muFP, a language for VLSI design

Mary Sheeran
Oxford University Computing Laboratory
Programming Research Group
8 Keble Road
Oxford OX1 3QD

Introduction

In this paper, we present a VLSI design language μFP, which is a variant of Backus' FP [Backus 78, 81]. μFP differs from conventional VLSI design languages in that it can describe both the semantics (or behaviour) of a circuit and its layout (or floorplan) [Sheeran 83].

We chose to base our design language on FP for several reasons. Functional programs are easier to write and to reason about than imperative ones. We hope to bring some of these benefits to IC design. FP, in particular, is designed to allow the programmer to reason about his or her programs by manipulating the programs themselves. Likewise, in μFP, 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 are concise and 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 floorplan. This interpretation exists because μFP expressions represent functions rather than objects, allowing us to associate a function with each section of the floorplan. Most VLSI design languages are designed either for layout description or for behavioural specification. μFP, with its dual interpretation, allows the designer to consider the effect on the final layout of a particular design decision or to manipulate the layout while keeping the semantics constant. In the following sections, we show how μFP is constructed from FP by the addition of a single combining form μ, which encapsulates a very simple notion of "state".

A Brief Introduction to FP

A program in FP is 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 into their sum. The objects on which our programs operate can be undefined (⊥), atoms or sequences of objects. We shall take the set of atoms to be the integers, with a "don't care" value, '?'. <. . .> denotes a sequence. Some possible objects are ⊥, 42, <> (the empty sequence) and <1, <1, <1, 0>>>. 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, 2, . . . i : <x1, x2, . . xn> = xi if n ⩾ i, ⊥ otherwise.
   append to the left, apndl e.g. apndl : <3, <4, 5, 6>> = <3, 4, 5, 6>.
   matrix transposition, zip e.g. zip : <<1, 2, 3>, <4, 5, 6>> = <<1, 4>, <2, 5>, <3, 6>>.
2) Arithmetic functions
   e.g. +, -, * + : <1, 2> = 3 + : <42> = ⊥ * : <1, 3, 5> = ⊥.
3) Predicates (we denote true by 1, false by 0)
   e.g. greater than gt : <4, 1> = 1 (true) not : 1 = 0.

© 1984 ACM 0-89791-142-3/84/008/0104 $00.75

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. Appendix 1 contains a list of the CFs which interest us. We can use the CFs and primitive functions to write new functions. A typical FP program is that which computes the length of a sequence :-

$$\text{length} = (/R +) \circ \alpha \underline{1}.$$

This remarkably short program treats each element of the sequence as a 1 ($\alpha \underline{1}$) and then ($\circ$) adds them up (/R +). It is important to note that $\circ$ "does" the function on the right "first". The combining forms of FP obey a series of algebraic laws, some of which are listed in Appendix 2. These identities follow from the definintions of the CFs and require no proof in the algebra.

The Basic Building Blocks of muFP

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. (Note that we describe only synchronous circuits.) This switch to the use of streams for input and output is necessary because we need to deal with the concept of state. If we were dealing only with combinatorial (or stateless) circuits, the original FP would be sufficient. However, most digital circuits, from shift registers to microprocessors, have some "memory" and we will introduce a new CF, μ, to deal with these sequential circuits. If we are to describe such circuits, we must use streams for input and output, since the output of a circuit with state may depend on all of its previous inputs. Thus, the + function in μFP takes the input stream <<1,2>,<3,4>,<5,6>,<7,8>, . .> and produces the output stream <3, 7, 11, 15, . . .>. This difference between μFP and FP is reflected in our first semantic equation. We introduce a "meaning" function, M, which gives the semantics of μFP in terms of FP. A stateless function is one which does not contain μ, the CF which gives state, that is, a purely combinatorial function.

$$f \text{ stateless} \Rightarrow M\{f\} = \alpha f \qquad \text{I}$$

Equation I is our base case. α means "apply to all". The meaning of a μFP function f which contains none of the combining forms (besides the constant function) is just αf (in FP). This gives us functions which work in a repetitive manner on a stream of inputs to produce a stream of outputs, as in the + example above. The constant function $\underline{r}$, is just a source of rs.

