MuseBox

Embedded Systems Design

CSEE4840

Final Report

Advisor: David Lariviere

Mark Aligbe ma2799

Sabina Smajlaj ss3912

Table of Contents

1. Background

Equalization is one of the most powerful techniques in audio manipulation. It allows you to accentuate or suppress certain frequencies, giving you the ability to heighten voices from a crowd, suppress a noisy frequency, or accentuate instruments in certain ranges of audio.

In order to be able to produce any of these effects, we first have to know the frequencies contained in the incoming audio. Once we know these frequencies, we can then use equalization to alter the amplitudes of the different frequencies.

Filters

The simplest kinds of filters are Low Pass Filters, High Pass Filters, and Band Pass Filters. Low Pass Filters pass low-frequency signals and attenuate signals with frequencies higher than the cutoff filter they're designed for. A High Pass Filter is the opposite of this; it passes high-frequency signals and cuts off all signals below a certain cutoff frequency. A Band Pass FIlter is a combination of these two. It attenuates frequencies outside of a given range (a band), while preserving frequencies in the band. Analog graphical equalizers are essentially a linear addition of bandpass filters, which operate in the time domain. For digital signals, this approach isn't possible as binary lacks the resolution of real voltage levels. An equivalent approach would be to compute the frequencies present in the signal, increase or decrease them accordingly.

Fast Fourier Transform

This is where the Fast Fourier Transform (FFT) comes in. The FFT is derived from the Discrete Fourier Transform (DFT), which is an algorithm that converts time to frequency and vice versa for discrete signals. In other words, we can use the FFT to read in an sampled signal (the audio) and convert it into frequencies that we can manipulate just like a graphical equalizer.

The FFT is computed in a similar way to the DFT but is much faster. whereas the DFT takes $O(N^2)$ and requires that the entire audio be present during processing, the FFT only takes at most O(N log N) and can be extended to compute inputs of arbitrary lengths. This is why we prefer to use the FFT for manipulating real-time audio.

The DFT is computed by the following equation:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1.$$

where $x_0...x_{N-1}$ are complex numbers. The DFT can be computed much more quickly by taking advantage of the independent nature of sums. By breaking up the total input into *r* blocks, the DFT can be computed significantly faster, and this divide and conquer approach forms the basis of the Cooley-Tukey that the Altera FFT MegaCore IP implements as well as FFTW (as well as Rader's and Bluestien's).

Equalizers

Equalizers come in various forms, with two of the most common forms being a parametric equalizer and a graphical equalizer. The graphical equalizer is simple a cascaded addition of multiple bandpass filters whereas parametric equalizers tend to be more configurable, and thus are usually digitally implemented (FFT). This it is more accurate to compare our implementation to that of a parametric filter, as it could be extended for such dynamic purposes, but its current implementation is equivalent to that of an analog graphical equalizer.

2. Proposal

MuseBox is an integrated audio equalizer and visualizer. We take audio from the input audio jack and output a waveform on the audio output jack. This waveform contains the equalized audio and will be used for creating a visual to display on a screen. The user is able to modify the equalizer; a 12 band equalizer ranging from frequencies 31Hz to 20KHz and from amplitudes -12dB to 12 dB. We configure the audio of the FPGA and provide an audio driver for Linux. We buffer in a fixed number of audio samples and use this buffer to compute the FFT of the audio waveform that we read in, using the FFTW library. Then, we scale the amplitudes by the values specified by the user. Finally, the inverse FFT is then computed and buffered into another audio out queue. This is the equalized audio. To create the visualizer, we take the digital audio input, compute the post equalization amplitudes of each frequency slot, and then send those to a display.

3. Design

Design

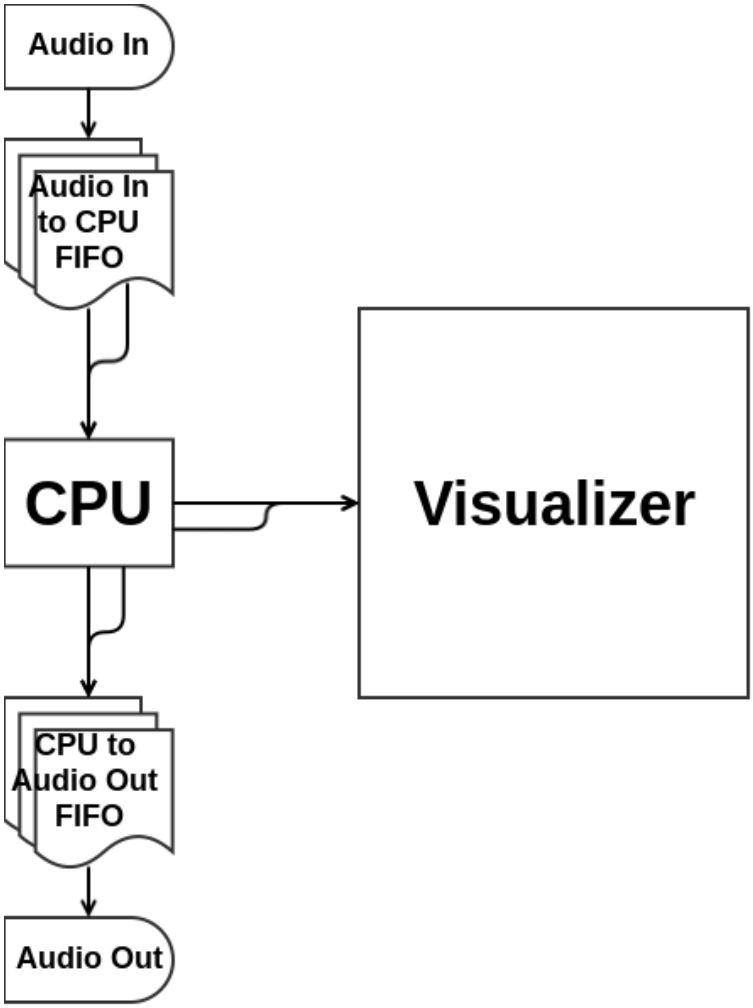

Figure 1: Our Project Design

Audio Input

The *Audio Input* is received serially by the *audio_codec* module as signed 16-bit values, representing a particular voltage level. These are then advertised and captured by a Dual Clock FIFO (Altera MegaCore IP), which buffers them until the CPU reads them (i.e. when the advertised FIFO size is the same or greater than the transform point size). These are then read by the CPU. Since the sample rate of the Audio Input is so much slower (kHz) than the CPU (GHz), it is necessary to buffer the inputs to achieve maximum efficiency, which is done in the *channel_master* module. Additionally, since the CPU is so much faster than the Audio Input, we can realistically guarantee that the CPU will complete its transform before the audio subsystem has produced another group of points to compute.

Equalizing/Visualizing

After the FFT is computed, the fixed point values will be read into memory. The userspace program will take this array of values and modify them according to user-specified boosts and cuts. The resulting amplitude levels will be displayed to the user as a spectrum of frequencies; a simple visualizer. The userspace program will compute the necessary pixel height to send to the framebuffer driver to draw the representative spectrum. The values will also be sent to the IFFT to compute the audio levels required to produce the desired sound.

Audio Output

The Audio Output Module will take the result from Inverse FFT and send it to audio out. This involves buffering the FFT output (in *channel_master*), and send that to audio out when requested by the *audio_codec*. The buffering is necessary since samples are sent to audio out on the order of kHz, but received on the order of MHz.

4. Implementation

FFT

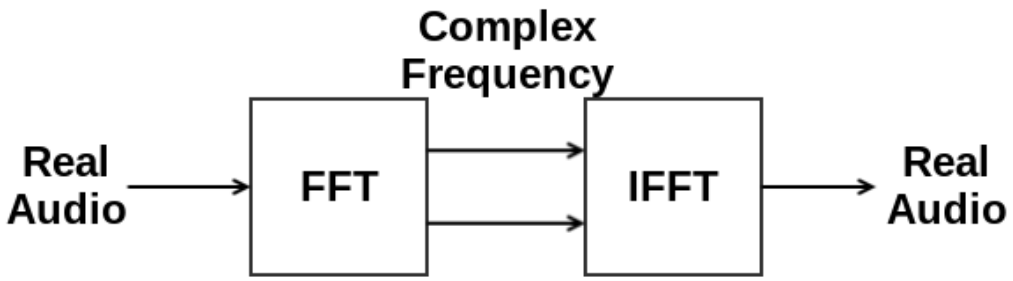

Figure 2: How the FFT interacts with Audio

To implement our design, we attempted two approaches. The first was purely hardware, where we created two, back-to-back FFTs (separated with a combinatorial equalizer) using the Altera FFT MegaCore function. We chose the variable streaming architecture, which implements a radix-2^2 single delay feedback architecture, using a fixed-point representation. The transform length of the FFT can be 2^m where 3 <= m <= 18. We read in 8k (m = 13), 16-bit samples from the audio and pass them into the buffer. This issue ran into many problems. First, the memory requirements were extremely high, and using a point size larger than 8k was impossible due to the memory requirements. The second issue is that of scaling. Since the FFT computes much faster than the source hardware, it is necessary to buffer the inputs and outputs, thus doubling the memory requirement for doubling the transform length. Additionally, using a single FFT

engine was no more efficient, as the intermediate FFT data had to be stored in a FIFO, which itself required on order the same amount of memory elements as an additional FFT. A third issue is that the appropriate scaling was hard to determine. Both FFT modes (inverse and direct) scale their inputs by a certain factor during computation, requiring the user to scale the output if they wish to achieve unity gain. There were many pitfalls associated with this in particular because the data was signed.

The second implementation, currently a work in progress, is sending the audio samples straight to the CPU, where the FFT will be performed in software. It is believed that this approach will be sufficiently fast, and also more scalable due to the ease of debugging and simplicity of scaling.

