RC 22568 (W0209-079) 09/15/2002
Computer Sciences/Mathematics

IBM Research Report

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

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

(Previously submitted to *IBM Journal of Research and Development.)*

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

LIMITED DISTRIBUTION NOTICE
This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Research Division
Almaden ● Austin ● China ● Haifa ● Tokyo ● T.J. Watson ● Zurich

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

J.H. Moreno,[1] V. Zyuban,[1] U. Shvadron,[2] F. Neeser,[3] J. Derby,[4]
M. Ware,[4] K. Kailas,[1] A. Zaks,[2] A. Geva,[2] S. Ben-David,[2]
S. Asaad,[1] T. Fox,[1] M. Biberstein,[2] D. Naishlos,[2] H. Hunter[1]

[1]IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA
[2]IBM Haifa Research Laboratories, Haifa, Israel
[3]IBM Zurich Reseach Laboratories, Zurich, Switzerland
[4]IBM Research Triangle Park, Raleigh, NC, USA

Abstract

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 (ILP) and data-level parallelism (SIMD) to achieve high-performance, its suitability as 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 architecture/compiler/implementation co-development, and the approach used for power-performance evaluation and trade-off 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 design 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 the case of 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 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 - ILP), simultaneous execution of the same instruction on multiple data elements (e.g., single-instruction multiple-data - 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 has 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 tolerate programming in Assembly language. Instead, DSPs are expected to be programmed using a high-level language such as C [1-2], similarly to the case of general-purpose processors. As a result, the cornerstone of contemporary digital communications are a new generation of digital signal processors characterized by their *power efficienc*y as well as their *improved programmability* in a high-level language.

Applications 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, wireless base-stations. Certain functions will 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 a xDSL transceiver, a wireless LAN transceiver, a speech coder for voice-over-DSL, and some network processing functions, whereas a 3G handset may integrate the wireless digital baseband, a speech coder, and audio and MPEG4 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 where the fundamental limiting factor is the power available to support electronics, but where performance requirements (in terms of instructions executed per second 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 where 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-stations equipment is housed in small cabinets with limited ventilation mounted outdoors.

Memory-bound applications. These are applications with nontrivial performance requirements but where 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 with relaxed performance requirements but where 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 thoroughly commodities.

The foregoing discussion indicates that many key applications will impose severe power and performance 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, MACs per second per mW).

A common thread that runs through all 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 DMT and 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 architectures 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 Figure 1. For example,

- 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 (*VLIW* and *SIMD with packed data*).
- 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*).
- 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*.

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
Functional units organization	- 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)

Figure 1: Distinctive features characterizing contemporary DSPs

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 at the instruction set architecture (ISA) and microarchitecture definition. At these stages, the potential for power savings is more significant than at lower-level stages, and the opportunity for making power-performance trade-offs 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 (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 impact the clocking rate.

Power-performance metrics of the form $MIPS^{\gamma}/Watt$ [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 architecture*, 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 type of designs. This effort grows from the understanding that the important matter is 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). In order to achieve these usually conflicting optimization goals, the design of the eLite DSP architecture and its implementations covers aspects ranging from algorithms, applications, and 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 operating on different registers (VLIW), in conjunction with single instructions operating on different registers (single instruction multiple disjoint data, SIMdD), as well as single instructions operating on packed data (single instruction multiple packed data SIMpD). A novel indirect register addressing mechanism enables the dynamic composition of vectors with four elements selected from a large multiported register file.

In the rest of this paper, we describe how all the aspects listed above have influenced the design choices made in the eLite architecture. In Section 2, we describe the co-design methodology applied to eLite, then in Section 3 we summarize the fundamental concepts behind the power/performance evaluation methodology deployed. In Section 4, we provide a brief description of the architecture, and Section 5 gives a description of the organization of the processor. In Section 6 we indicate the salient features of the associated optimizing compiler. Section 7 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. Section 8 provides an example of the power-performance evaluations carried out to resolve design trade-offs. Examples of kernels from representative algorithms coded for this architecture are illustrated in Section 9. Final comments and conclusions are given in Section 10.

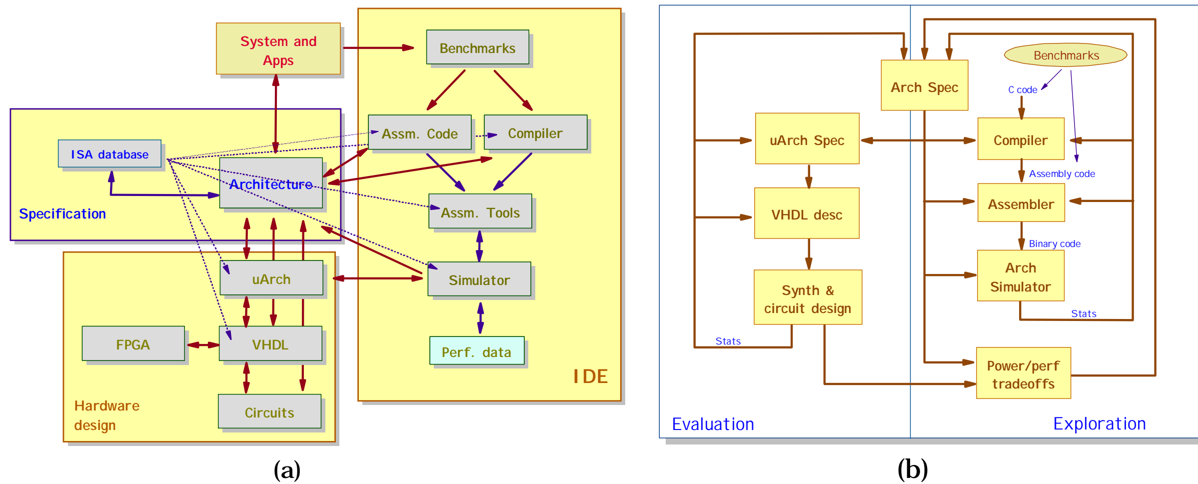
Unique research contributions arising from the eLite DSP include more than just the architecture and the processor. Other aspects are a power-performance methodology at the architecture level [12]; new circuit techniques for ultra-low power implementations, as described in [13]; important compiler optimizations for DSP operations; and system level design experience.

