An innovative low-power high-performance programmable signal processor for digital communications

J. H. Moreno
V. Zyuban
U. Shvadron
F. D. Neeser
J. H. Derby
M. S. Ware
K. Kailas
A. Zaks
A. Geva
S. Ben-David
S. W. Asaad
T. W. Fox
D. Littrell
M. Biberstein
D. Naishlos
H. Hunter

We describe an innovative, low-power, high-performance, programmable signal processor (DSP) for digital communications. The architecture of this processor is characterized by its explicit design for low-power implementations, its innovative ability to jointly exploit instruction-level parallelism and data-level parallelism to achieve high performance, its suitability as a target for an optimizing high-level language compiler, and its explicit replacement of hardware resources by compile-time practices. We describe the methodology used in the development of the processor, highlighting the techniques deployed to enable application/architecture/compiler/implementation co-development, and the optimization approach and metric used for power–performance evaluation and tradeoff analysis. We summarize the salient features of the architecture, provide a brief description of the hardware organization, and discuss the compiler techniques used to exercise these features. We also summarize the simulation environment and associated software development tools. Coding examples from two representative kernels in the digital communications domain are also provided. The resulting methodology, architecture, and compiler represent an advance of the state of the art in the area of low-power, domain-specific microprocessors.

1. Introduction

The demand for high-performance microprocessors continues to be fulfilled by designs which achieve increasing performance levels, but do so at increasing power consumption. This is particularly true for general-purpose microprocessors, whose computing capabilities double approximately every 18 months (following Moore's law), and whose power requirements grow at a proportional rate. In contrast, the domain of digital communications, as represented by mobile devices, embedded communications, and multichannel applications, is uniquely characterized by the *expectation of achieving higher performance at a very low increase in power consumption*, if any at all.

To achieve efficient solutions for specific application domains, digital signal processor (DSP) architectures are "tuned" to the target domains. In addition to leveraging the advances in silicon technology, improvements in DSP performance are being achieved by the exploitation of the regular (i.e., repetitive) computations on data streams that are present in signal processing algorithms, which are different from the control-flow-driven computations that characterize general-purpose applications. As a result, contemporary DSPs focus on exploiting the natural

0018-8646/03/$5.00 © 2003 IBM

parallelism found in the applications by including features such as parallel data and instruction memories, simultaneous execution of multiple instructions (e.g., instruction-level parallelism, or ILP), simultaneous execution of the same instruction on multiple data elements (e.g., single-instruction multiple-data, or SIMD), block-repeat operations, and so on [1]. These features are exploited using mechanisms with much lower complexity than those found in general-purpose processors, thereby requiring lower power consumption.

Exploiting the multiplicity of features of a DSP in a real-time environment, such as those in the digital communications domain, has traditionally required programming the DSPs in assembly language. However, new applications are demanding not only higher signal processing computing capabilities but also more complex and diverse algorithms executed on the same processor, all at the same time. For example, a handheld device is expected to be able to simultaneously provide complex communications functions as well as multimedia applications such as MPEG-4. The rising complexity of the applications, coupled with the demand for short time-to-market, no longer tolerates programming only in assembly language. Instead, DSPs are expected to be programmed using a high-level language such as C [1, 2], like general-purpose processors. As a result, the cornerstone of contemporary digital communications is a new generation of digital signal processors characterized by their *power efficiency* as well as their *improved programmability* in a high-level language.

Application space

There are several ways of characterizing the domains of application for next-generation DSPs. One is to distinguish between "client-end" applications and "network-end" applications. Client-end applications include home gateways, access routers, set-top boxes, wireless handsets, games, and other handheld devices. Network-end applications include digital subscriber line access multiplexers (DSLAMs), voice-over-net (VoNet) gateways, and wireless base-stations. Certain functions appear in both of these application categories. For example,

- xDSL transceivers are contained in DSLAMs and also in some home gateways, access routers, and set-top boxes.
- Wireless digital baseband functions are contained in wireless base-stations and also in wireless handheld devices.
- Speech coders are contained in VoNet gateways, wireless base-stations, and almost all client-end devices.

One major difference between client-end and network-end applications is the way in which integration of functions leads to optimized solutions. Network-end applications tend to be optimized by "horizontal" integration, i.e., the integration of many copies of identical or similar functions. For example, a DSLAM integrates a large number of similar, if not identical, xDSL transceivers, while a VoNet gateway integrates a large number of similar, if not identical, speech coders and echo cancelers. In contrast, client-end applications tend to be optimized by "vertical" integration, i.e., the integration of a variety of different functions. For example, a home gateway may integrate an xDSL transceiver, a wireless LAN transceiver, a speech coder for voice-over-DSL, and some network processing functions, whereas a third-generation (3G) handset may integrate the wireless digital baseband, a speech coder, and audio and MPEG-4 functionality.

Domains of application for next-generation DSPs can also be characterized by factors that are limiting with respect to their ability to satisfy application requirements. In particular, applications can be characterized in the following way:

- *Power-bound applications*. These are applications in which the fundamental limiting factor is the power available to support electronics, but performance requirements (in terms of millions of instructions executed per second, or MIPS) are also severe. Examples include most battery-powered handheld devices, notably 3G handsets. Note that both active power and standby power are critical.
- *Performance-bound applications*. These are applications in which the fundamental limiting factor is the performance, i.e., the processing capability, of the DSP cores. Examples include DSLAMs and 3G wireless base-stations. Note that power is an important consideration as well, since in some cases DSLAM and base-station equipment is housed in small cabinets with limited ventilation that are mounted outdoors.
- *Memory-bound applications*. These are applications in which performance requirements are nontrivial, but the limiting factor is the amount of memory required. A key example is the VoNet gateway, where one DSP chip may have the performance to support speech coders for a large number of channels but the chip becomes dominated by memory because the memory required increases linearly with the number of channels.
- *Area-bound applications*. These are applications in which performance requirements are relaxed, but the chip area occupied by a DSP subsystem must be kept as small as possible, usually for cost reasons. Examples include certain client-end solutions that have become commodities.

The foregoing discussion indicates that many key applications impose severe power and performance

Table 1 Distinctive features characterizing contemporary DSPs.

Feature	*Alternatives*
Number of instructions issued	Single-issue Multiple-issue
Instruction scheduling	Statically scheduled Dynamically scheduled
Register space	Centralized Partitioned/replicated Homogeneous Heterogeneous
Pipeline hazards	Interlocked Non-interlocked
Organization of functional units	Global to all registers Clustered Homogeneous Heterogeneous
Data parallelism	Single instruction on different registers (SIMD) Single instruction on subregisters (SIMD with packed data) Multiple instructions on different registers (VLIW) Multiple instructions on subregisters (VLIW and SIMD with packed data) Multiple instructions on different registers (VLIW), in conjunction with single instruction on different registers (SIMD with disjoint data)

requirements on next-generation DSPs. Indeed, it appears reasonable to define a figure of merit for DSPs that combines performance and power, as performance per unit power [e.g., MIPS per mW, multiply–accumulates (MACs) per second per mW].

A common thread that runs through all of the applications mentioned above is the ongoing development of new algorithms. A variety of new speech-compression techniques are being investigated by ITU-T, 3GPP, and other standards bodies. New modulation formats, such as discrete multitone (DMT) and orthogonal frequency-division multiplexing (OFDM), have been identified and are being implemented for applications such as xDSL and wireless LAN. Channel coding methods that lead to near-Shannon-limit throughput, such as turbo codes and low-density parity-check (LDPC) codes, have been discovered (or rediscovered) and are being considered for use in xDSL and 3G wireless. For implementers of systems to take advantage of these developments, time-to-market is critical. Thus, efficient implementations of these and other algorithms on DSP platforms must be realizable quickly, which means with an absolute minimum of hand-coding or hand-optimization performed in assembly language. In other words, a successful next-generation DSP must be "compiler-friendly"; ideally, code generated by a compiler should approach the performance and compactness of hand-written and optimized assembly code.

DSP architecture space

The characteristics of the applications and environments described above represent the target of most current efforts in the design of digital signal processors. In this context, it is interesting to inspect the architecture alternatives that are being pursued to fulfill those requirements. Some of the most distinctive of such features are listed in **Table 1**. For example,

- The StarCore SC140 [3] is a multiple-issue, statically scheduled processor with a centralized heterogeneous register space composed of data and address registers; dedicated functional units use the registers of its corresponding type (data units and address units, respectively). Parallelism is achieved by a combination of multiple instructions operating on different registers, with the ability to allocate multiple data items on a single register [very-long-instruction words (VLIW) and SIMD with packed data].
- The Texas Instruments C6x [4] is also a multiple-issue, statically scheduled processor with a homogeneous register space partitioned among two clusters, in which heterogeneous functional units use only the registers available within the cluster. As in SC140, data parallelism is also achieved by a combination of multiple instructions operating on different registers, with the

ability to allocate multiple data items on a single register (VLIW and SIMD with packed data).
- The Analog Devices Sharc** DSP [5] is another multiple-issue, statically scheduled processor with a heterogeneous register space composed of data and address registers, which is further partitioned among heterogeneous clusters (data clusters and address clusters). Functional units use only the registers available within each cluster. Parallelism is achieved in the same way as in SC140 and C6x, namely through a combination of VLIW and SIMD with packed data.

Power-aware co-design methodology

The development of a power-efficient microprocessor that meets performance requirements at minimum power dissipation requires that power consumption be considered at early stages in the design, particularly during the definition of the instruction set architecture (ISA) and microarchitecture. At these stages, the potential for power savings is more significant than at lower-level stages, and the opportunity for making power–performance tradeoffs is the largest, because even minor modifications may result in significant changes to the power–performance characteristics of a design [6–11]. Drawing a conclusion about the effectiveness of some architectural feature requires the evaluation of its effect on the architectural speed of the processor (instructions per cycle, IPC), its power, clocking rate, and cost. Certain features that improve the architectural speed may be very costly in terms of power dissipation, whereas others may affect the clocking rate.

Power–performance metrics of the form MIPS^{γ}/watt [6–11] are difficult to use for evaluating the energy efficiency of architectural features at early stages of design, for two reasons:

- Absolute power and performance data are typically unavailable.
- It is hard to reach an agreement between architects and circuit designers on the appropriate value of γ [11].

Consequently, to be able to perform a consistent power-efficiency analysis, a new power–performance metric is needed that combines relative changes in architectural speed, dynamic instruction count, average energy dissipated per executed instruction, and maximum clocking rate of the processor, resulting from design modifications at the architectural and microarchitectural levels. If an architectural feature improves such a power–performance metric, it would be considered energy-efficient according to the metric; that is, it results in a better design point in the power–performance optimization space.

Scope of the research

The observations summarized above are the basis for the eLite DSP project, an ongoing effort within the IBM Communications Research and Development Center, which is advancing the state of the art in power-efficient high-performance programmable DSP architectures as well as in methodologies for such designs. This effort grows from the understanding of the importance of an architecture that provides a balanced optimization of programmability in high-level language, power consumption, performance, development cost (hardware and software), and production cost (chip and system). To achieve these usually conflicting optimization goals, the design of the eLite DSP architecture and its implementations covers aspects ranging from algorithms, applications, and a high-level language compiler, down to circuit-level technology. The resulting architecture is a multiple-issue statically scheduled processor with a heterogeneous set of register files spread throughout specialized units. Parallelism is achieved by executing multiple instructions simultaneously (VLIW), in conjunction with single instructions operating on different registers from a large multi-port register file (single-instruction multiple disjoint data, SIMdD), as well as single instructions operating on packed data from a register file with few ports (single-instruction multiple packed data, SIMpD). A novel indirect register-addressing mechanism enables the dynamic composition of vectors with four elements selected from the large multi-port register file.

In the rest of this paper, we describe how all of the aspects listed above have influenced the design choices made in the eLite DSP architecture. In Section 2, we describe the co-design methodology applied to the eLite DSP. In Section 3, we provide a description of the architecture, and Section 4 gives a brief description of the organization of the processor. In Section 5 we indicate the salient features of the associated optimizing compiler. Section 6 presents a summary of the tools and architecture simulator used for software development and architecture evaluation, including a description of the features that enable their automatic generation. Examples of kernels from representative algorithms coded for this architecture are illustrated in Section 7. In Section 8 we summarize the fundamental concepts behind the power–performance evaluation methodology deployed. Section 9 provides an example of the power–performance evaluations carried out to resolve design tradeoffs. Final comments and conclusions are given in Section 10. The formal derivation of the power–performance optimization methodology and metric is given in the Appendix.

Unique research contributions arising from the eLite DSP include more than just the architecture and the processor. Other aspects are the power–performance

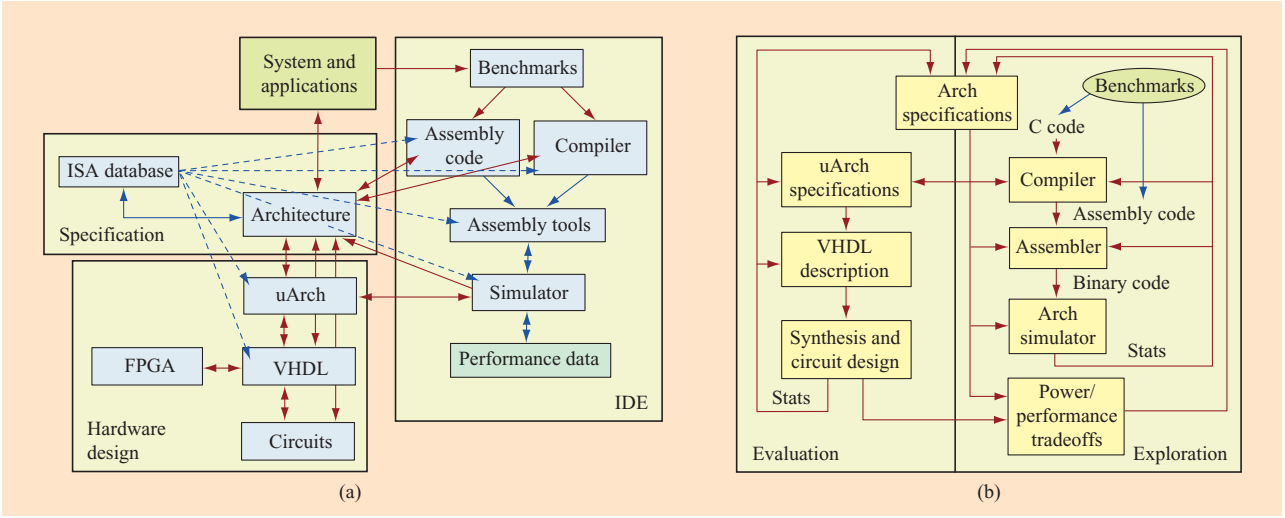

Figure 1

Co-development interactions and design methodology. FPGA = field-programmable gate array.

optimization methodology at the architecture level [12, 13]; new circuit techniques for ultralow-power implementations, as described in [14]; important compiler optimizations for DSP operations; and system-level design experience.

