ALGORITHM AND ARCHITECTURE DESIGNS FOR HIGH SPEED DIGITAL SIGNAL PROCESSING

by

Keshab K. Parhi

Memorandum No. UCB/ERL M88/61

27 September 1988

COVER PAGE

ALGORITHM AND ARCHITECTURE DESIGNS FOR HIGH SPEED DIGITAL SIGNAL PROCESSING

by

Keshab K. Parhi

Memorandum No. UCB/ERL M88/61

27 September 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALGORITHM AND ARCHITECTURE DESIGNS FOR HIGH SPEED DIGITAL SIGNAL PROCESSING

Ph.D. Keshab K. Parhi Department of EECS

ABSTRACT

This thesis explores systematic approaches to design of high-speed algorithms and architectures for real-time signal and image processing in general, and for one- and two-dimensional recursive and adaptive digital filters in particular. First we address rate-optimal software-programmable multiprocessor implementation of signal processing algorithms described by data-flow programs. We introduce the notion of *perfect data-flow programs*, and prove that fully-static rate-optimal multiprocessor schedules can always be constructed for such programs using no retiming at all. We study properties of *program unfolding transformations*, and derive an expression for the optimum unfolding factor to reduce any data-flow signal processing program to an equivalent perfect data-flow program, which can then be scheduled rate-optimally in a fully-static manner. An upper bound on the number of processors to achieve a rate-optimal schedule is also derived.

Next we develop high-speed algorithms for one- and two-dimensional recursive and adaptive digital filters. *Look-ahead* algorithms are proposed to change the basic linear filter structures (while maintaining identical input-output behavior) and to create additional concurrency. *Scattered look-ahead* and *decomposition* algorithms are used to implement high-speed recursive and adaptive digital filters using fine-grain pipelining, with logarithmic increase in hardware for a linear increase in the sample rate. A technique of *incremental output computation* is proposed and used to derive incremental block digital filters of linear multiplication complexity in block size, as opposed to the square multiplication complexity in all previous block filter structures. Two-dimensional recursive digital filters inherently possess large amount of concurrency. An *index mapping transformation* is used to exploit this concurrency, and to derive fine-grain pipelined and one-dimensional block implementation of two-dimensional recursive digital filters. Look-ahead and incremental computation techniques are extended to two-dimensions, and are used to derive an efficient incremental two-dimensional recursive block digital filter architecture.

Look-ahead and program unfolding transformations are performed on general iterative data-flow signal processing programs to increase concurrency. These transformations are useful where the designed algorithms are unable to meet the real-time requirements of the target application.

[illegible]

9/26/88

(David G. Messerschmitt)

Chairman of Committee

ACKNOWLEDGEMENTS

I am deeply indebted to my research advisor Prof. David G. Messerschmitt for his continuous advice, support and encouragement, and to Prof. David A. Hodges for his timely encouragement throughout the course of my study at Berkeley. I am also grateful to my thesis committee members Prof. Robert W. Brodersen, and Prof. Robert Leachman.

I have benefited from interactions with many individuals; they include (in alphabetical order) Graham Brand, Wen-Lung Chen, Mordechay Ilovich, Rajeev Jain, Edward Lee, Vijay Madisetti, Teresa Meng, Rajiv Ramaswami, and Gilbert Sih of U.C. Berkeley; David Schwartz of Georgia Institute of Technology; Peter Cappello and Cheng-Wen Wu of U.C. Santa Barbara; Bryan Ackland, Mehdi Hatamian, Adriaan Ligtenberg, and Sailesh Rao of AT&T Bell Laboratories, Holmdel; and Hu Chao, Jung Herng Chang, Jean-Paul Jacob, Fred Mintzer, and Gideon Shichman of IBM T.J. Watson Research Center, Yorktown Heights; and to all of them I remain thankful. I am most grateful to the IBM Corporation for providing me an IBM fellowship and a summer job in 1986, and to the AT&T Bell Laboratories for offering me a summer job in 1987. Without the support from Bell Laboratories, the algorithms in this thesis could never have been implemented on IC chips. Finally, I am grateful to my family members, relatives and friends for their love, support and encouragement.

The research carried out in this thesis was supported by a U.C. Regents Fellowship, an IBM Graduate Fellowship, the Defense Advanced Research Project Agency monitored by Naval Electronics Systems Command (under contract number N00039-86-R-0365), and the National Science Foundation (under contract number DCI-85-17339).

TABLE OF CONTENTS

1

INTRODUCTION

1.1. INTRODUCTION

The continuing advancement of scaled VLSI technologies has made it possible to implement very complex functions on single chips at low cost. The computer aids for design of integrated circuits (ICs) have also advanced to a point that a designer can quickly design a chip starting with architecture design specifications. As an example, many designs that required two or three years of design time five years ago can be done within only two to three months today (for an identical size design team). However, in order to fully utilize the silicon area in an efficient manner for any specific application, it is necessary to optimize the algorithms and/or architectures by applying suitable transformations. For example, one can make more dramatic improvements in silicon area of an IC chip implementation by finding a more efficient algorithm or a better architecture of a given algorithm, as opposed to finding a minimum transistor circuit realization for the building blocks (such as adder units or memory cells) of the system.

While the process of chip design from architecture specifications has been well understood and fairly well automated, the algorithm and architecture design from application or problem specifications still remains a difficult task. Over the past decade, researchers have succeeded in systematically mapping a class of regular iterative algorithms on systolic arrays starting with algorithm specifications (see [1-8]). However, finding efficient solutions to implementation of general irregular algorithms still remains an open problem. Several computer-aids for algorithm-specific custom implementations have also been developed in the last five years. Examples of these systems include the Lager design system at Berkeley [9], the Cathedral design system at IMEG, Belgium [10], and the silicon compiler developed at the GE R&D Center [11]. These design systems currently lack the ability to perform transformations on algorithms. These systems are also architecture-specific; they do not explore the entire algorithm and architecture design space.

The goal of this thesis is to develop algorithm and architecture designs for high-speed real-time digital signal and image processing systems. Systematic approaches are explored to transform existing algorithms to create concurrency. These transformation schemes can be applied to a class of algorithms without altering their input-output behavior or functionality. The transformation schemes developed in this thesis can be basically divided into four broad categories. They are program unfolding, retiming, look-ahead and decomposition, and index mapping transformations. *Program unfolding* transformation increases the number of tasks, which can then be distributed more evenly among multiple processors. This transformation does not alter the basic algorithm, but does alter the sequencing or scheduling of the tasks of the algorithm. With an optimum unfolding factor, we can always construct a minimum-time or rate-optimal fully-static multiprocessor schedule. The *retiming transformation* involves moving around the delays in a data-flow program. This transformation can lead to a reduced iteration period, but cannot guarantee a rate-optimal schedule. The third category of transformation involves *look-ahead computation* schemes, which change the structures of linear recursive algorithms (while maintaining identical input-output behavior) to *create* additional concurrency. The *decomposition* algorithms reduce the implementation complexity in these algorithms from linear to logarithmic (with respect to steps of look-ahead). The look-ahead schemes apply to linear recursive systems, and are demonstrated in this thesis in the context of digital filters. The *index mapping transformation* is used to exploit the inherent concurrency in the two-dimensional recursive digital filters.

These algorithm transformations can often result in an efficient high performance implementation (the efficiency measure is based on area-throughput tradeoff, and the throughput is assumed to be reflected by the sample rate of the system, and not necessarily the clock rate). This is because an algorithm transformation on a particular system can lead to a dramatic improvement in the implementation. The transformation techniques developed in this thesis can form the core of an architecture synthesis system, and can serve as the front end to one of the existing architecture-specific vertically-integrated computer-aided design systems.

1.2. CONTRIBUTIONS OF THE THESIS

Chapter 2 of this thesis concentrates on the program unfolding transformation, which leads to the construction of minimum-time fully-static multiprocessor schedules. The basic idea of this transformation is to exploit the repetitive nature of operation in signal and image processing systems (which operate on infinite time series). These systems and the corresponding iterative data-flow programs are non-terminating in nature. The scheduling and task sequencing issues have been studied in great detail over last two decades in the context of assembly line job scheduling and computer science. Minimizing the execution time over single pass of the program has been the goal in these systems. However, in signal and image processing systems, the same tasks are performed repetitively, and therefore we need to minimize the execution time of a single iteration while exploiting maximum possible overlap of successive iterations.

The loop or feedback or recursion in the algorithms described by iterative data-flow programs imposes a lower bound on the iteration or sample period. This bound is fundamental and cannot be broken even if infinite processors are available. The non-recursive systems do not have any feedback, and do not have any lower bound on the iteration period. The actual iteration period of any data-flow program may be much greater than the iteration bound. A *retiming* transformation can improve the iteration period, but cannot guarantee an iteration period equal to the iteration bound.

Unfolding a data-flow program leads to a new program with replicated tasks corresponding to successive iterations. For example, if a program contains 20 tasks and is unfolded by a factor of 5, then the unfolded program will contain 100 tasks belonging to 5 consecutive iterations. Unfolding of iterative data-flow programs can lead to greater concurrency in high performance implementations. Program unfolding increases the number of tasks to be executed, and these unfolded tasks can be more evenly distributed at compile time among multiple processors leading to reduced program execution time. Whether an iteration period equal to the iteration bound can always be achieved by program unfolding with a finite unfolding factor had remained an open question. We show that unfolding the program beyond a certain factor does not lead to any further reduction in the execution time. It is shown that this optimum unfolding factor is given by the least common multiple of the loop delay operators in the data-flow program graph. This unfolding factor leads to an *exactly even distribution* of the iterative tasks, and executes the program in minimum possible time. We derive upper bounds on the number of processors to achieve minimum time schedules. In this context, we introduce the notion of a *perfect data-flow program*, which can always be executed in minimum time without requiring any unfolding or any retiming operation at all. We study properties of unfolded data-flow programs, and show that an unfolding operation with the optimum unfolding factor reduces any iterative data-flow program to an equivalent perfect data-flow program. We also present extensions to multiple rate data-flow programs, and applications to scheduling in non-homogeneous processor system implementations.

Many image processing, video signal processing, radar, sonar, and seismic signal processing applications require very high sample rate implementations. As an example, consider an implementation of a 5×5 convolver image processing system implementation. For a 512×512 frame size, and a frame rate of 30 frames per second, we need a computation rate of 200 million multiply operations per second, which can never be achieved by using a general purpose programmable signal processor implementation. These high performance systems can be implemented with low-cost (i.e. with low silicon area) by using dedicated custom IC chips, which use fine-grain pipelining and parallelism. Many dedicated chips have been implemented using fine-grain pipelining in general and bit-level pipelining in particular in the last two decades using bit-parallel [12-23] and bit-serial [24-33] approaches.

The two basic approaches to achieving concurrency are pipelining and parallelism or block processing. Suppose a single multiply/add operation can be clocked at 25 Mhz in some technology, and we require a sample rate of 100 Mhz. Then, one way to implement this system is to pipeline the multiply/add operation by four stages, i.e. insert four pipeline delays or buffers or latches in intermediate portions of the multiply/add circuit. Another approach is to duplicate the hardware by four times. In this case, we can in each cycle operate on four input samples, and generate four output samples (the four non-overlapping samples form a block). With a clock period of 25 Mhz, and a block size of four, we can achieve an effective system system sample rate of 100 Mhz. Naturally, any combination of pipelining and parallelism can also be exploited. In the above example, yet another alternative would be to pipeline the multiply/add hardware by two stages, and duplicate the hardware by two times. In general, with M stages or levels of pipelining, and with a block size L, we can get an effective improvement in sample rate by a factor of LM. Pipelining is preferred to block processing, since pipelining exploits concurrency with reduced hardware penalty.

Exploiting fine-grain pipelining and block processing techniques in non-recursive systems is straightforward. However exploiting these techniques in recursive systems is a real challenge [23]. This is because the computational latency associated with the internal recursion or feedback in recursive systems limits the opportunities to use fine-grain pipelining and block processing techniques to achieve high sample rate realizations. In chapter 3, we develop techniques to pipeline recursive digital filters in an area-efficient manner. Fine-grain pipelining of recursive loops by simply inserting latches is useful for applications requiring moderate sample rates and where multiple independent computations are available to be interleaved in the pipeline; but not where a single recursive operation needs to be performed at very high sample rates.

We introduce a new look-ahead approach (referred to as *scattered look-ahead*) to pipeline recursive loops. In the look-ahead algorithm, we iterate the recursive state update representation, and implement the new recursion. This approach also improves the iteration bound of the realization. It is shown that the existing clustered look-ahead approach to pipelining recursive filters does not guarantee stability, whereas our new scattered look-ahead approach does guarantee stability. We also propose a new *decomposition technique* to implement the non-recursive portion (generated due to the scattered look-ahead process) in a decomposed manner (for cases where the number of loop pipeline stages can be expressed as a power of 2) to obtain concurrent stable pipelined realizations of logarithmic implementation complexity with respect to the number of loop pipeline stages (as opposed to linear). The upper bound on the roundoff error in these pipelined filters is shown to improve with an increase in the number of loop pipeline stages. We derive efficient pipelined realizations of both direct form and state space form recursive digital filters. Based on the scattered look-ahead technique, we present fully pipelined and fully hardware efficient linear bidirectional and unidirectional ring systolic arrays for recursive digital filters.

In chapter 4, we address block implementation and fine-grain pipelined block implementation of recursive digital filters. In a block implementation, we process samples in non-overlapping blocks. With a block size of L, we can increase the sample rate by a factor of L. We extend an existing linear complexity direct form block filter structure to higher order systems, and refer to it as the *incremental block filter*. Block implementation of state space recursive digital filters has been known for a long time. The two existing popular block structures are the block-state structure, and the parallel block-state structure. However the multiplication complexity of these structures is proportional to the square of the block size. The block state update operation in these filter structures is performed based on the *clustered look-ahead computation*, and requires a linear complexity in block size. But, the output computation of the complete block is done all at once and requires a square complexity in block size. We introduce a new technique of *incremental output computation* that has linear complexity in block size. Based on the clustered look-ahead and incremental output computation approaches, we derive our new *incremental block-state* structure for block implementation of state space filters of multiplication complexity linear in block size. The incremental block-state structure is also extended to the multirate recursive filtering case. We combine the techniques of scattered look-ahead, clustered look-ahead, decomposition, and incremental output computation to introduce several pipeline stages inside the recursive loop of the block filter. We derive deeply pipelined block filter structures for implementation of direct form and state space form recursive digital filters. The multiplication complexity of these deeply pipelined block filters is linear with respect to the block size, logarithmic with respect to the number of loop pipeline stages, and the complexities due to pipelining and block processing are additive. In summary, we can increase the sample rate in recursive digital filters by a factor LM with $O(L)+O(\log_2 M)$ multiplication complexity using our techniques, as opposed to $O(L^2M^2)$ multiplication complexity using previous approaches.

In chapter 5, we address high performance implementation of adaptive and time-varying recursive digital filters. We extend the look-ahead and decomposition algorithms to time-varying systems. Previous approaches to high sample rate adaptive lattice filter implementations have been based on word-level pipelined word-parallel (or "block") realizations. We show that adaptive filters can be implemented in an area-efficient manner by first using fine-grain pipelining, and then using block processing in combination with pipelining if further increase in the sample rate is needed. We show that with the use of the *decomposition technique*, high speed realizations can be achieved using pipelining with a logarithmic increase in hardware (the block realizations require a linear increase). We derive pipelined word-parallel realizations of high sample rate adaptive lattice filters using the techniques of *look-ahead computation*, *decomposed state update implementation*, and *incremental output computation*. These three techniques combined together make it possible to achieve asymptotically optimal complexity realizations (i.e. asymptotically the same complexity as non-recursive systems) of high speed adaptive lattice filters (in both bit-serial and bit-parallel methodologies) and provide a "system solution" to high speed adaptive filtering. The adaptive lattice filter structures are ideal for high sample rate implementations, since the coefficients of a particular stage are adapted order-recursively based on the error innovations of the previous stage, and the coefficient update recursion inside each stage is linear in nature. An example of a normalized stochastic gradient adaptive lattice filter is presented, and its complexity, latency, and implementation methodology tradeoffs are studied.

Chapter 6 focuses on exploiting concurrency in direct-form and local state-space form two-dimensional recursive digital filters to obtain efficient implementations. Unlike one dimensional recursive systems, two-dimensional recursive digital filter algorithms possess large amount of inherent concurrency, which can be exploited for fine-grain pipelining and/or parallelism. The locus of these concurrent computations is referred to as the *concurrent computation region*. We use an *index mapping transformation* to exploit this concurrency, and derive fine-grain pipelined and one-dimensional block filter architectures for the implementation of two-dimensional recursive digital filters. This transformation leads to appropriate interleaving (or indexing) of the input samples, and does not require any algorithm transformation, and does not lead to any hardware overhead.

Another approach to achieving concurrency is by two-dimensional block processing using algorithm transformation techniques. We extend the *look-ahead computation* and the *incremental output computation* principles to the two-dimensional case, and derive a new two-dimensional incremental block filter structure. The multiplication complexity of our new incremental block filter with a block size $L_1 \times L_2$ is $O(Max(L_1^2 L_2, L_1 L_2^2))$, as opposed to $O(L_1^2 L_2^2)$ of the existing block-state filter structures. We then combine pipeline interleaving and incremental block filtering approaches to derive efficient filter structures. The notion of an *index mapping function* is used to derive the *implementable delay and quasi delay operators* for the concurrent pipelined and/or blocked two-dimensional architectures. The quasi delay operators represent delay operator in one dimension and an advance operator in the other. We show that for an N-dimensional recursive filter, the concurrent computation region corresponds to an $(N-1)$-dimensional hyperplane. The pipeline interleaving and the block processing concepts are also extended to higher dimensional cases.

In chapter 7, we create concurrency in general iterative data-flow programs by applying the look-ahead transformations locally to some critical nodes. This chapter uses the theory of program unfolding and look-ahead transformations developed in chapters 2 through 6 of the thesis. The local transformations of iterative data-flow programs are particularly useful in systems where the algorithms cannot meet the real-time constraints of the target application. Chapter 8 concludes the dissertation with suggestions for future work.

1.3. REFERENCES

(1) Kung, H.T., "Why Systolic Architectures", *Computer*, January 1982, pp. 37-45

(2) Cappello, P.R., and Steiglitz, K., "Unifying VLSI Array Design with Linear Transformations of Space Time", *Advances in Computers Research*, Vol. 2, 1984, pp. 23-65

(3) Quinton, P., "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations", *Proc. of 11th Annual Symposium on Computer Architecture*, June 1984, pp. 208-214

(4) Moldovan, D.I., and Fortes, J.A.B., "Partitioning and Mapping of Algorithms into Fixed Size Systolic Arrays", *IEEE Trans. on Computers*, January 1986, pp. 1-12

(5) Li, G., and Wah, B.W., "The Design of Optimal Systolic Arrays", *IEEE Trans. on Computers*, Vol. 34, January 1985, pp. 66-77

(6) Chen, M.C., "A Design Methodology for Synthesizing Parallel Algorithms and Architectures", *Journal of Parallel and Distributed Computing*, December 1986, pp. 461-491

(7) Jagadish, H.V., Rao, S.K., and Kailath, T., "Array Architectures for Iterative Algorithms", *Proc. of the IEEE*, Vol. 75, No. 9, September 1987, pp. 1304-1321

(8) Kung, S.Y., *VLSI Array Processors*, Prentice Hall, 1988

(9) Rabaey, J.M., Pope, S.P., and Brodersen, R.W., "An Integrated Automatic Layout Generation System for DSP Circuits", *IEEE Trans. on CAD of ICs*, Vol. CAD-4, No. 3, July 1985, pp. 285-296

(10) Catthoor, F. *et al*, "Architectural Strategies for an Application Specific Synchronous Multiprocessor Environment", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. 36, No. 2, February 1988, pp. 265-284

(11) Yassa, F. *et al*, "A Silicon Compiler for Digital Signal Processing: Methodology, Implementation, and Applications", *Proc. of the IEEE*, Vol. 75, No. 9, September 1987, pp. 1272-1281

(12) McCanny, J.V., and McWhirther, J.G., "Completely Iterative Pipelined Multiplier Array Suitable for for VLSI", *Proc. of Inst. of Elect. Eng.*, Vol. 129, Part G, No. 2, pp. 40-46, April 1982

(13) Evans, R.A. *et al*, "A CMOS Implementation of a Multibit Convolver Chip", *Proc. of VLSI'83*, edited by F. Anceau and E.J. Aas, Elsevier, Amsterdam, 1983, pp. 227-235

(14) Ulbrich, W. *et al*, "MOS-VLSI Pipelined Digital Filters for Video Applications", *Proc. of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, San Diego, March 1984, pp. 44.7.1-44.7.4

(15) Cappello, P.R., and Steiglitz, K., "A Note on Free Accumulation in VLSI Filter Architectures", *IEEE Trans. on Circuits and Systems*, Vol. 32, No. 3, March 1985, pp. 291-296

(16) Cappello, P.R., and Wu, C.W., "Computer Aided Design of VLSI FIR Filters", *IEEE Proceedings*, Vol. 75, No. 9, September 1987, pp. 1260-1271

(17) Hauck, C.E., Bamji, C.S., and Allen, J., "The Systematic Exploration of Pipelined Array Multiplier Performance", *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Tampa, FL, April 1985, pp. 1461-1464

(18) Hatamian, M., and Cash, G.L., "A 70-MHz 8-bitX8-bit Parallel Pipelined Multiplier in 2.5 Micron CMOS", *IEEE Journal of Solid State Circuits*, Vol. 21, No. 4, August 1986, pp. 505-513

(19) Hatamian, M., and Cash, G.L., "Parallel Bit-Level Pipelined VLSI Designs for High Speed Signal Processing", *IEEE Proceedings*, Vol. 75, No. 9, September 1987, pp. 1192-1202

(20) Noll, T.G. *et al*, "A Pipelined 330 MHz Multiplier", *IEEE Journal of Solid State Circuits*, Vol. 21, No. 3, June 1986, pp. 411-416

(21) Noll, T.G., "Semi-Systolic Maximum Rate Transversal Filters with Programmable Coefficients", *Proc. of the International Conference on Systolic Arrays*, July 2-4, 1986, Oxford, England

(22) Jain, R., Ruetz, P.A., and Brodersen, R.W., "Architecture Strategies for Digital Signal Processing Circuits", *Proc. of the 1986 VLSI Signal Processing Workshop*, in VLSI Signal Processing II, IEEE Press, 1986

(23) Parhi, K.K., and Hatamian, M., "A High Sample Rate Recursive Digital Filter Chip", *in VLSI Signal Processing III*, IEEE Press, November 1988

(24) Jackson, L.B., Kaiser, J.F., and McDonald, H.S., "An Approach to Implementation of Digital Filters", *IEEE Transactions on Audio and Electroacoustics*, Vol. 16, No. 3, 1968, pp. 413-421

(25) Lyon, R.F., "Two's Complement Pipelined Multipliers", *IEEE Transactions on Communications*, 1976, pp. 418-424

(26) Freeny, S.L. "Special Purpose Hardware for Digital Filtering", *Proc. of IEEE*, Vol. 63, April 1975, pp. 633-648

(27) Denyer, P.B., and Renshaw, D., *VLSI Signal Processing: A Bit Serial Approach*, Addison Wesley, Reading, MA, 1986

(28) Denyer, P.B., Murray, A.F., and Renshaw, D., "FIRST: Prospect and Retrospect", *in VLSI Signal Processing I*, ed. by P.R. Cappello, IEEE Press, NY, 1984

(29) Smith, S.G., and Denyer, P.B., *Serial Data Computation*, Kluwer Academic Press, Boston, 1988

(30) Powell, N.R., "Functional Parallelism in VLSI Systems and Computations", *in VLSI Systems and Computations*, ed. by Kung, Sproull, and Steele, Springer-Verlag, 1981

(31) Van Ginderdeuren, J.K.J. *et al*, "Compact NMOS Building Blocks and a Methodology for Dedicated Digital Filter Applications", *IEEE Journal of Solid State Circuits*, Vol. 18, No. 3, June 1983, pp. 306-316

(32) Jain, R. *et al*, "Custom Design of a VLSI PCM-FDM Transmultiplexor from System Specification to Circuit Layout using a Computer Aided Design System", *IEEE Journal of Solid State Circuits*, Vol. 21, No. 1, February 1986, pp. 73-85

(33) Noujaim, S. *et al*, "30 Mhz Compiled Chipset for Graphics Computation", *Digest of 1987 ISSCC*, February 1987

2

PROGRAM UNFOLDING

2.1. INTRODUCTION

The data-flow representation of algorithms clearly exhibits the available concurrency, and forms a natural basis for program specifications in a multiprocessor environment. Although the concept of data-flow computation has existed for quite some time [1-2], only in recent years it has received wide attention [3-8]. In this chapter, we consider non-preemptible deterministic periodic scheduling of iterative static large-grain synchronous data-flow programs. In particular, we consider construction of minimum-time multiprocessor schedules for these programs. The iterative programs are assumed to be non-terminating in nature; in other words we assume the program is executed a very large number of times, such that it can be considered an infinite number of times. This nature of computation is frequently found in many real-time systems, typically in signal and image processing applications. The loops in these programs lead to a lower bound on the execution time of a single iteration, referred to as an *iteration bound* [9-11] (see also [8,12] in the context of asynchronous systems) in the remainder of the chapter. The execution time of a single iteration is referred to as an iteration period, and a schedule is said to be *rate-optimal*, if the iteration period is same as the iteration bound. Traditional multiprocessor scheduling of these iterative flow graphs are based on critical path methods, which minimize the iteration period over one iteration of the program [13-14]. These techniques do not exploit the repetitive nature of the program, and rarely achieve an iteration period equal to the iteration bound. Often the program can be *retimed* to achieve a reduced iteration period, but the retiming [15-17] of a program cannot guarantee a rate-optimal implementation.

Past efforts towards rate-optimal scheduling of iterative flow graphs have been based on construction of *cyclo-static* schedules [18-22]. These schedules exploit the repetitiveness of the data-flow programs. A cyclo-static schedule is characterized by a lattice P×T, where P corresponds to the processor displacement and T is the time displacement (same as the iteration period). A processor displacement P in a schedule implies that if the iteration i of a certain task is scheduled in processor p, then the iteration $(i+1)$ of the same task is scheduled in processor (p+P) modulo P_N, where P_N is the end point of the processor lattice (which can be multidimensional). A time displacement T implies that if the iteration i of a task is scheduled at time t, then the iteration $(i+1)$ of the same task is scheduled in time $(t+T)$. Table 2.1 shows a partial schedule of two iterations of a cyclo-static schedule (from [18]). The symbol A_i denotes the i-th invocation or i-th iteration of task A. In the example of Table 2.1, there are 10 tasks in each iteration, and 4 processors arranged in a 1-D space. The processor displacement in this schedule is 2 units, and the time displacement or the iteration period is also 2 units. As an example, task 3_1 (i.e. iteration 1 of task 3) is scheduled in processor P_1 at time unit 1, and task 3_2 in processor P_3 at time unit 3. Task 3_3 can be scheduled in processor P_1 (which is 3+2 modulo 4) at time unit 5 (not shown in the Table). The P×T lattice for this schedule is 2×2.

Table 2.1: A Cyclo-Static Schedule

Processor	Schedules						
P_4	-	-	-	1_2	4_2	2_2	10_2
P_3	-	-	3_2	7_2	8_2	9_2	-
P_2	-	1_1	4_1	2_1	10_1	-	-
P_1	3_1	7_1	8_1	9_1	-	-	-

A schedule is *fully-static*, if the processor displacement is zero, i.e. P component of the P×T lattice is 0. In other words, all iterations or invocations of a specified node or task are scheduled in the same processor. Table 2.2 shows a partial schedule of two iterations of a fully-static schedule [18], which has a processor-time lattice 0×2. In a fully-static schedule, all tasks corresponding to a single iteration are first scheduled, and this schedule is then replicated for all other iterations with 0 processor displacement.

Table 2.2: A Fully-Static Schedule

Processor	Schedules						
P_4	-	-	-	2_1	10_1	2_2	10_2
P_3	-	-	4_1	9_1	4_2	9_2	-
P_2	-	1_1	8_1	1_2	8_2	-	-
P_1	3_1	7_1	3_2	7_2	-	-	-

Whether fully-static rate-optimal multiprocessor schedules of iterative data-flow programs can always be constructed has so far remained an open question. In this chapter, we explore unfolding of these data-flow programs, and construction of fully-static rate-optimal schedules of the unfolded program. Unfolding of the program leads to an increased number of tasks, which can be more evenly distributed. Although unfolding or blocking of iterative data-flow programs has been considered in [7,18-22], systematic properties of unfolded data-flow graphs have so far not been studied. One question remains to be answered in this context. Is it possible to find an unfolding factor which can guarantee a rate-optimal fully-static schedule? We study properties of unfolded data-flow programs, and prove that an unfolding factor equal to the least common multiple of the delays in the loops of the program always results in an admissible rate-optimal fully-static schedule. It is also shown that the worst-case complexity of constructing fully-static rate-optimal schedules is polynomial (in number of nodes), as opposed to the exponential complexity of constructing cyclo-static rate-optimal schedules [18-22].

The outline of the chapter is as follows. Section 2.2 describes the static data-flow program model, which are described by data-flow graphs. Section 2.3 reviews the notion of iteration bound. Section 2.4 reviews retiming of data-flow programs. In section 2.5, we introduce the notion of a *perfect data-flow program.* These perfect programs always achieve rate-optimal schedules requiring no unfolding and no retiming operations at all. Section 2.6 studies properties of unfolded data-flow programs. The construction of rate-optimal schedules by optimum unfolding of the data-flow program is addressed in section 2.7. Section 2.8 outlines extensions of retiming and program unfolding techniques to multiple-rate data-flow program graphs, and section 2.9 addresses applications of program unfolding techniques to scheduling in non-homogeneous processor implementations.

2.2. ITERATIVE DATA-FLOW PROGRAM MODEL

The iterative data-flow programs are assumed to be synchronous in nature, and are represented by data-flow graphs (DFGs). The nodes in the DFGs represent program or code segments or tasks, and execute the code when invoked. The *directed* arcs correspond to communication between the nodes, and have delays associated with them.

These delays represent the *states* in the DFG, and are dictated by initial conditions during the first iteration of the DFG. An arc with a single delay from node u to node v implies that the instance v_1 depends upon u_0, v_2 depends upon u_1 etc. By transitivity, this implies that v_i depends upon u_{i-1} and all other past instances or iterations of u. Similar argument applies to self loops also, i.e. where u_i depends upon u_{i-1}. Thus, if a task u_i is dependent upon u_{i-1}, u_{i-2} etc., then we model this iterative computation with a self loop and a single delay around the loop (similar to the *reduced dependence graph* model [23]). It may be noted that modeling the program by reduced dependence is appropriate for large-grain parallel compilation. However, it is necessary to consider complete dependence to exploit fine-grain parallelism. The arcs without delays represent precedence relation, i.e., if there is an arc from node u to node v with no delay associated with it, then node v must be scheduled after execution of node u is complete. But the arcs with delays do not imply precedence. This is because if there is an arc from node u to node v with a delay, node v can be executed using the available state information due to the past iteration of u, and independent of the execution of the current iteration of u.

We assume the DFG to be computable, i.e. all loops in the DFG have one or more delays. We assume that the DFG performs repetitive tasks on infinite time series. In other words, we are concerned with non-terminating programs and periodic schedules. We assume the node computation times to be fixed, i.e. we are concerned with deterministic schedules. This is a natural model for most real-time signal and image processing systems. Each repetition of the DFG is referred to as an iteration, and the scheduling period of a single iteration is referred to as the iteration period. We assume that we cannot improve the functionality or granularity of any node in the DFG. By this it is meant that a node cannot be broken into two or more nodes. The DFGs considered in this chapter correspond to large-grain synchronous data-flow graphs. These DFGs can correspond to either homogeneous or multiple sample rate systems. In a homogeneous sample rate system, each node in the DFG is invoked only once during an iteration, and produces a single sample to each of its outgoing arcs and consumes a single sample from each of its incoming arcs when invoked. In multiple sample rate systems, different nodes are invoked a different number of times in a single cycle (see [7] for theory of multiple-rate DFGs). Sections 2.3 through 2.7 are devoted to study of homogeneous DFGs, and the discussion of the multiple-rate DFGs is addressed in section 2.8.

Now we define some terminologies in a DFG.

Definition 2.1: A node in a DFG is an *initial* node, if and only if all of its incoming arcs have delays.

Definition 2.2: A node in a DFG is a *terminal* node, if and only if all of its outgoing arcs have delays associated with them.

Definition 2.3: A node v is a *successor* of node u, if there is a path from u to v with no delay in the path. Then node u is referred to as the *predecessor* of node v.

Definition 2.4: A node v is an *immediate successor* of node u, if there is a directed arc from u to v with no delay. Then, node u is an *immediate predecessor* of node v.

Any node which is simultaneously an initial node and a terminal node is represented as an isolated component in the acyclic precedence graph. In the DFG of Fig. 2.1(a), node B is an initial node, and nodes A and C are terminal nodes. The acyclic precedence graph of this DFG is shown in Fig. 2.1(b).

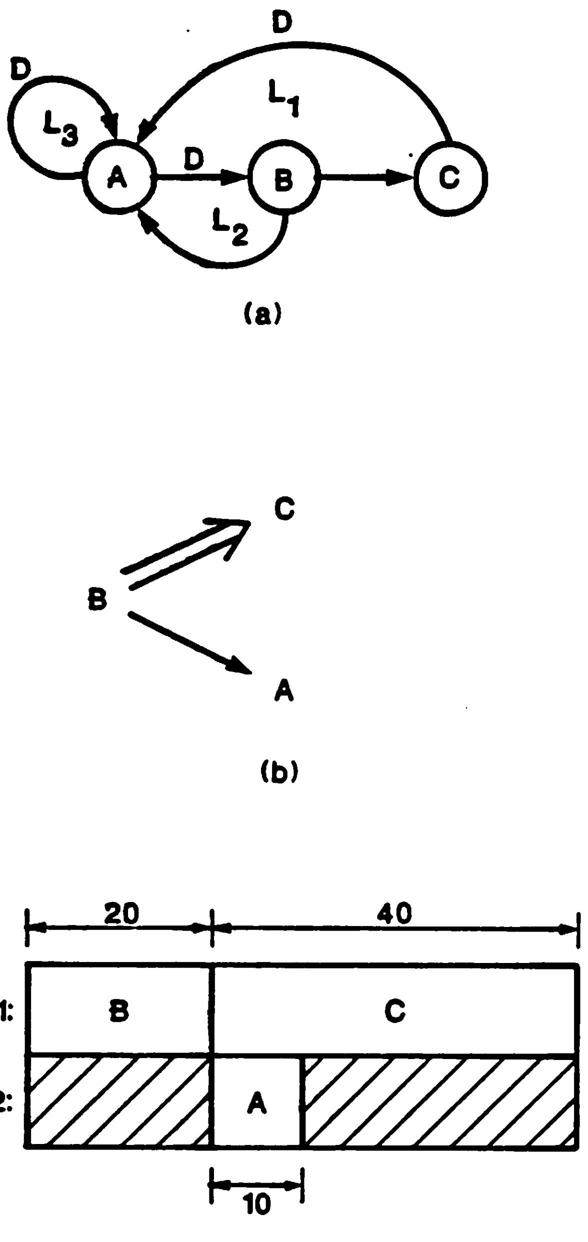


Fig. 2.1: (a) A DFG with three loops. L_3 corresponds to a self loop. The node computation times are 10, 20 and 40 units for A, B, and C respectively. The iteration bound is 35 units, and loop L_1 is the critical loop, (b) Directed acyclic precedence graph, (c): Schedule with iteration period of 60 units.

2.3. ITERATION BOUND

Any DFG involving loops or recursions or feedback has an upper bound on the computation rate or a lower bound on the iteration period [9-11]. This iteration period bound is given by

$$T_{\infty} = Max \left\{ \frac{T_l}{D_l} \right\}, \tag{2.1a}$$

where the maximum is taken over all loops l in the DFG, and T_l is the sum of the computation times associated with all the nodes in loop l, and D_l is the number of delay elements in loop l. The *loop bound* for the l-th loop can be written as

$$T_l \le D_l T_{\infty}. \tag{2.1b}$$

The loop l_0 for which $\frac{T_{l_0}}{D_{l_0}}$ is maximum is referred to as the *critical loop*, and the inequality becomes a strict equality for this loop.

Example 2.1: Consider the DFG in Fig. 2.1(a) with 3 loops. The bounds imposed on the iteration period by the three loops are respectively given by:

$$L_1: \quad t_a + t_b + t_c \le 2T_{\infty} \tag{2.2a}$$

$$L_2: \quad t_a + t_b \le T_{\infty} \tag{2.2b}$$

$$L_3: \quad t_a \le T_{\infty}, \tag{2.2c}$$

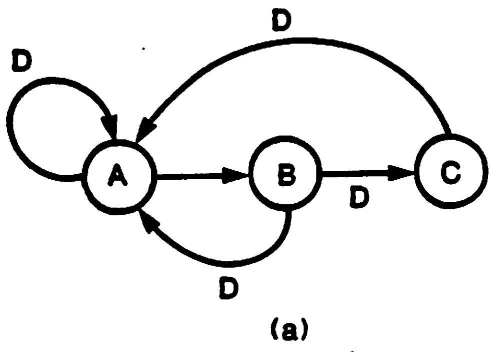
where t_a, t_b, and t_c respectively represent the computation times associated with the nodes A, B, and C. The iteration bound is given by

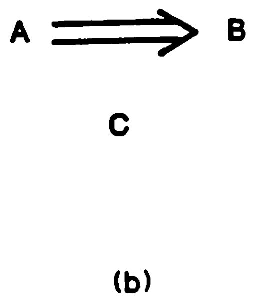
$$T_{\infty} = Max\left(\frac{t_a + t_b + t_c}{2}, t_a + t_b, t_a \right). \quad \square \tag{2.3}$$

Fig. 2.1(b) shows the acyclic precedence graph associated with the DFG. The double arrow represents the critical path in the precedence graph (a convention followed throughout this chapter). Since node B has a delay at its input, it can be invoked first. Nodes C and A can be invoked only after the execution of node B is complete. A two-processor schedule is shown in Fig. 2.1(c) for $t_a = 10$, $t_b = 20$, and $t_c = 40$. The actual iteration period is 60 units, although the iteration period bound is only 35 units for this example. The loop L_1 here is the critical loop.

2.4. RETIMING

Retiming was proposed by Leiserson, Rose, and Saxe to improve the clock rate in synchronous circuits (see [15]), and is applied here to improve the iteration period of multiprocessor schedules in DFGs. The process of retiming involves moving around the delays in the DFG such that the total number of delays in any loop remains unaltered, and the steady state input-output behavior of the system is preserved (see [15]). Removal of a fixed number of delays from each of the incoming arcs of any node, and addition of the same fixed number of delays to each of the outgoing arcs of the same node is an example of a valid retiming operation applied locally to a node. Note that this also corresponds to a cutset transformation around the node [16]. Thus, any local retiming operation can be performed at a node, only if all of its incoming arcs have delays associated with them. Any valid global retiming operation can always be described as a linear combination of such local retiming operations. Since the retiming operation preserves the number of delays in a loop [15] and the loop computation times, it also preserves the iteration bound of the DFG. The retiming operation can change the total number of delays in the DFG. This can be verified by locally retiming a node in a DFG, where





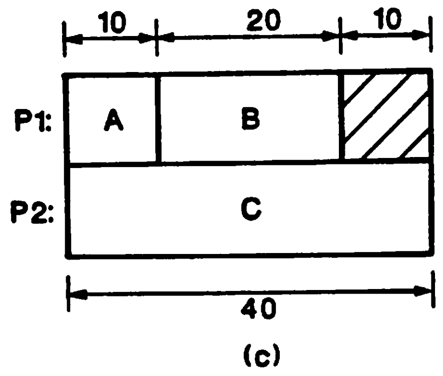


Fig. 2.2: (a) A retimed version of the DFG of Fig. 2.1(a), (b) Precedence graph, (c) Schedule with iteration period of 40 units.

the number of outgoing arcs is different from the number of incoming arcs.

Fig. 2.2(a) shows a retimed version of the DFG of Fig. 2.1(a), obtained by performing retiming operation locally at node B. Note that the number of delays in each loop is unaltered (but, the total number of delays in the DFG has changed). The retiming process creates new initial conditions, and therefore, new precedence relations, new initial and terminal nodes and new schedules. The precedence graph and the schedule corresponding to the retimed DFG in Fig. 2(a) are respectively shown in Fig. 2.2(b) and Fig. 2.2(c). The iteration period of the retimed DFG is 40 units, which is 5 units greater than the iteration period bound, but 20 units less than the iteration period corresponding to the schedule in Fig. 2.1(c).

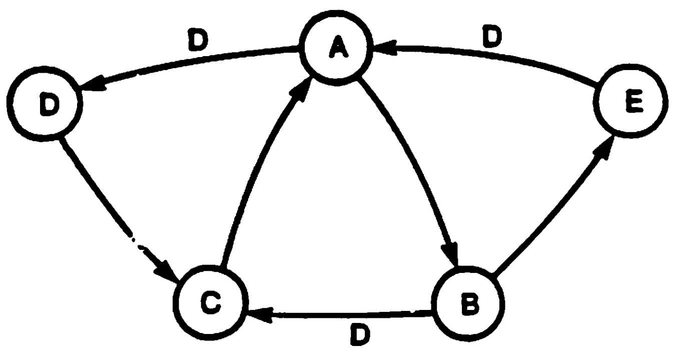
The retiming process *attempts* to evenly distribute the delays, i.e. it tries to equalize the computation times associated with all the nodes between any two delays in the critical loop. But it cannot guarantee an exactly even distribution of the delays, since an exactly even distribution would require splitting of nodes, which is not permitted. Because of this uneven distribution of delays, the actual iteration period is greater than the iteration bound.

2.5. PERFECT DATA-FLOW PROGRAMS

In this section, we introduce the notion of perfect data-flow programs described by perfect graphs. We will make considerable use of these graphs in later sections.

Definition 2.5: Any DFG which has one and only one delay in each loop is defined as a *perfect graph.*

The DFG shown in Fig. 2.3(a) is an example of a perfect graph. This DFG has one initial



(a)

$$\begin{tikzcd} D \arrow[r, Rightarrow] & C \arrow[r, Rightarrow] & A \arrow[r, Rightarrow] & B \arrow[r, Rightarrow] & E \end{tikzcd}$$

(b)

P1:	D_1	C_1	A_1	D_2	C_2	A_2		
P2:				B_1	E_1		B_2	E_2

(c)

Fig. 2.3: (a) A Perfect Graph, (b) Precedence graph, (c) Partial schedule of two iterations. The iteration period of 3 units is obtained by overlapping two successive iterations. This schedule is rate-optimal.

node (node D), one terminal node (node E), and three loops, and all the loops are critical (assuming unit execution time for each node or task). The iteration bound for this DFG is 3 units of time (u.t.). The precedence graph for the DFG is shown in Fig. 2.3(b), and the length of the critical path is 5 u.t. Hence, any CPM schedule would require an iteration period of 5 u.t. However, we can exploit the periodicity or cyclic nature of the schedules, and overlap consecutive iterations to obtain a rate-optimal fully-static schedule as shown in Fig. 2.3(c). Note that the DFG did not need to be *retimed* to obtain a rate-optimal schedule. In fact, the perfect graphs have the property that they directly lead to rate-optimal fully-static schedules (and therefore completely eliminate the need for retiming or program unfolding), and it is this property that makes the notion of perfect graphs useful and important. The schedule in Fig. 2.3(c) has an iteration period of 3 (which is rate-optimal), and an input-to-output delay of 5 (the input-to-output delay is defined to be the maximum delay or latency from any initial node to any terminal node).

Fig. 2.4 shows several retimed versions of the DFG in Fig. 2.3(a), and the corresponding rate-optimal fully-static schedules. Even though all the schedules in Fig. 2.3 and 2.4 are rate-optimal, the schedule in Fig. 2.4(c) is only delay-optimal, which has an input-to-output delay of 3 u.t. Thus, retiming perfect graphs does not improve the iteration period, but may improve the input-to-output delay.

One might have already observed that not all the initial nodes necessarily start at the same time. For example, the executions of the starting nodes D and E in the perfect graph of Fig. 2.4(a) are skewed by one unit of time. The skewing of initial nodes permits overlap of consecutive iterations, and is often essential for construction of rate-optimal schedules.

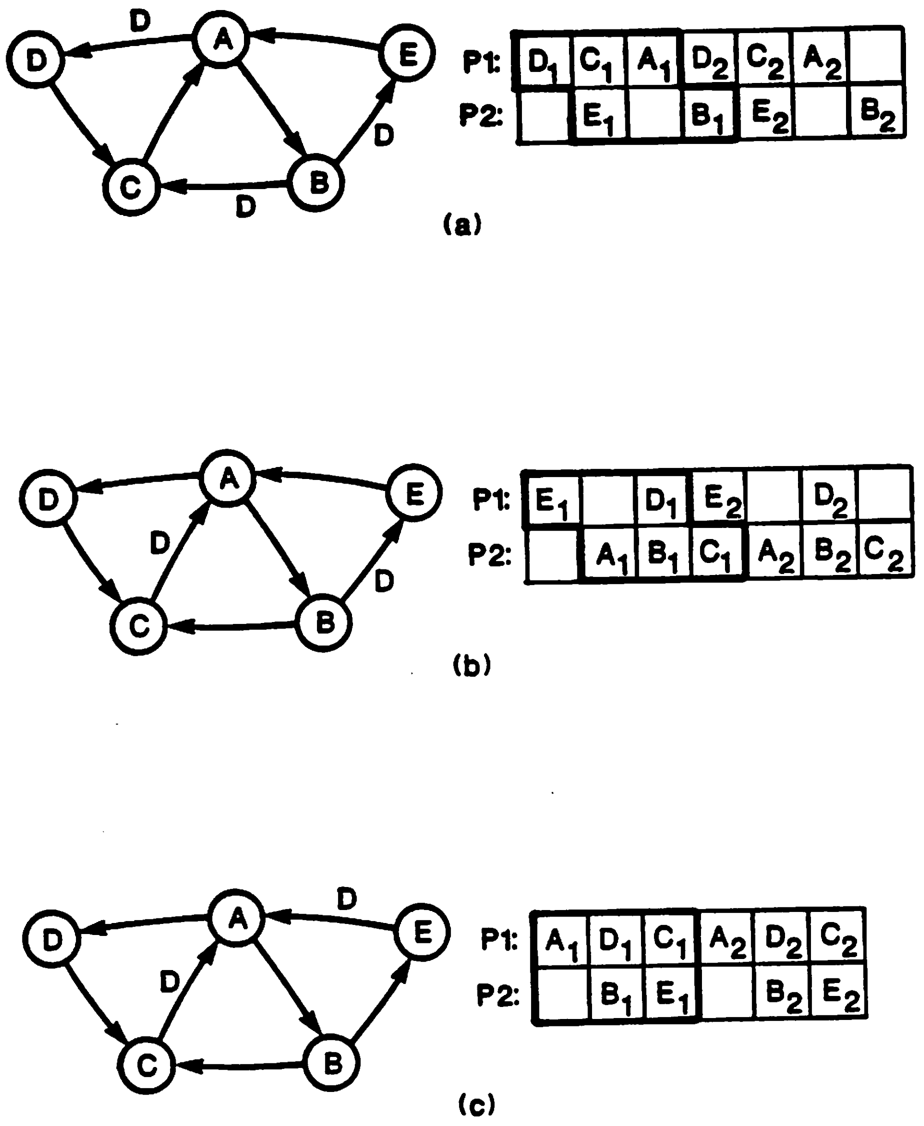


Fig. 2.4: Several retimed versions of the perfect graph of Fig. 2.3(a), and corresponding rate-optimal schedules.

Definition 2.6: An arc from node u to node v is said to be *transitive*, if there is a path from node u to v, and the number of delays associated with the arc $u \rightarrow v$ and the path from u to v are identical (similar to the definition in [24] for a directed acyclic graph). A single path can have more than one associated transitive arcs. The number of delays in the transitive arc $u \rightarrow v$ can be either 1 or 0 in a perfect DFG.

Example 2.2: See Fig. 2.5 for examples of transitive arcs. In Fig. 2.5(a) and Fig. 2.5(b), the path $A \rightarrow B \rightarrow C$ and the arc $A \rightarrow C$ contain equal number of delays (0 in Fig. 2.5(a) and 1 in Fig. 2.5(b)). Hence, the arc $A \rightarrow C$ is transitive. In Fig. 2.5(a), the path $A \rightarrow B \rightarrow C$ implies that there is a precedence constraint between invocations of task A and C. The transitive arc $A \rightarrow C$ also dictates the same constraint, and is therefore redundant. □

Definition 2.7: A loop is said to be a *maximal loop* if it does not contain any transitive arcs. A loop which is not maximal is referred to as a non-maximal loop.

Example 2.3: The perfect graph in Fig. 2.5(c) has a single maximal loop. This is because the graph contains two transitive arcs, and after deletion of these two transitive arcs the DFG contains a single maximal loop, which is $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. This maximal loop has 3 non-maximal loops associated with it. The loop bounds of the associated non-maximal loops can be derived from that of the corresponding maximal loop by deleting the computation times of the appropriate nodes. The loop bound for the maximal loop in Fig. 2.5(c) is given by

$$t_a + t_b + t_c + t_d < T_{\infty}. \tag{2.4a}$$

The loop bounds of the three associated non-maximal loops are given by

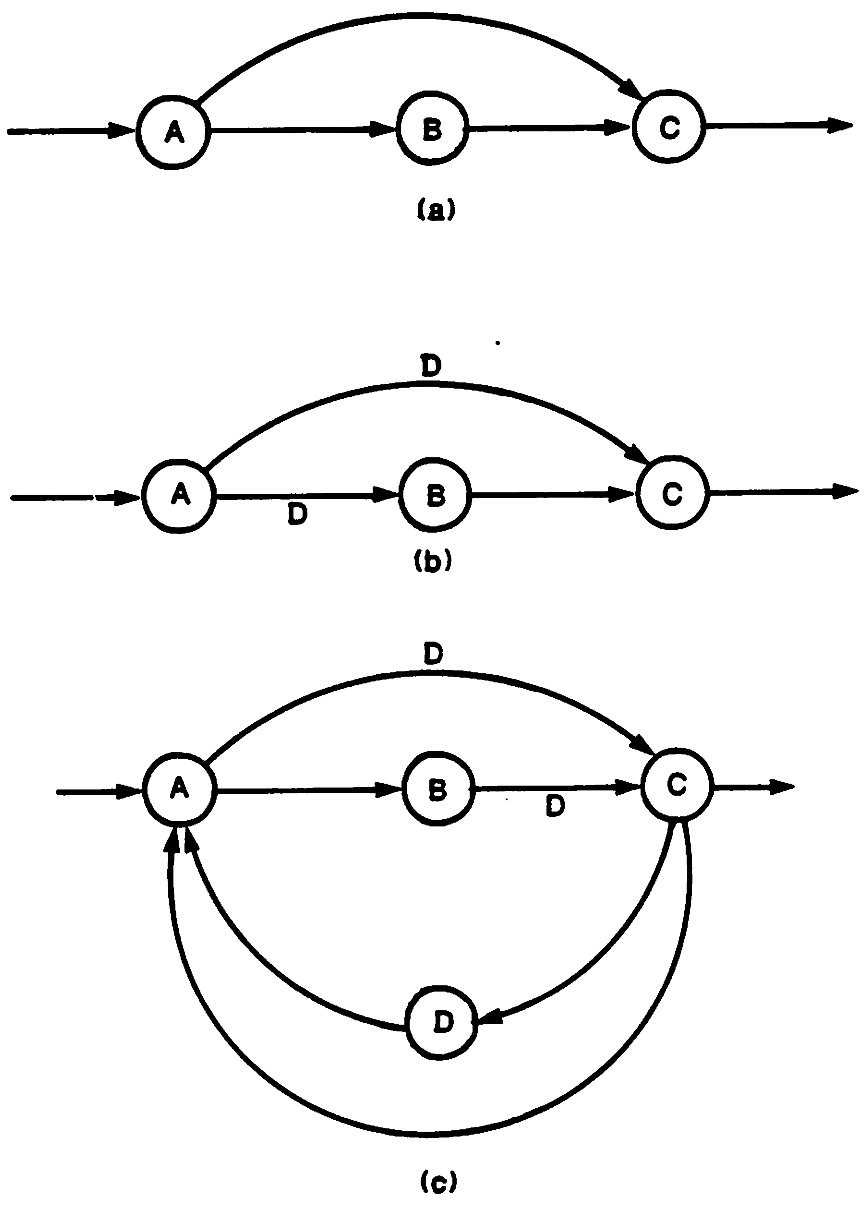


Fig. 2.5: Illustration of transitive arcs.

$$t_a + t_b + t_c < T_\infty, t_a + t_c < T_\infty, t_a + t_b + t_c < T_\infty. \qquad (2.4b)$$

The above non-maximal loop bounds are obtained by deleting the computation times of the appropriate nodes. Note that the loop bounds for the associated non-maximal loops are automatically satisfied if the loop bounds for the corresponding maximal loops are satisfied. □

Lemma 2.1: The number of loop delays in a maximal loop and its associated non-maximal loops are same. This is true since the non-maximal loop consists of transitive arcs and from definition 2.6, the number of delays in a transitive arc and its associated path are same.

Lemma 2.2: A non-maximal loop can never be a critical loop.

Proof: This follows from lemma 2.1 and definition 2.6. From lemma 2.1, the maximal loop and the associated non-maximal loops have the same number of delays, and from definition 2.6, the maximal loop contains all the nodes belonging to the associated non-maximal loop as well as additional nodes. The total computation time of the maximal loop is greater (for identical number of loop delay operators) than the non-maximal loop, and so the maximal loop has a more critical loop bound. □

Lemma 2.3: A schedule for the graph obtained after deleting all the transitive arcs in the original DFG is an admissible schedule for the original DFG.

Proof: Deletion of a transitive arc does not alter the precedence constraints. Thus, deletion of all transitive arcs from the DFG does not alter its precedence constraints. Hence an admissible schedule for the graph obtained from the DFG after deletion of all the transitive arcs is also an admissible schedule for the original DFG. □

Definition 2.8: A schedule of a list of Q nodes $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_Q$ is said to be contiguous if the nodes are scheduled in that order without any intermediate gap or idle time. Note that any node can be scheduled in any processor in a multiprocessor implementation.

Now consider the following algorithm for scheduling of the recursive nodes of the perfect DFG (that is, the scheduling of the nodes not belonging to any loop is not considered).

Algorithm 2.1: First, we remove all the transitive arcs from the perfect graph, since the precedence relations due to these are automatically satisfied (see lemma 2.3). All the remaining loops of the DFG are maximal. The maximal loops are then ordered, and scheduled according to the decreasing order of their loop computation times. The nodes in each maximal loop are also ordered to form a list with the node containing the loop delay at its input arc as the leading node of the list, and the other nodes are placed so as to satisfy the precedence constraints. A separate processor is assigned for scheduling of each maximal loop. First, the nodes of the critical loop are scheduled contiguously in processor 1. Then, the nodes of the next critical loop are scheduled in processor 2 such that the schedules completed so far are preserved. In other words, if some of the nodes of this loop also belong to the critical loop (and therefore have already been scheduled in processor 1), then their schedule should remain unaltered. This process is repeated until scheduling of all the maximal loops is complete.

Remark: Note that we do not assume the scheduling of all the maximal loops to begin at the same time unit. In other words, the scheduling of the maximal loops in different processors can be skewed in time. This skewed scheduling separates consecutive iterations of the DFG by a non-vertical boundary. Also note that it is possible to merge the tasks of two or more processors to reduce the number of processors in a post-processing step, but this is not considered here as a part of the algorithm. The complexity of the above fully-static scheduling algorithm is polynomial in the number of nodes, whereas the complexity of the cyclo-static scheduling algorithm proposed in [18-22] is exponential.

Some properties of scheduling algorithm 2.1 are summarized in the following lemmas.

Lemma 2.4: Nodes in any maximal loop of a perfect DFG are scheduled non-contiguously (i.e. with intermediate gaps) if and only if a path consisting of nodes of this loop has an associated parallel path with a longer path computation time.

Proof: In algorithm 2.1, nodes of maximal loops are ordered to form a list with the node containing the loop delay as its leading node. A path corresponding to the maximal loop refers to a set of connected nodes, which are members of this list. To prove the "if" portion, consider the path $N_1 \rightarrow N_2 \rightarrow N_4$ corresponding to some maximal loop, and its associated parallel path $N_1 \rightarrow N_3 \rightarrow N_4$, and assume the computation time of N_2 to be shorter than that of N_3. This precedence results in a gap in scheduling of the nodes of this loop, since N_4 can be scheduled only after the execution of N_3 is complete. To prove the "only if" portion, assume that there exists some gap between completion of N_2 and invocation of N_3 in the scheduling of some path $.... \rightarrow N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow ....$ (call this path P_1) corresponding to a maximal loop. This would occur if the invocation of N_3 is constrained by completion of another node (say N_5). Denote the path $..... \rightarrow N_5 \rightarrow N_3 \rightarrow .....$ as P_2. If N_3 is the only common node between P_1 and P_2 (i.e. P_1 and P_2 have no node in common to the left of N_3), then the schedule of the nodes in the list ....$\rightarrow N_1 \rightarrow N_2$ could be right-shifted so that completion of N_2 and N_5 coincide. However, the existence of the gap in the schedule implies that the the nodes to the left of N_3 in paths P_1 and P_2 are dependent, and have at least one node in common. This implies the existence of an associated parallel path. This associated parallel path has a longer computation time. □

Remark: Two parallel paths in a perfect DFG must have the same number of delays (which can be either 1 or 0). If this were not the case, the two maximal loops containing the two parallel paths would have different number of loop delays, and the DFG would be imperfect.

Lemma 2.5: A contiguous scheduling of the nodes of the critical loop of the perfect DFGs is admissible.

Proof: From lemma 2.4, two parallel paths with different path computation times lead tc a non-contiguous schedule, and the nodes of the path with less path computation time are scheduled with an intermediate gap. Since the paths of the critical loop have the largest path computation time, they can be scheduled without any intermediate gap. Any loop with gaps in the schedule must be non-critical. □

Now we define two different types of processor idle time in the scheduling of the recursive nodes of a perfect DFG (recall recursive nodes are nodes which belong to at least one loop in the DFG).

Definition 2.9: The idle time of a processor is referred to as a *gap delay* (or gap time), if this idle time is a result of a non-contiguous schedule (i.e. there exist two parallel paths with different path computation times). Any idle time, which is not a gap delay, is referred to as a *slack delay* or slack time (also referred to as skew delay or shimming delay).

Theorem 2.1: For any perfect graph, we can construct fully-static rate-optimal schedules without requiring any retiming transformation.

Proof: The nodes of the critical loop can be scheduled contiguously requiring a period equal to the critical loop computation time or the iteration bound. This schedule can be replicated over successive iterations with no gap at all in the same processor with a time displacement equal to the iteration bound. For each gap in the scheduling of nodes (of non-critical loops), there exists a path with longer computation time. This implies that the sum of the computation time and the gap time of any loop cannot exceed the critical loop computation time (or the iteration bound), and therefore the algorithm 2.1 results in a rate-optimal schedule. The schedule of the single iteration can be replicated with zero processor displacement and with a time displacement equal to the iteration period bound, and hence the schedule is fully-static. Note that this results in a non-negative loop slack time equal to the difference of the iteration bound and sum of the loop computation time and the loop gap time. □

Theorem 2.2: The number of maximal loops in a perfect graph represents an upper bound on the number of processors to schedule the recursive nodes in a fully-static and rate-optimal manner.

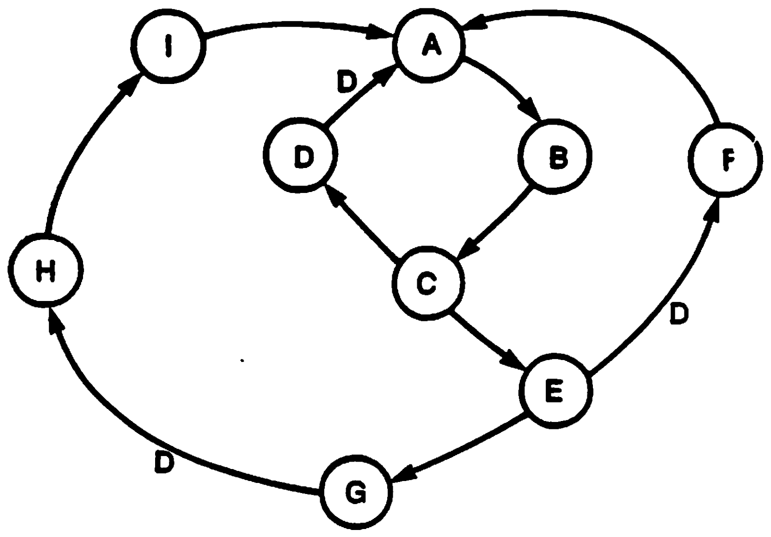


Fig. 2.6: A perfect graph with three loops.

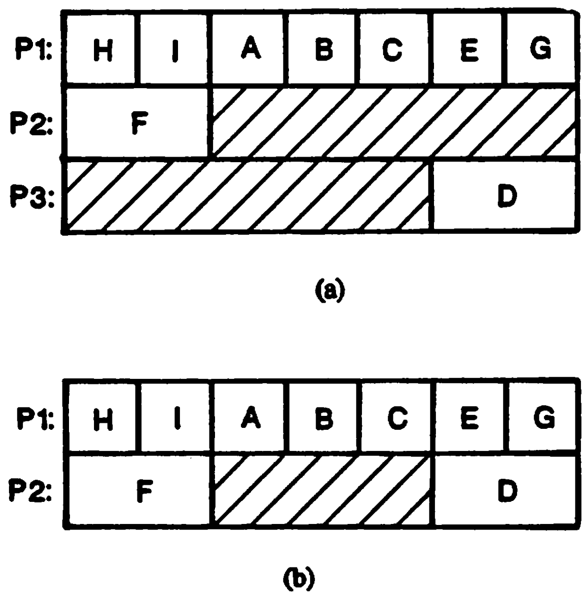


Fig. 2.7: (a) Rate-optimal scheduling with three processors, (b) Rate-optimal schedule with two processors.

Proof: The scheduling algorithm 2.1 assigns a separate processor to each maximal loop. Hence, the upper bounds on the number of processors is equal to the number of maximal loops of the perfect DFG. □

Example 2.4: Consider the DFG of Fig. 2.6, and the corresponding schedules in Fig. 2.7. The execution times of nodes D and F are 2 units each, and that of other nodes is 1 unit. The perfect graph has 2 initial nodes (nodes H and F), and two terminal nodes (nodes D and G). The loops are first ordered as $HIABCEGH$, $FABCEF$, and $ABCDA$. The critical loop $HIABCEGH$ is first scheduled in processor P1 (see Fig. 2.7(a)). Then, the nodes of the next critical loop are scheduled in processor P2. Finally, the nodes of the last loop are scheduled in processor P3. We observe that we can merge the tasks in processors P2 and P3 to a single processor as shown in Fig. 2.7(b). This permits us to obtain a rate-optimal fully-schedule using 2 processors. □

2.6. UNFOLDED DATA-FLOW PROGRAM GRAPHS

In this section, we study properties of unfolded or blocked data-flow program graphs. An unfolded DFG with an unfolding factor J contains J invocations of each node. The number of nodes and arcs in the unfolded DFG are respectively JN and JE, where N and E respectively represent the number of nodes and arcs in the original DFG. An execution cycle (or simply a cycle) of the unfolded DFG constitutes execution of JN nodes, and corresponds to merged execution of J successive iterations of the original DFG.

In the unfolded DFG, a node X_i computes the results of iteration x_{i+kJ} at the k-th cycle. As an example, let X be a node in the original DFG, and let X_1, X_2, and X_3 represent the corresponding three nodes in the unfolded DFG for $J = 3$.

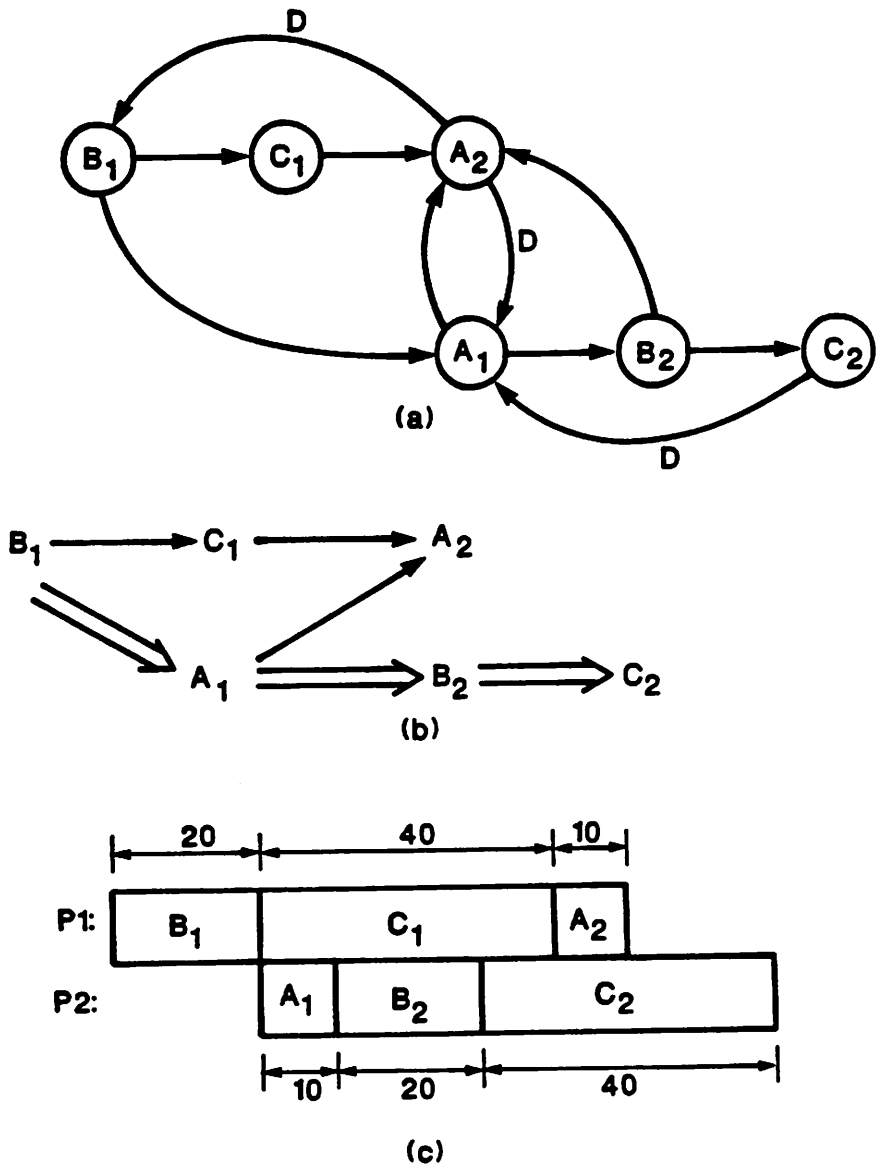


Fig. 2.8: (a) An equivalent unfolded DFG of the DFG in Fig. 2.1(a) for unfolding factor of two. This unfolded DFG is a perfect graph, (b) Precedence graph, (c) Rate-optimal schedule.

Then, the node X_1 performs the iterations x_1, x_4, x_7 in cycles 1, 2, and 3 respectively, node X_2 performs the iterations x_2, x_5, x_8 in cycles 1, 2, and 3 respectively, and similarly for X_3. We assume the convention that x_0 is the initial condition. This implies that the output of any arc with a unit delay from the node X_3 will correspond to the initial condition x_0. To be more precise, an arc with a delay from the node X_1 will not correspond to x_0, but to x_{-2}. This block delay notion is important in the framework of this chapter. To conclude, a unit delay in the unfolded DFG is a *J–slow* delay.

The unfolded DFG can be constructed very easily from the original DFG (see [18] for a systematic procedure). We illustrate this procedure here using the DFG of Fig. 2.1(a) as an example. Since each cycle in the unfolded DFG is periodic, and corresponds to J iterations of the original DFG, we need to consider only first J iterations. For $J = 2$, we consider only the first two iteraticns of each node. The precedence constraints for the two iterations of the DFG of Fig. 2.1(a) are summarized as below:

$$A_1 = f_A(A_0, B_1, C_0)\,, B_1 = f_B(A_0)\,, C_1 = f_C(B_1)\,, \tag{2.5a}$$

$$A_2 = f_A(A_1, B_2, C_1)\,, B_2 = f_B(A_1)\,, C_2 = f_C(B_2)\,, \tag{2.5b}$$

where $f_i(.)$ represents the functionality associated with the node i. The dependence relations in (2.5a) correspond to the first iteration, and those in (2.5b) for the next iteration. Note that the relations in (2.5b) are obtained from (2.5a) by shifting the indices appropriately. The unfolded DFG is constructed by incorporating the precedence relations in (2.5) and by inserting arcs with a unit delay from node A_2 to A_1 and B_1, and from C_2 to A_1 (to realize the initial conditions A_0 and C_0 respectively). The unfolded DFG is shown in Fig. 2.8(a). The only initial node of the DFG is B_1, and the terminal nodes are A_2 and C_2. The precedence relations for this DFG are shown in Fig. 2.8(b). This unfolded DFG has several nice properties. One can verify that the DFG is indeed a perfect graph (this is not a coincidence; systematic construction of perfect graphs from any DFG by unfolding is studied in section 2.7). Therefore, the unfolded DFG can be scheduled rate-optimally in a fully-static manner as shown in Fig. 2.8(c). Now we study some properties of the unfolded DFGs.

Lemma 2.6: The iteration bound associated with an unfolded DFG with an unfolding factor J is JT_∞, where T_∞ is the iteration bound of the original DFG. The unfolded DFG schedules J iterations of the original DFG in JT_∞ units of time, and the iteration bound per iteration is not altered by unfolding.

Property 2.1: The number of delays in the unfolded DFG is exactly the same as that in the original DFG.

Proof: The delays represent initial *states* at the beginning of the execution. These delays activate the invocations, and are updated each cycle. Let D_T denote the total number of delays in the DFG. Then each iteration of the DFG updates D_T states to be used during the next iteration. We know that the termination of each iteration updates D_T values and the termination of the J-th iteration of the execution cycle of the unfolded DFG must also update D_T values to be used for the next execution cycle of the unfolded DFG. Thus unfolding conserves the number of delays in a DFG. As an example, the DFG in Fig. 2.1(a) and the unfolded DFG in Fig. 2.8(a) both contain 3 delays. □

Property 2.2: Let T_i and D_i respectively correspond to the sum of all computation times and the delay count associated with the i-th loop in the unfolded DFG. Then

$$T_i \leq JT_\infty D_i \tag{2.6}$$

must hold.

Proof: Let T'_∞ be the iteration bound of the unfolded DFG. Then, $T_i \leq T'_\infty D_i$ must hold. But, $T'_\infty = JT_\infty$ (due to lemma 2.6), and hence (2.6) must hold. □

Lemma 2.7: Any linear additive combination of the non-critical maximal loop bounds of the original DFG can never correspond to a critical loop bound of the unfolded DFG.

Proof: Consider three non-critical maximal loops L_1, L_2, and L_3. Then, $T_1 < T_\infty D_1$ and $T_2 < T_\infty D_2$, and $T_3 < T_\infty D_3$. Thus the linear additive combination $T_1 + T_2 + T_3 < T_\infty(D_1 + D_2 + D_3)$ can never be a critical loop, since this contains a strict inequality, and a critical loop must contain a strict equality. The argument generalizes. □

Corollary: A critical loop bound in the unfolded DFG corresponds to a linear additive combination of critical loop bounds of the original DFG. However, any linear additive combination of loop bounds of the original DFG may not correspond to a critical loop bound of the unfolded DFG.

Property 2.3: Any loop bound relation of the type (2.6) in the unfolded DFG can be obtained either by multiplying a loop bound relation in the original DFG by a constant, or by taking linear additive combinations of the loop bounds of the original DFG such that the right side is a multiple of J.

Proof: The right side of the loop bound for any loop in the unfolded DFG must be a multiple of J (when expressed in terms of T_∞, the iteration bound of the original

DFG) due to property 2.2. Assume that the i-th loop of the original DFG has a bound $T_i \le D_i T_\infty$. Any linear additive combination of one or more loop bounds in the original DFG, which corresponds to a loop bound in the unfolded DFG, must be of the form

$$\sum_{i=1}^{N} \alpha_i T_i \le (\sum_{i=1}^{N} \alpha_i D_i) T_\infty, \tag{2.7}$$

where N is the number of loops of the original DFG, and $\sum_{i=1}^{N} \alpha_i D_i$ is divisible by J. Any loop bound in the unfolded DFG, which is not of the form (2.7), will imply an entirely new loop bound in the original DFG. But this is not possible, since unfolding does not create new loop bounds. □

Note that any linear additive combination of the loop bounds of the original DFG of the form (2.7) may not correspond to a loop bound in the unfolded DFG. Now we discuss four important special cases of (2.7) in the context of a single loop bound. Let the loop bound of some loop in the original DFG be $T \le DT_\infty$, where T is the loop computation time, D the loop delay count, and T_∞ the iteration bound of the original DFG. The iteration bound of the unfolded DFG is $T'_\infty = JT_\infty$.

Case I: J divisible by D: Let us assume $J = QD$ and Q is an integer. Then a loop bound of the unfolded DFG will be of the form $QT \le (JT_\infty)$ or $QT \le T'_\infty$. This implies that one loop of the unfolded DFG will contain a single delay and Q instances of the nodes of the loop of the original DFG. Since the unfolded DFG contains $J = QD$ instances of each node and D delays (due to delay conservation, property 2.1), it must contain D distinct loops with a single delay in each loop.

Case II: D divisible by J: Assume $PJ = D$. The loop bound of the unfolded DFG is of the form $T \leq PT'_{\infty}$. The unfolded DFG contains J distinct loops, and P delays in each of these loops.

Case III: D and J coprime: For this case, a loop bound in the unfolded DFG is of the form $JT \leq D\,(JT_{\infty})$ or $JT \leq DT'_{\infty}$. The unfolded DFG contains one distinct loop with D loop delays.

Case IV: General Case: Assume $PJ = QD$, where P and Q are coprime. The loop bound of the unfolded DFG is of the form $QT \leq PT'_{\infty}$. The unfolded DFG contains $\frac{J}{Q} = \frac{D}{P}$ distinct loops with P delays in each of these loops.

Example 2.5: Consider the DFG of Fig. 2.1(a) and its unfolded DFG of Fig. 2.8(a). The original loop bounds are

$$t_a \leq T_{\infty},\ t_a + t_b \leq T_{\infty},\ t_a + t_b + t_c \leq 2T_{\infty}. \tag{2.8a}$$

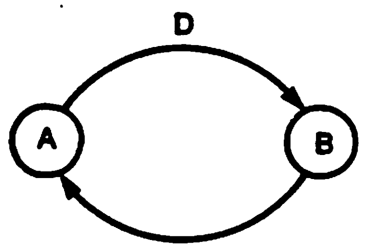
The loop bounds of the unfolded DFG (with unfolding factor two) are

$$2t_a \leq T'_{\infty},\ 2t_a + 2t_b \leq T'_{\infty},\ t_a + t_b + t_c \leq T'_{\infty},\ 2t_a + t_b \leq T'_{\infty},$$

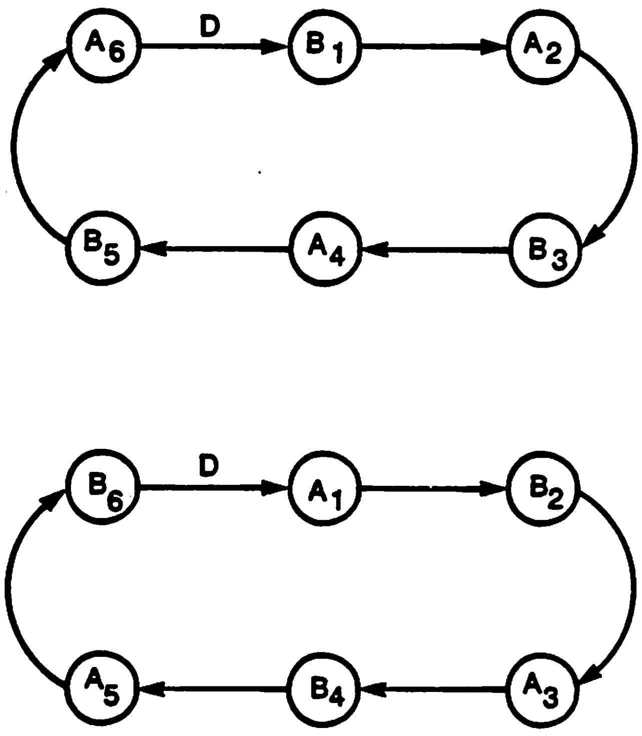
which are linear additive combinations of the original DFG loop bounds. □

Property 2.4: Any loop in the original DFG with D loop delays leads to D distinct loops in the unfolded DFG for an unfolding factor of KD. Each of these distinct loops contains a unit loop delay, and K instances of each node belonging to the loop in the original DFG.

