vFP: An Environment For The Multi-Level Specification, Analysis, and Synthesis Of Hardware Algorithms

Dorab Patel
Martine Schlag
Milos Ercegovac

January 1986
CSD-860052

Appeared in the Proceedings of the 1985 Functional Programming Languages & Computer Architecture Conference, Nancy, France, J. P. Jouannaud (Ed.) Lecture Notes in Computer Science 201, Springer Verlag, 1985, pp. 238-255.

vFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms

Dorab Patel, Martine Schlag and Miloš Ercegovac
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, USA

Abstract

This paper describes a method based on applicative languages for the specification, evaluation and synthesis of hardware algorithms. The goal of the research effort is to provide designers with an environment in which they can rapidly explore alternative designs for their algorithms throughout the synthesis process. It is possible to specify the algorithm at arbitrary levels of abstraction and have the system rapidly evaluate certain parameters (e.g. speed, area, etc.) so that designers can make informed decisions during the synthesis process. Layouts which are suitable as floor plans are extracted from high-level algorithms.

1 Introduction

The complexity of VLSI design can only be managed by the application of CAD tools at all levels of the design process. In order to be effective, these tools must be flexible enough to be tailored to any specific design. Generally, VLSI CAD tools may be distinguished as being of either or both of two types: bottom-up composition tools or top-down synthesis tools. For bottom-up composition tools, the user either exactly specifies the placement of modules and the interconnections between them, or relinquishes control over the layout to the tool's algorithm. Examples of composition tools are graphic layout editors (e.g. Caesar, Magic) [Ousterhout81, Ousterhout84] and placement and routing tools [Rivest82]. Top-down synthesis tools are capable of generating layouts from high-level specifications. Examples would include various register-transfer silicon compilers that have been proposed and built [Siskind82, Director81, Johannsen79]. Generally, these tools do not provide any estimate of the area or delays of the circuit during the synthesis process. That is, designers do not know the effects of their decisions on the performance until the design is complete.

Many of the current design approaches were largely developed for SSI/MSI technologies and are limited because of:

- Lack of paradigms to deal with topological and geometrical aspects of algorithm design in a hierarchical, multi-level fashion.
- Lack of adequate methods to deal with communication requirements of VLSI implementations during a multi-level algorithm design process.
- Lack of an adequate interface to lower-level VLSI CAD tools; most systems require a logic

diagram at the entry level, thus forcing designers to cope with details which are apt to be changed later in the implementation process.

– Lack of visual feedback; graphical representations, generated automatically from a high-level algorithm, showing details selected by the designer are highly desirable.

This paper describes a method based on applicative languages [Backus78] for the specification, evaluation and synthesis of hardware algorithms. This method is supported by a set of tools that is being developed at UCLA. The goal of this effort is to provide designers with an environment in which they can rapidly explore various alternative designs for their algorithms. Thus, it is possible to specify the algorithm at any arbitrary level of abstraction and have the system rapidly evaluate performance parameters (e.g. speed, area, etc.) so that designers can make informed decisions during the synthesis process. The advantage of using an applicative language is that it ties together the specification of the algorithm, the synthesis of the circuit and the evaluation of the implementation.

Others have explored incorporating applicative languages in VLSI design and have shown them to be viable. Lahti [Lahti81] used an applicative language to describe various combinational hardware structures. Johnson [Johnson84] utilized a demand-driven applicative language to describe and synthesize sequential digital circuits. Cardelli and Plotkin [Cardelli81] take a formal approach to describing sequential circuits with an emphasis on verification. Meshkinpour [Meshkinpour85] and Sheeran [Sheeran84] extended Backus' FP language with operators to handle sequential circuits.

2 Brief Introduction to νFP

νFP extends the language FP proposed by Backus [Backus78] with additional functional forms and primitives. In contrast to μFP [Sheeran84], which extends FP's semantics to operate on streams, the semantics of νFP are the same as those of FP when it is used to specify algorithms. A program in νFP (as in FP) is a function that maps objects into objects. Objects are either atomic (numbers or strings) or sequences of objects. The distinguished atom $\perp$ denotes an undefined value. By definition, any sequence which contains $\perp$ as an element is itself undefined and thus equal to $\perp$. The primitive functions of νFP consist of

arithmetic functions,	+ : (1,5) → 6	* : (3,2) → 6
logical functions,	andg : (1,0) → 0	org : (0,0) → 0
predicates,	atom : (1,2) → F	= : (3,3) → T
selector functions,	3 : (2,(4,5),6,(8,(9,10))) → 6	last : (1,4,6) → 6

and structure modifying functions.

trans : ((1,2,3),(4,5,6)) → ((1,4),(2,5),(3,6)) apndl : (1,(2,3,4)) → (1,2,3,4)

distl : (x, (a,b,c)) → ((x,a),(x,b),(x,c)) distr : ((a,b,c),x) → ((a,x),(b,x),(c,x))

Functional forms are used to combine primitive functions into more complex functions.

compose	(f @ g) : x → f : (g : x)
construct	[f,g,h] : x → (f:x, g:x, h:x)
apply to all	&f : (p,q,r) → (f:p, f:q, f:r)
constant	%k : x → k if x is not $\perp$
right insert	! f : $(x_1,...x_n)$ → f:(x_1, ! f:(x_2,...x_n))
tree insert	f : $(x_1,...x_n)$ → f:(f:(x_1,...$x_{\lceil n/2 \rceil}$), f:($x_{\lceil n/2 \rceil+1}$,...x_n))

A major syntactic difference between νFP and Backus' FP is that parameters to functions may be named and then referred to in the function body with the same restrictions as described in [Backus81]. In addition, the arithmetic, logical, and predicate functions may be used either in a prefix or an infix manner. This improves the readability of hardware specifications. For example, the following definition of a *FullAdder* in Backus' FP,

```
FullAdder =
  [org@[org@[andg@[1,2],andg@[2,3]],andg@[1,3]], xorg@[1,xorg@[2,3]]]
```

could alternatively be written in νFP as

```
defun FullAdder(a,b,Cin)
  [((a andg b) org (b andg Cin)) org (a andg Cin), a xorg (b xorg Cin)]
enddef
```

Owing to the natural specification of parallelism in FP-like languages, they are suited to describing parallel hardware algorithms. These specifications are executable. Since such programs are *referentially transparent*, it is possible to have an algebra of programs which may be used to reason about their behavior. These methods may be used in conjunction with each other to convince the designer that the program implements the envisioned algorithm. Specifications can also be executed symbolically using a symbolic input during which it is possible to extract the topological structure of the algorithm. Therefore, there is a direct relationship between the structure of an algorithm written in νFP and the planar topology of its layout.

3 Algorithm Synthesis in νFP

The designer first specifies the algorithm in νFP. The algebra of νFP programs may be used to reason about the algorithm. In addition, the specification may be executed with sample data to validate the program.

