

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



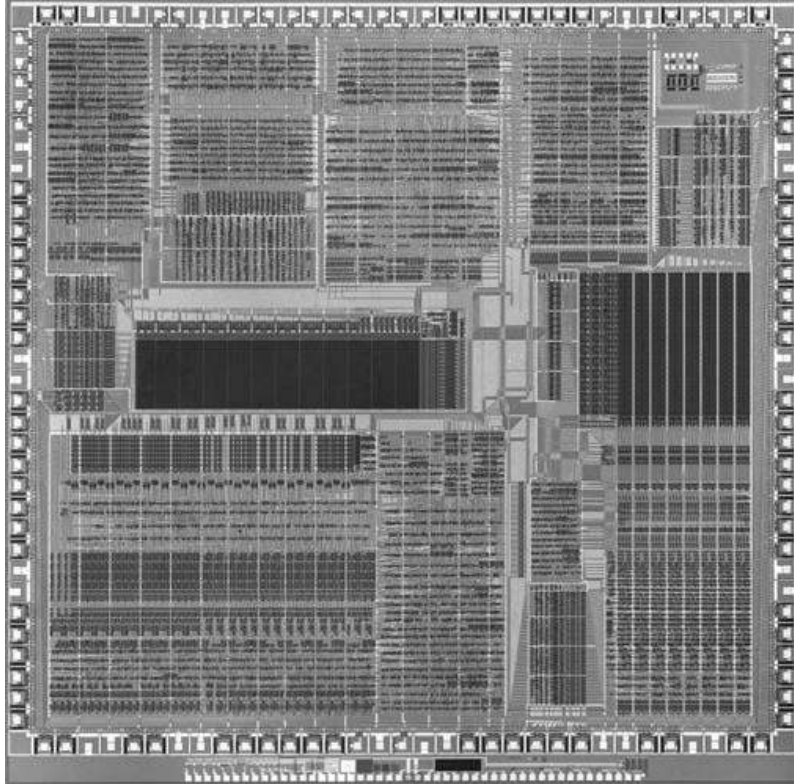
Kevin Cao, Whitley Forman, Dhruvit Naik,
Zachary Nelson* and Julie Swift

*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. 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
Acronyms	6
Introduction	7
Team Management.....	9
Specifications.....	10
Chapter 1: Background	11
Chapter 2: I2S_IN and I2S_OUT	15
Chapter 3: Digital Filtering	31
Chapter 4: Register Block	45
Chapter 5: I2C Interface.....	54
Chapter 6: Budget	67
Chapter 7: Schedule	68
Chapter 8: Conclusion.....	69
References.....	70
Appendix A: Team Management.....	71
Appendix B: Code.....	91
Appendix C: Test Benches.....	142
Appendix D: Industry Specifications.....	171

List of Tables

Table 1.1: Comparison of MOSIS Academic Account Types.....	12
Table 2.1: Interface Signals for I2S Input.....	18
Table 2.2: Interface Signals for Synchronizer Sub-Module	20
Table 2.3: Interface Signals for Deserializer Sub-Module	21
Table 2.4: Interface Signals for BIST Generator Sub-Module	23
Table 2.5: Interface Signals for Multiplexer Sub-Module.....	24
Table 2.6: Interface Signals for FIFO Sub-Module	25
Table 2.7: Interface Signals for I2S Output.....	27
Table 2.8: Interface Signals for Serializer Sub-Module	29
Table 3.1: Interface Signals for Filter.....	33
Table 3.2: Interface Signals for Filter Finite-State Machine	35
Table 3.3: Interface Signals for Filter Storage Module	37
Table 3.4: Interface Signals for Filter Mux Module	38
Table 3.5: Interface Signals for Filter Accumulator Module.....	38
Table 3.6: Interface Signals for Filter Barrel Shifter Module.....	39
Table 4.1: Register Mappings	47
Table 5.1: I2C Register Table	59

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	13
Figure 2.1: Overall Chip Diagram with I2S Interfaces Highlighted	16
Figure 2.2: I2S Input Block Diagram	17
Figure 2.3: Sample of I2S Input Interface Simulation	19
Figure 2.4: Sample of Synchronizer Simulation	21
Figure 2.5: Sample of Deserializer Simulation	22
Figure 2.6: Sample of BIST Generator Simulation	23
Figure 2.7: Sample of Multiplexer Simulation	24
Figure 2.8: Sample of FIFO Simulation	26
Figure 2.9: Block Diagram of I2S Output Interface	26
Figure 2.10: Sample of I2S Output Interface Simulation	28
Figure 2.11: Sample of Serializer Simulation	30
Figure 3.1: Frequency Response of Filter Structures	31
Figure 3.2: Chip Diagram with Filter Highlighted.....	32
Figure 3.3: Top Level Block Diagram of Filter	33
Figure 3.4: Detailed Block Diagram of Filter	34
Figure 3.5: Finite State Machine: States and Transitions	36
Figure 3.6: Impulse Response of 512-Tap Low-Pass Filter	39
Figure 3.7: Simulation of Barrel Shifter.....	40
Figure 3.8: Simulation of Accumulator	40
Figure 3.9: Simulation of Filter Storage Module-Test 2	41
Figure 3.10: Simulation of Filter Storage Module-Test 2	41
Figure 3.11: Simulation of Mux	41
Figure 3.12: Simulation of Filter-Test 1.....	42
Figure 3.13: Simulation of Filter-Test 2.....	42
Figure 3.14: Simulation of Filter-Test 3.....	42
Figure 3.15: Simulation of Filter-Test 4.....	43
Figure 3.16: Simulation of Filter-Test 5.....	44
Figure 4.1: Overall Chip Diagram with Register Highlighted.....	45
Figure 4.2: Internal Micro-Architecture	46
Figure 4.3: Top Level Interface Design of Reg.v	46
Figure 4.4: The Start of the Test Fixture Simulation	51
Figure 4.5: Bits trig_i2si_fifo_overrun_clr and trig_i2so_fifo_underrun_clr are Triggered	51
Figure 4.6: Bits trig_i2si_fifo_overrun_clr and trig_i2so_fifo_underrun_clr are Triggered in Second Testbench	52
Figure 5.1: I2C Chip Diagram Highlight.....	54
Figure 5.2: I2C Full Coefficient Time Load Calculation	55
Figure 5.3: I2C Block Diagram Input.....	57
Figure 5.4: Open Drain Schematic	58
Figure 5.5: Previous Block Diagram Version	61
Figure 5.6: I2C State Machine Flow Diagram	61
Figure 5.7: I2C Deserializer Block Diagram	63
Figure 5.8: I2C Sequence Block Diagram.....	64
Figure 5.9: I2C Serializer Block Diagram	65

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 interchangeably throughout the report and refer to designing 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, a smaller chip since there are no unwanted components included, higher clock rates, and lower power dissipation. 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.

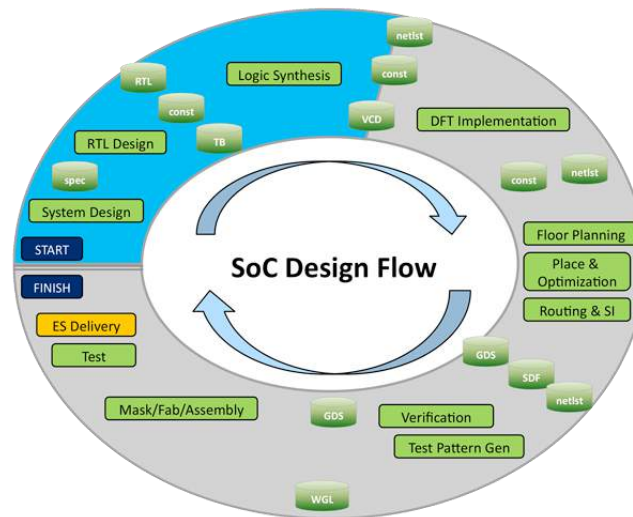


Fig I.1: Process of Designing a System on a Chip

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 described. 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 University 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 D
- 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 independently settable
- Programmable filter coefficients to achieve different filter types
- Maintain integer headroom of 4 bits

I2C:

- 8-bit slave address space
- 12-bit register address space
- Data transfer rate of up to 400 kbits/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 is 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.

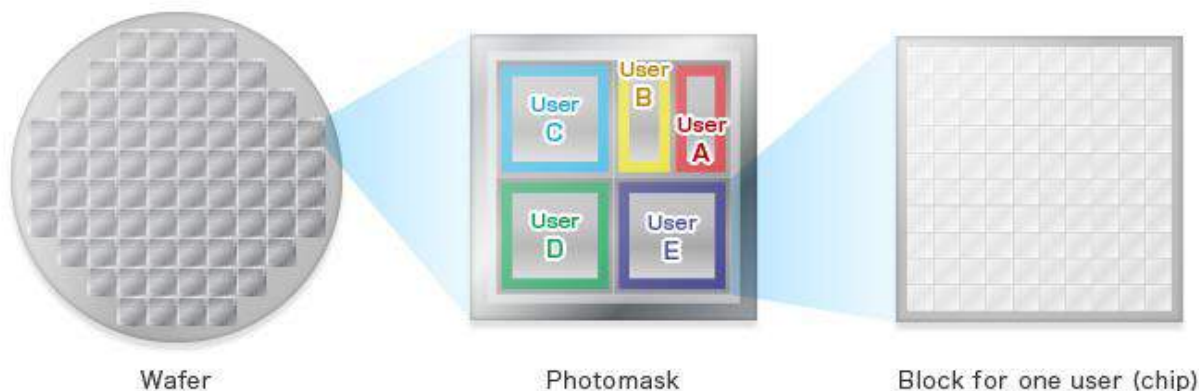


Fig 1.1: Multi-Project Wafer

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 GlobalFoundries 180 nm CMOS (7HV) process or GlobalFoundries 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 0.50 μ m CMOS (C5N) 2. GlobalFoundaries 180 nm CMOS (7HV)	1. ON Semi 0.50 μ m CMOS (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 16mm ²
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.

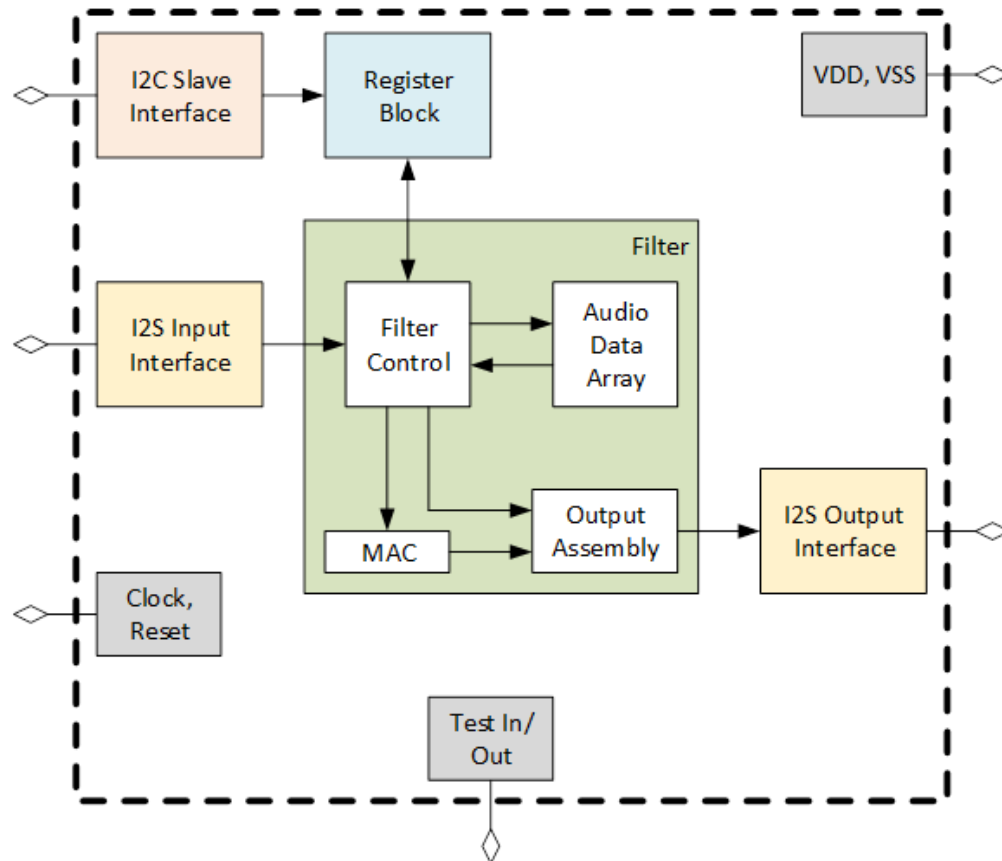


Fig 1.2: Top Level Drawing of Chip

The main purpose of the I2S input interface was to convert a serial audio 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 modules 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 filter 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 coefficients from the register block in order to implement the FIR filter.

Control Flow: I2C Slave Interface → Register Block → Filter

The PSoC microcontroller will be used to generate the audio stream for the chip. Since we will create the audio stream in the PSoC environment, we should know exactly what the output stream will be based on the filter coefficients we are using. The microcontroller will also have the ability to read and write filter coefficients to the chip. This is an important feature of our project because the user will be able to manually set and check the values of all 512 filter coefficients. We will also use an UDA 1380 board to encode and decode stereo audio for the I2S interface. This means that the chip will also be capable of using a stereo audio input instead of the input stream provided by the microcontroller. Either way, the chip should be able to successfully apply the 512-tap filter to the audio stream.

Chapter 2: I2S_IN and I2S_OUT - Kevin Cao

2.1 Introduction

Integrated Interchip Sound (I2S) buses are used to pass digital audio into our chip for processing, and to carry processed digital audio streams that are outputted by our chip. I2S is an electrical serial bus interface standard that is used for connecting digital audio devices together. I2S is used to communicate pulse-code modulation (PCM) audio data between integrated circuits in an electronic device. PCM is a standard format for digital audio in computers. It is a method used to digitally represent sampled analog signals.

The **i2s_in** block is responsible for receiving the I2S signals that are inputted to our chip, and converting this to a natural parallel PCM format, suitable for processing by the filter block. The **i2s_out** block is responsible for converting parallel PCM data produced by the filter block to a serial I2S stream, for output by the chip.

2.2 I2S Bus Specification

The I2S protocol contains three lines: serial data bus (SD), a bit clock (SCK), and word select (WS). These three lines are used to reduce the number of pins required and to keep wiring simple. The SCK signal pulses once for each discrete bit of data on the data line and its frequency 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 sent 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 audio sample frequency.

The SD signal is a serial representation of a two's complement signed value, and is transmitted with the most significant bit (MSB) first. The transmitter and receiver may have different natural word lengths. The transmitter and receiver doesn't need to know how many bits the other can handle. The least significant data bits are set to 0 when the receiver's 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 internally set to 0. Thus the MSB is always located in the same position, however, the least significant bit (LSB) position depends on the word length. The MSB of the new word 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 by 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 interfaces for the custom chip were split into two Verilog modules, **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.

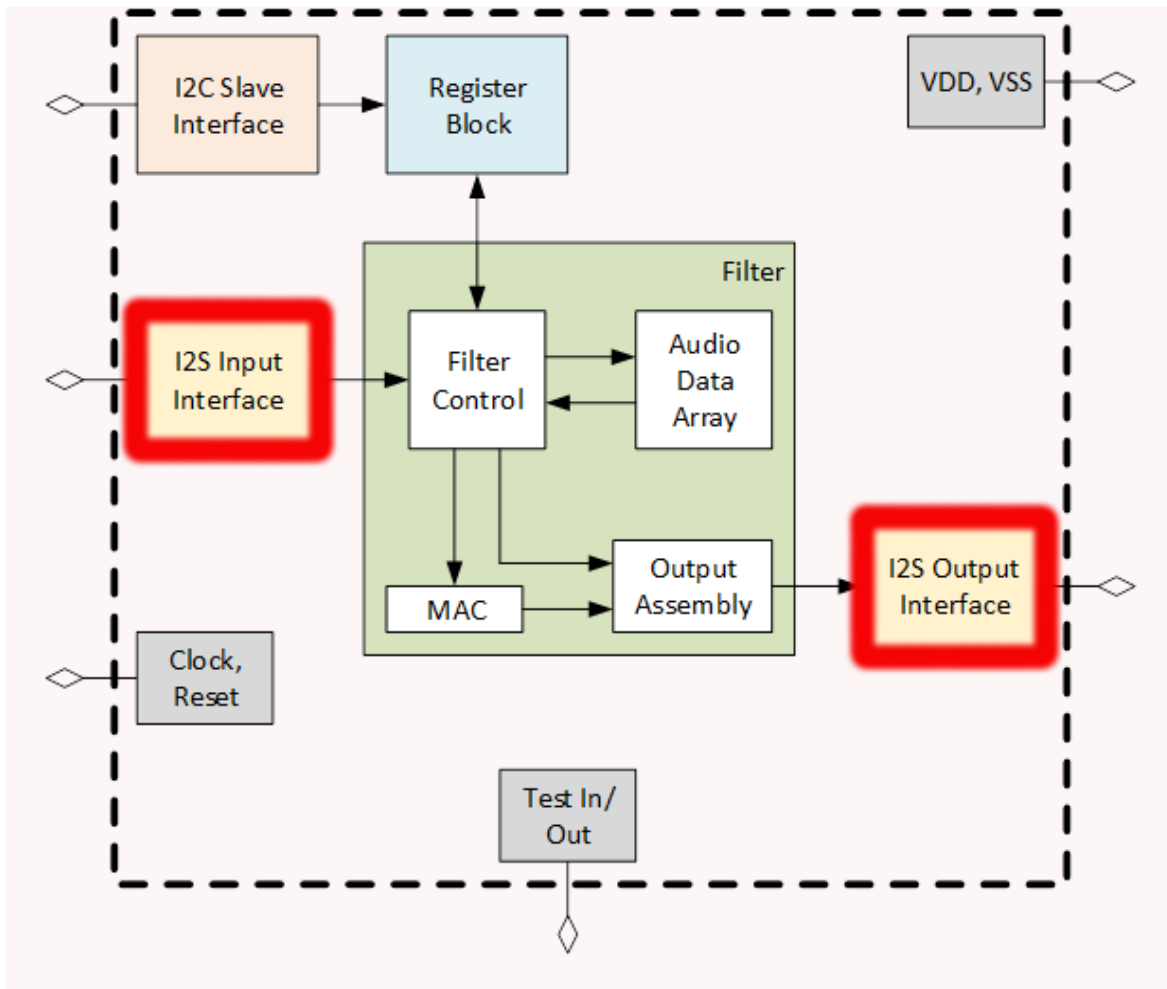


Fig 2.1: Overall Chip Diagram with I2S Interfaces Highlighted

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, Audio Built-In Self-Test (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.

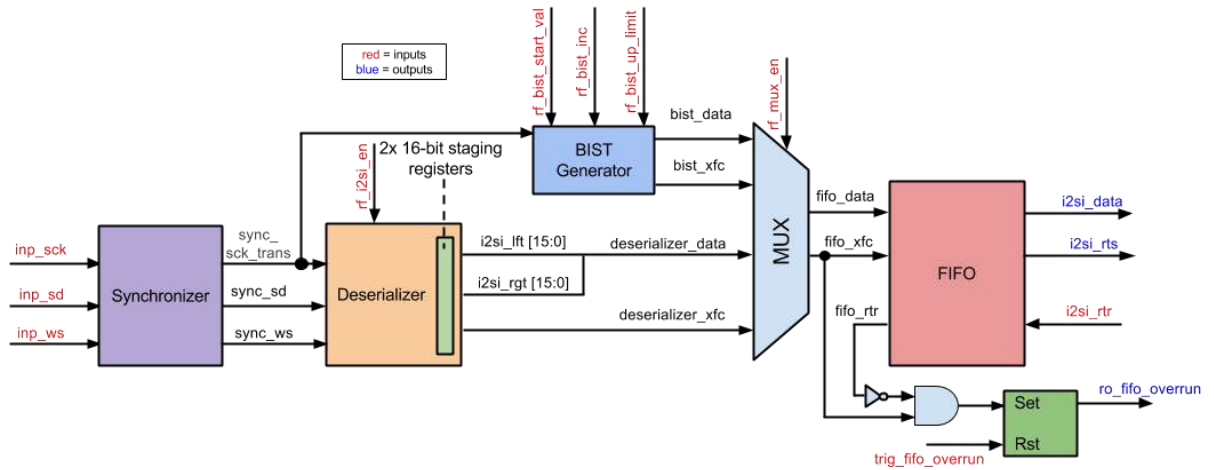


Fig 2.2: I2S Input Block Diagram

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: Interface Signals for I2S Input

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
I2S Input Interface			
inp_sck	in	1	Digital Audio Bit Clock
inp_sd	in	1	Digital Audio Serial Data
inp_ws	in	1	Word Select
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
Control/Status Register Fields			
trig_i2si_fifo_overnun_clr	in	1	Reset FIFO Overrun
ro_fifo_overnun	out	1	Input Audio FIFO Overrun
Streaming Audio Interface with Filter Block			
i2si_rtr	in	1	Ready to Receive
i2si_rts	out	1	Ready to Send
i2si_data	out	32	Output Digital Audio

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 filter block. They inform the I2S input interface when the filter block is ready to receive data, and inform the filter block when the I2S input interface is read to send data. The `i2si_data` signal contains the 32 bit words that will be sent to the filter block for

processing. The rf_i2si_en signal informs the I2S input block to start converting data from serial into parallel. The rf_mux_en signal is the select bit that informs the I2S input interface either to output the deserialized data or the BIST data to the filter block. The signals rf_bist_start_val, rf_bist_inc, and rf_bist_up_limit are used to initially test the chip when it first starts up. The signal ro_fifo_overnrun indicates whether data is written to the block faster than it can be read. The trig_fifo_overnrun signal indicates whether or not to clear the ro_fifo_overnrun signal.

The I2S input interface was verified by inputting a list of 32 bit words into the inp_sd signal line. The i2si_data signal was then observed to see if it outputted the same words inputted. It was observed that the same data inputted is also outputted within the block and appears to be working. The select bit for the multiplexer was also toggled to verify that the data outputted came from either the BIST or input serial data. It was confirmed that the data outputted was dependent on the select bit and the block was functioning properly. The i2si_rts signal line was also observed. The signal is supposed to pulse after each half word within the block. When observing the test benches this can be seen functioning correctly.

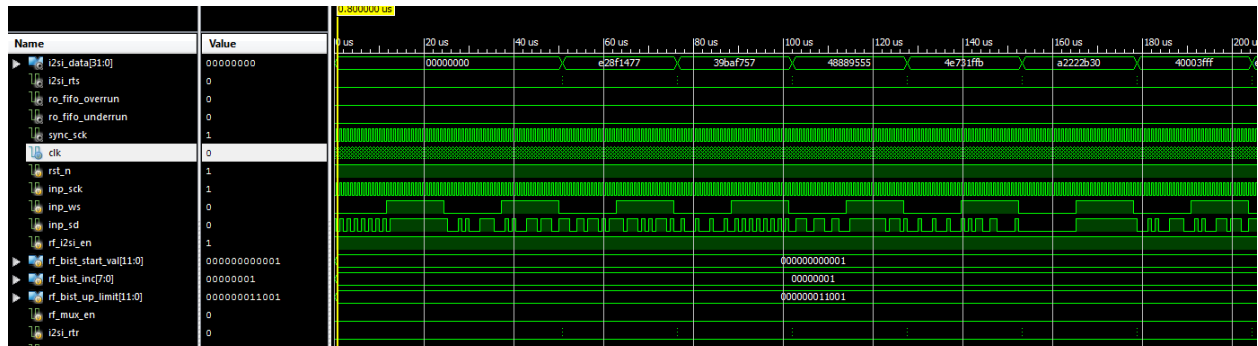


Figure 2.3: Sample of I2S Input Interface Simulation

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. 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: Interface Signals for Synchronizer Sub-Module

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
Synchronizer Input Interface			
_sck	in	1	Digital Audio Bit Clock
_ws	in	1	Word Select
_sd	in	1	Digital Audio Serial Data
Synchronizer Output Interface			
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.

The synchronizer was verified by checking if the _sck, _ws, and _sd signals were delayed by correct amount of clock cycles. It can be seen within the sample simulation that the sck signal was delayed by 2 clock cycles, and both the ws and sd signals were delayed by 4 clock cycles. In addition, the level to pulse converter for sck, sck_transition should have been high for every time sck transitioned from low to high. As displayed in the simulation, the level to pulse converter does work.

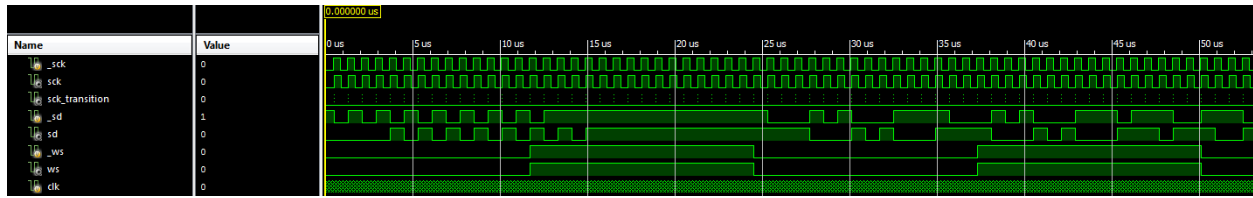


Figure 2.4: Sample of Synchronizer Simulation

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: Interface Signals for Deserializer Sub-Module

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
sck_transition	in	1	Serial Clock Level to Pulse Converter
Deserializer Input Interface			
rf_i2si_en	in	1	i2s input is enabled
in_sd	in	1	Digital Audio Serial Data
in_ws	in	1	Word Select (Left/Right Audio Channel)
Deserializer Output Interface			
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 serial data and word select signals. 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.

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.

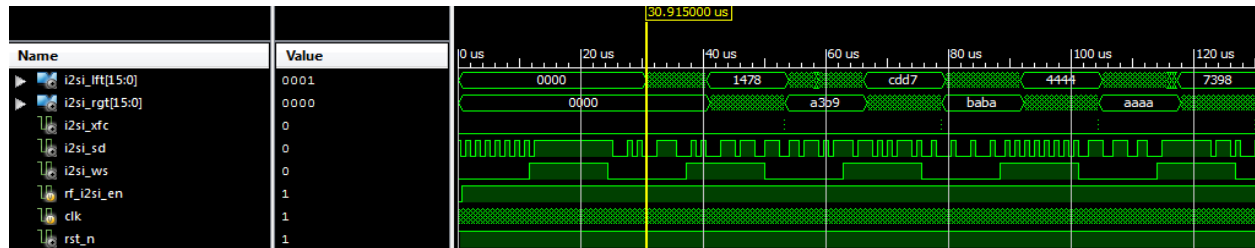


Fig. 2.5: Sample of Deserializer Simulation

2.4.3 Audio Built-In Self-Test (BIST) Generator

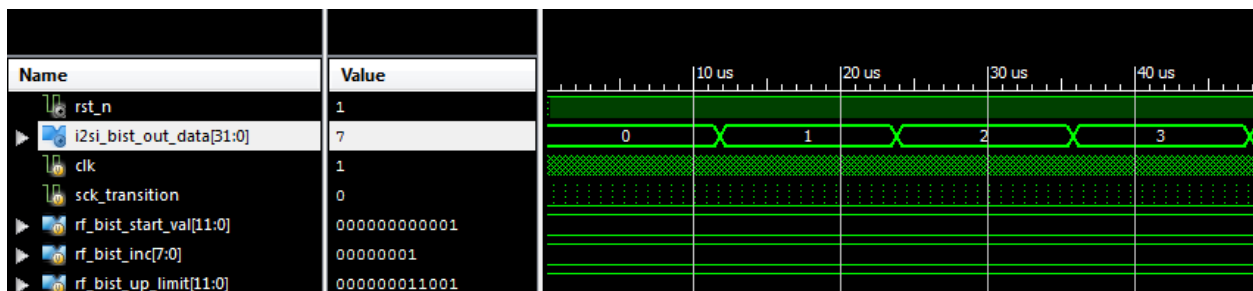
The BIST generator sub-block is used to create a saw-tooth wave to test the I2S input interface. The saw-tooth wave is then outputted to the multiplexer. The following table lists the inputs and outputs of the BIST generator sub-block.

Table 2.4: Interface Signals for BIST Generator Sub-Module

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
sck_transition	in	1	Serial Clock Level to Pulse Converter
BIST Input Interface			
rf_bist_start_val	in	12	Start Value
rf_bist_inc	in	8	Increment
rf_bist_up_limit	in	12	Upper Limit
BIST Output Interface			
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 32nd pulse of the serial clock. The xfc output signal was verified by checking if it was high after the data incremented 32 times. The sub-block was further tested by ensuring that the output was reset to 0 after the output data reaches the upper limit of the BIST generator.

**Fig. 2.6:** 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: Interface Signals for Multiplexer Sub-Module

Signal Name	Direction	Bits	Comment
Data Lines Input Interface			
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
Enabled Bit			
sel	in	1	Select Bit
Multiplexer Output Interface			
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.

The multiplexer was verified by running a simple simulation that inputted two sets of values for the pairs in_0 and in_1. The enabled bit was then toggled to see if the multiplexer outputted the correct pair of values.

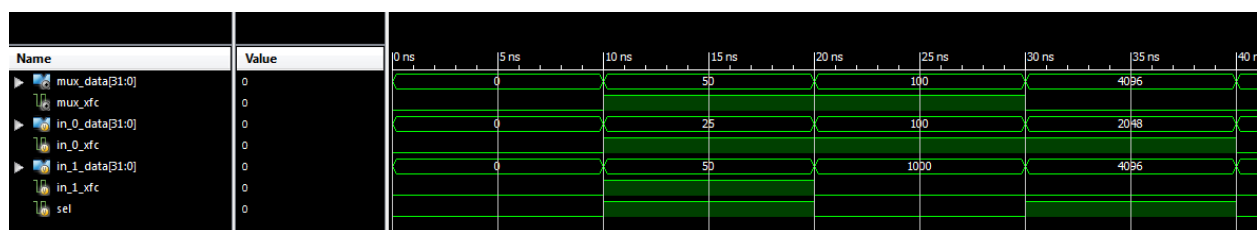


Figure 2.7: Sample of Multiplexer Simulation

2.4.5 First-In-First-Out (FIFO) Buffer

The FIFO sub-block organizes and manipulates a data buffer, where the first entry is the first output. In this particular block the FIFO 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: Interface Signals for FIFO Sub-Module

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
Streaming Audio Interface with Multiplexer			
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 Ready to Receive
Streaming Audio Interface with Filter Block			
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 queuing and de-queuing random values within a test bench. A quick simulation was run to check if the FIFO queued and de-queued the values defined within the test bench. If the FIFO was full the FIFO should have stopped queuing more data into the buffer, and if the FIFO was empty the FIFO should have outputted nothing from the buffer.

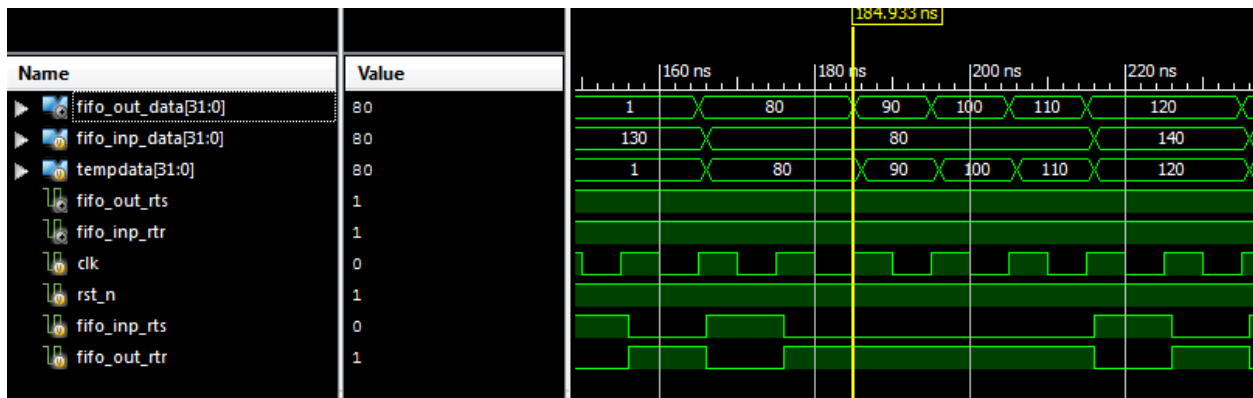


Fig. 2.8: Sample of FIFO Simulation

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.

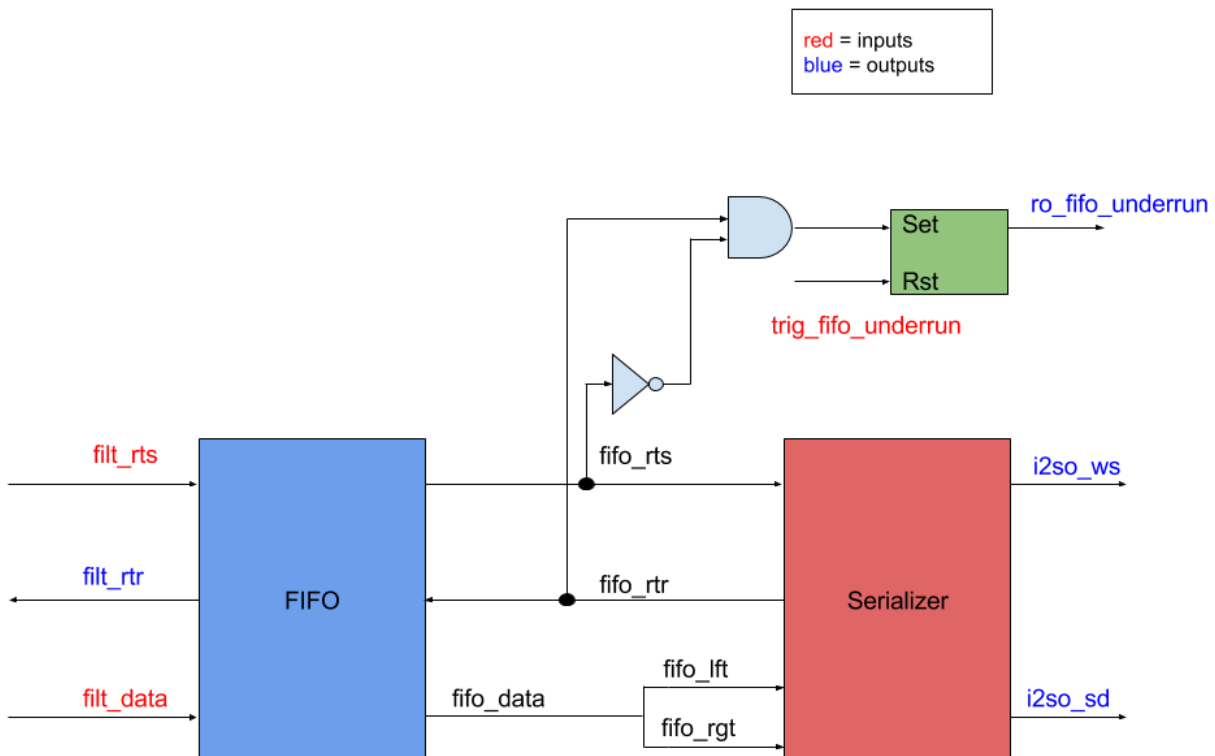


Fig 2.9: Block Diagram of I2S Output Interface

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: Interface Signals for I2S Output

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
sck_transition	in	1	Serial Clock Pulse Converter
Streaming Audio Interface with Filter Block			
filt_data	in	32	Parallel Digital Audio
filt_rtr	out	1	Ready to Receive
filt_rts	in	1	Ready to Send
I2S Input Interface			
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
Control/Status Register Fields			
trig_fifo_underrun_clr	in	1	Reset FIFO Underrun
ro_fifo_underrun	out	1	Output Audio FIFO Underrun

The sck_transition signal is the level to pulse converter signal that will be provided by the I2S input interface. The signal filt_data will be provided by the filter block that contains the 32-bit parallel audio data. The signals filt_rtr and filt_rts are the handshaked interface signals that indicate whether the filter is ready to send data and the I2S output interface is ready to receive data. The i2so_sck is the serial clock that is outputted in the I2S Input interface that will be transmitted to the I2S output interface. The i2so_ws is an output signal that indicates whether the data being transmitted belongs either to the left or right channel. The i2so_sd signal is the serial data that is converted from the parallel audio input data. The ro_fifo_underrun signal indicates whether the

data being fed to the I2S output interface is slower than the rate it can process the data, and the `trig_i2so_fifo_underrun_clr` signal clears the `ro_fifo_underrun` bit.

The I2S output interface was verified by inputting a list of 32 bit words and confirming that they were being serialized upon exiting the block. It was observed from the test benches that after a short delay the signals inputted into the I2S output interface exit the block in a serialized form. It was confirmed that the output word select signal properly lines up with the serial data signal and each bit is properly placed in their correct channel.

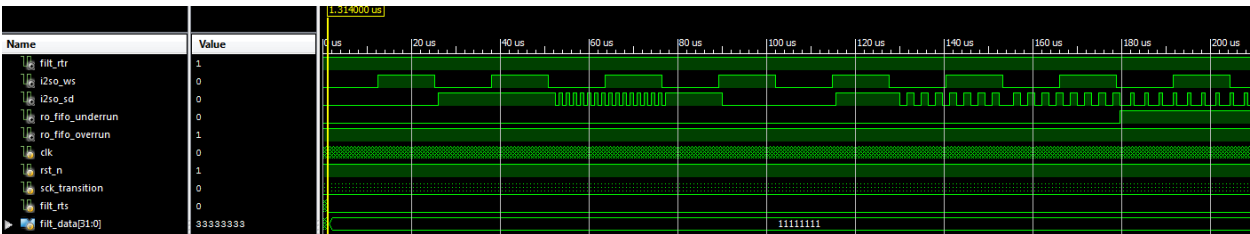


Figure 2.10: Sample of I2S Output Interface Simulation

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: Interface Signals for Serializer Sub-Module

Signal Name	Direction	Bits	Comment
General			
clk	in	1	Master Clock
rst_n	in	1	Reset
sck_transition	in	1	Serial Clock Level to Pulse Converter
Serializer Input Interface			
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
Serializer Output Interface			
i2so_sd	out	1	Digital Audio Serial Data
i2so_ws	out	1	Word Select

As previously stated the sck_transition 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. 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_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.

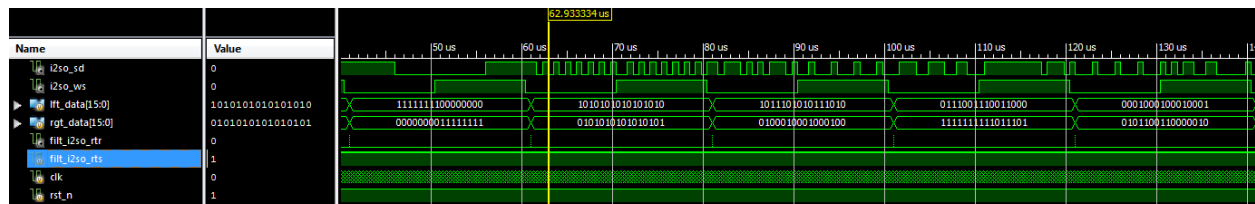


Fig. 2.11: Sample of Serializer Simulation

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, two of the most basic uses of a digital filter are 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. More advance filters are used for enhancements, compensation for listening room dynamics, special effects, and many other features. 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 an MP3 Player, digital filter structures are cascaded together. The four most basic structures for a filter are high pass, low pass, band pass, and band reject. Below in Figure 3.1, shows the frequency response of each filter structure. A filter can be represented with its frequency response. Which is the fast Fourier transform of the impulse response.

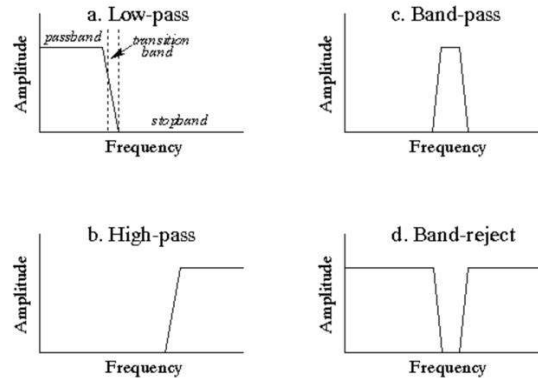


Fig. 3.1: Frequency Response of Filter Structures

Above, each filter structure has 3 interesting regions. The passband is the area in which frequency is allowed to pass freely. The stopband is the area in which the frequency is attenuated. Finally, the transition band is the transition between the passband and the stopband. 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 an FIR filter, the impulse response are the filter coefficients. The given filter coefficients characterize the type of the digital filter. Convolution is defined by the equation:

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$$

Here 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 chip will support sampling rates up to 48 kHz with precision of 16 bits per audio channel (left and right). The bit depth refers to the number of bits of precision when quantizing the sampled signal. The number of bits reduces the quantization error. Thus reducing the “tape hiss” associated with lower bit precision.

3.2 Filter

The filter block is a sub-block of the top level chip. Figure 3.2 shows the filter with respect to the other interfaces.

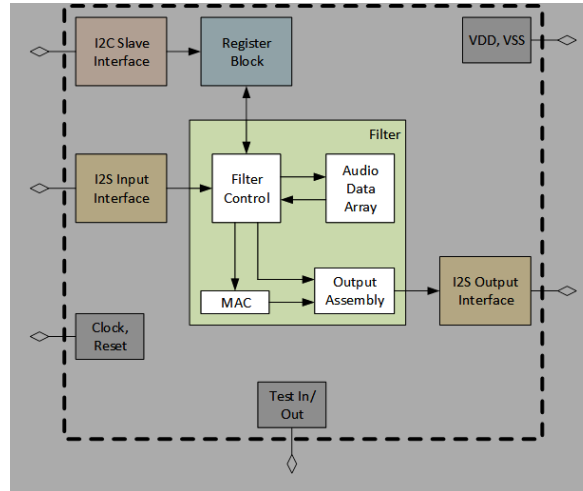


Fig 3.2: Chip Diagram with Filter Highlighted

The filter interfaces with the I2S input and output blocks, 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, including past input signals are convolved with the filter coefficients. Before the signal is sent to the I2S output, three operations are performed: rounding, shifting, and clipping. Figure 3.2 is the top level view of the filter. Table 3.1 describes the corresponding signals for the top level filter.

