Linear Sequential Arrays: Pipelining Arithmetic Data Paths

Marianne E. Louie and Miloš D. Ercegovac
Computer Science Department
University of California, Los Angeles, CA 90024

March 3, 1994

Abstract

Arithmetic algorithms typically use broadcasting for the distribution of control data to all bit slices in the data path. Since routing delays grow as a function of fanout and routing distance, broadcasting delays grow as a function of precision in arithmetic implementations. Broadcasting in the critical path degrades performance and causes implementations of differing precisions to experience different cycle times. We present the linear sequential array (LSA) that replaces broadcasting in iterative arithmetic data paths with a pipelining scheme. With an LSA's local routing, the critical path is independent of precision. The modular design of the LSA enables precision to be extended by simply appending modules. Combinational logic hardware requirements of the LSA are the same as that of a conventional design, but the LSA generally requires more latches for delaying the transfer of data to modules farther down the pipeline. This paper discusses the LSA for general arithmetic data paths. It provides a method to convert any iterative arithmetic data path from a broadcasting approach to an LSA scheme.

1 Introduction

Although routing delays grow with increases in fanout and routing distance, large fanout broadcasting has typically been used by arithmetic algorithms for passing control data to the data path. For example, conventional digit-recurrence dividers [Hwan79, ErLa93], add-andshift iterative multipliers [Hwan79], and various on-line units [Tu90, Fern93] currently employ broadcasting. Unfortunately, broadcasting delays in the critical path degrade performance and cause arithmetic implementations of differing precisions to experience different cycle times. For high precision designs, broadcasting delays can dominate the critical path.

Broadcasting can create serious cycle time delays when designing with Field Programmable Gate Array (FPGA) technology. The reprogrammability of FPGAs results in the magnification of routing delay problems due to support of many programmable interconnection points. FPGAs, such as the Xilinx XC4010 [Xil92], provide a few special interconnection lines for global broadcasting (e.g., for clocks and reset signals), but due to their very limited interconnection to resources, these are generally insufficient for arithmetic algorithms.

One approach to alleviating the performance degradation due to broadcasting has been prediction of the control data [ErLa85, LoEr92, LoEr93a, LoEr93c, MoCi94]. With prediction, broadcasting of the current iteration's control logic proceeds in parallel with computation of the next iteration's control logic. For very small precisions, broadcasting and the generally simple logic of the data path produce a smaller combined delay than that of the control logic. For these precisions, broadcasting does not appear in the critical path. However, for precisions greater than 25 bits, broadcasting delays can again lie in the critical path [LoEr93a].

The prediction concept can be extended to make the output of the current iteration dependent upon control logic computed several iterations earlier [ShVu93]. This pipelining approach in control logic enables high precision designs without having broadcasting in the critical path. With this technique the necessary amount of precomputing/pipelining depends upon the precision of the design, thus causing the control logic area to grow as a function of precision.

This paper generalizes the Linear Sequential Array (LSA) approach [Erce84, LoEr93a, LoEr93b] which replaces broadcasting in iterative arithmetic with a pipelined scheme (Fig. 1). The data path is developed as stages of uniform modules with each module extending the precision of the design. Modules are simply appended as needed to create a data path of the desired precision. For FPGAs the modular approach enables rapid construction of custom precision designs. Interconnections between modules are local with small fixed fanout to maintain low routing delays. Cycle time is therefore independent of precision without affecting the control logic. When the LSA is combined with prediction of the control logic, the data path will often have a smaller delay than the control logic and will not appear in the critical path [LoEr93a, LoEr93b]. The LSA modules also feature comparable combinational logic requirements as a conventional broadcasting scheme, but the LSA utilizes more latches for transferring data in the pipeline to modules operating at delayed algorithm steps.

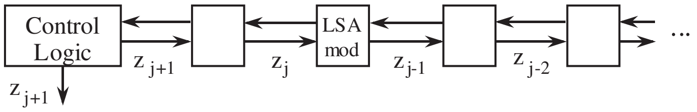


Figure 1: The Linear Sequential Array Organization

LSA pipelining is made possible due to the uniform and localized data dependencies that characterize arithmetic data paths. Converting a data path from broadcasting to an LSA design requires examining the bit level dependencies as a function of time. Section 3 describes the dependency types in the data path and their mapping to the LSA. Theorems deriving the theoretical basis for the LSA design are given in Appendix A. Section 4 presents the general conversion algorithm and examples. A discussion of LSA implementation characteristics as compared to a conventional broadcasting technique is given in section 5.

2 Notation and Definitions

The following notation and definitions are utilized throughout the text. Additional notation are defined as needed. In general, the digit labels on bit vectors have the format d_i where i indicates that the digit is of weight 2^i.

