Real-Time MP3 Decoding on FPGA: a Case Study of System Model Features

Xu Han and Gunar Schirner

Technical Report CECS-09-09
July 16, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hanx, hschirne}@uci.edu
http://www.cecs.uci.edu/

Real-Time MP3 Decoding on FPGA: a Case Study of System Model Features

Xu Han and Gunar Schirner

Technical Report CECS-09-09
July 16, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hanx, hschirne}@uci.edu
http://www.cecs.uci.edu

Abstract

System-level design methodology and supporting tools have been developed to address the complexity of embedded systems and to achieve shorter time-to-market. Modeling a complex system at higher level of abstraction has benefits of faster design space exploration and enables the path to automatic generation of low level models that connect with actual implementation. One trade-off of system models is related to the amount of captured implementation detail. Abstracting more details results in faster simulations, while preserving more details results in more accurate estimation. This work presents a case study of a manual implementation of a real-time MP3 decoder on FPGA, to identify system features that are important for modeling. Our experimental results show that cache configuration, interrupt overhead and compiler optimization options significantly influence system performance. Therefore, it is advantageous to include the timing effect of these features in a system model.

Contents

List of Figures

List of Tables

Real-Time MP3 Decoding on FPGA: a Case Study of System Model Features

Xu Han and Gunar Schirner

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{hanx, hschirne}@uci.edu
http://www.cecs.uci.edu

Abstract

System-level design methodology and supporting tools have been developed to address the complexity of embedded systems and to achieve shorter time-to-market. Modeling a complex system at higher level of abstraction has benefits of faster design space exploration and enables the path to automatic generation of low level models that connect with actual implementation. One trade-off of system models is related to the amount of captured implementation detail. Abstracting more details results in faster simulations, while preserving more details results in more accurate estimation. This work presents a case study of a manual implementation of a real-time MP3 decoder on FPGA, to identify system features that are important for modeling. Our experimental results show that cache configuration, interrupt overhead and compiler optimization options significantly influence system performance. Therefore, it is advantageous to include the timing effect of these features in a system model.

1 Introduction

Embedded systems are widely used today in various fields such as automobile industry, multimedia applications and medical applications. As complexity of embedded systems increases, traditional manually implementation's are no longer feasible. To cope with the problem, system-level design methodology and supporting tools are developed. Modeling systems with higher abstraction level enables fast design space exploration and hardware/software partitioning.

One essential aspect of system level design methodology is a sufficiently comprehensive model to capture necessary characteristics of the system at a high abstraction level. Timing accuracy is usually an important property we want to observe from these models, because timing accuracy is important to guarantee close relation between model and implementation. However, better estimated timing requires more detailed models, which result in slower simulations. This paper describes a case study of manually adapting an MP3 decoder reference source code for execution on an embedded system while meeting real-time constraints. The goal of this work is to identify some important model features that are necessary for expressive and accurate system simulations to accurately guide design space exploration.

MP3 decoder is a representative industrial multimedia application. MP3 is a popular audio compression scheme that can achieve 90% or more compression rate. A MP3 stream is composed of generally independent frames, each of which has its own header and audio information. It is computation consuming to decode MP3 stream during music play back. MP3 decoding poses real-time requirements for timely decoding of MP3 frames. By implementing such a decoder manually on FPGA, we are able to explore architectural and software alternatives of the design. From the experimental results, we identify desirable model features for more accurate modeling at an abstract level.

In following section, some work related with this study is introduced. Section 3 gives a brief description of MP3 decoder. Section 4 outlines the design platform used, the Altera DE2 board. Section 5 describes the main implementation work by adapting a decoder for desktop machines to embedded system. Section 6 discusses the design space exploration we performed to find suitable design choices. Finally, section 7 concludes this paper.

2 Related Work

Transaction-level modeling is emerging as a high-level modeling approach to deal with the rising complexity of embedded systems. System-level design methodology has been studied extensively via using the transaction level modeling language of SpecC or SystemC.

