Dominik Hollerweger, BSc

Design and Implementation of a Novel Power Aware Audio Interface

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Electrical Engineering and Audio Engineering

submitted to

Graz University of Technology

Supervisors

Ass.Prof Dipl-Ing. Dr.techn. Christian Steger

Dipl-Ing. Klaus Strohmayer (USound GmbH)

Institute of Technical Informatics
Head: Univ.-Prof. Dipl-Inform. Dr.sc.ETH Kay Uwe Römer

Graz, June 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Zusammenfassung

Ziel dieser Arbeit war es eine Optimierungsmöglichkeit für einen, sich in der Entwicklungsphase befindenden, ASIC zu finden. Dies soll sich in reduziertem Energieverbrauch äußern. Durch die Einschränkung auf den digitalen Bereich konnten zwei verschiedenen potentielle Verbesserungen gefunden werden. Eine Variante bezog sich auf die Anwendung verschiedener Kodierungsmöglichkeiten, mit dem Ausblick die Schaltaktivität zu reduzieren. Die andere Option war, zwei bereits auf dem Chip implementiert Schnittstellen durch eine neuere, verbesserte zu ersetzen.

Vielfältige Kodierungen von Audio Daten wurden getestet. Diese brachten keine zufriedenstellenden Ergebnisse, um sie in den Chip zu integrieren. Die Recherche bezüglich anwendbarer Schnittstellen, konnte eine vielversprechende Optimierung aufzeigen. Eine neuartige Schnittstelle wurde, auf Basis einer vorhandenen Spezifikation, entworfen und implementiert. Konfigurationsmöglichkeiten, welche nicht essenziell für die Datenübertragung sind, konnten stark reduziert werden. Durch die Kontrolle eines Slave-Gerätes wurde die Funktionalität demonstriert.

Nachteil dieser Schnittstelle ist die geringe Anzahl an weltweit vorhandenen Adaptierungen zum jetztigen Zeitpunkt. Durch diese Arbeit wurde es ermöglicht diese schneller zu integrieren sollte es zu einer Marktumstellung in diesem Segment kommen.

Abstract

The goal of this thesis is to find an optimization possibility in the sense of energy consumption for an ASIC in the development phase. This brings up a broad versatility. When reduced to digital field of improvement, two major opportunities spiked out. One concerned data encoding techniques with a view to reduce switching activity. The other treats interfaces including a alternative replacement, resulting in a reduced wiring.

Regarding primary, no deliverables for an enhancement could be achieved. Extensive search on various interfaces opened up design and implementation according to available specifications. A feature reduction to optimize expenditure on the ASIC was intended as some of them were insignificant. Controlling a slave device functionality was demonstrated, showing off a replacement possibility for two interfaces integrated at the moment.

The drawback is a weak market pervasion that has the potential for rapid change. Through this thesis a fast adaption to this novel interface is made easier.

Contents

List of Figures

List of Figures

Abbreviations

AC-Link	Audio Codec - Link
ACK	Acknowledge
ASIC	Application-Specific Integrated Circuit
APB	Advanced Peripheral Bus
A2B	Automotive Audio Bus
BRA	Bulk Register Access
CDC	Clock Domain Crossing
CMOS	Complementary Metal Oxid Semiconductor
CMT	Clock Management Tile
DAI	Digital Audio Input
DCM	Delay Locked Loop
DDR	Double Data Rate
DMA	Direct Memmory Access
DP	Data Port
DSP	Digital Signal Processor
ESAI	Enhanced Serial Audio Interface
ESSI	Enhanced Synchronous Serial Interface
FF	Flip-Flop
FFT	Fast Fourier Transformation
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
HDA-Link	High Definition Audio - Link
IC	Integrated Circuit
IDDR	Input Double Datarate Register
ILOGIC	Input Logic
I²S	Inter IC Sound
I²C	Inter-Integrated Circuit
LFSR	Linear Feedback Shift Register
MEMS	Micro-Electro-Mechanical System
MIPI	Mobile Industry Processor Interface
MLB	Media Local Bus
MMCM	Mixed-Mode Clock Manager
MOST	Media Oriented Systems Transport
MUX	Multiplexer
NAK	Not Acklowledge
NRZI	Non - Return to Zero Inverted

ODDR	Output Double Datarate Register
OEFNSC	Odd Even Full Normal Self Coupling
OLOGIC	Output Logic
PRBS	Pseudo Random Binary Sequence
PCM	Puls Code Modulation
PLL	Phase Locked Loop
POR	Power On Reset
QFN	Quad Flat No Leads Package
SAIF	Switching Activity Interchange Format
SAI	Serial Audio Interface
SCP	Slave Control Port
SDK	Software Development Kit
SDW	SoundWire
SLIMBus	Serial Low-power Inter-chip Media Bus
SPI	Serial Peripheral Interface
SSI	Synchronous Serial Interface
TDM	Time Division Multiplexing
WLP	Wafer Level Package

1 Introduction and Motivation

1.1 USound

USound GmbH is a fast growing fabless Audio company having the main facilities in Graz and Vienna. The so called Ganymede is the core product of USound. It's a Micro-Electro-Mechanical System (MEMS) Speaker (see Andrea Rusconi Clerici, 2018) based on piezoelectric technology. For this use-case the piezoelement is supplied by a voltage thus experiencing a deflection. Regarding dimensions of 4.7 x 6.7 x 1.6 millimeter Ganymede is unconquerable on the market making it the ideal for integration into all kinds of audio wearables. Recently developed product is a revolutionary audio eyewear FOCUS having MEMS integrated for playing music, answering phone calls and supporting daily routine. USounds R&D department includes an Application-Specific Integrated Circuit (ASIC) development team responsible for creating a special MEMS driver called Leda and opening up research strategies for improvement in efficiency.

1.1.1 Leda - Target of Interest

Devices such as MEMS gain higher value as their production and integration in novel systems are cost effective and open up various possibilities. Comparing common electrodynamic speakers and piezoelectric MEMS developed by USound, the main difference is a crossing from an inductive component to an capacitive one. This induces various advantages like fast fabrication or lower heat dissipation but also some drawbacks like the non linearity depending on applied voltage and membrane displacement. Due to the novelty no drivers, aware of specific MEMS behaviour are available yet. Thus USound initiated the development of it's own ASIC called Leda creating a highly efficient component. After their final development the chip and speaker should be available as an assembled all in one component thus making integration into audio wearables as easy as possible. Especially for this type of integration it's important to consider each possibility of power reduction and keeping things small. The novelty opens up many subjects for further research. This thesis focuses on power awareness of digital design, size, and wiring harness related to interfaces for chip communication.

1.2 Motivation for Digital Low Power Design

In previous decades development of digital low power design gained more and more importance as products shrank in size enabling integration into all kinds of wearables like wristbands, watches or glasses. One unique selling point of those gadgets is driven by run-time. Longer periods between charging the batteries, of course depending on item usage, reduces negative effects like lowered capacitance due to overridden battery life-time charging cycles. Additionally, consumers are able to use products for a longer time without renouncing on it. Thus, it's important to reduce power consumption to the minimum needed, implying elimination of energy waste related to temporarily unneeded parts as example. The following subsections discuss some of most common techniques to achieve this objective.

1.2.1 Switching Power Reduction

Switching power describes how much energy needs to be provided to charge the load capacitance in case of a state transition from 0 to V_{dd}. Complementary Metal Oxid Semiconductor (CMOS) power dissipation can be divided into mainly two components. Internal cell power and power driving the load capacitance (compare Abdellatif Bellaouar, 1995 and Panda Preeti, 2010). There are several contributing influences in regards to internal cell power.

One part is short circuit power. Each transistor switching causes this effect because of non zero rise and fall time. During the state change there is a small time slot where supply and ground are shorted establishing a current flow depending on the load capacitance. Especially for high load caps this effect has to be considered.

Another contributor to internal cell power dissipation is related to leakage power. During the powering of a transistor there will be a small amount of leakage caused by several effects namely "Reversed Biased Diode Leakage", "Gate Induced Drain Leakage", "Gate Oxide Tunneling" and "Subthreshold Leakage".

For later considerations power dissipation (dynamic power P_d) according to supply voltage V_{dd}, caused by driving load capacitances C_L, is relevant. Starting with the basic formula for the average dynamic power:

$$P_d = \frac{1}{T} \int_0^T i_o(t) v_o(t) dt \tag{1.1}$$

At charging phase one half of the energy is stored in the load capacitance and the other is dissipated into heat. By inserting output current i_o during charging and discharging phases this leads to equation 1.2.

$$P_d = C_L * {V_{dd}}^2 * f \tag{1.2}$$

Regarding equation 1.2, one would assume switching to take place each clock cycle according to frequency f . In general, one can say this is not the case. Thus formula 1.2 has to be extended by a factor. The switching activity factor α leading to 1.3.

$$P_d = \alpha * C_L * {V_{dd}}^2 * f \tag{1.3}$$

α is determined by the probability a capacitive node is being switched. This brings us to one key task of low power digital design, namely reducing α to the minimum needed for running expected operations. There are immense possibilities to approach a reduction of switching activity. Clock gating discussed in subsection 1.2.2 is one. Later in section 2.2, encoding techniques to reduce transitions are treated more in detail for a investigation of possible switching reduction.

1.2.2 Clock Gating

Clock Gating describes a technique very similar to the higher level power gating. In case of power gating, whole components are being turned OFF and ON again depending on their usage. Regarding clock gating the clock is enabled or disabled for a certain amount of time. E.g. consider a I²S interface responsible for audio transmission. In case no audio data is scheduled for transaction, there is no need for a Master Clock, Bit Clock or Word Select. Thus these clocks or the whole component could be switched off saving power.

The following example (compare Rakesh Chadha, 2012) shows the difference between a Flip-Flop (FF) enable signal and clock gating leading to the same output when required. Picture 1.1 shows a flip-flop and a multiplexer on the left side. The Multiplexer (MUX) feeds a FF with new input data if the enable signal is high. The other way round, Q (output of the Flip-Flop) is feed back and via the multiplexer applied to the FF again, leading to no change of the output. Regarding the clock input there is no change thus power is dissipated as long as the circuitry is turned on. This brings up a possibility for clock gating to take action.

The other displayed structure in 1.1 represents the kind of a Flip-Flop including clock gating. Input D is only captured if there is a clock applied to the FF, corresponding an enable signal input. This equals high input at the Clock Gating Cell. Assuming output of the Flip-Flop to be utilized every 10th clock cycle the reduction of power corresponds to about 90%. Inserting Clock Gating Cells should be considered wisely though. Applying clock gating to each and every FF can also lead to performance losses. One reason is due to timing requirements and the others is related to power consumption of Clock Gating

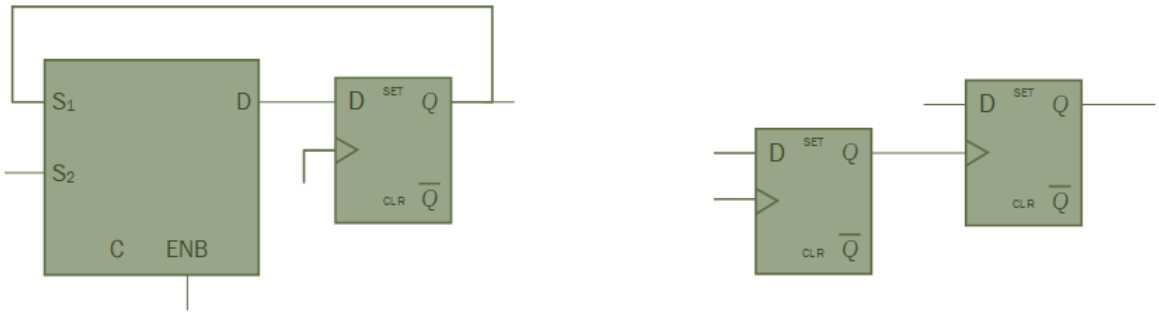


Figure 1.1: Clock Gating Versions

Cell's. The so called Switching Activity Interchange Format (SAIF) is used to determine the activity of a clock and provides decision driving data. One very effective integration treats systems where a whole bus can be turned OFF and ON e.g. a 32-bit wide bus is applied to 32 Flip-Flops. All clocks applied to those Flops can be switched by one Clock Gating Cell.

1.2.3 Gate Sizing

Gordon Moore stated in 1965 that the complexity of integrated circuits would double every two years. This statement well known as Moore's law is related to the number of components being integrated into the same area. Starting with the mass production of Integrated Circuit (IC)'s in 1960 at a technology node of 50 micrometer, in 1999 already 180 nanometre were offered by several semiconductor fabs. Since the beginning of 2019 it has been possible to order 7 nanometre processes at Samsung and TSMC.

Year		2009	2012	2015	2018	2021	2024
Physical gate length in a FET transistor inside of ICs (microprocessor unit – MPU)	nm	29	22	17	12.8	9.7	7.4
Clock frequency (on chip MPU)	GHz	5.45	6.82	8.52	10.6	13.3	16.6
Functionality of IC (number of transistors)	mln	2212	4424	8848	17696	35391	70782
Supply voltage	V	1.0	0.9	0.81	0.73	0.66	0.60
Dissipated power (cooling on)	W	143	158	143	136	133	130

Figure 1.2: Data of integrated circuits (IC) according to the Report of The International Technology Roadmap for Semiconductors (Edition 2009) (see Nawrocki, 2011)

Figure 1.2 shows the roadmap for semiconductors in 2009. As we know now 7nm nodes are already in production and further decreasing down to 1.5nm is predicted till 2029 (see Cutress, 2019). Various problems had to be solved to go further in this decreasing process. Following an overview, regarding key

issues to be solved, for scaling CMOS technology in 1997 below 100nm is given (compare literature Yuan et al., 1997):

- Lithography: A mask is used to transfer an image onto a light sensitive photoresist by lithography
- Power Supply: Voltage needs to be decreased as continuously more transistors are integrated
- Gate Oxide: Source, Drain and Channel are separated by a gate oxide which is desired to be reduced in relation to channel length keeping short-channel effect under control
- Short-channel Effect: Due to downscaling transistors lead to deviation of long-channel behaviour. Effects like increased punch-trough risk have to be considered
- High Field Effects: As supply power is not changed in same ratio as channel length, field strength increases
- Dopant Number Fluctuations: Reaching sub 100nm dopants are in the order of hundreds. Thus distribution of them generate non-negligible effects on threshold voltage
- Interconnect Delays: Increasing processor speed comes with improvement of device speed. Delays caused by wire resistance and capacitance have to be shrunk

By the process of downscaling depicted in 1.2 not only is lower size a positive effect, but also decreasing voltage opens up possibilities for increased clock frequency. This enables lower power dissipation in addition to higher processing speed. Though it's not always possible to implement this caused by restrictions related on application area.

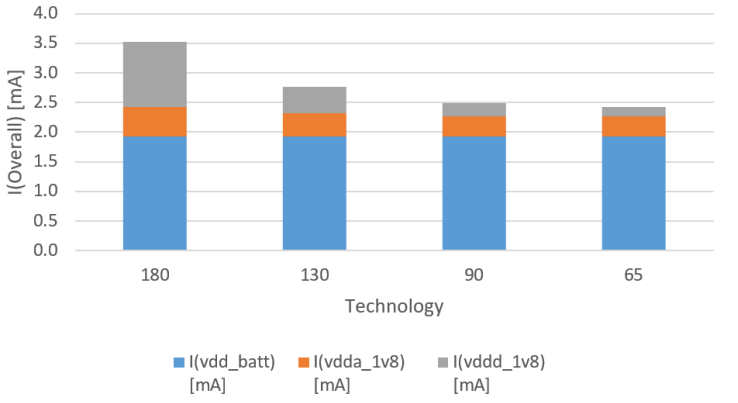


Figure 1.3: Technology Node depending Current of Leda

Currently USound's ASIC team utilizes a 180nm process of TSMC. Several considerations of gate size reduction were already discussed. As the semiconductor consists of a digital and analogue part there has to be a trade-off for

both. Figure 1.3 shows efficiency of the reduction regarding Leda's digital part. Development-phase of analogue functionality doesn't allow further reduced voltage. Thus minimizing the size to 45nm wouldn't be meaningful whereas 130nm opens up a opportunity. Changing to 130nm decreases digital (displayed in grey) power consumption by half. Due to several internal reasons within the company this can't be taken as advantage at the moment.

1.2.4 Voltage and Frequency Scaling

In general power consumption of a circuitry increases by it's number of computations. Delay of CMOS gates sets limits for applied frequency they can run at. Though by scaling supply voltage it's possible to vary clock frequency (direct relation to gate delay) with some restrictions. This technique is called voltage and frequency scaling. (compare Panda Preeti, 2010). Changing voltage and frequency at runtime is called dynamic voltage and frequency scaling.

Gate delay τ_G is defined as:

$$\tau_G = k \cdot C_L \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{1.4}$$

k corresponds to the gain factor, C_L is the load capacitance consisting of the gate capacity, wire capacity and the input capacity of following gate. Supply voltage is given as V_{dd} and the threshold voltage as V_t. A key challenge here is meeting timing requirements while saving energy by scaling voltage and frequency. This method gains relevance for systems having a dynamic processor speed adjustment depending on instantaneous load, especially when thinking about DSP's. Analysis showed that reduction of speed, including a delivery of computational results within time constraints, is more energy efficient than computing the result and running in idle operation mode. SoundWire, a protocol being introduced to you later is capable of frequency scaling as-well.

1.2.5 Further Important and Decision Driving Influences

Previously some basics of digital low power design had been discussed. At least one of those, namely Clock Gating is already implemented partially. Whereas others like Voltage and frequency scaling won't be considered further at the moment. In this section more decision driving influences regarding advancement of this thesis will be discussed including wiring harness, Area Expenditure and Data Encoding.

Wiring Harness

Powering silicon and communication with it is done via wires. Depending on implemented features, this can be significantly more or less. In comparison with on-chip connections a multiple of the space is required. Thus it has to be considered which kinds of signal output and input need to be provided. This includes not only signals which are planned to be used in the field of application after finishing the silicon, but also important test signals in the development and improvement process. Sometimes area is not as important as a very fast data transportation possibility. Herein implementation of parallel buses is recommended rather than an serial connection implementing Time Division Multiplexing (TDM).

