A NEW PARALLEL DSP WITH SHORT-VECTOR MEMORY ARCHITECTURE[‡]

Jose Fridman and William C. Anderson

Analog Devices, Inc.
One Technology Way
Norwood MA, 02062, USA
http://www.analog.com

ABSTRACT

This paper presents a new highly-parallel DSP architecture based on a short-vector memory system developed at Analog Devices, Inc. This DSP incorporates for the first time in an embedded processor a number of techniques found in general-purpose computing, such as branch prediction, deep and fully-interlocked pipeline, and SIMD instruction execution. By means of its short-vector high-bandwidth memory system it is able to deliver sustained performance that is close to its peak computational rates of 1.5 GFLOPS (32-bit floating-point), or 6 BOPS (16-bit fixed-point).

I. INTRODUCTION

In recent months, several new multi-datapath and pipelined *Digital Signal Processors* (DSPs) have been announced. This new generation of DSPs is taking advantage of higher levels of integration than was available for their predecessors, and are incorporating multiple execution units on a single core, as well as building deep execution pipelines. For comprehensive analysis on the state-of-the-art of DSP chips, see [1,2].

In order to sustain the high computation rates of cores with multiple execution units, memory subsystems must scale proportionately. In this paper we describe a new parallel DSP architecture from Analog Devices, Inc., called TigerSHARC™, and focus on the computational aspects of its core and on its on-chip memory architecture. Our solution to the high bandwidth demands of this parallel DSP core is based on an on-chip memory architecture that relies on short-vector processor techniques.

In addition to the architectural description, we also present an application example of a FIR filter. With this example, we illustrate that a large class of DSP algorithms (namely structures with register delays like FIR and IIR filters) presents a data alignment problem when mapped into vector-oriented processors. We show how specific alignment hardware can be used to mitigate the alignment problem, and that the proper choice of algorithmic map is required for high-efficiency solutions. In addition to data alignment, we show that the full SIMD dispatch mechanism, although very effective in simple vector and matrix operations, may be overly restrictive when applied to this class of DSP algorithms, and that non-SIMD execution is required for high efficiency.

The first device will deliver 1.5 GFLOPS (Giga Floating-Point Operations Per Second), or 6 GOPS of 16-bit arithmetic (Operations Per Second), and sustain an internal data bandwidth of 12 GB/s (Giga Bytes per second), at a clock rate of 250 MHz.

Recent trends in DSPs have also shown that techniques found in general-purpose computing have been migrating to DSP and embedded computing. The TigerSHARC has a number of general-purpose mechanisms that have not been incorporated on any DSP to date. The most significant aspects of this new DSP architecture are:

1. Register-based load-store static superscalar dispatch mechanism, with instruction parallelism determined prior to run time under compiler and programmer control.
2. Support for multiple data types, including IEEE single precision floating-point, 32-bit, 16-bit and 8-bit fixed-point.
3. Parallel arithmetic operations for 2 floating-point MACs or for 8, 16-bit MACs per cycle, with an optional *Single-Instruction Multiple Data* (SIMD) execution mechanism.
4. Highly parallel short-vector oriented memory architecture.
5. A total of 128 architecturally-visible and fully interlocked registers
6. 8-stage fully-interruptible pipeline, with a regular 2-cycle delay on all arithmetic and load/store operations, and a 128-entry, 4-way set-associative Branch Target Buffer.

This paper is organized as follows. In section II we present a brief description of the architecture. In section III we present DSP benchmark figures and an application example, and a summary in section IV.

II. ARCHITECTURAL DESCRIPTION

Figure 1 shows a block diagram with the major components of the architecture. Each computation block on the lower left of Figure 1 (CBX and CBY) consists of a 32-word general purpose register file, an ALU, a multiplier, a generalized bit manipulation unit (shifter), and a *Data Alignment Buffer* (DAB). The computation blocks constitute the two main data paths. Each

[‡] Work sponsored in part by DARPA-ITO contract number N66001-96-C-8610, under the Embeddable Systems Program.

computation block has two 128-bit ports that connect with the internal bus system. The bus system in turn consists of three, 128-bit buses, and acts essentially as a large crossbar connection between all the major blocks.

