Spring 2008, Columbia University

CSEE 6847
DISTRIBUTED
EMBEDDED
SYSTEMS

MP3 DECODING ON AN FPGA

Shrivathsa Bhargav and Bingrui Yang

TABLE OF CONTENTS

Abstract 3

1. Introduction 3

2. Design 3

2.1 System Overview 3

2.2 Hardware components 4

2.2.1 Avalon bus 4

2.2.2 Nios-II processor 4

2.2.3 SDRAM and Flash memory 4

2.2.4 USB 4

2.2.5 Audio Wrapper 4

2.3 Software components 4

2.3.1 uClinux 4

2.3.2 mpg123 4

3. Implementation 4

3.1 Nios-II 4

3.2 uClinux 5

3.3 Audio Codec Controller 6

3.4 FIFO 6

3.5 Audio Wrapper 6

3.6 mpg123 6

3.7 Problems 7

3.8 Solutions 7

4. Results 7

5. Future Directions 8

6. Conclusion 8

7. Acknowledgments 8

References 8

About the authors 8

HARDWARE/SOFTWARE CO-DESIGN OF AN FPGA-BASED MP3 DECODER

[CSEE 6847 Final Project Report - May 2008]

Shrivathsa Bhargav (sb2784@columbia.edu) Bingrui Yang (by2138@columbia.edu)

ABSTRACT

This project describes the implementation of an open-source MP3 [6] decoder (mpg123) on an FPGA. The final implementation is capable of decoding audio streams with bitrates of up to 64 kbps and pipes this data onto an external speaker. The platform of choice is the Altera Cyclone II FPGA DE2 board [1] with a Nios-II "soft-core" processor running on it. A flavor of Linux is used as the operating system, on top of which a software decoder is deployed. Performance bottlenecks and possible solutions are explored.

1. INTRODUCTION

The Altera Quartus II development board houses a very-capable Cyclone II FPGA, along with a host of other peripherals such as USB ports and VGA controllers. In addition to supporting up to 35,000 logic gates, the FPGA is capable of running a microprocessor logic block known as Nios-II, which has its own IDE [2]. It is even possible to run a flavor of Linux tailored specifically for the Nios-II on this FPGA called "uClinux" [3,4].

While the FPGA is powerful, it is still limited in many ways. The default on-board clock is sourced by a 50 MHz oscillator, although a Phase-Locked Loop (PLL) is used to overclock the processor. Hardware floating-point support is expensive (area-wise), and the number of on-board memory blocks is woefully small, thereby making smooth audio playback rather difficult (there only exist around a hundred 4kbit memory blocks that can be used). Therefore, the final capabilities for this project were carefully planned. The maximum final sampling rate is 16,000 kHz, and the bitrate of the MP3 stream may not exceed 64 kbps. The source MP3 data is stored on an external USB drive and retrieved on-the-fly by the FPGA.

The motivation of this project was to show that a popular piece of consumer technology can be recreated using hardware that already exists in our labs. The MP3 format has withstood the test of time, having been invented nearly 20 years ago. Furthermore, FPGAs are meant to reduce time-to-market and ease fast-prototyping. Attempting to integrate these two technologies promised an interesting endeavor. We also wanted to understand the intricacies underlying hardware/software co-design. Here, hardware design refers to programming in a Hardware Description Language (the project combined both Verilog and VHDL), while the software represents coding in a higher-level language (C).

The system design is presented in section two, after which implementation details, hurdles, and solutions are presented in section three. Results are discussed in section four, while future directions are explored in section five. Section six concludes this paper.

2. DESIGN

The initial design for this project comprised of an MP3 decoder that combined C and VHDL, and was implemented on the Cyclone II hardware platform. In this initial design, VHDL addressed the computationally-intensive part of the decoding algorithm, while C handled the less-intense tasks on Nios-II. However, after further exploration of the device's capabilities and limitations, the initial design underwent some changes. The final design and the decisions that led to it are described in this section.

2.1 SYSTEM OVERVIEW

The system block diagram is shown in the following figure.