2. Application/architecture/compiler/implementation co-design

The co-design and evaluation methodology used in the development of eLite has been built around the interaction among the multiple dimensions of the problem, as depicted in **Figure 1**. At the center of the interacting components [Figure 1(a)] lies a description of the architecture, represented by the instruction set architecture (ISA) database, which reflects the current view of the architecture at a given point in time. This database drives all of the components of the environment, which range from code generation and performance analysis [the integrated development environment, IDE, in Figure 1(a)], to analysis from logic implementation and circuit design (the block labeled "Hardware design" in the same figure).

The processes by which these components are exercised are illustrated in Figure 1(b). The exploration path is characterized by a fast turnaround time, wherein the focus of the evaluation is on instruction-set architecture performance measures and estimates of power–performance tradeoffs. The evaluation path is characterized by longer turnaround time but includes detailed analysis from hardware design and implementation. As their names imply, each path has a well-defined objective. The exploration path is used to evaluate proposed features and changes by modifying the different components of the environment as necessary and performing cycle-accurate simulation, although the simulation does not include all of the details of the implementation. In contrast, the evaluation path focuses on providing accurate performance and power consumption metrics, as obtained from detailed description of the hardware elements in an implementation.

A set of benchmarks has been used to optimize the eLite architecture according to a power–performance metric. This set was chosen on the basis of the following criteria:

- Relevance to the target applications, mainly wireless and line communication, voice applications, and media applications.
- Moderate-size functions.
- Coverage of the various units as well as system issues such as interrupts.

The set of benchmarks includes simple and very common functions such as finite impulse response (FIR) filters, infinite impulse response (IIR) filters, vector add, vector max, and control code. The benchmark suite also includes more complex functions such as fast Fourier transform (FFT), interpolator and decimator, inverse discrete cosine transform (2D-IDCT), and Viterbi decoder, among others. An instruction-set simulator was used for each version of the architecture to analyze the performance. Conclusions were made about areas in which

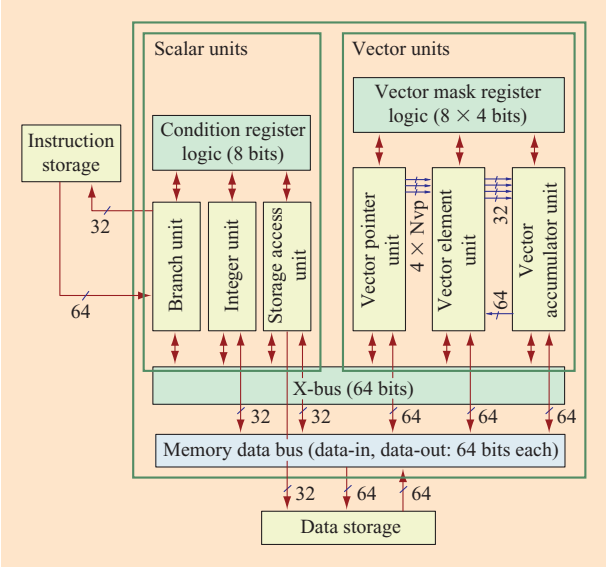

Figure 2

Block diagram of the eLite DSP architecture.

improvements to the architecture were needed. The decision on which solution should be adopted for a particular issue was made according to the power–performance methodology described later. This process was repeated for each version of the architecture, leading to significant performance improvement each time while maintaining a modest increase in power consumption and chip size.

3. Processor architecture

We now describe the major elements of the eLite DSP architecture, which is characterized by the following features:

- Exploitation of instruction-level parallelism through multiple independent instructions packed as long-instruction words (LIWs), exploitation of data parallelism through single-instruction multiple disjoint data on dynamically composed vectors (SIMdD operations), and exploitation of data parallelism through single-instruction multiple data on packed vectors (SIMpD operations).
- A heterogeneous clustered architecture, with specialized functional units and register files per cluster tuned to specific data types and computing requirements, with vector processing units interconnected in cascaded manner.
- A large number of internal registers, scalar and vector type, including a novel indirect register-addressing mechanism that enables the dynamic composition of vectors with four elements.
- Amenability as target for an optimizing compiler through an orthogonal instruction set based on explicit use of uniform register files, thereby enabling efficient programming in a high-level language.
- Suitability for low-power implementations through the replacement of run-time features by practices performed at code-generation time, in addition to well-known schemes such as shutting down or blocking the activity of unused registers/logic/functional units. Examples include static scheduling of instructions and static vectorization of operations, the use of predicated instructions, control of visible latencies (e.g., an "exposed pipeline" model), limited number of ports in register files, specialized processing units, constrained paths to functional units and memories, and other related features that are visible during code generation.

Because of its intended use as a target for an optimizing compiler and the reduction of hardware resources for the detection of hazards, the code executed in an eLite processor is expected to be generated mostly by an optimizing compiler. Such a compiler is required to ensure that the constraints imposed on the code are properly fulfilled. Code can also be generated and optimized at the assembly language level, of course, and tools that support such a level have also been developed. Assembly code can be mixed with code generated by a compiler.

Figure 2 is a logical representation of an eLite DSP processor, which consists of the following units:

- *Branch unit (BU):* Generates the storage address for the next long instruction to be fetched from memory, either sequential addressing or branches, and performs logic operations on eight single-bit condition registers.
- *Integer unit (IU):* Performs operations on data in 16 integer registers.
- *Storage access unit (AU):* Interacts with the data storage to transfer data between internal registers and storage, and performs operations on data in 16 address registers, which are used to address storage.
- *Vector pointer unit (VPU):* Performs operations on 16 vector pointer registers, which are used to access the contents of vector element registers.
- *Vector element unit (VEU):* Performs operations on data stored in vector element registers. The number of these registers is implementation-dependent, ranging from 64 to 4096.
- *Vector accumulator unit (VAU):* Performs operations on data in 16 vector accumulator registers, including reduction operations on the elements of a vector.

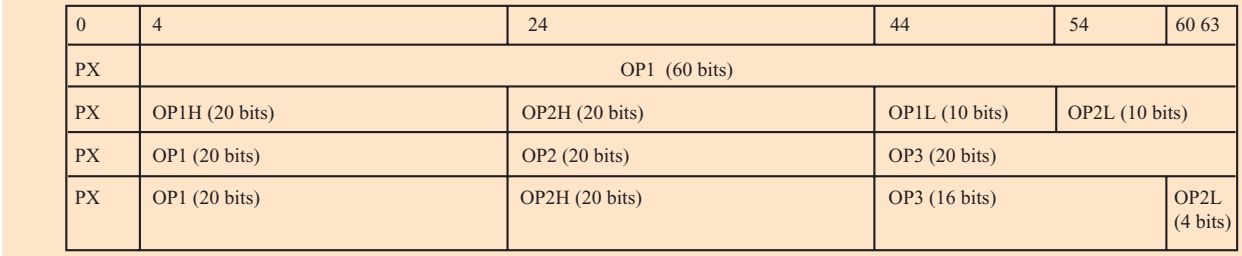

Figure 3

Long-instruction-word (LIW) formats.

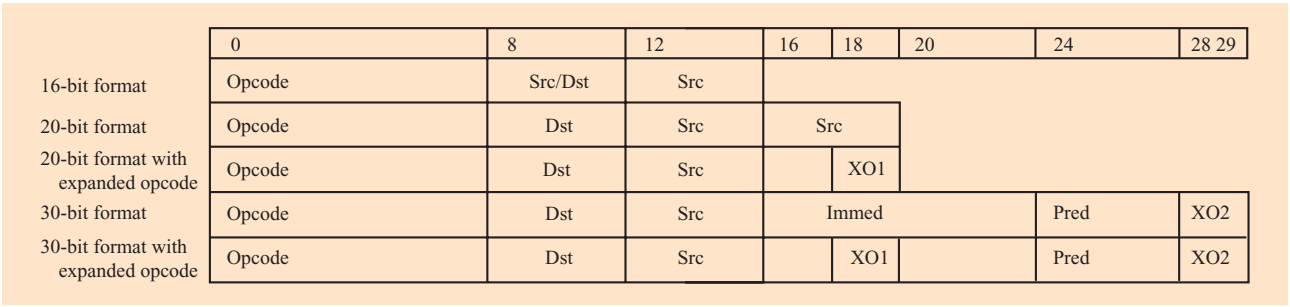

Figure 4

Instruction format.

The integer unit and storage access unit correspond to scalar units, operating on 32-bit integer data. In contrast, the vector element unit and the vector accumulator unit operate on four-element vectors in SIMD fashion (16-bit and 40-bit, respectively), containing fractional or integer data.

Program execution

A program in the eLite DSP architecture consists of a sequence of long-instruction words, each containing a four-bit prefix (PX) and one, two, or three instructions, as depicted in **Figure 3**. A long instruction is the minimum unit of program addressing possible, represented in memory as a 64-bit entity. Branching into an instruction other than the first instruction of a LIW is not possible. A processor fetches LIWs from instruction storage for execution; the instructions contained in a LIW are dispatched for either simultaneous or serial execution, as specified in the LIW prefix.

All instructions, regardless of their length, contain a fixed-size opcode in bits 0:7 specifying the operation to be performed. Some instructions specify an expanded opcode field in bits 18:19. Instructions of 30-bit length specify additional opcode information in bits 28:29. These formats are illustrated in **Figure 4**.

As in RISC processors, no instruction in the eLite architecture can perform a computational operation on data in memory, and no instruction other than store instructions can modify storage. To use a storage operand in a computation, the contents of storage must first be loaded into a register, and the operation is performed on the contents of the register. Similarly, to use a storage operand in a computation and then modify the same or another storage location, the contents of storage must be loaded into a register, modified, and then stored back to the target location. Direct memory access (DMA) operations may alter the storage contents independently.

The preferred programming model (**Figure 5**) consists of loading many data elements from storage into the registers, in particular into the vector element registers (VERs) and vector accumulator registers (VARs), and then operating on the contents of the registers with few intervening accesses to storage. VERs are accessed indirectly through vector pointer registers (VPRs), so that vectors are dynamically composed from four

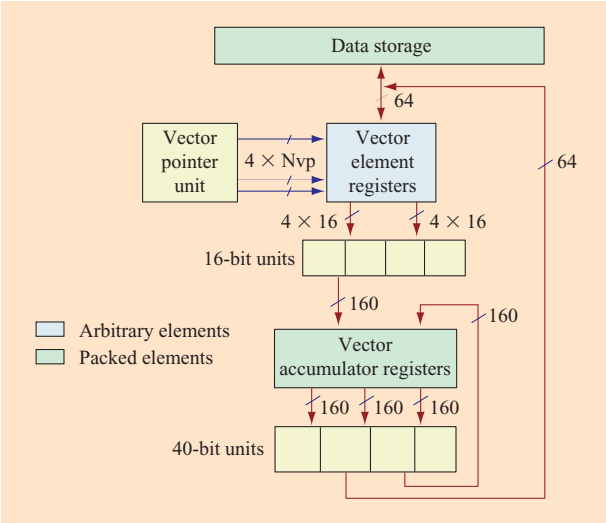

Figure 5

Vector programming model.

arbitrary VERs (SIMdD operation). Every vector element instruction specifies one or two VPRs, each of which in turn specifies four VERs used by the instruction. In contrast, four sets of VARs, each containing 40-bit elements, are used as operands to vector accumulator unit instructions (SIMpD operation). The elements from VARs, in conjunction with a reduction register, are also used as operands to a special reduction unit.

Vector units are characterized by a cascaded SIMD programming model: 16-bit data is loaded from adjacent memory locations into arbitrary locations in the VER file, 16-bit operations are performed on data from arbitrary locations within the VER (SIMdD), and the results are placed in the 4 × 40-bit VAR file, in a packed manner (in a single register). Also, 32-bit data is loaded directly from memory, in packed form, into the VAR, operations are performed on packed data read from VAR (SIMpD), and the results are placed in the same register file. Packed data can be transferred from the VAR file into the 16-bit VER file with arbitrary placement, after a suitable size-reduction operation, or it can be placed in adjacent memory locations. Because of the varying size, data is allocated to units according to the natural data type (size) throughout the computations.

Instructions are statically scheduled, taking into consideration their utilization of resources throughout the pipeline and the data dependencies with their dependent instructions (e.g., the "exposed pipeline" execution model). The execution pipelines are presented in **Table 2**. Most instructions are processed in five stages, but vector element instructions use one extra stage to read the VPRs and the succeeding stage to read the VERs, whereas memory instructions use dedicated stages for transferring the address and data from the processor to the memory subsystem. All instructions that are dispatched in the same cycle read the contents of the source registers at the same time, with the exception of VERs, which are read in the following cycle after reading the associated VPRs. An instruction completes execution when its results are placed in the destination registers; instructions complete execution according to their individual latencies. Instructions contained wholly within a functional unit (FU) have the same latency, with the exception of branches that are resolved earlier; instructions in different units, or instructions that place the result on a register in a different unit, may exhibit different latencies.

Instructions other than vector instructions can be predicated by specifying a condition that is evaluated dynamically, at execution time. The predicate is specified in a condition register. An instruction whose predicate evaluates to false is not completed; such an instruction is simply discarded. Vector instructions are not predicated as a whole; instead, each individual operation within a vector instruction can be executed conditionally (i.e., predicated) under control of a mask which is evaluated dynamically. The mask is specified in a vector mask register (VMR).

Vector units

The most salient computing resources within the eLite architecture are the three types of vector units, depicted in **Figure 6**. All of these units can operate in parallel.

