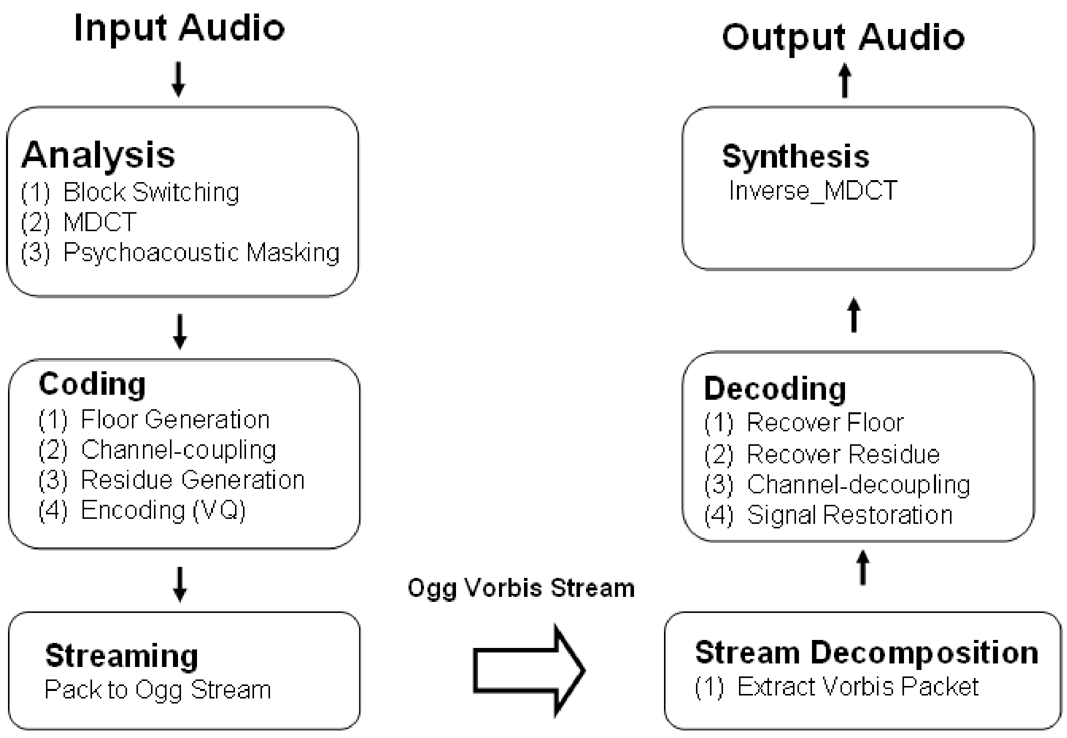
[Introduction]

OGG Vorbis[1] is a hybrid time/frequency transform coder, which is a lossy, asymmetrical audio compression format developed by the Xiph.org foundation. Audio data usually is encoded by the Vorbis encoder, packed into an Ogg bit-stream and then transported to the decoder.

Like mp3 (and virtually every other useful transform coder), the input audio is first analyzed for strong changes and natural breaks in, and then the encoder use this information to break up the incoming audio into differe nt sized blocks. Encoding and decoding of Ogg Vorbis can be classified in 6 stages [2]. Vorbis encoder takes the raw audio data as overlapped but contiguous short -time segments and analyses the audio data to find the optimal small representation. This s tage is called *Analysis*. After that, it encodes the audio data into a much smaller data representation as determined in the previous step. This stage is called *Coding*. Then the raw packages will be packed into streams, called *Streaming*. At the receiver end, the decoder extracts the sequence of raw packages from the stream, the stage is called streaming *Decomposition*. It then tries to reconstruct the sound signal representation from these packages, call *Decoding*. Lastly, the audio signal will be regene rated from the decoded representation in the *Synthesis* stage. The stages are shown below:



During the compression process, if one loses information in the frequency domain, the resulting noise spreads throughout the time domain. A very strong spike in time will get smoothed out by frequency quantization, so the larger the block, the more audible it is. Therefore, in *Analysis* stage, audio data is divided into overlapping blocks of 2 sizes: short (256 samples) or long (2048 samples). Normally long window will be used except when sudden attacks or explosive sounds occur because short window can prevent temporal spreading artifacts which might be produced by MDCT in case of long window.

