BIT-DRIVEN LOGIC:

A STYLE OF DIGITAL LOGIC FOR VLSI DESIGN

George E. Gerpheide

VLSI Research Group Memo No. 1

22 April 1980

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

This research was supported in part by the National Science Foundation under Grant MCS78-04853.

Table of Contents

Abstract

This memo describes a new style of low-level digital logic design called Bit-Driven Logic (BDL) which may prove attractive for the design of VLSI chips. BDL is an application of speed-independent, data-flow ideas to a very low level. It has the advantages of good locality, clockless operation, and inherent pipelining leading to high throughput. The G-Net, a graph model similar to the Petri Net, is presented to represent BDL circuits and the throughput of acyclic G-Nets is investigated. The concepts of skew and thickness, and the use of shims as well as drawings in Time Normal Form, are presented to enable acyclic G-Nets to be designed which support maximum throughput. The suitability of uniform-array concepts for VLSI implementation, particularly the SLA, is shown by means of examples, the most involved of which is an iterative array multiplier.

1 Introduction of Bit-Driven Logic Concept

This paper presents a new and rather unusual style of low-level digital logic design called Bit-Driven Logic (BDL) which seems to offer some significant advantages over the currently predominate style, Register-Transfer Logic (RTL). In BDL, every operator, even one as simple as a single conventional gate, is data-driven. That is, an operator is quiescent until required data becomes valid at its inputs; then it removes the data from its inputs and creates result data at its outputs. In contrast, the combinatorial logic operators of RTL continuously produce results based on the input values with no regard to data "validity". Propagation time assumptions are used to design clock timing which attempts to ensure that only "valid" data is clocked into registers.

Most conventional thinking in logic design segregates storage from logic, as exemplified by the synchronous state machine and the register-ALU organization of most computers. With BDL, in contrast, storage and logic are highly integrated; storage is associated with each gate-level operator. The result is inherent pipelining at an atomic level which gives BDL the potential

for very-high-throughput circuits.

BDL represents an extreme position in the data-driven (or data-flow) philosophy which many researchers are currently applying to issues of computer architecture and programming [Davis 78, Dennis 74, Keller 79, Misunas 77, Misunas 78]. Most of this data-driven research has dealt with modules of fairly high complexity, such as adders, multipliers or list-processors, and usually the internal organization of these modules has employed the conventional RTL style. BDL simply carries the data-driven philosophy to a lower level.

It has usually been felt that applying data-driven ideas to the low level that BDL does is hopelessly costly in components and therefore of only theoretical interest. The ready availability of numerous MSI integrated circuits oriented toward RTL, coupled with the lack of even the simplest integrated circuits expressly-suited for BDL has strongly encouraged such thinking. However, now that we are in the position of thinking about the design of an entire system on a VLSI chip, the component cost difference is no longer clear. To be sure, the BDL implementation of a simple operator will almost always require more silicon area than a similar RTL implementation, but it will be shown later that (given the right types of computation problems) BDL utilizes inherent pipelining with very fine granularity to achieve much higher throughput than is possible with RTL of the same switching speed and fan in/out. Thus it might turn out that a BDL system will actually require less silicon area than an RTL system of equal performance (i.e., one that can perform the same computation in the same amount of time).

Perhaps the most important feature of the BDL style of design is the high degree of locality which is rather naturally achieved; a locality both of design conceptualization and of space in the physical realization. Conceptual locality is important to simplify the design process, and to make verification of correctness easier. Spatial locality becomes very important for VLSI design as smaller geometries and high speeds are attempted [Seitz 79]. Good

locality is closely tied to modularity and results from most data-flow approaches. By carrying the data-flow philosophy to a lower level, and by eliminating clocks and their attendant timing problems completely, BDL is expected to achieve very good locality.

In addition, the BDL style of design appears to be very compatible with the concept of uniform arrays such as the SLA [Patil 79], which should make VLSI design easier in two important ways. The first is shared by any design style using uniform arrays. The time-consuming and costly process of IC layout is replaced by an array-programming process allowing the design of VLSI chips independent of transistor-level device characteristics or fabrication methods. Much like a higher-level software language, the SLA may lend portability to a chip design.

The second advantage of the BDL-SLA combination, which would not be shared by an RTL-SLA combination, is transfering circuit timing considerations from the logic designer's job to the initial (one-time) design of the SLA internal cell structure. A clocked RTL system requires consideration of operator delay times in the design of the clock timing. Without such knowledge, i.e., without knowledge of the delays of the component SLA cells, an RTL system can not be guaranteed to be functionally determinate. A BDL system, on the other hand, can be so guaranteed from structural considerations alone. Thus, a BDL system is truly portable from one SLA fabrication technology to another while an RTL system, with clock timing based on the cell delay characteristics, is not.[1]

As an added benefit, the locality of BDL may facilitate a simpler layout level design for the SLA cell than if the SLA were to be used for RTL, since

---

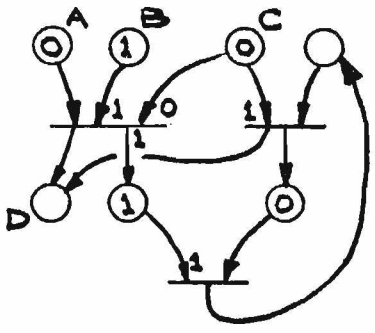
[1]For certain types of systems, in which a "real-time" interface with the external world places time constraints on the computations to be performed, a BDL system will not be entirely portable. It is hard to imagine any design style which would give an entirely portable system in such cases.

greater locality on a chip allows stronger internal timing constraints to be assumed.

2 G-Nets - A Graph Model for Bit-Driven Logic

2.1 Definition of G-Nets

Briefly, a G-Net (Figure 1 for example) resembles a Petri Net with colored tokens of two colors, called "0" and "1".



A, B, and C are input places.

D is an output place.

Figure 1: Example G-Net

Special labels on the input arcs control the colors of input tokens required to enable a transition, and labels on the output arcs determine the colors of tokens generated. An initial marking has at most one token per place and a strict firing rule prevents more than one from ever occupying a single place. As with Petri nets in full generality, conflicts are allowed.

In somewhat more detail, a G-Net is a directed graph with an initial marking. There are two types of vertices, called places and transitions, and two types of arcs. Input arcs originate at places and terminate at transitions, and output arcs are in the opposite direction. No other arcs are permitted, nor are multiple arcs (i.e., those with a common origin and destination). A marking of a G-Net associates either no, a single "0", or a single "1" token with each place of the net. At some marking, a place with no token is said to be empty and one with a token is full. Graphically, a G-Net uses circles to represent places and bars for transitions. The tokens of a marking are represented by writing nothing, "0" or "1" inside each place.

With respect to some transition, a place connected to that transition by an input arc is called an input place and one connected by an output arc is

called an output place[2]. With respect to some G-Net, an input place is one which is an input place to some transition in the net, but an output place for none. Similarly, an output place is an output for at least one transition but an input for none. All other places in the net are called internal places.

A simulation of a G-Net is a sequence of firings of transitions of the net; each firing is an integral event which occurs some time after a transition becomes enabled. The simulation assumes an input process which places tokens in input places of the net as they become empty, and an output process which removes tokens from full output places; these processes represent the interface between the net and the "outside world".

When firing, a transition assumes a temporary color based on its inputs, removes all tokens from its input places and creates tokens at each of its output places with colors based on its temporary color and markings on the output arcs. If two or more transitions with a common input or output place become simultaneously enabled, a conflict is said to occur. In case of a conflict, only one transition actually fires.

A transition is enabled when all the input places (A2, A, B, C, and D of Figure 2) contain specified tokens and all the output places (E, F, G and H) are empty.

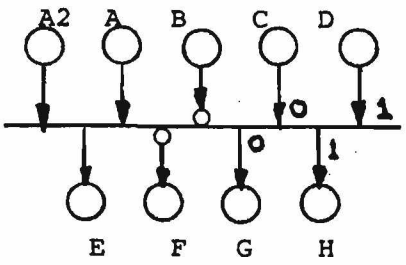


Figure 2: Example transition

Which colors of tokens are required at the input places is specified by

---

[2] A place which is both an input and output for a single transition is allowed, but due to the strict firing rule, it will prevent the transition from ever firing.

markings on the input arcs. A "0" next to the arc means that a 0 token is required; similarly a "1" means a 1 is required. An unmarked input may be either a 0 or a 1, but all unmarked inputs of one transition must be the same color. An input arc with a small inverting circle requires a token which is the complement of that on any unmarked inputs; if there are no unmarked input arcs, the inverting input may be either 0 or 1. In any case, all inverting inputs of one transition must be the same. Thus, for the transition of Figure 2 to be enabled, C must be 0, D must be 1, and either A2 and A are both 0 while B is 1 or A2 and A are 1 while B is 0.

The temporary color assumed by a transition during firing is determined as follows: If there are any unmarked inputs the transition takes the color of that input, otherwise if there are inverting inputs the complement of these inputs is used, and if neither then the transition is given the color 1. Each output place is given a token colored as follows: A "0" or a "1" means that a 0 or 1 token is to be output, an unmarked arc means that a token with the transition's color is to be output and an arc marked with an inverting circle means that the complement of the transition's color is to be output. For Figure 2, the color assumed by the transition will be the same as the token on the A or A input. E will receive a token with the transition's color, F the complement of that color, G a 0 and H a 1.

2.2 Using G-Nets for BDL

At this point, the relation between G-Nets and BDL should be considered. Transitions represent data-driven operations with complexity similar to single gates of conventional logic. Places represent data links from one operator to the next, and each provides storage for a single bit. Transitions are conjunctive, as each requires a full tuple (or one of two tuples if unmarked or inverting inputs are present) with a specific value of colored tokens present at the inputs to be enabled, and produces tokens at all the outputs when firing. Places are disjunctive since they can receive/distribute tokens from/to any one of several transitions.

Boolean logic operators can be implemented in a manner similar to conventional "AND-OR" 2-level logic by using one transition for each "AND" term and a place to "OR" the transitions together. For example, Figure 3(a) illustrates one possible half-adder circuit in G-Net notation. Note that no action will occur until tokens are present at both input places. Then one of the four transitions, depending on the input values, will fire and deliver the result to the output places. Also, Figure 3(b) depicts a 3-bit gate unit; tokens at the input places cannot pass to the output places until a token is delivered to the control place.

