The Front End Implementation of An Audio Beamformer for the RAW Processor

by

Judy L. Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master’s in Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2003

© Judy L. Chen, MMIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science
May 20, 2003

Certified by ............ .................................
Anant Agarwal
Associate Director, Laboratory of Computer Science
Thesis Supervisor

Accepted by ........................................... Arthur C. Smith
Chairman, Department Committee on Graduate Students
The Front End Implementation of An Audio Beamformer for
the RAW Processor

by

Judy L. Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2003, in partial fulfillment of the
requirements for the degree of
Master’s in Engineering in Electrical Engineering and Computer Science

Abstract

This thesis designed and implemented the front end of an audio beamformer for the
RAW processor, which involved handling and sorting digital audio data from an array
of microphones and formatting the audio data into data packets acceptable for the
RAW processor to perform its beamforming duties. The front end also handled the
beamformer controls and specifics that are given by the user. This was all design
and implemented in Verilog and synthesized with Synopsys. A series of tests were
performed to determine the quality of the data.

Thesis Supervisor: Anant Agarwal
Title: Associate Director, Laboratory of Computer Science
Acknowledgments

First, I would like to thank my family - Mom, Dad, Bernice, and Timmy - for their love and support during my five years here at MIT. Thank you for your encouraging words, funny comments, and prayers when times got stressful.

I would also like to thank Anant Agarwal for supervising this thesis. To Ken Steele for all his help and for answering all my questions. Also to Dave Wentzlaff, Jason Miller, Michael Taylor, Eugene Weinstein for helping me understand the RAW processor and not running away when I badger them with questions. Thanks to Prof. Anantha Chandrakasan for five years of academic advising and one term of 6.111 “fun.” Thanks to Ryan Lee for his computer and Latex help, and to Chia Hao La for helping me with Latex.

I would like to thank all the friends I’ve made during my time in Boston. Thanks you all for your encouragements, smiles, and care, especially through this last year. I would also like to thank Mikey, Shoey, Christine, Ryan, Amy, and Vivian for proofreading my thesis “for fun.” Thanks to Eric for finding last minute proofreading errors and Latex help. Thanks to Baoyi for moral support and an excuse to go to the Galleria. A special thanks to my roommate, Grace, for being your caring and wise self, and for adding silliness and fun to my last semester.

Most of all, I’d like to thank God for His faithfulness in humbling me and carrying me through MIT. Thank You for opening and closing doors, and letting me leave MIT dependent on You.
# Contents

1 Introduction 15
   1.1 Description of Beamforming 15
       1.1.1 Beamforming Algorithm 16
   1.2 Description of the RAW Processor 17
   1.3 Thesis Statement and Overview 18

2 Overall Design 19
   2.1 Microphone Array 19
   2.2 Front End Implementation 20

3 Network Interface 23
   3.1 Description of Network Interface 24
       3.1.1 Input Interface 24
       3.1.2 Output Interface 25
   3.2 Design and Implementation of Network Interface 25
       3.2.1 Network Input Interface 26
       3.2.2 Network Output Interface 27

4 Control Unit 29
   4.1 Description and Implementation of Control Register File 30
       4.1.1 Control Data 31
       4.1.2 Master Control Register: Register 0 31
       4.1.3 A/D Control Registers: Registers 01 to 07 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1.3</td>
<td>Verilog Source Code - Four Element FIFO</td>
<td>63</td>
</tr>
<tr>
<td>B.1.4</td>
<td>Verilog Source Code - Read-Val-in</td>
<td>66</td>
</tr>
<tr>
<td>B.1.5</td>
<td>Verilog Source Code - Pipelining</td>
<td>67</td>
</tr>
<tr>
<td>B.2</td>
<td>Network Output Interface</td>
<td>68</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Verilog Source Code - Output Interface</td>
<td>69</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Verilog Source Code - 255 Element FIFO</td>
<td>71</td>
</tr>
<tr>
<td>B.2.3</td>
<td>Verilog Source Code - Read-Val-out</td>
<td>73</td>
</tr>
<tr>
<td>B.2.4</td>
<td>Verilog Source Code - Pipelining</td>
<td>74</td>
</tr>
<tr>
<td>C</td>
<td>Verilog Source Code - Control Unit</td>
<td>75</td>
</tr>
<tr>
<td>C.1</td>
<td>Verilog Source Code - Control Register File</td>
<td>75</td>
</tr>
<tr>
<td>C.2</td>
<td>Verilog Source Code - SPI Control</td>
<td>77</td>
</tr>
<tr>
<td>C.3</td>
<td>Verilog Source Code - Control Ready</td>
<td>79</td>
</tr>
<tr>
<td>C.4</td>
<td>Verilog Source Code - Shift Register and Register Done</td>
<td>82</td>
</tr>
<tr>
<td>C.4.1</td>
<td>Shift Register</td>
<td>82</td>
</tr>
<tr>
<td>C.4.2</td>
<td>Done Register</td>
<td>84</td>
</tr>
<tr>
<td>D</td>
<td>Verilog Source Code - Formatting and Packaging Unit</td>
<td>85</td>
</tr>
<tr>
<td>D.1</td>
<td>Verilog Source Code - Data Count</td>
<td>85</td>
</tr>
<tr>
<td>D.2</td>
<td>Verilog Source Code - Data Shift Block</td>
<td>86</td>
</tr>
<tr>
<td>D.2.1</td>
<td>Verilog Source Code - 16-bit Shift Register</td>
<td>86</td>
</tr>
<tr>
<td>D.2.2</td>
<td>Verilog Source Code - Data Shift Block - Partial</td>
<td>88</td>
</tr>
<tr>
<td>D.2.3</td>
<td>Verilog Source Code - Multiplexer Selector</td>
<td>94</td>
</tr>
<tr>
<td>D.3</td>
<td>Verilog Source Code - Data Format Block</td>
<td>96</td>
</tr>
<tr>
<td>D.4</td>
<td>Verilog Source Code - Data Write Block</td>
<td>99</td>
</tr>
<tr>
<td>D.5</td>
<td>Verilog Source Code - SRAM</td>
<td>101</td>
</tr>
<tr>
<td>D.6</td>
<td>Verilog Source Code - Packaging Component</td>
<td>103</td>
</tr>
<tr>
<td>D.7</td>
<td>Verilog Source Code - Various Pipelining</td>
<td>110</td>
</tr>
<tr>
<td>D.7.1</td>
<td>Pipelining - Mux Select Enable</td>
<td>110</td>
</tr>
<tr>
<td>D.7.2</td>
<td>Pipelining - Microphone Enables</td>
<td>111</td>
</tr>
<tr>
<td>D.7.3</td>
<td>Pipelining - Data to Network Interface</td>
<td>111</td>
</tr>
</tbody>
</table>
List of Figures

1-1  Beamforming Example ......................................................... 16

2-1  Block Diagram of Beamformer .............................................. 19
2-2  Setup of Microphone Array .................................................. 21
2-3  Block Diagram of Beamformer Front End .................................. 22

3-1  Block Diagram of Network Interface ....................................... 23
3-2  Block Diagram of RAW handshake with Input Interface ............... 24
3-3  Block Diagram of RAW handshake with Output Interface ............. 25
3-4  Block Diagram of Network Input Interface ................................ 26
3-5  Block Diagram of Network Output Interface ............................. 28

4-1  The Top Level Block Diagram on the control data interaction between
     the A/D converters, FPGA, and RAW ........................................ 29
4-2  Block Diagram of Control Unit .............................................. 30
4-3  Format of the Control Data .................................................. 31
4-4  Format of the Master Control Register .................................... 32
4-5  Format of a A/D Control Register ........................................... 33
4-6  Format of Microphone Enable Example .................................... 34
4-7  Layout of the Control Register File ........................................ 35
4-8  Flowchart of Control Ready .................................................. 36
4-9  Layout of Control Data to the A/D Converter ............................ 37
4-10 Logic Diagram of Load(output) .............................................. 38
4-11 Logic Diagram of Done(output) ............................................. 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-12</td>
<td>Waveform of Done(output)</td>
<td>39</td>
</tr>
<tr>
<td>5-1</td>
<td>Block Diagram of Formatting and Packaging Unit</td>
<td>42</td>
</tr>
<tr>
<td>5-2</td>
<td>Waveforms of Digital Audio Data</td>
<td>42</td>
</tr>
<tr>
<td>5-3</td>
<td>Block Diagram of Data Shift</td>
<td>44</td>
</tr>
<tr>
<td>5-4</td>
<td>Waveform of Left Shifted Data</td>
<td>45</td>
</tr>
<tr>
<td>5-5</td>
<td>Waveform of Right Shifted Data</td>
<td>45</td>
</tr>
<tr>
<td>5-6</td>
<td>State Diagram of Data Format</td>
<td>47</td>
</tr>
<tr>
<td>5-7</td>
<td>Block Diagram of Data Write</td>
<td>48</td>
</tr>
<tr>
<td>5-8</td>
<td>Layout of Header Format Data Packet</td>
<td>51</td>
</tr>
<tr>
<td>5-9</td>
<td>Layout of Pure Audio Format Data Packet</td>
<td>51</td>
</tr>
<tr>
<td>5-10</td>
<td>State Diagram of Packaging FSM</td>
<td>53</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Corresponding FORMAT values to data formats ............... 46
Chapter 1

Introduction

Beamforming has had an important part in signal detection and filtering. Scientists and researchers have been able to improve upon signal detection and enhancement with various beamforming algorithms and applications.

An example of the advantages of beamforming applications is described by Gannot, Burshtein, and Weinstein [1]. With beamforming, they were able to enhance audio speech signals in a reverberating room. These kinds of enhancements are also seen in the real world. Luo, Yang, Pavlovic, and Nehorai used various beamforming algorithms to improve the quality of hearing aids [3].

1.1 Description of Beamforming

A beamformer is used to spatially detect a desired signal in the presence of noises and interferences [5]. A general beamforming architecture consists of a sensor array and a processor, which is actually called the beamformer.

Consider the sensor array and signal in Figure 1-1. As illustrated, the audio signal travels spatially and reaches the upper left microphone before the others. Also, the further away the microphone is from the location where the signal originated, the more intensity the signal loses. The signals from all the microphones are sent to the beamformer for processing. A beamformer can then determine and pick out the location of the desired signal based on the signal intensity and time delay between
the audio samples of all the microphones.

For this thesis, the beamformer is implemented with a microphone sensor array and the RAW processor[4] is used to perform the beamforming algorithms. Since we are implementing a digital beamformer, the microphone audio data is digitized by Analog/Digital (A/D) converters.

1.1.1 Beamforming Algorithm

The beamforming algorithm that is used is Delay-and-Sum. The basic idea of Delay-and-Sum is taking the signals from the microphones and delaying them appropriately so that the desired signal from each microphone, is added together and amplified[2].

The Delay-and-Sum algorithm used is described by Johnson and Dudgeon [2]. For an array of $M$ microphones, each microphones has an amplitude weight, $w_m$, which is dependent on the signal intensity and used to enhance the desired signal. As the signal travels to the microphone array, there are delays between when the closer and further microphones receive the desired signal. In the Delay-and-Sum beamforming algorithm, the outputs of microphones are appropriately delayed, $\Delta_m$ so that the desired signal from each microphones are aligned. Once these factors are determined, the Delay-and-Sum algorithm, shown in equation 1.1, is applied to the microphone outputs, $y_m$.
\[ z(t) \equiv \sum_{m=0}^{M-1} w_m y_m(t - \Delta_m) \]  

(1.1)

The appropriate delay is calculated by taking the dot product of the propagation direction, \( \vec{\zeta} \), and the microphone positions, \( \vec{x}_m \), and dividing it by the speed of sound, \( c \) (Eq. 1.2).

\[ \Delta_m \equiv \frac{-\vec{\zeta} \cdot \vec{x}_m}{c} \]  

(1.2)

### 1.2 Description of the RAW Processor

Since the beamformer continuously receives streams of audio data, it is important that the beamforming processor is able to handle the continuous data. The RAW processor is a tiled processor developed by the Computer Architecture Group at MIT which was developed to handle streams of data.

The RAW processor is based on a simple tile design. Each tile contains a tile processor, static switch, and dynamic router and can be placed next to each other to form as large of processor as desired [4]. The advantages of the RAW processor are its scalability and adaptability. The RAW processor’s scalability allows the incoming data to be streamed and routed be without going through the memory paths and caches that are used in other processors. These qualities makes RAW a favorable processor for beamforming.

The RAW processor has two different types of networks: static and dynamic. The static network is optimized for deterministic data. Each tile performs the necessary process on the incoming data before sending the data out through the static switch to the next tile. The dynamic network, is used for non-deterministic data. In the dynamic network, the incoming data packet can be up to 32 words long; the beginning of the data contains a header word that tells the RAW processor how many words
the data packet contains, where to route the data and where the data is coming from.

Since the digital audio data is constantly streaming and deterministic, the static network is used. This means that the front end, which is the component that prepares the microphone data for the processor, interfaces with the Static Input Block, which handles the handshaking and the sending and receiving of static data of the RAW processor. The I/O interface of the RAW processor is further described in Chapter 3.

1.3 Thesis Statement and Overview

This thesis discusses the design and implementation of the front end of this audio beamformer. The front end design consists of taking the streaming digital audio data from the A/D converters and formatting and packaging the data into data packets the RAW processor can use.

Chapter 2 discusses the overall design of the whole system along with the overall front end design. The front end is divided into three subsystems: Network Interface, Control Unit, and Formatting and Packaging Unit.

Chapter 3 discusses the first subsystem in the front end design and implementation, which is the Network Interface.

Chapter 4 discusses the design and implementation of the Control Unit.

Chapter 5 discusses the design and implementation of the Formatting and Packaging Unit.

Chapter 6 discusses the Conclusion and Future Considerations.
Chapter 2

Overall Design

As stated earlier, the beamformer uses the RAW processor to do the actual beam-forming. To implement the audio beamformer, microphones are used to capture the audio signal. Between the microphones and the RAW processor, the audio data needs to be formatted and packaged so that the processor can understand and process the data. This part is done in a Xilinx Field Programmable Gate Array, FPGA. The overall system is implemented as shown below in Figure 2-1:

2.1 Microphone Array

Microphones are used to detect and gather the audio data. As the microphones capture the audio data, Analog to Digital converters digitize the analog data. Cirrus

\footnote{A VirtexE Xilinx FPGA was used. The specifications for the Xilinx FPGA can be found at: http://www.xilinx.com/partinfo/ds022.pdf}
CS53L32A A/D converters were used. These A/D converters can run in different modes, and this thesis utilizes two modes: stand alone mode and serial peripheral interface, or SPI, mode. In stand alone mode, none of the A/D options, such as gain, format, and volume control, are specified. Instead, they are all in default mode. In SPI mode, the user is able to specify A/D options. These options are written into internal registers in the A/D converter.