Source operands for 16-bit vector element arithmetic and logic instructions (16-bit datapath) always originate from the VER file. The destination of all 16-bit vector

Table 2 Execution pipelines.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Base	IF	DEC	RD	EX	WR			
Other unit target	IF	DEC	RD	EX	XFR	WR		
Vector element instructions	IF	DEC	RD_VP	RD_VE	EX1	EX2	WR	
Load instructions	IF	DEC	RD	AG	XFR1	RD_M	XFR2	WR

element operations is a VAR (40-bit result). Each slice within the 16-bit datapath consists of a multiplier and an ALU that performs arithmetic, logic, shift, and select operations on the contents of VERs.

Every access to the VER file is performed indirectly through a VPR. Each VPR contains four elements which are used as indexes to the VER file, allowing access to four independent VERs. The VPRs can be automatically updated when used to access VERs, and can automatically implement "circular addressing" within a range of the VER file.

VARs are used as source and destination for 40-bit vector accumulator arithmetic and logic instructions (40-bit datapath). VARs are also used as destinations for 16-bit vector operations, as well as source operands in instructions to convert data from 40-bit to 16-bit whose result is placed in VERs. Regardless of the instruction type, VARs are accessed as 40-bit quantities. In the case of conversion to 16-bit, saturation and rounding rules can be applied.

Since a LIW may contain up to three instructions, the architecture supports "3-wide" instruction-level parallelism. Some of these instructions are compounded instructions which specify more than one operation. Moreover, vector instructions specify operations on vectors with four elements. As a result, the total parallel computing capability available in a single LIW is a large number of basic operations. For example,

- A single LIW may contain a *load vector with update* instruction, a *vector element multiply* instruction, and a *vector accumulator add* instruction, with each one of them producing as a result a vector with four elements; that is, four operations each, for a total of twelve operations.
- The *load vector with update* instruction implicitly specifies the automatic update of the address register and the elements of the VPR used by the instruction, including support for circular addressing on both, adding another five operations to the set performed within the LIW.
- The *vector multiply* instruction implicitly specifies the update of the two VPRs used to access the VERs containing the operands for the instruction, including circular addressing within the VER file, adding two sets of four update operations to the computations specified in the LIW.

Consequently, such a single LIW specifies transformations on 25 data items, for a total of 25 basic operations. These operations comprise one of the most common functions in signal processing—a convolution.

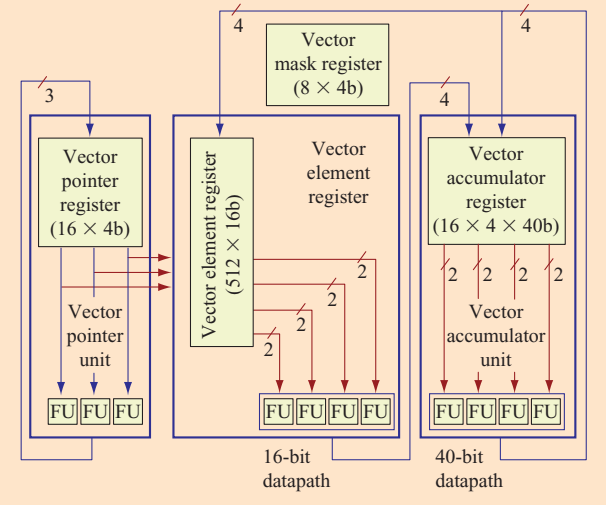

Figure 6

Vector units.

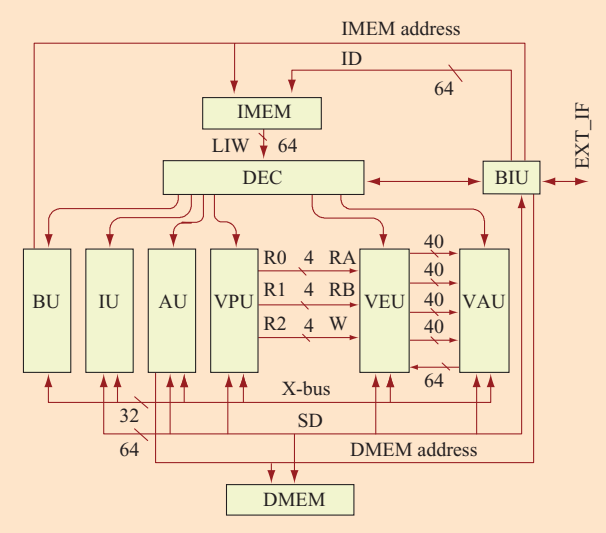

Figure 7

Block diagram of eLite implementation.

4. Hardware design

A block diagram of a prototype implementation of eLite is shown in **Figure 7**, depicting the scalar functional units (BU, IU, AU) and vector functional units (VPU, VEU, VAU). As already stated, each functional unit houses its corresponding register file. A shared 64-bit bus, called the X-bus, connects all functional units, allowing data movement among the various register files.

In Figure 7, instructions flow from top to bottom. The on-chip instruction memory (IMEM) holds the long-instruction words, each 64 bits wide. A LIW is decoded every cycle; the prefix (PX) field indicates how to interpret the LIW. The information in the prefix field includes parallel versus serialized execution, as well as the number and length of the individual instructions packed within the LIW. The decoder interprets and dispatches the individual instructions to their respective functional units along with the specified operands. In the following pipeline stages, each functional unit that received a valid instruction reads the operands from the register file in the corresponding unit, performs the required function, and writes the results to the destination register file.

Access to the data memory (DMEM) is accomplished through a common 64-bit-wide data bus (SD-bus). All functional units (except BU) can read data from or write data to the data memory. Memory addresses for all load-and-store instructions are generated in the AU. Scalar functional units connect to only the least significant half of the SD bus, whereas the vector functional units use all 64 bits of the bus.

In addition to the common buses, the vector functional units use several point-to-point connections to communicate among themselves. In particular, the VPU sends three sets of four indices to the read and write ports of the vector element file in the VEU. Furthermore, the VEU sends its execution results over four connections, each 40 bits wide, to be placed in the VAR file. Moving data back from the vector accumulator file to the vector element file in the VEU takes place on another dedicated 64-bit-wide connection, as shown in Figure 7. The aforementioned point-to-point connections are used frequently enough, and have enough of an impact on the overall performance, to justify adding them as separate connections.

The bus interface unit (BIU) handles the communications with the external world, including loading instructions into the internal memory, transferring data to and from the data memory, reading various processor state information, and handling external interrupt requests. The decoder (DEC) handles the arbitration between the internal units and the BIU for the data memory accesses, and the arbitration between its own requests and the BIU's requests for accessing the instruction memory.

5. Optimizing compiler

The eLite DSP compiler has evolved jointly with the eLite DSP architecture, guided by the performance evaluation of characteristic benchmarks in the intended application domain. Our primary goals are

1. To develop a compiler that generates efficient code for exploiting the data-level and instruction-level parallelism capabilities in the processor.
2. To make the processor programmable in the C language without resorting to any architecture-specific language extensions.

In this section, we describe some of the important issues that we had to address while designing the optimizing compiler for the eLite DSP architecture.

DSP compilation: Challenges and solutions

The basic data type in DSPs for digital communications is a saturating fractional fixed-point representation, whereas C-language constructs define integer modulo arithmetic. The traditional approach taken by compilers to deal with this mismatch between DSP data types and C-language constructs is based on the use of *intrinsics* [15, 16] and/or *C-language extensions* [17, 18]. Intrinsics allow a programmer to explicitly specify certain instructions in the architecture which cannot be easily described in a high-level language such as C. The use of intrinsics and C-language extensions suffers from disadvantages such as nonportable code which cannot be emulated easily on multiple platforms, the use of saturating data types, multiple memory spaces, and the need for explicit specification of data parallelism. While intrinsics can help in reducing complexity by providing "hints" to the instruction-selection process performed by a compiler, we believe that this is a step backward with respect to the use of portable high-level languages to program DSPs.

DSP applications and architectures are getting larger and more complex. We believe that DSP applications in the future will be programmed in standard high-level languages such as C, akin to writing programs for modern RISC microprocessors; consequently, DSPs should require a minimal amount of assembly language programming or intrinsic libraries. Our approach to this problem is based on a novel technique we refer to as *semantics analysis*, which essentially tries to search for and infer the meaning of sequences of C-language constructs used for typical DSP functions such as saturated arithmetic or circular addressing. The programmer should follow a few basic guidelines in order to simplify this inference process, enabling the compiler to generate code using the specialized DSP primitives available in the architecture.

The compiler also has to recognize code sequences that can be vectorized in order to efficiently exploit data parallelism in the vector units. It must generate efficient code by minimizing the movement of data between functionally partitioned register files. The compiler must also address issues such as dealing with 40-bit accumulators, circular buffers, explicit bypasses,

exposed pipeline latencies, delayed branches, and pipeline resource hazards.

Another important issue in a compiler for a DSP is the size of the resulting code, in view of the relatively small on-chip memory available in embedded processors. Since the eLite DSP architecture has a short LIW, there is no need for speculative code motion to fill instruction slots. Moreover, its serialization capabilities reduce the need to pad LIWs with no-op instructions. To increase code density even further, the compiler leverages architectural features such as SIMD instructions, and combines instructions of different widths.

Compiler implementation

The eLite DSP compiler is based on the IBM VLIW Research Compiler originally designed for tree-VLIW processors [19, 20]. This compiler uses an enhanced form of dependence flow graph (DFG) [21] for its internal representation, and also extensively uses static single assignment (SSA) [22] and reverse-SSA forms. It has a repertoire of standard SSA-based optimizations, and provides a rich collection of functions/macros for compiler development. Among other features, the compiler provides an excellent platform for developing and exploring new compilation techniques. (See [23, 24] for more details on the IBM VLIW Research Compiler.)

New optimizations and enhancements were introduced to the compiler in order to generate efficient code for the eLite DSP architecture. **Figure 8** shows the structure of the compiler. After transforming the dependence-flow graph through a number of optimizations, a novel vectorization phase attempts to identify vectorizable code sequences and updates the DFG with a vectorized version of loops. Separating vectorization from instruction scheduling and register allocation has advantages such as reducing the complexity of the code generator, and offering flexibility in scheduling and software-pipelining vectorized code and scalar code. The vectorized code is scheduled and register-allocated before the final assembly code is emitted.

The compiler also makes use of predication to collapse small blocks of conditionally executed code, thus eliminating branches and their overhead. Among other optimizations, the compiler does function in-lining, software-pipelining, synthetic branch-frequency-based optimizations, and interprocedural analysis.

Instruction selection

The internal representation of the compiler uses primitive operations similar to those found in RISC processors. Optimizations prior to scheduling and register allocation are performed on a DFG with such operations as nodes. The instruction selector provides a binding between such RISC operations and instructions in the eLite architecture.

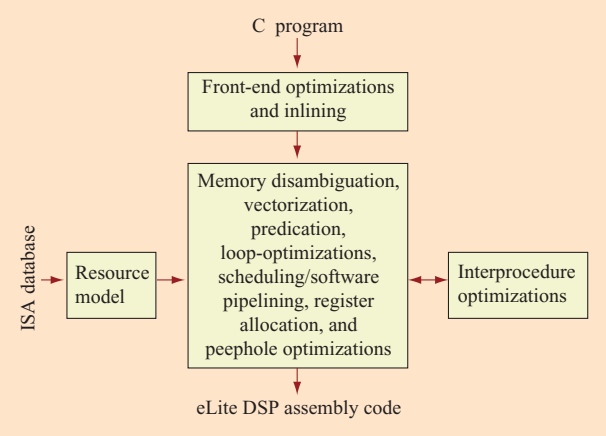

Figure 8

Block diagram of the eLite DSP compiler.

Instruction selection is carried out along with instruction scheduling based on a number of factors such as data type and size of operands, automatic update of registers, and resource requirements.

Vectorization

A vectorization phase has been developed [25] to make efficient use of the SIMD instructions and the unique VER file of the eLite DSP. The VER is modeled as a compiler-controlled memory which can be indirectly addressed using vector pointer registers. Because all VER allocations are done by the compiler, complete aliasing information for VER accesses is available. The compiler takes advantage of this fact by applying aggressive memory-related optimizations.

Our high-level vectorization scheme is similar to the unroll-and-jam technique [26]—the loop is unrolled by a factor of 4, and scalar operations are replaced with their SIMD versions. In some cases, the compiler performs additional optimizations, such as preloading data into the VER before the loop to eliminate redundant loads. An important stage of vectorization is setting up the vector pointers. Flexibility of VER access through vector pointers allows efficient compilation of computations that involve nonconsecutive data access patterns. This aggressive vectorization optimization is based on a set of innovative analyses, on top of the standard vectorization tests such as those presented in [27]. The compiler analyzes memory access patterns in order to effectively vectorize loads and stores within each iteration. *Loop context analysis* eliminates memory loads and stores across iterations and between different loops. Additional transformations which expose more parallelism, such as loop transformations [28, 29], are currently under development.

Functional partitioning of register files

The register name space in the eLite DSP architecture is functionally partitioned into multiple register files associated with respective functional units. This results in a set of register types such as the integer register (IR), address register (AR), vector element register (VER), and vector accumulator register (VAR). The compiler is responsible for assigning each SSA definition (Def) to a specific register file/type, and for inserting explicit instructions moving the data between different register files. Given an SSA Def, the operation defining it, and the operations using it, the choice of a register file is guided by the following considerations:

1. Availability of operations in different functional units.
2. Performance issues, such as pipeline latencies of instructions or possibilities of VER reuse.

The partitioning process based on the above features is augmented with an efficient min-cut graph-partitioning-based scheme that minimizes the communication between the register files in order to reduce the number of move instructions [30].

Resource modeling

The primary goal of the resource model in a compiler is one of providing an abstract view of the processor to the machine-dependent optimization routines. A cycle-accurate model of the processor has been developed to model pipeline resources, register file ports, and the interconnect bus. We have also developed some low-overhead techniques to model nonuniform port access delays resulting from microarchitecture techniques that reduce power and hardware complexity, such as restricted bypasses. In addition, the resource model provides internal-to-ISA instruction-mapping information for the instruction selector, resource conflict checking and reservation functions required during scheduling, and register Def/Use timing information required during register allocation. The entire resource model generation is automated and table-driven, based on machine-generated architectural descriptions shared by other tools, which also helps reduce the turnaround time for architectural exploration.

Scheduling and register allocation

Generating efficient code for long noninterlocked pipelines is a considerable challenge in itself. When scheduling instructions, the compiler must have accurate timing information about the access to data (read and write) and to the shared resources used by the instructions. Conservative bounds are used while compiling certain regions of code (e.g., across calls) where complete timing information is not available, which in turn prevent the compiler from carrying out aggressive scheduling techniques, such as scheduling instructions in the shadow of other instructions, in those regions.