Chandraiah et al. [2] have used MP3 audio decoder as an example to show the benefits of automated SoC design based on SpecC methodology. They first developed a specification model in system-level design language of SpecC from source C code manually, then performed automated architecture and communication exploration with a supporting tool SoC Environment (SCE) [3], and finally the refined model can be synthesized into software binary that can run on an Instruction Set Simulator (ISS). Their work, however, did not take caches into account.

Yu et al. [8] have developed a tool for automatic generation of TLM, using MP3 decoder as an example. With custom communication platforms and the decoder source code in C, they have generated SystemC TLM, which is ready for simulation. Their TLM includes communication models of bus and bridge.

Papakonstantinou et al. [6] have implemented a floating-point MP3 decoder on FPGA. By implementing floating point accelerators as custom instructions in Nios2 processor, they have achieved real-time decoding of MP3 data. Their work focused on accelerator designs, while our work focuses on overall system performance.

3 Design Drive: MP3 Decoder

MPEG-1 Audio Layer 3 (MP3) is a very efficient audio compression scheme. It can achieve more than 90% compression while keeping high quality of sound. Due to this reason, MP3 format is widely used in our daily life and a MP3 decoder can be a representative industrial multimedia application.

A MP3 decoder takes MP3 stream as input and generates PCM samples as output. A MP3 stream is composed of frames. Each frame contains 1152 encoded PCM samples. A frame consists of four main parts: header, side information, main data and ancillary data (Figure 1). The header begins with 12-bit sync-word indicating the starting point of each frame, and it also contains information about the layer, bitrate, sampling frequency and stereo mode. Side information contains necessary information to decode main data. Main data contains the coded scale factors and the Huffman coded frequency lines. Ancillary data contains some user-defined data such as identifying tag information. In our study, a sampling rate of 48 KHz is used. Thus, in order to decoding in real-time, each frame must be decoded in 24 ms (1152 samples per frame divided by 48 KHz).

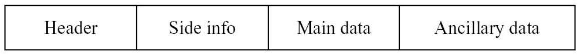

Figure 1: MP3 frame format [5]

Figure 2 shows the decoder data flow. The incoming data stream is read as individual frames (indicated by sync-word) and the correctness of each frame is checked. After Huffman decoding, requantization and reordering, the encoded audio samples are fed to the stereo decoder, which supports both MS stereo and intensity stereo formats. The alias reduction block then reduces the unavoidable aliasing effects of the encoding polyphase filter bank. Next, the IMDCT block converts the frequency domain samples to frequency subband samples. Finally, the polyphase filter bank transforms samples from different frequency subbands into PCM samples, which is ready to be played back through an audio codec.

4 Design Environment

FPGAs are a very popular means for computing and prototyping. They provide great design flexibility, fast turnaround time and simpler design flow. For these reasons, we choose FPGA as our design environment. Particularly, Altera DE2 board is used as our experiment platform. It hosts Cyclone II EP2C35 FPGA, SDRAM, SRAM, flash memory, SD memory card slot, 24-bit Audio codec (WM8731), VGA codec, LEDs and other components. Cyclone II is a low-cost FPGA, which has a capacity of 33,216 logical-elements and 105 M4K RAM blocks. The components we are using besides Cyclone II are: SDRAM and SRAM for storing data and instructions, audio codec for converting PCM samples to analog signals, and the SD card socket for input.

