Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at dsp.support@analog.com and at dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

Porting PC-Based MP3 Player Software to ADSP-21262 SHARC® Processors

Contributed by Srinivas K. and Kunal Singh

Rev 1 – November 16, 2004

Introduction

ADSP-21262 devices are a members of the third generation of SHARC® family of processors. ADSP-21262 processors offer SIMD architecture and are equipped with powerful DMA engines,ensuring high bandwidth data transfers to and from the processor. Data transfers are completely transparent to the processor core. ADSP-21262 processors operate up to 200 MIPS and provide several peripherals (e.g., SPORTs, PP, SPI, IDPs) that are well suited for audio applications.

MP3 is a standard for digitally compressed music. This compression algorithm is capable of up to 10:1 compression with no noticeable loss in quality of the audio data. MP3 (short for MPEG3) stands for Motion Picture Experts Group, Audio Layer 3. MP3 is becoming an increasingly popular way to store audio in electronic format. An MP3 decoder reads the compressed data from the storage media and performs various decoding steps to obtain the raw audio data. This audio data is in PCM audio format, which can be stored on storage media or played to an audio output device (speaker) in real time.

This application note is based upon experience gained while porting pure PC-based C code for an MP3-decoder to ADSP-21262 processors using the VisualDSP++® 3.5 tools suite. The target platform was the ADSP-21262 EZ-KIT Lite® evaluation system. This application note summarizes key considerations involved in porting general PC-based C-code to ADSP-21262 processors.

Data I/O - PC versus SHARC

As depicted in Figure 1, general PC-based code primarily uses file I/O for data input and output operation. The data may be stored in the form of the files on the PC's hard drive. The file I/Os on PC are supported by the OS running on the PC. For example, MP3 files for an MP3 decoder may be stored on the PC's hard drive.

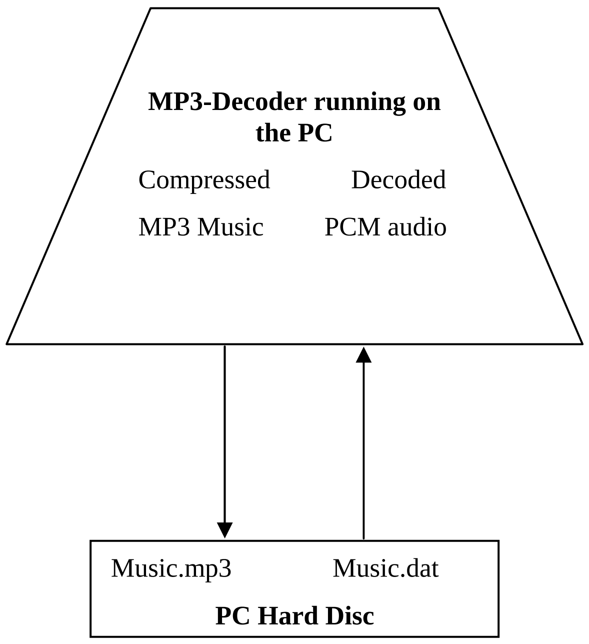

Figure 1. Data I/O Scheme for a PC-based System

Unlike a PC environment, the data on the embedded processor would be available from an external device (e.g., memory or a Host device). The data from the external device would be transferred in and out of the processor through its peripheral. Figure 2 depicts the data I/O scheme for an MP3 decoder ported onto an ADSP-21262 processor.

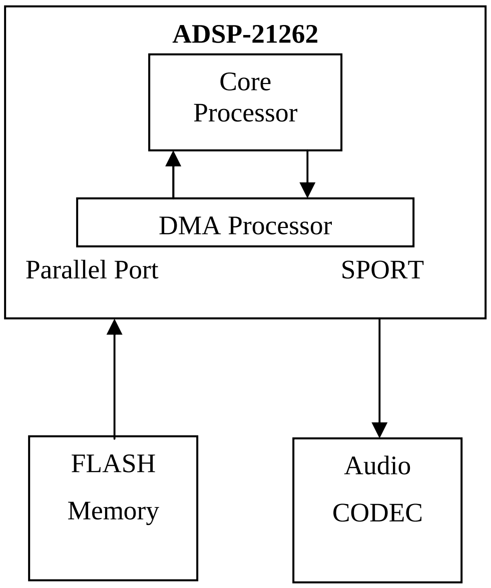

Figure 2. Data I/O Scheme for the SHARC-based MP3 Decoder

The first task in porting the PC-based code to the embedded platform, is to replace the file I/Os in the PC code with the peripheral-based I/Os on the SHARC processor.