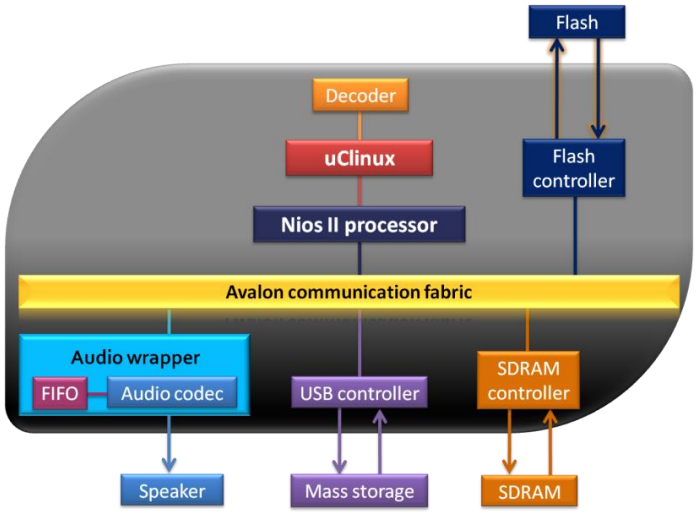

FIGURE 1 - SYSTEM BLOCK DIAGRAM

The gray background represents the FPGA; blocks inside this shape are modules on the device itself, while peripherals are shown outside. The main components of our design are

discussed below, while implementation details will be specified in section three.

2.2 Hardware components

The main hardware components are the Avalon bus, the Nios-II processor, memory and USB controllers, and the audio wrapper.

2.2.1 Avalon bus

The Avalon communication fabric [7], also known as the Avalon bus, is an essential component of the system. The bus enables the Nios-II processor to recognize the configuration of, and communicate with, any peripherals/modules that we choose to attach to the bus.

2.2.2 Nios-II processor

The Nios-II soft-core microprocessor is an instantly-deployable, 32-bit RISC processor that provides a plethora of configuration options. Programs can be written for Nios-II using the provided Nios-II IDE in C/C++.

2.2.3 SDRAM and Flash memory

The Nios-II processor (and, by proxy, uClinux) uses the 8 MB of available SDRAM (through the SDRAM controller) as its main memory for instructions and data. Additionally, uClinux also utilizes the 4 MB of Flash memory which is accessed using the Flash controller (implementing the Common Flash Interface).

2.2.4 USB

To maximize the usability of the system, USB support was added inside uClinux. To be able to access data stored on mass storage, the commonplace MS-DOS FAT/FAT32 file-system had to be supported, as well.

2.2.5 Audio Wrapper

The audio codec requires specific clocking and configuration, and this is why the audio wrapper was written. It encapsulates the codec controller, and integrates a FIFO into the process chain to buffer decoded data from the decoder.

2.3 Software components

The primary software components are uClinux and mpg123, the software decoder that's being run on top of uClinux.

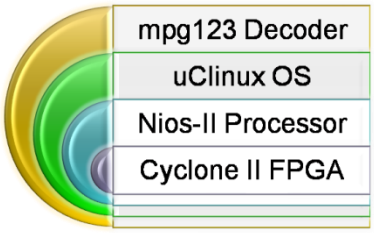

FIGURE 2 - THE SOFTWARE HIERARCHY

2.3.1 uClinux

uClinux is a lean variant of Linux designed specifically for microprocessors without Memory-Management Units (MMUs). It is targeted at embedded systems applications, and Nios-II is capable of running uClinux, given some modifications. It compiles into a 1 MB image, and decompresses into the available SDRAM on the board.

The primary advantage of uClinux is that it helps abstract away the complexities of writing drivers to talk to some of the peripherals on the board (expecially USB support, including support for the FAT/FAT32 file-system). Without uClinux, using a decoder directly on Nios-II would mean coming up with techniques to handle all the system calls that the decoder would make.

2.3.2 mpg123

The open-source decoder that was chosen for the implementation was mpg123 [8]. Other alternatives that were considered included LAME [5] and mp3play [9]. The former is a rather complex implementation designed for encoding, with decoding thrown in as an after-thought, while the latter has not been in active development for years.

mpg123 is fairly lightweight, with only around three dozen source files and accompanying headers, and is written in C. It also supports custom audio drivers, which we had to modify in order to get the decoder working with the digital-to-analog converter (DAC) available on our development board.