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

The co-design and evaluation methodology deployed in the development of eLite has been built around the interaction between the multiple dimensions of the problem, as depicted in Figure 2. At the center of the interacting components (see Figure 2a) 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 the components of the environment, which range from code generation and performance analysis (the block labelled “IDE” in Figure 2a), to analysis from logic implementation and circuit design (the block labelled “Hardware design” in the same figure).

The processes by which these components are exercised is illustrated in Figure 2b. 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 trade-offs. 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 fea-

tures and changes, by modifying the different components of the environment as necessary and by performing cycle-accurate simulation, although the simulation does not include all 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) (b)

Figure 2: Co-development interactions and design methodology

In order to optimize the eLite architecture according to the power/performance metric we have chosen to use a set of benchmarks. This set was chosen according to the following three criteria:

- relevance to the target applications, mainly wireless and line communication, voice applications, and media applications;
- moderate size functions; and
- coverage of the various units as well as system issues such as interrupts.

The set of benchmarks that has been chosen includes simple and very common functions such as FIR filters, IIR filters, vector add, vector max, control code, etc. The benchmark suite also includes more complex functions such as FFT, Interpolator and decimator, 2D-IDCT, and Viterbi decoder, among others. An instruction-set simulator was used for each version of the architecture to analyze the current performance. Conclusions were made about areas in which 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 metric while trying to maximize performance. This process was repeated for each version of the architecture, leading to significant performance improvement each time while maintaining modest rise in power consumption and chip size.

3. Methodology for power-performance trade-offs

We now describe the methodology and new metric developed for carrying out power-performance trade-offs in the design of the architecture [12].

Power-performance optimization

Let us consider the problem of optimizing the power-performance characteristics of a processor in the space of two variables: architectural complexity and power supply voltage. To allow a mathematical analysis of the problem, we introduce a discrete variable ξ that represents a measure of the architectural complexity of the processor. The domain of this variable can be defined by ordering all possible architectural alternatives, and assigning a numeric value to each of them. Then, any architectural modification to the processor results in an increment or decrement in the value of ξ. Examples of variations in 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. Architectural complexity is treated as an independent variable in the optimization process.

Power supply voltage v is treated as the second 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 the technology is qualified. Then, the performance and power characteristics of a processor can be viewed as functions of the independent variables ξ and v, where v is a continuous and ξ is a discrete variable, as follows:

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
Maximum clocking rate	$f = f(\xi, v)$	Clock frequency
Energy per instruction	$E = E(\xi, v)$	Average energy per instruction completed on the same benchmark suite

The average energy per instruction completed is calculated as $E = \Sigma_i w_i E_i$, wherein 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.

To a first approximation, N and I depend only on the architectural complexity ξ and are independent of the supply voltage. The clocking rate f and the average energy per instruction E depend both on the architectural complexity ξ and the supply voltage v. The processor performance P on the given benchmark suite can be expressed as

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

The expression for power dissipation $W(\xi, v)$ depends upon the implementation details of the processor. Particularly, the form of the expression for 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 4, 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*, 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

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

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

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 adjust the clocking rate of the processor by changing the power supply voltage within certain limits, 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 $W(\xi, v)$ in the space of two design variables ξ and v, under the constraint $P(\xi, v) = P_0$. Let us introduce the finite difference notation for the discrete variable ξ

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

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

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

where Δv is the adjustment in the supply voltage needed to compensate for the performance loss or gain resulting from the architectural modification $\Delta\xi$. Here, and in the remainder of the paper, we neglect second-order terms of the form $\frac{\partial^2 F}{\partial v^2}(\Delta v)^2$ and $\frac{\Delta \partial F}{\Delta \xi \partial v}\Delta\xi\Delta v$, where F is any function involved in the analysis such as W, P or f. 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 ($\frac{\Delta F}{F} \ll 1$, $\frac{\Delta F'}{F'} \ll 1$), and the relative changes in the supply voltage v needed to compensate the performance loss or gain resulting from architectural modifications $\Delta\xi$ are also small ($\frac{\Delta v}{v} \ll 1$).

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

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

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

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 Δv and $\Delta\xi$ subject to the constraint $P(\xi, v) = \pi$

$$\left.\frac{\Delta v}{\Delta\xi}\right|_{P = P_0} = -\frac{v}{fF_v}\left.\frac{\Delta f}{\Delta\xi}\right|_v - \frac{v}{IF_v}\frac{\Delta I}{\Delta\xi} + \frac{v}{NF_v}\frac{\Delta N}{\Delta\xi}$$

where F_v is the dimensionless partial derivative of the maximum clocking rate with respect to the supply voltage, $F_v = \frac{v\partial f}{f\partial v}$. The value of F_v can be estimated empirically for a selected technology, supply voltage and the selected 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 $\frac{\partial W}{\partial v}$ in the energy-efficiency formula is calculated as $\frac{\partial W}{\partial v} = \frac{IEf}{v}(E_v + F_v)$, where E_v is the 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, $E = \frac{Cv^2}{2}$. 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.

Substituting the calculated term into the energy-efficiency formula, and grouping terms in front of the partial derivatives, we arrive at the following criterion for energy efficiency:

$$-\frac{E_v}{F_v}\frac{1}{f}\left.\frac{\Delta f}{\Delta\xi}\right|_v - \frac{E_v}{F_v}\frac{1}{I}\frac{\Delta I}{\Delta\xi} + \frac{1}{E}\left.\frac{\Delta E}{\Delta\xi}\right|_v + \frac{F_v + E_v}{F_v}\frac{1}{N}\frac{\Delta N}{\Delta\xi} < 0$$

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

$$-\frac{E_v}{F_v}\frac{\Delta f}{f} - \frac{E_v}{F_v}\frac{\Delta I}{I} + \frac{\Delta E}{E} + \frac{F_v + E_v}{F_v}\frac{\Delta N}{N} < 0$$

Thus, the energy-efficiency criterion does not depend on the algorithm for enumerating architectural alternatives.

The increments of all quantities in the expression above appear in relative form, thus are dimensionless. This feature makes this formula easy to use as a negotiation basis between architects and circuit designers. For example, if $E_v = F_v = 2$, then 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%.

