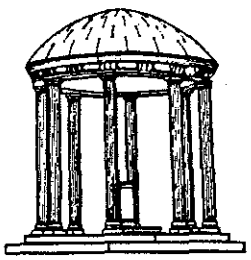
An Algebraic Model for Design Space Exploration*

TR90-009

February, 1990

Akhilesh Tyagi



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

An Algebraic Model for Design Space Exploration*

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

ABSTRACT

We believe that it is *the degrees of freedom in the physical realization of the communication component of a function* that gives the design space its complexity and diversity. The number of communication schemes exhibited by the datapath functions is very small: *three* to be precise. We give a general framework to model these communication schemes by simple algebraic structures which also provide the corresponding design spaces. These models are extremely simple, and hence easy to manipulate, but are sufficiently powerful to model the simple concepts of pipelining and synchronous designs.

1 Overview

The following quotation from McFarland, Parker and Camposano [MPC88] sums up the problem in a nutshell.

> *"The major problem underlying all these tasks is the extremely large number of design possibilities which must be examined in order to select the design which meets the constraints and is as near as possible to the optimal design. The "design space" that needs to be searched is multi-dimensional and discontinuous, and it is hard even to find a canonical set of operators that systematically take you through that space. Furthermore, the shape of the design space is often problem-specific, so that there is no methodology that is guaranteed to work in all cases."*

1.1 The Global Picture

The need for design space exploration is succinctly brought forth by Johansson [Joh89]. We see the design space exploration as a two step process. The first phase picks up an architecture† best suited to the specifications of the user. The second phase can then apply electrical optimizations and local transformations to achieve a better match with the user's requirements. At the function module level, consider the example of an adder. If a fast $O(\log n)$ time addition is desired, the architecture level exploration phase must identify the parallel-prefix adder as the architecture. The transistors along the parallel prefix tree chains can be sized up to match the exact delay specifications.

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

†We use the term *architecture* loosely here to mean an algorithm with a hardware implementation

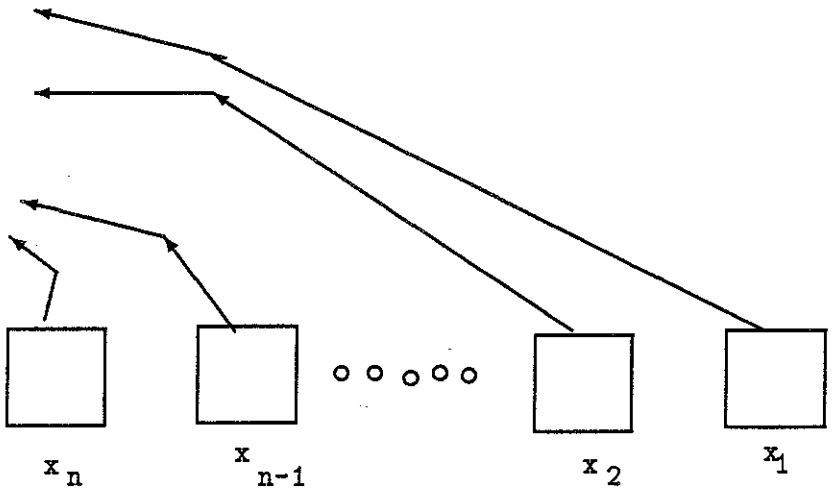


Figure 1: An Illustration of One-Dimensional Dataflow

Our work addresses the first phase in this scheme: architecture selection. There are at least two ways it could be done. One approach is to develop a language to describe Boolean networks / circuits. An architecture is described as a program in this language. Several design transformations attempt to generate many equivalent architectures from a given architecture for a function. Most often these transformations are very local in the sense that they massage a small part of the design rather than changing the underlying algorithm. A global transformation would be analogous to the program transformations in the software area. The need to make this language sufficiently expressive to tackle a wide variety of circuits makes it hard to perform these program transformations. The principal difficulty seems to be that the algorithmic description of hardware contains a wealth of information about the implementation such as clocking, timing and structural attributes. A transformation has to deal with all this information in the target design.

Let us now outline the second approach. An implementation of a function/algorithm can be considered to be performing two tasks: communication and computation. It is often the communication component that dominates both the asymptotic area and time requirements. Our approach takes this into account. We develop algebraic models for the communication pattern of a function. Once again, consider the adder example. The circuits to compute the individual *carry*, *kill* and *propagate* bits do not determine its complexity. It is the physical realization of the underlying communication pattern (the bit c_0 to $1-n$th bit positions, c_1 to $2-n$th positions and c_i to $i+1-n$th positions) that defines its design space. This realization could either be a carry-ripple chain or a look-ahead scheme or a carry-select scheme among many possible ways. An algebraic structure, *a monoid*, is a good model for this kind of one dimensional communication pattern. This is also a model for the design space. An essential difference between the two approaches is that the communication model does not contain sufficient information to reconstruct the underlying algorithm. But it is simple enough that one can traverse the design space in a relatively efficient way. We believe that the design space

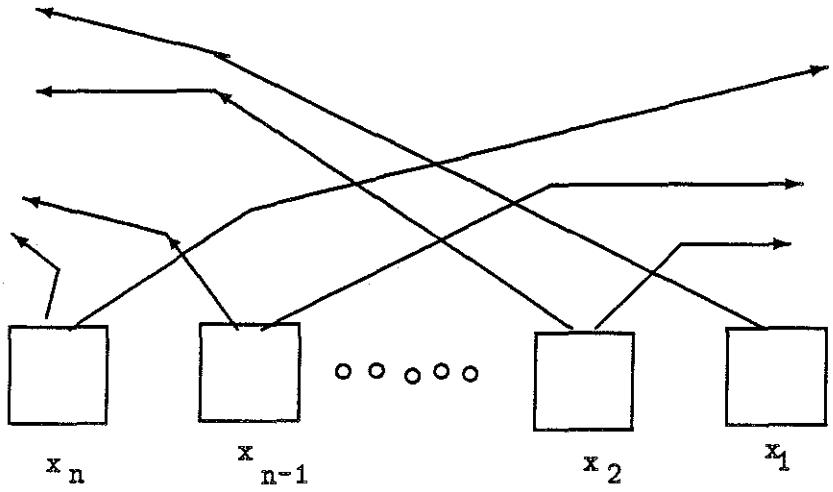


Figure 2: An Illustration of Two-Dimensional Dataflow

model should be added to the list of attributes of a design such as timing model and simulation model that are propagated across several design abstractions and across the design hierarchy.

This discussion still does not address the issue of suitability of this approach for silicon compilation. The communication pattern's regularity results from the regularity of the function in the following sense. Every bit position in an adder is executing the same algorithm. There are identifiable groups of variables/nodes such as *carry* that have the same semantics. This gives rise to a regular communication pattern between these bit slices. It follows from this discussion then that all the datapath functions can be modeled in this way. Admittedly, a silicon system consists of random logic components as well. At this point, we don't have a very clear understanding of how this approach can be applied to a random logic block. But can we extend this approach from a function module level model to a model for a datapath? We can build the design space models for a few primitive functions such as adder and shifter from our understanding of these functions. The next step involves being able to compose two models to form a composite model for the composition of two primitive functions. An example is a multiplier. A multiplication is a complex composition of addition and shifting. We show how we can derive a design space model for a multiplier from the models of an adder and a shifter. The situation is somewhat different when two functions communicate under the control of a finite state machine, as the communication between a program counter (PC) and a memory address register (MAR). This composition requires a different technique to derive the composite model.

