ELEKTRİK, VOL.6, NO.2, 1998, ©TÜBİTAK EMO

A Parallel Pipelined Computer Architecture for Digital Signal Processing

Halûk Gümüşkaya
TÜBİTAK-Ulusal Elektronik ve Kriptoloji Araştırma Enstitüsü
P.K.21 41470, Gebze, Kocaeli-TURKEY
Tel: 0.262.6412300/3234, e-mail: haluk@mam.gov.tr
Bülent Örencik
İstanbul Teknik Üniversitesi Bilgisayar Mühendisliği Bölümü,
Ayazağa 80626, İstanbul-TURKEY
Tel: 0.212.2853590, e-mail: orencik@cs.itu.edu.tr

Abstract

This paper presents a parallel pipelined computer architecture and its six network configurations targeted for the implementation of a wide range of digital signal processing (DSP) algorithms described by both atomic and large grain data flow graphs. The proposed architecture is considered together with programmability, yielding a system solution that combines extensive concurrency with simple programming. It is an SSIMD (Skewed Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data) machine depending on the algorithms implemented and the programming methodologies. The concurrency that can be exploited by the algorithms using this Parallel pipelined architecture is both temporal and spatial concurrency. The third level of concurrency (second spatial concurrency) can be also achieved by using input and output synchronized circular buses. An experimental parallel pipelined AdEPar (Advanced Educational Parallel) DSP system architecture, and its network configurations using printed circuit boards (as processing elements-PEs) based on DSP processors were designed and implemented. The hardware debugging of parallel programs and development of other high level programming tools such as automatic task schedulers and code generators) are relatively easy for the AdEPar architecture compared with other architectures having complicated interprocessor communications.

Key words: Digital signal processing, parallel processing, parallel pipelined architecture.

1. Introduction

The implementation of DSP algorithms using multiprocessors is a special case of the parallel processing [1-3]. DSP algorithms are known to be highly parallel. They are at the coarse and fine grain levels characterized by regular, computationally intense structures. Since they do not contain data dependent branching, the detection of parallelism is relatively easy. This parallelism can be exploited and implemented with specialized multiprocessors. The problem that remains is the translation and integration of these properties into economical and practical implementations, and the transfer of technology to applications.

The design of the parallel DSP computer architecture described in this paper was driven by target DSP algorithms and by a set of goals. First of all, it must perform a wide range of DSP algorithms.

Second, it must be scalable and expandable so that application specific systems could easily be configured according to the needs. The interconnection topology should be as simple as possible yet has to support the target DSP algorithms cleanly and efficiently. Our next goal was to design an environment which should address the implementation issues of multiprocessor systems such as processor and memory availability constraints, architectural features of DSP processors in the target architectures, processor communication delays, debugging of parallel programs and development of high level programming environments. The hardware of the experimental parallel pipelined AdEPar (Advanced Educational Parallel) DSP system architecture based on DSP processors which can be configured as different pipelined networks was designed and implemented with these considerations [3-16].

In the next sections, first, the target DSP algorithms, architectures and their concurrency will be given. Next, a Processing Element (PE) of AdEPar, its network configurations and DSP algorithms which can be efficiently mapped onto these networks will be presented. The communication modes and the implementation of DSP algorithms on AdEPar will be discussed. Finally, the AdEPar software environment is presented.

2. Target DSP Algorithms and Architectures

Successful matching of DSP algorithms and architectures requires a fundamental understanding of problem structures and an appropriate mathematical framework, as well as an appreciation of architecture and implementation constraints. DSP algorithms encompass a variety of mathematical and algorithmic techniques and are dominated by transform techniques, convolution, filtering and linear algebraic methods. These algorithms possess properties such as regularity, recursiveness, and locality. Most DSP algorithms can be decomposed into computational sections that can be processed on parallel pipelined processors. The computation graphs of DSP algorithms have both spatial and temporal (pipelining) concurrency. To maximize the throughput, both types have to be exploited. Many techniques have been proposed to schedule DSP programs onto multiprocessors. These techniques can be classified into two categories. Many approaches in the first class exploit only spatial concurrency [17-20]. The second class of concurrency [21-22] based on cyclostatic scheduling takes advantage of the stream processing nature of DSP to exploit temporal concurrency between iterations. While these can yield optimum solutions, their exponential search strategies preclude a practical implementation of a wide range of DSP applications.

Although many hardware and software solutions have been proposed for the multiprocessing scheduling problem of DSP algorithms, none has been able to fully exploit all types of concurrency present in DSP algorithms, and at the same time, address practical issues of multiprocessor systems such as processor and memory availability constraints, architectural features of DSP processors in the target architectures, processor communication delays, debugging of parallel programs and development of high level programming environments. At the same time, many of these are specific solutions for a small class of DSP problems.

In the AdEPar integrated hardware and software architecture, we simultaneously consider pipelining and parallel execution (two levels of spatial concurrency) of DSP algorithms to maximize throughput while meeting resource constraints, and the programmability of the system. The architecture of AdEPar is targeted for the multiprocessing scheduling problem of DSP algorithms which are commonly used in the DSP community and described by both atomic data flow (ADF) and large grain data flow (LGDF) graphs [23]. The structures of these algorithms are modular, and based on local feedback.

The AdEPar architecture is mainly based on pipelining techniques and also has similar characteristics like systolic or wavefront array architectures. The use of pipelining techniques to speed up the processing of data manipulations results in enhanced throughput for data operations [24]. The performance achievable by the use of pipelining is a function of many factors. The divisibility of the original task, the memory delays and the speed of sections all influence the basic data rate at which the pipe can operate. In addition to these, the independence of the operands needed by sections, and the ability of the system to handle data at a sufficiently high rate to keep the pipeline full also affect the operation of the pipe. The concept of systolic architecture is a general methodology for mapping high-level computations into hardware structures [25]. In a systolic system, data flows from the computer memory or an input device in a rhythmic fashion, passing through many processing elements before it returns to the memory or an output device. Systolic systems can be one or two dimensional to make use of higher degrees of parallelism. Moreover, to implement a variety of computations, data flow in a systolic system may be at multiple speeds in multiple directions-both inputs and (partial) results flow, whereas only results flow in classical pipelined systems. A systolic system is easy to implement and reconfigure because of its regularity and modularity.

The proposed parallel pipelined architecture and the experimental AdEPar DSP system architecture support the following six network topologies:

1. Pipelined Linear Network (PLN)
2. Parallel Pipelined Linear Networks (PPLN)
3. Pipelined Ring Network (PRN)
4. Parallel Pipelined Ring Networks (PPRN)
5. Pipelined Linear Networks with Forward Diagonal Switches (PLN with FDS)
6. Parallel Pipelined Linear Networks with Forward Diagonal Switches (PPLN with FDS)

The AdEPar architecture configurations are SSIMD (Skewed Single Instruction Single Data) or MIMD (Multiple Instruction Multiple Data) machines depending on the algorithms implemented and the programming methodologies. The first concurrency in the scheduling algorithms of these architecture configurations is temporal concurrency, where chains of tasks are divided into stages, with every stage handling results obtained from the previous stage. The second is spatial concurrency (parallelism), where tasks are executed by several PEs simultaneously. The third level of concurrency (second spatial concurrency) can be achieved by connecting the same pipelined structures in parallel through the input and output circular buses (ICB/OCB) with demultiplexer/multiplexer structures. This will be called IOCB spatial concurrency. The proposed algorithms, architectures and their concurrency are listed in Table 1 and will be explained below.

Table 1. The target algorithms, architectures and their concurrency.

Algorithms	**Architecture**	**Concurrency**
ADF graphs (Simple filters)	PLN (SSIMD)	Temporal
	PPLN (SSIMD)	Temporal, IOCB spatial
	PRN (SSIMD)	Spatial
	PPRN (SSIMD)	Spatial, IOCB spatial
LGDF graphs (General DSP algorithms)	PLN (MIMD)	Temporal
	PPLN (SSIMD)	Temporal, IOCB spatial
	PLN with FDS (MIMD)	Spatial
	PPLN with FDS (SSIMD)	Temporal, spatial, IOCB spatial

A multiprocessor system that uses DSP processors is inherently an MIMD machine, and there is no requirement that the processors be synchronized in any way. For the implementation of DSP algorithms, however, it is more important that the multiprocessor exhibit SIMD or synchronous MIMD modes. In other words, every cycle or every processor is devoted directly to the arithmetic implementation of the algorithm. Once the data areas and the registers are initialized, the processors commence synchronous operation, maintaining correct data precedences, if properly programmed. No overhead is deemed necessary.

3. A PE of AdEPar and Two-PE Base Architecture

A mother board was designed as a stand-alone board or a PE of the parallel AdEPar architecture at the Electronics Department of TÜBİTAK-MAM [3-5,13]. The AdEPar project was funded for three years. After the construction of the first PE prototype, many different PEs were also designed using customs ICs and off-the-shelf components suitable for the IBM PC/AT computer.

The AdEPar network configurations are constructed using these mother boards and the AdEPar PEs are accessed through a host computer as shown in Figure 1. The major functions of the host computer are to develop high level DSP algorithms using the AdEPar Integrated Simulation and Implementation Environment for DSP [6,8,9,15,16], to load application programs and data to the PEs, and to monitor and debug the execution of the programs on each PE using the AdEPar Parallel Hardware Debugger.

A PE consists of a DSP processor (TMS320C25) running at 40 MHz clock frequency, a local program memory, a data memory, communication ports and a dual port memory (DPM) for host computer interface [13]. The interface between the PE and the host PC and the interprocessor communication can be the most critical factor in the system and is often the primary bottleneck to throughput. There are several possible memory and I/O interfaces including an I/O port, a memory mapped I/O, a dual port memory, FIFO buffers, a DMA port, and a serial port. In the AdEPar DSP system, a DPM interface was chosen due to its higher performance/cost ratio.

Each PE has a dual port RAM interface for PC communication and a 2K$\times$16-bit dual port RAM is used to transfer data between the PC and the PE. The communication between two DSP processors is also performed by a 1K$\times$16-bit dual port RAM. This DPM interprocessor communication scheme features simplicity, speed, modularity and configurability to multiprocessing systems such as linear arrays, triangular arrays, meshes, systolic trees and hypercubes.

The communication ports of a PE are used to connect a PE to neighboring PEs through a DPM interface. A communication board is used to connect two PEs and it has a DPM for PEs communication and switches for use in communication and synchronization mechanisms as shown in Figure 2. An I/O device can be also interfaced to a PE using one of these communication ports. The communication between processors which are not nearest neighbors is achieved by passing messages through reserved areas in the dual port memories. This is a simple and low cost communication method.