Iterative arithmetic algorithms can be summarized as having the format $w[j+1] = f(w[j], z_{j+1})$. The following notation describes this format:

$w[j]$	The set of iteration variables with values at step $j+1$
z_{j+1}	control data for step $j+1$
v^k	variable (bit vector) in w where $w = \{v^1, v^2, \ldots\}$.
v_i^k	bit of weight 2^i in the bit vector v^k where each bit in a given vector is based on the same boolean expression

To simplify the description of the position/weights of modules and bits relative to another module or bit, we make the following definitions.

upstream module A module that precedes a reference module in the pipeline. An upstream module thus operates on step $j+i$, $i \geq 1$ when the reference module operates on step j.

upstream bit A bit that precedes a reference bit in vector position relative to the direction of the data path pipeline. For example, given the vector x, if the control data is pipelined from higher weighted bits to lower weighted bits, the bits x_i, $i > 5$ are upstream from the bit x_5. Alternatively, if the control data is pipelined from lower weighted bits to higher weighted bits, the bits x_i, $i < 5$ are upstream from the bit x_5.

downstream module A module that succeeds a reference module in the pipeline. A downstream module thus operates on step $j - i$, $i \geq 1$ when the reference module operates on step j.

downstream bit A bit that succeeds a reference bit in vector position relative to the direction of the data path pipeline. For example, given the vector x, if the control data is pipelined from higher weighted bits to lower weighted bits, the bits x_i, $i < 5$ are downstream from the bit x_5. Alternatively, if the control data is pipelined from lower weighted bits to higher weighted bits, the bits x_i, $i > 5$ are downstream from the bit x_5.

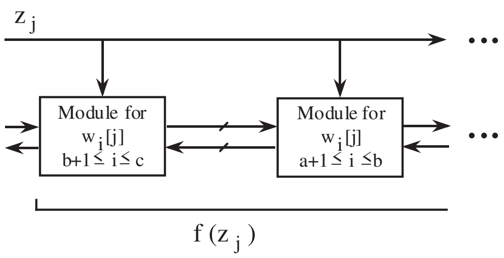
Lastly, arrays are used in the definition of the conversion algorithm. We define A_{ij} as the cell in the row labeled i and column labeled j in the vector A.

3 The Data Path

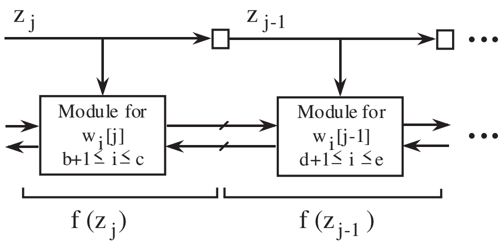
Throughout this section, the LSA modules are designed to directly satisfy the data path equation of $w[j+1] = f(w[j], z_{j+1})$ where z_{j+1} is the control value for step $j+1$.

3.1 Conventional versus LSA Modules: Overview

In both conventional broadcasting and the LSA approach, the data path for iterative arithmetic implementations can be described as a modular design. Figure 2 shows a comparison of conventional and LSA modules. Like conventional modules, each LSA module computes a unique set of weighted bits in the vectors of $w[j+1]$. Adjacent modules form adjacent bits according to the equation $w[j+1] = f(w[j], z_{j+1})$. Unlike conventional modules, LSA modules operate on different algorithm steps relative to other LSA modules. For the LSA pipelining, adjacent modules operate on adjacent steps.



(a) Conventional modules



(b) LSA modules

Figure 2: Conventional vs. LSA Modules

3.2 Data Dependencies Between LSA Modules

As in conventional modules, an LSA module may require data from other modules. Due to the step time differences between LSA modules, data can be required from modules operating at either a later time step $(j + T)$ or an earlier time step than the requesting module. This subsection shows the effects of both dependency types on the LSA design.

For the first dependency type (Fig. 3), a simple latching (delay) pipeline allows the correct transfer of data between modules. In this dependency, we are given that module A operating on step $j+1$ requires data of step j from any upstream module B. Module B, meanwhile, is currently operating on step $j + i$, $i > 1$ and would have computed its step j output i cycles earlier. By implementing a pipeline of i latches, the desired data is stored until needed.

For the second dependency as shown by Theorem 1 in Appendix A, module A operating at step j may depend on downstream data from only its neighboring module, C, operating on step $j-1$. In this dependency, (Fig. 4), module A must wait for data that a downstream module C has not yet computed. To prevent this second dependency type from causing iterative dependencies that ripple through all downstream modules, we require that step

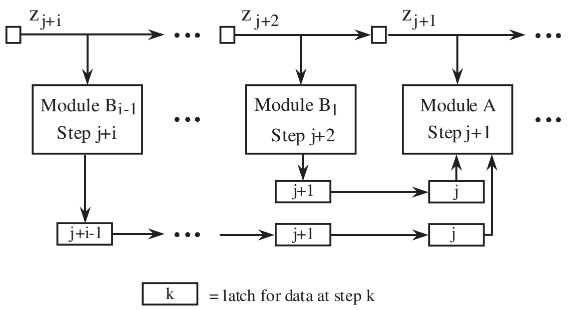


Figure 3: Transfer of Data to a Downstream Module

j data which is first calculated and subsequently transmitted to an upstream module be computable directly from currently latched values of step $j-1$. Theorem 2 in Appendix A demonstrates that this requirement limits the combinational logic delay of an LSA to twice that of a conventional module. Figure 5 shows the resulting two substages within an LSA module.

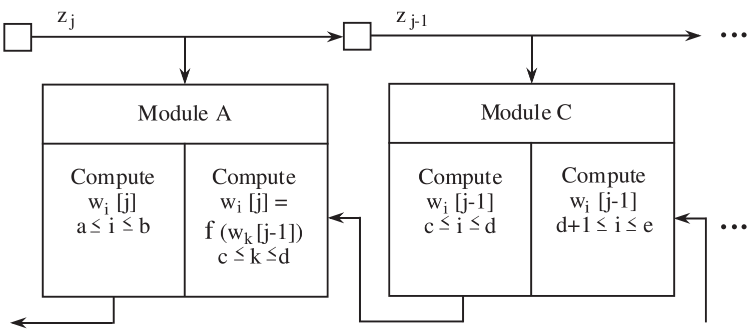


Figure 4: Transfer of Data to an Upstream Module

4 Conversion to LSA Modules

This section presents a general method to convert conventional modules to LSA modules. A brief overview of the method's approach is presented first, followed by the algorithm's details and examples.

