Design Space Exploration for The Beamformer System

Smita Bakshi
Daniel D. Gajski

Technical Report #93-34

August 15, 1993

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

sbakshi@ics.uci.edu
gajski@uci.edu

Abstract

We present a design exploration strategy for the beamformer system, an example of a typical DSP system. In order to do so, we first define a parameterized design template for the beamformer and for a FIR filter, since the filtering operation is a part of the overall beamformer system. We then discuss some approaches for varying the design parameters for the filter and the beamformer system, under constraints imposed by technology or the designer.

Contents

1 Introduction 4

2 Beamformer problem description 5

3 FIR filter 8

3.1 Design templates and parameters 8

3.1.1 Parallelism parameters 15

3.1.2 Pipelining parameters 15

3.2 Design procedure 16

4 Beamformer system 18

4.1 Design template and parameters 19

4.2 Design procedure 20

4.2.1 Approach 1 22

4.2.2 Approach 2 25

5 Conclusions 28

List of Figures

1 An N-element, M-beam beamformer system 6
2 Dataflow graph of a $4th$-order FIR filter 9
3 FIR design template 10
4 FIR examples illustrating parallelism parameters 12
5 Accumulator functionality and structure 12
6 FIR example illustrating a two-level summation 13
7 General structure of a summation level 15
8 FIR examples illustrating pipelining parameters 15
9 An algorithm for recursively enumerating filter designs 17
10 Design template for beamformer system 19
11 Examples illustrating design template and parameters of beamformer system 20
12 Two approaches for varying parallelism and pipelining in the beamformer system 21
13 Illustration of first approach 24
14 Tradeoff curve obtained by first approach 25
15 Illustration of second approach 26
16 Tradeoff curve obtained by second approach 28

List of Tables

1 Implementations obtained using approach 1 25
2 Implementations obtained using approach 2 27

1 Introduction

In exploring the design space of a DSP system, there are four design features that need to be considered : *parallelism, pipelining, blocking, and customization* [1].

A design is said to be maximally parallel if it completely exploits the inherent parallelism in a specification and hence, at any time, computes as many operations in parallel as possible. Thus, from the time that all the inputs to produce a given output are available, a maximally parallel design is one that will produce it in the minimum possible time. Design parallelism is thus a measure of the operations computed in parallel. It is obvious that while parallelism improves design performance, it also results in relatively expensive designs.

The second design feature, pipelining, is another means of increasing design performance, for a relatively small overhead in terms of pipelining register costs. This feature is all the more significant for DSP computations since they are regular and repetitive in nature, and yield well to pipelining techniques.

Blocking refers to partitioning or slicing the entire computation into smaller blocks, where a block is defined as a group of operations that can be executed as one "indivisible" unit. The direction of blocking is an important feature of designs that have a distinguished direction of data flow. The optimal partition in such a design is usually along the flow of data, so that the partial results of a computation readily flow from one component to the other with minimum delay. This concept is explained in more detail in [1].

The fourth design feature, customization, refers to the mix of standard and custom components in the design. A standard component is a predesigned, off-the-shelf component which may not be tuned to the problem at hand. Custom components, on the other hand, are designed from scratch, and hence can be designed to fit a given set of requirements. Generally speaking, standard components may not be suited for performance-critical applications, though they may be relatively inexpensive.

Customization also refers to the mix of hardware and software solutions in a design, where the software solution typically involves programming a part of the design on a standard component, such as a DSP processor, while the hardware solution involves custom designing components, usually, for the performance critical sections of the design.

We introduce the beamformer, an example of a typical DSP system, and present a method for exploring its design space by varying parallelism, pipelining, and, to an extent, the direction of blocking the computation. It is to be noted, that we only address custom designs in this report.

We first describe a design procedure for the FIR filter since a FIR filter is one of the

operations in the beamformer, and also because the nature of the FIR filter operation is similar to the beamformer. For both the systems, FIR filter and the beamformer, we start by defining and illustrating a parameterized design template using several simple examples. Next, we explain the algorithms for generating implementations under the given design template.

Current CAD tools for the automated design of VLSI FIR filters, [2] [3], do explore the design space, but to a limited extent and usually, at a "lower level". For example, GENRIF [2], has a fixed architecture for a P-order filter, but it offers the option of using either a carry save adder or a carry ripple adder for the implementation of the multiplier and the adder. Thus, the designer does not have too many implementations to chose from. Our design approach for the FIR filter varies the number of components, their interconnection, and the pipelining in the design. This leads to a much larger spectrum of implementations. In this report, we have not explicitly dealt with varying component types; however, our design approach can easily be extended to include this search.

Section 2 gives an overview of the beamforming operation in terms of mathematical formulae. The templates and design procedures for the FIR filter and beamformer are presented in Sections 3 and 4 respectively.

2 Beamformer problem description

The beamforming problem is formally described by the following equations:

$$y_e^b(i) = \sum_{k=0}^{P-1} D_e(i-k)c_e^b(k) \quad (1)$$

$$R^b(i) = \sum_{e=1}^{N} y_e^b(i)\omega_e^b \quad (2)$$

Where,

$b \in (1 \ldots M)$ is the beam index,

$e \in (1 \ldots N)$ is the receiving element index,

i is the sample index, k is the filter coefficient index,

$D_e(i)$ is the ith digitized sample received at element e,

$c_e^b(k)$ is the kth filter coefficient for element e and beam b,

P is the order (or the number of taps) of the FIR filter,

$y_e^b(i)$ is the filtered output for the ith sample of element e and beam b,

ω_e^b is a complex phase correcting coefficient,

$R^b(i)$ is the resultant response for beam b, (this is the output of the beamformer)

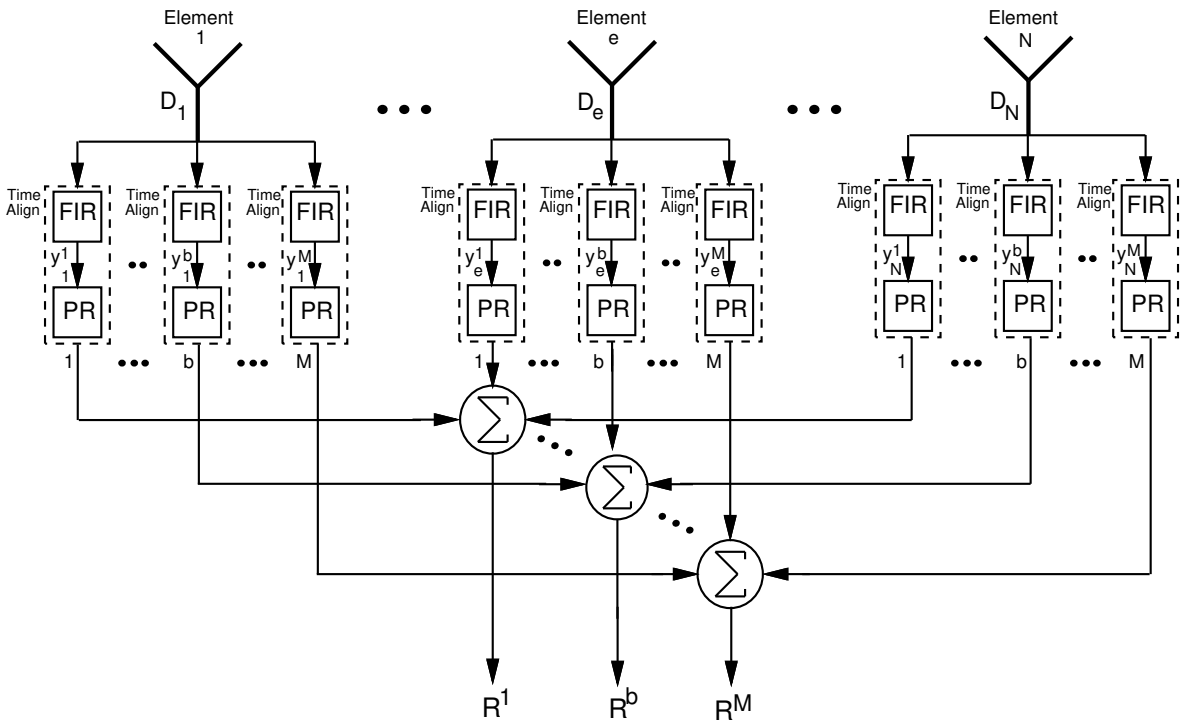

Figure 1: An N-element, M-beam beamformer system

N is the number of receiving elements in the beamformer, and

M is the number of beams to be formed.

