A HIGH-PERFORMANCE EMBEDDED DSP CORE WITH NOVEL SIMD FEATURES

Jeff H. Derby[1] and Jaime H. Moreno[2]
IBM Communications Research and Development Center
[1]IBM Corporation, Research Triangle Park, NC 27709
[2]IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

Abstract– A low-power, high-performance, compiler-friendly DSP core has been under development in the IBM Communications Research & Development Center, as part of its eLite DSP project. This DSP incorporates instruction-level parallelism through the packing of multiple instructions in 64-bit long-instruction words, while data-level parallelism is realized through the use of SIMD techniques, such that SIMD operations can be applied to both dynamically composed vectors and packed vectors. Dynamic composition of vectors is made possible through the use of a vector pointer mechanism, which permits the addressing in a very flexible way of groups of four 16-bit elements in a large, multiport, scalar register file.

This paper provides an overview of the architecture of this DSP core, with a focus on its SIMD features. We describe these features in some detail and discuss how they are used, with a block FIR filter and a radix-4 FFT taken as examples.

I. Introduction

A low-power, high-performance, compiler friendly DSP core has been under development in the IBM Communications Research & Development Center, as part of its eLite DSP project. It is intended for use as an embedded DSP core in systems-on-chip, targeted at communication and media-related applications.

The eLite project is an ongoing effort within the IBM CRDC that is advancing the state-of-the-art in power-efficient, high-performance, programmable DSP architectures as well as in methodologies for implementing such architectures. 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 meet these usually conflicting optimization goals, the design of the eLite DSP architecture and its implementations cover aspects ranging from algorithms, applications, and high-level-language compiler, down to circuit-level technology. Indeed, the project view has been as an explicit hardware-software codesign of the architecture with an optimizing compiler, and also as an explicit power-performance codesign of the architecture with its implementation. Details of these codesign methodologies can be found in [1],[2].

The eLite DSP architecture that is the result of the process outlined above is a load/store, RISC-like architecture that incorporates both instruction-level and data-level parallelism. Instruction-level parallelism is realized through the packing of multiple independent instructions in 64-bit long-instruction words (LIWs) and through the use of implicit operations in certain instructions that reference data memory or internal register files. Data-level parallelism is realized through the use of single-instruction multiple-data (SIMD) techniques, such that SIMD operations can be applied both to dynamically composed four-element vectors and to packed four-element vectors.

The focus of this paper is the SIMD behavior of the eLite DSP architecture. The technique used to realize data-level parallelism in this architecture differs from those employed in RISC processors with SIMD features. In the VMX (also called AltiVec[1]) extensions to the PowerPC[2] architecture [3], a "vector permute unit" is used to rearrange the order of data elements packed in a vector register, based on data elements packed in a second vector register. In contrast, the eLite DSP employs a set of four indices contained in a vector pointer register to address four data elements in a large *scalar* register array, the ***Vector Element File*** (VEF), thereby dynamically composing vectors for processing. The eLite DSP's SIMD structure also differs markedly from the SIMD-like capabilities that appear in current-generation DSP cores (see, e.g., [4],[5]). These DSPs often employ complex interfaces to data memory to enable flexibility in composing groups of data elements for processing. In contrast, eLite requires only a single, simple interface to data memory because of the structure of the VEF and the vector pointer mechanism. These and other novel features of the eLite DSP's SIMD architecture will be described in some detail.

Section II presents a brief overview of the eLite architecture. In Section III, we provide a more detailed discussion of the SIMD units and an appropriate programming model for using them. Algorithm kernels from a block FIR filter and an FFT are described in Sections IV and V, respectively. Finally, conclusions are stated in Section VI.

II. Architecture Overview

Fig. 1 shows a block diagram of the eLite DSP architecture. As can be seen, there are a number of functional units, including:

- **Branch unit:** generates the storage address for the next LIW to be fetched from memory; also performs logical operations on 8 single-bit Condition Registers, which are used for conditional branches and predication.

