μFP - AN ALGEBRAIC VLSI DESIGN LANGUAGE

Mary Sheeran

Oxford University
Computing Laboratory
Programming Research Group-Library
8-11 Keble Road
Oxford OX1 3QD
Oxford (0865) 54141

Technical Monograph PRG-39

November 1983
(published as a monograph September 1984)

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Oxford University Computing Laboratory
Programming Research Group
8 - 11 Keble Road
Oxford OX1 3QD

A thesis submitted for the degree of
Doctor of Philosophy in the University of Oxford,
November 1983

μFP, an Algebraic VLSI Design Language

Mary Sheeran. St. Cross College.

A thesis submitted for the degree of Doctor of Philosophy in the University of Oxford, Michaelmas Term, 1983.

Abstract

VLSI (Very Large Scale Integration) allows us to fit circuits of enormous complexity onto one chip. If we are to design successful circuits, we must find design methods which allow us to contain this complexity. One of the most vital components of such a design method is a language in which we can express our ideas and our design decisions.

In this thesis, we propose an IC design language, μFP, which can describe both the semantics (or behaviour) of a circuit and its layout. μFP is a variant of the Functional Programming language, FP. It is designed to allow the designer to reason about his circuit descriptions, by manipulating the descriptions themselves. The descriptions are just expressions, made from a small number of primitive functions and combining forms (functionals that map functions into functions). These functions and combining forms were chosen because they have nice algebraic properties. Also, each combining form has a simple geometric interpretation, so that every μFP expression has an associated floor-plan.

We give a formal semantics of μFP and we show how algebraic identities may be proved in the language. We illustrate the use of μFP in several examples, ranging from a combinatorial tally circuit to a systolic correlator.

Acknowledgements

I would like to thank my supervisor, Peter Henderson. His help and encouragement were invaluable to me throughout the project. He introduced me to VLSI and to functional programming, and this is the result. His work has influenced me greatly. I would also like to thank Bernard Sufrin, who gave encouragement when it was much needed. John Hughes, who has since become my husband, unwittingly introduced me to FP, when I overheard him explaining it to a friend. He also acted as my mathematical advisor when I first tried to describe μFP formally. More importantly, he managed to keep me calm at times when I found doing a D. Phil to be a great strain.

I have found the Programming Research Group to be a very stimulating place in which to work and I would like to thank Joe Morris, who persuaded me to come here in the first place.

I am very grateful to the Pirie-Reid fund who have supported me financially during the past three years. Without their scholarship, I would never have been able to come to Oxford. Finally, I would like to thank my parents, whose quiet encouragement has been a constant source of strength.

Contents

Chapter 1: Introduction

VLSI (Very Large Scale Integration) allows us to fit circuits of enormous complexity onto one chip. In order to design successful circuits, we must find design methods which allow us to contain this complexity. One of the most vital components of any such design method (whether or not it is formal) is a language in which we can express our ideas and our design decisions. Without an adequate design language, it is difficult for the designer to communicate his ideas to others, and especially to people who are not expert in digital design. It is also difficult for the "customer" to specify what he wants the chip to do and for the designer to check that his design actually does what he thinks it does. The analogy with the problems of designing computer programs is clear. Formal methods are now beginning to come to the rescue in the "software crisis". If we are to avoid the "VLSI crisis", we must also be willing to use formal, or at least semi-formal methods.

In software, the usefulness of high level languages has long been appreciated. In VLSI design, however, most of the available tools tackle the problem only at the lowest level of the design hierarchy, layout. This is in spite of the fact that the need for hierarchical design methods is, in general, accepted. Many of the benefits of such design methods are lost by the use of low level tools. We advocate the use of high level description languages, which allow the required behaviour of a circuit to be precisely specified. Designs can then be reasoned about and any proposed implementation can be proved correct. Simulation as a means of "verification" will become less feasible as circuits increase in complexity. We will have to use mathematical methods. Once we have a formal design language, we can begin to think of "compiling" circuit descriptions directly onto silicon. "Silicon compilation" promises to reduce design time and cost by relieving the designer of the need to consider irrelevant details, leaving him free to make the important decisions.

In this thesis, we propose an IC design language, μFP, which can describe the semantics (or behaviour) of a circuit and which can also capture details of its floor-plan (or layout). This dual power makes the language a good candidate for use in a silicon compiler. Most IC design languages describe either the semantics or the layout, but not both. This means that designers have to make a large (and artificial) jump from doing "semantic design" to doing "layout". However, the two are closely intertwined. It is important for the designer to be able to consider the effect on the final layout of a particular design decision. He must also be able to manipulate the layout, while keeping the semantics constant. μFP is designed to allow him to do both these things.

μFP is a variant of FP, the Functional Programming language introduced by Backus in [Backus 78]. There is a growing interest in functional programming languages because they allow us to write programs easily and quickly. The programs are written using "mathemetical" functions, and so are easier to reason about. We don't have to worry about irrelevant details such as the order in which the parameters of a function are evaluated. We have based μFP on a functional language in the hope of bringing some of these benefits to IC design. Obviously, an IC design language must allow the designer to reason about his circuits. μFP allows the programmer to reason about his programs by manipulating the programs themselves. The programs (or circuit descriptions) are just expressions "made" from a small number of primitive functions and combining forms (functionals that map functions into functions). These functions and combining forms (CFs) were chosen because they have nice algebraic properties. Thus, circuit descriptions can be easily manipulated, using the algebraic laws of the language. Also, each CF has a simple geometric interpretation, so that every μFP expression has an associated floor-plan.

μFP is designed to describe synchronous systems, using a discrete time model. It is not specific to a particular technology. Switching between technologies involves changing the primitive functions corresponding to circuit elements. Our aim is to support a structured hierarchical design method by allowing the designer to determine (by reasoning) whether a given combination of precisely specified components has the required behaviour.

This thesis includes a μFP manual. It also includes a study of the application of μFP and a discussion of our requirements for a VLSI design language. Chapter 2 is an introduction to the basic form of μFP. The first part of the chapter gives a relatively informal description of the language. The second part gives the formal semantics. We show how algebraic identities in μFP are proved and we demonstrate some transformations of μFP programs. In chapter 3, we give some examples of the use of μFP. We motivate the addition of some new combining forms which are appropriate to systolic arrays. Chapter 4 contains further examples of the use of the language, one of which is SADcell, the basic cell in a chip which we have designed. In chapter 5, we present our main example. We give a step by step derivation of a systolic correlator circuit. We introduce some techniques for the analysis of circuits in which 50% of the processors are active at any time. Chapter 6 describes how μFP may be "run" to give a simulator. We also describe a program which produces the floor-plan of a given μFP expression. The first part of chapter 7 is a review of design tools, ranging from "automated graph paper" systems to silicon compilers. In the second part of the chapter, we turn our attention to design languages. We consider the important properties of a high level VLSI design language and we compare our approach to that of others who have applied formal techniques to the problems of VLSI design. In chapter 8, we present our conclusions and our plans for future work.

Chapter 2: Simple μFP

Introduction

μFP is a variant of FP, the Functional Programming language introduced in [Backus 78]. We will not give a detailed description of FP but will include only such details as are necessary to the understanding of μFP. In the first part of this chapter, we will give a relatively informal description of μFP, placing the emphasis on the geometric interpretation. In the second part of the chapter, we will give the formal semantics of μFP. We will prove that some FP axioms hold in μFP and we will demonstrate transformations of μFP programs.

A brief introduction to FP

A program in FP is simply an expression representing a function that maps objects into objects [Backus 78; Williams 81]. For example, + is an FP program representing a function which maps a pair of numbers onto their sum. The objects on which our programs operate can be undefined (⊥), atoms or sequences of objects. Note that this is a recursive definition. We shall take the set of atoms to be the integers, with a "don't care" value, '?'. Some possible objects are

⊥ 42 ⟨⟩ (the empty sequence) ⟨1,⟨1,⟨1,1⟩⟩⟩.

⟨ , , ⟩ denotes a sequence. We will represent "don't care" sequences by '?' also, although we should, strictly, have a different symbol for every possible shape.

Next, we need a set of primitive functions to operate on our objects. These divide into three main categories. (: denotes function application.)

1) Functions for manipulating sequences

   e.g. selector functions 1:⟨x1,x2,...xn⟩ = x1 if n≥1, ⊥ otherwise.

   2:⟨x1,x2,...xn⟩ = x2 if n≥2, ⊥ otherwise.

   append to the left, apndl e.g. apndl:⟨3,⟨4,5,6⟩⟩ = ⟨3,4,5,6⟩.

   append to the right e.g. apndr:⟨⟨1,2⟩,3⟩ = ⟨1,2,3⟩.

   matrix transposition, zip zip = [α1, α2, . . . αn] where n is the length of each of the subsequences of the matrix to be transposed.

2) Arithmetic functions

e.g. +, -, * +:⟨1,2⟩ = 3 +:⟨42⟩ = ⊥ *:⟨3,4,5⟩ = ⊥.

3) Predicates (we denote true by 1, false by 0)

e.g. greater than gt : ⟨4,1⟩ = 1 (true) not : 1 = 0.

Finally, we need a set of combining forms (CFs). CFs map functions into functions and so allow us to build up the functions (or programs) that we require. We use the following CFs :-

Composition (f ∘ g) : x = f : (g : x)

Construction [f1, . . . fn]:x = ⟨f1:x, f2:x, . . . fn:x⟩

Apply to all αf:x = ⟨f:x1,f:x2,..f:xn⟩ if x=⟨x1,x2,...xn⟩, ⊥ otherwise

Conditional (p→f;g):x = f:x if p:x=1, g:x if p:x=0, ⊥ otherwise

Constant $\overline{r}$: y = r if y ≠ ⊥, ⊥ otherwise

Insert left (/f):⟨x1⟩ = x1, (/f):⟨x1,..xn⟩ = f:⟨(/f):⟨x1,..xn-1⟩,xn⟩

Insert right (/Rf):⟨x1⟩ = x1, (/Rf):⟨x1,..xn⟩ = f:⟨x1,(/Rf):⟨x2,..xn⟩⟩

We can use the CFs and primitive functions to write new functions or programs. To add 1 to each of the elements of a sequence, we simply write $\alpha\overline{+1}$, where $\overline{+1}$ is the "add one" function. To add all the elements of a sequence, we write /R+ (or /+). A typical FP program is that which computes the length of a sequence :-

length = /R+ ∘ $\alpha\overline{1}$.

This remarkably short program treats each element of the sequence as a 1 ($\alpha\overline{1}$) and then (∘) adds them up (/R+). It is important to note that ∘ "does" the function on the right "first".

The combining forms of FP obey a series of algebraic laws, some of which are listed later in this chapter. These identities follow from the definitions of the CFs and require no proof in the algebra.

The basic building blocks of μFP

All FP functions take one input and produce one output. In μFP, however, functions take a sequence of inputs (over time) and produce a sequence of outputs. For example, the + function in μFP first takes a pair of numbers and produces their sum. It then takes another pair of numbers and produces their sum, and so on. So, for an input stream that looks like ‹‹1,2›,‹3,4›,‹5,6›,‹7,8›,...›, the output stream looks like ‹3,7,11,15,...›. This is an important difference between μFP and FP, and it is reflected in the formal semantics of μFP, which are given later in this chapter. This transition to operations on streams of inputs to produce streams of outputs is necessitated by the fact that we would like (eventually) to be able to deal with state. We already have the rudiments of a chip design language and we need a set of combining forms. We will adopt the CFs of the original FP and we will formally define their new semantics in the second part of this chapter. Here, we give their geometric interpretations. This will show the simple relationship between the semantics and the layout (or floor-plan).

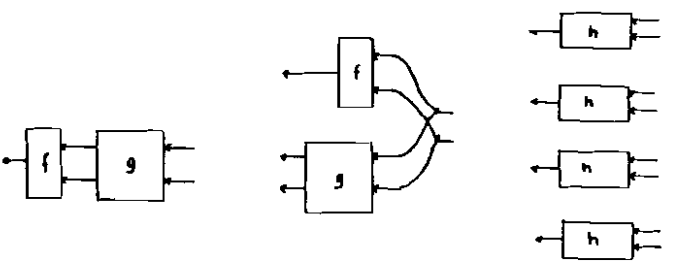


FIG 2.1 (a) Composition f°g (b) Construction (f,g) (c) Apply to all αh

FIG 2.1 shows the first three CFs of μFP. Note that data "flows" from right to left because of the definition of °.

If we want a circuit (or program) which repeatedly takes a pair of numbers, a and b, and gives us -(a+b), we would write -°+. If we wanted both the sum and the product of each pair of numbers, we would write [+, *]. A circuit which, on each cycle, takes a sequence of pairs of numbers and produces the sequence of their sums is given by α+.

Before considering the other CFs, let us describe a half-adder, ha, which (repeatedly) takes two bits and produces a sum bit and a carry bit. The sum bit is the exclusive or (xor) of the two input bits, while the carry bit is the and of the two bits. Thus, our half-adder is simply

ha = [xor, and] where

xor = and * [or, not * and].

This definition uses five "gates" and can be represented as follows :-

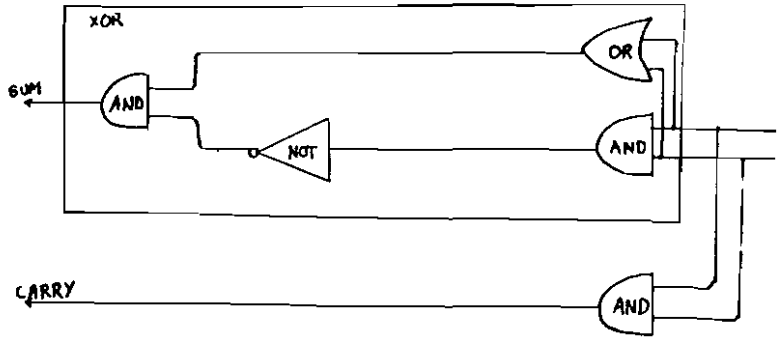


FIG 2.2 A half-adder with 5 "gates"

By applying the algebraic laws of μFP in a very simple way (as shown later in this chapter), we can transform our definition into

ha = [and * (1, not * 2), 2] * [or, and].

which has only four "gates".

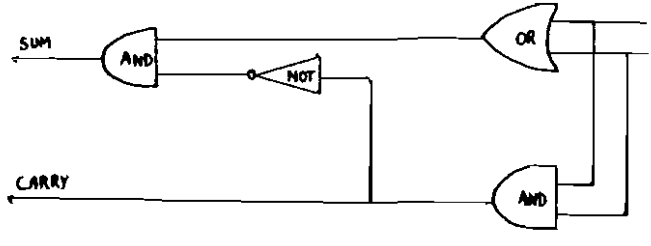


FIG 2.3 A half-adder with 4 "gates".

Note that both definitions of the half-adder have the same semantics, but different layouts.

We now introduce four more CFs. /\. and /\ can be used to "spread" inputs along a row of identical cells, as shown in FIG 2.4.

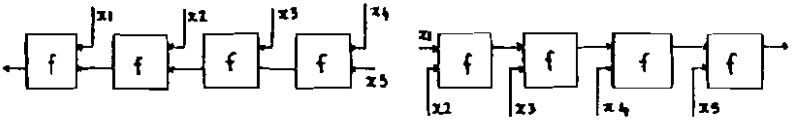


FIG 2.4 /\f : ‹x1, x2, x3, x4, x5› /\f : ‹x1, x2, x3, x4, x5›

This form is commonly used in circuits. Combined with the concept of state, it will allow us to describe the linear systolic arrays which are often used in signal-processing.

The interpretation of the conditional CF is shown in FIG 2.5.

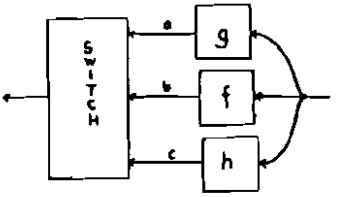


FIG 2.5 Conditional f → g; h

The switch chooses between inputs a and c, according to the value of b. So, a function which (repeatedly) selects the larger of two numbers is gt→1;2.

Finally, the constant function, $\overline{r}$, is just a source of rs, represented by  .

In the following section, we introduce a new CF, μ, which allows us to deal with the concept of state.

Dealing with State

The basic functions and combining forms introduced in the previous section can only be used to represent "combinatorial" functions. There is no way of representing the concept of state, which is central to digital design. Since most digital circuits, from shift registers to microprocessors, have some "memory", we are forced to add another combining form, μ. μ takes a function and produces a "function" which has internal state. FIG 2.6 shows the geometric interpretation of μ.

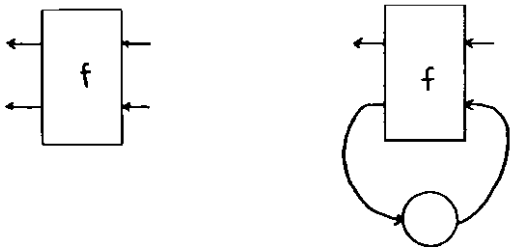


FIG 2.6 f μf

We use a "latch" to hold the state. Thus, the current state is supplied to the function as its second input and the second output of the function refreshes the state. The initial value in the latch is assumed to be '?', the "don't care" state. So, if

$$f:\langle\langle x1,?\rangle,\ \langle x2,s1\rangle,\ \langle x3,s2\rangle,\ \ldots\rangle = \langle\langle o1,s1\rangle,\ \langle o2,s2\rangle,\ \langle o3,s3\rangle,\ \ldots\rangle$$

then $\mu f : \langle x1,\ x2,\ x3,\ \ldots\rangle = \langle o1,\ o2,\ o3,\ \ldots\rangle$.

A formal definition of μ is given in the second part of this chapter. A simple example of its use should make its operation more clear. We would like to describe a shift-register cell, SR1. The output of an SR1 is its current state, and its new state is the input. We write

$$SR1 = \mu(2,\ 1).$$

The *output* function is 2, which selects the state. The *next state* function is 1, which selects the input. FIG 2.7 shows how we have used the internal latch to give us a shift-register cell.

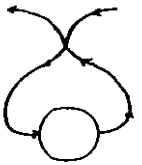


FIG 2.7 [2, 1] μ[2, 1]

The μFP function, [2, 1] operates on a sequence of pairs. So, for input

‹‹1, 0›, ‹0, 1›, ‹1, 0›, ‹1, 0›, . . .›,

its output would be

‹‹0, 1›, ‹1, 0›, ‹0, 1›, ‹0, 1›, . . .›.

μ[2, 1] operates on a sequence of inputs. The internal state is hidden and the initial state is assumed to be the "don't care" state, '?'. Its behaviour is best understood by considering the following transition table.

input	state	output	next state
0	?	?	0
1	0	0	1
0	1	1	0
0	0	0	0
1	0	0	1
0	1	1	0

For input

‹0, 1, 0, 0, 1, 0, . . . ›

the output from μ[2, 1] would be

‹?, 0, 1, 0, 0, 1, 0, . . .›,

as one would expect from a shift-register cell.

Whenever we write μf, the function f should be reducible to the form [x,y], where x is the *output* function and y is the *next state* function. If we wanted to be strictly formal, we would have to subscript every μ with the shape of its state. There is actually a different μ CF for every possible state shape. In the interests of legibility, we give all these combining forms the same name, μ.

Now that we know how to represent a one bit shift register cell, we can, of course, make a two bit shift register by composing two one bit shift registers

SR2 = SR1 * SR1

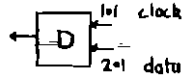
It is nice not to have to name or rename channels. The "joining" is taken care of by the definition of composition.

We can even represent a shift register of arbitrary length by

SR? = μ[1 * 2, apndr * [II * 2, 1]].

This captures the essence of any shift register. Its state is just a sequence of bits. Its output is always the first element of that sequence (*output function* = *1*2). Its new state is calculated by taking the rest of the sequence and sticking the new input bit on the end (*next state* function = apndr*[II*2,1]). This is an example of a circuit where the shape of the state implicit in the μ CF governs the behaviour. In this case, the μ really needs to be subscripted with the shape of the state. If SR? has a state of length n bits, then it behaves as an n bit shift register.

Another simple example is the D flip-flop.



D = μ[2, 1 * 1 → 2 * 1; 2]

This has inputs in the form of pairs containing clock (1*1) and data (2*1) signals. Our program represents the fact that

next output = current state

next state = data, if clock is high

current state otherwise.

Thus,

D : <<0,0>, <1,1>, <1,0>, <1,1>, . . .> = <?, ?, 1, 0, 1, . . .>.

We have represented very concisely the fact that data is "moved" into the flip-flop only when the explicit clock is high. (Note that we are using the usual convention that 1 on a data line is equivalent to true. So, we abbreviate (eq * [1 * 1, 1] → 2 * 1; 2) to (1 * 1 → 2 * 1; 2).)

Although we can now represent various circuits in our language, we cannot yet find out more about these circuits by manipulating their representations. We need to find (and prove) some theorems or algebraic identities about the language. We can then use those laws to transform circuit descriptions, to extract from them the information that we require.

Most of the laws of FP still hold in μFP. They must, of course, be proved. Some of the necessary proofs are given in the next section. We will, however, illustrate some of the useful identities, using the geometric interpretation of the CFs.

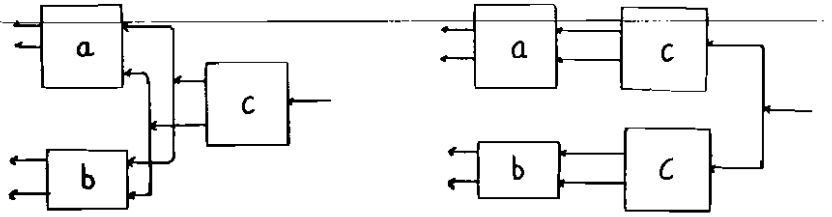


FIG 2.8 [a, b] ∘ c = [a ∘ c, b ∘ c]

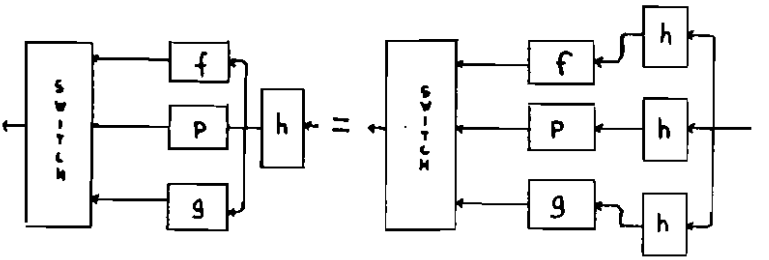


FIG 2.9 (p → f; g) ∘ h = p∘h → f∘h, g∘h

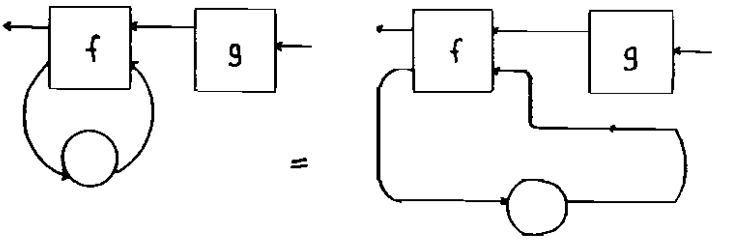


FIG 2.10 μf ∘ g = μ(f ∘ (g∘1, 2))

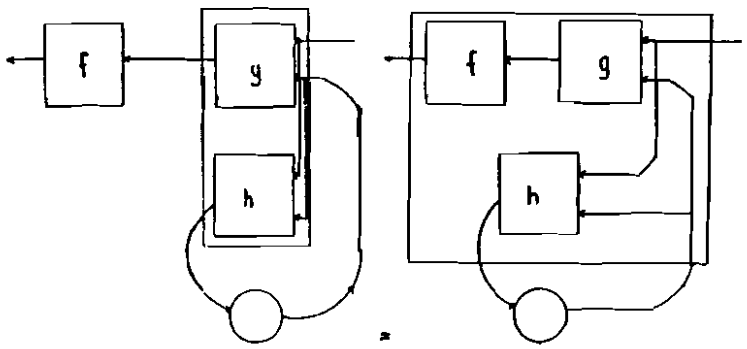


FIG 2.11 $1 \circ \mu[g, h] \approx \mu[f \circ g, h]$

One of the most useful laws is $C\mu$, which allows us to collapse the composition of two functions with state into one function with (larger) state. We combine the two old states into a pair.

$$\mu[f,g] \circ \mu[h,j] = \mu[f \circ [h \circ [1.2 \circ 2].1 \circ 2], [g \circ [h \circ [1,2 \circ 2].1 \circ 2], j \circ [1.2 \circ 2]]] \qquad C\mu$$

Although $C\mu$ looks quite complicated, its derivation is simple. It is shown in the next section. The application of $C\mu$ is just a question of mechanical substitution. In the next section, we show how $C\mu$ is used to transform $\mu[2, 1] \circ \mu[2, 1] = SR1 \circ SR1$ into $\mu[1 \circ 2, [2 \circ 2, 1]] = SR2$.

It might be useful to consider the geometric interpretations of both sides of this identity.

FIG 2.12 $\mu[2, 1] \circ \mu[2, 1]$ $\mu[1 \circ 2, [2 \circ 2, 1]]$

By following the arrows, we can see that both circuits behave in exactly the same way. The circuit on the right, though it seems to have a more complicated diagram, has only one μ and so it can easily be understood in terms of its *next output* and *next state* functions. Remembering that, because of the definition of composition, data "flows" from right to left, our state is of the form $\langle leftstate, rightstate \rangle$. The output is always the *leftstate* (1 ∘ 2) and the new state is always $\langle rightstate, input \rangle$ ([2 ∘ 2, 1]), giving us the expected behaviour of a two bit shift register.

For most circuits, the original high level specification will be in the form $\mu(x,y)$, a finite state machine type description. However, such a description is not, in general, suitable for implementation on silicon, because of the complexity of the *next state* function. In order to make a reasonable layout, we must decompose the state into small easily implementable blocks. We know, for example, how to implement $\mu(2, 1)$ as a shift register cell or latch on silicon. So, the process of translation from specification to implementation can be viewed as one of pushing the μs further and further down "into" the μFP expression, until they can go no further.

Whether we are trying to push down the μs to give a good layout, or to check that two subsections of our circuit behave correctly when put together, it is vital to have at our disposal a wide array of useful algebraic laws. In the following section, we will give the formal semantics of μFP. We will prove some algebraic identities and we will demonstrate some transformations of μFP programs.

Semantics, proofs and transformations

We introduce a "meaning" function, M, which gives the semantics of μFP in terms of FP.

$$f \text{ stateless} \quad \equiv \quad M[\![f]\!] = \alpha f \qquad \text{I}$$

Equation I is our base case. The meaning of a function which contains none of the combining forms (besides the constant function) is just αf (in FP). So, while the function 2, in FP, takes a sequence and produces its second element, the function 2, in μFP, takes a sequence of sequences and produces the sequence of their second elements. This gives us functions which work in a repetitive manner on a sequence of inputs, giving a sequence of outputs.

$$M[\![f \circ g]\!] = M[\![f]\!] \circ M[\![g]\!] \qquad \text{II}$$

Equation II says that the meaning of two composed functions is the composition of the meanings of the functions, as one might expect.

$$M[\![[f1, f2, \ldots fn]]\!] = zip \circ [M[\![f1]\!], M[\![f2]\!], \ldots M[\![fn]\!]] \qquad \text{III}$$

For the construction CF, we use zip, our matrix transpose function, to keep the types right. If the zip wasn't there, then the output of the right hand side, for m inputs, would be n m-element sequences, instead of m n-element sequences.

We can now use equation III to derive $M[\![\alpha f]\!]$.

$\alpha f = [f \circ 1, f \circ 2, \ldots f \circ n]$

$M[\![\alpha f]\!] = M[\![[f \circ 1, f \circ 2, \ldots f \circ n]]\!]$

$= zip \circ [M[\![f]\!] \circ \alpha 1, M[\![f]\!] \circ \alpha 2, \ldots M[\![f]\!] \circ \alpha n]$

$= zip \circ [M[\![f]\!] \circ \alpha 1, M[\![f]\!] \circ \alpha 2, \ldots M[\![f]\!] \circ \alpha n] \circ zip \circ zip$

(since $zip \circ zip = id$, the identity function)

$= zip \circ [M[\![f]\!] \circ \alpha 1 \circ zip, \ldots M[\![f]\!] \circ \alpha n \circ zip] \circ zip$

(Axiom 5, FP)

$= zip \circ [M[\![f]\!] \circ 1, M[\![f]\!] \circ 2, \ldots M[\![f]\!] \circ n] \circ zip$

($\alpha i \circ zip = i$)

$= zip \circ \alpha M[\![f]\!] \circ zip$

$\therefore \quad M[\![\alpha f]\!] = zip \circ \alpha M[\![f]\!] \circ zip \qquad \text{IV}$

(Note: Anything appearing inside M⟦ . . ⟧ is μFP. The rest is FP.) The equation for the conditional is quite straightforward. M⟦p → g; h⟧ must take a sequence of inputs and produce a sequence of booleans, which decide whether a particular output is given by M⟦g⟧ or M⟦h⟧.

$$M⟦p \rightarrow g;\ h⟧ = \alpha(1 \rightarrow 2;3) \circ zip \circ [M⟦p⟧, M⟦g⟧, M⟦h⟧] \qquad V$$

For /L and /R, we again use zip to keep the types right.

$$M⟦/Rf⟧ = /R(M⟦f⟧ \circ zip) \circ zip \qquad VI$$

$$M⟦/Lf⟧ = /L(M⟦f⟧ \circ zip) \circ zip \qquad VII$$

The equation for μ must take account of the fact that we are hiding the state

$$M⟦\mu f⟧ = out\ (M⟦f⟧)$$

where out g i = o

where ⟨o,s⟩ = zip ∘ g ∘ zip : ⟨i,?⟩:s⟩ VIII

O, s and I are sequences. : is infix "append to the left". The meaning of μf is defined in terms of the meaning of f. The functional out can be considered to be a new combining form which we have added to FP. It "hides" the state so that while M⟦f⟧ maps a sequence of input-state pairs to a sequence of output-state pairs, out M⟦f⟧ just maps a sequence of inputs to a sequence of outputs. The initial state is assumed to be '?', the "don't care" state.

These eight semantic equations define our design language, μFP. The definitions were chosen so that the truth of many of the theorems of FP is preserved in μFP.

Some algebraic laws of FP

(A1) $h \circ (p \rightarrow f; g) = p \rightarrow h \circ f; h \circ g$

(A2) $(p \rightarrow f; g) \circ h = p \circ h \rightarrow f \circ h; g \circ h$

(A3) $/f \circ [g1, \ldots, gn+1] = f \circ [/f \circ [g1, \ldots, gn], gn+1]$

(A4) $/f \circ [g] = g$

(A5) $[a, b] \circ c = [a \circ c, b \circ c]$

(A6) $1 \circ [a, b] = a$, in the domain of definition of b

(A7) $2 \circ [a, b] = b$, in the domain of definition of a

(A8) $a \rightarrow (a \rightarrow b; c); d = a \rightarrow b; d$

(A9) $\alpha f \circ apndl \circ [a, b] = apndl \circ [f \circ a, \alpha f \circ b]$

(A10) $/Rf \circ apndl \circ [a, b] = f \circ [a, /Rf \circ b]$

(A11) $\overline{r} \circ b = \overline{r}$, in the domain of definition of b

For instance, we would like to check that

$$(p \rightarrow f; g) \circ h = (p \circ h \rightarrow f \circ h; g \circ h)$$

holds in μFP. To do this, we check that the "meanings" of both sides are equal, as follows :-

We abbreviate M[[h]] to H, M[[p]] to P etc.

