THE TIGERSHARC DSP ARCHITECTURE

THIS HIGHLY PARALLEL DSP ARCHITECTURE BASED ON A SHORT-VECTOR MEMORY SYSTEM INCORPORATES TECHNIQUES FOUND IN GENERAL-PURPOSE COMPUTING. IT PROMISES SUSTAINED PERFORMANCE CLOSE TO ITS PEAK COMPUTATIONAL RATES OF 900 MFLOPS (32-BIT FLOATING-POINT) OR 3.6 BOPS (16-BIT FIXED-POINT).

Jose Fridman
Zvi Greenfield
Analog Devices, Inc.

In the past two years, several multiple data path and pipelined digital signal processors have been introduced to the marketplace. This new generation of DSPs takes advantage of higher levels of integration than were available for their predecessors. It also incorporates multiple execution units on a single core as well as deep execution pipelines. For an introduction to recent trends in DSPs see Eyer and Bier,[1] and for comprehensive analysis on DSP chips see the DSP buyer's guide[2] and Levy.[3]

Here, we describe a new parallel DSP architecture called TigerSHARC.[4,5] We focus on the computational aspects of its core and on-chip memory architecture. To sustain the high computation rates of cores with multiple execution units, memory subsystems must scale proportionately. We based our solution to the high-bandwidth demands of this parallel DSP core on a memory architecture characterized by what we call short-vector processor techniques. These techniques are essentially small-width vector processor interfaces.

In addition to the architectural description, we also present an application example of a finite-length impulse response, or FIR, filter. We use this example to illustrate a technique used to map this class of algorithms to a parallel, vector-oriented processor. The FIR filter is a representative member of a large class of DSP algorithms, namely any structure with delay lines such as infinite-length impulse response, or IIR, structures, equalizers, and multirate filters, all of which share similar solutions. (Two-dimensional extensions of these algorithms, such as 2D filtering and convolution used in imaging, can also be solved using extensions to the techniques presented here.) To efficiently map this class of algorithms to this parallel DSP, we must address two related problems: the distribution of computation among several execution units, and the provision of adequate alignment between data and filter coefficients.

To map the delay line structure of the FIR, we apply an algorithmic transformation to the algorithm, and, as a result, expose its parallelism in a form suited to the target architecture. This algorithmic transformation produces a high efficiency implementation by relying only on aligned short-vector memory accesses. This example also shows that the conventional single-instruction, multiple-data (SIMD) dispatch mechanism, although very effective in simple linear algebra and matrix operations, may be overly restrictive when applied to this class of DSP algorithms. And, as a result, non-SIMD execution is required to achieve high efficiency.

0272-1732/00/$10.00 © 2000 IEEE

TigerSHARC

The first implementation of the TigerSHARC architecture is in a 0.25-micron, five-level metal process at 150-MHz core clock speed. It delivers 900 Mflops (10^9 floating-point operations per second) of single-precision floating-point performance, or 3.6 GOPS (10^9 operations per second) of 16-bit arithmetic performance. It sustains an internal data bandwidth of 7.2 Gbytes/sec.

This TigerSHARC implementation (the ADSP-TS001) has several mechanisms found in general-purpose computing. To the best of our knowledge, this is the first time that all these techniques have been combined in a real-time embedded processor. Some of the most significant aspects of this new DSP are

- a register-based load-store architecture with a static superscalar dispatch mechanism in which instruction-level parallelism, or ILP, is determined prior to runtime under compiler and programmer control;
- highly parallel, short-vector-oriented memory architecture;
- support for multiple data types, including 32-bit IEEE single-precision floating point and 16-bit fixed point, with partial support for 8-bit fixed point;
- parallel arithmetic instructions for two floating-point multiply-accumulate (MAC) operations or for eight 16-bit MACs per cycle, with a SIMD execution mechanism;
- eight-stage, fully interruptible pipeline with a regular two-cycle delay on all arithmetic and load/store operations, and a 128-entry, four-way set-associative branch target buffer, or BTB; and
- 128 architecturally visible, fully interlocked registers in four orthogonal register files.

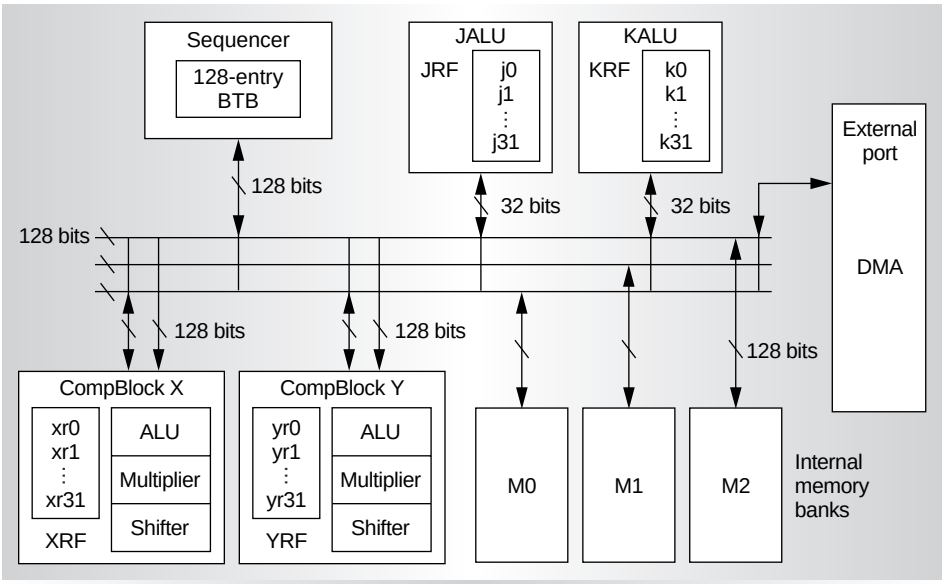

Figure 1. Top-level block diagram showing the major DSP subsystems and the data buses. Each data bus has associated 32-bit address and control buses. BTB: branch target buffer.

