An Algebraic Model for Design Space with Applications to Function Module Generation*

AKHILESH TYAGI

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

ABSTRACT

The design space exploration has been a goal of silicon-compilation for quite a while. But the function module generators (for functions such as adder, shifter and multiplier) do not have a concise model for their design space. This limits their ability to explore the design space. Hence they produce a fixed design which in turns hampers the design space exploration ability of the design synthesis environment. We describe an algebraic model of design space that helps incorporate this flexibility into module generators.

1 Overview

A *function module generator* refers to a layout/netlist module generator for a function such as multiplication as opposed to a module generator for a structure such as PLA or RAM. By the very definition of a structure, the structural design space of a structure is very constrained. There is not much latitude for a structure module generator to explore asymptotic area-time-power resource trade-offs. On the other hand, a function does not specify the underlying topological structure needed to realize it. Hence the design space for a function is extremely rich. For example, a multiplier can, on one extreme, be realized as a Wallace tree schema or it could be implemented as a bit-serial multiplier. However, we don't know of any research that has developed a methodology to build module generators exploring the design-space extensively.

We describe an algebraic approach that characterizes the design space of various functions very succinctly. For instance, a choice of algebraic group elements with a set of operators corresponds to a VLSI design for an adder. What does this gain us? Let a user specify the desired performance characteristics for an adder. The adder module generator has a syntax for an acceptable algebraic expression that corresponds to an adder design. Moreover, the asymptotic area-energy-time performance of this design can be derived from the set of group elements and operators in this expression. (*This gives an a priori measure of performance for every selection of group elements, and hence for the corresponding design.*) The original task is to explore the adder design space to find an adder design that matches the user specifications. An equivalent task is to traverse through a more limited space of the acceptable algebraic expressions. An expression with the performance parameters matching the user's specifications is chosen and mapped into *netlist*. The process of converting this algebraic expression to a netlist uses a simple recursive one-to-one mapping. We use this methodology to build very flexible module generators for adder, shifter and multiplier. However, this approach can also be used for a high level synthesis system's design space exploration in a way similar to Chen's [3].

*This research was supported in part by NSF Grant #MIP-8806169

There is nothing new about the process of module generation. What is novel about our approach is that we provide an efficient *back-end* to a module generator that explores the design space of the given function to make a good choice for the design. *Notice that we are not attempting to develop a language to describe circuits as in $\nu\mathcal{FP}$ [8] and then to compile this language into circuits.* Our objective is more limited and pragmatic. We wish to capture the attributes of the design space in a concise way. These attributes are then used to guide the module generator towards the optimal design space. Johnsson and Cohen [6] do this in a limited way.

In this way, our module generators would replace a family of module generators in a traditional design synthesis system. Due to their space requirements, these families of generators support a very small number of designs. Thus we believe that our paradigm of module generation does a better job of design space exploration than can be done with a small finite family of module generators for a function.

Motivation: Module generation has become an integral component of silicon-compilation [9] and [2]. A typical approach to module-generator design proceeds as follows. Let us assume that a generator for a *shifter* needs to be designed. We would first determine the most commonly used architecture for a shifter. Let us say that we settle on the barrel shifter design shown in Figure 2. Since this design consists of a very regular array of a "*switch*" cell, this will be our leaf cell. The module generator can easily put together an array of these cells for the desired *bus width*. We can either use a procedural system or a graphical system to build this generator. Other architectural options such as a shifter for a dual-bus datapath or electrical optimization options such as sizing of the power bus with the bus width can also be easily supported. Notice however that the area taken by all the shifter designs generated is proportional to n^2, where n is the bus width. Similarly the time taken by this design is proportional to n, assuming an *RC* delay model. The average power consumed by this design is approximately $\frac{n^2}{2}$ times the power consumed by the leaf cell. Hence in an asymptotic sense, we have fixed the area-power-time performance of the shifter designs generated by this module-generator. This in turn restricts the design-space of a silicon compiler incorporating this module generator.

