TeleBench™1.1

software benchmark data book

www.eembc.org

Table of Contents

An Industry-Standard Benchmark Consortium

TeleBench™ Version 1.1

Benchmark Name: Autocorrelation

Highlights

- **Calculation of a finite length fixed-point autocorrelation function.**
- **16 bit, fixed point (integer) arithmetic.**
- **Accumulation overflow protection (no pre-scaling).**
- **Multiple (3) data sets: sine, pulse, and speech.**

Application

Autocorrelation is one of the basic analysis tools in signal processing. It represents the second order statistics of a random process and is widely used for analysis and design in many telecommunications applications. The autocorrelation function R[k] is defined as the expected value of x[n]*x[n+k], where x[n] is random process (R[k] = E{x[n]*x[n+k]}, E – the expectation operator). In practical applications, the expected value operation is replaced by a sum operation, R[k] = 1/N*$\square_n$x[n]*x[n+k], over N samples as an estimation of R.

Practically, the autocorrelation coefficient at lag k R[k], represents the amount of correlation between two samples of the sequence x spaced by k samples apart. The amount of correlation can be translated into redundancy in compression applications, or system response in modeling and system identification.

Autocorrelation functions are widely used in many telecommunication applications such as speech compression, speech recognition, channel estimations, sequence estimation (maximum likelihood), system identification and for the solution of the well-known Yule-Walker equations.

Benchmark Description

This benchmark performs a fixed-point autocorrelation function calculation of a finite length input sequence according to the following formula:

AutoCorrData [k] = 1/N*$\square_n$ InputData [n]* InputData [n+k]. k=0,1,…K-1

InputData is the input sequence given in a 16-bit signed integer representation (“short”).
The benchmark implements a 32-bit wide accumulation along with an overflow protection (via scaling) and returns the K (NumberOfLags) length AutoCorrData sequence in 16-bit signed integer format (“short”).
The datasets for this benchmark comprises three signal shapes. These shapes are a sine wave of frequency Fs/32 and 1024 samples length, a 16 samples symmetric pulse function, and a segment of 500 samples voiced speech signal. The shape may not affect the timing and the accuracy of the output.

Analysis of Computing Resources	The arithmetic operations used in this benchmark are multiply, shift and add. The algorithm is implemented in two nested loops where the actual arithmetic is executed in the inner loop. The benchmark explores the target CPU's ability to efficiently perform multiply, parameterized shift, and add operations in a nested loop structure.
Special Notes	All of the data files must be run to obtain an EEMBC Telemark™ score.

An Industry-Standard Benchmark Consortium

TeleBench™ Version 1.1

Benchmark Name: Bit Allocation

Highlights

- **Benchmarks part of a DSL modem that uses discrete multi-tone (DMT) technology**
- **16 bit fixed point (integer) code**
- **Multiple (3) data sets**
- **Largely Integer Math with memory accesses, fits inside small L1 caches but based on a real algorithm in a real application**

Application

This benchmark performs a Bit Allocation algorithm for digital subscriber loop (DSL) modems that use discrete multi-tone (DMT) technology. The benchmark provides an indication of the potential performance of a microprocessor in a DMT based DSL modem system.

DMT modulation partitions a channel into a large number of independent subchannels (carriers), each characterized by a signal to noise ratio (SNR). A bit allocation algorithm is thus required to allocate a number of bits to these carriers according to the measured SNR of each carrier.

Benchmark Description

The benchmark initializes the number of carriers, which come from different data sets (256, 100 and 20). The SNR profile in dB for the carriers is contained in a 16-bit input array (CarrierSNRdB). The range of Carriers' SNR in dB, [-64.0, 63.998] (float), is represented by the range [-32768, 32767] in fixed-point format. The total number of bits (BitsPerDMTSymbol) is allocated to the carriers by using a "water level" algorithm: each carrier is compared with a "water level". Carriers whose SNR is below the water level have no bits allocated to them. Carriers with an SNR above the water level have bits allocated to them in proportion to the difference between the water level and that carrier's SNR. The maximum number of bits which can be allocated to a carrier is defined as MAX_BITS_PER_CARRIER. If the difference between a carrier's SNR and the current water level is larger than or equal to 32767, MAX_BITS_PER_CARRIER bits will be allocated to the carrier. Upon the start of the benchmark, the maximum SNR of the carriers is saved as the initial water level.