The beamforming operation involves the temporal alignment and summation of digitized signals from several antenna elements. This is illustrated in Figure 1 for the case of an N-element antenna array. Each element receives a new sample every S ns, where S is the sample period of the beamformer system. Since the N elements in the antenna array are spread over a distance of the order of a few kilometers, the samples arriving at the elements at any time t, having travelled different distances, actually correspond to different times. Thus, before these samples are summed, as the beamformer operation requires, it is essential to interpolate the samples such that they all correspond to the same time instant. This interpolation operation is also referred to as time alignment, as indicated in Figure 1. It involves delaying the samples from each of the N elements by certain amounts, such that the samples are all aligned in time before being summed, that is, they correspond to the same time instant. A *beam* is then formed by summing the E time-aligned signals.

To illustrate this point, consider an array of 4 antenna elements each placed 1 kilometer (km) apart, and the object to be detected directly above, and in line with the first antenna. If the object is 1 km above the first antenna, then samples arriving at the first antenna

travel a distance of 1 km, those arriving at the second antenna antenna travel a distance of $\sqrt{2}$ km[1], at the third antenna, a distance of $\sqrt{5}$ km, and at the last antenna, a distance of $\sqrt{10}$ km. Thus, if all the elements are sampled at a time t, the sample at the first antenna will correspond to a time $t - \alpha$, where α is the time for the signal to travel a distance of 1 km, the sample at the second antenna will correspond to a time $t - \beta$, where β is the time for the signal to travel a distance of $\sqrt{2}$ km, and finally at the last antenna the sample corresponds to a time $t - \delta$, where δ is the time for the signal to travel a distance of $\sqrt{10}$ km. In order to detect the object more accurately, it is essential that before the samples from all the 4 elements are summed, they correspond to the same time. The interpolation operation ensures that they do.

An antenna array, such as the one in the beamformer, is an alternative to having a single larger rotating antenna for purposes of radar detection. Whereas a "conventional" antenna rotates mechanically, the rotation of the antenna array is achieved electronically by varying the delay values associated with each antenna. M independent directions or angles can be achieved by associating M different delay values with each element. Every S ns, the new samples arriving at *each* of the N elements are delayed by M different values. The corresponding delayed signals from all the N elements are then summed to form M resultant beams, R^1 to R^M.

As an example, if $M = 30$, the beamformer produces 30 independent beams, where each beam could be pointing in any arbitrary direction (i.e. the angle that the beam makes with respect to a fixed reference line could vary between 0° and 359°, where 360° corresponds to one complete rotation of the antenna array). As as aside, a larger N and M correspond to a larger tracking range and resolution of the antenna array.

The delay or time alignment operation involves two computations: 1) filtering, and 2) phase rotation (PR), performed in that order. The filtering is performed by using a P-order finite impulse response (FIR) filter. For each element, e, and beam b there are a set of P filter coefficients, denoted by $c_e^b(k), k \in 0 \ldots (P-1)$. The phase rotation operation for an element e and beam b simply requires multiplying the filtered signal, $y_e^b(i)$, with a phase correction coefficient, ω_e^b, associated with that element and beam.

Equation (1) describes the filtering operation and equation (2), the phase rotation and subsequent summation. It should be noted that the nature of the two equations is very similar (a sum of products). Thus, the design procedures for both the FIR filter and the complete beamformer system, are, in principle, the same. Sections 3 and 4 outline design

[1] Using Pythagoras' theorem for the length of the hypotenuse of a right angled triangle.

procedures for both the systems.

3 FIR filter

In this section, we define a design template for the FIR filter, and illustrate its parameters with the help of several examples. Furthermore, we describe an algorithm to automatically generate "all possible" designs for a given constraint on the throughput (or, sample period) of the filter.

3.1 Design templates and parameters

A Pth-order FIR computation, as described by equation (3), is essentially a summation of P products of successive input signals and coefficients. This summation is performed once every S ns, where S is the sample period of the filter, (i.e. $1/S$ is the arrival rate of samples at the filter input). If i represents time or a sample number, then the input sequence that is used for the ith summation ranges from the ith, to the $(i-P+1)th$ input samples.

$$y(i) = \sum_{k=0}^{P-1} x(i-k)b(k) \tag{3}$$

The input is represented by the x array, the filter coefficients by the b array, and the outputs by the y array. The array indices, i and k, represent the sample number and filter coefficient number respectively. The dataflow graph of a $4th$-order FIR filter is shown in Figure 2.

In general, the ith output depends on the $(i)th$, $(i-1)th$, ..., $(i-P+1)th$ inputs. Similarly, the $(i+1)th$ output depends on the $(i+1)th$, $(i)th$, ..., $(i-P+2)th$ inputs. There is thus a "flow" of input values across consecutive outputs. In effect, we can think of this as a shift operation, where for the $(i+1)th$ output, the $(i-P+1)th$ input is shifted out since it is no longer needed, and the $(i+1)th$ input is shifted in. All other inputs are "shifted right", such that if an input was multiplied by coefficient $b(j)$ in the ith iteration, it is multiplied by $b(j+1)$ in the $(i+1)th$ iteration. This is clearly demonstrated in Figure 2 for the input x_2.

In designing the FIR filter, we keep in mind the inherent input or data flow in the computation, and thus incorporate a design feature that supports the data flow. The template we chose directly reflects the filter operations: several products followed by their summation, as well as a shift of input values. The template, thus, consists of three distinct blocks or sections corresponding to each of the operations. These are labeled *shift, product,* and *summation* in Figure 3 (a).