By definition, an LSA module computes a unique set of weighted bits for the step j. We

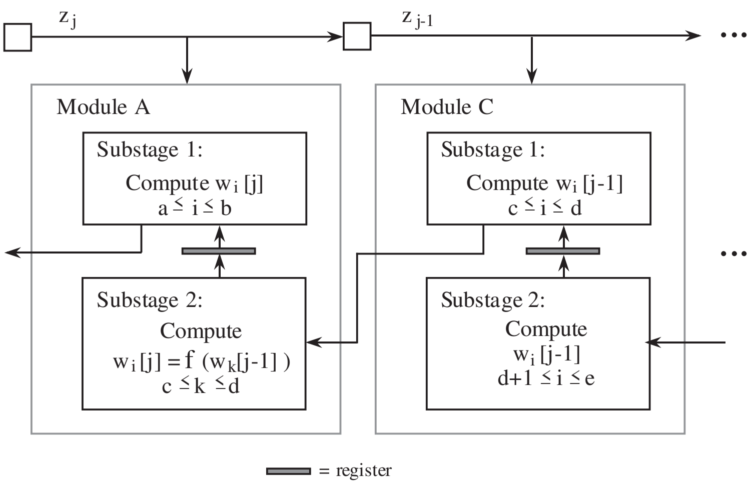


Figure 5: The Two LSA Module Substages

begin LSA construction with the minimum definition that one bit of the same weight from each vector in $w[j]$ is computed and latched within the same LSA module. The constraints in receiving data from a downstream module then expands the number of bits computed by an LSA module. The objective in creating the module is the maximal delay of computations until necessary. As shown by Theorems 2 and 4 of Appendix A, a maximum of two substages are created before the computation pattern repeats. This repetition delineates a new module. After the downstream dependencies are satisfied, latches are added to fulfill upstream dependencies. The algorithm for conversion is given below.

4.1 Conversion Algorithm

Let $w = \{v^1, v^2, \ldots, v^M\}$ = the set of all bit vectors in the arithmetic algorithm's data path where M is the total number of vectors in the data path. Let $v_b^k[j]$ represent a bit of weight 2^b in v^k at step j. Define three matrices, $S1$, $S2$, and $S3$, each having M rows. $S1$ and $S2$ represent the first and second substages, respectively, in an LSA module. $S3$ maintains a record of the number of latches (delays) required for passing data between different cycles. Each matrix row represents one vector, v^k. The matrix columns represent bit slices in the vectors. To represent the relative bit weights, the matrix columns are numbered increasingly from right to left. Bits in the matrices are denoted as $Sx_{k,c}$ where (1) x is either 1, 2, or 3 corresponding to matrix $S1$, $S2$, or $S3$, (2) k refers to the vector v^k, and (3) c denotes the bit of v^k of weight 2^c.

In the conversion algorithm, cells in matrices $S1$ and $S2$ are gradually marked to show the stage of maximum delay in which a particular bit must be available. Only downstream bits relative to the initial bit slice v_b^k are marked due to the constraints of receiving data from a downstream module. Once these constraints are satisfied, the remaining bits of the module are guaranteed by Theorem 3 of Appendix A to have available input data. The resulting marked bit slices in matrices $S1$, $S2$, and $S3$ define a single LSA module. If the computation of bit slice does not have dependencies on downstream bits, the conversion algorithm will not mark any bits in $S1$ and $S2$. In this case, the minimum width of an LSA module is one bit and only one substage is needed. Latches, though, continue to be required for the passing of data to modules operating on delayed algorithm steps. The steps of the conversion algorithm are given next:

Initialization. Unmark all matrix cells (cell value = 0):

For all k, c
$S1_{k,c} = S2_{k,c} = S3_{k,c} = 0$

Step 1. Satisfy the downstream input dependencies of the initial latched bits:

For all k, k' in $\{1, \ldots, M\}$
if $v_b^{k'}[j] = f(v_{b+i}^k[j-1])$ then $S1_{k,b+i} = 1$
where $i \leq -1$ if pipelining progresses from higher to lower weighted bits and $i \geq 1$ otherwise

Step 2. Mark dependencies on stage 1 by bits upstream from the initial latched bits.

Let $a = -1$ if pipelining proceeds from higher to lower weighted bits;
$a = 1$ otherwise
For each k in $\{1, \ldots, M\}$
{ Let L = index of farthest marked column in $S1$ from b for row k
For $c = b + a$ to L
$S1_{k,c} = 1$
}

Step 3. Satisfy the downstream input dependencies of marked bits in substage 1.

For all k, k' in $\{1, \ldots, M\}$ and $S1_{k',b+i} \neq 0$
if $v_{b+i}^{k'}[j-1] = f(v_{b+i+h}^k[j-2])$ then $S2_{k,b+i+h} = 1$
where $h \leq 0$, $i \leq -1$ if pipelining proceeds from higher to lower weighted bits and $h \geq 0$, $i \geq 1$ otherwise

Step 4. Since each bit is computed only once per cycle, unmark cells in $S2$ that are marked in both $S1$ and $S2$; also mark the cell's latch counter in $S3$.