Visualizer

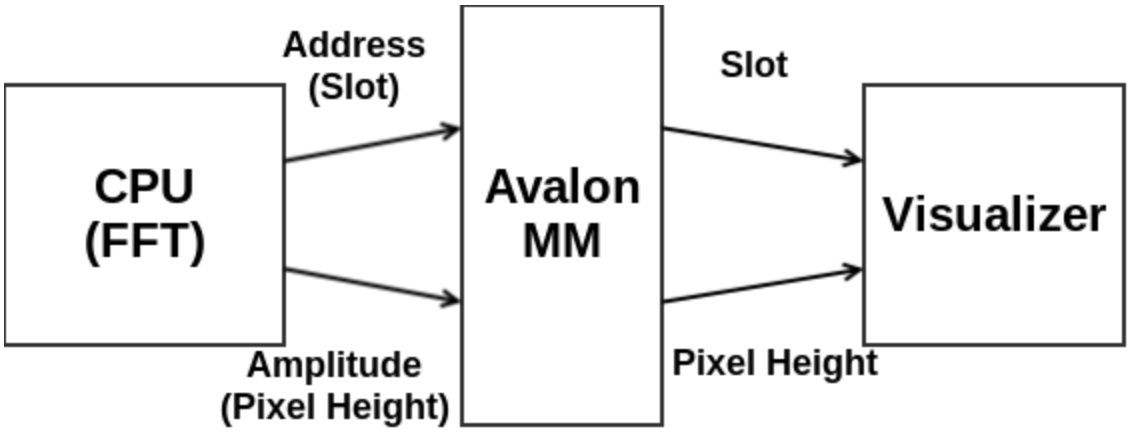

Figure 3: How the Visualizer interacts with the CPU and Avalon MM

For our visualizer, we followed a similar method to the third lab. We created a VGA peripheral which consists of a VGA raster to display to the screen, a user program to control it, and a device driver to facilitate coordination between the two. We used Quartus and Qsys to link our memory-mapped peripheral components to the ARM processor to be able to write directly to memory.

The device is described in *freq_spec.sv*. Our main user space visualizer program is described in *visualizer.c*. The device driver is described in *visualizer_driver.c* and *visualizer_driver.h*. This is the program that actually implements the FFT as described above, whereas the equalizer program just communicates occasionally with hardware.

When we compute the FFT, we want to display the information in a manner compatible with the equalizer. Hence we group the frequency bins into 12 “slots”, which are centered at 31, 72, 150….14000, and 20000. It then scales their amplitude in frequency using a triangular function

(Bartlett Window) where the center is the current slot, the left zero is the previous slot, and the right zero is the next slot. Each amplitude is computed as:

int amplitude = 20 log_10 [(real_numbers)^2 + (imaginary_numbers)^2]*

Finally, we divide the scaled amplitudes of each of these elements by the number of all elements in the window, thus ensuring that they form a probability function. This allows us to specify the bar height as a percentage of the screen occupied, allowing us to directly specify to *visualizer* the height it should take next for a bar. We take these amplitude heights and pass them to the device driver, *visualizer_driver.c*, which sends them to *freq_spec.sv*, which displays the frequencies at their respective heights in real time.

Equalizer

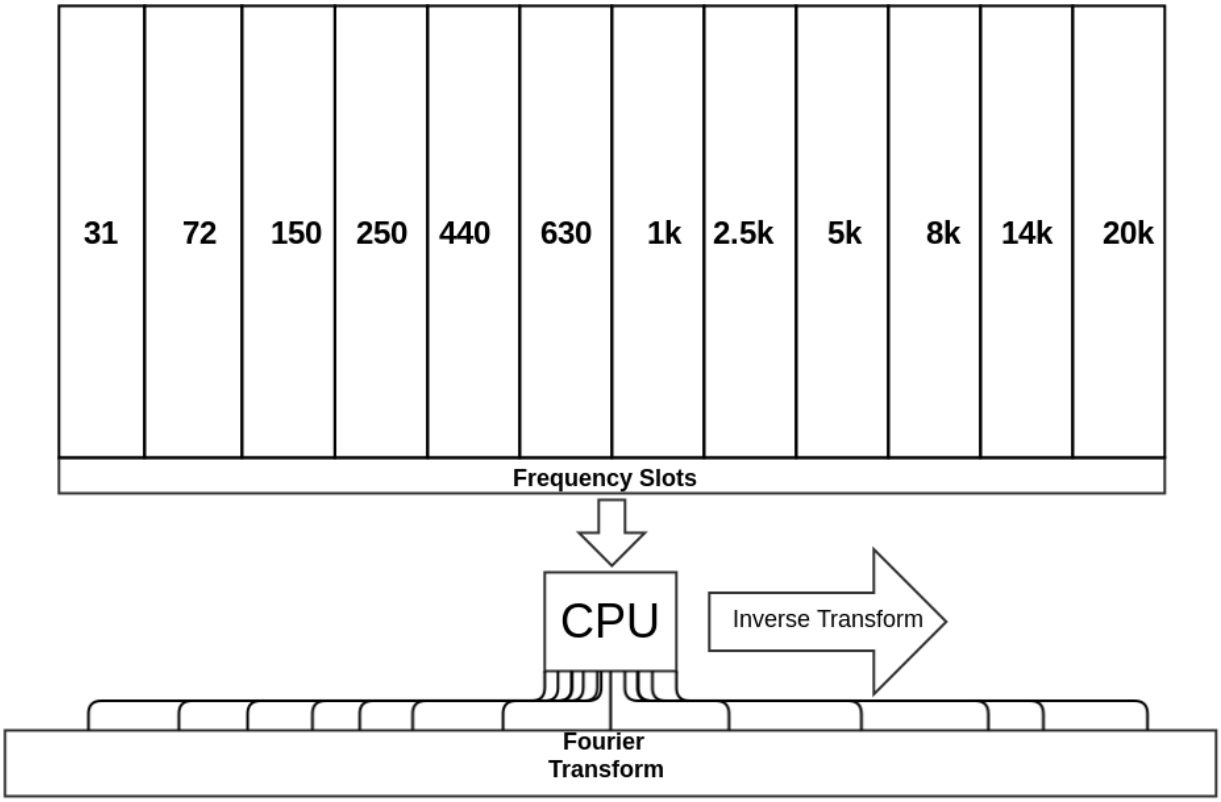

Figure 4: The interplay between the Fourier Transform and the Equalizer

We have talked at a high level of the actual equalization process, but now we go into the details. First, there is the very real problem of spectral leakage. A song is typically composed of many more samples than an FFT can process, which means that the FFT must process the complete signal in chunks. However, the FFT (and the original Fourier Transform) assumes that the signal it is computing over is complete. When this is not the case, the samples at the edge of the

sampled window can cause spectral leakage, meaning that they appear as frequencies other than what they actually are. To solve this issue, a method known as overlap-add is used:

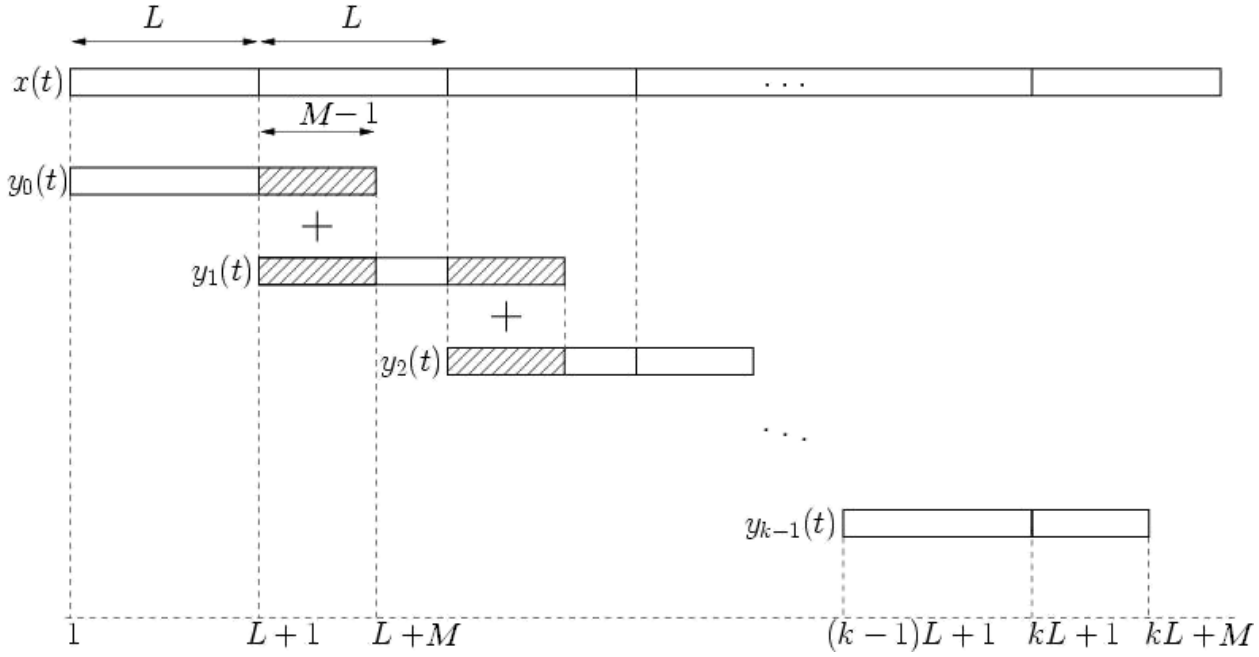

Figure 5: Overlap-Add visualized (source: http://en.wikipedia.org/wiki/Overlap%E2%80%93add_method)

Essentially, you compute the FFT of point size N, but for some M of those samples, you scale them by some function h(n) (typically some sort of window function).You use the N - M signals computed, but save the M samples for another computation. On the next computation, you compute M + (N - M), but scale the first M values (the same values scaled before) again, and add them (the old scaled M values and the new scaled M values), and restore the original stream integrity. For this purpose, a simple Bartlett window function with an M of 4096 is used. Our N, the FFT point size, is 32k (32,768).

For equalizing, we are scaling each bin (an FFT of point size 32k produces bins on a stream, sampled at 44.1kHz, of 44100/32000 ~ 1.38 Hz width) by the weighted average of the change in their nearest two neighbors. For example, bin 1000 has a frequency of approximately 32000/44100 * 1000 ~ 725 Hz. Thus, to smooth out frequency responses, the total change in amplitude would be:

$$\Delta B_{i-1} 32000/$$

Although we were originally planning on creating the equalizer interface in hardware (similar implementation to that of lab 3), we encountered roadblocks with the kernel and were unable to proceed.

Instead, we decided to take in the user input for the equalizer through the console. We query the user for the slot of frequency values (0 to 11) that they want to change the decibel value (1- to 25 to correspond to -12dB to +12dB) in. After the user enters this, we send this information to the peripheral equalizer via the *equalizer_driver.*

Audio Input

The audio input utilizes the SSM2603 audio codec, which is thinly wrapped by FPGA configurations. The codec is initialized through the use of *i2c_controller* and *i2c_av_config*. Afterwards, reading input is a process of shifting in the bit values on the ADC pin. The codec sends first the data on the left audio channel, then the data on the right side of the audio channel; in-between these pairs of data are 32 corresponding don't care bits. *audio_codec* takes care of managing the details of this, and sends out four signals indicating various parts of the audio path: next value for left channel DAC (sample_req[1]), next value for right channel DAC (sample_req[0]), next value for left channel ADC (sample_end[1]), and next value for right channel ADC (sample_end[0]). *channel_master* takes a particular action based on these signals, queueing the signals in the appropriate FIFO, or reading it out of a FIFO.

