VLSI SIGNAL PROCESSING

The field of very-large-scale integrated (VLSI) circuit signal processing concerns the design and implementation of signal-processing algorithms using application-specific hardware, including programmable digital signal processors and dedicated signal processors implemented with VLSI technology. In this article, we will survey important developments in this field, including algorithm design, architecture development, and design methodology.

Implementation of Digital Signal-Processing Algorithms

Signal processing concerns the acquisition, filtering, transformation, estimation, detection, compression, and recognition of signals represented in multiple media and multiple modalities, including sound, speech, image, video, and others. In *digital* signal processing, a natural signal is first sampled and quantized using an analog-to-digital converter. The result is a stream of numbers that will be processed using a digital computer. The results will be converted back to continuous form using digital-to-analog converter.

An implementation of a digital signal-processing algorithm consists of the computer program of that algorithm and the hardware on which the program is executed. In many signal-processing applications, real-time processing is an essential requirement. *Real time* implies that the results of a signal-processing algorithm must be computed by a predefined deadline after the inputs are sampled. For example, in a cellular phone, the speech coding signal-processing algorithm must be executed to match the speed of normal conversation. An implementation of a real-time signal processing application has three special characteristics:

1. Input signal samples are made available while the program is being executed. The computation cannot be started early until the input signal samples are received.
2. Results must be computed before the prespecified deadline. When real-time constraint is not met, the quality of services will be dramatically compromised.
3. A vast amount of operations must be computed. In Table 1, raw sampling rates, sometimes known as the *throughput rate,* of several different signals are listed. On average, each signal sample will require several

Table 1. Typical Sampling Rates of Signals

Type of Signal	Sampling Rate
Speech	22 kHz (22,000 samples per second)
CD Audio	44 kHz
Video phone (320 × 240 × 3 pixels/frame, 15 frame/s)	3.456 MHz (3,456,000 samples per second)

fixed-point or floating-point arithmetic operations to process. Hence signal-processing algorithms are often computation-intensive.

An efficient implementation of a real-time signal-processing algorithm must be able to perform extremely large amounts of arithmetic operations within a short duration. In other words, it must sustain high throughput rate.

Signal processing is often found in embedded systems such as electric appliances where the user interacts with the system's main function instead of specific signal-processing algorithms. For example, speech coding is regularly performed in a cellular phone while the user may never be aware of its existence.

A signal-processing algorithm can be implemented on a general-purpose computer, a special-purpose programmable digital signal processor, or even dedicated hardware. The tasks of implementation involve algorithm design, code generation (programming), and architecture synthesis. With the same integrated-circuit technology, a specialized hardware platform may offer better performance than a general-purpose hardware by eliminating redundant operations and components. However, the design and manufacturing cost will be higher.

Digital signal-processing algorithms distinguish themselves from general programs in a number of ways:

1. *Numerical Computation Intensive.* These programs often contain nested loops of numerical computations, including multiplication, division, and elementary function evaluations.
2. *Deterministic Control Flow.* Signal-processing algorithms, unlike general computer programs, often have predictable control flow. Data-dependent conditional branches are less likely to occur.
3. *Input/Output Intensive.* Signals are often processed as a stream of data samples. They are less likely to be referred to ones processed. However, they require sustained input and output operations at high speed in order to meet the required throughput rate demand.

VLSI Application-Specific Processors

In the early 1960s, most digital signal-processing algorithms, such as the fast Fourier transform (FFT), were implemented in Fortran programs, running on a general-purpose mainframe. It could take hours to process a short 30 second speech. Obviously, general-purpose computing systems are insufficient to meet the high throughput rate demanded by a real-time signal-processing algorithm. However, dedicated application-specific computing systems are too expensive to be a realistic solution for most commercial signal-processing applications.

This situation changed in mid-1970s. Quantum leaps in integrated-circuit (IC) manufacturing technology led to the era of VLSI systems. By 1980, hundreds of thousands of transistors could be reliably, economically fabricated on a single silicon chip. With the transistor count per chip growing exponentially, it became quite clear that to manage the design complexity of VLSI circuits, IC design methodologies must be revolutionized. In 1980, Mead and Conway championed the notion of structured VLSI design. In their seminal book *Introduction to VLSI Systems* (1), it is argued that a hierarchical design style that exhibits regularity and locality must be adopted in order to design millions of transistors on a single chip. A novel architecture called a *systolic array* was used as an example that satisfies all these requirements.

This idea of structured VLSI design further inspired the concept of a *silicon compiler,* which, in analogy with the software compiler, would automatically generate a silicon implementation starting from a high-level description. These pioneering ideas stimulated many important developments in the IC industry, such as the proliferation of electronic design automation (EDA) tools, the popularity of semicustom design styles, including gate array and standard cell layout, and the availability of silicon foundry services. By the mid-1980s, a new industry known as application specific IC (ASIC) design was thriving. Numerous chip sets for video coding, three-dimensional audio processing, and graphic rendering have been available on the market at appealing cost.

VLSI and Signal Processing

The VLSI revolution affected signal-processing system architecture in a number of important ways:

1. *High Speed.* As the IC manufacturing technology evolves, the feature dimensions of transistors continue to shrink. Smaller transistors mean faster switching speeds, and hence higher clock rates. Faster processing speed means that more demanding signal-processing algorithms can now be implemented for real-time processing.
2. *Parallelism.* Higher device density and larger chip area promised to pack millions of transistors on a single chip. This makes it feasible to exploit parallel processing in order to achieve an even higher throughput rate by processing multiple data streams concurrently. To exploit the benefit of parallel processing fully, however, the formulation of signal-processing algorithms must be reexamined. Algorithm transformation techniques are also developed to exploit maximum parallelism from a given digital signal-processing algorithm formulation.
3. *Local Communication.* As device dimensions continue to decrease and chip area continues to increase, the cost of intercommunication becomes significant in terms of both chip real estate and transmission delay. Hence, pipelined operation with a local bus is preferred to broadcasting using global interconnection links. Compiler and code-generation methods need to be updated to maximize the efficiency of pipelining.
4. *Low-Power Architecture.* Smaller transistor feature size makes it possible to reduce the operating voltage and thereby significantly reduces the power consumption of an IC chip. This trend makes it possible to develop digi-

tal signal-processing systems on portable or hand-held mobile computers.