In coding stage, information received from psychoacoustics masking proce ss will be used to create the spectral envelope of the signal and floor function. Channel coupling and residue generation will be generated afterwards. Small representation of the audio data (floor and residue) will be encoded using Vector Quantization (V Q) to form a Vorbis package which will be later, with the VQ codebook, packed into OGG bit-stream. The audio packet is finally embedded into an Ogg bitstream page and the page (when full of packets) is shipped out in the stream.

Decoding is straight forward and less complex than encoding. The decoder does the reverse, but without all the masking analysis. The decoder extracts the string of packets from the Ogg bitstream, extracts floor, residue and does channel-decoupling. The time-audio frame is recovered by inverse MDCT. Each frame is lapped and de-overlapped to form the output audio signal.

[Performance Observation]
It has been reported that the Vorbis player can run successfully on the ARM7TDMI embedded processor. However, for other low-end embedded processors, real-time decoding is not achievable. For optimization purposes, we used GNU *gprof* tool, and the UNIX's *top* command to observe the performance of a typical Ogg decoder (decoder_example.c) that came with the reference Vorbis library. These observations can provide the performance information of Vorbis decoding process. We used a RISC based Sun Sparc machine as out the testing platform. The information got from *top* shows that the memory consumption is around 1.36 MB. This memory usage makes is suitable to be executed on most embedded processor. The following profile shows for each function, which functions called it, which other function it calls and how many times:

```
Flat profile [Top 10 only]:
Each sample counts as 0.01 seconds.
```

time	Cumulative seconds	Self seconds	calls	self ms/call	Total ms/call	name
30.34	6.15	6.15	40160	0.15	0.15	mdct_backward
14.85	9.16	3.01	658029	0.00	0.01	orbis_book_decodevv_add
12.58	11.71	2.55				internal_mcount
8.04	13.34	1.63	1	1630.00	17599.89	main
5.23	14.40	1.06	40144	0.03	0.03	_vorbis_apply_window
4.88	15.39	0.99	20080	0.05	0.05	vorbis_synthesis_blockin
4.59	16.32	0.93	9277919	0.00	0.00	oggpack_look
3.26	16.98	0.66	40160	0.02	0.02	floor1_inverse2
3.01	17.59	0.61	5787	0.11	0.11	_libc_write
2.81	18.16	0.57	20080	0.03	0.69	mapping0_inverse

Time : the percentage of the total running time of the program used by this function.

cumulative seconds: a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds: the number of seconds accounted for by this function alone. This is the major sort for this listing.

Calls: the number of times this function was invoked, if this function is profiled, else blank.

self ms/call : the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

Total ms/call : the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

Name : the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

In summary, the profile information shows the high computation intensive parts of Vorbis decoder are:

(1) *Inverse MDCT* transform (*mdct_backward()* and its subroutines) with 30.5% of total computation time.

(2) Residue and codebook operation (*res2_inverse()* and its subroutines) with 23.2% of total computation time.

Fortunately, the C code of the most demanding part, the "*Inverse MDCT* transform" process is highly independent and repeated. In the following discussion, we will focus on some software techniques to improve its performance. Hopefully, these techniques can facilitate the execution of real-time Ogg Vorbis applications.

[Methods]

Ogg Vorbis is still an on-going project. One drawback of such hemi-complete software is that without software modification, it can be only executed on specific hardware/compiler:

(1) OGG encode/decode algorithms are machine dependent (i.e. based on the different data types(int, short, long, etc.) and floating operations each machine supports, different OGG algorithms will be compiled for that machine. Currently, OGG Vorbis only supports Alpha, HP-PA, x86, MIPS, POWERPC, SPARC, IA-64 and x86_64 CPUs.

(2) Some Ogg decoders are multi-threaded feather-rich audio players. These Ogg decodes utilize the pthread library, which limits them can only be compiled by latest version of gcc.

We have tried hard to port the Ogg decoder to available microprocessor simulators to evaluate its performance, including (1) ASIP Meister [3], which is an application specific DLX based microprocessor developing system developed by University of Osaka, Japen. (2) Simplescalar tool set [4], a suite of microprocessor instruction set simulation tool developed by Todd Austin, and (3) Tensilica Xtensa T1050.0 development tools [5].

Nevertheless, we found the academic projects (ASIP Meister and SimpleScalar) lack gcc update and they failed to compile the Ogg decoder. On the other hand, although we successfully built the Ogg decoder on Tensilica T1050.0 platform, our student license confined us to run further instruction set simulator. In the following sections, we suggest several methods to improve the Ogg performance so that it can be applied on other embedded processors; however, only roughly estimation is provided.