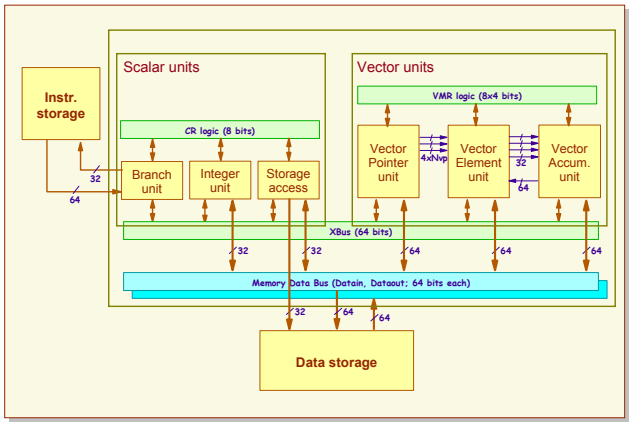


Fig. 1. Block diagram of the eLite DSP architecture.

1. AltiVec is a trademark of Motorola.
2. PowerPC is a trademark of International Business Machines.

- **Integer unit:** operates on data in 16 Integer Registers.
- **Storage access unit:** interacts with data memory to transfer data between internal registers and memory, and performs operations on data in 16 Address Registers.
- **Vector Pointer unit:** performs operations on 16 Vector Pointer Registers, which are used to access the contents of Vector Element Registers.
- **Vector Element unit:** 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:** performs operations on data in 16 Vector Accumulator Registers, including reduction operations on the elements of a vector.

The Integer Unit and Storage Access Unit correspond to scalar units, operating on 32-bit integer data. In contrast, the Vector Element Unit and the Vector Accumulator Unit operate on 4-element vectors in SIMD fashion (16-bit and 40-bit, respectively), containing fractional or integer data. Note also that each unit has its own register files. As a rule, results of an operation in a unit are written back to that unit's register file. The exception to this rule is that results of operations in the Vector Element Unit are always written to the Vector Accumulator register file.

Defined in the eLite DSP's instruction set are 16-bit, 20-bit, 24-bit, 30-bit, and 60-bit instruction formats. Instructions are packed into 64-bit LIWs. An LIW contains a 4-bit prefix and up to three instructions. Possible combinations in an LIW are: three 20-bit instructions; one 20-bit, one 24-bit, and one 16-bit instruction; two 30-bit instructions; one 60-bit instruction. Nominally, the instructions in an LIW are issued to the core in the same cycle. However, a serialization mechanism, controlled by the LIW prefix, permits instructions in a single LIW to be issued in successive cycles.

The scheduling of instructions, i.e. how instructions are packed in LIWs, is static, being determined by the compiler or by the assembly-language programmer. Instruction scheduling must take into consideration the utilization of resources throughout the pipeline and the data dependencies with dependent instructions, given that the pipeline is exposed. The pipelines are depicted in Fig. 2; most instructions are processed in six stages, vector element instructions use one extra stage to read the Vector 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 in a register in a different unit, may exhibit different latencies. Note that the first three stages in all pipelines, not shown in Fig. 2, are two instruction fetch stages and a decode stage.

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 that is evaluated dynamically. The mask is specified in a Vector Mask Register.

	4	5	6	7	8	9
Base	RD	EX	WR			
Other unit target	RD	EX	XFR	WR		
Vector element instructions	RD_VP	RD_VE	EX1	EX2	WR	
Load instructions	RD	AG	XFR1	RD_M	XFR2	WR

Fig. 2. Execution pipelines.

III. Vector Units

Several issues arise in incorporating efficient and effective data-level parallelism in a DSP architecture. One has to do with providing sufficient flexibility in how groups of data elements are selected and accessed for processing in the core. A second has to do with the different data widths that appear at different points in a DSP algorithm data path, e.g. 16-bit data at multiplier inputs and (32+guard)-bit data at multiplier outputs and accumulator inputs. A third issue has to do with compact encoding of operations being applied to a group of data elements.