Audio Output

The output audio is sent from the CPU into the inverse FFT, where the data is converted back from frequency domain into the time domain. It then communicates with *channel_master* or Avalon MM, where it queues the two channels it computed into their corresponding FIFOs. When read on a sample_req, *audio_codec* takes care of shifting the data out on the DAC pin for audio out.

5. Results

To test our equalizer design, we can test it subjectively or objectively. On the subjective side, we can input a 440Hz sound wave and listen to see if the loudness changes in accordance with the equalizer setting. On a more empirical side, we can input a source file with certain amplitudes and then view the output file to see if those amplitudes have been changed according to our the equalizer input.

To empirically test the visualizer subsystem (including the user space program that calculates the bar position), we can use a file that produces predetermined decibel values for each of the frequency slots, and record if the computed heights match with the expected ratio.

6. Contributions and Team Work

Because there were only two of us, we both had to work on many aspects of the project. In the beginning, we spent a considerable amount of time researching the topic and the technology available to us. We worked together to come up with a design and and a plan for how we would build our design. We soon realized that it would be more efficient to split tasks. Mark began to focus on the FFT, getting audio in and out of it, and on building the equalizer, while Sabina began to work on the device drivers, visualizer, and interface for the equalizer .

7. Challenges and Lessons Learned

Test As You Go

One of the most important lessons that we learned was to test everything as we were building it. Since part of the concepts we were using were similar to the previous labs we had worked on, we had a tendency to focus on the logic of the code rather than testing it. We were writing modules similar to what we had written in lab 2 and lab 3, maintaining our confidence that the code would work because we had already written something similar to it. This, however, led to undesirable circumstance where our untested code ended up interfering with the rest of our project beyond just a few debugging issues. Had we tested our code from the very beginning, we would have realized these problems and had more time to pivot and create better solutions.

Communication is Key

Even working with just two people, communication can become an issue. This is especially true when we are each working on the project at varying time and in varying places. Since many of the computers in the 1235 do not have git, we we not always able to keep each other up to date on what we were working on. This led to us rewriting code the other person had already written. It also led to us sometimes not understanding the other person's code if a lot of changes were made at once. Because many of the computers did not have git, we ended up storing important files on our project-specific kernels but under different repositories from one another. We could have communicated better about the constant changes we were making and about our daily progress.

Start Design Early

It is important to start high-level design as soon as possible. Not only does this help give everyone a good idea of what is involved in the project and what needs to be done in the end, but it also helps when you want to set deadlines. When there is a concrete plan of action, it is easy to determine which parts of the modules should be done by certain dates. This helps everyone organize their time and efforts so that the team can achieve as many of our goals as possible by the final deadline, and get back on track when progress doesn't go according to plan. Planning and time management are essential skills for group work such as this. We had roadblocks in the

form of kernel issues, Avalon ST restrictions, and missing files which set us back considerably. However, if we had planned earlier, we would have had more time to tackle these issues more efficiently.

8. Advice

Meet Often

It is important to meet often. When you have many team members (even when you have just 2), it can be difficult to have everyone on point and working together. Meeting at least twice a week goes a long way to making sure everyone has a task and knows what their contributions are.

Choose a Set of Features Early

You don't need to implement every feature you envisioned in the beginning. It's more important to have a small list of robust features working than to try get every feature done and overwhelm yourself. Differentiating your essential features from your nice-to-haves is thus something you should decide as a team early on as you start implementing your design to keep yourselves on track.

9. Conclusion

We set out to build an audio equalizer and visualizer. For now, we are able to take in audio and send it through the FFT and into the CPU. However, when we try to scale the data in the equalizer, we introduce a large amount of static. We are currently working on fixing this issue.

After trying to implement the visualizer by a few different implementations, we decided the best choice would be a kernel module to write to the sockit screen and a user program to adjust the amplitudes of the incoming data. This worked out well and we ended up with a fast and aesthetic visualizer implemented in hardware.

10. References

http://www.zytrax.com/tech/audio/equalization.html [equalizer]

http://www.altera.com/literature/ug/ug_fft.pdf [FFT]

http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html [audio codec]

http://zhehaomao.com/blog/fpga/2014/01/26/sockit-9.html [filter exmaples]

http://en.wikipedia.org/wiki/Overlap%E2%80%93add_method [overlapp add]

http://www.fftw.org/ [FFTW C Library]

