conjunction depending on whether one was modelling positive or negative logic. Thus instead of allowing:

we force one to explicitly represent the join with a device ; for example, by:

we thus avoid building a fixed meaning of joining into the model.

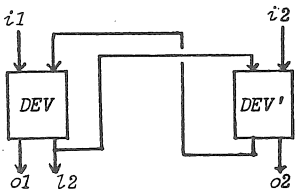
In our case study on nMOS algorithms we will use a join which is an extended disjunction - extended because it has to cope with a special value representing 'floating'. In contrast to the situation with output lines we do allow devices to share input lines. For example:



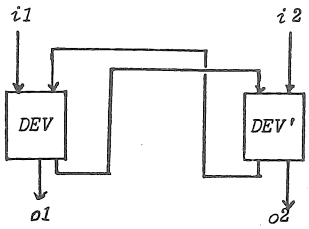
A value on a line which forks is just duplicated on to the two outgoing lines.

In summary if $f_i \in Seq[X_i;Y_i]$ and $Y_1,\ldots,Y_n$ are pairwise disjoint then we shall define $[\![f_1|\ldots|f_n]\!] \in Seq[\cup_i X_i - \cup_i Y_i; \cup Y_i]$

To model the hiding of internal lines ((ii) above), we shall define for each $l \in Line$ a restriction operator $\backslash l : Seq[X;Y] \to Seq[X;Y-\{l\}]$. Thus if f and f' represent the behaviour of the devices in Fig. IV, then $[\![f|f']\!]\backslash l1$ is the behaviour of:



and $[\![f|f']\!] \backslash l1 \backslash l2$ is the behaviour of:



Thus we have:

$[\![f|f']\!] \in Seq[\{i1, i2\}; \{o1, o2, l1, l2\}]$

$[\![f|f']\!] \backslash l1 \in Seq[\{i1, i2\}; \{o1, o2, l2\}]$

$[\![f|f']\!] \backslash l1 \backslash l2 \in Seq[\{i1, i2\}; \{o1, o2\}]$

Before defining the exact meaning of composition and restriction in our model we need some definitions.

Definition 4

If $s \in Sig[X]$ and $s' \in Sig[X']$ then $s[s'] \in Sig[X \cup X']$ is defined by:

$$s[s'](x) = \begin{cases} s'x & \text{if } x \in X' \\ sx & \text{otherwise} \end{cases}$$

and if $X \cap X' = \{\}$ we define $s.s' \in Sig[X \cup X']$ by:

$$s.s'(x) = \begin{cases} s'x & \text{if } x \in X' \\ sx & \text{if } x \in X \end{cases}$$

Intuitively $s[s']$ is the signal obtained by 'updating' s with the values specified by s', and $s.s'$ is the 'concatenation' of s and s'. It is convenient to have separate notations for these two conceptually distinct operations, although mathematically they are rather similar.

Definition 5

If $s \in Sig[X]$ and $X' \subseteq Line$ then $s \restriction X' \in Sig[X \cap X']$ is the restriction of s to X'.

Definition 6

If $f_i \in Com[X_i; Y_i]$ $(1 \leq i \leq n)$ and $Y_1, \ldots, Y_n$ are pairwise disjoint then

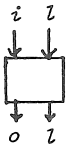
define $f_1|\ldots|f_n \in Com[\cup_i Xi\ ;\ \cup Y_i]$ by:

$$(f_1|\ldots|f_n)s = (f_1(s\restriction X_1))\ldots\ldots(f_n(s\restriction X_n))$$

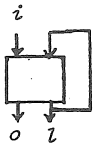
The expression $f_1|\ldots|f_n$ corresponds to putting $f_1,\ldots,f_n$ side by side but not joining any lines. For example, if f and f' are the behaviours of the devices in Fig. IV then $f|f' \in Seq[\{i1,l1,l2,i2\};\{o1,l2,l1,o2\}]$ is the behaviour of the device obtained by regarding the two devices in Fig. IV as one.

If f is a combinational behaviour then Definition 7 below defines $[\![f]\!]$ to be the behaviour obtained by joining input lines of f to output lines with the same name.

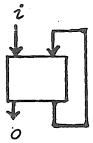
For example, if f is the behaviour of:



then $[\![f]\!]$ would be the behaviour of:



and so $[\![f]\!] \setminus l$ would be the behaviour of



Definition 7

If $f \in Com[X;Y]$ define $[\![f]\!] \in Com[X-Y;Y]$ recursively by:

$$[\![f]\!]s = f(s[\,[\![f]\!]s]\restriction X)$$

Combining Definition 6 and Definition 7 we see that for combinational behaviours we have defined $[\![f_1|\ldots|f_n]\!]$. We shall extend the definitions to the sequential case shortly, but first we state a lemma which helps to show that it works. In order to state this, and succeeding lemmas, we will abbreviate signal expressions of the form $\{x_1=E_1,\ldots,x_n=E_n\}$ by $\{x=E_x|x \in X\}$ where it is understood that $X=\{x_1,\ldots,x_n\}$ and $E_{x_i}=E_i$. This way of denoting signal expressions enables us to succinctly describe various expression manipulation laws. For example, we can express the fact that if $s=\{x_1=E_1,\ldots,x_m=E_m\}$ and $s'=\{x_1'=E_1',\ldots,x_n'=E_n'\}$ then $s.s'=\{x_1=E_1,\ldots,x_m=E_m,x_1'=E_1',\ldots,x_n'=E_n'\}$ by the law:

$$\{x=E_x|x \in X\}.\{x=E_x|x \in X'\}=\{x=E_x|x \in X \cup X'\} \quad \text{if} \quad X \cap X'=\{\}$$

It is such laws which give our model manipulative fluency and enable us to perform proofs by simple calculations.

We will also abbreviate expressions of the form $\lambda\{x_1,\ldots,x_n\}.E$ by $\lambda\{x|x \in X\}.E$ where $X=\{x_1,\ldots,x_n\}$. These abbreviations are, I hope, harder to explain than to understand!

Lemma 1: The Combinational Composition Lemma

$$[\![\lambda\{x|x \in X_1\}.\{y=E_y|y \in Y_1\}|\ldots|\lambda\{x|x \in X_n\}.\{y=E_y|y \in Y_n\}]\!]$$

$$= \lambda\{x|x \in \bigcup_i X_i - \bigcup_i Y_i\}.$$

$$\text{letrec } \{y=E_y|y \in \bigcup_i X_i \cap \bigcup_i Y_i\}$$

$$\text{in} \quad \{y=E_y|y \in \bigcup_i Y_i\}$$

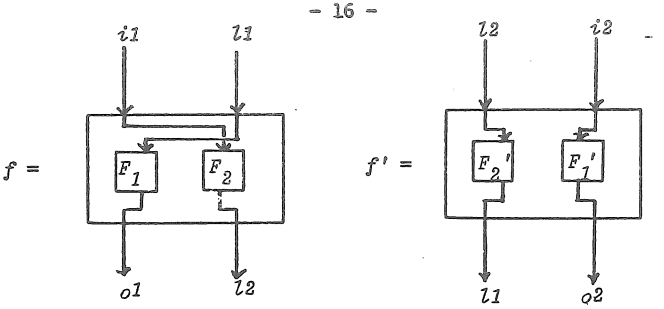
Example 4

Suppose the behaviour of the devices shown in Fig. IV are:

$$f=\lambda\{i1,l1\}.\{o1=F_1(l1),\ l2=F_2(i1)\}$$

$$f'=\lambda\{i2,l2\}.\{o2=F_1'(i2),\ l1=F_2'(l2)\}$$

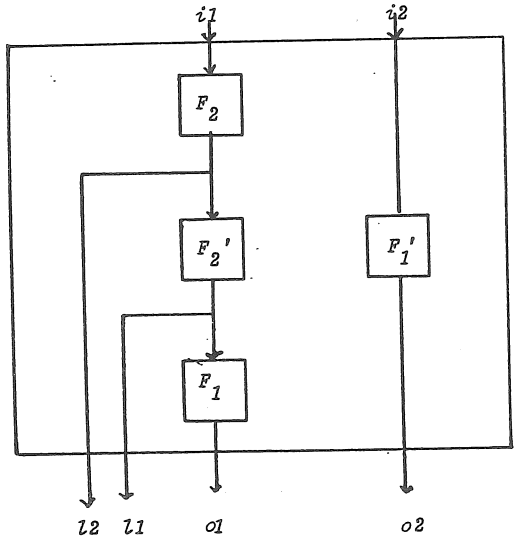
where $F_1,F_2,F_1',F_2': Val \to Val$. We might diagram this by drawing the two devices as:



Then by the Combinational Composition Lemma:

$$[\![f|f']\!] = \lambda\{i1,i2\}.$$
$$\text{letrec } \{l1=F_2'(l2),\ l2=F_2(i1)\}$$
$$\text{in } \{o1=F_1(l1),\ l2=F_2(i2),\ l1=F_2'(l2),\ o2=F_1'(i2)\}$$
$$= \lambda\{i1,i2\}.\{o1=F_1(F_2'(F_2 i2)),\ l2=F_2(i2),\ l1=F_2'(F_2 i2),\ o2=F_1'(i2)\}$$

Thus $[\![f|f']\!]$ is the behaviour of:



Definition 8

If $f \in Com[X;Y]$ and $l \in Line$ define $f\backslash l \in Com[X;Y-\{l\}]$ by:

$$(f\backslash l)s=(fs)\restriction Y-\{l\}$$

If $L=\{l1,\ldots,ln\}$ we write $f\backslash L=f\backslash l1\ l2\ldots ln=f\backslash l1\backslash l2\ldots\backslash ln$.

Lemma 2 below shows, among other things, that the order in which one combines devices, and the way one chooses to group them, does not effect the behaviour.

Lemma 2

If $f \in Com[X;Y]$, $f_i \in Com[X_i;Y_i](1 \le i \le n)$ and $Y_1,\ldots,Y_n$ are pairwise disjoint then:

1. If $X \cap Y=\{\}$ then $f=[\![f]\!]$
2. If $l \notin X \cap Y$ then $[\![f\backslash l]\!] = [\![f]\!]\backslash l$
3. $f|f_1|\ldots|f_n=f|(f_1|\ldots|f_n)$
4. $f_1|f_2=f_2|f_1$
5. $f_1|(f_2|f_3)=(f_1|f_2)|f_3$

The combinational composition theorem below shows the combined effect of composition and restriction.

Theorem 1: The Combinational Composition Theorem

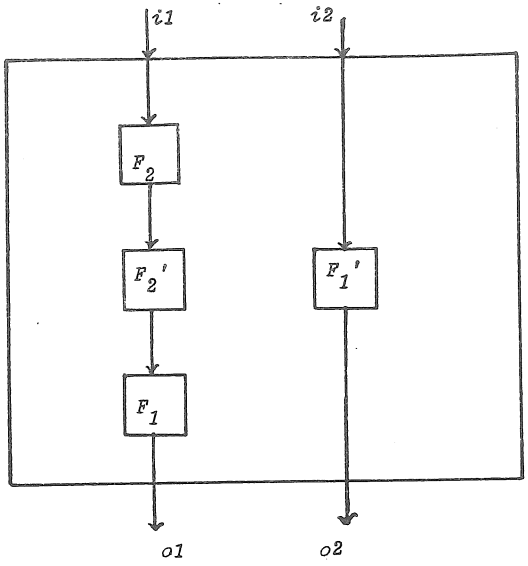
$$[\![\lambda\{x|x \in X_1\}.\{y=E_y|y \in Y_1\}|\ldots|\lambda\{x|x \in X_n\}.\{y=E_y|y \in Y_n\}]\!]\backslash L$$
$$=\lambda\{x|x \in \bigcup_i X_i - \bigcup_i Y_i\}.$$
$$\text{letrec } \{y=E_y|y \in \bigcup_i X_i \cap \bigcup_i Y_i\}$$
$$\text{in } \{y=E_y|y \in \bigcup_i Y_i - L\}$$

Example 5

If f and f' are as in Example 4 then:

$$[\![f|f']\!]\backslash l1\ l2$$
$$=\lambda\{i1,i2\}.$$
$$\text{letrec } \{l1=F_2'(l2),\ l2=F_2(i2)\}$$
$$\text{in } \{o1=F_1(l2),\ o2=f_1'(i2)\}$$
$$=\lambda\{i1,i2\}.\{o1=F_1(F_2'(F_2 i1)),\ o2=F_1'(i2)\}$$

which is the behaviour of:



We now extend the definitions of composition and restriction to sequential behaviours.

Definition 9

If $f \in Seq\,[X;Y]$, $f_i \in Seq[X_i;Y_i]$, $l \in Line$ and $Y_1,\ldots,Y_n$ are pairwise disjoint, then define:

$$[\![f]\!] \in Seq[X-Y;Y]$$

$$f\backslash l \in Seq[X;Y-\{l\}]$$

$$f_1|\ldots|f_n \in Seq[\,\cup_i X_i;\cup_i Y_i]$$

by:

$$[\![f]\!] = \lambda s.\ \text{let } \{s' = [\![fst \circ f]\!]\, s\} \text{ in } (s',\ [\![snd(f(s[s']\restriction X))]\!])$$

$$f\backslash l = \lambda s.\ (fst(fs)\restriction Y-\{l\},\ (snd(f\ s))\backslash l)$$

$$f_1|\ldots|f_n = \lambda s.\ \text{let}\{s_1 = s\restriction X_1,\ldots,s_n = s\restriction X_n\}$$
$$\text{in}(fst(f_1 s_1),\ldots,fst(f_n s_n),\ snd(f_1 s_1)|\ldots|snd(f_n s_n))$$

where the occurrence of $[\![fst \circ f]\!]$ in the definition of $[\![f]\!]$ is as defined in Definition 7. Lemma 2 can now be extended to sequential behaviours.

Lemma 3

If $f \in Seq[X;Y]$, $f_i \in Seq[X_i;Y_i] (1 \leq i \leq n)$ and $Y_1, \ldots, Y_n$ are pairwise disjoint then:

1. If $X \cap Y = \{\}$ then $f = [\![f]\!]$
2. If $l \notin X \cap Y$ then $[\![f \backslash l]\!] = [\![f]\!] \backslash l$
3. $f|f_1|\ldots|f_n = f|(f_1|\ldots|f_n)$
4. $f_1|f_2 = f_2|f_1$
5. $f_1|(f_2|f_3) = (f_1|f_2)|f_3$

Theorem 2: The Composition Theorem

$$[\![\lambda\{x|x \in X_1\}.(\{y=E_y|y \in Y_1\}, E_1) | \ldots | \lambda\{x|x \in X_n\}.(\{y=E_y|y \in Y_n\}, E_n)]\!] \backslash L$$
$$= \lambda\{x|x \in \cup_i X_i - \cup_i Y_i\}.$$

letrec $\{y=E_y|y \in \cup_i X_i \cap \cup_i Y_i\}$

in $(\{y=E_y|y \in \cup_i Y_i - L\}, [\![E_1|\ldots|E_n]\!] \backslash L)$

Example 6

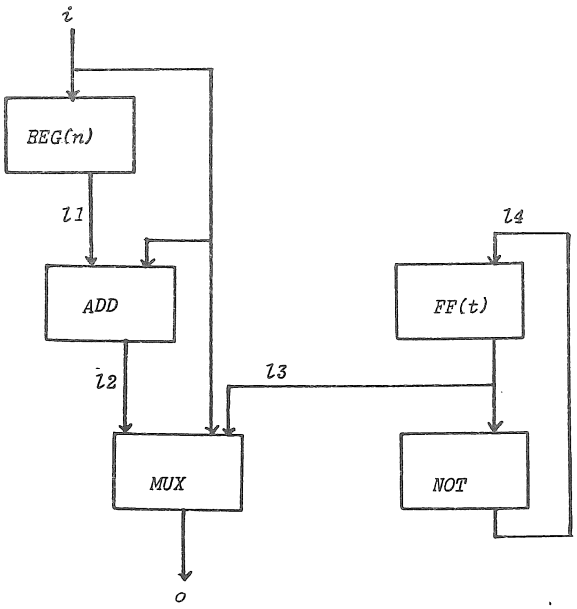
Using the Composition Theorem we can derive the behaviour of the device portrayed in Fig. I with components as defined in Fig. III. Putting the components together as in Fig. I corresponds to defining:

$$\begin{aligned} COUNT(n) &= [\![MUX|REG(n)|INC]\!] \backslash l1\,l2 \\ &= [\![\lambda\{switch, in, out\}.(\{l1=(switch \rightarrow in, out)\}, MUX) | \\ &\quad \lambda\{l1\}.(\{l2=n\}, REG(l1)) | \\ &\quad \lambda\{l2\}.(\{out=l2+1\}, INC)]\!] \backslash l1\,l2 \\ &= \lambda\{switch, in\}. \\ &\quad \text{letrec}\{l1=(switch \rightarrow in, out),\ l2=n, out=l2+1\} \\ &\quad \text{in} \quad (\{out=l2+1\}, [\![MUX|REG(l_1)|INC]\!] \backslash l1\,l2) \\ &\quad \text{(by Composition Theorem)} \\ &= \lambda\{switch, in\}.\{out=n+1\}, COUNT(switch \rightarrow in, n+1) \end{aligned}$$

which is precisely the behaviour specified in Fig. II.

