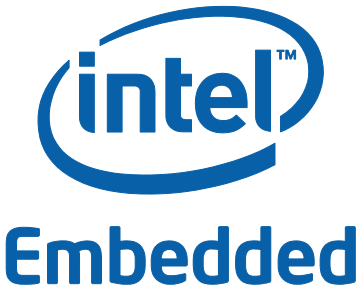


White Paper
Fritz Gerneth
Senior Technical Marketing Engineer
Intel Corporation

FIR Filter Algorithm Implementation Using Intel® SSE Instructions

Optimizing for Intel® Atom™ Architecture

March 2010

323411

Executive Summary

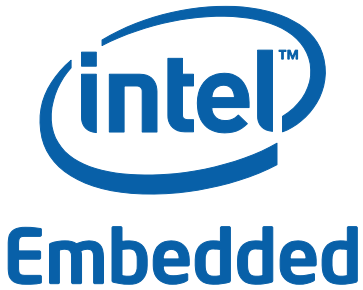
Intel® software products help accelerate development of efficient program code, which takes advantage of the performance potential of the Intel® Atom™ processor. In some scenarios, advanced compiler technologies or optimized off-the-shelf libraries are insufficient to meet extreme performance requirements such as signal processing tasks. In those cases, the extra effort of creating hand-optimized routines may be justified and necessary to maximize performance.

In some cases, the extra effort of creating hand-optimized routines is justified to maximize the performance of the Intel® Atom™ processor. This white paper describes a real life project where FIR filters were optimized for Intel® Atom™ processors to maximize performance.

This white paper describes a real life project, where simple but common types of digital filters, FIR filters, were optimized for Intel® Atom™ processors. Key architectural features of the Intel® Atom™ processors are outlined, such as SIMD, memory alignment and in-order execution. Also included is an explanation of optimization Best Known Methods (BKMs), which were applied to achieve a FIR filter throughput which is within a few percent of the theoretical limit of the Intel® Atom™ processor.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc. §

Contents



Finite Impulse Response Filters

Finite Impulse Response (FIR), filters are one of the primary types of filters used in Digital Signal Processing. The filter output of an mth-order filter can be expressed as a weighted sum of the current and m previous values of the input:

$$y[j] = \sum_{i=0}^{m} c_i x[j-i]$$

A small snippet of C-code to calculate 640 filter output values for a 63rd-order filter can be written as:

```
for ( j = 0; j < 640; j++) {
     int s = 0;                 // s = accumulator
     for ( i =0; i <= 63; i++)
     s += c[i] * x[i+j];        // x[] = input values
                                // c[] = filter coefficients
     y[j] = s;                  // y[] = output values
}
```

Note: The array c[] of filter coefficients is mirrored in memory for the rest of this document.

Preservation of the filter state across multiple invocations has to be handled by the caller (as for the rest of this document); this can be achieved by prefixing the input values x[] with m previous input values.

On computing platforms based on Intel® architecture, the usage of Intel® Streaming SIMD Extensions (Intel® SSE) instructions is essential for efficient implementations of FIR filters and other signal processing tasks.

The Intel® Atom™ processor supports all instructions up to the instruction set Intel® SSSE3. See the *Intel® 64 and IA-32 Architecture Software Developer's Manual* for details. Intel® SSE instructions operate on 128-bit data, which can be partitioned according to the desired data type; for 16-bit integer data, which is mathematically equivalent to a 16-bit fix point, one instruction can trigger eight operations. The Intel® Atom™ microarchitecture allows for up to two SIMD-integer-addition instructions per processor clock. A 1 GHz Intel® Atom™ processor can calculate 16 billion 16-bit-additions per second. Multiplications take more time, one SIMD integer multiply instruction every other processor clock, and the timing for floating point calculations is different as well.

The core of an Intel® SSE-based 16-bit FIR filter is the Intel® SSE instruction **pmaddwd**, executing eight 16x16-bit multiplications in parallel. The eight intermediate 32-bit results are pair wise summed to make them fit in a 128-bit register as shown in Figure 1.