νFP can be used to describe circuits at various levels of abstraction. Designers are free to choose whichever level is ''best'' for their current purposes. At some higher level of abstraction, the structure of the *FullAdder* may not be relevant, and thus the definitions given earlier would suffice to describe its behavior. At a lower level of abstraction, where the structure of a function is to be considered, an alternative definition which has a different structure may be substituted. For example, the *FullAdder* could be defined in terms of *HalfAdder*s.

```
defun FullAdder(a,b,Cin)
  [1 or 2, 3] @ apndl @ [1, HalfAdder@[2,3]] @ apndr @ [HalfAdder@[a,b], Cin]
enddef

defun HalfAdder(a,b)  [a andg b, a xorg b]  enddef
```

Transformations may be used to refine the program to whatever level of detail is required. In this way it is possible to first specify the algorithm at a level of abstraction that is high enough to aid validation and then refine it to the level at which it can be easily implemented.

4 The Evaluation of νFP Algorithms

It is possible to *tag* selected user-defined functions so that when a νFP specification is executed an estimate of the performance of the algorithm can be provided. Tagging a function tells the system that this is a function of interest at the current level of abstraction. As the execution proceeds, the interpreter keeps track of the *level* at which a tagged function is executed.

The *level* of a tagged function is defined as one plus the maximum of the *levels* associated with the atoms in its input object. The *level* of each atom is initially zero. Each time a tagged function is encountered, its level is determined and is assigned to the atoms in its output object. However, there is a problem when a tagged function occurs within another tagged function. In this case the level of the inner function is determined with respect to the outer function resulting in a hierarchy of levels. This is accomplished by assigning the level zero to each atom of the outer function's input object, and computing the level of tagged functions as before until the computation of the outer function has been completed. The level of the outer function and the atoms in its output object is determined as before and hence is independent of whether or not any tagged function occurs within the outer function. Levels are used to predict the speed at which the circuit would perform, to obtain an idea of where the parallelism in the algorithm is, and to get an estimate of the area that would be occupied by the circuit. A better estimate of the area is obtained by methods mentioned in the next section.

This capability of having the system estimate performance parameters is useful in tradeoff analyses. For example, consider the following function:

$$z = \begin{cases} 1 & \text{if } a=(b-1) \bmod 8 \\ 2 & \text{if } a=b \\ 0 & \text{otherwise} \end{cases}$$

schemeA and **schemeB**, below, are two algorithms for implementing the function. If the boolean functions (*andg*, *org*, *notg*, and *xorg*) are tagged, the results shown in Figure 1 are obtained.

```
# scheme A   inputs : ((a) (b))  outputs : (z1 z0)
defun schemeA
 &(org@&andg@trans)@distl@[1,[id,rotr]@2]@&decoder
enddef

# scheme B   inputs : ((a) (b))  outputs : (z1 z0)
defun schemeB(a,b)
 &compare @ distl @ [a, [b, tl @ adder @ [b, [%1, %1, %1]]]]
enddef

defun compare   notg@org@&xorg@trans   enddef

defun adder
 apndl@[xorg@[xorg@1,1@2],tl@2]
   @[1,!add@apndr@[tlr,HalfAdder@last]@tl] @ trans
enddef

defun add(a,b)   concat@[FullAdder@apndr@[a,1@b],tl@b]    enddef
```

statistics for schemeA

level	Andg	Org	Notg
1	2		6
2	10		
3	14		
4	14		
5		8	
6		4	
7		2	
Totals	40	14	6

statistics for schemeB

level	Andg	Xorg	Org	Notg
1	2	6		
2	1	2	1	
3		1	2	
4		1		1
5		1		
6			1	
7			1	
8				1
Totals	3	11	5	2

Figure 1: A comparison of two implementations

The results in Figure 1 show that *schemeA* uses a total of 60 gates, while *schemeB* uses a total of 21 gates. However, it is to be noted that 11 of the 21 gates are xor gates which would normally occupy a larger area. Given an estimate of the area occupied for each of the gates, it is possible to have an estimate of the area occupied by each implementation. Since *schemeA* has 7 levels while *schemeB* has 8, *schemeA* would be faster than *schemeB* under the assumption that all the tagged functions had the same delay,

In addition to the time and space estimates provided by the *level* mechanism, the system can be extended to allow the specification and calculation of user-specified parameters for each tagged function and for the algorithm as a whole.

5 Space Domain Implementations of vFP Algorithms

A vFP algorithm can be mapped into a structure corresponding to a combinational network by passing symbolic inputs to functions which in turn generate symbolic outputs. The unit of information represented by a symbolic atom can be anything corresponding to the level of abstraction. Thus, a symbolic atom may represent a wire, a set of wires, a bit vector, or an integer, as required. An acyclic computation graph with vFP primitives as nodes is obtained by tracing the application of a function to a symbolic input. This computation graph can be transformed into a layout using techniques described later. By tagging the appropriate functions, the layout may be generated at any desired hierarchical level. For example, Figure 2 shows a layout of a *FullAdder* using *and*, *or*, and *xor* gates; whereas Figure 3 shows the same *FullAdder* as being composed of *HalfAdders*.

The mapping from a vFP algorithm to a combinational network is allowed under the following restrictions. Functions like *iota*, whose output *structure* depends on an input *value*, cannot be laid out. In addition, during symbolic evaluation, the predicate part of a conditional must be evaluable to a boolean.

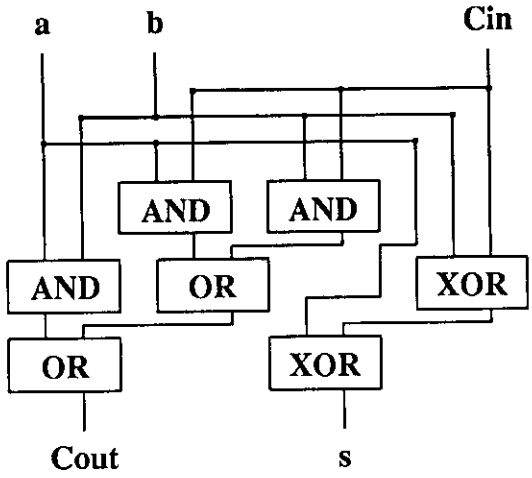

Figure 2: The structure of a FullAdder

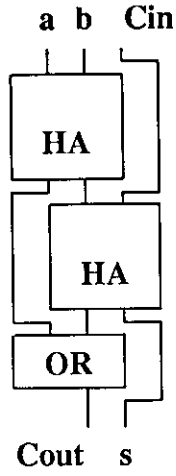

Figure 3: A FullAdder using HalfAdders

As in μFP [Sheeran84], *structural iterations* over the input of the circuit can be handled by the *insert* and *apply-to-all* functional forms. Other types of *structural recursions* are allowed in vFP since the *conditional* functional form is treated as a *structural* form for the purposes of layout. Depending on the value of the predicate of the conditional, either the consequent or the alternate part will be evaluated symbolically for its structure but no structure will be generated for the predicate part. A new primitive called sw (for switch) is provided in vFP which corresponds to the conditional form in μFP. This primitive takes three arguments. If the first is **1** then the output is the second argument; if it is **0** then the output is the third argument; else it is $\perp$. In addition, it is required that the *structures* of the second and third arguments be the same.