Example 7

Consider the system $DEV(n,t)$ shown below:



with components defined by:

$$REG(n) = \lambda\{i\}.\{l1=n\},\ REG(i)$$
$$ADD = \lambda\{l1,i\}.\{l2=l1+i\},\ ADD$$
$$MUX = \lambda\{l2,i,l3\}.\{o=(l3 \rightarrow i, l2)\}.\ MUX$$
$$NOT = \lambda\{l3\}.\{l4=\neg l3\},\ NOT$$
$$FF(t) = \lambda\{l4\}.\{l3=t\},\ FF(l4)$$

The whole system has two state variables n and t and is defined by:

$$DEV(n,t) = [\![REG(n) | ADD | MUX | NOT | FF(t)]\!] \setminus l1\ l2\ l3\ l4$$

Notice that the devices *MUX* and *REG(n)* of this example are different from those of Fig. III since they have differently named lines. A better notation might be something like $MUX[switch,in,out;l1]$, $REG[l1;l2](n)$ for the devices of Fig. III, and $MUX[l3,i,l2;o]$, $REG[i;l1](n)$ for the devices of this example. In fact the correct management of line names, in particular the way behaviour should be parameterised on them, is an important problem which needs care if one is being more formal (e.g. see [6]). However, at the informal level of this paper a fairly casual approach is sufficient.

By the Composition Theorem:

$$DEV(n,t) = \lambda\{i\}$$
$$\text{letrec}\{l1=n,\ l2=l1\rightarrow i,\ l3=t,\ l4=\neg l3\}$$
$$\text{in } \{o=(l3\rightarrow i, l2)\},\ DEV(i,l4)$$
$$= \lambda\{i\}.\{o=(t\rightarrow i, n\rightarrow i)\},\ DEV(i,\neg t)$$

Let us unfold $DEV(n,true)$ through two clock cycles:

$$DEV(n,true) = \lambda\{i\}.\{o=i\},\ DEV(i,false)$$

Now we must be careful in unfolding $DEV(i,false)$ because if we just substitute i for n then i gets captured by the $\lambda\{i\}$. We must thus revert to our less abbreviated notation, viz:

$$DEV(n,true) = \lambda\{i=x\}.\{o=x\},\ DEV(x,false)$$

Now we can expand $DEV(x,false)$ as:

$$DEV(x,false) = \lambda\{i=x'\}.\{o=x+x'\},\ DEV(x',true)$$

Hence:

$$DEV(n,true) = \lambda\{i=x\}.\{o=x\},\lambda\{i=x'\}.\{o=x+x'\}\ ,\ DEV(x',true)$$

From this we would expect intuitively that for all n. $DEV(n,true) = DEV1$ where $DEV1$ is defined by:

$$DEV1 = \lambda\{i=x\}.\{o=x\},\ \lambda\{i=x'\}.\{o=x+x'\},\ DEV1$$

To prove this we will need a property of sequential behaviours which derives from the fact that $Seq[X;Y]$ is the least solution of its defining domain equation.

This property is conveniently expressed as an 'induction rule' which says that to prove $F_1(x)=F_2(x)$, where $F_1(x)$, $F_2(x)$ are sequential behaviours, it is sufficient to prove for all x and all signals s that:

$$F_1(x)s = (s',\ F_1(x'))$$
$$F_2(x)s = (s',\ F_2(x'))$$

for some s' and x' (which may depend on s). More precisely:

Theorem 3: Simulation Induction

Let S be a set and $F_1,F_2:S\rightarrow Seq[X;Y]$ then $F_1=F_2$ if for all $x\in S$ and $s\in Sig[X]$ there exists $x'\in S$ such that:

(1) $fst(F_1(x)s) = fst(F_2(x)s)$, and

(2) $snd(F_1(x)s) = F_1(x')$ and $snd(F_2(x)s) = F_2(x')$

The reason for the name "Simulation Induction" is because if F_1 and F_2 are as above, then they give the behaviour of machines which simulate each

other in 'lockstep'. To see this observe that if $F:S \to Seq[X,Y]$ has the property that for all $x \in S$, $s \in Sig[X]$ there exists a member of $S, f(s,x)$ say, such that $snd(F\,x\,s) = F(f(s,x))$, then $F = \llbracket M \rrbracket$ where $M = (S, \lambda(x,s).fst(F\,x\,s), f)$.

Example 8

Let DEV and $DEV1$ be as in Example 7, i.e.

$$DEV(n,true) = \lambda\{i\}.\{o = i\}, DEV(i,false)$$
$$DEV(n,false) = \lambda\{i\}.\{o = n + i\}, DEV(i,true)$$

and

$$DEV1 = \lambda\{i = x\}.\{o = x\}, \lambda\{i = x'\}.\{o = x + x'\}, DEV1$$

We use simulation Induction to show that for all n

$$DEV(n,true) = DEV1$$

To do this let

$$DEV2(n) = \lambda\{i\}.\{o = n + i\}, DEV1$$

so that:

$$DEV1 = \lambda\{i\}.\{o = i\}, DEV2(i)$$

and then define:

$$F(n,t) = t \to DEV1,\ DEV2(n)$$

We then use Simulation Induction to show for all n,t that:

$$DEV(n,t) = F(n,t)$$

(1) $fst(DEV(n,t)s) = \{o = (t \to s\,i, n + (s\,i))\}$

$fst(F(n,t)s) = t \to fst(DEV1\,s), fst(DEV2(n)s)$

$= t \to \{o = s\,i\}, \{o = n + (s\,i)\}$

$= \{o = (t \to s\,i, n + (s\,i))\}$

(2) $snd(DEV(n,t)s) = DEV(s\,i, \neg t)$

$snd(F(n,t)s) = t \to snd(DEV1\,s), snd(DEC2(n)s)$

$= t \to DEV2(s\,i), DEV1$

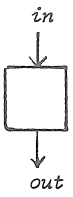
$= \neg t \to DEV1,\ DEV2(s\,i)$

$= F(s\,i, \neg t)$

Hence by Simulation Induction $DEV(n,t) = F(n,t)$, and so in particular

$$DEV(n,true) = F(n,true) = DEV1$$

Suppose we wish to compute some function $fun: Val \to Val$. To do this we could build a device:



with the property that if $x \in Val$ is put on the input line in, and then the device is clocked $time(x)$ times, where $time: Val \to Int$, then $fun(x)$ will appear on the output line out. We might also specify that whilst the device is computing $fun(x)$ it puts some 'safe' value $v \in Val$ on out (e.g. v might correspond to 'high impedance' as in three state busses). The behaviour of such a device would be $COMPUTE(fun, time, v)$ where:

Definition 10

If $fun: Val \to Val$, $time: Val \to Int$ and $v \in Val$ then define

$$COMPUTE(fun, time, v) \in Seq[\{in\}; \{out\}]$$

by:

$$COMPUTE(fun, time, v) = \lambda\{in\}.\{out = v\}, DELAY(COMPUTE(fun, time, v), v, fun(in), time(in))$$

where

$$DELAY(b, v, x, n) = (n = 0) \to (\lambda\{in\}.\{out = x\}, b), (\lambda\{in\}.\{out = v\}, DELAY(b, v, x, n-1))$$

intuitively $DELAY(b, v, x, n) \in Seq[\{in\}, \{out\}]$ is the behaviour corresponding to outputting v for n clock cycles, then outputting x and then becoming behaviour b.

Example 9

We shall do a top down design and verification of a device to compute $fact(n) = n!$ in n steps.

The specification of our device is that it has behaviour $FACT$ defined by:

$$FACT = COMPUTE(fact, (\lambda n.n), 0)$$

To implement this we shall use two devices $DOWN(n)$ and $MULT(m)$ for counting down on n and building up the results by multiplication respectively. The operation of these will be controlled by a device $TEST(t)$. Our implementation, obtained by connecting these devices together, is $FACTIMP(m, n, t)$ and is shown in Fig.V. below.

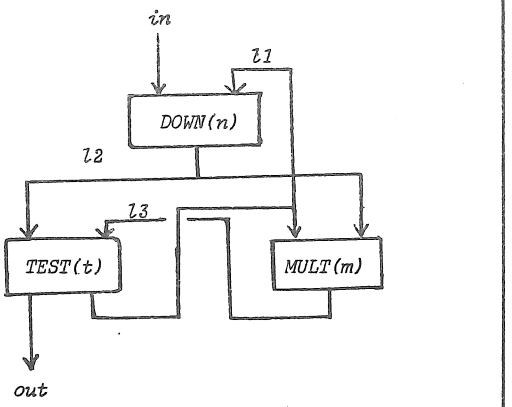


Fig.V. *FACT* implementation *FACT(m,n,t)*

FACTIMP(m,n,t) is defined by:

$$FACTIMP(m,n,t) = [\![MULT(m) \mid DOWN(n) \mid TEST(t)]\!] \setminus l1\ l2\ l3$$

where:

$$MULT(m) = \lambda\{l1,l2\}.\{l3 = m\}, MULT(l1 \to 1, m \times l2)$$
$$DOWN(n) = \lambda\{in,l1\}.\{l2 = n\}, DOWN(l1 \to in, n-1)$$
$$TEST(t) = \lambda\{l2,l3\}.\{out = ((l2 = 0) \wedge \neg t \to l3, 0), l1 = t\}, TEST((l2 = 0) \wedge \neg t)$$

Fig.VI. Specification of the components of *FACTIMP(m,n,t)*

MULT(m) and *DOWN(n)* are fairly natural, *TEST(t)* required some experimentation before its specification was got right. Before descending a level and implementing these devices we should check that they do lead to a correct implementation of *FACT*.

By the Composition Theorem:

$$FACTIMP(m,n,t)$$

$$= \lambda\{in\}$$
$$\quad letrec\{l1 = t, l2 = n, l3 = m\}$$
$$\quad in\{out = ((l2 = 0) \wedge \neg t \to l3, 0)\}, FACTIMP((l1 \to 1, m \times l2), (l1 \to in, n-1), (l2 = 0) \wedge \neg t)$$

$$= \lambda\{in\}.\{out = (n = 0 \wedge \neg t \to m, 0)\},\ FACTIMP((t \to 1, m \times n), (t \to in, n-1), n = 0 \wedge \neg t)$$

Define:

$$F(m,n,t) = t \to FACT,\ DELAY(FACT, 0, m \times n!, n)$$

Then we show by Simulation Induction that $FACTIMP=F$ and hence for all m and n:

$$FACTIMP(m,n,true) = F(m,n,true) = FACT$$

Thus as long as t is initialised to $true$ it doesn't matter what m and n are initialised to; we still compute $FACT$.

Case 1: $t=true$

$$FACTIMP(m,n,true)$$
$$= \lambda\{in\}.\{out=0\},\ FACTIMP(1, in, false)$$

$$F(m,n,true)$$
$$= FACT$$
$$= COMPUTE(fact, (\lambda n.n), 0)$$
$$= \lambda\{in\}.\{out=0\},\ DELAY(FACT, 0, in!, in)$$
$$= \lambda\{in\}.\{out=0\},\ F(1, in, false)$$

Case 2: $t=false$

$$FACTIMP(m,n,false)$$
$$= \lambda\{in\}\{out=(n=0 \to m, 0)\},\ FACTIMP(m \times n,\ n-1, n=0)$$

$$F(m,n,false)$$
$$= DELAY(FACT, 0, m \times n!, n)$$
$$= n=0 \to (\lambda\{in\}.\{out=m \times n!\}, FACT),\ (\lambda\{in\}.\{out=0\},\ DELAY(FACT, 0, m \times n!, n-1))$$
$$= \lambda\{in\}.\{out=(n=0 \to m, 0)\},\ (n=0 \to FACT,\ DELAY(FACT, 0, m \times n!, n-1))$$
$$= \lambda\{in\}.\{out=(n=0 \to m, 0)\}, F(m \times n, n-1, n=0)$$

So by Simulation Induction $FACTIMP=F$

Thus if we implement *MULT(m)*, *DOWN(n)* and *TEST(t)* to meet the specifications in Fig. VI, then Fig. V gives a correct implementation of *FACT*.

MULT(m) can be implemented by:

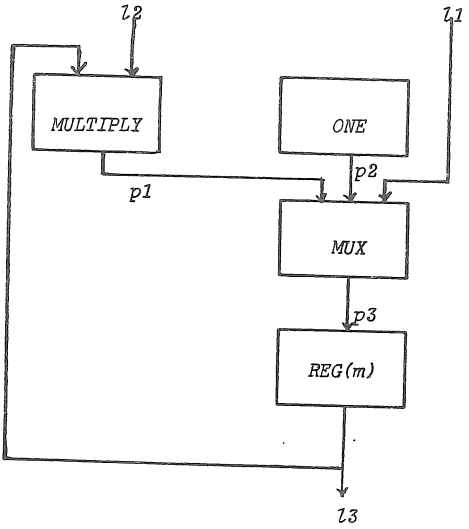


Fig. VIII. Implementation of *MULT(m)*

To model this we define:

$$MULT(m) = [\![MULTIPLY | ONE | MUX | REG(m)]\!] \setminus p1\ p2\ p3$$

where:

$$MULTIPLY = \lambda\{l2, l3\}.\{p1 = l2 \times l3\},\ MULTIPLY$$
$$ONE = \lambda\{\}.\{p2 = 1\},\ ONE$$
$$MUX = \lambda\{p1, p2, l2\}.\{p3 = (l1 \rightarrow p2, p1)\}, MUX$$
$$REG(m) = \lambda\{p3\}.\{l3 = m\},\ REG(p3)$$

By the Composition Theorem:

$$
\begin{aligned}
&MULT(m)\\
&= \lambda\{l1, l2\}.\\
&\quad \text{letrec } \{p1 = l2 \times l3,\ p2 = 1,\ p3 = (l1 \rightarrow p2, p1),\ l3 = m\}\\
&\quad \text{in } \{l3 = m\},\ MULT(p3)\\
&= \lambda\{l1, l2\}.\{l3 = m\},\ MULT(l1 \rightarrow 1,\ m \times l2)
\end{aligned}
$$

As required.

DOWN(n) can be implemented by:

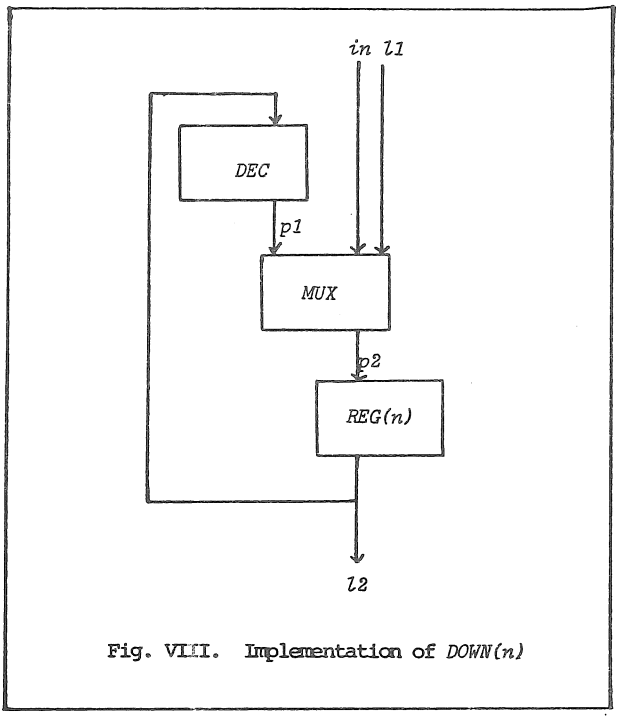


Fig. VIII. Implementation of *DOWN(n)*

To model this we define:

$$DOWN(n) = [\![DEC]REG(n)|MUX]\!] \setminus p1\ p2$$

where:

$$DEC = \lambda\{l2\}.\{p1=l2-1\}, DEC$$
$$REG(n) = \lambda\{p2\}.\{l2=n\}, REG(p2)$$
$$MUX = \lambda\{l1, in, p1\}.\{p2=(l1 \rightarrow in, p1)\}, MUX$$

By the Composition Theorem:

$$DOWN(n)$$
$$= \lambda\{in, l1\}.$$
$$\text{letrec } \{p1=l2-1,\ p2=(l1 \rightarrow in, p1),\ l2=n\}$$
$$\text{in } \{l2=n\}, DOWN(p2)$$
$$= \lambda\{in, l1\}.\{l2=n\}, DOWN(l1 \rightarrow in,\ n-1)$$

As required.