Methods 1. Advanced Multiplication

In the encoding process, each element of the MDCT coefficients will be quantized to a value with smaller range by the product of quant-scale and quant-table; therefore, in the decoding process this operation must be reversed to reconstruct the original data.

That is, each MDCT element will be multiplied by the product of quant-scale and quant-table. Owing to the infrequent change of the quant-table, the software developer can pre-compute and store this multiplication result. This process should save one multiplication when decoding each inverse MDCT element.

Methods 2 : Parallel Inverse MDCT

The greatest multimedia handling ability is achieved by utilizing the Single Instruction Multiple Data (SIMD) stream operation type. Most recent PC microprocessors have a SIMD instruction set which i s designed to accelerate the execution of multi-media applications. There are well known techniques to accelerate DCT based algorithm using multiply and accumulate instructions. SIMD instructions are known to be effective for accelerating such transform o perations. The basic idea is to find the parallelism in one transform operation and use SIMD instructions to execute multiple operations in one transform by one instruction.

For example, in the quantization process, the calculation of $M^{3/4}$ is executed repeatedly. In general, this calculation is performed as followed:

$$\mathrm{sqrt}(\mathrm{sqrt}(M) \times \mathrm{sqrt}(M)).$$

By using SIMD instructions, this calculation is done in parallel and this operation has potential to be accelerated 4X. However, it is worth to mention that the SIMD instruction sets have been criticized that 16 bits data precision could not fit the necessity of most high dimensional audio data and therefore, SIMD instruction sets are not always usable and their tolerance for computation error depends on the precision requirements of the application. In reality, programmers should be careful to compromise the precision according to the requirements. The inclusion of the new SIMD instruction sets promises more powerful computation ability for future multimedia applications.

Hardware Notion

The hardware architecture of our design is based on Tensilica's proprietary Xtensa hardware engine. We wish to test the viability of configuring a system design solely based on the tools provided by their company, and the abi lity of these tools to maximize the performance of our design.

Tensilica has provided the tools to put together a microprocessing environment consisting of their Xtensa core and combined with the optimizations and configurations desired by the designer. In their words Tensilica's mission is to provide configurable microprocessor cores plus a complete and totally integrated suite of software development tools for system-on-chip (SOC) applications for high-volume markets. Tensilica solutions allow design ers to create lower power, foundry-independent hardware and software specific to their applications. Tensilica technology leverages proven ASIC design methods, programming models and architectural standards to reduce risk and development time [5].

The Xtensa line used in our design is Tensilica's flagship processor providing all the devices one is accustomed to and expects from an advanced and purportedly adequate microprocessor, along with the extra inclusions that make it unique. Xtensa is a unique processor architecture designed exclusively for high volume, embedded applications. Designers can use the Xtensa Processor Generator – accessed via secure internet connection or on-site customer installation – to scale, configure and extend a powerful family of core processors and add memories, closely-coupled peripherals and special functions while doing concurrent, optimized, software development and testing. A complete GNU-based software development environment is created with each new processor configuratio n. The precision, flexibility and speed of implementation of Xtensa-based solutions enable substantial performance improvements, rapid migration to new silicon technologies, quick integration of new features, and significant cost reductions [5].

The true power and breadth of this product is that by using Xtensa technology, the system designer can mold the processor to fit the application by selecting and configuring predefined elements of the architecture and by inventing completely new instructions and hardware execution units that can deliver performance levels orders of magnitude faster than alternative solutions. These hardware extensions are implemented using the Tensilica Instruction Extension (TIE) language. By means of this language the designer can describe new instructions and registers and even execution units that are then automatically added to the predefined and configured Xtensa processor. TIE is a not unlike the widely known hardware definition language Verilog. It is used to describe i nstruction mnemonics, operands, encoding, and semantics. TIE files are inputs to the Xtensa Processor Generator (one of the software tools provided by them), which automatically builds a version of the Xtensa processor and the complete tool chain that inc orporates the newly written TIE instructions.

All of these tools are nicely integrated together in Tensilica's web-based Xtensa processor generator interface to select the instruction set options, memory hierarchy, and other building blocks and external i nterfaces required by the designer. The process generator then produces both the complete synthesizable hardware design and the tailored software environment, which consists of a profiler, various simulations models, and even overlays for some supported RTOSes to perform the required software development, system-level simulation, and tuning.