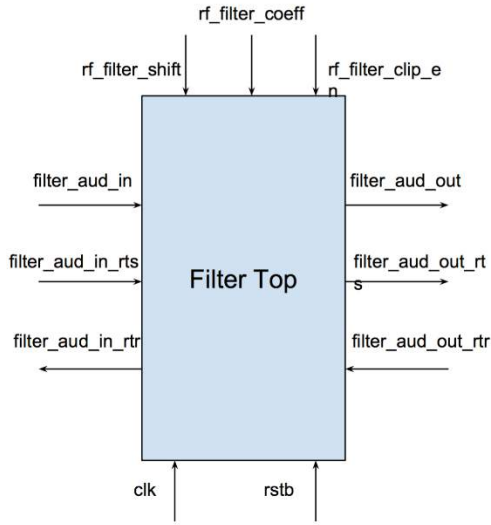


Fig 3.3: Top Level Block Diagram of Filter

Table 3.1: Interface Signals for Filter

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	5	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 right audio channel bits are stored in the first 16 LSB [15:0] and the left 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. Two things of note. First the multiplication will be signed multiplication. This is done by defining wires and registers as signed type. Secondly, there will be two instantiations of the accumulator for left and right audio signals. *Filter_aud_in* will be split into two before the multiplication and concatenated before the three operations.

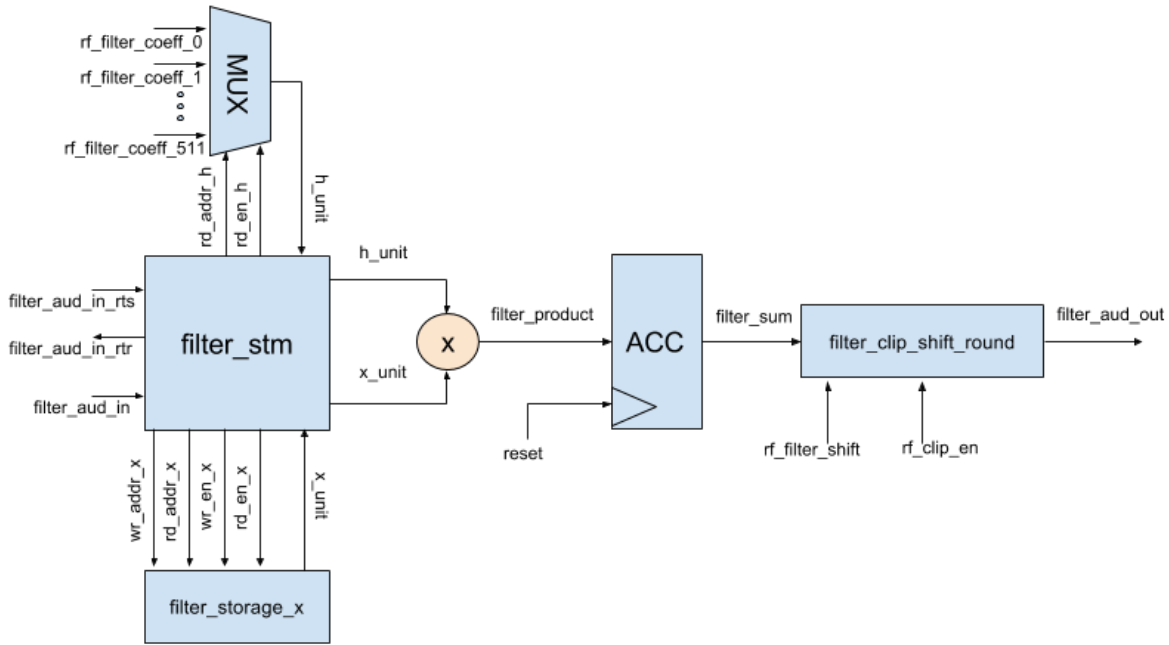


Fig 3.4: Detailed Block Diagram of Filter

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). The states of a FSM can be encoded in various ways – one convenient encoding is known as One-Hot. In a One-Hot encoding each state is represented by a single bit.

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. Table 3.2 shows the highlights and describes the signals for the FSM.

Table 3.2: Interface Signals for Finite State Machine

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.

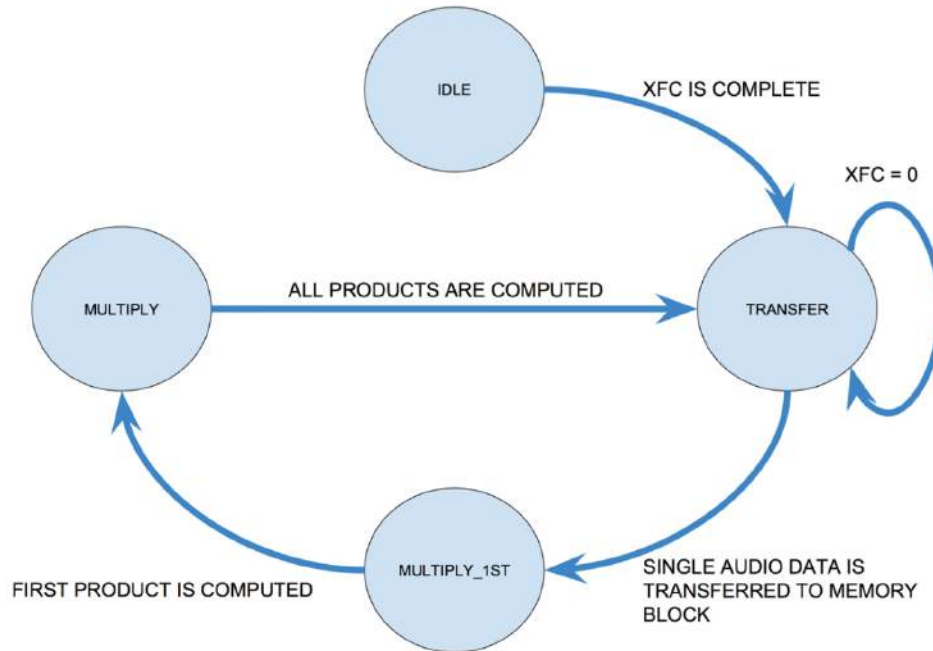


Fig 3.5: FSM: States and Transitions

IDLE:

The FSM enters the IDLE state if, and only if, the RESET signal is asserted. Upon assertion of XFC the FSM will transition to the TRANSFER state.

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 MULTIPLY_1ST after this is complete. If XFC is deasserted, it will wait until it is asserted to write *filter_aud_in* and transition.

MULTIPLY_1ST:

The convolution phase is separated into two states because the accumulator needs to load the first product. After the first computation is completed the state will transition to MULTIPLY.

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.

$$1) \ y[0] = h[0] x[0] + h[1] x[-1] + \dots + h[510] x[-510] + h[511] x[-511]$$

$$2) \ y[1] = h[0] x[1] + h[1] x[0] + \dots + h[510] x[-509] + h[511] x[-510]$$

$$3) \ y[2] = h[0] x[2] + h[1] x[1] + \dots + h[510] x[-508] + h[511] x[-509]$$

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. 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: Interface 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	9	Current Location of Write Pointer
wrdata	out	32	Data to be Written
rden	out	1	1-rddata will be read from rdptr
rdptr	in	9	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 Mux

The storage module is sufficient for indexing and reading filter coefficients but does not satisfy our requirements. Using the storage module will be inefficient in reading the filter coefficients because the data is already stored in Register block and would be redundant to write

them in an array located in the Filter block. Instead, a mux can be used to read the correct filter coefficients using a bus of wires. In the Register block, a single filter coefficient is represented by two signals: `rf_filter_coeffn_a` and `rf_filter_coeffn_b`. Instead of reading two signals for a single filter coefficient, the mux concatenates the two wires into a single wire. Signal `rf_filter_coeffn_a` is stored in the first 8 LSB [7:0] and `rf_filter_coeffn_b` is stored in the first 8 MSB [15:8]. Below in Table 3.4, shows the signals for the mux submodule. A clock is placed to keep it in sync with `filter_aud_in` values.

Table 3.4: Interface Signals for Mux Module

Signal Name	Direction	Bits	Comment
clk	in	32	Signal being performed on
rden	in	1	How many bits being shifted
rdptr	in	9	Location of data to be selected
rf_filter_coef[0:511]	in	8196	Filter Coefficient
rddata	out	15	Selected Data

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. The signals are defined in the below table. When *enable* is asserted, the accumulator will function as normal. When *load* is enabled, *D* is set to the new value.

Table 3.5: Interface Signals for Accumulator Module

Signal Name	Direction	Bits	Comment
clk	in	1	Master Clock
rstb	in	1	Asynchronous Reset
enable	in	1	1-Accumulate 0-Idle
load	in	1	Load new value
D	in	32	Input Value
Q	out	40	Current Total

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

Table 3.6: Interface 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

To verify the Filter Block, I performed two types of tests, individual unit testing for each submodule and full block level test. The full block level test was to verify the functionality of the block. I did preliminary testing of actual filter responses. The following sections will outline testing from the lowest module to the highest module. As far as functionality, the filter block passes all the current tests. For winter break, I plan on verifying if the filter produces actual filter responses. This will be done by using filter coefficients produced by the Matlab FDA toolbox. Below are is an example impulse response of 512-tap Low-Pass filter.

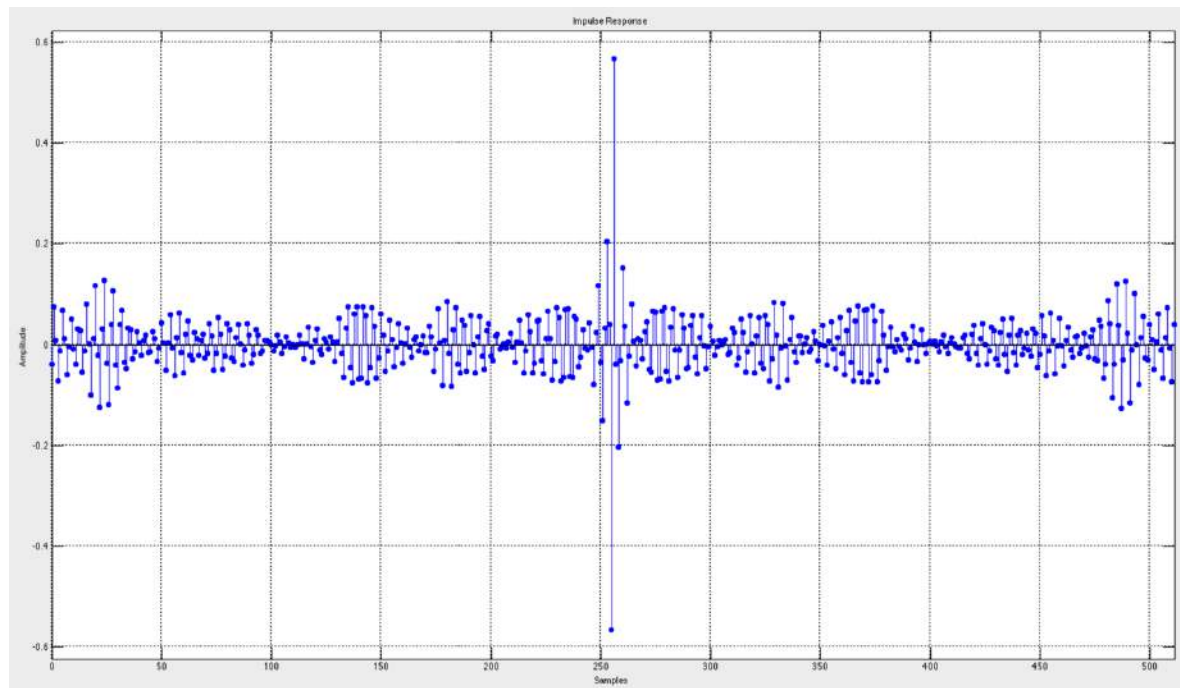


Fig 3.6: Impulse Response of 512-Tap Low-Pass Filter

3.3.1 Barrel Shifter

To verify the Barrel Shifter, I first inputted a 32-bit hexadecimal value of 0xABCD1234. Then I varied the sel_shift from 0 to 32 incrementing by 4 bits at a time to preserve the byte value when shifting. Below is the simulation output. I verified the barrel shifter works as designed.

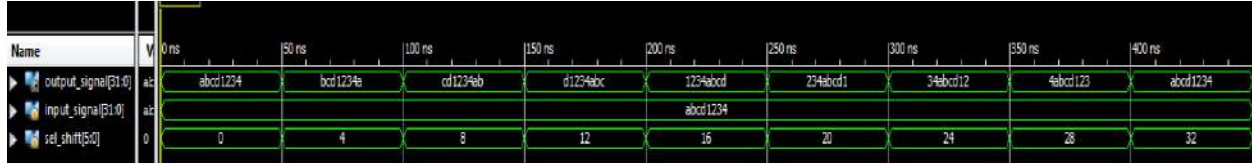


Fig 3.7: Simulation of Barrel Shifter

3.3.2 Accumulator

To verify the Accumulator, I tested three functions: loading, normal operation, and reset. The first test was to verify when load and enable were both asserted. What I expected was that the output would not accumulate and would equal to the input signal. The second test was to verify when load = 1'b0 and enable = 1'b1. I expected to see the accumulator increment by the input signal each clock cycle. The output signal has a 4 bit overhead to avoid overflow. The final test was to check what occurred when rstb = 1'b0. I expected the reset to take priority over the other signals and set the output to 1'b0. Below is the simulation output. All three of my assertions were proven correct.

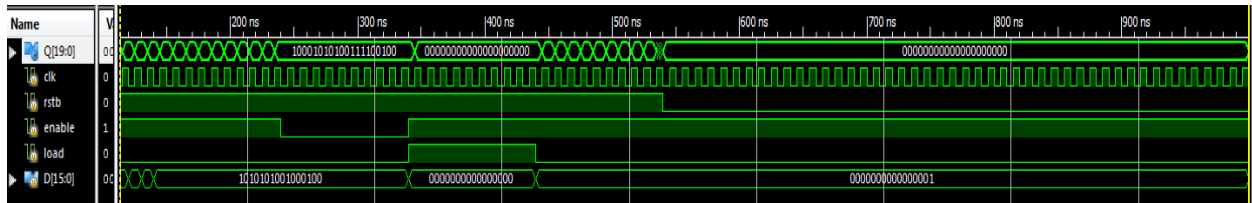


Fig 3.8: Simulation of Accumulator

3.3.3 Filter Storage Module

To verify the Filter Storage Module, I first tested basic writing and reading. I set the *wren* = 1'b1. I expect this to allow the module to write into the array at the given pointer. Starting with *wrdata* = 32'hAAAAAAAA, then *wrdata* = 32'hBBBBBBBB, and finally *wrdata* = 32'hCCCCCCCC. I increment the *wrptr* by 1'b1 after a 100 ns delay. To verify the values were stored correctly, I set *wren* = 1'b0 and *rden* = 1'b1. I read the array from ram[0] to ram[2], by incrementing *rdptr* by 1'b1 after a 100ns delay. To verify that the data is not lost when reading it, once I reached ram[2], I decremented by 1'b1 after 100ns delay. I expected to see *rddata* = 32'hAAAAAAAA, *rddata* = 32'hBBBBBBBB, *rddata* = 32'hCCCCCCCC. *rddata* = 32'hCCCCCCCC, *rddata* = 32'hBBBBBBBB, *rddata* = 32'hAAAAAAAA. Below is the simulation output. My assertions were proven correct.

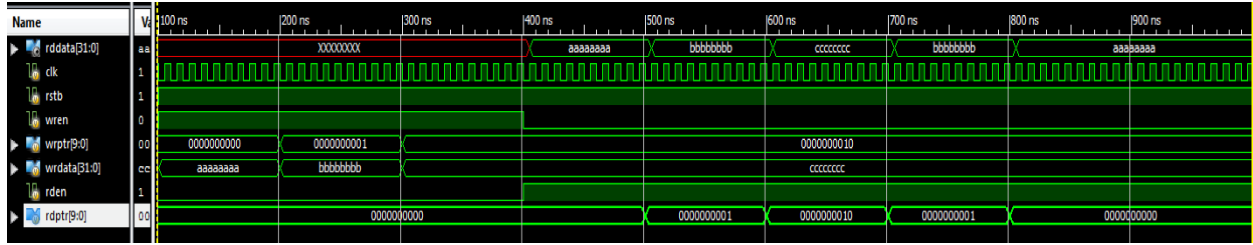


Fig 3.9: Simulation of Filter Storage Module Test 1

The second, test I performed was to check how storage module performed when the write and read pointer went above 511. I expect the storage module to perform circularly. To perform this test, I incremented the *wrptr* by 513. I expected the pointer to go back to *wrptr* = 1'd1. The write data was *wrdata* = 32'hBBBBBBBB. I then read *ram*[1]. I expected to have an output of *rddata* = 32'hBBBBBBBB. Below is the simulation output.

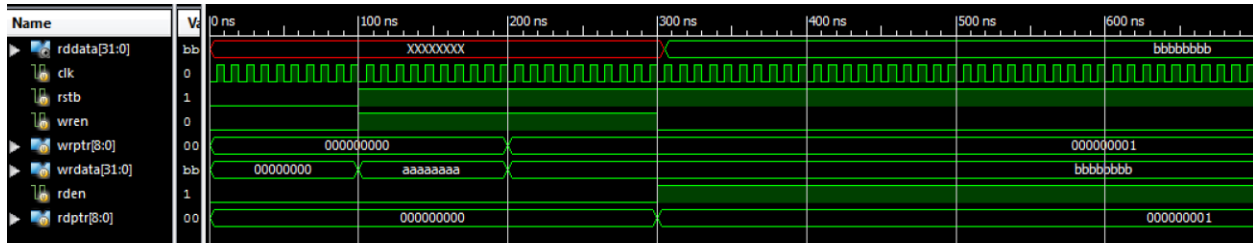


Fig 3.10: Simulation of Filter Storage Module Test 2

3.3.4 Mux

To verify the mux, I tested if I received the correct 16bit filter_coefficients. In addition, I tested if the mux correctly concatenated *rf_filter_coeffn_a* and *rf_filter_coeffn_b*. The 'a' component is stored from [7:0] and the 'b' component is stored from [15:8]. I set *rf_filter_coeff*(0-8)_a to 8'h00 to 8'h08, respectively. For *rf_filter_coeff*(0-8)_b, I set it to 8'h08 to 8'h00, respectively. For example, *rf_filter_coeff*0 should output 16'h0800. Another example would be, *rf_filter_coeff*3 should output 16'h0503. My test was successful as seen in the simulation output below.

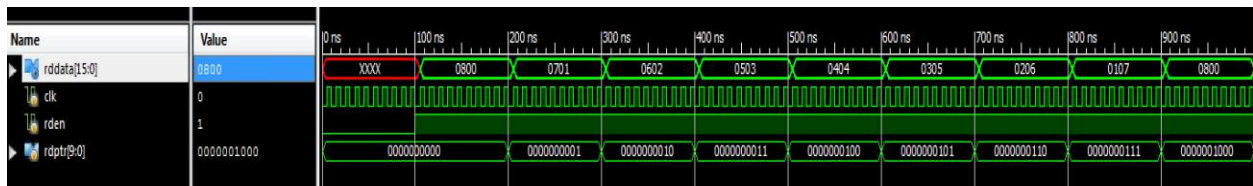


Fig 3.11: Simulation of Mux

3.3.4 Filter STM/Filter

I tested by filter state machine by testing my top level. For the first test I observed if the states were transitioning properly. For my design, the filter will kick-off when XFC is complete.

After a delay of 100ns, I asserted *aud_in_rts*. Below is my simulation. Note: for testing I am using an 8 tap filter. In the simulation once the filter kicks-off, it will cycle through the states until it is reset around 1000ns. The *do_transfer*, *do_multiply_1st*, and *do_multiply* represent when the states are running. For my STM implementation, it takes a single clock cycle to transition.

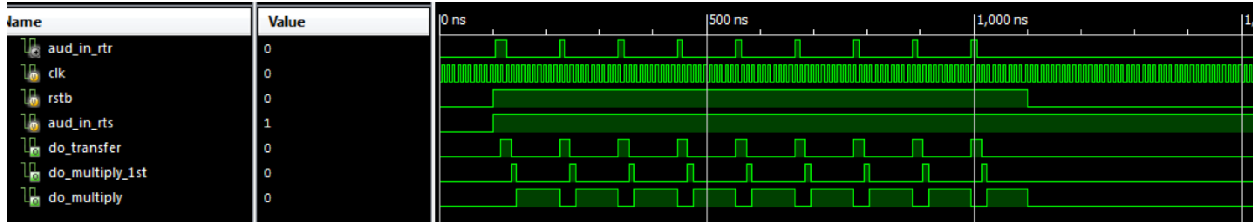


Fig 3.12: Simulation of Filter Test 1

The next test I performed was to vary the *aud_in_rts* signal from deasserted to asserted. It is possible for the I2S block to be not ready to send data when the filter block is ready to take data. To test this, I asserted *aud_in_rts* and deasserted it after 100ns. I then waited 400ns to asserted it again. The MULTIPLY states will not be affected by this, but the TRANSFER state will get delayed until *aud_in_rts* is asserted again.

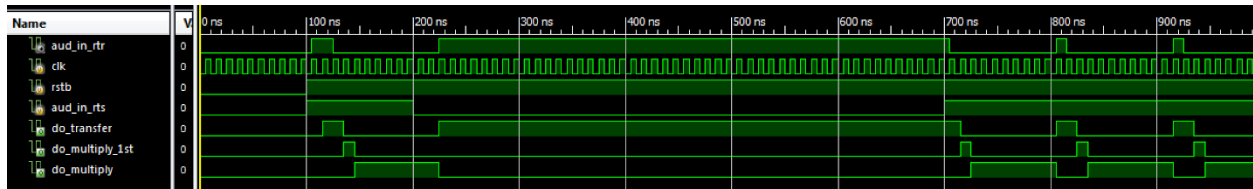


Fig 3.13: Simulation of Filter Test 2

Above the simulation, shows that the TRANSFER state is prolonged as it is waiting for XFC to be complete before it can transfer *aud_in* and change states. This can be proven by the simulation below. Here when *aud_in_rtr* and *aud_in_rts* are both deasserted the *aud_in* data will not be stored in the ram array. Around 700ns, *aud_in_rts* is asserted and *aud_in* is transferred into the current *wrptr*. After the transfer the state changes to the MULTIPLY_1ST stage.

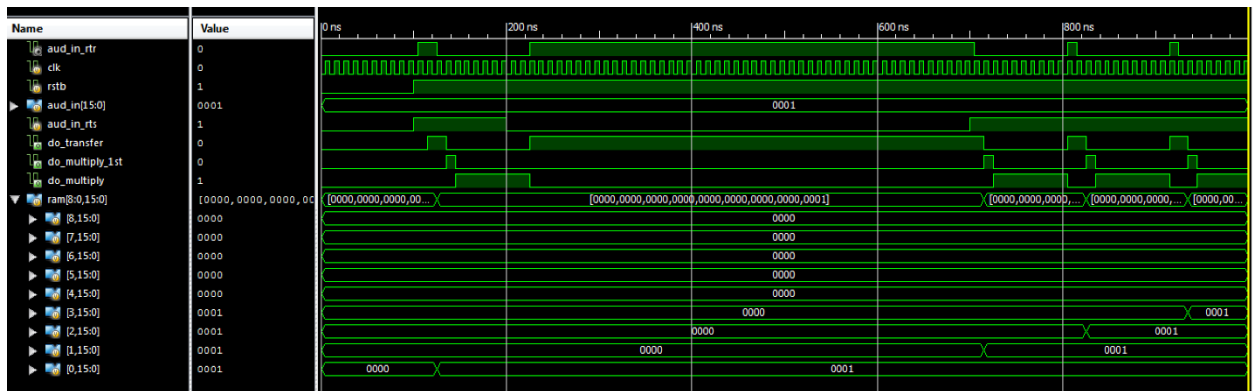


Fig 3.14: Simulation of Filter Test 3

The next test I performed was to observe the behavior of the read pointer for both the mux and storage module. I am expecting the read pointer for the storage module to decrement and increment for the mux. Each cycle should have 8 (or the number of Taps) increments or decrements.

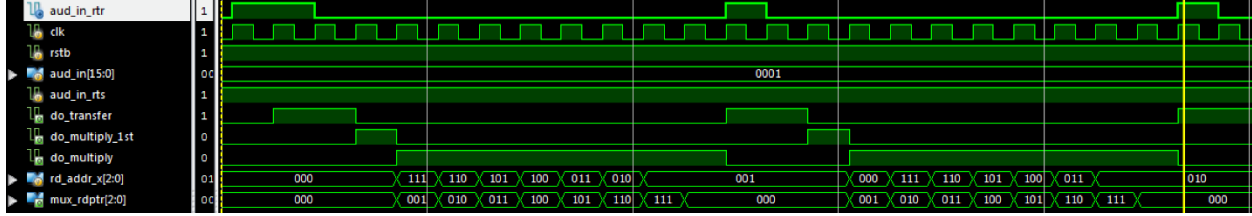


Fig 3.15: Simulation of Filter Test 4

Note that the last clock cycle of the multiply state is superfluous for the current cycle. Also for the read pointer for the storage module (*rd_addr_x*), it starts at the current location of the writer pointer. This proves that the filter follows, which is the convolution process.

- 1) $y[0] = h[0] x[0] + h[1] x[-1] + \dots + h[510] x[-510] + h[511] x[-511]$
- 2) $y[1] = h[0] x[1] + h[1] x[0] + \dots + h[510] x[-509] + h[511] x[-510]$
- 3) $y[2] = h[0] x[2] + h[1] x[1] + \dots + h[510] x[-508] + h[511] x[-509]$

The next test, was functionality of the filter. Here are the following testing parameters:

$$\begin{aligned}
 rf_coeff0_b / rf_coeff0_a &= 16'h0001 \\
 rf_coeff1_b / rf_coeff1_a &= 16'h0002 \\
 rf_coeff2_b / rf_coeff2_a &= 16'h0003 \\
 rf_coeff3_b / rf_coeff3_a &= 16'h0004 \\
 rf_coeff4_b / rf_coeff4_a &= 16'h0005 \\
 rf_coeff5_b / rf_coeff5_a &= 16'h0006 \\
 rf_coeff6_b / rf_coeff6_a &= 16'h0007 \\
 rf_coeff7_b / rf_coeff7_a &= 16'h0008
 \end{aligned}$$

The audio values vary from $32'h00000000$ to $32'h00FF00FF$, again the 16 most MSB are the left audio and 16 most LSB are the right. We are assuming that the I2S is always ready to send. In addition, I am only clipping and not shifting currently. The state machine outputs a 40bit audio out signal (due to the accumulator), so at the filter.v level the audio out is defined by:

assign aud_out = convo_signal[39:8];

Below is the simulation result:

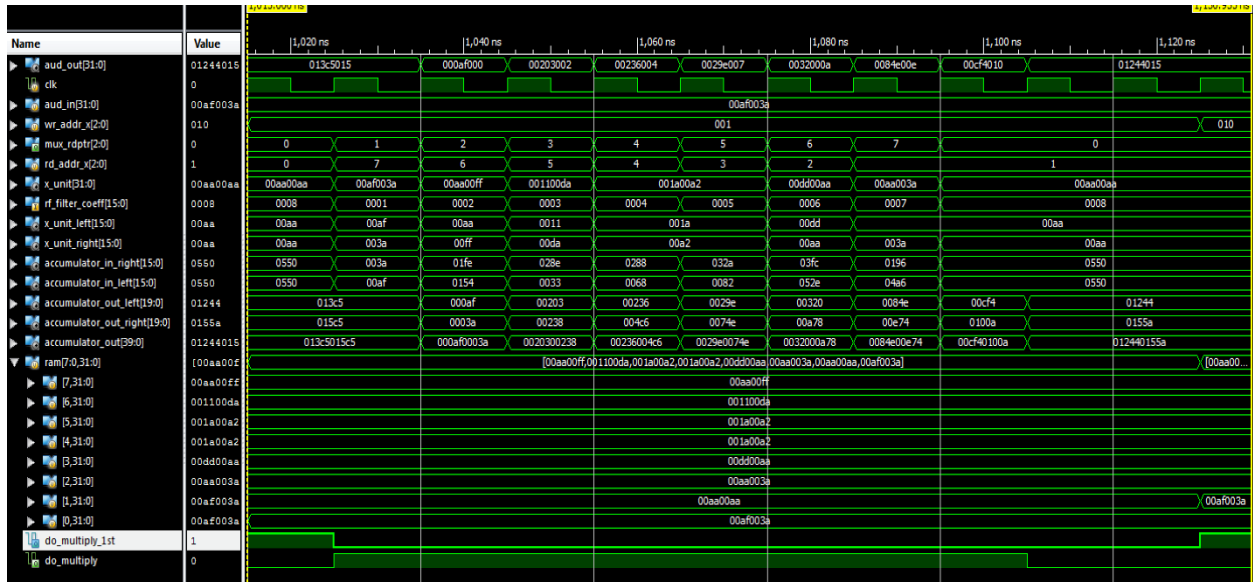


Fig 3.16: Simulation of Filter Test 5

To check our simulation, I am going to check the results by hand. Note: all values in Hex

Here is the input signal and impulse response:

$$\begin{aligned}
 x[n] &= \{00af003a, 00aa00aa, 00aa003a, 00dd00aa, 001a00a2, 001a00a2, 001100da, 00aa00ff\} \\
 xleft[n] &= \{00af, 00aa, 00aa, 00dd, 001a, 001a, 0011, 00aa\} \\
 xright[n] &= \{003a, 00aa, 003a, 00aa, 00a2, 00a2, 00da, 00ff\} \\
 h[n] &= \{0001, 0002, 0003, 0004, 0005, 0006, 0007, 0008\}
 \end{aligned}$$

The sum of products is defined by:

Note: I am using hex values for the index values

$$\begin{aligned}
 y[0] &= x[0]h[0] + x[7]h[1] + x[6]h[2] + x[5]h[3] + x[4]h[4] + x[3]h[5] + x[2]h[6] + \\
 &\quad x[1]h[7]
 \end{aligned}$$

There are two multipliers and accumulators for left and right audio. The calculations, done with WolframAlpha.com, are the following:

Left:

$$\begin{aligned}
 yLeft &= 00af*0001 + 00aa*0002 + 0011*0003 + 001a*0004 + 001a*0005 + 00dd*0006 \\
 &\quad + 00aa*0007 + 00aa*0008 \\
 yLeft &= 0x01244
 \end{aligned}$$

Right:

$$\begin{aligned}
 yRight &= 003a*0001 + 00ff*0002 + 00da*0003 + 00a2*0004 + 00a2*0005 + 00aa*0006 \\
 &\quad + 003a*0007 + 00aa*0008 \\
 yRight &= 0x0155a
 \end{aligned}$$

If I concatenate yLeft and yRight, I get $y = 0x012440155a$. Then we clip the least significant bit to get a 32 bit value of $0x01244015$.

This matches what we get in our simulation.

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 is flagged when the filter is producing data at a lower rate than the I2S Output is consuming.

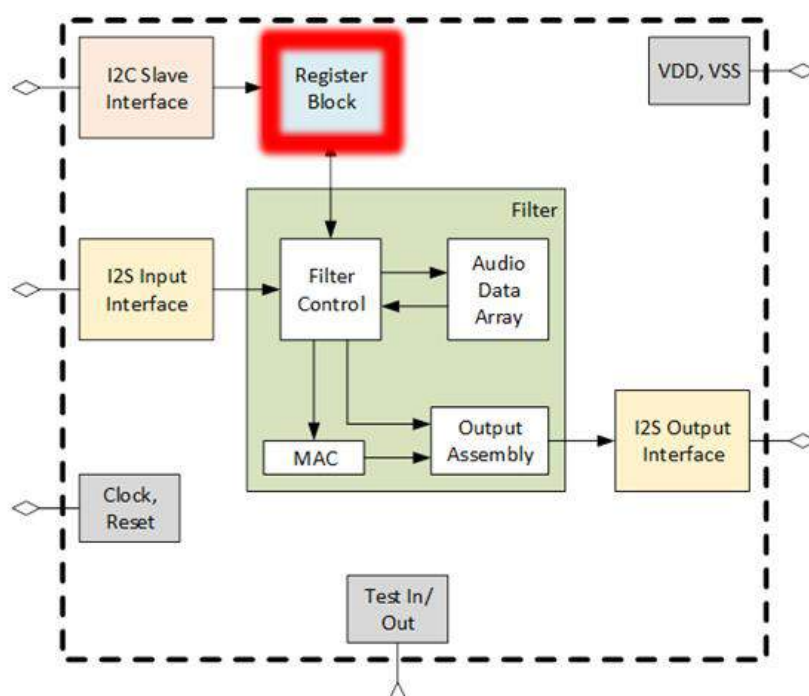


Fig 4.1: Chip Diagram with Register highlighted

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.

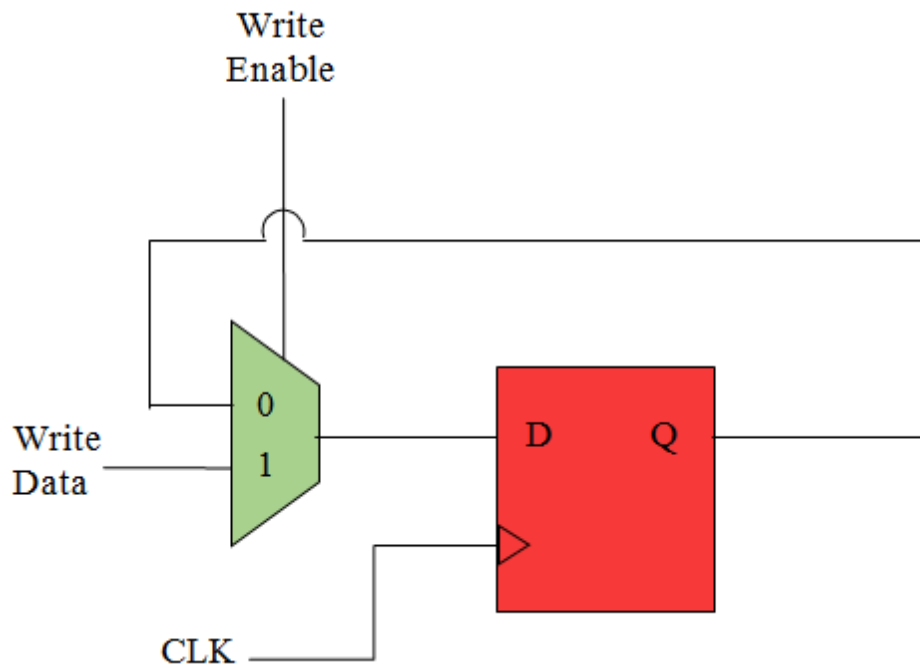


Fig 4.2: Internal Micro-Architecture

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 11-bit variable that holds the value of the address in the register block in which data can be read from or written to.

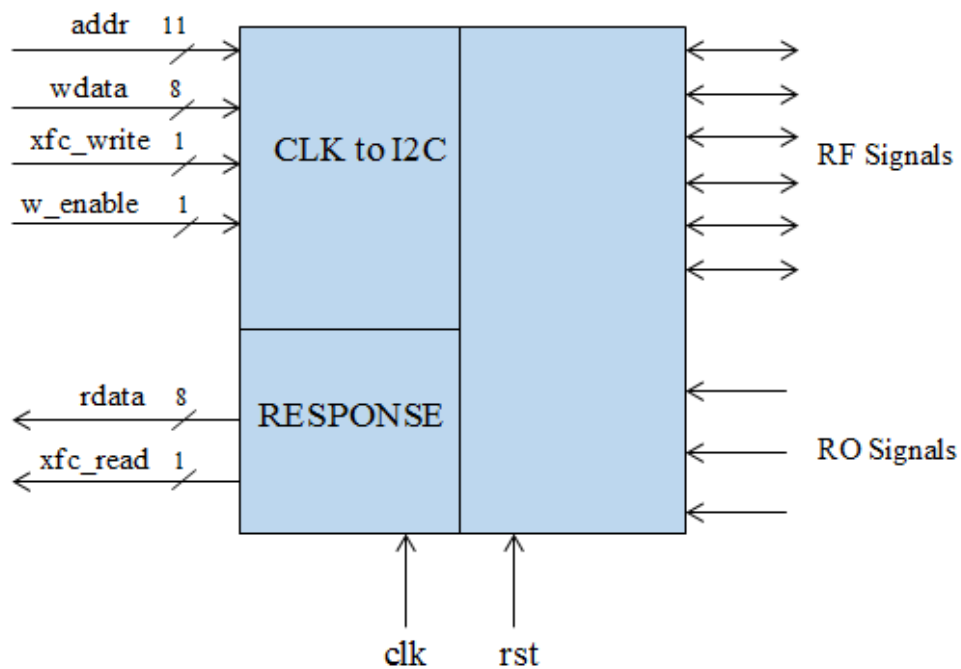


Fig 4.3: Top Level Interface Design of reg.v

The wdata is an input variable array that writes 8-bits of data to a specified 11-bit address. Similarly, the rdata output array returns the 8-bits of data demanded 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

Address	Register Name		Field Name	Bits	Description	RO/ WO/ RW	Default Value
0x004		1:1	rf_i2si_bist_en	1:1	0- audio source is i2si. 1- audio source is BIST	RW	0x1
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	15:1 2	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_overflow	0:0	fifo overflow clear	WO	NA
0x00C		1:1	ro_fifo_overflow	1:1	input audio fifo overflow	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

Address	Register Name		Field Name	Bits	Description	RO/ WO/ RW	Default Value
0x012		3:0	rf_i2si_bist_start_val _b				
0x012		7:4	rf_i2si_bist_upper_li mit_a	31:2 0	upper limit of the sawtooth wave	RW	0x7FF
0x013		7:0	rf_i2si_bist_upper_li mit_b				
0x014	I2C_REG_INDIR_ADDR						
0x014		7:0	rf_i2c_reg_indir_add r_a	11:0	address register used for indirect addressing via i2c	RW	0
0x015		3:0	rf_i2c_reg_indir_add r_b		address register used for indirect addressing via i2c	RW	0
0x0400	FILT_COEFF S_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	31:1 6	Filter Coefficient 1	RW	0x0
0x0403		7:0	rf_filter_coeff1_b		Filter Coefficient 1	RW	0x0
0x0404	FILT_COEFF S_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	31:1 6	Filter Coefficient 3	RW	0x0
0x0407		7:0	rf_filter_coeff3_b		Filter Coefficient 3	RW	0x0
0x7FC	FILT_COEFF S_510_511						

Address	Register Name		Field Name	Bits	Description	RO/ WO/ RW	Default Value
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	31:1 6	Filter Coefficient 511	RW	0x0
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 11-bit hexadecimal number 0x00C and the xfc_write 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 first initialized to zero 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 default values given in Table 4.1. When the write enable, w_enable, and the file transfer, 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. A second always block includes another case-statement that given a specified address, allows the stored data to be read. The register.v allows data to be written to an address that correlate to specific signal(s), and allows data to be read from signal(s) that are stored in a specific address.

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 11-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.

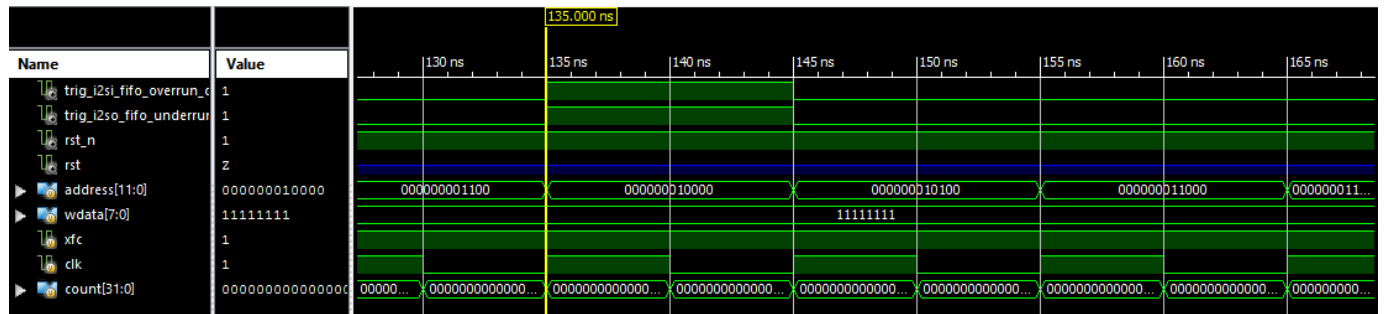


Fig 4.4: The start of the Test Fixture Simulation

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 become 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.

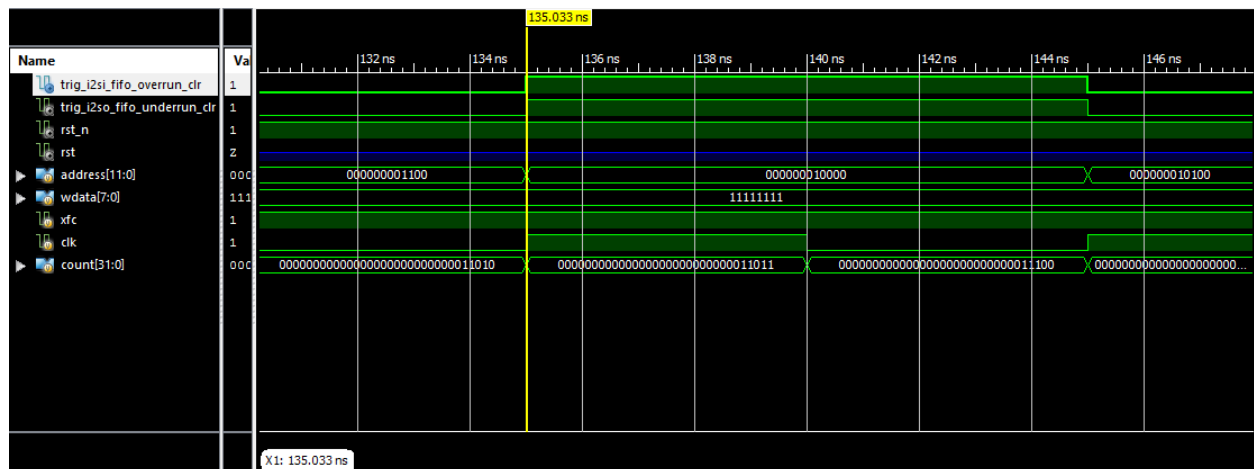


Fig 4.5: Bits trig_i2si_fifo_overrun_clr and trig_i2so_fifo_underrun_clr are Triggered

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 11-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.