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

Equation II says that the meaning of two composed functions is the composition of the meanings of the functions, as one might expect. Function composition also has the geometric interpretation illustrated in FIG 1(a).

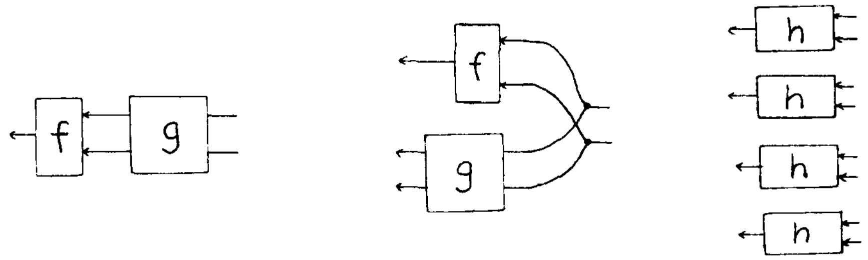


FIG 1. (a) composition $f \circ g$ (b) construction [f, g] (c) apply to all αh

For the "construction" CF (FIG 1(b)), we use zip, the matrix transpose function, to keep the types right. Zip is used in this way whenever we wish to convert a stream of tuples to a tuple of streams, or vice versa. It will be found throughout the semantic equations.

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

If the zip wasn't there, then the output of the right hand side, for m successive inputs, would be n m-element sequences, instead of m n-element sequences. For the "apply to all" CF (FIG 1(c)), we have to "massage" the types on both sides.

$M\{\alpha f\} = \text{zip} \circ \alpha M\{f\} \circ \text{zip}$ IV

To spread inputs along a row of identical cells, as shown in FIG 2, we use /L or /R.

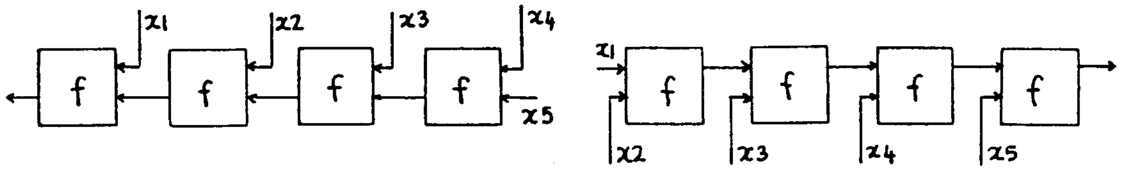


FIG2. /R f : <x1, x2, x3, x4, x5> /L f : <x1, x2, x3, x4, x5>

$M\{/R\ f\} = /R\ (M\{f\} \circ \text{zip}) \circ \text{zip}$ V

$M\{/L\ f\} = /L\ (M\{f\} \circ \text{zip}) \circ \text{zip}$ VI

For the conditional CF (FIG 3), $M\{p \rightarrow g\ ;\ h\}$ 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}.

$M\{p \rightarrow g\ ;\ h\} = \alpha(1 \rightarrow 2\ ;\ 3) \circ \text{zip} \circ [M\{p\}, M\{g\}, M\{h\}]$ VII

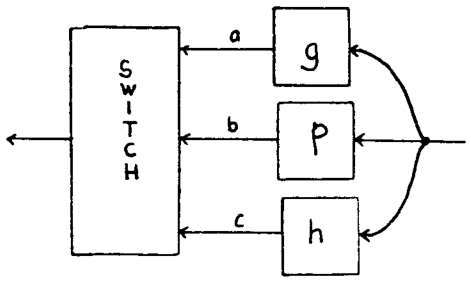


FIG 3. conditional $p \rightarrow g\ ;\ h$

The switch, represented by $\alpha(1 \rightarrow 2\ ;\ 3)$ in the equation, chooses between inputs a and c according to the value of b.

The seven semantic equations given above show how FP can be extended to use streams for input and output. In the following section, we introduce a new combining form, μ, to deal with the concept of state. The addition of μ has a surprisingly small effect on the algebraic properties of the language, as we will demonstrate in the next section.

