Landscape Generator
Based on FFT 3D-Spectral Visualization

Yuxiao Qu (yq2381), Ning Xia (nx2173), Yucong Li (yl5363), Yimin Yang (yy3352)

Contents

1 Project Overview

This embedded system effectively integrates audio capture, real-time FFT processing, and interactive data visualization, showcasing the capabilities of modern embedded systems in handling complex tasks. The user-controlled visualization adds an extra layer of interactivity, making the system not only functional but also engaging. Our system captures audio from a microphone, processes it using Fast Fourier Transform (FFT), and visualizes the results as a dynamic waterfall waveform on a VGA display. The system also features button controls that allow users to adjust the visualization's shape, providing a customizable and interactive experience. This project demonstrates the integration of audio processing, real-time data visualization, and user interaction within a single embedded platform.

1.1 System Block Diagram

The complete system, from audio input to VGA output, is illustrated in the following system block diagram. The diagram details the connections between the primary functional units of the system. The left side of the diagram represents the audio processing hardware, while the right side depicts the control and visualization components. The Avalon Bus connects between the two in the center. We will discuss each module and the data flow in detail in the subsequent sections.

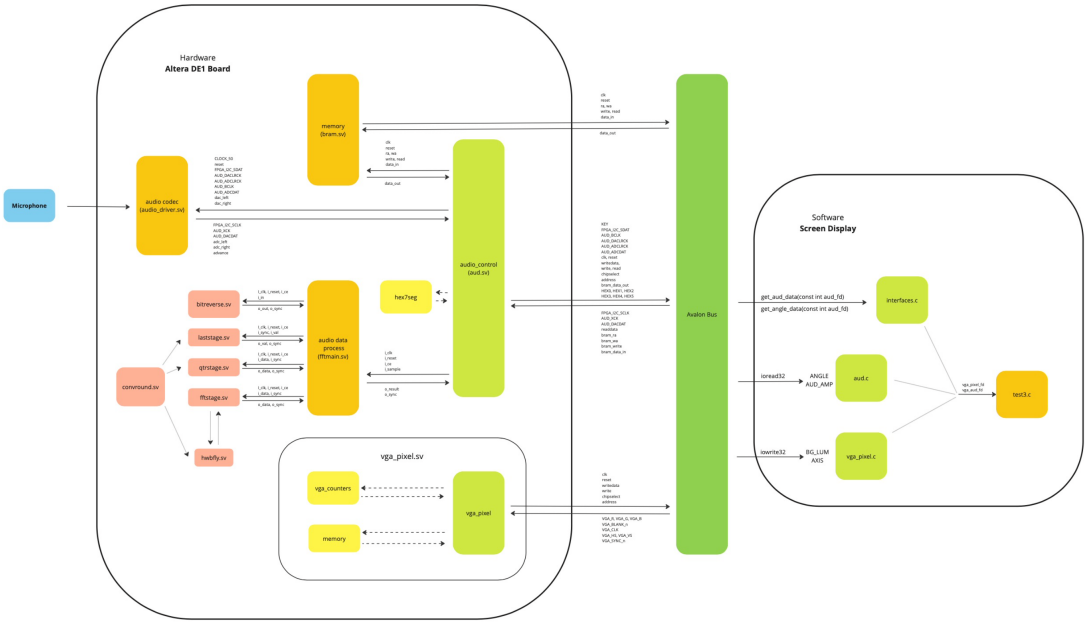

Figure 1: Diagram of the Embedded System.

1.2 Modules and Data Flow

Below, we outline each data transition as depicted in the system block diagram, explaining the data flow and transformations at each stage.

1.2.1 Microphone to Audio Processor

The system starts by capturing audio through an external microphone connected to the Altera DE1 board. The audio signal is fed into the audio codec module, which is responsible for digitizing the analog signal and sent it to the audio control module, preparing it for FFT calculation after combined the stereo sound data into mono sound data. This module uses an onboard analog to digital converter(ADC) to convert the analog audio input into a digital format.

1.2.2 FFT Calculation Process

The digitized audio data passed to the audio data process module. This module performs a Fast Fourier Transform on the audio data to convert the time-domain signal into its frequency-domain representation. The FFT results are crucial for visualizing the audio spectrum.

1.2.3 Button Control Interface

The audio control module interacts with the Button Control Interface, which allows users to adjust the shape of the visualization. By pressing buttons on board, users can change the angle of the waveform display from 30 to 60 degrees. The Button Control Interface updates the visualization parameters stored in ANGLE register accordingly, providing a customized viewing experience.

1.2.4 VGA Display

The output of the FFT Calculation module is sent to the avalon bus and stored in the AUD_AMP register. This module is responsible for rendering the FFT results as a dynamic waterfall waveform on a VGA display. The soft war process the audio data and the angle data to calculate the current waterfall plot and display the new real time waterfall on the screen accordingly. The visualization provides a real-time graphical representation of the audio spectrum, allowing users to observe frequency changes over time.

2 Hardware

2.1 Audio

The audio module of our FPGA project is designed to handle various aspects of audio processing and display, consisting of several key components:

2.1.1 CODEC Interface

The CODEC (Coder-Decoder) interface is crucial for converting analog audio signals to digital format and vice versa. This functionality is fundamental to bridging the gap between the analog world of audio signals and the digital processing capabilities of the FPGA.

To achieve this, we utilize an existing CODEC driver that manages the interface effectively. The module is configured to handle 24-bit output from the CODEC, specifically utilizing the adc_left_out[23:0] and adc_right_out[23:0] signals. This setup allows us to capture high-resolution audio data from both the left and right audio channels, ensuring detailed and accurate audio processing.

2.1.2 FFT Module

The Fast Fourier Transform (FFT) module is designed to analyze and transform audio signals from the time domain into the frequency domain. This transformation is essential for understanding the frequency components of the audio signal, which is a critical aspect of audio signal processing.

While a more detailed description of the FFT module will be provided later, it is important to note that this module is central to the frequency analysis and overall signal processing within the audio module. The FFT module enables the conversion and examination of the audio data in a way that reveals its underlying frequency characteristics.

2.1.3 Memory Buffer

The memory buffer plays a crucial role in storing the data processed by the FFT module for subsequent use. Its primary purpose is to retain the transformed audio data so that it can be accessed and utilized as needed for further processing or analysis.

The buffer is configured with a size of 512 x 42 bits. In this configuration, 42 bits represent the length of each FFT output, while 512 denotes the number of samples processed. This substantial buffer size ensures that there is ample storage capacity to handle the results of the FFT calculations efficiently, allowing for smooth data management and retrieval.

During the software part of the read, there is a readaddress that is incremented by 1 from 0, so that every time the software completes a 512-bit read, the readaddress is updated to 0, thus completing the cycle.

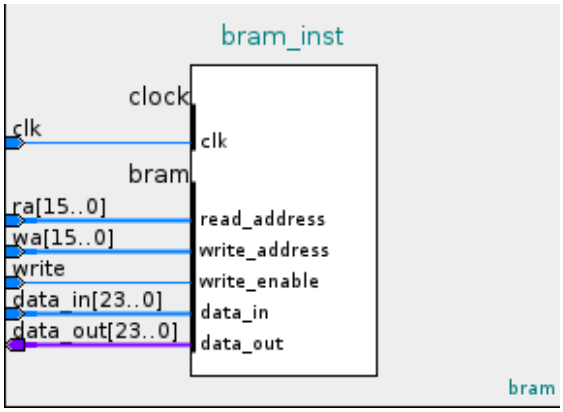

Figure 2: Block diagram of the memory Module

2.1.4 Button and Hex7Seg Display

This component is designed to offer user interaction and visualization capabilities directly on the FPGA board. It plays a vital role in providing a user-friendly interface for adjusting and viewing information.

The button functionality allows users to adjust the display angle, giving them control over how the information is presented. This interactive feature ensures that users can tailor the display to their preferences or requirements.

The Hex7Seg display complements this by visually representing the adjusted angle in a clear and readable format. It translates the angle information into a hexadecimal display, making it easy for users to interpret and understand the current settings.