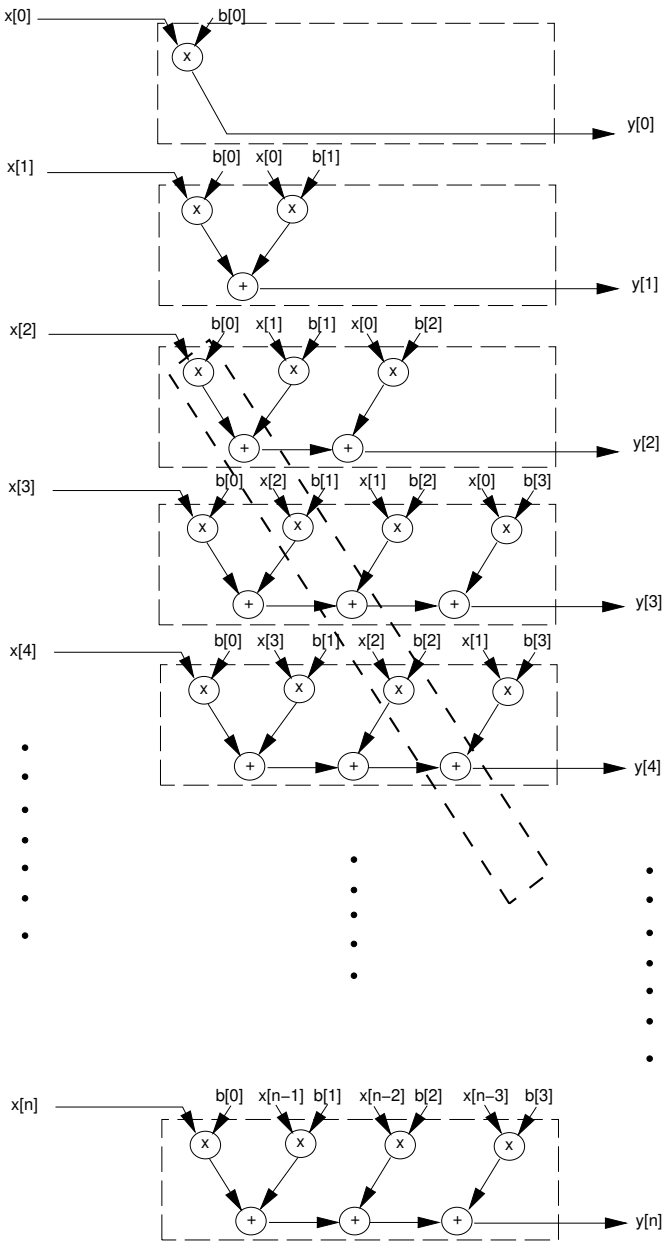

Figure 2: Dataflow graph of a $4th$-order FIR filter

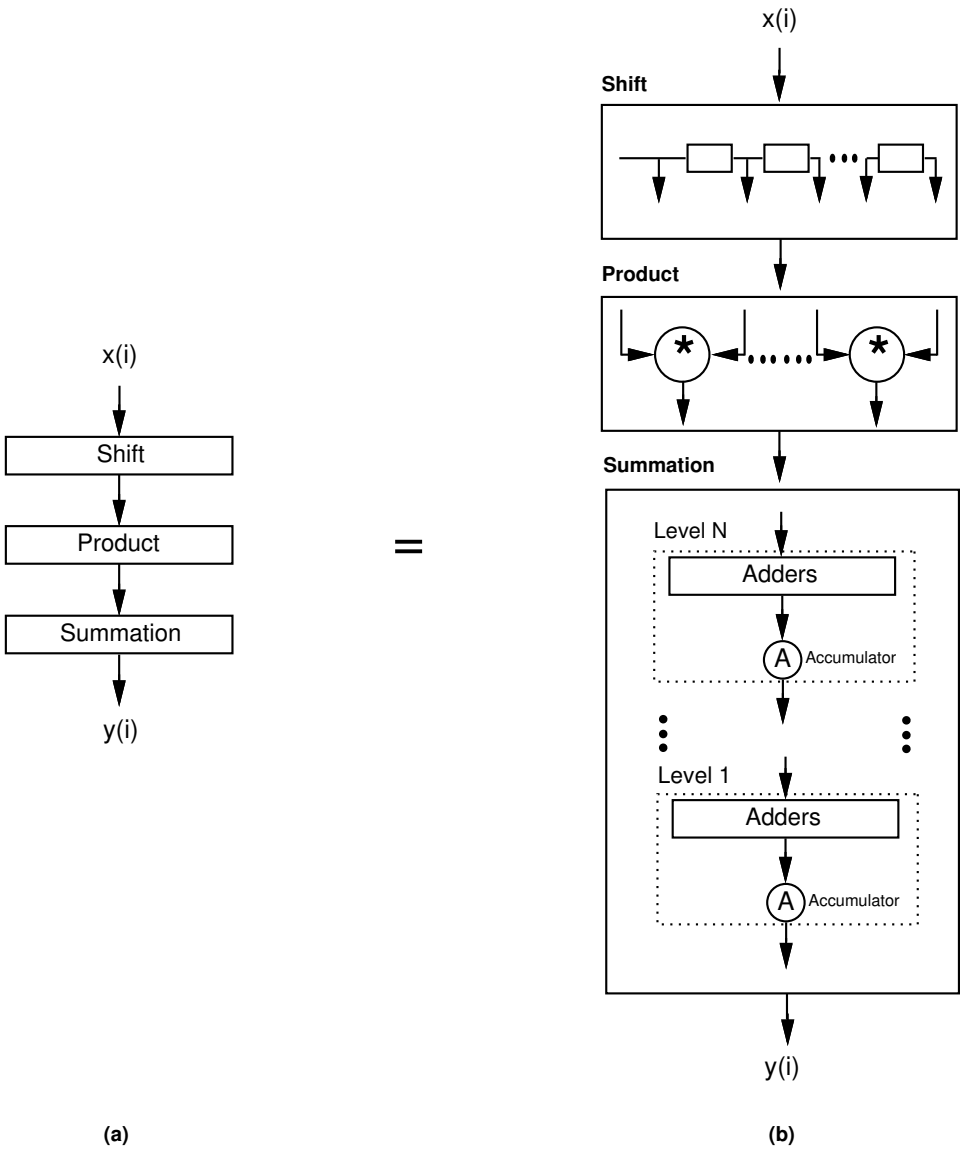

Figure 3: FIR design template

The shift block is required to shift the inputs in the manner described above, the product block is required to multiply the inputs with the filter coefficients and the summation block is required for summing the products. The shift block thus contains 1 to $P-1$ shift registers configured as in the shift block in Figure 3 (b), the product block contains a set of multipliers, and the summation block contains a set of adders and, possibly, accumulators. Different FIR implementations are, thus, obtained by varying the number of multipliers in the product block, and the number of adders and accumulators in the summation and their interconnection.

We first explain the summation block, since in our template, an FIR implementation is mostly specified by the structure of the summation block. We take an example of an $8th$-order FIR filter, where the ith output is given by

$$\begin{aligned} y(i) &= x(i)b(0)+x(i-1)b(1)+x(i-2)b(2)+x(i-3)b(3)+x(i-4)b(4) \\ &\quad +x(i-5)b(5)+x(i-6)b(6)+x(i-7)b(7) \end{aligned} \tag{4}$$

For simplicity, let us rewrite equation (4) as:

$$y(i) = p_0+p_1+p_2+p_3+p_4+p_5+p_6+p_7 \tag{5}$$

The job of the product block is to provide the products, $p_0 \ldots p_7$, and the job of the summation block is to sum these products. The summation can be done in one of many ways. Some of these are given in equations (6), (7), and (8).

$$y(i) = (((((p_0+p_1)+p_2)+p_3)+p_4)+p_5)+p_6)+p_7) \tag{6}$$

$$y(i) = ((((p_0+p_1)+(p_2+p_3))+(p_4+p_5))+(p_6+p_7)) \tag{7}$$

$$y(i) = (((p_0+p_1)+(p_2+p_3))+((p_4+p_5)+(p_6+p_7))) \tag{8}$$

The parenthesis indicate the order in which the operands are added. The corresponding hardware implementations for each of these equations is given in Figures 4 (a), (b) and (c) respectively. (Note that the summation block is enclosed within the dashed-box). We refer to the summation structure in Figure 4 (a) as a parallel sum, that in Figure 4 (c) as a serial sum, and the one in Figure 4 (b) as a hybrid. In general, for a Pth-order filter, there are $\log_2 P - 1$ such structures.

We have just explained what we mean by the structure of a summation. We now explain its "accumulation factor". However, before that we clarify the functionality of an accumulator. An accumulator is represented by the node labeled A in Figures 3 and 4, and it is elaborated in Figure 5. It consists of an adder and a register, with the output of the

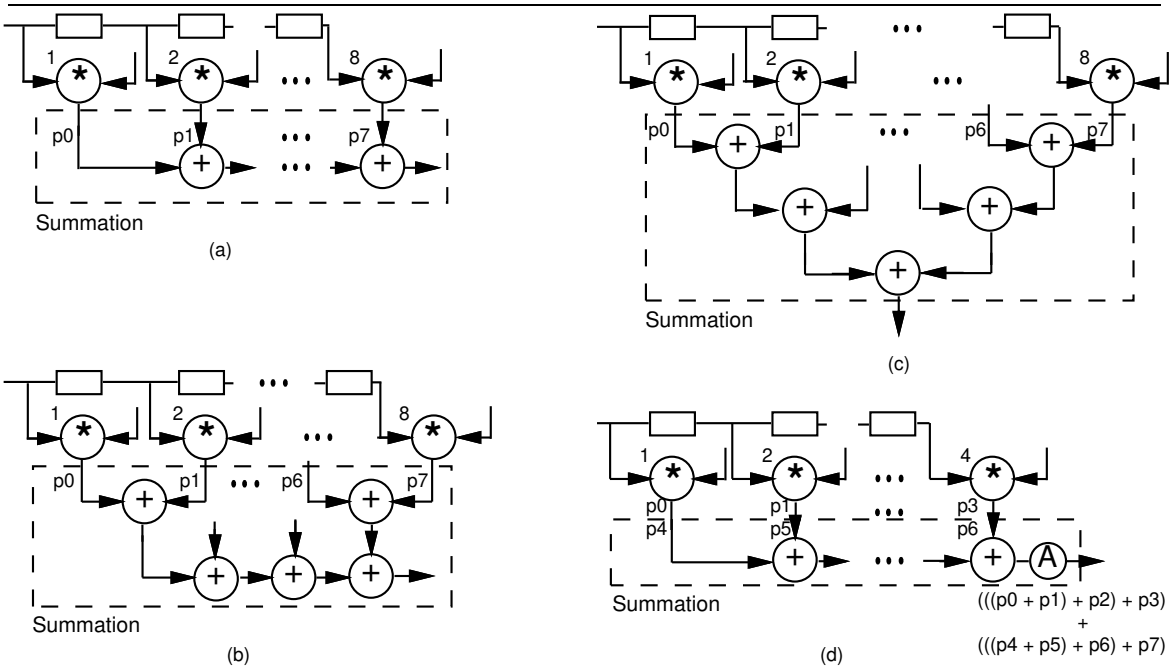

Figure 4: FIR examples illustrating parallelism parameters

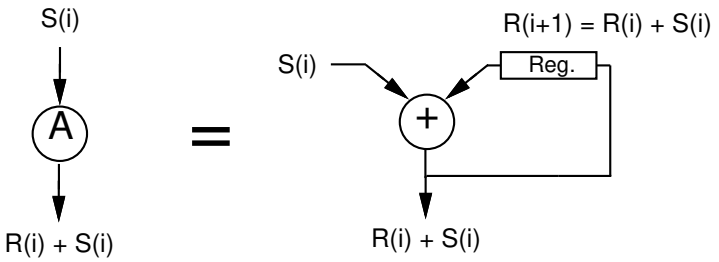

Figure 5: Accumulator functionality and structure

adder connected to the input of the register, and the output of the register connected to one of the inputs of the adder. Thus, if $S(i)$ is the input to the adder at the ith clock pulse, and $R(i)$ the contents of the register also at the ith clock pulse, then the contents of the register at the $(i+1)th$ clock pulse will be $R(i+1) = R(i) + S(i)$.

