ECE532: Digital System Design

Audio Visualization Device

Group Report

Students: Jennifer Mo 992476762
Gary Pong 992339144
Jason Wong 992353421

Date: April 2, 2007

Table of contents

1 Overview

The goal of this project is to implement an audio visualization device which takes an input sound signal and outputs a visualization of that signal onto a VGA monitor at a resolution of 640x480 in 24-bit colour. It is similar to the visualization feature found in popular audio players such as Windows Media Player and WinAmp.

Our hardware implementation of audio visualization uses a Microblaze CPU for processing and a rendering module to provide hardware acceleration for drawing to the VGA frame buffers. The AC97 audio codec samples the input sound signal and its result is processed by the Xilinx-provided Fast Fourier Transform block. Software running on the Microblaze then analyzes the frequency components of the signal and generates an appropriate visualization for it. The ZBT RAM modules are used as frame buffers to store the pixels to be drawn to the monitor by the VGA controller.

We have implemented three different visualizations that can be switched at run-time through the use of the multimedia board's push buttons. The block diagram for our system can be seen on the next page in Figure 1. The individual blocks will be discussed in more detail in Section 4: Description of Blocks.

2 Outcome

Our project has been successful in implementing three different audio visualizations. All three visualizations display the frequency spectrum of the input audio signal.

The most basic mode is implemented in software, bypassing the render module completely. In this mode, the CPU is responsible for setting each pixel on the screen individually by a write through the OPB bus to the ZBT RAM controller.

The second mode generates the same display as software rendering but is hardware accelerated. Operations such as clearing the back buffer and drawing a column of pixels are handled in hardware with minimal intervention by the CPU. The hardware is able to achieve much higher bandwidth to memory than the CPU using the OPB bus. This results in a significant speedup over the software implementation.

The third visualization mode that we have implemented is hardware rendering with fading effects. This mode builds upon the second mode listed above. However, instead of clearing the back buffer by writing black to every pixel, the back buffer is "cleared" by copying a faded version of the last frame that was drawn.

The frame rate for these three modes have been measured by timing the elapsed time in rendering 1000 frames and taking the average. The software implementation is able to render at 2.48 frames per second while the hardware implementations are able to render at 7.8 frames per second.