Finally $TEST(t)$ may be implemented by:

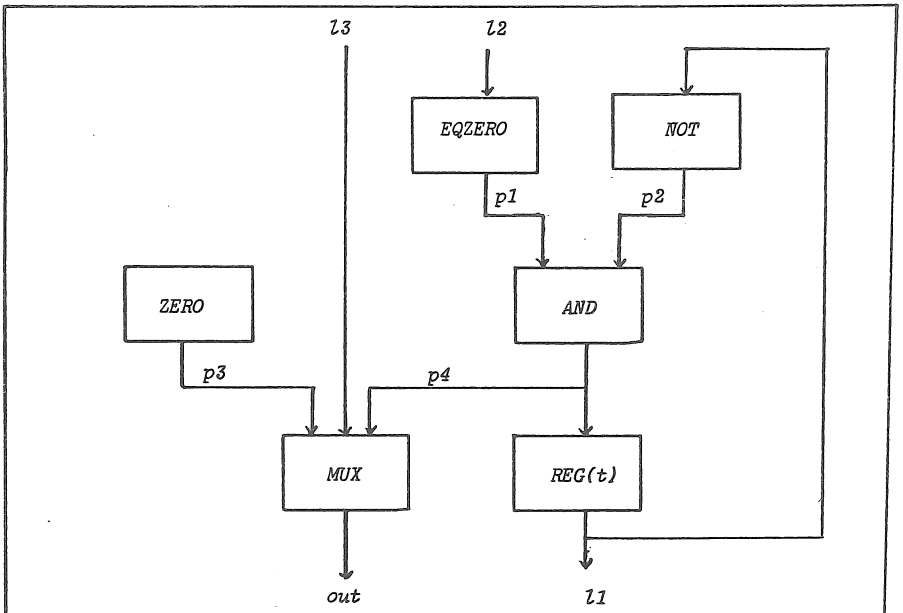


Fig. IX. Implementation of $TEST(t)$

To model this we define:

$$TEST(t) = [\![ZERO | EQZERO | NOT | AND | MUX | REG(t)]\!] \setminus p1\ p2\ p3\ p4$$

where:

$$ZERO = \lambda\{\}.\{p3=0\},\ ZERO$$
$$EQZERO = \lambda\{l2\}.\{p1=(l2=0)\},\ EQZERO$$
$$NOT = \lambda\{l1\}.\{p2=\neg l1\},\ NOT$$
$$AND = \lambda\{p1,p2\}.\{p4=p1 \wedge p2\},\ AND$$
$$MUX = \lambda\{p3,l3,p4\}.\{out=(p4 \rightarrow l3,p3)\},\ MUX$$
$$REG(t) = \lambda\{p4\}.\{l1=t\},\ REG(p4)$$

By the Composition Theorem:

$$TEST(t)$$
$$= \lambda\{l2,l3\}.$$
$$\quad \text{letrec } \{p1=(l2=0),\ p2=\neg l1,\ p3=0,\ p4=p1 \wedge p2,\ l1=t\}$$
$$\quad \text{in } \{out=(p4 \rightarrow l3,p3),\ l1=t\},\ TEST(p4)$$
$$= \lambda\{l2,l3\}.\{out=(l2=0) \wedge \neg t \rightarrow l3,0),\ l1=t\},\ TEST((l2=0) \wedge \neg t)$$

As required.

We have thus shown that the device specified by *FACT* is correctly implemented by *FACTIMP(m,n,true)* as shown in Fig. V, where the components are as specified in Fig. VI, and correct implementation of these components are shown in Fig. VII, Fig. VIII and Fig. IX.

This example illustrates how the verification of large systems can be made tractable. If we remove the structure we imposed on *FACTIMP(m,n,t)* we get the unintelligible tangle of devices shown in Fig. X below:

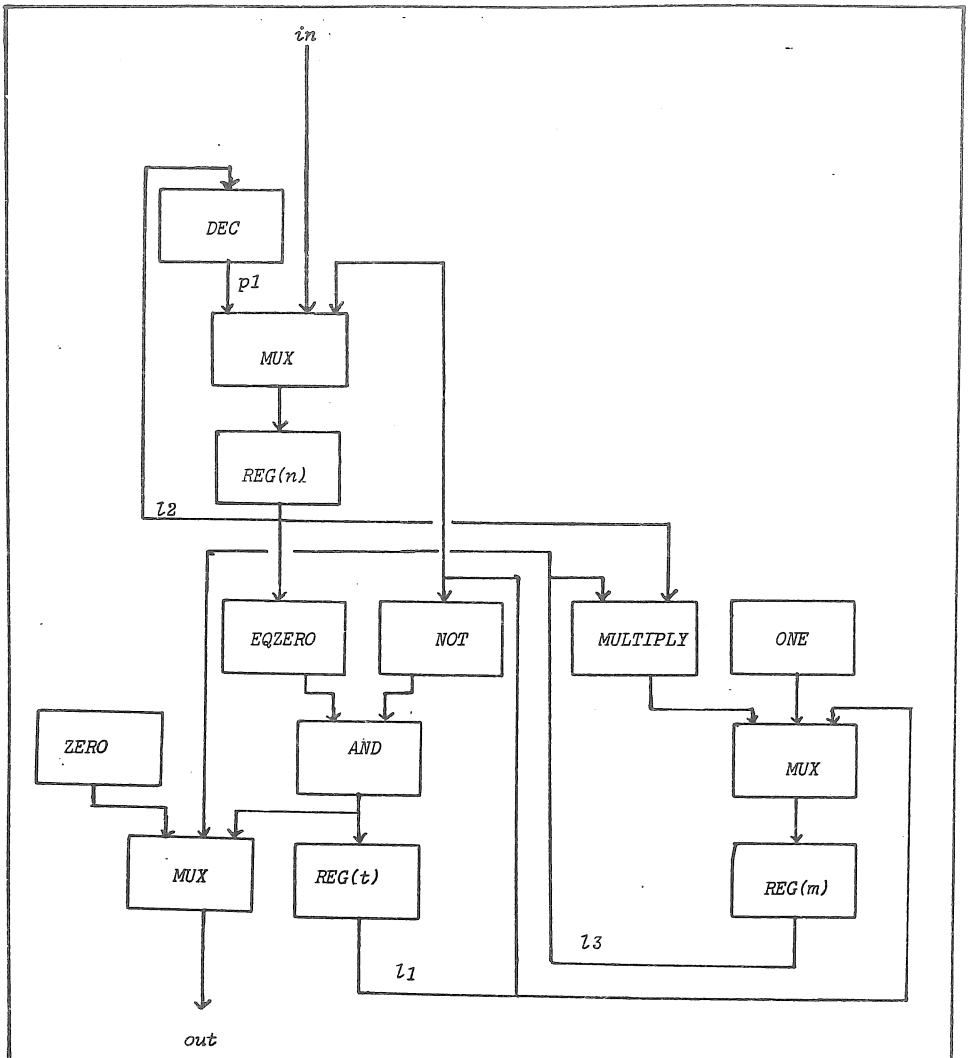


Fig. X Unstructured diagram of *FACTIMP(m,n,t)*

By analysing the whole system in terms of the subsystems *MULT(M)*, *DOWN(n)* and *TEST(t)* we not only made the verification much simpler, but also more robust. For example, if a change is made in a subsystem implementation we need only verify that it meets its specified sub-behaviour (see Fig. VI) - we do not have to repeat the entire proof, as we would if we had based our

analysis on Fig. X rather than Fig. V. The two case studies that follow illustrate this methodology on less trivial examples.

In all our examples so far the correctness of a device has been expressed by asserting that its behaviour should be equal to some specified behaviour. This requires that we fully determine each clock cycle - i.e. the specification and implementation are behaviours which run in 'lockstep'. It is often more natural not to require this, and to allow several 'microcycles' in an implementation to implement a single cycle in the specification. For instance, in our first case study below we will verify a host machine that implements a target machine that fetches, decodes and executes machine code instructions. At the target level fetch-decode-execute will be a single step, but down at the host level each such step will be implemented by several microinstructions and hence take several microcycles. To express the correctness of the host device we cannot simply assert that its behaviour must equal the target behaviour, instead we must first derive from the host's behaviour a 'courser' behaviour in which fetch, decode and execute microcycles are somehow merged into a single cycle, and then it is this derived behaviour which must equal the specified target behaviour. We must thus provide in our model a way of coalescing sequences of clock cycles into single cycles. The intuitive idea of what is needed is most simply explained with respect to machines (see page 8) rather than behaviours.

Suppose we have a machine $M=(S,out,next)$ where S is a set of states and

$$out : Sig[X] \times S \to Sig[Y]$$

$$next : Sig[X] \times S \to S$$

are the output and next-state functions respectively. Suppose that this machine implements a 'higher level' machine whose states correspond to some subset $S' \subseteq S$ - think of states in $(S - S')$ as occurring in the middle of microinstruction sequences. From M it seems natural to derive a new machine $M \downarrow S'=(S',out',next')$ where:

$$out' : Sig[X] \times S' \to Sig[Y]$$

$$next' : Sig[X] \times S' \to S'$$

and in which out' is the restriction of out to $Sig[X] \times S'$ and $next'(s,x)$ is obtained by repeatedly clocking machine M starting in state x and with constant input signal s until a member of S' is reached. Formally:

$$out'(s,x) = out(s,x)$$

$$next'(s,x) = (next(s,x) \in S') \rightarrow next(s,x),\ next'(s,next(s,x))$$

The idea is that given a target specification as a behaviour $f \in Seq[X;Y]$ we can express correctness by requiring that $f = \mathfrak{B}[M \downarrow S']$.

Unfortunately this construction of $M \downarrow S'$ from M does not respect behaviour in the sense that from $\mathfrak{B}[M_1] = \mathfrak{B}[M_2]$ it does not necessarily follow that $\mathfrak{B}[M_1 \downarrow S'] = \mathfrak{B}[M_2 \downarrow S']$. This has the unsatisfactory consequence that replacing a machine M_1 which is correct (in the sense that $\mathfrak{B}[M \downarrow S']$ has a specified behaviour) by a behaviourably identical machine M_2 does not necessarily preserve correctness (since $\mathfrak{B}[M_2 \downarrow S']$ might not have the specified behaviour).

Example 10

Let $M_1 = (S, out_1, next_1)$ and $M_2 = (S, out_2, next_2)$ where:

$$S = Int \times Bool$$
$$out_1(s,(n,t)) = out_2(s,(n,t)) = \{o = n\}$$
$$next_1(s,(n,t)) = (n+1, t)$$
$$next_2(s,(n,t)) = (n+1, \neg t)$$

Then it it easy to show that $\mathfrak{B}[M_1] = \mathfrak{B}[M_2] = \lambda(n,t).F(n)$ where:

$$F(n) = \lambda s.\{o = n\}.F(n+1)$$

Now let $S' = \{(n,t) \mid t = true\} \subseteq S$, then

$$M_1 \downarrow S' = (S', out_1', next_1')$$
$$M_2 \downarrow S' = (S', out_2', next_2')$$

where

$$out_1'(s,(n,t)) = out_2'(s,(n,t)) = \{o = n\}$$
$$next_1'(s,(n,true)) = (n+1, true)$$
$$next_2'(s,(n,true)) = (n+2, true)$$

and so $\mathfrak{B}[M_1 \downarrow S'] \neq \mathfrak{B}[M_2 \downarrow S']$.

To overcome the problem illustrated in this example we shall define an operation on machines which merges cycles, not on the basis of the states they pass through, but instead on the basis of the output signals produced.

Definition 11

Let M $(S, out, next)$ be a machine where:

S is the set of states
$out: Sig[X] \times S \to Sig[Y]$ is the output functions
$next: Sig[X] \times S \to S$ is the next-state function

If $P: Sig[Y] \to Bool$ is a predicate on output signals, then define the machine $M \downarrow P$ by $M \downarrow P = (S, out, next_P)$ where:

$$next_P(s,x) = P(out(s, next(s,x))) \to next(s,x),\ next_P(s, next(s,x))$$

Thus $M \downarrow P$ is got from M by changing the next-state function so that it moves to the next state in which the output satisfies P. While this moving is taking place the input signal is held constant.

This definition of $M \downarrow P$ has the desirable property that if $\mathfrak{B}[M_1] = \mathfrak{B}[M_2]$ then $\mathfrak{B}[M_1 \downarrow P] = \mathfrak{B}[M_2 \downarrow P]$. This follows from the fact that for all x $\mathfrak{B}[M \downarrow P](x) = \mathfrak{B}[M](x) \downarrow P$, where $\downarrow P$ is defined on behaviours by.

Definition 12

Let $f \in Seq[X;Y]$, $P: Sig[Y] \to Bool$. Define $f \downarrow P \in Seq[X;Y]$ by:

$$f \downarrow P = \lambda s.(fst(f\,s),\ run\ P(snd(f\,s))s \downarrow P)$$

where $run\ P\,f\,s = P(fst(f\,s)) \to f,\ run\ P(snd(f\,s))\,s$

intuitively $run\ P\ f\ s$ moves to the 'nearest' behaviour after f in which the output satisfies P.

Theorem 4

For all machines M, output predicates P and states x :

$$\mathfrak{B}[M \downarrow P](x) = \mathfrak{B}[M](x) \downarrow P$$

Example 11

Let $FACTORIAL$ be defined by:

$$FACTORIAL(n) = \lambda\{in = x\}.\{out = n\}, FACTORIAL(x!)$$

and $FACTIMP$ be as constructed in Example 9, i.e.

$$FACTIMP(m, n, t) = \lambda\{in\}.\{out = (n = 0 \wedge \neg t \rightarrow m, 0)\}, FACTIMP(t \rightarrow (1, in, false), (m \times n, n - 1, n = 0))$$

Then we show that if we define $P(s) \iff s(out) \neq 0$ then:

$$FACTIMP(m, n, true) \downarrow P = \lambda\{in\}.\{out = 0\} \ , \ FACTORIAL(m!)$$

$$FACTIMP(m, 0, false) \downarrow P = FACTORIAL(m)$$

This assertion is a weaker specification than the one in Example 9 since we only require that the factorial function is computed in some number of steps - we do not specify the number.

By Simulation Induction it is easy to see that $FACTIMP = \mathcal{B}[FACTMACHINE]$

where: $FACTMACHINE = (Int \times Int \times Bool, out, next).$

$$out(s, (m, n, t)) = \{out = (n = 0 \wedge \neg t \rightarrow m, 0)\}$$
$$next(s, (m, n, t)) = t \rightarrow (1, s(in), false), (m \times n, n - 1, n = 0)$$

Then $FACTMACHINE \downarrow P = (Int \times Int \times Bool, out, next_P)$ where if $n > 0$:

$$\begin{aligned} & next_P(\{in = x\}, (m, n, false)) \\ = \ & next_P(\{in = x\}, (m \times n, n - 1, false)) \\ = \ & next_P(\{in = x\}, (m \times n \times n - 1, n - 2, false)) \\ & \vdots \\ = \ & next_P(\{in = x\}, (m \times n \times n - 1 \times \ldots \times 2, 1, false)) \\ = \ & (m \times n! \ , 0, false) \end{aligned}$$

and hence if $x > 0$

$$\begin{aligned} & next_P(\{in = x\}, (m, n, true)) \\ = \ & next_P(\{in = x\}, (1, x, false)) \\ = \ & (x!, 0, false) \end{aligned}$$

and so:

$$next_P(\{in=x\},(m,0,false))$$
$$= next_P(\{in=x\},(0,-1,true))$$
$$=(x!,\ 0,false)$$

It follows that:

$$\mathcal{B}[\,FACTMACHINE \downarrow P\,](m,n,true)$$
$$= \lambda\{in=x\}.\{out=0\},\ \mathcal{B}[\,FACTMACHINE \downarrow P\,](x!,0,false)$$

and:

$$\mathcal{B}[\,FACTMACHINE \downarrow P\,](m,0,false)$$
$$= \lambda\{in=x\}.\{out=m\},\ \mathcal{B}[\,FACTMACHINE \downarrow P\,](x!,0,false)$$

and hence (by Simulation Induction)

$$\mathcal{B}[\,FACTMACHINE \downarrow P\,](m,0,false) = FACTORIAL(m)$$

and so

$$\mathcal{B}[\,FACTMACHINE \downarrow P\,](m,n,true) = \lambda\{m\}.\{out=0\},\ FACTORIAL\ (in!)$$

Actually, as the reader may have noticed, the proof is only partial; we have not considered the cases when the devices fail to terminate (i.e. when we input a negative integer). We leave this as an exercise for those who like fiddling with $\perp$. Alternatively, the problem could be eliminated by imposing suitable restrictions on the states and inputs.