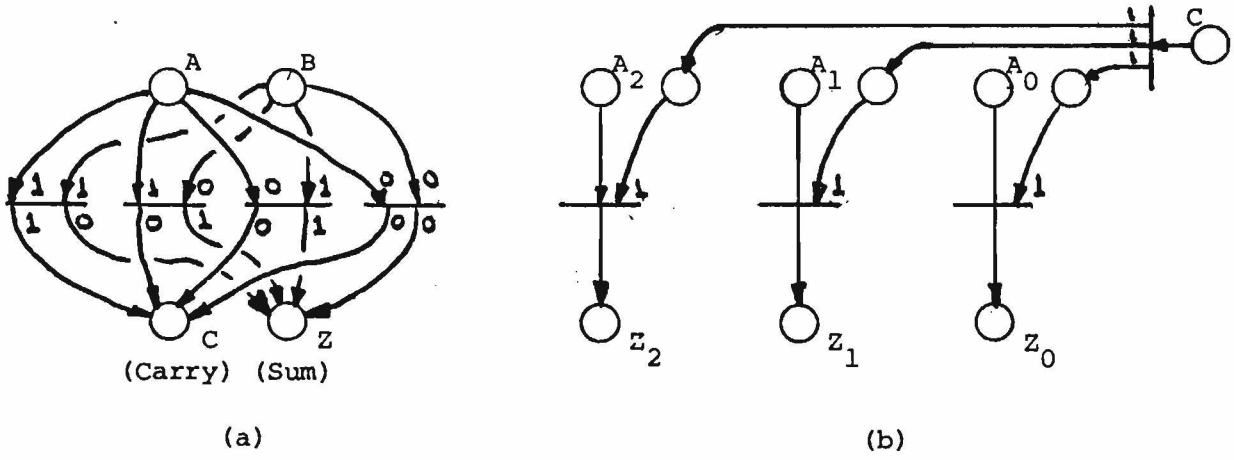


Figure 3: Example G-Nets: half-adder (a) and 3-bit gate (b)

It is often convenient to use some G-Net as a building block in the construction of a more complex G-Net. For this purpose, the G-Net module notation exists. A module is similar to a simple transition: it has input and output arcs and places, and its actions are enabled by tokens which are accepted from (a subset of) its input places and it creates tokens at (a subset of) its output places. The module is drawn as a box with double lines on the sides and single lines on top and bottom. Inside the box is a name which identifies that G-Net which the module, together with its input and output places, replaces. The input and output places are drawn external to the module with single arcs to or from the module. Thus, several modules or transitions can share input or output places. The number and functionality of input and output places of the module is the same as the G-Net for which it

substitutes. Each input and output arc of the module should be labelled to identify it with a specific arc of the reference G-Net; but if many modules are used in a single net, it is usually assumed that all have the same arrangement of arcs and only one module needs to be so labelled. Figure 4 is an example of a G-Net and the corresponding module representation.

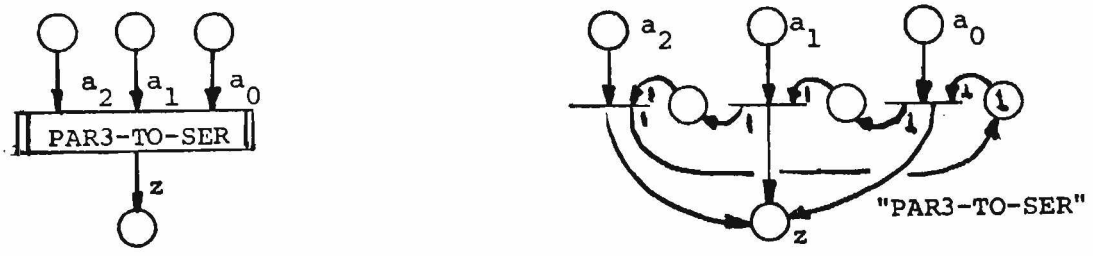


Figure 4: Example module and corresponding G-Net

A special type of module which satisfies certain restrictions is called a macro transition. It is drawn with only single lines on the sides but otherwise identically to the module notation. The firing properties of a macro transition mimic those of a simple transition: When a macro transition fires, a token is removed from every input place and one is created at every output place. There are no internal places and each transition of the reference G-Net has arcs to each of the input and output places. Figure 5 gives an example of a macro transition and the net it represents; note that the module of Figure 4, in contrast, cannot be a macro transition.

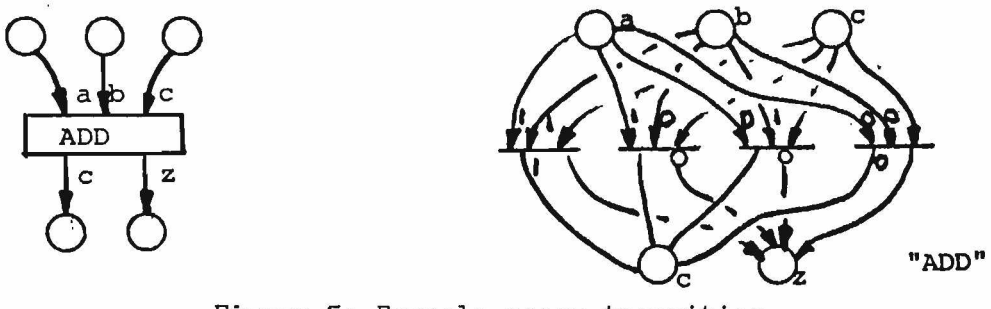


Figure 5: Example macro transition

3 N-Bit Ripple-Carry Adder Demonstrating Pipeline Effect

In this section, a concrete example is presented which performs useful computation and demonstrates the bit-level pipelining which occurs automatically, leading to very high throughput. Figure 6 illustrates a G-Net which adds two N-bit operands, A and B, to produce an N+1 bit result,

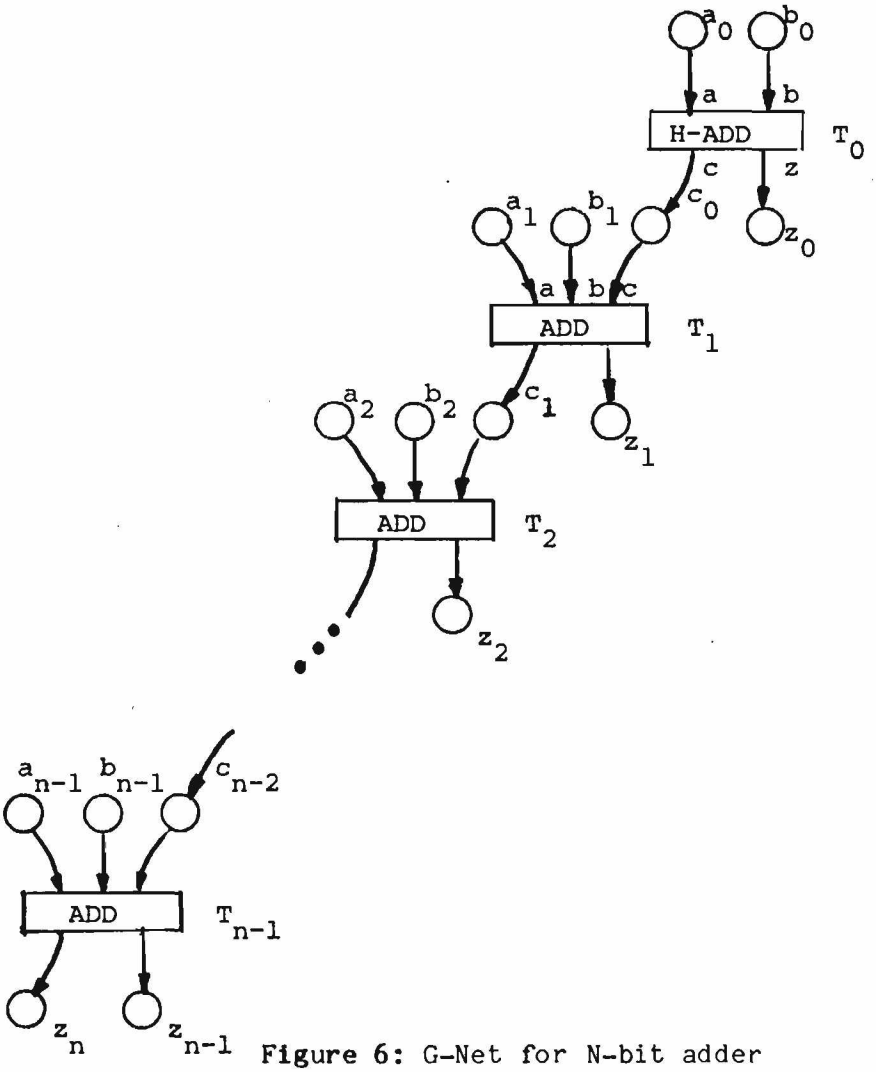


Figure 6: G-Net for N-bit adder

Note that ripple-carry organization is employed; therefore the fan-in and fan-out required of the elements is low, which may be an important practical consideration.

To analyze the adder's operation, let us first consider that all places in the net are initially empty. Assume that an infinite stream of operand pairs are to be added and that ideal operand sources are connected to the A and B

input places; as soon as any input place becomes empty the appropriate bit from the next operand to be used is put in that place. Similarly, an ideal operand sink is connected to the Z output places which consumes tokens of the result as soon as they are produced at Z.

No operation occurs until the A_0 and B_0 places receive tokens of the first pair of operand words. Then the 1/2-ADD transition T_0 fires, consuming the tokens at A_0 and B_0 and producing result tokens at Z_0 and the carry, C_0. Since A_0 and B_0 are now empty, the least significant bit tokens of the second pair of operands are placed in A_0 and B_0 and the pipe begins to fill. Concurrently, since a token has arrived at C_0 and the ideal operand sources will have filled the $A_1$and B_1 input places, the ADD transition T_1 fires consuming A_1, B_1 and C_0 and producing C_1 and Z_1. Also, the operand sink consumes Z_0 which was produced earlier.

Now C_0 and Z_0 are empty and T_0 can again fire, consuming the A_0 and B_0 tokens of the second operand pair and producing Z_0 and a carry C_0. Two other operations proceed at this same time: A_1 and B_1 receive tokens from the second operand pair and T_2 fires to consume A_2, B_2 and C_1 producing Z_2 and propagating the carry, C_2 of the first operand pair. With A_0 and B_0 again empty, the A_0 and B_0 tokens of the third operand pair enter the adder pipe. So far, Z_2, Z_1 and Z_0 outputs have been produced for the first operand pair and Z_0 for the second pair. As operation continues, carries propagate to the left, new pairs of operands enter the adder pipeline least-significant-bits-first and results for successive pairs leave the pipe (also least-significant-bits-first).

When the carry from the first operand pair reaches T_{N-1}, the pipe will have been filled. N/2 transitions at a time will be firing, odd-numbered ones alternating with even. With each volley, N/2 new output bits will be produced; therefore the throughput will be $1/2\tau$ adds per second where τ is the firing time for a transition. Note especially that this result is independent of word length, N!

4 Performance of RTL Versus BDL Operators

Given some computation which an operator is to perform, two important performance measures can be defined. The first is delay time, the time from arrival of (the first part of) the input to output of (the last part of) the result. The second is throughput, the rate at which the computation can be repeated given a stream of input values.[3] It should be remembered that BDL exhibits pipelining to a rather extreme degree; its performance is related directly to this pipelining. Delay time for a BDL operator is similar to that for an RTL operator, assuming equivalent complexity and switching times for the elements. BDL gains its advantage in throughput, as discussed below.

Throughput for an RTL combinatorial-logic operator is just the reciprocal of the maximum delay time, since the result from one input operand set must be computed and clocked into an output register before it is safe to change the input to the next operand set.[4] Since all bits of an operand word are clocked simultaneously, the maximum delay is the longest path from any input bit to any output bit, which may be much longer than any path from input to output for a single bit position. Considering an N-bit ripple-carry adder, the longest path follows the carry from the LSB (least significant bit) input to the MSB (most significant bit) output. If the delay of each full-adder element is τ, then the overall operator delay is $N\tau$ and the throughput is $1/N\tau$. Since N will often be fairly large, the throughput of the BDL N-bit ripple-carry adder ($1/2\tau$ as derived in section 3) can be much better than that of the equivalent RTL operator (by a factor of 32 for N = 64-bit words).

In current RTL practice, carry-lookahead rather than ripple-carry schemes

---

[3] These measures are also known as latency and bandwidth, respectively.

[4] Sometimes a minimum "storage time" as well as maximum delay time for an operator is known, in which case the throughput can be increased somewhat. It is difficult in practice to improve the throughput by more than a factor of about two in this way.

can be used which utilize higher fan in/out elements to decrease the number of stages through which the carry must propagate and thereby significantly decrease the operator delay. Unfortunately, elements become correspondingly less local which might be undesirable from the standpoint of VLSI implementation.