Some systems [4] allow for a limited design space exploration. For example, One may decide that only two designs: *a carry-ripple adder and an adder with a carry-look-ahead of 4 bits*, need to be supported. Then only two sets of leaf cells need be built, one to construct a a carry-ripple adder and the other one for the 4-bit carry-look-ahead adder. But this kind of enumerative approach has a very limited potential. This corresponds to using *table-lookup* as a programming solution to every problem.

2 An Algebraic Approach

Our approach does not attempt to understand and explore the intricate design space trade-offs at the mask geometry level. Instead, we study the structure of communication between n bit slices. This communication has a very rich mathematical (algebraic) structure for three functions we have considered: *addition, shifting* and *multiplication.* The leaf cells are designed for the basic elements of this structure. The larger blocks consisting of these leaf cells are equivalent to applying an operation on the basic elements. The area-power-time performance of a leaf cell (or any basic building block) can be related to the area-power-time performance of the complete design using this characterization. Let us clarify these points using two examples of *addition* and *shifting.*

2.1 Addition

The communication component of addition is not very complex and hence *addition* gives rise to a simple algebraic structure, *monoid**. Not surprisingly, then, the addition has a *space-time dimensionality* of *one* as defined in Chen [3]. The addition of two n-bit numbers, $a_n a_{n-1} \ldots a_1$ and $b_n b_{n-1} \ldots b_1$ can be looked upon as computing the *generate* and *propagate* bits, g_i and p_i, for all the n bit positions. The following relationships between g_i, p_i, a_i, b_i, c_i (carry bit) and s_i (sum bit) are well known (where $\oplus$, $\wedge$, $\vee$ stand for *exclusive-or, Boolean and, Boolean or* respectively). $g_i = a_i \wedge b_i$. $p_i = a_i \oplus b_i$. $c_0 = 0$. $c_i = g_i \vee (p_i \wedge c_{i-1})$. $s_i = p_i \oplus c_{i-1}$. First consider the tuples (g,p) as defined in Brent, Kung [1]. The first entry in the tuple, g, corresponds to the generate bit of a bit position while the second entry corresponds to the propagate bit. Note that in order to add we need to evaluate such tuples for every bit position $1 \leq i \leq n$. When two bit positions are put together, composite generate and propagate signals can be generated. Let us define an operator $\circ$ to model this: $(g,p) \circ (g',p') = (g \vee (p \wedge g'), p \wedge p')$.

Thus $(g,p) \circ (g',p')$ gives the composite generate and propagate signals for a pair of bit positions. But to build an adder, we need the concept of *block-generate* and *block-propagate* signals. The following definition extends the definition of $\circ$ to a block.

$$(G_i, P_i)(j) = \begin{cases} (0,1) & \text{if i=0} \\ (g_j, p_j) & \text{if i=1} \\ (G_{i-1}, P_{i-1})(j+1) \circ (G_1, P_1)(j) & \end{cases}$$

A tuple $(G_i, P_i)(j)$ denotes the block-generate and block-propagate signals of a block of i contiguous bit positions starting with the LSB of the block at the jth bit position. Recall that the formulation of parallel prefix adder in Brent, Kung [1] also defines syntactically similar looking operators. But the semantic difference is significant. Their (G_i, P_i) corresponds to the block carry for the block of the least significant i bits with $c_0 = 0$.

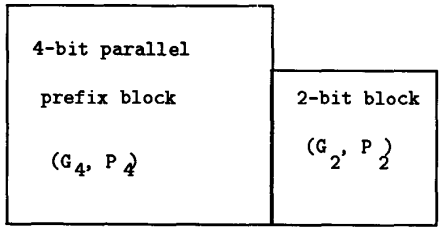

Figure 1: 6-bit Adder Given by $(G_4, P_4)(3) \circ (G_2, P_2)(1)$

Note that the set $\{(G_0, P_0), (G_1, P_1), \ldots, (G_n, P_n)\}$ forms a monoid of *order* n with the operator $\circ$ modified slightly as follows. The identity element for this monoid is $(0,1)$.