Dealing with State

The new combining form μ takes a function and produces a "function" which has internal state. FIG 4 shows its geometric interpretation.

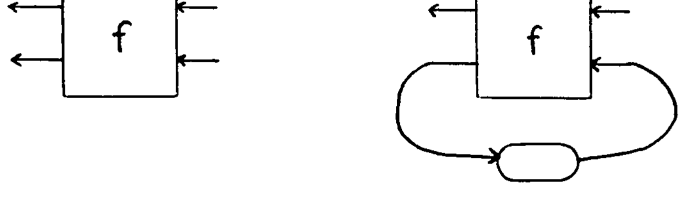


FIG 4. f μf

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.

Formally,

$$M\{\mu f\} = \text{out } M\{f\}$$

where out g i = o

where $\langle o, s\rangle = \text{zip} \circ g \circ \text{zip} : \langle i, ? \mid s\rangle$ 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 "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. For a given cycle, the next output and the next state depend on the current input and the current state.

A simple example of the use of μ should make its operation more clear. We would like to describe SR1, a shift register cell whose output is its current state and whose new state is its 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 5 shows how we have used the internal latch to give us a shift register cell.

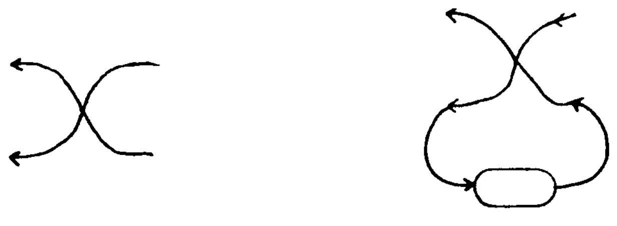


FIG 5 [2, 1] $\mu[2, 1]$

For input <0, 1, 0, 0, 1, 0, . . >, the output from $\mu[2, 1]$ would be <?, 0, 1, 0, 0, 1, 0, . . >, as one would expect from a shift register cell.

If μFP is to be useful as a design language, we must be able to manipulate our circuit descriptions to extract from them the information that we require. We need to find (and prove) some theorems or algebraic identities about the language. Appendix 2 lists some algebraic laws of FP [Williams 81]. A surprisingly large number of these laws remain true in μFP. For example, to prove that, in μFP,

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

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

$= \alpha(1 \rightarrow 2;\ 3) \circ \text{zip} \circ [P, F, G] \circ H$ VII

$= \alpha(1 \rightarrow 2 \,;\, 3) \circ \text{zip} \circ [P \circ H, F \circ H, G \circ H]$ A5

$= \alpha(1 \rightarrow 2 \,;\, 3) \circ \text{zip} \circ [M\{p \circ h\}, M\{f \circ h\}, M\{g \circ h\}]$ II

$= M\{p \circ h \rightarrow f \circ h \,;\, g \circ h\}$ VII Q.E.D.

In fact, of the 11 laws listed in Appendix 3, the only one which does not "translate" to μFP is A1. If we try to prove that, in μFP,

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

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.

In the search for new μFP laws, we have found the geometric interpretation of the combining forms very useful, since two μFP expressions which have exactly the same "picture" must also have the same semantics. As well as "reusing" old FP laws, we have derived some new laws concerning μ. The most important of these is that 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, as illustrated in FIG 6.