On the other hand, the stringent performance requirement and regular, deterministic formulation of signal processing applications also profoundly influenced VLSI design methodology.

1. *High-Level Synthesis Design Methodology.* The quest to streamline the process of translating a complex algorithm into a functional piece of silicon that meets stringent performance and cost constraints has led to significant progress in the area of high-level synthesis, system compilation, and optimal code generation. Ideas such as data-flow modeling, loop unrolling, and software pipelining, which were originally developed for general-purpose computing systems, have enjoyed great success when applied to aid the synthesis of an application-specific signal-processing system from high-level behavioral description.
2. *Multimedia Processing Architecture.* With the maturity and popularity of multimedia signal-processing applications, general-purpose microprocessors have incorporated special-purpose architecture such as the multimedia extension instruction set (e.g., MMX). Signal processors also led the wave of novel architectural concepts such as very long instruction word architecture. In fact, it is argued that incorporating multimedia features is the only way to sustain the exponential growth in processing performance through the next decade.

SYSTOLIC ARRAY

A systolic array (2,3) is an unconventional computer architecture proposed by H. T. Kung. It features a regular array of identical, simple processing elements operated in a pipelined fashion. This architecture is so named because data samples and intermediate results are processed in a systolic array in a manner analogous to the way blood is pumped by the heart—a phenomena called systole circulation.

A systolic array exhibits characteristics of parallelism (pipelining), regularity, and local communication. A large number of signal-processing algorithms and numerical linear algebra algorithms can be implemented using systolic arrays. For example, consider a convolution algorithm

$$y(n) = \sum_{k=0}^{\min(n,M-1)} h(k)x(n-k), \qquad n = 0, 1, \ldots$$

This algorithm is usually implemented with a two-level nested do loop:

```
For n = 0, 1, 2, . . .
  For k = 0 to min(n, M−1)
    y(n) = y(n) + h(k) * x(n − k)
  end
end
```

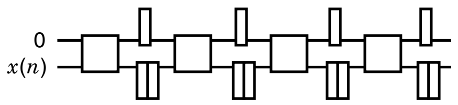


Figure 1. Convolution systolic array.

It can be implemented using a systolic array as depicted in Fig. 1, in which each square box represents a processing element as shown in Fig. 2. The narrow rectangular box represents delay.

The previous implementation corresponds to the following algorithm formulation:

```
s(n, 0) = x(n); g(n, 0) = 0; n = 0, 1, 2, . . .
g(n, k + 1) = g(n, k) + h(k) * s(n, k); n = 0,
  1, 2, . . .; k = 0, . . ., M − 1,
s(n, k + 1) = s(n, k); n = 0, 1, 2, . . .; k =
  0, . . ., M − 1
g(n + 1, k + 1) = g(n, k + 1); n = 0, 1, 2,
  . . .; k = 0, M − 1
s(n + 2, k + 1) = s(n, k + 1); n = 0, 1, 2,
  . . .; k = 0, . . ., M − 1
y(n) = g(n + M, M); n = 0, 1, 2, . . .
```

In this formulation, n is the time index and k is the processing element index. It can be verified manually that such a systolic architecture yields correct convolution results at the sampling rate of $x(n)$. M processing elements (PEs) are used. Moreover, every PE is identical and performs its computation in a pipelined fashion.

Systolic-Array Design Methodology

Given an algorithm represented as a nested do loop, a systolic-array structure can be obtained by the following.

1. Deduce a localized dependence graph of the computation algorithm. Each node of the dependence graph represents a computation of the innermost loop body of an algorithm represented in a regular nested loop format. Each arc represents an interiteration dependence relation. A more detailed introduction to the dependence graph will be given later in this article.
2. Project each node and each arc of the dependence graph along the direction of a projection vector. The resulting geometry gives the configuration of the systolic array.
3. Assign each node of the dependence graph to a schedule by projecting them along a scheduling vector.

To illustrate this idea, let us consider the convolution example. The dependence graph of the convolution algorithm is

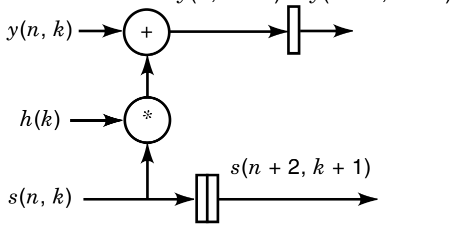


Figure 2. A processing element of the convolution systolic array.

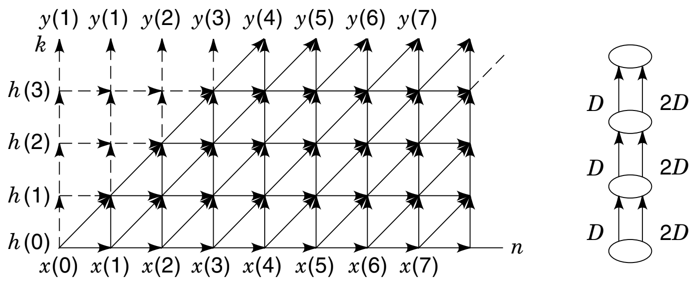

y(1) y(1) y(2) y(3) y(4) y(5) y(6) y(7)
k
h(3)
h(2)
h(1)
h(0)
x(0) x(1) x(2) x(3) x(4) x(5) x(6) x(7)
n
D
2D
D
2D
D
2D

Figure 3. Dependence graph (DG) of the convolution example (left) and the systolic array after projecting the DG to a processor array (right).

shown in Fig. 3 The input $x(n)$ is from the bottom. It will propagate its value (unaltered) along the northeast direction. Each of the coefficients $\{h(k)\}$ will propagate toward the east. The partial sum of $y(n)$ is computed at each node and propagated toward the north direction.

