IMPLEMENTATION OF A REAL-TIME AUDIO DECODER: ASCdec

LUÍS GUSTAVO P.M. MARTINS* (EMAIL: lmartins@inescn.pt)
GABRIEL F.P. FERNANDES* (EMAIL: gabriel.fernandes@inescn.pt)
ANÍBAL J.S. FERREIRA°* (EMAIL: ajf@inescn.pt)

°Department of Electrical and Computer Engineering of the University of Porto
*INESC Porto, Largo Mompilher 22, 4000 Porto, PORTUGAL

ABSTRACT

This paper describes the implementation of a proprietary perceptual audio decoder (*ASCdec*), for operation in real-time, in specific software and hardware platforms. The strengths of the algorithm will be highlighted, the implementation criteria and solutions will be described, and a perspective will be given on possible enhancements and system applications.

1. Introduction

Audio Spectral Coder (*ASC*) is a proprietary perceptual audio coder [1] that has been developed at INESC and that has been submitted as a coding proposal to the competitive phase of the MPEG-4 standardization activities in November 1995. Formal listening tests have indicated that in terms of coding efficiency, *ASC* compares to the well-known standard MPEG-1 Audio Layer 3 algorithm (Figure 1) [2]. In terms of coding performance, *ASC* compares favorably to the MPEG-1 Layer 3 standard since it achieves a similar subjective coding quality for the same bit rate, while implying a lower coding delay and system complexity [1] [3].

ASC has been optimized for the coding of monophonic audio, particularly at low bit rates, *i.e.* at less than 64 kbit/s per channel, and for a wide diversity of audio material, including speech. *ASC* allows a flexible choice of coding conditions and parameters namely:

- the sampling frequency that may range from 8 kHz till 48 kHz and includes the main sub-multiples of the professional and industry standards,
- the length of the basic audio frame which determines the coding delay and that may vary form 32 samples till 1024 samples,
- one out of three possible improved time resolution coding modes that is implemented by means of a window switching mechanism which is activated in non-stationary regions of the audio signal [1],
- the selectivity of the time analysis/synthesis filter (satisfying perfect reconstruction requirements) that depends on its shape,
- the average coding bit rate and the coding mode: constant or variable bit rate.

For a complete description of the *ASC* encoding algorithm please refer to [1][4].

The decoding counterpart of *ASC* consists of a set of actions that are taken while reading an *ASC* coded bit stream, the first of which is synchronization. These are illustrated by the diagram presented in Figure 2.

The demultiplexer insures the correct frame synchronization and verifies the decoder settings at each frame using the information that is sent in a fixed header for each coded audio frame. It then decodes information of a variable header (mainly related with the harmonic structure information) which is dependent of the information of the fixed header, and finally reads all Huffman codes related to the quantization step-sizes, quantized coefficients, and run-length values.

The interpolation of scale factors is performed as in the encoder [4] and any existing coded harmonic pattern is combined with the interpolated scale factors in order to reconstruct the quantization noise profile (QNP) [4].

After inverse quantization, an important operation takes place if a harmonic structure has been transmitted. In fact, in the case of a very coarse quantization of spectral coefficients, there is the possibility of missing harmonics appearing within a transmitted harmonic structure, *i.e.* there is the possibility of some relevant tonal components being quantized to zero. In this case, a better subjective quality is reached by synthesizing the missing harmonics using only the corresponding sign bits and quantization step-sizes. In order to allow for this processing, the encoder detects if relevant harmonic components are quantized to zero. If this is the case, the encoder sends the sign bit of each “missing” harmonic as side information [1].

The reconstructed spectral coefficients are then inverse transformed and the resulting time segments are overlapped and added in order to recover the audio signal with the same sample resolution and sampling frequency of the original signal.