1.2 Related Work

The systolic design community has done extensive work in transforming one algorithm into many systolic realizations [Che85], [Che87], [CS83], [Mol83], [LM82], [Qui84]. Parker, Park and Jain

have developed several empirical models for pipelined and nonpipelined area-time trade-offs [PP86], [JMP88]. The work that has come closest to performing design space exploration was done by Johnsson and Cohen [JC81]. But its application was limited to simple computational networks. Sheeran [She84] and Patel et. al. [PSE85] have used variants of FP to describe and generate circuits. Another formal means of realizing circuits was proposed by Johnson [Joh86]. But these formal methods are either too specialized to describe general circuits or they cannot consider the design space in its full generality.

1.3 Organization

Section 2 describes the notion of type-0, type-1 and type-2 functions and then develops the design space models for them. These functions encompass addition and shifting. Section 3 deals with the question of design space composition.

2 Design Space:

The term *design space* refers to the space of various physical incarnations of a function. For instance, addition can either be performed with a $O(n)$ area and $O(n)$ time carry-ripple adder or with a $O(n \log n)$ area and $O(\log n)$ time parallel-prefix adder [BK82]. More often than not, the design space does not consist of just a collection of discrete design points. For an adder, the family of k-bit carry look-ahead adders consisting of n/k look-ahead blocks is parametrized by k in the range 1 through n. We are interested in characterizing the design space of a function by the area-time requirements of its design points.

The notion of the area used by a design point and the time taken by a design point is well-defined. A word of clarification is in order here. We intend to characterize a design point according to its asymptotic area-time requirements. Thus a carry-ripple adder for an n-bit datapath is classified as n area and n time adder. In reality, every asymptotic design point is a *bubble* in the area-time space consisting of all the adders that can be derived from each other through local optimizations. The local optimizations will typically work within the context of a gate or a critical path such as transistor sizing, fanin reordering and critical load isolation. A catalog of such techniques used in the CAD tool POLO is described in Kotliar and Hedlund [KH89]. Figure 3 shows the bubbles corresponding to a carry-ripple adder and a parallel-prefix adder. This figure highlights several points.

1. The design space of a function is a family of such area-time graphs (one graph for every value of n – the datapath width) rather than a single graph.

2. A bubble is characterized by its asymptotic area-time requirements as a function of n. Thus a carry-ripple adder has area $\Theta(n)$ and time $\Theta(n)$. But the asymptotic notation Θ can hide many constants. Thus, for this approach to be feasible, the constants need to be determined. The constants depend on the idiosyncrasies of the designer's design style or on the characteristics

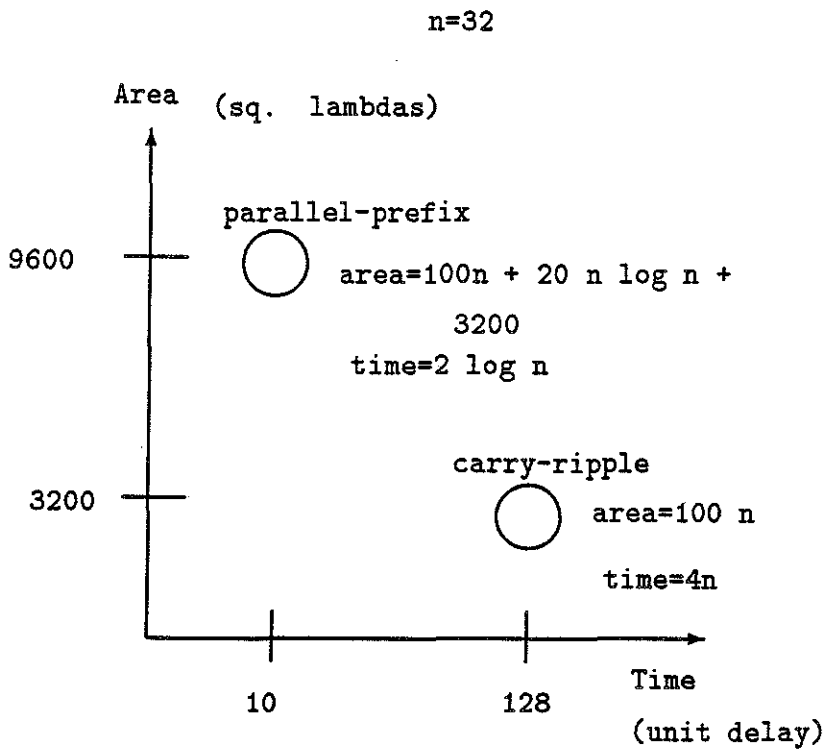


Figure 3: Two Bubbles in the Design Space for Adder

of the layout system. We generate the mask layouts using the placement and route system VPNR [KB88]. A set of constants can be derived for any layout system. Figure 3 uses a hypothetical example of a system where each cell of a carry-ripple adder takes up 100 λ^2 area and has delay of 4 units. Similarly, a parallel-prefix adder takes area $100n + 20n \log n + 3200$ lambda squares. Each level of the parallel-prefix network has depth of 2.

The most commonly used approach to design space exploration so far has been to get into one such bubble for a design on the basis of some optimization criterion such as area or time. Then local optimizations are applied to squeeze another 20% to 30% of the desired resource – which corresponds to a traversal within such a bubble. Our objective is to provide a capability to jump between these bubbles in accordance with the asymptotic resource requirements. The design space exploration at the *bubble* level is often more rewarding than the exploration within a bubble. The second phase of local traversal within a bubble for further tuning of parameters is still applicable and desirable. An approximate analogy is to an optimization problem with many local minima. The first approach puts us into some local valley, while our proposed approach enables us to sample all the valleys.

In the light of the preceding discussion, we choose to characterize a design point by the asymptotic resource usage of the bubble: we don't distinguish between the design points within a bubble. This develops the notion of the design space for a function.

3 Design Space Models

As we discussed earlier, our design space models encompass only the communication component of a design. The computation part can be bound to any circuit that exhibits the desired behavior. We classify the primitive communication patterns into two classes based on the *dimensionality* of the dataflow. The concept of dimensionality of dataflow was also used by Chen [Che85], [Che87]. In an intuitive way, if the value of a group of variables flows from the less significant bit positions to more significant bit positions only then the dataflow dimensionality is one. Figure 1 illustrates this. The examples of some functions with dataflow dimensionality 1 are addition, parity generation and counting. On the other hand, if the information flow can also be from more significant bit positions to less significant bit positions then the dimensionality is 2. *Shifting* is an example of a function with dataflow dimensionality 2. This is shown in Figure 2. Of course, in order to be complete, we should also consider the functions with dataflow dimensionality 0. These are the functions where there is no communication between any of the n bit slices. We also call these functions **type-0** functions. All the memory elements in a datapath, such as a register file, memory data register are examples of type-0 functions. For these functions, the communication *does not dominate* the design space.

