

# A Reconfigurable Data-driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms

Alfred K.W. Yeung and Jan M. Rabaey

Dept. of Electrical Engineering and Computer Sciences  
University of California, Berkeley,  
Berkeley, CA 94720.

## Abstract

*A data-driven multiprocessor architecture for the rapid prototyping of complex DSP algorithms, based on direct execution of data-flow graphs, is presented. High computation bandwidth is achieved by exploiting the fine-grain parallelism inherent in the target algorithms using simple processing elements called nanoprocessors interconnected by a configurable static communication network. The use of distributed control and the data-driven execution approach result in a highly scalable and modular architecture. A prototype chip, which is currently being designed, contains 64 nanoprocessors, 1KB of memory in four banks and eight 16b I/O ports, and provides 3.2 GOPS peak when running at 50 MHz. The benchmark results based on a variety of DSP algorithms in video processing, digital communication, digital filtering and speech recognition confirm the performance, efficiency and generality of the architecture.*

## 1. Introduction

During the development of real-time, complex and high performance DSP systems, hardware prototyping plays an important role in shortening design time and in improving the design quality. A prototype allows the designer to quickly explore a large design space, to fine-tune algorithm parameters and to verify and evaluate the performance of the algorithms under realistic conditions, e.g., to experiment with a video compression algorithm while transmitting over a noisy communication channel, all in real-time.

Given today's technology constraints, a single-chip solution cannot satisfy the computational requirements of complex algorithms. The traditional approach of putting together a system using existing chips requires dedicated interface design and depends on the availability of chips with the desired functionality. Moreover, automatic compilation is difficult for such a heterogeneous hardware platform.

A multiprocessor approach using general purpose DSP processors, though very flexible, does not offer a cost-effective solution. The Von Neumann architecture, which invests heavily into control to enable time-multiplexing of operations on a few execution units, are not efficient (or are too complex) for exploiting fine grain parallelism in high throughput algorithms, as they offer limited opportunity for hardware sharing.

Systolic or wavefront array processors achieve high performance using regular, fine grain, locally interconnected processing elements [1]. However, the technique is only applicable to a certain class of algorithms [2] and specially designed I/O units are often needed to provide high I/O bandwidth to sustain the high throughput [3].

Field-programmable gate-arrays (FPGA), e.g., Xilinx [4], are extremely flexible by providing programmability down to the gate level, but their bit-oriented architectures are both inefficient in terms of area and are too slow for implementing datapath intensive architectures commonly used for high throughput DSP algorithms.

PADDI, a reconfigurable multiprocessor architecture specially developed for the prototyping of high speed datapaths, demonstrated the efficiency of exploiting fine grain parallelism using simple execution units [5][6]. However, this architecture may run into instruction bandwidth problems and the use of a global controller limits the scalability of the architecture.

Another interesting approach to exploit fine grain parallelism using simple processing elements is the Phillips video signal processor (VSP) architecture [7]. Based on the cyclo-static scheduling technique, the compiler can achieve optimality with respect to several criteria, including maximum throughput and maximum processor utilization. And while the architecture allows multiprocessing with very little overhead, data dependent operations cannot be handled in general.

In this paper, a flexible and scalable multiprocessor architecture geared towards rapid prototyping of high sampling rate, high throughput DSP algorithms with fine grain parallelism is described. An overview of the architecture is

given in Section 3. The architecture and design of the processing element and interconnection network are discussed next. The chip and system architecture are described in Section 7. The scalability of the architecture is elaborated in Section 8. Benchmark results are presented in Section 10, followed by the conclusions.

## 2. DSP Architecture Requirements

The target architecture must be able to provide the high computation throughput required by high performance DSP algorithms. Computational requirements in the order of GOPS are common in both video [8] and speech recognition [9] applications. In addition, the architecture must provide high memory and I/O bandwidth in order to sustain the high computation throughput. In a typical video processing application with 1K by 1K color pixels, the minimum memory bandwidth requirement is in the order of 200 MByte/sec operating at 60 Hz frame rate.

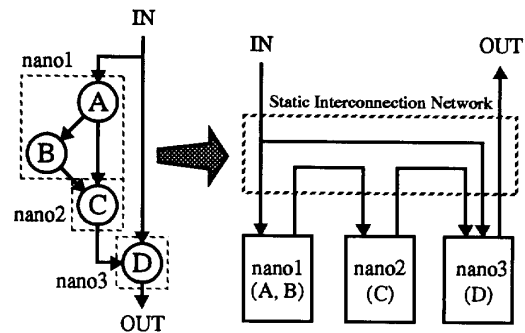
In addition, the architecture must be generic and flexible enough to handle a wide range of algorithms. Heavy emphasis is put on ease of programming, compilation and development of synthesis tools in order to achieve rapid prototyping.

Equally important is the scalability and modularity of the architecture. The architecture must be easily extended to tackle more and more complex problems and to keep pace with the rapid advances in technology without radical redesign. Field-programmability is desired to reduce turn-around time and cut costs.

## 3. A Data-driven DSP Architecture

In real-time, high throughput and high sampling rate DSP algorithms, operations in the kernels (inner loops) often need to be processed in a small number of clock cycles, implying limited opportunity for hardware sharing [10]. Researchers have found that an efficient way to exploit the fine grain parallelism in this type of algorithms is to use a network of simple processing elements, whose connection pattern closely matches the data flow pattern of the algorithm [6][11].