For all k, c
if $S2_{k,c} = S1_{k,c} = 1$ then $S2_{k,c} = 0$ and $S3_{k,c} = 1$

Step 5. Define the remaining bits to be computed by the LSA module. Specify a latch if the remaining bits are dependent on downstream bits in stage 1.

Let L = index of the farthest marked column in $S2$ from b
Let $a = -1$ if pipelining proceeds from higher to lower weighted bits;
$a = 1$ otherwise
For all k, k' in $\{1, \ldots, M\}$ and all i
For $c = b + a$ to L
if $S1_{k',c} = S2_{k',c} = 0$
then { $S2_{k',c} = 1$
if $((v_c^{k'}[j] = f(v_{c+a|i|}^k[j-1]))$ and $(S1_{k,c+a|i|} = 1))$
then $S3_{k,c+a|i|} = 1$ }

Step 6. Mark latches for all bits in the second step.

For all k, c
if $S2_{k,c} = 1$ then $S3_{k,c} = 1$

Step 7. Satisfy the upstream input dependencies and record the maximum number of required latches.

Let L = index of farthest marked column in $S2$ from b
Let $a = -1$ if pipelining proceeds from higher to lower weighted bits;
$a = 1$ otherwise
For all k, k' in $\{1, \ldots, M\}$ and all i
For $c = 0$ to $(b - L + a)$
{ if $v_{L+c}^{k'}[j] = f(v_{L+c+i}^k[j-1])$ where
$i \geq 0$ if pipelining proceeds from higher to lower weights,
and $i \leq 0$ otherwise
Let $e = \text{mod}\ (|c+i|, |L-b|)$
Let $d = \lfloor |c+i| / |L-b| \rfloor + 1$
$S3_{k,L-ae} = \max\ (d, S3_{k,L-ae})$
}

The marked cells provide a template for each LSA module where the number of columns spanned by nonzero cells in $S1$ and $S2$ defines the module's incremental precision. Marked (nonzero) cells in matrices $S1$ and $S2$ identify in which stage, respectively, that each bit must be computed. The first substage generates data that is always transferred to an upstream module but can also be stored for later use by downstream modules or the module that generated it. Likewise, the second substage generates data that is stored for later use only by downstream modules or the module that generated it. The combinational logic of the combined substages is identical to that used by a conventional module. Since no additional combinational logic is implemented in an LSA design, the difference between LSA hardware and conventional modules consists of the latching schemes and interconnections.

4.2 Examples

$$v_i^1[j+1] = f_1(v_{i-1}^1[j], v_{i-1}^2[j], v_{i-1}^3[j])$$
$$v_i^2[j+1] = f_2(v_{i-2}^1[j], v_{i-2}^2[j], v_{i-2}^3[j])$$
$$v_i^3[j+1] = f_3(z_{j+1})$$

(a) Data path equations

$$S3\begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \quad S2\begin{bmatrix} 0&1&1&1 \\ 0&1&1&1 \\ 0&1&1&1 \end{bmatrix} \quad S1\begin{bmatrix} 1&1&0&0 \\ 1&1&0&0 \\ 1&1&0&0 \end{bmatrix}$$

matrix column b-1   matrix column b-4

(b) After steps 1 & 2

$$S3\begin{bmatrix} 0&1&0&0 \\ 0&1&0&0 \\ 0&1&0&0 \end{bmatrix} \quad S2\begin{bmatrix} 0&0&1&1 \\ 0&0&1&1 \\ 0&0&1&1 \end{bmatrix} \quad S1\begin{bmatrix} 1&1&0&0 \\ 1&1&0&0 \\ 1&1&0&0 \end{bmatrix}$$

(c) After step 3 (same after step 4)

$$\begin{array}{cl} S3\begin{bmatrix} 0&1&1&1 \\ 0&1&1&1 \\ 0&1&1&1 \end{bmatrix} & \begin{array}{l} \leftarrow v^1 \\ \leftarrow v^2 \\ \leftarrow v^3 \end{array} \end{array} \quad S2\begin{bmatrix} 0&0&1&1 \\ 0&0&1&1 \\ 0&0&1&1 \end{bmatrix} \quad S1\begin{bmatrix} 1&1&0&0 \\ 1&1&0&0 \\ 1&1&0&0 \end{bmatrix}$$

(d) After step 5 (same after step 6)

Figure 6: Development of an LSA Module for SRT Division

As an example, an LSA design is created for SRT division. The SRT data path consists of three vectors, v^1 (sum), v^2 (carry), and v^3 (multiple of the divisor) where:

$$v_i^1[j+1] = f_1(v_{i-1}^1[j], v_{i-1}^2[j], v_{i-1}^3[j])$$

$$v_i^2[j+1] = f_2(v_{i-2}^1[j], v_{i-2}^2[j], v_{i-2}^3[j])$$

$v_i^3[j+1] = f_3(z_{j+1})$

The control data, z, is pipelined from higher to lower weighted bits. Only downstream and control logic dependencies are present in this example. Figures 6 and 7 show the output matrices and LSA design after applying the conversion algorithm. In the LSA design of Figure 7, the vectors are labeled s (v^1), c (v^2), and D (v^3) where the superscripts denote the algorithm step from which each bit is derived.

