Design and Verification of an Application Specific Integrated Circuit (ASIC)

Senior Project I



Kevin Cao, Whitley Forman, Dhruvit Naik,

Zachary Nelson\* and Julie Swift

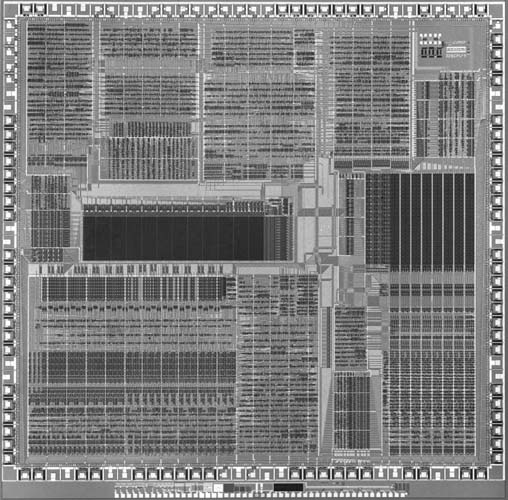
\*Team Leader

Dr. Orlando Hernandez and Dr. Larry A. Pearlstein

December 2015

Design and Verification of an Application Specific Integrated Circuit (ASIC)

**Fulfillment Page**



Submitted to the Faculty of the Electrical and Computer Engineering Department

of The College of New Jersey

Written and Researched by:

Kevin Cao

Whitley Forman

Dhruvit Naik

Zachary Nelson

Julie Swift

In partial fulfillment of the Bachelor of Science degree in Electrical and Computer Engineering

**Abstract**

Very-large-scale integration (VLSI) refers to electronic chips that contain from hundreds of thousands, to billions, of transistors. The goal of this project is to gain experience in VLSI design by creating a chip that will process digital streaming audio data. More specifically, we will implement a 512-tap digital finite impulse response (FIR) filter, which will be applied to an input stream in order to create an output stream. We will be using the I2C protocol to allow a host to control the chip and the serial I2S protocol for transferring digital audio streams.

The chip design was modularized into five blocks: I2S input interface, I2S output interface, filter, register, and I2C interface. Each member was assigned one of these blocks and given the tasks of creating documentation for the block, coding it, and testing it. Each person created a document that would hold all information about their block and updated it when any changes were made. Hardware designs were represented using Verilog register-transfer level (RTL) code. To date, development has been done using Xilinx ISE Design Suite 14.7. Test benches were also designed and implemented using Verilog. A large portion of the RTL design has been completed and tested. Some additional Verilog coding work remains to be done. We have created a top-level module that contains all the correct interface signals between each team members’ blocks, but the internal workings of each block need to be completed. The end goals of the project are to implement the design on a field-programmable gate array (FPGA) for testing purposes and then to transfer our design to MOSIS for fabrication.Finally, we will bring up our design with a realistic environment including an audio source, audio sink, and a microcontroller for reading and writing registers.

**Keywords:** Application-specific integrated circuit (ASIC), Very large scale integration (VLSI) I2S, I2C, Digital filtering

**Table of Contents**

List of Tables 4

List of Figures 5

Nomenclature 6

Introduction 7

Team Management 9

Specifications 10

Chapter 1: Background 11

Chapter 2: I2S Interface 13

Chapter 3: Digital Filtering 26

Chapter 4: Register Block 33

Chapter 5: I2C Interface 41

Chapter 6: Budget 55

Chapter 7: Schedule 56

Chapter 8: Conclusion 57

References 58

Appendix A: Project Overview 59

Appendix B: Management 61

Appendix C: Code 81

Appendix D: Test Benches 125

Appendix E: Industry Specifications 125

**List of Tables**

Table 1.1: Comparison of MOSIS Academic Account Types 12

Table 2.1: Interface Signals for I2S Input 16

Table 2.2: Interface Signals for Synchronizer Sub-Module 17

Table 2.3: Interface Signals for Deserializer Sub-Module 18

Table 2.4: Interface Signals for BIST Generator Sub-Module 19

Table 2.5: Interface Signals for Multiplexer Sub-Module 20

Table 2.6: Interface Signals for FIFO Sub-Module 21

Table 2.7: Interface Signals for I2S Output 23

Table 2.8: Interface Signals for Serialize Sub-Module 24

Table 3.1: Interface Signals for Filter 28

Table 3.2: Interface Signals for Filter Finite-State Machine 30

Table 3.3: Interface Signals for Filter Storage Module 32

Table 3.4: Interface Signals for Filter Barrel Shifter Module 32

Table 4.1: Register Mappings 35

Table 5.1: I2C Register Table 46

**List of Figures**

Figure I.1: Process of Designing a System on a Chip 7

Figure 1.1: Multi-Project Wafer 11

Figure 1.2: Top Level Drawing of Chip 11

Figure 2.1: Overall Chip Diagram with I2S Interfaces Highlighted 14

Figure 2.2: I2S Input Block Diagram 15

Figure 2.3: Sample of Deserializer Simulation 18

Figure 2.4: Sample of BIST Generator Simulation 19

Figure 2.5: Sample of FIFO Simulation 21

Figure 2.6: I2S Output Block Diagram 22

Figure 2.7: Sample of Serializer Simulation 25

Figure 3.1: Overall Chip Diagram with Filter Highlighted 27

Figure 3.2: Filter Block Diagram 27

Figure 3.3: Detailed Block Diagram of Filter 29

Figure 3.4: Finite State Machine - States and Transitions 30

Figure 4.1: Overall Chip Diagram with Register Highlighted 33

Figure 4.2: Top Level Interface Design of Register 34

Figure 4.3: The Start of the Test Fixture Simulation 34

Figure 4.4: Bits trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr are Triggered 39

Figure 5.1: I2C Chip Diagram Highlight 41

Figure 5.2: I2C Full Coefficient Time Load Calculation 42

Figure 5.3: I2C Block Diagram Input 44

Figure 5.4: Open Drain Schematic 44

Figure 5.5: Previous Block Diagram Version 46

Figure 5.6: I2C State Machine Flow Diagram 48

Figure 5.7: I2C Deserializer Block Diagram 49

Figure 5.8: I2C Sequence Block Diagram 50

Figure 5.9: I2C Serializer Block Diagram 51

**Acronyms**

**ASIC**: Application Specific Integrated Chip

**BIST**: Built in Self-Test

**CIF**: Caltech Intermediate Form

**DIP**: Dual in-line package

**EDA**: Electronic Design Automation

**FPGA**: Field Programmable Gate Architecture

**FSM**: Finite-State Machine

**GDSII**: Graphic Data System II

**I2C**: Inter-Integrated Circuit

**I2S**: Integrated Interchip Sound

**IC**: Integrated Circuit

**LSB**: Least Significant Bit

**MEP**: MOSIS Educational Program

**MPW**: Multi-Project Wafer

**MSB**: Most Significant Bit

**OCP**: Open Cavity Plastic

**PCB**: Printed Circuit Board

**RO**: Read Only

**RTL**: Register-Transfer Level

**RTR**: Ready to Receive

**RTS**: Ready to Send

**RW**: Read/Write

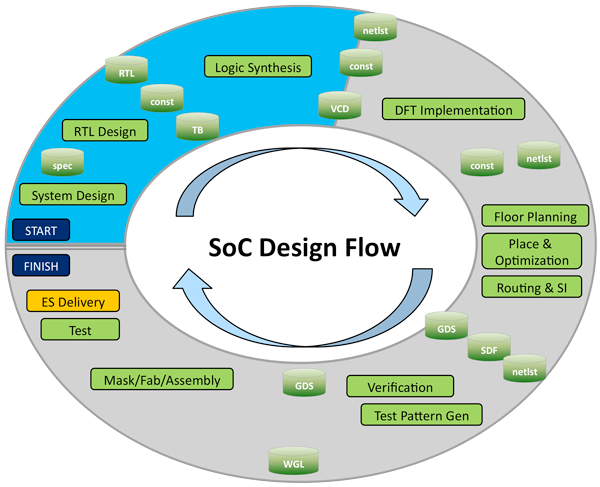
**WO**: Write Only

**XFC**: Transfer Complete

**VLSI:** Very Large Scale Integration

**Introduction**

The goal of this project is to produce a chip that is capable of processing digital streaming audio data and apply a 512-tap filter to the input stream. The design will be implemented on a Nexys 4 Artix-7 FPGA Board and later submitted to MOSIS for the fabrication of an application-specific integrated circuit (ASIC). It should be noted that the terms VLSI and ASIC will be used interchangabley throughout the report and refer to the design of the chip. We will mostly be using the FPGA as a way to verify that our design works correctly before sending it to MOSIS. In industry, some of the advantages of an FPGA are it can be reprogrammed, there is no manufacturing involved, and it is as simple as just downloading code onto a board. Some of the advantages of an ASIC include being a fully custom design so that the device is manufactured to the designer’s specifications, a lower unit cost when producing in high volume, and a smaller chip since there are no extras included. We chose our application to be audio processing because digital audio filtering has a wide range of real world applications and it would be nice to hear an audio signal to show our chip is working. The chip’s application is relatively simple because the bulk of the time is to be spent going through the chip design process since this is the first chip any of the students have designed.



**Fig I.1:** Process of Designing a System on a Chip

Even though we will not be following every step, Fig. I.1 shows the general process for designing a chip. The first task is system design where the requirements of the system are documented and the major functionality of the chip is designed. The second task is RTL design which involves writing code in a hardware description languages (HDL) such as Verilog in order to create a high-level representation of the chip. The next step is logic synthesis which involves turning the RTL design into a netlist of logic gates. This is done by an Electronic Design Automation (EDA) tool and we have primarily been using Xilinx ISE Design Suite 14.7 but plan on using the more powerful Mentor Graphics tool. The next step that our project will be doing is floor planning which is determining the location of the major blocks on the IC schematic. Place and optimization is placing all of the electronic components, circuitry, and logic elements into a limited amount of space in an effective manner. Routing is connecting all of the different components and most follow the rules and limitations of the fabrication process. This step is mostly done automatically by an EDA tool but sometimes has to be done manually. It should be noted that testing and verification will be performed throughout the entire process. The last step that we will be performing is submitting our verified design to be fabricated.

The report will first discuss the team management and how files were stored. Next, the specifications of the entire chip and each individual block will be listed. Chapter 1 talks about background information on MOSIS and the fabrication processes they offer. Chapters 2 to 5 were each written by a different team member describing the block they worked on. Chapter 6 discusses the budget in terms of software and hardware expenses. Chapter 7 describes the schedule that was developed and the current status of the project. Chapter 8 includes the concluding remarks about the paper.

**Team Management**

Zachary Nelson was designated as the team leader and is responsible for documenting the requirements in CORE 9 University, creating the schedule, keeping track of the budget, organizing presentations, updating the website, and performing other administrative tasks. The way the team split up the tasks was that each individual was put in charge of designing, coding, and testing a specific module of the project. Kevin Cao was put in charge of the I2S input and output interfaces, Dhruvit Naik was put in charge of the digital filtering, Julie Swift was put in charge of the chip’s register, and Whitley Forman was put in charge of the I2C interface. Zachary also contributed to the I2S Input Interface and was assigned with integrating all of the different modules into one top-level module. Other tasks were assigned to some of the team members as needed. For example, Dhruvit and Julie were assigned with the installation of the Electronic Design Automation (EDA) tools. It was expected that the team members that did not have any extra tasks assigned to them to contribute all of their senior project time to their specific module.

Another aspect of project management was how we stored source code and other documentation. We used Git/GitHub for storing our source code and non-confidential documents. All confidential files were stored on a separate Dropbox account. The following folders were created on the GitHub repository and a short description is provided:

* core (all CORE 9 Univeristy files)
* docs (any documentation about the project)
* dv (design verification)
* reg (register mapping)
* rtl (register transfer level)
* synth (synthesis)
* utils (utilities)

Using GitHub was extremely important to our project this semester because it enabled us to recover old versions of source code and allowed multiple team members to work on the same source code at the same time.

**Specifications**

**General:**

* Maximum Clock Rate: 100 MHz
* Clock frequency will be a minimum of 1200 times the audio sampling rate

**I2S:**

* I2S input and output interfaces comply with I2S standard, included here in Appendix E
* Maximum Serial Clock Rate: 1.44 MHz
* Audio input sample rates ranges of 8 kilosamples/sec - 48 kilosamples/sec
* Digital audio bit clock and word select lines will be controlled from I2S master, which can be our chip, or the external I2S source
* Audio input and output must have same sample rate – our chip will be the timebase master on the output I2S interface
* Audio input and output will be two 16 bit channels (i.e. one stereo pair)

**Filter:**

* Filter Design: FIR Filter
* 512-tap filter, 16-bit coefficients, all indecently settable
* Programmable filter coefficients to achieve different filter types
* Maintain integer headroom of 4 bits ????

**I2C:**

* 7-bit slave address space
* 12-bit register address space
* Data transfer rate of up to 400 kbit/second desired (both I2C standard and fast modes supported)
* Slave only capability
* Write operation (Burst write functionality)
* Read operation (Single read, optional burst read request)
* User selectable slave address with strap pins
* Simple strobe interface to register block (read and write)

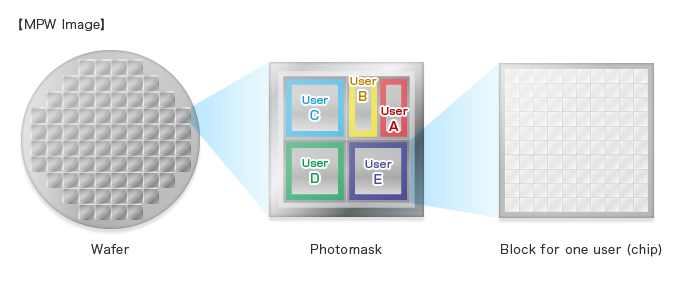
**Register:**

* Simple strobe interface to I2C block
* Provides read/write access to control/status registers, including the following
  + Source select bit (I2S vs. BIST)
  + Filter bypass bit (pass through or filter input streams)
  + 512 16-bit filter coefficients
  + Overflow/saturation detector
  + Audio FIFO overrun/underrun

**Chapter 1: Background - Zachary Nelson**

*1.1 Who is MOSIS?*

Our Application Specific Integrated Circuit (ASIC) will be produced by MOSIS, at no cost to The College of New Jersey. MOSIS was the first well-known multi-project wafer (MPW) service, and was established by DARPA (Defense Advanced Research Projects Agency) in 1981. The acronym MOSIS stands for Metal Oxide Semiconductor Implementation Service and the company has processed over 60,000 IC designs over the last 30 years [1]. The MOSIS Service is known for MPWS, which is how they are able to keep the cost of fabrication low. A MPW is when multiple IC designs are shared on a single wafer and an illustration of this concept in shown in Figure 1.1. This means that designs from private companies and students could be on the same wafer. The reasoning behind this approach is that the cost of fabrication can be kept at a reasonable price if the cost of mask making, wafer fabrication and assembly are shared throughout multiple projects. This idea of a MPW is also attractive because designers can create a prototype of their design without making a huge investment. It should also be noted that MOSIS offers using a single project for a wafer (dedicated run) for all processes and can start the fabrication at any time.



**Fig 1.1:** Multi-Project Wafer

*1.2 Fabrication Process*

MOSIS offers three different options for an accredited college or university to use. These options are registering for a Commercial account, a MOSIS Educational Program (MEP) Instructional account, or a MOSIS Educational Program (MEP) Research account. A comparison of these three accounts is listed on MOSIS’s website and is also shown in Table 1.1. Our advisors registered for a MEP Instructional account and we will either be using a GlobalFoundaries 180 nmCMOS (7HV) process or GlobalFoundaries 180 nm(7RF) process depending on the availability.

**Table 1.1:** Comparison of MOSIS Academic Account Types

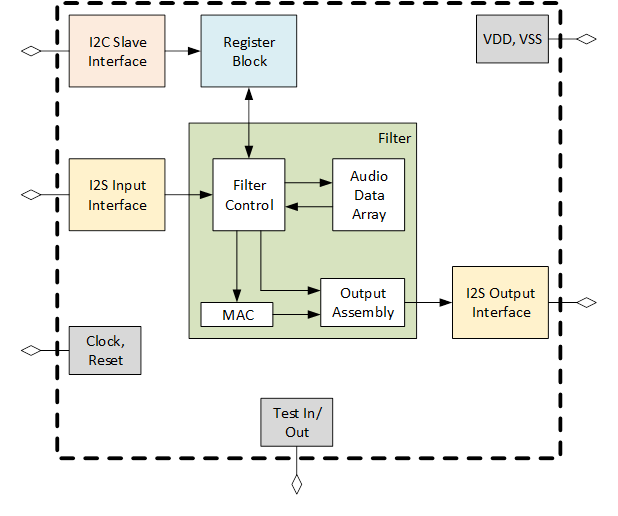
|  |  |  |  |
| --- | --- | --- | --- |
| **Topic** | **Commercial** | **MEP Instructional** | **MEP Research** |
| Primary Purpose | Customers pays for fabrication and packaging | Classroom instruction | Unfunded research |
| Available Processes | All | 1. ON Semi 0.50CMOS (C5N)  2. GlobalFoundaries 180 *nm* CMOS (7HV) | 1. ON Semi 0.50CMOS (C5N)  2. GlobalFoundaries 130 *nm* SiGe BICMOS (8HP)  3. GlobalFoundaries 130 *nm* CMOS (8RF-DM) |
| Size Limits | No restrictions | 5 deliverable parts of a project no larger than 1.5 mm x 1.5 mm | Less than 16 |
| Number of Submissions per Year | Unlimited | Annual request subject to review | One submission per academic year per institution after approval of proposal |
| Run Restrictions | All MPW runs except for MEP-only | MEP-only and space available for COM runs | Space available for COM runs |
| IP Access through MOSIS | Yes | No | Yes |
| Fabrication Costs | Customer pays fabrication cost | Free | Free |
| Packaging | No restrictions, customer pays | Free ceramic and OCP packaging; lids cannot be sealed. Fully encapsulated packaging not available. | No restrictions, customer pays |

*1.3 Submitting a Design to MOSIS and Due Date*

When submitting a MPW run, MOSIS has specified certain steps that need to be taken in order to submit a design. The first step is to submit a new project request by logging onto their website. The designer then needs to assign their Export Control Classification Number (ECCN) before they make a fabrication request. Next, a fabrication request needs to be made specifying the process that will be used. The last step is to submit that actual design layout to MOSIS in either Caltech Intermediate Form (CIF) or Graphic Data System II (GSDII) format. Both of these formats are used to describe the layout of the integrated circuit and will be generated by an EDA tool.According to MOSIS’s 2016 Fabrication Schedule, the customer submission date for the process we will be using is March 7th, 2016. MOSIS states on their website that designs can still be submitted after this but there is no guarantee they will be able to fabricate the design.

*1.4 Top Level Block Diagram*

A top level block diagram of the chip is shown in Figure 1.2. The chip was broken down into five major modules. These modules included the I2S input interface, I2S output interface, filter, register block, and I2C slave interface.

**

**Fig 1.2:** Top Level Drawing of Chip

The main purpose of the I2S input interface was to convert a serial audio data stream into 16-bit parallel data. This module also included a Built in Self Test (BIST) feature that would use a predefined sawtooth signal as the audio data stream. After the audio stream was converted into parallel data, it was put through a 512-tap FIR filter in the filter module. The filter audio signal was then converted back to the I2S serial standard in the I2S output interface. The combination of these three moudles was how data flowed within our chip.

**Data Flow:** I2S Input Interface 🡪 Filter 🡪 I2S Output Interface

The control flow of the chip started with the I2C slave interface module reading or writing the filter coefficient values to the chip. The register block is where the actual fitler coefficient values were stored along with other information such as chip info, control bits, and status bits. The filter block would then read the 512 filter coefficents from the register block in order to implement the FIR filter.

**Control Flow:** I2C Slave Interface 🡪 Register Block 🡪 Filter

**Chapter 2: I2S Interface - Kevin Cao**

*2.1 Introduction*

I2S stands for Integrated Interchip Sound. It is an electrical serial bus interface standard that is used for connecting digital audio devices together. I2S is used to communicate PCM audio data between integrated circuits in an electronic device. PCM stands pulse code modulation and is a standard form of digital audio in computers. It is a method used to digitally represent sampled analog signals. I2S separates clock and serial data signals and lowers jitter that is typical of communication systems that recover the clock from the data stream. Jitter is the deviation from true periodicity of a presumed periodic signal.

*2.2 I2S Bus Specification*

The I2S protocol contains three lines: serial bus, a bit clock, word clock, and a two time multiplexed data line These three lines are used to reduce the number of pins required and to keep wiring simple. Our bit clock is called the serial clock (sck), word clock is word select (ws), and the data line is serial data (sd). The bit clock pulses once for each discrete bit of data on the data line and can be calculated by multiplying the sample rate, number of bits per channel, and the number of channels. The ws signal informs the chip whether the left or right channel is being currently sent. I2S allows a device to have two channels send data over the same data line. In this chip when ws is 0 data is being send over the left channel and when ws is 1 data is being sent over the right channel. The ws signal is a 50% duty signal that has the same frequency as the sample frequency.

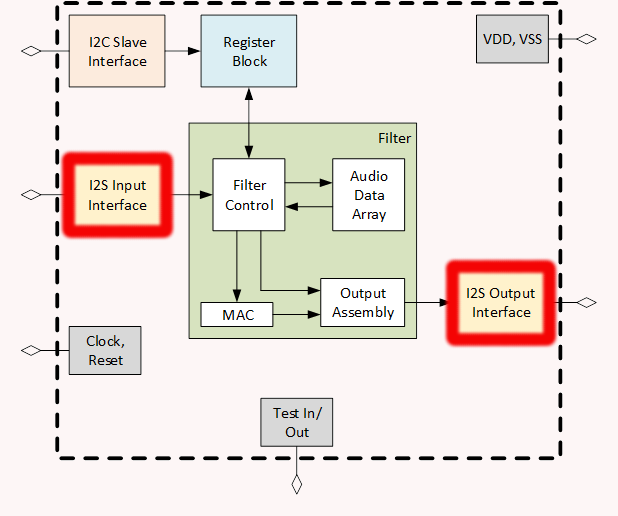
The sd signal is in two’s complement and is transmitted with the most significant bit first (MSB). The reason the serial data is MSB is because the transmitter and receiver may have different word lengths. The transmitter and receiver doesn’t need to know how many bits the other can handle or transmit. The least significant data bits are set to 0 when the system word length is greater than the transmitter word length. However, if the sent bits are larger than the word length of the receiver then the bits after the least significant bit are ignored. If the receiver receives less bits than its word length, then the missing bits are set to 0. Based off of these characteristics the MSB is always located in the same position, however, the LSB position depends on the word length. The MSB is also always sent one clock period after the WS signal changes. The serial data being transmitted can either be synchronized with either the trailing or leading edge of the clock signal. In this implementation the serial data is synchronized with the leading edge. However, the serial data has to be latched onto the receiver on the leading edge of serial clock signal.

The WS signal can change either on the leading falling or rising edge of the serial clock. In the slave, the signal is latched on the rising edge of the clock signal. The WS signal also changes one clock period before the MSB is transmitted from the other channel. This early transition allows the slave transmitter to derive synchronous timing of the serial data. It also enables the receiver to store the previous word and clear the input for the next word.

The device that provides the necessary clock signal is the master. The slave gets its internal clock signal from an external clock input. The total delay between the master clock and the data and word select signal is the sum of the delay between the external (master) clock and the slave’s internal clock, and the delay between the internal clock and the data and/or word select signals. The timing requirements specified are relative to the clock period of the minimum allowed clock period of a device. This allows higher data rate transfers to be used in the future.

*2.3 I2S Interfaces*

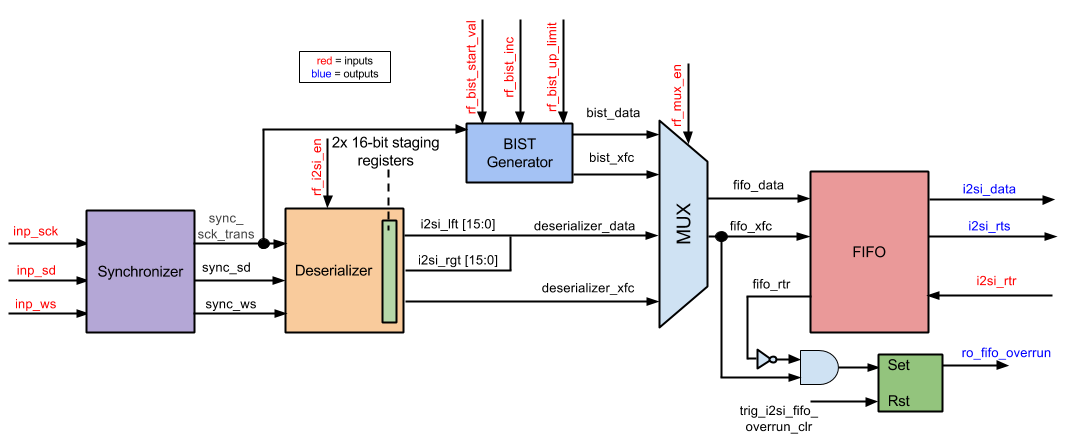
The I2s interface for the custom chip was split into two modules within Verilog, I2S\_in and I2S\_out. The I2S\_in module represented the receiver and handled all data being inputted into the I2S interface, and the I2S\_out module represented the transmitted and handled all data being outputted. The following figure displays the overall block diagram of the chip, with the I2S interfaces highlighted.



**Fig 2.1:** Overall Chip Diagram with I2S interfaces Highlighted

*2.4 I2S Input Interface*

The I2S input interface handles all input audio and relays it to the filter for processing. It is primarily responsible for reading in serialized data and converting it back into parallel data. The I2S Input interface is made of five different sub-modules, the synchronizer, deserializer, bist generator, mux, and FIFO. The following figure displays the block diagram of the I2S input interface with all of its signals and sub-modules.



**Fig 2.2:** I2S Input Block Diagram

The following table displays all the input and output signals the I2S input interface interacts with. The table provides the name, direction, bit size, and a short description of each signal.

**Table 2.1:** List of inputs and outputs of I2S Input Interface

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| inp\_sck | in | 1 | Digital Audio Bit Clock |
| inp\_sd | in | 1 | Digital Audio Serial Data |
| inp\_ws | in | 1 | Word Select |
| i2si\_rtr | in | 1 | Ready to Receive |
| i2si\_rts | out | 1 | Ready to Send |
| i2si\_data | out | 32 | Output Digital Audio |
| rf\_i2si\_en | in | 1 | Serializer Enabled Bit |
| rf\_mux\_en | in | 1 | Built in Self-Test (BIST) |
| rf\_bist\_start\_val | in | 12 | Start Value |
| rf\_bist\_inc | in | 8 | Increment |
| rf\_bist\_up\_limit | in | 12 | Upper Limit |
| trig\_i2si\_fifo\_overrun\_clr | in | 1 | Reset FIFO Overrun |
| ro\_fifo\_overrun | out | 1 | Input Audio FIFO Overrun |

The signal inp\_sck is the digital audio bit clock of the I2S interface, inp\_sd is the serial data that is being sent in, and inp\_ws is the word select signal that informs the module whether the left or right audio channel is being received. The signals i2si\_rtr and i2si\_rts are the handshake interface signals interacting with the FIFO sub-module. They inform when the filter block is ready to receive data and when the I2S input block is read to send data. The i2si\_data signal contains all of the audio data that is to be sent to the FIFO block. The signal rf\_i2si\_en is the signal that enables the serializer sub-block, rf\_mux\_en is the select bit for the multiplexer sub-block. The signals rf\_bist\_start\_val, rf\_bist\_inc, and rf\_bist\_up\_limit are used within the BIST generator and will be further explained in section 2.4.3. Signal ro\_fifo\_overrun indicates whether the data being transferred to the FIFO is exceeding the buffer’s boundary and is overwriting the adjacent memory locations. The signal trig\_i2si\_fifo\_overrun\_clr informs the module to whether or not to clear the ro\_fifo\_overrun signal.

*2.4.1 Synchronizer*

The synchronizer sub-block takes the inputs of serial clock, word select, and serial data and delays the signals and sync them with the master clock. This is done in order to ensure that no metastability occurs and to ensure the timing diagrams of the module is correct. The sub-block was also created to consolidate and shorten the amount of code needed to be written. Multiple blocks make use of the delayed and synced signals. It was appropriate to define these signals once and then to output them to the other blocks that need them. The inputs and outputs of the synchronizer sub-block can be seen in the following table.

**Table 2.2:** List of inputs and outputs of synchronizer sub-block

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| \_sck | in | 1 | Digital Audio Bit Clock |
| \_ws | in | 1 | Word Select |
| \_sd | in | 1 | Digital Audio Serial Data |
| sck | out | 1 | Delayed and Synced Serial Clock |
| sck\_transition | out | 1 | Serial Clock Level to Pulse Converter |
| ws | out | 1 | Delayed and Synced Word Select |
| sd | out | 1 | Delayed and Synced Serial Data |

The \_sck signal is the serial clock input signal and sck is the same signal synced and delayed by 2 master cycles. The sck\_transition signal is a level to pulse converter of the serial clock. The signal is high anytime sck transitions from 0 to 1. The \_ws signal is the word select input signal and ws is the same signal synced and delayed by 4 master clock cycles. The \_sd signal is the serial data input signal and sd is the same signal synced and delayed by 4 master clock cycles.

*2.4.2 Deserializer*

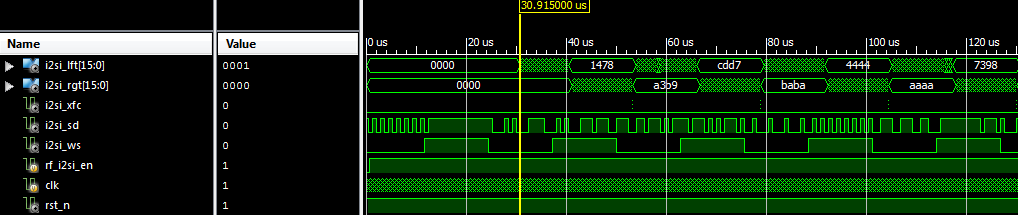
The deserializer sub-block is responsible for taking the audio serial data and converting it to parallel data. The list of inputs and outputs of the deserializer sub-block can be seen in the following table.