Examining the cause of this low throughput of the ripple-carry adder is helpful. Define meaningful switching as that operation between the arrival of "valid" data at all inputs of an element (gate, full-adder, etc.) and output of a valid result. For the RTL ripple-carry adder, the LSB full-adder element will first do some meaningful switching, then the next and so on, rippling along with the carry to the MSB. Thus, at any instant, only 1 of N elements will be doing meaningful switching while the others either have invalid inputs or are merely maintaining a result. It would seem more efficient if each element were spending nearly all its time doing meaningful switching. The key to achieving this end seems to be allowing different bits of the data words to arrive at different times. This idea of pipelining each bit independently has been explored in a synchronous clocked array-multiplier context [Agrawal 75]. BDL achieves this naturally since each bit carries its own timing information.

5 Throughput of Acyclic G-Nets

For the N-bit adder of the preceding section, it is fairly clear that, under the right conditions, a high throughput can be obtained. But it is not at all obvious how (or if) similarly high throughput can also be obtained at a system level. In this section, concepts pertinent to optimization of throughput in G-Nets will be developed. Particular emphasis will be given to interfacing between G-Net modules, and techniques will be developed to design entire systems with optimum throughput. Although admittedly requiring rather large G-Nets, such systems can be designed which carry out complex calculations with the same high throughput (independent of word length) as the N-bit adder.

5.1 Basic Concepts

First, a clarification of exactly what is meant by throughput of a G-Net is in order. Although in general one might talk about the throughput of entire N-bit operands at a particular place in complex structures, only the rate at which single-bit tokens pass through very simple non-cyclic G-Nets will be considered here. Throughout the section, throughput will be considered in the steady state assuming that appropriate input and output processes are operating to maintain that steady state. Additionally, the time for any transition to fire after being enabled is assumed to be constant and uniform, designated by τ.

Since tokens are conveyed through a net by the firing of transitions, the greater the transition firing rate, the higher the throughput. And the overall firing rate proves to be easier to study than the detailed flow of tokens through the net. Using the firing rate, an expression is obtained relating an upper bound on the maximum throughput which a simple linear chain (see Figure 7 for example) of N places (and N-1 transitions) can sustain to the number of tokens in the chain. Note that each transition has exactly one input and one output, therefore throughput is the same at all points in the net and the number of tokens in the net remains constant.

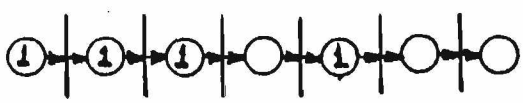


Figure 7: Simple linear chain of 7 places.
Active fraction at this marking is 2/7

Let ρ be the <u>active fraction</u>; that expected fraction of (the total N) places from which tokens are being removed (by transition firing) at any instant. The corresponding rate of token flow advances ρN tokens one place per τ in the N-place chain. To maintain the steady state, ρ tokens per τ must be entering (and leaving) the chain. The throughput is thus ρ/τ. In the remainder of this memo, ρ alone will be used for the <u>normalized</u> throughput (expressed in units of tokens per firing time).

In the case of the above simple linear chain, an upper bound, ρ_u, can

easily be placed on the active fraction and hence on the throughput by considering the fraction of places which contain tokens. Consider that to enable a transition, its input place must contain a token while its output place is empty. Thus, if N/2 of the N places contain tokens, at most N/2 transitions can fire and $\rho_u = 1/2$. On the other hand, if $j < N/2$ tokens are present, then at most j transitions can fire and $\rho_u = j/N$; or if $j > N/2$ are present (N-j empty places) then at most N-j can fire and $\rho_u = (N-j)/N$. More concisely, for the simple linear chain of N places of which j are kept filled,

$$\rho_u = \min[j/N,(N-j)/N] \qquad (1)$$

which has a maximum value of 1/2 when $j = N/2$.

Note that ρ_u is only an upper bound and might not be achievable. For example, suppose that the first N/2 places are filled and the rest empty; only a single transition can fire yet $\rho_u = 1/2$. On the other hand, if every other place is filled then it is fairly easy to see that the bound of 1/2 will be achieved.

Now consider a slightly more complex G-Net: two parallel linear chains, one of M places and the other of $N \geq M$ places, synchronized at the start and finish as shown in Figure 8.

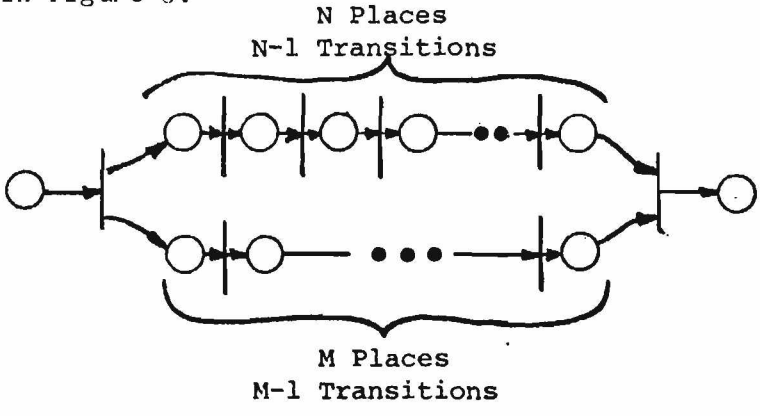


Figure 8: Two parallel linear chains

Assume that the net is initially empty. Note that whenever a token is introduced into one path by T_0 it is also introduced into the other path. Similarly, T_1 removes tokens equally from the two paths. Thus, the number of tokens present in one path is always the same as the number present in the

other. Further, notice that this same synchronizing action requires that the throughput of the whole net be the same as that of each of the two paths. Now we can simultaneously plot the upper bound which each of the two paths places on its throughput (and hence the overall throughput) for a given number of tokens, j, in each path. See Figure 9.

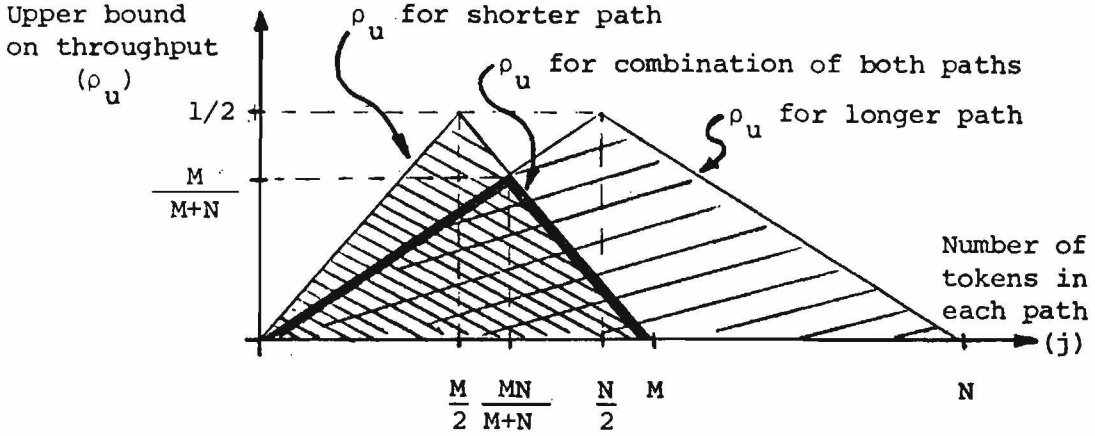


Figure 9: Plot of upper bounds for parallel paths

The point at which the two bounds intersect gives the maximum bound for the overall structure:

$$\rho_u = M/(M+N) \qquad (2)$$

which occurs when $j = MN/(M+N)$ tokens in each chain. If the two paths are balanced, ie., $M = N$, then $\rho_u = 1/2$ and it is easy to see that this bound can in fact be attained. Unbalanced paths have smaller values of ρ_u, decreasing asymptotically to a value of M/N for very unbalanced paths. Clearly, balanced paths are desired for optimum throughput.

5.2 G-Net Module Characterization

To obtain optimal system throughput, it is useful to characterize modules using two properties: skew and thickness. Also, by drawing G-Nets in Time Normal Form (TNF) it is easier to appreciate these properties. These concepts are made manageable and useful by the assumption throughout the following that all transition firings take nearly the same amount of time. Furthermore, time will be expressed in units of transition firing-times.

Input skew is a measure of the time difference between the arrivals of two tokens of one operation set at a single (if the two tokens are carried on the

same serial data path) or two different input places such that minimum waiting occurs. If no confusion will result, the word "input" is omitted. For example, in the N-bit adder of Figure 6, SKEW(a_0,a_1) = -SKEW(a_1,a_0) = +1 meaning that a token should arrive at place a_1 one firing-time after one arrives at a_0 to give minimum (in this case no) waiting. In many cases, including the N-bit adder of Figure 6, skews between any two successive bits of an operand are the same. Then, for brevity, we refer to the skew of the entire operand word; eg., SKEW(A) = 1.

Output skew is a measure of the time difference between production of two tokens of one operation set at a single or two different output places assuming that input tokens are received with minimum waiting. As with input skews, the word output is often omitted and when appropriate the term is applied to an entire result word. A very similar measure, internal skew, applies when one or both of the places under consideration is an internal place.

Thickness measures the time delay from arrival of an input token to production of an output token, again assuming that input tokens are received with minimum waiting. In simple cases, the thickness from input place a_i to output place z_i is just the number of transitions in the longest path from a_i to z_i. Thickness, like skew, is often applied to entire words when appropriate.

Using the above properties, the N-bit adder of Figure 6 can be characterized as follows:

1. SKEW(A) = SKEW(B) = SKEW(Z) = 1
2. SKEW(A,B) = 0
3. THICKNESS(A,Z) = THICKNESS(B,Z) = 1

Notice that 2. is actually implied by 3. and therefore redundant. Skew and thickness are sometimes not defined, or are data dependent as in Figure 10. But in most cases, they are quite useful measures.

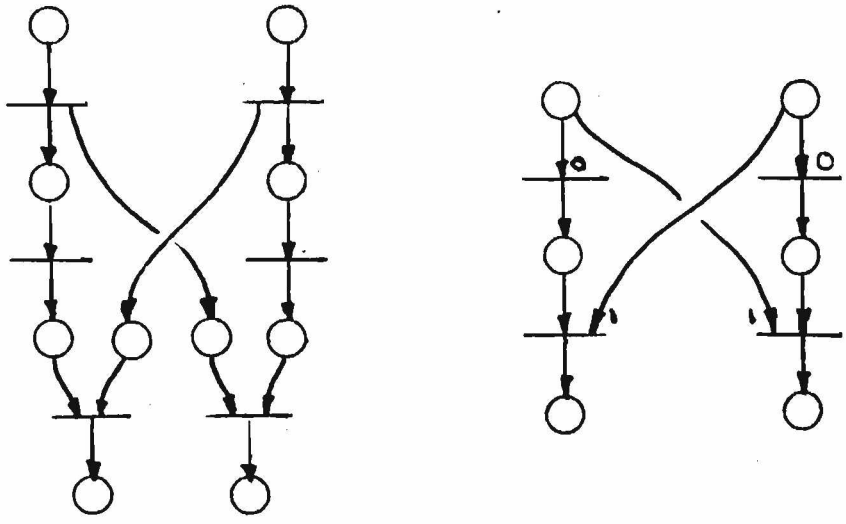


Figure 10: G-Nets with ill-defined skews and thicknesses

Time Normal Form (TNF) is a style of drawing acyclic G-Nets which illuminates skew and thickness. As an example, the G-Net of Figure 6 is drawn in TNF. To draw a TNF G-Net, the drawing surface is imagined to be divided into levels, numbered starting with 0 at the top and increasing toward the bottom. The net is then drawn according to the following rules:

1. Arcs of the net always point downwards, from a smaller to a greater level.

2. Places are always drawn directly on the levels, transitions between levels.

3. Arcs will traverse as few levels as possible subject to 1. and 2.

4. G-Nets for operators dealing with N-bit parallel data words should be drawn with the least significant bit position at the right and increasing significance toward the left.