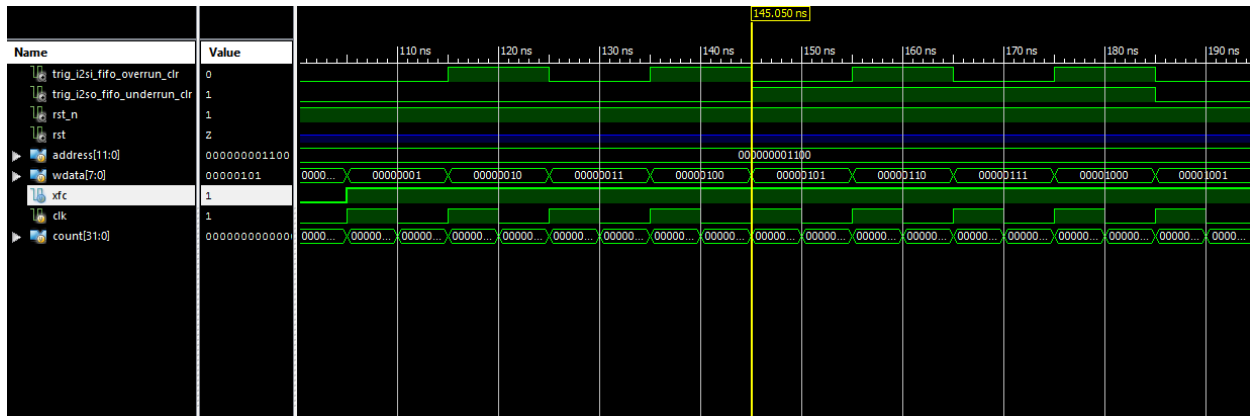


Fig 4.6: Bits trig_i2si_fifo_overrun_clr and trig_i2so_fifo_underrun_clr are Triggered in Second Testbench

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 Xilinx, 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 11-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 Xilinx directly.

4.6.3 set_read_coeffs.c

The set_read_coeffs.c program generates the correct formatting for the case-statement that varies slightly from set_coeffs.c. The fprintf() statement is rearranged from set_coeffs.c to have rdata point to the signal requesting to be read, instead of having the signal point to wdata. The 1024 filter coefficients are created that same way as set_coeffs.c, but printed differently to the text file such that the filter block is able to read the filter coefficients values that are written in the first always block.

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 its 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 function 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 5 MHz from the original 100 kHz speed.

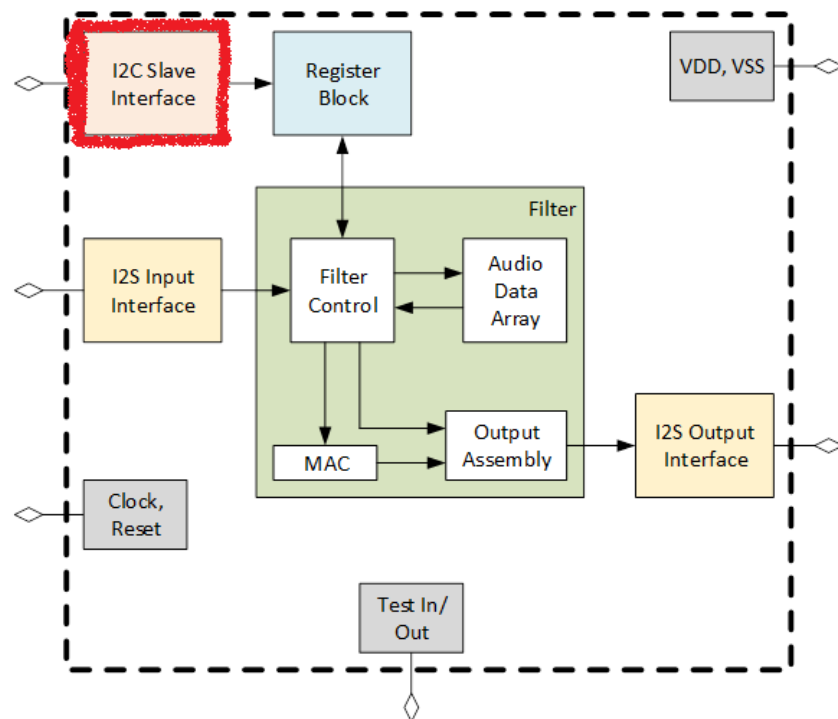


Fig 5.1: I2C Chip Diagram Highlight

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 and 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 8 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.

$$\begin{aligned} 512 \text{ registers} * 9 \frac{\text{bits}}{\text{transition}} &= 4608 \text{ bits total} \\ \frac{4608 \text{ bits}}{400 \text{ kbits/second}} &= 0.01152 \text{ seconds} \end{aligned}$$

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 a 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 serving 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 SCL and SDA lines which stand for serial clock and serial data respectively. Both of these single line wires are active high signals using pull-up resistors and the transitions of the signals are driven low by the transmitter during the transaction. The i2c_SCL input is only an 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.

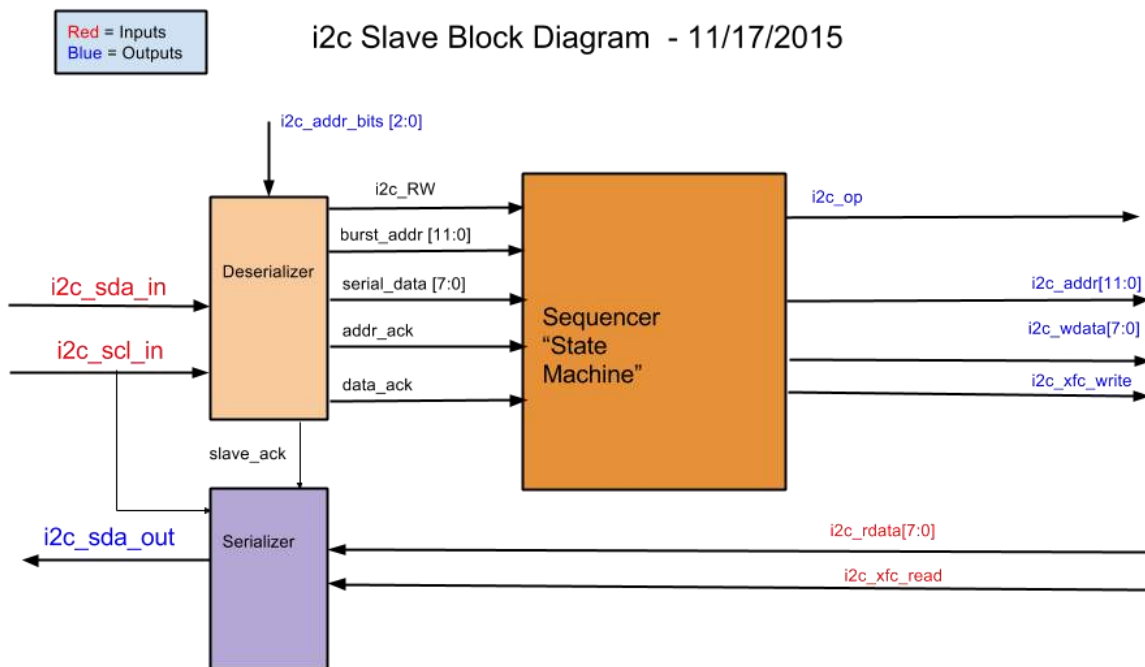


Fig 5.3: I²C block diagram in

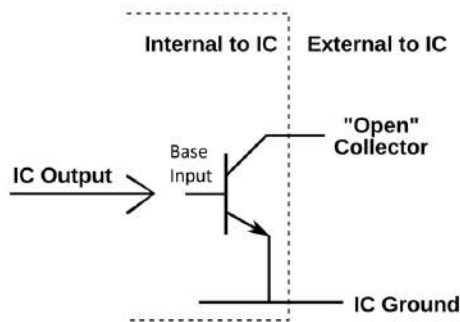


Fig 5.4: Open Drain Schematic

5.3.2 Register Block

Interface with the register block in contrast with the I²C serial interface is much simpler. The interface between blocks is parallel where all bits are sent at once with a distinct wired connection for each bit and utilizing a single clock cycle strobe as a transfer signal rather than a fully handshake interface reduces wiring and complexity.

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 read 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 is `i2c_rdata` which is 1 byte wide and holds the data from the requested read address, and the second is `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 user definable external address pins 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 package or a selected pin on an FPGA to be driven high and therefore would result in 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 pins without any delay as it was deemed unnecessary to clock this logic.

5.4 Register Mapping

The interconnections of the I²C block that do not interface outside 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 the table 5.1 register map as well as the control signals of the block diagram in figure 5.3.

Table 5.1: I²C Register table

Signal Name	Direction	Bits	Comment
clk	in	1	clock
rst	in	1	reset
i2c_scl	in	1	serial clock
i2c_sda_in	in	1	external pin combines in and out using open drain
i2c_sda_out	out	1	external pin combines in and out using open drain
i2c_op	out	1	1: write 0: read
i2c_addr	out	12	register address
i2c_wdata	out	8	data to be written for a write op
i2c_xfc_write	out	1	strobe transfer enable
i2c_rdata	in	8	read data
i2c_xfc_read	in	1	strobe transfer enable
i2c_addr_bits	in	3	External Pins, 3 LSB of slave address

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 signal.

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

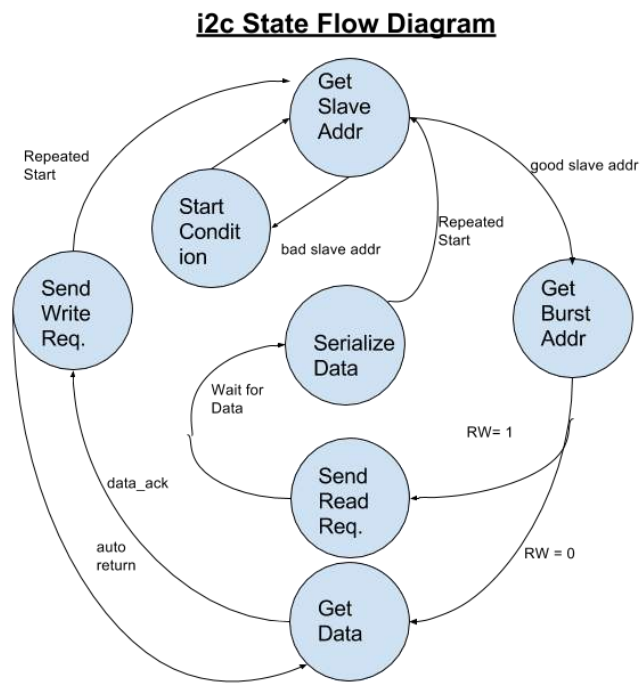
interface as the slave address call from the master and check it against the actual slave address. If the address is not the same then the system stays idle looking for another start condition, but if it is a match then the state changes to look for the burst start address after sending a pulse on the 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 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 received 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 line between the deserializer and the sequencer. Every time that the data_ack is pulsed, then the sequencer updates the register address and the data and when it is ready it strobes i2c_xfc_write for one chip clock cycle 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 repeated start condition to occur after the slave acknowledgement in which case the slave looks for a slave address again. It is possible that in the event of a stream write data coming in that it will in fact be more than what the register block has allocated as valid address space. During future testing the team will decide if this functionality will be necessary as a control for invalid data and addresses. This would enlarge the size of I²C block and its complexity as there would be a need for an entire register block map to be coded into the sequencer block. This itself could cause unwanted operation so it will have to be decided as a group during testing to see if it is necessary or not. 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. With the structure described above the deserializer block is the state machine for the entire I²C block as it drives the state of the other blocks depending on what the master controller is requesting.



*A return to the start state can happen during any part of the cycle if a stop or reset condition is seen

Fig 5.5: I²C State Machine Flow Diagram

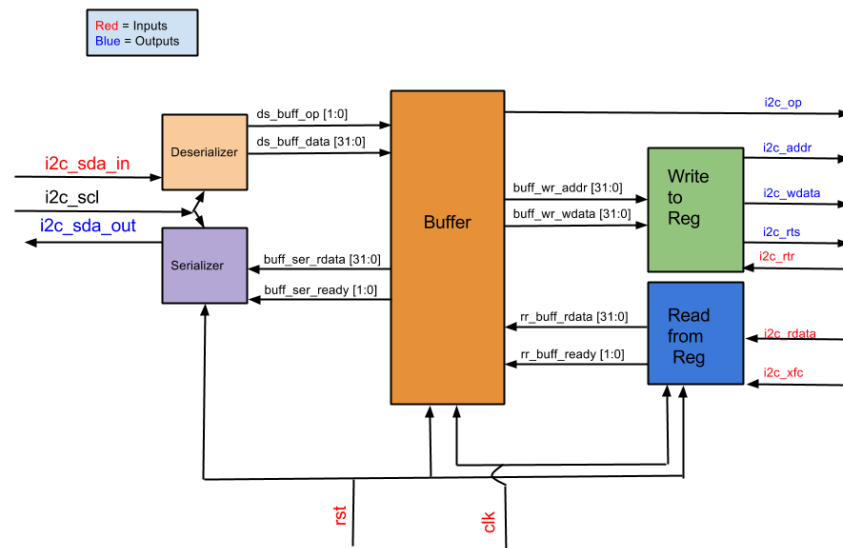


Fig 5.6: I²C Previous block diagram versions

5.5 Sub Blocks

The I²C block was broken up into 3 distinct parts including the deserializer, the sequencer and the serializer. The 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 sub-blocks 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 functions beyond simply deserializing the data coming from the master.

The deserializer must filter and condition the `isc_SCL` and the `i2c_SDA_in` signals in order to make them usable in the Verilog code. Edge detection pulses were extracted from each line after it went through the D flip flop ranking described above. States of the lines 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 are 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 statements become active. This allowed for the 8th case statement to include control signals to acknowledge, state change, and transfer data to different registers. Although 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` line. Once this happens, a state change occurs and an acknowledgement signal sent to the serializer if 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 to form the 12-bit register burst_addr signal. Once this occurs, 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 received.

Depending on 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.

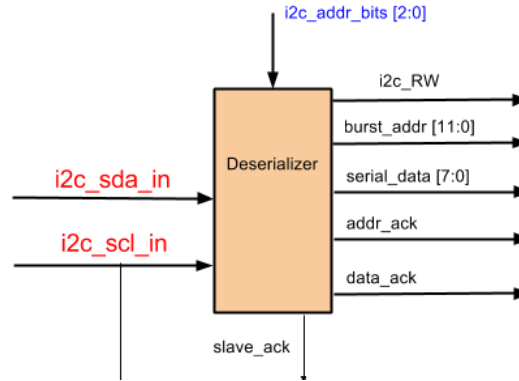


Fig 5.7: I²C Deserializer Block Diagram

5.5.2 Sequencer

The sequencer block acts as slave of sorts to the deserializer block. Its sole purpose is to transfer data, address and operation code information to the register block. Depending on 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 during a read and the second during a write.

Upon a read opcode the sequencer simply captures the `burst_addr` into the `i2c_addr` output line and then strobes the `i2c_xfc` output line, 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 opcode 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. The opcode acts as a control object for how the I²C block interacts with the register block determining a read or write transaction.

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 is received 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 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.

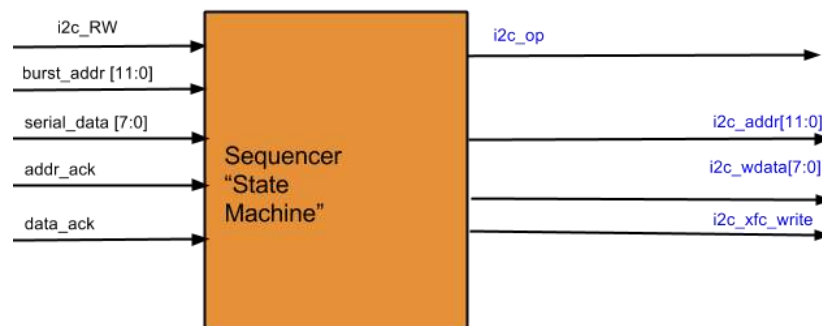


Fig 5.8: I²C Sequencer Block Diagram

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 to 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, 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 states.

Start and stop conditions are handled by using the slave_ack and i2c_xfc_read signals. If either one of these are asserted, 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 a case statement that will then write to the i2c_SDA_out line to serialize the data using the captured i2c_rdata 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 one 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.

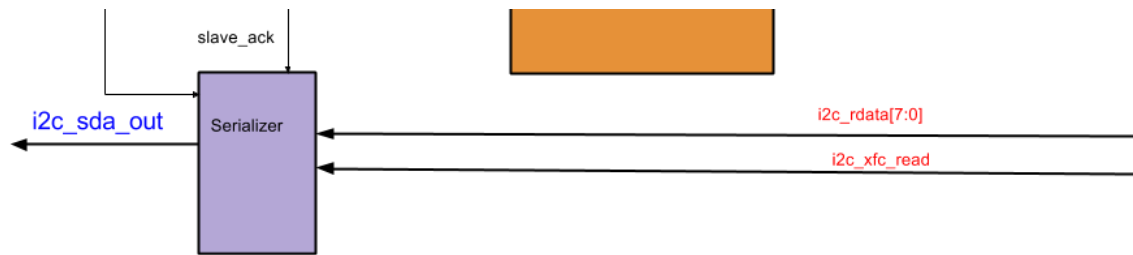


Fig 5.9: I²C Serializer Block Diagram

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 in over the SCL wire for the address bits were extracted successfully and that the correct state change and acknowledge signals were sent to the serializer.

Once the address serialization test was performed, a test was performed with both opcodes for read and write functionality to see if the correct state change would occur in the deserializer. The deserializer did correctly revert back to an idle state after a read request and the address had been transferred to the sequencer. The deserializer block also changed correctly to data fetch mode when a write was 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 line.

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 long.

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 controlling the frequency of the SCL line. During an 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. 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 A. 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 is 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 left over 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 University. The schedule was created in Microsoft Project 2013 in the form of a Gantt chart and is included in Appendix A. 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 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 Gantt 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. Overall, we recognize that we are 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 an 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. Many parts of the filter block have been coded and tested and the only thing that needs to be done is additional testing. 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: Team Management

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

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 an 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.

ID	Task Name	% Complete	August 1 8/2	8/16	September 1 8/30	9/13	October 1 9/27	10/11	November 1 10/25	11/8	Dec 11/22
1	System Design	100%									
2	General Planning	100%									
3	Complete I2S In Document	100%									
4	Complete I2S Out Document	100%									
5	Complete Filter Document	100%									
6	Complete I2C Document	100%									
7	Complete Register Document	100%									
8	Finish System Design	100%									
9	Document System in CORE	100%									
10	RTS/RTR and XFC Protocols	100%									
11	Create a Detailed Schedule	100%									
12	Install Microsoft Project	100%									
13	Install EDA Tools	50%									
14	Determine EDA Requirements	100%									
15	Install OS	69%									
16	Install EDA Tool	0%									
17	Create a Simple Test Case	0%									
18	Place and Route Test	0%									
19	Finish EDA Installation	0%									
20	Create Web Page	100%									
21	RTL Design and Testing	70%									

Task

Inactive Summary

Manual Task

Duration-only

Manual Summary Rollup

Manual Summary

Start-only

Finish-only

Split

Milestone

Summary

Project Summary

Inactive Task

Inactive Milestone

External Tasks

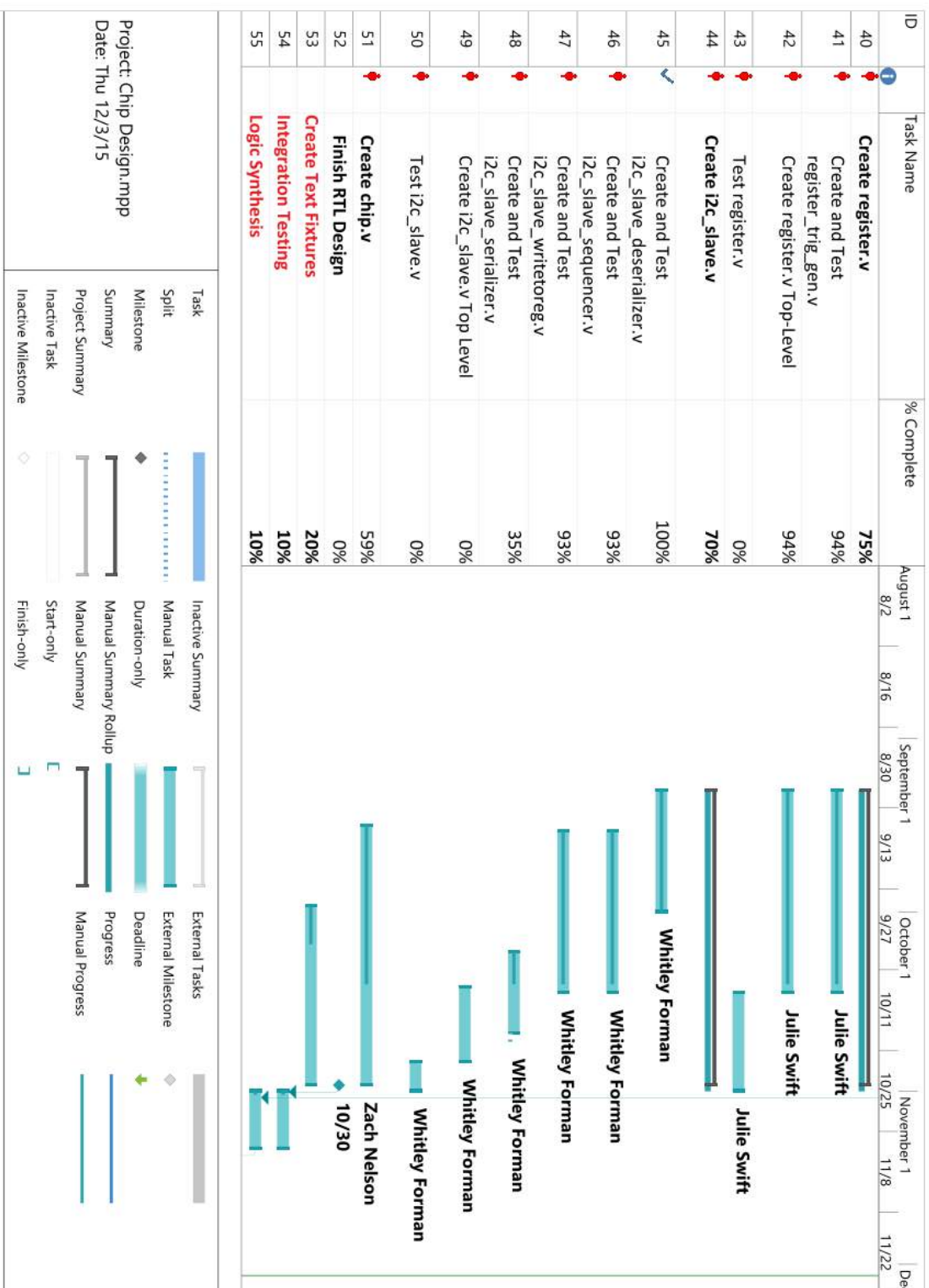
External Milestone

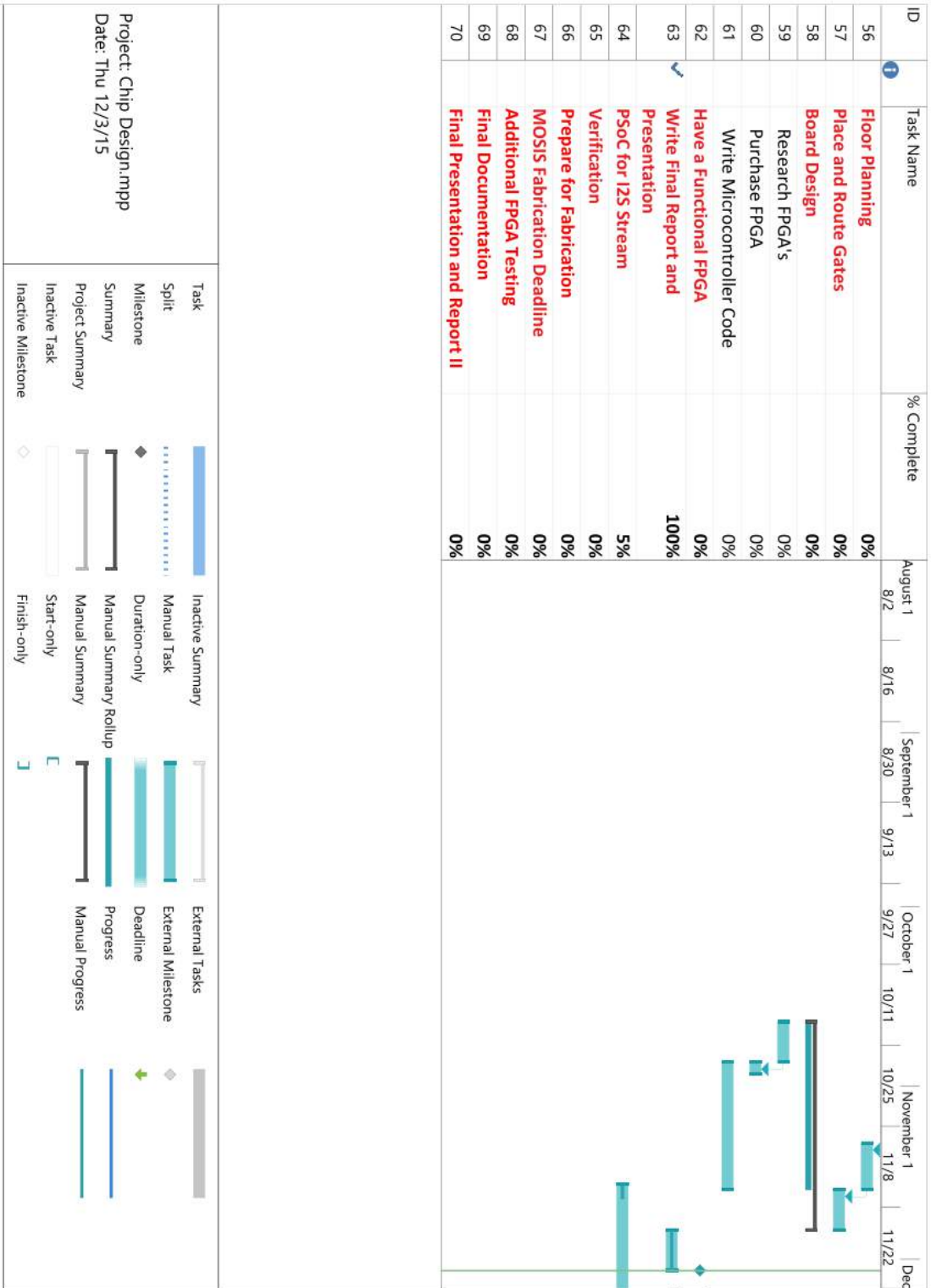
Deadline

Progress

Manual Progress

Project: Chip Design.mpp
Date: Thu 12/3/15





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_overflow_rst bit
 - Example:
if (ro_fifo_overflow) → happens when rts=1 and rtr=0
 ro_fifo_overflow = 1
else if (trig_i2sin_fifo_overflow_rst)
 ro_fifo_overflow = 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

September 30th Senior Project Meeting

Armstrong Hall 137: 3:30-3:30

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 (Dhruvit)
- 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 B: Verilog Modules

1. i2s_in.v
 - 1.1. i2s_in.v
 - 1.2. synchronizer.v
 - 1.3. i2si_deserializer.v
 - 1.4. i2si_bist_gen.v
 - 1.5. i2si_mux.v
 - 1.6. fifo.v
2. i2s_out.v
 - 2.1. i2s_out.v
 - 2.2. serializer.v
3. filter.v
 - 3.1. filter.v
 - 3.2. filter_stm.v
 - 3.3. filter_storage.v
 - 3.4. filter_mux.v
 - 3.5. filter_accumulator.v
 - 3.6. filter_barrel_shifter.v
4. register.v
 - 4.1. trig_generator.v
 - 4.2. register.v
 - 4.3. initialize_coeffs.c
 - 4.4. set_coeffs.c
 - 4.5. set_read_coeffs.c
5. i2c_slave.v
 - 5.1 deserializer.v
 - 5.2 sequencer.v
 - 5.3 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_overnrun,
    trig_i2si_fifo_overnrun_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_overnrun;    //FIFO: Input audio FIFO
overnrun - 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_overnrun_clr; //Signal to reset
ro_fifo_overnrun 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;
not gate                             //Wire connecting fifo_rtr to

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),
    and not just sck_transition?    //Remove? need to out sck too
    .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_overflow <= 0;
    else if (~fifo_rtr | deserializer_xfc)
        ro_fifo_overflow <= 1;
    else if (trig_i2si_fifo_overflow_clr)
        ro_fifo_overflow <= 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
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,i2s
i_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
    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
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
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),
    .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
    end

```

```
        end  
    end  
endmodule
```

3. filter.v:

3.1 filter.v

```
`timescale 1ns / 1ps
module filter(clk, rstb, aud_in, aud_in_rts, aud_in_rtr, aud_out, aud_out_rts,aud_out_rtr,
rf_filter_shift, rf_filter_clip_en,
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_coeff509_b,rf_filter_coeff510_a,
rf_filter_coeff510_b,rf_filter_coeff511_a, rf_filter_coeff511_b
);
input  clk;
input  rstb;
input  [31:0] aud_in;
input  aud_in_rts;
output aud_in_rtr;
output [31:0] aud_out;
output aud_out_rts;
input  aud_out_rtr;
input  rf_filter_shift;
input  rf_filter_clip_en;
input [7:0] rf_filter_coeff0_a, rf_filter_coeff0_b,
rf_filter_coeff1_a, rf_filter_coeff1_b,rf_filter_coeff2_a,
.....
rf_filter_coeff509_a, rf_filter_coeff509_b,rf_filter_coeff510_a,
rf_filter_coeff510_b,rf_filter_coeff511_a, rf_filter_coeff511_b;
wire          do_transfer;
wire          do_multiply_1st;
wire          do_multiply;
wire[15:0]    rf_filter_coeff;
wire[8:0]     mux_rdptra;
wire         mux_re;
wire         output_signal;
wire[39:0]    convo_signal;
assign aud_out = convo_signal[39:8];
//*****
filter_stm filter_stm_0 (
    .clk(clk),
    .rstb(rstb),
    .filter_aud_in_rts(aud_in_rts),
    .filter_aud_in_rtr(aud_in_rtr),
    .do_transfer(do_transfer),
    .do_multiply_1st(do_multiply_1st),
    .do_multiply(do_multiply),
    .filter_aud_in(aud_in),
    .filter_aud_out(convo_signal),
    .rf_filter_coeff(rf_filter_coeff),
```

```

        .mux_re(mux_re),
        .mux_rdptra(mux_rdptra)
    );
//*****
/*
filter_barrel_shifter filter_barrel_shifter_0
    (.input_signal      (convo_signal),
     .sel_shift         (sel_shift),
     .output_signal     (output_signal));
*/
//*****
filter_mux filter_mux_0 (
    .clk(clk),
    .rden(mux_re),
    .rdptr(mux_rdptra),
    .rddata(rf_filter_coeff),
    .rf_filter_coeff0_a(rf_filter_coeff0_a),
    .rf_filter_coeff0_b(rf_filter_coeff0_b),
    .rf_filter_coeff1_a(rf_filter_coeff1_a),
    .rf_filter_coeff1_b(rf_filter_coeff1_b),
    .....
    .rf_filter_coeff510_b(rf_filter_coeff510_b),
    .rf_filter_coeff511_a(rf_filter_coeff511_a),
    .rf_filter_coeff511_b(rf_filter_coeff511_b)
);
//*****
endmodule

```

3.2 filter_stm.v

```
`timescale 1ns / 1ps

module filter_stm( clk, rstb, filter_aud_in_rts, filter_aud_in_rtr, do_transfer, do_multiply_1st,
do_multiply, filter_aud_in, filter_aud_out, rf_filter_coeff, mux_re, mux_rdptr
);

input          clk;                                //Clock for State Machine
input          rstb;                                //Active -low reset signal
input          filter_aud_in_rts;                    //Ready to Send
input          [31:0] filter_aud_in;
input          [15:0] rf_filter_coeff;
output         filter_aud_in_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         mux_re;
output         [8:0] mux_rdptr;
output         [39:0] filter_aud_out;

//*****
localparam      I      DLE      = 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      = 9,
                  WIDTH   = 16,
                  TAPS     = 512;
//*****

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    accumulator_enable, accumulator_enable_nxt;
reg    accumulator_load, accumulator_load_nxt;
//*****
reg    filter_init, filter_init_nxt;
reg    filter_need_new, filter_need_new_nxt;
reg[PTR-1:0]filter_count, filter_count_nxt;
```



```

reg      filter_aud_in_rtr, filter_aud_in_rtr_nxt;
//*****
reg      [PTR-1:0]wr_addr_x, wr_addr_x_nxt;
reg      [PTR-1:0]rd_addr_x, rd_addr_x_nxt;
reg      arr_re_x, arr_re_x_nxt;
reg      arr_we_x, arr_we_x_nxt;
//*****
reg      [PTR-1:0]mux_rdp_ptr, mux_rdp_ptr_nxt;
reg      mux_re, mux_re_nxt;
//*****
wire      filter_xfc_in;
wire [31:0]  x_unit;
wire [15:0]  x_unit_left; // Left is 31:16
wire [15:0]  x_unit_right; // Right is 15:0
//*****
wire [31:0]  accumulator_in_left;
wire [39:0]  accumulator_out_left;
wire [31:0]  accumulator_in_right;
wire [39:0]  accumulator_out_right;
wire [39:0]  accumulator_out;
//*****

assign x_unit_left      = x_unit[31:16];
assign x_unit_right     = x_unit[15:0];
assign filter_xfc_in    = filter_aud_in_rtr && filter_aud_in_rts;
assign accumulator_out  = {accumulator_out_left, accumulator_out_right};
assign filter_aud_out   = accumulator_out ;
assign accumulator_in_left  = x_unit_left * rf_filter_coeff;
assign accumulator_in_right = x_unit_right * rf_filter_coeff;

//*****

always@(*)
begin

    filter_state_nxt      = filter_state;
    do_transfer_nxt       = 1'b0;
    do_multiply_1st_nxt   = 1'b0;
    do_multiply_nxt       = 1'b0;
//*****
    wr_addr_x_nxt         = wr_addr_x;
    rd_addr_x_nxt         = rd_addr_x;
    arr_re_x_nxt          = arr_re_x;
    arr_we_x_nxt          = arr_we_x;
//*****
    mux_rdp_ptr_nxt       = mux_rdp_ptr;
    mux_re_nxt            = mux_re;
//*****

```

```

        multiply_1st_check_nxt      = multiply_1st_check;
        filter_running_1st_nxt     = filter_running_1st;
        filter_running_nxt         = filter_running;
        filter_need_new_nxt        = filter_need_new;
        filter_count_nxt           = filter_count;
//*****
        filter_aud_in_rtr_nxt      = filter_aud_in_rtr;
        accumulator_enable_nxt     = accumulator_enable;
        accumulator_load_nxt       = accumulator_load;
        case(1'b1)
//*****
// IDLE STATE
//*****
        filter_state[IDLE_ID]:begin
            if(filter_xfc_in)
                begin
                    filter_state_nxt      = TRANSFER;
                    do_transfer_nxt       = 1'b1;
                    arr_we_x_nxt          = 1'b1;
                end
            else
                begin
                    filter_aud_in_rtr_nxt = 1'b1;
                end
            end
        end
//*****
// TRANSFER STATE
//*****
        filter_state[TRANSFER_ID]:begin
            if(filter_running_1st)
                begin
                    filter_state_nxt      = MULTIPLY_1ST;
                    do_multiply_1st_nxt   = 1'b1;
                    do_transfer_nxt       = 1'b0;
                    filter_running_1st_nxt = 1'b0;
                    filter_running_nxt    = 1'b1;
                    arr_re_x_nxt          = 1'b1;
                    mux_re_nxt            = 1'b1;
                    arr_we_x_nxt          = 1'b0;
                    wr_addr_x_nxt         = wr_addr_x + 1'b1;
                end
            else
                begin
                    do_transfer_nxt       = 1'b1;
                    if(filter_xfc_in)
                        begin
                            filter_aud_in_rtr_nxt = 1'b0;

```

```

arr_we_x_nxt          = 1'b1;
filter_running_1st_nxt = 1'b1;
end
end
end

//*****
// MULTIPLY STATE 1ST
//*****

filter_state[MULTIPLY_1ST_ID]:begin

    if(filter_running)
        begin
            filter_state_nxt          = MULTIPLY;
            do_multiply_1st_nxt        = 1'b0;
            do_multiply_nxt            = 1'b1;
            filter_running_nxt         = 1'b0;
            accumulator_load_nxt       = 1'b1;
            accumulator_enable_nxt     = 1'b1;
            rd_addr_x_nxt              = rd_addr_x - 1'b1;
            mux_rdp_ptr_nxt            = mux_rdp_ptr + 1'b1;
            filter_count_nxt           = filter_count + 1'b1;
        end
    else
        begin

        end
    end
end

//*****
// MULTIPLY STATE
//*****

filter_state[MULTIPLY_ID]:begin
    if(filter_need_new)
        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;
            filter_aud_in_rtr_nxt      = 1'b1;
            accumulator_enable_nxt     = 1'b0;
        end
    else
        begin

```

```

do_multiply_nxt      = 1'b1;
rd_addr_x_nxt        = rd_addr_x - 1'b1;
mux_rdp_ptr_nxt      = mux_rdp_ptr + 1'b1;
filter_count_nxt     = filter_count + 1'b1;
accumulator_load_nxt = 1'b0;
if (filter_count == TAPS-1)
    begin
        rd_addr_x_nxt      = rd_addr_x ;
        filter_need_new_nxt = 1'b1;
        arr_re_x_nxt       = 1'b0;
        mux_re_nxt         = 1'b0;
    end
end

end

default: begin end
endcase

end

//*****
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;
            arr_re_x           <= 1'b0;
            arr_we_x           <= 1'b0;
            mux_rdp_ptr        <= 1'b0;
            mux_re             <= 1'b0;
            filter_running     <= 1'b0;
            filter_running_1st <= 1'b0;
            filter_need_new    <= 1'b0;
            filter_count       <= 1'b0;
            filter_aud_in_rtr  <= 1'b0;
            accumulator_enable <= 1'b0;
            accumulator_load   <= 1'b0;
            multiply_1st_check <= 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;
        arr_we_x           <= arr_we_x_nxt;
        mux_rdp_ptr        <= mux_rdp_ptr_nxt;
        mux_re             <= mux_re_nxt;
        multiply_1st_check <= multiply_1st_check_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;
        filter_aud_in_rtr  <= filter_aud_in_rtr_nxt;
        accumulator_enable <= accumulator_enable_nxt;
        accumulator_load   <= accumulator_load_nxt;
    end
end
filter_storage filter_storage_x
    (.clk          (clk),
     .rstb         (rstb),
     .wren         (arr_we_x),
     .wr_ptr       (wr_addr_x),
     .wr_data      (filter_aud_in),
     .rden         (arr_re_x),
     .rd_ptr       (rd_addr_x),
     .rd_data      (x_unit));

filter_accumulator filter_accumulator_left
    (.clk          (clk),
     .rstb         (rstb),
     .enable       (accumulator_enable),
     .load         (accumulator_load),
     .D            (accumulator_in_left),
     .Q            (accumulator_out_left));
filter_accumulator filter_accumulator_right
    (.clk          (clk),
     .rstb         (rstb),
     .enable       (accumulator_enable),
     .load         (accumulator_load),
     .D            (accumulator_in_right),
     .Q            (accumulator_out_right));

endmodule

```

3.3 *filter_storage.v*

```
`timescale 1ns / 1ps
module filter_storage
( clk, rstb, wren, wrptr, wrdata, rden, rdptr, rddata);
input      clk;
input      rstb;
input      wren;
input  [8:0] wrptr;
input  [31:0] wrdata;
input      rden;
input  [8:0] rdptr;
output [31:0] rddata;

localparam DEPTH = 511; //2^9 <= 16'b0; 512
localparam WIDTH = 31;

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

always @(posedge clk or negedge rstb)
begin
    if (!rstb)
    begin
        ram[0] <= 16'b0;
        ram[1] <= 16'b0;
        .....
        ram[509] <= 16'b0;
        ram[510] <= 16'b0;
        ram[511] <= 16'b0;
    end
    else if(wren)
        ram[wrptr] <= wrdata;
    end

always @(posedge clk)
begin
    if(rden)
        rddata <= ram[rdptr];
    end
endmodule
```

3.4 filter_mux.v

```
`timescale 1ns / 1ps
module filter_mux(clk, rden, rdptr, rddata, rf_filter_coeff0_a,
rf_filter_coeff0_b, rf_filter_coeff1_a, .....
input [7:0] rf_filter_coeff0_a, rf_filter_coeff0_b,
rf_filter_coeff1_a, rf_filter_coeff1_b, rf_filter_coeff2_a, rf_filter_coeff2_b, .....
input      clk;
input      rden;
input  [8:0] rdptr;
output [15:0] rddata;

localparam DEPTH = 511; //2^9 = 512
localparam WIDTH = 15;

wire  [WIDTH:0] ram [DEPTH:0];
reg    [WIDTH:0] rddata;
assign ram[0] = {rf_filter_coeff0_b, rf_filter_coeff0_a};
assign ram[1] = {rf_filter_coeff1_b, rf_filter_coeff1_a};
assign ram[2] = {rf_filter_coeff2_b, rf_filter_coeff2_a};
assign ram[3] = {rf_filter_coeff3_b, rf_filter_coeff3_a};
assign ram[510] = {rf_filter_coeff510_b, rf_filter_coeff510_a};
assign ram[511] = {rf_filter_coeff511_b, rf_filter_coeff511_a};
always @(posedge clk)
    begin
        if(rden)
            rddata <= ram[rdptr];
    end
endmodule
```

3.5 *filter_accumulator.v*