The exact number of bits allocated to a carrier for a given delta from the water level is given by the allocation map array. This array is a pre-computed look-up table whose values range from 0 to MAX_BITS_PER_CARRIER.

The total bit allocation is clamped to the "Bits Per DMT Symbol", BPDS. This means that some carriers near the end of the allocation will have fewer or no bits allocated to them than the above calculation would assign. This is handled as follows to insure convergence: if the sum of the bits to be allocated to a carrier (CarrierBits) and the bits that have been allocated

before to the carrier (TotalBits) are larger than the total bit allocation (BitsPerDMTSymbol), then the "CarrierBits" will be set to the difference between the "BitsPerDMTSymbol" and "TotalBits".

If, for a given water level, fewer than BPDS bits are allocated, the water level will be lowered. The amount that the water level is lowered is proportional to the number of remaining bits, and inversely proportional to the number of carriers. The allocation is then performed again from the beginning with the new water level.

A single iteration of the benchmark is complete when all BPDS bits are allocated for a given water level. Upon exit, allocation results are returned in the 16-bit output array: CarrierBitAllocation and the final water level in dB are stored at "WaterLeveldB_out".

Analysis of Computing Resources

The Bit Allocation benchmark performs integer math on 16 bit signed quantities (e.g., the DeltadB and TotalBits calculation) as well as shift and logical compare operations (Water level comparison, etc.). These operations and accessing the data from memory are primarily what is tested by this benchmark. The buffer sizes in memory are relative small, which tend to take up residence in cache; therefore, this benchmark has a high cache hit rate for microprocessors with 16KB of Data Cache. The code size is small and easily fits in a small L1 Instruction Cache.

Special Notes

1. Each of the three packet sizes must be run to obtain an EEMBC Telemark™ score.

An Industry-Standard Benchmark Consortium

TeleBench™ Version 1.1

Benchmark Name: Convolutional Encoder

Highlights

- **Benchmark encodes data for forward error correction, as seen in wireless communication systems.**
- **16 bit & 8 bit integer math and logic.**
- **Multiple data sets (3)**
- **Fits inside small L1 caches to allow focus on CPU-centric performance.**

Application

This benchmark performs a generic Convolutional Encoder algorithm.

Convolutional Encoding adds redundancy to a transmitted electromagnetic signal to support forward error correction at the receiver. A transmitted electromagnetic signal in a noisy environment can generate random bit errors on reception. By combining Convolutional Encoding at the transmitter with Viterbi Decoding at the receiver, these transmission errors can be corrected at the receiver, without requesting a retransmission.

This benchmark provides an indication of the potential performance of a microprocessor , when used to generate convolutional codes as used in forward error correction.

Benchmark Description

The Convolutional Encoding benchmark provides a generic algorithm for producing a sequence of BranchWords from DataByteSize number of serial input DataBits. The algorithm is generic because generating polynomials are passed parameters from the EEMBC Test Harness. The characteristics of the generating polynomials are unique for each data set, and are controlled by NumberCodeVectors, ConstraintLength, and CodeMatrix.

NumberCodeVectors indicates the number of generating polynomials. ConstraintLength is equal to one plus the number of delayed DataBit values required for the generating polynomials. CodeMatrix is an array of size ConstraintLength by NumberCodeVectors. The values in a column of the code matrix (zeros or ones) correspond to the current and delayed DataBit values, indicating which terms are present in the generating polynomial.

By using generating polynomials that are functions of current and previous input DataBits, the Convolutional Encoder generates a number of output BranchWords per DataBit equal to the NumberCodeVectors.

The EEMBC Test Harness can request one of the three generating polynomials listed below. In these equations, the notation "D4", for example, means "the DataBit that occurred four bits prior to the current DataBit." G0 and G1 are the output BranchWords. The "+" operation is implemented as a bitwise exclusive OR in the benchmark.

Generating Polynomials:

- Test case **xk5r2dt** -- ConstraintLength=5, NumberCodeVectors=2
  G0 = 1+D2+D3+D4 (octal 27)
  G1 = 1+D+D4 (octal 31)

- Test case **xk4r2dt** -- ConstrantLength=4, NumberCodeVectors=2
  G0 = 1+D1+D2+D3 (octal 17)
  G1 = 1+D2+D3 (octal 13)
- Test case **xk3r2dt** -- ConstraintLength=3, NumberCodeVectors=2
  G0 = 1+D1+D2 (octal 7)
  G1 = 1+D2 (octal 5)

Analysis of Computing Resources

The Convolutional Encoder performs 16-bit signed & 8-bit unsigned operations, bitwise exclusive-OR operations, and bytewise shifts. This benchmark comprises 20 lines of executable C-code. Data sets use a maximum of 512 DataBits per iteration.