This paper describes the implementation of a real-time *ASC* decoder (*ASCdec*), in both hardware and software versions. The rest of this paper is organized as follows. In section 2 we describe the implementation of the *ASCdec* algorithm on a low-cost floating point DSP platform using the Texas Instruments TMS320C31 Digital Signal Processor. In section 3 we address the software implementation of a real-time *ASCdec* on a Windows 9x environment, and the advantage of linking the development of the hardware and software versions in a very intimate way will be highlighted since common problems are shared and may be conveniently anticipated, at each time, by the most revealing implementation perspective. In section 4 we anticipate future developments to our system and we draw a few scenarios of application. Section 5 concludes this paper.

2. *ASCdec* implementation on the ‘C31 DSP

Although the TMS320C3x family of processors is the entry level of the Texas Instruments 32 bit floating point DSP range, its internal busing and special DSP instruction set have the necessary speed and flexibility allowing to execute up to 150 million floating-point instructions per second (MFLOPS). This level of performance is achieved by the new TMS320VC33 processor [www.ti.com].

The TMS320C3x processors optimize speed by implementing important functions in hardware, such as parallel multiply and arithmetic logic unit (ALU) operations on integer

or floating point data in a single cycle. Also included on-chip are a general-purpose register file, a program cache, a dedicated auxiliary register arithmetic units (ARAU), internal dual-access memories and one DMA channel supporting concurrent I/O. Furthermore, applications can be enhanced by the large address space, multiprocessor interface, internally and externally generated wait states, external interface ports (one in the case of the ‘C31), two timers, serial ports (one in the case of the ‘C31) and multiple interrupt structure [8].

The TMS320C31 used in this implementation operates at 60 MHz (33ns cycle time) executing operations at a performance rate of up to 60 MFLOPS and 30 million instructions per second (MIPS). The device provides 28 registers in a multiport register file that is tightly coupled to the CPU. All of these registers can be operated by the multiplier and ALU and can be used as 32 bit general-purpose registers. However, several registers also have special functions:

- eight 40 bit extended-precision registers are especially suited for maintaining extended-precision floating-point results,
- eight auxiliary registers support a variety of indirect addressing modes (bit-reversed and circular addressing included),
- several registers provide such system functions as addressing, stack management, processor status, interrupts and block repeat [8].

The TMS320C31 memory map consists of:

- Internal Program/Data memory (2 Kwords)
- Internal peripherals
- External memory accessed through an external memory interface (up to 16 Mwords)

The ‘C31 on-chip memory (2 Kwords) is clearly insufficient for a demanding processing, such as the *ASCdec*, implying the mandatory use of external memory. The ‘C31 allows up to 16 Mwords of external memory to be easily connected through its external memory interface (address bus and data bus) but some care must be taken when optimizing this expansion for maximum data I/O. The existence of only one channel of DMA in the ‘C31 processor limits the use of concurrent accesses to external memory that would compensate the inherently slower external memory accesses (specifically the store operation that takes twice the number of cycles of a read instruction when performed to external memory [8]). Furthermore, the use of one-cycle multiply and accumulate instructions can only be maximized with the use of zero wait-state external memories, and this is important because operands are likely to be stored in external memory.

Taking these considerations in mind, the system architecture was designed with the objective of achieving maximum data I/O and processing performance by the use of fast zero wait-state external RAM. A simplified diagram of the system architecture is depicted in Figure 3.

A boot flash memory is installed in the system and stores the start-up code that is boot-loaded each time the system is powered-up or reset. The use of flash memories proved to be more flexible than their EPROM counterparts, specifically when performing

program code updates, since it is very convenient to erase and rewrite them by means of software intervention only.

A high-quality 24 bit D/A converter is connected to the ‘C31 serial port and is the responsible for the output of the decoded audio stream.

The *ASC* encoded audio data can be stored in a solid-state memory device connected to the system, such as a flash memory or PCMCIA cards. Audio frames are then read and decoded in real-time by the ‘C31 processor and the resulting PCM audio is then played by means of the D/A converter. These *ASC* encoded audio streams can also be downloaded from a HOST device, such as a PC, by means of a parallel port interface also connected to the system (Figure 3).