A vFP description of a circuit can be generic in the sense that the description is independent of the input dimensions of the circuit. For example, there needs to be only one description of a decoder. This same description works for a decoder independent of the number of inputs. The 3-to-8 decoder shown in Figure 4 is obtained by evaluating the description of the generic decoder with a symbolic argument of size 3. Figure 4 shows how the generic iterative decoder is formed by first applying 1-to-2 decoders (*Dec1*) to the inputs and then inserting the function *DecStage*. *DecStage* takes an n-to-2^n decoder and a new input to make a (n+1)-to-2^{n+1} decoder.

```
defun Decoder     !DecStage @ &Dec1   enddef

defun DecStage    &andg @ concat @ &distl @ distr    enddef

defun Dec1  [not,id]  enddef
```

As before, these implementations may be evaluated to get speed/area estimates, but now, since routing is taken into account, a better estimate of area can be provided.

Cell iterative networks are combinational circuits which are formed by interconnecting a particular cell in a regular pattern. Although combinational circuits without feedback can be described in vFP using the forms inherited from FP, some additional functional forms are provided to give designers more control over exactly how cell iterative networks are to be laid out. These networks are thus

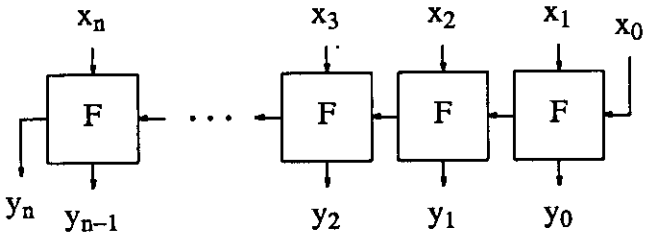

Figure 5: The **seq** functional form applied to F

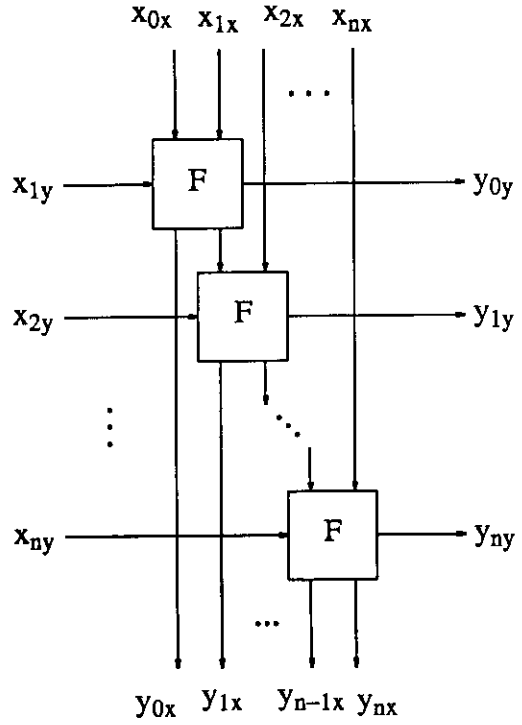

Figure 6: The **seqxy** functional form applied to F

readily described in vFP by invoking the form (corresponding to the interconnection pattern) on the function (corresponding to the cell). For example, Figure 5 shows the *seq* functional form pictorially. Sometimes it is necessary to have two inputs to the function F at each stage and to have one of those inputs come in from the x direction and the other from the y direction. This is accomplished by the *seqxy* functional form shown in Figure 6. Though both forms result in the same computation graph, their layout is different.

6 Time Domain Implementations of vFP Algorithms

As was shown in the previous section, virtually all vFP programs can be laid out. However, since iterations are unfolded in space, the resulting layout might occupy more area than is available. Also, the inputs to the circuit might be coming in one at a time along the same wire(s), serially, rather than all at once on separate wires, in parallel. In both these cases, it is convenient to implement the circuit (or parts of it) as a sequential circuit rather than a combinational circuit. In vFP this is done by replacing certain space-domain functional forms by their time-domain duals.

$$\&f \equiv D^{-1} \text{ @ POSI @ } \&^{T}f \text{ @ SOPI}$$

$$!f \equiv D^{-1} \text{ @ } !^{T}f \text{ @ apndr @ [SOPI @ tlr , last]}$$

$$\mathbf{seq}f \equiv D^{-1} \text{ @ apndl @ [1 , POSI @ tl] @ } \mathbf{seq}^{T}f \text{ @ apndr @ [SOPI @ tlr , last]}$$

In these equalities **POSI** is a *parallel-out-serial-in* shift register, **SOPI** is a *serial-out-parallel-in* shift register, and $\&^{T}$, $!^{T}$ and seq^{T} are the time-domain duals of the corresponding space-domain functional forms introduced earlier. The $!^{T}$ and seq^{T} forms use the first element of their input sequence as the initial value of the internal register (**R**). This makes it possible to specify the initial state of a sequential

system in vFP. D^{-1} is a phantom element that corresponds to an inverse time delay. It is used to keep track of the number of clock pulses the output is going to be delayed by. This information is needed by the *construct* functional form to synchronize its components since the semantics of the construct require that the outputs of its elements appear together. Generally the D^{-1} elements are moved, via transformations, to the outputs of the circuit where they serve to denote the delay.

When elements of a sequence are available serially in time along the same wire(s), it is necessary to know when each element is valid. This is accomplished, during symbolic simulation, by having each symbolic item carry the name of a clock with it. It is assumed that its value will be stable before every tick of the named clock. The system will automatically widen the intervals between clock ticks to ensure that this is true. Initially, all the inputs are associated with the same clock. Each combinational element will assign to its output the clock associated with its input. If there are n elements to the input sequence of a *SOPI*, then each of its output elements will be clocked by the clock $n\mathrm{C_k}$; and conversely for a *POSI*. A clock named $n\mathrm{C_k}$ denotes a clock which has n clock ticks in between consecutive ticks of the clock named $\mathrm{C_k}$. Though the description of a *SOPI* or *POSI* is generic, the value of n (the number of elements in the sequence) must be known at layout time.

As an example, consider a time-domain implementation of an inner-product algorithm (!+ @ &*). The straightforward implementation of the algorithm, using the equations given above, would result in the layout shown in Figure 7. Since there are two D^{-1} elements in the layout, the output will be delayed by two clock ticks from the input.