Architectural description

Figure 1 shows a block diagram with the major components of the architecture as well as the primary data buses. Each of the two computation blocks on the figure's lower left (CompBlock X and Y, or CBX and CBY) consists of a 32-entry general-purpose register file (XRF and YRF), ALU, multiplier, and shifter. The two computation blocks constitute the primary data path of the processor. Each computation block has two 128-bit ports that connect to the three internal 128-bit buses.

In the upper part of Figure 1 there are two integer units (JALU and KALU, collectively called the IALU). They function as generalized addressing units; each one includes a 32-entry general-purpose register file. Although used primarily for addressing, the IALU also supports general integer arithmetic and pointer manipulation. One of four masters (JALU, KALU, sequencer, or external port) produces addresses, and one of five slaves (CBX, CBY, M0, M1, or M2) consumes them. Each data bus has an associated address bus, which for clarity is not shown here. This figure also shows three internal SRAM banks (M0, M1, and M2), each with a 128-bit connection to the bus system.

The sequencer appears in the figure's upper left, along with a 128-entry, four-way set-associative branch target buffer. The sequencer, two IALUs, and the external port block are the four masters of the internal bus system and supply addresses and control to the memory banks.

The three internal buses provide a direct path for instructions into the sequencer. They

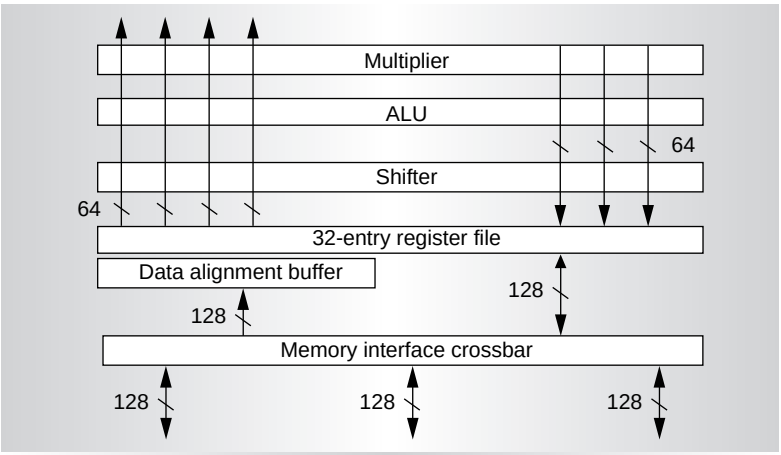

Figure 2. Internal block diagram for the computation blocks. The three internal buses of the system are at the bottom of this figure.

Table 1. Peak computation rates at 150 MHz.

Operation	Arithmetic operations per cycle	Rate at 150 MHz
IEEE floating-point arithmetic	2 multiplications, 2 additions, 2 subtractions	900 Mflops
Floating-point MACs	2 multiplications, 2 additions	600 Mflops
16-bit arithmetic	8 multiplications, 8 additions, 8 subtractions	3,600 MOPS
16-bit MACs	8 multiplications, 8 additions	2,400 MOPS
16-bit complex MACs	2 complex MACs	2,400 MOPS

also provide 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.

Figure 2 shows in greater detail the internal components and busing of the computation blocks. The register file supplies up to four operands to the multiplier, ALU, and shifter, and accepts three operands. On the side of the memory interface, the register file can either accept two operands from two load operations or accept one load and provide one store operand. Also shown in Figure 2 is the data alignment buffer, or DAB, which provides data steering for load operations.

Table 1 summarizes the peak computation rates achievable by this DSP at 150 MHz. "Mflops" represents 10^6 32- and 40-bit floating-point operations per second, and "MOPS" represents 10^6 16-bit operations per second. A 16-bit complex MAC has native support. It consists of four real multiplications, one real addition and real subtraction to form the complex number, and two real additions for the accumulation.

Sequencing and instruction flow

We based the sequencing mechanism on a static superscalar approach in which one to four instructions execute each cycle in an instruction line. Code generation tools or a programmer determines the instruction-level parallelism prior to runtime; the executable binaries contain this information. (In other processors, instruction lines are also referred to as instruction groups or bundles.) An instruction line may contain from one to four instructions, and the processor has a throughput of one instruction line per cycle. The idea of exposing instruction-level parallelism to the compiler comes from the very long instruction word (VLIW) approach. However, the sequencing in this DSP is a more general mechanism that avoids some of the shortcomings of conventional VLIW. Hennessy and Patterson[6] provides general overview material of VLIW. For the use of VLIW-related methods specific to DSP and media algorithms, see Faraboschi, Desoli, and Fisher.[7]

With the recent proliferation of processors that are statically scheduled by the compiler, VLIW has become so broad a term that it is virtually a synonym for any type of statically scheduled processor. However, most of these processors are in fact significantly different than raw VLIW. As a matter of nomenclature, we refer to the particular type of sequencing implemented in the TigerSHARC as static superscalar, because it is a statically scheduled multiple-issue processor. In addition, the sequencer supports the following three non-VLIW mechanisms: fully interlocked register files; all computation and memory access instructions with a regular two-cycle delay pipeline; and computation block instructions with optional SIMD capability in which a single instruction can be issued to two units in parallel.