The code generator considers reducing the code size as an objective in addition to minimizing the schedule length. We use several new techniques for scheduling and cycle-level register allocation. These techniques include schemes for scheduling instructions in the branch delay slots and instruction shadows, for scheduling predicated code, and for software pipelining. The detailed description of these optimizations is beyond the scope of this paper.

Interprocedural optimizations

The compiler performs interprocedural analysis for reducing the calling convention overhead by allowing the callee to modify the calling conventions according to the intended usage of the parameters. Other interprocedural optimizations being developed include schemes for sharing VER among procedures and for obtaining better bounds on resource utilization at procedure calls and returns.

We believe that the eLite DSP compiler, with the help of its powerful optimizations and further tuning, can generate executable code from source code written in C, with performance and code size comparable to hand-optimized assembly code. Preliminary results show that, for a set of signal-processing kernels, the compiler generates vectorized code of performance comparable (well within a factor of 2) to that of carefully hand-coded assembly.

6. Simulation and performance evaluation environment

Software development is an important part of developing any processor. In this section, we describe the software development tools that have been tailored for the research on the eLite DSP architecture.

The architecture specification is maintained in the centralized ISA database. This database contains all of the information describing the architecture, including instruction formats, operands for each instruction, and a machine-readable pseudocode that describes the behavior of each instruction at each stage of the pipeline. All of the tools have been built as semigeneric programs, providing a framework that takes most of its specific details from a set of configuration files automatically generated from the ISA database. The contents of the database reflect the attributes and behavior of the architecture, and the tools automatically track any changes to the architecture.

Binary code generation

The binary form of an instruction is a sequence of bits composed of a concatenation of constants and the binary encoding of the instruction operands (parameters). The instructions are bundled together to make long-instruction

words (LIWs). Finally, the sequence of LIWs is the output program code. The way bits are arranged in an instruction or LIW is often nontrivial, requiring a mechanism to insert (when assembling) and extract (when disassembling) information from the instruction or LIW. Since all of the details regarding the position of the fields are described in the ISA database, it is possible to automatically generate code to handle these insertions and extractions. For this purpose, the concept of *inserters* has been developed. An inserter is an abstract interface that provides the two basic operations *insert* and *extract*. Given a location within the program (e.g., LIW offset) and a value, the inserter places the value in the binary code at the given location. This concept is useful for the basic case of inserting the value of an operand into an instruction, up to inserting a whole instruction into the code section of the output program. Inserters are also useful for late binding of external symbols at link time.

The instruction syntax inside the tools is based on a free-form format string containing operand fields and any delimiting text. This approach permits custom formats to be set for various instructions without any modification to the assembler source. For example, an integer add instruction configuration is as follows: {"iadd", "OP=0×10", "$(RT),$(RA),$(RB)"}. A typical use of this format would be something like `iadd r1,r2,r3`. However, it can easily be changed to `iadd r1=r2+r3` by simply replacing the commas in the format string with the appropriate symbol, such as "$(RT)=$(RA)+$(RB)".

Conflict detection

The eLite DSP architecture has an exposed pipeline, thereby assigning to the compiler/programmer the responsibility of resolving data dependencies and resource conflicts in the program. The presence of instruction-level parallelism, combined with the pipeline latencies, makes this task quite difficult for the assembly-level programmer. A special tool was developed to help in this domain; this tool scans the assembly code, detecting data dependencies and machine resource conflicts. The tool is integrated into the development environment user interface, allowing the assembly-level programmer to visualize the conflicts in the source code.

Instruction set architecture simulator

Just like the machine code generation tools, the instruction set architecture (ISA) simulator is closely connected to the ISA database. Each instruction in the database contains a formal (machine-readable) definition of the behavior of the instruction, and the simulator guarantees that this behavior is kept and simulation results are consistent with expected results. As in the case of the operands in the assembler, a preprocessing program goes over the concise behavior information of the instructions and produces source code that is compiled into the simulator.

The instruction behavior description contains a local state for the instruction and a set of events describing what the instruction does at the different stages of the pipeline. The ISA simulator has an additional state for each instruction instance used for storing local temporary variables. This allows the instructions to be built in a modular fashion. There is, of course, a shared state of the machine, in the form of register files and memory. This state is accessed indirectly via a set of functions representing the hardware ports used to access these resources in the hardware implementation.

The event behavior code of the instruction is source code (C++ in this case) that operates on the local variables and the resource functions (ports) to perform its task. An event can be either a hardware-related event, such as performing the operations that are associated with a specific cycle in the execution of the instruction, or a software event artificially added to the ISA database as a function to help it perform its duties.

An example of the implementation of a simple instruction in the simulator is as follows:

```
[iadd]
      Int32 t,a,b;
3:    Ira(RA,a);
      Irb(RB,b);
4:    t=a+b;
5:    Iwp(RT,t);
```

The first line contains the instruction mnemonic in square brackets. The second line describes the local state of the instruction as a set of three 32-bit integers. Each of the numbers followed by a colon indicates that the following lines of code are associated with the execution cycle indicated by the number. In this example, both source operands are read in cycle 3 into local variables, using the integer register file ports `Ira, Irb.` The operands are added in cycle 4, and the result is written into the register file in cycle 5 through the port `Iwp.`

During run-time, the simulator loads the binary code of the application program. It then decodes instructions according to the ISA specification. The instruction code, as imported from the database, is then run one cycle at a time, allowing the instruction to perform its duties. Since the simulator implements the pipeline stages, there are normally several instructions in flight, each in its appropriate stage.

Instructions are not the only part of the ISA simulator that contains automatically generated code. The machine state, in the form of register files and memory banks, is also specified in a concise manner in a configuration file; source code is generated from the file using a preprocessing program. Read and write ports that access the various register files are also created in a similar fashion.

7. Programming examples

We now provide two examples of kernels from representative applications in the digital communications domain coded for the eLite DSP architecture.

Block FIR

The block finite impulse response (FIR) filter performs filtering of speech signals in modern voice coders such as the ETSI GSM EFR/AMR or ITU G.729 [31–33]. FIR filters are also used in many other signal processing areas, such as communications and echo-cancellation applications, to name just a few.

In the FIR algorithm, the filter coefficients are denoted $h(m)$, for $m = 0, \cdots, M - 1$, where M denotes the filter length. Typical values for M are 10 to 16 for voice-coding applications, and several hundreds or more in echo-cancellation applications. The input sequence is denoted $x(n)$, and the output sequence is denoted $y(n)$. The mathematical relationship among these signals in the time domain is

$$y(n) = \sum_{i=0}^{M-1} h(i)x(n-i).$$

Usually the output $y(n)$ is needed for several values of n, so several outputs may be computed in parallel. The number of outputs computed together is called "frame size," which is denoted by the symbol N. Typical values of N are 40 to 60 in voice-coding applications, and several hundreds in echo-cancellation applications.

To exploit the SIMD nature of the eLite architecture, several outputs are computed in parallel, resulting in the following expressions:

$$y(n) = (h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots + h(M-1)x(n-(M-1))),$$

$$y(n+1) = (h(0)x(n+1) + h(1)x(n) + h(2)x(n-1) + \cdots + h(M-1)x(n+1-(M-1))),$$

$$y(n+2) = (h(0)x(n+2) + h(1)x(n+1) + h(2)x(n) + \cdots + h(M-1)x(n+2-(M-1))),$$

$$y(n+3) = (h(0)x(n+3) + h(1)x(n+2) + h(2)x(n+1) + \cdots + h(M-1)x(n+3-(M-1))),$$

There are multiple alternatives for writing the code that computes FIR. Here we describe the case in which data is preloaded into the VER file (assuming that the file is large enough to hold the data). That is, we assume that the coefficients and input samples are loaded into the VER file prior to their use.

As can be seen from the expressions above, each filter coefficient is used in the four equations at the same place in the summation. Therefore, a vector pointer in which all of the elements point to the same entry in the VER file and are incremented by 1 after each use is suitable for addressing the filter coefficients.

For the input data, each sample is used in the four equations at a different place in the summation. Therefore, a vector pointer in which all of the elements point to consecutive entries in the VER file and are incremented by 1 after each use is suitable for addressing the samples.

At the end of the inner loop, the VPR used to access the filter coefficients must be rewound by $(M - 1)$ so that its elements will again point at $h(0)$ for the next iteration. Similarly, the VPR used to access the input data must be rewound by $(M - 1 + 4)$ so that its elements will point at $x(n + 4)$ through $x(n + 7)$ for the next iteration.

Unrolling is applied to the loop; this alternative achieves higher performance, albeit with the penalty of larger code size. The resulting assembly code for one iteration of the unrolled loop is shown in **Figure 9**; this implementation of the block FIR algorithm achieves asymptotically optimal performance (that is, four multiply/accumulate operations per cycle).

Vectorized Viterbi butterfly processing

An important application of the Viterbi algorithm [34, 35] is maximum-likelihood decoding of convolutional codes, which are employed for data transmission in many communications standards. We now demonstrate the efficiency of the eLite architecture for decoding rate $1/n$ binary convolutional codes, assuming binary antipodal signaling and a memoryless additive white Gaussian noise (AWGN) channel [35].

It is well known that a section of a trellis for a rate $1/n$ binary convolutional code can be decomposed into subgraphs called Viterbi butterflies [36]. The corresponding add–compare–select (ACS) operations [34–36] are referred to simply as butterfly operations. A straightforward implementation of the butterfly operations requires two memory buffers for holding state-dependent data, which are used in a ping-pong fashion [37].

By using the VPRs in eLite, a more sophisticated approach based on rotated metric indexing [36] can be used, allowing in-place metric updating. Compared to approaches using ping-pong buffers, $M = 2^m$ vector elements are saved, where M is the number of Viterbi decoder states. It can be shown that in-place butterfly operations are easily vectorizable. With our approach, a 512-element VER file is large enough to hold the state metrics and branch metrics for the $M = 256$-state 3GPP Viterbi decoder [38], eliminating power-consuming transfers of metric data from/to memory.

Figure 10 shows a fragment of C code that processes four butterflies in parallel using in-place metric updating;

```
          vemul va0, (vp2),(vp3)
          vemul va11,(vp2),(vp3)
          vemul va12,(vp2),(vp3)
          vemul va13,(vp2),(vp3) || mfictr ct0,r0      # inner loop counter

        filter.outer.loop:
        filter.inner.loop:
          vemul va14,(vp2),(vp3) || vaadd va0,va0,va11
          vemul va11,(vp2),(vp3) || vaadd va0,va0,va12 || bctnz ct0,filter.inner.loop
          vemul va12,(vp2),(vp3) || vaadd va0,va0,va13
          vemul va13,(vp2),(vp3) || vaadd va0,va0,va14

          vemul va14,(vp2),(vp3) || vaadd va0,va0,va11
          vemul va11,(vp2),(vp3) || vaadd va0,va0,va12
          vemul va12,(vp2),(vp3) || vaadd va0,va0,va13
          vemul.u va13,(vp2),(vp3),2,3 || vaadd va0,va0,va14 # rewind pointers

          vemul va0, (vp2),(vp3) || vaadd va1,va0,va11
          vemul va11,(vp2),(vp3) || vaadd va1,va1,va12 || bct 0, ct1,filter.outer.loop
          vemul va12,(vp2),(vp3) || vaadd va1,va1,va13 || mfictr ct0,r0
                                                              # inner loop counter
          vemul va13,(vp2),(vp3) || stvahu.i va1,8(a2)
```

Figure 9

Sample implementation of FIR filter.

```
p = i<<3;                          /* p = current state = 0, 8, 16, ... , M-8 */
s = i<<2;                          /* s = next state    = 0, 4,  8, ... , M/2-4 */

vi0[0] = p+0; vi0[1] = p+2; vi0[2] = p+4; vi0[3] = p+6;
vi1[0] = p+1; vi1[1] = p+3; vi1[2] = p+5; vi1[3] = p+7;

for (n=0; n<4; n++) vi0[n]  = rotlm(vi0[n], a, m);       /* a=mod(t,m) */
for (n=0; n<4; n++) vi1[n]  = rotlm(vi1[n], a, m);
for (n=0; n<4; n++) vi10[n] = bmi_table[i][n];
for (n=0; n<4; n++) vi11[n] = bmic_table[i][n];

for (n=0; n<4; n++) va0[n]  = metric[vi0[n]] + branchMetric[vi10[n]];
for (n=0; n<4; n++) va1[n]  = metric[vi1[n]] + branchMetric[vi11[n]];
for (n=0; n<4; n++) va2[n]  = metric[vi0[n]] + branchMetric[vi11[n]];
for (n=0; n<4; n++) va3[n]  = metric[vi1[n]] + branchMetric[vi10[n]];

for (n=0; n<4; n++) {
  if ((__int16) (va0[n]-va1[n])>=0) { va4[n] = va0[n]; tbr0 <<= 1; }
  else                              { va4[n] = va1[n]; tbr0 = (tbr0<<1) | 1;}
}
for (n=0; n<4; n++) {
  if ((__int16) (va2[n]-va3[n])>=0) { va5[n] = va2[n]; tbr1 <<= 1; }
  else                              { va5[n] = va3[n]; tbr1 = (tbr1<<1) | 1;}
}

for (n=0; n<4; n++) = metric[vi0[n]] = (__int16)va4[n];
for (n=0; n<4; n++) = metric[vi1[n]] = (__int16)va5[n];
```

Figure 10

Processing four butterflies with in-place metric updating in C.

```
...
veaddn va0,(vp0),(vp10)                                  ||  ...
veaddn va1,(vp1),(vp11)
veaddn va3,(vp1),(vp10)                                  ||  ldvpu vp10,8(a8)
veaddn va2,(vp0),(vp11)
veaddn va0,(vp2),(vp12)  ||  vtmax (vp0),va0,va1,vm3     ||  ldvpu vp11,8(a9)
veaddn va1,(vp3),(vp13)
veaddn va3,(vp3),(vp12)  ||  vtmax (vp1),va2,va3,vm7     ||  ...
veaddn va2,(vp2),(vp13)  ||  vmshl
veaddn va0,(vp0),(vp10)  ||  vtmax (vp2),va0,va1,vm3
veaddn va1,(vp1),(vp11)
veaddn va3,(vp1),(vp10)  ||  vtmax (vp3),va2,va3,vm7
veaddn va2,(vp0),(vp11)  ||  vmshl
...
```

Figure 11

Software-pipelined version of Viterbi decoder kernel.

this code nicely matches the SIMD features of the eLite DSP architecture.