Inspection of a TNF G-Net illuminates the throughput which that net can sustain. If rule 3. is obeyed, any arc traversing more than one level must be part of the longer of two synchronized parallel paths. The throughput of this parallel structure is less than optimum, as expressed by equation (2). Such a bottleneck can be corrected by inserting extra places and identity transitions into the shorter path until the two paths are balanced. TNF G-Nets thus point out possible design deficiencies.

In addition, thickness and skew of a TNF G-Net are readily apparent: the skew between tokens arriving at two places is just the difference in levels of the places. Skew of an entire operand is just the slope of the line passing through all the places which hold that operand, where slope from a to b is defined as

[Level(b)-Level(a)]/[BitPosition(b)-BitPosition(a)].

Thickness is simply the level of the output place minus the level of the corresponding input place.

Block diagrams can also utilize the advantages of TNF. A block diagram is essentially a G-Net which contains modules, and for convenience, lines between modules are assumed to implicitely contain places. The shape of the module's box conveys information: its height corresponds to the height of the operator (some scale being assumed) and the slopes of the top and bottom edges (inputs and outputs respectively) give the skews. Figure 11 gives such a TNF block diagram which adds three numbers using the N-bit adder of Figure 6 as a building block and Figure 12 gives the TNF black-box version of this 3-operand adder.

5.3 Parallel combination of G-Net Modules

Using the above concepts, the efficient combination of G-Net modules can be studied. Consider first the effect of two operators with different thickness in parallel as in Figure 13(A). If the thickness of A is M and of B is N, then expression (2) gives $\rho_u = M/(M+N)$ as an upper bound on the throughput compared to $\rho = 1/2$ which could be obtained if the operators were equal in thickness.

The rather obvious solution is to insert a flat shim of thickness N-M in series with operator A as shown in Figure 13(B). The flat shim is simply an identity operator of the specified thickness, implemented with parallel chains of equal length. The chains are uncoupled, so a flat shim may be used to increase the thickness of an operator with a skewed output (input) as well as one with no skew. Thus, operators of any thickness may be efficiently

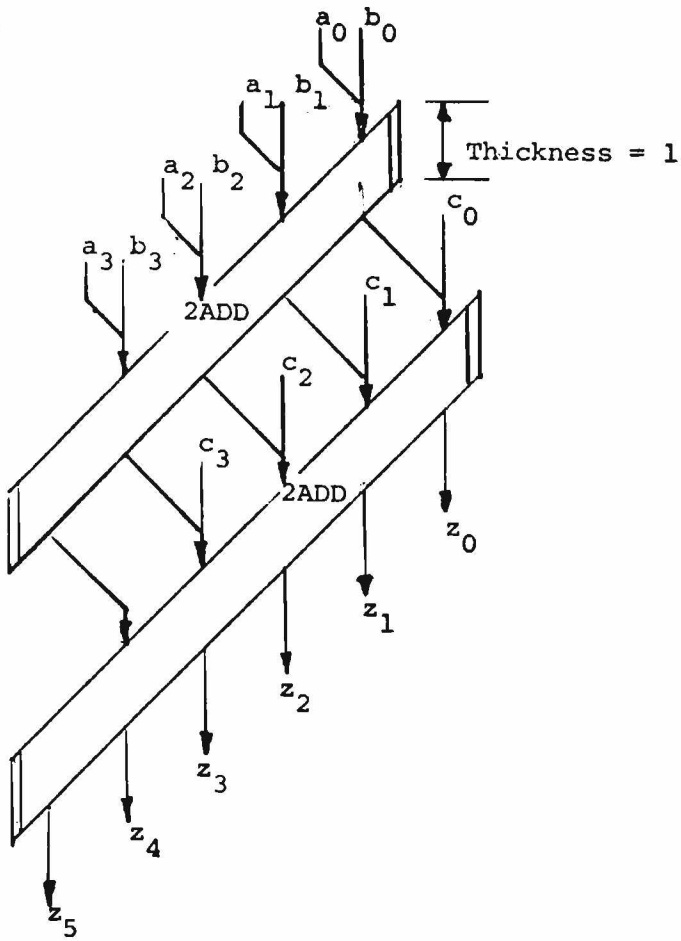


Figure 11: TNF Block diagram of 3-operand adder

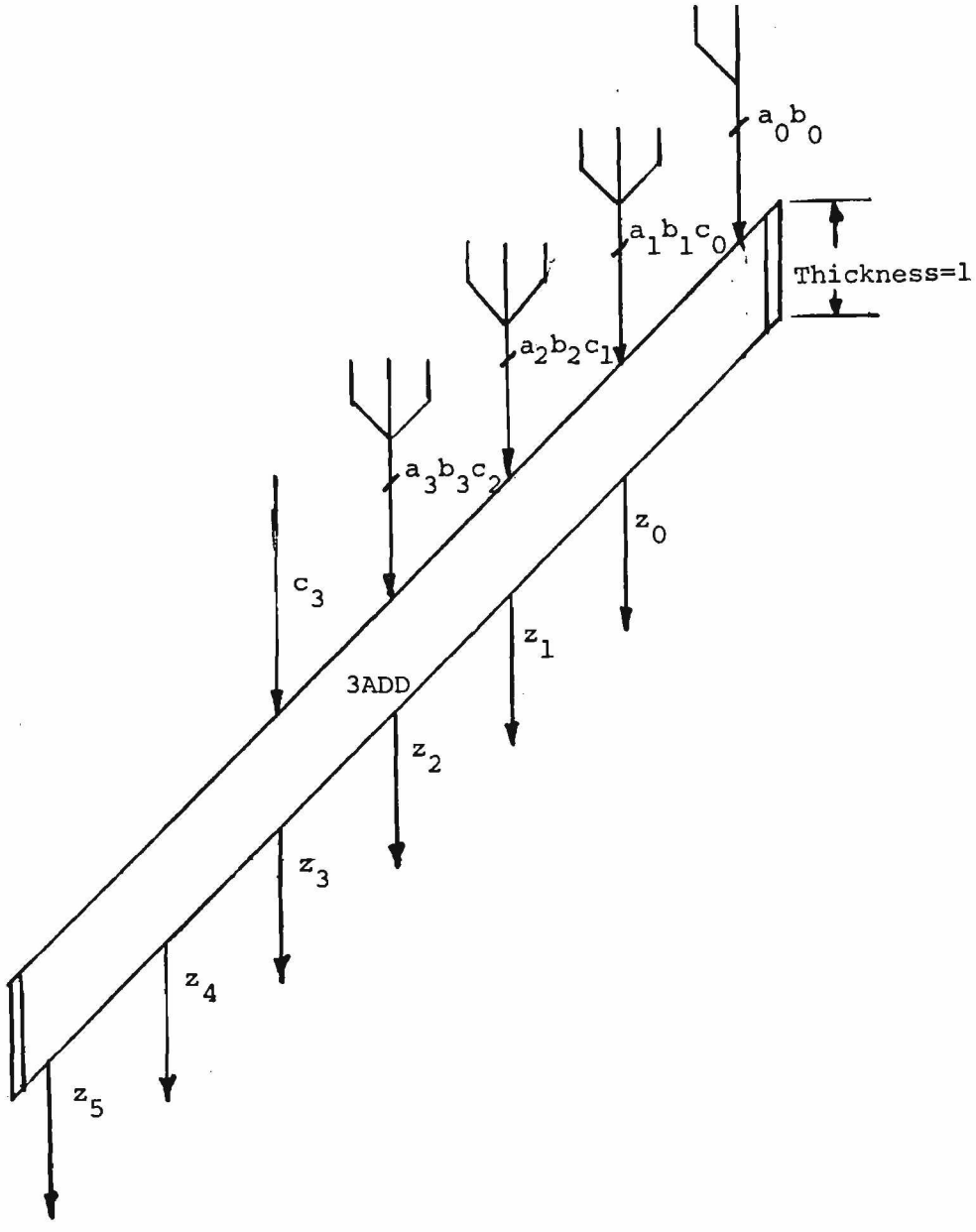


Figure 12: Module in TNF for Figure 11

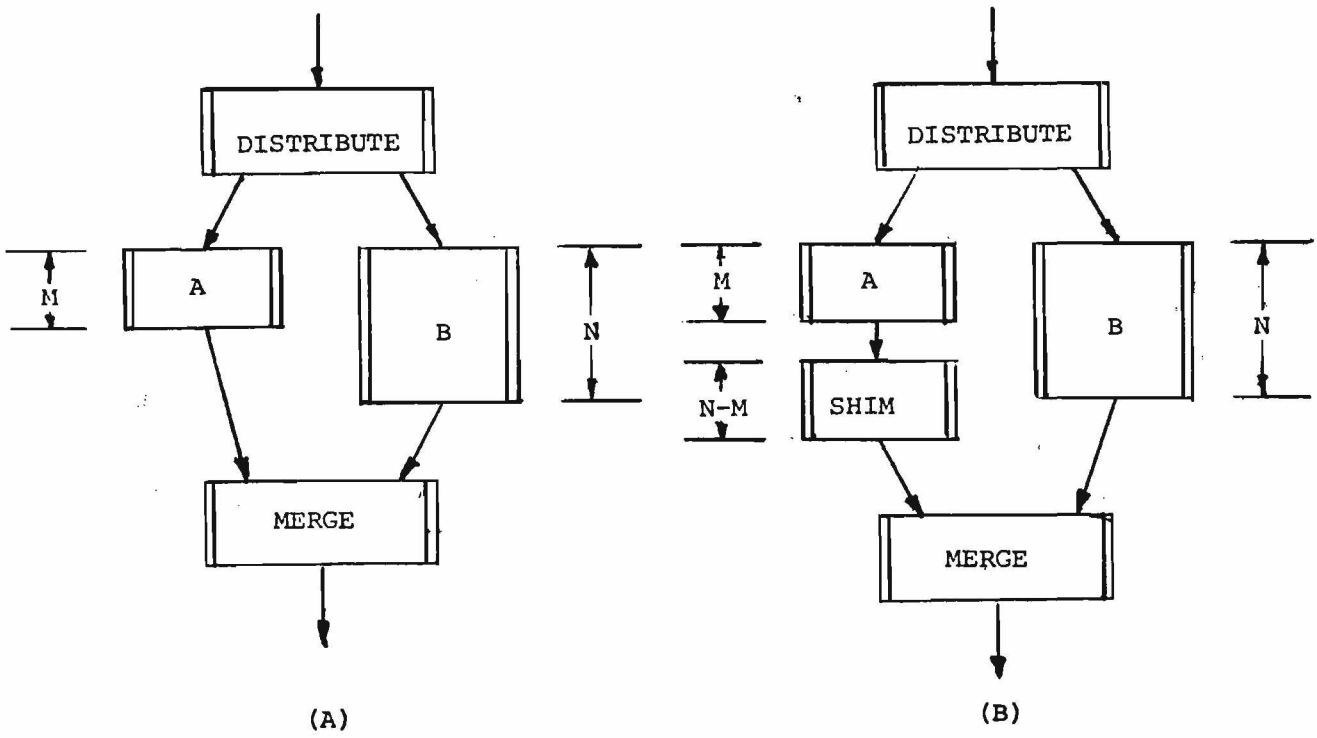


Figure 13: Parallel operators: unbalanced (A) and balanced with shim (B)

paralleled.