In the upper part of Figure 1 there are two integer units (JALU and KALU), collectively called IALU, which function as generalized addressing units, each with a general-purpose 32-word register file.

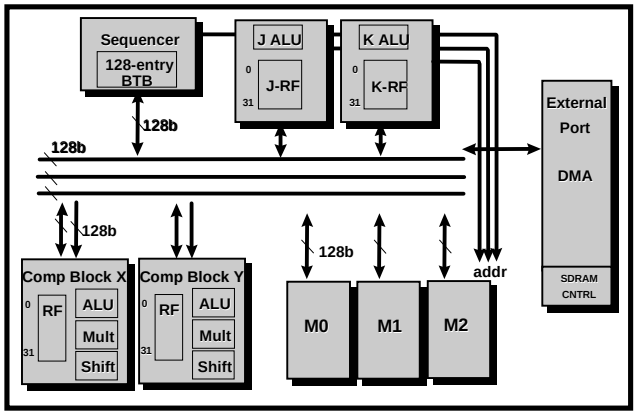


Figure 1: Block diagram showing the major sub-systems of the DSP.

There are three internal SRAM memory blocks, each with a 128-bit port into the internal bus system. These are shown as the three blocks labeled “M0” “M1” and “M2” in Figure 1.

The sequencer is shown in the upper-left of Figure 1, along with a 128-entry, 4-way set-associative *Branch Target Buffer* (BTB). The three internal buses provide a direct path for instructions into the sequencer, as well as two independent paths that may connect each memory block with each computation block, where a path can carry up to four, 32-bit words per cycle.

The peak computation rates achievable by this DSP at 250 MHz are summarized in Table 1.

	Ops/cycle	Rate at 250 MHz
IEEE floating-point arithmetic	6	1500 MFLOPS
floating-point MACs	2	1000 MFLOPS
16-bit arithmetic	24	6000 MOPS
16-bit MACs	8	4000 MOPS
16-bit complex MACs	2	4000 MOPS

Table 1: Peak computation rates at 250 MHz

A. Sequencer

The DSP’s sequencing mechanism is based on a static superscalar approach, where one to four instructions are executed per cycle in what we call an *instruction line*, with *instruction-level parallelism* (ILP) determined prior to run time by code generation tools or a programmer. An instruction line may contain from 1 to 4 instructions, and the processor has a throughput of one instruction line per cycle. The idea of exposing ILP to the compiler comes from the older *Very-Long Instruction Word* (VLIW) approach, but the sequencing in this DSP is a much more general mechanism that avoids notorious VLIW shortcomings.

The sequencer has three basic attributes:

1. All registers are fully interlocked,
2. All compute and memory access instructions have a regular 2 cycle delay pipeline, and
3. Compute block instructions have optional SIMD capability, where a single instruction is issued to two units in parallel.

Similarly to general-purpose RISC processors, the interlocking register files enable a programming model that is functionally-defined at instruction line boundaries. There are two important implications of this.

One is that in this interlocked environment, code scheduling around the processor pipeline is required for *performance* only, but not to guarantee program *correctness*. In addition, the programming model guarantees that all the instructions in the same instruction line commit at the same time, and hence program correctness is preserved in spite of possible code scheduling imperfections. Simple programming models like this one are particularly important in embedded processing, where a relatively large portion of the most time-critical code base is still developed directly by assembly programmers.

The second implication is that the core supports a fully interruptible and responsive system, also a fundamental requirement in embedded computing. This also includes precise software exceptions.

An additional aspect of the programming model related to code scheduling is that all computation block instructions, as well as memory load instructions, have a pipeline delay of exactly 2 cycles. That is, the results of an instruction that executes at cycle 0 are always available at cycle 2. All IALU address calculations have a single cycle pipeline delay, and hence there is no scheduling required for address calculations.