In regard to the automotive industry, there is an ongoing weight economization process in each and every possible part. Reducing for example the hi-fi audio connectivity in cars from 5 down to 2 wires will also have an impact on fuel consumption, a key requirement for the automotive market. Wiring harness is a decision driving influence in development process of USounds Leda silicon. Audio wearables are very small devices decreasing further at the moment. One main field of application will be in-ear headphones. Thus there is few space for integration of a chip. Another section is free field usage. Therefore MEMS need support to drive lower frequencies by another speaker. Specifically this scenario makes it important to lower wiring.

In figure 1.4 one can see a pinout including 48 pins. Most of them are used for testing reasons. Actually needed are at least connections for I²C, I²S and power supply requiring 7 wires:

- I²S: 3 wires

- I²C: 2 wires

- Power Supply: 2 wires

Compared to conventionally in-ear headsets connected by two wires, this points out the serious effect. Not only the wire thickness and weight are affected, but also the chance breaking one wire out of seven is much higher than one out of two. Aware of those impacts USound wanted to know if there has been any change in technology in last years opening up a possibility to reduce wiring harness. In chapter 2 familiar interfaces are discussed as well as one novel transmission protocol, leading to the main part of this thesis including a design and implementation chapter (3 & 4) of the SoundWire Interface.

Area Expenditure

On-chip area consumption by different components is shown in this subsection. Figure 1.4 depicts Leda (version 1.1 of tape-out April 2019). Table 1.1 lists some of digital components including area expenditure on silicon. Contemplating

Table 1.1: Digital Leda Area Expenditure

Module	Cell Area [nm²]	Cell Area [%]
Digital Core	2765482.263	100
DCDC	1782873.203	64.5
Upsampling	268427.174	9.7
Filter	256805.212	9.3
I²C	78825.701	2.9
I²S	36204.538	1.3
...	...	...

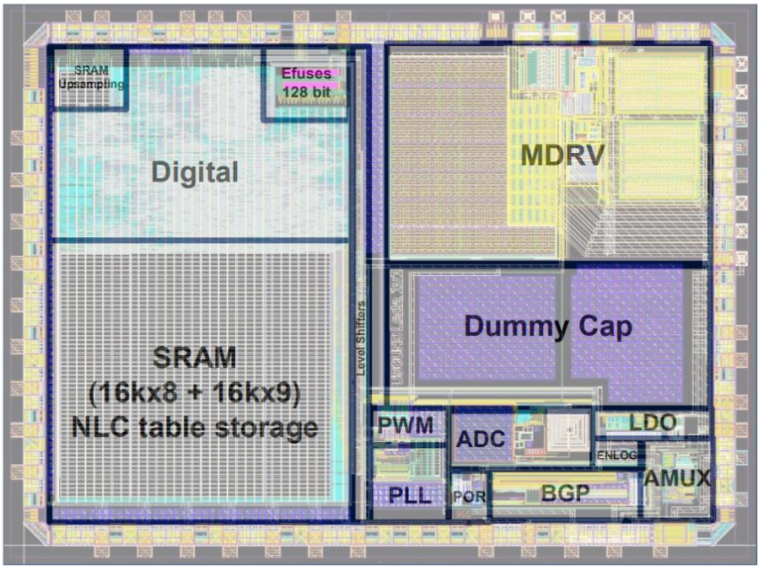


Figure 1.4: Leda Chip Plot

the figure, one can immediately observe a big difference between the section related to digital on the left and the analogue part on the right. Surrounded by red is the area related to the Digital Core in table 1.1.

For later considerations we shall remember area related to data interfaces I²S and I²C. Only 4.2% of area related to digital core is used for both interfaces in total 115030.239nm². Comparison to a novel audio interface is made in section 4.4 under subsection 4.4.1.

Data Coding

Nowadays data coding is used in almost every electronic device. Especially removing redundant data to save data-storage and reducing transmission amount is significant. This is called data compression. Important differentiation can be made between lossy and lossless encoding. Comparing images and audio, lossy data processing leads to fuzziness on the one hand and dullness on the other. Application of lossless encoding instead results in a fully recoverable data set.

Due to quantitative data reduction, transmission is already more power efficient. The relationship between lowering energy consumption and switching power reduction is treated below (compare section 1.2.1). Regarding Leda there are two kinds of data namely audio data supplied by the I²S interface and register data for setup brought up by using a I²C port. After short discussion about the types of data an example for lowering switching activity will be shown.

Audio Data: Waveforms are picked up by a microphone transforming alternating pressure into a voltage. The voltage is sampled by a certain sampling-rate f_s and each sample quantized depending on the desired resolution. Multiplying those leads to the bitrate (1.5) increasing by the amount of used channels x as well. One commonly used combination is 48kHz sampling rate quantized by 16-bits. If only one channel is used (x = 1) bitrate is 768000 bit/s.

$$Bit/s = f_s \cdot bits \cdot x \tag{1.5}$$

One sample is mapped to a number between 0 to 65536 or -32768 to 32767 (scope of 2^{16}) depending on changing voltage. For transmission via I²S the resulting number has to be encoded into the so called two's complement. Ranging from -2^{n-1}, ...0,...,$2^{n-1}-1$ this number format mapps them into binary values like the following (only 4-bits are used to keep it simple).

- 16 $\rightarrow$ 1111
- 0 $\rightarrow$ 0000
- 15 $\rightarrow$ 0111

The first digit is used for signalling positive or negative values. Positive values include zero thus maximum value is counted minus one. Thinking about digital transmission high levels refer to logic ones whereas low level is mapped to zeros. This is the most intuitive way. There are other possibilities as well like mapping the other way around. Considering a value in two's complement. Lets take number 85 equally 01010101 in the binary format. This example shows the maximum of possible level changes regarding 8 bit digits. Decreasing the amount of logical switches is the same as lowering switching thus saving power. Several options for data encoding application related to different points of operation are discussed more precisely in section 2.2.

Register Data: Register data is used for device setup and configuration. I²C is a proper interface transmitting related data. Therefore, an address and register-information to be stored are needed (further description in 2.1.1). For switching reduction applications one could consider mapping high frequented registers to low switching addresses and the other way for lower ones. Another influence for switching power reduction can be a reordering of register content. E.g. turning on a device could be done via one bit out of 8. Using the first or last one of those

digits is more meaningful than putting it into the middle as there won't often be a change. Together this kind of switching reduction is negligible compared to others, although it should be considered for high performance systems.

2 State of the Art

This chapter is primarily split into two parts. The first one, 2.1 speaks to audio interfaces. The general structure and their field of application are discussed. Additionally, examples of present integration are described. Encoding of audio data is the basis for the second chapter. Various possible techniques are listed whereas some of them have been tested for meaningfulness reasons. An adjusted encoding possibility has also been tried (see paragraph 2.2.3).

At the end of both the results are discussed leading to chapter 3 Design of the SoundWire Controller Interface.

2.1 Interfaces

In the following subsections several evaluated audio interfaces according to table 2.1 are introduced. A description of an I²C port is given in advance due to the fact of its implementation in Leda and relevancy for comparing reasons in the sense of wiring and area economization.

2.1.1 I²C

I²C, short for Inter Integrated Circuit, is a standard interface introduced by Philips Semiconductors in 1982 (Compare further to NXP, 2014). Since then, it has spread worldwide, implemented into a huge number of devices by over 50 companies. An advantage of this direct on-chip interface is that it enables communication to others independent of either system designers or equipment manufacturers.

With two wires the whole connectivity is set up:

- SCL: Serial Clock Line

- SDL: Serial Data Line

Devices attached to the interface are addressable by a unique ID. Throughout operation a master slave behaviour is set up. Data transfer is either unidirectional or bidirectional, both 8-bit oriented. In general, it's possible to connect as many devices as desired just limited by bus capacitance. The transactions are started and ended by a combined signal behaviour of SDA and SCL. Selection of a connected slave in between others is done by sending the ID a unique address

on the bus which has to be confirmed by the appealed, sending an acknowledge. All others disconnect themselves from the bus until the finished exchange. Data is transferred in both directions depending on read or write commands coming from the master. As long as there have been no end signalling this connection stays stable, enabling further data transmission all requiring an receiver Acknowledge (ACK). Using this interface, all control functionalities of Leda are set up by configuration of registers.

2.1.2 Audio Interfaces

Table 2.1 lists spotted audio interfaces. For USound it's not practicable utilizing an interface based on parallel transmission thus "8-bit Parallel" (P in the table) is not mentioned further. Puls Code Modulation (PCM) actual refers to a data compression type applying a-law or u-law (contrary encoding types for different countries). Some companies like Texas Instruments designed a PCM codec (see TexasInstruments, 2001) specifying the interface PCM. in general it's designed likewise the Bi-directional I²S in sense of wiring TDM and other characteristics (see table 2.1). For this reason the interface is not mentioned any further.

Table 2.1: Evaluation of Audio Interfaces

Interface Feature	P	I²S	Bi-I²S	PCM	AC	HDA	SSI	ESSI	ESAI	MLB	A²B	SLIM	SW
Industry Standard	+	+	-	+	+	+	-	-	-			?	?
Device Control	+	-	-	-	+	+	-			+	+	+	+
Bi-directional Data	+	-	+	+	+	+	+	+		+		+	+
Supported Devices		1	1	1	up to 4	up to 4	1-32		4-?	1-64	1-64	1-32	11
16-bit 48kHz str. chan		1	1 I/O	1 I/O	4 I/3 O	15 I/O	8 I/O				14	14	
Pins	12	3	4	4	5-7	min 5	3-6	3	3	3	2	2	2
Isochronus Data Flow		-	-	-	+	+	-			+	?	+	+
Bit Width Audio Data	8	any	any	any	16-20	8-32	24	8-32	8-32		8-32	4-124	
Samplerate (sup.)		16/32	16-48	44,1-88,2	8-192	8-48			44,1/48	44,1/48			+
Samplerate Change		-	-	-	+		-			-	-	+	+
Seamless CLK Scaling		-	-	-	-		-	-	-			+	-?
Frame Rate		var	var	var	48	48	var			var		var	var

2.1.3 I²S - Inter IC Sound

Inter IC Sound or more common I²S (see PhilipsSemiconductors, 1996) is a transmission protocol developed by Philips Semiconductors (nowadays NXP Semiconductors) in 1986. Integrated in a huge number of devices this port is one of the most common interfaces. It defines a serial link specially created to send digital audio data from one integrated circuit to another. It's possible to configure different bit rates e.g. 8, 16 or more. The interface utilizes 3 to 4 physical wires described below:

- SD = Signal Data
- WS = Word Select
- $SCLK$ = Serial Clock
- $MCLK$ = Master Clock

SD is in charge of transmitting the desired audio data. Word Select line defines which channel is being transmitted at the moment. Thinking of TDM it's possible to send stereo audio data via the channel. WS changes the level one clock period before the next data word starts. This is because the slave requires some amount of time to store the received data and pick up the next ones. Serial Clock defines the time between two consecutive bits. This timeslot has to be complied by the SD line. Differences caused by any participant can lead to wrong assignment of the audio data thus create transmission errors. SCK can be derived by the Master Clock. The MCK has to be provided by the master interface. Using any other Clock e.g. from another quartz crystal can cause substantial Jitter errors. There are three common system configurations deciding about master and slave hierarchy in the communication system depicted in figure 2.1. It's possible for either transmitter or receiver to take over the master position as well as being external controlled by another device. The Master device effectively provides the MCK.

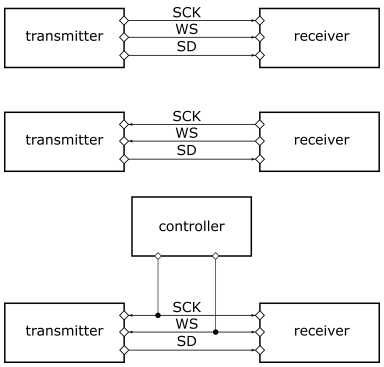


Figure 2.1: I²S Configurations

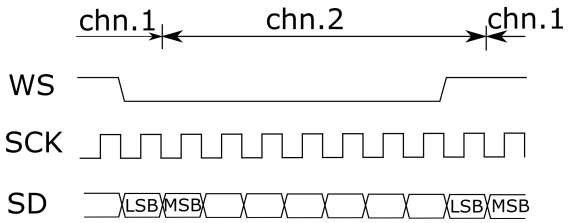


Figure 2.2: I²S Protocol

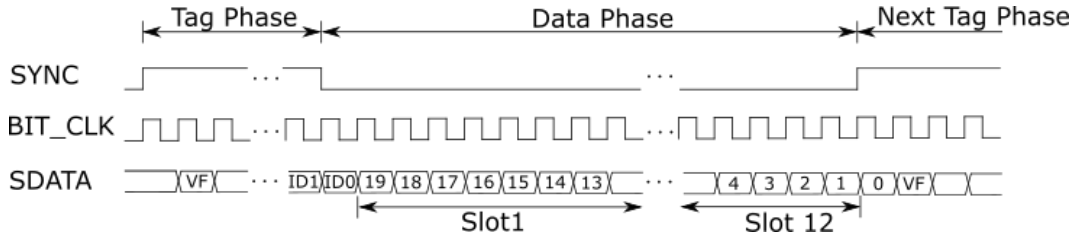


Figure 2.3: AC-Link Protocol

2.1.4 Bi-Directional I²S

Bi-Directional I²S extends the previously described I²S interface. An additional physical signal data line enables data transmission in both directions. Configurations can all be done in same manner as before. Applications for this interface concern e.g. Noise Cancelling (microphone and speaker integrated in one device). To process required audio data the transmission must be bi-directional. Data rate stays the same as clock and word select are unique.

2.1.5 AC-97 - Audio Link 97'

Audio Codec - Link (AC-Link) refers to an interface developed by Intel Architecture Labs in 1997 (see Intel, 2002). It was designed for connecting sound-cards, modems and motherboards in a PC environment. There are five wires are necessary for a connection comprising of:

- $SYNC$ = Synchronization
- BIT_CLK = Bit Clock
- SD_O = Signal Data Output
- SD_I = Signal Data Input
- RST = Reset

This interface is capable of transmitting TDM audio data for 44, 1kHz and 48kHz qualities. It's possible to send 12x 20-bit slots on the bi-directional interface via the channels SD_I and SD_O. Thus, a master clock fixed at 12.288MHz is needed to provide necessary granularity. In the AC-Link a power saving mode is implemented allowing to halt the clock, sync and data signals. Figure 2.3 shows the structure of an AC-Link output frame which is very close to the input frame format. Each frame consists of 16-bits tag phase determined by a logic high period of the SYNC signal. This it then followed by an up to 12x 20-bits wide Data Phase on logic low of SYNC. Further description of the frame construction can be looked up in Intel, 2002. Generally tag phase concerns validation of the whole frame (slot 0, bit 15) as well as confirmation of different data channels transporting information.

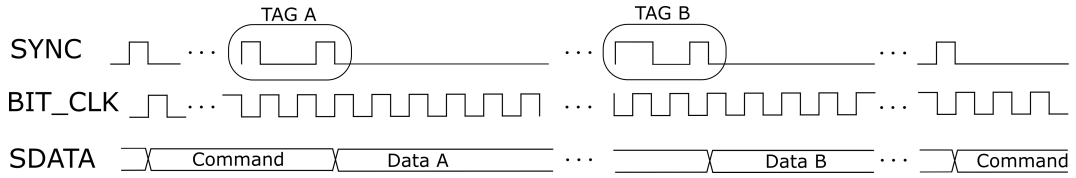


Figure 2.4: HDA-Link Protocol

2.1.6 HDA Link - High Definition Audio Link

High Definition Audio is the successor of AC'97 released by Intel in 2004. It's not designed to be backward compatible to its predecessor. PCI-Express defines the connection to PC's motherboards. The interface connecting HD-Audio controllers to different HD-Audio codecs is called High Definition Audio - Link (HDA-Link). This protocol is purely isochronous based on a 48kHz framing period. Similar to the previously described AC-Link, the following five listed wires are needed for connection:

- $SYNC$ = Synchronization
- BIT_CLK = Bit Clock
- SD_O = Signal Data Output
- SD_I = Signal Data Input
- RST = Reset

Bit Clock is fixed to the 24MHz clock sourced from the controller device and provided to all attached peripheries up to a maximum of 15 codecs. Frame synchronization is done via an 8-bit wide word whereas the falling edge of the eighth bit indicates the beginning of the next frame. A frame consists of a command proportion and different streams indicated by a stream tag of the SYNC wire. The tag structure includes a 4-bit preamble followed by the Stream ID of same size shown in figure 2.4. Bi-directionality is implemented again by an additional line.

For further reading concerning the HDA-Link see Intel, 2010.

2.1.7 SSI - Synchronous Serial Interface

Synchronous Serial Interface, developed by Max Stegmann GmbH in 1984, refers to a point to point connection between master and slave. Initially it was designed as differential non-multiplexed and simplex interface using two twisted pair connections. The advancement of Synchronous Serial Interface (SSI), done by Freescale/NXP/Motorola, is (depending on configuration) capable of full-duplex transmission to several devices. Three different modes can be selected: normal, on-demand and network. Normal mode is for periodic transmission of a single word, on-demand is selected for asynchronous device communication and network mode built on TDM is for transmitting up to 32 words.

A feasible configuration for audio data transmission includes listed connections:

- TXD = Transmit Data
- SC = Frame Synchronization
- $SCLK$ = Serial Clock

Depending on the interfaces operating mode, various wiring is necessary. Applicable configurations including different wiring of a continuous (normal/network) mode and gated (on-demand) modes can be checked in the specification Freescale Semiconductor, 1994. Area of application treats all kind of sensors data transfer.

2.1.8 ESSI - Enhanced Synchronous Serial Interface

Enhanced Synchronous Serial Interface (ESSI) describes a further development of the previous SSI. It's capable of full-duplex transmission created for codecs Digital Signal Processor (DSP)'s and other peripherals. Identical to SSI, three different modes (normal, on-demand, network) are adjustable. The enhancements concern following subjects:

- network enhancements (e.g. end of frame interrupt)
- audio enhancements (three transmitters per ESSI)
- general enhancements (e.g. trigger DMA interrupts)
- other changes (e.g. gated clock mode not is not available)

Further in-depth explanation can be found in M. Inc., 2001. The number of pins and wiring for SSI depends on the different operation modes. As ESSI is capable of receiving one signal and transmitting three signals, two units can be merged to provide surround sound.