5.4 Serial composition of G-Nets

The harder case is the series composition of operators with different skews. Consider the output of an N-bit operator A with output skew of 1 and thickness of 1 connected to an operator B with input skew of 2 and thickness of 1 as shown in time normal form in Figure 14. From place a_0 to place z_i (other than z_0), there are several parallel paths. The shortest is almost entirely within operator A, and contains i+1 places (not counting a_0 or z_i). The longest is contained almost entirely within B and has 2i+1 places. (The shortest and longest paths for i = 3 are enhanced in Figure 14.) The greatest unbalance occurs when i = N-1, ie., the most significant bit, in which case the short path is N and the long path 2N-1 places. By equation (2), the throughput is less than optimal: $\rho_u = N/(3N-1)$. In the limit as the word length becomes very long, $\rho_u = 1/3$.

Using the same approach, upper bounds on the throughput for the composition of two arbitrarily skewed operators can be obtained. Again assume that the

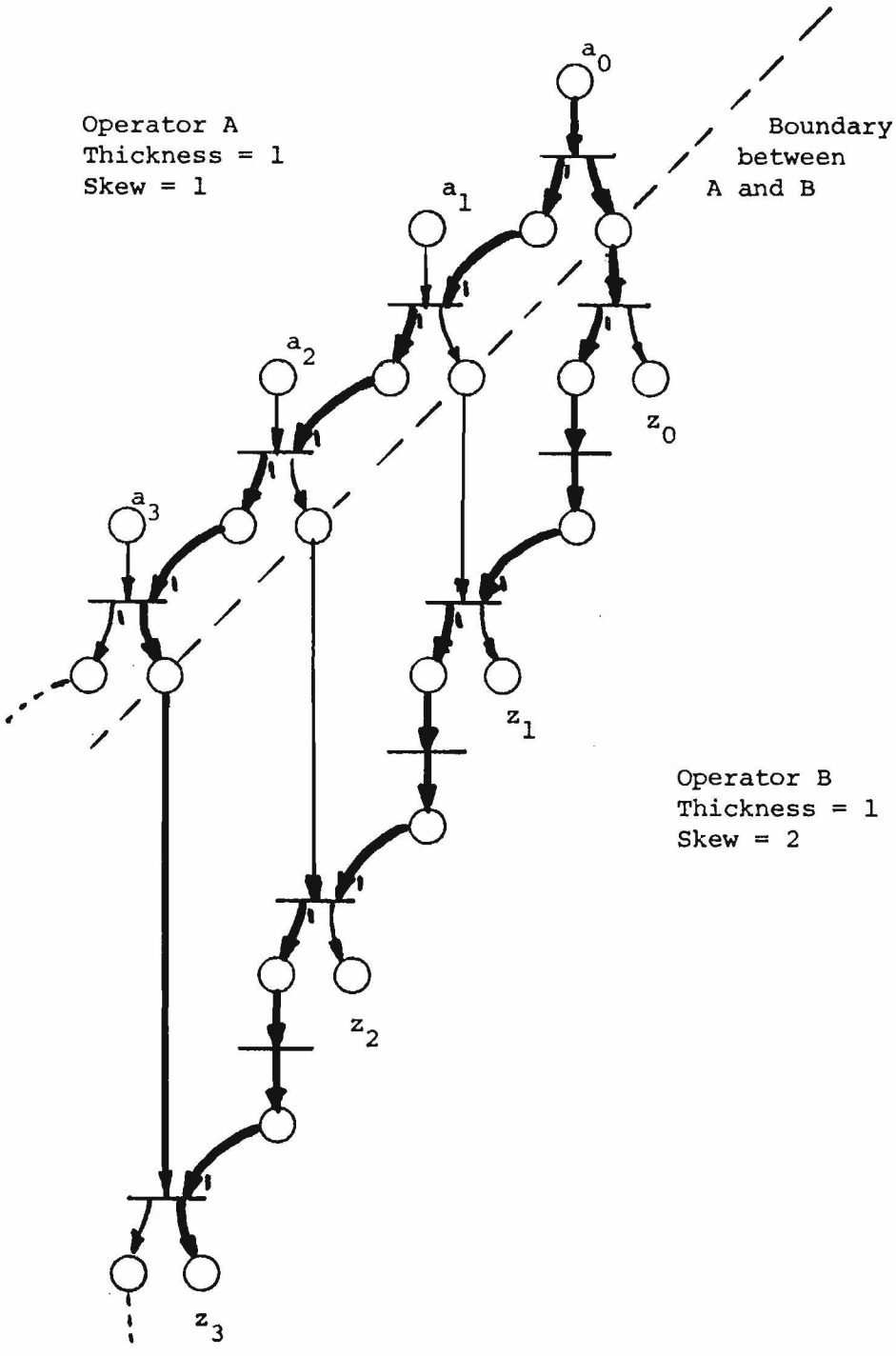


Figure 14: Cascading operators with different skews

output of A has skew a and is connected to the input of B which has skew b, and each has a thickness of 1. There are three basic cases:

1. If a and b have the same sign, and are both non-zero, then $\rho_u = [(N-1)\max(|a|,|b|)+1]/[|a+b|(N-1)+1]$ which in the limit for long words (large N) is $\rho_u = \max(|a|,|b|)/|a+b|$.

2. If a or b but not both is zero, then $\rho_u = 1/N|a+b|$.

3. If a and b have opposite signs, both non-zero, then $\rho_u = 1/[(|a|+|b|)(N-1)+2]$, which in the limit is $\rho_u = 1/(|a|+|b|)N$.

The results for case 1. are not too bad; with a and b both small integers, ρ_u is typically 1/3 - 1/5 which might be tolerable in some cases. But since N is typically fairly large (16 - 64), cases 2. and 3. rarely give acceptable throughput.[5] Fortunately, in all of these cases full throughput of 1/2 can be realized by inserting a skewed shim with skew a-b between the two operators. A skewed shim, like a flat shim, is an identity operator but its thickness varies linearly with bit position rather than being constant. A skewed shim with a skew of s, $s > 0$, has thickness of s*i at bit position i, and if $s < 0$ the thickness is s(N-1-i) at bit i. In both cases, the minimum thickness will be 0 and the maximum s(N-1). Like a flat shim, the bit positions are uncoupled. Thus a skewed shim with a skew of s can be used to convert an operand with any skew a into one with skew a+s. Figure 15 illustrates the use of a skewed shim.

Note that several cases have been presented in which throughput can be improved by the introduction of shims, which delay the flow of data. This apparently counter-intuitive result becomes clearer if one realizes that shims have storage as well as delay, and it is the increased storage for intermediate results which allows the throughput to rise.

---

[5] It is interesting though not precisely correct to consider RTL in this light. A ripple carry adder, for example, has a non-zero "skew" while the registers that feed and receive the data have a "skew" of 0.

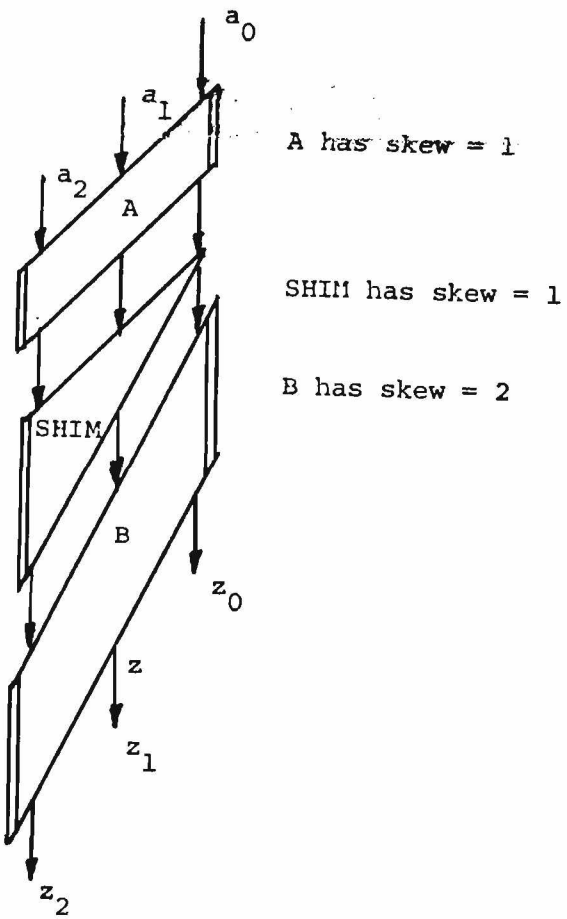


Figure 15: Use of skewed shim to give optimum throughput

In this section, concepts were presented which allow non-cyclic systems composed of serial and parallel combinations of G-Net operator modules to be designed which achieve optimum throughput. By drawing the G-Net operators in time-normal form and inserting extra places into unbalanced parallel paths, optimum throughput of the operator can be ensured and the operator can be characterized using the properties of skew and thickness. At the system level, one can deal with modules so characterized and, by inserting flat or skewed shims as required, ensure that the resulting system has optimum throughput.

6 VLSI Implementation of G-Nets

From the material so far one might view BDL as a theoretically interesting but hard-to-implement concept. Fortunately, however, it is rather well suited for implementation using uniform-array VLSI circuits such as the SLA. The SLA contains storage elements (columns) and combinational operators (rows) mixed with a very fine level of granularity throughout the array. Such a mixture is at the very heart of the BDL concept, as exemplified by the places and transitions of the G-Net representation. In this context, BDL provides an elegant low-level basis for efficient usage and design of VLSI chips.

This section presents a general mapping from G-Nets into standard SLA programs and also a more efficient mapping into programs for a specially-suited variation of the basic SLA, the G-SLA. Also a G-SLA program for the N-bit adder is presented as an example. To understand this section, the reader should be familiar with the SLA operation and programming language, as presented in [Patil 79].

6.1 Mapping G-Nets into Standard SLA Programs

Basically, places of the G-Net map into one or two columns of the SLA and transitions into one or two rows. Each column has two stable states, 0 and 1. But a place can be in one of three stable states, E (empty), 0 or 1. Generally therefore, two columns, p0 and p1, are required for each place P: both being 0 represents P being empty, p0 being 1 represents P containing a 0, p1 being 1 represents P containing a 1 and both being 1 should never occur. If it is known that a place will only contain a single type of token (perhaps because it performs a control function) then only one column is needed. In general, each transition can take on either a value of 0 or of 1 when it fires; accordingly one SLA row is used for each event. Again, if it is known that the transition can only take one of these values, then only one row is required.

The procedure for translating from a G-Net to a standard SLA program is as follows:

- For each place P in the net, one or both of columns p0 and p1 are created in the SLA as above.

- For each transition T which can assume a 1 value when firing, a row t1 is created which becomes activated for such a firing. Row t1 should contain an entry at columns $p1_i$ and $p0_i$ corresponding to each input place P_i of T as follows:

    * If P_i is unmarked or marked with a 1, enter a 1R at column $p1_i$.

    * If P_i is marked with a complementing circle or a 0, enter a 1R at column $p0_i$.

- Also, at each column $p1_o$ and $p0_o$ corresponding to an output place P_o of T, make an entry in row t1 as follows:

    * If P_o is unmarked or marked with a 1, enter a 0S at column $p1_o$ and a 0 at $p0_o$.

    * If P_o is marked with a complementing circle or a 0, enter a 0 at column $p1_o$ and a 0S at $p0_o$.

- For each transition T which can assume a 0 value when firing, a row t0 is created which becomes activated for such a firing and contains entries at columns $p1_i$ and $p0_i$ corresponding to each input place P_i of T as follows:

    * If P_i is unmarked or marked with a 0, enter a 1R at column $p0_i$.