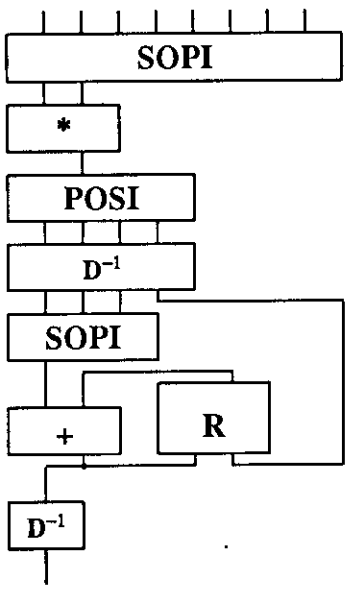

Figure 7: Initial Implementation of the Inner Product

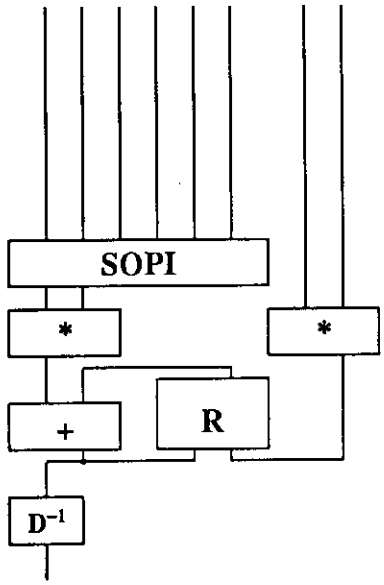

Figure 8: Optimized Implementation of the Inner Product

However, using the identities

$$D^{-1} \text{ @ POSI @ } f \text{ @ SOPI} \equiv \text{apndr @ [}D^{-1} \text{ @ POSI @ } \&^{T}f \text{ @ SOPI @ tlr , } f \text{ @ last]}$$

$$POSI \circ D^{-1} \circ SOPI \equiv SOPI \circ D^{-1} \circ POSI \equiv id$$

$$f \circ D^{-1} \equiv D^{-1} \circ f$$

$$[tlr , last] \circ apndr \equiv apndr \circ [tlr , last] \equiv id$$

the program may be transformed into the following

$$D^{-1} \circ !^{T}+ \circ apndr \circ [\&^{T}* \circ SOPI \circ tlr , * \circ last]$$

whose layout is shown in Figure 8. The single D^{-1} element denotes that the output is delayed only one clock tick from the input.

This implementation accepts all its inputs simultaneously and eventually gives its result. It will only work for input sequences of one particular length, since only fixed size *SOPI*s can be laid out. However, if each element of the input sequence was input serially to the implementation, a corresponding *POSI* could be introduced at the input and then used to transform out the *SOPI* that exists in the current implementation. This would make the implementation generic in the sense that it would be able to handle arbitrary length sequences as its inputs.

7 Layouts from νFP Specifications

In this section the mapping from νFP algorithms to layouts is briefly described. A more detailed exposition and a description of its implementation can be found in [Schlag84]. The intent of this system is to provide the νFP designer with an interactive environment in which the design can be viewed as it is constructed. The mapping from νFP is actually the composition of two mappings. An νFP function is first mapped to an intermediate form (IF) which reflects the planar topology of the function and then this IF is mapped to fixed geometry by selecting and resolving relative position constraints (compaction).

The rationale for dividing the mapping in two steps is the observation that a certain portion of the mapping from νFP should be functional even though the entire map cannot be. That is, a particular νFP function applied to a particular symbolic object should define an IF uniquely, while the geometry of the function should depend on its environment. Fixing the geometry of a sub-function may create wiring and shape incompatibilities with other sub-functions which would require additional area to resolve. Functionality has two advantages.

1. The mapping can be implemented as an application of an νFP function to an object.

2. Algebraic transformations on the νFP function have predictable effects on the IF.

The extraction of the topology (IF) of an νFP expression is implemented as an interpreter. A function applied to an object generates an IF and each combining form dictates a topological organization of the IFs of its sub-functions. The routing is the direct result of the routing primitives and the combining forms of the function. This implementation generates a sketch of the νFP specification in terms of ''boxes'' and ''wires'' by symbolically tracing the νFP function and representing each atom as a wire. The level of abstraction of the sketch can be controlled by selecting which functions to represent as

boxes and what objects each atom represents. The IF consists of a list of horizontal cross-sections each of which is a left to right ordering of the ''boxes'' and ''wires'' which intersect the cross-section. Any cross-overs are represented explicitly; each cross-section corresponds to a horizontal track. The IF generated by the interpreter is fed to a program which resolves the cross-sections using horizontal compaction and displays the sketch on a graphics terminal.

An example is presented to illustrate how vFP facilitates the transition from algorithm to implementation taking into account the layout. The vFP specification of a carry chain adder [Brent80] is considered. The specification is generic in that it adds two bit vectors of length 2^n for $n>0$. The input consists of the 2^n pairs of bits to be added with the leftmost pair, containing the least significant bits, $((a_1,b_1),(a_2,b_2),(a_3,b_3),\ldots,(a_{2^n},b_{2^n}))$.
For $1\le i\le j\le 2^n$,

$$P_{i,j}=\begin{cases}1 & \text{if a carry into column i would propagate as a carry out of column j}\\ 0 & \text{otherwise}\end{cases} \quad \text{and} \quad G_{i,j}=\begin{cases}1 & \text{if adding columns i through j causes a carry out of column j}\\ 0 & \text{otherwise}\end{cases}$$

The computation is performed by computing the carries for each column, $G_{1,i}$ and then obtaining the sum bit using, [7.1]

$$s_i=G_{1,i-1}P_{i,i} \text{ for } 1<i\le 2^n, \quad s_1=P_{1,1}, \quad \text{and } s_{2^n+1}=G_{1,2^n}$$

$P_{1,i}$ and $G_{1,i}$ are computed for each i by using the following identities, implemented by the function **PG**. [7.2]

$$\text{For } i\le j<h,\ P_{i,h}=P_{i,j}P_{j+1,h} \text{ and } P_{i,h}=(G_{i,j}P_{j+1,h})+G_{j+1,h}$$

The initial $P_{i,i}$ and $G_{i,i}$ are computed by the function **PG1**. [7.3]

$$P_{i,i}=a_i\ b_i, \quad G_{i,i}=a_ib_i \quad .$$

The computation of $P_{1,i}$ and $G_{1,i}$ is achieved in two steps by the function **getcarries**. The following is the specification of **getcarries**.

```
# input = ((a0,b0),(a1,b1),(a2,b2),...,(a2**n - 1,b2**n - 1))
defun getcarries    secondhalf@split@1@firsthalf@&PG1    enddef

defun firsthalf    if eq@[length,%1] then id else firsthalf@&stage1@pair fi    enddef

defun stage1    concat@[&D@1,&D@tlr@2,[PG@[last@1,last@2]]]    enddef

defun secondhalf    if eq@[length,%1]@1 then done else secondhalf@concat@
  [split@&D@1,stage2@tl]@apndr@[concat@&[id,last]@tlr,last] fi
enddef

defun stage2    concat@
  &([apndr@[&D@tlr@1@2,PG@[1,last@1@2]],&D@2@2]@[1,split@2])@pair
enddef

defun D id enddef
```