Figure 1. Instruction pmaddwd Executes Eight 16-Bit Multiplications and Adds the Results Pair Wise

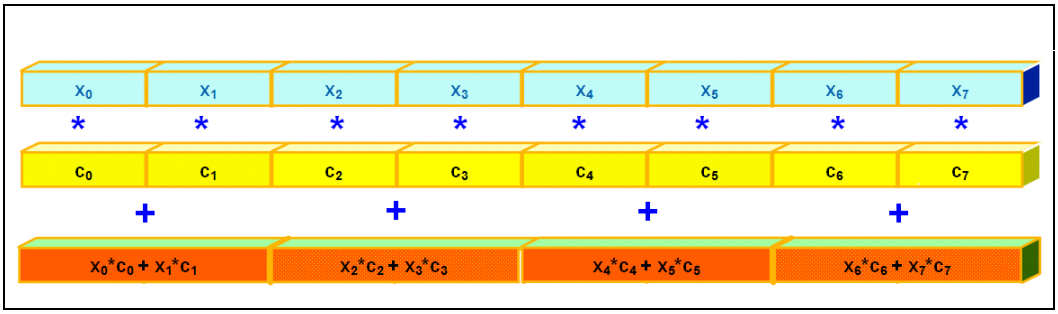


Figure 2 lists assembly code for the C-code shown previously. The multiply-operations do not depend on each other, and the additions can be made in any order. Therefore, the filter output can be processed in parallel. The inner loop generates one filter output value per iteration, four partial values, requiring a final accumulation to one single value, which is then handled in the outer loop. The outer loop runs through the input array one by one.

The following boundary conditions need to be preserved: the filter order has to be a multiple of eight, minus one (8*n-1), and the array of filter coefficients has to be 16-byte memory aligned, due to the 16-byte memory alignment requirement for most Intel® SSE instructions.

Figure 2. FIR Filter Implementation Using Intel® SSE Instructions

```
outerloop:
       pxor xmm0, xmm0      // initialize 4 accumulators (32 Bit each) by 0
       sub ebp, ebp         // initialize Index in filter taps mit 0

innerloop:
       movups xmm1, qword ptr [esi+ebp]  // load 8 vector elements in 128 Bit
register
       pmaddwd xmm1, qword ptr [edi+ebp] // .. pair wise multiply with TAPs
       paddd xmm0, xmm1     // accumulate results in xmm0 (4 x 32 Bit)
       add ebp, 2*8
       cmp ebp, 2*64        // repeat for all 64 Taps
       jnz innerloop

       phaddd xmm0, xmm0
       phaddd xmm0, xmm0    // accumulate four partial values to one result
       psrad xmm0, 15       // scale to 16 Bit
       movd qword ptr [eax], xmm0 // store result (only 16 Bit are relevant)
       add esi, 2
       add eax, 2              // increment pointer for input & output
       sub ecx, 1              // repeat for all input samples
```

Parallel computations makes this naive Intel® SSE-based implementation already significantly faster than the reference implementation in C code; processing 640 values takes ~70480 CPU clocks. However the Intel® Atom™ architecture allows for 8 multiply-accumulate operations every other clock, and so could handle 640*64=40960 multiplications in 10240 clocks, almost 7x the throughput measured so far! How can this discrepancy be explained, and, most important, how can the filter performance be improved?

A closer look at the assembly code, considering architectural characteristics of the Intel® Atom™ CPU, possibly assisted by the VTune™ Performance Analyzer, reveals the following possible root causes for performance losses:

1. **Multiply latency:** While it is possible to execute a SIMD multiplication every other processor clock (throughput), it takes five processor clocks until the result becomes available and can be further processed by other instructions, known as latency. If the result is used too early, the Intel® Atom™ CPU will stall due to an in-order execution, which is in contrast to out-of-order execution for the Intel® Core™2 processor family and Intel® Core™ i7 processors; for those the CPU rearranges the instruction order at runtime to avoid stalls.
2. **Unaligned load:** The set of 8 input values processed in the inner loop is not guaranteed to be 16-byte memory-aligned. Alignment is hardly achievable, since the input pointer is advanced by one value, two bytes, in the outer loop. For this reason, the load instruction **movups** must be used. **Movups** handles unaligned memory references, but implies a processor stall of several clocks.
3. **Scalar Code:** The payload in the outer loop (an accumulation of partial results, scaled to 16 bits, storing the result) uses SIMD instructions, but essentially doesn't take advantage of the SIMD parallelism, thereby compromising the high throughput achieved in the inner loop.
4. **Horizontal Operations:** the "horizontal" accumulation of four 32-bit partial result values is performed using the **phaddd** instruction, a very convenient but relatively expensive instruction that requires several processor clocks.