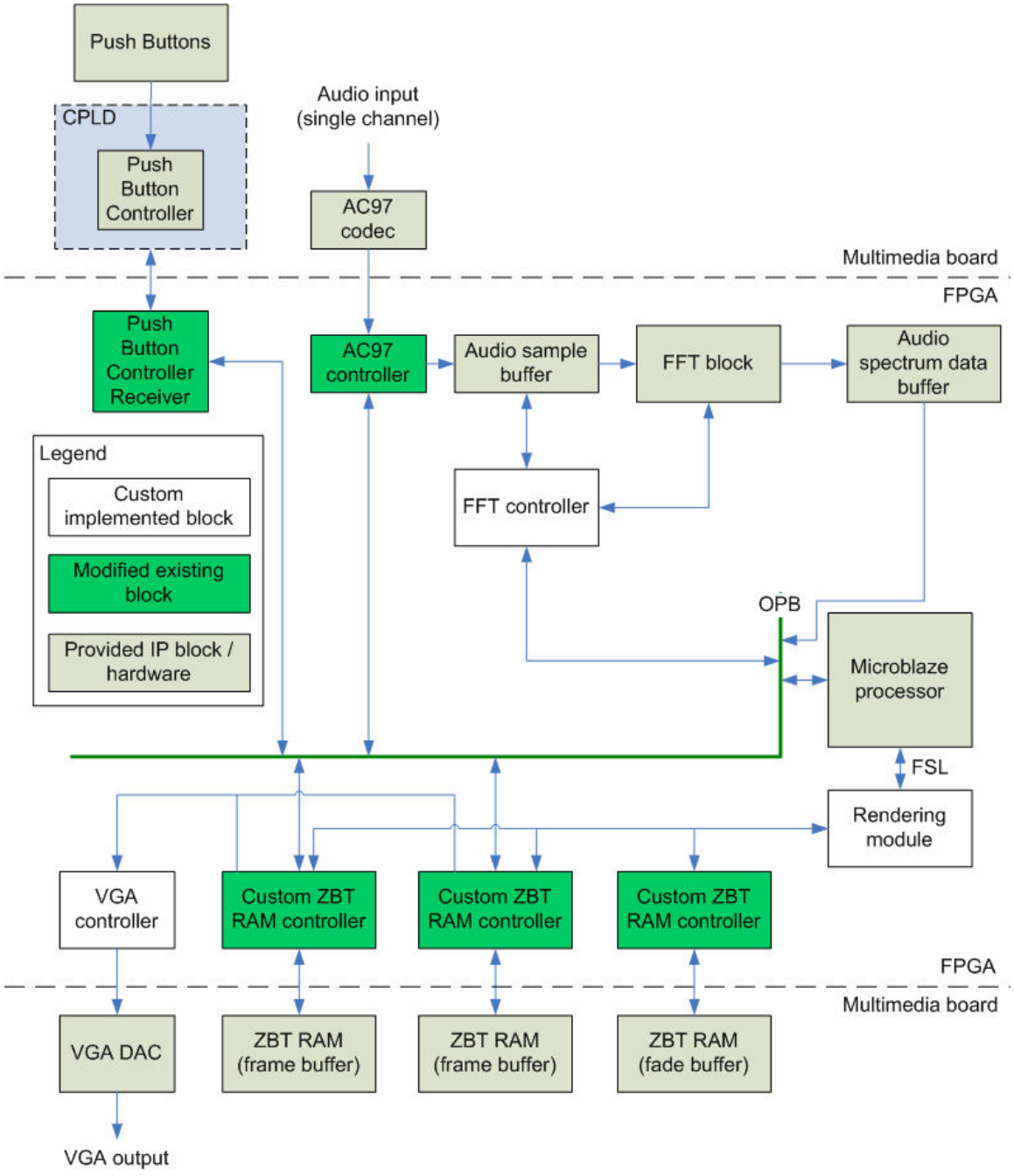

Figure 1: System block diagram for the audio visualization device.

3 Suggestions for Further Work

The VGA controller can be modified to provide a higher resolution output rather than the 640x480 currently supported. Each 2 MB module of ZBT RAM can support a back buffer of 800x600 at 32-bit colour.

The rendering module can be improved to support more visualization modes than the three currently supported. For example, it should be feasible to implement a visualization that looks like a water droplet rippling out from the middle of the screen in concentric circles.

In our current visualizations, only the FFT of the left audio channel is used. In most cases, the left and right channels are very similar so this does not make much of a difference. In future iterations, both audio channels can be used for some other interesting effects.

Finally, the fact that the hardware implementation with and without fading renders at the same number of frames per second suggests that the audio visualization device is now limited by something other than the rendering portion. The snoopy core can be used to profile the code to see where most time is now spent. The software portion of our algorithm contains many calculations that could potentially be improved upon in a hardware module.

4 Description of Blocks

This section describes the blocks found in the system block diagram seen in Figure 1.

4.1 Push button controller (on CPLD)

This module is contained in the *normal.jed* CPLD design that is programmed onto the CPLD by default. Due to pin-out limitations on the FPGA, the 10 push buttons on the multimedia board are connected to the CPLD. When the push button states have changed and the ENTER push button is pressed, the new push button states are sent serially by this push button controller to the FPGA.

4.2 Push button controller receiver (on FPGA)

This module is based on the "Push button scan" design provided by Xilinx at *http://www.xilinx.com/products/boards/multimedia/examples.htm*.

This module decodes the serial push button states received from the CPLD and stores it into a register. We have added an OPB interface to this module so that the Microblaze CPU can read the push button states directly from the register.