We have now concluded the description of our model. In the next two sections we present two case studies which illustrate the range of applications we have in mind. In the first of these we completely specify a small general purpose computer, and then prove correct a microcoded implementation. In the second we show how devices like those used to implement the computer can be realised in nMOS. We would have like to have combined the two studies to yield a verification of an nMOS implementation of the computer, but there are too many details to manage by hand. We hope, however, to convince the reader that our model is capable of supporting such an analysis, and that by careful structuring of behaviour an explosion of proof size can be avoided.

First case study: correctness of architecture and microcode

In this case study we illustrate how register transfer systems can be used to prove microcode correct. Our hope (and expectation) is that the kind of analysis described below can be scaled up to non-trivial examples, but to do this we will need machine assistance.

The system we shall verify implements the computer shown in Fig. XI.

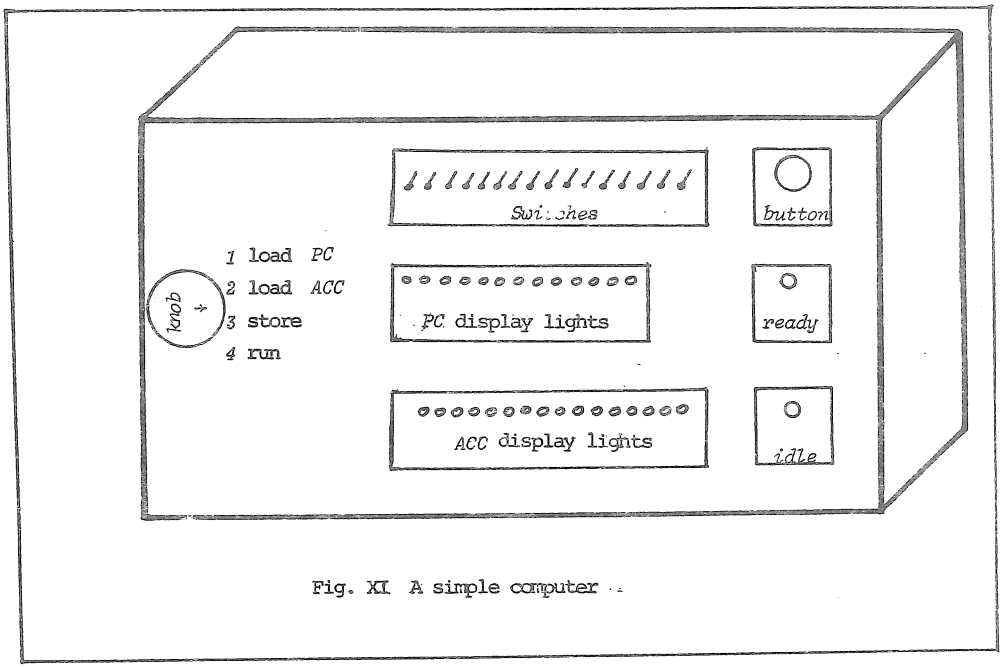


Fig. XI A simple computer

This computer has two registers: the program counter *PC* which is *13* bits wide, and the accumulator *ACC* which is *16* bits wide. It has a random access memory which can store 2^{13} *16* bit words.

On the front panel there is a four position knob which determines what happens when the button on the right of the panel is pressed. There are three sets of lights: thirteen *PC* display lights which show the contents of the program counter; sixteen *ACC* display lights which show the contents of the accumulator; the ready light which is on when the computer is interruptable, and the idle light which is on when the computer is idling - i.e. not executing a program. There is also a bank of sixteen two position switches which are used for manually inputting data.

If the knob is in position $n (1 \leq n \leq 4)$, and the computer is ready and idling (i.e. the ready and idle lights are on), then pressing the button will cause the following to happen:

$n = 1$: The word determined by the state of the thirteen rightmost switches is loaded into *PC*.

$n = 2$: The word determined by the state of the sixteen switches is loaded into *ACC*.

$n = 3$: The contents of *ACC* will be stored in memory at the location stored in *PC*.

$n = 4$: The program stored in memory will be executed starting at the location in *PC*. When the execution starts the idle and ready lights will go off. The idle light will stay off until the execution stops; this happens either when a halt instruction is reached or when an interrupt is generated by pressing the button. Interrupts are only accepted at the end of execution of each machine code instruction; readiness to accept an interrupt is indicated by the ready light being on. Thus to stop an executing program one must keep one's finger on the button until the ready light is on.

The instruction set for our little computer is shown in Fig. XII below

Format

Instructions:	Assembler mnemonic	Meaning
0|0|0 L	*HALT*	Stops execution of program
0|0|1 L	*JMP L*	Jumps to *L*
0|1|0 L	*JZRO L*	Jumps to *L* if *ACC* contains *0*
0|1|1 L	*ADD L*	Add contents of *L* to *ACC*
1|0|0 L	*SUB L*	Subtract contents of *L* from *ACC*
1|0|1 L	*LD L*	Load contents of *L* into *ACC*
1|1|0 L	*ST L*	Store contents of *ACC* in *L*
1|1|1 L	*SKIP*	Skip to next instruction

Fig. XII Machine Code Instructions

Before formally specifying the semantics of our computer we need some definitions and notation for talking about bitstrings.

Let *Word*[n] be the (flat) domain of n bit words. We will also regard members of *Word*[n] as integers less than 2^n and as n-tuples of truthvalues (so *Word*[n] = *Bool*×...×*Bool*). We interpret + (plus) and - (minus) on words as two's complement operators.

If $w \in$ *Word*[n] and $a \leq b \leq n$ then $w\langle a{:}b\rangle \in$ *Word*[$b-a+1$] is the word corresponding to bits a to b of w. If $a \leq n \leq b$ then $w\langle a{:}b\rangle$ is the top $n-a+1$ bits of w extended with 0's to a word of size $b-a+1$.

For example, if $w \in Word[16]$ then $w\langle 0:12\rangle$ is the address field of w and $w\langle 13:15\rangle$ is the opcode field; if $w \in Word[13]$ then $w\langle 0:15\rangle \in Word[16]$ is the 16 bit word obtained by extending w with three 0's. Thus $(w\langle 0:12\rangle)\langle 0:15\rangle$ is the word obtained from $w \in Word[16]$ by zeroing the opcode. Note that from our conventions it follows that if $w \in Word[13]$ then $(w\langle 0:15\rangle + 1)\langle 0:12\rangle = w + 1$. We use $w\langle a\rangle$ to mean the a^{th} bit of w; if we regard bits as words of length one then $w\langle a\rangle = w\langle a:a\rangle$.

In our computer, addresses are 13 bits long, and contents of addresses are 16 bits long, hence to model the memory we define:

$$Mem = Word[13] \to Word[16]$$

A member $m \in Mem$ is a function giving the contents $m(w) \in Word[16]$ of each address $w \in Word[13]$. To model the semantics of storing we define $m[w_2/w_1] \in Mem$ where $m \in Mem$, $w_1 \in Word[13]$, $w_2 \in Word[16]$ to be the function identical to m except at w_1 which it maps to w_2, i.e:

$$(m[w_2/w_1])w = \begin{cases} w_2 & \text{if } w = w_1 \\ m(w) & \text{otherwise} \end{cases}$$

The state of our computer is characterised by the contents of the memory, the program counter and the accumulator, hence we define:

$$State = \underset{\substack{\uparrow \\ \text{contents} \\ \text{of memory}}}{Mem} \times \underset{\substack{\uparrow \\ \text{contents} \\ \text{of } PC}}{Word[13]} \times \underset{\substack{\uparrow \\ \text{contents} \\ \text{of } ACC}}{Word[16]}$$

Thus, for example, if the computer is in state $(m, w_1, w_2) \in State$ and the knob is set to position 3 and the idle and ready lights are on, and the button is pushed, then the machine will move to state $(m[w_2/w_1], w_1, w_2)$.

We shall specify the semantics of the computer with two functions:

$$COMPUTER,\ EXECUTE\ :\ State \to Seq[\{knob, switches, button\}; \{pc, acc, ready, idle\}]$$

$COMPUTER(m, w_1, w_2)$ gives the behaviour of the machine when it is ready and idling; $EXECUTE(m, w_1, w_2)$ gives its behaviour when it is ready (for interrupts) but executing code (i.e. not idling). We do not specify the behaviour when the machine is not ready; why not should become clear later. The definitions of *COMPUTER* and *EXECUTE* are shown in Fig.XIII below; this constitutes the machine code, or 'target level' specification of the computer. The implementation which we will prove correct is based on the 'host' machine shown in Fig. XIV.

The host machine, which we will microprogram to emulate the target machine specified in Fig. XIII, is shown in Fig. XIV and has behaviour:

$$HOST\ r\ (w, m, w_0, w_1, w_2, w_3, w_4, w_5) \in Seq[\{knob, button, switches\}; \{pc, acc, ready, idle\}]$$

We have separated out the *ROM* contents r because later we shall characterise $HOST(mucode)$ where *mucode* is a particular microprogram. The host's behaviour is much too complicated to define directly by a recursive definition like those in Fig. XIII, instead we will express it as a composition of the simpler behaviours of its various components. These naturally fall into two parts: first, a data part consisting of the memory $MEM(m)$, the memory address register $MAR(w_0)$, the program counter $PC(w_1)$, the accumulator $ACC(w_2)$, the instruction register $IR(w_3)$, the argument register $ARG(w_4)$, the buffer register $BUF(w_5)$, the bus *BUS*, the arithmetic unit *ALU*, and the microcode controlled gates *G0, G1, G2, G3, G4*, and second, a microprogrammed control part $CONTROL(r, w)$, which is shown in the dotted box, and has behaviour determined by *ROM* contents r (the microcode) and start address w.

The microcode stored in the read-only memory $ROM(r)$ is shown in Fig.XVI written in a 'microassembler' notation which we explain shortly. If we name this chunk of microcode *mucode* and if we define the predicate P_{ready} by:

$$P_{ready}(s) = true \iff s(ready) = 1$$

then we will verify our implementation by proving for all $m, w_0, \ldots, w_5$ that

$$COMPUTER(m, w_1, w_2) = HOST\ mucode\ (0, w_0, w_1, w_2, w_3, w_4, w_5) \downarrow P_{ready}$$

i.e. if we merge all non-interruptable 'microcycles' into a single 'macrocycle' then the behaviour of the computer and the host are equal.

```
COMPUTER(m,w1,w2)
  = λ{knob,switches,button},
      {pc=w1, acc=w2, ready=1, idle=1},
        (¬button → COMPUTER(m,w1,w2),
          button → (knob = 1 → COMPUTER(m,switches<0:12>,w2),
                    knob = 2 → COMPUTER(m,w1,switches),
                    knob = 3 → COMPUTER(m[w2/w1],w1,w2),
                    knob = 4 → EXECUTE(m,w1,w2)))


EXECUTE(m,w1,w2)
  = λ{knob,switches,button},
      {pc=w1, acc=w2, ready=1, idle=0}
        let {op=m(w1)<13:15>, addr=m(w1)<0:12>}
        in  (op = 0 ∨ button → COMPUTER(m,w1,w2),
             op = 1          → EXECUTE(m,addr,w2),
             op = 2          → EXECUTE(w2 = 0 → (m,addr,w2), (m,w1 + 1,w2)),
             op = 3          → EXECUTE(m,w1 + 1,w2 + m(addr)),
             op = 4          → EXECUTE(m,w1 + 1,w2 - m(addr)),
             op = 5          → EXECUTE(m,w1 + 1,m(addr)),
             op = 6          → EXECUTE(m[w2/addr],w1 + 1,w2),
             op = 7          → EXECUTE(m,w1 + 1,w2))
```

FIG. XIII. Specification of the computer

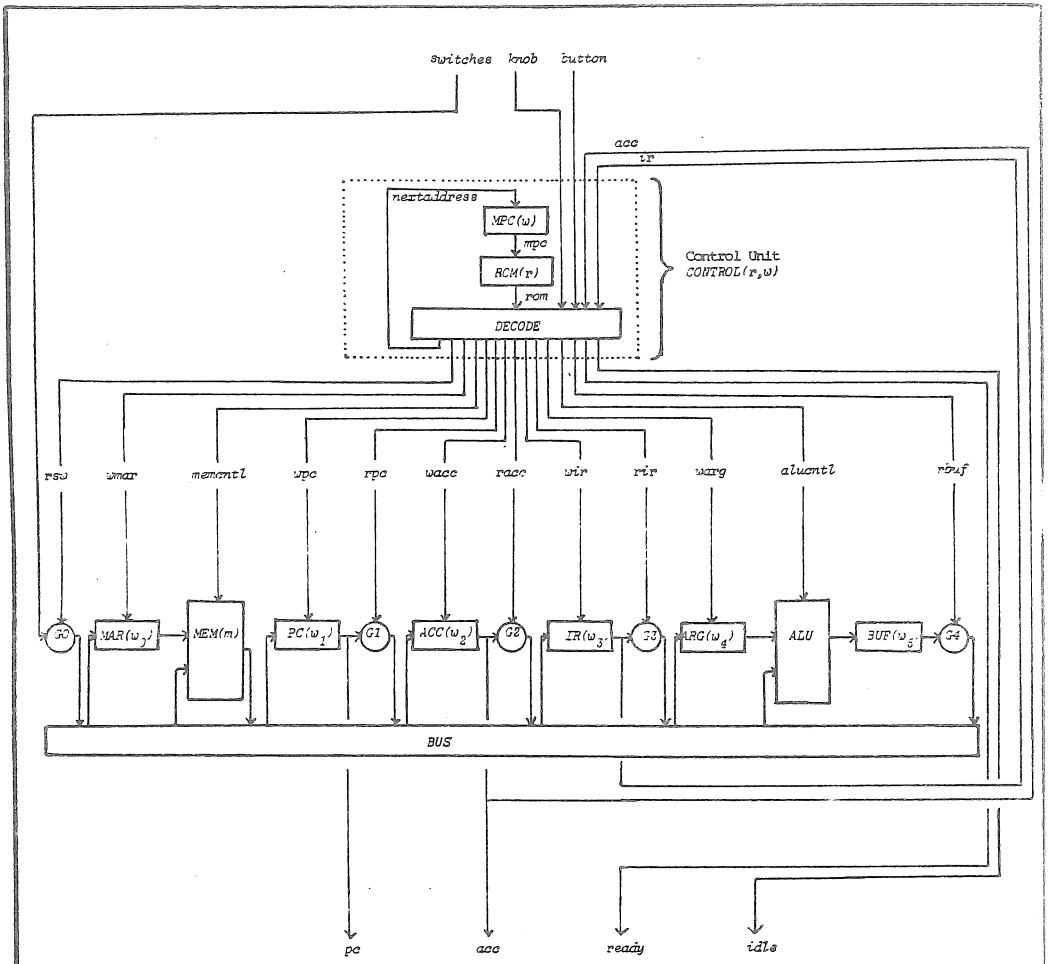


Fig. XIV: The host machine for implementing the computer

The control lines *rsw, wwar, wpc, rpc, wacc, racc, wir, rir, warg, rbuf, ready, idle* are all one bit wide as is the input line *button* from the button. The line *knob* from the knob is two bits wide (to encode the four positions), as are the control lines *memcntl* and *alucntl* to the memory and arithmetic unit respectively. All other lines are sixteen bits wide (*PC* and *MAR* ignore the three most significant bits of their input, and pad their outputs to sixteen bits with *0*'s). When a gate (i.e. *G0, G1, G2, G3* or *G4*) is off (i.e. its control input is *0*) then it outputs a special value $\emptyset$ on to its output line. The idea is that this represents 'floating' or 'high impedance'; only one non-$\emptyset$ must be put on the bus at once for correct operation. Thus the bus is defined by:

$$BUS = \lambda\{g0, g1, g2, g3, g4, mem\}.\{bus = join(g0, g1, g2, g3, g4, mem)\},\ BUS$$

where $$join\ (x_1, x_2, \ldots, x_n) = \begin{cases} x_i & \text{if } x_j = \emptyset \text{ for all } j \neq i \\ \text{undefined otherwise} & \end{cases}$$

and a typical gate G with input in, Control $cntl$ and output out is defined by:

$$G = \lambda\{in, cntl\}.\{out = (cntl \rightarrow in, \emptyset)\},\ G$$

The exact definition of gates $G0, G1, G2, G3, G4$ are:

$$G0 = \lambda\{switches, rsw\}.\{g0 = (rsw \rightarrow switches, \emptyset)\},\ G0$$

$$G1 = \lambda\{pc, rpc\}.\{g1 = (rpc \rightarrow pc, \emptyset)\},\ G1$$

$$G2 = \lambda\{acc, racc\}.\{g2 = (racc \rightarrow acc, \emptyset)\},\ G2$$

$$G3 = \lambda\{ir, rir\}.\{g3 = (rir \rightarrow ir, \emptyset)\},\ G3$$

$$G4 = \lambda\{buf, rbuf\}.\{g4 = (rbuf \rightarrow buf, \emptyset)\},\ G4$$

Notice that all these definitions are obtained from the definition of the 'generic gate' G by renaming lines. Thus - in what we hope is a self explanatory notation - we could have written:

$$G0 = G\{in \mapsto switches, cntl \mapsto rsw, out \mapsto g0\}$$

$$G1 = G\{in \mapsto pc, cntl \mapsto rpc, out \mapsto g1\}$$

Such a notation is essential to handle devices built by connecting together large numbers of identical components. However, for simplicity we shall avoid introducing definitions and manipulative laws necessary to support line renaming. This will lead to some verbosity both in this case study and the next one.