Two processors can use three different synchronization mechanisms:

1. Special 16-bit flags in DPM reserved areas for processor synchronization.

2. Polling the XF (External Flag) pin of the other processor using its own BIO pin through the communication board.

3. Interrupts through the communication board (The XF pin of one processor is connected to the interrupt pin (INT2) of the other processor).

The XF pin of one processor can activate INT2 or BIO inputs of the other processor. The user selects INT2 or BIO pins by setting the switches on the communication board.

Forward Diagonal Switches (FDS) are used to connect a PE to neighboring PEs in two dimensional network configurations (PLN with FDS and PPLN with FDS configurations). FDS can be implemented using the DPM or FIFO memory interface techniques.

The base architecture has two PEs connected to each other through a communication board having a DPM interface as shown in Figure 2. This interface can be considered as an independently accessible memory buffer located between pipelined PEs. The AdEPar architecture configurations are based on this simple interconnection network regardless of the algorithms to be implemented. This not only makes it easy to increase the number of PEs in the system but also is good for implementing various algorithms with different structures.

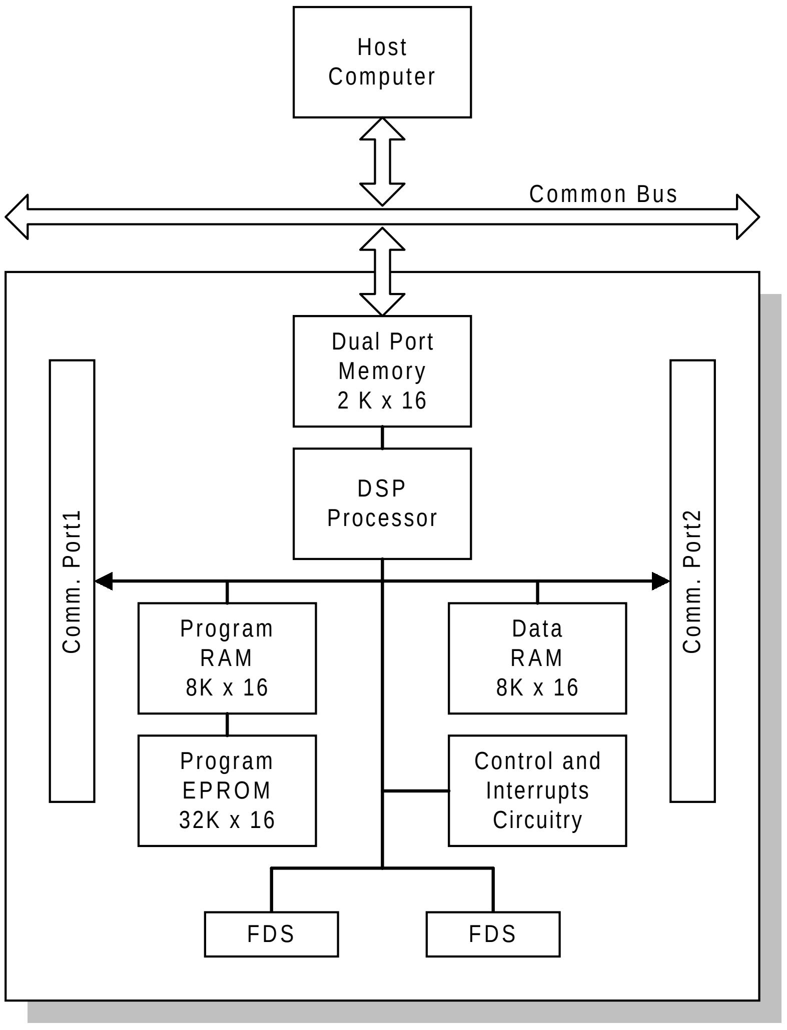


Figure 1. A PE of AdEPar

The hardware debugging of parallel programs is much more difficult than that of sequential programs and is a hot research area [26]. For this two-PE base architecture, a powerful window-based hardware debugger that displays the states of the PEs simultaneously was developed and it was shown that the hardware debugging of parallel programs on the AdEPar architecture is relatively easier compared with other complex architectures [13]. The parallel debugger displays the internal registers, the data memory and provides a reverse assembly of program memory. Single step-run, run-to-break point operations are possible for two PEs. The other high level programming tools (such as automatic task schedulers and code generators), which enable the user to efficiently and easily use and exploit the AdEPar system features, are also relatively easy to develop for this architecture compared with other architectures with complicated interprocessor communications.

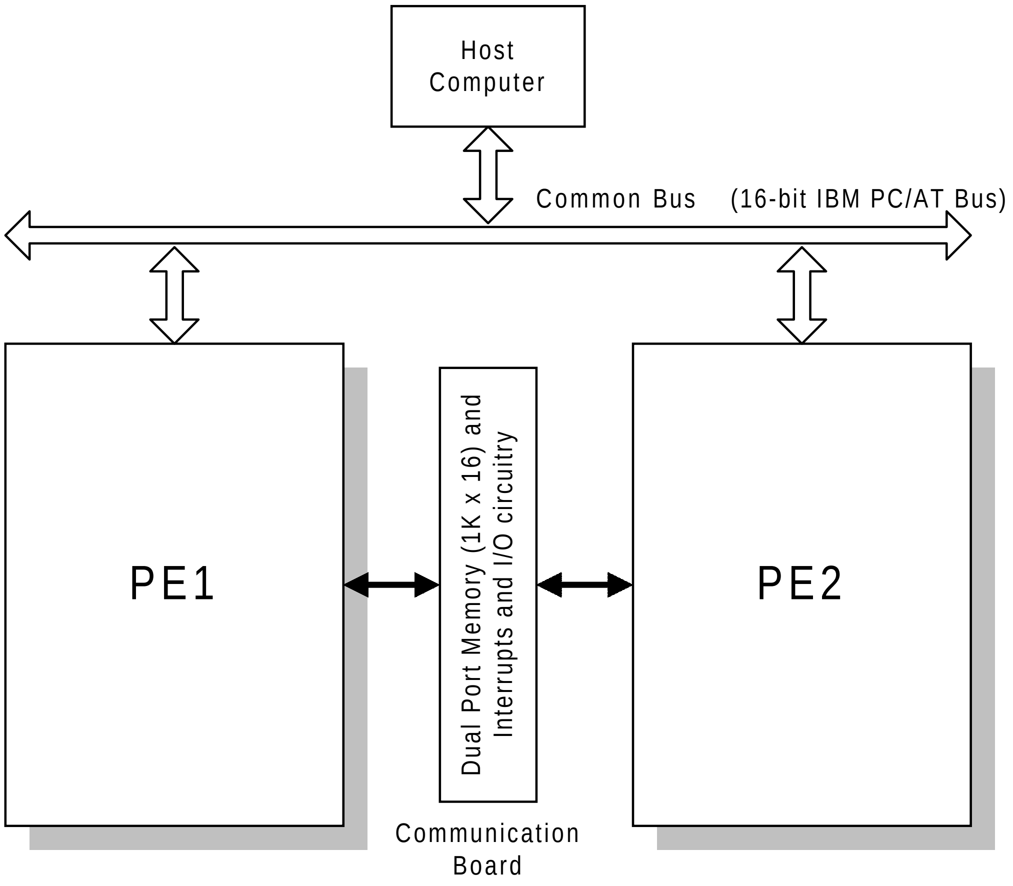

Host Computer
Common Bus (16-bit IBM PC/AT Bus)
PE1
Dual Port Memory (1K x 16) and Interrupts and I/O circuitry
PE2
Communication Board

Figure 2. A two-PE base architecture.

This two-PE base system and its hardware debugger has been mostly used in practical experiments. The base architecture has allowed us to develop, trace and examine real parallel DSP programs, and then to generalize these results to other network configurations. The results obtained on the base system have been used in the TMS320C25 simulator program and the AdEPar DSP environment. The real AdEPar hardware system, the TMS320C25 simulator results and the AdEPar DSP environment helped us to model practical parallel architectures for a wide range of DSP algorithms described by both ADF and LGDF graphs.

4. PLN and PPLN Architectures

The network configurations of AdEPar are constructed using dual port memories between PEs. These memory elements synchronize the actions of the system and hold the data needed by the PEs. When the two-PE base architecture is extended linearly as shown in Figure 3 (a), a PLN architecture is obtained. A PPLN architecture can be constructed by connecting PLNs to each other in parallel through ICB/OCB as shown in Figure 3 (b).