Although the energy-efficiency formula was derived for minimizing power, it is also valid for the reciprocal problem of performance maximization subject to the constant power constraint, $W(\xi, v) = \pi$. For some combinations of the values of E_v and F_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- per Watt" metrics are special cases of our energy-efficiency criterion, written in the integral form. For example, $E_v = 2$, $F_v = 1$ leads to

“MIPS^3 per Watt”; E_v = 2, F_v = 2 leads to “MIPS^2 per Watt”; $F_v \gg E_v$ leads to “MIPS per Watt”, whereas E_v = 2, F_v = 0.5 leads to “MIPS^5 per Watt.” Advantages of the new metric are its generality and the ability to calculate the parameter for every particular case, taking into account technology and circuit characteristics.

Effect of circuit and technology characteristics

Although theoretical formulas could be used to determine F_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 F_v, it is important to select functional blocks that can potentially be 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 the 4-read/4-write ports full-custom register file (just simulation); and a 2-read/2-write 16-entry semi-custom register file built with latches and multiplexors, all implemented in a 0.13um technology. For the energy analysis, we simulated all blocks with PowerMill [14], 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), and a clocking rate of 100MHz for all values of Vdd. We also used PathMill static timer for delay analysis. All derivatives were calculated by the 3-point formula.

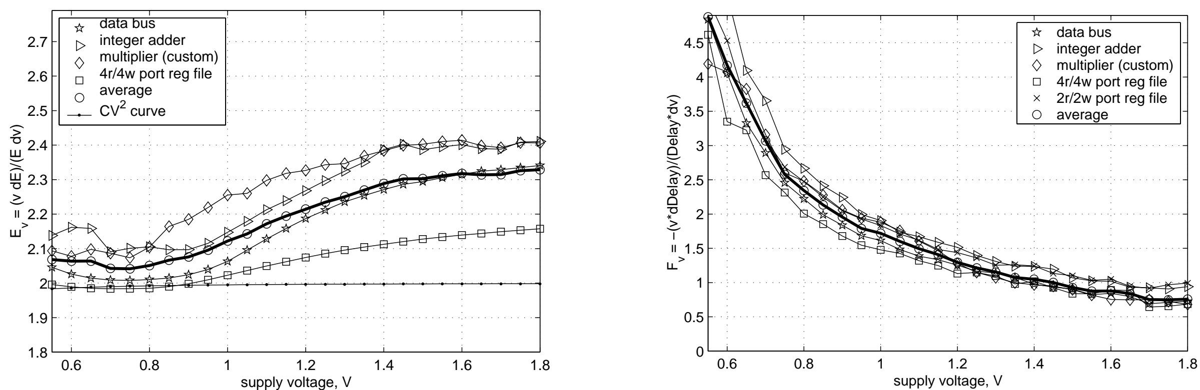


Figure 3: Simulation results for E_v and F_v

Figure 3a 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 3a shows that, for all the blocks, the value of E_v is higher than the value 2 that corresponds to the $E = Cv^2/2$ dependence. This super-Vdd^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.

Figure 3b shows simulation results for F_v. The curves on the graph correspond to the previously described blocks. For all blocks, F_v increases rapidly for low values of Vdd, especially as Vdd approaches the transistor threshold voltage. For high values of Vdd, F_v drops below unity because of the velocity saturation effect. For custom-designed blocks, F_v tends to be smaller

than for ASIC-synthesized blocks, especially at low values of Vdd, 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.

4. Processor architecture

We now describe the major features of the eLite DSP architecture, which have been derived from a thorough performance/power/applications/compiler analysis using the methodology outlined in the previous sections. This architecture 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 (SIMdD) operations on dynamically composed vectors of four elements, and exploitation of data parallelism through single-instruction multiple-packed data (SIMpD).
- 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 a 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.

As a result of its intended use as a target for an optimizing compiler, as well as due to 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; such code can be mixed with code generated by a compiler.

Figure 4 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 8 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 to/from 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, and ranges 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.

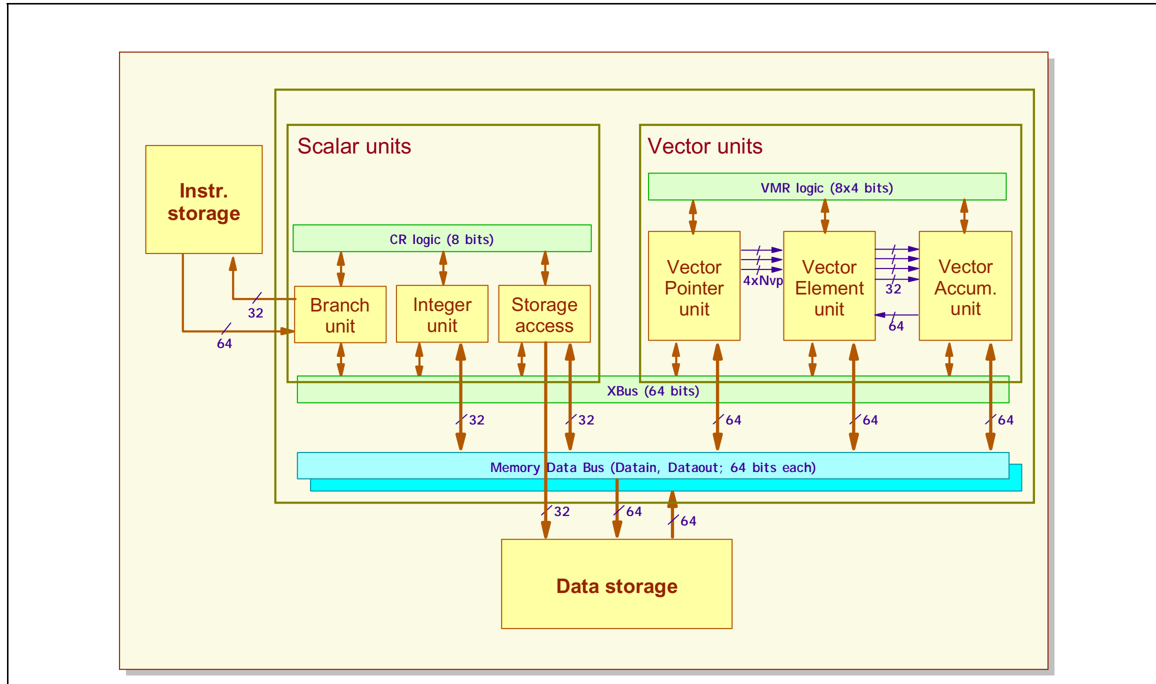


Figure 4: Block diagram of the eLite DSP architecture

The Integer Unit and Storage Access Unit correspond to scalar units, operating mostly on integer data. In contrast, the Vector Element Unit and the Vector Accumulator Unit operate on 4-element vectors in SIMD fashion, mostly containing fractional data (16-bit and 40-bit, respectively).