For each A/D converter, there are two microphone channels: left and right. One of the most important signals in the A/D converter is the LRCLK, which specifies which channel is being converted and the sampling rate. Every column of microphones, which consists of 16 A/D converters, are grouped together called microphone boards. The microphone array setup is shown in Figure 2-2. The data is interleaved in each CPLD on the microphone A/D boards before being sent to the FPGA. The importance and structure of the interleaving data will be discussed in 5.1. The A/D converter continuously outputs streams of the digital audio data.

Once the data is interleaved, it is sent to the FPGA for front end processing. The FPGA takes the incoming serial data and converts it to packets of 32-bit data for the RAW processor. The FPGA also keeps track of control options, such as which microphones the user wants to turn on or off. The data packets also contain the relative time difference between data packets. The RAW processor performs beamforming algorithms on the data packets.

### 2.2 Front End Implementation

The front end implementation of the beamformer is contained in the FPGA, which is divided into three major subsystems: Network Interface, Control Unit, and Formatting and Packaging Unit. The basic interactions between the three systems is illustrated in Figure 2-3. The Network Interface handles the handshakes between the FPGA and the RAW processor and ensures that the FPGA receives all the data the

---

2 The specifications for the A/D converters can be found at: http://www.cirrus.com/en/products/pro/detail/P140.html
processor has sent out and that the processor receives all of the audio data.

The Control Unit takes the data RAW has sent out and determines whether or not it is control data. Control data specifies different beamforming options. These options are either sent to the A/D converter or kept in the FPGA for further use.

The Formatting and Packaging Unit takes the continuous streams of data from the microphone array, extracts the audio data from the data stream, and stores it in an internal buffer. The Formatting and Packing Unit then reads the data from the internal buffer and formats it into data packets. The specifics of the data packets are discussed in section 5.6.1.
Figure 2-3: Block Diagram of Beamformer Front End
Chapter 3

Network Interface

During setup, the RAW processor sends the beamformer specifications into the FPGA. During beamforming, the FPGA sends the digital audio data to the RAW processor. Handshaking between the RAW processor and the FPGA ensures that no data is dropped. The FPGA communicates with RAW on its static network. This is executed in the Network Interface.

The Network Interface consists of two components - the Input Interface and the Output Interface. A block diagram is shown in Figure 3-1. The Input Interface is responsible for the handshaking of the beamformer specifications sent from RAW to the rest of the FPGA. The Output Interface is responsible for the handshaking of audio data packets from the FPGA sent to RAW.

![Figure 3-1: Block Diagram of Network Interface](image-url)
3.1 Description of Network Interface

As the Network Interface performs the handshake with the RAW processor, there are three important signals the Network Interface needs to handle: Data, Ready, and Yummy. By examining these signals properly, the Network Interface ensures that all of the incoming and outgoing data from the FPGA is processed and not dropped.

3.1.1 Input Interface

To perform the static network handshake for the Input Interface, the RAW processor outputs two signals - DATA-IN and READY-IN - and expects one signal from the FPGA - YUMMY-OUT - as illustrated in Figure 3-2. The RAW processor sends out the beamformer specifications on a 32-bit DATA-IN bus. When the DATA-IN is valid, RAW outputs a high READY-IN output to the FPGA. To finish the handshake, RAW expects the YUMMY-OUT signal from the FPGA. The YUMMY-OUT signal indicates when the FPGA is done processing and using the current signal and is ready for the next one.

![Figure 3-2: Block Diagram of RAW handshake with Input Interface](image)

With the handshaking protocol and the architecture of the RAW processor, it is not optimal for RAW to send out one data word at a time. In the static network of the RAW processor, RAW receives data at its Static Input Block, which then sends the data to its Switch Processor to be routed to the correct tile for processing. Thus, the information between the Static Input Block and the Switch Processor are a cycle apart, which means that at any given time the exact space available on the RAW side cannot be determined by the FPGA, and vice versa [4]. Sending out one data at a
time leads to stalling cycles as the RAW processor waits for its Switch Processor to catch up, which decreases the throughput. To remedy this, RAW can send out more than one data at a given time without dropping any data. To increase throughput, it is optimal for RAW to send out at least three elements of data before receiving a YUMMY-OUT signal. To handle this, the RAW processor has a small First In First Out Buffer, or FIFO, at every one of its DATA input ports. Since RAW is a tiled processor, it also expects the components connected to its DATA-IN output ports to have a small FIFO [4].

### 3.1.2 Output Interface

The output interface is very similar to the input one, shown in Figure 3-3. Likewise, the RAW processor expects a 32-bit DATA-OUT bus along with the appropriate Ready signal. The FPGA sends out the digital audio data through the DATA-OUT bus, and when the RAW processor is ready for the next audio data, it sends a YUMMY-IN signal into the FPGA. With a similar handshaking protocol to the input interface, the RAW processor also has a FIFO at its DATA input to improve throughput.

![Figure 3-3: Block Diagram of RAW handshake with Output Interface](image)

### 3.2 Design and Implementation of Network Interface

The Network Interface is implemented with two blocks - the Network Input Block and the Network Output Block. These blocks handle the described handshakes and
prepare the data for their respective destinations. The Network Input Block, which takes the data coming out of RAW, includes the Network Input Interface component and a four element FIFO.

3.2.1 Network Input Interface

The Network Input Interface determines when to send out YUMMY-OUTs and when valid data is received from RAW. Once valid data is received, it is written into the four-element FIFO. As described earlier, the RAW processor sends out at most three valid data before receiving a YUMMY-OUT signal. The four-element FIFO ensures that data from RAW is held until it is needed by the rest of the system. Since only three valid data are output at any given time, a three-element FIFO would have been enough. The extra element was added for safekeeping to ensure that no data is dropped.

The implementation of the Network Input Interface component was quite trivial as shown in Figure 3-4. Once the Network Input Interface receives a READY-IN signal from RAW, the incoming data is written into the four-element FIFO. Every time data is read from the four-element FIFO, a YUMMY-OUT signal is sent out to RAW. By doing this, RAW sends data out to the FPGA when it can.

![Figure 3-4: Block Diagram of Network Input Interface](image)

The FIFO receives its read signal when the entire system is done processing the current data. There are three kinds of data coming from RAW: SPI data, Microphone...
Enable data, and Master Control data. All of these data are written into the Control Register File, which is further discussed in section 4.1, but the SPI data, which is written into the A/D converters, needs to be transformed into serial data. The system waits until the SPI data is transformed into serial data before it is ready for the next beamformer data. Once the Microphone Enable data and Master Control data, are written into the Control Register File, the system is ready for the next incoming beamformer control data. Based on the type of data, the system determines when it is ready for the next data. Once it is ready, the next data is read from the FIFO.

This is implemented in the Read-Val-in component. Based on the type of data that is sent out, Read-Val-in determines when to read data out of the four-element FIFO and signal to the rest of the system that there is new data with the DATA-AVAIL signal. It also automatically reads the first data out of the FIFO after a reset to start the beamformer controls and sends it to the Control Unit. To determine when to signal DATA-AVAIL to the rest of the system, Read-Val-in also keeps track of the status of the FIFO, especially when the FIFO is empty, and the read requests that are sent to the FIFO.

### 3.2.2 Network Output Interface

The Network Output Interface also contains a FIFO, and a Read-Val-out component, along with the Network Output Interface shown in Fig. 3-5. Like the FIFO in the Network Input Block, this FIFO is for safety and to ensure that none of the outgoing data is dropped because the FPGA is running at a different clock than the RAW processor. The FIFO is then connected to the Network Output Interface. This determines when to send data to RAW. Since RAW has a FIFO at its DATA-OUT input, three valid data can be sent out at any given time. To implement this, there is a two-bit counter in the Network Output Interface to keep track of how many valid data has been sent out. Every time the Network Output Interface asks for data from the FIFO, the counter increments by one. Every time a YUMMY-IN is received from the RAW processor, the counter decrements by one. Thus, if there are already three data sent out, the Network Output waits for a YUMMY-IN before sending another
valid data into RAW. Conversely, if less than three data is out, the Network Output Interface sends out more data.

![Block Diagram of Network Output Interface](image)

Figure 3-5: Block Diagram of Network Output Interface

With this protocol, there is a special case that needs consideration. When the FIFO is empty, the Network Output Interface cannot read any data. So for that case, the Network Output Interface stalls until the FIFO is not empty. But, there is a one cycle delay between the time when the FIFO is really empty and the time that the Network Output Interface receives the empty signal. So, the Network Output Interface can inadvertently ask for data when there is no data to give. This situation can throw off the counter by incrementing when the FIFO is empty. The Read-Val-out component keeps track of this, and signals to the counter when it increments when the FIFO is empty. When the counter gets the No-Next signal from the Read-Val-out component, it decrements.

The Network Output Interface also determines when the data from the FIFO is valid and when the data at the input has already been sent out. Every time the Network Output Interface wants to send data to RAW, a read is sent to the FIFO. Since the valid data outputs from the FIFO at the next clock cycle, the read is tracked and compared to the FIFO’s empty signal in Read-Val-out, to make sure the Network Output Interface receives the data at the same time it receives a signal telling it that the data is valid. If the FIFO is empty, the Network Output Interface waits until data is written into the FIFO. Otherwise, if the read is valid, Read-Val-out signals the Network Output Interface with an asserted DATA-COMP signal. The Network Output Interface outputs the incoming data, DATA-OUT, and a READY-OUT signal.
Chapter 4

Control Unit

As the RAW processor, A/D converters, and the FPGA interact with each other, as illustrated in Figure 4-1, information about the data packets needs to be communicated with each other. The RAW processor specifies which microphones are enabled. The RAW processor also determines the register options, such as data format and gain, for the A/D converter. The Control Unit stores and packages this information into the necessary format for the A/D converter.

![Figure 4-1: The Top Level Block Diagram on the control data interaction between the A/D converters, FPGA, and RAW](image)

The Control Unit consists of five components: Control Register File, SPI Control, Control Ready, Shift Register, and Register Done (Fig. 4-2). The Control Unit receives all of the beamformer control specifics from the Network Interface. First,
all of this information is stored in the Control Register File in the FPGA. Once the beamforming specifics are stored in the Control Register File, the SPI Control determines which module to send the information. For the data that sets the control registers for the A/D converter, the Control FSM packages the data correctly for SPI and the Shift Register outputs the data serially. Once the Control Unit is done storing and sending each beamformer specification, the Register Done signals that it is ready for the next one.

![Figure 4-2: Block Diagram of Control Unit](image)

**4.1 Description and Implementation of Control Register File**

The Control Register File is a dual port register file that contains 16-bit words with a depth of 72. The Control Register File contains a Master Control Register, seven A/D Control Registers, and Microphone Enable Registers. The Master Control Register specifies whether or not the beamformer is collecting data from the microphones. The A/D Control Register specifies the register options for SPI mode in the A/D converter. The Microphone Enable Registers determine which microphones are enabled. All of
this information is specified by the Control Data coming out of the RAW processor and written into the appropriate control registers. The address and the data-input of the first port is controlled by the Control Data coming from the RAW processor through Network Interface. The data-output of the first port is connected to the Control Ready component. This is illustrated in Figure 4-2.

4.1.1 Control Data

The Control Data are inputs into the FPGA from the RAW processor. The Control Data are 32-bit words, whose format is illustrated in Figure 4-3. When information is specified by RAW, for the A/D Control and Microphone Enable Registers, it is written into the specified registers. The most significant bit, bit[31], is the Read/Write, which specifies whether the data should be written into, with a '0,' or read from, with a '1,' the Control Register File. The bits[23:16] contain the Register Address of the Control Register File that is read or written. Bits[15:0] are the Register Data, and the remaining seven bits, bits[30:24], are unused bits whose values are arbitrary. In the event of a write, the lowest 16 bits of the Control Data is written into the specified register.

<table>
<thead>
<tr>
<th>Read/Write</th>
<th>Don't Cares</th>
<th>Register Address</th>
<th>Register Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 24 23</td>
<td>16 15 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-3: Format of the Control Data

4.1.2 Master Control Register: Register 0

The first and most important register is the Master Control Register, which has the Register Control File address of 0x00. The Master Control Register controls the overall function of the A/D converter and the beamformer with enable bits, illustrated in Figure 4-4. These enable signals are the MASTER MICROPHONE ENABLE, A/D
reset/run, and header enable. The A/D reset/run sets the /RST bit of the A/D converter. The least significant bit of the Master Control Register signifies the A/D reset/run, with a ‘1’ as reset and a ‘0’ as run. The master microphone enable, which is bit[1] of the Master Control Register, either turns on, with a ‘1’, and off, with a ‘0,’ all the microphones specified by the Microphone Enable Registers. The Master Control Register also picks from the two formats, pure audio data format or header format, for the audio data to RAW. The header enable, which is assigned to bit[2], is sent to the Packaging and Formatting Unit to either specify the Pure Audio Data format, with a ‘0,’ or the header format, with a ‘1’. The formats and their descriptions are further discussed in section 4.5.1. The remaining bits of the Master Control Register are currently unused. If the Master Control Register is not specified, the beamformer will be in reset mode.

![Format of the Master Control Register](image)

Figure 4-4: Format of the Master Control Register

### 4.1.3 A/D Control Registers: Registers 01 to 07

The A/D Control Registers determine the specifications for the A/D converter. These are only used when the A/D converters are in SPI mode. There are seven registers that specify the gain, volume control, and data format for the output digital data from the A/D converter. The registers 0x01 to 0x07 of the Control Register File are designated for the A/D Control Registers. The data written into the A/D Control Registers consists of the A/D Control Data that is written into the specified register in the A/D converter, which is the eight least significant bits, the eight most significant bits are ignored (Figure 4-5). The details of the control registers in the A/D are
included in the specifications of the A/D converters\textsuperscript{1}. The address of the Control Register File corresponds with the address of the registers in the A/D converters.

![A/D Control Data](Image)

Figure 4-5: Format of a A/D Control Register

The data from the A/D Control Registers is written to the A/D registers through the SPI port. The data from the A/D Control Registers is sent to the pin, CDIN, serially. To do this, the data is passed through a shift register, which takes in multiple bit words and outputs the data serially. The data from each A/D Control Register is written to all the A/D converters in the microphone array.