If we *project* this dependence graph along the [1 0] direction, with a schedule vector [1 1], we obtain the systolic-array structure shown to the right. To be more specific, each node at coordinate (n,k) in the dependent graph is mapped to processing element k in the systolic array. The coefficient $h(k)$ is stored in each PE. The projection of the dependence vector [1 1] associated with the propagation of $x(n)$ is mapped to a physical communication link with two delay elements (labeled by $2D$ in the figure to the right). The dependence vector [0 1] is mapped to the upward communication link in the systolic array with one delay. Figure 1 is identical to the right side of Fig. 3 except more details are given.

The systolic design methodology by mapping a dependence graph into a lower-dimensional systolic array is intimately related to the loop transformation methods developed in the parallel processing community. A detailed description of the loop transform can be found in the corresponding section entitled "Loop Transformation."

PROGRAMMABLE DIGITAL SIGNAL PROCESSORS

Programmable digital signal processors (PDSPs) are microprocessors designed specifically for digital signal-processing applications. They contain special instructions and special architecture supports that will execute computation-intensive DSP algorithms more efficiently.

Notably, all PDSPs have a multiply-and-accumulate (MAC) instruction, which can perform fixed-point multiply and add operations

```
R4 ← R1 + R2 * R3
```

in a single clock cycle. To support this operation, almost all PDSPs have a hardware parallel multiplier. For DSP applications, PDSPs often contain instructional support of *saturation arithmetics.* With a conventional binary adder, the result of addition is subject to a modulo 2^N operation. Hence if $N = 4$, the result of 9 + 9 becomes 2 (1001 + 1001 = 10010 modulo 10000 = 0010). Here we assume unsigned number representation. In saturated arithmetics, if the result of computation exceeds the dynamic range, it is clamped to the maximum. That is, 9 + 9 = 15 (1001 + 1001 = 1111) when $N = 4$. While the majority DSP applications use fixed-point arithmetic, later generations of PDSPs also include floating-point multipliers. Many DSP algorithms contain multiple nested loops. A number of PDSPs contain a special REPEAT instruction to support efficient execution of loop nests using dedicated counters to keep track of loop indices.

Another key feature of PDSPs is the adoption of a *Harvard memory architecture,* which contains separate program memory and data memory so as to reduce delay in fetching instruction and data samples. This is different from the conventional *Von Neuman architecture* in which program and data are stored in the same physical memory.

To emphasize the intensive input and output demands of most signal-processing applications, several PDSPs have built-in direct memory access (DMA) channels and a dedicated DMA bus to handle data input/output. To maximize data input/output efficiency, some new PDSPs even contain a dedicated video port or high-speed telecommunication port.

Some PDSPs, such as the TMS32040 and INMOS T800, provide multiple point-to-point serial input/output links to facilitate distributed memory parallel processing. In the TMS3208x series, a single-chip parallel computer contains a floating-point general-purpose core processor, four 16-bit DSP processors, a cross-bar switch, and four separate data memory modules. Now we will first give some historical notes on the evolution of PDSP architecture. Then we will briefly survey several representative PDSPs.

Evolution of PDSP Architecture

Fully stand-alone programmable digital signal processors appeared in the early 1980s. The first commercially available DSP processor, the NEC 7720, had a hardware multiplier and a Harvard architecture, which supported concurrent memory access to data memory and program memory, but the instruction set did not support saturation arithmetic. Intel introduced the 2920 processor in 1980. A unique feature is that it includes analog-to-digital and digital-to-analog converters on-chip. But its performance for computationally intensive tasks was not impressive because it had no hardware multiplier. In addition, it was hard to get parameters into the chip because it lacked a digital interface. Texas Instruments (TI) introduced the TMS32010 processor in 1982. The 32010 processor had a hardware multiplier and Harvard architecture with separate on-chip buses for data memory and program memory. This was the first programmable DSP to support executing instructions from off-chip program random access memory (RAM) without any performance penalty. This feature brought programmable DSPs closer to the microprocessor–microcontroller programming model. In addition, TI's emphasis on development tools and libraries led to widespread use. At about the same time, there were many vendors with competing products. Some of their architectural and performance characteristics are summarized in Table 2.

In these early PDSPs, DSP-tailored instructions such as MAC, delay elements (DELAY), loop control (REPEAT), and other flow-control instructions were added to improve the programmability of the processors. Moreover, a special address generator unit with bit-reversal addressing support has been incorporated to support DSP algorithms such as the fast Fourier transform (FFT). We note that the internal data and program memories are relatively small. A significant performance penalty will be paid if the program does not fit the on-chip memory.

Table 2. Early (before 1990) Implementations Programmable Digital Signal Processors