Program execution

A program in the eLite DSP architecture consists of a sequence of long-instructions words, each containing a 4-bit prefix (PX) and either one, two or three instructions, as depicted in Figure 5. A long-instruction is the minimum unit of program addressing possible, and is 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; all instructions contained in a LIW are dispatched for simultaneous execution, unless they are specified as executable in serial manner, as indicated 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 whose length is 30-bits specify additional opcode information in bits 28:29. These formats are illustrated in Figure 6.

Similar to RISC processors, no instruction in the eLite architecture can perform a computational operation on data in memory. Conversely, 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.

0	4	24	44	54	60 63
PX	OP1 (60 bits)				
PX	OP1H (20 bits)	OP2H (20 bits)	OP1L (10 bits)	OP2L (10 bits)	
PX	OP1 (20 bits)	OP2 (20 bits)	OP3 (20 bits)		
PX	OP1 (20 bits)	OP2H (20 bits)	OP3 (16 bits)		OP2L (4 b)

Figure 5: Long-instruction word (LIW) formats

	0	8	12	16	18	20	24	28 29
16-bit format	Opcode	Src/Dst	Src					
20-bit format	Opcode	Dst	Src	Src				
20-bit format with exp. opcode	Opcode	Dst	Src		XO1			
30-bit format	Opcode	Dst	Src	Immed			Pred	XO2
30-bit format with exp. opcode	Opcode	Dst	Src		XO1		Pred	XO2

Figure 6: Instruction formats

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 (see Figure 7) consists of loading many data elements from storage into the registers, in particular into the Vector Element Registers and Vector Accumulator Registers, and then operate on the contents of the registers with few intervening accesses to storage. Vector Element Registers are accessed indirectly through Vector Pointer Registers, so that vectors are dynamically composed from four arbitrary Vector Element Registers (SIMdD operation). Every Vector Element instruction specifies one or two Vector Pointer Registers, which in turn specify the four Vector Element Registers actually used by the instruction. In contrast, Vector Accumulator Registers contain four 40-bit elements each which are accessed simultaneously in the order they are stored in the VARs, and used in that same order as operands to Vector Accumulator Unit instructions (SIMpD operation). The elements from Vector Accumulator Registers, 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 are loaded from adjacent memory locations into arbitrary locations in the Vector Element Register file (VER), 16-bit operations are performed on data from arbitrary locations within the VER (SIMdD), and the results are placed into the 4x40-bit Vector Accumulator Register file (VAR), in packed manner (in a single register). 32-bit data can also be 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 on 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 can be placed into adjacent memory locations. Due to 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., “exposed pipeline” execution model). The pipelines are depicted in Figure 8; most instructions are processed in six stages, vector element instructions use one extra stage to read the Vector

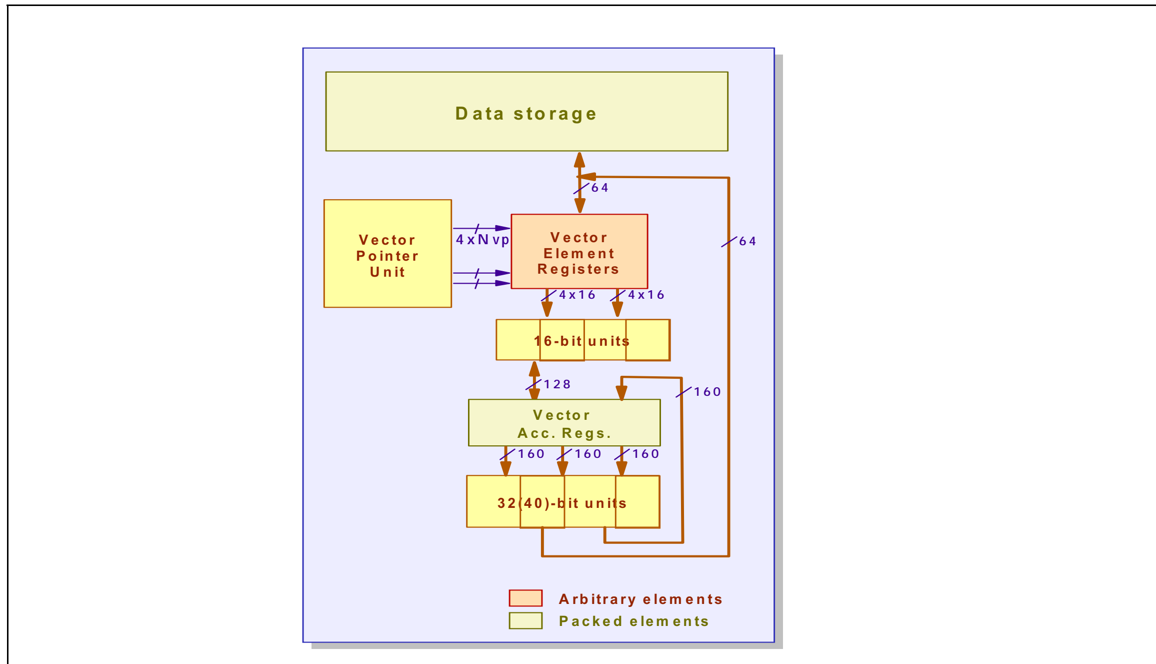


Figure 7: Vector programming model

Pointer Registers and the succeeding stage to read the Vector Element Registers, 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 their source registers at the same time, with the exception of Vector Element Registers which are read the following cycle after reading the associated Vector Pointer Registers. An instruction completes execution when its results are placed in their destination locations; instructions complete execution according to their individual latencies. Instructions contained wholly within a functional unit have the same latency, with the exception of branches which are resolved earlier; instructions in different units, or instructions that place the result on a register in a different unit, may exhibit different latencies.