11. Source Code

```
/*
* fft_driver.c
* Device driver for FFT
* Stephen A. Edwards
* Columbia University
* edited by ma2799 and ss3912
*
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "fft_driver.h"

#define DRIVER_NAME "fft_driver"

/*
* Information about our device
*/
struct fft_driver_dev {
  struct resource res; /* Resource: our registers */
  void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

// Read the whole transform length from the fft
static void readTransform(struct complex_num *dataArray)
```

```
{
  int amountRead = 0;
  s16 *cnum = kmalloc(2 * sizeof(s16), GFP_KERNEL);
  s16 *ack = kmalloc(2 * sizeof(s16), GFP_KERNEL);
  // Keep reading while we haven't retrieved all values
  while (amountRead <= SAMPLENUM) {
        *((unsigned int *) cnum) = ioread32(dev.virtbase);
        *((unsigned int *) ack) = ioread32(dev.virtbase + 4);
        // If the data was good
        if (ack[1]) {
               if (amountRead != SAMPLENUM) {
                      dataArray[amountRead].real = cnum[0];
                      dataArray[amountRead].imag = cnum[1];
               } else {
                      dataArray[amountRead].real = ack[0];
                      dataArray[amountRead].imag = ack[1];
               }
               amountRead++;
        }
       }
       kfree(cnum);
       kfree(ack);
}

/*
* Handle ioctl() calls from userspace:
* Note extensive error checking of arguments
*/
static long fft_driver_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
       struct complex_num *dataArray = kmalloc(SAMPLENUM * sizeof(struct complex_num),
GFP_KERNEL); //allocating space for data array

  switch (cmd) {
  case FFT_DRIVER_READ_TRANSFORM:
        if (copy_from_user(dataArray, (struct complex_num *) arg,
                              sizeof(struct complex_num) * SAMPLENUM + 1)) {
               kfree(dataArray);
               return -EACCES;
        }
        readTransform(dataArray); //read into dataArray
        if (copy_to_user((struct complex_num *) arg, dataArray,
                       sizeof(struct complex_num) * SAMPLENUM + 1)) {
               kfree(dataArray);
               return -EACCES;
        }
        break;

  default:
        kfree(dataArray);
```

```
        return -EINVAL;
  }

  kfree(dataArray);
  return 0;
}

/* The operations our device knows how to do */
static const struct file_operations fft_driver_fops = {
  .owner        = THIS_MODULE,
  .unlocked_ioctl = fft_driver_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice fft_driver_misc_device = {
  .minor        = MISC_DYNAMIC_MINOR,
  .name         = DRIVER_NAME,
  .fops         = &fft_driver_fops,
};

/*
* Initialization code: get resources (registers) and display
* a welcome message
*/
static int __init fft_driver_probe(struct platform_device *pdev)
{
  int ret;

  /* Register ourselves as a misc device: creates /dev/fft_driver */
  ret = misc_register(&fft_driver_misc_device);

  /* Get the address of our registers from the device tree */
  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
  if (ret) {
        ret = -ENOENT;
        goto out_deregister;
  }

  /* Make sure we can use these registers */
  if (request_mem_region(dev.res.start, resource_size(&dev.res),
                       DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
  }

  /* Arrange access to our registers */
  dev.virtbase = of_iomap(pdev->dev.of_node, 0);
  if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
```

```
  }


  return 0;

out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&fft_driver_misc_device);
  return ret;
}

/* Clean-up code: release resources */
static int fft_driver_remove(struct platform_device *pdev)
{
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&fft_driver_misc_device);
  return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id fft_driver_of_match[] = {
  { .compatible = "altr,aud_to_fft" },
  {},
};
MODULE_DEVICE_TABLE(of, fft_driver_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver fft_driver_driver = {
  .driver    = {
        .name    = DRIVER_NAME,
        .owner    = THIS_MODULE,
        .of_match_table = of_match_ptr(fft_driver_of_match),
  },
  .remove    = __exit_p(fft_driver_remove),
};

/* Calball when the module is loaded: set things up */
static int __init fft_driver_init(void)
{
  pr_info(DRIVER_NAME ": init\n");
  return platform_driver_probe(&fft_driver_driver, fft_driver_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit fft_driver_exit(void)
{
```

```
  platform_driver_unregister(&fft_driver_driver);
  pr_info(DRIVER_NAME ": exit\n");
}

module_init(fft_driver_init);
module_exit(fft_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("MA2799, SS3912");
MODULE_DESCRIPTION("FFT Driver for MuseBox");
```

```
/*fft_driver.h */

#ifndef _FFT_DRIVER_H
#define _FFT_DRIVER_H

#define SAMPLENUM 8192

#include <linux/ioctl.h>

struct complex_num {
  s16 real;
  s16 imag;
};

#define FFT_DRIVER_MAGIC 8417

/* ioctls and their arguments */
#define FFT_DRIVER_READ_TRANSFORM  _IOWR(FFT_DRIVER_MAGIC, 2, struct complex_num *)

#endif
```

```
/* visualizer.c */

/*creates frequency spectrum visualizer for different audio frequencies
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "visualizer_driver.h"
```

```
#define USER_SPACE_FFT
#define VISUALIZER_MAGIC 4112
#define VISUALIZER_DRIVER_WRITE_FREQ _IOW(VISUALIZER_MAGIC, 1, int *)
#define VISUALIZER_DRIVER_READ_FFT _IOW(VISUALIZER_MAGIC, 1, int *)

#define SAMPLENUM 8192
#define H25K 4096

static int slot_values[12] = {31, 72, 150, 250, 440, 630, 1000, 2500, 5000, 8000, 14000,
20000};
static int bin_centers[12] = {6, 13, 28, 46, 82, 117, 186, 464, 929, 1486, 2601, 3715};
static int slot_heights[12];
static struct freq_slot slot_amps[12];
static struct complex_num freq_data[SAMPLENUM];

void read_samples()
{
    int fd = fopen("/dev/freq_spec", O_RDWR);
    struct complex_num *freq_data;
    freq_data = (complex_num*) malloc(SAMPLENUM*2);

    if (ioctl(fd, VISUALIZER_DRIVER_READ_FFT, freq_data) == -1)
        printf("VISUALIZER_DRIVER_READ_FFT failed: %s\n",
            strerror(errno));
    else {
        if (status & VISUALIZER_DRIVER_READ_FFT)
            puts("VISUALIZER_DRIVER_READ_FFT is not set");
        else
            puts("VISUALIZER_DRIVER_READ_FFT is set");
    }

}

void write_samples(int* dataArray)
{
    int fd = open("/dev/freq_spec", O_RDWR);
    if (ioctl(fd, VISUALIZER_DRIVER_WRITE_FREQ, dataArray) == -1){
        printf("VISUALIZER_DRIVER_WRITE_FREQ failed: %s\n", strerror(errno));
    }
    else {
        if (status){
            puts("VISUALIZER_DRIVER_WRITE_FREQ is not set");
        }
        else{
            puts("VISUALIZER_DRIVER_WRITE_FREQ is set");
        }
    }
}

int main()
```

```
{
    read_samples();
    int i, j;

    for(i= 1; i< 13; i++){
            double ampl_real = 0;
            double ampl_imag = 0;
                for(j=slot_values[i-1]; j< slot_values[i+1]; j++){
                    if(j<slot_values[i]){
                        ampl_real = ampl_real + (freq_data[i].real) * ((freq_data[j].real
- slot_values[i-1])/(slot_values[i]- slot_values[i-1]));
                        ampl_imag = ampl_imag + (freq_data[i].imag) * ((freq_data[j].imag
- slot_values[i-1])/(slot_values[i]- slot_values[i-1]));
                    }
                    else if(j>slot_values[i]){
                        ampl_real = ampl_real + (freq_data[i].real) *
((slot_values[i+1]-slot_values[i])/(slot_values[i+1]-freq_data[j].real));
                        ampl_imag = ampl_imag + (freq_data[i].imag) *
((slot_values[i+1]-slot_values[i])/(slot_values[i+1]-freq_data[j].imag));
                    }
                    else{
                        ampl_real = ampl_real + freq_data[j].real;
                        ampl_imag = ampl_imag + freq_data[j].imag;
                    }
                }

            double amp = (20*(log10((ampl_real)*(ampl_real) + (ampl_imag)*(ampl_imag))));
            slot_amps[i] = amp;
    }


    for( i = 1; i < 13; i++){
        int height;
        height = 479 - ((slot_amps[i])/186)*479;
        slot_heights[i] = height;

    }

    write_samples(slot_heights);

    printf("Visualizer Userspace program terminating\n");
    return 0;
}
--------------------------------------------------------------------------------
/* visualizer_driver.c */

/*
 * Stephen A. Edwards
 * Columbia University
 * edited by ma2799 and ss3912
```

```
 *
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
 #include "visualizer_driver.h"

 #define DRIVER_NAME "visualizer"
#define SAMPLENUM 8192

/*
 * Information about our device
 */
struct visualizer_driver_dev {
	struct resource res; /* Resource: our registers */
	void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
 * write slot_heights[12] to freq_spec.vs
 */
static void write_freq_mem(freq_slot *slot)
{
	iowrite16(slot->height, dev.virtbase + slot->addr);
}


/*
 * Handle ioctl() calls from userspace:
 * Note extensive error checking of arguments
 */
static long visualizer_driver_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    freq_slot bin;

	switch (cmd) {
	case VISUALIZER_WRITE_FREQ:
		if (copy_from_user(&bin, (freq_slot *) arg, sizeof(freq_slot)))
			return -EACCES;
```

```
                write_freq_mem(&bin); //write dataArray
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations visualizer_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = visualizer_driver_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice visualizer_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &visualizer_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init visualizer_probe(struct platform_device *pdev)
{
        int ret;

        /* Register ourselves as a misc device: creates /dev/visualizer */
        ret = misc_register(&visualizer_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
```

```
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}


	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&visualizer_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int visualizer_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&visualizer_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id visualizer_of_match[] = {
	{ .compatible = "altr,frec_spec" },
	{},
};
MODULE_DEVICE_TABLE(of, visualizer_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver visualizer_driver = {
	.driver	= {
		.name	= DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(visualizer_of_match),
	},
	.remove	= __exit_p(visualizer_remove),
};

/* Calball when the module is loaded: set things up */
static int __init visualizer_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&visualizer_driver, visualizer_probe);
}
```

```
/* Calball when the module is unloaded: release resources */
static void __exit visualizer_exit(void)
{
       platform_driver_unregister(&visualizer_driver);
       pr_info(DRIVER_NAME ": exit\n");
}

module_init(visualizer_init);
module_exit(visualizer_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("MA2799, SS3912");
MODULE_DESCRIPTION("VISUALIZER");
-------------------------------------------------------------------------------------------

/* visualizer_driver.h */

#ifndef _VISUALIZER_DRIVER_H
#define _VISUALIZER_DRIVER_H

#include <linux/ioctl.h>

#define VISUALIZER_MAGIC 4112

typedef struct {
    int16_t real;
    int16_t imag;
} complex_num;
typedef struct {
    int addr;
    int height;
} freq_slot;

/* ioctls and their arguments */
#define VISUALIZER_DRIVER_WRITE_FREQ _IOW(VISUALIZER_MAGIC, 1, u32 *)
#define VISUALIZER_DRIVER_READ_FFT _IOW(VISUALIZER_MAGIC, 1, u32 *)

 #endif
-------------------------------------------------------------------------------------------
/* cpu_audio.c */

/*
  * Stephen A. Edwards
 * Columbia University