$$(G_i, P_i) \circ (G_l, P_l) = \begin{cases} (G_n, P_n) & \text{if } i+l > n \\ (G_i \vee (P_i \wedge G_l), P_i \wedge P_l) & \end{cases}$$

Adder Design Space: The use of an element (G_i, P_i) corresponds to using a carry-look-ahead block with a span of i bits†. One can prove by induction that $(G_i, P_i)(j) = (g_{i+j-1}, p_{i+j-1}) \circ (g_{i+j-2}, p_{i+j-2}), \ldots, (g_j, p_j)$. To realize an adder, we need to compute $(G_n, P_n)(1)$. *The selection of the elements from this monoid to realize (G_n, P_n) corresponds to a design for an adder.* On one extreme one could choose only $(G_n, P_n)(1)$ which gives us the parallel prefix adder of Brent and Kung [1]. The other extreme would be to use n copies of (G_1, P_1) elements (as $(G_n, P_n) = \overbrace{(G_1, P_1) \circ (G_1, P_1) \circ \ldots \circ (G_1, P_1)}^{n \text{ copies}}$). This corresponds with the complete carry-ripple adder. Thus, in general, a collection of elements from this monoid such that $(G_n, P_n) = (G_{i_1}, P_{i_1}) \circ (G_{i_2}, P_{i_2}) \circ \ldots \circ (G_{i_k}, P_{i_k})$ with $\sum_{l=1}^{k} i_l = n$ uniquely identifies a design for an adder. For example, $(G_4, P_4)(3) \circ (G_2, P_2)(1)$ gives a 6-bit adder as shown in Figure 1. In a practical design, one would probably choose all the carry-look-ahead blocks to be the same size, $i_1 = i_2 = \ldots = i_k$.

So far, we can handle adders with n/k carry-look-ahead blocks of lookahead k for $1 \leq k \leq n$ with carry rippling between these blocks. As we mentioned earlier, a carry-look-ahead block with look-ahead of k is just a k-bit parallel-prefix block. Architecturally, all the look-ahead schemes are equivalent to parallel-prefix. An optimization program to increase the fanin from 2 to a larger number will convert a parallel-prefix block netlist to a netlist for any other carry-look-ahead scheme.

How does this description of adders handle *carry-select* blocks? One can encode this information in the type of operators used in an algebraic expression to realize (G_n, P_n). Thus there is another operator $*$ whose semantics is exactly that of the operator $\circ$. But the design corresponding to $(G_i, P_i) * (G_j, P_j)$ will make two copies of the design corresponding to (G_i, P_i). One copy evaluates with $(1,0)$ (carry 1) as the input and the other one evaluates with $(0,0)$ (carry 0). Then a carry-select mux will choose between the output values of these two blocks on the basis of the carry-out value of the (G_j, P_j) block. Now a specification an n-bit adder can consist of expressions containing both $\circ$ and $*$ operators as long as the indices (span of look-ahead) of the monoid elements sum upto n.

Design Space Exploration: Every bit position $1 \leq k \leq n$ should be covered by a $(G_i, P_i)(j)$ such that $j \leq k \leq j+i-1$. There is an additional choice of the operator, $\circ$ or $*$, between two elements $(G_i, P_i)(l+j)$ and $(G_l, P_l)(j)$ (between bit positions $l+j-1$ and $l+j$). The operator $\circ$ just abuts the ciruit segments corresponding to $(G_i, P_i)(l+j)$ and $(G_l, P_l)(j)$. While

*A monoid is just a set closed under an associative operation $\circ$ with an identity element.