Control signals such as PLAY, STOP, PAUSE are also received through this parallel port interface when it is connected to an user interface.

The current ‘C31 *ASC* Player implementation favors processing speed over memory requirements by the use of extensive optimization methods like inline functions, look-up tables (LUTs) and compiler options that perform speed optimizations at the cost of code size.

In the current version, the ‘C31 *ASC* Player needs a total of 76 Kwords (about 300Kbytes) of RAM memory (including the ‘C31 on-chip 2 Kwords of memory) and 210 Kbytes of Boot Flash memory. These figures are likely to decrease in future versions due to additional code optimizations.

3. *ASCdec*: from simulation code to the software and hardware implementation

The implementation of the *ASCdec* “C” source code on the ‘C31 platform was, at each and every step, anticipated by the simulation of each decoder block as a “C” language Windows 9x application, ultimately resulting in two *ASC* player versions: a Windows 9x software player and a hardware ‘C31 based player.

The software version consists of a real-time Windows9x software player whose graphical user interface is depicted in Figure 4. This software player reads an *ASC* compressed audio stream stored on disk, decodes it and sends the resulting PCM decoded audio to the Windows9x multimedia system for real-time reproduction.

Similarly, the ‘C31 hardware implementation of the *ASC* decoder reads the compressed audio data stored in solid-state memory (downloaded from a HOST PC through the parallel port interface to a FLASH memory or PCMCIA card installed in the system), decodes it and sends the recovered PCM audio through the CS4390 D/A converter for reproduction.

These two players reflect the same algorithm structure (depicted in Figure 2) and share the same “C” source code with the exception of several optimizations introduced in critical processing sections and that are platform dependent (*e.g.* assembly language programming and I/O routines).

This very intimate linking of the development of these two *ASC* player versions has as main advantage: the sharing of common problems that can be conveniently anticipated, at each time, by the more revealing implementation perspective. In fact, this approach has

allowed us to interactively introduce optimizations in one specific implementation platform and then reflect them into the other implementation platform. The cross-match of test results has indicated that this procedure led to improved solutions that would otherwise be difficult to reach on an individual implementation basis.

The implementation of the *ASC* decoder on the ‘C31 started by directly compiling the “C” source code that resulted from the development of the Windows 9x real-time *ASC* player generating by itself an already clean ‘C31 object code.

Unfortunately this code is far away from real-time performance on the ‘C31 platform. Using the profiler it is possible to determine the execution time in number of cycles for each “C” function and consequently, it has been possible to identify inefficient sections of the “C” code. Time-consuming routines like IODFT, Overlap-Add and some mathematical power functions were the first targets to be replaced by highly optimized (manually written) ‘C31 assembly code.

One of the important assembly optimization aspects concern pipeline conflicts. Correct handling of branch, register and memory conflicts allow a better algorithm performance. Before the IODFT computation, data buffers to be processed are placed in internal memory, since write instructions to external memory require two bus cycles to complete, instead of one.

The use of Decrement and Branch Delayed instructions (DBD) in replacement of nested Repeat Block (RPTB) ones (these require multiple PUSH and POP instructions for context switching) also led to time savings.

Particular attention was given to the parallel execution of the maximum possible number of instructions. The butterfly loop of the IODFT is one of the most intensive parts of the algorithm. Having this particular part implemented with as many parallel instructions as possible, resulted in a great performance improvement.

The replacement of some mathematical functions by look-up tables (LUTs) and the optimization of the Huffman decoding routines also resulted in substantial additional reduction of the computational load.

Table 1 presents the profiler worst case results for each of the most critical code sections of the *ASCdec* algorithm, when executed on the ‘C31 processor.

Different levels of optimization are considered, which places in evidence the importance of manual assembly language programming and the effects of turning on the “C” optimizer (flag –o3 [9]).