Hiding the Multiply Latency

The key to an improved version of the filter routine is a technique known as **Loop Unrolling**: the inner loop, executed eight times at filter length 63, is completely unrolled. Per instance, a different 128-bit CPU register is used, e.g., xmm2 instead of xmm1, then xmm3, etc. Loop unrolling eliminates the small overhead of the inner loop. More important, the resulting instruction sequence can be rearranged to put a sufficient number of other instructions between a multiplication and the subsequent use of the result. Figure 3 shows the inner loop of the filter routine completely unrolled. The eight instances are indented differently to illustrate the relationship to the original code from Figure 2. It is easy to see how the results from a **pmaddwd** instruction are only used five or more instructions later, to avoid processor stalls.

Figure 3. The Inner Loop of the FIR-Filters, Eight Times Unrolled and Instructions Rearranged to Avoid Stalls After pmaddwd

```
movups xmm0, qword ptr esi[0]
pmaddwd xmm0, qword ptr edi[0]
     movups xmm1, qword ptr esi[16]
     pmaddwd xmm1, qword ptr edi[16]
        movups xmm2, qword ptr esi[32]
        pmaddwd xmm2, qword ptr edi[32]
            movups xmm3, qword ptr esi[48]
            pmaddwd xmm3, qword ptr edi[48]
     paddd xmm0, xmm1
                movups xmm4, qword ptr esi[64]
                pmaddwd xmm4, qword ptr edi[64]
        paddd xmm0, xmm2
                    movups xmm5, qword ptr esi[80]
                    pmaddwd xmm5, qword ptr edi[80]
            paddd xmm0, xmm3
                        movups xmm6, qword ptr esi[96]
                        pmaddwd xmm6, qword ptr edi[96]
                paddd xmm0, xmm4
                            movups xmm7, qword ptr esi[112]
                            pmaddwd xmm7, qword ptr edi[112]
                    paddd xmm0, xmm5
                        paddd xmm0, xmm6
```

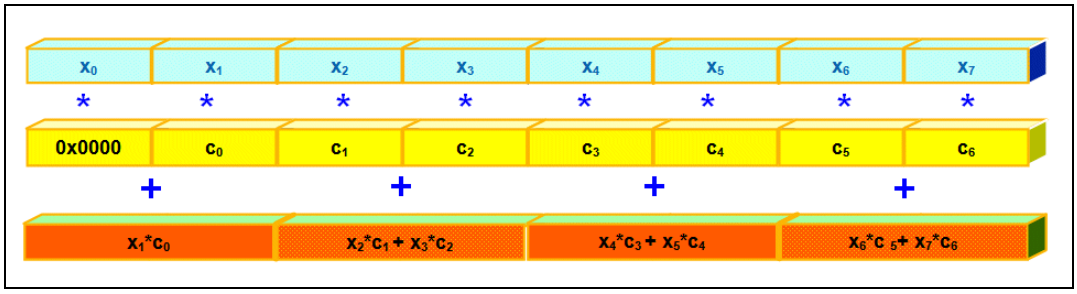
The execution time for filtering an input data set of 640 values is significantly reduced and now is ~45000 processor clocks.

Unaligned Load

The pointer [esi], used to address the filter input in chunks of 128-bit, or eight 16-bit values at a time, is incremented 2 bytes per iteration of the outer loop. Under the assumption that the input array is 16-byte aligned, the input array will be nicely aligned for the first iteration, but will be offset by 2 bytes for the second iteration, then offset by 4 bytes, etc. The 9th iteration will again exhibit 16-byte alignment. Seven out of eight iterations are unaligned.

To achieve 16-byte alignment for all cases, it is necessary to unroll the outer loop eight times, resulting in eight instances. For the first instance, all load operations are aligned, so the fast aligned-load instruction **movaps** can be used right away. For the second instance, the following trick helps: instead of an unaligned load from [esi+2] an aligned load from [esi] is performed, yielding seven usable values. Using **pmaddwd**, these are multiplied with a copy of the filter coefficients where all coefficients are also shifted by one slot (Figure 4).