3. Implementation

3.1 Nios-II

There are three variants of the Nios-II processor, representing the spectrum of balance between area consumption and performance. The larger processors also offer more features, such as caches and hardware math support, while occupying more logic elements and memory bits, which are the two main storage metrics on the FPGA.

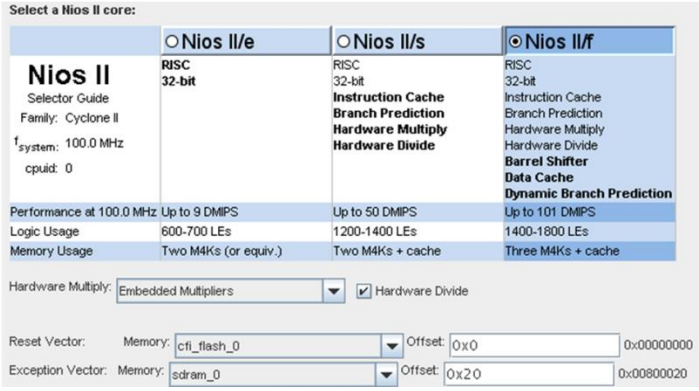

FIGURE 3 - NIOS-II PROCESSOR CONFIGURATION

There are several entries of interest in the Nios-II/f(ast) configuration:

- The instruction and data caches assist in the repetitive nature of the MP3 decoding algorithm.
- Hardware math helps speed up the decoding process, especially the barrel shifter, which is of importance since the software extracts 16 bits of data to send to the hardware using a shift operation.
- (Dynamic) Branch prediction is helpful during the frequent polling of the FIFO status from software.

Nios-II *can* fit entirely on the 512 KB of SRAM on the board, but this is insufficient to run uClinux, and so the SDRAM was used as main memory. Additionally, flash memory is also used.

The cache size for Nios-II was determined experimentally, and the greater the cache size, the less room there is on the FPGA for a larger FIFO. Reducing the cache size, however, results in much longer uClinux boot times (up to a minute), and the decoder startup time is also increased to almost 30 seconds. However, the actual decoding time is mostly unchanged by a change in cache size. In the end, the data and instruction caches were reduced to 512 bytes and 4 kb respectively.

3.2 uCLINUX

Compact, versatile and powerful as it is, uClinux is a complex beast to compile as Linux always has been. That is why pre-compiled distributions like Ubuntu are gaining popularity so quickly. The number of configuration options is staggering, and the odds are against the user to successfully compile the first, tenth, or hundredth time. A Linux environment is needed to compile uClinux, and we used CentOS [10], which is a clone of Red Hat Enterprise Linux, except without the professional customer support. Since uClinux is designed for a large variety of platforms (including the iPod), it may compile successfully on CentOS, but the actual transfer of the compiled image to the FPGA may still fail entirely.

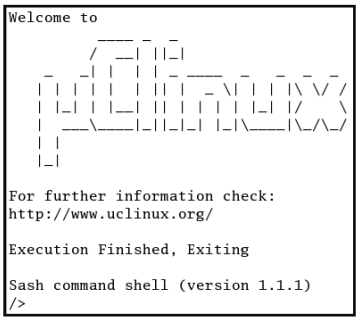

FIGURE 4 - THE UCLINUX BOOT SCREEN

All the necessary hardware descriptions (including any VHDL modules) must first be analyzed, synthesized, and burnt on the FPGA through Quartus. Once this is successful, the board is deemed ready to accept a Nios+uClinux image. This image is compiled by analyzing the system.ptf that is generated by SOPC builder (a system-architecture tool within the Quartus development environment), and is then downloaded through the Nios terminal, which then becomes the communication interface to uClinux.

After getting a vanilla version of uClinux running, we started adding other features to uClinux. The first important step is to provide USB Mass Storage Support for storing our mp3 files on commonplace Flash memory drives. We have achieved this by adding multiple USB-related modules such as FAT/FAT32 file-system, and the nXp-IS1362 USB Controller on the uClinux configuration. Upon uClinux bootup, a USB Flash Drive is detected as soon as it is plugged into the port. The *mount* command is used to mount the USB Flash Drive to "/mnt" directory for file access.