 * edited by ma2799 and ss3912
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
```

```
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
 #include "cpu_audio.h"

 #define DRIVER_NAME "cpu_audio"
#define SAMPLENUM 32768

/*
 * Information about our device
 */
struct cpu_audio_dev {
       struct resource res; /* Resource: our registers */
       void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
 * write slot_heights[12] to freq_spec.vs
 */
static void read_audio_bank(sample_t *samples)
{
       *((unsigned int *) samples) = ioread32(dev.virtbase);
}

static void write_audio_bank(sample_t *samples)
{
       iowrite32(samples, dev.virtbase);
}


/*
 * Handle ioctl() calls from userspace:
 * Note extensive error checking of arguments
 */
static long cpu_audio_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    sample_t *samples = (sample_t *) kmalloc(SAMPLENUM * sizeof(sample_t), GFP_KERNEL);
    if (samples == NULL) {
       printk("samples is null\n");
       return -ENOMEM;
    }
       switch (cmd) {
       case CPU_AUDIO_READ_SAMPLES:
```

```
		if (copy_from_user(samples, (sample_t *) arg, sizeof(sample_t) *
SAMPLENUM))
			return -EACCES;
		read_audio_bank(samples); //write dataArray
		if (copy_to_user((sample_t *) arg, samples,
				sizeof(sample_t) * SAMPLENUM))
			return -EACCES;
		break;

	case CPU_AUDIO_WRITE_SAMPLES:
		if (copy_from_user(&samples, (sample_t *) arg, sizeof(sample_t) *
SAMPLENUM))
			return -EACCES;
		write_audio_bank(samples); //write dataArray
		break;

	default:
		return -EINVAL;
	}

	return 0;
}

/* The operations our device knows how to do */
static const struct file_operations cpu_audio_fops = {
	.owner		= THIS_MODULE,
	.unlocked_ioctl = cpu_audio_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice cpu_audio_misc_device = {
	.minor		= MISC_DYNAMIC_MINOR,
	.name		= DRIVER_NAME,
	.fops		= &cpu_audio_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init cpu_audio_probe(struct platform_device *pdev)
{
	int ret;

	/* Register ourselves as a misc device: creates /dev/visualizer */
	ret = misc_register(&cpu_audio_misc_device);

	/* Get the address of our registers from the device tree */
	ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
	if (ret) {
```

```
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }


        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&cpu_audio_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int cpu_audio_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&cpu_audio_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id cpu_audio_of_match[] = {
        { .compatible = "altr,cpu_audio" },
        {},
};
MODULE_DEVICE_TABLE(of, cpu_audio_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver cpu_audio_driver = {
        .driver        = {
                .name   = DRIVER_NAME,
```

```
                .owner = THIS_MODULE,
                .of_match_table = of_match_ptr(cpu_audio_of_match),
        },
        .remove        = __exit_p(cpu_audio_remove),
};

/* Calball when the module is loaded: set things up */
static int __init cpu_audio_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&cpu_audio_driver, cpu_audio_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit cpu_audio_exit(void)
{
        platform_driver_unregister(&cpu_audio_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(cpu_audio_init);
module_exit(cpu_audio_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("MA2799, SS3912");
MODULE_DESCRIPTION("CPU Audio with Equalizer");
-------------------------------------------------------------------------------------------
/* cpu_audio.h */

#ifndef _CPU_AUDIO_H
#define _CPU_AUDIO_H

#include <linux/ioctl.h>

#define CPU_AUDIO_MAGIC 4104

#ifndef CPU_AUDIO_US
struct sample {
    s16 left;
    s16 right;
 };
#else
struct sample {
        int16_t left;
        int16_t right;
};
#endif

typedef struct sample sample_t;
```

```
/* ioctls and their arguments */
#define CPU_AUDIO_READ_SAMPLES _IOR(CPU_AUDIO_MAGIC, 1, struct sample *) //writes to
freq_spec.sv
 #define CPU_AUDIO_WRITE_SAMPLES _IOW(CPU_AUDIO_MAGIC, 1, struct sample *) //writes to
freq_spec.sv

 #endif
--------------------------------------------------------------------------------
/* sample Makefile */

ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := cpu_audio.o visualizer_driver.o

else

# We are being compiled as a module: use the Kernel build system

       KERNEL_SOURCE := /usr/src/linux
        PWD := $(shell pwd)

default: module audio-test

module:
       ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

clean:
       ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
       ${RM} visualizer

socfpga.dtb : socfpga.dtb
       dtc -O dtb -o socfpga.dtb socfpga.dts

endif
--------------------------------------------------------------------------------
/* equalizer.sv */

// Frequency positions
parameter H31 = 14'd6, H72 = 14'd13, H150 = 14'd28, H250 = 14'd46, H440 = 14'd82, H630 =
14'd117, H1K = 14'd186, H2_5K = 14'd464, H5K = 14'd929, H8K = 14'd1486, H14K = 14'd2601,
H20K = 14'd3715;

module equalizer (
       // Frequency components
       input  [15:0] inreal,
       input  [15:0] inimag,
       output [15:0] outreal,
       output [15:0]  outimag,
       input insop,
```

```
        output outsop,
        input ineop,
        output outeop,
        input insovalid,
        output outsivalid,
        input out_out_can_accept_input,
        output in_out_can_accept_input,
        // The equalizer operates on a single clock
    input  clk,
        input  system_clk,
        input  reset,
        // The equalizer values. Only 5 bits matter
        input  [7:0]   writedata,
        output [7:0]   readdata,
        input  [3:0]   address,
        input  chipselect,
        input  write,
        input  read
);

// Shifting function
function [15:0] shift_num;
        input signed [6:0] exponent;
        input [15:0] num;

        begin
                case (exponent)
                        -7'sd15  : shift_num = {num[15],15'b0};
                        -7'sd14  : shift_num = {num[15],num[0],14'b0};
                        -7'sd13  : shift_num = {num[15],num[1:0],13'b0};
                        -7'sd12  : shift_num = {num[15],num[2:0],12'b0};
                        -7'sd11  : shift_num = {num[15],num[3:0],11'b0};
                        -7'sd10  : shift_num = {num[15],num[4:0],10'b0};
                        -7'sd9   : shift_num = {num[15],num[5:0],9'b0};
                        -7'sd8   : shift_num = {num[15],num[6:0],8'b0};
                        -7'sd7   : shift_num = {num[15],num[7:0],7'b0};
                        -7'sd6   : shift_num = {num[15],num[8:0],6'b0};
                        -7'sd5   : shift_num = {num[15],num[9:0],5'b0};
                        -7'sd4   : shift_num = {num[15],num[10:0],4'b0};
                        -7'sd3   : shift_num = {num[15],num[11:0],3'b0};
                        -7'sd2   : shift_num = {num[15],num[12:0],2'b0};
                        -7'sd1   : shift_num = {num[15],num[13:0],1'b0};
                        7'sd0    : shift_num = num;
                        7'sd1    : shift_num = {num[15],num[15:1]};
                        7'sd2    : shift_num = {num[15],num[15],num[15:2]};
                        7'sd3    : shift_num = {num[15],num[15],num[15],num[15:3]};
                        7'sd4    : shift_num = {num[15],num[15],num[15],num[15],num[15:4]};
                        7'sd5    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15:5]};
                        7'sd6    : shift_num =
```

```
{num[15],num[15],num[15],num[15],num[15],num[15],num[15:6]};
                        7'sd7    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:7]};
                        7'sd8    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:8]};
                        7'sd9    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:9]};
                        7'sd10   : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:1
0]};
                        7'sd11   : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],
num[15:11]};
                        7'sd12   : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],
num[15],num[15:12]};
                        // It's too low, just zero it
                        default  : shift_num = {16'b0};
                endcase
        end
endfunction

// The equalizer banks
reg [4:0] b31, b72, b150, b250, b440, b630, b1k, b2_5k, b5k, b8k, b14k, b20k;
reg signed [6:0] s31, s72, s150, s250, s440, s630, s1k, s2_5k, s5k, s8k, s14k, s20k;

// Signed assigner
always_comb begin
        s31 = b31;
        s31 = 7'sd13 - s31;
        s72 = b72;
        s72 = 7'sd13 - s72;
        s150 = b150;
        s150 = 7'sd13 - s150;
        s250 = b250;
        s250 = 7'sd13 - s250;
        s440 = b440;
        s440 = 7'sd13 - s440;
        s630 = b630;
        s630 = 7'sd13 - s630;
        s1k = b1k;
        s1k = 7'sd13 - s1k;
        s2_5k = b2_5k;
        s2_5k = 7'sd13 - s2_5k;
        s5k = b5k;
        s5k = 7'sd13 - s5k;
        s8k = b8k;
        s8k = 7'sd13 - s8k;
        s14k = b14k;
        s14k = 7'sd13 - s14k;
```

```
        s20k = b20k;
        s20k = 7'sd13 - s20k;
end

// Equalizer
reg [13:0] pos = 14'b0;
always_comb begin
        outsop = insop;
        outeop = ineop;
        outsivalid = insovalid;
        in_out_can_accept_input = out_out_can_accept_input;
        outimag = inimag;
        outreal = inreal;
        if (insop) begin
                pos = 14'b0;
        end else begin
                pos = pos + 14'b1;
        end
        // Scale by the ranges
        if (pos < H31) begin
                outimag = shift_num(s31, inimag);
                outreal = shift_num(s31, inreal);
        end else if (pos < H72) begin
                outimag = shift_num(s72, inimag);
                outreal = shift_num(s72, inreal);
        end else if (pos < H150) begin
                outimag = shift_num(s150, inimag);
                outreal = shift_num(s150, inreal);
        end else if (pos < H250) begin
                outimag = shift_num(s250, inimag);
                outreal = shift_num(s250, inreal);
        end else if (pos < H440) begin
                outimag = shift_num(s440, inimag);
                outreal = shift_num(s440, inreal);
        end else if (pos < H630) begin
                outimag = shift_num(s630, inimag);
                outreal = shift_num(s630, inreal);
        end else if (pos < H1K) begin
                outimag = shift_num(s1k, inimag);
                outreal = shift_num(s1k, inreal);
        end else if (pos < H2_5K) begin
                outimag = shift_num(s2_5k, inimag);
                outreal = shift_num(s2_5k, inreal);
        end else if (pos < H5K) begin
                outimag = shift_num(s72, inimag);
                outreal = shift_num(s72, inreal);
        end else if (pos < H8K) begin
                outimag = shift_num(s8k, inimag);
                outreal = shift_num(s8k, inreal);
        end else if (pos < H14K) begin
```

```
                outimag = shift_num(s14k, inimag);
                outreal = shift_num(s14k, inreal);
        end else if (pos < H20K) begin
                outimag = shift_num(s20k, inimag);
                outreal = shift_num(s20k, inreal);
        end
end
reg read_results;

// Initialization sequence
initial begin
        b31 <= 5'd13;
        b72 <= 5'd13;
        b150 <= 5'd13;
        b250 <= 5'd13;
        b440 <= 5'd13;
        b630 <= 5'd13;
        b1k <= 5'd13;
        b2_5k <= 5'd13;
        b5k <= 5'd13;
        b8k <= 5'd13;
        b14k <= 5'd13;
        b20k <= 5'd13;
        readdata <= 8'd0;
end
// End equalizer

// Bus interface logic
always_ff @(posedge system_clk) begin
        if (reset) begin