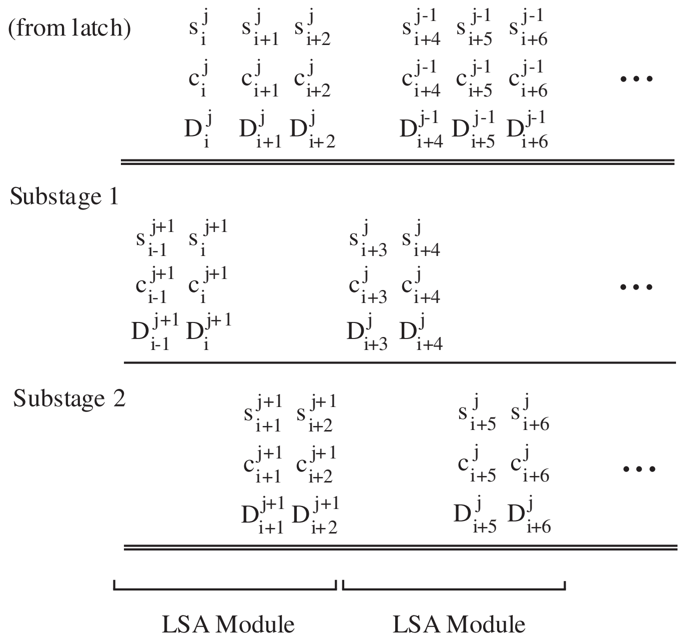


Figure 7: LSA Design for the SRT Division Example

A second example (Fig. 8) demonstrates all steps in the conversion. Again, the control data is passed from higher to lower weighted bits. In Figure 8, the data path vectors are defined as:

$$v_i^1[j+1] = f_4(v_{i-1}^2[j], v_{i-2}^2[j], v_{i-3}^2[j], v_{i+3}^3[j])$$

$$v_i^2[j+1] = f_5(v_{i-1}^1[j], v_{i-2}^1[j], v_{i+8}^3[j])$$

$$v_i^3[j+1] = f_6(v_i^2[j])$$

$$v_i^1[j+1] = f_4(v_{i-1}^2[j], v_{i-2}^2[j], v_{i-3}^2[j], v_{i+3}^3[j])$$
$$v_i^2[j+1] = f_5(v_{i-1}^1[j], v_{i-2}^1[j], v_{i+8}^3[j])$$
$$v_i^3[j+1] = f_6(v_i^2[j])$$

(a) Data path equations

$$S3\begin{bmatrix} 00000 \\ 00000 \\ 00000 \end{bmatrix} \quad S2\begin{bmatrix} 01111 \\ 01111 \\ 00000 \end{bmatrix} \quad S1\begin{bmatrix} 11000 \\ 11100 \\ 00000 \end{bmatrix}$$

matrix column b-1 matrix column b-5

(b) After steps 1 & 2

$$S3\begin{bmatrix} 01000 \\ 01100 \\ 00000 \end{bmatrix} \quad S2\begin{bmatrix} 00111 \\ 00011 \\ 00000 \end{bmatrix} \quad S1\begin{bmatrix} 11000 \\ 11100 \\ 00000 \end{bmatrix}$$

(c) After step 3

$$S3\begin{bmatrix} 01000 \\ 11100 \\ 00000 \end{bmatrix} \quad S2\begin{bmatrix} 00111 \\ 00011 \\ 11111 \end{bmatrix} \quad S1\begin{bmatrix} 11000 \\ 11100 \\ 00000 \end{bmatrix}$$

(d) After step 4

$$S3\begin{bmatrix} 01111 \\ 11111 \\ 11111 \end{bmatrix} \quad S2\begin{bmatrix} 00111 \\ 00011 \\ 11111 \end{bmatrix} \quad S1\begin{bmatrix} 11000 \\ 11100 \\ 00000 \end{bmatrix}$$

(e) After step 5

$$S3\begin{bmatrix} 01111 \\ 11111 \\ 22333 \end{bmatrix}\begin{array}{l} \leftarrow v^1 \\ \leftarrow v^2 \\ \leftarrow v^3 \end{array} \quad S2\begin{bmatrix} 00111 \\ 00011 \\ 11111 \end{bmatrix} \quad S1\begin{bmatrix} 11000 \\ 11100 \\ 00000 \end{bmatrix}$$

(f) After step 6

Figure 8: Conversion Example

4.3 Multi-output Operators

The conversion algorithm creates basic LSA modules for any given set of arithmetic data path vectors. The algorithm is simple, but to maintain this feature it does not identify multi-output operators (multiple functions using the same group of inputs, e.g. full adder) and group them within a single LSA module. As a result, if the multiple output bits are of different weights, they may be placed on different modules. Latches are then needed to store these inputs for use during two different cycles. In cases where these latches are needed only for computation of the delayed multi-output bits, the rejoining of these outputs can reduce the required number of latches and conserve area. Rejoining is achieved by computing the delayed outputs at the same time as the earliest computed member of the multi-output group. Latches must be placed on the moved output bits to ensure their availability at their previously computed time, but generally the number of latched output bits is fewer than that needed before the rejoining.

An example of separated multi-output bits in the SRT LSA module of Figure 7 are the pairs of sum and carry bits $(S_{i+3}^j, c_{i+2}^{j+1})$ and $(s_{i+7}^{j-1}, c_{i+6}^j)$. The SRT LSA design after rejoining these bits is shown in Figure 9 where the superscripts denotes the algorithm step of each bit. For the SRT example, rejoining the separated bits saves two latches per module.