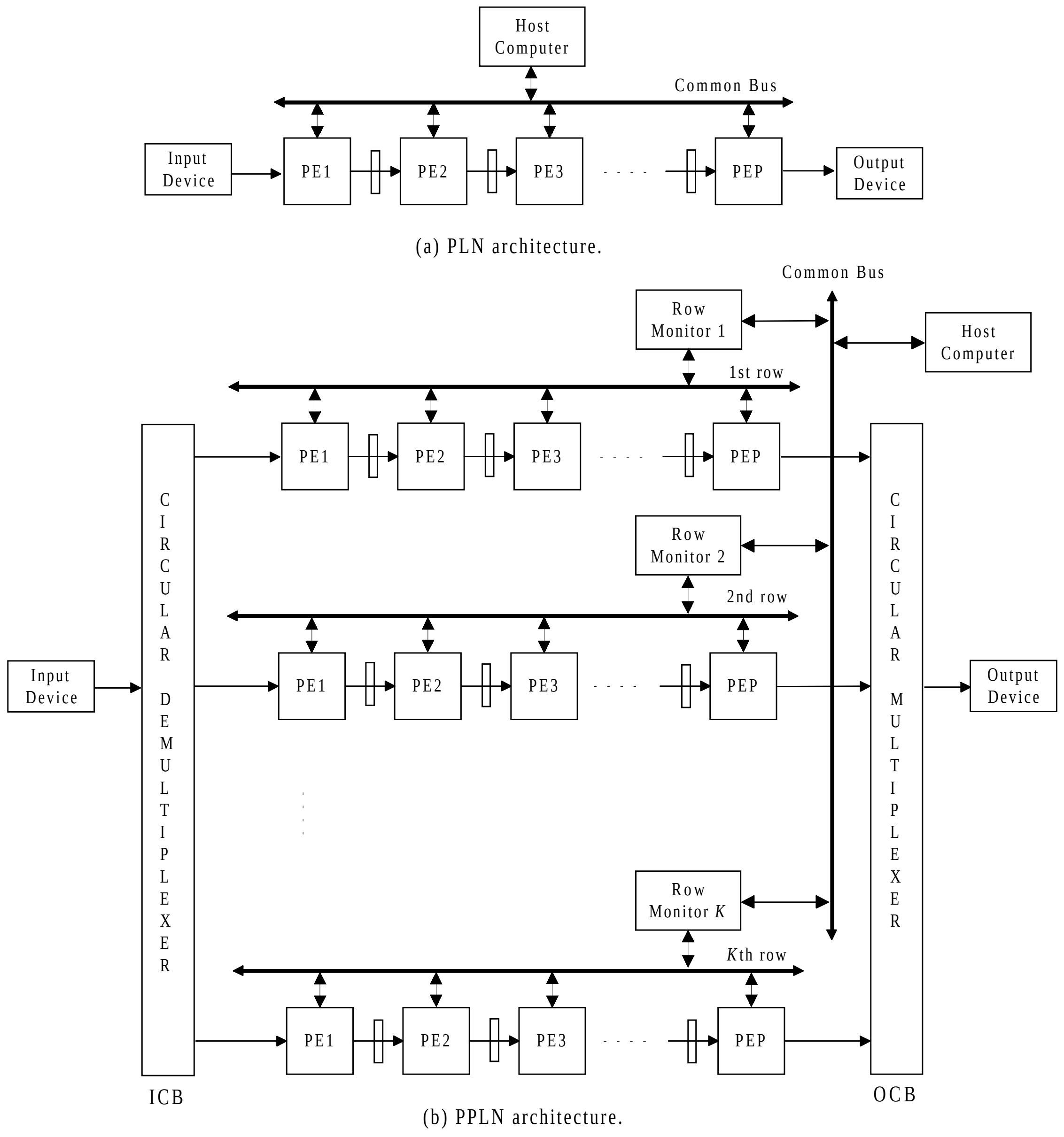


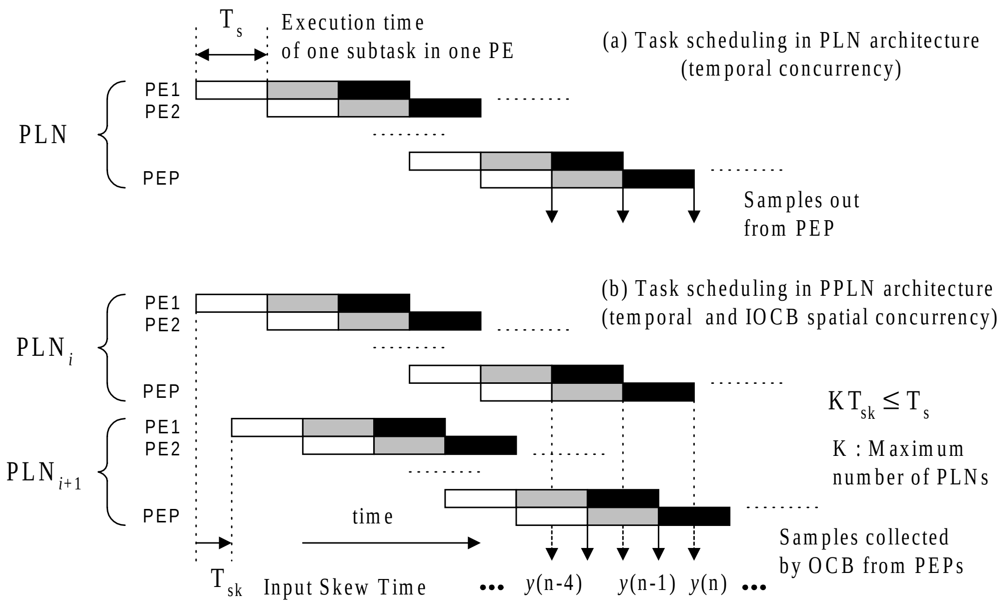
Figure 3. PLN and PPLN architectures.

The input data to the PLN architecture is supplied by an input device from the first PE, namely PE1, and the output data is collected from the last PE, namely PEP, in the PLN chain. The input device of the PPLN architecture supplies the input data to the first PEs in rotation through the ICB. The output device also collects the output data from the last PEs in rotation through the OCB. The ICB and OCB are synchronized to give the highest throughput rate. Each row of PEs in a PLN architecture is monitored by a row monitor, a personal computer collecting necessary low level debug information for the high level debugger running at the host computer in the PPLN architecture. These row monitors are connected to the host computer through a high speed common bus.

These two PLN and PPLN architectures are well suited for the class of FFT, IIR, FIR, basic and adaptive lattice filters, and similar structures which are modular, based on local feedback and can be easily pipelined. They are also good targets to map many DSP algorithms specified by LGDF graphs as described below.

A PLN architecture having P number of PEs produces results P times faster than the nonpipelined implementation. This observation is valid if we disregard the time required for the storage function in dual port memories. This is quite a reasonable assumption for the target PLN architecture and the target algorithms; since a dual port memory access requires one processor cycle and the synchronization code between PEs needs only a few memory cycles. The second assumption is the continuous data input and output flow. This is also true for real time operations, since the real time DSP algorithms operate on infinite data stream. Once the pipeline fill time operation has been completed, the remaining operations will follow at the rate of one result per clock cycle; then the system starts to operate at its highest efficiency.

The timing diagram given in Figure 4 shows the overlapped execution of operations in the PLN and PPLN architecture. In Figure 4 (b), $KT_{sk} \leq T_s$, where K is the number of rows, each having P number of PEs, and T_{sk} is the input data skew time. For $K = T_s/T_{sk}$, the speedup becomes KP in the PPLN architecture. In this architecture the throughput of the system depends on the data rate of the ICB and OCB regardless of the algorithm implemented.



T_s
Execution time of one subtask in one PE
(a) Task scheduling in PLN architecture (temporal concurrency)
PLN
PE1
PE2
PEP
Samples out from PEP
(b) Task scheduling in PPLN architecture (temporal and IOCB spatial concurrency)
PLN_i
PLN_{i+1}
$KT_{sk} \leq T_s$
K : Maximum number of PLNs
time
Samples collected by OCB from PEPs
T_{sk} Input Skew Time
••• $y(n-4)$ $y(n-1)$ $y(n)$ •••

Figure 4. Overlapped execution of operations in the PLN (a) and PPLN (b) architectures.

5. The Performance Measures of Digital Filtering Algorithms on the PLN and PPLN Architectures

In multiprocessor systems there are generally two kinds of performance to be measured: computational performance and communication performance. The detailed performance measures of the AdEPar architecture were published in [10,13,14]. In these measures, FFT (Fast Fourier Transform), DES (Data Encryption Standard), IIR (Infinite Impulse Response), FIR (Finite Impulse Response) and lattice filter algorithms were used. These algorithms are computationally intensive and require heavy processor communication traffic. In this section, the task partition and scheduling strategies are shown by simple IIR and FIR digital filtering examples and the performance measures are also presented to give some idea about the AdEPar architecture performance measures.

In the algorithms, we choose "a number of instructions or cycles per generic unit of computation" to describe the execution of DSP algorithms on AdEPar. Such a choice leads to very useful and descriptive units such as: (i) the number of instruction per tap (FIR filters), (ii) the number of instructions per biquad (IIR filters), (iii) the number of instructions per butterfly (FFT), and (iv) the number of instructions per section (lattice structures). Such benchmarking effectively illustrates the power of the algorithm. However, if necessary, it can be converted easily to speed (absolute time, frequency) for a specific application and machine characteristics.

5.1. Pipelined Second Order IIR Cascades

A simple IIR filter example will be given as a parallelization and implementation example of DSP algorithms described by ADF graphs for PLN and PPLN architectures. When implementing IIR filters, the direct form II (second order) structure will be used. Since the direct form II has the minimum number of delays, it requires the minimum number of storage registers for computation. This structure has advantages for minimizing the amount of data memory used in the implementation of IIR filters. The other advantage is the fact that with proper ordering (pipelining) of the second order cascades, the resulting filter has better immunity to quantization noise than the direct form IIR filter implementation.

PLN Implementation: The simplified program and control structures in each PE and overall data flow in the PLN architecture for the implementation of IIR filters are shown in Figure 5. This implementation uses DPM reserved areas for processor communication and synchronization. The condition variables Flag1 to FlagP-1 are special 16-bit flags in DPM reserved areas and are used for processor synchronization. Actually, they are memory locations which can be accessed by both processors in the PLN chain. For example, PE1 has its own flag, Flag1, indicating the completion of its subtask. After PE1 produces its data it checks if its old data was consumed by the next processor PE2. If its old data was consumed, it writes its new data to the right PDM reserved area, and sets its Flag1 to 1, and reads a new data sample from the input device. Similarly, PE2 also has a flag, Flag2, indicating the completion of PE2's subtask. After PE2 produces its data it checks if its old data was consumed by the next processor PE3. If its old data was consumed, it writes its new data to the right PDM reserved area, and sets its Flag2 to 1, and reads Flag1 from the left DPM reserved area to get the next data from PE1 in a synchronized manner. The other processing elements perform similar synchronized operations to produce filtered data from the last PE, PEP.

In this particular example, if the given IIR filter has an order of eight, and the system has two PEs available, the schedule that has 100 % PE efficiency distributes two second order filter sections to each PE. If we have an unlimited number of PEs, the optimum schedule has four PEs and each PE implements one second order filter section. In this implementation approach, the maximum sampling rate depends on the execution time of one second order section. As the filter order increases, the sampling rate does not change if we have an unlimited number of PEs and each PE implements one second order section. In the PLN implementation, at the maximum sampling rate, each PE has the same program, except the first and the last PE. The first PE samples data from an input device, and the last PE sends the filtered sample to an output device. They also perform the second order cascades. The execution time of each program on PEs is skewed, thus the system can be considered an SSIMD machine.