**Table 2.3:** List of inputs and outputs of deserializer sub-block

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| sck\_transition | in | 1 | Serial Clock Level to Pulse Converter |
| in\_sd | in | 1 | Digital Audio Serial Data |
| in\_ws | in | 1 | Word Select (Left/Right Audio Channel) |
| rf\_i2si\_en | in | 1 | i2s input is enabled |
| out\_lft | out | 16 | Left Audio Channel |
| out\_rgt | out | 16 | Right Audio Channel |
| out\_xfc | out | 1 | Read Data Transfer Complete |

The inputs sck\_transition, in\_sd, and in\_ws are the outputs coming from the synchronizer sub-block. These are the level to pulse converter of the serial clock and the delayed and synced signals of the word select and serial data signals. As previously stated the rf\_i2si\_en signal is an enable bit for the deserializer sub-block. The signals out\_lft and out\_rgt are the outputs of the left and right audio parallel data respectively. The signal out\_xfc informs the FIFO that the transfer of a word has been completed. Below is a sample simulation of the deserializer.

The deserializer was verified by running a simulation that checked if the left and right output channels properly read in the input of the serial data signal. First it was verified that the serializer began properly outputting data after it becomes active. After properly becoming active the inputted data was verified to match up with the output data. It was also ensured that the data was properly stored into the proper channels depending on the value of the word select signal. Next the least significant bit was checked to see if it was stored after the first clock cycle after the word select signal transitioned. Lastly, the xfc signal was verified to pulse after the last bit of the right channel was stored.



**Fig. 2.3:** Sample of deserializer simulation

*2.4.3 BIST Generator*

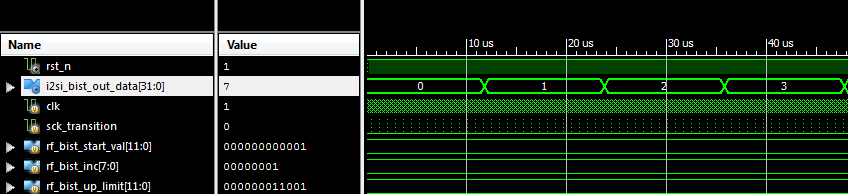
The BIST generator sub-block is used to create a saw-tooth wave to test the I2S input interface. The saw-tooth wave is then outputted to the multiplexer. The following table lists the inputs and outputs of the BIST generator sub-block.

**Table 2.4:** List of inputs and outputs of BIST generator sub-block

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| sck\_transition | in | 1 | Serial Clock Level to Pulse Converter |
| rf\_bist\_start\_val | in | 12 | Start Value |
| rf\_bist\_inc | in | 8 | Increment |
| rf\_bist\_up\_limit | in | 12 | Upper Limit |
| i2si\_bist\_out\_xfc | out | 1 | Transfer Complete |
| i2si\_bist\_out\_data | out | 32 | Output Data |

Signals rf\_bist\_start\_val, rf\_bist\_inc, and rf\_bist\_up\_limit are used to create a saw-tooth wave. The signals inform what value to start at, how much to increment by, and what the upper limit of the saw-tooth wave is respectively. The signals i2si\_bist\_out\_data is the output data describing the saw-tooth wave, and i2si\_bist\_out\_xfc is the output that represents that the saw-tooth wave data transfer was complete.

The BIST generator was verified by running a simple test that set the output value of the data to 0. The data value was then incremented on every 16th pulse of the serial clock. The xfc output signal was verified by checking if it was high after the data incremented 16 times. The sub-block was further tested by ensuring that the output resetted to 0 after the output data reaches the upper limit of the BIST generator.



**Fig. 2.4:** Sample of BIST generator simulation

*2.4.4 Multiplexer*

The multiplexer sub-block is responsible for outputting either the BIST data and xfc output signals or the deserializer data and xfc output signals. The following table lists the multiplexer’s input and output signals.

**Table 2.5:** List of inputs and outputs of multiplexer sub-block

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| in\_0\_data | in | 32 | Input 0 Data |
| in\_0\_xfc | in | 1 | Input 0 Transfer Complete |
| in\_1\_data | in | 32 | Input 1 Data |
| in\_1\_xfc | in | 1 | Input 1 Transfer Complete |
| sel | in | 1 | Select Bit |
| mux\_data | out | 32 | Data Output |
| mux\_xfc | out | 1 | Transfer Complete Output |

The signals in\_0\_data, in\_0\_xfc, in\_1\_data, and in\_1\_xfc are the outputs of either the BIST generator or deserializer data and xfc signals. The sel signal is the select bit that either outputs the BIST data and xfc or deserializer data and xfc signals. The signals mux\_data and mux\_xfc represent the data and xfc signals being outputted to the FIFO sub-block.

*2.4.5 FIFO*

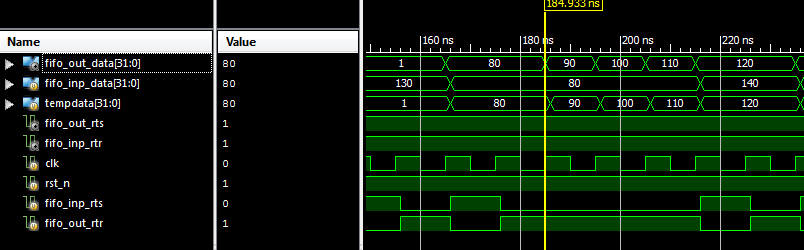
The FIFO sub-block organizes and manipulates a data buffer, where the first entry is processed first. In this particular block is is responsible for manipulating the audio data inputted and outputted in the I2S interfaces. The following table lists the inputs and outputs of the sub-block.

**Table 2.6:** List of inputs and outputs of FIFO sub-block

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| fifo\_inp\_data | in | 32 | Input Data |
| fifo\_inp\_rts | in | 1 | Write Client Asserts Ready to Send |
| fifo\_inp\_rtr | out | 1 | Output FIFO Asserts Read to Receive |
| fifo\_out\_data | out | 32 | Output Data |
| fifo\_out\_rts | out | 1 | Output FIFO Asserts Ready to Send |
| fifo\_out\_rtr | in | 1 | Read Client Asserts Read to Receive |

The fifo\_inp\_data is the parallel audio data being transferred into the FIFO block. The fifo\_inp\_rts and fifo\_inp\_rtr signals are the handshaked interface signals that inform whether data is ready to be sent and read respectively between the multiplexer and filter if in the I2S input interface or between the filter and FIFO if in the I2S output interface. The fifo\_out\_data signal is the parallel audio output data being sent to either the filter block if in the I2S input interface or serializer if in the I2S output interface. The fifo\_out\_rts and fifo\_out\_rtr signals are the handshaked interface signals that whether data is ready to be sent and read respectively between the filter and FIFO if in the I2S input interface or the FIFO and filter if in the I2S output interface.

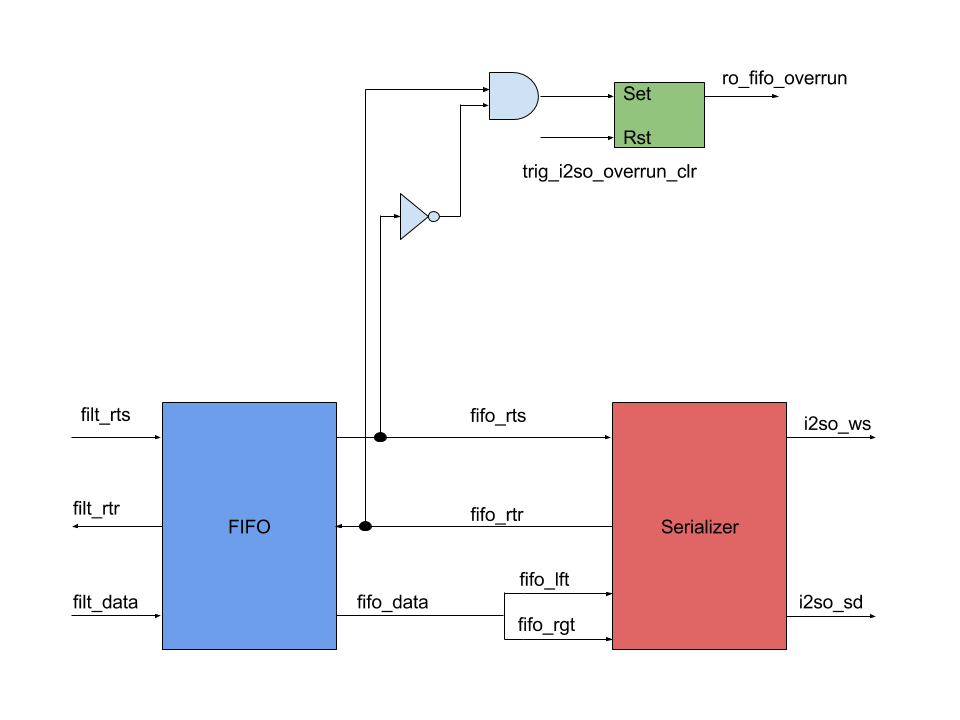
The FIFO sub-block was verified by popping and pushing random values within a test bench. A quick simulation was run to check if the FIFO pushed and popped the values defined within the test bench. A sample simulation of the FIFO block can be seen below.



**Fig. 2.5:** Sample of FIFO simulation

*2.5 I2S Output Interface*

The I2S output interface is primarily responsible for converting the received parallel audio data from the filter block to serial data as an output. The I2S output interface is made of two sub-blocks, the serializer and FIFO. The following diagram shows the sub-blocks that make the I2S\_out block and displays input, output, and interconnecting signals.



**Fig 2.6:** Block Diagram of I2S Output Interface

The following table displays all the input and output signals the I2S output interface interacts with. The table provides the name, direction, bit size, and a short description of each signal.

**Table 2.7:** List of inputs and outputs of I2S Output Interface

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| sck\_transition | in | 1 | Serial Clock Pulse Converter |
| filt\_data | in | 32 | Parallel Digital Audio |
| filt\_rtr | out | 1 | Ready to Receive |
| filt\_rts | in | 1 | Ready to Send |
| i2so\_sck | out | 1 | Digital Audio Bit Clock |
| i2so\_ws | out | 1 | Word Select (Left/Right Audio Channel) |
| i2so\_sd | out | 1 | Digital Audio Serial Data |
| trig\_i2so\_fifo\_overrun\_clr | in | 1 | Reset FIFO Overrun |
| trig\_i2so\_fifo\_underrun\_clr | in | 1 | Reset FIFO Underrun |
| ro\_fifo\_underrun | out | 1 | Output Audio FIFO Underrun |
| ro\_fifo\_overrun | out | 1 | Output Audio FIFO Overrun |

The sck\_transition signal is the level to pulse converter signal that will be provided by the synchronizer block in the I2S input interface. The signal filt\_data will be provided by the filter block that contains the parallel audio data. The signals filt\_rtr and filt\_rts come from the filter block and are the handshaked interface signals that inform whether the filter is ready to send and the fifo and is ready to receive data. The i2so\_sck is the serial clock that is outputted in synchronizer in the I2S Input interface that will be transmitted through the I2S output interface. The i2so\_ws is an output signal that indicates whether the data being transmitted is either the left or right channel. The i2so\_sd signal is the serial data that was converted from the parallel audio input data. The ro\_fifo\_overrun signal indicates whether the data being written to the FIFO is greater than the buffer’s boundary, and when trig\_i2so\_fifo\_ovverun\_clr signal is 1 it resets the ro\_fifo\_overrun signal as 0. The ro\_fifo\_underrun signal indicates whether the data being fed to the buffer is slower than is being read from it, and when the trig\_i2so\_fifo\_underrun\_clr signal is 1 the ro\_fifo\_underrun signal is 0.

*2.5.1 Serializer*

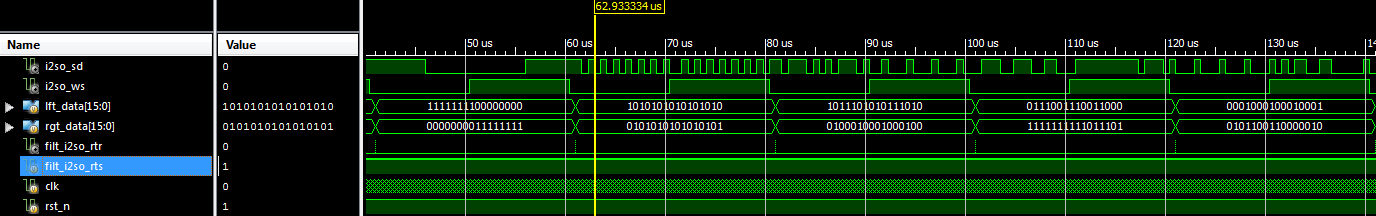
The serializer sub-block is where the parallel audio data is converted into serial data. The following table shows the inputs and outputs of the serializer sub-block.

**Table 2.8:** List of inputs and outputs of serializer sub-block

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rst\_n | in | 1 | Reset |
| sck\_transition | in | 1 | Serial Clock Level to Pulse Converter |
| filt\_i2so\_lft | in | 16 | Left Audio Channel |
| filt\_i2so\_rgt | in | 16 | Right Audio Channel |
| filt\_i2so\_rts | in | 1 | Ready to send |
| filt\_i2so\_rtr | out | 1 | Ready to read |
| i2so\_en | in | 1 | i2s output is enabled |
| i2so\_sd | out | 1 | Digital Audio Serial Data |
| i2so\_ws | out | 1 | Word Select |

As previously stated the sck\_transitoin signal is the output created from the synchronizer sub-block. The signals filt\_i2so\_lft and filt\_i2so\_rgt are the parallel audio data channels that were outputted by the FIFO. As previously mentioned the filt\_i2so\_rts and filt\_i2so\_rtr are the handshaked interface signals that indicate whether the FIFO is ready to send and the serializer is ready to read data. However, it is important to note in this case that the serializer only cares when the rtr signal becomes high on its first instance. The serializer will continue to be active regardless if the rtr signal becomes 0 again. The i2so\_en signal is the enabled bit that enables the serializer. The i2so\_sd and i2so\_ws as mentioned before are the output of the I2S output interface of the serial data and word select signals.

The serializer sub-block was verified by running a test bench that fed in data to the left and right channels. To ensure that the block was working properly it needed to become active after the first rtr signal that goes from low to high. After becoming active the serial data output should correspond to the input data coming in from the left and right channel depending on the value of word select.



**Fig. 2.7:** Sample of serializer simulation

**Chapter 3: Digital Filtering - Dhruvit Naik**

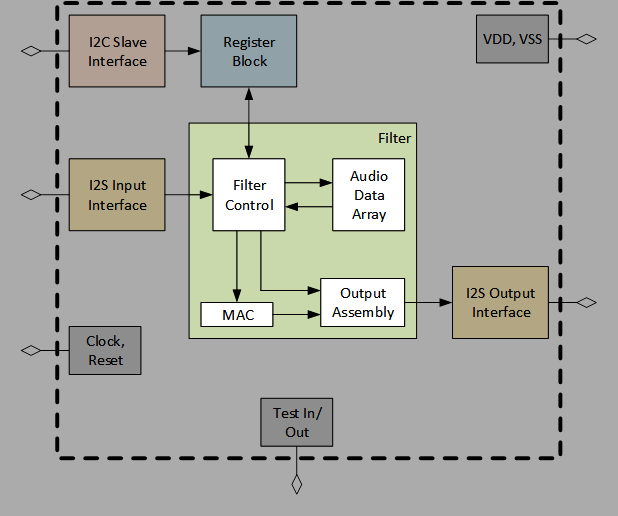
*3.1 Introduction*

The main function of the ASIC chip is as a digital filter for audio processing. A digital filter operates on an input signal and produces a new signal, or filtered signal. In DSP, filters have two uses: signal separation and signal restoration. Separation is done when an input signal is polluted with interference, noise, or other signals. Filtering, removes any unwanted features from an input signal. Restorations is used when the signal has been distorted. One example of restorations is when an image is focused after being acquired due to a shaky lens. These filters are implemented in all types of devices and range in complexity. For example, MP3 players have a multitude of digital filters structures for different applications within the system. In order to form an equalizer (EQ) in a MP3 Player, digital filter structures are cascaded together. Digital filters can be implemented two ways: by convolution and by recursion. Convolution is also called finite impulse response or FIR and recursion is called infinite impulse response or IIR.The filter coefficients are derived from the impulse response of the system. For a FIR filter, the impulse response are the filter coefficients. The given filter coefficients characterize the type of the digital filter. Convolution is defined .Where x is the discrete time input signal and h is the filter coefficients/impulse response. For the TCNJ ASIC, the filter will have 512 taps or 512 delays. Thus the filter will need to store, present and past inputs and index them appropriately.

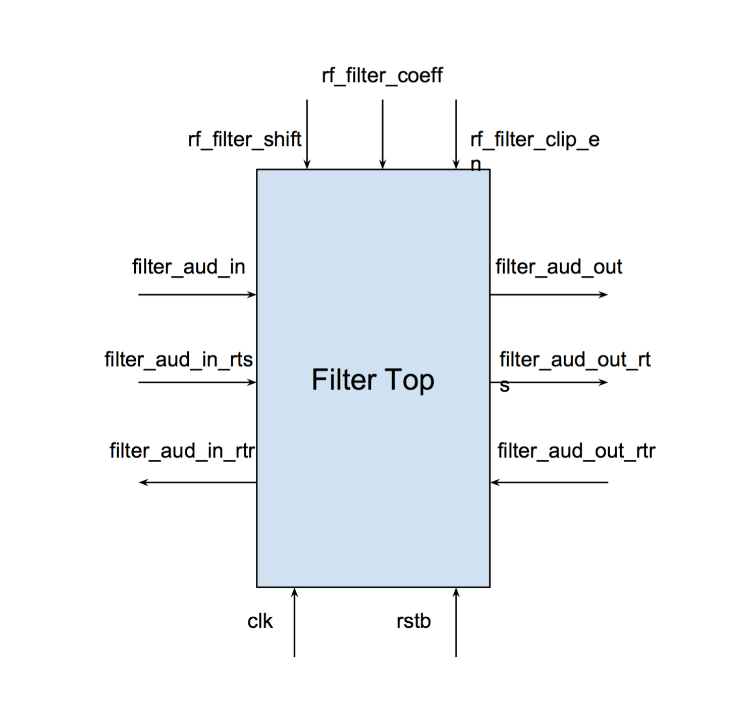
In order to design a digital filter for audio processing, the human ear needs to be taken into consideration. The perception of continuous sound, is separated into three parts: loudness, pitch, and timbre. The audio quality will be equal to the sound quality from a high fidelity compact disc. The specifications for the sound quality are the following. The sampling rate is 44.1 kHz and the data bits are 16 bits per audio channel (left and right).

*3.2 Filter*

The filter block is a sub-block of the top level chip. Figure 3.1 shows the filter with respect to the other interfaces.

**Fig 3.1:** Chip Diagram with Filter Highlighted 

The filter interfaces with the I2S input and output and the Register block. The incoming audio signal is streamed in from the I2S input as serial data. Both channels (right and left) are sent as the same data line. In addition to providing control bits for the shift and clip, the register is responsible for providing the filter with filter coefficients. The input signal is convolved with the filter coefficients. Before the signal is sent to the I2S output, three operates are performed: rounded, shifted, and clipped. Figure 3.2 is the top level view of the filter. Table 3.1 describes the corresponding signals for the top level filter.



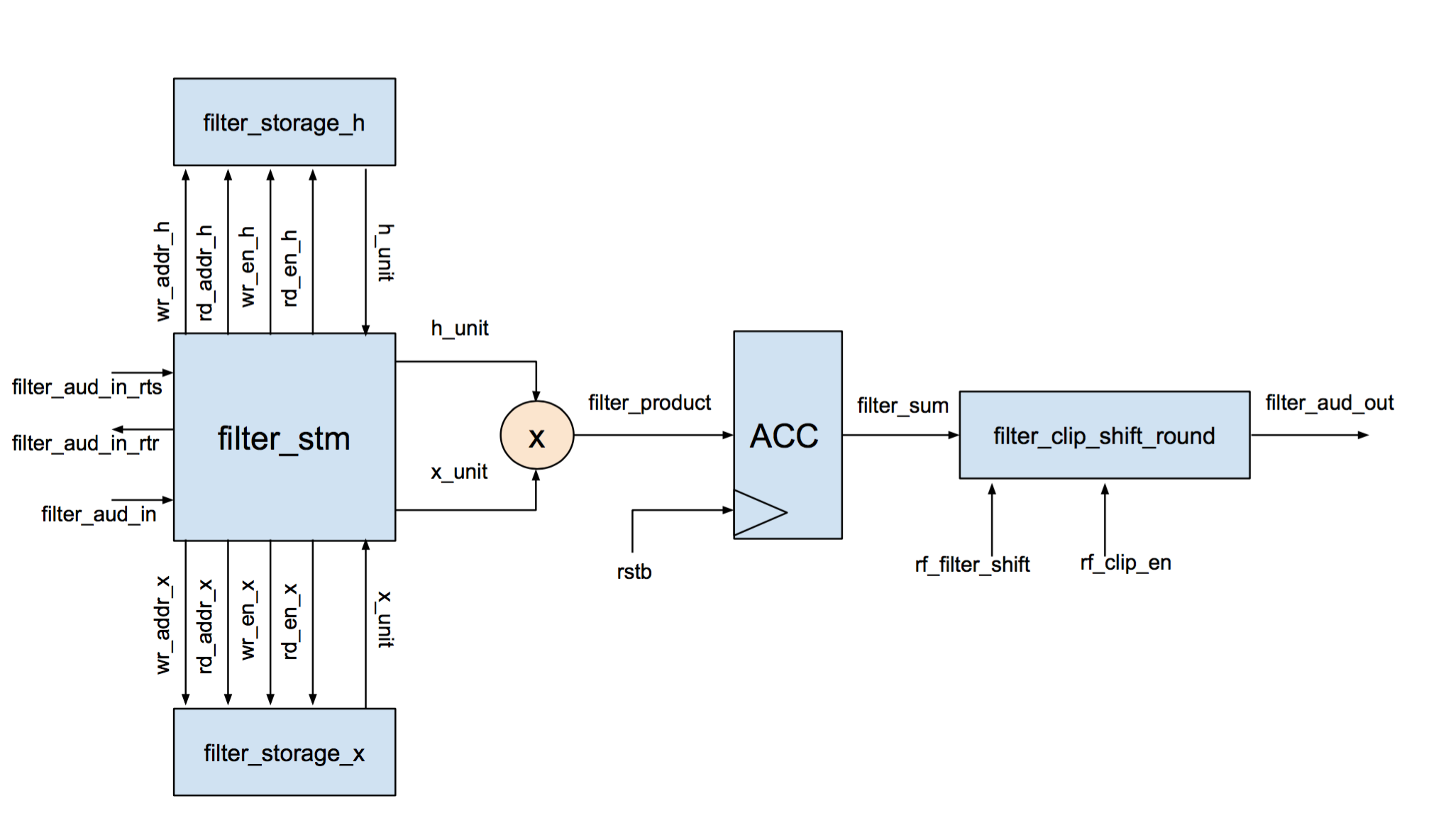
**Fig 3.2:** Top Level Block Diagram of Filter

**Table 3.1:** Input and Output Signals for Filter Top

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rstb | in | 1 | Asynchronous Reset |
| filter\_aud\_in | in | 32 | Input Digital Audio |
| filter\_aud\_in\_rts | in | 1 | Input FIFO Asserts Ready to Send |
| filter\_aud\_in\_rtr | out | 1 | Filter STM Asserts Ready to Receive |
| filter\_aud\_out | out | 32 | Output Digital Audio |
| filter\_aud\_out\_rts | out | 1 | Filter STM Asserts Ready to Send |
| filter\_aud\_out\_rtr | in | 1 | Output FIFO Asserts Read to Receive |
| rf\_filter\_shift | in | 4 | number of bit positions to shift after filter acc. |
| rf\_filter\_clip\_en | in | 1 | 1-perform clipping 0-no clipping |
| rf\_filter\_coeff [0:511] | in | 8196 | filter coefficient |

There are some considerations with some of the signals. First, *rstb*, represents an asynchronous reset. An asynchronous reset, triggers on the negedge as opposed to a synchronous reset, which triggers on the posedge. In simpler terms, when *rstb* is asserted the filter will perform normally. The advantage of using an asynchronous reset is, the reset has priority over any signal, including the master clock. This allows the block to reset with or without the presence of a clock. The second signals to consider are *filter\_aud\_in* or *filter\_aud\_out*. These digital audio signals contain bits for both audio channels. The left audio channel bits are stored in the first 16 LSB [15:0] and the right audio channel bits are stored in the first 16 MSB [31:16]. The final signal to consider further is *rf\_filter\_coeff*. The filter coefficients are assumed to be stable. If they change during any calculation of an output sample the result will be unpredictable.

Figure 3.3 is a more detailed block diagram of the filter.

**Fig 3.3:** Detailed Block Diagram of Filter 

The following subsections, will highlight specific submodules of the filter and describe them in detail.

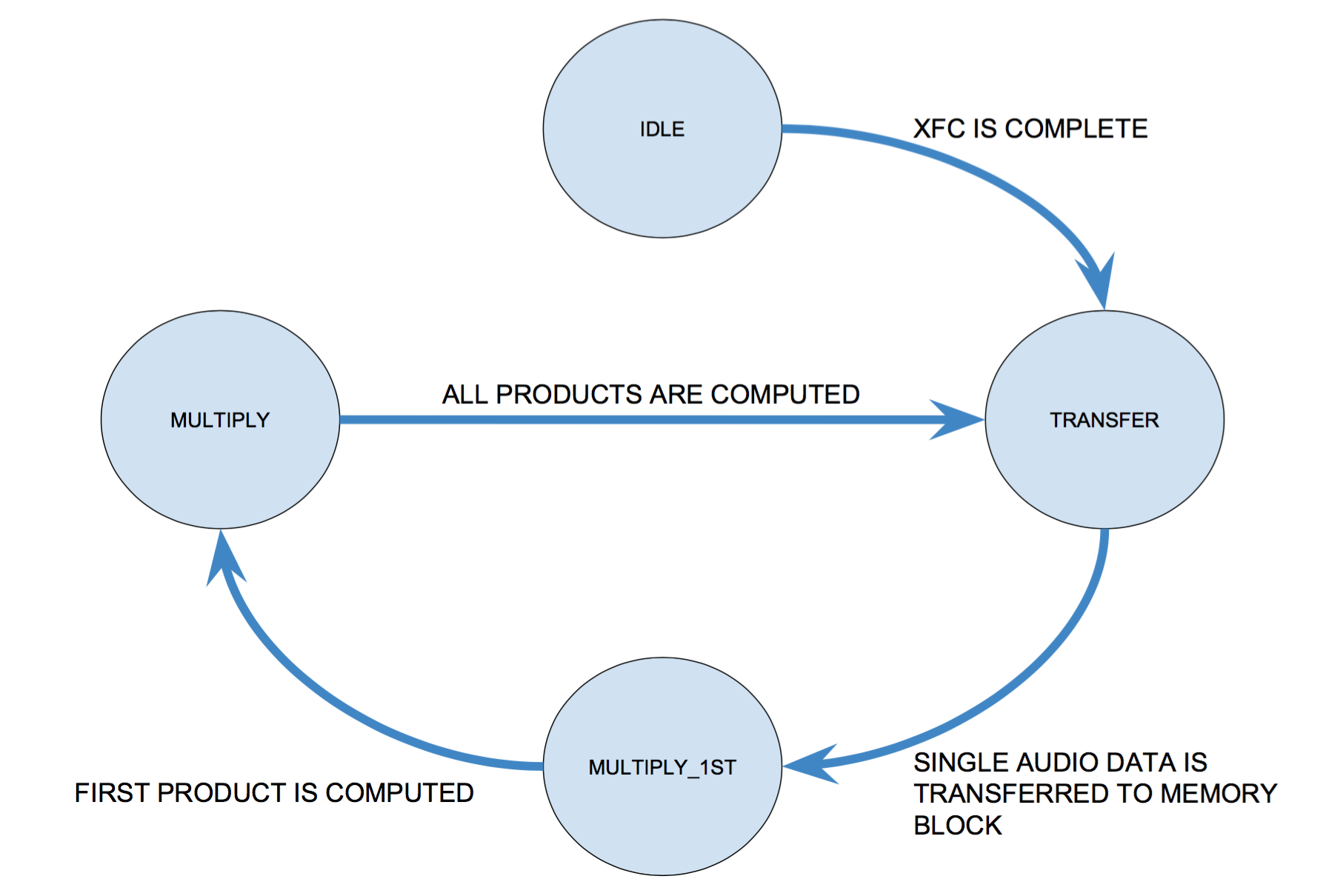
*3.2.1 Filter State Machine*

The filter is controlled by a centralized finite state machine (FSM). A FSM can only function in a single state, or the current state. It is defined by its different states and trigger conditions (transitions). In verilog, the FSM can be written as a One-Hot Encoded FSM. In a One-Hot each state is represented by a single flip-flop. For the TCNJ ASIC, the filter has 4 different states, so for this instance 4 flip-flops will be reserved to represent the filter FSM. For the TCNJ ASIC applications, the number of available flip-flop is abundant for the FPGA implementation and for the ASIC, so a One-Hot will not affect performance. Table 3.2 shows the highlights and describes the signals for the FSM.

**Table 3.2:** Input and Output Signals for Filter FSM

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| rstb | in | 1 | Asynchronous Reset |
| filter\_aud\_in | in | 32 | Input Digital Audio |
| rf\_filter\_coeff | in | 16 | Filter Coefficient |
| filter\_aud\_out | out | 32 | Output Digital Audio |
| do\_transfer | out | 1 | Indicate Filter is In Transfer State |
| do\_multiply\_1st | out | 1 | Indicate Filter is In Multiply 1st State |
| do\_multiply | out | 1 | Indicate Filter is In Multiply State |
| filter\_aud\_in\_rts | in | 1 | Input FIFO Asserts Ready to Send |
| filter\_aud\_in\_rtr | out | 1 | Filter STM Asserts Ready to Receive |

In addition to the signals described above, the submodule uses additional internal wires and signals. These specific signals can be found in the Source Code section in the Appendix. Below, in Figure 3.4 is a bubble diagram for the FSM. Each state is discussed further in detail.



**Fig 3.4:** FSM: States and Transitions

**IDLE:**

The first state of the FSM is the IDLE state. The IDLE state functions as a buffer for the block. It will kick off the filter as the first XFC is complete. The IDLE state will not be a current state again unless the chip is reset. At this state, the filter will check if XFC is asserted. If XFC is not asserted, RTR will be asserted and the filter will wait until XFC is asserted.

**TRANSFER:**

Once XFC is asserted, the filter will transition into the TRANSFER state. In this state, *filter\_aud\_in* is written into the storage module and the write pointer is incremented. Once the transfer is complete, read enable for the storage module and *filter\_aud\_in\_rtr* are deasserted. The FSM will transition to the convolution phase after this is complete.