Code Profiling

The next step is to obtain an estimate of the MIPS consumption. The optimization process can be an iterative procedure where MIPS for the different functions would be measured, changes would be made to the code structure, and the effect on the MIPS utilization would be evaluated.

The first step in optimization is code profiling. The entire code is split into a set of smaller modules for analysis. The benchmarks (in terms of MIPS consumed) for these different modules can be obtained, and the information can be stored in an Excel spreadsheet.

Using the above profile, identify the functions that consume the most MIPS. Devote your efforts toward optimizing these functions. Don't bother with the functions that require fewer MIPS.

The following paragraphs summarize different techniques that may be used to optimize the different code modules.

Table 1 shows the instruction count for various functions optimizing the MP3 code.

Function	Cycle Count
Huffman Decode	82327
De-quantize Sample	239079
Anti_alias	4292
Inverse MDCT	52770
Hybrid Synthesis	1201638
Sub-band Synthesis	186984

Table 1. Instruction Count for Various Functions Measured Before Optimization

Using DMA Engines

The data I/O operations through the peripherals can be performed in core mode or in DMA mode. For core-mode data transfers, the processor must execute a read/write instruction to an address to which the particular peripheral has been mapped. These transfers involve one instruction cycle for ever data transfer.

For DMA-mode data transfers, the SHARC processor's I/O handles all of the data transfers. The core processor needs only to initialize the DMA control/parameter registers with appropriate values, which may involve only a few instructions cycles. Thus, while using the DMA based transfers, the processor core is relieved of the instruction penalties that would have occurred with core-mode transfers. The

DMA scheme is particularly suitable for real-time applications in which huge amounts of data must be moved in and out of the processor in real time.

ADSP-21262 processors offer powerful DMA engines to perform data transfers across:

- Internal and external memory
- Internal memory and an external peripheral

The above data transfers are completely transparent to the core.

Parallel Data Fetch and SIMD

ADSP-21262 processors offer dual data fetches and a MAC operation in a single cycle. The internal bus architecture of the ADSP-21262 processor consists of separate PM and DM buses. In normal scenarios, the PM bus fetches instructions from Program Memory and the DM bus reads/writes data from Data Memory. While executing computation instructions with dual data fetch, one operand is fetched on the PM bus and the second operand is fetched on the DM bus. Having the executed instructions available in the Instruction Cache (so instruction fetches are not needed and the PM bus is free to access data) is a prerequisite for the above operation to complete in a single cycle.

Instructions involving dual operands are encountered frequently in typical signal-processing code. Some examples include FIR/IIR filter loops, DCT, FFT, and other transforms.

The above routines involve MAC operations on two vectors. The operations are performed in a loop (so all instructions are moved to Instruction Cache during the first execution of the loop, and no instruction fetches are required for subsequent loop iterations). If the two data vectors are located in different memory blocks (PM and DM), it may be possible to use a dual fetch in a single cycle.

Another important feature of the ADSP-21262 processor is its SIMD architecture. The ADSP-21262 has two parallel compute units which can execute same instructions on different data sets in parallel.

Consider the following multiplication loop:

```
float operand1[1024];
float operand2[1024];
float result;
{
  int j;
  result = 0;
  for (j= 0; j<1024; j++)
  {
     result += operand1[j] *
operand2[j];
  }
}
```

Listing 1. Multiplication Loop Without Optimization

In the absence of a dual data fetch, the inner multiplication loop in the above example would require 2048 cycles to finish the execution. This is because the fetching of operand1 and operand2 for each instruction requires a total of two cycles.