The above is the overall design of the audio module, which as the main part of the hardware design of our group plays a key role in collecting the sound data, processing the sound data and transmitting the data results. The following is its block design:

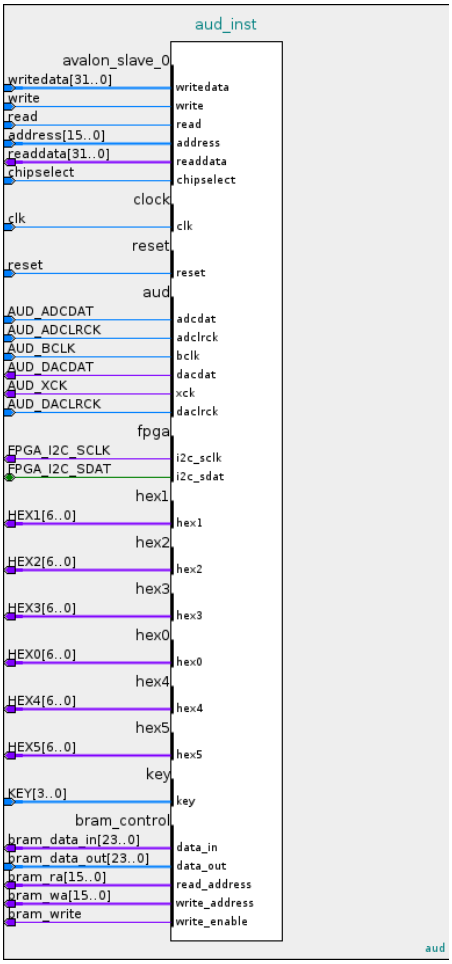

Figure 3: Block diagram of the Audio Module

2.2 FFT

2.2.1 DIF FFT

Fourier analysis converts a signal from its original domain to a representation in the frequency domain. It is an useful analytical in many applications in digital signal processing. In real signal processing tasks, the input function can be any quantity or signal that varies over time. Therefore, in order to better analyze the input signals, a common way is to taking measurements at regular intervals, so that the continuous signal can be sampled into a set of discrete values. Then the finite sequence of equally spaced samples of a function can be converted into a sequence of coefficients representing the signal's frequency components. That is the Discrete Fourier Transform(DFT). The DFT can be seen as a sampled version of the Fourier Transform. To simplify the computations of the DFT, the Fast Fourier Transform(FFT) algorithm was proposed to compute the DFT. The basic idea of FFT is based on decomposition and breaking the transform into smaller transforms and combining them to get the total transform.

According to their different decomposition, FFT algorithms can be classified into two different

types: Decimation in Time FFT (DIT FFT) and Decimation in Frequency FFT(DIF FFT). The DIT FFT algorithm splits the input sequence (time domain) into smaller sequences based on even and odd indices. The DIF FFT algorithm splits the output sequence (frequency domain) into smaller parts based on even and odd indices. In our project, we implemented a DIF FFT block in our data processing. The derivation of the DIF radix-2 FFT begins by splitting the DFT coefficients X[k] into even- and odd-indexed values. The DFT of the N-point signal x[n] can be written as:

$$X[k] = \sum_{\substack{n=0 \\ \text{even}}}^{N-1} x[n] \cdot W_N^{nk} + \sum_{\substack{n=0 \\ \text{odd}}}^{N-1} x[n] \cdot W_N^{nk} \tag{1}$$

, which can also be written as

$$X[k] = \sum_{n=0}^{\frac{N}{2}-1} x[2n] \cdot W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x[2n+1] \cdot W_N^{(2n+1)k} \tag{2}$$

The even values the given by:

$$X\left[2k\right] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{2kn} = \sum_{n=0}^{N-1} x[n] \cdot W_{\frac{N}{2}}^{kn} \tag{3}$$

Splitting this sum into the first N/2 and second N/2 terms gives:

$$\begin{aligned} X\left[2k\right] &= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_{\frac{N}{2}}^{kn} + \sum_{n=\frac{N}{2}}^{N-1} x[n] W_{\frac{N}{2}}^{kn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_{\frac{N}{2}}^{kn} + \sum_{n=\frac{N}{2}}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_{\frac{N}{2}}^{k(n+\frac{N}{2})} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_{\frac{N}{2}}^{kn} + \sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_{\frac{N}{2}}^{kn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(x[n] + x\left[n+\frac{N}{2}\right]\right) W_{\frac{N}{2}}^{kn} \\ &= \mathrm{DFT}_{\frac{N}{2}} \left\{ x[n] + x\left[n+\frac{N}{2}\right] \right\} \end{aligned} \tag{4}$$

That is, the even DFT values $X[2k]$ for $0 \leq 2k \leq N-1$ are given by the $\frac{N}{2}$-point DFT of the $\frac{N}{2}$-point signal $x[n] + x[n+\frac{N}{2}]$.

Similarly, the odd values can be given by:

$$
\begin{aligned}
X\left[2k+1\right] &= \sum_{n=0}^{\frac{N}{2}-1} x[n] W_N^n W_{\frac{N}{2}}^{kn} + \sum_{n=\frac{N}{2}}^{N-1} x[n] W_N^n W_{\frac{N}{2}}^{kn} \\
&= \sum_{n=0}^{\frac{N}{2}-1} \left(x[n] + x\left[n+\frac{N}{2}\right] \right) W_N^n W_{\frac{N}{2}}^{kn} \\
&= \mathrm{DFT}_{\frac{N}{2}} \left\{ W_N^n \left(x[n] - x\left[n+\frac{N}{2}\right] \right) \right\}
\end{aligned}
\tag{5}
$$

That is, the odd DFT values $X[2k+1]$ for $0 \leq 2k+1 \leq N-1$ are given by the $\frac{N}{2}$-point DFT of the $\frac{N}{2}$-point signal $W_N^n \left(x[n] - x[n+\frac{N}{2}] \right)$.

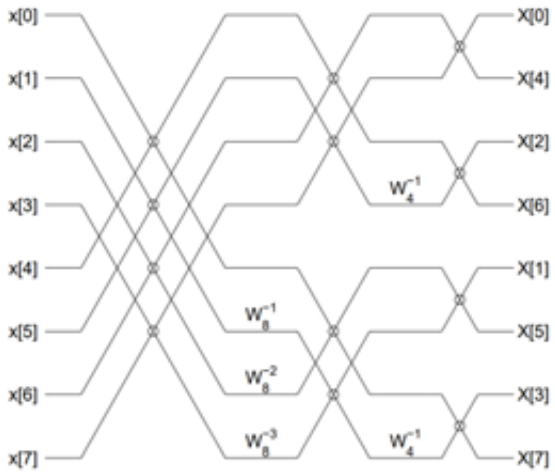

Figure 4: The illustration of the DIF radix-2 FFT

Based on this concept, the flowchart of DIF FFT for an 8-point DFT is illustrated in the Figure 4 above. N-point DFT is decomposed into N/2-point DFTs repeatedly. And the 2-point DFT consists of two parts: 1) Butterfly 2) The twiddle factors. Unlike the DIT FFT, the output of DIF FFT is bit-reversed order and the input is natural order. So the butterfly operation flow graph is showed in Figrue 5. There are 2 complex additions and 1 complex multiplication in each stage. Unlike the DIT-FFT, the multiplication is done after additions.

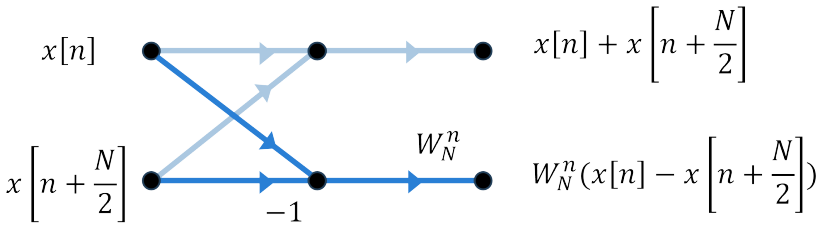

Figure 5: The flow graph for butterfly operation

2.2.2 Module Architecture

In this project, we applied a DIF FFT with a 32-bit input structure, as illustrated in Figure 6. The first 16 bits represent the real portion of the sample data, while the second 16 bits are the imaginary portion of the sample data. The real portion of the input is derived from the output data from audio CODEC, which produces a 24-bit signal. Then we take the first 16 bits as the audio real part input. The imaginary portion of the input data is padded with 16'b0.