**MULTIPLY\_1ST:**

The convolution phase is separated into two states because the accumulator needs to be reset for the new set of products. After the first computations is completed the state will transition.

**MULTIPLY:**

In the MULTIPLY\_1ST AND MULTIPLY states the FSM access the storage module for computation. The TCNJ ASIC filter is a 512 tap filter. The 512, represents the order of the transfer function that describes the filter. In simpler terms, the filter needs to access previous inputs, as far as 512 inputs from the past. Further discussion of the storage module is found in the next subsection. The 512 inputs are convolved with a corresponding filter coefficient. The filter coefficients are accessed from 0 to 511 for each iteration of the states. The inputs are accessed from the newest input to the input from the last iteration. Below are a few iterations of the convolution process.

1. y[0] = h[0] x[0] + h[1] x[1] + … + h[510] x[510] + h[511] x[511]
2. y[0] = h[0] x[1] + h[1] x[2] + … + h[510] x[511] + h[511] x[0]
3. y[0] = h[0] x[2] + h[1] x[3] + … + h[510] x[0] + h[511] x[1]

Each of these products are computed each clock cycle. Each of these products are sent to an accumulator, which stores the sum of products until the next iteration. The accumulator will store a 40-bit signal, so there is little chance of an overflow. This 40-bit signal is rounded, clipped, and shifted and is ready to be sent to the Output I2S. Once the state is ready for a new input, read enable for both storage modulus is deasserted and *filter\_aud\_in\_rtr* is asserted. The filter will go through each of the discussed states until the chip is reset and the filter will transition into the IDLE state.

*3.2.2 Filter Storage Module*

The storage module is responsible for storing, indexing, writing, and reading the input signal and the filter coefficients. Table 3.3 shows the highlights and describes the signals for the Filter Storage Module. These signals are representations for both instances of the submodule.

**Table 3.3:** Input and Output Signals for Filter Storage Module

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| clk | in | 1 | Master Clock |
| wren | in | 1 | 1-wrdata will be written into wrptr |
| wrptr | in | 8 | Current Location of Write Pointer |
| wrdata | out | 32 | Data to be Written |
| rden | out | 1 | 1-rddata will be read from rdptr |
| rdptr | out | 8 | Current Location of Read Pointer |
| rddata | out | 32 | Data Being Read |

In addition to the signals stated above, the storage module defines a 2D array. The width of the array is 32 bits and the depth is 512 locations. The array behaves in a circular manner. Once the *wrptr* or *rdptr* indexes the last position it will wrap back to the beginning of the array. Consequently, the data will be overwritten. The signals *wren*, *rden*, *wrptr*, and *rdptr* are controlled by the state machine.

*3.2.3 Accumulator/ Barrel Shifter*

An accumulator, at the posedge clock, takes the input signal and adds it to the current value that is stored in a D flip-flop. The resulting value is stored for future computations until the accumulator is reset. To avoid overflow, the value being stored will be 40 bits long.

A barrel shifter is capable of shifting n number of bits in a single clock cycle. The type of shift is a right shift. The signals­ are defined in the below table.

**Table 3.4:** Input and Output Signals for Barrel Shifter Module

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Bits** | **Comment** |
| input\_signal | in | 32 | Signal being performed on |
| sel\_shift | in | 5 | How many bits being shifted |
| output\_signal | out | 32 | Shifted Signal |

The barrel shifter defines all the different outputs for a given input signal. A case statement determines which output signal will be selected based on the signal *sel\_shift*.

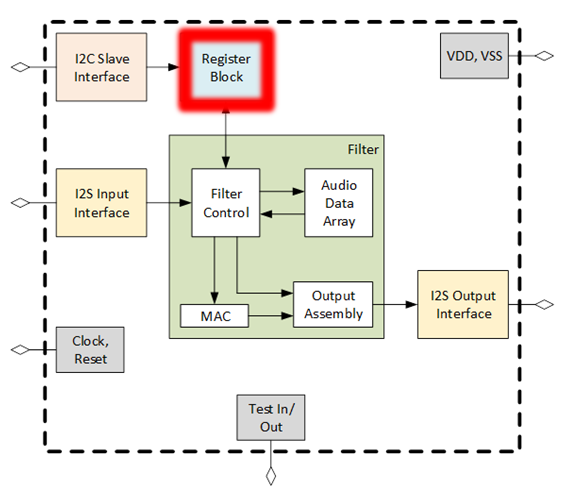
*3.3 Verification*

TODO

**Chapter 4: Register Block - Julie Swift**

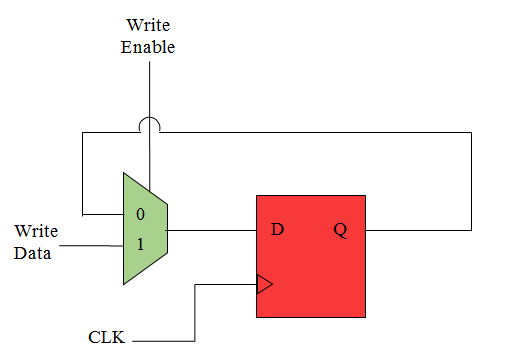
*4.1 Introduction*

The register block provides storage and access for control and status information. In Fig 4.1 the interaction of register block and the other blocks can be observed. The I2C slave interface sends byte-wide control data to the register block. Many of the registers represent filter coefficients and act as the filter’s parameters. These parameters can be changed by the user and written to the register block to control the audio filter behavior. Each filter coefficient is represented by two 8 bit values that are concatenated when sent to the filter control block. The I2C must be able to specify an address and write data to the specified address for the register to hold. When reading data, the I2C block will be able to read the stored data of a specified address. When an underrun occurs in the I2S’s FIFO, the register block will be able to clear the sticky bit that flags the problem where the filter is producing data at a lower rate than the I2S Output is consuming it



**Fig 4.1:** Chip Diagram with Register highlighted

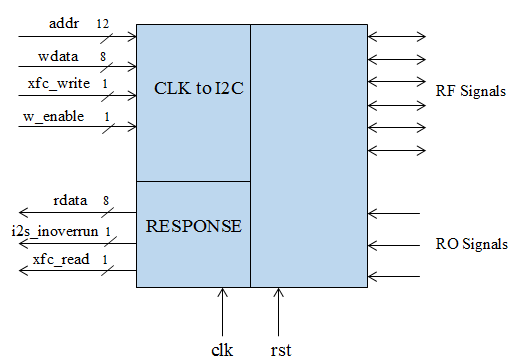
Internally, the register block is broken down into a data demultiplexer that takes the write enable, address, and write data, and writes to the appropriate register bits based on the address. With the rising edge of the write enable, the data demultiplexer allows data to be stored in the register. This low level logic is controlled by the rising edge of the clock, which can be seen in Fig 4.2.



**Fig 4.2:** Internal Micro-Architecture

*4.2 Interfaces*

The top level interface of the register block is described in Fig 4.3 which shows the signal’s flow of direction and number of bits in each signal. The addr input signal is an 12-bit variable that holds the value of the address in the register block in which data can be read from or written to.



**Fig 4.3:** Top Level Interface Design of reg.v

The wdata is an input variable array that writes 8 bits of data to a specified 12-bit address. Similarly, the rdata output array returns the 8 bits of data requested by any signal request of a specified address. The write\_xfc is an input 1-bit transfer signal that indicates when the transfer of the data written is complete. The read\_xfc is an output 1-bit signal that similarly goes high when read data transfer is complete. Master clock, clk, and reset, rst, are 1 bit signals that regulate activity to occur on the positive edge of the clock or the negative edge of the rst signal. Lastly, the i2s\_inoverrun will turn into a sticky bit that is manipulated when I2S FIFO has an overrun or underrun condition.

*4.3 Register Mapping*

The register map seen in Table 4.1 includes a list of all the signals assigned to an address and shows the breakdown of bits in each address. The natural 32-bit word aggregation is indicated in bold. Each signal is prefixed with “ro” or “rf” in the beginning of the field name. When a signal is prefixed with “ro” it means the signal represents a “Read Only” field, and is an input signal to the register block.  The “rf” prefix indicates that the signal is a “Read/Write” register bit, and is an output signal. The CONTROL signals are either on, meaning they are set to the value 1, or off when set to a value of 0. The I2S\_CLOCK\_CONTROL are addresses that control the clock for the I2S module. The STATUS addresses are 1 bit signals used when the I2S FIFO exhibits an overrun or underrun in which the signals will then be set to a value of 1 to alert the FIFO malfunction. BIST addresses will be dedicated to I2S’s predefined sawtooth wave. The read/write BIST signals are responsible for the starting value of the sawtooth wave, incrementing the sawtooth wave, and determining the upper limit of the sawtooth wave. The I2C\_REG\_INDIR\_ADDR is the address register used for indirect addressing via I2C. Lastly, the 512 filter coefficients are broken down into two 8 bit coefficients with parts a and b. Each subdivided coefficient is assigned to an address in which data can be written and read from.

**Table 4.1:** Register Mapping of each bit in the Address it is Stored

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Address** | **Register Name** |  | **Field Name** | **Bits** | **Description** | **RO/WO/RW** | **Default Value** |
| **0x000** | **CHIP\_INFO** |  |  |  |  |  |  |
| 0x000 |  | 7:0 | ro\_chip\_id | 7:0 | Fixed Chip ID | RO | 0x1234 |
| 0x001 |  | 7:0 | ro\_revision\_id | 15:8 | Fixed Revision ID | RO | 0 |
| **0x004** | **CONTROL** |  |  |  |  |  |  |
| 0x004 |  | 0:0 | rf\_soft\_reset | 0:0 | 0- normal operation. 1- assert soft reset | RW | 0 |
| 0x004 |  | 1:1 | rf\_i2si\_bist\_en | 1:1 | 0- audio source is i2si. 1- audio source is BIST | RW | 0x1 |
| **Address** | **Register Name** |  | **Field Name** | **Bits** | **Description** | **RO/WO/RW** | **Default Value** |
| 0x004 |  | 5:2 | rf\_filter\_shift | 5:2 | number of bit positions to shift after filter accumulator | RW | 0xF |
| 0x004 |  | 6:6 | rf\_filter\_clip\_en | 6:6 | 1- performs clipping 0- no clipping | RW | 0x1 |
| 0x005 | optional? | 3:0 | rf\_i2si\_dec\_factor | 11:8 | sample and hold audio values | RW | 0 |
| 0x005 | optional? | 7:4 | rf\_i2so\_dec\_factor | 15:12 | sample and hold audio values | RW | 0 |
| **0x008** | **I2S\_CLOCK\_CONTROL** |  |  |  |  |  |  |
| 0x008 |  | 7:0 | rf\_i2so\_clk2sck\_div\_a | 15:0 | half of the clock frequency divided by this # | RW | 0x40 |
| 0x009 |  | 7:0 | rf\_i2so\_clk2sck\_div\_b |  |  | RW | 0x40 |
| **0x00C** | **STATUS** |  |  |  |  |  |  |
| 0x00C |  | 0:0 | trig\_fifo\_overrun | 0:0 | fifo overrun clear | WO | NA |
| 0x00C |  | 1:1 | ro\_fifo\_overrun | 1:1 | input audio fifo overrun | RO | 0 |
| 0x00C |  | 2:2 | trig\_fifo\_underrun | 2:2 | fifo underrun clear | WO | NA |
| 0x00C |  | 3:3 | ro\_fifo\_underrun | 3:3 | output audio fifo underrun | RO | 0 |
| **0x010** | **BIST** |  |  |  |  |  |  |
| 0x010 |  | 7:0 | rf\_i2si\_bist\_incr | 7:0 | increment of sawtooth wave | RW | 0x010 |
| 0x011 |  | 7:0 | rf\_i2si\_bist\_start\_val\_a | 19:8 | start value of sawtooth wave | RW | 0x800 |
| 0x012 |  | 3:0 | rf\_i2si\_bist\_start\_val\_b |  |  |  |  |
| 0x012 |  | 7:4 | rf\_i2si\_bist\_upper\_limit\_a | 31:20 | upper limit of the sawtooth wave | RW | 0x7FF |
| **Address** | **Register Name** |  | **Field Name** | **Bits** | **Description** | **RO/WO/RW** | **Default Value** |
| 0x013 |  | 7:0 | rf\_i2si\_bist\_upper\_limit\_b |  |  |  |  |
| **0x014** | **I2C\_REG\_INDIR\_ADDR** |  |  |  |  |  |  |
| 0x014 |  | 7:0 | rf\_i2c\_reg\_indir\_addr\_a | 11:0 | address register used for indirect addressing via i2c | RW | 0 |
| 0x015 |  | 3:0 | rf\_i2c\_reg\_indir\_addr\_b |  | address register used for indirect addressing via i2c | RW | 0 |
| **0x0400** | **FILT\_COEFFS\_0\_1** |  |  |  |  |  |  |
| 0x0400 |  | 7:0 | rf\_filter\_coeff0\_a | 15:0 | Filter Coefficient 0 | RW | 0x0 |
| 0x0401 |  | 7:0 | rf\_filter\_coeff0\_b |  | Filter Coefficient 0 | RW | 0x0 |
| 0x0402 |  | 7:0 | rf\_filter\_coeff1\_a | 31:16 | Filter Coefficient 1 | RW | 0x0 |
| 0x0403 |  | 7:0 | rf\_filter\_coeff1\_b |  | Filter Coefficient 1 | RW | 0x0 |
| **0x0404** | **FILT\_COEFFS\_2\_3** |  |  |  |  |  |  |
| 0x0404 |  | 7:0 | rf\_filter\_coeff2\_a | 15:0 | Filter Coefficient 2 | RW | 0x0 |
| 0x0405 |  | 7:0 | rf\_filter\_coeff2\_b |  | Filter Coefficient 2 | RW | 0x0 |
| 0x0406 |  | 7:0 | rf\_filter\_coeff3\_a | 31:16 | Filter Coefficient 3 | RW | 0x0 |
| 0x0407 |  | 7:0 | rf\_filter\_coeff3\_b |  | Filter Coefficient 3 | RW | 0x0 |
| **0x7FC** | **FILT\_COEFFS\_510\_511** |  |  |  |  |  |  |
| 0x07FC |  | 7:0 | rf\_filter\_coeff510\_a | 15:0 | Filter Coefficient 510 | RW | 0x0 |
| 0x07FD |  | 7:0 | rf\_filter\_coeff510\_b |  | Filter Coefficient 510 | RW | 0x0 |
| 0x07FE |  | 7:0 | rf\_filter\_coeff511\_a | 31:16 | Filter Coefficient 511 | RW | 0x0 |
| **Address** | **Register Name** |  | **Field Name** | **Bits** | **Description** | **RO/WO/RW** | **Default Value** |
| 0x07FF |  | 7:0 | rf\_filter\_coeff511\_b |  | Filter Coefficient 511 | RW | 0x0 |

*4.4 Sub-Blocks*

*4.4.1 trig\_generator.v*

The trig\_generator.v is responsible for generating a signal to clear the overrun and underrun status bits. The two output bits, trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr, are initialized to zero when not rst. If the address is the 12-bit hexadecimal number 0x00C and the XFC is set to one, then the if-statement returns true and the trigger bits are edited. Trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr are initialized to zero first in order to allocate them later only in the case of an overrun or underrun. When the data transfer is complete, meaning the XFC is 1, and the address is 0x00C, two if-statements are then checked. One stating that when data written to 0x00C is 0, the trig\_i2si\_fifo\_overrun\_clr bit is set to 1, and the other stating that when data written to 0x00C is 2, the trig\_i2so\_fifo\_underrun\_clr bit is set to 1. These 2 bits in the address 0x00C report back to the I2S block where the FIFO is signaled to either have an overrun or underrun.

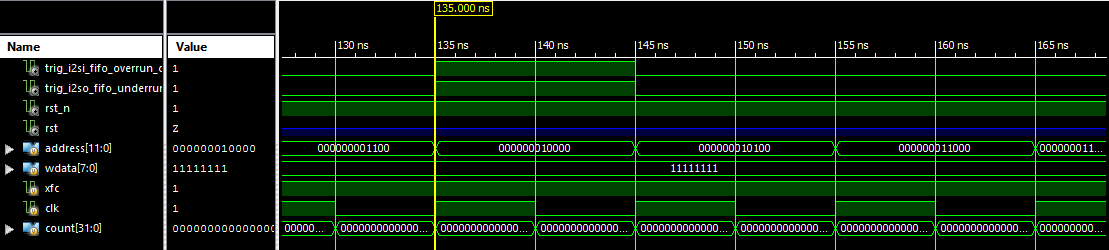
*4.4.2 register.v*

The register.v is responsible for initializing all the signals and filter coefficients along with assigning all signals/coefficients to the proper address. The 16-bit filter coefficients are initialized to zero and the rest of the signals are initialized to the hexadecimal values given in Table 4.1. When the write enable, w\_enable, and the file transfer, i2c\_xfc\_write, are both 1, a case statement is entered; given an address, the corresponding bit(s) within the address are broken down and assigned to the data written to the register. The register.v block will allow addresses to be specified and give blocks the address the data can be read.

*4.5 Test Fixtures*

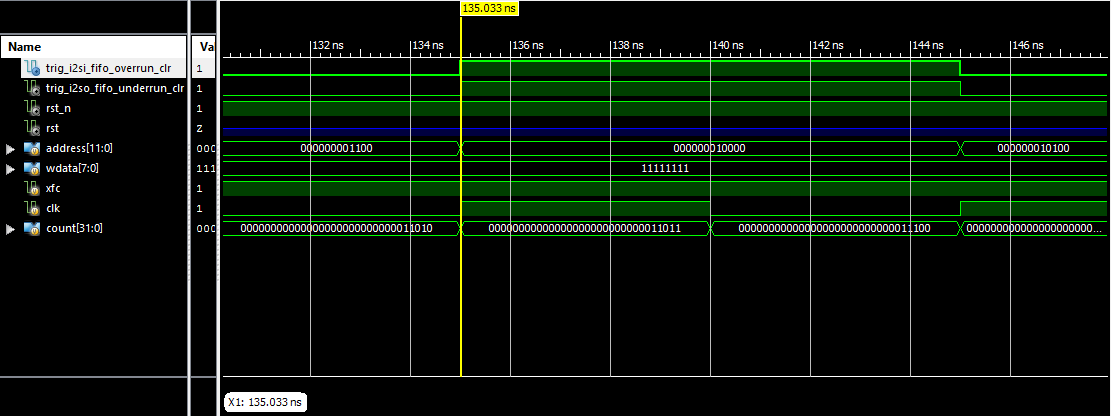
*4.5.1 trig\_generator\_testbench.v*

The purpose of the trig\_generator\_testbench.v is to write data to the address and trigger the trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr bits at the appropriate time. First the test fixture initializes the count and clk variables to zero to create a fresh simulation. The wdata is set to 8 bit address 8’hFF. The count variable counts to the hexadecimal value 20 where the 12-bit address is then initialized by 4 every time the count variable is incremented as seen in Fig 4.4. The rising edge of the XFC signals that the transfer of data has been completed, which allows data to be written to the address space and increment the address signal by 4 from 0000 to 0100. This test fixture writes to every address in order to ensure data is properly being written and stored.



**Fig 4.4:** The start of the Test Fixture Simulation

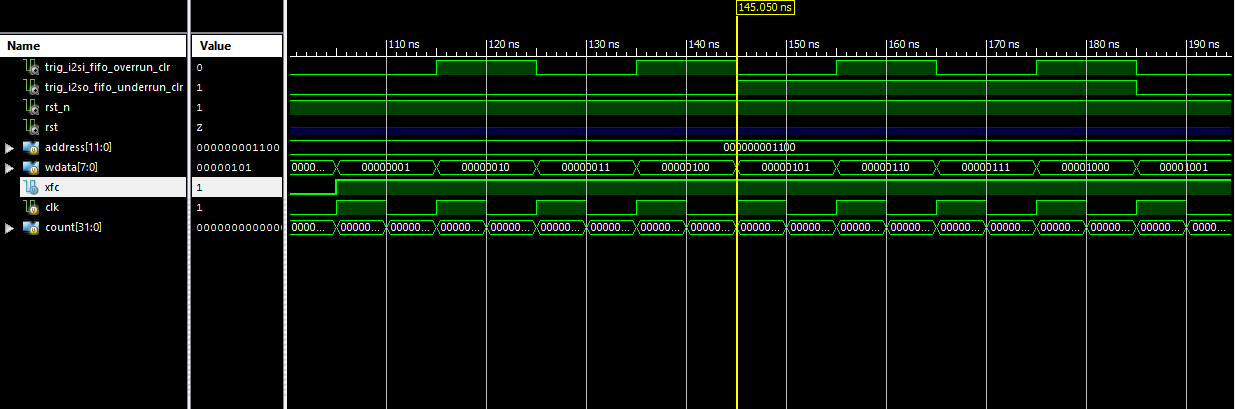
When the count variable reaches 20 and the hexadecimal value 0x00C is reached, the trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr bits are triggered to becomes 1 as seen in Fig 4.5. After the address 0x00C, or binary address 1100, goes low, the trigger bits go high for the length of the next address. This shows that the trigger bits are recognizing an overrun and underrun from the I2S FIFO because the conditions, XFC = 1, and address = 0x00C, are true. After the next clock cycle, the trigger bits will reset back to 0 because the I2S will have been signaled that an error has occurred in the buffer. A clock cycle after the address is incremented to 0x020, the XFC goes low to 0 and everything has reset again.



**Fig 4.5:** Bits trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr are Triggered

*4.5.2 trig\_generator\_testbench1.v*

The second testbench for the trigger generator differs from the first testbench by initializing the address first to the 12 bit address 0x00C. Like the first testbench, when wdata is less than the address 0x020, wdata is incremented by 1 and the XFC is set to 1. The overrun trigger bit is signaled every clock cycle when the XFC goes high. The underrun trigger bit is signaled through the duration of four addresses, or two clock cycles, and when the XFC is set to 1. The testbench in Fig 4.6 indicates the trig\_generator.v is fully functional because the overrun trigger bit is cleared every time data is trying to be written to the full I2S FIFO, and the underrun trigger bit is cleared every four addresses when the empty I2S FIFO is trying to be read.



**Fig 4.6:** Bits trig\_i2si\_fifo\_overrun\_clr and trig\_i2so\_fifo\_underrun\_clr are Triggered in Second Testbench

*4.6 C Code Involvement*

*4.6.1 initialize\_coeffs.c*

To initialize all 512 16-bit filter coefficients to a hexadecimal value of 0, a C program was generated in order to generate the thousand lines of code. Each coefficient is split into 8 bits, creating coefficients a and b. This created 1024 lines of verilog code that was generated into a filter\_coeffs\_initialized text file that would loop 512 times and print the correct formatting of each coefficient. The C program generated both coefficients a and b with a variable y incremented each loop cycle. The code outputs the two tabs such that when copying the code from the text file into Xlinix, it is formatted correctly. A pointer to the 16 bit hexadecimal 0 value is next to each coefficient in order to initialize all filter coefficients to zero.

*4.6.2 set\_coeffs.c*

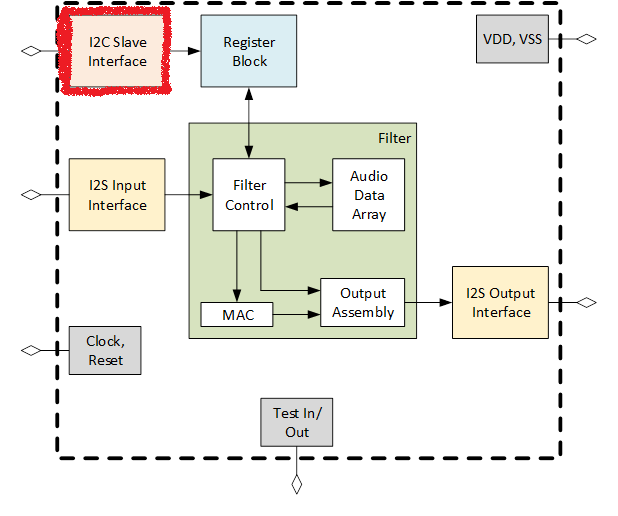
To assign all 1024 8-bit filter coefficients to the correct address, a C program was created to generate the correct number of tabs, case statement of the 16-bit hexadecimal address, and a new line of the name of the coefficient pointing to the 8-bit written data variable. In C, there is a compatible hexadecimal format that permits an integer to be incremented by hexadecimal 0x01 while still including the correct hexadecimal character formatting as hexadecimal should. After the address case statement, the correct name of the filter coefficient is expressed in which coefficients a and b are toggled between such that every other for loop iteration switches between coefficient a and coefficient b. The 1024 coefficients are properly formatted such that the contents of filter\_coeffs\_set.txt could be copied and pasted into Xlinix directly.

**Chapter 5: I2C Slave Interface - Whitley Forman**

*5.1 Introduction*

The I2C Slave interface of this chip serves a basic function to the system as a whole, to take filter coefficient values defined by the user outside of the chip and transfer them to the chip’s register block. As an additional verification functionality it can then read the values from the register block to ensure that they were written properly. The I2C bus was chosen over other busses such as RS-485, RS-232, CAN-bus and SPI because of it’s slave acknowledgement capabilities, multiple clock rates, low pin count and widespread usage in the industry. The function of the bus is to take in serial data from a master controller and convert it to a parallel format and then deliver it to the register block with the operation code, data and register address at the same time using a single strobe transfer for all parameters. If the operation is a read operation, then there is an added functionality to then take data information from the register block as parallel data and then transmit it back to the master I2C controller over the bus in a serial format. The block can be seen in Fig. 5.1 of the overall system chip diagram highlighted in red.

I2C was created and maintained by Philips Semiconductor now known as NXP semiconductor. Several updates to the original specification from 1982 have been made to keep up with the changing semiconductor industry to allow for faster transfer speeds up to 5Mhz from the original 100khz speed.



**Fig 5.1:** I2C Chip Diagram Highlight

*5.2 Requirements*

When analyzing the scope of the project and the *I2C-bus specification and user manual [2]* document UM10204 from NXP semiconductor, several decisions were made about the functionality of the I2C block in regards to the necessity of the project and the time constraints to implement them. Several functions were not included in this project and the overall requirements were defined before any RTL coding was started. A more concise definition of the current requirements can be seen in section 5.2.1 and the tradeoff study which decided what functionality to include and exclude is found in section 5.2.2.

*5.2.1 Current Requirements*

Below is a list of the requirements for the current design of the I2C block of the chip being made with reasoning and some details of each.

* The I2C-bus specifies that a slave of the bus must be able to respond to a master unit when called. There are 2 different slave address sizes to choose from, 7 bit and 10 bit, our group decided that the 7-bit address mode was optimal as there would more than likely only be 1 slave on the bus at any given time so the 10-bit addressing would not be necessary.
* The register address size is 12 bits so in the case of I2C where data is sent only in bytes, 8 bits, before an acknowledge bit, the address space must be broken up into 2-byte sized data transmissions separated by acknowledge bits.
* The data to be written to each register is 8 bits so a normal byte transaction is sufficient to transfer the data to be written or read.
* The data transfer rate of I2C comes in several speeds. For the needs of this project the standard mode speed of 100 khz maximum and fast mode of 400 khz maximum were considered. The chip clock frequency is to be 10 Mhz so the fast mode plus at 1 Mhz was considered to be too fast as the Nyquist ratio could possibly cause problems as far as sampling at 1:10, so the fast mode with a ratio of 1:25 was chosen to be the fastest speed. Also the group considered that with an address space of 12 bits and a data width of bits that all 4096 filter coefficients could be written in 0.0921 seconds, as seen in figure ##, which would be fast enough for our purposes.

**Fig 5.2:** I2C Full Coefficient Time Load Calculation

For this application, our chip needed only to be a slave for to a user.

* The burst write functionality is necessary to efficiently transmit large amount of sequential addressed data. Our I2C interface will be made to have an start address transmitted and then repeated bytes of data come in. Part of the functionality of the I2C block will be to increment the address associated with each byte of data by incrementing the address from the original address with each incoming data byte.
* The read functionality will be similar to the write functionality where the master will transmit an address to be read from and then the I2C block will be capable of repeated sequential read requests.
* The slave address of the chip could have been hardcoded into the chip, but it was decided by the group that having user selectable off chip switched for the 3 LSB of the address was appropriate. This will be realized through pull-up resistors or DIP switches in the FPGA board and test PCB that will be created when the final chip comes back from manufacturing.
* The transfer method from the I2C block to the register block will be done with a simple strobe. When the data, address and operational code are ready for transmission then a single bit for 1 clock cycle will go high and then low.

*5.2.2 Trade Off Study*

Several I2C functions found in the specification were left out due to the scope of this project. Below is a list of functions that were left out and short reasoning for them not being necessary.

* Clock stretching by a slave was considered unnecessary as our chip will not be running other processes during operation and there in theory will never be a time when it is busy and needs to delay transmission.
* Arbitration was deemed unnecessary as for our purposes we will be testing one chip on the bus at a time with a single master.
* Software reset was deemed unnecessary as this is an optional feature of I2C and not widely utilized.
* Bus clear was deemed unnecessary as this functionality would be useful in critical applications or a product for sale, but not for the scope of this project.
* Device ID was deemed unnecessary as our chip is not made by a manufacturer, is a single and only design by our team and will be the only revision.

*5.3 Top Block Interfaces*

The I2C Interface of this chip has only 3 external interfaces, the I2C master unit, the register block and the user definable slave address pins. Below is a detail of how each interface was implemented and in section 5.4.2 some of the previous designs that were changed.

*5.3.1 I2C Master*

The I2C master interfaces with the chips I2C slave module via a 2 wire interface using an scl and sda lines which stand for serial clock and serial data respectively. Both of these single line wires are active high signals using pullup resistors and the transitions of the signals are driven low by the transmitter during the transaction. The i2c\_scl input is only and input read by the slave and driven by the master, but the sda line has both input and output directionality to it. Although specifically called i2c\_sda\_in and i2c\_sda\_out these two input and output lines would meetup during the EDA tools floor planning and place & route stages to become one line by using a MOSFET open drain configuration as seen in figure 5.4.

Data coming in on the sda input line is sent in 1 byte increments so it was necessary that the deserializer be implemented in such a way that the first 2 bytes of data would be concatenated into a single 12-bit address register and the 4 MSB be discarded as they would all be zero.

During transmission of any data, the scl line acts as a clock and the sda line transition is only allowed during the low cycle of scl. This fact was a large driver in signal filtering and conditioning of both scl and sda lines. To do this it was decided that the scl line would be double ranked through D flip flops and the sda line would be triple ranked through D flip flops. This would stabilize the rise and fall times as well as filter out any spikes on each line that the logic circuitry would see. This also has a dual benefit as the signal transitions would now be matched in the chip’s clock domain making it easier to use in RTL synthesis.