```
`timescale 1ns / 1ps
module filter_accumulator(clk, rstb, enable,load, D, Q
    );
input  clk, rstb,enable,load;
input  [31:0] D;
output [40:0] Q;
reg    [40:0] tmp;
assign Q = tmp;
    always @(posedge clk or negedge rstb)
        begin
            if (!rstb)
                tmp <= 1'b0;
            else if (enable)
                begin
                    if (load)
                        tmp <= D;
                    else
                        tmp <= tmp + D;
                end
            end
        end
endmodule
```


3.6 *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
```

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_overflow_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_overflow_clr <= 0;
        trig_i2so_fifo_underrun_clr <= 0;
    end

    else
    begin
        // initializing trigger bits to zero
        trig_i2si_fifo_overflow_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_overflow_clr is triggered
            if (wdata[0])
                trig_i2si_fifo_overflow_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 [10:0] addr,
    input [7:0] wdata,
    input w_enable,
    input i2c_xfc_write,
    input ro_i2c_reg_indir_data, ro_fifo_underrun, ro_fifo_overflow,
    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_overflow, 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_coeff510_b, rf_filter_coeff511_a, rf_filter_coeff511_b
);

always @(posedge clk or negedge rst)

    if (~rst)
    begin
        rf_soft_reset <= 1'h0;
        rf_i2si_bist_en <= 1'h1;
        rf_filter_shift <= 4'hf;
        rf_filter_clip_en <= 1'h1;
        rf_i2si_dec_factor <= 4'h0;
        rf_i2so_dec_factor <= 4'h0;
        rf_i2so_clk2sck_div_a <= 8'h40;
        rf_i2so_clk2sck_div_b <= 8'h40;
        trig_fifo_overflow <= 1'h0; //NA?
        ro_fifo_overflow <= 1'h0;
        trig_fifo_underrun <= 1'h0; //NA?
        ro_fifo_underrun <= 1'h0;
        rf_i2si_bist_incr <= 8'h010;
        rf_i2si_bist_start_val_a <= 8'h800;
        rf_i2si_bist_start_val_b <= 4'h800;
        rf_i2si_bist_upper_limit_a <= 4'h7ff;
        rf_i2si_bist_upper_limit_b <= 8'h7ff;
        rf_i2c_reg_indir_addr_a <= 8'h000;
        rf_i2c_reg_indir_addr_b <= 3'h000;
        rf_filter_coeff0_a <= 11'h000;
        rf_filter_coeff0_b <= 11'h000;
        ...
        rf_filter_coeff511_a <= 11'h000;
        rf_filter_coeff511_b <= 11'h000;
    end
    else if (i2c_xfc_write && w_enable)
    begin
        // Given the address, the signals are assigned to their correlated bits of data
        case(addr)
            11'h004:
                rf_soft_reset <= wdata[0];
                rf_i2si_bist_en <= wdata[1];
                rf_filter_shift <= wdata[5:2];
                rf_filter_clip_en <= wdata[6];
            11'h005:
                rf_i2si_dec_factor <= wdata[3:0];
                rf_i2so_dec_factor <= wdata[7:4];
            11'h008:
                rf_i2so_clk2sck_div_a <= wdata[7:0];
            11'h009:
                rf_i2so_clk2sck_div_b <= wdata[7:0];
```

```

11'h00c:
    trig_fifo_overflow <= wdata[0];
    //ro_fifo_overflow <= wdata[1];
    trig_fifo_underrun <= wdata[2];
    //ro_fifo_underrun <= wdata[3];
11'h010:
    rf_i2si_bist_incr <= wdata[7:0];
11'h011:
    rf_i2si_bist_start_val_a <= wdata[7:0];
11'h012:
    rf_i2si_bist_start_val_b <= wdata[3:0];
    rf_i2si_bist_upper_limit_a <= wdata[7:4];
11'h013:
    rf_i2si_bist_upper_limit_b <= wdata[7:0];
11'h014:
    rf_i2c_reg_indir_addr_a <= wdata[7:0];
11'h015:
    rf_i2c_reg_indir_addr_b <= wdata[3:0];
11'h400
    rf_filter_coeff0_a <= wdata[7:0];
11'h401
    rf_filter_coeff0_b <= wdata[7:0];
    ...
11'h7fc:
    rf_filter_coeff510_a <= wdata[7:0];
11'h7fd:
    rf_filter_coeff510_b <= wdata[7:0];
11'h7fe:
    rf_filter_coeff511_a <= wdata[7:0];
11'h7ff:
    rf_filter_coeff511_b <= wdata[7:0];
endcase
end
end
//read mux

always @ (posedge clk or negedge rst)
    if (i2c_xfc_read && w_enable)
        begin
            // Given the address, the signals are assigned to their correlated bits of data
            case(addr)
                11'h004:
                    rdata[0] => rf_soft_rest;
                    rdata[1] => rf_i2si_bist_en;
                    rdata[5:2] => rf_filter_shift;
                    rdata[6] => rf_filter_clip_en;
                11'h005:
                    rdata[3:0] => rf_i2si_dec_factor;
                    rdata[7:4] => rf_i2so_dec_factor;
                11'h008:
                    rdata[7:0] => rf_i2so_clk2sck_div_a;
                11'h009:
                    rdata[7:0] => rf_i2so_clk2sck_div_b;
                11'h00c:
                    rdata[0] => trig_fifo_overflow;
                    rdata[1] => ro_fifo_overflow;
                    rdata[2] => trig_fifo_underrun;
                    rdata[3] => ro_fifo_underrun;
                11'h010:
                    rdata[7:0] => rf_i2si_bist_incr;
                11'h011:
                    rdata[7:0] => rf_i2si_bist_start_val_a;
                11'h012:
                    rdata[3:0] => rf_i2si_bist_start_val_b;
                    rdata[7:4] => rf_i2si_bist_upper_limit_a;
                11'h013:
                    rdata[7:0] => rf_i2si_bist_upper_limit_b;
                11'h014:
                    rdata[7:0] => rf_i2c_reg_indir_addr_a;
                11'h015:

```

```

        rdata[3:0] => rf_i2c_reg_indir_addr_b;
11'h400
        rdata[7:0] => rf_filter_coeff0_a;
11'h401
        rdata[7:0] => rf_filter_coeff0_b;
        ...
11'h7fe
        rdata[7:0] => rf_filter_coeff511_a;
11'h7ff
        rdata[7:0] => rf_filter_coeff511_b;
    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<=511; i++)
    {
        y = y + 1;
        fprintf (fp, "\t\ttrf_filter_coeff%d_a <= 11'h000;\n", y);
        fprintf (fp, "\t\ttrf_filter_coeff%d_b <= 11'h000;\n", y);
    }
    return 0;
}
```



```
#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\t\tl1'h%x\n\t\t\t\t\t\t%s%d%s\n", address, name, coeffNum, ab,
pointer);
    }
    return 0;
}
```

4.5 *set_read_coeffs.c*

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

int main(void)
{
    // Creating and opening file
    FILE *fp;
    fp = fopen("filter_coeffs_read.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[] = "rdata[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\t\tl1'h%x\n\t\t\t\t\t%s%s%d%s;\n", address, pointer, name, coeffNum,
ab);
    }
    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 burst start address
                                bit_counter_burst_addr <= 4'b1100;
                            end
        endcase
    end
end

```

```

                                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
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 C: Test Benches

1. i2s_in.v
 - 1.1. i2si_deserializer_testbench.v
 - 1.2. bist_test.v
 - 1.3. fifo_test.v
2. i2s_out.v
 - 2.1. serializer_testbench.v
3. filter.v
 - 3.1. filter_tf.v
 - 3.2. filter_stm_tf.v
 - 3.3. filter_storage_tf.v
 - 3.4. filter_mux_tf.v
 - 3.5. filter_accumulator_tf.v
 - 3.6. filter_barrel_shifter_tf.v
4. register.v
 - 4.1. trig_generator_testbench.v
 - 4.2. trig_generator_testbench1.v
5. i2c_slave.v
 - 5.1. deserial_sequencer_test.v
 - 5.2. deserial_test2.v
 - 5.3. 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
end
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
        end
    end
endmodule
```

```

        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*

[illegible]

```

        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;
    reg [15:0] test_data [0:9] [0:1];
    test_data [# of entities in test] [Left/Right]

    // word counter
    // [Bits Per Word]

    // 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. filter.v:

3.1 filter_tf.v

```
initial begin

    // Initialize Inputs
    clk = 0;
    rstb = 0;
    aud_in = 0;
    aud_in_rts = 0;
    aud_out_rtr = 0;
    rf_filter_coeff0_a = 8'h01;
    rf_filter_coeff0_b = 8'h00;
    rf_filter_coeff1_a = 8'h02;
    rf_filter_coeff1_b = 8'h00;
    rf_filter_coeff2_a = 8'h03;
    rf_filter_coeff2_b = 8'h00;
    rf_filter_coeff3_a = 8'h04;
    rf_filter_coeff3_b = 8'h00;
    rf_filter_coeff4_a = 8'h05;
    rf_filter_coeff4_b = 8'h00;
    rf_filter_coeff5_a = 8'h06;
    rf_filter_coeff5_b = 8'h00;
    rf_filter_coeff6_a = 8'h07;
    rf_filter_coeff6_b = 8'h00;
    rf_filter_coeff7_a = 8'h08;
    rf_filter_coeff7_b = 8'h00;

    // Wait 100 ns for global reset to finish
    #100;
    rstb = 1;
    aud_in_rts = 1;
    // Add stimulus here
    aud_in = 32'h00AA00AA;
    #150;
    aud_in = 32'h00AA003A;
    #150;
    aud_in = 32'h00DD00AA;
    #150;
    aud_in = 32'h001A00A2;
    #150;
    aud_in = 32'h001100DA;
    #150;
    aud_in = 32'h00AA00FF;
    #150;
    aud_in = 32'h00AF003A;
    #150;
    aud_in = 32'h00AA009F;
```

```

        #150;
        aud_in = 32'h00FF0022;
        #150;
        aud_in = 32'h00670029;
        #150;
        aud_in = 32'h00120074;
        #150;
        aud_in = 32'h004A0030;
        #150;
        aud_in = 32'h00C10021;
        #150;
        aud_in = 32'h00B90019;
        #150;
        aud_in = 32'h00A50003;
        #150;
        aud_in = 32'h00550032;
        #150;
    end
always
begin
    forever #5 clk = ~clk;
end

```

3.2 *filter_stm_tf.v*

```
initial begin
    // Initialize Inputs
    clk = 0;
    rstb = 0;
    filter_aud_in_rts = 0;
    filter_aud_in = 7;
    rf_filter_coeff = 0;

    // Wait 100 ns for global reset to finish
    #100;
    rstb = 1;
    filter_aud_in_rts = 1;
    rf_filter_coeff = 1;
    #100;
    filter_aud_in = 13;
    #100;
    filter_aud_in = 3;
    #100;
    filter_aud_in = 5;
    #100;
    filter_aud_in = 8;
    #100;
    filter_aud_in = 18;
    #100;
    filter_aud_in = 4;
    #100;
    filter_aud_in = 7;
    #100;
    filter_aud_in = 2;
    #100;
    filter_aud_in = 1;
    // Add stimulus here

end

always
begin
    forever #5 clk = ~clk;
end
```

3.3 *filter_storage_tf.v*

```
initial begin
    // Initialize Inputs
    clk = 0;
    wren = 0;
    wrptr = 0;
    wrdata = 0;
    rden = 0;
    rdptr = 0;
    wren = 0;
    rstb = 0;
    // Wait 100 ns for global reset to finish
    #100;
    rstb = 1;
    wren = 1;
    wrdata = 32'hAAAAAAAA;

    #100;
    wrptr = wrptr + 513;
    wrdata = 32'hBBBBBBBB;
    #100;
    wren = 0;
    rden = 1;
    rdptr = 1;

end

always
begin
    forever #5 clk = ~clk;
end
```

3.4 *filter_mux_tf.v*

```
initial begin
    // Initialize Inputs
    clk = 0;
    rden = 0;
    rdptr = 0;
    rf_filter_coeff0_a = 8'h00;
    rf_filter_coeff0_b = 8'h08;
    rf_filter_coeff1_a = 8'h01;
    rf_filter_coeff1_b = 8'h07;
    rf_filter_coeff2_a = 8'h02;
    rf_filter_coeff2_b = 8'h06;
    rf_filter_coeff3_a = 8'h03;
    rf_filter_coeff3_b = 8'h05;
    rf_filter_coeff4_a = 8'h04;
    rf_filter_coeff4_b = 8'h04;
    rf_filter_coeff5_a = 8'h05;
    rf_filter_coeff5_b = 8'h03;
    rf_filter_coeff6_a = 8'h06;
    rf_filter_coeff6_b = 8'h02;
    rf_filter_coeff7_a = 8'h07;
    rf_filter_coeff7_b = 8'h01;
    rf_filter_coeff8_a = 8'h08;
    rf_filter_coeff8_b = 8'h00;
    .....
    rf_filter_coeff509_b = 0;
    rf_filter_coeff510_a = 0;
    rf_filter_coeff510_b = 0;
    rf_filter_coeff511_a = 0;
    rf_filter_coeff511_b = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    rden = 1;
    rdptr = 0;
    #100;
    rdptr = 1;
    #100;
    rdptr = 2;
    #100;
    rdptr = 3;
    #100;
    rdptr = 4;
    #100;
```

```
        rdptr = 5;
        #100;
        rdptr = 6;
        #100;
        rdptr = 7;
        #100;
        rdptr = 8;
    end

    always
begin
    forever #5 clk = ~clk;
end
```

3.5 *filter_accumulator_tf.v*

```
initial begin
    // Initialize Inputs
    clk = 0;
    rstb = 0;
    enable = 0;
    load = 0;
    #100;
    rstb = 1;
    D = 16'h000A;
    enable = 1;
    load = 1;
    #10;
    load = 0;
    D = 16'hFFFF;
    #10;
    D = 16'h0334;
    #10;
    D = 16'hFFFF;
    #10;
    D = 16'hAA44;
    #100;
    enable = 0;
    #100;
    enable = 1;
    load = 1;
    D = 16'h0000;
    #100;
    load=0;
    D = 16'h0001;
    #100;
    rstb = 0;
end

always
begin
    forever #5 clk = ~clk;
end
```


3.6 *filter_barrel_shifter_tf.v*

```
initial begin
    // Initialize Inputs
    input_signal = 32'hABCD1234;
    sel_shift = 0;
    // Wait 100 ns for global reset to finish
    #50;
    sel_shift = 4;
    #50;
    sel_shift = 8;
    #50;
    sel_shift = 12;
    #50;
    sel_shift = 16;
    #50;
    sel_shift = 20;
    #50;
    sel_shift = 24;
    #50;
    sel_shift = 28;
    #50;
    sel_shift = 32;
    #50 $finish;

    // Add stimulus here

end
```

4. register.v:

4.1 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_overflow_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_overflow_clr(trig_i2si_fifo_overflow_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
end
```

4.2 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_overnrun_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_overnrun_clr(trig_i2si_fifo_overnrun_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
end
endmodule
```

5. i2c_slave.v:

5.1 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;
    end

//`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;
    end
endmodule
```

```

#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;
#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

```

5.2 *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
        end
    end
```


[illegible]

```

#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

```

5.3 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 D: Industry Specifciations

1. I2S
2. I2C

Included in the PDF version

I²S bus specification

1.0 INTRODUCTION

Many digital audio systems are being introduced into the consumer audio market, including compact disc, digital audio tape, digital sound processors, and digital TV-sound. The digital audio signals in these systems are being processed by a number of (V)LSI ICs, such as:

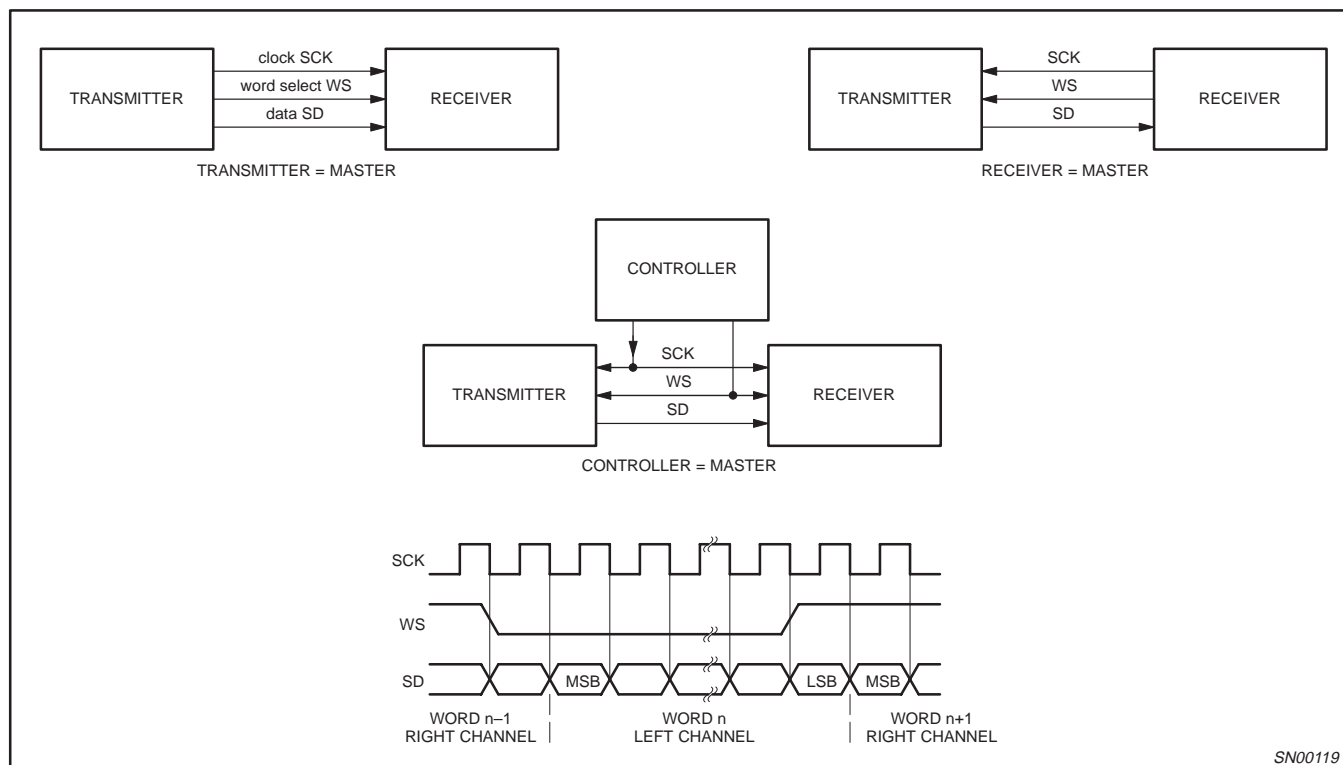
- A/D and D/A converters;
- digital signal processors;
- error correction for compact disc and digital recording;
- digital filters;
- digital input/output interfaces.

Standardized communication structures are vital for both the equipment and the IC manufacturer, because they increase system flexibility. To this end, we have developed the inter-IC sound (I²S) bus – a serial link especially for digital audio.

2.0 BASIC SERIAL BUS REQUIREMENTS

The bus has only to handle audio data, while the other signals, such as sub-coding and control, are transferred separately. To minimize the number of pins required and to keep wiring simple, a 3-line serial bus is used consisting of a line for two time-multiplexed data channels, a word select line and a clock line.

Since the transmitter and receiver have the same clock signal for data transmission, the transmitter as the master, has to generate the bit clock, word-select signal and data. In complex systems however, there may be several transmitters and receivers, which makes it difficult to define the master. In such systems, there is usually a system master controlling digital audio data-flow between the various ICs. Transmitters then, have to generate data under the control of an external clock, and so act as a slave. Figure 1 illustrates some simple system configurations and the basic interface timing. Note that the system master can be combined with a transmitter or receiver, and it may be enabled or disabled under software control or by pin programming.



SN00119

Figure 1. Simple System Configurations and Basic Interface Timing

I²S bus specification

3.0 THE I²S BUS

As shown in Figure 1, the bus has three lines:

- continuous serial clock (SCK);
- word select (WS);
- serial data (SD);

and the device generating SCK and WS is the master.

3.1 Serial Data

Serial data is transmitted in two's complement with the MSB first. The MSB is transmitted first because the transmitter and receiver may have different word lengths. It isn't necessary for the transmitter to know how many bits the receiver can handle, nor does the receiver need to know how many bits are being transmitted.

When the system word length is greater than the transmitter word length, the word is truncated (least significant data bits are set to '0') for data transmission. If the receiver is sent more bits than its word length, the bits after the LSB are ignored. On the other hand, if the receiver is sent fewer bits than its word length, the missing bits are set to zero internally. And so, the MSB has a fixed position, whereas the position of the LSB depends on the word length. The transmitter always sends the MSB of the next word one clock period after the WS changes.

Serial data sent by the transmitter may be synchronized with either the trailing (HIGH-to-LOW) or the leading (LOW-to-HIGH) edge of the clock signal. However, the serial data must be latched into the receiver on the leading edge of the serial clock signal, and so there are some restrictions when transmitting data that is synchronized with the leading edge (see Figure 2 and Table 1).

3.2 Word Select

The word select line indicates the channel being transmitted:

- WS = 0; channel 1 (left);
- WS = 1; channel 2 (right).

WS may change either on a trailing or leading edge of the serial clock, but it doesn't need to be symmetrical. In the slave, this signal

is latched on the leading edge of the clock signal. The WS line changes one clock period before the MSB is transmitted. This allows the slave transmitter to derive synchronous timing of the serial data that will be set up for transmission. Furthermore, it enables the receiver to store the previous word and clear the input for the next word (see Figure 1).

4.0 TIMING

In the I²S format, any device can act as the system master by providing the necessary clock signals. A slave will usually derive its internal clock signal from an external clock input. This means, taking into account the propagation delays between master clock and the data and/or word-select signals, that the total delay is simply 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.

For data and word-select inputs, the external to internal clock delay is of no consequence because it only lengthens the effective set-up time (see Figure 2). The major part of the time margin is to accommodate the difference between the propagation delay of the transmitter, and the time required to set up the receiver.

All timing requirements are specified relative to the clock period or to the minimum allowed clock period of a device. This means that higher data rates can be used in the future.

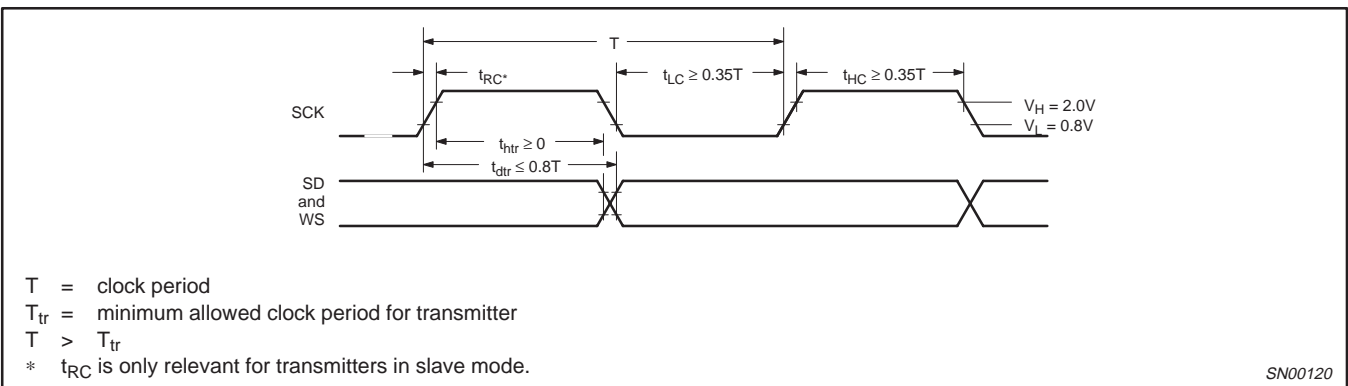


Figure 2. Timing for I²S Transmitter

I²S bus specification

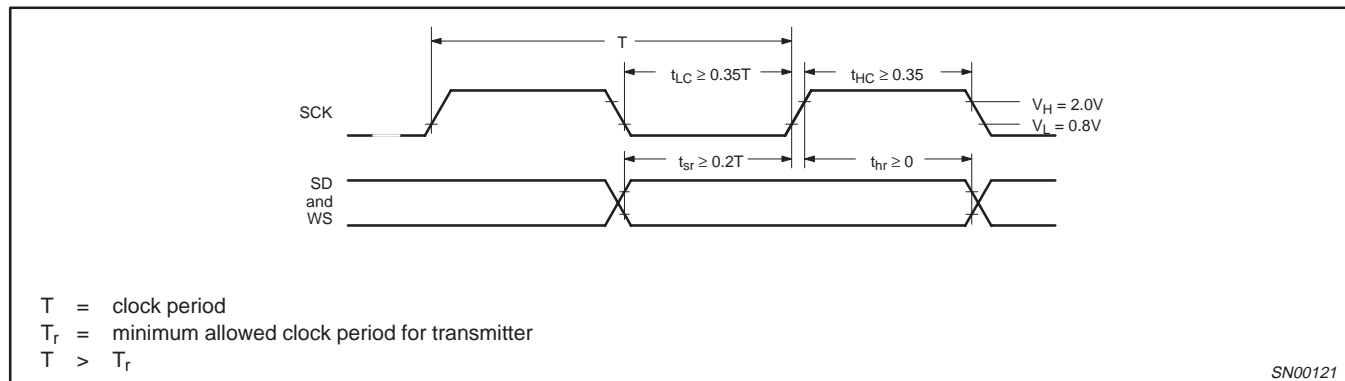


Figure 3. Timing for I²S Receiver

Note that the times given in both Figures 2 and 3 are defined by the transmitter speed. The specification of the receiver has to be able to match the performance of the transmitter

Example: Master transmitter with data rate of 2.5MHz ($\pm 10\%$) (all values in ns)

	MIN	TYP	MAX	CONDITION
clock period T	360	400	440	$T_{tr} = 360$
clock HIGH t_{HC}	160			min $> 0.35T = 140$ (at typical data rate)
clock LOW t_{LC}	160			min $> 0.35T = 140$ (at typical data rate)
delay t_{dtr}			300	max $< 0.80T = 320$ (at typical data rate)
hold time t_{htr}	100			min > 0
clock rise-time t_{RC}			60	max $> 0.15T_{tr} = 54$ (only relevant in slave mode)

Example: Slave receiver with data rate of 2.5MHz ($\pm 10\%$) (all values in ns)

	MIN	TYP	MAX	CONDITION
clock period T	360	400	440	$T_{tr} = 360$
clock HIGH t_{HC}	110			min $< 0.35T = 126$
clock LOW t_{LC}	110			min $< 0.35T = 126$
set-up time t_{sr}	60			min $< 0.20T = 72$
hold time t_{htr}	0			min < 0

I²S bus specification

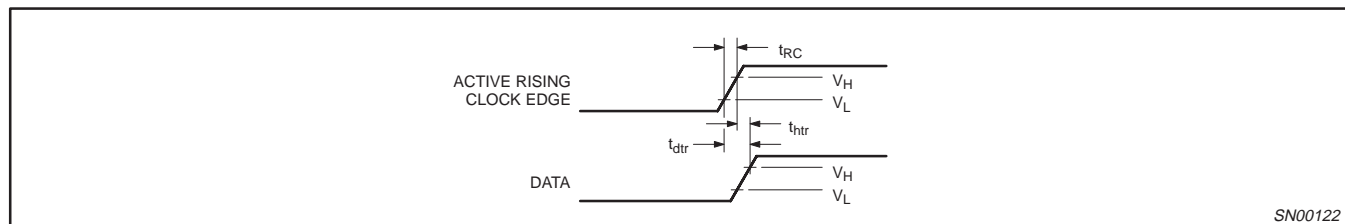
Table 1. Timing for I²S transmitters and receivers

	TRANSMITTER				RECEIVER				NOTES
	LOWER LIMIT		UPPER LIMIT		LOWER LIMIT		UPPER LIMIT		
	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	
Clock period T	T _{tr}				T _r				1
MASTER MODE: clock generated by transmitter or receiver: HIGH t _{HC} LOW t _{LC}	0.35T _{tr} 0.35T _{tr}				0.35T _{tr} 0.35T _{tr}				2a 2a
SLAVE MODE: clock accepted by transmitter or receiver: HIGH t _{HC} LOW t _{LC} rise-time t _{RC}		0.35T _{tr} 0.35T _{tr}	0.15T _{tr}			0.35T _r 0.35T _r			2b 2b 3
TRANSMITTER: delay t _{dtr} hold time t _{htr}	0			0.8T					4 3
RECEIVER: set-up time t _{sr} hold time t _{hr}						0.2T _r 0			5 5

All timing values are specified with respect to high and low threshold levels.

NOTES:

1. The system clock period T must be greater than T_{tr} and T_r because both the transmitter and receiver have to be able to handle the data transfer rate.
- 2a. At all data rates in the master mode, the transmitter or receiver generates a clock signal with a fixed mark/space ratio. For this reason t_{HC} and t_{LC} are specified with respect to T.
- 2b. In the slave mode, the transmitter and receiver need a clock signal with minimum HIGH and LOW periods so that they can detect the signal. So long as the minimum periods are greater than $0.35T_r$, any clock that meets the requirements can be used (see Figure 3).
3. Because the delay (t_{dtr}) and the maximum transmitter speed (defined by T_{tr}) are related, a fast transmitter driven by a slow clock edge can result in t_{dtr} not exceeding t_{RC} which means t_{htr} becomes zero or negative. Therefore, the transmitter has to guarantee that t_{htr} is greater than or equal to zero, so long as the clock rise-time t_{RC} is not more than t_{RCmax} , where t_{RCmax} is not less than $0.15T_{tr}$.
4. To allow data to be clocked out on a falling edge, the delay is specified with respect to the rising edge of the clock signal and T, always giving the receiver sufficient set-up time.
5. The data set-up and hold time must not be less than the specified receiver set-up and hold time.


Figure 4. Clock rise-time definition with respect to the voltage levels

I²S bus specification

5.0 VOLTAGE LEVEL SPECIFICATION

5.1 Output Levels

$$V_L < 0.4V$$

$V_H > 2.4V$ both levels able to drive one standard TTL input ($I_{IL} = -1.6mA$ and $I_{IH} = 0.04mA$).

5.2 Input Levels

$$V_{IL} = 0.8V$$

$$V_{IH} = 2.0V$$

Note: At present, TTL is considered a standard for logic levels. As other IC (LSI) technologies become popular, other levels will also be supported.

6.0 POSSIBLE HARDWARE CONFIGURATIONS

6.1 Transmitter (see Figure 5)

At each WS-level change, a pulse WSP is derived for synchronously parallel-loading the shift register. The output of one of the data latches is then enabled depending on the WS signal. Since the serial data input is zero, all the bits after the LSB will also be zero.

6.2 Receiver (see Figure 6)

Following the first WS-level change, WSP will reset the counter on the falling edge of SCK. After decoding the counter value in a "1 out of n" decoder, the MSB latch (B1) is enabled ($EN1 = 1$), and the first serial data bit (the MSB) is latched into B1 on the rising edge of SCK. As the counter increases by one every clock pulse, subsequent data bits are latched into B2 to Bn.

On the next WS-level change, the contents of the n latches are written in parallel, depending on WSD, into either the left or the right data-word latch. After this, latches B2 to Bn are cleared and the counter reset. If there are more than n serial data bits to be latched, the counter is inhibited after Bn (the receiver's LSB) is filled and subsequent bits are ignored.

Note: The counter and decoder can be replaced by an n-bit shift-register (see Figure 7) in which a single '1' is loaded into the MSB position when WSP occurs. On every subsequent clock pulse, this '1' shifts one place, enabling the N latches. This configuration may prove useful if the layout has to be taken into account.

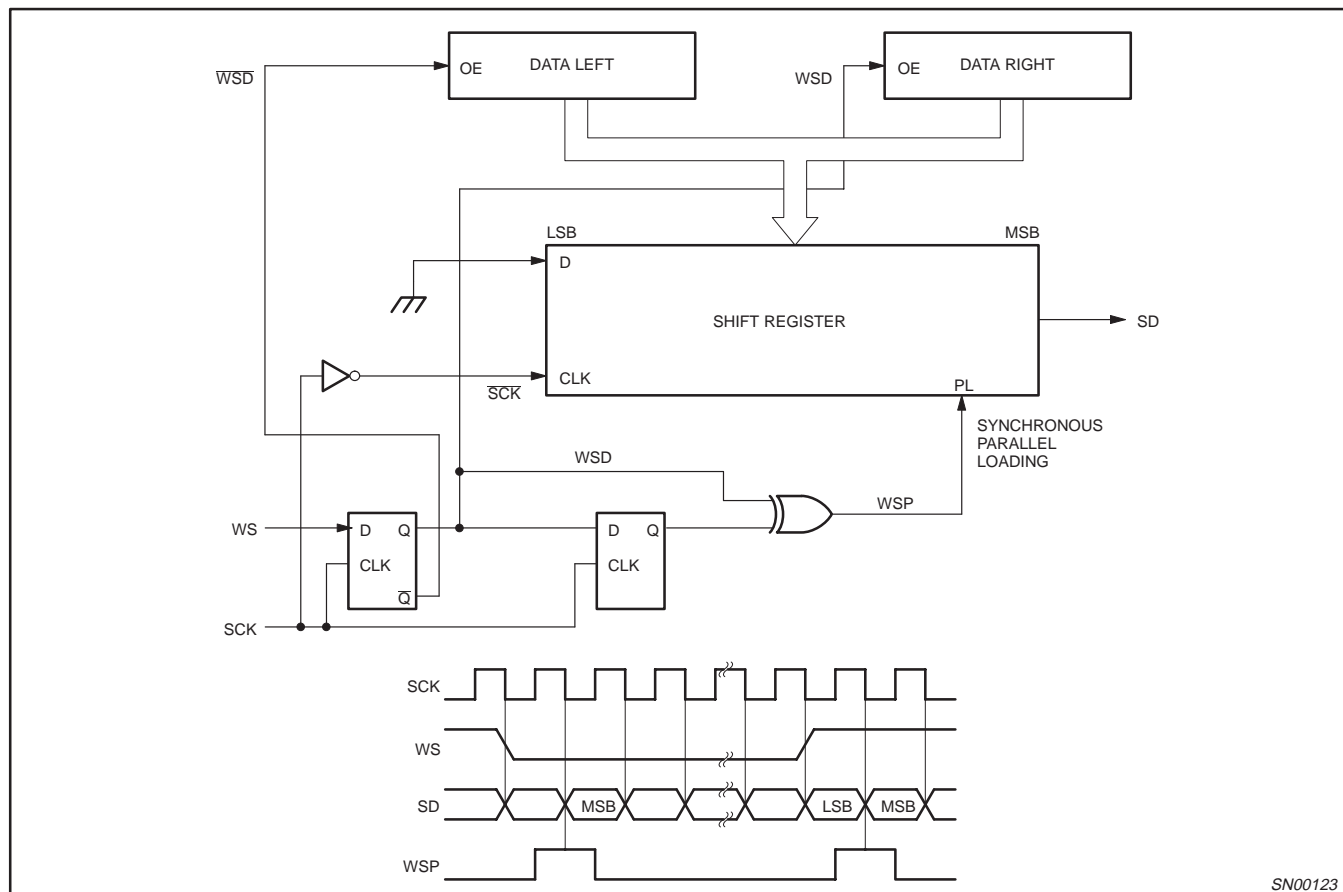
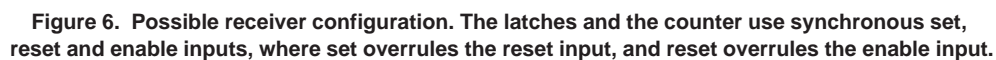


Figure 5. Possible transmitter configuration



I²S bus specification

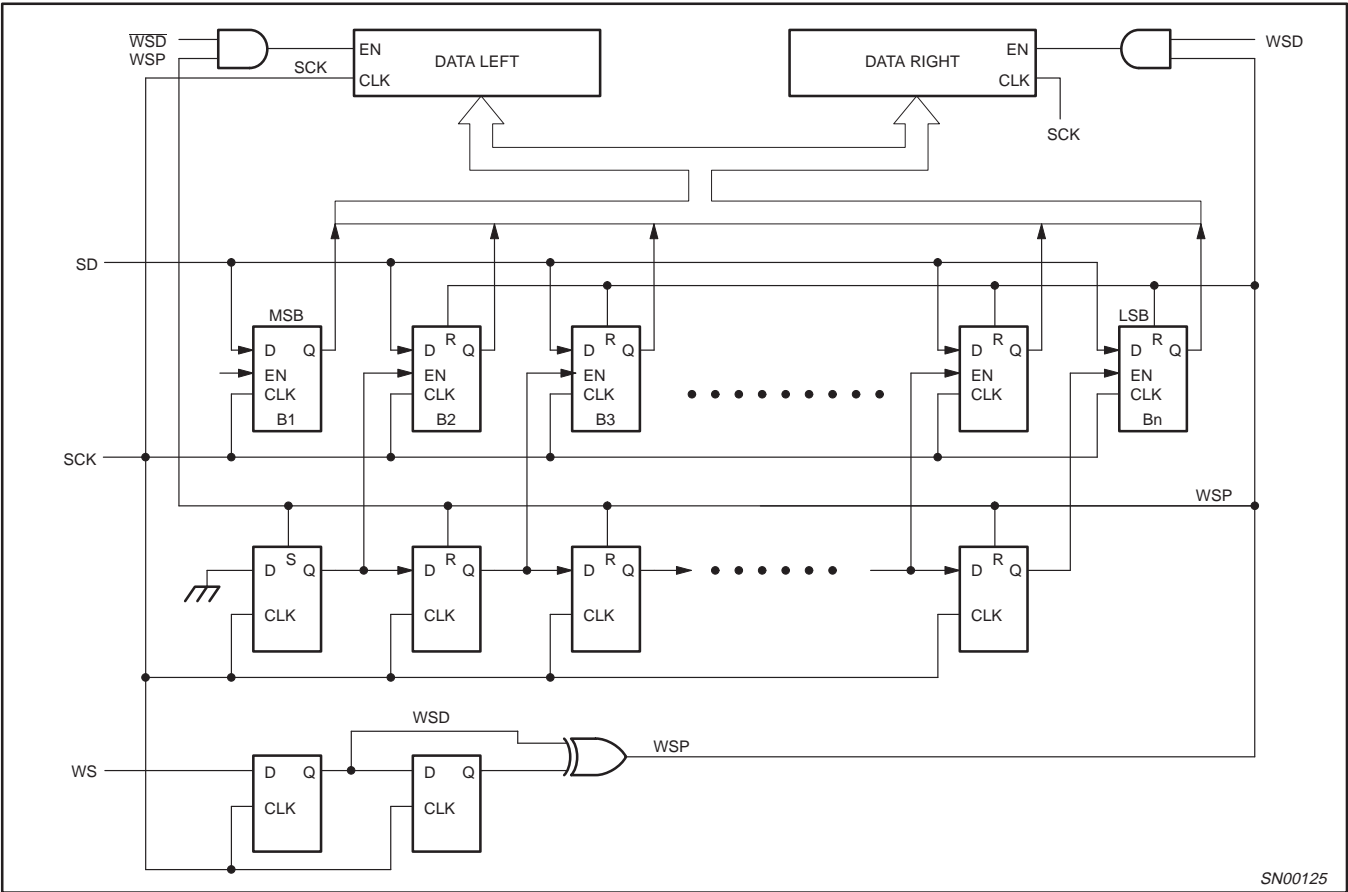


Figure 7. Possible receiver configuration, using an n-bit shift-register to enable control of data input register.



UM10204

I²C-bus specification and user manual

Rev. 6 — 4 April 2014

User manual

Document information

Info	Content
Keywords	I2C, I2C-bus, Standard-mode, Fast-mode, Fast-mode Plus, Fm+, Ultra Fast-mode, UFm, High Speed, Hs, inter-IC, SDA, SCL, USDA, USCL
Abstract	Philips Semiconductors (now NXP Semiconductors) developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter-IC or I ² C-bus. Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL). Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in the Fast-mode Plus (Fm+), or up to 3.4 Mbit/s in the High-speed mode. The Ultra Fast-mode is a uni-directional mode with data transfers of up to 5 Mbit/s.



Revision history

Rev	Date	Description
v.6	20140404	User manual; sixth release
Modifications:		<ul style="list-style-type: none">• Figure 41 "R_{p(max)} as a function of bus capacitance" updated (recalculated)• Figure 42 "R_{p(min)} as a function of V_{DD}" updated (recalculated)
v.5	20121009	User manual; fifth release
v.4	20120213	User manual Rev. 4
v.3	20070619	Many of today's applications require longer buses and/or faster speeds. Fast-mode Plus was introduced to meet this need by increasing drive strength by as much as 10× and increasing the data rate to 1 Mbit/s while maintaining downward compatibility to Fast-mode and Standard-mode speeds and software commands.
v2.1	2000	Version 2.1 of the I ² C-bus specification
v2.0	1998	The I ² C-bus has become a de facto world standard that is now implemented in over 1000 different ICs and licensed to more than 50 companies. Many of today's applications, however, require higher bus speeds and lower supply voltages. This updated version of the I ² C-bus specification meets those requirements.
v1.0	1992	Version 1.0 of the I ² C-bus specification
Original	1982	first release

Contact information

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

The I²C-bus is a de facto world standard that is now implemented in over 1000 different ICs manufactured by more than 50 companies. Additionally, the versatile I²C-bus is used in various control architectures such as System Management Bus (SMBus), Power Management Bus (PMBus), Intelligent Platform Management Interface (IPMI), Display Data Channel (DDC) and Advanced Telecom Computing Architecture (ATCA).

This document assists device and system designers to understand how the I²C-bus works and implement a working application. Various operating modes are described. It contains a comprehensive introduction to the I²C-bus data transfer, handshaking and bus arbitration schemes. Detailed sections cover the timing and electrical specifications for the I²C-bus in each of its operating modes.

Designers of I²C-compatible chips should use this document as a reference and ensure that new devices meet all limits specified in this document. Designers of systems that include I²C devices should review this document and also refer to individual component data sheets.