	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

Figure 8: Execution pipelines

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.

Vector Units

The most salient computing resources within the eLite architecture are the three type of Vector Units, depicted in Figure 9. All these units can operate in parallel.

Figure 9: Vector units

Source operands for 16-bit Vector Element arithmetic and logic instructions (16-bit datapath) always originate from the Vector Element Register file. The destination of all 16-bit vector element operations is a Vector Accumulator Register (40-bit result). Each block within the 16-bit datapath consists of a multiplier and an ALU that performs arithmetic, logic, shift, and select operations on the contents of Vector Element Registers.

Every access to the Vector Element Register file is performed indirectly through a Vector Pointer Register. Each Vector Pointer Register contains four elements which are used as indexes into the Vector Element Register file, allowing the access to four independent Vector Element Registers. The Vector Pointer Registers can be automatically updated when used to access Vector Element Registers, and automatically implement "circular addressing" within a range of the Vector Element Register file.

Vector Accumulator Registers are used as source and destination for 40-bit Vector Accumulator arithmetic and logic instructions (40-bit datapath). Vector Accumulator Registers are also used as destination 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 Vector Element Registers. Regardless of the instruction type, Vector Accumulator Registers 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 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 Vector Pointer Register 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 Vector Pointer Registers used to access the Vector Element Registers containing the operands for the instruction, including circular addressing within the Vector Element Register file, adding two sets of four update operations to the computations specified in the LIW.

Consequently, such a single LIW actually specifies transformations on 25 data items, for a total of 25 basic operations.

5. Hardware design

A block diagram of the current prototype implementation of eLite is shown in Figure 10, 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 XBus, connects all functional units, allowing data movement among the various register files.

Referring to Figure 10, instructions flow from top to bottom. The on-chip instruction memory (IMEM) holds the long-instructions words, each 64-bit wide. A LIW is decoded every cycle, wherein the prefix 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, called 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 only to 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 busses, the vector functional units use several point-to-point connections to communicate among themselves. In particular the VPU sends 3 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 4 connections, each 40 bits wide, to be placed into the Vector Accumulator register 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 10. The aforementioned point-to-point connections are used frequently enough, and have a big 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/from the data memory, reading various processor state information, and handling external interrupt requests. The decoder 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.

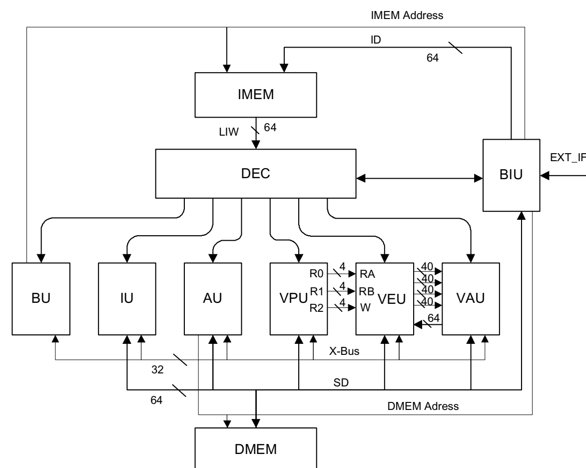


Figure 10: Block diagram of eLite implementation

6. 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) the development of a compiler that generates efficient code for exploiting the data and instruction-level parallelism capabilities in the processor; and

(2) 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 using *intrinsics* [15-16] and/or *C-language extensions* [17-18]. Intrinsics allow a programmer to explicitly specify certain instructions in the architecture which can not be easily described in a high-level language such as C. The use of intrinsics and C-language extensions suffers from disadvantages such as non-portable code which can not 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 this is a step backward regarding the use of portable high-level languages to program DSPs.

DSP applications and architectures are getting bigger and more complex. We believe that DSP applications in the near 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 and infer the meaning of sequences of C language constructs. The programmer should follow a few basic guidelines in-order to simplify this inference process.

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 these processors. Unlike typical VLIW architectures that use RISC primitives, the eLite DSP architecture does not suffer from serious “code-bloat” problems, due to its relatively short LIW and serialization capabilities. To increase code density even further, the compiler leverages architectural features such as SIMD instructions, combining instructions of different widths, and LIW-encoding to specify serialization.

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

Implementation

The eLite DSP compiler is based on *Chameleon*, the IBM VLIW Research Compiler originally designed for tree-VLIW processors [19-20]. Chameleon 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, Chameleon provides an excellent platform for developing and exploring new compilation techniques. (See [23-24] for more details on Chameleon).

New optimizations and enhancements were introduced to Chameleon in order to generate efficient code for the eLite DSP architecture. Figure 11 shows the structure of the compiler. After transforming the dependence-flow graph through a number of optimizations, a novel vectorization phase tries 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 to schedule and software-pipeline vectorized code along with scalar code. The vectorized code is scheduled and register allocated before emitting the final Assembly code.

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 inlining, software-pipelining, synthetic branch frequency based optimizations, as well as inter-procedural 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. 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, resource requirements, etc.