Moreover, very simple controllers are often sufficient because of the dedicated nature of the processing elements. This implies that in the data-flow graph representing the target algorithms, the majority of the nodes perform useful data processing and a few nodes are required for control or data routing. If the data-flow graphs of such algorithms are implemented by assigning a processing element to a node or a cluster of a small number of nodes, the resulting architecture would be efficient in the sense that the number of overhead processing elements is small. An extreme case is a systolic algorithm whose data-flow graph has no overhead nodes.

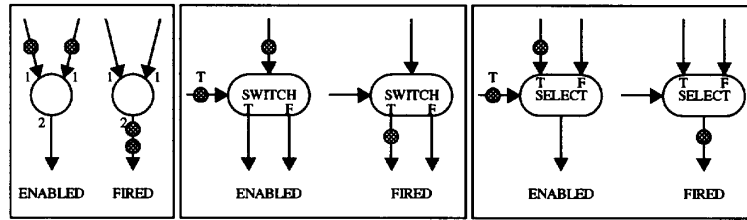


**Fig. 1. Direct execution of data-flow graph using statically interconnected Nanoprocessors.**

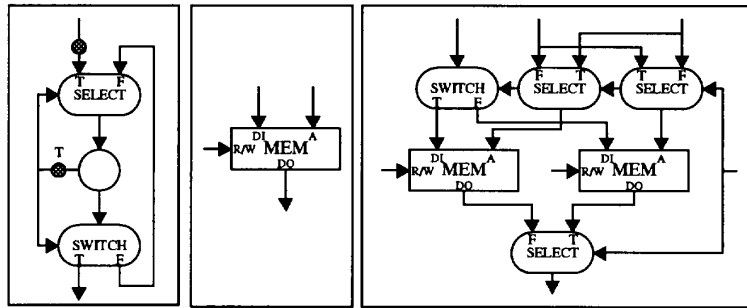
This motivated us to investigate a multiprocessor architecture which directly executes a data-flow graph by mapping the graph onto simple processing elements, called *nanoprocessors*, connected by a static interconnection network. As illustrated in Fig. 1, a nanoprocessor is simply a hardware realization of a node or an encapsulation of several nodes. Like their data-flow counterparts, nanoprocessors communicate with each other via static communication channels (arcs in the graph) using streams. Execution of an operation (the firing of a node) by a nanoprocessor is data-driven, i.e., execution depends on the availability of the required operands. In essence, the architecture is very much like a message-passing multicomputer system, which is considered to be the most scalable multicomputer system of all [12], except that very simple computers and static communication are used.

It is widely accepted that a data-flow graph is a natural, or perhaps the best, representation for signal processing algorithms [13] because it exposes all parallelism in the algorithms. Moreover, real signals are naturally represented by the data streams implied by the data-flow model. Since our architecture directly supports the semantics of data-flow, the task of developing an efficient compiler for DSP algorithms for our architecture, which is crucial for rapid prototyping, is greatly simplified.

A group of researchers at the University of Massachusetts at Amherst has taken a similar approach and designed a data-driven array for arbitrary algorithms [14]. However the local interconnection between processing elements result in low processor utilization and the long communication latency caused by routing data successively through processing elements may hurt the performance for iterative algorithms.



(a) Examples of Data-flow Operators.



(b) While-loop. (c) Memory Node. (d) Swapping Memory Banks.

Fig. 2. Static Data-flow Model.

#### 4. Static Data-flow

In this section, the data-flow model used for describing DSP algorithms is explained. Among the many variations of data-flow models [15], we adopt Dennis' static data-flow model [16]. A data-flow program is described by a directed graph where the nodes denote operations and the arcs denote data dependencies. Dynamic behavior caused by data-dependent iterations and conditionals is handled by the SWITCH and SELECT operators. Fig. 2a show examples of the basic data-flow operators. A data-flow graph implementing a while-loop is shown in Fig. 2b. (The number next to the arc denotes the number of tokens consumed or produced on each firing and is omitted if it equals one.)

The arcs going in and out of the nodes represent physical connections called *channels* between the nanoprocessors. To make the implementation of channels practical, the First-in-first-out (FIFO) schema is enforced on the arcs, i.e., the arcs are simply FIFO queues of bounded capacity.

Any algorithm can be described by using only the basic data-flow nodes shown in Fig. 2a [16]. Fig. 3a shows the *minimal nanoprocessor* which can implement these basic data-flow nodes. Therefore by interconnecting multiple minimal nanoprocessors, any algorithm can be implemented. However, the implementation is most efficient for algorithms which are computation intensive and require simple control. These are precisely the characteristics of our target applications.

A *pure* data-flow node does not have internal state. By allowing internal states, a node can be extended to represent an encapsulation of several nodes. A cluster of nodes simply denotes a number of primitive operations executed in a specific order. Thus data-flow nodes can be implemented either spatially by a network of nanoprocessors or temporally by executing a sequence of instructions on a single nanoprocessor. In the later case, the nanoprocessor functions like a traditional Von Neumann machine. To support the time multiplexing of several nodes onto a single nanoprocessor, a small program store is added to the minimal nanoprocessor as shown in Fig. 3b.