### 4.1.4 Microphone Enable Registers

Microphone Enable Registers are 16-bit registers that enable and disable the output of each individual microphone. For an array of 1024 microphones, 64 Microphone Enable Registers are needed. There is one bit for each microphone to specify whether or not the data coming out is needed, with a ‘1’, or unneeded, with a ‘0’. Only the microphones with a corresponding enable of ‘1’ is sent out to the RAW processor. This information can only be specified and changed when the beamformer is not running. To change the Microphone Enable Register, the user needs to overwrite the desired Microphone Enable Register with the new values. This is done through the Control Data from RAW.

For example, if the data from the first five microphones are needed, the Microphone Control Registers is ‘1’ for the corresponding bits and ‘0’ for all the rest as shown in Figure 4-6. This means the data coming out of the FPGA and into the RAW processor includes a header followed by the data from the first five microphones.

\textsuperscript{1}The specifications for the A/D converters can be found at: [http://www.cirrus.com/en/products/pro/detail/P140.html](http://www.cirrus.com/en/products/pro/detail/P140.html)
4.1.5 Control Register File Layout

To include all the needed registers, the Control Register File needs 72 registers. The first register is the Master Control Register, followed by the A/D Control Registers, and finally the Microphone Enable Registers. The Control Register Layout is illustrated in Figure 4-7.

4.2 SPI Control

Based on the Address of the Control Register File, the SPI Control determines the type of data and which module in the FPGA the data goes to. The SPI Control receives three types of data: A/D Control, Master Control, and Microphone Enable. When the Control Unit receives an A/D Control or Master Control data, the SPI Control signals the Control FSM with the SPI-REG and CONTROL-REG outputs to start processing these signals. When the Microphone Enables are received, they are just stored into the Control Register until the beamformer starts running and processing the digital audio data from the A/D converter. The SPI Control also signals to the Formatting and Packaging Unit when the beamformer specifies the Digital Audio Data Format, which happens when address 0x02 of the Control Register File is written into. At this event, the SPI Control sends a FORMAT-REG signal to the Formatting and Packaging Unit. The FORMAT-REG signal and the specifications of the Digital Audio Data Format is discussed in section 5.4.
4.3 Control Ready

The Control Ready Block takes the outputs of the Control Register File, which are called CONTROL-INTERNAL, and prepares them for their destinations - which is either the A/D converters or the Formatting and Packaging Unit. The Control Ready receives the control-reg and spi-reg signals from the SPI Control to determine which function is needed and what values need to be output, as illustrated in the flowchart in Figure 4-8. When a control-reg is received, the Control Ready starts extracting the MASTER MICROPHONE ENABLE, HEADER ENABLE, and A/D RESET/RUN signals.
from the Master Control Register. The **MASTER MICROPHONE ENABLE** is then sent to
the Packaging Unit, which handles all the microphone enables. The **A/D RESET/RUN**
signal is also sent to the Packaging Unit. When the **A/D RESET/RUN** is asserted, the
Packaging Unit is turned off, otherwise it is running.

![Flowchart of Control Ready](image)

**Figure 4-8: Flowchart of Control Ready**

The **SPI-REG** signals that the incoming data needs to be sent out to the A/D
converter. Then, Control Ready packages the control data into a format the A/D
converters understands (Fig. 4-9). The A/D converter expects a string of ‘001000,’
which is the A/D SPI chip address, followed by a **READ/WRITE** bit, Register Address
and 8-bit Register Data, in order to signal a write. The **READ/WRITE** bit acts as a
read when high, and a write when low. The **READ/WRITE** signal is taken from bit
31 from the Control Unit data input. Next, is the A/D Control Address. Since the
**SPI-REG** signals that the address is between 0x0 and 0x7, only the lowest three bits
are needed. The upper bits are always ‘0.’ Thus, the A/D Control Address is in the
form of a string of ‘00000’ followed by the lowest three bits of the Register Address.
Finally, the register data, which is the lowest eight bits in the control register, is
 appended to the end. Once this is finished, the Control FSM outputs the data and
signals the Shift register to load the data.
4.3.1 Change in Clock Domains

One of the most important things to keep track of is when signals cross clock domain. In the Control-Ready component, LOAD signals to the Shift Register when to load the valid data. The LOAD signal needs to be high for one outgoing clock cycle. Thus, the LOAD signal needs to be synchronized to the outgoing clock. The clock of the incoming data to the Control Ready is the same as the Control Register, Network Interface, and RAW, which is called RAW CLOCK. However, the clock of the outgoing data from the Control FSM is the clock of the Shift Register and the A/D Converter’s control sampling clock, CCLK. Since the CCLK is running at a much slower clock rate than the RAW CLOCK, the LOAD needs to be synchronized to CCLK and held for one cycle of CCLK.

This is implemented by a series of registers running with either the RAW CLOCK or CCLK and an AND gate, illustrated in Figure 4-10. The first register synchronizes the INTERNAL LOAD signal with CCLK before sending it through two more registers running with the CCLK. The INTERNAL LOAD signal is held until a CLEAR is received. Then, the output signal of the registers is synchronized with RAW CLOCK to clear the INTERNAL LOAD signal after one CCLK cycle. The output of the RAW CLOCK registers becomes the CLEAR signal for Control Ready. Once the FSM receives this clear signal, the INTERNAL LOAD is cleared, which eventually clears all the registers. The LOAD output of the Control Ready is taken from the output of the AND gate. The AND gate looks at the difference between the outputs of the two registers. This allows it to ensure that the load output is held for exactly one cycle of CCLK.
4.4 Shift Register and Register Done

Since the A/D converter has a SPI interface, the control data going in needs to be serial data. This is implemented with the Shift Register, which loads the 24 bit output of the Control Ready and serially outputs the data at every rising edge of CCLK. The Register Done signals when the Shift Register has finished outputting the data serially. The Register Done is implemented as a counter. It counts how many bits have been shifted and once it determines that the Shift Register has finished, it outputs a DONE signal to the Network Interface. The DONE signal tells the Network Interface that the Control Unit is ready for the next data.

4.4.1 Change in Clock Domains

Like Control Ready, the Register Done also needs to handle the differences in clock speed. Register Done runs on CCLK but, since DONE goes to the Network Interface, the output needs to be synchronized to the RAW CLOCK. Since the RAW CLOCK is running at a faster speed than CCLK, the DONE signal needs to be shortened and synchronized to the RAW CLOCK. When Control Ready generates a DONE signal, which is referred to as DONE (INTERNAL), the output of Control Ready needs to be
a RAW clock cycle pulse, which is called DONE (OUTPUT). This is implemented with three registers, an AND gate, and an INVERTER (Fig. 4-11). The first register synchronizes the signal to the RAW CLOCK. The AND gate and the INVERTER compares the outputs of the D-Flip Flops, which are one cycle apart, and when it detects that the output of the first Flip Flop, output A, is high and the output of the second register, output B, is low, an one cycle pulse is generated. By using the logic shown, the DONE (OUTPUT) signal becomes one cycle of the RAW clock, as illustrated in Figure 4-12.

![Logic Diagram of Done(output)](image)

Figure 4-11: Logic Diagram of Done(output)

![Waveform of Done(output)](image)

Figure 4-12: Waveform of Done(output)
4.5 Interaction with the System

All of the controls must be specified while the beamformer is in Reset mode, and not processing any audio data. Once the user is done specifying the controls for the beamformer, the beamformer can start receiving and processing data. While the beamformer is processing the digital audio data, the controls cannot be changed until another Reset is asserted. This constraint was made to ensure the uniformity of data formats and microphone enables because a change in specifications cannot be pinpointed quickly, which could cause an invalid packet to be sent to RAW.

This can be illustrated in the following example. The beamformer is recording data from microphones 0 to 49, but the user wants to change so that the beamformer is recording data from microphones 50 to 99. If the user changes the microphone enables while the beamformer is still running, the beamformer may receive less data in each packet than it expects. This is because it takes time before the beamformer is updated with the new microphone enables.
Chapter 5

Formatting and Packaging Unit

The Formatting and Packaging Unit takes the digital audio data from the A/D converters and packages them for the RAW processor. This unit must handle the streaming data from all of the microphones, increment the Dropped Packets Register in the Control Register File accordingly, and keep track of the Microphone Enables. The Formatting and Packaging Unit consists of five different components as shown in Figure 5-1. First, the Data Count takes in a clock from the A/D converters and generates a clock for the Formatting and Packaging Unit. The Data Shift shifts the streaming data from the microphones accordingly. The Data Format puts the incoming data into the requested format. Next, Data Write writes and stores the incoming data into a SRAM. Finally, the Data Package packages the data into data packets for the RAW processor. The source code for the Formatting and Packaging Unit is found in Appendix D.

5.1 Digital Audio Data

The digital audio data coming from the A/D converters is divided into groups of sixteen boards with two microphones each, which are separated into left and right channels. There are two clocks of importance: SCLK and LRCLK. The SCLK is the sampling clock. Each bit of the digital audio data is clocked serially by the SCLK.
The LRCLK is the left/right clock, which indicates which of the two microphones connected to the A/D converter the data is coming out from and specifies the sampling frequency. Since the A/D converters are divided into groups of sixteen, the digital audio data is interleaved serially, and clocked to the FPGA by a clock that is sixteen times SCLK, which is called SCLKx16. The FPGA receives the streaming data, along with LRCLK and SCLKX16. The waveforms describing the Digital Audio Data is shown below in Figure 5-2.
5.2 Description and Implementation of Data Count Block

Since the incoming data is interleaved data from all sixteen microphones, the Formatting and Packaging Unit needs to know when and how to parse the incoming data. The Data Count Block takes this data and signals when to separate bit by bit. Data Count Block is made up of a counter. The counter aligns itself to the LRCLK toggle and start counting. Once it counts to 16, Data Count Block asserts VALID, which signals to the rest of the Formatting Unit when it receives one bit of data from all the microphones. This VALID signal serves as a clock to many components of the Formatting and Packaging Unit.

5.3 Description and Implementation of Data Shift Block

For every group of sixteen A/D converters, there is a Data Shift Block that handles the incoming serial data stream. Since the incoming data stream is interleaved microphone data, Data Shift Block separates the data accordingly, into 16 24-bit parallel data. The Data Shift Block shifts the data serially into the appropriate format, which is determined by Data Format. Data Shift Block consists of one 16-bit shift register, sixteen 24-bit shift registers, sixteen parallel registers, a sixteen-input multiplexer, and a multiplexer-select counter and is implemented as shown in Figure 5-3.

Data Shift Block receives the serial data stream and the sCLKX16 from the A/D converters. The 16-bit shift register continuously receives the serial data and outputs a 16-bit parallel data. Once the 16-bit shift register is done shifting 16 valid data bits, the Data Shift Block receives a VALID signal from Data Count Block. The VALID signal serves as the clock to the sixteen 24-bit shift registers. Thus, at every VALID signal, the 16-bit shift register output is shifted into each of the 24-bit shift registers.
Depending on the specified format, the Data Shift Block receives a DONE signal from the Data Format component. The DONE signal tells Data Shift Block that the data in the 24-bit shift register is correct. Each of the sixteen 24-bit data is held in one of the 16 parallel registers. The sixteen parallel registers serve as inputs to the sixteen-input multiplexer.

Once the correct data is loaded into the parallel registers, the multiplexer-select counter is signaled to start. The multiplexer-select counter determines which input of the multiplexer to output. After the data loads, the multiplexer-select counter increments at every VALID signal and counts from 0 to 15. The output is then connected
to the SELECTOR bits of the multiplexer. Also, the multiplexer-select counter outputs an ENABLE signal whenever it is counting. This signals to the rest of the Formatting and Packing Unit that the data from Data Shift is correct. Thus, at every toggle of LRCLK, the Data Shift component outputs a stream of sixteen 24-bit data.

5.4 Description and Implementation of the Data Format Block

The Data Format Block handles the formatting of the serial data stream. The incoming data can be in three different formats: 24-bit left shifted, 16-bit right shifted, 18-bit right shifted, 20-bit right shifted, and 24-bit right shifted. When the data is left shifted, at every toggle of LRCLK, the incoming data is valid as illustrated in Figure 5-4. When the data is right shifted, the incoming data is valid right before the LRCLK toggle like in Figure 5-5. There is a Data Format Block for each Data Shift Block.

![Figure 5-4: Waveform of Left Shifted Data](image)

![Figure 5-5: Waveform of Right Shifted Data](image)
When the SPI Control of the Control Unit determines that the data format is being specified, it sends the Data Format Block a FORMAT-REG signal along with 3-bit FORMAT signal. The FORMAT-REG signal tells the Data Format Block that the FORMAT signal is valid. Depending on the values of FORMAT, the Data Format Block configures the data accordingly. The FORMAT values and corresponding data format is shown in Table 5.1. Data Format consists of a counter and a finite state machine. The counter divides the SCLKX16 by sixteen. The counter is aligned with the LRCLK toggle and increments from zero to fifteen. The counter outputs a VALID signal after every sixteen cycles, which is connected to the Valid signal of the Data Shift.

Table 5.1: Corresponding FORMAT values to data formats

<table>
<thead>
<tr>
<th>FORMAT Values</th>
<th>Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>I²S up to 24-bit Left Shifted</td>
</tr>
<tr>
<td>001</td>
<td>up to 24-bit Left Shifted</td>
</tr>
<tr>
<td>010</td>
<td>Reserved</td>
</tr>
<tr>
<td>011</td>
<td>16-bit Right Shifted</td>
</tr>
<tr>
<td>100</td>
<td>24-bit Right Shifted</td>
</tr>
<tr>
<td>101</td>
<td>18-bit Right Shifted</td>
</tr>
<tr>
<td>110</td>
<td>20-bit Right Shifted</td>
</tr>
<tr>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

5.4.1 Data Format Finite State Machine Description

The FSM does the actual formatting. The FSM consists of three major states: IDLE, LEFT, and RIGHT (Fig 5-6). When Data Format receives the GO signal, which is connected to the FORMAT-REG signal, the FSM determines which state to go to: Left or Right. When it is in the LEFT state, the data is left shifted, which means valid data is coming from the A/D converter right after every LRCLK toggle. First, the FSM aligns itself with the LRCLK toggle. Since the left shifted data is always 24-bits, the LEFT state increments up to 24 at each VALID signal from the Data Format Counter, or DFC. Once it increments up to 24, a DONE signal is sent out to the Data Shift Block to signal the end of the valid data stream. If the data is right shifted,
the FSM goes to the RIGHT state, which means the end of the valid data stream is aligned with every LRCLK toggle. In this case, once a LRCLK toggle is detected, a DONE signal is sent out to the Data Shift Block to signal the end of the valid data stream. The LRCLK toggles are detected by comparing the LRCLK to LRCLK-OLD, which is a one cycle delayed LRCLK, and taking the XOR of the two values. Since the right shifted data can be either 16-bit, 18-bit, 20-bit, or 24-bit wide, the RIGHT state also takes care of the width. When the DONE signal is asserted, the FSM takes the output from Data Shift Block and selects the correct number of bits. Then, it pads the remaining bits by sign extending the most significant bit.