2. I²C-bus features

In consumer electronics, telecommunications and industrial electronics, there are often many similarities between seemingly unrelated designs. For example, nearly every system includes:

- Some intelligent control, usually a single-chip microcontroller
- General-purpose circuits like LCD and LED drivers, remote I/O ports, RAM, EEPROM, real-time clocks or A/D and D/A converters
- Application-oriented circuits such as digital tuning and signal processing circuits for radio and video systems, temperature sensors, and smart cards

To exploit these similarities to the benefit of both systems designers and equipment manufacturers, as well as to maximize hardware efficiency and circuit simplicity, Philips Semiconductors (now NXP Semiconductors) developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter IC or I²C-bus. All I²C-bus compatible devices incorporate an on-chip interface which allows them to communicate directly with each other via the I²C-bus. This design concept solves the many interfacing problems encountered when designing digital control circuits.

Here are some of the features of the I²C-bus:

- Only two bus lines are required; a serial data line (SDA) and a serial clock line (SCL).
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.
- It is a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.

- Serial, 8-bit oriented, unidirectional data transfers up to 5 Mbit/s in Ultra Fast-mode
- On-chip filtering rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance. More capacitance may be allowed under some conditions. Refer to [Section 7.2](#).

[Figure 1](#) shows an example of I²C-bus applications.

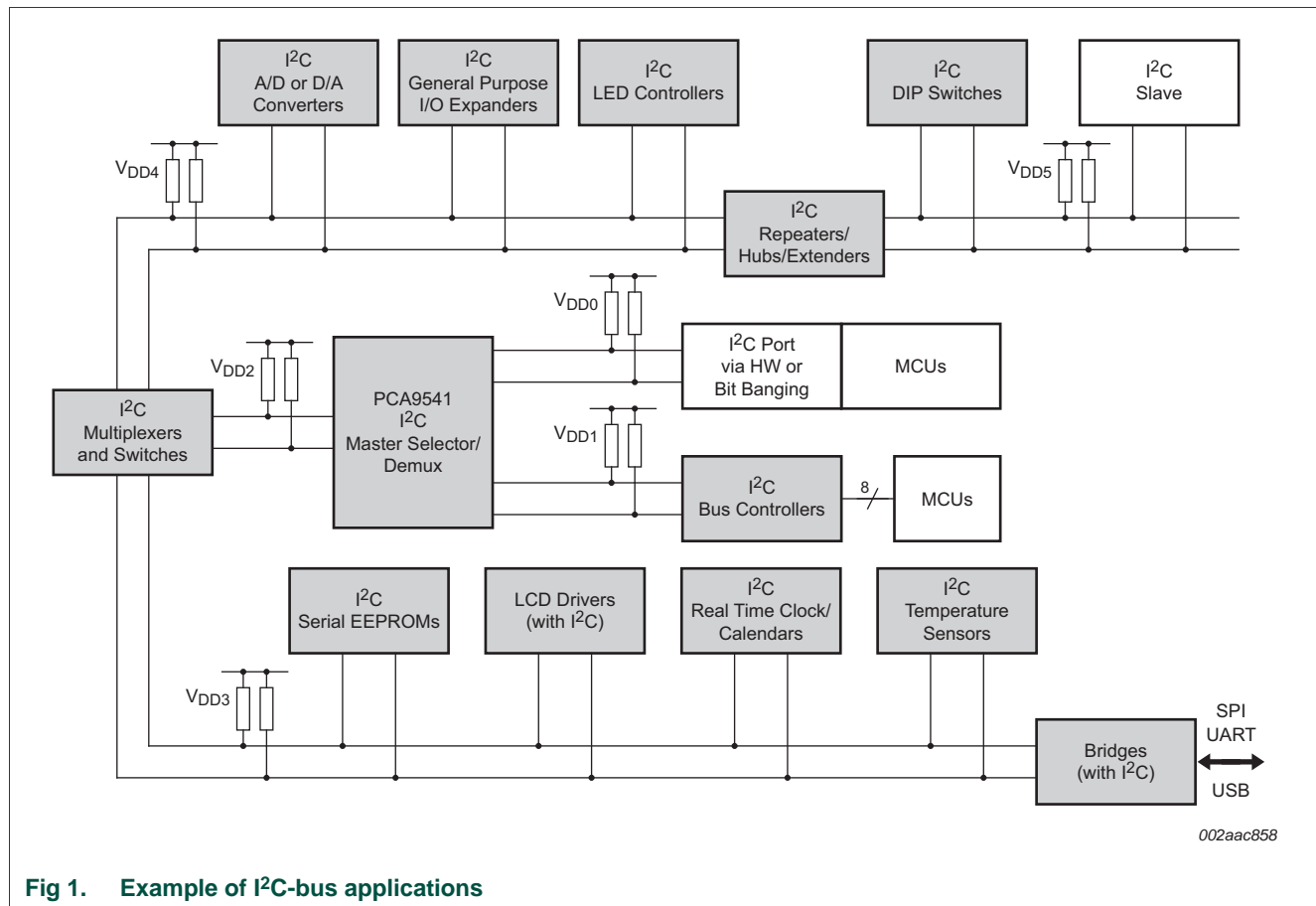


Fig 1. Example of I²C-bus applications

2.1 Designer benefits

I²C-bus compatible ICs allow a system design to progress rapidly directly from a functional block diagram to a prototype. Moreover, since they 'clip' directly onto the I²C-bus without any additional external interfacing, they allow a prototype system to be modified or upgraded simply by 'clipping' or 'unclipping' ICs to or from the bus.

Here are some of the features of I²C-bus compatible ICs that are particularly attractive to designers:

- Functional blocks on the block diagram correspond with the actual ICs; designs proceed rapidly from block diagram to final schematic.
- No need to design bus interfaces because the I²C-bus interface is already integrated on-chip.

- Integrated addressing and data-transfer protocol allow systems to be completely software-defined.
- The same IC types can often be used in many different applications.
- Design-time reduces as designers quickly become familiar with the frequently used functional blocks represented by I²C-bus compatible ICs.
- ICs can be added to or removed from a system without affecting any other circuits on the bus.
- Fault diagnosis and debugging are simple; malfunctions can be immediately traced.
- Software development time can be reduced by assembling a library of reusable software modules.

In addition to these advantages, the CMOS ICs in the I²C-bus compatible range offer designers special features which are particularly attractive for portable equipment and battery-backed systems.

They all have:

- Extremely low current consumption
- High noise immunity
- Wide supply voltage range
- Wide operating temperature range.

2.2 Manufacturer benefits

I²C-bus compatible ICs not only assist designers, they also give a wide range of benefits to equipment manufacturers because:

- The simple 2-wire serial I²C-bus minimizes interconnections so ICs have fewer pins and there are not so many PCB tracks; result — smaller and less expensive PCBs.
- The completely integrated I²C-bus protocol eliminates the need for address decoders and other 'glue logic'.
- The multi-master capability of the I²C-bus allows rapid testing and alignment of end-user equipment via external connections to an assembly line.
- The availability of I²C-bus compatible ICs in various leadless packages reduces space requirements even more.

These are just some of the benefits. In addition, I²C-bus compatible ICs increase system design flexibility by allowing simple construction of equipment variants and easy upgrading to keep designs up-to-date. In this way, an entire family of equipment can be developed around a basic model. Upgrades for new equipment, or enhanced-feature models (that is, extended memory, remote control, etc.) can then be produced simply by clipping the appropriate ICs onto the bus. If a larger ROM is needed, it is simply a matter of selecting a microcontroller with a larger ROM from our comprehensive range. As new ICs supersede older ones, it is easy to add new features to equipment or to increase its performance by simply unclipping the outdated IC from the bus and clipping on its successor.

2.3 IC designer benefits

Designers of microcontrollers are frequently under pressure to conserve output pins. The I²C protocol allows connection of a wide variety of peripherals without the need for separate addressing or chip enable signals. Additionally, a microcontroller that includes an I²C interface is more successful in the marketplace due to the wide variety of existing peripheral devices available.

3. The I²C-bus protocol

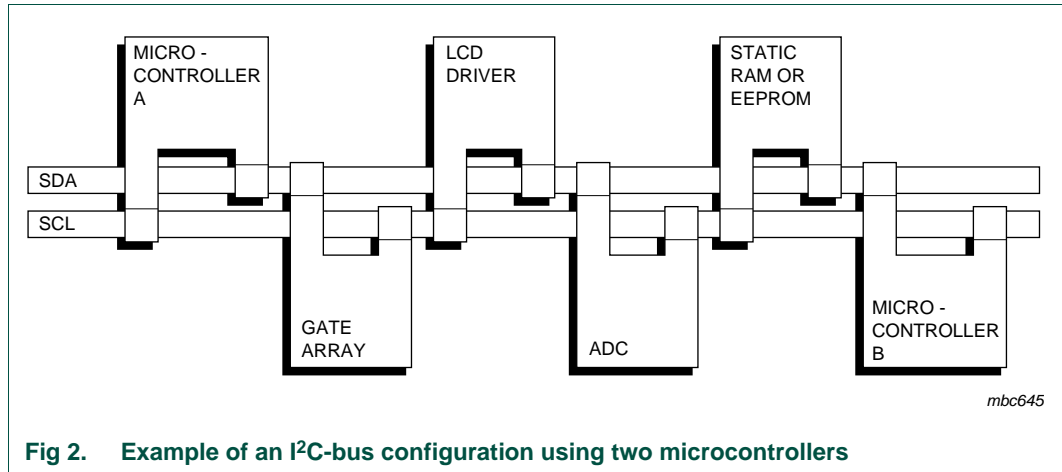
3.1 Standard-mode, Fast-mode and Fast-mode Plus I²C-bus protocols

Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus. Each device is recognized by a unique address (whether it is a microcontroller, LCD driver, memory or keyboard interface) and can operate as either a transmitter or receiver, depending on the function of the device. An LCD driver may be only a receiver, whereas a memory can both receive and transmit data. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers (see [Table 1](#)). A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.

Table 1. Definition of I²C-bus terminology

Term	Description
Transmitter	the device which sends data to the bus
Receiver	the device which receives data from the bus
Master	the device which initiates a transfer, generates clock signals and terminates a transfer
Slave	the device addressed by a master
Multi-master	more than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	procedure to synchronize the clock signals of two or more devices

The I²C-bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. As masters are usually microcontrollers, let us consider the case of a data transfer between two microcontrollers connected to the I²C-bus (see [Figure 2](#)).



This example highlights the master-slave and receiver-transmitter relationships found on the I²C-bus. Note that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows:

1. Suppose microcontroller A wants to send information to microcontroller B:
 - microcontroller A (master), addresses microcontroller B (slave)
 - microcontroller A (master-transmitter), sends data to microcontroller B (slave-receiver)
 - microcontroller A terminates the transfer.
2. If microcontroller A wants to receive information from microcontroller B:
 - microcontroller A (master) addresses microcontroller B (slave)
 - microcontroller A (master-receiver) receives data from microcontroller B (slave-transmitter)
 - microcontroller A terminates the transfer.

Even in this case, the master (microcontroller A) generates the timing and terminates the transfer.

The possibility of connecting more than one microcontroller to the I²C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid the chaos that might ensue from such an event, an arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I²C interfaces to the I²C-bus.

If two or more masters try to put information onto the bus, the first to produce a 'one' when the other produces a 'zero' loses the arbitration. The clock signals during arbitration are a synchronized combination of the clocks generated by the masters using the wired-AND connection to the SCL line (for more detailed information concerning arbitration see [Section 3.1.8](#)).

Generation of clock signals on the I²C-bus is always the responsibility of master devices; each master generates its own clock signals when transferring data on the bus. Bus clock signals from a master can only be altered when they are stretched by a slow slave device holding down the clock line or by another master when arbitration occurs.

[Table 2](#) summarizes the use of mandatory and optional portions of the I²C-bus specification and which system configurations use them.

Table 2. Applicability of I²C-bus protocol features*M = mandatory; O = optional; n/a = not applicable.*

Feature	Configuration		
	Single master	Multi-master	Slave ^[1]
START condition	M	M	M
STOP condition	M	M	M
Acknowledge	M	M	M
Synchronization	n/a	M	n/a
Arbitration	n/a	M	n/a
Clock stretching	O ^[2]	O ^[2]	O
7-bit slave address	M	M	M
10-bit slave address	O	O	O
General Call address	O	O	O
Software Reset	O	O	O
START byte	n/a	O ^[3]	n/a
Device ID	n/a	n/a	O

[1] Also refers to a master acting as a slave.

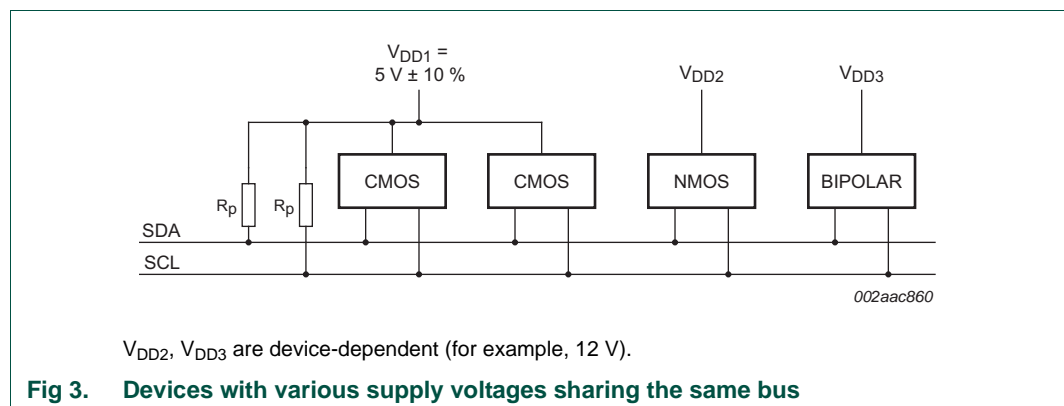
[2] Clock stretching is a feature of some slaves. If no slaves in a system can stretch the clock (hold SCL LOW), the master need not be designed to handle this procedure.

[3] 'Bit banging' (software emulation) multi-master systems should consider a START byte. See [Section 3.1.15](#).

3.1.1 SDA and SCL signals

Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a current-source or pull-up resistor (see [Figure 3](#)). When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function. Data on the I²C-bus can be transferred at rates of up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode. The bus capacitance limits the number of interfaces connected to the bus.

For a single master application, the master's SCL output can be a push-pull driver design if there are no devices on the bus which would stretch the clock.



3.1.2 SDA and SCL logic levels

Due to the variety of different technology devices (CMOS, NMOS, bipolar) that can be connected to the I²C-bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the associated level of V_{DD} . Input reference levels are set as 30 % and 70 % of V_{DD} ; V_{IL} is $0.3V_{DD}$ and V_{IH} is $0.7V_{DD}$. See [Figure 38](#), timing diagram. Some legacy device input levels were fixed at $V_{IL} = 1.5$ V and $V_{IH} = 3.0$ V, but all new devices require this 30 %/70 % specification. See [Section 6](#) for electrical specifications.

3.1.3 Data validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see [Figure 4](#)). One clock pulse is generated for each data bit transferred.

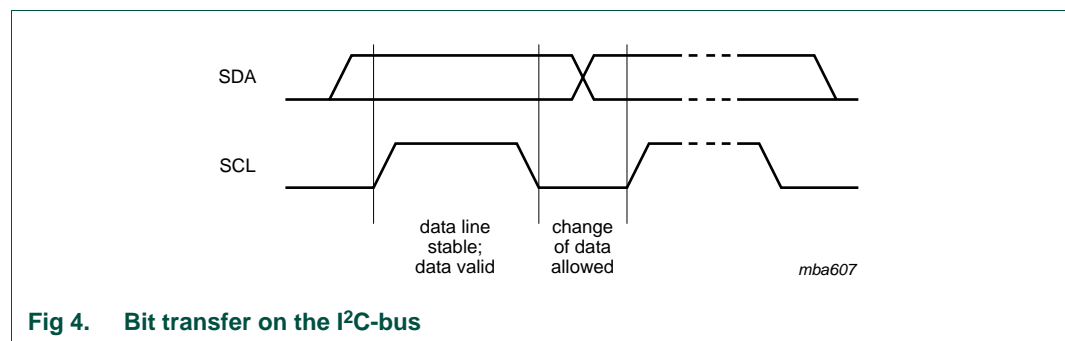


Fig 4. Bit transfer on the I²C-bus

3.1.4 START and STOP conditions

All transactions begin with a START (S) and are terminated by a STOP (P) (see [Figure 5](#)). A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

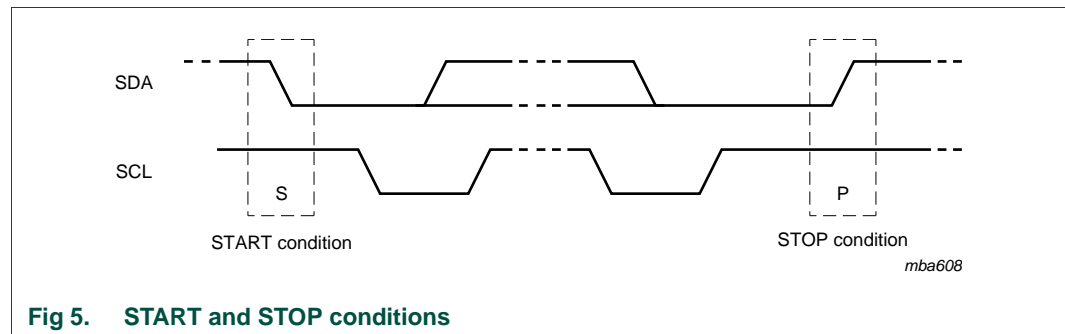


Fig 5. START and STOP conditions

START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. This bus free situation is specified in [Section 6](#).

The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical. For the remainder of this document, therefore, the S symbol is used as a generic term to represent both the START and repeated START conditions, unless Sr is particularly relevant.

Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period to sense the transition.

3.1.5 Byte format

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge bit. Data is transferred with the Most Significant Bit (MSB) first (see [Figure 6](#)). If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

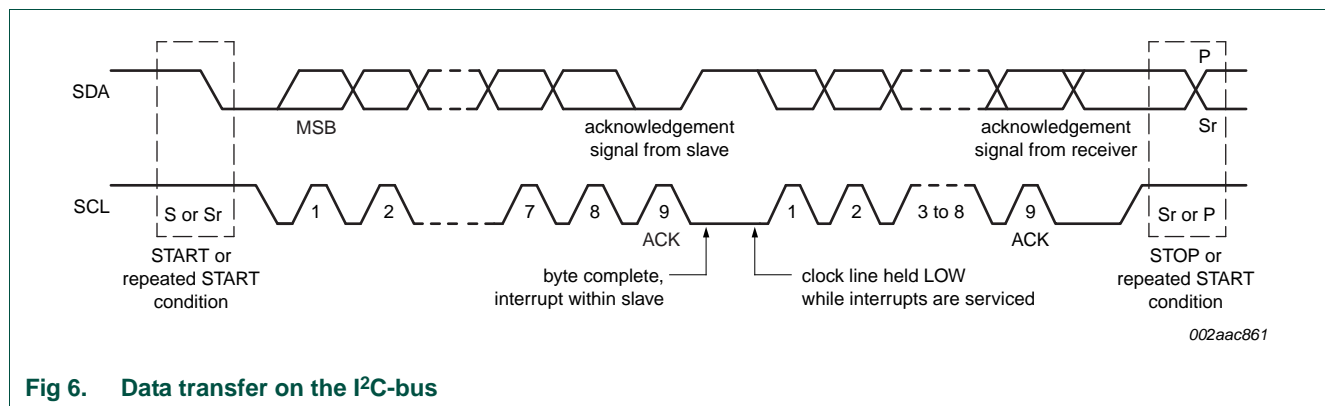


Fig 6. Data transfer on the I²C-bus

3.1.6 Acknowledge (ACK) and Not Acknowledge (NACK)

The acknowledge takes place after every byte. The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. The master generates all clock pulses, including the acknowledge ninth clock pulse.

The Acknowledge signal is defined as follows: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse (see [Figure 4](#)). Set-up and hold times (specified in [Section 6](#)) must also be taken into account.

When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal. The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer. There are five conditions that lead to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

3.1.7 Clock synchronization

Two masters can begin transmitting on a free bus at the same time and there must be a method for deciding which takes control of the bus and complete its transmission. This is done by clock synchronization and arbitration. In single master systems, clock synchronization and arbitration are not needed.

Clock synchronization is performed using the wired-AND connection of I²C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line causes the masters concerned to start counting off their LOW period and, once a master clock has gone LOW, it holds the SCL line in that state until the clock HIGH state is reached (see [Figure 7](#)). However, if another clock is still within its LOW period, the LOW to HIGH transition of this clock may not change the state of the SCL line. The SCL line is therefore held LOW by the master with the longest LOW period. Masters with shorter LOW periods enter a HIGH wait-state during this time.

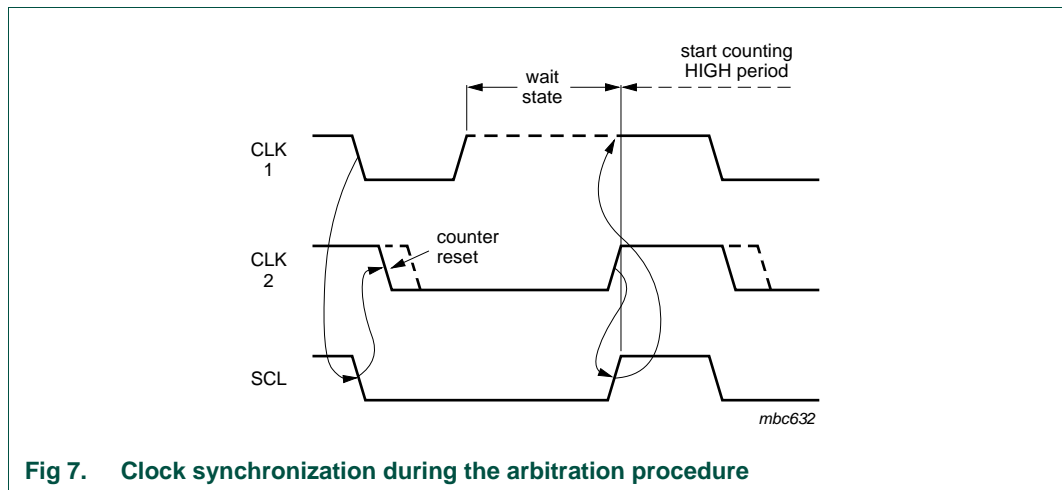


Fig 7. Clock synchronization during the arbitration procedure

When all masters concerned have counted off their LOW period, the clock line is released and goes HIGH. There is then no difference between the master clocks and the state of the SCL line, and all the masters start counting their HIGH periods. The first master to complete its HIGH period pulls the SCL line LOW again.

In this way, a synchronized SCL clock is generated with its LOW period determined by the master with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period.

3.1.8 Arbitration

Arbitration, like synchronization, refers to a portion of the protocol required only if more than one master is used in the system. Slaves are not involved in the arbitration procedure. A master may start a transfer only if the bus is free. Two masters may generate a START condition within the minimum hold time ($t_{HD,STA}$) of the START condition which results in a valid START condition on the bus. Arbitration is then required to determine which master will complete its transmission.

Arbitration proceeds bit by bit. During every bit, while SCL is HIGH, each master checks to see if the SDA level matches what it has sent. This process may take many bits. Two masters can actually complete an entire transaction without error, as long as the

transmissions are identical. The first time a master tries to send a HIGH, but detects that the SDA level is LOW, the master knows that it has lost the arbitration and turns off its SDA output driver. The other master goes on to complete its transaction.

No information is lost during the arbitration process. A master that loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration and must restart its transaction when the bus is free.

If a master also incorporates a slave function and it loses arbitration during the addressing stage, it is possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave mode.

[Figure 8](#) shows the arbitration procedure for two masters. More may be involved depending on how many masters are connected to the bus. The moment there is a difference between the internal data level of the master generating DATA1 and the actual level on the SDA line, the DATA1 output is switched off. This does not affect the data transfer initiated by the winning master.

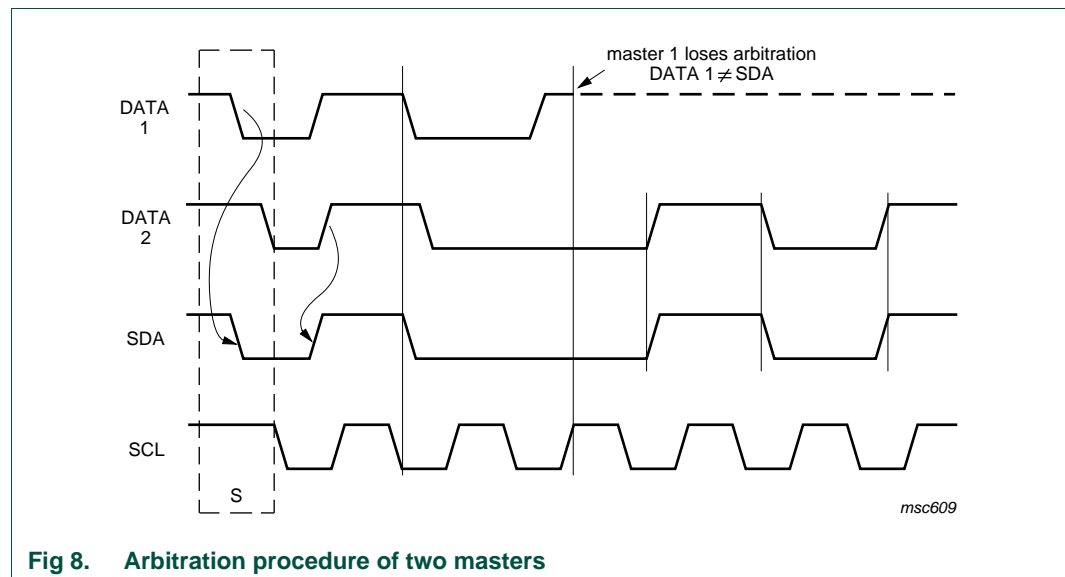


Fig 8. Arbitration procedure of two masters

Since control of the I²C-bus is decided solely on the address and data sent by competing masters, there is no central master, nor any order of priority on the bus.

There is an undefined condition if the arbitration procedure is still in progress at the moment when one master sends a repeated START or a STOP condition while the other master is still sending data. In other words, the following combinations result in an undefined condition:

- Master 1 sends a repeated START condition and master 2 sends a data bit.
- Master 1 sends a STOP condition and master 2 sends a data bit.
- Master 1 sends a repeated START condition and master 2 sends a STOP condition.

3.1.9 Clock stretching

Clock stretching pauses a transaction by holding the SCL line LOW. The transaction cannot continue until the line is released HIGH again. Clock stretching is optional and in fact, most slave devices do not include an SCL driver so they are unable to stretch the clock.

On the byte level, a device may be able to receive bytes of data at a fast rate, but needs more time to store a received byte or prepare another byte to be transmitted. Slaves can then hold the SCL line LOW after reception and acknowledgment of a byte to force the master into a wait state until the slave is ready for the next byte transfer in a type of handshake procedure (see [Figure 7](#)).

On the bit level, a device such as a microcontroller with or without limited hardware for the I²C-bus, can slow down the bus clock by extending each clock LOW period. The speed of any master is adapted to the internal operating rate of this device.

In Hs-mode, this handshake feature can only be used on byte level (see [Section 5.3.2](#)).

3.1.10 The slave address and R/W bit

Data transfers follow the format shown in [Figure 9](#). After the START condition (S), a slave address is sent. This address is seven bits long followed by an eighth bit which is a data direction bit (R/W) — a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ) (refer to [Figure 10](#)). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. Various combinations of read/write formats are then possible within such a transfer.

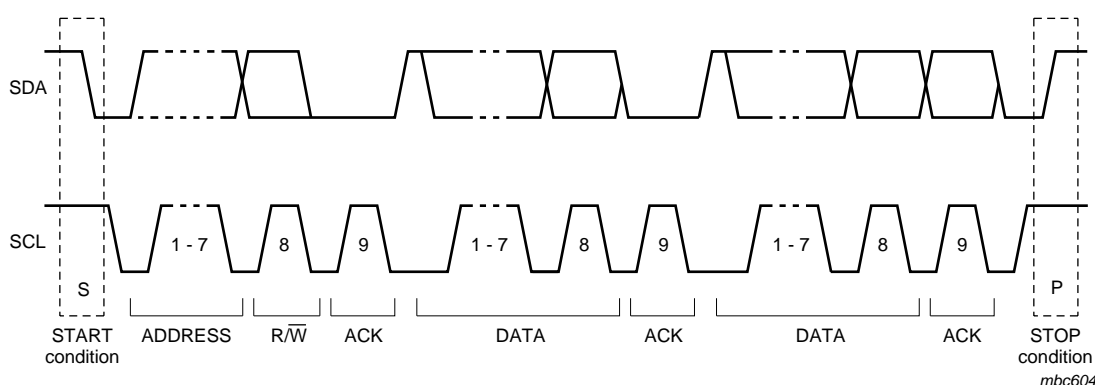


Fig 9. A complete data transfer



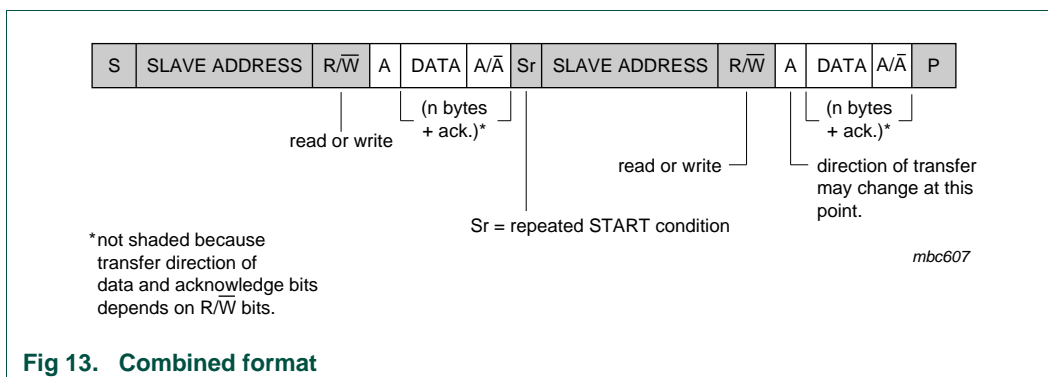
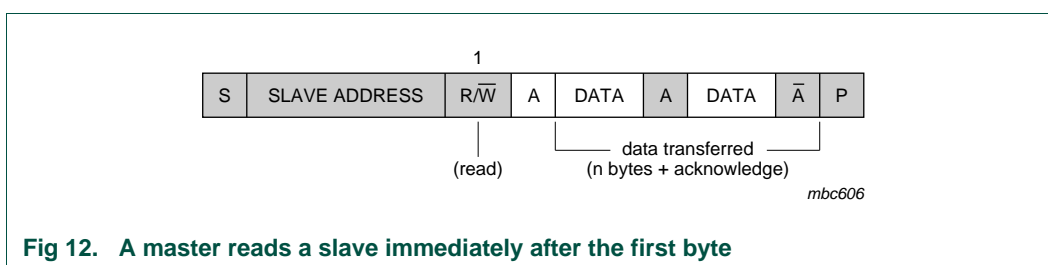
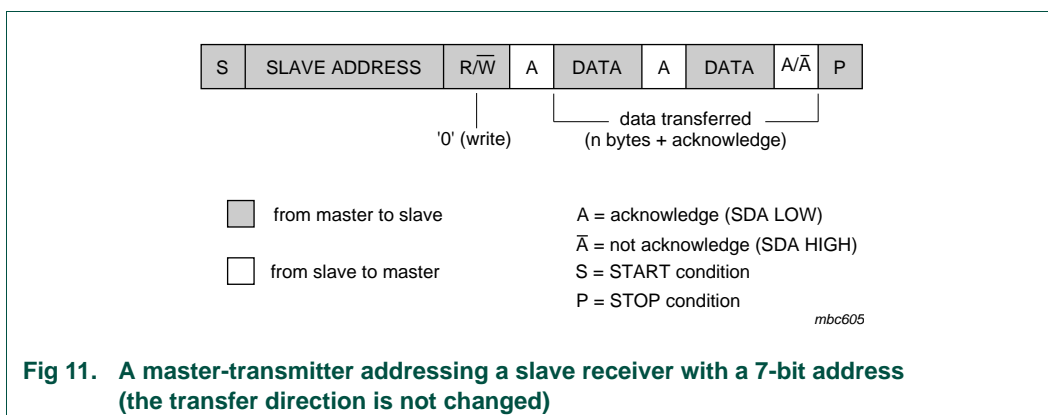
Fig 10. The first byte after the START procedure

Possible data transfer formats are:

- Master-transmitter transmits to slave-receiver. The transfer direction is not changed (see [Figure 11](#)). The slave receiver acknowledges each byte.
- Master reads slave immediately after first byte (see [Figure 12](#)). At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This first acknowledge is still generated by the slave. The master generates subsequent acknowledges. The STOP condition is generated by the master, which sends a not-acknowledge (\bar{A}) just before the STOP condition.
- Combined format (see [Figure 13](#)). During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed. If a master-receiver sends a repeated START condition, it sends a not-acknowledge (\bar{A}) just before the repeated START condition.

Notes:

1. Combined formats can be used, for example, to control a serial memory. The internal memory location must be written during the first data byte. After the START condition and slave address is repeated, data can be transferred.
2. All decisions on auto-increment or decrement of previously accessed memory locations, etc., are taken by the designer of the device.
3. Each byte is followed by an acknowledgment bit as indicated by the A or \bar{A} blocks in the sequence.
4. I²C-bus compatible devices must reset their bus logic on receipt of a START or repeated START condition such that they all anticipate the sending of a slave address, even if these START conditions are not positioned according to the proper format.
5. A START condition immediately followed by a STOP condition (void message) is an illegal format. Many devices however are designed to operate properly under this condition.
6. Each device connected to the bus is addressable by a unique address. Normally a simple master/slave relationship exists, but it is possible to have multiple identical slaves that can receive and respond simultaneously, for example in a group broadcast. This technique works best when using bus switching devices like the PCA9546A where all four channels are on and identical devices are configured at the same time, understanding that it is impossible to determine that each slave acknowledges, and then turn on one channel at a time to read back each individual device's configuration to confirm the programming. Refer to individual component data sheets.



3.1.11 10-bit addressing

10-bit addressing expands the number of possible addresses. Devices with 7-bit and 10-bit addresses can be connected to the same I²C-bus, and both 7-bit and 10-bit addressing can be used in all bus speed modes. Currently, 10-bit addressing is not being widely used.

The 10-bit slave address is formed from the first two bytes following a START condition (S) or a repeated START condition (Sr).

The first seven bits of the first byte are the combination 1111 0XX of which the last two bits (XX) are the two Most-Significant Bits (MSB) of the 10-bit address; the eighth bit of the first byte is the R/W bit that determines the direction of the message.

Although there are eight possible combinations of the reserved address bits 1111 XXX, only the four combinations 1111 0XX are used for 10-bit addressing. The remaining four combinations 1111 1XX are reserved for future I²C-bus enhancements.

All combinations of read/write formats previously described for 7-bit addressing are possible with 10-bit addressing. Two are detailed here:

- Master-transmitter transmits to slave-receiver with a 10-bit slave address.
The transfer direction is not changed (see [Figure 14](#)). When a 10-bit address follows a START condition, each slave compares the first seven bits of the first byte of the slave address (1111 0XX) with its own address and tests if the eighth bit ($\overline{R/W}$ direction bit) is 0. It is possible that more than one device finds a match and generate an acknowledge (A1). All slaves that found a match compare the eight bits of the second byte of the slave address (XXXX XXXX) with their own addresses, but only one slave finds a match and generates an acknowledge (A2). The matching slave remains addressed by the master until it receives a STOP condition (P) or a repeated START condition (Sr) followed by a different slave address.
- Master-receiver reads slave-transmitter with a 10-bit slave address.
The transfer direction is changed after the second $\overline{R/W}$ bit ([Figure 15](#)). Up to and including acknowledge bit A2, the procedure is the same as that described for a master-transmitter addressing a slave-receiver. After the repeated START condition (Sr), a matching slave remembers that it was addressed before. This slave then checks if the first seven bits of the first byte of the slave address following Sr are the same as they were after the START condition (S), and tests if the eighth ($\overline{R/W}$) bit is 1. If there is a match, the slave considers that it has been addressed as a transmitter and generates acknowledge A3. The slave-transmitter remains addressed until it receives a STOP condition (P) or until it receives another repeated START condition (Sr) followed by a different slave address. After a repeated START condition (Sr), all the other slave devices will also compare the first seven bits of the first byte of the slave address (1111 0XX) with their own addresses and test the eighth ($\overline{R/W}$) bit. However, none of them will be addressed because $\overline{R/W} = 1$ (for 10-bit devices), or the 1111 0XX slave address (for 7-bit devices) does not match.



Fig 14. A master-transmitter addresses a slave-receiver with a 10-bit address

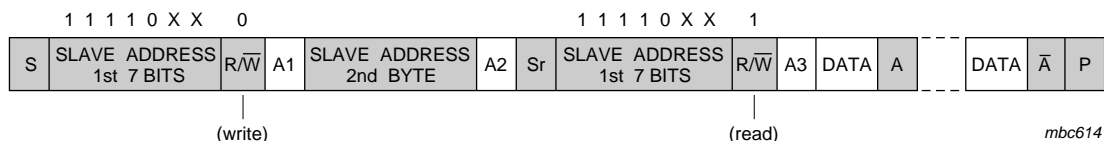


Fig 15. A master-receiver addresses a slave-transmitter with a 10-bit address

Slave devices with 10-bit addressing react to a 'general call' in the same way as slave devices with 7-bit addressing. Hardware masters can transmit their 10-bit address after a 'general call'. In this case, the 'general call' address byte is followed by two successive bytes containing the 10-bit address of the master-transmitter. The format is as shown in [Figure 15](#) where the first DATA byte contains the eight least-significant bits of the master address.

The START byte 0000 0001 (01h) can precede the 10-bit addressing in the same way as for 7-bit addressing (see [Section 3.1.15](#)).

3.1.12 Reserved addresses

Two groups of eight addresses (0000 XXX and 1111 XXX) are reserved for the purposes shown in [Table 3](#).

Table 3. Reserved addresses

X = don't care; 1 = HIGH; 0 = LOW.

Slave address	R/W bit	Description
0000 000	0	general call address ^[1]
0000 000	1	START byte ^[2]
0000 001	X	CBUS address ^[3]
0000 010	X	reserved for different bus format ^[4]
0000 011	X	reserved for future purposes
0000 1XX	X	Hs-mode master code
1111 1XX	1	device ID
1111 0XX	X	10-bit slave addressing

- [1] The general call address is used for several functions including software reset.
- [2] No device is allowed to acknowledge at the reception of the START byte.
- [3] The CBUS address has been reserved to enable the inter-mixing of CBUS compatible and I²C-bus compatible devices in the same system. I²C-bus compatible devices are not allowed to respond on reception of this address.
- [4] The address reserved for a different bus format is included to enable I²C and other protocols to be mixed. Only I²C-bus compatible devices that can work with such formats and protocols are allowed to respond to this address.

Assignment of addresses within a local system is up to the system architect who must take into account the devices being used on the bus and any future interaction with other conventional I²C-buses. For example, a device with seven user-assignable address pins allows all 128 addresses to be assigned. If it is known that the reserved address is never going to be used for its intended purpose, a reserved address can be used for a slave address.

3.1.13 General call address

The general call address is for addressing every device connected to the I²C-bus at the same time. However, if a device does not need any of the data supplied within the general call structure, it can ignore this address by not issuing an acknowledgment. If a device does require data from a general call address, it acknowledges this address and behave as a slave-receiver. The master does not actually know how many devices acknowledged if one or more devices respond. The second and following bytes are acknowledged by every slave-receiver capable of handling this data. A slave who cannot process one of these bytes must ignore it by not-acknowledging. Again, if one or more slaves acknowledge, the not-acknowledge will not be seen by the master. The meaning of the general call address is always specified in the second byte (see [Figure 16](#)).

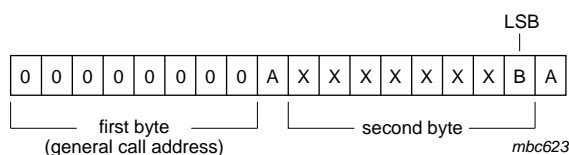


Fig 16. General call address format

There are two cases to consider:

- When the least significant bit B is a 'zero'.
- When the least significant bit B is a 'one'.

When bit B is a 'zero', the second byte has the following definition:

- **0000 0110 (06h): Reset and write programmable part of slave address by hardware.** On receiving this 2-byte sequence, all devices designed to respond to the general call address reset and take in the programmable part of their address. Precautions must be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus.
- **0000 0100 (04h): Write programmable part of slave address by hardware.** Behaves as above, but the device does not reset.
- **0000 0000 (00h): This code is not allowed to be used as the second byte.**

Sequences of programming procedure are published in the appropriate device data sheets. The remaining codes have not been fixed and devices must ignore them.

When bit B is a 'one', the 2-byte sequence is a 'hardware general call'. This means that the sequence is transmitted by a hardware master device, such as a keyboard scanner, which can be programmed to transmit a desired slave address. Since a hardware master does not know in advance to which device the message has to be transferred, it can only generate this hardware general call and its own address — identifying itself to the system (see [Figure 17](#)).