$M[\![(p \rightarrow f; g) \circ h]\!] = M[\![(p \rightarrow f; g)]\!] \circ H$	II
$= \alpha(1 \rightarrow 2; 3) \circ zip \circ [P, F, G] \circ H$	V
$= \alpha(1 \rightarrow 2; 3) \circ zip \circ [P \circ H, F \circ H, G \circ H)$	A5
$= \alpha(1 \rightarrow 2; 3) \circ zip \circ M[\![p \circ h]\!], M[\![f \circ h]\!], M[\![g \circ h]\!]]$	II
$= M[\![p \circ h \rightarrow f \circ h; g \circ h]\!]$	Q.E.D.

It is even simpler to show that, in μFP,

$$[a, b] \circ c = [a \circ c, b \circ c]$$

$M[\![[a, b] \circ c]\!] = zip \circ [A, B] \circ C$	II,III
$= zip \circ [A \circ C, B \circ C]$	A5
$= M[\![[a \circ c, b \circ c]]\!]$	III
	Q.E.D.

To show.

$$\alpha f \circ apndl \circ [a, b] = apndl \circ [f \circ a, \alpha f \circ b]$$

(Lemma1 $zip \circ \alpha apndl \circ zip \circ [A, B] = apndl \circ [A, zip \circ B]$ (In FP)

proof: $1 \circ LHS = \alpha 1 \circ \alpha apndl \circ zip \circ [A, B]$

$= \alpha 1 \circ zip \circ [A, B] = A = 1 \circ RHS$

$tl \circ LHS = tl \circ zip \circ \alpha apndl \circ zip \circ [A, B]$

$= zip \circ \alpha tl \circ \alpha apndl \circ zip \circ [A, B]$

$= zip \circ \alpha 2 \circ zip \circ [A, B] = zip \circ 2 \circ [A, B]$

$= zip \circ B = tl \circ RHS$

$\therefore$ RHS = LHS)

Proof: $zip \circ M[\![\alpha f \circ apndl \circ [a, b]]\!]$

$= \alpha F \circ zip \circ \alpha apndl \circ zip \circ [A, B]$

$= \alpha F \circ apndl \circ [A, zip \circ B]$ lemma1

$= apndl \circ [F \circ A, \alpha F \circ zip \circ B]$ A9

$= apndl \circ [F \circ A, zip \circ zip \circ \alpha F \circ zip \circ B]$ (zip∘zip=id)

$= zip \circ \alpha apndl \circ zip \circ [F \circ A, zip \circ \alpha F \circ zip \circ B]$ lemma1

$= zip \circ M[\![apndl \circ [f \circ a, \alpha f \circ b]]\!]$ I,II,III,IV

Q.E.D.

We can use the same lemma to prove that

$/^R f \circ apndl \circ [a, b] = f \circ [a, /^R f \circ b]$.

$M[\![/^R f \circ apndl \circ [a, b]]\!] = /^R(F \circ zip) \circ zip \circ \alpha apndl \circ zip = [A, B]$ I,II,III,VI

$= /^R(F \circ zip) \circ apndl \circ [A, zip \circ B]$ lemma1

$= F \circ zip \circ [A, /^R(F \circ zip) \circ zip \circ B]$ A10

$= M[\![f \circ [a, /^R f \circ b]]\!]$ Q.E.D

To prove $/\lambda f \circ [g1, \ldots gn+1] = f \circ [/\lambda f \circ [g1, \ldots gn], gn+1]$

$M[\![/\lambda f \circ [g1, \ldots gn+1]]\!]$

$= /\lambda(F \circ zip) \circ zip \circ zip \circ [G1, \ldots Gn+1]$ II,III,VII

$= F \circ zip \circ [/\lambda(F \circ zip) \circ zip \circ zip \circ [G1, \ldots Gn], Gn+1]$ A3

$= M[\![f \circ [/\lambda f \circ [g1, \ldots gn], gn+1]]\!]$ II,III,VII

Q.E.D.

As an example of a less formal proof, let us show that

$$a \longrightarrow (a \longrightarrow b;\ c);\ d = a \longrightarrow b;\ d$$

Lemma2 $\alpha(1 \longrightarrow 2;3) \cdot zip \cdot [A,B,C]$

$= \alpha(1 \longrightarrow 2;3) \cdot [(1 \cdot A, 1 \cdot B, 1 \cdot C), (2 \cdot A, 2 \cdot B, 2 \cdot C, \ldots)$

$= [(1 \cdot A \longrightarrow 1 \cdot B; 1 \cdot C), (2 \cdot A \longrightarrow 2 \cdot B; 2 \cdot C), \ \ldots\ ]$

Proof: $M[\![a \longrightarrow (a \longrightarrow b;\ c);\ d]\!]$

$= \alpha(1 \longrightarrow 2;3) \cdot zip \cdot [A,\ \alpha(1 \longrightarrow 2;3) \cdot zip \cdot [A,\ B,\ C],\ D]$ V

$= \alpha(1 \longrightarrow 2;3) \cdot zip \cdot [A, [(1 \cdot A \longrightarrow 1 \cdot B, 1 \cdot C), (2 \cdot A \longrightarrow 2 \cdot B, 2 \cdot C), \ldots], D]$ lemma2

$= [(1 \cdot A \longrightarrow (1 \cdot A \longrightarrow 1 \cdot B; 1 \cdot C); 1 \cdot D),\ (2 \cdot A \longrightarrow (2 \cdot A \longrightarrow 2 \cdot B; 2 \cdot C); 2 \cdot D), \ldots]$ lemma2

$= [(1 \cdot A \longrightarrow 1 \cdot B;\ 1 \cdot D),\ (2 \cdot A \longrightarrow 2 \cdot B;\ 2 \cdot D),\ \ldots\ ]$ A8

$= \alpha(1 \longrightarrow 2;\ 3) \cdot zip \cdot [A,\ B,\ D]$ lemma2

$= M[\![a \longrightarrow b;\ d]\!]$ V

Q.E.D.

If we try to prove that

$$h \cdot (p \longrightarrow f;\ g) = p \longrightarrow h \cdot f;\ h \cdot g.$$

we find that we must have an additional constraint - h must be stateless. This is because the output of a function with state depends not only on its current input, but on all of its previous inputs.

We have derived some useful identities concerning μ :-

$\mu f \cdot g = \mu(f \cdot (g \cdot 1,\ 2))$

$f \cdot \mu[g,\ h] = \mu(f \cdot g,\ h)$

$\mu[f,g] \cdot \mu[h,j] = \mu(f \cdot [h \cdot (1.2 \cdot 2), 1 \cdot 2),\ [g \cdot [h \cdot [1.2 \cdot 2], 1 \cdot 2),\ j \cdot (1.2 \cdot 2)]]$ Cμ

$\mu\mu[[f,\ g],\ h] = \mu([f,\ [g\ \ h]] \cdot ([1,\ 1 \cdot 2),\ 2 \cdot 2])$

To prove these, we use a slightly different method.

To prove $\mu f \circ g = \mu(f \circ [g \circ 1, 2])$

$M[\![\mu f \circ g]\!]:i = \text{out } F : G : I = o$

where $\langle o.s\rangle = zip \circ F \circ zip : \langle G:I, ?Is\rangle$ II.VIII

$M[\![\mu(f \circ [g \circ 1, 2])]\!]:i = \text{out } (F \circ zip \circ [G \circ \alpha 1, \alpha 2])):I = o$

where $\langle o.s\rangle = zip \circ F \circ zip \circ [G \circ \alpha 1, \alpha 2] \circ zip : \langle I, ?Is\rangle$

$= zip \circ F \circ zip \circ [G \circ \alpha 1 \circ zip, \alpha 2 \circ zip] : \langle I, ?Is\rangle$

$= zip \circ F \circ zip \circ [G \circ 1, 2] : \langle I, ?Is\rangle$

$= zip \circ F \circ zip : \langle G:I, ?Is\rangle$

$\therefore \quad M[\![\mu f \circ g]\!]:i = M[\![\mu(f \circ [g \circ 1, 2])]\!]:i$

$\Rightarrow \quad \mu f \circ g = \mu(f \circ [g \circ 1, 2])$ Q.E.D

To prove $f \circ \mu[g, h] = \mu[f \circ g, h]$

$M[\![f \circ \mu[g,h]]\!].i = F \circ (\text{out } zip \circ [G, H]):I$ I.III.VIII

$= F:o$ where $\langle o.s\rangle = zip \circ zip \circ [G, H] \circ zip:\langle I, ?Is\rangle$

$= F:G:zip:\langle I, ?Is\rangle$ where $s = H:zip:\langle I, ?Is\rangle$

$M[\![\mu(f \circ g, h)]\!]:i = (\text{out } zip \circ [F \circ G, H]):I$ II.III.VIII

$= o$ where $\langle o.s\rangle = zip \circ zip \circ [F \circ G, H] \circ zip:\langle I, ?Is\rangle$

$= F:G\ zip\ \langle I, ?Is\rangle$ where $s = H:zip\ \langle I, ?Is\rangle$

$\therefore \quad f \circ \mu[g, h] = \mu[f \circ g, h]$ Q.E.D

To prove $\mu\mu[[f, g], h] = \mu[[f, [g, h]] \cdot [[1, 1\cdot 2], 2\cdot 2]]$

M⟦ $\mu\mu[[f, g], h]$ ⟧:i

= (out (out zip ∘ [zip ∘ [F, G], H])):i

= o where ⟨o,s⟩ = zip:(out zip ∘ [zip ∘ [F, G], H]):zip:⟨i,?is⟩

= zip:o1 where ⟨o1,s1⟩ = [zip ∘ [F, G], H]:zip:⟨zip:⟨i,?is⟩,?is1⟩

= [F, G]:zip:⟨zip:⟨i, ?is⟩, ?is⟩ where s1 = H:zip:⟨zip:⟨i,?is⟩,?is1⟩

= F:zip:⟨zip:⟨zip:⟨i, ?is⟩, ?is1⟩ where s = G:zip:⟨zip:⟨i,?is⟩,?is1⟩

& where s1 = H:zip:⟨zip:⟨i,?is⟩,?is1⟩

M⟦ $\mu[[f, [g, h]] \cdot [[1, 1\cdot 2], 2\cdot 2]]$ ⟧:i

= out zip∘[F, zip∘[G, H]] ∘ zip∘[zip∘[α1, α(1∘2)], α(2∘2)]:i

= o where ⟨o,s⟩ = [F, zip∘[G, H]]∘zip∘[zip∘[α1, α(1∘2)], α(2∘2)]:zip:⟨i,?is⟩

= F:zip:⟨zip:⟨i,?is1⟩,?is2⟩ where s = zip:⟨s1,s2⟩ (?is = zip:⟨?is1,?is2⟩)

= zip ∘ [G, H]:zip:⟨zip:⟨i,?is1⟩,?is2⟩

= F:zip:⟨zip:⟨i,?is1⟩,?is2⟩ where s1 = G:zip:⟨zip:⟨i,?is1⟩,?is2⟩

& where s2 = H:zip:⟨zip:⟨i,?is1⟩,?is2⟩

∴ $\mu\mu[[f, g], h] = \mu[[f, [g, h]] \cdot [[1, 1\cdot 2], 2\cdot 2]]$ Q.E.D.

$C\mu$ is by far the most important algebraic law concerning μ. The following diagram illustrates its derivation.

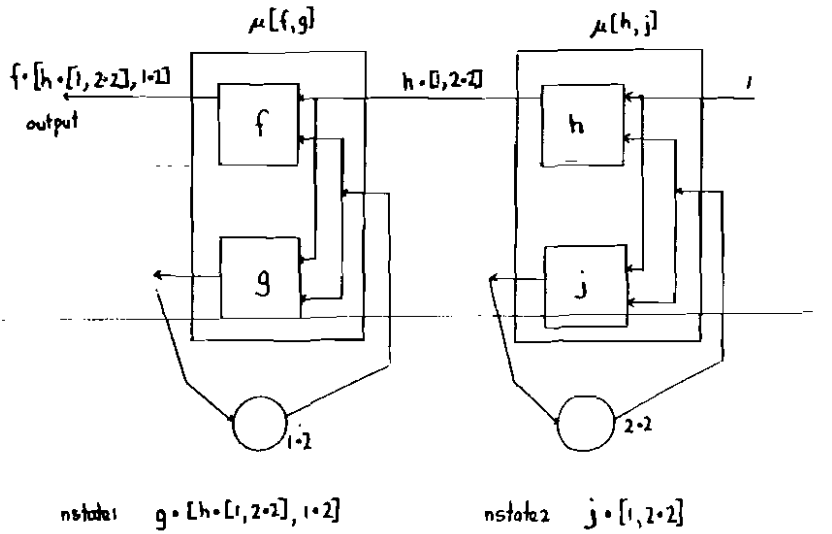


FIG 2. 13 Derivation of $C\mu$ $\mu(f,g)*\mu(h,j) = \mu(output,(nstate1),nstate2))$

$\mu(f,g)*\mu(h,j) = \mu(f*(h*(1,2*2),1*2),\ (g*(h*(1,2*2),1*2),\ j*(1,2*2)))$ $C\mu$

Proof: M[[$\mu(f,g) * \mu(h,j)$]]:I

= (out zip * (F, G)).(out zip * (H, J)) I

= o where <o,s> = (F, G) * zip <out zip * (H, J)>I, ?Is>

= F.zip:<H zip:<I, ?Is1>, ?Is>

where s = G zip <H zip.<I, ?Is1>, ?Is>

& where s1 = J zip <I, ?Is1>

M[[$\mu(f*(h*(1,2*2),1*2),\ (g*(h*(1,2*2),1*2),\ j*(1,2*2)))$]]:I

= (out zip*(F*zip*(H*zip*(α1,α(2*2)),α(1*2)),

zip*(G*zip*(H*(α1,α(2*2)),α(1*2)), J*zip*(α1,α(2*2))))):I

= F:zip:<H zip:<I, ?Is1>, ?Is2>

where s = zip:<s2,s1> (?Is = zip:<?Is2,?Is1>)

& where s2 = G:zip:<H:zip:<I, ?Is2>, ?Is1>

& where s1 = J:zip:<I, ?Is2> Q.E.D.

As a first example of the use of Cμ, let us compute

$\mu[2,1] \cdot \mu[2, 1]$ (= SR1 • SR1).

f = h = 2. g = j = 1

$\mu[2, 1]\cdot\mu[2,1] = \mu(2\cdot[2\cdot[1,2\cdot2],1\cdot2],\ [1\cdot[2\cdot[1,2\cdot2],1\cdot2],\ 1\cdot[1,2\cdot2]])$

Well, $2\cdot[1,2\cdot2] = 2\cdot2$. $2\cdot[2\cdot2,1\cdot2] = 1\cdot2$

$1\cdot[2\cdot2,1\cdot2] = 2\cdot2$. $1\cdot[1,2\cdot2] = 1$

So, we get $\mu[1\cdot2,\ [2\cdot2,\ 1]] = SR2$.

A slightly more complicated transformation is that which transforms a half-adder with 5 "gates" into one with 4 "gates".

ha = [and • [or, not • and], and]

= [and • [or, not • and], 2 • [or, and]] A7

= [and•[1•[or,and], not•2•[or,and]], 2•[or,and]] A6,A7

= [and • [1, not • 2] • [or, and], 2 • [or, and]] A5

= [and • [1, not • 2], 2] • [or, and] A5

Another useful algebraic law is that which allows us to change from using $/_L$ to using $/_R$. The law, in FP, is

(A12) $/_L(f \cdot rev) \cdot rev = /_R f$

(rev is the list reverse function

rev:<> = <> rev:<x1,x2,...xn> = apndr:<rev:<x2,..xn>,x1>)

We must prove it in μFP

$M[\![/_L(f \cdot rev) \cdot rev]\!]$

$= /_L(F \cdot \alpha rev \cdot zip) \cdot zip \cdot \alpha rev$

$= /_L(F \cdot zip \cdot rev) \cdot rev \cdot zip$

(since zip • αrev • zip = rev

$= /_R(F \cdot zip) \cdot zip$ A12

$= M[\![/_R f]\!]$ Q.E.D.

Of course,

$/\!\!\backslash(f \circ rev \circ rev) \circ rev \circ rev = /\!\!R(f \circ rev) \circ rev = /\!\!L f.$

FIG 2.14 illustrates A12.

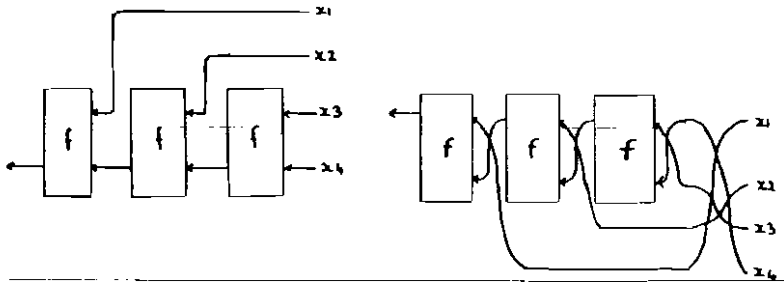


FIG 2.14 $/\!\!R f$ $/\!\!\backslash(f \circ rev) \circ rev$

If we wish to find a good "layout" for a circuit, laws such as A12 are obviously useful, particularly since we are not very keen on having "crossed wires" on silicon.

In the following chapter, we will give some examples of the use of μFP, and we will demonstrate the need for some new combining forms

Chapter 3: Some examples. Extensions to allow us to deal with more complicated data flow

Some examples

In order to help us to become familiar with μFP, let us try to describe some simple circuits. We will first consider some of the different types of shift register. We have already seen how to denote a serial in/ serial out (SISO) shift register of arbitrary length :-

SISO? = μ[1*2, apndr * [ti*2, 1]].

A serial in/ parallel out (SIPO) shift register is very similar, except that its output is the whole of the state, not just its leftmost element. So,

SIPO? = μ[2, apndr * [ti*2, 1]].

If we want to have a CLEAR facility, we write

CSIPO? = μ[2, 2*1 → $\alpha\bar{0}$*2; apndr * [ti*2, 1*1]].

The input is now in the form ‹serial input, clear›. If CLEAR (2*1) is high, we set all of the bits of the state to be zero, using $\alpha\bar{0}$*2.

A slightly more complicated example is a parallel in/ serial out (PISO) shift register which has parallel inputs and a shift/load switch, and which accepts serial data when it is shifting.

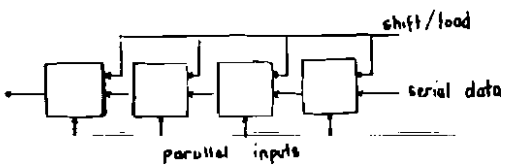


FIG 3.1 A parallel in/ serial out shift register, with serial data

The input is in the form ‹shift/load, serial data, parallel inputs›. If shifting is enabled, the circuit behaves like a SISO shift register. Otherwise, the parallel inputs overwrite the state. So, the μFP description is

PISO? = μ[1*2, 1*1 → apndr * [ti*2, 2*1]; 3*1]

With a CLEAR, we get

CPISO? = μ[1*2, 4*1 → $\alpha\bar{0}$* *1 → apndr * [ti*2, 2*1]; 3*1]

At an even more abstract level, let us consider very simple multiplexer and demultiplexer circuits. A MUX has as inputs a "switch" value and a pair of data inputs ⟨switch,⟨d1,d2⟩⟩. If the switch is high, the MUX passes the first data input through. Otherwise, it passes the second data input. Thus,

$$MUX = 1 \rightarrow 1*2;\ 2*2.$$

A demultiplexor has a switch, a single data input and two data outputs. If the switch is high, DEMUX passes its input through on the first output "channel" and sets the other output to zero. If the switch is low, DEMUX passes the input through on the second output channel and sets the first to zero. So,

$$DEMUX = 1 \rightarrow (2, \bar{0});\ (\bar{0}, 2).$$

We would like to check that the circuit shown in FIG 3.2 is the identity on the data input, 2.

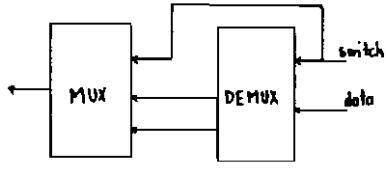


FIG 3.2 MUX * [1, DEMUX]

MUX * [1, DEMUX]

$= (1 \rightarrow 1*2;\ 2*2) * [1, (1 \rightarrow (2,\bar{0});\ (\bar{0},2))$

$= 1 \rightarrow 1*(1 \rightarrow (2,\bar{0})\ (\bar{0},2));\ 2*(1 \rightarrow (2,\bar{0});\ (\bar{0},2))$ A2,A6,A7

$= 1 \rightarrow (1 \rightarrow 2,\ \bar{0}),\ (1 \rightarrow \bar{0};\ 2)$ A1(1 stateless),A6,A7

$= 1 \rightarrow 2;\ 2$ A8

$= 2$

This means that our MUX and DEMUX circuits "match" each other correctly.

Another familiar example is the building of a full-adder from half-adders.

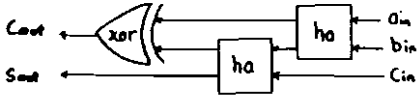


FIG 3.3 A full-adder made from 2 half-adders

A half-adder which produces a ‹carry-bit, sum-bit› pair is

ha = [and, xor]

The circuit shown in FIG 3.3 is

fa = [xor * [1, 1*2), 2*2] * [1*1, ha*[2*1, 2)] * [ha*1, 2].

When we expand out the half-adders, we find that

fa = [xor * [and*1, and*[xor*1, 2)), xor*[xor*1, 2)],

where the inputs are in the form ‹‹ain,bin›,cin› and the outputs are ‹cout, sout› pairs.

In order to add two M-bit numbers, we need a cascade of M full-adders, as shown in FIG 3.4.

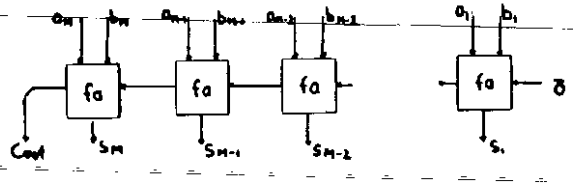


FIG 3.4 An M-bit adder

Although we can describe this circuit, it poses us some problems because the simple version of μFP which we have described is suited only to describing circuits of the form shown in FIG 3.5.

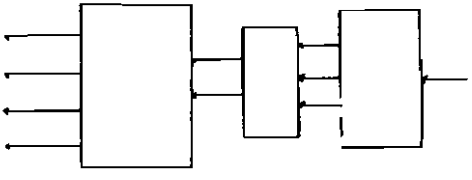


FIG 3.5. A "typical" μFP circuit

That is, the data is input to the rightmost "block" of the circuit it then passes through and is processed by the various blocks in the circuit and the outputs emerge from the leftmost block. This doesn't seem to be too much of a constraint until we try to put our M-bit adder into this form. The result is shown in FIG 3.6.

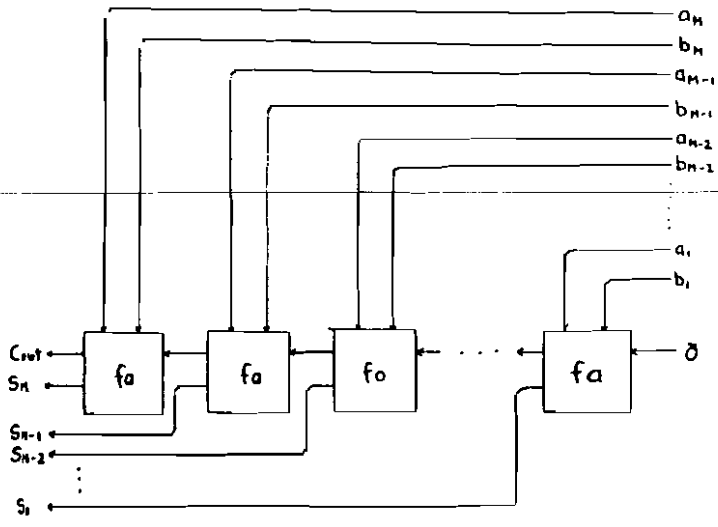


FIG 3.6 The adder in "μFP form"

We can describe this circuit in μFP by dividing it up into blocks which not only act as full-adders but which also pass on that part of the sum which has been calculated by the previous full-adders in the chain. Thus, each block accepts inputs of the form ‹‹ai,bi›,‹ci,si-1,si-2,...s1››. Obviously, for the rightmost block, the list of previous sum-bits is empty. So, our extended full-adder is

efa = concat * [fa * [1, 1*2], 1!*2].

For the ith block, we work out the carry/sum pair ‹ci,si› and we concatenate it with the list of previous sum-bits ‹si-1,si-2,...s1›, which gives us ‹ci,si,s-1,si-2,...s1›. This is the second input to the i+1th block.

If we assume that each input to the adder is a pair ⟨A,B⟩, where A and B are the two M-bit numbers, then the circuit of FIG 3.6 is given by

adder = [1, 1] * /R(efa) * apndr * [zip, [ū]].

We use the matrix transpose function, zip, to transform each pair of M-bit numbers into a list of M ⟨ai,bi⟩ pairs, which can then be passed to the full-adders. We apply [1,1] to the output of /R(efa) because we want our outputs to be carry/sum pairs.

Although we we have managed to describe our cascade of full-adders, the description is not as simple as we might have hoped. This is because the data-flow in the circuit of FIG 3.4 does not correspond to our simple standard type of data-flow. The problem is made more acute because we are seeking to capture information about circuit layouts, as well as about their semantics. Once we have transformed our original circuit specification into its final form, we will want to lay it out automatically. When we compare FIGs 3.4 and 3.6, it is immediately clear that FIG 3.6 is not a satisfactory layout of the circuit. It has too many unnecessary long wires. Yet it is the circuit layout that would be generated by our μFP description of the adder! μFP must be extended to deal with circuits in which the data does not simply "flow" from one side of the circuit to the other. Before we describe the extensions which we have made to μFP, we will give one more example of the use of the original μFP.

We would like to describe a pattern matcher [Foster, Kung 80]. Let us suppose that our pattern, a1,a2...an, is fixed and that we want to compare it to a series of n-tuples. We use n cells, each of which has one of a1..an "hard-wired" into it

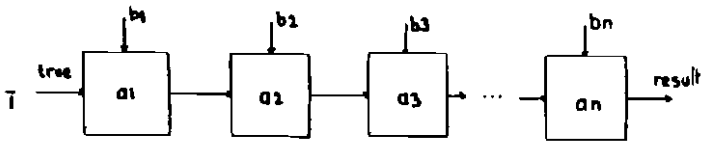


FIG 3.7 An n cell pattern matcher

The action of each cell is to compute

and * [1, eq * [a!, 2]].

The leftmost cell receives a 1 (or true) as its first input and the result is then chased all the way down the array, emerging at the right hand end as a 1 if ⟨a1,a2,..an⟩=⟨b1,b2,...bn⟩.

When describing the circuit in μFP, it is easiest to give each cell its 'hard-wired' reference value as a constant input. The action of each cell is then given by

PM1 = and * [1, eq * [1*2, 2*2]].

The whole circuit is

matcher = /PM1 * apndl * [$\overline{1}$, zip * [id, [$\overline{a}$1, $\overline{a}$2, . . . $\overline{a}$n]]].

Each input to the /PM1 is a list whose first element is the initialising input ($\overline{1}$), and whose tail is a list of input character/ reference character pairs.

Now, let us suppose that we have a long sequence of input characters and that we want to compare each sub-sequence of length n with the pattern ⟨a1,a2,..an⟩. To do this, we can put the input through an n bit serial in/ parallel out shift register (SIPO) before passing it to the matcher. This gives us the circuit shown in FIG 3.8

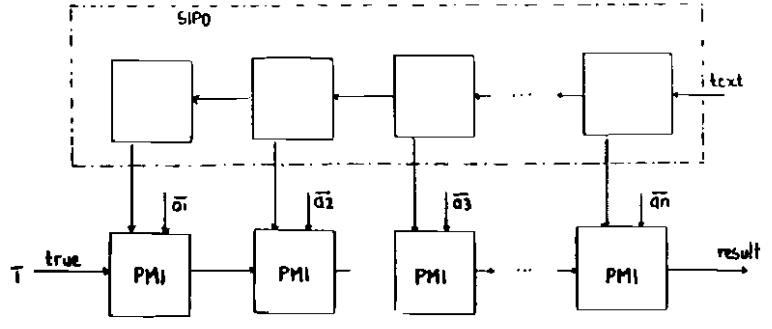


FIG 3.8. pattern matcher = /PM1 * apndl * [$\overline{1}$, zip * [SIPO, [$\overline{a}$1, $\overline{a}$2, .. $\overline{a}$n]]]

Although our circuit behaves correctly as a pattern matchor, it has a period of O(n) because, in each cycle, the result value must pass through all n PM1 cells before being output. Our aim is to have a period of O(1). In a later chapter, we will analyse a very similar circuit (the systolic correlator) and we will show how pipelining can be introduced to improve the performance of the circuit.

An interesting feature of our pattern matcher is that we have managed, by the "trick" of using a serial in/ parallel out shift register, to achieve quite a complicated pattern of data flow. The circuit is, in fact, a series of cells through which the text flows in one direction and the pattern in the other, as shown in FIG 3.9.

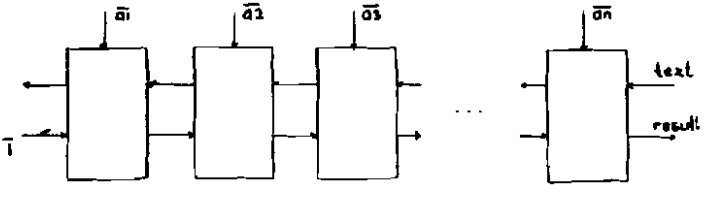


FIG 3.9 A different view of our pattern matcher

The only kind of composition which the current version of μFP allows is one in which all the data flow is in one direction. This means that we are not "allowed" to divide our circuits up into cells which communicate in both directions. This is a grave restriction since many systolic arrays have data stepping through the array in one direction and a result stream moving through in the other direction, interacting with the data stream in the processors. One of the new combining forms which we will add to μFP is one which will allow us to specify that two adjacent cells communicate in both directions.

The extensions to μFP detailed in the following section allow the user to describe circuits in which the data flow is more complicated than a simple unidirectional flow, without having to resort to tricks. For such circuits, a description using the new extensions is likely to be simpler and easier to read than one written in the original μFP. This is because some of the routing of data, which would have to be specified explicitly in simple μFP, is taken care of by the new combining forms. The new CFs also give the user more control over the layouts associated with his circuit descriptions since he can use "cells" which have vertical inputs and outputs, instead of having to force all of his data to flow horizontally. Many systolic arrays consist of either a linear array or a grid of processing elements in which data fronts move both horizontally and vertically. The new combining forms are useful for describing such circuits.

Extensions to allow us to deal with more complicated data flow

Our first extension introduces vertical data flow. We move to blocks of the form shown in FIG 3.10.

