ABSTRACT

In this paper, we describe the implementation of fixed point MAD mp3 decoder on the SandBlaster DSP [1]. We describe the compute intensive blocks in the MAD decoder, and how they can be optimized using Sandblaster tools.

1. INTRODUCTION

The MPEG-1 standard was developed to transmit audio and video data at 1.5 Mbps. The audio portion of the standard is divided into three layers (MPEG-1 Layer 1, MPEG-1 Layer II, MPEG-1 Layer III), each of which successively provides a better compression at the cost of higher computation for sampling rates of 44.1Khz. The MPEG-1 Layer III, which of interest here, is also called mp3. (The MPEG-1 standard was later extended by MPEG-2, which provided support for additional lower sampling frequencies).

The mp3 coder/decoder (codec) compresses/decompresses an incoming audio stream. In absence of the compression/decompression, a digital audio stream containing 16 bit samples, sampled at 44.1Mhz requires as high as 1.4 Mbps data rate for stereo music. By using an mp3 codec, one can compress the original sound data from a CD by a factor of 12, without losing sound quality. This is achieved by perceptual coding techniques addressing the perception of sound waves by the human ear. [2].

In this paper, we discuss the implementation of the MAD mp3 decoder on the Sandblaster platform. In the next two sections, we describe the MAD decoder and the Sandblaster platform. Then we discuss the major blocks and the source level changes made to the code.

2. MAD MP3 DECODER

The MAD mp3 decoder is a public domain decoder which is available in source form under GNU Public Licence [3]. It currently supports the MPEG-1 (Layers 1-III) and the MPEG-2 extension for lower sampling frequencies, as well as the de facto MPEG 2.5 format.

This is a fixed point decoder, and has the following main features:

- **24-bit PCM output**: MAD provides full 24-bit PCM output. Even when the output device supports only 16-bit PCM, applications can use the extra resolution to increase the audible dynamic range through the use of dithering or noise shaping.

- **Fixed-point (integer) computation**: As MAD uses integer computation rather than floating point, it is well suited for architectures without a floating point unit. All calculations are performed with a 32-bit fixed-point integer representation.

- **ISO/IEC standards compliant**: MAD is a new implementation of the ISO/IEC standards. The output satisfies the ISO/IEC 11172-4 computational accuracy requirements for compliance. In almost all configurations, MAD is a Full Layer III ISO/IEC 11172-3 audio decoder as defined by the standard.

The decoder has three major blocks as shown in Figure 1. The Decode Bitstream block reads the input bit stream containing frames of audio data. Each frame has a frame header that defines the data format. This block parses the frame header and synchronizes with a sequence of 12 1s in it. After synchronizing with this sequence, the decoder determines other relevant information such as sampling rate of uncompressed data.

![Figure 1: MP3 Decoder](image_url)
and data rate of the compressed data stream. The decoder then uses this information to decode the other frames.

In the Inverse Quantize/Huffman decode phase, tables are used to decode the variable length coded symbols. These symbols would have been generated by the encoder on the basis of frequency of occurrence of each symbol.

In the Synthesis/Filterbank stage, two steps are performed. In the first step, groups of 32 subband samples provided by initial decoding phase are converted to a 64 entry arrays using a discrete cosine transform (DCT) (There are 36 such groups, and each sample in a subband represents the amplitude for a particular frequency). At any point in time, the synthesis/filterbank phase keeps a set of 16 64 entry arrays in a rotating window fashion. In the second step, the 64 entry arrays are windowed using a set of 512 coefficients to produce 32 PCM samples. Thus, the 36 groups per channel in a frame produce a total of 1152 decoded audio samples.

3. SANDBLASTER DSP

Sandbridge Technologies has developed the Sandblaster architecture for a convergence device. As handsets are converging to multimedia multi-protocol systems, the Sandblaster architecture supports the data types necessary for convergence devices including RISC control code, DSP, and Java.

As shown in the Figure 2, the design includes a unique combination of modern techniques such as a SIMD Vector/DSP unit, a parallel reduction unit, and a RISC-based integer unit. Each processor core provides support for concurrent execution for up to eight threads of execution. All states may be saved from each individual thread and no special software support is required for interrupt processing. Instruction space is conserved through the use of compound instructions that are grouped into packets for execution.

The memory subsystem has been designed carefully to minimize power dissipation. The pipeline design in combination with the memory design ensures that all memories are single ported and yet the processor can sustain nearly 4 taps per cycle for a filter (the theoretical maximum) in every thread unit simultaneously. A RISC-based execution unit, depicted in the center of the figure, assists with control processing.