At the same time, there are several characteristics that are shared by many DSP algorithms and that provide hints about potentially useful approaches to data-level parallelism. Many (but certainly not all) DSP algorithms have implementations in which single operations may be applied simultaneously to several sets of data elements, i.e. they appear well-matched to SIMD architectures. Moreover, many (but certainly not all) DSP algorithms have implementations that involve extensive data reuse.

As noted at the outset, the eLite DSP employs a SIMD approach to data-level parallelism. In fact, two SIMD computational units are included that are effectively connected in cascade, as shown in Fig. 3. The Vector Element Unit (VEU) operates on pairs of 4-element vectors with 16-bit elements. The elements of each vector are selected from the Vector Element File (VEF), a large, multiport, scalar register file containing $2^{N_{VP}}$ independently addressable 16-bit elements. N_{VP} is a parameter associated with implementations of the architecture, with an architectural limit value of 12 (4096-element VEF) and typical values for initial implementations in the range of 6 to 9 (64-element to 512-element VEF). The selection of elements from the VEF to make up the VEU input vectors is indirect, via indices specified in Vector Pointer Registers (VPRs), as described below. Thus the vectors operated on by the VEU are dynamically composed, with entries selected from the large array of 16-bit data elements in the VEF. Moreover, extensive reuse of data in the

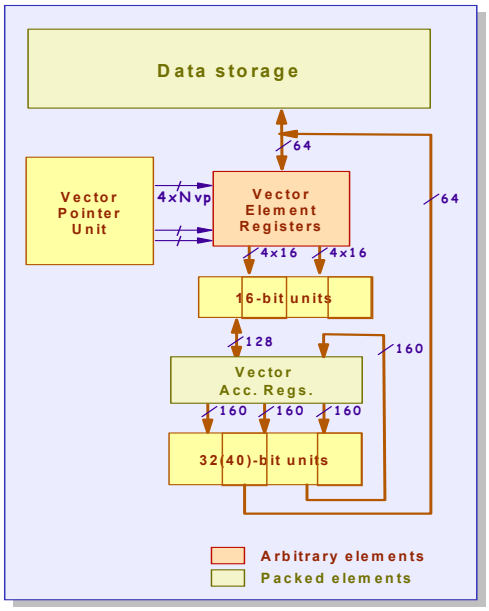


Fig. 3. Vector units and vector programming model.

VEF is possible by appropriate configuration and updating of indices in the VPRs. The VEU is where multiplication is performed, and so the output of the VEU is nominally a vector with four 32-bit elements. Rather than parsing these into 16-bit pieces that are placed into the VEF, they are maintained as complete 32-bit elements and are placed into the Vector Accumulator File (VAF), which is part of the Vector Accumulator Unit (VAU). In fact, all VEU operations target the VAF. The VAU operates on pairs of 4-element vectors with 40-bit elements. The input vectors and output vector of a VAU operation are Vector Accumulator Registers (VARs) in the VAF, each of which holds a single vector with four 40-bit data elements. There is essentially no capability in the VAU for rearranging elements within a VAR, so the vectors operated on by the VAU are "packed".

It remains to describe how the VPRs are used to index elements in the VEF. A VPR contains four indices, each in the range of 0 to $2^{N_{VP}} - 1$. In addition, associated with each VPR is a mechanism for incrementing the indices using modulo arithmetic. The default use of a VPR involves the following steps:

1. use the four VPR indices to select four elements from the VEF; then
2. update the four indices by adding a stride value to each index, with the addition performed using modulo arithmetic.

There are also alternate forms in which the VPR indices are not updated. These options are available for all instructions that use VPRs to address the VEF, including:

- VEU instructions, which reference two VPRs (one for each input vector to the operation)
- load/store vector element instructions, which reference a single VPR (for the target VEF elements to be loaded or stored)
- move from VAF to VEF instructions, which reference a single VPR

Note that in all cases the index updating takes place in the Vector Pointer Unit (VPU), itself a SIMD functional unit, and the updated indices are written back into the same VPR.