4.3 AC97 Codec

This AC97 codec chip samples an input audio signal with 16-bits of accuracy for each channel (left and right) at sampling rates of up to 48 kHz. We have configured the audio input to be sampled from the RCA input jacks rather than the microphone. Please see the datasheet found here: http://www.national.com/pf/LM/LM4549A.html

4.4 AC97 Controller and Audio Sample Buffer

The AC97 controller was taken from the AudioToMIDI project in 2005. We have modified this controller to store the audio samples in the audio sample buffer to be accessed later by the FFT block. The audio sample buffer is a 1024-word, 16-bit wide FIFO created using the Xilinx CoreGen tool. For simplicity, only samples from the left channel are recorded for use in our FFT.

The AC97 controller is controlled by the Microblaze processor via the OPB bus and the audio sample buffer is directly connected to the FFT controller pcore.

For more information on the AC97 controller, please see the "AC97 Sound Controller with Device Driver" document located here:
http://www.eecg.toronto.edu/~pc/courses/432/2004/projects/ac97controller.doc.

4.5 FFT Controller, FFT Block, and Audio Spectrum Data Buffer

The FFT controller is a custom-designed module connected to the OPB bus and is used by the CPU to initiate a FFT operation on the audio samples. The CPU can check the status of the FFT operation by reading from the FFT controller.

When a FFT operation is initiated, the FFT controller loads the data from the audio sample buffer into the FFT block. The FFT block is an instance of Xilinx's FFT v3.1 core configured as a radix-2 implementation of a 1024-point FFT.

When the FFT block is finished, the results are unloaded into the audio spectrum data buffer. This data buffer is a 1024-word, 32-bit wide memory block formed from an instance of Xilinx's blk_mem_gen_v2_3 core. There are two such data buffers: one for the real component of the FFT and one for the imaginary component of the FFT.

The audio spectrum data buffer is connected to the OPB interface such that the CPU can perform random access reads to the data.

4.6 Rendering Module

The rendering module is a custom-designed module connected to the Microblaze processor by a FSL bus. Blocking FSL writes are used by the processor to send commands to the render module. Commands understood by the rendering module include: clear back buffer, fade back buffer, flip front and back buffers, and draw fft bin (draws a column of pixels representing one FFT bin).

Commands are sent as a packet starting with a 32-bit opcode followed by a variable number of 32-bit arguments depending on the opcode. The different command packets are shown below in Table 1.

Opcode	OP_CLEAR_BUFFER	OP_FADE_BUFFER	OP_FLIP_BUFFER	OP_DRAW_FFT_BIN
Arg #1	Colour used to clear buffer	Fade value to subtract from each pixel		Height of FFT bin
Arg #2				FFT bin to draw (x coordinate)
Arg #3				Colour used to draw bin

Table 1: The different command packets supported by the render module.

This module provides all of the hardware acceleration required to render an image into the back buffer. Each command written to the FSL replaces many OPB writes that the processor would otherwise have needed to do had it been performing software rendering.

The commands are also parameterized such that there is great flexibility in the visualizations that could be generated without needing to change the hardware. We found the parameterization to be exceedingly helpful when we were fine-tuning our visualization. We could change the rate of fading and experiment with the colour schemes without ever modifying the hardware.

4.7 ZBT RAM Controller

This module is based on the "ZBT RAM Memory Controller" design provided by Xilinx at *http://www.xilinx.com/products/boards/multimedia/examples.htm*.

The example module's interface consists of the actual signals sent to the ZBT RAM chips. To simplify access to memory, the example controller was placed within a wrapper module which exposed a higher-level interface. The wrapper module also provided support for overlapped read/write transactions and burst transactions. An OPB interface and two custom interfaces have also been added to the wrapper module to multiplex requests from the three possible sources.

The OPB interface is used in the software rendering mode and the two custom interfaces are used in the hardware rendering modes. One custom interface is connected to the VGA controller for use when the RAM is used as a front buffer and the remaining interface is connected to the render module when the RAM is used as a back buffer.