![State Diagram of Data Format](image)

Figure 5-6: State Diagram of Data Format

## 5.5 Design and Implementation of Data Write

The Data Write Block writes and stores the data into a SRAM as data packets for each group of sixteen A/D converters, and also increments the Dropped Packets Register whose Block Diagram is shown in Figure 5-7. A data packet contains the data from all of the microphones for a given time. The Data Write Block takes the 24-bit output from the Data Format Block and the ENABLE output of the Data Shift Block to determine when the data is valid. Once the data is valid, the Data Write

---

47
Block starts storing the data in the SRAM. The SRAM has a depth of 64 and a width of 24-bits. Since each group of sixteen A/D converters contains 32 microphones, the SRAM can store up to two whole data packets from each group of A/D converters. The SRAM is divided into two buffers, a writing buffer and a reading buffer. The Data Write Block also updates the Dropped Packets Register by looking at the most significant bit, bit[5], of the SRAM's reading address.

Figure 5-7: Block Diagram of Data Write

### 5.5.1 Dropped Packets Register

When the incoming data from the A/D converter is coming in faster than how fast the RAW chip is processing the data, it is possible that the internal buffering fills up. To make room for the current incoming data, entire data packets need to be dropped. When this happens, the Dropped Packets Register is incremented. This is important because the RAW processor needs to know the relative time span between each data packet. After each data packet is sent out to RAW, the Dropped Packets Register resets. If desired, the Dropped Packets Register is sent out to RAW with each data packet in the header format, which is described in section 5.6.1.
5.5.2 Data Write Implementation

The Data Write is synchronized with the Data Format and Data Shift. Once Data Write receives an asserted Enable signal from Data Shift, it starts writing into the SRAM. Initially, the writing buffer of the SRAM is the first 32 addresses. Data Write also keeps track of the writing address and the reading address, which is determined by the Packaging component. Once the data from microphone 32 is written into the SRAM, Data Write checks the most significant bit of the reading address. If the most significant bit of the reading address is different than the writing address, the Packaging Component is not done reading the current data packet in the reading buffer. Thus, Data Write directs the writing address back to the beginning of the current writing buffer. The newly written data packet is overwritten by the most current data; one whole data packet is “dropped,” and the Dropped Packets Register increments by one. When the reading address is the same as the writing address, this means the Packaging component is done reading the data packet in the reading buffer and has moved on to the writing buffer. Then, the reading and writing buffers switch. Data Write directs the writing address to the reading buffer, which then becomes the current writing buffer.

5.6 Design and Implementation of the Packaging Component

The Packaging Component handles the data packets from all of the Data Writes, in this case 32. Once the Data Write starts writing data packets into SRAM, the Packaging Component starts reading the data packets and packaging them for the RAW processor. The Packaging Component also looks at the Microphone Enable Registers in the Control Register File and determines whether or not the data should be sent to RAW.

The Packaging Component keeps track of the writing address to make sure that the data packets that are read are correct. If the Packaging Component is reading from
the SRAM faster than the Data Write is writing to it, the Packaging Component waits for the Data Write Block. As the Packaging Component is reading the data, it is also getting the corresponding microphone enables from the Microphone Enable Registers in the Control Register File of the Control Unit. It also determines which Microphone Enable Register is needed by specifying the Enable Address to the Control Register File. When the enable is high, the VALID signal is asserted, otherwise, the VALID signal is low. Since the RAW processor expects 32-bit data, the Packaging Component pads the 24-bit data from the SRAM by sign extending the most significant bit. As the Packaging Component generates data packets, the outgoing 32-bit data is sent to the Network Interface before it is sent to RAW.

The Packaging Component also receives and considers other status signals. From the Control Unit, the Packaging Component receives the HEADER ENABLE signal. This signal chooses which data packet format the outgoing data packets is in. The Packaging Component also checks the status of the Output Interface FIFO in the Network Interface. The importance of these status signals is further described in section 5.6.2.

5.6.1 Outgoing Data Packet Formats

With the HEADER ENABLE input from the Control Unit, the Packaging Component determines which of the two data formats to generate. When the Header Enable is asserted, the outgoing data is in the Header format, otherwise it is in the Pure Audio Format.

Header Format Data Packets

In the Header Format, the Packaging Component adds specific data words to signal beginning and end of each data packet. At the beginning of each data packets, the RAW processor expects a Heading Word, followed by the Dropped Packets Register, microphone data, and an Ending Word. The Heading and Ending Words signals to the RAW processor when to expect a new data packet. This is illustrated in Figure
Before the Packaging Component starts reading each data packet, the Beginning Word, which is, in hexadecimal, 0x01000000, is generated, followed by the Dropped Packets Register in the Control Register File. Once the Packaging Component reads the last microphone data in the data packet, the Ending Word, is generated. The Ending Word is 0x02000000. Once the data packets are ready, they are sent to the Network Interface to be sent out to the RAW processor.

**Pure Audio Format Data Packet**

The Pure Audio Format is exactly as the name implies. There are no Heading or Ending Words, and Dropped Packets Register is also ignored. Instead, the Pure Audio Format gives a steady stream of digital audio data. Each data packet contains the audio data of the desired microphones. An example of a Data Packet with audio data from Microphones 0, 1, 2, and others is shown in Figure 5-9.
5.6.2 Packaging FSM

The Packaging Component is implemented as a Finite State Machine (Fig 5-10). The Packaging FSM is a step by step process of gathering all the digital audio data into data packets. The first state is the IDLE STATE, which is the state the FSM initializes to and resets to. While the FSM is in the IDLE STATE it waits for the data to be formatted. Once it is formatted, the FSM moves to the next state, depending on the Header Enable. If enabled, the FSM moves to the HEADER STATE, which generates the Heading Word for the data packet. If the Header Enable is not asserted, the FSM moves to the MIC-0 STATE, which starts reading the stored microphone data from the SRAM and the microphone enables from the Control Register File.

For the Header Format case, the FSM generates the Heading Word in the HEADER STATE, and then move to the DROPPED STATE, which outputs the Dropped Packets Register. Next, it moves to the MIC-0 STATE and start gathering the microphone data. The MIC-0 STATE gets the data from the first address and moves the MIC STATE. MIC STATE increments the read address of the SRAM. It also increments the MICROPHONE ENABLE bit by one. Once the FSM has gotten the data from 16 microphones and has finished reading the 16-bit Microphone Enable, it increments the Enable Address and assert DONE-SINGLE. Once the Microphone Enable address is incremented, there is a delay of a clock cycle before the current enable data is received by the Packaging Component. Thus, the FSM moves to the WAIT STATE, which waits for two clock cycles, and, once the correct microphone enable data is received, loops back to the MIC STATE. The FSM loops around MIC STATE and WAIT STATE until it reads the data from all the microphones. Once it is done, a DONE signal is asserted, and the next state is the DONE STATE. This state generates the Ending Word and loops back to the HEADER STATE. When the DONE signal is asserted, it also switches the most significant bit of the read address. As it was discussed earlier, the SRAM is split up into two buffers, so, by switching the most significant bit of the read address, the Packaging Component starts reading from the next buffer. For the Pure Audio Format case, the Finite State Machine goes directly
to the MIC STATE and loops around the MIC-0 STATE, MIC STATE, and WAIT STATE continuously.

There are also some special cases to be considered. One important case is when the write and read address to the SRAM are the same, and another is when the FIFO in the Network Interface is full. The first case is important because, in order to have an accurate data packet, the data from all the desired microphones needs to be from the same sample. Thus, it is important that the read address never overtakes the write address. So, the Finite State Machine also compares the read and write addresses of the SRAM. In the MIC STATE, if the read and write addresses are the same, the read address, Enable Address, and the Enable Bit Number does not increment, and the microphone data is not read. Once the write address of the SRAM moves on, the

Figure 5-10: State Diagram of Packaging FSM
MIC state reads the current microphone data and increments accordingly.

When the 255 FIFO in the Network Interface is full, the FSM stalls in whatever state it is in, and does not output anything until the FIFO has more room. This solution is actually less trivial than it seems. Because of pipelining and the architecture of the FIFO, the full signal is always a clock cycle behind. So, it would be really easy to drop data values. To remedy this, the FSM waits a cycle after it writes to the FSM to get the correct full signal before writing the next value.
Chapter 6

Conclusion

6.1 Summary

The beamformer front end has been designed, implemented, and tested and debugged on the RAW emulator and the RAW processor. One thing to note is that the emulator is running a lot slower than the actual RAW processor. When the beamformer is actually run on the RAW processor, the change in clock speed will have to be noted. The changes are noted and commented out in the Verilog code in the Appendix.

With the slower emulator clock, there are a lot of dropped packets at certain times. When the 255 depth FIFO is saturated, and the Packaging Component of the Formatting and Packaging unit is waiting for more room in the FIFO, there are a lot of packets dropped - roughly 60 to 70. Fortunately, this even does not happen often. Most of the time, the dropped packets are less than four. This issue will not be apparent since the actual RAW clock is two orders of magnitude greater than the emulator clock.

On the RAW processor, digital audio data can be gathered from multiple microphones. The RAW processor drops significantly less data than the emulator, but the amount of data dropped is not helpful in performing the beamforming algorithms. Fortunately, the dropped data is due to the testing setup, not the actual processor or front end.

Aside from this case, the front end implementation of the audio beamformer is
able to choose the desired microphones and change the audio data format. The biggest difficulty was keeping track of the various clocks within the front end, and synchronizing the necessary signals to the clocks.

Integrating the front end with the RAW processor and the A/D boards also had its difficult moments. The importance of the four-element FIFO in the Network Interface was soon realized when control signals from the RAW processor were randomly dropped.

6.2 Future Directions

For the future, it will be exciting to see the processing capabilities of the RAW processor. Once the beamformer is moved from its current setup, the most optimal microphone array scheme and the optimal number of microphones can be determined.
Appendix A

Top Level Verilog Source Code

module top (POO, YIO, RDYOO, LED, SCLK, LRCLK, RST, PIO, RDYIO, LE_CLK, LE_XVPI_RW_N, SDOUT, YOO);

    input SDOUT;
    input YOO;
    input LE_CLK;
    input LE_XVPI_RW_N;
    input [1:0] RDYOO;
    input [31:0] POO;
    input SCLK;
    input LRCLK;

    output [31:0] PIO;
    output [1:0] RDYIO;
    output RST;
    output [3:0] LED;
    output [2:0] YIO;

    wire [31:0] data_internal, reg_data_internal, data_p;
    wire [15:0] enable_internal, dummy, enable_p;
    wire [15:0] data_16, data_16_sync;
    wire [23:0] data_sram_internal, sram_dummy, data_shift_internal, data_package;
    wire valid_internal, start, r_set, lo, valid_p, r_en, l_en;
    wire s_clk, rdy, rdy_s, hi, valid_clk;
    wire [5:0] w_addr, r_addr;
    wire [6:0] en_addr;
    wire [3:0] mux_sel;
    wire [11:0] dropped;
    wire [2:0] format;
wire we, mux_valid, sram_we, fifo_we, full, format_go, data_comp, read;
wire almost_full, almost_empty, en_next;
reg go, go_0;
assign format_go = go && !go_0;
assign format = 3'b0;
assign YI0[2:1] = 2'b0;
assign sram_dummy = 24'b0;
assign dummy = 16'b1111111111111111;
assign lo = 1'b0;
assign RDYI0[1] = 1'b0;
assign LED[0] = LE_XVPI_RW_N; //D24
assign RDYI0[0] = rdy;
assign rdy_s = !RDYOO[1] && RDYOO[0];
assign hi = 1'b1;
assign r_set = !go;

always @(posedge LE_CLK or posedge LE_XVPI_RW_N)
begin
if (LE_XVPI_RW_N)
begin
  go <= 1'b0;
end
else
begin
  // holds previous value and determines if there is a start signal from
  go <= go || start;
  go_0 <= go;
end
end

BUFG (.I(SCLK), .O(s_clk));

network_input ni (.data_out(reg_data_internal), .data_avail(start), .yummy(YI0[0]), .clk(LE_CLK), .reset(LE_XVPI_RW_N), .thanks(hi), .valid(rdy_s), .data_inn(P00));

data_done sdd (.valid(valid_clk), .clk(s_clk), .reset(r_set), .lrclk(LRCLK));

data_out_reg sdr (.CLK(s_clk), .SDIN(SDOUT), .Q(data_16));
data_sync dsync (.data_sync(data_16_sync), .s_clk(s_clk), .valid_clk(valid_clk), .reset(r_set), .data_16(data_16));
data_format_revised dfr (.valid(valid_internal), .clk(valid_clk), .reset(r_set));
et), .lrclk(LRCLK), .format(format), .format_go(go));

data_write dw (.drop_num(dropped), .sram_we(sram_we), .data_out(data_sram_internal), .sram_addr(w_addr), .clk(s_clk), .reset(r_set), .data_in(data_shift_internal), .valid(we), .rd_msb(r_addr[5]), .format_go(go), .format(format));

package pg (.en_next(en_next), .data_out(data_internal), .reg_addr(en_addr), .mic_addr(r_addr), .write(fifo_we), .clk(s_clk), .reset(r_set), .reg_data_in(enable_internal), .mic_data_in(data_package), .data_addr(w_addr), .dropped(dropped), .master_mic(hi), .full(almost_full), .header(hi));

data_sram dsrm (.addra(w_addr), .addrb(r_addr), .clka(s_clk), .clkb(s_clk), .dina(data_sram_internal), .dinb(sram_dummy), .douta(), .doutb(data_package), .ena(!r_set), .enb(!r_set), .wea(sram_we), .web(lo)); //

data_shift dst (.data_out(data_shift_internal), .clk(valid_clk), .mux_clk(s_clk), .reset(LE_XVPI_RW_N), .data_in(data_16_sync), .valid(valid_internal), .addr(mux_sel));

mux_select ms (.we(we), .mux_sel(mux_sel), .clk(s_clk), .reset(LE_XVPI_RW_N), .valid(mux_valid));

comp4_asynch cout (.almost_full(almost_full), .almost_empty(almost_empty), .data_comp(data_comp), .read(read), .counter(LED[3:2]), .fifo_empty(LED[1]), .fifo_full(full), .data_out(P10), .valid(rdy), .w_clk(s_clk), .r_clk(LE_CLK), .reset(r_set), .yummy(YOO), .write(valid_p), .data_in(data_p));