2.1.9 ESAI - Enhanced Serial Audio Interface

Enhanced Serial Audio Interface (ESAI) was developed by Freescale Semiconductor, Inc. (nowadays NXP Semiconductors). It's a combination of the ESSI and Serial Audio Interface (SAI) (not further described in this thesis see Pavel Bohacik and Group, 2012) designed for the Motorola Symphony audio processors family (take a look at FreescaleSemiconductor, 2000). I/O sections of ESAI can be used to interface several of most common digital audio protocols like I²S. Usually, they have a 3 wire structure including one bit clock, one frame synchronization clock and one data line. ESAI shows full efficiency in multichannel processing e.g. cinema and home theatre. Therefore it contains 2 output pins as well as 4 pins programmable for input or output. Standard configurations and applications for data transfer concerning ESAI interface are further described in FreescaleSemiconductor, 2000.

All three interfaces (SSI, ESSI & ESAI) concerning the Motorola's Symphony DSP processing are not practicable solutions for USound due to their application limitation.

2.1.10 MLB - Media Local Bus

Media Local Bus (MLB) was created to provide a bridge from the Media Oriented Systems Transport (MOST) network protocol to several subsidiary devices (see figure 2.5). An MLB Controller (serving as MOST to MLB mapper) is an connected to at least one MLB device. The controller supports all methods of data transport concerning MOST architecture. Scope of this is interconnection of different audio devices in automotive application by usage of lower complexity enabling faster time-to-market. Pin count in respect to the MOST bus is kept as low as possible in between of 3 and 6 pins.

As this interface depends on MOST networks protocol it's not described any further being no optionality for integration into Leda. Interested can look up the MLB interface in the literature of SMSC, 2010.

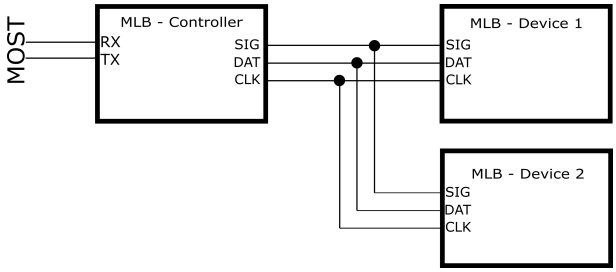


Figure 2.5: MLB Configuration example

2.1.11 A²B - Automotive Audio Bus

Automotive Audio Bus (A2B), as the name suggests, was developed for Automotive application by Analog Devices. It's a 2 wired single master to multiple slave topology capable of transporting I²S data, in addition to I²C data, over a maximum distance of 15m pin to pin. This interface provides standard audio sample rates 44.1kHz, 48kHz and supports 12, 16 and 24-bit channel width. Primarily it has been designed to reduce the weight of an in car cable harness where it can compress up to 75% of weight. The following are some target applications:

- vital signs monitoring
- smart radio connectivity
- hands-free/speech recognition/in-car communication

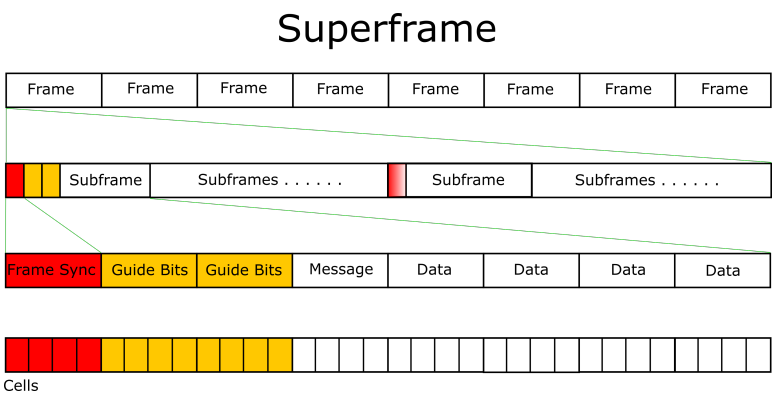


Figure 2.6: SLIMBus Superframe Structure

A very interesting feature is the possibility of applying Phantom Power by this interface. Nevertheless, it displays no viable options in terms of intended operation mode due to automotive restrictions. Further reading can be found in A. Inc., 2016.

2.1.12 SLIMBus - Serial Low-power Inter-chip Media Bus

Serial Low-power Inter-chip Media Bus (SLIMBus), developed by the so called Mobile Industry Processor Interface (MIPI) Alliance, is one of two MIPI standards mentioned in this thesis. The main features concerning this transmission protocol are low pin count (2 pins) and the possibility to transmit both audio and control data. Changing clock speed for reasons of energy saving is possible too. Nevertheless, this interface has not taken a serious position on the market now (see Alexander Khazin, 2018). As a consequence, the MIPI Alliance defined a new standard namely SoundWire discussed in next subitem 2.1.13.

SLIMBus describes a model connecting different devices applied to the same bus. By this control data such as filter coefficients in addition to audio data can be sent. It's also able to handle isochronous and asynchronous transport. For a setup connecting several devices it can be configured as synchronous multi-drop bus utilizing TDM frame structure.

The structure includes a Super-frame divided into Frames, Subframes, Slots and Cells. A Super-frame describes the superior element consisting of a total of 1536 slots consuming always the size of eight frames. A frame is constructed by several subframes but the size of them is not fixed. Subframes can be set up using 6, 8, 24 or 32 slots. One slot is related to 4 cells in turn a cell corresponds to 1 bit. (see 2.6).

The whole Super-frame is nothing but a bitstream of 6144 bits including framing information, control data and audio data (depending on setup). Each frame's 0-th and 96-th slot is reserved for framing information. In Frame zero of each Super-frame ,additional 2 slots are reserved for so called guide channel bits. Those are used to provide necessary information to the connected devices. On

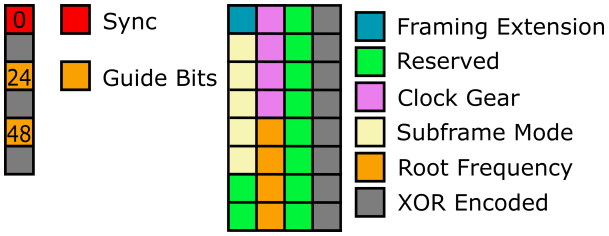


Figure 2.7: SLIMBus Control Structure

left side of figure 2.6 a frame snippet including first synchronization bit and the guide bits is shown (one square equals 4 bits). Right side represents content of guide bits (each square equals 1 bit). For further insight into details take a look at M. A. Inc., 2008.

2.1.13 SoundWire

SoundWire refers to the second audio interface concerning the MIPI Alliance. As well as previous protocol, SoundWire is based on a TDM frame and has multi-drop capabilities. It's a fairly new interface published in 2015. The market has not adopted the protocol in a big sense, but there are several modules available especially for the mobile phone industry like the CL42L42 audio codec and MAX98374 digital amplifier. On October 8th 2018, Intel released the Z390 chipset (see Intel, 2019) including a SoundWire interface in their "Intel® HD Audio Technology".

SoundWire provides audio data transmission and has embedded control and commands. Thus, it can displace need of a control interface like I²C or Serial Peripheral Interface (SPI). Bandwidths up to 20 Mbits/s are possible for fast device setup. There is a clock stop mode integrated for reduction of power consumption after Power On Reset (POR).

Appendix A shows the typical frame structure. 2 up to 16 columns are configurable as well as a minimum of 48 rows to a maximum of 256 rows, defining the protocol edges. 48-bits, out of the basic 48x2 frame (depicted in 2.8), are reserved for a control word in each Frame, remaining part can be either configured to further control or audio data. The 48-bit control-word can operate in three different conditions. Ping, Read and Write. Read and Write are relevant for register configuration and collecting device information. Standard operation is a Ping request displaying status of devices attached to the bus allowing enumeration of newly connected. A maximum of 11 devices can be supported by the SoundWire master interface.

This interface has been chosen as relevant for an optional exchange of I²S and I²C in 2.1.14. Thus further detailed explanation follows about functionality based on Pierre-Louis Bossart, 2014 and MIPI Alliance, 2019.

Figure 2.8: Basic frame of Soundwire Protocol

By displaying figure 2.8, a general overview about the SoundWire protocol in basic frame format shall be given. Rows and columns are flipped by 90 degrees. The upper line includes audio data signed as A. In this show-case 16 bit audio data is intended for transmission. Remaining 32-bits are not used. The lower part is control word related data placed in the first column of a SoundWire (SDW) frame.

The numbers listed below describe control word related bits and usage:

1	Ping Request: Owned by slaves to signal requests and status changes
2-4	Opcode: Decision for Read, Ping or Write commands
5-8	Either address for choosing slaves to operate read/write commands or used for stream synchronization/transfer bus master responsibility to another SDW Controller
9-24	Stores Register Address for read/write and slave status of devices 11-4
15-32	Static Synchronization: Bit word used to make synchronization easier for devices
33	PHY Sync: Signals physical limits (high or low) of used version
34-41	Read/write register data or slave status 3-0
42-45	Dynamic Synchronization: Pattern sequence used for slave synchronization operations
46	Parity: Even and Odd parity are used to discover bus clashed
47	NAK: Not acknowledge signals not accepted data read/write operation
48	ACK: Register read/write was successful

SoundWire slaves have a number of registers whereas audio port dependent ones are used to define space and kind of data on the frame others are relevant for clock speed scaling and stopping, frame size, interrupts, device ID etc.

Transmission of SDW data follows the prinicple of Non - Return to Zero Inverted (NRZI) encoding. Switching of the signal corresponds to transmitting a digital one. The constant line level delivers digital zeroes. As the interface transmits data according to Double Data Rate (DDR), both clock edges are used for sampling data. A controller starts framing on a falling edge. Thus, control data is transferred on this clock edge.

At the start-up phase, a consecutive switching by data line sends a sequence of 2048 ones, signalling slaves, that a running protocol is coming up.

To enable functional audio transfer, a slave has to be enumerated (see 3.8.4) and signed to 1 out of 11 possible device numbers. After this process, slaves

can display their status by requesting ping commands. Untreated slaves-status information can lead to dropping them from the bus.

2.1.14 Discussion and Results

11 audio interfaces have been introduced previously and listed in table 2.1, which includes their main properties. Achieving an optimization of Leda implies either lowering power consumption, wiring or area economization. A high valued impact of applicability is given by interfaces market pervasion.

The audio interfaces MLB and A2B are excluded as opportunities as they are developed for automotive industry. A dependency on MOST bus systems is given by those additionally. Bi-directional I²S is nothing others than an expansion (1 extra pin for duplex transmission) of I²S. Until now, Leda (subitem 1.1.1) is not designed to support audio data exchange in both directions as it is meant to be the amplifier for Ganymede's (MEMS piezo speakers). In case of extending the primary use-case, concerning implementation of a microphone, this interface may be an option.

The AC-Link and HDA-Link are both developed by Intel as link frame format's for PC's audio sub-systems. A minimum of 5 pins and application area eliminates any utilization option of this interface. SSI is initially developed as simplex, non-multiplexed differential interface. Compared to this, the currently implemented I²S has the advantage being capable of transmitting a second channel for example, a woofer speaker supporting the MEMS. Further SSI development and "enhancement" for, specifically Motorola DSP's has been done by Freescale introducing ESSI and ESAI. Both are able to transmit up to 32 channels per frame. As there are more wires needed for realization (min. 3) compared to SSI and both are, in turn, used only for special applications and not strong in market penetration they drop out.

Finally, MIPI standards SLIMBus and SoundWire are left as possible I²S and I²C replacements. Both use a 2 wire multi-drop structure capable of wide configuration possibilities in the sense of audio channel and control data transmission, as well as supporting different clock speeds. The interfaces are not strongly present at the market until now which clearly would be a reason for no further treatment. As SoundWire, especially, is a fairly new development there might be an increased market penetration in the near future (TDK announces Microphone with integrated SDW protocol in Jannuary 2020). SoundWire is dealt as the successor of SLIMBus (see Alexander Khazin, 2018) although it has some advantages like seamless clock frequency scaling. Those two standards present the only possible solution of decreasing pin count, as well as the transmission both audio and control data.

As a result, SoundWire interface is chosen to be implemented as a controller port on a Field Programmable Gate Array (FPGA) for further testing reasons. On-chip area is a key part of analysation. The slimmed down version of SDW

is designed in chapter 3 and implementation is shown in chapter 4. Further results concerning SoundWire controller port are discussed in 4.4.

2.2 Audio Data Encoding Techniques

At the moment, audio data is transmitted via an I²S interface to Leda. Thus, information is encoded in two's complement. For implementation of a different encoding technique, data has to be converted back to the origin after transmission for further processing. Data transmitted by the audio interface has been taken into account for encoding consideration along with on-chip communication. There are three different transmission variations. In the next session those are discussed depending on the application.

2.2.1 Self-transition versus Coupling-transition

The impact of capacitive losses on transmission lines are described by two kinds. Self-transitions are related to single wire transmission. Coupling-transitions are treated in respect to parallel wiring having narrow spacing. As the first one is especially important for connections to a chip, the second one treats an on-chip influence. In the following both are further described.

Self-transition: The wire capacitance is responsible for energy losses causes by switching the line on and off again. Power consumption increases by the capacitive losses. External influences are not taken into account for self-transition behaviour.

To get an idea of the capacitive behaviour related to a conductor, the Leda evaluation board has been taken into account. The conductors length was measured by hand in Altium (PCB designer Program), leading to an overall length of 97.65mm. Using Saturn PCB Toolkit (Program for PCB related calculations), one can calculate the conductors capacitance per centimetre. The parameters used for the calculation are conductor width (0.2mm), conductor height (0.325mm) and the frequency (12.288MHz) whereas latter has a minor impact on the capacitance. Conductor height describes the track distance to the ground plane. As result 0.6981pF/cm has been calculated and used for further processing. Additional to the capacitance one can output resistance by the used program leading to a value of 0.3Ω.

Coupling-transition: Figure 2.9 shows capacitances related to a bus having several parallel wires being close together. The capacitances of one to the ground plane is named C_L. The capacitance among the conductors is signed as C_I.

Considering coupling transitions, one assumes that switching ON and OFF a wire is additionally affecting wires besides the capacitive effect (C_I). Depending on the power level state of wires related to change of one conductor, interacting capacitances are charged or discharged. The encoding technique OEFNSC gives insight in this process and how this can be influenced. Generally spoken, coupling transitions are triggered by self transitions.

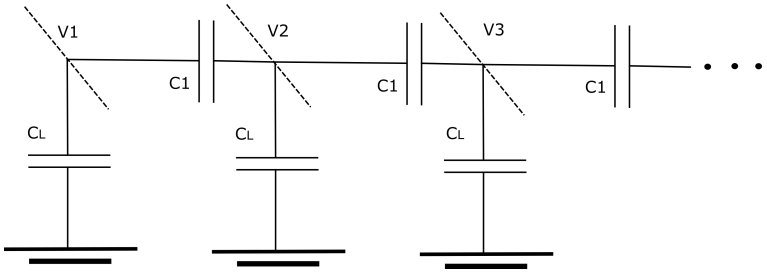


Figure 2.9: Capacitances relevant using Bus systems

2.2.2 Potential Field of Application and Related Encoding Techniques

Overall, there are three meaningful areas being considered for applying encoding techniques.

On the left side of figure 2.10, the I²S interface is characterized. Herein, data is transmitted serial. Switching signals at this point are related to self-transitions (take a look at previous section 2.2.1).

The other is based on parallel wiring since these are on-chip connections. Linking of the I²S interface and signal processing unit is again affected by self-transition. Regarding conductors, after ADC and output of DSP, coupling-transitions may have an influence due to a longer cabling distance.

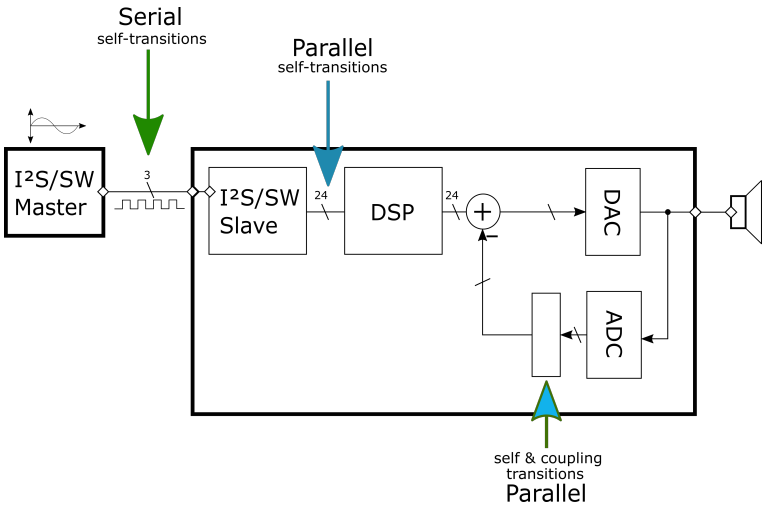


Figure 2.10: Field of different encoding applications

2.2.3 Investigation of Data Encoding Techniques

For both, parallel and serial, three encoding variations have been tested. To compare influences, two of them have been taken into account twice. Regarding serial part, a so called pattern encoding has been tested based on the thoughts of my supervisor and myself. The last one (OEFNSC), treats coupling transitions as well, whereas all of the others just affected self-transitions. Tests were done by using a sine signal at 1kHz without dc offset. An period of one second, sampling the signal by 48000Hz and quantization done with 16 and 32 bit, has been considered.

Serial:

Serial connections are used if lower wiring harness is important to reduce area consumption as example. For data transport, time-division multiplexing is applied. Thus, the clock speed needs to be increased if same data rate has to be transmitted.

Binary Inversion: For binary inversion, one takes all logical values and switches high ones to low and vice versa. If data is equally distributed, the outcome is resulting in an overshoot. The same amount of switches will occur. Additional logical circuitry is needed for inversion. Inverting binary words like this makes no sense for reducing switches at serial transmission. One possible application is reduction of ones or zeroes. But overall number of switches will stay the same. Thus, an implementation is not meaningful at all but for reasons of comparison and impact of encoding-expenditure, binary inversion for serial connection, is listed.