Proof: This follows from special Case I of property 2.3. □



(a)



(b)

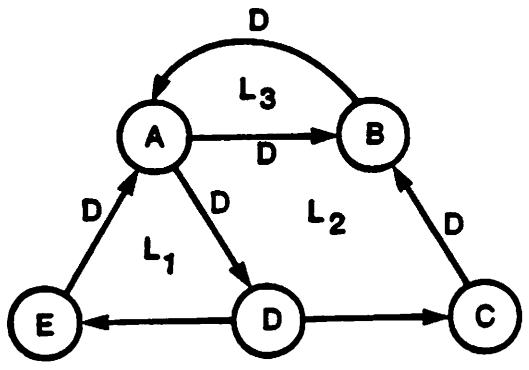
Fig. 2.9: (a) A DFG, (b) Equivalent unfolded DFG for unfolding factor 6.

Example 2.6: Consider the simple example of a two node DFG in Fig. 2.9(a). Fig. 2.9(b) shows an equivalent unfolded DFG for an unfolding factor 6. The original DFG has 2 delays in the loop, and the unfolded DFG has 2 distinct loops with a single delay in each loop. Each loop in the unfolded DFG contains 3 instances of the nodes of the original loop for an unfolding factor of 6. □

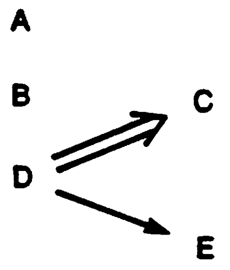
2.7. FULLY-STATIC RATE-OPTIMAL SCHEDULING

This section uses the results of the previous sections to prove that the tasks of any DFG can be scheduled rate-optimally in a fully-static manner.

One might conjecture that we can always achieve a rate-optimal schedule by using an unfolding factor equal to the delay count in the critical loop and then by retiming the unfolded DFG. This is because, the critical loop in the equivalent unfolded DFG would contain a single delay, and the tasks in the critical loop of the unfolded DFG can then be evenly distributed. However, this conjecture is not true! Although the single delay in the critical loop permits an even distribution of the tasks in that loop of the unfolded DFG, another non-critical loop might suffer from an uneven distribution of tasks, and may lead to an iteration period greater than the iteration bound. This is illustrated using the DFG example in Fig. 2.10, where the loop delay counts in the DFG are 2 and 3 respectively. The execution times of nodes A, B, C, D, and E in Fig. 2.10(a) are respectively 20, 5, 10, 10, and 2, and the iteration bound is 16, and corresponds to the critical loop L_1. The precedence relation of the DFG is shown in Fig. 2.10(b), and the length of the critical path (or equivalently the iteration period for this DFG) is 20 units.

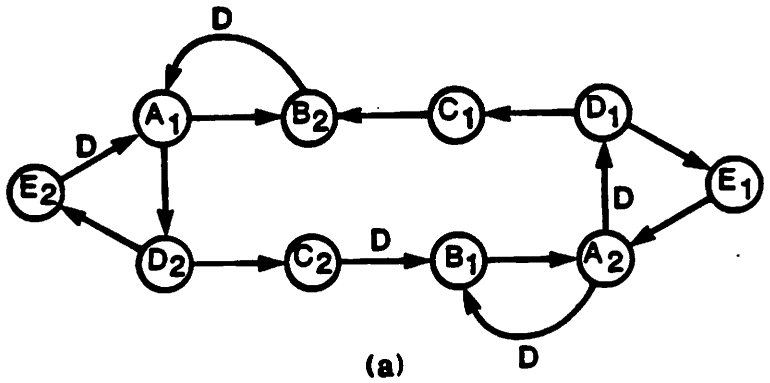


(a)



(b)

Fig. 2.10: (a) A DFG. The node execution times are 20, 5, 10, 10, and 2 units for nodes *A*, *B*, *C*, *D*, and *E* respectively, (b) Precedence graph.



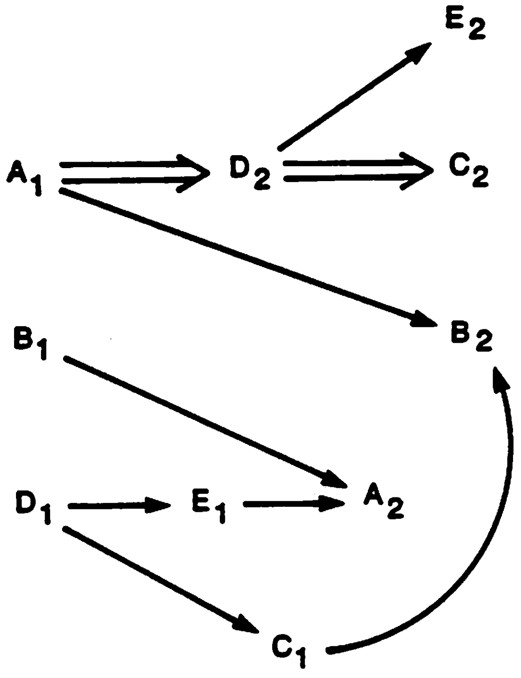


Fig. 2.11: (a) Unfolded DFG for the DFG in Fig. 2.10(a) with unfolding factor 2, (b) Precedence graph

Since the number of delays in the critical loop is 2, we construct an equivalent unfolded DFG with $J = 2$ as shown in Fig. 2.11(a). The precedence graph for the unfolded DFG is shown in Fig. 2.11(b), and leads to an iteration period of 20 units. We can improve the iteration period by retiming the unfolded DFG (since this unfolded DFG is not a perfect graph). Fig. 2.12(a) shows the retimed version of the unfolded DFG, and Fig. 2.12(b) shows the corresponding precedence relation. From the critical path in the unfolded DFG, we observe that the cycle time corresponds to 35 units, or equivalently the iteration period is $\frac{35}{2} = 17.5$ units (which is greater than the bound by 1.5 units). This is the minimum iteration period that can be achieved with an unfolding factor of 2.

Now we proceed to prove that an unfolding factor given by the least common multiple of the number of delays of the loops in a DFG will lead to a perfect unfolded DFG, which can then be scheduled in a fully-static rate-optimal manner without requiring any retiming at all. Before we prove this, let us consider the example of the DFG in Fig. 2.10(a). Since the delay counts in the maximal loops are 2 ad 3 respectively, the least common multiple is 6, and hence we need an unfolding factor of 6 to obtain a rate-optimal schedule. The unfolded DFG is shown in Fig. 2.13(a) for an unfolding factor of 6, and one can verify that it is indeed a perfect graph. The precedence graph of this unfolded DFG is shown in Fig. 2.13(b), and the length of the critical path is 96, which corresponds to an iteration period of 16 units, equal to the iteration bound. Now we proceed to prove that fully-static rate-optimal scheduling of DFGs is admissible, and we then derive an upper bound on the number of processors to achieve a rate-optimal schedule.

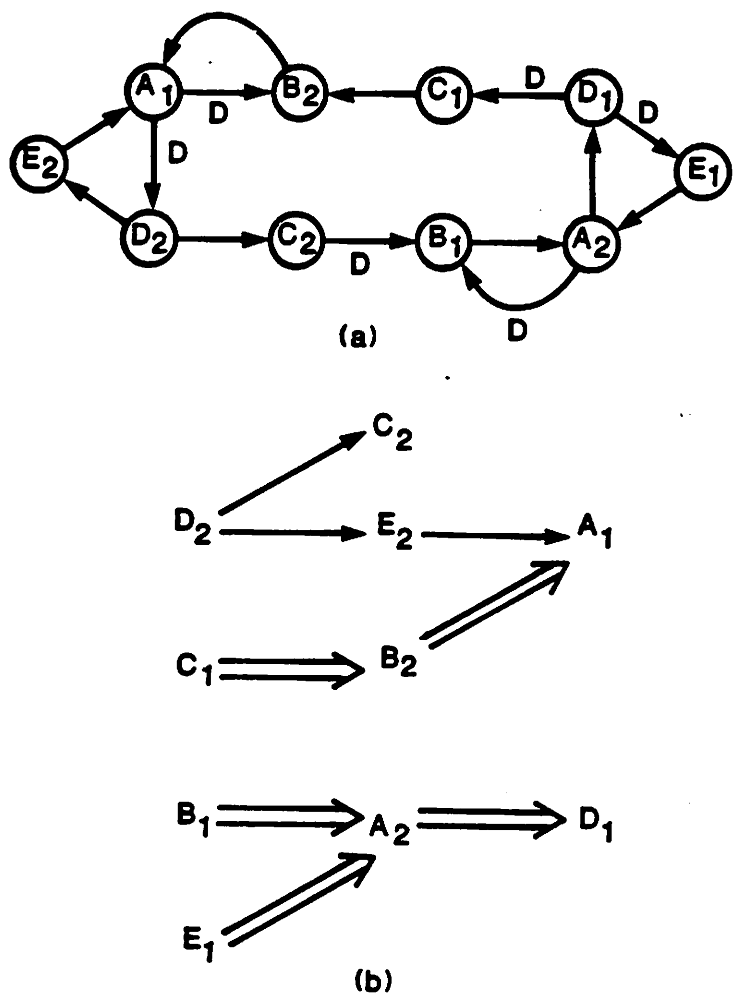


Fig. 2.12: (a) Retimed version of the unfolded DFG in Fig. 2.11(a), (b) Precedence graph

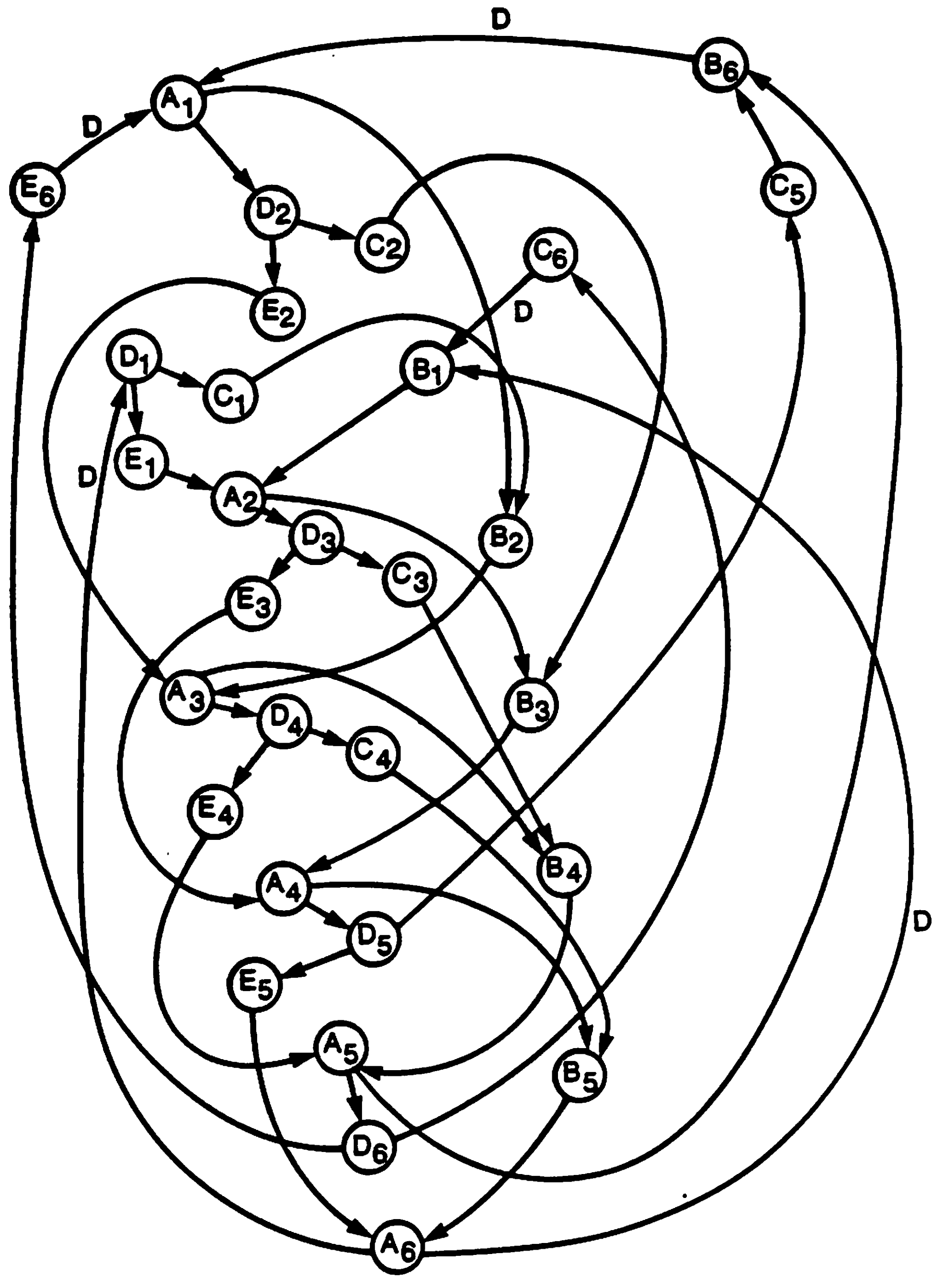


Fig. 2.13(a): Unfolded DFG for the DFG in Fig. 2.10(a) for unfolding factor of six.

Theorem 2.3: Any unfolded DFG with an unfolding factor equal to the least common multiple of the loop delay counts of the original DFG corresponds to a perfect graph.

Proof: Since the unfolding factor is the least common multiple of the delay counts, it is a multiple of the delay count in every loop. From property 2.4, any loop with D delays must transform to D distinct maximal loops in the unfolded DFG, and all loops in the unfolded DFG must have a single delay inside each loop. Since all loop delay counts in the original DFG are less than the unfolding factor, any linear additive combination will also lead to a single loop delay in the unfolded DFG. Since every loop in the unfolded DFG has a single delay, the unfolded DFG is a perfect graph. From theorem 2.1, the nodes in perfect graphs can always be scheduled in a fully-static rate-optimal manner. □

Theorem 2.4: Recursive nodes (i.e. nodes belonging to one or more loops) of any DFG can be scheduled in a rate-optimal fully-static manner by using at most P processors, where P is the sum of the delay counts in all the maximal loops in the original DFG.

Proof: Since the unfolding factor is the least common multiple of the delay counts in all the loops, each maximal loop with K delays transforms to K distinct maximal loops in the unfolded DFG. Thus, the upper bound on the number of distinct maximal loops in the unfolded DFG (which is a perfect graph) is equal to the sum of the delay counts in all the maximal loops of the original DFG. This is the upper bound on the number of processors to schedule all the recursive nodes, since the upper bound on the number of processors to achieve a rate-optimal fully-static schedule in a perfect graph is equal to the number of maximal loops.

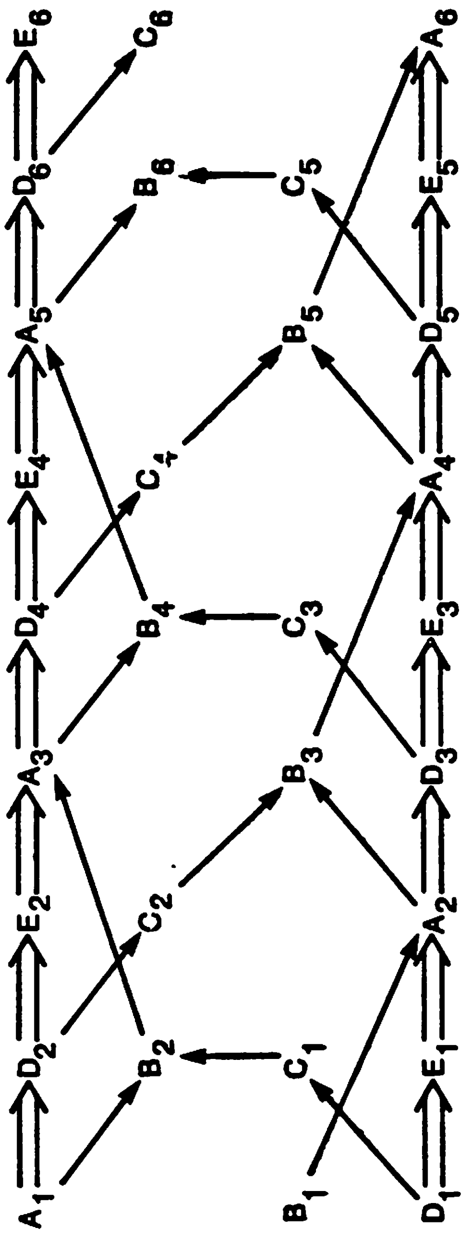


Fig. 2.13(b): Precedence graph of the unfolded DFG in Fig. 2.13(a).

Example 2.7: Fig. 2.13(b) shows the precedence graph of the unfolded DFG of Fig. 2.13(a). From the precedence graph, it is clear that the rate-optimal fully-static schedule can be achieved with 4 processors. The upper bound on the number of processors for this example is 5, since the 2 maximal loops in the original DFG contain respectively 2 and 3 loop delays. □

2.8. MULTIPLE-RATE DFGS

Multiple-rate DFG representations are useful in many signal and image processing applications, typically in interpolation and decimation schemes. In such systems, different nodes are invoked different number of times in a cycle; the nodes consume different number of samples from each input arc, and produce different number of samples to each outgoing arc [7].

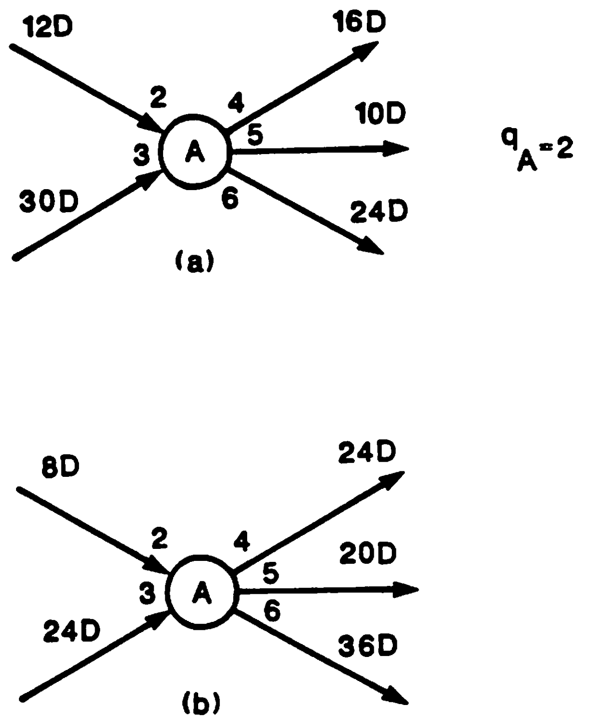


Fig. 2.14: Local retiming in multirate DFG. The node *A* is executed two times in each cycle.

Fig. 2.14(a) shows an example of a node of a multiple-rate DFG, which shows that each invocation of node A consumes 2 and 3 samples from the two input arcs, and produces 4, 5, and 6 samples to the three outgoing arcs respectively. The numbers associated with the arcs near the nodes represent number of samples consumed or produced and the numbers in the middle of any arc represent the number of delays or buffer locations associated with that arc. Any multiple-rate DFG can be equivalently described in terms of a homogeneous or single-rate DFG. Therefore, the framework discussed so far will be directly applicable to the homogeneous equivalents of the multiple-rate DFGs. From the equivalent homogeneous DFG, we can determine the iteration bound, optimum unfolding or blocking factor, and the processor bound of the multiple-rate DFG. The objective of this section is to extend the notions of retiming to the case of multiple-rate DFGs, and to use this extension to find an upper bound on the unfolding factor for the multiple-rate DFGs, thereby bypassing the need to obtain an equivalent homogeneous DFG.

In homogeneous DFGs, removal of C number of delays from each input arc and addition of C delays to each outgoing arc of a node constitutes a valid local retiming [15] or cutset transformation [16] (where C is some fixed number). In multirate DFGs, removal of Q_i number of delays from the i-th incoming arc, and addition of Q_j number of delays to the j-th outgoing arc constitutes a valid local retiming, where

$$Q_i = Cqb_i \text{ , } Q_j = Cqa_j \text{ ,}$$

and q is the number of invocations of the node in each cycle, a_j is the number of samples produced by the node to its j-th outgoing arc, and b_i is the number of samples consumed from the i-th incoming arc by the node in each invocation. An example of a local retiming at a node in a multirate DFG is illustrated in Fig. 2.14. The node A is assumed to be executed twice in a cycle. One invocation of A consumes 2 and 3 samples respectively from the two incoming arcs, and produces 4, 5, and 6 samples to each of the outgoing arcs respectively. The retimed graph is obtained by removing 4 and 6 delays from the two incoming arcs, and by adding 8, 10, and 12 delays to the three outgoing arcs respectively.

In a homogeneous DFG, the retiming does not change the number of delays in a loop. We can easily derive a similar condition for the multirate DFG case. We define a *normalized delay* for the i-th arc $u \rightarrow v$ as

$$\hat{D}_i = \frac{D_i}{a_i q_u} = \frac{D_i}{b_i q_v}, \tag{2.9}$$

where $\hat{D}_i$ is the normalized delay of the i-th arc, D_i is the number of delays associated with the i-th arc of the multirate DFG, a_i is the number of samples produced by the node u in each invocation on arc i, b_i is the number of samples consumed by node v in each invocation from the arc i; and q_u and q_v respectively represent the number of invocations of nodes u and v in each cycle. It is easy to verify that the retiming operation conserves the sum of the normalized delays in any loop in a multirate DFG. One can also verify that the total number of delays in a maximal loop of the homogeneous equivalent DFG is equal to the sum of the normalized delays in the corresponding loop in the multirate DFG. Therefore, from our earlier results, the upper bound on the unfolding factor is the least common multiple of the normalized loop delay counts in the maximal loops of the multirate DFG. A loop in a multirate DFG is maximal if it does not contain any transitive arc. An arc $u \rightarrow v$ in a multirate DFG is transitive if there exists a path from u to v such that the sum of the normalized delays in the path equals the normalized delay of the arc $u \rightarrow v$.

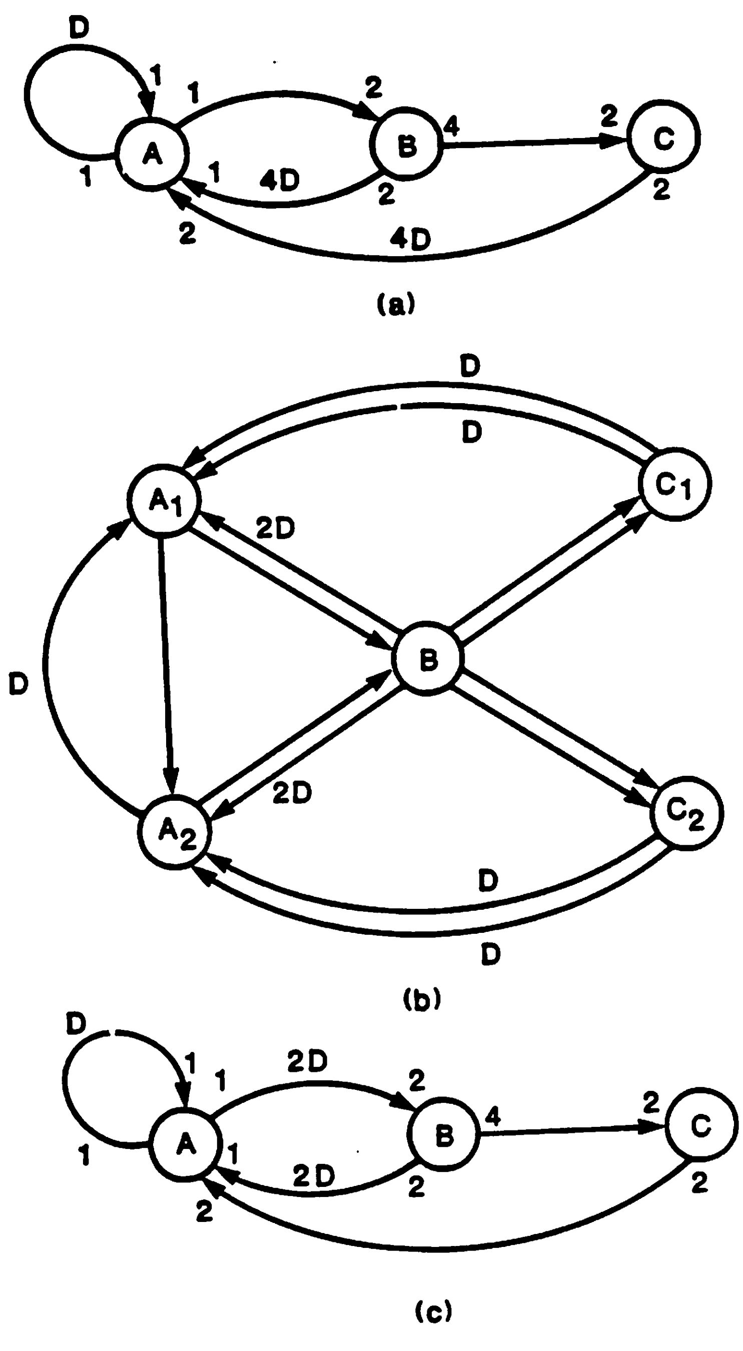


Fig. 2.15: (a) A multirate DFG, (b) An equivalent homogeneous DFG, (c) A retimed multirate DFG.

Example 2.8: Consider the multirate DFG in Fig. 2.15(a). In each cycle, the nodes A, B and C are respectively invoked 2, 1, and 2 times. This DFG contains 2 maximal loops $A \rightarrow B \rightarrow A$ and $A \rightarrow B \rightarrow C \rightarrow A$, and the normalized delay counts in the loops are respectively 2 and 1. Fig. 2.15(b) shows the equivalent homogeneous DFG. One can verify that the loop delay counts in the maximal loops in the equivalent homogeneous DFG are 2 and 1 also. Fig. 2.15(c) shows a retimed version of the multirate DFG, which is obtained by applying a local retiming operation at node A in the multirate DFG of Fig. 2.15(a). □

2.9. NON-HOMOGENEOUS PROCESSOR SCHEDULING

The rate-optimal fully-static scheduling approach presented in this chapter is based on ideal inter-processor interconnection, availability of large number of processors, and homogeneity of the processors with respect to functionality and speed. Often real implementations are based on a fixed number of processors, or fixed interconnections, or processors non-homogeneous in functionality and/or speed. Unfortunately, the rate-optimal schedules under these assumptions belong to the class of NP-Complete problems [25]. Most of these problems will need to be solved by the use of heuristics. The objective of this section is to give an example to show that the program unfolding approach can be used to exploit fine-grain parallelism, and to obtain efficient schedules in the context of non-homogeneous processor implementations.

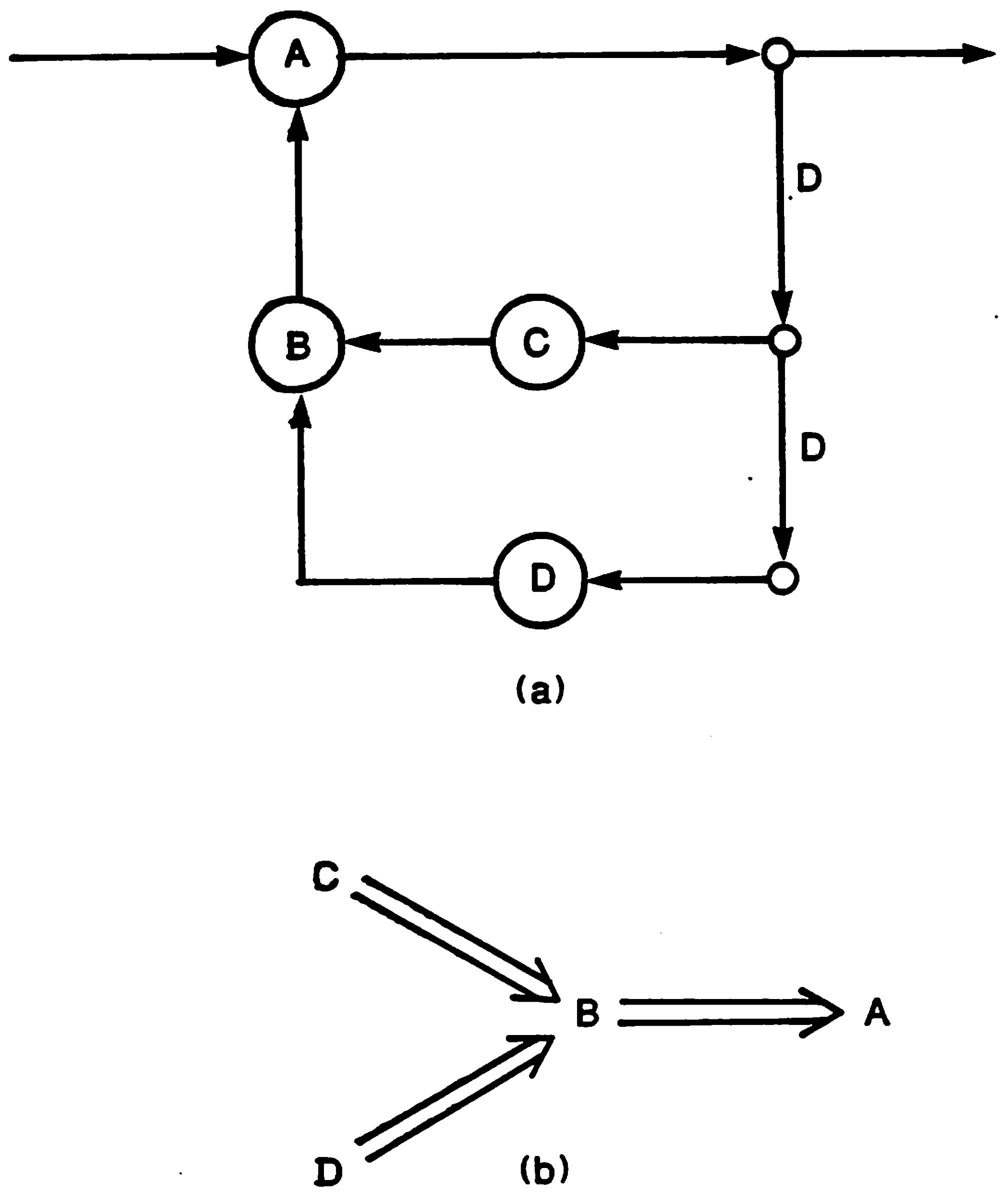


Fig. 2.16: DFG corresponding to a second order all pole digital filter

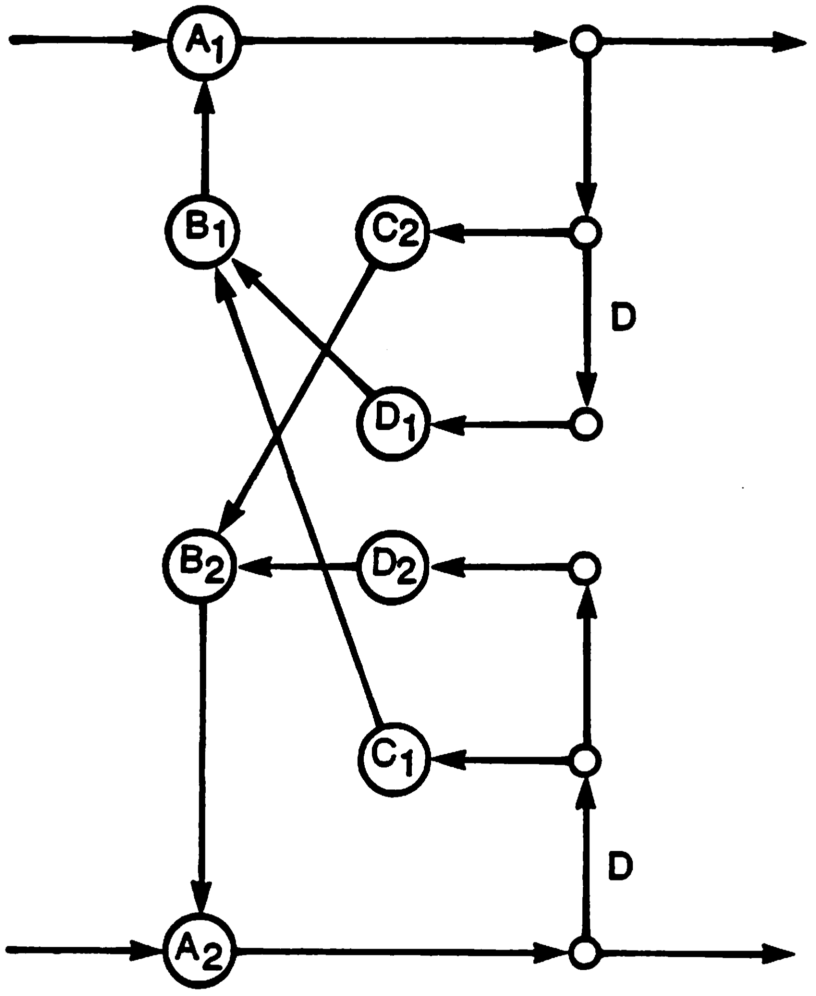
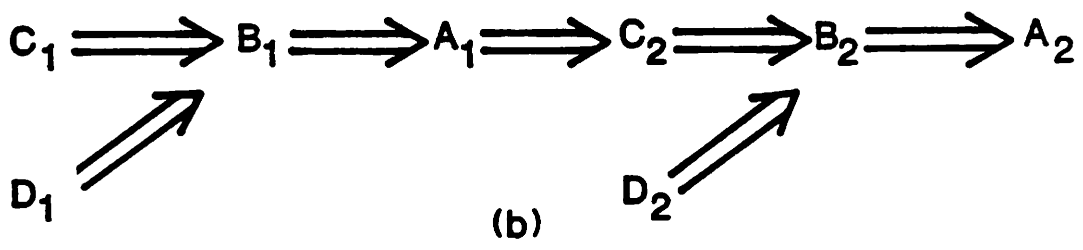


Fig. 2.17(a): Unfolded DFG of the all pole filter for unfolding factor of two.



$$\begin{tikzcd}
C_1 \arrow[r, Rightarrow] & B_1 \arrow[r, Rightarrow] & A_1 \arrow[r, Rightarrow] & C_2 \arrow[r, Rightarrow] & B_2 \arrow[r, Rightarrow] & A_2 \\
D_1 \arrow[ur, Rightarrow] & & & D_2 \arrow[ur, Rightarrow] & &
\end{tikzcd}$$

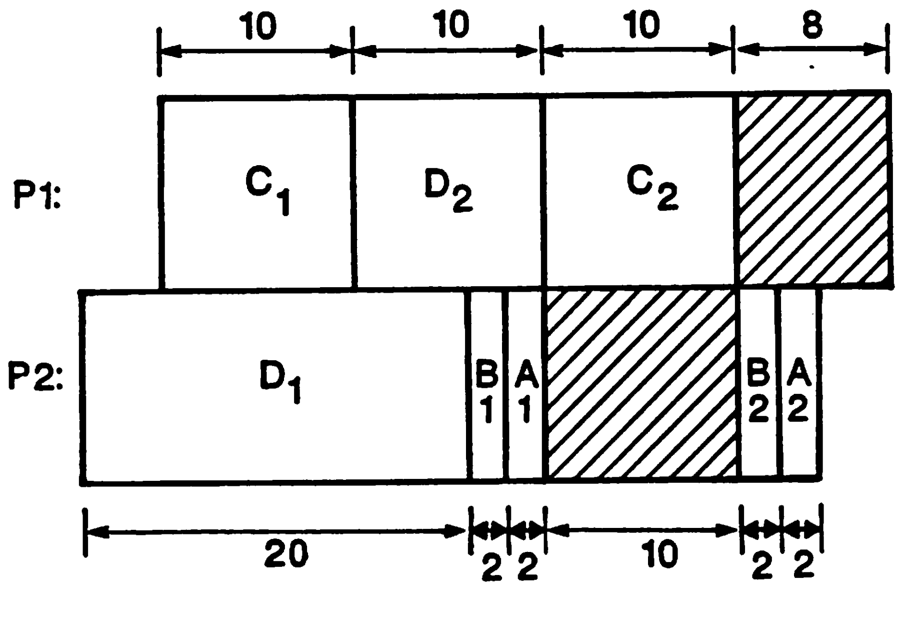


Fig. 2.17: (b) Precedence graph of the unfolded DFG in Fig. 2.17(a), (c) Partial schedule for non-homogeneous processors.

Consider the example of a second order all-pole filter DFG shown in Fig. 2.16. In the figure, the small circles represent fork nodes, the operations A and B are addition operations, and C and D are multiplication operations. Let us assume that we are given two processors; processor P_1 capable of performing only multiplications in 10 units of time, and the processor P_2 capable of performing addition in 2 units of time and multiplication in 20 units of time (i.e. processor P_2 is slower). Simple CPM schedule for this case would require an iteration period of 24 units of time (as evident from the precedence graph of Fig. 2.16). However, we can obtain an iteration period of 19 units by first obtaining the unfolded DFG with an unfolding factor of 2 and then by scheduling the tasks as illustrated in Fig. 2.17. We are able to exploit concurrency to the finest possible granularity, because the unfolding by a factor of 2 reduced the second order all pole filter to an equivalent perfect data-flow program graph. The non-homogeneous processors lead to efficient hardware utilization, because we are able to assign a slower processor to the tasks corresponding to the outer non-critical loop and a faster processor to the tasks of the innermost or critical loop.

2.10. CONCLUSION

The major contribution of this chapter is finding the optimum unfolding factor for any data-flow program, and the use of this unfolding to systematically prove the existence of fully-static rate-optimal multiprocessor schedules in iterative data-flow program models. This is an important result, since the existence of rate-optimal fully-static schedules had so far remained an open question. This approach also clearly shows the synergy between the existing cyclo-static scheduling and our approach to fully-static scheduling. This synergy is demonstrated by the fact that the scheduling of several iterations of a specified task in a single cycle are carried out by different processors, and hence are not fully-static within the cycle of the unfolded program. With optimum unfolding, we can construct rate-optimal fully-static multiprocessor schedules *without requiring any retiming* operation at all. It is hoped that the approach provided in this chapter will find use in task scheduling and synthesis of multiprocessor programmable and/or custom VLSI digital signal processors, both in the context of homogeneous and non-homogeneous processors.

In this chapter, we have established the notion of the iteration bound in recursive or iterative algorithms. The subsequent chapters focus on linear recursive systems, and are devoted to developing algorithm transformation techniques, which can find equivalent alternative hardware-efficient architectures for these systems.

2.11. REFERENCES

(1) Karp, R.E., and Miller, R.E., "Properties of a Model for Parallel Computations: Determinacy, Termination, and Queueing", *SIAM Journal of Applied Mathematics*, Vol. 14, No. 6, November 1966, pp. 1390-1411

(2) Miller, R.E., "Scheduling Parallel Computations", *Journal of the Association for Computing Machinery*, Vol. 15, No. 4, October 1968, pp. 590-599

(3) Crochiere, R.E., and Oppenheim, A.V., "Analysis of Linear Digital Networks", *Proc. of the IEEE*, Vol. 63, No. 4, April 1975, pp. 581-595

(4) Dennis, J.B., "Data Flow Supercomputers", *IEEE Computer*, November 1980, pp. 48-56

(5) Davis, A.L., and Keller, R.M., "Data Flow Program Graphs", *IEEE Computer*, Vol. 15, No. 2, Feb. 1982, pp. 26-41

(6) Ackerman, W.B., "Data Flow Languages", *IEEE Computer*, Vol. 15, No. 2, Feb. 1982, pp. 15-25

(7) Lee, E.A., and Messerschmitt, D.G., "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. C-36, No. 1, January 1987, pp. 24-35,

(8) Kung, S.Y., Lewis, P.S., and Lo, S.C., "Performance Analysis and Optimization of VLSI Data Flow Arrays", *Journal of Parallel and Distributed Computing*, Vol. 4, 1987, pp. 592-618

(9) Fettweis, A., "Realizability of Digital Filter Networks", *Arch. Elek. Ubertragung*, Feb. 1976, pp. 90-96

(10) Renfors, M., and Neuvo, Y., "The Maximum Sampling Rate of Digital Filters under Hardware Speed Constraints", *IEEE Trans. on Circuits and Systems*, Vol. CAS-28, No. 3, March 1981, pp. 196-202

(11) Barnwell, T.P. III, and Hodges, C.J.M., "Optimal Implementation of Signal Flow Graphs on Synchronous Multiprocessors", *Proc. of 1982 International Conf. on Parallel Processing*, Belaire, MI, August 1982

(12) Ramamoorthy, C.V., and Ho, G.S., "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, September 1980, pp. 440-449

(13) Brafman, J.P., Szczupak, J., and Mitra, S.K., "An Approach to Implementation of Digital Filters using Microprocessors", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. 26, No. 5, Oct. 1978, pp. 442-446

(14) Zeman, J., and Moschytz, G.S., "Systematic Design and Programming of Signal Processors using Project Management Techniques", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. 31, No. 6, December 1983, pp. 1536-1549

(15) Leiserson, C.E., Rose, F., and Saxe, J., "Optimizing Synchronous Circuitry by retiming", *Third Caltech Conference on VLSI*, Pasadena, CA, March 1983, pp. 87-116

(16) Kung., S.Y., "On Supercomputing with Systolic/Wavefront Array Processors", *Proceedings of IEEE*, Vol. 72, No. 7, July 1984

(17) Schwartz, D.A., and Barnwell, T.P. III, "A Graph Theoretic Technique for the Generation of Systolic Implementations for Shift Invariant Flow Graphs", *Proc. of ICASSP-84*, San Diego, March 1984

(18) Schwartz, D.A., "Synchronous Multiprocessor Realizations of Shift Invariant Flow Graphs", *Ph.D. Dissertation*, Georgia Institute of Technology, Technical Report DSPL-85-2, July 1985

(19) Schwartz, D.A., and Barnwell, T.P. III, "Cyclostatic Multiprocessor Scheduling for the Optimal Implementation of Shift Invariant Flow Graphs", *Proc. of ICASSP-85*, Tampa, FL, March 1985

(20) Schwartz, D.A., "Cyclo-Static Realizations: Loop Unrolling and CPM, Optimal Multiprocessor Scheduling", *Proc. of the 1987 Princeton Workshop on Algorithms, Architecture, and Technology Issues in Models of Concurrent Computations*, Sept. 30 - Oct. 1, 1987

(21) Lee, S.H., and Barnwell, T.P. III, "Optimal Multiprocessor Implementation from a Serial Algorithm Specification", *Proc. of ICASSP-88*, NY, April 1988, pp. 1694-1697

(22) Forren, H., and Schwartz, D.A., "Transforming Periodic Synchronous Multiprocessor Programs", *Proc. of ICASSP-87*, Dallas, TX, April 1987, pp.1406-1409

(23) Rao, S.K., "Regular Iterative Algorithms and their Implementation on Processor Arrays", *Technical Memorandum*, Information Systems Laboratory, Stanford University, Ph. D. Dissertation, October 1985

(24) Coffman, E.G. Jr. (ed.), *Computer and Job Scheduling Theory*, Wiley, New York, 1976

(25) Garey, M.R., and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., New York, 1979

3

PIPELINING IN RECURSIVE FILTERS

3.1. INTRODUCTION

In order to exploit VLSI for high performance dedicated system implementations, we need to understand the characteristics of the scaled VLSI technologies. For example, VLSI offers a greater potential for complexity than speed, favors replication of one function, and imposes a high cost in performance for non-localized communication. Design costs can be minimized by composing the system as a replication of simple processing elements. These considerations favor implementations which feature arrays of identical or easily parametrized processing elements (since, these are easily given a software procedural definition) with mostly localized interconnections (for reduced communication costs). This has led to an interest in systolic- and wavefront-array implementations [1,2].

High performance can be achieved by either using exotic high speed technologies, such as bipolar or GaAs which allow us to gain performance without modification of the algorithm. On the other hand, we can use a low cost VLSI technology such as CMOS and yet gain impressive performance by exploiting concurrency. Concurrency is usually manifested in the form of pipelining or parallelism or both. Concurrent architectures can be derived by implementing the existing algorithms in new ways. To be more precise, we do not change the transfer function or the input-output characteristics of the algorithm, but we do change the internal structure of the algorithm, thereby impacting the finite precision effects but nothing else. This is referred to as *recasting the structure* of the algorithm. Different forms of recasting a specified algorithm can lead to realizations with entirely different properties and implementation complexities. In this chapter, we show that *appropriately* recasting the structure of an algorithm can have a dramatic effect on the performance of an implementation.

The challenge in achieving high performance implementations is mostly in recursive systems, since the recursion or the internal feedback negates the most obvious ways of improving performance. This is because the computational latency associated with the feedback loop in recursive systems limits the opportunities for pipelining and/or parallel processing. In non-recursive systems, we can place latches across any *feed-forward cutset* without changing the transfer function (at the expense of latency) and achieve desired level of pipelining. However, recursive systems cannot be pipelined at arbitrary level by simply inserting latches, since the pipelining latches would change the number of delay operators in the loop, and hence the transfer function of the implementation. We can overcome this recursive bottleneck by changing the internal structure of the algorithm to *create* additional logical delay operators inside the recursive loop, which can then be used for pipelining.

High sampling rate realizations of recursive digital filters using block processing have been suggested [3-17]. In block processing, input samples are processed in the form of non-overlapping blocks and outputs are also generated block by block. We can increase the block size arbitrarily to achieve arbitrarily high sampling rate recursive system realizations. The best known block structures reported so far for recursive digital filtering require a square multiplication complexity with respect to the block size. In the next chapter, we discuss block processing structures, and derive our new incremental block filter with linear complexity in block size.

Loomis and Sinha recently used the concept of block processing to derive a pipelined realization of direct form recursive digital filters [18]. Similar approaches have also been followed for recursive filter implementations using charge domain devices [19]. The block state update operation as well as the pipelining technique used by Loomis and Sinha belong to the class of *look-ahead computation* techniques [20], and lead to a linear complexity with respect to the block size or number of loop pipeline stages. This look-ahead process is referred to as *clustered look-ahead* throughout this chapter.

We use look-ahead and decomposition algorithms to pipeline a first-order system (i.e. where the state $x(n)$ is expressed as a function of $x(n-1)$) [21-23]. In the look-ahead scheme, the algorithm is iterated as many times as desired to create the necessary level of concurrency, and the iterated version is implemented. Specifically, for the first order recursion, the state $x(n)$ is expressed as a function of $x(n-M)$ to create M delay operators inside the loop so that the loop can be pipelined by M stages. This iteration process contributes to a non-recursive $O(M)$ multiplication complexity. For cases where M can be expressed as a power of 2, a *decomposition technique* is proposed to implement the non-recursive overhead in a decomposed manner to obtain a logarithmic multiplication complexity. This first order pipelined realization was also studied in [24] in the context of static data-flow computer implementation, but without the decomposition technique.

In this chapter, we study efficient pipelining of higher order recursive systems. In an N-th order recursive system, the state $x(n)$ is expressed as a function of the past N states $x(n-1)$, $x(n-2)$, ..., and $x(n-N+1)$. There are two types of look-ahead schemes in the context of higher order filters; they include clustered look-ahead and scattered look-ahead. In the clustered look-ahead pipelining scheme [18], the algorithm is iterated to express the state $x(n)$ as a function of N past consecutive or clustered states $x(n-M)$, $x(n-M-1)$, ..., and $x(n-M-N+1)$. This look-ahead process creates M loop delay operators, which can be used to pipeline the loop by M stages. In this technique, the original N-th order filter is emulated by an $(N+M-1)$-th order filter ($(M-1)$ canceling poles and zeros have been added). The multiplication complexity of the resulting pipelined filter is $O(M)$, which is linear with respect to M. Since the pipelined filter is derived by adding poles and zeros, some of the modes or eigenvalues are either uncontrollable or unobservable or both. Unfortunately, for higher order systems, the clustered look-ahead process does not guarantee all the additional poles to lie inside the unit circle, and hence does not guarantee stability.

We introduce a new *scattered look-ahead* approach to derive stable pipelined filters. In this new look-ahead process, we express $x(n)$ as a function of past N *scattered* states $x(n-M)$, $x(n-2M)$, ..., and $x(n-NM)$, thus emulating the original N-th order filter by an NM-th order filter. Note that the clustered look-ahead and the scattered look-ahead approaches are identical for the first order case (since in both cases $x(n)$ is expressed as a function of $x(n-M)$). In the scattered look-ahead process, for *each* existing pole in the original filter, we add $(M-1)$ additional poles (and zeros at identical locations) with equal angular spacing at a distance from the origin same as that of the original pole. The scattered look-ahead process leads to $O(NM)$ complexity (much larger than that for clustered look-ahead), but guarantees stability. For cases where M can be expressed as a power of 2, we use the *decomposition technique* to implement the non-recursive portion with $O(N \log_2 M)$ multiplication complexity. The upper bound on roundoff noise in these pipelined filters improves with M. Based on the scattered look-ahead and the decomposition techniques, we derive pipelined realizations of direct form and state space form recursive digital filters. Several pipelined bidirectional systolic arrays for recursive digital filtering have been proposed in [25-29]. However these structures require interleaving of independent time series when pipelined. In this chapter, we present fully pipelined and fully hardware efficient linear bidirectional and unidirectional ring systolic arrays for recursive filtering using the scattered look-ahead technique.

The organization of this chapter is as follows. The iteration period bound in recursive computations is reviewed in section 3.2. In section 3.3, we review the notion of pipeline interleaving in the context of recursive digital filtering. Sections 3.4 and 3.5 address pipelined realization of direct form and state space form linear time-invariant recursive digital filters respectively using scattered look-ahead and the decomposition techniques.

3.2. ITERATION BOUND REVISITED

Let S_l represent the set of loops in the recursive computation graph, D_l represent the latency associated with the computation in loop l, and M_l represent the number of latches or logical delay operators inside the loop l. Let each latch in the computation graph be $L-slow$, i.e. the clock rate of each latch is L times slower than the sample rate, or equivalently the implementation corresponds to a block implementation with block size L (the block size is assumed to be constant throughout). Then the iteration period bound is given by

$$T_\infty = \frac{1}{L} \underset{l \in S_l}{MAX} \left[\frac{D_l}{M_l} \right] . \tag{3.1}$$

The *maximum achievable sampling rate* for the computation graph is $\frac{1}{T_\infty}$. The loop l_0 for which $T_\infty = \frac{D_{l_0}}{LM_{l_0}}$ is satisfied is called the *critical loop*. For unity block size (i.e., $L = 1$), this definition reduces to the iteration bound definition in [30-31].

The iteration period bound can be improved by increasing either the number of pipeline stages inside the recursive loop (M_l), or the block size (L) or both. In the sequel, we assume M_l to be same for all loops and refer to it as M. By using M pipeline stages inside the recursive loop and a block size of L, the sample rate can be increased by a factor LM. Since, pipelined realizations can be achieved with logarithmic increase in hardware (as opposed to linear increase as in block processing), it is efficient to use pipelined algorithms (i.e. with $L = 1$) first for high speed IIR filter implementations, and then combine block processing with pipelining only if sufficient speed cannot be generated by using pipelining alone. Thus, block processing in itself is an inefficient way of implementing high-speed *custom* IIR digital filters. However, block processing is useful for software-programmable implementations on general-purpose coarse-grain multiprocessors.

3.3. PIPELINE INTERLEAVED DIGITAL FILTERS

Pipeline interleaving notion is an old idea, and has been used in general purpose computers. Pipeline interleaving approach has also been proposed for programmable implementation of signal processing systems using deeply pipelined programmable digital signal processors [32], and for cyclostatic implementation of these systems [33].

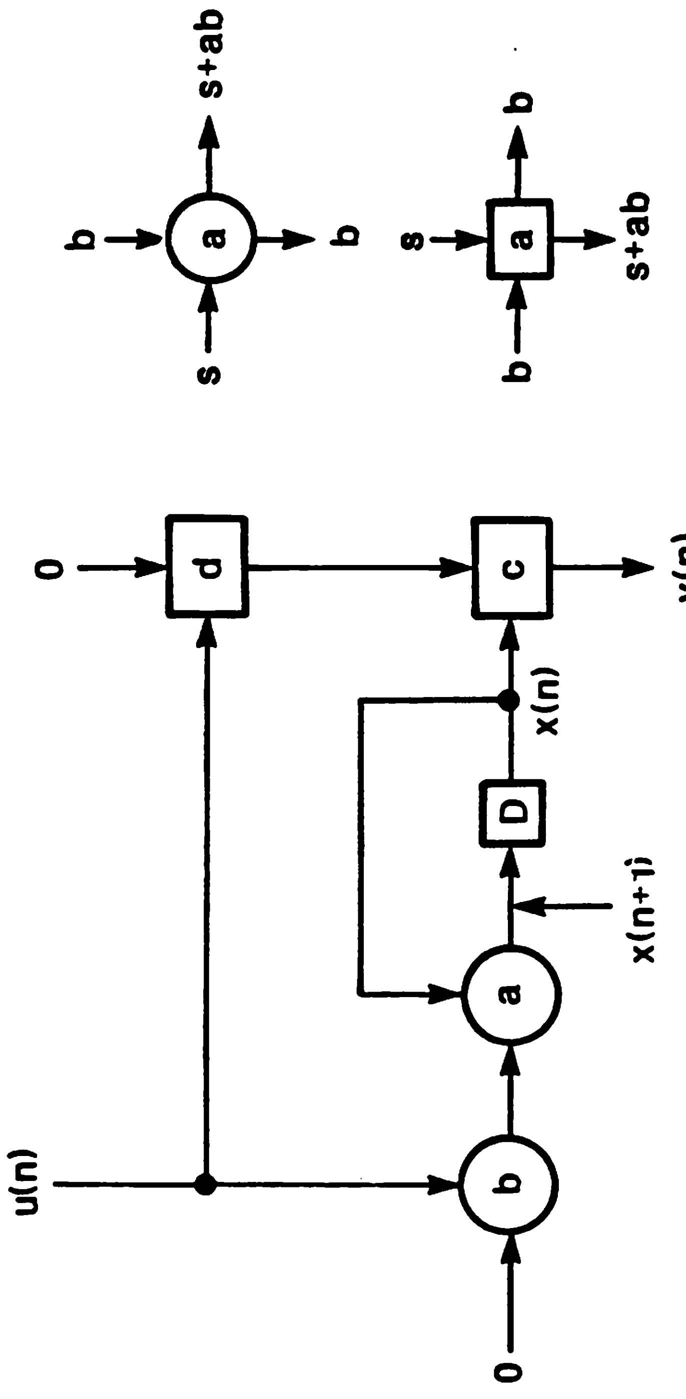


Fig. 3.1(a): A Simple first order LTI recursion.

In this section, we review the notion of pipeline interleaving in the context of a simple first order recursive digital filter. In particular, we discuss three forms of pipeline interleaving, (i) Inefficient Single/multi-channel interleaving, (ii) Efficient single channel interleaving, and (iii) Efficient multi-channel interleaving. In (i), the loop is pipelined without changing the structure of the algorithm and hardware is not fully utilized, since zero samples need to be interleaved to preserve the integrity of the algorithm. In (ii) and (iii), the internal structure of the algorithm is changed in a way that the pipeline is maximally or fully utilized.

3.3.1. Inefficient Single/Multi-Channel Interleaving

Consider a first-order linear time-invariant recursion described by

$$x(n+1) = ax(n) + bu(n) \tag{3.2}$$

and shown in Fig. 3.1(a) in the form of a computation graph. The iteration period bound of this computation graph is $(T_m + T_a)$, where T_m and T_a respectively represent the word-level multiplication time, and addition time. Consider obtaining a M*−stage* pipelined version of this implementation by placing or *inserting* $(M-1)$ additional latches inside the loop as shown in Fig. 3.1(b) (at the appropriate places). Then the clock period of this implementation can, in principle, be reduced by M times, but the latency associated with the loop computation and the sample period of the implementation will increase to M clock periods. As an example for $M = 5$, if we begin with a state $x^1(0)$ in clock period 0, the next state $x^1(1)$ will be available in clock period 5. Hence for the case of a single time series, this array will be useful for only 20% of the time. (Trying to input samples of a single time series each clock period would implement a different algorithm, since the number of logical delays inside the loop has been changed.)

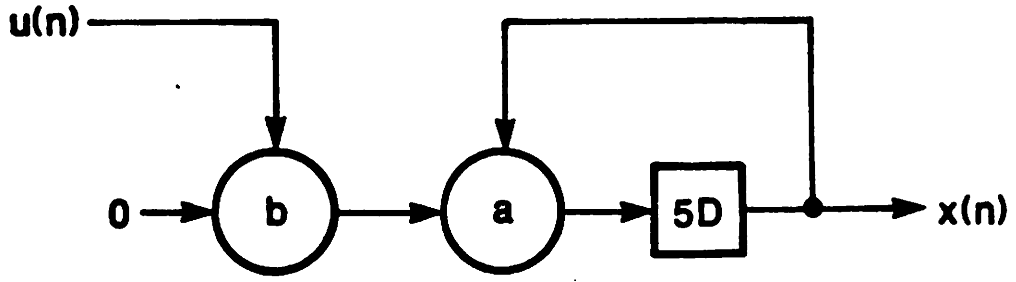
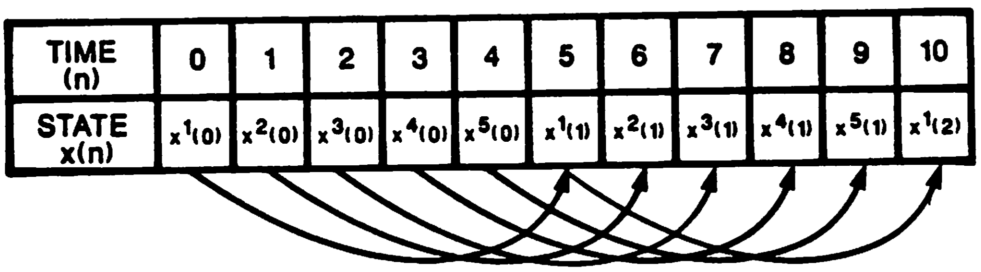


Fig. 3.1(b): The first order LTI recursion after inserting $(M-1)$ delay operators inside the loop (for $M = 5$).



TIME (n)	0	1	2	3	4	5	6	7	8	9	10
STATE x(n)	$x^1(0)$	$x^2(0)$	$x^3(0)$	$x^4(0)$	$x^5(0)$	$x^1(1)$	$x^2(1)$	$x^3(1)$	$x^4(1)$	$x^5(1)$	$x^1(2)$

Fig.3.1(c): A partial schedule for the implementation in (b). The input time series are 5-way interleaved, i.e. 5 independent time series are being filtered simultaneously. The state $x^i(n)$ corresponds to the state of the i-th time series at time index n.

Hence, the sampling rate of this implementation is 5 times slower than the clock rate, and is no higher than that of the unpipelined version (in fact is worse due to the delay time introduced due to the additional latches). However, if 5 independent time series are available to be filtered by the same hardware, then the hardware can be fully utilized as shown in the schedule of Fig. 3.1(c), although all the independent time series must be filtered at the slow rate. Independent time series can correspond to outputs of each first or second order cascade stage (since these elements can be separated by a feed-forward cutset), or can correspond to independent channels requiring identical filtering operation. As an example, for a 10-th order recursive filter implemented as cascaded second order sections, the five section outputs are independent and can be interleaved in the pipeline (of course, each at 5-slow rate). Thus pipeline interleaving approach is well suited for applications requiring nominal concurrency. To conclude, if a recursive loop with a single delay element is pipelined by M-stages by *inserting* $(M-1)$ additional delay elements, then the input data must be M-way interleaved, i.e. $(M-1)$ zero time series or independent time series are interleaved with the given data stream (otherwise, the transfer function of the algorithm will be changed), and nothing has been achieved with respect to the sample rate with which a single time series can be filtered. This implementation has also been referred to as M-slow circuit in the literature [1,34-36]. The hardware in this slow interleaved implementation is inefficiently utilized if M independent computations are not available to be interleaved.

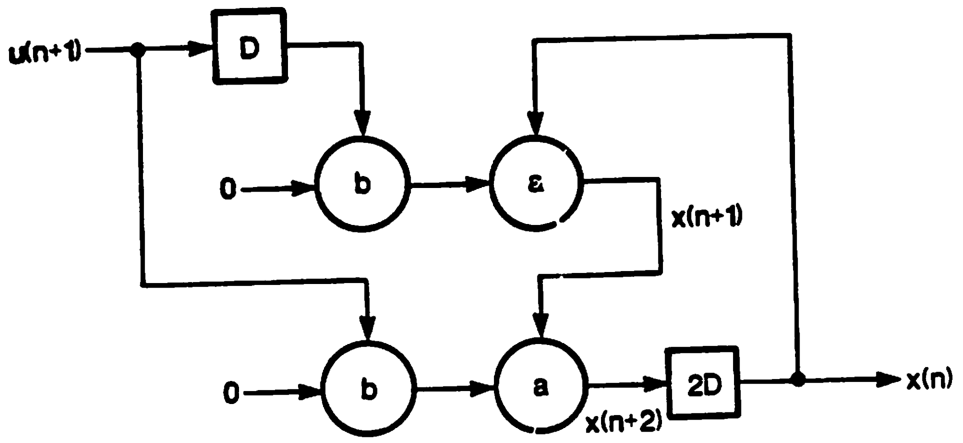


Fig. 3.2(a): An equivalent realization of Fig. 3.1(a) obtained without the use of the look-ahead transformation.

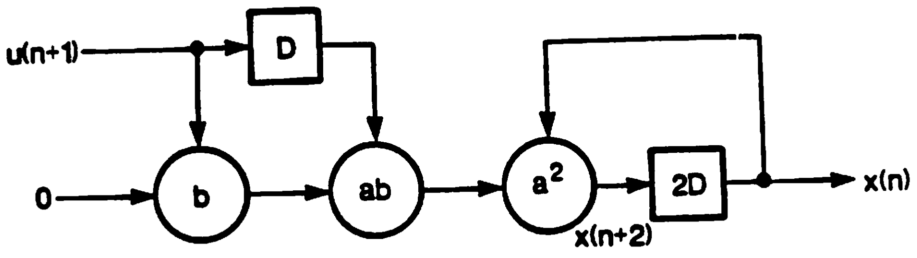


Fig.3.2(b): Another equivalent first order LTI recursion obtained with the use of look-ahead computation.

3.3.2. Efficient Single-Channel Interleaving

Ruling out the interleaving of independent time series, the two problems with M-slow implementations are (i) a sampling rate M times slower than the clock rate, and (ii) inefficient utilization of processing elements. Now we show that both these problems can be overcome by using the look-ahead transformation [20,21], in which the given linear recursion is first iterated a few times to *create* additional concurrency.

Consider the first order LTI recursion of (3.2). By recasting this recursion, we can express $x(n+2)$ as a function of $x(n)$ to obtain

$$x(n+2) = a\left[ax(n) + bu(n)\right] + bu(n+1). \tag{3.3a}$$

A realization of this recursion is shown in Fig. 3.2(a). The iteration bound of this recursion is $\frac{2(T_m+T_a)}{2}$ and is same as that of Fig. 3.1(a). This is because, the amount of computation and the number of logical delays inside the recursive loop are both doubled as compared to that in Fig. 3.1(a) leading to no net improvement. However, another recursion equivalent to that of (3.3a) is

$$x(n+2) = a^2x(n) + abu(n) + bu(n+1) \tag{3.3b}$$

as shown in Fig. 3.2(b). The iteration period bound of this realization, $\frac{T_m+T_a}{2}$, is a factor of two lower than that of the realizations in Fig. 3.1(a) and Fig. 3.2(a)!

Applying $(M-1)$-steps of look-ahead to the iteration of (3.2), we can obtain an equivalent implementation described by

$$x(n+M) = a^M x(n) + \sum_{i=0}^{M-1} a^i bu(n+M-1-i) \tag{3.3c}$$

and shown in Fig. 3.3(a). Note that the loop delay corresponds to z^{-M} instead of z^{-1}. This implies that the loop computation must be completed after M iteration cycles rather

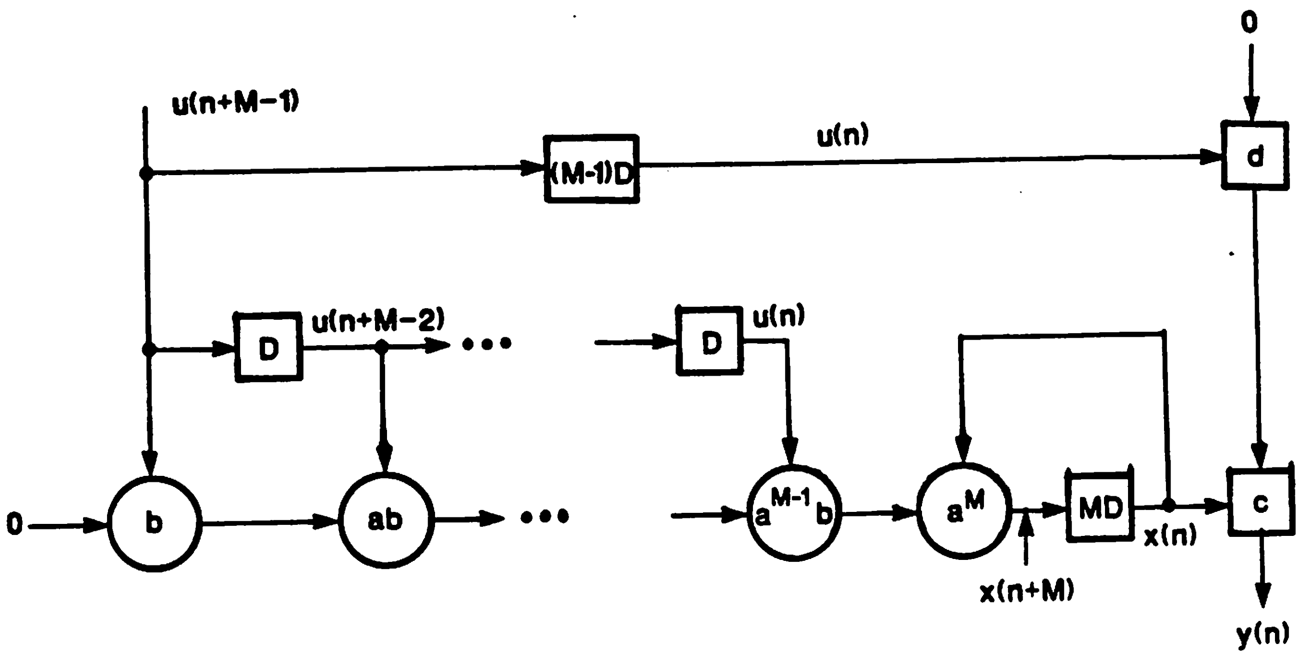
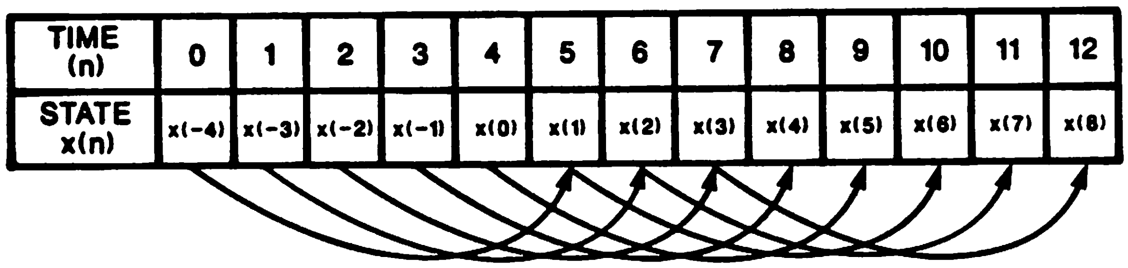


Fig. 3.3(a): Equivalent first order LTI recursion obtained using $(M-1)$ steps of look-ahead.



TIME (n)	0	1	2	3	4	5	6	7	8	9	10	11	12
STATE x(n)	x(−4)	x(−3)	x(−2)	x(−1)	x(0)	x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)

Fig. 3.3(b): A partial schedule for the structure in Fig. 3.3(a) for $M = 5$.

than 1. The iteration period bound of this computation graph is $\frac{(T_m + T_a)}{M}$, which corresponds to a sample rate M times higher than that for the original computation graph (although the complexity and system latency are now linearly increased). A portion of the schedule for the realization of Fig. 3.3(a) is shown in Fig. 3.3(b) for $M = 5$. The terms $ab, a^2b, \ldots, a^{M-1}b, a^M$ in (3.3c) can be precomputed and are referred to as the *precomputation terms*. The second term on the right hand side of (3.3c) represents the look-ahead computation term, and its complexity is referred to as the *look-ahead complexity*. Since the look-ahead computation term is non-recursive, it can be pipelined by placing latches at the appropriate feed-forward cutsets.

The steady state input-output behavior is not altered by the look-ahead technique. By this it is meant that for sufficiently old inputs, the outputs of the transformed system and the original systems will be identical. However, it is also possible to recaste the initial states of the transformed system so that the input-output behavior of the transformed and the original system are identical for *all* inputs, as long as the original system is causal. Consider the schedule shown in Fig. 3.3(b) corresponding to the implementation of Fig. 3.3(a), where we start with M independent initial states $x(-M+1)$, $x(-M+2)$, ..., $x(0)$ (for $M = 5$). In the original system of (3.2), the state $x(1)$ is computed in terms of the initial state $x(0)$,

$$x(1) = ax(0) + bu(0)\,. \tag{3.4a}$$

For the transformed system of (3.3c), the state $x(1)$ is calculated in terms of $x(-M + 1)$,

$$x(1) = a^5x(-4) + bu(0) \tag{3.4b}$$

for $M = 5$ (since $u(-4), \cdots, u(-1)$ are all 0 due to causality). From (3.4a) and (3.4b),

$$x(-4) = a^{-4}x(0)\,. \tag{3.4c}$$

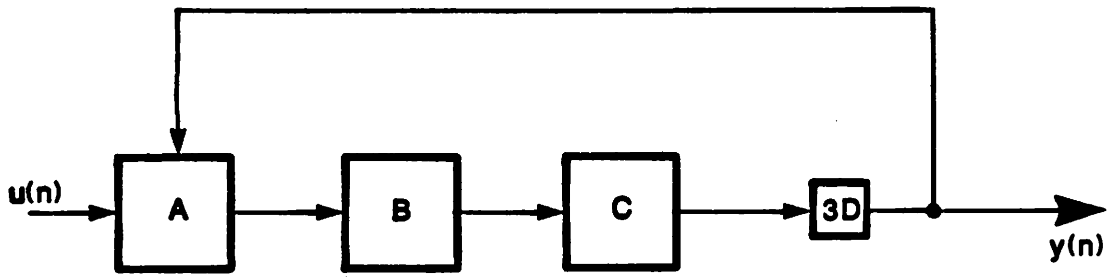


Fig. 3.4(a): A computation graph.

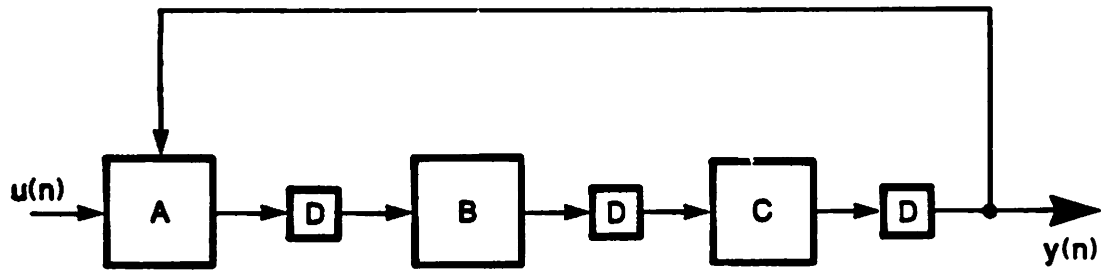


Fig. 3.4(b): Equivalent *retimed* computation graph.

A similar analysis can be carried out to obtain the M initial states

$$x(-i) = a^{-i}x(0), \quad i = 1,2,\ldots,(M-1). \tag{3.5}$$

In the transformed system, we start with M initial states and compute the next M states in a pipelined interleaved manner (see Fig. 3.3(b)). In this regard, look-ahead computation can be treated as an application of pipeline interleaving. Look-ahead computation has allowed us to transform a single serial computation into M independent concurrent computations, and to pipeline the feedback loop to achieve high speed filtering of a single time series while maintaining full hardware utilization.

Provided the multiplier and the adder can be conveniently pipelined, the iteration bound can be achieved by *retiming* or *cutset transformation* [1, 34-36]. The retiming process involves moving the delays around in the feedback loop in such a way that the number of delays in any loop remains unaltered (thereby not affecting the transfer function). A simple example of retiming is illustrated in Fig. 3.4. The iteration period bound for the realization in Fig. 3.4(a) is $\frac{(T_A+T_B+T_C)}{3}$, whereas the actual iteration period is $(T_A+T_B+T_C)$, where T_i corresponds to the computation time of block i. The iteration period for an equivalent realization in Fig. 3.4(b) (obtained after redistributing the delays) is $Max(T_A, T_B, T_C)$. If the computational latencies T_A, T_B, and T_C are identical, then this realization has an iteration period equal to the iteration period bound.

Another example of retiming is illustrated in Fig. 3.5. Fig. 3.5(a) shows a pipelined cellular array multiplier in two's complement arithmetic for a multiplier and multiplicand word-length of three. In a non-recursive implementation, this multiplier can be pipelined as shown in Fig. 3.5(a). This multiplier has a latency of five cycles when pipelined at bit-level. However, when used inside a recursive loop, the multiplier cannot be pipelined. In Fig. 3.5(b), we use look-ahead to create four additional latches. The structure in Fig. 3.5(b) is retimed to obtain an equivalent structure in Fig. 3.5(c). The number of delays in any loop in Fig. 3.5(c) is five. Furthermore, the input-to-output delay in any path is also constant. The serious reader will observe that, in order to perform the retiming operation in a rigorous manner, all inputs in Fig. 3.5(b) should have four extra latches, and all outputs in Fig. 3.5(c) should have the same four extra latches. We have omitted these latches to keep the illustration simple.

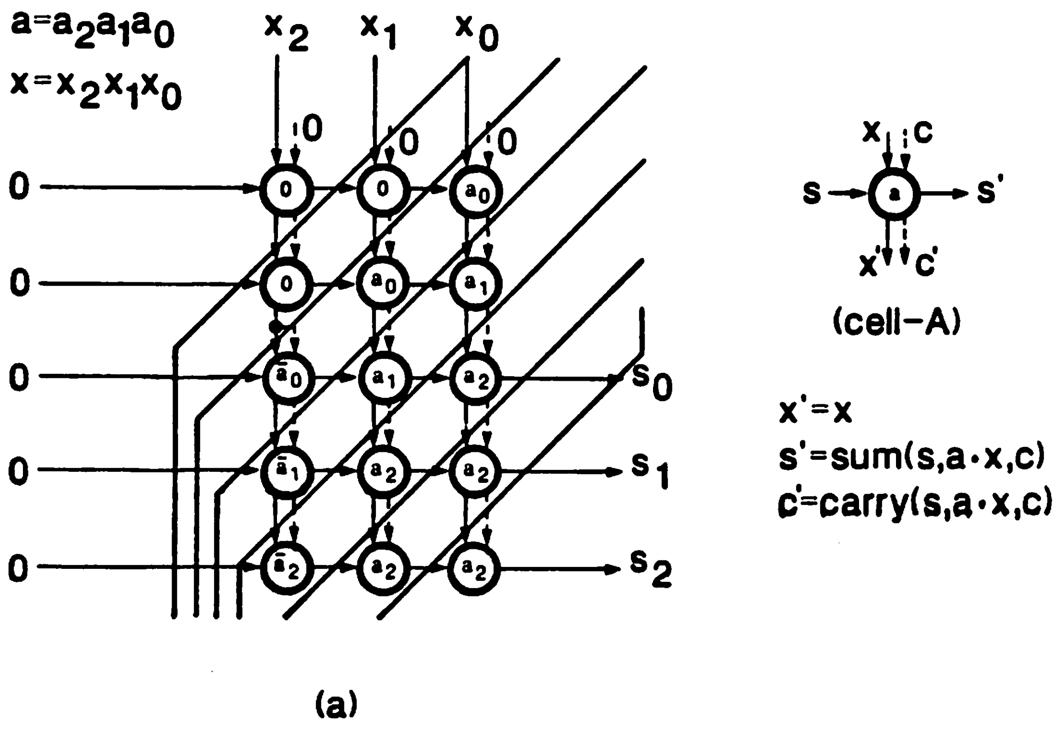


Fig. 3.5(a): A bit-level pipelined array multiplier for word-length of 3.