register_file_dual_port rdp (.addra(reg_data_internal[22:16]), .addrb(en_addr), .clka(LE_CLK), .clkb(s_clk), .dina(reg_data_internal[15:0]), .dinb(dummy), .doutb(enable_internal), .ena(hi), .enb(hi), .sinita(LE_XVPI_RW_N), .sinitb(LE_XVPI_RW_N), .wea(reg_data_internal[31]), .web(lo));

pipeline_data pd (.data_mic_p(data_p), .valid_mic_p(valid_p), .clk(s_clk), .reset(LE_XVPI_RW_N), .valid_mic(fifo_we), .data_mic(data_internal));

pipeline_enable pe (.enable_p(enable_p), .clk(s_clk), .reset(LE_XVPI_RW_N), .enable(enable_internal));

pipeline_mux pm (.valid_p(mux_valid), .clk(valid_clk), .reset(LE_XVPI_RW_N), .valid(valid_internal)); //

endmodule
Appendix B

Verilog Source Code - Network Interface

B.1 Network Input Interface

B.1.1 Top Level of Network Input Interface

module input_test (avail, empty, data-out, yummy, clk, reset, done, valid, start, thanks, data-in);

    input [31:0] data_in;
    input valid;
    input done;
    input reset;
    input clk;
    input start;
    input thanks;

    output yummy;
    output [31:0] data-out;
    output empty;
    output avail;

    wire [31:0] data_internal, data_p;
    wire write;
    wire avail_p;
    wire yummy_internal;
assign yummy = yummy_internal;

network_input ni (.data_out(data_internal), .data_avail(write), .yummy(yummy_internal), .clk(clk), .reset(reset), .thanks(thanks), .valid(valid), .data_in(data_in));

read_val rv (.read(avail_p), .clk(clk), .reset(reset), .done(done), .start(start), .empty(empty), .read_fifo(avail_p), .yummy(yummy_internal));

fifo_4_2 f4 (.empty(empty), .data_o(data_p), .reset(reset), .clk(clk), .data_i(data_internal), .write_in(write), .read_in(avail_p));

pipeline_network pn (.data_out(data_out), .valid_out(avail), .clk(clk), .reset(reset), .data_in(data_p), .valid_in(avail_p));
endmodule

B.1.2 Verilog Source Code - Input Interface

module network_input (data_out, data_avail, yummy, clk, reset, thanks, valid, data_in);

//inputs
input [31:0] data_in;
input valid;
input thanks;
input reset;
input clk;

//outputs
output [31:0] data_out;
output data_avail;
output yummy;

//internal wires and regs
reg [31:0] data_out;
reg data_avail;
reg yummy;

always @ (posedge clk or posedge reset)
begin
if (reset)
begin
    data_out <= 32'b0;
    data_avail <= 1'b0;
    yummy <= 1'b0;
end
else if (valid)
begin
  data_out <= data_in;
  data_avail <= valid;
  yummy <= thanks;
end
else if (~valid)
begin
  data_avail <= valid;
  yummy <= thanks;
end
else
begin
  data_out <= 32'b0;
  data_avail <= 1'b0;
  yummy <= 1'b0;
end
endmodule

B.1.3 Verilog Source Code - Four Element FIFO

module fifo_4_2 (empty, full, data_o, element, reset, clk, data_i, write_in, read_in);

  //defines
  'define DATAWIDTH 32
  'define DATADEPTH 4
  'define ELMNT_WIDTH 2

  //inputs
  input [(‘DATAWIDTH - 1):0] data_i;
  input write_in;
  input read_in;
  input clk;
  input reset;

  //outputs
  output [(‘DATAWIDTH - 1):0] data_o;
  output [(‘ELMNT_WIDTH:0] element;
  output empty;
  output full;

  // regs
  reg empty;
  reg full;
  reg first;
  reg second;
  reg third;
reg ['ELMNT_WIDTH:0] counter;
reg [(‘ELMNT_WIDTH - 1):0] read_p; // read pointer
reg [(‘ELMNT_WIDTH - 1):0] write_p; // write pointer

// FIFO
reg [(‘DATA_WIDTH - 1):0] FIFO [0:(‘DATA_DEPTH - 1)];

// wires
wire read = read_in;
wire write = write_in;

// assigns
assign element = counter;
assign data_o = FIFO[read_p];

always @(posedge clk or posedge reset)
begin
if (reset)
begin
read_p <= 2'b11;
write_p <= 2'b00;
counter <= 3'b000;
end
else
begin
if (write && ~full)
write_p <= write_p + 1; // incrementing write pointer
if (read && ~empty)
read_p <= read_p + 1; // incrementing read pointer
if (write && ~read && ~full)
counter <= counter + 1; // incrementing data counter
else if (~write && read && ~empty)
counter <= counter - 1; // incrementing data counter
end
end

// if the fifo is empty
always @(posedge clk or posedge reset)
begin
if (reset)
empty <= 1'b1; // reset clears the data
else
begin
if (empty && write)
empty <= 1'b0; // deasserts empty when there's a write
else if (first && read && ~write)
empty <= 1'b1; // asserts empty
end
end

// if there is one element in the fifo
always @(posedge clk or posedge reset)
begin
if(reset)
    first <= 1'b0;
else
    begin
        if (empty && write || second && read && ~write)
            first <= 1'b1;
        else if (first && write && ~read || first && read && ~write)
            first <= 1'b0;
    end
end

// if there are two elements in the fifo
always @(posedge clk or posedge reset)
begin
if(reset)
    second <= 1'b0;
else
    begin
        if (first && write && ~read || third && ~write && read)
            second <= 1'b1;
        else if (second && read && ~write || second && write && ~read)
            second <= 1'b0;
    end
end

// if there are three elements in the fifo
always @(posedge clk or posedge reset)
begin
if (reset)
    third <= 1'b0;
else
    begin
        if (second && write && ~read || full && read && ~write)
            third <= 1'b1;
        else if (third && read && ~write || third && write && ~read)
            third <= 1'b0;
    end
end

// if the fifo is full
always @ (posedge clk or posedge reset)
begin
if (reset)
    full <= 1'b0;
else
    begin
        if (third && write && ~read)
            full <= 1'b1;
        else if (full && read && ~write)
            full <= 1'b0;
        end
    end
end

// writing and reading
always @ (posedge clk or posedge reset)
begin
    if (reset)
        begin
            FIFO[0] <= 32'b0;
            FIFO[1] <= 32'b0;
            FIFO[2] <= 32'b0;
            FIFO[3] <= 32'b0;
        end
    else if (write & ~full)
        begin
            FIFO[write_p] <= data_i;
        end
end
endmodule

B.1.4 Verilog Source Code - Read-Val-in

module read_val_in (read, clk, reset, done, start, empty, yummy, read_fifo);

    input done;
    input start;
    input empty;
    input reset;
    input clk;
    input yummy;
    input read_fifo;

    output read;

    wire avail;
    wire empty_read;
    wire yummy_check;
    reg empty_check;

endmodule
module pipelinenetwork (dataout, validout, clk, reset, data-in, valid-in);

input [31:0] data_in;
input valid_in;

reg read_new;
reg [1:0] count;

assign avail = ~empty && (done || start);
assign empty_read = empty_check && ~read_new && ~empty;
assign read = ((empty_read || ~empty) && yummy_check) || (avail && done);
assign yummy_check = ~count;

// counts yummy sent and data received
always @ (posedge clk or posedge reset)
begin
    if (reset)
        count <= 2’b0;
    else
        count <= count + read_fifo - yummy;
end

always @ (posedge clk or posedge reset)
begin
    if (reset)
        begin
            empty_check = 1’b1;
            read_new = 1’b0;
        end
    else if (empty)
        begin
            empty_check = 1’b1;
            read_new = 1’b0;
        end
    else if (empty_read)
        begin
            empty_check = 1’b0;
            read_new = 1’b1;
        end
    else
        begin
            empty_check = 1’b0;
            read_new = 1’b0;
        end
end
endmodule

B.1.5 Verilog Source Code - Pipelining

module pipeline_network (data_out, valid_out, clk, reset, data_in, valid_in);

input [31:0] data_in;
input valid_in;
```vhdl
input reset;
input clk;

output [31:0] data_out;
output valid_out;

reg valid_internal;
reg [31:0] data_out;
reg valid_out;

always @ (posedge clk or posedge reset)
begin
if (reset)
begin
valid_out <= 1'b0;
data_out <= 32'b0;
valid_internal <= 1'b0;
end
else
begin
valid_out <= valid_internal;
data_out <= data_in;
valid_internal <= valid_in;
end
end
endmodule

B.2 Network Output Interface

module comp4_asynch (almost_empty, almost_full, data_comp, read, counter, fifo_empty, fifo_full, data_out, valid, n_full, w_clk, r_clk, reset, yummy, write, data_in);

//inputs
input [31:0] data_in;
input write;
input yummy;
input reset;
input w_clk;
input r_clk;

//outputs
output [31:0] data_out;
output valid;
output n_full;
output fifo_empty;
output fifo_full;
```
output [1:0] counter;
output data_comp;
output read;
output almost_empty;
output almost_full;

//internal regs and wires
wire [31:0] data, data_0;
wire read_0;
wire full;
wire empty, empty_0, noread;
assign fifo_empty = empty;
assign fifo_full = full;
assign n_full = ˜full;

asynch_fifo_32x255 fO (.wr_clk(w_clk), .rd_clk(r_clk), .ainit(reset), .din(data_in), .wr_en(write), .rd_en(read_0), .dout(data), .full(full), .empty(empty), .almost_empty(almost_empty), .almost_full(almost_full));

pipeline1 p1_0 (.data_out(data_0), .empty_out(empty_0), .rd_out(read_0), .clk(r clk), .reset(reset), .data_in(data), .rd_in(read), .empty_in(almost_empty));

read_hold rh (.readout(data-comp), .no-read(no-read), .clk(r_clk), .reset(reset), .read_in(read_0), .empty(empty));

network_output no (.counter(counter), .data_out(data_out), .valid(valid), .thanks(read), .clk(r_clk), .reset(reset), .empty(empty_0), .yummy(yummy), .data_comp(data_comp), .no_read(no_read), .data_in(data_0));
endmodule

B.2.1 Verilog Source Code - Output Interface

module network_output (counter, data_out, valid, thanks, clk, reset, empty, yummy, data_comp, no_read, data_in);

    //inputs
    input [31:0] data_in;
    input yummy;
    input empty;
    input reset;
    input clk;
    input data_comp;
    input no_read;

    //outputs
output thanks;
output valid;
output [31:0] data_out;
output [1:0] counter;

//regs and wires
reg [1:0] counter; //holds state
reg [31:0] data_out;
reg valid;
reg thanks;
reg empty_old; //holds state
wire n_yummy;
wire [2:0] statements;
wire empty_old_next;
reg valid_next;
wire [1:0] counter_next;
reg [32:0] data_out_next;
reg thanks_next;

assign n_yummy = ~yummy;
assign statements[0] = empty_old;
assign statements[1] = data_comp;
assign statements[2] = counter[1];
assign counter_next = counter + thanks - yummy - no_read;
assign empty_old_next = empty;

// interface with RAW
always @ (posedge clk or posedge reset)
begin
if (reset)
begin
data_out <= 32'b1;
valid <= 1'b0;
counter <= 2'b0;
empty_old <= 1'b1;
thanks <= 1'b0;
end
else
begin
data_out <= data_out_next;
valid <= valid_next;
counter <= counter_next;
empty_old <= empty_old_next;
thanks <= thanks_next;
end
end
end
always @(statements or data_in or empty)
  begin
  case (statements)
    3'b0: begin
       //when counter < 2'b0
       data_out_next = data_in;
       valid_next = 1'b0;
       thanks_next = !empty;
    end

    3'b010: begin
       //when data_comp
       data_out_next = data_in;
       valid_next = 1'b1;
       thanks_next = !empty;
    end

    3'b110: begin
       data_out_next = data_in;
       valid_next = 1'b1;
       thanks_next = 1'b0;
    end

    default: begin
       data_out_next = data_in;
       valid_next = data_comp;
       thanks_next = 1'b0;
    end
  endcase
  end
endmodule

B.2.2 Verilog Source Code - 255 Element FIFO

//Generated by the Xilinx Core Generator

module asynch_fifo_32x255 ( 
  din,
  wr_en,
  wr_clk,
  rd_en,
  rd_clk,
  ainit,
  dout,
  full,
  empty,
  almost_full,
almost_empty);

input [31 : 0] din;
input wr_en;
input wr_clk;
input rd_en;
input rd_clk;
input ainit;
output [31 : 0] dout;
output full;
output empty;
output almost_full;
output almost_empty;

// synopsys translate_off

ASYNC_FIFO_V4_0 #(  
32,  // c_data_width  
0,   // c_enable_rlocs  
255, // c_fifo_depth  
1,   // c_has_almost_empty  
1,   // c_has_almost_full  
0,   // c_has_rd_ack  
0,   // c_has_rd_count  
0,   // c_has_rd_err  
0,   // c_has_wr_ack  
0,   // c_has_wr_count  
0,   // c_has_wr_err  
0,   // c_rd_ack_low  
2,   // c_rd_count_width  
0,   // c_rd_err_low  
1,   // c_use_blockmem  
0,   // c_wr_ack_low  
2,   // c_wr_count_width  
0)   // c_wr_err_low

inst (  
  .DIN(din),  
  .WR_EN(wr_en),  
  .WR_CLK(wr_clk),  
  .RD_EN(rd_en),  
  .RD_CLK(rd_clk),  
  .AINIT(ainit),  
  .DOUT(dout),  
  .FULL(full),  
  .EMPTY(empty),  
  .ALMOST_FULL(almost_full),  
  .ALMOST_EMPTY(almost_empty),
.WR_COUNT(),
.RD_COUNT(),
.RD_ACK(),
.RD_ERR(),
.WR_ACK(),
.WR_ERR());

// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of async_fifo_32x255 is "true"
endmodule

B.2.3 Verilog Source Code - Read-Val-out

//holds the read

module read_hold (read_out, no_read, clk, reset, read_in, empty);

    //inputs
    input read_in;
    input empty;
    input clk;
    input reset;

    //outputs
    output read_out;
    output no_read;

    reg       temp_0, temp_1;
    reg       read_out, no_read;
    always @ (posedge clk)
        begin
        if (reset)
            begin
            read_out <= 1'b0;
            temp_0 <= 1'b0;
            temp_1 <= 1'b0;
            end
        else
            begin
            temp_0 <= read_in & !empty;
            temp_1 <= read_in & empty;
            no_read <= temp_1;
        end
    end
read_out <= temp_0;
end
endmodule