                b31 <= 5'd13;
                b72 <= 5'd13;
                b150 <= 5'd13;
                b250 <= 5'd13;
                b440 <= 5'd13;
                b630 <= 5'd13;
                b1k <= 5'd13;
                b2_5k <= 5'd13;
                b5k <= 5'd13;
                b8k <= 5'd13;
                b14k <= 5'd13;
                b20k <= 5'd13;
                readdata <= 8'd0;
        end else if (chipselect && write) begin
                case (address)
                        4'd0 : b31 <= writedata[4:0];
                        4'd1 : b72 <= writedata[4:0];
                        4'd2 : b150 <= writedata[4:0];
                        4'd3 : b250 <= writedata[4:0];
                        4'd4 : b440 <= writedata[4:0];
```

```
                        4'd5 : b630 <= writedata[4:0];
                        4'd6 : b1k <= writedata[4:0];
                        4'd7 : b2_5k <= writedata[4:0];
                        4'd8 : b5k <= writedata[4:0];
                        4'd9 : b8k <= writedata[4:0];
                        4'd10 : b14k <= writedata[4:0];
                        4'd11 : b20k <= writedata[4:0];
                endcase
        end else if (chipselect && read) begin
                case (address)
                        4'd0 : readdata <= {3'd0, b31};
                        4'd1 : readdata <= {3'd0, b72};
                        4'd2 : readdata <= {3'd0, b150};
                        4'd3 : readdata <= {3'd0, b250};
                        4'd4 : readdata <= {3'd0, b440};
                        4'd5 : readdata <= {3'd0, b630};
                        4'd6 : readdata <= {3'd0, b1k};
                        4'd7 : readdata <= {3'd0, b2_5k};
                        4'd8 : readdata <= {3'd0, b5k};
                        4'd9 : readdata <= {3'd0, b8k};
                        4'd10 : readdata <= {3'd0, b14k};
                        4'd11 : readdata <= {3'd0, b20k};
                endcase
        end
end

endmodule
--------------------------------------------------------------------------------------------
/* aud_to_fft */

parameter SAMPLES = 14'd8192;
parameter SAMPLES2 = 14'd4096;

module add_signed (
        input  signed [5:0] A,
        input  signed [5:0] B,
        output signed [6:0] sum
);

assign sum = A + B + 6'sd13;

endmodule

module audio_to_fft (
   input  aud_clk,
        input  fft_clk,
        input  system_clk,
        input  reset,
        // If true, then we should read the data for our fft
        input  chan_req,
```

```
        input  chan_end,
   input  [15:0] audio_input,
        output [15:0] audio_output,
        output [15:0] vga_dat,
        output vga_dowrite,
        output vga_select,
        output [1:0] vga_addr,
        output [31:0] readdata,
        input  [1:0] address,
        input  chipselect,
        input  read
);

// FIFO
wire wrreq;
wire rdreq;
wire [15:0] buf_cnt;
// The output of the fifo goes right to the FFT
wire [15:0] dc_out;
wire fifo_in_aclr;
dc_fifo       fifo (
        .wrclk (aud_clk),
        .wrreq (wrreq),
        .rdreq (rdreq),
        .data (audio_input),
        .rdclk (fft_clk),
        .rdusedw (buf_cnt),
        .q (dc_out),
        .aclr (fifo_in_aclr)
);
// End FIFO

// FFT for audio input
wire fft_in_can_accept_input;
wire fft_in_sisop;
wire fft_in_sieop;
wire fft_in_sivalid;
wire fft_in_sovalid;
wire fft_in_sosop;
wire fft_in_soeop;
wire [15:0] fft_in_soreal;
wire [15:0] fft_in_soimag;
wire source_can_accept_input;
wire [5:0] fft_in_exp;
reg  [5:0] fft_in_exp_reg;

fft_module fft_in (
        .clk (fft_clk),
        .reset_n (!reset),
        .sink_error (2'b0),
```

```
        .sink_ready (fft_in_can_accept_input),
        .sink_real (dc_out),
        .sink_imag (16'b0),
        .sink_sop (fft_in_sisop),
        .sink_valid (fft_in_sivalid),
        .sink_eop (fft_in_sieop),
        // We're always ready for action
        .source_ready (source_can_accept_input),
        .source_real (fft_in_soreal),
        .source_imag (fft_in_soimag),
        .source_sop (fft_in_sosop),
        .source_eop (fft_in_soeop),
        .source_valid (fft_in_sovalid),
        .source_exp (fft_in_exp),
        .inverse (1'b0)
);

// CPU FIFO
wire cpu_rdreq;
wire [15:0] cpu_cnt;
wire cpu_is_empty;
wire [31:0] cpu_q;
cpu_fifo fft_dat (
        .rdclk (system_clk),
        .wrclk (fft_clk),
        .data ({fft_in_soreal, fft_in_soimag}),
        .wrreq (fft_in_sovalid),
        .rdreq (cpu_rdreq),
        .rdusedw (cpu_cnt),
        .rdempty (cpu_is_empty),
        .q (cpu_q)
);

// FFT for equalizer to audio
wire [15:0] fft_out_sireal;
wire [15:0] fft_out_siimag;
wire fft_out_sivalid;
wire fft_out_sovalid;
wire fft_out_can_accept_input;
wire fft_out_sisop;
wire fft_out_sieop;
wire fft_out_sosop;
wire fft_out_soeop;
wire [15:0] fft_out_soreal;
wire [5:0] fft_out_exp;
reg  [5:0] fft_out_exp_reg;
fft_module fft_out (
        .clk (fft_clk),
        .reset_n (!reset),
        .sink_error (2'b0),
```

```
        .sink_ready (fft_out_can_accept_input),
        .sink_real (fft_out_sireal),
        .sink_imag (fft_out_siimag),
        .sink_sop (fft_out_sisop),
        .sink_valid (fft_out_sivalid),
        .sink_eop (fft_out_sieop),
        // We're always ready for action
        .source_ready (1'b1),
        .source_real (fft_out_soreal),
        // We can't have imaginary audio
        .source_imag (GND),
        .source_sop (fft_out_sosop),
        .source_eop (fft_out_soeop),
        .source_valid (fft_out_sovalid),
        .source_exp (fft_out_exp),
        .inverse (1'b1)
);
// End FFT

equalizer equal (
        .inreal (fft_in_soreal),
        .inimag (fft_in_soimag),
        .outreal (fft_out_sireal),
        .outimag (fft_out_siimag),
        .insop (fft_in_sosop),
        .outsop (fft_out_sisop),
        .insovalid (fft_in_sovalid),
        .outsivalid (fft_out_sivalid),
        .out_out_can_accept_input (fft_out_can_accept_input),
        .in_out_can_accept_input (source_can_accept_input),
   .clk (fft_clk)
);

// FIFO for writing to audio out
wire fifo_out_wrreq;
wire fifo_out_rdreq;
wire [15:0] fifo_out_out;
wire [15:0] fifo_out_cnt;
wire fifo_out_aclr;
dc_fifo_sm     fifo_out_real (
        .wrclk (fft_clk),
        .wrreq (fifo_out_wrreq),
        .rdreq (fifo_out_rdreq),
        .data (fft_out_soreal),
        .rdclk (aud_clk),
        .rdusedw (fifo_out_cnt),
        .q (fifo_out_out),
        .aclr (fifo_out_aclr)
);
```

```
// Shifting function
function [15:0] shift_num;
	input signed [6:0] exponent;
	input [15:0] num;

	begin
		case (exponent)
			-7'sd15  : shift_num = {num[15],15'b0};
			-7'sd14  : shift_num = {num[15],num[0],14'b0};
			-7'sd13  : shift_num = {num[15],num[1:0],13'b0};
			-7'sd12  : shift_num = {num[15],num[2:0],12'b0};
			-7'sd11  : shift_num = {num[15],num[3:0],11'b0};
			-7'sd10  : shift_num = {num[15],num[4:0],10'b0};
			-7'sd9   : shift_num = {num[15],num[5:0],9'b0};
			-7'sd8   : shift_num = {num[15],num[6:0],8'b0};
			-7'sd7   : shift_num = {num[15],num[7:0],7'b0};
			-7'sd6   : shift_num = {num[15],num[8:0],6'b0};
			-7'sd5   : shift_num = {num[15],num[9:0],5'b0};
			-7'sd4   : shift_num = {num[15],num[10:0],4'b0};
			-7'sd3   : shift_num = {num[15],num[11:0],3'b0};
			-7'sd2   : shift_num = {num[15],num[12:0],2'b0};
			-7'sd1   : shift_num = {num[15],num[13:0],1'b0};
			7'sd0    : shift_num = num;
			7'sd1    : shift_num = {num[15],num[15:1]};
			7'sd2    : shift_num = {num[15],num[15],num[15:2]};
			7'sd3    : shift_num = {num[15],num[15],num[15],num[15:3]};
			7'sd4    : shift_num = {num[15],num[15],num[15],num[15],num[15:4]};
			7'sd5    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15:5]};
			7'sd6    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15:6]};
			7'sd7    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:7]};
			7'sd8    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:8]};
			7'sd9    : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:9]};
			7'sd10   : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15:1
0]};
			7'sd11   : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],
num[15:11]};
			7'sd12   : shift_num =
{num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],num[15],
num[15],num[15:12]};
			// It's too low, just zero it
			default  : shift_num = {16'b0};
		endcase
	end
```

```
endfunction

// Set the exponent
reg signed [6:0] total_exponent;
add_signed adder (
	.A (fft_out_exp_reg),
	.B (fft_in_exp_reg),
	.sum (total_exponent)
);

// Always write to the input fifo
reg in_fifo_counter;
always @(posedge aud_clk) begin
//	if (reset) begin
//		fifo_in_aclr <= 1'b1;
//		in_fifo_counter <= 1'b1;
//		wrreq <= 1'b0;
//	end else if (in_fifo_counter) begin
//		fifo_in_aclr <= 1'b0;
//		in_fifo_counter <= 1'b0;
	if (chan_end) begin
		wrreq <= 1'b1;
	end else begin
		wrreq <= 1'b0;
	end
end

// Always read from the output fifo
always @(posedge aud_clk) begin
	// The fifo is not empty
	if (fifo_out_cnt >= 12'b0 && chan_req) begin
		audio_output <= shift_num(total_exponent, fifo_out_out);
		// Acknowledge that we've read this data
		fifo_out_rdreq <= 1'b1;
	end else begin
		audio_output <= 16'b0;
		fifo_out_rdreq <= 1'b0;
	end
end

// Logic for reading from the output FFT
reg [15:0] outpos;

always @(posedge fft_clk) begin
	if (fft_out_sovalid && fft_out_soeop) begin
		outpos <= 16'd0;
		fifo_out_wrreq <= 1'b1;
		fft_out_exp_reg <= fft_out_exp;
	end else if (fft_out_sovalid) begin
		fifo_out_wrreq <= 1'b1;
```

```
                outpos <= outpos + 16'b1;
        end else if (!fft_out_sovalid) begin
                fifo_out_wrreq <= 1'b0;
        end
end

reg [15:0] pos;
reg [15:0] fill;
reg fill_fft;

// Logic for filling the input FFT
always @(posedge fft_clk) begin
        // Check to see if the fifo is "full". If it is, begin reading
        if (buf_cnt >= SAMPLES && !fill_fft) begin
                fill_fft <= 1'b1;
        end
        // Writing to the FFT
        else if (fill_fft && fft_in_can_accept_input) begin
                // Indicate to the FFT that we can send data and send the first sample
                if (pos == 16'd0) begin
                        // Acknowledge that we've read this word
                        rdreq <= 1'b1;
                        // Assert that the data on the bus is valid
                        fft_in_sivalid <= 1'b1;
                        // Indicate that this is the first packet
                        fft_in_sisop <= 1'b1;
                        pos <= pos + 1'b1;
                // Deassert SOP on the next cycle
                end else if (pos == 16'd1) begin
                        // Assert that the data on the bus is valid
                        fft_in_sivalid <= 1'b1;
                        // Acknowledge that we've read this word
                        rdreq <= 1'b1;
                        // Deassert SOP
                        fft_in_sisop <= 1'b0;
                        pos <= pos + 1'b1;
                // For the last sample, assert eop
                end else if (pos == SAMPLES - 15'b1) begin
                        fft_in_sieop <= 1'b1;
                        // Acknowledge that we've read this word
                        rdreq <= 1'b1;
                        // Assert that the data on the bus is valid
                        fft_in_sivalid <= 1'b1;
                        pos <= pos + 1'b1;
                // Deassert all other signals
                end else if (pos == SAMPLES) begin
                        fft_in_sieop <= 1'b0;
                        fft_in_sivalid <= 1'b0;
                        rdreq <= 1'b0;
                        pos <= 16'b0;
```

```
                // For all the other positions
                end else begin
                        // Assert that the data on the bus is valid
                        fft_in_sivalid <= 1'b1;