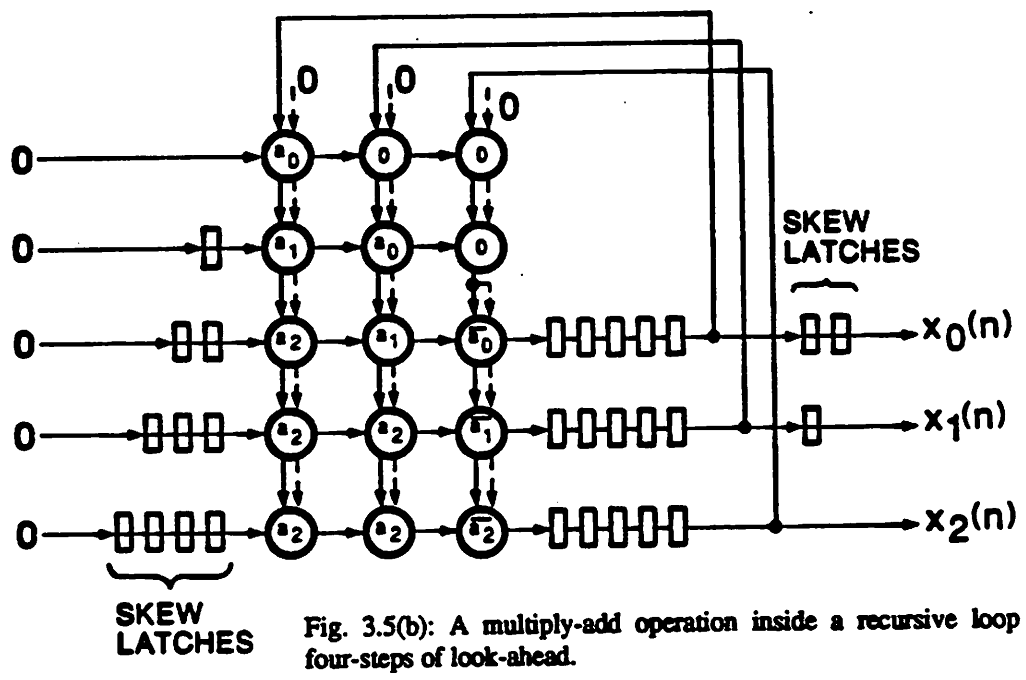


Fig. 3.5(b): A multiply-add operation inside a recursive loop after four-steps of look-ahead.

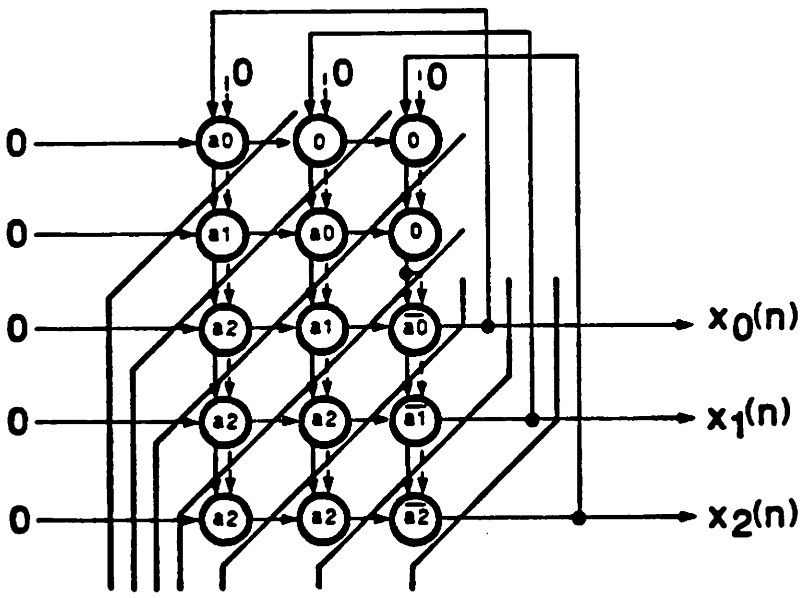


Fig. 3.5(c): Retimed multiply-add structure.

3.3.3. Efficient Multi-Channel Interleaving

We can extend look-ahead to the case where multiple independent channels require identical filtering operation. Consider the same first-order linear recursion of (3.2) for the case of two channels, and six pipeline stages inside the recursive loop. Then, without use of look-ahead, the hardware will be utilized only one third of the time. To get full utilization of hardware, we iterate the recursion two times, and interleave the computation of two time series. In general, if P independent time series are available, and the loop is pipelined by M-stages (assume $M = PQ$), then the recursion needs to be iterated $(Q - 1)$ times. For this example, the iterated recursion corresponds to

$$x^i(n+3) = a^3x^i(n) + a^2bu^i(n) + abu^i(n+1) + bu^i(n+2),\ i = 1, 2 \tag{3.6}$$

Fig. 3.6 shows a partial schedule corresponding to the processing of time series x^1 and x^2 in an interleaved manner.

3.4. PIPELINING DIRECT FORM RECURSIVE FILTERS

The *clustered look-ahead* based pipelining in [18] requires a linear complexity in the number of loop pipeline stages, and does not guarantee stability. In this section, we present a *scattered look-ahead* approach to derive stable pipelined filters of complexity linear with respect to the number of loop pipeline stages. We then introduce a *decomposition technique* to obtain an implementation with logarithmic increase in hardware with respect to the number of loop pipeline stages. The decomposition technique is the key in obtaining area-efficient implementations, and makes pipelined realizations attractive for high speed VLSI IIR filter implementations. We also present fully pipelined and fully hardware efficient linear bidirectional systolic arrays for recursive filters based on scattered look-ahead.

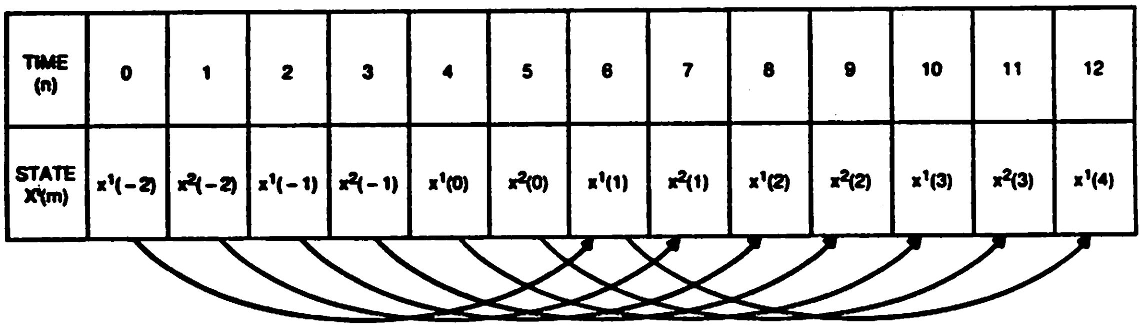
Let the transfer function of a direct form recursive filter be described by

$$H(z) = \frac{\sum_{i=0}^{N} b_i z^{-i}}{1 - \sum_{i=1}^{N} a_i z^{-i}} . \tag{3.7}$$

Equivalently, the output sample $y(n)$ can be described in terms of the input sample $u(n)$, and the past input and output samples, and is given by

$$y(n) = \sum_{i=1}^{N} a_i y(n-i) + \sum_{i=0}^{N} b_i u(n-i) = \sum_{i=1}^{N} a_i y(n-i) + z(n) . \tag{3.8}$$

The sample rate of this recursive filter realization is limited by the throughput of a single multiplication and N additions (since the critical loop contains a single delay operator or latch).



TIME (n)	0	1	2	3	4	5	6	7	8	9	10	11	12
STATE $X^i(m)$	$x^1(-2)$	$x^2(-2)$	$x^1(-1)$	$x^2(-1)$	$x^1(0)$	$x^2(0)$	$x^1(1)$	$x^2(1)$	$x^1(2)$	$x^2(2)$	$x^1(3)$	$x^2(3)$	$x^1(4)$

Fig. 3.6: A partial schedule for a two channel implementation with six loop pipelining stages obtained using two steps of look-ahead.

3.4.1. Clustered Look-Ahead Computation

We can transform the transfer function of (3.7) such that the coefficients of z^{-1}, ..., $z^{-(M-1)}$ in the denominator of the transfer function are zero, i.e. the denominator contains the terms z^{-M}, $z^{-(M+1)}$, ..., and $z^{-(N+M-1)}$. Such a transfer function corresponds to an M-stage pipelined implementation, since the output sample $y(n)$ can be described in terms of the *cluster* of N past outputs $y(n-M)$, $y(n-M-1)$, ..., and $y(n-M-N+1)$. A time domain description of such an equivalent filter is given by (see appendix 3.2)

$$y(n)=\sum_{j=0}^{N-1}\left[\sum_{k=j+1}^{N} a_k\, r_{j+M-k}\right] y(n-j-M)+\sum_{j=0}^{M-1} r_j\, z(n-j) \tag{3.9a}$$

where

$$z(n)=\sum_{i=0}^{N} b_i\, u(n-i)\,, \tag{3.9b}$$

and the sequence r_i is defined in appendix 3.1. The equivalent transfer function of this pipelined realization is given by (see appendix 3.2)

$$H(z)=\frac{\sum_{i=0}^{M-1} r_i \sum_{j=0}^{N} b_j z^{-(i+j)}}{1-\sum_{i=0}^{N-1}\left(\sum_{j=i+1}^{N} a_j r_{i+M-j}\right) z^{-(i+M)}}\,. \tag{3.10}$$

Note that the coefficients in brackets in (3.9) and (3.10) are computed off line. This transfer function has been derived by multiplying $\sum_{i=0}^{M-1} r_i z^{-i}$ both in the numerator and the denominator, introducing $(M-1)$ additional canceling poles and zeros.

Since the critical loop of this implementation contains M delay operators and a single multiplication operation, this loop can be pipelined by M stages, and the sample rate can be increased by a factor of M. The numerator or the non-recursive portion of (3.10) can be implemented with $(N+M)$ multiplications, and the denominator or the recursive portion can be implemented with N multiplications. The total complexity of this pipelined implementation is $(2N+M)$ multiplications, and is linear with respect to the number of loop pipeline stages (M) or speedup or increase in the sample rate.

We illustrate the instability problem in the pipelined recursive filters derived by using the clustered look-ahead approach using an example.

Example 3.1: Consider the example of an all-pole second order IIR filter with poles at $z=\frac{1}{2}$ and $z=\frac{3}{4}$ (see Fig. 3.7(a)). This original filter is described by the transfer function

$$H(z)=\frac{1}{1-\frac{5}{4}z^{-1}+\frac{3}{8}z^{-2}}. \tag{3.11a}$$

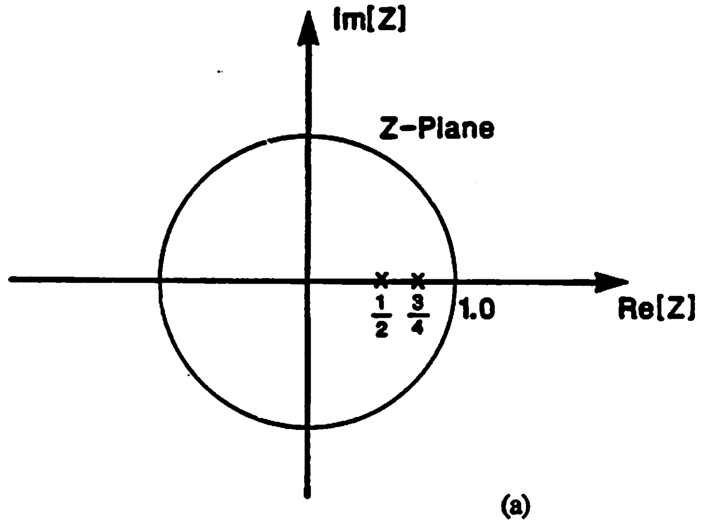
A 2-stage pipelined equivalent recursive digital filter can be derived by multiplying the numerator and denominator by $(1+\frac{5}{4}z^{-1})$, or equivalently by introducing a pole and a zero at $z=-\frac{5}{4}$ (see Fig. 3.7(b)), and is given by

$$H(z)=\frac{1+\frac{5}{4}z^{-1}}{1-\frac{19}{16}z^{-2}+\frac{15}{32}z^{-3}}. \tag{3.11b}$$

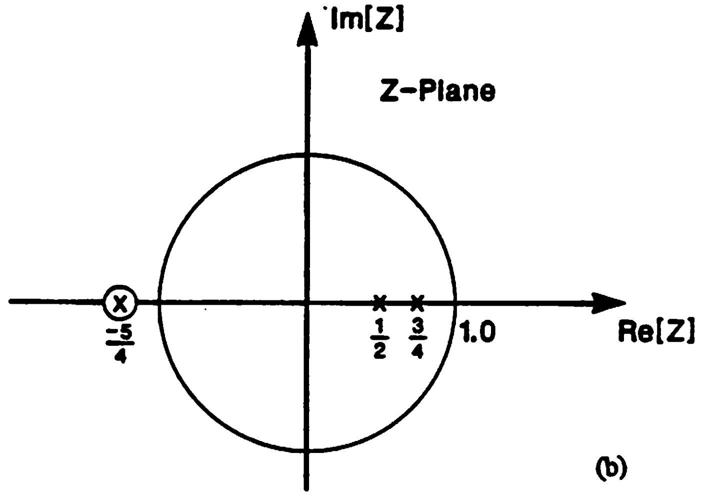
Similarly a 3-stage pipelined realization can be derived by eliminating the z^{-1} and z^{-2} terms in the denominator of (3.11a) and is given by

$$H(z)=\frac{1+\frac{5}{4}z^{-1}+\frac{19}{16}z^{-2}}{1-\frac{65}{64}z^{-3}+\frac{57}{128}z^{-4}}, \tag{3.11c}$$

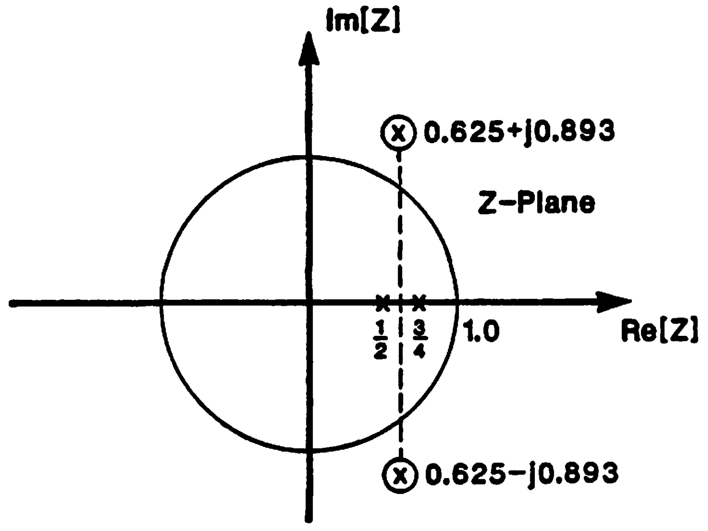
and has poles at $z=0.5, 0.75$ and $z=0.625\pm j0.893$ (see Fig. 3.7(c)). Note that the complex conjugate canceling poles are outside the unity circle. Thus both the 2- and 3-stage equivalent pipelined realizations in (3.11b) and (3.11c) are unstable, even though the original configuration of (3.11a) is stable.



(a)



(b)



(c)

Fig. 3.7: (a) Pole zero representation of a stable second order recursive filter, (b) Pole zero representation of a 2-stage pipelined equivalent unstable filter derived using clustered look-ahead, (c) Pole zero representation of a 3-stage pipelined equivalent unstable filter derived using clustered look-ahead.

3.4.2. Scattered Look-Ahead Without Decomposition

In scattered look-ahead approach [37], the denominator of the transfer function in (3.7) is transformed in a way that it contains the N terms z^{-M}, z^{-2M}, ..., and z^{-NM}. Equivalently, the state $y(n)$ is computed in terms of N past *scattered* states $y(n-M)$, $y(n-2M)$, ..., and $y(n-NM)$. In this look-ahead process, for each pole in the original filter, we introduce $(M-1)$ canceling poles and zeros with equal angular spacing at a distance from the origin same as that of the original pole. For example, if the original filter has a pole at $z=p$, we add $(M-1)$ poles and zeros at $z=pe^{\frac{j2\pi k}{M}}$ for $k=1,2,.....,(M-1)$ to derive a pipelined realization with M loop pipeline stages. The pipelining process using scattered look-ahead approach can be described by

$$H(z)=\frac{N(z)}{D(z)}=\frac{N(z)\prod_{k=1}^{M-1}D(ze^{\frac{j2\pi k}{M}})}{\prod_{k=0}^{M-1}D(ze^{\frac{j2\pi k}{M}})}=\frac{N'(z)}{D'(z^M)}. \tag{3.12}$$

Now we illustrate scattered look-ahead based pipelining in recursive filters using the following examples.

Example 3.2: Consider the first order filter

$$H(z)=\frac{1}{1-az^{-1}}, \tag{3.13a}$$

which has a pole at $z=a$. A 3-stage pipelined equivalent stable filter can be derived by adding poles and zeros at $z=ae^{\frac{j2\pi}{3}}$ and $z=ae^{-\frac{j2\pi}{3}}$, and is given by

$$H(z)=\frac{1+az^{-1}+a^2z^{-2}}{1-a^3z^{-3}}. \tag{3.13b}$$

Example 3.3: Consider the second order filter transfer function

$$H(z)=\frac{1}{1-a_1z^{-1}-a_2z^{-2}}. \tag{3.14a}$$

A 3-stage equivalent pipelined filter is given by

$$H(z)=\frac{1+a_1z^{-1}+(a_1^2+a_2)z^{-2}-a_1a_2z^{-3}+a_2^2z^{-4}}{1-(a_1^3+3a_1a_2)z^{-3}-a_2^3z^{-6}}. \tag{3.14b}$$

Example 3.4: Consider the second order filter with complex conjugate poles at $z=re^{\pm j\theta}$. The transfer function of the filter is given by

$$H(z)=\frac{1}{1-2r\cos\theta z^{-1}+r^2z^{-2}}. \tag{3.15a}$$

We can pipeline this filter by three stages by introducing four additional poles and zeros at $z=re^{\pm j(\theta+\frac{2\pi}{3})}$, $z=re^{\pm j(\theta-\frac{2\pi}{3})}$. The equivalent pipelined filter is given by

$$H(z)=\frac{1+2r\cos\theta z^{-1}+(1+2\cos 2\theta)r^2z^{-2}+2r^3\cos\theta z^{-3}+r^4z^{-4}}{1-2r^3\cos 3\theta z^{-3}+r^6z^{-6}}. \tag{3.15b}$$

Example 3.5: Consider the second order filter with real poles at $z=r_1$ and $z=r_2$. The transfer function is given by

$$H(z)=\frac{1}{1-(r_1+r_2)z^{-1}+r_1r_2z^{-2}}. \tag{3.16a}$$

A 3 stage pipelined realization is derived by adding poles (and zeros) at $z=r_1e^{\pm\frac{j2\pi}{3}}$, $z=r_2e^{\pm\frac{j2\pi}{3}}$. The pipelined realization is given by

$$H(z)=\frac{1+(r_1+r_2)z^{-1}+(r_1^2+r_1r_2+r_2^2)z^{-2}+r_1r_2(r_1+r_2)z^{-3}+r_1^2r_2^2z^{-4}}{1-(r_1^3+r_2^3)z^{-3}+r_1^3r_2^3z^{-6}}. \tag{3.16b}$$

The scattered look-ahead approach leads to stable pipelined filters if the original filter is stable, since the distance of the canceling poles from the origin is same as that of the original pole. The complexity of the non-recursive portion in (3.12) is $(NM+1)$, and of the recursive portion is N, leading to a total complexity $(NM+N+1)$ pipelined multiplications, a linear complexity with respect to M. Even though this complexity is linear with respect to M, it is much greater than that of clustered look-ahead.

The scattered look-ahead algorithm is different from the *recursive doubling* algorithm, developed by Kogge and Stone [38-40], used for parallel implementation of higher order linear recurrence systems [41-43] (they are identical for the first order system). Although the recursive doubling algorithm leads to a logarithmic complexity, for higher order systems the coefficient of the logarithmic complexity in [40] is much greater than ours. Indeed, the scattered look-ahead algorithm is similar to the *cyclic reduction* algorithm discovered by Hockney [44] and used in the context of parallel solution of partial differential equations [45-49]. The scattered look-ahead approach has also been discussed in [50] in the context of zero-input recursive systems (not a filtering operation). The denominator of the pipelined filter transfer function also has the same form as in each phase of a polyphase network [51]. The pipelining of the recursive filters using scattered look-ahead algorithm and the canceling pole-zero interpretation was first discovered in [37].

We now derive another pipelined realization using a *decomposition technique* which leads to a logarithmic increase in hardware with respect to speedup or increase in the sampling rate.

3.4.3. Scattered Look-Ahead with Power of Two Decomposition

In this decomposed implementation, the output sample $y(n)$ is computed using N past scattered output samples $y(n-M), y(n-2M), \cdots, y(n-NM)$ and the numerator (or the non-recursive portion) is implemented in a decomposed or factored form (for cases where M can be expressed as a power of 2) [22-23]. The use of this technique leads to a logarithmic increase in hardware with respect to M.

Let the recursive portion of a digital filter with K pipeline latches inside the critical loop be described by

$$H(z) = \frac{1}{1 - \sum_{i=1}^{N} q_i(K) z^{-iK}} . \tag{3.17}$$

The original transfer function corresponds to a single stage pipelined implementation for $K = 1$, and hence $q_i(1) = a_i$. We can derive an equivalent $2K$-stage pipelined implementation by multiplying by $(1 - \sum_{i=1}^{N} (-1)^i q_i(K) z^{-iK})$ in the numerator and denominator. The equivalent $2K$-stage pipelined implementation is described by

$$H(z) = \frac{1 - \sum_{i=1}^{N} (-1)^i q_i(K) z^{-iK}}{(1 - \sum_{i=1}^{N} q_i(K) z^{-iK})(1 - \sum_{i=1}^{N} (-1)^i q_i(K) z^{-iK})} = \frac{1 - \sum_{i=1}^{N} (-1)^i q_i(K) z^{-iK}}{1 - \sum_{i=1}^{N} q_i(2K) z^{-2iK}} \tag{3.18}$$

where the sequence $q_i(2K)$ is derived in terms of the sequence $q_i(K)$ in appendix 3.3.

We can apply this transformation to the original single stage pipelined transfer function to obtain a two stage pipelined implementation, and subsequent transformations lead to four, eight, and sixteen stage pipelined implementations respectively. Thus to obtain an M-stage pipelined implementation, we need to apply $\log_2 M$ sets of such transformations. Each transformation leads to an increase in multiplication complexity by N while increasing the speed (or sample rate) or the number of pipeline stages inside the critical recursive loop by a factor 2. A series of such transformations then lead to a geometric increase in the number of loop pipeline stages or speed while requiring only an arithmetic increase in hardware complexity!

We apply $(log_2 M - 1)$ sets of such transformations to derive an equivalent transfer function (with M pipelining stages inside the recursive loop), which is described by

$$H(z) = \frac{(\sum_{i=0}^{N} b_i z^{-i}) \prod_{k=0}^{\log_2 M-1} (1 - \sum_{i=1}^{N} (-1)^i q_i(2^k) z^{-i2^k})}{1 - \sum_{i=1}^{N} q_i(M) z^{-iM}} \tag{3.19}$$

and requires a complexity of $(2N + N\log_2 M + 1)$ multiplications, a logarithmic complexity with respect to speedup or M. Note that although the number of multiply operations is logarithmic, the number of delays or latches is linear. The total number of latches in the implementation is approximately $NM(log_2 M + 1)$, out of which about NM delays are required for implementation of the non-recursive portions, and about $NM\log_2 M$ delays are required to pipeline each of the $N\log_2 M$ multiplications by M stages. This implementation has been derived by incorporating $N(M-1)$ canceling poles and zeros. In the decomposed realization, the first stage implements an N-th order non-recursive section, and the subsequent stages respectively implement $2N$, $4N$, ..., $\frac{NM}{2}$ order non-recursive sections. Due to the symmetry of coefficients, each of these non-recursive sections can be implemented with N multiplications independent of the order of that section! An alternative treatment of this decomposition algorithm is given in [52]. Now we consider examples to illustrate scattered look-ahead and decomposition based pipelining in recursive filters.

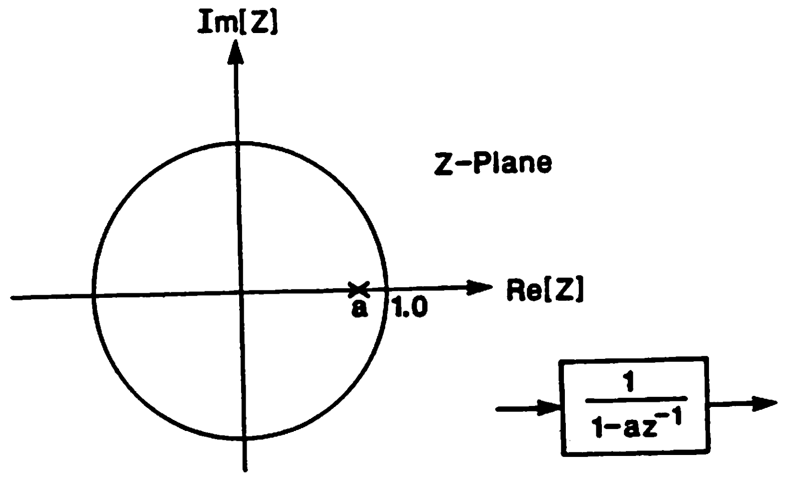


Fig. 3.8(a): A single pole filter.

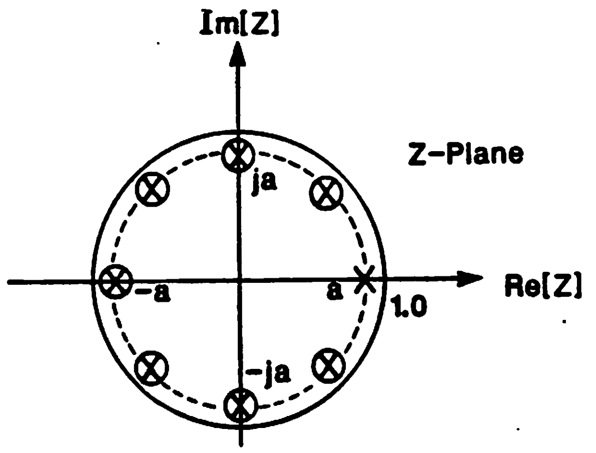


Fig. 3.8(b): Pole zero representation of an 8-stage pipelined single pole filter.

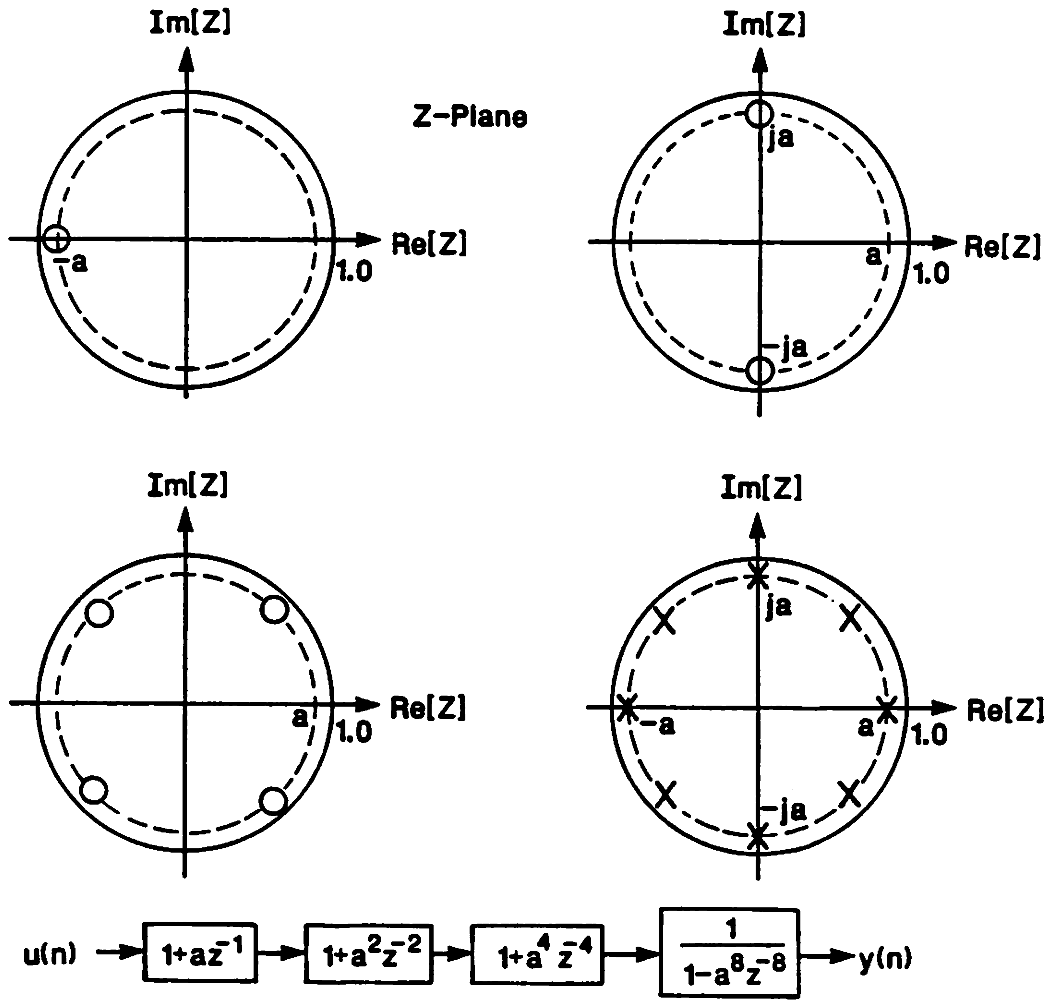


Fig. 3.8(c): Decomposition based pipelined implementation

Example 3.6: First-Order Section:

Consider a first-order recursive filter transfer function described by

$$H(z) = \frac{bz^{-1}}{1 - az^{-1}} . \tag{3.20a}$$

For this transfer function,

$$q(1) = a, \; q(2K) = q^2(K) = a^{2K} . \tag{3.20b}$$

The equivalent pipelined transfer function can be derived using the decomposition technique, and is described by

$$H(z) = \frac{bz^{-1} \prod_{i=0}^{\log_2 M - 1} (1 + q(2^i) z^{-2^i})}{1 - q(M) z^{-M}} = \frac{bz^{-1} \prod_{i=0}^{\log_2 M - 1} (1 + a^{2^i} z^{-2^i})}{1 - a^M z^{-M}} . \tag{3.20c}$$

This pipelined implementation has been derived by adding $(M-1)$ poles and zeros at identical locations. The original transfer function has a single pole at $z = a$ (see Fig. 3.8(a)). The pipelined transfer function has poles at locations a, $ae^{j\frac{2\pi}{M}}$, $ae^{j\frac{2(2\pi)}{M}}$, $ae^{j\frac{3(2\pi)}{M}}$, ..., $ae^{j\frac{(M-1)(2\pi)}{M}}$ (see Fig. 3.8(b) for $M = 8$). The decomposition of canceling zeros in the pipelined transfer function is shown in Fig. 3.8(c). The i-th stage of the non-recursive portion implements 2^i zeros located at

$$z = ae^{j\frac{(2n+1)\pi}{2^i}}, n = 0,1, \cdots ,(2^i - 1) \tag{3.20d}$$

and requires a single pipelined multiplication operation (independent of the stage number i). The total complexity of the pipelined implementation is $(log_2 M+2)$ multiplications.

The decomposition based pipelined implementation can also be equivalently explained using the time domain approach. The original recursive filter description is given by

$$y(n+1) = ay(n) + bu(n) \tag{3.21a}$$

and the pipelined realization is given by

$$y(n+M) = a^M y(n) + \sum_{i=0}^{M-1} a^i bu(n+M-1-i)\,. \tag{3.21b}$$

As an example, for M=8, we have

$$\begin{aligned} y(n+8) &= a^8 y(n) + \sum_{i=0}^{7} a^i bu(n+7-i) \qquad (3.21c)\\ &= a^8 y(n) + \sum_{i=0}^{7} a^i f_0(n+7-i), \text{ where } f_0(n) = bu(n)\\ &= a^8 y(n) + \sum_{i=0}^{3} a^{2i} f_1(n+7-2i), \text{ where } f_1(n) = af_0(n-1) + f_0(n)\\ &= a^8 y(n) + \sum_{i=0}^{1} a^{4i} f_2(n+7-4i), \text{ where } f_2(n) = a^2 f_1(n-2) + f_1(n)\,. \end{aligned}$$

A block diagram of an 8-stage pipelined decomposed implementation is shown in Fig. 3.8(d).

Although the pipelined recursive filter realizations are stable under infinite precision conditions, they are sensitive to filter coefficients under finite precision. In a finite precision implementation, the poles of the first order M-stage pipelined filter are located at

$$p = (a^M + \Delta)^{\frac{1}{M}} = a\ (1 + \frac{\Delta}{Ma^M})\,,$$

where Δ corresponds to the finite precision error in representing a^M. This pole location is more sensitive for smaller values of a (that is when poles are closer to the origin). Fortunately this is not a problem, since the instability problem for the filter with poles closer to origin is not severe.

In addition to the instability problem, finite precision pipelined filters suffer from inexact pole zero cancelation (see Fig. 3.8(e)), which leads to magnitude and phase error. These errors can be reduced by increasing word-length, but a thorough analysis of this is beyond the scope of this thesis.

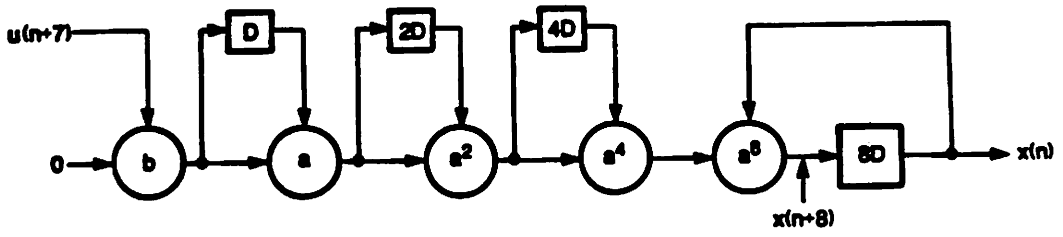


Fig. 3.8(d): Time domain decomposition of the pipelined filter.

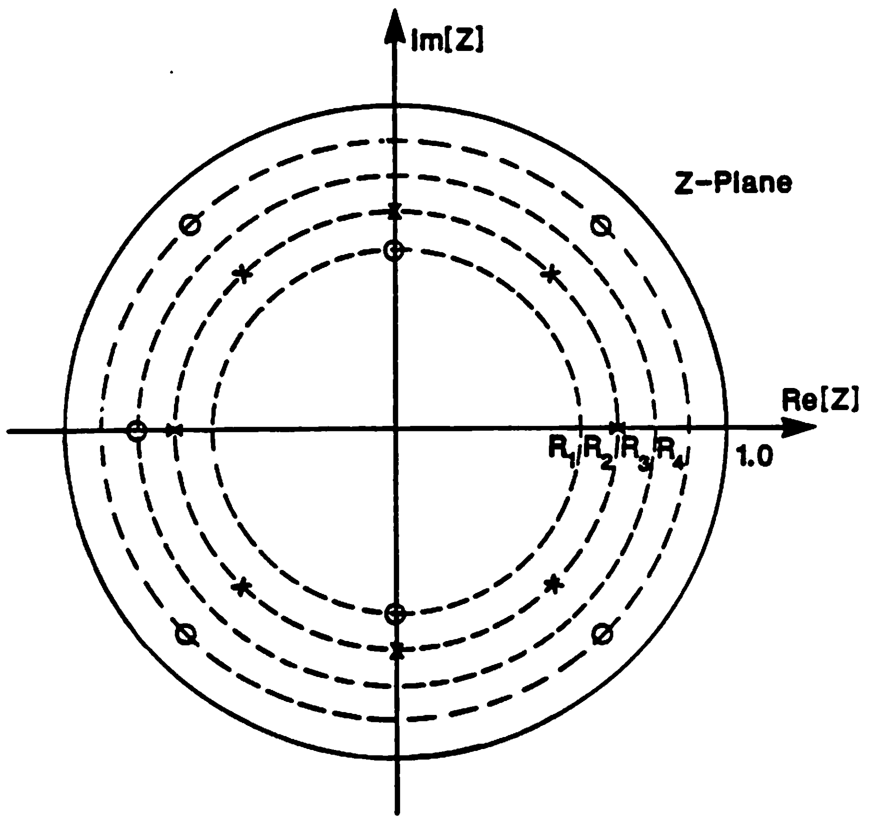


Fig. 3.8(e): Inexact pole zero cancelation in a finite word-length 8-stage pipelined first order filter.

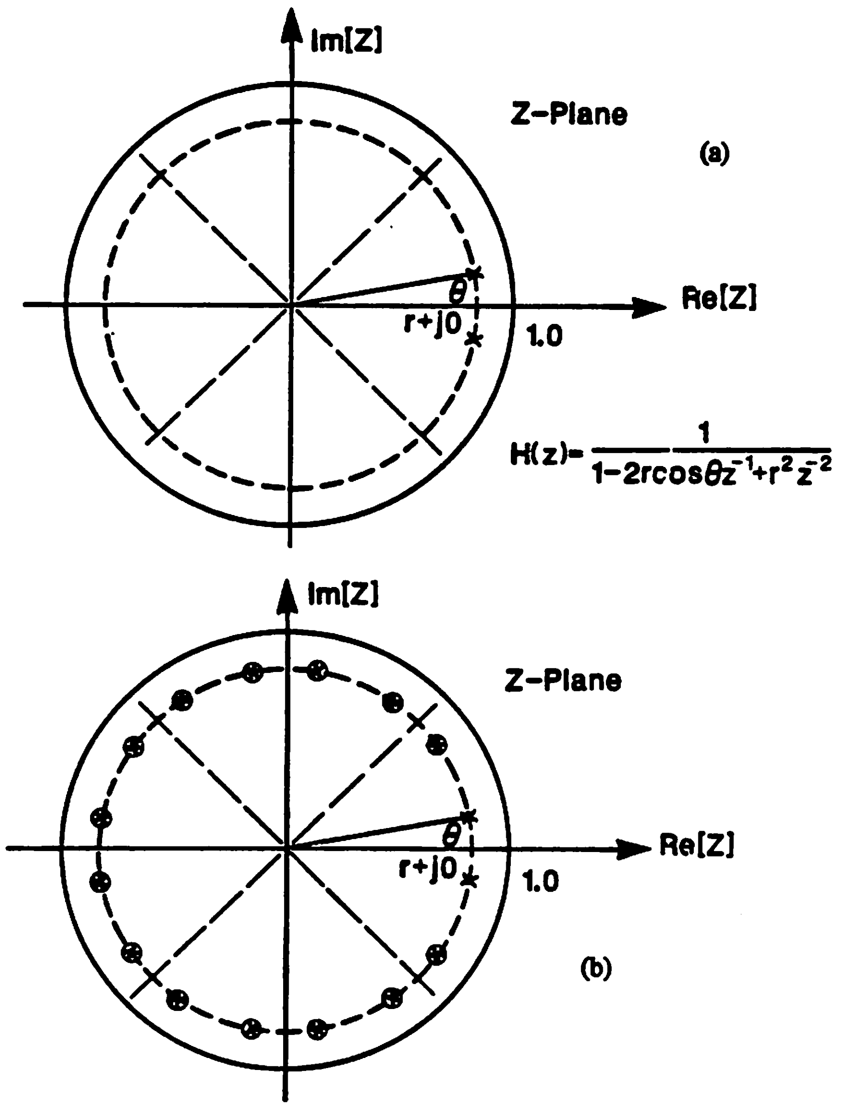


Fig. 3.9: (a) Pole zero diagram of the second order filter, (b) Pole zero representation of the pipelined second order direct form filter with 8 loop pipelining stages.

Example 3.7: Second Order System:

Consider a second order recursive filter described by

$$H(z)=\frac{Y(z)}{U(z)}=\frac{b_0+b_1z^{-1}+b_2z^{-2}}{1-2r\cos\theta z^{-1}+r^2z^{-2}}\,. \tag{3.22a}$$

The poles of the system are located at $re^{+j\theta}$ and $re^{-j\theta}$ (see Fig. 3.9(a)). For this filter,

$$q_1(1)=2r\cos\theta,\ q_2(1)=-r^2, \tag{3.22b}$$

and

$$q_1(2K)=q_1^2(K)+2q_2(K)=2r^{2K}\cos 2K\theta\ ,q_2(2K)=-q_2^2(K)=-r^{4K}\ . \tag{3.22c}$$

The pipelined transfer function is described by

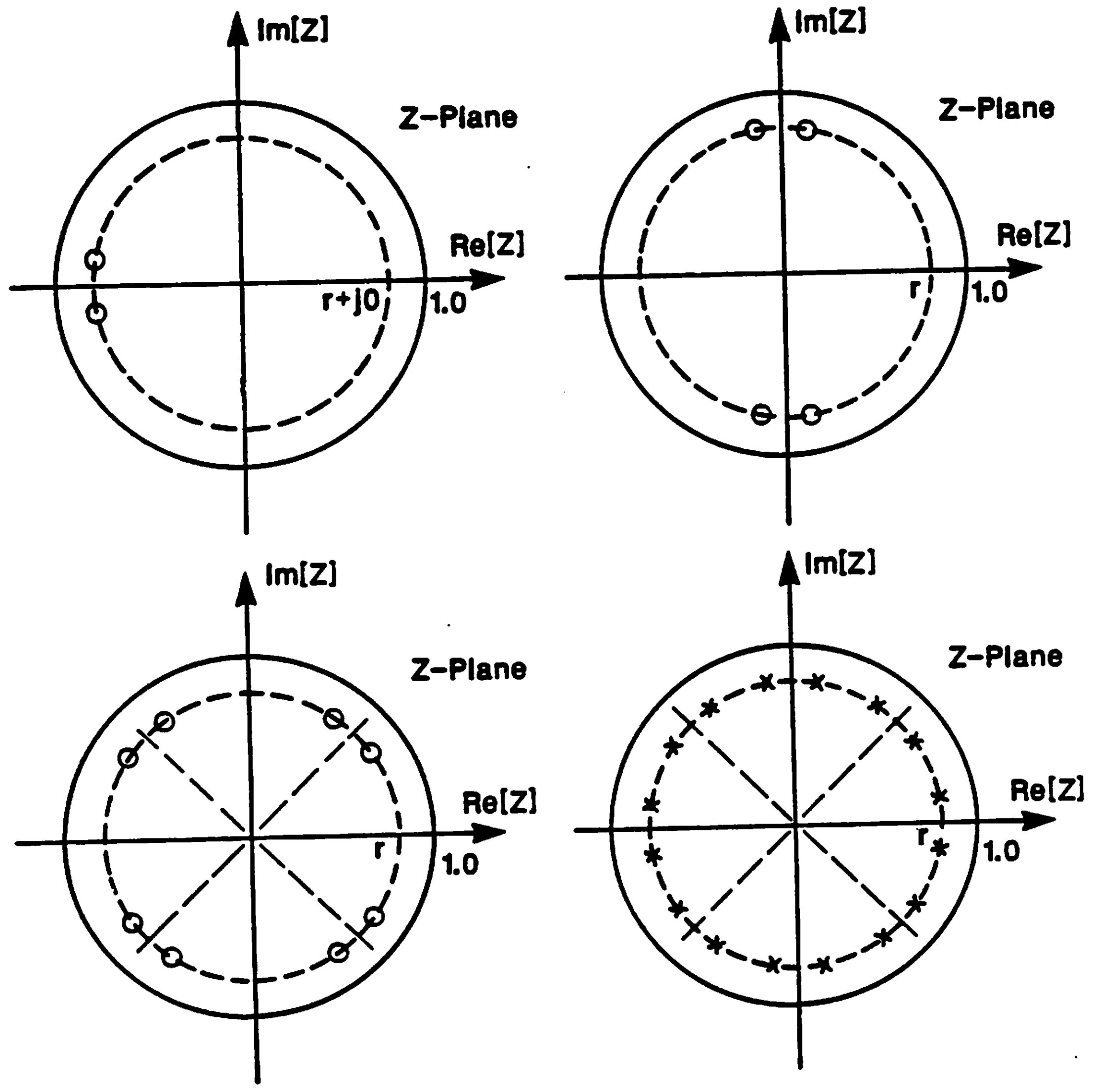
$$H(z)=\frac{(\sum_{i=0}^{2}b_iz^{-i})\prod_{i=0}^{\log_2 M-1}(1+q_1(2^i)z^{-2^i}-q_2(2^i)z^{-2^{i+1}})}{1-q_1(M)z^{-M}-q_2(M)z^{-2M}} \tag{3.23}$$

$$=\frac{(\sum_{i=0}^{2}b_iz^{-i})\prod_{i=0}^{\log_2 M-1}(1+2r^{2^i}\cos 2^i\theta z^{-2^i}+r^{2^{i+1}}z^{-2^{i+1}})}{1-2r^M\cos M\theta z^{-M}+r^{2M}z^{-2M}}\,.$$

The $2M$ poles of the transformed transfer function are located at

$$z=re^{\pm j(\theta+i\frac{2\pi}{M})},\ i=0,1,2,\ldots,(M-1)$$

and are shown in Fig. 3.9(b). The decomposed implementation of the pipelined filter is shown in Fig. 3.9(c) and 9(d). The pipelined filter can be implemented with an implementation complexity of ($2\log_2 M$+5) multiplications. The quantization error due to the recursive portion of a pipelined second order section is studied in appendix 3.4, and it is shown that the upper bound on the quantization error in the pipelined filter decreases with increase in the number of loop pipeline stages. Intuitively this should be expected, since as M increases, the IIR filter closely approximates an FIR filter, for which the quantization error is inherently less.



$$H(z) = \frac{(1+2r\cos\theta z^{-1}+r^2z^{-2})\,(1+2r^2\cos2\theta z^{-2}+r^4z^{-4})\,(1+2r^4\cos4\theta z^{-4}+r^8z^{-8})}{1-2r^8\cos8\theta z^{-8}+r^{16}z^{-16}}$$

Fig. 3.9(c): Decomposition of poles and zeros of the 8-stage pipelined second order filter.

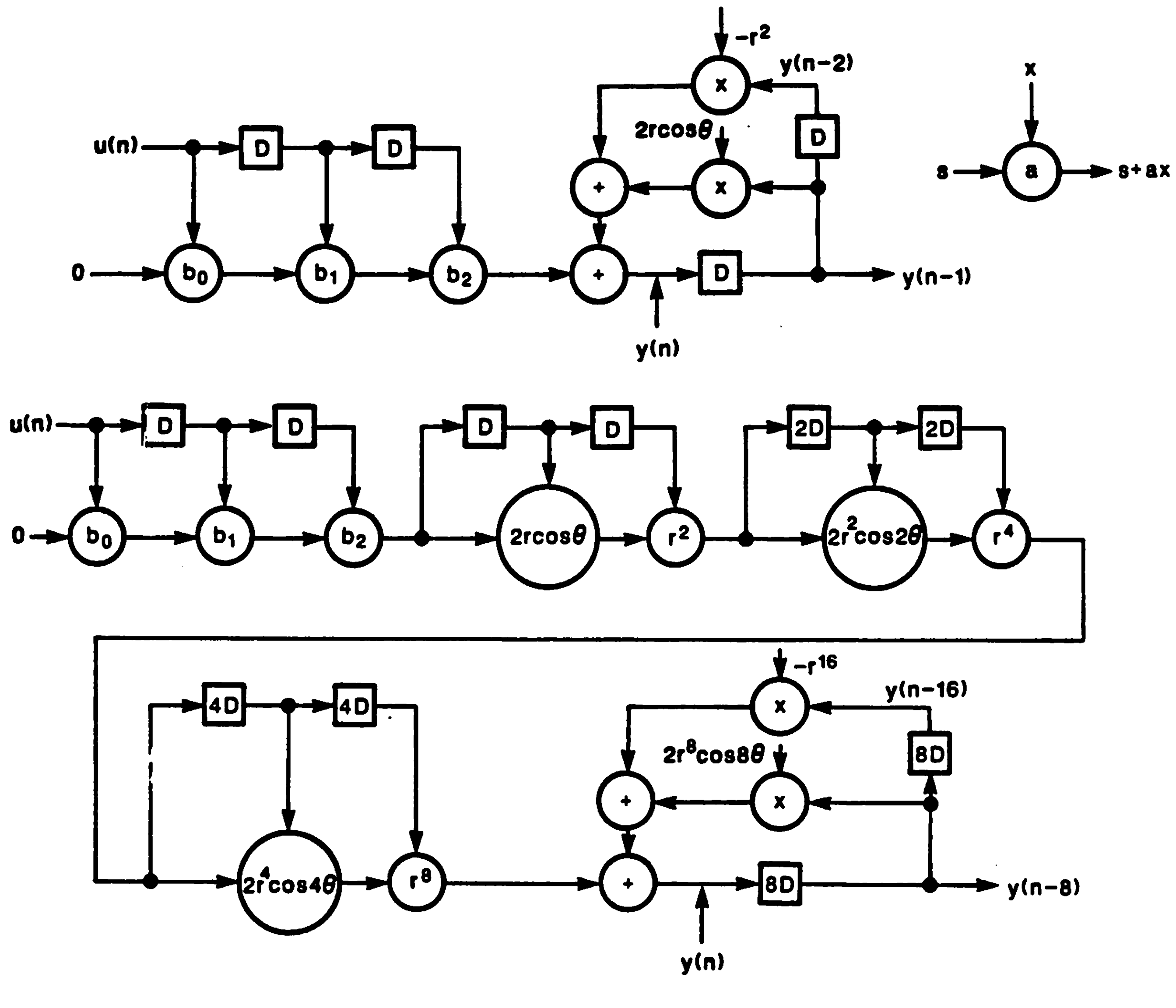


Fig. 3.9(d): Implementation of the original and the pipelined second order recursive digital filter using decomposition technique for 8 pipelining stages inside the recursive loop.

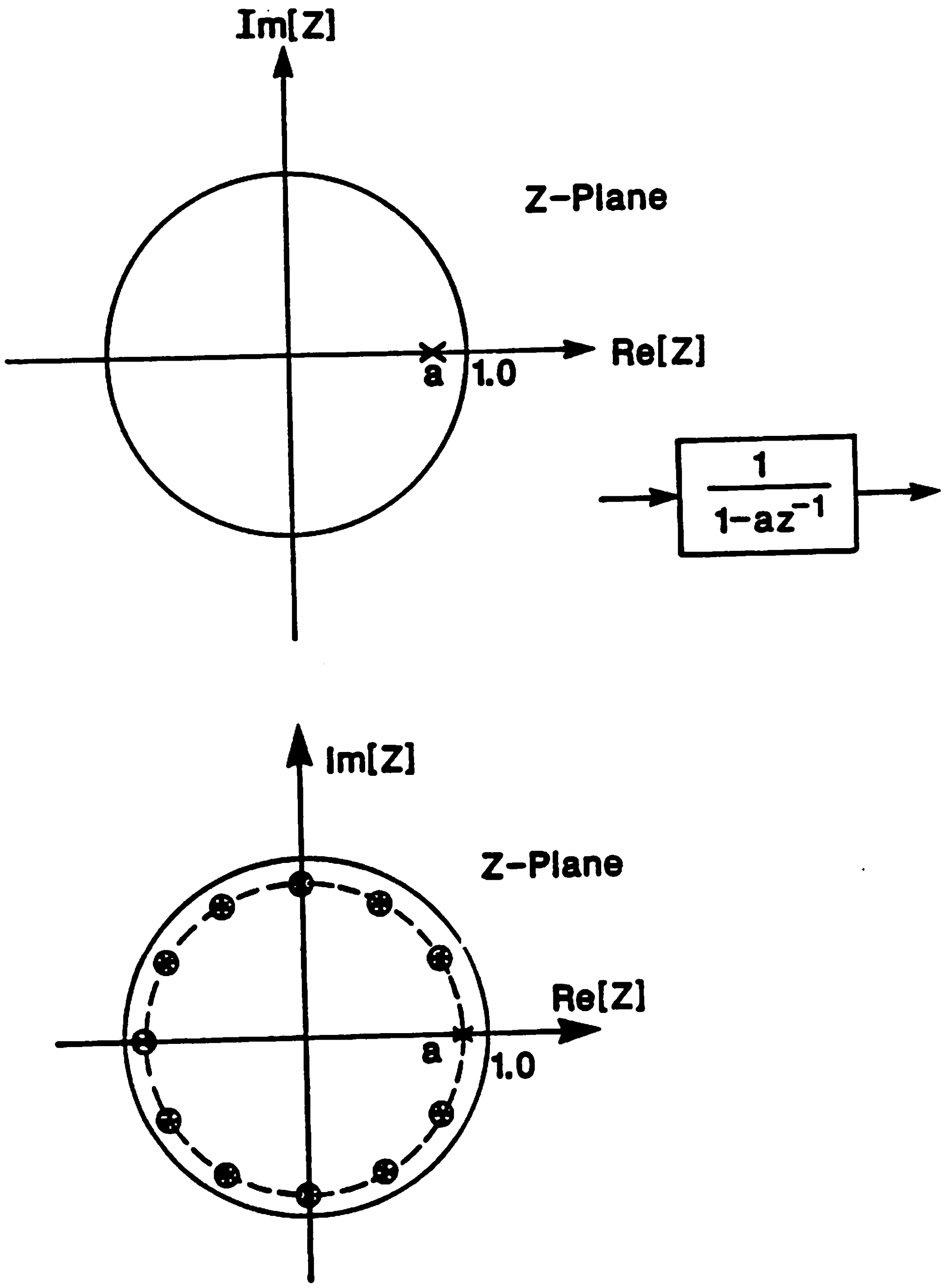


Fig. 3.10(a): Pole zero representation of a 12-stage pipelined first-order filter.

A single-chip implementation of a fourth-order recursive digital filter (organized as two cascaded second order sections) using four stages of loop pipelining and running at hundred million samples per second rate has been reported in [53]. This chip uses the scattered look-ahead and the decomposition algorithms developed in this thesis. The chip is implemented in 0.9 micron double-layer metal CMOS technology by the AT&T Bell Laboratories. It uses a silicon area of $14mm^2$, and has a transistor density of 0.6 million devices per cm^2. The total computing power of the chip is 1.7 billion multiply operations per second. The reader is referred to [53] for details of the integrated circuit chip implementation aspects.

3.4.4. Scattered Look-Ahead With General Decomposition

We have so far concentrated on power-of-two decompositions only, which leads to hardware-efficient implementations. However, the decomposition of canceling zeros extends for any arbitrary number of loop pipeline stages. The time-domain interpretation of simple M_1M_2 decomposition was studied in [54]. We now illustrate decomposition of canceling zeros for arbitrary M.

In an N-th order filter with M-levels of pipelining, there are $N(M-1)$ canceling zeros. First consider the simple case of $M = M_1M_2$ decomposition [54]. In this implementation, the system has $N(M_1M_2-1)$ canceling zeros. The first stage implements $N(M_1-1)$ zeros, and the second stage implements $NM_1(M_2-1)$ zeros. In a $M_1M_2M_3$ decomposition, the first stage implements $N(M_1-1)$ zeros, the second stage implements $NM_1(M_2-1)$ zeros, and the third stage implements $NM_1M_2(M_3-1)$ zeros. In general, in a $M = M_1M_2 \ldots M_P$ decomposition, the P non-recursive stages respectively implement

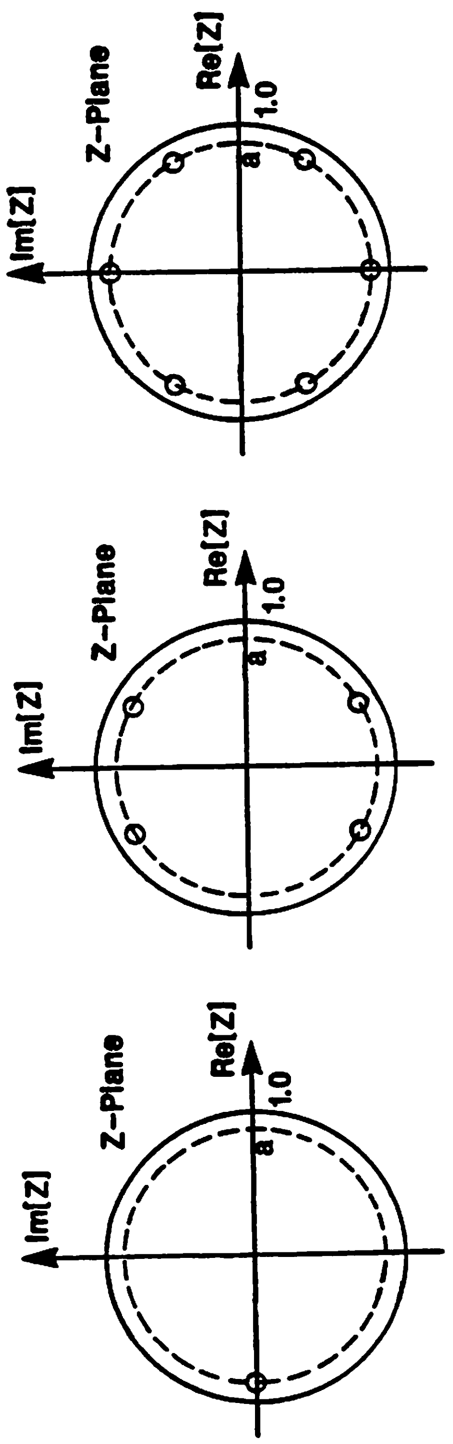


Fig. 3.10(b): 2×3×2 decomposition of the zeros.

$N(M_1-1)$, $NM_1(M_2-1)$, ..., $NM_1M_2 \cdots M_{P-1}(M_P-1)$ zeros, totaling $N(M-1)$ zeros. The non-recursive portion of the general decomposition requires about NM delays and $N\sum_{i=1}^{P}(M_i-1)$ multipliers (each of these multipliers also requires M latches for pipelining).

Example 3.8: Consider the first order transfer function in (3.20a). A 12-stage pipelined decomposed implementation is given by

$$H(z)=\frac{\sum_{i=0}^{11} a^i z^{-i}}{(1-a^{12}z^{-12})}=\frac{(1+az^{-1})(1+a^2z^{-2}+a^4z^{-4})(1+a^6z^{-6})}{(1-a^{12}z^{-12})}. \qquad (3.24a)$$

The above implementation corresponds to a 2×3×2 decomposition. The pole-zero configuration of the 12 stage pipelined filter is shown in Fig. 3.10(a). The decomposition of 11 canceling zeros of this filter is shown in Fig. 3.10(b), where the three sections respectively implement 1, 4, and 6 zeros respectively. Here the first section implements the zero at $-a$, the second section implements four zeros at $ae^{\pm\frac{j\pi}{3}}$ and $ae^{\pm\frac{j2\pi}{3}}$, and the third section implements six zeros at $\pm ja$, $ae^{\pm\frac{j\pi}{6}}$, and $ae^{\pm\frac{j5\pi}{6}}$. Another decomposed transfer function given by

$$H(z)=\frac{(1+az^{-1})(1+a^2z^{-2})(1+a^4z^{-4}+a^8z^{-8})}{(1-a^{12}z^{-12})} \qquad (3.24b)$$

corresponds to 2×2×3 decomposition. In this implementation, the first non-recursive section implements one zero at $-a$, the second section implements two zeros at $\pm ja$, and the third section implements eight zeros at $ae^{\pm\frac{j\pi}{6}}$, $ae^{\pm\frac{j\pi}{3}}$, $ae^{\pm\frac{j2\pi}{3}}$, and $ae^{\pm\frac{j5\pi}{6}}$. The 3×2×2 decomposition is given by

$$H(z)=\frac{(1+az^{-1}+a^2z^{-2})(1+a^3z^{-3})(1+a^6z^{-6})}{(1-a^{12}z^{-12})}, \qquad (3.24c)$$

and the three sections respectively implement 2, 3, and 6 zeros. The first section implements two zeros at $ae^{\pm\frac{j2\pi}{3}}$, the second implements three zeros at $-a$ and $ae^{\pm\frac{j\pi}{3}}$, and the third section implements six zeros at $ae^{\pm\frac{j\pi}{6}}$, $\pm ja$, and $ae^{\pm\frac{j5\pi}{6}}$.

Any higher order recursive filter can be factored in terms of first order sections. Decomposition similar to the above example can be applied to the first order sections, and then the complex conjugate sections can be combined to obtain the decomposed form in terms of real multiplications. A matrix interpretation of the above transfer function decomposition has been studied in [52]. The ordering of decomposition factors can be exploited to minimize roundoff errors.

3.4.5. FIR *vs* IIR Filters

We can start with frequency domain specifications of a digital filter, and implement the filter as an FIR or an IIR filter. Let the order of an FIR filter be N_{FIR} and the order of an IIR filter to satisfy the same requirement be N_{IIR}. For the same speed (or equivalently, for same level of pipelining), the complexity of the FIR filter in terms of M-stage pipelined multipliers is N_{FIR}, and that for the IIR filter is $(2N_{IIR} + N_{IIR}\log_2 M + 1)$. Hence, the IIR filter realization is preferable if

$$2N_{IIR} + N_{IIR}\log_2 M + 1 < N_{FIR}\ , \tag{3.25a}$$

or equivalently, if

$$M < 2^{\lfloor \frac{N_{FIR}-1}{N_{IIR}} - 2 \rfloor}\ , \tag{3.25b}$$

where $\lfloor x \rfloor$ represents the floor function of x. As an example, in Rabiner *et al* [55], it is shown that a filter spectrum can be implemented as a 6-th order IIR filter or as a 41-st order FIR filter. Then for this filter, M must be less than 16 for the IIR filter to be hardware-efficient as compared with its FIR counterpart.

3.4.6. Linear Bidirectional Systolic Array Architectures

All bidirectional systolic array implementations of pipelined recursive digital filters presented so far require many-way interleaving [1, 25-29]. In this section, we derive linear bidirectional pipelined systolic arrays for direct form recursive digital filters using the scattered look-ahead algorithm. These arrays are highly concurrent, fully pipelined, and do not require any interleaving of input samples. Since the non-recursive portion can be implemented with arbitrary level of pipelining, we restrict our attention to only the recursive portion.

Consider the recursive algorithm described by

$$y(n) = \sum_{i=1}^{N} q_i(M) y(n-iM) + x(n) \tag{3.26}$$

where $x(n)$ corresponds to the output of the non-recursive portion. This algorithm corresponds to an M stage pipelined implementation. A flow graph corresponding to the above algorithm is shown in the Fig. 3.11(a). For $M = 1$, the bidirectional array cannot be fully pipelined without requiring interleaving. However, a pipeline interleaved version can be achieved, which is useful for applications requiring moderate amount of concurrency, and where multiple independent time series need to be filtered similarly in an interleaved manner.

For $M \geq 2$, a fully pipelined systolic array can be implemented. In this implementation, all the processing elements operate in a pipelined manner, and the operations inside each processing element can also be deeply pipelined. The M delays or latches can be moved around the loop to pipeline inter-stage operations as well as the multiplication/addition operation (intra-stage pipeline). Out of M delays, 2 delays are used for inter-stage pipelining, and the $(M-2)$ delays are used to pipeline the

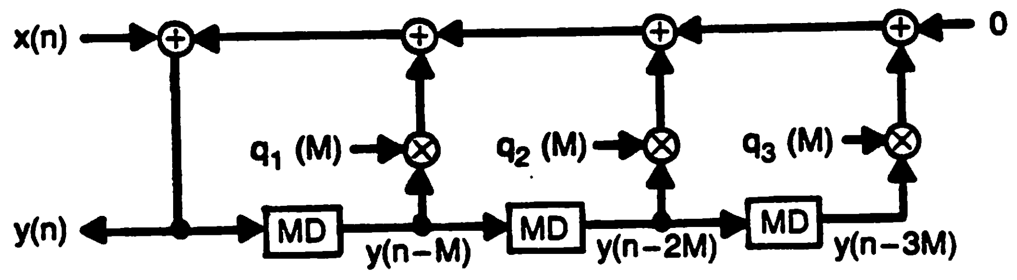


Fig. 3.11(a): Linear systolic implementation of a recursive filter.

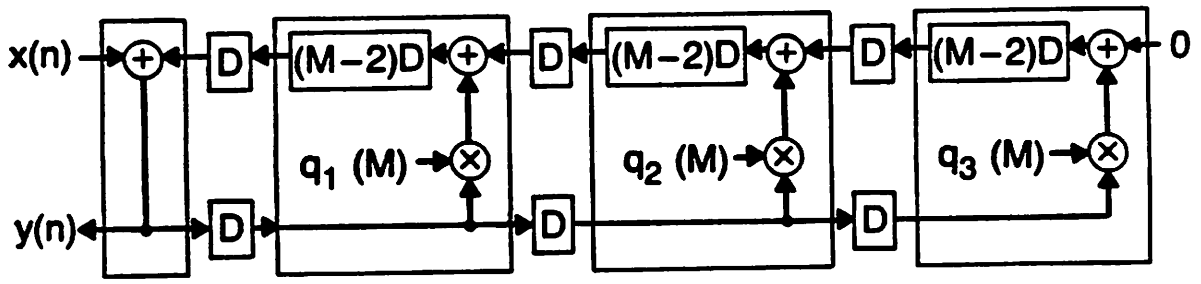


Fig. 3.11(b): Fully pipelined linear bidirectional systolic array.

multiplication/addition operation inside each stage. This technique of moving around delays without changing the input-output behavior is referred to as retiming or cutset transformation [1,34-36]. The pipelined linear systolic array implementation is shown in Fig. 3.11(b).

3.4.7. Pipelined Systolic Ring Implementation

We first review the unidirectional pipelined systolic ring implementation of the direct form recursive filter algorithm, which was presented in [29], and then present pipelined ring implementation of the scattered look-ahead recursive filter algorithm.

Consider implementation of the $(N-1)$-th order recursion

$$y(n)=\sum_{i=1}^{N-1} a_i y(n-i) \tag{3.27}$$

using R unidirectional systolic rings as shown in Fig. 3.12(a). Note that the input and output connections to the external world from the ring architecture have been omitted for clarity. Since the total number of multiplication/addition operations needed is N (where 1 dummy operation has been included) and R processor rings are available, any output computation traverses each processor ring $\frac{N}{R}$ number of times. For instance, for $N=9$ and $R=3$, each computation requires a total of 9 operations, and 3 operations per ring. However, since the order of the filter is 8, each processor holds a single output for 9 consecutive cycles in a single 9-slow (in general N-slow) latch and uses this to contribute to the computation of $(N-1)$ consecutive outputs. For example, processor P_1 uses y_3 for 9 cycles for computation of y_4 through y_{11} (with one dummy cycle). Consider the computation of the output

$$y_8=a_8y_0+a_7y_1+a_6y_2+a_5y_3+a_4y_4+a_3y_5+a_2y_6+a_1y_7\,, \tag{3.28}$$

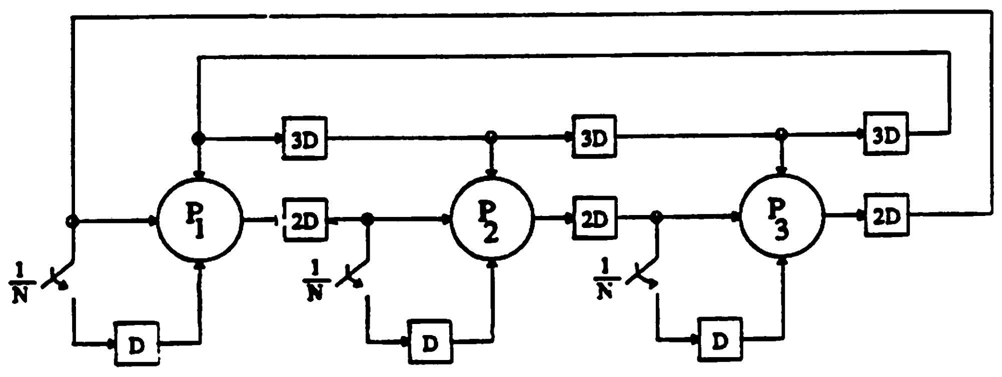


Fig. 3.12(a): Ring implementation of a recursive filter for $N = 8$ and $R = 3$.

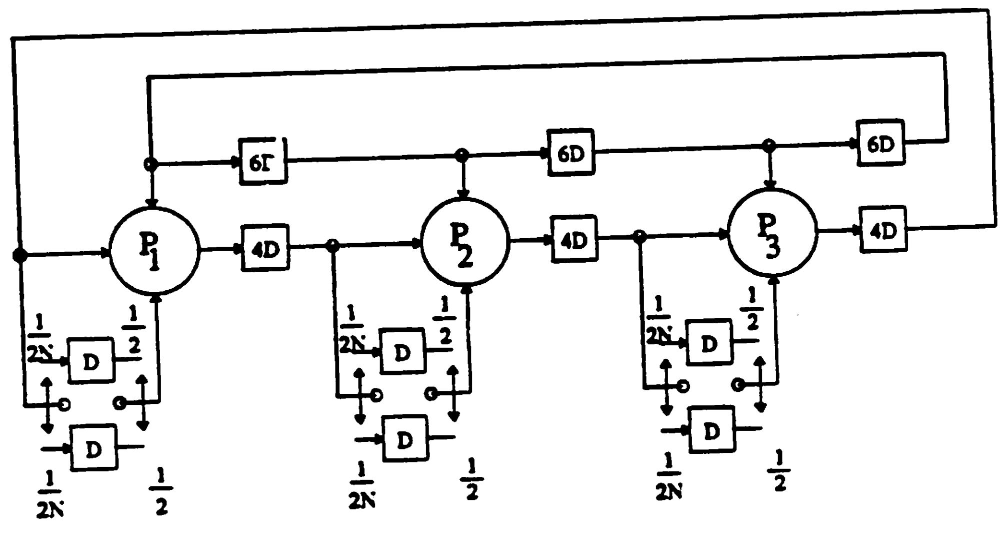


Fig. 3.12(b): Fully pipelined ring implementation for $N = 8$ and $R = 3$, and $M = 2$.

and let processor P_1 compute the term $a_8 y_8$ in cycle 0 (note that processor P_1 doesn't need to store y_0 any more). The processors P_2 and P_3 contribute to the terms $a_7 y_1$ and $a_6 y_2$ respectively. Processor P_1 can contribute to the term $a_5 y_3$ only in cycle 6 (since, y_3 is stored during cycles 1 through 9, and is used for computations of y_4 through y_{11} following a dummy operation step). Hence, the total number of pipeline delays inside the loop must be 6, or in other words, each processor ring can be pipelined by 2 stages in a fully hardware efficient realization (pipelining each ring by more than 2 stages will lead to interleaving and inefficient hardware utilization). In general, for an $(N-1)$-order filter and R rings, each processor can be pipelined by $(\frac{N}{R}-1)$ pipeline stages in a fully hardware efficient realization. In this realization, R outputs are computed in N cycles, which implies a single output is available in $\frac{N}{R}$ cycles, i.e. this realization is $\frac{N}{R}$-slow. The processor P_i computes $y_{kRi+i-1}$. The N coefficients (where the dummy coefficient is 0) in this realization are implemented using another ring consisting of N latches. Furthermore, the latency of a single computation is $R(N-R)$ cycles, where the latency corresponds to the number of cycles between the beginning and the end of computation of a single sample.

Now consider the implementation of the recursive portion of the scattered look-ahead algorithm

$$y(n) = \sum_{i=1}^{(N-1)} q_i(M)\, y(n-iM) \tag{3.29}$$

using unidirectional pipelined systolic rings. The scattered look-ahead algorithm leads to M independent computations which can be performed in an interleaved manner, thereby permitting the rings to be pipelinable by $M(\frac{N}{R}-1)$ stages. This pipelined implementation is still M-slow and requires identical number of cycles per output sample as in the case of $M = 1$, but since the processor ring is pipelined by a factor of M stages higher, the cycle time and the sample period are a factor of M lower. Because of the interleaving, each processor uses M past output samples in an interleaved manner for NM consecutive cycles. For example, for the 8-th order filter and $M = 2$, processor P_1 uses the sample y_6 in all even cycles and the sample y_7 in all odd cycles between the cycles 0 through 18. Between cycles 19 through 37, it uses y_{12} and y_{13} in even and odd cycles respectively. This is realized by using M NM-slow latches for sampling the outputs, and by using M M-slow latches each switched at a rate $\frac{1}{M}$ for computing the outputs as shown in Fig. 3.12(b). Furthermore, because the M independent computations require identical coefficients, each coefficient is repeatedly used for M cycles by each processor. Thus, the NM coefficients (N coefficients and M consecutive copies of each coefficient) are stored in a ring fashion using NM latches. The latency of a single computation is $MR(N-R)$ cycles (which is independent of M in terms of absolute time). In this realization, processor P_i computes $y_{kRMi+i-1}, y_{kRMi+i}, \ldots, y_{kRMi+i+M-2}$ in an interleaved manner.

3.5. PIPELINING IN STATE SPACE FILTERS

The clustered look-ahead and scattered look-ahead processes are identical for the state space filter. Pipelining in state space filters using the look-ahead computation technique (without the use of decomposition) was introduced in [21] at the expense of a linear increase in complexity with respect to loop pipeline stages. In this section, we derive a decomposition based pipelined realization for state space recursive digital filters of logarithmic complexity with respect to the number of loop pipeline stages.

Consider the state space recursive filter described by

$$\mathbf{x}(n+1) = \mathbf{A}\mathbf{x}(n) + \mathbf{b}u(n) \tag{3.30a}$$

$$y(n) = \mathbf{c}^T\mathbf{x}(n) + du(n), \tag{3.30b}$$

where the state $\mathbf{x}(n)$ is $N\times 1$, the state update matrix $\mathbf{A}$ is $N\times N$, $\mathbf{b}$ and $\mathbf{c}$ are $N\times 1$, and d, input sample $u(n)$ and output sample $y(n)$ are scalars, and N is the order of the filter. Fig. 3.13(a) shows a block diagram corresponding to (3.30). The transfer function of the state space filter is given by

$$H(z) = \mathbf{c}^T\,(z\mathbf{I} - \mathbf{A})^{-1}\mathbf{b} + d\ . \tag{3.30c}$$

The state space representation of any transfer function is not unique. The transfer function remains unaltered if the state space representation undergoes a similarity transformation

$$\mathbf{x}\rightarrow\Lambda^{-1}\mathbf{x};\ \mathbf{A},\mathbf{b},\mathbf{c},d\rightarrow\Lambda^{-1}\mathbf{A}\Lambda,\Lambda^{-1}\mathbf{b},\Lambda^T\mathbf{c},d\ . \tag{3.30d}$$

The complexity of the implementation will depend upon the number of non-zero elements in the state update matrix, which in turn depends upon the form of digital filter realization. A parallel realization of first order sections with real coefficients can be described in terms of a diagonal state update matrix, and a cascaded realization of these sections can be described by a triangular state update matrix. Second order sections can be described by a quasi-diagonal state update matrix when implemented in a parallel manner, or a quasi-triangular matrix when implemented in a cascade manner. State space representation of lattice filters can be described by a quasi-triangular state update matrix. Full, triangular and quasi-triangular state update matrices lead to $O(N^2)$ multiplication complexity, whereas diagonal and quasi-diagonal matrices lead to $O(N)$ multiplication complexity, where N is the order of the filter. In what follows, we assume the filter to be described by a quasi-diagonal state update matrix, i.e. the filter has no real

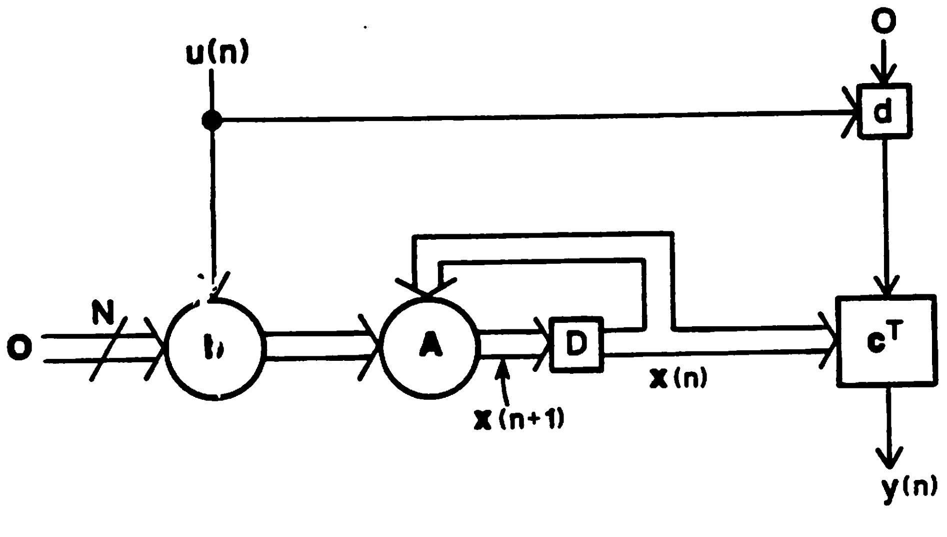


Fig. 3.13(a): A state space recursive digital filter.