By allowing internal states in data-flow nodes, memory can be easily incorporated into the data-flow model (Fig. 2c), preserving a consistent programming paradigm across the architecture. A memory node is similar to other data-flow nodes except that it has a very large number of internal states. It consumes an address token and a control token specifying either a read or write operation. For a read operation, it generates an output token containing data stored at the location pointed at by the address token. For a write operation, it consumes an additional input data token and simply updates its internal states by storing the data. Memory nodes, together with the SWITCH and SELECT operators, can create many powerful shared memory structures. An example of swapping memory banks often used in video processing to buffer successive pixel frames is shown in Fig. 2d.

Since data-flow nodes communicate with each other using streams, all data structures have to be represented as

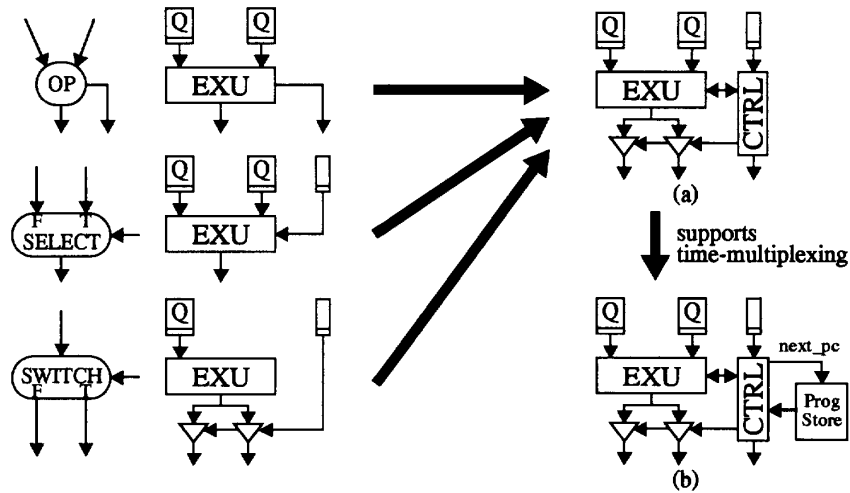


Fig. 3. Minimal Nanoprocessor.

such. Some examples of data structures commonly used in signal processing are the infinite stream (e.g., continuous speech) and the 2-dimensional matrix (e.g., a block of image). More complex data structures can be represented by multiple streams. Stream manipulations such as merging or splitting can be handled by simple data-flow nodes like SELECT or SWITCH, e.g., to partition a frame of pixels into multiple blocks for parallel processing.

## 5. Processing Element: Nanoprocessor

### 5.1 Simple Nanoprocessor

In multiprocessing, the choice of the complexity of the processing elements has a big impact on the efficiency of the architecture. If the processing element is overly simple, more of them are needed to achieve a certain throughput and more pressure is put on the interconnection network design. On the other hand, if the processing element is too complex, a good portion of it may be underutilized, wasting valuable silicon area and often resulting in a slower processor because of the overhead. Obviously, the ideal processing element depends heavily on the particular application at hand. One compromising solution is to use heterogeneous processing elements. However, the heterogeneous resources are likely to complicate the compilation process and further jeopardize the chance of developing an efficient compiler.

Due to the limited opportunity for hardware sharing and high computation requirement, a nanoprocessor is often dedicated to the processing of only a small number of operations. It will be inefficient in terms of area if complex processing element is used. We believe that more efficient use of hardware and flexibility are achieved by

defining a simple homogeneous nanoprocessor architecture. More complex functions can then be performed by configuring these basic units to work in parallel. In addition, by keeping the nanoprocessor simple, fast cycle time can be achieved. Finally, the regularity also simplifies communication network design and improves layout density, resulting in a high integration level (64 nanoprocessors on a die).

As an example, if a hardwired multiplier is included in each processing element, a multiplication operation can be implemented quickly by a single processing element. However, many processing elements are still needed to handle the other miscellaneous operations such as address generation, data routing and simple arithmetic operations like add/subtract, shift, etc. Moreover, in many applications such as filtering and low-level image processing, the coefficients are often set up in such a way that a simple execution unit with support for add-shift operation suffices. In situations where the execution of the multiplication operation is not time-critical, it can be performed by a simple execution unit in an iterative fashion. If higher throughput is needed, several processing elements can be linked together to implement a pipelined multiplier. Therefore more often than not, a built-in hardwired multiplier is not absolutely required.

### 5.2 Data-driven Distributed Control

The data-flow principle implies distributed control, i.e., each nanoprocessor is equipped with its own local controller. Moreover, execution of a program in one nanoprocessor has no *side-effect* on the others. By eliminating global control and side-effect, the scalability and modularity of the architecture is ensured. The local controller relieves

instruction bandwidth bottleneck and allows the use of long and fully decoded instructions, resulting in a fast cycle time and a powerful instruction set. Another advantage is that zero-cycle branch latency as well as multi-way branching can be achieved with very little speed penalty.

Nanoprocessors communicate with each other explicitly using data and control streams. The data-driven principle of execution gracefully synchronizes a large number of nanoprocessors typically needed for the target algorithms. The data-driven handshake protocol provides a uniform interface between *all* elements of the architecture and improves the modularity of the architecture.

### 5.3 Nanoprocessor Architecture

Fig. 4 shows the block diagram of a nanoprocessor: it contains an 8-word by 50-bit instruction store, a 16-bit ALU which performs arithmetic and logic operations with special support for a modified Booth multiplication step, a 4-bit shifter and one 4-word data buffer (DQ) for each of the three inputs. Conditional execution is provided to speed up the MAXIMUM and MINIMUM operations often used in signal processing. All instructions take one cycle to complete.