defun **done** id enddef

First *getcarries* computes $(P_{i,i},G_{i,i})$ by applying **PG1** to the two bits in each column and then it applies **firsthalf**. **firsthalf** computes $(P_{i-2^k+1,i},G_{i-2^k+1,i})$ for each column $i=(2m+1)2^k$ where m is an integer. This is accomplished by arranging each column (i.e. its pair (P,G)) in a group of its own and then recursively applying the function **stage1** to pairs of groups until only a single group remains. **stage1** combines a pair of groups computing a new (P,G) for the last column of the second group by applying **PG** to the last columns of the two groups; the pair of groups is then concatenated to form one group. All other columns are unchanged; the function **D** which is given the definition **id** is applied to them. When all columns are in a single group **getcarries** applies the function **secondhalf** to compute the final (P,G)'s. **secondhalf** is also recursive, terminating when each column is in a group by itself. At each step the final (P,G)'s of decreasing multiples of powers of 2 are computed. Assume that in the previous step $P_{1,i}$ and $G_{1,i}$ have been computed for each column $i=m2^k$. In the next step to compute the (P,G)'s of columns which are multiples of 2^{k-1}, it is necessary only to compute new (P,G)'s for columns $i=(2m+1)2^{k-1}=m2^k+2^{k-1}$, the odd multiples of 2^{k-1}. The current (P,G) in column i is $(P_{i-2^{k-1}+1,i},G_{i-2^{k-1}+1,i})$. $(P_{1,i},G_{1,i})$ can be obtained by applying **PG** to the current (P,G) and $(P_{1,m2^k},G_{1,m2^k})$. Initially the columns are divided into two groups and since **firsthalf** computed the final (P,G)'s for powers of 2, the last column (a multiple of 2^{n-1}) has its final value. At each step **secondhalf** duplicates the last column from each group and then applies **stage2** after removing the first group. **stage2** takes each group, splits it into two and computes new (P,G)'s for the last column in the left group of each new pair using the duplicated column immediately to the left of the group. The first group is then appended to the result of **stage2**.

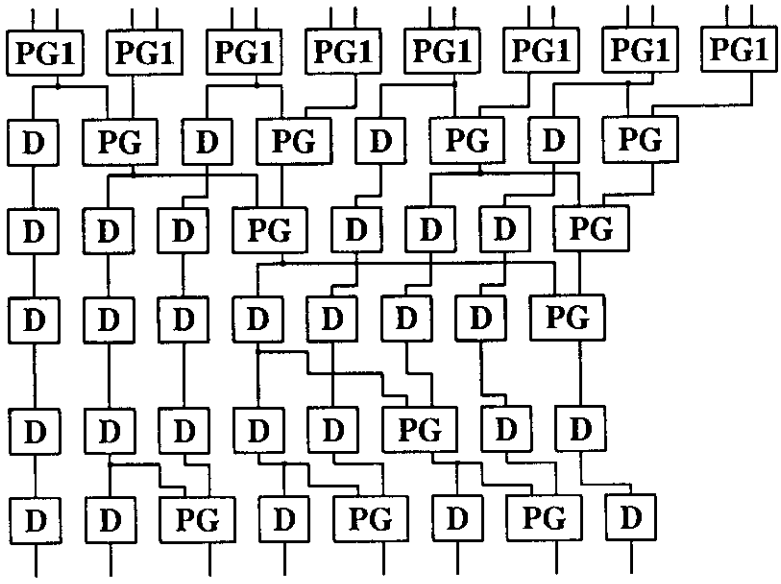

Figure 9: The sketch of **getcarries** with each *(P,G)* as a wire

Figure 9 is the sketch of **getcarries** in which the pair (P,G) for each column is represented by a single wire. This is accomplished by directing the interpreter to draw **PG1**, **PG** and **D** as boxes and by giving **PG1** and **PG** symbolic definitions.

(define-symbolic **PG1** input=(a b) output = (c))

(define-symbolic **PG** input=(a b) output = c)

(drawbox **PG1** label=PG1 ht=2)

(drawbox **PG** label=PG ht=2)
(drawbox **D** label=D ht=2)

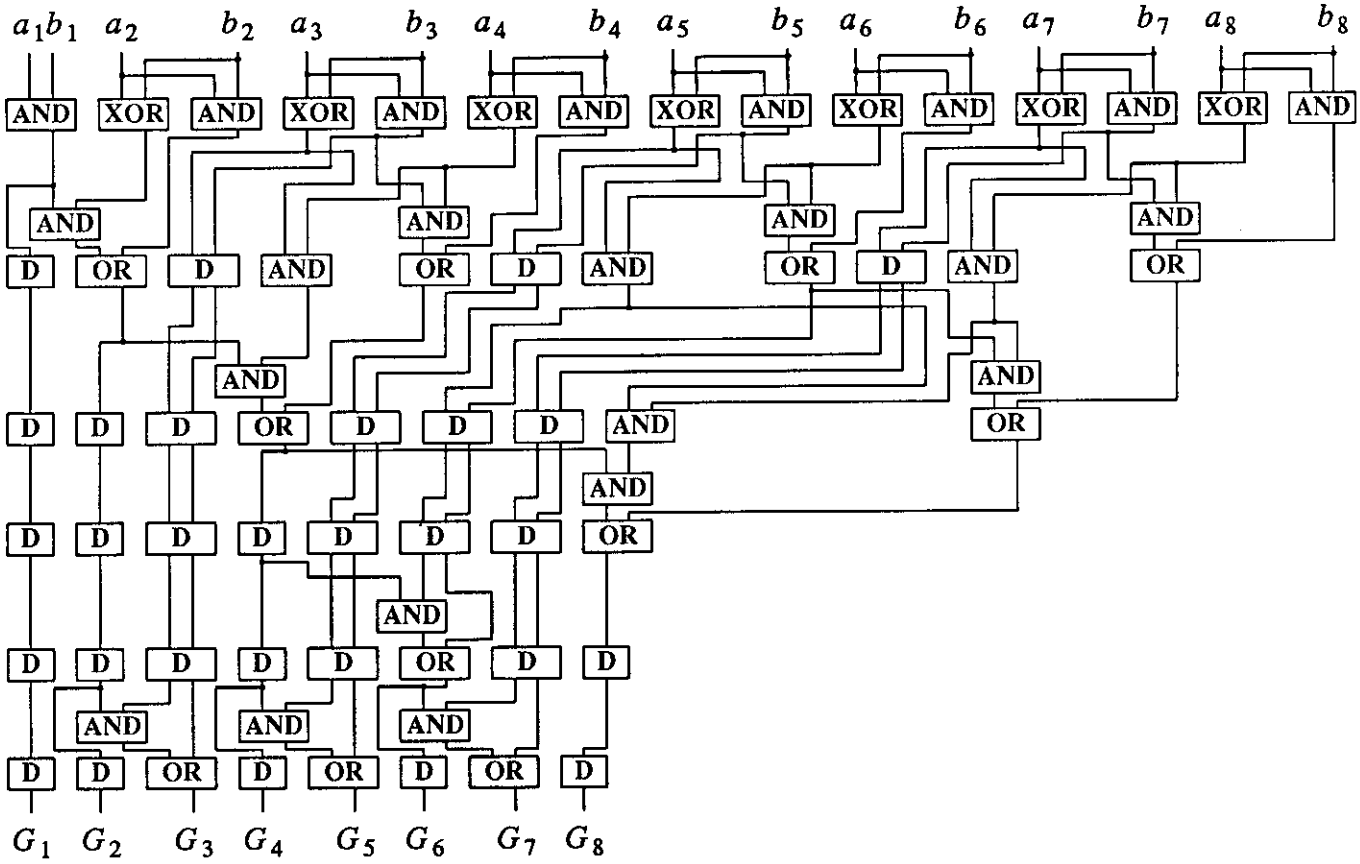

Figure 10: The sketch of **getcarries** with the definitions of **PG1** and **PG** filled in