After adding the precomputed branch metrics, metric comparisons are performed to select the survivor metrics and the corresponding trace-back bits. Four trace-back bits are shifted into trace-back registers tbr0 and tbr1 for each group of four butterflies. The trace-back registers are stored to memory whenever they are filled with valid trace-back bits (not shown in Figure 10).

The Viterbi decoder kernel in assembly language is shown in **Figure 11**. Each color represents a basic Viterbi decoder kernel, defined as a set of instructions processing four butterflies in parallel.

The M-element state metric array is allocated in a VER and aligned on an M-element boundary. The branch metric array is also allocated in a VER. The vector indices vi0, vi1, vi10, and vi11 in the C code correspond to vector pointers vp0, vp1, vp10, and vp11 (in the first basic Viterbi decoder kernel), which are reused several times after setup. Each basic Viterbi decoder kernel uses four vector element instructions (*veaddn*), two vector accumulator instructions (*vtmax*), and a vector mask instruction (*vmshl*). The incremented metrics are temporarily stored in the VAR. Of the four *veaddn* instructions, the two instructions targeting VARs va0 and va1 (or va2 and va3) perform the metric additions for eight branches that are merging in four states. Each *vtmax* instruction performs a selection of four survivor metrics, which are written to the VER for in-place metric updating, and outputs a four-bit result indicating the selection to vector mask register vm3 or vm7. The *vmshl* instruction shifts the eight vector mask registers to the left by one register. As a result, vm0 to vm3 and vm4 to vm7 are used to hold the trace-back data in the C variables tbr0 and tbr1, respectively. The rotate amount $a = \mathrm{mod}(t, m)$ has been preloaded into scalar register sr0, which is used by instruction *vtmax* for auto-updating the target vector pointer.

The vector element and vector accumulator units are used in tandem, with a continuous flow of 16-bit metric data from the VER to the VAR and back to the VER. Due to the instruction-level parallelism in the architecture, software pipelining can be used to start a second basic kernel while the *vtmax* instructions of the first basic kernel are being executed. As shown in Figure 11, tiling of two threads permits a basic kernel to be started every four cycles, resulting in a one-cycle-per-butterfly performance. Note that register usage repeats after each pair of threads. Figure 11 also shows vector pointer initialization from tables in memory that are pointing to precomputed branch metrics in the VER.

8. Methodology for power–performance tradeoffs

We now describe the methodology and metric developed for carrying out power–performance tradeoffs in the design of the eLite DSP architecture [12, 13].

We consider the problem of optimizing the power–performance characteristics of a processor in a space of three variables: architectural complexity, hardware intensity, and power-supply voltage.

To allow a mathematical analysis of the problem, we introduce an independent discrete variable ξ that represents a measure of the *architectural complexity* of a processor. The domain of this variable can be defined by ordering all possible processor configurations and assigning a numeric value to each of them. Then, any architectural alternative results in an increment or decrement in the value of ξ. Examples of variations in

Table 3 Processor performance and power characteristics.

Dynamic instruction count	$N = N(\xi)$	Total number of dynamic instructions executed on a given benchmark suite
Architectural speed (IPC)	$I = I(\xi)$	Average number of instructions completed per clock cycle on the same benchmark suite
Energy per instruction	$E = E(\xi, \eta, v)$	Average energy per instruction completed on the same benchmark suite
Maximum clocking rate	$f = f(\xi, \eta, v)$	Clock frequency

architectural complexity include the addition of instructions to the architecture, or modifying the definitions of existing instructions; at the microarchitecture level, these include changing the pipeline latency, adding or removing hardware functionality such as bypasses, functional units, read or write ports to access various structures, changing the width of the datapath, and so on.

The second independent variable in the optimization problem is *hardware intensity* η, introduced in [13] as a quantitative measure of how aggressively the circuits in a processor are tuned to meet a target clock frequency. Hardware intensity shows the energy cost (percentagewise) required to improve the delay of a circuit by 1% through restructuring and retuning the circuit. Although the values of hardware intensity in different pipeline stages are different, the aggregate hardware intensity—calculated as an energy-weighted average over all pipeline stages—can be used to represent the whole processor pipeline under optimal tuning conditions [13]. In this paper, all references to hardware intensity imply such an aggregate value.

Power-supply voltage v is the third independent variable in the optimization process; this is based on the assumption that, to achieve the desired power and performance characteristics, the power-supply voltage can be set to any value within the range for which a technology is qualified.

Then, the performance and power characteristics of a processor can be viewed as functions of the independent variables ξ, η, and v, where v and η are continuous and ξ is a discrete variable, as shown in **Table 3**. The average energy per instruction completed is calculated as $E = \Sigma_i w_i E_i$, where E_i is the average energy dissipated on the execution of instruction i, and w_i is the normalized dynamic frequency of the corresponding instruction in the benchmark suite. In fine-grain clock-gated designs, the use of average energy per instruction (rather than power) as a metric allows the energy dissipation to be decoupled from the architectural performance of a processor. Moreover, average energy per instruction appears as a term in the expression for *average* power dissipation, whose minimization is one of the primary goals in the design of the eLite DSP architecture.

To a first approximation, N and I depend only on the architectural complexity ξ and are independent of the hardware intensity and supply voltage. In contrast, the clocking rate f and the average energy per instruction E depend on ξ, η, and v.

The processor performance P (which is the inverse of execution time or of delay, D) on a given benchmark suite can be expressed as

$$P(\xi, \eta, v) = \frac{f(\xi, \eta, v)I(\xi)}{N(\xi)}.$$

Power dissipation $W(\xi, \eta, v)$ depends upon the implementation details of a processor. In particular, power dissipation is determined by the portion of the hardware covered by clock gating, granularity of clock gating, and speculative execution capabilities, if any. As described in Section 3, eLite is an "exposed pipeline" processor, so the overhead of clock gating is very low because all necessary control signals are generated at the instruction decode stage and propagated down the pipeline to achieve the stage-by-stage clock gating. Then, assuming an ideal clock-gating model, the only resources that dissipate power are those accessed by the instructions executed, and all unused hardware is gated off (using the finest-grain clock-gating mechanism or some sort of transition barrier mechanism,[1] or a combination of both). In this case, the average power is directly proportional to the average number of instructions executed per cycle and the average energy dissipated per completed instruction,

$$W(\xi, \eta, v) = f(\xi, \eta, v)I(\xi)E(\xi, \eta, v),$$

where E is the average energy per executed instruction, as defined above.

The energy efficiency metric used in this work (see the Appendix) has been derived by formally solving

[1] Transition barriers are placed before functional units (FUs) to prevent signal switching when they are unused, without the overhead of duplicating operand latches.

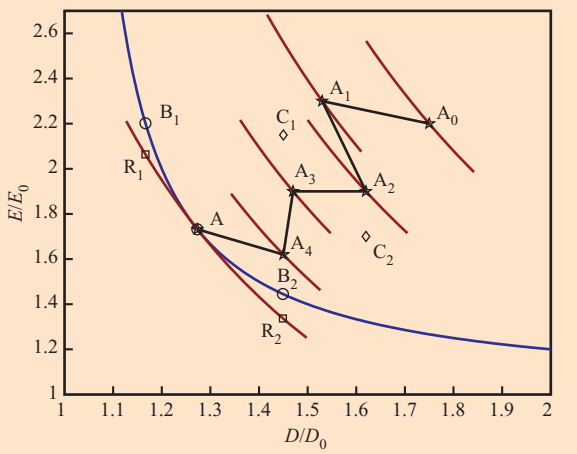

Figure 12

Exploring architectural energy–delay space.

the problem of minimizing W subject to a constant performance requirement P_0. The resulting formula is

$$\theta \frac{\Delta I}{I} > \frac{\Delta E}{E} + \Sigma \eta_i w_i \frac{\Delta D_i}{D} + (1 + \theta) \frac{\Delta N}{N},$$

where $\Delta I/I$, $\Delta E/E$, and $\Delta N/N$ are relative increments of the processor characteristics arising from an architecture modification $\Delta\xi = \xi_1 - \xi_0$, and $\Delta D_i/D_i$ are increments in critical path delays through stages i of the pipeline (can be positive, zero, or negative). The parameter θ is the energy–delay growth ratio that takes into account technology, power supply, and circuit style (library). The terms $\Delta I/I$ and $\Delta N/N$ can be measured by running the set of benchmarks on an architecture simulator. Terms $\Delta E/E$ and $\Delta D_i/D_i$ are estimated, assuming that *no* returning is done to compensate for a possible variation in frequency as a result of modifying the architecture.

We now provide an intuitive description of the power–performance optimization methodology, whose formal derivation is given in the Appendix. **Figure 12** shows a graphical interpretation of the process of refining the architecture of a processor in the energy–delay space. Each possible configuration ξ is represented by a point in the energy–delay space. Energy, plotted on the y-axis, is the average energy dissipated on a set of benchmarks, measured in joules. Delay, plotted on the x-axis, is the average execution time for the same set of benchmarks, measured in seconds. Thus, points closer to the coordinate origin represent more energy-efficient configurations. The blue curve represents the architecture energy–delay curve for fixed hardware intensity and power supply, defined as the optimal envelope of all feasible architecture/microarchitecture configurations. Each point on this curve represents a configuration ξ that delivers the maximum performance for a given power; alternatively, each point represents a configuration that achieves a given performance at the minimum power. The red curves in Figure 12 correspond to circuit energy–delay curves, each representing implementations of a given architecture ξ with varying power supply and circuit hardware intensity, such that the condition for the optimal hardware intensity $\eta = \theta$ is observed at each point [13]. As an example, the base hardware intensity is set to the value $\eta = 2$ for each circuit energy–delay curve in Figure 12. Thus, by fixing the architecture and microarchitecture, designers can move the energy–delay point of a processor along one of the red curves in the vicinity of the base point.

According to our methodology, only one point on the architecture energy–delay curve represents a design that achieves the optimal architecture/circuit energy–delay balance. This is the point at which the tangent to the architecture energy–delay curve has the same value as the tangent to the circuit energy–delay curve at the base hardware intensity and power supply (marked as point A in Figure 12). At this point, the processor delivers the required delay (performance) while dissipating the minimum power among all possible combinations of architecture configurations, circuit-level implementations, and power-supply voltages. To graphically demonstrate this statement, let us assume that the processor pipeline is designed for hardware intensity $\eta = 2$, and the initial configuration delivering a predefined delay (performance) $D = 1.16D_0$ is represented by point B_1 on the architecture energy–delay curve. Reducing the architectural complexity moves the processor energy–delay point along the blue curve from B_1 to A. Then, by raising the power supply and tuning the circuits for a higher value of hardware intensity, the design point can accordingly be moved along the red curve from A to R_1, thereby recovering the performance lost. Since the architecture energy–delay curve is steeper at point B_1 than the circuit energy–delay curve, R_1 is a better design than B_1 because it delivers the same performance at lower power. Notice that point R_1 represents the same architecture configuration as point A, just implemented with different circuits.

Similarly, if an original design point B_2 is below the optimal point A, increasing the architectural complexity allows the design point to be moved along the blue curve to point A. If the higher performance at design point A is not needed, the design point can be moved along the red curve to point R_2 by reducing the power supply and tuning the circuits accordingly for lower hardware intensity. Since the circuit energy–delay curve at this point is steeper than the architecture energy–delay curve, the resulting design

point R_2 delivers the same performance as B_2 but at lower power.

An advantage of using the derived formula is that it allows architecture alternatives to be compared without knowing the shapes of the energy–delay curves, even without knowing the exact position of the corresponding points in the energy–delay space.

In the development of a new microprocessor there is a high probability that an initial design point, marked as A_0 in Figure 12, is located above the architecture energy–delay curve. Transferring the initial design into a point on the architecture energy–delay curve is a challenge on its own, usually requiring several iterations. The path marked with black lines illustrates the process of architectural refinement using our methodology. Segments of red curves at each intermediate point (A_0, A_1, $\cdots$, A_4) in the optimization process represent circuit energy–delay curves for each iteration, drawn assuming a baseline hardware intensity and energy delay growth ratio $\eta = \theta = 2$. Using the same reasoning as presented above, since A_2 is below the circuit energy–delay curve passing through point A_1, it represents a more energy-efficient design than point A_1. Indeed, raising the power supply and tuning the circuits accordingly in the implementation of configuration A_2 leads to a design that delivers the same performance as A_1 but at lower power consumption. Similarly, every successive point A_i in Figure 12 represents a better design than the previous point A_{i-1}. On the other hand, since points C_1 and C_2 are less energy-efficient than A_3, A_3 is the point chosen at the end of the iteration initiated from point A_2.

A limitation of the methodology is that the relative differences in the characteristics of a processor, corresponding to alternatives for evaluation, must be small; a 10% limit can be used for most practical purposes. Special care would be needed to use the methodology for evaluating the energy efficiency of architectural features that result in significant changes in the characteristics of a processor, such as changing the issue width.[2] The selection of the initial design point is important for convergence to the optimal solution rather than to a local extremum. Thus, the methodology does not replace the experience and intuition of designers; instead, it provides help in evaluating various architectural configurations in the power–performance space.

The strength of our methodology is that it allows a designer to compare the energy efficiency of different design points without redesigning the implementations for every configuration in this iterative refinement process. However, designers' expertise is required to generate the various configurations to be evaluated using the methodology.

[2] *Issue width* is the maximum number of instructions that a processor can issue for execution in a single cycle.

9. Example of architecture/power/implementation tradeoffs

The energy-efficiency metric described in Section 8 and in the Appendix was consistently used in the process of refining the eLite DSP architecture. Most architectural features were carefully evaluated for energy efficiency before being committed to the architectural specification. Examples include reducing the number of pipeline stages; uniform bypasses on vector accumulator, integer, and address register files; hardware support for context savings at interrupts; adding ports to various register files; bypasses on the vector element file; changing functionality of several vector and accumulator instructions; and many other proposals. As a demonstration of the power–performance optimization methodology, we now describe the evaluation of the energy efficiency of alternative proposals for bypasses in the VER file.

Because of the number of read and write ports and the size of the VER file, the hardware complexity of bypasses is potentially significant, both in terms of power overhead and impact on the maximum clocking rate of the processor. On the other hand, the lack of bypasses increases the latency of the vector pipeline, which affects the architectural performance of the processor.