The i2c\_sda\_out wire acts as the data transmission line for the data read request implemented by the serializer but also the acknowledgement sent by the slave when it sends an ACK bit by driving the sda line low after each byte received.

The I2C bus specification calls for many design details such as maximum bus capacitance, timing for different data conditions such as rise and fall times and frequencies. For the scope of the project for RTL synthesis rise and fall times and frequencies were considered as bus capacitance would not be a factor until EDA tools were being utilized as well as the test PCB being created. The frequencies for this block had already been chosen during the requirements phase of the project.



**Fig 5.3:** I2C block diagram in

***Needed***

**Fig 5.4:**Open Drain Schemati*c*

*5.3.2 Register Block*

Interface with the register block in contrast with the I2C serial interface is much simpler. For this chip there is input and output functionality but because we are using a parallel interface inputs and outputs are not shared on connections.

The output of the I2C to the register block consists of 4 lines, i2c\_op, i2c\_addr, i2c\_wdata and i2c\_xfc\_write and each has a specific role in the data transmission transaction. The i2c\_op wire is the operational code of the transaction and is either a high for a write and low for a write. The i2c\_addr wire is actually 12 bits wide and this designates the current register address of the register block to be written to or read from depending on the operation code. The i2c\_wdata wire is a byte wide and it has on it the data to be written during a write request and is neglected during a read request. When all data is ready in the operational code, address and data output wires to the register block the i2c\_xfc\_write, which is the strobe, is driven high for 1 clock cycle at which point the register block is told to capture the data, address and op code and perform its action depending on the operation code. If a write is requested, then the register write the data to that address and if a read is requested then the I2C block waits for the register to transmit back.

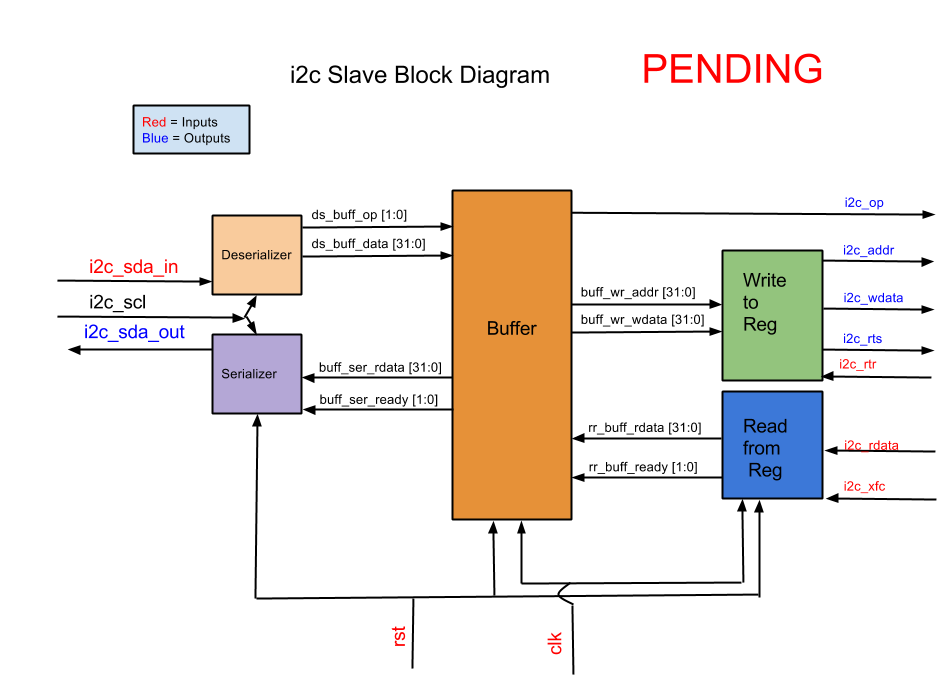
The input of the I2C block from the register block consists of 2 wires, the first being i2c\_rdata which is 1 byte wide and holds the data from the requested read address and the second being i2c\_xfc\_read which is the strobe of the transaction where the wire goes high for 1 clock cycle telling the I2C block to capture the data to be serialized on the isc\_sda\_out wire of the I2C master interface.

*5.3.3 Slave Address Pins*

The input address pins purpose is to select the 3 LSB of the chip’s slave address. This would allow for more than 1 device to be put on the bus. The 3 bits would be selected by simply driving a pin on the chip’s final package or a selected pin on an FPGA to be driven high and therefore would result a user defined LSB address. The other 4 bits of the address are coded into the RTL and cannot be changed. 1010 was selected as the 4 MSB as this address space was allowed and open according the I2C bus specification. The logic for defining the address was not clocked and at any given moment is exactly what is defined by the pines without any delay as it was deemed unnecessary to clock this logic.

*5.3.4 Design Changes*

Previously the register block interface was designed as a fully handshaked interface between the two blocks for both input and output between them. After much consideration about clock speeds, data transmission rates and if there was any need for it it was decided that simpler strobe approach made sense. When considering the clock rates, even at the high speed I2C bus frequency there would never be a need for a buffer or FIFO functionality as the data could be written much faster than the I2C bus could take it in. The register block could possibly write and read at a speed of 1 byte for every clock cycle at 10 Mhz while the I2C bus would only be taking in data at 1 byte for every 9 clock cycles at 400 Khz. There was a very large amount of time in between data transmissions that we felt it realistic that the simple strobe approach would work best.

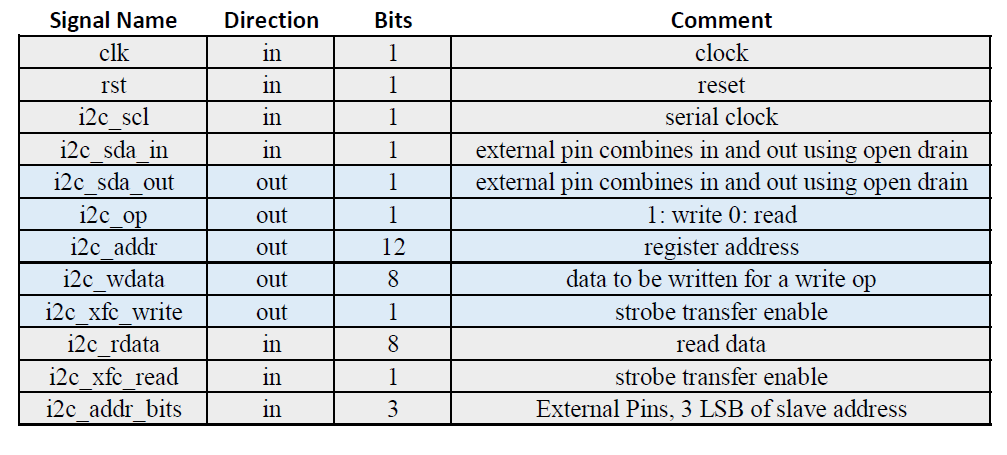


**Fig 5.5:** Previous block diagram version

*5.4 Register Mapping*

The interconnections of the I2C block that do not interface outside of the block connects all of the sub block together to form a cohesive unit. All of the data signals being transmitted can be seen in table 5.1 register map as well as the control signals of the block the block diagram in figure 5.3.

**Table 5.1:** I2C Register table



The data signal flow between the deserializer and the sequencer is similar to the interface between the sequencer and the register block i2c\_RW is the same as i2c\_op as well as serial\_data being the same as i2c\_wdata in width and information. The wire burst\_addr is similar to i2c\_addr in the fact that they are the same width and represent the data register address but burst\_addr is the first address specified by the I2C master during a burst write or read. The sequencer increments the address as each new byte of data comes in. This is all controlled by the state of the machine and what is happening in regards to the data transitions being sent on the i2c\_sda\_in and i2c\_scl interface wires.

*5.4.1 States*

The states of the I2C block is driven by events on the i2c\_sda\_in and i2c\_scl interface wires and how and when their transitions happen with regard to one another. After certain timed events in the amount of data byte transactions as well as start and stop conditions drive how the control logic of the block that can been in the four wires addr\_ack, data\_ack, slave\_ack and stop.

After a reset or a stop condition the block is an idle state. During this state the deserializer is the only active block and it is looking for a start condition on the sda and scl lines. When this occurs it is still the only active block and it starts taking in 1 byte of serial data from the I2C master interface as the slave address call from the master and check it against the actual slave address. If the address is not the same then the system stays idle looking for another start condition, but if it is a match then the state changes to look for the burst start address after sending a pulse on the slave\_ack wire to control the serializer to send a single ack pulse on the i2c\_sda\_out wire and tell the I2C master that the slave is waiting for data.

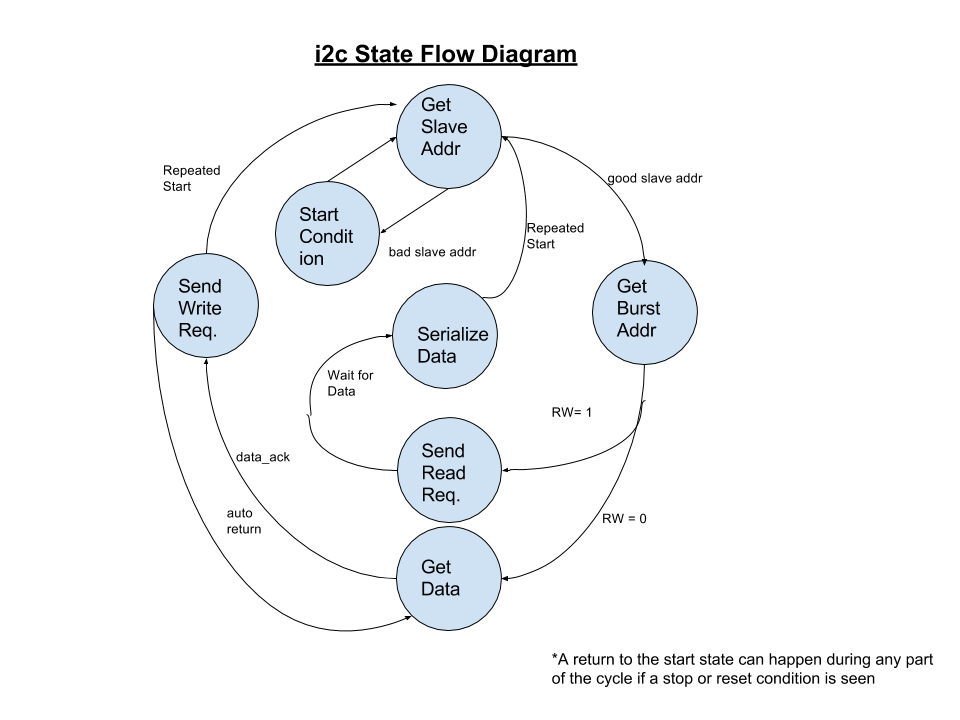
Once this state is active the deserializer takes in 2 more bytes of data and concatenates them together to become the 12 bit burst\_addr wire between the deserializer and the sequencer. Once the burst\_addr has been captured then the slave\_ack is driven high again as it does in between all received bytes of data to acknowledge to the master of the reception of the data. The addr\_ack also pulses so that the sequencer can take in the burst\_addr and use it for sequencing.

Once these 2 bytes of data have been taken in as the burst\_addr and the acknowledgement has been sent a state change is driven in the deserializer and now it knows that depending on the opcode bit that the data coming in will be of data to be filled into registers or that the address and opcode should be sent to the register block and a read was requested.

If a write request happens then after each data byte transmitted the deserializer pulses slave\_ack for the acknowledgement over sda as well as pulsing data\_ack to the sequencer to notify it that the data has been captured is on the serial\_data wire between the deserializer and the sequencer. Every time that the data\_ack is pulse then the sequencer updates the register address and the data and when it is ready it strobes i2c\_xfc\_write for one chip clock pulse to send the information to be written and then clears itself. This will happen continually until a stop or reset condition occurs. It is possible for a repeatedstart conditionto occur after the slave acknowledgement in which case the slave looks for a slave address again. During future testing with the FPGA and a whole chip test we will see if there is a necessity to have a stop condition for data overrun if the register address size is too large.

If a read was requested in the opcode, then the sequencer does not wait for data to be ready as it is not necessary and the sequencer will prepare the address and opcode to be strobe transferred to the register block. If a repeated start condition is send in lieu of an acknowledgement by the master, then this sequence will look again for a slave address and start the process over again.

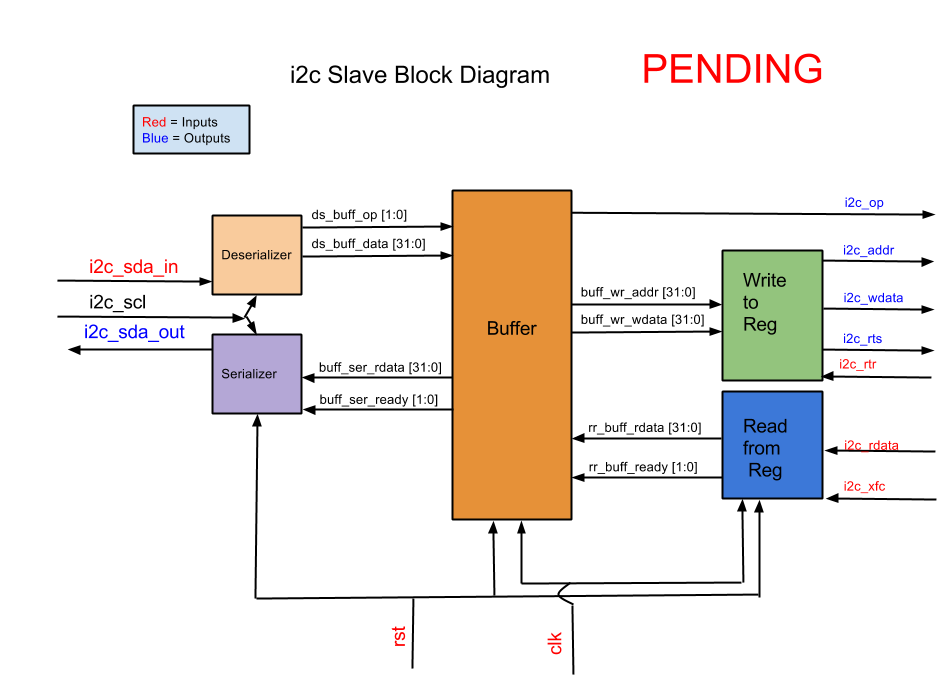
The states can always be altered by a reset from the chip or from a stop condition seen on the I2C master interface. Either of these will force a reset of the whole block and goes back to state 0 and look for a start condition.



**Fig 5.6:** I2CState Machine Flow Diagram

*5.4.2 Design Changes*

Previous designs of the control state included transfer complete signals and did not include other acknowledge signals by using the sequencer as a state machine as well as a data buffer. This over time was ruled out as a robust control system as the deserializer was in fact driving the states and therefore has become the control center for the block using the addr\_ack, data\_ack, slave\_ack and stop signals to control other blocks.



**Fig 5.7:** I2C Previous block diagram versions

*5.5 Sub Blocks*

The I2C block was broken up into 3 distinct parts including the deserializer, the sequencer and the serializer. Several iterations of the design were made as progress was made during the planning stages and this design came out as the best implementation. These design considerations are outlined in section 5.5.4 and details about the design of each sub block is outlined below in the following sections.

All subblocks use always statements for posedge clock and negedge reset to control when a reset happens. If statements are used inside of the always blocks to decide what happens based upon the state of each sub block. All always statements include as a first if statement a reset capability of all registers to be sent back to initial conditions.

*5.5.1 Deserializer*

The deserializer of this block became the defacto state control machine for the other sub blocks as it would be being controlled by the I2C master. The deserializer block has to perform several function other than simply deserializing the data coming from the master.

The deserializer must filter and condition the isc\_scl and the i2c\_sda\_in wires in order to make them usable in the verilog code. Edge detection pulses were extracted from each wire after it went through the D flip flop ranking described above. States of the wires were also made during this conditioning step to have chip clock synched signals that were highly reliable during RTL synthesis.

To deserialize the data there were 2 known possibilities in verilog to accomplish this. The first being a shift register and the second being a case statement. Although the shift register was simple, the case statement seemed to give more control of multiple events occurring at the same time. The case statement is very similar to using a MUX where individual bits of a register are populated as a bit counter increments and sequential case statement become active. This allowed for the 8th case statement to include control signals to acknowledge, state change, and transfer data to different registers. Although thought and logic intensive, this gave the designer a great deal of ease when it came to doing the multiple functionality of the block. The slave address, register address and data are all deserialized in the same way, but the different states of the machine drive how the control signals are handled.

The deserializer has to look for start and stop conditions coming from the I2C master. These conditions either turn on the deserializer to start deserializing data as well as checking the slave address coming in against the slave’s programmed address as well as extract the read or write enable bit from the LSB of the slave address sent and assign it to the i2c\_RW wire. Once this happens then a state change occurs and an acknowledgement signal sent to the serializer if the the addresses match or the deserializer resets itself and continues looking for another start condition.

Once a state change occurs from the slave address check then the deserializer goes into register address fetch mode where it takes in 2 bytes of serial data from the I2C master and then concatenates them together to form the 12-bit register burst\_addr signal. Once this occurs then another acknowledge signal is sent to the serializer in between the 2 bytes sent and after the second byte sent. At the same time the addr\_ack signal is enabled going to the sequencer sub block to inform it that the address is ready to be taken in.

Depending on the the i2c\_RW bit at the strobe of the addr\_ack signal the deserializer will either reset itself if a read is requested as it has no purpose in the future of the transaction or it will go into data fetch mode. During data fetch mode the deserializer will continually deserialize data write it to the serial\_data wire and then strobe the data\_ack wire and then loop itself for another byte of data until a stop condition occurs.

The deserializer also has the capability to send out a stop signal when either a stop condition occurs or a reset occurs to ensure all other sub blocks have reset themselves.



**Fig 5.7:** I2CDeserializer Block Diagram

*5.5.2 Sequencer*

The sequencer block acts a slave of sorts to the deserializer block. Its sole purpose is to transfer data, address and operation code information to the register block. Depending o the acknowledge signals coming in it acts accordingly.

The sequencer acts similarly to the deserializer in that it conditions the acknowledge signals into single cycle pulses so that a signal is not captured more than once with the system clock always blocks. From this it will capture the address and data when these strobes occur.

This data is handled in two separate ways the first being during a read and the second being a write.

During a read opcode the sequencer simply captures the burst\_addr into the i2c\_addr output wire connection and then strobes the i2c\_xfc output wire and then after that resets itself. This is done using 3 if statements under the main always statement. A state variable for this sequence is used called xfc\_ready which goes high when the data is captured so that at the next clock cycle the strobe occurs. Using the op code as part of the if statement makes sure that this only occurs when a read is requested. The last step is to reset each signal again using an if statement and the i2c\_xfc to control it. This is like using a mini state machine to control how the sequence of the transaction with the register block.

During a write operation a very similar sequence occurs, but the strobe does not occur until a data-ack signal strobe has been seen. Using a similar if statement setup and the same state variable. this same variable is used because the if statements can be differentiated by the operational code. This sequence takes 4 if statement to complete and the first is the same as the read sequence where the address is captured as well as the opcode. Next the sequence waits until a data\_ack signal comes in to capture the data and then to enable xfc\_ready. During this time another variable comes into play. The address is rewritten to have an increment variable added to the address. For the first instance of this if statement a 0 is added, but later the increment itself increments by 1 so during a burst write cycle the address is incremented each time this if statement goes active. The next if statement then strobes the i2c\_xfc for the opcode, address and data to be sent to the register block. This sequence will continue to loop and not reset itself until a stop condition is seen as a reset or a stop signal from the deserializer.



**Fig 5.8:** I2CSequencer Block Diagram

*5.5.3 Serializer*

The serializer, like the sequencer, acts as a slave of sorts to the register block and to the deserializer. Its sole purpose is the either send acknowledge bits during the 9 bit of the byte transaction over the sda line or to serialize the data that it receives from register block. It has no output except that of the sda line.

Similarly, and actually the exact same way, the serializer conditions that data on the sda\_in and scl lines so that it is aware of when it can make transitions to the sda out line which is during scl low cycles.

Start and stop conditions are handled by using the slave\_ack and i2c\_xfc\_read signals. If either one of these is seen, then the system goes into a start condition.

To serialize the data there is an if statement under an always block that uses a new variable of serialize\_done and the i2c\_neg\_edge\_pulse signals. Inside of this if statement there is case statement that will then write to the i2c\_sda\_out line to serialize the data using the captured i2c\_rdata wires from the register block during the i2c\_xfc\_read strobe sent by the register block. This case statement also uses a counter to sequentially increment the case statement MUX and at the last of the 8 bits it then writes to the serialize\_done register. This register will then deactivate the if statement so no more data is written and causes a stop condition inside the sub block.

If the acknowledge signal is to be serialized, then another if statement under the same always block is enabled using the slave\_ack signal and the i2c\_neg\_edge\_pulse. Only 2 statements exist under this if statement and tone is to set i2c\_sda\_out high and then activate the serialize\_done register. this will cause a stop condition and the whole sub block goes back to waiting for a signal from the deserializer of the register block.

There is essentially a mini state machine inside of the serializer that dictates how the data is serialized taking into account what is being serialized, an ack or data, and if that has occurred or not.



**Fig 5.9:** I2CSerializer Block Diagram

*5.5.4 Design Changes / Previous Sub Blocks*

Several changes took place during synthesis of the verilog code for all of these blocks. Much of it had to do with the state changes and control. Some of the iterations were very difficult and had many variables and the latest design uses a minimum of them. This allows for easier control of what each block was doing and how they interact.

Handshake interfaces were the first iterations of the design, but it was realized that this was too complex and prone to errors in coding. So a simple strobe implementation was used in between all sub blocks and the register interface. This made for a streamlined state control of the system.

*5.6 Test Fixtures*

Testing this system called for lengthy test bench modules which could simulate serial data coming in. A rubric for a serial data stream was created and then different values for data and timing were changed to show different clock cycles and data entries. The section below outlines the test benches that have utilized so far for testing each subblock and the top block.

*5.6.1 Deserializer*

For the deserializer a test bench was written using the rubric described above. To test this sub block we wanted to test multiple aspects of it. First was to test that it would only acknowledge and state transition when the correct address was called. Several different parameters were written for the user definable address pins to vary the slave address and then the first byte of the serialized data was changed to match or not match this. It was found that the device will only state change and acknowledge to the correct slave address. During this test it was also seen that the opcode bit was successfully extracted only when the slave address match was made.

The next test performed was the register address data and acknowledgement. After several tests it was seen that any data coming for the address bit was extracted successfully and that the correct state change and acknowledge signals were sent. It should be noted that a design flaw was seen during this process that the design called for a 12-bit address and was coded to take all 12 bits at once. This was incorrect and changes to the state change and size of deserialized data must be changed. A plan for this has already been outlined above and the changes will be made immediately to reflect the actual functionality of the i2c bus specification.

After the address deserialization it was tested whether the deserializer would change states correctly depending on the opcode. It did correctly revert back to an idle state after a read request after the address had been transferred and that it changed to data fetch mode when it was a write request.

Furthering the serialized bit stream defined in the test bench it was seen that if data kept coming in after the address fetch mode during a read request that nothing happened as the block was looking for a start condition. If it was a write request the block would continually capture serial bytes of data and transfer them to the sequencer using the data\_ack wire.

Several different scl and sda\_in frequencies were used and it was noted that it will work at very low frequencies down to 100 khz and was tested up to 400 khz as per spec.

*5.6.2 Sequencer*

The sequencer test bench was fairly simple as compared to the deserializer since it did not utilize serial data. The sequencer handled and conditioned data coming in correctly and managed to transfer it in 3 clock cycles once the ack signals went high. This system did not need to be tested for frequency so much because of the fact that the time between strobes would be large.

*5.6.3 Serializer*

The serializer block is still having trouble with its testing. The sda\_out line seems to not be initializing correctly and no data comes out. A test bench to show register data has been written and the block code as well as the test bench code is being analyzed for flaws.

*5.6.4 I2C Top Block*

The top block test bench utilizes serial data simulating an I2C master pushing data over the sda line as well as the controlling the frequency of the scl line. during and instance of the top level simulation using a schematic capture method all signals went through to the register block as expected from the sub block test, but this uncovered the problem with the serializer block. Time had to be devoted to writing this document so further evaluation, testing and corrections were not possible due to time constraints. Plans have been made to correct issues with the deserializer burst address functionality and time has also been allotted to finalize the serializer block.

**Chapter 6: Budget - Zachary Nelson**

The College of New Jersey’s School of Engineering allocates each senior project $100.00 for every student member and the projects need to put in a request if they require a higher budget. Since our project has 5 members, we were given a total budget of $500.00. The list of materials that were required for this project and the financial budget details are included in Appendix B. Since our project uses a large amount of software, our list of materials was broken down into software and hardware sections.

The first piece of software that our project required was ISE Design Suite 14.7 so that we could synthesize and analyze our hardware description language designs. This program is installed on the laboratory computers and was able to be installed on all team member’s personal machines at no cost. The second piece of software that was required was CORE 9 University and was used to document requirements and formulate use cases for the project. The students were given temporary licenses to this piece of software as part of the CORE University Program. The third piece of software that we used was Git/GitHub version control. We used this to keep track of the source code revisions and to store all other non-confidential materials. A free GitHub organization can be made if the files are made public. Since we could not make the fabrication process documents public, we used Dropbox to store all confidential files. Our project used Microsoft Project 2013 for project management and was given to us at no cost by the Computer Science Department’s DreamSpark program. Mentor Graphics is the EDA tool that we plan to use and is free for us because TCNJ already has a subscription for it. Mentor Graphics requires the Linux RedHat operating system which is also free for us to use because TCNJ has a subscription for it.

The only piece of hardware that we have purchased up to this point it the Nexys 4 Artix-7 FPGA Board. The board was purchased and received from Digilent for $192.41. The UDA 1380 board needs to be purchased in order to code and decode audio from the I2S interface. We estimate the total cost for one of these boards to be no more than $20.00. We also need to purchase crystal oscillators so a stable clock signal can be provided to our integrated circuit. We estimate that the total cost for multiple crystal oscillators to be no more than $20.00. The last piece of hardware we will be using is the CY8CKIT-050 PSoC 5LP Development Kit. The laboratory stock room already has multiple PSoC microcontrollers that we are allowed to use. Overall, our project is well under-budget and we currently anticipate to have at least $267.59 leftover after all purchases have been made.

**Chapter 7: Schedule - Zachary Nelson**

The schedule for the fall semester was created in September 2015 based off of the information recorded in CORE 9 Univeristy. The schedule was created in Microsoft Project 2013 in the form of a Gantt chart and is included in Appendix B. The schedule includes a name, task ID, status, duration, start date, finish date, and an assignee for all tasks.

The first main task on the schedule was System Design and started on May 14th, 2015. The System Design task consisted of general planning, every person creating a block document for the module they were in charge of, documenting requirements and use cases in CORE 9, and documenting the RTS/RTR and XFC protocols to use. The System Design was successfully completed on September 14th, 2015. Creating the schedule and installing Microsoft Project 2013 were listed as separate tasks and were also completed by September 14th, 2015. The next major task was the the installation of EDA tools. This involved determining the requirements for Mentor Graphics, installing the RedHat Linux operating system, and creating a test design to ensure the tools are properly working. As shown on the Gannt Chart, this task is yet to be completed because there were numerous issues with the software license and hardware problems with the laboratory computer that they were being installed on. The next task was to create a web page for the group and is located at http://tcnjchip.pages.tcnj.edu/.

The majority of our time fell into the task of RTL Design and Testing. Each person’s module was divided into submodules and were listed on the schedule. The goal was to develop and test each submodule separately and then integrate the submodules into the 5 main modules. More testing was supposed to be done with these five modules and then integrate the design into one project to implement on a FPGA. As seen from our schedule, we fell behind in this task and look to pick up on lost time over winter break. Instead of having a fully functional design, we created a single project that has all the correct interfaces between the modules and the internal workings of some of the modules need to be created or fixed. Overall, we recognize that we are slightly behind schedule but are confident that we can recover by putting extra time in over winter break.

**Chapter 8: Conclusion**

The intent of this project is to design an ASIC that is capable of digitally filtering an input audio stream. The chip will allow 512 filter coefficients to be uploaded in order to define the characteristics of the filter. Since the design will not be submitted for fabrication until March 2016, the chip will probably not be returned to us in time for graduation. In order to show that we have a functional design, we have proposed also implementing the design on a FPGA.

Currently, the entire system has been designed and documented and we are in the process of finishing the coding and testing of the system. More specifically, the I2S input interface has been completely coded and all the submodules have been tested. The only task left to do for this module is to perform top-level testing of the entire I2S input interface. The I2S output interface has also been completely coded and all the submodule have been tested. Again, top-level testing needs to be performed for this block. Certain parts of the filter block have been coded, but a functional top-level module still remains to be created and tested. The trigger generator aspect of the register block has been coded and tested but the coding and testing for the rest of the block still needs to be done. The I2C interface has been completely coded and the testing for the submodules and top-level module still needs to be done.

Even though our group is behind schedule, we feel like we have made significant progress because the entire system has been designed and we have all become comfortable with writing Verilog code. We believe that finishing the RTL design and testing over winter break will allow us to implement a fully functional design on an FPGA board by mid-January. We can then focus our time on using the EDA tools to prepare for the March fabrication deadline. In conclusion, this is an extremely challenging yet interesting project and we are planning on putting in extra time over winter break so we can have a quality design ready for fabrication.

**References**

[1] Mosis.com, ‘About Us’, 2015.[Online]. Available: https://www.mosis.com/what-is-mosis. [Accessed: 11 November 2015].