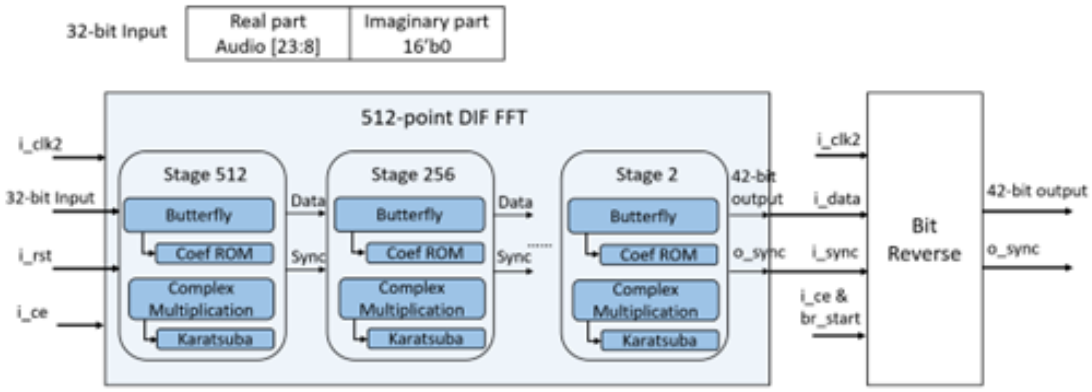

Figure 6: Diagram of the FFT Module

A. FFT Stage Processing

In each stage, the data undergoes a butterfly operation, which includes addition and subtraction. Then, the difference between the two samples is multiplied by the corresponding coefficient, i.e., the twiddle factor. The complex multiplication follows the Karatsuba algorithm, which can improve the efficiency of multiplication compared with traditional multiplication methods.

The output of the previous stage serves as the input of the next stage. Additionally, there is a sync signal to signify that the output of the stage is valid data. The *o_sync* signal is set to 1 (true) when the output data *o_data* corresponds to the start of a new block of data after a complete FFT stage has been processed.

B. Output Data Format and Bit-Reversal

The output of the FFT block is 42 bits. The first 21 bits represent the real portion, while the second 21 bits are the imaginary portion. In the last stage, the *o_sync* is set to 1 when the values of the butterfly then match the first sample out of the stage. It indicates the start of the valid data. Then the 42-bit output data from FFT enters the bit reverse stage. As is mentioned before, the output of DIF FFT is bit-reversed order and the input is natural order. Therefore, the output of the FFT operation should be bit-reversed to be reordered before being implemented in other modules.

2.2.3 Stage Architecture

A. Butterfly Operation

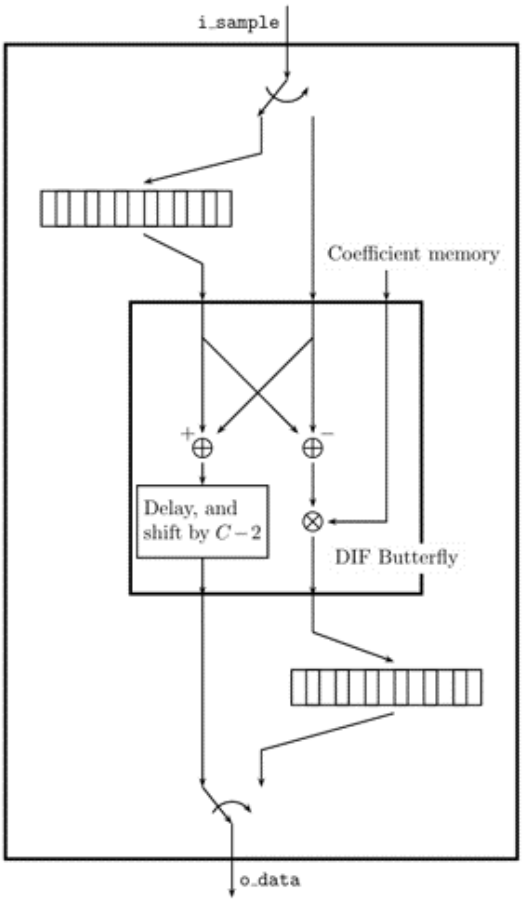

Figure 7: The block diagram for each stage in the FFT

The butterfly operation includes an addition and a subtraction. Based on the description of DIF FFT mentioned before, the operation in the FFT stage can be simplified as:

$$O_1 = A + B \tag{6}$$

$$O_2 = (A - B) \times C \tag{7}$$

in which A and B are the complex input and C is the complex twiddle factor. So the input of the stage undergoes an addition and subtraction first, just as showed in the code snipping in the Figure 8 below.

```
always @(posedge i_clk)
if (i_ce)
begin
    // One clock just latches the inputs
    r_left <= i_left;   // No change in # of bits
    r_right <= i_right;
    r_coef  <= i_coef;
    // Next clock adds/subtracts
    r_sum_r <= r_left_r + r_right_r; // Now IWIDTH+1 bits
    r_sum_i <= r_left_i + r_right_i;
    r_dif_r <= r_left_r - r_right_r;
    r_dif_i <= r_left_i - r_right_i;
    // Other inputs are simply delayed on second clock
    ir_coef_r <= r_coef[(2*CWIDTH-1):CWIDTH];
    ir_coef_i <= r_coef[(CWIDTH-1):0];
end
```

Figure 8: Code snipping of the addition and subtraction

B. Coefficient – Twiddle Factor

The twiddle factor in FFT is defined as

$$W_N^n = e^{-j\frac{2\pi n}{N}} \tag{8}$$

where N is the size of the FFT, n is the index of the twiddle factor for the specific stage. They are used to shift the frequency components in the FFT. As for each stage, the N is a known and fixed number, so the twiddle factors can easily be pre-computed and stored in lookup tables. This improves the efficiency of the FFT. Similar as the format of the input and output, the upper bits are the real portion of the twiddle factors and the lower bits are the imaginary part.

C. Fast Multiplication Algorithm

In each stage in this module, complex multiplication is performed where the difference between the two input samples is multiplied by the pre-stored twiddle factors. To execute the multiplication effectively, the Karatsuba algorithm is applied here.

Consider the multiplication of two complex numbers: $M = m_1 + jm_2$ and $N = n_1 + jn_2$. The standard calculation involves:

$$M \times N = m_1 \cdot n_1 - m_2 \cdot n_2 + j\,(m_1 \cdot n_2 + m_2 \cdot n_1)$$

So, there will be 4 multiplications. However, the Karatsuba algorithm can simplify this process. The basic idea of the algorithm is 'divide and conquer'. Three products $P1 = m_1 \cdot n_1$; $P2 = m_2 \cdot n_2$; $P3 = (m_1 + m_2) \cdot (n_1 + n_2)$ are required to calculate. The final multiplication result is then given by:

$$M \times N = (P1 - P2) + j \cdot (P3 - P1 - P2)$$

So, instead of 4 multiplications, only 3 multiplications are needed in this algorithm, which can save hardware resources.

Therefore, the multiplication here follows this concept. Specifically, the products are calculated as `rp_one`, `rp_two`, and `rp_three` in the code below in the Figure 9:

$$P_1 = (a_1 - b_1) \cdot c_1$$

$$P_2 = (a_2 - b_2) \cdot c_2$$

$$P_3 = [(a_1 - b_1) + (a_2 - b_2)] \cdot (c_1 + c_2)$$

```
always @(posedge i_clk)
if (i_ce)
begin
    // Second clock, pipeline = 1
    p1c_in <= ir_coef_r;  //c1
    p2c_in <= ir_coef_i;  //c2
    p1d_in <= r_dif_r;    //a1-b1
    p2d_in <= r_dif_i;    //a2-b2
    p3c_in <= ir_coef_i + ir_coef_r; // c1+c2
    p3d_in <= r_dif_r + r_dif_i;  //a1-b1+a2-b2
end
// }}}

// Perform our multiplies

always @(posedge i_clk)
if (i_ce)
begin
    // Third clock, pipeline = 3
    //   As desired, each of these lines infers a DSP48
    rp_one   <= p1c_in * p1d_in;
    rp_two   <= p2c_in * p2d_in;
    rp_three <= p3c_in * p3d_in;
end
```

Figure 9: Code snipping for multiplication based on Karatsuba algorithm

2.2.4 FFT Module Simulation

In order to check the correctness of the 512-point FFT block, we use a testbench where a series of audio input and the FFT block are simulated on an EDA platform. The simulation waveform results are checked with the mathematical FFT output. The simulation output values are scaled down by a factor of 16 compared to the mathematical FFT results. Figure 10 shows the input in the testbench. Figure 11 shows the mathematical referenced FFT calculation results with the same input. Figure 12 shows the output waveform from the simulation.

```
//Test stimulus
initial begin

    // Test Case 1
    i_sample = {16'd200, 16'd0}; #200
    i_sample = {16'd100, 16'd0}; #400
    i_sample = {16'd475, 16'd0}; #424
```