Several architectural alternatives for vector element file bypasses were evaluated using the methodology described in Section 8. Some of the alternatives considered were the following:

A. No bypasses.
B. Element-wise bypass. By labeling the read ports RA0, $\cdots$, RA3, RB0, $\cdots$, RB3, and the write ports W0, $\cdots$, W3, an address match is determined as a result of comparing the concatenated indices (RA0, $\cdots$, RA3) with (W0, $\cdots$, W3). In the case of a match, Wi is bypassed to RAi, for $i = 0, 1, 2, 3$; on the other hand, none of the elements is bypassed when there is no match.
C. Conventional full bypass.

Figure 13 shows proposed floorplans for these alternatives. Both read and write latches are physically placed on top of the corresponding vector datapaths. For the element-wise bypasses (alternative B), all bypass wires run vertically from inputs and outputs of the write-back latch, through the bypass multiplexor, to the inputs of the read latches. Thus, there are no wires running across the vector unit. Each bypass multiplexor (one for each read port) multiplexes data from three directions: the output of the array, the output of the write-back latch (write-through bypass) and the input to the write-back latch (execution bypass). Each read index is compared to two write indices (one in the write-back stage and one in the execution stage) of the write port to the same slice,

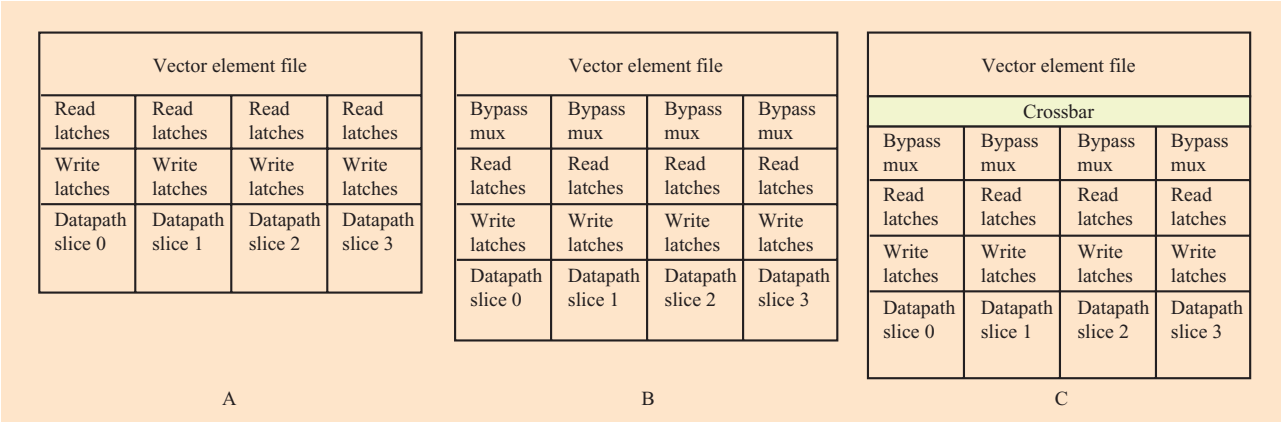

Figure 13

Alternative implementations of the vector element file bypass.

Table 4 Energy overhead of bypasses in the vector element file (no retuning).

Component	*Element-wise bypass* (fJ)		*Full bypass* (fJ)	
	Read access	*Write access*	*Read access*	*Write access*
Comparators	268	536	1072	2144
Bypass mux	69	284	69	1112
Crossbar	—	—	—	5226
Height increase	31	31	184	184
Total	368	851	1325	8666

treating the four indices as a concatenated number. Then, for a 512-entry register file, there are $2 * 2 = 4$ comparators of $9 * 4 = 36$ bits each (two comparators for each read port).

In the case of full bypasses (alternative C), data from any of the four write ports can be passed to any of the eight read ports, which requires a crossbar of $2 * 4 * 16 = 128$ lines. Each bypass multiplexor receives data from nine directions. A total of $8 * 4 * 2 = 64$ index comparators with 9 bits each are required to control the multiplexors.

Table 4 shows the major components of the energy overhead of the bypass implementations, assuming that no retuning is done to compensate for an increase in access time over the implementation with no bypass, for one read access (through four ports) and one write access (through four write ports). The average number of vector element file read and write accesses on the set of kernels used as a benchmark suite was measured to be 0.6 and 0.3, respectively. Then, assuming no retuning, the average energy overhead per instruction is 0.48 pJ and 3.6 pJ for the element-wise and full-bypass implementations, respectively. The average energy dissipated per instruction on the same set of kernels is estimated to be 50 pJ. Thus, assuming no retuning, the relative energy-per-instruction overhead of the element-wise and full-bypass implementations are respectively

$$\left.\frac{\Delta E}{E}\right|_{\text{no retuning}} = 1\% \quad \text{and} \quad \left.\frac{\Delta E}{E}\right|_{\text{no retuning}} = 7.2\%.$$

Since the vector element file is potentially on the critical path of the processor, adding bypasses might also affect the clocking rate. In both implementations, a 2-to-1 multiplexor is added on the critical path, while all remaining inputs are pre-multiplexed and therefore are off the critical path. At 0.9 V power supply, a latch with a built-in multiplexor has a 60-ps-higher setup time than a latch without a multiplexor. Thus, for a frequency of 250 MHz at 0.9 V, the delay overhead of the register file access stage, assuming no retuning, is

$$\left.\frac{\Delta D_{\text{RF}}}{D_{\text{RF}}}\right|_{\text{no retuning}} = 1.5\%.$$

There is no measurable delay overhead in other stages of the pipeline,

$\Delta D_i|_{\text{no retuning}} = 0$.

Adding bypasses does not affect the dynamic instruction count; i.e., $(\Delta N/N) = 0$. Consequently, according to the criterion, the question about the energy efficiency of implementing bypasses is reduced to evaluation of the expression

$$\theta \frac{\Delta I}{I} > \left.\frac{\Delta E}{E}\right|_{\text{no retuning}} + \eta_{\text{RF}} w_{\text{RF}} \left.\frac{\Delta D_{\text{RF}}}{D_{\text{RF}}}\right|_{\text{no retuning}}.$$

The energy and delay growth rates E_v and D_v required to compute θ can be estimated using the graphs in Figure 14 (shown later, in the Appendix). At 0.9 V power supply, the values are $E_v = 2.07$ and $D_v = 2.0$, leading to $\theta = 1.04$. Then, estimating the hardware intensity in the register file as $\eta_{\text{RF}} = 3$, and the corresponding energy weight as $w_{\text{RF}} = 0.2$, we conclude that in order to be energy-efficient, the element-wise bypass implementation must result in an increase in the architectural performance (IPC) of at least 1.8%, whereas the full bypass implementation must result in an architectural performance improvement of at least 7.8%.

10. Concluding remarks

We have described the development of the eLite DSP architecture, including a design methodology deployed for these purposes characterized by the thorough analysis of power–performance tradeoffs at early stages in the design. We have described how this effort is advancing the state of the art in power-efficient high-performance programmable DSP architectures, as well as in methodologies for such designs. This effort reflects our understanding of the sound balance among performance, power consumption, programmability, development cost (hardware and software), and production cost (chip and system) that is required for a target field (in our case, digital communications). The energy-efficiency metric described in Section 8 and in the Appendix provides a single easy-to-use objective formula for reaching an optimized balance.

The design of the eLite DSP architecture and its implementations covers aspects ranging from algorithms, applications, and a high-level language compiler, down to microarchitecture, logic design, and circuit-level technology. The result is an innovative architecture that breaks new ground in terms of the power consumed, the performance achieved, and scalability. Implementations of this architecture are expected to reduce power consumption by a factor of 4 to 6 in comparison to other DSPs offering similar performance and computing capabilities. The scalability features in the architecture enable implementations with varying SIMD width, varying number of registers in the 16-bit datapath, various organizations of the register files, varying memory bandwidth, and so on.[3]

The eLite architecture is characterized as multiple-issue statically scheduled, with a heterogeneous set of register files distributed throughout specialized units; parallelism is achieved by executing multiple instructions operating on different registers, in conjunction with single instructions operating on different registers (VLIW and SIMD). These features make it possible to achieve the performance requirements expected from next-generation digital communication applications. Some of the most salient features of the architecture and its associated implementations are related to the ability to perform computations in the SIMD manner, which is common on contemporary DSPs. However, in contrast to other DSPs, the eLite architecture is characterized by a large number of internal registers, reducing the need to access memory, as well as by the ability to dynamically create four-element vectors from arbitrary elements in the registers (SIMD with disjoint data) and operate on four-element vectors from a single register (SIMD with packed data).

Equally relevant, the eLite architecture has been designed in conjunction with an optimizing compiler to ensure the programmability of the processor in high-level language. To that effect, the architecture offers a load/store model of computation, with an orthogonal instruction set. The compiler leverages extensive research on very-long-instruction-word parallelism, enhanced with vectorization techniques as well as novel mechanisms to identify DSP-specific semantics expressed through standard C language code.

The performance potential of the eLite architecture has been shown through two examples of important computational kernels in the target domain. We have shown that the architecture can achieve asymptotically optimal performance in vector-oriented algorithms such as FIR, through the exploitation of its flexible mechanism for addressing data from the VERs. We have also shown that the architecture is well suited for contemporary data-intensive algorithms such as Viterbi decoding, in which the large VER file and the unique vector indexing capabilities permit all state metrics to be kept in registers for a frame or block of data, even for a moderately high number of states.

The advances resulting from the eLite DSP research have already led to multiple patent applications and are opening the area of low-power high-performance programmable architectures for DSP applications for further investigation in new directions. Further work in this field includes 1) the extension of the eLite architecture with cache memory mechanisms that are

[3] Because of space constraints, not all of the scalability features have been described in this paper.

suitable for their use in real-time environments, in which the nondeterministic behavior of cache memories is an issue; 2) the investigation of mechanisms to further extend the ability to exploit instruction-level parallelism and data parallelism without compromising the benefits of design simplicity, low-power consumption, and programmability; and 3) additional support for domain-specific applications. We also believe that concepts developed as part of the eLite DSP research have applicability in areas other than digital communications, in which the innovations in flexible manipulation of data offered by the eLite SIMdD model of computation, low-power architectural features, and implementation, as well as the benefits in programmability, represent major advantages over existing approaches.

11. Appendix

This section describes the performance–power optimization methodology in formal terms. For these purposes, let us consider the problem of minimizing the average power dissipation given a performance requirement $P = P_0$. The designer is allowed to modify the architecture (both ISA and microarchitecture), and also to adjust the clocking rate of the processor by retuning the circuits and changing the power-supply voltage within certain limits in order to satisfy the performance requirement at minimum power dissipation. In mathematical terms, the problem of power minimization can be reduced to the problem of minimizing the function $W(\xi, \eta, v)$ in the space of the three design variables ξ, η, and v, under the constraint $P(\xi, \eta, v) = P_0$.

Let us introduce a finite difference notation for the discrete variable ξ,

$$\left.\frac{\Delta F(\xi, \eta, v)}{\Delta \xi}\right|_{\eta v} = \frac{F(\xi + \Delta\xi, \eta, v) - F(\xi, \eta, v)}{\Delta \xi},$$

where $F(\xi, \eta, v)$ is any function of variables ξ, η, and v involved in the analysis. Neglecting the second-order terms, the constraint condition $P(\xi, \eta, v) = P_0$ can be expressed in differential form as

$$\left.\frac{\Delta P}{\Delta \xi}\right|_{\eta v} \Delta\xi + \frac{\partial P}{\partial v}\Delta v + \frac{\partial P}{\partial \eta}\Delta\eta = 0,$$

where $\Delta\eta$ and Δv are adjustments in the hardware intensity and supply voltage needed to compensate for the performance loss or gain resulting from the architectural modification $\Delta\xi$. In the remainder of the paper, we neglect second-order terms. Thus, the methodology described here is applicable only for "small" variations to the architecture, so that the resulting relative increments in all involved functions and their derivatives are small $[(\Delta F/F) \ll 1, (\Delta F'/F') \ll 1]$, and relative changes in the supply voltage v and hardware intensity η needed to compensate the performance loss or gain resulting from architectural modifications $\Delta\xi$ are also small $[(\Delta v/v) \ll 1, (\Delta\eta/\eta) \ll 1]$.

Under these assumptions, the problem of establishing the energy efficiency of a particular modification to the architecture $\Delta\xi$ can be reduced to that of finding a relation between relative changes in processor characteristics for which

$$\left.\frac{\Delta W}{\Delta \xi}\right|_{P=P_0} = \left.\frac{\Delta W}{\Delta \xi}\right|_{\eta v} + \left.\frac{\partial W}{\partial \eta}\frac{\Delta \eta}{\Delta \xi}\right|_{P=P_0} + \left.\frac{\partial W}{\partial v}\frac{\Delta v}{\Delta \xi}\right|_{P=P_0} < 0.$$

Neglecting the second-order terms in the calculation of the finite differences in the constraint formula, we arrive at the following expression for the ratio of finite differences $\Delta\eta$, Δv, and $\Delta\xi$ subject to the constraint $P(\xi, \eta, v) = P_0$:

$$\left.\frac{D_v}{v}\frac{\Delta v}{\Delta \xi}\right|_{P=P_0} - \left.\frac{1}{D}\frac{\partial D}{\partial \eta}\frac{\Delta \eta}{\Delta \xi}\right|_{P=P_0} = -\left.\frac{\Delta f}{f \Delta \xi}\right|_{\eta v} - \frac{\Delta I}{I \Delta \xi} + \frac{\Delta N}{N \Delta \xi},$$

where D_v is the delay growth rate, defined as a dimensionless partial derivative of the critical path delay D with respect to the supply voltage,

$$D_v = -\frac{v}{D}\frac{\partial D}{\partial v}.$$

The value of D_v can be estimated empirically for the selected technology, supply voltage, and circuit style. To evaluate it, the designer can simulate the dependence of the delay through the hardware blocks that are expected to be on the critical path upon the supply voltage.