Nios II EDS tool set is used for the implementation. A general Nios II system design flow starts with defining and generating the system in SOPC Builder, where standard hardware components and a proper Nios II core are configured. Then the whole system is synthesized using Quartus2,

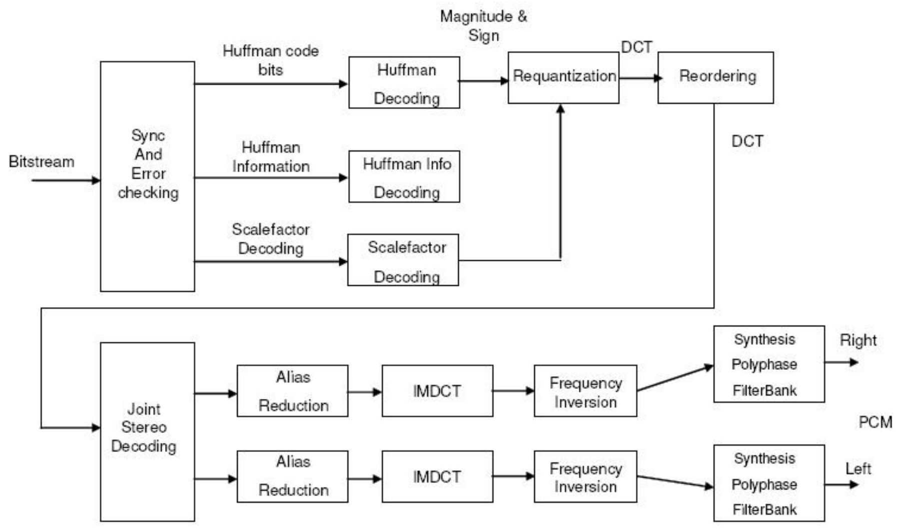

Figure 2: MP3 decoder block diagram [5]

which can produce two files. One is an SOPC system file (.ptf) which is needed by Nios2 IDE to compile any software to target hardware system, and the other is a SRAM Object File (.sof) which is the compiled hardware description of the system and ready to be downloaded to target board. With the generated system, we can develop our software application and run/debug on an ISS or directly on board. Often we may find the generated system does not fulfill all the requirements, such as memory capacity or execution speed. We can iteratively go back to modify or add components in SOPC builder and re-generate the hardware system.

It is not necessary to perform all the steps in our study. An existing system from DE2's demonstration package [1], including the SOPC system file and SRAM Object File, is used as a starting point.

4.1 System Architecture

Figure 3 presents the main components of the system. Nios2 processor version 'fast' working at 100 MHz is selected. It is tightly connected to an instruction cache and a data cache, both of which have configurable sizes. Nios2 communicates with the memories, SD card and audio_DAC_FIFO via Avalon bus interface. The main memory consists of 512KB SRAM and 8MB SDRAM. Input MP3 stream is read from SD card. The audio_DAC_FIFO, an IP component, is an audio DAC controller with 256$\times$16 FIFO memory. The audio DAC controller provides clock generation and dataflow control for the audio codec via serial I2C bus interface. The audio codec receives audio samples from the FIFO and send them to its line out port to play the music.