(from latch)			s^{j}_{i+1}	s^{j}_{i+2}	s^{j}_{i+3}		s^{j-1}_{i+5}	s^{j-1}_{i+6}	s^{j-1}_{i+7}	
			c^{j}_{i+1}	c^{j}_{i+2}			c^{j-1}_{i+5}	c^{j-1}_{i+6}		...
			D^{j}_{i+1}	D^{j}_{i+2}			D^{j-1}_{i+5}	D^{j-1}_{i+6}		
Substage 1										
		s^{j+1}_{i}				s^{j}_{i+4}				
	c^{j+1}_{i-1}	c^{j+1}_{i}			c^{j}_{i+3}	c^{j}_{i+4}				...
	D^{j+1}_{i-1}	D^{j+1}_{i}			D^{j}_{i+3}	D^{j}_{i+4}				
Substage 2 (to latch)			s^{j+1}_{i+1}	s^{j+1}_{i+2}	s^{j+1}_{i+3}		s^{j}_{i+5}	s^{j}_{i+6}	s^{j}_{i+7}	
			c^{j+1}_{i+1}	c^{j+1}_{i+2}			c^{j}_{i+5}	c^{j}_{i+6}		...
			D^{j+1}_{i+1}	D^{j+1}_{i+2}			D^{j}_{i+5}	D^{j}_{i+6}		

Figure 9: The LSA for SRT Division after Rejoining Full Adder Outputs

5 A Comparison of LSA and Conventional Modules

Since the conversion algorithm assumes use of the same boolean expressions as the conventional module, the combinational logic requirements of the LSA is identical to that in a conventional design. Some opportunities for logic minimization may occur where the two substages interface neighboring modules. A logic minimized LSA module may then require less combinational logic than its corresponding conventional module.

In support of the sequential logic, the LSA generally requires more latches than a conventional module. Additional latches over conventional requirements are necessary for both pipelining the control values and for saving data to be transferred to downstream bits or downstream modules. If each bit in the data path vectors is independent of upstream bits in the data path, the latter additional latches are unnecessary. In this case, the LSA may use fewer latches than conventional modules because the output from the first substage is immediately used by an adjoining module and does not need to be stored.

With respect to delay in the data path, the two LSA substages double the combinational logic delay over that of conventional modules. However, the LSA does not experience the additional broadcasting delays of conventional designs. Unlike broadcasting delays in arithmetic which grow unboundedly as a function of precision, the delay of the LSA is independent of precision.

In many iterative arithmetic algorithms, each cycle can be modeled (Fig. 10) as (1) computation of the control logic, (2) distribution of the control logic to the data path, followed by (3) updating of the data path. When using broadcasting with this model, the entire broadcasting delay appears in the critical path. When large precision designs use an LSA scheme instead, the delay of the extra substage is smaller than a broadcasting delay. However this overhead is also undesirable when in the critical path. The use of prediction schemes for the control logic [ErLa85, LoEr92, LoEr93a, LoEr93c, MoCi94] helps to remove the LSA overhead from the critical path. Control logic prediction enables the distribution of control values and updating of the data path to occur in parallel with computation of the next iteration's control. A simple method for creating a prediction scheme is achieved by repositioning the registers [LoEr93c]. The resulting iterative arithmetic model using both the LSA and prediction appears in Figure 11. Since control logic is generally more complex than data path logic, the combination of (1) parallel computation of the control data and data path through prediction and (2) limited data path delay through the LSA provides opportunities in which the data path can be completely removed from the critical path.

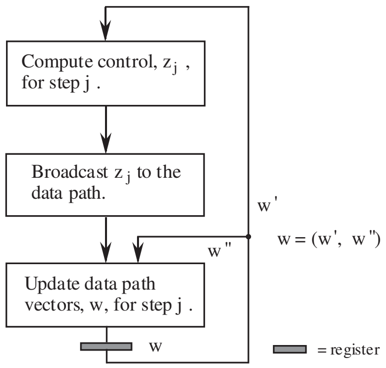


Figure 10: Arithmetic Model with an LSA and Prediction

The LSA technique is especially useful for implementations with Field Programmable Gate Array (FPGA) technology where broadcasting delays can equal a logic stage delay at precisions as low as 25 bits [LoEr93a]. FPGAs have an especially large growth rate for interconnection delays as a function of fanout and routing distance due to support of the FPGA's many programmable interconnection points. By using the LSA, interconnection delays along the data path remain small local routing delays. As demonstrated with an SRT divider [LoEr93a] and a radix-2 square rooter [LoEr93b], the combination of the LSA with control logic prediction enables the data path to be removed from the critical path of these designs for any precision.