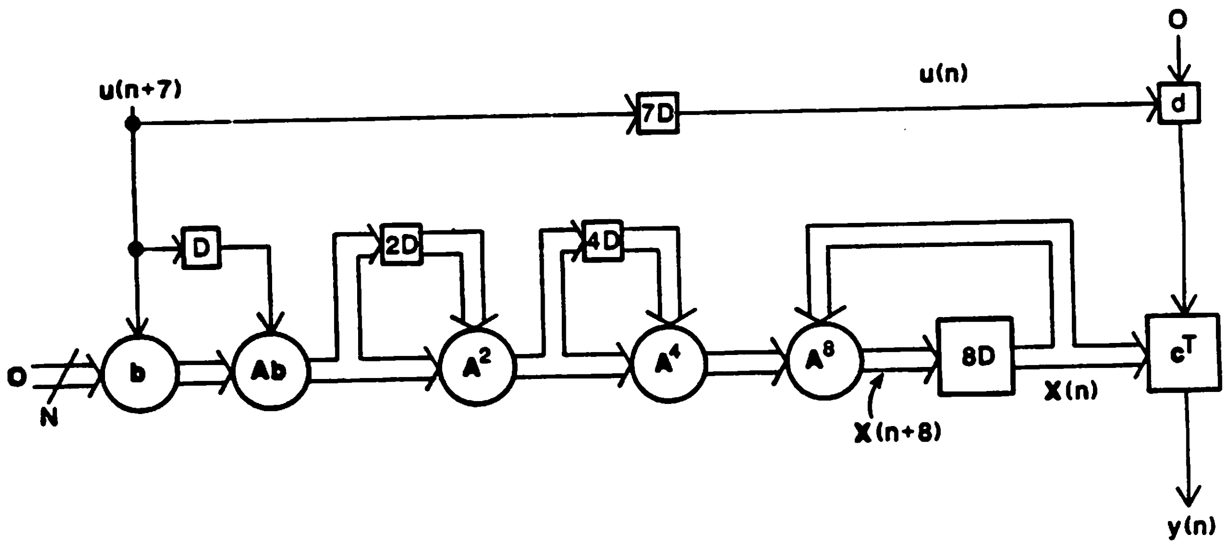


Fig. 3.13(b): A pipelined state space recursive digital filter with 8 loop pipelining stages obtained using the decomposition algorithm.

pole of multiplicity greater than two, and no complex pole of multiplicity greater than unity. Similar results can be easily derived for all other configurations.

Let the maximum number of non-zero elements among all rows of the state update matrix be N'. Then the iteration period of this implementation corresponds to the time required for a single multiplication and N' additions, and the sample rate corresponds to the reciprocal of the iteration period. Applying M steps of look-ahead, we obtain an equivalent M stage pipelined algorithm

$$\mathbf{x}(n+M) = \mathbf{A}^M \mathbf{x}(n) + \sum_{i=0}^{M-1} \mathbf{A}^i \mathbf{b} u(n+M-1-i), \tag{3.31}$$

which has an iteration period bound (sample rate) M times lower (higher) than the original algorithm. The output equation (3.30b) is non-recursive, and does not require any transformation. Let N be the filter order, and N_1 represent the number of real first order poles. Then the state update implementation complexity in (3.31) corresponds to $(NM+2N-N_1)$ multiplications, and the output computation complexity in (3.30b) corresponds to $(N+1)$ multiplications. The total complexity is $(NM+3N+1-N_1)$ multiplications, which is linear with respect to the number of pipeline stages M.

Now we illustrate use of decomposition for the case where M is a power of 2. The decomposed stages of the pipelined state update realization are described by:

$$\mathbf{z}_1(n+M-1) = \mathbf{b}u(n+M-1) + \mathbf{A}\mathbf{b}u(n+M-2) \tag{3.32a}$$

$$\mathbf{z}_{i+1}(n+M-1) = \mathbf{z}_i(n+M-1) + \mathbf{A}^{2^i}\mathbf{z}_i(n+M-1-2^i),\ i = 1,2,\ldots,(\log_2 M-1) \tag{3.32b}$$

$$\mathbf{x}(n+M) = \mathbf{A}^M \mathbf{x}(n) + \mathbf{z}_{\log_2 M}(n+M-1), \tag{3.32c}$$

and is shown in Fig. 3.13(b) (for $M = 8$). This pipelined algorithm leads to a multiplication complexity $\left[2N(\log_2 M + 1) - N_1 \log_2 M\right]$ for state update implementation, and $(N+1)$ for output computation; leading to a total complexity of

$\left\lceil 2N(\log_2 M + \frac{3}{2}) - N_1 \log_2 M + 1 \right\rceil$ multiplications, which is logarithmic with respect to the number of loop pipeline stages.

The roundoff error in the state space pipelined filter is studied in appendix 3.5, and is shown to improve monotonically with increase in the number of loop pipeline stages.

3.6. CONCLUSIONS

We have presented a new scattered look-ahead approach and a decomposition technique to transform recursive filter algorithms to derive equivalent area-efficient pipelined realizations. Another approach to pipeline recursive filters using signed-digit redundant arithmetic has recently been proposed in [55]. Our approaches can be combined with the approach in [55] to pipeline signed-digit recursive filters at bit or digit level with minimum look-ahead. A drawback of the signed-digit representations is that they require a longer word-length for a specified dynamic range (compared to two's complement representation). These representations may also suffer from degraded performance due to overflow problems. These issues require further study.

The pipelinability criteria derived in this chapter can be used to synthesize pipelined filter transfer functions directly from frequency domain specifications thereby eliminating the intermediate transformation procedure. Further research is needed in synthesis of these pipelined recursive filters using constrained iterative design techniques. The iterative techniques have been successfully used in the context of traditional recursive filter design [56-59]. We hope by appropriately constraining these iterative techniques, we can satisfy the pipelinability criteria and design pipelined filters directly from specifications.

The pipelined algorithm described in this chapter suffers from inexact cancellation of poles and zeros, which will lead to error in magnitude and phase response of the filter. However, the word length can be increased to minimize this error. The word-length and roundoff error tradeoffs in the pipelined filters requires further study. Since the coefficients of the pipelined filter correspond to higher power of the original coefficients, they are too small, and may be less than what a finite number of bits can hold. This is another issue that needs further study.

The ultimate speed in the pipelined implementations will be limited by practical limitations such as clock skew, packaging delay etc. Hence, once pipelining is used to maximum possible extent, we need to combine pipelining with block processing to achieve further speedup in the sample rate. In the next chapter, we combine pipelining and incremental block filtering approaches to derive area-efficient architectures for direct form and state space form recursive digital filters.

3.7. APPENDICES

3.7.1. Appendix 3.1

In this appendix, we define the sequence r_i , and study some related recursive relations. The sequence r_i is useful in the context of clustered look-ahead based pipelined and/or block implementation of direct form recursive digital filters. For an N-th order direct form recursive digital filter, we define

$$r_{-i} = 0 \quad \text{for } i = 1, 2, \ldots, (N-1) \, , r_0 = 1$$

and

$$r_i = \sum_{k=1}^{N} a_k \, r_{i-k}, \; i > 0 \tag{A3.1}$$

For the sequence r_i, we can prove the following theorem:

Theorem A3.1: The values of r_{L+m} can be computed using:

$$r_{L+m} = \sum_{k=0}^{N-1} \left[\sum_{j=0}^{k} a_{N-j} r_{m+j-k} \right] r_{L-N+k} \; . \tag{A3.2}$$

Proof (by induction): Assume the theorem to hold for m, and prove that theorem also holds for m+1. The value of r_{L+m+1} is given by

$$\begin{aligned} r_{L+m+1} &= \sum_{l=1}^{N} a_l \left[\sum_{k=0}^{N-1} \left\{ \sum_{j=0}^{k} a_{N-j} \, r_{m-l+j-k+1} \right\} r_{L-N+k} \right] \\ &= \sum_{k=0}^{N-1} \left[\sum_{l=1}^{N} \sum_{j=0}^{k} a_l \, a_{N-j} \, r_{m-l+j-k+1} \right] r_{L-N+k} \\ &= \sum_{k=0}^{N-1} \left[\sum_{j=0}^{k} a_{N-j} \sum_{l=1}^{N} a_l \, r_{m-l+j-k+1} \right] r_{L-N+k} \end{aligned}$$

from which (A3.2) follows.

3.7.2. Appendix 3.2

In this appendix, we derive a pipelined realization of the direct form recursive digital filter using the *clustered look-ahead* computation technique. We also study the time domain and frequency domain interpretations of this transformation.

Theorem A3.2: Any direct form recursive digital filter of the form (3.8) can be equivalently described by

$$y(n) = \sum_{j=0}^{N-1} \left[\sum_{k=j+1}^{N} a_k \, r_{j+M-k} \right] y(n-j-M) + \sum_{j=0}^{M-1} r_j \, z(n-j) \tag{A3.3}$$

which corresponds to an M-stage pipelined realization.

Proof (by induction): Assume the above to be true for M, and prove that it also holds for M+1. The above expression can be rewritten as

$$y(n)=r_M\, y(n-M)+\sum_{j=1}^{N-1}\left[\sum_{k=j+1}^{N} a_k\, r_{j+M-k}\right] y(n-j-M)+\sum_{j=0}^{M-1} r_j\, z(n-j) \quad \text{(A3.4)}$$

$$=r_M\left[\sum_{j=1}^{N} a_j y(n-M-j)+z(n-M)\right]+\sum_{j=1}^{N-1}\left[\sum_{k=j+1}^{N} a_k r_{j+M-k}\right] y(n-M-j)+\sum_{j=0}^{M-1} r_j z(n-j)$$

$$=\sum_{j=1}^{N}\left[\sum_{k=j}^{N} a_k\, r_{j+M-k}\right] y(n-M-j)+\sum_{j=0}^{M} r_j\, z(n-j)$$

from which (A3.3) follows.

Theorem A3.3: Any transfer function of the form (3.7) is equivalent to the form (3.10)

Proof: Multiply the numerator and denominator by a $(M-1)$ order polynomial $\sum_{j=0}^{M-1} r_j\, z^{-j}$.

The denominator $D(z)$ can be expressed as:

$$D(z)=\left[1-\sum_{i=1}^{N} a_i z^{-i}\right]\left[1+\sum_{j=1}^{M-1} r_j\, z^{-j}\right] \quad \text{since } r_0=1 \qquad \text{(A3.4)}$$

$$=1-\sum_{i=1}^{N} a_i z^{-i}+\sum_{i=1}^{M-1} r_i z^{-i}-\sum_{i=1}^{N}\sum_{j=1}^{M-1} a_i r_j z^{-(i+j)}.$$

The last term of the right hand side of the above equation is given by

$$\sum_{i=1}^{N}\sum_{j=1}^{M-1} a_i r_j z^{-(i+j)}=\begin{cases}\displaystyle\sum_{i=2}^{M-1}\left[\sum_{j=1}^{i-1} a_j r_{i-j}\right] z^{-i}+\sum_{i=M}^{N}\left[\sum_{j=1}^{M-1} a_{i-j} r_j\right] z^{-i}+\sum_{i=N+1}^{N+M-1}\left[\sum_{j=i-N}^{M-1} a_{i-j} r_j\right] z^{-i}, & N\geq M\\ \displaystyle\sum_{i=2}^{N}\left[\sum_{j=1}^{i-1} a_j r_{i-j}\right] z^{-i}+\sum_{i=N+1}^{M-1}\left[\sum_{j=1}^{N} a_j r_{i-j}\right] z^{-i}+\sum_{i=M}^{N+M-1}\left[\sum_{j=i-M+1}^{N} a_j r_{i-j}\right] z^{-i} & N<M\end{cases} \quad (A\,3.5)$$

from which (3.10) follows.

3.7.3. Appendix 3.3

Theorem A3.4: The sequence $q_i(2K)$ is related to the sequence $q_i(K)$ by the following relations:

$$q_1(2K) = q_1^2(K) + 2q_2(K), \quad q_N(2K) = (-1)^{N+1} q_N^2(K) \tag{A3.6}$$

For even filter order N,

$$q_i(2K) = \begin{cases} 2q_{2i}(K) + (-1)^{i+1} q_i^2(K) + 2\sum_{j=1}^{i-1} (-1)^{j+1} q_j(K) q_{2i-j}(K), & i = 2, \ldots, \frac{N}{2} \\ (-1)^{i+1} q_i^2(K) + 2\sum_{j=i+1}^{N} (-1)^{j+1} q_j(K) q_{2i-j}(K), & i = \frac{N}{2}+1, \ldots, N-1 \end{cases} \tag{A3.7}$$

For odd filter order N,

$$q_i(2K) = \begin{cases} 2q_{2i}(K) + (-1)^{i+1} q_i^2(K) + 2\sum_{j=1}^{i-1} (-1)^{j+1} q_j(K) q_{2i-j}(K), & i = 2, \ldots, \frac{N-1}{2} \\ (-1)^{i+1} q_i^2(K) + 2\sum_{j=1}^{i-1} (-1)^{j+1} q_j(K) q_{2i-j}(K), & i = \frac{N+1}{2} \\ (-1)^{i+1} q_i^2(K) + 2\sum_{j=i+1}^{N} (-1)^{j+1} q_j(K) q_{2i-j}(K), & i = \frac{N+3}{2}, \ldots, N-1 \end{cases} \tag{A3.8}$$

Proof: We have

$$1 - \sum_{i=1}^{N} q_i(2K) z^{-2iK} = \left[1 - \sum_{i=1}^{N} q_i(K) z^{-iK}\right] \left[1 - \sum_{i=1}^{N} (-1)^i q_i(K) z^{-iK}\right]$$

$$= 1 - 2\sum_{i=1}^{\lfloor \frac{N}{2} \rfloor} q_{2i}(K) z^{-2iK} + \sum_{i=1}^{N} (-1)^i q_i^2(K) z^{-2iK}$$

$$+ 2\sum_{i=2}^{\lceil \frac{N}{2} \rceil} \sum_{j=1}^{i-1} (-1)^j q_j(K) q_{2i-j}(K) z^{-2iK} + 2\sum_{i=\lceil \frac{N}{2} \rceil + 1}^{N-1} \sum_{j=i+1}^{N} (-1)^j q_j(K) q_{2i-j}(K) z^{-2iK}$$

Matching the powers of z, the above relations can be derived.

3.7.4. Appendix 3.4

In this appendix, we study the quantization error due to the recursive portion of a pipelined second order recursive filter derived by using scattered look-ahead and decomposition techniques. We show that as the number of pipeline stages inside the recursive loop increases, the upper bound on the quantization error of the pipelined filter strictly decreases. Consider the second order recursive filter transfer function

$$H(z) = \frac{1}{(1-re^{j\theta}z^{-1})(1-re^{-j\theta}z^{-1})} = \frac{1}{1-q_1(1)z^{-1}-q_2(1)z^{-1}} \cdot \tag{A3.9}$$

In the pipelined filter $2(M-1)$ poles are introduced on the circle r units apart in the Z-plane. Since, the distance of the additional poles remains unaltered from the origin, the pipelined filter remains stable if the original filter is stable. The recursive portion of the pipelined transfer function with M pipeline stages inside the recursive loop is given by

$$H_1(z) = \frac{1}{\prod_{i=0}^{M-1}(1-re^{j(\theta+\frac{2\pi i}{M})}z^{-1})(1-re^{-j(\theta-\frac{2\pi i}{M})}z^{-1})} \cdot \tag{A3.10}$$

The quantization error of this filter can be derived by evaluating the residues of $\frac{H_1(z)H_1(z^{-1})}{z}$ for all poles inside the unit circle, and then taking the sum. We derive the quantization error of this filter in three steps.

Step 1: Consider the pole p_i at $re^{j(\theta+\frac{2\pi i}{M})}$, and let its residue be denoted as R_{i1}. Let R_{i2} denote the residue at the complex conjugate pole of p_i. Then we can prove that R_{i2} is the complex conjugate of R_{i1}. This proof is straightforward and is omitted.

Step 2: In the pipelined filter, $(M-1)$ additional poles are introduced for each pole in the original filter. We show that the residue at each of the added pole is identical to that at the corresponding pole of the original filter.

Proof: For the pipelined filter, we have

$$H_1(z)H_1(z^{-1}) = \frac{1}{\prod_{i=0}^{M-1}(1-re^{j(\theta+\frac{2\pi i}{M})}z^{-1})(1-re^{-j(\theta-\frac{2\pi i}{M})}z^{-1})} \tag{A3.11}$$

$$\times \frac{1}{\prod_{i=0}^{M-1}(1-re^{j(\theta+\frac{2\pi i}{M})}z)(1-re^{-j(\theta-\frac{2\pi i}{M})}z)} .$$

We derive the residue at the pole for $i = I$ and show that this residue is independent of I. After some manipulation, the residue R_{I1} can be derived to be

$$R_{I1} = \frac{1}{\prod_{i=1}^{M-1}(1-e^{\frac{j2\pi i}{M}})\prod_{i=0}^{M-1}((1-r^2e^{j(2\theta+\frac{2\pi i}{M})})(1-e^{-j(2\theta-\frac{2\pi i}{M})})(1-r^2e^{j\frac{2\pi i}{M}}))} \tag{A3.12}$$

$$= \frac{1}{M(1-r^{2M})(1-r^{2M}e^{j2M\theta})(1-e^{-j2M\theta})} ,$$

which is independent of I.

Step 3: The total quantization error is proportional to (also referred to as normalized error)

$$E = M(R_{i1}+R_{i2}) = \frac{1+r^{2M}}{1-r^{2M}}\frac{1}{1-2r^{2M}\cos 2M\theta+r^{4M}} . \tag{A3.13}$$

In terms of the coefficients of the pipelined filter, the normalized error is given by

$$E = \frac{1-q_2(M)}{1+q_2(M)}\frac{1}{1-q_1(2M)-q_2(2M)} . \tag{A3.14}$$

The upper bound on the error expression in (A3.13) strictly decreases with increase in M. Furthermore, the expression (A3.14) also holds for the case of two real poles. A similar analysis can be carried out for any arbitrary order recursive digital filter, and the bound on quantization error of the pipelined realization can be shown to improve with increase in M.

3.7.5. Appendix 3.5

In this appendix, we study the roundoff noise error in pipelined state space filters, and show that the roundoff error strictly improves with increase in the number of loop pipeline stages M. We assume the roundoff operation to be performed at the output of the state variables and at system outputs. The noise sources are assumed to be stationary white with zero mean, and are assumed to be statistically independent of signals.

The output error at the summing node of the pipelined state space filter with M loop pipeline stages is given by

$$\tilde{\mathbf{x}}(n+M) = \mathbf{A}^M \tilde{\mathbf{x}}(n) + \mathbf{e}_s(n)\,, \tag{A3.15}$$

where $\mathbf{e}_s$ is of dimension $N \times 1$ and

$$E\left[\mathbf{e}_s \mathbf{e}_s^T\right] = \sigma_0^2 \mathbf{I}_N\,. \tag{A3.16}$$

The matrix $\mathbf{I}_N$ represents the unity matrix of dimension N. The variance of the errors at the summing nodes of the state variables is described by the covariance matrix

$$\mathbf{Q} = E\left[\tilde{\mathbf{x}}\tilde{\mathbf{x}}^T\right] = \mathbf{A}^M \mathbf{Q} (\mathbf{A}^T)^M + \sigma_0^2 \mathbf{I}_N = \sigma_0^2 \sum_{p=0}^{\infty} \mathbf{A}^{pM} (\mathbf{A}^T)^{pM}\,. \tag{A3.17}$$

The error at the summing node of the output is described by

$$\tilde{y}(n) = \mathbf{c}^T \tilde{\mathbf{x}}(n) + e(n)\,, \tag{A3.18}$$

where the last term corresponds to the error at the output summation node and

$$E\left[e^2(n)\right] = \sigma_0^2.$$

The variance of the error at the output summing node is given by (using (A3.17))

$$\frac{\sigma_{\tilde{y}}^2}{\sigma_0^2} = \mathbf{c}^T \sum_{p=0}^{\infty} \mathbf{A}^{pM} (\mathbf{A}^T)^{pM} \mathbf{c} + 1\,, \tag{A3.19}$$

which is a strictly decreasing function in M.

3.8. REFERENCES

(1) Kung., S.Y., "On Supercomputing with Systolic/Wavefront Array Processors", *Proceedings of IEEE*, Vol. 72, No. 7, July 1984

(2) Kung, H.T., "Why Systolic Architectures", *IEEE Computer*, Vol. 15, No. 1, January 1982

(3) Gold, B. and Jordan, K.L.,"A Note on Digital Filter Synthesis", *Proceedings of IEEE*, Vol. 65, pp.1717-1718, Oct. 1968

(4) Voelcker, H.B., and Hartquist, E.E.,"Digital Filtering Via Block Recursion", *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-18, pp. 169-176, June 1970.

(5) Burrus, C.S., "Block Implementation of Digital Filters ", *IEEE Transactions on Circuit Theory*, Vol. CT-18, pp. 697-701, Nov. 1971.

(6) Moyer, A.L., "An Efficient Parallel Algorithm for Digital IIR Filters", *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 525-528

(7) Mitra, S.K. and Gnanasekaran, R., "Block Implementation of Recursive Digital Filters - New Structures and Properties ", *IEEE Transactions on Circuits and Systems*, Vol. CAS-25, pp.200-207, April 1978.

(8) Barnes, C.W. and Shinnaka, S.,"Block Shift Invariance and Block Implementation of of Discrete-Time Filters", *IEEE Trans on* Circuits and Systems, Vol. CAS-27, pp.567-672, Aug. 1980.

(9) Zeman, J., and Lindgren, A.G., "Fast Digital Filters with Low round-off Noise, " *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, pp.716-723, July 1981.

(10) Schwartz, D. A. and Barnwell, T.P. III, "Increasing the Parallelism of Filters Through Transformation to Block State Variable Form", *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 1984.

(11) Lu, H.H., Lee, E.A. and Messerschmitt, D.G., "Fast Recursive Filtering with Multiple Slow Processing Elements", *IEEE Trans. on Circuits and Systems*, Vol. CAS-32, No.11, November 1985, pp. 1119-1129.

(12) Nikias, C.L.,"Fast Block Data Processing Via a New IIR Digital Filter Structure", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 32, No.4, August 1984.

(13) Parhi, K.K., and Messerschmitt, D.G., "Block Digital Filtering via Incremental Block-State Structure", *Proceedings of the IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987

(14) Meng, T., and Messerschmitt, D.G., "Implementation of Arbitrarily Fast Adaptive Lattice Filters with Multiple Slow Processing Elements", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, 1986

(15) Sung, W., and Mitra, S.K., "Efficient Multi-Processor Implementation of Recursive Digital Filters", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, April 1986, pp. 257-260

(16) Wu, C.W., and Cappello, P.R., "Computer-Aided Design of VLSI Second Order Sections", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987

(17) Arun, K.S., "Ultra-High-Speed Parallel Implementation of Low-Order Digital Filters", *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1986, pp. 944-946

(18) Loomis, H.H., and Sinha, B., "High Speed Recursive Digital Filter Realization", *Circuits, Systems, and Signal Processing*, Vol. 3, No.3, pp. 267-294

(19) Tiemann, J.J., and Vogelsong, T.L., "Charge Domain Integrated Circuits for Signal Processing", *Proc. of the 1984 International Symposium on Circuits and Systems*

(20) Parhi, K.K., and Messerschmitt, D.G., "Look-Ahead Computation: Improving Iteration Bound in Linear Recursions", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1987, Dallas

(21) Parhi, K.K., and Messerschmitt, D.G., "A Bit-Parallel Bit Level Recursive Filter Architecture", *Proc. of the IEEE International Conference on Computer Design*, NY, 1986.

(22) Parhi, K.K., and Messerschmitt, D.G., "Area-Efficient High Speed VLSI Adaptive Filter Architectures", *Proceedings of the IEEE International Conference on Communications*, Seattle, June 1987

(23) Parhi, K.K., and Messerschmitt, D.G., "Concurrent Cellular VLSI Adaptive Filter Architectures", *IEEE Transactions on Circuits and Systems*, Vol. CAS-34, October 1987

(24) Gao, G.R., "Maximum Pipelining Linear Recurrence on Static Data Flow Computers", *International Journal of Parallel Programming*, Vol. 15, No. 2, 1986, pp. 127-149

(25) Kung, H.T., "Special Purpose Devices for Signal and Image Processing: An Opportunity in Very Large Scale Integration (VLSI)", *Proc. of SPIE*, Vol. 241, Real Time Signal Processing III, July 1980, pp. 76-84

(26) Heller, D.E., "Decomposition of Recursive Filters for Linear Systolic Arrays", *Proc. of SPIE*, Vol. 431, Real Time Signal Processing VI, 1983, pp. 55-59

(27) Kung, S.Y., "From Transversal Filter to Wavefront Array", *Proc. of International Conference on VLSI*, edited by F. Anceau and E.J. Aas, Elsevier Publishers, IFIP, North-Holland, pp. 247-261, 1983

(28) Rao, S.K. and Kailath, T., "Digital Filtering in VLSI", *Proc of 22nd Allerton Conference*, 1984

(29) Kung, H.T. and Lam, M.S., "Wafer Scale Integration and Two-Level Pipelined Implementation of Systolic Arrays", *Journal of Parallel and Distributed Computing*, Vol. 1, 1984, pp. 32-63

(30) Renfors, M., and Neuvo, Y., "The Maximum Sampling Rate of Digital Filters under Hardware Speed Constraints", *IEEE Trans. on Circuits and Systems*, Vol. CAS-28, No. 3, March 1981, pp. 196-202

(31) Fettweis, A., "Realizability of Digital Filter Networks", *Arch. Elek. Ubertrangung*, Vol. 30, Feb 1976, pp. 90-96

(32) Lee, E.A., and Messerschmitt, D.G., "Pipeline Interleaved Programmable DSP's: Architecture", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-35, No. 9, September 1987, pp. 1320-1333

(33) Schwartz, D.A., "Synchronous Multiprocessor Realizations of Shift Invariant Flow Graphs", *Ph.D. Dissertation*, Georgia Institute of Technology, Technical Report DSPL-85-2, July 1985

(34) Leiserson, C., Rose, F., and Saxe, J., "Optimizing Synchronous Circuitry by retiming", *Third Caltech Conference on VLSI*, Pasadena, CA, March 1983

(35) Leiserson, C.E., and Saxe, J.B., "Optimizing Synchronous Systems", *Proceedings of 22nd Annual Symposium on Foundations of Computer Science*, 1981

(36) Schwartz, D.A., and Barnwell III, T.P., "A Graph Theoretic Technique for the Generation of Systolic Implementations for Shift Invariant Flow Graphs", *Proc of ICASSP-84*, San Diego, March 1984

(37) Parhi, K.K., and Messerschmitt, D.G., "Pipelined VLSI Recursive Filters using Scattered Look-Ahead and Decomposition", *Proc. of 1988 IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1988, NY, pp. 2120-2123

(38) Stone, H.S., "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, January 1973, pp. 27-38

(39) Kogge, P.M., and Stone, H.S., "A Parallel Algorithm for the Efficient Solution of A General Class of Recurrence Equations", *IEEE Transactions on Computers*, Vol. C-22, August 1973, pp. 786-792

(40) Kogge, P.M., "Parallel Solution of Recurrence Problems", *IBM Journal of Research and Development*, Vol. 18, March 1974, pp. 138-148

(41) Chen, S-C., and Kuck, D.J., "Time and Parallel Processor Bounds for Linear Recurrence Systems", *IEEE Transactions on Computers*, Vol. C-24, No. 7, July 1975, pp. 701-717

(42) Gajski, D.D., "Processor Array for Computing Linear Recurrence Systems", *Proceedings of IEEE International Conference on Parallel Processing*, 1978, pp. 246-256

(43) Gajski, D.D., "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines", *IEEE Trans. on Computers*, Vol. C-30, No. 3, March 1981, pp. 190-206

(44) Hockney, R.W., "A Fast Direct Solution of Poisson Equation using Fourier Analysis", *Journal of the Assoc. Comput. Machinery*, Vol. 12, No. 1, January 1965, pp. 95-113

(45) Buzzbee, B.L., Golub, G.H., and Nielson, C.W., "On Direct Methods for Solving Poisson's Equations", *SIAM Journal of Numerical Analysis*, Vol. 7, No. 4, Dec. 1970, pp. 627-656

(46) Sweet, R.A., "A Generalized Cyclic Reduction Algorithm", *SIAM Journal of Numerical Analysis*, Vol. 11, No. 3, June 1974, pp. 506-520

(47) Swarztrauber, P.N., "A Direct Method for Discrete Solution of Separable Elliptic Equations", *SIAM Journal of Numerical Analysis*, Vol. 11, No. 6, December 1974, pp. 1136-1150

(48) Swarztrauber, P.N., "The Methods of Cyclic Reduction, Fourier Analysis, and the FACR Algorithm for the Discrete Solution of Poisson's Equations on a Rectangle", *SIAM Review*, Vol. 19, No. 3, July 1977, pp. 490-501

(49) Hockney, R.W., and Jesshope, C.R., *Parallel Computers: Architecture, Programming, and Algorithms*, Adam Hilger, Bristol, 1981

(50) Oed, W., and Lange, O., "The Solution of Linear Recurrence Relations on Pipelined Processors", *Proceedings of the International Conference on Parallel Processing, 1983, pp. 545-547*

(51) Bellanger, M.G., Bonnerot, G., and Coudreuse, M., "Digital Filtering by Polyphase Network: Application to Sample Rate Alteration and Filter Banks", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-24, April 1976, pp. 109-114

(52) Zhou, B.B., and Brent, R.P., "A Two-Level Pipelined Implementation of Direct Form Recursive Digital Filters", *Technical Report*, Computer Science Laboratory, The Australian National University, 1988

(53) Parhi, K.K., and Hatamian, M., "A High Sample Rate Recursive Digital Filter Chip", *Proceedings of the 1988 IEEE VLSI Signal Processing Workshop*, November 1988, Monterey, CA, IEEE Press

(54) Parhi, K.K., Chen, W.L., and Messerschmitt, D.G., "Architecture Considerations for High Speed Recursive Filtering", *Proc. of the 1987 IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987, pp. 374-377

(55) R.F. Woods, S.C. Knowles, J.V. McCanny, and J.G. McWhirther, "Systolic IIR Filters with Bit Level Pipelining", *Proceedings of the 1988 International Conference on Acoustics, Speech, and Signal Processing*, New York, April 1988, pp. 2072-2075

(56) Rabiner, L.R., Kaiser, J.F., Herrman, O., and Dolan, M.T., "Some Comparisons between FIR and IIR Digital Filters", *Bell System Technical Journal*, Vol. 53, March 1981, pp. 305-331,

(57) Steiglitz, K., "Computer Aided Design of Recursive Digital Filters", *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-18, No. 2, June 1970, pp. 123-129

(58) Deczky, A.G., "Synthesis of Recursive Digital Filters using the Minimum p-Error Criterion", *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-20, No. 4, October 1972, pp. 257-263

(59) Rabiner, L.R. *et al*, "Linear Programming Design of IIR Digital Filters with Arbitrary Magnitude Function", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, No. 2, April 1974, pp. 117-123

4

PIPIELINED INCREMENTAL BLOCK FILTER

4.1. INTRODUCTION

In chapter 3, we proposed the *scattered look-ahead* and *decomposition* techniques to pipeline direct form and state space form recursive digital filters with a logarithmic increase in hardware with respect to the number of loop pipeline stages.

Another approach to achieving concurrency in recursive digital filters is by the use of "block processing" [1-19]. In block realizations, input samples are processed in the form of non-overlapping blocks to generate non-overlapping blocks of output samples (see Fig. 4.1). The block of multiple inputs are derived from the single serial input by using a serial-to-parallel converter at the input, and the serial output is derived from the block of outputs by using a parallel-to-serial converter at the output. Because of this serial-to-parallel conversion, the multiple-input-multiple-output (MIMO) or the block system operates at a rate L times slower than that of the converter circuits, where L is the block size. The clock period of each latch in the MIMO system is L times greater than that of the sample period, or equivalently each *block delay operator* in the MIMO system is L*−slow* [20-22]. Hence, for a given technology, we can increase the block size to obtain arbitrarily high sampling rate filter realizations. Due to this L*−slow* block delay operator, the block state update operation requires updating the state $\mathbf{x}(kL+L)$ based on the past state $\mathbf{x}(kL)$. Note that in the block state update operation, the $(L-1)$ intermediate states $\mathbf{x}(kL+1)$, $\mathbf{x}(kL+2)$, ..., and $\mathbf{x}(kL+L-1)$ have been missed, unlike in pipelined realizations where each state is computed. This block state update operation can be achieved by iterating the original recursion $(L-1)$ times (using the clustered look-ahead approach) to create a single $L-slow$ delay operator in the recursive loop.

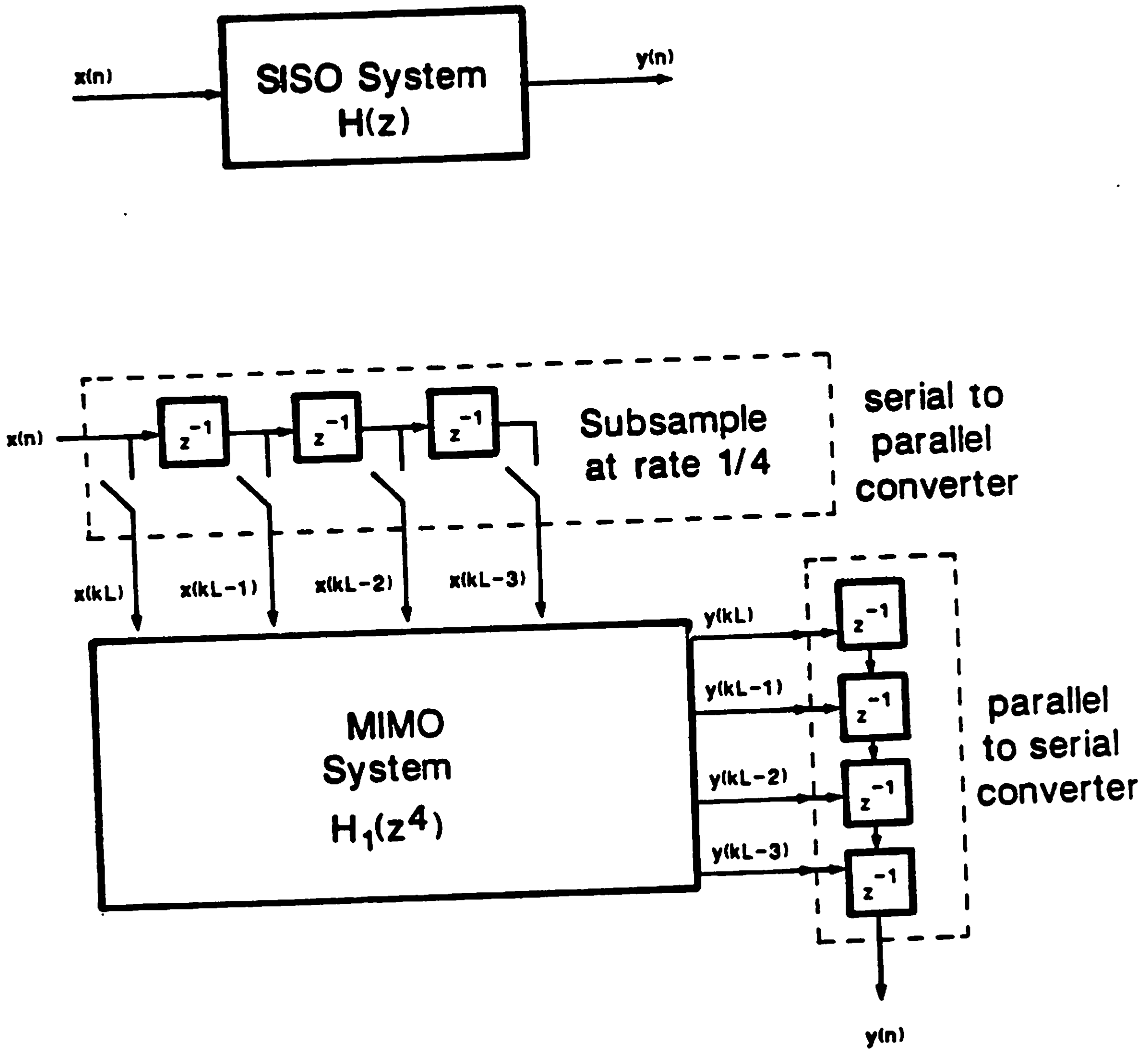


Fig. 4.1. Block implementation of digital filters (for block size of 4). Each latch in the MIMO system is 4-slow.

In the block filter, for a stable system, the poles (eigenvalues) of the block filter move closer to the origin than that for the word-serial system (since, the eigenvalues of the block system are L-th power of those of the word-serial system). These block structures offer several advantages over the word-serial realizations. These include (i) a linearly proportional increase (decrease) in the sampling rate (iteration period) with increase in block size, and (ii) a linearly proportional decrease in the average roundoff noise at the output. One approach to implement the block filter is to use the concurrent scattered look-ahead algorithms without decomposition derived in the last chapter with replicated parallel hardware instead of pipelined hardware. However, each output requires an $O(NL)$ multiplication complexity, and this block structure requires a total of $O(NL^2)$ multiplication/addition operations, which is square in block size. In this parallel hardware block implementation, the decomposition technique can no longer be exploited because each delay now is a block delay. Furthermore, all existing approaches to block recursive digital filtering also lead to a square multiplication complexity in block size. It is the objective of this chapter to derive block recursive digital filter structures with multiplication complexity linear in block size.

In direct form recursive block digital filters, the block of L outputs are computed using past block of outputs, and this leads to a square complexity (since the block state update operation is expensive). Sung and Mitra recently computed L blocks of outputs (with block size L) and first exploited inter-block parallelism and then intra-block parallelism to get a linear speedup with respect to the number of processors, at the expense of a larger memory space [12]. Wu and Cappello proposed a new scheme to implement second order direct form recursive digital filters [13-14] of complexity linear in block size L. Instead of updating the whole block of L outputs, they updated only N outputs (where N is the filter order) and computed the remaining $(L-N)$ outputs in a non-recursive or sequential manner using these updated N outputs. This process of computing the outputs is referred to as *incremental output computation*. In this chapter, we extend Wu and Cappello's direct form structure from second order to higher order case (with a slight variation, we update the first N outputs in each block, whereas the structure in [13-14] updates the last N outputs in each block). This block filter complexity is linear with respect to block size, and the complexity per output sample is independent of block size.

We propose a new technique of incremental output computation in state space digital filters. Here we compute the outputs incrementally in a sequential manner using the non-recursively computed intermediate states (which were missed in the block state update process). As an example, for a block size of 20 and an *increment size* of 5, we compute $y(20k)$ through $y(20k+4)$ using the state $\mathbf{x}(20k)$, then we compute the intermediate state $\mathbf{x}(20k+5)$ non-recursively (which was missed due to the block state update process), and compute the incremental outputs $y(20k+5)$ through $y(20k+9)$ using this state. Then we compute the state $\mathbf{x}(20k+10)$ and use this to compute $y(20k+10)$ through $y(20k+14)$, and finally compute $\mathbf{x}(20k+15)$ and use this to compute the last incremental output $y(20k+15)$ through $y(20k+19)$. A family of filter structures can be described with different values of increment size. In particular, the existing block state filter structure corresponds to the case where the increment size equals the block size. We derive the optimum increment size as a function of the filter order in a way that minimizes the multiplication complexity of the incremental block filter. The incremental block state filter is also extended to the multirate filtering case.

It is preferable to use pipelining to the maximum possible extent first, since pipelining exploits concurrency with reduced hardware (i.e. with logarithmic increase as opposed to linear as in block processing). This conclusion is clear from Table 4.1, which compares the number of multiply/add operations for a second order recursive filter, for direct and state space forms, for pipelining and incremental block processing approaches, for typical factors of speedup or increase in the sample rate.

Table 4.1: Complexity of Pipelined and Block Second Order Filters

Speedup	Direct Form		State Space Form	
	Pipelined	Block	Pipelined	Block
1	5	5	9	9
2	7	11	13	15
4	9	25	17	30
8	11	53	21	67
16	13	109	25	142

Note that all the multipliers in the pipelined filter are pipelined by M levels or stages, and these pipelined multipliers require additional area for the latches. In contrast, the multipliers in the block implementation require single stage pipelining. The latch areas in the pipelined implementation cannot be simply neglected, since each binary latch costs about one third to one fourth of a binary adder in terms of silicon area. However, if we compare the complexities of the block filter and the pipelined filter, we observe that the pipelined filter is far more attractive for implementation, even after accounting for the latch areas in the pipelined structure. If sufficient speed cannot be generated by pipelining alone, then we can combine pipelining with block processing (i.e. we can get a speedup by a factor of LM using a block size of L, and M pipeline stages inside the recursive loop of the block filter) [16-18].

In this chapter, we combine pipeline interleaving and incremental block filtering approaches to derive extensively pipelined direct form and state space form incremental block filters by introducing several pipeline stages inside the recursive loop of the incremental block filter. The pipelined block realizations are derived by using the techniques of scattered look-ahead computation (to introduce several loop pipeline stages), decomposition (to obtain logarithmic complexity realization with respect to pipelining), clustered look-ahead computation (for block state update operation), and incremental output computation (for linear complexity in block size). The total multiplication complexity of our pipelined block filters is linear in block size, logarithmic with respect to the number of loop pipeline stages, and the complexities due to pipelining and block processing are additive. Because of the scattered look-ahead approach, the distance of the poles or the eigenvalues of the fine-grain pipelined block filter from the origin is identical to that of the block filter with a single latch inside the recursive loop. Thus, the stability of these filters is not affected due to fine-grain pipelining inside the block filter.

The organization of this chapter is as follows. The incremental block filter structure is derived in sections 4.2 and 4.3 respectively for direct form and state space form recursive filters. In sections 4.4 and 4.5, we derive fine-grain pipelined block filter structures for direct form and state space form recursive filters respectively.

4.2. DIRECT FORM BLOCK FILTERS

Consider the N-th order direct form filter described by

$$y(n) = \sum_{i=1}^{N} a_i y(n-i) + z(n) \,, \tag{4.1a}$$

$$z(n) = \sum_{i=0}^{N} b_i u(n-i) \,. \tag{4:1b}$$

We can transform the above description to an equivalent block description, where we compute L outputs using L inputs for a block size of L. The state update operation in block filters is based on the clustered look-ahead approach described in chapter 3. In the clustered look-ahead approach, the output sample $y(n)$ is computed in terms of past N clustered samples $y(n-M)$, $y(n-M-1)$, ..., and $y(n-M-N+1)$ (i.e. bypassing $(M-1)$ immediate past outpus), and is given by

$$y(n) = \sum_{j=0}^{N-1} \left[\sum_{k=j+1}^{N} a_k r_{j+M-k} \right] y(n-j-M) + \sum_{j=0}^{M-1} r_j \, z(n-j) \, . \tag{4.2}$$

This relation was derived in appendix 3.2 in the context of pipelining, and is used here in the context of block filtering. Let a block of samples be denoted by

$$\mathbf{y}^{(L)}(i) = \left[y(i), y(i+1), \ldots, y(i+L-1) \right]^T . \tag{4.3}$$

In the existing block structures, the block of outputs $\mathbf{y}^{(L)}(kL+L)$ are updated based on the past block of outputs $\mathbf{y}^{(L)}(kL)$, with a square complexity in block size.

In the incremental block filter, we update only N states recursively using the clustered look-ahead approach, and compute the remaining $(L-N)$ states in a non-recursive or sequential manner using these N updated states. The schedule of such an incremental block filter is shown in Fig. 4.2, where the outputs $y(kL+L)$ through $y(kL+L+N-1)$ are updated recursively, and the remaining $(L-N)$ outputs $y(kL+L+N)$ through $y(kL+2L-1)$ are computed in a non-recursive manner. This incremental output computation is the key in obtaining a linear complexity implementation in block size. If the block size is less than the filter order, the complexity is inherently proportional to square of the block size, but this is not a problem since the block size is not large. Below we formulate the block filter for the cases where the block size is greater than and less than the filter order respectively.

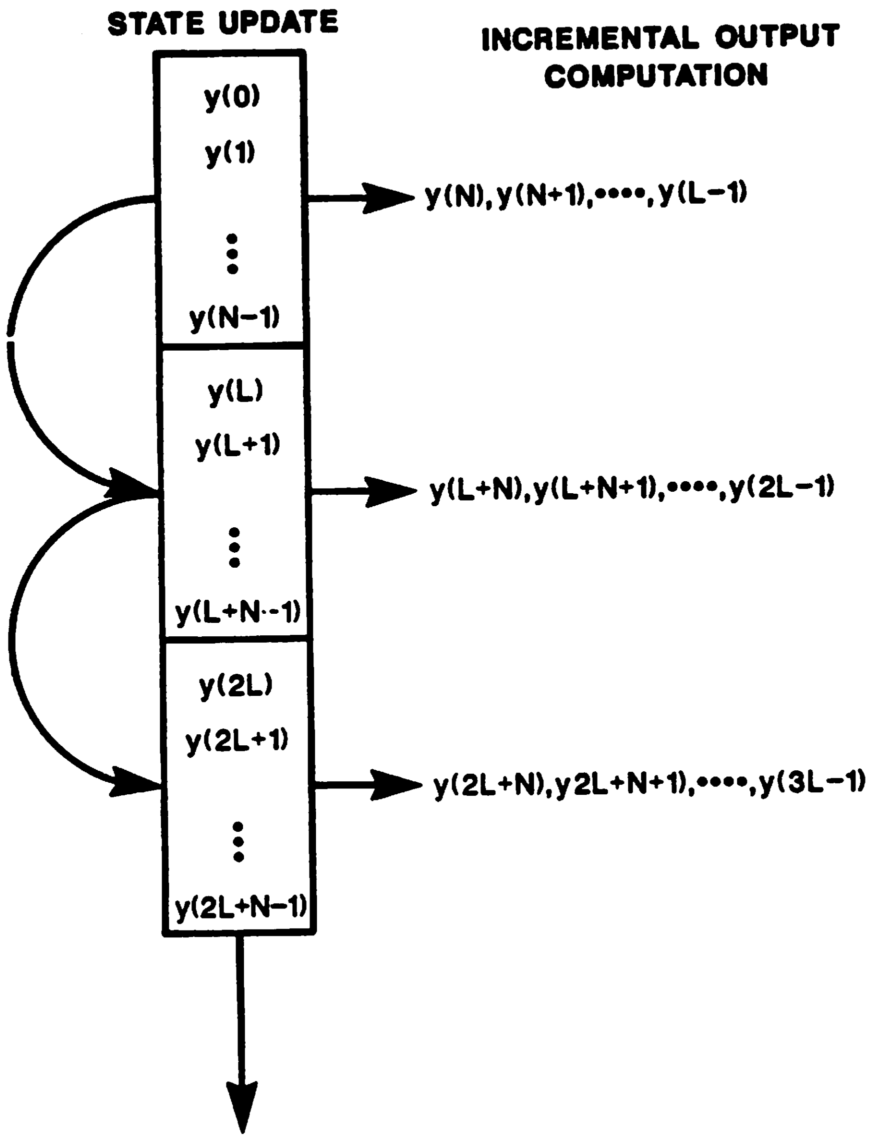

STATE UPDATE
INCREMENTAL OUTPUT COMPUTATION
y(0)
y(1)
y(N−1)
y(N), y(N+1),••••, y(L−1)
y(L)
y(L+1)
y(L+N·-1)
y(L+N),y(L+N+1),••••,y(2L−1)
y(2L)
y(2L+1)
y(2L+N−1)
y(2L+N),y2L+N+1),••••,y(3L−1)

Fig. 4.2: Incremental output computation in direct form block recursive digital filter.

Case I: $L \geq N$

We can update N out of the L outputs using (see appendix 4.1)

$$\mathbf{y}^{(N)}(kL+L) = \mathbf{A}(L)\mathbf{y}^{(N)}(kL) + \mathbf{B}(L)\mathbf{z}^{(L)}(kL+N) \tag{4.4a}$$

where $\mathbf{y}^{(N)}(i)$ is a $N\times 1$ column vector, $\mathbf{A}(L)$ is a $N\times N$ matrix, and $\mathbf{B}(L)$ is a $N\times L$ matrix. The elements of $\mathbf{A}(L)$ and $\mathbf{B}(L)$ are defined by (see appendix 4.1)

$$\left[\mathbf{A}(L)\right]_{ij} = \sum_{l=1}^{i} a_{N-l+1} r_{L-N+i-j+l-1}\, , \quad i, j = 1, 2, \ldots, N \tag{4.4b}$$

$$\left[\mathbf{B}(L)\right]_{ij} = r_{L-N+i-j}\, . \tag{4.4c}$$

The $(L-N)$ outputs $y(kL+N)$, $y(kL+N+1)$, ..., $y(kL+L-1)$ can be computed non-recursively in a sequential manner using the past outputs and the corresponding inputs.

The multiplication complexity of the above state update implementation (i.e. for updating N outputs or states) is $\left[LN + N^2 - \frac{N(N+1)}{2}\right]$, of which $(LN - \frac{N(N+1)}{2})$ is due to the $\mathbf{B}(L)$ matrix and N^2 is due to the $\mathbf{A}(L)$ matrix. The computation of $z(kL)$, $z(kL+1)$, ..., $z(kL+L-1)$ requires $(N+1)L$ multiplications, and the computation of the last $(L-N)$ outputs can be done using the past outputs with $N(L-N)$ multiplications. The total multiplication complexity of the direct form block filter is $\left[L(3N+1) - \frac{N(N+1)}{2}\right]$, which is linear in block size L. Another direct form block structure [15] has a multiplication complexity $\left[2LN + \frac{L(L+1)}{2}\right]$, which is square in block size.

Fig. 4.3(a) shows a second order direct form word-serial recursive digital filter, and Fig. 4.3(b) shows the corresponding block filter for a block size of 5. In this structure, the states $y(5k)$ and $y(5k+1)$ are updated each block, and the outputs $y(5k+2)$, $y(5k+3)$, and $y(5k+4)$ are computed incrementally in a non-recursive or sequential manner.

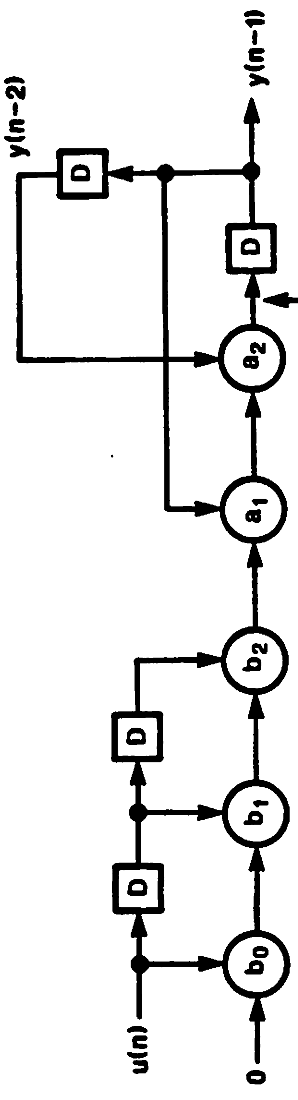


Fig. 4.3(a): A second order word-serial recursive filter.

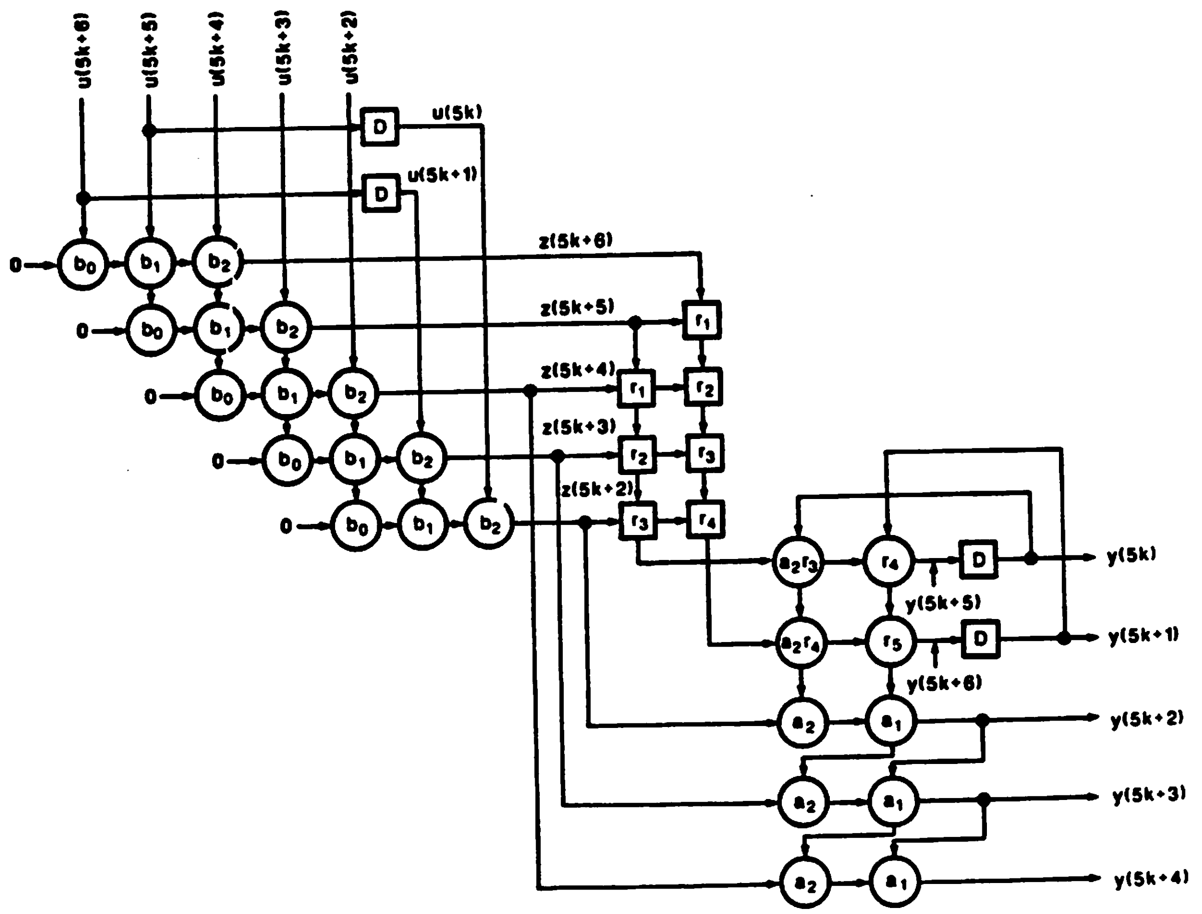


Fig. 4.3(b): Block implementation of the second order recursive digital filter for block size of 5.

The multiplication complexity of this second order filter is $(7L-3)$, and is same as that of the structure proposed in [13-14].

Case II: $L<N$

For the case when block size L is less than the filter order N, only L states need to be updated, each based on N past states using the clustered look-ahead approach, and the remaining $(N-L)$ states can be derived by delaying the available L states appropriately. For example if $L=3$ and $N=8$, then we can compute the outputs $y(3k+3)$, $y(3k+4)$, and $y(3k+5)$ using $y(3k)$, $y(3k+1)$, and $y(3k+2)$ and their delayed samples $y(3k-1)$, ..., $y(3k-5)$. Note that $y(3k-1)$ and $y(3k-4)$ can be obtained by delaying $y(3k+2)$, $y(3k-2)$ and $y(3k-5)$ can be derived by delaying $y(3k+1)$, and $y(3k-3)$ can be obtained by delaying $y(3k)$.

The direct form block filter can be described by (see appendix 4.1)

$$\mathbf{y}^{(L)}(kL+L) = \mathbf{A}(L)\mathbf{y}^{(N)}(kL+L-N) + \mathbf{B}(L)\mathbf{z}^{(L)}(kL+L) \tag{4.5a}$$

where

$$\left[\mathbf{A}(L)\right]_{ij} = \sum_{l=1}^{i} a_{N-l+1} r_{i-j+l-1}\, , \; i = 1,2,\ldots,L; j = 1,2,\ldots,N \tag{4.5b}$$

$$\left[\mathbf{B}(L)\right]_{ij} = r_{i-j}\, , \tag{4.5c}$$

and A(L) is $L\times N$, B(L) is $L\times L$.

The complexity required for computing $\mathbf{z}^{(L)}(kL+L)$ is $L(N+1)$, for computing $\mathbf{B}(L)\mathbf{z}^{(L)}(kL+L)$ is $\frac{L(L-1)}{2}$, and that due to $\mathbf{A}(L)$ is NL multiplications. Thus the total multiplication complexity is $(2N+\frac{L+1}{2})L$, which is square in block size. Fortunately, for this case L is small.

4.3. STATE SPACE BLOCK DIGITAL FILTERS

Consider the state space recursive filter described by

$$\mathbf{x}(n+1) = \mathbf{A}\mathbf{x}(n) + \mathbf{b}u(n) \tag{4.6a}$$

$$y(n) = \mathbf{c}^T\mathbf{x}(n) + du(n) \tag{4.6b}$$

where the state $\mathbf{x}(n)$ is $N\times1$, the state update matrix $\mathbf{A}$ is $N\times N$, $\mathbf{b}$ and $\mathbf{c}$ are $N\times1$, and d, input sample $u(n)$ and output sample $y(n)$ are scalars, and N is the order of the filter. Fig. 4.4(a) shows a block diagram corresponding to (4.6). For the purposes of this chapter, the state update matrix is assumed to be quasi-diagonal (i.e. all real poles of the system are of multiplicity less than or equal to two, and all complex poles are of multiplicity unity), and N_1 is assumed to represent the number of real poles of unity multiplicity. The quasi-diagonal state update matrix has N_1 blocks of dimension unity, and $\frac{(N-N_1)}{2}$ blocks of dimension 2x2. The total number of non-zero elements in $\mathbf{A}$ is $(2N-N_1)$, which is linear in filter order N.

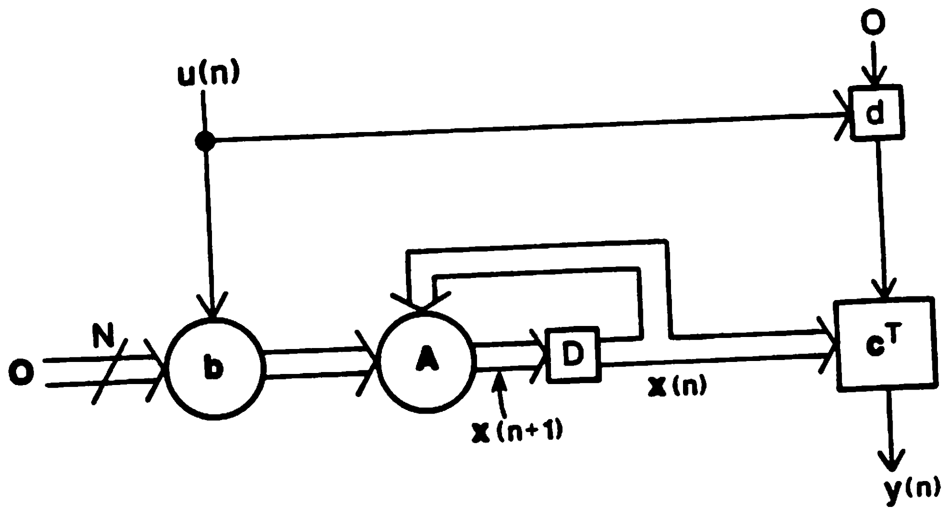


Fig. 4.4(a): A word-serial state space recursive filter.

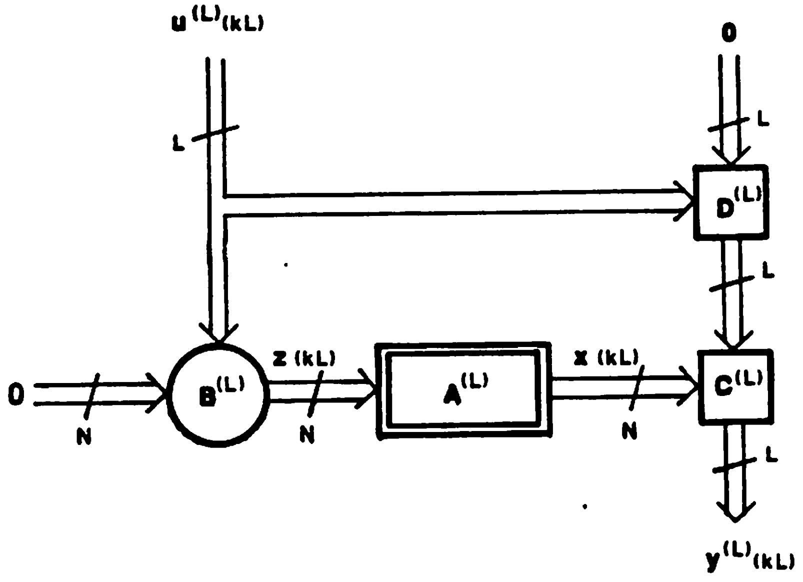


Fig. 4.4(b): Block-state space recursive digital filter.

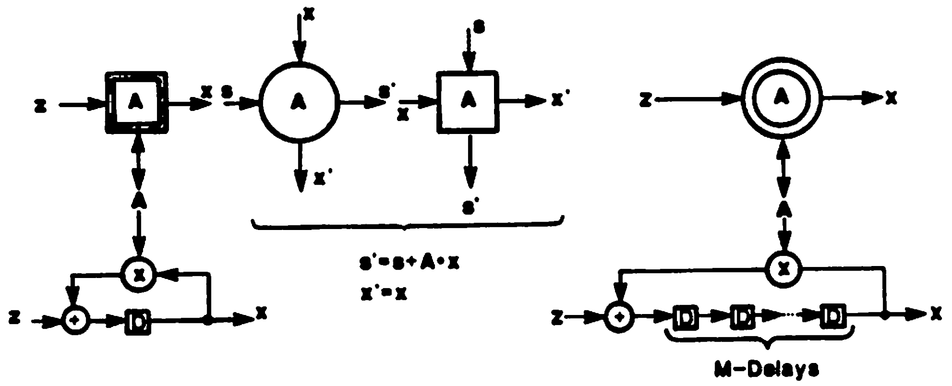


Fig. 4.4(c): Definition of processing elements.

Unlike the direct form representation, the state space representation consists of a state update portion, and an output computation portion. Similarly, the block state space filter representation also consists of a block state update portion and a block output computation portion. The block state update operation has a linear complexity in block size. The block output computation in the existing block-state and parallel block-state structures leads to a square complexity in block size. In this section, we review the existing block state [6] and the parallel block state filter structures [10]. Then we present the incremental output computation approach, and using this we derive the incremental block state filter of complexity linear in block size [11]. The average quantization noise at the output of the incremental block-state structure is same as that of the block-state structure and less than the parallel block-state structure. We also extend the incremental block state filter structure for the case of multirate recursive filtering.

4.3.1. Block-State Implementation

In the block structure with block size L, each implementable latch is L*−slow*, i.e. the clock rate of the latch in the block filter is L times slower than the input sample rate. Hence, the state of the system needs to be updated block by block, i.e. the state $\mathbf{x}(kL+L)$ is updated using $\mathbf{x}(kL)$, and the $(L-1)$ intermediate states $\mathbf{x}(kL+1)$, ..., $\mathbf{x}(kL+L-1)$ are missed in the block state update process. The state update representation of the block-state structure [6] can be derived by iterating the single-input-single-output (SISO) state update equation (4.6a) $(L-1)$ times, and is given by

$$\mathbf{x}((k+1)L) = \mathbf{A}^{(L)}\mathbf{x}(kL) + \mathbf{B}^{(L)}\mathbf{u}^{(L)}(kL) \tag{4.7a}$$

where

$$\mathbf{A}^{(L)} = \mathbf{A}^{L} \tag{4.7b}$$

$$\mathbf{B}^{(L)} = \left[\mathbf{A}^{L-1}\mathbf{b} \;\; \mathbf{A}^{L-2}\mathbf{b} \;\; \ldots \;\; \mathbf{b}\right] \tag{4.7c}$$

$$\mathbf{u}^{(L)}(n) = \left[u(n) \;\; u(n+1) \;\; \ldots \;\; u(n+L-1)\right]^T \tag{4.7d}$$

and $\mathbf{A}^{(L)}$ is $N\times N$, $\mathbf{B}^{(L)}$ is $N\times L$, $\mathbf{u}^{(L)}(kL)$ is $L\times 1$.

In the block-state structure, the block of outputs $y(kL)$, $y(kL+1)$, ..., and $y(kL+L-1)$ are computed based on the single state $\mathbf{x}(kL)$ and the corresponding inputs. This is based on the assumption that the $(L-1)$ intermediate states are not available (since they are missed due to the block state update process). The block output equation is described by

$$\mathbf{y}^{(L)}(kL) = \mathbf{C}^{(L)}\mathbf{x}(kL) + \mathbf{D}^{(L)}\mathbf{u}^{(L)}(kL) \tag{4.7e}$$

where

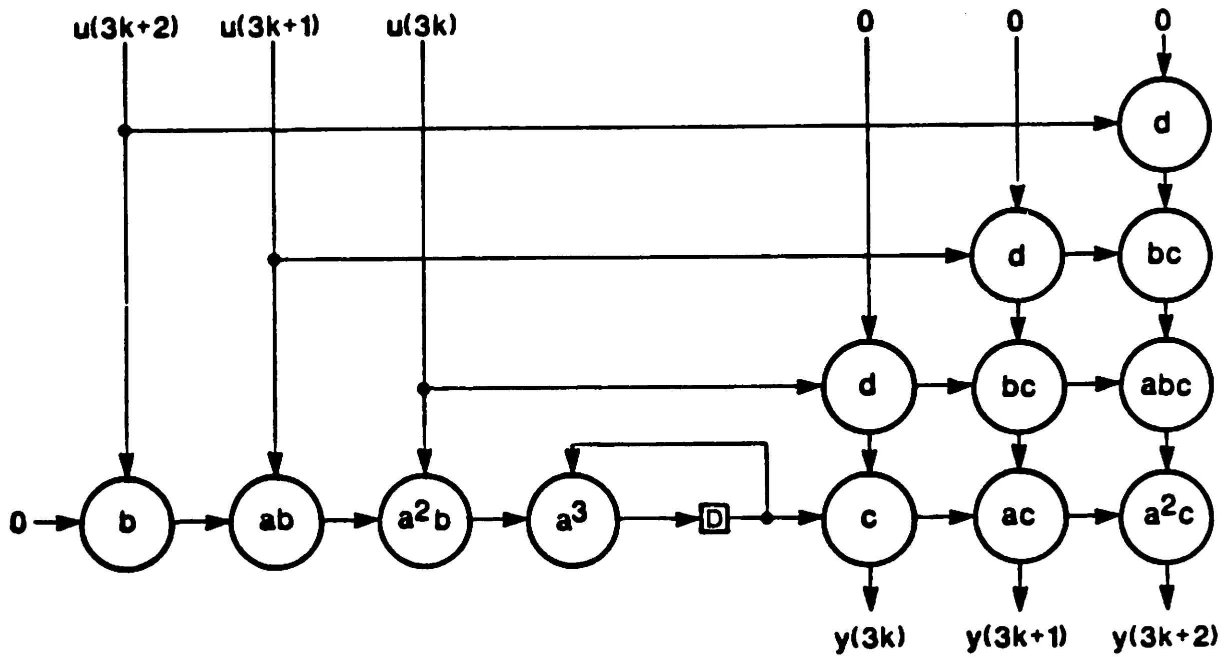


Fig. 4.5: Block state implementation of a first order state space recursive filter for block size of 3.

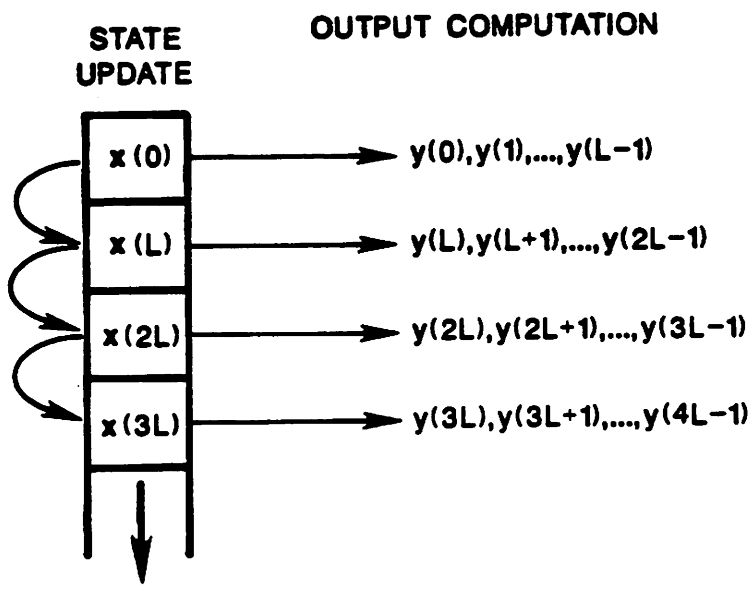


Fig. 4.6: Partial schedule of a block state space implementation.

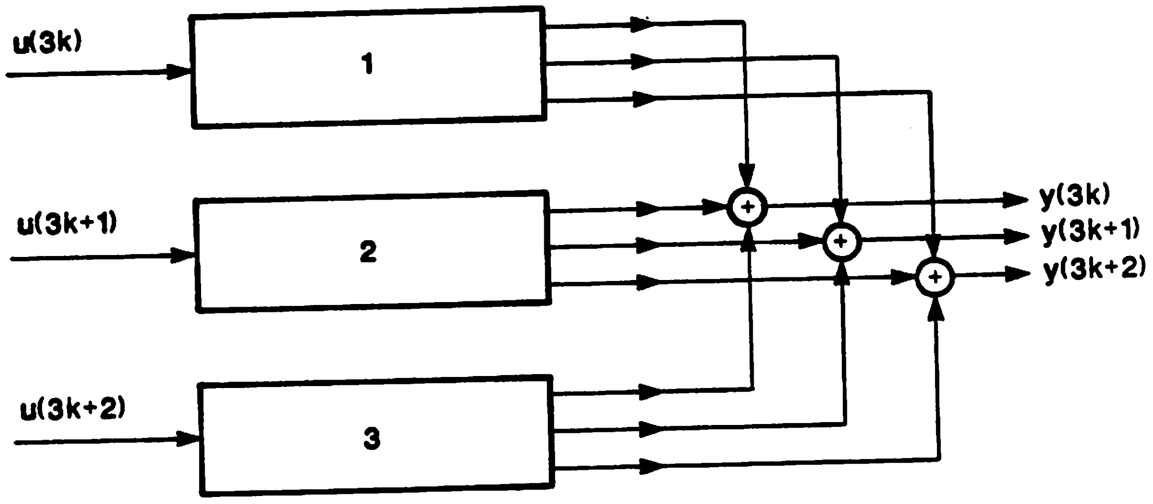


Fig. 4.7(a): A parallel block state implementation for block size 3.

$$\mathbf{C}^{(L)} = \left[\mathbf{c}^T \ \mathbf{c}^T\mathbf{A} \ \cdots \ \mathbf{c}^T\mathbf{A}^{(L-1)}\right]^T \tag{4.7f}$$

$$\left[\mathbf{D}^{(L)}\right]_{ij} = \begin{cases} 0 & i<j \\ d & i=j \\ \mathbf{c}^T\mathbf{A}^{(i-j-1)}\mathbf{b} & i>j \end{cases} \tag{4.7g}$$

and

$$\mathbf{y}^{(L)}(n) = \left[y(n) \ y(n+1) \ .... \ y(n+L-1)\right]^T . \tag{4.7h}$$

In the output equation, $\mathbf{C}^{(L)}$ is $L \times N$, $\mathbf{D}^{(L)}$ is $L \times L$ and lower triangular, and $\mathbf{y}^{(L)}$ is $L \times 1$. The block diagram of the block-state filter is shown in Fig. 4.4(b). The blocks marked $\mathbf{B}^{(L)}$, $\mathbf{C}^{(L)}$, $\mathbf{D}^{(L)}$ represent matrix vector multipliers, and the block marked $\mathbf{A}^{(L)}$ represents the state update network (see Fig. 4.4(c) for definition of the processing elements). Fig. 4.5 shows the block state implementation of a first order recursive filter for a block size of three. In the block state filter, we use $\mathbf{x}(0)$ to compute the block of outputs $\mathbf{y}^{(L)}(0)$, and to update $\mathbf{x}(L)$ (see partial schedule in Fig. 4.6). In the next cycle, $\mathbf{x}(L)$ is used to compute next block of outputs $\mathbf{y}^{(L)}(L)$, and to update the state $\mathbf{x}(2L)$, and the schedule repeats itself.

For the case of a quasi-diagonal state update matrix, the complexity (in terms of multiplications) of the state update representation is $(NL+2N-N_1)$, and the output representation is $(NL+\frac{L(L+1)}{2})$ where N, L, and N_1 respectively represent the order of the system, block size, and number of real poles with unity multiplicity. The total multiplication complexity of the block-state structure (C_b) is given by

$$C_b = 2N(L+1) + \frac{L(L+1)}{2} - N_1 , \tag{4.8}$$

and is $O(L^2)$ for a block size of L. The asymptotic complexity per output sample is $(2N+\frac{L+1}{2})$.