Special Notes

1. All three convolutional encoder data sets must be run to obtain an EEMBC Telemark™ score.

TeleBench™ Version 1.1

Benchmark Name: Fast Fourier Transform (FFT)

Highlights

- **Implements a decimation in time 256 fixed point 16 bit FFT using a Butterfly technique**
- **Allows for interleaved or non-interleaved data**
- **Typical FFT used in Telecommunications applications (e.g. a mobile/cellular phone)**
- **Multiple (3) data sets: sine, pulse, and high frequency**
- **FFT is benchmarked; Inverse FFT (iFFT) is included in the code for analysis purposes only**
- **Integer Math with complexity; fits inside small L1 caches.**

Application

The Fast Fourier transform benchmarks perform tests of a very fundamental algorithm that underlies a wide variety of signal processing applications. A Fourier transform performs a frequency analysis of a signal and therefore can be used for filtering frequency-dependent noise or interference of a transmission, for identifying the information content of a frequency-modulated signal, and many other purposes. A good general reference for Fourier transforms, algorithms for computing them, and some of their signal processing applications may be found in *The Digital Signal Processing Handbook*, Vijay K. Madisetti and Douglas B. Williams, Eds. (CRC Press, Boca Raton, FL, 1998). **The benchmark provides an indication of the potential performance of a microprocessor in a core task used in a wide variety of telecommunications applications.**

The FFT benchmarks apply to discrete data, which may be obtained for example from an analog-to-digital converter applied to a continuous signal. All benchmark FFTs use decimation in time and are performed on 256 16-bit complex points. All data are in fixed-point format, and therefore scaling must be performed, as needed, to prevent arithmetic overflow. Three different varieties of input data are used: a square pulse, a high-frequency test module, and a sine wave of a certain frequency. Benchmark scores for each variety are reported separately. The initial bit-reversal step is explicitly included.

Benchmark Description

A Fourier transform operates on the principle that a set of N stochastic input data points can be Fourier expanded in terms of N orthogonal exponentials of period N, taken as exp[-j(2□/N)kn]. The n^{th} element of the data is expressed as a sum over k=0,…,N-1 of these trigonometric factors (known as twiddle factors), each multiplied by a frequency coefficient. In most cases the data are available and the frequency coefficients are desired. These are obtained by a similar expansion in terms of the data; this direction is known as the *forward* transform. The absolute squares of these coefficients specify the strength of each frequency in the variations of the data. By convention the input data are said to lie in the time domain, such that each data point comes

at a fixed time interval from the preceding. The output coefficients then are said to lie in the frequency domain. An FFT algorithm will produce N such coefficients separated by a fixed frequency interval.

For example, if all data points had exactly the same value then the only non-zero frequency coefficient would be the one at zero frequency (the DC component), since no variation is present.

A fast Fourier transform takes advantage of the fact that the trigonometric factors repeat periodically. Therefore partial sums can be formed that can be reused many times, so an FFT will take an amount of time proportional to Nlog(N) instead of N^2, as brute-force computation of a Fourier transform to obtain N coefficients from N data points would require. The base to which the logarithm is taken depends on the algorithm employed. Very many algorithms are available, depending primarily on the number of points N. Many common applications rely on the fact that a single Fourier transform can be decomposed into two equal-sized transforms that are combined at the end. Proceeding in this manner to repeatedly subdivide the data in halves, one will arrive at a set of simple two-component transforms that must be combined, provided the number N of data points is a power of 2. Each subdivision is a "stage;" there will be $\log_2(N)$ stages in the computation using this technique. This is the basis of the "radix-2" algorithm, the implementation used in the EEMBC Out-of-the-Box FFT algorithm.

A "bit-reversal" re-ordering step is required to complete an FFT because the output coefficients do not otherwise occur in increasing frequency order. This step may be performed on the input data ("decimation in time," DIT) or on the output frequencies ("decimation in frequency," DIF). All EEMBC benchmarks use DIT.

The execution speed of an FFT has had a revolutionary impact on the digital signal-processing industry. The FFT is a fundamental component of very many signal-processing applications.