Transition Inversion: This encoding describes a preliminary stage of later pattern encoding. Considering a binary word like 01010101, this is the most switching active number one can imagine for an 8-bit word. If the switches counter exceeds a certain limit we may set to 4 (it's major then the half of maximum 7 switches per word), the encoding has to be applied. Therefore, all pairs of 01 will be switched into 00 and all pairs of 10 into 11. Previous assumed binary word results in 00000000 having no switches at all.

Pattern Encoding: Patterns are related to bit sets a binary word consists of. Assuming a cipher 011000101, five toggles are needed for a transmission. Applying a pattern change like the following, 0110 remains constant. 0101 changes to 0000. The new pattern looks now like 01100000. A switching reduction from 5 to 2 occurred. There are several buttons to turn on for optimization. Pattern size is one important variable and another is the decision threshold of switches since an encoding has to be applied. Another consideration is a Volume depending

Serial:				
	org.	-0000111100001111	inv.	-1111000011110000
Parallel:				
	org.	-0101	inv.	-1010
		-0101		-1010
		-0101		-1010
		-0101		-1010

decision threshold. Thinking about 16-bit word, if the Volume is set to -20 dBFS, at least the first 3 bit stays the same except the negation bit.

Parallel:

Implementation of parallel wiring is related to the on-chip connections in our case (see 2.10). As described before (2.2.1) for parallel transitions both effects ()self- and coupling-transitions) can be relevant. Binary and transition-inversion are related to self-transitions. The so called Odd Even Full Normal Self Coupling (OEFNSC) has been suggested for minimization of coupling transitions.

Binary Inversion: Binary inversion for parallel wiring is based on the same principle as serial with the difference being that each conductor transmits only bits related to one position of binary numbers. Thus, an implementation of encoding, including an dependency on previous word, is possible.

Previously bit words of size 4 in serial and parallel manners are displayed. As one can see for serial alternative, only two bits on both ends are able to influence switching. In contrast parallel transmission opens up the chance of reducing toggle rate significantly.

Transition Inversion: Applying transition inversion encoding on parallel wires can be improved, as well as binary inversion, by adding a dependency on previously transmitted binary-word. Nevertheless, one has to keep the size of patterns and related coding effort in mind.

OEFNSC: The power of Odd Even Full Normal Self Coupling (compare Chennakesavulu, .Jayachandra Prasad, and Sumalatha, 2018 and Jafarzadeh et al., 2014) lies in its potential of reducing self and coupling-transitions. Dynamic power is then calculated in dependency of a factor α_S and α_C. C_C is the capacitance related to the interconnections, C_S the capacitive load due to the substrate and C_L, the load capacitance.

$$P_{dyn} = (\alpha_S * (C_S + CL) + \alpha_C * C_C) * \frac{{V_{DD}}^2 * F_{CLK}}{2} \tag{2.1}$$

The coupling activity α_C is calculated in relation to four different types of switching. Coupling transitions are classified as Type-I, Type-II, Type-III, and Type-IV in two wire models (take a look at table 2.2). Type-I coupling transitions are happening when one of the interconnect switches and other interconnect remains same. Type-II coupling transitions are happening when two interconnects are switched simultaneously in opposite. Type-III coupling transitions are happening when two interconnects are switched simultaneously. Type-IV coupling transitions are happening when two interconnects are not switched.

Table 2.2: Different Types of Coupling

Time	Type 1	Type 2	Type 3	Type 4
t-1	00,11,00,11,01,10,01,10	01,10	00,11	00,11,01,10
t	10,01,01,10,00,11,11,00	10,01	11,00	00,11,01,10

2.2.4 Encoding Results & Conclusion

Figures 2.11 (a-f) show computed results regarding switching activity α of different encoding techniques. The graphics can be split into a left column (a,c,e) related to serial encodings and a right one (b,d,f) related to parallel ones. Rows distinguish different types of applied audio data. Primary (a,b) a noise signal has been tested to get an independent insight on efficiency. The second row (c,d) shows a sine signal including added noise. This should be more accurate to the field of application in the end. In the last row (e,f) a sine sweep has been applied without any noise. Each figure consists of three different encoding techniques either BINV, TINV, Pattern or BINV, TINV, OENFSC which in turn are split into using full scale dynamic range or -20 dBFS. This is marked in the legend according to V = 30 and V = 3 based on applied output voltage ,where 30V will be used rarely and 3V is the more common amplitude. Each of these shows then the original toggle rate, the toggle rate after encoding and the switching after encoding with included additional switches due to the encoding technique. For all examples, additional lines have been assumed transmitting information if encoding is applied rather than reducing bits of resolution. Calculated numbers used for those graphics can all be found in the appendix B.

As already mentioned, at the beginning of previous section, binary inversion for serial transmission (figure 2.11 a) has a negative influence on toggling rate. Alpha stays nearly the same when encoding is applied and increases for inclusion of coding related switching. ”Nearly” same due to lining up words, each crossing from one bit-word to the next could cause a difference of one toggle. Maximum 48000 switches (equally the number of samples) could add up

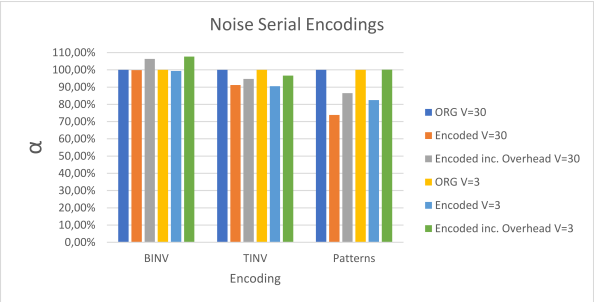
or lower the overall amount by this. Inverting bits can, in addition, cause a little variation regarding the crossover. Level control has no influence on that just like different signals. Compared to this, transition inversion shows an impact although it's little. Full scale audio switching can be reduced by 5.26%, whereas it's only 3.36% for standard application ratio. One has to keep relation to a pure noise signal in mind. Audio variation effects are discussed later. Pattern encoding performs best at this without taking a lower level into account for the latter encoding related switching amount relativises results to equal.

Results for parallel encodings according a noise signal is shown in picture 2.11 b. Here, switching reduction by binary inversion has a significant influence. Full scale noise can be reduced by almost 20%. For a lower noise level toggling is decreased further to 71.8%. Including the additional expenditure for determining the encoding, this ends up in a reduction by 13.62% to 22%. Considering 0dBFS transition inversion performs similar to the serial one except the lower level allows further reduction down to 84.9% (serial transition inversion incl. overhead adjusts at 96.64%). Where serial considerations are extended by a pattern encoding, parallel ones treat OEFNSC as addition. Toggling reductions by 12.49% for full drive and 16.65% for a scaled level (including all encoding switches) are possible.

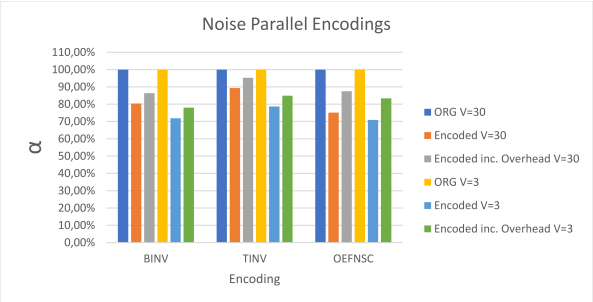
The previously discussed has all been related to a pure noise signal. Extending this by a sine-wave (figures 2.11 c and d) to simulate a more accurate transmission of audio music or speech signals (noise -20dB in relation to the sine-wave), the following findings are significant. Binary, as well as transition-inversion, for serial connection is nullified. Pattern encoding shows little optimization by a maximum of 6.06% reduced switching. Looking at the parallel coding first two can be neglected (2% to 3% improvement) and the OEFNSC overshoots original toggles.

A sweep signal has been tested aswell. Depicted in figures 2.11 e and f, all algorithms (with one exception) are either increasing basis switching amount or equalling them. Single exception is full-scale serial pattern encoding showing an improvement of about 10% reduced switching activity.

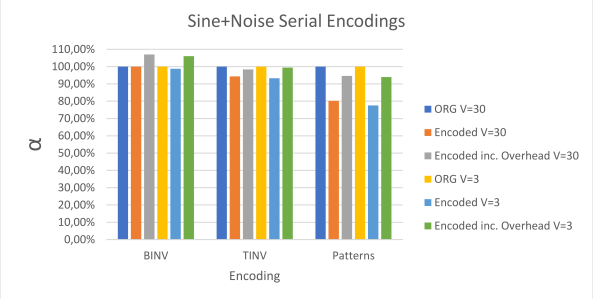
Overall, the investigation of those encoding algorithms didn't show a common thread throughout tested signals. Whereas noise related improvement can be shown especially for parallel transmission lines inclusion of sine waves denied primary refinements. According to this, no special audio data coding technique is being suggested for integration into Leda.



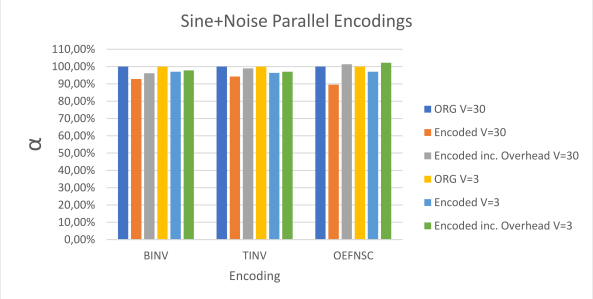
(a) Noise Serial Encodings



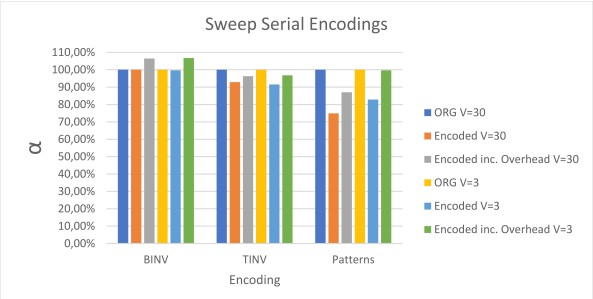
(b) Noise Parallel Encodings



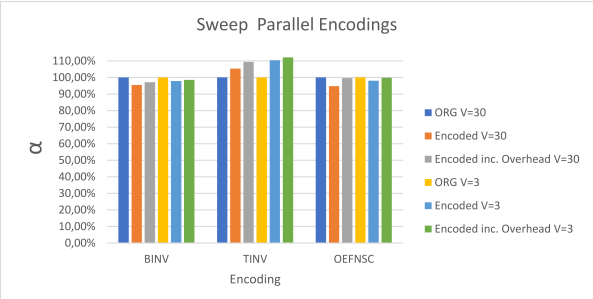
(c) Sine+Noise Serial Encodings



(d) Sine+Noise Parallel Encodings



(e) Sweep Serial Encodings



(f) Sweep Parallel Encodings

Figure 2.11: Results of Encoding Algorithms

3 Design of the SoundWire Controller Interface

Attention in this chapter is put on general setup and structure 3.4, the corresponding components (3.1 & 3.2.2), integration on the audio-chip and wanted/needed SoundWire interface modifications (3.5 & 3.7). As there is no audio component including a SoundWire controller interface on the market that allows physical access (e.g. Intel 300 Chipset Intel, 2019) and configuration for own setups (rarely companies such as Intel give insight into their data sheets), the decision was made to implement the controller protocol on an FPGA and operate a slave.

Slaves can be found more easily and whilst authoring this theses new were presented (TDK announced a microphone TDK, 2020). Cirrus Logic developed an Audio Codec called CS42L42 (see CirrusLogic, 2019) allowing it to be controlled via SoundWire. It's decided to use Class-D Amplifier MAX98374 (MaximIntegrated, 2018), produced by Maxim Integrated, as it's in the scope of a potential use-case for USound as well as it's cheaper containing an Evaluation kit.

Detailed implementation and related upcoming problems, including further approaches, are discussed in the next chapter, chapter 4.

Due to the previously mentioned lack of SoundWire controller interfaces on the market (no fast switchover in prospect at the moment), possible applications other than using it for LEDA setup and control were considered. Ganymede is a speaker able to perform well in in-ear implementations. When it comes to free field acoustic irradiation, there is a known lack of low frequencies. Audio wearables such as glasses without transmission via bone contact will need support by a woofer. This is a point for the novel interface to step in action. Shown in picture 3.1, LEDA is still controlled by an I²C connection and audio transmission is done using I²S. The lower part depicts a woofer amplifier, in our case a MAX98374, being operated by the two wired SoundWire connection.

Whole implementation is done due to testing reasons. Thus, direct integration into the current programmed LEDA is avoided. Interface placement on the FPGA is explained in section 3.4. Following sections contain explanations for all hardware components needed in order to accomplish the previous defined.

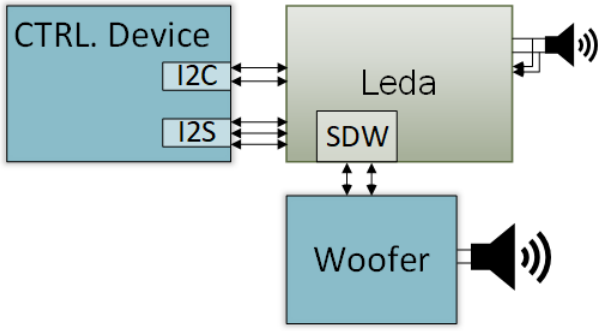


Figure 3.1: Future on-chip SoundWire application alternative

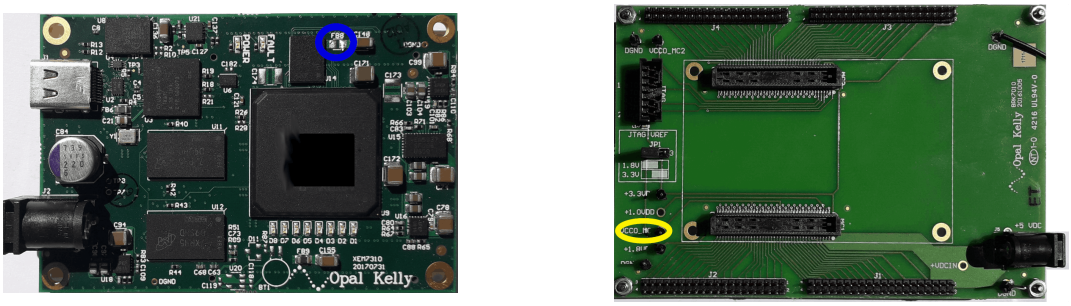
3.1 XEM7310 - Board

The XEM7310 is a compact USB 3.0 (SuperSpeed) FPGA integration module featuring the Xilinx Artix-7 FPGA, 8 Gib (256 M x 32-bit) DDR3 SDRAM, two 128 Mib SPI Flash devices, high-efficiency switching power supplies, and two high-density 0.8-mm expansion connectors. The USB 3.0 SuperSpeed interface provides fast configuration downloads and PC-FPGA communication, as well as easy access with our popular FrontPanel application and Software Development Kit (SDK) (compare Sanchez, 2018).

There are different board versions available depending on features to be unlocked. Also, the mounted clock is changing due to this. XEM7310-A200 is the actual bought version including a low-jitter 200 MHz crystal oscillator attached to the FPGA. Further upgrades to predecessors are increased number of Clock Management Tile (CMT)'s, slices, RAM and MULT/DSP.

- CMT's is the number of different clocks to be configured and used at the same time
- A slice defines four 6-LUT including 8 D-FF's
- The MULT/DSP is related to the calculation units in this case there are 740 DSP48E1 units
- Ram is the Block Ram used for storing large amount of FPGA data (13,140Kib)

For physical input and output access a breakout board is needed. MB2 (Main Board 2), designed for LEDA evaluation is not usable for this due to its special design for the chip. Thus, basic board BRK7010 from Opal Kelly has been used (see 3.1.2).



(a) XEM7310

(b) BRK7010

Figure 3.2: FPGA setup

3.1.1 ARTIX-7 - FPGA

ARTIX-7 FPGA, mounted on the XEM7310, was picked according to requirements and specification of LEDA. As it's common for ASIC development, the capabilities of such an FPGA are chosen much higher than actually needed to avoid limitations in the development phase. Thus, additional integration of the SoundWire interface in the current LEDA setup experiences no boundaries. Special modifications are discussed later in 3.7. Most important is, for general interface functionality, the availability of DDR flip-flops has to be given for output and for input handling. Those are able to provide data on both clock edges (see Output Double Datarate Register (ODDR)-register in 4.2.5).

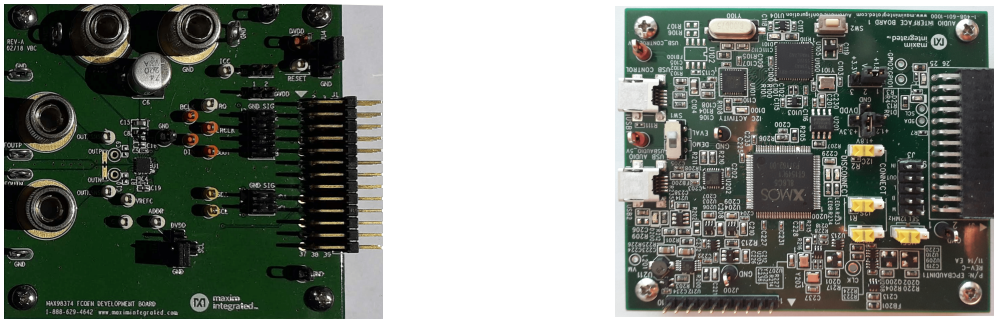
3.1.2 BRK7010

The two layer breakout board BRK7010 (3.2 b) is designed for mounting either the XEM7010 or XEM7310 on it. Two high density connectors are provided for this. Each of those are split up into two areas of double row headers spaced by 2 millimetres. +5VDC powering can be done via this board or the XEM7310. Additionally an JTAG interface is integrated.

Using this board, one is able to handle all 80 FPGA IO pins. This helped a lot in debugging problems by usage of a SALEAE digital analyser.