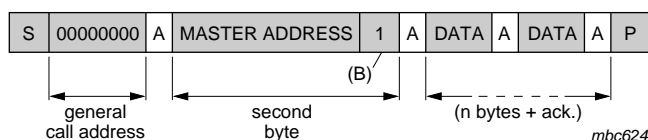
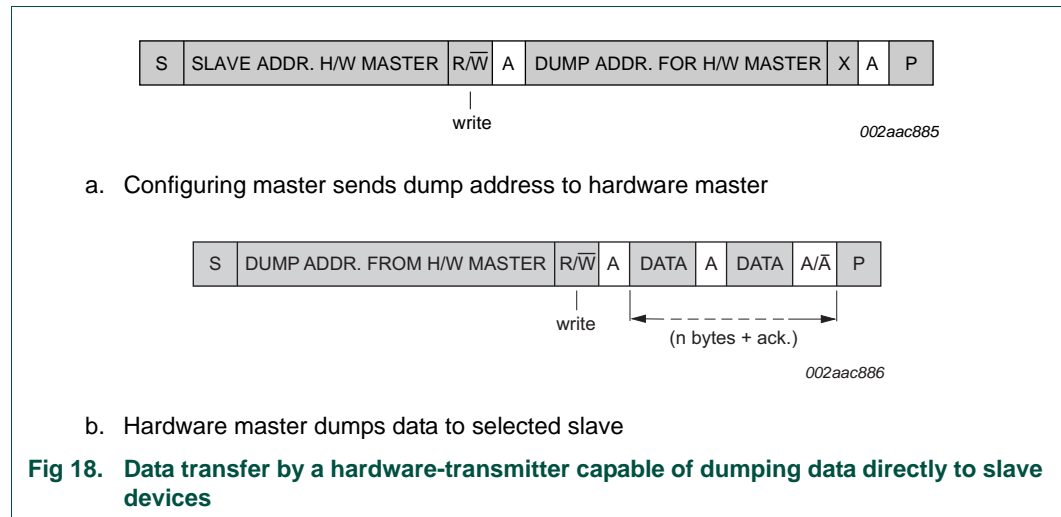


Fig 17. Data transfer from a hardware master-transmitter

The seven bits remaining in the second byte contain the address of the hardware master. This address is recognized by an intelligent device (for example, a microcontroller) connected to the bus which then accepts the information from the hardware master. If the hardware master can also act as a slave, the slave address is identical to the master address.

In some systems, an alternative could be that the hardware master transmitter is set in the slave-receiver mode after the system reset. In this way, a system configuring master can tell the hardware master-transmitter (which is now in slave-receiver mode) to which address data must be sent (see [Figure 18](#)). After this programming procedure, the hardware master remains in the master-transmitter mode.



3.1.14 Software reset

Following a General Call, (0000 0000), sending 0000 0110 (06h) as the second byte causes a software reset. This feature is optional and not all devices respond to this command. On receiving this 2-byte sequence, all devices designed to respond to the general call address reset and take in the programmable part of their address. Precautions must be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus.

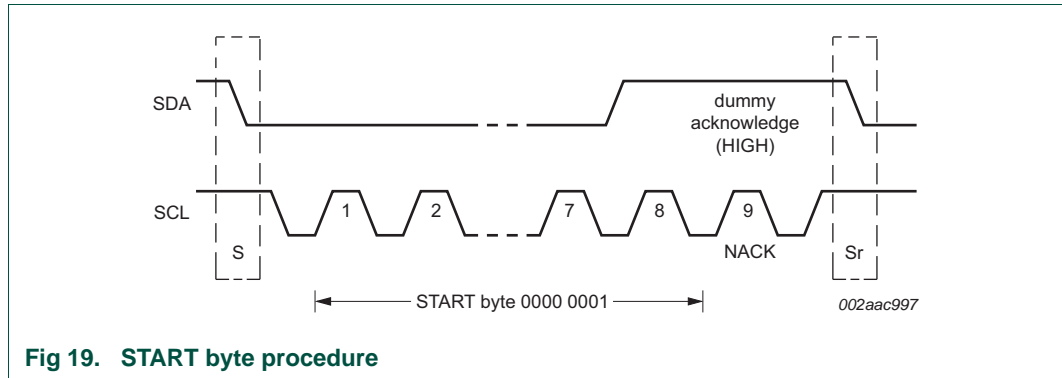
3.1.15 START byte

Microcontrollers can be connected to the I²C-bus in two ways. A microcontroller with an on-chip hardware I²C-bus interface can be programmed to be only interrupted by requests from the bus. When the device does not have such an interface, it must constantly monitor the bus via software. Obviously, the more times the microcontroller monitors, or polls the bus, the less time it can spend carrying out its intended function.

There is therefore a speed difference between fast hardware devices and a relatively slow microcontroller which relies on software polling.

In this case, data transfer can be preceded by a start procedure which is much longer than normal (see [Figure 19](#)). The start procedure consists of:

- A START condition (S)
- A START byte (0000 0001)
- An acknowledge clock pulse (ACK)
- A repeated START condition (Sr).



After the START condition S has been transmitted by a master which requires bus access, the START byte (0000 0001) is transmitted. Another microcontroller can therefore sample the SDA line at a low sampling rate until one of the seven zeros in the START byte is detected. After detection of this LOW level on the SDA line, the microcontroller can switch to a higher sampling rate to find the repeated START condition Sr which is then used for synchronization.

A hardware receiver resets upon receipt of the repeated START condition Sr and therefore ignores the START byte.

An acknowledge-related clock pulse is generated after the START byte. This is present only to conform with the byte handling format used on the bus. No device is allowed to acknowledge the START byte.

3.1.16 Bus clear

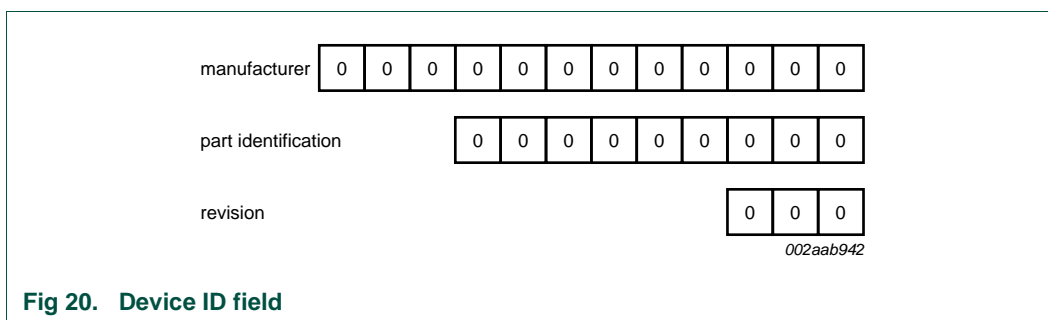
In the unlikely event where the clock (SCL) is stuck LOW, the preferential procedure is to reset the bus using the HW reset signal if your I²C devices have HW reset inputs. If the I²C devices do not have HW reset inputs, cycle power to the devices to activate the mandatory internal Power-On Reset (POR) circuit.

If the data line (SDA) is stuck LOW, the master should send nine clock pulses. The device that held the bus LOW should release it sometime within those nine clocks. If not, then use the HW reset or cycle power to clear the bus.

3.1.17 Device ID

The Device ID field (see [Figure 20](#)) is an optional 3-byte read-only (24 bits) word giving the following information:

- Twelve bits with the manufacturer name, unique per manufacturer (for example, NXP)
- Nine bits with the part identification, assigned by manufacturer (for example, PCA9698)
- Three bits with the die revision, assigned by manufacturer (for example, RevX)



The Device ID is read-only, hard-wired in the device and can be accessed as follows:

1. START condition
2. The master sends the Reserved Device ID I²C-bus address followed by the $\overline{R/W}$ bit set to '0' (write): '1111 1000'.
3. The master sends the I²C-bus slave address of the slave device it must identify. The LSB is a 'Don't care' value. Only one device must acknowledge this byte (the one that has the I²C-bus slave address).
4. The master sends a Re-START condition.

Remark: A STOP condition followed by a START condition resets the slave state machine and the Device ID Read cannot be performed. Also, a STOP condition or a Re-START condition followed by an access to another slave device resets the slave state machine and the Device ID Read cannot be performed.

5. The master sends the Reserved Device ID I²C-bus address followed by the $\overline{R/W}$ bit set to '1' (read): '1111 1001'.
6. The Device ID Read can be done, starting with the 12 manufacturer bits (first byte + four MSBs of the second byte), followed by the nine part identification bits (four LSBs of the second byte + five MSBs of the third byte), and then the three die revision bits (three LSBs of the third byte).
7. The master ends the reading sequence by NACKing the last byte, thus resetting the slave device state machine and allowing the master to send the STOP condition.

Remark: The reading of the Device ID can be stopped anytime by sending a NACK.

If the master continues to ACK the bytes after the third byte, the slave rolls back to the first byte and keeps sending the Device ID sequence until a NACK has been detected.

Table 4. Assigned manufacturer IDs

Manufacturer bits												Company
11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	0	0	NXP Semiconductors
0	0	0	0	0	0	0	0	0	0	0	1	NXP Semiconductors (reserved)
0	0	0	0	0	0	0	0	0	0	1	0	NXP Semiconductors (reserved)
0	0	0	0	0	0	0	0	0	0	1	1	NXP Semiconductors (reserved)
0	0	0	0	0	0	0	0	0	1	0	0	Ramtron International
0	0	0	0	0	0	0	0	0	1	0	1	Analog Devices
0	0	0	0	0	0	0	0	0	1	1	0	STMicroelectronics
0	0	0	0	0	0	0	0	0	1	1	1	ON Semiconductor
0	0	0	0	0	0	0	0	1	0	0	0	Sprintek Corporation
0	0	0	0	0	0	0	0	1	0	0	1	ESPROS Photonics AG
0	0	0	0	0	0	0	0	1	0	1	0	Fujitsu Semiconductor
0	0	0	0	0	0	0	0	1	0	1	1	Flir
0	0	0	0	0	0	0	0	1	1	0	0	O ₂ Micro
0	0	0	0	0	0	0	0	1	1	0	1	Atmel

Designers of new I²C devices who want to implement the device ID feature should contact NXP at i2c.support@nxp.com to have a unique manufacturer ID assigned.

3.2 Ultra Fast-mode I²C-bus protocol

The U^{Fm} I²C-bus is a 2-wire push-pull serial bus that operates from DC to 5 MHz transmitting data in one direction. It is most useful for speeds greater than 1 MHz to drive LED controllers and other devices that do not need feedback. The U^{Fm} I²C-bus protocol is based on the standard I²C-bus protocol that consists of a START, slave address, command bit, ninth clock, and a STOP bit. The command bit is a 'write' only, and the data bit on the ninth clock is driven HIGH, ignoring the ACK cycle due to the unidirectional nature of the bus. The 2-wire push-pull driver consists of a U^{Fm} serial clock (USCL) and serial data (USDA).

Slave devices contain a unique address (whether it is a microcontroller, LCD driver, LED controller, GPO) and operate only as receivers. An LED driver may be only a receiver and can be supported by U^{Fm}, whereas a memory can both receive and transmit data and is not supported by U^{Fm}.

Since U^{Fm} I²C-bus uses push-pull drivers, it does not have the multi-master capability of the wired-AND open-drain S_m, F_m, and F_m+ I²C-buses. In U^{Fm}, a master is the only device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. All other devices addressed are considered slaves.

Table 5. Definition of U^{Fm} I²C-bus terminology

Term	Description
Transmitter	the device that sends data to the bus
Receiver	the device that receives data from the bus
Master	the device that initiates a transfer, generates clock signals and terminates a transfer
Slave	the device addressed by a master

Let us consider the case of a data transfer between a master and multiple slaves connected to the U^{Fm} I²C-bus (see [Figure 21](#)).

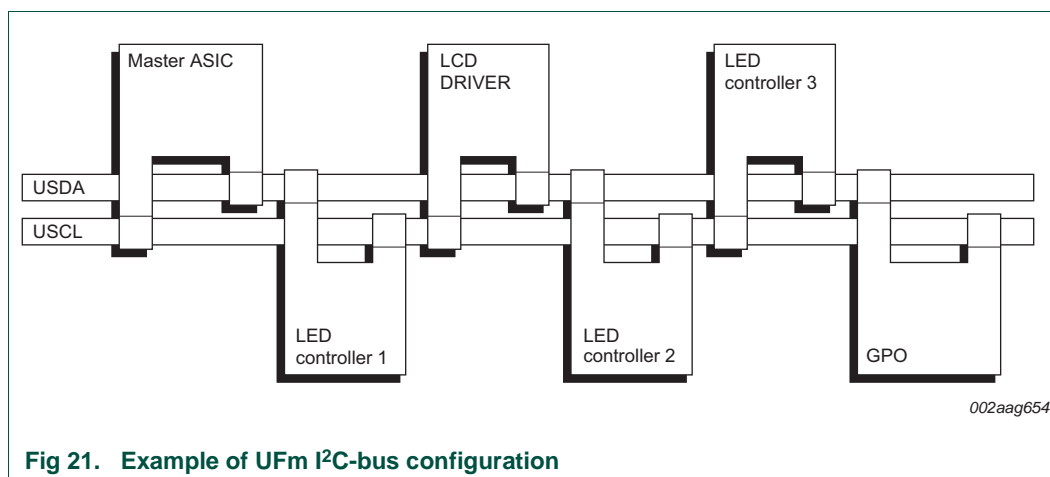


Fig 21. Example of U^{Fm} I²C-bus configuration

This highlights the master/transmitter-slave/receiver relationship found on the U^{Fm} I²C-bus. Note that these relationships are permanent, as data transfer is only permitted in one direction. The transfer of data would proceed as follows:

Suppose that the master ASIC wants to send information to the LED controller 2:

- ASIC A (master-transmitter), addresses LED controller 2 (slave-receiver) by sending the address on the USDA and generating the clock on USCL.
- ASIC A (master-transmitter), sends data to LED controller 2 (slave-receiver) on the USDA and generates the clock on USCL.
- ASIC A terminates the transfer.

The possibility of connecting more than one U^{Fm} master to the U^{Fm} I²C-bus is not allowed due to bus contention on the push-pull outputs. If an additional master is required in the system, it must be fully isolated from the other master (that is, with a true 'one hot' MUX) as only one master is allowed on the bus at a time.

Generation of clock signals on the U^{Fm} I²C-bus is always the responsibility of the master device, that is, the master generates the clock signals when transferring data on the bus. Bus clock signals from a master cannot be altered by a slave device with clock stretching and the process of arbitration and clock synchronization does not exist within the U^{Fm} I²C-bus.

[Table 6](#) summarizes the use of mandatory and optional portions of the U^{Fm} I²C-bus specification.

Table 6. Applicability of I²C-bus features to U^{Fm}

M = mandatory; O = optional; n/p = not possible

Feature	Configuration
	Single master
START condition	M
STOP condition	M
Acknowledge	n/p
Synchronization	n/p
Arbitration	n/p
Clock stretching	n/p
7-bit slave address	M
10-bit slave address	O
General Call address	O
Software Reset	O
START byte	O
Device ID	n/p

3.2.1 USDA and USCL signals

Both USDA and USCL are unidirectional lines, with push-pull outputs. When the bus is free, both lines are pulled HIGH by the upper transistor of the output stage. Data on the I²C-bus can be transferred at rates of up to 5000 kbit/s in the Ultra Fast-mode. The number of interfaces connected to the bus is limited by the bus loading, reflections from cable ends, connectors, and stubs.

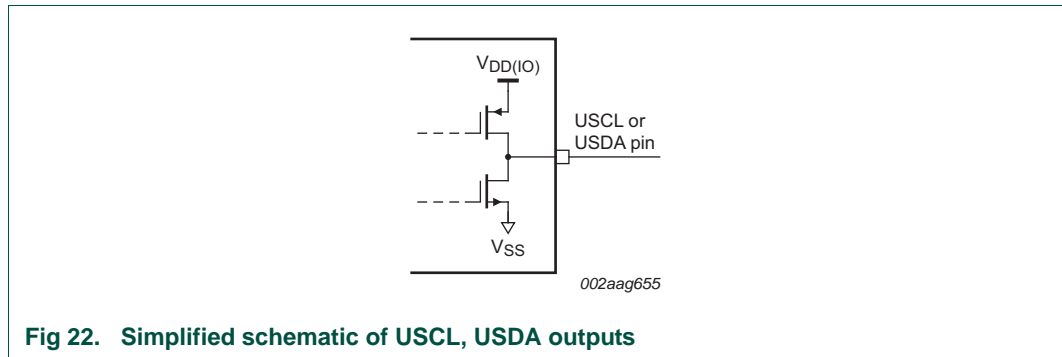


Fig 22. Simplified schematic of USCL, USDA outputs

3.2.2 USDA and USCL logic levels

Due to the variety of different technology devices (CMOS, NMOS, bipolar) that can be connected to the I²C-bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the associated level of V_{DD}. Input reference levels are set as 30 % and 70 % of V_{DD}; V_{IL} is 0.3V_{DD} and V_{IH} is 0.7V_{DD}. See [Figure 40](#), timing diagram. See [Section 6](#) for electrical specifications.

3.2.3 Data validity

The data on the USDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the USCL line is LOW (see [Figure 23](#)). One clock pulse is generated for each data bit transferred.

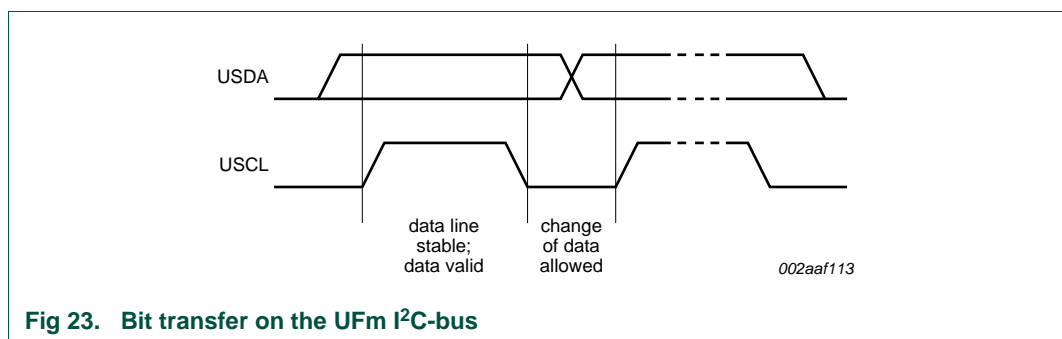
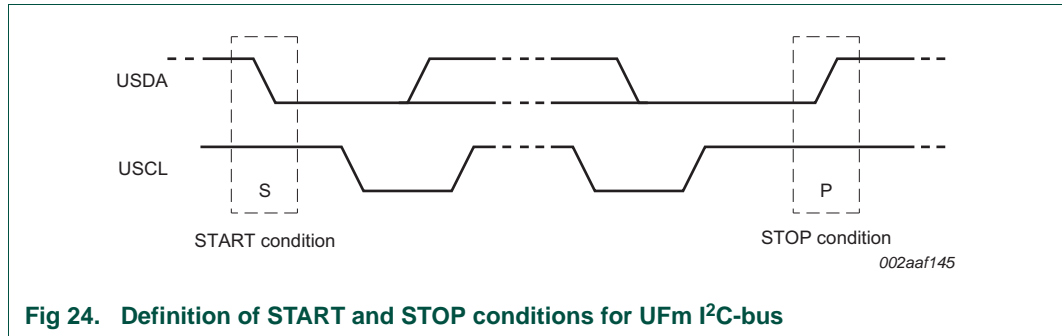


Fig 23. Bit transfer on the UFM I²C-bus

3.2.4 START and STOP conditions

Both data and clock lines remain HIGH when the bus is not busy. All transactions begin with a START (S) and can be terminated by a STOP (P) (see [Figure 24](#)). A HIGH to LOW transition on the USDA line while USCL is HIGH defines a START condition. A LOW to HIGH transition on the USDA line while USCL is HIGH defines a STOP condition.

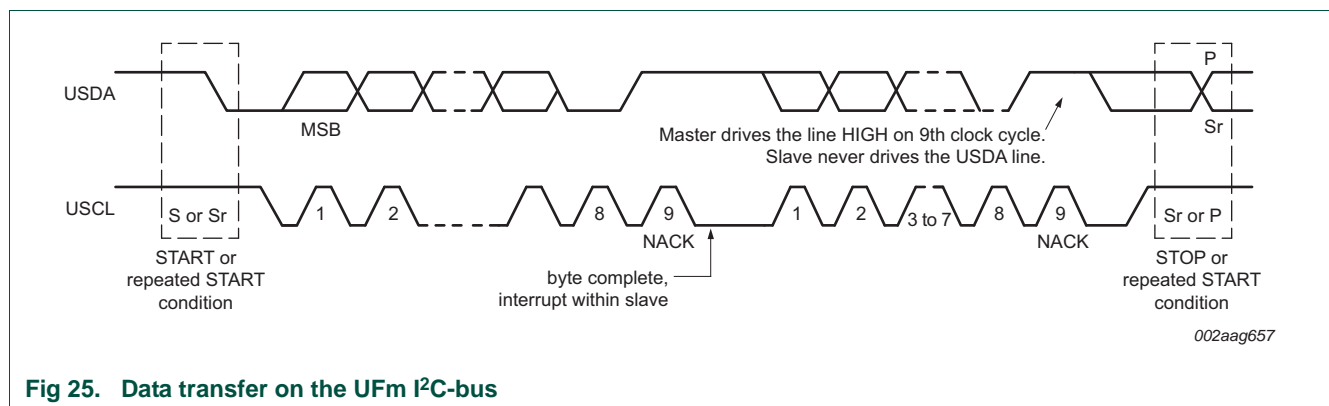


START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. This bus free situation is specified in [Section 6](#). The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical. For the remainder of this document, therefore, the S symbol is used as a generic term to represent both the START and repeated START conditions, unless Sr is particularly relevant.

Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the USDA line at least twice per clock period to sense the transition.

3.2.5 Byte format

Every byte put on the USDA line must be eight bits long. The number of bytes that can be transferred per transfer is unrestricted. The master drives the USDA HIGH after each byte during the Acknowledge cycle. Data is transferred with the Most Significant Bit (MSB) first (see [Figure 25](#)). A slave is not allowed to hold the clock LOW if it cannot receive another complete byte of data or while it is performing some other function, for example servicing an internal interrupt.



3.2.6 Acknowledge (ACK) and Not Acknowledge (NACK)

Since the slaves are not able to respond the ninth clock cycle, the ACK and NACK are not required. However, the clock cycle is preserved in the U^{Fm} to be compatible with the I²C-bus protocol. The ACK and NACK are also referred to as the ninth clock cycle. The master generates all clock pulses, including the ninth clock pulse. The ninth data bit is always driven HIGH ('1'). Slave devices are not allowed to drive the SDA line at any time.

3.2.7 The slave address and R/W bit

Data transfers follow the format shown in [Figure 26](#). After the START condition (S), a slave address is sent. This address is seven bits long followed by an eighth bit which is a data direction bit (\overline{W}) — a 'zero' indicates a transmission (WRITE); a 'one' indicates a request for data (READ) and is not supported by U^{Fm} (except for the START byte, [Section 3.2.12](#)) since the communication is unidirectional (refer to [Figure 27](#)). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition.

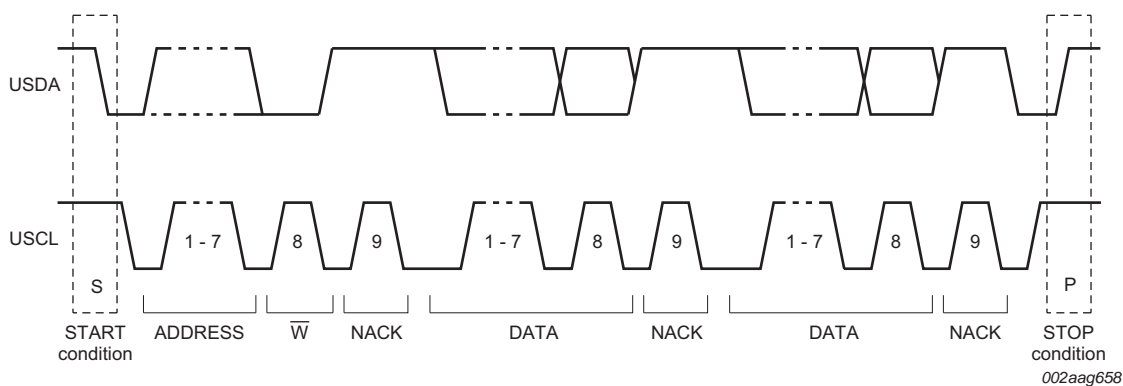


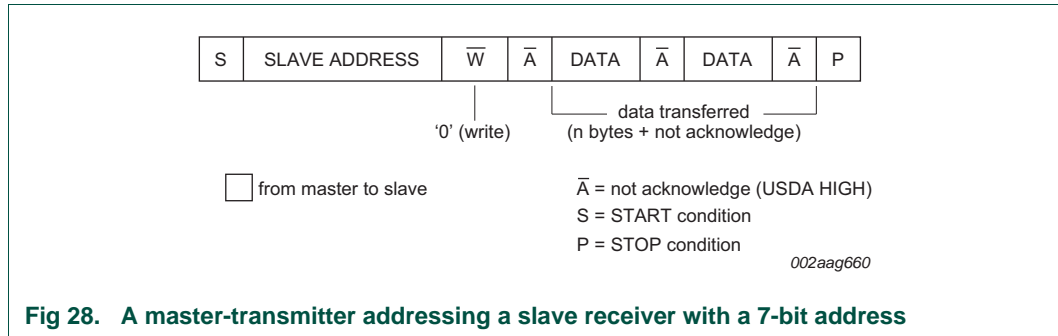
Fig 26. A complete U^{Fm} data transfer



Fig 27. The first byte after the START procedure

The U^{Fm} data transfer format is:

- Master-transmitter transmits to slave-receiver. The transfer direction is not changed (see [Figure 28](#)). The master never acknowledges because it never receives any data but generates the '1' on the ninth bit for the slave to conform to the I²C-bus protocol.

**Notes:**

1. Individual transaction or repeated START formats addressing multiple slaves in one transaction can be used. After the START condition and slave address is repeated, data can be transferred.
2. All decisions on auto-increment or decrement of previously accessed memory locations, etc., are taken by the designer of the device.
3. Each byte is followed by a Not-Acknowledgment bit as indicated by the \overline{A} blocks in the sequence.
4. I²C-bus compatible devices must reset their bus logic on receipt of a START or repeated START condition such that they all anticipate the sending of a slave address, even if these START conditions are not positioned according to the proper format.
5. A START condition immediately followed by a STOP condition (void message) is an illegal format. Many devices however are designed to operate properly under this condition.
6. Each device connected to the bus is addressable by a unique address. A simple master/slave relationship exists, but it is possible to have multiple identical slaves that can receive and respond simultaneously, for example, in a group broadcast where all identical devices are configured at the same time, understanding that it is impossible to determine that each slave is responsive. Refer to individual component data sheets.

3.2.8 10-bit addressing

10-bit addressing expands the number of possible addresses. Devices with 7-bit and 10-bit addresses can be connected to the same I²C-bus, and both 7-bit and 10-bit addressing can be used in all bus speed modes.

The 10-bit slave address is formed from the first two bytes following a START condition (S) or a repeated START condition (Sr). The first seven bits of the first byte are the combination 1111 0XX of which the last two bits (XX) are the two Most Significant Bits (MSBs) of the 10-bit address; the eighth bit of the first byte is the $\overline{R/W}$ bit that determines the direction of the message.

Although there are eight possible combinations of the reserved address bits 1111 XXX, only the four combinations 1111 0XX are used for 10-bit addressing. The remaining four combinations 1111 1XX are reserved for future I²C-bus enhancements.

Only the write format previously described for 7-bit addressing is possible with 10-bit addressing. Detailed here:

- Master-transmitter transmits to slave-receiver with a 10-bit slave address. The transfer direction is not changed (see [Figure 29](#)). When a 10-bit address follows a START condition, each slave compares the first seven bits of the first byte of the slave address (1111 0XX) with its own address and tests if the eighth bit (R/W direction bit) is 0 (\overline{W}). All slaves that found a match compare the eight bits of the second byte of the slave address (XXXX XXXX) with their own addresses, but only one slave finds a match. The matching slave remains addressed by the master until it receives a STOP condition (P) or a repeated START condition (Sr) followed by a different slave address.

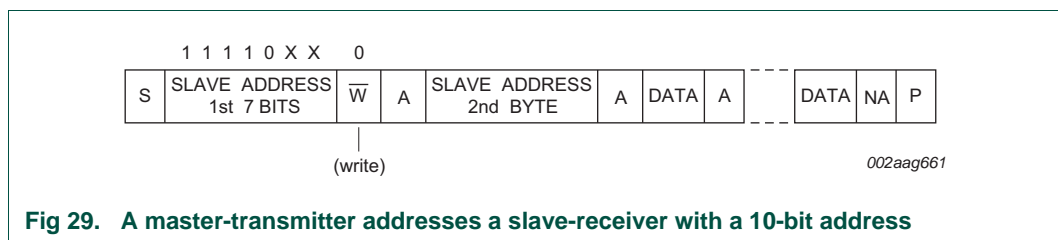


Fig 29. A master-transmitter addresses a slave-receiver with a 10-bit address

The START byte 0000 0001 (01h) can precede the 10-bit addressing in the same way as for 7-bit addressing (see [Section 3.2.12](#)).

3.2.9 Reserved addresses in U^Fm

The U^Fm I²C-bus has a different physical layer than the other I²C-bus modes. Therefore the available slave address range is different. Two groups of eight addresses (0000 XXX and 1111 XXX) are reserved for the purposes shown in [Table 7](#).

Table 7. Reserved addresses

X = don't care; 1 = HIGH; 0 = LOW.

Slave address	R/W bit	Description
0000 000	0	general call address ^[1]
0000 000	1	START byte ^[2]
0000 001	X	reserved for future purposes
0000 010	X	reserved for future purposes
0000 011	X	reserved for future purposes
0000 1XX	X	reserved for future purposes
1111 1XX	X	reserved for future purposes
1111 0XX	X	10-bit slave addressing

[1] The general call address is used for several functions including software reset.

[2] No U^Fm device is allowed to acknowledge at the reception of the START byte.

Assignment of addresses within a local system is up to the system architect who must take into account the devices being used on the bus and any future interaction with reserved addresses. For example, a device with seven user-assignable address pins allows all 128 addresses to be assigned. If it is known that the reserved address is never going to be used for its intended purpose, then a reserved address can be used for a slave address.

3.2.10 General call address

The general call address is for addressing every device connected to the I²C-bus at the same time. However, if a device does not need any of the data supplied within the general call structure, it can ignore this address. If a device does require data from a general call address, it behaves as a slave-receiver. The master does not actually know how many devices are responsive to the general call. The second and following bytes are received by every slave-receiver capable of handling this data. A slave that cannot process one of these bytes must ignore it. The meaning of the general call address is always specified in the second byte (see [Figure 30](#)).

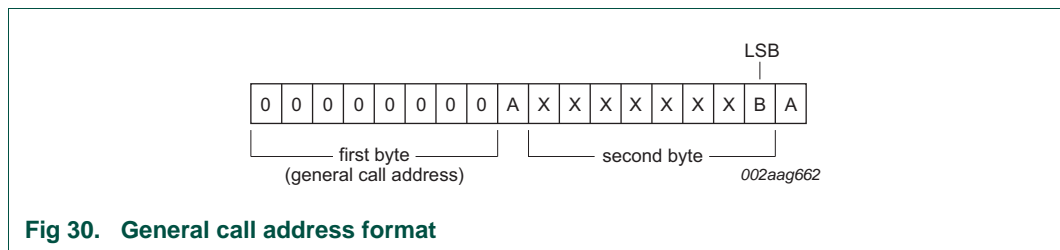


Fig 30. General call address format

There are two cases to consider:

- When the least significant bit B is a 'zero'
- When the least significant bit B is a 'one'

When bit B is a 'zero', the second byte has the following definition:

0000 0110 (06h) — Reset and write programmable part of slave address by hardware. On receiving this 2-byte sequence, all devices designed to respond to the general call address reset and take in the programmable part of their address.

0000 0100 (04h) — Write programmable part of slave address by hardware. Behaves as above, but the device does not reset.

0000 0000 (00h) — This code is not allowed to be used as the second byte. Sequences of programming procedure are published in the appropriate device data sheets. The remaining codes have not been fixed and devices must ignore them.

When bit B is a 'one', the 2-byte sequence is ignored.

3.2.11 Software reset

Following a General Call, (0000 0000), sending 0000 0110 (06h) as the second byte causes a software reset. This feature is optional and not all devices respond to this command. On receiving this 2-byte sequence, all devices designed to respond to the general call address reset and take in the programmable part of their address.

3.2.12 START byte

Microcontrollers can be connected to the I²C-bus in two ways. A microcontroller with an on-chip hardware I²C-bus interface can be programmed to be only interrupted by requests from the bus. When the device does not have such an interface, it must constantly monitor the bus via software. Obviously, the more times the microcontroller monitors, or polls the bus, the less time it can spend carrying out its intended function.

There is therefore a speed difference between fast hardware devices and a relatively slow microcontroller which relies on software polling.

In this case, data transfer can be preceded by a start procedure which is much longer than normal (see [Figure 31](#)). The start procedure consists of:

- A START condition (S)
- A START byte (0000 0001)
- A Not Acknowledge clock pulse (NACK)
- A repeated START condition (Sr)

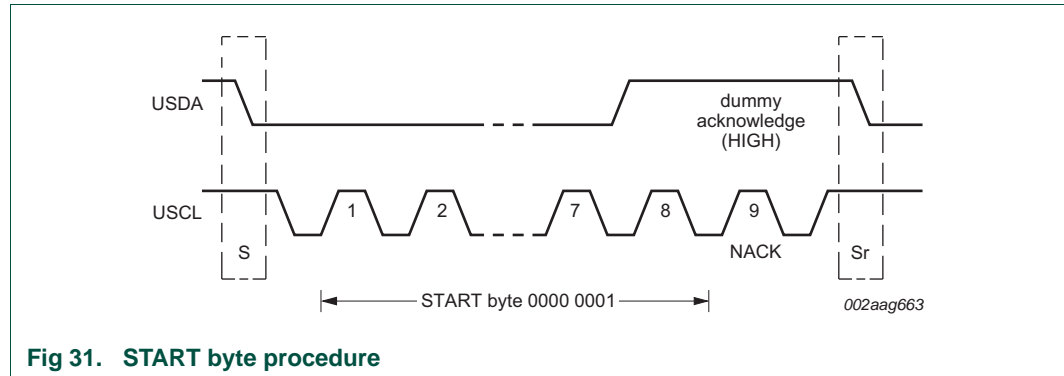


Fig 31. START byte procedure

After the START condition S has been transmitted by a master which requires bus access, the START byte (0000 0001) is transmitted. Another microcontroller can therefore sample the USDA line at a low sampling rate until one of the seven zeros in the START byte is detected. After detection of this LOW level on the USDA line, the microcontroller can switch to a higher sampling rate to find the repeated START condition Sr, which is then used for synchronization. A hardware receiver resets upon receipt of the repeated START condition Sr and therefore ignores the START byte. An acknowledge-related clock pulse is generated after the START byte. This is present only to conform with the byte handling format used on the bus. No device is allowed to acknowledge the START byte.

3.2.13 Unresponsive slave reset

In the unlikely event where the slave becomes unresponsive (for example, determined through external feedback, not through UFM I²C-bus), the preferential procedure is to reset the slave by using the software reset command or the hardware reset signal. If the slaves do not support these features, then cycle power to the devices to activate the mandatory internal Power-On Reset (POR) circuit.

3.2.14 Device ID

The Device ID field is not supported in UFM.

4. Other uses of the I²C-bus communications protocol

The I²C-bus is used as the communications protocol for several system architectures. These architectures have added command sets and application-specific extensions in addition to the base I²C specification. In general, simple I²C-bus devices such as I/O extenders could be used in any one of these architectures since the protocol and physical interfaces are the same.

4.1 CBUS compatibility

CBUS receivers can be connected to the Standard-mode I²C-bus. However, a third bus line called DLEN must then be connected and the acknowledge bit omitted. Normally, I²C transmissions are sequences of 8-bit bytes; CBUS compatible devices have different formats.

In a mixed bus structure, I²C-bus devices must not respond to the CBUS message. For this reason, a special CBUS address (0000 001X) to which no I²C-bus compatible device responds has been reserved. After transmission of the CBUS address, the DLEN line can be made active and a CBUS-format transmission sent. After the STOP condition, all devices are again ready to accept data.

Master-transmitters can send CBUS formats after sending the CBUS address. The transmission is ended by a STOP condition, recognized by all devices.

Remark: If the CBUS configuration is known, and expansion with CBUS compatible devices is not foreseen, the designer is allowed to adapt the hold time to the specific requirements of the device(s) used.

4.2 SMBus - System Management Bus

The SMBus uses I²C hardware and I²C hardware addressing, but adds second-level software for building special systems. In particular, its specifications include an Address Resolution Protocol that can make dynamic address allocations.

Dynamic reconfiguration of the hardware and software allow bus devices to be 'hot-plugged' and used immediately, without restarting the system. The devices are recognized automatically and assigned unique addresses. This advantage results in a plug-and-play user interface. In both those protocols, there is a very useful distinction made between a System Host and all the other devices in the system that can have the names and functions of masters or slaves.

SMBus is used today as a system management bus in most PCs. Developed by Intel and others in 1995, it modified some I²C electrical and software characteristics for better compatibility with the quickly decreasing power supply budget of portable equipment. SMBus also has a 'High Power' version 2.0 that includes a 4 mA sink current that cannot be driven by I²C chips unless the pull-up resistor is sized to I²C-bus levels.

4.2.1 I²C/SMBus compliancy

SMBus and I²C protocols are basically the same: A SMBus master is able to control I²C devices and vice versa at the protocol level. The SMBus clock is defined from 10 kHz to 100 kHz while I²C can be 0 Hz to 100 kHz, 0 Hz to 400 kHz, 0 Hz to 1 MHz and 0 Hz to 3.4 MHz, depending on the mode. This means that an I²C-bus running at less than 10 kHz is not SMBus compliant since the SMBus devices may time-out.

Logic levels are slightly different also: TTL for SMBus: LOW = 0.8 V and HIGH = 2.1 V, versus the 30 %/70 % V_{DD} CMOS level for I²C. This is not a problem if $V_{DD} > 3.0$ V. If the I²C device is below 3.0 V, then there could be a problem if the logic HIGH/LOW levels are not properly recognized.

4.2.2 Time-out feature

SMBus has a time-out feature which resets devices if a communication takes too long. This explains the minimum clock frequency of 10 kHz to prevent locking up the bus. I²C can be a 'DC' bus, meaning that a slave device stretches the master clock when performing some routine while the master is accessing it. This notifies the master that the slave is busy but does not want to lose the communication. The slave device will allow continuation after its task is complete. There is no limit in the I²C-bus protocol as to how long this delay can be, whereas for a SMBus system, it would be limited to 35 ms.

SMBus protocol just assumes that if something takes too long, then it means that there is a problem on the bus and that all devices must reset in order to clear this mode. Slave devices are not then allowed to hold the clock LOW too long.

4.2.3 Differences between SMBus 1.0 and SMBus 2.0

The SMBus specification defines two classes of electrical characteristics: low power and high power. The first class, originally defined in the SMBus 1.0 and 1.1 specifications, was designed primarily with Smart Batteries in mind, but could be used with other low-power devices.

The 2.0 version introduces an alternative higher power set of electrical characteristics. This class is appropriate for use when higher drive capability is required, for example with SMBus devices on PCI add-in cards and for connecting such cards across the PCI connector between each other and to SMBus devices on the system board.

Devices may be powered by the bus V_{DD} or by another power source, V_{BUS} (as with, for example, Smart Batteries), and will inter-operate as long as they adhere to the SMBus electrical specifications for their class.

NXP devices have a higher power set of electrical characteristics than SMBus 1.0. The main difference is the current sink capability with $V_{OL} = 0.4$ V.

- SMBus low power = 350 μ A
- SMBus high power = 4 mA
- I²C-bus = 3 mA

SMBus 'high power' devices and I²C-bus devices will work together if the pull-up resistor is sized for 3 mA.

For more information, refer to: www.smbus.org/.

4.3 PMBus - Power Management Bus

PMBus is a standard way to communicate between power converters and a system host over the SMBus to provide more intelligent control of the power converters. The PMBus specification defines a standard set of device commands so that devices from multiple sources function identically. PMBus devices use the SMBus Version 1.1 plus extensions for transport.

For more information, refer to: www.pmbus.org/.

4.4 Intelligent Platform Management Interface (IPMI)

Intelligent Platform Management Interface (IPMI) defines a standardized, abstracted, message-based interface for intelligent platform management hardware. IPMI also defines standardized records for describing platform management devices and their characteristics. IPMI increases reliability of systems by monitoring parameters such as temperatures, voltages, fans and chassis intrusion.

IPMI provides general system management functions such as automatic alerting, automatic system shutdown and restart, remote restart and power control. The standardized interface to intelligent platform management hardware aids in prediction and early monitoring of hardware failures as well as diagnosis of hardware problems.

This standardized bus and protocol for extending management control, monitoring, and event delivery within the chassis:

- I²C based
- Multi-master
- Simple Request/Response Protocol
- Uses IPMI Command sets
- Supports non-IPMI devices
- Physically I²C but write-only (master capable devices); hot swap not required
- Enables the Baseboard Management Controller (BMC) to accept IPMI request messages from other management controllers in the system
- Allows non-intelligent devices as well as management controllers on the bus
- BMC serves as a controller to give system software access to IPMB.

Hardware implementation is isolated from software implementation so that new sensors and events can then be added without any software changes.

For more information, refer to: www.intel.com/design/servers/ipmi/ipmi.htm.

4.5 Advanced Telecom Computing Architecture (ATCA)

Advanced Telecom Computing Architecture (ATCA) is a follow-on to Compact PCI (cPCI), providing a standardized form-factor with larger card area, larger pitch and larger power supply for use in advanced rack-mounted telecom hardware. It includes a fault-tolerant scheme for thermal management that uses I²C-bus communications between boards.

Advanced Telecom Computing Architecture (ATCA) is backed by more than 100 companies including many of the large players such as Intel, Lucent, and Motorola.

There are two general compliant approaches to an ATCA-compliant fan control: the first is an Intelligent FRU (Field Replaceable Unit) which means that the fan control would be directly connected to the IPMB (Intelligent Platform Management Bus); the second is a Managed or Non-intelligent FRU.

One requirement is the inclusion of hardware and software to manage the dual I²C-buses. This requires an on-board isolated supply to power the circuitry, a buffered dual I²C-bus with rise time accelerators, and 3-state capability. The I²C controller must be able to support a multi-master I²C dual bus and handle the standard set of fan commands outlined in the protocol. In addition, on-board temperature reporting, tray capability reporting, fan turn-off capabilities, and non-volatile storage are required.

For more information, refer to: www.picmg.org/v2internal/resourcepage2.cfm?id=2.