Figure 10: Testbench for audio input simulation

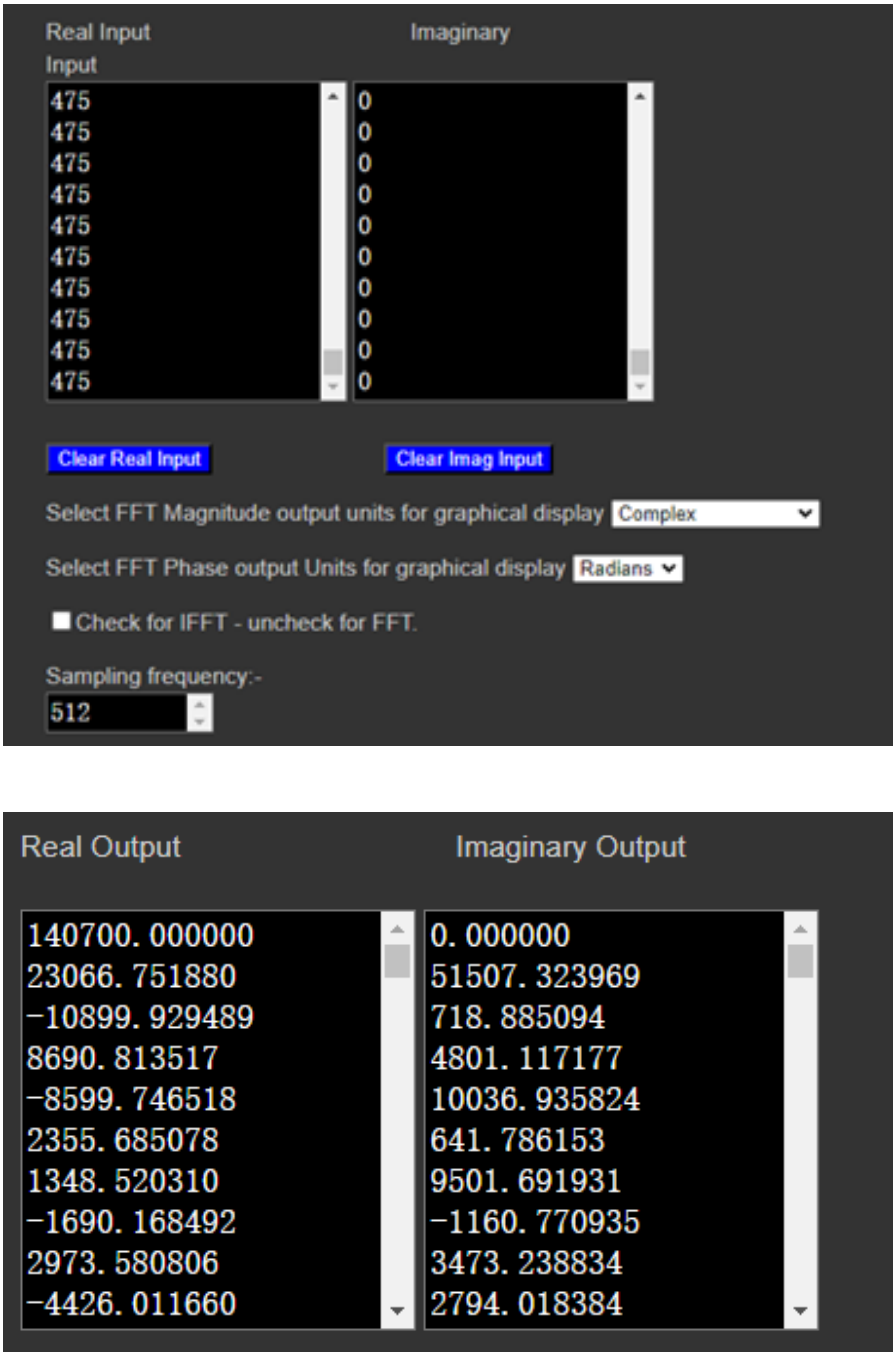

Figure 11: The mathematical input and output of FFT

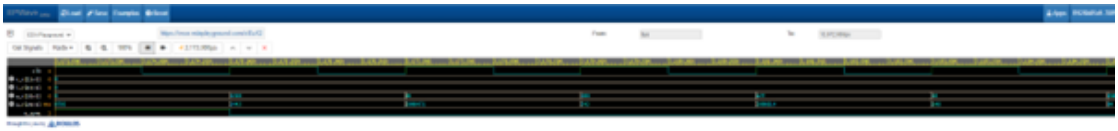

Figure 12: The simulation results of the FFT block

2.2.5 Bit Reverse Stage

The bit reversal in this module is achieved with a memory, as illustraed in the Figure 13. For this project, the output of the FFT calculation is 512 42-bit samples. So the size of the memory used to store these data is 29. The `wraddr` keeps track of the current write address in the memory. The `rdaddr` is used to read the data from memory in bit-reversed order. The `rdaddr` (read address) is the bit-reversed version of `wraddr` (write address). On each clock cycle where *i_ce* is high, data *i_in* (the raw output from the FFT) is written to the memory at the current `wraddr`. After the writing, data is read from the memory at the address specified by `rdaddr` and assigned to *o_out*. When the data has been processed, the *o_sync* is asserted to 1 to signal that the data is valid and corresponds to the first sample out of the stage.

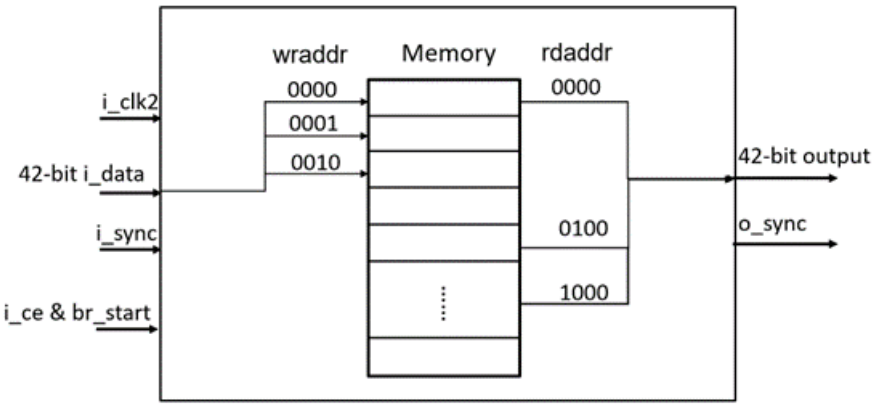

Figure 13: The block diagram for the bit reversal

2.2.6 Square Root Calculator

The square root operation that needs to be used when integrating the real and imaginary parts after the fft output for the effective value calculation. In this function we use the bit by bit to determine the answer to effectively avoid the square operation, only the use of bitwise operations and signed numbers can be added and subtracted to get the result.

The bitwise square root algorithm is a hardware-efficient method for calculating the square root of a number using bit shifts and comparisons. This technique involves initializing with the input number and iterating from the most significant bit (MSB) to the least significant bit (LSB) to progressively determine each bit of the square root. The process begins by shifting the input number left and setting a divisor aligned with the MSB. For each bit position, the algorithm tests whether setting the corresponding bit in the result keeps the squared value less than or equal to the input number. If successful, it updates the result and subtracts the squared value from the input number. This bitwise approach relies on simple bit shifts and arithmetic operations,

making it highly suitable for efficient hardware implementation, especially in FPGA or ASIC designs where resource optimization and speed are crucial.

Input: X=01111001 (decimal 121)

Step	A	X	T	Q	Description
	00000000	01111001	00000000	0000	Starting values.
1	00000001	11100100			Left shift X by two places into A.
			00000000		Set T = A - {Q,01}: 01 - 01.
				0000	Left shift Q.
	00000000			0001	Is T≥0? Yes. Set A=T and Q[0]=1.
2	00000011	10010000			Left shift X by two places into A.
			11111100		Set T = A - {Q,01}: 11 - 101.
				0010	Left shift Q.
					Is T≥0? No. Move to next step.
3	00001110	01000000			Left shift X by two places into A.
			00000101		Set T = A - {Q,01}: 1110 - 1001
				0100	Left shift Q.
	00000101			0101	Is T≥0? Yes. Set A=T and Q[0]=1.
4	00010101	00000000			Left shift X by two places into A.
			00000000		Set T = A - {Q,01}: 10101 - 10101.
				1010	Left shift Q.
	00000000			1011	Is T≥0? Yes. Set A=T and Q[0]=1.