We develop more formal definitions for the other two types. Let us first recall the definitions of two algebraic structures: a *monoid* and a *group*. A monoid is a set closed under an associative operation $\circ$ with an identity element. A set with an associative operation $\circ$ is a group if it is closed, has an identity element and has an inverse for every element. A *permutation group* is a group of permutations. A permutation $\pi(x_1, x_2, \cdots, x_n)$ permutes its input to give $(x_{i_1}, x_{i_2}, \cdots, x_{i_n})$. The

composition of two permutations is defined very naturally where $(\pi_i \circ \pi_j)(x_1, x_2, \cdots, x_n)$ gives π_i applied to $\pi_j(x_1, x_2, \cdots, x_n)$. Now let us define the concept of a function with dataflow dimensionality one. We refer to these functions as *type-1* functions.

3.1 Type-1 Functions

Definition 1 *Let a function $f(x_1, x_2, \cdots, x_n)$ have an output $(y_1, y_2 \cdots, y_n)$ where x_i and y_j $1 \leq i, j \leq n$ are bits. f computes a monoid if there exists a set of bits $(m_1, m_2, \cdots, m_n)$ computed by f and an operation $\circ$ such that the set $\{m_1, m_2, \ldots, m_n\}$ alongwith $\circ$ forms a monoid. In this case, the dataflow for f has a dimensionality of at least one. We refer to f as a* type-1 function.

An alternative definition considers the communication complexity [Yao79] of the functions. The functions with $O(1)$ communication complexity are type-1 functions. Let us consider some examples of functions that compute a monoid. Addition is one of them. Let us consider the addition of two words $a_n \ldots a_2 a_1$ and $b_n \ldots b_2 b_1$.

3.1.1 Addition

Let g_i and p_i be the generate and propagate bits for the ith bit position. The following relations are well known.

$$g_i = a_i \wedge b_i \tag{1}$$

$$p_i = a_i \oplus b_i \tag{2}$$

Brent and Kung [BK82] show the following. Let (g, p) be a tuple associated with every bit slice. When two bit positions are put together, composite generate and propagate signals can be generated. The operator $\circ$ models this as follows.

$$(g, p) \circ (g', p') = (g \vee (p \wedge g'),\ p \wedge p')$$

We define the concept of *block-generate* and *block-propagate* signals for the blocks spanning the bit positions 1 through i.

$$(G_i, P_i) = \begin{cases} (g_1, p_1) & \text{if i=1} \\ (g_i, p_i) \circ (G_{i-1}, P_{i-1}) & 2 \leq i \leq n \end{cases}$$

(G_i, P_i) represent the final generate and propagate values at the bit position i. Brent and Kung [BK82] go on to show that $c_i = G_i$. However, this perspective does not help us construct a monoid. We need to represent the communication structures that can bridge the carry by i positions for $1 \leq i \leq n$. Hence we modify the definition of (G_i, P_i) given by Brent and Kung in the following way.

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

Now $(G_i, P_i)(j)$ represents the block-generate and block-propagate signals of a block of length i starting at bit position j (bit positions j through $i+j-1$). Now with the following definition of $\circ$, the set $\{(G_0, P_0), (G_1, P_1), (G_2, P_2), \ldots, (G_n, P_n)\}$ forms a monoid.

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

The identity element for this monoid is $(G_0, P_0) = (0,1)$. This shows that the dimension of the dataflow for addition is at least one.

3.1.2 Parity and Counting

Now let us consider parity generation. The parity of n bits $x_1, x_2, \ldots x_n$ is 1 if an odd number of input bits have value 1. Otherwise the parity is 0. Note that at least one bit of information needs to flow across any partition of the input bits. Let p_i be the parity of the input bits $x_1, x_2, \ldots, x_i$. Once again we can define a block parity signal, P_i, that indicates the parity of a given block of span i.

$$P_i(j) = \begin{cases} 0 & \text{if i=0} \\ x_j & \text{if i=1} \\ P_{i-1}(j+1) \circ P_1(j) & \text{otherwise} \end{cases}$$

The operation $\circ$ corresponds to exclusive-or $\oplus$. Then the set $\{P_0, P_1, \ldots, P_n\}$ alongwith the operation $\circ$ defined as follows forms a monoid.

$$P_i \circ P_l \;=\; \begin{cases} P_n & \text{if } i+l > n \\ P_i \oplus P_l & \text{otherwise} \end{cases}$$

The monoid identity is $P_0 = 0$. This establishes parity as a type-1 function. *Counting* can be shown to be computing a monoid in a similar way as addition.

3.2 Type-2 Functions

The type-2 functions compute a permutation group. Notice that the computation of a permutation group requires more communication than the computation of a monoid. An information-theoretic definition will consider the functions with $O(n)$ communication complexity to be type-2 functions. A formal definition of a type-2 function follows.

Definition 2 *Let a function* $f(x_1, x_2, \cdots, x_n \ : \ c_1, c_2, \ldots, c_{\log n})$ *have an output* $(y_1, y_2 \cdots, y_n)$ *where* $x_1, x_2, \cdots, x_n$ $(y_1, y_2 \cdots, y_n)$ *are input (output) bits and* $c_1, c_2, \ldots, c_{\log n}$ *are the control input bits.* f *computes a permutation group if every control input value permutes the input bits and all the permutations encoded by the control values form a permutation group. In this case, the dataflow for* f *has a dimensionality of at least two. We refer to* f *as a* type-2 function.

The clearest example of a type-2 function is shifting.

3.2.1 Shifting

We consider right cyclic shift as an example of type-2 function. Let π_i for $0 \leq i \leq n$ represent the right cyclic shift by i bit positions. In particular, π_i corresponds to the permutation $\begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ x_{(1-i) \bmod n} & x_{(2-i) \bmod n} & \cdots & x_{n-i} \end{pmatrix}$. It is an accepted practice to use only the bit positions in this notation. Then π_i is given by $\begin{pmatrix} 1 & 2 & \ldots & n \\ (1-i) \bmod n & (2-i) \bmod n & \ldots & n-i \end{pmatrix}$. Consider the set of permutations $G = \{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$. The composition operator is defined as $\pi_i \circ \pi_l$ is $\pi_{(i+l) \bmod n}$. Notice that G forms a group. The identity element is π_0 and the inverse of π_i is $\pi_{(n-i) \bmod n}$. This demonstrates that shifting has a dataflow of dimensionality two. In addition, it is a type-2 function.

3.2.2 Transitive Functions

The notion of transitive functions was defined by Vuillemin [Vui83]. These functions embed a computation of a permutation group. The examples include shifting, multiplication, linear transforms and three matrix product. Note that not every transitive function is a type-2 function due to the requirement in Definition 2 that *all the control values encode a permutation of a group.* But all the type-2 functions are transitive. The transitive functions are compositions of type-1 and type-2 functions. We will look at multiplication in the next section.