The data buffer can be configured as a simple data queue or as a small random-access register file. Data at the head of the queue can be held for reuse. The registers are scannable so constants or filter coefficients can be loaded at start-up.

The local controller keeps a 3-bit program counter. A next program counter field (NPC) is available in the instruction word. Bits of the NPC can be modified depending on the status of the ALU or the token at the head of the control queue to implement multi-way branch with zero-cycle latency. The nanoprocessor is stalled if any of the required operands (data or control tokens) are not available or if any of the output channels are blocked. Since the nanoprocessor is used to emulate dedicated hardware

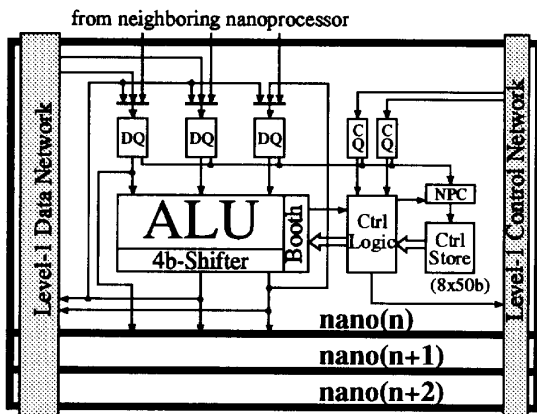


Fig. 4. Nanoprocessor block diagram.

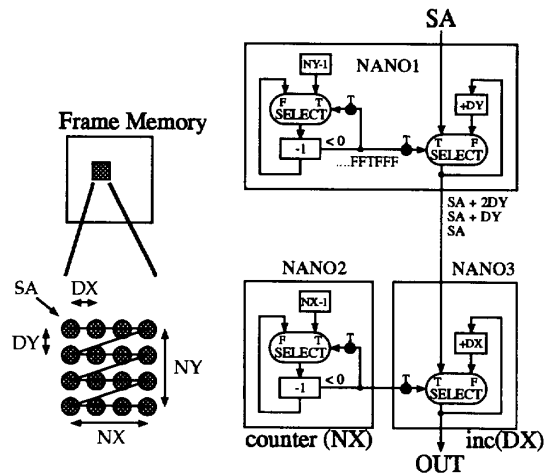


Fig. 5. 2-dimensional Space Address Generator using 3 Nanoprocessors.

(implementing just a few data-flow nodes), the control flow is usually very simple. Hence, the small instruction store rarely imposes any serious constraints.

In order to implement the SWITCH node, two output connections to the communication network are available. To facilitate neighboring communication and reduce network bandwidth demand, direct connections between neighbors are implemented without access to the Level-1 network.

As shown in Fig. 4, the inputs of two data buffers are connected to either the communication network or its immediate neighbor, while the third buffer connects only to the neighbor. This third buffer is specifically provided to allow for a pipelined implementation of a 16b x 16b multiplier using a single nanoprocessor cluster consisting of 4 nanoprocessors.

Similarly, two input control channels, each buffered by a control queue (CQ) and one output channel are available for the control streams.

Fig. 5 is a simple example that illustrates how several nanoprocessors can be connected to implement more complex function. A two-dimensional space address generator [17] commonly used in video applications is implemented using three nanoprocessors. Counter(N) generates an infinite control stream whose tokens are always zero with the exception of every N tokens. Inc(D) simply copies the input to output if the control token is one and increments the previous output by D otherwise. To provide a throughput of one address per cycle, the Counter (NANO2) and Inc (NANO3) for the x-direction have to operate every cycle but the corresponding blocks for the y-direction can be run at a much lower rate and therefore they can be implemented using a single nanoprocessor (NANO1).

## 6. A Static Hierarchical Interconnection Network

In an effort to reduce area overhead imposed by the interconnection networks, many array architectures provide rather restrictive interconnection networks [1][14]. This limits the scope of the application, (e.g., to systolic algorithms only,) and may degrade the performance for algorithms with recursions. Moreover, the lack of a flexible network further increases the burden on the compiler to generate a good processor placement scheme. To overcome the above inefficiencies, our architecture provides a flexible communication network to match the communication patterns of a wide range of algorithms and to maximize nanoprocessor usage.

Motivated by the static interconnection used in connecting nodes in data-flow graphs, the architecture provides only static point-to-point communications between nanoprocessors. This greatly simplifies network design and eliminates the needs for the sophisticated network routing hardware commonly required in general multi-computer systems [12]. The speed of the network is also improved because the large transistors typically used to implement the switches in the network, do not have to be switched every cycle.

The lack of dynamic interconnection does not impose a serious restriction. First, because nanoprocessors are usually dedicated to the processing of a few operations, channels are likely to be saturated with data. Therefore the opportunity for channel sharing is limited. A study by Chen [6] on the interconnection patterns of a set of high throughput, high sampling rate DSP algorithms confirms the observation and shows that 95% of the communication channels are either not shared or shared by fewer than two clients. Second, dynamic interconnection patterns, if needed, can be emulated by programming the nanoprocessors to implement the SWITCH and SELECT nodes.

To trade off flexibility in interconnection with area cost and cycle time, and to exploit the locality of communication, a 2-level hierarchical network is used. As shown in the chip block diagram in Fig. 6, the level-1 network is used for interconnecting 4 nanoprocessors within a nanoprocessor cluster and is laid out directly on top of the cluster as data-path feed-throughs to save area.