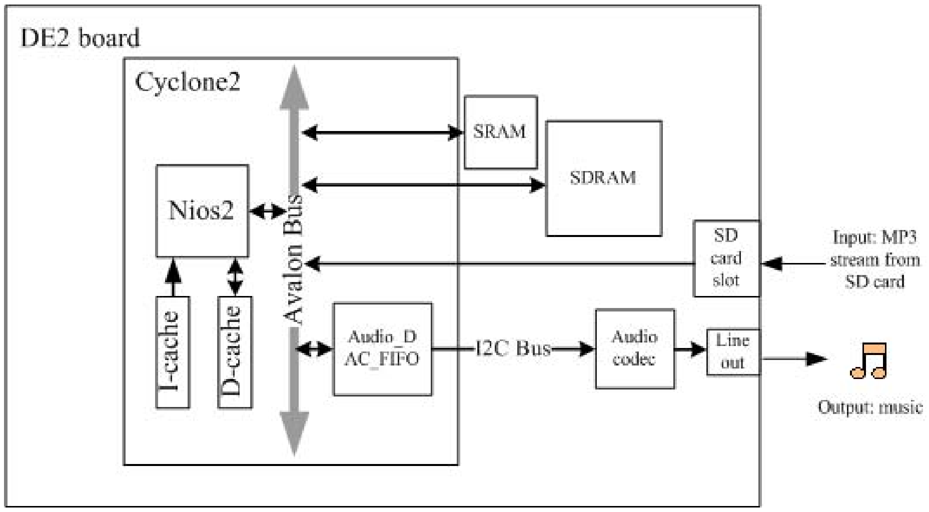

Figure 3: System Architecture

5 Reference Code Modifications For Embedded Constraints

Our implementation is based on Underbit MPEG audio decoder library [7]. The source code is an implementation based on the ISO/IEC standards. It only uses fixed-point computation and has an output of 16-bit stereo PCM. Since the reference source code is a desktop application, our main work is to design input and output routines for FPGA, and optimize for memory usage and execution speed.

Size constrained input: In the PC source code, an entire MP3 file is read into memory as input. However, unlike on a PC, embedded systems usually have very limited memory; therefore it is not safe to read an MP3 file with unbounded size in this case.

This point is illustrated again in Figure 4. The PC application reads the entire stream into a buffer. Later, it advances the frame pointer through the buffer while decoding. Such large buffers are not available in embedded systems. A software solution is to use a small input buffer which can hold a few MP3 frames and read from SD card only on demand. Since the API of reading the SD card only supports block-by-block read (512 bytes per block), it is not feasible to read the exact size of each frame. Instead, we read several blocks at a time. Once a frame header is decoded, the size of next frame is known. If the data left in the input buffer is less than the required frame size, we first move the not-yet-decoded data to the beginning of input buffer, and then read in new blocks and place them right after the shifted old data. In this way, we maintain a small window of a few frames and move it through large MP3 source stream.

Concurrent output: In the PC source code, the output of PCM samples is written to a file during decoding. In our case, the audio FIFO is too small to hold an entire decoded frame. Therefore, we need to introduce a circular software buffer between the decoder and the output FIFO. We use an interrupt in order to fill the output FIFO in parallel to decoding the data. However, the output FIFO implementation does not include an interrupt generation on an empty FIFO. It only provides

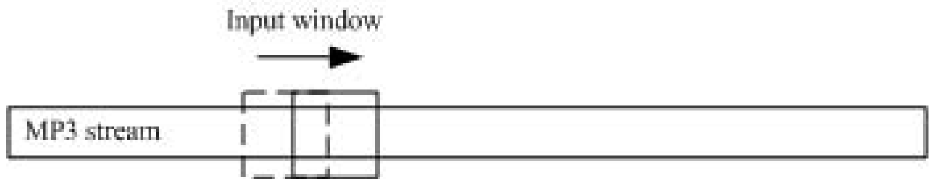

Figure 4: Illustration of input routine

a pollable flag indicating full FIFO.

Our alternative is to use a periodic timer interrupt to decouple the routine for decoding and that for output. It is feasible, because the output rate of audio codec is constant and we can know when the FIFO will be close to empty. In this way, we can have the concurrent music output in a periodical interrupt service routine. However, the next issue is selecting a proper timer period in order to play back music continuously. We can firstly find out how long in real-time the music samples can be hold in the FIFO. Knowing the FIFO has 16-bit data width, 256 in depth, duo-channels and uses a sampling rate of 48 KHz, a calculation can be:

Length of music the FIFO can hold
= FIFO size / number of channels / sampling rate
= 256 / 2 / 48000 = 2.66 ms

The result suggests if we use a timer interrupt for continuously playing back music, the upper bound of the timer interval is 2.66 ms. A smaller timer period is necessary to keep a safety margin avoiding that the FIFO runs empty due to the interrupt latency. Choosing this safety margin has an effect on system performance. With a too fast timer period the interrupt overhead will reduce performance. On the other hand, a too long period - a too small safety margin may cause an incorrect output if the FIFO runs empty due to too long interrupt latency. In our case, the application runs without an RTOS and avoids long critical sections that disable interrupt and thus would increase the latency. We therefore can use a very small safety margin of only 0.06 ms. An experiment on the effect of different timer period to system performance is provided in section 6.