3.2 MAX98374 Evaluation KIT

As slave device for testing the SoundWire interface, the MAX98374 class-D amplifier was chosen. The evaluation kit consists of the MAX98734 (3.2.2) board and a Audio interface (AUDINT 1 3.2.1) board. Both can be interlocked via an 3x13 pins connector. Additionally an evaluation software is delivered. MAX98374 can be packaged in either Wafer Level Package (WLP) or a Quad Flat No Leads Package (QFN).



(a) MAX98374

(b) AUDINT1

Figure 3.3: MAX98374 Evaluation Kit

3.2.1 AUDINT 1

The audio interface board AUDINT 1 can be used to configure the MAX98374. I²C and I²S interfaces are provided. Those can be controlled by a PC via the USB connections to bring up audio and controlling data. Having the evaluation software installed, one is easily able to configure the amp. Master clock sources are being provided by this board. Different voltages (1.2 to 3.3) can be selected for powering the evaluation board. There are two modes available. Demo mode is used if no PC is available/wanted. Using customized firmware, it's possible to operate via one USB connector. For every other usage, the EVAL mode is applied, controlled by a pc including relevant software. Regarding SoundWire interface, there is no usage for having the AUDINT 1 board connected as there is no controller interface present. Thus it's possible to drive the board Class-D amplifier in stand-alone mode as it was done for testing the SoundWire protocol.

3.2.2 Class-D Amplifier MAX98374

The MAX98374 is a high-efficiency, mono Class D speaker amplifier featuring dynamic headroom tracking (DHT) and brownout protection. As the power-supply voltage varies due to sudden transients and declining battery life, DHT automatically optimizes the headroom available to the Class D amplifier to maintain a consistent listening experience. A wide 5.5V to 16V supply range allows the device to exceed 16W into an 8Ω load. The flexible digital interface supports either the MIPI SoundWire compatible interface for audio and control data, or the PCM interface for audio data and a standard I²C interface for control data. The PCM interface supports I²S, left-justified, and TDM audio data formats at 16-, 32-, 44.1-, 48-, 88.2-, and 96kHz sample rates with 16-, 24-, and 32-bit data. In TDM mode, the device can support up to 16 channels of audio data. A unique clocking structure eliminates the need for an external master clock for PCM communication, which reduces pin count and simplifies board layout. Active emissions limiting (AEL), spread spectrum modulation (SSM), and edge rate control minimize EMI and eliminate the need for the output

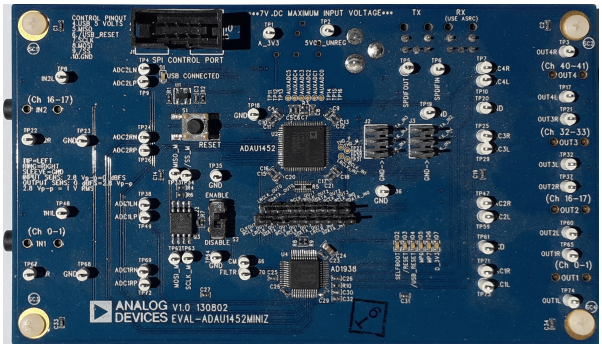


Figure 3.4: ADAU1452

filtering found in traditional Class D devices. Thermal foldback protection ensures robust behavior when the thermal limits of the device are reached. When enabled, it automatically reduces the output power when the temperature exceeds a user specified threshold.

This allows for uninterrupted music playback even at high ambient temperatures. Traditional thermal protection is also available in addition to robust over-current protection. A flexible brownout-detection engine (BDE) can be programmed to initiate various gain reductions, signal limiting, and clip functions based on supply voltage status. Threshold, hysteresis, and attack/release rates are programmable. The device is available in a 0.4mm pitch, 25-bump waferlevel package (WLP), or in a 0.4mm pitch, 22-pin FCQFN package. The device operates over the extended -40°C to +85°C temperature range (compare MaximIntegrated, 2018).

3.3 ADAU1452

ADAU1452 is an automotive-qualified audio processor that far exceeds the digital signal processing capabilities of earlier SigmaDSP devices. Its restructured hardware architecture is optimized for efficient audio processing (see AnalogDevices, 2014).

ADAU1452 is not a must have for this project. It has been used after functional testing SoundWire protocol to bring up I²S audio data to the FPGA further to the novel interface and playback music over a speaker attached to the MAX98374. A sine signal can be sent as well as connecting a mobile phone with 3.5mm jack cable to it and provide any kind of audio. ADAU setup was done via USB and SigmaStudio software. A pre setup configured by a employee of USound was available just for this kind of application. In appendix C SigmaStudio setup is pictured. Right side of this shows enable buttons to changing number of transmitted channels. Audio jack input to the board is shown via the path called FreqGen and on the upper paths different kinds of sine/noise sweep signals can be adjusted and added up. For each path, a volume control is available.

3.4 SoundWire Module Integration (FPGA)

FPGA structure is mainly split into two parts. One concerning digital part of Leda and the other responsible for control. It was essential to not interfere the development of Leda. The controlling unit of the digital chip on FPGA was found to be the best placement.

Before going into details, an overview of current structure is given. Figure 3.5 depicts the insight into upcoming description.

FrontPanel SDK is the physical frontend connecting the board via USB to the PC. FrontPanel SDK allows the user to either download new FPGA configuration or perform operations like register read write on the modules by a Python script.

Mux BM Bus facilitates switching between digital Leda configuration or data transfer to the control part of it.

Leda1 dig_testmux is the digital part of Leda consisting in turn of I²S & I²C interfaces, switched by the MUX, and modules like filters, upsamling, DCDC-control and ADC-control.

PWM transmits information to the power stage whether to increase or decrease voltage according to the width of the pulse.

Leda1_ctrl is the controlling unit for Leda. Thus, both interfaces I²S and I²C are present in addition to a realtime debug interface, a stimulus generator for sine (and other specific signal forms) generation and an additional control block including enable, clock GPIO's and interrupt request handling. The modules highlighted in red are the newly added sources, that includes the SoundWire interface and a stimulus generator for it being a copy of the already existing one. Modules within the Leda1_ctrl are connected via a Advanced Peripheral Bus (APB) interface similar as in the digital Leda.

3.5 Common SoundWire Design Structure

This section is related to the general SoundWire-Controller interface IP by Arasan (see Arasan, 2020). Other IP providers like cadence have little differences, but overall it's the same. Due to some restrictions and lowering expenditure thus saving power, several modifications have been done and are discussed in section 3.7. The main structure of a SoundWire controller interface consists of three

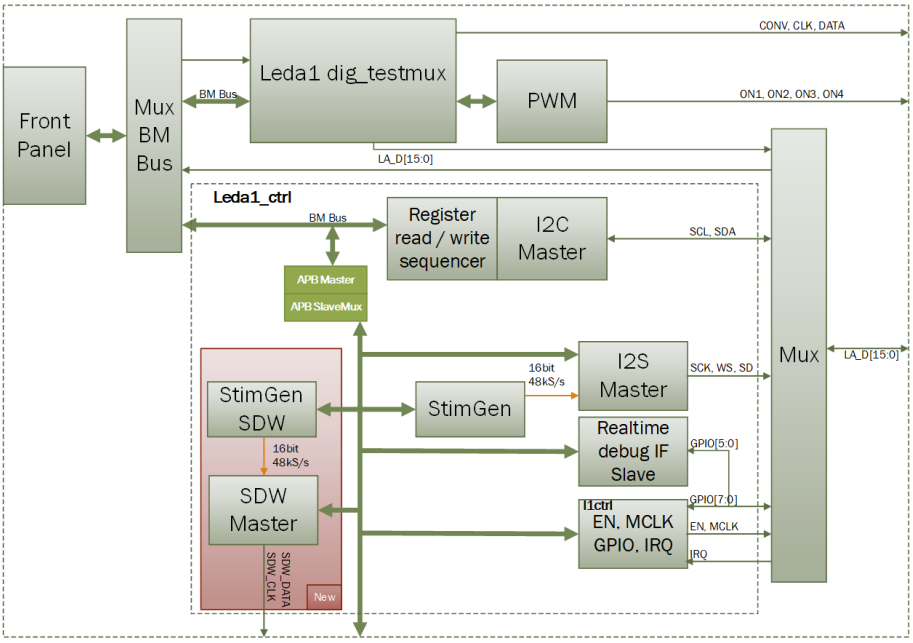


Figure 3.5: Setup for Leda evaluation on the FPGA

subdivisions. These are: Transport, Framer and PHY. First one is responsible for data handling to and from the module. The framer does everything about mapping and de-mapping data, on transmission line to be transmitted or received. PHY is related to physical interface part. Things like encoding and bus clash detection are placed here.

3.5.1 Transport

Concerning the SoundWire module different ports for data, depending on field of application, are included here. Especially, all audio data is provided in both directions by those. In certain circumstances, data can be cached in a FIFO Register in-between the ports and connected equipment. Furthermore, test signals can be generated either static or Pseudo Random Binary Sequence (PRBS) ones. Another important part of the Transport area is the ”Control Port”. The whole setup and configuration of controller module is done in here. Operations like reset or frame size changing is controlled by this. For communication an APB slave is implemented writing to the internal and command registers. Bulk Register Access (BRA) is used for a fast initial configuration of a SoundWire slave providing more than one 8-bit register data (up to 511) per frame. Interrupts for error handling and informing the main component of special states is captured by the control port as well.

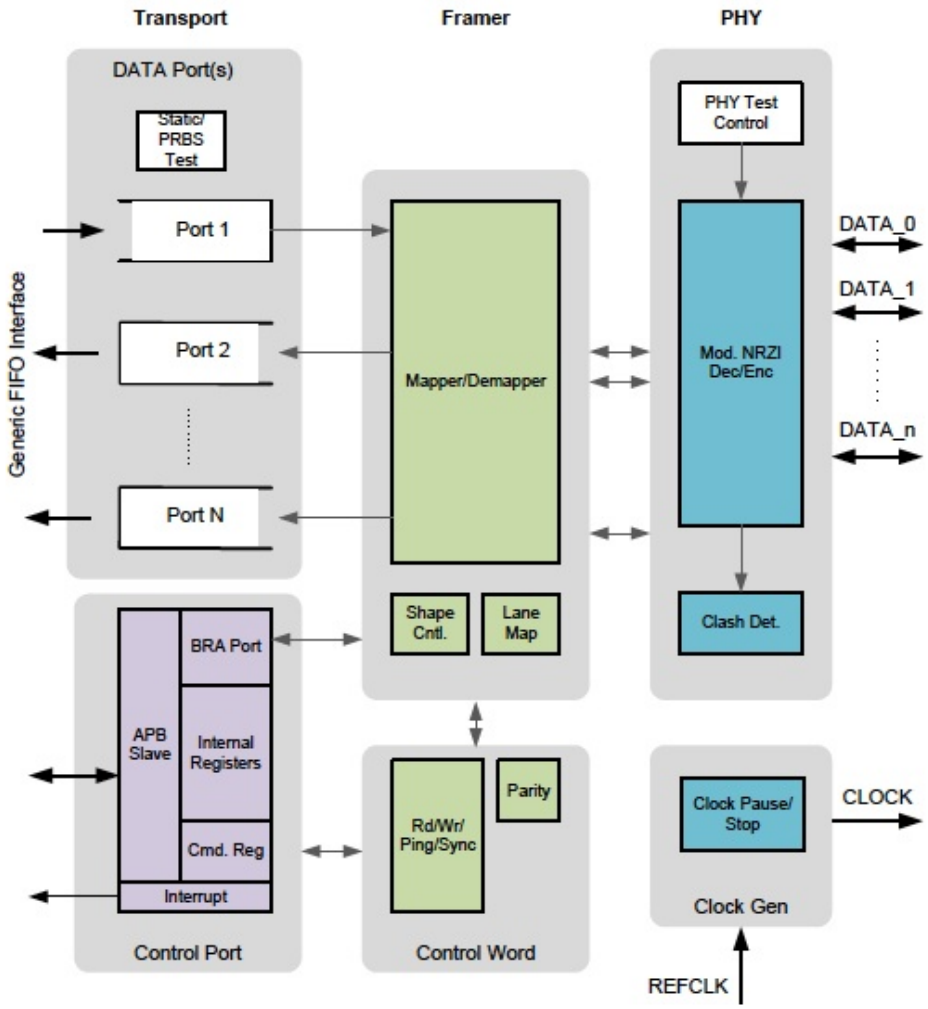


Figure 3.6: Arasan SoundWire Controller Block Diagramm (see Arasan, 2020)

3.5.2 Framer

The Framer is again split into two main sections. Control word is responsible for all 48-bit related to the first column of a frame (BRA excluded). Particularly, the operation sequences read, ping and write handling is done in here, including the addresses and dynamic synchronization bits generation. Calculation of parity is another task of the controlling interface, helping to detect bus clash errors in PHY whether data can be stored or should be discarded. Core of the whole protocol is a Mapper/Demapper. Each bit, no matter if it's audio or control data, is put to transmission by this module. Furthermore, all received data has to be assigned to its place of destination. Shape control is necessary after the primary setup (always 48x2 frame) to create other desired frame sizes. Last part of the Framer is a so called "Lane Map". The interface is capable of transmitting audio data of more than one device. This means more than one data port is feeding/receiving data. Positioning of the data on a frame is done via this lane mapper.

3.5.3 PHY

The third block of this overview is the physical connection with other SoundWire devices. Modified NRZI encoding and decoding belongs to this. Testing features for PHY and Bus Clash detection related to the parity calculation are further blocks in here. Clock handling at start-up or pausing while the slave is working in stand-alone mode or being switched off is placed in PHY as well. If the reference clock signal is too fast or too slow, the clock generation can be done here and even slowing or speeding up the clock is possible.

3.6 Measurement and Supply Hardware

3.6.1 SALEAE - Digital Analyser

SALEAE develops signal analysers specially for digital signals. Still, analogue measurement is possible but for that a much lower sampling rate available being not capable to show 12.288MHz signals. Thus, analogue testing has been done using a Gwinstek GDS-2204A. A Logic Pro 8 device has been provided by the company for development phase including two test lead sets, four lines each, capable testing 100MHz digital signals.

3.6.2 Oszilloscope

For analogue measuring the signals, a Gwinstek GDS-2204A was available. This oscilloscope is capable of displaying digital signals by using a plug-in

card. Unwieldiness and a poor overview using several digital inputs lets the previously mentioned SALEAE come into considerations.

3.6.3 Power Supply

HMC 8043, developed by Rhode&Schwarz, has been used as a power supply due to its capability of outputting three different voltages, whereas at least two (1.8V and 10V) were needed.

3.7 Modifications

Several modifications were implemented compared to previous controller structure. This section discusses those as well as hardware modifications such as ferrite bead de-soldering on XEM7310. All changes here were known and considered before actual implementation began. Some special effects are coming up in the development phase. Those are explained in the implementation chapter.

3.7.1 XEM7310

Necessary for a functional SoundWire implementation is providing 1.8V for signal transmission. The XEM7310 allows setting different I/O voltages to support various standards. Installed on the board are ferrite beads to both VCCO banks attaching 3.3V supply. Thus, de-soldering ferrites is inevitable to prevent destructing devices. Two ferrite beads (FB8 & FB9) are placed on the board. FB8, shown as blue ring in figure 3.2a is responsible for SoundWire development. The power of 1.8V is supplied then by the additional onboard connection shown as yellow in 3.2b.

3.7.2 MAX98374

When connecting MAX98374 to a SoundWire compatible controller interface, some hardware configuration must be done. Internal behaviour of the amp differs as well due to fact register handling is done by the same interface.

Hardware Modification

Referring to evaluation kit MAX98374 development board is used in stand-alone setup without the AUDINT 1 board being connected.

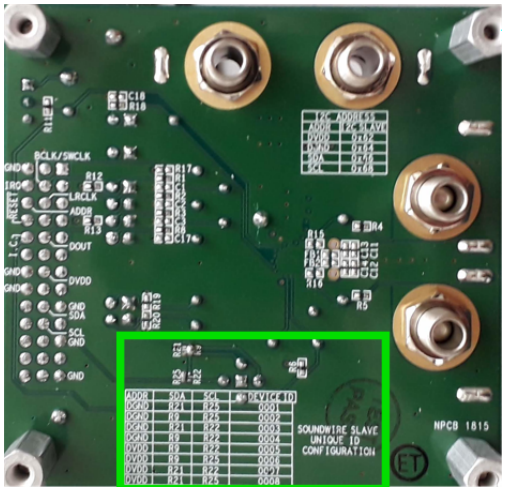


Figure 3.7: Back Side of MAX98374

Wiring: FPGA output is conjuncted by two wires to the Digital Audio Input (DAI) designated J3 header. Pin 2 connects clock whereas 6 is the data line. To avoid floating ground potential one of the pins 1,3,5,7 has to be additionally applied to the XEM board. Furthermore, wires for powering the board and supply for the speaker were put to DVDD (1.8V) and PVDD (10V). Digital amplifier outputs are FOUTP and FOUTN. A 8Ω dynamic speaker is plugged in capable processing 16W according to the specification.

Soldering: The possibility to connect more than one amplifier, of the same kind, to a SoundWire (or I²C) interface demands a distinct characterisation. Integration of a slave device unique ID enables this. On MAX98374, jumper JU6 is responsible for that. 0Ω resistors have to be placed on the board according to table 7 in the specification (marked on the board as shown in figure 3.7) depending on desired device ID. Upon delivery, no pre-configuration is done and all connections related on unique ID are openly circuited. In this case, only one slave is put together, thus placing R21 and R25 ,meaning short-circuit them, is sufficient (green rectangle in figure 3.7).

Software Modification

Software modification for a correct FPGA setup was necessary as the registers of the SoundWire interface are now included. The register placement has to be described for enabling the APB bus access.

Python Leda Control Software: For debugging Leda, a large amount of python software has been implemented. Using this, a direct ”live” connection via the USB Front Panel SDK to the FPGA further the APB bus in the Leda Control part is enabled. Software for the novel interface is extended as following.

Instantiation of Modules: Both new modules (StimGEN & SDW Controller)in the Leda control part got their own address assigned to be independently accessible.

