1999

Design of low-complexity pipeline digital systems

Anita Rao
Lehigh University

Rao, Anita

Design of Low-Complexity Pipeline Digital Systems

January 2000

Design of Low-Complexity Pipeline Digital Systems

by

Anita Rao

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Engineering

Lehigh University

June 1, 1999

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering.

JUNE 1, 1999
Date

Thesis Advisor

Chairperson of Department

Acknowlegdements

I am indebted to my mentor and thesis advisor Dr. Meghanad Wagh who has instructed me, stimulated my research and encouraged my learning. Everyone of our meetings always left me highly motivated. This thesis which represents the culmination of all that I have learnt in the course of my study at Lehigh University, was made possible by his guidance and patience. I would like to acknowledge here, his breadth as well as depth of knowledge.

I am grateful to Zhenyu Zhu of the Image Processing and Patern Analysis Lab, for providing me with material that was helpful in preparing the applications of the model.

And, I wish to acknowledge the support from my husband Venkatesh Rao, for his valuable feedback on this thesis and his help in the preparation of this document.

Contents

List of Figures

Abstract

This thesis introduces a novel architecture suitable for VLSI implementation for linear as well as nonlinear systems. The simple design techniques can be applied to diverse applications to obtain architectures with minimal hardware and optimal time complexities.

In order to facilitate the model and design of these systems, the concept of *Generator Polynomial* is introduced. An analysis of the generator polynomial and it's salient properties are presented so as to aid in the design methodology. The architecture consists of several identical modules connected serially. The internal hardware of each module depends on the specificity of the application. Two algorithms have been derived to design the hardware and the architecture structure.

To illustrate the wide potential of this model, design and implementations of practical applications have been incorporated into the thesis. For applications that do not directly embrace the technique, heuristic algorithms have been proposed. Generalization of the concept for $2-D$ applications has also been introduced. It is believed that this is a first ideal step toward design and implementation of powerful, low-complexity pipeline architectures.

Chapter 1

Introduction

Very large-scale integrated (VLSI) circuits have permeated the electronics industry in the form of consumer and commercial chips such as microprocessors, memory, digital signal processors, and embedded controllers. These mass-market chips have found their way into an astounding range of consumer products from computers down to simple appliances. Another wave of VLSI chips that are now becoming prevalent in many commercial ventures are the application-specific integrated circuits (ASICs). These are designed for specialized applications and produced in much lower quantities than the previously described chips.

The exponential advances in VLSI technology has made complex systems possible on single chips. For example, it is now possible to implement a simple risc core on 1 square millimeter. It is projected that with the ongoing phenomenal progress, logic implementation with several hundreds of millions of transistors will soon be integrated on a single IC. In case of memory chips, the density is expected to be even higher.

The major advantages of a "system on a chip" are highly increased speed (because of short interconnects), low cost (because of high throughput and greater applicability (because of fewer

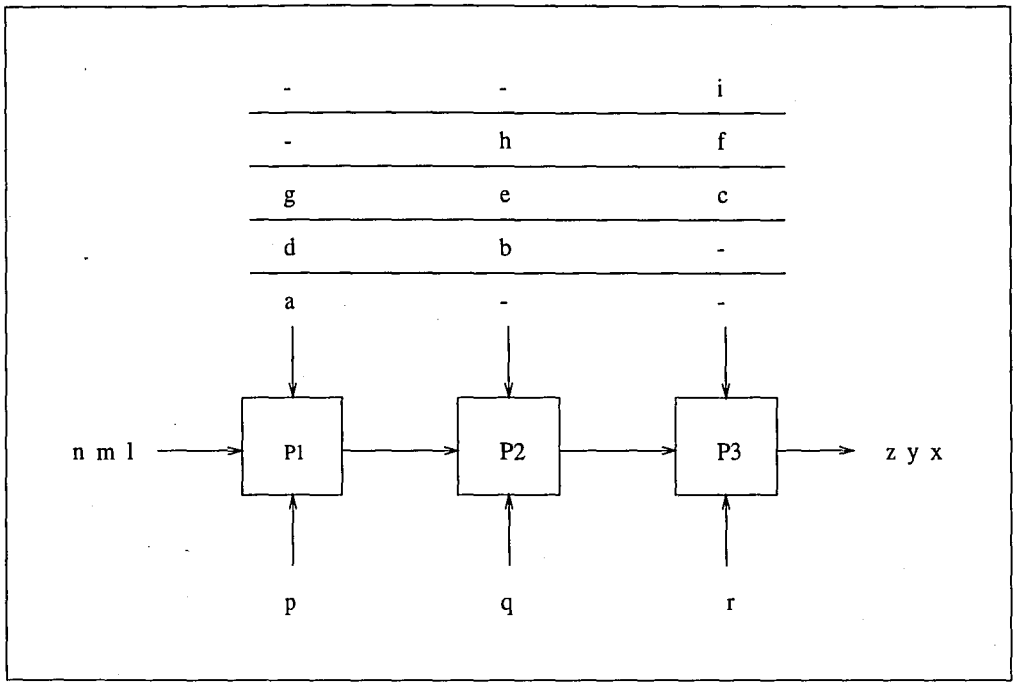


Figure 1.1: Example for Systolic Architecture : Matrix Multiplication

ICs, smaller real estate and power). Further, there is high throughput since concurrent processes are possible.

However, to realize these advantages, VLSI designs need to be modular with each module individually optimized; they also need to be repetitive so that the layout is area efficient and layout time is optimized. Therefore, unfortunately, the design is deprived of it's flexibility. Added to all these, is the constrain that data paths should not cross each other.

An important class of architectures that exploits the recent trends in VLSI technology are the systolic architectures,[1], introduced by H. T. Kung and Charles Leiserson in 1978. A systolic architecture is an arrangement of processors in an array (often rectangular) where data flows synchronously across the array between neighbors. Each processor is a Moore machine and data is exchanged only between adjacent processors. In this architecture, alternate processors are clocked together.

A processor at each step takes in data from one or more neighbors (e.g. top and left), processes it and, in the next step, outputs results in the opposite direction (downward and right).

Thus Systolic computation employs both pipelining and parallel computation efficiently. Because of it's modularity and regularity, it is ideal for VLSI implementation. Finally, because of it's communication only with the neighbors and the Moore model of each processor, the clock speed of a systolic architecture is independent of the size of the data sequence.

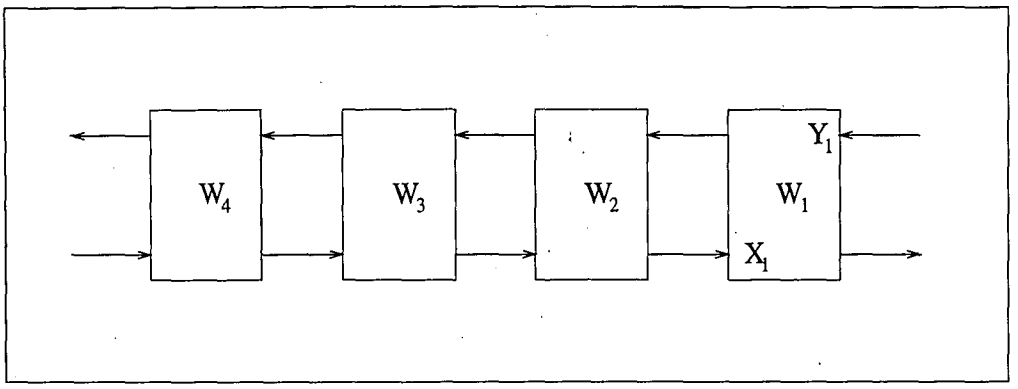
An example of a systolic algorithm might be matrix multiplication. One matrix is fed in a row at a time from the top of the array and is passed down the array, the other matrix is fed in a column at a time from the left hand side of the array and passes from left to right. Dummy values are then passed in until each processor has seen one whole row and one whole column. At this point, the result of the multiplication is stored in the array and can now be output a row or a column at a time, flowing down or across the array.

As illustrated in Fig 1.1, each cell (P1, P2, P3) does just one instruction - Multiply the top and bottom inputs, add the left input to the product just obtained, output the final result to the right. The cells are simple with just one adder and a few registers.

The order in which one feeds input into the systolic array is very important. At time t_0, the array receives l, a, p, q, and r (the other inputs are all zero). At time t_1, the array receives m, d, b, p, q, and so on. Results emerge after 5 steps.

Systolic arrays have traditionally provided efficient, high performance execution for computation intensive applications. The idea is to exploit VLSI efficiently by laying out algorithms (and hence architectures) in $2-D$ (not all systolic machines are $2-D$, but probably most are). The architectures thus produced are not general but tied to specific algorithms.

However, this is good for computation-intensive tasks but not I/O-intensive tasks, for example, signal processing. Most designs are simple and regular in order to keep the VLSI implementation



Each Module: Multiplier and Accumulator

Yout = Yin + WXin

O = n

Xout = Xin

Figure 1.2: Example for Systolic Architecture : Matrix Multiplication

costs low. Programs with simple data and control flow are best. Hence we have developed a new architecture which addresses all the areas - computation as well as I/O intensive applications. It is not necessary to flush all the buffers and refill with data when the processing sequences are very long. For example Fig 1.2 shows Linear Convolution implemented with systolic arrays. It can be seen that for very long sequences of data, the modules have to be flushed and refilled.

In this thesis, it is shown that many problems could be solved using much lower hardware and time complexities. The architecture has pipelining and the partial data actually moves through cascaded units reducing computation time and the amount of hardware.

For example, using this strategy, it is possible to find a running maximum of every 16 consecutive numbers in a sequence. By using only four adders and nineteen registers, one can find each maximum within time equal to one adder delay.

A Laplacian filter used in image processing may likewise be implemented. The hardware complexity for this is a mere four adders and ten registers. The time to process each set of data is just one adder delay. Similarly, point detection and line detection filters can be realized us-

ing this strategy requiring four adders and eleven registers, and, six adders and fifteen registers respectively.

Another example of where this idea may be applied is numerical integration which uses a linear combination of data. The hardware complexity for using the trapezoidal rule is just two adders and four registers.

Chapter 2 of this thesis formulates the problem and models the architecture. All the theoretical results necessary to obtain final implementation are derived here. We also provide implementation of an example application which clearly illustrates the superiority of our technique over others. Chapter 3 is devoted to four diverse applications. In particular, we derive architecture for Running Sum, Running Maximum, Sobel Operator and Extrapolation. The hardware and time complexities of each architecture are also discussed. Chapter 4 deals with generalizations including one to $2-D$ applications. Finally, Chapter 5 summarizes the conclusion and presents avenues for further research.

Chapter 2

Base Module and An Example

In the previous chapter, we talked about how our architecture was faster and reduced hardware requirements. Beginning with an example, we illustrate this point further. For any design, there have to be specific ways or rules to follow so as to realize it. In this chapter, we lay the groundwork for designing architectures using our method. The theory behind the results has been explained and substantiated. We have introduced here, the base module and the generator polynomial which are the building blocks for our architecture. This chapter also gives the algorithms one would require to design the modules for a particular problem. Another important aspect that is dealt with here is translation of the design method into the architecture. Issues that come up during this mapping are discussed.

2.1 Example

Suppose $y_i = x_i + 3x_{i-1} + 3x_{i-2} + x_{i-3}$. This problem may be implemented in a number of ways. We have shown how with our design, low hardware complexity may be achieved.