    * If P_i is marked with a complementing circle or a 1, enter a 1R at column $p1_i$.

- Also, at each column $p1_o$ and $p0_o$ corresponding to an output place P_o of T, make an entry in row t1 as follows:

    * If P_o is unmarked or marked with a 0, enter a 0 at column $p1_o$ and a 0S at $p0_o$.

    * If P_o is marked with a complementing circle or a 1, enter a 0S at column $p1_o$ and a 0 at column $p0_o$.

Figure 16 gives some simple examples of the translation process. In addition to this straightforward procedure, consideration must be given in complex nets to physical arrangement of places and columns in the SLA if efficient use of split rows and columns is to be made.

```
ah al bh bl

1R    0S 0  {copies a 1}
   1R 0  0S {copies a 0}
```

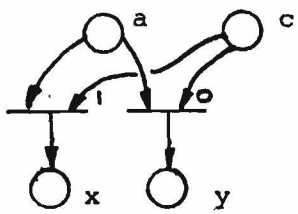
Standard SLA program G-Net

(a) Identity

```
ah al ch cl xh xl yh yl

1R    1R    0S 0          {a -> x}
   1R 1R    0  0S

1R       1R       0S 0    {a -> y}
   1R    1R       0  0S
```



Standard SLA program G-Net

(b) Switch

Figure 16: Simple G-Nets translated into standard SLA programs

6.2 The G-SLA

Standard SLAs, while functionally adequate, are somewhat cumbersome for implementing BDL circuits. The G-SLA presented here is a simple variation of the standard SLA which is tailored for the implementation of G-Nets.

Like the standard SLA, the G-SLA is a rectangular array of cells. Each column of cells stores one "state" variable while each row performs tests and/or actions on the columns which it intersects. The basic difference between the G-SLA and the standard SLA is that the G-SLA uses tristable, rather than bistable, flip flops for the columns. The three stable column states are called E (empty), 0 and 1, and testing and setting of states is implemented in the same manner as in the standard SLA.

Each cell, which is at the intersection of some particular row and column of the G-SLA, may contain either a test-action character pair, an input character or an output character, or it may be empty. If the cell is empty,

the row does not interact directly with the column. Otherwise, the characters in the cell specify a test which the column must meet for the row to be enabled, and an action to be performed on that column should the row be enabled. The tests in a row are conjunctive, i.e., tests of all cells in the row must be met for the row to be enabled.

The first character of a test-action character pair specifies the test and is one of "E", "0", "1" or "-"; "E", "0" or "1" mean that the column must be in the "E", "0" or "1" state for the row to be enabled; "-" means that the state of the column irrelevant. The second character is the action character and is one of "x", "r", "s" or "-"; "x", "r" or "s" mean that the column is to be put into the "E", "0" or "1" state if the row is enabled; "-" means no action is to be performed.

The input and output characters are shorthand versions of test-action pairs which are especiallly suited for G-Net implementation. The input characters "0" and "1" are exactly equivalent to the test-action pairs "0x" and "1x" respectively, while the output characters "r" and "s" are equivalent to "Er" and "Es". Note that test-action characters always occur in pairs while input or output characters occur singly within cells.

It is assumed that rows and columns of the G-SLA may be split at any point. One tristable flip-flop is required for each section of column; this flip-flop is assumed to require an area one column wide and two rows tall, and may be located anywhere along the column section. The flip-flop area can not be crossed by any row and is indicated in the G-SLA program by a shaded region. The flip-flop will normally be initialized to the "E" state on power-up; writing a "0" or a "1" in a circle within the shaded flip-flop region will cause it to be initialized to the "0" or "1" state.

6.3 Mapping G-Nets into G-SLA Programs

The procedure for translating a G-Net into a G-SLA program is very similar to that for the standard SLA, but only a single column is needed for each place. The resulting, slightly simpler, procedure follows:

- For each place P in the net, a column p of the SLA is created.

- For each transition T which can assume a 1 value when firing, a row t1 is created which becomes activated for such a firing. Row t1 should contain an entry at each column p_i corresponding to an input place P_i of T as follows:

    * If P_i is unmarked or marked with a 1, enter a 1 at column p_i.

    * If P_i is marked with a complementing circle or a 0, enter a 0 at column p_i.

- Also at each column p_o corresponding to an output place P_o of T, make an entry in row t1 as follows:

    * If P_o is unmarked or marked with a 1, enter an s at column p_o.

    * If P_o is marked with a complementing circle or a 0, enter an r at column p_o.

- For each transition T which can assume a 0 value when firing, a row t0 is created which becomes activated for such a firing and contains an entry at each column p_i corresponding to an input place P_i of T as follows:

    * If P_i is unmarked or marked with a 0, enter a 0 at column p_i.

    * If P_i is marked with a complementing circle or a 1, enter a 1 at column p_i.

- Also, at each column p_o corresponding to an output place P_o of T, make an entry in row t1 as follows:

    * If P_o is unmarked or marked with a 0, enter an r at column p_o.

    * If P_o is marked with a complementing circle or a 1, enter an s at column p_o.

6.4 G-SLA Program for an N-Bit Ripple-Carry Adder

Using the above translating procedure, G-SLA programs for the N-bit adder of Figure 6 are given in Figure 17 and Figure 18. Each program is composed of N nearly identical sections, each section handling one bit position, but in one program the sections are stacked vertically while in the other they are arranged horizontally. Such flexibility of form factor in G-SLA programs should prove extremely useful.

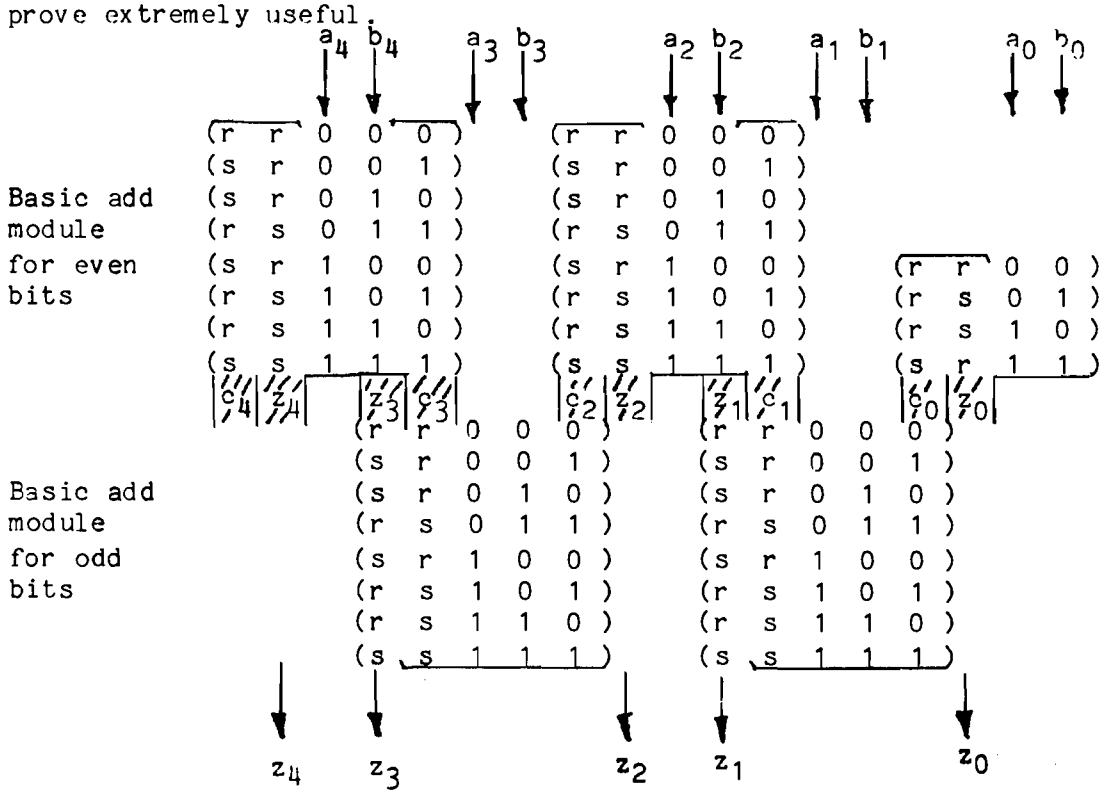


Figure 18: Horizontally organized adder G-SLA program

Notice the extensive use of split rows and columns in both programs, which highlights the fact that interconnections for this N-bit adder are highly local. That is, the maximum length of any row or column is some small fixed value (in these programs, 28 and 18) of elements independent of the word length, N. This locality is a direct result of the ripple-carry structure, which is feasible only because of the bit-level pipelining obtained with BDL. Such locality is an important beneficial feature for the design of fast, long word-length VLSI circuits.

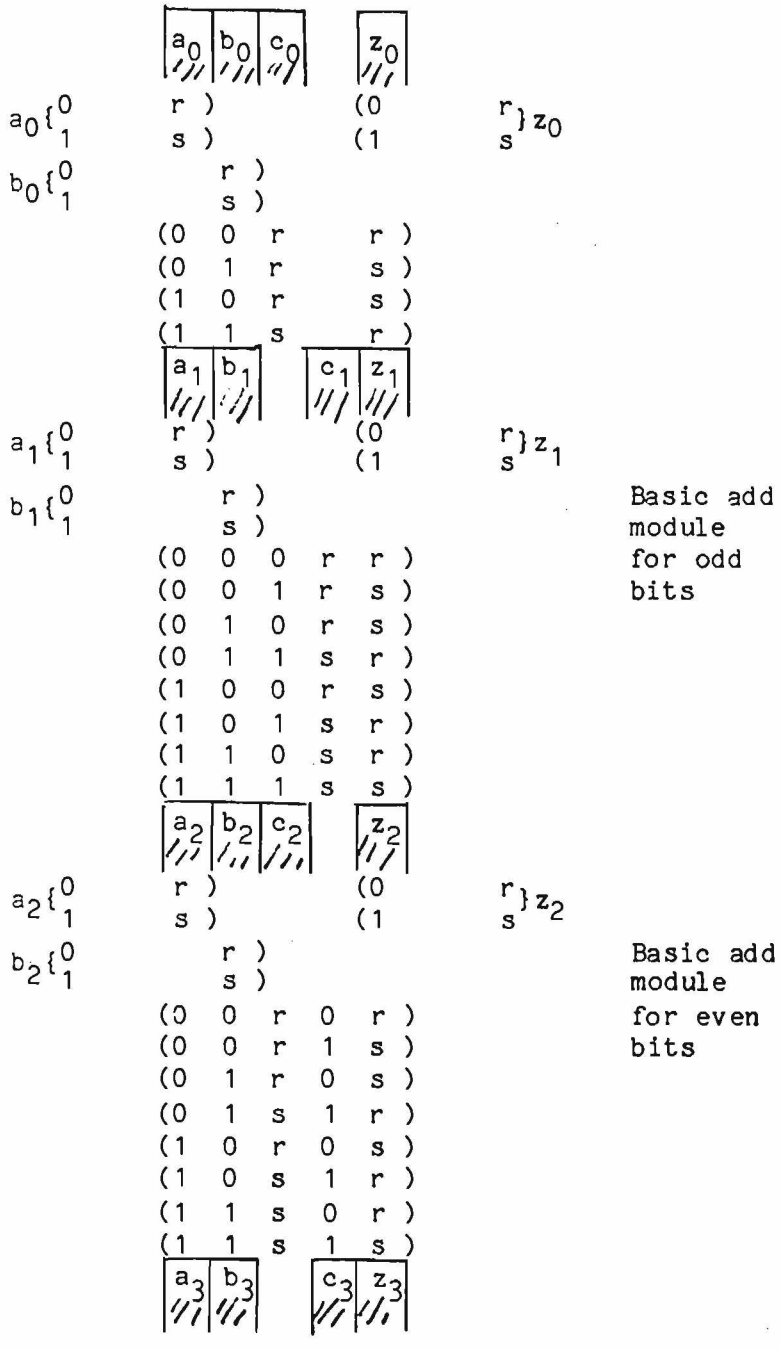


Figure 17: Vertically organized adder G-SLA program

7 Array-Organized Multiplier Module

The previous sections have introduced the BDL concept and the associated issues of notation, inherent pipelining, throughput, and implementation using the G-SLA. This section ties all of these issues together by presenting a start-to-finish design example of a non-trivial high-throughput multiplier module. The design will be presented in a structured, top-down manner.