Interlocking register files (similar to general-purpose superscalar processors) support a programming model that is functionally well defined at instruction line boundaries. That is, when the result of a computation or load is not available at the next instruction cycle, the processor stalls until that data becomes available. This contrasts with the VLIW model in

which the compiler directly controls all aspects of scheduling, including instruction timing (that is, scheduling an operation before all its source operands are available results in wrong data used as input).

The TigerSHARC architecture implements a model where the program specifies instruction-level parallelism only, but hardware dynamically resolves instruction timing. (There are also DSPs that are fully dynamically scheduled in which the processor determines instruction-level parallelism at runtime with some compiler assistance. See Infineon's TriCore[8] and ZSP Corp.'s ZSP16401.[9])

The following are the benefits of an interlocked superscalar-like programming model:

- The processor supports a fully interruptible system, which is required in embedded real-time computing, and significantly simplifies the implementation of the interrupt system. (This also includes precise software exceptions.) A processor with no interlocks may be unable to interrupt the pipeline and hold all data that is in flight, as more than one data value may be targeted to a register.
- The architecture supports code compatibility across different processor implementations without the need for recompilation, which conventional VLIW processors typically are unable to support.
- The architecture simplifies and enhances programmability. Code scheduling for the processor pipeline is required for performance only not to guarantee program correctness. Since the programming model guarantees that all the instructions in the same instruction line commit by the same time, program correctness is preserved despite code scheduling imperfections. Simple programming models like this are particularly important in embedded processing where assembly programmers develop large portions of time-critical code.
- There are no wasted instruction slots due to vertical and horizontal no-ops,[7] which in conjunction with SIMD result in high code density. Vertical no-ops come from exposed operation latencies, and horizontal no-ops come from unused instruction slots in the raw, horizontally microcoded VLIW model.

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 two cycles. (That is, the results of an instruction that executes at cycle i are always available at cycle $i + 2$.) All IALU address calculations have a single-cycle pipeline delay, requiring no scheduling for address and pointer calculations.

SIMD execution improves code density by simultaneously issuing one computation block instruction to two execution units. Next, we present an example in which portions of an algorithm exhibit a type of symmetry that can be used to take advantage of SIMD execution (namely, the adder chains of a vector product). We also show that in other similar algorithms, SIMD execution may be overly restrictive, and a direct (non-SIMD) execution mechanism is also required.

Types of parallelism

TigerSHARC has three distinct forms of data parallelism:

- parallel computation by means of two mechanisms: SIMD execution and subword parallelism,
- a short-vector memory architecture, and
- a two-cycle-deep computational pipeline.

SIMD and subword parallelism. SIMD refers to the method by which one instruction operates on more than one data item. This DSP implements SIMD dispatch by optionally issuing one computational instruction to both CBX and CBY computational units. (All computational instructions are encoded with two bits that determine whether the target computational box is CBX, CBY, or both.)

Subword parallelism is another distinct architectural technique. It increases parallelism at the data-element level by means of partitioning a processor's data path and performing more than one parallel computation on a single composite word. Subword parallelism is also sometimes referred to as multimedia extensions or packed operations.[7,10]

The literature often uses SIMD to denote subword parallelism. However, although subword parallelism is a specialized form of SIMD, in this DSP it is very important to distinguish

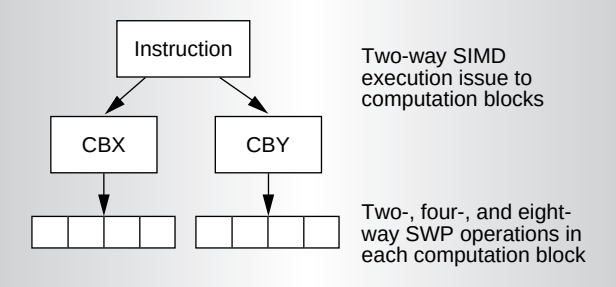

Figure 3. SIMD execution and subword parallel operations.

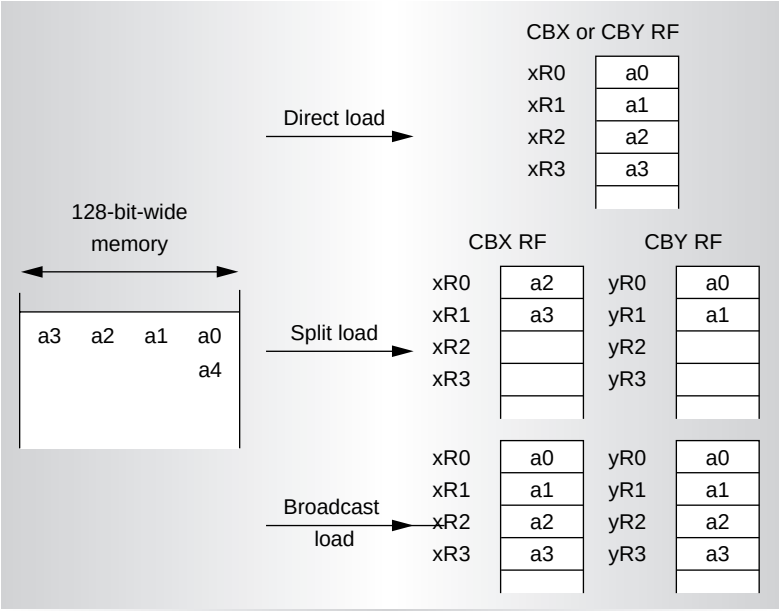

Figure 4. Three access types for quadword loads: direct, split, and broadcast.

between SIMD execution and subword parallelism operations. The TigerSHARC architecture makes use of both techniques, but in different and quite distinct situations. Generally, computation at the 32-bit level is organized with SIMD execution only (that is, one 32-bit operation per computation block). However, computation at the 16- and 8-bit level is organized with SIMD execution *and* subword parallelism (four packed, 16-bit operations per computation block). Figure 3 illustrates this conceptually, showing the two-way SIMD execution at the computation block level, and four-way subword parallelism at the subword level.