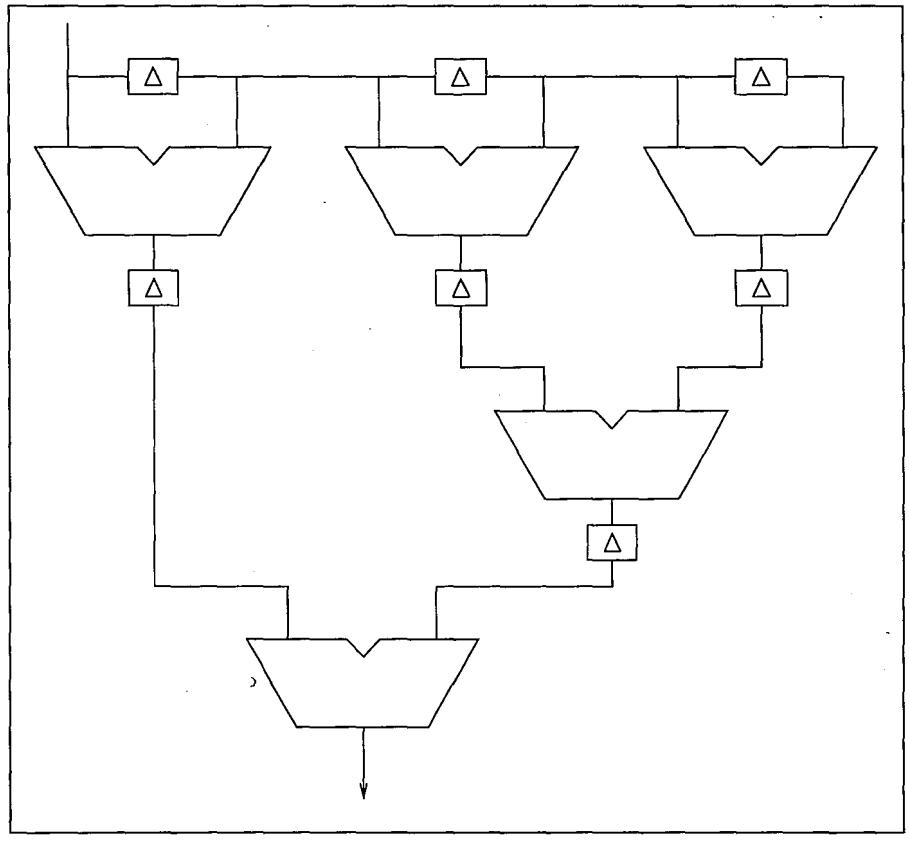


Figure 2.1: A bad implementation of $y_i = x_i + 3x_{i-1} + 3x_{i-2} + x_{i-3}$

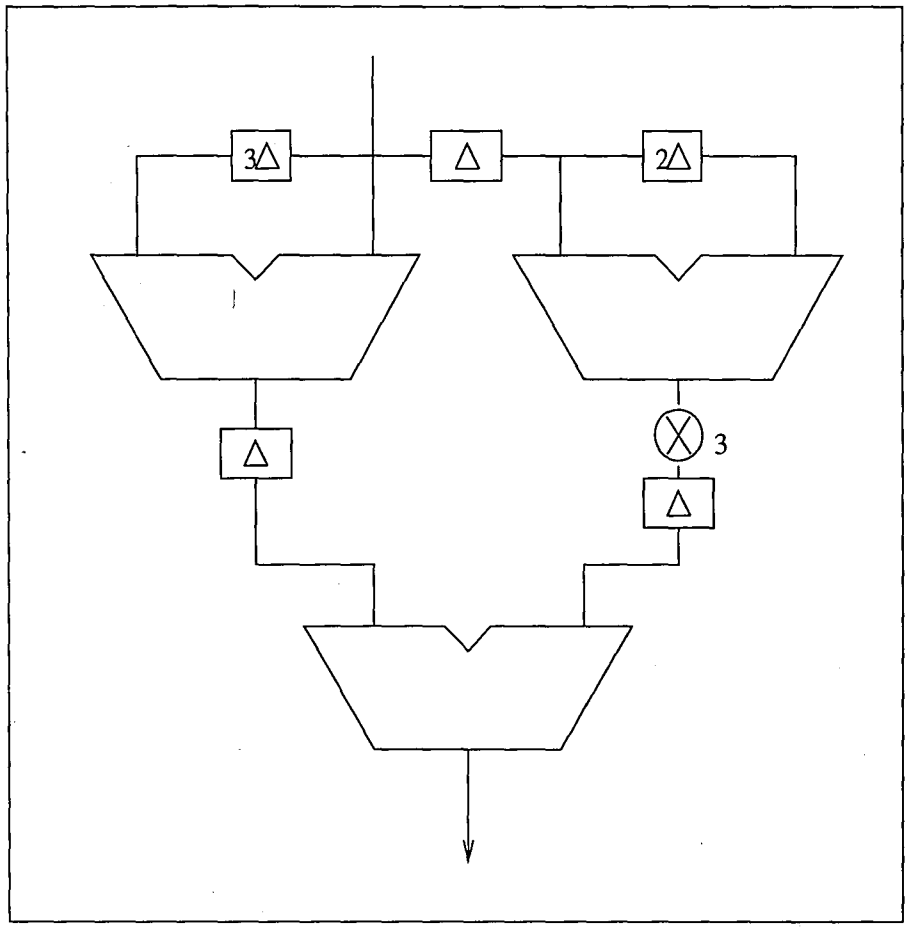


Figure 2.2: Another Implementation of $y_i = x_i + 3x_{i-1} + 3x_{i-2} + x_{i-3}$

One possible implementation :

Fig 2.1 shows one of the ways in which the stated problem may be realised. It uses 5 adders and seven registers.

Another implementation :

Fig 2.2 shows another implementation of the stated problem. It is uses just three adders as compared to five in the previous case. But the number of registers is nine. And the greatest drawback is the presence of the multiplier in this architecture which increases hardware and computation time.

Our Design

Our architecture for the same problem shown in Fig 2.3 is optimum for hardware as well as computational time. It uses three adders and 6 registers and the result is available for every adder time.

In this design, we use cascaded adders so that the solution propagates in a pipeline fashion. For the first adder, one input is the data x_i and the second input is x_i delayed by 1 unit. The output of this adder and it's delayed version are the inputs to the second adder and so on.

In this particular implementation, the inputs to the first adder are x_i and x_{i-1}. Therefore the output of the first adder is $x_i + x_{i-1}$. Hence, the second adder sums the terms $x_i + x_{i-1}$ and $x_{i-1} + x_{i-2}$. The output of this is $x_i + 2x_{i-1} + x_{i-2}$. Propagating through the third adder in the same fashion, the final output is $y_i = x_i + 3x_{i-1} + 3x_{i-2} + x_{i-3}$.

The last implementation uses just three adders and five registers. And it may be pointed out that one clock cycle of this architecture is equivalent to one adder time.

2.2 Modules: Registers and Adders

In this thesis, we will study architectures which look similar to that of the last implementation. One can notice that there is a basic module which is repeated three times to provide the needed result.

One may observe that using this module yields a large degree of flexibility towards solving a problem. There may be a few different ways in which to group the modules. Some methods may require additional adders, and others may require additional delay elements. It is interesting to note that requirements of a Delay FF and a Full Adder are comparable [2], thus giving not much

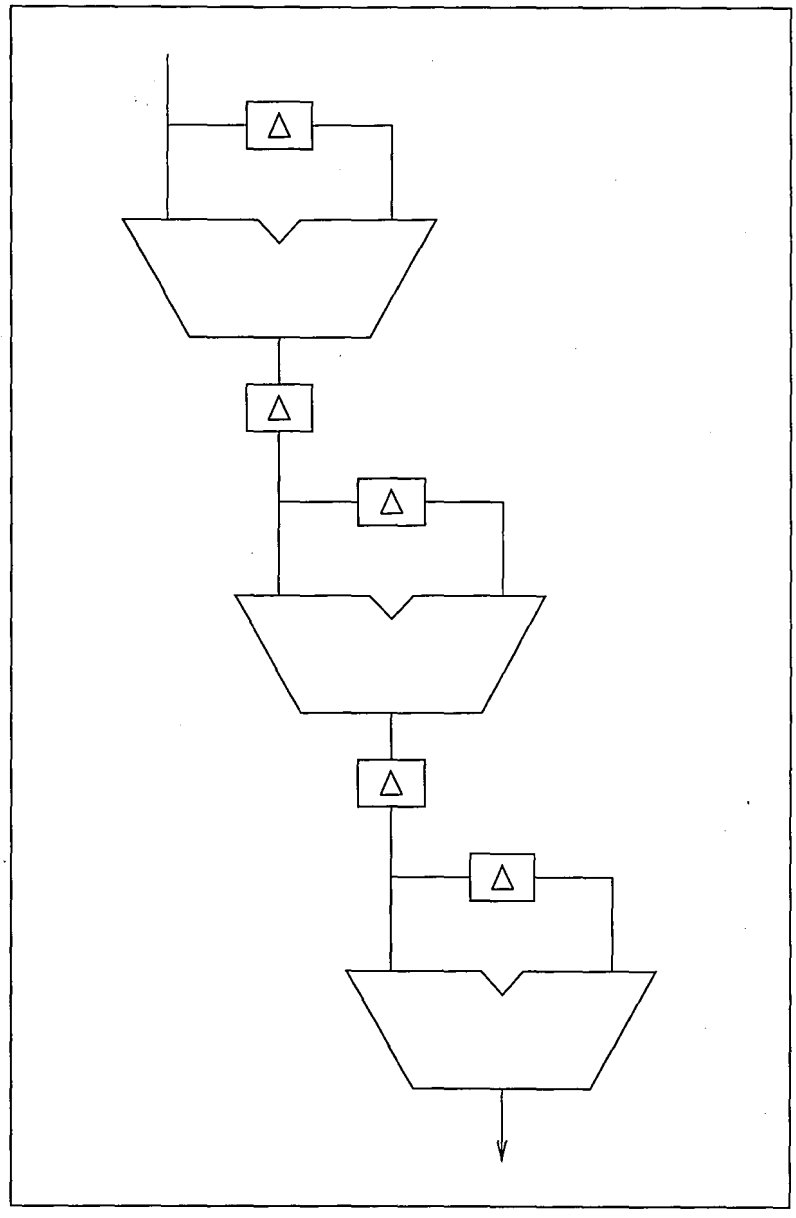


Figure 2.3: Our Implementation of $y_i = x_i + 3x_{i-1} + 3x_{i-2} + x_{i-3}$

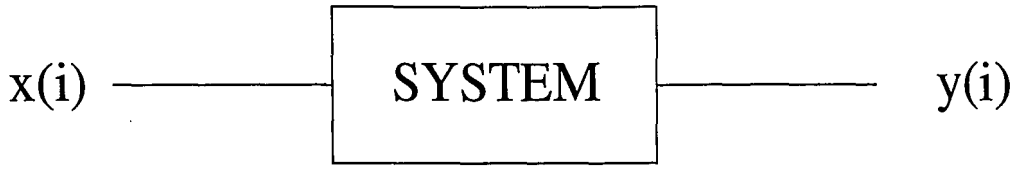


Figure 2.4: Basic Problem

reason to prefer one method to the other. A delay FF would require 22 transistors and 16 grids, while a FA would require 24 transistors and 15 grids. put reference here.