†We support the carry-look-ahead of parallel-prefix variety.

the operator $*$ gives rise to additional circuitry for carry-select interface between $(G_i, P_i)(l+j)$ and $(G_l, P_l)(j)$. We maintain an array of n bit positions. This is where we record the element that covers a bit position and the type of operator if that bit position is at the interface of two elements. This provides a rich design space. But many designs in this scheme are clearly suboptimal. For instance, the adder in Figure 1 corresponding to $(G_4, P_4)(3) \circ (G_2, P_2)(1)$ is clearly suboptimal. Thus we explore only the expressions with (G, P) elements with the same look-ahead value (equivalently the same index value). Additionally, *all* the interfaces are either all abut ($\circ$) kind or all carry-select ($*$) kind. Let us note here that we can build parallel-prefix blocks that generate the block carry-out signal for both the cases (block carry-in 0 and 1) at a very small additional cost. It was shown in [1] that the block carry-out for $(G_i, P_i)(1)$ equals G_i when the block carry-in is 0. We can prove that the block carry-out is $G_i \vee P_i$ when the block carry-in is 1. Thus for carry-select operation, rather than duplicating the circuitry for a block, we use the optimized version of the block. Similarly, there is no need to duplicate all the circuitry of a a carry-ripple block to get carry-out for two cases: carry-in being 0 and 1. We can share most of the circuitry and generate the sum and carry bits for the two values of carry-in in a bit-slice at an additional cost of 3 gates [10].

type	*area*	*time*
carry-ripple with look-ahead k	$n \log k$	$\frac{n \log k}{k}$
carry-select with look-ahead k	$\frac{2n}{k} + 1.2n$	$k + \frac{n}{k}$
parallel-prefix with look-ahead k	$\frac{n \log n}{2}$	$\log n$

Table 1: Area-Time Performance of Several Adders

The time taken by an adder specified by the expression $(G_{i_1}, P_{i_1}) \circ (G_{i_2}, P_{i_2}) \circ \ldots \circ (G_{i_k}, P_{i_k})$ is given by $\sum_{l=1}^{k} \log(i_l + 1)$. The area is given by $\sum_{l=1}^{k} i_l \log(i_l + 1)$ and the average case energy consumption is $\sum_{l=1}^{k} i_l$. Let us tabulate the area-time performances of the design options *actually* generated by our system in Table 1. Notice that we don't really explore the whole design space for a given user specification. This table along with the user specifications directs us towards a subspace right away. The choice of the parameter k gives us the flexibility of satisfying the user specifications.

User Specifications: The user specification for time must be in the unit transistor delay units. We chose not to work with absolute time units to keep the technology independence. For the same reason, the area should be specified in terms of the number of transistors. Since we generate the output in MIT netlist format, we generate CMOS transistors and wires. The wire crossings sometimes contribute more to the area of a circuit than the number of transistors. For this reason, *each wire crossing counts as one transistor in our area estimates.*

2.2 Shifter

shifting has a very rich communication between bit-slices. It is a transitive function as observed by Vuillemin [12]. Hence it embeds a computation of a permutation group [‡]. We have looked at several designs for a shifter. Table 2 summarizes the area-energy-time performance of these shifters. We use the familiar cyclic notation (1 3) (2 4) to denote the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}$. The result of applying the permutation (1 3) (2 4) to a 4-bit input $(x_1\, x_2\, x_3\, x_4)$ is $(x_3\, x_4\, x_1\, x_2)$. The (1 3) part of (1 3) (2 4) specifies that the bit in the first position should be routed to the third position and the bit in the third position goes to the first position.

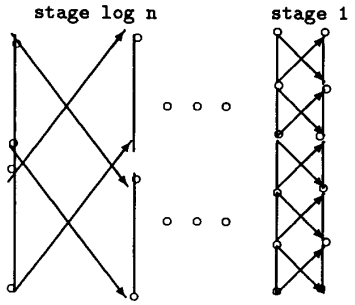

Figure 2: A Barrel Shifter

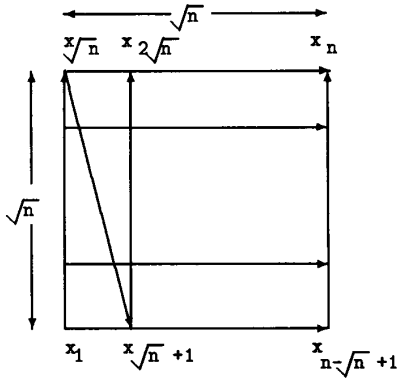

Figure 3: A Square Shifter

type	*energy*	*area*	*time*	*group*
linear, one	n^2	n	n	G_1
barrel, one	n^2	n^2	$\log n$	G_2
square, one	$n^{3/2}$	n	$\sqrt{n}$	G_3
linear, k	$n^2k^2 + \frac{n^2}{k}$	$nk + \frac{n}{k}$	$\frac{n}{k}$	G_4
barrel, k	$nk + \frac{n^2}{k}$	$nk + \frac{n^2}{k}$	$\log n$	G_5
square, k	$\frac{n^{3/2}}{\sqrt{k}}$	n	$k + \sqrt{\frac{n}{k}}$	G_6