[2] nxp.com, ‘*I2C-bus specification and user manual - UM10204*’, 2015. [Online]. Available: <http://www.nxp.com/documents/user_manual/UM10204.pdf>. [Accessed: 10 November 2015]

**Appendix A: Project Overview**

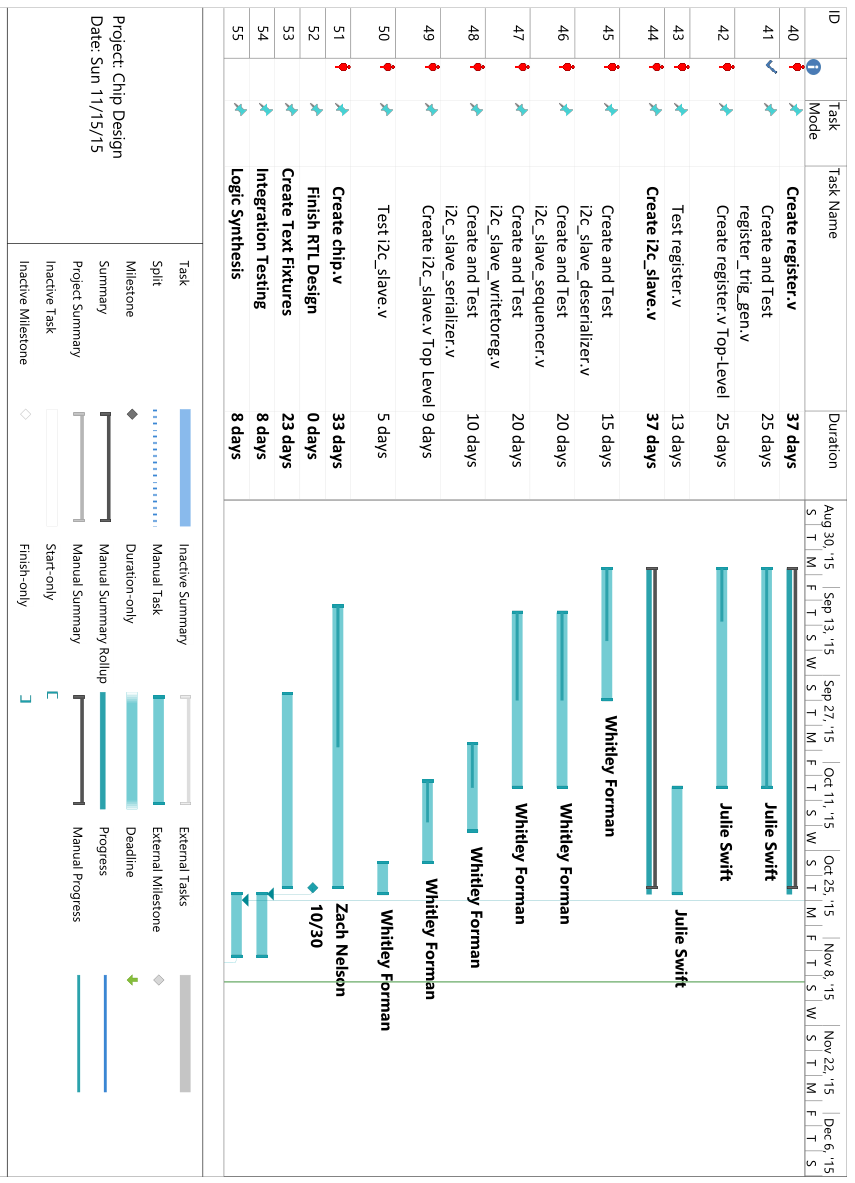
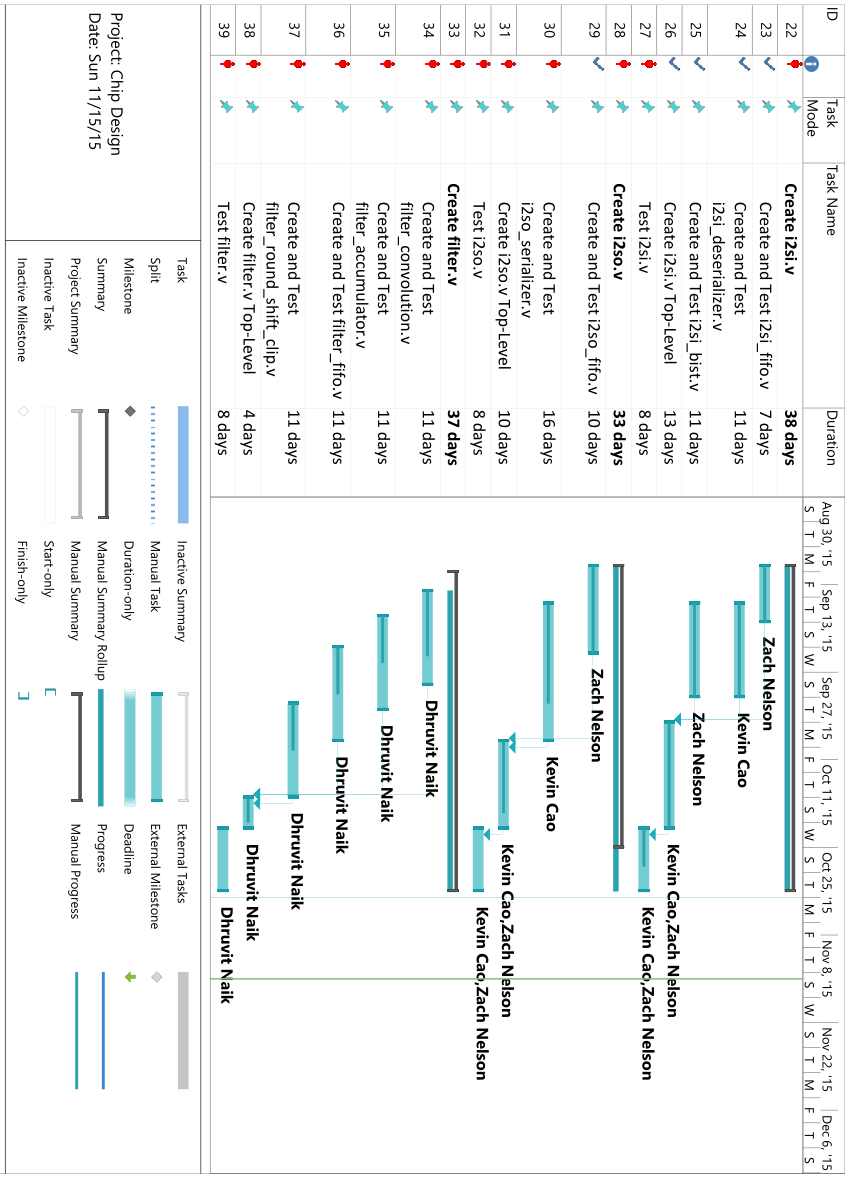
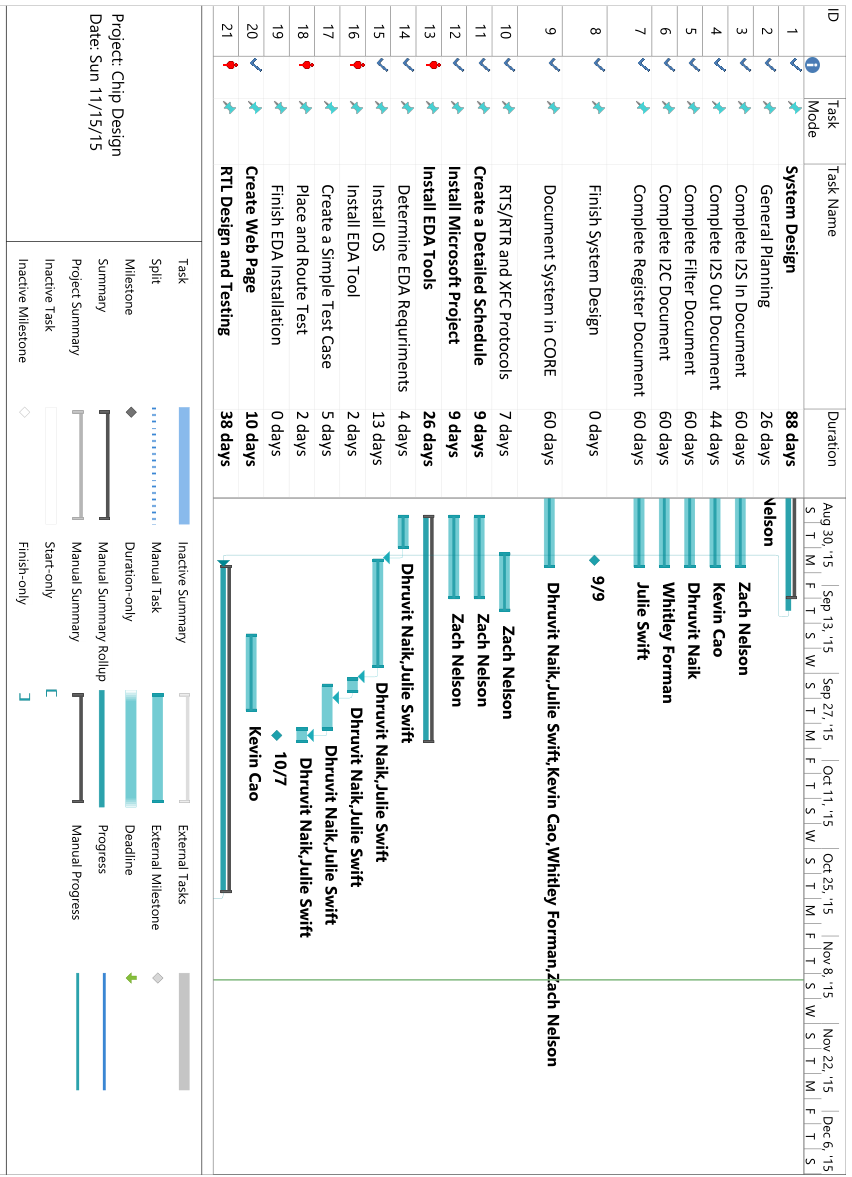
1. Biography

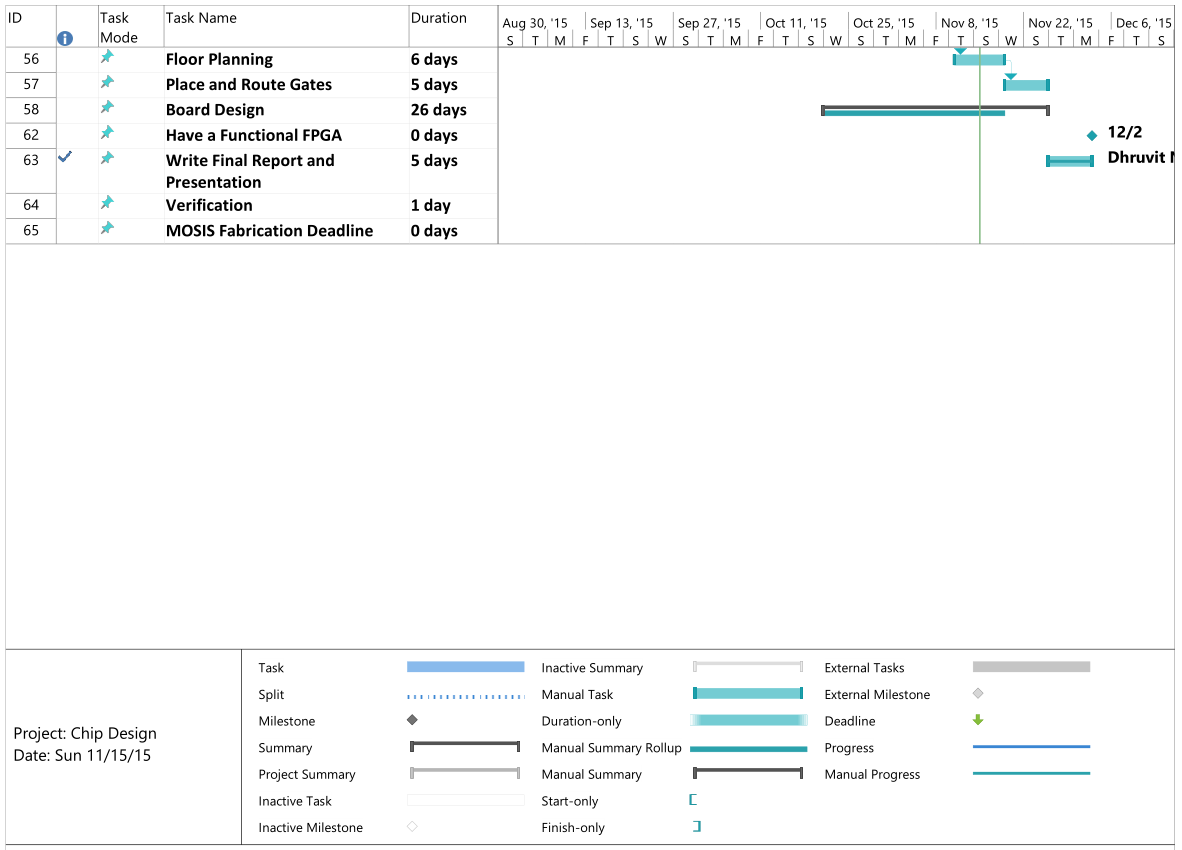
**Biography:**

* Kevin Cao
  + Kevin is from Morris Plains, NJ and is a Computer Engineering major who is planning to enter the workforce after graduating at TCNJ. Kevin has interned as a software engineer at LGS Innovations, located in Florham Park, NJ.
* Whitley Forman
  + Whitley is from Ocean Grove, NJ and is an Electrical Engineering major who is continuing his education in the electrical field and will be obtaining his Master Electrician’s license after graduation. He is planning on using his new knowledge and experience with his current business to expand into new ventures.
* Dhruvit Naik
  + A resident of Mount Laurel, NJ, Dhruvit is a senior Computer Engineering major at TCNJ. He plans on entering the workforce after graduation and continuing his education in the coming years. He is the Vice-President of a startup, ThinkSOAS, INC.
* Zachary Nelson
  + From Cream Ridge, NJ, Zachary is a Computer Engineering major who is planning on attending graduate school after graduation. He has experience as a software engineering intern at Teletronics Technology Corporation and an undergraduate student researcher at TCNJ as part of the MUSE program.
* Julie Swift
  + From Robbinsville, NJ Juliann is a Computer Engineering major who is planning on entering the workforce after college. She has experience as a AutoCAD Designer interning at Linearization Technology and a Software Development Life Cycle Analyst interning at Educational Testing Services. She participated in the undergraduate student research program, MUSE, at TCNJ.

**Appendix B: Team Management**

1. Gantt Chart
2. Meeting Minutes
3. List of Contacts
4. Material List
5. Financial Budget





**Chip Requirements (June 11th Meeting)**

**Members in Attendance:** Dr. Pearlstein, Julie Swift and Zachary Nelson

* **Overall Goal**
  + **Produce an Audio Processing Integrated Circuit**
* I2S Interface
  + Audio Input: 2 channel stereo channel I2S (master interface)
  + Support audio input sample rates of 8kilosample/sec – 48kilosamples/sec
  + Output is the same sample rate as the input and I2S
  + Digital audio bit clock and the word select (ws) line will be controlled from master
  + Input and Output will be 2 channel 16 bits
* I2C Interface
  + Support single master configuration
  + 7-bit addressing and we will consume entire I2C address space
* Uses an external clock input
  + Clock frequency will be a minimum of 1200 times the audio sampling rate
  + Maximum clock rate will be 100 MHz
* Has an external reset pin
  + Power on reset
* Register Block
  + 512 bit frequency coefficients
  + 10k register bits
  + Register fields will include
    - Source select bit
      * Allows user to select between I2S and BIST (built in self test)
    - Filter Bypass Bit
      * 0 doesn’t bypass, 1 bypass
    - 512 16-bit signed coefficients stored as 2’s compliment
      * The effective radix point of the coefficients (4 bit number for data point, 4 bit number for coefficient)
    - Read only status register bits
      * Overflow/saturation detector – audio clipping
      * Input FIFO overrun
      * Output FIFO underrun
    - Control Bit Fields
      * 1 sticky bit to clear overrun
      * 1 sticky bit to clear underrun (stay until you clear them)
      * Clear Overflow Flag
      * Filter Order to Support
        + 9 bit number to represent 1 to 512
  + Presents an array of registers for hosts control and status monitoring (through I2C read and write operations).
* Provide built in self test function
  + Test gadgets for I2S
    - Test by converting I2S to audio coeffs
    - Audio coeffs to I2S
* Chips made by service call MOSIS
  + IBM7RF process, geometry, drawn gate length, 180nm gates, mixed signals,
  + Chip Area: no more than 3 by 3 squared mm
* Filter Audio
  + Implement an FIR filter on the input data based on the coefficients stored in the programmable registers
    - Programmable from 1-512 taps
    - Filter Order Control (9-bit number)
      * Support filter from 1-512 taps
    - Maintain intermediate precision of 4
* Produce a microcontroller platform to configure the audio input and output modules
  + UDA 1380
  + Produce a test fixture to show that the chip works
    - Sample analog audio and covert it to I2S
    - Receive I2S and convert it to analog audio
    - Allow user to create filter coeffs and upload them to the chip that was designed
    - Has to have software to configure the analog audio subsystem
    - Parametrize low pass filters, high pass, band pass, and comb filters
      * Use a button on a PSOC?
      * Could use slider on PSOC to change the frequency and upload the new coeffs to the chip
    - Create a board that we can plug the chip into to do testing
      * Bread board?
    - Will not have to design a printed circuit board for this chip

**June 18th Senior Project Meeting**

**Armstrong 144, 3:30 P.M – 5:05 P.M.**

**Members in Attendance:** Dr. Pearlstein, Zachary Nelson, Julie Swift, Whitley Forman and Dhruvit Naik

* Went through the CORE requirements hierarchy for the chip **(System Design Step)**
  + Zach will make the top-level requirements the elements from the chip block diagram so that the requirements are broken down in a more logical way
  + Zach will make an Enhanced Functional Flow Block Diagram (EFFBD) on CORE with the functions that were generated as part of creating the requirements
* Discussed the design flow of creating an integrated circuit
  + A figure illustrating this is on GitHub under Chip-Design/proj\_asic/docs/design\_flow.png
  + Major Steps
    - System Design
    - RTL Design
    - Logic Synthesis
    - Design for Test (DFT) Implementation – may be able to skip this
    - Floor Planning
    - Place and Optimization
    - Routing
    - Verification
* Deadline for the chip
  + November 30th: We would get the chip back in the Spring semester and be able to see if it works
  + March 2016: Would get the chip back after graduation and would implement our design on a FPGA during the Spring semester instead
* Dropbox will be used for storing private files
  + The rest of the code and documents will be stored on the public GitHub account
* Assigned Verilog modules to group members **(RTL Design Step)**
  + Everyone will start working on the modules over the summer
  + Zach: i2s\_in.v and i2s\_out.v
  + Julie: register.v
  + Whitley: i2c\_slave.v
  + Dhruvit: filter.v
  + Kevin: chip.v
* Julie and Whitley need to hand in the NDA forms (electronically or in person)
* Kevin and Dhruvit need to accept the invite to the GitHub account

**July 7th I2S Senior Project Meeting**

**Dr. Pearlstein’s Office, 2:00 P.M – 5:10 P.M.**

**Members in Attendance:** Dr. Pearlstein and Zachary Nelson

* Register block bits will either start with “ro”, “rf”, or “trig”.
  + ro label for input.
  + rf label for outputs.
  + trig label when the bit causes something to trigger.
* Add the bit i2s\_in\_en to the I2S\_IN block.
  + Low to high: coming out of reset.
  + High to low: goes back into a reset state.
* Need variables to characterize the BIST saw-tooth wave.
  + rf\_bist\_start\_val (16 bit signed value) – start value.
  + rf\_bist\_inc (8 bit integer) – increment between 0 and 255.
  + rf\_bist\_upper\_limit (16 bit signed value) – upper limit.
* We will create a synthesized sclk by dividing the system clock.
* Make all the flops clock with the system clock (always statements must be with clk).
* We will have a register holder that takes in one bit at a time.
* Add trig\_i2sin\_fifo\_overrun\_rst bit
  + Example:

if (ro\_fifo\_overrun) 🡪 happens when rts=1 and rtr=0

ro\_fifo\_overrun = 1

else if (trig\_i2sin\_fifo\_overrun\_rst)

ro\_fifo\_overrun = 0

**July 30th Senior Project Meeting**

**Dr. Pearlstein’s Office, 2:00 P.M – 3:40 P.M.**

**Members in Attendance:** Dr. Pearlstein and Zachary Nelson

* Discussed how register.v will use register addressing to access data.
* Discussed cell utilization and a rough approximation about how much area our chip will use (60% utilization)
* Went through **register\_map.xlsx** with Dr. Pearlstein
  + Added new registers
  + Fixed register addressing
  + Added missing default values
* Added register fields that enable clipping for the filter and one that shifts the number of bit positions after the filter accumulator.
* Reviewed the block diagram of the **i2si.v** block.
  + This diagram seems to be correct and the block can start being designed and verification tests can be created.
* Discussed details of how the **i2so.v** will work.
  + Clock divider will go in this block
  + Zach wrote down a quick block diagram
* Block Documents Status
  + register.v – not started
  + i2si.v – near completion
  + i2so.v – not started
  + filter.v - not started
  + i2c.v - not started
* Zach and Kevin will switch roles.
  + Zach will be responsible for chip.v and can help with all blocks (especially i2s blocks)
  + Kevin will be responsible for the i2s blocks.

**September 2nd Register Senior Project Meeting**

**Dr. Pearlstein’s Office, 1:00-2:30**

**Members in Attendance:** Dr. Pearlstein, Zachary Nelson, and Julie Swift

* Project Goals Slide
  + Why are we doing this project?
    - This is not a common project for undergraduates because of the high cost associated with the fabrication of a chip.
    - Describe the MOSIS Education Service
    - Free fabrication for 3mm by 3mm
  + Also refer to this as VLSI
  + Explain the difference between an FPGA and an Integrated Circuit
  + Why did we choose to process audio as our application?
    - Complex and interesting enough
    - Not too hard so that this project is able to be completed
  + Take out parameterized filters
  + Put a picture of a chip
* Chip Overview Slide
  + I2S is digital but represents an analog signal
* DropBox is for confidential files while GitHub is for everything else
* I2C Slide
  + sda\_in and sda\_out are both interfaces
* Filter Slide
  + Fix the summation sign
* DFT Slide
  + We will most likely not be doing this task but it is good to have as background information.
* Gate Level Simulation
  + We will not do gate level simulation that intensively.
* Place and Route
  + We will not manually do the place and route, the EDA tools will do it for us.
  + We may be doing manual floor planning.
* Create a Block Level Testbenches Slide
  + We will have tb for each individual block as well as the overall chip
* Discussed interfaces for the register.v module
* Discussed a block diagram of the register.v module.
  + Ask Julie or Zach for more specifics
* MOSIS information
  + We can order one lot of 40 chips
  + We will fab using the Global Foundries 180 nm CMOS (7HV) process
  + MOSIS technology code for the 7HV process is GF\_7HV
  + Customer Submission date for 7HV is **March 7th, 2016**

**September 2nd Senior Project Meeting**

**Armstrong Hall 137, 3:30 - 4:15 P.M.**

**Members in Attendance:** All Members

* Add a presentation slide on input/output cells
* The Format of the weekly meeting will be:
  + Discuss what the team has done over the past week
  + Discuss what the team is going to do over the next week and beyond
  + The most important thing is to discuss any roadblocks that we have
* One goal is to have the design implemented on a FPGA by the end of the fall semester
* Some tasks that can take place during the spring semester:
  + Place and Route
  + Verification
* We need to adopt a project management tool and create a schedule that is more sophisticated than the one that Dr. Katz has provided us with
  + One potential tool that we will look into is Microsoft Project
  + Be specific enough for about 3 milestones per week
* Manage the amount of time being put into senior project
  + Put the needed amount of time into the project but if we find ourselves each putting much more than 6 hours/week, we may need to narrow the scope of the project.
* Grading for Senior Project
  + 2 students will be graded by Dr. Hernandez and 3 students will be graded by Dr. Pearlstein
  + The rubrics for Senior Project I and II was distributed by Dr. Katz in an email
* What FPGA will we use for the project?
  + We could use one that is already in stock
  + We could purchase one with the $500 that the group has
* Julie and Dhruvit will be responsible for EDA Installation
  + Step 1: Figure out the EDA requirements and install OS
  + Step 2: Install EDA
  + Step 3: Simple test case with libraries from MOSIS and doing a place and route test
* We need to discuss who will be responsible for configuring the test fixtures.
* We will hold off on the website design until we hear more information.

**For Next Week:**

* We need to create a detailed plan/schedule for the project using a tool like Microsoft Project
* Everybody needs to finish their block documents for their specific modules
* Everybody needs to add the information about their module to the CORE 9 project

**September 9th Senior Project Meeting**

**Armstrong Hall 137 and 144: 4:00 – 5:15 P.M.**

**Members in Attendance:** All Members

**Next Week’s Work Plans:**

* **Zach**
  + Fix Schedule
  + RTR/RTS and XFC Protocol
  + Buying an FPGA and seeing if the school one will work
  + CORE for I2S Blocks
  + Start coding one of the I2S submodules (BIST Generator?)
* **Whitley**
  + Start coding the I2C module
  + CORE 9 for I2C Requirements
* **Julie**
  + EDA Tool Installation
  + Complete Register Block Documentation
  + Start coding the Register module
  + CORE 9 for Register Requirements
* **Kevin**
  + Complete one submodule for the I2S Block (Deserializer?)
  + Create test benches and test this submodule
  + CORE 9 for I2S Blocks
* **Dhruvit**
  + EDA Tool Installation
  + Start coding the Filter module
  + CORE 9 for the Filter module

**Meeting Notes:**

* The RTS/RTR and XFC protocols need to be in one place
  + Zach assigned as the owner to this
  + Should be finished by next Wednesday
* EDA Tools should be installed and a simple place and route test should be completed by next week
  + We will be downloading Mentor Tools
  + An action item is to contact by email or in person Mike about installing the tools
* Things that need to be added/changed for the Microsoft Project schedule
  + Creating test benches
  + Create test cases for individual blocks
  + Creating test cases
  + Correction: Microcontroller not “board” will be linked to the chip
  + Shorten the milestone names
  + Block level synthesis
  + Full chip simulation
* CORE requirements and use cases can be integrated to write test specifications
  + The new CORE license works
* Dr. Hernandez will give feedback on the schedule
* Dhruvit and Julie have access to 144-B
  + This is where we will be installing the EDA tools
  + Can we remotely connect to this computer-Ask Mike?
* Chip.v
  + Will instantiate everyone’s modules
  + We will need to create input/output submodules
* Create model in another environment that takes in the register states and inputs and produces the correct output
  + An end to end test case
  + This is used to verify that we are producing the correct outputs in our chip
  + Does not need to be the most sophisticated model due to time
* Test benches will be stored on the “tb” folder on GitHub
* FIFO will be very similar for all blocks
  + The only difference will be the size and the width
* Need to buy crystal oscillators (10, 20, or 100?)
* Whitley will do board design in spring semester
* PSoC configuration will take 1-2 weeks of time
  + Whitley may be assigned to this task

**September 23rd Senior Project Meeting**

**Armstrong Hall 137: 3:30-4:00 P.M.**

**Members in Attendance:** Dr. Orlando Hernandez, Zachary Nelson, Julie Swift, Dhruvit Naik and Kevin Cao

**Last Week’s Work:**

* Zach
  + Cleaned up RTS/RTR and XFC Protocols
  + Updated CORE 9 Files
  + Updated Schedule on Microsoft Project
    - Everyone agreed on due dates
    - Created a calendar on Google
  + Continued working on i2si\_bist\_gen.v
* Kevin
  + Continued working on i2si\_deserializer.v
  + Looked into setting up the project website
* Dhruvit
  + Continued working on filter\_convolution.v
* Julie
  + Continued working on register block document and register.v
* Whitley
  + Continued working on i2c\_slave\_deserializer.v

**Next Week’s Due Dates:**

* Friday, September 25th
  + Install Linux on Machine in Room 144B (Dhruvit and Julie)
* Monday, September 28th
  + Finish Design and Testing: filter\_convolution.v (Dhruvit)
  + Install EDA Tool on Machine in Room 144b (Dhruvit)
* Wednesday, September 30th
  + Finish Design and Testing: i2c\_salve\_deserializer.v (Whitley)
  + Finish Design and Testing: i2si\_bist\_gen.v (Zach)
  + Finish Design and Testing: i2si\_deserializer (Kevin)

**Meeting Notes:**

* We should have a testing specification when we perform testing
  + Multiple smaller unit tests are better than a couple of large comprehensive tests for our blocks
* Dr. Hernandez offered to present his presentation slides on Verilog to the group
  + All of the Verilog modules will have 3 always statements
  + The group will look over the slides and let Dr. Hernandez know if this will be beneficial
  + These slides would take approximately 6 hours in total to go through

**September 30th Senior Project Meeting**

**Armstrong Hall 137: 3:30-????**

**Members in Attendance:** All Members

**Last Week’s Work:**

* Linux Installed
  + Permission errors
  + Need to email passwords
  + Root password: icchip2015
* Zach
  + Bist generator
  + Rts and rtr

**Next Week’s Due Dates:**

* Last Week:
  + Finish Design and Testing of i2si\_deserializer.v (Zach and Kevin)
    - 1 week for the deserializer.v
  + filter\_convolution.v (Dhruvit)
  + Testing of Whitley’s block
    - 1 week for testing
* Friday, September 2nd
  + Create Web Page (Kevin)
  + Finish Design and Testing of filter\_accumulator.v (Dhuvit)
* Wednesday, September 7th
  + Finish Design and Testing of i2so\_serializer.v (Zach and Kevin)
  + Finish EDA Tool Installation (Dhruvit and Julie)
  + Place and Route Test (Dhruvit and Julie)
  + PDR Presentation #2 (All)

**Meeting Notes:**

* Zach will make a test fixture
* Action items
  + Investigate the use of mentor tools for place and route ()
* Asynchronous vs synchronous reset
  + We will use asynchronous
  + Add to rtr protocols

**List of Contacts:**

* Chris Collins - Lab Technician
* Katie Thacker - CORE University Program Coordinator
* Ann Zsilavetz - Computer Science Program Assistant

**Materials List:**

* Software Materials
  + ISE Design Suite 14.7
  + CORE 9 University
  + Git/GitHub Desktop
  + Dropbox
  + Microsoft Project 2013
  + Mentor Graphics
  + Linux RedHat Operating System
* Hardware Materials
  + Nexys 4 Artix-7 FPGA Board
  + UDA 1380 Board
  + Crystal Oscillators
  + CY8CKIT-050 PSoC 5LP Development Kit

**Financial Budget:**

|  |  |  |
| --- | --- | --- |
| **Item** | **Cost** | **Comment** |
| ISE Design Suite 14.7 | $0.00 | Free software |
| CORE 9 University | $0.00 | Vitech’s CORE in the Classroom program |
| Dropbox | $0.00 | Free up to 2GB of storage |
| Git/GitHub Desktop | $0.00 | Free software |
| Microsoft Project 2013 | $0.00 | Free from the Computer Science Department’s DreamSpark Program |
| Mentor Graphics | $0.00 | TCNJ Subscription |
| Linux RedHat Operating System | $0.00 | TCNJ Subscription |
| Nexys 4 Artix-7 FPGA Board | $192.41 | Purchased from Digilent |
| UDA 1380 Board | $20.00 | Needs to be purchased |
| Crystal Oscillators | $20.00 | Needs to be purchased |
| CY8CKIT-050 PSoC 5LP Development Kit | $0.00 | School already owns |
| **Total Budget** | $500.00 |  |
| **Total Costs** | $232.41 |  |
| **End Balance** | $267.59 | Well under-budget |

**Appendix C: Verilog Modules**

1. i2s\_in.v
   1. i2s\_in.v
   2. synchronizer.v
   3. i2si\_deserializer.v
   4. i2si\_bist\_gen.v
   5. i2si\_mux.v
   6. fifo.v
2. i2s\_out.v
   1. i2s\_out.v
   2. serializer.v
3. filter.v
   1. filter\_stm.v
   2. filter\_storage.v
   3. filter\_accumulator.v
   4. filter\_barrel\_shifter.v
   5. filter.v
4. register.v
   1. trig\_generator.v
   2. register.v
   3. initialize\_coeffs.c
   4. set\_coeffs.c

6. i2c\_slave.v

6.1 deserializer.v

6.2 sequencer.v

6.3 serializer.v

**1. i2s\_in.v:**

*1.1 i2s\_in.v:*

module i2s\_in( clk, rst\_n,

inp\_sck, inp\_ws, inp\_sd, rf\_i2si\_en,

rf\_bist\_start\_val, rf\_bist\_inc, rf\_bist\_up\_limit,

rf\_mux\_en,

i2si\_rtr, i2si\_data, i2si\_rts, ro\_fifo\_overrun,

trig\_i2si\_fifo\_overrun\_clr, sync\_sck

);

input clk; //Master clock

input rst\_n; //Reset

input inp\_sck; //Deserializer: Digital audio bit clock

input inp\_ws; //Deserializer: Word select - selects what audio channel is being read. 0 = left channel, 1 = right channel

input inp\_sd; //Deserializer: Digital audio serial data

input rf\_i2si\_en; //Deserializer: Idle when rf\_i2si\_en = 0 and active upon the first high-to-low transition of word select (ws) and rf\_i2si\_en = 1.

input [11:0] rf\_bist\_start\_val; //Bist: start value

input [11:0] rf\_bist\_up\_limit; //Bist: upper limit

input [7:0] rf\_bist\_inc; //Bist: increment signal by this much

input rf\_mux\_en; //Mux: Enabled bit for Bist

output ro\_fifo\_overrun; //FIFO: Input audio FIFO overrun - The FIFO buffer is full and no more can be added to the buffer

input i2si\_rtr; //FIFO: Data input to be pushed to buffer

output [31:0] i2si\_data; //FIFO: Output Data

output i2si\_rts; //FIFO: Output FIFO asserts ready to send

input trig\_i2si\_fifo\_overrun\_clr; //Signal to reset ro\_fifo\_overrun signal

output sync\_sck;

//wire sync\_sck;

wire sck\_transition; //Wire connecting sck\_transition signal to other blocks

wire sync\_ws; //Wire connecting ws synchronizer to ws deserializer

wire sync\_sd; //Wire connecting sd synchronizer to sd deserializer

wire [31:0] deserializer\_data; //Wire connecting lft and rgt deserializer channels to mux. Connects to mux\_in\_0

wire deserializer\_xfc; //Wire output of i2si\_xfc connecting to or gate, i2si\_fifo\_inp\_rtr, and input for BIST'

wire [31:0] bist\_data;

wire bist\_xfc;

wire [31:0] fifo\_data;

wire fifo\_xfc;

wire fifo\_rtr; //Wire connecting fifo\_rtr to not gate

reg ro\_fifo\_overrun;

synchronizer Synchronizer(

.clk (clk),

.rst (rst\_n),

.\_sck (inp\_sck),

.\_ws (inp\_ws),

.\_sd (inp\_sd),

.sck\_transition (sck\_transition),

.sck (sync\_sck), //Remove? need to out sck too and not just sck\_transition?