As an example of SIMD, consider the following instruction line that consists of four instructions. The first two instructions are quadword direct loads: IALU address registers j0 and k0 point to two memory locations and are post-incremented by 4. CBX registers xr0 to xr3 (denoted xr3:0) hold the four 32-bit data words loaded from the memory location pointed to by j0 (Q denotes quadword access) and similarly for the load to CBY. (See Figure 1 for the location of the register files.)

The third instruction is a floating-point SIMD multiplication: the contents of CBX registers xr4 and xr5 are multiplied and stored in register xr10; and in parallel, the contents of these registers in CBY are multiplied. In this case, the absence of an x or y prefix denotes execution in both computation units, and the f prefix denotes floating-point arithmetic. The fourth is a floating-point SIMD add and subtract, also executed in both computation blocks. The four instructions in the instruction line are separated by semicolons, and instruction lines are separated by double semicolons. All four instructions in this line execute with a throughput of one cycle.

```
xr3:0=Q[j0+=4];  // load quad-word to regs.
    //xr3:0 in CBX, update ptr. j0
yr3:0=Q[k0+=4];  // load quad-word to regs.
    //yr3:0 CBY, update pointer k0
fr10=r4*R5;      // 2 SIMD float multiplies in
    //both CBX and CBY
fr9:8=r6+/-r7;;  // 2 SIMD float adds, 2 float
    //subtracts in CBX and CBY
```

As an example of both SIMD and subword parallelism, consider the following instruction line. The third instruction is an eight-way multiplication. In one compute block (CBY), register pair yr5:4 holds four 16-bit input values, and the four results are stored in register pair yr7:6. The same occurs for CBX with register pairs xr5:4 and xr7:6. The fourth instruction performs a total of eight additions and eight subtractions. The s prefix indicates that these instructions operate on short-word (16-bit) data.

```
xr3:0=Q[j0+=4];
yr3:0=Q[k0+=4];
sr7:6=r5:4*r5:4;
   // 8 16-bit multiplies, 4 in each comp block
sr11:10=r9:8+/-r9:8;;
   // 8 16-bit adds, 8 16-bit subtracts
```

The examples presented in the remainder of this article illustrate SIMD processing on 32-bit data; for additional examples of 16-bit sub-

word parallelism programming, see Fridman.[11]

Short-vector memory. To sustain high core compute rates, the memory architecture handles transactions that carry several words of data per access (short-vector access). A memory transaction can carry from one to four words of consecutive data, with up to two simultaneous transactions per cycle. A memory access can move data from any one of the three internal memory blocks to and from any one of the four register files.

The architecture supports three types of memory access: direct, split, and broadcast. These accesses vary according to the way a short vector is routed to the computation blocks or to the memory blocks. Figure 4 gives an example of direct, split, and broadcast quadword loads, in which {a3,a2,a1,a0} represents four consecutive 32-bit words.

Arithmetic capability and instruction set

Table 2 summarizes the TigerSHARC instruction set, and the diagram in Figure 5 (next page) illustrates the peak ALU and multiplier subword parallelism operations in each computation block.

In addition to single- and extended-precision floating-point support, the instruction set directly supports all 16- and 32-bit fixed-point DSP, image, and video data formats. They include fractional, integer, signed, and unsigned data types, with additional partial support for 8-bit data types. All fixed-point data formats have optional support for saturation arithmetic. The instruction set, rather than hardware modes, directly supports all the combinations of data types. For instance, there are three distinct instructions for performing fixed-point addition: signed with saturation, unsigned with saturation, and without saturation. Specifying arithmetic data types by means of instructions rather than by hardware modes is

Table 2. Instruction set summary. B: byte; S: short; N: normal; L: long, F: float

Word width					ALU operations	Instruction
B	S	N	L	F	Instruction description	Syntax example
x	x	x	x	x	Add or subtract two operands	r0 = r1 + r2
x	x	x	x	x	Absolute value of the sum or difference of two operands	r0 = ABS(r1 – r2)
x	x	x	x	x	Sum or difference of two operands; divide result by 2	r0 = (r1 + r2)/2
x	x	x	x	x	Min or max	r0 = MIN (r1, r2)
x	x	x	x		Increment or decrement	r0 = INC r1
x	x	x	x		Add (or subtract) with carry	r0 = r1 + r2 + CI
x	x	x	x	x	Compare	r0 = COMP (r1, r2)
x	x	x	x	x	Clip	r0 = CLIP r1 BY r2
x	x	x	x	x	Absolute value	r0 = ABS r1
x	x	x	x	x	Negate	r0 = -r1
		x	x		Logical (AND, OR, XOR, NOT, AND-NOT)	r0 = r1 AND r2
x	x	x			Expand	r1:0 = EXPAND sr1
	x	x			Compact	sr1 = COMPACT r1:0
x	x				Merge	sr1:0 = MERGE r2, r3
x	x				Sideways sum	r0 = SUM sr1
		x	x		Count ones	r0 = ONES r1
x	x				Sum of absolute values of differences	pr0+=ABS (sr1:0 - sr3:2)
x	x	x	x	x	Dual add-subtract	r0=r1 + r2, r3=r1 – r2
				x	Reciprocal seed	fr0 = RECIPS r3
				x	Reciprocal square root seed	fr0 = RSQRTS r3
					Shifter operations	
x	x	x	x		Logical or arithmetic shift by operand or immediate value	r0 = ASHIFT r1 BY r2
		x	x		Rotate by operand or immediate value	r0 = ROT r1 BY r2
		x	x		Field deposit/extract	r0 += FDEP r1 BY r2
		x	x		Apply mask	r0 += MASK r1 BY r2
		x	x		Bit-manipulation instructions: set, clear, toggle, test	r0 = BSET r1 BY r2
					Multiplier operations	
	x	x		x	Multiply	r0 = r1 * r2
	x	x			Multiply-accumulate	mr1:0 += r1 * r2
	x				Complex MAC	mr1:0 += r1 ** r2
	x	x			Compact	r0 = COMPACT mr1:0
					Memory load and store	
128-bit move					Quadword load/store direct	xr3:0 = Q[IALU]
128-bit move					Quadword load/store broadcast	r3:0 = Q[IALU]
128-bit move					Quadword load/store split	r3:2 = Q[IALU]
32- or 64-bit move					Normal or long load/store (direct, broadcast, split)	r1 = [IALU]