3.3 Design Space of Type-1 Functions

The definition of a type-1 function tells us that it computes a monoid $(M, \circ)$. Let M be the set $\{M_0, M_1, M_2, \ldots, M_n\}$. The reader is encouraged to think of the adder monoid described in Subsection 3.1 as a more concrete example of the following concepts. Recall from Subsection3.1 that the monoid element $M_i(j)$ denotes a block computation of the monoid element with the block span of i bits and the jth bit being the least significant bit of the block. There are many physical realizations for a communication scheme M_i. But given the limited fanin, all of them require $\log i$ levels and $i \log i$ gates. A parallel prefix scheme as described in Brent, Kung [BK82] or Ngai, Irwin [NI85] can be adapted to realize M_i for any monoid M. The underlying communication scheme remains the same for any monoid. Only the cells computing the composition differ. To realize the function f, we need to compute $M_n(1)$. *The selection of the elements from this monoid to realize*

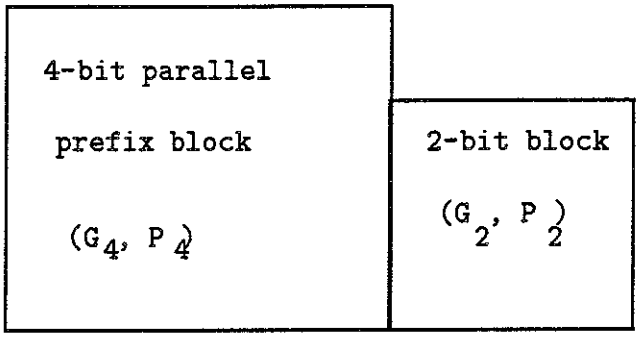


Figure 4: 6-bit Adder Given by $(G_4, P_4)(3) \circ (G_2, P_2)(1)$

M_n corresponds to a design for the communication component of f. On one extreme one could choose only $M_n(1)$ which gives us the parallel prefix realization. This design takes up the most area, $O(n \log n)$, but is the fastest with $O(\log n)$ delay. The other extreme would be to use n copies of M_1 elements (as $M_n = \overbrace{M_1 \circ M_1 \circ \ldots \circ M_1}^{n \text{ copies}}$). This corresponds with the complete ripple communication. This design is the slowest data parallel design (delay $O(n)$) but takes only $O(n)$ area. Thus, in general, a collection of elements from this monoid such that $M_n = M_{i_1} \circ M_{i_2} \circ \ldots \circ M_{i_k}$ with $\sum_{l=1}^{k} i_l = n$ uniquely identifies a design for the communication component of f. Taking an adder example, $(G_4, P_4)(3) \circ (G_2, P_2)(1)$ gives a 6-bit adder as shown in Figure 4. In a practical design, one would probably choose all the carry-look-ahead blocks to be the same size, $i_1 = i_2 = \ldots = i_k$.

We can get fancier in the realizations of a monoid communication to achieve the design points between the ripple scheme and the parallel prefix scheme. We can have selection communication analogous to *carry-select* blocks. This information can be encoded in the type of operators used in an algebraic expression to realize P_n. In addition to $\circ$, we introduce another operator $*$ whose semantics is exactly that of the operator $\circ$. But the design corresponding to $M_i * M_j$ will make two copies of the design corresponding to M_i. One copy evaluates with 1 (monoid input 1) as the input and the other one evaluates with 0 (monoid input 0). Then a selection mux will choose between the output values of these two blocks on the basis of the monoid output value of the M_j block. Now a specification of an n-bit function f 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. Every bit position $1 \leq k \leq n$ should be covered by a $M_i(j)$ such that $j \leq k \leq j+i-1$. There is an additional choice of the operator, $\circ$ or $*$, between two elements $M_i(l+j)$ and $M_l(j)$ (between bit positions $l+j-1$ and $l+j$). This provides a rich design space. But many designs in this scheme are clearly suboptimal. For example, only the expressions with M elements with the same span need be explored.

Space-Time Mapping and Combinational Vs. Synchronous Designs: Consider the spec-

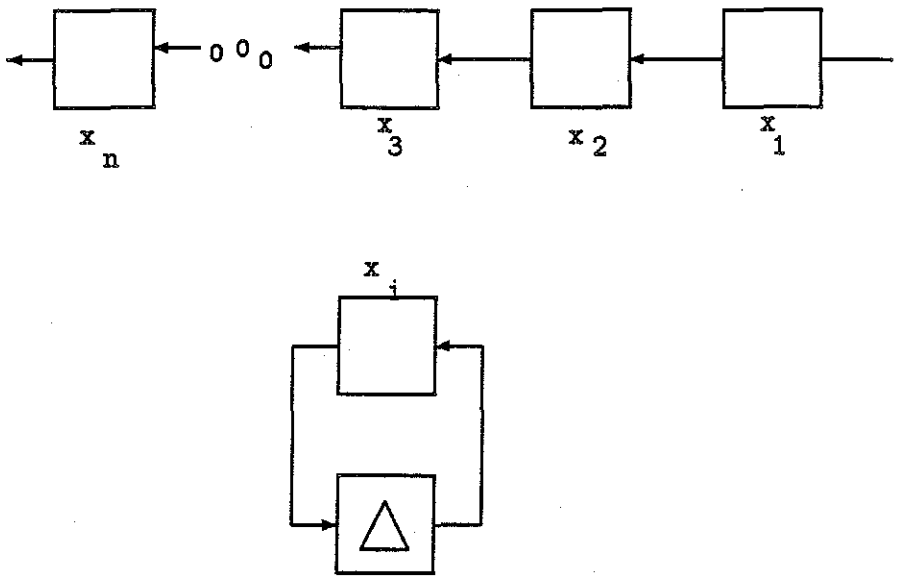


Figure 5: Space-Time Mapping

ification for a ripple-communication scheme: $\overbrace{M_1 \circ M_1 \circ \ldots \circ M_1}^{n \text{ copies}}$. This specification is doing the requisite monoid computation in space (all the compositions are done in space). One easily notices that the n instances of M_1 can be folded upon one instance of M_1, if the composition $\circ$ were to be done in time rather than space. The time realization of a composition introduces a delay element along every signal path going across the space realization. Figure 5 shows this. Note that the input bit stream x will be serialized in the time composition schema.

Since we leave the internals of a monoid element M_i unspecified, we assume a combinational implementation, to be on the safer side. The following scheme can sometimes be used to derive a synchronous design. The ripple-communication specification $\overbrace{M_1 \circ M_1 \circ \ldots \circ M_1}^{n \text{ copies}}$ gives a combinational design. To make it synchronous, a latch (delay element) is associated with the composition operator. We use the notation $\circ_\Delta$ for a synchronous composition operator. The synchronous specification $\overbrace{M_1 \circ_\Delta M_1 \circ_\Delta \ldots \circ_\Delta M_1}^{n \text{ copies}}$ gives a ripple realization as in Figure 5 except that every wire is cut with a delay latch. For a monoid computation, the storage requirement is only a constant, as can be shown using Baudet's ideas [Bau81]. Hence this schema can also be time mapped to derive a bit-serial design.