.ws (sync\_ws),

.sd (sync\_sd)

);

i2si\_deserializer Deserializer(

.clk (clk),

.rst\_n (rst\_n),

.sck\_transition (sck\_transition),

.i2si\_ws (sync\_ws),

.i2si\_sd (sync\_sd),

.rf\_i2si\_en (rf\_i2si\_en),

.i2si\_lft (deserializer\_data [31:16]),

.i2si\_rgt (deserializer\_data [15:0]),

.i2si\_xfc (deserializer\_xfc)

);

i2si\_bist\_gen Bist(

.clk (clk),

.rst (rst\_n),

.sck\_transition (sck\_transition),

.rf\_bist\_start\_val (rf\_bist\_start\_val),

.rf\_bist\_up\_limit (rf\_bist\_up\_limit),

.rf\_bist\_inc (rf\_bist\_inc),

.i2si\_bist\_out\_data (bist\_data),

.i2si\_bist\_out\_xfc (bist\_xfc)

);

i2si\_mux Mux(

.sel (rf\_mux\_en),

.in\_0\_data (deserializer\_data),

.in\_0\_xfc (deserializer\_xfc),

.in\_1\_data (bist\_data),

.in\_1\_xfc (bist\_xfc),

.mux\_data (fifo\_data),

.mux\_xfc (fifo\_xfc)

);

fifo #(3, 8, 32) i2si\_Fifo(

.clk (clk),

.rst (rst\_n),

.fifo\_inp\_data (fifo\_data),

.fifo\_inp\_rts (fifo\_xfc),

.fifo\_inp\_rtr (fifo\_rtr),

.fifo\_out\_data (i2si\_data),

.fifo\_out\_rtr (i2si\_rtr),

.fifo\_out\_rts (i2si\_rts)

);

always @ (posedge clk or negedge rst\_n)

begin

if (!rst\_n)

ro\_fifo\_overrun <= 0;

else if (~fifo\_rtr | deserializer\_xfc)

ro\_fifo\_overrun <= 1;

else if (trig\_i2si\_fifo\_overrun\_clr)

ro\_fifo\_overrun <= 0;

end

endmodule

*1.2 synchronizer.v:*

module synchronizer(clk, rst\_n, \_sck, sck, sck\_transition, \_sd, sd, \_ws, ws

);

input clk; //Master clock

input rst\_n; //Reset

input \_sck; //Non-delayed and non-synchronized sck signal

input \_sd; //Non-delayed and non-synchronized serial data signal

input \_ws; //Non-delayed and non-synchronized word select signal

output sck; //Delayed and Synchronized sck

output sck\_transition; //Signal that represents when sck goes from low to high. Helps define when particular actions should occur

output sd; //Delayed and Synchronized serial data signal

output ws; //Delayed and Synchronized word select signal

reg [2:0] sck\_vec;

reg [3:0] sd\_vec;

reg [3:0] ws\_vec;

wire sck\_delay; //Delayed sck signal that helps define sck\_transition

//Delay sck by 2 clk cycles and synchronize with clk

//sck[1] = sck synchronized with clk

//sck[2] = sck delay signal to help create sck\_transition

always @(posedge clk or negedge rst\_n)

begin

if (!rst\_n)

sck\_vec <= 3'b000;

else

begin

sck\_vec[0] <= \_sck;

sck\_vec[2:1] <= sck\_vec[1:0];

end

end

//Re-assigning sck to be more readable

assign sck = sck\_vec[1];

assign sck\_delay = sck\_vec[2];

//Defines sck\_transition as high when sck transitions from low to high

//helps define when particular actions should occur

assign sck\_transition = sck && !sck\_delay;

//Delay and synchronize sd by 4 clock cycles

//sd[3] is the synchronized signal

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

sd\_vec <= 3'b000;

else if(sck\_transition)

begin

sd\_vec[0] <= \_sd;

sd\_vec[3:1] <= sd\_vec[2:0];

end

end

//Re-assigning sd to be more readable

assign sd = sd\_vec[3];

//Delay and synchronize ws signal by 4 clock cycles

//ws[3] is the synchronized signal

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

ws\_vec <= 4'b0000;

else

begin

ws\_vec[0] <= \_ws;

ws\_vec[3:1] <= ws\_vec[2:0];

end

end

//Re-assigning ws to be more readable

assign ws = ws\_vec[3];

endmodule

*1.3 i2si\_deserializer.v:*

module i2si\_deserializer(clk, rst\_n, sck\_transition, in\_ws, in\_sd, rf\_i2si\_en, out\_lft, out\_rgt, out\_xfc);

input clk; //Master clock

input rst\_n; //Reset

input sck\_transition; //Sck transitions from 0 -> 1. Helps tell the deserializer when to perform certain actions

input in\_ws; //Word select: defines if left or right channel is being read from. 0 = Left Channel, 1 = Right Channel

input in\_sd; //Digital audio serial data

input rf\_i2si\_en; //Enabled bit that helps define if the deserializer is active or idle

output [15:0] out\_lft; //Parallel output data of left channel

output [15:0] out\_rgt; //Parallel output data of right channel

output out\_xfc; //Transfer Complete

reg [15:0] out\_lft;

reg [15:0] out\_rgt;

reg out\_xfc;

reg [1:0] rst\_n\_vec; //Used to check when rst\_n goes from low to high and to trigger armed1

reg armed1; //First signal that helps define idle and active

reg armed2; //Second signal that helps define idle and active

reg active; //Defines if the deserializer is active or not

reg ws\_d; //Delayed signal of in\_ws

reg in\_left; //Defines when the deserializer should read in the left channel

reg in\_left\_delay; //Delayed signal to help define pre\_xfc and out\_xfc

wire ws\_delay; //Delayed signal of ws

wire ws\_transition; //Check if ws goes from 1 -> 0 when en = 1

wire pre\_xfc; //Unsynchronized transfer complete signal

//delay ws signal

//used to help create ws\_transition to define the deserializer as active

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

ws\_d <= 1'b0;

else if(sck\_transition)

ws\_d <= in\_ws;

end

//Re-assigning ws to be more readable

assign ws\_delay = ws\_d;

//ws\_transition becomes high when ws goes from 1 -> 0

//used to help define if deserializer is active

assign ws\_transition = !in\_ws && ws\_delay;

//Used to help define active when rst\_n goes from low to high

always @(posedge clk)

begin

rst\_n\_vec[0] <= rst\_n;

rst\_n\_vec[1] <= rst\_n\_vec[0];

end

//Intermediate step to help define active

//checks if rst\_n goes from high to low

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

armed1 <= 1'b0;

else if(!rst\_n\_vec[1] && rst\_n\_vec[0])

armed1 <= 1'b1;

else if(ws\_transition)

armed1 <= 1'b0;

end

//Intermediate step to help define active

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

armed2 <= 1'b0;

else if(armed1 && ws\_transition)

armed2 <= 1'b1;

else if(sck\_transition)

armed2 <= 1'b0;

end

//Defines when deserializer is idle or active

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

active <= 1'b0;

else if(!rf\_i2si\_en)

active <= 1'b0;

else if(armed2 && sck\_transition)

active <= 1'b1;

end

//Tells deserializer when to start reading in data from left channel

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

in\_left <= 1'b1;

else if (!in\_ws && sck\_transition && active)

in\_left <= 1'b1;

else if (in\_ws && sck\_transition && active)

in\_left <= 1'b0;

end

//Used to help trigger out\_xfc

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

in\_left\_delay <= 1'b0;

else

begin

in\_left\_delay <= in\_left;

end

end

//Triggers out\_xfc when ws[3] goes from high to low

//In other words when the system is done reading in the right channel

//and begins reading in the left channel, trigger xfc.

assign pre\_xfc = in\_left && !in\_left\_delay;

//synchronizing xfc with master clock

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

out\_xfc <= 1'b0;

else

out\_xfc <= pre\_xfc;

end

//Store data into either the left or right channel when

//the deserializer is active and when sck\_transition is high

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

out\_lft[15:0] <= 16'b0;

out\_rgt[15:0] <= 16'b0;

end

else if(active)

begin

if(sck\_transition)

begin

if (in\_left)

begin

out\_lft[15:1] <= out\_lft[14:0];

out\_lft[0] <= in\_sd;

end

else

begin

out\_rgt[15:1] <= out\_rgt[14:0];

out\_rgt[0] <= in\_sd;

end

end

end

end

endmodule

*1.4 i2si\_bist\_gen.v*

// Creates a saw-tooth wave based on the bist register values

module i2si\_bist\_gen(clk,rst\_n,sck\_transition,rf\_bist\_start\_val,rf\_bist\_inc,rf\_bist\_up\_limit,i2si\_bist\_out\_data, i2si\_bist\_out\_xfc);

input clk; //Master Clock

input rst\_n; //Reset

input sck\_transition; //Serial Clock Level to Pulse Converter

input [11:0] rf\_bist\_start\_val; //Start value

input [11:0] rf\_bist\_up\_limit; //Upper limit

input [7:0] rf\_bist\_inc; //Increment signal by this much

output[31:0] i2si\_bist\_out\_data; //Output data

output i2si\_bist\_out\_xfc; //Transfer Complete

reg [31:0] i2si\_bist\_out\_data;

reg i2si\_bist\_out\_xfc;

reg counter=12'b0; //Counter

reg [3:0] sck\_cnt=4'd0;

always@(posedge clk)

begin

if(sck\_transition)

sck\_cnt=sck\_cnt+1;

if(sck\_cnt==4'd15)

begin

//If counter is just starting

if(counter==12'b0)

begin

//Output signal = start value

i2si\_bist\_out\_data<=rf\_bist\_start\_val;

counter<=counter+1'b1;

end

//If signal exceeds the limit

else if(i2si\_bist\_out\_data>=rf\_bist\_up\_limit)

begin

//Signal goes back to start value

i2si\_bist\_out\_data<=rf\_bist\_start\_val;

end

//If the signal is within normal range

else

//Increment the signal

i2si\_bist\_out\_data<=i2si\_bist\_out\_data+rf\_bist\_inc;

sck\_cnt=0;

end

end

endmodule

*1.5 i2si\_mux.v:*

module i2si\_mux(in\_0\_data, in\_0\_xfc, in\_1\_data, in\_1\_xfc, sel, mux\_data, mux\_xfc);

input [31:0] in\_0\_data; //First input data value

input in\_0\_xfc; //First input xfc value

input [31:0] in\_1\_data; //Second input data value

input in\_1\_xfc; //Second input xfc value

input sel; //select input to select either input 0 or 1

output [31:0] mux\_data; //mux data output value

output mux\_xfc; //mux xfc output value

reg [31:0] mux\_data;

reg mux\_xfc;

always @ (sel or in\_0\_data or in\_0\_xfc or in\_1\_data or in\_1\_xfc)

begin

if (sel == 1'b0)

begin

mux\_data <= in\_0\_data;

mux\_xfc <= in\_0\_xfc;

end

else

begin

mux\_data <= in\_1\_data;

mux\_xfc <= in\_1\_xfc;

end

end

endmodule

*1.6 fifo.v:*

module fifo(clk,rst,fifo\_inp\_data,fifo\_out\_data,fifo\_inp\_rts,fifo\_out\_rtr,fifo\_out\_rts,fifo\_inp\_rtr);

parameter BUF\_WIDTH = 3; //Number of bits to be used in pointer

parameter BUF\_SIZE = (1 << BUF\_WIDTH); //Number of elements allowed in buffer = 2^Buffer Width

parameter DATA\_SIZE = 32; //Number of bits for fifo data

input clk; //Master clock

input rst\_n; //Reset

input fifo\_inp\_rts; //Write client asserts ready to send

output fifo\_inp\_rtr; //Write client asserts ready to read

input [DATA\_SIZE - 1:0] fifo\_inp\_data; //Data input to be pushed to buffer

output fifo\_out\_rts; //Output FIFO asserts read to send

input fifo\_out\_rtr;

output [DATA\_SIZE - 1:0] fifo\_out\_data; //Port to output the data using pop.

reg fifo\_inp\_rtr;

reg fifo\_out\_rts;

reg [DATA\_SIZE - 1:0] fifo\_out\_data;

reg [BUF\_WIDTH:0] fifo\_counter;

reg [BUF\_WIDTH - 1:0] rd\_ptr; //Pointer to read

reg [BUF\_WIDTH - 1:0] wr\_ptr; //Write addresses

reg [DATA\_SIZE - 1:0] buf\_mem[BUF\_SIZE - 1:0]; //Buffer memory

always @(fifo\_counter)

begin

//Output FIFO is ready to send if the buffer is not empty

fifo\_out\_rts = (fifo\_counter != 0);

//Output FIFO is ready to receive if the buffer is not full

fifo\_inp\_rtr = (fifo\_counter != BUF\_SIZE);

end

//Update counter based on state of fifo

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

fifo\_counter <= 0;

//if both read and write occur

else if((fifo\_inp\_rtr && fifo\_inp\_rts) && (fifo\_out\_rts && fifo\_out\_rtr))

fifo\_counter <= fifo\_counter;

//write enabled and buffer is not full

else if(fifo\_inp\_rtr && fifo\_inp\_rts)

//counter increased by 1

fifo\_counter <= (fifo\_counter + 1'b1) & (BUF\_SIZE-1'b1);

//read enabled and buffer is not empty

else if(fifo\_out\_rts && fifo\_out\_rtr)

//counter decreased by 1

fifo\_counter <= (fifo\_counter - 1'b1) & (BUF\_SIZE-1'b1);

else

fifo\_counter <= fifo\_counter;

end

//Update output of the fifo

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

fifo\_out\_data <= 0;

else

begin

//If read enabled and buffer is empty

if(fifo\_out\_rtr && fifo\_out\_rts)

//Output is equal to the data in memory at the read pointer

fifo\_out\_data <= buf\_mem[rd\_ptr];

else

fifo\_out\_data <= fifo\_out\_data;

end

end

//Update fifo memory

always @(posedge clk)

begin

//If write is enabled and the buffer is not full

if(fifo\_inp\_rts && fifo\_inp\_rtr)

//Memory at location write pointer is now equal to the input

buf\_mem[wr\_ptr] <= fifo\_inp\_data;

else

buf\_mem[wr\_ptr] <= buf\_mem[wr\_ptr];

end

//Update pointers

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

wr\_ptr <= 0;

rd\_ptr <= 0;

end

else

begin

//If buffer is not full and write is enabled then the write pointer is increased by one

if(fifo\_inp\_rtr && fifo\_inp\_rts)

wr\_ptr <= wr\_ptr + 1;

else

wr\_ptr <= wr\_ptr;

//If buffer is not empty and read is enabled then the read pointer is increased by one

if(fifo\_out\_rts && fifo\_out\_rtr)

rd\_ptr <= rd\_ptr + 1;

else

rd\_ptr <= rd\_ptr;

end

end

endmodule

**2. i2so.v:**

*2.1 i2s\_out.v:*

module i2s\_out( clk, rst\_n, sck\_transition,

filt\_rts, filt\_data, filt\_rtr,

i2so\_ws, i2so\_sd,

ro\_fifo\_overrun, trig\_i2si\_fifo\_overrun\_clr

);

input clk; //Master clock

input rst\_n; //Reset

input sck\_transition; //Signal when sck goes from low to high

output i2so\_ws; //Serializer: Word select - selects what audio channel is being read. 0 = left channel, 1 = right channel

output i2so\_sd; //Serializer: Digital audio serial data

//input rf\_i2si\_en; //Serializer: Idle when rf\_i2si\_en = 0 and active upon the first high-to-low transition of word select (ws) and rf\_i2si\_en = 1.

input filt\_rts; //FIFO: FIFO asserts ready to send

output filt\_rtr; //FIFO: Output FIFO asserts ready to read

input filt\_data; //FIFO: Output Data

input trig\_i2si\_fifo\_overrun\_clr;

output ro\_fifo\_overrun; //FIFO: Input audio FIFO overrun - The FIFO buffer is full and no more can be added to the buffer

//wire sck; //Wire connecting sck synchronizer to sck deserializer NOT NEEDED???

wire sck\_transition; //Wire connecting sck\_transition signal to other blocks

wire fifo\_rts;

wire fifo\_rtr;

wire [31:0] fifo\_data; //serializer input data

reg ro\_fifo\_overrun;

serializer Serializer(

.clk (clk),

.rst\_n (rst\_n),

.sck\_transition (sck\_transition),

// .rf\_i2si\_en (rf\_i2si\_en),

.filt\_i2so\_lft (fifo\_data [31:16]),

.filt\_i2so\_rgt (fifo\_data [15:0]),

.filt\_i2so\_rts (fifo\_rts),

.filt\_i2so\_rtr (fifo\_rtr),

.i2so\_ws (i2so\_ws),

.i2so\_sd (i2so\_sd)

);

fifo #(3, 8, 32) i2so\_Fifo(

.clk (clk),

.rst (rst\_n),

.fifo\_inp\_rts (filt\_rts),

.fifo\_inp\_rtr (filt\_rtr),

.fifo\_inp\_data (filt\_data),

.fifo\_out\_rts (fifo\_rts),

.fifo\_out\_rtr (fifo\_rtr),

.fifo\_out\_data (fifo\_data)

);

always @ (posedge clk or negedge rst\_n)

begin

if (!rst)

ro\_fifo\_overrun <= 0;

else if (~fifo\_rts | fifo\_rtr)

ro\_fifo\_overrun <= 1;

else if (trig\_i2si\_fifo\_overrun\_clr)

ro\_fifo\_overrun <= 0;

end

endmodule

*2.2 serializer.v:*

module serializer(clk, rst\_n, filt\_i2so\_rts, i2so\_sd, i2so\_ws, filt\_i2so\_lft, filt\_i2so\_rgt, filt\_i2so\_rtr, sck\_transition

);

input clk; //Master Clock

input rst\_n; //Reset

input sck\_transition; //Pulse when sck transitions from low to high

input filt\_i2so\_rts; //ready to send

output filt\_i2so\_rtr; //Ready to receive

input [15:0] filt\_i2so\_lft; //Left parallel digital audio data

input [15:0] filt\_i2so\_rgt; //Right parallel digital audio data

output i2so\_sd; //i2s output serial data

output i2so\_ws; //i2s output word Select

reg serializer\_active;

reg i2so\_sd; //i2s output serial data

reg i2so\_ws; //i2s output word select

reg filt\_i2so\_rts\_delay; //Delay signal of ready to send

reg [15:0] lft\_data; //Captures the data of filt\_i2so\_lft

reg [15:0] rgt\_data; //Captures the data of filt\_i2so\_rgt

reg LR; //Left Right Counter: keeps track of which parallel digital audio to read from

reg [3:0] bit\_count; //Bit Counter: keeps track of which bit to read in

wire filt\_i2so\_rtr; //Ready to read

wire filt\_i2so\_rts\_transition; //High when filt\_i2so\_rts goes from low to high

//Helps create filt\_i2so\_rts\_transition signal to define when the serializer is in the active state

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

filt\_i2so\_rts\_delay <= 1'b0;

else

filt\_i2so\_rts\_delay <= filt\_i2so\_rts;

end

assign filt\_i2so\_rts\_transition = filt\_i2so\_rts && !filt\_i2so\_rts\_delay;

//Serializer becomes active when filt\_i2so\_rts\_transitions from low to high

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

serializer\_active <= 0;

else if(filt\_i2so\_rts\_transition)

serializer\_active <= 1'b1;

end

//Tells the serializer to read from filt\_i2so\_lft or filt\_i2so\_rgt

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

LR <= 1'b1;

else if(bit\_count == 0 && sck\_transition && serializer\_active)

LR <= ~LR;

end

assign filt\_i2so\_rtr = sck\_transition && serializer\_active && (bit\_count == 0) && LR;

//Capture data in filt\_i2so\_lft or filt\_i2so\_rgt during first filt\_i2so\_rts\_transition or during LR\_transition

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

lft\_data <= 0;

rgt\_data <= 0;

end

else if(serializer\_active && filt\_i2so\_rtr)

begin

lft\_data <= filt\_i2so\_lft;

rgt\_data <= filt\_i2so\_rgt;

end

end

//Keeps track of which bit of the channel to read from to store in i2so\_sd

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

bit\_count <= 4'd0;

else if(filt\_i2so\_rtr)

bit\_count <= 4'd15;

else if(sck\_transition && serializer\_active)

bit\_count <= bit\_count - 4'd1;

end

//Change ws when channel is on 15th bit or bit [1]

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

i2so\_ws <= 1'b0;

else if (serializer\_active && bit\_count == 4'd1 && sck\_transition)

begin

i2so\_ws <= ~i2so\_ws;

end

end

//Store bit data from filt\_i2so\_lft or filt\_i2so\_rgt into i2so\_sd

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

i2so\_sd <= 1'b0;

else if(serializer\_active)

begin

if(LR == 1'b0)

begin

i2so\_sd <= lft\_data[bit\_count];

end

else

begin

i2so\_sd <= rgt\_data[bit\_count];

end

end

end

endmodule

**3. filter.v:**

*3.1 filter\_stm.v*

`timescale 1ns / 1ps

module filter\_stm( clk, rstb, rts, rtr, do\_transfer, do\_multiply\_1st, do\_multiply, aud\_in, aud\_out, rf\_filter\_coeff

);

input clk; //Clock for State Machine

input rstb; //Active -low reset signal

input rts; //Ready to Send

input [15:0] aud\_in;

input [15:0] rf\_filter\_coeff;

output rtr; //Ready to Recieve

output do\_transfer;

output do\_multiply\_1st; //when 1, data will transfer to memory block

output do\_multiply; //when 1, data is in memory block and is ready to multiply

//output wr\_addr;

//output rd\_addr;

output [15:0] aud\_out;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

localparam IDLE = 4'b0001,

TRANSFER = 4'b0010,

MULTIPLY\_1ST = 4'b0100,

MULTIPLY = 4'b1000;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

localparam IDLE\_ID = 0,

TRANSFER\_ID = 1,

MULTIPLY\_1ST\_ID = 2,

MULTIPLY\_ID = 3;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

localparam PTR = 3,

WIDTH = 16;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reg [3:0] filter\_state, filter\_state\_nxt; //Current State | State for next Clock Cycle

reg do\_transfer, do\_transfer\_nxt; //Current Status | Status for next Clock Cycle

reg do\_multiply\_1st, do\_multiply\_1st\_nxt; //Current Status | Status for next Clock Cycle

reg do\_multiply, do\_multiply\_nxt; //Current Status | Status for next Clock Cycle

reg filter\_running\_1st, filter\_running\_1st\_nxt;

reg filter\_running, filter\_running\_nxt;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reg filter\_need\_new, filter\_need\_new\_nxt;

reg [PTR:0] filter\_count, filter\_count\_nxt;

reg rtr, rtr\_nxt;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reg [PTR:0] wr\_addr\_x, wr\_addr\_x\_nxt;

reg [PTR:0] rd\_addr\_x, rd\_addr\_x\_nxt;

reg arr\_re\_x, arr\_re\_x\_nxt;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reg [PTR:0] wr\_addr\_h, wr\_addr\_h\_nxt;

reg [PTR:0] rd\_addr\_h, rd\_addr\_h\_nxt;

reg arr\_re\_h, arr\_re\_h\_nxt;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

wire filter\_xfc;

wire arr\_we\_x; // Array X Write Enabled

wire arr\_we\_h;

wire [32:0] x\_unit;

wire [32:0] h\_unit;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

assign filter\_xfc = rtr && rts;

assign arr\_we\_x = filter\_xfc;

assign arr\_we\_h = 1'b1;

always@(\*)

begin

filter\_state\_nxt = filter\_state;

wr\_addr\_x\_nxt = wr\_addr\_x;

rd\_addr\_x\_nxt = rd\_addr\_x;

arr\_re\_x\_nxt = arr\_re\_x;

wr\_addr\_h\_nxt = wr\_addr\_h;

rd\_addr\_h\_nxt = rd\_addr\_h;

arr\_re\_h\_nxt = arr\_re\_h;

filter\_running\_1st\_nxt = filter\_running\_1st;

filter\_running\_nxt = filter\_running;

filter\_need\_new\_nxt = filter\_need\_new;

filter\_count\_nxt = filter\_count;

rtr\_nxt = rtr;

do\_transfer\_nxt = 1'b0;

do\_multiply\_1st\_nxt = 1'b0;

do\_multiply\_nxt = 1'b0;

case(1'b1)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// IDLE STATE

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

filter\_state[IDLE\_ID]:begin

if(filter\_xfc) //XFC when ready to recieve and send are equal it will begin the transfer state

begin

filter\_state\_nxt = TRANSFER;

do\_transfer\_nxt = 1'b1;

end

else

begin

//filter\_count\_nxt = 1'b0;

rtr\_nxt = 1'b1; // We can assume input fifo is always ready to recieve data

end

end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// TRANSFER STATE

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

filter\_state[TRANSFER\_ID]:begin

if(filter\_running\_1st) //Condition for state to change to the MULTIPLY STATE

begin

filter\_state\_nxt = MULTIPLY\_1ST;

do\_multiply\_1st\_nxt = 1'b1;

do\_multiply\_nxt = 1'b0;

do\_transfer\_nxt = 1'b0;

filter\_running\_1st\_nxt = 1'b0;

arr\_re\_x\_nxt = 1'b1;

arr\_re\_h\_nxt = 1'b1;

end

else // What occurs during the TRANSFER STATE

begin

do\_transfer\_nxt = 1'b1;

wr\_addr\_x\_nxt = wr\_addr\_x + 1'b1;

wr\_addr\_h\_nxt = wr\_addr\_h + 1'b1;

if(arr\_we\_x)

begin

rtr\_nxt = 1'b0;

filter\_running\_1st\_nxt = 1'b1;

end

end

end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// MULTIPLY STATE 1ST

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

filter\_state[MULTIPLY\_1ST\_ID]:begin

// when it goes through all 512pts

if(filter\_running) //Condition for state to change to the TRANSFER STATE

begin

filter\_state\_nxt = MULTIPLY;

do\_transfer\_nxt = 1'b0;

do\_multiply\_1st\_nxt = 1'b0;

do\_multiply\_nxt = 1'b1;

filter\_need\_new\_nxt = 1'b0;

filter\_running\_nxt = 1'b0;

end

else // What Occurs during the MULTIPLY STATE

begin

rd\_addr\_x\_nxt = rd\_addr\_x + 1'b1;

rd\_addr\_h\_nxt = rd\_addr\_h + 1'b1;

filter\_running\_nxt = 1'b1;

end

end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// MULTIPLY STATE

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

filter\_state[MULTIPLY\_ID]:begin

// when it goes through all 512pts

if(filter\_need\_new) //Condition for state to change to the TRANSFER STATE

begin

filter\_state\_nxt = TRANSFER;

do\_multiply\_nxt = 1'b0;

do\_multiply\_1st\_nxt = 1'b0;

do\_transfer\_nxt = 1'b1;

filter\_need\_new\_nxt = 1'b0;