                        // Acknowledge that we've read this word
                        rdreq <= 1'b1;
                        pos <= pos + 16'b1;
                        vga_dowrite <= 1'b1;
                        vga_select <= 1'b1;
                        if (fill % 4 == 2'd3) begin
                                vga_dat <= audio_input;
                        end else begin
                                vga_dat <= fill;
                        end
                        vga_addr <= fill % 4;
                        fill <= fill + 16'd1;
                end
        // Don't acknowledge that we've read anything yet
        end else if (!fft_in_can_accept_input)
                rdreq <= 1'b0;
        else begin
                // Assert that the data on the bus is not valid
                fft_in_sivalid <= 1'b0;
                rdreq <= 1'b0;
                vga_dowrite <= 1'b1;
                vga_select <= 1'b1;
        end
        // Record the exponent of the fft input block
        if (fft_in_sosop) begin
                fft_in_exp_reg <= fft_in_exp;
        end
end

reg reset_cycle;

initial begin
        reset_cycle <= 1'b0;
        pos <= 16'd0;
        fill <= 16'd0;
        fill_fft <= 1'd0;
        outpos <= 16'd0;
        wrreq <= 1'b0;
        rdreq <= 1'b0;
        fifo_out_wrreq <= 1'b0;
        fifo_out_rdreq <= 1'b0;
        fft_in_sivalid <= 1'b0;
        fft_in_sisop <= 1'b0;
        fft_in_sieop <= 1'b0;
        fifo_in_aclr <= 1'b0;
        fifo_out_aclr <= 1'b0;
```

```
	in_fifo_counter <= 1'b0;
end

// Bus interface logic
reg cpu_queue_empty_last_read = 1'b0;

always_ff @(posedge system_clk) begin
	if (reset) begin
		// I'll figure out what to do later
	end else if (chipselect && read) begin
		case (address)
			2'd0 : begin
				// Check to see if the queue is empty
				cpu_queue_empty_last_read <= cpu_is_empty;
				readdata <= cpu_q;
				cpu_rdreq <= 1'b1;
			end
			2'd1 : begin
				readdata <= {31'b0, cpu_queue_empty_last_read};
				cpu_rdreq <= 1'b0;
			end
			default : cpu_rdreq <= 1'b0;
		endcase
	end else begin
		cpu_rdreq <= 1'b0;
	end
end

endmodule
-------------------------------------------------------------------------------------------
/* freq_spec_slave.sv */

module frec_spec_slave (
 input logic       clk50, reset,
 input logic [8:0]  b31, b72, b150, b250, b440, b630, b1k, b2_5k, b5k, b8k, b14k, b20k,
 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic             VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *              _______________              ________
 * ____________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _____________________      _____________
 * |____|       VGA_HS          |____|
 */
```

```
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

   logic [10:0]                          hcount; // Horizontal counter
                                             // Hcount[10:1] indicates pixel column
(0-639)
   logic                         endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else               hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   // Vertical counter
   logic [9:0]                           vcount;
   logic                         endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
       if (endOfField)   vcount <= 0;
       else              vcount <= vcount + 10'd 1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280           01 1110 0000  480
   // 110 0011 1111  1599           10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                        !( vcount[9] | (vcount[8:5] == 4'b1111) );
```

```
   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk50    __|  |__|  |__|
    *
    *            _____       __
    * hcount[0]__|     |_____|
    */
   assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

       // Assign ranges for them
   parameter X31 = 10'd10, X72 = 10'd62, X150 = 10'd114, X250 = 10'd166, X440 = 10'd218,
X630 = 10'd270, X1K = 10'd322, X2_5K = 10'd374, X5K = 10'd426, X8K = 10'd478, X14K =
10'd530, X20K = 10'd582;

       // Those offsets were computed with the width:
       parameter WIDTH = 10'd48;
       // The spacing is 4 pixels

       // Registers to hold the blue color
       reg [17:0] blue31, blue72, blue150, blue250, blue440, blue630, blue1k, blue2_5k,
blue5k, blue8k, blue14k, blue20k;
       always_comb begin
              blue31 = ({9'b0, b31} * 18'd255) / 18'd480;
              blue72 = ({9'b0, b72} * 18'd255) / 18'd480;
              blue150 = ({9'b0, b150} * 18'd255) / 18'd480;
              blue250 = ({9'b0, b250} * 18'd255) / 18'd480;
              blue440 = ({9'b0, b440} * 18'd255) / 18'd480;
              blue630 = ({9'b0, b630} * 18'd255) / 18'd480;
              blue1k = ({9'b0, b1k} * 18'd255) / 18'd480;
              blue2_5k = ({9'b0, b2_5k} * 18'd255) / 18'd480;
              blue5k = ({9'b0, b5k} * 18'd255) / 18'd480;
              blue8k = ({9'b0, b8k} * 18'd255) / 18'd480;
              blue14k = ({9'b0, b14k} * 18'd255) / 18'd480;
              blue20k = ({9'b0, b20k} * 18'd255) / 18'd480;
       end

       // Color each bar
       always_comb begin
              {VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20};
              if (hcount[10:1] >= X31 && hcount[10:1] < X31 + WIDTH) begin
                     if (vcount[8:0] >= b31 && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                            {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, blue31[7:0]};
                     end
              end
              if (hcount[10:1] >= X72 && hcount[10:1] < X72 + WIDTH) begin
                     if (vcount[8:0] >= b72 && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                            {VGA_R, VGA_G, VGA_B} = {8'd21, 8'h0, blue72[7:0]};
```

```
                end
            end
            if (hcount[10:1] >= X150 && hcount[10:1] < X150 + WIDTH) begin
                if (vcount[8:0] >= b150 && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd43, 8'h0, blue150[7:0]};
                end
            end
            if (hcount[10:1] >= X250 && hcount[10:1] < X250 + WIDTH) begin
                if (vcount[8:0] >= b250 && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd64, 8'h0, blue250[7:0]};
                end
            end
            if (hcount[10:1] >= X440 && hcount[10:1] < X440 + WIDTH) begin
                if (vcount[8:0] >= b440 && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd85, 8'h0, blue440[7:0]};
                end
            end
            if (hcount[10:1] >= X630 && hcount[10:1] < X630 + WIDTH) begin
                if (vcount[8:0] >= b630 && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd106, 8'h0, blue630[7:0]};
                end
            end
            if (hcount[10:1] >= X1K && hcount[10:1] < X1K + WIDTH) begin
                if (vcount[8:0] >= b1k && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd128, 8'h0, blue1k[7:0]};
                end
            end
            if (hcount[10:1] >= X2_5K && hcount[10:1] < X2_5K + WIDTH) begin
                if (vcount[8:0] >= b2_5k && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd149, 8'h0, blue2_5k[7:0]};
                end
            end
            if (hcount[10:1] >= X5K && hcount[10:1] < X5K + WIDTH) begin
                if (vcount[8:0] >= b5k && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd170, 8'h0, blue5k[7:0]};
                end
            end
            if (hcount[10:1] >= X8K && hcount[10:1] < X8K + WIDTH) begin
                if (vcount[8:0] >= b8k && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                    {VGA_R, VGA_G, VGA_B} = {8'd191, 8'h0, blue8k[7:0]};
                end
            end
```

```
              if (hcount[10:1] >= X14K && hcount[10:1] < X14K + WIDTH) begin
                     if (vcount[8:0] >= b14k && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                            {VGA_R, VGA_G, VGA_B} = {8'd212, 8'h0, blue14k[7:0]};
                     end
              end
              if (hcount[10:1] >= X20K && hcount[10:1] < X20K + WIDTH) begin
                     if (vcount[8:0] >= b20k && !(vcount[8:0] % 48 <= vcount[8:0] % 2))
begin
                            {VGA_R, VGA_G, VGA_B} = {8'd234, 8'h0, blue20k[7:0]};
                     end
              end
       end
endmodule
--------------------------------------------------------------------------------
/* freq_spec.sv */

module frec_spec(input logic        clk,
              input logic      reset,
              input logic [15:0]  writedata,
              input logic      write,
              input            chipselect,
              input logic [3:0]  address,

              output logic [7:0] VGA_R, VGA_G, VGA_B,
              output logic     VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
              output logic     VGA_SYNC_n);

       // Keep a bank of heights for the bars
       reg [8:0] b31 = 9'd240, b72 = 9'd240, b150 = 9'd240, b250 = 9'd240, b440 = 9'd240,
b630 = 9'd240, b1k = 9'd240, b2_5k = 9'd240, b5k = 9'd240, b8k = 9'd240, b14k = 9'd240,
b20k = 9'd240;

   frec_spec_slave slave(.clk50(clk), .*);

       always_ff @(posedge clk)
              if (reset) begin
                     b31    <= 9'd240;
                     b72    <= 9'd240;
                     b150   <= 9'd240;
                     b250   <= 9'd240;
                     b440   <= 9'd240;
                     b630   <= 9'd240;
                     b1k    <= 9'd240;
                     b2_5k <= 9'd240;
                     b5k    <= 9'd240;
                     b8k    <= 9'd240;
                     b14k   <= 9'd240;
                     b20k   <= 9'd240;
              end else if (chipselect && write)
```

```
                    case (address)
                        4'd0   :      b31    <= writedata[8:0];
                        4'd1   :      b72    <= writedata[8:0];
                        4'd2   :      b150   <= writedata[8:0];
                        4'd3   :      b250   <= writedata[8:0];
                        4'd4   :      b440   <= writedata[8:0];
                        4'd5   :      b630   <= writedata[8:0];
                        4'd6   :      b1k    <= writedata[8:0];
                        4'd7   :      b2_5k <= writedata[8:0];
                        4'd8   :      b5k    <= writedata[8:0];
                        4'd9   :      b8k    <= writedata[8:0];
                        4'd10  :      b14k   <= writedata[8:0];
                        4'd11  :      b20k   <= writedata[8:0];
                    endcase

endmodule
--------------------------------------------------------------------------------
/* audio_codec.sv //heavily modified from Howie's code */

module audio_codec (
    input  clk,
    input  reset,
    output [1:0]  sample_end,
    output [1:0]  sample_req,
    input  [15:0] audio_output_l,
        input  [15:0] audio_output_r,
    output [15:0] audio_input_l,
    output [15:0] audio_input_r,
    // 1 - left, 0 - right
    input  [1:0] channel_sel,

    output AUD_ADCLRCK,
    input AUD_ADCDAT,
    output AUD_DACLRCK,
    output AUD_DACDAT,
    output AUD_BCLK,
        input [3:0] control
);

reg [7:0] lrck_divider;
reg [1:0] bclk_divider;

reg [15:0] shift_out;
reg [15:0] shift_in;
reg [15:0] shift_in_right;

wire lrck = !lrck_divider[7];

assign AUD_ADCLRCK = lrck;
assign AUD_DACLRCK = lrck;
```

```
assign AUD_BCLK = bclk_divider[1];
assign AUD_DACDAT = shift_out[15];

always @(posedge clk) begin
    if (reset) begin
        lrck_divider <= 8'hff;
        bclk_divider <= 2'b11;
    end else begin
        lrck_divider <= lrck_divider + 1'b1;
        bclk_divider <= bclk_divider + 1'b1;
    end
end

assign sample_end[1] = (lrck_divider == 8'h40);