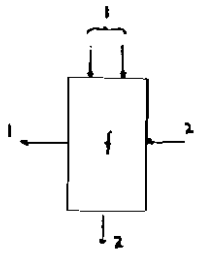


FIG 3.10

That is, both the inputs and the outputs are pairs. The first element of the input forms the vertical input. The second is horizontal, as shown. For the output, the first element is horizontal and the second vertical. The block is still just a function from a sequence of inputs to a sequence of outputs. We have merely put a constraint on the shape of the inputs and outputs so there is no need make any extensions to our formal semantics.

First, we introduce two new combining forms which allow us to compose two such blocks together either horizontally or vertically

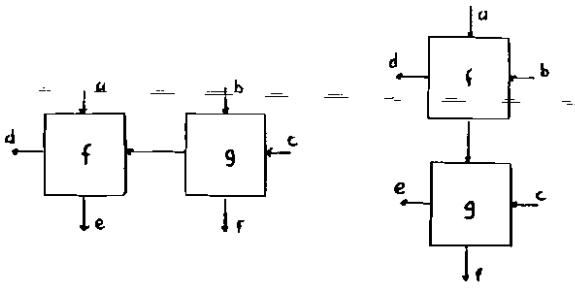


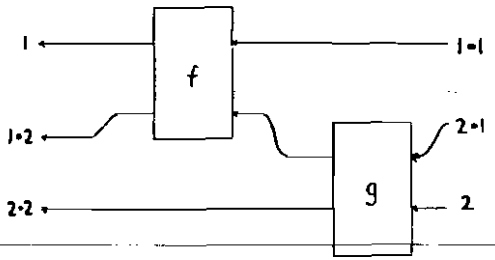
FIG 3.11

The two new CFs, ← and ↓, can be defined in μFP.

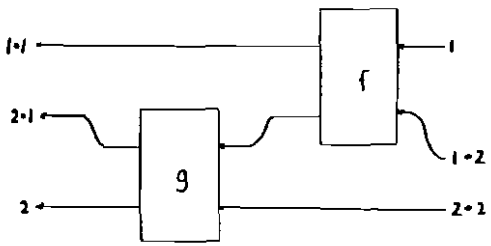
f ← g = [1*1, [2*1, 2]] • [f • [1, 1*2], 2*2] • [1*1, g • [2*1, 2]] IX

f ↓ g = [[1, 1*2], 2*2] • [1*1, g*[2*1, 2]] • [f • [1, 1*2], 2*2] X

FIG 3.12 illustrates these definitions.



f ← g = [1*1, [2*1, 2]] • [f • [1, 1*2], 2*2] • [1*1, g • [2*1, 2]] IX



f ↓ g = [[1, 1*2], 2*2] • [1*1, g*[2*1, 2]] • [f • [1, 1*2], 2*2] X

FIG 3.12 An illustration of the definitions of ← and ↓

The reader is invited to compare FIGs 3.11 and 3.12. The important thing about f ← g and f ↓ g is that the blocks which they produce are of the same form as the original f and g blocks. Thus, for f ← g, each input is of the form ‹‹a,b›,c› and each output of the form ‹d,‹e,f›› (cf. FIG 3.11). Similarly, for f ↓ g, each input is of the form ‹a,‹b,c›› and each output of the form ‹‹d,e›,f›. In each case, the resulting block is in our standard configuration (cf. FIG 3.10). This allows us to use the two combining forms freely, without having to worry about the "types" of the circuits which we compose. Any circuit (or program) whose inputs and outputs are pairs is of the right type to be used with these combining forms. For circuits of this type, there is an easy translation between a layout appropriate to simple μFP and one appropriate to this new version (and vice versa), as shown in FIG 3.13.

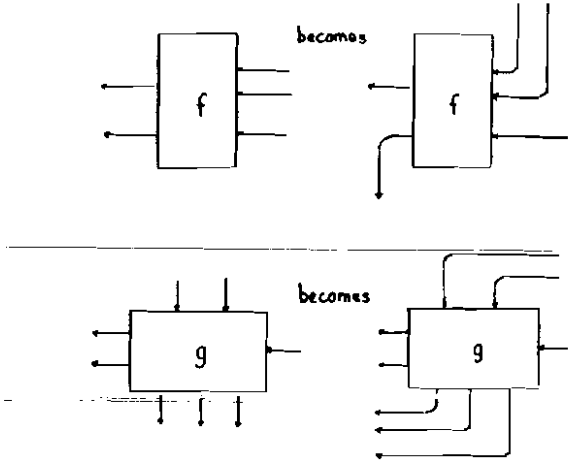


FIG 3.13 Translating between the two standard configurations

We will now derive an algebraic law which relates ← and ↓. FIG 3.14 shows two circuits which are semantically equivalent.

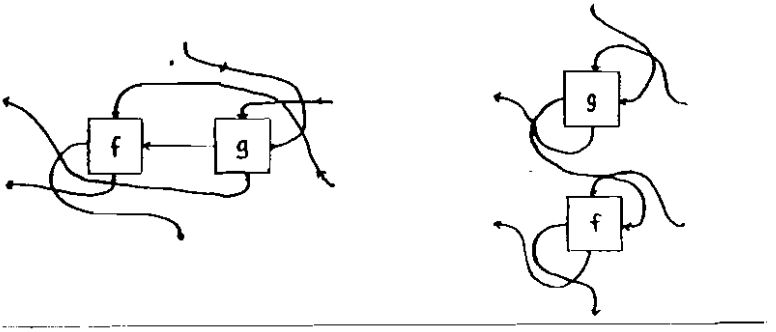


FIG 3.14 $[rev^*2,1]^*(f \leftarrow g)^*[rev^*2,1] = (rev^*g^*rev) \downarrow (rev^*f^*rev)$ B1

Proof: (rev is reverse)

$[rev^*2,1]^*(f \leftarrow g)^*[rev^*2,1]$

$= [rev^*2,1]^*[1^*1,[2^*1,2]]^*[[1^*[1,1^*2),2^*2]^*[1^*1,g^*[2^*1,2]]^*[rev^*2,1]$ IX

$= [[2,2^*1],1^*1]^*[1^*[1,1^*2],2^*2]^*[2^*2,g^*[1^*2,1]]$ A5.rev

$= [[2,2^*1],1^*1]^*[[1^*[2,1^*1),2^*1]^*[g^*[1^*2,1],2^*2]$

$= [[1,2^*2],1^*2]^*[2^*1,[1^*[2,1^*1)]^*[g^*[1^*2,1),2^*2]$

$= [[1,1^*2),2^*2]^*[1^*1,rev^*[1^*rev^*[2^*1,2]^*[rev^*g^*rev^*[1,1^*2],2^*2]$

$= (rev^*g^*rev) \downarrow (rev^*f^*rev)$ Q.E.D.

The algebraic law, B1, is potentially useful in the search for an efficient layout of a given circuit. Although both circuits in FIG 3.14 appear to have many "crossed wires", it may be that (rev^*g^*rev) and (rev^*f^*rev) are circuits which we "know" how to lay out. For instance, if we know the layout for g, and if all the inputs are one wire inputs, then we can obtain a layout for (rev^*g^*rev) simply by flipping the cell about the diagonal which starts in the lower left corner. It may also be that vertical *rather* than horizontal stacking (or vice versa) in one section of the circuit may allow us to lay out the whole circuit more efficiently.

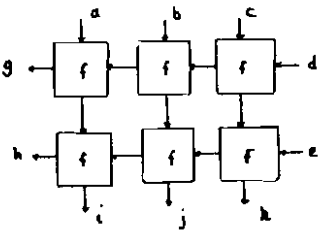
By replacing f by rev°f°rev, and g by rev°g°rev in B1, we can derive a similar law which gives ↓ in terms of ←.

(rev°2,1]°((rev°f°rev) ← (rev°g°rev))°[rev°2,1] = g ↓ f (by B1)

[2,rev°1] ° [rev°2,1] = [rev°2,1] ° [2,rev°1] = id

[2,rev°1] ° g↓f ° [2,rev°1] = (rev°f°rev) ← (rev°g°rev) B2

Next, we introduce a combining form, \, which allows us to form a two-dimensional grid of orthogonally connected cells, as shown in FIG 3.15.



\f for inputs of the form <<a,b,c>,<d,e>> outputs are of the form <<g,h>,<i,j,k>>

FIG 3.15

The ↓ and ← combining forms have exactly the same definitions in FP as they have in μFP. Thus, we can consider that ↓ and ← are new CFs in FP. Their definitions are given in FIG 3.12. This allows us to define \ in FP in terms of the FP versions of ↓ and ←. We could define \ directly in μFP (in terms of /L and /R) but, in this case, we have found it easier to add a new CF to FP and to add one equation to the list of equations which define μFP in terms of FP. We define \ by case analysis

1) \f:<<x>,<y>> = f:<x,y>

2) \f:<<x1, ..xn>,<y1,...ym>> =

[1,concat°2] ° (\f←\f) ° [[mst°1,[last°1][],2]:<<x1,..xn>,<y1,...ym>>

(if m > n)

[concat°1,2] ° (\f↓\f) ° [1,[[hd°2],tl°2]]:<<x1,..xn>,<y1,...ym>>

(if n > m)

(concat is list concatenation, concat:<<x1,x2>,<x3,x4>> = <x1,x2,x3,x4>, mst = rev ° tl ° rev)

We will illustrate the second definition.

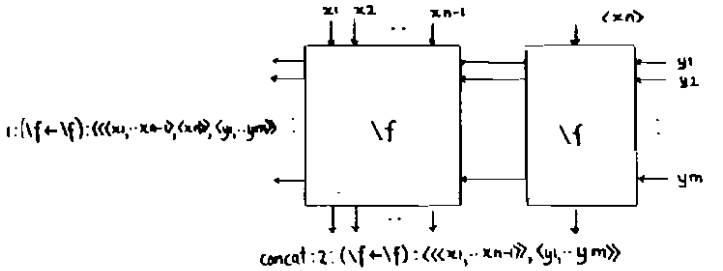


FIG 3.16 \f:<<x1...xn>,<y1...ym>> (in terms of ←, for n ≥ m)

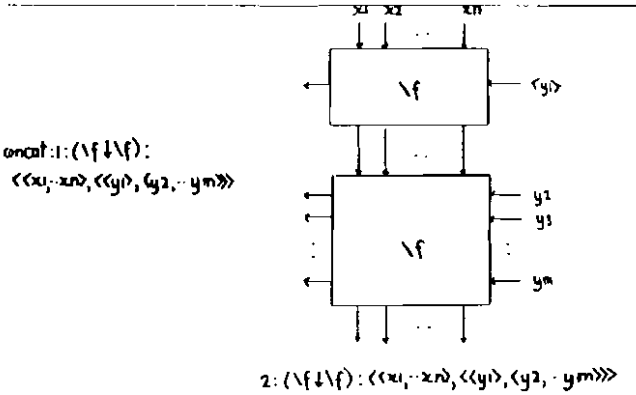


FIG 3.17 \f:<<x1...xn>,<y1...ym>> (in terms of ↓, for m ≥ n)

An interesting law about \ is illustrated in FIG 3.18.

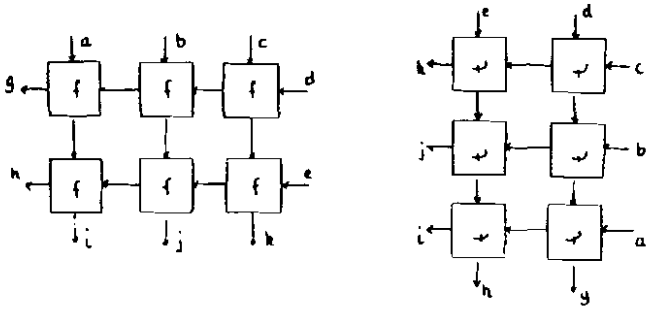


FIG 3.18 rev ° αrev ° \f ° αrev ° rev = \(rev ° f ° rev) FB3

To prove rev ° αrev ° \f ° αrev ° rev = \(rev ° f ° rev) (in FP)

1) LHS:<<x>,<y>> = rev:α(id):f:<y,x>

RHS:<<x>,<y>> = α(id):rev:f:<y,x> = rev:α(id):f:<y,x>

2) for n ≥ m

LHS:<<x1...xn>,<y1...ym>>

= [rev°concat°2,rev°1]°(\f←\f)°[[mst°1,[last°1]],2]:<<ym,...y2>,<y1>>,<xn,...x1>>

= [concat°rev°αrev°2,rev°1]°(\f←\f):<<<ym,...y2>,<y1>>,<xn,..x1>>

since rev°concat = concat°rev°αrev

= [concat°rev°2,1]°[rev°1,αrev°2]°(\f←\f)°[αrev°1,rev°2]:
<<<y2...ym>,<y1>>,<x1...xn>>

= [concat°rev°2,1]°((αrev°\f°αrev)←(αrev°\f°αrev))°[rev°2,1]
<<x1...xn>,<<y1>,<y2...ym>>>

[rev°1,αrev°2]°(f←g)°[αrev°1,rev°2]=(αrev°f°αrev)←(αrev°g°αrev))

= [concat°2,1]°(\(rev°f°rev)↓\(rev°f°rev)):<<x1...xn>,<<y1>,<y2,..ym>>>

by inductive hypothesis and B1 (in FP)

= RHS:<<x1...xn>,<y1,..ym>> Q.E.D.

The proof for m > n is as above, but backwards. We must now give the semantic equation which defines \ in μFP.

M⟦\f⟧ = zip ∘ αzip ∘ \(zip ∘ M⟦f⟧ ∘ zip) ∘ αzip ∘ zip XI

This is the most complicated semantic equation so far, and it requires some explanation. We must remember that the right hand side of the equation is an FP program. The program takes a stream of inputs, each of the form ⟨⟨x1,..xn⟩,⟨y1,..ym⟩⟩. However, \(zip∘M⟦f⟧∘zip) expects its entire input to be in the form ⟨⟨w1,..wn⟩,⟨z1,..zm⟩⟩. So, as before, we must use the matrix transpose function, zip, to keep the types right. αzip∘zip changes an N-list of n-list/m-list pairs into a pair whose first element is an n-list of N-lists and whose second element is an m-list of N-lists, where N is the length of the input stream. Let us take a simple example :-

αzip∘zip:⟨⟨a,b,c⟩,⟨d,e,f⟩⟩,⟨⟨g,h,i⟩,⟨j,k,l⟩⟩⟩

= αzip:⟨⟨a,b,c⟩,⟨g,h,i⟩⟩,⟨⟨d,e,f⟩,⟨j,k,l⟩⟩⟩

= ⟨⟨⟨a,g⟩,⟨b,h⟩⟨c,i⟩⟩,⟨⟨d,j⟩,⟨e,k⟩,⟨f,l⟩⟩⟩

On the output side of \(zip∘M⟦f⟧∘zip), we perform the same procedure in reverse, to ensure that the outputs of the program are in the same form as the inputs. We must now use equation XI to prove that

rev ∘ αrev ∘ \f ∘ αrev ∘ rev = \(rev ∘ f ∘ rev) B3

holds in μFP.

Proof: Let F = M⟦f⟧

M⟦LHS⟧

= α(rev∘αrev)∘zip∘αzip∘\(zip∘F∘zip)∘αzip∘zip∘α(αrev∘rev)

= zip∘rev∘ααrev∘αzip∘\(zip∘F∘zip)∘αzip∘ααrev∘rev∘zip

since zip∘ααrev = αrev∘zip and αrev∘zip = zip∘rev

= zip∘αzip∘rev∘αrev∘\(zip∘F∘zip)∘αrev∘rev∘αzip∘zip

since αrev∘zip = zip∘rev and rev∘αzip = αzip∘rev

= zip∘αzip∘\(rev∘zip∘F∘zip∘rev)∘αzip∘zip by FB3

= zip∘αzip∘\(zip∘αrev∘F∘αrev∘zip)∘αzip∘zip

= M⟦RHS⟧ Q.E.D.

Another law concerning \ is

$$\alpha\alpha rev \cdot \backslash f \cdot \alpha\alpha rev = \backslash(\alpha rev \cdot f \cdot \alpha rev) \qquad B4$$

We will not prove B4 as the proof is similar to that for B3. However, FIG 3.19 illustrates the equality.

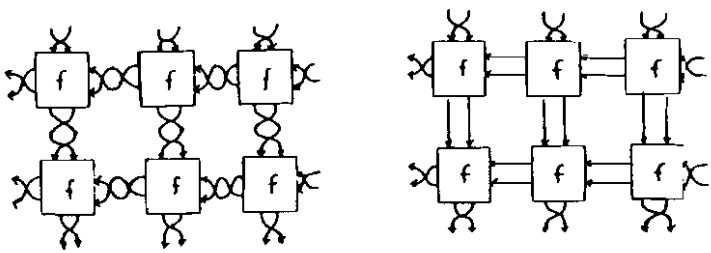


FIG 3.19 $\backslash(\alpha rev \cdot f \cdot \alpha rev) = \alpha\alpha rev \cdot \backslash f \cdot \alpha\alpha rev$

In fact, B4 generalises to

$$p \cdot p = id \Rightarrow \alpha\alpha p \cdot \backslash f \cdot \alpha\alpha p = \backslash(\alpha p \cdot f \cdot \alpha p) \qquad B4'$$

Laws such as these are potentially useful in the search for "good" layouts. For instance, the transformation from $\alpha\alpha rev \cdot \backslash(\alpha rev \cdot f \cdot \alpha rev) \cdot \alpha\alpha rev$ to $\backslash f$ eliminates many unwanted "reverses".

In the following section, we will introduce some further extensions to μFP, which allow us to compose cells together so that they communicate in both directions.

Extensions to allow bidirectional data flow

We now move to blocks of the form shown in FIG 3.20.

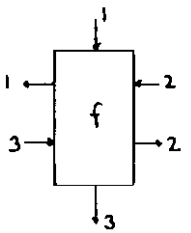


FIG 3.20

That is, both the inputs and the outputs are triples. The first element of each input is vertical, and the second two are horizontal, in opposite directions. For the output, the first two elements are horizontal in opposite directions, and the third is vertical. This scheme is obviously lop-sided in that the bidirectional flow is all horizontal. A scheme in which each edge of the block had both inputs and outputs would have a pleasing symmetry. However, we have chosen, as a first step, to investigate the scheme shown in FIG 3.20, because it is simpler and because we have not yet encountered a circuit whose data flow follows the more general scheme. We intend, later (but not in this thesis), to move to the more general scheme and to investigate some relevent systolic algorithms.

Next, we introduce two new CFs, which allow us to compose two of our blocks together, either vertically or horizontally.

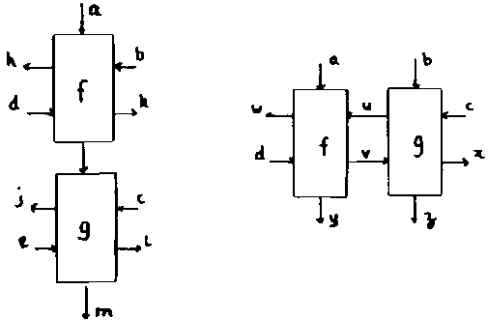


FIG 3.21 (a) f ! g (b) f ⟷ g

f ! g can be expressed in μFP, as f and g communicate in only one direction. The important point is that that the resulting circuit should have the same form as f and g.

f!g = [[1,1*3],[2,2*3],3*3]*[1*1,2*1,g*[3*1,2,3]]*[1*[1,1*2,1*3],2*2,2*3] XII

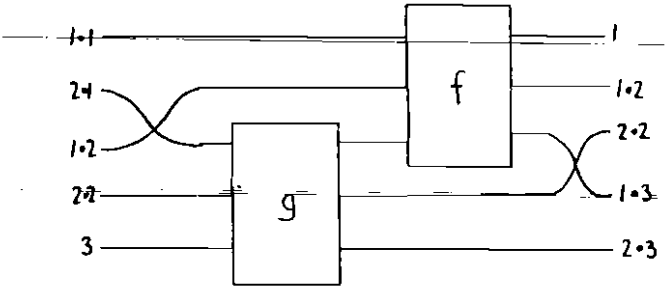


FIG 3.22

So, for an input of the form <a,<b,c>,<d,e>>, the corresponding output is of the form <<h,j>,<k,l>,m> (cf. FIG 3.21).

The horizontal composition, $\leftrightarrow$, cannot be expressed in μFP. We are forced to describe the two way communication between the cells using a recursive equation. So, in FP,

$$f \leftrightarrow g = \lambda\langle\langle a,b\rangle,c,d\rangle.$$
$$\text{letrec } \langle u,x,z\rangle = g : \langle b,c,v\rangle,$$
$$\langle w,v,y\rangle = f : \langle a,u,d\rangle$$
$$\text{in } \langle w,x,\langle y,z\rangle\rangle \qquad \text{(cf. FIG 3.21)}$$

We can think of the outputs of $f \leftrightarrow g$ as being calculated by a method of successive approximations. However, if we create a loop of dependencies, where u is absolutely dependant on v and v is absolutely dependent on u, then there is no route to the solution and we have deadlock. This mechanism is a possible model of the way in which circuits which communicate in both directions come to "agreement" about their common signals and so settle, to give stable outputs. Circuits too can deadlock or become unstable if we create an unresolved loop of dependencies.

We must now write a semantic equation defining $\leftrightarrow$ in μFP.

$$M[\![f \leftrightarrow g]\!] = zip \circ [1.2.zip \circ 3] \circ (zip \circ F \circ zip) \leftrightarrow (zip \circ G \circ zip)) \circ [zip \circ 1.2.3] \circ zip$$
$$\text{where } F = M[\![f]\!],\ G = M[\![g]\!] \qquad \text{XIII}$$

Again, we use our matrix transpose function to ensure that each subprogram receives input of the appropriate shape. This definition is written in terms of the FP version of $\leftrightarrow$, and so, we must avoid creating deadlock when we cause cells to communicate in both directions. In many systolic arrays, a data stream flows in one direction, while a result stream flows in the opposite direction. In that case, since the data never depends on the result, there is no danger of deadlock. For instance, the systolic correlator which is taken as the example in chapter 5, is of this form. One can conceive, however, of more complicated "handshaking" mechanisms which, while being deadlock free, require several "iterations" to calculate the results of $f \leftrightarrow g$

The algebraic law, B5, describes the effect of "flipping" f⟷g about a vertical axis, as shown in FIG 3.23.

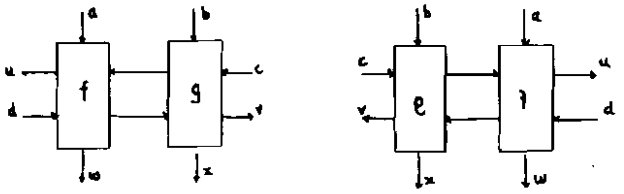


FIG 3.23

[2,1,rev*3]*((f⟷g)*[rev*1,3,2]=([2,1,3]*g*[1,3,2])⟷([2,1,3]*f*[1,3,2]) B5

If we use recrev, a recursive reverse, which not only reverses a list, but calls itself recursively on all the elements of the list which are lists, then we can rewrite B5 as

flipl*((f⟷g)*flipr = ((flipl*g*flipr)⟷(flipl*f*flipr)

where flipl = [2,1,recrev*3], flipr = [recrev*1,3,2] B5'

This law now says that flipping f⟷g over is the same as flipping f and g over separately and composing them properly. Another law about f⟷g allows us to move "detachable" circuit segments from the f part to the g part, or vice versa.

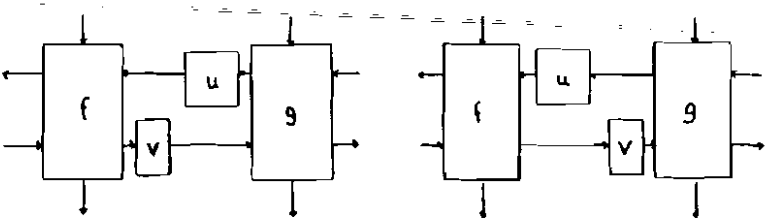
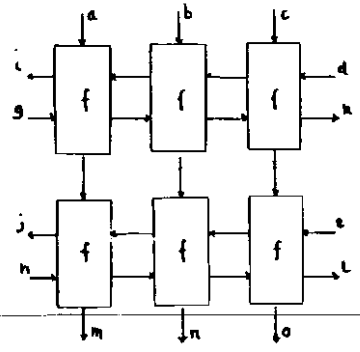


FIG 3.24 ([1,v*2,3]*f)⟷([u*1,2,3]*g = (f*[1,u*2,3])⟷(g*[1,2,v*3]) B6

Such a transformation may allow us to simplify the circuits on either side of the ⟷. Our final CF, \\, forms a grid of cells, as shown in FIG 3.25.



for inputs of the form <<a,b,c>,<d,e>,<g,h>>, outputs are of the form <<i,j>,<k,l>,<m,n,o>>

FIG 3.25 \\f

As with \, we will add \\ first to FP, and then to μFP. We have already defined ⟷ in FP, and ↑ is defined in FP exactly as it is in μFP. Then,

1) \\f:<<x,y>,<z>> = α[id]:f:<x,y,z>

2) \\f:<<x1,..xn>,<y1,..ym>,<z1,..zm>> =

[concat°1,concat°2,3]°(\\f↑\\f)°[1,[[1],tl]°2,[[1],tl]°3]:

<<x1,..xn>,<y1,..ym>,<z1,..zm>>

if m > n

[1,2,concat°3]°(\\f⟷\\f)°[[mst,[last]]°1,2,3]:<<x1,..xn>,<y1,..ym>,<z1,..zm>>

if n > m

The second definition uses a recursive "divide and conquer" technique, as illustrated in FIG 3.26.

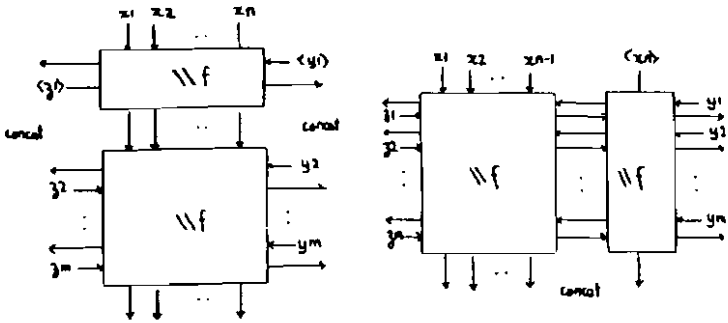


FIG 3.26 \\f in terms of f \\f in terms of ←

The semantic equation defining \\ in μFP is

M[\\f] = zip • αzip • \\(zip • M[f] • zip) • αzip • zip XIV

This has exactly the same form as the equation for \. As usual, we use zip to ensure that each subprogram gets input of the correct type. The algebraic law B7, for \\, is analagous to B4 for \

αarev • \\f • αarev = \\(arev • f • arev) B7

Our final algebraic law relates \ and \\. It is illustrated in FIG 3.27.

~~(\(f•(1,rev•2)•g•(rev•1,2)))•[1,[2],[3,]]=\\(([1•1,2•1,2]•(f•g)•[1,[2,3]]))•[1,[2],[3]]~~B8

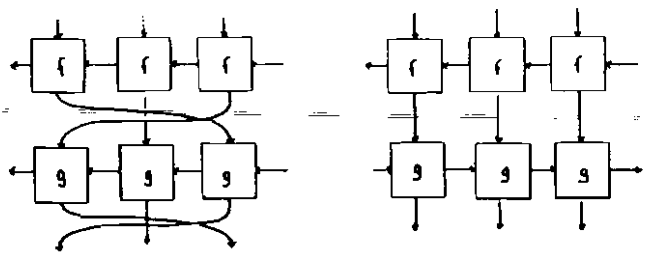


FIG 3.27

By eliminating the two reverses, we have introduced bidirectional data flow. It is a question of whether we "divide" the circuit up horizontally or vertically. In the systolic correlator circuit of chapter 5, we find that it is easiest to consider the two horizontal slices of the circuit separately at first. This allows us to introduce two dimensional data flow in a relatively painless way. However, in the second stage of the design, it is useful to consider the circuit as a row of identical cells. This freedom to decompose the circuit in a number of different ways is a useful feature of the new combining forms. In the following two chapters, we give some examples of the use of simple μFP and of the new combining forms.

Chapter 4: Case Studies: Some Small Examples

The Tally Circuit

The tally function has n inputs and n+1 outputs. The kth output (starting from 0) is to be high, and all other outputs low, if k of the inputs are high [Mead, Conway 80]

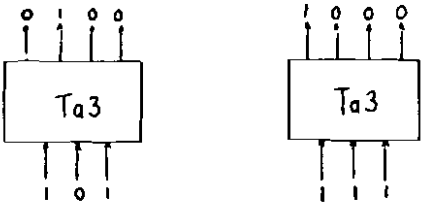


FIG 4.1 Two examples of the behaviour of a 3 input tally

We can see that the 1 in the output divides the rest of the output into two groups of 0s. The number of 0s in the lefthand group is the same as the number of low inputs to the tally. The number of 0s in the righthand group is the same as the number of high inputs to the tally. The outputs are numbered from right to left. Naturally, the total number of 0s is n.

The basic cell used in the tally circuit is TA.

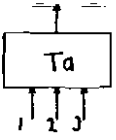
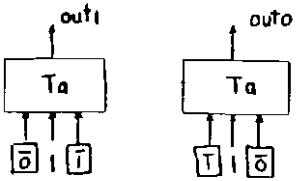


FIG 4.2 TA = 2 → 3; 1

TA is a stateless function. Let us use two TA cells to construct a one input tally.




```
inv = id → ō: ī

Ta1 = [TA*[ō,1,ī], TA*[ī,1,ō]]

    = [1, inv * 1]
```

FIG 4.3 A 1 bit tally circuit

The input is a list containing one bit. The selector function 1 is used to pass that bit to the two TA cells. The lefthand TA produces a 1 if the bit is 1, and a 0 otherwise. That is, it passes the bit through. The righthand tally inverts the bit.

A two input tally is more interesting. We add three TA cells to Ta1 to make it

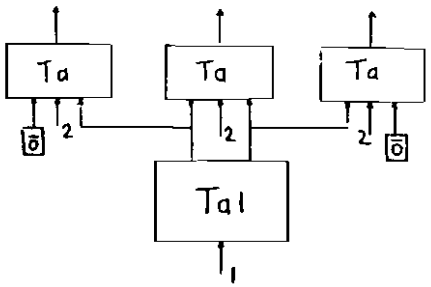


FIG 4.4 A 2 bit tally circuit

Ta2 = [TA * [$\overline{0}$,2,1*1], TA * [1*1,2,2*1], TA * [2*1,2,$\overline{0}$]]*[Ta1*[1],2]

= [2 → 1; $\overline{0}$, 2 → inv * 1; 1, 2 → $\overline{0}$; inv * 1]

This seems reasoneble. The functions calculated are

out2 = 1^2 out1 = (1^~2) v (~1^2) out0 = (~1^~2)

where ^ is logical and, v is logical or and ~ is negation.