The time taken by a design specified by the expression $M_{i_1} \circ M_{i_2} \circ \ldots \circ M_{i_k}$ 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 $\sum_{l=1}^{k} i_l$. We use this formulation to build a module generator for an adder [Tya90]. Let us tabulate the area-time performances of several design options *actually* generated by our system in Table 1. This table

type	area	time
synchronous time-mapped ripple	$O(1)$	$O(n)$
ripple with look-ahead k	$n \log k$	$\frac{n \log k}{k}$
selection 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 Monoid Designs

along with the user specifications directs us towards a design subspace right away. The choice of the parameter k gives us the flexibility of satisfying the user specifications.

3.4 Design Space of Type-2 Functions

We need to realize a permutation group to build the communication structure of a type-2 function. The design space for a permutation group, G, depends on many parameters for the group such as the number of equivalence classes and the order of a generator element. We enhance on this aspect in the following. Each permutation acts on the set $S = \{1, 2, \ldots, n\}$, the set of positions. A permutation group defines an equivalence relation on this set as follows [[Rob84], pages 290-291]. Two elements $a, b \in S$ are related by this equivalence relation r if there exists a permutation π s.t. $\pi(a) = b$. Burnside's Lemma [Rob84] gives a way of counting the number of equivalence classes introduced by this relation. We can use the familiar cyclic notation (1 3) (2 4) to denote the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}$. Then the number of equivalence classes induced by a group is at most as high as the number of cycles in any permutation of G. *The number of equivalence classes, $c(G)$ is a good measure of the routing complexity of a physical realization of the permutation group G.* Let $S_1, S_2, \ldots, S_k$ be the partition of S induced by the equivalence relation r. Then, we can show that the area of a pipelined realization of such a group is $\Omega\left(n \max_{i=1}^{k} |S_i|\right)$.

Another important notion is that of the *group generators*. Let us use π^2 to denote $\pi \circ \pi$ and use a similar interpretation for π^i. Each permutation π generates a subgroup of G, $< \pi >$ as $\{\pi^i\}_{i=0}^{n}$. Here π is referred to as a *generator* of $< \pi >$. For the cyclic shifting group π_1 generates the whole group. Once again, a relation r relates two permutations π_j and π_l if $\pi_j \in < \pi_l >$. *The number of equivalence classes in this relation is the least number of generators we need to physically design to realize the permutation group.* The most compact design for a permutation group will contain as few physical permuting structures as the number of equivalence classes. Each physical permuting structure performs all the permutations required by an equivalence class through time-mapped permutation compositions (similar to the one discussed in Subsection 3.3).

The cyclic shifting group discussed in Subsection 3.2 perhaps is the most commonly occuring permutation group in VLSI designs. In view of the preceding discussion, our discussion centers around the design space of the cyclic shifting group only.

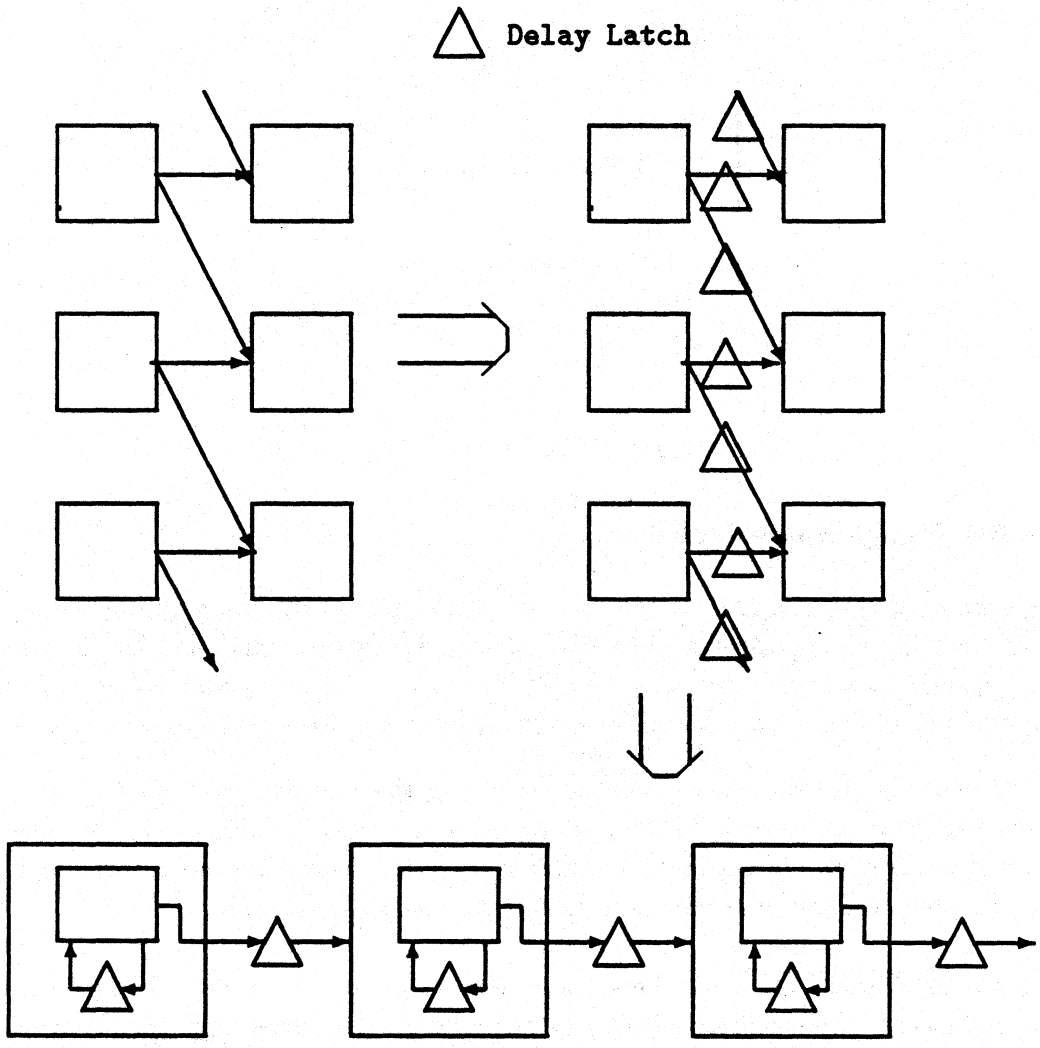


Figure 6: Space-Time Mapping to Derive a Linear Shift Register from $\pi_1 \circ \pi_1 \circ \ldots \circ \pi_1$