4.3.2. Parallel Block-State Implementation

In parallel block-state structure [10], the *system state* $\mathbf{x}(n)$ is first decomposed into L *section states* $\mathbf{x}_1(n), \mathbf{x}_2(n), \cdots, \mathbf{x}_L(n)$, which are related by

$$\mathbf{x}(n) = \begin{bmatrix} \mathbf{A}^{L-1} \, \mathbf{A}^{L-2} \ldots 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1(n) \\ \mathbf{x}_2(n) \\ \vdots \\ \mathbf{x}_L(n) \end{bmatrix} . \tag{4.9a}$$

Since each implementable latch of the block structure is L*–slow*, the states in each section are updated block by block, i.e. in section i the state $\mathbf{x}_i(kL+L)$ is computed based on the state $\mathbf{x}_i(kL)$. Using each section state, L *partial outputs* are computed and the L *system outputs* are obtained by adding the corresponding L partial outputs (see Fig. 4.7(a)). Substituting (4.9a) in (4.7a) and (4.7e), the state update and output representations of the parallel block-state structure can be derived to be [10]

$$\mathbf{x}_i(kL+L) = \mathbf{A}^L \mathbf{x}_i(kL) + \mathbf{b}u(kL+i-1), \; i=1,2, \ldots, L \tag{4.9b}$$

$$\begin{bmatrix} y(kL) \\ y(kL+1) \\ \vdots \\ y(kL+L-1) \end{bmatrix} = \begin{bmatrix} \mathbf{c}^T\mathbf{A}^{L-1} & \mathbf{c}^T\mathbf{A}^{L-2} & \cdots & \mathbf{c}^T \\ \mathbf{c}^T\mathbf{A}^{L} & \mathbf{c}^T\mathbf{A}^{L-1} & \cdots & \mathbf{c}^T\mathbf{A} \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{c}^T\mathbf{A}^{2L-2} & \mathbf{c}^T\mathbf{A}^{2L-3} & \cdots & \mathbf{c}^T\mathbf{A}^{L-1} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1(kL) \\ \mathbf{x}_2(kL) \\ \vdots \\ \mathbf{x}_L(kL) \end{bmatrix} + \begin{bmatrix} d & 0 & \cdots & 0 \\ \mathbf{c}^T\mathbf{b} & d & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{c}^T\mathbf{A}^{L-2}\mathbf{b} & \mathbf{c}^T\mathbf{A}^{L-3}\mathbf{b} & \cdots & d \end{bmatrix} \begin{bmatrix} u(kL) \\ u(kL+1) \\ \vdots \\ u(kL+L-1) \end{bmatrix} . \tag{4.9c}$$

For the case of a quasi-diagonal state update matrix, the state update complexity is $L(3N-N_1)$ and the output computation complexity is $(NL^2 + \frac{L(L+1)}{2})$, leading to a total multiplication complexity of

$$C_p = NL(L+3) + \frac{L(L+1)}{2} - N_1L . \tag{4.10}$$

The asymptotic complexity of the parallel block-state structure is $O(NL^2)$, which is N

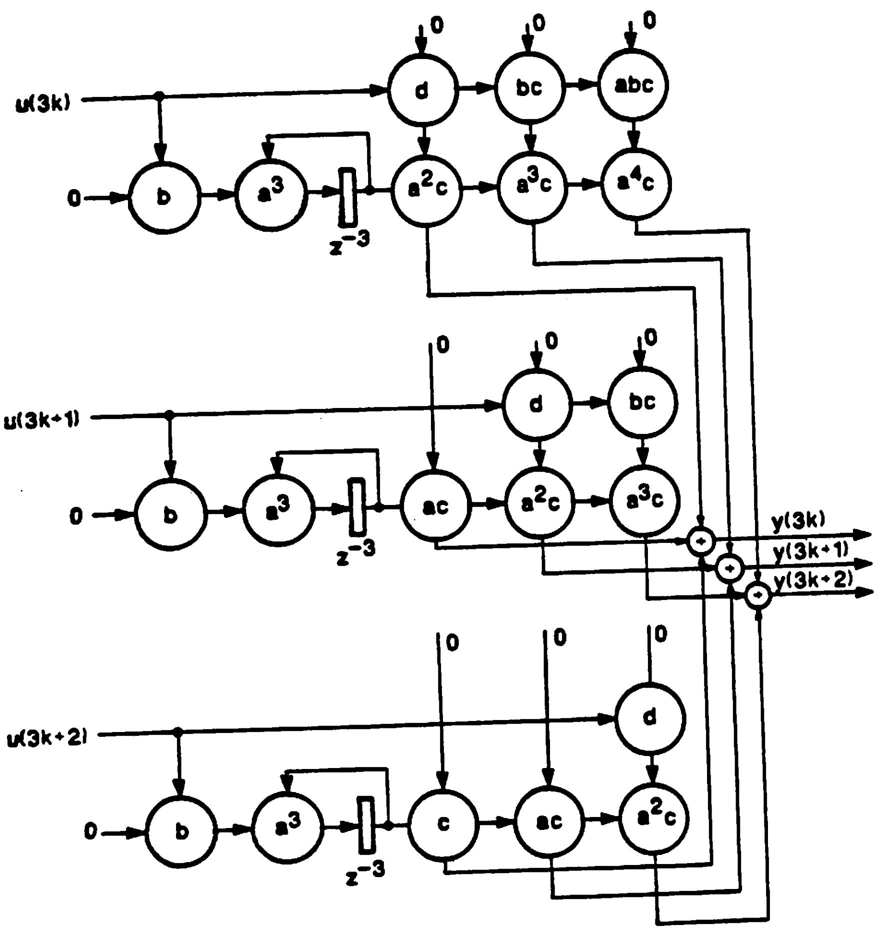


Fig. 4.7(b): Parallel block implementation of a first order recursive filter.

times higher than that of the block-state structure. The complexity per output sample is $(N(L+3)+\frac{L+1}{2}N_1)$. The decomposition of the system state leads to this higher complexity. Fig. 4.7(b) shows the parallel block implementation of a first order recursive filter for a block size of 3. (Note that in [10], the author represented the multiplication complexity of the parallel block-state structure to be much less than that of the block-state structure. The error resulted from not including a factor of L in the complexity expression.)

4.3.3. Incremental Block-State Implementation

In the incremental block-state structure, the state $\mathbf{x}(kL+L)$ is updated based on the state $\mathbf{x}(kL)$ as in the block-state case. However, the output computation, which is non-recursive, is done in a different way. Rather than calculating the block of outputs in terms of the updated state every L samples, we first calculate the intermediate states (which were missed due to the block state update process) every I samples nonrecursively (where I is the increment size), and then calculate the outputs incrementally in a sequential manner using these intermediate states. It is this novel output computation which leads to an $O(L)$ complexity in the incremental block-state structure for a block size of L.

In the incremental block-state structure with increment I, I outputs $y(kL+pI)$, $y(kL+pI+1)$, ..., $y(kL+pI+I-1)$ are computed using the state $\mathbf{x}(kL+pI)$ and the inputs $u(kL+pI)$, $u(kL+pI+1)$, ..., $u(kL+pI+I-1)$ respectively, and the intermediate state $\mathbf{x}(kL+pI+I)$ is non-recursively computed using $\mathbf{x}(kL+pI)$ for computation of next I outputs. Thus using $\mathbf{x}(kL)$, we can compute $y(kL)$, ..., $y(kL+I-1)$ and can non-recursively compute $\mathbf{x}(kL+I)$ for computation of next I outputs. The size of the increment I is

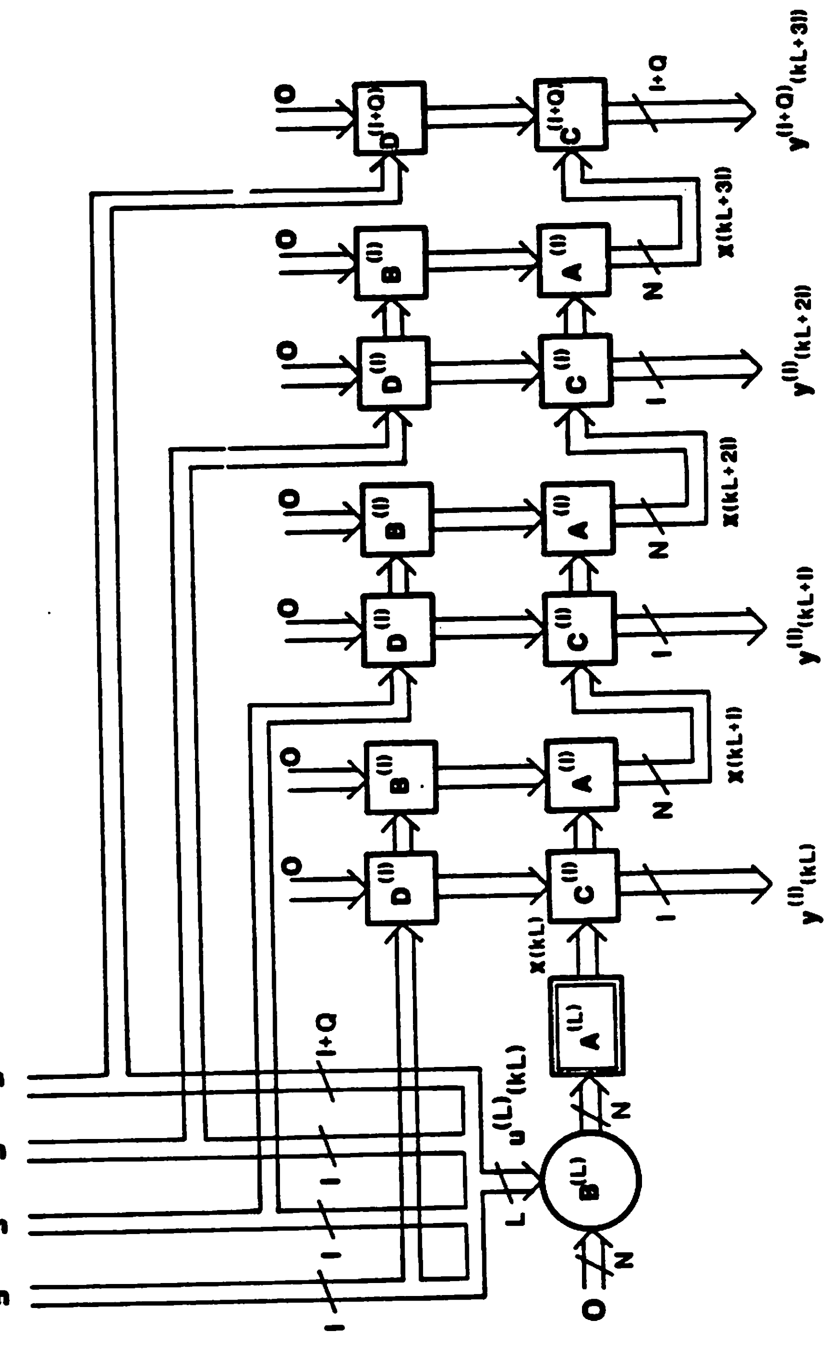


Fig. 4.8: Incremental block state implementation of a state space filter ($L = 4I + Q$).

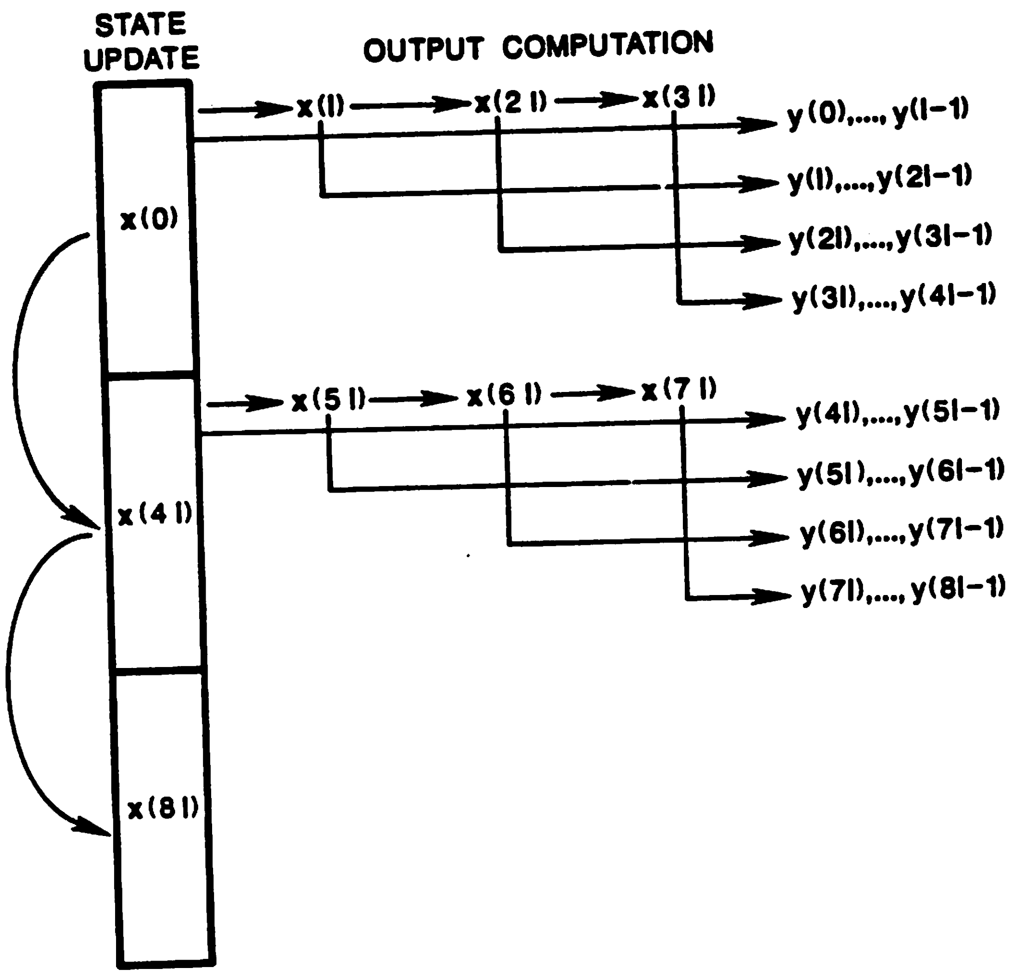


Fig. 4.9: Partial schedule of an incremental block state digital filter ($L = 4l$).

chosen to minimize the multiplication complexity. Let the block size L correspond to $(PI+Q)$, where I is the increment, and P and Q respectively represent the quotient and remainder of $\frac{L}{I}$. Then the size of the last increment is $(I+Q)$, i.e. the computation of the last $(I+Q)$ outputs is performed based on the state $\mathbf{x}(kL+PI-I)$.

The state update representation of the *incremental block-state structure* is identical to that of the block-state structure and is described by (4.7a). The computation of I outputs $y(kL+pI)$, ..., $y(kL+pI+I-1)$ and the intermediate state $\mathbf{x}(kL+pI+I)$ is performed using

$$\begin{bmatrix} \mathbf{y}^{(I)}(kL+pI) \\ \mathbf{x}(kL+pI+I) \end{bmatrix} = \begin{bmatrix} \mathbf{C}^{(I)} & \mathbf{D}^{(I)} \\ \mathbf{A}^{(I)} & \mathbf{B}^{(I)} \end{bmatrix} \begin{bmatrix} \mathbf{x}(kL+pI) \\ \mathbf{u}^{(I)}(kL+pI) \end{bmatrix}, \; p=0,1,...,P-2 . \tag{4.11a}$$

The computation of the last $(I+Q)$ outputs is carried out using

$$\mathbf{y}^{(I+Q)}(kL+PI-I) = \mathbf{C}^{(I+Q)}\mathbf{x}(kL+PI-I) + \mathbf{D}^{(I+Q)}\mathbf{u}^{(I+Q)}(kL+PI-I) , \tag{4.11b}$$

where the meaning of the symbols stand as defined in (4.7). A block diagram of the incremental block-state structure is shown in Fig. 4.8, and its partial schedule is shown in Fig. 4.9 for $L=4I$. Fig. 4.10(a) and 4.10(b) show the incremental block state implementation of a first order state space recursive digital filter for a block size of four for increment sizes one and two respectively.

A family of block structures can be described by the incremental block-state structure for different values of the increment I. A value of $I \geq \frac{L}{2}$ (i.e. $P=1$) leads to the block-state structure described in section 4.3.1. An optimum value of the increment I can be derived to minimize the multiplication complexity of the incremental block-state structure. The optimum increment will depend upon the exact custom VLSI implementation architecture, or scheduling in case of a software-programmable parallel processor

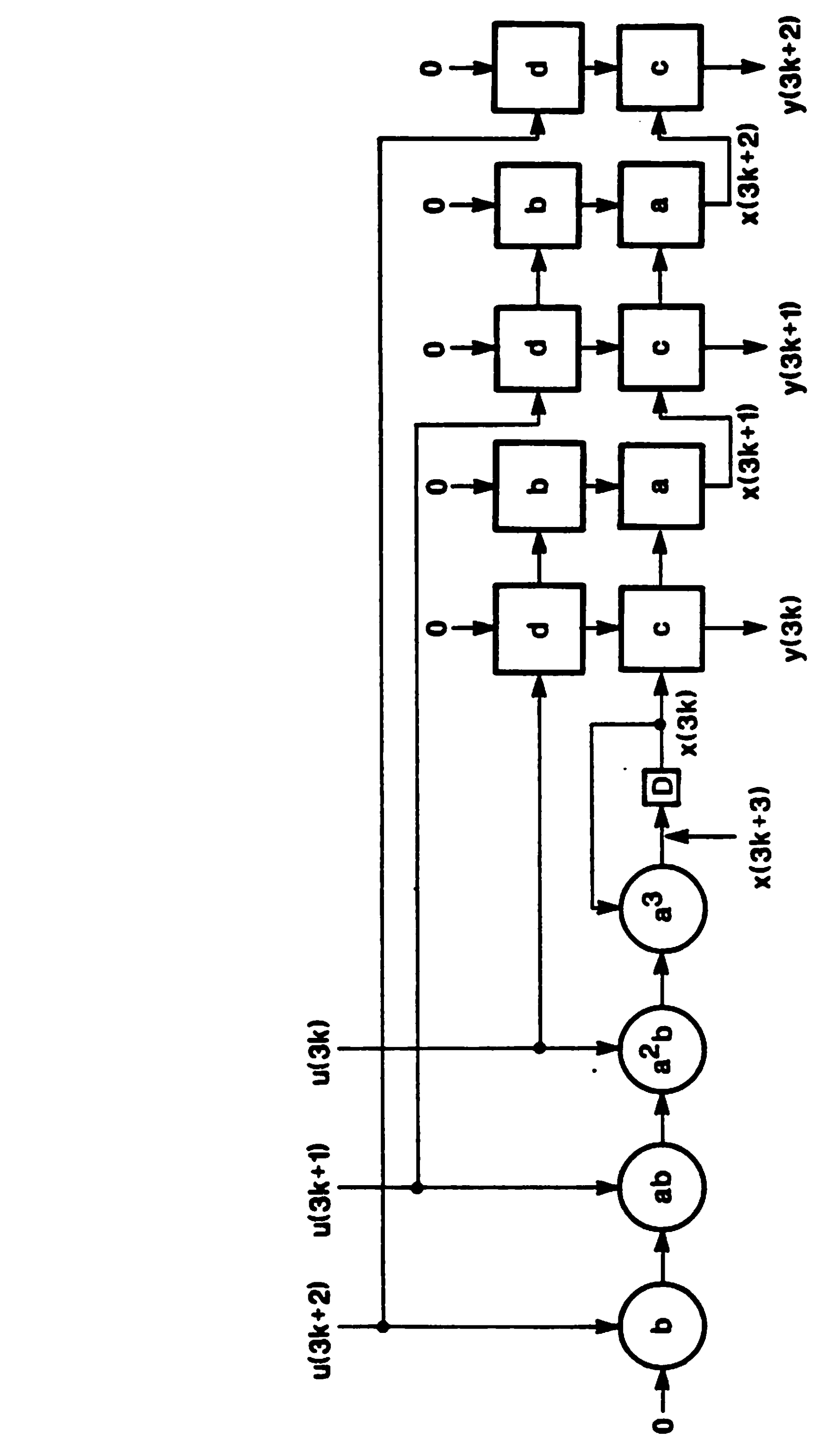


Fig. 4.10(a): Incremental Block-State implementation of a first order recursive system for block size of four, and an increment size of one.

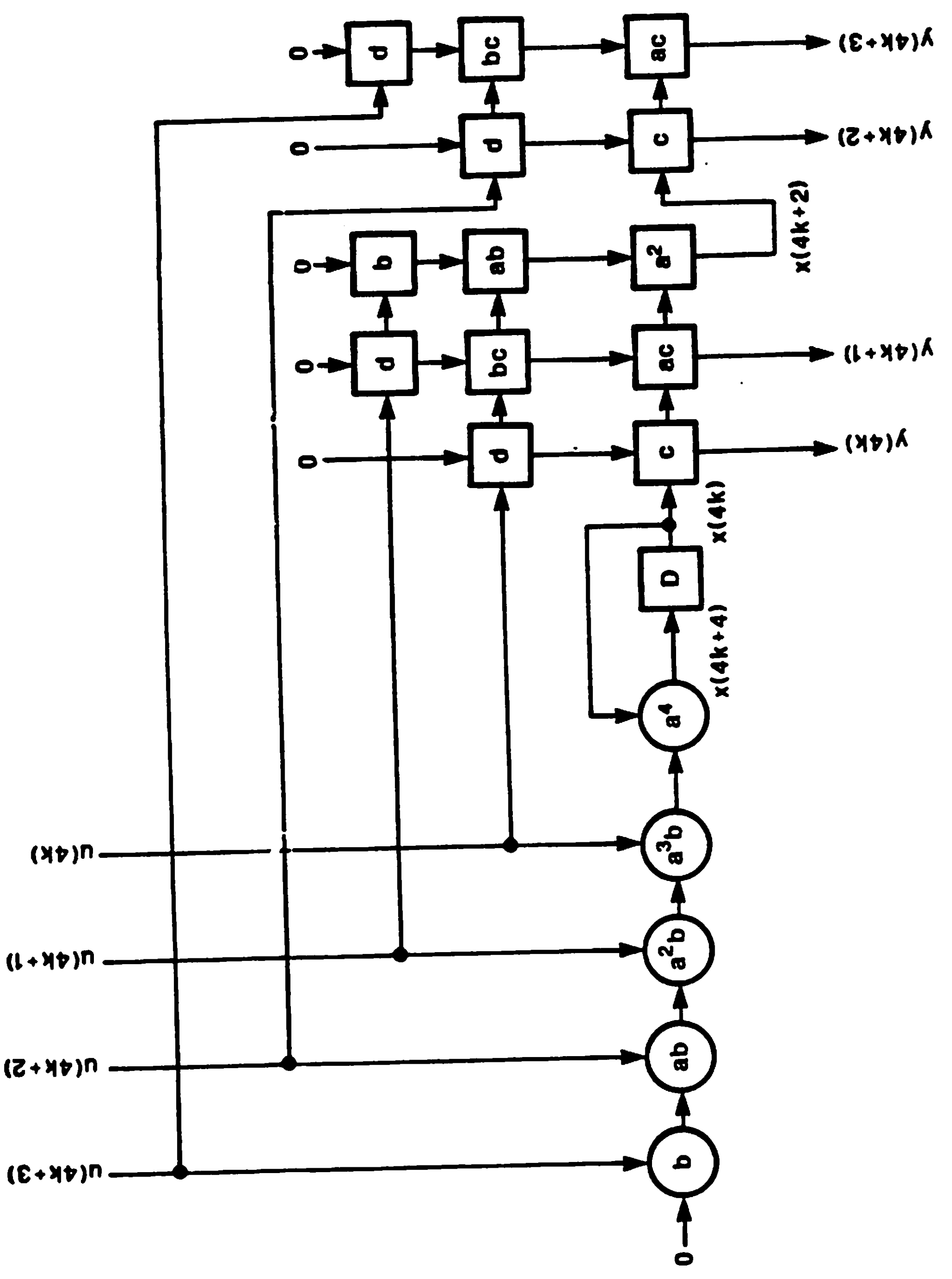


Fig. 4.10(b): Incremental Block-State implementation of a first order recursive system for block size of four, and an increment size of two.

realization. For example, if it is not possible to exploit the lower triangular nature of the **D** matrix, then the complexity of the full matrix will need to be accounted for. In the sequel, we assume that it is possible to exploit the lower triangular nature of **D** matrix, and consider the case of the quasi-diagonal state update matrix. A similar analysis can be carried out for all other cases.

In an incremental block-state realization, the complexity of the state update equation is $(2N-N_1+NL)$, and is independent of the increment I. The output computation complexity for first $(P-1)$ increments (of size I each) is $(P-1)(2IN+2N-N_1+\frac{I(I+1)}{2})$, and the last increment (of size $(I+Q)$) is $((I+Q)N+\frac{(I+Q)(I+Q+1)}{2})$. The total complexity of the incremental block-state structure is given by

$$C_i = 2N(P+L)+\frac{L(I+1)}{2}+\frac{Q(I+Q)}{2}+NI(P-1)-N_1P\ , \tag{4.12}$$

where P and Q are respectively quotient and remainder of $\frac{L}{I}$. In the asymptotic case, we need to minimize the complexity per output in the output equation which is given by

$$C'_{io} = 2N+\frac{2N-N_1}{I}+\frac{I+1}{2}\ , \tag{4.13}$$

and is minimized for

$$I = [\![\sqrt{2(2N-N_1)}]\!]\ , \tag{4.14}$$

where $[\![x]\!]$ represents the integer nearest to x. At the optimum increment value, the multiplication complexity associated with the computation of the intermediate state and that of the I outputs are approximately the same. In the asymptotic case (i.e. very large L and $Q=0$), the complexity of the incremental block-state structure with optimum I (for a quasi-diagonal state update matrix) is given by

$$C_i = L\left[3N + \frac{2(2N-N_1) + [\sqrt{2(2N-N_1)}]^2}{2[\sqrt{2(2N-N_1)}]} + \frac{1}{2}\right] - N[\sqrt{2(2N-N_1)}], \qquad (4.15)$$

and is linear in block size. For the special case of the incremental block-state structure with unity increment ($I = 1$), the complexity per output sample is $(5N - N_1 + 1)$.

Table 4.2 summarizes the asymptotic complexity (in terms of number of multiplications) per output sample for various structures for the case of a quasi-diagonal state update matrix.

Table 4.2: Asymptotic Implementation Complexity per Output Sample

Implementation	Number of Multiplications per Sample
Direct	$2N + 1$
SISO State Space	$4N - N_1 + 1$
Block-State	$2N + \frac{(L+1)}{2}$
Parallel Block-State	$3N + NL + \frac{(L+1)}{2} - N_1$
Incremental Block-State	$3N + \frac{2(2N-N_1) + [\sqrt{2(2N-N_1)}]^2}{2[\sqrt{2(2N-N_1)}]} + \frac{1}{2}$

Table 4.3 summarizes the multiplication complexity (Comp) in terms of number of multiplications for typical filter orders (N) and block sizes (L) for block-state (BS), parallel block-state (PBS), and incremental block-state (IBS) structure with optimum increment (I) for $N_1 = 0$.

Table 4.3: Complexity Comparison for Different Block Structures

N	L = 20				L = 40				L = 60			
	BS	PBS	IBS		BS	PBS	IBS		BS	PBS	IBS	
	Comp	Comp	Comp	I	Comp	Comp	Comp	I	Comp	Comp	Comp	I
2	294	1130	179	3	984	4260	366	3	2074	9390	554	3
4	378	2050	310	6	1148	7700	644	4	2318	16950	974	4
6	462	2970	425	7	1312	11140	892	6	2562	24510	1372	7
8	546	3890	527	7	1476	14580	1134	7	2806	32070	1742	7

For large block sizes, it is possible to achieve linear speedup (in execution time) in case of a software-programmable parallel processor implementation (using the incremental block-state recursive digital filter) as the number of processors increases. In the case of a custom VLSI realization, a linear increase in sampling rate can be achieved by using block processing at the expense of a linearly proportional increase in hardware.

Since the recursive state update for block-state and incremental block-state structures are identical, the average roundoff noise at the output is the same for these two structures (under the assumption that the roundoff is performed at the output of state variables and at the system outputs, and the noise sources are white stationary with zero mean and statistically independent). The roundoff noise for the incremental block filter is derived in appendix 4.2. It has been shown in [10] that the average roundoff noise at the output of the parallel block-state structure is greater than that of the block-state structure (and hence than the incremental block-state structure).

4.3.4. Multi-Rate Incremental Block-State Filter

In this section, we study the incremental block state realization of multi-rate recursive digital filters. Multi-rate block recursive filtering based on the block-state structure has been studied in [19].

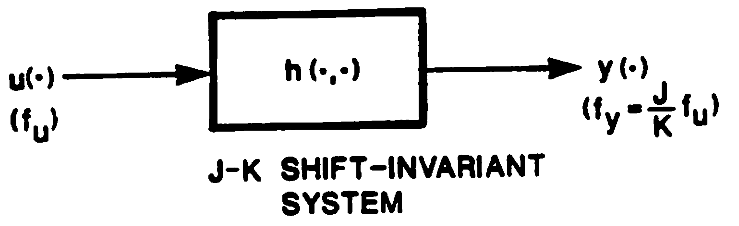


Fig. 4.11(a): A multirate recursive digital filter.

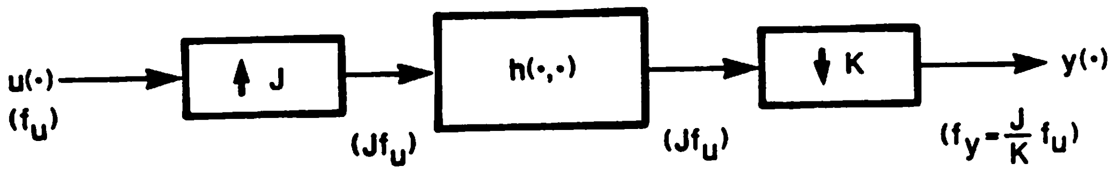


Fig. 4.11(b): An equivalent representation the filter in (a).

Any multirate system with sampling rate alteration by a factor of $\frac{J}{K}$ can be realized using a 1-to-J interpolator at the input, and a K-to-1 decimator at the output (where the greatest common divisor of J and K is unity) (see Fig. 4.11(a) and 11(b)). Thus, only one out of J inputs is non-zero and only one out of K outputs needs to be computed. We assume each J-th input to be non-zero and the 1-st (of each of the K) outputs to be computed to obtain minimum complexity. Thus for an input (output) block size of KL (JL), KL non-zero inputs $u(pKJL+J-1)$, $u(pKJL+2J-1)$, ..., $u(pKJL+KLJ-1)$ are processed to generate JL non-zero outputs $y(pKJL)$, $y(pKJL+K)$, ..., $y(pKJL+(JL-1)K)$ (see Fig. 4.11(c)). In the incremental block-state structure with an input (output) increment KI (JI), JI outputs $y(pKJL+qKJI)$, $y(pKJL+qKJI+K)$, ..., $y(pKJL+qKJI+(JI-1)K)$ are computed using the state $\mathbf{x}(pKJL+qKJI)$, and KI non-zero inputs $u(pKJL+qKJI+J-1)$, $u(pKJL+qKJI+2J-1)$, ..., $u(pKJL+qKJI+KIJ-1)$. The state $\mathbf{x}(pKJL+(q+1)KJI)$ is then computed non-recursively using the state $\mathbf{x}(pKJL+qKJI)$ and the KI non-zero inputs (to be used for the computation of the next output increment). The size of the first $(P-1)$ output increments is JI, and that of the last is $J(I+Q)$ where P and Q are respectively the quotient and remainder of $\frac{L}{I}$.

The state update equation of the incremental block-state recursive multirate filter is described by

$$\mathbf{x}((p+1)KJL) = \mathbf{A}^{(KJL)}\mathbf{x}(pKJL) + \mathbf{B}^{(KJL)}\mathbf{u}^{(KJL)}(pKJL) \tag{4.16}$$

and has a multiplication complexity of $(2N-N_1+KLN)$ (since number of non-zero inputs is only KL).

The computation of J outputs $y(pKJL+qKJI+rKJ)$, $y(pKJL+qKJI+rKJ+K)$, ..., $y(pKJL+qKJI+rKJ+KJ-K)$ of the $(q+1)$-th increment can be described by

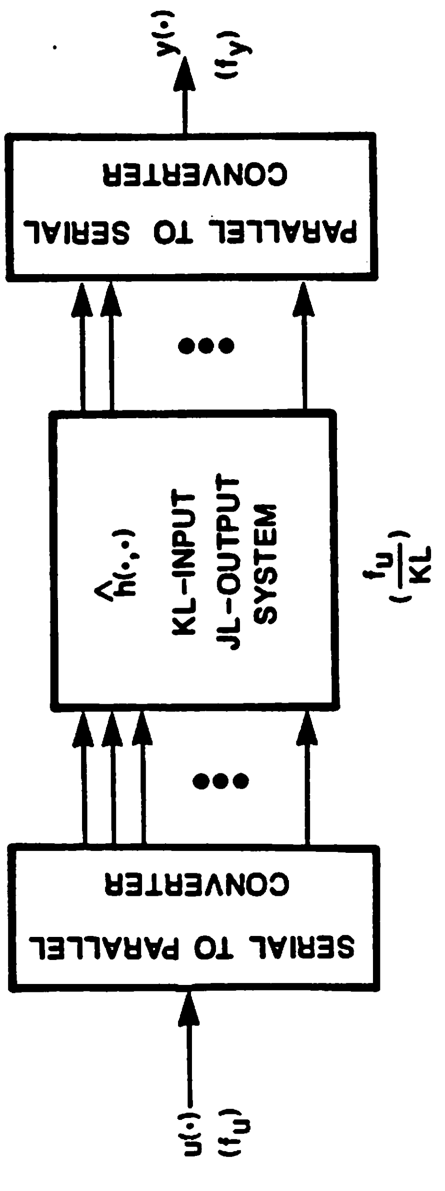


Fig. 4.11(c): Block implementation of the multirate recursive digital filter.

$$\begin{bmatrix} y(pKJL+qKJI+rKJ) \\ y(pKJL+qKJI+rKJ+K) \\ \vdots \\ y(pKJL+qKJI+rKJ+KJ-K) \end{bmatrix} = \begin{bmatrix} \mathbf{c}^T\mathbf{A}^{rKJ} \\ \mathbf{c}^T\mathbf{A}^{rKJ+K} \\ \vdots \\ \mathbf{c}^T\mathbf{A}^{rKJ+KJ-K} \end{bmatrix} \mathbf{x}(pKJL+qKJI) +$$

$$\begin{bmatrix} \mathbf{c}^T\mathbf{A}^{(rK-1)J}\mathbf{b} & \mathbf{c}^T\mathbf{A}^{(rK-2)J}\mathbf{b} & \cdots \\ \mathbf{c}^T\mathbf{A}^{(rK-1)J+K}\mathbf{b} & \mathbf{c}^T\mathbf{A}^{(rK-2)J+K}\mathbf{b} & \cdots \\ \vdots & \vdots & \cdots \\ \mathbf{c}^T\mathbf{A}^{(rK-1)J+(J-1)K} & \mathbf{c}^T\mathbf{A}^{(rK-2)J+(J-1)K} & \cdots \end{bmatrix} \begin{bmatrix} u(pKJL+qKJI+J-1) \\ u(pKJL+qKJI+2J-1) \\ \vdots \\ u(pKJL+qKJI+rKJ+KJ-1) \end{bmatrix}$$

$$q=0,1,\ldots,(P-2);r=0,1,\ldots,(I-1). \tag{4.17a}$$

The complexity associated with the computation of above J outputs is $(JN+rKJ+\Delta)$ multiplications, where

$$\Delta=\lfloor \frac{1}{J} \rfloor + \lfloor \frac{K+1}{J} \rfloor + \ldots + \lfloor \frac{(J-1)K+1}{J} \rfloor \tag{4.17b}$$

and $\lfloor x \rfloor$ represents the floor function of x. The complexity associated with computation of JI outputs (i.e. for r = 0-to-$(I-1)$) is $(JIN+I\Delta+\frac{JKI(I-1)}{2})$ multiplications. A partial schedule of a multirate incremental block recursive digital filter is shown in Fig. 4.12 (for $L=4I$).

The non-recursive state update equation for the $(q+1)$-th increment is described by

$$\mathbf{x}(pKJL+(q+1)KJI) = \mathbf{A}^{(KJI)}\mathbf{x}(pKJL+qKJI) + \mathbf{B}^{(KJI)}\mathbf{u}^{(KJI)}(pKJL+qKJI), \tag{4.18}$$

and leads to a complexity of $(2N-N_1+KIN)$ multiplications (since only KI inputs are non-zero). Thus the complexity associated with each (except last) increment (of input size KI and output size JI) is given by

$$JIC'_{io} = I\Delta + \frac{JKI(I-1)}{2} + JIN + (2N-N_1) + KIN, \tag{4.19}$$

where C'_{io} represents the complexity per output sample in the output equation of the incremental block structure. The above complexity is minimized for

$$I = \left\lceil \sqrt{\frac{2(2N-N_1)}{KJ}} \right\rceil . \tag{4.20}$$

With the above optimum increment, the asymptotic complexity per output sample (in terms of number of multiplications) is approximately given by

$$C_{io} = \frac{2KN}{J} + N + \frac{KJ\left\lceil \sqrt{\frac{2(2N-N_1)}{KJ}} \right\rceil^2 + 2(2N-N_1)}{2J\left\lceil \sqrt{\frac{2(2N-N_1)}{KJ}} \right\rceil} - \frac{K}{2} + \frac{\Delta}{J} , \tag{4.21}$$

and is independent of block size. Note that the asymptotic complexity per output sample of the block-state recursive filter in [19] is $\left[\frac{KN}{J} + N + \frac{KL}{2} - \frac{K}{2} + \frac{\Delta}{J} \right]$, which is linearly proportional to block size, and much larger than that of the incremental block-state structure presented in this chapter.

Interpolation and decimation by integer factors are special cases of the general multi-rate filtering case. A sampling rate increase (interpolation) by factor J can be obtained with unity K, and a sampling rate decrease (decimation) by factor K can be obtained with unity J. For both these cases, the value of Δ is unity.

4.4. DIRECT FORM PIPELINED BLOCK RECURSIVE FILTERS

We can get a speedup by a factor of LM by using a block size L and M stages of pipelining inside the recursive loop [16-18]. The pipelined block state update operations for the cases when the block size (L) is greater than the filter order (N) and less than the filter order need to be studied separately. We assume M to be a power of 2 to exploit the decomposition of the non-recursive portion in an efficient manner. We first consider the case $L \geq N$ and then the case $L < N$.

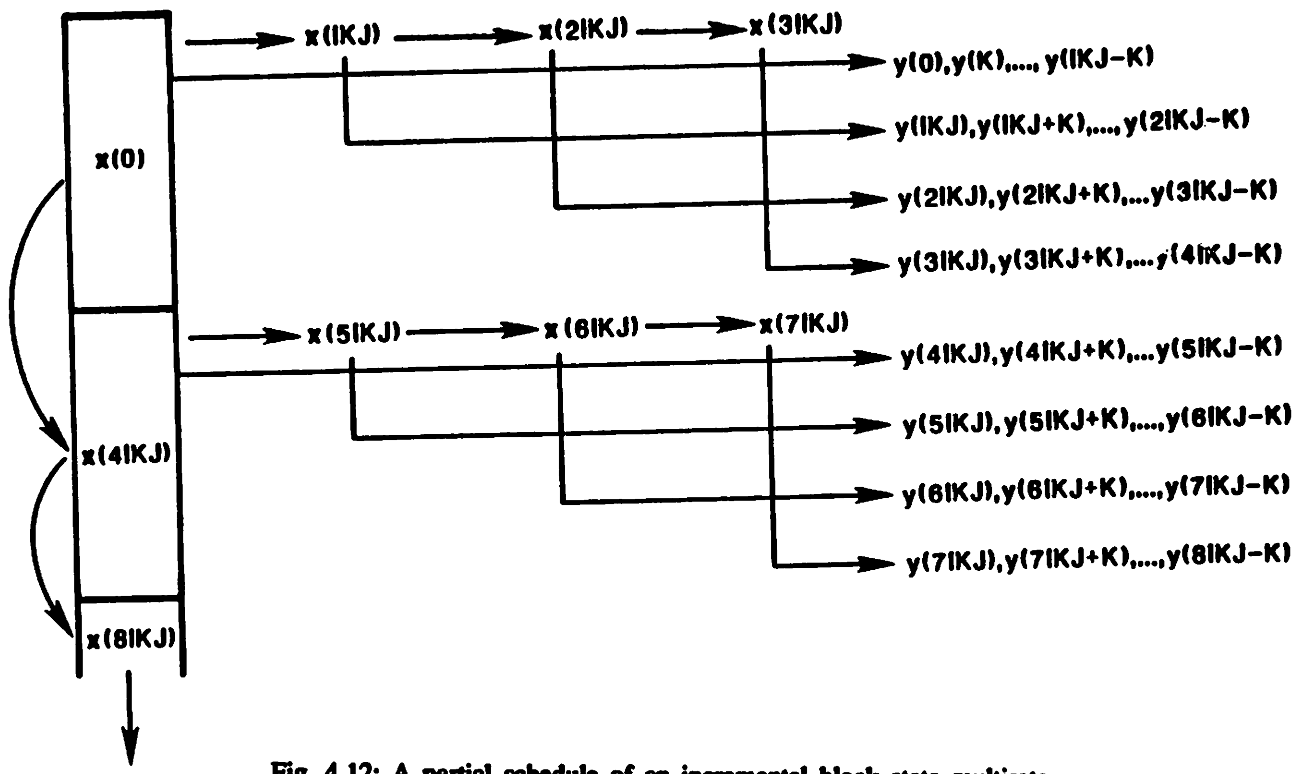

x(0)
x(4lKJ)
x(8lKJ)
x(lKJ)
x(2lKJ)
x(3lKJ)
x(5lKJ)
x(6lKJ)
x(7lKJ)
y(0),y(K),...., y(lKJ-K)
y(lKJ),y(lKJ+K),....,y(2lKJ-K)
y(2lKJ),y(2lKJ+K),...y(3lKJ-K)
y(3lKJ),y(3lKJ+K),...,y(4lKJ-K)
y(4lKJ),y(4lKJ+K),...y(5lKJ-K)
y(5lKJ),y(5lKJ+K),....,y(6lKJ-K)
y(6lKJ),y(6lKJ+K),....,y(7lKJ-K)
y(7lKJ),y(7lKJ+K),....,y(8lKJ-K)

Fig. 4.12: A partial schedule of an incremental block state multirate recursive digital filter ($L = 4l$).

Case I: $L \geq N$:

The direct form block filter is described by

$$\mathbf{y}^{(N)}(kL+L) = \mathbf{A}(L)\mathbf{y}^{(N)}(kL) + \mathbf{B}(L)\mathbf{z}^{(L)}(kL+N) . \tag{4.22}$$

In a pipelined block realization with M loop pipeline stages, we need to update $\mathbf{y}^{(N)}(kL+ML)$ using the state $\mathbf{y}^{(N)}(kL)$. We can derive a pipelined block state update realization by iterating (4.1) by $(M-1)$ times to be given by

$$\mathbf{y}^{(N)}(kL+ML) = \mathbf{A}^{M}(L)\mathbf{y}^{(N)}(kL) + \sum_{i=0}^{M-1} \mathbf{A}^{i}(L)\mathbf{z}_0((k+M-i-1)L+N) , \tag{4.23a}$$

where

$$\mathbf{z}_0(m) = \mathbf{B}(L)\mathbf{z}^{(L)}(m) , \tag{4.23b}$$

and the elements of $\mathbf{A}^{M}(L)$ are derived in appendix 4.3. The representation of (4.23) can be rewritten (using the decomposition technique) as

$$\mathbf{z}_{i+1}(m) = \mathbf{z}_i(m) + \mathbf{A}^{2^i}\mathbf{z}_i(m - 2^iL), \ i = 0,1, \ldots, (log_2M - 1) \tag{4.24a}$$

$$\mathbf{y}^{(N)}(kL+ML) = \mathbf{A}^{M}(L)\mathbf{y}^{(N)}(kL) + \mathbf{z}_{\log_2 M}((k+M-1)L+N) . \tag{4.24b}$$

The above pipelined block state update operation of (4.23) can also be derived alternatively starting from the block filter representation. We can us; a block size of ML in (4.4a) to get

$$\mathbf{y}^{(N)}(kL+ML) = \mathbf{A}(ML)\mathbf{y}^{(N)}(kL) + \mathbf{B}(ML)\mathbf{z}^{(ML)}(kL+N) , \tag{4.25}$$

which reduces to (4.23) after using (A4.11) and (A4.13).

In the pipelined block implementation, $\mathbf{z}^{(L)}((k+M-1)L+N)$ is computed using the L inputs, and is successively delayed to obtain $\mathbf{z}^{(L)}((k+i)L+N)$ for $i = (M-2)$ through 0. This computation requires $L(N+1)$ multiplication operations. The computation of $\mathbf{z}_0(m)$ requires $(NL - \frac{N(N+1)}{2})$ multiplication operations. The state update implementation of

(4.24) requires $N^2(log_2 M + 1)$ multiplication operations. The final $(L-N)$ outputs are computed non-recursively in an incremental manner using $N(L-N)$ multiplication operations. Thus, the total multiplication complexity of the pipelined direct form filter is $\left[L(3N+1)+N^2\log_2 M - \frac{N(N+1)}{2}\right]$, which is linear in block size, logarithmic in loop pipeline stages, and pipelining and block processing complexities are additive. Fig. 4.13 shows the implementation of a second order direct form recursive digital filter for a block size of 5, and 4 pipelining stages inside the recursive loop using the decomposition, and incremental output computation techniques.

Case II: $L<N$:

For this case, we need to compute (as well as update) only L states, and the $(N-L)$ states can be derived from the L available states. The L outputs $y^{(L)}(kL+ML)$ could be updated using the states $y^{(N)}(kL-(N-L))$ using the clustered look-ahead approach. In this implementation, each of the states $y(kL-(N-L))$ through $y(kL-1)$ can be derived by delaying the L available states $y(kL)$, $y(kL+1)$, ..., and $y(kL+L-1)$. However, such a realization will contain a single isolated loop delay operator, and hence the decomposition technique cannot be exploited thus leading to a linear complexity in M. Instead we can obtain another realization in which all loops contain M delay operators, so that we can exploit the decomposition technique to get a logarithmic complexity with respect to M. In this new pipelined block realization, we express $y^{(L)}(kL+ML)$ in terms of $y^{(L)}(kL)$, $y^{(L)}(kL-ML)$, ..., and $y^{(L)}(kL-RML)$, where R is $\lfloor \frac{N}{L} \rfloor$.

Let $N=RL+S$, where R and S respectively correspond to the quotient and remainder of $\frac{N}{L}$. In the block filter in (4.5), we update the block of states $y^{(L)}(kL+L)$

using $\mathbf{y}^{(L)}(kL)$, $\mathbf{y}^{(L)}(kL-L)$, ..., and $\mathbf{y}^{(L)}(kL-RL)$. The block filter in (4.5) can be rewritten as

$$\mathbf{y}^{(L)}(kL+L)=\sum_{i=0}^{R}\mathbf{Q}_{i+1}(1)\mathbf{y}^{(L)}((k-i)L)+\mathbf{B}\mathbf{z}^{(L)}(kL+L)\,, \tag{4.26a}$$

where

$$\left[0 \mid \mathbf{A}(L)\right]=\left[\mathbf{Q}_{R+1}(1) \mid \mathbf{Q}_{R}(1) \mid \cdots \mid \mathbf{Q}_{1}(1)\right]\,, \tag{4.26b}$$

and the explicit dependence of the $\mathbf{Q}_i$ and $\mathbf{B}$ matrices on L has been omitted for simplicity, and $\mathbf{Q}_i(1)$'s and $\mathbf{B}$ are $L\times L$, and 0 is $L\times(L-S)$. Since only last S states of $\mathbf{y}^{(L)}((k-R)L)$ are needed, the first $(L-S)$ columns of $\mathbf{Q}_{R+1}(1)$ correspond to zero. Hence, each matrix vector multiplication $\mathbf{Q}_{i+1}(1)\mathbf{y}^{(L)}((k-i)L)$ leads to L^2 multiplications for i ranging from 0 to $R-1$, and SL multiplications for i equal to R. The B matrix has $\frac{L(L-1)}{2}$ elements which are neither zero nor unity, and leads to $\frac{L(L-1)}{2}$ multiplication complexity. The derivation of $\mathbf{z}^{(L)}(kL+L)$ requires $L(N+1)$ multiplications. Thus, the total multiplication complexity of the block filter is $RL^2+SL+\frac{L(L-1)}{2}+L(N+1)$ or $2NL+\frac{L(L+1)}{2}$.

In the decomposition based fine-grain pipelined block processing implementation, we compute the L states $\mathbf{y}^{(L)}(kL+ML)$ using the states $\mathbf{y}^{(L)}(kL)$, $\mathbf{y}^{(L)}(kL-ML)$, ..., $\mathbf{y}^{(L)}(kL-RML)$. Here each of the states $\mathbf{y}^{(L)}(kL-ML)$, ..., $\mathbf{y}^{(L)}(kL-RML)$ can be derived from $\mathbf{y}^{(L)}(kL)$ by using M delay operators. By going through the decomposition steps as in section 3.4.3, we can derive the M stage pipelined block realization to be given by

$$\mathbf{y}^{(L)}(kL+ML)=\sum_{i=0}^{R}\mathbf{Q}_{i+1}(M)\mathbf{y}^{(L)}(kL-iML)+\mathbf{z}_{\log_2 M}(kL+ML)\,, \tag{4.27a}$$

where

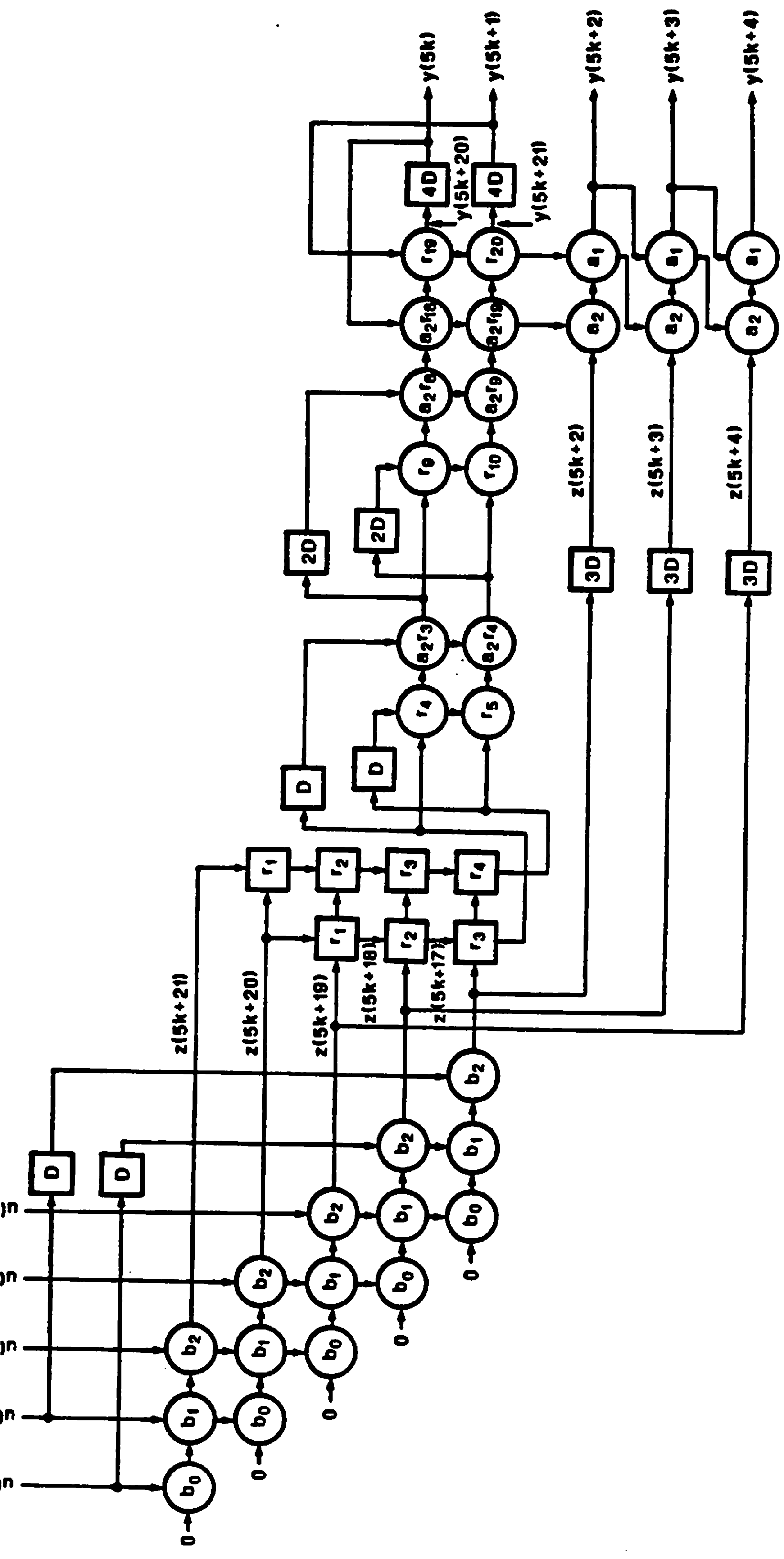


Fig. 4.13: Block implementation of a second order direct form recursive digital filter for block size of 5 and 4 loop pipelining stages obtained using the decomposition technique.

$$\mathbf{z}_{i+1}(kL+ML)=\mathbf{z}_i(kL+ML)+\sum_{j=0}^{R}(-1)^j\mathbf{Q}_{j+1}(2^i)\mathbf{z}_i(kL+ML-2^i(j+1)L)\,,\quad (4.27b)$$

$$i=0,1,\ldots,(\log_2 M-1)$$

$$\mathbf{z}_0(kL+ML)=\mathbf{B}\mathbf{z}^{(L)}(kL+ML)\,,\quad (4.27c)$$

and $\mathbf{Q}_i(2K)$ can be expressed in terms of $\mathbf{Q}_i(K)$ using the matrix versions of (A3.6) through (A3.8). The above can be proved by induction or by following the method outlined in section 3.4.3. Notice that the first $(L-S)$ columns of $\mathbf{Q}_{R+1}(2^k)$ are zero for any k. The complexity corresponding to the implementation of $\mathbf{z}^{(L)}(kL+ML)$ is $L(N+1)$ multiplications, and that for $\mathbf{B}\mathbf{z}^{(L)}(kL+ML)$ is $\frac{L(L-1)}{2}$ multiplications. The multiplication complexity to implement (4.27b) is NL for each i or $NL(\log_2 M)$ for all i's. The complexity of implementing (4.27a) is also NL. The total multiplication complexity of this implementation is $L\left[2N+\frac{L+1}{2}\right]+NL(\log_2 M)$, which is logarithmic with respect to M, linear in L, and is additive with respect to combining pipelining and block processing.

4.5. STATE SPACE FORM PIPELINED BLOCK DIGITAL FILTERS

We can use the techniques of decomposition and incremental output computation to obtain efficient realization of pipelined state space block recursive digital filters. The state update representation in (4.6) can be recaste as

$$\mathbf{x}(kL+ML)=\mathbf{A}^{ML}\mathbf{x}(kL)+\left[\mathbf{B}^{(L)}\mid\mathbf{A}^{(L)}\mathbf{B}^{(L)}\mid\cdots\mid\mathbf{A}^{((M-1)L)}\mathbf{B}^{(L)}\right]\begin{bmatrix}\mathbf{u}^{(L)}((k+M-1)L)\\ \mathbf{u}^{(L)}((k+M-2)L)\\ \vdots\\ \mathbf{u}^{(L)}(kL)\end{bmatrix}$$

(4.28a)

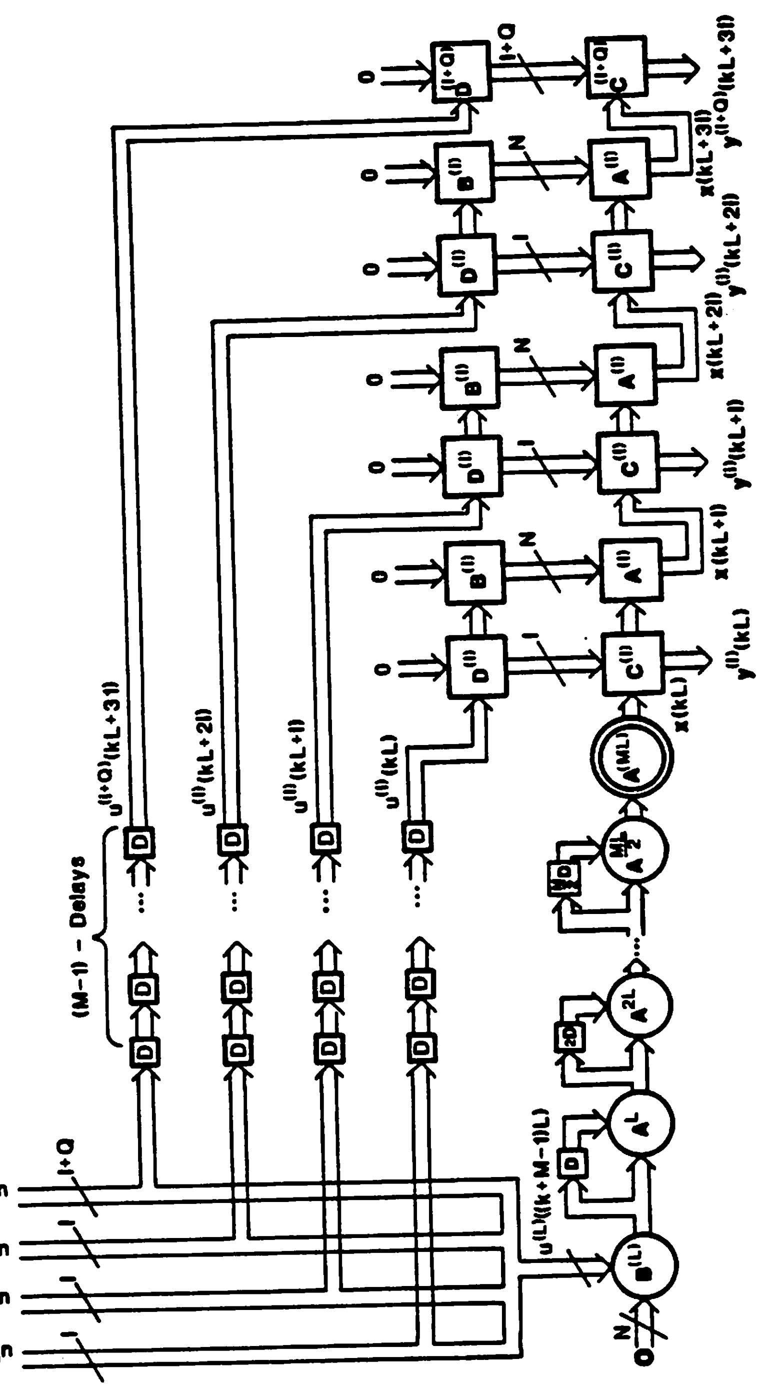


Fig. 4.14: *M*-stage pipelined incremental block filter for a block size of $L = 4l + Q$ and increment size of l obtained using the decomposition technique.

where

$$\mathbf{A}^{(L)} = A^{L} \tag{4.28b}$$

$$\mathbf{B}^{(L)} = \left[\mathbf{A}^{L-1}\mathbf{b} \mid \mathbf{A}^{L-2}\mathbf{b} \mid \cdots \mid \mathbf{b} \right] \tag{4.28c}$$

$$\mathbf{u}^{(L)}(n) = \left[u(n)\, u(n+1) \, \cdots \, u(n+L-1) \right]^{T} \tag{4.28d}$$

and $\mathbf{A}^{(L)}$ is $N \times N$, $\mathbf{B}^{(L)}$ is $N \times L$, and $\mathbf{u}^{(L)}$ is $L \times 1$. Using the decomposition technique, the state update representation of (4.28) can be rewritten as (for the case where M can be expressed as a power of 2)

$$\mathbf{x}(kL+ML) = \mathbf{A}^{ML}\mathbf{x}(kL) + \mathbf{z}_{\log_2 M}((k+M-1)L) \tag{4.29a}$$

where

$$\mathbf{z}_{i+1}((k+M-1)L) = \mathbf{z}_i((k+M-1)L) + \mathbf{A}^{L2^i}\mathbf{z}_i((k+M-1-2^i)L), \qquad i = 0,2, \ldots, \log_2 M - 1 \tag{4.29b}$$

$$\mathbf{z}_0((k+M-1)L) = \mathbf{B}^{(L)}\mathbf{u}^{(L)}((k+M-1)L)\,. \tag{4.29c}$$

The multiplication complexity to implement (4.29c) is NL, (4.29b) is $(2N - N_1)(log_2 M)$ for the case of a quasi-diagonal state update matrix, and that for (4.29a) is $(2N - N_1)$. The total state update implementation complexity is $(2N - N_1)(log_2 M + 1) + NL$ multiplications. The L outputs $y(kL)$, ..., $y(kL+L-1)$ are computed incrementally using (4.11) (exactly in the same manner as described in section 4.3.3). Adding the output computation complexity of (4.13) to the state update complexity, we can derive the total multiplication complexity of this realization to be

$$C_i = L\left[3N + \frac{2(2N-N_1) + \lceil \sqrt{2(2N-N_1)} \rceil^2}{2\lceil \sqrt{2(2N-N_1)} \rceil} + \frac{1}{2} \right] + (2N-N_1)\log_2 M - N \lceil \sqrt{2(2N-N_1)} \rceil\,, \tag{4.30}$$

which is linear in L, logarithmic in M, and the complexities due to pipelining and block processing are additive. Fig. 4.14 shows a pipelined block implementation of a pipelined block filter with $L = 4I + Q$, and M loop pipeline stages. Fig. 4.15 shows pipelined block implementation of a first order recursive digital filter for block size of 4, increment size 2, and 4 pipeline stages inside the recursive loop. The roundoff error in pipelined incremental block state filter is studied in appendix 4.4, and it is shown that the average roundoff error strictly improves with increase in the number of loop pipeline stages.

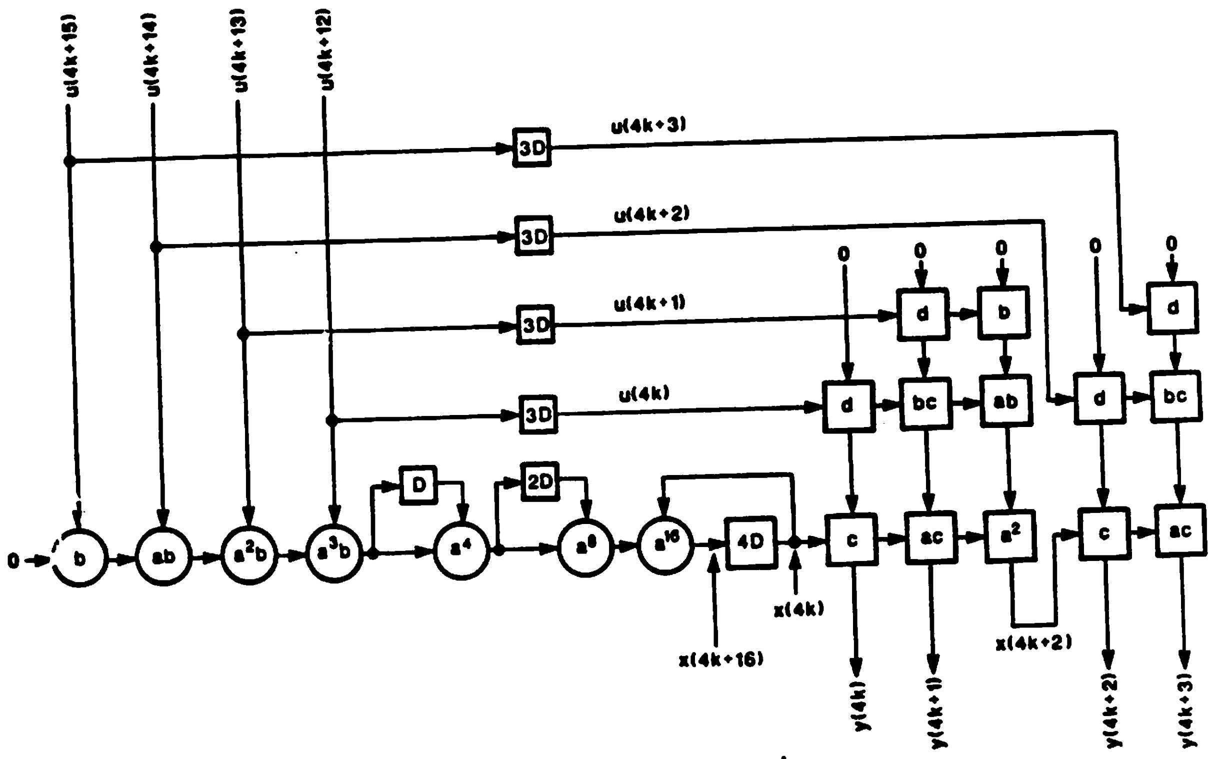


Fig. 4.15: 4-stage pipelined incremental first order block filter for a block size of four and increment size of two obtained using the decomposition technique.

To conclude, in a block realization, the eigenvalues of the system move much closer to the origin (the eigenvalues of the block filter are L-th power of that of the original filter for block size L). In a pipelined filter with M loop pipeline stages, the distance of the canceling poles from the origin is same as that of the original filter, and for each pole in the original filter, $(M-1)$ additional poles are added at equal angular and radial spacing. In a pipelined block filter with M loop pipeline stages and block size L, the distance of the poles are same as that of the block filter with block size L (and $M = 1$). As an example, for a first order system with block size 3, and 4 loop pipeline stages, the eigenvalues are at $\pm a^3$ and $\pm ja^3$, where a is the eigenvalue of the original system. The same sample rate can also be achieved with a block structure with block size 12, and this system would have a single pole at a^{12}. If we pipeline the loop by 12 stages to achieve the same sample rate, then the pipelined system would have 12 poles spaced 30 degrees apart on the circle at a distance a units apart from the origin (the pole at a is the original pole at a, and the remaining 11 are canceling poles).

4.6. CONCLUSION

We have proposed an incremental output computation technique, and using this we have derived incremental block filter structures for direct form and state space form recursive digital filters of complexity linear in block size. We have combined the incremental block filtering and the scattered look-ahead and decomposition based pipelining approaches to derive fine-grain pipelined block filter realizations of direct form and state space form recursive digital filters with complexity linear in block size, logarithmic in number of loop pipeline stages, and additive with respect to combining pipelining and block filtering.

In the next chapter, we extend the look-ahead and scattered look-ahead computation, and incremental block filtering techniques to derive high performance architectures of time-varying and adaptive digital filters.

4.7. APPENDICES

4.7.1. Appendix 4.1

In this appendix, we derive the block state update operation of the direct form recursive digital filter.

In appendix 3.2, we derived

$$y(n)=\sum_{j=0}^{N-1}\left[\sum_{l=j+1}^{N} a_l r_{j+M-l}\right] y(n-j-M)+\sum_{j=0}^{M-1} r_j z(n-j) \tag{A4.1}$$

in the context of clustered look-ahead based pipelined realization of recursive digital filters. The above can be rewritten as

$$\begin{aligned} y(n) &=\sum_{j=0}^{N-1}\left[\sum_{l=N-j}^{N} a_l r_{N-1-j+M-l}\right] y(n-M-N+1+j)+\sum_{j=0}^{M-1} r_{M-j-1} z(n-M+1+j) \qquad \text{(A4.2)} \\ &=\sum_{j=0}^{N-1}\left[\sum_{l=1}^{j+1} a_{N-l+1} r_{M+l-j-2}\right] y(n-M-N+1+j)+\sum_{j=0}^{M-1} r_{M-j-1} z(n-M+1+j) \end{aligned}$$

Case I: $L \geq N$

For the case where the block size is greater than the filter order, we need to express the N states $y(kL+L)$ through $y(kL+L+N-1)$ in terms of the N clustered past states $y(kL)$ through $y(kL+N-1)$. Substituting $n=kL+L+i$ and $M=L-N+i+1$ in (A4.2), we have

$$y(kL+L+i)=\sum_{j=0}^{N-1}\left[\sum_{l=1}^{j+1} a_{N-l+1} r_{L-N+l+i-j-1}\right] y(kL+j)+\sum_{j=0}^{L-N+i} r_{L-N+i-j} z(kL+N+j)\,. \tag{A4.3}$$

The matrix formulation for computing the N outputs can be derived by substituting

$i = 0,1, \ldots, N-1$ in the above equation, and is given in (4.4a).

Case II: $L < N$

For the case when the block size is less than the filter order, we need to compute the L outputs $y(kL+L)$ through $y(kL+2L-1)$ using the clustered N states $y(kL+L-N)$ through $y(kL+L-1)$. Substituting $n = kL+L+i$ and $M = i+1$ in (A4.2), we have

$$y(kL+L+i) = \sum_{j=0}^{N-1} \left[\sum_{l=1}^{i+1} a_{N-l+1} r_{l+i-j-1} \right] y(kL+L-N+j) + \sum_{j=0}^{i} r_{i-j} z(kL+L+j) \ . \quad \text{(A4.4)}$$

The matrix formulation for computing the L outputs can be derived by substituting $i = 0,1, \ldots, L-1$ in (A4.4), and is given by (4.5a).

4.7.2. Appendix 4.2

In this appendix, we derive an expression for the roundoff noise error in an incremental block state filter, and show that the average noise level at the outputs of the incremental block-state filter is same as that of the block-state filter and less than the parallel block-state filter. For comparison purposes, we assume that the roundoff is performed only at the outputs of the state-variable summing nodes, and at the summing nodes of the filter outputs. All roundoff noise sources are assumed to be statistically independent and stationary white with zero mean.

The roundoff error at the summing nodes of the state variables can be described by

$$\tilde{\mathbf{x}}(kL+L) = \mathbf{A}^L \tilde{\mathbf{x}}(kL) + \mathbf{e}_s(kL) , \quad \text{(A4.5)}$$

where $\mathbf{e}_s$ is of dimension $N \times 1$ and

$$E\left[\mathbf{e}_s \mathbf{e}_s^T\right] = \sigma_0^2 \mathbf{I}_N \ . \quad \text{(A4.6)}$$

The matrix $\mathbf{I}_N$ represents the unity matrix of dimension N. The variance of the errors at the summing nodes of the state variables is described by the covariance matrix

$$\mathbf{Q}=E\left[\tilde{\mathbf{x}}\tilde{\mathbf{x}}^T\right]=\mathbf{A}^L\mathbf{Q}(\mathbf{A}^T)^L+\sigma_0^2\mathbf{I}_N=\sigma_0^2\sum_{p=0}^{\infty}\mathbf{A}^{pL}(\mathbf{A}^T)^{pL}\,. \tag{A4.7}$$

The error at the summing node of the i-th output is described by

$$\tilde{y}(kL+i)=\mathbf{c}^T\mathbf{A}^i\tilde{\mathbf{x}}(kL)+e(kL+i)\,, \tag{A4.8}$$

where the last term corresponds to the error at the output summation node and

$$E\left[e^2(kL+i)\right]=\sigma_0^2\,.$$

The variance of the error at the i-th output summing node is given by (using (A4.7))

$$\frac{\sigma_i^2}{\sigma_0^2}=\mathbf{c}^T\mathbf{A}^i\sum_{p=0}^{\infty}\mathbf{A}^{pL}(\mathbf{A}^T)^{pL}(\mathbf{A}^T)^i\mathbf{c}+1\,. \tag{A4.9}$$

The average roundoff noise at the outputs is given by (using (A4.9))

$$\frac{\sigma_{av}^2}{\sigma_0^2}=\frac{\sum_{i=0}^{L-1}\sigma_i^2}{L\sigma_0^2}=\frac{1}{L}\mathbf{c}^T\sum_{p=0}^{\infty}\mathbf{A}^p(\mathbf{A}^T)^p\mathbf{c}+1\,. \tag{A4.10}$$

The average roundoff noise for the block-state filter has been verified to be exactly same as that given by (A4.10) [8]. In [11] the roundoff noise of the parallel block-state has been derived and shown to be greater than that of the block-state filter. Hence, to conclude, the average roundoff noise of the incremental block-state filter is same as that of the block-state filter and less than that of the parallel block-state filter.

4.7.3. Appendix 4.3

In this appendix, we derive an expression for the elements of $\mathbf{A}^M(L)$ as a function of filter coefficients and the sequence r_i (for the case when M can be expressed as a power of 2). This expression is useful for deriving pipelined block realization of direct form recursive digital filters for the case $L \ge N$. The sequence r_i has been defined in appendix 3.1.

Theorem A4.1: The elements of $\mathbf{A}^M(L)$ are given by

$$\left[A^{M}(L)\right]_{ij} = \sum_{k=0}^{i-1} a_{N-k} r_{LM-N+i-j+k} \tag{A4.11}$$

Proof: (by induction): Assume (A4.11) is true for M, and then prove that it also holds for $2M$. The elements of $A^{2M}(L)$ are given by

$$\begin{aligned}\left[A^{2M}(L)\right]_{ij} &= \sum_{l=1}^{N} \left[A^{M}(L)\right]_{il} \left[A^{M}(L)\right]_{lj} \\ &= \sum_{l=1}^{N} \left[\sum_{k=0}^{l-1} a_{N-k} r_{LM-N+i-l+k}\right] \left[\sum_{m=0}^{j-1} a_{N-m} r_{LM-N+l-j+m}\right] \\ &= \sum_{m=0}^{j-1} a_{N-m} \sum_{l=1}^{N} \left[\sum_{k=0}^{l-1} a_{N-k} r_{LM-N+i-l+k}\right] r_{LM-N+l-j+m} \\ &= \sum_{m=0}^{j-1} a_{N-m}\, r_{2LM-N+i-j+m} \quad \text{using theorem (A3.1)} \quad \textit{QED}\ .\end{aligned} \tag{A4.12}$$

Corollary: $A^{M}(KL) = A^{KM}(L)$.

Theorem A4.2:

$$B(ML) = \left[A^{M-1}(L)B(L) \;\; A^{M-2}(L)B(L) \;\; \cdots \;\; B(L)\right] \tag{A4.13}$$

Proof: From (4.5c), we have

$$\left[B(ML)\right]_{i,kL+j} = r_{ML-N+i-kL-j}\ . \tag{A4.14a}$$

We need to prove that the above element must be the ij-th element of $A^{M-1-k}(L)B(L)$.

This element is given by

$$\begin{aligned}\left[A^{M-1-k}(L)B(L)\right]_{ij} &= \sum_{m=1}^{N} \left[A^{M-1-k}(L)\right]_{im} \left[B(L)\right]_{mj} \\ &= \sum_{m=1}^{N} \left[\sum_{s=0}^{M-1} a_{N-s} r_{(M-1-k)L-N+i-m-s}\right] r_{L-N+m-j} \\ &= r_{ML-N+i-kL-j} \quad \text{using theorem A3.1} \quad \textit{QED}\ .\end{aligned} \tag{A4.14b}$$

4.7.4. Appendix 4.4

In this appendix, we derive the roundoff error in a pipelined block state space filter with block size L and M loop pipeline stages, and show that this error is strictly less than that of a block filter with block size L (i.e. with $M = 1$) under the assumptions stated in appendix 4.2.

The roundoff error at the summing nodes of the state variables of the pipelined block filter is described by

$$\tilde{\mathbf{x}}(kL + ML) = \mathbf{A}^{ML}\tilde{\mathbf{x}}(kL) + \mathbf{e}_s(kL + ML), \tag{A4.15}$$

where $\mathbf{e}_s$ is of dimension $N \times 1$ and

$$E\left[\mathbf{e}_s\mathbf{e}_s^T\right] = \sigma_0^2\mathbf{I}_N . \tag{A4.16}$$

The matrix $\mathbf{I}_N$ represents the unity matrix of dimension N. The variance of the errors at the summing nodes of the state variables is described by the covariance matrix

$$\mathbf{Q} = E\left[\tilde{\mathbf{x}}\tilde{\mathbf{x}}^T\right] = \mathbf{A}^{ML}\mathbf{Q}(\mathbf{A}^T)^{ML} + \sigma_0^2\mathbf{I}_N = \sigma_0^2\sum_{p=0}^{\infty}\mathbf{A}^{pML}(\mathbf{A}^T)^{pML} . \tag{A4.17}$$

The error at the summing node of the i-th output of the block filter is described by

$$\tilde{y}(kL+i) = \mathbf{c}^T\mathbf{A}^i\tilde{\mathbf{x}}(kL) + e(kL+i), \tag{A4.18}$$

where the last term corresponds to the error at the output summation node and

$$E\left[e^2(kL+i)\right] = \sigma_0^2 .$$

The variance of the error at the i-th output summing node is given by (using (A4.17))

$$\frac{\sigma_i^2}{\sigma_0^2} = \mathbf{c}^T\mathbf{A}^i\sum_{p=0}^{\infty}\mathbf{A}^{pML}(\mathbf{A}^T)^{pML}(\mathbf{A}^T)^i\mathbf{c} + 1 . \tag{A4.19}$$

The average roundoff noise at the outputs is given by (using (A4.19))

$$\frac{\sigma_{av}^2}{\sigma_0^2} = \frac{\sum_{i=0}^{L-1}\sigma_i^2}{L\sigma_0^2} = \frac{1}{L}\mathbf{c}^T\sum_{j=0}^{L-1}\mathbf{A}^j\left[\sum_{p=0}^{\infty}\mathbf{A}^{pML}(\mathbf{A}^T)^{pML}\right](\mathbf{A}^T)^j\mathbf{c} + 1 , \tag{A4.20}$$

which is a strictly decreasing function in M.