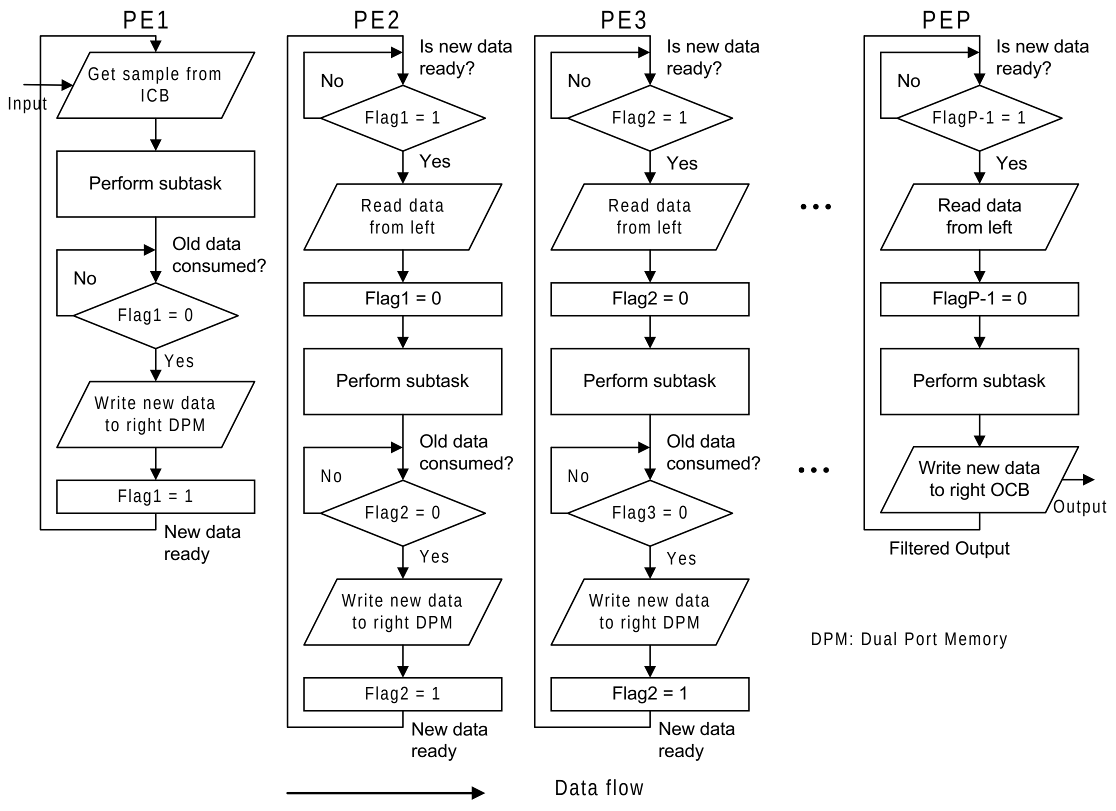


Figure 5. The simplified program and control structures in each PE and overall data flow.

PPLN Implementation: The PPLN configuration for the implementation of the 8-order IIR filter is shown in Figure 6. In this case the input data is divided into K rows each having P number of PEs by ICB and, the output data is collected from PEPs by OCB in rotation. This type of data division to PEs is similar to the interleaved-memory operation in some parallel computers [1,2]. This is the second level concurrency, IOCB spatial concurrency, in pipelined data processing.

In general, if K rows each having P number of PEs are used in the PPLN architecture, the input data sampling time becomes the input data acquisition skew time T_{sk}, where $T_{sk} = T_s/K$. Two rows each having four PEs are used in this particular example.

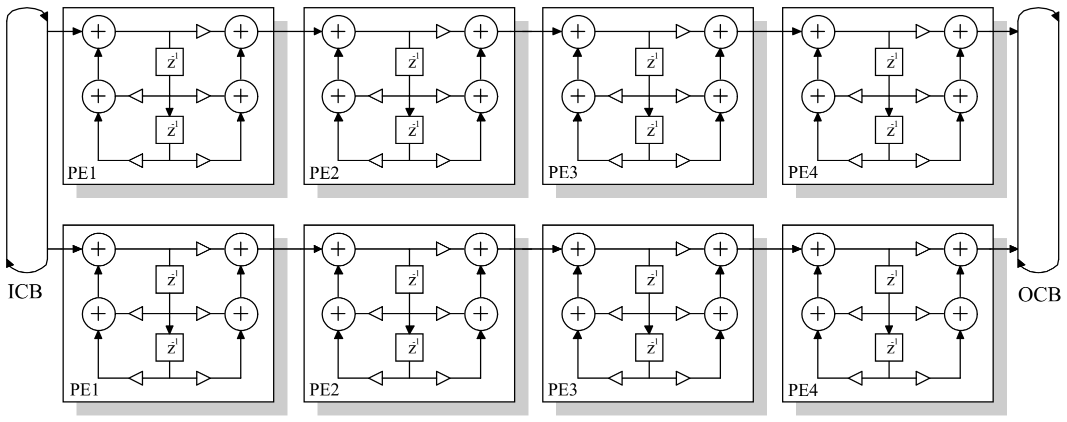


Figure 6. 8-PE PPLN configuration for the 8-order IIR filter.

Figure 7 shows the execution times of 2-PE, 4-PE and 8-PE schedules to implement the 8-order IIR filter on 2 and 4-PE PLN (a), (b) and 8-PE PPLN (c).

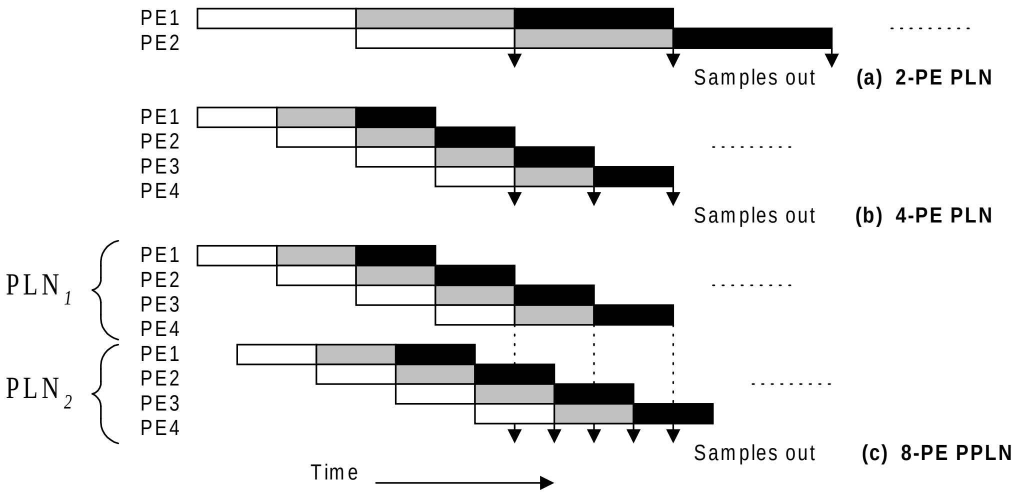


Figure 7. The scheduling diagrams of PEs for 8-order IIR filter on 2 and 4-PE PLN (a), (b) and 8-PE PPLN (c) architectures.

The same parallelization and implementation approach as given for the IIR filter example can be applied to the similar filter structures and DSP algorithms such as FIR, basic and adaptive lattice filters. The structures of these filters and algorithms are composed of modular sections connected in cascade and are based on local feedback. The input signals while propagating from left to right, are processed by each section and finally sent out from the last section. For this kind of structure, the pipelined operation of PLN or PPLN architectures can be efficiently and easily applied.

5.2. Pipelined FIR Filter

The direct form of the FIR filter shown in Figure 8 (a) is not suitable for parallel execution in the target PLN pipelined architecture. The transposed form of the same filter shown in Figure 8 (b) does the same operation and has a parallel structure suitable for pipelined operation. In the FIR implementations, the transposed form was used.

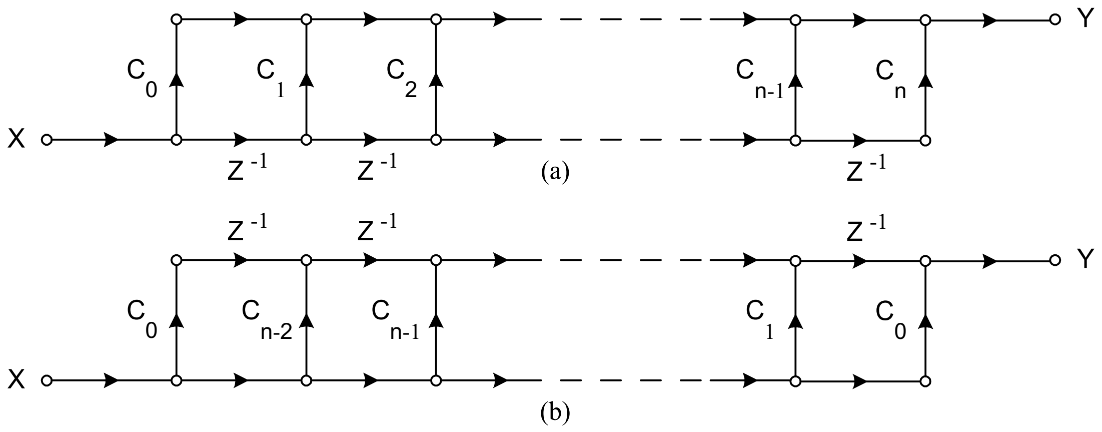


Figure 8. FIR filter: (a) direct structure, and (b) transposed form.

The performance measures of different order IIR and FIR filters are shown in Figure 9.

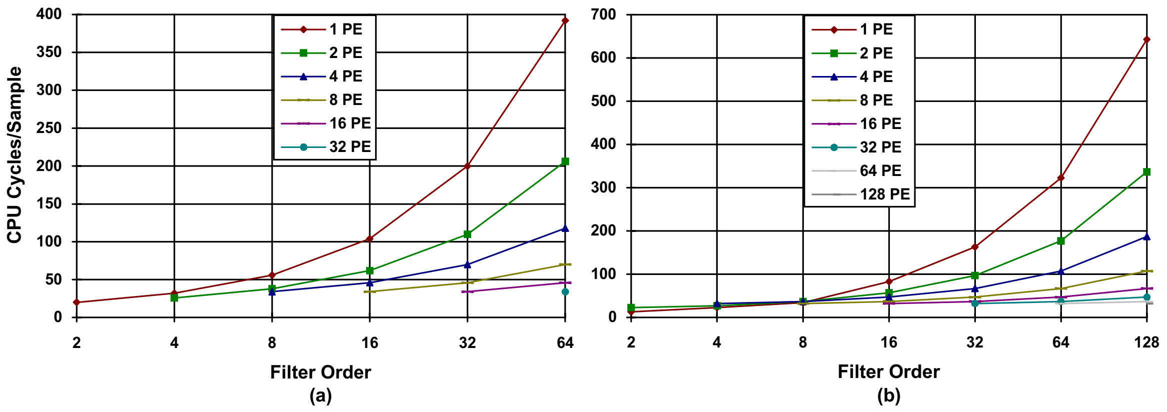


Figure 9. The performance measures of IIR (a) and FIR (b) filters on the PLN architecture.