Similar to the interconnect matrix design in [18], a semi-crossbar with switchable buses is used to implement the level-2 network (Fig. 7). Again, by taking advantage of the fact that local communications are far more frequent than distant communications, programmable switches can be opened to break up a bus (track) in the crossbar into a few buses of shorter lengths, thus increasing the effective number of buses in the network without area penalty. To prevent the delay in the communication network from

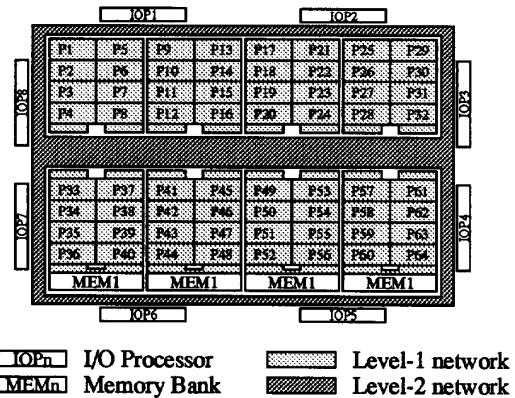


Fig. 6. Block diagram of the prototype chip.

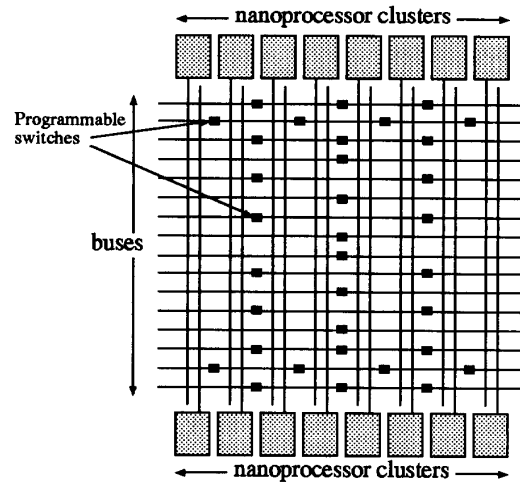


Fig. 7. Basic Structure of the Semi-crossbar in Level-2 Interconnection Network.

dominating the cycle time, a channel can go through no more than one closed switch in the Level-2 network.

Two lines (*send* and *busy*) are associated with each channel to implement the handshake protocol. Data broadcasting is supported to reduce demand on communication channels and to facilitate distributing data when exploiting spatial concurrency. The mechanism is implemented efficiently using wired-logic for the *busy* signal.

## 7. Chip and System Architecture

A block diagram of the prototype chip is shown in Fig. 6. Each chip contains 16 nanoprocessor clusters organized in 2 banks and four on-chip memory banks (4 x 128 words x 16 bits) interconnected via a 2-level configurable network. Eight 16-bit I/O ports, each controlled by a special-

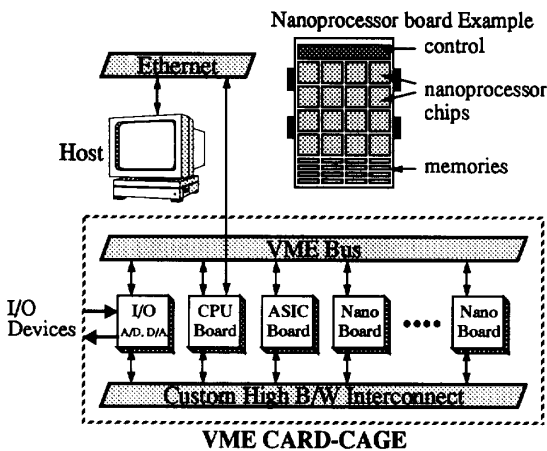


Fig. 8. Prototyping System.

ized I/O nanoprocessor called IOP, provide high I/O bandwidth needed for sustaining the high throughput. Two ports can be linked to provide 32-bit I/O if necessary.

The primary function of the IOP is to control off-chip communication with other nanoprocessor chips or memory modules. It handles handshake protocols, memory strobe generation, memory sharing and arbitration. It has built-in multiplexers and the necessary glue logic and control to implement shared memory structures as shown in Fig. 2d.

A serial port handles the downloading of nanoprocessor instructions, network configuration and data buffer initialization. Selective internal scan chains and a boundary scan chain are provided for system testing and run-time debugging support. Nanoprocessors can be programmed to collect relevant system statistics such as how often an arithmetic overflow occurs at a certain node of the algorithm and the results can be obtained by halting the system and scanning out the relevant information.

A prototype chip of the architecture, which is being designed in a 1.2um CMOS technology, can provide 3.2 GOPS, 400 MB/sec on-chip memory bandwidth and 800 MB/sec I/O bandwidth at the estimated clock frequency of 50 MHz. The frequency estimate is based on simulations of the critical paths.