Table 2: Area-Energy-Time Performance of Several Barrel Shifters

[‡]A permutation group consists of a set of permutations, Π, that permute a set $\{1\, 2 \ldots n\}$. The set Π is closed under permutation composition. There is an identity permutation and every permutation has an inverse.

Shifter Design Space: Table 2 specifies the design space of shifters. The type can be either a barrel shifter as shown in Figure 2 or the square shifter as described in Ullman [11] (shown in Figure 3). It can also be a linear shift-register. A linear shift-register is a one-dimensional array of shift-store elements. They can either shift their value to their right hand side neighbor or they can just retain it. A barrel shifter works as follows. The control input specifying the amount of shift, $0 \leq c \leq n-1$, can be considered as a $\log n$ bit binary number, $c = c_{\log n} \cdots c_2 c_1$. Each c_i corresponds to a stage in the barrel shifter that can shift by 2^{i-1} bits. Hence there are $\log n$ such stages in a barrel shifter with an interconnection pattern similar to that in a Butterfly network. Since the value of c is $\sum_{i=1}^{\log n} c_i 2^{i-1}$, all the stages combined shift the input by c positions. In the unit delay model, this takes time proportional to $\log n$ with an area requirement of n^2. A square shifter saves area by giving up speed. It is designed as a $\sqrt{n} \times \sqrt{n}$ array. The input bits $x_1 \ldots x_n$ are stored in this array as follows. Let the *lower-left* corner be the array position $(1,1)$ and the *upper-right* corner be $(\sqrt{n}, \sqrt{n})$. Then the array position (i, j) stores the input bit $x_{i+(j-1)\sqrt{n}}$. The cell in this array is capable of shifting either up or to the right. Notice that the top cell in each column shifts to the bottom cell of the next column during an upshift. The shift value $c = c_{\log n} \cdots c_1$ can be split into two values: $c_{\text{up}} = c_{\log n} \cdots c_{\frac{\log n}{2}+1}$ and $c_{\text{right}} = c_{\frac{\log n}{2}} \cdots c_1$. A shift by c consists of shifting all the values right by c_{right} in time $\sqrt{n}$ followed by shifting up by c_{up} in time $\sqrt{n}$. Thus the complete shift takes time $\sqrt{n}$ with area n.

The second argument in type indicates the number of times an input bit is available. In the most likely situation, this argument is 1.

Some observations regarding this group decomposition schema are in order. The group computed by a cyclic shifter must contain the permutations corresponding to all the values for the shift. This consists of permutations $(1\,2 \ldots n)$, $(1\,3\,5 \ldots)$, $(1\,4\,8 \ldots) \cdots (1\,n \ldots)$. The first permutation corresponds to a left shift by one, the second one to a left shift by two and the last one to a left shift by $n-1$. *We don't need physical circuitry corresponding to each of these permutations.* Only a few of these permutations generate the whole shifting group. We call such a set of permutations, Π, *a set of generators* for the shifting group, *i.e.*, any permutation in the shifting group is a composition of permutations from Π. We wish to analyze the mimimal sets of generators. Then a minimal amount of hardware is needed to implement a minimal set of generators to provide a shifter.

LINEAR SHIFTER: A linear shifter is generated by $\{(1\,2 \ldots n)\}$. The permutation $(1\,2 \ldots n)$ shifts every bit by one. The repeated shifting provides a shift by any amount. An implementation of $(1\,2 \ldots n)$ gives a linear shifter. This is the group G_1 in Table 2.