The registers *MAR, PC, ACC, IR, ARG* all have a microprogram controlled write line. A typical one, with input *in*, write line *wreg*, and output *out* has behaviour:

$REG(w) = \lambda\{in,wreg\}.\{out=w\},\ REG(wreg \rightarrow in,\ w)$

Since *MAR* and *PC* only hold thirteen bit words we must adjust the size of values got off and put onto the sixteen bit wide bus. The behaviour of these two registers is thus specified to be:

$MAR(w_0) = \lambda\{bus,wmar\}.\{mar=w_0 <0:15>\},\ MAR(wmar \rightarrow bus\ <0:12>,\ w_0)$

$PC(w_1) = \lambda\{bus,wpc\}.\{pc = w_1 <0:15>\},\ PC(wpc \rightarrow bus\ <0:12>,\ w_1)$

The other registers are sixteen bits wide, hence:

$ACC(w_2) = \lambda\{bus,wacc\}.\{acc=w_2\},\ ACC(wacc \rightarrow bus,\ w_2)$

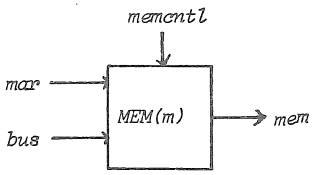
$IR(w_3) = \lambda\{bus,wir\}.\{ir=w_3\},\ IR(wir \rightarrow bus,\ w_3)$

$ARG(w_4) = \lambda\{bus,warg\}.\{arg=w_4\},\ ARG(warg \rightarrow bus,\ w_4)$

The *ALU's* buffer register *BUF* has no write line as it always stores the value on its input, hence:

$BUF(w_5) = \lambda\{alu\}.\{buf=w_5\},\ BUF(alu)$

The memory

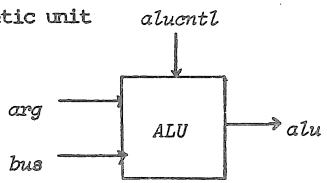


is a sequential device whose state is determined by a memory function $m \in Mem$ giving the contents of each address. Its behaviour is:

$$MEM(m) = \lambda\{mar,bus,memcntl\}$$
$$(memcntl = 1 \rightarrow (\{mem=m(mar)\},\ MEM(m)),$$
$$(memcntl = 2 \rightarrow (\{mem=\oplus\},\ MEM(m[bus/mar])),$$
$$(\{mem=\oplus\},\ MEM(m))))$$

Thus *1* on *memcntl* causes a read - i.e. the contents of the value on line *mar* is output; *2* on *memcntl* causes a write - i.e. results in the address on line *mar* being updated to contain the value on *bus*. Except during a read the memory puts $\oplus$ on to the bus.

The arithmetic unit



is a combinational device with behaviour given by:

$$ALU = \lambda\{arg, bus, alucntl\}.$$
$$\{alu = (alucntl = 0 \rightarrow bus,$$
$$alucntl = 1 \rightarrow bus+1$$
$$alucntl = 2 \rightarrow arg+bus$$
$$alucntl = 3 \rightarrow arg-bus)\}, ALU$$

Thus 0 on the control line causes the value on the bus to be passed through unchanged; 1 on the control line causes one plus the value on the bus to be output; 2 on the control line causes the sum of the two inputs to be output and 3 on the control line causes their difference to be output.

We can now define the data part of the host machine by:

$$DATA(m, w_0, w_1, w_2, w_3, w_4, w_5)$$
$$= [\![MEM(m) | MAR(w_0) | PC(w_1) | ACC(w_2) | IR(w_3) | ARG(w_4) | BUF(w_5) |$$
$$G0 | G1 | G2 | G3 | G4 | ALU | BUS]\!] \setminus L$$
$$\text{where } L = \{mem, mar, arg, buf, g0, g1, g2, g3, g4, alu, bus\}$$

The control unit $CONTROL(r,w)$ emits a sequence of signals to the other devices which cause them to perform the appropriate operations. It is a sequential device whose behaviour is defined by a microprogram stored in the read only memory ROM starting at address w. We shall go into details of the microcode soon put first, to give an idea of how the control unit works, we describe two examples which illustrate how emitting the appropriate sequence of signals causes the right thing to happen.

Suppose the knob is set to position 1 (load PC), the ready and idle lights are on, and the button is pushed. The control unit will then put 1 on the lines rsw and wpc and 0 on all other lines. Thus gate $G0$ will open and the value from the switches - i.e. value on line $switches$ will be put on the bus. During the same cycle, since the program counter write line wpc carries a 1, the value on the bus (i.e. the value from the switches) will be written into PC. Thus the word set up on the switches is loaded into the program counter in one microcycle.

Suppose next that the knob is set to position 3 (store) and the button is pushed. The control unit in this case will first signal 1 on lines *rpc* and *wmar* and *0* on all other lines. This causes *G1* to open, and so the contents of *PC* is put on the bus and then this is written into the memory address register *MAR*. To complete the store operation, on the next cycle the control unit signals *1* on line *racc* to put the contents of the accumulator on the bus, and during the same cycle signals *2* (i.e. write) on *memcntl* causing the memory to store the value which is on the bus (viz. the contents of *ACC*) at the address in *MAR* (viz. the contents of *PC*). Thus the contents of *ACC* is stored at the address in *PC* in two microcycles.

Note that the descriptions above are imprecise and incomplete; full details can be found in the microcode, which we now describe.

In order to complete the description of the host machine all we have to do is specify the control unit, and to do this we must describe the microinstructions which it interprets. These microinstructions are represented by *30*-bit words stored in a read only memory (the *ROM*) which can hold thirty-two microinstructions.

Each microinstruction has a *3*-bit test field, two *5*-bit address fields (the A-address field and the B-address field), two *2*-bit control fields (for *memcntl* and *alucntl*), twelve single bit control fields and one unused bit (just in case we need it later!). The format is shown in Fig. XV below:

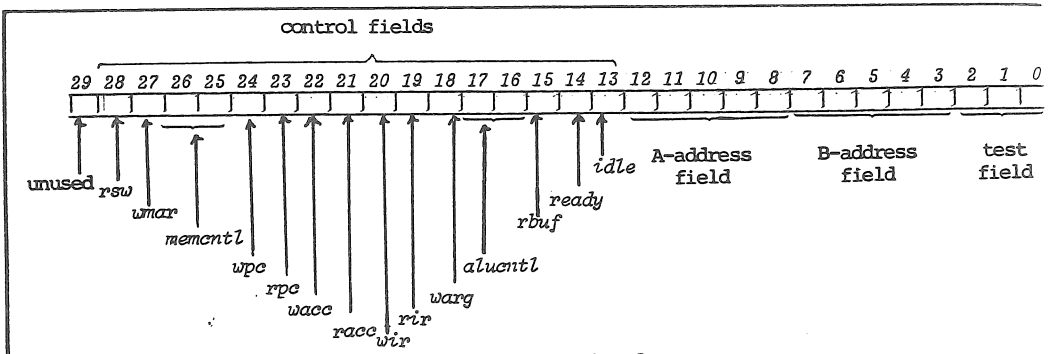


Fig. XV Microinstruction format

During each (micro) cycle the value put on a control line is the value in the corresponding control field of the current microinstruction. For example, the microinstruction to load the program counter from the switches must cause a *1* on lines *rsw* and *wpc* and *0* on all other lines, thus it has bits *28* and *24* set to *1* and all other control bits set to *0*.

The *ROM*-address of the next microinstruction to be interpreted after the current one (i.e. on the next microcycle) is normally the contents of the A-address field, unless:

(i) The test field contains *1* (i.e. binary *001*) and *1* is input on the *button* line, or the test field contains *2* (i.e. binary *010*) and *0* is on the *acc* line. In either of these two cases the next address is the contents of the B-address field.

(ii) The test field contains *3* (i.e. binary *011*). In this case the next address is obtained by adding the value on the *knob* line to the contents of the A-address field.

(iii) The test field contains *4* (i.e. binary *100*). In this case the next address is obtained by adding the opcode field (i.e. bits *13* to *15*) of the value on the *ir* line to the contents of the B-address field.

Thus if the test field of the current microinstructions contains *1* then pressing the button causes a branch to the B-address; if the test field contains *2* then a branch to the B-address occurs if the accumulator contains *0*; if the test field contains *3* then a branch to a microinstruction determined by the position of the knob occurs, and if the test field contains *4* then a branch to a microinstruction determined by the opcode of the current machine code instruction (i.e. the contents of the instruction register) occurs.

Before describing the microcode resident in the control unit's *ROM* we need a 'microassembler' notation to enable us to write down microinstructions compactly and help us remember their effect.

To specify the control bits of a microinstruction we define a set of 'microoperations' - i.e. atomic control signals. These consist of the signals for reading the contents of the registers on to the bus:

rsw, rpc, racc, rir, rbuf

The signals for writing the value on the bus into the various registers:

wmar, wpc, wacc, wir, warg

The signals controlling the memory:

read, write

The signals controlling the arithmetic unit:

inc, sum, dif

and the signals controlling the two single lights:

ready, idle

We say a microinstruction w causes a microoperation *op* if:

(i) $op \in \{rsw, rpc, racc, rir, rbuf, wmar, wpc, wacc, wir, ready, idle\}$ and the bit in w corresponding to *op* via Fig. XV is *1*. For example, if bits *24* and *28* of w are *1* then w causes microoperations *wpc* and *rsw*.

(ii) *op=read* and bits *25* and *26* of w contain *1* and *0* respectively (so line *memcntl* carries *1*)

(iii) *op=write* and bits *25* and *26* of w contain *0* and *1* respectively (so line *memcntl* carries *2*)

(iv) *op=inc* and bits *16* and *17* of w contain *1* and *0* respectively (so line *alucntl* carries *1*)

(v) *op=sum* and bits *16* and *17* of w contain *0* and *1* respectively (so line *alucntl* carries *2*)

(vi) *op=dif* and bits *16* and *17* of *1* contain *1* and respectively (so line *alucntl* carries *3*)

We specify the control bits of a microinstruction by giving a list of the microoperations it causes, and requiring that all fields not specified default to *0*. For example, the list *racc, write* specifies that bits *21* and *26* contain *1* and all other control bits contain *0*.

To describe a complete microinstruction we use the notation

$op_1, \ldots, op_n$; *address*

where $op_1, \ldots, op_n$ is a list of the microoperations to be caused, and *address* is an expression specifying the test and address fields as follows:

(i) To specify that the address of the next microinstruction is $n \in \{0, 1, \ldots, 31\}$ we write:

n

(i.e. we take *address=n*). This assembles into a microinstruction with *0* in the test field and *n* in the A-address field.

(ii) To specify that the address of the next microinstruction is *n* if the button is pressed and *m* otherwise we write:

button → n,m

This assembles into a microinstruction with *1* in the test field and *m,n* in the A- and B-address fields respectively.

(iii) To specify that the address of the next microinstruction is *n* if the accumulator contains *0* and *m* otherwise we write:

acc=0 → n,m

This assembles into a microinstruction with *2* in the test field and *m,n* is then A- and B-address fields respectively.

(iv) To specify that the address of the next microinstruction is *n* plus the knob position we write:

knob+n

This assembles into a microinstruction with *3* in the test field and *n* in the A-address field.

(v) To specify that the address of the next microinstruction is *n* plus the contents of the opcode field of the current machine code instruction we write:

opcode+n

This assembles into a microinstruction with *4* in the test field and *n* in the A-address field.

For example:

ready, idle; button → 1, 0

denotes the microinstruction which signals *1* on the *ready* and *idle* lines and *0* on all other control lines. The address of the next microinstruction (in the *ROM*) is *1* if the button is being pushed and *0* otherwise. The actual microinstruction denoted by this expression is:

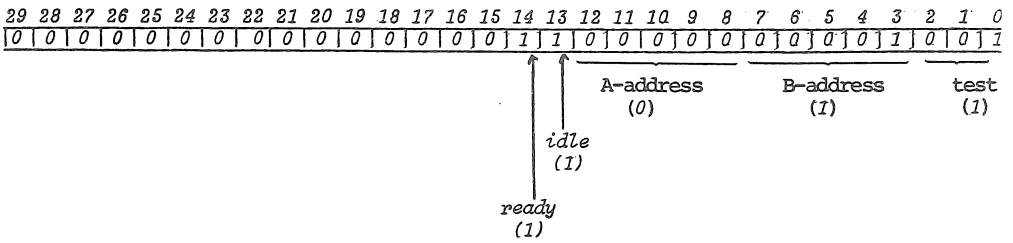


Fig. XVI shows the microcode for our computer - i.e. the contents of the *ROM*. Only *26* of the *32 ROM* - address are used. The comments should be self explanatory. For example, $PC \rightarrow MAR$ means move contents of *PC* into *MAR*; $ARG+MEM(MAR) \rightarrow BUF$ means add the contents of *ARG* and the value in memory addressed by *MAR*, and put the result into *BUF*.

In order to prove that the control unit is correct we must first specify the semantics of microinstruction formally.

Let

$$Rom = Word[5] \rightarrow Word[30]$$

members $r \in Rom$ are functions which represent a possible *ROM* contents by specifying for each *ROM* address $w \in Word[5]$ a microinstruction $r(w) \in Word[30]$. The semantics of a microprogram r with starting address w is:

$$MICROCODE(r,w) \in Seq[\{knob, button, ir, acc\};$$
$$rsw, wmar, memcntl, wpc, rpc, wacc, racc, wir, rir,$$
$$warg, alucntl, rbuf, ready, idle\}]$$

which is defined in Fig. XVII below.