4.8. REFERENCES

(1) Gold, B. and Jordan, K.L.,"A Note on Digital Filter Synthesis", *Proceedings of IEEE*, Vol. 65, pp.1717-1718, Oct. 1968

(2) Voelcker, H.B., and Hartquist, E.E.,"Digital Filtering Via Block Recursion", *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-18, pp. 169-176, June 1970.

(3) Burrus, C.S., "Block Implementation of Digital Filters ", *IEEE Transactions on Circuit Theory*, Vol. CT-18, pp. 697-701, Nov. 1971.

(4) Moyer, A.L., "An Efficient Parallel Algorithm for Digital IIR Filters", *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 525-528

(5) Mitra, S.K. and Gnanasekaran, R., "Block Implementation of Recursive Digital Filters - New Structures and Properties ", *IEEE Transactions on Circuits and Systems*, Vol. CAS-25, pp.200-207, April 1978.

(6) Barnes, C.W. and Shinnaka, S.,"Block Shift Invariance and Block Implementation of of Discrete-Time Filters", *IEEE Trans on* Circuits and Systems, Vol. CAS-27, pp.667-672, Aug. 1980.

(7) Zeman, J., and Lindgren, A.G., "Fast Digital Filters with Low round-off Noise, " *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, pp.716-723, July 1981.

(8) Schwartz, D. A. and Barnwell, T.P. III, "Increasing the Parallelism of Filters Through Transformation to Block State Variable Form", *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 1984.

(9) Lu, H.H., Lee, E.A. and Messerschmitt, D.G., "Fast Recursive Filtering with Multiple Slow Processing Elements", *IEEE Trans. on Circuits and Systems*, Vol. CAS-32, No.11, November 1985, pp. 1119-1129.

(10) Nikias, C.L.,"Fast Block Data Processing Via a New IIR Digital Filter Structure", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 32, No.4, August 1984.

(11) Parhi, K.K., and Messerschmitt, D.G., "Block Digital Filtering via Incremental Block-State Structure", *Proceedings of the IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987

(12) Sung, W., and Mitra, S.K., "Efficient Multi-Processor Implementation of Recursive Digital Filters", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, April 1986, pp. 257-260

(13) Wu, C.W., and Cappello, P.R., "Computer-Aided Design of VLSI Second Order Sections", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987

(14) Wu, C.W., and Cappello, P.R., "Application Specific CAD of VLSI Second Order Sections", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, May 1988, pp. 813-825

(15) Arun, K.S., "Ultra-High-Speed Parallel Implementation of Low-Order Digital Filters", *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1986, pp. 944-946

(16) Parhi, K.K., Chen, W.L., and Messerschmitt, D.G., "Architecture Considerations for High Speed Recursive Filtering", *Proceedings of the 1987 IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987

(17) Parhi, K.K., and Messerschmitt, D.G., "A Bit-Parallel Bit Level Recursive Filter Architecture", *Proc. of the IEEE International Conference on Computer Design*, NY, 1986.

(18) Parhi, K.K., and Messerschmitt, D.G., "Concurrent Cellular VLSI Adaptive Filter Architectures", *IEEE Transactions on Circuits and Systems*, Vol. CAS-34, October 1987

(19) Miyawaki, T., and Barnes, C.W., "Multirate Recursive Digital Filters - A General Approach and Block Structures", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-31,

No. 5, October 1983, pp. 1148-1154

(20) Leiserson, C., Rose, F., and Saxe, J., "Optimizing Synchronous Circuitry by retiming", *Third Caltech Conference on VLSI*, Pasadena, CA, March 1983

(21) Kung., S.Y., "On Supercomputing with Systolic/Wavefront Array Processors", *Proceedings of IEEE*, Vol. 72, No. 7, July 1984

(22) Schwartz, D.A., and Barnwell III, T.P., "A Graph Theoretic Technique for the Generation of Systolic Implementation of Shift Invariant Flow Graphs", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, San Diego, March 1984

5

ADAPTIVE DIGITAL FILTERS

5.1. INTRODUCTION

In chapters 3 and 4, we proposed look-ahead and decomposition techniques to implement recursive digital filters using fine-grain pipelining, and developed the incremental block filter structure for block implementation of recursive digital filters. Many real-time applications, such as high-end storage devices, system identification, spectrum estimation, and image processing require high sample rate implementation of adaptive digital filters. Unlike in recursive filters, the coefficients in adaptive digital filters need to be updated each sample period (so as to minimize some error criterion). The tap coefficients are usually updated using the error innovations and the past coefficients. This updating of the tap coefficients in the adaptive filters in each sample period makes their high-speed implementation difficult and challenging.

The notion of block processing [1-12] was used in [13] to derive an architecture for high-speed implementation of adaptive digital filters. Although the block architecture improves the iteration bound [14-15] in adaptive filters, it does so at a considerable expense in hardware. The block structures are also often referred to as word-parallel or vector processing in the literature (see Fig. 5.1 for definition of word-serial/word-parallel, bit-serial/bit-parallel terminologies). These block filters contain a single L-slow [16] latch inside the loop, and belong to the class of word-level pipelined architectures. In this chapter, we use the look-ahead and decomposition techniques [6,17-19] to develop high sampling rate architectures for adaptive digital filters using fine-grain pipelining. Our implementations are pipelined at bit- or multi-bit level, and require logarithmic complexity with respect to speedup or level of loop pipelining. When pipelining is not adequate to achieve the desired speedup, we combine incremental block processing with fine-grain pipelining to achieve high-speed [12].

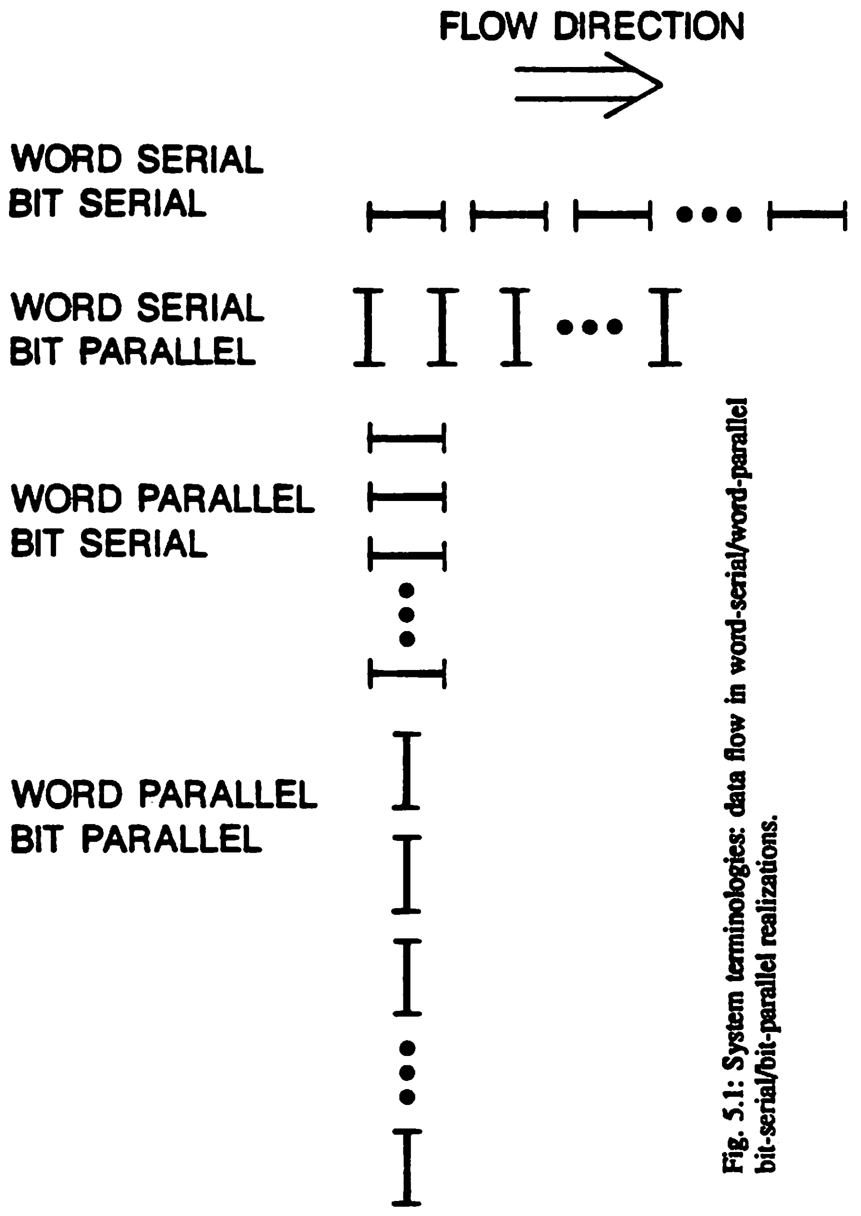


Fig. 5.1: System terminologies: data flow in word-serial/word-parallel bit-serial/bit-parallel realizations.

In a block implementation with block size L and M loop pipeline stages, the *implementable loop delay operator* corresponds to z^{-LM} (at the sample rate). However, the delay operator available in most recursive algorithms corresponds to z^{-1}. The additional concurrency is created by using $(LM-1)$ steps of look-ahead and LM-way interleaving is not needed. The complexity of our fine-grain pipelined block architecture is linear in block size and is asymptotically same as that of the non-recursive systems (for both bit-serial and bit-parallel realizations).

Look-ahead computation applies to digital filters, and also to adaptive filters since the recursive portion of such filters is linear for all the adaptation algorithms proposed to date. Adaptive systems can be implemented by using transversal structures [20, ch.3], triangular arrays based on QR decomposition [21-25], or lattice structures [20, ch.4,6]. The triangular arrays and state space based transversal filters require $O(N^2)$ complexity, where N is the order of the adaptive filter. Furthermore, pipelining the least square structures will lead to much higher complexity since the loop computation in these structures involves square root and division, and these operations lead to an $O(W^2)$ latency as compared to $O(W)$ latency in multiplications for a word length of W. To avoid the global error adaptation bottleneck in the adaptive filter (as shown in Fig. 5.2), *block adaptation* scheme (not to be confused with block or word-parallel structures discussed in this chapter) has been proposed [26-27], but these structures suffer from slower tracking capability at high sampling rates. Adaptive lattice filters lead to a complexity linear in filter order, and avoid the global error bottleneck, since the coefficient of each stage is adapted order-recursively based on the error residual of the previous stage. These structures are best suited for high sampling rate realizations, since the adaptation coefficient recursion inside each stage is linear in nature. Hence, we have concluded that the best adaptive filter structure for high sampling rate implementations is the adaptive lattice filter. Specifically, the coefficient update of any lattice stage simplifies to a first-order linear time-varying recursion, which can be implemented with arbitrary concurrency.

The organization of the chapter is as follows. The look-ahead and decomposition techniques are extended to linear time-varying recursive systems in section 5.2. Fine-grain pipelined block architectures for adaptive lattice filters are derived in section 5.3 using the normalized stochastic gradient lattice filter as an example. The complexity and latency issues are addressed in section 5.4, and the implementation methodology trade-offs are discussed in section 5.5.

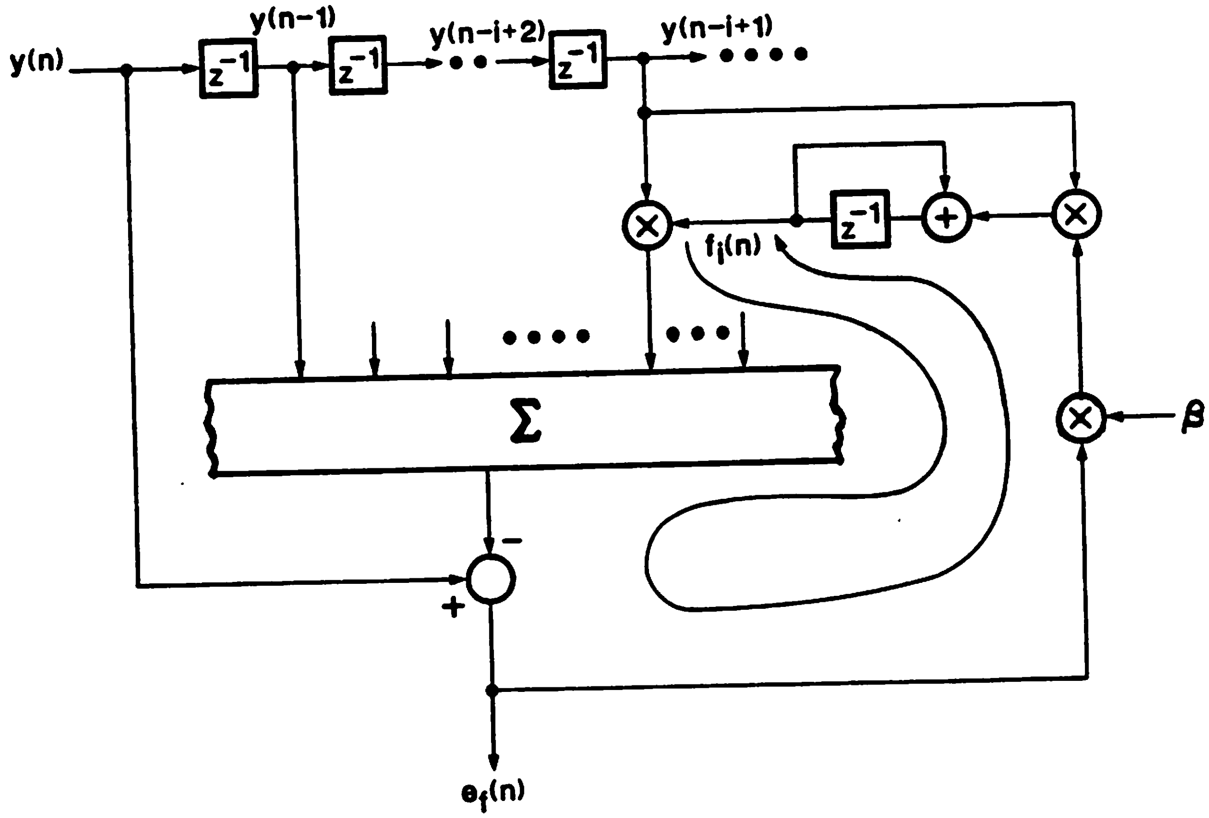


Fig. 5.2: Global error bottleneck in transversal adaptive filter.

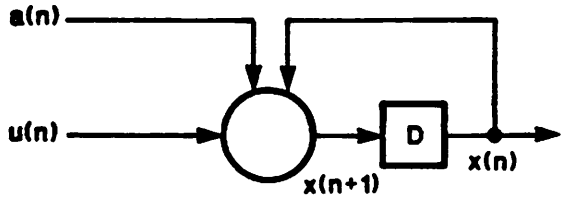


Fig. 5.3(a): A first-order linear time-varying recursion.

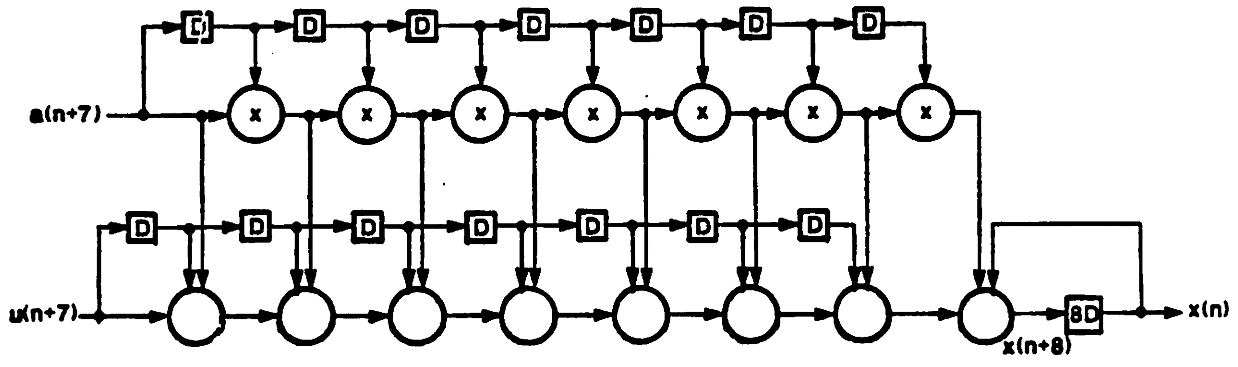


Fig. 5.3(b): An equivalent pipelined recursion obtained using look-ahead with 8 pipeline stages inside the recursive loop,

5.2. LOOK-AHEAD IN TIME-VARYING FILTERS

First we address look-ahead and decomposition in first-order linear time-varying system and then in higher order time-varying systems.

5.2.1. First Order Linear Time-Varying Recursion

Consider the first-order linear time-varying recursion shown in Fig. 5.3(a) and described by

$$x(n+1) = a(n)x(n) + u(n),\ x(0) = x_0 . \tag{5.1a}$$

An equivalent recursion using $(M-1)$-steps of *look-ahead* can be obtained as:

$$x(n+M) = \prod_{i=0}^{M-1} a(n+M-i-1)x(n) + \left[1\ a(n+M-1) \prod_{i=0}^{1} a(n+M-i-1)\ \cdots\ \prod_{i=0}^{M-2} a(n+M-i-1) \right] \begin{bmatrix} u(n+M-1) \\ u(n+M-2) \\ \vdots \\ u(n) \end{bmatrix} . \tag{5.1b}$$

The initial states can again be precomputed as in the time-invariant case. For a causal input sequence, the starting initial states can be derived to be [17]

$$x(0) = x_0,\ x(-i) = \left[\prod_{j=1}^{i} a(-j) \right]^{-1} x_0 ,\quad i = 1,2,\ldots,(M-1) , \tag{5.1c}$$

where the values of the non-zero time-varying coefficients $a(-1), \cdots ,a(-M+1)$ can be chosen arbitrarily. The implementation of this recursion is shown in Fig. 5.3(b), and has a multiplication complexity $(2M-1)$, which is linear in steps of look-ahead. The product of the coefficients can no longer be pre-computed because of the time-varying nature of the recursion. However, these can be dynamically computed by a separate array, which can be fully pipelined because it is non-recursive. Hence, as in the time-invariant filter, full pipelining of the recursive portion of the system can be achieved using look-ahead computation.

For situations where M can be expressed as a power of 2, we can use the decomposition technique to obtain an implementation, which requires a logarithmic increase in hardware with respect to the number of steps of look-ahead. The decomposed state update implementation is described by

$$x(n+M) = f_{\log_2 M}(n+M-1)x(n) + z_{\log_2 M}(n+M-1), \quad (5.2a)$$

where

$$f_{i+1}(n+M-1) = f_i(n+M-1)f_i(n+M-1-2^i), \; f_0(n+M-1) = a(n+M-1), \quad (5.2b)$$

$$z_{i+1}(n+M-1) = z_i(n+M-1) + f_i(n+M-1)z_i(n+M-1-2^i), \quad (5.2c)$$

$$z_0(n+M-1) = u(n+M-1), \; i = 0,1, \ldots, (\log_2 M-1),$$

and requires a complexity of $(2\log_2 M + 1)$ multiplications. A pipelined decomposed implementation of the first-order time-varying system is shown in Fig. 5.3(c) for $M = 8$.

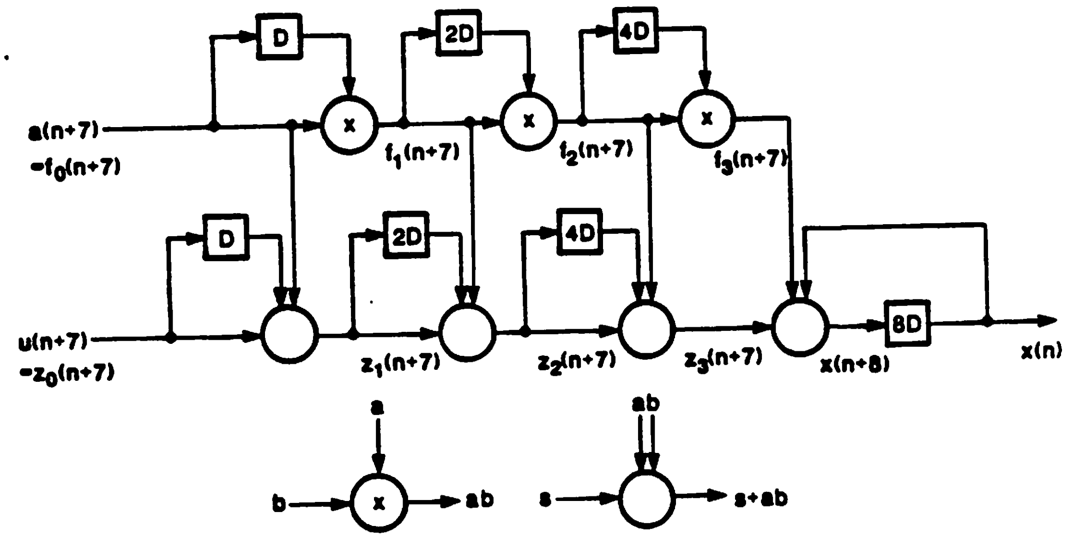


Fig. 5.3(c): Pipelined decomposed first-order time-varying recursion.

5.2.2. Higher Order Linear Time-Varying Filters

Now we illustrate the application of scattered look-ahead and decomposition principles for pipelining of higher order time-varying filters using a second-order time-varying filter as an example. Consider the system

$$x(n) = a_1(n)\,x(n-1) + a_2(n)\,x(n-2) + u(n)\,. \tag{5.3a}$$

We can use the scattered look-ahead approach to express $x(n)$ as a function of $x(n-M)$ and $x(n-2M)$, and exploit the decomposition technique to obtain a logarithmic complexity. After some manipulation, the M stage pipelined filter is

$$x(n+M) = f_{\log_2 M}(n+M)\,x(n) + g_{\log_2 M}(n+M)\,x(n-M) + z_{\log_2 M}(n+M)\,, \tag{5.3b}$$

where

$$f_{i+1}(n+M) = f_i(n+M)f_i(n+M-2^i) + g_i(n+M) + \frac{f_i(n+M)g_i(n+M-2^i)}{f_i(n+M-2^{i+1})} \tag{5.3c}$$

$$g_{i+1}(n+M) = -\,\frac{f_i(n+M)g_i(n+M-2^i)g_i(n+M-2^{i+1})}{f_i(n+M-2^{i+1})} \tag{5.3d}$$

$$z_{i+1}(n+M) \;= z_i(n+M) + f_i(n+M)z_i(n+M-2^i) - \frac{f_i(n+M)g_i(n+M-2^i)}{f_i(n+M-2^{i+1})}z_i(n+M-2^{i+1}) \tag{5.3e}$$

$$\begin{aligned} f_0(n+M) &= a_1(n+M) \\ g_0(n+M) &= a_2(n+M), \quad i = 0, 1, \ldots, (\log_2 M - 1)\,. \\ z_0(n+M) &= u(n+M) \end{aligned} \tag{5.3f}$$

This realization can be implemented with ($5\log_2 M + 2$) multipliers (each pipelined by M stages), and $\log_2 M$ pipelined dividers. Note that although the original realization did not require a divider, the pipelined realization does require dividers for the scattered look-ahead technique to be applicable. Since the adaptive lattice filter stages correspond to a first order time-varying filter, we will not pursue the higher order time-varying filters further in this chapter.

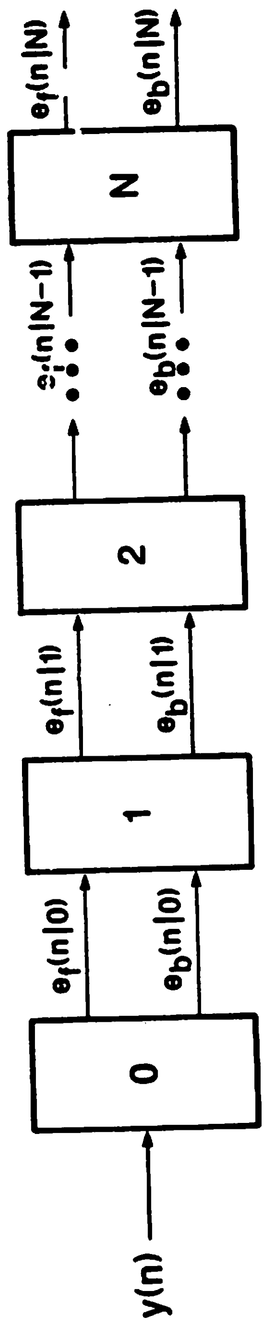


Fig. 5.4(a): Word-serial lattice filter.

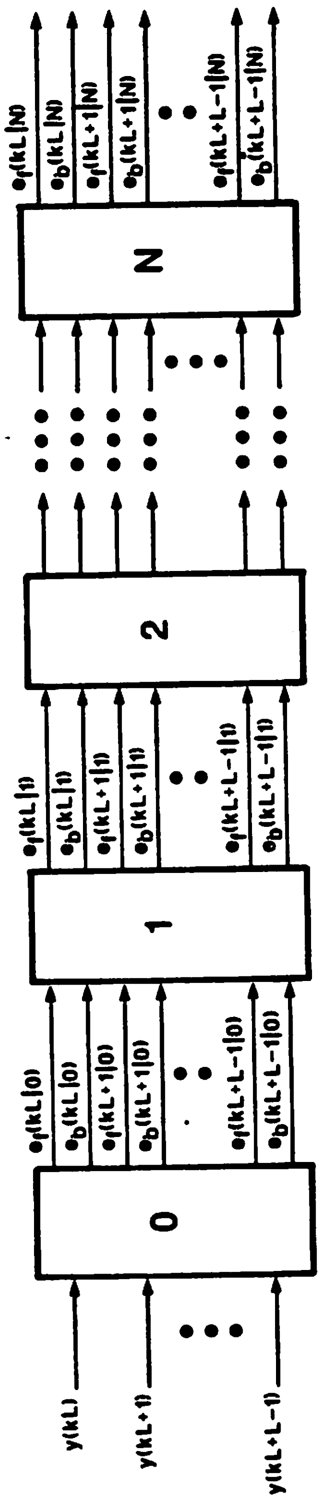


Fig. 5.4(b): Word-parallel lattice filter.

5.3. HIGH SAMPLE RATE ADAPTIVE FILTERING

In this section, we derive high sampling rate architectures for adaptive lattice filters based on the techniques of look-ahead computation, decomposition, and incremental output computation using the normalized stochastic gradient lattice filter algorithm as an example. These techniques combined together lead to asymptotically optimal realizations and provide a "system solution" to area-efficient high speed adaptive filtering. These basic techniques apply to other lattice filter and joint process estimator algorithms as well.

The block diagram of a word-serial lattice filter realization is shown in Fig. 5.4(a) and the word-parallel or block or vectorized version is shown in Fig. 5.4(b). First we define the symbols. The time indices are referred to as n for a word-serial realization and kL in a word-parallel realization with block size L. The adaptive filter is assumed to be of order N and any intermediate stage is referred to as the p-th stage.

5.3.1. Initialization Lattice Stage

The initialization section of the normalized stochastic gradient lattice filter is described by

$$\varepsilon(n) = \lambda\varepsilon(n-1) + y^2(n), \quad \varepsilon(-1) = 0 \tag{5.4a}$$

$$e_f(n|0) = e_b(n|0) = \frac{y(n)}{\sqrt{\varepsilon(n)}}, \tag{5.4b}$$

and is shown in Fig. 5.5(a). This implementation can be transformed into an equivalent pipelined block implementation using the look-ahead computation technique. An equivalent state update realization for a block size L with M stages of pipelining inside the recursive loop is described by

$$\varepsilon(kL+ML-1) = \lambda^{ML}\varepsilon(kL-1) + \begin{bmatrix} 1 \ \lambda \ \lambda^2 \ \cdots \ \lambda^{ML-1} \end{bmatrix} \begin{bmatrix} y^2(kL+ML-1) \\ y^2(kL+ML-2) \\ \vdots \\ y^2(kL) \end{bmatrix}, \qquad (5.5a)$$

and has a multiplication/square complexity of $2ML$. The non-recursive look-ahead term of the above state update implementation can again be implemented with the use of the decomposition technique (see section 5.2). The $L \times M$ decomposed implementation is described by

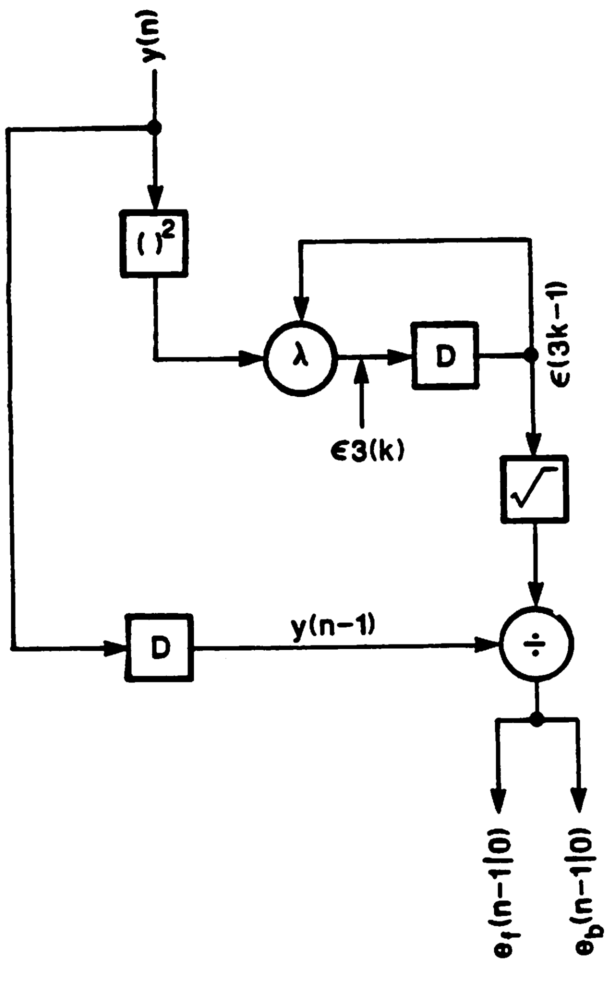


Fig. 5.5(a): Word-serial representation of initialization normalized stochastic gradient adaptive lattice stage. The processing elements are defined in Fig. 5.5(b).

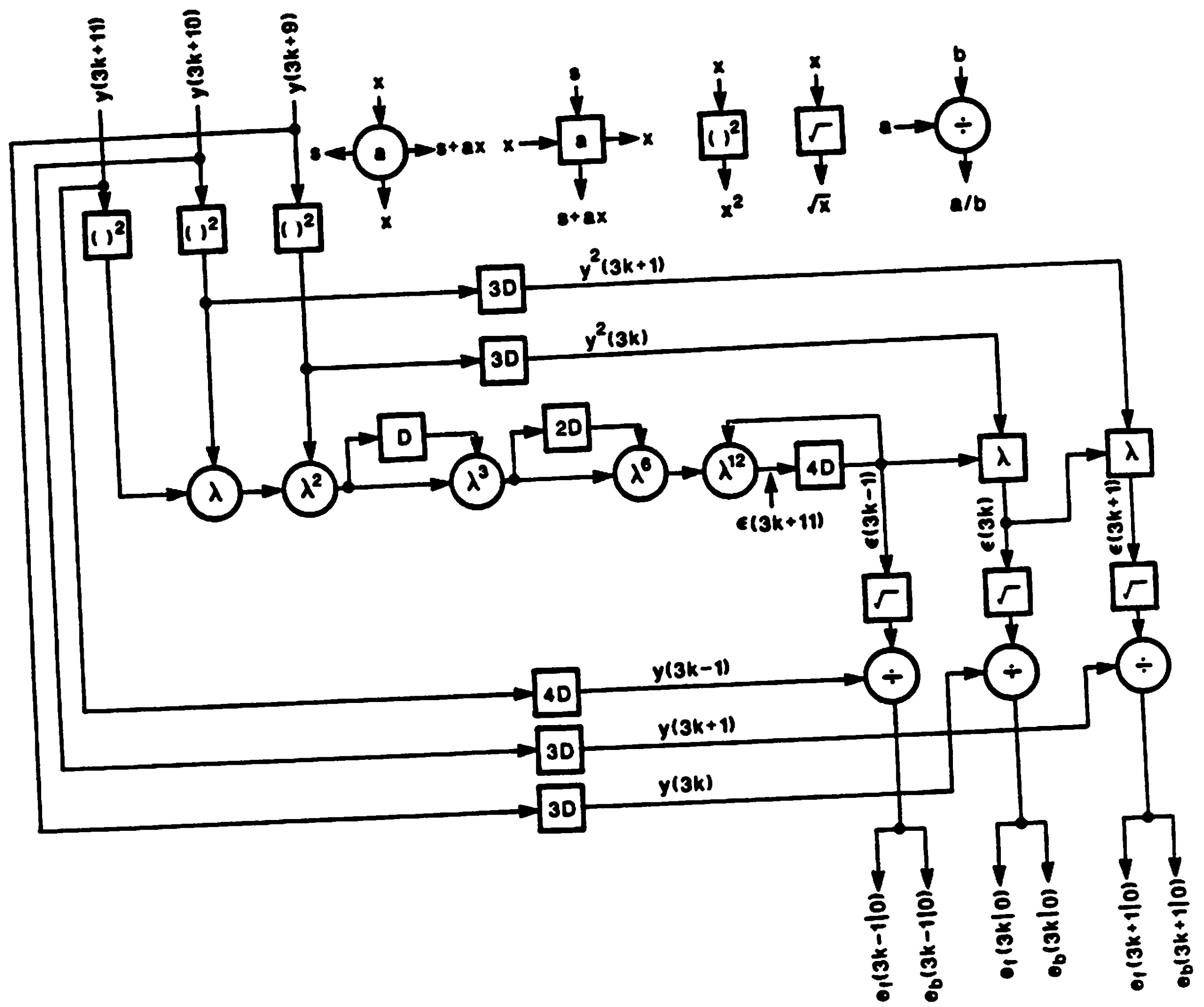


Fig. 5.5(b): Word-parallel initialization lattice stage with block size of 3 and 4 stages of loop pipelining obtained using look-ahead, decomposition, and incremental omputation.

$$\varepsilon(kL+ML-1)=\lambda^{ML}\varepsilon(kL-1)+\left[1\ \lambda^{L}\ \lambda^{2L}\ \cdots\ \lambda^{(M-1)L}\right]\begin{bmatrix} z((k+M-1)L) \\ z((k+M-2)L) \\ z((k+M-3)L) \\ \vdots \\ z(kL) \end{bmatrix} \quad (5.5b)$$

where

$$z(kL)=\sum_{i=0}^{L-1}\lambda^{i}y^{2}(kL+L-1-i)\,. \quad (5.5c)$$

The decomposition technique can be exploited further to obtain an implementation with state update complexity of $(2L+log_2M)$ multiplication/square operations (see Fig. 5.5(b)).

The forward and backward error innovations are calculated in a sequential or iterative manner using the incrementally computed intermediate states, i.e. $\varepsilon(kL)$, $\varepsilon(kL+1)$, ..., $\varepsilon(kL+L-2)$ (which were missed due to the block state update process) based on the known state $\varepsilon(kL-1)$, and the corresponding inputs. Based on the techniques of look-ahead computation, decomposed state update implementation, and incremental output computation, a pipelined block realization of the initialization section is shown in Fig. 5.5(b) for a block size of 3, and 4-stages of pipelining inside the recursive loop. The implementation complexity of the initialization stage is $(3L+log_2M-1)$ multiplication/square operations, and $2L$ division/square root operations.

5.3.2. Typical Lattice Stage

In a typical lattice stage, the error innovations are calculated in an order-recursive manner based on the error innovations generated by the previous stage, and the coefficients inside each stage are linearly adapted in a time-recursive manner. A typical lattice stage is described by

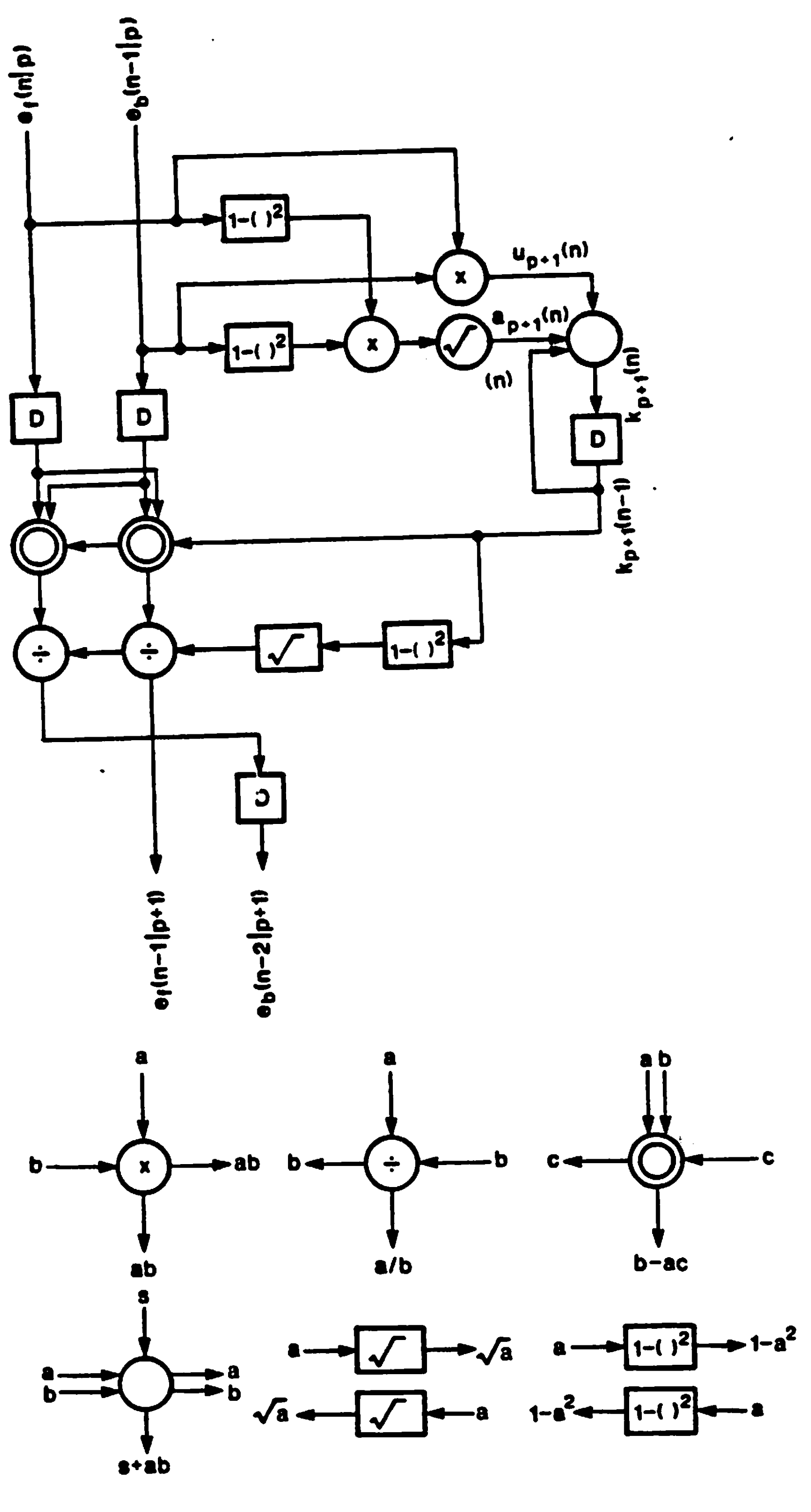


Fig. 5.6(a): Word-serial typical lattice stage.

$$e_f(-1\,|p)=e_b(-1\,|p)=k_{p+1}(-1)=0,\quad 0\le p\le N-1 \tag{5.6a}$$

$$e_f(n\,|p+1)=\frac{e_f(n\,|p)-k_{p+1}(n)e_b(n-1\,|p)}{\sqrt{1-k^2_{p+1}(n)}} \tag{5.6b}$$

$$e_b(n\,|p+1)=\frac{e_b(n-1\,|p)-k_{p+1}(n)e_f(n\,|p)}{\sqrt{1-k^2_{p+1}(n)}} \tag{5.6c}$$

$$k_{p+1}(n)=\left[\left\{1-e^2{}_f(n\,|p)\right\}\left\{1-e^2{}_b(n-1\,|p)\right\}\right]^{1/2}k_{p+1}(n-1) \tag{5.6d}$$

$$+e_f(n\,|p)e_b(n-1\,|p),$$

and is shown in Fig. 5.6(a). The time-recursive coefficient or state update corresponds to a first-order linear time-varying recursion (as described in section 5.2) where $a_{p+1}(n)$ and $u_{p+1}(n)$ are described by

$$a_{p+1}(n)=\left[\left\{1-e^2{}_f(n\,|p)\right\}\left\{1-e_b{}^2(n-1\,|p)\right\}\right]^{1/2} \tag{5.7a}$$

$$u_{p+1}(n)=e_f(n\,|p)e_b(n-1\,|p)\,. \tag{5.7b}$$

An example of a word-parallel or block implementation for a block size of 3 and 4-stage pipelining of the multiplier inside the recursive loop is shown in Fig. 5.6(b) for a typical normalized stochastic gradient lattice stage using the techniques of look-ahead transformation, decomposition, and incremental output computation.

5.4. COMPLEXITY AND LATENCY

Now we study the complexity and latency aspects of high-speed pipelined block filters using the typical normalized stochastic gradient lattice stage as an example. The complexity of the pipelined block stochastic gradient realization is $(10L+2\log_2 M-2)$ multiplications/squares, $2L$ divisions, and $2L$ square roots for a block size of L and M stages of pipelining inside the recursive loop (assuming M to be a power of two) (see Fig. 5.6(b)).

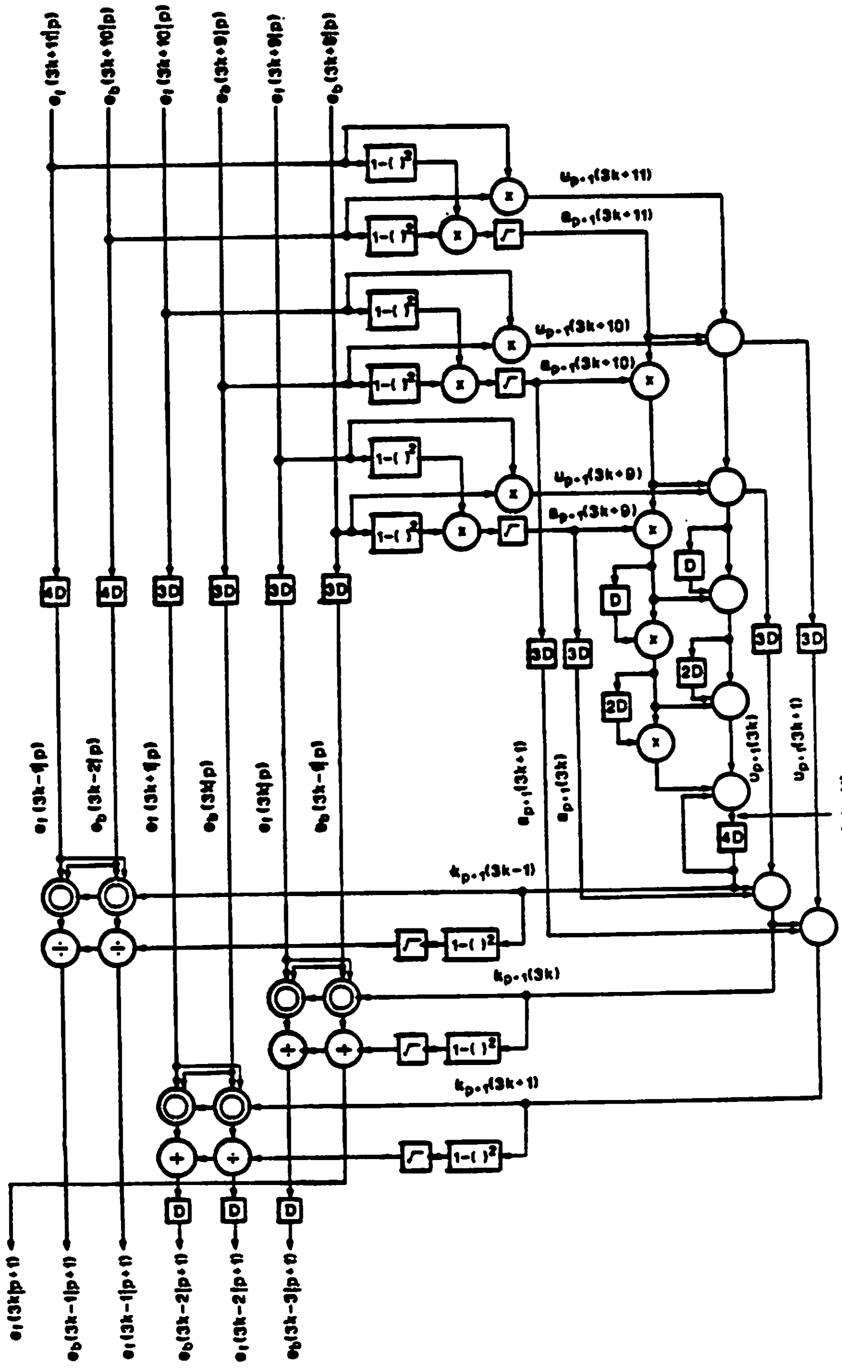


Fig. 5.6(b): Word-parallel typical lattice stage with block size of 3 and 4 stages of pipelining.

Before we derive the system latency, let us consider the latency of each computational element such as multiplication/square, square root and division. For a bit-level pipelined implementation with clock period t_a, an upper bound on latency is $2Wt_a$ for a multiplication/square operation, and W^2t_a for a division/square root operation (t_a corresponds to the clock period of a one-bit controlled binary adder-subtractor cell). In our implementation, the recursive loop involves a multiplication operation and is pipelined by M stages. Hence, the clock period of the M stage pipelined multiplier *approximately* corresponds to $\frac{2W}{M}t_a$. With this clock rate, the division/square root operation will require $\frac{WM}{2}$ stages of pipelining. Thus, the latency of a multiplication/square operation (T_m) is about M clock periods, and that of a division/square root operation (T_d) is about $\frac{WM}{2}$ clock periods or cycles. Note that, each clock period corresponds to L sample periods in a word-parallel or block implementation with block size L.

The latency of each stage is $\left[(2L+log_2M+2)T_m + 4T_d\right]$, where T_m and T_d respectively represent word-level multiplication/square and division/square root computational latency. The per-stage latency in terms of clock periods is $M\left[2L+log_2M+2W+2\right]$ and in terms of sample periods is $LM\left[2L+log_2M+2W+2\right]$, where W is the word-length.

5.5. IMPLEMENTATION METHODOLOGY TRADEOFFS

In this section, we compare the complexity and latency of a word-level pipelined (i.e. with one pipelining stage inside the recursive loop) word-parallel architecture of the typical normalized stochastic gradient adaptive filter with that of the fine-grain pipelined word-parallel architecture using M-stages of pipelining inside the recursive loop. We show that for a specified sampling rate realization of the adaptive lattice filter, the amount of hardware and system latency can be saved by about a factor of M in the latter case. These conclusions hold good for any other high-speed adaptive filter or joint process estimator architectures as well.

The pipelined word-parallel implementations can be implemented using either bit-serial [29-32] or bit-parallel cellular arithmetic structures [33-35]. Then, we compare performance of a bit-level pipelined bit-serial word-parallel (BSWP) architecture with bit-level pipelined bit-parallel word-parallel (BPWP) architecture for a specified sampling rate realization.

5.5.1. Word-level Pipelining *vs* M-stage Pipelining

Let the specified sampling rate of an adaptive filter be KMf_c, where f_c is the clock rate of a word-level pipelined word-serial filter (i.e. with one pipelining latch inside the recursive loop). The specified sampling rate can be achieved by using a word-level pipelined word-parallel architecture with block size KM, or using a pipelined word-parallel architecture with M-stages of pipelining inside the recursive loop, and a block size of K. The complexity of the word-level pipelined word-parallel realization with block size KM is $(10KM-2)$ multiplications/square operations, and $4KM$ division/square root operations, and has a latency of $KM(2KM+2W+2)$ sample periods. The complexity of the word-parallel realization using M stages of pipelining inside the recursive loop and block size K is $(10K+2\log_2 M-2)$ multiplication/square operations, and $4K$ division/square root operations, and has a latency of $KM(2K+log_2 M+2W+2)$ sample periods. Thus, by using M-stages of pipelining inside the recursive loop, we can save the amount of hardware by about a factor of M and reduce the system latency by a factor of M asymptotically with respect to K.

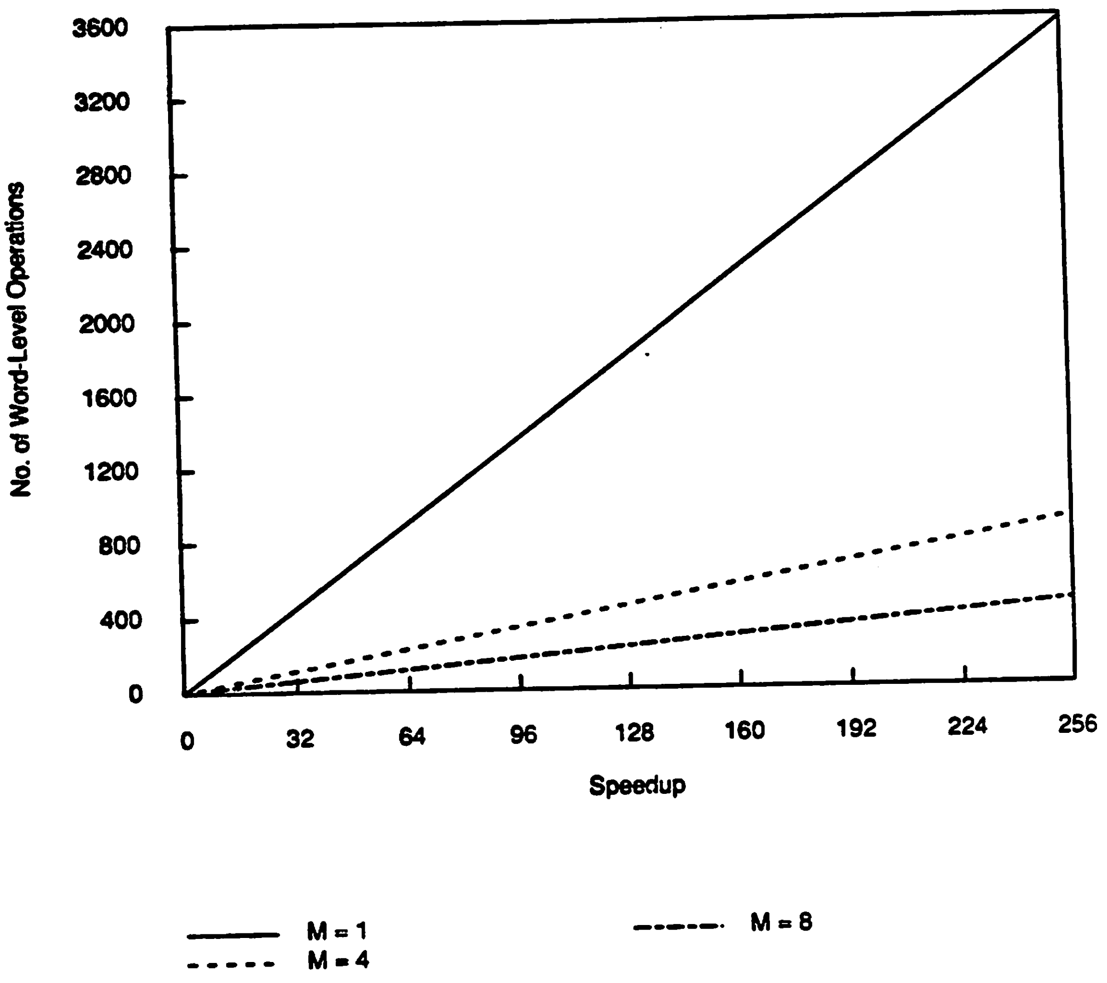


Fig. 5.7: Number of word-level operations *vs* speedup as a function of number of pipelining stages inside the recursive loop.

The number of word-level operations are compared in Fig. 5.7 for a desired speedup (or increase in sampling rate) for various stages of pipelining inside the recursive loop. In the figure, the speedup corresponds to the product LM, and the number of word-level operations corresponds to the sum of word-level multiplication, division, square and square root units. We observe that for a desired speedup, the number of word-level operations reduces about linearly as M increases.

5.5.2. Bit-Serial *vs* Bit-Parallel

In this section, we compare the performance of bit-level pipelined bit-serial word-parallel (BSWP) and bit-level pipelined bit-parallel word-parallel (BPWP) methodologies.

For a word-parallel implementation with M-stages of pipelining in the recursive loop and block size L, each implementable delay corresponds to z^{-LM} at the sample rate. In a bit-level pipelined multiplication operation, the latency corresponds to $2W$ clock periods (one clock period represents shimming delay). Hence, M is $2W$ in bit-parallel methodology. Since $2W$ clock or bit periods represent 2 sample periods in bit-serial methodology, M corresponds to 2 in bit-serial.

With a block size of unity, the achievable sample rate corresponds to f_a in bit-parallel methodology, and $\frac{f_a}{W}$ in bit-serial methodology, where W is word length, and f_a corresponds to the throughput of a latched binary controlled adder-subtractor (a single cell). The slow speed of the bit-serial can be made up with larger block sizes to match with the speed of the bit-parallel. Let us assume that a sampling rate of Kf_a is required. This can be achieved with a block size of K in bit-level pipelined bit-parallel methodology, and KW in bit-serial methodology. Note that, the latches in the converter circuits operate at a speed Kf_a in bit-parallel methodology, and KWf_a in bit-serial methodology. The technology boundary requirement in the bit-serial methodology is thus W times higher than that in the bit-parallel methodology. Table 5.1 summarizes the complexities in bit-serial and bit-parallel methodologies for a sampling rate of Kf_a.

Table 5.1: Bit-Serial Word-Parallel *vs* Bit-Parallel Word-Parallel Comparison

Characteristics	BPWP		BSWP	
	Mult/Sqr	Div/Sqrt	Mult/Sqr	Div/Sqrt
Complexity (word-level)	$10K+log_2W$	$4K$	$10KW$	$4KW$
Complexity (cell-level)	$(10K+log_2W)W^2$	$4KW^2$	$10KW^2$	$4KW^2$

The complexity of bit-serial and bit-parallel are asymptotically identical for a specified sampling rate. The system complexity (of all N stages) of the pipelined block filter is linearly proportional to the system order and block size. This complexity requirement is asymptotically identical to that of the non-recursive systems.

For a sampling rate of Kf_a, the latency of the bit-level pipelined bit-parallel word-parallel implementation is $2KW(2K+log_2W+2W+3)$ sample periods, and that of the bit-serial word-parallel is $2KW(2KW+2W+3)$ sample periods. Thus for a specified sampling rate, the latency of the bit-serial is much longer than that of the bit-parallel. The bit-serial implementation can realize a wider range of sampling rates, and is particularly desirable for low sampling rates. Furthermore, bit-serial implementations are easy to test. The pinout requirements are identical for bit-serial and bit-parallel. However, for a specified sampling rate, the bit-serial requires a wider technology boundary. These implementation methodology comparisons are consistent with that for recursive filters as well [19].

5.6. CONCLUSION

Based on the techniques of look-ahead computation, decomposition based state update implementation, and incremental output computation, we have presented pipelined block implementations of high sampling rate adaptive lattice filters of complexity linear in filter order and block size. The techniques presented in this chapter are also applicable to joint process estimators. The implementations derived in this chapter belong to the class of systolic [36] and wavefront type architectures [28]. The complexity of each cell in our realization is that of a latched binary-controlled adder-subtractor. The cellular arrays for the division, square, and square root can be found in [33-35]. The data flow in division/square root arrays in [33-35] is bidirectional, and will lead to interleaving when pipelined. The cellular array based architectures with unidirectional data flow can be derived with some modifications, and these can be pipelined without requiring interleaving. We have developed pipelinable cellular architectures for arithmetic computations (such as, multiplication, division, and square root) in both bit-serial and bit-parallel methodologies, which are beyond the scope of this thesis.

The word-level pipelined architecture is suitable for software-programmable implementation using a coarse-grain parallel processor and can lead to linear speedup with respect to the number of processors. The pipelined word-serial architecture can lead to an exponential speedup with respect to the number of processors when a fine-grain parallel processor is available. The pipelined word-parallel architecture (with M-stages of loop pipelining and L block size) can be implemented using a fine-grain parallel processor to achieve speedup by about a factor of M with respect to the number of processors.

So far we have addressed high-speed implementation of one-dimensional recursive and adaptive filters. In the next chapter, we derive high-speed implementation of two-dimensional recursive digital filters, which are useful for real-time digital filtering of video images.

5.7. REFERENCES

(1) Barnes, C.W. and Shinnaka, S., "Block Shift Invariance and Block Implementation of of Discrete-Time Filters", *IEEE Trans on* Circuits and Systems, Vol. CAS-27, pp.667-672, Aug. 1980

(2) Zeman, J., and Lindgren, A.G., "Fast Digital Filters with Low round-off Noise, " *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, pp.716-723, July 1981

(3) Schwartz, D.A., and Barnwell, T.P. III, "Increasing the Parallelism of Filters Through Transformation to Block State Variable Form", *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 1984

(4) Lu, H.H., Lee, E.A. and Messerschmitt, D.G., "Fast Recursive Filtering with Multiple Slow Processing Elements", *IEEE Trans. on Circuits and Systems*, Vol. CAS-32, No.11, November 1985, pp. 1119-1129

(5) Meyer, R.A., and Burrus, C.S., "A Unified Analysis of Multirate and Periodically Time-Varying Digital Filters", *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, No. 3, March 1975, pp. 162-168

(6) Parhi, K.K., and Messerschmitt, D.G., "A Bit Parallel Bit Level Recursive Filter Architecture", *Proc. of the IEEE International Conference on Computer Design*, NY, 1986

(7) Nikias, C.L.,"Fast Block Data Processing Via a New IIR Digital Filter Structure", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 32, No.4, August 1984

(8) Arun, K.S., "Ultra-High-Speed Parallel Implementation of Low-Order Digital Filters", *Proceedings of the IEEE International Symposium on Circuits and Systems*, San Jose, May 1986

(9) Sung, W., and Mitra, S.K., "Efficient Multi-Processor Implementation of Recursive Digital Filters", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, April 1986

(10) Wu, C.W., and Cappello, P.R., "Computer-Aided Design of VLSI Second Order Sections", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987

(11) Parhi, K.K., and Messerschmitt, D.G., "Block Digital Filtering via Incremental Block-State Structure", *Proceedings of the IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987

(12) Parhi, K.K., and Messerschmitt, D.G., "Area-Efficient High Speed VLSI Adaptive Filter Architectures", *Proceedings of the IEEE International Conference on Communications*, Seattle, June 1987

(13) Meng, T., and Messerschmitt, D.G., "Arbitrarily High Sampling Rate Adaptive Filters", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, April 1987

(14) Renfors, M., and Neuvo, Y., "The Maximum Sampling Rate of Digital Filters under Hardware Speed Constraints", *IEEE Trans. on Circuits and Systems*, Vol. CAS-28, No. 3, March 1981, pp. 196-202

(15) Fettweis, A., "Realizability of Digital Filter Networks", *Arch. Elek. Ubertrangung*, Vol. 30, Feb 1976, pp. 90-96

(16) Leiserson, C., Rose, F., and Saxe, J., "Optimizing Synchronous Circuitry by retiming", *Third Caltech Conference on VLSI*, Pasadena, CA, March 1983

(17) Parhi, K.K., and Messerschmitt, D.G., "Look-Ahead Computation: Improving Iteration Bound in Linear Recursions", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987

(18) Parhi, K.K. *et al*, "Architecture Considerations for High Speed Filtering", *Proceedings of the IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987

(19) Parhi, K.K., and Messerschmitt, D.G., "Design Methodologies for VLSI Digital Filters", *unpublished work*

(20) Honig, M. and Messerschmitt, D.G., *Adaptive Filters: Structures, Algorithms, and Applications*, Kluwer Academic Press, Hingham, Mass., 1985

(21) Gentleman, W.M., "Least-Squares Computations by Givens Rotations Without Square Roots", *J. Inst. Maths. Applics.*, Vol. 12, pp. 329-336, 1973

(22) Gentleman, W.M., and H.T. Kung, "Matrix Triangularization by Systolic Arrays", *Proc. of SPIE*, Real Time Signal Processing IV, Vol. 298, pp. 298-303, 1981

(23) McWhirther, J.G., "Recursive Least-Squares Minimization using a Systolic Array", *Proc. of SPIE*, Real Time Signal Processing VI, Vol. 431, pp. 105-112

(24) Schreiber, R.J. and Kuekes, P.J., "Systolic Linear Algebra Machines in Digital Signal Processing", in *VLSI and Modern Signal Processing*, edited by S.Y. Kung, H.J. Whitehouse, and T. Kailath, Prentice-Hall, Englewood Cliffs, N.J., pp. 389-405

(25) Haykin, S., *Adaptive Filter Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1986 (chapter-10)

(26) Clark, G.A., and Mitra, S.K., "Block Implementation of Adaptive Digital Filters", *Trans on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 3, pp. 744-752, June 1981

(27) Cioffi, J.M., "The Block-Processing FTF Adaptive Algorithm", *Trans on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34, No. 1, February 1986

(28) Kung., S.Y., "On Supercomputing with Systolic/Wavefront Array Processors", *Proceedings of IEEE*, Vol. 72, No. 7, July 1984

(29) Jackson, L.B., Kaiser, J.F., and McDonald, H.S., "An Approach to Implementation of Digital Filters", *IEEE Trans on Audio and Electroacoustics*, Vol. AU-16, No. 3, pp. 413-421, 1968

(30) Lyon, R.F., "Two's Complement Pipeline Multipliers", *IEEE Trans. on Communication*, 1976, pp. 418-424

(31) Van Ginderdeuren, J.K.J., De Man, H.J., Gon Calves, N.F., Van Noije, W.A.M., "Compact NMOS Building Blocks and a Methodology for Dedicated Digital Filter Applications", *IEEE Journal of Solid State Circuits*, Vol. SC-18, No. 3, 1983, pp. 306-315

(32) Freeny, S.L., "Special Purpose Hardware for Digital Filtering", *Proc. of IEEE*, Vol. 63, April 1975, pp. 633-648

(33) Hwang, Kai, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley, NY, 1979

(34) Guild, H.H., "Some Cellular Logic Arrays for Non-Restoring Binary Division", *The Radio and Elec. Engr.*, Vol. 39, pp. 345-348, June 1970

(35) Majithia, J.C., "Cellular Array for Extraction of Squares and Square-Roots of Binary Numbers", *IEEE Trans. on Computers*, Vol. C-20, No. 12, pp. 1617-18, December 1971

(36) Kung, H.T., "Why Systolic Architectures", *IEEE Computer*, Vol. 15, No. 1, January 1982

6

DIGITAL FILTERS FOR IMAGE PROCESSING

6.1. INTRODUCTION

In this chapter, we propose high sample rate architectures for two-dimensional recursive filters. These filters are useful for high-speed filtering of digital images, and will find applications in high-definition televisions (which require bandwidths of order of 100 to 200 Mhz), high-speed image transmission, and other real-time image processing applications. Achieving concurrency in two-dimensional non-recursive systems by the use of pipelining and/or block processing is straightforward [1]. However, the inherent sequential nature of recursive computations limits the opportunities for achieving high speed by the use of pipelining and/or parallelism. This inherent bottleneck in recursive computations dictates a lower (upper) bound on the iteration period (maximum achievable sampling rate) [2-3]. This iteration period bound in a two-dimensional recursive system with a block size $L_1 \times L_2$ is given by

$$T_\infty = \frac{1}{L_1 L_2} \max_{l \in S_l} \left[\frac{D_l}{M_l} \right] , \qquad (6.1)$$

where S_l represents the set of loops in the computation graph, D_l represents the total computational latency associated with loop l, and M_l represents the number of latches or delay operators inside the loop l. The maximum achievable sampling rate for the computation graph is $\frac{1}{T_\infty}$. For a non-block implementation (i.e. with unity L_1 and L_2), this definition of iteration period bound reduces to that in [2-3].

It is necessary to transform these sequential computations into equivalent concurrent computations, which can then be used to achieve high speed realizations by exploiting pipelining and/or parallelism. In the previous chapters, we studied these transformation techniques in the context of one dimensional recursive and feed-forward adaptive lattice filters [4-7]. Specifically, we introduced new look-ahead and decomposition techniques to pipeline one dimensional recursive and adaptive filters with logarithmic increase in hardware with respect to the number of loop pipeline stages [6-7]. A second approach to achieving concurrency is by block processing, where a block of inputs are processed concurrently to generate a block of outputs [4-6, 8-31]. Several block structures have been presented in [4-6, 8-20] for block implementation of one dimensional recursive digital filters. In chapter 4, we introduced an *incremental block filter* structure [5] for block implementation of one dimensional state space recursive digital filters with linear complexity in block size as opposed to the square complexity needed in the block-state structure first proposed by Barnes [12] or the parallel block-state structure proposed by Nikias [16]. A direct form one-dimensional recursive filter block structure with linear complexity has also been recently introduced by Wu and Cappello in [19-20].

This chapter addresses efficient implementation of concurrent two-dimensional recursive digital filters using the techniques of pipeline interleaving with M pipeline stages inside the recursive loop and/or block processing with block size $L_1 \times L_2$. In a pipelined implementation with M loop delays, M independent computations are processed in an interleaved manner. In a block implementation with block size $L_1 \times L_2$, we process $L_1 L_2$ samples simultaneously. First we define two types of block implementations for a two dimensional filter.

Definition 6.1: A block implementation is referred to as a *one-dimensional block* implementation if either L_1 or L_2 i. unity.

Definition 6.2: A block implementation is referred to as a *two-dimensional block* implementation if both L_1 and L_2 are greater than unity.

In general, if the dimension of block size of a block structure (K) is less than the filter dimension (N), then this block filter is referred to as a K-dimensional block filter. When the dimension of the block size equals the filter dimension N, we refer to it as an N-dimensional block filter, or simply a block filter. Unlike one dimensional systems, two and multi-dimensional linear recursive computations possess large amount of inherent concurrency. We refer to the locus of the samples which can be concurrently computed as the *concurrent computation region*. We exploit this concurrency to derive pipelined and one-dimensional block architectures for two-dimensional recursive digital filter, *without requiring any algorithm transformation, and without any extra hardware overhead* (except the area required for pipeline latches). Full hardware utilization is achieved by interleaving or indexing the input samples in a way that eliminates the sequential dependency problem [32-34]. The major attractiveness in achieving high speed using one-dimensional block processing approach results due to the fact that it does not require any hardware overhead (unlike two-dimensional block processing approach), and the implementation aspect of the system is exactly identical to that of a system without any recursion or feedback.

Two-dimensional block implementation of two-dimensional recursive digital filters are derived using algorithm transformation techniques. Many two-dimensional block structures have been presented in [21-30]. In this chapter, we extend the *look-ahead technique* to two-dimensional case, and use this to extend the one-dimensional *incremental block filter* [5,19-20] to the two-dimensional case [31]. This two-dimensional incremental block filter has a multiplication complexity $O(Max(L_1^2L_2, L_1L_2^2))$ for a block size of $L_1{\times}L_2$, as opposed to an $O(L_1^2L_2^2)$ multiplication complexity in the two-dimensional block structures in [21-30]. It is also possible to exploit fast transform [35-39] and short convolution techniques [40] in our new incremental block filter to reduce the multiplication complexity of the block structure (in a manner similar to the existing block filters [25,27]).

The pipeline interleaving approach is area-efficient, since it exploits concurrency with no hardware penalty. If sufficient speed cannot be achieved by pipelining alone, then we need to combine pipelining with one-dimensional block processing (since, one-dimensional block processing requires a linear increase in hardware), and if the speedup is still not adequate, then only we need to combine pipelining with (two-dimensional) block processing.

The organization of the chapter is as follows. Section 6.2 addresses the inherent concurrency in two-dimensional recursive systems, and presents techniques to exploit this concurrency in the context of one-dimensional block processing and pipeline interleaving, and pipelined one-dimensional block processing. The two-dimensional incremental block digital filter is derived in section 6.3. Efficient structures which combine pipeline interleaving and two-dimensional block filtering approaches are studied in Section 6.4. The extension of the concurrent computation region, pipeline interleaving and the incremental block filtering concepts to higher dimensions is outlined in section 6.5.

6.2. PIPELINING AND ONE-DIMENSIONAL BLOCK PROCESSING

In this section, we discuss concurrency in direct form and local state space form 2D recursive digital filters. Let a linear shift invariant quarter plane two-dimensional causal recursive digital filter be described by

$$H(z_1,z_2)=\frac{Y(z_1,z_2)}{U(z_1,z_2)}=\frac{\sum_{i_1=0}^{N_a}\sum_{i_2=0}^{N_b} b_{i_1,i_2} z_1^{-i_1} z_2^{-i_2}}{1-\sum_{\substack{i_1=0 \\ (i_1,i_2)\neq(0,0)}}^{N_a}\sum_{i_2=0}^{N_b} a_{i_1,i_2} z_1^{-i_1} z_2^{-i_2}}\,. \tag{6.2}$$

6.2.1. Direct Form 2D Filters

The direct form representation of the 2D filter in (6.2) is described by

```
for each ($n_2 = 0$ to $(J_2 - 1)$ ) {
for each ($n_1 = 0$ to $(J_1 - 1)$ ) {
    $y(n_1,n_2)=\sum_{\substack{i_1=0 \\ (i_1,i_2)\neq(0,0)}}^{N_a}\sum_{i_2=0}^{N_b} a_{i_1,i_2} y(n_1-i_1,n_2-i_2)+\sum_{i_1=0}^{N_a}\sum_{i_2=0}^{N_b} b_{i_1,i_2} u(n_1-i_1,n_2-i_2)$    (6.3)
}},
```

where $u(n_1,n_2)$ and $y(n_1,n_2)$ respectively represent input and output samples. The "for each" statements indicate sequential processing with respect to the corresponding loop index. The numbering of the first and second indices of the samples in a frame of size $J_1 \times J_2$ is shown in Fig. 6.1. The above sequential processing requires $J_1 J_2$ steps or cycles. The samples in a frame can be processed row by row, column by column, or diagonally. For the purposes of this chapter, we assume row by row processing of the samples.

6.2.1.1. Sequential Processing

In a traditional row by row processing of the filter in (6.3), the outputs are processed sequentially in the order $y(0,0)$, $y(1,0)$, ..., $y(J_1-1,0)$; $y(0,1)$, $y(1,1)$, ..., $y(J_1-1,1)$; ....; $y(0,J_2-1)$, $y(1,J_2-1)$, ..., $y(J_1-1,J_2-1)$. In this ordering, the computation of the output $y(n_1,n_2)$ begins at time index or cycle $(n_1+J_1n_2)$. The mapping of the spacial sample location to its time index is referred to as an *index mapping function* [41], and is given by

$$I(n_1,n_2) = n_1 + J_1 n_2 \tag{6.4}$$

for a row by row processing. This index mapping function is not unique, and similar index mapping functions can be obtained for column by column and diagonal processing.

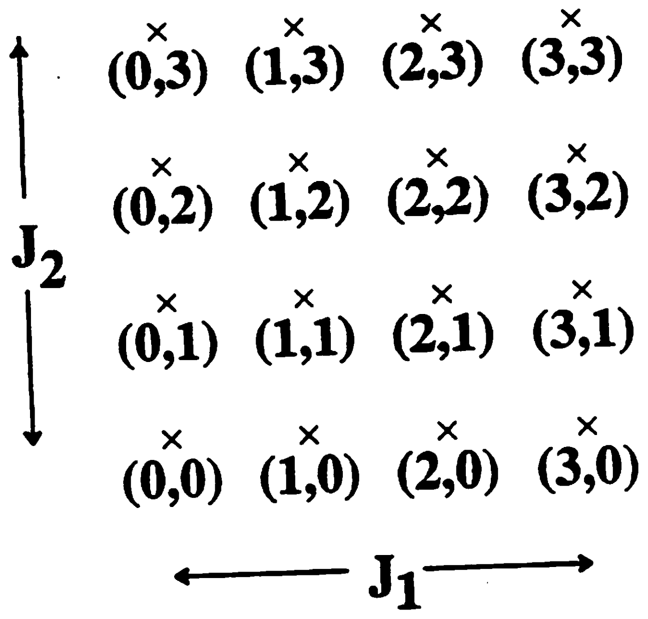


Fig. 6.1: Indexing of samples in a 4×4 Frame.

The index mapping function is sufficient to describe the delay operators in two-dimensional filtering. We illustrate this with a simple example. The sample (n_1,n_2) will be processed in cycle $(n_1+J_1n_2)$ (from (6.4)). The next sample in the row (n_1+1,n_2) will be processed in cycle $(n_1+J_1n_2+1)$. This implies the row delay operator z_1^{-1} corresponds to a single delay. Similarly the sample $y(n_1,n_2+1)$ will be processed in cycle $(n_1+J_1n_2+J_1)$. This implies the column delay operator z_2^{-1} represents J_1 delays (often referred to as a line delay in literature). These delay operators are obvious from examination of the coefficients of n_1 and n_2 in (6.4) also. In (6.4), the coefficients of n_1 and n_2 are respectively 1 and J_1, and these correspond to the row and column delay operators. Fig. 6.2 shows a block diagram of a two-dimensional block filter for $N_a = N_b = 1$ using appropriate row and column delay operators. The iteration bound for this implementation is the time required for a word level multiplication and two additions.

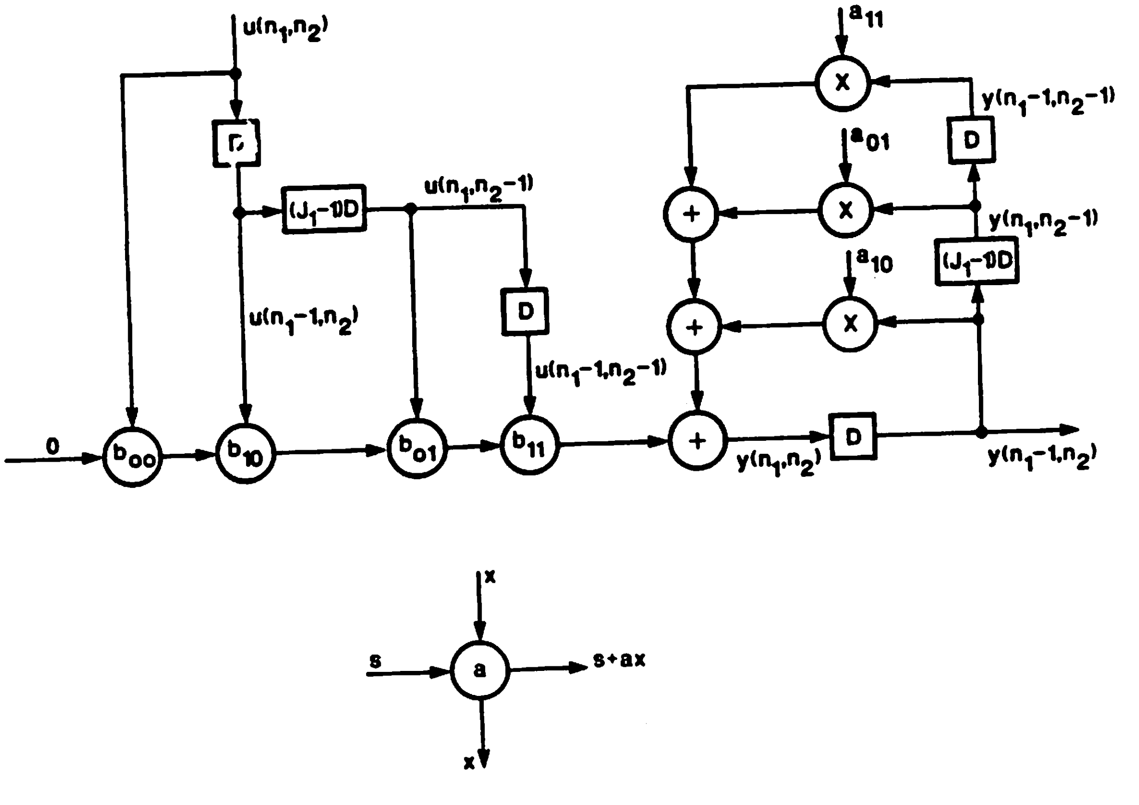


Fig. 6.2: A two-dimensional recursive digital filter.