The processor supports many levels of parallelism. Thread-level parallelism is supported by providing hardware support for up to 8 independent programs to be simultaneously active on a single Sandblaster core. The data-level parallelism (SIMD) is supported through the use of a Vector unit. In addition, the compound word instruction set provides instruction level parallelism.

The processor is well suited to Video and Audio processing. Video & Audio (i.e. multimedia) codec processing often consists of control structures with compute-intensive inner loops. For the control code, a 16 entry, 32-bit register file per thread unit provides for very efficient control processing. Common integer data types are typically stored in the register file. This allows for branch bounds to be computed and addresses to be efficiently generated. Intensive loop processing is performed in the SIMD/Vector unit depicted on the right side of the figure. Each cycle, a 4x16-bit vector may be loaded into the register file while two vectors are being multiplied, saturated, reduced (e.g. summed), and saturated again. In the inner kernel of audio and video codecs such as filtering, dct etc, the computations appear as vector operations of moderate length.

The Sandblaster platform contains four Sandblaster DSP’s communicating with one another in a ring. In addition, an ARM926 is provided for the execution of protocol stack.

4. SANDBLASTER TOOLS

The Sandblaster platform also has a software tool chain to help the development communications & multimedia applications on the Sandblaster Platform. The software tool chain is primarily dedicated towards generating efficient code for this platform. The basic philosophy behind the tool chain is that the user should program in a higher-level language such as C, and be able to simulate the generated code. He/She should not have to write any assembly language code.

The tool chain primarily consists of an Integrated Devp. Environment (IDE), an ANSI C compiler, a functional simulator, and a real time operating system. More details can be found in [4].
5. MAD MP3 DECODER

The mad decoder has three major code blocks.

**Libmad**: This is the main component of the decoder. This decodes the input bit stream, performs the Huffman decode and generates the 24 bit audio.

**Libitag**: This processes the ID3 tags which determine the song name, song author etc.

**Madplay**: This is the wrapper code around libmad and libitag which provides the I/O functionality for the encoded and decoded streams.

The decoder uses a 32 bit fixed point format, which can be represented as $0xABB BBBB$ where A is the whole part (sign + 3 bits) while B is the fractional part (28 bits). Since the values are signed two's complement, the effective range is $0x80000000$ to $0x7ffffff$ (or -8.0 to +7.99). The smallest representable value is $0x00000001$ (3.725e-9).

The decoder implementation involved numerous multiplication operations. When two 32 bit numbers are multiplied, the 64 bit result can be scaled back to 32 bits by extra shifting and rounding. This converts the result back to 4 whole bits and 28 fractional bits.

The decoder already has some nice techniques to minimize computation. It performs Subband synthesis optimization (SSO), which eliminates the additional shift and add instructions. A 32X32->bit multiply of two values requires the result to be right-shifted 28 bits to be properly scaled to the same fixed-point format. Right shifts can be applied at any time to either operand or to the result - so the optimization involves careful placement of these shifts to minimize the loss of accuracy. First, a 14-bit shift is applied with rounding at compile-time to the table of coefficients for the sub band synthesis window. This only loses 2 bits of accuracy because the lower 12 bits are always zero. A second 12-bit shift occurs after the DCT calculation, which is the first stage of the sub band synthesis process. This loses 12 bits of accuracy. Finally, a third 2-bit shift occurs after the second stage of the sub band synthesis process (in routine synth_full) and just before the sample is saved in the PCM buffer. This a total of $14 + 12 + 2 = 28$ bits remain.

Another interesting routine is the dct [6]. In the current implementation, vectors for subband synthesis have a 1024-entry array made up of 16 individual 64-entry arrays. A single 64-entry array is generated from a complete set of 32 subband samples via something resembling a Discrete Cosine Transformation (DCT). It turns out that due to trigonometric symmetry, it is only necessary to calculate half of the 32->64 transformations. This relationship between the two halves is described in comments in the original public domain code.

$$x[i] = x'[i + 16] \quad i = 0..15$$
$$x[i + 17] = -x'[31 - i] \quad i = 0..15$$
$$x[i + 32] = -x'[16 - i] \quad i = 0..15$$
$$x[i + 48] = -x'[i] \quad i = 0..15$$
$$x[16] = 0$$

where $x[]$ is the 64-entry output array recognized in the standard, and $x'[i]$ is the output of a 32-point DCT of the 32-entry input array. The dct32 used is a based on Lee’s fast DCT algorithm.

Now, rather than expand the output of the 32-point DCT into an array of 64, it is more efficient to leave them alone and simply take into account the necessary changes in sign (seen above) later on in the calculations. Also, according to the standard, each computed 64-value vector is supposed to be shifted into the 1024-value block, shifting out the oldest 64-value vector from the block. A circular buffer is used for this in MAD rather than shuffling memory around. Also, it’s only half as big since its storing 32 values instead of 64. The 32 values are also split in half because of the way the PCM samples are constructed, into a lo half and a hi half.