A simulation of burst memory operations using the custom memory interface is shown below in Figure 2. A burst read is initiated in the second clock cycle, corresponding to the assertion of the `MIF1_read` signal. The burst size of the operation is indicated by the `MIF1_burst_size` signal, which has a decimal value of 3 in this case – this corresponds to a burst size of 4 words. The data read from the ZBT RAM chip appears on the read bus `MIF_read_data` in cycles 7 to 10, corresponding to when the `MIF1_read_data_ready` signal is asserted.

As can be seen in the simulation, a write operation is requested by asserting the `MIF1_write` signal starting on cycle 7. This overlaps with the previous read operation but is a valid operation because the signals necessary to perform a burst read have already been sent to the ZBT RAM chip. This is signified by the assertion of the `MIF1_op_done` signal in cycle 6, which means that another memory operation can be issued to the RAM chip.

For the burst write operation, the `MIF1_write_data_request` signal is asserted whenever the controller wants the next data word to be written. This signal is asserted for three cycles to request the three additional words for a total of 4 words; the first word was provided when the write was first requested (in cycle 7). The data word to be written is provided on the `MIF1_write_data` bus. Once again, the `MIF1_op_done` signal is asserted (in cycle 11) after all the necessary signals have been sent to the ZBT RAM chip. Internally, the write operation is still in progress in cycle 11, but this is not a detail the user of the RAM interface needs to consider.

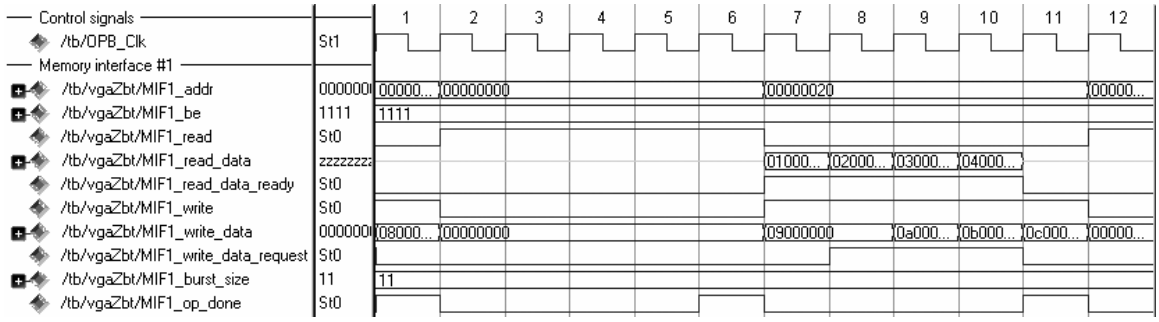

Figure 2: ZBT RAM custom interface simulation

4.8 VGA Controller

The VGA controller is a custom-designed module designed to output VGA video at a resolution of 640x480 at a refresh rate of 60 Hz using 24-bit colour (8-bits per channel). It is connected to the OPB bus to allow the processor to signal when back and front buffers should be flipped. It is also directly connected to two of the ZBT RAM controllers. Only the RAM acting as the current front buffer is used at any given time.

There is a pixel buffer in the VGA controller to prefetch pixels from ZBT RAM. This buffer hides the ZBT RAM latencies and also serves as a convenient way for the pixel data to be crossed from the ZBT RAM clock domain into the VGA clock domain.

For VGA timing values such as the front porch and back porch, we followed the numbers found on this page: http://web.mit.edu/6.111/www/s2004/NEWKIT/vga.shtml

4.9 Microblaze Processor and Software

Despite having hardware accelerated rendering, the Microblaze processor still performs most of the computations required in generating our visualization. Part of the reason for this is by design. By leaving our visualization algorithm in software, we were able to tweak and tune it until we were satisfied with the result. The hardware rendering module was designed and parameterized with this in mind.