Finally, it is useful to consider the net parallelism possible in an LIW given the SIMD features described above. A possible (and common) set of three instructions in a single LIW is: a load VEF instruction with update of the address register; a VEU multiply; and a VAU add. The first performs 9 operations: load four elements in the VEF, update four VPR indices, and update an address register. The second performs 12 operations: multiply four pairs of elements from the VEF, and update two sets of four VPR indices. The third performs four operations: add four pairs of elements in the VAU. This single LIW thus specifies 25 operations in all.

IV. An FIR Example

We have coded a real block FIR filter on the eLite DSP with four output samples computed in parallel. Given an K-tap filter with impulse response $h(k), k = 0, 1, \ldots, K-1$, and input $x(n)$. Four successive output samples $y(n-i), i = 0, 1, 2, 3$, are computed as:

$$y(n-i) = h(0)x(n-i) + h(1)x(n-i-1) + \ldots + h(K-1)x(n-K-i+1) \quad (1)$$

These four outputs are accumulated in the four elements of a single VAR. As an example, consider a filter with $K = 16$, and with a total of 40 output sample values computed each time the filter is executed, so that 56 input sample values are used. Assume for now that these values and the filter coefficients are already loaded in the VEF, with the input samples at indices $0, 1, \ldots, 55$ and the coefficients at indices $64, 65, \ldots, 79$. A segment of the code for this filter is shown in Fig. 4. The four lines of "inner loop" shown in the figure accumulate four successive output sample values in the elements of VAR VA0. Referring to the figure, the '`vefmul`' instructions are vector multiplies performed in the VEU, the '`vaadd`' instructions are vector adds performed in the VAU, and instructions grouped on the same line are issued in parallel. We will look at the vector pointer usage in a moment, but first note the following:

```
vefmul va1,(vp0),(vp1) || vasubf va0,va0,va0          (1)
vefmul va2,(vp0),(vp1)                                (2)
vefmul va3,(vp0),(vp1)                                (3)
inner_loop:
vefmul va4,(vp0),(vp1) || vaadd va0,va0,va1 || bnz ct0,inner_loop   (4)
vefmul va1,(vp0),(vp1) || vaadd va0,va0,va2           (5)
vefmul va2,(vp0),(vp1) || vaadd va0,va0,va3           (6)
vefmul va3,(vp0),(vp1) || vaadd va0,va0,va4           (7)
```

Fig. 4. Segment of FIR filter code.

- The VEU pipeline is deeper than the VAU pipeline (see Fig. 2), so the results of a VEU instruction can only be used by a VAU instruction issued at least three cycles later. Thus the first three '`vefmul`'s are peeled out of the inner loop. The results of the '`vefmul`' at line 1 are available in VA1 for the '`vaadd`' at line 4.
- Because of branch latency and the exposed pipeline, the three instructions following the branch ('`bnz`') at line 4 are always executed. Thus the loop consists of four LIWs.
- All the instructions shown in the figure are in 20-bit format.

We consider now the VPR usage. In the example, VP0 indexes the coefficients, and VP1 indexes the input samples. We have the indices in VP0 initialized to {64,64,64,64} and those in VP1 initialized to {0,1,2,3}, both set with stride of 1. The first '`vefmul`' (at line 1) thus computes the four products $h(0)x(n)$, $h(0)x(n-1)$, $h(0)x(n-2)$, and $h(0)x(n-3)$ in parallel; these are the first terms in the sums for $y(n)$, $y(n-1)$, $y(n-2)$, and $y(n-3)$ in Eqn. (1). The VPR indices are updated using the default procedure outlined in Section III. Thus when the second '`vefmul`', at line 2, is executed, the indices in VP0 are {65,65,65,65}, the indices in VP1 are {1,2,3,4), and so this instruction computes in parallel the four products that are the second terms in the sums for $y(n)$, $y(n-1)$, $y(n-2)$, and $y(n-3)$ in Eqn. (1). When the inner loop completes, all 16 products for each of these four outputs have been accumulated.