Four main blocks of the *ASC* decoder and the overall decoding process (TOTAL column) are profiled. These profiler results do not take into consideration the I/O instructions responsible for the output of the decoded audio since they do not directly make part of the decoding process core. Its suffices to mention that these I/O routines implement a double-buffered structure and minimize CPU disturbance by the use of highly optimized data transfer techniques such as DMA transfers.

An impressive overall processing reduction (79%) is achieved by the simultaneous use of manual assembly language programming and maximum level of “C” compiler optimization (flag –o3 [9]) (please refer to the TOTAL column in Table 1). This reduction was mainly due to the direct assembly programming in an optimized way of

important sections of *ASCdec* such as the IODFT and overlap-add, as mentioned previously.

The IODFT implementation, although clearly benefits from the manual assembly code optimization (53% reduction, maximum), takes a much more significant advantage of the "C" optimizer (79% reduction, maximum), when considering their separate effects. But the best results are achieved combining both techniques, resulting in a joint performance gain of about 90%. In this situation the effect of the "C" optimizer is not so obvious because the IODFT core is now implemented in highly optimized assembly code, and thus its importance lies mainly in the optimization of the assembly/"C" code integration.

A performance gain of about 97% resulted by combining the overlap-add routine manual assembly code optimization with the elimination of some unnecessary fixed-point conversion routines that are present in the original "C" source code (the 'C31 provides one-cycle instructions that perform floating-point to fixed-point conversions [8]), turning the "C" optimizer impact almost negligible.

Considering the additional performance gains achieved in the Huffman decoding and spectral reconstruction routines (by means of the use of both manual and automatic optimization methods) these results clearly demonstrate the importance of manual assembly language optimizations in critical code sections besides the use of automatic "C" optimizers.

4. Future Developments and Applications

Taking into account that Texas Instruments has just released a new faster low-cost 'C31 code compatible TMS320VC33 DSP (150 MFLOPS max.), it will be possible in the near future to extend the system's capabilities to handle stereo and multi-channel *ASC* decoding, following, this way, the expected evolution path for a system of this nature.

Furthermore, the decomposition of the TDAC filter bank in the *ASCdec* algorithm structure as a two-step operation [5], one of them being the inverse odd discrete Fourier transform (IODFT) (Figure 2), makes available a convenient spectral representation of the decoded audio signal that easily allows the future integration of several post-processing functionalities, like equalization [6] and time-scale expansion [7], taking good advantage of the extra processing power available on these new faster processors.

A secure coding system based on the proprietary *ASCdec* algorithm, which is amenable to be implemented on a single low-cost floating point DSP and capable of integrating many additional features, as described in this paper and elsewhere [4], has potential application in many areas, such as:

- solid-state portable audio players,
- music preview, on a local/narrow utilization scope such that associated to a music store, real or virtual,
- automatic public announcement or information systems (*e.g.* museums),
- automatic broadcasting based on playing lists of compressed audio material (*e.g.* radio stations),

- high-quality communications (*e.g.* via ISDN between reporters on the field and the radio station),
- high-quality audio books for children and blind people.

In addition, given that *ASC* encoded bit streams embed specific information concerning important features of the audio signal such as stationarity, pitch and harmonicity, makes that *ASC* finds natural application whenever a requirement exists to classify or to access audio documents using semantic criteria, by just looking at their representation in the compressed domain.

5. Conclusions

Audio Spectral Coder (ASC), a proprietary perceptual audio coder developed at INESC, has been successfully implemented in real-time as a Windows 9x application and as a DSP based hardware player. We have described its implementation as a hardware player based on a single low-cost Texas Instruments TMS320C31 as well as a software player on the Windows 9x environment. A few profiler results have been presented and several solutions for DSP code optimization have been discussed. We have also addressed additional work to be carried out in the future in order to further optimize the two *ASCdec* versions (hardware and software) so as to target a wide range of applications.