Vectorization

A vectorization phase has been developed to make efficient use of the SIMD instructions and the unique Vector Element Register file (VER) 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

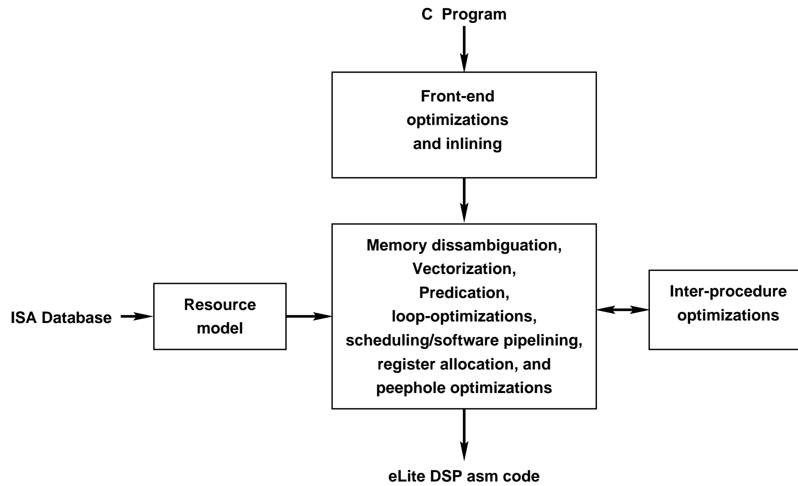


Figure 11: Block diagram of eLite compiler

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 [25] - 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 pre-loading data into the VER before the loop to eliminate redundant loads. An important stage of vectorization is setting up vector pointers. Flexibility of VER access through vector pointers allows efficient compilation of computations that involve non-consecutive data access patterns. This aggressive vectorization optimization is based on a set of innovative analyses, on top of the standard vectorization tests such as [26]. 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 [27-28], 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 integer register (IR), address register (AR), vector element register (VER), and vector accumulator register (VAR). The compiler is responsible for assigning each register Def to a specific register file/type, and for inserting explicit instructions moving the data between different register files. Given a register 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, possibilities of VER reuse, etc.

The partitioning process based on the above 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 [29].

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 non-uniform 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 turn-around time for architectural exploration.

Scheduling and register allocation

Generating efficient code for long non-interlocked pipelines is a big challenge by 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.

Inter-procedure optimizations

The compiler performs inter-procedural analysis for reducing the calling convention overhead by allowing callee to modify the calling conventions according to the intended usage of the parameters. Other inter-procedure 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-language code. Preliminary results show that, for a set of signal-processing kernels, the compiler generates vectorized code of comparable performance (well within a factor of 2) to carefully hand-coded assembly.

7. 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 the information describing the architecture, including instructions formats, operands for each instruction, and a machine-readable pseudo-code that describes the behavior of each instruction at each stage of the pipeline. All the tools have been built as semi-generic programs, providing a framework that takes most of the 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's operands (parameters). The instructions are bundled together to make Long Instruction Words (LIW). Finally, the sequence of LIWs is the output program code. The way that bits are arranged in an instruction or LIW is often non-trivial, requiring a mechanism to insert (when assembling) and extract (when disassembling) information from the instruction or LIW. Since all the details regarding the position of the fields are described in the ISA database, it is possible to automatically generate code that will handle these insertions and extracts. For this purpose, the concept of *Inserters* has been developed. An Inserter is an abstract interface that provides the two basic operations of insert and extract. Given a location within the program (e.g., LIW offset) and a value, the inserter places the value into 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 allows setting custom formats for various instructions without any modification to the assembler source. For example, an integer add instruction configuration is as follows: {"iadd", "OP=0x10", "$(RT),$(RA),$(RB)"} A typical use of this format would be something like `iadd r1,r2,r3`. However, it can easily be `iadd r1=r2+r3` by simply replacing the commas in the format string with the appropriate symbol, such as "$(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, make 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 (ISA) 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 has 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 additional state for each instruction instance used for storing local temporary variables. This allows building the instructions 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 either be a hardware related event, such as performing the operations that are associated with a specific cycle in the instruction's execution, or a software event artificially added to the ISA database as a helper function to 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 runtime, the simulator loads the binary code of the application program. It then decodes instructions according to the ISA specification. The instruction's 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, normally there are several instructions in flight, each in its appropriate stage.

Instructions are not the only part of ISA simulator that has 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.

8. Example of architecture/power/implementation trade-offs

The energy-efficiency metric derived in Section 3 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; context savings at interrupts; adding ports to various register files; bypasses on the vector element file; 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 Vector Element Register file.

Due to the number of read and write ports and the size of the Vector Element Register 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 unavailability of the bypasses increases the latency of the vector pipeline, which has a significant impact on the architectural performance of the processor.

Several architectural alternatives for Vector Element file bypasses were evaluated using the methodology described in Section 3. Some of the alternatives considered were:

A: No bypasses.

B: Element-wise bypass. Labelling the read ports RA0, …, RA3, RB0, …, RB3, and the write ports W0, …, W3, then an address match is determined as a result of comparing the concatenated indices (RA0,…, RA3) with (W0,…, 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.

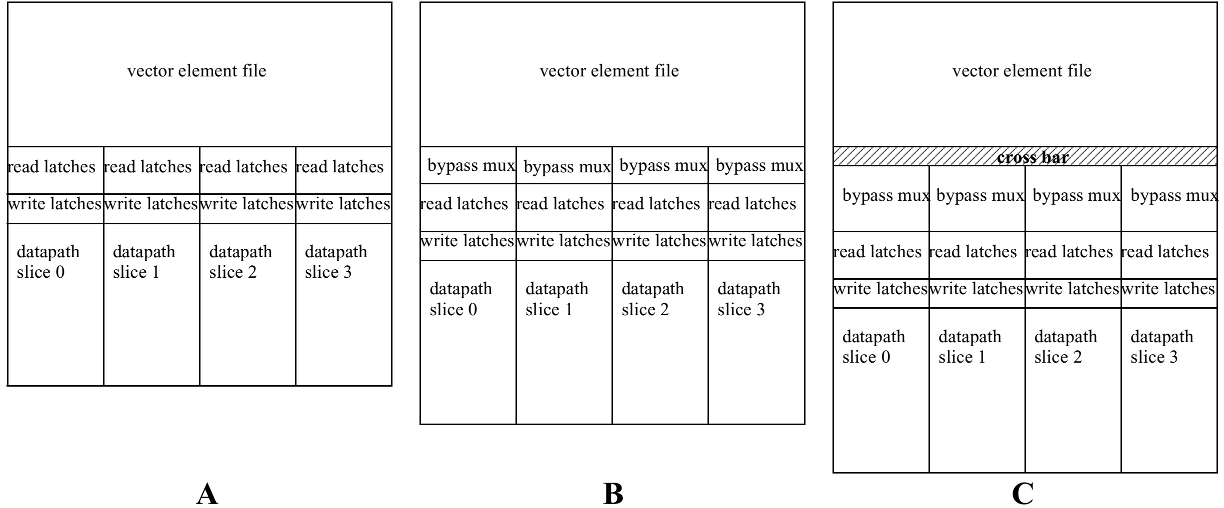


Figure 12: Alternative implementations of the vector element file bypass.