Digital Signal Processor	Manufacturer	Year[a]	Package	Internal Data RAM	Internal Data ROM	Internal Program RAM	Multiply Format	Multiply and Clock Cycle	1024 Point FFT Time (Fixed Pt.)
TMS32010	TI	1982	40 DIP	144×16		$1.5K \times 16$	$16 \times 16 \rightarrow 32$	200 ns	42 ms
TMS320C25	TI	1986	40 DIP	288×16		$4K \times 16$	$16 \times 16 \rightarrow 32$	100 ns	7.1 ms
TMS320C30	TI	1988	176 PGA[c]	$2K \times 32$		$4K \times 32$	$32 \times 32 \rightarrow 32 \times 10^8$	60 ns	
DSP56000	Motorola	1986	88 PGS[d]	512×24	512×24	512×24	$24 \times 24 \rightarrow 56$	97.5 ns	4.99 ms
DSP96001	Motorola	1988	163 PGA	$1K \times 32$	$1K \times 32$	544×32	$32 \times 32 \rightarrow 96$	75 ns	<2 ms
DSP16	AT&T		84 PLCC	512×16	$2K \times 16$		$16 \times 16 \rightarrow 32$	55 ns	
DSP32	AT&T	1984	100 PGA[f]	$1K \times 32$	512×32		$32 \times 32 \rightarrow 40$	244 ns	20 ms
DSP32C	AT&T	1988	133 PGA[f]	$1K \times 32$	$2K \times 32$		$32 \times 32 \rightarrow 40$	80 ns	
MPD 7720	NEC	1981		128×16	512×13	512×23	$16 \times 16 \rightarrow 31$	250 ns	77 ms
NEC 77230	NEC	1986	68 PGA	$1K \times 32$	$1K \times 32$	$2K \times 32$	$24 \times 10^8 \rightarrow 47 \times 10^8$	150 ns	12.3 ms
Intel 2920	Intel	1979		40×25		192×24		NA	
IBM RSP	IBM	1983	171 pins					2 bit per cyc.	
ADSP2100	Analog Device	1986	100 PGA				$16 \times 16 \rightarrow 32$	125 ns	7.2 ms
DSSP-VLSI	NTT	1986		512×18		$4K \times 18$	18-bit 12×10^6		
MSM 6992	OKI	1986	132 PGA	256×32		$1K \times 32$	22-bit 16×10^6	100 ns	
MSP32	Mitsubishi		124 PGA[f]	256×16		$1K \times 16$	$32 \times 16 \rightarrow 32$	150 or 450 ns	
MB8764	Fujitsu		88 PGA[h]	256×16		$1K \times 24$			
TS68930[9]	Thomson		48 DIP	256×16	512×16	$1K \times 32$	$16 \times 16 \rightarrow 32$	160 ns	
NS LM32900	National		172 PGA[f]				$16 \times 16 \rightarrow 32$	100 ns	13.4 ms
ZR34161 VSP	Zoran		48 DIP	128×32	$1K \times 16$		16 bit vector eng.	100 ns	2.4 or 3.3 ms
A100	Inmos		84 PGA				4, 8, 12, 16 bit		

[a] Year of publication sometime used.
[b] Peak performance is used unless noted.
[c] Also in an 100 QFP package.
[d] Also in an 88 surface-mount package.
[e] 13.33 MIPS sustained.
[f] Also in a 40 DIP package.
[g] Gate count was listed.
[h] Also in an 84 PLCC package.

Later, several floating-point PDSPs, such as TMS32030 and Motorola DSP96001, appeared in the market. A key advantage of a floating-point arithmetic unit is its large dynamic range. With fixed-point arithmetic, the dynamic range of the intermediate results must be carefully monitored. Sometimes as much as one-quarter of the instruction cycles are wasted on checking the overflow condition of intermediate results. Design rule specifications of near 1 μm allowed most of these processors to integrate a large number of peripherals into the chip, as well as implementing extensive input/output (I/O) facilities in addition to extended basic services such as registers and word lengths. To provide more efficient I/O facilities, some PDSPs also provide on-chip DMA controllers, as well as dedicated DMA buses that allow the true concurrent operation of both DMA and CPU. Although DSP hardware has advanced dramatically, DSP software and productivity tools lag far behind. PDSPs lack effective programming environments. Engineers still have to hand-code the time- and space-critical segments of the DSP algorithms, leaving only rudimentary tasks to the high-level language compiler.

Recent Developments in Programmable Digital Signal Processors

Native Signal Processing: Subword Parallelism via Multimedia Extension. Native signal processing refers to the processing of a multimedia data stream using the host general-purpose microprocessor rather than application-specific PDSPs. Native signal processing become possible because the increase of raw processing power of general-purpose microprocessors in the late 1990s is sufficient to handle certain real-time signal-processing functions such as speech coding or telephone modem function. To further enhance the processing power of the native host microprocessor to handle video data streams, new generations of general-purpose microprocessors incorporated new instructions. For Intel's Pentium processor, it is called MMX, which stands for multimedia extension.

A key feature of these multimedia extension instructions is the use of subword parallelism. Most multimedia (especially video) data streams use an 8 bit data sample, while the latest general-purpose microprocessors use a 64 bit word length. Thus a 64 bit data path should be able to perform as many as eight 8 bit arithmetic or logic operations in parallel, with the potential to increase speed by a factor of 8 for these multimedia operations. Subword parallelism basically is an application of the single-instruction, multiple-data (SIMD) parallel programming model. While it promises significant performance enhancement for general-purpose microprocessors to process real-time signal-processing tasks, programs using MMX instructions are just as difficult to optimize compared with PDSPs if not worse. Programmers must handle issues such as data alignment and instruction pipelining with great care.

Custom DSP Core Processors. A new trend in designing new processors without redevelopment of new programs is to design a new chip with an existing processor as a building block. This is possible because the advances of VLSI technology allow more transistors to be put on the same chip. An advantage of this approach is that the software development cost can be greatly reduced, while the hardware performance can be significantly improved due to smaller feature size, higher clock frequency, and the larger scale of integration.

Table 3. Features of Modern Programmable Digital Signal Processors

Name	TMS320C82	Mpact 2	TMS320C6201	Trimedia TM1	MSP
Architecture	Multiproc.	VLIW	VLIW	VLIW	Multiproc.
CMOS[a] technology (μm)	0.5	0.35	0.25	0.35	0.35
V_{cc} (V) operating voltage	3.3	3.3	2.5	3.3	3.3
Power (W)	3 (at 50 MHz)	4.45	0.75	4	4
Clock frequency (MHz)	50,60	125	200	100	100
Performance (BOPS[b] 8-bit integer)	1.5	6	1.6 (32 bit)	4	64
Manufacturer	TI[c]	Chromatic res.	TI[c]	Philips	Samsung

[a] CMOS stands for complementary metal-oxide semiconductor.
[b] Billions of operations per second.
[c] Texas Instruments.

Low-Power DSPs. Battery-powered products, such as multimedia notebook personal computers (PCs) and digital cellular phones, are driving the demand for lower-power DSP processors. Power reduction in DSPs is achieved using three design techniques: low voltage, gated clocks, and sleep modes. Low voltage is a result of reducing transistor feature size and is closely tied to the IC manufacturing process used. Many modern DSP processors now have low-power versions. The gated clock is a logic-level design methodology to block the clock signal from reaching portions of the function unit on chip, thereby partially shutting down the hardware that is not needed. An example is the TMS320C54 PDSP, which is targeting speech coder applications. A related approach is to offer multiple sleep modes to provide better power management at the system level. A balance must be sought between low-power consumption and delay incurred to resume normal operations.