B.2.4 Verilog Source Code - Pipelining

// pipeline between fifo and network output
// Judy Chen

module pipelinel (data-out, empty-out, rdout, clk, reset, data-in, empty-in, rd_in);

// inputs
input [31:0] data-in;
input empty-in;
input rd-in;
input clk;
input reset;

// outputs
output [31:0] data-out;
output empty-out;
output rd-out;

reg [31:0] data_out;
reg empty_out;
reg rd_out;
always @ (posedge clk)
begin
  case (reset)
    1'b0: begin
      data_out = data_in;
      empty_out = empty_in;
      rd_out = rd_in;
    end
    1'b1: begin
      data_out = 32'b0;
      empty_out = 1'b1;
      rd_out = 1'b0;
    end
  endcase
end
endmodule
Appendix C

Verilog Source Code - Control Unit

C.1 Verilog Source Code - Control Register File

//Generated by Xilinx Core Generator

module register_file_dual_port ( 
    addra, 
    addrb, 
    clka, 
    clkb, 
    dina, 
    dinb, 
    douta, 
    doutb, 
    ena, 
    enb, 
    sinita, 
    sinitb, 
    wea, 
    web);

    input [6 : 0] addra; 
    input [6 : 0] addrb; 
    input clka; 
    input clkb; 
    input [15 : 0] dina; 
    input [15 : 0] dinb; 
    output [15 : 0] douta; 
    output [15 : 0] doutb; 
    input ena;
input enb;
input sinita;
input sinitb;
input wea;
input web;

// synopsys translate_off

BLKMEMDP_V4_0 #(
  7, // c_addra_width
  7, // c_addrb_width
  "0", // c_default_data
  75, // c_depth_a
  75, // c_depth_b
  0, // c_enable_rlocs
  1, // c_has_default_data
  1, // c_has_dina
  1, // c_has_dinb
  1, // c_has_douta
  1, // c_has_doutb
  1, // c_has_ena
  1, // c_has_enb
  0, // c_has_limit_data_pitch
  0, // c_has_nda
  0, // c_has.ndb
  0, // c_has.rdya
  0, // c_has.rdyb
  0, // c_has.rfda
  0, // c_has.rfdb
  1, // c_has.sinita
  1, // c_has.sinitb
  1, // c_has.wea
  1, // c_has_web
  18, // c_limit_data_pitch
  "mif_file_16_1", // c_mem_init_file
  0, // c_pipe_stages_a
  0, // c_pipe_stages_b
  0, // c_reg_inputsa
  0, // c_reg_inputsb
  "0", // c_sinita_value
  "0", // c_sinitb_value
  16, // c_width_a
  16, // c_width_b
  0, // c_write_modea
  0, // c_write_modeb
  "0", // c_ybottom_addr
  1, // c_yclka_is_rising

76
C.2 Verilog Source Code - SPI Control

// determines when to send out data to cdin
module spi_control (format_reg, mic_reg, spi_reg, control_set, clk, reset, control_addr, we);

`define ADDR_WIDTH 7

//inputs
input we;
input [(ADDR_WIDTH - 1):0] control_addr;
input reset;
input clk;

//outputs
output control_set;
output spi_reg;
output mic_reg;
output format_reg;

wire spi_addr, con_addr, format_addr;
reg spi_reg, control_set, mic_reg, format_reg;

assign spi_addr = ~| control_addr[(ADDR_WIDTH - 1):3]; // Check bits above bit 4 are zero
assign con_addr = ~| control_addr[2:0]; // check for zero in lower 3 bits
assign format_addr = ~(control_addr[0] | control_addr[2]) || control_addr[1]; //

always @ (posedge clk or posedge reset)
begin
if (reset)
begin
  spi_reg <= 1'b0;
  control_set <= 1'b0;
  mic_reg <= 1'b0;
  format_reg <= 1'b0;
end //
else if (we)
begin
  control_set <= con_addr & spi_addr;
  spi_reg <= !con_addr & spi_addr;
  mic_reg <= !spi_addr;
  format_reg <= spi_addr & format_addr;
end
else
begin
  spi_reg <= 1'b0;
end

C.3  Verilog Source Code - Control Ready

//Block governing the control register

module control_fsm (ad_rset, mastermic_en, control_out, sregjload, clk, reset,
                     control_set, spi_reg, control_addr, control_in, we, cclk_in);

    //constant defines
    'define CDIN_START 7'b0010000
    'define ADDR_WIDTH 7
    'define DATA_WIDTH 16
    'define CONTROL_WIDTH 24
    'define STATE_WIDTH 2

    //state defines
    'define IDLESTATE 0
    'define SETSTATE 1
    'define REGSTATE 2

    //inputs
    input [(DATA_WIDTH - 1):0] control_in;
    input [(ADDR_WIDTH - 1):0] control_addr;
    input reset;
    input clk;
    input spi_reg;
    input control_set;
    input we;
    input cclk_in;

    //outputs
    output [(CONTROL_WIDTH - 1):0] control_out;
    output sregjload;
    output ad_rset;
    output mastermic_en;

    //state regs
    reg [(CONTROL_WIDTH - 1):0] control_f;
    reg s_load;
    reg ad_rset_f, mastermic_en_f;
reg  spi_reg_start; //c_spi_reg_start_d1;
reg  load_em_0, load_em_1;
//  reg  c_load, c_load_d1;
//  reg  load_clr_d1, clr_spi_reg_start;

//assigns
//assign hi = 1'b1;
assign  control_out = control_f;
assign  s_reg_load = s_load;
assign  master_mic_en = master_mic_en_f;
assign  ad_rset = ~(~ad_rset_f || reset); //low enabled reset f
for the a/d

// Master Control Register (addr 0)
always @ (posedge clk or posedge reset)
begin
if (reset)
begin
    ad_rset_f <= 1'b0;
    master_mic_en_f <= 1'b0;
end  //
else if (control_set)  //
begin
    ad_rset_f <= not control_in[1];  //0' not reset, 1' reset
    master_mic_en_f <= control_in[2];  // 1' all mics on
end  //
end

always @ (posedge clk or posedge reset)
begin
if (reset)
begin
    spi_reg_start <= 1'b0;
    control_f <= 24'b0;  // Don't really need to clear this, but why not
end
else if (spi_reg)
begin
    control_f <= {`CDIN_START, `we, 5'b0, control_addr[2:0], control_in[7:0]};
    spi_reg_start <= 1'b1;  // Load SPI shift register, but first synchronize with cclk domain
end
else
    spi_reg_start <= 1'b0;
//  else if (clr_spi_reg_start)
//    begin

80
// Signal back from cclk domain that this bit was seen and can be cleared
//   spi_reg_start <= 1'b0;
// end
end

// Synchronize SPI shift register load signal into CCLK domain and make it one clock cycle long.
// Don’t need to synchronize the CDIN data, since it will be stable until the Load signal is resynchronized back to the Master CLK
//
// always @(posedge cclk_in or posedge reset)
// begin
//   if (reset)
//     begin
//       c_spi_reg_start_d1 <= 1'b0;
//       c_load <= 1'b0;
//       c_load_d1 <= 1'b0;
//       s_load <= 1'b0;
//     end
//   else
//     begin
//       // Delay spi_reg_start to synchronize with cclk domain
//       c_spi_reg_start_d1 <= spi_reg_start;
//       c_load <= c_spi_reg_start_d1;
//       c_load_d1 <= c_load;
//       // Want a one cclk long pulse on s_load (c_load stays high)
//       // Look for the rising edge by seeing the difference in the delayed signals
//       if ( c_load && !c_load_d1)
//         s_load <= 1'b1;
//       else
//         s_load <= 1'b0;
//     end
// end

// Synchronize c_load from CCLK domain back into clk_fsm domain to clear spi_reg_start
//
// always @(posedge clk or posedge reset)
// begin
//   if (reset)
//     begin

81
// load_clr_d1 <= 1'b0;
// clr_spi_reg_start <= 1'b0;
//
// else
// begin
// load_clr_d1 <= c_load;
// clr_spi_reg_start <= load_clr_d1;
//
// end
// end

// In emulation, the cclk is faster than clk_fsm, which will not be the case later
// synchronize to cclk, and make load one cycle long.
always @ (posedge cclk_in or posedge reset)
begin
  if (reset)
    begin
      s_load <= 1'b0;
      load_em_0 <= 1'b0;
      load_em_1 <= 1'b0;
    end
  else
    begin
      load_em_0 <= spi_reg_start;
      load_em_1 <= load_em_0;
      s_load <= load_em_0 & ~load_em_1;
    end
end
endmodule     //

C.4 Verilog Source Code - Shift Register and Register Done

C.4.1 Shift Register

// Generated by Xilinx Core Generator

module shift_reg (  
  CLK,
  SDOUT,
  P_LOAD,
  D);

input CLK;
output SDOUT;
input P_LOAD;
input [23 : 0] D;

// synopsys translate_off

C_SHIFT_FD_V5_0 #(
  "00000000111111111111111111", // c_ainit_val
  1, // c_enable_rlocs
  0, // c_fill_data
  0, // c_has_aclr
  0, // c_has_ainit
  0, // c_has_aset
  0, // c_has_ce
  1, // c_has_d
  0, // c_has_lsb_2_msb
  0, // c_has_q
  0, // c_has_sclr
  0, // c_has_sdin
  1, // c_has_sdout
  0, // c_has_sinit
  0, // c_has_sset
  0, // c_shift_type
  "00000000000000000000000000", // c_sinit_val
  0, // c_sync_enable
  1, // c_sync_priority
  24) // c_width
inst (
  .CLK(CLK),
  .SDOUT(SDOUT),
  .P_LOAD(P_LOAD),
  .D(D),
  .SDIN(),
  .LSB_2_MSB(),
  .CE(),
  .Q(),
  .SCLR(),
  .SINIT(),
  .SSET(),
  .ACLRO(),
  .AINIT(),
  .ASET());

// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of shift_reg is "true"

class C.4.2 Done Register

// synchronizing data_16 with valid_clk

module data_sync (data_sync, s_clk, valid_clk, reset, data_16);

input s_clk;
input valid_clk;
input reset;
input [15:0] data_16;

output [15:0] data_sync;

reg [15:0] data_internal_0, data_internal_1;
reg [15:0] data_sync;

always @ (posedge s_clk or posedge reset)
begin
if (reset)
begin
    data_internal_0 <= 16'0;
    data_internal_1 <= 16'0;
end
else if (valid_clk)
begin
    data_internal_0 <= data_16;
    data_internal_1 <= data_internal_0;
end
end

always @ (posedge valid_clk or posedge reset)
begin
if (reset)
    data_sync <= 16'0;
else
    data_sync <= data_internal_1;
end
endmodule
Appendix D

Verilog Source Code - Formatting and Packaging Unit

D.1 Verilog Source Code - Data Count

//determines if the serial data is complete

module sdout_done (valid, clk, reset, lrclk);

    input clk;
    input reset;
    input lrclk;

    output valid;

    reg [4:0] counter, counter_next;
    reg valid, valid_next;
    wire [1:0] statements;
    reg data, lrclk_old_0, lrclk_old_1, lrclk_old_2;
    wire old_next, data_next;

    assign statements[0] = counter[4];
    assign statements[1] = lrclk_old_1 ^ lrclk_old_2;
    assign old_next = lrclk;
    assign data_next = lrclk_old_0;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
            begin


85
counter <= 5'b00001;
valid <= 1'b0;
lrclock_old_0 <= 1'b0;
lrclock_old_1 <= 1'b0;
lrclock_old_2 <= 1'b0;
data <= 1'b0;
end
else
begin
  counter <= counter_next;
  valid <= valid_next;
lrclock_old_0 <= old_next;
lrclock_old_1 <= data_next;
lrclock_old_2 <= lrclock_old_1;
data <= data_next;
end
end
always @ (statements or counter)
begin
  case(statements)
  2'b0: begin
    counter_next = counter + 1;
    valid_next = 1'b0;
  end
  2'b01: begin
    counter_next = 5'b1;
    valid_next = 1'b1;
  end
  //
  default: begin
    counter_next = 5'b1;
    valid_next = 1'b0;
  end
  //
  endcase
  end
endmodule

D.2 Verilog Source Code - Data Shift Block

D.2.1 Verilog Source Code - 16-bit Shift Register

//Generated by Xilinx Core Generator
module dout_reg (  
  CLK,  
  SDIN, 
  Q);
input CLK;
input SDIN;
output [15 : 0] Q;

// synopsys translate_off

C_SHIFT_FD_V5_0 #(
  "0000000000000000", // c_ainit_val
  1, // c_enable_rlocs
  5, // c_fill_data
  0, // c_has_aclr
  0, // c_has_ainit
  0, // c_has_aset
  0, // c_has_ce
  0, // c_has_d
  0, // c_has_lsb_2_msb
  1, // c_has_q
  0, // c_has_sclr
  1, // c_has_sdin
  0, // c_has_sdout
  0, // c_has_sinit
  0, // c_has_sset
  0, // c_shift_type
  "0000000000000000", // c_sinit_val
  0, // c_sync_enable
  1, // c_sync_priority
  16) // c_width
inst (n
  .CLK(CLK),
  .SDIN(SDIN),
  .Q(Q),
  .D(),
  .P_LOAD(),
  .LSB_2_MSB(),
  .CE(),
  .SDOUT(),
  .SCLR(),
  .SINIT(),
  .SSET(),
  .ACL(R),
  .AINIT(),
  .ASET());

// synopsys translate_on

// FPGA Express black box declaration
D.2.2 Verilog Source Code - Data Shift Block - Partial

//shifts data into shift registers, gets data from the 16 a/d

module data_shift (data_out, clk, mux_clk, reset, data_in, addr, valid);