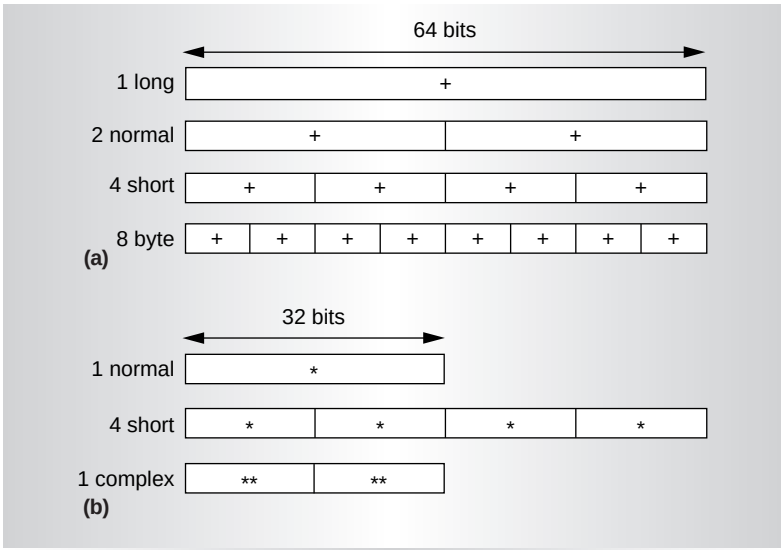

Figure 5. Peak ALU (a) and multiplier (b) subword parallelism operations per cycle, for each computation block.

important in enabling a compiler to make effective use of DSP data types.

Application examples

The next two examples illustrate methods that take advantage of the three types of parallelism present in this DSP architecture. The techniques used to program the two-cycle computational pipeline are conventional loop unrolling and software pipelining.[6] The techniques used to distribute computation to multiple computation units (CBX and CBY) and to a parallel memory system (using long-word and quadword accesses) are based on data-dependence analysis and high-level algorithmic transformations. The transformations expose parallelism in an algorithm in forms that are suitable for the target hardware. For a general introduction to data-dependence analysis, see Kumar et al.,[12] and in the context of DSP algorithms see Moldovan.[13]

Vector dot product

Figure 6 shows the memory organization used for the vector product of two sequences, A[n] and B[n], as well as the code sample of an inner loop that implements the dot product. In this memory diagram, IALU address register j0 points to data element A[0], and k0 points to B[0]. To sustain a peak rate of two MACs per cycle, we need a core-to-memory bandwidth of 128 bits/cycle (since two MACs consume four input data values). For this reason, both data vectors can reside in the same internal memory block (block M0 or M1 in Figure 1). When the bandwidth required by an application exceeds 128 bits/cycle, two memory banks can supply data to the core in the conventional DSP model of two data blocks.

The vector product of two length-L sequences is given by

$$P = \sum_{k=0}^{L-1} A(k)B(k)$$

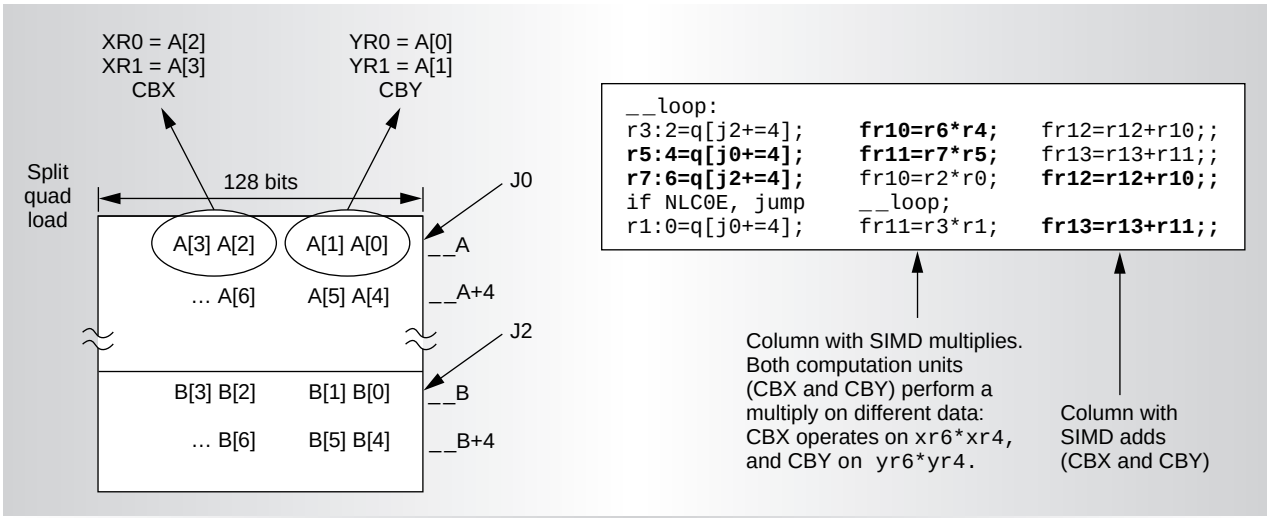

Figure 6. Memory organization and inner-loop code for the vector product of A[n] and B[n].