Finally, we can make a 3 input tally as shown in FIG 4.5, and the picture begins to become clear

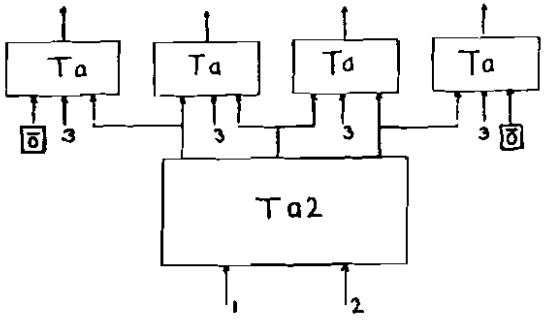


FIG 4.5 A 3 input tally circuit

Ta3 = [TA*[$\overline{0}$,2,1*1], TA*[1*1,2,2*1],

TA*[2*1,2,3*1], TA*[3*1,2,$\overline{0}$]]*[Ta2*[1,2],3]

= [3→(2→1;$\overline{0}$);$\overline{0}$, 3→(2→inv*1;1);(2→1;$\overline{0}$),

3→(2→$\overline{0}$;inv*1);(2→inv*1;1), 3→$\overline{0}$;(2→$\overline{0}$;inv*1)]

So,

out3 = 1 ^ 2 ^ 3 (the leftmost output)

out2 = (1 ^ 2 ^ ~3) v (1 ^ ~2 ^ 3) v (~1 ^ 2 ^ 3)

out1 = (1 ^ ~2 ^ ~3) v (~1 ^ ~2 ^ 3) v (~1 ^ 2 ^ ~3)

out0 = ~1 ^ ~2 ^ ~3,

as we might expect.

We would like to describe the generic form of this circuit. Let us try a recursive description (cf. FIG 4.6).

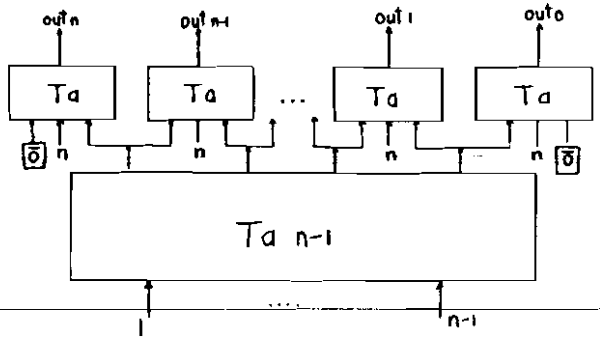


FIG 4.6 An n input tally constructed from an n-1 input tally

We explained above that the output of a tally can be considered to be two groups of 0s separated by the single high output. The number of zeros on the right gives the number of high inputs, the number on the left the number of low inputs. We would like to build an n input tally from an n-1 input tally. If the nth input is high, we must add a 0 to the right of the output of the n-1 input tally, as we have increased the number of high inputs by one. If the extra input is low, the 0 must be added to the left of the output of the smaller tally. The tally circuit constructs the two possible outputs and uses the final input to select between them. A series on n+1 TA cells is used to make the selection. Each TA has the final input as its middle (or selecting) input. The list containing 0 followed by the output of the smaller tally is spread along the lefthand inputs of the TA cells. The righthand inputs receive the output of the n-1 bit tally followed by 0. Thus, the row of TA cells selects between the two possible outputs, giving us an n input tally.

From the diagram, we can see that

$Ta(n) = \alpha TA * zip * [apndl * [\overline{0}, Ta(n-1) * most],$

$[n,n,...,n],$

$apndr * [Ta(n-1) * most, \overline{0}]]$

where $most = reverse * tl * reverse$

We use zip to form the list of 3 element inputs and then we apply TA to each of these 3-lists to give the result. The basis of this recursive definition is very simple.

$$Ta(0) = [\overline{1}]$$

The list has no 0s since there are no inputs.

The recursive definition is interesting, but it would be nice to give a more FP-like description. $/\!\!\backslash$ may be used to eliminate the recursive call. We can view the circuit as shown in FIG 4.7, in which case, a tally circuit of any size ($n > 1$) would be given by

$$/\!\!\backslash f * apndl * [[\overline{1}], id]$$

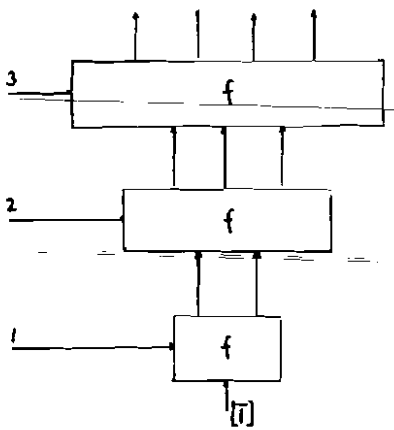


FIG 4.7 The tally circuit as $/\!\!\backslash f * apndl * [[\overline{1}], id]]$

We need to find a function which describes the operation of the boxes in the diagram. One possibility is

f = α(TA°[1°1, 2, 2°1])°distr°(zip°[apndl°[$\overline{0}$,1], apndr°[1,$\overline{0}$]], 2).

This may seem rather complicated, but it is closely related to the recursive definition given above. f takes as input the last of the bits being tallied and the result of tallying the rest of the bits. It then "calculates" the appropriate inputs to the extra string of TA cells necessitated by the last bit, just as in the recursive case shown in FIG 4.6 As as example, let us try working out a 2 bit tally

Λt ° [[$\overline{1}$], 1, 2] =

f ° [f ° [[$\overline{1}$], 1], 2] =

f ° [α(TA ° [1°1,2,2°1]) ° distr ° [[[$\overline{0}$,$\overline{1}$],[$\overline{1}$,$\overline{0}$]],1], 2] =

f ° [(TA ° [$\overline{0}$, 1, $\overline{1}$], TA ° [$\overline{1}$, 1, $\overline{0}$]], 2] =

f ° [[1, inv°1], 2] =

α(TA ° [1°1,2,2°1])°distr°[[[$\overline{0}$,1],[1,inv°1],[inv°1,$\overline{0}$]], 2] =

[TA ° [$\overline{0}$, 2, 1], TA ° [1, 2, inv°1], TA ° [inv°1, 2, $\overline{0}$]] =

[2 ⟶ 1: $\overline{0}$, 2 ⟶ inv°1; 1, 2 ⟶ $\overline{0}$; inv°1]

which is the two input tally that we had before. So, we have given a non-recursive description of a tally circuit of arbitrary size. The circuit is, of course, stateless and could just as easily be described in the original FP language. In the next section, we will describe the Muller C element, which is a simple circuit with state.

The Muller C element - a rendezvous, join or last-of circuit

The output of a C element becomes 1 only after all of its inputs have become 1. It becomes 0 only after all of its inputs are 0. The C element acts as a latch, responding to the "last of" a set of signals which change in the same direction. FIG 4.8 gives an example of the behaviour of a C element

in1	0	0	0	1	1	0	1	1	0	1	0	0	0	1	1	1	1	0	0	0
in2	0	1	1	1	0	0	1	0	0	1	1	1	0	0	1	1	1	1	0	1
out	0	0	0	1	1	0	1	1	0	1	1	1	0	0	1	1	1	1	0	0

time →

FIG 4.8 Example behaviour of a C element

C elements are used in self-timed circuits. We describe a 2 input C element as

$$C = \mu[(eq \cdot 1 \rightarrow 1 \cdot 1;\ 2),\ (eq \cdot 1 \rightarrow 1 \cdot 1;\ 2)]$$
$$= \mu[[id,\ id] \cdot (eq \cdot 1 \rightarrow 1 \cdot 1;\ 2)].$$

In this circuit, the *output* function is the same as the *next state* function. We want the output to respond immediately to the inputs, and the more usual form, $\mu[2, \ldots]$ would introduce unwanted delay into the circuit. The conditional $(eq \cdot 1 \rightarrow 1 \cdot 1;\ 2)$ expresses the fact that the state and the output can change only when the inputs (1) are equal.

We can connect several C elements together as shown in FIG 4.9 merely by writing A C.

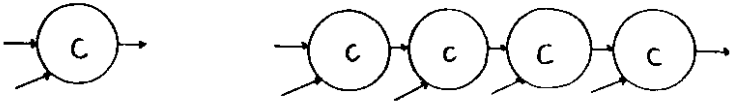


FIG 4.9 Connecting C elements together

The number of C elements is determined by the number of inputs. The circuit acts as a C element (for any number of inputs) provided that its inputs "transit only to the condition complementary to the output" (Mead, Conway 80 p. 24). If the condition is not obeyed, the circuit does not behave as a C element. Consider, for example, a three input circuit made of two C elements. Let us start with inputs and states all zero. If we now change the two inputs to the first C element to 1, both its state and its output become 1. That output is fed into the second C element, with the third input. Since the third input is still 0, the state and output of the second C element remain at 0. Now, if we change one of the first two inputs back to 0 and the third input to 1, both the state and the output of the second C element flip to 1. This means that the output of the whole circuit is 1 despite the fact that all three inputs have never been high at the same time. It was the fact that we changed one of the inputs back to 0 while the output was 0 that caused the problem.

Let us write out in full the description of our 3 input circuit.

$$\Lambda\ C \circ [1, 2, 3] = C \circ [C \circ [1, 2], 3]$$

$$= \mu[f, f] \circ [\mu[f, f] \circ [1, 2], 3]$$

(where $f = (eq \circ [1 \circ 1, 2 \circ 1] \rightarrow 1 \circ 1; 2)$)

$$= \mu[f, f] \circ \mu[[f, 3 \circ 1] \circ [1, 2]]$$

(since $[\mu[e, g], h] = \mu[[e, h \circ 1], g]$ for h stateless)

$$= \mu[f \circ [[f, 3 \circ 1] \circ [1, 2 \circ 2], 1 \circ 2], [f \circ [[f, 3 \circ 1] \circ [1, 2 \circ 2], 1 \circ 2], f \circ [1, 2 \circ 2]]]$$

(by $C\mu$)

$$= \mu[(eq \circ [(eq \circ [1 \circ 1, 2 \circ 1] \rightarrow 1 \circ 1; 2 \circ 2), 3 \circ 1] \rightarrow 3 \circ 1; 1 \circ 2),$$
$$[(eq \circ [(eq \circ [1 \circ 1, 2 \circ 1] \rightarrow 1 \circ 1; 2 \circ 2), 3 \circ 1] \rightarrow 3 \circ 1; 1 \circ 2),$$
$$eq \circ [1 \circ 1, 2 \circ 1] \rightarrow 1 \circ 1; 2 \circ 2]]$$

$$= \mu[g, [g, eq \circ [1 \circ 1, 2 \circ 1] \rightarrow 1 \circ 1; 2 \circ 2]]$$

(where $g = ((1 \circ 1 = 2 \circ 1 = 3 \circ 1) \vee (1 \circ 1 \neq 2 \circ 1 \wedge 2 \circ 2 = 3 \circ 1) \rightarrow 3 \circ 1; 1 \circ 2)$)

We have switched to infix notation for legibility. The expression has, as one might expect, a two bit state. However, our condition, that inputs transit only to the complement of the output, tell us that if the first two inputs are not equal, then both states must equal the output. Hence,

$$1 \circ 1 \neq 2 \circ 1 \implies 1 \circ 2 = 2 \circ 2$$

where 1 is the input and 2 is the state. So, we can reduce the function g (which appears inside a μ) to

$$(1 \circ 1 = 2 \circ 1 = 3 \circ 1) \rightarrow 3 \circ 1; 1 \circ 2$$

So, our expression becomes

$$\mu[1 \circ 1 = 2 \circ 1 = 3 \circ 1 \rightarrow 3 \circ 1, 1 \circ 2, [1 \circ 1 = 2 \circ 1 = 3 \circ 1 \rightarrow 3 \circ 1; 1 \circ 2, 1 \circ 1 = 2 \circ 1 \rightarrow 1 \circ 1; 2 \circ 2]]$$

It is clear that the second bit of state is superfluous, since the output does not depend on it in any way. This allows us to reduce our expression to

$$\mu[1 \circ 1 = 2 \circ 1 = 3 \circ 1 \rightarrow 3 \circ 1; 2, 1 \circ 1 = 2 \circ 1 = 3 \circ 1 \rightarrow 3 \circ 1; 2]$$

$$= \mu[[id, id] \circ (1 \circ 1 = 2 \circ 1 = 3 \circ 1 \rightarrow 3 \circ 1; 2)],$$

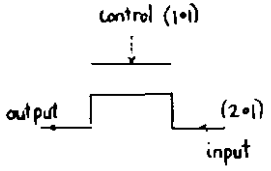
which is a three input C element.

In the next section, we move to a lower level, to describe pass transistors and inverters in the manner of Gordon [Gordon 81, 82].

Pass transistors and inverters

As in [Gordon 81], we use a third "boolean" value, θ, which indicates a value that is "floating" between 0 and 1. Then, a pass transistor, PT, is given by

$$PT = \mu([or * [1*1, and * [eq * [1*1, \theta], 2]] \rightarrow 2*1; \theta), 1*1]$$



That is, if the control input is high, or if control is floating and the state is high, the output of the pass transistor is its input. Otherwise, its output is floating. The new state is always the input. This is a very simple model of a pass transistor, but it is sufficient for our purposes. More complex models of circuit behaviour could also be embodied in μFP.

A pullup transistor is pulled low only when the input is low. It is given by

$$PU = eq * [id, \overline{0}] \rightarrow \overline{0}; \overline{1}.$$

Ground is just a source of zeroes.

$$GND = \overline{0}$$

We can describe the inverter shown in FIG 4.10 by combining our three definitions

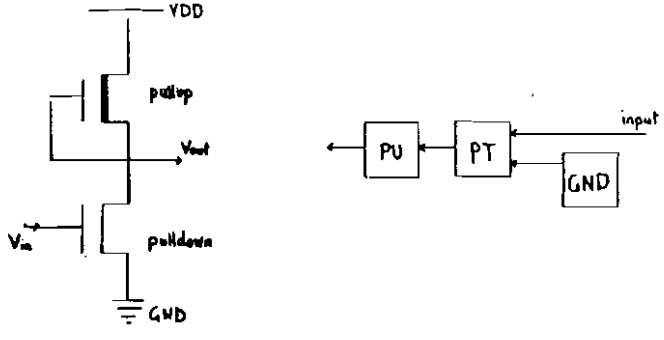


FIG 4.10 Making an inverter from a pullup and a pass transistor

INV = PU • PT • [id, GND]

PU • PT =

(eq•[id,$\overline{0}$] → $\overline{0}$; $\overline{1}$) • μ[(or•[1•1,and•[eq•[1•1,$\overline{0}$],2]] → 2•1; $\overline{0}$), 1•1]

= μ[(eq•[id,$\overline{0}$] → $\overline{0}$; $\overline{1}$)•(or•[1•1,and•[eq•[1•1,$\overline{0}$],2]] → 2•1; $\overline{0}$), 1•1]

(since f • μ[g, h] = μ[f • g, h])

= μ[(or•[1•1,and•[eq•[1•1,$\overline{0}$],2]] → (eq•[2•1,$\overline{0}$] → $\overline{0}$; $\overline{1}$); $\overline{1}$), 1•1]

⇒

PU • PT • [id, GND] =

μ[((or•[1•1,and•[eq•[1•1,$\overline{0}$],2]]→(eq•[2•1,$\overline{0}$]→$\overline{0}$;$\overline{1}$);$\overline{1}$), 1•1)•[[1,$\overline{0}$],2]]

(since μf • g = μ(f • [g • 1, 2]))

= μ[(or • [1, and • [eq • [1, $\overline{0}$], 2]] → $\overline{0}$; $\overline{1}$), 1] = INV

If we extend our boolean operator not, such that not : θ = 1, we can see that

$$INV = \mu[eq \bullet [1, \overline{0}] \rightarrow not \bullet 2; not \bullet 1, 1]$$

This is the inverter with "memory" which is much used in [Mead, Conway 80].

A two-dimensional shift register

We have become more familiar with the process of circuit design by designing and laying out a very simple integrated circuit. For our first effort at IC design, we thought it wise to choose a very simple problem, but one which would introduce us to some of the problems which have to be faced in designing more complicated chips. So, we wanted to design something which was slightly more complicated than the Mead and Conway shift register cell, and which was in the form of a two-dimensional systolic array. The obvious choice was a two-dimensional shift register, that is, one which could Shift a bit Across or Down according to a control signal. The basic cell is called SADcell. SADcells can be placed in an array, as shown in FIG 4.11.

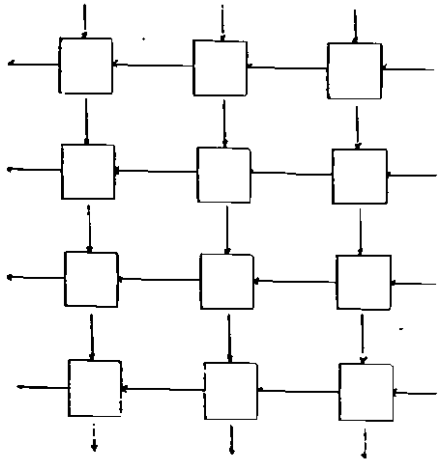


FIG 4.11 A 3 by 4 array of SADcells

Let us first describe SADcell in μFP. Each cell must accept (and pass on) the two control signals, SA and EN. When EN (ENable) is high, shifting is enabled. When it is low, each cell is to retain its datum by refreshing itself. When SA (Shift from Above) and EN are both high, all cells read from above, and shifting is from top to bottom. When SA is low, and EN high, all cells read from the right, and shifting is from right to left. Each cell has two data inputs and two data outputs, as shown in FIG 4.12.

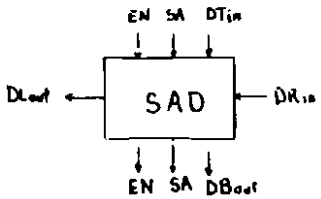


FIG 4.12 SADcell

So, the inputs to SADcell are in the form ‹‹‹EN, SA›, DTin›, DRin› and the outputs are in the form ‹DLout, ‹‹ENout, SAout›, DBout››. The cell must have one bit of state. Hence,

SADcell = μ[[2,[1*1*1,2]], (1*1*1→(2*1*1*1→2*1*1;2*1);2)]

In the *next output* function, both data bits (DLout and DBout) come from the state, and the control bits are just the appropriate part of the input. The *next state* function indicates that, if ENable is low, the state remains the same, and if EN is high, the state is set either to DTin or to DRin, depending on the value of SA.

If we use DTin, EN, etc. to represent the appropriate selection functions, and if we consider only the data outputs, SADcell becomes

μ[[2, 2], EN → (SA → DTin; DRin); 2]

This makes its function more clear. FIG 4.13 shows our implementation of SADcell in terms of pass transistors and inverters.

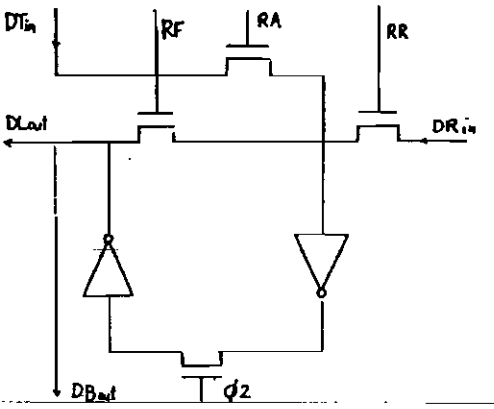


FIG 4.13 An implementation of SADcell using pass transistors

It uses a two phase clock ($\phi 1$ and $\phi 2$).

The logic equations for the control signals are as follows :—

RF = ¯EN ∧ $\phi 1$	refresh
RA = EN ∧ SA ∧ $\phi 1$	read from above
RR = EN ∧ ¯SA ∧ $\phi 1$	read from right

The fourth pass transistor is controlled by $\phi 2$. It is clear that only one of the four control signals is high at any time. This allows us to understand the circuit by analysing the four cases. Using the very simple switch model of pass transistors, the four circuits are as shown in FIG 4.14.

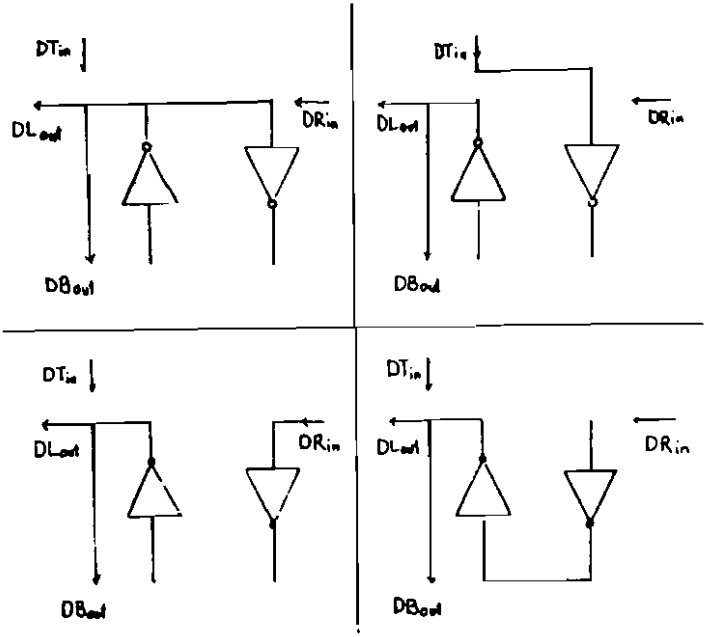


FIG 4.14 The 4 possible "configurations" of SADcell

All of these circuits have two-bit state, one bit for each inverter. All of the outputs are of the form ~(1 ° 2), that is, the inverse of the lefthand state bit. (We store information on the input gate capacitence of inverters.) The four circuits differ, however, in their *next state* functions. When RF is high, we can see that the *next state* function is [1°2, not°1°2] since the righthand inverter is refreshed by the lefthand one. When RA is high, the *next state* function is [1°2, DTin] since a data bit is "read" into the input of the righthand inverter. Similarly, when RR is high, the *next state* is [1°2, DRin]. Finally, when $\phi 2$ is high, the *next state* is [not°2°2, 2°2]. Following this analysis, we write SADcell as

μ[[not°1°2, not°1°2),

$\phi 2 \rightarrow$[not°2°2,2°2];RF$\rightarrow$[1°2,not°1°2];RA$\rightarrow$[1°2,DTin);[1°2,DRin]]

Obviously, we would like to check that our two phase clock version is an acceptable implementation of the original SADcell. One clock cycle consists of $\phi 1$ going high (with $\phi 2$ low), followed by $\phi 2$ going high (with $\phi 1$ low). Since we know that $\phi 1$ and $\phi 2$ always have this pattern, we can modify our equation for SADcell accordingly. We make $\phi 1$ our "master" clock and we assume that each time $\phi 1$ goes high, $\phi 2$ goes high immediately afterwards. Thus, we make $\phi 2$ "invisible" and since we had the conditional $(\phi 2 \rightarrow (not*2*2,2*2))$, we compose [not * 2, 2] with each of the remaining three possibilities. (The 2 which has been dropped just stands for the state.) SADcell then becomes

μ[[not*1*2, not*1*2],

not*EN→[not*2,2)*[1*2,not*1*2];and*[EN,SA]→[not*2,2]*[1*2,DTin];

not*SA→[not*2,2]*[1*2,DRin],2)

=

μ[[not*1*2, not*1*2],

not*EN→[1*2,not*1*2];SA→[not*DTin,DTin];

not*SA→[not*DRin,DRin];2)

which reduces to

$$\mu[(2,\ 2),\ EN \rightarrow (SA \rightarrow DTin,\ DRin);\ 2)$$

This is our original SADcell and we have shown that our implementation using a two phase clock is a satisfactory one. The model of the pass transistor as a simple switch is, of course, far from the physical reality. However, the method which we have just used could be applied to many circuits which are simple combinations of inverters and pass transistors, giving us a new way of understanding their operation.

If we wish to make a grid of SADcells, as shown in FIG 4.11, we simply write \SADcell.

This concludes our chapter of small examples. In the following chapter, we will present a larger case study, the systolic correlator.

Chapter 5: Case Studies: The Main Example

The Systolic Correlator

Introduction

In this chapter, we will give a step by step derivation of a systolic correlator circuit. The final circuit is an orthogonally connected systolic array of simple processing elements. While the processing elements themselves are simple, the behaviour of the circuit as a whole is far from obvious, because of the complex nature of the data flow. Our formal derivation of the circuit can be considered to be a proof of its correctness.

Before we proceed to the example itself, we will introduce an important idea which we use in the derivation.

The importance of "triangles of shift register cells"

In VLSI circuits, it is common for one or more of the data streams to "flow" through the circuit unchanged. The data items move from processor to processor, interacting with other data items which cross their paths.

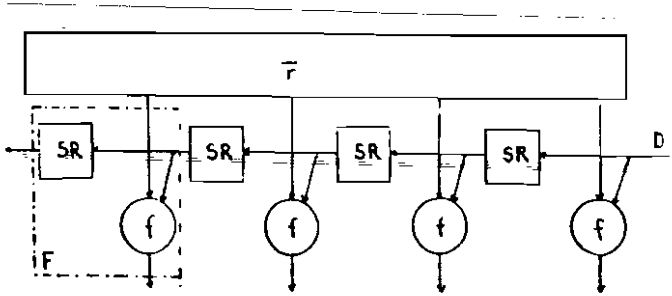


FIG 5.1 A circuit in which a data stream flows through a series of SRcells

FIG 5.1 is a simple example of such a circuit. The data stream, D, flows through a series of shift register cells (SRcells). It also interacts with elements of the vertical data stream (provided by $\overline{r}$) in the processors, to give the results. The cell F can be written as

$$[\mu[2,\ 2 * 1],\ f).$$

The circuit shown is just

$$2 * \backslash([\mu[2,\ 2 * 1),\ f]) * [\overline{r},\ [id]].$$

We assume, for simplicity, that f is a stateless function. We would like to investigate the effect on this circuit of the addition of a "triangle of SRcells". FIG 5.2 shows the new circuit.

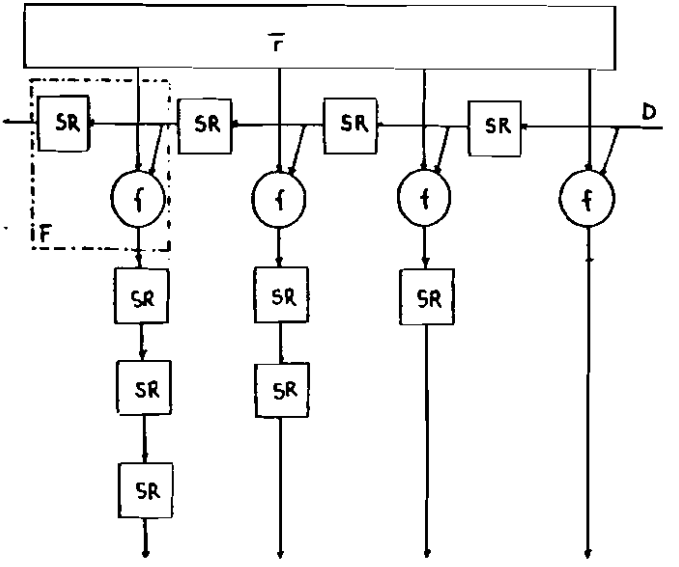


FIG 5.2 $[triang * 2 * \backslash([\mu[2,\ 2 * 1],\ f]) * [\overline{r},\ [id]]$

We call the triangle of SRcells ltriang where

ltriang * (1, 2, . . N) = (SRN-1 * 1, SRN-2 * 2, . . SR1 * (N-1), N)

(SRN = SR * SRN-1 SR (⇐ SR1) = μ(2, 1))

Let us assume that our input is the stream ‹x1, x2, x3, . . .›. Then, FIG 5.3 is a "freeze frame" of the circuit in action, with the arrows labelled with values.

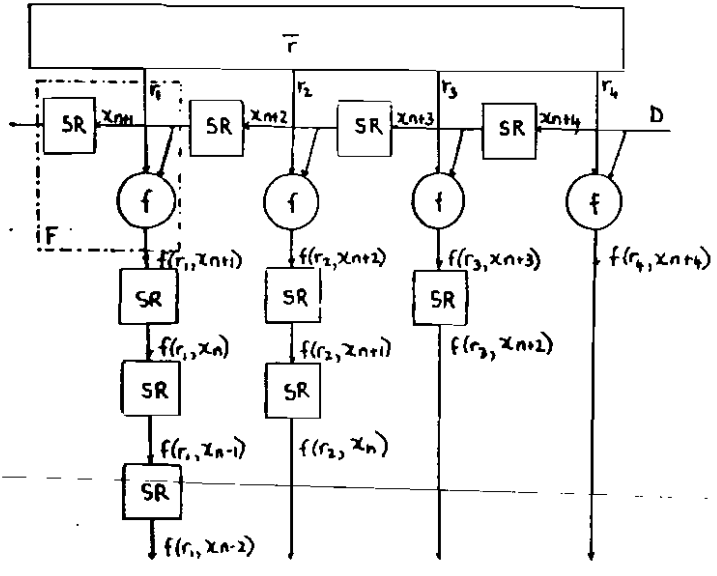


FIG 5.3 Snap-shot of the circuit in action

While the original circuit, 2 * \F * [r, [id]], gives as its output

‹f(r1, xn+1) f(r2, xn+2), f(r3, xn+3), f(r4, xn+4)›,

the new circuit gives

‹f(r1, xn-2), f(r2, xn), f(r3, xn+2), f(r4, xn+4)›.

The interesting point to note is that this output depends on <u>alternate</u> inputs. This is because, for every element of this output, we have doubled the number of SRcells through which the relevant data must pass. For example, the x input to the leftmost processor passes through 3 SRcells, and the addition of ltriang forces the output through 3 more SRcells.

If we again freeze our circuit after one more clock cycle, we find that the relevant outputs are

«f(r1, xn+2), f(r2, xn+3), f(r3, xn+4), f(r4, xn+5)» and

«f(r1, xn-1), f(r2, xn+1), f(r3, xn+3), f(r4, xn+5)».

So, it can be seen that the circuit with Hrlang is not just ignoring every second input. It "takes account of" all the inputs, but, for a given output, the relevant inputs are "two apart" in the input stream. In the next section, we will formalise these ideas.

Slowmodels: a predicate about circuits

The function ev2 takes every second element of a sequence or list, i.e. it gives the first, third, fifth, seventh, . . . elements. We say that

f slowmodels g if

ev2(F : «x1,y1,x2,y2,...xn,yn,..») = G : ev2(«x1,y1,x2,y2,...xn,yn,...»)

where F = M[[f]], G = M[[g]]. M is our "meaning function".

Knowing that f slowmodels g tells us that, if we are trying to implement g, f "will do" provided we are willing to interpose a "don't care" between each of our inputs and to have a "don't care" interposed between each of our outputs. Some theorems about slowmodels are :-

f stateless ⇒ f slowmodels f S1

SR2N slowmodels SRN S2

Since this seems almost counter-intuitive, let us take an example.

M[SR2N] : «x1, y1, . . . xn, yn» = «?, ?, ?, . . . ?, x1, y1, . . . xn, yn»

M[SRN] : «x1, x1, . . . xn-1, xn» = «?, ?, ?, . . . ?, x1, x2, . . . xn-1, xn»

So, ev2(M[SR2N] : «x1,y1...xn,yn» = M[SRN] : ev2(«x1,y1,..xn,yn»).

f slowmodels g ∧ h slowmodels j ⟹ f * h slowmodels g * j S3

Proof:

Assume ev2(F : ⟨x1, . . xn, . .⟩) = G : ev2(⟨x1, . . xn, . .⟩)

and ev2(H : ⟨x1, . . xn, . .⟩ = J : ev2(⟨x1, . . xn, . .⟩)

Then, G * J : ev2(⟨x1, . . xn, . .⟩) =

G : ev2(H : ⟨x1, . . xn, . .⟩) = ev2(F : H : ⟨x1, . . xn, . .⟩)

f slowmodels g ∧ h slowmodels j ⟹ [f, h] slowmodels [g, j] S4

f slowmodels g ⟹ /f slowmodels /g (for both /L and /R) S5

Finally, returning to the example of the previous section, we find that

ltriang*2*\[μ[2,2*1],f] * [r̄,[id]] slowmodels 2*\[μ[2,2*1],f] * [r̄,[id]].