A few additional comments regarding this example are in order here. First, to begin computation of the next four output samples, the VPR indices need to be rewound. The indices in VP0 need to be reset to {64,64,64,64}; the indices in VP0 are reset as a consequence of the modulo arithmetic. The indices in VPR1 need to be rewound to {4,5,6,7}; one way to accomplish this is with an explicit VPR increment instruction in the epilog of the inner loop.

Second, the example assumes that the VEF has been preloaded with the coefficients and input data. It is possible to load these quantities into the VEF during the computation of the first four output samples. Note that there are three open 20-bit LIW slots in the inner-loop code in Fig. 4. These can be filled with 20-bit "load vector element" instructions, which will perform the desired loading of the VEF from data memory. Once these loads are complete, all the necessary data is in the VEF, and the successive sets of four outputs can be computed with no additional loads from data memory. Thus full advantage is taken of the rich opportunities for data reuse in the FIR structure.

Finally, details of the completion of the inner loop (e.g. compensating for the peeling of the first two '`vefmul`' instructions out of the inner loop), the remainder of the outer loop, the initialization of the VPRs, and the storing of the computed outputs are not shown in the example. Note, however, that while the computed outputs would normally be written back to data memory, it may make sense to save them in the VEF if they are to be used as inputs to the next block in the overall function being implemented.

V. An FFT Example

We have investigated the implementation on the eLite DSP of a number of algorithms that appear to permit extensive leverage of data reuse but for which the data access patterns are far more complicated than for block FIR discussed above. We used

the results of these investigations to drive the architectural optimization of the vector units, and in particular of the vector pointer architecture and the vector instruction formats. The algorithms considered in these studies included the Viterbi decoder and the FFT. For a description of the Viterbi decoder on the eLite DSP, see [1],[2]. We provide a brief discussion of the FFT here.

A number of approaches to implementing the FFT were considered. These included: radix-2 and radix-4; decimation-in-time (DIT) and decimation-in-frequency (DIF); with multiplications at the butterfly inputs and with multiplications at the butterfly outputs; where the data and twiddle-factor arrays fit completely in a "realistic" VEF (no more than 512 elements) and where they do not. All implementations were in-place, in that the output data array is returned to the same locations in data memory from which the input data array is taken.

We now summarize a particular FFT implementation, namely a radix-4 256-point complex FFT with the twiddle-factor array kept in the VEF and the data array read from data memory and written back to data memory in each stage. Note that for a 256-point FFT both arrays do not fit in a 512-element VEF. We chose to keep the twiddles rather than the data in the VEF because the access patterns for the twiddle array are more complicated and so the VPR indexing mechanism provides a more significant benefit.

The structure implemented is a straightforward DIT structure. This structure can be derived in several ways, for example from the methodology employed in [6]. It employs radix-4 "butterflies", referred to here as "spiders", with multiplications preceding the spiders as shown in Fig. 5. Referring to the figure, $\mathbf{T}_4$ is the 4-point DFT matrix as shown, and the twiddle factors are such that $W = e^{-j2\pi/N}$ with $N = 256$ and k determined by the stage and the position in the stage at which a particular spider is located. The dots in the figure indicate multiplication by the twiddle factors. Reasons for choosing the DIT structure include:

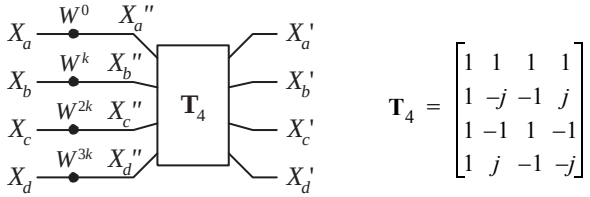
- the loading of the twiddle-factor array into the VEF, and also the radix-4 "bit-reversed" indexing of the data, are performed in the first stage, where there are no multiplications;
- placing the multiplications at the spider inputs, ahead of the adds and subtracts, is most consistent with the natural dataflow through the vector units (recall Fig. 3).