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

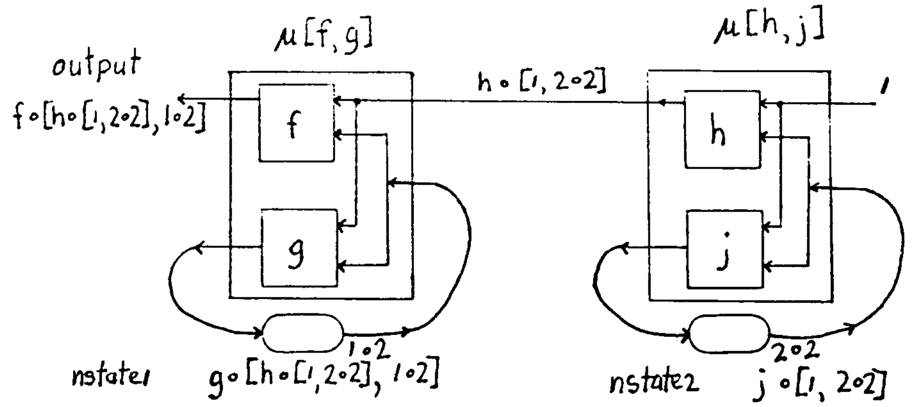


FIG 6. Derivation of $C\mu$ $\mu[f, g] \circ \mu[h, j] = \mu[\text{output}, [\text{nstate1}, \text{nstate2}]]$

Although the law looks complicated, its application is just a question of mechanical substitution. Since the useful laws of μFP are, in general, identities of this form, the language is suitable for use with an automatic transformation system. Finn [Finn 83] has written a simple μFP transformation system in 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.

Some Examples

The Tally Circuit [Mead, Conway 80]

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.

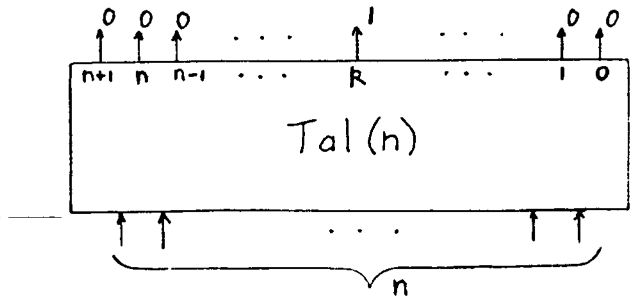
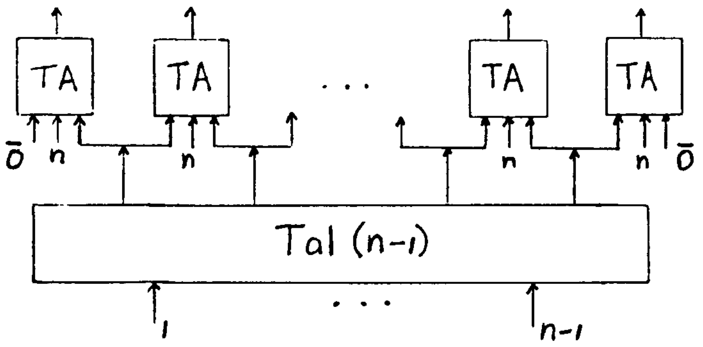


FIG 7. an n bit tally

From FIG 7, 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 left hand group is the number of low inputs to the tally and the number in the right hand group is the number of high inputs. This view of the circuit allows us to give a recursive definition of its operation. To create a tally for n bits from one for n-1 bits, we look at the nth output, and depending on whether it is high or low, we add a 0 either to the right or to the left of the output of the n-1 bit tally. We can do this by creating both the possible outputs and selecting between them using the final input. We use TA, a basic cell which selects between two bits depending on whether a third bit is high. This is the basic cell used in the Mead and Conway circuit. FIG 8(b) shows how the selection is performed using two pass transistors controlled by the selecting input and its inverse. Only one pass transistor is on (or conducting) at any time and the selector input determines which of the other inputs passes through to the output.



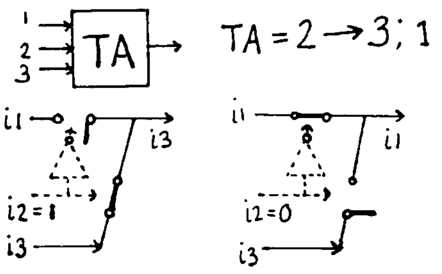


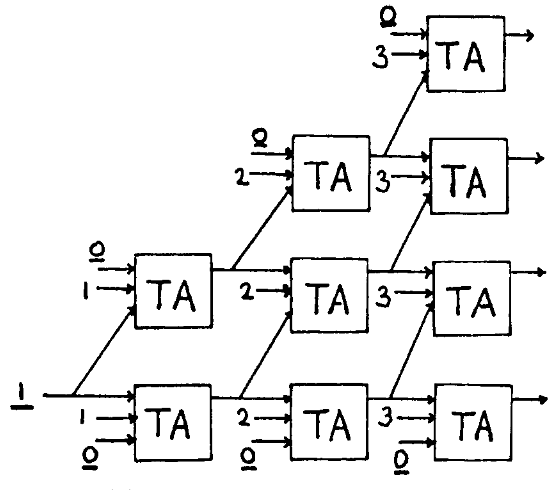
FIG 8. (a) An n input tally (b) The basic cell