As shown, the optimum results for the PLN architecture were obtained using 4 or 8-PE based systems in the implementations of IIR and FIR filters. If the number of PEs is increased above 8, it does not increase the performance remarkably.

6. PRN and PPRN Architectures

A PRN architecture is constructed by connecting the first and the last PE to each other in the PLN architecture as shown in Figure 10 (a). The second two dimensional architecture, PPRN, shown in Figure 10 (b) is a generalization of a PRN architecture and it has a similar scheduling as PPLN architecture. In these architectures, the inputs and outputs of PEs are connected to the I/O devices in parallel, usually A/D and D/A converters, through the common I/O bus. The parallel connection equally distributes the I/O operations to all PEs, and thereby reduces the I/O overhead in each PE.

An approach to high-speed digital filtering algorithms is to use block processing algorithms processing sequences of input samples instead of processing one sample at a time [27]. The primary motivation behind the block processing approach is the possibility of using FFT techniques for intermediate computations. Block structures promise to be computationally more efficient for high-order filters and less sensitive to round-off errors and coefficient accuracy, which are often limiting factors for recursive filter realizations.

The basic block processing algorithms for the IIR, FIR, basic and adaptive lattice and some other filter structures can be efficiently implemented on a PRN architecture [11,28]. Consider an Mth order IIR digital filter characterized by a transfer function given by

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}{1 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_M z^{-M}} = \frac{\sum_{k=0}^{N} a_k z^{-k}}{1 + \sum_{k=1}^{M} b_k z^{-k}} \tag{1}$$

where the a_k and b_k are real coefficients. In terms of convolution, (1) can be written as

$$y_n * b_n = x_n * a_n \tag{2}$$

where y_n and x_n are the output and input sequences, respectively. Writing (2) in matrix form and applying block partitioning with block length $L \geq M$, Burrus [27] arrived at the basic block structure given by

$$Y_k = KY_{k-1} + H_0 X_k + H_1 X_{k-1} \tag{3}$$

where

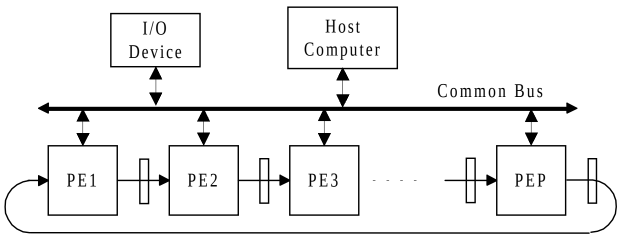
$$K = -B_0^{-1} B_1, \quad H_0 = B_0^{-1} A_0, \quad H_1 = B^{-1} A_1 \tag{4}$$

Y_k and X_k are the kth output/input vectors of length L, block number is denoted by subscript. A_0, A_1, B_0, and B_1 are $L \times L$ matrices $(L \geq M)$ formed by filter coefficients.

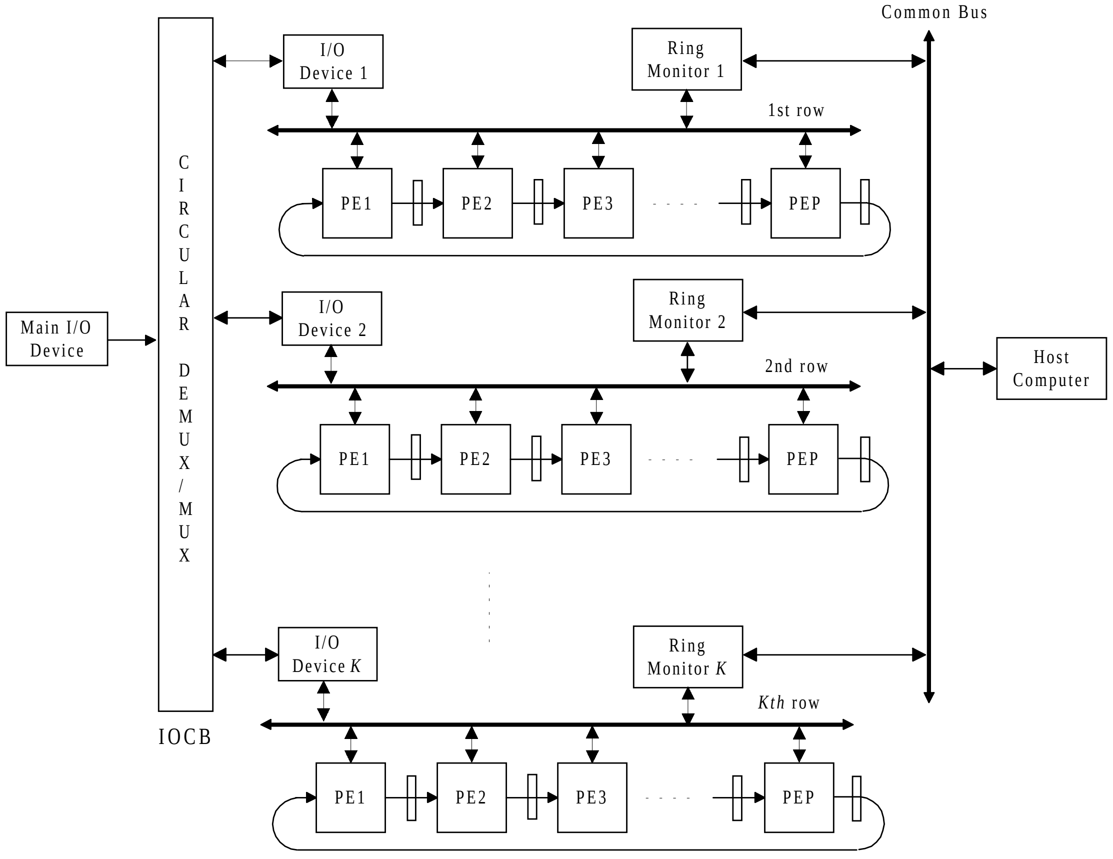
Similarly the block filtering equation for the FIR filter can be obtained as

$$Y_k = A_0 X_k + A_1 X_{k-1} \tag{5}$$

Equations (3) and (5) can be represented as shown in Figure 11.

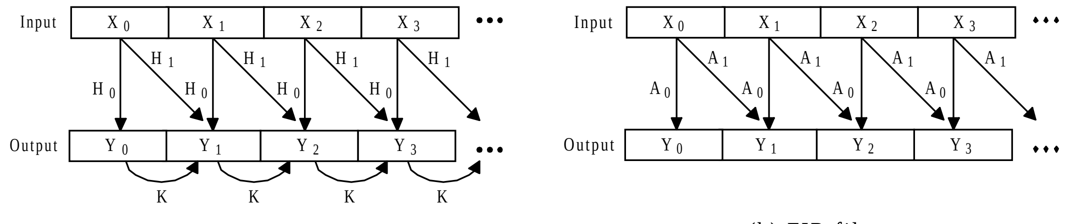


(a) PRN architecture.



(b) PPRN architecture.

Figure 10. PRN and PPRN architectures.



(a) IIR filter

(b) FIR filter

Figure 11. Block processing of a signal for the IIR and FIR filters.

The block implementation achieves the parallelism by dividing the input data into blocks, $X_0, X_1, \ldots X_k$, and assigns a block of data to each PE in rotation at the rate of $R = 1/T_L$, where T_L is the block input time. PEs in the ring network perform all the operations needed for the processing of the given blocks and output result blocks, $Y_0, Y_1, \ldots Y_k$, in rotation. The programming and scheduling of the PEs are as easy as those of a single processor system, since all the PEs need the same copy of a program, except the execution time is skewed, and each processor is working as if it is a stand-alone single PE system for an assigned block of data. Thus, the system can be considered an SSIMD machine. Block processing throughput in the PRN architecture is limited by the complexity of the algorithm [11]. In the implementation on the PRN architecture, T_L must be greater than or equal to the algorithm dependence time limit T_d.

In PPRN, IOCB divides the input signal into K different signals, where K is the number of rows each having P number of PEs. Each I/O device corresponding to a row of PEs collects input data samples coming from the main I/O device and forms input data blocks. These blocks become the input blocks of a particular PRN architecture. The scheduling of PEs in this architecture is similar to the diagram given in Figure 4 for the PPLN architecture. In this PPRN case, if we take $T_{sk} = T_d/K$, where T_{sk} is the input data skew time for a row and, T_d is the algorithm dependence time limit, the system throughput increases to its near theoretical data acquisition throughput limits.

The main drawbacks of PRN and PPRN compared with PLN and PPLN are their large memory requirements. The block processing approach to implement IIR and FIR filters is not practical on the architectures with DSP processors, since the DSP processors have some special architectural features (like MAC, MACD instructions and special addressing modes suitable for DSP) designed for filtering and DSP computations. So there is no need to use block processing techniques to implement high order filters on these architectures. The implementation of filtering algorithms using a DSP processor family gives much higher throughput rates. For example, the IIR second order cascade throughput rate for one PE and at the same time the maximum throughput of the PLN system, 30 cycles/sample, cannot be improved using a block processing approach. This rate can be improved only slightly by using a second dimension in the PPLN architecture provided that high speed ICB/OCB structures are available.

7. PLN and PPLN with FDS Architectures

These two architectures, as shown in Figure 12, use FDS in two dimensional networks and are targeted for the implementation of DSP algorithms described by LGDF graphs. The PLN with FDS architecture has both temporal and spatial concurrency and can be called an MIMD machine, since each PE in this architecture generally will have different programs.

A PLN with FDS architecture implementing a particular DSP algorithm is called a cluster. If the same clusters are connected to each other in parallel through ICB/OCB as shown in Figure 13, a PPLN with FDS architecture is obtained.

This architecture has three levels of concurrency: temporal, spatial, and IOCB spatial. In this architecture, all clusters have the same program structure, except that the execution time is skewed, and each cluster is working as if it is a stand-alone single system for an assigned block of data. Thus, the PPLN with FDS architecture is considered an SSIMD machine.

8. AdEPar Architectural Constraints and Communication Modes

The AdEPar architectural constraints can be summarized as follows:

1. All processors are homogenous.

2. The system is fully synchronous (MIMD or SSIMD).

3. The computational latency of all operations is data independent.

4. For a given operation, all operands can be fetched from adjacent PE or from local storage, and the result can be stored in adjacent PE or in local storage, with computational latency independent of local versus adjacent processor storage location.

5. Interprocessor communications between adjacent processor pairs are independent (with respect to delay, conflict, etc.) of all other adjacent processor communications. If a processor has four adjacent processors it can output to all four adjacent processors (dual: it can input/output from/to all adjacent processors) simultaneously and without conflict.