ROM address	Microinstruction		Comments
0	ready,idle;	button → 1,0	branch to 1 if button pressed, otherwise loop
1	;	knob+1	decode knob position
2	rsw,wpc ;	0	switches → PC
3	rsw,wacc ;	0	switches → ACC
4	rpc,wmar ;	7	PC → MAR
5	ready ;	button → 0,6	begin fetch-decode-execute cycle
6	rpc,wmar ;	8	PC → MAR
7	racc,write;	0	ACC → MEM(MAR)
8	read,wir ;	9	MEM(MAR) → IR
9	;	opcode+10	decode
10	;	0	halt
11	rir,wpc ;	5	JMP: IR → PC
12	;	acc=0 → 11,17	JZRO:
13	racc,warg ;	19	ADD:ACC → ARG
14	racc,warg ;	22	SUB:ACC → ARG
15	rir,wmar ;	24	LD:IR → MAR
16	rir,wmar ;	25	ST:IR → MAR
17	rpc,inc ;	18	PC+1 → BUF
18	rbuf,wpc ;	5	BUF → PC
19	rir,wmar ;	20	IR → MAR
20	read,add ;	21	ARG + MEM(MAR) → BUF
21	rbuf, wacc ;	17	BUF → ACC
22	rir,wmar ;	23	IR → MAR
23	read,sub ;	21	ARG-MEM(ARG) → BUF
24	read,wacc ;	17	MEM(MAR) → ACC
25	racc,write;	17	ACC → MEM(MAR)

fetch

increment program counter

Fig. XVI Microinstruction in control units' *ROM*: the microprogram *mucode*

```
MICROCODE(r,w)
    λ{knob,button,ir,acc}.
     {rsw    = r(w) <28>,
      wmar   = r(w) <27>,
      memcntl = r(w) <25:26>,
      wpc    = r(w) <24>,
      rpc    = r(w) <23>,
      wacc   = r(w) <22>,
      racc   = r(w) <21>,
      wir    = r(w) <20>,
      rir    = r(w) <19>,
      warg   = r(w) <18>,
      alucntl = r(w) <16:17>,
      rbuf   = r(w) <15>,
      ready  = r(w) <14>,
      idle   = r(w) <13>},

    MICROCODE(r,(r(w) <0:2> = 1 ∧ button → r(w) <3:7>,
                 r(w) <0:2> = 2 ∧ acc=0  → r(w) <3:7>,
                 r(w) <0:2> = 3          → knob+r(w) <8:12>,
                 r(w) <0:2> = 4          → ir <13:15> + r(w) <8:12>,
                                           r(w) <8:12>))
```

Fig. XVII Semantics of Microcode

The behaviour defined in Fig. XVII specifies the control signals to be generated by the microcode. These signals are actually invoked by the control part of the host machine shown in Fig. XIV whose behaviour is defined by

$$CONTROL(r,w) = [\![ROM(r) \mid MPC(w) \mid DECODE]\!] \backslash mpc\ rom\ nextaddress$$

where the microprogram counter *MPC* is defined by:

$$MPC(w) = \lambda\{nextaddress\}.\{mpc=w\},\ MPC(nextaddress)$$

The *ROM* is defined by

$$ROM(r) = \lambda\{mpc\}.\{rom=r(mpc)\},\ ROM(r)$$

and the microinstruction decoder is a complicated combinational device defined by:

$$\begin{aligned}
&DECODE \\
&= \lambda\{rom, knob, button, acc, ir\}, \\
&\quad \{nextaddress = (rom\ \langle 0:2\rangle = 1 \wedge button \rightarrow rom\ \langle 3:7\rangle, \\
&\qquad\qquad rom\ \langle 0:2\rangle = 2 \wedge acc=0 \rightarrow rom\ \langle 3:7\rangle, \\
&\qquad\qquad rom\ \langle 0:2\rangle = 3 \rightarrow knob+rom\ \langle 8:12\rangle, \\
&\qquad\qquad rom\ \langle 0:2\rangle = 4 \rightarrow in\ \langle 13:15\rangle + rom\ \langle 8:12\rangle, \\
&\qquad\qquad\qquad rom\ \langle 8:12\rangle), \\
&\qquad rsw = rom\ \langle 28\rangle, \\
&\qquad wmar = rom\ \langle 27\rangle, \\
&\qquad memcntl = rom\ \langle 25:26\rangle, \\
&\qquad wpc = rom\ \langle 24\rangle, \\
&\qquad rpc = rom\ \langle 23\rangle, \\
&\qquad wacc = rom\ \langle 22\rangle, \\
&\qquad racc = rom\ \langle 21\rangle, \\
&\qquad wir = rom\ \langle 20\rangle, \\
&\qquad rir = rom\ \langle 19\rangle, \\
&\qquad warg = rom\ \langle 18\rangle, \\
&\qquad alucntl = rom\ \langle 16:17\rangle, \\
&\qquad rbuf = rom\ \langle 15\rangle, \\
&\qquad ready = rom\ \langle 14\rangle, \\
&\qquad idle = rom\ \langle 13\rangle\ \}, DECODE
\end{aligned}$$

It would be easy to implement *DECODE* as a composition of simpler combinational devices, but we shall not bother to do so.

It is straightforward to prove that *CONTROL* is correct by using the composition theorem to prove:

$$CONTROL = MICROCODE$$

We can at last define our implementation of the computer by:

$$HOST\ r\ (w, m, w_0, w_1, w_2, w_3, w_4, w_5) = [\![CONTROL(r, w) \mid DATA(m, w_0, w_1, w_2, w_3, w_4, w_5)]\!] \setminus L$$

where $L = \{rsw, wmarymemcntl, wpc, rpc, wacc, racc, wir, rir, warg, alucntl, rbuf, ir\}$

To prove this correct we must show:

$$COMPUTER(m, w_1, w_2) = HOST(0, mucode, m, w_0, w_1, w_2, w_3, w_4, w_5) \dagger P_{ready}$$

where *COMPUTER* is defined in Fig. XIII and $P_{ready}(s) = true$ if an only if $s(ready) = 1$.

The bulk of the proof consists in a completely routine, but extremely tedious, application of the composition theorem and simulation induction to show that

$$HOST(mucode) = \mathfrak{B}[HOSTMACHINE]$$

where *HOSTMACHINE* is the machine shown in Fig.XVIII. The proof of this, which we omit, is the kind of thing for which we need machine assistance.

The next step is to show that $HOSTMACHINE \dagger P_{ready}$ has the properties shown in Fig.XIX. This follows directly from Fig XVIII and Definition 12.

Finally, to complete the proof, we show from Fig. XIII and Fig.XIV by simulation induction that:

$$COMPUTER(m, w_1, w_2) = \mathfrak{B}[HOSTMACHINE \dagger P_{ready}](0, m, w_0, w_1, w_2, w_3, w_4, w_5)$$

$$EXECUTE(m, w_1, w_2) = \mathfrak{B}[HOSTMACHINE \dagger P_{ready}](5, m, w_0, w_1, w_2, w_3, w_4, w_5)$$

and hence

$$COMPUTER(m, w_1, w_2) = \mathfrak{B}[HOSTMACHINE](0, m, w_0, w_1, w_2, w_3, w_4, w_5) \dagger P_{ready}$$

$$= HOST\ mucode\ (0, m, w_0, w_1, w_2, w_3, w_4, w_5) \dagger P_{ready}$$

which establishes correctness.

$HOSTMACHINE = (S, out, next)$

$S = Word[5] \times Mem \times Word[13] \times Word[13] \times Word[16] \times Word[16] \times Word[16] \times Word[16]$

MPC, memory, MAR, PC, ACC, IR, ARG, BUF

$out(\{knob = k,\ button = b,\ switches = sw\},\ (w, m, w_0, w_1, w_2, w_3, w_4, w_5))$

$= \{pc = w_1, acc = w_2,\ ready = (w = 0 \vee w = 5 \to 1, 0),\ idle = (w = 0 \to 1, 0)\}$

$next(\{knob = k,\ button = b,\ switches = sw\}, (w, m, w_0, w_1, w_2, w_3, w_4, w_5))$

$=$

$(w = 0 \to ($	$($	$b \to 1, 0)$	m	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 1 \to ($		0	m	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 2 \to ($		0	m	w_0	$sw\langle 0{:}12\rangle$	w_2	w_3	w_4	w_5	$),$
$w = 3 \to ($		0	m	w_0	w_1	sw	w_3	w_4	w_5	$),$
$w = 4 \to ($		7	m	w_1	w_1	w_2	w_3	w_4	w_5	$),$
$w = 5 \to ($ | | $b \to 0, 6)$	m	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 6 \to ($		8	m	w_1	w_1	w_2	w_3	w_4	w_5	$),$
$w = 7 \to ($		0	$m[w_2/w_1]$	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 8 \to ($		9	m	w_0	w_1	w_2	$m(w_0)$	w_4	w_5	$),$
$w = 9 \to ($		$w_3\langle 13{:}15\rangle + 10$	m	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 10 \to ($		0	m	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 11 \to ($		5	m	w_0	$w_3\langle 0{:}12\rangle$	w_2	w_3	w_4	w_5	$),$
$w = 12 \to ($	$(w_2 = 0 \to 11, 17)$		m	w_0	w_1	w_2	w_3	w_4	w_5	$),$
$w = 13 \to ($		19	m	w_0	w_1	w_2	w_3	w_2	w_5	$),$
$w = 14 \to ($		22	m	w_0	w_1	w_2	w_3	w_2	w_5	$),$
$w = 15 \to ($		24	m	$w_3\langle 0{:}12\rangle$	w_1	w_2	w_3	w_4	w_5	$),$
$w = 16 \to ($		25	m	$w_3\langle 0{:}12\rangle$	w_1	w_2	w_3	w_4	w_5	$),$
$w = 17 \to ($		18	m	w_0	w_1	w_2	w_3	w_4	$w_1\langle 0{:}12\rangle + 1)$	$),$
$w = 18 \to ($		5	m	w_0	$w_5\langle 0{:}12\rangle$	w_2	w_3	w_4	w_5	$),$
$w = 19 \to ($		20	m	$w_3\langle 0{:}12\rangle$	w_1	w_2	w_3	w_4	w_5	$),$
$w = 20 \to ($		21	m	w_0	w_1	w_2	w_3	w_4	$w_4 + m(w_0)$	$),$
$w = 21 \to ($		17	m	w_0	w_1	w_5	w_3	w_4	w_5	$),$
$w = 22 \to ($		23	m	$w_3\langle 0{:}12\rangle$	w_1	w_2	w_3	w_4	w_5	$),$
$w = 23 \to ($		21	m	w_0	w_1	w_2	w_3	w_4	$w_4 - m(w_0)$	$),$
$w = 24 \to ($		17	m	w_0	w_1	$m(w_0)$	w_3	w_4	w_5	$),$
$w = 25 \to ($		17	$m[w_2/w_0]$	w_0	w_1	w_2	w_3	w_4	w_5	$))$

Fig. XVIII. *HOSTMACHINE*

$HOSTMACHINE \; +P_{ready} = (S, out, next')$

where S and out are as in Fig. XVIII, and hence:

$out(s, (0, m, w_0, w_1, w_2, w_3, w_4, w_5)) = \{pc=w_1, acc=w_2, ready=1, idle=1\}$

$out(s, (5, m, w_0, w_1, w_2, w_3, w_4, w_5)) = \{pc=w_1, acc=w_2, ready=1, idle=0\}$

and

$$
\begin{aligned}
&next'(\{knob=k, button\ b, switches=sw\}, (0, m, w_0, w_1, w_2, w_3, w_4, w_5)) \\
&\quad (\neg b \rightarrow (0, m, w_0, w_1, w_2, w_3, w_4, w_5), \\
&\quad\ b \rightarrow (k=1 \rightarrow (0, m, w_0, sw\langle 0:12\rangle, w_2, w_3, w_4, w_5), \\
&\qquad\ k=2 \rightarrow (0, m, w_0, w_1, sw, w_3, w_4, w_5), \\
&\qquad\ k=3 \rightarrow (0, m[w_2/w_1], w_0, w_1, w_2, w_3, w_4, w_5), \\
&\qquad\ k=4 \rightarrow (5, m, w_0, w_1, w_2, w_3, w_4, w_5))
\end{aligned}
$$

$$
\begin{aligned}
&next'(\{knob=k, button=b, switches=sw\}, (5, m, w_0, w_1, w_2, w_3, w_4, w_5)) \\
&\quad = \text{let} \ \{op=m(w_1)\langle 13:15\rangle, \ addr=m(w_1)\langle 0:12\rangle\} \\
&\quad\ \text{in} \ (b \rightarrow (0, m, w_0, w_1, w_2, w_3, w_4, w_5), \\
&\qquad\ \neg b \rightarrow (op=0 \rightarrow (0, m, w_1, w_1, w_2, m(w_1), w_4, w_5), \\
&\qquad\qquad\ op=1 \rightarrow (5, m, w_1, addr, w_2, m(w_1), w_4, w_5), \\
&\qquad\qquad\ op=2 \rightarrow (w_2=0 \rightarrow (5, m, w_1, addr, w_2, m(w_1), w_4, w_5), \\
&\qquad\qquad\qquad\qquad (5, m, w_1, w_1+1, w_2, m(w_1), w_4, w_1\langle 0:15\rangle+1)), \\
&\qquad\qquad\ op=3 \rightarrow (5, m, addr, w_1+1, w_2+m(addr), m(w_1), w_2, w_1\langle 0:15\rangle+1), \\
&\qquad\qquad\ op=4 \rightarrow (5, m, addr, w_1+1, w_2-m(addr), m(w_1), w_2, w_1\langle 0:15\rangle+1), \\
&\qquad\qquad\ op=5 \rightarrow (5, m, addr, w_1+1, m(addr), m(w_1), w_4, w_1\langle 0:15\rangle+1), \\
&\qquad\qquad\ op=6 \rightarrow (5, m[w_2/addr], addr, w_1+1, w_2, m(w_1), m_4, w_1\langle 0:15\rangle+1), \\
&\qquad\qquad\ op=7 \rightarrow (5, m, w_0, w_1+1, w_2, m(w_1), w_4, w_1\langle 0:15\rangle+1)))
\end{aligned}
$$

Fig. XIX Properties of $HOSTMACHINE + P_{ready}$

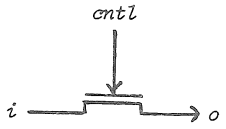
Second case study: correctness of nMOS systems

In this case study we illustrate how our model of register transfer systems might be used to analyse the kind of algorithms which are directly implemented in nMOS chips. It is important to be clear that we are not trying to model the implementations themselves, but only their idealized behaviour. This behaviour is designed to be as simple as possible for the representation

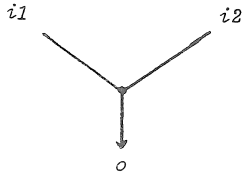
and verification (by proof) of functional correctness; it is not intended to be suitable for modelling electrical properties.

We will build devices as compositions of the following four primitives:

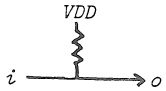
1. Gates



2. Joins



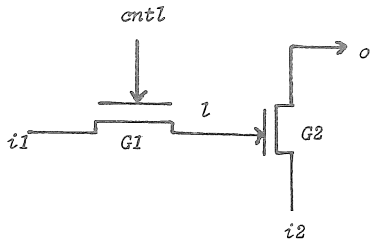
3. Pullups



4. Ground

A good elementary tutorial account of these is given in [2]. We will model them by sequential behaviours whose lines carry three possible values: high which we denote by *true* or *1*, low which we denote by *false* or *0* and null which we denote by $\emptyset$. The null value, which we also used in the previous case study, is a trick; it is the value output by a gate that is off and roughly corresponds to 'floating' or 'high impedance' - but this interpretation must not be taken too seriously.

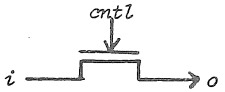
The null value is needed to model the capacitive effects of circuits like



Here the output of gate *G1* is connected to the control of gate *G2*. Normally a gate conducts (is "on") if its control line is high, so in the device above if the value on *cntl* is high then the value on the control of *G2* - i.e. on line *l* - will be the value input on *i1*. Now if *cntl* goes low then *G1* stops conducting (goes "off") and the charge (or lack of charge) on line *l* will be trapped and will continue to control the behaviour of *G2* until it decays away. For example, if both *cntl* and *i1* are high then both *G1* and *G2* will be on; now if on the next cycle *cntl* goes low, then the positive charge on *l* will be trapped and *G2* will continue to conduct. Similarly if *i1* had been low then when *cntl* went low *G2* would continue to be off. We shall assume decay times are just one cycle long. This is rather pessimistic, but it is 'safe' and enables us to give an interesting discussion of refreshing; longer decay times are easy to model, as we shall show.

The way we model the behaviour just described is to represent gates as devices with memory, such that the 'effective' control value is the value on the control input if this is not null, but if the control input is null, then the stored value is the effective control value. If the effective control value is *1* then the gate outputs its input otherwise it outputs $\oplus$.

Thus a gate



has behaviour defined by:

$$G(t) = \lambda\{i, cntl\}.\{o=(cntl=1 \vee (cntl=\oplus \wedge t=1) \rightarrow i,\ \oplus\ )\},\ G(cntl)$$

This assumes a decay time of one cycle. The behaviour of a gate with infinite decay time - i.e. the charge trapped on the control remains as long as the control stays isolated - is simply:

$$G(t) = \lambda\{i, cntl\}.\{o=(cntl=1 \vee (cntl=\oplus \wedge t=1) \rightarrow i,\ \oplus\ )\},\ G(cntl=\oplus \rightarrow t, cntl)$$

We shall assume one cycle decay time from now on - i.e. the first definition of *G* will be used.

In what follows it will simplify calculations if we allow ourselves to restrict certain lines to only carry non-null values. Doing this is a special case of using a slightly generalised in which lines are typed. In this generalised model we assume each line $l \in Line$ has an associated domain $Dom[l]$. For example, we could assume that for all $l \in Line$ that $Dom[l] = Val$ - this would correspond to what we have been doing up until now, i.e. to the ungeneralised model. If we assign domains to lines we must modify the definition of $Sig[X]$ to be

$$Sig[X] = \{s \mid \text{for all } l \in X : \ s(l) \in Dom[l]\}$$

Since $Com[X;Y]$ and $Seq[X;Y]$ are defined in terms of $Sig[X]$ and $Sig[Y]$ we must modify them too - all we need to do is plug the new $Sig[X]$ and $Sig[Y]$ into Definitions 2 and 3.