The partial derivative $\partial W/\partial\eta$ in the energy-efficiency formula is calculated as

$$\frac{\partial W}{\partial \eta} = IEf\left(\frac{\partial E}{E\partial \eta} - \frac{\partial D}{D\partial \eta}\right),$$

and the partial derivative $\partial W/\partial v$ is calculated as

$$\frac{\partial W}{\partial v} = \frac{IEf}{v}(E_v + D_v),$$

where E_v is the energy growth rate, defined as a dimensionless partial derivative of the average energy dissipated per instruction with respect to the supply voltage,

$$E_v = \frac{v}{E}\frac{\partial E}{\partial v}.$$

The value of E_v for CMOS circuits is typically close to 2, since the energy of the charged capacitance is proportional to the square of the supply voltage, $E = (Cv^2/2)$. A more accurate estimate for the value of E_v for a selected technology and circuit style can be obtained by simulating representative circuits over a range of supply voltages.

Assuming that circuits in the pipeline are tuned according to the optimal balance between the power supply and the hardware intensity [13], $\eta = E_v/D_v$, the expressions above can be reduced. Substituting the calculated terms into the energy-efficiency formula, and taking advantage of the property of hardware intensity,

$$\frac{1}{E}\frac{\partial E}{\partial \eta} = -\frac{\eta}{D}\frac{\partial D}{\partial \eta},$$

we arrive at the following criterion for energy efficiency, after grouping terms in front of the partial derivatives:

$$-\theta\frac{1}{f}\frac{\Delta f}{\Delta \xi}\bigg|_{\eta v} - \theta\frac{1}{I}\frac{\Delta I}{\Delta \xi} + \frac{1}{E}\frac{\Delta E}{\Delta \xi}\bigg|_{\eta v} + (1+\theta)\frac{1}{N}\frac{\Delta N}{\Delta \xi} < 0,$$

where θ is the energy delay growth ratio, defined as $\theta = E_v/D_v$.

The increments of the architectural complexity $\Delta\xi$s can be omitted from the formula, as long as a fixed hardware intensity and supply voltage are assumed when calculating the finite increments ΔE and Δf so that the meaning of partial derivatives with respect to the architectural complexity is preserved. Then, a simplified form of the criterion can be used:

$$-\theta\frac{\Delta f}{f} - \theta\frac{\Delta I}{I} + \frac{\Delta E}{E} + (1+\theta)\frac{\Delta N}{N} < 0.$$

The increments of all quantities in the expression above appear in relative form and are thus dimensionless. This feature makes this formula easy to use as a basis for negotiation between architects and circuit designers. For example, if $E_v = D_v = 2$ ($\theta = 1$), and if some microarchitectural enhancement (say, adding a bypass) increases the average energy per instruction by 5% and potentially increases the delay on the critical path by 2%, without any effect on the dynamic instruction count, then it will be energy-efficient only if the resulting increase in the architectural speed I is at least 7%.

The assumption about the optimal tuning of circuits in every pipeline stage for every architectural alternative and unchanged hardware intensity, used in deriving the formula above, imposes special rules on calculating terms $\Delta f/f$ and $\Delta E/E$ in the energy-efficiency criterion. In particular, these relative increments must be calculated assuming that the processor pipeline is re-optimized after every modification to the microarchitecture, in such a way that the optimal balance of hardware intensity ($\eta = \theta$) in the pipeline is preserved [13]. This leads to the following formula for calculating $\Delta E/E$:

$$\frac{\Delta E}{E} = \frac{\Delta E}{E}\bigg|_{\text{no retuning}} + \Sigma\eta_i w_i \frac{\Delta D_i}{D}\bigg|_{\text{no retuning}} + \theta\frac{\Delta f}{f},$$

where

$$\frac{\Delta D_i}{D}\bigg|_{\text{no retuning}} \quad \text{and} \quad \frac{\Delta E}{E}\bigg|_{\text{no retuning}}$$

are "naive" increments in delays and energies in pipeline stages i, calculated assuming that no retuning is done in any circuits. Circuit designers usually have no difficulty estimating these quantities. In the above formulas, w_i are the energy weights of the pipeline stages, calculated taking into account activity factors, and η_i are hardware intensities in the corresponding pipeline stages, which show how close to the performance limit the circuits are in those pipeline stages. Then, the higher the energy weight w_i and hardware intensity η_i of pipeline stage i, the higher the weight of the corresponding "naive" increment in the critical path delay in the weighted average above. These sensitivities η_i can be measured from energy–delay curves by tuning tools such as the IBM EinsTuner [39]. Notice that all coefficients in the above formulas have to be measured just once for the baseline pipeline, after which they can be used for evaluating numerous architectural alternatives. Combining the two formulas, we arrive at the form of the energy-efficiency criterion that does not require estimating term $\Delta f/f$:

$$\theta\frac{\Delta I}{I} > \frac{\Delta E}{E} + \Sigma\eta_i w_i \frac{\Delta D_i}{D} + (1+\theta)\frac{\Delta N}{N}.$$

Although the energy-efficiency formula has been derived for minimizing power for a given performance requirement, the same formula is also valid for the reciprocal problem of performance maximization subject to a constant power constraint, $W(\xi, \eta, v) = W_0$. For some combinations of the values of E_v and D_v, this energy-efficiency criterion can be viewed as a differential form of one of the conventional power–performance metrics [6, 7, 9–11]. It is easy to show that the "MIPS-to-the-power-of-γ-per-watt" metrics are special cases of our energy-efficiency criterion, written in the integral form. For example,

- $E_v = 2$, $D_v = 1$ ($\theta = 2$) leads to "MIPS^3 per watt."
- $E_v = 2$, $D_v = 2$ ($\theta = 1$) leads to "MIPS^2 per watt."
- $D_v \gg E_v$ ($\theta \ll 2$) leads to "MIPS per watt."
- $E_v = 2$, $D_v = 0.5$ ($\theta = 4$) leads to "MIPS^5 per watt."

In addition to its formal derivation, other advantages of our new metric are its generality and the ability to calculate the exponent in the expressions above as $\theta + 1$ for every particular case, taking into account technology and circuit characteristics. It also provides a method for calculating increments in MIPS and watts in the "MIPS-to-the-power-of-γ-per watt" with no need to retune a design.

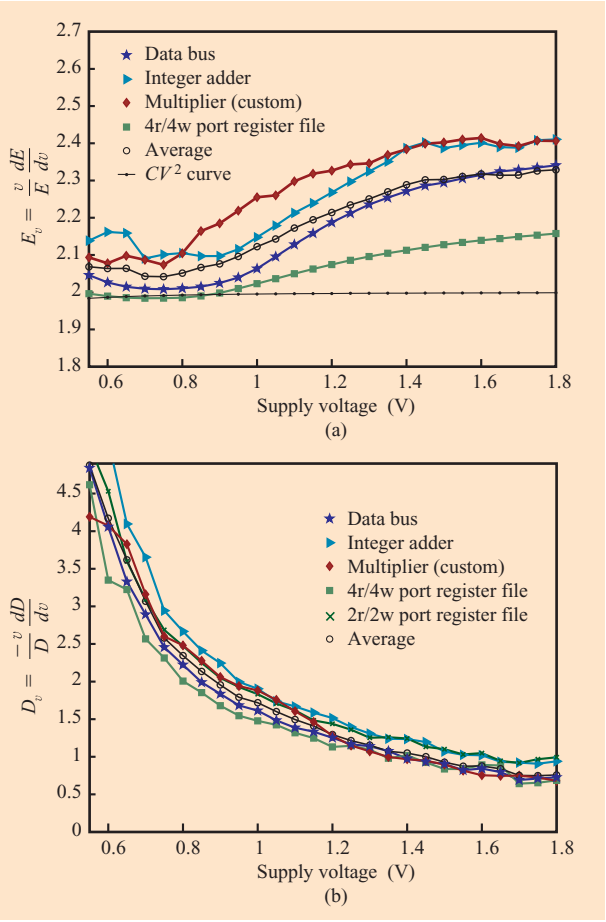

Figure 14

Simulation results for E_v and D_v. Reprinted from [12], with permission; © 2002 ACM, Inc.

Effect of circuit and technology characteristics

Although theoretical formulas could be used to determine the energy and delay growth rates D_v and E_v, a more practical way to calculate the values of these coefficients is through the simulation of representative circuits over a range of power-supply voltages. For the evaluation of D_v, it is important to select functional blocks that can potentially be located on the critical path; on the other hand, the most significant power consumers should be simulated for the evaluation of E_v. As an illustration, we describe the case of a representative set of blocks in a typical microprocessor, such as an inter-unit star-connect data bus; a synthesized 32-bit integer adder; a full-custom 16-bit multiplier; the critical read path of a 4-read/4-write-port full-custom register file; and a 2-read/2-write 16-entry semicustom register file built with latches and multiplexors, all implemented in a 0.13-μm technology. For the energy analysis, we simulated all blocks with PowerMill** [40], applying random patterns to the inputs with a switching factor of 0.3 for 200 to 500 cycles (depending on the size of the circuit). We also used the PathMill** static timer for delay analysis.

Figure 14(a) shows simulation results for E_v. The curves on the graph correspond to the blocks described above. As a reference, a curve corresponding to the $E = Cv^2/2$ dependence is also plotted. Figure 14(a) shows that, for all of the blocks, the value of E_v is higher than the value 2.0 that corresponds to the $E = Cv^2/2$ dependence. This super-V_{dd}^2 dependence of energy on the supply voltage is partially explained by higher glitching activity at higher supply voltages. Those blocks that have more significant glitching factors also demonstrate higher values of E_v, especially at high supply voltages. Another reason is the faster than square growth of the short circuit power component [41].

Figure 14(b) shows simulation results for D_v. The curves on the graph correspond to the previously described blocks. For all blocks, D_v increases rapidly for low values of V_{dd}, especially as V_{dd} approaches the transistor threshold voltage. For high values of V_{dd}, D_v drops below unity because of the velocity saturation effect. For custom-designed blocks, D_v tends to be smaller than for ASIC-synthesized blocks, especially at low values of V_{dd}, because of the (selective) use of low-threshold devices in custom circuits, and low-voltage circuit styles (e.g., smaller transistor stacks). The thick lines on the graphs, marked with circles, represent the averages over all simulated blocks, calculated for unity weight factors.

**Trademark or registered trademark of Analog Devices, Inc. or Synopsys, Inc.

References

1. J. Eyre, "The Digital Signal Processing Derby," *IEEE Spectrum*, **30,** No. 6, 62–68 (2001).
2. J. Glossner, J. H. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, and M. Ware, "Trends in Compilable DSP Architectures," *Proceedings of the 2000 IEEE Workshop on Signal Processing Systems (SiPS)*, October 2000, pp. 181–199.
3. StarCore, *SC140 DSP Core Reference Manual*, December 1999; see *http://e-www.motorola.com/brdata/PDFDB/docs/MNSC140CORE.pdf.*
4. Texas Instruments, Inc., *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000; see *http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf*.
5. J. Tomarakos and C. Duggan, "32-bit SIMD Sharc Architecture Digital Audio Signal Processing Applications," *J. Audio Eng Soc*. **48,** No. 3, 220 (2000).
6. D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. Cook, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro* **20,** No. 6, 26–44 (November 2000).
7. T. Burd, "Energy-Efficient Processor System Design," Ph.D. Thesis, University of California, Berkeley, 2001.

8. A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-Power CMOS Digital Design," *IEEE J. Solid-State Circuits* **27,** No. 4, 473–484 (April 1992).
9. R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *IEEE J. Solid-State Circuits* **31,** No. 9, 1277–1283 (September 1996).
10. M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-Power Digital Design," *Proceedings of the IEEE Symposium on Low Power Electronics and Design*, October 1994, pp. 8–11.
11. V. Zyuban and P. Kogge, "Optimization of High-Performance Superscalar Architectures for Energy Efficiency," *Proceedings of the IEEE Symposium on Low Power Electronics and Design*, August 2000, pp. 84–89.
12. V. Zyuban, "Unified Architecture Level Energy-Efficiency Metric," *Proceedings of the Great Lakes Symposium on VLSI*, April 2002, pp. 24–29.
13. V. Zyuban and P. Strenski, "Unified Methodology for Resolving Power–Performance Tradeoffs at the Microarchitectural and Circuit Levels," *Proceedings of the International Symposium on Low Power Electronics and Design*, July 2002, pp. 166–171.
14. S. V. Kosonocky, A. J. Bhavnagarwala, K. Chin, G. D. Gristede, A.-M. Haen, W. Hwang, M. B. Ketchen, S. Kim, D. R. Knebel, K. W. Warren, and V. Zyuban, "Low-Power Circuits and Technology for Wireless Digital Systems," *IBM J. Res. & Dev.* **47,** No. 2/3, 283–298 (2003, this issue).
15. R. Stallman, "Using and Porting GNU CC," Free Software Foundation, version 2.7.2.1, June 1996; see *http://hal.csd.auth.gr/thelug/fags/gcc.html.*
16. D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy, "A New Approach to DSP Intrinsic Functions," *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, January 2000, pp. 2892–2901.
17. K. W. Leary and W. Waddington, "DSP/C: A Standard High-Level Language for DSP and Numeric Processing," *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, 1990, pp. 1065–1068.
18. B. Krepp, "DSP-Oriented Extensions to ANSI C," *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, 1997, pp. 658–664.
19. J. H. Moreno, K. Ebcioglu, M. Moudgill, and D. Luick, "ForestaPC (Scalable VLIW) User Instruction Set Architecture," *Research Report RC-20733*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1996.
20. K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software," *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, 1988, pp. 3–21.
21. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill, "Dependence Flow Graphs: An Algebraic Approach to Program Dependencies," *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1991, pp. 67–78.
22. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Program. Lang. & Syst.* **13,** No. 4, 451–490 (October 1991).
23. M. Moudgill, J. H. Moreno, K. Ebcioglu, E. R. Altman, S.-K. Chen, and A. Polyak, "Compiler/Architecture Interaction in a Tree-Based VLIW Processor," *IEEE Technical Committee on Computer Architecture Newsletter*, June 1997, pp. 332–335.
24. J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak, "Simulation/Evaluation Environment for a VLIW Processor Architecture," *IBM J. Res. & Dev.* **41,** No. 3, 287–302 (1997).
25. D. Naishlos, M. Biberstein, and A. Zaks, "Compiler Vectorization Techniques for Disjoint SIMD Architectures," *Research Report H-0146*, Haifa Research Laboratory, Haifa, Israel, November 2002.
26. S. Carr and K. Kennedy, "Improving the Ratio of Memory Operations in Floating-Point Operations in Loops," *ACM Trans. Program. Lang. & Syst.* **16,** No. 6, 1768–1810 (November 1994).
27. G. Goff, K. Kennedy, and C.-W. Tseng, "Practical Dependence Testing," *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, June 1991, pp. 15–29.
28. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1996.
29. D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformation for High-Performance Computing," *ACM Computing Surv.* **26,** No. 4, 345–420 (1994).
30. M. Moudgill and A. Zaks, "Minimizing Inter-File Transfers in Architectures with Separate Address Registers," *Research Report RC-21884*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, November 2000.
31. ETSI, *Digital Cellular Telecommunications System (Phase 2+) (GSM); Enhanced Full Rate (EFR) Speech Processing Functions. General Description (GSM 06.51 Version 7.0.2)*, Sophia Antipolis, France, 1998.
32. ETSI, *Digital Cellular Telecommunications System (Phase 2+) (GSM); Adaptive Multi-Rate (AMR) Speech Processing Functions. General Description (GSM 06.71 Version 7.0.1)*, Sophia Antipolis, France, 1998.
33. *ITU-T, Recommendation G.729, Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, Geneva, Switzerland, March 1996.
34. S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
35. R. Johannesson and K. Sh. Zigangirov, *Fundamentals of Convolutional Coding*, IEEE Press, New York, 1999.
36. M. Biver, H. Kaeslin, and C. Tommasini, "In-Place Updating of Path Metrics in Viterbi Decoders," *IEEE J. Solid-State Circuits* **24,** No. 4, 1158–1160 (August 1989).
37. Motorola, Inc., and Agere Systems, "How To Implement a Viterbi Decoder on the StarCore SC140," *Application Note ANSC140VIT/D*, July 18, 2000; see *http://e-www.motorola.com/brdata/PDFD/docs/ANSC140VIT.pdf.*
38. 3rd Generation Partnership Project (3GPP), *Technical Specification 3G TS 25.212, Multiplexing and Channel Coding (FDD)*, September 2002; see *http://www.3gpp.org/ftp/Specs/latest/Rel-4/25_series/25212-460.zip.*
39. A. R. Conn, I. M. Elfadel, W. W. Molzen, Jr., P. R. O'Brien, P. N. Strenski, C. Visweswariah, and C. B. Whan, "Gradient-Based Optimization of Custom Circuits Using a Static-Timing Formulation," *Proceedings of the Design Automation Conference*, June 1999, pp. 452–459.
40. see *http://www.synopsys.com/*.
41. J. Veendrick, "Short-Circuit Dissipation of Static CMOS Circuitry and Its Impact on the Design of Buffer Circuits," *IEEE J. Solid-State Circuits* **19,** No. 4, 468–473 (August 1984).