REFERENCES

[1] Aníbal J.S. Ferreira, *"Audio Spectral Coder"*, 100th Convention of the Audio Engineering Society, May 1996, Preprint n. 3992.

[2] ISO/IEC JTC 1/SC 29/WG 11, ISO/IEC N1144, 33rd MPEG Meeting, January 1996.

[3] Nikil Jayant, James Johhnston and Robert Safranek, *"Signal Compression Based on Models of Human Perception"*, Proceedings of the IEEE, Vol. 81, no. 10, pp. 1385-1422, October 1993.

[4] Aníbal J.S. Ferreira, *"Spectral Coding and Post-Processing of High Quality Audio"*, PhD thesis, Faculdade de Engenharia da Universidade do Porto – Portugal, November 1998.

[5] Aníbal J.S. Ferreira, *"Perceptual Audio Coding and the Choice of an Analysis/Synthesis Filter Bank and Psychoacoustic Model"*, 104th Convention of the Audio Engineering Society, May 1998, Preprint n. 4671.

[6] Gabriel F.P. Fernandes, Luís G.P.M. Martins, Miguel F.M. Sousa, Filipe S. Pinto, Aníbal J.S. Ferreira, *"Implementation of a new Method to Digital Audio Equalization"*, 106th Convention of the Audio Engineering Society, May 1999, Preprint n. 4895.

[7] Aníbal J.S. Ferreira, *"A New Frequency Domain Approach to Time-Scale Expansion of Audio Signals"*, ICASSP '98 Proceedings, Volume: 6, Page: 3577, Paper number:1671, May 1998.

[8] Texas Instruments, *"TMS320C3x User's Guide"*, Digital Signal processing Products, SPRU031D, 1994.

[9] Texas Instruments, *"TMS320C3x/C4x Optimizing C Compiler User's Guide"*, Digital Signal Processing Solutions, SPRU034G, 1997

TRIAL	ASCDEC 'C31 CYCLES – WORST CASE				
	HUFFMAN DECODING	SPECTRAL RECONSTRUCTION	IODFT	OVERLAP-ADD	TOTAL
UN-OPTIMIZED "C"■	**206 730**	**124 967**	**598 708**	**253 325**	**1 249 143**
UN-OPTIMIZED "C" MIXED ASSEMBLY*	**192 152**	**73 461**	**284 914**	**8 405**	**623 376**
OPTIMIZED (-O3) "C"■	**154 533**	**93 663**	**129 284**	**89 630**	**488 647**
OPTIMIZED (-O3) "C" MIXED ASSEMBLY*	**138 149**	**41 496**	**62 142**	**8 397**	**271 228**