2.3 Modeling and Analysis

Consider a system which has input $x(i)$ and output $y(i)$ as shown in Fig 2.4. Design of such a system is often a complicated task. Further, for a linear system, $y(i)$ may be expressed as a linear convolution of the input $x(i)$ and the *system impulse response*. While dealing with nonlinear systems, designing the hardware becomes an especially involved process. In this section, we show how we try to achieve this by modeling the problem. We write the output as:

$$y(i) = \Xi\ x(i-j)c_j$$

where Ξ is some arbitrary linear or nonlinear operation. c_j's are integers. We will assume that Ξ is associative and commutative. However, the multiplication may not be distributive over Ξ and therefore we require a mapping of the problem so that the system can be implemented. Examples of Ξ are *addition*, *subtraction*, *maximum*, *minimum*, *parity*, *weights*, etc.

In order to study these systems, we use the concept of a generator polynomial represented by

$$g_n(z) = \sum_{i=0}^{n} c_i z^i$$

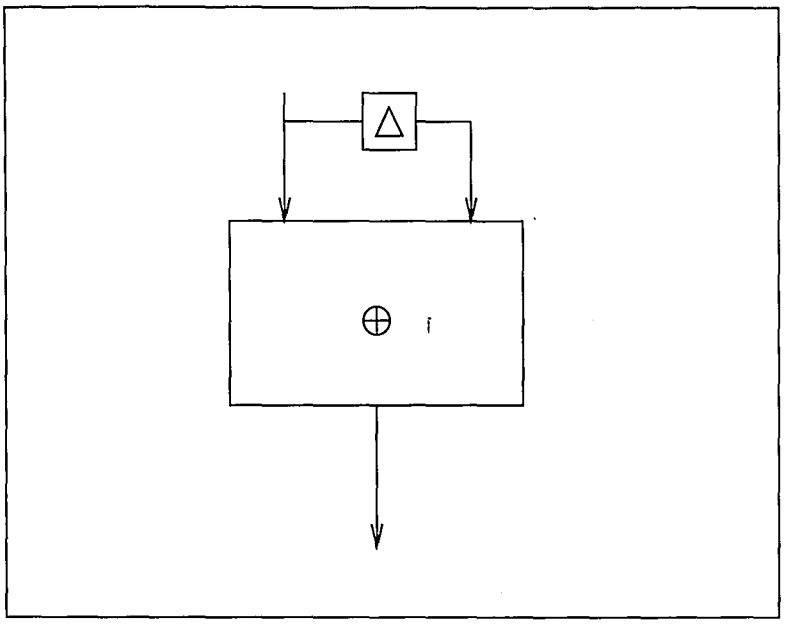


Figure 2.5: Base Module

We will show that for certain forms of the generator polynomial, a system can be implemented in a modular fashion with minimized hardware and simultaneously optimal time. All modules are identical and their functionality mimics Ξ. Fig 2.5 shows one such module. The overall architecture, on the other hand, will have a structure based only upon the generator polynomial.

We now present the desirable properties [3][4] of such a generator polynomial.

2.3.1 Properties of Generator Polynomial

P1. Roots on Unit Circle The generator polynomial $g_n(z) = \sum_{i=0}^{n} c_i z^i$ has all its roots on unit circle.

Proof (by induction): Consider the first stage of an architecture where the module looks like in Fig 2.5. If the delay is k units, and $x(i)$ is the input, then the output will be

$$y(i) = x(i) \oplus x(i-k)$$

And the polynomial corresponding to this output will be

$$g(z) = 1 + z^k$$

Thus for stage 1, the roots of $g(z)$ are on unit circle.

We assume that the Generator Polynomial at intermediate stage in the architecture has roots on the unit circle. $\sum_{i=0}^{N} c_i x_i = \Pi_{i=1}^{n}(1 + x^{d(i)})$. The degree of the polynomial N = $\sum_{i=1}^{n} d(i)$. Also, it is assumed that $c_0 = 1$. The polynomial may be represented as $x(i) + c_1 x(i+1) + c_2 x(i+2) + \cdots$

Let $y(i) = \sum_{t=0}^{N} c_t x(i+t)$. Consider $y(i)$ propagating through one module with a delay $d(n+1)$ as shown in 2.6. The output of the adder will be

$$\begin{aligned}
&= \sum_{t=0}^{N} c_t x(i+t) + \sum_{t=0}^{N} c_t x(i - d(n+1) + t) \\
&= \sum_{t=0}^{N} c_t x(i + d(n+1) + t) + \sum_{t=0}^{N} c_t x(i+t) \\
&= \sum_{t=0}^{N+d(n+1)} c'_t x(i+t), where, c'_t = c_t + c_{t-d(n+1)}
\end{aligned}$$

Note that $c_t = 0$ if $t < 0$ or $t > N$.

Now, the new representation of the polynomial is

$$\begin{aligned}
\sum_{i=0}^{N+d(n+1)} c'_i z^i &= \sum_{i=0}^{N+d(n+1)} (c_i + c_{i-d(n+1)}) z^i \\
&= \sum_{i=0}^{N+d(n+1)} c_i z^i + \sum_{i=0}^{N+d(n+1)} c_{i-d(n+1)} z^i \\
&= \sum_{i=0}^{N} c_i z^i + \sum_{i=d(n+1)}^{N+d(n+1)} c_{i-d(n+1)} i z^i \\
&= \sum_{i=0}^{N} c_i z^i + \sum_{i=0}^{N} c_i z^{i+d(n+1)}
\end{aligned}$$

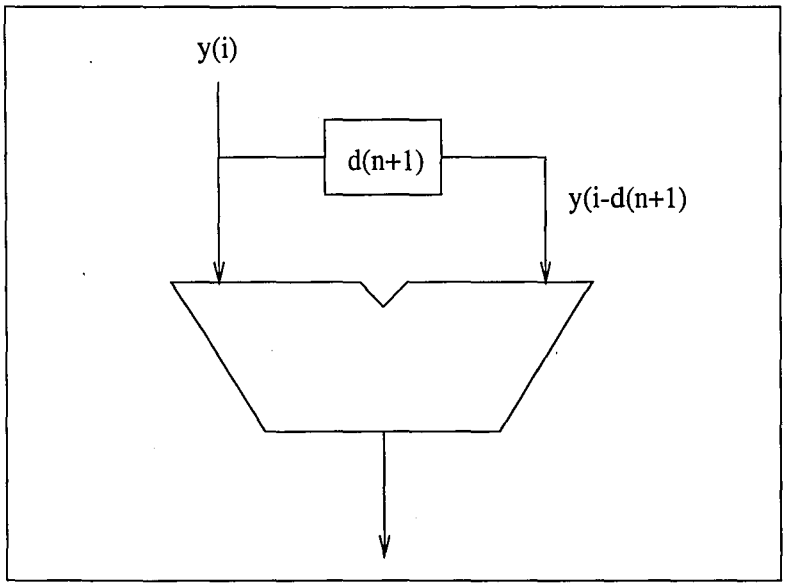


Figure 2.6: Propagation through a single module

$$= 1 + z^{d(n+1)} \sum_{i=0}^{N} c_i z^i$$

$$= \prod_{i=0}^{n+1} (1 + z^{d(i)})$$

P1. Symmetry The coefficients of the generator polynomial are symmetric, i.e., the coefficient of z^i is the same as the coefficient of z^j if $i + j =$ degree of the polynomial.

Proof : Now, for a polynomial $g_n(z)$ to be symmetric, it has to satisfy the following condition.

$$z^n g_n(z^{-1}) = g_n(z)$$

The generator polynomial is represented by

$$g_{n-1}(z) = \sum_{i=0}^{n-1} c_i z^i$$

Using the above representation, it follows that

$$\begin{aligned} g_n(z) &= \sum_{i=0}^{n-1} c_i z^i + \sum_{i-0}^{n-1} c_i z^{i+t} \\ &= g_{n-1}(z) + z^t g_{n-1}(z) \\ &= (1+z^t) g_{n-1}(z) \end{aligned}$$

Therefore, $g_{n+1}(z) = (1+z^t)g_n(z)$

Now, $g_{n+1}(z^{-1}) = (1+z^{-t})g_n(z-1)$

Multiplying both sides of the equation by the highest power in $g_n(z-1)$,

$$z^{t+n-1} g_{n+1}(z^{-1}) = (1+z^t) g_{n-1}(z)$$

Hence, $z^{t+n-1} g_{n+1}(z^{-1}) = g_{n+1}(z)$

Therefore it follows that the new polynomial is symmetric. ∎

P2. Power of 2 The sum of coefficients of the generator polynomial is a power of 2, i.e., $\sum_{i=0}^{N} c_i = 2^p$ where p is an integer.

Proof : Let us assume that the generator polynomial $g_{n-1}(z) = \sum_{i=0}^{n-1} c_i z^i$ has sum of its coefficients to be a power of 2.

The new polynomial may be written as

$$g_n(z) = (1+z^t) g_{n-1}(z)$$

In the above equation, $g_n(z)$ may be treated as a product of two polynomials. Then the sum of the coefficients of $(1+z^t)$ is 2. It has already been assumed that the sum of coefficients of

$g_{n-1}(z)$ is a power of two, say 2^p where p is any integer. Then the sum of coefficients of $g_n(z)$ will be the product of 2 and 2^p which is again a power of 2. Hence the sum of coefficients of the new polynomial is still a power of two. ∎

The architecture design is clearly related to the factors of the generator polynomial. The following section details how the polynomial may be factorized and implemented to facilitate modular implementation [5]. The complexity of the algorithms proposed is also discussed.

2.4 Module Design Algorithms

Factor $(1+x^k)$ of a degree n polynomial $P(x)$ can be determined using the following algorithm.

2.4.1 Algorithm 1 (Factor $(1+x^k)$)

1. (initialization) Let $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$, where a_i, $0 \leq i \leq n$ are constant coefficients.

2. (computation) Compute constants b_i, $0 \leq i \leq n$ as:

$$b_i = \begin{cases} a_i & i \leq k \\ a_i - b_{i-k} & \text{otherwise} \end{cases} \tag{2.1}$$

3. (checking) If $b_i = 0$ for *all* $n-k < i \leq n$, then $(1+x^k)$ is a factor of $P(x)$, and $P(x)$ may be written as $P(x) = (1+x^k)P_1(x)$ where,

$$P_1(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-k}x^{n-k}.$$

Proof : Polynomial $P(x)$ of degree n may be written as $\sum_{i=0}^{n} a_i x^i$. If $(1+x^k)$ is a factor of $P(x)$, then

$$(1+x^k) | \sum_{i=0}^{n} a_i x^i$$

$$\text{Thus,} \quad \sum_{i=0}^{n} a_i (-1)^{\lfloor \frac{i}{k} \rfloor} x^{imodk} = 0$$

The LHS may be written as

$$\sum_{j=0}^{\lfloor \frac{n}{k} \rfloor - 1} a_{t+kj}(-1)^j$$

$$= \sum_{t=0}^{k-1} \sum_{j=0}^{q} a_{t+kj}(-1)^j$$

The degree of the polynomial n, may be expressed as $n = qk + t$ where q and t are integers and, $0 \leq t < k$. Hence, the last k terms in $P_1(x)$,

$$\begin{aligned} b_{n-t} &= a_{n-t} - b_{n-t-k} \\ &= a_{n-t} - (a_{n-t-k} - b_{n-t-2k}) \\ &= \sum_{j=0}^{q} a_{n-t-kj}(-1)^j \end{aligned}$$

∎

The number of terms in the polynomial are $(n+1)$. The computation of $P(x)$ when $x^i = 1$ is blown up and shown below:

$$
\begin{array}{ccccccc}
a_0 & + & a_1x & + & a_2x^2 & + \cdots + & a_nx^n \\
a_i & + & a_{i+1}x & + & a_{i+2}x^2 & + \cdots + & a_{i+n}x^n \\
a_{2i} & + & a_{2i+1}x & + & a_{2i+2}x^2 & + \cdots + & a_{2i+n}x^n \\
a_{3i} & + & \cdots & & & &
\end{array}
$$

Notes :

For each $(1 + x^k)$ that is tested as a factor, the number of additions involved are $(n + 1 - k)$, where $1 \leq k \leq n$. The number of terms in the polynomial are $(n + 1)$.