The first three constraints result from the characteristics of DSP applications where the partitioning of the problem into subtasks is usually regular. Constraints four and five are necessary to reduce the complexity of the search algorithm, which can grow exponentially with these additional degrees of freedom.

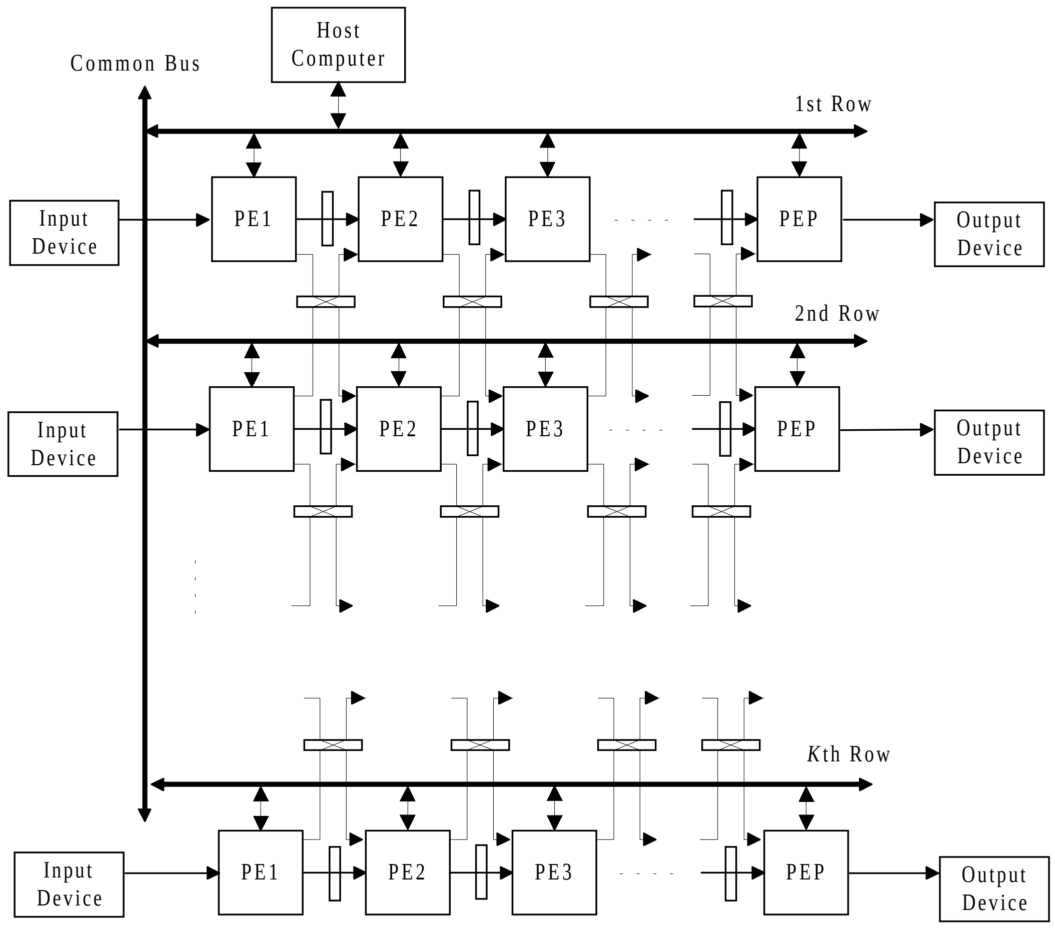


Figure 12. PLN with FDS architecture.

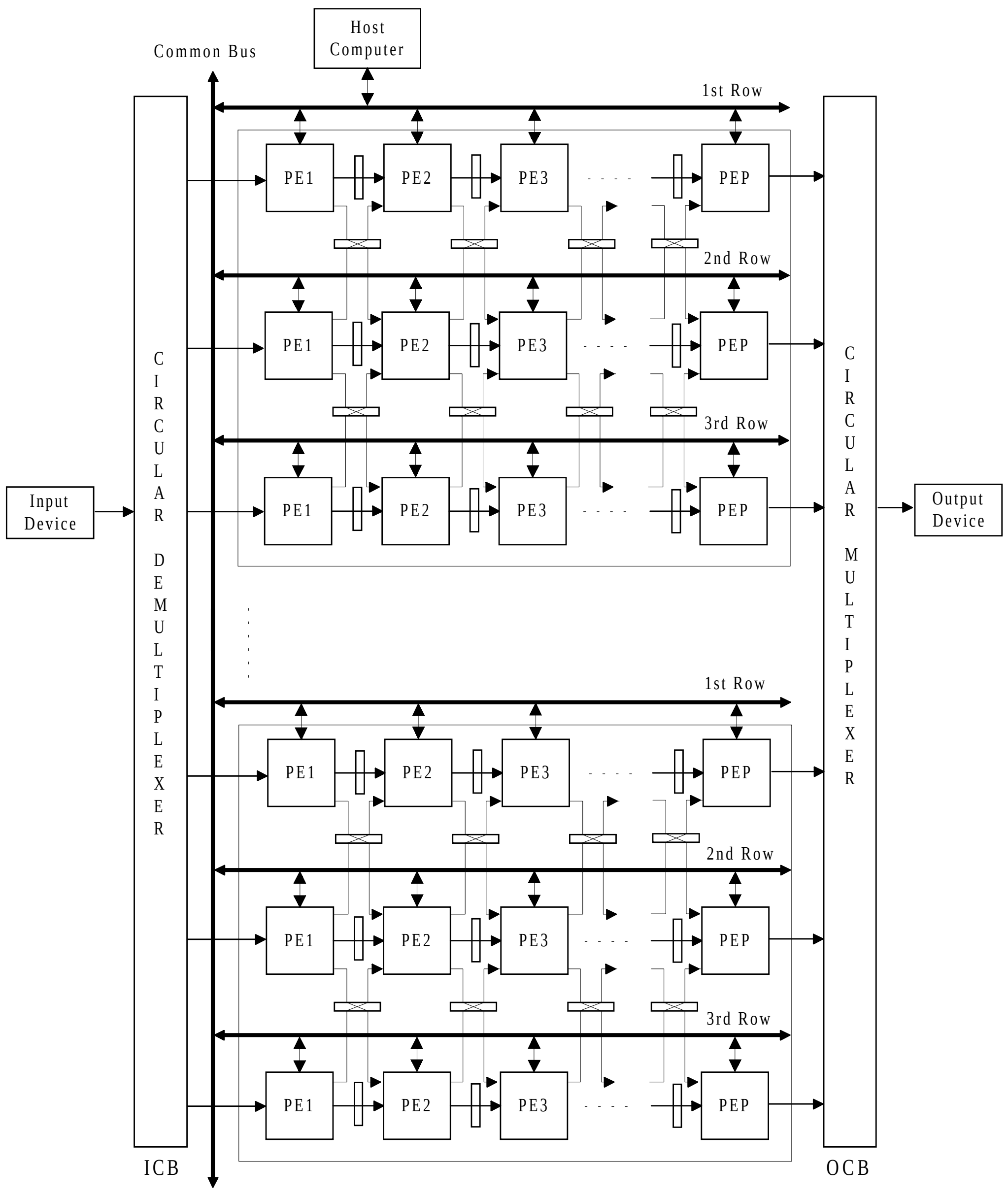


Figure 13. 3-row cluster PPLN with FDS architecture.

The AdEPar architecture admits many communication modes. The bus oriented modes are host initiated transfers and point to point transfers. The communication mode supported by the ring (or linear) pipeline is systolic communication.

First consider the bus oriented communication. Bus transfers may be initiated by either the host or the individual PEs. The memory map of the system permits the host and each of the PEs to directly access any portion of the memory map except the protected system memory. Therefore, the host can initiate transfers directly into any of the PEs local memories. This is the mechanism by which the PEs are loaded with their programs and data. The PEs can also initiate transfers on the bus. Point to point direct memory access requires one more level of bus arbitration. Once the system bus is attained, a second bus arbitration process is initiated for the local bus of the destination PE.

Systolic communications take place over the ring (or linear) pipeline. In systolic communication, the PEs have direct access to the message queue as the message is being sent. Algorithms will exploit this mode by forming a pipeline of PEs that interleave communication with computation. In a typical algorithm, data will enter PE1 from the bus either as a block or on a word by word basis. When PE1 accesses the first element of the data stream for local processing, it will also forward the word onto PE2. Once this is done, PE1 can access the word for its own use, and pass it onto the next PE. In the DSP algorithms of interest, each of the PEs will perform the same task (for many ADF graphs) or a different task (for LGDF graphs), on different data, so that a loose form of program synchronization assures that the load will be balanced and waiting time will be kept to a minimum. Two communication ports on each PE are used as linear or ring connectors in systolic communications. These ports provide a means to connect groups of PEs residing on different cards to be connected into the linear or ring pipeline. With this arrangement, flexible systems can be configured to meet specific processing, memory, and I/O requirements of the application.