5.1 Software Execution

Figure 5 presents the final software flow of the implementation. Program starts with initializing timer interrupt, SD card and decoder datastructures. Decoding begins with reading and decoding the first MP3 frame header. It checks if the whole frame exists in the input buffer; if not, read more source data from SD card. Then each frame is decoded and the output of PCM data is stored in an output buffer. The output buffer is processed by the timer interrupt routine, which writes the PCM samples to the audio FIFO and releases the buffer space. In parallel, the FIFO forwards the data to audio codec which plays the music.

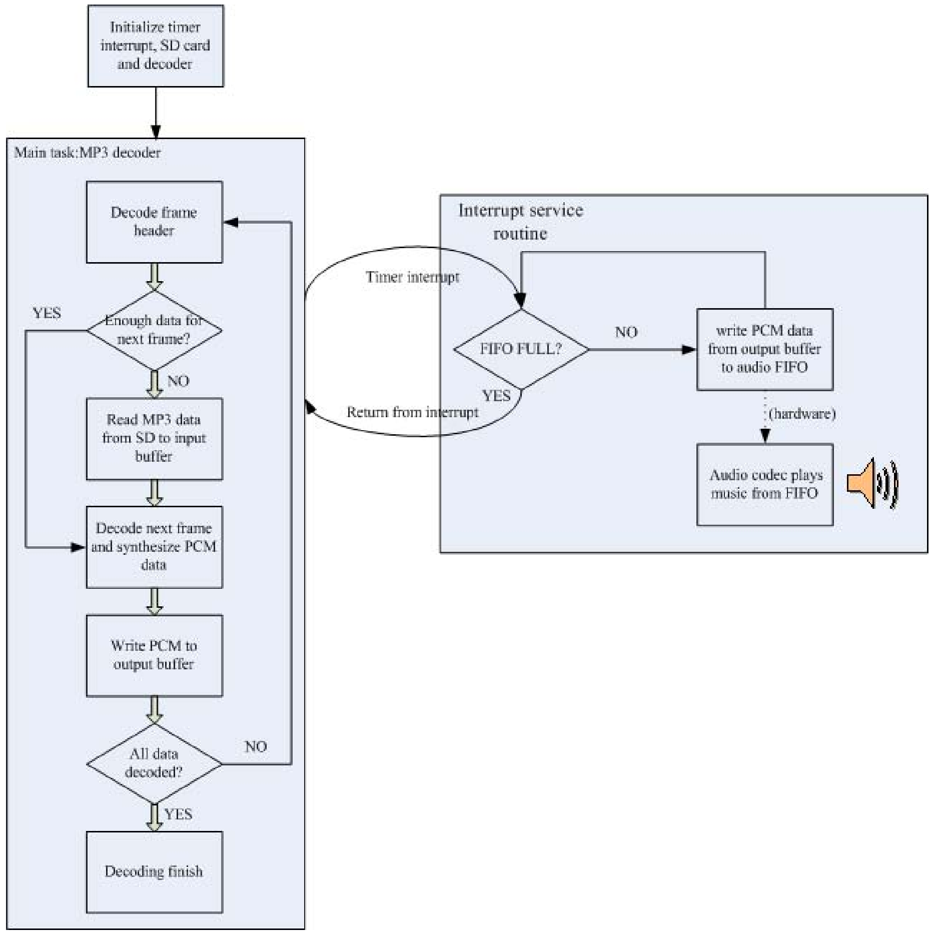
Initialize timer interrupt, SD card and decoder
Main task:MP3 decoder
Decode frame header
Enough data for next frame?
YES
NO
Read MP3 data from SD to input buffer
Decode next frame and synthesize PCM data
Write PCM to output buffer
All data decoded?
NO
YES
Decoding finish
Timer interrupt
Return from interrupt
Interrupt service routine
FIFO FULL?
NO
YES
write PCM data from output buffer to audio FIFO
(hardware)
Audio codec plays music from FIFO