The above code structure can be modified such that one of the operands lies in the PM block. With the above modification, the two operands can be fetched in a single cycle. Since the multiplication is being performed within a loop, the instruction would get cached after the first execution, so that processor can fetch the two operands in a single cycle.

```
float PM operand1[1024];
// the "PM" command would place
operand1
// in PM
float operand2[1024];
float result;
{
  int j;
  result = 0;
  for (j= 0; j<1024; j++)
  {
   result += operand1[j] * operand2[j];
  }
}
```

Listing 2. Multiplication Loop with Dual Data Fetch

As shown above, the `PM` command instructs the compiler that this particular variable must be stored in the PM block. The above loop would take approximately 1024 cycles to execute. The code can be structured further, allowing the compiler to use SIMD mode.

```
float PM operand1[1024];
// the "PM" command would place
operand1
// in PM
float operand2[1024];
float result1, result2;
{
  int j;
  result1 = 0;
  result1 = 0;
  for (j= 0; j<512; j++)
  {
    result1+= operand1[j] *
operand2[j];
    result2+=
operand1[j+1]*operand2[j+1];
  }
  result = result1 +result2;
}
```

Listing 3:

Listing 3. Multiplication Loop with Dual Data Fetch and SIMD

With the multiplication loop re-structured in the above fashion, the compiler would enable SIMD mode and execute the instructions for result1 and result2 on different processing elements. The above loop would take approximately 512 cycles to execute.

Native Instructions

Instructions in the processor's instruction set can be executed in a single cycle. However, operations that are not native to the instruction set take multiple cycles.

Some complex operations can be performed in alternative ways that rely only on the native instructions to perform the operation. For example, signal-processing code frequently involves division by a factor of 2/4/8 and so on, which take approximately 40 cycles. However, these divisions can be replaced by right-shift operations which would be performed in a single cycle.

Function Calls

Another important consideration is function calls. The C run-time manager must save/restore the context information across the function calls. The context information is pushed onto the stack while calling a new function and is popped from the stack when returning from the function call. If frequent function calls are made to a relatively smaller function, large overheads are required. These overheads can be eliminated by replacing such function calls with inline code.

The VisualDSP++ 3.5 compiler also provides built-in versions of some C library functions. The compiler immediately recognizes them and replaces them with inline assembly code instead of a function call. Inline assembly code is faster than an average library routine, and it does not incur the calling overhead.

Processor Built-In Functions

The VisualDSP++ compiler supports intrinsic (built-in) functions that enable efficient use of hardware resources. These functions are different from the built-in library functions which we discussed above, in which the function call is replaced by inline assembly. Rather, the processor built-in functions provide a means to use the processor's hardware efficiently.

The built-in functions can be used to:

(a) Access the System registers: Some intrinsic functions provide efficient access to registers, modes, and addresses not normally accessible from C source. This can be achieved through a set of functions defined in the “`sysreg.h`” file. Examples include reading/writing System registers and setting/clearing particular System register bits.

(b) Instruct the compiler to use circular buffer indexing. This is important for access to a data array with a fixed offset between two accesses. Decimation, filtering, and FFTs are examples of algorithms that may utilize the above function. Consider the following example:

```
int m, jj;
float sum, COS[SIZE];
#define circindex __builtin_circindex
  for (m=0;m<N ; m++)
  {
     sum += COS [jj];
     jj = circindex (jj, MODIFY, SIZE);
  }
```

Listing 4. circindex Function for Circular Buffering

In the above example, the `COS` array is accessed inside a loop. The `circindex` function instructs the compiler to access `COS` using circular buffering with Mx = modify and Lx = length. However, if `circindex` is not specified in the above example, the compiler may not implement the accesses to `COS` with index registers. Instead, it may use other calculations to calculate the index for each access, which would consume extra cycles.

Other Optimizations

C code that performs satisfactorily on a PC may not be MIPS efficient on a processor. The MIPS on the processor are constrained generally and may require further optimizations specific to the algorithms being used. For example, DCT computations may be replaced by fast DCT algorithms.