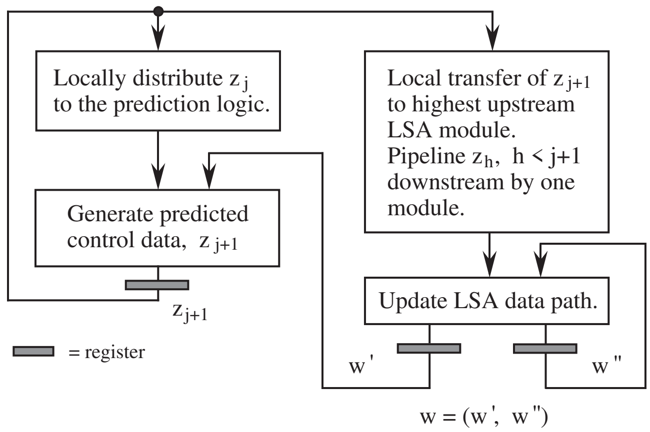


Figure 11: Arithmetic Model with an LSA and Prediction

6 Conclusion

Broadcasting delays in the critical path degrade the performance of arithmetic implementations. Since routing delays grow with increasing fanout and routing distance, broadcasting delays can dominate the critical path in high precision designs. We present the linear sequential array which enables the cycle time of arithmetic implementations to be independent of precision. By using our conversion algorithm, data paths that currently rely on broadcasting are easily changed to a linear sequential array. A converted data path will feature local interconnects and a simple modular design so that precision is expanded by simply appending LSA modules. The combinational logic delay of an LSA data path is limited to twice that of a conventional design. For many arithmetic algorithms where the data path consists primarily of a carry-save adder, the LSA delay is small. As a result, the LSA in combination with prediction of the control data may successfully remove the data path from the critical path for any precision.

For designs with FPGAs, the LSA provides a simple solution to the problem of excessive routing delays in arithmetic designs. When combined with control logic prediction, this approach has been demonstrated to eliminate the data path from the critical path [LoEr93a, LoEr93b]. An LSA's modular approach provides simple and rapid construction of custom precision arithmetic designs in FPGAs. Thus, in cases where software has typically performed the very high precision computations due to a lack of high precision arithmetic chips, cost effective hardware can be created [LoEr93b, BuWa89].

Appendix A

The following theorems and corollary derive the theoretical basis for the LSA conversion algorithm. Theorem 1 demonstrates that an LSA module successfully generates its output even though other modules are operating on different algorithm steps. Theorem 2 then shows that the maximum combinational logic delay of the data path when using LSA pipelining is limited to twice the combination logic delay (no broadcasting delays) of a conventional design. For the LSA conversion algorithm, Theorem 3 and Corollary 1 state that an LSA module can be created by satisfying the input dependencies of one bit slice of the data path vectors. Finally, Theorem 4 shows that the creation of one LSA module is sufficient to serve as a template for other modules in the LSA design.

For each of the following theorems, we are given that:

1. At each cycle, an LSA module receives the control logic for and computes the output of only one algorithm step.
2. Adjacent LSA modules operate on adjacent algorithm steps.
3. Data transmitted from a module at step j to a module operating on step $j+1$ must be directly computable from data of step $j-1$ that is already latched and readily available.
4. The LSA directly implements the data path equation of $w[j+1] = f(w[j], z_{j+1})$.

Theorem 1 *An LSA module operating on step j may receive data only from modules operating on step $j+i$ where $i \geq -1$.*

Proof: Define module A to be operating on step j while module B operates on step $j+i$. Let module A require data of step $j-1$ from module B.

case (1) $i \geq 0$: From the first given, if module B is currently operating on step $j+i$, module B must have already computed its output for step $j+i-1$. Therefore for $(i \geq 0)$, module B has completed step $j-1$ and the data is readily available through latching.

case (2) $i = -1$: Since module B has not yet completed step $j-1$, module A must wait for this data. As specified by the first given, module B, which is currently operating on step $j-1$, completes the step within the cycle. Since module A has not yet completed its algorithm step, according to the first given module A must then receive this data and complete its operation on step j within the current cycle.

case (3) $i < -1$: For $i < -1$, module B is operating at most on step $j-2$. Since only one algorithm step can be computed per cycle, module B cannot compute the output of step $j-1$ in the given cycle. Therefore, since module A computes $w[j] = f(w[j-1], z_j)$ module A cannot receive input from module B when $i < -1$.

Therefore, from cases (1), (2), and (3) the statement of Theorem 1 is demonstrated to be true.

Theorem 2 *The logic delay of an LSA module is at most twice that of computing* $w[j] = f(w[j-1], z_j)$.

Proof: From Theorem 1, LSA module A operating on step j may receive data from modules operating on (1) step $j+i$, $i \geq 0$ or (2) step $j-1$. If module A requires data only from modules satisfying case 1, the data is readily available from latches; for this case, module A has only the delay of computing $w[j] = f(w[j-1], z_j)$. If, however, module A also requires data from module B operating on step $j-1$, module A must wait for the computation of the data it needs from $w[j-1] = f(w[j-2], z_{j-1})$. From the third given, the inputs to compute this data is readily available. Thus, the logic expression for the output of module A is summarized as $w[j] = f((w[j-1], f(w[j-2], z_{j-1})), z_j)$ which is twice the combinational logic delay of computing the general expression $w[t] = f(w[t-1], z_t)$ where $w[t-1]$ and z_t are readily available.

Theorem 3 *When the input data to compute* $v_i^k[j]$ *from step* $j-1$ *are available, the inputs to compute* $v_h^k[j]$ *in the same LSA module are also available where* $v_h^k[j]$ *are upstream bits from* $v_i^k[j]$.