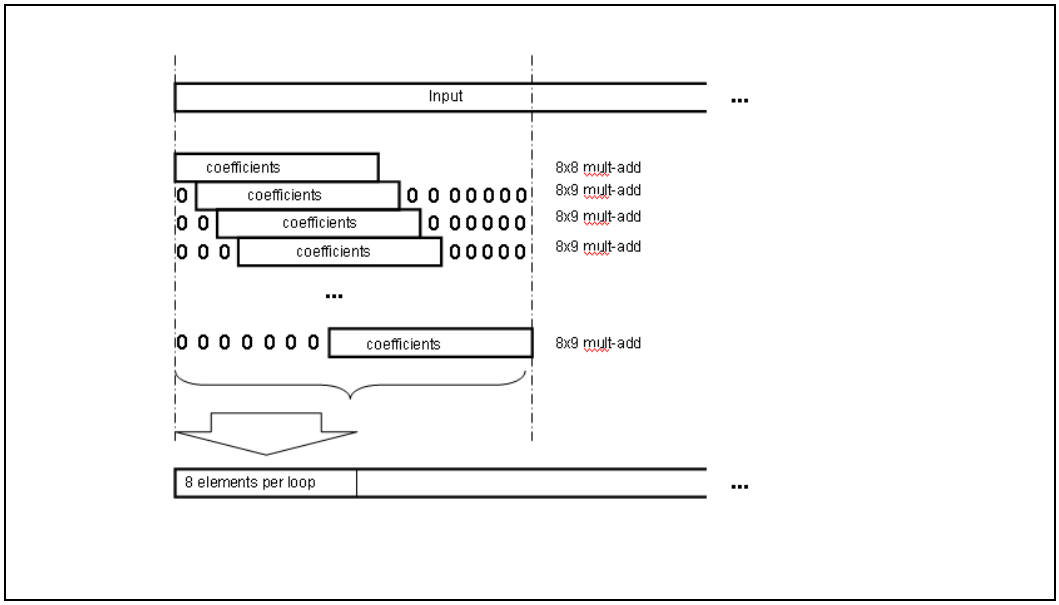
Figure 4. The Instruction pmaddwd at Work For the Case Where Input Values Starting at x1 Are Multiplied With Filter Coefficients Starting at c0



This results in the first partial values for the second filter output. To prevent corruption of the result by the bogus value in x0, the copy of the coefficients array is zero-padded at slot 0. The remaining 57 coefficients are processed in the same manner. It is important to apply zero-padding for the last 64th coefficient to avoid corruption of the result. For the third instance, again input values are read starting from [esi], and processed with a copy of the coefficients where all entries are moved by two slots (4 bytes), etc.

A total of seven copies with different memory alignment is required in addition to the coefficients' original (Figure 5). Creation of these will happen only once and therefore will not have a big performance impact. The required padding increases the length of the coefficient copies from 64 to 71, which also results in an 11% increase of computational effort. On the other hand, forcing all memory references to 16-byte alignment allows usage of the faster **movaps** instruction. In summary, the processing time decreases dramatically to ~17000 processor clocks despite the incremental processing effort.

Figure 5. Unrolled Loop Instances: First Instance Uses Original Coefficients; Others Use Proper Memory Alignment and Zero-padding



Eliminating Scalar Code

Unrolling the outer loop also opens up the opportunity to eliminate the scalar code for result accumulation and scaling. In the unrolled code, each loop iteration calculates 32 partial 32-bit results, which need to be summed up and scaled to eight 16-bit results. This task can efficiently be done in SIMD fashion, as illustrated in Figure 6; condensing 8x4 32-bit values to eight 16-bit values takes 6 horizontal adds, 2x128 Bit shifts, 1x128-bit pack (instead of 16 horizontal adds, 8x128-bit shifts, 8x128 Bit pack). See the *Intel® 64 and IA-32 Architecture Software Developer's Manual* for a detailed explanation of the instructions used.

Figure 6. Optimized Summing and Scaling of the Eight Results Calculated per Iteration of the Unrolled Loop