SIMD execution improves code density by issuing one compute block instruction to two execution units simultaneously. In the section below, we present an example where portions of an algorithm exhibit a type of symmetry that can be used to take advantage of SIMD execution (namely, the addition portions of a FIR filter). We also show that in other portions of the same algorithm, SIMD execution may be overly restrictive, and a direct (non-SIMD) execution mechanism is also required.

B. Short-Vector Memory

In order to sustain high core compute rates, the memory architecture is capable of handling transactions that carry several words of data per access, in what we call a short-vector access. A memory transaction can carry from 1 to 4 words of consecutive data, and there may be up to 2 simultaneous transactions per cycle. A memory access can move data from any one of the 3 internal memory blocks to/from any one of the 4 register files.

There are 3 distinct types of memory access: (a) direct, (b) split, and (c) broadcast. These accesses vary according to the way that a short vector is routed to the compute blocks or to the memory blocks. In Figure 2 we show an example of a direct, a split, and a broadcast quad-word load, where {a3,a2,a1,a0} represent 4 consecutive 32-bit words in little-endian order (word with smallest address on the right).

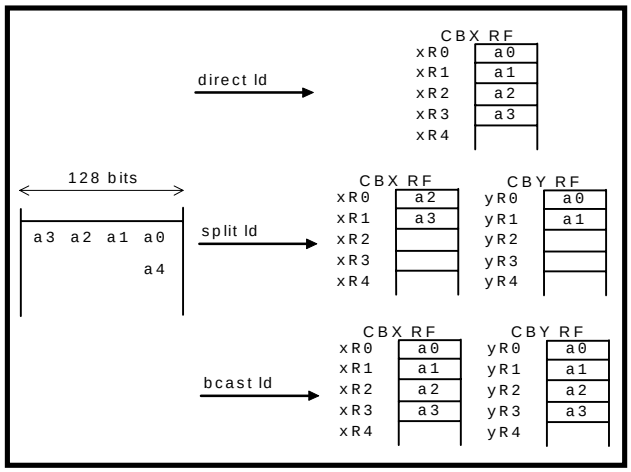


Figure 2: Three access types for quad-word loads.

C. Computation Blocks and Arithmetic Capability

In addition to single and extended precision floating-point support, the instruction set directly supports most 16- and 32-bit fixed point DSP, image, and video data formats including: fractional, integer, signed and unsigned. There is partial support for 8-bit data types. All the fixed point data formats have optional direct support for saturation arithmetic, and data type combinations are supported by means of instructions, rather than by hardware modes. For instance, there are 3 different instructions for performing fixed point addition: (a) add signed saturate, (b) add unsigned saturate, and (c) add and do not saturate. Specifying arithmetic data types by means of instructions rather by hardware modes is important in enabling a compiler to make effective use of DSP data types.

At 32-bit, the peak computation rate is 6 arithmetic operations, or 2 MACs per cycle. And in parallel, the memory architecture can sustain 2, 128-bit wide load/store operations, as well as 2 address calculations. The following instruction line is an example of this peak rate:

```
xR3:0=Q[j0+=4]; // load to CBX ptr. j0
yR3:0=Q[k0+=4]; // load to CBY ptr. k0
FR5=R4*R4;      // 2 SIMD multiplies
FR9:8=R6+/-R7;; // 2 SIMD adds, 2 subtracts
```

In this instruction line, the first two instructions are quad direct loads, the third is a SIMD multiplication (executed in both compute blocks), and the fourth is a SIMD add and subtract. (See Section III for an example and use of non-SIMD instructions.) Instructions are separated by semi-colons, and linstruction lines by double semi-colons. All 4 instructions in this line execute with throughput of one cycle. The "F" prefix indicates that these instructions operate on floating-point data.

At 16-bit, the peak computation rate is 24 arithmetic operations, or 8 MACs per cycle, i.e.,

```
xR3:0=Q[j0+=4];
yR3:0=Q[k0+=4];
sR7:6=R5:4*R5:4;     // 8 SIMD multiplies
sR11:10=R9:8+/-R9:8;;// 8 adds, 8 subtracts
```

The third instruction is an 8-way SIMD multiplication. In one compute block, say CBY, the register pair `YR5:4` holds 4, 16-bit input values, and the 4 results are stored in register pair `YR7:6`. And similarly for CBX, with register pairs `XR5:4` and `XR7:6`. In a sense, this is a two-level SIMD instruction: at the high level this instruction is issued to both multipliers, and at the low level it specifies that 4 packed multiplications be carried out in each multiplier. The "S" prefix indicates that these instructions operate on short-word (16-bit) data.