Figure 10 is the sketch of **getcarries** with each wire corresponding to a bit this time and with the specifications of the functions **PG** and **PG1** "filled in." The definition of **done** is changed since only the G of each column is required to compute the final sum. Notice that the layout interpreter generates only those wires and boxes which have paths to an output. The previous specification would be extended as follows.

defun **done** &(2@1) enddef
defun **PG1** [[xorg,andg]] enddef
defun **PG** [andg@[1@1,1@2],org@[andg@[2@1,1@2],2@2]] enddef
(drawbox **D** label=D ht=2)

To obtain the final sum by (7.1) it is necessary to combine the first P in each column, $P_{i,i}$ with the G of its left neighbor $G_{1,i-1}$. One way of doing this would be to duplicate each $P_{i,i}$ generated by **PG1**, route them along the side and then merge them back into the columns to compute the final sum as in Figure 11. The additional area required for routing makes this an unattractive alternative. A better design would be to route the $P_{i,i}$ along with the (P,G) down its own column. This extension is easily handled in vFP by modifying the function **PG** so that it duplicates its first argument if it receives only two arguments in a column and simply passes on the extra argument otherwise. The specification is modified as follows.

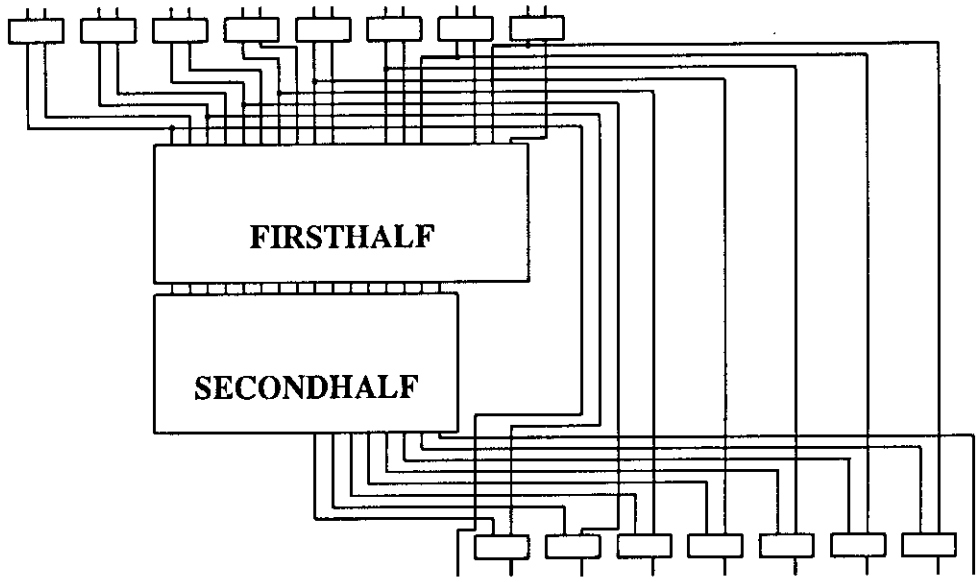

Figure 11: An inefficient design

```
defun add
  concat@[[1],&xorg@pair@tl@tlr,[last]]
      @concat@apndl@[1@1,&([1,3]@1)@tl]@getcarries
enddef

defun PG
  apndl@[D@1@2, if null@tl@tl@2 then oldPG else oldPG@[1,tl@2] fi ]
     @ if null@tl@tl@1 then id else [tl@1,2] fi
enddef

defun oldPG   [andg@[1@1,1@2],org@[andg@[2@1,1@2],2@2]]   enddef

defun done id enddef

(drawbox D label=D ht=2)
```

The function **add** applies **getcarries** and then handles the columns according to (7.1) to obtain the final sum bits. Figure 12 is the sketch of **add**.

Figure 13 is the sketch obtained of a carry save array multiplier. This example is presented to illustrate the geometric flexibility of fixing only the planar topology in the specification. The functions **HA***, **FA***, **FA****, and **HalfAdder** are represented as primitives. The specifications of the functions **HA***, **FA***, and **FA**** are

```
# FA* op2 : ((a b) (y x)) ---> ((c x) s) where 2c + s = (a + b + yx)
defun op2      [[org@[1,1@2],3],2@2]@
  [1@1,HalfAdder@[2@1,2],3]@[HalfAdder@1,andg@2,2@2]
enddef

# HA* op1 : ((a) (y x)) ---> ((c x) s) where 2c + s = (a + yx)
defun op1 [[1@1,2],2@1]@[HalfAdder@[1@1,andg@2],2@2] enddef
```