Listing 3.1: extension

```
...
  SDW_ADDR         = 0x2500
  STIMGENSDW_ADDR = 0x2600
  def __init__(self):
  ...
     self.sdw =fpga_register.FPGA_Reg_sdw()
     self.sdw.applyAddrOffset(self.SDW_ADDR)
     self.stimgensdw =fpga_register.FPGA_Reg_stimgen()
     self.stimgensdw.applyAddrOffset(self.STIMGENSDW_ADDR)
     return
 def getRegName(self, addr):
  ...
  reg = self.sdw.getRegName(addr)
  if (reg != "-"):
    sreturn "SDW_" + reg
  reg = self.stimgensdw.getRegName(addr)
  if (reg != "-"):
    return "STIMGENSDW_" + reg
  return "unkown"
```

Testcase Config. extension: For accessing implemented configuration files over the Graphical User Interface (GUI) adding the SDWMaster paraphrase.

Test configuration: For setting the registers test configuration register files are being created. Several, such as the listed below, were added to setup the devices.

Listing 3.2: Registerconfig

```
    def test_SDWMasterWriteooGLOBEN(self, param_string="-1"):
"""[No Parameter], Write Global Enable"""
reg_sdw = self.fpga.reg.sdw
self.fpga.sim.info(" Write  Global Enable")
self.fpga.bm_write(reg_sdw.SDW_SLVDEV_ADDR,0x00000001,1)
self.fpga.bm_write(reg_sdw.SDW_SLVLOW_ADDR,0x000000ff,1)
self.fpga.bm_write(reg_sdw.SDW_SLVHIGH_ADDR,0x00000020,1)
self.fpga.bm_write(reg_sdw.SDW_SLV_ADDR,0x00000001,1)
self.fpga.bm_write(reg_sdw.SDW_SLV_OP_ADDR,0x00000002,1)
```

All processes behind python GUI were already implemented and didn't require further action.

Register Modification: In contrast to later discussed registers of the SoundWire controller, this part concentrates on slave register values to be modified. MAX98374 registers are separated in ones belonging to SoundWire slave interface and others to general controlling. In turn, SoundWire registers are split up in audio related Data Port (DP) and essential ones Slave Control Port (SCP) for start-up and functionality. Be aware of different nomenclature of hexadecimal numbers. In documentations, often "0x" as separator before the actual number is used. For System Verilog primary number defines bit size followed by an "'h" and then the actual number is set (0x0001 vs. 4'h0001).

SoundWire SCP:
To establish a connection and configure any of the registers on MAX98374 slave, the very first change is due to the enumeration phase classifying the slave by a Device Number 1 - 11. Register 0x046 with a POR 0x00 will be set to 0x01. All other SCP related values are not varied by the controller (in the case of this

thesis) but some give behavioural information about the slave like the 0x0041 interrupt register.

SoundWire DP:
The data port registers shall be setup before enabling audio transmission and playback. If the on-chip signal generator is used this can be skipped. DP1 and DP1 - Bank 0 registers need to be configured such that the slave knows the position of audio data in a frame. POR value 0x20 is correctly set for 0x0102 register specifying direction of data transmission and kind of transportation. Register 0x0103 needs to be set to 0x0F for 16-bit audio data. Modifying 0x0120 to 0x01 will start data receiving for the amp. If more than one sample is being transmitted within one frame, the incremental can be determined by the sample interval 0x0122 & 0x0123 in the list. This is not used in the setup, thus it's configured to 0x5F repeating after one frame (96-bit). Registers 0x0124 - 0x0126 are set to 0x03, 0x00, 0x11.

General Control Register Map:
For audio playback following setup has to be configured:

0x20FF	Global Enable	=>	0x01
0x2043	Speaker Enable	=>	0x01
0x2025	Interface Mode	=>	0x03
0x2028	Speaker Path Sample Rate	=>	0x80
0x203D	Speaker Volume	=>	0x6F
0x2040	Tone Generator	=>	0x03

The last register is only set if signal generation by the tone generator is wanted instead of transmitting it via the interface additional.

3.8 SoundWire-Controller Design

Inspired by the Arasan block design, the result for implemented controller is shown in figure 3.11. Some indispensable changes had to be made especially when referring to clocking.

3.8.1 Clocking

To establish a communication channel between FPGA SoundWire controller and the MAX98374 a clock signal of 12.288MHz has to be provided. Leda is using a 41.472MHz clock. As there is no possibility to generate an integer divided SoundWire clock signal, the clock generation was done with an extra unit of available clock management tile on the FPGA. If this system is being integrated in the audio chip, either a change of the used clock is required or an additional Phase Locked Loop (PLL) has to be implemented. Dividing the Leda system

clock by 27 delivers an 1.536MHz clock. Using a PLL and increasing this by factor 8, the necessary clock can be produced.

The clock module was generated by the Xilinx Clocking Wizard (6.0). A 100MHz input is used to acquire desired 12.288MHz. Thus, the clock is depicted in a separated block in the diagram.

3.8.2 Transport

Modifications of modules related to the Transport block were unavoidable as a result of clocking issues, in respect to arasan design, discussed in section 3.5 before. Data provided by the system has to be safely handed over using Clock Domain Crossing (CDC) to avoid meta-stability conditions. Referring to figure 3.11 coloured in white, all data in system clock domain is processed. The orange blocks are driven by the 12.288MHz SoundWire clock.

Audio data is provided at the Data Port. An AXI streaming interface is responsible for data requesting and delivery. Three connections namely update, data, valid are necessary. The update line is requesting for new set of data. Confirming new audio data is done by the valid connector. Third is the conjunction of data to be implemented in desired sizing according the audio data. AXI streaming interface is connected to a First In First Out (FIFO). As soon as Data is taken from the FIFO new data is being requested by the AXI interface. The idea behind this is collecting 2-3 samples and transmitting them in a package per frame at a lower clock speed. The actual implemented design is not aware of clock changes yet. Thus a FIFO is not crucial. Important for later considerations and the conclusion of this thesis is the implementation effort of such a FIFO (chapter 4.4).

Second communication port is the APB consisting of 10 lines. Already implemented for Leda, only little name and register changing had to be done for further module connection. Via this, all register read and write instructions are processed. The main difference of overall connections in respect to the specification is omitting a PSTRB (Peripheral Strobe = enabling sparse data transfer on a write data bus) line.
An in-depth description of both interfaces can be found in Xilinx, 2011 and ARM, 2004.

Register: SoundWire registers are structured in 8bits. Specified in MIPI Alliance, 2019, a placement of them should be done according to this. MAX98374 slave applied the address ranges in that way. As following, a controller interface is being implemented for special application and integration into an already existing system and the common register map is obsolete.
In Appendix E, the whole registry design is shown. Included in the list are essential ones as well as not implemented ones, but reserved for later consideration (e.g. all interrupt requests can be neglected). The list contains Addresses

, Register Names, Bit Names, Access, HW, HW-trigger, Privilege, MSB, LSB, Reset and a Description:

- Address — This is the value for register entries to be stored. The range is given by the implementation specification of Leda. Each module has a possible address range of 8'h00 to 8'hFF. For a higher address range a further instance has to be made.
- Register Name — 8-bit register names
- Bit Name — The name of available sub grouping down to one bit in each register
- Access — Information about possible methods of accessing the bits by the APB bus. Either read -r , write -w or both rw is possible
- HW — Described is the HW register access. data = can be modified by hardware, cfg = hardware modification not possible, const = constant value provided by HW, pw1 and pwo are used for single bit fields creating a pulse if a one or zero are written to that bit
- HW-Trigger — Triggers signalling a r, w, or rw access by APB
- Privilege — Privilege modes are used to avoid unwanted access to critical registers (no relevance here)
- MSB — Is the Most Significant Bit of the bit subgroup
- LSB — Is the Least Significant Bit of the bit subgroup
- Reset — The value of the register after resetting the system
- Description — Information about each register entry

Special registers are Addresses 8'hD0 - 8'hD5. Those are used to read or write bit fields of MAX98374.

3.8.3 Framer

The Framer is responsible for a correct frame shape and mapping the control data on the frame as well as audio data. Implementation does not necessarily need any other frame shape than a required 48x2 form for the start-up thus no other shapes are made for configuration. Further modules and functions placed here are two Finite State Machine (FSM)'s (one for command handling and one for the start up enumeration phase), a PRBS pattern creation for dynamic synchronisation, a Bus clash detection (responsible for resetting in extreme conditions) and a stream synchronization which is not implemented as it's not essential.

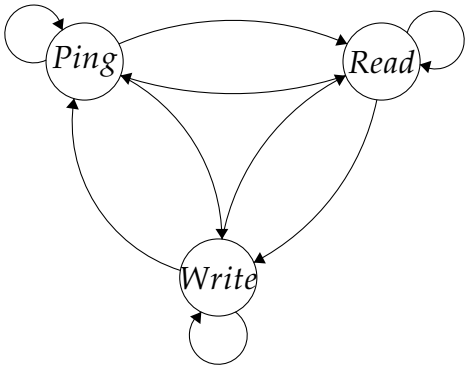


Figure 3.8: Read, Ping, Write - Finite State Machine

3.8.4 FSM

Two finite state machines, one for the enumeration process and one for the general functionality had to be designed. Figure 3.8 depicts the FSM responsible for choosing the command words to be transmitted. Changing between each available state without restrictions is possible. The SoundWire controller in idle state sends ping commands as long as no new slave has been attached or no read or write commands have to be sent.
Thinking about the later discussed enumeration process starting with pings. When a slave is attached, read commands have to follow and afterwards a write to a slave register is necessary. In the end, the process jumps to the ping state again.
Hard implemented sequences of register read and writes can be executed. In case of doing incremental read or writes using the Python software, one has to consider the I²C, APB interfaces and, additionally the clock domain crossing. Thus, in between write and read commands always some frames having a ping command will be transmitted.

Enumeration describes the process in a start-up phase between controller and the attached slave. Without successful enumeration, no communication is possible. This includes at least one ping (Idle), six read (R1-R6) and one write (W) command. Pinging is necessary to see whether a slave is attached or not. This is being signalled in the SlaveStat00 bit fields. Slaves demanding for enumeration will indicate this in intended frame section. Included in the enumeration process is a de-attaching functionality implemented in each slave to prevent wrong communication attempts. The different stages of enumeration are listed below:

1) Ping: Slave signals ready for enumeration state
2) Read: Controller reads SCP_DevId_0 register
3) Read: Controller reads SCP_DevId_1 register
4) Read: Controller reads SCP_DevId_2 register
5) Read: Controller reads SCP_DevId_3 register
6) Read: Controller reads SCP_DevId_4 register
7) Read: Controller reads SCP_DevId_5 register
8) Write: Controller writes to Device Number register

This sequence has to be followed strictly. Each interruption shall end in aborting the process and restarting it if the slave is still attached, rather than retrying a command at a advanced stage of enumeration. The reading submits integrated and unchangeable hardware information about the slave. Last command is necessary to give the slave a name to be remembered by the controller and for distinction if other slaves (up to 11) would have been attached.

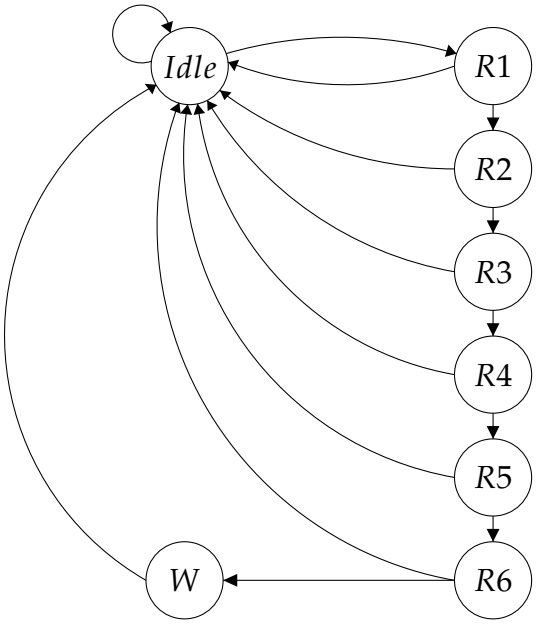


Figure 3.9: Enumeration Finite State Machine

3.8.5 Synchronization

Each control-word includes synchronization bits for a stable connection. Slaves can derive the current position in a frame by them. The synchronization consists of two parts. The static sync-word is transmitted in every frame on the same position in the control word. For the dynamic pattern, a PRBS has to be implemented. Using a Linear Feedback Shift Register (LFSR), the required dynamic words can be generated in the desired sequence. Four Flip-Flops starting in state 1 each. The end of a frame signals enables generation of the next pattern. Input

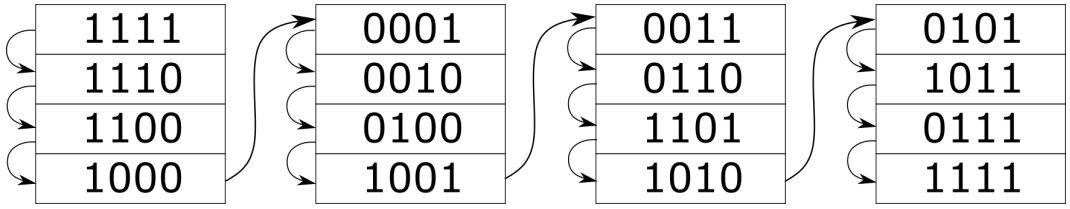


Figure 3.10: Sequence of Dynamic Synchronisation Patterns

of FF two is the output of the first. This sequence is continued for the following Flip-Flops. Output of FF number three and four are feedback including a XNOR to the input of FF one. Depicted in 3.10 is one run through the pattern sequence to be generated.

3.8.6 PHY

PHY is the module responsible for a DDR transmission. Controlling the ODDR and Input Double Datarate Register (IDDR) is implemented here. Encoding the NRZI and in combination the parity control was specified in the PHY to allow a fast reaction time (max 2 clk periods).

ODDR and IDDR are used to handle the signal output for both clock edges. One Flip-Flop is not aware of driving signals like this. Thus instantiating those dedicated registers was necessary. FPGA's include those functional modules in their Output Logic (OLOGIC) and Input Logic (ILOGIC). Each in turn can be split-up into two FF's and two MUX's. The block in-depth description can be found in Xilinx, 2018. Configuration modes OPPOSITE_EDGE and SAME_EDGE are valuable for timing issues.

- OPPOSITE_EDGE: The date inputs are sampled at both rising and falling clock edges
- SAME_EDGE: Both data inputs to the ODDR are sampled at the same time

The opposite edge is used for outputing the SoundWire signal. As the data line can be driven by a master and slave having different driving slots, the line switching has to be considered in the encoding for each other. Thus, the time between rising and falling edge is needed. For the input signals, this can also be applied.

Bus holder is another element placed in PHY. SoundWire connections submit their information by changing the signal level. If no informational content has to be transmitted, accordingly zero sequences, the voltage on the line must not switch. A combination of two twisted inverters enable such a behaviour. The bus-holder needs to be placed after the OLOGIC or before ILOGIC block.

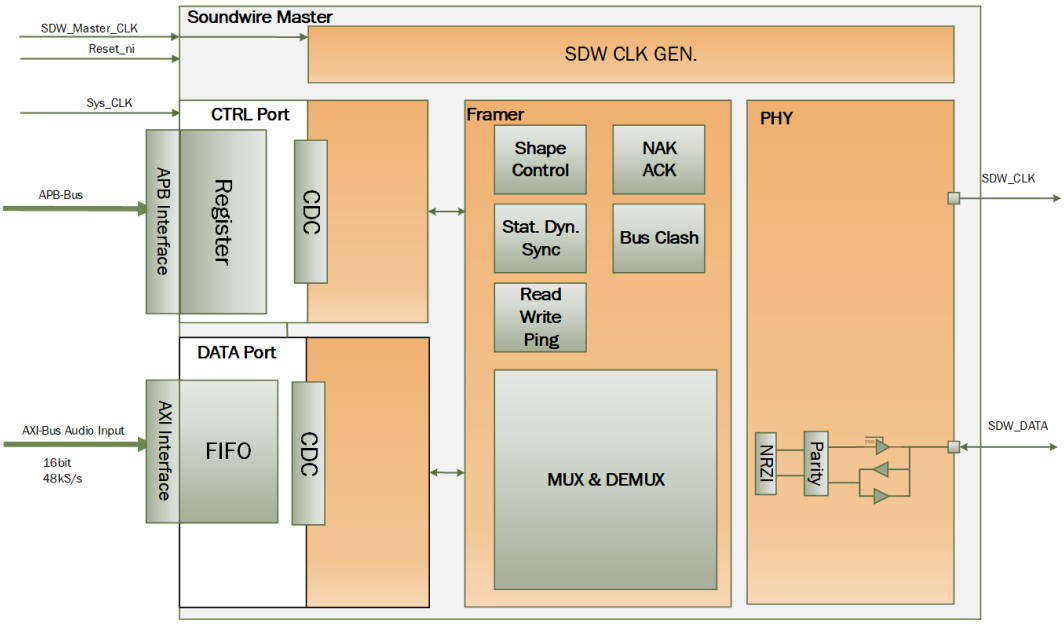


Figure 3.11: Block Diagram of Implemented SoundWire Controller

4 Implementation and Evaluation of the SoundWire Controller Interface

In this chapter, the detail interface implementation is described. Displayed code is written in System Verilog language. Xilinx Vivado 18.3 was the software used for programming. In the following section, 4.1, some basic styles of System Verilog code and the cell/block they will result in are shown.

4.1 System Verilog Basic

Each module is defined at the beginning by its inputs and outputs (lines 3-9 in 4.1). System Verilog is capable of recognizing net-types. To be aware and self responsible of the actual design, it's possible to prevent this by setting a ‘default_nettype none in front of a module. This can be seen in the first line of the example code snipped. At the end this must be reverted ‘default_nettype wire. Definitions like registers for counting, which are only used in this module, are being placed below (lines 11-12). Afterwards, functional programming is done.