Figure 5: Software Flow Chart

6 Design Space Exploration

In order to find suitable configurations for the system, design space explorations are necessary to be performed. The most important issue during the implementation is to achieve real-time decoding. We focus for our experiments on the single core architecture with one Nios processor, an output FIFO, and vary the memory configuration. A test based on the initial system configuration shows that it take 800% of real-time to decode a music sample (e.g. 8 sec decoding time for 1 sec of music) with 4KB I-cache, 2KB D-cache, SDRAM as main memory and no compiler optimization. To address this problem, we have used a faster memory, SRAM (which is off-chip and therefore slower to access than on-chip instruction cache) to store program text, and explored the parameter space of cache configurations, compiler optimizations and timer interrupt period.

Cache configurations: Nios2 has separated instruction and data cache. Both caches are direct-mapped because of its efficiency on FPGA. The I-cache block size is 32 Byte. Note that the as-

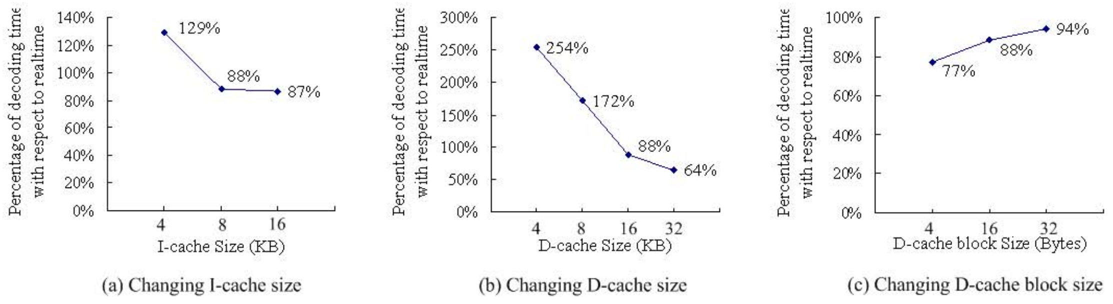

Figure 6: Decoding time as percentage of music length in real-time when each cache parameter changes

sociativity and I-cache block size are not configurable in the system. In addition, since Cyclone II is a relatively small FPGA with limited RAM blocks, the cache parameter space we can search is limited to a small scope.

Figure 6 shows the result of the cache parameter space exploration in order to fulfill real-time requirements. We use a set of baseline parameters of 8KB I-cache, 16KB D-cache and 16B D-cache block size, and then change one parameter at a time with other parameters constant to observe the performance.

First we change the I-cache size. Figure 6(a) shows that decoding is slower than real-time at 4KB (129% means it takes 1.29 sec to decode 1 sec of music). It has significant improvement when I-cache size reaches 8 KB, but stays at the same level at 16 KB. The result suggests 8KB is a proper size for I-cache because it holds the first working set of instructions. Next, we vary the D-cache size. Figure 6(b) shows that the decoding time steadily decreases when D-cache size increases. It suggests the working set of data is relatively large in our application, so that it can always benefit from larger cache size. Finally, we vary D-cache block size. As we know, larger block size will introduce less compulsory misses but more conflict misses. Figure 6(c) shows that the negative effect of larger block size is dominant in our case, and the performance is better when smaller block size is used with other parameters unchanged.

The best performance found is at the configuration of 8K I-cache, 32K D-cache, 16B D-cache block size, when decoding is done in 64% of real-time.