'define DATA_WIDTH 24
'define MIC_WIDTH 16

input [(MIC_WIDTH - 1):0] data_in;
input valid;
input reset;
input clk;
ininput mux_clk;
ininput [3:0] addr;

output [(DATA_WIDTH - 1):0] data_out;

wire [(DATA_WIDTH - 1):0] data_internal_0, data_internal_1, data_internal_2, data_internal_3, data_internal_4, data_internal_5, data_internal_6, data_internal_7, data_internal_8, data_internal_9, data_internal_10, data_internal_11, data_internal_12, data_internal_13, data_internal_14, data_internal_15;
wire [(DATA_WIDTH - 1):0] data_out_reg_0, data_out_reg_1, data_out_reg_2, data_out_reg_3, data_out_reg_4, data_out_reg_5, data_out_reg_6, data_out_reg_7, data_out_reg_8, data_out_reg_9, data_out_reg_10, data_out_reg_11, data_out_reg_12, data_out_reg_13, data_out_reg_14, data_out_reg_15;
wire sclk;

wire sclk = clk & go;

data_reg dr0 (.CLK(clk), .SDIN(data_in[0]), .Q(data_internal_0));
data_reg dr1 (.CLK(clk), .SDIN(data_in[1]), .Q(data_internal_1));
data_reg dr2 (.CLK(clk), .SDIN(data_in[2]), .Q(data_internal_2));
data_reg dr3 (.CLK(clk), .SDIN(data_in[3]), .Q(data_internal_3));
data_reg dr4 (.CLK(clk), .SDIN(data_in[4]), .Q(data_internal_4));
data_reg dr5 (.CLK(clk), .SDIN(data_in[5]), .Q(data_internal_5));
data_reg dr6 (.CLK(clk), .SDIN(data_in[6]), .Q(data_internal_6));
data_reg dr7 (.CLK(clk), .SDIN(data_in[7]), .Q(data_internal_7));
data_reg dr8 (.CLK(clk), .SDIN(data_in[8]), .Q(data_internal_8));
data_reg dr9 (.CLK(clk), .SDIN(data_in[9]), .Q(data_internal_9));
data_reg dr10 (.CLK(clk), .SDIN(data_in[10]), .Q(data_internal_10));
data_reg dr11 (.CLK(clk), .SDIN(data_in[11]), .Q(data_internal_11));
data_reg dr12 (.CLK(clk), .SDIN(data_in[12]), .Q(data_internal_12));
data_reg dr13 (.CLK(clk), .SDIN(data_in[13]), .Q(data_internal_13));
data_reg dr14 (.CLK(clk), .SDIN(data_in[14]), .Q(data_internal_14));
data_reg dr15 (.CLK(clk), .SDIN(data_in[15]), .Q(data_internal_15));

pipeline_sdata ps0 (.data_out(data_out_reg_0), .clk(clk), .reset(reset), .data_in(data_internal_0), .en(valid));
    pipeline_sdata ps1 (.data_out(data_out_reg_1), .clk(clk), .reset(reset), .data_in(data_internal_1), .en(valid));
    pipeline_sdata ps2 (.data_out(data_out_reg_2), .clk(clk), .reset(reset), .data_in(data_internal_2), .en(valid));
    pipeline_sdata ps3 (.data_out(data_out_reg_3), .clk(clk), .reset(reset), .data_in(data_internal_3), .en(valid));
    pipeline_sdata ps4 (.data_out(data_out_reg_4), .clk(clk), .reset(reset), .data_in(data_internal_4), .en(valid));
    pipeline_sdata ps5 (.data_out(data_out_reg_5), .clk(clk), .reset(reset), .data_in(data_internal_5), .en(valid));
    pipeline_sdata ps6 (.data_out(data_out_reg_6), .clk(clk), .reset(reset), .data_in(data_internal_6), .en(valid));
    pipeline_sdata ps7 (.data_out(data_out_reg_7), .clk(clk), .reset(reset), .data_in(data_internal_7), .en(valid));
    pipeline_sdata ps8 (.data_out(data_out_reg_8), .clk(clk), .reset(reset), .data_in(data_internal_8), .en(valid));
    pipeline_sdata ps9 (.data_out(data_out_reg_9), .clk(clk), .reset(reset), .data_in(data_internal_9), .en(valid));
    pipeline_sdata ps10 (.data_out(data_out_reg_10), .clk(clk), .reset(reset), .data_in(data_internal_10), .en(valid));
    pipeline_sdata ps11 (.data_out(data_out_reg_11), .clk(clk), .reset(reset), .data_in(data_internal_11), .en(valid));
    pipeline_sdata ps12 (.data_out(data_out_reg_12), .clk(clk), .reset(reset), .data_in(data_internal_12), .en(valid));
    pipeline_sdata ps13 (.data_out(data_out_reg_13), .clk(clk), .reset(reset), .data_in(data_internal_13), .en(valid));
    pipeline_sdata ps14 (.data_out(data_out_reg_14), .clk(clk), .reset(reset), .data_in(data_internal_14), .en(valid));
    pipeline_sdata ps15 (.data_out(data_out_reg_15), .clk(clk), .reset(reset), .data_in(data_internal_15), .en(valid));

mux_16 m16 (.MA(data_out_reg_0), .MB(data_out_reg_1), .MC(data_out_reg_2), .MD(data_out_reg_3), .ME(data_out_reg_4), .MF(data_out_reg_5), .MG(data_out_reg_6), .MH(data_out_reg_7), .MAA(data_out_reg_8), .MAB(data_out_reg_9), .MAC(data_out_reg_10), .MAD(data_out_reg_11), .MAE(data_out_reg_12), .MAF(data_out_reg_13), .MAG(data_out_reg_14), .MAH(data_out_reg_15), .S(addr), .Q(data_out), .CLK(mux_clk));
endmodule
Verilog Source Code - Data Reg (24-bit Shifter)

// Generated by Xilinx Core Generator

module data_reg ( 
    CLK, 
    SDIN, 
    Q); 

input CLK; 
input SDIN; 
output [23 : 0] Q; 

// synopsys translate_off

C_SHIFT_FD_V5_0 #(
    "000000000000000000000000",   // c_ainit_val 
    1,   // c_enable_rlocs 
    5,   // c_fill_data 
    0,   // c_has_aclr 
    0,   // c_has_ainit 
    0,   // c_has_aset 
    0,   // c_has_ce 
    0,   // c_has_d 
    0,   // c_has_lsb_2_msb 
    1,   // c_has_q 
    0,   // c_has_sclr 
    1,   // c_has_sdin 
    0,   // c_has_sdout 
    0,   // c_has_sinit 
    0,   // c_has_sset 
    0,   // c_shift_type 
    "000000000000000000000000",   // c_sinit_val 
    0,   // c_sync_enable 
    1,   // c_sync_priority 
    24)  // c_width 

inst ( 
    .CLK(CLK), 
    .SDIN(SDIN), 
    .Q(Q), 
    .D(), 
    .P_LOAD(), 
    .LSB_2_MSB(), 
    .CE(), 
    .SDOUT(), 
    .SCLR(), 
    .SINIT(), 
    .SSET(), 

90
Verilog Source Code - Parallel Registers

module pipeline_sdata (data_out, clk, reset, data_in, en);

    input [23:0] data_in;
    input   clk;
    input   reset;
    input   en;

    output [23:0] data_out;
    reg [23:0]    data_out;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
            data_out <= 24'b0;
        else if (en)
            data_out <= data_in;
    end
endmodule

Verilog Source Code - 16 input Multiplexer

// Generated by Xilinx Core Generator
module mux_16 (  
    MA, 
    MB, 
    MC, 
    MD, 
    ME, 
    MF, 
    MG, 
    MH, 
    MAA,
MAB,
MAC,
MAD,
MAE,
MAF,
MAG,
MAH,
S,
Q,
CLK);

input [23 : 0] MA;
input [23 : 0] MB;
input [23 : 0] MC;
input [23 : 0] MD;
input [23 : 0] ME;
input [23 : 0] MF;
input [23 : 0] MG;
input [23 : 0] MH;
input [23 : 0] MAA;
input [23 : 0] MAB;
input [23 : 0] MAC;
input [23 : 0] MAD;
input [23 : 0] MAE;
input [23 : 0] MAF;
input [23 : 0] MAG;
input [23 : 0] MAH;
input [3 : 0] S;
output [23 : 0] Q;
input CLK;

// synopsys translate_off

C_MUX_BUS_V5_0 #(  
"000000000000000000000000",  // c_init_val
1,  // c_enable_rlocs
0,  // c_has_aclr
0,  // c_has_ainit
0,  // c_has_aset
0,  // c_has_ce
0,  // c_has_en
0,  // c_has_o
1,  // c_has_q
0,  // c_has_sclr
0,  // c_has_sinit
0,  // c_has_sset
16,  // c_inputs
92
1,    // c_latency
0,    // c_mux_type
4,    // c_sel_width
"00000000000000000000000000",    // c_sinit_val
0,    // c_sync_enable
1,    // c_sync_priority
24)   // c_width

inst (  
  .MA(MA),  
  .MB(MB),  
  .MC(MC),  
  .MD(MD),  
  .ME(ME),  
  .MF(MF),  
  .MG(MG),  
  .MH(MH),  
  .MAA(MAA),  
  .MAB(MAB),  
  .MAC(MAC),  
  .MAD(MAD),  
  .MAE(MAE),  
  .MAF(MAF),  
  .MAG(MAG),  
  .MAH(MAH),  
  .S(S),  
  .Q(Q),  
  .CLK(CLK),  
  .MBH(),  
  .MBG(),  
  .MBF(),  
  .MBE(),  
  .MBD(),  
  .MBC(),  
  .MBB(),  
  .MBA(),  
  .MCH(),  
  .MCG(),  
  .MCF(),  
  .MCE(),  
  .MCD(),  
  .MCC(),  
  .MCB(),  
  .MCA(),  
  .CE(),  
  .ASET(),  
  .SSET(),  
  .EN(),  

93
// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of mux_16 is "true"

endmodule

D.2.3 Verilog Source Code - Multiplexer Selector

module mux_select (we, mux_sel, clk, reset, valid);

    input clk;
    input reset;
    input valid;

    output [3:0] mux_sel;
    output we;

    reg [4:0]    mux_sel_internal;
    reg        we, we_0, valid_0, valid_1, valid_internal;
    wire        done;
    reg        valid_hold;

    assign    done = mux_sel_internal[4];
    assign    mux_sel = mux_sel_internal;

    //synchronizing valid signal with s_clk

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            begin
                valid_0 <= 1'0;
                valid_1 <= 1'0;
                valid_internal <= 1'0;
            end
        else
            begin
                valid_0 <= valid;
            end

    end

endmodule
valid_1 <= valid_0;
valid_internal <= valid_0 && !valid_1;
end
end

always @ (posedge clk or posedge reset)
begin
if (reset)
begin
  muxselinternal <= 5'bx0;
  we <= 1'b0;
  we_0 <= 1'b0;
  valid_hold <= 1'b0;
end
else if (valid_internal)
begin
  muxsel internal <= 5'bx0;
  we_0 <= 1'b1;
  we <= we_0;
  valid_hold <= 1'b1;
end
else if (done)
begin
  muxselinternal <= 5'bx0;
  we_0 <= 1'b0;
  we <= we_0;
  valid_hold <= 1'b0;
end
else if (valid_hold)
begin
  muxselinternal <= muxselinternal + we_0;
  we_0 <= !we_0;
  we <= we_0;
  valid_hold <= valid_hold;
end
else
begin
  muxselinternal <= 5'bx0;
  we_0 <= 1'b0;
  we <= we_0;
  valid_hold <= valid_hold;
end
endmodule
module data-format_revised (valid_hold, valid, clk, reset, lrclk, format, format_go);

    //constant defines
    'define DATA_WIDTH 24
    'define STATE_WIDTH 2

    //state defines
    'define IDLE_STATE 0
    'define LEFT_STATE 1
    'define RIGHT_STATE 2

    input clk;
    input reset;
    input lrclk;
    input format_go;
    input [2:0] format;

    output valid;
    output valid_hold;

    reg [4:0] counter, counter_next;
    reg lr;
    reg lr_0, lr_toggle;
    reg [(STATE_WIDTH - 1):0] state_f, state_next;
    reg valid_hold, valid_hold_next, valid, valid_next;
    wire count_done;

    assign count_done = counter[4] && counter[3];

    always @(format_go)
    begin
        case(format)
            3'b011: lr = 1'b1;
            3'b100: lr = 1'b1;
            3'b101: lr = 1'b1;
            3'b110: lr = 1'b1;
            default: lr = 1'b0;
        endcase
    end

    //state transitions

    always @(state_f or lr or reset or count_done or lr_toggle or counter or format_go or counter)
    begin
        //
state_next = state_f;
valid_next = 1'b0;
counter_next = 5'b0;
valid_hold_next = 1'b0;

case(state_f)
  'IDLE_STATE:
    begin
      if (reset)
        state_next = 'IDLE_STATE;
      else if (format_go)
        begin
          if (lr)
            state_next = 'RIGHT_STATE;
          else
            state_next = 'LEFT_STATE;
        end
    end
  'LEFT_STATE:
    begin
      if (reset)
        state_next = 'IDLE_STATE;
      else if (count_done)
        begin
          counter_next = 5'b1;
          valid_next = 1'b1;
          valid_hold_next = 1'b1;  //signals that a valid has been sent out
        end
      else if (lr_toggle)  //start counting again
        begin
          counter_next = 5'b1;
          valid_hold_next = 1'b0;
          valid_next = 1'b0;
        end
      else if (valid_hold)
        begin
          counter_next = 5'b1;
          valid_next = 1'b0;
          valid_hold_next = 1'b1;
        end
      else
        counter_next = counter + !valid_hold;
        // valid_next = 1'b0;
    end
  'RIGHT_STATE:
begin
    if (reset)
        state_next = 'IDLE_STATE;
    else if (lr_toggle)
        begin
            valid_next = 1'b1;
        end
    else
        valid_next = 1'b0;
    endcase
end

// updating states
always @(posedge clk or posedge reset)
    begin
        if (reset)
            begin
                state_f <= 'IDLE_STATE;
                valid <= 1'b0;
                counter <= 5'b1;
                valid_hold <= 1'b0;
            end
        else
            begin
                state_f <= state_next;
                valid <= valid_next;
                counter <= counter_next;
                valid_hold <= valid_hold_next;
            end
        end
    end

// determines whether or not LRclk toggled
always @(posedge clk or posedge reset)
    begin
        if (reset)
            begin
                lr_0 <= 1'b0;
                lr_toggle <= 1'b0;
            end
        else
            begin
                lr_0 <= lrclk;
                lr_toggle <= lr_0 ^ lrclk;
            end
        end
D.4 Verilog Source Code - Data Write Block

// writing the data into the SRAMs

module data_write (drop_num, sram_we, data_out, sram_addr, clk, reset, data_in, valid, rd_msib, format_go, format);

    // constant defines
    `define ADDR_WIDTH 6
    `define DATA_WIDTH 24
    `define DROP_WIDTH 12
    `define DATA_16_BIT 16
    `define DATA_18_BIT 18
    `define DATA_20_BIT 20
    `define NUM_WIDTH 1

    input [('DATA_WIDTH - 1):0] data_in;
    input clk;
    input reset;
    input valid;
    input rd_msib; // determines how much of the data is sent to RAW;
    input format_go;
    input [2:0] format;

    output [('ADDR_WIDTH - 1):0] sram_addr;
    output sram_we;
    output [('DATA_WIDTH - 1):0] data_out;
    output [('DROP_WIDTH - 1):0] drop_num;