Fig. 8 shows the system architecture of the DSP prototyping platform. The major components of the system are the host system, the I/O unit, the CPU board and the nanoprocessor boards. The host is a workstation that functions as a software development station and as a large auxiliary data storage. It is connected to the CPU control board in the VME cardcage via the ethernet. The CPU board is responsible for configuring and monitoring the other system boards via the VME bus. The I/O unit provides inter-

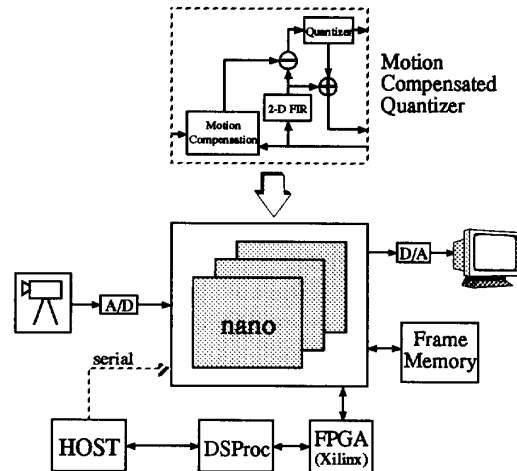


Fig. 9. Example of Prototype DSP system.

faces to real-time input devices such as cameras and sensors, and output devices such as speakers and monitors.

The bulk of the computation is performed by the nanoprocessor boards. An example of a nanoprocessor board contains 16 mesh-connected nanoprocessor chips and several memory modules. Nanoprocessor chips on the periphery of the array are connected to either the memory modules or the nanoprocessor chips on another boards.

An example of a DSP system prototyped by the nanoprocessor boards is depicted in Fig. 9. One nanoprocessor board can implement the computation intensive motion compensated quantizer block which is used in a variety of video compression algorithms. Another nanoprocessor board may be needed to implement the frame memory depending on its size.

## 8. Scalable Architecture

The programmability provided by the proposed architecture is very different from that of a conventional Von Neumann computer in which programmability is obtained by time sharing a general-purpose execution unit according to an instruction sequence. In this architecture, programmability is primarily effected by configuring an interconnection network connecting many nanoprocessors. While the former is very efficient in the use of silicon area, its throughput performance is poor. Our architecture obtains high performance and flexibility at the expense of lower processor utilization and area overhead in implementing the communication network. More importantly, by adopting the data-driven principle of execution and a distributed control strategy, the proposed architecture achieves high scalability and modularity. Many nanopro-

processors can be easily incorporated into the system to solve more complex problems.

In the days of large and very costly processors, processor utilization was a prime measure of computational efficiency. Today's ULSI technology has drastically reduced the processor cost by providing millions of transistors on a single die. The Law of Diminishing Returns applies to computer systems in that beyond a certain point, the cost of trying to eliminate the waste is higher than the cost of the waste itself. We believe it is more important to design a scalable architecture which can easily be extended to allow more resources to work together rather than to improve the utilization of every kind of resources in the system.

New technologies will continue to provide more resources at a lower cost. Advanced IC processes provide more routing layers and small metal pitches to enable design of dense interconnection network. Wafer-scale integration, multi-chip modules and new packaging technologies increase I/O bandwidth, improve speed and effectively bring chips much closer together to facilitate multiprocessing. The high scalability put the architecture in a strong position to keep pace with and take advantage of the rapid advances in technology.

## 9. CAD Support

Since our goal is rapid prototyping, special attention has been paid to develop CAD tools which will provide an automated compilation path from a high level data-flow language, e.g. Silage [19], to the nanoprocessor hardware platform.

The software environment being developed is an extension of the HYPER synthesis system [10], which targets semi-custom architectures for high performance DSP algorithms. Its tool suite includes behavioral simulation, estimation, transformations, resource allocation, partitioning, assignment and code generation. The user interacts with these tools via an X-based graphic user interface. A typical compilation session which usually involves several iterations of estimation, partitioning, assignment and various transformations is depicted in Fig. 10.

## 10. Benchmark Results

A benchmark set consisting of algorithms in the areas of video processing, speech recognition, digital communication and digital filtering was analyzed to justify the architectural choices and verify the performance and generality of the architecture.

Fig. 11 shows the implementation of a 2-dimensional 8x8 Discrete Cosine Transform (DCT) at a 25 MHz sampling rate using 50 nanoprocessors and two on-chip memory banks used as transpose buffers. The algorithm is

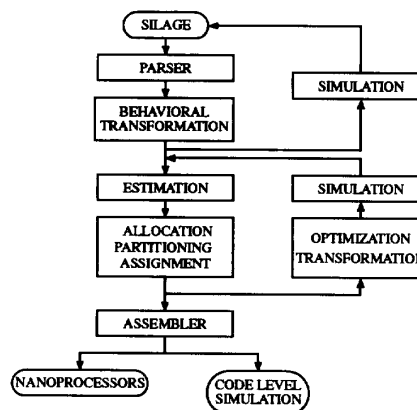


Fig. 10. Nanoprocessor Compilation Session.

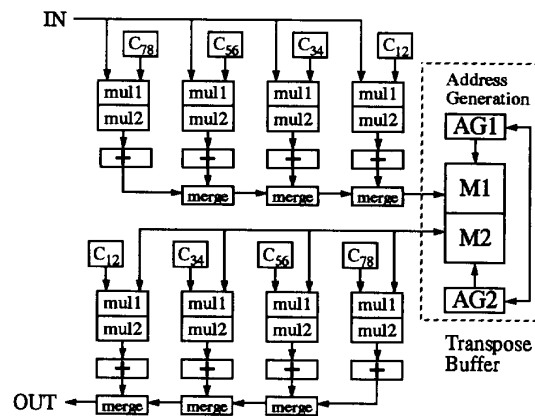
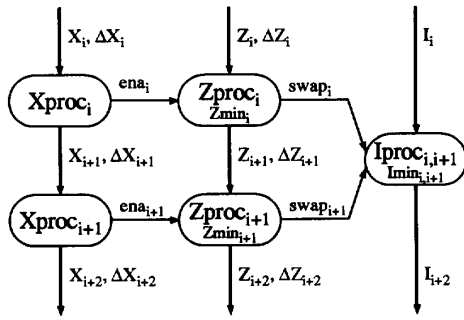