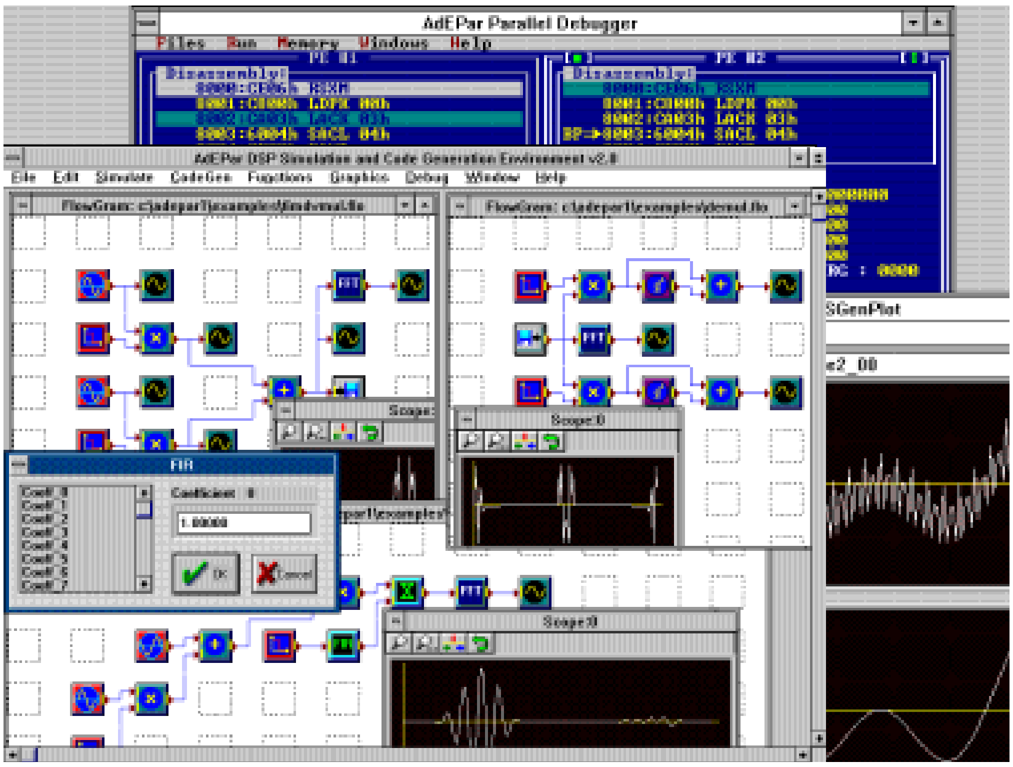
The system can be viewed as a message passing machine. The programs on the PEs communicate using the send and receive message commands. The messages can be routed through the linear or ring pipe or over the common bus. The system can be programmed as a linear or ring systolic array (with broadcast). The systolic mode is the fastest mode if local PE to PE communication is required by the application algorithm. In this mode, a steady data stream flows through the ring network in lockstep with processing. The PE architecture is optimized so that systolic communication and computation overlap. Most of the algorithms of interest make extensive use of the ring (or linear) pipeline. The pipeline can be used to send messages or systolic data streams and can support "chatter" of intermediate results between adjacent PEs. The simplicity of the pipeline structure permits it to be implemented as a full 16-bit wide path.

The fact that the architecture is targeted to DSP algorithms means that its communication structure is greatly optimized. The architecture only implements a few very high speed communication paths in hardware, while it provides complete connectivity in software. The system common bus is used primarily for I/O, but also supports broadcasts and nonlocal point to point communications as may be required by various algorithms. It is used by the host to fork processes to the PEs and for data I/O.

9. AdEPar Simulation and Implementation Environment

The current difficulties in parallel DSP computing are on the software side, as opposed to hardware. The major obstacle to the widespread use of multiprocessor systems as well as parallel DSP systems has been the absence of adequate high level programming tools to automatically schedule the programs onto the multiprocessors and to perform the hardware debugging of parallel programs. The AdEPar hardware architecture was considered together with programmability, yielding an integrated system solution that combines extensive concurrency with simple programming. The AdEPar integrated architecture for DSP programming has broad capabilities in graphical design and analysis, data acquisition, signal generation and processing, digital filter design and implementation, real-time instruments, DSP code generation, and algorithm development and verification [15].

To have the highest flexibility for DSP programming, an advanced DSP environment based on a parallel DSP architecture must include all programming levels from assembly code to high-level abstractions to enable the user to use and exploit the system features efficiently and easily. The AdEPar Software Environment supports users with varying levels of sophistication. The environment has tools for design, simulation, implementation and debugging of parallel DSP programs. The algorithm designer with little knowledge (or interest) in the AdEPar hardware architecture or parallel program debugging can quickly and easily produce a simulation or a real-time program using blocks from the standard AdEPar library. A slightly more experienced user can define algorithms more suitable for code generation for the target architecture and can use the parallel hardware debugger. In Figure 14, a computer screen with three different AdEPar software modules is shown. The AdEPar software architecture is mainly based on these three Windows applications: AdEParGUI, SGenPlot and Parallel Hardware Debugger.


AdEPar Parallel Debugger
Files Run Memory Windows Help
PE 01
PE 02
Disassembly:
AdEPar DSP Simulation and Code Generation Environment v2.0
File Edit Simulate CodeGen Functions Graphics Debug Window Help
FlowGram: c:\adepar1\examples\demod.flo
FFT
Scope
Scope:0
FIR
Coefficient 0
OK
Cancel
SGenPlot

```
0000:CE06h RSXM
0001:C800h LDPK 00h
0002:CA03h LACK 03h
0003:6004h SACL 04h
```

```
0000:CE06h RSXM
0001:C800h LDPK 00h
0002:CA03h LACK 03h
BP→0003:6004h SACL 04h
```

Figure 14. A screen example of the AdEPar DSP environment based on the target architecture.

AdEParGUI: The five program components of AdEPar (Text Editor, Block Diagram Editor, ADF and LGDF code generator programs and LGDF Simulator) are built into one main graphics user interface (GUI) program module (AdEParGUI) running under Windows. The descriptions of ADF graphs for real time code generation can be prepared using the AdEPar built-in Text Editor. The generated C or assembly text files can also be opened by this editor to examine the generated code. The Block Diagram Editor is a visually programmed signal processing tool. By arranging and connecting icons, DSP and other algorithms can be programmed, adjusted, and easily tested. Each icon represents an input, output, display, or processing function. Blocks can handle multiple inputs and outputs, and can run on the host's processor (mainly a PC) or on a DSP processor on an add-in board (PE of AdEPar). A full library of functions is included, and the user can add new functions to the menu by writing blocks in standard C. The code generation software (ADF and LGDF CGEN programs) allows automatic generation of source code for implementation of DSP algorithms. The code generated may be either C source or native assembly code for the specific DSP chip. TMS320C25 DSP processor is currently supported. The resulting code is highly optimized for extremely efficient DSP applications. The AdEPar DSP algorithm development program, LGDF Simulator, enables the simulation of DSP algorithms, the generation of various signals, and complete analysis of signals.

Parallel Hardware Debugger: The environment has a powerful window-based hardware debugger that displays the states of the PEs in the system simultaneously. The debugger screen displays internal registers, data memory and provides a reverse assembly of program memory. Single step-run, run-to-break point operations are possible for PEs.

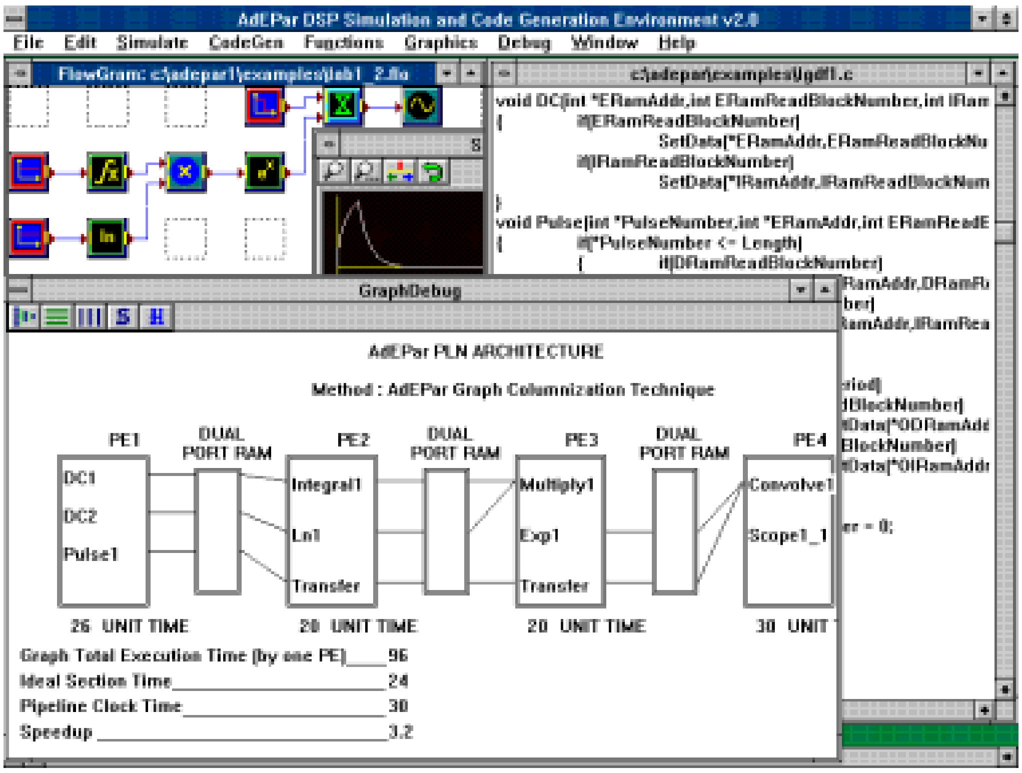
SGenPlot: This is a signal generator and plotter program used to generate and analyze time and frequency domain signals graphically. Any number of display windows, variable sizing and zooming of signals are allowed. SGenPlot has connections to other AdEPar tools (AdEParGUI, Parallel Hardware Debugger) and Texas Instruments' TMS320C25 Simulator.

10. Implementation of DSP Algorithms on AdEPar

In the AdEPar DSP environment for both simulation and code generation, an efficient algorithm, blocks columnization, that considers temporal and spatial concurrency and the target AdEPar architecture is developed for a wide range of directed acyclic graph structures. The blocks columnization algorithm maximizes the throughput of the system while meeting resource constraints and programmability of the system. It mainly applies chain partitioning which pipelines serial tasks onto a linear array of PEs to maximize throughput [29]. The basic idea of this algorithm is to divide a DSP algorithm into columns according to the blocks' precedence and their computational load. The AdEPar scheduler automatically schedules the blocks to be executed in the order of signal flow as determined by the topology of interconnections. This means that when a block is executed, all the blocks which are connected to its output have been given the opportunity to execute since the last execution of that block, and data are likely to be available in the input FIFO buffers. The simple heuristics and characteristics of different blocks are used to construct a list of all nodes approximately ordered according to their precedence. The blocks will be invoked in the order specified by this list.

The code generation process involves translating the graphical data flow graph into subgraphs, and generating assembly or C code for each partitioned subgraph. The code generator is essentially a text merging tool constructing the whole implementation programs by consecutively merging code sequences from the AdEPar library in an order which is determined by the block interconnections. In Figure 15, a computer screen with a simulation, load distribution and code generation example for a given graphical description of a DSP algorithm is shown.