While the advantages of uClinux were known early on (and have been described in the design section), the disadvantages were only apparent after implementation and testing. The main disadvantage is that uClinux adds some non-determinism to the execution time of the decoder. This made benchmarking difficult, since a decoding run's initial decode time can fluctuate from five seconds to seven seconds without rhyme or reason. Additionally, the Nios-II code profiler, which is useful for time-profiling C code running on the processor, cannot be used when there is an additional processing layer on top of the Nios-II processor. This made debugging even more difficult. We worked around this by utilizing the slow, limited console output options on the decoder itself.

3.3 Audio Codec Controller

To interface the Wolfson WM8731 audio codec, a controller is written in VHDL to generate the necessary clocks and shifting data from the FIFO to the DAC. WM8731 is operating under a main clock of 18.432MHz generated from an external crystal oscillator on the DE2 board. For the data transfer, we need two additional clocks. We are using 48kHz as the frame rate because this is the standard sampling rate for PCM audio. This is generated by a hardware clock divider in the audio codec controller. Apart from the frame clock, we also need a bit clock to shift the 16-bit data to the DAC. The bit clock rate is 16 times that of the frame clock which is 16*48kHz. The reason we need the shifter is because data from the FIFO is parallel and the data bus of the DAC is serial so that we need to serialize data from the FIFO for the DAC.

3.4 FIFO

Due to the uncertainty of decoding, we need a buffer to temporarily store the data before the data is piped into the DAC to ensure smooth playback. Although we have other choices such as on-board SDRAM, on-chip SRAM a, we choose FIFO (First In First Out) as the buffer for the following reason. As an external memory, on-board SDRAM is slower comparing to the on-chip memory and it's mainly occupied by the uClinux. SRAM is fast, but you cannot read, write access it at the same time. On the other hand, FIFO has dual-port access to support simultaneous read/write operation. Therefore, FIFO is the ideal choice for buffer data.

We use a dual clock FIFO so that read and write operation are clocked at different rate with reading clock of 18.432MHz to match that of the codec and writing clock of 100MHz, same as the Nios-II. The FIFO is configured in legacy mode so that a read/write request signal need to be asserted and the data will be put on to the FIFO port at the next read/write cycle. Apparently, the bigger the FIFO is, the better performance we should be able to achieve. The largest FIFO we can fit within our design on the Cyclone II FPGA is 16k deep by 16-bit wide, but at the cost of cutting Nios-II cache. Before, we were using 8k FIFO, which don't need extra sacrifice in other components.

We also turned on the port wrusedw to check how much amount of space in the FIFO is being used. By pulling the status of the FIFO, we are able to take necessary actions in both software and hardware when the FIFO is full. This is very important for smooth playback.

3.5 Audio Wrapper

For mpg123 decoder to access the audio resources, we wrote an audio wrapper to interface FIFO with the DAC and encapsulate them as a whole unit so that it can be accessed through the address and data bus.

We receive decoded PCM data from the mpg123 through the Avalon bus and it's written into the FIFO by activating write request signal every write cycle. We also receive rate data in each frame and use this to adjust out sampling rate so that it matches with the 48kHz sampling rate of the DAC.

We also check the status of the FIFO through wrusedw signal constantly and generate a FIFO full signal that we used to activate waitrequest and pauses the Nios-II CPU.

3.6 mpg123

We picked an open-source MP3 decoder - mpg123 because it is a well known high-efficiency MP3 decoder. It also supports multiple audio interfaces including a "dummy" audio driver. This basically allows us to write our own audio driver to interface the WM8371 audio codec on the DE2 board since it's not pre-supported by any known decoders.

The decoder outputs 16-bit raw PCM audio data to the audio wrapper. We specify the audio wrapper address in the dummy.c and write decoded data as well as the sampling rate of each frame from the software buffer in mpg123 to the FIFO which is a hardware buffer. We also implemented a software pulling before writing data to the FIFO to check the status of the FIFO. If the FIFO is full, we don't write any data into it until it's no longer full.

Since mpg123 is made to support a wide variety of platforms, several compiling flags need to be address using configure.ac before compilation to customize the mpg123 for uClinux on Nios-II, otherwise it won't even compile with the gcc cross-compiler. As we are compiling the mpg123 on CentOS, we need the cross-compiler to compile mpg123 for Nios2-uClinux.