Figure 14: Example of square root algorithm

2.3 VGA Display

In the VGA section of our design, we have carefully considered the need to process and display a full image to capture the intricate details and complexities of our output results. To achieve this, we must read and handle all pixel points of the image, as using only a limited amount of data would be inadequate for accurately reflecting the rich features of our display. Therefore, we have opted for a memory configuration that accommodates a resolution of 640 x 400 pixels, with each pixel represented by an 8-bit grayscale value. This configuration provides ample storage and enables us to effectively read and render a grayscale image of this size, ensuring that the visual representation is both detailed and precise.The visual effect is as follows:

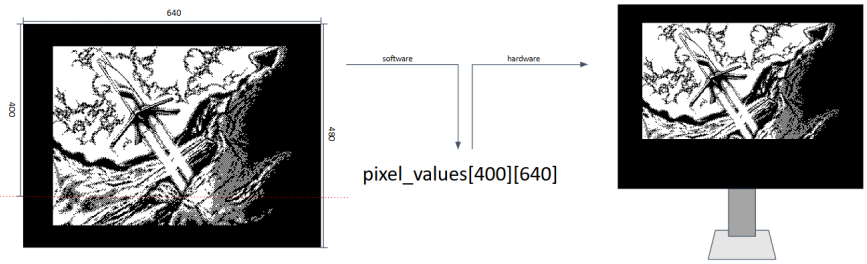

Figure 15: VGA display demonstration

To further enhance visual performance and maintain a high refresh rate, which is crucial for a smooth and dynamic display, we have employed an advanced data handling technique that reads four pixels simultaneously. This method significantly boosts the speed of data transfer between the hardware and software components of our system. By reading multiple pixels in parallel, we reduce the time required for data access and update, which translates into a smoother and more fluid screen refresh. This efficient management of memory access and data exchange not only accelerates performance but also improves the overall user experience by delivering a more responsive and visually appealing display.

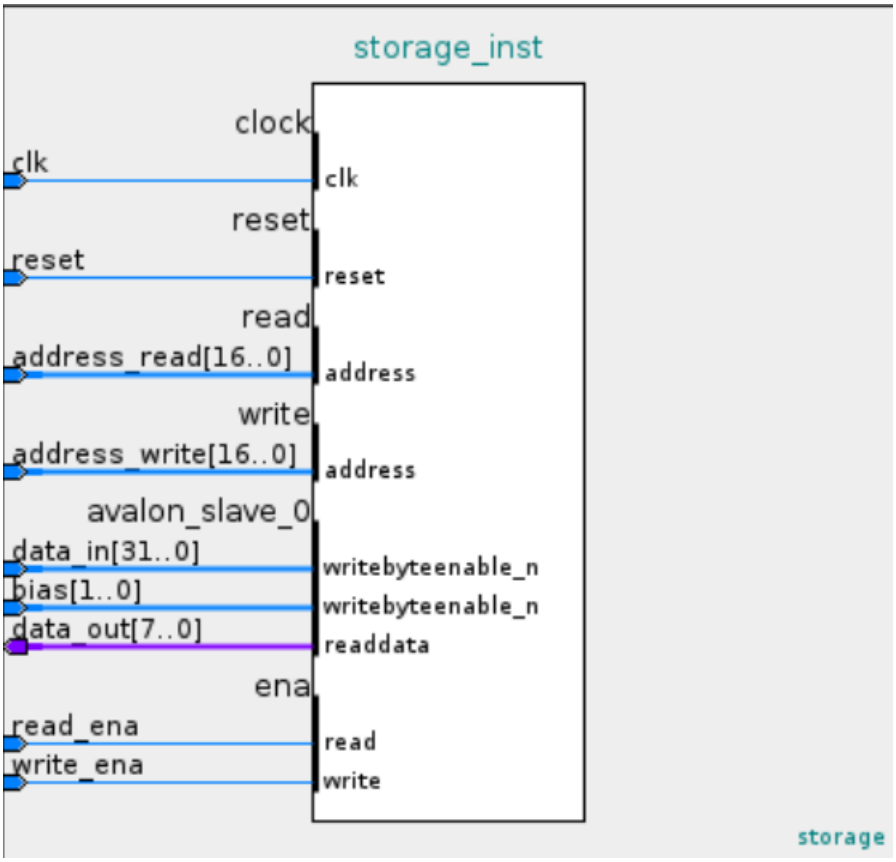

Figure 16: The block diagram for the VGA

3 Hardware-Software Interface

At the core of this design are memory-mapped registers, which serve as direct communication channels between the software and the underlying hardware. By mapping specific memory addresses to hardware functions, the software can read and write data to these registers, effectively controlling hardware behavior and monitoring its status.

		Bits																														
	Address	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
vga_pixel	0	8-bit pixel value								8-bit pixel value								8-bit pixel value								8-bit pixel value							write 4 pixel color data
	1	16-bit x-axis																16-bit y-axis																write pixel axis data
Aud	2																																	read angle
	3	32-bit aud data																																read hardware aud data

Figure 17: Register Map

3.1 *vga_pixel*

For the VGA display, the design includes two key registers: one dedicated to storing pixel color values and another for pixel positioning. This segmentation allows the software to precisely

control both the color and placement of each pixel on the screen, which is crucial for generating accurate and dynamic visualizations of the audio data. The software sends pixel colors and positions to vga hardware by writing them to registers.

3.1.1 Registers for `vga_pixel`

- **Address 0**: This register is divided into four 8-bit sections, each representing the pixel values for a VGA display. These values correspond to the intensity or color of the pixels displayed on the screen. We store pixel values in a packet of 4 for efficiency.
- **Address 1**: This register is used for positioning on the VGA display. The upper 16 bits (bits 31-16) represent the x-axis, and the lower 16 bits (bits 15-0) represent the y-axis. This allows the software to specify a pixel's position on the screen.

3.1.2 Functions

- `write_background(vga_pixel_color_t *background)`: Writes the luminance value to the background color register and updates the `background` field in the `vga_pixel_dev` structure.
- `write_pixel(vga_pixel_axis_t *position)`: Writes the pixel position to the axis register and updates the `position` field in the `vga_pixel_dev` structure.

3.1.3 Ioctl Handling (`vga_pixel_ioctl`)

This function handles ioctl commands from user-space applications:

- `VGA_PIXEL_WRITE_BACKGROUND`: Copies data from user space, writes it to the background register, and updates the device structure.
- `VGA_PIXEL_READ_BACKGROUND`: Copies the current background color data back to user space.
- `VGA_PIXEL_WRITE_POSITION`: Copies data from user space, writes it to the position register, and updates the device structure.
- `VGA_PIXEL_READ_POSITION`: Copies the current position data back to user space.

3.2 aud

In the audio subsystem, the interface is designed to handle 32-bit audio data, which the software reads a dedicated register. This data is then processed by the software to calculate waveforms based on processed audio data and updated angle data.

3.2.1 Registers for aud

- **Address 2**: This register holds the angle data which determines the rotation of waveforms.
- **Address 3**: This register stores the 32bit audio data processed by fft hardware module.

3.2.2 Functions

- `read_memory(aud_mem_t *memory)`: Reads data from the `AUD_AMP` register and stores it in the provided `memory` structure.
- `read_angle(aud_mem_t *memory)`: Reads data from the `ANGLE` register and stores it in the provided `memory` structure.

3.2.3 Ioctl Handling (`aud_ioctl`)

This function handles ioctl commands from user-space applications:

- `ANGLE_READ_DATA`: Copies the current angle data back to user space.
- `AUD_READ_DATA`: Copies the current audio data back to user space.

4 Software

4.1 Data Capture

- **Audio Data Acquisition**

  The program continuously captures audio data from the hardware using the `get_aud_data(aud_fd)` function. This data represents audio signals that are being processed or captured by the hardware.

- **Angle Data Acquisition**

  It also retrieves an angle value using `get_angle_data(aud_fd)`, which is used to adjust the visualization of the waveform. This angle likely represents a rotation or tilt of the waveform display.

4.2 Waveform Generation

- **Frames and CUR Arrays** The program uses two 2D arrays (`Frames` and `CUR`) to store and process the waveform data. `Frames` holds the audio data captured over time, while `CUR` is used to store the current state of the waveform to be displayed on the screen.
- **Waveform Calculation** The `f(int** CUR, int** Frames, int ang)` function calculates the waveform's position on the screen based on the audio data and the angle. It performs transformations on the data to create a visual representation of the waveform that takes the specified angle into account.