Compiler optimizations: Compiler optimizations have a great impact on software performance. As shown in Table 1, the decoding time decreases more than 40% by turning on optimizations. In addition, among all optimize options, -Os (which includes all options in -O2 and further optimizes for size) provides the best performance, and -O3 (which uses a more aggressive inlining alias over -O2) provides slightly worse performance than -O2. The explanation for the result is that, the code size is having more impact on system than some other optimize options. Given a relatively small instruction cache of 8KB, -O3, compared with -O2, inlines all functions, which increases code size and negatively impacts cache locality in our case and therefore shows worse performance. We see an indication for this by enabling *-finline-functions* over -O2, and decoding time will be slowed down to 5101.26 ms. Overall, we see that the compiler optimization dramatically influence system performance and therefore are important modeling aspect.

Optimize option	Decoding time
no optimization (-O0)	115 s
optimize (-O1)	59.2 s
optimize more (-O2)	47.9 s
optimize most (-O3)	50.9 s
optimize size (-Os)	47.3 s

Table 1: Decoding time with different compiler optimizations. Test with input of 128 KB stereo MP3 data at 128 Kbps, i.e. 7.51 second of music in real-time, and with cache configuration of 8KB I-cache, 32KB D-cache, 16B D-cache block size

Timer period: Previously, we have calculated an upper-bound for timer period in order to have concurrent output. Since there is latency between releasing the timer interrupt and the actual execution of the interrupt handler (i.e. interrupt latency), a shorter timer period is needed to include a safety margin and avoid that the FIFO runs empty. On the other hand, shorter timer period slows down the computation due to more frequent interrupts of the CPU (interrupt overhead). To study its effect, system performance is tested with different timer period. However, as Table 2 shows, the influence of different timer period is minor, compared with that of cache configurations and compiler optimizations. Decoding takes from 4.77 seconds with a 2.6 ms timer to 4.93 seconds with a 1 ms timer. We attribute the relatively low overhead to the fact that our system does not execute an RTOS and therefore avoids scheduling overhead.

Timer interrupt period	Decoding time
2.6 ms	4.77 s
2.3 ms	4.79 s
2.0 ms	4.81 s
1.0 ms	4.93 s

Table 2: Decoding time with different timer period settings. Test with input of 128 KB stereo MP3 data at 128 Kbps, i.e. 7.51 second of music in real-time, and with cache configuration of 8KB I-cache, 32KB D-cache, 16B D-cache block size

7 Conclusion

In this paper, we have presented a case study of a manual implementation of a real-time MP3 decoder. Our manual exploration process has shown that memory configuration, cache configuration, compiler optimization and interrupt overhead show significant influence to the performance of our software centric system. For a timing accurate model of a system, it is beneficial to include these features in an abstract model. Particularly, the memory hierarchy and the compiler optimization are of interest, since they had most performance impact.

References

[1] Altera de2 board demonstration package. ftp://ftp.altera.com/up/pub/de2/DE2_System_v1.2.zip.

[2] P. Chandraiah and R. Dömer. Specification and design of a mp3 audio decoder. Technical Report CECS-TR-05-04, Center for Embedded Computer Systems, University of California, Irvine, 2005.

[3] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. System-on-chip environment: a specc-based framework for heterogeneous mpsoc design. *EURASIP Journal on Embedded Systems*, 2008:1–13, 2008.

[4] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[5] K. Lagerstrom. Design and implementation of an mpeg-1 layer-3 audio decoder. Master's thesis, Chalmers Institute of Technology, May 2001.

[6] A. Papakonstantinou, Y. Kifle, G. Lucas, and D. Chen. Mp3 decoding on fpga: A case study for floating point acceleration. In *Proceedings of Reconfigurable Systems*, University of Illinois, July 2008.

[7] MAD MP3 decoder reference code. http://www.underbit.com/products/mad/.

[8] L. Yu and S. Abdi. Automatic systemc tlm generation for custom communication platforms. In *Proceedings of International Conference on Computer Design*, Lake Tahoe, USA, October 2007.