7.1 Basic Operation

The left-shift-and-add algorithm (Figure 19) is used as a basis for this 4-bit multiplier module with the computation performed in a spatial array of 4 levels.

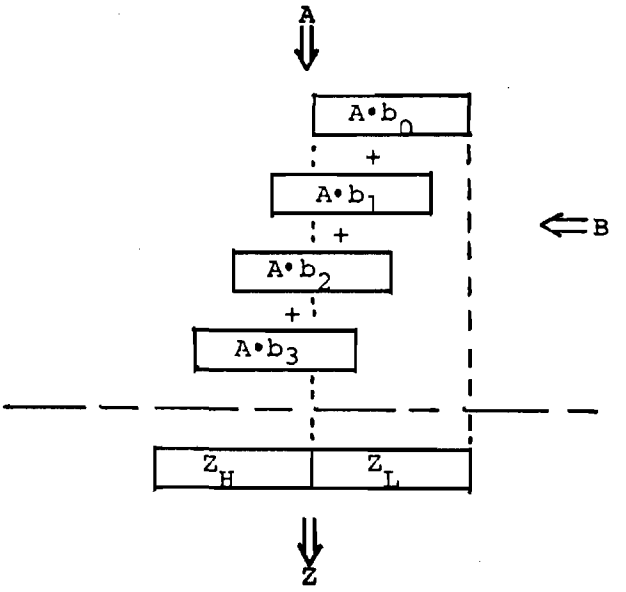


Figure 19: Basic multiplier Algorithm

At each level the A operand is multiplied by the appropriate bit of the B operand and added to a partial result. The new partial result along with a shifted copy of the A operand is passed down to the next level. The LSB of the partial result at each level passes directly to become part of Z_L, the lower order 4 bits of the result.

7.2 G-Net Description of Multiplier

Each level of computation can be performed by an "A.b" module which is an enhanced adder. Figure 20 gives a block diagram for the multiplier using these modules. Note that the first level A.b module does not use its P inputs and the last level does not produce any A' outputs, as indicated by the "X" mark on those lines.

Each A.b module can be characterized as follows:

- InputSkew(A) = InputSkew(P) = OutputSkew(A') = OutputSkew(P') = 1.
- InputSkew(a_0,b_0) = InputSkew(A,P) = 0.
- OutputSkew(P',A') = 1.
- Thickness(P,P') = 1.

The entire multiplier can also be characterized, as follows:

- InputSkew(A) = 1, InputSkew(B) = 2, InputSkew(a_0,b_0) = 0.
- OutputSkew(Z_L) = 2, OutputSkew(Z_H) = OutputSkew(z_3,z_4) = 1.
- Thickness(a_0,z_0) = 1.

The A.b module can be decomposed using "a.b" macro transitions as shown in Figure 21. Each a.b macro transition accepts p, a, b and c as inputs and produces p', a', b' and c' as outputs. Outputs a' and b' are simply copies of the a and b inputs. Outputs p' and c' are the sum and carry, respectively, resulting from the expression p+c+ab. The G-Net decomposition for the a.b macro transition is not given since it is fairly tedious and the above provides a complete description.

Note that the most significant macro transition has no b' output and the least significant has no c input. Also, recall that the first and last level A.b modules are special. In the last level, a.b transitions should have no a' output. In the first level they should have no p' input, hence no addition is performed and no c input or c' output is needed.

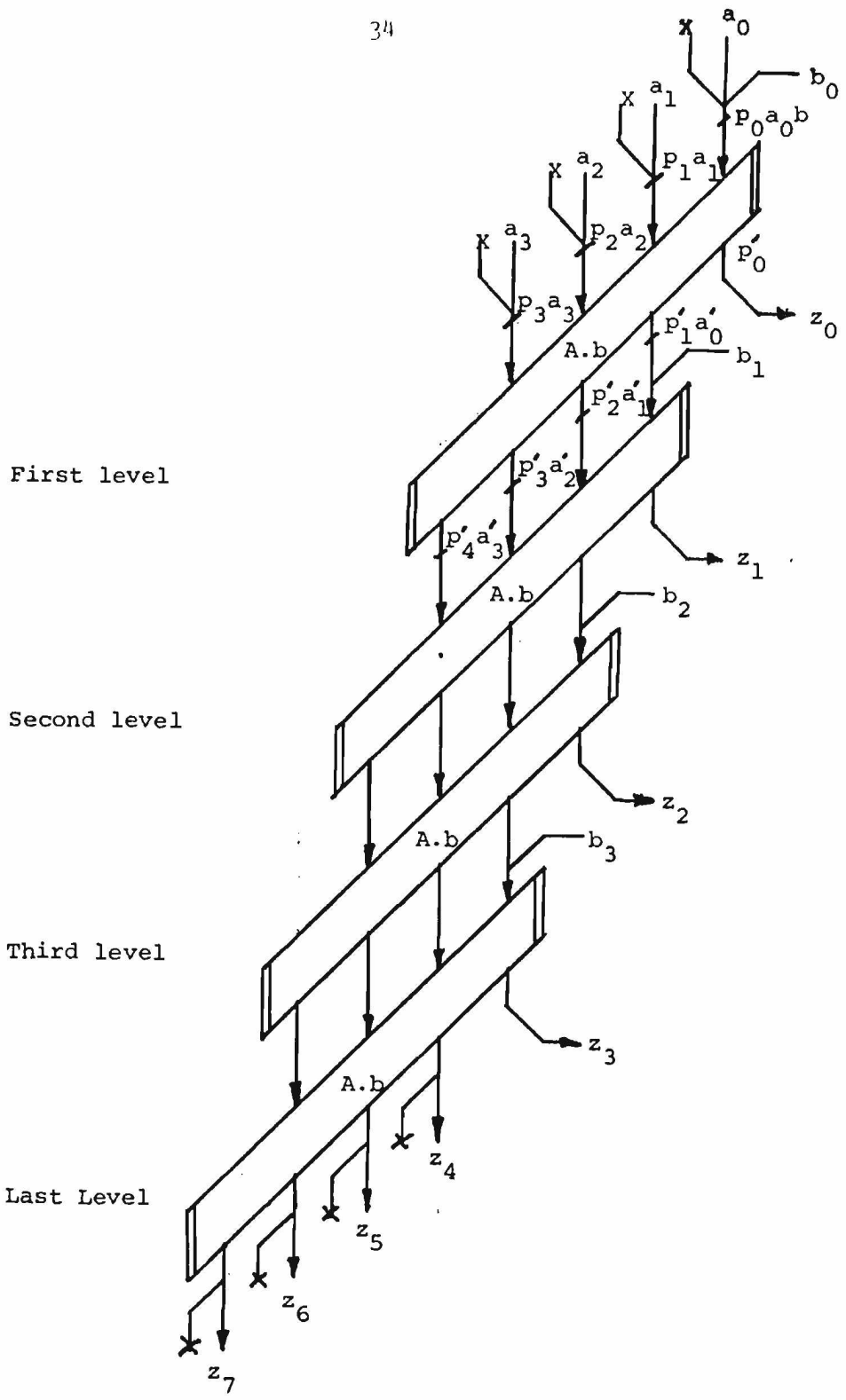


Figure 20: TNF block diagram for 4 x 4 multiplier

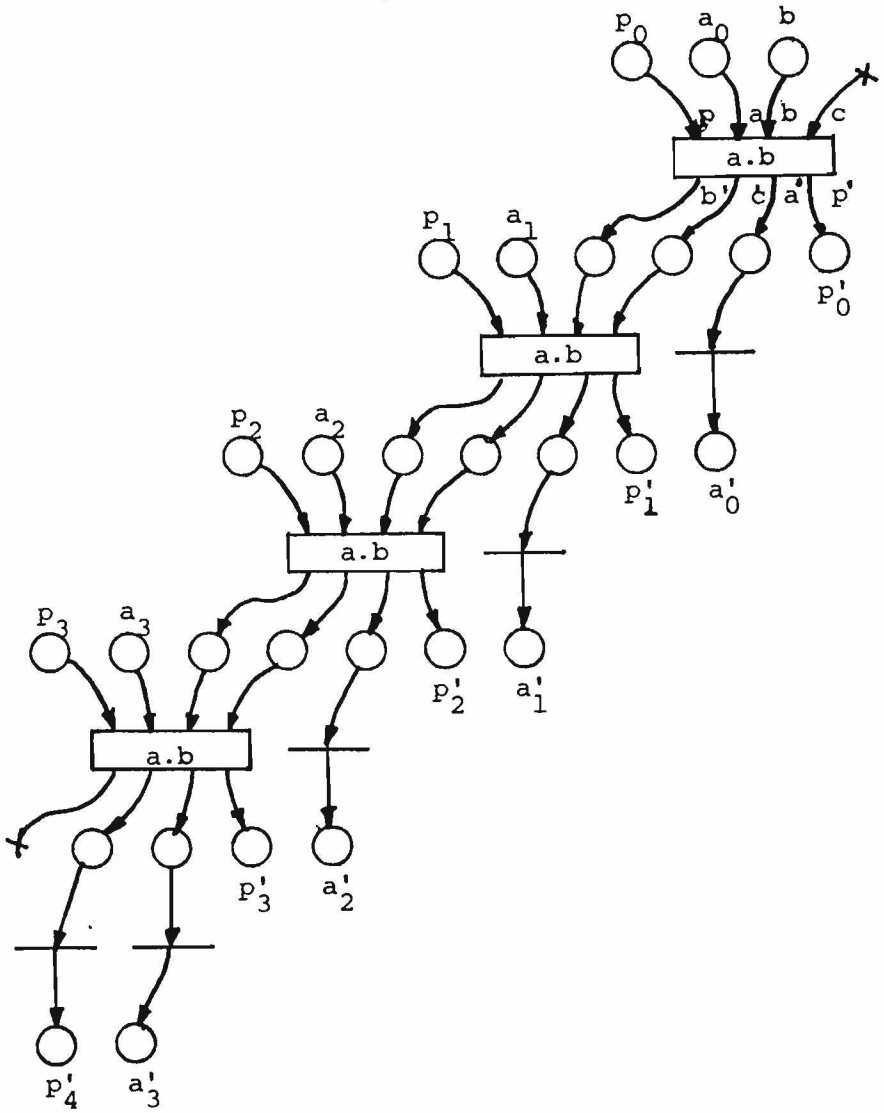


Figure 21: TNF G-net for A.b module

7.3 G-SLA implementation

The major consideration in the design of a G-SLA program for the above multiplier is the choosing of a physical layout which leads to efficient spatial arrangement of the modules. The layout presented, which might seem obvious in retrospect, required several attempts to evolve. Figure 22 gives a block diagram of the program showing how a.b macro transitions are concatenated to form A.b modules, A.b modules are concatenated to form the entire multiplier, and the interconnecting data paths. Each rectangular block represents an a.b macro transition G-SLA program, those along the periphery are special types and are labelled "a.b0L", "a.bL" and so on. The program for each type of block is given in Figure 23. These programs are not optimal;