Proof: let r̄ = [r̄1, r̄2, . . r̄N]

ltriang * 2 * \[μ[2, 2 * 1], f] * [r̄, [id]] =

[SRN-1*f * [r̄1,SRN-1], SRN-2*f * [r̄2,SRN-2],...,f * [r̄N, [id]]] =

[SRN-2 * f * [r̄1, [id]], SR2N-4 * [r̄2, [id]], . . ,f * [r̄N, [id]]]

~~2 * \[μ[2, 2 * 1], f] * [r̄, [id]] =~~

[f * [r̄1, SRN-1], f * [r̄2, SRN-2], . . ,f * [r̄N, [id]]] =

[SRN-1 * f * [r̄1, [id]], SRN-2 * f * [r̄2, [id]], . . ,f * [r̄N, [id]]]

f stateless ⟹ f * [r̄, [id]] slowmodels [* [r̄, [id]] for any f

SR2N-2 slowmodels SRN-1

So, SR2N-2 * f * [r̄1, [id]] slowmodels SRN-1 * f * [r̄, [id]].

Similarly,

SR2N-4 * f * [r̄2, [id]] slowmodels SRN-2 * f * [r̄2, [id]] and so on.

Thus,

ltriang*2*\[μ[2,2*1],f] * [r̄,[id]] slowmodels 2*\[μ[2,2*1],f] * [r̄,[id]].

This result concerning the addition of "left triangles of SRcells" to circuits of a particular form will be surprisingly useful in the example which follows.

The Systolic Correlator

We want to compute

$$c(k + N) = \sum_{i=0}^{N-1} r(i) \cdot d(k+i)$$

That is, we wish to calculate the scalar product of r with each of the N element subsections of the data stream d. If we think of the reference inputs as being single bits, then we are checking each subsection of the data to see whether it is of a particular form. Only numbers in the columns which interest us will contribute to the final sum. This form of correlation is used in signal processing, for example in the recognition of radar signals of a certain type. We are assuming, for simplicity, that the reference inputs are constant. Our first attempt is the circuit shown in FIG 5.4

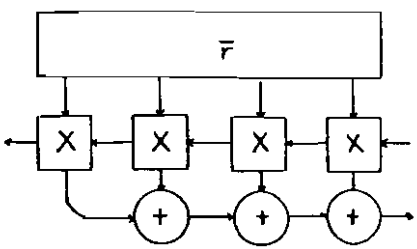
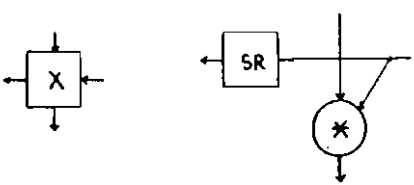


FIG 5.4 A circuit which computes /\+ * 2 * \Xcell * [r, Iid])

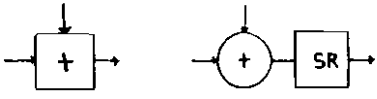
In this circuit, each of the Xcells is of the form



= [μ[2, 2 * 1], *]

where * is stateless multiplication. We use N such cells to shift the data, d, across the circuit and to do the necessary multiplications (r(i) . d(k+i) in the equation). The /\+ adds up the N results of these multiplications. The N bit wide reference data are constantly fed in at the top as shown.

This circuit, although it fulfils our specification, is not suitable for implementation on silicon because it uses a large stateless adder to give the result. This means that, in each clock cycle, some signals have to travel all the way across the chip. We would like to use two input adders with state so that, in each clock cycle, a given signal need only travel the short distance between adjacent cells. This would allow us to operate the circuit at a higher clock rate. These +cells are of the form



= SR * +

Now, we must find a way to introduce these cells into the circuit without upsetting its semantics. Obviously,

$/\!\backslash+ = /\!\backslash+ * apndl * [\bar{u}, id]$

since 0 + x = x.

So, we can introduce an "initializing zero" without difficulty. However, the introduction of the SRcells is more complicated. In FIG 5.5, we show the effect on $/\!\backslash+$ of the addition of successive SRcells.

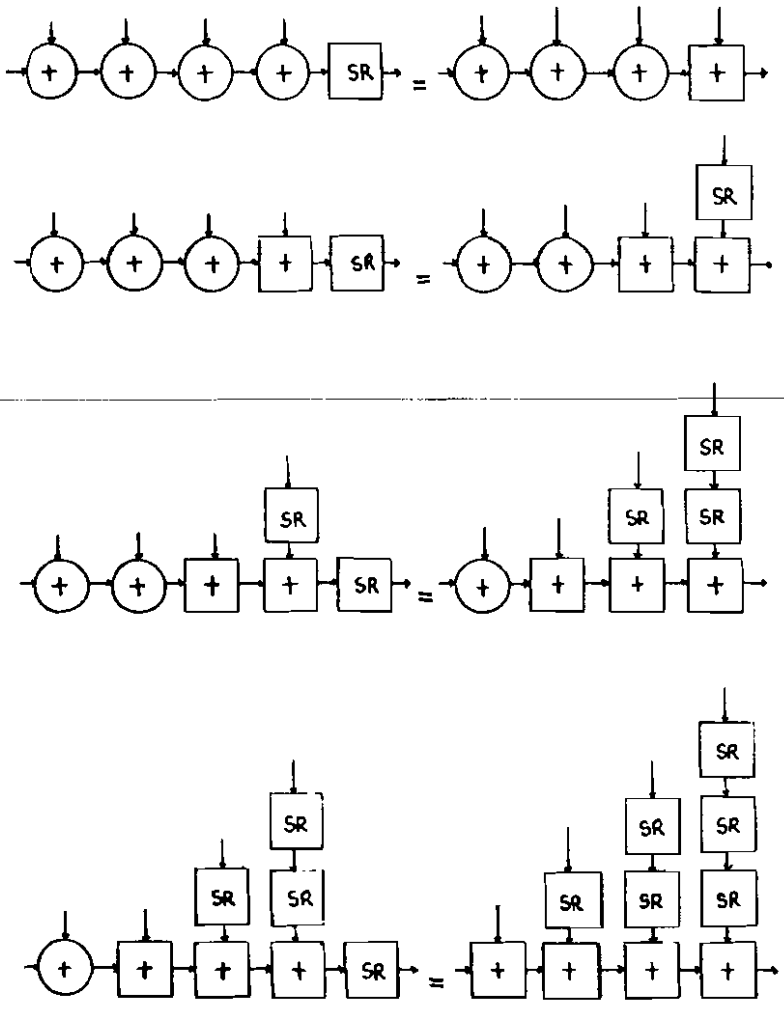


FIG 5.5 Transforming a stateless adder into one with state

The transformations are based on the fact that having an SRcell on the output of a stateless function has the same effect as having SRcells on all of its inputs. Surprisingly, we find that a "right triangle" of shift registers is introduced.

rtriang ° [1,2...N] = [1, SR ° 2, . . SRN-2 ° (N-1), SRN-1 ° N]

SRN ° /\+ ° apndl ° [$\bar{0}$, id] = /\(SR ° +) ° apndl ° [$\bar{0}$, rtriang]

Proof:

f stateless ⇒

SRN-1 ° /\f ° [1,2. .N] = SR°SRN-2 ° /\f ° [1,2.. N] =

SR ° f ° (SRN-2 ° /\f ° [1,2...N-1], SRN-2 ° N] =

SR°f ° [SR°1 ° (SRN-3°/\f ° [1,2...N-2], SRN-3 ° N-1], SRN-2 ° N] =

.

.

/\(SR ° f) ° [1, 2, SR ° 3, . . SRN-3 ° N-1, SRN-2 ° N]

Therefore,

SRN ° /\+ ° apndl°[$\bar{0}$,id] = /\(SR°+) ° apndl°[$\bar{0}$,rtriang]

So, we now know that

SRN ° /\+ ° apndl ° [$\bar{0}$, 2) ° \Xcell ° [$\bar{7}$, [id]] =

/\(SR°+) ° apndl°[$\bar{0}$,rtriang°2) ° \Xcell ° [$\bar{7}$,[id]]

and the circuit shown in FIG 5.6 does the required correlation. If we are willing to accept the fact that the circuit takes N cycles longer to produce the first result.

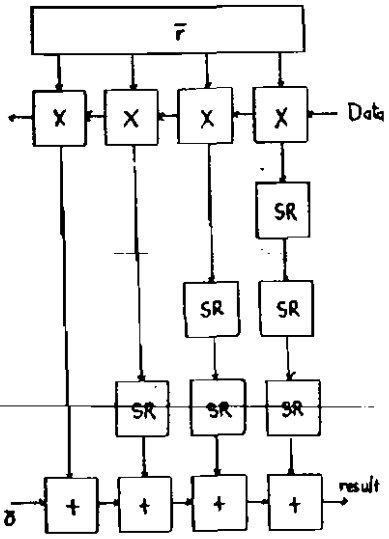


FIG 5.6 Our second attempt: a circuit in which all cells have state

The triangle of shift registers is, however, an embarrassment. Let us remove it. From an earlier section, we know that

itriang • 2 • \Xcell • [r̄,Id]] slowmodels 2 • Xcell • [r̄,[Id]] and
SR2N • Λ+ • apndl•[ō,Id] slowmodels SRN • Λ+ • apndl•[ō,Id].
Thus,
SRN•Λ(SR•+)•apndl•[ō,rtriang] slowmodels Λ(SR•+)•apndl•[ō,rtriang].
SRN • Λ(SR•+) • apndl•[ō,rtriang] • itriang•2•\Xcell • [r̄,Id]]
slowmodels
Λ(SR•+) • apndl•[ō,rtriang] • 2 • \Xcell • [r̄, [Id]] and
SRN • Λ(SR•+) • apndl•[ō,SRN-1] • 2 • \Xcell • [r̄,[Id]] slowmodels
SRN • Λ+ • apndl•[ō,2] • \Xcell • [r̄,[Id]] (cf. FIG 5.7)

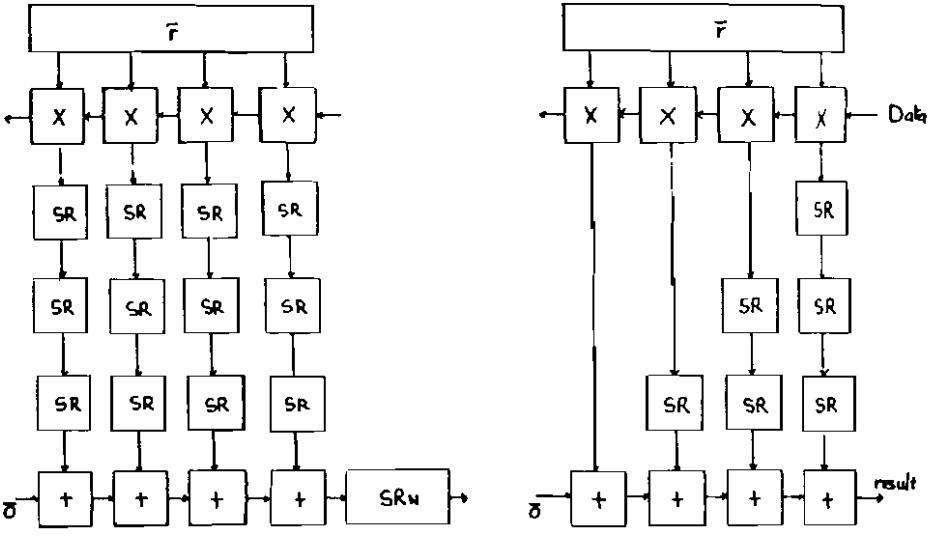


FIG 5.7 (a) slowmodels (b)

The circuit on the left has the kind of regular layout that we were looking for and we can improve it further by "moving" the bank of shift registers

$\Lambda(SR^{*}+) * apndl^{*}[\overline{a}.SRN-1] = SRN-1 * \Lambda(SR^{*}+) * apndl^{*}[\overline{a}.ld]$

$\Rightarrow$

$SR * \Lambda(SR^{*}+) * apndl^{*}[\overline{c}.2) * \backslash Xcell * [\overline{r}.ld]$ slowmodels

$SR * \Lambda+ * apndl^{*}[\overline{c}.2) * \backslash Xcell * [\overline{r}.ld] =$

This equation tells us that the circuit shown in FIG 5.8 can be used to implement the required correlation function provided that we are willing to interpose a "don't care" between each input and to have a "don't care interposed between each output.

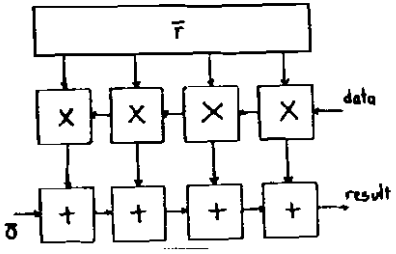


FIG 5.8 A systolic correlator

In most applications of correlation, the reference inputs are one bit wide but the data stream (and hence the result stream) may be several bits wide. We would like to refine the cells used in FIG 5.8 down to the bit level. The circuit is just a string of cells of the form shown in FIG 5.9.

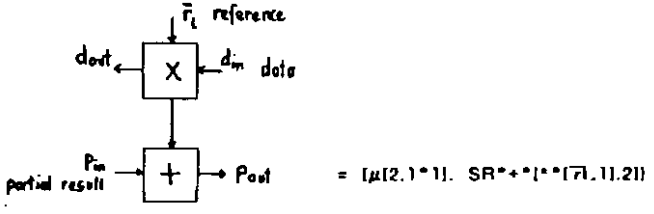


FIG 5.9 The correlator cell

Since the reference inputs are only one bit wide, the function of the cell shown reduces to

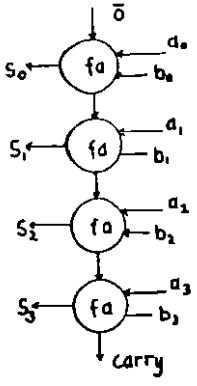
[μ[2, 1 * 1]. SR * + * [r̄] → 1; 0̄, 2]]

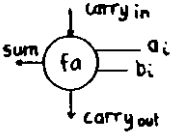
That is, because the reference input is always either 1 or 0, we no longer need the multiplier. The new function of the cell is clearer if we transform the μFP expression slightly, to give

[μ[2, 1 * 1]. SR * (r̄ → + * [1, 2]; 2)].

Now, it can be seen that the cell's main task is to perform the gated addition of the data word to the partial result.

It would be nice to reduce this cell to a vertical array of simpler cells, each of which deals with one bit of the data word and one bit of the partial result. We already know how to add two M bit numbers, using a cascade of full adders (cf. FIG 5.10)






```
fa  =  [xor*(1,xor*2),  xor*[and*2,xor*[and*[1,1*2],and*[1,2*2]]]]
sum = 1 * \fa * [[0], zip * [1, 2]]
```

FIG 5.10 Adding two M bit numbers using M full adders

In our cell, we must perform the gated addition of two numbers, so we must pass the reference input down through our vertical array, and do a gated full-addition at each stage. Also, we must pass the data word (and hence each bit) through a latch to the next cell. This leads us to the vertical array of primitive cells shown in FIG 5.11. The number of Scells must be big enough to hold the largest possible result.

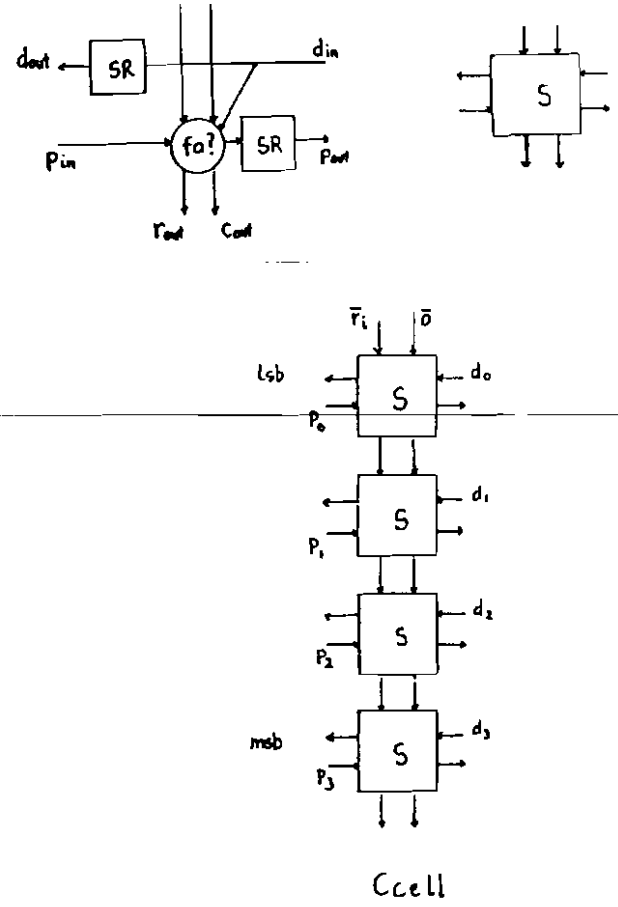


FIG 5.11 Implementing the correlator cell of FIG 5.9 using more primitive cells

```
fa? = [(1 * 1 ⟶ ta * [2 * 1, [2, 3]]; [2, 3]), 1 * 1]
Scell = [SR * 2, SR * 1 * 1 * fa?, [2, 2 * 1] * fa?]]
  = [SR * 2, SR * 1 * 1 * fa?, [1*1, 2*1*fa?]]
Ccell = \\Scell * [[[r̄ᵢ, ō]], id]
```

fa? performs gated full addition for inputs of the form ‹‹rin, cin›, din, pin›. It also passes the reference input through. [1, 2] * Ccell has the same function as our original correlator cell (cf. FIG 5.9). However, the circuit suffers from the same problem as our *first* attempt at the systolic correlator as a whole. We have connected together a series of stateless gated full adders and so, some of our signals must cover long distances in one clock cycle. Ideally, we would like all of our signals to be latched, as shown in FIG 5.12.

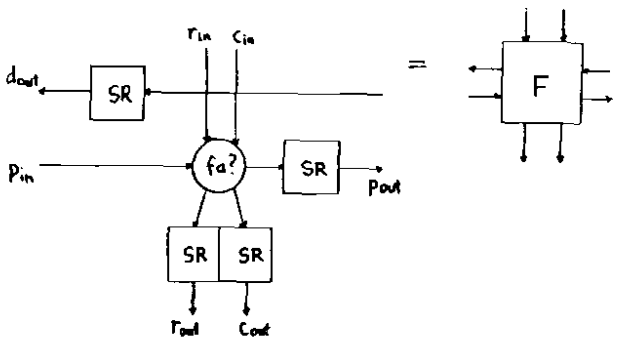


FIG 5.12 Fcell = [SR*2, SR1*1*1*fa?, [SR*1*1, SR*2*1*fa?]]

In a previous section, we analysed the effect of replacing f by SR * f in $\mathcal{A}$f, where f is stateless (cf. FIG 5.5). In this case, we are concerned with \ rather than with $\mathcal{A}$. FIG 5.13 illustrates an important equivalence and FIG 5.14 indicates how it may be proved by induction. The results from our analysis of the \ case are sufficient for our purposes, as they can easily be extended to the \\ case.

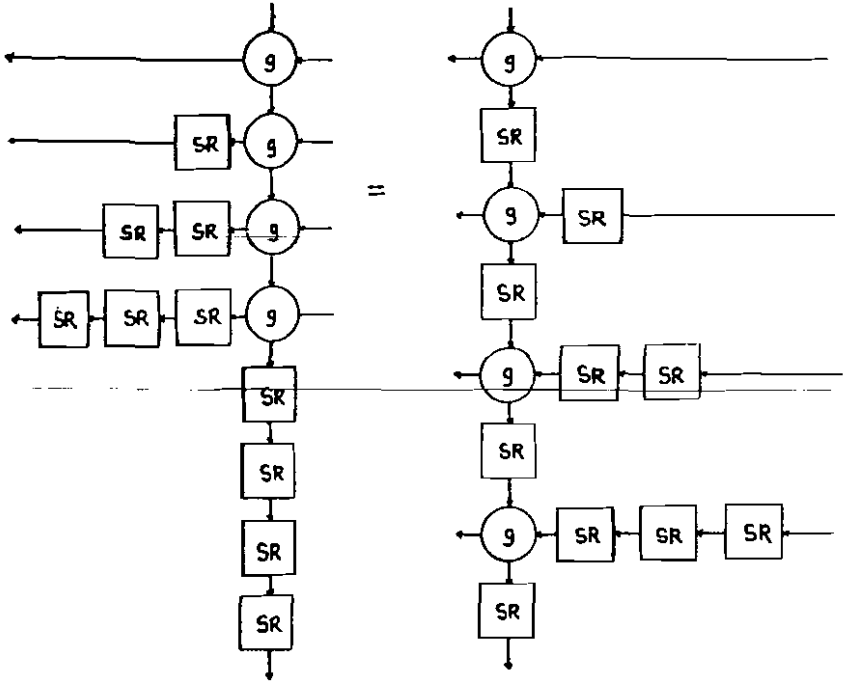


FIG 5.13 (rtriang*1,SRN*2) ∘ \g ∘ [(1),2) = \((1,SR*2)*g) ∘ [(1),rtriang*2)

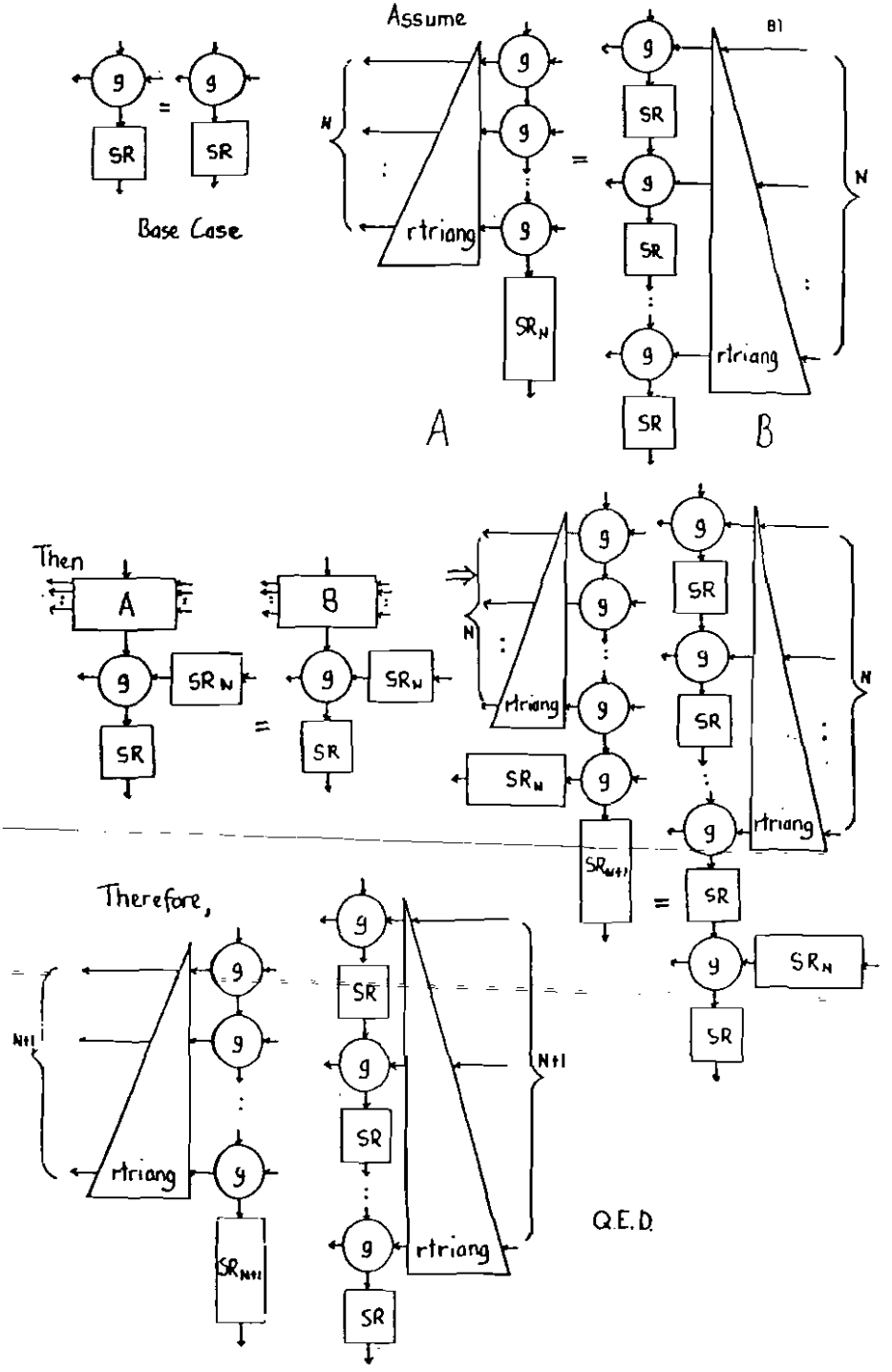


FIG 5.14 A "proof" by induction of the property illustrated in FIG 5.13

In this case, we want to replace the Scells of FIG 5.11 by the Fcells of FIG 5.12. By analogy with the example shown in FIG 5.13, we find that we must insert triangles of shift registers, as shown in FIG 5.15.

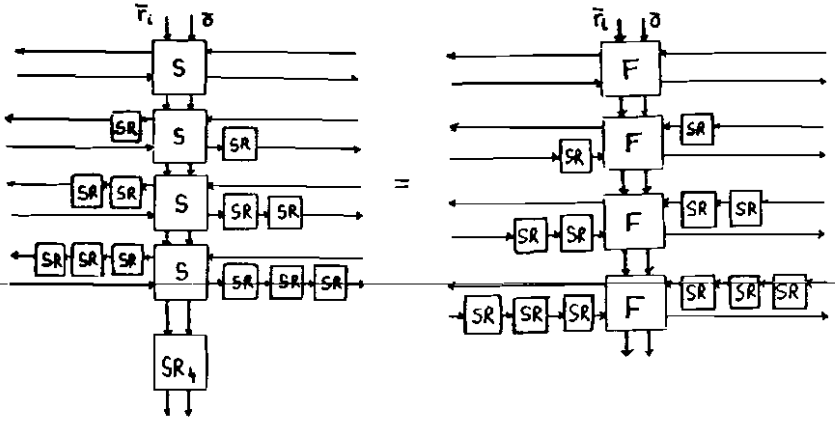


FIG 5.15 An illustration of the effect of changing from Scell to Fcell

This rather daunting diagram merely indicates that, if we want to use cells in which all the signals are latched, we must be willing to present the data and partial result inputs to the vertical array of cells in time-skew format. That is, the bits of the data and result words must, in some sense, be passed through ritriangs. This causes the bits of each word to be presented in successive clock periods (starting with the least significant bit). This method of presenting inputs to a circuit is commonly used in systolic arrays

The diagram also indicates (by the presence of the ritriangs on the lefthand side) that, if we insert the extra latches, we must expect the outputs of the vertical array of cells to be time-skewed. Neither of these results is very surprising, and neither presents any difficulty when we combine several columns of Fcells to make our final correlator circuit.

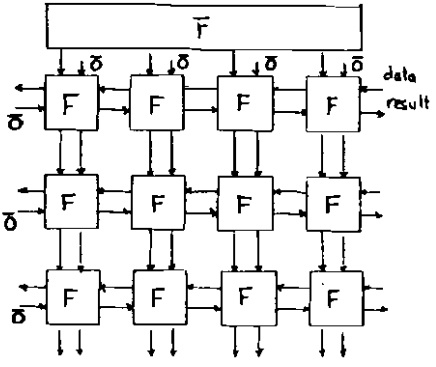


FIG 5 16 The systolic Correlator circuit

If we input our data words in time-skewed format, then the skewed data front thus created moves steadily across the circuit from right to left. It interacts with the partial result data front, which is travelling in the opposite direction, and which is also skewed. Thus, our results are produced in time-skewed format.

We must also remember that the circuit which we have defined can be used to implement the required correlation function only if we are willing to interpose a "don't care" between each input, and to have a "don't care" interposed between each output. Thus, our circuit produces results at half the circuit clock rate. It should be noted that, throughout this analysis, we have assumed the existence of an implicit master-clock. In a real circuit, a single clock would be used to control all the cells.

Conclusion

In this chapter, we have introduced two related concepts which are useful in the design and analysis of digital circuits. The concept of "triangles of shift register cells" is a remarkably powerful tool for the analysis of the behaviour of regular arrays of identical cells. The predicate slowmodels allows us to analyse circuits in which only 50% of the processors are "active" at any given time. Such circuits are common in VLSI and we now have a means of showing that a circuit of this type is "sufficient" to fulfil a given specification, provided that we are willing to have the necessary "don't cares" in the input and output streams.

We have used both these techniques to derive a simple systolic array which implements our correlation function. Systolic arrays can be used to implement many useful functions such as matrix multiplication, pattern-matching and many signal processing functions. We have shown that μFP can easily be used not only to describe such circuits but to reason about their behaviour.

Chapter 6: The simulation and layout programs

The μFP Simulator

Introduction

The denotational semantics of μFP, given in chapter 2, can be considered to be a functional program. We simply code up the meaning function, M, considering the eight cases given by equations I – VIII. This gives us a recursive function whose base case or escape clause is equation I.

i.e. f stateless ⇒ M ⟦f⟧ = αf.

In all other cases, the function calls itself recursively. The meaning function is a higher order function because it takes a μFP program and produces an FP program, which is itself a function. If we give this FP function our input stream, it will calculate the corresponding output stream, giving us a simulator. A colleague, John Hughes, has written a μFP interpreter in a purely functional language, Lispkit Lisp (Henderson, Jones, Jones 83).

We have, however, chosen to implement the operational semantics of μFP, in Pascal, using a mainly functional style. We hoped that considering the operational rather than the denotational semantics of the language would increase our understanding of it. Also, we wanted to combine the simulator with a layout program which was written in Pascal to take advantage of the only available Functional Geometry implementation.

The Pascal implementation of μFP, SIMUFP

The first input to SIMUFP is the μFP description of the system which we wish to simulate. The description is input in abstract syntax form and it encodes a tree structure. For example, [f * g, h → j; k] is represented by the tree shown in FIG 6.1.

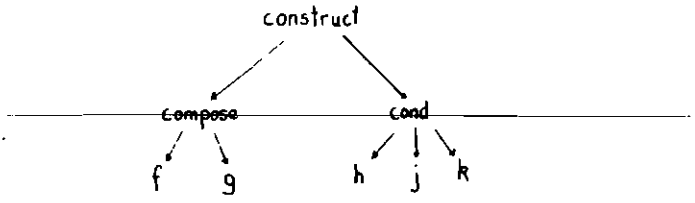


FIG 6.1 The abstract syntax tree for [f * g, h → j, k]

For each combining form, there is a Pascal function whose only task is to construct the corresponding abstract syntax tree. For example, the function for composition (*) is

```
function compose(p1,p2 : muprog) : muprog;
var p : muprog
begin
  new(p);
  with p^ do begin
    fl:=p1; fr:=p2; tag:=tcompose end;
  compose:=p
end;
```

This function takes two abstract syntax trees (or muprogs) and returns the tree representing their composition. When using μ, the user must provide the *initial state* and this is encoded in the appropriate node of the tree. This is because the shape of the state of a μ cannot always be deduced from the context.