For each of the implementations in Figures 4 (a), (b) and (c), we had 7 adders. If we now reduce the number of adders, the computation will be done in two or more "batches" or accumulations. For example, if we have only three adders and an accumulator, as in Figure 4 (d), we can first compute $(((p_0+p_1)+p_2)+p_3)$, store the result temporarily in the accumulator, then compute $(((p_4+p_5)+p_6)+p_7)$, and add this result to the previous one. In this example, the addition is thus performed in two batches or accumulations. We can, likewise, do the addition in four accumulations by having one adder and an accumulator, or in eight accumulations by having just an accumulator.

Next, we illustrate a third feature of the summation block, which we refer to as the number of levels of accumulation. All the implementations considered so far are one-level structures. We illustrate a two-level structure in Figure 6 (a). For purposes of comparison we also illustrate a one-level structure (Figure 6 (b)), that may appear to be similar to the two-level structure.

In the first step, adder S_1 performs $(p_0 + p_1)$ and adder S_2 performs $(p_2 + p_3)$. The result of these additions is stored in accumulators A_1 and A_2 respectively. In the next step, operations (p_4+p_5) and (p_6+p_7) are performed using adders S_1 and S_2 and their results are accumulated in A_1 and A_2 respectively. The final result is obtained by adding the partial results in A_1 and A_2 using S_3.

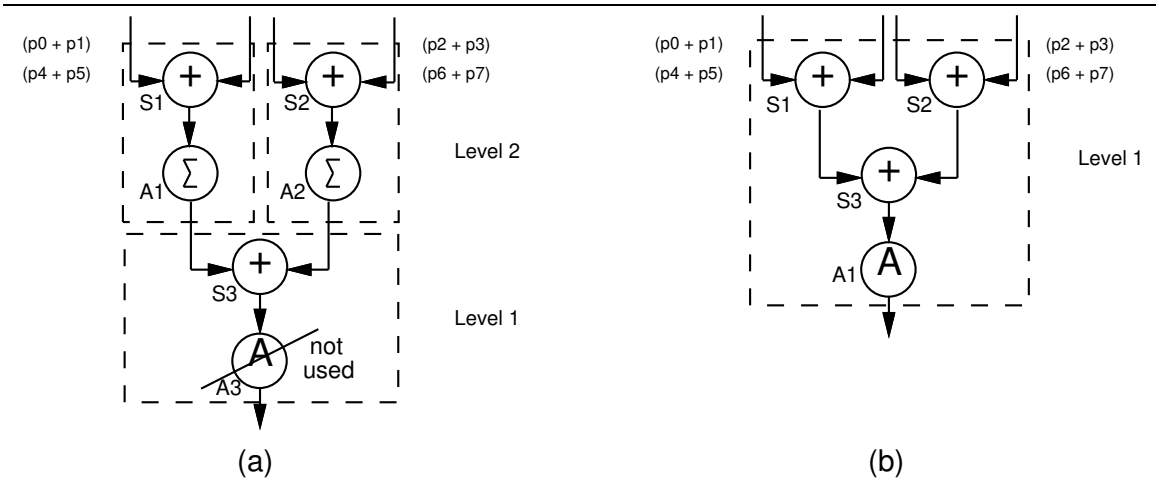

Figure 6: FIR example illustrating a two-level summation

As a comparison, in the one-level implementation, adder S_1 performs (p_0+p_1), adder S_2 performs (p_2+p_3), and adder S_3 performs $((p_0+p_1)+(p_2+p_3))$, in the first step. This partial result is stored in the accumulator and in the second step it is added to $((p_4+p_5)+(p_6+p_7))$, which is computed by adders S_1, S_2, and S_3. If we assume that accumulations and additions take one time unit each, then the two level summation requires 5 time units, while the one-level summation requires 6 time units.

In general, for a Pth-order summation the number of levels can vary from 1 to $log_2 P$.

It is to be noted that for a given summation block, the shift and product blocks are fixed, i.e, the number of shift registers and multipliers is directly determined by the summation block. For example, for a one level summation of 7 adders, the product block will necessarily contain 8 multipliers. This is illustrated in the examples in Figure 4.

Thus, the design template can be completely specified by a set of 5-tuples for the summation block and one pipelining parameter for the product block. Each element of the 5-tuple set corresponds to a level of the summation block. The parameters are represented by:

$$\begin{aligned} Summation &: (S_i, A_i, T_i, P_{s_i}, P_{p_i}) \\ Product &: P_* \end{aligned}$$

Where,

$i \in (1 \ldots \log_2 P)$: summation level index

$S_i \in (0 \ldots P-1)$: sum size (or the number of additions performed),

$A_i \in (0 \ldots P-1)$: accumulation size (or the number of accumulations performed),

$T_i \in (1 \ldots log_2 P)$: sum structure factor (or the number of "completely balanced levels" + 1)

P_{s_i}: "serial-sum" pipeline factor,

P_{p_i}: "parallel-sum" pipeline factor, and

P_*: product pipeline factor.

S_i, A_i, and T_i, specify the parallelism in the design while P_{s_i}, P_{p_i} and P_* specify the extent of pipelining. Each of these factors is now explained by taking implementations of the $8th$-order filter in Figure 4.

3.1.1 Parallelism parameters

S_i gives the total number of additions that are performed by the *ith* summation level, and A_i, the number of accumulations that are required in order to add the S_i data values. Thus, for the implementations given in Figures 4 (a), (b), and (c) the sum size is 8, and the accumulation size is 1. In these implementations the accumulator serves no purpose (or, it is used just once), while in Figure 4 (d), the accumulator is used twice. In this implementation the sum size is still 8, while the accumulation size is now 2. In effect, A_i indicates the number of iterations through the summation in order to complete the S_i additions.

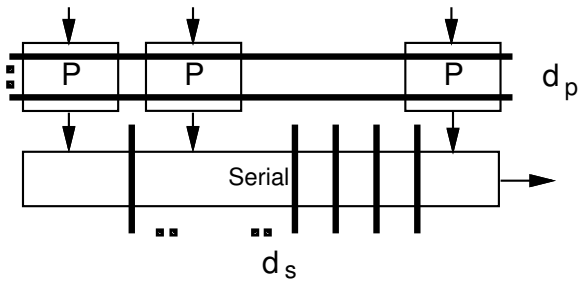

Figure 7: General structure of a summation level

The general form of the summation structure is shown in Figure 7. At one extreme, the size of the parallel block is zero, and we have a completely serial structure (Figure 4 (a)). At the other extreme, the serial block contains just one adder, and we have a completely balanced binary tree structure (Figure 4 (c). The parameter T_i represents the number of levels in the parallel block. Thus the structure factor in the implementations of Figures 4 (a), (b), and (c) is 1, 2 and 3 respectively.

3.1.2 Pipelining parameters

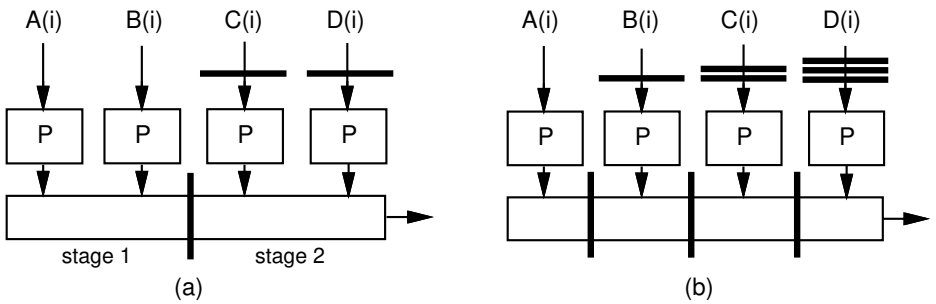

Figure 8: FIR examples illustrating pipelining parameters

We illustrate the summation pipelining factors, P_{s_i} and P_{p_i}, by considering a general summation block as shown in Figure 7. The delay, and hence the throughput, of the non-pipelined structure is $d_p + d_s$, where d_p is the delay of the parallel block and d_s is the delay of the serial block. After pipelining the parallel block into P_p equal stages, and the serial block into P_s equal stages, the throughput reduces to $d_p/P_p + d_s/P_s$.

Pipelining of the serial block requires us to introduce appropriate delay elements (or latches) at the inputs of the parallel block. (These latches do not have to be added at the inputs; they could also be introduced within the appropriate parallel blocks or at the output of the parallel block). For example, if we pipeline the serial block shown in Figure 8 into two equal stages, the inputs C(i) and D(i) are required only one clock cycle after A(i) and B(i). At any clock pulse, as stage 2 is completing the ith sum, stage 1 is starting the $(i+1)th$ sum. In order to maintain correct operation of the filter, two latches are thus introduced at inputs $C(i)$ and $D(i)$.