The pseudocode for our algorithm is as follows:

start AC97 recording;

```
while(true) {
      start FFT;
      wait for FFT;

      calculate height of each FFT bin from its magnitude squared (sum
      the squares of the real and imaginary components of the FFT bin);

      calculate average height;

      from average height, calculate the colour intensity (the higher
      the average height, the higher the volume of the song and thus
      the brighter the colour intensity);

      for each fft bin {
            figure out the colour to use for this bin;
            draw the bin either in software or using rendering module;
      }

      flip back buffer to front;
      clear new back buffer or apply fading to new back buffer;
}
```

The most complex part of our algorithm is in calculating which colour to use for each FFT bin. Currently, our algorithm creates a shifting colour gradient that produces a pleasing rainbow effect when combined with fading. Please see our source code for more details.

5 Clock Domains

The following clock domains are present in our system:

Clock	Components
50 MHz system clock	OPB, FSL, Microblaze, Rendering module, ZBT RAM, FFT, AC97 controller
25.2 MHz VGA clock	VGA controller
12.288 MHz bit clock	AC97 codec and controller

The transition between clock domains are made exclusively using FIFOs. The AC97 bit clock and 50 MHz system clock boundary is crossed using the audio sample FIFO. The 50 MHz system clock and 25.2 MHz VGA clock boundary is crossed using the VGA's pixel buffering FIFO.

6 Description of Design Tree

The following is a description of our directory structure. For the directories shown, only the files of interest are listed.

avfpga\	Project root directory.
avfpga\code\	Contains the software component of our project.
├system.c	Entry point to our software with main algorithm.
├colours.h	Predefined constants for colours.
├renderops.h	Function prototypes for render module operations.
└renderops.c	Function definitions for render module operations
avfpga\drivers\ac97_v1_00_a\	Directory containing the AC97 driver.
avfpga\pcores\	Directory for user created peripheral cores.
├clk_align_v1_00_a\	Directory containing the clock align module used to assert reset until the DCMs have locked.
├opb_ac97_controller_v1_00_a\	Directory containing the AC97 controller.
│ ├hdl\vhdl\fft_fifo.vhd	Added FIFO to store audio samples for the FFT.
│ └hdl\vhdl\opb_ac97_core.vhd	Modified to store audio samples directly into FIFO.

├opb_fft_controller_v1_00_a\	Directory containing the FFT controller.
│ ├hdl\verilog\fft1024radix2.v	Xilinx FFT v3.1 IP core.
│ ├hdl\verilog\fft_result_mem.v	Memory to store real and imaginary FFT results.
│ └hdl\verilog\opb_fft_controller.v	FFT controller with OPB interface.
├opb_pb_v1_00_a\	Directory containing the push button controller.
│ └hdl\verilog\opb_pb.v	Push button controller with OPB interface.
├opb_vga_v1_00_a\	Directory containing the VGA controller.
│ ├hdl\verilog\opb_vga.v	VGA controller with OPB interface.
│ ├hdl\verilog\vga_fifo.v	FIFO used to prefetch pixel data from RAM.
│ └sim\	Directory containing simulation test benches
├opb_vga_zbt_v1_00_a\	Directory containing the ZBT RAM controller.
│ ├hdl\verilog\zbt_rw_sm.v	ZBT RAM read/write state machine module.
│ ├hdl\verilog\zbt_arbiter.v	Arbiter used when multiplexing memory requests.
│ ├hdl\verilog\opb_vga_zbt.v	Top-level entity with OPB and custom interface.
│ ├hdl\verilog\PIPELINES.v	Module that performs pipelining for reads/writes.
│ └sim\	Directory containing simulation test benches
└render_module_v1_00_a\	Directory containing the render module.
├hdl\verilog\render_module.v	Top-level entity with OPB and FSL interface.
├hdl\verilog\clear_buffer.v	Module that performs the clear buffer operation.
├hdl\verilog\draw_fft_bin.v	Module that draws a column of pixel for a FFT bin.
├hdl\verilog\fade_buffer.v	Module that performs the fading effect.
└sim\	Directory containing simulation test benches