Fig. 11. 2-D 8x8 DCT at 25 MHz sampling rate using 50 Nanoprocessors.

decomposed into two one-dimensional DCT's and a transpose operation. Multiplication is implemented in a pipeline fashion using two nanoprocessors. The transform coefficients stored in the data buffers of the nanoprocessors are streamed into the multiplication nanoprocessors together with the input pixels. The address generation unit (AG2), which is very similar to the 2-dimensional space address generator in Fig. 5, produces the desired address sequence for the transpose operation.

As a comparison, a general purpose DSP processor such as the TMS320C25 can only achieve a sampling rate of 250 KHz for the 2-dimensional 8x8 DCT with optimized code [20].

Another example is the Hidden Surface Processor used in 3-dimensional computer graphics [21]. The processor eliminates hidden portions of surfaces using a Z-buffer algorithm on a raster scan-line basis. Each pixel processor



**Fig. 12. Data-flow of Hidden Surface Processor.**

handles a single pixel and  $N$  processors connected in a linear array are needed to process an  $N$ -pixel scan-line. Data structures of line segments are fed into one end of the array and processed systolically. The segment data structure consists of:

- $X$  starting x-coordinate of the line segment.
- $\Delta X$  length of the line segment.
- $Z$  z-coordinate at the starting x-coordinate.
- $\Delta Z$  tilt of the line segment.
- $I$  intensity of the line segment.

Fig. 12 shows the implementation of 2 pixel processors using 5 nanoproductors. (The Iproc is shared between 2 pixel processors.)  $X$  is decremented as it propagates through the Xproc's until it becomes zero. When  $X_i$  becomes zero, Xproc<sub>*i*</sub> is at the beginning of the line segment. The subsequent Xproc's then decrement  $\Delta X$  instead. When  $\Delta X$  also becomes zero, the end of the line segment is reached. Xproc<sub>*i*</sub> sends an enable flag to Zproc<sub>*i*</sub> if it is within the line segment, i.e., when  $X_i$  is zero and  $\Delta X_i$  is not.

Upon receiving the enable flag, the Zproc updates the locally stored  $Zmin$  by comparing the incoming  $Z$  with  $Zmin$ . A swap flag is sent to the Iproc if  $Zmin$  has been updated. It also computes the new z-coordinate of the line segment at the next pixel by summing  $Z$  and  $\Delta Z$ . The Iproc simply updated  $Imin$  with the incoming  $I$  if the swap flag is on.

After the processing of the line segments is finished, a special *shift-out* data structure is sent which will reset  $Zmin$  in the Zproc's and cause the Iproc's to scan out the stored  $Imin$ 's so that processing of the next scan-line can begin.

This implementation can process one line segment data structure every 4 clocks and that translates into 12.5 millions line segments per second. 10 nanoproductor chips are needed to support a 256-pixel scan-line. Moreover, the throughput can be doubled to 25 millions segments per second by doubling the number of nanoproductors used.

Table 1 lists the number of core operations (not counting overhead) per data sample, the performance achieved and the amount of on-chip hardware used for the benchmark algorithms. All operations are 16 bit unless specified otherwise. Since most general purpose DSP processors are equipped with parallel multipliers, a multiplication is counted as only one operation in the throughput calculation to allow fair comparison although 4 nanoproductor operations are actually executed.

Most algorithms require irregular communications between nanoproductors. Algorithms 1, 2 and 3 demonstrate how algorithms with recursive bottlenecks can sometimes be transformed to allow more parallelism (more pipeline stages). The results show good correlation between performance gain and hardware cost. The Viterbi decoder in 4 contains a tight recursive loop which limits performance. The normalized throughput per nanoproductor

**Table 1: Benchmark Results.**

#	Algorithms	Core Operations (per sample)	Performance (Sampling-Rate)	Throughput	Hardware
1	2nd-order IIR filter [22] (original)	5 mult, 4 add	8.3 MHz	75 MOPS	6 nano
2	2nd-order IIR filter [22] (pipeline and multiplexed)	10 mult, 11 add	25 MHz	525 MOPS	27 nano
3	2nd-order IIR filter [22] (pipeline)	10 mult, 11 add	50 MHz	1050 MOPS	54 nano
4	Viterbi ACS unit [23]	15 comp & select	5.6 MHz (9 Mb/s)	168 MOPS	10 nano
5	2-dim space address generator [17]	2 add, 1 comp	50 MHz	150 MOPS	3 nano
6	3x3 sorting filter	15 comp & select	50 MHz	1500 MOPS	30 nano
7	Video Matrix conversion (8b pixel)	4 mult, 6 add	50 MHz	500 MOPS	14 nano
8	2-dim 8x8 DCT (8b pixel, multiplexed)	16 mult-add	25 MHz	800 MOPS	50 nano, 2 mem
9	2-dim 8x8 DCT (8b pixel, non-multiplexed)	16 mult-add	50 MHz	1600 MOPS	82 nano, 2 mem
10	Viterbi Word Processor [9]	5 add, 3 comp, 3 select, 3 mem rd & wr	3 state updates at 25 MHz	275 MOPS	13 nano, 3 mem
11	Motion Vector Estimation (8x8 blk, 16x16 search blk)	64 sub, comp & add	22.2 MHz	4267 MOPS	4 nano-chips
12	Hidden Surface Pixel Processor [21] (256-pixel line)	3 comp, 3 add, 4 select	12.5M segments/s	32 GOPS	10 nano-chips

sor chip averaged over the benchmark set is 1542 MOPS (harmonic mean).