In our implementation, the inner-loop kernel performs four spiders with associated multiplications in 22 cycles. This kernel is structured with three parallel threads:

1. multiplications, performed in the VEU, with results placed in the VAF; there are 16 real multiplies per spider, so 16 cycles are required for the real multiplies for four spiders.
2. adds/subtracts to complete the complex multiplications and to implement the spiders, performed in the VAU, with results placed in the VAF; there are 22 such adds/subtracts per spider, so 22 cycles are required for four spiders.
3. loads of the data array to the VEF and stores of the data array from the VAF; eight load instructions and eight store instructions are required for four spiders.

Software pipelining is used to deal with pipeline latencies. The parallel execution of the three threads is organized as follows:

- loads of the data for the next set of spiders execute in parallel with completion of the computation for the current set.
- stores of the data for the current set of spiders execute in parallel with the beginning of the computation for the next set.



$$\mathbf{T}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}$$

Fig. 5. A radix-4 spider, preceded by twiddle-factor multiplications.

These two groups of operations execute sequentially in the inner loop. Note that the loads and stores cannot execute in parallel, because only one Address Unit instruction can be issued per cycle and there is only one 64-bit port to data memory. It can be seen, however, that there is no limitation on performance imposed on the inner loop as a result. The loop cycles are bounded by the number of required VAU operations, and in fact the number of loop cycles taken is equal to the number of VAU instructions.

Unfortunately, a thorough discussion of our implementation and the associated code for the FFT is beyond the scope of this paper. We should, however, point out the following in particular with respect to usage of the VPRs and the VEF:

- The VEF is used as a buffer for the data read from memory. In fact, two circular buffers are created in the VEF, one for the real parts and one for the imaginary parts, each holding 32 16-bit data elements. These buffers serve two purposes. First, they permit data for the next set of spiders to be loaded while data for the current set is being processed and is still in the VEF. Second, the VPRs can be configured to move through these buffers in such a way that the real and imaginary parts of the spider outputs (16 outputs for four spiders) are properly organized in VARs for the write back to data memory.
- The radix-4 "bit-reversed" indexing is performed by appropriate configuration of the VPRs used to access the data from the buffers in the VEF for the first-stage processing. Thus the input data array can be read from memory in natural order into the VEF, accessed in "bit-reversed" order from the VEF, with the results written back to data memory in what is essentially the natural order for the FFT output array.
- The access patterns for the twiddle-factor array in the VEF, i.e. dealing with the variation of k (see Fig. 5) for spiders within a stage and then stage-to-stage, are managed through the modulo arithmetic on VPR indices and by appropriately varying the VPR strides.

The implementation of the 256-point complex FFT we have outlined executes in 1503 cycles.

VI. Conclusion

This paper has presented an overview of the architecture of the eLite DSP, with a particular focus on its SIMD characteristics. The eLite DSP architecture incorporates multiple SIMD execution units, nominally used in a cascade fashion. A Vector Element Unit operates on vectors of four 16-bit elements, with these vectors dynamically composed from a large, scalar register file through indexing in Vector Pointer Registers, and with the results of VEU operations placed in vector registers in the Vector Accumulator Unit with 40 bits per element. Aspects of the operation of these SIMD units were described in the context of FIR and FFT examples.

References

[1] J. H. Moreno et.al., "An innovative low-power high-performance programmable signal processor for digital communications", *IBM J. Res. Devel.,* to appear.

[2] http://www.research.ibm.com/elite/.

[3] M. S. Schmookler et.al., "Low-power, high-speed implementation of a PowerPC microprocessor vector extension", in *Proc. 14th IEEE Symp. Comput. Arith.,* Adelaide, Aust., Apr. 1999, pp. 12-19.

[4] StarCore, *SC140 DSP Core Reference Manual,* December 1999.

[5] Texas Instruments, Inc., *TMS320C6000 CPU and Instruction Set Reference Guide,* 2000.

[6] H. Sloate, "Matrix representations for sorting and the fast Fourier transform", *IEEE Trans. Circ. Syst.,* vol. CAS-21 no. 1, pp. 109-116, Jan. 1974.