When writing down behaviours we must be sure they are 'well-typed' - e.g. in expressions of the form $\{y = E_y \mid y \in Y\}$ the value of E_y must be in $Dom[Y]$. If $Dom[l] = D$ then we will write $l:D$. In fact we only need two types of lines. Let $Bool^+$ be the domain containing $1,0$ and $\emptyset$ and $Bool$ be the usual domain containing just 1 and 0. For this case study we shall only use lines l such that either $l:Bool$ or $l:Bool^+$, the latter being the default. Thus unless we explixitly declare that $l:Bool$ assume that $l:Bool^+$ - i.e. that l might carry $\emptyset$.

Recall now the behaviour of a gate (with one cycle delay time):

$$G(t) = \lambda\{i,cntl\}.\{o = (cntl = 1 \vee (cntl = \emptyset \wedge t = 1) \rightarrow i, \emptyset)\},G(cntl)$$

If we declare $cntl:Bool$ then the case $cntl = \emptyset$ becomes impossible and the definition simplifies to

$$G(t) = \lambda\{i,cntl\}.\{o = (cntl \rightarrow i, \emptyset)\},G(cntl)$$

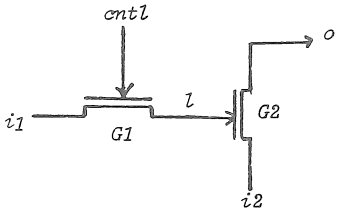
which, by simulation induction, is equivalent to the purely combinational G defined by:

$$G = \lambda\{i,cntl\}.\{o = (cntl \rightarrow i, \emptyset)\},G$$

Notice that we cannot declare $o:Bool$ for this example because $G(t)(\{i = x, cntl = 0\}) = \{o = \emptyset\}$. To make our notation really safe and rigorous we need to specify type checking rules for it.

Example

If we assume $i1, cntl : Bool$ then the behaviour of the circuit :



is given by:

$$B(t) = [\![G1 \mid G2(t)]\!] \setminus l$$

where $G1 = \lambda\{i1, cntl\}.\{l = (cntl \to i1, \Phi)\}, G1$

$$G2(t) = \lambda\{i2, l\}.\{o = (l = 1 \vee (l = \Phi \wedge t = 1) \to i2, \Phi)\}, G2(l)$$

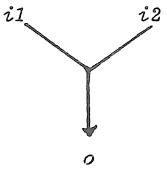
Then by the composition theorem

$$B(t) = \lambda\{i1, i2, cntl\}.$$
$$\text{letrec } \{l = (cntl \to i1, \Phi)\}$$
$$\text{in } \{o = (l = 1 \vee (l = \Phi \wedge t = 1) \to i2, \Phi)\}, B(l)$$

$$= \lambda\{i1, i2, cntl\}.$$
$$\{o = ((cntl \wedge i1) \vee (\neg cntl \wedge t = 1) \to i2, \Phi)\}, B(cntl \to i1, \Phi)$$

All the other primitives besides gates are purely combinational.

A join J



is defined by

$$J = \lambda\{i1, i2\}.\{o = (i1 = \Phi \to i2, (i2 = \Phi \to i1, i1 \vee i2))\}, J$$

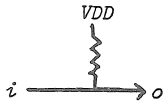
Thus joining null and a value gives the value and joining two non null values gives their disjunction. If we extend $\vee$ by defining

$$\Phi \vee x = x \vee \Phi = x$$

Then the definition of J becomes:

$J = \lambda\{i1, i2\}.\{o = i1 \vee i2\}, J$

A pullup *PU*



has behaviour:

$PU = \lambda\{i\}.\{o = (i = 0 \rightarrow 0, 1)\}, PU$

Thus a pullup transmits non-null values unchanged, and pulls $\oplus$ up to *1*. This is our version of Ohm's Law!

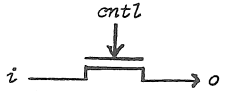
The final primitive ground *GND*

has behaviour:

$GND = \lambda\{\}.\{o = 0\}, GND$

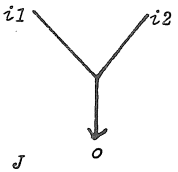
We summarise our nMOS primitives in Fig. XX

Gate *G(t)*



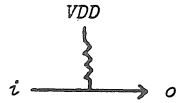
$G(t) = \lambda\{i, cntl\}.\{o = \{cntl = 1 \vee (cntl = \oplus \wedge t = 1) \rightarrow i, \oplus\)\}, G(cntl)$

Join *J*



$J = \lambda\{i1, i2\}.\{o = i1 \vee i2\}, J$

Pullup *PU*



$PU = \lambda\{i\}.\{o = (i = 0 \rightarrow 0, 1)\}, PU$

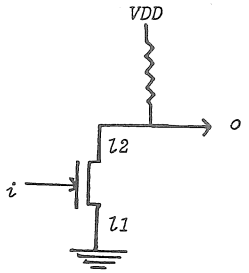
Ground *GND*

$GND = \lambda\{\}.\{o = 0\}, GND$

Fig XX. nMOS primitives and their behaviour

Example 12.

A conventional nMOS invertor implementation is:



within our model the behaviour of this is:

$$NOT(t) = [\![G(t) \mid GND \mid PU]\!] \backslash l1\ l2$$

where: $G(t) = \lambda\{l1,i\}.\{l2=(i=1 \vee (i=\oplus \wedge t=1) \rightarrow l1, \oplus)\},\ G(i)$

$GND = \lambda\{\}.\{l1=0\},\ GND$

$PU = \lambda\{l2\}.\{o=(l2=0 \rightarrow 0,1)\},\ PU$

and hence by the composition theorem and some 3-valued boolean algebra:

$NOT(t)$

$= \lambda\{i\}.$

letrec $\{l1=0,\ l2=(i=1 \vee (i=\oplus \wedge t=1) \rightarrow l1, \oplus)\}$

in $\{o=(l2=0 \rightarrow 0,1)\},\ NOT(i)$

$= \lambda\{i\}.\{o=(i=1 \vee (i=\oplus \wedge t=1) \rightarrow 0,1)\},\ NOT(i)$

If we extend $\neg$ from $Bool$ to $Bool^{+}$ by defining:

$\neg\oplus = 1$

then the inverter's behaviour can be written as:

$NOT(t) = \lambda\{i\}.\{o=(i=\oplus \rightarrow \neg t,\ \neg i)\},\ NOT(i)$

Thus for non-null inputs the input is just inverted, and when null is input the negation of the value stored is output; the value stored is the value on the input line i during the previous cycle.

If we declare $i:Bool$ (i.e. only use the inverter in a context where non-null values are input) then the behaviour simplifies to

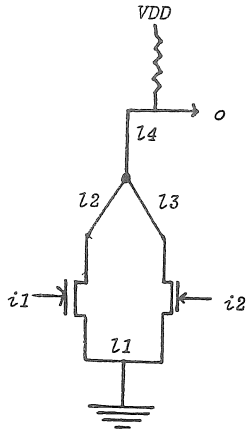
$NOT(t) = \lambda\{i\}.\{o=\neg i\},\ NOT(i)$

which by simulation induction is equal to NOT defined by

$NOT = \lambda\{i\}.\{o=\neg i\},\ NOT$

Example 13

The standard nMOS implementation of a *NOR* element is:



In our model the behaviour of this is given by:

$NOR(t_1,t_2) = [\![G1(t_1)|G2(t_2)|GND|J|PU]\!]\backslash l1\ l2\ l3\ l4$

where $G1(t_1) = \lambda\{l1,i1\}.\{l2=(i1=1 \vee (i1=\oplus \wedge t_1=1) \rightarrow l1,\oplus)\},\ G1(i1)$

$G2(t_2) = \lambda\{l1,i2\}.\{l3=(i2=1 \vee (i2=\oplus \wedge t_2=1) \rightarrow l1,\oplus)\},\ G2(i2)$

$J = \lambda\{l2,l3\}.\{l4=l2 \vee l3\},\ J$

$PU = \lambda\{l4\}.\{o=(l4=0 \rightarrow 0,1)\},\ PU$

$GND = \lambda\{\}.\{l1=0\},\ GND$

Hence by the composition theorem and some boolean algebra using the extended meanings of $\neg$ and $\vee$ on $Bool^+$.

$NOR(t_1,t_2)$

$= \lambda\{i1,i2\}.$

letrec{ $l1=0,$

$l2=(i1=1 \vee (i1=\oplus \wedge t_1=1) \rightarrow l1,\oplus),$

$l3=(l2=1 \vee (i2=\oplus \wedge t_2=1) \rightarrow l1,\oplus),$

$l4=l2 \vee l3$}

in{ $o=(l4=0 \rightarrow 0,1)$}, $NOR(i1,i2)$

$= \lambda\{i1,i2\}.\{o=\neg((i1=\oplus \rightarrow t_1,i1) \vee (i2=\oplus \rightarrow t_2,i2))\},\ NOR(i1,i2)$

If we declare $i1,i2:Bool$ then this simplifies to

$NOR(t_1,t_2) = \lambda\{i1,i2\}.\{o=\neg(i1 \vee i2)\},\ NOR(i1,i2)$

which by simulation induction is equivalent to *NOR* defined by:

$NOR = \lambda\{i1,i2\}.\{o=\neg(i1 \vee i2)\},\ NOR$

In the stack controller which we analyse later we will use *NOR* gates in a context in which one of the inputs can never have a null value, but not necessarily the other one. If $i1:Bool$ then the behaviour simplifies to

$$NOR(t) = \lambda\{i1,i2\}.\{o= \neg(i1 \vee (i2=\oplus \rightarrow t,i2))\},\ NOR(i2)$$

We now analyse the stack cell shown in Fig.XXI which is taken from Mead and Conway [*10*] Plate 6.

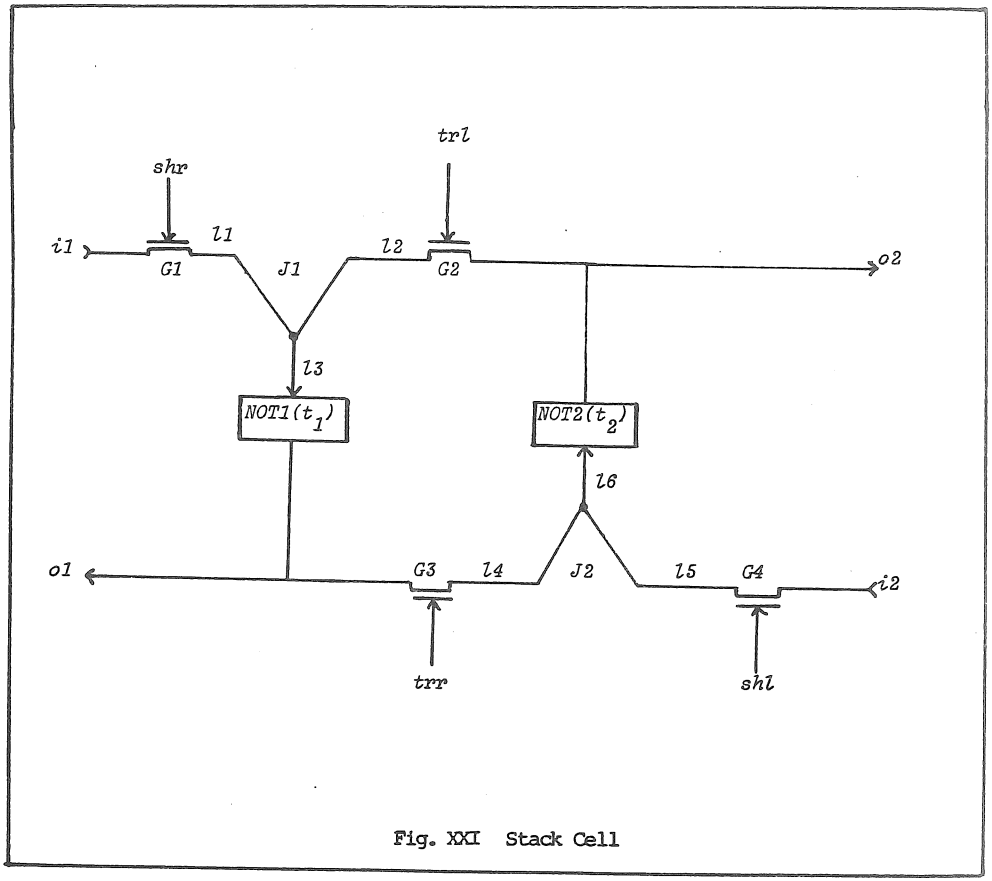


Fig. XXI Stack Cell

We assume that $shr, trl, trr, shl, i1, i2 : Bool$, and hence the behaviour of the stack cell is given by:

$$STACKCELL(t_1, t_2) = [\![G1 | G2 | G3 | G4 | J1 | J2 | NOT1(t_1) | NOT2(t_2)]\!] \setminus l1\ l2\ l3\ l4\ l5\ l6$$

where
$$G_1 = \lambda\{i1, shr\}.\{l1 = (shr \to i1, \emptyset)\}, G1$$
$$G_2 = \lambda\{o2, trl\}.\{l2 = (trl \to o2, \emptyset)\}, G2$$
$$G_3 = \lambda\{o1, trr\}.\{l4 = (trr \to o1, \emptyset)\}, G3$$
$$G_4 = \lambda\{i2, shl\}.\{l5 = (shl \to i2, \emptyset)\}, G4$$
$$J1 = \lambda\{l1, l2\}.\{l3 = l1 \vee l2\}, J1$$
$$J2 = \lambda\{l4, l5\}.\{l6 = l4 \vee l5\}, J2$$
$$NOT1(t_1) = \lambda\{l3\}.\{o1 = (l3 = \emptyset \to \neg t_1, \neg l3)\}, NOT1(l3)$$
$$NOT2(t_2) = \lambda\{l6\}.\{o2 = (l6 = \emptyset \to \neg t_2, \neg l6)\}, NOT2(l6)$$

By the composition Theorem we can easily show that the behaviour of the stack cell is as show in Fig. XXII

$$\begin{aligned}
&STACKCELL(t_1, t_2) \\
&= \lambda\{shr, trl, trr, shl, i1, i2\}. \\
&\quad \text{letrec } \{l3 = (shr \to i1, \emptyset) \vee (trl \to o2, \emptyset), \\
&\qquad l6 = (trr \to o1, \emptyset) \vee (shl \to i2, \emptyset), \\
&\qquad o1 = (l3 = \emptyset \to \neg t_1, \neg l3), \\
&\qquad o2 = (l6 = \emptyset \to \neg t_2, \neg l6)\} \\
&\quad \text{in } \{o1 = (l3 = \emptyset \to \neg t_1, \neg l3), o2 = (l6 = \emptyset \to \neg t_2, \neg l6)\}, STACKCELL(l3, l6)
\end{aligned}$$

Fig. XXII Behaviour of stack cell

The controller, which we shall use to drive the stack cell (and which is described below), will always drive exactly one of the control lines shr, trl, trr, shl high at a time. If we assume $t_1, t_2 : Bool$ then we can derive from Fig. XXII

$STACKCELL(t_1, t_2)$

$\lambda\{shr, trl, trr, shl, i1, i2\}.$

$(shr = 1 \wedge trl = 0 \wedge trr = 0 \wedge shl = 0 \rightarrow (\{o1 = \neg i1, o2 = \neg t_2\}, STACKCELL(i1, \Phi)),$

$shr = 0 \wedge trl = 1 \wedge trr = 0 \wedge shl = 0 \rightarrow (\{o1 = t_2, o2 = \neg t_2\}, STACKCELL(\neg t_2, \Phi)),$

$shr = 0 \wedge trl = 0 \wedge trr = 1 \wedge shl = 0 \rightarrow (\{o1 = \neg t_1, o2 = t_1\}, STACKCELL(\Phi, \neg t_1)),$

$shr = 0 \wedge trl = 0 \wedge trr = 0 \wedge shl = 1 \rightarrow (\{o1 = \neg t_1, o2 = \neg i2\}, STACKCELL(\Phi, i2)),$

$shr = 0 \wedge trl = 0 \wedge trr = 0 \wedge shl = 0 \rightarrow (\{o1 = \neg t_1, o2 = \neg t2\}, STACKCELL(\Phi, \Phi)),$

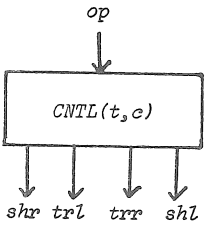
$\vdots$

Notice that if all the control lines are kept low during a cycle then the values stored on both gates decay to null. To avoid this the controller must constantly refresh the data. The control scheme described by Mead and Conway is based on a two phase clock (as are all their nMOS algorithms). During the first phase (phase 1) either *shr* or *trl* is driven high and during the second phase (phase 2) either *trr* or *shl* is driven high. The idea is that during phase 1 the value stored in *NOT1* (i.e. t_1) is refreshed or updated whilst the value stored in *NOT2* decays to Φ, and during phase 2 the value stored is *NOT1* (i.e. t_2) is refreshed or updated, whilst the value stored in *NOT1* decays to Φ. More specifically we see from the expression for $STACKCELL(t_1, t_2)$ just derived that:

1. Driving *shr* high stores in *NOT1* the value input on *i1* - i.e. does a right shift.
2. Driving *trl* high stores in *NOT1* the negation of the value in *NOT2* - i.e. does a left recirculate.
3. Driving *trr* high stores in *NOT2* the negation of the value in *NOT1* - i.e. does a right recirculate.
4. Driving *shl* high stores in *NOT2* the value input on *i2* - i.e. does a left shift.