Very Long Instruction Word. Very long instruction word (VLIW) architectures have become widely adopted by a new set of media processors. Examples include the Philips Semiconductor's Trimedia chip, Chromatic's MPACT 3000, as well as Texas Instruments' TMS320C6201. In a VLIW architecture, a very long instruction word is used to control multiple function units to operate on different data streams concurrently. In view of the fact that many DSP algorithms have a high degree of inherent parallelism that can be exploited, VLIW architectures are well suited to DSP applications. Some characteristics of these processors are summarized in Table 3.

The VLIW architecture is similar to the superscalar architectures, which is the state of the art in general-purpose microprocessor architecture, as both use multiple function units to exploit instruction level parallelism. However, in a VLIW architecture, the parallelism is explored statically during compile time while in a superscalar architecture, the parallelism is exploited dynamically during run time. For DSP applications, VLIW is a good match.

DESIGN TOOLS AND DESIGN METHODOLOGIES

In order to implement a given digital signal-processing algorithm efficiently, it is important to have powerful design tools and a suite of proven design methodologies. Among numerous EDA tools, the silicon compiler is closely related to synthesizing digital signal-processing applications. One specific focus of the field of VLSI signal processing is to achieve the highest performance by carefully matching the signal-processing algorithm and the underlying architecture. A few mature design methodologies that explore both inter- and intraiteration parallelism will be surveyed in this section.

Silicon Compiler and System Compiler

A direct impact of the Mead–Conway structured design style is accelerated research and development in EDA tools, in particular, design synthesis tools. The objective is to generate a layout of subsystems automatically without performing actual layout. In the mid-1980s, a number of *module generators* had been developed that were capable of synthesizing high-quality subsystems, such as memory, register files, ALU (arithmetic logic unit), and random logic blocks by choosing appropriate parameters on a form-based interface. The notion of module generation was subsequently generalized to a *silicon compiler*. A silicon compiler, on the other hand, is analogous to a high-level language compiler that promises to translate high-level behavioral or structural level descriptions of the system into low-level chip layout. In 1981, the FIRST silicon compiler from the University of Edinburgh was developed that could generate customized IC chip layout of a given DSP algorithm using bit-serial architecture. Direct synthesis of a general-purpose microprocessor using a silicon compiler was still not practical until now. Most successful examples of silicon compilers have focused on application-specific digital signal-processing applications. It was not until the early 1990s that designers realized that in addition to layout generation, the design of a digital signal-processing system requires concurrent hardware and software design. Hence a new notion of *system compiler* emerged which attempts to guide the designer through the entire design process, starting from behavioral specification, system partitioning, hardware and software trade-off analysis, all the way to chip layout generation.

Exploring Interiteration Parallelism via Loop Transformation

Many digital signal-processing algorithms contain the formulation of nested do-loops, which are time-consuming to execute. Parallel execution of several loops simultaneously can often be realized via proper transformation of nested do-loop programs. In this subsection, loop transformation techniques that exploit interiteration parallelism will be surveyed. The inner loop body will be treated as an atomic task and executed in a single processing element. The derivation and notation follows roughly the content of Ref. 4.

Regular Iterative Nested Loop Algorithms. A general m-level nested loop has the following format:

```
L1: DO i1 = p1, q1
L2:      DO i2 = p2, q2
 ⋮           ⋮
Lm:           DO im = pm, qm
                    H(i1, i2, . . ., im)
               Enddo
                 ⋮
         Enddo
    Enddo
```

The loop indices $\{i_k;\ 1 \le k \le m\}$ form an $m \times 1$ index vector $\boldsymbol{i} = (i_1, i_2, \ldots, i_m]^T$, which corresponds to a lattice point in the m-dimensional space. All lattice points that may occur between the loop bounds form an index space $\mathscr{R}$ of this nest loop. The integers $\{p_k, q_k;\ 1 \le k \le m\}$ are *loop bounds*. $H(i_1, i_2, \ldots, i_m)$ is called the *loop body*. In the loop transformation, we assume that the entire loop body, which may contain more than one statement, is to be executed in a single processor. We will study methods to reformulate the loop indices and their bounds so that more than one iteration in the loop nest can be executed in parallel using multiple processors.

If the loop bounds are all constant, the index space forms a rectangular parallelepiped. A more general situation is that the loop bounds are a linear (affine) function with integer coefficients of the outer loop indices. In this case, the loop bounds can be formulated as two inequalities:

$$\boldsymbol{p}_0 \le \boldsymbol{Pi} \quad \text{and} \quad \boldsymbol{Pi} \le \boldsymbol{q}_0$$

where $\boldsymbol{p}_0$ and $\boldsymbol{q}_0$ are constant integer-valued vectors, and $\boldsymbol{P}$ and $\boldsymbol{Q}$ respectively, are integer-valued upper triangular coefficient matrices. If $\boldsymbol{P} = \boldsymbol{Q}$, then the corresponding loop nest can be transformed in the index space such that the transformed algorithm has constant iteration bounds. Such a nested loop is called a *regular* nested loop.

Dependence Vector and Dependence Matrix. A schedule $\mathscr{S}$: $\boldsymbol{i} \to \mathrm{t}(\boldsymbol{i})$ is a mapping from each index point $\boldsymbol{i}$ in the index space $\mathscr{R}$ to a positive integer $\mathrm{t}(\boldsymbol{i})$ that dictates when this iteration is to be executed.