We based our choice on the fact that the Ogg encoding/decoding suite of tools which we wished to profile is a viable alternative to the very popular MP3 format. However, as such an option it will need to be implemented in numerous embedded applications as its main rival has, and this is a process which cannot take years for fruition, as any delay will only see the continued dwindling of the available market share in this niche. Hence the choice was made based on the various claims made by the Tensilica Corporation.

Hardware Architecture

The core of the Xtensa processor is built around a 32 -bit architecture featuring a compact instruction set (one unique to it) opt imized for embedded designs. This base 32-bit architecture has a 32 -bit ALU, up to 64 general purpose registers, six special purpose registers, and 80 base instructions including 16 - and 24-bit instruction encoding. The rest of the details which are up t o the designer were configured as follows:

Configuration Details

Target options	
Frequency	281 MHz
Functional Clock Gating	yes
Global Clock Gating	yes
Register File implementation	Latches
Instruction options	
16-bit MAC with 40 bit Accumulator	yes
16-bit Multiplier	yes
Boolean Registers	no
Coprocessor Count	0
Enable Density Instructions	yes
Zero-overhead loop instructions	yes
Interrupts enabled?	no
High Priority Interrupts	no
Timers	no
Byte Ordering	Little Endian

Address registers available for call windows	32
Misc. special register count	0
Instruction Cache (Bytes) / Line size (Bytes)	1KB / 16
Associativity	Direct
Line Locking	no
Data Cache (Bytes) / Line size (Bytes)	4KB / 16
Associativity	Direct
Write Back	no
Line Locking	no
Debug	no
Xtensa Exception Architecture	XEA2
Memory Protection/MMU	Region Protection
System RAM start address / size	0x60000000 / 1MB
System ROM start address / size	0x40000000 / 128KB
Local Instruction RAM start address / size	0x5fff8000 / 1KB
Local Instruction ROM start address / size	Not selected
Local Instruction RAM/ROM busy signal	yes
Local Data RAM start address / size	0x5fff0000 / 32KB
Local Data RAM Inbound PIF request	no
Local Data ROM start address / size	0x5ffef800 / 2KB
User (Program) Exception Vector start address / size	0x60000220 / 0x1c
Kernel (Stacked) Exception Vector start address / size	0x60000200 / 0x1c
Window Register Overflow Vector start address / size	0x60000000 / 0x180
Reset Vector start address / size	0x40000020 / 0x2e0
Double Exception Vector start address / size	0x60000260 / 0xe0
TIE source for configuration	-
TIE Xtensions / Coprocessors	0
CAD Options	

RTL description	Verilog
Simulation selections	VCS XL NC
Synthesis selections	Design Compiler
Place and Route selections	Apollo Silicon Ensemble

These architectural decisions were established on many facts. For instance profiling showed that integer multiplication consumed a large number of decoding cycles. Just including the Xtensa microprocessor's 16x16-bit hardware multiplier configuration option does away with going about in circles to perform this operation. Also the imdct operation consumes the vast majority of run-time, and since it performs many multiplications and additions on the same data it was only logical to include the 16-bit multiply and accumulate unit with its 40-bit accumulator.

Had we encountered more success in our efforts we were planning to utilize the TIE features provided to reduce execution cycles by combining multiple operations into one TIE instruction, or by operating on multiple data elements simultaneously. Where as the second approach would have provided the most promising results, because the audio decoding performs the same operations on large data blocks. Consider for instance a two-dimensional 8x8 iDCT block: using regular issue we would have to apply the algorithm sequentially to the columns and then to the rows. However, with the addition of certain TIE extensions we would be able to execute eight 1-D iDCT's in parallel. Another addition certain to speed up processing would have been to add a specialized read only memory to store all cosine and sine values used in the algorithms rather than performing the requisite calculations over and over again.

[Reference]
[1] The Ogg Vorbis Codec Project. http://www.xiph.org/ogg/vorbis/
[2] Luis Azuara and Oattara Kiatiseve (2002) "Design of an audio player as system - on-a-chip", Technical Report, University of Stuttgart.
[3] ASIP Meister homepage, http://www.eda-meister.org/asip-meister/
[4] SimpleScalar homepage, http://www.simplescalar.com
[5] Tensilica homepage, http://www.tensilica.com