■ "C" code without manual assembly optimizations

* "C" code with manual assembly optimizations

Table 1 – TMS320C31 *ASCdec* code profiling

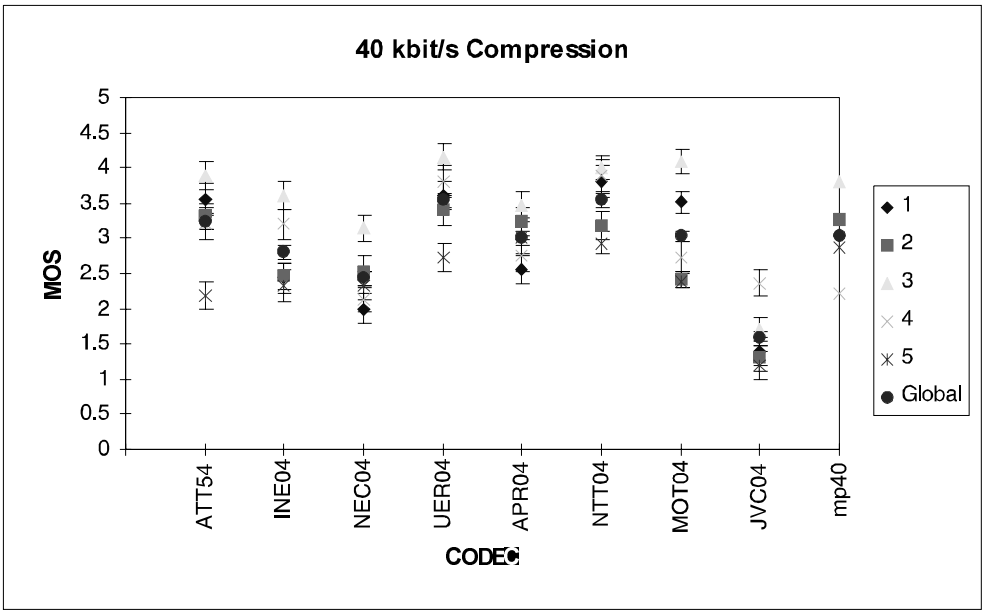

Figure 1 – MPEG-4 Audio test results for the functionality of compression at 40 Kbit/s. The *ASC* coder is identified by "INE04" and the MPEG-1 Layer 3 coder is identified by "mp40". The symbols numbered 1-5 denote specific test items [2].

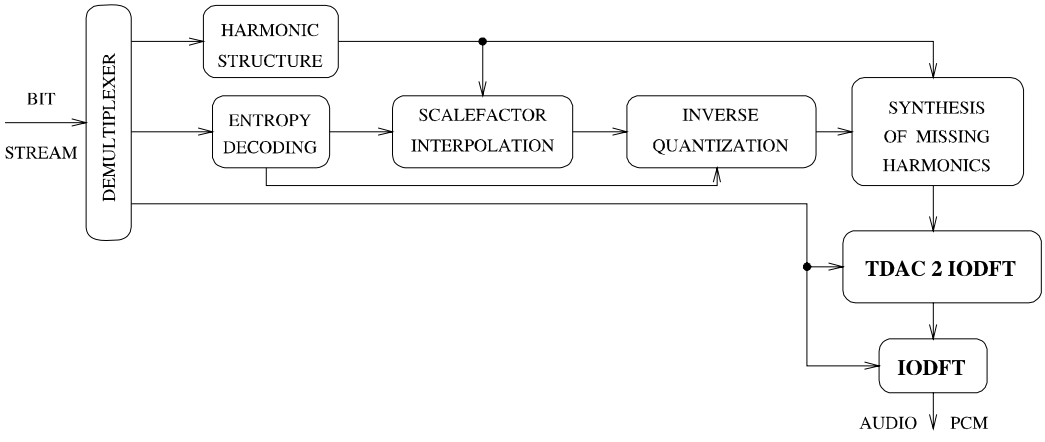

Figure 2 – *ASCdec* Algorithm Structure

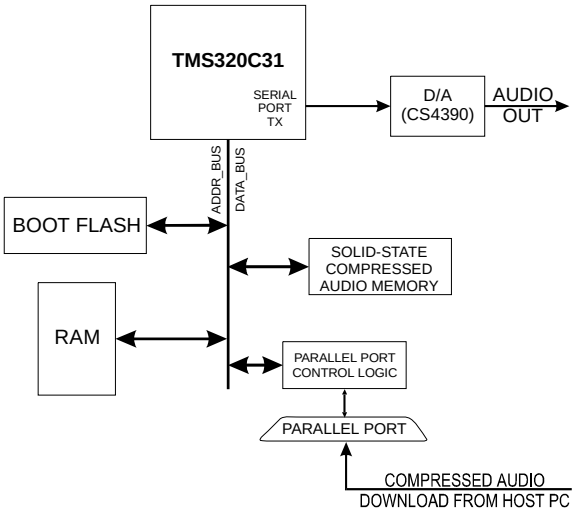

Figure 3 – System Architecture

Figure 4 – Windows 9x Graphical User Interface