Improved Algorithms for Implementation of MPEG2 AAC Decoder on FPGA

Rajath R. Shenoy, Sudhir S. Naik, and Sumam David S., *Senior Member IEEE*
Deptartment of Electronics & Communication Engineering
National Institute of Technology Karnataka, Surathkal
Mangalore, INDIA
sumam@ieee.org

***Abstract*— This paper discusses methods to implement the IMDCT filter bank, Noiseless decoder, Inverse quantiser and Scale factor application modules of MPEG-2 Advanced Audio Coding decoder more efficiently when implemented on FPGAs. The efficiency of the algorithms has been validated through implementation on Xilinx Virtex II FPGAs.**

I. INTRODUCTION

The goal is to implement MPEG-2, Advanced Audio Coding (AAC) decoder efficiently on System on Programmable Chip (SOPC) or Field Programmable Gate Arrays (FPGA).

MPEG-2 AAC is a state of art audio coding standard which is replacing the in vogue MP3, as portable music download standard. It has better compression rates and perceptual quality better than CD quality audio at bit rates of 96kbps [1] as compared to MP3. The basic structure of the MPEG-2 AAC system is shown in Fig. 1. The functions of the decoder are to find the description of the quantized audio spectra in the bit stream encoded as per [2], decode the quantized values and other reconstruction information, and reconstruct the quantized spectra using tools activated by the bit stream. The signal spectra as described by the input bit stream, is then converted from frequency domain to the time domain, with or without an optional gain control tool. Following the initial reconstruction and scaling of the spectrum reconstruction, there are many optional tools that modify one or more of the spectra in order to provide more efficient decoding.

Traditionally the design of audio decoding systems would be centred on a low-power, low-cost Digital Signal Processor (DSP) as central processing core. The general-purpose DSPs by definition perform relatively well over a wide range of applications rather than addressing special-purpose needs. The serial instruction stream inherent to microprocessor architectures limits DSP performance.

DSPs do well in applications dominated by control flow or decision making but FPGAs are more suited for computationally intensive tasks that can be executed in parallel. FPGA offer flexibility and scalability in hardware. The ability to manipulate FPGA logic at the gate level allows designers to create a custom processor that can efficiently implement the function the application requires and simultaneously perform

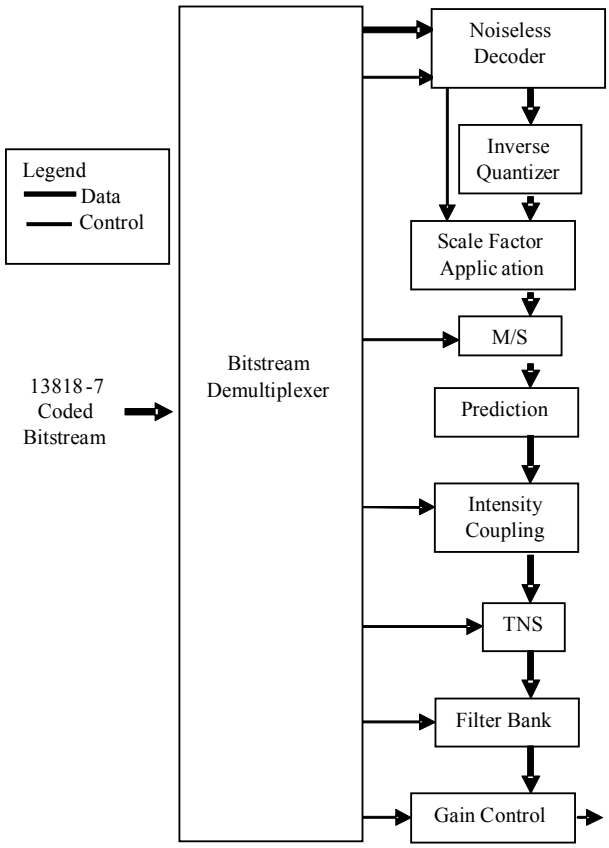

Figure 1 MPEG-2 AAC Decoder Block Diagram [2]

all application sub functions in parallel. Further, new families of FPGAs are integrated with lots of block memory, signal processing function blocks, are of low power, and are competitively priced, especially on a price performance scale [3].

The input to the bit stream demultiplexer is the MPEG-2 AAC bit stream. The demultiplexer separates the data stream into the parts for each tool. The noiseless decoding tool parses the bit stream, decodes the Huffman coded data and reconstructs the quantised spectra and the Huffman and DPCM coded scale factors. The inverse quantiser tool takes the quantised values for the spectra, and converts the integer values to the non-scaled, reconstructed spectra. The M/S tool converts

spectra pairs from Mid/Side to Left/Right under control of the M/S decision information in order to improve coding efficiency. The prediction tool reverses the prediction process carried out at the encoder. The intensity stereo/coupling tool implements intensity stereo decoding on pairs of spectra. The temporal noise shaping (TNS) tool implements a control of the fine time structure of the coding noise. The filterbank tool applies the inverse of the frequency mapping that was carried out at the encoder. An inverse modified discrete cosine transform (IMDCT) is used for the filterbank tool. When present, the gain control tool applies a separate time domain gain control to each of 4 frequency bands that have been created by the gain control filterbank in the encoder.