## 11. Conclusion

A data-driven multiprocessor architecture for rapid prototyping of complex DSP algorithms, based on direct execution of data-flow graphs, is presented. High computation bandwidth is achieved by exploiting fine-grain parallelism inherent in the target algorithms using simple nanoprocessors. The hierarchical network provides flexibility to accommodate a wide range of algorithms. The distributed control strategy and the data-driven principle of execution result in a highly scalable and modular architecture that can be easily extended to solve more complex problems. The simplicity and homogeneity of the architecture, the consistent programming paradigm and the direct architectural supports for the execution of data-flow graphs facilitate development of compiler and synthesis tools needed for rapid prototyping.

A prototype chip consisting of 64 nanoprocessors, 1KB of data memory and eight 16-bit I/O ports is currently being designed and is expected to tape out early next year. Circuit simulations of critical paths showed possible clock frequency in excess of 50 MHz.

A software environment based on HYPER, which aims at providing an automatic compilation process from algorithm specification to hardware, is currently being developed as a parallel research project.

The benchmark results showed that the architecture is flexible and can satisfy the high throughput requirement of a large variety of algorithms. Complex algorithms such as the Hidden Surface Processor and the motion compensated quantizer can be implemented using a small number of nanoprocessor boards.

## 12. Acknowledgments

This research was funded by DARPA.

## 13. References

- [1] S. Y. Kung, "On supercomputing with systolic/wavefront array processors," *Proceedings of the IEEE*, pp. 39-46, 1984.
- [2] S. K. Rao, *Regular iterative algorithms and their implementation on processor arrays*. Ph.D. thesis, Stanford University, 1985.
- [3] H. Volkens et al., "Cache Memory Design for the Data Transport to Array Processors". ICASSP, 1990.
- [4] R. Freeman. "User-programmable Gate Arrays". *IEEE Spectrum*, Dec. 1988.
- [5] D. Chen and J. Rabaey. "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Real-time Data Paths", *ISSCC*, 1992.
- [6] D. Chen, "Programmable Arithmetic Devices for High Speed Digital Signal Processing", Ph.D. thesis, University of California at Berkeley, May, 1992.
- [7] R. J. Sluyter et al., "A Programmable Video Signal Processor", *ICASSP*, 1989, Glasgow, U. K.
- [8] R. K. Jurgen, "The Challenges of Digital HDTV", *IEEE Spectrum*, April, 1991.
- [9] J. Rabaey et al., "A Large Vocabulary Real-Time Continuous Speech Recognition System", *VLSI Signal Processing III*, pp. 62-74, *IEEE Press*, Nov., 1989.
- [10] Rabaey, C. Chu, P. Hoang, M. Potkonjak, "Fast Prototyping of Data Path Intensive Architecture", *IEEE Design and Test*, pp.40-51, 1991.
- [11] W. Geurts, F. Catthoor, "DSP Applications suited for Lowly Multiplexed Architectures", *ASICS Open Workshop on Synthesis Techniques for (Lowly) Multiplexed Datapaths*, Aug. 1990.
- [12] S. Borkar et al., "iWarp: An Integrated Solution to High-Speed Parallel Computing", *Proceedings Supercomputing*, Nov 1988.
- [13] E. A. Lee, "Consistency in Dataflow Graphs", *IEEE Transactions on Parallel and Distributed Systems*, April, 1991.
- [14] I. Koren, B. Mendelson, "A Data-driven VLSI Array for Arbitrary Algorithms", *Computer Magazine*, Oct., 1988.
- [15] Arvind, D. Culler, "Dataflow Architectures", *Annual Reviews in Computer Science*, MIT, Laboratory for Computer Science, 1986.
- [16] J. B. Dennis, "First Version Data Flow Procedure Language", *Technical Memo MAC TM61*, May 1975, MIT Laboratory for Computer Science.
- [17] M. Toyokura et al., "A Video Digital Signal Processor with a Vector-Pipeline Architecture", *ISSCC*, 1992.
- [18] J. Wawrznyek, "A Reconfigurable Concurrent VLSI Architecture for Sound Synthesis", *VLSI Signal Processing II*, Nov. 1986.
- [19] P. Hilfinger, "A High Level Language and Silicon Compiler for Digital Signal Processing", *Proc. IEEE Custom Integrated Circuits Conference*, May 1985.
- [20] P. Papamichalis, "Digital Signal Processing Applications with the TMS320 Family, Vol. 3", *Prentice Hall*.
- [21] T. Nishizawa et al., "A Hidden Surface Processor for 3-Dimension Graphics", *ISSCC*, 1988.
- [22] K. Parhi and D. Messerschmitt, "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining using Scattered Look-ahead and Decomposition", *IEEE Transactions on ASSP*, p 1109, vol.37, July 1989.
- [23] C. Shung et al., "A 30 MHz Trellis CODEC Chip for Partial-Response Channels", *ISSCC*, 1991.