Received March 12, 2002; accepted for publication October 15, 2002

Jaime H. Moreno *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (jhmoreno@us.ibm.com).* Dr. Moreno is a Senior Manager in the Computer Architecture Department. He received a degree in electrical engineering from the University of Concepcion, Chile, in 1979, and M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles in 1985 and 1989, respectively. In 1992, Dr. Moreno joined the IBM Research Division, where he initially performed research on very-long-instruction word processor architectures for server systems, later in superscalar processors, and more recently in embedded systems, including recent research on digital signal processor architectures. Before joining IBM, he was a faculty member in the Department of Electrical Engineering at the University of Concepcion, Chile, and collaborated as a postdoctoral researcher at UCLA. Dr. Moreno is coauthor of the books *Introduction to Digital Systems* (Wiley, 1999) and *Matrix Computations on Systolic-Type Arrays* (Kluwer, 1992). He holds several patents in processor architecture, and has been recognized as an IBM Master Inventor. His interests include processor architectures, instruction-level and data-level parallelism, and application-specific and domain-specific architectures.

Victor Zyuban *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (zyuban@us.ibm.com).* Dr. Zyuban received his B.S. and M.S. degrees from the Moscow Institute of Physics and Technology in 1993 and 1995, respectively, and his Ph.D. degree in computer science and engineering from the University of Notre Dame in 2000. From 1995 to 1996, he worked in the Moscow Center for SPARC Technologies. He is currently a Research Staff Member at the IBM Thomas J. Watson Research Center. Dr. Zyuban is working on a low-power DSP research project in which he has been involved in ISA definition, microarchitecture, and physical design. He is currently leading the development of a semicustom eLite core test chip. His research interests include low-power circuitry, microarchitecture, and methodologies for low-power design.

Uzi Shvadron *IBM Israel Science and Technology, Haifa Research Laboratory, Haifa 31905, Israel (shvadron@il.ibm.com).* Mr. Shvadron joined IBM in 1983 after receiving B.S. and M.S. degrees in electrical engineering from the Technion–Israel Institute of Technology. Since 1984 he has been a Research Staff Member of the Haifa Research Laboratory. He has participated in several research projects in the fields of voice and audio signal processing, VoIP gateway, image coding, optical inspection, and electron beam lithography techniques. Mr. Shvadron's primary experience is in real-time signal processing using IBM signal processor architectures, writing sophisticated algorithms for new applications on DSPs. His main research interests are in the areas of signal and image processing. He is currently the leader of a software development team for a new DSP architecture—the eLite DSP core. The work includes close interaction with other IBM laboratories in the areas of architecture design, benchmarks, and tools development.

Fredy D. Neeser *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland (nfd@zurich.ibm.com).* Dr. Neeser received Dipl. Ing. and Ph.D. degrees in electrical engineering from the Swiss Federal Institute of Technology (ETH) in 1986 and 1993, respectively. In 1994, he joined the IBM Research Division, where he designed key digital signal processing (DSP) algorithms for IBM ThinkPad V.34 and V.90 modem subsystems, for which he received two IBM Outstanding Technical Achievement Awards. He holds numerous patents in this field. Since 1999, he has been involved in the design of the IBM eLite VLIW/SIMD DSP architecture, working on architectural enhancements for filtering and Viterbi decoding and on a cycle-accurate simulator for the eLite DSP. Dr. Neeser's research interests include equalization, coding and turbo-receiver techniques, and their applications in wireless LAN systems.

Jeff H. Derby *IBM Microelectronics Division, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (jhderby@us.ibm.com).* Dr. Derby received his Ph.D. degree in electrical engineering from Columbia University in 1975. In 1982 he joined IBM in Research Triangle Park, North Carolina, after spending seven years with Bell Laboratories in Whippany, New Jersey. At IBM, he spent ten years working on network architectures and switching-system design for high-speed packet networks including frame relay and ATM. Dr. Derby received an IBM Outstanding Technical Achievement Award for this work in 1992. Since 1992, his work has focused on analog and digital signal processing subsystems with application to telecommunications. He has been involved in the definition of digital signal processor architectures, in the evaluation of DSP architectures for broadband wired and wireless access applications, and in the development of system-on-a-chip architectures incorporating signal processing capabilities. Dr. Derby is an inventor or co-inventor of 20 U.S. patents. He is a Senior Member of the IEEE and a member of the IEEE Communications Society and the IEEE Signal Processing Society. He is also Adjunct Associate Professor in the Department of Electrical and Computer Engineering, Duke University, Durham, North Carolina.

Malcolm S. Ware *IBM Microelectronics Division, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (mware@us.ibm.com).* Mr. Ware received his B.S. degree in electrical engineering from Purdue University in 1983 and his M.S. degree in computer architecture and communications from North Carolina State University in 1986. He spent his first ten years with IBM at the Research Triangle Park facility developing speech and image coding algorithms, music synthesizers, and low-speed modems for the Mwave DSP. In 1993 he went on international assignment for five years to the IBM Zurich Research Laboratory in Switzerland, and worked with IBM Fellow Gottfried Ungerboeck on high-speed modems including V.34 and V.90 for the Mwave products shipped in IBM ThinkPads. After returning to Research Triangle Park for two years to examine broadband and network processing opportunities, he spent 16 months at the Zurich Research Laboratory developing prototypes of ADSL, SHDSL, and VDSL broadband transceivers. For the last year, Mr. Ware has been studying wireless transmission systems at the IBM Research Triangle Park site. He holds more than ten U.S. patents, with another 20 under consideration at the U.S. Patent Office.

Krishnan Kailas *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (kailas@us.ibm.com).* Dr. Kailas received his M.S. (1998) and Ph.D. (2001) degrees in electrical and computer

engineering from the University of Maryland, College Park, and his B.Tech. (Hons) degree in electronics and communication engineering from the University of Calicut, India. His doctoral dissertation research was on microarchitectures and compilation support for clustered ILP processors. While he was a graduate student, he worked on VLIW processor architectures for the dynamic binary translation (DAISY) project at IBM Thomas J. Watson Research Center from 1997 to 1999. He was an R&D engineer from 1988 to 1994 at Bhabha Atomic Research Center, Bombay, India, where he designed mission-critical microprocessor-based real-time systems. Since 2001, he has been a Research Staff Member at the IBM Thomas J. Watson Research Center, working on high-performance, low-power DSPs. His current research interests include microarchitecture and code-generation techniques for real-time operating systems. Dr. Kailas is a member of ACM, IEEE, and Sigma Xi.

Ayal Zaks *IBM Israel Science and Technology, Haifa Research Laboratory, Haifa 31905, Israel (zaks@il.ibm.com).* Dr. Zaks received B.Sc., M.Sc., and Ph.D. degrees in mathematics/operations research from Tel Aviv University. He joined the IBM Haifa Research Laboratory in 1997, initially to work on compiler back-end optimizations for the AS/400. Since 2000, Dr. Zaks has worked on developing an optimizing compiler for eLite, spending one year at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York.

Amir Geva *IBM Israel Science and Technology, Haifa Research Laboratory, Haifa 31905, Israel (geva@il.ibm.com).* In 1997 Mr. Geva received a B.Sc. degree in computer engineering from the Technion–Israel Institute of Technology. He joined IBM as an intern in 1994 and became a full-time employee after receiving his degree. Mr. Geva worked for a year on evaluation benchmarks of the PowerPC SIMD extensions (VMX), later known as AltiVec. He then moved to programming DSP code and integration of host and embedded code for a voice over IP telephony gateway. Mr. Geva has been working on the eLite DSP project since mid-2000 and has been involved primarily in producing development tools.

Shay Ben-David *IBM Israel Science and Technology, Haifa Research Laboratory, Haifa 31905, Israel (bendavid@il.ibm.com).* Mr. Ben-David received B.Sc. and M.Sc. degrees in electrical engineering from the Technion–Israel Institute of Technology, Haifa. He joined the Haifa Research Laboratory in 1994, initially working on DSP application in the Mwave project. Mr. Ben-David received an IBM Outstanding Technical Achievement Award for his work on the Java Media Framework. He holds several pending patents. He currently works in the Signal Processing Group in the Haifa Research Laboratory developing DSP applications and architectures.

Sameh W. Asaad *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (asaad@us.ibm.com).* Dr. Asaad received his B.S. degree in communications engineering from Cairo University, Egypt, in 1989. He received his M.S. and Ph.D. degrees in electrical and computer engineering from Vanderbilt University in 1995 and 2000, respectively. He is an Advisory Engineer at the Thomas J. Watson Research Center, where he joined IBM in 1996. Dr. Asaad's current interests include modeling, design, and implementation of domain-specific architectures and low-power systems.

Thomas W. Fox *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (foxy@us.ibm.com).* Mr. Fox is an Advisory Engineer at the IBM Thomas J. Watson Research Center, where he has worked on low-power DSP microarchitectures and designs since 2001. He received his B.S. and M.E.E. degrees in electrical engineering from Rensselaer Polytechnic Institute in 1989 and 1990, respectively. Prior to joining the IBM Research Division, he was an architect and designer of 3D geometry graphics chips for the IBM Server Group in Austin, Texas. Mr. Fox received an IBM Outstanding Technical Achievement Award in 1999 for design innovations in 3D geometry lighting, and he represented IBM on the OpenGL Architecture Review Board. He holds three U.S. patents in the graphics field, and has seven patents pending evaluation in the U.S., Germany, and Japan in the graphics and floating-point domains. His professional interests include processor architectures, floating-point arithmetic, and 3D graphics.

Daniel Littrell *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (littrell@us.ibm.com).* Mr. Littrell received his B.S. degree in electrical engineering from the University of California at Santa Cruz; he is currently pursuing his M.S.E.E. degree at Columbia University. He has worked at the IBM Thomas J. Watson Research Center since April 2001, during which time he has focused mainly on low-power DSP realization. His professional interests include computer architectures, data visualization, memory structures, and logic synthesis.

Marina Biberstein *IBM Science and Technology, Haifa Research Laboratory, Haifa 31905, Israel (biberste@il.ibm.com).* Ms. Biberstein received her B.A. degree in mathematics and her M.Sc. degree in computer science from the Technion–Israel Institute of Technology in 1995 and 1999, respectively. Since 1999, she has been with the IBM Research Laboratory in Haifa. Her research interests include code optimization technologies, program analysis, and error-correcting codes.

Dorit Naishlos *IBM Israel Science and Technology, Haifa Research Laboratory, Haifa 31905, Israel (dorit@il.ibm.com).* Ms. Naishlos received a B.Sc. degree in computer science from the Technion–Israel Institute of Technology in 1998 and an M.Sc. degree in computer science from the University of Maryland in 2000. During her studies she worked at IBM on post-link optimizations (Haifa Research Laboratory, 1997–1998) and parallel languages for shared and distributed memory platforms (Thomas J. Watson Research Center, 1999). While at Maryland, she specialized in compiling for an explicit multi-threaded framework that exploits fine-grained parallelism on-chip. In 2001 Ms. Naishlos joined IBM, where she has been working on compiler development and optimizations, in particular vectorization, in the Code Optimizations group of the IBM Research Laboratory in Haifa. Her interests include compilation, code optimization, and parallel computing.

H. Hunter *Computer and System Research Laboratory, University of Illinois at Urbana–Champaign, Urbana, Illinois 61801.* Ms. Hunter received B.S. (1999) and M.S. (2002) degrees from the University of Illinois and is currently pursuing a Ph.D. in electrical engineering. She has been a Tau Beta Pi Fellow and holder of a National Science Foundation Fellowship, University of Illinois Distinguished Fellowship, and University of Illinois ECE Department Koehler Fellowship. During the summer of 2000, she worked in the IBM S/390 Microprocessor Development group in Boeblingen, Germany, and held summer intern positions at the IBM Thomas J. Watson Research Center in 2001 and 2002. Her research interests are in compiler/architecture co-design for embedded and DSP processors.