This paper suggests methods for implementing the following four AAC decoder modules more efficiently.

- Filterbank
- Inverse Quantization
- Scale factor Application
- Noiseless Decoder

II. FILTERBANK

The filterbank converts the frequency-domain signals into time-domain signals. The encoder uses Modified Discrete Cosine Transform (MDCT) to convert the signals from time domain to the frequency domain where it is then compressed and transmitted. At the decoder the signals in frequency-domain are transformed into time-domain using IMDCT. This is then followed by windowing and overlap addition of the windowed coefficients.

$$x(n)=\frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1}X[k]\cos\left(\frac{2\pi}{N}(n+n_0)\left(k+\frac{1}{2}\right)\right),\quad 0\le n<N \qquad (1)$$

where, n is the sample index; k is the spectral coefficient index; N is the window length based on window sequence value and $n_0= (N/2 +1)/2$

A. Butterfly implementation

Fast algorithms for implementing Discrete Cosine Transform (DCT) exist [4]. Implementation of IMDCT is almost equivalent to that of DCT-IV.

DCT-IV(N)=A(N-1)+M(N)+DCT-IV(N/2)+DCT-III(N/2) (2)

where DCT-III(N) and DCT-IV(N) are the arithmetic complexity of the type-III and type-IV DCT with length N. A(x) and M(y) indicate that number of real additions and multiplications required are x and y, respectively [5]. Though this implementation reduces the number of additions and multiplications, it inherently comes with complexity. Study of previous implementations show that a straight forward implementation with ROMs to store the DCT and windowing coefficients, was more efficient. This is because the time taken by arithmetic operations is not the bottleneck in the implementation of the entire block. Another benefit from this approach is that the amount of logic used in the DCT i.e. the area occupied on the FPGA is small. Other approaches to calculate IMDCT faster and in a more efficient way were also examined.

B. Using Fast Fourier Transform (FFT)

A relation between IMDCT and FFT was found and this implementation was used to calculate IMDCT [3].

$$\begin{aligned}
x(n) &= \frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1}X[k]\cos\left(\frac{2\pi}{N}(n+n_0)\left(k+\frac{1}{2}\right)\right)\\
&= \frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1}X[k]\,\mathrm{Re}\left\{e^{\frac{-j2\pi}{N}(n+n_0)\left(k+\frac{1}{2}\right)}\right\}\\
&= \frac{2}{N}\mathrm{Re}\left\{\sum_{k=0}^{N-1}X[k]e^{\frac{-j2\pi k}{N}(n+n_0)}e^{\frac{-j\pi}{N}(n+n_0)}\right\}\\
&= \frac{2}{N}\mathrm{Re}\left\{NptFFT\left(X(k)e^{\frac{-j2\pi k}{N}n_0}\right)e^{\frac{-j\pi}{N}(n+n_0)}\right\}
\end{aligned} \qquad (3)$$

$$x(n)=\frac{2}{N}\mathrm{Re}\left\{NptFFT\left(X(k)e^{\frac{-j\pi k}{N}\left(\frac{N}{2}+1\right)}\right)e^{\frac{-j\pi}{N}(n+n_0)}\right\} \qquad (4)$$

Two functions were written in MATLAB to calculate 2048 point IMDCT; first based on (1) and the second based on (4). On running these two functions on a Pentium 4, 2.0GHz, 512 MB system, we obtained the following results averaged over 10 runs.

Time taken by first function = 0.603 seconds
Time taken by second function = 0.0122 seconds

The second function is 49.426 times faster than the first function.

The IMDCT module using (4) was implemented in VHDL using Xilinx System Generator. The implementation block diagram is shown in Fig. 2. The FPGA chosen was Virtex II XC2v2000 bf 957-6. The device utilization summary for Filterbank is

No. of slices	4075 of 10752	38%
No. of Flip Flops	5158 of 21504	24%
No. of 4 input LUTs	5557 of 21504	26%
No. of Block RAMs	33 of 56	59%
No. of 18*18 Multipliers	31 of 56	55%

III. INVERSE QUANTIZATION

A non uniform quantiser is used in the encoder for quantisation of the spectral coefficients. Therefore the decoder must perform the inverse non uniform quantisation after the Huffman decoding of the scale factors and spectral data. The inverse quantization is described by

$$x\_invquant = sign(x\_quant) * |x\_quant|^{4/3} \qquad (5)$$

where *x_invquant* is the inverse quantized value, *x_quant* is the quantized value and *sign(x)* gives the sign of *x*.