An iteration $H(\boldsymbol{i})$ will be executed before $H(\boldsymbol{j})$ if its index vector $\boldsymbol{i}$ lexicographically proceeds index vector $\boldsymbol{j}$. That is, $\boldsymbol{i} < \boldsymbol{j}$. This implies there exists an integer r, $1 \le r \le m$, such that $i_k = j_k$ for $k < r$, and $i_r < j_r$. For example, $[1\ 3\ 4] < [2\ 1\ 1]$. An iteration $H(\boldsymbol{j})$ is dependent on iteration $H(\boldsymbol{i})$ if (a) $\boldsymbol{i} < \boldsymbol{j}$ and (b) $H(\boldsymbol{j})$ will read from a memory location (including registers) whose value is last written during execution of iteration $H(\boldsymbol{i})$. The corresponding *dependence vector* $\boldsymbol{d}$ is defined as

$$\boldsymbol{D} = \boldsymbol{j} - \boldsymbol{i} \succ \boldsymbol{0}$$

A matrix $\boldsymbol{D}$ consisting of all dependence vectors of an algorithm is called a *dependence matrix.*

OBSERVATION. If $H(\boldsymbol{j})$ is dependent on $H(\boldsymbol{I})$, then $t(\boldsymbol{i}) < t(\boldsymbol{j})$.

Hence the dependence relation imposes a partial ordering on the execution of the iterative loop nest. If the last row of the dependence matrix contains all zero entries, the innermost loop can be replaced by a do-all loop to have all iterations executed concurrently.

Given a dependence vector $\boldsymbol{d} = (d_1, d_2, \ldots, d_m)^T$. For a positive integer $l < m$, if

$$d_1 = \cdots = d_{l-1} = 0$$

we say the *level* of this dependence vector is l. Moreover, we say that loop L_l carries a dependence. For example, consider the following dependence matrix

$$\boldsymbol{D} = [\boldsymbol{d}_1 \quad \boldsymbol{d}_2 \quad \boldsymbol{d}_3] = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & -1 \end{bmatrix}$$

the levels of dependence vectors $\boldsymbol{d}_1$, $\boldsymbol{d}_2$, and $\boldsymbol{d}_3$ are, respectively, 2, 3, and 1. Each of the three loops carries a dependence relation. However, if we interchange loop L_2 and loop L_3 by interchanging the second and the third rows of the $\boldsymbol{D}$ matrix, we have a new dependence matrix

$$\tilde{\boldsymbol{D}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix}$$

where the innermost loop carries no dependence because the levels of dependence of the transformed dependence vectors are respectively, 2, 2, and 1. As such the innermost loop can be executed in parallel (replaced by a do-all loop).

OBSERVATION. Exploiting Inner-Loop Parallelism: If the first nonzero element in each dependence vector is above loop level k, then all inner-loop nests, starting from level k, can be executed in parallel.

OBSERVATION. Exploiting Outer-Loop parallelism: To execute an outer loop in parallel (where each inner-loop nest is executed sequentially), the corresponding dependence matrix must have at least a row containing only zero entries.

If the objective is to exploit maximum parallelism, one would then want to transform the loop formulation so that there are as many loops as possible carrying no dependence.

Unimodular Loop Transformation. A square matrix containing integer entries with its determinant equal to 1 or -1 is called a *unimodular* matrix. A unimodular transformation of a loop nest is a linear affine transformation of each iteration index vector

$$\boldsymbol{i} \mapsto \boldsymbol{Ui} = \boldsymbol{k}$$

Such a transformation facilitates origin shift and rotation of the index space axis. If used properly, a unimodular transformation enables more loops to be executed in parallel.

A loop transformation matrix $\boldsymbol{U}$ is *valid* if for each $\boldsymbol{d}$ in $\boldsymbol{D}$, $\boldsymbol{Ud} \succ \boldsymbol{0}$. The dependence matrix of the transformed loop is $\boldsymbol{UD}$.

For example, consider the nested loop:

```
for i = 0,3
  for j = 0,3
    A(i, j) = B(i, j − 1) + C(i − 1, j)
  end
end
```

The dependence matrix is

$$\boldsymbol{D} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Now consider a unimodular matrix

$$\boldsymbol{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

The index vector $\boldsymbol{i}$ can be transformed into

$$\boldsymbol{Ui} = \begin{bmatrix} i+j \\ j \end{bmatrix} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix}$$

The indices of the variable B are transformed to:

$$\boldsymbol{U}\left\{\begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right\} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} k_1 - 1 \\ k_2 \end{bmatrix}$$

and indices of the variable C become

$$\boldsymbol{U}\left\{\begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} k_1 - 1 \\ k_2 - 1 \end{bmatrix}$$

The loop bounds can also be transformed with the $\boldsymbol{U}$ matrix. This leads to a transformed loop nest as follows:

```
for k1 = 1,6
  for k2 = max{0, k13}, min{3, k1}
    A(k1, k2) = B(k1 − 1, k2) + C(k1 − 1,
      k2 − 1)
  end
end
```

From the preceding discussion, in order to exploit inner-loop parallelism, we need to apply unimodular transformation to the dependence matrix so that each dependence vector has a nonzero element as high as possible. Specifically, the objective is to rotate the coordinate so that after transformation, each transformed dependence vector has a nonzero projection to the index axis corresponding to the outermost loop. On the other hand, in order to exploit maximum outer-loop parallelism, we should use unimodular transformation to create as many zero rows in the transformed dependence matrix.

Recurrent Algorithm Transformation

Recurrent algorithms are DSP algorithms that have both strong inter- and intraiteration dependence relations. Unlike loop transformation, here the granularity of each task is a basic arithmetic operation such as addition and multiplication of two numbers. The objective here is to implement a given recurrent algorithm on a parallel processor array of an unspecified configuration so as to achieve the desired throughput rate. Two types of transformations are essential in this case: (1) look-ahead transformation and (2) loop unrolling. The purpose of a look-ahead transformation is to reduce the theoretical minimum initiation interval—the minimum duration between the execution of two successive iterations, regardless of the number of available processing elements. The loop unrolling enables one to devise an efficient periodic schedule to implement the given recurrent algorithm on a fine-grained parallel processor array. The information presented in this section partially follows that given in Ref. 5.

Iterative Computation Dependence Graph and Minimum Initiation Interval. Let us consider a simple infinite impulse response (IIR) digital filter:

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

This algorithm can be represented by a data flow graph as given in Fig. 3.