III. Benchmarks and Application Example

In Table 2 we show the kernel benchmarks for the FFT and the FIR filter, both for floating-point and 16-bit data types. These figures include all software overheads required to accomplish the tasks. In terms of peak MAC rates, the 32-bit FIRs achieve an efficiency of 90% (1.8 MACs/cycle, given a theoretical maximum of 2 MACs/cycle), and the 16-bit FIRs an efficiency of 88% (7.1 given 8 MACs/cycle). The complex FIR filters are programmed using native hardware support for complex 16-bit MACs.

	Execution time	Clock cycles
Floating-point:		
1k complex FFT, radix 2	41μs	10,300
50-tap FIR on 1k-samples	110μs	27,500
Single FIR MAC	2.2ns	0.55
16-bit fixed point:		
256-pnt. Complex FFT, rad 2	4.4μs	1,100
50-tap FIR on 1k-samples	29μs	7,200
Single FIR MAC	0.56ns	0.14
Single complex FIR MAC	2.28ns	0.57

Table 2: DSP benchmarks at 250 MHz

There are 3 distinct types of parallelism available in this DSP: (a) latency-2 computational pipeline, (b) multiple compute units, and (c) short-vector memory. These 3 forms of parallelism are complementary to each other, and all 3 must be used in order to achieve a high level of efficiency.

To take advantage of the computational pipeline, conventional loop unrolling and software pipelining techniques are applied [3]. Distributing computations to compute blocks may be accomplished by performing data-dependence analysis, a subject has been treated extensively in the general context of parallel computing [4], and also from the point of view of DSP

algorithms [5]. In the remainder of this section, we present an example that illustrates the use of these parallel techniques.

A. FIR Filters

FIR filters are closely related in structure to the vector product. Since vector-oriented processors are very effective in computing vector product operations, and linear algebra in general, on the surface it may seem that it is straight-forward to apply vector techniques to DSP algorithms like FIR filters.

However, data alignment requirements of FIR filters are different than those of linear algebraic algorithms, and in general this represents a barrier to realizing high efficiency implementations of FIR filters in vector-oriented processors. In order to achieve appropriate data to coefficients alignment, the interface of the compute blocks to the memory system has a data alignment buffer (DAB), that provides quad-word alignment at 32-bit and 16-bit boundaries.

The upper part of Figure 3 shows a diagram with the location of input data and coefficients in quad-word memory. In the lower part of this figure, we show the assembly code segment for the inner loop of the FIR. All instructions in a line are executed in one cycle and commit at the same time. In this code segment, the left-most column of instructions has data loads via the DAB (using pointer `j0`), and coefficient loads (using pointer `j1`, and with automatic circular buffering). Both loads are of type quad-word broadcast, so that the same data is replicated and shared in both compute blocks. This figure also shows that the result of quad DAB loads are the 4, 32-bit elements `{a[5] a[4] a[3] a[2]}`.