6. PORTING & OPTIMIZING DECODER

The porting of the mad decoder written in fixed point C involved compiling the code, executing it on the simulator, collecting performance metrics to determine algorithmic optimization possibilities and repeating the process again. The process is shown in the figure 3.

For Sandblaster, we had to select the correct data types in the code, eliminate any GNU extensions, and use the proper implementation of certain macros (For most platforms, there is an assembly language implementation of these macros. However, for Sandbridge, only the C implementation is used) Then the code was compiled using the highest level of optimizations.

To execute the application, the Sandblaster simulator was used and a set of performance numbers were generated. using the profiler The profile information from the initial run indicates for a 44kHz sample and 256kbps input stream (The Decoration of Christmas Tree by Tchaikovsky), the computation is spread across a handful of functions, of which dct32, which is first step of the subband synthesis required 16% of the processing, while the synth_full (second step of subband synthesis) required 43% of the processing.
The Subband synthesis process has been described earlier. To compute the PCM samples, the sub band synthesis process requires input from the dct32() function and a set of windowing coefficients. The standard defines 512 windowing coefficients. Due to the way values are shifted into the block, each windowing pass covers alternating halves of each array in the block.

Now from the earlier section, it is clear that a 14 bit shift is applied to the table of coefficients at compile time. This reduces the useful information to 18 bits in the coefficients. However, when the coefficients are used in synth_full() for multiplication with the values provided by the dct32(), it requires a 32X32 multiplication. This is a very expensive operation on the Sandblaster DSP, as a 32X32 multiply is generated using a sequence of 16X16 multiplications.

A closer look at the coefficients indicates that the upper three bits of the coefficient are always zero. So, one can drop the top two bits, and save the remaining contents in a 16 bit number (short) without loss of precision. Then, when the 16 bit coefficients are multiplied with a 32 bit operand supplied by the dct32(), it is a 16X32 multiplication rather than a 32X32 multiplication, which is cheaper.

In addition to above, the original code for the synth_full is written as a straight-line code. The code takes a set of the DCT output, and uses a set of coefficients to generate the subband samples. Here is a code snippet:

```c
MLA(hi, lo, (*fe)[0], ptr[0]);
MLA(hi, lo, (*fe)[1], ptr[14]);
MLA(hi, lo, (*fe)[2], ptr[12]);
MLA(hi, lo, (*fe)[3], ptr[10]);
MLA(hi, lo, (*fe)[4], ptr[8]);
MLA(hi, lo, (*fe)[5], ptr[6]);
MLA(hi, lo, (*fe)[6], ptr[4]);
MLA(hi, lo, (*fe)[7], ptr[2]);
```

In the above code, the macro MLA multiplies the DCT output (fe[]) with the coefficients (ptr()). However, the generated code is inefficient. One can simplify the address computation and make the code more readable if it is written in the form of a loop. To do so, one needs to change the contents of the coefficient table, and then convert the code into a loop form.

```c
temp = ptr[16];
ptr[16] = ptr[0];
for (i = 0; i < 8; i++)
    MLA(hi, lo, (*fe)[7 - i], ptr[(i+1)<<1]);
ptr[16] = temp;
```

This indicates that the synth_full & dct32 functions would be good candidates for optimization. To improve the performance, we follow an iterative process described earlier.
Note that we have replaced the contents of ptr[16] temporarily with the contents of ptr[0], which enables the code to be written in a loop form.

Similar optimizations are applied throughout the function.

We have explained the scaling process in the dct32() earlier. This can be optimized further, so that the input provided to the synth_full() function is a 16 bit input rather than a 32 bit input. As mentioned earlier, a twelve bit shift is applied at the end of dct computation. This produces a 32 bit quantity of which only 20 bits are useful. However, one can apply a 16 bit shift instead of a 12 bit shift. It does not cause any significant loss of precision in our test cases. This shift generates a 16 bit quantity, which in turn can be used in the synth_full() function to generate a 16X16 multiply. A side effect of this is that the multiplications get vectorized, improving the performance further. Also, at the end of the synth_full(), a shift left by two is generated instead of a shift right by two as in the original code.

In addition to above, changes were made to the imdct() function to increase the amount of compile time computations.

The optimizations described above improved the performance by 32 percent. The distribution of computation is shown in Figure 5.

In addition to the changes described here, opportunities to optimize the source code further exist.

7. SUMMARY

In this paper, we have described the implementation of MP3 decoder on the Sandblaster processor. The original source code has been changed to facilitate faster arithmetic operations.

8. REFERENCES