As discussed already, great benefits may be achieved by using processor native instructions in place of complex computations. We would like to share the following example:

```
N=36;
for( p= 0;p<N; p++)
{
   sum = 0.0;
   for(m=0;m<N/2;m++)
   sum += in [m] *

COS[((2*p+1+N/2)*(2*m+1))%(4*36)];
   out [p] = sum * win [block_type]
 [p];
}
```

Listing 5. An IDCT Loop

The above code section (taken from the MP3 algorithm) is used for the inverse DCT computation.

The instruction in the innermost `FOR` loop uses complex logic to calculate the index of a `COS` table. In the particular algorithm, the `FOR` loop instruction would execute 41472 (2x32x36x18) times to process each audio frame. Using the above code on ADSP-21262 processor in place the algorithm takes 330 MIPS.

We have tried to use simple logic to index the `COS` table in the above example. A “%” is not a native operation for the processor. It would be performed using a certain library function (C-library) that would introduce additional overheads (due to the function call) each time the index in the above code is calculated.

We tried to implement the index calculation logic using the additions and comparisons. These operations can be performed easily on the processor. We can replace the earlier code with:

```
int start = 19 ;
int modify = 38 ;
N=36;
for (p= 0;p<N ;p++)
{
  int jj = start ;
  double temp1, temp2 ;
  sum = 0.0;
  for(m=0;m<N/2;m++)
  {
     sum += in [m] * COS [jj];
     jj = circindex (jj, modify, 144);
  }
  start += 2 ;
  modify += 4 ;
  if (modify > 144)
  {
    modify = modify - 144 ;
  }
   out [p] = sum * win [block type]
[p];
}
```

Listing 6. IDCT Loop with Optimizations

The above code section is functionally equivalent to the earlier code example. Implementing the above changes to the original code reduces the MIPS count for the algorithm dramatically from 330 to 110.

Summary

Key guidelines that permit efficient use of processor resources include: exploring DMA capabilities, using parallel data fetches (from PM and DM), exploiting the processor's SIMD architecture, and using native instructions to replace complex computations.

Table 2 shows the instruction count for various functions after the optimization and the performance improvement as a percentage of initial count.

The Figure 3 depicts the optimization results graphically.

Function	Instruction Count	Reduction in Cycle Count (%)
Huffman Decode	76655	6.8
De-quantize Sample	44822	81.3
Anti_alias	3117	27.4
Inverse MDCT	5196	90.1
Hybrid Synthesis	119348	90.1
Sub-band Synthesis	30345	83.8

Table 2. Final Instruction Count for Various Functions and % Reduction in Cycle Count

The Figure 3 depicts the optimization results graphically. The color bars represent various functions (similar to the color in Table 2) versus % reduction in the cycle count

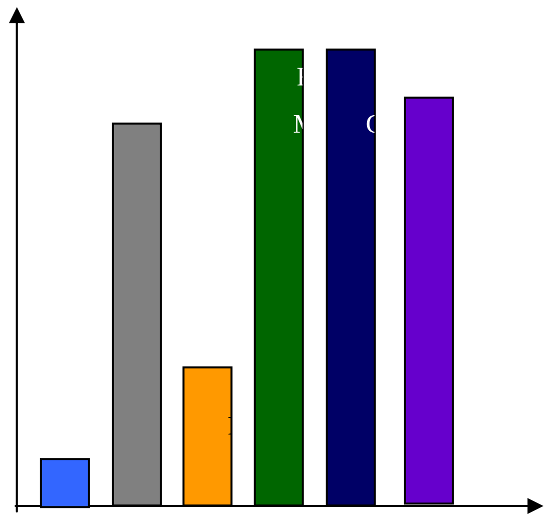

Figure 3. Graphical Representation of Optimization Results.

References

[1] *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for SHARC® Processors*. Revision 4.0, January 2003. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 1 – November 16, 2004 by Srinivas K. and Kunal Singh	Initial Release