4.6 Display Data Channel (DDC)

The Display Data Channel (DDC) allows a monitor or display to inform the host about its identity and capabilities. The specification for DDC version 2 calls for compliance with the I²C-bus standard mode specification. It allows bidirectional communication between the display and the host, enabling control of monitor functions such as how images are displayed and communication with other devices attached to the I²C-bus.

For more information, refer to: www.vesa.org.

5. Bus speeds

Originally, the I²C-bus was limited to 100 kbit/s operation. Over time there have been several additions to the specification so that there are now five operating speed categories. Standard-mode, Fast-mode (Fm), Fast-mode Plus (Fm+), and High-speed mode (Hs-mode) devices are downward-compatible — any device may be operated at a lower bus speed. Ultra Fast-mode devices are not compatible with previous versions since the bus is unidirectional.

- Bidirectional bus:
 - **Standard-mode (Sm)**, with a bit rate up to 100 kbit/s
 - **Fast-mode (Fm)**, with a bit rate up to 400 kbit/s
 - **Fast-mode Plus (Fm+)**, with a bit rate up to 1 Mbit/s
 - **High-speed mode (Hs-mode)**, with a bit rate up to 3.4 Mbit/s.
- Unidirectional bus:
 - **Ultra Fast-mode (UFm)**, with a bit rate up to 5 Mbit/s

5.1 Fast-mode

Fast-mode devices can receive and transmit at up to 400 kbit/s. The minimum requirement is that they can synchronize with a 400 kbit/s transfer; they can then prolong the LOW period of the SCL signal to slow down the transfer. The protocol, format, logic levels and maximum capacitive load for the SDA and SCL lines are the same as the Standard-mode I²C-bus specification. Fast-mode devices are downward-compatible and can communicate with Standard-mode devices in a 0 to 100 kbit/s I²C-bus system. As Standard-mode devices, however, are not upward compatible; they should not be incorporated in a Fast-mode I²C-bus system as they cannot follow the higher transfer rate and unpredictable states would occur.

The Fast-mode I²C-bus specification has the following additional features compared with the Standard-mode:

- The maximum bit rate is increased to 400 kbit/s.
- Timing of the serial data (SDA) and serial clock (SCL) signals has been adapted. There is no need for compatibility with other bus systems such as CBUS because they cannot operate at the increased bit rate.
- The inputs of Fast-mode devices incorporate spike suppression and a Schmitt trigger at the SDA and SCL inputs.
- The output buffers of Fast-mode devices incorporate slope control of the falling edges of the SDA and SCL signals.
- If the power supply to a Fast-mode device is switched off, the SDA and SCL I/O pins must be floating so that they do not obstruct the bus lines.

The external pull-up devices connected to the bus lines must be adapted to accommodate the shorter maximum permissible rise time for the Fast-mode I²C-bus. For bus loads up to 200 pF, the pull-up device for each bus line can be a resistor; for bus loads between 200 pF and 400 pF, the pull-up device can be a current source (3 mA max.) or a switched resistor circuit (see [Section 7.2.4](#)).

5.2 Fast-mode Plus

Fast-mode Plus (Fm+) devices offer an increase in I²C-bus transfer speeds and total bus capacitance. Fm+ devices can transfer information at bit rates of up to 1 Mbit/s, yet they remain fully downward compatible with Fast- or Standard-mode devices for bidirectional communication in a mixed-speed bus system. The same serial bus protocol and data format is maintained as with the Fast- or Standard-mode system. Fm+ devices also offer increased drive current over Fast- or Standard-mode devices allowing them to drive longer and/or more heavily loaded buses so that bus buffers do not need to be used.

The drivers in Fast-mode Plus parts are strong enough to satisfy the Fast-mode Plus timing specification with the same 400 pF load as Standard-mode parts. To be backward compatible with Standard-mode, they are also tolerant of the 1 μ s rise time of Standard-mode parts. In applications where only Fast-mode Plus parts are present, the high drive strength and tolerance for slow rise and fall times allow the use of larger bus capacitance as long as set-up, minimum LOW time and minimum HIGH time for Fast-mode Plus are all satisfied and the fall time and rise time do not exceed the 300 ns t_f and 1 μ s t_r specifications of Standard-mode. Bus speed can be traded against load capacitance to increase the maximum capacitance by about a factor of ten.

5.3 Hs-mode

High-speed mode (Hs-mode) devices offer a quantum leap in I²C-bus transfer speeds. Hs-mode devices can transfer information at bit rates of up to 3.4 Mbit/s, yet they remain fully downward compatible with Fast-mode Plus, Fast- or Standard-mode (F/S) devices for bidirectional communication in a mixed-speed bus system. With the exception that arbitration and clock synchronization is not performed during the Hs-mode transfer, the same serial bus protocol and data format is maintained as with the F/S-mode system.

5.3.1 High speed transfer

To achieve a bit transfer of up to 3.4 Mbit/s, the following improvements have been made to the regular I²C-bus specification:

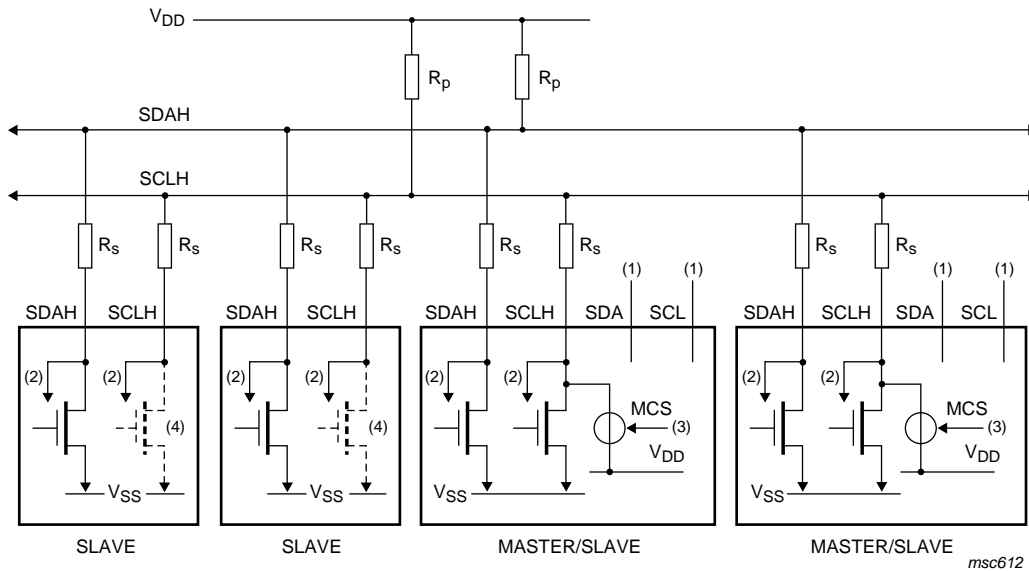
- Hs-mode master devices have an open-drain output buffer for the SDAH signal and a combination of an open-drain pull-down and current-source pull-up circuit on the SCLH output. This current-source circuit shortens the rise time of the SCLH signal. Only the current-source of one master is enabled at any one time, and only during Hs-mode.
- No arbitration or clock synchronization is performed during Hs-mode transfer in multi-master systems, which speeds-up bit handling capabilities. The arbitration procedure always finishes after a preceding master code transmission in F/S-mode.
- Hs-mode master devices generate a serial clock signal with a HIGH to LOW ratio of 1 to 2. This relieves the timing requirements for set-up and hold times.
- As an option, Hs-mode master devices can have a built-in bridge. During Hs-mode transfer, the high-speed data (SDAH) and high-speed serial clock (SCLH) lines of Hs-mode devices are separated by this bridge from the SDA and SCL lines of F/S-mode devices. This reduces the capacitive load of the SDAH and SCLH lines resulting in faster rise and fall times.
- The only difference between Hs-mode slave devices and F/S-mode slave devices is the speed at which they operate. Hs-mode slaves have open-drain output buffers on the SCLH and SDAH outputs. Optional pull-down transistors on the SCLH pin can be used to stretch the LOW level of the SCLH signal, although this is only allowed after the acknowledge bit in Hs-mode transfers.
- The inputs of Hs-mode devices incorporate spike suppression and a Schmitt trigger at the SDAH and SCLH inputs.
- The output buffers of Hs-mode devices incorporate slope control of the falling edges of the SDAH and SCLH signals.

[Figure 32](#) shows the physical I²C-bus configuration in a system with only Hs-mode devices. Pins SDA and SCL on the master devices are only used in mixed-speed bus systems and are not connected in an Hs-mode only system. In such cases, these pins can be used for other functions.

Optional series resistors R_s protect the I/O stages of the I²C-bus devices from high-voltage spikes on the bus lines and minimize ringing and interference.

Pull-up resistors R_p maintain the SDAH and SCLH lines at a HIGH level when the bus is free and ensure that the signals are pulled up from a LOW to a HIGH level within the required rise time. For higher capacitive bus-line loads (>100 pF), the resistor R_p can be replaced by external current source pull-ups to meet the rise time requirements. Unless

preceded by an acknowledge bit, the rise time of the SCLH clock pulses in Hs-mode transfers is shortened by the internal current-source pull-up circuit MCS of the active master.



- (1) SDA and SCL are not used here but may be used for other functions.
- (2) To input filter.
- (3) Only the active master can enable its current-source pull-up circuit.
- (4) Dotted transistors are optional open-drain outputs which can stretch the serial clock signal SCLH.

Fig 32. I²C-bus configuration with Hs-mode devices only

5.3.2 Serial data format in Hs-mode

Serial data transfer format in Hs-mode meets the Standard-mode I²C-bus specification. Hs-mode can only commence after the following conditions (all of which are in F/S-mode):

1. START condition (S)
2. 8-bit master code (0000 1XXX)
3. Not-acknowledge bit (\bar{A})

[Figure 33](#) and [Figure 34](#) show this in more detail. This master code has two main functions:

- It allows arbitration and synchronization between competing masters at F/S-mode speeds, resulting in one winning master.
- It indicates the beginning of an Hs-mode transfer.

Hs-mode master codes are reserved 8-bit codes, which are not used for slave addressing or other purposes. Furthermore, as each master has its own unique master code, up to eight Hs-mode masters can be present on the one I²C-bus system (although master code 0000 1000 should be reserved for test and diagnostic purposes). The master code for an Hs-mode master device is software programmable and is chosen by the System Designer.

Arbitration and clock synchronization only take place during the transmission of the master code and not-acknowledge bit (\bar{A}), after which one winning master remains active. The master code indicates to other devices that an Hs-mode transfer is to begin and the connected devices must meet the Hs-mode specification. As no device is allowed to acknowledge the master code, the master code is followed by a not-acknowledge (\bar{A}).

After the not-acknowledge bit (\bar{A}), and the SCLH line has been pulled-up to a HIGH level, the active master switches to Hs-mode and enables (at time t_H , see [Figure 34](#)) the current-source pull-up circuit for the SCLH signal. As other devices can delay the serial transfer before t_H by stretching the LOW period of the SCLH signal, the active master enables its current-source pull-up circuit when all devices have released the SCLH line and the SCLH signal has reached a HIGH level, thus speeding up the last part of the rise time of the SCLH signal.

The active master then sends a repeated START condition (S_r) followed by a 7-bit slave address (or 10-bit slave address, see [Section 3.1.11](#)) with a R/W bit address, and receives an acknowledge bit (A) from the selected slave.

After a repeated START condition and after each acknowledge bit (A) or not-acknowledge bit (\bar{A}), the active master disables its current-source pull-up circuit. This enables other devices to delay the serial transfer by stretching the LOW period of the SCLH signal. The active master re-enables its current-source pull-up circuit again when all devices have released and the SCLH signal reaches a HIGH level, and so speeds up the last part of the SCLH signal's rise time.

Data transfer continues in Hs-mode after the next repeated START (S_r), and only switches back to F/S-mode after a STOP condition (P). To reduce the overhead of the master code, it is possible that a master links a number of Hs-mode transfers, separated by repeated START conditions (S_r).

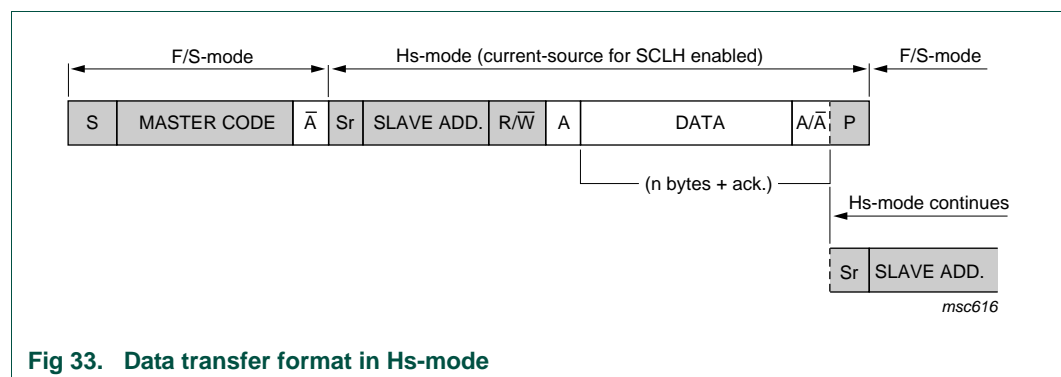


Fig 33. Data transfer format in Hs-mode

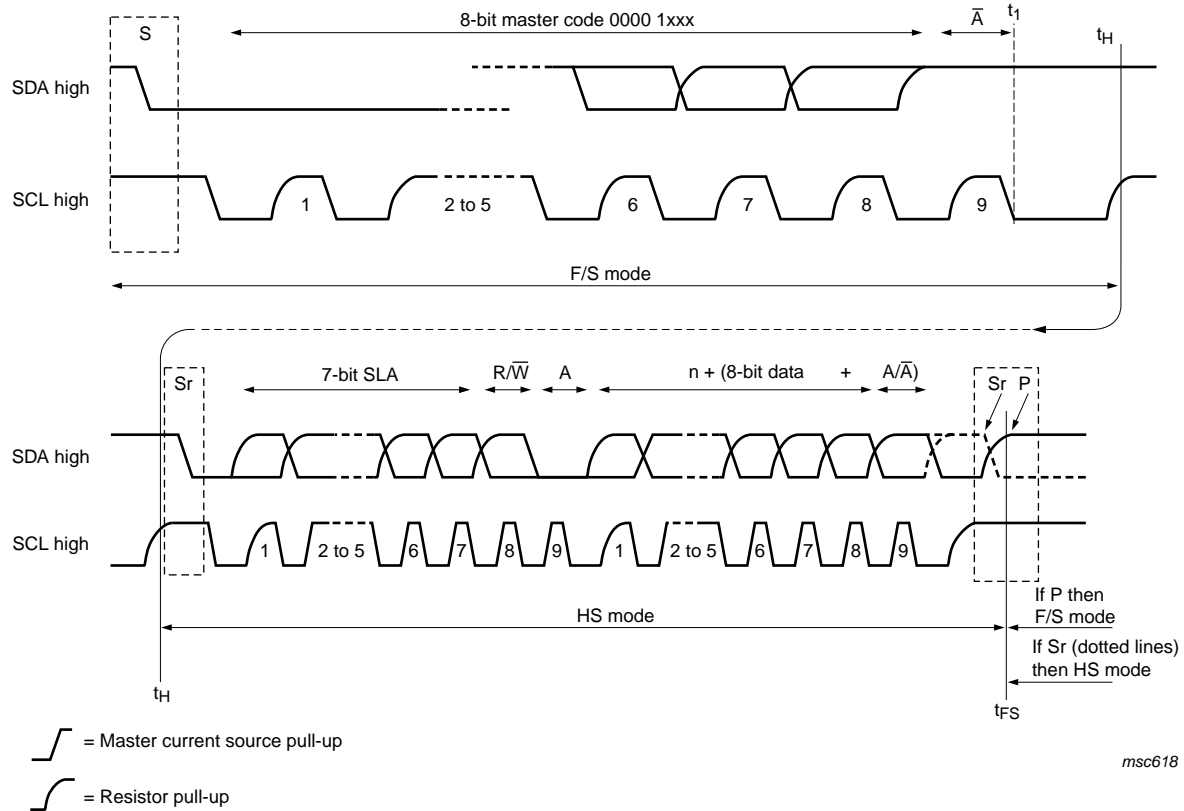


Fig 34. A complete Hs-mode transfer

5.3.3 Switching from F/S-mode to Hs-mode and back

After reset and initialization, Hs-mode devices must be in Fast-mode (which is in effect F/S-mode, as Fast-mode is downward compatible with Standard-mode). Each Hs-mode device can switch from Fast-mode to Hs-mode and back and is controlled by the serial transfer on the I²C-bus.

Before time t_1 in [Figure 34](#), each connected device operates in Fast-mode. Between times t_1 and t_H (this time interval can be stretched by any device) each connected device must recognize the 'S 00001XXX A' sequence and has to switch its internal circuit from the Fast-mode setting to the Hs-mode setting. Between times t_1 and t_H , the connected master and slave devices perform this switching by the following actions.

The active (winning) master:

1. Adapts its SDAH and SCLH input filters according to the spike suppression requirement in Hs-mode.
2. Adapts the set-up and hold times according to the Hs-mode requirements.
3. Adapts the slope control of its SDAH and SCLH output stages according to the Hs-mode requirement.
4. Switches to the Hs-mode bit-rate, which is required after time t_H .
5. Enables the current source pull-up circuit of its SCLH output stage at time t_H .

The non-active, or losing masters:

1. Adapt their SDAH and SCLH input filters according to the spike suppression requirement in Hs-mode.
2. Wait for a STOP condition to detect when the bus is free again.

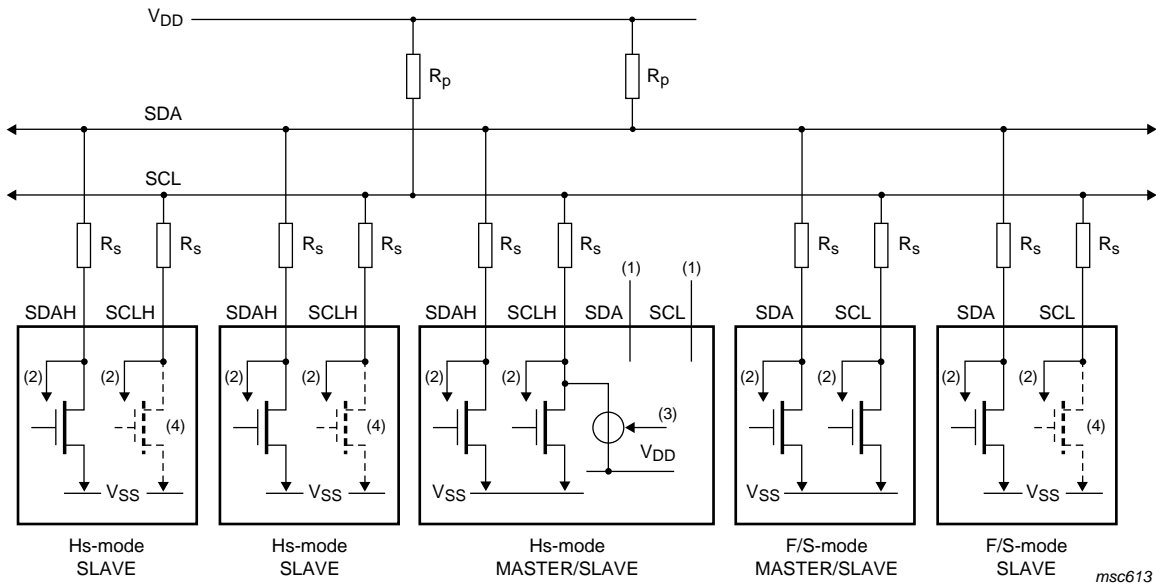
All slaves:

1. Adapt their SDAH and SCLH input filters according to the spike suppression requirement in Hs-mode.
2. Adapt the set-up and hold times according to the Hs-mode requirements. This requirement may already be fulfilled by the adaptation of the input filters.
3. Adapt the slope control of their SDAH output stages, if necessary. For slave devices, slope control is applicable for the SDAH output stage only and, depending on circuit tolerances, both the Fast-mode and Hs-mode requirements may be fulfilled without switching its internal circuit.

At time t_{FS} in [Figure 34](#), each connected device must recognize the STOP condition (P) and switch its internal circuit from the Hs-mode setting back to the Fast-mode setting as present before time t_1 . This must be completed within the minimum bus free time as specified in [Table 10](#) according to the Fast-mode specification.

5.3.4 Hs-mode devices at lower speed modes

Hs-mode devices are fully downwards compatible, and can be connected to an F/S-mode I²C-bus system (see [Figure 35](#)). As no master code is transmitted in such a configuration, all Hs-mode master devices stay in F/S-mode and communicate at F/S-mode speeds with their current-source disabled. The SDAH and SCLH pins are used to connect to the F/S-mode bus system, allowing the SDA and SCL pins (if present) on the Hs-mode master device to be used for other functions.



- (1) Bridge not used. SDA and SCL may have an alternative function.
- (2) To input filter.
- (3) The current-source pull-up circuit stays disabled.
- (4) Dotted transistors are optional open-drain outputs which can stretch the serial clock signal SCL.

Fig 35. Hs-mode devices at F/S-mode speed

5.3.5 Mixed speed modes on one serial bus system

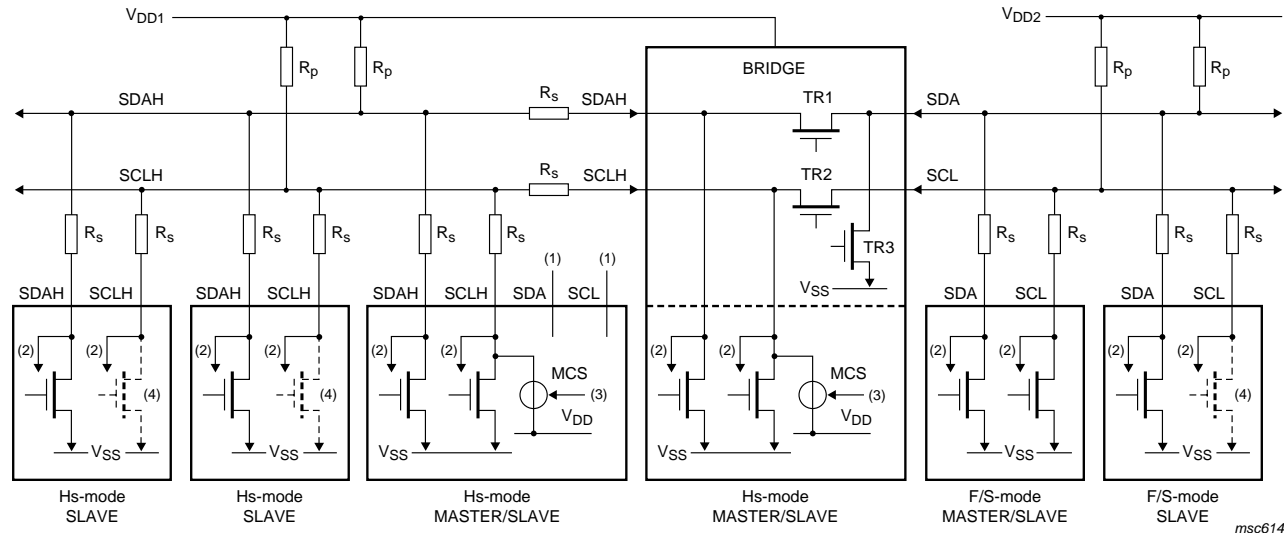
If a system has a combination of Hs-mode, Fast-mode and/or Standard-mode devices, it is possible, by using an interconnection bridge, to have different bit rates between different devices (see [Figure 36](#) and [Figure 37](#)).

One bridge is required to connect/disconnect an Hs-mode section to/from an F/S-mode section at the appropriate time. This bridge includes a level shift function that allows devices with different supply voltages to be connected. For example F/S-mode devices with a V_{DD2} of 5 V can be connected to Hs-mode devices with a V_{DD1} of 3 V or less (that is, where $V_{DD2} \geq V_{DD1}$), provided SDA and SCL pins are 5 V tolerant. This bridge is incorporated in Hs-mode master devices and is completely controlled by the serial signals SDAH, SCLH, SDA and SCL. Such a bridge can be implemented in any IC as an autonomous circuit.

TR1, TR2 and TR3 are N-channel transistors. TR1 and TR2 have a transfer gate function, and TR3 is an open-drain pull-down stage. If TR1 or TR2 are switched on they transfer a LOW level in both directions, otherwise when both the drain and source rise to a HIGH level there is a high-impedance between the drain and source of each switched-on transistor. In the latter case, the transistors act as a level shifter as SDAH and SCLH are pulled-up to V_{DD1} and SDA and SCL are pulled-up to V_{DD2} .

During F/S-mode speed, a bridge on one of the Hs-mode masters connects the SDAH and SCLH lines to the corresponding SDA and SCL lines thus permitting Hs-mode devices to communicate with F/S-mode devices at slower speeds. Arbitration and synchronization are possible during the total F/S-mode transfer between all connected devices as described in [Section 3.1.7](#). During Hs-mode transfer, however, the bridge

opens to separate the two bus sections and allows Hs-mode devices to communicate with each other at 3.4 Mbit/s. Arbitration between Hs-mode devices and F/S-mode devices is only performed during the master code (0000 1XXX), and normally won by one Hs-mode master as no slave address has four leading zeros. Other masters can win the arbitration only if they send a reserved 8-bit code (0000 0XXX). In such cases, the bridge remains closed and the transfer proceeds in F/S-mode. [Table 8](#) gives the possible communication speeds in such a system.



- (1) Bridge not used. SDA and SCL may have an alternative function.
- (2) To input filter.
- (3) Only the active master can enable its current-source pull-up circuit.
- (4) Dotted transistors are optional open-drain outputs which can stretch the serial clock signal SCL or SCLH.

Fig 36. Bus system with transfer at Hs-mode and F/S-mode speeds

Table 8. Communication bit rates in a mixed-speed bus system

Transfer between	Serial bus system configuration			
	Hs + Fast + Standard	Hs + Fast	Hs + Standard	Fast + Standard
Hs ↔ Hs	0 to 3.4 Mbit/s	0 to 3.4 Mbit/s	0 to 3.4 Mbit/s	-
Hs ↔ Fast	0 to 100 kbit/s	0 to 400 kbit/s	-	-
Hs ↔ Standard	0 to 100 kbit/s	-	0 to 100 kbit/s	-
Fast ↔ Standard	0 to 100 kbit/s	-	-	0 to 100 kbit/s
Fast ↔ Fast	0 to 100 kbit/s	0 to 400 kbit/s	-	0 to 100 kbit/s
Standard ↔ Standard	0 to 100 kbit/s	-	0 to 100 kbit/s	0 to 100 kbit/s

Remark: [Table 8](#) assumes that the Hs devices are isolated from the Fm and Sm devices when operating at 3.4 Mbit/s. The bus speed is always constrained to the maximum communication rate of the slowest device attached to the bus.

5.3.6 Standard, Fast-mode and Fast-mode Plus transfer in a mixed-speed bus system

The bridge shown in [Figure 36](#) interconnects corresponding serial bus lines, forming one serial bus system. As no master code (0000 1XXX) is transmitted, the current-source pull-up circuits stay disabled and all output stages are open-drain. All devices, including Hs-mode devices, communicate with each other according to the protocol, format and speed of the F/S-mode I²C-bus specification.

5.3.7 Hs-mode transfer in a mixed-speed bus system

[Figure 37](#) shows the timing diagram of a complete Hs-mode transfer, which is invoked by a START condition, a master code, and a not-acknowledge \bar{A} (at F/S-mode speed). Although this timing diagram is split in two parts, it should be viewed as one timing diagram where time point t_H is a common point for both parts.

The master code is recognized by the bridge in the active or non-active master (see [Figure 36](#)). The bridge performs the following actions:

1. Between t_1 and t_H (see [Figure 37](#)), transistor TR1 opens to separate the SDAH and SDA lines, after which transistor TR3 closes to pull-down the SDA line to V_{SS} .
2. When both SCLH and SCL become HIGH (t_H in [Figure 37](#)), transistor TR2 opens to separate the SCLH and SCL lines. TR2 must be opened before SCLH goes LOW after Sr.

Hs-mode transfer starts after t_H with a repeated START condition (Sr). During Hs-mode transfer, the SCL line stays at a HIGH and the SDA line at a LOW steady-state level, and so is prepared for the transfer of a STOP condition (P).

After each acknowledge (A) or not-acknowledge bit (\bar{A}), the active master disables its current-source pull-up circuit. This enables other devices to delay the serial transfer by stretching the LOW period of the SCLH signal. The active master re-enables its current-source pull-up circuit again when all devices are released and the SCLH signal reaches a HIGH level, and so speeds up the last part of the SCLH signal rise time. In irregular situations, F/S-mode devices can close the bridge (TR1 and TR2 closed, TR3 open) at any time by pulling down the SCL line for at least 1 μ s, for example, to recover from a bus hang-up.

Hs-mode finishes with a STOP condition and brings the bus system back into the F/S-mode. The active master disables its current-source MCS when the STOP condition (P) at SDAH is detected (t_{FS} in [Figure 37](#)). The bridge also recognizes this STOP condition and takes the following actions:

1. Transistor TR2 closes after t_{FS} to connect SCLH with SCL; both of which are HIGH at this time. Transistor TR3 opens after t_{FS} , which releases the SDA line and allows it to be pulled HIGH by the pull-up resistor R_p . This is the STOP condition for the F/S-mode devices. TR3 must open fast enough to ensure the bus free time between the STOP condition and the earliest next START condition is according to the Fast-mode specification (see t_{BUF} in [Table 10](#)).
2. When SDA reaches a HIGH (t_2 in [Figure 37](#)), transistor TR1 closes to connect SDAH with SDA. (Note: interconnections are made when all lines are HIGH, thus preventing spikes on the bus lines.) TR1 and TR2 must be closed within the minimum bus free time according to the Fast-mode specification (see t_{BUF} in [Table 10](#)).

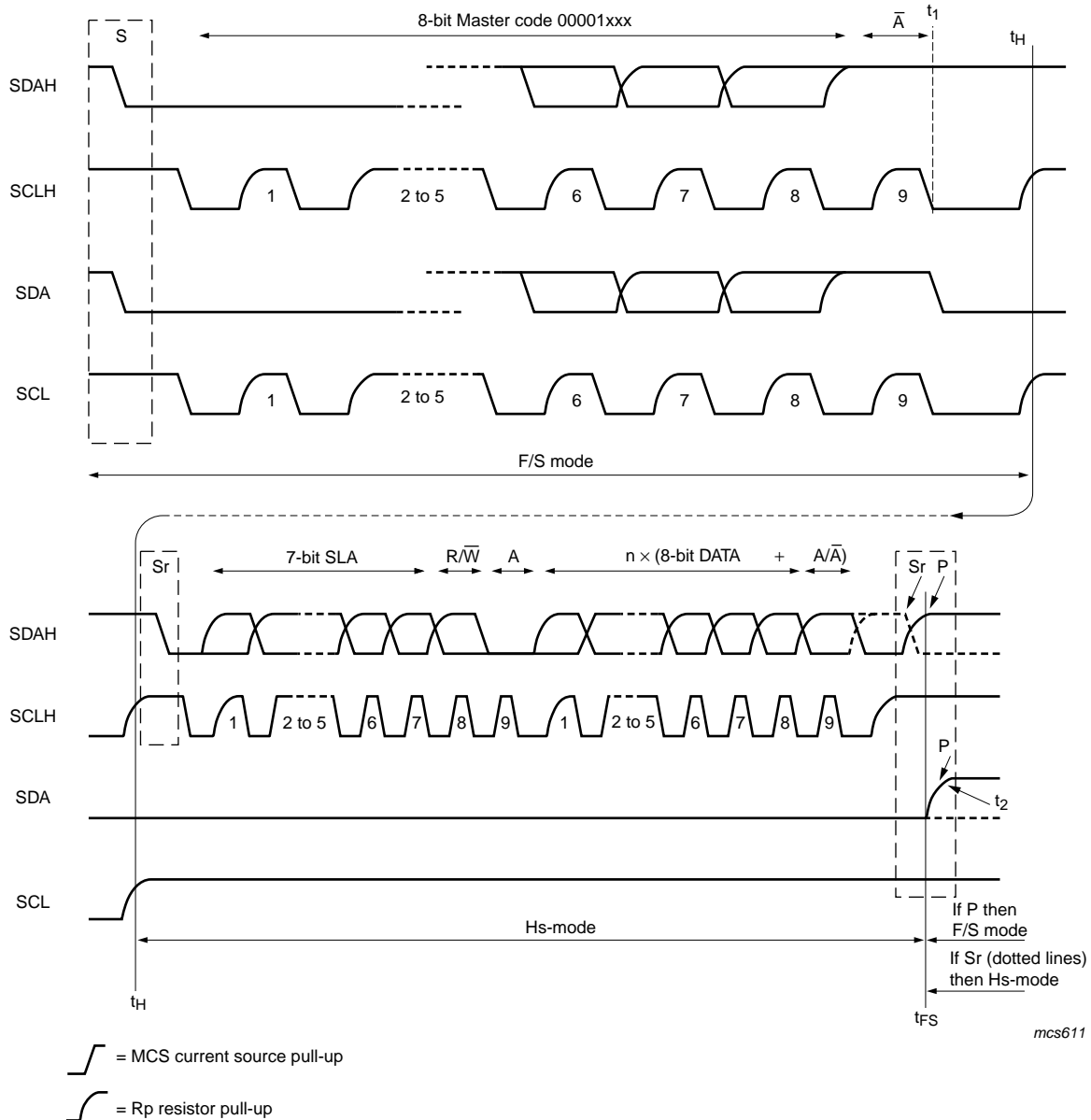


Fig 37. A complete Hs-mode transfer in a mixed-speed bus system

5.3.8 Timing requirements for the bridge in a mixed-speed bus system

It can be seen from [Figure 37](#) that the actions of the bridge at t_1 , t_H and t_{FS} must be so fast that it does not affect the SDAH and SCLH lines. Furthermore the bridge must meet the related timing requirements of the Fast-mode specification for the SDA and SCL lines.

5.4 Ultra Fast-mode

Ultra Fast-mode (UFm) devices offer an increase in I²C-bus transfer speeds. UFm devices can transfer information at bit rates of up to 5 Mbit/s. UFm devices offer push-pull drivers, eliminating the pull-up resistors, allowing higher transfer rates. The same serial bus protocol and data format is maintained as with the Sm, Fm, or Fm+ system. UFm bus devices are not compatible with bidirectional I²C-bus devices.

6. Electrical specifications and timing for I/O stages and bus lines

6.1 Standard-, Fast-, and Fast-mode Plus devices

The I/O levels, I/O current, spike suppression, output slope control and pin capacitance are given in [Table 9](#). The I²C-bus timing characteristics, bus-line capacitance and noise margin are given in [Table 10](#). [Figure 38](#) shows the timing definitions for the I²C-bus.

The minimum HIGH and LOW periods of the SCL clock specified in [Table 10](#) determine the maximum bit transfer rates of 100 kbit/s for Standard-mode devices, 400 kbit/s for Fast-mode devices, and 1000 kbit/s for Fast-mode Plus. Devices must be able to follow transfers at their own maximum bit rates, either by being able to transmit or receive at that speed or by applying the clock synchronization procedure described in [Section 3.1.7](#) which forces the master into a wait state and stretch the LOW period of the SCL signal. In the latter case, the bit transfer rate is reduced.

Table 9. Characteristics of the SDA and SCL I/O stages*n/a = not applicable.*

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
V _{IL}	LOW-level input voltage ^[1]		−0.5	0.3V _{DD}	−0.5	0.3V _{DD}	−0.5	0.3V _{DD}	V
V _{IH}	HIGH-level input voltage ^[1]		0.7V _{DD}	^[2]	0.7V _{DD}	^[2]	0.7V _{DD} ^[1]	^[2]	V
V _{hys}	hysteresis of Schmitt trigger inputs		-	-	0.05V _{DD}	-	0.05V _{DD}	-	V
V _{OL1}	LOW-level output voltage 1	(open-drain or open-collector) at 3 mA sink current; V _{DD} > 2 V	0	0.4	0	0.4	0	0.4	V
V _{OL2}	LOW-level output voltage 2	(open-drain or open-collector) at 2 mA sink current ^[3] ; V _{DD} ≤ 2 V	-	-	0	0.2V _{DD}	0	0.2V _{DD}	V
I _{OL}	LOW-level output current	V _{OL} = 0.4 V	3	-	3	-	20	-	mA
		V _{OL} = 0.6 V ^[4]	-	-	6	-	-	-	mA
t _{of}	output fall time from V _{IHmin} to V _{ILmax}		-	250 ^[5]	20 × (V _{DD} / 5.5 V) ^[6]	250 ^[5]	20 × (V _{DD} / 5.5 V) ^[6]	120 ^[7]	ns
t _{SP}	pulse width of spikes that must be suppressed by the input filter		-	-	0	50 ^[8]	0	50 ^[8]	ns
I _i	input current each I/O pin	0.1V _{DD} < V _I < 0.9V _{DDmax}	−10	+10	−10 ^[9]	+10 ^[9]	−10 ^[9]	+10 ^[9]	μA
C _i	capacitance for each I/O pin ^[10]		-	10	-	10	-	10	pF

[1] Some legacy Standard-mode devices had fixed input levels of V_{IL} = 1.5 V and V_{IH} = 3.0 V. Refer to component data sheets.

[2] Maximum V_{IH} = V_{DD(max)} + 0.5 V or 5.5 V, which ever is lower. See component data sheets.

[3] The same resistor value to drive 3 mA at 3.0 V V_{DD} provides the same RC time constant when using <2 V V_{DD} with a smaller current draw.

[4] In order to drive full bus load at 400 kHz, 6 mA I_{OL} is required at 0.6 V V_{OL}. Parts not meeting this specification can still function, but not at 400 kHz and 400 pF.

[5] The maximum t_f for the SDA and SCL bus lines quoted in [Table 10](#) (300 ns) is longer than the specified maximum t_{of} for the output stages (250 ns). This allows series protection resistors (R_s) to be connected between the SDA/SCL pins and the SDA/SCL bus lines as shown in [Figure 45](#) without exceeding the maximum specified t_f.

[6] Necessary to be backwards compatible with Fast-mode.

[7] In Fast-mode Plus, fall time is specified the same for both output stage and bus timing. If series resistors are used, designers should allow for this when considering bus timing.

[8] Input filters on the SDA and SCL inputs suppress noise spikes of less than 50 ns.

[9] If V_{DD} is switched off, I/O pins of Fast-mode and Fast-mode Plus devices must not obstruct the SDA and SCL lines.

[10] Special purpose devices such as multiplexers and switches may exceed this capacitance because they connect multiple paths together.

Table 10. Characteristics of the SDA and SCL bus lines for Standard, Fast, and Fast-mode Plus I²C-bus devices^[1]

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
f _{SCL}	SCL clock frequency		0	100	0	400	0	1000	kHz
t _{HD;STA}	hold time (repeated) START condition	After this period, the first clock pulse is generated.	4.0	-	0.6	-	0.26	-	μs
t _{LOW}	LOW period of the SCL clock		4.7	-	1.3	-	0.5	-	μs
t _{HIGH}	HIGH period of the SCL clock		4.0	-	0.6	-	0.26	-	μs
t _{SU;STA}	set-up time for a repeated START condition		4.7	-	0.6	-	0.26	-	μs
t _{HD;DAT}	data hold time ^[2]	CBUS compatible masters (see Remark in Section 4.1)	5.0	-	-	-	-	-	μs
		I ² C-bus devices	0 ^[3]	- ^[4]	0 ^[3]	- ^[4]	0	-	μs
t _{SU;DAT}	data set-up time		250	-	100 ^[5]	-	50	-	ns
t _r	rise time of both SDA and SCL signals		-	1000	20	300	-	120	ns
t _f	fall time of both SDA and SCL signals ^{[3][6][7][8]}		-	300	20 × (V _{DD} / 5.5 V)	300	20 × (V _{DD} / 5.5 V) ^[9]	120 ^[8]	ns
t _{SU;STO}	set-up time for STOP condition		4.0	-	0.6	-	0.26	-	μs
t _{BUF}	bus free time between a STOP and START condition		4.7	-	1.3	-	0.5	-	μs
C _b	capacitive load for each bus line ^[10]		-	400	-	400	-	550	pF
t _{VD;DAT}	data valid time ^[11]		-	3.45 ^[4]	-	0.9 ^[4]	-	0.45 ^[4]	μs
t _{VD;ACK}	data valid acknowledge time ^[12]		-	3.45 ^[4]	-	0.9 ^[4]	-	0.45 ^[4]	μs
V _{nL}	noise margin at the LOW level	for each connected device (including hysteresis)	0.1V _{DD}	-	0.1V _{DD}	-	0.1V _{DD}	-	V
V _{nH}	noise margin at the HIGH level	for each connected device (including hysteresis)	0.2V _{DD}	-	0.2V _{DD}	-	0.2V _{DD}	-	V

[1] All values referred to V_{IH(min)} (0.3V_{DD}) and V_{IL(max)} (0.7V_{DD}) levels (see [Table 9](#)).

[2] t_{HD;DAT} is the data hold time that is measured from the falling edge of SCL, applies to data in transmission and the acknowledge.

[3] A device must internally provide a hold time of at least 300 ns for the SDA signal (with respect to the V_{IH(min)} of the SCL signal) to bridge the undefined region of the falling edge of SCL.

[4] The maximum t_{HD;DAT} could be 3.45 μs and 0.9 μs for Standard-mode and Fast-mode, but must be less than the maximum of t_{VD;DAT} or t_{VD;ACK} by a transition time. This maximum must only be met if the device does not stretch the LOW period (t_{LOW}) of the SCL signal. If the clock stretches the SCL, the data must be valid by the set-up time before it releases the clock.