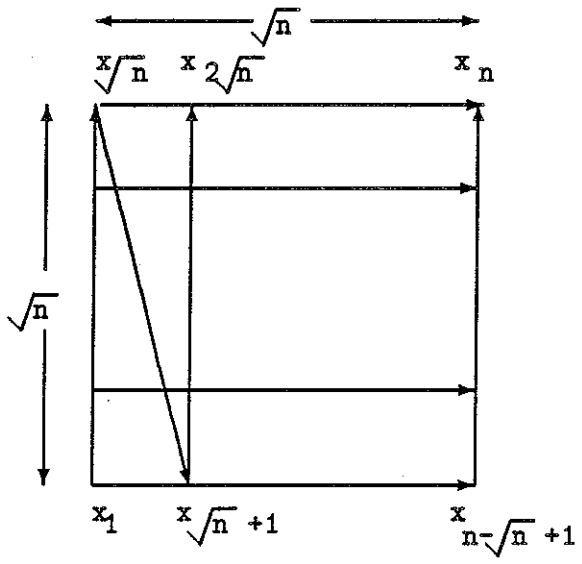


Figure 7: A Square Shifter

3.4.1 Shifter Design Space Model

The cyclic permutation group is $G = \{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$. Note that the permutation π_i performs a right-cyclic shift by i bit positions. The permutation π_1 forms a generator for the whole group G. We can specify a shifter design in a way similar to the monoid communication design. The most obvious specification is the composition of a generator element repeated n (its order) times. For the cyclic shift group, we get $\overbrace{\pi_1 \circ \pi_1 \circ \ldots \pi_1}^{n \text{ copies}}$. Using the rule to map a space realization of a composition to a time realization (as used in adders and monoid computation), we can derive a linear shift register as shown in Figure 6. We cannot map this linear structure into a single cell since n storage elements is the minimum required to compute a type-2 function.

The expression $\overbrace{\pi_1 \circ \pi_1 \circ \ldots \pi_1}^{n \text{ copies}}$ can be grouped into $\pi_{n/2} \circ \pi_{n/4} \circ \ldots \circ \pi_2 \circ \pi_1$ to derive a *barrel shifter*. Each element in this expression corresponds with a stage of a barrel shifter. We can derive a pipelined barrel shifter by changing the operator $\circ$ to a synchronous operator $\circ_\Delta$ giving us the expression $\pi_{n/2} \circ_\Delta \pi_{n/4} \circ_\Delta \ldots \circ_\Delta \pi_2 \circ_\Delta \pi_1$.

Another interesting way to realize a shifter is to decompose the domain of permutation elements and then use smaller permutation elements over several domains to realize larger permutations. Let us explain this for the square shifter shown in Figure 7. This shifter is described in Ullman [Ull84]. 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

type	*energy*	*area*	*time*	*group specification*
linear	n^2	n	n	π_1
barrel	n^2	n^2	$\log n$	$\pi_{n/2} \circ \pi_{n/4} \circ \ldots \circ \pi_1$
square	$n^{3/2}$	n	$\sqrt{n}$	as described

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

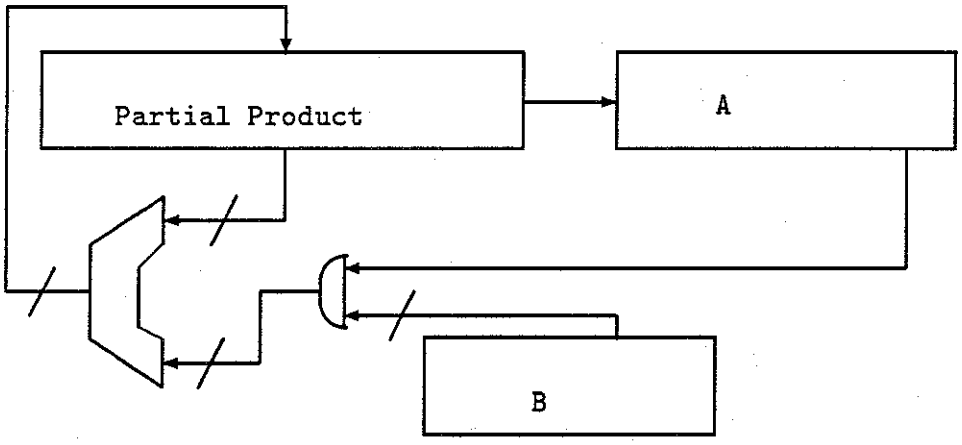


Figure 8: A Parallel-Serial Multiplier

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: $c_{\text{up}} = c_{\frac{\log n}{2}} \cdots c_1$ and $c_{\text{right}} = c_{\log n} \cdots c_{\frac{\log n}{2}+1}$. 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 permutation group has been split into $2\sqrt{n}$ domains as follows. The $\sqrt{n}$ rows are realized by the permutations $(1\ \sqrt{n}+1 \ldots n-\sqrt{n}+1)\ (2\ \sqrt{n}+2 \ldots n-\sqrt{n}+2) \cdots (\sqrt{n}\ 2\sqrt{n} \ldots n)$. The columns are realized by the $\sqrt{n}$ permutations $(1\ 2 \ldots \sqrt{n})\ (\sqrt{n}+1 \ldots 2\sqrt{n}) \cdots (n-\sqrt{n}+1 \ldots n)$. The permutation groups are realized by the π_1 linear shift registers. The groups are bridged by inter-column wires. For more examples of such decompositions, the reader is referred to Tyagi [Tya90]. We also describe some shifting schemes based on the repeated input bit availability and their physical realizations. Table 2 summarizes a part of the design space of shifters.

4 Design Space Composition to Derive Multiplier Design Space

The design space models for an adder and a shifter were built from our understanding of these two functions. This approach would not be very practical for a silicon compilation environment. For

a composite function $f \circ g$, where the design space models of f and g are already known, we should be able to derive the design space model for $f \circ g$ from the models for f and g. We have made limited progress in this direction. We illustrate the derivation of the design space model for multiplication from the models for addition and shifting. We also give a brief sketch of the composition process when the communication between f and g is not combinational but is directed by a finite state machine. This is the case for any pair of modules in a datapath.

4.1 Multiplication Model

The most common algorithm for multiplication is the *shift and add* algorithm, we all learn in the grade school. We start with a specification of a parallel/serial multiplier shown in Figure 8. We wish to multiply two $n+1$-bit integers $B = b_n \ldots b_0$ and $A = a_n \ldots a_1 a_0$. The result can be written as $\pi_0(a_0.B) + \pi_1(a_1.B) + \ldots + \pi_n(a_n.B)$. This defines a generic parallel-parallel multiplier, which can be refined to give more compact designs. Let us first transform this expression into $\pi_1 (\ldots (\pi_1 (\pi_1 (\pi_1 (a_n B) + a_{n-1} B) + a_{n-2} B) + a_{n-3} B) \ldots) + a_0 B)$. This could be done by observing the arithmetic progression of the indices for π in the parallel-parallel specification. Notice that now we have the shift composition performed in space. Applying the earlier space-time mapping rule, we can realize this expression as shown in Figure 8. The design space model would consider this design to be suboptimal for the following reason. The asymptotic time to perform π_1 (shift by 1) is some unit time c_0. The adder is in series with π_1 in this composition. The adder design space shows the fastest adder (parallel-prefix) at $\log n$ time. This mismatch can be resolved in one of the following two ways.

We can convert the adder to a carry-save adder as shown in Figure 9. Then the time per carry-save addition comes down to unit time c_1. *This is a match within a constant satisfying the design space model.* Hence this design is considered an optimal design. Note that a carry-save adder is just a bit serial adder, which is a time-domain realization of a carry-ripple adder as shown in Figure 5. When a full adder (G_1, P_1) gets used for an n-bit long stream in this way, we denote it by $(G_1, P_1)^n$. We denote this scheme by explicitly stating the domain of the linear shift register π_1 as $\pi_1 \left(\overbrace{(G_1, P_1)^n, \ldots, (G_1, P_1)^n}^{n \text{ copies}}, a_n, a_{n-1}, \ldots, a_1, a_0 \right)$.