BARREL SHIFTER: Note that the realization of the permutation $(1\,2 \ldots n)$ provides the complete shifting group. A barrel shifter realizes $(1\,2 \ldots n)$ in $\log n - 1$ compositions of $\log n$ permutations of the following type. A permutation $(1\ \frac{n}{2^i}+1)(2\ \frac{n}{2^i}+2) \cdots (\frac{n}{2^i}\ \frac{2n}{2^i})(\frac{2n}{2^i}+1\ \frac{3n}{2^i}+1) \cdots (\frac{3n}{2^i}\ \frac{4n}{2^i}) \cdots (n-\frac{2n}{2^i}+1\ n-\frac{n}{2^i}+1) \cdots (n-\frac{n}{2^i}\ n)$ corresponds to the $(\log n - i + 1)$th stage of a barrel shifter. For example, $(1\ \frac{n}{2}+1)(2\ \frac{n}{2}+2) \cdots (\frac{n}{2}\ n)$ shifts every bit by $n/2$ positions. Thus it corresponds to $\log n$th stage of Figure 2. The dimensionality of data flow is still one. A cell for a barrel shifter can be derived from the one for a linear shifter. A linear shifter has only one outgoing path from every cell. A barrel shifter, on the other hand needs two paths. The cell in stage i for bit j participates in the cycle $(j\ j+2^{i-1})$. Hence it needs paths to bits j and $j+2^{i-1}$ in stage $i-1$. Group G_2 in Table 2

SQUARE SHIFTER: It realizes the permutation $(1\,2 \ldots n)$ in an interesting way. It breaks up the cycle $(1\,2 \ldots n)$ into $\sqrt{n}$ cycles $(1\,2 \ldots \sqrt{n})\ (\sqrt{n}+1 \ldots 2\sqrt{n}) \cdots (n-\sqrt{n}+1 \ldots n)$ corresponding to the $\sqrt{n}$ columns of Figure 3. To jump between these cycles, another group of $\sqrt{n}$ cycles is created. To connect the first elements of the previous cycles, we need $(1\ \sqrt{n}+1 \ldots n-\sqrt{n}+1)$. Similarly to connect the ith elements $(i\ \sqrt{n}+i \ldots n-\sqrt{n}+i)$ is needed. This gives rise to the permutation relating to the rows of a square shifter. This set of generators is called G_3 in Table 2. Notice that a square shifter leaf cell has 2 outgoing paths as well. Hence two copies of a leaf cell for a linear shifter can be combined to give a leaf cell for a square shifter. Also note that each generator can be decomposed into m cycles which corresponds to m slices of data flow. Thus in the *square, one* design there are $\sqrt{n}$ rows (columns) corresponding to the first (second) generator's cyclic representation. This gives us a control over the *aspect ratio* of the design. To achieve an aspect ratio of a, one needs to decompose the horizontal group generator into b cycles and the vertical one into c cycles such that $n = b+c$ and $a = b/c$.

The groups G_4, G_5 and G_6 are the groups G_1, G_2 and G_3 respectively when each input bit is repeated k times.

Design Space Exploration: The task of creating a shifter design is equivalent to choosing a set of generators as described in the preceding discussion. One can automatically verify if a given collection of permutations generates the shifting group. The area and time performance of the corresponding design can be deduced in the following way. Count the number of cycles a given position participates in. This number gives a worst case time bound on shifting that input bit position by any value. Similarly, the number of permutations in the set of generators is an indicator of the area requirements. But one need not attempt to walk through the space of all the sets of generators blindly. We consider only the following design space in our generators.

Given the user specifications in terms of gate delay units and number of gates, the *shifter generator* determines whether Group G_4, G_5 or Group G_6 is required. Then the parameter k is chosen to give a tight fit with the user specifications. The leaf cells for G_1, G_2 and G_3 based shifters have a close relationship as observed earlier. We use this fact to keep only one leaf cell: a shift-register cell for G_1. The other leaf cells are built from this cell very efficiently. The generation of the netlist for a given group and k is described later.