As we increase the pipelining of the serial block, latches are introduced to "skew" each of the inputs by appropriate amounts. In Figure 8 the serial block is pipelined into four stages; 2, 3 and 4 latches are thus required at inputs B(i), C(i), and D(i) respectively.

Finally, P_* indicates the number of pipeline stages in the multipliers constituting the product block. The parameter P_* is required because the number of pipeline stages in the product block is independent of the summation block. Whereas the number of multipliers in the product block is completely determined by the summation structure, the number of pipeline stages in the product block is an orthogonal feature.

3.2 Design procedure

To summarize so far, we have just defined an FIR design template consisting of three blocks: shift, product, and summation. The template is parameterized by a set of 5-tuples for the summation block and a pipelining parameter for the product block. We now give a procedure for varying these parameters so as to generate all possible FIR implementations for a given throughput constraint, C.

In explaining the algorithm, we use the same notation introduced earlier for the parameters; however, for the sake of convenience, we drop the subscript i from the notation. Thus, for a summation level, the parameters are (S, A, T, P_s, P_p), where S indicates the sum size, A the accumulation size, T the structure factor, and P_s and P_p the pipelining factors.

The recursive algorithm $FIR()$ takes as its parameter P, the order of the filter. It starts by initializing S to P, A to unity, and T to $\log S$ (which corresponds to a balanced

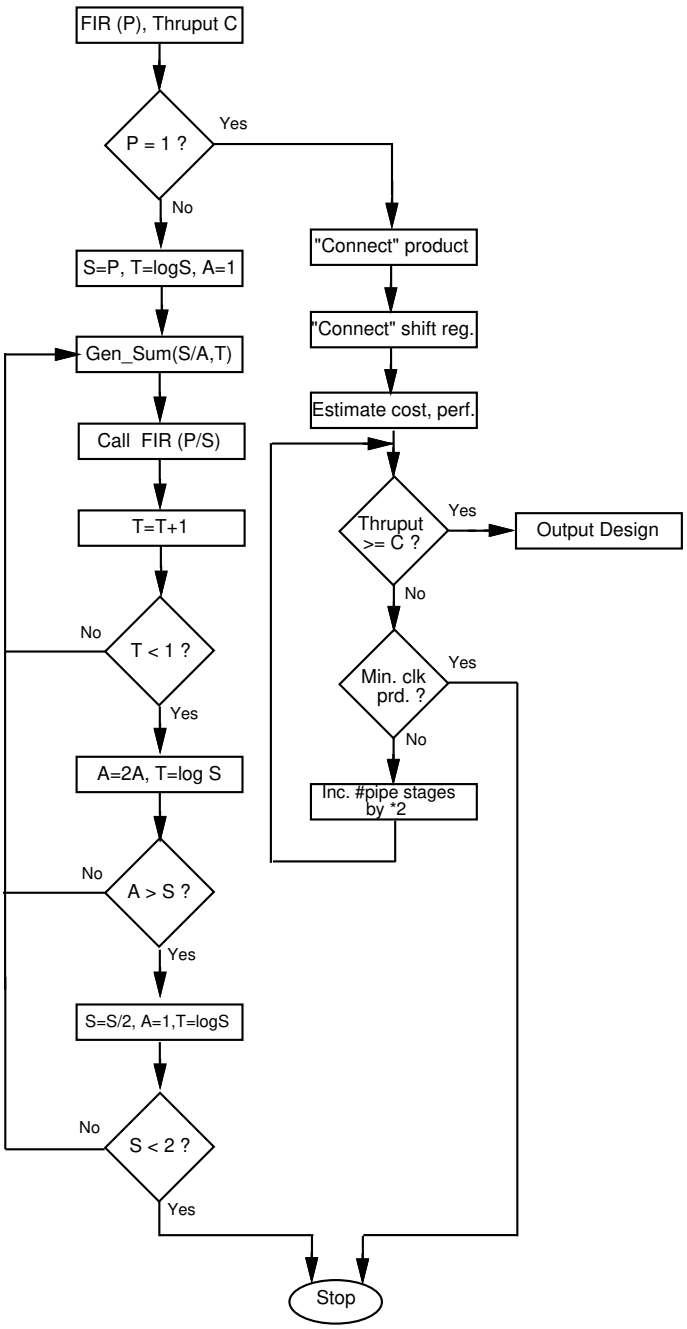

Figure 9: An algorithm for recursively enumerating filter designs

tree structure). It then makes a call to the procedure Gen_Sum with the parameters S/A and T. This procedure returns a one-level summation of the form shown in Figure 7. T is the number of levels in the parallel block and S/A is effectively the number of inputs in the summation structure. Next, a recursive call is made to the $FIR()$ procedure with the parameter P/S. This is how multiple summation levels are generated. For example, if $P = 8$, in the first iteration a one-level tree would be generated since S is set to P, $P/S = 1$, and thus a call to FIR(1) would result in the execution of it's base case.

The base case of the recursive algorithm FIR is reached when the summation block is completed and the $FIR()$ algorithm is called with its parameter set to unity. The product and shift blocks are then "connected", and the performance of the resultant design is estimated. If the design does not satisfy the throughput constraint it is pipelined in incremental steps till the minimum clock period is reached or the throughput constraint is met. Pipelining the design involves varying the three pipelining parameters, P_s, P_p, and P_*. This has not been shown in the flowchart. If the design still cannot satisfy the throughput requirement after it is maximally pipelined, then no further FIR implementations are generated and the algorithm terminates. This is because the algorithm starts from the most parallel end of the design spectrum and proceeds towards the serial end. Thus, if a design cannot satisfy throughput requirements, neither will subsequent designs generated by the algorithm.

The calls to the two procedures, Gen_Sum and FIR, are nested within three loops. The inner most loop generates all possible summation structures (from parallel to completely serial) for a given S, sum size and A, accumulation size. The second loop varies A from $1 \ldots S$ for a given S, and the third loop varies S from P down to 2 for a given P.

Thus, the summation blocks are generated recursively and the three parallelism parameters are varied using the three loops. Also, for a given implementation, the pipelining parameters are varied using a triple nested loop. In this way, the algorithm generates all FIR implementations that satisfy a given throughput requirement. It is to be noted that the algorithm can be easily modified to accommodate a different set of constraints, such as minimum cost or latency.

4 Beamformer system

In this section, we define a parameterized template for the beamformer system and a procedure for obtaining a cost-effective implementation under a given set of constraints.

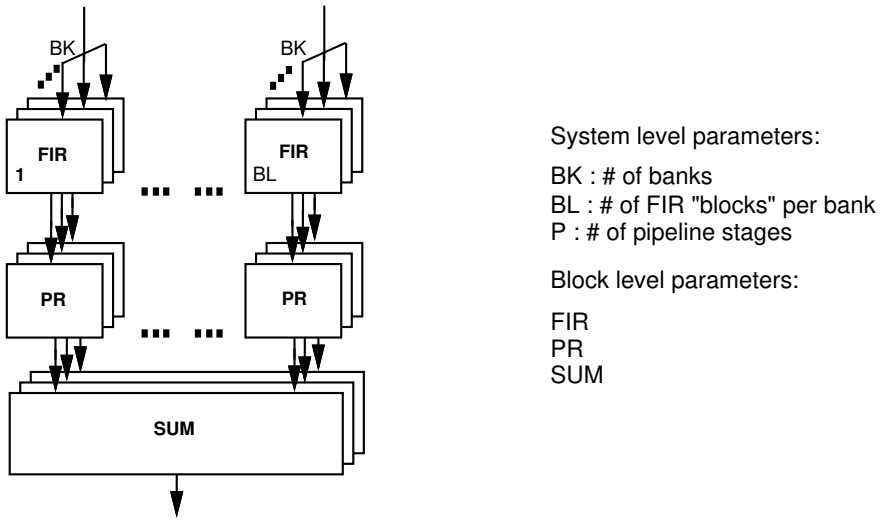

Figure 10: Design template for beamformer system

4.1 Design template and parameters

The design template shown in Figure 10 contains three distinct blocks: *FIR*, *PR*, and *SUM*. The FIR block is used to compute equation (1), and the PR and SUM blocks compute equation (2). However, one or more of these blocks can be merged, for example PR can be merged with either FIR or SUM, without significantly affecting the parameters or the design procedure.