We construct the two possible outputs and spread the first along the left hand inputs of the n+1 TA cells, and the second along the right hand inputs. Each tally has the nth input as its middle (or selecting) input and the result is an n input tally.

Tal(n) = αTA $_0$ zip $_0$ [apndl $_0$ [0, Tal(n-1)$_0$ mst], [n, n, . . n], apndr $_0$ [Tal(n-1)$_0$ mst, 0]]

where mst = reverse $_0$ tl $_0$ reverse

The basis of this recursive definition is Tal(0) = [1]. The list has no 0s since there are no inputs. FIG 9 shows our tally circuit for 3 inputs (i.e. Tal(3)) and the tally circuit from [Mead, Conway 80]. It is clear that we have described the circuit exactly. Our constant 0 (0) inputs correspond to Ground and our call of Tal(0) corresponds to VDD.



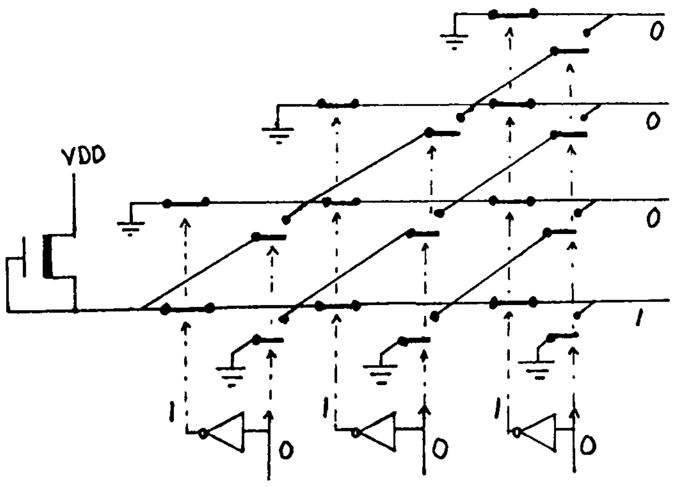


FIG 9. (a) Tal(3) (b) The original tally circuit

Other examples which we have tackled range from the construction of inverters from pullup and pulldown transistors to a formal derivation of a systolic correlator circuit which has been designed and fabricated at GEC Hirst Research Laboratories, London. Because the form of μFP described here implies a simple "right to left" flow of data, we have added some new CFs to give vertical as well as horizontal data flow. These CFs are particularly suitable for describing regular arrays such as the correlator. A full derivation of the correlator is beyond the scope of this paper but we can give an informal description of the process. We wish to calculate

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

Our first description is at the word level and is illustrated in FIG 10. (Cells shown as circles are "stateless".)