Suppose $k = 1$. Then $x^k = x$. Therefore, all the terms in $P(x)$ add together making the total number of additions = total number of terms in $P(x)$ - 1, which is $(n + 1 - 1) = n$. If $k = 2$, then $\lceil \frac{n+1}{2} \rceil$ terms would be of the form cx^0 and the rest, of the form dx^1 where c and d may be any integer. So the number of additions in each set will be (number of terms in set - 1). On the whole, number of additions is given by total number of terms in $P(x)$ - 2 = $(n - 1)$.

Hence, it may be observed that for some k, the number of sets is k, and each group of $\frac{n+1}{k}$ would add together totaling $(n + 1 - k)$ additions.

A degree n polynomial may be factorized completely with the following algorithm.

2.4.2 Algorithm 2 *(Complete Factorization)*

1. (initialization) Let $P_1(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$, where a_i, $0 \leq i \leq n$ are constant coefficients and $(1 + x^{k_1})$ be a factor of $P_1(x)$.

2. (quotient representation) From previous step,

$$P_1(x) = (1 + x^{k_1})P_2(x)$$

where $P_2(x)$ may be written as $b_0 + b_1x + b_2x^2 + \cdots + b_{n-k_1}x^{n-k_1}$.

3. (iterative computation) Using *Factor* $(1+x^k)$ *Algorithm*, a factor of the form $(1+x^{k_2})$ may be found for $P_2(x)$. This process is continued until $P_1(x)$ is reduced to the form $(1+x_1^k)(1+x_2^k)\cdots P_j(x)$, where, $P_j(x)$ is either $(1+x_j^k)$ or cannot be reduced further using Algorithm 1.

Proof (by induction:) Let $Q(x) = b_0 + b_1x^1 + \cdots + b_{n-k}x^{n-k}$ be the quotient when $(1+x^k)$ divides $P(x) = \sum_{i=0}^{n} a_i x^i$.

$$b_0 = a_0$$

$$b_1 = \begin{cases} a_1 & k \neq 1 \\ a_1 - b_0 & \text{otherwise} \end{cases}$$

Let us assume that it is true for b_i. Then for the next term,

$$b_{i+1} = \begin{cases} a_{i+1} & i+1 \leq k \\ a_{i+1} - b_{i+1-k} & \text{otherwise} \end{cases} \tag{2.2}$$

∎

Proposition :

Factoring a degree n polynomial $(1+x^{i_1})(1+x^{i_2})(1+x^{i_3})\ldots(1+x^{i_t})$ requires exactly

$$\sum_{n}^{j=n+1-i_1-i_2-\ldots\ldots-i_t} (j)$$

additions or subtractions.

Proof (by induction) : Let $i_1 \leq i_2 \leq \ldots\ldots \leq i_t$. For the first factor, we test (1 + x) which amounts to one addition. If (1 + x) is indeed a factor, then since $i_1 = 1$, the number of additions required was $\sum_{j=n+1-i_1}^{n}(j)$ = n. For the second factor, then first $(1 + x)$ is tested and then $(1 + x^2)$ is tested tested. If this satisfies as a factor, then the required number of additions is $\sum_{j=n-1-i_1-i_2}^{n-i_1}(j)$. Hence now the total number of additions for determining factors one and two are:

$$\begin{aligned} \text{Total number of additions} &= \sum_{j=n+1-i_1}^{n}(j) + \sum_{j=n-1-i_1-i_2}^{n-i_1}(j) \\ &= \sum_{j=n-1-i_1-i_2}^{n}(j) \end{aligned}$$

Now, let us assume that the proposition is true for factors determined up to i_t.

Then, if the next factor is $(1 + x^{i_t+m})$, the number of additions/subtractions required are

$$\sum_{j=n+1-i_1-\cdots-i_t}^{n}(j) + (n+1-i_1-\cdots-i_t-i_{t+1}) + \cdots + (n+1-i_1-\cdots-i_t-i_{t+m})$$

$$= \sum_{j=n+1-i_1-i_2-\cdots-i_{t+m}}^{n}(j)$$

∎

2.5 Architecture Design

Until this point, the theory suggests what has to be done to a given particular problem. The problem is written in terms of a pertinent polynomial which is then subject to the two algorithms proposed in the previous section. Still, the translation of the result of these algorithms into the

final architecture remains to be discussed. We will do so in this section of Architecture Design.

From the second algorithm proposed in the previous section, all the factors are extracted. These factors are the key to what the architecture will look like.

Consider a problem which has been written in the form of a polynomial $P(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$. When algorithms *Factor* $(1+x^k)$ and *Complete Factorization* are applied $P(x)$ reduces to $(1+x_1^k)(1+x_2^k)\ldots(1+x_j^k)$. From these factors, we can say what the architecture looks like. The number of adders in the design are the number of factors obtained from the second algorithm. Meaning, in the above case, each factor translates to an adder. And the delay units depend on the power to which the x terms in the factors are raised. For example, for the first adder, the delay would be equal to k_1, taken from the first factor $(1+x_1^k)$ and the second adder's delay would be k_2 and so on.

2.6 Implementation Issues

Hardware and Time Complexities

The hardware reduces drastically with our architecture. This is due to the fact that partial results are moved into the next cascaded stage and thus using the previously computed saves hardware and time. The hardware complexity is determined by the factors to which the problem has been reduced to. The number of adders and delay units required to implement the problem define the hardware complexity. If $P(x)$ is factorizable in unitary factors, then the total delays required in its implementation = degree of $P(x)$ and the number of adders = the number of factors. The output of the system depends on the adder time. It is exactly equal to one adder time and this gives the time complexity.

Complete Factorization

The complexity involved in the design is reduced when the pertinent equation has been fully

factorized. And as discussed in section 4 of this chapter, the *Complete Factorization* algorithm is simple in that it can be applied and result obtained very fast.

Subtractors

In some cases, one or more factors of the polynomial may be of the form $(1 - x^k)$. Then the module corresponding to this factor will be a subtractor and not an adder. The architecture complexity does not differ in this case and the time also remains very much the same.

Chapter 3

Applications of the Model

In the previous chapter, the theory of the generator that fits into the architecture was developed. It looks limited in scope, but as illustrated here, the applications of our architecture span many different areas. The first section shows how our technique may be used to solve an age old but very simple problem - the Running Sum. This is a non-linear application and has been solved by many people in different ways. Using our method, we show that there is a drastic reduction in hardware and time complexities. The second section deals with an interesting problem : The Running Maximum. Again the simplicity of the design solution shows how potent our technique is. In the next section, we deal with more particular application in Image Processing. Filtering and working on images is a slightly more complex process. The Sobel Operator is extensively used in Spatial Filtering. Though the function looks like it may involve complex architecture, it really works out simple using our technique. Moving on to a more involved application in our section, we have shown how something like extrapolation fits into the realm of our design.

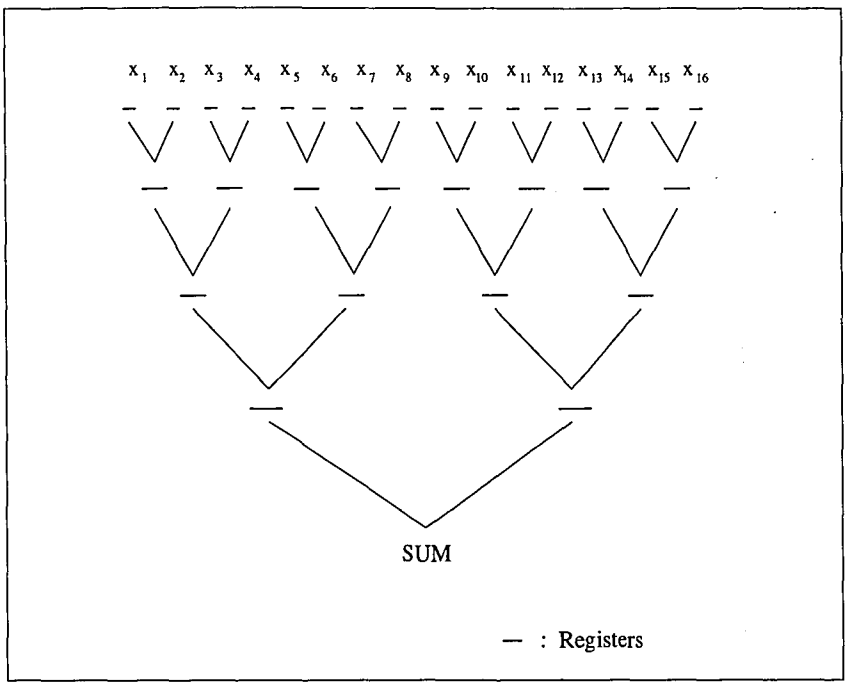


Figure 3.1: Running Sum: A General Implementation

3.1 Running Sum of 16 Numbers:

Consider calculating the running sum of 2^n data points. The first step in the design process would be to translate the data into a polynomial say $P(x)$. Referring to section 2.4 of chapter 2, algorithms *Factor* $(1+x^k)$ and *Complete Factorization* are applied on this polynomial. Once all the factors have been determined, from observation, the number of adders as well as the delay units required are determined. Now, for 2^n data values, the polynomial can be reduced to n factors and will look like this.

$$\begin{aligned} P(x) &= 1+x+x^2+\cdots+x^{2n-1} \\ &= (1+x)(1+x^2)(1+x^4)\ldots(1+x^k) \end{aligned}$$

Hence since there are n factors, the number of adders in the architecture will be n and the total

number of delay registers used will be $2n - 1$.

As an example, consider calculating the running sum of 16 data points. A general implementation would use the Divide and Conquer algorithm as shown in Fig 3.1. All the 16 values are first latched into registers. They are then summed in pairs in the next stage when we have 8 values. These inturn are summed in pairs and so on until the final sum is arrived at. This particular architecture requires 15 adders and 30 registers for it to function. And the hardware complexity is $O(n)$.

Now, if our architecture is applied to solving the same problem, there is a drastic reduction in hardware complexity. The number of adders required are a mere 4 and registers, 14 as compared to 15 adders and 30 registers in the previous implementation. The hardware complexity is now $O(logn)$.