As Figure 10 clearly shows, the beamformer template has a two level hierarchy. The top (or system) level is an interconnection of FIR, PR and SUM blocks, and the bottom (or block) level is an interconnection of adders, multipliers and registers. Corresponding to each level we define a set of parameters:

1. System level:
   (a) BK : number of BanKs,
   (b) BL : number of FIR BLocks per bank, and
   (c) P : number of Pipeline stages (at the system level).
2. Block level:
   (a) FIR parameters
   (b) PR parameters
   (c) SUM parameters

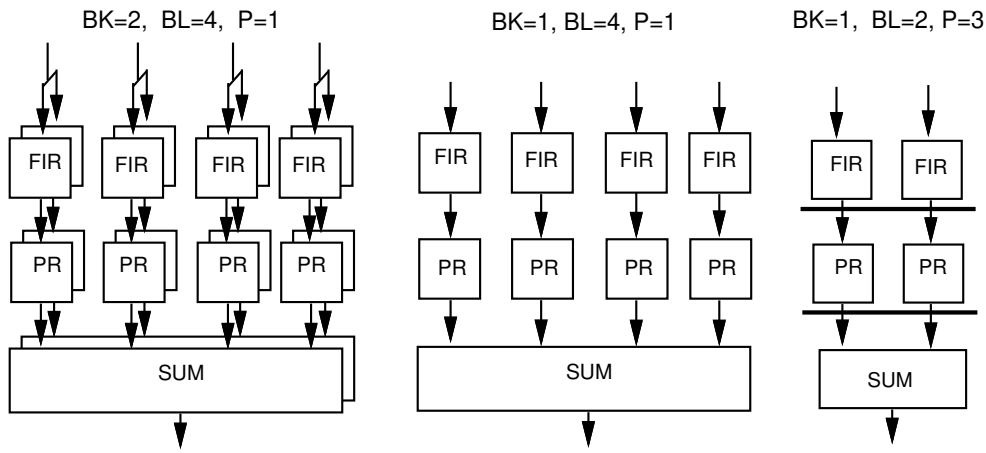

Figure 11: Examples illustrating design template and parameters of beamformer system

The examples in Figure 11 illustrate the system level parameters. A bank refers to one "plane" of interconnected FIR, PR and SUM blocks. Thus the first example has two banks, while the other two have just one bank each. The size of a bank is given in terms of the number of FIR blocks, and this is refered to as the block size (BL). BL varies from 4 to 2 in the examples.

The FIR computation (sum of products) can be viewed as a superset of the PR (products) and SUM computations; hence, the design template and parameters given in Section 2 suffice as an explanation for the block level parameters.

4.2 Design procedure

We now describe a procedure for obtaining a minimum cost beamformer implementation by varying parallelism and pipelining. This variation is limited by constraints on the sample period (S) and the latency (L) of the system, as well as the minimum clock period, a constraint imposed by technology. The sample period (in ns) is the rate at which digitized samples are sent to the FIR block(s), and the latency (in number of sample periods) is the total time for an input to be processed.

We present two design approaches, both of which lead to the same (or very similar) implementations for a given set of constraints. In the first approach, we trade cost for sample period, while keeping the clock rate constant. This is essentially done by serializing the design, such that design costs drop and the sample period increases due to the decrease in the number of components in the design. In the second approach we keep the sample period of the design constant, and trade cost for the clock rate, where a decrease in the clock rate leads to a drop in design costs.

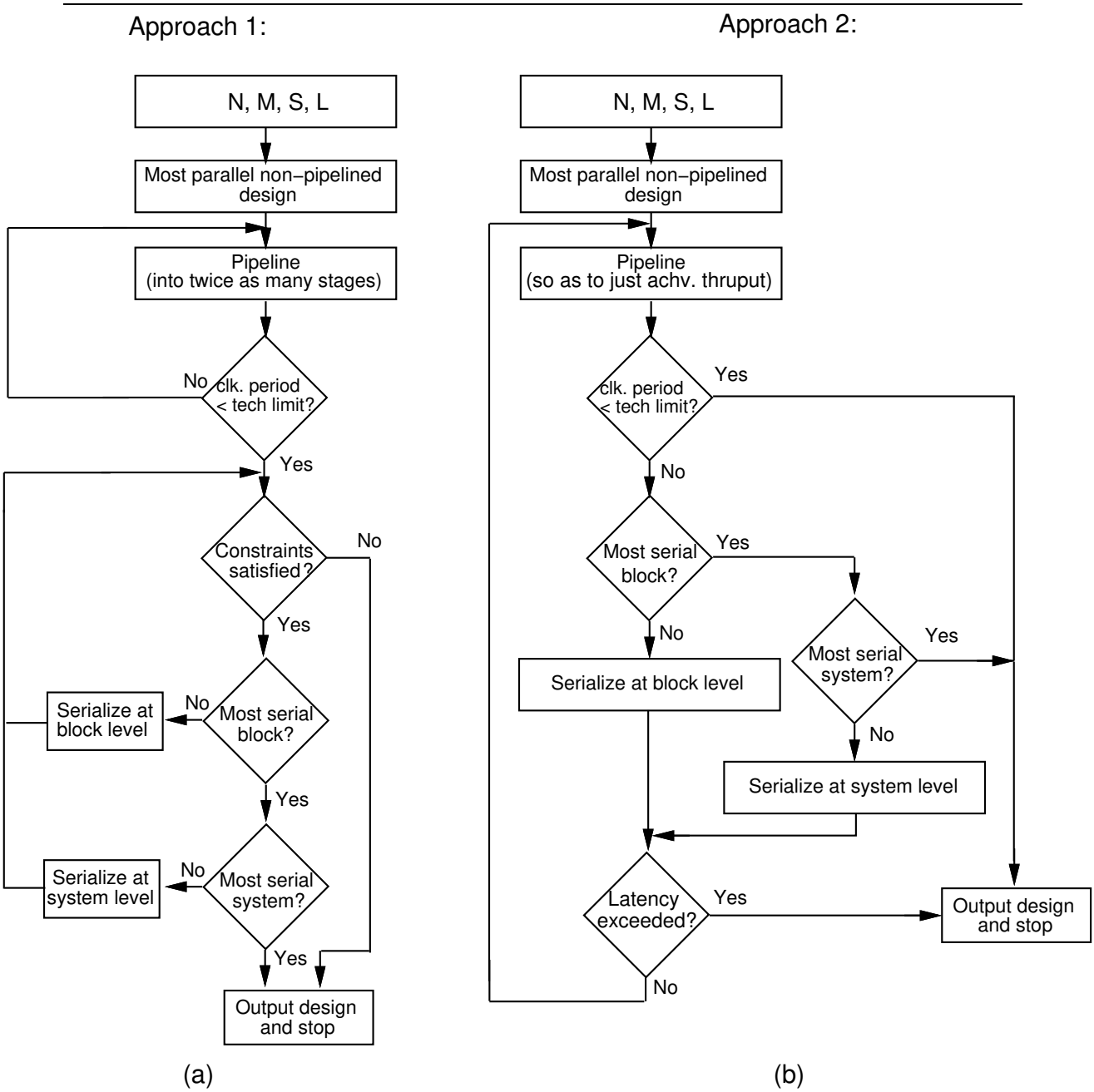

Figure 12: Two approaches for varying parallelism and pipelining in the beamformer system

Both these approaches are explained by taking an example with the following specifications:

1. N, the number of elements = 512
2. M, the number of beams = 4
3. S, the sample period = 512 ns
4. L, the latency limit = 4 sample periods.

4.2.1 Approach 1

This approach is illustrated in Figure 12 (a). We start with the most parallel non-pipelined design (at the system and block levels), and pipeline it till the minimum clock period is reached. Then we serialize it while making sure that constraints are still satisfied.

At the system level a most parallel design is one in which $BK = M$ (one bank per beam), and $BL = N$ (one FIR block per element). At the block level, the FIR, PR and SUM blocks also have the most parallel implementations. For example, for a P-order filter, the FIR block will have P multipliers and a summation tree of P-1 adders. This can be obtained by making a call to the $FIR()$ procedure (outlined in the previous section), and selecting the most parallel design that meets the performance constraint of S. The $FIR()$ procedure not only returns the implementation, but also an estimate of its cost and execution time. The most parallel PR and SUM implementations can also be obtained by making calls to similar procedures. These procedures are not given in this report since they are almost identical to the procedure for the FIR block.

The maximally parallel design is then pipelined both at the system and block levels. At the system level, a maximally pipelined beamformer simply contains three pipeline stages, the FIR, PR and SUM stages. At the block level, the number of stages in each block depends on the execution time of the block and the minimum allowable clock rate in the given technology.

The next step in the design process consists of serializing the maximally parallel and pipelined design (and hence, the most expensive one in the design spectrum), till the constraints on S and L are just satisfied. The resultant design, is thus always maximally pipelined. Since pipelining increases the performance for a relatively small increase in cost, by exploiting pipelining we always obtain the minimum cost implementation.

In order to search the design space exhaustively, serialization has to be performed at the block and system levels. Serialization can be viewed as consisting of a double nested

loop, where the inner loop performs block serialization, and the outer loop serializes at the system level. That is, serialization is first performed at the block level until the throughput or latency constraints are no longer met. Then the *system* is serialized by a factor of two, and the block serialization is repeated.