TABLE 1 - COMPILATION FLAGS

--host=nios2-linux-uclibc --target=nios2-linux-uclibc	Specify the target platform
--with-cpu=generic_nofpu	Specify CPU type (generic) and disable floating point support
--with-optimization=4	gcc Optimization
--enable-gapless=no	Disable gapless for performance
--with-audio=dummy	Specify the dummy audio driver
CC="nios2-linux-uclibc-gcc -D__KERNEL__ -elf2flt"	Enable gcc cross-compiler and convert header for compatibility
#define NOXFERMEM	no memory transfer

The most computational intensive part of the MP3 decoding is the DCT calculation.

3.7 Problems

The main problem we encountered is the stuttering sound during playback. Initially, we thought this was because the Nios-II is not fast enough to decode. Therefore, we intended to implement parts of the decoding algorithm in hardware, hoping to improve overall system performance. However, we later discovered - by polling the status of the FIFO from software - that the FIFO is almost always full during decoding. This means the speed that the decoder is writing data into the FIFO is faster than the speed that the DAC is reading data from the FIFO. This is quite surprising. To further ensure that the decoder is not the system bottleneck but the DAC, we performed an experiment.

Dct.c is where the decoder computes the Discrete Cosine Transform (DCT). In this file, we put a single-tone sine-wave generator after the DCT calculation (we discarded the decoded data that is the result of the DCT operation). We then put the sine wave data to the software buffer. This gives out clipping sound as expected. Then we totally removed the DCT calculation in the C file, and just put the sine-wave data to the buffer. If it was the DCT calculation that was taking too much time resulting the intermittent clipping in the sound, then, by removing the DCT calculation, the removal should speed up the time needed to "decode" each frame and improve the sound quality. However, no significant difference is observed after removing the DCT. This further confirmed that the decoder is not the bottleneck of the system.

3.8 Solutions

To solve the stuttering in the sound, we tried various methods.

First, we tried using the *waitrequest* signal on the Avalon bus to pause the CPU when the FIFO is full. When the processor senses the *waitrequest* signal, it freezes itself (by sending NOPS into the pipeline). When the FIFO's fill-limit falls below a certain threshold, we revoke the *waitrequest*. Unfortunately, for reasons unknown, it seems that pausing and resuming uClinux degrades the audio quality further, leading to noticeable pauses in the music.

Second, we tried simply discarding the decoded data, both at the source (in software), and at the destination (in hardware) when the FIFO was detected to be full. This led to better performance, but the music now skipped whenever the FIFO was full since frames were being lost.

Finally, we tried to increase the size of the FIFO. Earlier in the course of the project, we thought 8k was the largest FIFO that we could fit on the FPGA. After tweaking with the hardware configuration and reducing the CPU cache from 16k to 4k, we managed to fit a 16k FIFO. However, the sound quality is not significantly better than with the 8k FIFO. The most likely explanation for this behavior is that the audio-quality degradation due to reduction in the cache size of the processor is not outweighed by the increase in the FIFO size.

4. Results

Since audio quality is utterly subjective, we've cataloged the average quality of the decoded audio, both in terms of clarity, as well as the lack of deleterious effects such as stutters, pauses, and skips. In addition, we measured the time between the decoder starts up, and the time the audio starts playing, which we refer to as initial decoding time. The results of varying the FIFO size are shown in the graph below, and the format of the data is as follows: *<size of FIFO>*, *<bitrate of audio file>*, *<sampling rate of audio file>*.