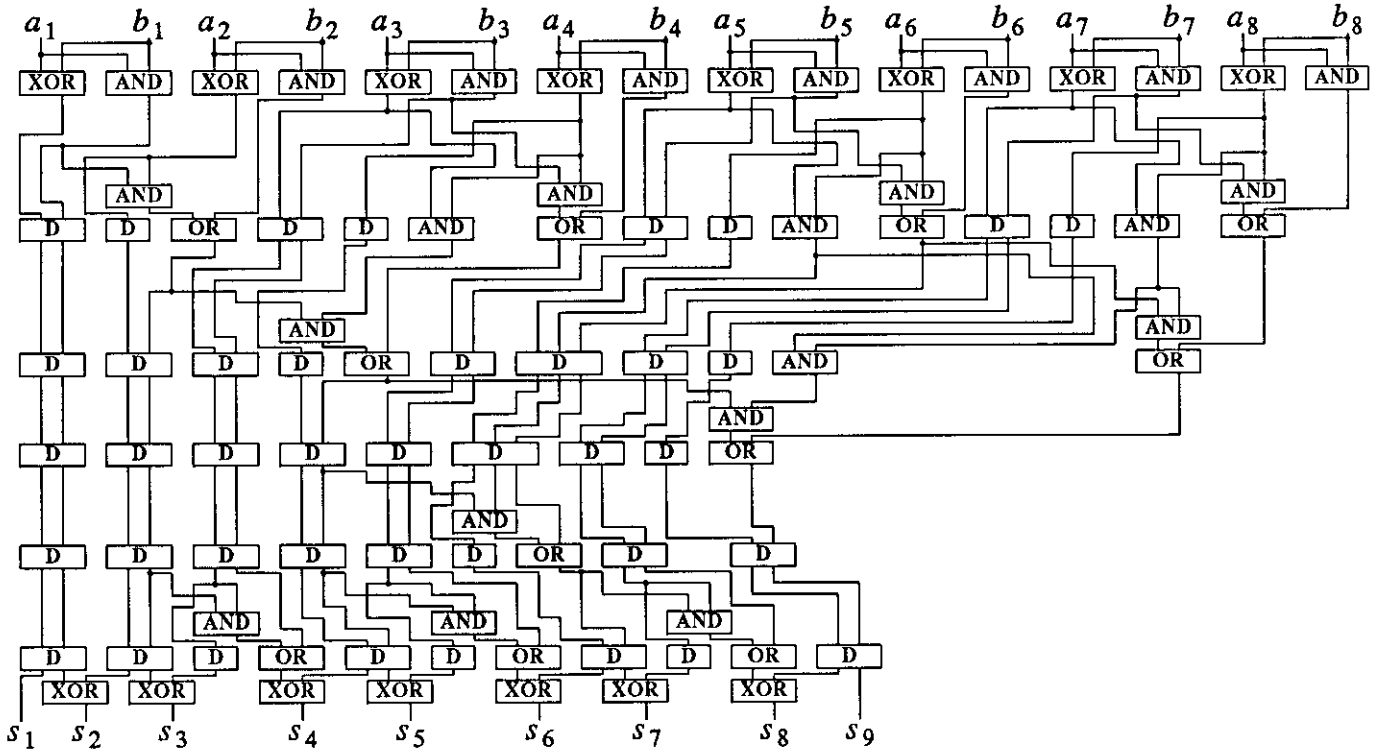

Figure 12: The sketch of **add**

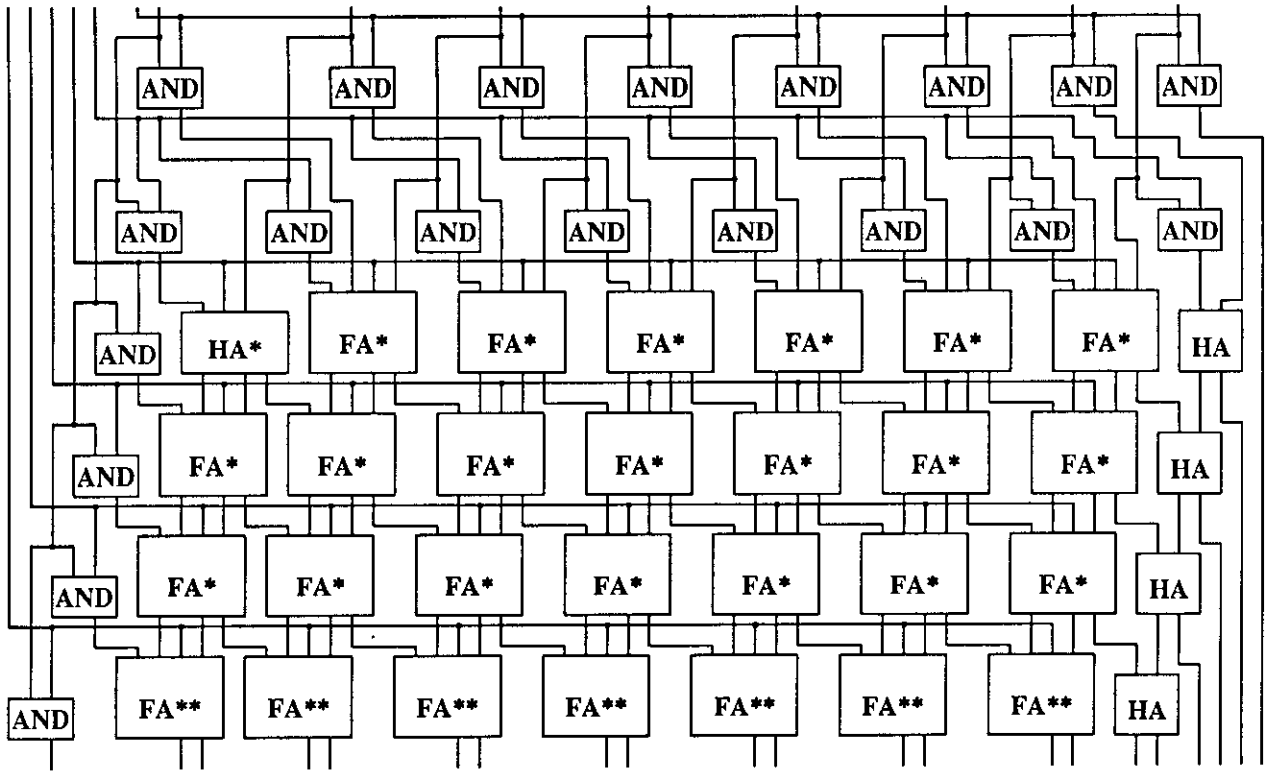

Figure 13: A Carry Save Array Multiplier with **FA***, **HA*** and **FA**** as primitives

```
# FA** lop2 : ((a b) (y x)) ---> (c s) where 2c + s = (a + b + yx)
defun lop2
  [org@[1,1@2],2@2]@[1@1,HalfAdder@[2@1,2]]@[HalfAdder@1,andg@2]
enddef

defun op0 [1,andg] enddef
```

Figure 14 contains the sketches of these functions, while in Figure 15 the same carry save array multiplier is represented in terms of lower level primitives. Note that the geometry of the functions **HA***, **FA*** and **FA**** varies; each instance has some flexibility in adapting to the particular geometric constraints it encounters.

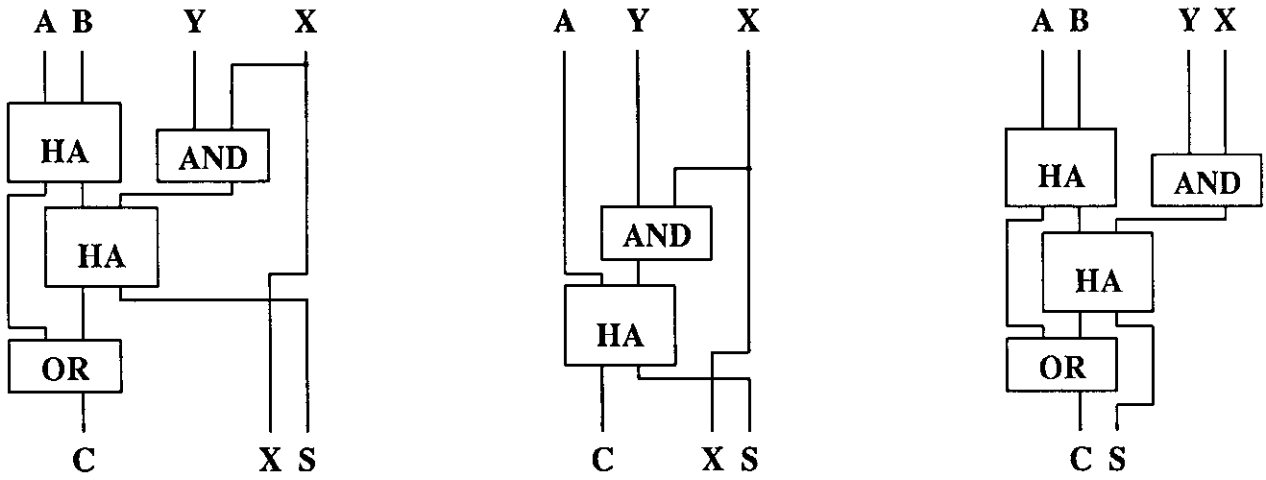

Figure 14: **FA***, **HA*** and **FA****

All of the figures in the paper with the exception of Figures 5 and 6 were generated by this system. It is limited in that the data flow is vertical with a function's inputs and outputs on the top or bottom. More sophisticated layout techniques which would not suffer from this limitation are being examined. However this system is useful in that it provides visual feedback quickly allowing the designer to see the planar implications of the specification.