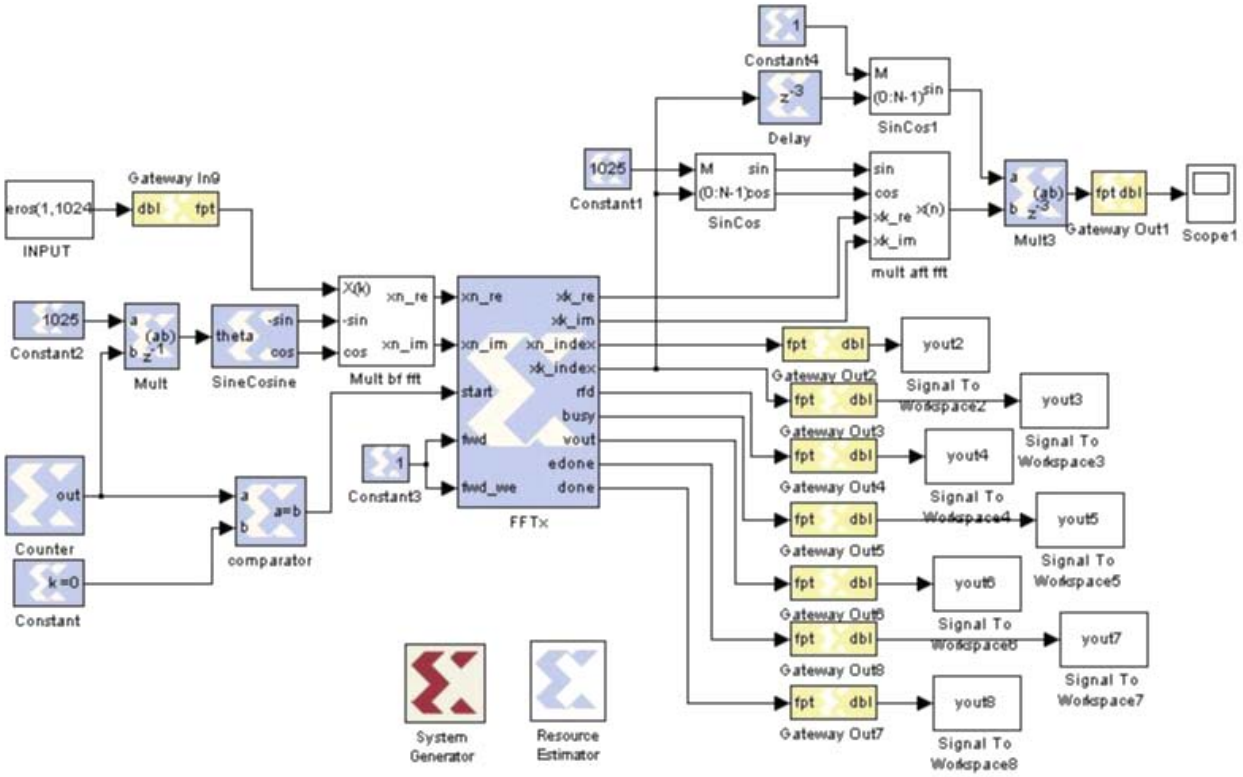

Figure 2 Simulink block diagram of Filter bank module

A. *Implementation*

The expression for inverse quantization is non-linear. Fixed point implementation of such non-linear expressions in VHDL is very difficult and consumes a lot of resources like multipliers and takes a lot of time for calculation. So a more efficient method of calculation of inverse quantized values is needed. One approach is the use of fixed point approximations of the function that could be used for approximate calculations.

An approximation using second order regression is given as [5]

$$x\_invquant = 0.04 * x\_quant^2 + 1.98 * x\_quant - 1.77 \quad (6)$$

Comparisons between actual values and these approximated values in Fig. 3 show that the approximations are valid only when the values of *x_quant* are small.

A Look Up Table (LUT) based implementation was chosen to compute inverse quantized values of noiselessly decoded spectral values. In this approach the values of inverse quantized values are computed in a floating point environment like C/ MATLAB and these are stored in Read Only Memory (ROM) in the FPGA after truncation at certain precision during conversion from floating point to fixed point The ROM locations are then addressed accordingly and the inverse quantized values corresponding to the quantized values are read out.

This implementation requires just one Block RAM on Xilinx Virtex II FPGAs.

IV. SCALEFACTOR APPLICATION

The role of the scale factor application tool in the decoder is to calculate the value of gain from the obtained scale factor values and to multiply the scale factor bands with corresponding gains. The scale factors are Huffman decoded and their actual value is decoded from their differentially encoded values. The analytic expression for calculation of gain from scale factor values according to ISO/IEC 13818-7 is as follows:

$$\text{Gain} = 2^{(0.25(SF-100))} \quad (7)$$

where SF is the scale factor value of a particular scale factor band.