The window GraphDebug shows the optimum load distribution of the algorithm onto four PEs. The number of PEs in the code generation is a user parameter from the menu of the main window. After the user simulates his or her real-time algorithm many times and finds good results using the LGDF Simulator in the AdEPar DSP Environment, he or she goes to load distribution phase in the code generation process. Depending on the PEs available in the system, and the speedup gained in the final load distribution produced by the scheduler, he or she determines the number of PEs. Actually, the scheduler can find the optimum load distribution for the given algorithm, but the program gives the flexibility to the user to change the number of PEs to obtain the desired performance for the available configuration. W_{total}, the total execution time of the graph by one PE, is calculated for the given graph. T, the ideal pipeline section time, is defined as $T = W_{total}/P$. The algorithm is based on performing a search to find the minimal T given P, the number of PEs. The upper bound of T, denoted as UB, is initialized to W_{total}, the computation time of the entire graph. The lower bound LB is set to W_{total}/P, the ideal section time, given P PEs. The scheduler traverses the graph from output to input, partitioning the graph into sections of pipelines. Nodes are scheduled onto a PE until the total computation cost of the nodes exceeds the section time T. Once a pipeline is filled, the scheduler proceeds to schedule the remaining nodes on the next pipeline section. At the end, the graph is partitioned into a number of pipeline sections.


AdEPar DSP Simulation and Code Generation Environment v2.0
File Edit Simulate CodeGen Functions Graphics Debug Window Help
FlowGram: c:\adepar1\examples\lab1_2.flo
c:\adepar\examples\lgdfl.c
GraphDebug
AdEPar PLN ARCHITECTURE
Method : AdEPar Graph Columnization Technique
PE1
DUAL PORT RAM
PE2
DUAL PORT RAM
PE3
DUAL PORT RAM
PE4
DC1
DC2
Pulse1
Integral1
Ln1
Transfer
Multiply1
Exp1
Transfer
Convolve1
Scope1_1
26 UNIT TIME
20 UNIT TIME
20 UNIT TIME
30 UNIT

```
void DC(int *ERamAddr,int ERamReadBlockNumber,int IRam
{               if(ERamReadBlockNumber)
                                SetData(*ERamAddr,ERamReadBlockNu
                if(IRamReadBlockNumber)
                                SetData(*IRamAddr,IRamReadBlockNum
}
void Pulse(int *PulseNumber,int *ERamAddr,int ERamReadE
{               if(*PulseNumber <= Length)
                {               if(DRamReadBlockNumber)
```

Graph Total Execution Time (by one PE)____96
Ideal Section Time____24
Pipeline Clock Time____30
Speedup____3.2

Figure 15. The simulation, automatic task partition onto 4 PEs, and C program generation of a DSP algorithm in the AdEPar DSP Environment.

In this load distribution example, the total execution time of the graph by one PE is 96 (unit time). For the 4-PE PLN architecture, the execution time of the graph becomes 30 and the speedup will be 3.2 compared with one PE. In PE2 and PE3, there are two Transfer blocks whose functions are to transfer data from their left to right neighbors. These two Transfer blocks convey the data produced by the Pulse1 block which is in PE1 to the Convolve1 block which is in PE4. For these Transfer blocks, the code is generated in the last code generation phase. If the user finds this load distribution satisfactory, he or she chooses the code generation process from the CodeGen menu. For each PE, one C file is generated. The generated C files are compiled by the commercial DSP cross C compiler. Finally, the files generated by the C compiler are loaded to the system using the AdEPar Parallel Hardware Debugger. In the PPLN configuration, the same code will be loaded to each PLN row.

11. Conclusion

The architecture given in this paper needs a simple interconnection network and less interprocessor communication and I/O operations. It is scalable and expandable so that application specific systems could easily be configured according to the needs. This architecture can be used for implementing various DSP algorithms. The architecture is an SSIMD or MIMD machine depending on the algorithms implemented and the programming methodologies. The concurrency in the scheduling algorithms of this parallel pipelined architecture is both temporal and spatial concurrency. This architecture was considered together with programmability, yielding a system solution that combines extensive concurrency with simple programming. High level programming tools are relatively easy to develop for this pipelined architecture.

The integrated AdEPar architecture has the following important features:

Hardware and software coupled practical architecture;

General architecture for implementing a wide range of DSP algorithms described by both ADF and LGDF graphs;

Temporal and spatial concurrency;

Scalability and expandability;

Simplicity to construct the architecture (IBM PC/AT bus based);

Easy hardware debugging support for parallel processing;

Easy to develop schedulers, high level programming tools and code generators; and

Easy to use even for undergraduate students.

References

[1] T. F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, 1992.

[2] A. Y. Zomaya (Ed.), *Parallel and Distributed Computing Handbook*, McGraw-Hill, New York, 1996.

[3] F. Kurugöllü, "A Parallel Signal Processing System and an Application", *Master Thesis* (in Turkish), İstanbul Technical University, Computer Engineering, February, 1993.

[4] H. Gümüşkaya, "A Parallel Computer Hardware and Software Architecture for Digital Signal Processing", *Ph. D. Dissertation*, İstanbul Technical University, Computer Engineering, June, 1995.

[5] F. Kurugöllü, A. E. Harmancı, H. Gümüşkaya, "TMS320C25 Yongaları ile Gerçekleştirilmiş Çift İşlemcili Paralel Sayısal İşaret İşleme Sistemi", *1. Sinyal İşleme ve Uygulamaları Kurultayı (SİU'93)*, Sayfa:198-202, 21-22 Nisan 1993, Boğaziçi Üniversitesi.

[6] H. Gümüşkaya, B. Örencik, A. Akman, "Blok Diyagram Tanımlamalı ve Asenkron Veri Akışı Tabanlı bir Sayısal İşaret İşleme Ortamı", *2. Sinyal İşleme ve Uygulamaları Kurultayı (SİU'94)*, Sayfa: 276-281, 8-9 Nisan 1994, Gökova, Muğla.

[7] H. Gümüşkaya, B. Örencik, "Sayısal İşaret İşleme için Paralel Mimariler", *3. Sinyal İşleme ve Uygulamaları Kurultayı (SİU'95)*, Cilt B, Sayfa: 291-296, 26-28 Nisan 1995, Kapadokya, Nevşehir.

[8] H. Gümüşkaya, B. Örencik, "Sayısal İşaret İşleme için bir Donanım ve Yazılım Ortamı", *4. Sinyal İşeme ve Uygulamaları Kurultayı (SİU'96)*, Sayfa: 337-342, 5-6 Nisan 1996, Kemer, Antalya.

[9] H. Gümüşkaya, B. Örencik, "ADF Grafları ile Tanımlı Gerçek-Zaman Süzgeçleri için Paralel Mimariler ve Otomatik Program Üretimi", *4. Sinyal İşleme ve Uygulamaları Kurultayı (SİU'96)*, Sayfa: 343-348, 5-6 Nisan 1996, Kemer, Antalya.

[10] H. Gümüşkaya, C. Z. Tan, B. Örencik, "Bir Paralel Bilgisayar Mimarisinde DSP Uygulamalarının Başarım Analizi", *5. Sinyal İşleme ve Uygulamaları Kurultayı (SİU'97)*, 1-3 Mayıs 1997, Sayfa: 667-672, Kuşadası, İzmir.

[11] H. Gümüşkaya, H. Palaz, F. Kurugöllü, B. Örencik, "Practical Scheduling of Real-Time Digital Filtering Algorithms onto Multiprocessors", Euromicro 94, September 5-8, 1994, Liverpool, England.

[12] H. Gümüşkaya, B. Örencik, F. Kurugöllü, H. Palaz, "Automatic Scheduling of Real-Time Digital Filtering Algorithms onto Processors", 5th International Conference on Signal Processing Applications and Technology, (ICSPAT 94), Dallas-Texas, USA, October, 1994.

[13] F. Kurugöllü, H. Palaz, H. Gümüşkaya, A. E. Harmancı, B. Örencik, "Advanced Educational Parallel DSP System Based on TMS320C25 Processors", Microprocessors and Microsystems, pp. 147-156, April, 1995.

[14] H. Gümüşkaya, B. Örencik, "Networks Analysis and Comparison of a Parallel Pipelined Architecture for DSP", International Symposium on Computer and Information Sciences X (ISCIS'10), October 30 - November 1, 1995, Efes, İzmir.

[15] H. Gümüşkaya, Örencik, "The Design of a Block Diagram Based Object-Oriented Graphical User Interface for a DSP Environment", The Journal of İstanbul Technical University, Vol. 49, pp. 441-457, 1996.

[16] H. Gümüşkaya, B. Örencik "AdEPar Integrated Simulation and Implementation Environment for DSP", Simulation, pp. 335-349, December, 1997.

[17] J. P. Brafman, J. Szczupack, and S. K. Mitra, An Approach to the Implementation of Digital Filters Using Microprocessors, IEEE Transactions on Acoustics Speech, and Signal Processing, Vol. ASSP-26, No. 5, August, 1978.

[18] J. Zeman, G. S. Moschytz, "Systematic Design and Programming of Signal Processors, Using Project Management Techniques", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-31, pp. 1536-1549, December, 1983.

[19] H. Kasahara, S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", IEEE Trans. on Computers, Vol. C-33, No. 11, pp. 1023-1029, November, 1984.

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

[21] T. P. Barnwell, V. K. Madisetti, S. J. A. McGrath, "The Georgia Tech Digital Signal Multiprocessor", IEEE Trans. on Signal Processing, Vol. 41, July, pp. 2471-2487, 1993.

[22] P. R. Gelabert, T. P. Barnwell, "Optimal Automatic Periodic Multiprocessor Scheduler for Fully Specified Flow Graphs", IEEE Transactions on Signal Processing, Vol. 41, No. 2, pp. 858-888, February, 1993.

[23] R. G. Babb, "Parallel Processing with Large-Grain Data Flow Techniques", IEEE Computer, Vol. 17, No. 7, pp. 55-61, July, 1984.

[24] P. R. Cappello, K. Steiglitz, "Completely Pipelined Architectures for Digital Signal Processing", IEEE Transactions on Acoustics Speech, and Signal Processing, Vol. ASSP-31, August, 1983.

[25] S. Y. Kung, "Why Systolic Architectures?", IEEE Computer, pp. 37-46, Jan, 1982.

[26] J. Tatemura, H. Koike, H. Tanaka, "HyperDEBU: a Multiwindow Debugger for Parallel Logic Programs", Proc. of the IFIP, Workshop on Programming Environments for Parallel Computing, pp. 87-105, 1992.

[27] C. S. Burrus, "Block Implementation of Digital Filters", IEEE Transactions on Circuit Theory, Vol. CT-18, No. 6, pp 697-701, November, 1971.

[28] W. Sung, S. K. Mitra, B. Jeren, "Multiprocessing Implementation of Digital Filtering Algorithms Using a Parallel Block Processing Method", IEEE Transactions on Parallel and Distributed Systems, Vol 3, No. 1, January, pp. 110-120, 1992.

[29] P. D. Hoang, J. M. Rabaey, "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput", IEEE Transactions on Signal Processing, Vol. 41, No. 6, pp. 2225-2235, June, 1993.