We define a *segment* to correspond to the number of rows being processed concurrently, and we assume the filtering process to be performed by appending the beginning of the next segment to the end of the current segment temporally. In the traditional row by row processing, a single row is processed at any time, and the beginning of the second row is appended to the end of the first row, and the beginning of the third row is appended to the end of the second row etc. Thus, a *segment* in this realization corresponds to a single row.

6.2.1.2. Concurrency in Two-Dimensional Filters

In a two-dimensional recursive system, several outputs can be computed concurrently, since these computations are not mutually constrained by any precedence relation. For example, the outputs $y(5,0)$, $y(4,1)$, $y(3,2)$, $y(2,3)$, $y(1,4)$, and $y(0,5)$ are mutually independent and can be computed in parallel. Fig. 6.3 illustrates the precedence relation in a two-dimensional processing system. From these precedence relations, one can observe that the samples along the diagonals $n_1 + n_2 = c$, where c is a constant, can be computed in parallel. The locus of the sample locations, which can be computed in parallel, is a straight line with slope "-1". We refer to this locus of the mutually independent set of computations as the *concurrent computation region* (CCR). In general, the CCR corresponds to a $(N-1)$-dimensional hyperplane for a N-dimensional filter. For the one-dimensional filter, the CCR is a single sample, and for a two-dimensional filter, the CCR consists of samples along a straight line.

The parallel computation of all the samples along the diagonals can be described by the parallel loop

```
for each ($n'_1 = 0$ to $(J_1 + J_2 - 2)$ ) {
for all ($n'_2 = Max(0, n'_1 - J_1 + 1)$ to $Min(n'_1, J_2 - 1)$ ) {
```

$$y(n'_1 - n'_2, n'_2) = \sum_{\substack{i_1=0 \\ (i_1,i_2)\neq(0,0)}}^{N_4} \sum_{i_2=0}^{N_4} a_{i_1,i_2} y(n'_1 - n'_2 - i_1, n'_2 - i_2) \tag{6.5}$$

$$+ \sum_{i_1=0}^{N_4} \sum_{i_2=0}^{N_4} b_{i_1,i_2} u(n'_1 - n'_2 - i_1, n'_2 - i_2)$$

}},

where the "for all" statement represents parallel computation of all the values with respect to the corresponding loop index. The parallel loop is obtained from (6.3) by an index transformation,

$$n'_1 = n_1 + n_2,\ n'_2 = n_2\,. \tag{6.6}$$

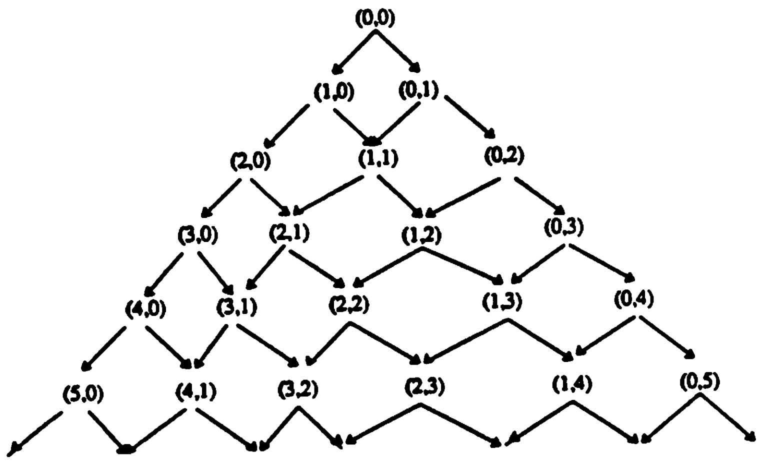


Fig. 6.3: Precedence relation in a two-dimensional recursive system.

The parallel loop in (6.5) can be executed in (J_1+J_2-1) steps using $Min(J_1,J_2)$ processors, as opposed to J_1J_2 steps needed in the sequential computation in (6.3) using a single processor. Thus, although the available concurrency is greater in a two-dimensional system, it is still bounded by the number of diagonal lines in a frame. This bound is overcome by two-dimensional look-ahead and two-dimensional block processing in section 6.3.

The representation in (6.6) exploits the maximal amount of parallelism in the two-dimensional filtering problem, and requires maximum number of processors. Often, we may exploit the parallelism only partially using reduced number of processors. Let us assume that P processors are available, and assume P to be divisible by J_2 for simplicity. We rearrange the samples of the frame in the following manner. We augment the samples in rows P through $(2P-1)$ to the end of rows 0 through $(P-1)$. Similarly, the samples in rows $2P$ through $(3P-1)$ are augmented to the end of rows P through $(2P-1)$. The rearranged frame now has P rows and J_1J_2/P columns. The rearranged data can be processed in $(J_1J_2/P+P-1)$ steps using P processors. This augmented processing corresponds to a segmented processing with segment size P, and is illustrated in Fig. 6.4.

In the subsequent sections, we exploit this inherent concurrency in the context of one-dimensional block processing, pipeline interleaving, and combination of pipeline interleaving and one-dimensional block processing.

6.2.1.3. One-Dimensional Block Processing

In one-dimensional block processing with size $1 \times L_2$, we process L_2 samples along a column in one cycle. The processing takes place in an augmented manner as shown in Fig. 6.4 with $P = L_2$. A typical way to do the one-dimensional block processing (for $L_2 = 3$) is to compute $y(0,0)$, $y(0,1)$, $y(0,2)$ in cycle 0; $y(1,0)$, $y(1,1)$, $y(1,2)$ in cycle 1; $y(2,0)$, $y(2,1)$, $y(2,2)$ in cycle 2 etc. Although this is a valid one-dimensional block processing sequence, the samples in a block do not form a set of independent computations. For example, the samples $y(0,0)$, $y(0,1)$, and $y(0,2)$ cannot be computed within a single cycle, as the computation of these samples is interdependent (although these can be computed concurrently with look-ahead computation and at the expense of hardware overhead, see section 6.3). Thus, we need to slightly alter the sequence such that the L_2 samples belong to the concurrent computation region. This is achieved by computing the L_2 samples along the diagonal concurrently. Table 6.1 shows a portion of the parallel schedule for $L_2 = 3$ using three processors, where the processor P_i operates on row i.

Table 6.1 : Concurrent One-Dimensional Block Processing

Processor versus Time Schedule								
P1	y(0,0)	y(1,0)	y(2,0)	y(3,0)	y(4,0)	y(5,0)	y(6,0)	y(7,0)
P2	-	y(0,1)	y(1,1)	y(2,1)	y(3,1)	y(4,1)	y(5,1)	y(6,1)
P3	-	-	y(0,2)	y(1,2)	y(2,2)	y(3,2)	y(4,2)	y(5,2)

In this segmented processing, the output $y(n_1,n_2)$ is processed at time index

$$I(n_1,n_2) = n_1 + n_2 + \left\lfloor \frac{n_2}{L_2} \right\rfloor (J_1 - L_2), \tag{6.7}$$

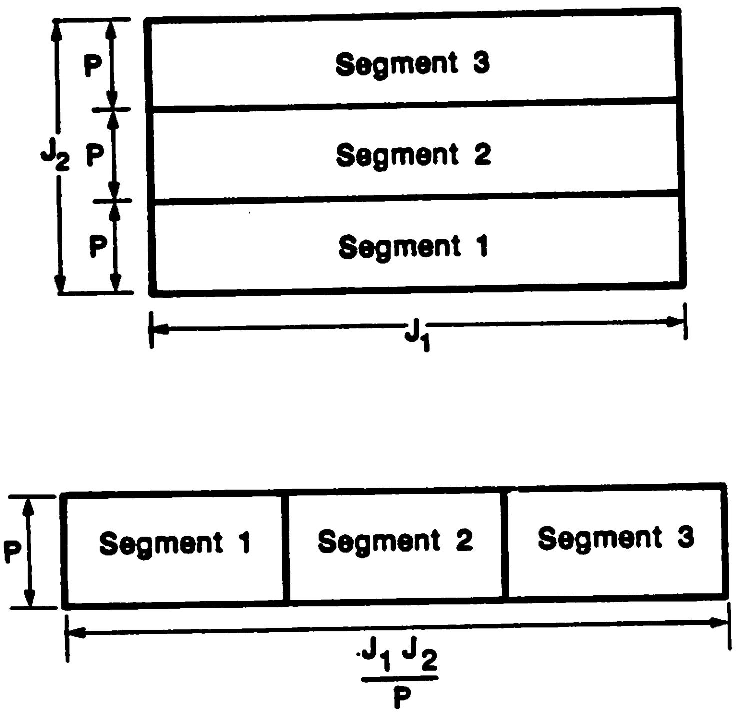


Fig. 6.4: Segment by segment processing in a two dimensional system.

where $\lfloor x \rfloor$ represents the integer part of x (or the floor function). To illustrate the use of index mapping function, consider the processing of the sample $y(14,0)$. Substituting value of $n_1 = 14$ and $n_2 = 0$, we find that this sample is processed in cycle 14. As another example, the sample $y(0,3)$ is processed in cycle 15 for a frame size of 15×15. We can derive the delay operators by examining the coefficients of n_1 and n_2 in (6.7). A row delay operator z_1^{-1} in this realization corresponds to a single delay (since the coefficient of n_1 in the index mapping function in (6.7) is 1). The column delay operator z_2^{-1} corresponds to a single delay if the two consecutive samples (of a specified column) belong to the same segment, or $(J_1 - L_2 + 1)$ delays if they belong to two consecutive segments. However, in the block implementation, single column delay operators are *not implementable*. Instead, we need to implement block column delay operators. A *block column delay operator* is defined as $z_2^{-L_2}$ or L_2 column delays at the sample rate. As an example, we derive $y(0,2)$ by delaying $y(0,5)$, and not by delaying $y(0,3)$ or $y(0,4)$ (for $L_2 = 3$). The element $y(n_1, k_2 L_2)$ is obtained by delaying the corresponding sample $y(n_1, k_2 L_2 + L_2)$ of the consecutive segment using J_1 delays (which is a line delay). This is because

$$I(n_1, k_2 L_2 + L_2) - I(n_1, k_2 L_2) = J_1 .$$

Thus the block column delay operator corresponds to J_1 delays. The row and column delay operators are shown in Fig. 6.5(a).

The samples inside a single segment are obtained by using appropriate number of row delay operators z_1^{-1}. As an example, $y(n_1 - 1, k_2 L_2 + i)$ is obtained by delaying $y(n_1, k_2 L_2 + i)$ using one delay or latch (since a row delay operator corresponds to a single delay).

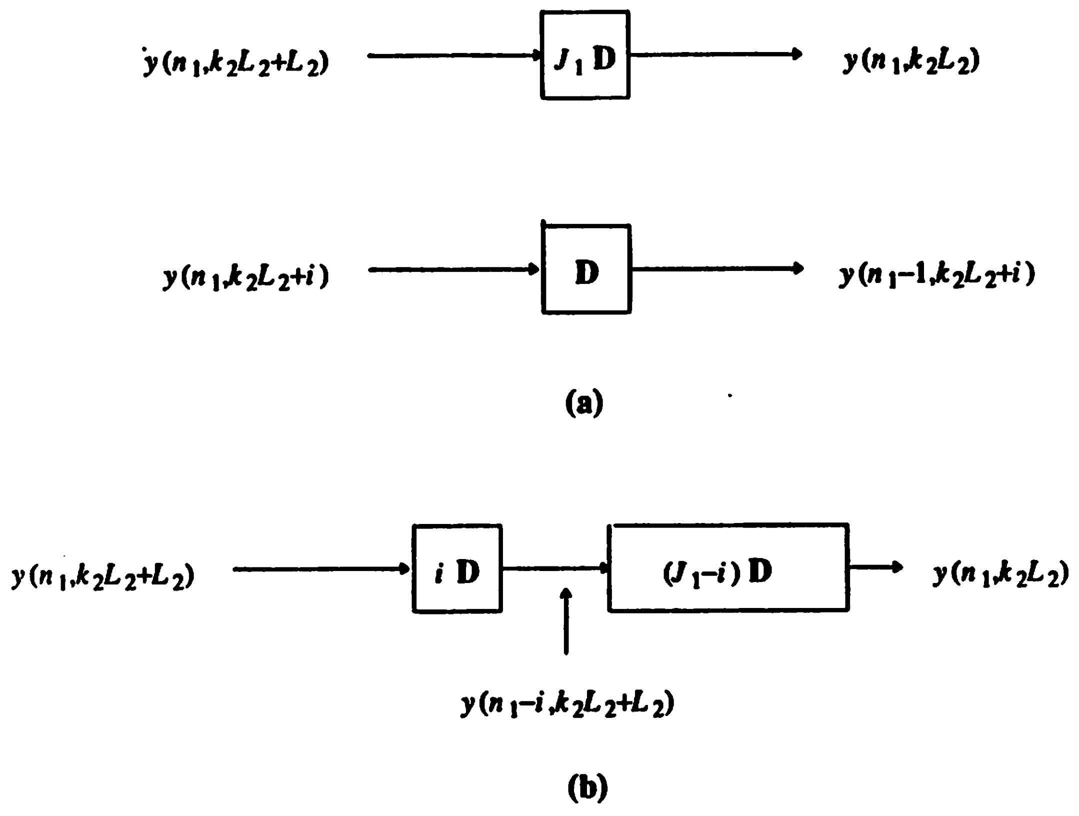


Fig. 6.5: (a) Row and Column delay operators in one dimensional block filter, (b) Quasi block column delay operator.

Now we introduce the notion of a ***quasi block column delay operator***. The J_1 delays of the block column delay operator can be split to two parts, one part of i delays, and another of $(J_1 - i)$ delays. If the input to the block column delay operator is $y(n_1, k_2L_2 + L_2)$, then the output of the first part is $y(n_1 - i, k_2L_2 + L_2)$, and the output of the second part is $y(n_1, k_2L_2)$. Using the ***delay splitting*** principle, we can derive $y(n_1, k_2L_2)$ from $y(n_1 - i, k_2L_2 + L_2)$ using $(J_1 - i)$ delays. This operation corresponds to i advance operations along the n_1 dimension and a block delay operation along the n_2 dimension (therefore the name quasi column block delay operator). This delay splitting principle of the block column delay operator is shown in Fig. 6.5(b).

A parallel hardware architecture for concurrent processing of L_2 samples can be derived using appropriate row, block column, and split block column delay operators, and is shown in Fig. 6.5(c) for the case $N_a = N_b = 1$ and $L_2 = 3$. We input L_2 samples $u(n_1, k_2L_2)$, $u(n_1, k_2L_2 + 1)$, ..., $u(n_1, k_2L_2 + L_2 - 1)$, and compute the corresponding L_2 outputs. In the snapshot of Fig. 6.5(c), the outputs $y(6,3)$, $y(5,4)$, and $y(4,5)$ are computed in parallel. These outputs are respectively given by

$$y(6,3) = a_{0,1}y(6,2) + a_{1,0}y(5,3) + a_{1,1}y(5,2) \quad (6.8a)$$
$$+ b_{0,0}u(6,3) + b_{0,1}u(6,2) + b_{1,0}u(5,3) + b_{1,1}u(5,2)$$
$$y(5,4) = a_{0,1}y(5,3) + a_{1,0}y(4,4) + a_{1,1}y(4,3) \quad (6.8b)$$
$$+ b_{0,0}u(5,4) + b_{0,1}u(5,3) + b_{1,0}u(4,4) + b_{1,1}u(4,3)$$
$$y(4,5) = a_{0,1}y(4,4) + a_{1,0}y(3,5) + a_{1,1}y(3,4) \quad (6.8c)$$
$$+ b_{0,0}u(4,5) + b_{0,1}u(4,4) + b_{1,0}u(3,5) + b_{1,1}u(3,4) .$$

Note that $y(6,2)$ is derived from $y(3,5)$ using a quasi block column delay operator with $(J_1 - 3)$ delays. Fig. 6.5(c) needs some additional clearing signals to accommodate initial conditions at the frame boundary, and these have been omitted for clarity.

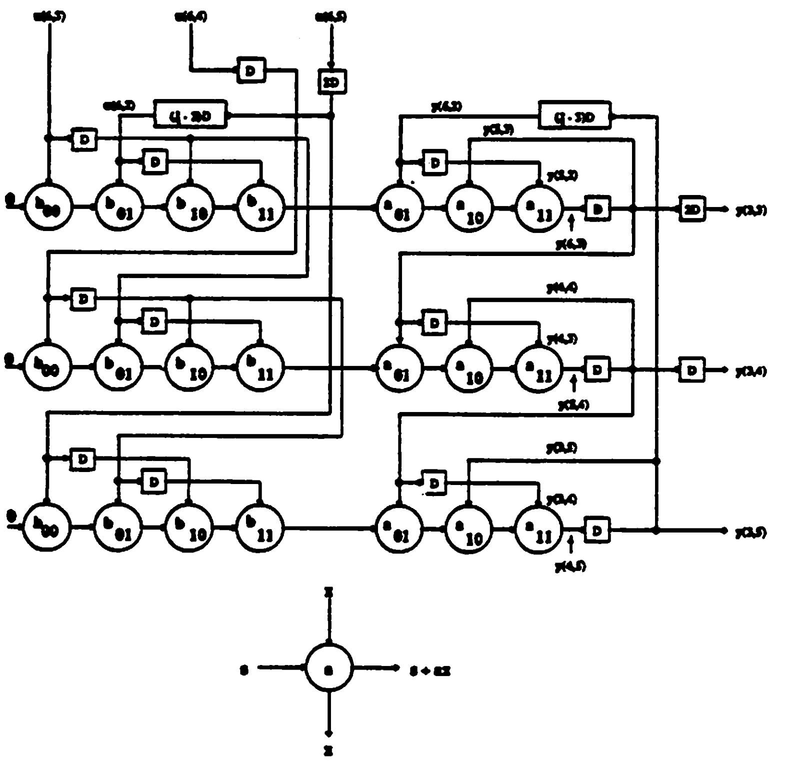
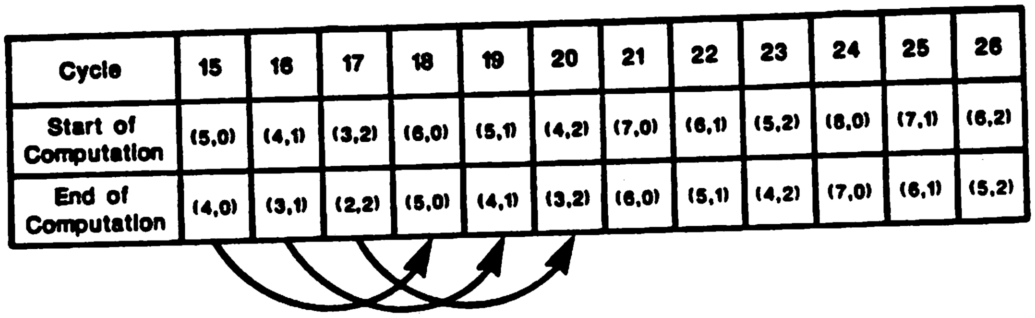


Fig. 6.5(c): Concurrent processing of a 2D recursive filter using parallel hardware for $N_a = N_b = 1$ and $L_2 = 3$.

The one-dimensional block processing of the complete frame requires (J_1J_2/L_2+L_2-1) number of cycles. This realization does not require any algorithm transformation (since these block of samples form a set of independent computations), and leads to a complexity $L_2(2(N_a+1)(N_b+1)-1)$, which is linear with respect to speedup L_2. In contrast, the L_1L_2 outputs in a two-dimensional block processing structure with block size $L_1 \times L_2$ do not form a set of independent computations, and hence require algorithm transformations. These general block structures are addressed in section 6.3.

6.2.1.4. Fine-Grain Pipelined 2D Filtering

Any set of M independent jobs requiring identical processing can be processed in an interleaved manner through a single set of resources or processing units with M stages of pipelining or buffering. The pipelined computation is hardware efficient, since it achieves an equivalent speedup of factor of M with just M buffering units, as opposed to replicating the set of resources or processors by a factor of M as in one-dimensional block processing case (note that in a M-stage pipelined realization, the system latency grows by a factor of M also. But in most applications, the system throughput is much more important than the system latency). We have already seen that the samples along the diagonals in a two-dimensional filter form a set of independent computations, and we process these samples concurrently in an interleaved manner. One should note that the M-stages of pipelining here is at the processor level, i.e. each set of processing units (which consists of one multiply and few add operations) is pipelined at M levels (as opposed to at word level as in the block filter structures). This is also often referred to as two-level pipelining in the literature.



Cycle	15	16	17	18	19	20	21	22	23	24	25	26
Start of Computation	(5,0)	(4,1)	(3,2)	(6,0)	(5,1)	(4,2)	(7,0)	(6,1)	(5,2)	(8,0)	(7,1)	(6,2)
End of Computation	(4,0)	(3,1)	(2,2)	(5,0)	(4,1)	(3,2)	(6,0)	(5,1)	(4,2)	(7,0)	(6,1)	(5,2)

Fig. 6.6: A partial schedule for the pipelined filter in Fig. 6.2 for $M = 3$ with row by row processing.

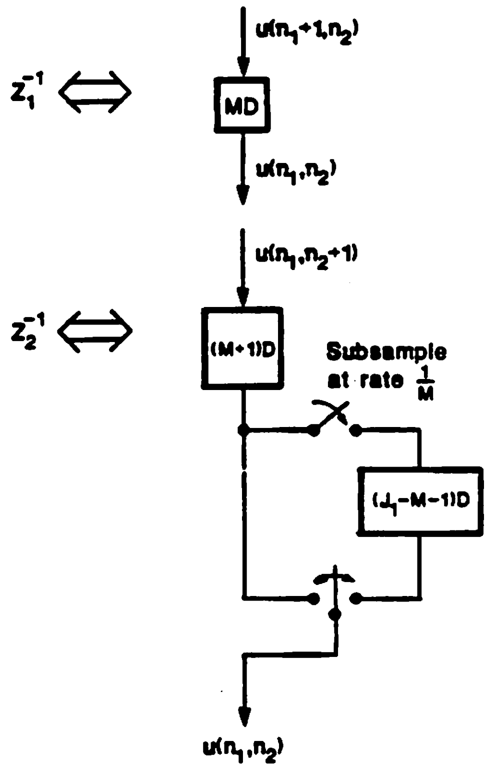


Fig. 6.7: Row and Column delay operators in a M-stage pipelined two dimensional recursive digital filter.

Now consider pipelining the recursive state update loop of Fig. 6.2 by M stages by introducing $(M-1)$ additional latches or delays (recall that the state variables here refer to the outputs $y(n_1,n_2)$). In this realization, the output of a computation is available only after M cycles (see the schedule of Fig. 6.6). In Fig. 6.6, the computation of $y(5,0)$ begins in cycle 15 and ends in cycle 18 (for $M = 3$). Hence the computation of $y(6,0)$ can begin only in cycle 18 (since this computation can begin only after $y(5,0)$ is available). If we process the samples row by row, then the hardware will be utilized only one third of the time. However as described in the context of parallel hardware, there are enough concurrent tasks in two-dimensional system, and these concurrent tasks can be processed in an interleaved manner with full hardware efficiency.

For the case when $M = 3$, we can process the samples along the diagonals of the segment consisting of one segment of M rows in an interleaved manner (note that P in Fig. 6.4 corresponds to M for this case). Since the computations $y(n_1,0)$, $y(n_1-1,1)$, and $y(n_1-2,2)$ belong to the concurrent computation region, these can be processed in an interleaved manner as illustrated in the schedule of Fig. 6.6. While we wait for the computation of $y(5,0)$ to finish, we can begin computation of $y(4,1)$ and $y(3,2)$. Similarly the computations $y(6,0)$, $y(5,1)$, and $y(4,2)$ can be concurrently processed, etc. This way, we can obtain full hardware utilization by processing a segment of M rows simultaneously in a skew interleaved manner.

In the pipelined implementation, some cycles are wasted at the beginning and end of the computation for setting up of the pipeline (or skewing the samples) and for flushing the pipeline (or deskewing the samples) respectively. The total number of wasted cycles is $M(M-1)$, of which $\frac{M(M-1)}{2}$ is wasted at the beginning and $\frac{M(M-1)}{2}$ is wasted at the end of computation of the complete frame. Tables 6.2 and 6.3 respectively illustrate the wasted cycles at the beginning and end of of a 15×15 size frame.

Table 6.2: Initial Portion of the Schedule

Cycle	0	1	2	3	4	5	6	7	8
Output	y(0,0)	-	-	y(0,1)	y(1,0)	-	y(2,0)	y(1,1)	y(0,2)

Table 6.3: Final Portion of the Schedule

Cycle	222	223	224	225	226	227	228	229	230
Output	y(14,12)	y(13,13)	y(12,14)	-	y(14,13)	y(13,14)	-	-	y(14,14)

The hardware utilization factor of this architecture is given by

$$\eta = \frac{J_1J_2}{J_1J_2+M(M-1)} \cdot \tag{6.9}$$

With the interleaved processing, the index mapping function is characterized by

$$I(n_1,n_2) = n_1M + n_2(M+1) + \lfloor \frac{n_2}{M} \rfloor M(J_1-1-M), \tag{6.10}$$

where $\lfloor x \rfloor$ represents the floor function, and $J_1 \times J_2$ is the frame size. We use this index mapping function to derive the implementable delay operators in the pipeline interleaved realization. Because of the skew pipelined interleaved processing of the rows, the processing periods of the two consecutive samples in a row are separated by M time indices (since the coefficient of n_1 is M in (6.10)). The operator z_1^{-1} corresponds to M delays or latches. This can also be verified from the schedule of Fig. 6.6,

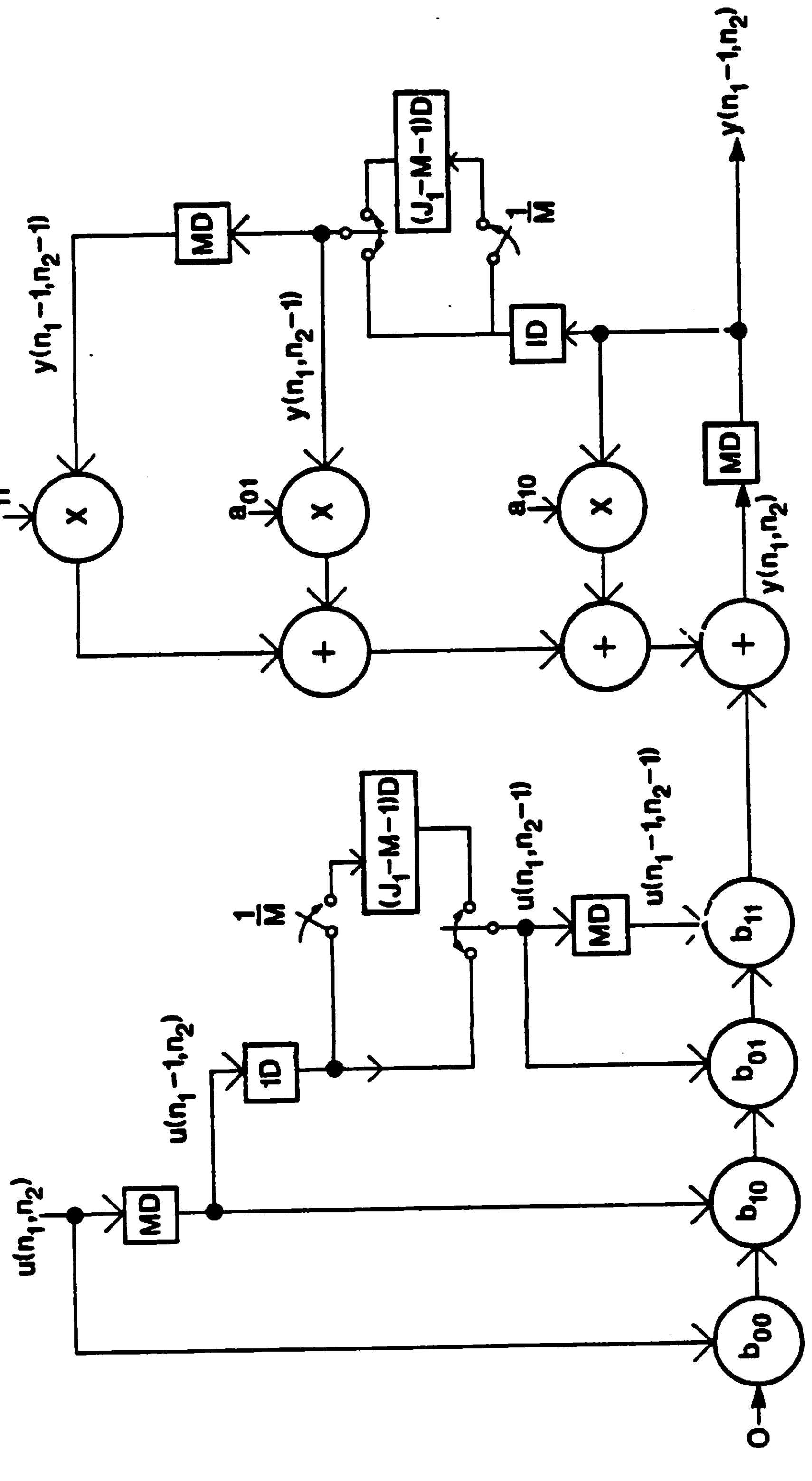


Fig. 6.8: Block diagram of a M-stage pipelined direct form 2D recursive filter for $N_a = N_b = 1$. The latches inside the loops are *retimed* or appropriately distributed to pipeline the loop. The pipelining latches in the feed forward portion have not been shown for clarity.

which shows processing of sample (5,0) in cycle 15 and of sample (6,0) in cycle 18 for $M = 3$ case. Similarly, the operator z_2^{-1} corresponds to $(M+1)$ delays if the two consecutive elements of a column belong to the same segment. However, if they belong to two consecutive segments, then they are separated by $(M+1+M(J_1-M-1))$ time indices (see (6.10) also). One way of implementing the z_2^{-1} operator is to have a two-way multiplexed delayed path, one with $(M+1)$ delay operators and the other with $(M+1+M(J_1-M-1))$ delay operators. However, this implementation is inefficient. Notice that we need to store only the top most row outputs in each segment (since the other outputs can be derived from these outputs). Hence the z_2^{-1} operator can be efficiently realized with $(M+1)$ delay operators followed by a two-way multiplexed delayed path, one with no delay and the other with (J_1-M-1) *M–slow* delays (i.e. by subsampling the output samples at rate $\frac{1}{M}$), leading to considerable saving in required number of memory elements. These delay operators are depicted in Fig. 6.7, and with the use of these operators, a block diagram of an M-stage pipelined architecture of a direct form 2D filter for $N_a = N_b = 1$ is shown in Fig. 6.8. The innermost loop now has M delays, and these delays can be redistributed to pipeline the multiplier and adders by M stages. The implementable delay operator in the pipelined two-dimensional filter architecture is derived by appropriate indexing of the input samples and without any hardware overhead unlike in one dimensional systems, where this operator was derived by algorithm transformation and at the expense of hardware increase [4-7].

The improved iteration bound is actually achieved by using the technique of *retiming* [42] or *cutset transformation* [43]. The retiming process involves moving the delays around the feedback loop in such a way that the number of delays in any loop remains unaltered, and the actual period is as close as possible to the iteration bound of the computation graph. A simple example of system retiming is illustrated in Fig. 6.9. The iteration period bound for the realization in Fig. 6.9(a) is $\frac{(T_M + T_A)}{3}$, whereas the actual iteration period is $(T_M + T_A)$, where T_M and T_A represent computation time of the multiplier and adder blocks M and A respectively. The iteration period for an equivalent retimed realization in Fig. 6.9(b) (obtained after redistributing the delays) is $Max(T_{M_1}, T_{M_2}, T_{M_3} + T_A)$. Here, the multiplier component is broken into three components M_1, M_2 and M_3. If the computational latencies of blocks M_1, M_2, and M_3 and A are identical, then this realization has an iteration period equal to the iteration bound.

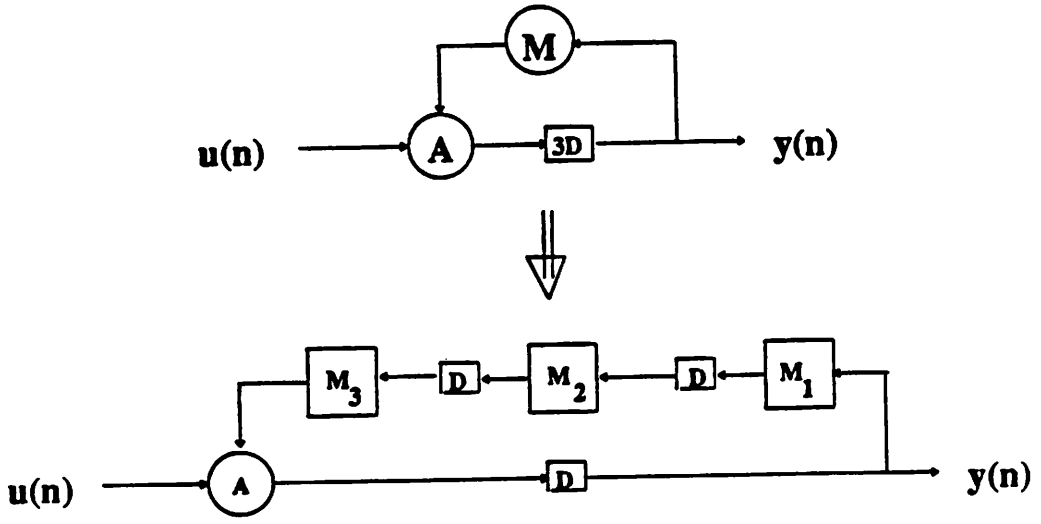


Fig. 6.9(a): A computation graph, *M* and *A* respectively represent multiply and add operations, (b) An equivalent computation graph obtained using the *retiming* technique, the multiplier block has been decomposed to three blocks.

6.2.2. Pipelining and One-Dimensional Block Processing

We can pipeline the loop of a one-dimensional block architecture with block size $1 \times L_2$ to get a speedup by a factor of $L_2 M$ as compared with sequential processing. This pipelined one-dimensional block structure can be realized without requiring any algorithm transformation (by performing computations within the concurrent computation region).

Here M sets of L_2 independent samples are processed in an interleaved manner. With M loop pipeline stages, each computation is completed in M cycles, and the implementation is M-way interleaved. For example, if we start processing the samples $y(3,0)$, $y(2,1)$ in cycle 6 (for $L_2 = 2$), this computation will be completed in cycle 8 for $M = 2$. The computation of the samples $y(4,0)$, $y(3,1)$ can begin in cycle 8 (since, these computations can begin only after the samples $y(3,0)$ and $y(2,1)$ are available). However, we can fill up the pipeline by interleaving the computations of the samples $y(1,2)$, $y(0,3)$ in cycle 7 (which will be available in cycle 9, and $y(2,2)$, $y(1,3)$ in cycle 9, etc. Thus we process $L_2 M$ rows in an interleaved parallel manner. One segment in this realization corresponds to $L_2 M$ rows, i.e. $P = L_2 M$ in Fig. 6.4. Table 6.4 shows the initial portion of the schedule for $L_2 = 2$ and $M = 2$.

Table 6.4 : Pipelined One-Dimensional Block Processing

Processor versus Time Schedule										
P1	y(0,0)	-	y(1,0)	-	y(2,0)	y(0,2)	y(3,0)	y(1,2)	y(4,0)	y(2,2)
P2	-	-	y(0,1)	-	y(1,1)	-	y(2,1)	y(0,3)	y(3,1)	y(1,3)

The index mapping function for this pipeline interleaved one-dimensional block processor is given by

$$I(n_1,n_2) = n_1M + n_2M + \lfloor \frac{n_2}{L_2} \rfloor + \lfloor \frac{n_2}{L_2M} \rfloor M\,(J_1 - L_2M - 1) \qquad (6.11)$$

which reduces to (6.7) and (6.10) as special cases. In this processing, a pipelined row delay operator corresponds to M delays, and a pipelined block column delay operator (that is L_2-slow column delay operator, which represents $z_2^{-L_2}$ at the sample rate) corresponds to $(L_2M + 1)$ delays if the two samples separated by a block belong to the same segment, or to $(L_2M + 1 + M(J_1 - L_2M - 1))$ delays if the two samples are spaced L_2 distance apart and lie on the same column, but belong to two consecutive segments. This implementable pipelined block column delay operator can be implemented with (L_2M+1) delays, followed by a multiplexed path, one with no delay, and the other with $(J_1 - L_2M - 1)$ M-slow delays. However, we can reduce the number of memory locations further by implementing the block column delay operator with one delay, followed by L_2 M-slow delays, followed by a multiplexed path, one with no delay, and the other with $(J_1 - L_2M - 1)$ M-slow delays. The quasi block column delay operators can also be interpreted in a manner similar to that in section 6.2.1.3. To derive $y(n_1,k_2L_2)$ from $y(n_1 - i,k_2L_2 + L_2)$, we need one delay followed by $(J_1 - L_2M + L_2 - i - 1)$ M-slow delays (i.e. the inputs to these are delays subsampled at a rate $1/M$) if the samples belong to two consecutive segments, and one delay followed by $(L_2 - i)$ M-slow delays if the samples belong to the same segment. This is because,

$$I(n_1 - i,k_2L_2+L_2) - I(n_1,k_2L_2) = \begin{cases} 1 + M(J_1 - L_2M + L_2 - i - 1) & \text{for same segment} \\ 1 + M(L_2 - i) & \text{for consecutive segments}. \end{cases}$$

The row, block column, and quasi block column delay operators are illustrated in Fig. 6.10(a). Fig. 6.10(b) shows the recursive portion of a pipelined one-dimensional block processor for $L_2 = 2$ and $M = 2$, and for $N_a = N_b = 1$. Each loop has at least 2 delays, and two sequences of tasks (each task of length 2) are being processed in an interleaved manner. The impressive performance of pipelining can only be achieved after retiming the flow graph, i.e. after redistributing the two loop delays to pipeline the multiplier and the two adders. The snap shot shows processing of $y(6,4)$ and $y(5,5)$ interleaved with processing of $y(3,6)$ and $y(2,7)$. In the figure, $z(n_1,n_2)$ represents the effects of the current input term $u(n_1,n_2)$ and its delayed versions. The output $y(6,3)$ is derived from $y(4,5)$ using a quasi block column delay operator.

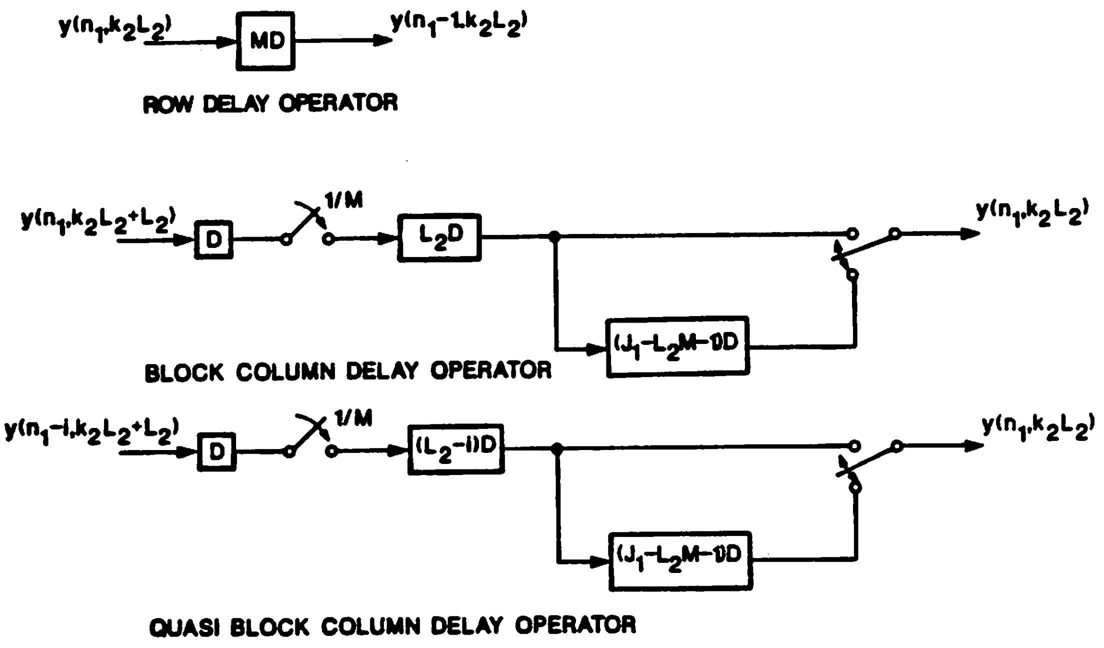


Fig. 6.10(a): Delay operators in pipelined one dimensional block processor.

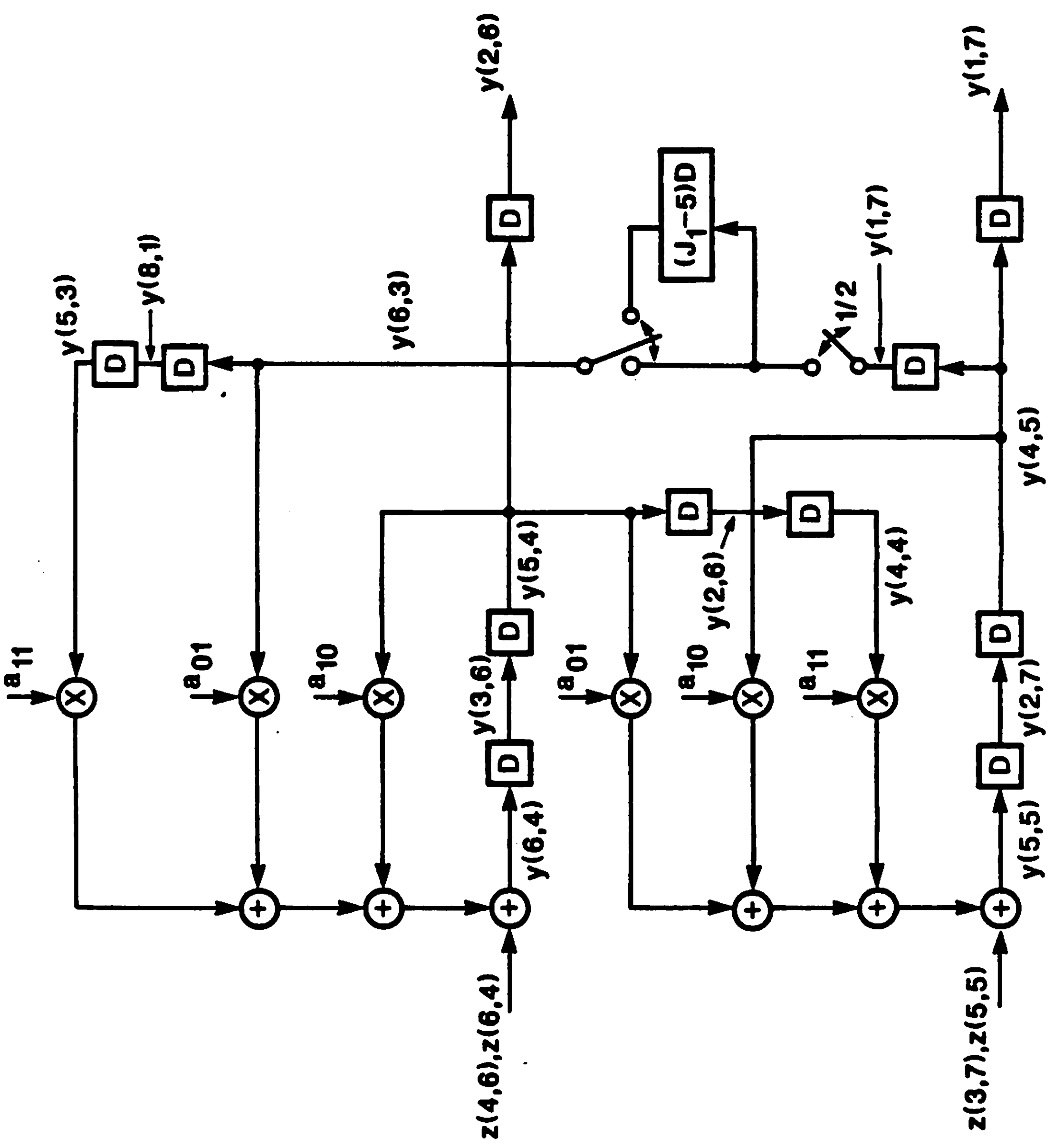


Fig. 6.10(b): Block diagram of a pipelined one dimensional block processor.

6.2.3. Local State Space Form 2D Filters

The transfer function of (6.2) can be equivalently represented in terms of a local state space description given by [44-45]

$$\begin{bmatrix} \mathbf{h}(n_1+1,n_2) \\ \mathbf{v}(n_1,n_2+1) \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{h}(n_1,n_2) \\ \mathbf{v}(n_1,n_2) \end{bmatrix} + \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} u(n_1,n_2) \tag{6.12a}$$

$$y(n_1,n_2) = \mathbf{c}_1\mathbf{h}(n_1,n_2) + \mathbf{c}_2\mathbf{v}(n_1,n_2) + du(n_1,n_2)\,, \tag{6.12b}$$

where the horizontal state $\mathbf{h}(n_1,n_2)$ is $N_1\times1$, the vertical state $\mathbf{v}(n_1,n_2)$ is $N_2\times1$, the input $u(n_1,n_2)$ and output $y(n_1,n_2)$ are scalars, and all other matrices are of appropriate dimensions. If the order of horizontal states N_1 is chosen to be same as N_a, then a local state space realization with the order of vertical states (N_2) less than or equal to $2N_b$ can always be found [44].

This recursive state update representation takes the horizontal and vertical states at sample point (n_1,n_2) and computes the horizontal state at (n_1+1,n_2) and vertical state at (n_1,n_2+1). Similar to the direct form filter case, many independent computations also exist in the state space form filter, and this concurrency can be exploited in the context of either parallel (i.e. one-dimensional block processing) and/or pipelined hardware implementations. In the context of parallel hardware realization, we can process the L_2 output samples $y(n_1,k_2L_2)$, $y(n_1-1,k_2L_2+1)$, ..., and $y(n_1-L_2+1,k_2L_2+L_2-1)$ in parallel using $O(L_2)$ processors (which is a linear increase with respect to L_2). This one-dimensional block implementation is useful for software programmable coarse-grain parallel processor implementations. Alternatively, we can exploit this concurrency in the context of pipeline interleaving to obtain a hardware efficient custom VLSI realization.

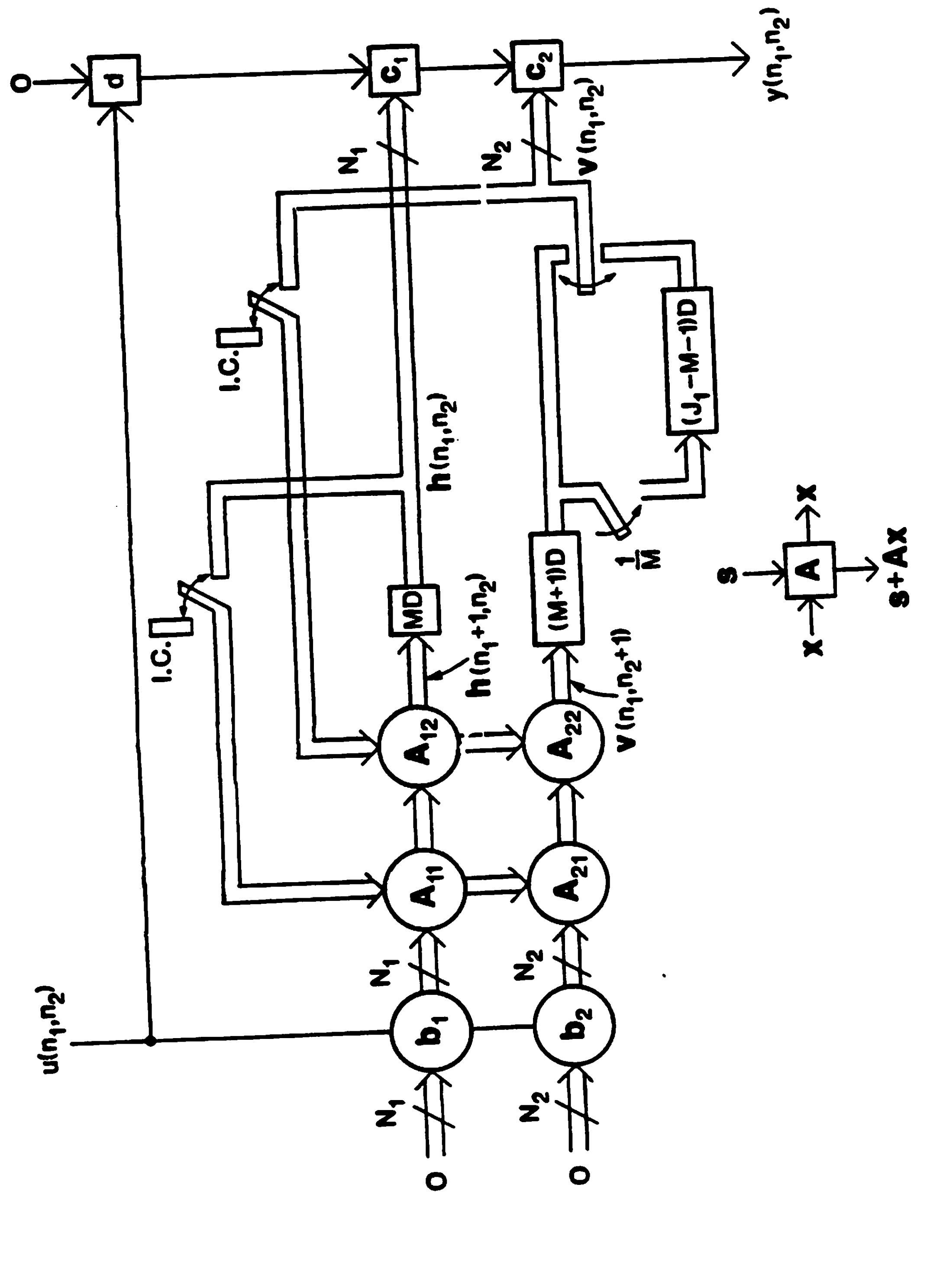


Fig. 6.11(a): Block diagram of a *M*-stage pipelined local state space two dimensional recursive digital filter. Pipelining latches in the feed forward sections have not been shown. Also note that the multiplication operations with A_{11}, A_{12}, A_{21}, and A_{22} are performed simultaneously, and not sequentially. IC corresponds to initial condition.

Cycle	15	16	17	18	19	20	21	22	23	24	25	26
Start of Computation	h(5,0) v(5,0)	h(4,1) v(4,1)	h(3,2) v(3,2)	h(6,0) v(6,0)	h(5,1) v(5,1)	h(4,2) v(4,2)	h(7,0) v(7,0)	h(6,1) v(6,1)	h(5,2) v(5,2)	h(8,0) v(8,0)	h(7,1) v(7,1)	h(6,2) v(6,2)
End of Computation	h(5,0) v(4,1)	h(4,1) v(3,2)	h(3,2) v(2,3)	h(6,0) v(5,1)	h(5,1) v(4,2)	h(4,2) v(3,3)	h(7,0) v(6,1)	h(6,1) v(5,2)	h(5,2) v(4,3)	h(8,0) v(7,1)	h(7,1) v(6,2)	h(6,2) v(5,3)

Fig. 6.11(b): A partial schedule for the pipelined local state space filter in Fig. 6.11(a) for $M = 3$.

In the pipelined realization with M loop pipeline stages, if we start the processing of the sample point (n_1,n_2) using $\mathbf{h}(n_1,n_2)$ and $\mathbf{v}(n_1,n_2)$ at time index n, then the states $\mathbf{h}(n_1+1,n_2)$ and $\mathbf{v}(n_1,n_2+1)$ will be available only at time index $(n+M)$. We cannot begin the computation at the sample point (n_1+1,n_2) until time index $(n+M)$. However, just as in the direct form filter case, we can fill up the pipeline by processing M rows concurrently in a skew pipelined interleaved manner, and improve the iteration bound by about a factor of M. In the pipelined implementation, the state update operations (for the first segment) at sample points $(n_1,0)$, $(n_1-1,1)$, ..., and $(n_1-M+1,M-1)$ are performed concurrently in an interleaved manner. The efficiency and the index mapping function of this implementation are same as in (6.9) and (6.10) respectively. The delay operators in Fig. 6.7 also hold good for this case. In the local state space realization, the updated horizontal states are used up within the same segment, and do not need to be stored. However, we need to store the vertical states of the top row only for each segment to be used for processing of the next segment, and these are stored using M-slow latches. A pipelined architecture for local state space filter is shown in Fig. 6.11(a) and the corresponding partial schedule is shown in Fig. 6.11(b) for $M=3$. Once again, we will need to retime the flow graph to redistribute the delays for pipelining the multiply/add operations. We can also combine pipeline interleaving and parallelism (or one-dimensional block processing) as described in section 6.2.2, and the index mapping function of (6.11) and the corresponding operators hold good for this case.

6.3. TWO-DIMENSIONAL INCREMENTAL BLOCK FILTER

In this section, we study two-dimensional block implementation of direct form and local state space form two dimensional recursive digital filters. In a two-dimensional block (or simply block) implementation with block size $L_1 \times L_2$, L_1L_2 outputs are computed in each cycle. Let the column by column representation of a block of outputs be denoted by

$$y^{(L_1,L_2)}(k_1L_1,k_2L_2) = \begin{bmatrix} y^{(1,L_2)}(k_1L_1,k_2L_2) \\ y^{(1,L_2)}(k_1L_1+1,k_2L_2) \\ \vdots \\ y^{(1,L_2)}(k_1L_1+L_1-1,k_2L_2) \end{bmatrix} \tag{6.13a}$$

and a row by row representation be given by

$$\rho y^{(L_1,L_2)}(k_1L_1,k_2L_2) = \begin{bmatrix} y^{(L_1,1)}(k_1L_1,k_2L_2) \\ y^{(L_1,1)}(k_1L_1,k_2L_2+1) \\ \vdots \\ y^{(L_1,1)}(k_1L_1,k_2L_2+L_2-1) \end{bmatrix} \tag{6.13b}$$

where

$$y^{(1,L_2)}(k_1L_1,k_2L_2) = \Big[y(k_1L_1,k_2L_2), y(k_1L_1,k_2L_2+1), \ldots.., y(k_1L_1,k_2L_2+L_2-1)\Big]^T \tag{6.13c}$$

$$y^{(L_1,1)}(k_1L_1,k_2L_2) = \Big[y(k_1L_1,k_2L_2), y(k_1L_1+1,k_2L_2), \ldots.., y(k_1L_1+L_1-1,k_2L_2)\Big]^T \tag{6.13d}$$

and ρ is the column by column to row by row transformation operator.

Thus, block processing with a block size $L_1 \times L_2$ improves the iteration period (and the sample rate) by a factor of L_1L_2 (see (6.1)). Often in the literature, L_1 and L_2 are optimized with respect to the filter order to minimize the multiplication complexity of the realization. But, it should be noted that the parameters L_1 and L_2 are independent of the filter order, and are dictated by the speedup required, or equivalently the number of processing elements available. In the sequel, we assume that J_1L_2 samples of L_2 consecutive rows are processed in J_1/L_1 time steps and the J_1J_2 samples of the complete frame are processed in J_1J_2/L_1L_2 number of such steps. For simplicity, we assume J_1 and J_2 to be respectively divisible by L_1 and L_2 (otherwise, 0 samples can be appended to the end of the frame to satisfy this). For this case, a segment corresponds to L_2 rows, since L_2 rows are processed concurrently. The index mapping function for the block filter realization is given by

$$I(n_1,n_2)=\lfloor \frac{n_1}{L_1} \rfloor + \frac{J_1}{L_1} \lfloor \frac{n_2}{L_2} \rfloor . \tag{6.14}$$

The delay operators for the block realization are shown in Fig. 6.12. A row delay operator corresponds to a single L_1–*slow* latch, and a column delay operator corresponds to J_1/L_1 L_2–*slow* latches.

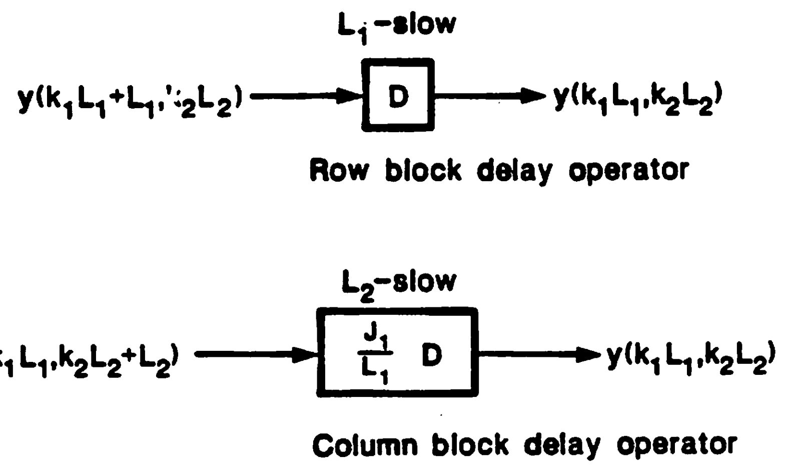


Fig. 6.12: Row and column delay operators in two-dimensional block realizations

In section 6.2, it was pointed out that the L_2 samples $y(n_1,k_2L_2)$, $y(n_1-1,k_2L_2+1)$, ..., $y(n_1-L_2+1,k_2L_2+L_2-1)$ can be processed in parallel without using any algorithm transformation and with linear increase in hardware. In contrast, we need to use algorithm transformation for block filters if the block of samples of size $L_1 \times L_2$ encompass a *rectangular region* (that is if both L_1 and L_2 are greater than unity). In this transformation, the necessary level of concurrency is derived by using the *look-ahead computation* techniques [4], i.e. iterating the original recursion as many times as desired (of course, at the expense of an increase in hardware), and this concurrency is exploited to obtain implementable row and column block delay operators in the block filter structure. Thus, unlike pipeline interleaving approach or one-dimensional block filtering with $1 \times L_2$ blocks of samples belonging to the concurrent computation region, two-dimensional block filtering approach leads to an increase in hardware, and is *not area efficient for high sampling rate realizations*.

Several block filters have been proposed for block implementation of direct form as well as local state space form filters. The block filters in [21-29] require a square multiplication complexity with respect to the block size. The parallel block filter proposed in [30] requires much higher complexity than the block state case [21-29]. In this section, we derive our new *incremental block filters* for implementation of direct form as well as local state space form filters of multiplication complexity much less than the existing structures. The direct form incremental block filter is an extension of the one dimensional structure in [19-20], and the local state space form incremental block filter is an extension of the one dimensional incremental block state filter presented in [5]. We derive the look-ahead computation principle for 2D recursive computations, and use this to derive the direct form incremental block filter structure. The state update computation is same for the incremental block state and the existing block state structures (for block implementation of local state space filters), but not the output computation. We reformulate the state update computation for the local state space based block filters, which are more appealing and easily extensible to higher dimensions. The output computation in our incremental block state filter is done incrementally in a sequential manner rather than all at once as in the block state case.

6.3.1. Direct Form 2D Incremental Block Filter

In the direct form block filter in [24-28], all the block of L_1L_2 outputs are computed using outputs of past blocks, i.e. $y^{(L_1L_2)}(k_1L_1+L_1,k_2L_2+L_2)$ is computed in terms of $y^{(L_1L_2)}(k_1L_1+L_1,k_2L_2)$, $y^{(L_1L_2)}(k_1L_1,k_2L_2+L_2)$, and $y^{(L_1L_2)}(k_1L_1,k_2L_2)$. This output update operation requires that the L_1L_2 outputs be updated in each block. Since the block output update operation is expensive (as we will see later), this leads to an implementation complexity $O(L_1^2L_2^2)$. However, for the case where $L_1>N_a$, and $L_2>N_b$, we can use an incremental output computation technique [5,19-20], where we need to update $(N_aL_2+N_bL_1-N_aN_b)$ outputs only, and use these to compute the remaining $(L_1-N_a)(L_2-N_b)$ outputs in a non-recursive or sequential manner with total multiplication complexity $O(Max(L_1^2L_2,L_1L_2^2))$. For the case, where either $L_1<N_a$ or $L_2<N_b$ or both, the multiplication complexity is inherently $O(L_1^2L_2^2)$, which may not be critical since L_1 and L_2 are small.

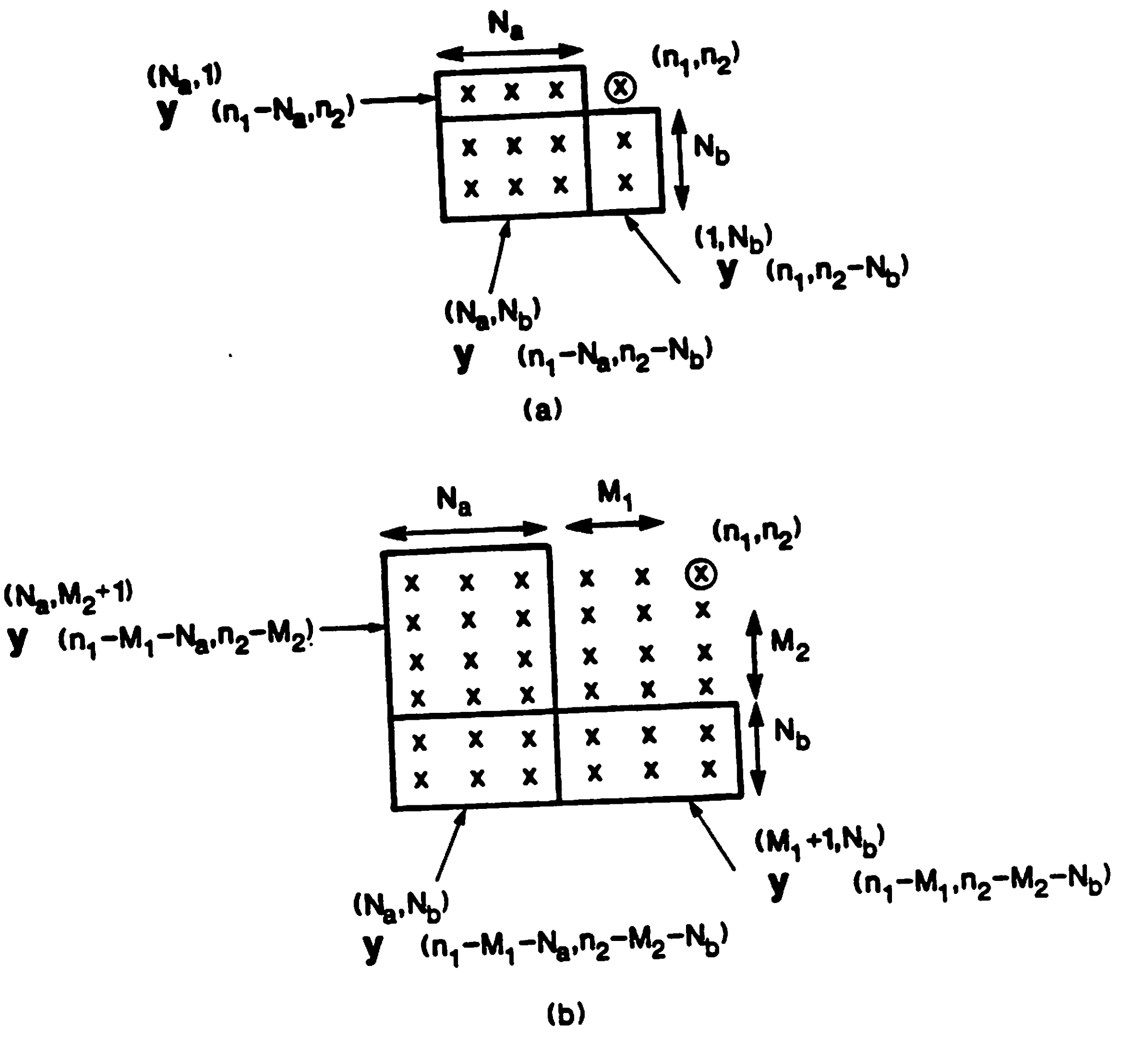


Fig. 6.13: Illustration of look-ahead in direct form 2D recursive computations: (a) traditional computation (b) look-ahead version where the output is computed bypassing $(M_1+1)(M_2+1)-1$ neighboring outputs

In the direct form representation in (6.3), $y(n_1,n_2)$ is expressed in terms of neighboring $(N_a+1)(N_b+1)-1$ outputs, i.e. $y(n_1,n_2)$ is expressed in terms of $y^{(N_a,1)}(n_1-N_a,n_2)$, $y^{(N_a,N_b)}(n_1-N_a,n_2-N_b)$, and $y^{(1,N_b)}(n_1,n_2-N_b)$ (see Fig. 6.13(a)). Now consider expressing $y(n_1,n_2)$ as a function of $(N_a+M_1+1)(N_b+M_2+1)-1$ outputs, M_1 samples apart in n_1 direction and M_2 samples in n_2 direction, i.e. as a function of $y^{(N_a,M_2+1)}(n_1-M_1-N_a,n_2-M_2)$, $y^{(N_a,N_b)}(n_1-M_1-N_a,n_2-M_2-N_b)$, and $y^{(M_1+1,N_b)}(n_1-M_1,n_2-M_2-N_b)$ (see Fig. 6.13(b)). The look-ahead iteration is given by

$$
\begin{aligned}
y(n_1,n_2) = &\sum_{\substack{i_1=0 \\ (i_1,i_2)\neq(0,0)}}^{N_a} \sum_{i_2=0}^{N_b} \left[\sum_{j_1=i_1}^{N_a}\sum_{j_2=i_2}^{N_b} a_{j_1,j_2} r_{i_1-j_1+M_1,i_2-j_2+M_2}\right] y(n_1-i_1-M_1,n_2-i_2-M_2) \\
&+\sum_{i_1=0}^{M_1-1}\sum_{i_2=1}^{N_b}\left[\sum_{j_1=0}^{i_1}\sum_{j_2=i_2}^{N_b} a_{j_1,j_2} r_{i_1-j_1,i_2-j_2+M_2}\right] y(n_1-i_1,n_2-i_2-M_2) \\
&+\sum_{i_1=1}^{N_a}\sum_{i_2=0}^{M_2-1}\left[\sum_{j_1=i_1}^{N_a}\sum_{j_2=0}^{i_2} a_{j_1,j_2} r_{i_1-j_1+M_1,i_2-j_2}\right] y(n_1-i_1-M_1,n_2-i_2) \\
&+\sum_{i_1=0}^{M_1}\sum_{i_2=0}^{M_2} r_{i_1,i_2} z(n_1-i_1,n_2-i_2) \qquad (6.15a)
\end{aligned}
$$

where

$$
z(n_1,n_2)=\sum_{i_1=0}^{N_a}\sum_{i_2=0}^{N_b} b_{i_1,i_2} u(n_1-i_1,n_2-i_2)\,, \qquad (6.15b)
$$

and the sequence r_{n_1,n_2} is defined in appendix 7.1. In look-ahead, we compute $y(n_1,n_2)$ while bypassing its $(M_1+1)(M_2+1)-1$ neighboring past outputs. This representation is used to derive the two-dimensional direct form block filters.

Case I: $L_1 \geq N_a$ and $L_2 \geq N_b$

We divide the L_1L_2 outputs in a block into two portions, (i) $(L_1N_b+L_2N_a-N_aN_b)$ outputs $y^{(N_a,N_b)}(k_1L_1+L_1,k_2L_2+L_2)$, $y^{(L_1-N_a,N_b)}(k_1L_1+L_1+N_a,k_2L_2+L_2)$, and

$y^{(N_a,L_2-N_b)}(k_1L_1+L_1,k_2L_2+L_2+N_b)$, which are updated every cycle in a recursive manner, i.e. these outputs are *states* which are updated; (ii) the remaining $(L_1-N_a)(L_2-N_b)$ outputs $y^{(L_1-N_a,L_2-N_b)}(k_1L_1+L_1+N_a,k_2L_2+L_2+N_b)$, which are computed non-recursively in a sequential manner using these updated outputs (see Fig. 6.14(a)). The updated outputs are computed using look-ahead computation, where as the non-recursively incremented outputs are simply calculated using (6.3) and the available outputs without any algorithm transformation. For example, for the case where $L_1=6$, $L_2=8$, $N_a=2$ and $N_b=3$, we update the outputs $y^{(2,3)}(6k_1+6,8k_2+8)$, $y^{(4,3)}(6k_1+8,8k_2+8)$, and $y^{(2,5)}(6k_1+6,8k_2+11)$ using corresponding delayed block of outputs. The incremental output computation proceeds as follows. First we use these updated outputs to compute $y(6k_1+8,8k_2+11)$ using (6.3). Then we use the updated outputs and $y(6k_1+8,8k_2+11)$ to compute $y(6k_1+9,8k_2+11)$. We continue this process until we compute $y(6k_1+11,8k_2+11)$; then we compute $y(6k_1+8,8k_2+12)$ through $y(6k_1+11,8k_2+12)$. We continue until we finish the computation of all 20 incremental outputs in a sequential manner ending with $y(6k_1+11,8k_2+15)$. Thus, this incremental output computation is carried out outside the feedback loop. The recursive output update operation is performed by using (6.15) with $M_1=L_1-N_a+p$ and $M_2=L_2-N_b+q$ to update the state $y(k_1L_1+L_1+p,k_2L_2+L_2+q)$. For example with above values of L_1,L_2, N_a and N_b, (M_1,M_2) respectively correspond to (5,5) for $y(6k_1+7,8k_2+8)$, and (9,7) for $y(6k_1+11,8k_2+10)$. The general process of incremental computation is illustrated in Fig. 6.14(b), where the hashed portions represents outputs which are updated, and the blank or white portions are outputs which are computed sequentially. The numbering of the blocks represents the processing sequence.

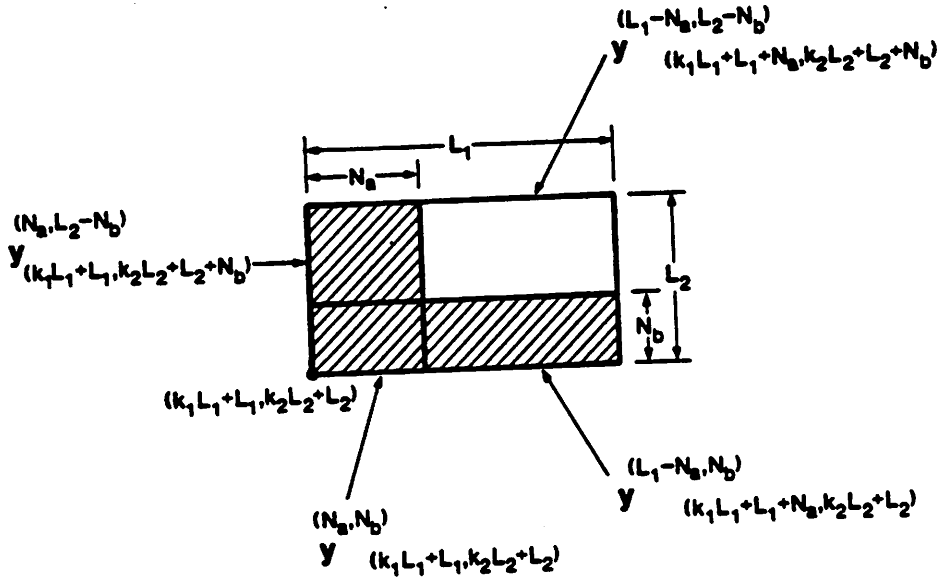
A matrix formulation for the above block output update computation can be derived to be of the form

$$\begin{aligned}\hat{\mathbf{y}}(k_1L_1+L_1,k_2L_2+L_2) = {} & \mathbf{A}_1\hat{\mathbf{y}}(k_1L_1,k_2L_2+L_2) + \mathbf{A}_2\hat{\mathbf{y}}(k_1L_1+L_1,k_2L_2) + \mathbf{A}_3\hat{\mathbf{y}}(k_1L_1,k_2L_2) \\ & + \mathbf{B}_0\mathbf{z}^{(L_1,L_2)}(k_1L_1+L_1,k_2L_2+L_2) + \mathbf{B}_1\mathbf{z}^{(L_1,L_2)}(k_1L_1,k_2L_2+L_2) \\ & + \mathbf{B}_2\mathbf{z}^{(L_1,L_2)}(k_1L_1+L_1,k_2L_2) + \mathbf{B}_3\mathbf{z}^{(L_1,L_2)}(k_1L_1,k_2L_2) \quad (6.16a)\end{aligned}$$

where

$$\hat{\mathbf{y}}(k_1L_1+L_1,k_2L_2+L_2) = \begin{bmatrix} \mathbf{y}^{(N_a,L_2-N_b)}(k_1L_1+L_1,k_2L_2+L_2+N_b) \\ \mathbf{y}^{(N_a,N_b)}(k_1L_1+L_1,k_2L_2+L_2) \\ \mathbf{y}^{(L_1-N_a,N_b)}(k_1L_1+L_1+N_a,k_2L_2+L_2) \end{bmatrix}, \qquad (6.16b)$$

and the elements of the matrices can be obtained in terms of the filter coefficients and the sequence r_{n_1,n_2} using appropriate values of M_1 and M_2 in (6.15). The vectors $\mathbf{z}^{(L_1,L_2)}(k_1L_1,k_2L_2+L_2)$, $\mathbf{z}^{(L_1,L_2)}(k_1L_1+L_1,k_2L_2)$, and $\mathbf{z}^{(L_1,L_2)}(k_1L_1,k_2L_2)$ are derived by delaying the vector $\mathbf{z}^{(L_1,L_2)}(k_1L_1+L_1,k_2L_2+L_2)$. The total complexity of this incremental direct form block implementation is $O(Max(L_1^2L_2, L_1L_2^2))$ (see appendix 7.2) as opposed to $O(L_1^2L_2^2)$ as in the block filter in [21-28].



(L_1-N_a,L_2-N_b)
$y\ (k_1L_1+L_1+N_a,k_2L_2+L_2+N_b)$
L_1
N_a
(N_a,L_2-N_b)
$y\ (k_1L_1+L_1,k_2L_2+L_2+N_b)$
L_2
N_b
$(k_1L_1+L_1,k_2L_2+L_2)$
(L_1-N_a,N_b)
$y\ (k_1L_1+L_1+N_a,k_2L_2+L_2)$
(N_a,N_b)
$y\ (k_1L_1+L_1,k_2L_2+L_2)$

Fig. 6.14(a): State update and incremental output computation in a direct form 2D block filter. The hashed portion corresponds to the recursive state update portion, and the remaining portion corresponds to the incremental output computation operation.

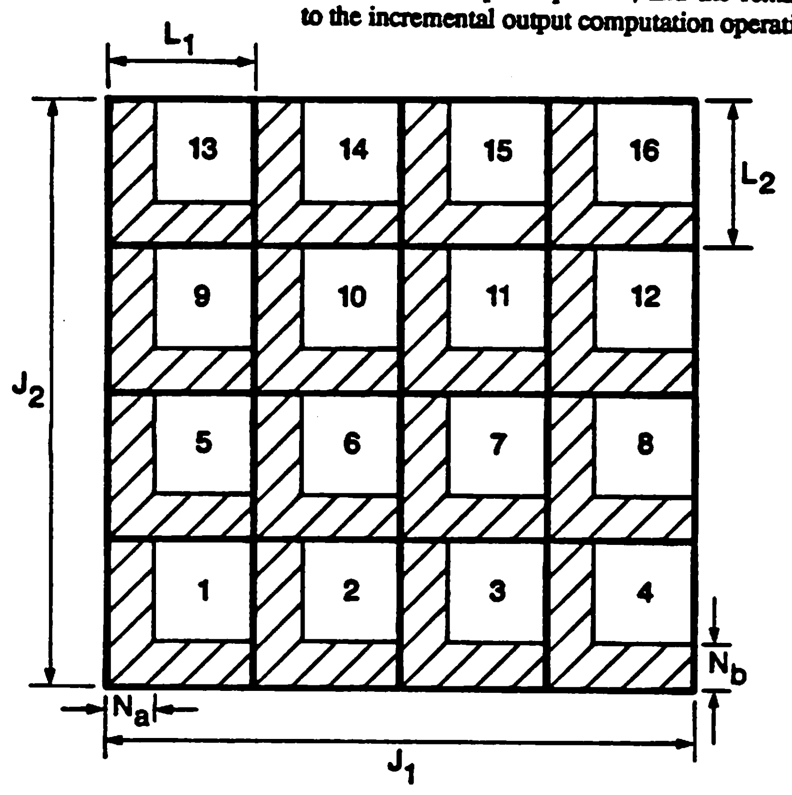


Fig. 6.14(b): Incremental Two-Dimensional block processing.