We would further like to point out that in serializing at the system level, the number of banks is first reduced by factors of two, and when the design is down to one bank, the number of FIR blocks is reduced by factors of two. Serializing at the FIR level, refers to varying the three parallelism parameters mentioned in the previous section, in an order of decreasing parallelism. That is, successive FIR designs will have a fewer number of multipliers or a more serial adder structure.

Figure 13 shows a few designs that are obtained by using this approach on the example mentioned earlier. We assume certain delay values for each of the blocks, and also a technological limit of 64 ns on the clock period.) In our evaluations, we also assume that serializing an FIR implementation by a factor of two results in a design with half the number of multipliers and double the execution time. This is a simplifying assumption, but is sufficient to illustrate the design approach.

Design #1 is the most parallel and pipelined design. The total execution time (or the critical path) of the implementation is 1024 ns, and with 16 pipeline stages, the clock period is 64 ns. However, the circuit can support a sample period of 64 ns - this is far greater than the constraint of 512 ns. This design is then serialized at the block level, that is, the FIR block is serialized by a factor of two. The delay of the FIR block thus increases to 768 ns from 384 ns. This is the second design we obtain (not shown in the figure). Also note that the clock period is the same as before, though the sample rate increases to 128 ns. When the FIR block is further serialized, the latency of the resulting design (4.25), exceeds the user-imposed constraint of 4.0 sample periods. The serialization step thus falls out of the inner loop of block serialization, and executes the outer loop of system serialization. The number of banks now reduces from 2 to 4. This is Design #3, and it is depicted in Figure13(b). Design #3 is then serialized at the block level, resulting in Design #4. Design #5 and #7 shown in Figures13(c) and (d) correspond to the implementations obtained after further serialization at the system level, and Design #6 is obtained after serializing Design #5 at the block level.

Table 1 gives an approximate analysis of the designs and Figure 14 (a) graphically depicts the seven design points. Sample period is in ns, latency in number of sample periods, and cost is in terms of adder units, where the cost of a multiplier is 4 adder units, and the cost

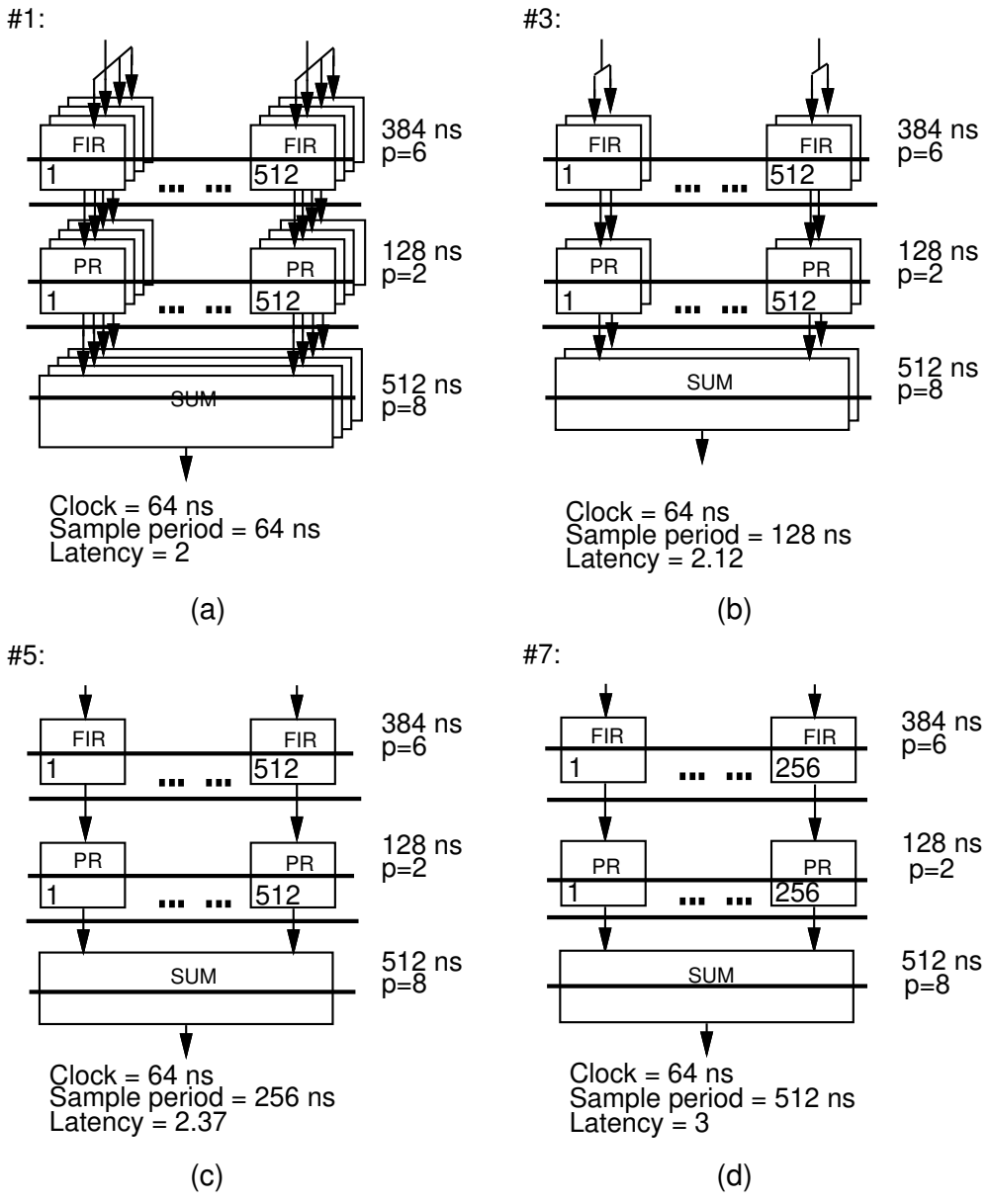

Figure 13: Illustration of first approach

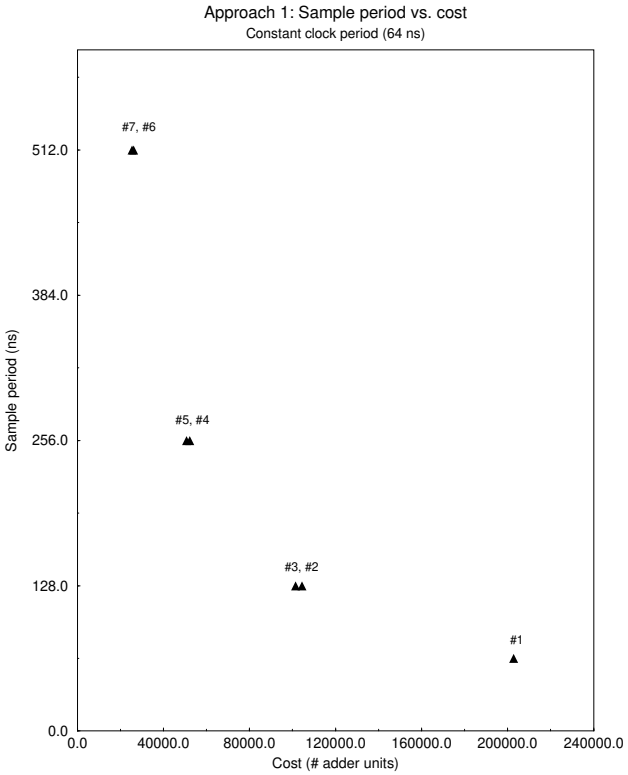

Figure 14: Tradeoff curve obtained by first approach

of a register is one adder unit.

We would like to point out that in the example discussed above only the FIR block was serialized, and that too in a limited manner. We also did not take into account the overhead in cost or delay due to pipelining. This was to reduce the complexity of the estimations since they are all manually derived.