4.3 Displaying the Waveform

- **Pixel and Background Setting** The program clears the screen and then iteratively sets the pixel positions and their respective colors based on the calculated waveform data. The pixel color intensity is derived from the amplitude of the audio signal, and the position is determined by the processed coordinates.
- **Screen Refresh** The screen is refreshed in a loop, where new audio data is continuously captured, processed, and displayed. This loop runs until a predetermined number of frames have been processed (`FRAMENUMBERS`). The old waveforms are moved left and decommissioned, then new waveforms are incorporated on the right.

5 Further Thoughts

In this project, our hardware development journey began with exploring how to handle audio input on an FPGA using a CODEC interface. This foundational step provided us with a deep understanding of audio signal processing and interfacing. We then advanced to implementing the Fast Fourier Transform (FFT) module, which allowed us to delve into complex operations like the butterfly algorithm and state machines, crucial for accelerating hardware operations. These experiences were complemented by utilizing input/output interfaces from previous lab sessions, which deepened our grasp of practical data handling and memory management.

We also gained valuable insights into managing data transfer and memory operations, particularly how interrupts trigger data read and write processes. Transitioning from using pre-existing code to designing our own data transfer mechanisms marked a significant advancement in our skills. The project not only enhanced our technical expertise but also improved our teamwork, as we moved from experiencing slow and error-prone screen refreshes to achieving faster and more accurate performance. The collective efforts of the team were crucial to this progress.

However, there are areas for improvement. The FFT results, while correct in simulation, exhibit inaccuracies in the hardware implementation, possibly due to issues with sampling time or delays between software samples. Additionally, optimizing the software code for image generation could enhance the visual transition between consecutive waveforms, resulting in smoother and more aesthetically pleasing displays. Furthermore, increasing the efficiency of data transfer between hardware and software could further boost the display frame rate, improving overall performance.

6 Contribution

In this project, Yuxiao Qu and Ning Xia focused on the design of the hardware-software interface and the software part.

Yucong Li and Yimin Yang were responsible for microphone configuration, audio processing, FFT operation, and VGA display in the hardware part.

7 Reference

[1] THE FAST FOURIER TRANSFORM (FFT). (n.d.-a). `https://eeweb.engineering.nyu.edu/iselesni/EL713/zoom/fft`

[2] Dan Gisselquist (2017). A Generic Piplined FFT Core Generator. `https://github.com/ZipCPU/dblclockfft/tree/master`

[3] Square-root-in-verilog, `https://projectf.io/posts/square-root-in-verilog/`

[4] Digital Signal Processing Principles, Algorithms and Applications by J.G. Proakis and D.G. Manolakis page-464

8 Appendix

8.1 Brief explanation for uploaded hardware file

A brief explanation of the hardware files:
audio_driver.sv is the hardware file for audio CODEC interface configuration.
aud.sv is one of the core hardware files that have multiple functions in it including audio data controlling and processing, instantiating the FFT block and square root calculation of the FFT results.
vga_pixel.sv is the file related to plotting data visualization figures pixel by pixel on the VGA.
fftmain.sv is the top level FFT file.
fftstage.sv calculates one FFT stage
hwbfly.sv implements a butterfly that uses the * operator for its multiply
Longbimpy.sv is the logic binary multiply.
Bimpy.sv multiplies a small set of bits together. It is a component of longbimpy
qtrstage.sv is the 4-pt stage of the FFT
laststage.sv is the 2-pt stage of the FFT
bitreverse.sv, the final step in the multiply, bit-reverses the outgoing data.

8.2 Code for Hardware

8.2.1 aud.sv

```
`include "global_variables.sv"
`include "./AudioCodecDrivers/audio_driver.sv"

//`define RAM_ADDR_BITS 5'd16
//`define RAM_WORDS 16'd48000

// 7-Seg dispaly for debugging

module hex7seg(input logic [3:0] a,
               output logic [6:0] y);
    always_comb
        case (a)       //     gfe_dcba
            4'h0:       y = 7'b100_0000;
            4'h1:       y = 7'b111_1001;
            4'h2:       y = 7'b010_0100;
            4'h3:       y = 7'b011_0000;
            4'h4:       y = 7'b001_1001;
            4'h5:       y = 7'b001_0010;
            4'h6:       y = 7'b000_0010;
            4'h7:       y = 7'b111_1000;
            4'h8:       y = 7'b000_0000;
            4'h9:       y = 7'b001_0000;
            4'hA:       y = 7'b000_1000;
            4'hB:       y = 7'b000_0011;
            4'hC:       y = 7'b100_0110;
            4'hD:       y = 7'b010_0001;
            4'hE:       y = 7'b000_0110;
```

```
            4'hF:       y = 7'b000_1110;
            default:    y = 7'b111_1111;
        endcase
endmodule

module audio_control(
        input logic [3:0]       KEY, // Pushbuttons; KEY[0] is rightmost
        // 7-segment LED displays; HEX0 is rightmost
        //output logic [6:0]     HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,

        //Audio pin assignments
        //Used because Professor Scott Hauck and Kyle Gagner
        output logic            FPGA_I2C_SCLK,
        inout                   FPGA_I2C_SDAT,
        output logic            AUD_XCK,
        input logic             AUD_ADCLRCK,
        input logic             AUD_DACLRCK,
        input logic             AUD_BCLK,
        input logic             AUD_ADCDAT,
        output logic            AUD_DACDAT,

        //Driver IO ports
        input logic             clk,
        input logic             reset,
        input logic [31:0]      writedata,
        input logic             write,
        input logic             read,
        input                   chipselect,
        input logic [15:0]      address,
        output logic [31:0]     readdata,

   output logic [6:0]      HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
        //Bram controls
        output logic [15:0]     bram_wa,
        output logic [15:0]     bram_ra,
        output logic            bram_write = 0,
        output logic [23:0]     bram_data_in,
        input logic [23:0]      bram_data_out

        );

    //Audio Controller
    reg [23:0]     dac_left_in;
    reg [23:0]     dac_right_in;
    logic [23:0]    adc_left_out;
    logic [23:0]    adc_right_out;

    reg [20:0]     sqr;
    // wire advance;

    //Device drivers from Altera modified by Professor Scott Hauck and Kyle Gagner in
        Verilog
    audio_driver aDriver(
```

```
    .CLOCK_50(clk),
    .reset(reset),
    .dac_left(dac_left_in),
    .dac_right(dac_right_in),
    .adc_left(adc_left_out),
    .adc_right(adc_right_out),
    .advance(advance),
    .FPGA_I2C_SCLK(FPGA_I2C_SCLK),
    .FPGA_I2C_SDAT(FPGA_I2C_SDAT),
    .AUD_XCK(AUD_XCK),
    .AUD_DACLRCK(AUD_DACLRCK),
    .AUD_ADCLRCK(AUD_ADCLRCK),
    .AUD_BCLK(AUD_BCLK),
    .AUD_ADCDAT(AUD_ADCDAT),
    .AUD_DACDAT(AUD_DACDAT)
    );

//Instantiate hex decoders
logic [23:0]  hexout_buffer;


reg [15:0] ra = 0,wa = 0;
reg [23:0] rd,wd;
reg       wena = 0,rena = 0;
logic     clk_2;

twoportbram Bram(
.clk(clk_2),
.reset(reset),
.ra(ra),
.wa(wa),
.write(wena),
.read(rena),
.data_in(wd),
.data_out(rd)
);

hex7seg h5( .a(angle[5:4]),.y(HEX5) ), // left digit
       h4( .a(angle[3:0]),.y(HEX4) ),
       h3( .a(0),.y(HEX3) ),
       h2( .a(0),.y(HEX2) ),
       h1( .a(0),.y(HEX1) ),
       h0( .a(0),.y(HEX0) );



//Convert stereo input to mono
logic [23:0]   adc_mono;
logic [23:0]   buffer;
logic [25:0]   CC = 26'b0;
logic [31:0]   CC2 = 32'b1;
logic [5:0]    angle = 6'd45;
//debounce
```

```
 always_ff @(posedge clk)begin
   logic ena0 = 1,ena1 = 1;
   logic [31:0] presstime0,presstime1;
CC2 <= CC2 + 1;
if (CC2 == 32'hffffffff) CC2 <= 1;
   if (KEY[0] == 0 && ena0 == 1) begin
       presstime0 <= CC2;
       if (angle < 6'd60) angle <= angle + 1;
   end
   if (CC2 < presstime0 + 1 || KEY[0] == 0) begin
       ena0 <= 0;
   end else begin
       ena0 <= 1;
       presstime0 <= 0;
   end
   if (KEY[1] == 0 && ena1 == 1) begin
       presstime1 <= CC2;
       if (angle > 6'd30) angle <= angle - 1;
   end
   if (CC2 < presstime1 + 1 || KEY[1] == 0) begin
       ena1 <= 0;
   end else begin
       ena1 <= 1;
       presstime1 <= 0;
   end
 end

 logic [3:0]    bram_input_ctrl;
 logic [23:0]   result_buffer;
 logic [23:0]   adc_out_buffer;
 logic          write_clk;
 logic      left_fft_ce;
 logic      left_fft_o_sync;
 logic [41:0]   o_result;
 logic[15:0]    inp;

 reg [23:0]     Ctst = 0;
 logic cena = 1;

 assign left_fft_ce = 1;
 assign clk_2 = CC[13];
 assign wd = {3'b0, sqr};

 fftmain left_fft(
   .i_clk(clk_2),
   .i_reset(reset),
   .i_ce(left_fft_ce),
   .i_sample({adc_mono[23:8], 16'b0}),
   .o_result(o_result),
   .o_sync(left_fft_o_sync));
 function [23:0] A;
        input [23:0] num; //declare input
```

```
  //intermediate signals.
       reg [23:0] absn;
       begin
          if (num[23] == 0)
       absn = {1'b0,num[23:1]};
     else
       absn = {1'b1,num[23:1]};
     A = absn;
  end
endfunction //end of Function
function [20:0] sqrt;
     input [20:0] num1; //declare input
     input [20:0] num2;

    //intermediate signals.
     reg [20:0] absn1,absn2;
     logic [41:0] n1 = 0,n2 = 0;
     reg [41:0] a;
     reg [23:0] q;
     reg [25:0] left,right,r;
  begin
     //do square
     if (num1[20] == 0)
      absn1 = num1;
    else
      absn1 = -num1;
     if (num2[20] == 0)
      absn2 = num2;
    else
      absn2 = -num2;
      for(integer i = 0; i < 20; i++)begin
      if (absn1[i])
        n1 += absn1 << i;
      if (absn2[i])
        n2 += absn2 << i;
    end
    //initialize all the variables.
    a = n1 + n2;
    q = 0;
    left = 0;  //input to adder/sub
    right = 0; //input to adder/sub
    r = 0; //remainder
    //run the calculations for 16 iterations.
     for(integer i = 0; i < 21; i++) begin
     right = {q,r[25],1'b1};
        left = {r[23:0],a[41:40]};
        a = {a[39:0],2'b00};  //left shift by 2 bits.
     if (r[25] == 1) //add if r is negative
           r = left + right;
        else   //subtract if r is positive
           r = left - right;
     q = {q[22:0],!r[25]};
    end
```

```
    sqrt = q;  //final assignment of output.
 end
 endfunction //end of Function

 always_comb begin
sqr = sqrt(o_result[20:0],o_result[41:21]);
    buffer = rd;
    //buffer = A(adc_mono);
 end

 //Determine when the driver is in the middle of pulling a sample
 //by default dont use the BRAM module
 logic          bram_writing = 0;
 logic          bram_reading = 0;
 logic [31:0]   driverReading = 31'd0;
 logic [15:0]   limit;

 always_ff @(posedge clk) begin
   adc_mono <= A(adc_right_out + adc_left_out);
//adc_mono <= adc_left_out;
CC += 1;
 end

 always_ff @(posedge clk_2) begin
if(wena) wa += 1;
 end

 always_ff @(posedge clk) begin : IOcalls
    // ioread recieved
    //adc_mono <= adc_left_out;
    if (chipselect && read) begin
        case (address)
            16'h0002: begin
                readdata[31:6] <= 0;
                readdata[5:0] <= angle;
            end
            16'h0003 : begin
      rena <= 1;
      wena <= 1;
      ra <= ra + 1;
                // return padded buffer
                if (buffer[23] == 1) begin
                    readdata[23:0] <= buffer[23:0];
                    readdata[31:24] <= 8'b11111111;
                end
                else if (buffer[23] == 0) begin
                    readdata[23:0] <= buffer[23:0];
                    readdata[31:24] <= 8'b00000000;
                end

            end
        endcase
    end
```

```
    end

    wire sampleBeingTaken;
    assign sampleBeingTaken = driverReading[0];

    //Map timer(Sample) counter output
    parameter readOutSize = 16'hffff;
    //Sample inputs/Audio passthrough

endmodule
```