Listing 4.1: example code snipped of System Verilog language

```
1  `default_nettype none
2
3  module sdw_... (
4    input   wire       sdw_clk,
5    input   wire       sdw_reset,
6    input   wire       sdw_x,
7    output  wire       sdw_y,
8    output  reg [4:0] sdw_o
9  );
10
11 reg [3:0] reg_a;
12 reg [3:0] reg_b;
13
14 always_ff @(negedge sdw_clk or negedge sdw_reset) begin
15   if(~sdw_reset) begin
16     reg_a <= 4'b0000;
17     reg_b <= 4'b0000;
18   end else begin
19     if (sdw_x) begin
20       reg_a <= reg_a + 1'b1;
21       reg_b <= reg_b + 2'b10;
22     end
23   end
24 end
25
26 always_comb begin
27   sdw_o = reg_a + reg_b;
28 end
29 endmodule
30 `default_nettype wire
```

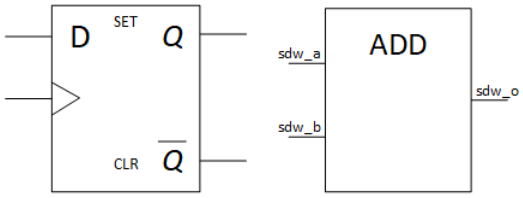


Figure 4.1: Generated Flip-Flop and Combinatorial Blocks

Flip-Flop

Example 4.1 shows how to generate a Flip-Flop by adding _ff to the always statement. A clock input "sdw_clk" and a reset signal "sdw_reset" are mandatory. For proper functionality, all signals shall be set to a value by the reset signal to avoid unpredictable conditions. In this case, this is done by negative edge of the signal. After reset conditions, the intended functionality can be set. reg_a is incremented by a value of one in binary and reg_b by two. Line 19 indicates a additional enable input for the FF. Both registers are incremented only if this is true.

Combinatorial

Another block integrated in the example is a combinatorial. reg_a and reg_b are added up and stored to sdw_o. Always_comb blocks are often used for case statements e.g. a test multiplexer.

Further, two important behaviours are additionally shown in the example. There is a significant difference between "=" and "¡=" statements. Primary is a so called blocking assignment meaning the code is being handled in sequences. The other is a non-blocking assignment. Those can be processed simultaneously.

4.2 SoundWire Implementation

In the following, not the whole code is being displayed but valuable snippets have been taken out and are discussed.

4.2.1 Top

Instantiation of all sub-blocks is done in the SoundWire top module. Signals are passed trough and split up here. By setting the right register, a switch from audio data, generated by the stimgen module to the data via I²S, interface can be done.

Listing 4.2: Audio Data Switch

```
1  ...
2      3'h0: begin // STIMGEN
3              sdw_dp_tx_slv_tdata_i          = sdw_dp_tx_slv_stimgentdata_i;
4              sdw_dp_tx_slv_tvalid_i         = sdw_dp_tx_slv_stimgentvalid_i;
5              sdw_dp_tx_slv_stimgentready_o = sdw_dp_tx_slv_tready_o;
6            end
7      3'h1: begin // I2S
8              sdw_dp_tx_slv_tdata_i          = sdw_dp_tx_slv_i2stdata_i;
9              sdw_dp_tx_slv_tvalid_i         = sdw_dp_tx_slv_i2stvalid_i;
10             sdw_dp_tx_slv_i2stready_o      = sdw_dp_tx_slv_tready_o;
11           end
12 ...
```

Testmux

For testing reasons, a so called Test-MUX is integrated into the top module. By this, MUX, an easy swap between signals intended for output is realized. Minimizing the actual amount of output pins as well as increasing the possibility to display a huge number of signals via them is the advantage. In this case a 8-bit (per signal) test-MUX is implemented. mcp_test_mux_q (line 3) is the register responsible for switching the signal set. Listed in 4.3 is the first and later mostly used signal set (lines 7-15). Especially, the accurate slave status slv_ctrl_word_slv_stat_01, physical output sdw_phy_data_bit_dd_i_phy_o and displaying enumeration steps by next_enum_step_trig was helpful.

Listing 4.3: Testmux

```
1  always_comb begin
2  test_mux_val = 16'hoooo;
3  case (mcp_test_mux_q)
4  4'h0: begin
5  test_mux_val = 16'hoooo;
6  end
7  4'h1: begin
8  test_mux_val[0]  = sdw_clk_o;
9  test_mux_val[1]  = sdw_frame_data_bit_i_one;
10 test_mux_val[2]  = cnt_reset;
11 test_mux_val[3]  = sdw_phy_data_bit_dd_i_phy_o;
12 test_mux_val[4]  = sdw_frame_data_bit_i_two;
13 test_mux_val[5]  = sdw_frame_data_bit_i_one;
14 test_mux_val[6]  = next_enum_step_trig;
15 test_mux_val[7]  = slv_ctrl_word_slv_stat_01[0] ;
16 end
```

4.2.2 Control

Implementation of the SoundWire control part included four subcategories: Handling the clock (along with turning it ON and OFF synchronously), the start-up POR condition processing, a finite state machine and clock domain crossing for reading and writing slave registers.

Clock Core of each interface is a proper performing master clock. To process the MAX98374, a 12.288MHz SDW-Clock is necessary. As mentioned in design chapter 3, the clock is generated by an extra module (created using a Vivado clocking wizard) on the FPGA and couldn't be created by an internal divider due to a non-integer division. Configuration of the SoundWire module is all done by the 41.472MHz system clock. That includes switching the generated sdw-clock

ON and OFF. Doing so a synchronization of the clock enable signal is needed. In the code snippet sdw_master_clk_i represents the generated 12.288MHz clock. mcp_ctrl_clkoff_i is the register related switch in system clock domain. By this mcp_ctrl_clkoff_en is generated. A synchronized signal, capable of turning ON and OFF the clock, avoids metastability (lines 8-15).

Listing 4.4: SDW Clock

```
1  std_sync2_ff_rn u_sdw_clk_en_sync(
2    .RN (sdw_reset_ni),
3    .CLK(sdw_master_clk_i),
4    .D  (mcp_ctrl_clkoff_i),
5    .Q  (mcp_ctrl_clkoff_en)
6  );
7
8  always_comb begin
9    if (mcp_ctrl_clkoff_en) begin
10     sdw_data_clk_sig = sdw_master_clk_i;
11   end else begin
12     sdw_data_clk_sig = 1'b0;
13   end
14 end
```

POR At each start-up or reset, the controller performs a special procedure. A sequence of at least 4096 continuous ones has to transmitted on the data line. Therefore, DDR has to be considered. A high level of sdw_ctrl_por_start_o (line 4) is the stimulus for a power on reset start condition till sdw_ctrl_por_end_o goes high. This signal is the result of the counter displayed from row 11 to 22 and the combinatorial afterwards. As soon as the counter attains hexadecimal 12'h7FF sdw_por_cnt_ff is being reset and the POR-phase ends thus sdw_ctrl_por_end_o is on high level (line 25). Hexadecimal 7FF is 2048 in decimal. Due to the fact that in one clock period two bits are being transmitted, this results in 4096 bits.

Listing 4.5: POR

```
1  always_ff @ (negedge sdw_data_clk or negedge sdw_reset_ni) begin
2    if (~sdw_reset_ni) begin
3      sdw_ctrl_por_start_o <= 1'b1;
4    end else begin
5      if (sdw_ctrl_por_end_o) begin
6        sdw_ctrl_por_start_o <= 1'b0;
7      end
8    end
9  end
10
11 always_ff @(negedge sdw_data_clk or negedge sdw_reset_ni) begin
12   if(~sdw_reset_ni) begin
13     sdw_por_cnt_ff <= 12'h000;
14   end else begin
15     if (sdw_ctrl_por_end_o) begin
16       sdw_por_cnt_ff <= 12'h000;
17     end else
18     if (sdw_ctrl_por_start_o) begin
19       sdw_por_cnt_ff <= sdw_por_cnt_ff + 12'h001;
20     end
21   end
22 end
23
24 always_comb begin
25   sdw_ctrl_por_end_o = (sdw_por_cnt_ff == 12'h7ff);
26 end
```

Register FSM In this section the finite state machine for processing slave register read and write commands is discussed. The first lines (5-11) of the code are reset values of following signals.

- sdw_slv_rdy: Low if an read or write process is going on
- sdw_slv_read_start: Signals starting a read command
- sdw_slv_write_start: Signals starting a write command
- sdw_slv_op_read_den_ff: Signals finished read command
- sdw_slv_op_write_den_ff: Signals finished write command

At the beginning, the FSM stays in IDLE mode. If mcp_ctrl_enumen_i (row 14) is set, enumeration can not be processed due to repeatedly IDLE state. As soon as enumeration is done, read or write commands can be processed depending on the slaves ready status. Incoming trigger slv_rw_trig causes a jump to line 26 CMD_DEC. Determined by slv_rw, either a read or write command will follow. Exceptions are if both emerge at the same time or the content of slv_rw is low. A jump to SLV_ERROR and further to IDLE mode will follow restarting requests.

As the FSM reaches CMD_READ_EXE, sdw_slv_read_start which is part of cmd_read_exe_state is set high. This is needed to grant a successful transmission via the clock domain crossing explained in the next section. Response by the CDC in sdw_slv_op_read_den_ff leads to a finished read command, thus a jump to CMD_DONE (row 86) where signals are set to default mode again and further to IDLE happens. The same procedure is followed by a write condition.

Listing 4.6: Register FSM

```
1  always_comb begin //ff @(posedge clk or negedge sdw_reset_ni) begin
2    if (~sdw_reset_ni)
3      fsm_irq = 1'b0;
4      next = IDLE;
5      sdw_slv_rdy = 1'b0;
6      sdw_slv_read_start  = 1'b0;
7      sdw_slv_write_start = 1'b0;
8      sdw_slv_op_read_den_ff = 1'b0;
9      sdw_slv_op_write_den_ff = 1'b0;
10   case (state)
11     IDLE:      begin
12                  if (mcp_ctrl_enumen_i ) begin
13                    next = IDLE;
14                    sdw_slv_rdy = 1'b1;
15                  end else
16                  if (slv_rw_trig) begin
17                    next = CMD_DEC;
18                    sdw_slv_rdy = 1'b0;
19                  end else begin
20                    next = IDLE;
21                    sdw_slv_rdy = 1'b1;
22                  end
23                end
24     CMD_DEC:   begin
25                  sdw_slv_rdy = 1'b0;
26                  case (slv_rw)
27                    SLV_IDLE:  begin
28                                 next     = IDLE;
29                                 fsm_irq = 1'b0;
30                                 sdw_slv_read_start  = 1'b0;
31                                 sdw_slv_write_start = 1'b0;
32                                 sdw_slv_op_read_den_ff = 1'b0;
33                               end
34                    SLV_READ:  begin
35                                 next     = CMD_READ_EXE;
36                                 fsm_irq = 1'b1;
37                               end
38                    SLV_WRITE: begin
39                                 next     = CMD_WRITE_EXE;
40                                 fsm_irq = 1'b1;
41                               end
42                    SLV_ERROR: begin
43                                 next     = IDLE;
44                                 fsm_irq = 1'b0;
45                                 sdw_slv_read_start     = 1'b0;
46                                 sdw_slv_write_start    = 1'b0;
47                                 sdw_slv_op_read_den_ff = 1'b0;
48                               end
49                      default: begin
```

```
50                                next     = IDLE;
51                                fsm_irq = 1'bo;;
52                              end
53                endcase
54              end
55      CMD_READ_EXE: begin
56                      sdw_slv_rdy     = 1'bo;
57                      sdw_slv_op_read_den_ff = 1'bo;
58                      case (cmd_read_exe_state)
59                        2'boo:  begin
60                                  next = CMD_READ_EXE;
61                                  sdw_slv_read_start     = 1'b1;
62                                  sdw_slv_op_read_den_ff = 1'bo;
63                                end
64                        2'b10:  begin
65                                  next = CMD_READ_EXE;
66                                  sdw_slv_read_start     = 1'b1;
67                                  sdw_slv_op_read_den_ff = 1'bo;
68                                end
69                        2'b11:  begin
70                                  next = CMD_DONE;
71                                  sdw_slv_op_read_den_ff = 1'b1;
72                                end
73                        2'bo1:  begin
74                                  next = CMD_DONE;
75                                  sdw_slv_op_read_den_ff = 1'b1;
76                                end
77                       default: begin
78                                  next = IDLE;
79                                  sdw_slv_op_read_den_ff = 1'bo;
80                                end
81                      endcase
82                    end
83    CMD_WRITE_EXE: begin // Write Register to Slave
84                    ...
85                    end
86        CMD_DONE:   begin
87                      next                    = IDLE;
88                      sdw_slv_rdy             = 1'bo;
89                      fsm_irq                 = 1'bo;
90                      sdw_slv_read_start      = 1'bo;
91                      sdw_slv_write_start     = 1'bo;
92                      sdw_slv_op_write_den_ff = 1'bo;
93                      sdw_slv_op_read_den_ff  = 1'bo;
94                    end
95    endcase
96  end
```

Register CDC Processing the steps of enumeration is captured all over in SoundWire clock domain as no input from the outside is required. Thus, crossing clock domains is not necessary. In case of transmitting data to slave registers, by the APB related registers in system clock domain, a switch to SDW clock is unavoidable. To prevent metastability, a CDC has been implemented in accordance to the example block diagram 4.2.

In the following, actual code snippets related to each part of the block diagram will be listed and discussed to get into the functionality.

At the beginning, a start signal reg_cdc_start is captured by the FF (sign 1).

Listing 4.7: Register CLock Domain Crossing Start Signal

```
1  always_ff @(posedge clk or negedge sdw_reset_ni) begin
2    if (~sdw_reset_ni) begin
3      reg_cdc_start_ff <= 1'bo;
4    end else begin
5      reg_cdc_start_ff <= reg_cdc_start;
6    end
7  end
```

The resulting Q signal reg_cdc_start_ff is part of two further steps that are both still in system clock domain. Information necessary for reading or writing slave registers like addresses, device number, op-code and data will be "prestored" by FF's when the start signal enables this (see 4.8 line 7). Shown in the block diagram, this captured by the big sized flip-flop numbered as 2.

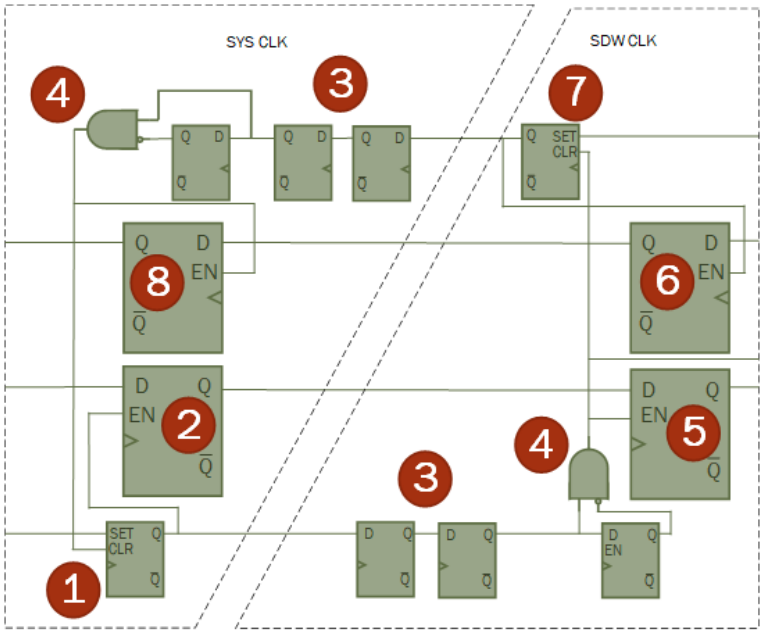


Figure 4.2: Clock Domain Crossing

Listing 4.8: Register CLock Domain Register Data

```
always_ff @(posedge clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    reg_address_sys_clk_ff    <= 16'h0000;
    reg_rw_sys_clk_ff         <= 3'b000;
    reg_write_data_sys_clk_ff <= 8'h00;
  end else begin
  if (reg_cdc_start_ff) begin
    reg_address_sys_clk_ff     <= {sdw_slvhigh_addr_i, sdw_slvlow_addr_i};
    sdw_slvdev_addr_sys_clk_ff <= sdw_slvdev_addr_i;
  ...
```

Further usage of reg_cdc_start_ff is synchronizing into SDW clock domain. A pre-made cell std_sync2_ff_rn is instantiated for this. The cell consists of two flip-flops related to number 3 in the figure. Triggered by the new SoundWire clock signal, the reg_cdc_start_ff can be safely passed through without causing metastability problems.

Listing 4.9: Register CLock Domain Synchronization Cell

```
std_sync2_ff_rn u_sdw_reg_start_sync(
  .RN (sdw_reset_ni),              // I; async reset (active low)
  .CLK(sdw_data_clk),              // I; clock
  .D  (reg_cdc_start_ff),          // I; data in
  .Q  (reg_cdc_sdw_start_sync_q)   // O; registered data
);
```

The output of the synchronization cell is then transformed into a pulse. Combinatorial AND delivers this by applying input signals reg_cdc_sdw_start_sync_q and negated reg_cdc_sdw_start_sync_q_ff which is the output of another FF, capturing again reg_cdc_sdw_start_sync_q. This can be seen in the block design enumerated as point 4.

Listing 4.10: Register CLock Domain Pulse

```
always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    reg_cdc_sdw_start_sync_q_ff <= 1'b0;
  end else begin
    reg_cdc_sdw_start_sync_q_ff <= reg_cdc_sdw_start_sync_q;
  end
end

always_comb begin
  reg_cdc_sdw_start = reg_cdc_sdw_start_sync_q & ~reg_cdc_sdw_start_sync_q_ff;
end
```

The resulting pulse reg_cdc_sdw_start of previous processing steps provides the enable for all register data FF's (number 5) to take over the output data of data FF's in system clock domain (number 2). All of this securely handled data is transmitted to the slave either to read or write registers.