The mapping of this algorithm to the two computation blocks is given by

$$P^0 = \sum_{k=0}^{L/2-1} A(k)B(k)$$

and

$$P^1 = \sum_{k=L/2}^{L-1} A(k)B(k),$$

where P^0maps to CBX, and P^1 to CBY.

The loop shown in Figure 6 has been unrolled and pipelined; the computations of one iteration are shown in bold type. (At the end of the inner loop, an interblock transfer is required to add the two partial results P^0 and P^1.) Loop unrolling requires that four partial results be maintained in registers xr12, xr13 (for P^0), and yr12, yr13 (for P^1). The multiplications and additions in this inner loop are both issued with SIMD execution (denoted by the absence of x and y prefixes). This example shows that a simple algorithm with obvious symmetry can be directly implemented, making full use of SIMD execution to improve code density. This is generally not the case, as illustrated in the following example.

FIR filters

FIR filters are related in structure to the vector product, but the data alignment patterns required by the FIR are significantly different than those of linear algebra algorithms. Vector techniques like the one used in the previous section (quadword memory accesses) cannot be applied directly to the FIR. To map this algorithm, we need to achieve proper data-to-coefficient alignment, using either of two approaches: algorithmic restructuring or dedicated hardware support for misaligned computation (data alignment buffer block used for data alignment between register file and internal buses in Figure 2.) In this example we present a solution using an algorithmic transformation. For an example of using hardware for misaligned computation with subword parallelism, see Fridman.[11]

The FIR filter is given by

$$b(n) = \sum_{k=0}^{L-1} c(k)a(n-k),$$

where $0 \le n < M$. An obvious way to parallelize this algorithm would be to split it into two sequences:

$$b^0(n) = \sum_{k=0}^{L/2-1} c(k)a(n-k)$$

and

$$b^1(n) = \sum_{k=L/2}^{L-1} c(k)a(n-k)$$

Then, distribute the computations of sequence $b^0(n)$ into one computation block, and $b^1(n)$ into the other, similarly to the partitioning used for the vector product.

However, this partitioning can result in significantly suboptimal results for two reasons: 1) It does not solve the underlying alignment problem. 2) To terminate the computation of every output value, we must perform a final addition of $b^0(n) + b^1(n)$ outside the inner loop. This problem is severe for short filters, but affects medium to large filters as well. Note also that the bounds of the summation have been reduced by half. Any scheduling imperfection in the inner loop of this algorithm is magnified by a factor of two, since the number of inner-loop iterations has been reduced by half.

An alternate approach is to split computation as

$$b(n) = \sum_{k=0}^{L-1} c(k)a(n-k)$$

and

$$b(n+1) = \sum_{k=0}^{L-1} c(k)a(n-k+1)$$

Now, rather than create two sequences of partial results, we assign all the computations of output value $b(n)$ to a computation block, and computations of $b(n + 1)$ to the other, hence avoiding final partial result additions. This approach is an improvement over the former, but still does not fully resolve quadword alignment.

To provide full quadword alignment, we express the FIR as the four output equations

$$b\left(4n'+i\right) = \sum_{k=0}^{L-1} c\left(k\right)a\left(4n'-k+i\right), \quad \text{for } 0 \le i < 4, 0 \le n' < M/4$$

(Note that $N = 4n'$). Here, the data array is only accessed by a stride of four. For instance,

```
for(n=0; n<M/4–1; n+=4){
   for(k=0; k<L–1; k++){
      b0[n]   += c[k]*a[n–k];
      b0[n+1] += c[k]*a[n–k+1];
      b0[n+2] += c[k]*a[n–k+2];
      b0[n+3] += c[k]*a[n–k+3];
   }
}
```

Figure 7. C algorithm with quadword-aligned data. Note the stride of four for data a[n].

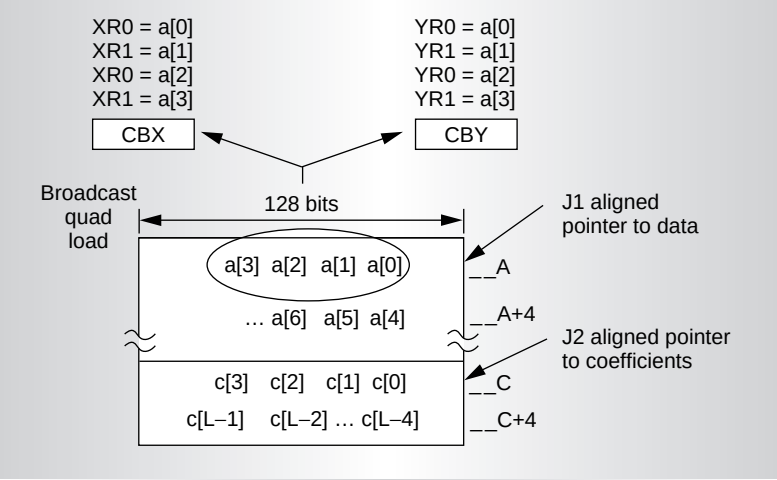

Figure 8. Memory organization for FIR filter.