    // state registers
    reg msb, we;
    reg [('ADDR_WIDTH - 2):0] addr_0, addr; //
    reg [('DATA_WIDTH - 1):0] data_internal, data_temp;
    reg [('DROP_WIDTH - 1):0] drop_num;
    wire done, msb_next, drop;

    assign done = & addr;
    assign msb_next = ~rd_msib;
    assign drop = rd_msib ^ msb;
    assign sram_addr = {msb, addr};
    assign sram_we = we;
    assign data_out = data_internal;
always @(format_go or data_in)
begin
  case (format)
    3'b011: data_temp = {8'b0, data_in['DATA_16_BIT - 1):0]};
    3'b100: data_temp = data_in;
    3'b101: data_temp = {6'b0, data_in['DATA_18_BIT - 1):0]};
    3'b110: data_temp = {4'b0, data_in['DATA_20_BIT - 1):0]};
    default: data_temp = data_in;
  endcase
end

// writing into the sram
always @(posedge clk or posedge reset)
begin
  if (reset)
  begin
    msb <= 1'b0;
    addr <= 5'b0;
    addr_0 <= 5'b0;
    we <= 1'b0;
    data_internal <= 24'b0;
    drop_num <= 12'b0;
  end
  else if (done)
  begin
    addr <= 5'b0;
    addr_0 <= 5'b0;
    we <= valid;
    msb <= msb_next;
    data_internal <= data_temp;
    if (drop)
      drop_num <= drop_num + 1;
    else
      drop_num <= 12'b0;
  end
  else if (valid)
  begin
    addr_0 <= addr_0 + 1;
    addr <= addr_0;
    we <= 1'b1;
    data_internal <= data_temp;
  end
  else
  begin
    we <= 1'b0;
    data_internal <= 24'b0;
  end
end
D.5 Verilog Source Code - SRAM

//Generated by Xilinx Core Generator

module data_sram (  
    addra,  
    addrb,  
    clka,  
    clkb,  
    dina,  
    dinb,  
    douta,  
    doutb,  
    ena,  
    enb,  
    wea,  
    web);

input [5 : 0] addra;  
input [5 : 0] addrb;  
inpu t clka;  
in pu t clkb;  
inpu t [23 : 0] dina;  
inpu t [23 : 0] dinb;  
output [23 : 0] douta;  
output [23 : 0] doutb;  
inpu t ena;  
inpu t enb;  
inpu t wea;  
inpu t web;

// synopsys translate_off

BLKMEMDP_V4_0 #(  
  6,   // c_addra_width  
  6,   // c_addrb_width  
"0",  // c_default_data  
  64,  // c_depth_a  
  64,  // c_depth_b  
  0,   // c_enable_rlocs  
  1,   // c_has_default_data  
  1,   // c_has_dina
inst(
    .ADDRA(addra),
    .ADDRB(addrb),
    .CLKA(clka),
    .CLKB(clkb),
    "mif_file_16_1", // c_mem_init_file
    "0", // c_reg_inputsa
    "0", // c_reg_inputsb
    "0", // c_sinita_value
    "0", // c_sinitb_value
    "0", // c_ybottom_addr
    "0", // c_yclka_is_rising
    "0", // c_yclkb_is_rising
    "0", // c_yena_is_high
    "0", // c_yenb_is_high
    "1024", // c_ytop_addr
    "4kx1", // c_yprimitive_type
    "1024", // c_yuse_single_primitive
    "1", // c_ywea_is_high
    "1", // c_yweb_is_high
)
DINA(dina),
DINB(dinb),
DOUTA(douta),
DOUTB(doutb),
ENA(ena),
ENB(enb),
WEA(wea),
WEB(web),
NDA(),
NDB(),
RFDA(),
RFDB(),
RDYA(),
RDYB(),
SINITA(),
SINITB();

// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of data_sram is "true"

endmodule

D.6 Verilog Source Code - Packaging Component

//packaging data for RAW
module package (en_next, data_out, reg_addr, mic_addr, write, clk, reset, reg_data_in, mic_data_in, data_addr, dropped, master_mic, full, header);

  //constants
  `define ZEROS 8'b0
  `define ONES 9'b111111111
  `define MIC_NUM_ADDR 8
  `define MIC_ADDR_WIDTH 6
  `define REG_ADDR_WIDTH 7
  `define DATA_RAW_WIDTH 32
  `define DATA_RAM_WIDTH 24
  `define DATA_REG_WIDTH 16
  `define DROPPED_WIDTH 12

  //states
  `define IDLE_STATE 0
  `define HEADER_STATE 1
'define DRP_WAIT_STATE 2
'define DROPPED_STATE 3
'define MIC_0_STATE 4
'define MIC_WAIT_STATE 5
'define MIC_STATE 6
'define WAIT_STATE 7
'define DONE_STATE 8
'define HDR_WAIT_STATE 9

//inputs
input [(DATA_RAM_WIDTH - 1):0] mic_data_in; //data from microphones
input [(DATA_REG_WIDTH - 1):0] reg_data_in; //data from register file - m
microphone enables,
//dropped packets, length
input [(MIC_ADDR_WIDTH - 1):0] data_addr; //address that current microphone

//data is written into
input [(DROPPED_WIDTH - 1):0] dropped; //the number of package dropped

input clk;
input reset;
input master_mic; //turns on and off mics
input full; //determines whether or not the e FIFO from comp4 is full

//output
output [(DATA_RAM_WIDTH - 1):0] data_out; //data being sent out to RA
AW interface
output [(MIC_ADDR_WIDTH - 1):0] mic_addr; //address of microphone sra
m
output [(REG_ADDR_WIDTH - 1):0] reg_addr; //address of the register f

output write; //writes the data into the output fi

output en_next;

reg we_en, we_en_next, we, we_next;
wire done, done_next, next, done_single;
wire addr_check, reg_next;
reg [31:0] data_out, data_next;
reg [4:0] bit_count, bit_next;
reg drp, dne, hdr;
wire [3:0] bit;
reg [5:0] mic_addr;
reg [4:0] mic_addr_next;
reg [6:0] reg_addr, reg_addr_next;
reg [3:0] state_f, state_next;
reg msb, msb_next, bit_0, bit_reg, wait_count, wait_count_t_next;

assign addr_check = (mic_addr == data_addr); // || (mic_addr + 1 == data_addr); // to ensure that only the current data will be read
assign done = & mic_addr_next;
assign write = we || (we_en && !addr_check);
assign reg_next = bit_reg;
assign done_next = done && header;
assign done_single = bit_reg;
assign next = !(addr_check || full);
assign bit = bit_next[3:0];
assign en_next = reg_data_in[bit];

always @(posedge clk or posedge reset)
begin
    if (reset)
    begin
        state_f <= 'IDLE_STATE;
data_out <= 32'b0;
mic_addr <= 6'b111111;
reg_addr <= 7'b001000;
bit_count <= 5'b0;
we_en <= 1'b0;
we <= 1'b0;
msb <= 1'b0;
wait_count <= 1'b0;
    end //
else if (!master_mic)
begin
    state_f <= state_next;
data_out <= 32'b0;
mic_addr <= 6'b0;
reg_addr <= 7'b0;
bit_count <= 5'b0;
we_en <= 1'b0;
we <= 1'b0;
msb <= msb_next;
wait_count <= 1'b0;
end
else
begin
    state_f <= state_next;
data_out <= data_next;
mic_addr <= {msb, mic_addr_next};
reg_addr <= reg_addr_next;
bit_count <= bit_next;
we <= we_next;
we_en <= we_en_next;
msb <= msb_next;
wait_count <= wait_count_next;
end //
end

always @ (state_f or reset or mic_addr or full or next or header or done_next
or wait_count or we or en_next)
begin

data_next <= 32'b0;
mic_addr_next <= mic_addr[4:0];
bit_next <= bit_count;
reg_addr_next = reg_addr;
we_en_next <= 1'b0;
state_next <= state_f;
msb_next <= msb;
wait_count_next <= 1'b0;
we_next <= 1'b0;

case(state_f)
  'IDLE_STATE:
    begin
      data_next <= 32'b0;
mic_addr_next <= 5'b0;//
      bit_next <= 0;
      reg_addr_next = 7'b0;
we_en_next <= 1'b0;
we_next <= 1'b0;  //
      drp <= 1'b0;
dne <= 1'b0;
hdr <= 1'b0;
        if (reset)
        state_next <= 'IDLE_STATE;
        else if (header)
        state_next <= 'HEADER_STATE;
        else
        state_next <= 'MIC_STATE;
          end

  'HEADER_STATE:
    begin
      data_next <= {{{7{1'b0}}, {1{1'b1}}, {24{1'b0}}}};
mic_addr_next <= 5'b0;
      bit_next <= 0;
      reg_addr_next = 7'b00100;
we_next <= !full;

end
we_en_next <= 1'b0;
msb_next <= msb;
if (reset)
  state_next <= 'IDLE_STATE;
else if (full)
  state_next <= 'HEADER_STATE;
else
  state_next <= 'DRP_WAIT_STATE;
end

'DRP_WAIT_STATE:  //waits to ensure up to date fifo information
begin
  data_next <= 32'b0;
  we_en_next <= 1'b0;
  we_next <= 1'b0;
  wait_count_next <= 1'b0;
  if (reset)
    state_next <= 'IDLE_STATE;
  else
    state_next <= 'DROPPED_STATE;
end

'DROPPED_STATE:
begin
  data_next <= dropped;
  mic_addr_next <= 5'b0;
  bit_next <= 0;
  reg_addr_next = 7'b001000;
  we_en_next <= 1'b0;
  we_next <= !full;
  wait_count_next <= 1'b0;
  if (reset)
    state_next <= 'IDLE_STATE;
  else if (full)
    state_next <= 'DROPPED_STATE;
  else
    state_next <= 'MIC_0_STATE;
end

'MIC_0_STATE:
begin
  data_next <= {'ZEROS, mic_data_in};
  we_next <= 1'b0;
  if (reset)
    begin
      state_next <= 'IDLE_STATE;
      we_en_next <= 1'b0;
    end

end
else
begin
    state_next <= 'MIC_WAIT_STATE;
    we_en_next <= en_next;
end
end

'MIC_WAIT_STATE: //waits to ensure up to date fifo information
begin
    data_next <= {'ZEROS, mic_data_in};
    we_next <= 1'b0;
    reg_addr_next <= reg_addr + (reg_next && !full);
if (reset)
begin
    state_next <= 'IDLE_STATE;
    we_en_next <= 1'b0;
end
else if (done_single)
begin
    state_next <= 'WAIT_STATE;
    we_en_next <= 1'b0;
end
else
begin
    state_next <= 'MIC_STATE;
    we_en_next <= 1'b0;
end
end

'MIC_STATE:
begin
    msb_next <= msb;
    we_next <= 1'b0;
    data_next <= {'ZEROS, mic_data_in};
    mic_addr_next <= mic_addr[4:0] + next;
    bit_next <= bit_count + next;
    reg_addr_next <= reg_addr + (reg_next && !full);
    wait_count_next <= 1'b0;
if (reset)
state_next <= 'IDLE_STATE;
else if (full)
begin
    state_next <= 'MIC_STATE;
    we_en_next <= 1'b0;
end
else if (done_next)
begin
    state_next <= 'DONE_STATE;
    we_en_next <= en_next;
end
else if (done_single)
begin
    state_next <= 'WAIT_STATE;
    we_en_next <= en_next;
end
else
begin
    state_next <= 'MIC_WAIT_STATE;
    we_en_next <= en_next;
end
end

'WAIT_STATE: //waits for the new enable data to be ready
begin
    data_next <= 32'b0;
    we_en_next <= 1'b0;
    we_next <= 1'b0;
    wait_count_next <= !wait_count;
    if (reset)
        state_next <= 'IDLE_STATE;
    else if (wait_count)
        state_next <= 'MIC_STATE;
    else
        state_next <= 'WAIT_STATE;
end

'DONE_STATE:
begin
    data_next <= {{6{1'b0}}, {1{1'b1}}, {25{1'b0}}};
    mic_addr_next <= 6'b0;
    bit_next <= 0;
    reg_addr_next = 7'b001000;
    we_en_next <= 1'b0;
    msb_next <= !msb;
    if (reset) //
        state_next <= 'IDLE_STATE;
    else if (full)
        begin
            state_next <= 'DONE_STATE;
            we_next <= 1'b0;
        end
    else
        begin

109
module pipeline_mux (valid_p, clk, reset, valid);

  input valid;
  input clk;
  input reset;

  output valid_p;
  reg       valid_p;

D.7  Verilog Source Code - Various Pipelining

D.7.1  Pipelining - Mux Select Enable

module pipeline_mux (valid_p, clk, reset, valid);

  input valid;
  input clk;
  input reset;

  output valid_p;
  reg       valid_p;

  always @(posedge clk or posedge reset)
    begin
      if (reset)
        begin
          bit_0 <= 1'b0;
          bit_reg <= 1'b0;
        end //
      else
        begin
          bit_0 <= bit_next[4];
          bit_reg <= bit_0 ^ bit_next[4];
        end //
    end
always @ (posedge clk or posedge reset)
  begin
  if (reset)
    valid_p <= 1'b0;
  else
    valid_p <= valid;
  end
endmodule

D.7.2 Pipelining - Microphone Enables

module pipeline_enable (enable_p, clk, reset, enable);

  input [15:0] enable;
  input    clk;
  input    reset;

  output [15:0] enable_p;

  reg [15:0] enable_p;

always @ (posedge clk or posedge reset)
  begin
  if (reset)
    begin
      enable_p <= 16'b0;
    end
  else
    begin
      enable_p <= enable;
    end
  end
endmodule

D.7.3 Pipelining - Data to Network Interface

//pipelining

module pipeline_data (data_mic_p, valid_mic_p, clk, reset, valid_mic, data_mic);

  input [31:0] data_mic;
  input    valid_mic;
  input    clk;
  input    reset;

  output [31:0] data_mic_p;
  output    valid_mic_p;
reg [31:0]    data_mic_p;
reg    valid_mic_p;

always @ (posedge clk or posedge reset)
  begin
  if (reset)
    begin
      data_mic_p <= 32'b0;
      valid_mic_p <= 1'b0;
    end
  else
    begin
      data_mic_p <= data_mic;
      valid_mic_p <= valid_mic;
    end
  end
endmodule
Bibliography