Figure 12 shows proposed floor plans 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 to 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 3 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, 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-bit each (two comparators for each read port).

In 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 cross bar of 2*4*16 = 128 lines. Each bypass multiplexor multiplexes data from 1+2*8=17 directions. A total of 8*4*2=64 index comparators with 9-bits each are required to control the multiplexors.

Table 1: Energy overhead of bypasses in the Vector Element file

Component	Element-wise bypass		Full bypass	
	Read access	Write access	Read access	Write access
Comparators	268fJ	536fJ	1072fJ	2144fJ
Bypass mux	69fJ	284fJ	69fJ	1112fJ

Table 1: Energy overhead of bypasses in the Vector Element file

Component	Element-wise bypass		Full bypass	
	Read access	Write access	Read access	Write access
Cross bar	-	-	-	5226fJ
Height increase	31fJ	31fJ	184fJ	184fJ
Total	368fJ	851fJ	1325fJ	8666fJ

Table 1 shows the major components of the energy overhead of the bypass implementations 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, the average energy overhead per instruction is 0.48nJ and 3.6nJ 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 50nJ. Thus, the relative energy-per-instruction overhead of the element-wise and full bypass implementations is $\frac{\Delta E}{E} = 1\%$, and $\frac{\Delta E}{E} = 7.2\%$, respectively.

The vector element file is potentially on the critical path of the processor, therefore adding bypasses might also impact 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 0.9V power supply, a latch with built-in multiplexor has a 40ps higher setup time than a latch without multiplexor. Thus, for a frequency of 250 MHz at 0.9 V, the clocking rate overhead of both implementations is $\frac{\Delta f}{f} = -1\%$.

Adding bypasses does not affect the dynamic instruction count, i.e. $\frac{\Delta N}{N} = 0$. The coefficients E_V and F_V can be estimated using the graphs on Figure 3. At 0.9V power supply, the values are $E_V = 2.07$ and $F_V = 2.0$. Then, the question about the energy efficiency of implementing bypasses is reduced to evaluating the expression $-1.04\frac{\Delta f}{f} - 1.04\frac{\Delta I}{I} + \frac{\Delta E}{E} < 0$. In order to be energy-efficient, the element-wise bypass implementation must result in an increase in the architectural performance (IPC) of at least 2%, whereas the full bypass implementation must result in an architectural performance improvement of at least 8%.

9. 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 FIR (Finite Impulse Response) filter performs filtering of speech signals in modern voice coders such as the ETSI GSM EFR/AMR or ITU G.729 [30-32]. 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 $m = 0,\ldots, M-1$, wherein 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" and 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

$$
\begin{aligned}
y(n) &= (h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \ldots + h(M-1)x(n-(M-1))) \\
y(n+1) &= (h(0)x(n+1) + h(1)x(n) + h(2)x(n-1) + \ldots + h(M-1)x(n+1-(M-1))) \\
y(n+2) &= (h(0)x(n+2) + h(1)x(n+1) + h(2)x(n) + \ldots + h(M-1)x(n+2-(M-1))) \\
y(n+3) &= (h(0)x(n+3) + h(1)x(n+2) + h(2)x(n+1) + \ldots + h(M-1)x(n+3-(M-1)))
\end{aligned}
$$

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

As it 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 the elements point to the same entry in the Vector Element Register 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 the elements point to consecutive entries in the Vector Element Register file and are incremented by 1 after each use, is suitable for addressing the samples.

At the end of inner loop, the Vector Pointer Register used to access the filter coefficients needs to be rewound by (M-1), so its elements will point again at $h(0)$ for the next iteration. Similarly, the Vector Pointer Register used to access the input data needs to be rewound by (M-1+4), so 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 at the penalty of larger code size. The resulting Assembly code for one iteration of the unrolled loop is shown in Figure 13; this implementation of the block FIR algorithm achieves asymptotically optimal performance (that is, 4 multiply/accumulate operations per cycle).

Vectorized Viterbi butterfly processing

An important application of the Viterbi algorithm [33-34] 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 memory-less AWGN channel [34].

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 [35]. The corresponding add-compare-select (ACS) operations [33-35] will be referred to simply as butterfly operations. A straightfor-

```
    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 13: Sample implementation of FIR filter

ward implementation of the butterfly operations requires two memory buffers for holding state-dependent data, which are used in a ping-pong fashion [36].

By using the Vector Pointer Registers in eLite, a more sophisticated approach based on rotated metric indexing [35] can be used, which allows 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 Vector Element Register file (VER) is large enough to hold the state metrics and branch metrics for the $M = 256$-state 3GPP Viterbi decoder [37], eliminating power-consuming transfers of metric data from/to memory.

Figure 14 shows a fragment of C code that processes four butterflies in parallel using in-place metric updating; this code matches nicely the SIMD features of the eLite 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 each tbr0 and tbr1 per 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 14).

The Viterbi decoder kernel in Assembly language is shown in Figure 15. 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 VER and aligned on a M-element boundary. The branch metric array is also allocated in 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 Vector Accumulator Register file (VAR). Of the four veaddn instructions, the two instructions targeting vector accumulator registers va0 and va1 (or va2 and va3) perform the metric addi-