The second way to achieve this match is by noticing that the time-mapped realization of a carry-ripple adder would work right with a linear shift register as long as the input B (in Figure 8) can be serialized in the right way. This is one example of the information that this design space model lacks. This limitation is also the reason that we can compose models this easily. Figure 10 illustrates the serial-serial multiplier. The specification for this multiplier consists of $\pi_1 ((G_1, P_1)^n, 2, \ldots, n)$.

Another transformation we can consider is to notice the symmetry of multiplication with respect to its two input words. This requires looking at the parallel-parallel specification and taking its dual with respect to the two inputs. Then one can derive the column-add serial-parallel multiplier shown in Figure 11. The specification here is $\pi_1 (Ab_0, (G_1, P_1)^n, Ab_1, (G_1, P_1)^n, \ldots, Ab_n, (G_1, P_1)^n)$. One

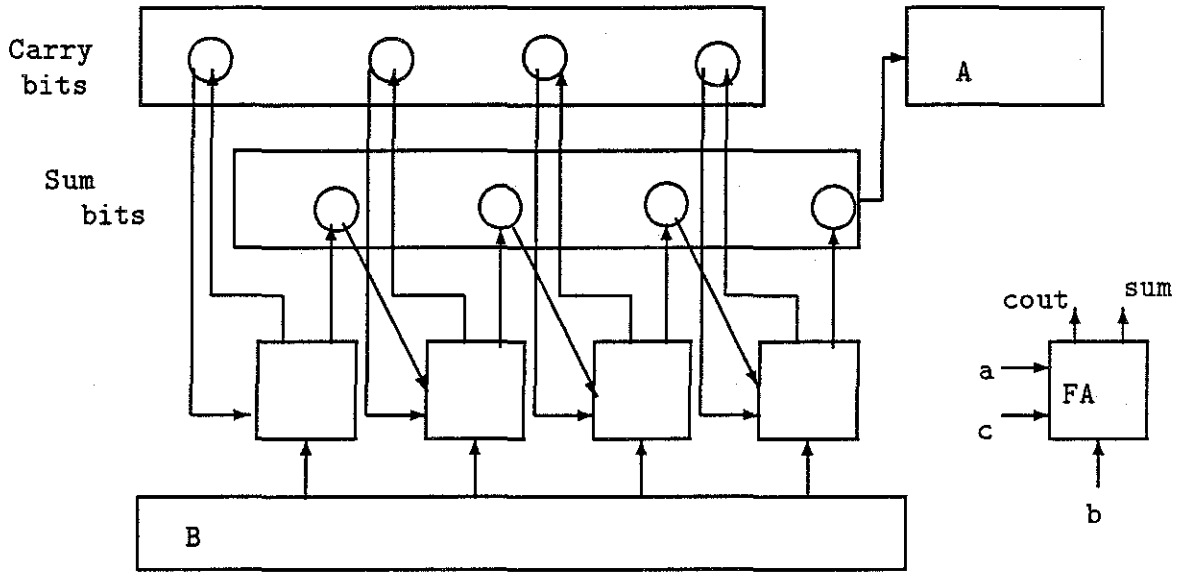


Figure 9: A Parallel-Serial Multiplier with Carry-save Adder

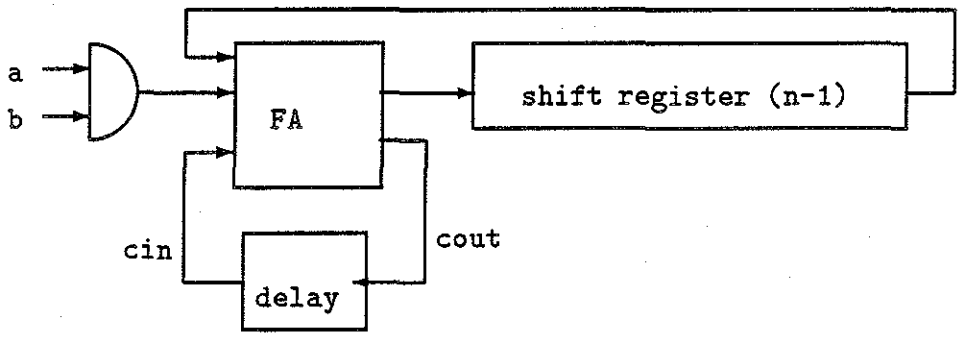


Figure 10: A Serial-Serial Multiplier

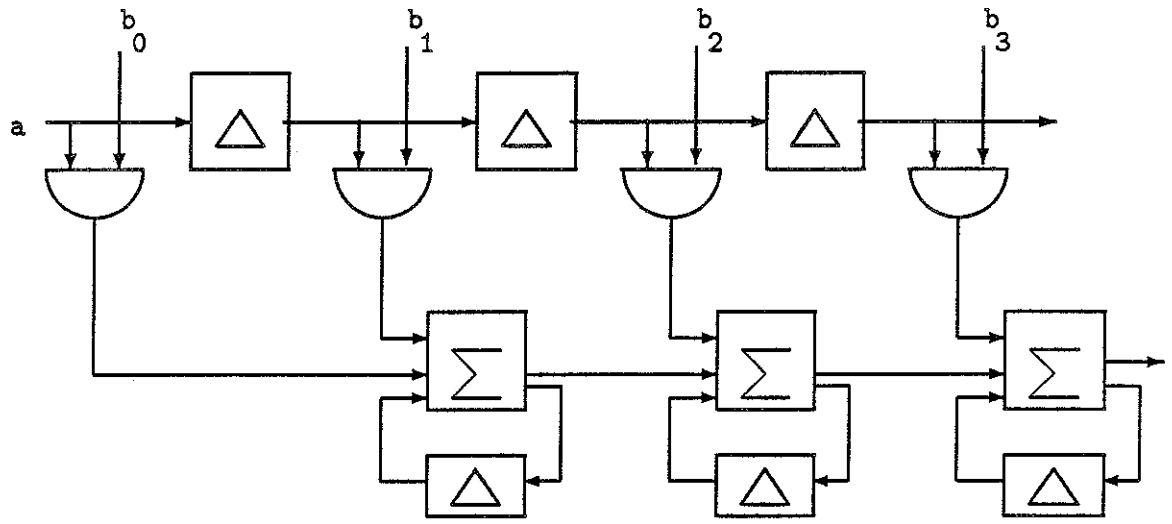


Figure 11: A Column Add Serial-Parallel Multiplier

can derive Lyon's pipelined version [Lyo76] from this by introducing delay latches along every signal path across the shift boundaries.