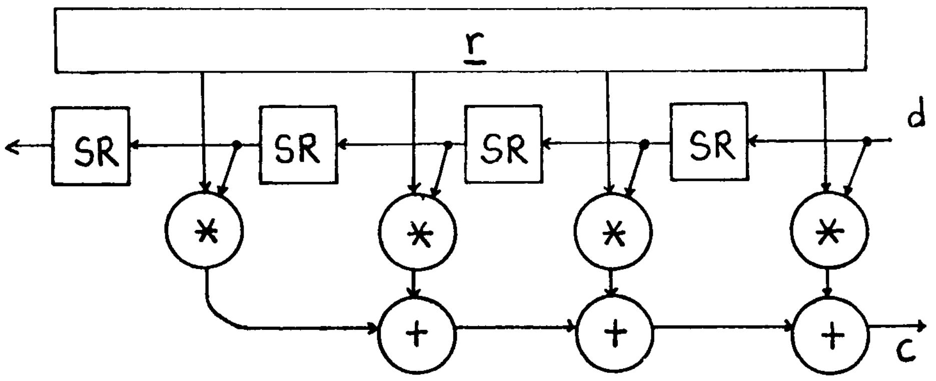


FIG 10. Our first attempt at the correlator
The row of Xcells shifts the data d across the circuit and does the necessary multiplications. The row of adders sums the N results of these multiplications. The reference data r are constantly fed in at the top, as shown. Using the algebraic laws of μFP, we transform this circuit first into a linear systolic array at the word level. This forces us to accept the existence of "don't cares" in the input and output streams. Next, we decompose our word-level processing elements into vertical bit-level systolic arrays to give the final orthogonally connected grid (FIG 11). Each basic cell (F) is just a gated full adder with latches on all of the outputs. The full derivation appears in [Sheeran 83].

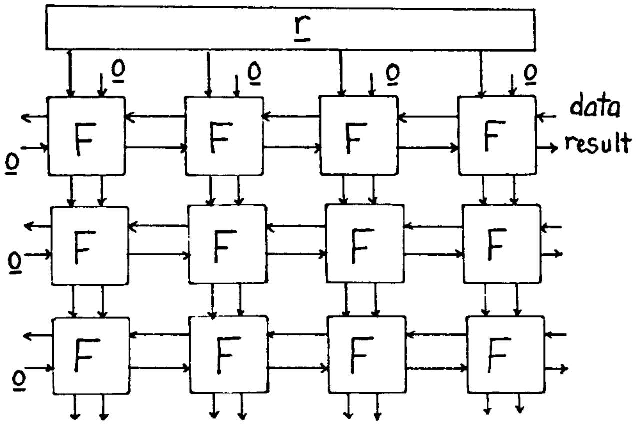


FIG 11. The final correlator circuit
We have used exactly the same technique to develop a systolic pipelined binary multiplier. Our approach seems particularly suited to the design of regular array architectures and we intend to develop a formal methodology and some tools for this purpose.

Discussion

Circuit descriptions can be run to give a simulator.

The denotational semantics of μFP given earlier can be considered to be a functional program. We can, therefore, run this program, giving it a μFP program and some inputs. It will then calculate the appropriate outputs, giving us a simulator. Such an interpreter for μFP has been written by John Hughes [Henderson, Jones, Jones 83]. We have also implemented the operational semantics of the language in Pascal, using an almost functional style. The program first constructs the abstract syntax tree of the μFP description. It then transforms this tree, using the information contained in the first input to the μFP program, to eliminate all αs and /s. This is possible because the first input gives us the "type" (i.e. the size and shape) of the data. To perform the simulation, we use a function, apply, which takes a transformed tree and an input (typed by the user) and works out the corresponding output, making the necessary state changes.

μFP can be translated to Functional Geometry to produce layout.

Functional Geometry [Henderson 82, Sheeran 81] allows us to describe pictures easily and readably, using a small set of geometric functions. The available functions are above, beside, rotate, flip and overlay, and the pictures are described in a hierarchical manner, using combinations of these functions. Similarly, in μFP, circuits are described in a hierarchical manner using the combining forms. The abstract syntax tree can be thought of as representing a picture. The leaves of the tree are represented by the pictures corresponding to the circuit layouts for those basic functions. Selector functions, which are also leaves, correspond to wires in the circuit. These basic pictures are combined by using functional geometry to implement the geometric interpretation of the combining forms, to give the final layout. As a first step towards the production of actual circuit layout, we have written a program which draws a "sized" floorplan for a given μFP expression. The program demonstrates that it would be possible to produce actual layout provided we were given the artwork for the basic cells, detailed information about their inputs and outputs, and the design rules which govern the placement of "wires". However, much work remains to be done in this area. For instance, we have not yet considered the problems of power and ground distribution. We intend to work on this area in collaboration with workers who have a strong background in layout techniques. Initially, we will concentrate on regular array architectures which combine mathematical tractability with ease of layout.

μFP has nice algebraic properties, despite having state.