8 Concluding Remarks

The objective of this research is to develop a formal high-level language approach to specification, simulation, performance evaluation, and chip layout planning for VLSI systems. Our approach takes a high-level applicative language (vFP) and programming style as its basis. The rationales for using vFP and and its potential in dealing with several specification and implementation aspects are the subject of this paper. Specifically, a few examples have illustrated how vFP can be used to specify combinational, iterative, and sequential circuits. User-specifiable performance parameters may be used at any abstraction level to provide a basis for making design decisions during the synthesis process. Layouts which are suitable as floor plans are extracted from high-level algorithms. Currently, an automated attribute system is under development. More sophisticated layout techniques and topological optimizations are being examined, as are techniques to handle other classes of sequential circuits.

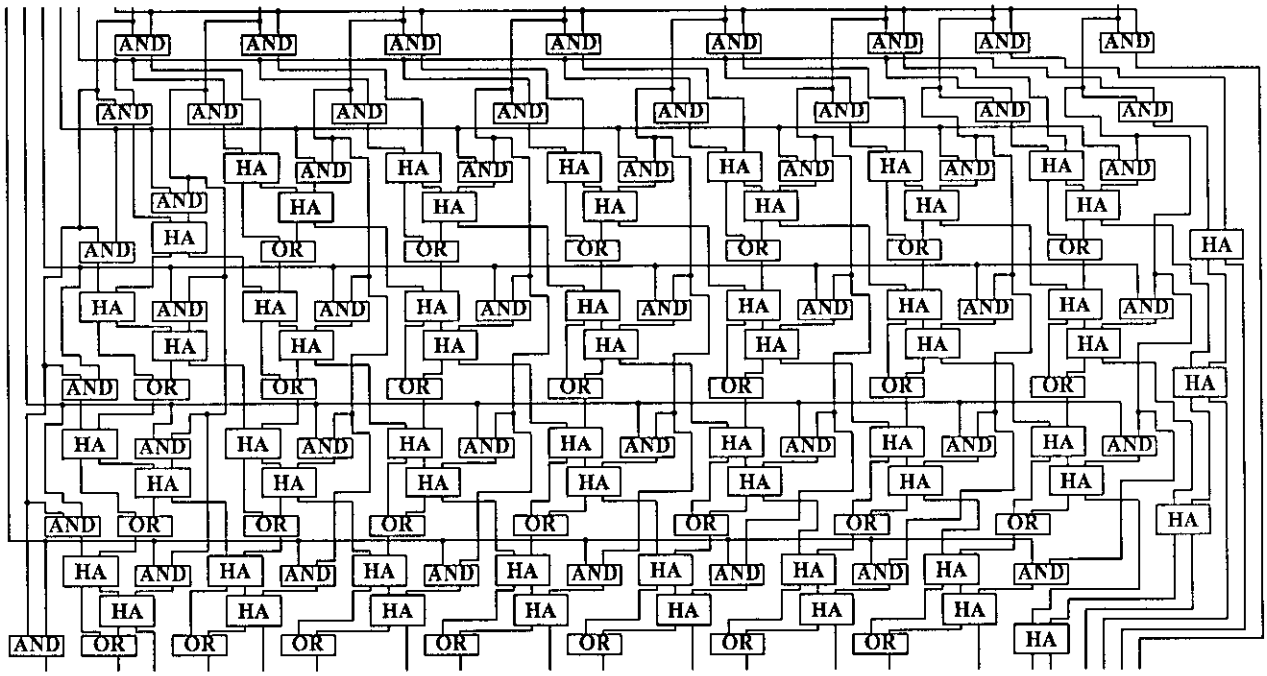

Figure 15: The Carry Save Array Multiplier with **FA***, **HA*** and **FA**** filled in

9 Acknowledgements

This work was supported in part by ONR Contract N00014-83-K-0493, and by Rockwell/UC MICRO Grant 157. The presentation and content of this paper has benefited greatly from the detailed comments provided by one of the referees.

REFERENCES

[Backus78] John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM* **21**(8), pp. 613-641 (August 1978). 1977 ACM Turing award lecture.

[Backus81] John Backus, "The Algebra of Functional Programs: Function level Reasoning, Linear Equations, and Extended Definitions," *Proceedings International Colloqium on Formalization of Programming Concepts Lecture Notes in Computer Science #107*, pp. 1-43, Springer Verlag (1981).

[Brent80] R. P. Brent and H. T. Kung, "The Chip Complexity of Binary Arithmetic," *Proceedings 12th ACM Symposium on the Theory of Computing*, pp. 190-200 (May 1980).

[Cardelli81] Luca Cardelli and Gordon Plotkin, "An Algebraic Approach to VLSI Design," pp. 173-192 in *VLSI 81 - Very Large Scale Integration First International Conference on VLSI*, ed. John P. Gray (1981).

[Director81] S. Director, A. Parker, D. Siewiorek, and D. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Transactions Circuits and Systems* **CAS-28**(7), pp. 634-645 (July 1981).

[Johannsen79] David Johannsen, "Bristle Blocks: A Silicon Compiler," *Proceedings 16th Design Automation Conference*, pp. 310-313 (June 1979).

[Johnson84] Steven Johnson, *Synthesis of Digital Designs from Recursion Equations*, MIT Press (1984).

[Lahti81] D. O. Lahti, "Applications of a Functional Programming Language," Tech. Rep. CSD-810403, UCLA Computer Science Department, Los Angeles, California, (April 1981).

[Meshkinpour85] F. Meshkinpour and M. D. Ercegovac, "A Functional Language for Description and Design of Digital Systems: Sequential Constructs," *Proceedings of the 22nd Design Automation Conference*, pp. 238-244 (June 23-26, 1985).

[Ousterhout81] John Ousterhout, "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design* **II**(4), pp. 34-38 (fourth quarter 1981).

[Ousterhout84] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "Magic: A VLSI Layout System," *Proceedings of the 21st Design Automation Conference*, pp. 152-159 (June 25-27, 1984).

[Rivest82] Ronald L. Rivest, "The 'PI' (Placement and Interconnect) System," *Proceedings of the 19th Design Automation Conference*, pp. 475-481 (June 1982).

[Schlag84] Martine Schlag, "Extracting Geometry from FP for VLSI Layout," Tech. Rep. CSD-840043, UCLA Computer Science Department, Los Angeles, California, (October 1984).

[Sheeran84] Mary Sheeran, "muFP, a language for VLSI design," *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pp. 104-112 (August 6-8, 1984).

[Siskind82] Jeffrey Mark Siskind, Jay Roger Southard, and Kenneth Walter Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," *1982 MIT Conference on Advanced Research in VLSI*, pp. 28-40 (January 1982).