The abstract syntax tree constructed in this way is then transformed, using the information contained in the first input to the μFP program, to eliminate all αs and /s. A μFP program containing αs or /s cannot be "solidified" into its final form until we know the shape of its inputs. Thus, αf represents [f°1, f°2, f°3] for 3-element inputs, and [f°1, f°2] for inputs which are pairs. /f represents f°[f°[1,2],3] for 3-element inputs, and f for pairs. The function trans takes a tree and an input and replaces all αs and /s by the appropriate combinations of construction and composition. Let us take some examples to show how trans works. T represents trans.

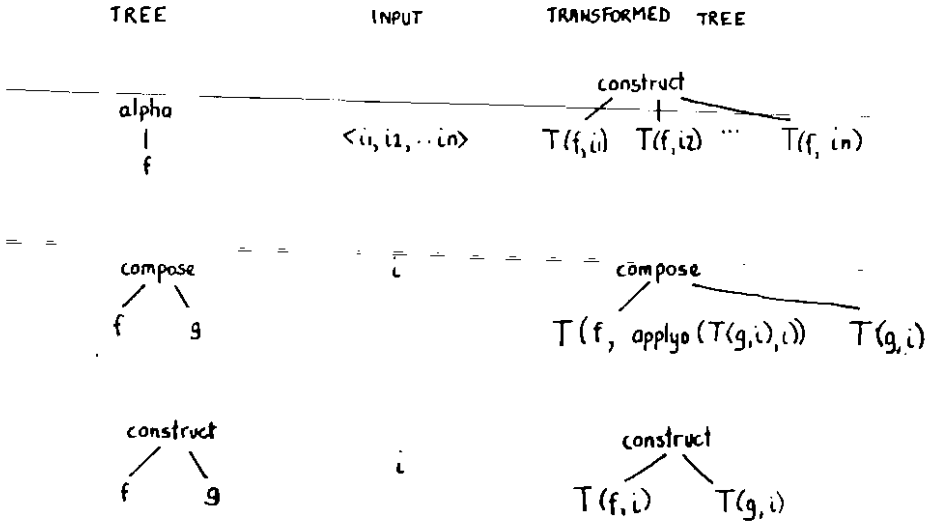


FIG 6.2 The effect of trans on abstract syntax trees for α, ° and []

For α, the transformation reflects the fact that αf=[f°1,f°2,...f°n] for n-element inputs. We must call trans recursively on each of the elements of this new construction as f might, itself, contain αs and /s. Transforming compose(f,g) with input i is slightly more complicated. The result is a composition whose right branch is trans (g,i). The left branch must be trans of f with some input. To calculate that input, we compute the output which trans (g,i) gives for input i, using the function apply0. Apply0 takes a transformed tree (or μFP program) and an input and computes the corresponding output. It will be described below. So, the left branch of the composition is trans (f, apply0 (trans (g, i), i)), as shown in FIG 6.2.

In a construction, each of the elements receives the same input and so we transform each of them with that input. Similarly for the conditional.

trans (cond(p, f, g)) = cond (trans (p, i), trans (f, i), trans (g, i)).

For basic functions (e.g. and, or, +), the selection functions and constant functions, trans (f) = f. FIG 6.3 illustrates how trees for μ and /R are transformed.

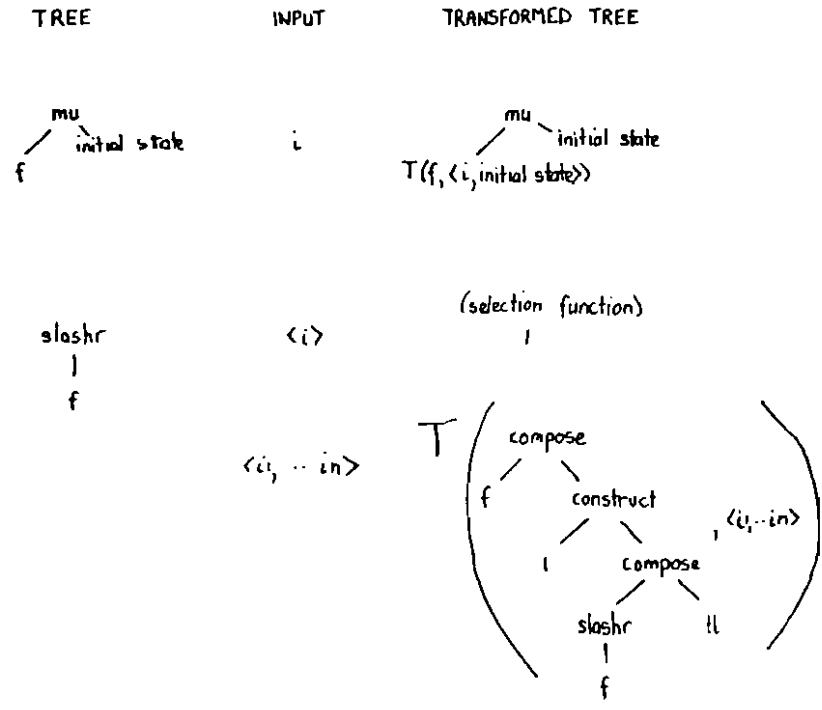


FIG 6.3 The effect of trans on abstract syntax trees for μ and /R

For μf, we transform f not only with the input i, but with the input/initial state pair. The transformation for /R relies on the FP definition :-

/Rf : $\langle x \rangle = x$, /Rf : $\langle x_1, \ldots x_n \rangle = f : \langle x_1, /Rf : \langle x_1, \ldots x_n \rangle \rangle$.

/L is treated analogously.

Once we have transformed the abstract syntax tree to eliminate all αs and βs, we can "apply" the tree to the input stream to produce the output stream, giving us our simulator. The function apply takes a transformed tree and an element of the input stream and computes the corresponding element of the output stream. It also updates the internal states of the "circuit", which are contained in the μ nodes of the tree. (The apply0 function used by the function trans above is exactly the same except that it makes no changes to the state of the tree.) Apply does a case analysis on the types of tree which it might encounter. FIG 8.4 shows how the output and the new tree (with updated states) are calculated. A represents apply.

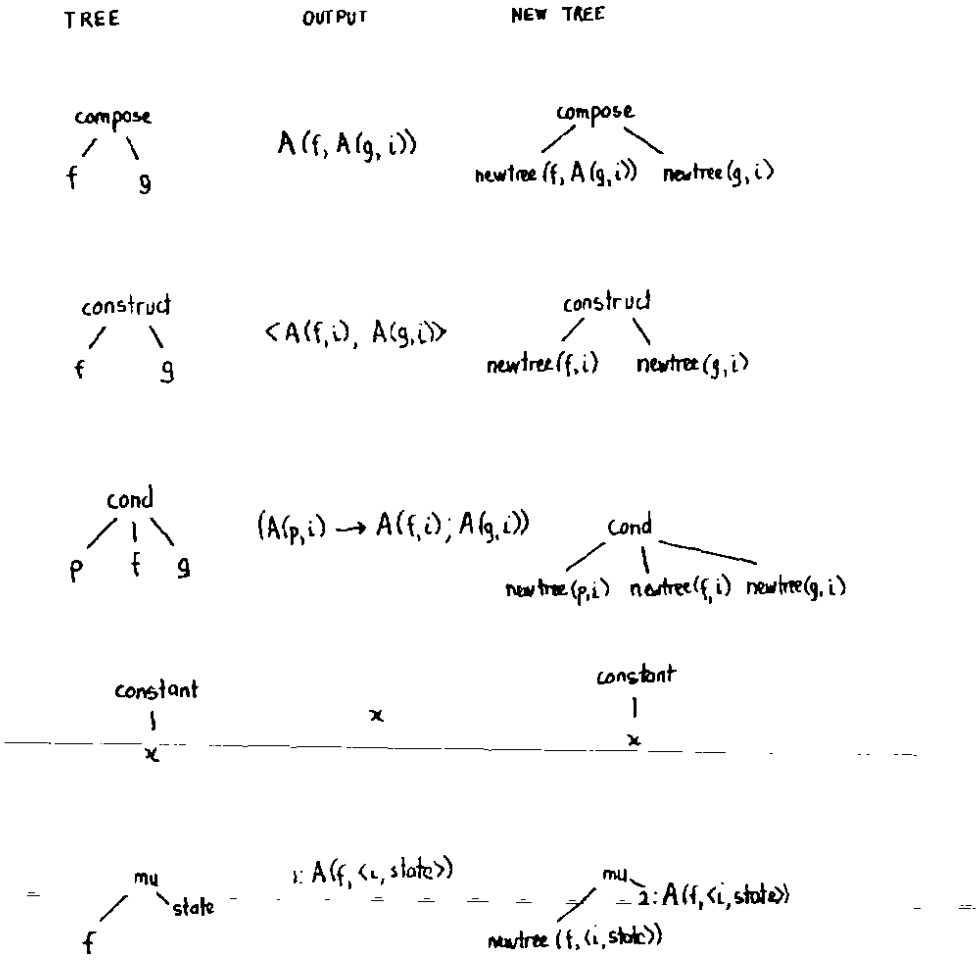


FIG 6.4 How apply calculates the output and the newtree for input i

For selection functions and basic functions, apply (f,i) is just f(i). These, together with the constant functions, form the leaves of our tree. Apply calls itself recursively, as shown, until all of the leaves of the tree have been reached, and the output is known. The tree is updated only at the μ nodes, where the new state, 2 : A(f, <i, old state>), overwrites the old state.

We have, in effect, constructed a very simple data flow machine, through which we push our inputs, one at a time. An input can change the machine in that it can cause the state stored at μ nodes to be changed. This continuously changing machine simulates the circuit represented by our μFP expression.

Thus, μFP can easily be interpreted in either a functional or an imperative language (with records) to give a simple logic level simulator. The simulator can be useful when writing the original high level specification or when designing circuit sub-blocks. It should be used as an aid to formal reasoning, rather than as an alternative to it.

In the following section, we will describe a program which takes a μFP expression and a sample input, and plots the associated "floor-plan", using the geometric interpretation of the combining forms.

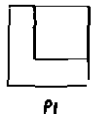
The μFP floor-plan drawing program

Introduction

In chapter 2, we gave a geometric interpretation for each of our combining forms. Thus, every μFP expression not only encodes a particular semantics but also has a particular floor-plan or layout associated with it. This very simple relationship between the semantic and geometric interpretations of a μFP expression is an important feature of the language. Many integrated circuit design languages capture information about either layout or semantics, but μFP attempts to integrate the two. Two μFP expressions with exactly the same semantics may have very different floor-plans. The half-adder of chapter 2 is a very simple example of this. The systolic correlator description of chapter 5 contains some more complicated examples of circuits which have different layouts but the same behaviour. The simulator described in the previous section "implements" the semantics of a μFP expression. The program described in this section plots the geometric interpretation of a μFP expression (on a HP plotter). The program uses Functional Geometry, which is a very simple way of describing pictures. We will first give a brief introduction to Functional Geometry and then we will describe the program itself.

A brief description of Functional Geometry

Functional Geometry (Henderson 82, Sheeran 81), which was first proposed in (Henderson 80), allows us to describe pictures easily and readably, using a small set of geometric functions. We construct our pictures using these functions and so our functions take pictures as arguments and produce pictures as results. The available functions are above, beside, rot(ate), flip and overlay. FIG 6.5 shows some examples of how pictures are combined to make more complicated pictures. The numerical arguments to above and beside give the ratios of the picture heights and widths respectively. So, in above (3,1,p3,p2), the p3 part is three times as tall as the p2 part.

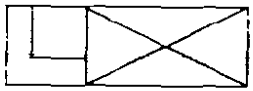


P1

P2

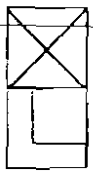
P3

P4 = rot(P2)

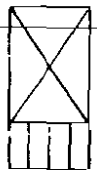


P5 = beside(1,2, P1, P3)

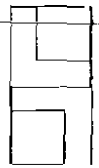
P6 = overlay(P1, P3)



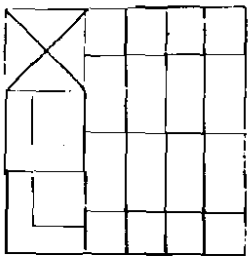
P7 = above(1,1, P3, P1)



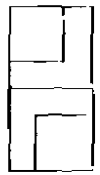
P8 = above(3,1, P3, P2)



P9 = above(1,1, P1, rot(rot(P1)))



P10 = beside(1,2, above(2,1, P7, P1), overlay(P2, P4))



P11 = flip(P9)

FIG 8.5 Examples of the use of Functional Geometry

The basic pictures, which we use as building blocks, are created using the function grid. Grid allows us to specify a picture by giving two integers, m and n, and a list of lines which are to be placed on an m by n grid. Each line is represented by a list of the form (x1 y1 x2 y2), where the coordinates of the end points of the lines are (x1 y1) and (x2 y2).

Once we have constructed our picture with these geometric functions, we can give it its final shape and size, using three vectors, as shown in FIG 6.6.

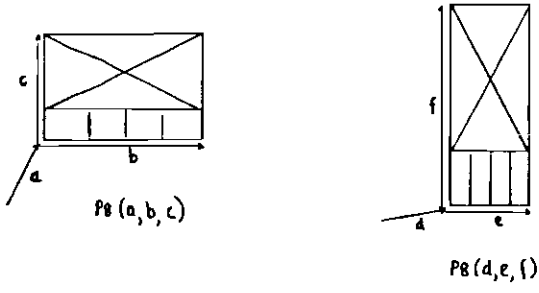


FIG 6.6 Giving pictures their final shape and size

Our Pascal implementation of Functional Geometry has been used to lay out an actual integrated circuit, as described in Appendix A. Our floor-plan drawing program uses the same Functional Geometry implementation, and so is written in Pascal

The floor-plan drawing program, GEOMUFP

In chapter 2, we gave a simple geometric interpretation for each of the combining forms. The aim of GEOMUFP is to use this interpretation to produce a floor-plan for a given μFP expression. Earlier in this chapter, we showed how any μFP expression can be represented by an abstract syntax tree. We also showed that, when we give a sample input for the circuit represented by the expression, we can transform the abstract syntax tree to get rid of all αs and /s. We use exactly the same technique for the layout program. What we wish to "plot" is the transformed abstract syntax tree. Thus, we are plotting the μFP expression for a particular input. We cannot, in general, draw a floor-plan of a circuit unless we know the type of its inputs. The floor-plan program deals only with simple μFP (as described in chapter 2) and so the combining forms in question are composition, construction, conditional, constant and mu.

Returning to Functional Geometry, it is interesting to note that a picture can be represented as an abstract tree, with the basic pictures created using grid at the leaves. For example, the picture p10 of FIG 6.5 has the abstract syntax tree shown in FIG 6.7.

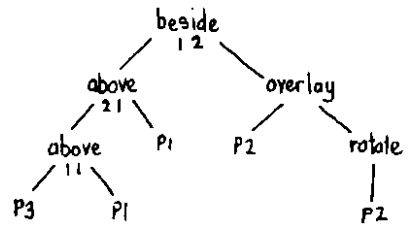


FIG 6.7 The abstract syntax tree for the picture p10 (cf FIG 6.5)

The abstract syntax tree of a μFP expression can be thought of as representing a picture in a similar way. The leaves of the tree, which are constant functions or basic functions, are represented by boxes with the function name inside and with "wires" attached to represent the inputs and outputs. One wire carries one atom, which could be a boolean value or a large integer, depending on the types used in the μFP expression. Selector functions, which are also leaves, correspond to wires. The floor-plan is meant to be abstract and wires are represented by single lines. However, users are asked to give width and height to their basic function boxes, where the unit of measurement is the width needed by a single wire. A "1 by 2" AND gate is drawn as shown in FIG 6.8.

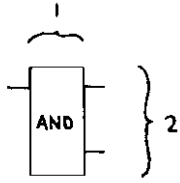


FIG 6.8 A 1 by 2 AND gate

A 1 by 1 AND gate would not be allowed as there would not be enough "room" for both the inputs. All our basic pictures have an associated width / height pair and, in any plot of a circuit, a particular basic function always has the same shape and size. This allows the user to gain some information about the eventual shape and size of the circuit. It is more realistic than the alternative approach in which the circuit is made to fill all the available space, sometimes causing the same function to be represented by boxes of widely different sizes

Now that we know how to represent the leaves of our tree, we can consider the combining forms at the nodes to be geometric functions which take a number of pictures and produce a new picture. The analogy with functional geometry continues when we consider function composition. To plot f * g with input i, we plot g with input i and f with input g:i and we place the two pictures <u>beside</u> each other, joining up the lines in the middle (cf. FIG 6.9).

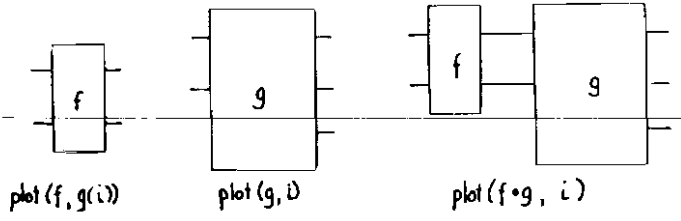


FIG 6.9 Function composition - pictures are placed <u>beside</u> each other

On the other hand, construction causes pictures to be placed <u>above</u> each other, as shown in FIG 6.10.

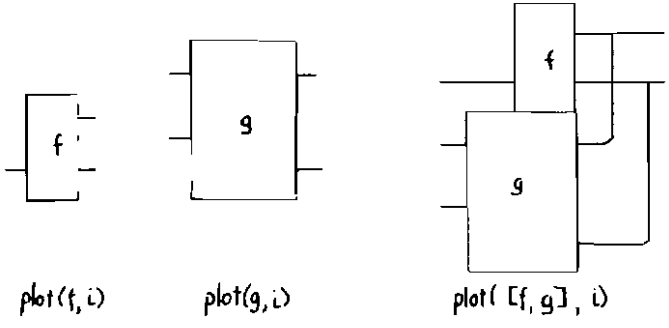


FIG 6.10 Construction causes pictures to be placed <u>above</u> each other

The inputs must be passed to *all* the elements of the construction. For the conditional, (p ⟶ f; g), we make a construction of the pictures for f, p and g and we use a switch "box" to represent the choice between the outputs of f and g, depending on the output of p.

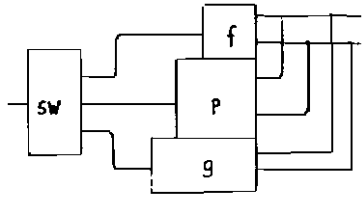


FIG 6.11 The plot for (p ⟶ f; g)

The last remaining combining form is μ. To plot μf, we plot f and then pass the second element of the output through a "latch" of the appropriate size, into the second element of the input.

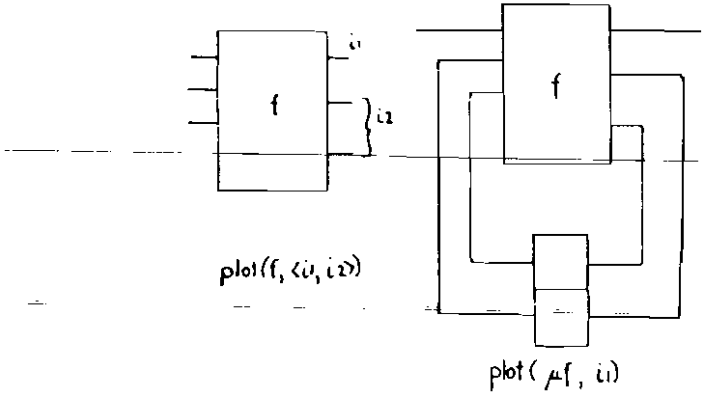


FIG 6.12 The floor-plan for μf

The selection functions are represented in the obvious way. FIG 6.13 gives some examples.

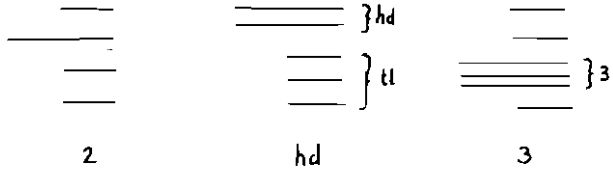


FIG 6.13 The selection functions pass on the appropriate wires

So, a μFP expression can represent a picture. The leaves of the abstract syntax tree are known pictures whose widths and heights are known. We combine these pictures, using the geometric interpretation of the combining forms, to build up a picture of the final circuit.

To facilitate the connecting of wires, every circuit plot is represented by three Functional Geometry pictures, two edge pictures showing the inputs and outputs and e main picture showing the circuit itself. The edge pictures have associated with them information about the type of the input or output and they "remember" the edge pictures from which they have been constructed. This information is used when we join edge pictures and when we select particular parts of an edge picture. For selection functions, the output edge picture is easily obtained from the input edge picture and there is no need for a main picture (cf. FIG 6.13). FIG 6.14 shows how our compound pictures are combined both horizontally and vertically. We place Nil pictures as required.

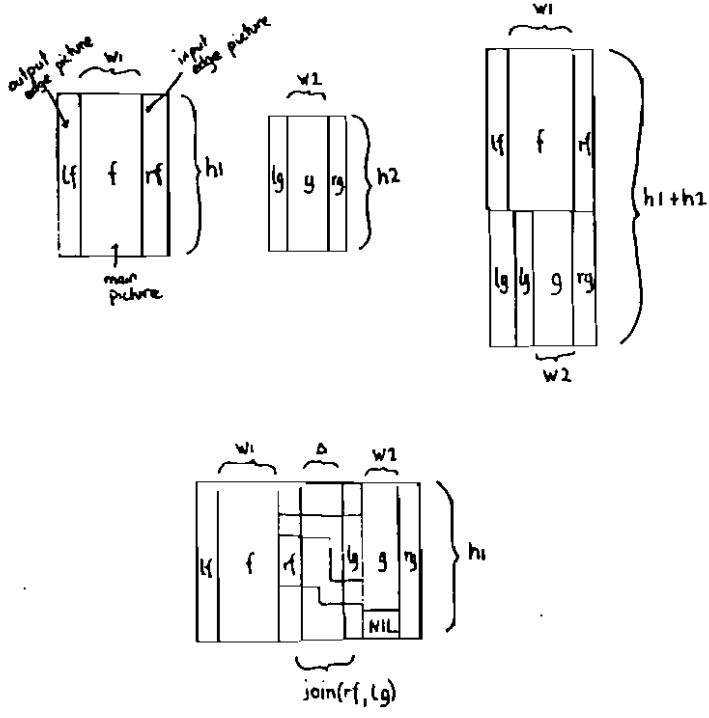


FIG 6.14 Combining compound pictures horizontally and vertically

This method allows us to implement the geometric interpretation of the combining forms as illustrated in FIGs 6.9 - 6.13. Thus, we can plot any simple μFP expression for a particular input type. If the user doesn't wish to plot the whole circuit in detail, he can supply a picture for any subsection of the circuit, overwriting what would otherwise have been plotted. This allows him to draw the floorplan to the required level of detail

The program which we have just described drives a Hewlett Packard plotter. FIGs 6.15 - 6.17 are some examples of the output which it produces

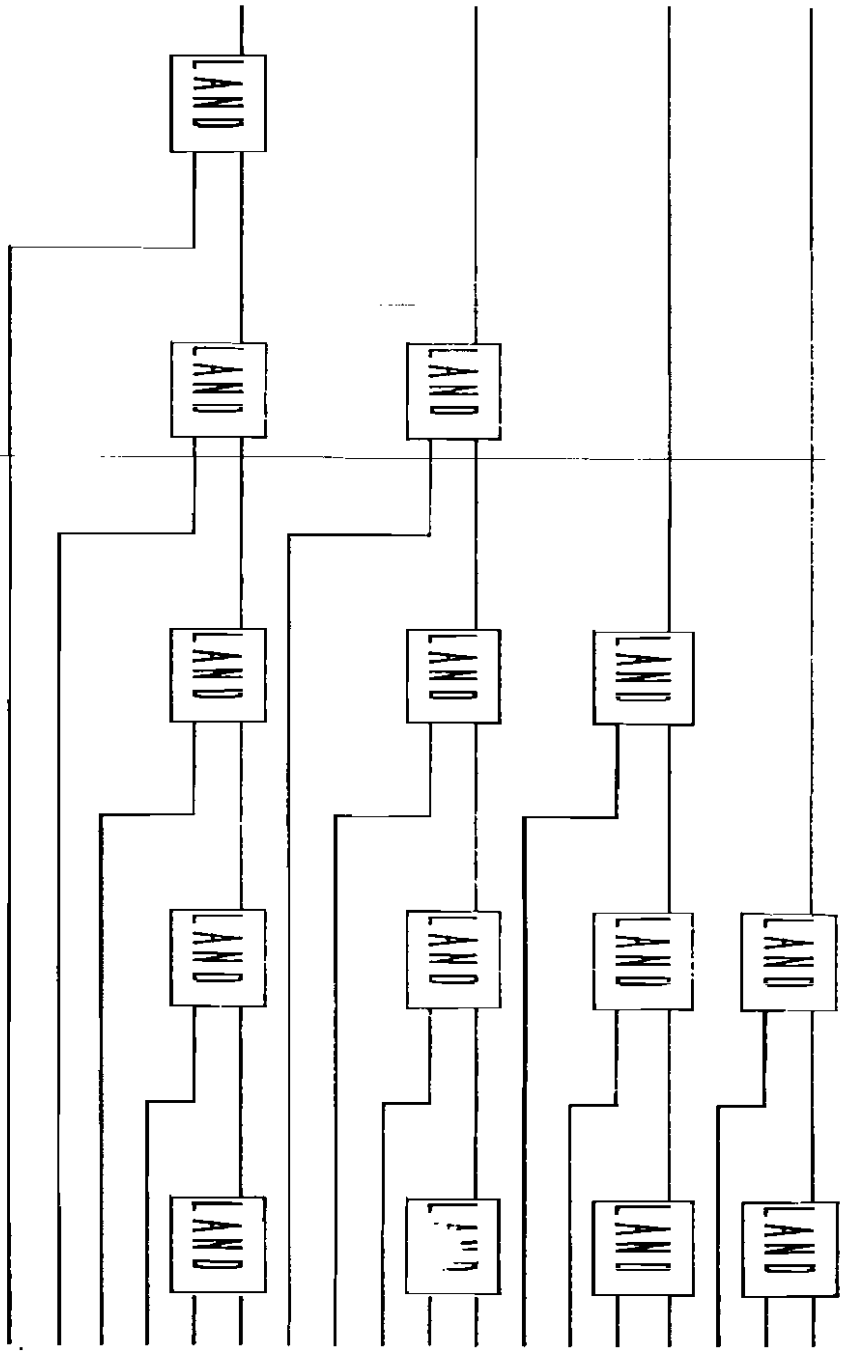


FIG 6.15 d / L LAND input shape <<1,1,1>,<1,1,1,1>,<1,1,1,1,1>,<1,1,1,1,1,1>>

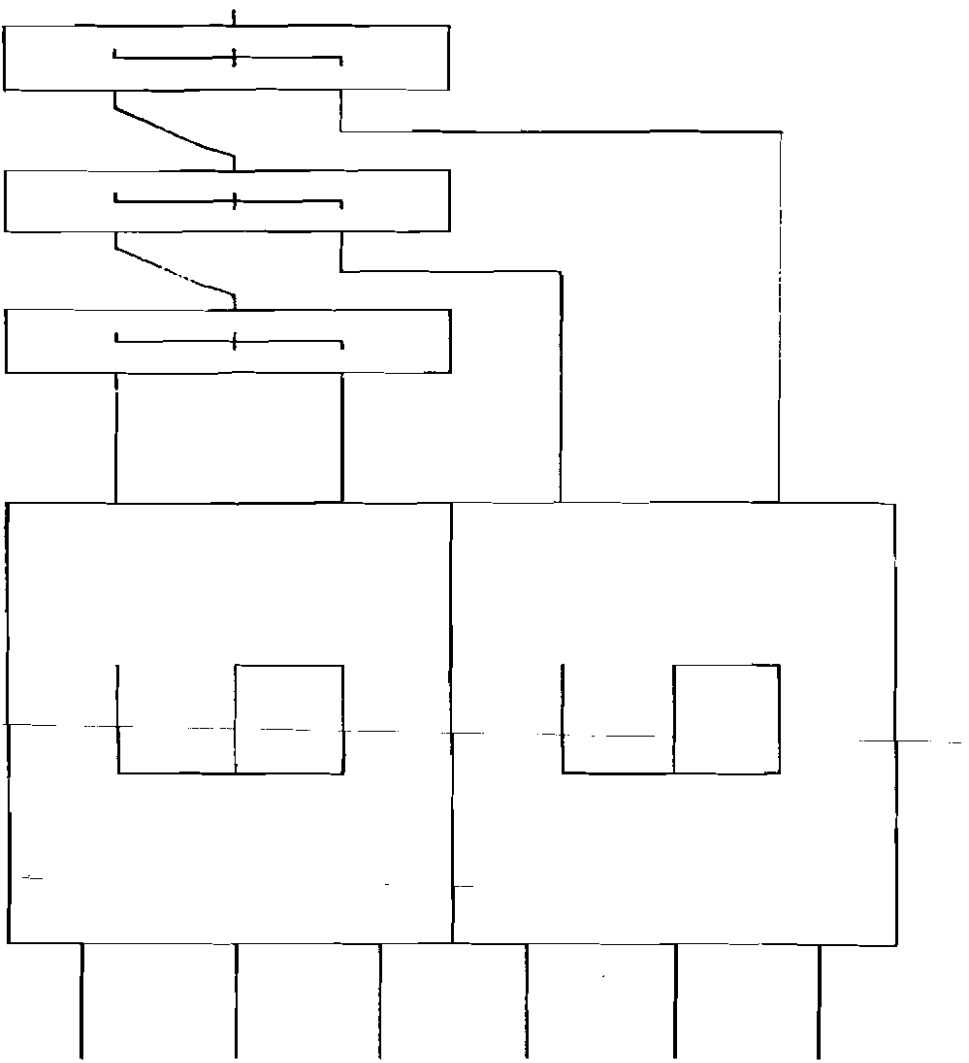


FIG 6.16 $/_R f \circ \&_y$ input shape $\langle\langle 1,1,1\rangle, \langle 1,1,1\rangle\rangle$

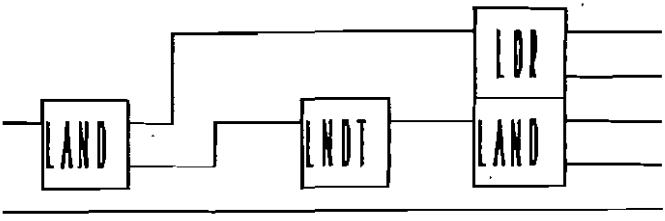


FIG. 6.17 [LAND ∘ [LOR ∘ 1, LNOT ∘ LAND ∘ 2], 3]

input shape ⟨⟨1,1⟩, ⟨1,1⟩, 1⟩

Chapter 7: Related Work and Discussion

Introduction

In this chapter, we review recent developments in the field of VLSI design which are relevent to this thesis. Until a few years ago, VLSI was strictly the domain of microelectronic engineers and trained circuit designers. The revolution came with the publication of [Mead, Conway 80]. The simple structured design methodology presented in [Mead, Conway 80] allowed "outsiders" (particularly computer scientists) to "discover" VLSI. The realisation that hierarchical design methods are vital to the management of complexity in VLSI systems came at a time when the older ad hoc methods were floundering. Although the VLSI community has, in general, accepted the need for hierarchical design methods, most work in the area of VLSI design is concentrated at the lowest end of the hierarchy, the layout level. In the first section, we give a brief survey of the available design tools, ranging from "automated graph paper" systems for cell layout to silicon compilers.