Although the formulation of (6.16) appears similar to the direct form filter formulations in [21-28], the number of outputs that need to be updated every block in these representations differ. The size of the updated states in (6.16) is $(L_1N_b+L_2N_a-N_aN_b)$, whereas the size of the outputs to be updated in [21-28] is L_1L_2. The incremental computatior permits us to update a reduced number of outputs recursively and compute the remanining outputs in a non-recursive manner, i.e. outside the recursive or feedback loop. Table 6.5 compares the multiplication complexity per sample for the standard direct form filter and the incremental filter. The standard direct form filter complexity is based on the formulae given in [25]. The incremental filter has much less complexity compared to that of standard block filter.

Table 6.5: Per Output Complexities of Direct Form Block and Incremental Block Filters

(L_1,L_2)	$(N_a=N_b=2)$		$(N_a=N_b=4)$	
	block	inc. block	block	inc. block
(8,8)	64	65	124	133
(16,16)	148	115	240	243
(32,32)	412	212	568	443
(64,64)	1324	404	1608	831
(128,128)	4684	789	5224	1601
(256,256)	17548	1557	18600	3128

Several high speed algorithms for direct form block filter implementations using fast convolution and transform techniques [35-40] have been reported [25,27]. These techniques are applicable to the incremental block filter presented in this chapter as well. For example, we can achieve a multiplication complexity identical to the fast or short convolution [40] and FFT based block filter implementation in [27] by computing the $\mathbf{B}_0$,

$\mathbf{B}_1$, $\mathbf{B}_2$, and $\mathbf{B}_3$ in (6.16a) via FFT; and $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ by short convolutions.

Case II: **$L_1<N_a$ and/or $L_2<N_b$**

In this case, we need to update all L_1L_2 states in a block by block manner, and the technique of incremental output computation is not applicable. This block state update operation can be carried out using (6.15) with $M_1=p$ and $M_2=q$ for updating the state $y(k_1L_1+L_1+p, k_2L_2+L_2+q)$. The multiplication complexity in this case is $O(L_1^2L_2^2)$ (see appendix 7.2). This complexity may not be very large, since N_a and N_b (and therefore L_1 and L_2) are small.

6.3.2. Local State Space 2D Incremental Block State Filters

A block state space recursive digital filter has been proposed in [27-29] for block implementation of the local state space model in (6.12). However, this structure has a complexity $O(L_1^2L_2^2)$ multiplications. In this section, we derive the incremental block state structure, which has a complexity $O(Max(L_1^2L_2, L_1L_2^2))$ multiplications. This structure is based on a novel output computation strategy (although the state update is same as in the block state case), and it is this novel output computation which leads to reduced complexity.

Let the column by column representation of the block of horizontal and vertical states corresponding to block size $L_1 \times L_2$ be defined by

$$\mathbf{h}^{(L_1L_2)}(k_1L_1, k_2L_2) = \begin{bmatrix} \mathbf{h}^{(1L_2)}(k_1L_1, k_2L_2) \\ \mathbf{h}^{(1L_2)}(k_1L_1+1, k_2L_2) \\ \vdots \\ \mathbf{h}^{(1L_2)}(k_1L_1+L_1-1, k_2L_2) \end{bmatrix} \tag{3.17a}$$

and the row by row representation be defined by

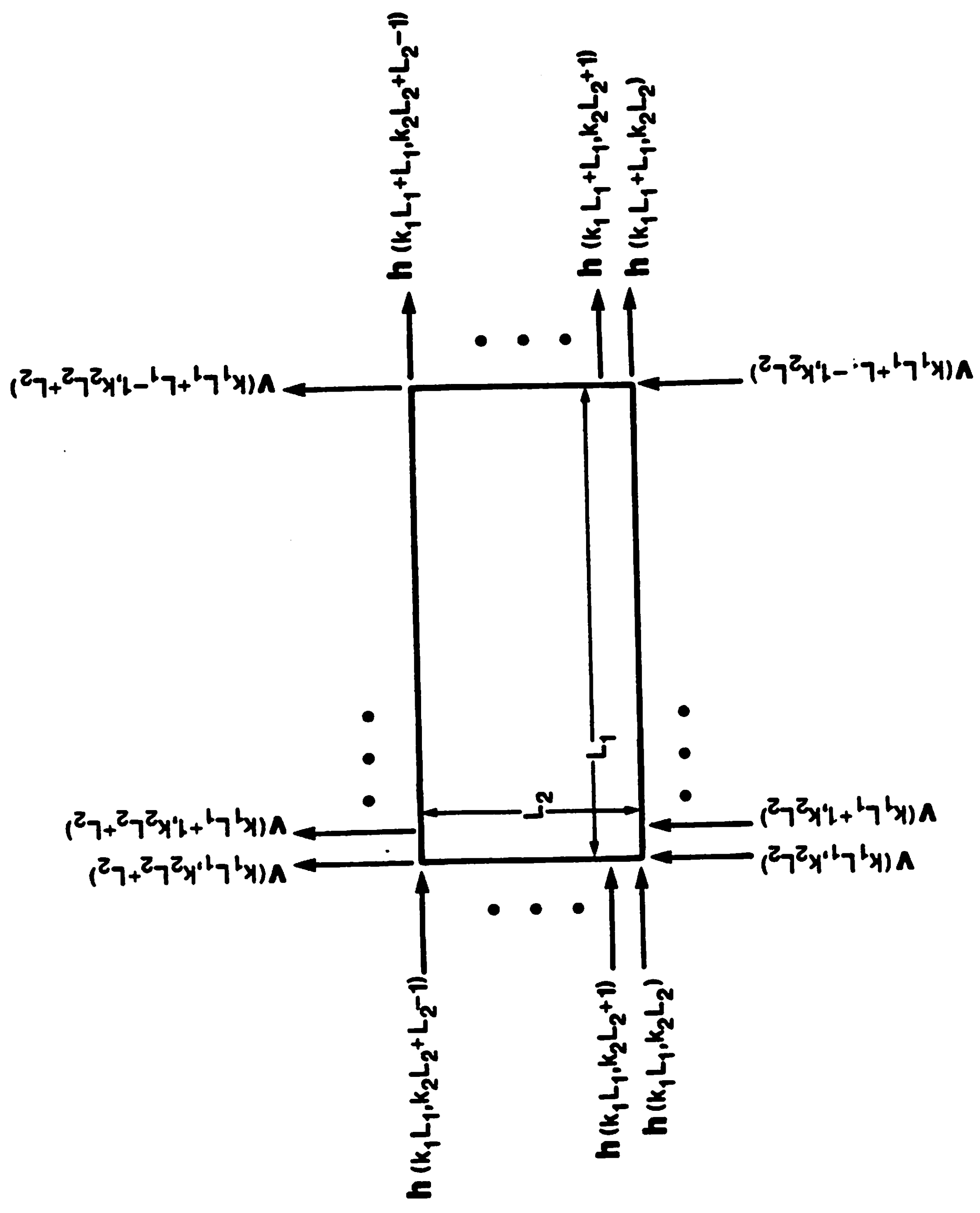


Fig. 6.15: Block by block state update operation in a 2D local state space recursive filter.

$$\rho\mathbf{h}^{(L_1,L_2)}(k_1L_1,k_2L_2)=\begin{bmatrix}\mathbf{h}^{(L_1,1)}(k_1L_1,k_2L_2)\\ \mathbf{h}^{(L_1,1)}(k_1L_1,k_2L_2+1)\\ \vdots\\ \mathbf{h}^{(L_1,1)}(k_1L_1,k_2L_2+L_2-1)\end{bmatrix} \qquad (6.17b)$$

where

$$\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)=\left[\mathbf{h}(k_1L_1,k_2L_2),\mathbf{h}(k_1L_1,k_2L_2+1),\ldots,\mathbf{h}(k_1L_1,k_2L_2+L_2-1)\right]^T \qquad (6.17c)$$

$$\mathbf{h}^{(L_1,1)}(k_1L_1,k_2L_2)=\left[\mathbf{h}(k_1L_1,k_2L_2),\mathbf{h}(k_1L_1+1,k_2L_2),\ldots,\mathbf{h}(k_1L_1+L_1-1,k_2L_2)\right]^T, \qquad (6.17d)$$

and the block of vertical states are also defined similarly. In the existing block state structure as well as our new incremental block-state structure, the horizontal states $\mathbf{h}^{(1,L_2)}(k_1L_1+L_1,k_2L_2)$ and the vertical states $\mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2+L_2)$ are updated using the corresponding past horizontal and vertical states $\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)$ and $\mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2)$ (see Fig. 6.15). In this block state update process, the horizontal states $\mathbf{h}^{(L_1-1,L_2)}(k_1L_1+1,k_2L_2)$ and the vertical states $\mathbf{v}^{(L_1,L_2-1)}(k_1L_1,k_2L_2+1)$ are missed.

In the existing block-state structure, the block of outputs $\mathbf{y}^{(L_1,L_2)}(k_1L_1,k_2L_2)$ are computed using the available horizontal and vertical states $\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)$ and $\mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2)$ only. This output computation leads to an $O(L_1^2L_2^2)$ multiplication complexity. In our new incremental block state structure, instead of computing all L_1L_2 outputs all at once, we compute the outputs increment by increment in a sequential manner. Let the size of an increment be $I_1{\times}I_2$, and let I_1 and I_2 be divisible by L_1 and L_2 for simplicity (however this need not be the case). In the incremental block state filter, we use the states $\mathbf{h}^{(1,I_2)}(k_1L_1,k_2L_2)$ and $\mathbf{v}^{(I_1,1)}(k_1L_1,k_2L_2)$ to compute the incremental output $\mathbf{y}^{(I_1,I_2)}(k_1L_1,k_2L_2)$, and to non-recursively compute the intermediate horizontal and vertical states $\mathbf{h}^{(1,I_2)}(k_1L_1+I_1,k_2L_2)$ and $\mathbf{v}^{(I_1,1)}(k_1L_1,k_2L_2+I_2)$ (which were missed due to the block state update process, but are computed non-recursively) (see Fig. 6.16).

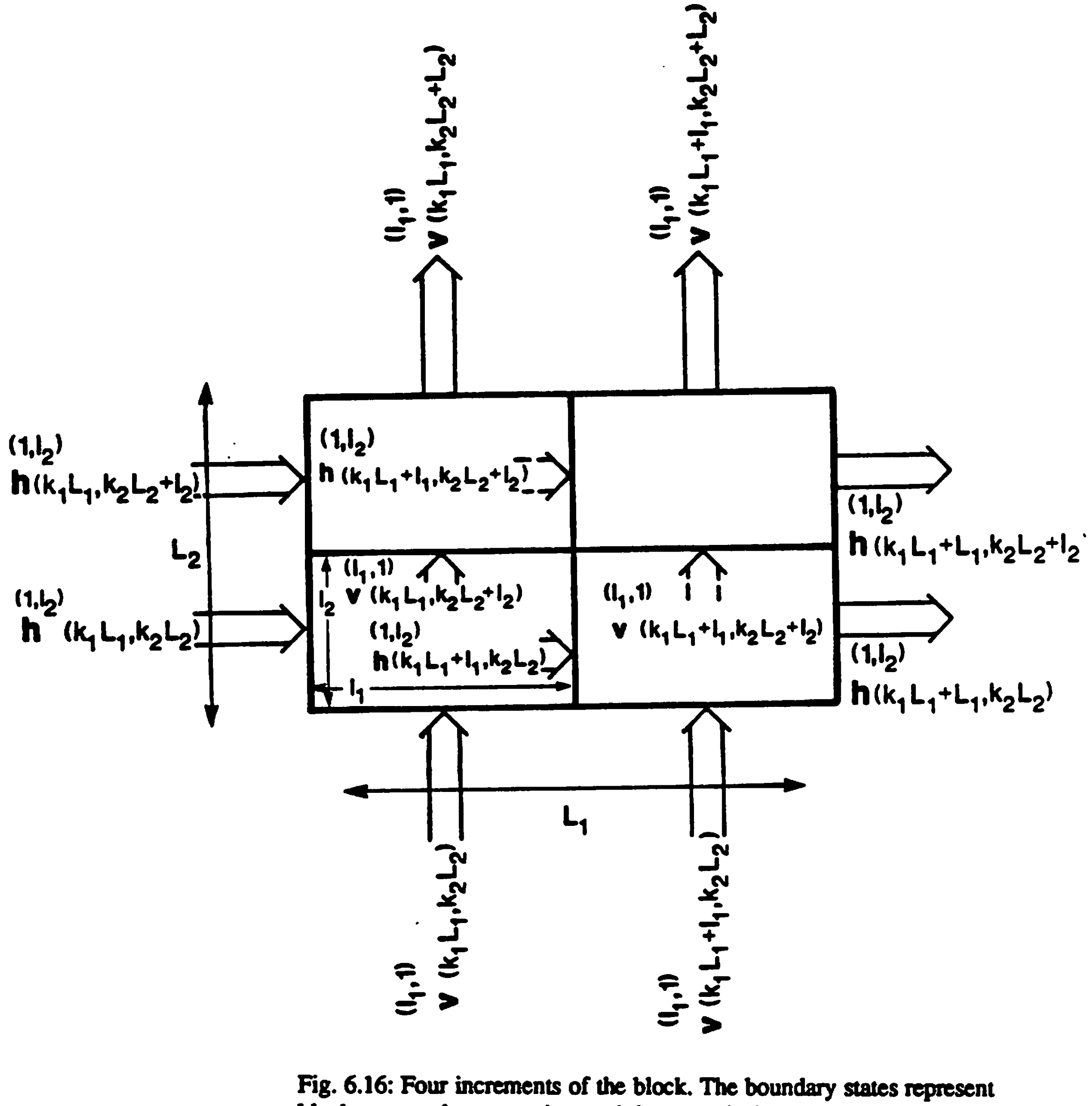


Fig. 6.16: Four increments of the block. The boundary states represent block state update operation, and the states inside the block correspond to non-recursive or sequential intermediate state computation (which were missed due to block state update) to be used for incremental output computation.

By non-recursive computation, it is meant that these states are not updated in a recursive manner using their past values, but are computed using the available states outside the state update feedback loop. These intermediate states are used for computing the next incremental outputs. A family of block structures can be derived with different values of increment sizes, and the existing block state structure is a special case of the incremental block state structure where the increment size equals the block size. The size of the increment is chosen to minimize the total multiplication complexity. The typical increment size parameters are 1×1 or 2×2.

As an example, for $L_1=4, L_2=6$ and $I_1=2$ and $I_2=3$, each block consists of 4 increments. First, we use $\mathbf{h}^{(1,3)}(4k_1,6k_2)$ and $\mathbf{v}^{(2,1)}(4k_1,6k_2)$ to compute $\mathbf{y}^{(2,3)}(4k_1,6k_2)$, and the missed intermediate states $\mathbf{h}^{(1,3)}(4k_1+2,6k_2)$, and $\mathbf{v}^{(2,1)}(4k_1,6k_2+3)$. We use these intermediate horizontal states and the available vertical states $\mathbf{v}^{(2,1)}(4k_1+2,6k_2)$ to compute $\mathbf{y}^{(2,3)}(4k_1+2,6k_2)$, and the missed intermediate vertical states $\mathbf{v}^{(2,1)}(4k_1+2,6k_2+3)$. Then we use available horizontal states $\mathbf{h}^{(1,3)}(4k_1,6k_2+3)$ and already computed vertical states $\mathbf{v}^{(2,1)}(4k_1,6k_2+3)$ to compute $\mathbf{y}^{(2,3)}(4k_1,6k_2+3)$, and the missed horizontal states $\mathbf{h}^{(1,3)}(4k_1+2,6k_2+3)$. Finally, we compute the output of the last increment $\mathbf{y}^{(2,3)}(4k_1+2,6k_2+3)$ using currently available states $\mathbf{h}^{(1,3)}(4k_1+2,6k_2+3)$ and $\mathbf{v}^{(2,1)}(4k_1+2,6k_2+3)$.

We can use the technique of look-ahead and iterate (6.12) successively to obtain

$$\mathbf{h}(k_1L_1+L_1,k_2L_2+s)=\sum_{t=0}^{s}\mathbf{A}_{11}^{L_1,s-t}\mathbf{h}(k_1L_1,k_2L_2+t)+\sum_{t=0}^{L_1-1}\mathbf{A}_{12}^{L_1-t,s}\mathbf{v}(k_1L_1+t,k_2L_2) \qquad (6.18a)$$

$$+\sum_{t=0}^{s}\sum_{r=1}^{L_1}\mathbf{b}_1^{L_1-r+1,s-t}u(k_1L_1+r-1,k_2L_2+t),\quad s=0,1,\ldots.,(L_2-1)$$

$$\mathbf{v}(k_1L_1+p, k_2L_2+L_2) = \sum_{t=0}^{L_2-1} \mathbf{A}_{21}^{p, L_2-t}\mathbf{h}(k_1L_1, k_2L_2+t) + \sum_{t=0}^{p} \mathbf{A}_{22}^{p-t, L_2}\mathbf{v}(k_1L_1+t, k_2L_2) \quad (6.18b)$$

$$+ \sum_{t=0}^{p}\sum_{r=1}^{L_2} \mathbf{b}_2^{p-t, L_2-r+1} u(k_1L_1+t, k_2L_2+r-1), p = 0, 1, \ldots, (L_1-1)$$

where

$$\mathbf{A}_{11}^{i,j} = \mathbf{A}_{11}\mathbf{A}_{11}^{i-1,j} + \mathbf{A}_{12}\mathbf{A}_{21}^{i-1,j}, \ \mathbf{A}_{11}^{0,0} = 1 \quad (6.18c)$$

$$\mathbf{A}_{12}^{i,j} = \mathbf{A}_{11}\mathbf{A}_{12}^{i-1,j} + \mathbf{A}_{12}\mathbf{A}_{22}^{i-1,j}, \ \mathbf{A}_{12}^{0,0} = 0 \quad (6.18d)$$

$$\mathbf{A}_{21}^{i,j} = \mathbf{A}_{21}\mathbf{A}_{11}^{i,j-1} + \mathbf{A}_{22}\mathbf{A}_{21}^{i,j-1}, \ \mathbf{A}_{21}^{0,0} = 0 \quad (6.18e)$$

$$\mathbf{A}_{22}^{i,j} = \mathbf{A}_{21}\mathbf{A}_{12}^{i,j-1} + \mathbf{A}_{22}\mathbf{A}_{22}^{i,j-1}, \ \mathbf{A}_{22}^{0,0} = 1 \quad (6.18f)$$

for $(i,j)>(0,0)$ and

$$\mathbf{A}_{11}^{i,j} = \mathbf{A}_{12}^{i,j} = \mathbf{A}_{21}^{i,j} = \mathbf{A}_{22}^{i,j} = 0 \ \text{for} \ (i,j)<(0,0) \quad (6.18g)$$

$$\mathbf{A}_{11}^{0,k} = \mathbf{A}_{12}^{0,k} = \mathbf{A}_{21}^{k,0} = \mathbf{A}_{22}^{k,0} = 0 \ \text{for} \ k > 0 \quad (6.18h)$$

The recurrence relations for $\mathbf{b}_1^{i,j}$ and $\mathbf{b}_2^{i,j}$ are given by

$$\mathbf{b}_1^{i,j} = \mathbf{A}_{11}^{i-1,j}\mathbf{b}_1 + \mathbf{A}_{12}^{i,j-1}\mathbf{b}_2, \ (i,j)>(0,0) \quad (6.18i)$$

$$\mathbf{b}_2^{i,j} = \mathbf{A}_{21}^{i-1,j}\mathbf{b}_1 + \mathbf{A}_{22}^{i,j-1}\mathbf{b}_2, \ (i,j)>(0,0) \quad (6.18j)$$

and

$$\mathbf{b}_1^{0,k} = \mathbf{b}_2^{k,0} = 0, \ k>0$$

These sequences can also be alternatively defined as

$$\mathbf{b}_1^{i,j} = \mathbf{A}_{11}\mathbf{b}_1^{i-1,j} + \mathbf{A}_{12}\mathbf{b}_2^{i-1,j}, \ (i,j)>(1,1), \ \mathbf{b}_1^{1,0} = b_1 \quad (6.18k)$$

$$\mathbf{b}_2^{i,j} = \mathbf{A}_{21}\mathbf{b}_1^{i,j-1} + \mathbf{A}_{22}\mathbf{b}_2^{i,j-1}, \ (i,j)>(1,1), \ \mathbf{b}_2^{0,1} = b_2 \quad (6.18l)$$

These recursive relations can be derived by successive iterations or by the method of induction. These relations are different from those derived in [27] and are easily extensible to higher dimensions. In this state update reformulation, we do not need any dummy zero states as in [27].

Using (6.18), we can derive a matrix version of the two-dimensional block update operation of the horizantal and vertical states, and this formulation is given in (A6.7). The multiplication complexity of this formulation is given by

$$C_s(L_1,L_2) = \frac{L_2(L_2+1)}{2}N_1^2 + \frac{L_1(L_1+1)}{2}N_2^2 + 2N_1N_2L_1L_2 \tag{6.19}$$
$$+ \frac{L_1L_2(L_1+1)}{2}N_2 + \frac{L_1L_2(L_2+1)}{2}N_1$$

Now we derive the output computation representation for the block state filter, and then for our incremental block state filter. The output $y(k_1L_1+p, k_2L_2+q)$ can be computed in terms of the available boundary states using

$$y(k_1L_1+p, k_2L_2+q) = \sum_{t=0}^{q} \mathbf{c}_1^{p,q-t}\mathbf{h}(k_1L_1, k_2L_2+t) + \sum_{t=0}^{p} \mathbf{c}_2^{p-t,q}\mathbf{v}(k_1L_1+t, k_2L_2) \tag{6.20a}$$
$$+ \sum_{t=0}^{q}\sum_{r=1}^{p+1} d^{p-r+1,q-t}u(k_1L_1+r-1, k_2L_2+t)$$

where

$$\mathbf{c}_1^{i,j} = \mathbf{c}_1\mathbf{A}_{11}^{i,j} + \mathbf{c}_2\mathbf{A}_{21}^{i,j} \tag{6.20b}$$

$$\mathbf{c}_2^{i,j} = \mathbf{c}_1\mathbf{A}_{12}^{i,j} + \mathbf{c}_2\mathbf{A}_{22}^{i,j} \tag{6.20c}$$

$$\mathbf{d}^{i,j} = \mathbf{c}_1\mathbf{b}_1^{i,j} + \mathbf{c}_2\mathbf{b}_2^{i,j} + d\,\delta(i,j),\ d^{0,0} = d\ . \tag{6.20d}$$

Using (6.20), a matrix version of the block output computation is formulated in (A6.8) (see appendix 7.3). The multiplication complexity for computing $L_1{\times}L_2$ block of outputs is

$$C_o(L_1,L_2) = \frac{N_1L_1L_2(L_2+1)}{2} + \frac{N_2L_1L_2(L_1+1)}{2} + \frac{L_1L_2(L_1+1)(L_2+1)}{4} \tag{6.21}$$
$$= \frac{L_1L_2}{4}\left[(L_1+1)(L_2+1) + 2N_1(L_2+1) + 2N_2(L_1+1)\right]$$

and is $O(L_1^2L_2^2)$. The total multiplication complexity of the block state filter is the sum of the state update complexity and the output computation complexity, and is given by

$$C_b = C_s(L_1,L_2) + C_o(L_1,L_2)\,. \qquad (6.22)$$

In our incremental block state filter, the outputs are computed using the non-recursively computed intermediate states (which were missed due to the block state update process) and the corresponding inputs. For a block size of $L_1 \times L_2$ and an increment size $I_1 \times I_2$, the total number of state computation operations (each state computation operation includes computing I_1 vertical states and I_2 horizontal states) amounts to $(\frac{L_1}{I_1} - 1)(\frac{L_2}{I_2} - 1)$ (since the boundary states are already computed by the block state update operation). The multiplication complexity of the incremental block state filter is the sum of the recursive state update complexity, the non-recursive intermediate state computation complexity, and the incremental output computation complexity, and is given by

$$C_i = C_s(L_1,L_2) + (\frac{L_1}{I_1} - 1)(\frac{L_2}{I_2} - 1)C_s(I_1,I_2) + \frac{L_1L_2}{I_1I_2}C_o(I_1,I_2)\,. \qquad (6.23)$$

This complexity can be verified to be independent of $O(L_1^2L_2^2)$ term. The size of the increment I_1 and I_2 are chosen to minimize the output computation complexity.

Table 6.6 shows the multiplication complexities per sample for a 2D recursive digital filter with $N_1 = N_2 = 4$ for the block state structure and the incremental block state structure with typical increment values. We observe that for large values of block size, we can save the multiplication complexity by about an order of magnitude by using the incremental block state structure as compared with the existing block state structure.

Table 6.6: Per Output Complexities of Block State and Incremental Block State Filters

(L_1,L_2)	Block State	Incremental Block State	
	Complexity	Complexity	(I_1,I_2)
(16,16)	257	188	(1,1)
(32,32)	586	254	(2,2)
(64,64)	1635	390	(2,2)
(128,128)	5250	671	(1,1)
(256,256)	18311	1159	(1,1)
(512,512)	68665	2174	(1,1)

The roundoff noise for the block state and the incremental block state structures are the same under the assumption of linear additive white gaussian noise, since the recursive state update operation is the same for these two schemes. Ju and Alexander have studied the quantization noise of multidimensional block filters in [46-47], and these results are also applicable to the structures proposed in this chapter. They also studied the stability aspects of multidimensional IIR filters [48] using methods which are also applicable to polyphase networks [49]. The standard direct form block filters for half plane recursive filters have been derived in [50]. Our techniques presented here can also be extended to derive efficient architectures for the half plane recursive digital filters.

6.4. PIPELINING AND TWO-DIMENSIONAL BLOCK PROCESSING

In this section, we achieve a speedup by a factor of L_1L_2M with a block size $L_1 \times L_2$ and M stages of pipelining inside the recursive loop.

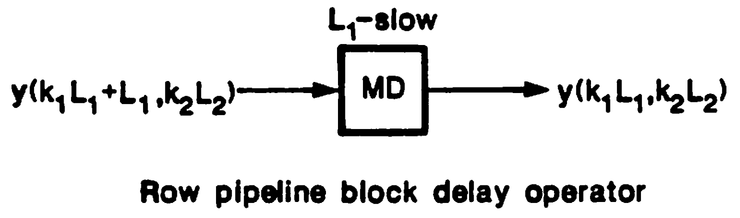
Consider pipelining the 2D block recursive digital filter by M stages. For this case, if we begin the computation of the block $y^{(L_1L_2)}(k_1L_1,k_2L_2)$ at time index n, then the result of this computation will be available at time index $(n+M)$ due to the latency introduced due to pipelining. We cannot begin the computation of the next block $y^{(L_1L_2)}(k_1L_1+L_1,k_2L_2)$ until time index $(n+M)$. This might imply that $(M-1)$ cycles out of M cycles have been wasted, which is not true! Just as in the non-blocked pipelining case (see section 6.2), we can keep the pipeline busy by proper indexing or interleaving of blocks of input samples (as opposed to interleaving one by one input sample as in non-blocked pipelining case). Thus the implementable delay operator is obtained by appropriate interleaving of blocks of samples, and a pipelined realization is obtained with no hardware overhead. Let a *segment* be defined to consist of L_2M consecutive rows of input samples. Then with M stages of pipelining and for processing the first segment, the M blocks $y^{(L_1L_2)}(k_1L_1,k_2L_2)$, $y^{(L_1L_2)}((k_1-1)L_1,(k_2+1)L_2)$, ..., and $y^{(L_1L_2)}((k_1-M+1)L_1,(k_2+M-1)L_2)$ can be skew interleaved and processed in a pipelined manner. Similarly in the case of a local state space filter, the block of horizontal and vertical states corresponding to M skewed blocks can be interleaved in the pipeline. The index mapping function for pipelined block implementation is given by

$$I(n_1,n_2)=\lfloor \frac{n_1}{L_1} \rfloor M + \lfloor \frac{n_2}{L_2} \rfloor (M+1) + \lfloor \frac{n_2}{L_2M} \rfloor M(\frac{J_1}{L_1}-M-1)\,. \tag{6.24}$$

This index mapping function holds good for direct form as well as local state space form filters, and reduces to those in (6.10) and (6.14) as special cases. The *delay operator* can be derived by inspecting the index mapping function. The loop delay operator with respect to the index n_1 corresponds to M L_1*–slow* delays, and that with respect to the index n_2 corresponds to $(M+1)$ L_2*–slow* delays followed by a multiplexed path, one with no delay (for the case where the signal and its delayed version belong to the same segment), and the other with $(\frac{J_1}{L_1} - M - 1) L_2 M$*-slow* delays (for the case where the signal and its delayed version belong to two consecutive segments). These delay operators are shown in Fig. 6.17. Using these loop delay operators, a block diagram of the state update portion of a local state space form 2D recursive digital filter with block size $L_1 \times L_2$ and M stages of loop pipelining latches is shown in Fig. 6.18.

Similar to the non-blocked pipelining case, the total number of wasted cycles is $M(M-1)$, and hence, the hardware utilization efficiency of this realization is given by

$$\eta = \frac{\frac{J_1 J_2}{L_1 L_2}}{\frac{J_1 J_2}{L_1 L_2} + M(M-1)} . \tag{6.25}$$



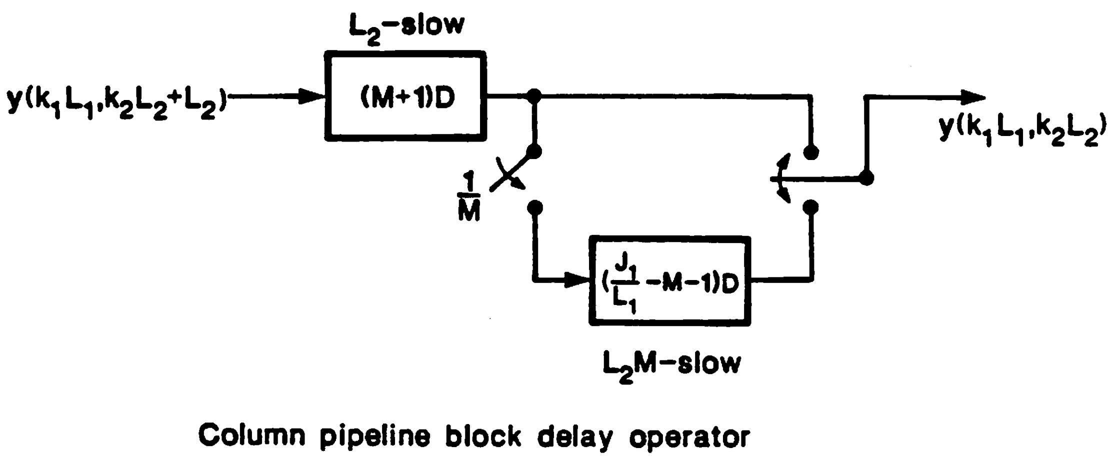
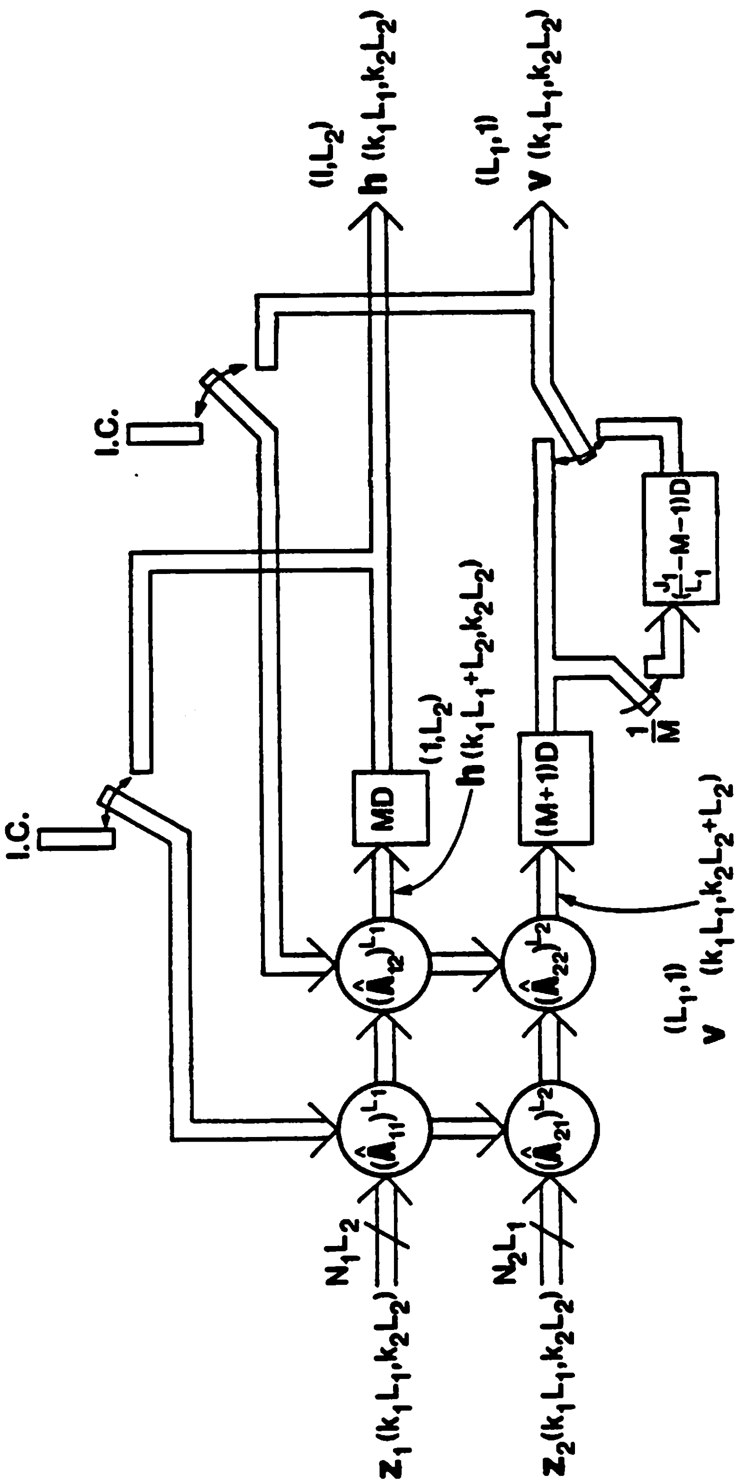


Fig. 6.17: Row and column delay operators in a M stage pipelined block realization with block size $L_1 \times L_2$.



$\mathbf{z}_1(k_1L_1,k_2L_2)$
N_1L_2
$\mathbf{z}_2(k_1L_1,k_2L_2)$
N_2L_1
$(\hat{A}_{11})^{L_1}$
$(\hat{A}_{12})^{L_1}$
$(\hat{A}_{21})^{L_2}$
$(\hat{A}_{22})^{L_2}$
I.C.
I.C.
MD
(M+1)D
$\mathbf{h}^{(1,L_2)}(k_1L_1+L_2,k_2L_2)$
$\mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2+L_2)$
$\frac{1}{M}$
$(\frac{J_1}{L_1}-M-1)D$
$\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)$
$\mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2)$

Fig. 6.18: State update portion of a *M*-stage pipelined block realization of a local state space 2D recursive filter.

6.5. MULTIDIMENSIONAL RECURSIVE FILTERS

In this section, we outline the extension of the pipeline interleaving and the incremental block filtering concepts to N-dimensional recursive digital filters.

With the increase in the number of dimensions, the number of independent computations (that is computations which are not mutually constrained by precedence relations) also increases, and the size of the concurrent computation region (CCR) grows. For an N dimensional filter, the CCR corresponds to the $(N-1)$-dimensional hyperplane $\sum_{i=1}^{N} n_i = c$ (where c is as constant). For example, the CCR in a three-dimensional system corresponds to a triangle (which is a two-dimensional surface or a plane). Therefore, pipeline interleaving or sub-dimensional block processing approach can be adopted with no hardware penalty using the samples belonging to the CCR in a manner similar to the two dimensional case. The implementable delay operator can be derived by choosing an appropriate index mapping function (which is non-unique).

The incremental block digital filtering approach can also be used in the context of higher dimensional digital filters. In the case of direct form N dimensional digital filters with filter orders $N_1 \times N_2 \times .... \times N_N$, and block size $L_1 \times L_2 \times .... \times L_N$, $(\prod_{k=1}^{N} L_i - \prod_{k=1}^{N} (L_i - N_i))$ states or outputs can be recursively updated in each block, and the remaining $\prod_{k=1}^{N} (L_i - N_i)$ outputs can be incrementally computed in a sequential manner. In the case of local state space N dimensional recursive digital filter case, the states can be updated block by block; but the outputs can be incrementally computed using non-recursively computed intermediate states (which were missed due to the block by block state update process) in a sequential manner. The multiplication complexity of the N dimensional incremental block filter can be verified to be $O(\underset{i \in [1,N]}{Max} (L_i \prod_{k=1, k \neq i}^{N} L_k^2))$ as opposed to $O(\prod_{k=1}^{N} L_k^2)$ in higher dimensional block filters [47]. Note that the dominant complexity in the incremental block recursive digital filter is due to the block state update operation, unlike that due to the output computation as in the existing block filters. Furthermore, pipeline interleaving and incremental block filtering approaches can also be combined to obtain efficient multidimensional filters as in the two dimensional case.

6.6. CONCLUSION

We have presented several approaches to achieving concurrency in two dimensional recursive digital filters. These approaches lead to an understanding of concurrent processing of tasks in image processing systems. We have shown that it is much more hardware efficient to exploit the inherent concurrency available in the processing of two-dimensional data, either in the context of pipeline interleaving and/or one-dimensional block processing, rather than using two-dimensional block processing. Although several concurrent techniques have been presented in this chapter, most of them are too complex to implement on VLSI chips due to large hardware complexity and severe I/O constraints, except the case of pipeline interleaved processing (i.e. with no block processing) and one-dimensional block processing.

So far we have studied transformations on specific linear digital filter algorithms. These techniques can also be locally applied to linear recursive nodes in any general data flow signal processing program. These issues are studied in the next chapter.

6.7. APPENDICES

6.7.1. Appendix 7.1

In this appendix, we define a sequence r_{n_1,n_2} which is used in the context of the direct form block digital filter. This sequence is defined by

$$r_{n_1,n_2} = \sum_{\substack{i_1=0 \\ (i_1,i_2)\neq(0,0)}}^{N_a} \sum_{i_2=0}^{N_b} a_{i_1,i_2} r_{n_1-i_1,n_2-i_2} \tag{A6.1}$$

and

$$r_{0,0} = 1,\ r_{-i_1,i_2} = 0 \text{ for } i_1>0,\ r_{i_1,-i_2} = 0 \text{ for } i_2>0\,. \tag{A6.2}$$

In (A6.1), r_{n_1,n_2} is expressed recursively as a function of neighboring $((N_a+1)(N_b+1)-1)$ values of r_{i_1,i_2}. Suppose we are interested in expressing r_{n_1,n_2} as a function of values of r_{i_1,i_2} M_1 samples away with respect to the index n_1, and M_2 samples away with respect to the index n_2 (see Fig. 6.13(b)), then we can iterate (A6.1) appropriately. to obtain:

$$\begin{aligned} r_{n_1,n_2} = & \sum_{\substack{i_1=0 \\ (i_1,i_2)\neq(0,0)}}^{N_a} \sum_{i_2=0}^{N_b} \left[\sum_{j_1=i_1}^{N_a} \sum_{j_2=i_2}^{N_b} a_{j_1,j_2} r_{i_1-j_1+M_1,i_2-j_2+M_2} \right] r_{n_1-i_1-M_1,n_2-i_2-M_2} \\ & + \sum_{i_1=0}^{M_1-1} \sum_{i_2=1}^{N_b} \left[\sum_{j_1=0}^{i_1} \sum_{j_2=i_2}^{N_b} a_{j_1,j_2} r_{i_1-j_1,i_2-j_2+M_2} \right] r_{n_1-i_1,n_2-i_2-M_2} \\ & + \sum_{i_1=1}^{N_a} \sum_{i_2=0}^{M_2-1} \left[\sum_{j_1=i_1}^{N_a} \sum_{j_2=0}^{i_2} a_{j_1,j_2} r_{i_1-j_1+M_1,i_2-j_2} \right] r_{n_1-i_1-M_1,n_2-i_2}\,. \end{aligned} \tag{A6.3}$$

This iterated relation can be derived either by successively iterating (A6.1) or by using induction. Here the value of r_{n_1,n_2} is expressed as a function of $(N_aM_2+N_bM_1+(N_a+1)(N_b+1)-1)$ values of r_{i_1,i_2}.

6.7.2. Appendix 7.2

In this appendix, we compute the multiplication complexity of a direct form two-dimensional block recursive digital filter with block size $L_1 \times L_2$ for the cases where the block sizes are greater than filter order as well as less than filter order. Note that the multiplication complexity for updating a single state using previous states M_1 samples apart in n_1 direction and M_2 in n_2 direction can be obtained from (6.15) and is given by

$$f(M_1, M_2) = (N_a+1)(N_b+1)+N_b M_1+N_a M_2+(M_1+1)(M_2+1)-2 \qquad \text{(A6.4)}$$

where multiplication by unity has been excluded.

Case I: $L_1 > N_a$ and $L_2 > N_b$

The total multiplication complexity for this case is the sum of the multiplication complexity due to the state update operation of $(N_a N_b+(L_1-N_a)N_b+N_a(L_2-N_b))$ states with appropriate (M_1, M_2) values, that needed for computation of $z^{(L_1, L_2)}(k_1 L_1+L_1, k_2 L_2+L_2)$ and that for incremental output computation of $(L_1-N_a)(L_2-N_b)$ outputs. The total multiplication complexity is given by

$$C = \sum_{M_1=L_1-N_a}^{L_1-1} \sum_{M_2=L_2}^{2L_2-N_b-1} f(M_1, M_2) + \sum_{M_1=L_1-N_a}^{L_1-1} \sum_{M_2=L_2-N_b}^{L_2-1} f(M_1, M_2) \qquad \text{(A6.5)}$$

$$+ \sum_{M_1=L_1}^{2L_1-N_a-1} \sum_{M_2=L_2-N_b}^{L_2-1} f(M_1, M_2) + L_1 L_2 (N_a+1)(N_b+1) + (L_1-N_a)(L_2-N_b)((N_a+1)(N_b+1)-1) .$$

The above can be simplified to verify that C is independent of $O(L_1^2 L_2^2)$ term.

Case II: $L_1 < N_a$ and/or $L_2 < N_b$

For this case, the multiplication complexity is the sum of the state update complexity and the complexity to compute $z^{(L_1, L_2)}(k_1 L_1+L_1, k_2 L_2+L_2)$. The multiplication complexity is given by

$$C = \sum_{M_1=0}^{L_1-1}\sum_{M_2=0}^{L_2-1} f(M_1,M_2) + L_1L_2(N_a+1)(N_b+1) \tag{A6.6}$$

and is an $O(L_1^2L_2^2)$ complexity.

6.7.3. Appendix 7.3

We use use (6.18) to derive a matrix version of the two-dimensional block state update operation to be

$$\begin{bmatrix} \mathbf{h}^{(1,L_2)}(k_1L_1+L_1,k_2L_2) \\ \mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2+L_2) \end{bmatrix} = \begin{bmatrix} (\hat{\mathbf{A}}_{11})^{L_1} & (\hat{\mathbf{A}}_{12})^{L_1} \\ (\hat{\mathbf{A}}_{21})^{L_2} & (\hat{\mathbf{A}}_{22})^{L_2} \end{bmatrix} \begin{bmatrix} \mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2) \\ \mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2) \end{bmatrix} \tag{A6.7a}$$

$$+ \begin{bmatrix} (\hat{\mathbf{B}}_1)^{L_1}\rho \\ (\hat{\mathbf{B}}_2)^{L_2} \end{bmatrix} \mathbf{u}^{(L_1,L_2)}(k_1L_1,k_2L_2) \, ,$$

where $\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)$ is $N_1L_2\times1$, $\mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2)$ is $N_2L_1\times1$, $\mathbf{u}^{(L_1,L_2)}(k_1L_1,k_2L_2)$ is $L_1L_2\times1$, and other submatrices are of appropriate dimensions. Note that the submatrices $(\hat{\mathbf{A}}_{11})^{L_1}$, $(\hat{\mathbf{A}}_{22})^{L_2}$, $(\hat{\mathbf{B}}_1)^{L_1}$ and $(\hat{\mathbf{B}}_2)^{L_2}$ are triangular. Using (6.18), the elements of these matrices can be derived to be

$$(\hat{\mathbf{A}}_{11})^{K}_{i,j} = \mathbf{A}_{11}^{K,i-j},\ (\hat{\mathbf{A}}_{12})^{K}_{i,j} = \mathbf{A}_{12}^{K-j+1,i-1} \tag{A6.7b}$$

$$(\hat{\mathbf{A}}_{21})^{K}_{i,j} = \mathbf{A}_{21}^{i-1,K-j+1},\ (\hat{\mathbf{A}}_{22})^{K}_{i,j} = \mathbf{A}_{22}^{i-j,K} \tag{A6.7c}$$

$$\left[(\hat{\mathbf{B}}_1)^{K}_{i,j}\right]_l = \mathbf{b}_1^{K-l+1,i-j},\ \left[(\hat{\mathbf{B}}_2)^{K}_{i,j}\right]_l = \mathbf{b}_2^{i-j,K-l+1} \, . \tag{A6.7d}$$

Note that $(\hat{\mathbf{B}}_1)^{K}_{i,j}$ represents a $N_1\times L_1$ vector and $\left[(\hat{\mathbf{B}}_1)^{K}_{i,j}\right]_l$ represents the l-th element of the row vector of dimension $N_1\times1$. Similarly $(\hat{\mathbf{B}}_2)^{K}_{i,j}$ represents a $N_2\times L_2$ vector and $\left[(\hat{\mathbf{B}}_2)^{K}_{i,j}\right]_l$ represents the l-th element of the row vector of dimension $N_2\times1$. The multiplication complexity for the above block state update operation is given by (6.19).

We use (6.20) to derive a matrix version of the output computation

$$\mathbf{y}^{(L_1,L_2)}(k_1L_1,k_2L_2)=\begin{bmatrix}(\hat{\mathbf{C}}_1)^{L_1,L_2} & (\hat{\mathbf{C}}_2)^{L_1,L_2}\end{bmatrix}\begin{bmatrix}\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)\\ \mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2)\end{bmatrix} \tag{A6.8a}$$

$$+(\hat{\mathbf{D}})^{L_1,L_2}\mathbf{u}^{(L_1,L_2)}(k_1L_1,k_2L_2)\,,$$

or equivalently,

$$\begin{bmatrix}\mathbf{y}^{(1,L_2)}(k_1L_1,k_2L_2)\\ \mathbf{y}^{(1,L_2)}(k_1L_1+1,k_2L_2)\\ \vdots\\ \mathbf{y}^{(1,L_2)}(k_1L_1+L_1-1,k_2L_2)\end{bmatrix}=\begin{bmatrix}(\hat{\mathbf{C}}_1)_1 & (\hat{\mathbf{C}}_2)_1\\ (\hat{\mathbf{C}}_1)_2 & (\hat{\mathbf{C}}_2)_2\\ \vdots & \vdots\\ (\hat{\mathbf{C}}_1)_{L_1} & (\hat{\mathbf{C}}_2)_{L_1}\end{bmatrix}\begin{bmatrix}\mathbf{h}^{(1,L_2)}(k_1L_1,k_2L_2)\\ \mathbf{v}^{(L_1,1)}(k_1L_1,k_2L_2)\end{bmatrix} \tag{A6.8b}$$

$$+\begin{bmatrix}(\hat{\mathbf{D}})_1 & \mathbf{0} & \cdots & \mathbf{0}\\ (\hat{\mathbf{D}})_2 & (\hat{\mathbf{D}})_1 & \cdots & \mathbf{0}\\ \vdots & \vdots & \cdots & \vdots\\ (\hat{\mathbf{D}})_{L_1} & (\hat{\mathbf{D}})_{L_1-1} & \cdots & (\hat{\mathbf{D}})_1\end{bmatrix}\begin{bmatrix}\mathbf{u}^{(1,L_2)}(k_1L_1,k_2L_2)\\ \mathbf{u}^{(1,L_2)}(k_1L_1+1,k_2L_2)\\ \vdots\\ \mathbf{u}^{(1,L_2)}(k_1L_1+L_1-1,k_2L_2)\end{bmatrix}$$

where $(\hat{\mathbf{C}}_1)_p$ is $L_2{\times}N_1L_2$, $(\hat{\mathbf{C}}_2)_p$ is $L_2{\times}N_2L_1$, and $(\hat{\mathbf{D}})_p$ is $L_2{\times}L_2$ and are triangular. The ij-th elements of these submatrices are given by

$$\begin{bmatrix}(\hat{\mathbf{C}}_1)_p\end{bmatrix}_{ij}=\mathbf{c}_1^{p-1,i-j},\ p=1,2,\ldots,L_1 \tag{A6.8c}$$

$$\begin{bmatrix}(\hat{\mathbf{C}}_2)_p\end{bmatrix}_{ij}=\mathbf{c}_2^{p-j,i-1},\ p=1,2,\ldots,L_1 \tag{A6.8d}$$

$$\begin{bmatrix}(\hat{\mathbf{D}})_p\end{bmatrix}_{ij}=d^{p-1,i-j}\,. \tag{A6.8e}$$

The multiplication complexity for the the block output computation is given by (6.21).

6.8. REFERENCES

(1) Cappello, P.R., "Towards an FIR Filter Tissue", *Proceedings of the 1985 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tampa, 1985

(2) Fettweis, A., "Realizability of Digital Filter Networks", *Arch. Elek. Ubertrangung*, Vol. 30, February 1976, pp. 90-96

(3) Renfors, M., and Neuvo, Y., "The Maximum Sampling Rate of Digital Filters under Hardware Speed Constraints", *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, No. 3, March 1981, pp. 196-202

(4) Parhi, K.K., and Messerschmitt, D.G., "Look-Ahead Computation: Improving Iteration Bound in Linear Recursions", *Proceedings of the 1987 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987, pp. 1855-58

(5) Parhi, K.K., and Messerschmitt, D.G., "Block Digital Filtering via Incremental Block-State Structure", *Proceedings of the IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987, pp. 645-648

(6) Parhi, K.K., and Messerschmitt, D.G., "Concurrent Cellular VLSI Adaptive Filter Architectures", *IEEE Transactions on Circuits and Systems*, October 1987, pp. 1141-1151

(7) Parhi, K.K., and Messerschmitt, D.G., "Pipelined Recursive Filters using Scattered Look-Ahead and Decomposition", *Proceedings of the 1988 IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1988, NY, pp. 2120-2123

(8) Gold, B. and Jordan, K.L.,"A Note on Digital Filter Synthesis", *Proceedings of IEEE*, Vol. 65, pp.1717-1718, Oct. 1968

(9) Voelcker, H.B., and Hartquist, E.E.,"Digital Filtering Via Block Recursion", *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-18, pp. 169-176, June 1970.

(10) Burrus, C.S., "Block Implementation of Digital Filters ", *IEEE Transactions on Circuit Theory*, Vol. CT-18, pp. 697-701, Nov. 1971.

(11) Mitra, S.K. and Gnanasekaran, R., "Block Implementation of Recursive Digital Filters - New Structures and Properties ", *IEEE Transactions on Circuits and Systems*, Vol. CAS-25, pp.200-207, April 1978.

(12) Barnes, C.W. and Shinnaka, S.,"Block Shift Invariance and Block Implementation of of Discrete-Time Filters", *IEEE Trans on* Circuits and Systems, Vol. CAS-27, pp.667-672, Aug. 1980.

(13) Zeman, J., and Lindgren, A.G., "Fast Digital Filters with Low round-off Noise, " *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, pp.716-723, July 1981.

(14) Schwartz, D. A. and Barnwell, T.P. III, " Increasing the Parallelism of Filters Through Transformation to Block State Variable Form", *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 1984.

(15) Lu, H.H., Lee, E.A. and Messerschmitt, D.G., "Fast Recursive Filtering with Multiple Slow Processing Elements", *IEEE Trans. on Circuits and Systems*, Vol. CAS-32, No.11, November 1985, pp. 1119-1129. No.4, August 1984.

(16) Nikias, C.L.,"Fast Block Data Processing Via a New IIR Digital Filter Structure", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 32, No. 4, August 1984

(17) Sung, W., and Mitra, S.K., "Efficient Multiprocessor Implementation of Recursive Digital Filters", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tokyo, April 1986, pp. 257-260

(18) Arun, K.S., "Ultra-High-Speed Parallel Implementation of Low-Order Digital Filters", *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1986, pp. 944-946

(19) Wu, C.W., and Cappello, P.R., "Computer-Aided Design of VLSI Second Order Sections", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987

(20) Wu, C.W., and Cappello, P.R., "Application Specific CAD of VLSI Second Order Sections", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, May 1988, pp. 813-825

(21) Loney, J., and Strintzis, M.G., "Block Implementation of Two Dimensional Digital Filters", *Proceedings of IEEE Conference on Pattern recognition and Image Processing*, May 31 - June 2, 1978, Illinois, pp. 119-122

(22) Azimi-Sadjadi, M.R., and King, R.A., "Block Implementation of 2-D Digital Filters", *Proceedings of 22nd Midwest Symposium on Circuits and Systems*, June 1979, pp. 658-662

(23) Azimi-Sadjadi, M.R., "Block Implementation of Two Dimensional Digital Filters", *Proceedings of IEEE MEDCOMP*, Philadelphia, September 1982, pp. 160-169

(24) Mitra, S.K., and Gnanasekaran, R., "Block Implementation of Two Dimensional Digital Filters", *Journal of Franklin Institute*, Vol. 316, October 1983, pp. 299-316

(25) Mertzios, B.G., "Block Realization of 2-D IIR Digital Filters", *Signal Processing*, Vol. 7, No. 2, October 1984, pp. 135-149

(26) Azimi-Sadjadi, M.R., "A 2-D Block-State Realization Model", *Proceedings of IEEE International Symposium on Circuits and Systems*, Newport Beach, CA, 1983, pp. 919-922

(27) Azimi-Sadjadi, M.R., and King, R.A., "Two-Dimensional Block Processors - Structures and Implementations", *IEEE Transactions on Circuits and Systems*, Vol. CAS-33, No. 1, January 1986, pp. 42-50

(28) Azimi-Sadjadi, M.R., Rostampour, A.R., and Lu, T., "Parallel Architectures for 2-D Block Processing", *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, Victoria BC, Canada, June 1987, pp. 455-460

(29) Alexander, W.E., and Ju, C.J., "1-D to N-D Highly Parallel Computing Structures for IIR Digital Filters", *Proceedings of IEEE International Symposium on Circuits and Systems*, Philadelphia, May 1987, pp. 538-541

(30) Chiang, H.H., and Nikias, C.L., "Parallel Block Realization of 2-D IIR Digital Filters", *Proceedings of the 1985 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tampa, Florida, 1985, pp. 1301-1304

(31) Parhi, K.K., and Messerschmitt, D.G., "Two-Dimensional Recursive Digital Filtering: Pipelining, One-, and Two-Dimensional Block Processing", *Proceedings of IEEE International Symposium on Circuits and Systems*, Helsinki, Finland, June 1988

(32) Aboulnasr, T., and Steenart, W., "Real-Time Systolic Array Processor for 2-D Spatial Filtering", *Proceedings of the 1986 EUSIPCO*, Holland, Sept. 1986, pp. 687-690

(33) Aboulnasr, T., and Steenart, W., "Real Time Systolic Array Processor for 2D Spatial Filtering", *IEEE Trans. on Circuits and Systems*, Vol. CAS-35, No. 4, April 1988, pp. 451-455

(34) Lamport, L., "The Parallel Execution of Do Loops", *Communications of the ACM*, Vol. 17, No. 2, pp. 83-93, February 1974

(35) Martens, J.B., "Discrete Fourier Transform Algorithms for Real Valued Sequences", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-32, pp. 390-396, April 1984

(36) Agarwal, R.C., and Burrus, C.S., "Number Theoretic Transforms to Implement Fast Digital Convolution", *IEEE Proceedings*, Vol. 63, pp. 550-560, April 1975

(37) Martens, J.B., "Two-Dimensional Convolutions by means of Number Theoretic Transforms over Residue Class Polynomial Rings", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-32, No. 4, August 1984

(38) Nussbaumer, H.J., and Quandalle, P., "Computation of Convolutions and Discrete Fourier Transforms by Polynomial Transforms", *IBM J. of Research and Development*, Vol. 22, pp. 134-144, March 1978

(39) Nussbaumer, H.J., "Fast Polynomial Algorithms for Digital Convolution", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-28, pp. 205-215, April 1980

(40) Agarwal, R.C., and Burrus, C.S., "Fast One Dimensional Digital Convolution by Multidimensional Techniques", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, pp. 1-10, Feb. 1974

(41) Dudgeon, D.E., and Mersereau, R.M., "Multidimensional Digital Signal Processing", Prentice Hall, 1984

(42) Leiserson, C.E., Rose, F., and Saxe, S., "Optimizing Synchronous Circuitry by Retiming, *Proceedings of the Third Caltech Conference on VLSI*, Pasadena, March 1983

(43) Kung, S.Y., "On Supercomputing with Systolic/Wavefront Array Processors", *Proceedings of the IEEE*, Vol. 72, No. 7, July 1984

(44) Kung, S.Y. *et al*, "New Results in 2D Systems Theory, Part II: 2D State Space Models - Realization, and the Notions of Controllability, Observability, and Minimality", *Proceedings of the IEEE*, Vol. 65, June 1977, pp. 945-961

(45) Roesser, R.P., "A Discrete State Space Model for Linear Image Processing", *IEEE Transactions on Automatic Control*, Vol. AC-20, No. 1, January 1975, pp. 1-10

(46) Ju, C.J., and Alexander, W.E., "Performance Analysis of Multidimensional Block State-Space Models", *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 1987, Philadelphia, pp. 698-701

(47) Ju, C.J., and Alexander, W.E., "Block Realization of Multidimensional IIR Digital Filters and its Finite Word Effects", *IEEE Trans. on Circuits and Systems*, Vol. CAS-34, No. 9, September 1987, pp. 1030-1044

(48) Ju, C.J., and Alexander, W.E., "Derivation and Stability Ananlysis of Multidimensional IIR Block Digital Filters", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Dallas, April 1987, pp. 1673-1676

(49) Bellanger, M.G., Bonnerot, G., and Coudreuse, M., "Digital Filtering by Polyphase Network: Application to Sample Rate Alteration and Filter Banks", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-24, April 1976, pp. 109-114

(50) Lee, J.H., "Two-Dimensional Block Implementation of Symmetric Half Plane Recursive Digital Filters", *IEEE Trans. on Circuits and Systems*, Vol. CAS-35, No. 6, June 1988, pp. 736-742

7

CONCURRENT DATA-FLOW SIGNAL PROCESSING

7.1. INTRODUCTION

In chapter 2, we studied the program unfolding approach for creating concurrency in data-flow programs. In chapters 3 through 6, we proposed look-ahead transformations to derive high sample rate realizations of one- and two-dimensional recursive and adaptive digital filters. In this chapter, we combine look-ahead and program unfolding approaches to perform local transformations in general data-flow signal processing programs to create additional concurrency.

In section 7.2, we review the look-ahead transformation in single node programs. All the digital filter algorithms studied in earlier chapters can be described by these programs. In section 7.3, we illustrate application of look-ahead transformation in general single-rate iterative data-flow programs, and in section 7.4, we illustrate transformations in multirate iterative data-flow programs. These transformations improve the iteration period bound of these programs, and increase the level of concurrency in their implementations. These transformations are useful for high-performance implementation of real-time applications, especially where the iteration period bound of the algorithms is greater than the iteration period required by real-time constraints.

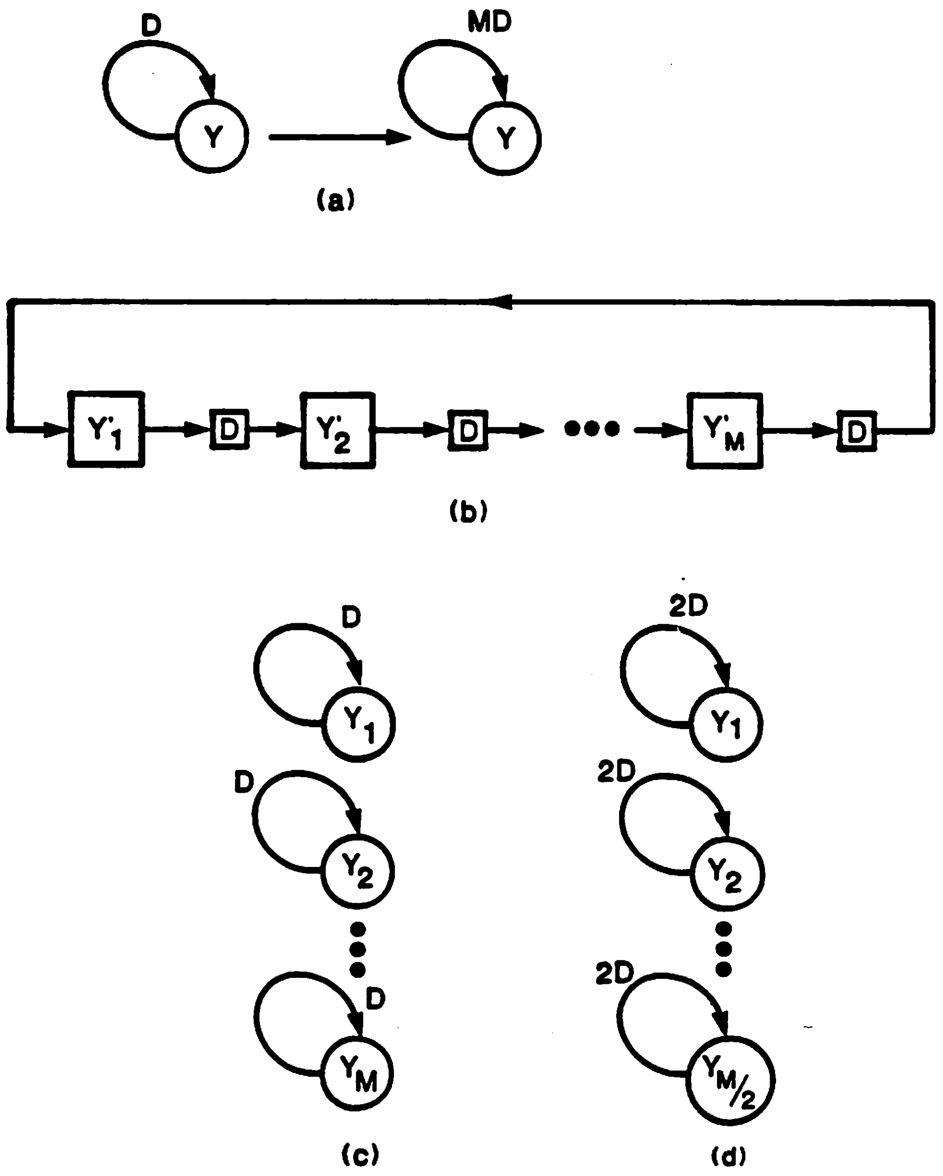


Fig. 7.1: (a) Transformation of a single node program by look-ahead, (b) Pipelining and retiming of the transformed program, (c) Unfolding of the transformed program by a factor of M. This corresponds to a block implementation with block size M. (d) Unfolding of the transformed program by a factor of $M/2$. This corresponds to a block implementation with block size $M/2$ and for 2-stages of loop pipelining.

7.2. SINGLE-NODE PROGRAMS

Consider the single node program in Fig. 7.1(a), which shows the node variable to be Y with a self loop containing a single delay. The self loop implies that the invocation y_n is a function of the past invocations $y_{n-1}, y_{n-2}, ..., y_{n-N}$, where N represents the memory of the system, also called the order of the system. Let the iteration period bound of this program be t_y. We can apply the scattered look-ahead algorithm described in chapter 2 of this thesis to transform the program to another equivalent program with the same variable Y with a self loop containing M delays (see Fig. 7.1(a)). In the transformed program, the invocation y_n is a function of $y_{n-M}, y_{n-2M}, ..., y_{n-NM}$. The iteration period bound of the new program is improved by a factor of M. However, in order to exploit the new iteration period bound, we need to either pipeline and retime the entire loop operation as shown in Fig. 7.1(b), or perform program unfolding (as discussed in chapter 2). If we unfold the transformed program by a factor of M, then we will have M independent or isolated loops, where each loop contains a single delay. This follows from the unfolding property of chapter 2. Furthermore, unfolding by a factor of M leads to a *perfect program*. This perfect program (which corresponds to a block implementation with block size M in the context of earlier chapters) is shown in Fig. 7.1(c). It is also possible to combine pipelining and retiming with program unfolding. This operation corresponds to unfolding by a divisor of M. Fig. 7.1(d) shows unfolded program with unfolding factor $M/2$. The program in Fig. 7.1(d) has $M/2$ isolated loops each containing 2 delays in the self loop (note that delay conservation property is satisfied). In this program, the unfolding factor is $M/2$, and each unfolded program is pipelined (and retimed) by two stages.

7.3. SINGLE-RATE ITERATIVE PROGRAMS

Consider the two node iterative program in Fig. 7.2(a). Assume that the execution time of node X is greater than that of node Y. The iteration bound of this program is

$$T_{\infty} = Max\,(t_x, \frac{t_x + t_y}{2}) = t_x \ . \tag{7.1}$$

This iteration bound can be improved by performing local transformation around node X (since node X is the critical node). If we perform scattered look-ahead at node X by a factor of 2 (assume the operation at node X is linear), then we obtain the transformed program shown in Fig. 7.2(b), which has an iteration bound of $\frac{t_x + t_y}{2}$. Fig. 7.2(c) shows the unfolded version of the program in Fig. 7.2(b) for an unfolding factor of two, and one can verify that the unfolded program is a perfect program.

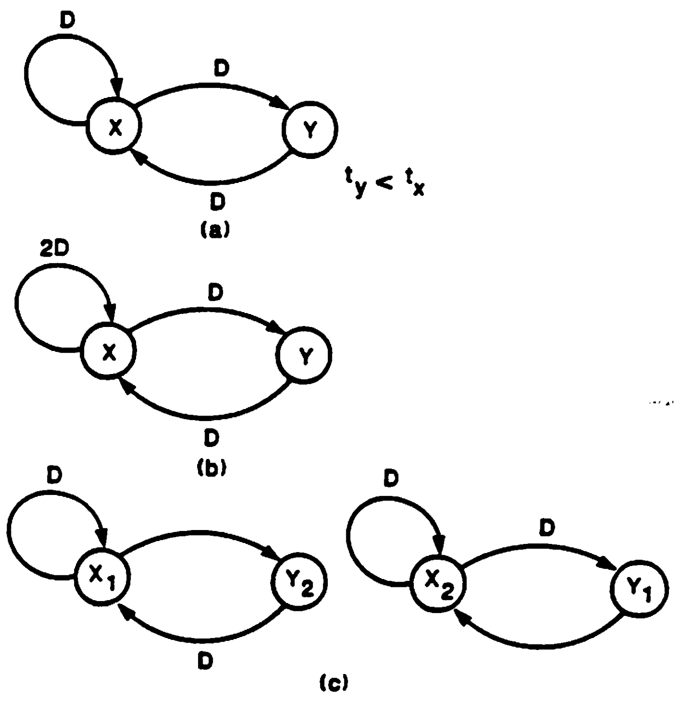


Fig. 7.2: (a) A two-node single-rate iterative Data Flow program, (b) Transformed data flow program obtained by applying look-ahead locally at node X, (c) Unfolding of the program in Fig. 7.2(b) by a factor od 2 leads to a perfect program.

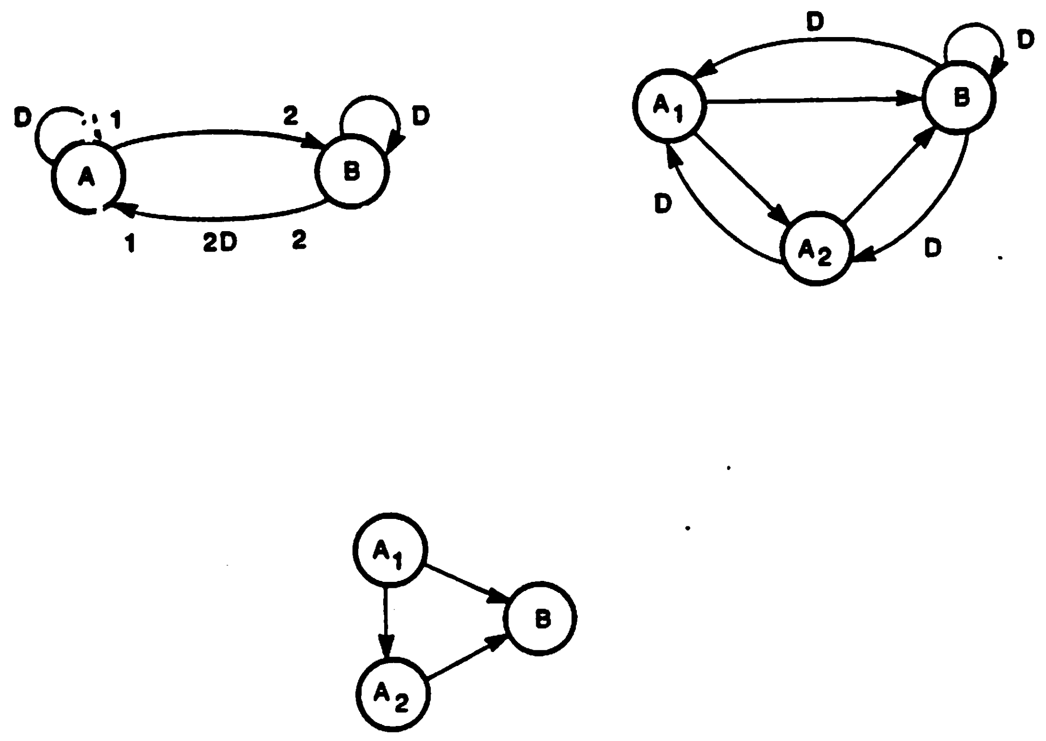


Fig. 7.3: (a) A two-node multiple rate data flow program, (b) An equivalent single rate data flow program, (c) Directed acyclic precedence graph.

7.4. MULTIRATE ITERATIVE PROGRAMS

Now we illustrate the application of the look-ahead computation technique to minimize the iteration period in a class of multiple rate iterative data flow programs.

Consider the multi-rate data flow example of Fig. 7.3(a), where each cycle requires invocation of two instances of the node A and one instance of the node B. The equivalent single-rate data flow program is shown in Fig. 7.3(b) and the corresponding precedence relation is shown in Fig. 7.3(c). From the precedence relation, it can be seen that the iteration period is $(2t_a + t_b)$ for processing of two inputs, where t_a and t_b respectively stand for the time required for executing single instances of nodes A and B respectively.

Assuming the recursive computation associated with node A to be linear in nature, we can apply one-step of look-ahead *locally* at node A to create additional concurrency in the recursive self-loop of node A, which can then be used to make the schedule of the instance A_2 independent of that of A_1 (see Fig. 7.4), and the iteration period is reduced to $(t_a + t_b)$ for processing of two inputs. In Fig. 7.4, the node is marked as A' since it differs slightly from node A in functionality. In order to achieve further concurrency, the steps of look-ahead must be proportionately increased (i.e. for $(L-1)$-steps of look-ahead at node B, $(2L-1)$-steps of look-ahead must be applied at node A'). Of course, the transformed data flow program must be *retimed* in order to actually achieve this improved iteration bound.

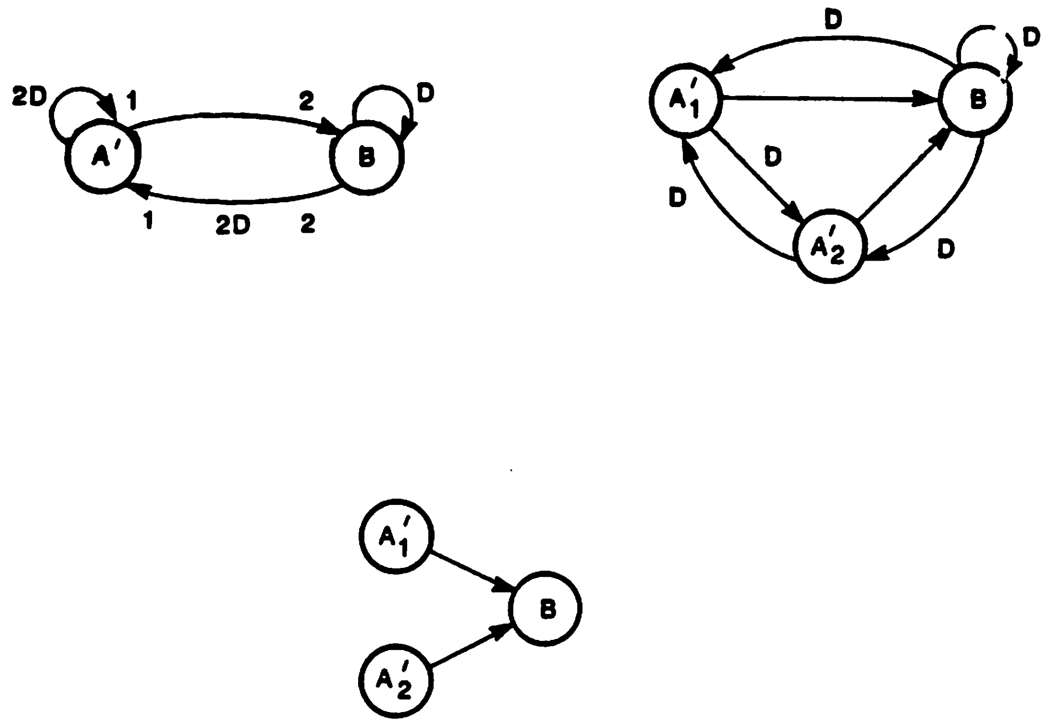


Fig. 7.4: (a) An equivalent multiple rate data flow program obtained by applying look-ahead locally at node A, (b) An equivalent single rate data flow program, (c) Precedence graph of the transformed program.

7.5. CONCLUSION

In this chapter, we have illustrated the use of look-ahead algorithms to perform local transformations in general iterative data flow programs. The key idea is to create additional concurrency in some arc of the *critical loop*, which also belongs to a non-critical loop. The advantage of this technique is that the look-ahead computation penalty is reduced, since the penalty introduced is due to the non-critical loop only. These transformations may prove useful in systems, where the algorithms are unable to meet the requirements of the real-time constraints.

8

FURTHER WORK

The program unfolding and the look-ahead algorithms are both very useful techniques for high performance implementation of signal processing systems. The program unfolding approach leads to rate-optimal fully-static multiprocessor schedules. The look-ahead and its companion algorithms create concurrency, and therefore lead to high sample rate implementations.

Although several algorithms for high performance implementations have been proposed, still many open issues remain. They are

- Study of finite precision effects, which accounts for inexact pole-zero cancellation in pipelined recursive filters,

- Synthesis of pipelined recursive filters directly from filter spectrum using constrained filter design techniques,

- Hardware implementation of an adaptive filter chip using fine-grain pipelining,

- Filter layout generation of digital filters which can be used as a front-end of an existing architecture-specific computer-aided design system,

- Partitioning techniques to obtain multiple chip systems,

- Construction of architecture and interconnection constrained multiprocessor schedules (for homogeneous and non-homogeneous processors),

- Construction of an architecture synthesis system using the techniques discovered in this thesis.

All the above problems can be independently pursued to greater depth. A computer aided design system, capable of simulations in the front-end to account for finite precision effects, and of generating chip layout to meet specified performance constraints would be very useful. Out of the above list of problems, the first one is of theoretical nature, and the remaining involve either hardware and/or software implementations.