In summary, the following techniques/operators guide the search of the design space (massaging of the algebraic expression) with respect to a given set of user area and time requirements. In all these cases, the constituent primitive components are matched in speed.

space-time mapping: When repeated subexpressions occur, the compositions can either be realized in the expanded form in space or folded in time. This space-time mapping can either reduce space into time to realize a smaller area design point or map time into space to realize a faster design point.

communication bandwidth: Algebraic elements with larger indices have larger communication bandwidth at the cost of larger area. A subexpression that evaluates to a larger-index element can be replaced for a faster design point. This process can also replace a large-index element by a sub-expression consisting of small index elements giving rise to a lower area but slower design point.

factoring: When elements with indices in an arithmetic progression are found, they can be folded to use the least index element repeatedly. A parallel-parallel square multiplier can be derived from a Wallace tree multiplier in this way.

symmetry: An axis transformation for the expression gives rise to a whole new space of designs. For example, the multiplication can be seen as adding a new row to the partial product at

every stage. Alternately, it can be seen as the addition of columns from right to left. This perspective leads to the Lyon's pipelined multiplier.

4.2 Composition under FSM Control

In the multiplier case (combinational composition), the design space model has to determine the speed of two components so that one can feed another at the rate the other can consume the data. But in this situation the two components need not exhibit the sequential dependence. For example, a program counter (PC) needs to compute the new address in parallel with the ALU operation in a pipelined machine such as MIPS-X [CH87]. The additional complexity is that every pipe stage in MIPS-X is divided into two non-overlapped clock phases ϕ_1 and ϕ_2. Hence we need to determine if both the PC and ALU have nonoverlapped operations or simultaneous operations. The simultaniety argues for the designs with as matched a time performance as possible, so that no component is overdesigned.

We have made the picture look very simplistic. One needs to perform the same kind of slack analysis as is performed in high-level synthesis (behavior level) [MPC88]. But the domain is a finite state machine where an output signal indicates if a module is activated or not. The same technique will work nonetheless. Only the pairs of design points for two modules that satisfy the FSM constraints and minimize the total area are propagated up as the design points at this composite level.

5 Conclusions and Future Work

We have described a design space model that models only the communication component of a function. But for the functions that are complex (and thus hard to characterize), it is the communication component that dominates the area-time requirements of a design. We believe that this constitutes the minimal information a design space model needs to carry forward to be a successful model. For the most purposes, this information is also sufficient. We gave a general framework to model all the datapath functions in terms of type-0, type-1 and type-2 functions or a composition thereof. We have used these models to build function module generators for *adder*, *shifter* and *multiplier*. These module generators exhibit enough flexibility to satisfy most of the user specifications.

We also proposed a technique for composing two design space models to derive a composite model for a larger system in the hierarchy. This was sketched with the example of the derivation of the design space model of a multiplier from the models for adder and shifter.

This work needs to be extended to do this composition for a system whose modules communicate under the control of a finte state machine. We gave a preliminary method to do this.

As pointed out in the discussion, a model of timing and clocking is missing from this model. There is a macro-concept of a delay-latch to allow us to consider pipelined and synchronous implementations. An interesting direction will be to consider if different clocking schemes can fit

into this framework and if a new, consistent clocking scheme can be derived from the same design transformation.

In summary, although the complete designs have been characterized by formal methods, we believe that this is the first attempt at modeling the design space by formal means. This is a more pragmatic way to handle the design space exploration tasks.

References

[Bau81] G. M. Baudet. On the Area Required by VLSI Circuits. In *Proceedings of CMU Conference on VLSI*, pages 100–107. CMU, Computer Science Press, 1981.

[BK82] R.P. Brent and H.T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, pages 260–264, March 1982.

[CH87] P. Chow and M. Horowitz. Architectural in the Design of MIPS-X. In *Proceedings of the 14th ACM International Symposium on Computer Architecture*, pages 300–308. ACM, 1987.

[Che85] 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 International Conference on Computer Design.* IEEE, 1985.

[Che87] M.C. Chen. The Generation of a Class of Multipliers: Synthesizing Highly Parallel Algorithms in VLSI. Technical Report YALEU/DCS/RR-406, Yale University, May 1987. To appear in IEEE Transactions on Computers.

[CS83] P.R. Cappello and K. Steiglitz. Unifying VLSI Array Designs with Geometric Transformations. In *Proceedings of International Conference on Parallel Processing.* IEEE, 1983.

[JC81] 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.

[JMP88] R. Jain, M. J. Milner, and A. C. Parker. Area-Time Model for Synthesis of Non-Pipelined Designs. In *Proceedings of ICCAD-88*, pages 48–51. IEEE, 1988.

[Joh86] S. D. Johnson. Digital Design in a Functional Calculus. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 45–57. Elsevier Science Publishers (North-Holland), Amsterdam, Holland, 1986.

[Joh89] D. Johanssen. Silicon Compilation. In *Proceedings of the 1989 Decennial Caltech Conference on VLSI*, pages 17–36. MIT Press, 1989.

[KB88] G. Kedem and F. Brglez. OASIS: Open Architecture Silicon Implementation System. Technical Report MCNC TR 88-06, Microelectronics Center of North Carolina, February 1988.

[KH89] M.S. Kotliar and K.S. Hedlund. Speed Optimization of Combinational Circuits. In *Proceedings of the 1989 International Workshop on Logic Synthesis.* MCNC/ACM SIGDA, 1989.

[LM82] T. Lin and C. Mead. The Application of Group Theory in Classifying Systolic Arrays. Technical Report 5006:DF:82, California Institute of Technology, Pasadena, California, April 1982.

[Lyo76] R. F. Lyon. Two's Complement Pipeline Multipliers. *IEEE Transactions on Communications*, April 1976.

[Mol83] D. I. Moldovan. On The Design of Algorithms for VLSI Systolic Arrays. *IEEE Transactions on Computers*, 71(1), January 1983.

[MPC88] M. C. McFarland, A. C. Parker, and R. Camposano. Tutorial on High Level Synthesis. In *Proceedings of the 25th Design Automation Conference*, pages 330–336. ACM/IEEE, 1988.

[NI85] T-F. Ngai and M. J. Irwin. Regular, Area-Time Efficient Carry-Lookahead Adders. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, pages 9–15. IEEE, 1985.

[PP86] N. Park and A. C. Parker. Sehwa : A Program for Synthesis of Pipelines. In *Proceedings of the 23rd Design Automation Conference.* IEEE-ACM, 1986.

[PSE85] 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.

[Qui84] P. Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *Proceedings of the 11th Annual Symposium on Computer Architecture.* IEEE, 1984.

[Rob84] F. S. Roberts. *Applied Combinatorics.* Prentice Hall Inc., 1984.

[She84] M. Sheeran. muFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 104–112. ACM, 1984.

[Tya90] A. Tyagi. An Algebraic Model for Design Space with Applications to Module Generation. In *Proceedings of the First IEEE European Design Automation Conference.* IEEE Computer Society Press, 1990. *Also available as The Dept. of Computer Science, UNC, Chapel Hill, TR89-032.*

[Ull84] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Md., 1984.

[Vui83] J. Vuillemin. A Combinatorial Limit to the Computing Power of VLSI Circuits. *IEEE Transactions on Computers*, pages 294–300, March 1983.

[Yao79] A.C. Yao. Some Complexity Questions Related to Distributed Computing. In *ACM Symposium on Theory of Computing.* ACM-SIGACT, 1979.