```
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 14: 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,vm0   ||  ldvpu vp11,8(a9)
veaddn va1,(vp3),(vp13)
veaddn va3,(vp3),(vp12)  ||  vtmax (vp1),va2,va3,vm4   ||  ...
veaddn va2,(vp2),(vp13)  ||  vmshl
veaddn va0,(vp0),(vp10)  ||  vtmax (vp2),va0,va1,vm0
veaddn va1,(vp1),(vp11)
veaddn va3,(vp1),(vp10)  ||  vtmax (vp3),va2,va3,vm4
veaddn va2,(vp0),(vp11)  ||  vmshl
...
```

Figure 15: Software-pipelined version of Viterbi decoder kernel

tions for eight branches that are merging in four states. Each vtmax instruction performs a selection of four survivor metrics, which are written to VER for in-place metric updating, and outputs a 4-bit result indicating the selection to vector mask register vm0 or vm1. 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 ker-

nel while the vtmax instructions of the first basic kernel are being executed. As shown inFigure 15, tiling of two threads permits starting a basic kernel every four cycles, resulting in a 1 cycle/ butterfly performance. Note that register usage repeats after each pair of threads. Figure 15 also shows vector pointer initialization from tables in memory that are pointing to precomputed branch metrics in the VER.

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 trade-offs 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 type of designs. This effort reflects our understanding of the sound balance between performance, power consumption, programmability, development cost (hardware and software), and production cost (chip and system) that are required in the target field, namely digital communications. The energy-efficiency metric described in Section 3 provides a single and 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 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 provide a factor of 4 to 6 reduction in power consumption, when compared 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.*

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 enable achieving the performance requirements expected from next-generation digital communication applications. Some of the most salient features of the architecture and its associated implementations are concentrated in the ability to perform computations in SIMD manner, which are characteristic of contemporary DSPs. However, in contrast to other DSPs, the eLite architecture is characterized by a large number of internal registers, thereby reducing the need to access memory, as well as the ability to dynamically create 4-element vectors from arbitrary elements in the registers (SIMD with disjoint data), as well as operate on 4-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 haven 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 Vector Element

---

* Due to space constraints, not all the scalability features have been described in this paper.

Registers. We have also shown that the architecture is well-suited for contemporary data-intensive algorithms such as Viterbi decoding, wherein the large Vector Element Register file and the unique vector indexing capabilities allow keeping all state metrics 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 has already led to multiple patent applications, and is 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 suitable for their use in real-time environments, wherein the non-deterministic behavior of cache memories is an issue; (2) the investigation of mechanisms to further extend the abilities 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, wherein the innovations in flexible manipulation of data offered by eLite's SIMdD model of computation, low-power architectural features and implementation, and the benefits in programmability, represent major advantages over existing approaches.

11. References

[1] J. Eyre, "The digital signal processing derby," *IEEE Spectrum*, June 2001.

[2] J. Glossner, J.H. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, M. Ware, "Trends in compilable DSP architectures," *2000 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 181-199, October 2000.

[3] StarCore, *SC140 DSP core reference manual,* December 1999.

[4] Texas Instruments, Inc., *TMS320C6000 CPU and instruction set reference guide,* 2000.

[5] J. Tomarakos, C. Duggan, and S. Steyerl, 32-bit SIMD Sharc DSP processor architecture for digital audio signal processing applications." AES 106th Convention, Germany, May 1999.

[6] D. Brooks, P. Bose, et al., "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," *IEEE MICRO*, Vol. 20, No. 6, pp. 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 Journal of Solid-State Circuits*, Vol. 27, No. 4, pp. 473–484, April 1992.

[9] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 9, pp. 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*, pages 8–11, October 1994.

[11] V. Zyuban and P. Kogge, "Optimization of high-performance superscalar architectures for energy efficiency," *IEEE Symposium on Low Power Electronics and Design*, August 2000.

[12] V. Zyuban and P. Strenski, "Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels," *2002 International Symposium on Low Power Electronics and Design*, pp. 166-171, July 2002.

[13] S. Kosonocky et al., "Low Power Circuits and Technology for Wireless Digital Systems," submitted to *IBM Journal of Research and Development*, 2002.

[14] http://www.synopsys.com

[15] R. Stallman, "Using and porting GNU CC," Free Software Foundation, version 2.7.2.1, June 1996.

[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.

[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*, pp. 1065-1068, 1990.

[18] B. Krepp, “DSP-Oriented extensions to ANSI C,” *Proceedings of the International Conference on Signal Processing Applications and Technology* (ICSPAT), pp. 658-664, 1997.

[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 IFIP WG 10.3 Working Conference on Parallel Processing*, Italy, pp. 3-21, 1988.

[21] K. Pingali, M. Beck, R. Johnson, M. Moudgill and P. Stodghill, “Dependence flow graphs: an algebraic approach to program dependencies,” *Proceedings of the 18-th annual ACM Symposium on Principles of Programming Languages* (POPL), pp. 67-78, 1991.

[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 Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp.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*, pp. 332-335, June 1997.

[24] J. H. Moreno, M. Moudgill, K. Ebcioglu, E.R. Altman, C. B. Hall, R. Miranda, S.-K. Chen and A. Polyak, “Simulation/evaluation environment for a VLIW processor architecture,” *IBM Journal of Research and Development*, Vol. 41, No. 3, pp.287-302, 1997.

[25] S. Carr and K. Kennedy, “Improving the ratio of memory operations in floating-point operations in loops,” *ACM Transactions on Programming Languages and Systems*, Vol.16, No.6, pp. 1768-1810, November 1994.

[26] G. Goff, K. Kennedy, C-W. Tseng, “Practical dependence testing,” *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (PLDI), pp. 15-29, Toronto, Ontario, June 1991.

[27] M. Wolfe, *High performance compilers for parallel computing*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[28] D.F.Bacon, S.L. Graham, and O.J. Sharp, “Compiler transformation for high-performance computing,” *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345-420, 1994.

[29] M. Moudgill and A. Zaks, “Minimizing inter-file transfers in architectures with separate address registers,” *IBM Research Report RC21884*, November 2000.

[30] 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.

[31] 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.

[32] *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.

[33] S. Lin, D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1983.

[34] R. Johannesson, K. Sh. Zigangirov, *Fundamentals of Convolutional Coding*, IEEE Press, 1999.

[35] M. Biver, H. Kaeslin, C. Tommasini, “In-place updating of path metrics in Viterbi decoders”, *IEEE J. of Solid-State Circuits*, Vol. 24, No. 4, pp. 1158-1160, August 1989.

[36] Motorola, Inc., and Agere Systems, “How to Implement a Viterbi Decoder on the StarCore SC140”, Application Note ANSC140VIT/D, July 18, 2000.

[37] 3rd Generation Partnership Project (3GPP), *Technical Specification 3G TS 25.212, Multiplexing and channel coding (FDD)*, March 2000.