Each module shown is an adder. The number of registers are indicated by the number of delays used. Each module sums the value available to it and its delayed version. At the end of the first module, sum of two number are available. So the second module sums four data points. The third stage outputs the sum of eight values. Hence at the end of the fourth stage, the sum of sixteen values is obtained.

3.2 Running Maximum

Running Maximum is an interesting application and it too fits in very well with the generator polynomial. It is an associative operation which makes it feasible to carry the results computed through the cascaded modules. The Maximum would be Max ... {Max {Max $\{x_1, x_2\}$, Max $\{x_3, x_4\}$}} .... Maximum of 16 numbers could be implemented with an architecture similar to that of the *running sum* application. Each module would contain a subtractor and logic for

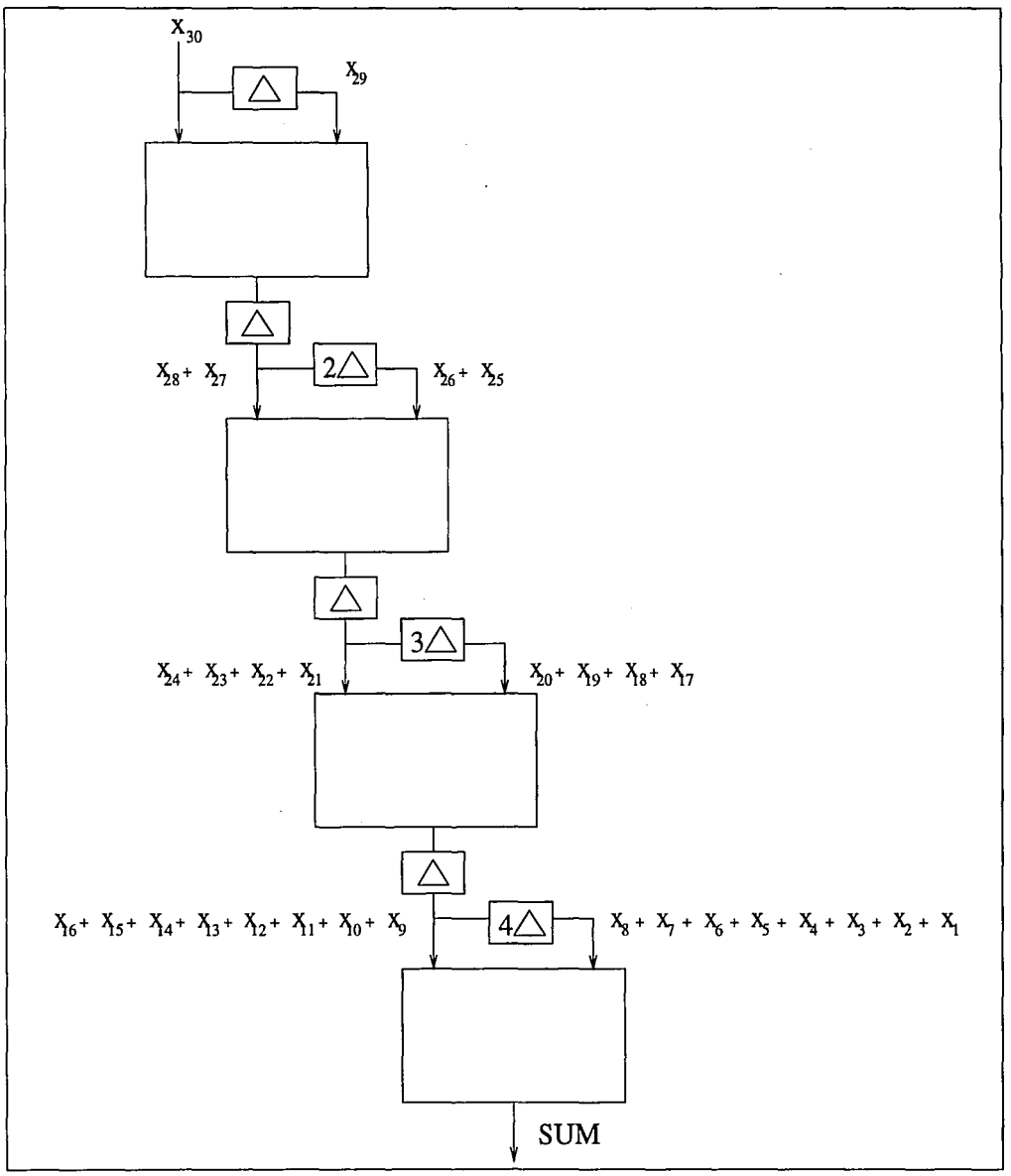


Figure 3.2: Running Sum: Our Implementation

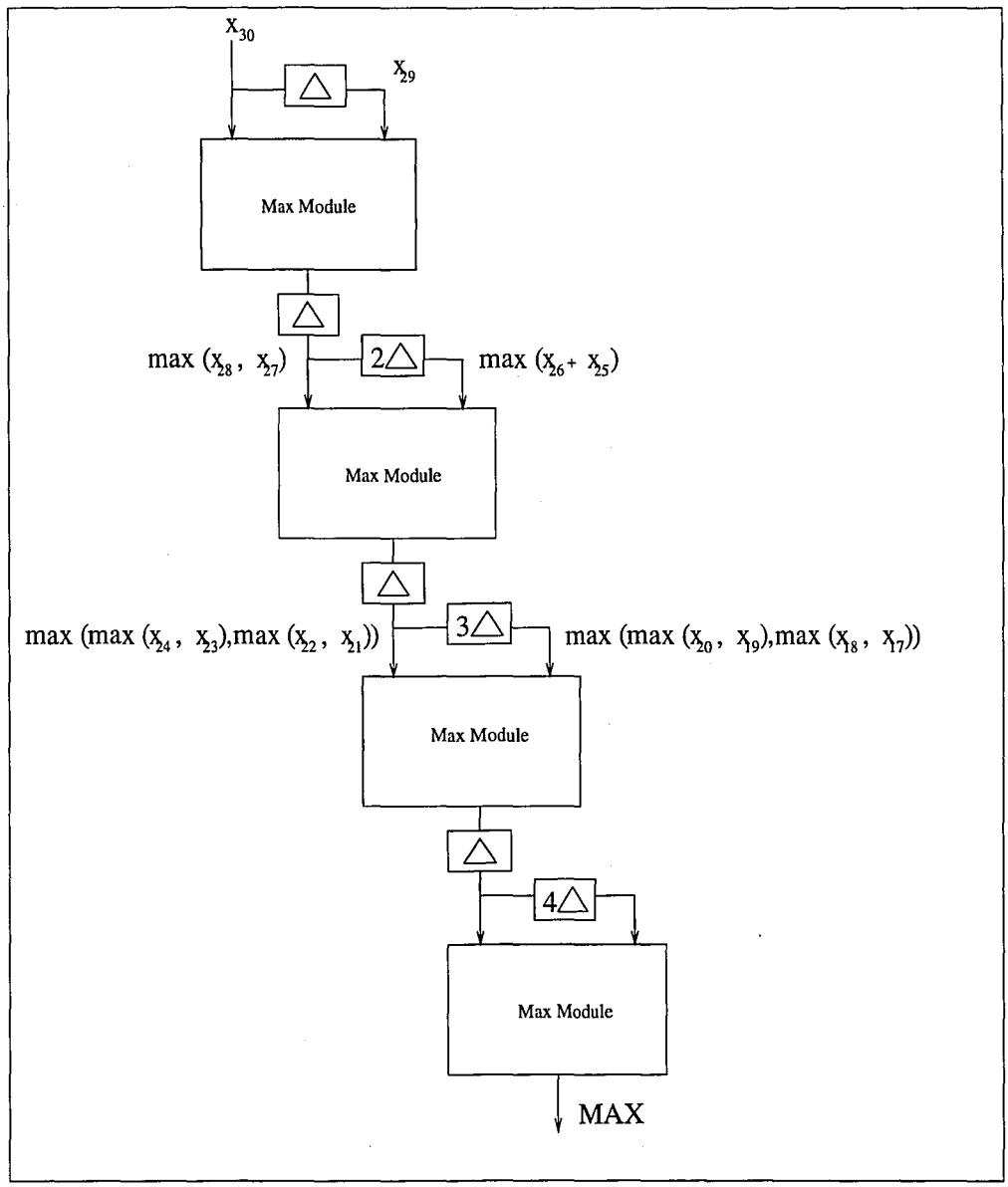


Figure 3.3: Running Maximum

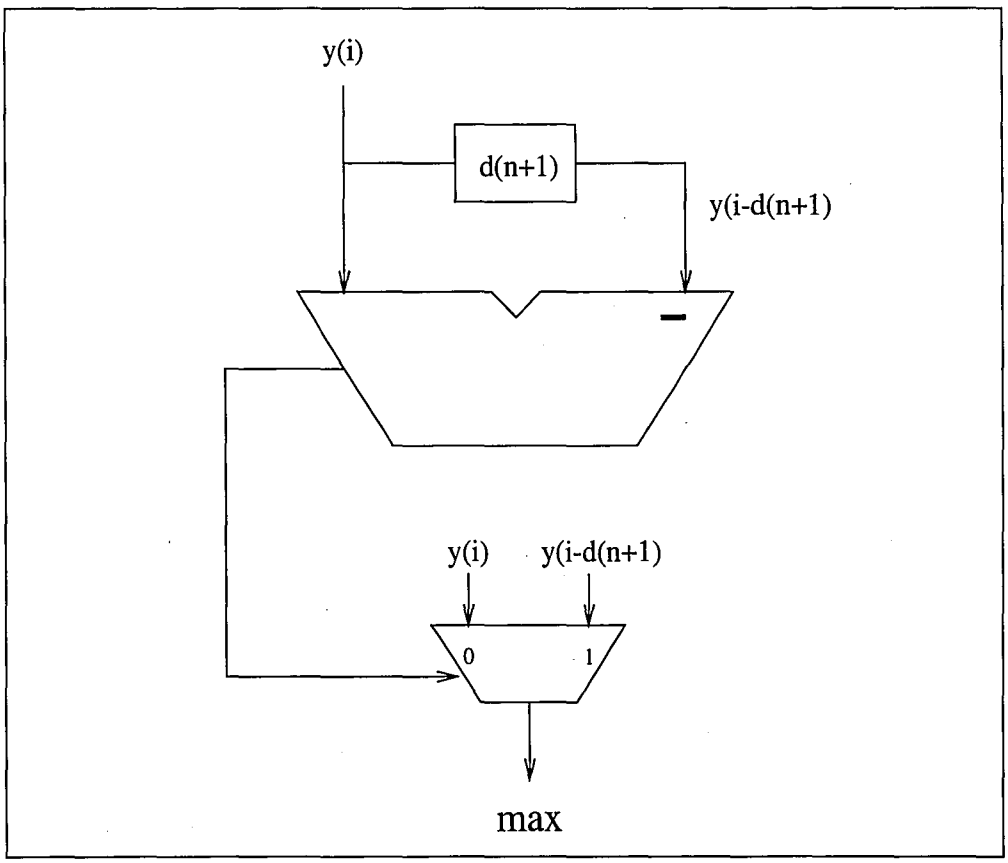


Figure 3.4: Running Maximum Module

determining the maximum of two values as shown in Fig 3.4. The architecture is shown in Fig 3.3. And as there are 16 numbers which is 2^4, the architecture comprises of 4 modules and the total delay units required would be $n-1$ which in this case is 15.