$n' = k =$ and $0 \le i < 4$ define the quadword vector consisting of the four adjacent data values $a(0),a(1),a(2),a(3)$. These four data values can be combined with the single coefficient, $c(0)$. Subsequently, the four outputs, $b(i),0 \le i < 4$, are computed this way. (The assembly code in Figure 7 illustrates how four data values are combined with a single coefficient.) The computation of any four output sets $b(4n' + i),0 \le i < 4$ can be carried out with only four-element vector accesses. Hence, this partitioning distributes computations to CBX and CBY, and provides proper data-to-coefficient alignment. These equations are expressed in algorithmic form in Figure 7.

An additional benefit of this partitioning is that it reduces the data traffic between the memory system and the register files, as input data elements are reused for more than output computation. This results in a reduction of overall power consumption, showing the important effect of software on system power.

The downside of this technique is that it requires sufficient registers to hold all intermediate results and input data, and can only operate on sequences of a length that is a multiple of four. In addition, it can result in longer inner loops as compared to other solutions that use hardware support for data alignment.[4]

Figure 8 shows the memory organization of data and coefficients. In this implementation, prior to the computation of four output samples, four data values are loaded into a buffer that holds the internal state of the FIR filter. This follows the standard convention that allows the invocation of an FIR routine by supplying a coefficient with an internal filter state.

Figure 9 shows the assembly code segment for this FIR routine. This code segment is organized as four columns, zero through three. Column zero in the inner loop consists of data loads (using data pointer j1 guarded with circular buffering) and coefficient loads (using pointer j2). Both of these memory accesses are the quad-word broadcast type illustrated in Figure 8. The registers shown in bold type indicate four adjacent data values that are loaded as one quadword (into registers r3:0). These four data values are consumed at different times, depending on which output computation is taking place. Inside the rectangle we show that these four data values are combined with a single coefficient that resides in registers xr11 and yr11. For reasons of space, we abbreviated the inner-loop epilog.

Columns one and two have multiplications in CBX and CBY, respectively. To provide appropriate data-to-coefficient alignment, this code segment issues multiplications with different source registers to each computation block in the same instruction line. (For instance, the first instruction line inside the rectangle is multiplying the contents of register xr2 times xr11 in CBX, but yr0 times yr11 in CBY.) We see that these multiplications need to override the SIMD mechanism. In general, full SIMD execution can often be overly restrictive and can result in performance loss. The vector

dot product mentioned earlier is an example of an algorithm that can take advantage of SIMD execution. However, the FIR example shows that DSP algorithms may not exhibit the precise type of symmetry required to fully use SIMD, and it is necessary to provide flexible execution mechanisms to realize high efficiency.

Column three in Figure 9 shows that the add instructions are issued in SIMD. Each one of the add instructions specifies two floating-point additions, one in each computation block. The four outputs are computed in registers xr18, xr19, yr18, and yr19. Note also that at the conclusion of the inner loop, these four outputs are immediately available and do not require final accumulations or inter-block transfers.

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.

```
LC1 = DATA_SIZE/4;;
j3  = output_pointer;;  /* pointer to output buffer */
j2  = __C;;             /* circular pointer to coefficeients */
j10 = -4;;              /* coeff pointer increment */
j0  = input_data_pointer;;
j1  = __A;;             /* pointer to data state */
__outer:
xr3:0 = q[j0+=4];;      /* move quad input data into state buffer */
cb q[j1+=0x4] = xr3:0;; /* state is circularly addressed (cb) */
LC0 = FILTER_SIZE/8;;
r11:8 = q[j2+=j10];;    /* coefficient quad load */
r3:0 = cb q[j1+=4];;    /* data quad load */
                    xfr20=r2*r11; yfr20=r0*r11;;
r7:4 = cb q[j1+=4];  xfr21=r3*r11; yfr21=r1*r11;;
                    xfr16=r3*r10; yfr16=r1*r10;;
                    xfr17=r4*r10; yfr17=r2*r10;;
                    xfr16=r4*r9;  yfr16=r2*r9;;  fr18=r20+r16;;
                    xfr17=r5*r9;  yfr17=r2*r9;;  fr19=r21+r17;;
__inner:
r15:12 = q[j2+=j10]; xfr16=r5*r8;  yfr16=r3*r8;   fr18=r18+r16;;
                    xfr17=r6*r8;  yfr17=r4*r8;;  fr19=r19+r17;;
                    xfr17=r5*r15; yfr16=r4*r15;; fr18=r18+r16;;
r3:0 = cb q[j1+=4];  xfr17=r7*r15; yfr17=r5*r15;  fr19=r19+r17;;
                    xfr16=r7*r14; yfr16=r5*r14;  fr18=r18+r16;;
                    xfr17=r0*r14; yfr17=r6*r14;  fr19=r19+r17;;
                    xfr16=r1*r13; yfr16=r6*r13;  fr18=r18+r16;;
                    xfr17=r1*r13; yfr17=r7*r13;  fr19=r19+r17;;
r11:8 = q[j2+=j10];  xfr16=r1*r12; yfr16=r7*r12;  fr18=r18+r16;;
                    xfr17=r2*r12; yfr17=r0*r12;  fr19=r19+r17;;
r7:4 = cb q[j1+=4];  xfr16=r2*r11; yfr16=r0*r11;  fr18=r18+r16;;
                    xfr17=r3*r11; yfr17=r1*r11;  fr19=r19+r17;;
                    xfr16=r3*r10; yfr16=r1*r10;  fr18=r18+r16;;
                    xfr17=r4*r10; yfr17=r2*r10;  fr19=r19+r17;;
                    xfr17=r4*r9;  yfr16=r2*r9;   fr18=r18+r16;;
 if NLC0E, jump__inner;
                    xfr17=r5*r9;  yfr17=r3*r9;   fr19=r19+r17;;
/* INNER LOOP EPILOG  (ABBREVIATED) */
                    xfr17=r2*r12; yfr17=r0*r12;  fr19=r19+r17;;
                                                 fr18=r18+r16;;
                                                 fr19=r19+r17;;
if NLC1E, jump__outer;
      q[j1+=4] = r19:18;; /* four output samples store */
```

Four data points (in r3:0) combine with one coefficient (in r11) to form four output partial results.