assign sample_end[0] = (lrck_divider == 8'hc0);
assign audio_input_l = shift_in;
assign audio_input_r = shift_in_right;
assign sample_req[1] = (lrck_divider == 8'hfe);
assign sample_req[0] = (lrck_divider == 8'h7e);

wire clr_lrck = (lrck_divider == 8'h7f);
wire set_lrck = (lrck_divider == 8'hff);
// high right after bclk is set
wire set_bclk = (bclk_divider == 2'b10 && !lrck_divider[6]);
// high right before bclk is cleared
wire clr_bclk = (bclk_divider == 2'b11 && !lrck_divider[6]);

// For producing a sine wave
reg [15:0] romdata [0:99];
reg [3:0]  modIdx = 4'b0;
reg [6:0]  index = 7'd0;
parameter SINE     = 0;
parameter BYPASS   = 2;

always @(posedge clk) begin
    if (reset) begin
        shift_out <= 16'h0;
        shift_in <= 16'h0;
                shift_in_right <= 16'h0;
    end else if (set_lrck || clr_lrck) begin
        // check if current channel is selected
        if (channel_sel[set_lrck]) begin
                            if (control[BYPASS])
                                    shift_out <= shift_in;
                            else
                                    shift_out <= audio_output_l;
        // repeat the sample from the other channel if not
        end else begin
                            if (control[BYPASS])
                                    shift_out <= shift_in_right;
```

```
                            else
                                    shift_out <= audio_output_r;
                        end
    end else if (set_bclk == 1) begin
        // only read in if channel is selected
        if (channel_sel[lrck]) begin
                            if (control[SINE]) begin
                                    shift_in <= romdata[index];
                            end else begin
                                    shift_in <= {shift_in[14:0], AUD_ADCDAT};
                            end
                end else begin
                            if (control[SINE]) begin
                                    shift_in_right <= romdata[index];
                                    modIdx <= modIdx + 4'b1;
                                    // Increase the index
                                    if (modIdx == 4'b1111) begin
                                            modIdx <= 4'b0;
                                            index <= index + 7'b1;
                                            if (index == 7'd99) begin
                                                    index <= 7'd00;
                                            end
                                    end
                            end else begin
                                    shift_in_right <= {shift_in_right[14:0], AUD_ADCDAT};
                            end
                end
    end else if (clr_bclk == 1) begin
        shift_out <= {shift_out[14:0], 1'b0};
    end
end

initial begin
    romdata[0] = 16'h0000;
    romdata[1] = 16'h0805;
    romdata[2] = 16'h1002;
    romdata[3] = 16'h17ee;
    romdata[4] = 16'h1fc3;
    romdata[5] = 16'h2777;
    romdata[6] = 16'h2f04;
    romdata[7] = 16'h3662;
    romdata[8] = 16'h3d89;
    romdata[9] = 16'h4472;
    romdata[10] = 16'h4b16;
    romdata[11] = 16'h516f;
    romdata[12] = 16'h5776;
    romdata[13] = 16'h5d25;
    romdata[14] = 16'h6276;
    romdata[15] = 16'h6764;
    romdata[16] = 16'h6bea;
```

```
romdata[17] = 16'h7004;
romdata[18] = 16'h73ad;
romdata[19] = 16'h76e1;
romdata[20] = 16'h799e;
romdata[21] = 16'h7be1;
romdata[22] = 16'h7da7;
romdata[23] = 16'h7eef;
romdata[24] = 16'h7fb7;
romdata[25] = 16'h7fff;
romdata[26] = 16'h7fc6;
romdata[27] = 16'h7f0c;
romdata[28] = 16'h7dd3;
romdata[29] = 16'h7c1b;
romdata[30] = 16'h79e6;
romdata[31] = 16'h7737;
romdata[32] = 16'h7410;
romdata[33] = 16'h7074;
romdata[34] = 16'h6c67;
romdata[35] = 16'h67ed;
romdata[36] = 16'h630a;
romdata[37] = 16'h5dc4;
romdata[38] = 16'h5820;
romdata[39] = 16'h5222;
romdata[40] = 16'h4bd3;
romdata[41] = 16'h4537;
romdata[42] = 16'h3e55;
romdata[43] = 16'h3735;
romdata[44] = 16'h2fdd;
romdata[45] = 16'h2855;
romdata[46] = 16'h20a5;
romdata[47] = 16'h18d3;
romdata[48] = 16'h10e9;
romdata[49] = 16'h08ee;
romdata[50] = 16'h00e9;
romdata[51] = 16'hf8e4;
romdata[52] = 16'hf0e6;
romdata[53] = 16'he8f7;
romdata[54] = 16'he120;
romdata[55] = 16'hd967;
romdata[56] = 16'hd1d5;
romdata[57] = 16'hca72;
romdata[58] = 16'hc344;
romdata[59] = 16'hbc54;
romdata[60] = 16'hb5a7;
romdata[61] = 16'haf46;
romdata[62] = 16'ha935;
romdata[63] = 16'ha37c;
romdata[64] = 16'h9e20;
romdata[65] = 16'h9926;
romdata[66] = 16'h9494;
```

```
    romdata[67] = 16'h906e;
    romdata[68] = 16'h8cb8;
    romdata[69] = 16'h8976;
    romdata[70] = 16'h86ab;
    romdata[71] = 16'h845a;
    romdata[72] = 16'h8286;
    romdata[73] = 16'h8130;
    romdata[74] = 16'h8059;
    romdata[75] = 16'h8003;
    romdata[76] = 16'h802d;
    romdata[77] = 16'h80d8;
    romdata[78] = 16'h8203;
    romdata[79] = 16'h83ad;
    romdata[80] = 16'h85d3;
    romdata[81] = 16'h8875;
    romdata[82] = 16'h8b8f;
    romdata[83] = 16'h8f1d;
    romdata[84] = 16'h931e;
    romdata[85] = 16'h978c;
    romdata[86] = 16'h9c63;
    romdata[87] = 16'ha19e;
    romdata[88] = 16'ha738;
    romdata[89] = 16'had2b;
    romdata[90] = 16'hb372;
    romdata[91] = 16'hba05;
    romdata[92] = 16'hc0df;
    romdata[93] = 16'hc7f9;
    romdata[94] = 16'hcf4b;
    romdata[95] = 16'hd6ce;
    romdata[96] = 16'hde7a;
    romdata[97] = 16'he648;
    romdata[98] = 16'hee30;
    romdata[99] = 16'hf629;
end

endmodule
-------------------------------------------------------------------------------------------
/* audio_effects.sv */

module audio_effects (
    input  clk,
    input  [1:0] sample_end,
    input  [1:0] sample_req,
    output [15:0] audio_output_l,
    output [15:0] audio_output_r,
    input  [15:0] audio_input_l,
    input  [15:0] audio_input_r,
    input  [3:0]  control
         // If chan is true, then the channel is the right channel. Otherwise, it is the
left
```

```
        // input chan
);

reg [15:0] dat_l;
reg [15:0] dat_r;
reg [15:0] last_sample_l;
reg [15:0] last_sample_r;

initial begin
       dat_l <= 16'd0;
       dat_r <= 16'd0;
       last_sample_l <= 16'd0;
       last_sample_r <= 16'd0;
end

assign audio_output_l = dat_l;
assign audio_output_r = dat_r;

parameter FEEDBACK = 1;

always @(posedge clk) begin
       if (sample_end[1]) begin
              last_sample_l <= audio_input_l;
       end else if (sample_end[0]) begin
              last_sample_r <= audio_input_r;
       end

       if (sample_req[1]) begin
              if (control[FEEDBACK]) begin
                     dat_l <= last_sample_l;
              end else begin
                     dat_l <= 16'd0;
              end
       end else if (sample_req[0]) begin
              if (control[FEEDBACK]) begin
                     dat_r <= last_sample_r;
              end else begin
                     dat_r <= 16'd0;
              end
       end
end

endmodule
--------------------------------------------------------------------------------
/* equalizer.c */

//takes in 8k samples from fft_driver, equalizes them, passes them back to fft_driver

#include <stdio.h>
#include <unistd.h>
```

```
#include <stdint.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "equalizer_driver.h"

#define EQUALIZER_WRITE_DIGIT _IOW(EQUALIZER_MAGIC, 1, u32 *)

#define DRIVER_NAME "equalizer_driver"
#define SAMPLENUM 8
#define SAMPLEBYTES SAMPLENUM*2


void write_db(send_info* send)
{
    int fd = open("/dev/freq_spec", O_RDWR);
    if (ioctl(fd, EQUALIZER_DRIVER_WRITE_DIGIT, send) == -1)
        printf("EQUALIZER_DRIVER_WRITE_DIGIT failed: %s\n",
            strerror(errno));
    else {
        if (status)
            puts("EQUALIZER_DRIVER_WRITE_DIGIT is not set");
        else
            puts("EQUALIZER_DRIVER_WRITE_DIGIT is set");
    }
}

int main()
{
    int freq,db;
    printf("Please type in the frequency you want to change (0 to 11):\n");
    scanf("%d",&freq);
    printf("Please type in the decibal value you would like to change it to (1 to
25):\n");
    scanf("%d",&db);

    int addr, db_value;
    addr = (int) freq;
    db_value = (int) db;

    struct send_info sendi;
    sendi = (send_info *) malloc(SAMPLEBYTES);
    sendi.addr = addr;
    sendi.db = db_value;

    write_db(sendi);
```

```
    printf("Equalizer Userspace program terminating\n");
    return 0;
}
-------------------------------------------------------------------------------------------
/* equalizer_driver.c */

/*
 * Device driver for Equalizer
 * A Platform device implemented using the misc subsystem
 * Stephen A. Edwards
 * Columbia University
 * edited by ma2799 and ss3912
 *
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
 #include "equalizer_driver.h"

#define DRIVER_NAME "equalizer"
#define SAMPLENUM 8
#define SAMPLEBYTES SAMPLENUM*2

/*
 * Information about our device
 */
struct equalizer_driver_dev {
       struct resource res; /* Resource: our registers */
       void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
 * read from array (user gives us) and write to memory address (0x0 to 0x65536)
 */
static void write_mem(send_info *send  )
{
       iowrite16(send->db, dev.virtbase + send->addr); //is __iomem compatable with u16
}
```

```
/*
 * Handle ioctl() calls from userspace:
 * Note extensive error checking of arguments
 */
static long equalizer_driver_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	send_info send;

	switch (cmd) {
	case EQUALIZER_DRIVER_WRITE_DIGIT:
		if (copy_from_user(&send, (send_info *) arg, sizeof(send_info)))
			return -EACCES;
		write_mem(&send); //write send
		break;

	default:
		return -EINVAL;
	}

	return 0;
}

/* The operations our device knows how to do */
static const struct file_operations equalizer_fops = {
	.owner		= THIS_MODULE,
	.unlocked_ioctl = equalizer_driver_ioctl,

};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice equalizer_misc_device = {
	.minor		= MISC_DYNAMIC_MINOR,
	.name		= DRIVER_NAME,
	.fops		= &equalizer_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init equalizer_probe(struct platform_device *pdev)
{
	int ret;

	/* Register ourselves as a misc device: creates /dev/equalizer */
	ret = misc_register(&equalizer_misc_device);

	/* Get the address of our registers from the device tree */
```

```
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }


        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&equalizer_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int equalizer_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&equalizer_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id equalizer_of_match[] = {
        { .compatible = "altr,equalizer" },
        {},
};
MODULE_DEVICE_TABLE(of, equalizer_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver equalizer_driver = {
```

```
	.driver		= {
		.name	= DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(equalizer_of_match),
	},
	.remove		= __exit_p(equalizer_remove),
};

/* Calball when the module is loaded: set things up */
static int __init equalizer_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&equalizer_driver, equalizer_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit equalizer_exit(void)
{
	platform_driver_unregister(&equalizer_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(equalizer_init);
module_exit(equalizer_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("MA2799, SS3912");
MODULE_DESCRIPTION("EQUALIZER");
--------------------------------------------------------------------------------
/* equalizer_driver.h */

#ifndef _EQUALIZER_DRIVER_H
#define _EQUALIZER_DRIVER_H

#include <linux/ioctl.h>

typedef struct{
	int addr;
	int db;
 }send_info;

 #define EQUALIZER_MAGIC 'q'

/* ioctls and their arguments */
#define EQUALIZER_DRIVER_WRITE_DIGIT _IOW(EQUALIZER_MAGIC, 1, u32 *)

#endif
```