- [5] A Fast-mode I²C-bus device can be used in a Standard-mode I²C-bus system, but the requirement $t_{\text{SU;DAT}}$ 250 ns must then be met. This will automatically be the case if the device does not stretch the LOW period of the SCL signal. If such a device does stretch the LOW period of the SCL signal, it must output the next data bit to the SDA line $t_{\text{r(max)}} + t_{\text{SU;DAT}} = 1000 + 250 = 1250$ ns (according to the Standard-mode I²C-bus specification) before the SCL line is released. Also the acknowledge timing must meet this set-up time.
- [6] If mixed with Hs-mode devices, faster fall times according to [Table 10](#) are allowed.
- [7] The maximum t_f for the SDA and SCL bus lines is specified at 300 ns. The maximum fall time for the SDA output stage t_f is specified at 250 ns. This allows series protection resistors to be connected in between the SDA and the SCL pins and the SDA/SCL bus lines without exceeding the maximum specified t_f .
- [8] In Fast-mode Plus, fall time is specified the same for both output stage and bus timing. If series resistors are used, designers should allow for this when considering bus timing.
- [9] Necessary to be backwards compatible to Fast-mode.
- [10] The maximum bus capacitance allowable may vary from this value depending on the actual operating voltage and frequency of the application. [Section 7.2](#) discusses techniques for coping with higher bus capacitances.
- [11] $t_{\text{VD;DAT}}$ = time for data signal from SCL LOW to SDA output (HIGH or LOW, depending on which one is worse).
- [12] $t_{\text{VD;ACK}}$ = time for Acknowledgement signal from SCL LOW to SDA output (HIGH or LOW, depending on which one is worse).

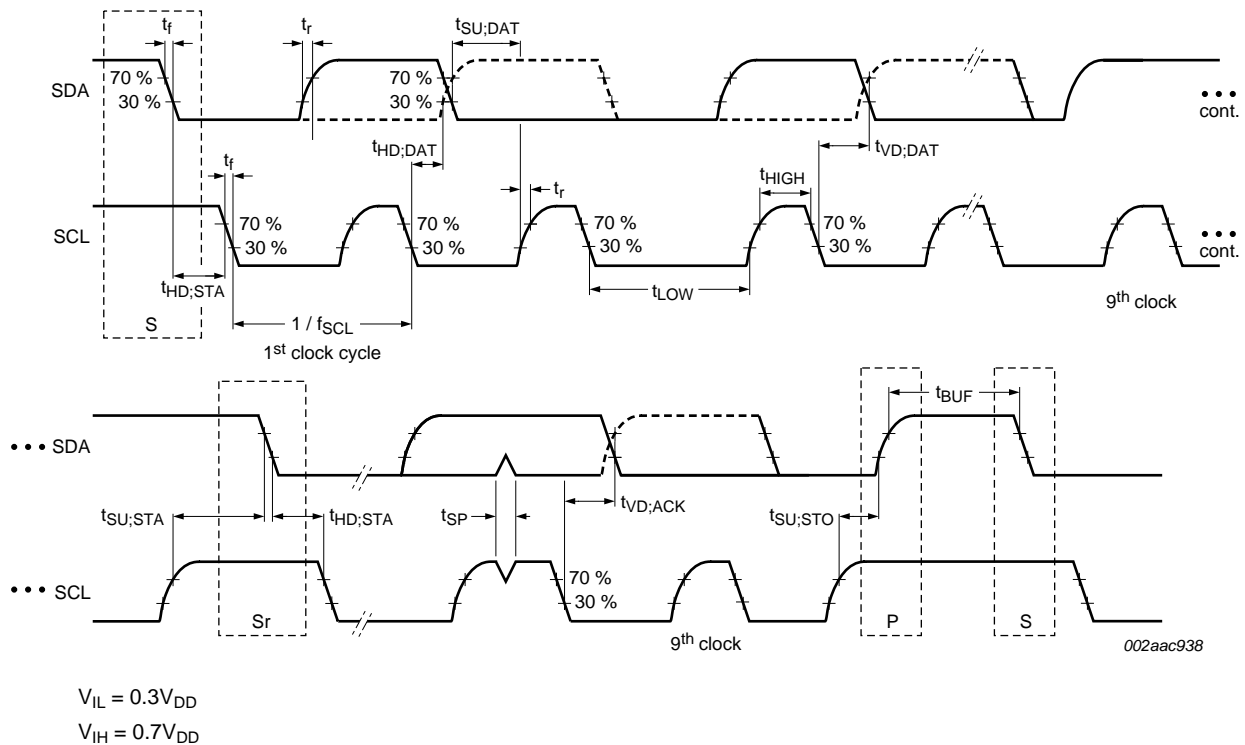


Fig 38. Definition of timing for F/S-mode devices on the I²C-bus

6.2 Hs-mode devices

The I/O levels, I/O current, spike suppression, output slope control and pin capacitance for I²C-bus Hs-mode devices are given in [Table 11](#). The noise margin for HIGH and LOW levels on the bus lines are the same as specified for F/S-mode I²C-bus devices.

[Figure 39](#) shows all timing parameters for the Hs-mode timing. The 'normal' START condition S does not exist in Hs-mode. Timing parameters for Address bits, R/W bit, Acknowledge bit and DATA bits are all the same. Only the rising edge of the first SCLH clock signal after an acknowledge bit has a larger value because the external R_p has to pull up SCLH without the help of the internal current-source.

The Hs-mode timing parameters for the bus lines are specified in [Table 12](#). The minimum HIGH and LOW periods and the maximum rise and fall times of the SCLH clock signal determine the highest bit rate.

With an internally generated SCLH signal with LOW and HIGH level periods of 200 ns and 100 ns respectively, an Hs-mode master fulfills the timing requirements for the external SCLH clock pulses (taking the rise and fall times into account) for the maximum bit rate of 3.4 Mbit/s. So a basic frequency of 10 MHz, or a multiple of 10 MHz, can be used by an Hs-mode master to generate the SCLH signal. There are no limits for maximum HIGH and LOW periods of the SCLH clock, and there is no limit for a lowest bit rate.

Timing parameters are independent for capacitive load up to 100 pF for each bus line allowing the maximum possible bit rate of 3.4 Mbit/s. At a higher capacitive load on the bus lines, the bit rate decreases gradually. The timing parameters for a capacitive bus load of 400 pF are specified in [Table 12](#), allowing a maximum bit rate of 1.7 Mbit/s. For

capacitive bus loads between 100 pF and 400 pF, the timing parameters must be interpolated linearly. Rise and fall times are in accordance with the maximum propagation time of the transmission lines SDAH and SCLH to prevent reflections of the open ends.

Table 11. Characteristics of the SDAH, SCLH, SDA and SCL I/O stages for Hs-mode I²C-bus devices

Symbol	Parameter	Conditions	Hs-mode		Unit
			Min	Max	
V _{IL}	LOW-level input voltage		-0.5	0.3V _{DD} ^[1]	V
V _{IH}	HIGH-level input voltage		0.7V _{DD} ^[1]	V _{DD} + 0.5 ^[2]	V
V _{hys}	hysteresis of Schmitt trigger inputs		0.1V _{DD} ^[1]	-	V
V _{OL}	LOW-level output voltage	(open-drain) at 3 mA sink current at SDAH, SDA and SCLH			
		V _{DD} > 2 V	0	0.4	V
		V _{DD} ≤ 2 V	0	0.2V _{DD}	V
R _{onL}	transfer gate on resistance for currents between SDA and SDAH or SCL and SCLH	V _{OL} level; I _{OL} = 3 mA	-	50	Ω
R _{onH} ^[2]	transfer gate on resistance between SDA and SDAH or SCL and SCLH	both signals (SDA and SDAH, or SCL and SCLH) at V _{DD} level	50	-	kΩ
I _{CS}	pull-up current of the SCLH current-source	SCLH output levels between 0.3V _{DD} and 0.7V _{DD}	3	12	mA
t _{rCL}	rise time of SCLH signal	output rise time (current-source enabled) with an external pull-up current source of 3 mA			
		capacitive load from 10 pF to 100 pF	10	40	ns
		capacitive load of 400 pF ^[3]	20	80	ns
t _{fCL}	fall time of SCLH signal	output fall time (current-source enabled) with an external pull-up current source of 3 mA			
		capacitive load from 10 pF to 100 pF	10	40	ns
		capacitive load of 400 pF ^[3]	20	80	ns
t _{fDA}	fall time of SDAH signal	capacitive load from 10 pF to 100 pF	10	80	ns
		capacitive load of 400 pF ^[3]	20	160	ns
t _{SP}	pulse width of spikes that must be suppressed by the input filter	SDAH and SCLH	0	10	ns
I _i ^[4]	input current each I/O pin	input voltage between 0.1V _{DD} and 0.9V _{DD}	-	10	μA
C _i	capacitance for each I/O pin ^[5]		-	10	pF

[1] Devices that use non-standard supply voltages which do not conform to the intended I²C-bus system levels must relate their input levels to the V_{DD} voltage to which the pull-up resistors R_p are connected.

[2] Devices that offer the level shift function must tolerate a maximum input voltage of 5.5 V at SDA and SCL.

[3] For capacitive bus loads between 100 pF and 400 pF, the rise and fall time values must be linearly interpolated.

[4] If their supply voltage has been switched off, SDAH and SCLH I/O stages of Hs-mode slave devices must have floating outputs. Due to the current-source output circuit, which normally has a clipping diode to V_{DD}, this requirement is not mandatory for the SCLH or the SDAH I/O stage of Hs-mode master devices. This means that the supply voltage of Hs-mode master devices cannot be switched off without affecting the SDAH and SCLH lines.

[5] Special purpose devices such as multiplexers and switches may exceed this capacitance because they connect multiple paths together.

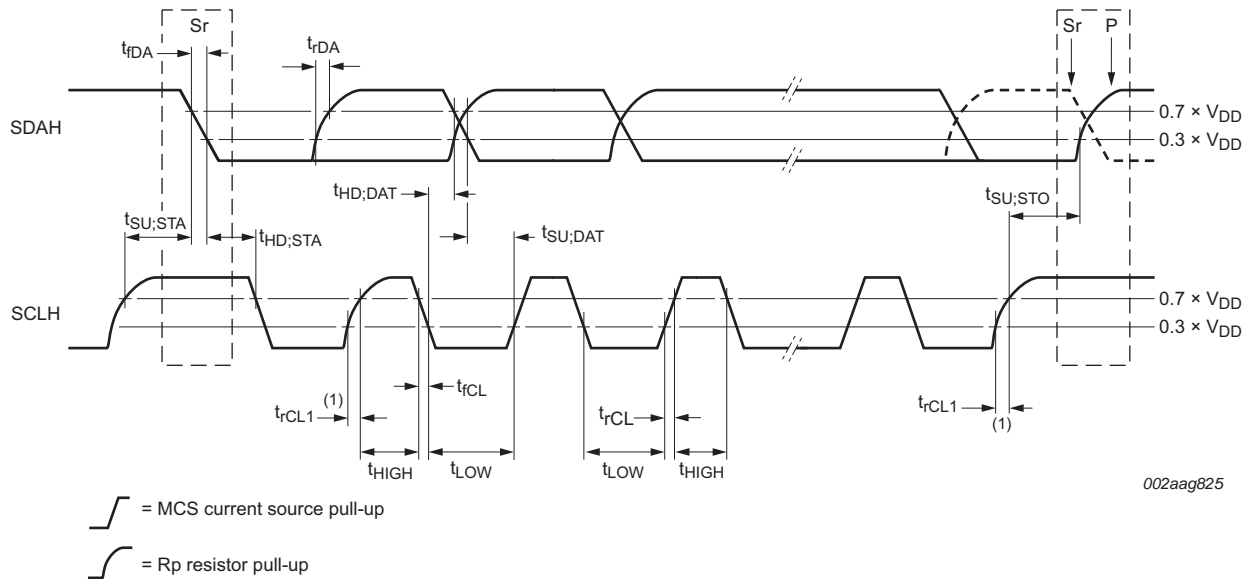
Table 12. Characteristics of the SDAH, SCLH, SDA and SCL bus lines for Hs-mode I²C-bus devices^[1]

Symbol	Parameter	Conditions	C _b = 100 pF (max)		C _b = 400 pF ^[2]		Unit
			Min	Max	Min	Max	
f _{SCLH}	SCLH clock frequency		0	3.4	0	1.7	MHz
t _{SU;STA}	set-up time for a repeated START condition		160	-	160	-	ns
t _{HD;STA}	hold time (repeated) START condition		160	-	160	-	ns
t _{LOW}	LOW period of the SCL clock		160	-	320	-	ns
t _{HIGH}	HIGH period of the SCL clock		60	-	120	-	ns
t _{SU;DAT}	data set-up time		10	-	10	-	ns
t _{HD;DAT}	data hold time		0 ^[3]	70	0 ^[3]	150	ns
t _{rCL}	rise time of SCLH signal		10	40	20	80	ns
t _{rCL1}	rise time of SCLH signal after a repeated START condition and after an acknowledge bit		10	80	20	160	ns
t _{fCL}	fall time of SCLH signal		10	40	20	80	ns
t _{rDA}	rise time of SDAH signal		10	80	20	160	ns
t _{fDA}	fall time of SDAH signal		10	80	20	160	ns
t _{SU;STO}	set-up time for STOP condition		160	-	160	-	ns
C _b ^[2]	capacitive load for each bus line	SDAH and SCLH lines	-	100	-	400	pF
		SDAH + SDA line and SCLH + SCL line	-	400	-	400	pF
V _{nL}	noise margin at the LOW level	for each connected device (including hysteresis)	0.1V _{DD}	-	0.1V _{DD}	-	V
V _{nH}	noise margin at the HIGH level	for each connected device (including hysteresis)	0.2V _{DD}	-	0.2V _{DD}	-	V

[1] All values referred to V_{IH(min)} and V_{IL(max)} levels (see [Table 11](#)).

[2] For bus line loads C_b between 100 pF and 400 pF the timing parameters must be linearly interpolated.

[3] A device must internally provide a data hold time to bridge the undefined part between V_{IH} and V_{IL} of the falling edge of the SCLH signal. An input circuit with a threshold as low as possible for the falling edge of the SCLH signal minimizes this hold time.



(1) First rising edge of the SCLH signal after Sr and after each acknowledge bit.

Fig 39. Definition of timing for Hs-mode devices on the I²C-bus

6.3 Ultra Fast-mode devices

The I/O levels, I/O current, spike suppression, output slope control and pin capacitance are given in [Table 13](#). The UFM I²C-bus timing characteristics are given in [Table 14](#). [Figure 40](#) shows the timing definitions for the I²C-bus. The minimum HIGH and LOW periods of the SCL clock specified in [Table 14](#) determine the maximum bit transfer rates of 5000 kbit/s for Ultra Fast-mode. Devices must be able to follow transfers at their own maximum bit rates, either by being able to transmit or receive at that speed.

Table 13. Characteristics of the USDA and USCL I/O stages

n/a = not applicable.

Symbol	Parameter	Conditions	Ultra Fast-mode		Unit
			Min	Max	
V _{IL}	LOW-level input voltage ^[1]		-0.5	+0.3V _{DD}	V
V _{IH}	HIGH-level input voltage ^[1]		0.7V _{DD} ^[1]	- ^[2]	V
V _{hys}	hysteresis of Schmitt trigger inputs		0.05V _{DD}	-	V
V _{OL}	LOW-level output voltage	at 4 mA sink current; V _{DD} > 2 V	0	0.4	V
V _{OH}	HIGH-level output voltage	at 4 mA source current; V _{DD} > 2 V	V _{DD} - 0.4	-	V
I _L	leakage current	V _{DD} = 3.6 V	-1	+1	μA
		V _{DD} = 5.5 V	-10	+10	μA
C _i	input capacitance	^[3]	-	10	pF
t _{SP}	pulse width of spikes that must be suppressed by the input filter	^[4]	-	10	ns

[1] Refer to component data sheets for actual switching points.

[2] Maximum V_{IH} = V_{DD(max)} + 0.5 V or 5.5 V, whichever is lower. See component data sheets.

[3] Special purpose devices such as multiplexers and switches may exceed this capacitance because they connect multiple paths together.

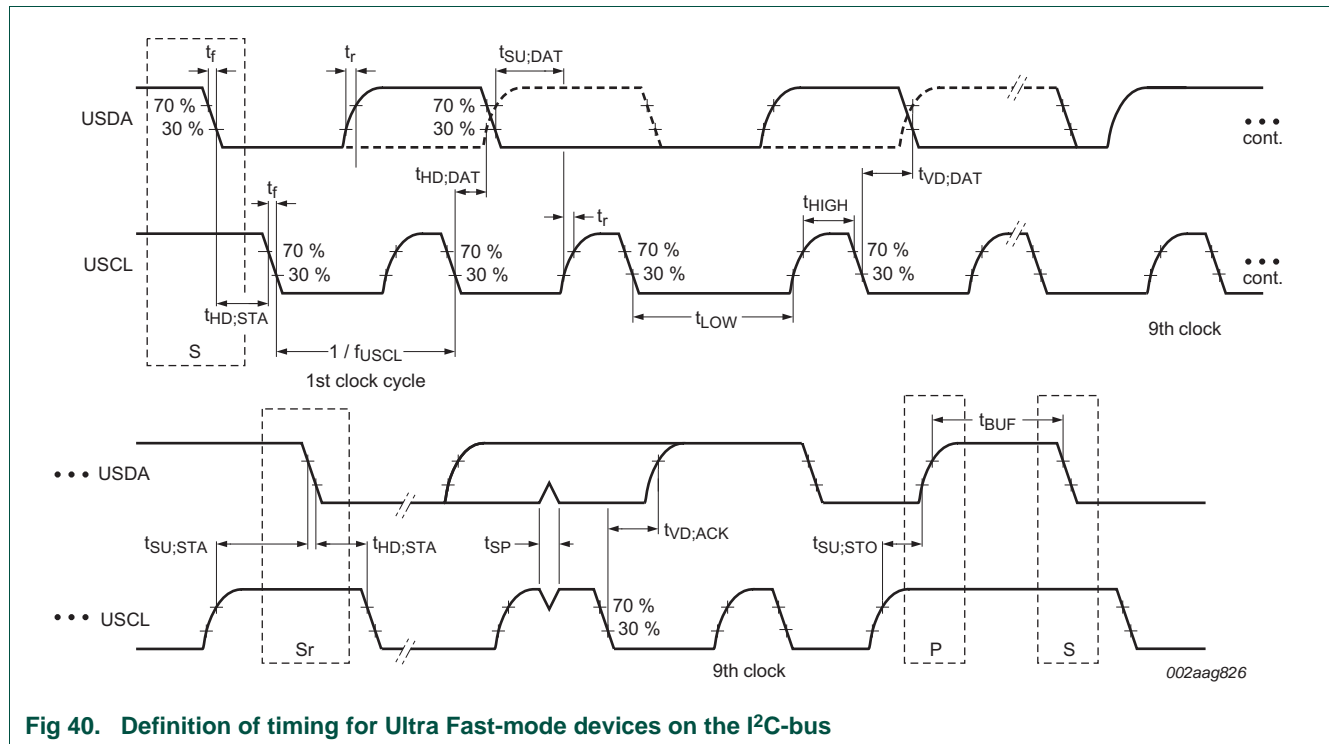
[4] Input filters on the USDA and USCL slave inputs suppress noise spikes of less than 10 ns.

Table 14. UFM I²C-bus frequency and timing specifications

Symbol	Parameter	Conditions	Ultra Fast-mode		Unit
			Min	Max	
f_{USCL}	USCL clock frequency		0	5000	kHz
t_{BUF}	bus free time between a STOP and START condition		80	-	ns
$t_{HD;STA}$	hold time (repeated) START condition		50	-	ns
$t_{SU;STA}$	set-up time for a repeated START condition		50	-	ns
$t_{SU;STO}$	set-up time for STOP condition		50	-	ns
$t_{HD;DAT}$	data hold time		10	-	ns
$t_{VD;DAT}$	data valid time	[1]	10	-	ns
$t_{SU;DAT}$	data set-up time		30	-	ns
t_{LOW}	LOW period of the USCL clock		50	-	ns
t_{HIGH}	HIGH period of the USCL clock		50	-	ns
t_f	fall time of both USDA and USCL signals		-[2]	50	ns
t_r	rise time of both USDA and USCL signals		-[2]	50	ns

[1] $t_{VD;DAT}$ = minimum time for USDA data out to be valid following USCL LOW.

[2] Typical rise time or fall time for UFM signals is 25 ns measured from the 30 % level to the 70 % level (rise time) or from the 70 % level to the 30 % level (fall time).

Fig 40. Definition of timing for Ultra Fast-mode devices on the I²C-bus

7. Electrical connections of I²C-bus devices to the bus lines

7.1 Pull-up resistor sizing

The bus capacitance is the total capacitance of wire, connections and pins. This capacitance limits the maximum value of R_p due to the specified rise time. [Figure 41](#) shows $R_{p(max)}$ as a function of bus capacitance.

Consider the V_{DD} related input threshold of $V_{IH} = 0.7V_{DD}$ and $V_{IL} = 0.3V_{DD}$ for the purposes of RC time constant calculation. Then $V(t) = V_{DD} (1 - e^{-t/RC})$, where t is the time since the charging started and RC is the time constant.

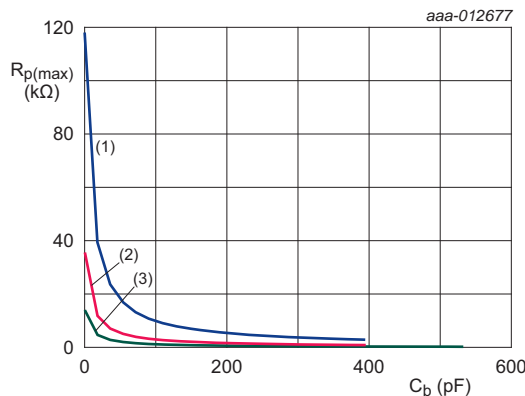
$$V(t_1) = 0.3 \times V_{DD} = V_{DD} (1 - e^{-t_1/RC}); \text{ then } t_1 = 0.3566749 \times RC$$

$$V(t_2) = 0.7 \times V_{DD} = V_{DD} (1 - e^{-t_2/RC}); \text{ then } t_2 = 1.2039729 \times RC$$

$$T = t_2 - t_1 = 0.8473 \times RC$$

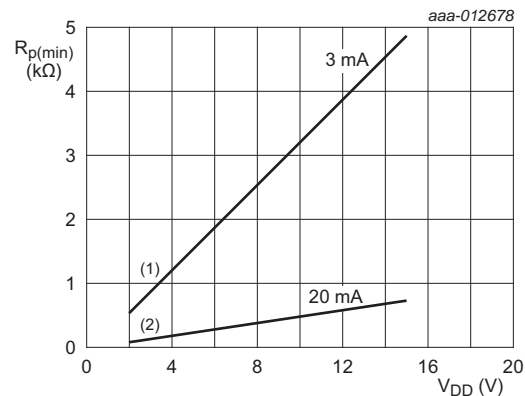
[Figure 41](#) and [Equation 1](#) shows maximum R_p as a function of bus capacitance for Standard-, Fast- and Fast-mode Plus. For each mode, the $R_{p(max)}$ is a function of the rise time maximum (t_r) from [Table 10](#) and the estimated bus capacitance (C_b):

$$R_{p(max)} = \frac{t_r}{0.8473 \times C_b} \quad (1)$$



- (1) Standard-mode
- (2) Fast-mode
- (3) Fast-mode Plus

Fig 41. $R_{p(max)}$ as a function of bus capacitance



- (1) Fast-mode and Standard-mode
- (2) Fast-mode Plus

Fig 42. $R_{p(min)}$ as a function of V_{DD}

The supply voltage limits the minimum value of resistor R_p due to the specified minimum sink current of 3 mA for Standard-mode and Fast-mode, or 20 mA for Fast-mode Plus. $R_{p(min)}$ as a function of V_{DD} is shown in [Figure 42](#). The traces are calculated using [Equation 2](#):

$$R_{p(min)} = \frac{V_{DD} - V_{OL(max)}}{I_{OL}} \quad (2)$$

The designer now has the minimum and maximum value of R_p that is required to meet the timing specification. Portable designs with sensitivity to supply current consumption can use a value toward the higher end of the range in order to limit I_{DD} .

7.2 Operating above the maximum allowable bus capacitance

Bus capacitance limit is specified to limit rise time reductions and allow operating at the rated frequency. While most designs can easily stay within this limit, some applications may exceed it. There are several strategies available to system designers to cope with excess bus capacitance.

- Reduced f_{SCL} ([Section 7.2.1](#)): The bus may be operated at a lower speed (lower f_{SCL}).
- Higher drive outputs ([Section 7.2.2](#)): Devices with higher drive current such as those rated for Fast-mode Plus can be used (PCA96xx).
- Bus buffers ([Section 7.2.3](#)): There are a number of bus buffer devices available that can divide the bus into segments so that each segment has a capacitance below the allowable limit, such as the PCA9517 bus buffer or the PCA9546A switch.
- Switched pull-up circuit ([Section 7.2.4](#)): A switched pull-up circuit can be used to accelerate rising edges by switching a low value pull-up alternately in and out when needed.

7.2.1 Reduced f_{SCL}

To determine a lower allowable bus operating frequency, begin by finding the t_{LOW} and t_{HIGH} of the most limiting device on the bus. Refer to individual component data sheets for these values. Actual rise time (t_r) depends on the RC time constant. The most limiting fall time (t_f) depends on the lowest output drive on the bus. Be sure to allow for any devices that have a minimum t_r or t_f . Refer to [Equation 3](#) for the resulting f_{max} .

$$f_{max} = \frac{1}{t_{LOW(min)} + t_{HIGH(min)} + t_{r(actual)} + t_{f(actual)}} \quad (3)$$

Remark: Very long buses must also account for time of flight of signals.

Actual results are slower, as real parts do not tend to control t_{LOW} and t_{HIGH} to the minimum from 30 % to 70 %, or 70 % to 30 %, respectively.

7.2.2 Higher drive outputs

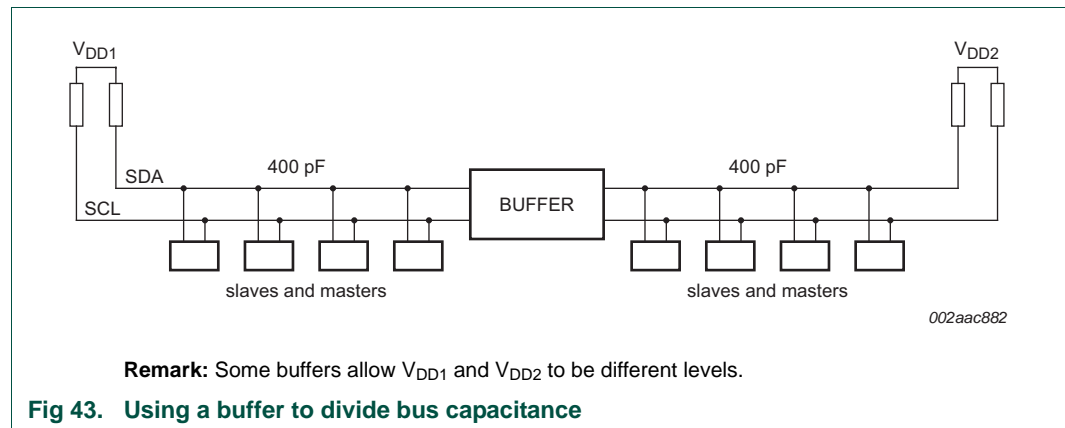
If higher drive devices like the PCA96xx Fast-mode Plus or the P82B bus buffers are used, the higher strength output drivers sink more current which results in considerably faster edge rates, or, looked at another way, allows a higher bus capacitance. Refer to individual component data sheets for actual output drive capability. Repeat the calculation above using the new values of C_b , R_p , t_r and t_f to determine maximum frequency. Bear in mind that the maximum rating for f_{SCL} as specified in [Table 10](#) (100 kHz, 400 kHz and 1000 kHz) may become limiting.

7.2.3 Bus buffers, multiplexers and switches

Another approach to coping with excess bus capacitance is to divide the bus into smaller segments using bus buffers, multiplexers or switches. [Figure 43](#) shows an example of a bus that uses a PCA9515 buffer to deal with high bus capacitance. Each segment is then allowed to have the maximum capacitance so the total bus can have twice the maximum

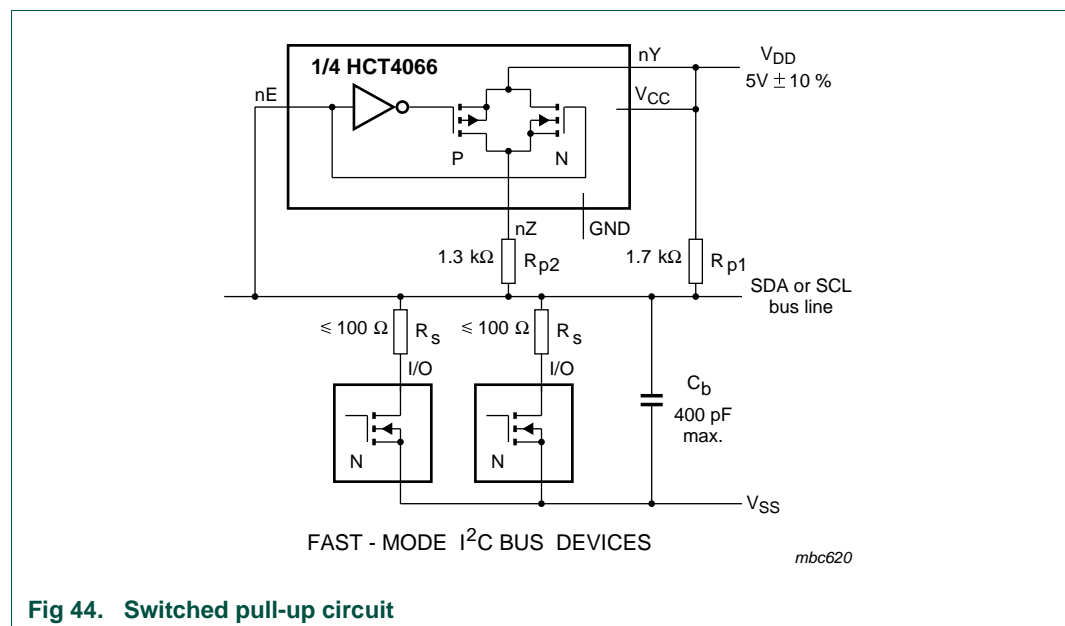
capacitance. Keep in mind that adding a buffer always adds delays — a buffer delay plus an additional transition time to each edge, which reduces the maximum operating frequency and may also introduce special V_{IL} and V_{OL} considerations.

Refer to application notes *AN255, I²C / SMBus Repeaters, Hubs and Expanders* and *AN262, PCA954x Family of I²C / SMBus Multiplexers and Switches* for more details on this subject and the devices available from NXP Semiconductors.



7.2.4 Switched pull-up circuit

The supply voltage (V_{DD}) and the maximum output LOW level determine the minimum value of pull-up resistor R_p (see [Section 7.1](#)). For example, with a supply voltage of $V_{DD} = 5\text{ V} \pm 10\%$ and $V_{OL(max)} = 0.4\text{ V}$ at 3 mA, $R_{p(min)} = (5.5 - 0.4) / 0.003 = 1.7\text{ k}\Omega$. As shown in [Figure 42](#), this value of R_p limits the maximum bus capacitance to about 200 pF to meet the maximum t_r requirement of 300 ns. If the bus has a higher capacitance than this, a switched pull-up circuit (as shown in [Figure 44](#)) can be used.



The switched pull-up circuit in [Figure 44](#) is for a supply voltage of $V_{DD} = 5\text{ V} \pm 10\%$ and a maximum capacitive load of 400 pF. Since it is controlled by the bus levels, it needs no additional switching control signals. During the rising/falling edges, the bilateral switch in the HCT4066 switches pull-up resistor R_{p2} on/off at bus levels between 0.8 V and 2.0 V. Combined resistors R_{p1} and R_{p2} can pull up the bus line within the maximum specified rise time (t_r) of 300 ns.

Series resistors R_s are optional. They protect the I/O stages of the I²C-bus devices from high-voltage spikes on the bus lines, and minimize crosstalk and undershoot of the bus line signals. The maximum value of R_s is determined by the maximum permitted voltage drop across this resistor when the bus line is switched to the LOW level in order to switch off R_{p2} .

Additionally, some bus buffers contain integral rise time accelerators. Stand-alone rise time accelerators are also available.

7.3 Series protection resistors

As shown in [Figure 45](#), series resistors (R_s) of, for example, 300 Ω can be used for protection against high-voltage spikes on the SDA and SCL lines (resulting from the flash-over of a TV picture tube, for example). If series resistors are used, designers must add the additional resistance into their calculations for R_p and allowable bus capacitance.

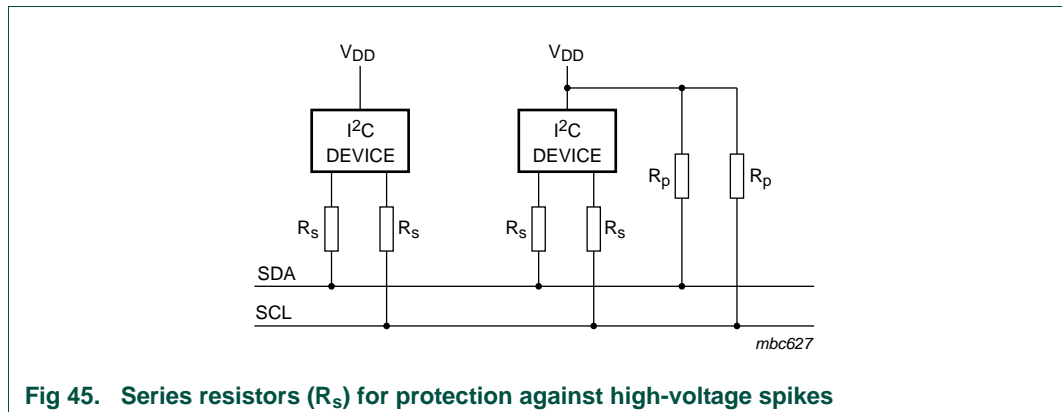


Fig 45. Series resistors (R_s) for protection against high-voltage spikes

The required noise margin of $0.1V_{DD}$ for the LOW level, limits the maximum value of R_s . $R_{s(max)}$ as a function of R_p is shown in [Figure 46](#). Note that series resistors affect the output fall time.

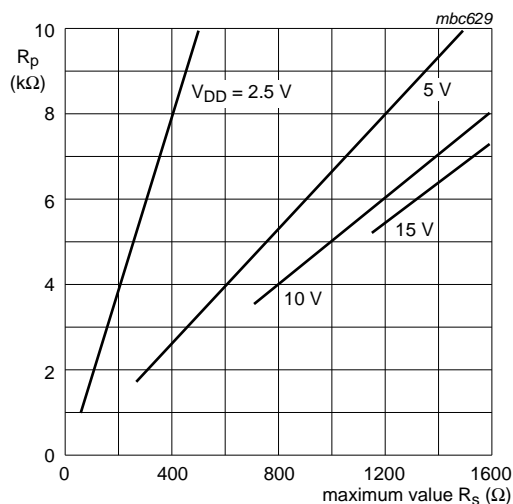


Fig 46. Maximum value of R_s as a function of the value of R_p with supply voltage as a parameter

7.4 Input leakage

The maximum HIGH level input current of each input/output connection has a specified maximum value of 10 μ A. Due to the required noise margin of $0.2V_{DD}$ for the HIGH level, this input current limits the maximum value of R_p . This limit depends on V_{DD} . The total HIGH-level input current is shown as a function of $R_{p(max)}$ in [Figure 47](#).

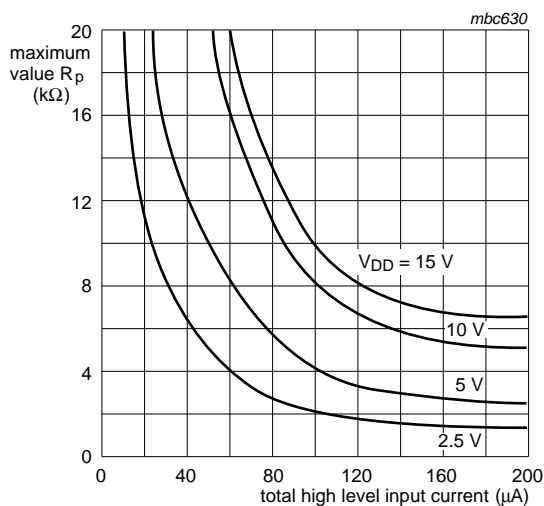


Fig 47. Total HIGH-level input current as a function of the maximum value of R_p with supply voltage as a parameter

7.5 Wiring pattern of the bus lines

In general, the wiring must be chosen so that crosstalk and interference to/from the bus lines is minimized. The bus lines are most susceptible to crosstalk and interference at the HIGH level because of the relatively high impedance of the pull-up devices.

If the length of the bus lines on a PCB or ribbon cable exceeds 10 cm and includes the V_{DD} and V_{SS} lines, the wiring pattern should be:

SDA _____
 V_{DD} _____
 V_{SS} _____
SCL _____

If only the V_{SS} line is included, the wiring pattern should be:

SDA _____
 V_{SS} _____
SCL _____

These wiring patterns also result in identical capacitive loads for the SDA and SCL lines. If a PCB with a V_{SS} and/or V_{DD} layer is used, the V_{SS} and V_{DD} lines can be omitted.

If the bus lines are twisted-pairs, each bus line must be twisted with a V_{SS} return. Alternatively, the SCL line can be twisted with a V_{SS} return, and the SDA line twisted with a V_{DD} return. In the latter case, capacitors must be used to decouple the V_{DD} line to the V_{SS} line at both ends of the twisted pairs.

If the bus lines are shielded (shield connected to V_{SS}), interference is minimized. However, the shielded cable must have low capacitive coupling between the SDA and SCL lines to minimize crosstalk.

8. Abbreviations

Table 15. Abbreviations

Acronym	Description
A/D	Analog-to-Digital
ATCA	Advanced Telecom Computing Architecture
BMC	Baseboard Management Controller
CMOS	Complementary Metal-Oxide Semiconductor
cPCI	compact Peripheral Component Interconnect
D/A	Digital-to-Analog
DIP	Dual In-line Package
EEPROM	Electrically Erasable Programmable Read Only Memory
HW	Hardware
I/O	Input/Output
I ² C-bus	Inter-Integrated Circuit bus
IC	Integrated Circuit
IPMI	Intelligent Platform Management Interface
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LSB	Least Significant Bit
MCU	Microcontroller
MSB	Most Significant Bit
NMOS	Negative-channel Metal-Oxide Semiconductor
PCB	Printed-Circuit Board
PCI	Peripheral Component Interconnect
PMBus	Power Management Bus
RAM	Random Access Memory
ROM	Read-Only Memory
SMBus	System Management Bus
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

9. Legal information

9.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

9.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental

damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

9.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are the property of their respective owners.

I²C-bus — logo is a trademark of NXP Semiconductors N.V.

10. Contents

1	Introduction	3	4.2.2	Time-out feature	33
2	I²C-bus features	3	4.2.3	Differences between SMBus 1.0 and SMBus 2.0	33
2.1	Designer benefits	4	4.3	PMBus - Power Management Bus	34
2.2	Manufacturer benefits	5	4.4	Intelligent Platform Management Interface (IPMI)	34
2.3	IC designer benefits	6	4.5	Advanced Telecom Computing Architecture (ATCA)	35
3	The I²C-bus protocol	6	4.6	Display Data Channel (DDC)	35
3.1	Standard-mode, Fast-mode and Fast-mode Plus I ² C-bus protocols	6	5	Bus speeds	35
3.1.1	SDA and SCL signals	8	5.1	Fast-mode	36
3.1.2	SDA and SCL logic levels	9	5.2	Fast-mode Plus	36
3.1.3	Data validity	9	5.3	Hs-mode	37
3.1.4	START and STOP conditions	9	5.3.1	High speed transfer	37
3.1.5	Byte format	10	5.3.2	Serial data format in Hs-mode	38
3.1.6	Acknowledge (ACK) and Not Acknowledge (NACK)	10	5.3.3	Switching from F/S-mode to Hs-mode and back	40
3.1.7	Clock synchronization	11	5.3.4	Hs-mode devices at lower speed modes	41
3.1.8	Arbitration	11	5.3.5	Mixed speed modes on one serial bus system	42
3.1.9	Clock stretching	13	5.3.6	Standard, Fast-mode and Fast-mode Plus transfer in a mixed-speed bus system	44
3.1.10	The slave address and R/W bit	13	5.3.7	Hs-mode transfer in a mixed-speed bus system	44
3.1.11	10-bit addressing	15	5.3.8	Timing requirements for the bridge in a mixed-speed bus system	45
3.1.12	Reserved addresses	17	5.4	Ultra Fast-mode	46
3.1.13	General call address	17	6	Electrical specifications and timing for I/O stages and bus lines	46
3.1.14	Software reset	19	6.1	Standard-, Fast-, and Fast-mode Plus devices	46
3.1.15	START byte	19	6.2	Hs-mode devices	50
3.1.16	Bus clear	20	6.3	Ultra Fast-mode devices	53
3.1.17	Device ID	20	7	Electrical connections of I²C-bus devices to the bus lines	55
3.2	Ultra Fast-mode I ² C-bus protocol	23	7.1	Pull-up resistor sizing	55
3.2.1	USDA and USCL signals	25	7.2	Operating above the maximum allowable bus capacitance	56
3.2.2	USDA and USCL logic levels	25	7.2.1	Reduced f _{SCL}	56
3.2.3	Data validity	25	7.2.2	Higher drive outputs	56
3.2.4	START and STOP conditions	25	7.2.3	Bus buffers, multiplexers and switches	56
3.2.5	Byte format	26	7.2.4	Switched pull-up circuit	57
3.2.6	Acknowledge (ACK) and Not Acknowledge (NACK)	27	7.3	Series protection resistors	58
3.2.7	The slave address and R/W bit	27	7.4	Input leakage	59
3.2.8	10-bit addressing	28	7.5	Wiring pattern of the bus lines	60
3.2.9	Reserved addresses in U _{Fm}	29	8	Abbreviations	61
3.2.10	General call address	30			
3.2.11	Software reset	30			
3.2.12	START byte	30			
3.2.13	Unresponsive slave reset	31			
3.2.14	Device ID	31			
4	Other uses of the I²C-bus communications protocol	32			
4.1	CBUS compatibility	32			
4.2	SMBus - System Management Bus	32			
4.2.1	I ² C/SMBus compliancy	32			

continued >>

9	Legal information	62
9.1	Definitions.....	62
9.2	Disclaimers.....	62
9.3	Trademarks.....	62
10	Contents	63

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.

© NXP Semiconductors N.V. 2014.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 4 April 2014

Document identifier: UM10204