A. *Implementation*

The expression for gain is also a non-linear relation like that of inverse quantization. It was also observed that the scale factor values have only integer values between 0 and 255 (included). So we resort back to the implementation strategy used in inverse quantization tool, i.e., we calculate gain values according to the 256 SF values in C/MATLAB and store it in a ROM of depth 256 and desired precision.

One other thing to be noticed here is that the value of gain becomes very large with increasing values of SF; this will lead to a lot of truncation errors in the tools that follow. In our decoder we found that with such high gain values, a lot of truncation was introduced in the tools that followed and also by the sound card of the computer and due to this a noisy output

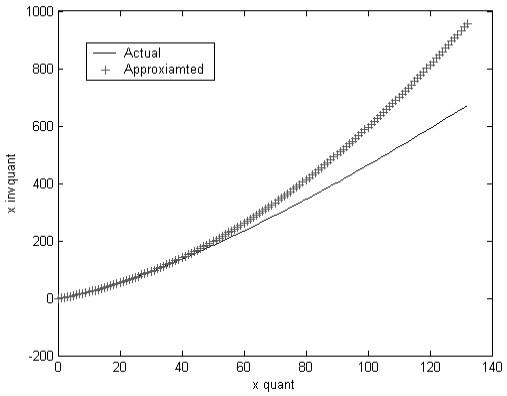

Figure 3 Comparison between Approximated and Actual Inverse Quantized Values

was obtained. Hence there is a need to reduce the value of gain. So a minor change was incorporated which then removed the noise from the output. The new expression for gain is:

$$\text{Gain} = 2^{(0.25(SF-156))} \quad (8)$$

The value 156 was chosen because the maximum value of gain was in the range of 2^{14}, and hence to make the range between -1 and 1, the gain had to be divided by 2^{14}. On dividing the value of gain by 2^{14} we get the above relation.

The implementation of this block too requires only one Block RAM in Xilinx Virtex II implementation.

V. NOISELESS DECODING

To reduce the redundancy of the scale factors and the quantized spectrum of each audio channel they are subjected to noiseless coding at the encoder. The spectral data and scale factor data for particular scale factor bands are differentially Huffman encoded as explained in [2]. The differential offsets are stored in 11 spectral data code books, and 1 scale factor code book. The process of decoding switching through various codebooks is explained in [6].

A. Implementation

We implemented the implementation strategy in [7] and adopted a memory usage optimized implementation scheme because the on board memory is being used up by other blocks. If this resource is still in abundance then the implementation algorithm choice can just be made on ease of decoding.

The algorithm would be to switch through a binary tree depending on every incoming bit as explained in [7], when we encounter a valid code, an index value is returned. A table containing linking data for each of these nodes will have to be stored in the memory. On getting the index then we can calculate 4-tuple or 2-tuple samples depending on codebook [2]. Evident performance enhancement could be obtained from software simulations.

To further improve memory usage, we can recognize that a part of the Huffman code follows a binary pattern. Hence the approach was to truncate the Huffman code recognition to just the non-binary portion of the content and adding the rest of bits as offset to get address.

VI. CONCLUSION

We have validated the proposed methods to implement the IMDCT filter bank, Noiseless decoder, Inverse quantiser and Scale factor application modules of MPEG-2 Advanced Audio Coding decoder more efficiently when implemented on Xilinx Virtex II FPGAs. Currently, we are integrating all the decoder tools on Xilinx Virtex II FPGAs. We also plan to try it on a SOPC environment.

REFERENCES

[1] www.mp3-tech.org/aac.html

[2] ISO/IEC “IS 13818-7 Information Technology- Generic Coding of Moving Pictures and Associated Audio Information - Part 7 Advanced Audio Coding (AAC)”.

[3] H. Najafzadeh-Azghandi and P. Kabal, “Perceptual Coding of Narrow band Audio Signals,” *Proc. IEEE Conference on Acoustics, Speech, Signal Processing* , Phoenix, AZ, March 1999, pp. 913-916.

[4] Chi-Min Liu and Wen-Chieh Lee, “A Unified Fast Algorithm for Cosine Modulated Filter Banks in Current Audio Coding Standards,” *Proc. AES 104th Convention*, Amsterdam, 1998, pp. 16-19.

[5] R. Linneman, “AAC on FPGA for MPEG-2,” *Proc. Innovation Expo* 2002.

[6] J. Kreiter , “Emerging Markets Create Demand for FPGA," *Chip Design*, Mar. 2004, pp. 42.

[7] Byeong-II Kim, Tae-Gyu Chang and Jong-Hoon Jeong, “An efficient search of binary tree for Huffman decoding based on numeric interpretation of codewords,” *Proc. ITC-CSCC2002*, Thailand, 2002.