This benchmark performs an FFT with three different assumptions on the shape of the input data. These shapes are a sine wave, a square pulse, and a high-frequency test module. The shape may or may not affect the timing and the accuracy of the output. All input data is 16-bit complex and the FFTs are performed on N=256 points.

The twiddle factors are supplied in the test harness. Input data and/or twiddle factors may have real and imaginary parts interleaved or sequential, as specified by C preprocessor parameters (i.e., defined at compile time). Whichever choice is made, both have to be treated the same. These choices do not affect timing. Default is for both to be interleaved. The bit-reversal indices are also pre-computed and supplied as part of the test harness.

An inverse Fourier transform is also possible (see separate datasheet). This computation is very similar except that it begins with N equally-spaced

frequency coefficients and returns N equally-spaced time-domain data. A Fourier transform followed by its inverse should yield the original data, unchanged except for computation errors. The default for all EEMBC benchmarks is in the forward direction. The direction may be set by a preprocessor parameter.

Analysis of Computing Resources

The forward FFT benchmark performs integer math on 16-bit signed quantities (the time-domain input data and twiddle factors). Both data and twiddle factors are assumed to be complex and will therefore each require 256×2 16-bit locations in cache or memory; the output frequency coefficients will require the same amount.

The code size is small and fits ina small L1 instruction cache.

It is left to the user to run the benchmark through enough iterations to amortize the overhead associated with the test harness and initial cache misses. The default for this benchmark is 1000 iterations. ECL will double-check this with other values. Assumptions about data values would be imprudent, as ECL has its own private data sets.

Special Notes

1. Each of data files must be run to obtain an EEMBC Telemark™ score.

TeleBench™ Version 1.1

Benchmark Name: Viterbi Decoder

Highlights

- **Benchmarks ability to process a forward error corrected stream**
- **Algorithm handles IS-136 channel**
- **Input is packet of 344 6-bit values**
- **Implements add-compare-select**
- **Includes four distinct data sets**

Application

The Viterbi Decoder benchmark exploits redundancy in a received data stream to be able to recover the originally transmitted data. **The benchmark provides an indication of the potential performance of a microprocessor to be able to process a forward error corrected (FEC) stream using the Viterbi algorithm for decode.**

A communication channel that is corrupted by noise typically uses FEC to maintain transmission quality and efficiency. One such FEC mechanism is the use of Convolutional encoding (see the Convolutional Encoding EEMBC benchmark datasheet) at the transmitter and the use of Viterbi decoding at the receiver. The Viterbi decode process is an "asymptotically optimum" approach to the decoding of Convolutional codes in a memory-less noise environment. This benchmark implements a Viterbi decoder that would be used to handle an embedded IS-136 channel.

Benchmark Description

The benchmark implements a soft decision Viterbi decoder. The input is a packet of 344 6-bit values each of which represents a pair of encoded bits (i.e. the input bit stream was produced by a ½ rate Convolutional encoder which generates a pair of output bits for each input bit). The 3-bit value of each bit represents a soft decision value in the range 0 to 7. The value 0 indicates a strong indication that a "1" has been received whilst a value 7 indicates a strong indication that a "0" has been received. The generator polynomials used for the Convolutional encode process are:

$$1 + x + x^3 + x^5$$
$$1 + x^2 + x^3 + x^4 + x^5$$

The Viterbi decoding algorithm is best viewed from the perspective of the trellis, for which the reader is referred to the relevant literature. The trellis describes the state diagram of the convolutional encoder as it evolves through time.

The decode process consists of a number of processes which are described below:

Compute Branch Metrics

This process progresses forwards through the trellis and attempts to calculate

at each stage the distance between the received code word and all of the possible channel code words that could have been received.

Add Compare And Select (ACS)

Takes the branch metrics and computes the partial path metrics at each node in the trellis. The surviving path at each node is identified and the state history table updated accordingly.

Select Minimum Path Metric

Once the computation of branch metrics and ACS is the complete the state with the minimum path metric from the last stage of state history table is selected. This is the starting point for the trace back.

Trace Back And Recover Data

Using the starting point at the end of the state history table with the minimum path metric iterate back through the state history table, compute and then store the bit that would cause each state transition.

Analysis of Computing Resources

Viterbi decode is a computationally expensive process. The benchmark explores the target CPU's ability to perform loops, bit-wise operations, table-lookups, comparisons and basic arithmetic operations.