rtr\_nxt = 1'b1;

end

else // What Occurs during the MULTIPLY STATE

begin

//arr\_x\_re\_nxt = 1'b1;

do\_multiply\_nxt = 1'b1;

rd\_addr\_x\_nxt = rd\_addr\_x + 1'b1;

rd\_addr\_h\_nxt = rd\_addr\_h + 1'b1; //increment + 1 until

filter\_count\_nxt = filter\_count + 1'b1;

if (filter\_count == 1'b1111)

begin

//filter\_count\_nxt = filter\_count - 1'b1;

filter\_need\_new\_nxt = 1'b1;

arr\_re\_x\_nxt = 1'b0;

arr\_re\_h\_nxt = 1'b0;

end

end

end

default: begin end

endcase

end

assign aud\_out = x\_unit + h\_unit ;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always@(posedge clk or negedge rstb)

begin

if(!rstb)

begin

filter\_state <= IDLE;

do\_transfer <= 1'b0;

do\_multiply\_1st <= 1'b0;

do\_multiply <= 1'b0;

wr\_addr\_x <= 1'b0;

rd\_addr\_x <= 1'b0;

wr\_addr\_h <= 1'b0;

rd\_addr\_h <= 1'b0;

filter\_running <= 1'b0;

filter\_running\_1st <= 1'b0;

filter\_need\_new <= 1'b0;

filter\_count <= 1'b0;

rtr <= 1'b0;

end

else

begin

filter\_state <= filter\_state\_nxt;

do\_transfer <= do\_transfer\_nxt;

do\_multiply\_1st <= do\_multiply\_1st\_nxt;

do\_multiply <= do\_multiply\_nxt;

wr\_addr\_x <= wr\_addr\_x\_nxt;

rd\_addr\_x <= rd\_addr\_x\_nxt;

arr\_re\_x <= arr\_re\_x\_nxt;

wr\_addr\_h <= wr\_addr\_h\_nxt;

rd\_addr\_h <= rd\_addr\_h\_nxt;

arr\_re\_h <= arr\_re\_h\_nxt;

filter\_running <= filter\_running\_nxt;

filter\_running\_1st <= filter\_running\_1st\_nxt;

filter\_need\_new <= filter\_need\_new\_nxt;

filter\_count <= filter\_count\_nxt;

rtr <=rtr\_nxt;

end

end

filter\_storage filter\_storage\_x

(.clk (clk),

.wren (arr\_we\_x),

.wrptr (wr\_addr\_x),

.wrdata (aud\_in),

.rden (arr\_re\_x),

.rdptr (rd\_addr\_x),

.rddata (x\_unit));

filter\_storage filter\_storage\_h

(.clk (clk),

.wren (arr\_we\_h),

.wrptr (wr\_addr\_h),

.wrdata (rf\_filter\_coeff),

.rden (arr\_re\_h),

.rdptr (rd\_addr\_h),

.rddata (h\_unit));

endmodule

*3.2 filter\_storage.v*

‘timescale 1ns / 1ps

module filter\_storage

( clk, wren, wrptr, wrdata, rden, rdptr, rddata);

input clk;

input wren;

input [3:0] wrptr;

input [15:0] wrdata;

input rden;

input [3:0] rdptr;

output [15:0] rddata;

localparam DEPTH = 15; //2^9 = 512

localparam WIDTH = 15;

reg [WIDTH:0] ram [DEPTH:0];

reg [WIDTH:0] rddata;

always @(posedge clk)

begin

if(wren)

ram[wrptr] = wrdata;

end

always @(posedge clk)

begin

if(rden)

rddata = ram[rdptr];

end

endmodule

*3.3 filter\_accumulator.v*

`timescale 1ns / 1ps

module filter\_accumulator(clk, rstb, D, Q

);

input clk, rstb;

input [31:0] D;

output [40:0] Q;

reg [40:0] tmp;

assign Q = tmp;

always @(posedge clk or negedge rstb)

begin

if (!rstb)

begin

tmp <= 1'b0;

end

else

begin

tmp <= tmp + D;

end

end

endmodule

*3.4 filter\_barrel\_shifter.v*

`timescale 1ns / 1ps

module filter\_barrel\_shifter(input\_signal, sel\_shift, output\_signal

);

input [31:0] input\_signal;

input [5:0] sel\_shift;

output [31:0] output\_signal;

reg [31:0] output\_signal;

wire [31:0] output\_signal\_1;

wire [31:0] output\_signal\_2;

wire [31:0] output\_signal\_3;

wire [31:0] output\_signal\_4;

wire [31:0] output\_signal\_5;

wire [31:0] output\_signal\_6;

wire [31:0] output\_signal\_7;

wire [31:0] output\_signal\_8;

wire [31:0] output\_signal\_9;

wire [31:0] output\_signal\_10;

wire [31:0] output\_signal\_11;

wire [31:0] output\_signal\_12;

wire [31:0] output\_signal\_13;

wire [31:0] output\_signal\_14;

wire [31:0] output\_signal\_15;

wire [31:0] output\_signal\_16;

wire [31:0] output\_signal\_17;

wire [31:0] output\_signal\_18;

wire [31:0] output\_signal\_19;

wire [31:0] output\_signal\_20;

wire [31:0] output\_signal\_21;

wire [31:0] output\_signal\_22;

wire [31:0] output\_signal\_23;

wire [31:0] output\_signal\_24;

wire [31:0] output\_signal\_25;

wire [31:0] output\_signal\_26;

wire [31:0] output\_signal\_27;

wire [31:0] output\_signal\_28;

wire [31:0] output\_signal\_29;

wire [31:0] output\_signal\_30;

wire [31:0] output\_signal\_31;

assign output\_signal\_1 = {input\_signal[30:0],input\_signal[31]};

assign output\_signal\_2 = {input\_signal[29:0],input\_signal[31:30]};

assign output\_signal\_3 = {input\_signal[28:0],input\_signal[31:29]};

assign output\_signal\_4 = {input\_signal[27:0],input\_signal[31:28]};

assign output\_signal\_5 = {input\_signal[26:0],input\_signal[31:27]};

assign output\_signal\_6 = {input\_signal[25:0],input\_signal[31:26]};

assign output\_signal\_7 = {input\_signal[24:0],input\_signal[31:25]};

assign output\_signal\_8 = {input\_signal[23:0],input\_signal[31:24]};

assign output\_signal\_9 = {input\_signal[22:0],input\_signal[31:23]};

assign output\_signal\_10 = {input\_signal[21:0],input\_signal[31:22]};

assign output\_signal\_11 = {input\_signal[20:0],input\_signal[31:21]};

assign output\_signal\_12 = {input\_signal[19:0],input\_signal[31:20]};

assign output\_signal\_13 = {input\_signal[18:0],input\_signal[31:19]};

assign output\_signal\_14 = {input\_signal[17:0],input\_signal[31:18]};

assign output\_signal\_15 = {input\_signal[16:0],input\_signal[31:17]};

assign output\_signal\_16 = {input\_signal[15:0],input\_signal[31:16]};

assign output\_signal\_17 = {input\_signal[14:0],input\_signal[31:15]};

assign output\_signal\_18 = {input\_signal[13:0],input\_signal[31:14]};

assign output\_signal\_19 = {input\_signal[12:0],input\_signal[31:13]};

assign output\_signal\_20 = {input\_signal[11:0],input\_signal[31:12]};

assign output\_signal\_21 = {input\_signal[10:0],input\_signal[31:11]};

assign output\_signal\_22 = {input\_signal[9:0],input\_signal[31:10]};

assign output\_signal\_23 = {input\_signal[8:0],input\_signal[31:9]};

assign output\_signal\_24 = {input\_signal[7:0],input\_signal[31:8]};

assign output\_signal\_25 = {input\_signal[6:0],input\_signal[31:7]};

assign output\_signal\_26 = {input\_signal[5:0],input\_signal[31:6]};

assign output\_signal\_27 = {input\_signal[4:0],input\_signal[31:5]};

assign output\_signal\_28 = {input\_signal[3:0],input\_signal[31:4]};

assign output\_signal\_29 = {input\_signal[2:0],input\_signal[31:3]};

assign output\_signal\_30 = {input\_signal[1:0],input\_signal[31:2]};

assign output\_signal\_31 = {input\_signal[0],input\_signal[31:1]};

always @\*

begin

output\_signal = input\_signal;

case(sel\_shift)

5'd1: output\_signal = output\_signal\_1;

5'd2: output\_signal = output\_signal\_2;

5'd3: output\_signal = output\_signal\_3;

5'd4: output\_signal = output\_signal\_4;

5'd5: output\_signal = output\_signal\_5;

5'd6: output\_signal = output\_signal\_6;

5'd7: output\_signal = output\_signal\_7;

5'd8: output\_signal = output\_signal\_8;

5'd9: output\_signal = output\_signal\_9;

5'd10: output\_signal = output\_signal\_10;

5'd11: output\_signal = output\_signal\_11;

5'd12: output\_signal = output\_signal\_12;

5'd13: output\_signal = output\_signal\_13;

5'd14: output\_signal = output\_signal\_14;

5'd15: output\_signal = output\_signal\_15;

5'd16: output\_signal = output\_signal\_16;

5'd17: output\_signal = output\_signal\_17;

5'd18: output\_signal = output\_signal\_18;

5'd19: output\_signal = output\_signal\_19;

5'd20: output\_signal = output\_signal\_20;

5'd21: output\_signal = output\_signal\_21;

5'd22: output\_signal = output\_signal\_22;

5'd23: output\_signal = output\_signal\_23;

5'd24: output\_signal = output\_signal\_24;

5'd25: output\_signal = output\_signal\_25;

5'd26: output\_signal = output\_signal\_26;

5'd27: output\_signal = output\_signal\_27;

5'd28: output\_signal = output\_signal\_28;

5'd29: output\_signal = output\_signal\_29;

5'd30: output\_signal = output\_signal\_30;

5'd31: output\_signal = output\_signal\_31;

endcase

end

endmodule

*3.5 filter.v*

`timescale 1ns / 1ps

module filter(clk, rst, aud\_in, aud\_in\_rts, aud\_in\_rtr, aud\_out, aud\_out\_rts, aud\_out\_rtr );

input clk;

input rst;

input [31:0] aud\_in;

//input [15:0] aud\_in\_rgt;

input aud\_in\_rts;

output aud\_in\_rtr;

output [31:0] aud\_out;

//output [15:0] aud\_out\_rgt;

output aud\_out\_rts;

input aud\_out\_rtr;

wire output\_signal;

wire aud\_in\_acc;

wire aud\_out\_acc;

wire aud\_out\_multiply;

wire rstb;

assign rstb = !rst;

assign aud\_out = output\_signal;

assign aud\_in\_acc = aud\_out\_multiply;

filter\_stm filter\_stm\_0

(.clk (clk),

.rstb (rstb),

.rts (aud\_in\_rts),

.aud\_in (aud\_in),

.rtr (aud\_in\_rtr),

.do\_transfer (do\_transfer),

.do\_multiply\_1st (do\_multiply\_1st),

.do\_multiply (do\_multiply),

.wr\_addr (wr\_addr),

.rd\_addr (rd\_addr),

.aud\_out (aud\_out\_multiply));

filter\_accumulator filter\_accumulator\_0

(.clk (clk),

.rstb (rstb),

.D (aud\_in\_acc),

.Q (aud\_out\_acc));

filter\_barrel\_shifter filter\_barrel\_shifter\_0

(.input\_signal (aud\_out\_acc),

.sel\_shift (sel\_shift),

.output\_signal (output\_signal));

endmodule

**4. register.v:**

*4.1 trig\_generator.v*

`timescale 1ns / 1ps

module trig\_generator(

input [11:0] address,

input [7:0] wdata,

input xfc,

input clk,

input rst,

output reg trig\_i2si\_fifo\_overrun\_clr, //address = 0x00C bit 0

output reg trig\_i2so\_fifo\_underrun\_clr // address = 0x00C bit 2

);

always @ (posedge clk or negedge rst)

begin

if (~rst)

begin

trig\_i2si\_fifo\_overrun\_clr <= 0;

trig\_i2so\_fifo\_underrun\_clr <= 0;

end

else

begin

// initializing trigger bits to zero

trig\_i2si\_fifo\_overrun\_clr <= 0;

trig\_i2so\_fifo\_underrun\_clr <= 0;

// triggering when file transfer is complete and address being written to is 0x00c

if (address == 12'h00c && xfc ==1)

begin

// if written to bit 0 of 0x00c, trig\_i2si\_fifo\_overrun\_clr is triggered

if (wdata[0])

trig\_i2si\_fifo\_overrun\_clr <= 1;

// if written to bit 2 of 0x00c, trig\_i2so\_fifo\_underrun\_clr is triggered

if (wdata[2])

trig\_i2so\_fifo\_underrun\_clr <= 1;

end

end

end

endmodule

*4.2 register.v*

`timescale 1ns / 1ps

module register(

input rst,

input clk,

input [11:0] addr,

input [7:0] wdata,

input w\_enable,

input xfc\_write,

input ro\_i2c\_reg\_indir\_data, ro\_fifo\_underrun, ro\_fifo\_overrun,

output reg rf\_soft\_reset, rf\_i2si\_bist\_en, rf\_filter\_shift, rf\_filter\_clip\_en, rf\_i2si\_dec\_factor, rf\_i2so\_dec\_factor,

output reg rf\_i2so\_clk2sck\_div\_a, rf\_i2so\_clk2sck\_div\_b,

output reg trig\_fifo\_overrun, trig\_fifo\_underrun,

output reg rf\_i2si\_bist\_start\_val\_a, rf\_i2si\_bist\_start\_val\_b, rf\_i2si\_bist\_incr, rf\_i2si\_bist\_upper\_limit\_a, rf\_i2si\_bist\_upper\_limit\_b,

output reg rf\_i2c\_reg\_indir\_addr\_a, rf\_i2c\_reg\_indir\_addr\_b,

output reg rf\_filter\_coeff0\_a, rf\_filter\_coeff0\_b,rf\_filter\_coeff1\_a, rf\_filter\_coeff1\_b,rf\_filter\_coeff2\_a, rf\_filter\_coeff2\_b,rf\_filter\_coeff3\_a,

...

rf\_filter\_coeff509\_b,rf\_filter\_coeff510\_a, rf\_filter\_coeff510\_b,rf\_filter\_coeff511\_a, rf\_filter\_coeff511\_b

);

always @(posedge clk or negedge rst)

// initialize all signals and bits

if (~rst)

begin

rf\_soft\_reset <= 12'h000;

rf\_i2si\_bist\_en <= 12'h001;

rf\_filter\_shift <= 12'h00f;

rf\_filter\_clip\_en <= 12'h001;

rf\_i2si\_dec\_factor <= 12'h000;

rf\_i2so\_dec\_factor <= 12'h000;

rf\_i2so\_clk2sck\_div\_a <= 12'h040;

rf\_i2so\_clk2sck\_div\_b <= 12'h040;

trig\_fifo\_overrun <= 12'h000; //NA?

ro\_fifo\_overrun <= 12'h000;

trig\_fifo\_underrun <= 12'h000; //NA?

ro\_fifo\_underrun <= 12'h000;

rf\_i2si\_bist\_incr <= 12'h010;

rf\_i2si\_bist\_start\_val\_a <= 12'h800;

rf\_i2si\_bist\_start\_val\_b <= 12'h800;

rf\_i2si\_bist\_upper\_limit\_a <= 12'h7ff;

rf\_i2si\_bist\_upper\_limit\_b <= 12'h7ff;

rf\_i2c\_reg\_indir\_addr\_a <= 12'h000;

rf\_i2c\_reg\_indir\_addr\_b <= 12'h000;

rf\_filter\_coeff0\_a <= 16'h000;

rf\_filter\_coeff0\_b <= 16'h000;

rf\_filter\_coeff1\_a <= 16'h000;

rf\_filter\_coeff1\_b <= 16'h000;

...

rf\_filter\_coeff510\_a <= 16'h000;

rf\_filter\_coeff510\_b <= 16'h000;

rf\_filter\_coeff511\_a <= 16'h000;

rf\_filter\_coeff511\_b <= 16'h000;

end

else if (xfc\_write && w\_enable)

begin

// Given the address, the signals are assigned to their correlated bits of data

case(addr)

12'h004:

rf\_soft\_rest <= wdata[0];

rf\_i2si\_bist\_en <= wdata[1];

rf\_filter\_shift <= wdata[5:2];

rf\_filter\_clip\_en <= wdata[6];

12'h005:

rf\_i2si\_dec\_factor <= wdata[3:0];

rf\_i2so\_dec\_factor <= wdata[7:4];

12'h008:

rf\_i2so\_clk2sck\_div\_a <= wdata[7:0];

12'h009:

rf\_i2so\_clk2sck\_div\_b <= wdata[7:0];

12'h00c:

trig\_fifo\_overrun <= wdata[0];

ro\_fifo\_overrun <= wdata[1];

trig\_fifo\_underrun <= wdata[2];

ro\_fifo\_underrun <= wdata[3];

12'h010:

rf\_i2si\_bist\_incr <= wdata[7:0];

12'h011:

rf\_i2si\_bist\_start\_val\_a <= wdata[7:0];

12'h012:

rf\_i2si\_bist\_start\_val\_b <= wdata[3:0];

rf\_i2si\_bist\_upper\_limit\_a <= wdata[7:4];

12'h013:

rf\_i2si\_bist\_upper\_limit\_b <= wdata[7:0];

12'h014:

rf\_i2c\_reg\_indir\_addr\_a <= wdata[7:0];

12'h015:

rf\_i2c\_reg\_indir\_addr\_b <= wdata[3:0];

16'h400:

rf\_filter\_coeff0\_a <= wdata[7:0];

16'h401:

rf\_filter\_coeff0\_b <= wdata[7:0];

16'h402:

rf\_filter\_coeff1\_a <= wdata[7:0];

16'h403:

rf\_filter\_coeff1\_b <= wdata[7:0];

...

16'h7fc:

rf\_filter\_coeff510\_a <= wdata[7:0];

16'h7fd:

rf\_filter\_coeff510\_b <= wdata[7:0];

16'h7fe:

rf\_filter\_coeff511\_a <= wdata[7:0];

16'h7ff:

rf\_filter\_coeff511\_b <= wdata[7:0];

endcase

end

end

endmodule

*4.3 Initialize\_coeffs.c*

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int main(void)

{

// Creating and opening file

FILE \*fp;

fp = fopen("filter\_coeffs\_initialized.txt", "w");

if (fp == NULL)

{

perror("Error opening file");

return(-1);

}

/\*

generate 1024 reference strings with incrementing coeff number

and initializing all 512, 2 part coeffs to 0

\*/

int y=-1;

int i;

for (i=0; i<=512; i++)

{

y = y + 1;

fprintf (fp, "\t\trf\_filter\_coeff%d\_a <= 16'h000;\n", y);

fprintf (fp, "\t\trf\_filter\_coeff%d\_b <= 16'h000;\n", y);

}

return 0;

}

*4.4 set\_coeffs.c*

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

// Creating and opening file

FILE \*fp;

fp = fopen("filter\_coeffs\_set.txt", "w");

if (fp == NULL)

{

perror("Error opening file");

return(-1);

}

/\*

Generate 1024 reference strings of 512 two part coeffs with incrementing

hex addresses starting from 0x0404 and ending at 0x07FF.

\*/

int i;

char name[] = "rf\_filter\_coeff";

int coeffNum = -1;

char \* ab;

int abToggle = 0;

char pointer[] = " <= wdata[7:0];";

int address = 0x03FF;

for (i=0; i<1024; i++) {

if (abToggle % 2 == 0) {

ab = "\_a";

coeffNum = coeffNum +1;

} else {

ab = "\_b";

}

address = address + 0x1;

abToggle = abToggle + 1;

fprintf(fp, "\t\t\t\t16'h%x\n\t\t\t\t\t%s%d%s%s\n", address, name, coeffNum, ab, pointer);

}

return 0;

}

**5. i2c\_slave.v:**

*5.1 deserializer.v*

`timescale 1ns / 1ps

module Deserializer(

input Clock,

input Reset,

input i2c\_sda\_raw,

input i2c\_scl\_raw,

input [2:0] i2c\_addr\_bits,

output reg i2c\_RW,

output reg [10:0] i2c\_addr ,

//output reg serial\_data\_xfc,

output reg addr\_ack,

output reg slave\_ack,

output reg data\_ack,

output reg [7:0] serial\_data,

output reg stop\_out

);

//Double and triple rank sda and scl

reg scl\_p1;

reg sda\_p1;

reg sda\_p2;

reg i2c\_scl;

reg i2c\_sda;

always@(posedge Clock)

begin

scl\_p1 <= i2c\_scl\_raw;

i2c\_scl <= scl\_p1;

sda\_p1 <= i2c\_sda\_raw;

sda\_p2 <= sda\_p1;

i2c\_sda <= sda\_p2;

end

//Create SDA and SCL Pulse Signals, and states

wire i2c\_sda\_pos\_pulse;

wire i2c\_sda\_neg\_pulse;

wire i2c\_scl\_pos\_pulse;

wire i2c\_scl\_neg\_pulse;

reg Q\_sda;

reg Q\_scl;

reg sda\_state;

reg scl\_state;

always@(posedge Clock)

begin

Q\_sda = !i2c\_sda;

Q\_scl = !i2c\_scl;

end

assign i2c\_sda\_neg\_pulse = !Q\_sda && !i2c\_sda;

assign i2c\_sda\_pos\_pulse = Q\_sda && i2c\_sda;

assign i2c\_scl\_neg\_pulse = !Q\_scl && !i2c\_scl;

assign i2c\_scl\_pos\_pulse = Q\_scl && i2c\_scl;

always@(posedge i2c\_sda\_pos\_pulse or posedge i2c\_sda\_neg\_pulse /\*or Clock\*/)

begin

if (i2c\_sda\_pos\_pulse)

begin

sda\_state <= 1;

end

else if (i2c\_sda\_neg\_pulse)

begin

sda\_state <= 0;

end

else if(stop)

begin

sda\_state <= 1;

end

end

always@(posedge i2c\_scl\_pos\_pulse or posedge i2c\_scl\_neg\_pulse /\*or Clock\*/)

begin

if (i2c\_scl\_pos\_pulse)

begin

scl\_state <= 1;

end

else if (i2c\_scl\_neg\_pulse)

begin

scl\_state <= 0;

end

else if (stop)

begin

scl\_state <= 1;

end

end

//Deserializer State Initialize

reg [1:0] deserial\_state; //added 9/29/2015 for control of deserializer state, slaveaddress-RW 00, burst\_addr 01, data 10

//initial deserial\_state = 2'b00;

//Deserializer State Update

always@(posedge Clock)

begin

if(slave\_ack)

begin

deserial\_state <= 2'b01;

end

if(addr\_ack && i2c\_RW)

begin

deserial\_state <= 2'b10;

end

if(addr\_ack && !i2c\_RW)

begin

deserial\_state <= 2'b00;

end

if(stop)

begin

deserial\_state <= 2'b00;

end

end

//Set address with offboard bits 1010XXX

wire [6:0] slave\_addr ;

assign slave\_addr = {4'b1010 , i2c\_addr\_bits}; //set slave\_addr = 1010XXX with external pins

//Start and Stop conditions

reg stop;

//initial stop = 1;

//reg stop\_out;

// initial stop\_out = 1;

always@(posedge Clock or negedge Reset)

begin

if (!Reset || slave\_addr\_stop || (i2c\_sda\_pos\_pulse && scl\_state))

begin

stop <= 1;

stop\_out <= 1;

end

else if (stop || stop\_out)

begin

stop\_out <= 0;

end

else if(slave\_addr\_stop || (i2c\_sda\_pos\_pulse && scl\_state))

begin

stop <= 1;

stop\_out <= 1;

end

else if (i2c\_sda\_neg\_pulse && scl\_state)

begin

stop <= 0;

stop\_out <= 0;

end

end

//Deserialize slave address

reg [3:0] bit\_counter\_slave\_addr = 4'b0;

//initial bit\_counter\_slave\_addr = 4'b0;

reg [7:0] incoming\_slave\_addr = 8'b0;

//initial incoming\_slave\_addr = 8'b0;

reg got\_slave\_addr;

//initial got\_slave\_addr = 0;

//reg slave\_addr\_ack;

//initial slave\_addr\_ack = 0;

always@(posedge Clock)

begin

if(stop == 1)

begin

got\_slave\_addr <= 0;

incoming\_slave\_addr <= 8'b0;

bit\_counter\_slave\_addr <= 4'b0;

end

if (deserial\_state == 2'b00 && i2c\_scl\_pos\_pulse && !stop) //when looking for slave address - do

begin

case(bit\_counter\_slave\_addr) //when bit counter = do these, set incoming address bit

4'b0000 : begin

incoming\_slave\_addr [7] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0001;

end

4'b0001 : begin

incoming\_slave\_addr [6] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0010;

end

4'b0010 : begin

incoming\_slave\_addr [5] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0011;

end

4'b0011 : begin

incoming\_slave\_addr [4] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0100;

end

4'b0100 : begin

incoming\_slave\_addr [3] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0101;

end

4'b0101 : begin

incoming\_slave\_addr [2] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0110;

end

4'b0110 : begin

incoming\_slave\_addr [1] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b0111;

end

4'b0111 : begin

incoming\_slave\_addr [0] <= sda\_state;

bit\_counter\_slave\_addr <= 4'b1000;

got\_slave\_addr <= 1;

end

4'b1000 : begin

bit\_counter\_slave\_addr <= 4'b0000;

got\_slave\_addr <= 0;

end

endcase

end

end

//Check Address

reg slave\_addr\_stop;

//initial slave\_addr\_stop = 0;

always@(posedge Clock)

begin

if(stop)

begin

slave\_ack <= 0;

i2c\_RW <= 0;

slave\_addr\_stop <= 0;

end

else if(slave\_ack & i2c\_scl\_neg\_pulse)

begin

slave\_ack <= 0;

end

else if ((deserial\_state == 2'b00) & i2c\_scl\_neg\_pulse & got\_slave\_addr & (incoming\_slave\_addr [7:1] == slave\_addr [6:0]))

begin

slave\_ack <= 1;

i2c\_RW <= incoming\_slave\_addr [0];

end

else if (got\_slave\_addr & (incoming\_slave\_addr [7:1] !== slave\_addr [6:0]))

begin

slave\_addr\_stop <= 1;

end

end

//Deserialize burst address

reg [10:0] burst\_start\_addr ;

//initial burst\_start\_addr = 1'h0;

reg [3:0] bit\_counter\_burst\_addr;

//initial bit\_counter\_burst\_addr = 1'h0; //initialize bit counter to 0

reg got\_addr;

always@(posedge Clock)

begin

if (stop == 1)

begin

bit\_counter\_burst\_addr <= 0;

//addr\_ack <= 0;

burst\_start\_addr <= 0;

i2c\_addr <= 0;

got\_addr <= 1'b0;

end

else if(addr\_ack)

begin

got\_addr <= 0;

end

else if (!slave\_ack && (deserial\_state == 2'b01) && i2c\_scl\_pos\_pulse)

begin

case(bit\_counter\_burst\_addr)

4'b0000 : begin

burst\_start\_addr [10] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0001;

end

4'b0001 : begin

burst\_start\_addr [9] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0010;

end

4'b0010 : begin

burst\_start\_addr [8] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0011;

end

4'b0011 : begin

burst\_start\_addr [7] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0100;

end

4'b0100 : begin

burst\_start\_addr [6] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0101;

end

4'b0101 : begin

burst\_start\_addr [5] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0110;

end

4'b0110 : begin

burst\_start\_addr [4] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b0111;

end

4'b0111 : begin

burst\_start\_addr [3] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b1000;

end

4'b1000 : begin

burst\_start\_addr [2] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b1001;

end

4'b1001 : begin

burst\_start\_addr [1] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b1010;

end

4'b1010 : begin

burst\_start\_addr [0] <= sda\_state;

bit\_counter\_burst\_addr <= 4'b1011;

end

4'b1011 : begin

i2c\_addr [10:0] <= burst\_start\_addr [10:0]; //set busrt start address

bit\_counter\_burst\_addr <= 4'b1100;

got\_addr <= 1'b1;

end

4'b1100 : begin

bit\_counter\_burst\_addr <= 4'b0000;

got\_addr <= 1'b0;

end

endcase

end

end

//Address ack

always@(posedge Clock)

begin

if(stop)

begin

addr\_ack <= 0;

//i2c\_RW <= 0;

//slave\_addr\_stop <= 0;

end

else if(addr\_ack & i2c\_scl\_neg\_pulse)

begin

addr\_ack <= 0;

end

else if ((deserial\_state == 2'b01) & i2c\_scl\_neg\_pulse & got\_addr)

begin

addr\_ack <= 1;

end

end

//Deserialize data

//reg data\_ack = 1'b0;

// initial data\_ack = 0;

reg [3:0] bit\_counter\_data;

//initial bit\_counter\_data = 0;//initialize bit counter to 0

reg got\_data;

always@(posedge Clock)

begin

if(stop)

begin

serial\_data <= 0;

bit\_counter\_data <= 0;

//data\_ack <= 0;

got\_data <= 0;

end

else if(data\_ack)

begin

got\_data <= 0;

bit\_counter\_data <= 4'b0000;

serial\_data <= 8'b00000000;

end

else if (!addr\_ack & !got\_data & i2c\_scl\_pos\_pulse && (deserial\_state == 2'b10) & (i2c\_RW == 1))

begin

case (bit\_counter\_data)

4'b0000 : begin

serial\_data [7] <= sda\_state;

bit\_counter\_data <= 4'b0001;

end

4'b0001 : begin

serial\_data [6] <= sda\_state;

bit\_counter\_data <= 4'b0010;

end

4'b0010 : begin

serial\_data [5] <= sda\_state;

bit\_counter\_data <= 4'b0011;

end

4'b0011 : begin

serial\_data [4] <= sda\_state;

bit\_counter\_data <= 4'b0100;

end

4'b0100 : begin

serial\_data [3] <= sda\_state;

bit\_counter\_data <= 4'b0101;

end

4'b0101 : begin

serial\_data [2] <= sda\_state;

bit\_counter\_data <= 4'b0110;

end

4'b0110 : begin

serial\_data [1] <= sda\_state;

bit\_counter\_data <= 4'b0111;

end

4'b0111 : begin

serial\_data [0] <= sda\_state;

got\_data <= 1'b1;

end

endcase

end

end

always@(posedge Clock)

begin

if(stop)

begin

data\_ack <= 0;

end

else if(data\_ack & i2c\_scl\_neg\_pulse)

begin

data\_ack <= 0;

end

else if ((deserial\_state == 2'b10) & i2c\_scl\_neg\_pulse & got\_data)

begin

data\_ack <= 1;

end

end

endmodule

*5.2 sequencer.v*

`timescale 1ns / 1ps

module Sequencer(

input Clock,

input i2c\_RW,

output reg i2c\_op,

input [10:0] i2c\_addr\_in,

output reg [10:0] i2c\_addr\_out,

input [7:0] i2c\_data\_in,

output reg [7:0] i2c\_data\_out,

input i2c\_addr\_ack,

input i2c\_data\_ack,

output reg i2c\_xfc,

input reset,

input stop

);