3.3 Digital Filters

Another area where applications fit into our architecture nicely is Image Processing. Many of the digital filters [6] used for image enhancement, detection, etc., can be realised using this model.

For example, the Sobel operator. In Spatial filtering, many gradient operators such as Robert Cross, Isotropic and Sobel and Prewitt operators are used. These gradient edge detectors emphasise regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. One of the convolution kernels of the sobel operator [7] has been implemented in 3.5.

It may be observed here that image processing mainly uses filters of a 3×3 dimension through which the pixels are processed. With this in view, our architecture is ideally suited to implementing these filters. The 3×3 pixel points from the image are input serially through the realised filter and the image is thus processed.

The 3×3 kernel is written as a function $Y(x) = -X_1 - 2X_2 - X_3 + X_7 + 2X_8 + X_9$. Inorder to fit it into the generator equation, $Y(i)$ is written as $-1 - 2X - X^3 + X^6 + 2X^7 + X^8$. Using the *Factor* $(1 + x^k)$ and *Complete Factorization* algorithms developed in Chapter 2, the pertinent equation factorizes into $(1+X)(1+X)(-1+X^6)$. From the factors it is obvious that the architecture has 3 modules - two adders with one delay unit each, and a subtractor with six delay units.

Problem:

$$Y(i) = -X_1 - 2X_2 - X_3 + X_7 + 2X_8 + X_9$$

Pertinent Equation:

$$Y(i) = -1 - 2X - X^2 + X^6 + 2X^7 + X^8$$

$$= (1 + X^2)(-1 + X^6)$$

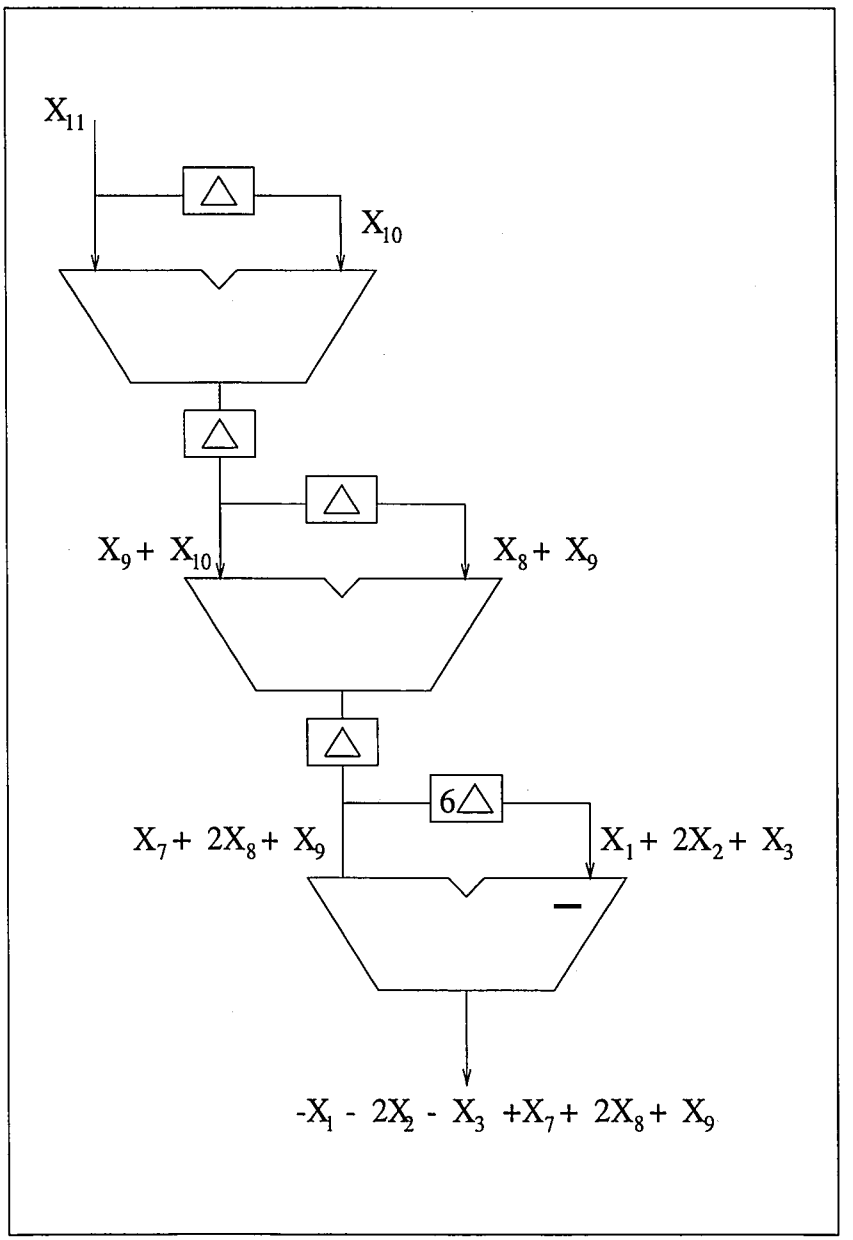


Figure 3.5: Sobel Operator

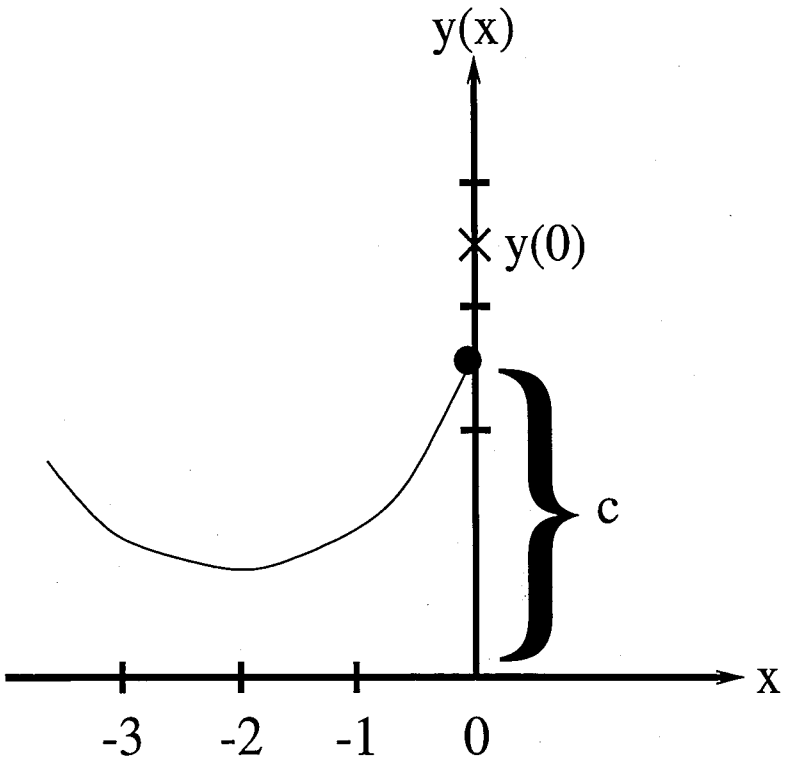


Figure 3.6: Extrapolation

3.4 Extrapolation

To illustrate the varied application potential, we show how this architecture may be used for *extrapolation.* Extrapolation is a complex and involved operation to implement. But in the following section we show that using our technique reduces it to an extremely simple solution.

We propose to derive the error of extrapolation for a n degree polynomial $y(x)$ and show that it fits into our generator equation. And then design the architecture for say, a second degree fit equation.

Let us assume that the n degree curve has to be extrapolated for y_0. We write an equation that fits the points $y_{-n}, y_{-n+1}, \cdots, y_{-1}$.

$$y(x) = \sum_{i=0}^{n-1} c_i x^i$$

Using the Van-der-Monde matrix V,

$$Y = Vc$$

$$\begin{pmatrix} y(-n) \\ y(-n+1) \\ \vdots \\ \vdots \\ y(-1) \end{pmatrix} = \begin{pmatrix} (-n)^0 & (-n)^1 & \cdots & (-n)^{n-1} \\ (-n+1)^0 & (-n+1)^1 & \cdots & (-n+1)^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ (-1)^0 & (-1)^1 & \cdots & (-1)^{n-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ \vdots \\ c_{n-1} \end{pmatrix}$$

Now, error $= y(0) - \sum_{i=0}^{n-1} c_i 0^i$

V is invertible because $-n$, $-n+1$, ... , -1 are distinct. Therefore,

$$|V_{-n}| = (-n+1)(-n+2)\ldots(-1)$$

$$\text{Now,} \quad c_0 = \frac{|V_{-n+i}|}{|V|} y(-n+i)$$

$$\text{where,} \quad |V-n+i| = (-n)(-n+1)(-n+2)....(-1) \qquad \text{except } (-n+i)$$

Therefore,

$$\begin{aligned} c_0 &= \frac{(-n)(-n+1)(-n+2)....(-1) \qquad \text{except } (-n+i)}{(n-i-1)(n-i-2)....(1)(-1)(-2)....(i)} \\ &= \frac{n(n-1)(n-2)....(1)(-1)^{n-1}}{(n-i)(n-i-1)(n-i-2)....(1)(-1)(-2)....(i)} \\ &= \frac{n!}{(n-i)!i!}(-1)^{n-1} \\ &= C_i^n(-1)^{n-1} \end{aligned}$$

$$\text{Therefore,} \quad c_0 = -\sum_{i=0}^{n-1} C_i^n(-1)^n y(-n+i)$$

$$\begin{aligned} \text{Error} &= y(0) - c_0 \\ &= C_n^n(-1)^{n-n} y(-n+n) - c_0 \\ &= \sum_{i=0}^{n} C_i^n(-1)^{i-n} y(i-n) \end{aligned}$$

Thus the Error between the actual and predicted values is seen to fit the generator polynomial exactly making it implementable using our technique.

To illustrate this point, let us consider a second degree polynomial $y(x) = ax^2 + bx + c$. In Fig 3.6, c gives the extrapolated point and $y(0)$ is the actual data point in some second degree curve. The error which is $y(0) - c$ can be calculated by solving the polynomial equation for c, and subtracting it from $y(0)$. It amounts to $y(0) - 3y(1) + 3y(2) + y(3)$.

As shown in the previous illustration, this function is now tested for a fit in the generator equation as depicted in Fig 3.7. Once it is written in polynomial representation, the *Complete Factorization* algorithm is used to determine the fit. From the factors, we can observe that three subtractor modules with one delay element each are required in the architecture to implement this function.