8.2.2 *vga_pixel*.sv

```
    /*
 * Avalon memory-mapped peripheral that generates VGA
 * Columbia University
 */



module vga_pixel(input logic     clk,
          input logic    reset,
      input logic [31:0] writedata,
      input logic     write,
      input         chipselect,
      input logic [7:0] 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);

   logic [10:0]    hcount;
   logic [9:0]    vcount;
   logic [7:0]     background_r;
   logic [15:0]    h,v;
   logic      read_ena = 0, write_ena = 1;
   logic [16:0]   address_read, address_write;
   logic [31:0]   data_in;
   logic [7:0]    data_out;
   logic [7:0]   vgalum;
   logic [18:0]  temp_add;
   logic [7:0]   lum[3:0];
   logic [1:0]   bias;

   vga_counters counters(.clk50(clk), .*);


   memory mem(.*);
```

```
   always_ff @(posedge clk)begin
      if (chipselect && write)
         case (address)
         4'h0 : begin
            data_in <= writedata;
            write_ena <= 1;
         end
         4'h1 : begin
            h <= writedata[31:16];
            v <= writedata[15:0];
         end
        endcase
        address_write <= (v * 640 + h) >> 2;
        if (write_ena == 1) write_ena <= 0;
   end
   reg [31:0] temp;

   always_ff @(posedge clk)begin
   address_read = (vcount * 640 + hcount[10:1]) >> 2;
   bias = hcount[10:1] % 4;
   read_ena = (vcount * 640 + hcount[10:1] < 256000 && hcount[10:1] > 0 && hcount[10:1]
       < 632) ? 1 : 0;
   {VGA_R,VGA_G,VGA_B} <= {data_out, data_out, data_out};
   end

endmodule


module vga_counters(
 input logic      clk50, reset,
 output logic [10:0] hcount, // hcount[10:1] is pixel column
 output logic [9:0] vcount, // vcount[9:0] is pixel row
 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 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;

  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'b0; // For putting sync on the green signal; unused

  // 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: rising edge sensitive

endmodule
```

```
module memory(
   input logic clk, reset,
   input logic [16:0] address_read, address_write,
   input logic [31:0] data_in,
   input logic [1:0] bias,
   output logic [7:0] data_out,
   input logic read_ena, write_ena);

   reg [31:0] mem [65535:0];
   reg [31:0] A;

   always_ff@(posedge clk) begin
        if (write_ena) begin
                mem[address_write] = data_in;
               end
          if (read_ena) begin
             A = mem[address_read];
                data_out <=
                    {A[(bias<<3)+7],A[(bias<<3)+6],A[(bias<<3)+5],A[(bias<<3)+4],A[(bias<<3)+3],A[(bias<<3)
                end else
                data_out <= 0;


   end

endmodule
```