There are three computation tasks: task A is the multiplication of a and $y(n-1)$, task C is the multiplication of b and $u(n)$, and task B is the summation of these two products. Let us use t_A, t_B, and t_C, respectively, to denote the time taken to execute each of these three tasks. In general, the time taken to execute different tasks need not be the same.

The dependency arcs from A to B and from C to B indicate that in order to execute task B, tasks A and C must be performed first. The arc from task B to task A with label 1 indicates that the result of task B will be used by task A during the next iteration. Thus, the label 1 indicates that one buffer is required to store $y(n)$ temporarily. Clearly, according to Eq. (1), the computation of $y(n)$ cannot be initiated until $y(n-1)$ is computed. Even if one has more than two processing elements to compute the two multiplications in parallel, it would still take $t_A + t_B$ units of time to compute $y(n)$. Hence the minimum time interval between successive iterations of this IIR filter that can be initiated is bounded by $t_A + t_B$. Note that this is equal to the sum of the computing time of all tasks in the loop divided by the number of buffers in that loop.

Based on this simple example, we can define some important terms. An iterative computation dependence graph (ICDG) is a directed graph consisting of a set of nodes, each indicating a computation task, and a set of arcs, each indicating a data-dependence relation between the source and destination tasks. A dependence arc labeled with a positive integer number indicates the number of delays inserted due to interiteration dependence.

An ICDG is computable if every loop of it contains at least one delay. The minimum initiation interval of an ICDG G, denoted by $I_{\min}(G)$, is defined as

$$I_{\min}(G) = \max_{C \in G} \frac{T_C}{\Delta_C}$$

where C is a loop in G, T_C is the sum of computing time of all tasks in C, and Δ_C is the sum of all delays on arcs of C.

Periodic Schedule and Static Task Assignment. Let us assume that $t_A = 20\ \mu$s, $t_B = 10\ \mu$s, and $t_C = 25\ \mu$s. A schedule of a

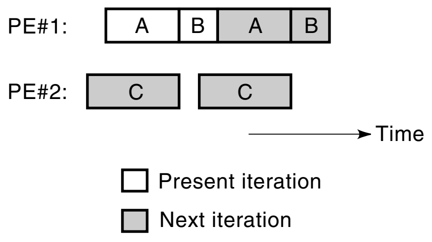


Figure 4. A schedule of a two-processor implementation of the IIR filter.

two-processor implementation of the IIR filter is given in Fig. 4.

In this schedule, tasks A and B are both assigned to (processing element) PE1, and task C is assigned to PE2. These assignments are *static* in that once a task is assigned to a PE, it will always be executed in that PE in every iteration.

The horizontal axis is time. The shaded boxes indicate the schedule of the next iteration. Since the schedule for every iteration will be the same, often one needs only to schedule one iteration of these periodic tasks. Thus, in a *periodic schedule,* only tasks within one iteration of the recurrent algorithm are scheduled.

Note also that the execution of task C overlaps with the execution of task B in the previous iteration. This is possible because these two tasks are assigned to two different PEs. A schedule that allows the execution of tasks in different iterations to occur simultaneously at different PEs is called an *overlapping* schedule.

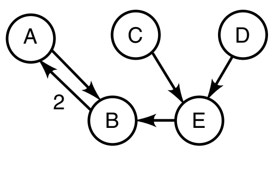
Look-Ahead Transformation: Unwinding Loops. Due to the limitation of the minimum initiation interval, no matter how many processing elements one may have, it would be impossible to realize the IIR filter in Eq. (1) so that the successive iterations can be initiated with an interval $d < I_{\min}$. When this happens, one may perform a look-ahead transformation to reformulate the recurrent algorithm. A look-ahead transformation is accomplished by substituting the iteration equation for $y(n-1)$ into that of $y(n)$. As such,

$$\begin{aligned} y(n) &= a[ay(n-2) + bu(n-1)] + bu(n) \\ &= a^2 y(n-2) + abu(n-1) + bu(n) \end{aligned} \tag{2}$$

The new coefficients a^2 and ab can be computed in advance. Eq. (2) corresponds to a new ICDG depicted in Fig. 5.

Although two new tasks are added, the minimum initiation interval of this transformed algorithm becomes

$$I_{\min}(G') = (t_A + t_B)/2 = 1/2 I_{\min}(G)$$



Task A: $a^2 \times y(n-2)$

Task B: Add results of tasks A and E

Task C: $b \times u(n)$

Task D: $ab \times u(n-1)$

Task E: Add results of tasks C and D

Figure 5. Modified iCDG of the IIR filter after look-ahead transformation.

In other words, the initiation interval is halved. After substituting N times, one has

$$y(n) = a^{N+1} y(n-N) + b \sum_{m=0}^{K} a^m u(n-m) \tag{3}$$

Thus, the new $I_{\min}$ is $1/N$ of the original one. The second term in Eq. (3) is a convolution operation and hence can be realized at any given rate as long as there are a sufficient number of PEs and the data $\{u(n)\}$ can be distributed into those PEs fast enough.

Loop Unfolding Transformation—Block Implementation. Loop unfolding is an algorithm transformation technique that exploits parallelism between successive loops. Let us consider the following recurrent equation: Given $y(-2)$, $y(-1)$, for $n = 0, 1, \ldots$

$$y(n) = y(n-2) + u(n) \tag{4}$$

Define $y_1(m) = y(2m)$, $y_2(m) = y(2m+1)$ and $u_1(m) = u(2m)$, $u_2(m) = u(2m+1)$ for $m = 0, 1, \ldots$, one may convert this single input, single-output linear system into a two-input, two-output system:

$$\begin{bmatrix} y_1(m) \\ y_2(m) \end{bmatrix} = \begin{bmatrix} y_1(m-1) \\ y_2(m-1) \end{bmatrix} + \begin{bmatrix} u_1(m) \\ u_2(m) \end{bmatrix} \tag{5}$$

Such a conversion is called *loop unfolding.* The purpose of loop unfolding is to exploit interiteration parallelism. In this example, since the even-number iterations are independent of odd-number iterations, they can be computed in parallel. In some works, Eq. (5) is called *block processing* because both the input and output of the original system are processed in blocks of two samples.