Problem: $P(i) = Y_0 - 3Y_1 + 3Y_2 - Y_3$

Pertinent Equation: $P(i) = 1 - 3X + 3X^2 - X^3$

$= (1 - X)(1 - X)(1 - X)$

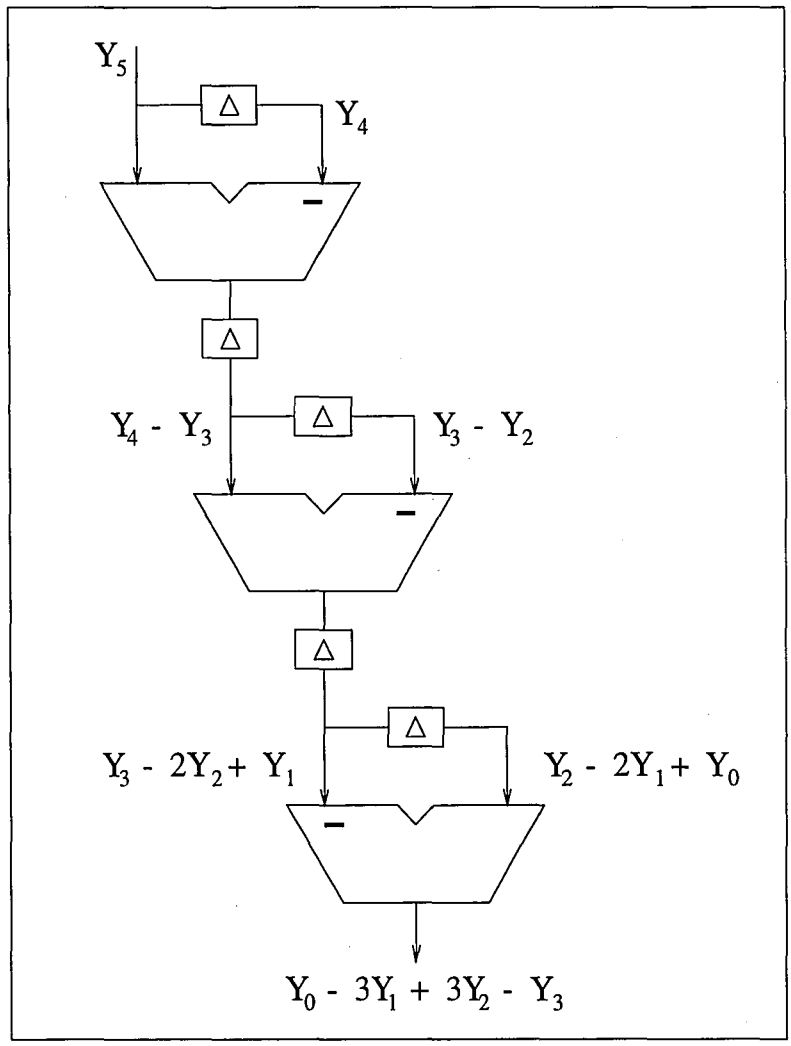


Figure 3.7: Implementation

Chapter 4

Generalizations and Extensions

The previous chapter has shown the immense potential that our technique has. And chapter 2 discussed how a system may be fitted into the Generator Polynomial and factorised completely using Algorithms *Factor* $(1+x^k)$ and *Complete Factorization.* Thus the number of modules, and module designs are determined which gives rise to the overall architecture. We have shown that using this methodology, the systems can be implemented with minimal hardware and optimal time.

The third chapter discusses applications which fit the Generator Equation. These systems can modelled such that they are completely factorized into the form of the generator equation. The Complete Factorization Algorithm is applied successively until the starting degree n polynomial is exactly of the form $(1 + x^{i_1})(1 + x^{i_2}) \cdots (1 + x^{i_j})$ where, $\sum_{m=1}^{j} i_m = n$.

In this chapter we focus on systems that don't fit the Generator Polynomial directly. When the *Complete Factorization* algorithm is applied to such a system, the end result is a term that is not of the form $(1 + x^k)$ and also cannot be reduced further.

Also, there are problems that may be two dimensional in nature. Design of such systems brings

in new issues that make the architecture design very complicated. Keeping this in mind, we present some preliminary ideas on how to approach the design of such systems.

4.1 Design of Non-Conforming Systems

In a situation when a generator polynomial cannot be decomposed into ideal factors, one can still try to partition the polynomial into two or more components each of which satisfy the properties specified in Chapter 2. This problem, in general, is very hard and we can only provide heuristic algorithms to attack it. However, as the following proposition shows, one should isolate all the possible factors of the type $(1 + x^k)$ before these algorithms are attempted.

Proposition. It is most optimal to factorize the polynomial until the last possible stage.

Whatever be the method of design, it is most ideal to apply the *Complete Factorization* Algorithm from section 2.4 before any attempt is made to fit the system.

We have developed some techniques of getting around the limitation of some systems that do not fit very well into our design methodology. These work well and provide optimal architecture designs for these systems in most cases. However, there are instances where these algorithms do not perform optimally. Having said this, we now present these algorithms. We have also included counter examples which illustrate the fact that they do not provide satisfactory design solutions in all cases.

4.1.1 Heuristic Algorithms

The following algorithms partition a generator polynomial into multiple polynomials each having the properties described in section 2.4 of Chapter 2. The process is different in each case and most of the time, the split terms can be incorporated into an optimal architecture design.

Algorithm 3 (Partition - I)

1. (initialization) Apply the *Complete Factorization* algorithm to the generator polynomial. Denote the last quotient by $Q(x) = c_0 + c_1 x + \cdots + c_j x^j$. Note that $Q(x)$ does not have anymore factors of the form $(1 + x^k)$.

2. (expansion) Expand $Q(x)$ in the following fashion :

$$1 + \cdots + 1 \ (c_0\text{times}) \ + \ x + \cdots + x \ (c_1\text{times}) + \ \cdots \ + \ x^j + \cdots + x^j \ (c_j\text{times})$$

3. (grouping) Group each 1 term with x^k terms beginning with the lowest k and moving onto successive higher powers until all the 1 terms are exhausted.

4. (factorization) Factor out the lowest power x from the remaining terms

5. (iteration) Repeat the previous two steps *grouping* and *factorization* until remaining terms are of the form $(1 + x^k)$ or $(1 + x^k + x^m)$, in which case it is treated as two partitions $(1 + x^k)$ and x^m

Example 1. $P(x) = 1 + 3x + 4x^2 + x^3 + x^4$.

Initialization: Algorithm *Complete Factorization* yields no factors for $P(x)$. Therefore, $Q(x) = P(x)$.

Expansion: $Q(x)$ is expanded as follows:

$$\begin{aligned} Q(x) &= 1 + 3x + 4x^2 + x^3 + x^4 \\ &= 1 + x + x + x + x^2 + x^2 + x^2 + x^2 + x^3 + x^4 \end{aligned}$$

Grouping:

$$Q(x) = (1+x)^2 + (x + x^2 + x^2 + x^2 + x^3 + x^4)$$

Factorization:

$$Q(x) = (1+x)^2 + x(1 + x + x + x + x^2 + x^3)$$

Iteration:

$$\begin{aligned} Q(x) &= (1+x)^2 + x(1+x+x+x+x^2+x^3) \\ &= (1+x)^2 + x((1+x)^2 + x + x^3) \\ &= (1+x)^2 + x((1+x)^2 + x(1+x^2)) \\ &= (1+x)^3 + x^2(1+x)^2 \end{aligned}$$

Now, these two partitions can be implemented separately and then combined using an adder to give the overall result.

Example 2. $P(x) = 2 + 5x + 5x^2 + 3x^3 + x^4$.

Initialization: Algorithm *Complete Factorization* yields factor $(1+x)^2$ for $P(x)$. Therefore, $Q(x) = 2 + x + x^2$.

Expansion: $Q(x)$ is expanded as follows:

$$\begin{aligned} Q(x) &= 2 + x + x^2 \\ &= 1 + 1 + x + x^2 \end{aligned}$$

Grouping:

$$Q(x) = (1+x) + (1+x^2)$$

Since this is the minimal form that can be reached, the end result is:

$$P(x) = (1+x)^2(1+x) + (1+x)^2(1+x^2)$$

Now, these two partitions can be implemented separately and then combined using an adder to give the overall result.

Algorithm 4 (Partition - II)

1. (initialization) Apply the *Complete Factorization* algorithm to the generator polynomial. Denote the last quotient from the previous step by $Q(x) = c_0 + c_1x + \cdots + c_jx^j$. Note that $Q(x)$ does not have anymore factors of the form $(1 + x^k)$.

2. (evaluation) Compute the polynomial $Q(x)$ at $x^k = \pm 1$ where, $1 \leq k \leq n$

3. (check) If the result is a power of 2, then for the corresponding k, $(1 \mp x^k)$ corresponds to a possible module

Example 1. $P(x) = 2 + 2x + 3x^2 + x^3 + x^4$.

Initialization: Algorithm *Complete Factorization* yields factor $(1 + x^2)$ for $P(x)$. Therefore, $Q(x) = 2 + x + x^2$.

Evaluation: Computing the polynomial $Q(x)$ at $x^k = \pm 1$ where, $1 \leq k \leq n$, we get:

x	$Q(x)$	*Possible Partition*
$x = 1$	$6\ (\neq 2^p)$	None
$x = -1$	$4\ (= 2^2)$	$(1 + x)$
$x^2 = 1$	$4 + 2x\ (4 + 2 \neq 2^p)$	None
$x^2 = -1$	$2x\ (2 = 2^1)$	$(1 + x^2)$

Check: For this problem, $P(x)$ decomposes into

$$\begin{aligned} P(x) &= (1+x^2)((1+x)+(1+x^2) \\ &= (1+x)^2(1+x)+(1+x^2)^2 \end{aligned}$$

The architecture will have three modules for the first partition, with 1 delay element (register) each. The second partition will have two modules with two registers each.

4.2 Design and Implementation of 2-D Applications

The fields of two-dimensional Digital Signal Processing and Digital Image Processing have maintained tremendous vitality over the past two decades and there is every indication that this trend will continue. In this section, we pick from some specific two-dimensional filters [8] and illustrate how our model can be extended.

4.2.1 The Laplacian Filter

Laplacian Filter is an important tool in the area of Digital Image Processing and is used for Edge Detection. The filter can be defined as a 3×3 kernel and depending on the image and the requirements, the kernel values are chosen accordingly. The pixels from the image are processed

through this filter to obtain the desired result. One such Laplacian filter is [7]:

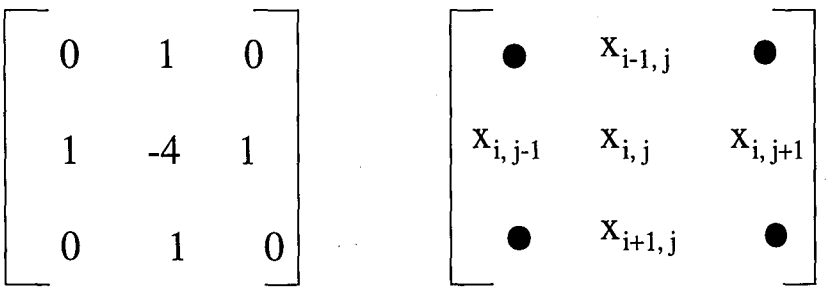
$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Since the Laplacian is a two dimensional filter, handling this architecture using the techniques discussed so far is not in keeping with the general concepts proposed. Hence, it cannot be treated as a one-dimensional system. Unlike in the previous problems, here there are n^2 output points and hardware complexity would also be of the order of n^2.