Although a great deal has been written about VLSI design, only a tiny fraction of the literature is concerned in any way with the use of formal methods. In the second section, we turn our attention to design languages as distinct from design tools. We discuss the important properties of a good VLSI design language and we compare our aproach to that of other workers in the field. We argue the need for a mathematical approach. The software industry is slowly accepting the need for formal methods and we contend that only by the use of such methods will integrated circuit designers avoid a "VLSI crisis".

The reader seeking a good introduction to VLSI design in general is referred to [Mead, Conway 80] and to [Clark 80].

Design Tools

Most of the design tools currently available concentrate on the lowest end of the design hierarchy, layout. Layout systems remove much of the tedium of the traditional "graph paper and coloured pencils" method. Many systems use an interactive graphics display so that the user is always aware of the "shape" of the design. Graphic design languages tend to be popular with circuit designers, who are more used to drawing shapes than to writing programs. Textual design (or layout) languages, on the other hand, may be more powerful than their graphical equivalents. They allow the use of parameterization, which is very important, since the aim of structured design methodologies is to produce highly regular structures.

Early design systems were textual, and described the geometry rather than the topology of the circuit (for example CIF [Mead, Conway 80; Hon, Sequin 80], ICL [Ayres 79] and LAP [Locanthi 78]). GAELIC [Boyd 79], the simple symbolic layout language which is the standard input to SERC mask-making facilities, can be produced using either an ordinary text editor or an interactive graphics editor. The ICARUS system is geometry based, but has a graphics interface, using a "mouse".

Buchanan and Gray [Buchanan, Gray 79] have devoloped a SIMULA-based system where a design is defined in terms of blocks. Blocks have physical, structural and behavioural descriptions, which must be consistent. A block is a collection of connected components, which may be other blocks or primitive components such as transistors or wires. Components are connected together by nodes. Since these nodes have both physical and structural significance, they have been named coordinodes. Blocks may be parameterized, allowing the deformation of cells and the conditional inclusion of circuit elements. The system provides facilities for logic and timing simulation and for electrical and dimensional design rule checking.

More recently, Buchanan [Buchanan 82] has proposed SCALE, which is a range of VLSI design languages. LARGE SCALE is an explicit description of both the structural and physical attributes of the circuit. SMALL SCALE concentrates on the structural features. The range from LARGE to SMALL SCALE gives a set of languages requiring varying degrees of automated layout. The system makes use of the idea of coordinodes developed in the earlier work.

Stick diagrams, which were first proposed in [Williams 77] and which are described in [Mead, Conway 80], can be used to express the topology or connectivity of a circuit. STICKS [Williams 78] is a graphical compiler for high level VLSI design. REST [Mosteller 81] is the prototypical leaf cell design system, using the connectivity approach. The input to REST is just a simple colour sketch or rough stick diagram of the leaf cell. The sketching is done on a graphics terminal. The REST process then produces a compacted sticks representation, which has a unique physical representation. The leaf cells thus designed are then composed textually, using the SPAM language [Segal 80]. The composition cells which are used to combine other cells contain behavioural as well as structural data and a multi-valued functional simulator is used to check the functional correctness of the chip.

The Daedalus/DPL design environment [Shrobe 83] combines both text and graphics. The DPL layout language [Batali, Mayle, Shrobe, Sussman, Weise 81] uses parametric cell definitions and symbolic descriptions and it manipulates a hierarchical object-oriented data base. The whole system is embedded in LISP, allowing the user to define his own functions. Daedalus is a graphics editor which allows one to edit graphics objects, but which outputs DPL code. The whole system was used in the design of the Scheme chip at M.I.T. [Steele, Sussman 80].

Sticks & Stones [Cardelli, Plotkin 81] is a language which is designed to express the hierarchical structure and topological properties of VLSI circuits. The language can be used to specify and communicate stick diagrams in texual form. A more concrete form, which includes the necessary geometric details, has been implemented [Cardelli 81]. The abstract data type "picture" and the algebraic operators on it have been embedded in a general purpose applicative programming language, giving a powerful chip assembly tool. (We will return to the work of Cardelli in the next section.)

Pooh [Whitney, Mead 83] is a symbolic "sticks-like" representation for circuit level designs and a set of algorithms which operate on this representation. The Pooh system maintains connectivity, circuit schematic and port placement information. It defines an automatically checked representation for circuit level designs and it allows mask geometry to be automatically generated. The user interface may be either graphical or language based. Individual structures are "forced" to obey the layout rules by construction. Some analysis is needed to ensure the correct spacing and angles between adjacent structures. The Pooh language system has been embedded in the programming language Mainsail (TM). The system has been used as the base-level representation in the Siclops silicon compiler project [Hedges, Slater, Clow, Whitney 82].

The ASTRA CAD system [Revett, Ivey 83] being developed by British Telecom aims to support the design of high-complexity integrated circuits. It encourages and supports a structured approach to design, with the aim of managing interconnection efficiently from the earliest stages of the design. The system uses floor-plans to define the topology of the layout and the hierarchical structure of the design. A form of symbolic layout is used to design low level cells and geometric layout is automatically generated. The floor-plans have the role of composition-cells when the leaf-cells are finally pliced. When the chip is being assembled, leaf-cells are automatically streiched to ensure that pitch matching is maintained. The experimental system is currently being applied to a realistic design example.

The Berkeley Building-Block system (BBB) [Chen, Hsu, Kuh 83] is a hierarchical automated layout system for IC design which is designed to interface to other design aids through a general purpose data base. The Silicon Assembler LUBRICK [Schoellkopf 83] has been developed at Grenoble as part of a silicon compiler project. It allows the hierarchical design of functional cells using basic interconnection structures. The CHIPMASON automatic layout program [Wu, Parker, Conner 83] operates recursively on a tree-like structure which represents the design floor-plan including implementation alternatives. The output is mask data.

Two systems which combine textual and graphical representations with the use of both connectivity and geometry are Electric [Rubin 83] and MULGA [Weste, Ackland 81; Ackland, Weste 83]. MULGA is a symbolic design system which uses the notion of a virtual or topological grid on which symbolic circuit components are placed. The grid defines a relative layout topology without specifying the actual distances between components. After some checking for circuit inconsistencies, the cell is compacted by "moving" the virtual grid lines. A detailed symbolic floor plan is used to give a structural definition of the chip. This floor-plan, together with a number of structural design rules, determines how the leaf cells are assembled to make the final chip. MULGA is the prototypical chip assembly system and it appears to set the standard by which other systems are judged.

There are many CAD systems for integrated circuit design which we have not mentioned. Those we have mentioned do, we feel, give the flavour of the kind of conventional tool which is available.

The introduction of a regular structured design methodology [Mead, Conway 80] and of powerful CAD tools has had a great effect on IC design in general. In 1979, Digital Equipment Corporation set up a small group to investigate the use of a structured design methodology in an industrial setting. The result was a remarkable ten-fold decrease in design time for VLSI circuits [Mudge, Herrick, Walker 80].

In fact, the idea of a hierarchical design method [Rowson 80] has been widely accepted in the VLSI community, as evidenced by the number of manufacturers who are keen to point out that their particular CAD tool supports structured design. However, most of the design tools currently available concentrate on the layout of circuits which have, in a sense, already been designed. Because they operate at such a low level, many of these tools spend much of their time trying to recover structure which has been lost (or forgotten). For example, many systems use circuit or node extractors to try to recover the original intent of the designer. This would not be necessary if the designer could describe the circuit in a structured way, at a higher level. One loses much of the benefit of a hierarchical design method if one presents one's design system only with a "flattened" representation of the final circuit, while denying it information about the other stages of the design. A system which represents a circuit as a "meaningless" collection of coloured rectangles is bound to need complex electrical and layout design rule checkers which try to "interpret" the circuit in a way which allows them to check the design rules. Systems which operate at a higher level can ensure that circuits obey the design rules by construction. A design system which operates at too low a level obstructs the use of abstraction as a design technique and forces design aids to work at an inappropriate level of detail.

Another consequence of working at the layout level is the heavy reliance on simulation as a means of verifying circuit correctness. If one's circuit is represented as a collection of rectangles placed on various layers, then one has little choice but to "extract" the transistors and/or gates and run a simulator. In some cases, however, the simulator ends up checking properties which should have been checked "syntactically" at a higher level. At present, the VLSI community relies almost entirely on simulation to detect errors. However, the feasibility of this approach decreases as the complexity of the chips increases. When we have million transistor chips, it will take

much too long to perform all possible experiments using a simulator. Even if it could be done, the results may be insufficient to ensure that the chip itself is correct. One approach is to have a functional specification of the chip which tells us how it should behave in all circumstances. If we have such a thing, why can we not compile it (in a provably correct manner) directly onto silicon?

This brings us to a whole new area of VLSI research, but one which is sadly underpopulated. Many of the theoretical computer scientists who have been attracted to VLSI in recent years have concentrated their efforts in complexity theory. This is, undoubtedly, a very interesting and important field. Much important work has been done [Thompson 80; Preperata, Vuillemin 80, 81; Leiserson 81; Brent, Kung 81; Chazelle, Monier 81; Baudet 82 and many more], and much remains to be done. It would be nice, however, if more computer scientists applied their knowledge and experience in the areas of compiler techniques, specification methods and language design to the complex problems of "silicon compilation". Some are daunted by the enormity of the problem. Others are merely prejudiced against working with engineers. In any case, a unique opportunity for fruitful collaboration between computer scientists and engineers seems to be being missed.

The pioneering work in silicon compilation was done by Johannsen with his Bristle Blocks Silicon Compiler [Johannsen 79]. The Bristle Blocks system aims to allow the user to design an integrated circuit, with as little concern as possible for the mechanics involved. The fundamental unit is a cell, which may contain geometric primitives such as lines, boxes and polygons, as well as references to other cells. A cell has "bristles" around the edges, and these connection points form its interface with the outside world. The lowest level cells are defined by a description of their actual layout. In a standard cell design language, Bristle Blocks cells are procedural. They are, in fact, small programs which can, among other things, draw themselves. Cells may, for example, stretch themselves, to allow power lines to expand as power demands increase. A particular Bristle Blocks system is designed to compile a particular class of chip architecture.

MACPITTS [Siskind, Southard, Crouch 82a, 82b] provides an alternative to the "hand-crafted" method for designs which fit into the framework of microprogram sequenced data path operations. The system uses a state-oriented register transfer language which has multiple way branching, nested conditions, subroutines and parallel processes. The language is compiled directly to mask-level specifications, with a finite state machine for each parallel process and a data path unit. The data path unit contains operators, as well as registers. The compilation proceeds in two stages. First, a technology independent intermediate description is produced. This can be used to drive a functional simulator, removing the need for a node extractor and switch level simulator. The intermediate description is used to produce technology dependent mask layouts in CIF. These layouts are correct by synthesis. Thus, the designer describes an IC in terms of the algorithm it is to perform, rather than the geometry or topology of its layout. The system is particularly suitable for signal processing applications and the explicit specification of parallelism aids the design of high throughput circuits.

Another successful silicon compiler which allows the rapid implementation of LSI and VLSI signal processors is reported in [Denyer, Renshaw, Bergmann 82]. FIRST is based on the methodology for bit serial architectures of [Lyon 81] and so it works within specific architectural, topological, timing, circuit and layout conventions. It uses a fixed floor-plan format, with ranks of bit serial processors attached to a central communication channel. The system not only gives good compact layouts but it also gives massive cost and time reductions over conventional LSI design techniques.

[Rupp 81] describes DEA, an experimental silicon compiler system which aims to allow the designer to explore alternative architectures for a design, to achieve the required performance, size and power consumption. It also aims to generate the geometric description of the circuit for fabrication with a quality comparable to hand-drawn designs. The system draws on known software compilation techniques and, as a consequence, the DEA language

is a subset of the C programming language. The language allows the user to specify memory arrays, combinatorial elements and cyclic relationships which define discrete memory components. Once the designer is satisfied with the behavioural description of the circuit, he adds geometric information by ordering the behavioural equations and adding intermediate "slicing" levels. This allows the system to produce "soft" cells, which are rectangular, but are arbitrarily flexible. Finally, the soft cells are composed and "hardened" to produce mask geometry.

Some systems have customised PLAs (with feedback) as their target architecture. For example, (Forrest, Edwards 83) describes a system in which a textual language which specifies the behaviour of a finite state machine is compiled automatically into a near-optimum PLA structure.

(Floyd, Ullman 82) considers the design of integrated circuits to implement arbitrary regular expressions. Regular expressions can be used to specify any finite state process (though not necessarily succinctly). A regular expression with n operands can be converted into a nondeterministic finite automaton, with at most n states and n transitions. This finite state automaton can be implemented using an n by 2n PLA. Alternatively, a hierarchical layout can be produced, reflecting the hierarchical nature of the automaton. Regular expressions are presented as a possible useful component of a general purpose language for silicon compilation.

Silicon compiler projects such as these represent a great step forward. The designer gives a behavioural or algorithmic specification of the circuit and the emphasis is on correctness by synthesis. The designer can concentrate on the architectural or logical design by abstracting away from the details of the physical layout.

However, the systems which we have described rely either on the correctness of the input or on simulation as their means of verifying the correctness of the final circuit. In some cases, for example in signal processing applications, this is reasonable because the algorithms being implemented are well known and have been mathematically verified. However, VLSI design, in general, clamours for a mathematical approach. As circuits become more complex, simulation becomes less and less feasible as a means of verification. Martin Rem puts the case more strongly when he says that "there is no future for simulators" [Rem 81]. We should learn from the software engineers, who are beginning to apply formal methods to the design of complex systems, with considerable success. The software engineers have learnt the value of using a powerful notation and the majority of programs are now written in high level languages. Functional languages, in particular, make programs easier and quicker to write because the programmer is able to abstract away from such machine-dependent notions as order of evaluation. In the next section, we will consider the desired properties of a high level VLSI design language. We will compare our approach to that of others who have applied formal techniques to the problems of VLSI design.

Design Languages

One of the most important components of any design method is a language or notation in which to express our ideas and our design decisions. Iverson, in his Turing award lecture on "Notation as a Tool of Thought" (Iverson 80), uses three well-chosen quotations :-

> "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted." *George Boole*
>
> "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race." *A. N. Whitehead*
>
> "The quantity of meaning compressed into small space by algebraic signs is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid." *Charles Babbage*.

We will follow these great scientists in stressing the importance of choosing a design language which is appropriate to the task in hand. In the following subsections, we list and discuss the properties which we consider to be vital to a good VLSI design language.

Simplicity

We place simplicity before all other properties because we feel that a language which is over-complicated has little hope of fulfilling our other requirements. A simple language is easier to learn, easier to use and easier to read. These factors are important in an industrial environment where designers' time is an expensive and scarce resource. Designers have long used the simple expressive notations of block, logic and circuit diagrams. These diagrams can be made to contain exactly the required information. They are concise, yet they are easy to read. A notation which is over complicated will not compete against the techniques already in use.

A simple language gives some hope of mathematical tractability. μFP is made up of a small number of primitive functions and combining forms. Each combining form has a simple geometric interpretation. This is important as our ultimate goal is to produce chip layout from a behavioural description. As explained in the previous chapter, the block diagram corresponding to a μFP expression is calculated in a "hierarchical" manner. The block diagram of an expression is some simple combination of the block diagrams of its subexpressions. We are, in some sense, "tiling the plane" with block diagrams. The basic tiles are those for wires and those for primitive functions. These are the leaves of the tree corresponding to the μFP expression. The combining forms at the nodes of the tree tell how the tiles are to be laid out. If we assume that primitive functions have fixed size, then we will need extra "wire" and "blank space" tiles to ensure that adjcent blocks are properly joined up. It has been suggested that one of the reasons why silicon compilation is still in its infancy is that it is difficult to express a circuit, which is a two-dimensional object, in a one-dimensional language. Our solution to this is to consider useful ways of constructing circuits by tiling the plane with rectangular subblocks. This approach restricts our choice of combining forms to those which are appropriate to our final goal - layout.

Another reason for choosing a simple language is that it is more likely to have a formal semantics. If we are to eschew simulation for rigorous proof techniques, our design language must have a complete formal semantics. To be useful, a formal semantics must be comprehensible. Because we have a very restricted notion of state, the operational semantics of μFP are simple. A μFP expression can be thought of as a simple data-flow machine, through which we push our inputs, one at a time. In one cycle, data "ripples" through the machine from input to output. The machine may change during a cycle, as the stored state may be updated. The denotational semantics of μFP is given in terms of FP. This allows us to describe functions which take a stream of inputs to a stream of outputs in terms of simpler functions which take an input to an output. The matrix transpose function, zip, is used to keep the types right. This approach allows us to prove new algebraic laws in μFP using the known laws of FP.

An obvious and gross simplification which we have made is to abstract away from details of timing. μFP describes synchronous systems in which all parts of the circuit operate in lock-step. There is no notion of a cycle lasting a particular length of time. We are concerned with the ordering of events, not with their duration. We realise that timing is important, but we feel that timing details should only be "added" to a circuit which is known to be functionally correct at a more abstract level. The task of proving a circuit correct in the presence of timing details is enormously complicated. The circuit may be incorrect because there is an error in the design or because there is a timing error and so, one should be doing two separate proofs.

Some VLSI description languages include timing details. For example, Noon's Design Specification Language [Noon 77] has basic statements of the form

WHEN () IF () MAKE () WITHIN () UNLESS ();

So, a possible statement would be

```
WHEN A RISES OR B FALLS OR C CHANGES
  IF D=1 AFTER 20ns
  MAKE E = (F & G) AND H=0
  WITHIN 50ns TO 75ns
  UNLESS B RISES OR J CHANGES;.
```

The language is, of course, not designed to be reasoned about using mathematical techniques. It is designed for use with a functional simulator. The design is simulated with certain stimuli and the designer compares the output with what he "expects" and tracks down bugs. How the designer calculates what to expect is not specified. Languages of this type are very popular, but they are not suitable for use with formal techniques. Temporal Logic promises to be useful for reasoning about circuit timing (Bochmann 82, Moszkowski 83), but the use of an over-complicated language will prevent us from exploiting its power. Many of the researchers whose work we will refer to in the following sections have deliberately chosen small simple languages, to facilitate formal reasoning.

In the next section, we consider our second requirement for a good VLSI design language, expressive power.

Expressive Power

Obviously, it is not enough to demand that a language be simple, clear and easy to read and use. It must be able to express any system that the designer might wish to describe. We wish to enhance the skills and techniques of the designer, not to constrain them. We feel that a good design language should not impose a particular low level design technique on the user

There are obvious trade-offs between expressive power and some of our other requirements, for example simplicity and mathematical tractability Clearly, there is no point in using a language which curbs the creative talents of the designer by restricting him to a smaller design space. On the other hand, a language which is too powerful can cause problems We must bear in mind our aims - to verify formally the correctness of a circuit and to produce a layout of that circuit Both of these aims place constraints on the expressive power of the design language As Cardelli [Cardelli 82] points out, the activities of formal description and formal verification are often inversely proportional, in that a very precise and detailed description of a system can blind one to it general properties. So, the desired expressive power of our language is constrained "from both ends"

In μFP, we have chosen to represent state using the simple notion of a feedback latch. Using the μ combining form, we can place those signals which we wish to remember in a latch of the required size. The signals are then available to the function on the next clock cycle This is a restricted way of introducing state since the signals are remembered only for one clock cycle We are not provided with variables into which we can place values for safe-keeping, until they are needed later in the computation. Such a language might be able to express some computations more naturally and concisely, but the increase in expressive power would be far outweighed by the increase in complexity of the formal semantics, with its consequent decrease in our ability to reason about the language. In μFP, a circuit with state is expressed as a finite state machine, with *next output* and *next state* functions Thus, we can express any function which is suitable for implementation on silicon, while retaining most of the algebraic properties of the original applicative language, FP

Other workers have (for similar reasons) placed restrictions on their notions of state. [Babiker, Fleming, Milne 83], in their language LTS, ('Layout and Timing for Structures'), use a non-procedural style of description, with a backward-looking synchronous treatment of time. The behavioural description of a circuit is given in a functional language, with a signal being viewed as a mathematical function taking times to values. They introduce state by using last a function from signals to signals. Last(x) is a signal which, at any instant, has the value that x had at the previous instant. The formal semantics of LTS has not yet been published but we expect that (as in μFP) the combination of an applicative language and a restricted notion of state will give a simple elegant semantics.

Gordon [Gordon 82] uses a similarly restricted notion of state and he chooses to work with sequential behaviours, rather than with machines. The domains Com[X; Y] and Seq[X; Y] represent combinatorial and sequential behaviours respectively. The combinatorial behaviour of a device, whose set of input lines is X and whose set of output lines is Y, is a function from Sig[X] to Sig[Y]. Members of Sig[X] (and Sig[Y]) are called signals.

Com[X; Y] = Sig[X] → Sig[Y] is the domain of combinatorial behaviours from X to Y. Seq[X; Y], the domain of sequential behaviours from X to Y, is the least solution of the domain equation

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

(with $f : D \rightarrow (D' \times B)$, $fst \circ f : (D \rightarrow D')$ and $snd \circ f : (D \rightarrow B)$).

Informally, a device with semantics f in Seq[X; Y] behaves like a combinatorial device with behaviour

$$(fst \circ f) : Sig[X] \rightarrow Sig[Y]$$

until it is clocked with input signal $s \in Sig[X]$. It then changes behaviour to

$$(snd(f))\ s \in Seq[X; Y]$$

to give us a "new" sequential behaviour.

Clearly, there are parallels between this definition of the semantics of sequential behaviours and our definition of the semantics of the combining form μ.

$$M[\![\mu f]\!] = \text{out } M[\![f]\!]$$
$$\underline{\text{where}} \text{ out } g\ i = o$$
$$\underline{\text{where}} \langle o,s\rangle = zip \circ g \circ zip : \langle i,?is\rangle$$

Our semantics automatically maps functions along a sequence of inputs to give a sequence of outputs, and so we do not recurse explicitly. Gordon shows how sequential machines are related to the domain of sequential behaviours. For a machine M, with output function OUTM, next state function NEXTM and set of states SM, the sequential behaviour of M in state x is given by

$$B[M]x = \lambda s.\langle OUTM(s, x), B[M]\ (NEXTM(s, x))\rangle$$
$$(B[M] : SM \rightarrow Seq[X, Y]).$$

This equation is related to equation I above. Our equation refers to the whole sequence of states, rather than to a particular state. In the last part of equation I, the fact that we have s on the lefthand side and ?is on the right has the same effect as the recursive call of B with the next state. In a given cycle, both equations involve the production of an output and a new state which is then used in the calculation of the next output.

Gordon works with behaviours rather than machines because, when writing specifications, he wishes just to give desired behaviour, not a machine realising that behaviour. Also he wishes to avoid having to express what it means for an implementing machine to meet a specification. His choice is motivated by the fact that he is primarily concerned with verification. His aim is to show that the expression corresponding to the structure of the system being verified has the desired behaviour.

Our aims are slightly different. We wish to transform an initial description of the required circuit (normally in the form $\mu[x, y]$) into one which is suitable for layout on silicon. The final circuit must have the same semantics as the original one and so we must use only "semantics-preserving" transformations. We aim to produce a machine and so we start with one. In both cases, the choice between behaviours and machines was made on the grounds of elegance and manipulative ease. The fact that the two languages were required to encapsulate slightly different types of information caused us to make different choices.

In the next section, we exlain why notions of elegance and manipulative ease are valid criteria in the design of languages for VLSI description.

Mathematical Tractability

One of the main problems in VLSI systems design is the management of complexity. The introduction of regular hierarchical design methods, such as those advocated in [Mead, Conway 80], has been a great breakthrough in this area. However, as systems become more and more complex, structured design methodologies must be combined with formal methods for circuit verification, if we are to produce correct circuits in reasonable timescales. One of the features which VLSI design shares with software design is that, as the design of a system proceeds, the cost of repairing errors grows exponentially. Also, the later an error is found, the more likely it is to have originated early in the design procedure. So, it is important to see verification not as something which happens after the circuit has been designed, but as an important part of the design process itself.

In μFP, we use transformations based on the axioms of the language to give this kind of continuous verification. The functions and combining forms of FP were chosen because they have nice algebraic properties. The language was designed to allow the programmer to reason about his programs by manipulating the programs themselves. We were persuaded to use FP as

the basis of our language, because it has an associated algebra of functions. A good VLSI design language must allow us to show that two circuits have the same "meaning" or behaviour. As explained in the previous section, we use a restricted notion of state to ensure that μFP retains most of the algebraic properties of FP. We give the semantics of μFP in terms of FP so that we can draw on the reservoir of known algebraic laws in FP to prove new laws in μFP. We are, in general, concerned with proving two circuit descriptions equivalent and this is done by applying a sequence of transformations based on algebraic identities to one of the descriptions to produce the other. Based on the ideas developed in this dissertation, Simon Finn [Finn 83] has implemented a transformation system for μFP in the purely functional language, Lispkit Lisp [Henderson, Jones, Jones 83]. The system allows one to transform a μFP expression (into a semantically equivalent one) by applying tactics, which may be axioms or combinations of tactics. Finn concluded that a more realistic and usable system might adopt the ideas of [Feather 79] where one could set goals for the system to try to achieve. Such a system would reflect more closely the manner in which design is actually done. The use of transformations relies heavily on the fact that μFP is a tractable notation because of its algebraic properties. A language which is not tractable will not only be difficult to use but also difficult to "automate". If we are to formally verify complex systems, we need languages which are suitable for use with software tools such as transformation systems and proof checkers.

Our transformations are, of necessity, semantics-preserving and so the laws of μFP which concern us are algebraic identities. In chapter 5, however, we introduce the predicate slowmodels which allows us to analyse circuits in which 50% of the processors are active at any given time. Slowmodels allows us to show that a circuit of this type is "sufficient" to fulfil a given specification (in terms of an "ordinary" circuit) provided we are willing to have the necessary "don't cares" in the input and output streams. μFP proved to be useful in reasoning about such circuits. We hope to generalise this technique to deal with such things as micro-cycles in micro-coded machine implementations.

Cardelli, in his thesis [Cardelli 82], shows how algebraic techniques can be applied to many aspects of VLSI description and verification. He introduces a simple and uniform notation for the description of networks of hardware components. A network is a structured graph with an interface and the notation for structured graphs is designed to be "formally tractable, expressive enough to be used as a programming language, and easily convertible into useful data structures". He uses an abstract data type of networks over which certain operations, such as the composition of subnetworks, can be performed. He formalises these ideas in an algebraic framework and he gives some examples of how networks may be programmed in net algebras. The main example is the systolic pattern matcher of [Foster, Kung 79] which is very similar to the systolic correlator which is our main example. His examples are designed to show that the approach can be applied to various levels of description, in the range from abstract behavioural specifications to actual circuits. A formal semantics for the topmost level of description, Clocked Transition Algebra, is given in terms of Milner's Synchronous CCS [Milner 82]. CTA deals with the behavioural specification of synchronous systems. Cardelli points out that formal proofs about such systems are good candidates for mechanisation. He also tackles the formalisation of real-time systems, using both denotational and operational semantics techniques. He considers the problems of translating between the various levels of description and shows how to translate purely topological planar stick diagrams into grid structures, which can be used to generate layout. He gives an efficient algorithm for stretching grid structures to ensure port-matching. Finally, he describes the implementation of an experimental VLSI design system, in which the geometric details of layouts are hidden from the user by the use of algebraic operations.

Cardelli's thesis demonstrates that algebraic techniques can effectively be applied to several aspects of VLSI design. He places emphasis on casting the problems involved in a simple framework, involving a small number of primitive concepts. In comparison, our thesis is both less formal and less

wide-ranging We consider only synchronous systems, using a discrete time model. Our notation is not designed for use with asynchronous or real-time systems. This simplification allows us to use a small number of combining forms to describe the various combinations of subcircuits. In particular, we are free to use a very simple form of composition, which, in turn allows us to use a variable free language. This makes it easy for us to perform proofs showing that a given combination of circuit elements has a certain behaviour Cardelli uses a more general form of composition, and so, proofs of equivalence between circuits are more difficult. Cardelli uses formal algebraic techniques and provides his own mathematical basis We rely on others to provide our formal basis, in that we rely on the existence of the algebra of functions of FP. What the theses do have in common is an attempt to find simple solutions to the complex problems of VLSI design

Other workers who apply algebraic techniques to the problems of VLSI design are Gordon [Gordon 82], whose work was mentioned earlier in this chapter, Milne [Milne 82a, 82b] and Subrahmanyam [Subrahmanyam 83].

Gordon models register transfer systems using sequential behaviours. He uses algebraic techniques to express and reason about specifications and implementations at several different levels of abstraction. His verification of nMOS devices inspired our attempts to do the same The third example of chapter 4 is a repeat of one of his examples. Gordon also proves correct a micro-coded implementation of a small general purpose computer He is currently investigating methods of automating his proofs of correctness He has also extended his model to cope with bidirectional devices such as pass transistors.

Milne works on the development of calculi or languages which can be used to design circuits functionally and to verify their correctness with respect to specifications. He has developed a calculus for the description of circuit behaviour, CIRCAL. The calculus is event-driven and a CIRCAL expression describes how a computing agent reacts to the exchange of stimuli with

its environment. A computing agent has named ports and the composition operator links similarly named ports belonging to different agents. The agents exchange information over the lines thus created. CIRCAL has a set of laws which facilitate reasoning about circuits by the algebraic manipulation of CIRCAL expressions. In [Milne 82b], the correctness of a simple silicon compiler is demonstrated. A "high" level language of Nor Expressions is compiled into a layout language with a single primitive, the NOR gate. Semantic functions are given, which map both the high level Nor Expressions and the circuit level Layout Expressions into CIRCAL. The silicon compiler is proved correct by showing that both the Nor Expression and its corresponding circuit layout produce equivalent CIRCAL expressions. The technique is applicable to more complicated silicon compilers, though an automatic transformation system for CIRCAL would probably be required, to assist in the proof.

[Subrahmanyam 83] is concerned with the synthesis of VLSI circuits from high level behavioural descriptions. The formalism introduced is designed to provide a rigorous basis for the construction of transformation systems and a framework for proving correctness. The same set of algebraic primitives is used to model the three levels of abstraction - functional, architectural and electrical. The paper describes the transformation of high level behavioural descriptions into lower level architectural descriptions. These low level descriptions are expressed in a high level programming language with some special constructs for hardware description. The next step is to transform these structural descriptions into collections of state machines which control the flow of data between hardware representations of the required data structures. The state machine descriptions are translated into symbolic circuit layouts, and hence into mask geometry.