4.2.2 Approach 2

Design #	BK # of banks	BL # of blocks	FIR Delay (ns)	Cost # of adders	Sample period (ns)	Latency # sample periods
1	4	512	384	202748	64	2.00
2	4	512	768	104444	128	2.75
3	2	512	384	101374	128	2.13
4	2	512	768	52222	256	2.88
5	1	512	384	50687	256	2.37
6	1	512	768	26111	512	3.13
7	1	256	384	25344	512	3.00

Table 1: Implementations obtained using approach 1

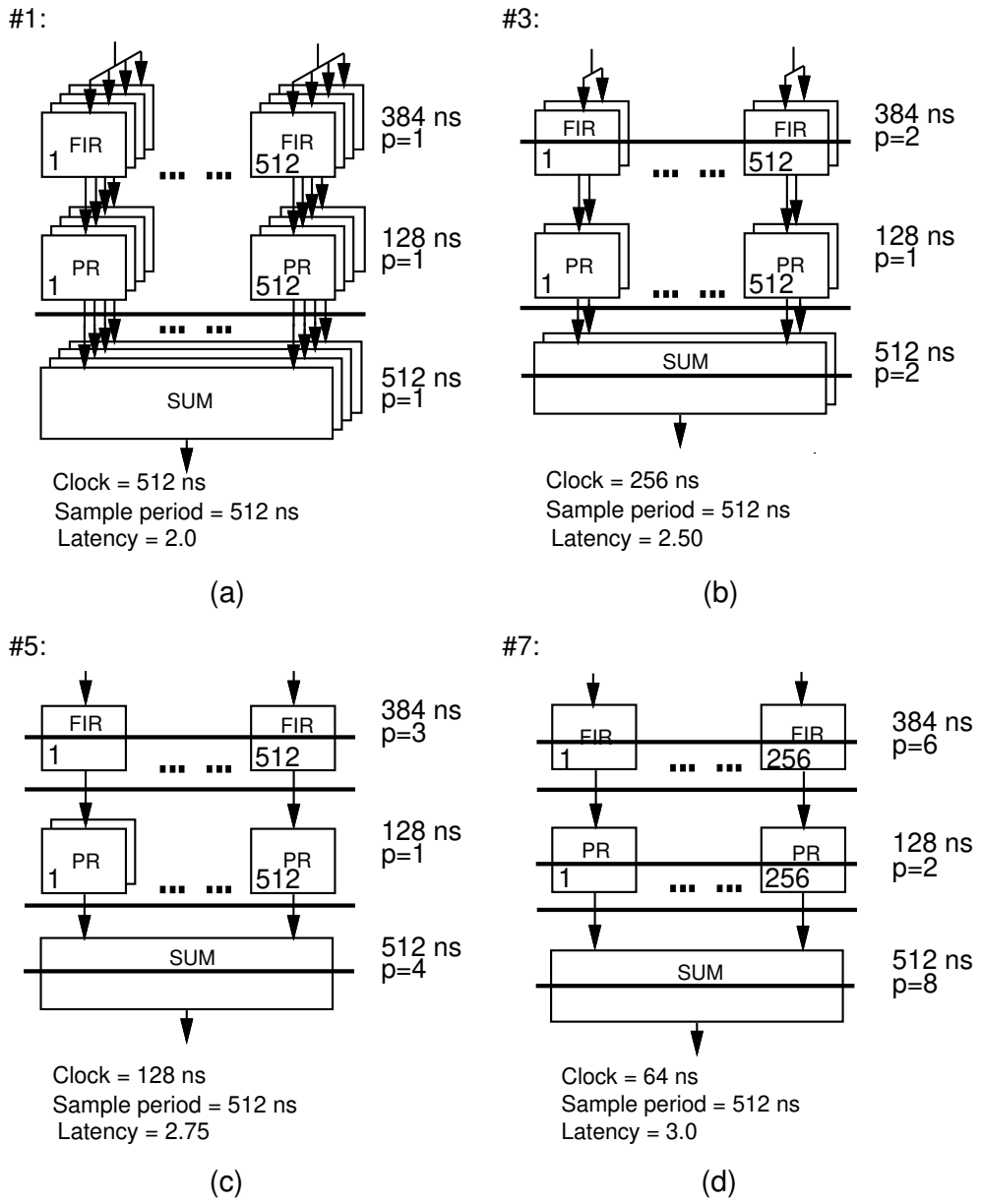

Figure 15: Illustration of second approach

Design #	BK # of banks	BL # of blocks	FIR Delay (ns)	Cost # of adders	Clock period (ns)	Latency # sample periods
1	4	512	384	202748	512	2.00
2	4	512	384	104444	256	2.75
3	2	512	384	101374	256	2.50
4	2	512	384	52222	128	3.25
5	1	512	384	50687	128	2.75
6	1	512	384	26111	64	3.50
7	1	256	384	25344	64	3.00

Table 2: Implementations obtained using approach 2

In the first approach, the most pipelined and parallel design is serialized till either of the constraints are violated. In this approach (Figure 12 (b)), a design is pipelined till design constraints are violated or the minimum clock period is reached. Once again, we start with the most parallel non-pipelined design and pipeline it just enough to satisfy the performance constraint. This design is then serialized, and for it to meet the performance requirement its pipelining is increased. This process of serialization and pipelining is carried on till we obtain the most serial design, till the latency constraint is violated, or till the technology limit is reached.

Once again, serialization can be viewed as a double nested loop, where the inner loop corresponds to block level serialization and the outer loop to system level serialization.

We demonstrate this approach on the same example given above. We start with the most parallel implementation (Design #1 in Figure 15) pipelined into two equal stages of 512 ns each, and proceed towards more serial and pipelined designs. In Design #2 the FIR is serialized by a factor of two. In order for this design to satisfy the sample period requirement, its clock period is reduced to 256 ns. Design #2 cannot be serialized any further at the block level since its latency exceeds the user given constraint of 4 sample periods. Serialization is then performed at the system level and this results in Design #3. Once again, Design #3 is serialized at the block level, resulting in Design #4. This process of block and system level serialization goes on until Design #7 when the technological limit of 64 ns is met, and the design cannot be serialized any further.

Table 2 gives details of each of the designs. The clock period vs. cost tradeoff is depicted in Figure 16 for each of these designs.

We would like to point out that, typically, a user imposes cost or performance constraints on the *overall* system, and in a hierarchical system such as the beamformer, this corresponds to constraints on the top level of hierarchy. These constraints have to be propagated to lower

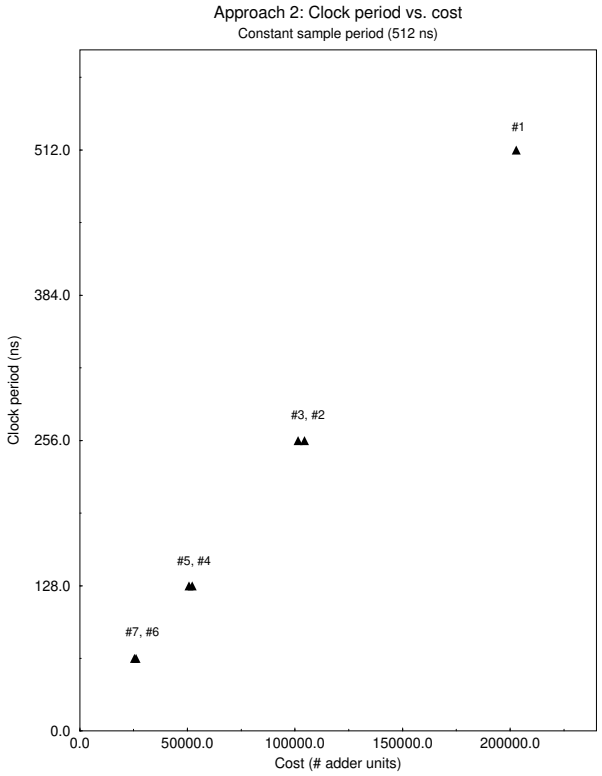

Figure 16: Tradeoff curve obtained by second approach

levels of hierarchy, and very often this implies having to distribute or split the constraints amongst different "modules" of the design. For example, if the designer had a constraint on the total cost of the beamformer the cost constraint would have to be split up appropriately amongst the FIR, PR and SUM blocks. This can pose a considerable problem, since there is no obvious way of determining how to distribute constraints. However, we did not face this problem while designing the beamformer system, since in the special case that we were considering, the constraint was on the throughput or the sample period of the beamformer, and since the FIR, PR and SUM blocks are all connected linearly, the throughput constraint on all of them was the same as that on the complete beamformer system.

5 Conclusions

We have just demonstrated that by varying parallelism and pipelining we can explore the design space of FIR filters and the beamformer system. The design procedure serves little purpose, if it can only be applied to these two systems or systems similar in nature (i.e those that can be formulated as sums of products). As a part of future work, we will investigate the possibility of applying this or a similar design procedure on a broader range

of problems.

Acknowledgements

This work was partially supported by the Semiconducter Research Corporation grant #92-DJ-316, and we gratefully acknowledge their support. We also extend our gratitude to Mike Butler for providing the beamformer example, and for his insightful comments regarding the design procedure outlined in this report.

References

[1] S.Bakshi, and D. Gajski, *A strategy for design space exploration*, UC Irvine, Dept. of ICS, Technical Report 93-10, 1993.

[2] E.Bidet, C. Joanblanq, and P.Senn, *GENRIF: An Integrated FIR Filter Compiler*, Proceedings of EDAC, 1993.

[3] P. Cappello, and C. Wu, *Computer-Aided Design of VLSI FIR Filters*, Proceedings of the IEEE, September 1987.

[4] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[5] A. Oppenheim, A. Willsky and I. Young, *Signals and Systems*, Prentice Hall Inc.,1983

[6] B. Preas and M Lorenzetti, *Physical Design Automation of VLSI Systems,* Benjamin/Cummings, 1988.