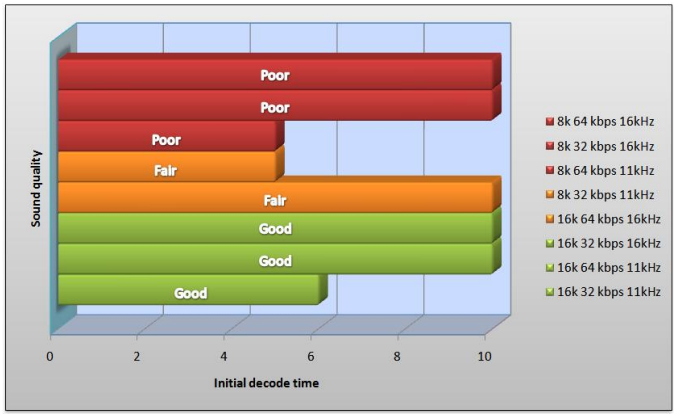

FIGURE 5 - BENCHMARKS AND RESULTS

Clearly, having a larger FIFO provides better-sounding music, but the initial decode time is also increased, although not noticeably. Increasing the bitrate of the audio file results in almost a doubling of the initial decode time, and also a noticeable decrease in the audio quality, though this latter effect only happens when paired with a change in the sampling frequency. The sampling frequency of the audio file is by far the biggest differentiator in terms of audio quality; increasing the sampling frequency drastically increases the presence of stutters and skips in the playback.

5. Future Directions

As mentioned in the previous section, we have noticed some shortcomings in our implementation. Future work would involve the following:

- Speed up the initial decoding time. This would involve finding a better balance between cache and FIFO sizes.
- Playback contains artifacts. There are several workarounds to this.
  - Making the FIFO bigger would help alleviate the issue where the decoder is blocked from writing into a full FIFO. However, this would involve migrating the design onto a larger FPGA (such as the Stratix family of devices).
  - Decoded data can be shunted into a new buffer in the SRAM until enough room exists in the FIFO. At this point, data from the SRAM is sent into the FIFO, and normal decoder<->FIFO interaction resumes. This change would require precise synchronization between the SRAM, software, and FIFO.
- The buffer in software can be dynamically allocated; i.e., more memory is allocated only if the buffer is filled, meaning that the decoder is producing data faster than it can be written into the FIFO.

6. Conclusion

Our implementation attempt was mostly successful, but other than general slowdowns due to the slow system clock there are only a few minor artifacts in the music. uClinux has proven beneficial, but has taken its toll on the debugging process and could have possibly introduced further delay in the system. This project has shown that it is feasible to port an open-source MP3 decoding implementation onto the FPGA platform.

7. Acknowledgments

The authors thank Prof. Luca Carloni. We would also like to thank David Lariviere for his help in this (sometimes daunting) project.

References

[1] "Cyclone II FPGA." Altera. <http://www.altera.com/products/devices/cyclone2/overview/cy2-overview.html>.
[2] "Nios-II, the World's Most Veratile Embedded Processor." Altera. <http://www.altera.com/products/ip/processors/nios2/ni2-index.html>.
<http://www.altera.com/products/devices/cyclone2/overview/cy2-overview.html>
[2] Nios-II. “The world’s most versatile embedded processor” <http://www.altera.com/products/ip/processors/nios2/ni2-index.html>
[3] "uClinux - Embedded Linux Microcontroller Project."for processors lacking Memory Management Units. <http://www.uclinux.org/>
[4] The Nios-II uClinux. <http://www.uclinux.org>.
[4] “uClinux – JotSpot Wiki.” JotSpot. <http://nioswiki,jotcom/%C2%B5linux>.
[5] "LAME MP3 Encoder ::: About." SourceForge. <http://lame.sourceforge.net/about.php>.
[6] "MP3." Wikipedia. <http://en.wikipedia.org/wiki/MP3>.
[7] The Avalon Fabric Specifcation. <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>. http://en.wikipedia.org/wiki/MP3
[7] The Avalon communication fabric. <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>
[8] "Mp123, Fast MP3 Player for Linux and UNIX Systems." Mpg123. <http://www.mpg123.de/>. http://www.mpg123.de/
[9] "Index of mp3 Subversion Tree." Digium. <http://svn.digium.com/view/aadk/uClinux/trunk/uClinux-dist/user/mp3play/?pathrev=249>.
[10] "CentOS - the Community ENTerprise Operating System." CentOS. <http://centos.org/>.
[9] mp3play. <http://svn.digium.com/view/aadk/uClinux/trunk/uClinux-dist/user/mp3play/?pathrev=249>

About the Authors

Shrivathsa Bhargav graduated from the State University of New York at Stony Brook with a Bachelors’ degree in Electrical Engineering (*Cum Laude*). His current interests include digital VLSI circuit design, and embedded systems.

Bingrui (Revan) Yang graduated from Fudan University, Shanghai with a Bachelors’ degree in Telecommunications Engineering. His interests are in analog IC design and RF communications systems.