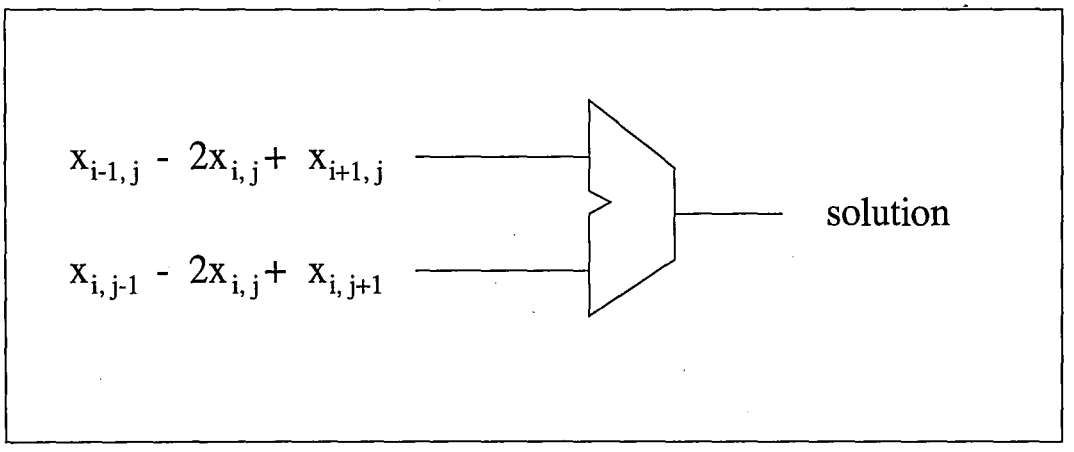
However, since it is a two-dimensional system, it is naturally favourable to partitioning into two components. So we propose to use the generator polynomial separately for rows and columns. We would be operating now with *row modules* and *column modules.*

For a 3×3 part of an image on which laplacian filter is to be used, Fig 4.1 shows how the design methodology works. Part (a) is the Laplacian filter kernel. Part (b) denotes how the space in the image is defined. The i terms correspond to row pixels and the j terms correspond to the column pixels in the image. The image pixels corresponding to the non-zero values of the filter kernel are operated upon. So the row module handles

$$\begin{pmatrix} \bullet & \bullet & \bullet \\ 1 & -2 & 1 \\ \bullet & \bullet & \bullet \end{pmatrix} \quad \text{of} \quad \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



(a) (b)



(c)

Figure 4.1: Row and Column Modules for Laplacian Filter

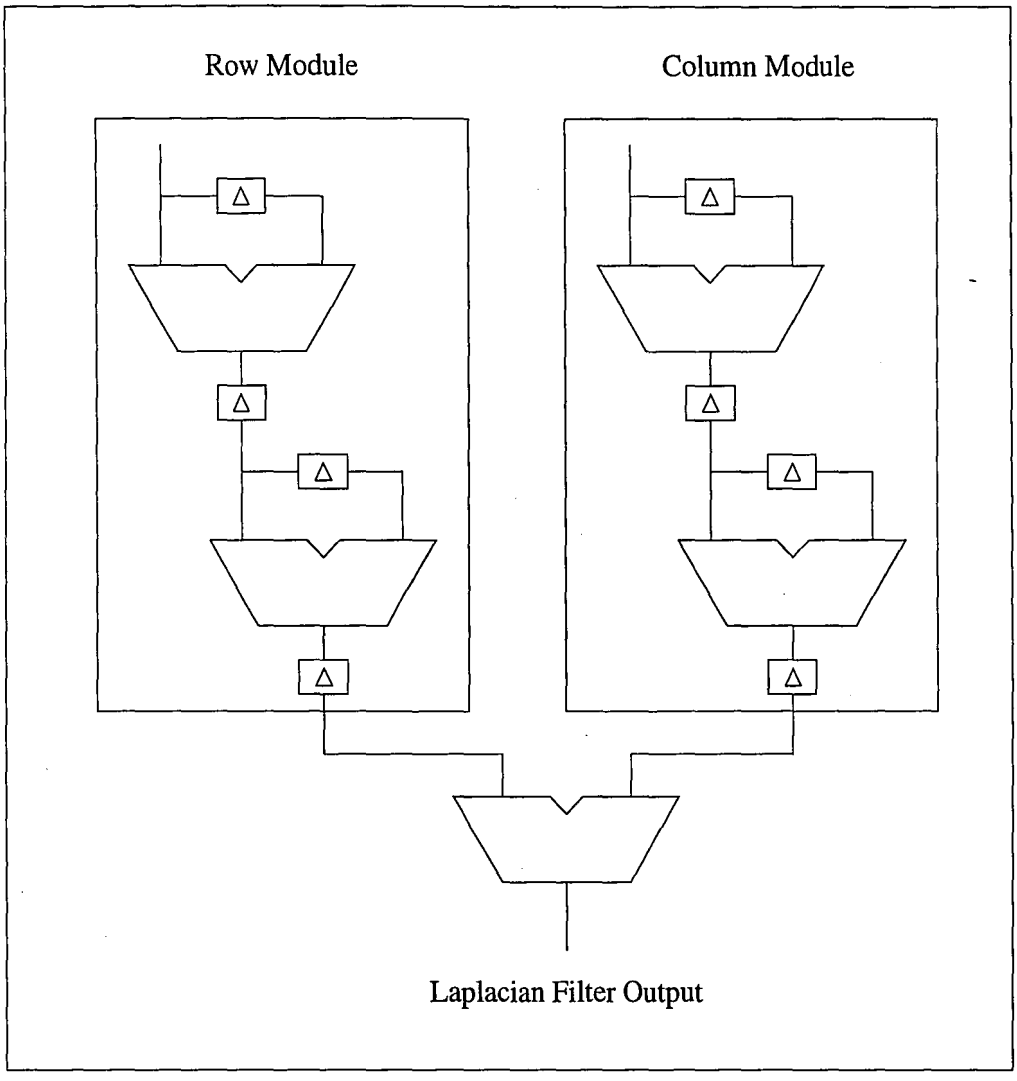


Figure 4.2: Implementation of Laplacian Filter

And the column module handles

$$\begin{pmatrix} \bullet & 1 & \bullet \\ \bullet & -2 & \bullet \\ \bullet & 1 & \bullet \end{pmatrix} \quad \text{of} \quad \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Part (c) of Fig 4.1 shows how the two modules interact to produce the filter result. The output

from the two modules are processed through an adder and this would result in a two-dimensional Laplacian Filter.

The overall architecture for to process one set of 3×3 image pixels is shown in Fig 4.2. Each of the modules consist of two adders. They are then tied together with a single adder to produce the desired result.

To operate on the complete image, one would require an array of $n \times n$ such architectures. This would mean a hardware complexity $\bigcirc(n^2)$ and time complexity $\bigcirc(n)$. One other possibility is to reduce the number of the modules and thus reduce hardware. However, this would result in a compromise in time complexity. The number of modules can be restricted to n which translates to a hardware complexity $\bigcirc(n)$ and time complexity $\bigcirc(n^2)$.

Chapter 5

Conclusion

In this thesis, we have presented a new architecture with optimal hardware and time complexities. The design approach proposed is simple and the application potential spans many varied fields. Our technique may be used to design and implement architectures of linear as well as nonlinear operation such as addition, maximum, minimum, and more. Operations that fall within the purview of our methodology are associative and commutative. The third chapter has been dedicated to illustrating the wide potential of our design methodology.

In order to study these systems, we have introduced the concept of the *Generator Polynomial*

$$g_n(z) = \sum_{i=0}^{n} c_i z^i$$

. An analysis of the generator polynomial yielded some interesting properties. These have been discussed in section 2.4 and are important since they are extensively referred to during our designs. It is imperative that these be satisfied in order for a system to be designed using this technique.

It might be worthwhile to highlight the how and where the generator polynomial influences the design. The overall structure of the architecture is dictated by the generator polynomial. The architecture has identical modules arranged serially and their functionality depends on the specificity of the operation being performed. Once the generator polynomial has been determined for that particular system, the algorithms *Factor* $(1+x^k)$ and *Complete Factorization* are applied. The result will decide if the generator polynomial fulfills the properties discussed in section 2.4.

We have proposed some heuristic algorithms - *Partition-I* and *Partition-II* in Chapter 4 toward determining design solutions when the generator polynomial does not satisfy the required properties. As illustrated in that chapter, these perform well in most cases. However, in some instances, they do not provide optimal architecture designs. Consider the following example:

Example 1. $P(x) = 1 + 2x + x^3 + x^4 + x^5$.

Let us apply Algorithm 4 (Partition-II) to this polynomial.

Initialization: Algorithm *Complete Factorization* yields no factors for $P(x)$. Therefore, $Q(x) = P(x)$.

Evaluation: Computing the polynomial $Q(x)$ at $x^k = \pm 1$ where, $1 \leq k \leq n$, we get:

x	$Q(x)$	*Possible Partition*
$x = 1$	5 ($\neq 2^p$)	None
$x^2 = 1$	6 ($\neq 2^p$)	None
$x^3 = 1$	6 ($\neq 2^p$)	None
$x^4 = 1$	6 ($\neq 2^p$)	None
$x^5 = 1$	6 ($\neq 2^p$)	None
$x = -1$	-2 ($= 2^1$)	$(1 + x)$
$x^2 = -1$	4 ($= 2^2$)	$(1 + x^2)$
$x^3 = -1$	0	$(1 + x^3)$
$x^4 = -1$	2 ($= 2^1$)	$(1 + x^4)$
$x^5 = -1$	4 ($2 = 2^2$)	$(1 + x^5)$

Check: For this problem, there are multiple possibilities. It has to be determined as to which combination of factors give rise to the optimal hardware design.

The various possible partitions are

$$P(x) = (1 + x^5) + x(1 + x^2) + x(1 + x^3)$$

$$P(x) = (1 + x)(1 + x^4) + x(1 + x^2)$$

$$P(x) = (1 + x)(1 + x^3) + x(1 + x^4)$$

Of these, the last structure has the lowest hardware and time complexities and hence is the most optimal one.

From the above example, it is clear that the algorithm has to be improved so as to encompass the last logic decision also. More work is required so that this design process may be perfected.

Another area open to extensions is that of $2-D$ applications. This was touched upon in Chapter 4 through the Laplacian filter. A generalized technique needs to be developed for design of $2-D$ systems.

Bibliography

[1] A. Aggoun, "Systolic arrays for digital filters," in *International Society for Optical Engineering*, vol. 3217, pp. 162–168, 1997.

[2] AT&T, *AT&T Application Specific Integrated Circuits : 1.25U CMOS Library Standard Cells and Function Blocks.*

[3] P. Erdos, "Some remarks on polynomials," *American Mathematical Society*, vol. 53, pp. 1169–1176, 1947.

[4] N. G. DeBruijn, "On the zeros of a polynomial and it's derivatives," in *Nederlands Akad. Wetensch Proceedings*, vol. 49, pp. 1037–1044, 1946.

[5] G. Polya and G. Szego, *Problems and Theorems in Analysis VI.* 1970.

[6] R. W. Hamming, *Digital Filters.* 1977.

[7] J. S. Lim, *Two-Dimensional Signal and Image Processing.* 1990.

[8] S. K. Mitra, *Digital Signal Processing - A Computer-based Approach.*

Vita

Anita Rao was born in Chennai, India, to C. S. Kalavathy and C. B. Sivasubramaniam. Her initial schooling was at Rosary Matriculation Higher Secondary School, Chennai and National College, Bangalore. She earned her Bachelor of Engineering Degree, majoring in Electrical and Electronics Engineering, from Regional Engineering College, Tiruchirapalli, India. A wish to pursue specialization in the Computer Engineering field brought her to Lehigh University, PA, USA. She currently lives in Chicago with her husband and will be working at ADI, IL.

END OF TITLE