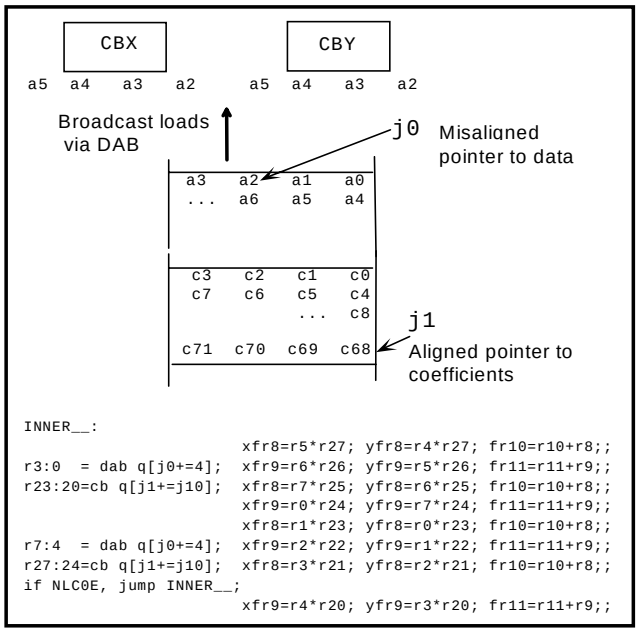
Similarly to the way that short-vector memory operations can result in poor performance if not aided by hardware alignment, the full SIMD execution mechanism can in some instances be overly-restrictive, and may be another cause of performance loss. Some DSP algorithms exhibit a type of symmetry that allows the use of SIMD execution, but very often there are symmetries (either in complete algorithms or portions of them) that cannot be efficiently parallelized with SIMD.

In this FIR filter example, the add instructions (in the right-most column) are issued as SIMD instructions. Each one of the add instructions specifies a total of 2 floating-point adds, one in each computation block (this is denoted by the absence of "`x`" or "`y`" instruction prefix).

However, the second and third columns of multiplications are *not issued in SIMD mode*, each instruction individually controls a multiplication in each computation block. The second column controls CBX, and the third controls CBY. This type of non-SIMD execution is used in this example to achieve single-element misalignment between the computation blocks, which is required to compute two output samples per inner loop iteration.

We use a simple partitioning where the *Data Dependence Graph* [5] of the FIR is split so that all the MACs of output `b[n]` are mapped to CBY, and all MACs of `b[n+1]` are mapped to CBX, where `b[n]` is the output signal, and `n` the time index.

The efficiency of this example for filters in the range of 50 taps is 90% relative to the peak MAC rate of 2 MACs/cycle. This figure accounts for all software overheads such as loop prolog and epilog, initialization code, as well as branch misprediction losses.




```
INNER__:
                          xfr8=r5*r27; yfr8=r4*r27; fr10=r10+r8;;
r3:0  = dab q[j0+=4];     xfr9=r6*r26; yfr9=r5*r26; fr11=r11+r9;;
r23:20=cb q[j1+=j10];     xfr8=r7*r25; yfr8=r6*r25; fr10=r10+r8;;
                          xfr9=r0*r24; yfr9=r7*r24; fr11=r11+r9;;
                          xfr8=r1*r23; yfr8=r0*r23; fr10=r10+r8;;
r7:4  = dab q[j0+=4];     xfr9=r2*r22; yfr9=r1*r22; fr11=r11+r9;;
r27:24=cb q[j1+=j10];     xfr8=r3*r21; yfr8=r2*r21; fr10=r10+r8;;
if NLC0E, jump INNER__;
                          xfr9=r4*r20; yfr9=r3*r20; fr11=r11+r9;;
```

Figure 3: Code segment for FIR filter.

IV. SUMMARY

In this paper we have presented a new highly-parallel DSP architecture that incorporates for the first time in an embedded processor a number of mechanisms found in general-purpose computing. These mechanisms include branch prediction, a deep and fully-interlocked pipeline, SIMD instruction execution, and register-based load/store instruction set. In addition, this architecture uses a short-vector memory system to sustain high computational core rates.

V. REFERENCES

[1] *"Buyer's Guide to DSP Processors,"* Berkeley Design Technology, Inc., 3rd edition. 2107 Dwigth Way, Second Floor, Berkeley, CA, 94704, USA. http://www.bdti.com.

[2] M. Levy, *"EDN's 1998 DSP-Architecture Directory,"* EDN Magazine, April 23, 1998. http://www.ednmag.com.

[3] J. Hennessy, D. Patterson, *Computer Architecture, A Quantitative Appproach*, second edition, Morgan Kaufmann Publishers, Inc. 1996.

[4] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, The Benjamin/Cummings Publishing Company, 1994.

[5] D. Moldovan, *Parallel Processing: From Applications to Systems*, Morgan Kaufmann Publishers, 1993.