Figure 9. Code sample for a FIR filter. Shown in bold type are the four data values of a one quadword load.

Application benchmarks summary

Table 3 (next page) lists the benchmarks for the fast Fourier transform (FFT) radix-2 algorithm with full bit reversal and for the FIR filter, both floating-point and 16-bit data types. In peak MAC rates, the 32-bit FIRs achieve an average efficiency of 90% (1.8 MACs/cycle, given a theoretical maximum of 2 MACs/cycle), and the 16-bit FIR, an average efficiency of 88% (7.1 MACs/cycle, given a maximum of 8 MACs/cycle). The complex FIR filters are programmed using native hardware support for complex 16-bit MACs.

The TigerSHARC architecture solves a number of problems in very high performance DSP computation by applying parallelism at the data element level and at the instruction level. The first implementation of this architecture is intended to support applications that use floating-point data types, for example, in radar, imaging, and medical computer graphics. This first implementation will also support applications that use 16-bit, fixed-point data types, primarily in telecommunications infrastructure. MICRO

Table 3. DSP benchmarks at 150 MHz. "1k" represents 1,024 samples.

Operation	Execution time	Clock cycles
Floating point:		
1k complex FFT, radix 2	68.0 µs	10,300
50-tap FIR on 1k samples	183 µs	27,500
Single FIR MAC	3.6 ns	0.55
16-bit fixed point:		
256-point complex FFT, radix	7.3 µs	1,100
50-tap FIR on 1k samples	48.0 µs	7,200
Single FIR MAC	0.92 ns	0.14
Single complex FIR MAC	3.76 ns	0.57

Acknowledgments

The material presented in this article represents the work of a very large group of people at Analog Devices, including the software tools, product engineering, Israel design teams, and in particular Doug Garde.

References

1. J. Eyre and J. Bier, "DSP Processors Hit the Mainstream," *Computer,* Vol. 31, No. 8, Aug. 1998.
2. *Buyer's Guide to DSP Processors,* 3rd ed., Berkeley Design Technology, 2107 Dwight Way, Second Floor, Berkeley, CA, 94704; http://www.bdti.com.
3. M. Levy, "EDN's 1998 DSP-Architecture Directory," *EDN,* Apr. 23, 1998; http://www.ednmag.com.
4. O. Wolf and J. Bier, "Sharc Attack: Analog Devices Discloses New High-End DSP," *Microprocessor Report,* Dec. 7, 1998, MicroDesign Resources, Sebastopol, Calif., pp. 12-15.
5. J. Fridman and W. Anderson, "A New Parallel DSP With Short-Vector Memory Architecture," *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing,* IEEE Press, Piscataway, N.J., 1999.
6. J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach,* 2nd ed., Morgan Kaufmann Publishers, San Francisco, Calif., 1996.
7. P. Faraboschi, G. Desoli, and J. Fisher, "The Latest Word in Digital and Media Processing," *IEEE Signal Processing,* Vol. 15, No. 2, Mar. 1998.
8. J. Eyre and J. Bier, "Infineon's TriCore Tackles DSP," *Microprocessor Report,* Apr. 19, 1999, MicroDesign Resources, pp. 12-14.
9. ZSP16401, ZSP Corporation; http://www.zsp.com/Products/Notes/pn-16401-06.html.
10. A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for Multimedia PCs," *Communications of the ACM,* Vol. 40, No. 1, Jan. 1997, pp. 25-38.
11. J. Fridman, *"Data Alignment for Sub-Word Parallelism in DSP," Proc. IEEE Workshop on Signal Processing Systems*, *SiPS 99,* IEEE Press, 1999, pp. 251-260.
12. V. Kumar et al., *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms,* Addison-Wesley Publishing Co., Melbourne, Australia, 1994.
13. D. Moldovan, *Parallel Processing: From Applications to Systems*, Morgan Kaufmann Publishers, 1993.

Jose Fridman is a DSP architect at Analog Devices, Norwood, Massachusetts, where he is involved in developing next-generation DSP architectures. His current interests are in the area of DSP processors and systems. Fridman received the MS and BS degrees in electrical engineering from Boston University and a PhD degree in electrical and computer engineering from Northeastern University. He is a member of the IEEE and the IEEE Design and Implementation of Signal Processing Systems (DISPS) and IEEE Industry DSP (IDSP) committees.

Zvi Greenfield is principal manager of architecture and design verification for Analog Devices TigerSHARC DSPs. He has also worked with National Semiconductors, Israel Design Center, where he was involved in the development of the NS32000 microprocessors and in the architecture and design of digital signal processors for modem, fax, and digital answering machine applications. Greenfield holds a BS degree in electronic engineering with a specialization in computer hardware, software, and communications from Technion, Israel Institute for Technology, Haifa.

Direct comments concerning this article to Jose Fridman, Analog Devices, One Technology Way, Norwood, MA, 02062; jose.fridman@analog.com.