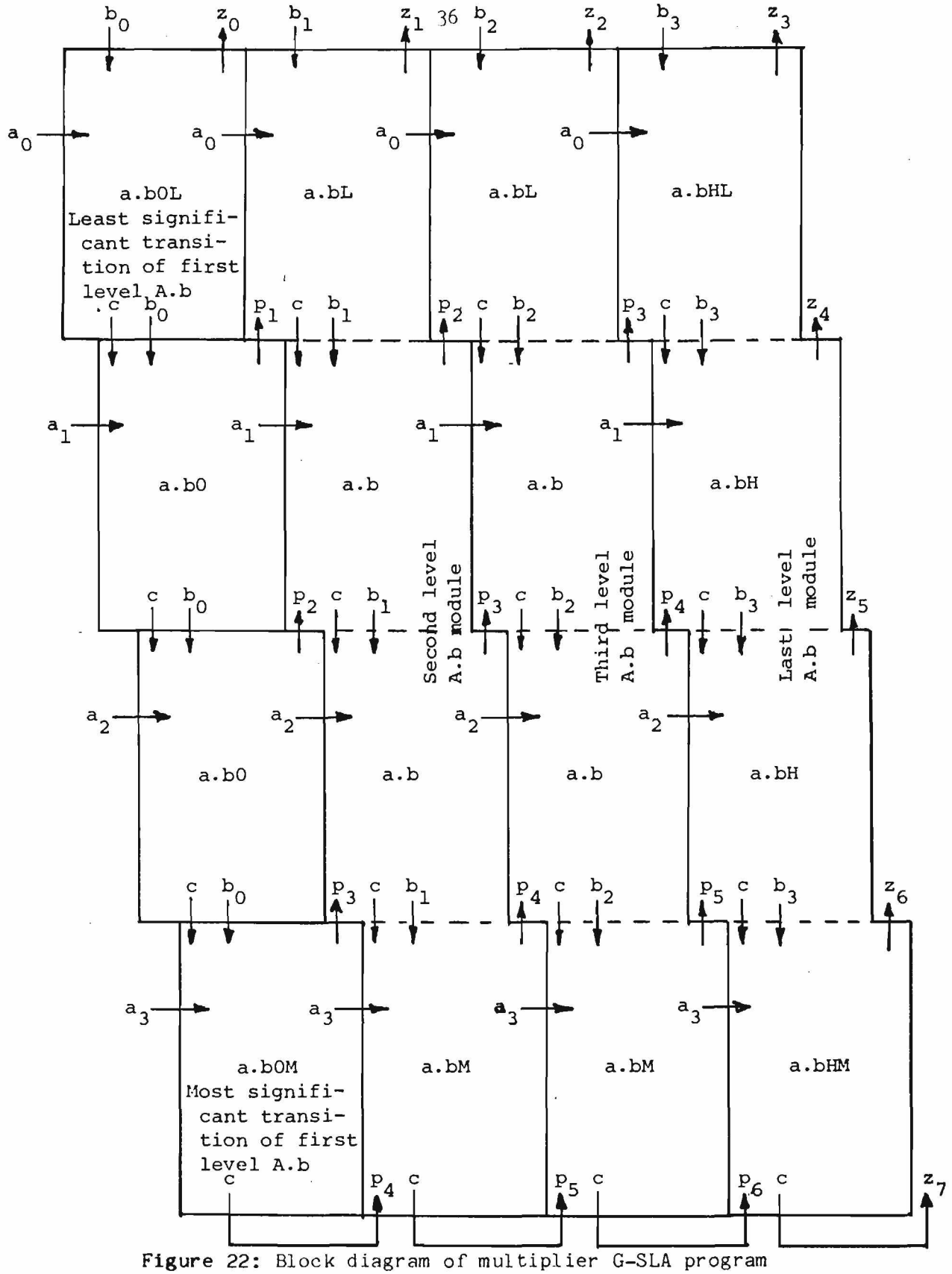


Figure 22: Block diagram of multiplier G-SLA program

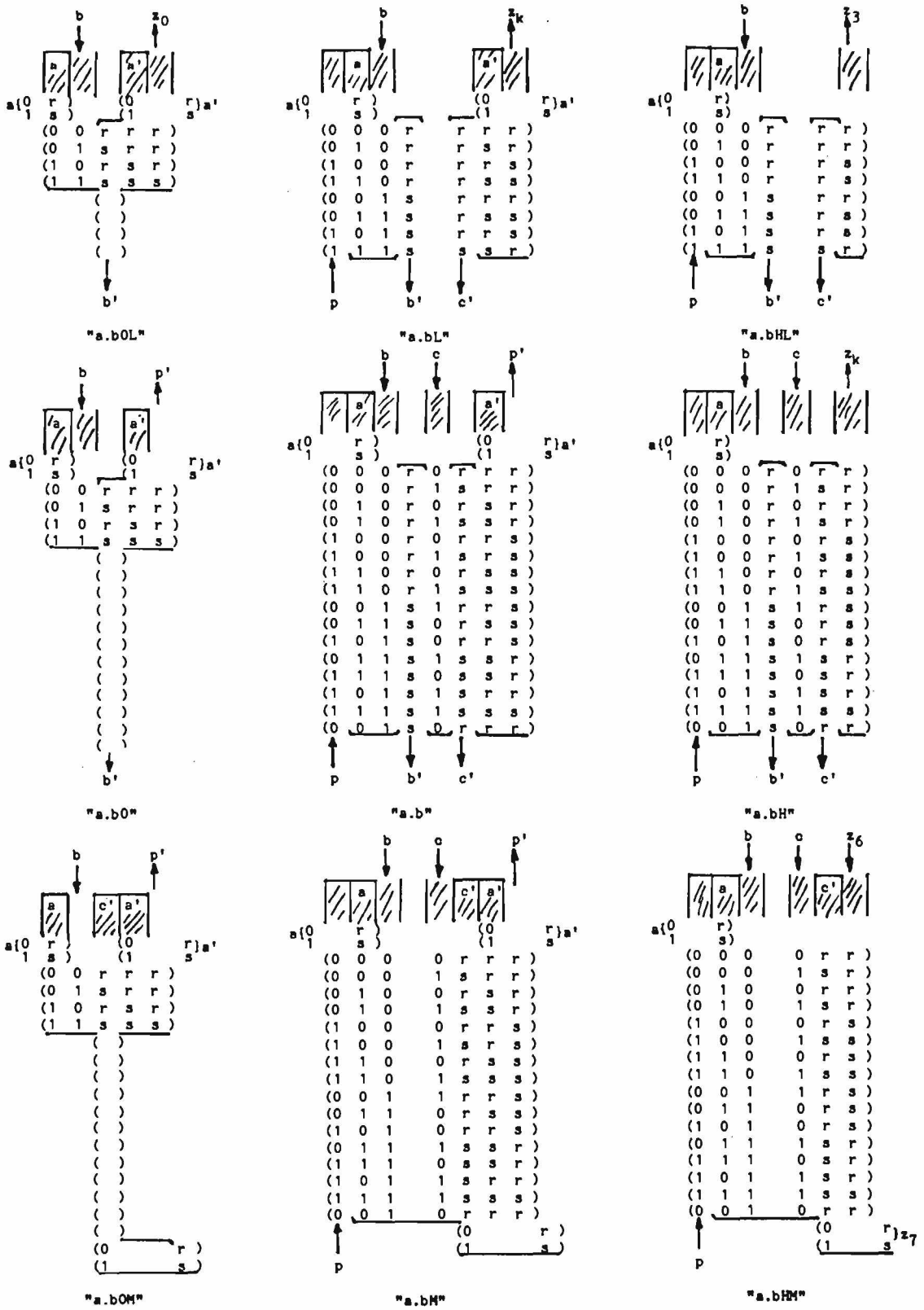


Figure 23: G-SLA programs for a.b macro transitions

there are ways in which they could be made smaller but the form presented is felt to be most easily understood.

Again note the locality and regularity of interconnections in this program. It can be easily expanded to handle any given word length and will operate with the same throughput while requiring no greater fan in or fan out of the constituent elements.

8 Summary

Bit-Driven Logic, a rather unorthodox concept which may have considerable advantages for VLSI chip design, has been presented. Strong points of circuits using the BDL style of design include:

- Good locality, which aids design and implementation.
- A finely-grained mix of storage and logic, which automatically promotes extreme pipelining leading to high throughput given suitable computation problems.
- The asynchronous, data-driven philosophy at the heart of BDL eliminates clocks and the complex timing-design problems that accompany them.

The G-Net, a graph model similar to the Petri Net, was developed for drawing and modelling BDL circuits. Using G-Nets, throughput in acyclic BDL circuits was investigated and some basic results developed:

- In a simple linear chain (like a FIFO), best throughput can be achieved if half the places are full.
- The throughput of a parallel combination of two linear chains, synchronized at begining and end, is best when the chains are of equal length.
- Using flat and skewed shims, a designer can interconnect BDL modules characterized by the properties of skew and thickness to achieve optimum throughput in an acyclic system.
- Time Normal Form is a very useful way of drawing G-Nets to illuminate throughput-limiting design deficiencies as well as skew and thickness properties of the net.

The SLA concept provides an elegant implementation vehicle for BDL. It is well-suited for implementation of BDL circuits for at least two reasons:

- Both concepts feature finely-grained mixtures of storage and logic. (The rows and columns of the SLA, and the transitions and places of the G-Net, respectively.)

- The locality of BDL is beneficial to the SLA concept. Not only might the SLA cell's internal design be simplified, but better opportunity to use split rows and columns is created.

To exploit this suitability, algorithms mapping G-Nets into standard SLA and specially-tailored G-SLA programs were developed.

A complete design for an array-organized multiplier circuit demonstrates that the BDL style, implemented by a G-SLA program, can solve meaningful design problems.

REFERENCES

[Agrawal 75] Agrawal, D. P.
Optimum Array-Like Structures for High-Speed Arithmetic.
In 3rd Symposium on Computer Arithmetic, pages 208-219. IEEE Computer Society, November, 1975.

[Davis 78] Davis, A. L.
Data Driven Nets: A Maximally Concurrent Procedural, Parallel Process Representation for Distributed Control Systems.
Technical Report UUCS-78-108, University of Utah, July, 1978.

[Dennis 74] Dennis, J. B. and Misunas, D. P.
A Computer Architecture for Highly Parallel Signal Processing.
In Proceedings of the ACM 1974 National Conference, pages 402-409. ACM, November, 1974.

[Keller 79] Keller, R. M., Lindstrom, G. and Patil, S. S.
A Loosely-Coupled Applicative Multi-Processing System.
In AFIPS Proceedings 48, pages 861-870. AFIPS, June, 1979.

[Misunas 77] Misunas, D. P.
Report on the Workshop on Data-Flow Computer and Program Organization.
Technical Memo MIT/LCS/TM-92, MIT Laboratory for Computer Science, November, 1977.
Contains a large bibliography of the field.

[Misunas 78] Misunas, D. P.
A Computer Architecture for Data-Flow Computation.
Technical Memo MIT/LCS/TM-100, MIT Laboratory for Computer Science, March, 1978.

[Patil 79] Patil, S. S. and Welch, T.
A Programmable Logic Approach to VLSI.
IEEE Transactions on Computers C-28(9):594-601, September, 1979.

[Seitz 79] Seitz, C. L.
Self-Timed VLSI Systems.
In Seitz, C. L., editor, Proccedings of Caltech Conference on Very Large Scale Integration, pages 345-356. Caltech Computer Science Department, January, 1979.