//Create address incremental value register

reg [10:0] addr\_increment;

initial addr\_increment = 0;

reg xfc\_ready;

initial xfc\_ready=0;

//single cycle address acknowledge

wire addr\_ack\_temp;

wire data\_ack\_temp;

reg Q\_addr;

reg Q\_data;

always@(posedge Clock)

begin

Q\_addr <= !i2c\_addr\_ack;

Q\_data <= !i2c\_data\_ack;

end

assign addr\_ack\_temp = Q\_addr && i2c\_addr\_ack;

assign data\_ack\_temp = Q\_data && i2c\_data\_ack;

/\*

always@(posedge Clock)

begin

if(!reset | stop)

begin

addr\_ack\_temp = 0;

data\_ack\_temp = 0;

end

if(

\*/

//always block to perform address and data strobe

//wire ack\_not\_RW = addr\_ack\_temp & !i2c\_RW;

reg stop\_read;

reg [10:0] i2c\_addr\_write;

always@(posedge Clock or negedge reset)

begin

if(stop | !reset | stop\_read)

begin

i2c\_op <= 0;

i2c\_addr\_out <= 0;

i2c\_data\_out <= 0;

i2c\_xfc <= 0;

addr\_increment <= 0;

stop\_read <= 0;

i2c\_addr\_write <= 0;

end

//Read Request Sequence

else if(addr\_ack\_temp & !i2c\_RW) //read request

begin

i2c\_addr\_out <= i2c\_addr\_in;

i2c\_op <= 0;

xfc\_ready <= 1;

end

else if(xfc\_ready & !i2c\_RW) //xfc high on address ready, READ

begin

i2c\_xfc <= 1;

xfc\_ready <= 0;

end

else if(i2c\_xfc & !i2c\_RW) //zero xfc after 1 clock cycle for read, READ

begin

i2c\_xfc <= 0;

stop\_read <= 1;

end

//Write Request Sequence

else if(addr\_ack\_temp & i2c\_RW) //write request store address

begin

i2c\_op <= 1;

i2c\_addr\_write <= i2c\_addr\_in;

xfc\_ready <= 1;

end

else if(data\_ack\_temp & i2c\_RW) //write send

begin

i2c\_data\_out <= i2c\_data\_in;

i2c\_addr\_out <= i2c\_addr\_write + addr\_increment;

xfc\_ready <= 1;

end

else if(xfc\_ready & i2c\_RW) //xfc high on data ready

begin

i2c\_xfc <= 1;

xfc\_ready <= 0;

end

else if(i2c\_xfc & i2c\_RW) //zero xfc after 1 clock cycle for write

begin

i2c\_xfc <= 0;

addr\_increment <= addr\_increment + 1;

i2c\_data\_out <= 0;

i2c\_addr\_out <= 0;

end

end

endmodule

*5.3 serializer.v*

`timescale 1ns / 1ps

module Serializer(

input i2c\_scl,

input i2c\_sda,

output reg i2c\_sda\_out,

// input i2c\_sda\_in,

input data\_ack,

input slave\_ack,

input addr\_ack,

input Clock,

input reset,

input [7:0] i2c\_rdata,

input i2c\_xfc\_read,

input stop\_in

);

//Create SCL Pulse Signals, and states

wire i2c\_sda\_pos\_pulse;

wire i2c\_sda\_neg\_pulse;

wire i2c\_scl\_pos\_pulse;

wire i2c\_scl\_neg\_pulse;

reg Q\_sda;

reg Q\_scl;

reg sda\_state;

reg scl\_state;

always@(posedge Clock)

begin

Q\_sda = !i2c\_sda;

Q\_scl = !i2c\_scl;

end

assign i2c\_sda\_neg\_pulse = !Q\_sda && !i2c\_sda;

assign i2c\_sda\_pos\_pulse = !Q\_sda && i2c\_sda;

assign i2c\_scl\_neg\_pulse = !Q\_scl && !i2c\_sda;

assign i2c\_scl\_pos\_pulse = !Q\_scl && i2c\_sda;

always@(posedge i2c\_sda\_pos\_pulse or posedge i2c\_sda\_neg\_pulse)

begin

if (i2c\_sda\_pos\_pulse)

begin

sda\_state <= 1;

end

else if (i2c\_sda\_neg\_pulse)

begin

sda\_state <= 0;

end

end

always@(posedge i2c\_scl\_pos\_pulse or posedge i2c\_scl\_neg\_pulse)

begin

if (i2c\_scl\_pos\_pulse)

begin

scl\_state <= 1;

end

else if (i2c\_scl\_neg\_pulse)

begin

scl\_state <= 0;

end

end

//Stop Conditions

reg stop;

//initial stop = 0;

always@(posedge Clock or negedge reset)

begin

if (!reset | (!scl\_state & i2c\_sda\_pos\_pulse) | serialize\_done | stop\_in)

begin

stop <= 1;

end

else if (i2c\_xfc\_read)

begin

stop <= 0;

end

end

//Take in transfer Data

reg [7:0] data\_read;

//initial data\_read = 0;

reg serialize\_ready;

//initial serialize\_ready = 0;

always@(posedge Clock)

begin

if(stop)

begin

data\_read = 0;

serialize\_ready = 0;

end

else if(i2c\_xfc\_read)

begin

data\_read = i2c\_rdata;

serialize\_ready = 1;

end

end

//Serialize State

reg serialize\_ok;

//initial serialize\_ok = 0;

always @ (posedge Clock)

begin

if(stop)

begin

serialize\_ok = 0;

end

else if(serialize\_ready & !serialize\_done)

begin

serialize\_ok = 1;

end

end

//Serialize data

reg [4:0] serialize\_bit\_counter;

//initial serialize\_bit\_counter = 0;

reg serialize\_done;

//initial serialize\_done = 0;

always@(posedge Clock)

begin

if(!serialize\_ok | stop)

begin

serialize\_bit\_counter = 0;

serialize\_done = 0;

end

else if(data\_ack | slave\_ack | addr\_ack | !stop)

begin

i2c\_sda\_out = 1;

serialize\_done =1;

end

else if(serialize\_ok & i2c\_scl\_neg\_pulse)

begin

case(serialize\_bit\_counter)

4'b0000 : begin

i2c\_sda\_out = data\_read[7];

serialize\_bit\_counter = 4'b0001;

end

4'b0001 : begin

i2c\_sda\_out = data\_read[6];

serialize\_bit\_counter = 4'b0010;

end

4'b0010 : begin

i2c\_sda\_out = data\_read[5];

serialize\_bit\_counter = 4'b0011;

end

4'b0011 : begin

i2c\_sda\_out = data\_read[4];

serialize\_bit\_counter = 4'b0100;

end

4'b0100 : begin

i2c\_sda\_out = data\_read[3];

serialize\_bit\_counter = 4'b0101;

end

4'b0101 : begin

i2c\_sda\_out = data\_read[2];

serialize\_bit\_counter = 4'b0110;

end

4'b0110 : begin

i2c\_sda\_out = data\_read[1];

serialize\_bit\_counter = 4'b0111;

end

4'b1000 : begin

i2c\_sda\_out = data\_read[0];

serialize\_bit\_counter = 4'b0000;

serialize\_done = 1;

end

endcase

end

end

endmodule

**Appendix D: Test Benches**

1. i2s\_in.v
   1. i2si\_deserializer\_testbench.v
   2. bist\_test.v
   3. fifo\_test.v
2. i2s\_out.v
   1. serializer\_testbench.v
3. register.v
   1. trig\_generator\_testbench.v
   2. trig\_generator\_testbench1.v
4. i2c\_slave.v
   1. deserial\_sequencer\_test.v
   2. deserial\_test2.v
   3. sequencertest2.v

**1. i2si.v:**

*1.1 i2si\_deserializer\_testbench.v*

module i2si\_deserializer\_testbench;

// Inputs

reg clk; // master clock

reg i2si\_sck; // serial clock

reg rf\_i2si\_en; // i2si enable

// Outputs

wire [15:0] i2si\_lft; // left audio dataF

wire [15:0] i2si\_rgt; // right audio data

wire i2si\_xfc; // transfer complete

wire rst\_n; // reset not

wire i2si\_sd; // serial data

wire i2si\_ws; // word select

// Internal Variables

reg [15:0] test\_data [11-1:0] [0:1]; // [Bits Per Word] test\_data [# of entities in test] [Left/Right]

reg sck\_d1; // serial clock delay

reg [31:0] count; // clock counter

reg [31:0] sck\_cnt; // serial clock counter

reg [31:0] bit\_cnt; // bit number counter

reg lr\_cnt; // left right counter

reg [31:0] word\_cnt; // word counter

reg [31:0] cyc\_per\_half\_sck = 40; // about (100 MHz / 1.44 MHz)/2

reg [31:0] bit\_tc = 15; // number of bits in a word

// Instantiate the Unit Under Test (UUT)

i2si\_deserializer uut (

.clk(clk),

.rst\_n(rst\_n),

.i2si\_sck(i2si\_sck),

.i2si\_ws(i2si\_ws),

.i2si\_sd(i2si\_sd),

.rf\_i2si\_en(rf\_i2si\_en),

.i2si\_lft(i2si\_lft),

.i2si\_rgt(i2si\_rgt),

.i2si\_xfc(i2si\_xfc)

);

initial begin

// Initialize Inputs

clk = 0;

i2si\_sck = 0;

rf\_i2si\_en = 0;

// Test Data

test\_data [0] [0] = 16'hAAAA;

test\_data [0] [1] = 16'hFFFF;

test\_data [1] [0] = 16'h1478;

test\_data [1] [1] = 16'hA3B9;

test\_data [2] [0] = 16'hCDD7;

test\_data [2] [1] = 16'hBABA;

test\_data [3] [0] = 16'h4444;

test\_data [3] [1] = 16'hAAAA;

test\_data [4] [0] = 16'h7398;

test\_data [4] [1] = 16'hFFDD;

test\_data [5] [0] = 16'h1111;

test\_data [5] [1] = 16'h5982;

test\_data [6] [0] = 16'h0001;

test\_data [6] [1] = 16'hFFFF;

test\_data [7] [0] = 16'h1478;

test\_data [7] [1] = 16'hA3B9;

test\_data [8] [0] = 16'hF8D5;

test\_data [8] [1] = 16'hD55A;

test\_data [9] [0] = 16'h99C5;

test\_data [9] [1] = 16'h7435;

test\_data [10] [0] = 16'h69D9;

test\_data [10] [1] = 16'hABCD;

#694

rf\_i2si\_en = 1; // enable i2si after 694ns

end

always

begin

count = 0; // set clock counter to zero

forever

begin

#5 clk = ~clk; // 100 MHz clock

count = count + 1; // increment clock counter

end

end

assign rst\_n = !(count < 20); // turn on reset not after 10 clock cycles

assign rst = ~rst\_n; // reset is the opposite of reset not

assign i2si\_ws = ((0<=bit\_cnt& bit\_cnt<=16'd14)&lr\_cnt==1)|((bit\_cnt==16'd15)&(lr\_cnt==0));

assign i2si\_sd = test\_data [word\_cnt][lr\_cnt][bit\_tc-bit\_cnt]; // assign serial data from the test\_data

always @ (posedge clk or negedge rst)

begin

if(rst!=0)

begin

sck\_cnt<=0; // counts master clock cycles, causes sck to toggle each time it hits cyc\_per\_half\_sck

bit\_cnt<=0; // count number of bits

word\_cnt<=0; // count the word number

lr\_cnt <= 0; // left=0 and right=1

i2si\_sck<=0; // serial clock

sck\_d1<=0; // serial clock delayed by one clock cycle

end

else

begin

if (sck\_cnt == cyc\_per\_half\_sck-1) // cyc\_per\_half\_sck ~ (100 MHz/1.44 MHz)/2

begin

sck\_cnt <= 0; // reset serial clock counter

i2si\_sck <= ~i2si\_sck; // toggle serial clock

end

else

sck\_cnt <= sck\_cnt + 1; // increment serial clock counter

sck\_d1<=i2si\_sck; // generate 1 cycle delay of i2si\_sck

if(i2si\_sck & ~sck\_d1) // on a positive transition of sck...

begin

if (bit\_cnt==bit\_tc) // bit\_tc = 15

begin

if (lr\_cnt == 1) // if right

begin

word\_cnt<=word\_cnt+1; // words in the testbench array

lr\_cnt<=0; // set to left

end

else

lr\_cnt<=1; // set to right

bit\_cnt<=0; // reset bit counter

end

else

bit\_cnt<=bit\_cnt+1; // increment bit counter

end

end

end

endmodule

*1.2 bist\_test.v*

module bist\_test;

// Inputs

reg clk;

wire rst\_n;

reg sck\_transition;

reg [11:0] rf\_bist\_start\_val;

reg [7:0] rf\_bist\_inc;

reg [11:0] rf\_bist\_up\_limit;

// Internal Variables

reg i2si\_sck;

reg sck\_d1; // serial clock delay

reg [31:0] count;

reg [31:0] sck\_cnt; // serial clock counter

reg [31:0] cyc\_per\_half\_sck = 40; // about (100 MHz / 1.44 MHz)/2

// Outputs

wire [31:0] i2si\_bist\_out\_data;

// Instantiate the Unit Under Test (UUT)

i2si\_bist\_gen uut (

.clk(clk),

.rst\_n(rst\_n),

.sck\_transition(sck\_transition),

.rf\_bist\_start\_val(rf\_bist\_start\_val),

.rf\_bist\_inc(rf\_bist\_inc),

.rf\_bist\_up\_limit(rf\_bist\_up\_limit),

.i2si\_bist\_out\_data(i2si\_bist\_out\_data)

);

initial begin

// Initialize Inputs

clk = 0;

i2si\_sck=0;

rf\_bist\_start\_val = 12'h001;

rf\_bist\_inc = 12'h001;

rf\_bist\_up\_limit = 12'h019;

end

assign rst\_n = !(count < 20); // turn on reset not after 10 clock cycles

always

begin

count = 0; // set clock counter to zero

forever

begin

#5 clk = ~clk; // 100 MHz clock

count = count + 1; // increment clock counter

end

end

always @(\*)

begin

sck\_transition <= i2si\_sck & ~sck\_d1;

end

always @ (posedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

sck\_cnt<=0; // counts master clock cycles, causes sck to toggle each time it hits cyc\_per\_half\_sck

i2si\_sck<=0; // serial clock

sck\_d1<=0; // serial clock delayed by one clock cycle

end

else

sck\_cnt <= sck\_cnt + 1; // increment serial clock counter

sck\_d1<=i2si\_sck; // generate 1 cycle delay of i2si\_sck

begin

if (sck\_cnt == cyc\_per\_half\_sck-1) // cyc\_per\_half\_sck ~ (100 MHz/1.44 MHz)/2

begin

sck\_cnt <= 0; // reset serial clock counter

i2si\_sck <= ~i2si\_sck; // toggle serial clock

end

end

end

endmodule

*1.3 fifo\_test.v*

`define BUF\_WIDTH 3 // set the buffer width equal to 3

`define DATA\_SIZE 32 // no. of bits for fifo data

module fifo\_test();

reg clk, rst\_n, fifo\_inp\_rts, fifo\_out\_rtr ; // clock, reset, write enabled, read enabled

reg[`DATA\_SIZE-1:0] fifo\_inp\_data; // buffer input

reg[`DATA\_SIZE-1:0] tempdata; // temporary data

wire [`DATA\_SIZE-1:0] fifo\_out\_data; // buffer output

fifo ff( .clk(clk), .rst\_n(rst\_n), .fifo\_inp\_data(fifo\_inp\_data), .fifo\_out\_data(fifo\_out\_data),

.fifo\_inp\_rts(fifo\_inp\_rts), .fifo\_out\_rtr(fifo\_out\_rtr), .fifo\_out\_rts(fifo\_out\_rts),

.fifo\_inp\_rtr(fifo\_inp\_rtr));

initial

begin

clk = 0; // clock starts at false

rst\_n = 0; // reset starts at true

fifo\_out\_rtr = 0; // read enabled starts at false

fifo\_inp\_rts= 0; // write enabled starts at false

tempdata = 0; // temporary data starts at zero

fifo\_inp\_data = 0; // buffer input is intially zero

#15 rst\_n = 1; // after 15ns change the reset to false

push(1); // push the value of 1

fork // push the value of 2 and pop at the same time

push(2);

pop(tempdata);

join

push(10); // push the value of 10

push(20); // push the value of 20

push(30); // push the value of 30

push(40); // push the value of 40

push(50); // push the value of 50

push(60); // push the value of 60

push(70); // push the value of 70

push(80); // push the value of 80

push(90); // push the value of 90

push(100); // push the value of 100

push(110); // push the value of 110

push(120); // push the value of 120

push(130); // push the value of 130

pop(tempdata); // pop

push(tempdata); // push the data that was just popped

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

push(140); // push the value of 140

pop(tempdata); // pop

push(tempdata); // push the data that was just popped

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

pop(tempdata); // pop

push(5); // push the value of 5

pop(tempdata); // pop

end

always

#5 clk = ~clk; // change the clock every 5ns

task push; // define the push task

input[7:0] data; // the data to be pushed

if( !fifo\_inp\_rtr ) // if buffer is full display warning

$display("---Cannot push: Buffer Full---");

else // if buffer is not full

begin

$display("Pushed ",data ); // display that the data was pushed

fifo\_inp\_data = data; // the input to the buffer is set as the data

fifo\_inp\_rts = 1; // write is enabled

@(posedge clk); // checks if clock is at postive edge

#1 fifo\_inp\_rts = 0; // set write enabled equal to zero then

end

endtask

task pop; // define the pop task

output [7:0] data; // the data to be popped

if( !fifo\_out\_rts ) // if the buffer is empty display a warning

$display("---Cannot Pop: Buffer Empty---");

else // if buffer is not empty

begin

fifo\_out\_rtr = 1; // read is enabled

@(posedge clk); // checks if clock is at postive edge

#1 fifo\_out\_rtr = 0; // set read enabled qual to zero then

data = fifo\_out\_data; // the data is set as the output of the buffer

$display("-------------------------------Poped ", data); // display that the data was pushed

end

endtask

endmodule

**2. i2so.v:**

*2.1 serializer\_testbench.v*

module serializer\_testbench;

// Inputs

reg clk;

reg rst\_n;

reg filt\_i2so\_rts;

reg [15:0] filt\_i2so\_lft;

reg [15:0] filt\_i2so\_rgt;

reg sck\_transition;

// Internal Variables

reg i2si\_sck;

reg [2:0] sck\_vec;

reg sck;

reg sck\_delay;

reg [31:0] count;

reg [31:0] word\_count; // word counter

reg [15:0] test\_data [0:9] [0:1]; // [Bits Per Word] test\_data [# of entities in test] [Left/Right]

// Outputs

wire i2so\_sd;

wire i2so\_ws;

wire filt\_i2so\_rtr;

// Instantiate the Unit Under Test (UUT)

serializer uut (

.clk(clk),

.rst\_n(rst\_n),

.filt\_i2so\_rts(filt\_i2so\_rts),

.i2so\_sd(i2so\_sd),

.i2so\_ws(i2so\_ws),

.filt\_i2so\_lft(filt\_i2so\_lft),

.filt\_i2so\_rgt(filt\_i2so\_rgt),

.filt\_i2so\_rtr(filt\_i2so\_rtr),

.sck\_transition(sck\_transition)

);

always

begin

count = 0;

forever

begin

#5 clk = ~clk;

count = count + 1; // increment clock counter

end

end

always

begin

forever

begin

#312.5 i2si\_sck = ~i2si\_sck;

end

end

always @(posedge clk or negedge rst\_n)

begin

if (!rst\_n)

sck\_vec <= 3'b000;

else

begin

sck\_vec[0] <= i2si\_sck;

sck\_vec[2:1] <= sck\_vec[1:0];

end

end

always @(\*)

begin

rst\_n = !(count < 21); // turn on reset not after 10 clock cycles

sck <= sck\_vec[1];

sck\_delay <= sck\_vec[2];

sck\_transition <= sck && !sck\_delay;

if(word\_count >= 0 && word\_count <= 9)

begin

filt\_i2so\_lft = test\_data [word\_count][0];

filt\_i2so\_rgt = test\_data [word\_count][1];

end

end

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

word\_count <= -1;

end

else if(filt\_i2so\_rtr)

begin

word\_count <= word\_count + 1;

end

end

initial begin

// Initialize Inputs

clk = 0;

rst\_n = 0;

filt\_i2so\_rts = 0;

filt\_i2so\_lft = 0;

filt\_i2so\_rgt = 0;

sck\_transition = 0;

i2si\_sck = 0;

// Test Data

test\_data [0] [0] = 16'h0000;

test\_data [0] [1] = 16'hFFFF;

test\_data [1] [0] = 16'hFF00;

test\_data [1] [1] = 16'h00FF;

test\_data [2] [0] = 16'hAAAA;

test\_data [2] [1] = 16'h5555;

test\_data [3] [0] = 16'hBABA;

test\_data [3] [1] = 16'h4444;

test\_data [4] [0] = 16'h7398;

test\_data [4] [1] = 16'hFFDD;

test\_data [5] [0] = 16'h1111;

test\_data [5] [1] = 16'h5982;

test\_data [6] [0] = 16'h0001;

test\_data [6] [1] = 16'hFFFF;

test\_data [7] [0] = 16'h1478;

test\_data [7] [1] = 16'hA3B9;

test\_data [8] [0] = 16'hF8D5;

test\_data [8] [1] = 16'hD55A;

test\_data [9] [0] = 16'h99C5;

test\_data [9] [1] = 16'h7435;

#694

filt\_i2so\_rts = 1;

end

endmodule

**3. register.v:**

*3.1 trig\_generator\_testbench.v*

`timescale 1ns / 1ps

module trig\_generator\_testbench;

// Inputs

reg [11:0] address;

reg [7:0] wdata;

reg xfc;

reg clk;

reg [31:0] count;

// Outputs

wire trig\_i2si\_fifo\_overrun\_clr;

wire trig\_i2so\_fifo\_underrun\_clr;

wire rst\_n;

// Instantiate the Unit Under Test (UUT)

trig\_generator uut (

.address(address),

.wdata(wdata),

.xfc(xfc),

.clk(clk),

.rst(rst),

.trig\_i2si\_fifo\_overrun\_clr(trig\_i2si\_fifo\_overrun\_clr),

.trig\_i2so\_fifo\_underrun\_clr(trig\_i2so\_fifo\_underrun\_clr)

);

always

begin

begin

count = 0;

clk = 0;

end

forever

begin

#5 clk = ~clk;

count = count + 1;

end

end

assign rst\_n = !(count < 20);

initial begin

// Initialize Inputs

wdata = 8'hFF;

// Wait 100 ns for global reset to finish

#1000;

end

always @(posedge clk or negedge rst\_n)

begin

if (~rst\_n)

begin

address <= 0;

xfc <= 0;

end

else if (address < 12'h20) //hex 20 12 bits of data

begin

address <= address + 4;

xfc <= 1;

end

*3.2 trig\_generator\_testbench1.v*

`timescale 1ns / 1ps

module trig\_generator\_testbench1;

// Inputs

reg [11:0] address;

reg [7:0] wdata;

reg xfc;

reg clk;

reg [31:0] count;

// Outputs

wire trig\_i2si\_fifo\_overrun\_clr;

wire trig\_i2so\_fifo\_underrun\_clr;

wire rst\_n;

// Instantiate the Unit Under Test (UUT)

trig\_generator uut (

.address(address),

.wdata(wdata),

.xfc(xfc),

.clk(clk),

.rst(rst),

.trig\_i2si\_fifo\_overrun\_clr(trig\_i2si\_fifo\_overrun\_clr),

.trig\_i2so\_fifo\_underrun\_clr(trig\_i2so\_fifo\_underrun\_clr)

);

always

begin

begin

count = 0;

clk = 0;

end

// Generates a clock with a clock cycle of 10 ns

forever

begin

#5 clk = ~clk;

count = count + 1;

end

end

assign rst\_n = !(count < 20);

initial begin

// Initialize Inputs

address = 12'h00c;

// Wait 100 ns for global reset to finish

#1000;

end

always @(posedge clk or negedge rst\_n)

begin

// initializing xfc and wdata to 0

if (~rst\_n)

begin

wdata <= 0;

xfc <= 0;

end

/\*

if wdata is 12 bits of data and less than the hex value 20 wdata is

and file transfer is set to 1 - complete

\*/

else if (wdata < 12'h020)

begin

wdata <= wdata + 1;

xfc <= 1;

end

else

xfc <=0;

end

endmodule

**4. i2c\_slave .v:**

*4.1 deserial\_sequencer\_test.v*

// Verilog test fixture created from schematic C:\Users\formanw2\Downloads\i2c\_slave 11\_4\_15 (2)\i2c\_slave 11\_4\_15\i2c\_Top\_Blcok\i2c\_Top\_Block.sch - Tue Nov 10 17:31:33 2015

`timescale 1ns / 1ps

module i2c\_Top\_Block\_i2c\_Top\_Block\_sch\_tb();

// Inputs

reg i2c\_SDA;

reg i2c\_SCL;

reg [2:0] i2c\_addr\_bits;

reg Clock;

reg Reset;

// Output

wire i2c\_op;

wire i2c\_xfc;

wire [10:0] i2c\_addr\_out;

wire [7:0] i2c\_data\_out;

wire slave\_ack;

// Bidirs

// Instantiate the UUT

i2c\_Top\_Block UUT (

.i2c\_SDA(i2c\_SDA),

.i2c\_SCL(i2c\_SCL),

.i2c\_addr\_bits(i2c\_addr\_bits),

.i2c\_op(i2c\_op),

.i2c\_xfc(i2c\_xfc),

.i2c\_addr\_out(i2c\_addr\_out),

.i2c\_data\_out(i2c\_data\_out),

.slave\_ack(slave\_ack),

.Clock(Clock),

.Reset(Reset)

);

// Initialize Inputs

//`ifdef auto\_init

initial begin

i2c\_SDA = 0;

i2c\_SCL = 0;

i2c\_addr\_bits = 0;

Clock = 0;

Reset = 0;

//`endif

// Wait 100 ns for global Reset to finish

#100;

end

always #50 Clock = !Clock;

always #1250 i2c\_SCL = !i2c\_SCL;

//always #200 i2c\_SDA = !i2c\_SDA;

initial begin

i2c\_SCL = 1;

i2c\_SDA = 1;

i2c\_addr\_bits = 3'b101;

Reset = 1;

#10

Reset = 0;

#10

Reset = 1;

#3225

i2c\_SDA = 0; //Addr Start

#1500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1; //RW Bit

#2500

i2c\_SDA = 1; //nothing

#2500

i2c\_SDA = 0; //Daat addr MSB

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0; //Data Addr LSB

#2500

i2c\_SDA = 1; //nothing

#2500

i2c\_SDA = 1; //Data MSB

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1; //Data LSB

#2500

i2c\_SDA = 0; //nothing

#2500

i2c\_SDA = 0; //Data MSB

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0; //Data LSB

#2500

i2c\_SDA = 1; //nothing

#2500

i2c\_SDA = 1; //Data MSB

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 1;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 0; //Data LSB

#2500

i2c\_SDA = 0; //nothing

#2500

i2c\_SDA = 1; //Data MSB

#2500

i2c\_SDA = 0;

#2500

i2c\_SDA = 1;

#100 //Random Reset press

Reset = 0;

#10

Reset = 1; //Reset unpress

// Add stimulus here

end

endmodule

*4.2 deserial\_test2.v*

`timescale 1ns / 1ps

module deserial\_test2;

// Inputs

reg Clock;

reg Reset;

reg i2c\_sda;

reg i2c\_scl;

reg [2:0] i2c\_addr\_bits;

// Outputs

wire i2c\_RW;

wire [10:0] i2c\_addr;

wire addr\_ack;

wire slave\_ack;

wire data\_ack;

wire [7:0] serial\_data;

wire stop\_out;

// Instantiate the Unit Under Test (UUT)

i2c\_slave\_deserializer uut (

.Clock(Clock),

.Reset(Reset),

.i2c\_sda(i2c\_sda),

.i2c\_scl(i2c\_scl),

.i2c\_addr\_bits(i2c\_addr\_bits),

.i2c\_RW(i2c\_RW),

.i2c\_addr(i2c\_addr),

.addr\_ack(addr\_ack),

.slave\_ack(slave\_ack),

.data\_ack(data\_ack),

.serial\_data(serial\_data),

.stop\_out(stop\_out)

);

initial begin

// Initialize Inputs

Clock = 0;

Reset = 0;

i2c\_sda = 0;

i2c\_scl = 0;

i2c\_addr\_bits = 0;

// Wait 100 ns for global reset to finish

#100;

end

always #5 Clock = !Clock;

always #200 i2c\_scl = !i2c\_scl;

//always #200 i2c\_sda = !i2c\_sda;

initial begin

i2c\_scl = 1;

i2c\_sda = 1;

i2c\_addr\_bits = 3'b101;

Reset = 1;

#10

Reset = 0;

#10

Reset = 1;

#120

i2c\_sda = 0;

#190

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 1;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#400

i2c\_sda = 0;

#400

i2c\_sda = 1;

#1000

Reset = 0;

#10

Reset = 1;

// Add stimulus here

end

endmodule

*4.3 sequencer\_test1.v*

initial begin

// Initialize Inputs

Clock = 0;

i2c\_RW = 0;

i2c\_addr\_in = 0;

i2c\_data\_in = 0;

i2c\_addr\_ack = 0;

i2c\_data\_ack = 0;

reset = 0;

stop = 0;

// Wait 100 ns for global reset to finish

#100;

end

always #50 Clock = !Clock;

always #1250 i2c\_scl = !i2c\_scl;

//always #200 i2c\_sda = !i2c\_sda;

// Add stimulus here

reset = 1'b1;

#100;

reset = 1'b0;

#100

i2c\_RW = 0;

#5000

i2c\_addr\_in [10:0] = 11'b11111100000;

#2000

i2c\_addr\_ack = 1;

#1250

i2c\_addr\_ack = 0;

end

endmodule

**Appendix E: Industry Specifciations**

1. I2S
2. I2C

Included in the PDF version