3 Implementation

The adder and shifter generators have been implemented in 'C' programming language. A module generator for multiplier is being developed. The basic methodology is as follows. Note that our formalism attempts to capture the nature of the communication in a function. Hence there are parts of the circuit that essensially remain invariant within the design-space. This invariant part is our primitive leaf cell. A primitive leaf cell is identified: a full-adder cell for adder, a shift-register cell for shifter. This cell

is built in the VPNR *netlist* format [7]. The generator program reads this cell and builds a corresponding circuit data-structure. After the user specifications are read, the system explores the design space as described. A group or an algebraic expression is chosen and the corresponding circuit is built. The next phase transforms the circuit data-structure into a netlist file.

The generator programs are modularized around the group/monoid element units. For example, the adder generator has a procedure to build a parallel-prefix carry block of k bits given a parallel-prefix carry block of $k/2$ bits. Similarly, the operators have corresponding procedures to achieve the desired affect. For instance, the adder generator has a procedure corresponding to the $*$ operator that takes two arguments (G_i, P_i) and (G_j, P_j). It builds a new block that selects the carry-out and sum bits of (G_i, P_i) block on the basis of the carry-out of (G_j, P_j) block.

4 Conclusions and Future Research

The need for the design space exploration at the architecture level is succintly brought out by Johannsen [5]. CATHEDRAL [9] also uses the knowledge about its domain (DSP) to make a good choice for the design. However, providing this capability to module-generators as well enhances the quality of designs produced by a silicon compiler. We proposed an algebraic design space model that facilitates easy design space exploration at function level. The models for the design space of adder and shifter were built from our knowledge about these functions. If a function h is derived as a composition of two functions f and g, where the models of f and g are already known, can we derive the model for h from the models of f and g in an automatic way? In practice, two modules rarely have a clean mathematical composition. But the designers tend to use very few ways of putting together two modules, such as: a bus, a latch. We are working on a chracterization of these hardware composition schemes in terms of our algebraic models. A global design can be seen as a function composition of constituent functions. For instance, an ALU in a microprocessor is a complex composition of shifter, ALU, memory and some other functions. This extends the applicability of our work to very large systems.

References

[1] R.P. Brent and H.T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, 260–264, March 1982.

[2] M. R. Burich. Design of Module Generators and Silicon Compilers. In D.D. Gajski, editor, *Silicon Compilation*, chapter 2, pages 49–94, Addision-Wesley Publishing Company, Reading, Mass., 1988.

[3] M. C. Chen. The Generation of a Class of Multipliers : A Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI. In *Proceedings of the International Conference on Computer Design: VLSI in Computer*, IEEE, 1985.

[4] K. Chu and R. Sharma. A Technology Independent MOS Multiplier Generator. In *Proceedings of the 21st Design Automation Conference*, IEEE-ACM, 1984.

[5] D. Johanssen. Silicon Compilation. In *Proceedings of the 1989 Decennial Caltech Conference on VLSI*, pages 17–56, MIT Press, 1989.

[6] L. Johnsson and D. Cohen. A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks. In *Proceedings of the CMU Conference on VLSI*, pages 213–225, CMU, Computer Science Press, 1981.

[7] G. Kedem and F. Brglez. *OASIS: Open Architecture Silicon Implementation System.* Technical Report MCNC TR 88-06, Microelectronics Center of North Carolina, February 1988.

[8] D. Patel, M. Schlag, and M. Ercegovac. $\nu\mathcal{FP}$: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms. In *Proceedings of the Functional Programming Language and Computer Architecture Conference*, pages 233–255, 1985.

[9] P. Six, L. Claesen, J. Rabaey, and H. De Man. An Intelligent Module Generator Environment. In *Proceedings of the 23rd Design Automation Conference*, IEEE-ACM, 1986.

[10] A. Tyagi. A Reduced Area Scheme for Carry Select Adders. 1989. submitted for publication.

[11] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Md., 1984.

[12] J. Vuillemin. A Combinatorial Limit to the Computing Power of VLSI Circuits. *IEEE Transactions on Computers*, 294–300, March 1983.