We have added a very restricted notion of state to FP. Data values can be explicitly "remembered" for one clock cycle. This amount of state allows us to write circuit descriptions in the form of finite state machines, where we give a next output and a next state function. Thus, we can describe any circuit which is suitable for implementation on silicon. Surprisingly, we have gained this power without paying a high price for it. Many of the theorems of FP hold also in μFP and the new CF, μ, obeys simple laws. In particular, we can combine two functions with state into one with larger state. It is important to note that we can also apply this law in reverse.

In fact, the process of design can be viewed as the repeated application of laws such as this. For sequential circuits, the most abstract form is one which has a single μ on the outermost level. This form specifies a combinatorial block and a register bank through which signals are fed back. The designer must find a more elegant and efficient implementation of this behaviour on silicon. He does this by transforming the μFP description of the circuit, and hence its corresponding layout, using the algebraic laws. As the design proceeds, the combinatorial and the memory elements become more and more "mixed up". The designer hopes to reach a satisfactory layout in which memory elements are placed as near as possible to where they are "needed", thus minimising expensive interconnections. 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.

Conclusion

We have shown that μFP, an extension of FP, is suitable for use as a VLSI design language. Reasoning in μFP is at the function level, which allows us to associate a floorplan with each μFP expression by giving a simple geometric interpretation for each of the CFs. We have added a notion of state to FP, but have retained many of the algebraic properties of the language. μFP can describe both combinatorial circuits and those which can be represented as finite state machines. Because μFP combines semantic and geometric information, the process of design can be seen as the application of program transformation to an initial "abstract" specification, to produce the final "efficient" version of the circuit. The transformations which are applied to the circuit description must correspond to valid algebraic laws of the language, if the final circuit is to have the same semantics as the original one. Thus, design and verification go hand in hand. This is very important if we are to achieve our goal of producing verifiably correct layout from high level circuit descriptions.

References

[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.

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

[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" Tech. Monograph PRG-32(1), Programming Research Group, University of Oxford, June 1983.

[Mead, Conway 80] C. Mead, L. Conway: "Introduction to VLSI Systems" Addison-Wesley, 1980.

[Sheeran 81] M. Sheeran: "Functional Geometry and Integrated Circuit Layout" M. Sc. Dissertation, Programming Research Group, University of Oxford, Sept. 1981.

[Sheeran 83] M. Sheeran: "μFP, an Algebraic VLSI Design Language" D. Phil. Thesis, Programming Research Group, University of Oxford, 1983.

[Williams 81] J. Williams: "Notes on the FP Style of Functional Programming" in "Functional Programming and its Applications", Cambridge University Press, 1981.

Appendix 1

The combining forms of FP

Composition	$(f \circ g) : x = f : (g : x)$
Construction	$[f1, f2, .. fn] : x = \langle f1{:}x, f2{:}x, .. fn{:}x \rangle$
Apply to all	$\alpha f : x = \langle f{:}x1, f{:}x2, .. f{:}xn \rangle$ if $x = \langle x1, x2, .. xn \rangle$, $\perp$ otherwise
Conditional	$(p \rightarrow f ; g) : x = f{:}x$ if $p{:}x = 1$, $g{:}x$ if $p{:}x = 0$, $\perp$ otherwise
Constant	$\underline{r} : y = r$ if $y \neq \perp$, $\perp$ otherwise
Insert left	$(/L\ f) : \langle x \rangle = x$, $(/L\ f) : \langle x1, .. xn \rangle = f : \langle (/L\ f) : \langle x1, .. xn\text{-}1 \rangle, xn \rangle$
Insert right	$(/R\ f) : \langle x \rangle = x$, $(/R\ f) : \langle x1, .. xn \rangle = f : \langle x1, (/R\ f) : \langle x2, .. xn \rangle \rangle$

Appendix 2

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) $(/L\ f) \circ [g1, .. gn{+}1] = f \circ [(/L\ f) \circ [g1, .. gn], gn{+}1]$

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

(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) $\alpha f \circ \text{apndl} \circ [a, b] = \text{apndl} \circ [f \circ a, \alpha f \circ b]$

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

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