Loop unfolding does not alter the minimum initiation interval of a recurrent algorithm. In Eq. (4) the minimum initiation interval is $I_{\min} = t_{\text{add}}/2$, where t_{add} is the time taken to perform the addition operation. Thus to implement it in real time, the sampling period of $u(n)$, d, must satisfy $d \geq t_{\text{add}}/2$. On the other hand, the sampling periods of both $u_1(m)$ and $u_2(m)$ are now $2d$. The minimum initiation interval is the same for both: $I_1 = I_2 = t_{\text{add}}/1$.

Therefore, we must have $2d \geq t_{\text{add}}$. Hence from the overall input/output point of view, the initiation interval remains the same. This is illustrated in Fig. 6.

On the left of this figure is a realization of the system in Eq. (4). On the right is a realization of the unfolded system described in Eq. (5). Note that in the original system, the adder is to perform an addition for each $u(n)$. Hence the addition must be performed within d units of time. On the other hand, with the unfolded system, each adder receives a new input every $D' = 2d$ units of time. Hence additions can take twice as long to perform. In other words, slower hardware can be used to achieve the same throughput rate, taking advantage of the parallelism exploited via loop unfolding.

A *perfect rate graph* (PRG) is an ICDG such that every loop consists of one and only one delay. The significance of a perfect rate graph is that no further interiteration parallelism can be exploited by unfolding a PRG.

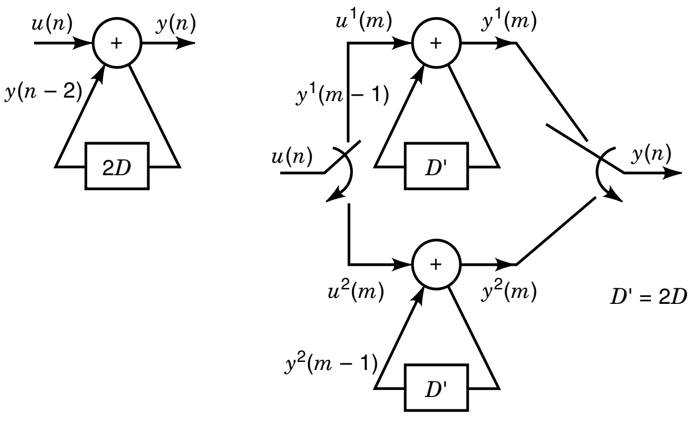


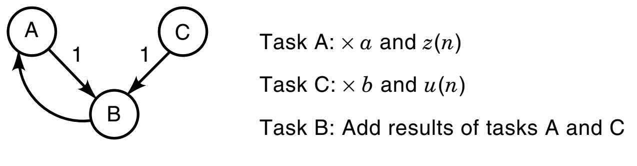
Figure 6. Original ICDG corresponding to Eq. (4) (left) and the equivalent ICDG after loop unfolding once (right). The clock cycle time of the right side circuit is twice as long as that of the left figure.

Retiming. Let us consider the substitute of variable $z(n) = y(n-1)$. Then Eq. (1) can be rewritten as $z(n) = az(n-1) + bu(n-1)$, which has a corresponding ICDG shown in Fig. 7. This ICDG can also be obtained from the ICDG corresponding to Eq. (1) via a procedure called node retiming. If the outgoing arcs from a node in an ICDG all have at least one delay on it, then one may remove the same number of delays from all the outgoing arcs, and add the same number of delays to all the in-coming arcs of the same node, without changing the behavior of the algorithm. This is illustrated in Fig. 8. What may be affected by retiming is the initial condition and perhaps the latency. The minimum initiation interval will remain the same as illustrated in preceding example.

Cut-Set Retiming and Node Retiming. A signal flow graph (SFG) can be made fully pipelined using retiming. The basic technique is called cut-set retiming.

Cut-Set Retiming Procedure.

1. Identify a *retimable cut set,* which consists of a set of edges in the ICDG such that (a) the graph G will be separated into two parts if these edges are removed; (b) there are *no* zero-delay edges of opposing directions across the cut set.
2. If necessary, scaling the delay by multiplying the delay by an integral factor, say α.
3. Transfer delays by subtracting one delay from edges of the same direction in the cut set and add it to edges of the opposing direction in the same cut set. Note that all the inputs (outputs) should remain at the same side of the cut set to ensure proper timing.



A
C
B
1
1
Task A: $\times a$ and $z(n)$
Task C: $\times b$ and $u(n)$
Task B: Add results of tasks A and C

Figure 7. Modified ICDG of the IIR filter in Eq. (1) after retiming.

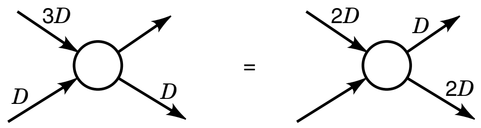


Figure 8. Illustration of delay transfer through a node during retiming.

Using cut-set retiming, one may enforce at least one delay element on each edge of the SFG, making it a systolic array. Node retiming is a special case of cut-set retiming where the cut set consists of all edges to and from a particular node in the graph.

BIBLIOGRAPHY

1. C. Mead and L. Conway, *Introduction to VLSI Design,* Reading, MA: Addison-Wesley, 1980.
2. H. T. Kung, Why systolic architectures, *IEEE Comput.,* **15** (1): 37–46, 1982.
3. S. Y. Kung, *VLSI Array Processors,* Englewood Cliffs, NJ: Prentice-Hall, 1988.
4. U. Banerjee, *Loop parallelization,* Boston: Kluwer, 1994.
5. K. K. Parhi, Algorithm transformation techniques for concurrent processors, *Proc. IEEE,* **77**: 1879–1895, 1989.

YU HEN HU
University of Wisconsin–Madison

VOICE CODING. See SPEECH CODING.

VOICE COMMUNICATION, NOISE. See SPEECH ENHANCEMENT.

VOICE COMMUNICATIONS. See TELEPHONE NETWORKS.