Proof: Let $v_h^k[j]$ and $v_i^k[j]$ be generated by module m. By definition, the bits of a data path vector are computed with the same boolean expression, but with skewed inputs such that $v_{i+s}^k[j] = f(v_{i_1+s}^q[j-1], v_{i_2+s}^{q_1}[j-1], \ldots)$. Since $v_h^k[j] = v_{i+s}^k[j]$ is upstream from the bit $v_i^k[j]$, all inputs for computing $v_h^k[j]$ are either upstream from or overlap the inputs for computing $v_i^k[j]$. Overlaps can occur when $v_i^k[j]$ is a function of contiguous bits in the same vector. Since inputs are available for $v_i^k[j]$, overlapping inputs are also available for $v_h^k[j]$. Regarding the remaining inputs for $v_h^k[j]$, if the inputs are computed by module m, they were computed during the previous cycle. Similarly, if the inputs are generated by a module upstream from m, they were also derived in earlier cycles. In both cases the inputs are available through latches (delays). For the case that the inputs are from a downstream module, from the third given the data needed to compute the inputs transferred to module m must be currently available in latches. Since Theorem 3 states that all inputs to compute $v_i^k[j]$ are available, the downstream data for $v_h^k[j]$ which are computed by the same boolean expressions as those for $v_i^k[j]$ must also have been generated and are available.

Corollary 1 *If an LSA module design is based on satisfying the input requirements of the most downstream bit slice in the module, the requirements of upstream bits within the same module are also satisfied.*

Proof: For any given vector at step j, the corollary assumes that the inputs are available to compute its most downstream bit in a particular LSA module. Within the same module, all other bits of that vector for step j are upstream to the given bit. Thus, from Theorem 3 the inputs to compute the other bits of the vector for the LSA module are also available.

Theorem 4 *The modules in an LSA design are identical when all bits in a data vector utilize the same boolean expression.*

Proof: From Corollary 1, an LSA module can be constructed by satisfying the input requirements for one bit slice along all vectors, where the bits in this slice are defined as the most downstream bits in the LSA module. Let this reference bit slice, s, belong to module A. Define module B as the neighboring downstream module to module A. From Theorem 2, after two substages of computation, the input requirements of bit slice s are satisfied. From Theorem 3 the input requirements of bits upstream from s are also satisfied. Since s is the most downstream bit slice of module A, the two substages belong to module B. Those bits already defined in the two substages are the only dependencies within module B that are involved in data transfer to an upstream module. The remaining bits that are not currently defined in the two substages will be used only in later cycles by downstream modules or module B. Let bit slice t represent the most downstream slice currently defined in the two substages and define t as the most downstream slice in module B. We then define those bits of module B that are not currently defined in the two substages as being computed by the second substage, where substage 2 was created by satisfying the downstream dependencies of substage 1 (from Theorem 2). Bit slice t is therefore computed entirely in substage 2. If we again use t as our reference bit slice and repeat the design process for another 2 substages, another module will be created. These two modules will be identical since both were generated from one bit slice and utilize the same boolean expressions. In continuing this process, all successive modules utilizing the same boolean expressions will be identical. Therefore, the creation of one LSA module suffices as a template for the other LSA modules in the same design.

Acknowledgements

This work has been supported in part by the State of California MICRO Grant "Field-Programmable Application-Specific Processor Arrays" and Xilinx, Inc.

References

[BuWa89] D. Buell, R. Ward, "A Multiprecise Integer Arithmetic Package," Journal of Supercomputing, 3:89-107, 1989.

[Erce84] M.D. Ercegovac, "On-Line Arithmetic: An Overview," Proc. SPIE Vol. 495 – Real Time Signal Processing VII, 1984, pp.86-92.

[ErLa85] M.D. Ercegovac, T. Lang, "A Division Algorithm with Prediction of Quotient Digits," Proc. 7th IEEE Symposium on Computer Arithmetic, 1985, pp. 51-56.

[ErLa93] M.D. Ercegovac, T. Lang, *Digit-Recurrence Algorithms and Implementations for Division and Square Root*, Kluwer Academic Publishers, 1994.

[Fern93] J.S. Fernando, *Design Alternatives for Recursive Digital Filters Using On-Line Arithmetic*, Ph.D Thesis, UCLA, 1993.

[Hwan79] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley and Sons, 1979.

[LoEr92] M.E. Louie, M.D. Ercegovac, "Mapping Division Algorithms to Field Programmable Gate Arrays," Proc. 26th Asilomar Conference on Signals, Systems, and Computers, Oct. 26-28, 1992, Pacific Grove, CA, pp. 371-375.

[LoEr93a] M.E. Louie, M.D. Ercegovac, "On Digit-Recurrence Division Implementations for Field Programmable Gate Arrays," Proc. of the 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 202-209.

[LoEr93b] M.E. Louie, M.D. Ercegovac, "A Variable Precision Square Root Implementation for Field Programmable Gate Arrays," to appear in the Journal of Supercomputing, 1994.

[LoEr93c] M.E. Louie, M.D. Ercegovac, "Implementing Division with Field Programmable Gate Arrays," to appear in a special issue on computer arithmetic in the Journal of VLSI Signal Processing, 1994.

[MoCi94] P. Montuschi, L. Ciminiera, "Radix-2 Division with Quotient Digit Prediction without Prescaling," Proceedings of the HICSS, Jan. 1994.

[ShVu93] M. Shand, J. Vuillemin, "Fast Implementations of RSA Cryptography," Proc. of the 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 252-259.

[Tu90] P. K.-G. Tu "On-Line Arithmetic Algorithms for Efficient Implementation," Ph.D. Dissertation, Computer Science Department, UCLA, CSD-900029, Sept. 1990.

[Xil92] Xilinx, "XC4000 Logic Cell Array Family - Technical Data," San Jose, 1992.