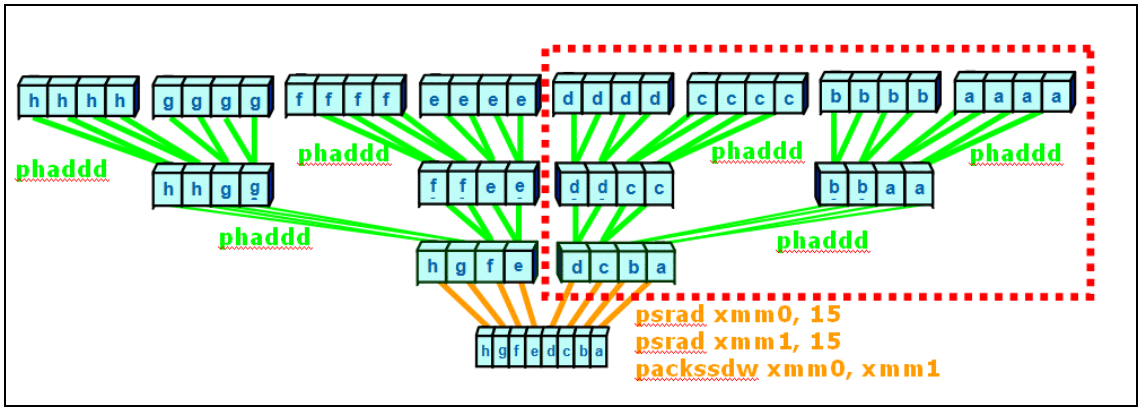
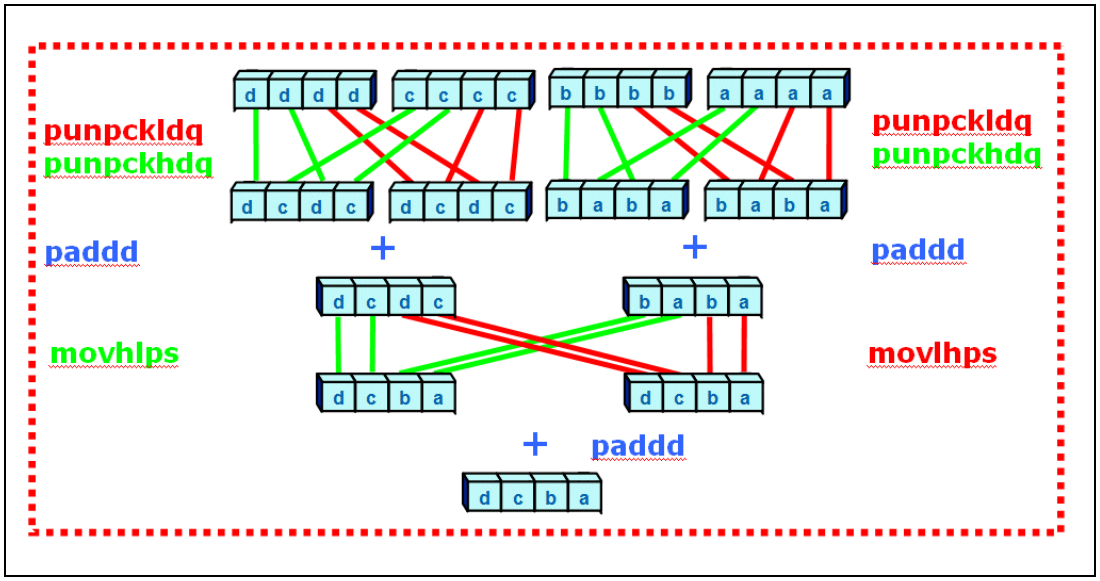


Figure 7. Optimized Implementation for Horizontal Sum Up of Partial Results



After this final optimization step, the computation time decreases to ~13680 CPU clocks. The remaining gap to the architectural limit of 10240 clocks is mostly due to the additional computational work added by introducing zero padded coefficients, plus some extra effort for accumulation, scaling and storing of the result values.

Summary

This paper describes the optimization of a 16-bit fix point FIR filter of order 63. In several steps, the filter performance was improved by a factor of more than 5 and was brought close to the theoretical limit of the current architecture of Intel® Atom™ processors. This became possible through loop unrolling, forceful use of Intel® SSE instructions, consideration of memory alignment and smart choice of the most efficient rather than the most obvious SSE instructions.

The BKMs described here can be applied to other FIR filters with little or no changes. The filter order and number of output values can be changed easily, though the number of output values must be a multiple of eight; the benefit from Intel® SSE instructions will increase with the increasing number of output values and higher order filters. Floating point-based FIR filters can be optimized following the same recipes; here the Intel® SSE instruction set allows for four-way parallelism.

Multirate filters are commonly implemented using FIR filters. Using interpolation and decimation, the output is resampled to a different data rate; for example, 640 input values may result in 480 output values. The optimization steps described in this white paper allow for optimal performance of multirate filters on Intel® architecture-based processors, too.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and

connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

References

[1] *Intel® 64 and IA-32 Architecture Software Developer's Manual*

Volume 2, http://www.intel.com/products/processor/manuals/

[2] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*

http://www.intel.com/products/processor/manuals/

Authors

Fritz Gerneth is a Senior Technical Marketing Engineer with the Software and Services Group at Intel Corporation.

Acronyms

BKM	Best Known Method
FIR	Finite Impulse Response (filter)
Intel® SSE	Intel® Streaming SIMD Extensions