8.2.3 fftmain.sv

```
    ////////////////////////////////////////////////////////////////////////////////
//
// Filename: fftmain.v
// {{{
// Project: A General Purpose Pipelined FFT Implementation
//
// Purpose: This is the main module in the General Purpose FPGA FFT
//   implementation. As such, all other modules are subordinate
// to this one. This module accomplish a fixed size Complex FFT on
// 512 data points.
// The FFT is fully pipelined, and accepts as inputs one complex two's
// complement sample per clock.
//
// Parameters:
// i_clk The clock. All operations are synchronous with this clock.
// i_reset Synchronous reset, active high. Setting this line will
//      force the reset of all of the internals to this routine.
//      Further, following a reset, the o_sync line will go
//      high the same time the first output sample is valid.
// i_ce A clock enable line. If this line is set, this module
//      will accept one complex input value, and produce
//      one (possibly empty) complex output value.
```

```
// i_sample The complex input sample. This value is split
//      into two two's complement numbers, 16 bits each, with
//      the real portion in the high order bits, and the
//      imaginary portion taking the bottom 16 bits.
// o_result The output result, of the same format as i_sample,
//      only having 21 bits for each of the real and imaginary
//      components, leading to 42 bits total.
// o_sync  A one bit output indicating the first sample of the FFT frame.
//      It also indicates the first valid sample out of the FFT
//      on the first frame.
//
// Arguments: This file was computer generated using the following command
//   line:
//
//   % ./fftgen -v -d ../fft-sv -f 512 -1 -k 1 -p 100000 -n 16 -a ../bench/cpp/fftsize.h
//
// This core will use hardware accelerated multiplies (DSPs)
// for 7 of the 9 stages
//
// Creator: Dan Gisselquist, Ph.D.
//   Gisselquist Technology, LLC
//
////////////////////////////////////////////////////////////////////////////////
// }}}
// Copyright (C) 2015-2024, Gisselquist Technology, LLC
// {{{
// This file is part of the general purpose pipelined FFT project.
//
// The pipelined FFT project is free software (firmware): you can redistribute
// it and/or modify it under the terms of the GNU Lesser General Public License
// as published by the Free Software Foundation, either version 3 of the
// License, or (at your option) any later version.
//
// The pipelined FFT project is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTIBILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
// General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program. (It's in the $(ROOT)/doc directory. Run make
// with no target there if the PDF file isn't present.) If not, see
// <http://www.gnu.org/licenses/> for a copy.
// }}}
// License: LGPL, v3, as defined and found on www.gnu.org,
// {{{
//   http://www.gnu.org/licenses/lgpl.html
//
// }}}
////////////////////////////////////////////////////////////////////////////////
//
//
//
//
```

```
//
module fftmain #(
   parameter IWIDTH=16,
   parameter OWIDTH=21
   // LGWIDTH=9;
   //
)
(i_clk, i_reset, i_ce,
      i_sample, o_result, o_sync);
   // The bit-width of the input, IWIDTH, output, OWIDTH, and the log
   // of the FFT size. These are localparams, rather than parameters,
   // because once the core has been generated, they can no longer be
   // changed. (These values can be adjusted by running the core
   // generator again.) The reason is simply that these values have
   // been hardwired into the core at several places.

   input wire         i_clk, i_reset, i_ce;
   //
   input wire [(2*IWIDTH-1):0] i_sample;
   output  reg  [(2*OWIDTH-1):0] o_result;
   output  reg          o_sync;


   // Outputs of the FFT, ready for bit reversal.
   wire         br_sync;
   wire [(2*OWIDTH-1):0] br_result;


   // A hardware optimized FFT stage
   wire    w_s512;
   wire [33:0]  w_d512;
   fftstage #(
      // {{{
      .IWIDTH(IWIDTH),
      .CWIDTH(IWIDTH+4),
      .OWIDTH(17),
      .LGSPAN(8),
      .BFLYSHIFT(0),
      .OPT_HWMPY(1),
      .CKPCE(1),
      .COEFFILE("cmem_512.hex")
      // }}}
   ) stage_512(
      // {{{
      .i_clk(i_clk),
      .i_reset(i_reset),
      .i_ce(i_ce),
      .i_sync(!i_reset),
      .i_data(i_sample),
      .o_data(w_d512),
      .o_sync(w_s512)
      // }}}
   );
```

```
// A hardware optimized FFT stage
wire    w_s256;
wire [35:0]  w_d256;
fftstage #(
   // {{{
   .IWIDTH(17),
   .CWIDTH(21),
   .OWIDTH(18),
   .LGSPAN(7),
   .BFLYSHIFT(0),
   .OPT_HWMPY(1),
   .CKPCE(1),
   .COEFFILE("cmem_256.hex")
   // }}}
) stage_256(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s512),
   .i_data(w_d512),
   .o_data(w_d256),
   .o_sync(w_s256)
   // }}}
);

// A hardware optimized FFT stage
wire    w_s128;
wire [35:0]  w_d128;
fftstage #(
   // {{{
   .IWIDTH(18),
   .CWIDTH(22),
   .OWIDTH(18),
   .LGSPAN(6),
   .BFLYSHIFT(0),
   .OPT_HWMPY(1),
   .CKPCE(1),
   .COEFFILE("cmem_128.hex")
   // }}}
) stage_128(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s256),
   .i_data(w_d256),
   .o_data(w_d128),
   .o_sync(w_s128)
   // }}}
);
```

```
// A hardware optimized FFT stage
wire    w_s64;
wire [37:0]  w_d64;
fftstage #(
   // {{{
   .IWIDTH(18),
   .CWIDTH(22),
   .OWIDTH(19),
   .LGSPAN(5),
   .BFLYSHIFT(0),
   .OPT_HWMPY(1),
   .CKPCE(1),
   .COEFFILE("cmem_64.hex")
   // }}}
) stage_64(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s128),
   .i_data(w_d128),
   .o_data(w_d64),
   .o_sync(w_s64)
   // }}}
);

// A hardware optimized FFT stage
wire    w_s32;
wire [37:0]  w_d32;
fftstage #(
   // {{{
   .IWIDTH(19),
   .CWIDTH(23),
   .OWIDTH(19),
   .LGSPAN(4),
   .BFLYSHIFT(0),
   .OPT_HWMPY(1),
   .CKPCE(1),
   .COEFFILE("cmem_32.hex")
   // }}}
) stage_32(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s64),
   .i_data(w_d64),
   .o_data(w_d32),
   .o_sync(w_s32)
   // }}}
);
```

```
// A hardware optimized FFT stage
wire    w_s16;
wire [39:0]  w_d16;
fftstage #(
   // {{{
   .IWIDTH(19),
   .CWIDTH(23),
   .OWIDTH(20),
   .LGSPAN(3),
   .BFLYSHIFT(0),
   .OPT_HWMPY(1),
   .CKPCE(1),
   .COEFFILE("cmem_16.hex")
   // }}}
) stage_16(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s32),
   .i_data(w_d32),
   .o_data(w_d16),
   .o_sync(w_s16)
   // }}}
);

// A hardware optimized FFT stage
wire    w_s8;
wire [39:0]  w_d8;
fftstage #(
   // {{{
   .IWIDTH(20),
   .CWIDTH(24),
   .OWIDTH(20),
   .LGSPAN(2),
   .BFLYSHIFT(0),
   .OPT_HWMPY(1),
   .CKPCE(1),
   .COEFFILE("cmem_8.hex")
   // }}}
) stage_8(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s16),
   .i_data(w_d16),
   .o_data(w_d8),
   .o_sync(w_s8)
   // }}}
);

wire    w_s4;
```

```
wire [41:0]  w_d4;
qtrstage #(
   // {{{
   .IWIDTH(20),
   .OWIDTH(21),
   .LGWIDTH(9),
   .INVERSE(0),
   .SHIFT(0)
   // }}}
) stage_4(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s8),
   .i_data(w_d8),
   .o_data(w_d4),
   .o_sync(w_s4)
   // }}}
);
// verilator lint_off UNUSED
wire    w_s2;
// verilator lint_on UNUSED
wire [41:0]  w_d2;
laststage  #(
   // {{{
   .IWIDTH(21),
   .OWIDTH(21),
   .SHIFT(0)
   // }}}
) stage_2(
   // {{{
   .i_clk(i_clk),
   .i_reset(i_reset),
   .i_ce(i_ce),
   .i_sync(w_s4),
   .i_val(w_d4),
   .o_val(w_d2),
   .o_sync(w_s2)
   // }}}
);


wire  br_start;
reg   r_br_started;
initial r_br_started = 1'b0;
always @(posedge i_clk)
if (i_reset)
   r_br_started <= 1'b0;
else if (i_ce)
   r_br_started <= r_br_started || w_s2;
assign  br_start = r_br_started || w_s2;
```

```
  // Now for the bit-reversal stage.
  bitreverse #(
    // {{{
    .LGSIZE(9), .WIDTH(21)
    // }}}
  ) revstage (
    // {{{
    .i_clk(i_clk),
    .i_reset(i_reset),
    .i_ce(i_ce & br_start),
    .i_in(w_d2),
    .o_out(br_result),
    .o_sync(br_sync)
    // }}}
  );


  // Last clock: Register our outputs, we're done.
  initial o_sync = 1'b0;
  always @(posedge i_clk)
  if (i_reset)
    o_sync <= 1'b0;
  else if (i_ce)
    o_sync <= br_sync;

  always @(posedge i_clk)
  if (i_ce)
    o_result <= br_result;
  else
    o_result <= 0;


endmodule
```