Thus when not shifting one must refresh by recirculating the data inside the stack cell by alternatively driving trl and trr high.

In order to enable the stack cell to be driven from a single line Mead and Conway describe a controller ([10] page 73) which we will model with a device:



where $t:Bool$ is the control state and $c:Bool$ is the clock state and:

$$CNTL(t,c)$$
$$= \lambda\{op\}.\{trr = (c \to 0, \neg t), shl = (c \to 0, t), trl = (c \to \neg t, 0), shr = (c \to t, 0)\}, CNTL(op, \neg c)$$
$$= \lambda\{op\}.$$
$$(c \wedge t \to \{shr = 1, trl = 0, trr = 0, shl = 0\},$$
$$c \wedge \neg t \to \{shr = 0, trl = 1, trr = 0, shl = 0\},$$
$$\neg c \wedge \neg t \to \{shr = 0, trl = 0, trr = 1, shl = 0\},$$
$$\neg c \wedge t \to \{shr = 0, trl = 0, trr = 0, shl = 1\}), CNTL(op, \neg c)$$

The idea is that $c = 0$ during phase 1 and $c=1$ during phase 2; t, which is the value input on op during the previous cycle, determines whether to shift or recirculate.

If we combine the controller $CNTL$ with a stack cell the resulting system has behaviour:

$$STACKSYS(t_1, t_2, t, c) = [[STACKCELL(t_1, t_2) \mid CNTL(t, c)]] \backslash shr\ trl\ trr\ shl$$

and we can then show:

$$
\begin{aligned}
&STACKSYS(t_1, t_2, t, c) \\
&\quad = \lambda\{op, i1, i2\}. \\
&\qquad (c \wedge t \rightarrow (\{o1 = \neg i1, o2 = \neg t_2\}, STACKSYS(i1, @, op, 0)), \\
&\qquad\ c \wedge \neg t \rightarrow (\{o1 = t_2, o2 = \neg t_2\}, STACKSYS(\neg t_2, @, op, 0)), \\
&\qquad \neg c \wedge \neg t \rightarrow (\{o1 = \neg t_1, o2 = t_1\}, STACKSYS(@, \neg t_1, op, 1)), \\
&\qquad \neg c \wedge\ t\ \ (\{o1 = \neg t_1, o2 = \neg i2\}, STACKSYS(@, i2, op, 1)).
\end{aligned}
$$

From this we see that if *op* is driven low during phase 1 then during the next phase 2 a left recirculate will occur, and if *op* is driven low during phase 2 then during the next phase 1 a right recirculate will occur. Thus a constant stream of *0*'s on *op* keeps the data recirculating in the stack cell so that it does not decay away. We can also see that to shift in the value on the *i1* line during phase 1 one must drive *op* high during the preceding phase 2, and to shift in the value on the *i2* line during phase 1 one must drive *op* high during the preceding phase 2.

We thus see that a controller with behaviour $CNTL(t, c)$ is correct; the implementation of this behaviour given in Mead and Conway is shown in Fig XXIII.

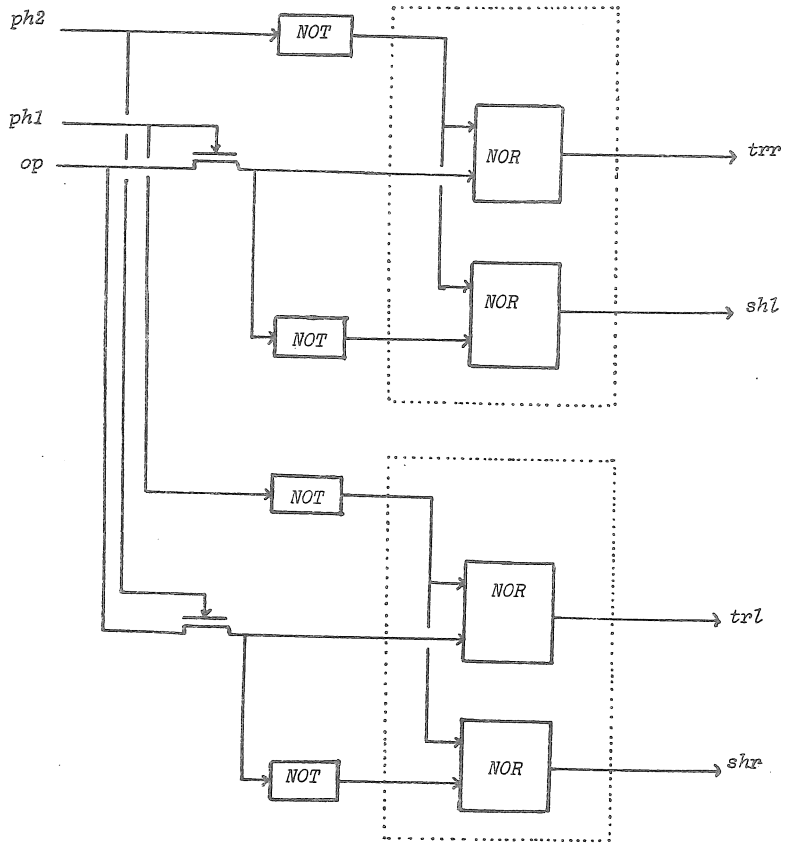
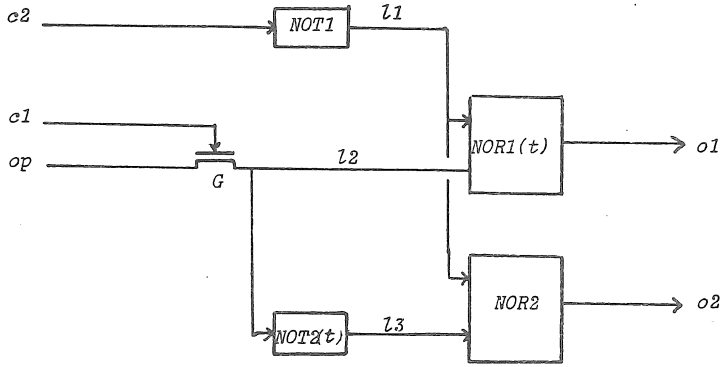


Fig. XXIII Implementation of the stack controller

The lines *ph1,ph2:Bool* are connected to the clock. If we also declare *op:Bool* then the only lines in Fig.XXIII that will carry $\mathcal{O}$ are the outputs of the two gates. We can thus use simplified behaviours for all the gates and half the *NOT*'s and *NOR*'s.

Note also that the two subsystems inside the dotted line boxes are identical. Each subsystem (in view of the remarks above about which lines carry $\mathcal{O}$) has a behaviour *CON(t)* defined by the device:



Thus:

$$CON(t) = [\![NOT1 \mid G \mid NOR2 \mid NOR1(t) \mid NOT1(t)]\!] \setminus l1\ l2\ l3$$

$$NOT1 = \lambda\{c2\}.\{l1 = \neg c2\}, NOT1$$

$$G = \lambda\{op, c1\}.\{l2 = (c1 \to op, \mathcal{O})\}, G$$

$$NOR2 = \lambda\{l1, l3\}.\{o2 = \neg(l1 \vee l3)\}, NOR2$$

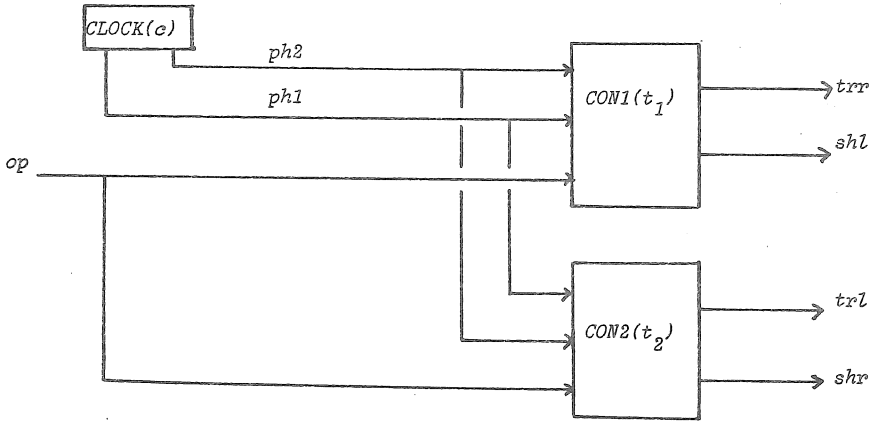
$$NOR1(t) = \lambda\{l1, l2\}.\{o1 = \neg(l1 \vee (l2 = \mathcal{O} \to t, l2))\}, NOR1(l2)$$

$$NOT2(t) = \lambda\{l2\}. \quad \{l3 = (l2 = \mathcal{O} \to \neg t, \neg l2)\}, NOT2(l2)$$

By the composition Theorem

$$
\begin{aligned}
&CON(t)\\
&= \lambda\{op, c1, c2\}.\\
&\quad \text{letrec } \{l1 = \neg c2,\ l2 = (c1 \to op, @),\ l3 = (l2 = @ \to \neg t, \neg l2)\}\\
&\quad \text{in} \quad \{o1 = \neg(l1 \vee (l2 = @ \to t, l2)),\ o2 = \neg(l1 \vee l3)\}, CON(l2)\\
&= \lambda\{op, c1, c2\}.\\
&\quad \{o1 = \neg(\neg c2 \vee (\neg c1 \to t, op)),\ o2 = \neg(\neg c2 \vee (\neg c1 \to \neg t, \neg op))\}, CON(c1 \to op, @)\\
&= \lambda\{op, c1, c2\}.\\
&\quad \{o1 = c2 \wedge (c1 \to \neg op, \neg t),\ o2 = c2 \wedge (c1 \to op, t)\}, CON(c1 \to op, @)
\end{aligned}
$$

The complete controller can now be assembled from two subsystems:



The behaviour of the complete controller is thus:

$$CONTROL(t_1, t_2, c) = [\![CON1(t_1) \mid CON2(t_2) \mid CLOCK(c)]\!] \backslash ph1\ ph2$$

where $CON1(t_1) = \lambda\{op, ph1, ph2\}.$
$\{trr = ph2 \wedge (ph1 \rightarrow \neg op, \neg t_1), shl = ph2 \wedge (ph1 \rightarrow op, t_1)\},$
$CON1(ph1 \rightarrow op, @)$

and $CON2(t_2) = \lambda\{op, ph1, ph2\}.$
$\{trl = ph1 \wedge (ph2 \rightarrow \neg op, \neg t_2), shr = ph1 \wedge (ph2 \rightarrow op, t_2)\},$
$CON2(ph2 \rightarrow op, @)$

and $CLOCK(c) = \lambda\{\}.\ \{ph1 = c, ph2 = \neg c\}, CLOCK(\neg c)$

From which it follows that:

$CONTROL(t_1, t_2, c)$
$= \lambda\{op\}$
letrec $\{ph1 = c, ph2 = \neg c\}$
in $\{trr = ph2 \wedge (ph1 \rightarrow \neg op, \neg t_1),$
$shl = ph2 \wedge (ph1 \rightarrow op, t_1),$
$trl = ph1 \wedge (ph2 \rightarrow \neg op, \neg t_2),$
$shr = ph1 \wedge (ph2 \rightarrow op, t_2)\},$
$CONTROL((ph1 \rightarrow op, @), (ph2 \rightarrow op, @), \neg c)$

and hence

$CONTROL(t_1, t_2, c)$
$= \lambda\{op\},$
$\{trr = (c \rightarrow 0, \neg t_1), shl = (c \rightarrow 0, t_1),\ trl = (c \rightarrow \neg t_2, 0), shr = (c \rightarrow t_2, 0)\},$
$CONTROL((c \rightarrow op, @), (c \rightarrow @, op), \neg c)$

Thus during phase 1 $(c=0)$ $\emptyset$ is stored in *CON1* and *op* in *CON2*, and during phase 2 $(c=1)$ *op* is stored in *CON1* and $\emptyset$ in *CON2*. Thus, in general, for some t the value stored in *CON1* is $(c \to \emptyset, t)$ and in *CON2* is $(c \to t, \emptyset)$ and thus to show *CONTROL* is correct we must show:

$$CNTL(t,c) = CONTROL((c \to \emptyset, t), (c \to t, \emptyset), c)$$

which follows easily by simulation induction.

This completes the verification of both the stack cell and the stack controller. Although our model of nMOS is very, very simple, we hope this case study has shown that nevertheless it enables us to make a non-trivial analysis of functional behaviour.

Conclusions

We have tried to show that, by modelling register transfer systems with sequential behaviours, we can express and reason about both specifications and implementations, in a clean and uniform way, at many different levels of abstraction. What we have not shown is whether our techniques can be scaled up to 'real' examples. This, we feel, is the crucial test. We hope that by splitting up behaviours into compositions of simpler ones (as in the case studies) we can reduce large problems to collections of small ones. One of the goals of our future work is to test this hypothesis by attempting to model and verify larger and larger systems. We believe machine assistance is essential for this, and we have already done some experiments in compiling behaviour definitions into executable code useful for simulation. However, although simulation can be a powerful tool for verification, we are still pursuing the more rigorous goal of verification by proof. To this end we plan to experiment with using an interactive metalanguage to control manipulations of expressions denoting behaviours. This methodology is based on a similar one used with some success in the LCF project [4].

Acknowledgements

Most of my theoretical ideas derive from the work of Robin Milner. Sequential behaviours were first used by him [7], although for a different purpose. The idea of representing two dimensional diagrams by one dimensional expressions using composition and restriction comes from [6]. More recently CCS[8] has set a

standard of "articulacy" - i.e. expressive and manipulative fluency - which I have tried to copy (e.g. the composition theorem is inspired by CCS's Expansion theorem). Finally there is some hope that my model might just be a special case of the new synchronous CCS [9].

I have had a number of useful conversations with Luca Cardelli, especially concerning his interesting language ANALOG[1], and with Matthew Hennessy on the relation between synchrony and asynchrony. I have also benefited from talking to Jacek Leszczylowski and Igor Hansen about their work on microcode description and analysis [3].

The work reported here was partly supported by the University of Southern California's Information Sciences Institute. Whilst I was there I had valuable interactions with Steve Crocker and his microcode verification group. For the rest of the time I was supported by an SRC Advanced Fellowship.

Finally, I would like to especially thank Dorothy McKie and Gina Temple for successfully completing the heroic task of typing this particularly difficult manuscript.

References

[1] L. Cardelli, "Analog Processes", Proc. 9th MFCS Symposium, Poland, Springer Verlag LNCS 88, 1980.

[2] W.A. Clark, "From Electron Mobility to Logical Structure: A View of Integrated Circuits", A.C.M. Computing Surveys, Vol. 12, No. 3, September 1980.

[3] P. Dembinski, S. Budkowski, I. Hansen, J. Leszczylowski, A. Paplinski, "Verification Oriented Microprogramming Language", Technical Report 293, Institute of Computer Science, Polish Academy of Sciences, Warsaw, 1977.

[4] M. Gordon, R. Milner, C. Wadsworth, "Edinburgh LCF: A Mechanised Logic of Computation", Springer Verlag LNCS 78, 1979.

[5] M. Gordon, "The Denotational Semantics of Sequential Machines", Information Processing Letters, Vol. 10, No. 1, 1980.

[6] G. Milne, R. Milner, "Concurrent Processes and their Syntax", J.A.C.M., 26,2, 1979.

[7] R. Milner, "Processes: A Mathematical Model of Computing Agents", Proc. Logic Coll. '73, ed. Rose and Shepherdson, North Holland, 1973.

[8] R. Milner, "A Calculus of Communicating Systems", Springer Verlag LNCS 9., 1980.

[9] R. Milner, "On relating Synchrony and Asychrony", Department of Computer Science Internal Report CSR-75-80, University of Edinburgh, 1980.

[10] C. Mead, L. Conway, "Introduction to VLSI Systems", Addison-Wesley, 1980.

[11] D. Scott, "Data Types as Lattices", SIAM Journal of Computing, Vol. 5, No. 3, 1976.