Many of the researchers mentioned above have been greatly influenced by Milner's work on CCS [Milner 80]. More recently, Milner has introduced Synchronous CCS, [Milner 82] which can model both synchronous and asynchronous computations. He first considers synchronous systems and then

shows that asynchronous systems can be characterized as a subclass. He shows that CCS is precisely derivable from SCCS. SCCS contains only four combinators and a construct for recursion but it is remarkably expressive. The calculus is appropriate to the description of distributed programs and it may also be useful in hardware description. One of the examples given is a proof that the familiar combination of two NOR gates to make an RS flip-flop behaves as one might expect.

Although much work in the areas of VLSI description languages and design tools takes no account of the need for formal verification, the success of those who try to provide simple solutions to complex problems by the use of algebraic methods gives some hope for the future.

Constrained Communication

Constrained communication is an important requirement of a VLSI design language, but one which is often ignored. It is widely agreed that one of the fundamental properties of VLSI is that complex computations can be realised by a large number of processes, operating concurrently. The challenge of VLSI is to find ways of harnessing this concurrency, without being overwhelmed by it. Merely to implement bigger faster sequential machines is to fail to take full advantage of the potential. New architectures must be discovered. Research into systolic architectures, which are designed to make optimal use of pipelining and parallelism to give high performance, is probably the most advanced work in this area [Kung 79; Brent, Kung 82; Foster, Kung 79; Leiserson 81; Kramer, van Leeuwen 83; Evans, McWhirter, Wood, McCanny, McCabe 83; McCabe, McCabe, Arambepola, Robinson, Corry 82].

Motivated by the importance of concurrency, most researchers choose to characterize the circuits themselves as collections of sequential processes, which communicate with each other along named lines or channels or through named ports. This is an unconstrained form of communication. The network of processes can form an arbitrarily complex graph, with spaghetti-like communication lines. This causes problems. First, the layout, in two dimensions, of an arbitrary graph is a hopelessly difficult task. Second, this approach ignores an extremely important property of VLSI, the fact that while local communication is cheap, global communication is enormously expensive. A system in which communication is through named ports fails to distinguish global and local communications. Third, unconstrained communication greatly complicates the proofs which must be done when reasoning about circuits which are composed of subcircuits.

When reasoning about circuit behaviours, the most common form of identity which we need to prove is that "the composition of A and B has the same behaviour as C". In a design language, the way in which the composition of subblocks is defined (which determines the allowed forms of communication) has a great bearing on the ease with which proofs about the language can be performed. One of our requirements for μFP was that it should allow the designer to reason about his circuits, using simple algebraic laws. Another requirement was that it should capture details of layout. Unconstrained communication makes both reasoning and layout difficult and so we restrict communication by allowing subblocks to be combined only in simple ways. We consider only rectangular subblocks and useful ways of tiling the plane with them. In the simplest form of μFP, communication only takes place across the vertical "tile" boundaries. In the extended form, there is communication across the horizontal boundaries. In either case, only adjacent subblocks communicate. The result is that the circuit is easy to lay out. We use a simple recursive technique, as explained in chapter 6.

Although we use a simple notion of composition, we can still express systems in which there is global communication (for example, a wire which "bypasses" several subblocks). The difference is that such wires are regarded as functions, and must be represented explicitly. The designer may, if he wishes, transform his μFP expressions to minimise such global communications. The fact that only local communications are implied by the use of combining forms facilitates proofs about circuits, by allowing the problem to be subdivided cleanly.

We feel that our use of constrained communication distinguishes our work from that of others in the field. [Rupp 81] identifies a problem which arises when performing hardware compilation using a completely general composition scheme. The "side assignment" problem occurs when attempting to wire up a set of components in a bottom up manner. The components are divided hierarchically into one-dimensional slices. A slice in one direction contains an ordered list of slices in the orthogonal direction. Thus, a two-dimensional arrangement can be specified. A circular problem arises when one tries to orient the components of a slice and to assign logical signals to the sides of the slice. To perform either operation, one would like to have already performed the other! This is the kind of problem that arises when one tries to lay out an arbitrary graph. Rupp points out that the ultimate quality of a silicon compiler appears to be limited by its solution to the leaf-cell side assignment problem. We avoid the problem by demanding that global communication be represented explicitly. The amount of global communication in a circuit description could be used as a measure of the quality of the corresponding layout. We hope, in the coming year, to investigate heuristics for finding "optimal" layouts.

Function Level Reasoning

An important property of the FP programming language [Backus 78, 81] is that it has an associated algebra of functions, rather than an algebra of compuled objects. The "cells" on integrated circuits correspond to functions. Even the wires can be thought of as identity functions. Objects such as signals "appear" from outside the chip and are manipulated by these functions. Since we wanted to reason about circuit behaviours and to transform circuit descriptions to investigate different layouts, we were drawn to FP, which allows the programmer to reason about his programs by manipulating them.

The work of Kieburtz and Shultis [Kierbutz, Shultis 81] is related to ours in that they transform FP program schemes to improve efficiency. They have found that, in earlier work on transformation, the use of variables in Lisp-like languages has interfered with the identification of superficially dissimilar programs as instances of a common scheme. The variable free notation of FP seems to be easier to work with. We find that it greatly facilitates reasoning about circuits by allowing us to avoid the uncontrolled communication which complicates proofs of correctness. It is the absence of subexpressions which denote data values which permits the embedding of layout information in our behavioural descriptions. Our descriptions consist only of combinations of functions, and each function can be associated with a "block" in the circuit layout in an unambiguous way.

A circuit expression is either a primitive function or a combination of circuit expressions and so, a circuit can be described in a clearly hierarchical manner. A very general composition scheme may blur the edges of the levels of hierarchy.

Some researchers use imperative languages for circuit description [Barbacci 78; Desmarais, Shaw, Wilcox 82; Roth 81; Lewke, Ramming 83]. However, the destructive assignment of imperative languages not only complicates reasoning about programs but it also "loses" valuable structural information. If we are free to assign to a variable anywhere in the program, it becomes difficult to associate a simple circuit structure with that program. For these reasons, we have chosen to use an applicative language as our basis. Many have made the same decision [Babiker, Fleming, Milne 83; Cardelli 81; Gordon 81; Johnson 81]. Johnson describes how recursive systems of equations can be used to express circuit behaviour. As in μFP, he uses streams, which are infinite sequences of finite elements for input and output. Time is "encoded" in these streams and he uses an output driven list processing system, with lazy evaluation, to run his programs. As in our semantic equations, he must face the problem of keeping the types right by transforming streams of sequences into sequences of streams and vice versa. He uses a form of function application which automatically transposes the input and a function wire which repeatedly applies the identity function to perform the two required transpositions.

Our conclusion has been that if a description language is to be used for reasoning about circuits, it must be free from side effects. Reasoning at the function level allows us to associate a floor-plan with each μFP expression in a hierarchical manner, since each combining form has a simple geometric interpretation.

Abstraction and Hierarchical Structure

Hierarchical decomposition is an important way of managing complexity in VLSI design. [Suzuki, Burstall 82] describes a VLSI Modelling Language and system in which user-defined abstraction (or the ability to partition a system into arbitrary subparts) is an important feature.

[Masuzawa, Nakauchi, Wada, Hagihara, Tokura 83] describes a Systolic Algorithm Description Language, SADL. The language is designed for use with a CAD system which supports the algorithmic design phase. It allows VLSI algorithms to be designed hierarchically. An algorithm is implemented by a network of interconnected cells. A cell has an external specification, which gives details of how the cell behaves against input data. It also has a realization definition, which describes how the external specification is implemented, using a network of lower level cells. The cell also has information on its performance, for example, delay time, power consumption and area. These may be estimated or actual measurements. This system makes a definite attempt to separate concerns in a hierarchical way. The system is currently used with simulators. However, the authors hope eventually to use some formal descriptive method for the external specification of cells, so that they can perform automatic or semi-automatic formal verification of their designs.

Rem [Rem 81, 83; Rem, van de Snepscheut, Udding 83] argues eloquently the need for hierarchical structuring of designs. He argues that the specification of a component should not reflect its internal structure but should define only how the component looks from the outside. The correctness of a component is checked (mathematically) by comparing the behaviour of the set of subcomponents (each of which has a precise specification) with the required specification. Rem characterizes a component by the set of traces of its possible communications with the outside world

He then describes ways of composing components. Communication is the elimination of common atoms in traces. He shows how trace theory can be used to reason about and prove properties of hierarchical components. By using a composition operator which expresses the delay between the sending and reception of signals, he formalizes the notion of delay-insensitivity. He gives some examples of the translation of hierarchical components into self-timed circuits. [van de Snepscheut 83] extends the work of Rem by deriving circuits from the programs describing his hierarchical components. First, the component is transformed so that the composition of subcomponents is self-timed. The subcomponents are then implemented as Mealy- or Moore-like finite state machines, with a communications protocol between them. Trace theory has similarities to Hoare's CSP [Hoare 81] but, at present, it lacks the expressive power of CSP. Reasoning at the level of individual traces is difficult. It is hoped that new theorems about trace structures will be found, to allow reasoning about components.

Although we have taken a very different approach, we agree with Rem both on the need to use formal methods and on the importance of hierarchical decomposition. In μFP, the original circuit, whose precise specification is known, can be decomposed into subcircuits, each of which has a precise specification. Using the specifications of the subcircuits, we can check that the chosen combination obeys the original specification of the circuit. Only when this has been done need we consider how the subcircuits are implemented. Thus, at any level, the designer need only consider details which are appropriate to the design decisions which must be made at that level. This ability to abstract away from irrelevant details is vital if we are to design large systems reliably.

Chapter 8: Conclusion and Future Work

We have presented μFP which has many of the properties which we have suggested go to make a good VLSI design language. It is simple, being made up of primitive functions and a small number of combining forms. Because the combining forms have geometric as well as semantic interpretations, the language can be used to describe both the behaviour and the layout of circuits. It is concise and mathematically tractable. The conciseness can make the language difficult to read at first, but we have found that one becomes fluent with practice. We give the formal semantics of μFP in terms of FP and the fact that we use only a restricted notion of state allows us to retain many of the nice algebraic properties of FP. Thus, the designer can reason about his circuit descriptions by manipulating the descriptions themselves. A proof that two circuit descriptions have the same semantics is done by transforming one of the descriptions into the other, using the algebraic laws.

μFP is a variable free language in which we reason about functions rather than about objects. The variable free notation allows us to use a constrained form of communication which facilitates proofs about combinations of subcircuits. It is difficult to perform such proofs in a system where communication is through named ports or along named lines or channels. Unconstrained communication also complicates layout. Our definitions of composition are such that they imply only local communication. We consider only rectangular subblocks and useful ways of tiling the plane with them. The result is that the circuit corresponding to any μFP expression can be laid out using a simple recursive technique.

The use of function level reasoning is appropriate to the structured hierarchical design methodologies which are vital to the management of complexity. μFP allows circuits to be described in a clearly hierarchical manner. A circuit expression is either a primitive function or a combination of circuit expressions. The designer can decompose his original circuit (whose precise specification is known) into subcircuits, each of which has

a precise specification. Using the algebraic laws of μFP, he can determine whether the combination of subcircuits obeys the original specification. To do this, he need consider only the specifications of the subcircuits, not their implementations. This ability to abstract away from irrelevant details is an important part of the hierarchical design method.

We have demonstrated the flexibility of the language by using it to describe circuits of various types, ranging from a combinatorial tally circuit to our main example, the systolic correlator. In that example, we introduced some simple but powerful techniques for analysing systolic circuits. Our derivation of the circuit can be considered to be a proof of its correctness.

We envisage μFP being used in the following way. The required behaviour of the chip will first be specified. Our most abstract form of circuit description is one which has a single μ on the outermost level. This form specifies a combinatorial circuit and a register bank through which signals are fed back. The designer will use his skill to discover a more elegant and efficient implementation of this behaviour on silicon. He will normally proceed through several design iterations in this process, moving from the abstract specification to a real implementation. He can, at any stage, simulate his design to check that his refinements have been made correctly. We stress, however, that simulation should be used as an aid to formal reasoning, not as a substitute for it. At each stage, the designer should prove (or at least satisfy himself) that the design still obeys the specification. This can be done with the aid of a transformation system which "implements" the algebraic laws. As he proceeds through the design, the combinatorial and the memory elements will become more and more "mixed up". Eventually, he hopes to reach a satisfactory layout in which memory elements have been placed as near as possible to where they are "needed". So, the process of translating from specification to implementation can be viewed as one of pushing the μs further and further down into the μFP expression, until they can go no further. When the design is finished, the μFP description of the implementation can be passed to the layout program, which will do the tedious work of laying out the chip.

The work just described opens many interesting avenues of research. The following sections contain brief descriptions of six areas of research which particularly interest us.

1: Tackle a wider range of examples and refine the language based on the experience gained

It would be useful to tackle some complete designs (not just specifications) in μFP and in some other formalisms. This would allow us to compare the expressive power and ease of manipulation of the languages being considered. We would like to refine μFP into a small but powerful language whose combining forms are exactly those which we need to describe general integrated circuits. As my supervisor has suggested, there may be a smaller, more elegant language in there, waiting to be released.

2: Investigate the usefulness of temporal logic and of other calculi for reasoning about sequential integrated circuits

Temporal logic seems well suited to reasoning about integrated circuits as they change over time. Conventional logic can be used to reason about combinatorial circuits and temporal logic has the necessary extensions to allow us to reason about circuits with state. We would like to investigate the relationship between temporal logic and μFP.

We hope to investigate the usefulness of various other calculi for reasoning about integrated circuits. The functional models of CSP developed at Oxford by Reinecke may provide an appropriate formal basis. An alternative would be to use a subset of Z, the formal specification language developed at Oxford by Abrial, Sufrin et al (Abrial 82, Sufrin 82). Milner's Synchronous CCS also seems to have the required combination of expressive power and manipulative fluency.

1: Investigate the extension of μFP to deal with asynchronous circuits

μFP, as it stands, is suitable for describing synchronous circuits Synchronous circuits, especially those using a two phase clock, are very common. However, in some applications, there are problems with the distribution of a clock, and it is necessary to use asynchronous circuits. In a self-timed system, there is no master clock and temporal control is delegated to the elements of the system. A self-timed system is either a self-timed element or a legal interconnection of self-timed systems. The design of self-timed elements is difficult. However, the system designer, equipped with these elements, is free to abstract away from many of the physical details of the system. We hope to extend μFP to deal with self-timed systems and, to investigate the usefulness of temporal logic in reasoning about such systems.

4: Write a prototype compiler from μFP to "gate-array"

The building of a general silicon compiler is obviously an enormous task. A "gate-array compiler" is a less ambitious project. An Uncommitted Logic Array (ULA) or gate-array is a chip laid out as a regular array of identical cells. The cells are usually two-input logic gates, for example NAND or NOR gates. The chip is customized by wiring up the gates to perform the required function. There are two wiring layers, one for horizontal segments, the other for vertical ones. At the VLSI 83 conference, M. Burstein presented an elegant hierarchical routing algorithm for gate-arrays [Burstein, Hong 83] We intend to use that algorithm in a simple compiler.

5: Make the μFP transformation system implemented by Simon Finn "smarter". Use the system to investigate useful strategies for proceeding from a high level μFP description to one which is suitable for layout on silicon

We would like to increase the usefulness of the μFP transformation system by incorporating some form of "goal directed" transformation. The system would then reflect more closely the way in which design is normally done. We would like to use the system to investigate strategies for transforming a μFP expression into one which gives an "efficient" layout. There will be a close connection between this work and that on the gate-array compiler

6: Investigate systolic architectures

VLSI gives us the possibility of performing enormous numbers of computations concurrently. We must find ways of harnessing this ultra-concurrency. Systolic architectures, in which a large number of identical cells operate rhythmically on the data, can be used to implement some algorithms very efficiently. We hope to investigate ways of transforming non-systolic implementations of algorithms into systolic ones.

References

[Abrial 82] J.-R. Abrial: "Specification and Construction of Machines" notes, semantics workshop, Wolfson College, Oxford, Sept. 1982.

[Ackland, Weste 83] B. Ackland, N. Weste: "An Automatic Assembly Tool for Virtual Grid Symbolic Layout". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983

[Ayres 79] R. Ayres "IC Specification Language", Proc. 16th Design Automation Conference (IEEE), 1979

[Babiker, Fleming, Milne 83] S.A. Babiker, R.A. Fleming, R.E. Milne "A Tutorial for LTS" Standard Telecommunication Laboratories Limited, Internal Technical Memorandum No. 224.83.3 Commercial in Confidence, 1983

[Backus 78] J. Backus "Can Programming Be Liberated from the von Neumann Style?", Communications of the A.C.M., Vol. 21, No. 8, pp. 613-641, Aug. 1978.

[Backus 81] J. Backus, "The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions", Proc. Symposium on Functional Languages and Computer Architecture, Gothenberg, June 1981

[Barbacci 78] M.R. Barbacci "The Symbolic Manipulation of Computer Descriptions: An Introduction to ISPS", Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Aug. 1978.

[Batali, Mayle, Shrobe, Sussman, Weise 81] J. Batali, N. Mayle, H. Shrobe, G. Sussman, D. Weise: "The DPL/Daedalus Design Environment", In "VLSI 81", J. Gray (ed), Academic Press, 1981

[Baudet 82] G.M. Baudet, "Design and Complexity of VLSI ALgorithms" In "Foundations of Computer Science IV: Part 1", J.W. de Bakker, J. van Leeuwen (eds), Mathematisch Centrum, Amsterdam, 1983.

[Bochmann 82] G.V. Bochmann: "Hardware Specification with Temporal Logic: An Example", IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982.

[Boyd 79] D.R.S. Boyd: "GAELIC Language", Technology Division, Rutherford Laboratory, 1979

[Brent, Kung 81] R.P. Brent, H.T. Kung: "The Area-Time Complexity of Binary Multiplication", Journal of the A.C.M., Vol. 28, No. 3, July 1981.

[Brent, Kung 82] R.P. Brent, H.T. Kung "Systolic VLSI Arrays for Polynomial GCD Computations", Technical Report, Dept. of Computer Science, Carnegie-Mellon University, March 1982.

[Buchanan 82] I. Buchanan: "SCALE, a VLSI Design Language" Technical Report CSR-117-82, Dept. of Computer Science, University of Edinburgh, May 1982.

[Buchanan, Gray 79] I. Buchanan, J.P. Gray: "Models for Structured Integrated Circuit Design", Technical Report CSR-48-79, Dept. of Computer Science, University of Edinburgh, 1979.

[Burstein, Hong 83] M. Burstein, S.J. Hong "Simultaneous Placement and Wiring of Gate Arrays" In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983

[Cardelli 81] L. Cardelli: "Sticks & Stones An Aplicative VLSI Design Language", Technical Report CSR-85-81, Dept. of Computer Science, University of Edinburgh, July 1981

[Cardelli 82] L. Cardelli: "An algebraic Approach to Hardware Description and Verification", Ph. D. Thesis, Dept. of Computer Science, University of Edinburgh, 1982

[Cardelli, Plotkin 81] L. Cardelli, G. Plotkin: "An Algebraic Approach to VLSI Design". In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Chazelle, Monier 81] B. Chazelle, L. Monier: "Optimality in VLSI". Technical Report CMU-CS-81-141, Dept. of Computer Science, Carnegie-Mellon University, Sept. 1981.

[Chen, Hsu, Kuh 83] N.P. Chen, C.P. Hsu, E.S. Kuh: "The Berkeley Building-Block (BBB) Layout System for VLSI Design" In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Clark 80] W.A. Clark: "From Electron Mobility to Logical Structure: A View of Integrated Circuits". Computing Surveys 12 (3), 1980.

[Denyer, Renshaw, Bergmann 82] P.B. Denyer, D. Renshaw, N. Bergmann "A Silicon Compiler for VLSI Signal Processors" Proc. ESSCIRC 82, Brussels, 1982.

[Desmarais, Shew, Wilcox 82] P.J. Desmarais, E.S.Y. Shew, P.S. Wilcox "A Functional Level Modelling Language for Digital Simulation" Proc. 19th Design Automation Conference (IEEE), 1982.

[Evans, McWhirter, Wood McCanny McCabe 83] R. Evans, J.G. McWhirter, D. Wood, J.V. McCanny, A.P.H. McCabe: "Multibit Convolution Using a Bit Level Systolic Array" In "VLSI 83" F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Fairbairn, Rowson 78] D.G. Fairbairn, J.A. Rowson. "ICARUS An Interactive Integrated Circuit Layout Program". Proc. 15th Design Automation Conference (IEEE) pp. 188-192, June 1978.

[Feather 79] M. Feather: "A System for Developing Programs by Transformation". Ph. D. Thesis, Dept. of Computer Science, University of Edinburgh, 1979.

[Finn 83] S. Finn: "LVIS - A VLSI Transformation System". M. Sc. Dissertation. Programming Research Group, University of Oxford. Sept. 1983.

[Floyd, Ullman 82] R.W. Floyd, J.D. Ullman: "The Compilation of Regular Expressions into Integrated Circuits". Journal of the A.C.M. Vol 29, No 3, July 1982.

[Flynn 81] B. Flynn: "An introduction to the use of GAELIC and SPICE from the LSI Design Facility" Technical Report, Dept of Electrical Engineering, University of Edinburgh. Oct. 1981

[Forrest, Edwards 83] J. Forrest, M.D. Edwards: "The Automatic Generation of Programmable Logic Arrays from Algorithmic State Machine Descriptions". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Foster, Kung 79] M.J. Foster, H.T. Kung: "Design of Special-Purpose VLSI Chips: Example and Opinions". Technical Report CMU-CS-79-147, Dept. of Computer Science, Carnegie-Mellon University, Sept. 1979.

[Gordon 81] M. Gordon. "A Very Simple Model of Sequential Behaviour of nMOS". In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Gordon 82] M. Gordon: "A Model of Register Transfer Systems with Application to Microcode and VLSI Correctness". Technical Report CSR-82-81, Dept of Computer Science, University of Edinburgh. May 1982

[Hedges, Slater, Clow, Whitney 82] T.S. Hedges, K.H. Slater, G.W. Clow, T.E. Whitney: "The Siclops Silicon Compiler". Proc IEEE ICCC, pp. 277-280, Sept. 1982.

[Henderson 80] P. Henderson: "Functional Programming: Application and Implementation". Prentice-Hall, 1980.

[Henderson 82] P. Henderson: "Functional Geometry". Proc A.C.M Symposium on LISP and Functional Programming. 1982.

[Henderson, Jones, Jones 83] P. Henderson, G.A. Jones, S.B. Jones: "The LispKit Manual. Volume 1". Technical Monograph PRG-32(1), Programming Research Group, University of Oxford, 1983.

[Hoare 81] C.A.R. Hoare: "A Model for Communicating Sequential Processes". Technical Monograph PRG-22. Programming Research Group. University of Oxford. June 1983

[Hon Sequin 80] R.W. Hon, C. Sequin: "A Guide to LSI Implementation" Xerox, PARC, (second edition), Jan. 1980.

[Iverson 80] K.E. Iverson: "Notation as a tool of Thought". Communications of the A.C.M., Vol. 23, No. 3, pp 444-469, Aug. 1980.

[Johannsen 79] D. Johannsen: "Bristle Blocks: A Silicon Compiler". Proc. 16th Design Automation Conference (IEEE), 1979

[Johnson 81] S.D. Johnson: "Circuits and Systems: Applicative Examples". Proc. Symposium on Functional Languages and Computer Architecture. Gothenberg, June 1981.

[Kieburtz, Shultis 81] R.B. Kieburtz, J. Shultis: "Transformation of FP Program Schemes". Proc. Symposium on Functional Languages and Computer Architecture, Gothenberg, June 1981.

[Kramer, van Leeuwen 83] M.R. Kramer, J. van Leeuwen: "Systolic Computation and VLSI". In "Foundations of Computer Science IV: Part 1", J.W. de Bakker, J. van Leeuwen (eds), Mathematisch Centrum, Amsterdam, 1983.

[Kung 79] H.T. Kung: "Let's Design Algorithms for VLSI Systems". Technical Report CMU-CS-79-151, Dept of Computer Science. Carnegie-Mellon University. Jan. 1979.

[Kung, Leiserson 79] H.T. Kung, C.E. Leiserson: "Systolic Arrays for VLSI". Technical Report CMU-CS-79-103, Dept. of Computer Science. Carnegie-Mellon University. 1979.

[Leiserson 81] C.E. Leiserson "Area-efficient VLSI Computation". Ph D Thesis CMU-CS-82-108. Dept. of Computer Science. Carnegie-Mellon University. Oct. 1981.

[Lewke, Rammig 83] K.-D. Lewke, F.J. Rammig: "Description and Simulation of MOS Devices in Register Transfer Languages". In "VLSI 83". F. Anceau, E.J. Aas (eds), North-Holland, 1983.

[Locanthi 78] B. Locanthi "LAP: A SIMULA Package for IC Layout". Display File 21862, California Institute of Technology, 1978.

[Lyon 81] R.F. Lyon: "A Bit-Serial VLSI Architectural Methodology for Signal Processing". In "VLSI 81". J. Gray (ed), Academic Press, 1981

[Masuzawa, Nakauchi, Wada, Hagihara, Tokura 83] T. Masuzawa, S. Nakauchi, K. Wada, K. Hagihara, N. Tokura: "Systolic Algorithm Description Language, SADL, and Support System for Systolic Algorithm Design". Proc. International Symposium on VLSI Techniques, Systems and Applications,Taipei, Taiwan March 1983.

[McCabe, McCabe, Arambepola, Robinson, Corry 82] M.M. McCabe, A.P.H. McCabe, B. Arambepola, I.N. Robinson, A.G. Corry: "New Algorithms and Architectures for VLSI". G.E.C. Journal of Science and Technology, Vol. 48, No. 2, 1982.

[Mead, Conway 80] C. Mead, L. Conway: "Introduction to VLSI Systems", Addison-Wesley, 1980

[Milne 82a] G.J. Milne: "CIRCAL, a Calculus for Circuit Description", Copies of Slides, Feb. 1982

[Milne 82b] G.J. Milne "A Simple Silicon Compiler and its Correctness" Technical Report, Dept. of Computer Science, University of Edinburgh, June 1982.

[Milner 80] R. Milner: "A Calculus of Communicating Systems", Springer Verlag Lecture Notes in Computer Science, No. 92, 1980.

[Milner 82] R. Milner: "Calculi for Synchrony and Asynchrony", Technical Report, Dept. of Computer Science, University of Edinburgh, Feb. 1982.

[Mosteller 81] R.C. Mosteller: "REST - A Leaf Cell Design System" In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Moszkowski 83] B. Moszkowski: "Reasoning about Digital Circuits" Ph D. Thesis STAN-CS-83-970, Dept. of Computer Science, Stanford University, July 1983

[Mudge, Herrick, Walker 80] J.C. Mudge, W.V. Herrick, H. Walker "A Single-Chip Floating-Point Processor: A Case Study in Structured Design" In "Design Methodologies for VLSI Circuits", NATO Advanced Summer Institute, Belgium, 1980

[Noon 77] W.A. Noon: "A Design Verification and Logic Validation System" Proc. 14th Design Automation Conference (IEEE), June 1977.

[Preperata, Vuillemin 80] F.P. Preperata, J.E. Vuillemin: "Area-time Optimal VLSI Networks for Multiplying Matrices", Information Processing Letters, Vol 11, No. 2, Oct. 1980.

[Preparata, Vuillemin 81] F.P. Preparata, J.E. Vuillemin: "The Cube-Connected Cycles: A Versatile Network for Parallel Computation". Communications of the A.C.M., Vol. 24, No. 5, pp. 300-309, May 1981.

[Rem 81] M. Rem: "The VLSI Challenge: Complexity Bridling". In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Rem 82] M. Rem: "Partially Ordered Computations, with Applications to VLSI Design". Technical Report MR 82/3, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, 1982.

[Rem, van de Snepscheut, Udding 83] M. Rem, J. van de Snepscheut, J.T. Udding: "Trace Theory and the Definition of Hierarchical Components" Technical Report, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, 1983

[Revett, Ivey 83] M.C. Revett, P.A. Ivey: "ASTRA - A CAD System to Support a Structured Approach to IC Design". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983

[Roth 81] J.P. Roth: "Automatic Synthesis, Verification and Testing" In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Rowson 80] J.A. Rowson, "Understanding Hierarchical Design". Ph. D. Thesis, California Institute of Technology, 1980.

[Rubin 83] S.M. Rubin: "An Integrated Aid for Top-Down Electrical Design" In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983

[Rupp 81] C.R. Rupp: "Components of a Silicon Compiler System". In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Schoellkopf 83] J.-P. Schoellkopf: "LUBRICK: A Silicon Assembler and Its Application to Data-Path Design for FISC". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Segal 80] R. Segal: "SPAM". California Institute of Technology Silicon Structures Project, SSP MEMO £4029, 1980.

[Sheeran 81] M. Sheeran: "Functional Geometry and Integrated Circuit Layout". M. Sc. Dissertation, Programming Research Group, University of Oxford, 1981.

[Shrobe 83] H.E. Shrobe: "AI Meets CAD". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Shute 83] M.J. Shute: "The Role of Simulation in the Study of Multiprocessor, Control Flow, and Data Flow Systems". Ph. D. Thesis, Westfield College of the University of London, 1983.

[Siskind, Southard, Crouch 82a] J.M. Siskind, J.R. Southard, K.W. Crouch: "Algorithmically Specified Custom IC Design, An Example". Technical Report, M.I.T., 1982.

[Siskind, Southard, Crouch 82b] J.M. Siskind, J.R. Southard, K.W. Crouch: "Generating Custom High Peformance VLSI Designs from Succinct Algorithmic Descriptions". Proc. M.I.T. Conference on Advanced Research in VLSI, Jan. 1982.

[van de Snepscheut 83] J.L.A. van de Snepscheut: "Deriving Circuits from Programs". Technical Report, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, 1983.

[Steele, Sussman 80] G.L. Steele Jr., G.J. Sussman: "Design of a LISP-Based Microprocessor". Communications of the A.C.M., Vol. 23, No. 11, 1980.

[Subrahmanyam 83] P.A. Subrahmanyam: "Synthesizing VLSI Circuits from Behavioural Specifications: A Very High Level Silicon Compiler and its Theoretical Basis". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Sufrin 82] B. Sufrin: "Formal System Specifications. Notation and Examples" Class Notes, Programming Research Group, University of Oxford, Dec. 1982

[Suzuki, Burstall 82] N. Suzuki, R. Burstall: "Sakura A VLSI Modelling Language", Proc. M.I.T. Conference on Advanced Research in VLSI, Jan. 1982

[Thompson 80] C.D. Thompson: "A Complexity Theory for VLSI" Ph. D Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1980

[Weste, Ackland 81] N. Weste, B. Ackland: "A Pragmatic Approach to Topological Symbolic IC Design". In "VLSI 81", J. Gray (ed), Academic Press, 1981.

[Whitney, Mead 83] T. Whitney, C. Mead: "Pooh: A Uniform Representation for Circuit Level Designs". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.

[Williams 77] J.D. Williams: "Sticks - A New Approach to LSI Design" M.S.E.E. Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., 1977

[Williams 78] J.D. Williams: "STICKS - A Graphical Compiler for High Level LSI Design", Proc. N.C.C., May 1978.

[Williams 81] J. Williams: Working Material for Newcastle Functional Programming Course, July 1982.

[Wu, Parker, Conner 83] K.-H. Wu, A.C. Parker, K. Conner: "Procedural Layout: Some Practical Experience for Production-Quality Integrated Circuits". In "VLSI 83", F. Anceau, E. J. Aas (eds), North-Holland, 1983.