Listing 4.11: Register SDW CLock Domain

```
always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    reg_address_sdw_clk_ff     <= 16'h0000;
    reg_rw_sdw_clk_ff          <= 3'b000;
    reg_write_data_sdw_clk_ff  <= 8'h00;
    reg_cdc_rw_req             <= 1'b0;
    sdw_slvdev_addr_sdw_clk_ff <= 4'b0000;
    end else begin
      if (reg_cdc_sdw_start) begin  // rise request condition for frame fsm
        reg_address_sdw_clk_ff     <= reg_address_sys_clk_ff;
        sdw_slvdev_addr_sdw_clk_ff <= sdw_slvdev_addr_sys_clk_ff;
        reg_write_data_sdw_clk_ff  <= reg_write_data_sys_clk_ff;
        reg_rw_sdw_clk_ff          <= reg_rw_sys_clk_ff;
        reg_cdc_rw_req             <= 1'b1;
      end else begin
      if (reg_cdc_sdw_done) begin
        reg_cdc_rw_req <= 1'b0;
      end
    end
  end
end
```

The slave responds by ACK that operation was functional or by Not Acklowledge (NAK), that a failure occurred. This answer is processed by the framer and if implemented, another try to process the request can be started or it must be aborted. However after finishing this reg_cdc_sdw_done (signed in figure as 7), it results in finishing the CDC (see listing 4.12).

Listing 4.12: Register SDW CLock Domain Done

```
always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    reg_cdc_sdw_done_pre <= 1'b0;
  end else begin
    reg_cdc_sdw_done_pre <= (reg_cdc_rw_req && reg_cdc_data_hand_fin_i);
  end
end

always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    reg_cdc_sdw_done <= 1'b0;
  end else begin
    reg_cdc_sdw_done <= reg_cdc_sdw_done_pre;
  end
end
```

Another synchronization cell is placed now in the system clock domain (number 3). Input to this is the flip-flop captured reg_cdc_sdw_done. The same is used to take over register data, read from the slave signed as 6.

Listing 4.13: Register SYS CLock Domain Done

```
1  std_sync2_ff_rn u_sdw_reg_stop_sync(
2    .RN (sdw_reset_ni),              // I; async reset (active low)
3    .CLK(clk),                       // I; clock
4    .D  (reg_cdc_sdw_done),          // I; data in
5    .Q  (reg_cdc_sys_done_sync_q)    // O; registered data
6  );
```

Now, the signal is back, triggered by the system clock. The same procedure as previously seen in enumeration 4 is then processed. The resulting signal reg_cdc_sys_done is then used to pass data (sign 8), read from the slave, safely into system clock domain as shown in 4.14 if available and to finalize the whole read or write processes by setting the signals sdw_slv_read_end and sdw_slv_write_end high (see listing 4.14 from lines 11 to 30).

Listing 4.14: Register Read Slave Data

```
1  always_ff @(posedge clk or negedge sdw_reset_ni) begin
2    if (~sdw_reset_ni) begin
3      sdw_slv_data_d_o <= 8'h00;
4    end else begin
5      if (reg_cdc_sys_done) begin
6        sdw_slv_data_d_o <= reg_read_data_sdw_clk_ff;
7      end
8    end
9  end
10
11 always_ff @(posedge clk or negedge sdw_reset_ni) begin
12   if (~sdw_reset_ni) begin
13     reg_cdc_sys_done_ff <= 1'b0;
14     sdw_slv_read_end    <= 1'b0;
15     sdw_slv_write_end   <= 1'b0;
16   end else begin
17     if (sdw_slv_read_start & reg_cdc_sys_done) begin
18       reg_cdc_sys_done_ff <= reg_cdc_sys_done;
19       sdw_slv_read_end    <= 1'b1;
20     end else
21     if (sdw_slv_write_start & reg_cdc_sys_done) begin
22       reg_cdc_sys_done_ff <= reg_cdc_sys_done;
23       sdw_slv_write_end <= 1'b1;
24     end else begin
25       sdw_slv_read_end    <= 1'b0;
26       sdw_slv_write_end   <= 1'b0;
27       reg_cdc_sys_done_ff <= 1'b0;
28     end
29   end
30 end
```

This whole approach ensures safely passing data in-between two clock domains without problems of metastability. Especially for audio related systems, this is needed to avoid glitches. Thus, I²S data needs the same process (as can be seen in following subsection).

4.2.3 Data Port

Soundwire protocol would be capable of organizing multiple audio data ports. Due to the field of application, only one input port is implemented. As the transport of audio data is enabled, a request to pass I²S data is sent by the framer to provided data port (line 1 in listing 4.15). The request sets sdw_rx_update_tgl_ff to a high level and after a successful clock domain crossing this is converted into sdw_rx_done signal responsible to reset the start signal to it's primary zero condition (line 11).

Listing 4.15: Data Port CDC start

```
assign sdw_rx_data_cnt_rdy = new_samp_trig;

always_ff @ (posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    sdw_rx_update_tgl_ff <= 1'b0;
  end else begin
    if (sdw_rx_data_cnt_rdy) begin
      sdw_rx_update_tgl_ff <= 1'b1;
    end else begin
      if (sdw_rx_done) begin
        sdw_rx_update_tgl_ff <= 1'b0;
      end
    end
  end
end
```

Regarding block diagram 4.2 in this case the process starts at sign 7 as the requests origin is in SDW clock domain. Flip-Flops signed as 2 and 5 for transmitting data in the other direction can be neglected. After synchronizing (listing 4.16), the request to system clock it's used to demand data from the FIFO via sdw_rx_update_q.

Listing 4.16: Data Port CDC synchronization

```
std_sync2_ff_rn u_tx_sdw_req_sync(
  .RN (sdw_reset_ni),              // I; Async Reset (Active Low)
  .CLK(clk),                       // I; Clock
  .D  (sdw_rx_update_tgl_ff),      // I; Data IN
  .Q  (sdw_rx_update_tgl_sync_clk) // O; Registered Data
);

always_ff @ (posedge clk or negedge sdw_reset_ni) begin
  if (~sdw_reset_ni) begin
    sdw_rx_update_tgl_sync_clk_q <= 1'b0;
  end else begin
    sdw_rx_update_tgl_sync_clk_q <= sdw_rx_update_tgl_sync_clk;
  end
end

always_comb begin
  sdw_rx_update <= sdw_rx_update_tgl_sync_clk & ~sdw_rx_update_tgl_sync_clk_q;
end

always_ff @ (posedge clk or negedge sdw_reset_ni)begin
  if (~sdw_reset_ni) begin
    sdw_rx_update_q <= 1'b0;
  end else begin
    sdw_rx_update_q <= sdw_rx_update;
  end
end
```

In the following section, the instantiation of a 32 bit FIFO register is shown. This structure was taken over completely from the generated VIVADO FIFO implementation. As the decision fo maximum 16-bit data was made later, this has not been changed. This led to an interesting point for comparison in area expense being discussed in later result section 4.4. Taking data output of the register dp_fifo_data launches another data requesting sequence related to the AXI - streaming interface.

Listing 4.17: Data Port FIFO Instantiation

```
sdw_fifo_w32_d8_A #(
  .DATA_WIDTH (32),  // Data width
  .ADDR_WIDTH ( 8),  // Address width
  .DEPTH      (64)   // depth
) u_sdw_fifo_w32_d8(
  .clk         (clk_gated),                  // I; Gated Clock
  .reset_ni    (sdw_reset_ni),               // I; Active low Reset
  .clear       (),                           // Clear FIFO
  .if_empty_n  (fifo_if_empty_n),            // O; FIFO Empty
  .if_read_ce  (),                           // I; FIFO read
  .if_read     () ,                          // I; FIFO read
  .if_dout     (dp_fifo_data),               // O; FIFO Data output
  .if_full_n   (fifo_if_full_n),             // O; FIFO full
  .if_write_ce (sdw_rx_data_cnt_rdy),        // I; STIMGEN DATA ready
  .if_write    (sdw_rx_data_cnt_rdy),        // I; STIMGEN DATA ready
  .if_din      (sdw_dp_rx_stimgen_tdata)     // I; FIFO Data Input
);
```

sdw_rx_update transmits data request information to the streaming interface. In case of available valid data (sdw_dp_tx_slv_tvalid_i), the FIFO is allowed to take over audio data sdw_dp_rx_stimgen_tdata. This cycle ensures a stable and ongoing audio data transportation.

After audio data has been taken out of the FIFO based on sdw_rx_update_q inquiry (line 5 in 4.18 and point number 8 in the block diagram), the process of turning the request into sdw_rx_done signal starts. Further explanation can be seen in previous register clock domain crossing as the process is the same.

Listing 4.18: FIFO Data

```
1 always_ff @(posedge clk or negedge sdw_reset_ni)begin
2   if (~sdw_reset_ni) begin
3     dp_rx_data_pre_sync <= 32'hoooooooo;
4   end else begin
5     if (sdw_rx_update_q) begin
6       dp_rx_data_pre_sync <= dp_fifo_data;
7     end
8   end
9 end
```

4.2.4 Framer

The SoundWire Framer is responsible for the mapping, de-mapping and correct assignment of audio and control data on the tranmission/receiver line. Thus, all input and output data is processed by it. As previously mentioned, frame sizing is placed here. Despite only 48x2 frames are used, others would be available via setting register, but will not work properly due to uncorrelated and not implemented bit mapping. The most relevant modules for proper protocol functionality are instantiated here. The LFSR responsible for dynamic synchronization as well as the finite state machines for enumeration and read,ping, write commands. Further, the data line output enable and handling of various inquiries coming from e.g. enumeration or control module are controlled here.

The beginning of list 4.19 displays requests passed by the control unit. Either register-write or reads can be further operated (lines 1-4). The remaining part is handling the next steps to be processed. Distinction can be made between enumeration or register related. enum_fsm_ack in row 14 enables treatment of enumeration concerning steps and refuses any others. Lines 15 to 22 are needed to signal further device ID readings.

The last step of enumeration is writing a device number to the slave. This has to be accepted by the slave signalling via an ACK. If the combination of enum_pos , cmd_ok and end_of_frame is matching, the process can be finalized causing the slave to be available via his new ID and enabling register read or write command starting from lines 31 to 38. As before, those commands have to be acknowledged by the slave or aborted if not.

Listing 4.19: Request Management

```
always_comb begin
  reg_cdc_r_ack = sdw_slv_read_start_i && reg_cdc_rw_ack;
  reg_cdc_w_ack = sdw_slv_write_start_i && reg_cdc_rw_ack;
end

always_comb begin
  if(~sdw_reset_ni)begin
    reg_read_data_sdw_clk_ff = 8'h00;
    reg_cdc_data_hand_fin    = 1'b0;
    reg_enum_read_data_ff    = 8'h00;
    next_enum_step           = 1'b0;
    next_slv_id              = 1'b0;
  end else begin
    if (enum_fsm_ack) begin
      if ((reg_cdc_ack_cnt >= 12'h040) && end_of_frame) begin
        next_enum_step = 1'b1;
        next_slv_id    = 1'b0;
        reg_enum_read_data_ff = slv_ctrl_word_read_reg_data;
      end else begin
        next_enum_step = 1'b0;
        next_slv_id    = 1'b0;
      end
    end else begin
      if (enum_pos == 3'b111 && (cmd_ok && end_of_frame)) begin
        next_enum_step = 1'b1;
        next_slv_id = 1'b1;
        end else begin
        next_enum_step = 1'b0;
        next_slv_id = 1'b0;
        reg_cdc_data_hand_fin = 1'b0;
        if (reg_cdc_r_ack | reg_cdc_w_ack) begin
          reg_cdc_data_hand_fin = 1'b0;
        end else begin
          if ((reg_cdc_ack_cnt >= 12'h040) && (cmd_ok && end_of_frame)) begin
            reg_read_data_sdw_clk_ff = slv_ctrl_word_read_reg_data;
            reg_cdc_data_hand_fin    = 1'b1;
          end else begin
            reg_read_data_sdw_clk_ff = 8'h00;
          end
        end
      end
    end
  end
end
```

The whole process of acceptation or aborting requests is done by evaluating slaves ACK and NAK responses stored in the nak_ack signal. A combinatorial decides in respect to nak_ack how to further process requests.

Listing 4.20: Acknowledge and Not Acknowledge handling

```
assign  nak_ack = {slv_ctrl_word_nak,slv_ctrl_word_ack};

always_comb begin
  case (nak_ack)
  2'b00: begin
           cmd_ignored = 1'b1;
           cmd_ok      = 1'b0;
           cmd_abort   = 1'b0;
           cmd_failed  = 1'b0;
         end
  ...
```

After a decision is made whose operation is accepted and can be treated further, the control word structure can be set. Different combinations of slots regarding register addresses or reading data in contrast to write data has to be considered. Combinatorial decision of structure is made based on cdc_enum_rpw. Shown in the listing below is the composition of a ping request. Just like there are switching parts of the protocol, there are some always stuck with the same data or manager of data. Especially the synchronization is considered to be controlled by the same.

Listing 4.21: Control Word Structure

```
1  assign cdc_enum_rpw = {read_ping_data,enum_fsm_ack,reg_cdc_r_ack,reg_cdc_w_ack,mcp_ctrl_enumen_i};
2
3  always_comb begin
4    case (cdc_enum_rpw)
5      5'b10001: begin
6                  write_check = 1'b0;
7                  ctrl_word_ping_req       = 1'b0;
8                  ctrl_word_prw            = 3'b000;
9                  ctrl_word_ssp            = 1'b0;
10                 ctrl_word_dev_addr[3]    = 1'b0;
11                 ctrl_word_dev_addr[2]    = ctrl_word_ssp;
12                 ctrl_word_dev_addr[1]    = ctrl_word_bus_req;
13                 ctrl_word_dev_addr[0]    = ctrl_word_bus_rel;
14                 ctrl_word_reg_addr       = 4'h0000;
15                 ctrl_word_write_reg_data = 8'h00;
16               end
17       ...
```

LFSR - Dynamic Synchronization

A LFSR is needed to create the dynamic synchronization patterns in the correct sequence. In section 3.8.5, the chronology is shown starting by pattern 1111. Line 2 in the listing defines a reset condition active low resulting in executing the following statement. If the signal sdw_reset_ni switches to a low level, the lfsr_dyn_sync pattern is set to the starting 1111 word. The same as previous has to be performed after a power on reset condition when 2048 ones are transmitted and then the first frame is sent. cnt_reset defines the end of a frame and results in generating a new bit-word (rows 8-11). Depending on the end of frame state, the lfsr_dyn_sync is shifted by one to the left. Bit 0 is the result of merging the last two by an XOR.

Listing 4.22: LFSR

```
1  always_ff @(negedge sdw_data_clk or negedge sdw_reset_ni) begin
2    if(~sdw_reset_ni) begin
3      lfsr_dyn_sync <= 4'b1111;
4    end else begin
5      if (sdw_por_start_i) begin
6        lfsr_dyn_sync <= 4'b1111;
7      end else
8      if (cnt_reset) begin
9        lfsr_dyn_sync <= lfsr_dyn_sync << 1'b1;
10       lfsr_dyn_sync[0] <= lfsr_dyn_sync[3]^lfsr_dyn_sync[2];
11     end
12   end
13 end
```

RPW-FSM

The Read Ping Write - Finite State Machine handles requests for different opcodes and controls the order of consecutive frames. After a maximum of 32 frames read or write commands, at least one ping command has to follow. Shown in the first part of the code is a counter responsible for that. Basic transmission opcode is a ping command, thus it's set in the reset condition. In case of a read or write (read_ping_data_o equals a high level), the counter next_ping_cnt is incremented after a frame shown in rows 14-15. Line 6 is an input of the FSM, signalling a sent ping and resetting the counter. When incrementation catches up, 32 signal ping_slv is set high to force a ping opcode.

Rows 21-31 set states for the state machine. cnt_reset_i enables determining the state for the next frame. Otherwise results are either remaining in the same state or a reset to PING. The FSM itself can be separated into mainly two functionalities. Ping command starting in line 35 on the one hand and read or write on the others starting at 46. Listed signals below come up in both.

- reg_cdc_rw_ack_o: If a read or write is requested this signal is set high if conditions match.
- enum_fsm_ack_o: The Enumeration is done primary after POR condition and is privileged
- read_ping_data_o: This is used for resetting previous mentioned ping command incremental counter
- ctrl_word_prw_o: The control word which has to be transmitted in the op-code to enable a ping (000), read (010) or write (011)

In state PING, a query by the if statement in line 41 is responsible to determine whether remaining in the same state or jumping to state READ_WRITE. Two conditions have to be fulfilled. The main thing is a finished POR and the other is either a read-write request or the Enumeration-FSM signalling that an enumeration of a slave is pending. In turn, those two are the main contributors to the READ_WRITE state. Starting at row 47, the enumeration is treated. At 57, the start condition for a register read or write is coded. This can only be accessed if the process of enumeration is finished. Both of them can be suspended by the ping_slv if no ping command was sent within 32 frames.

Listing 4.23: RPW-FSM

```
1  always_ff @(negedge sdw_data_clk or negedge sdw_reset_ni) begin
2    if(~sdw_reset_ni) begin
3      next_ping_cnt <= 5'b00000;
4      ping_slv      <= 1'b1;
5    end else begin
6      if (read_ping_data_o) begin
7        next_ping_cnt <= 5'b00000;
8        ping_slv <= 1'b0;
9      end else
10       if (next_ping_cnt == 5'b00111) begin
11       ping_slv <= 1'b1;
12     end else begin
13       ping_slv <= 1'b0;
14       if (cnt_reset_i) begin
15         next_ping_cnt <= next_ping_cnt + 1'b1;
16       end
17     end
18   end
19 end
20
21 always_ff @(negedge sdw_data_clk or negedge sdw_reset_ni) begin
22   if(~sdw_reset_ni) begin
23     state <= PING;
24   end else begin
25     if (cnt_reset_i) begin
26       state <= next_state;
27     end else begin
28       state <= state;
29     end
30   end
31 end
32
33 always_comb begin
34   case (state)
35     PING:  begin
36              reg_cdc_rw_ack_o  = 1'b0;
37              enum_fsm_ack_o    = 1'b0;
38              read_ping_data_o  = 1'b1;
39              ctrl_word_prw_o   = 3'b000;
```

```
40              if (~sdw_por_start_i & (reg_cdc_rw_req_i | enum_fsm_req_i)) begin
41                next_state = READ_WRITE;
42              end else begin
43                next_state = PING;
44              end
45            end
46      READ_WRITE: begin
47                    if (enum_fsm_req_i)begin
48                      reg_cdc_rw_ack_o  = 1'b0;
49                      enum_fsm_ack_o    = 1'b1;
50                      read_ping_data_o  = 1'b0;
51                      ctrl_word_prw_o   = 3'b010;
52                      if (ping_slv) begin
53                        next_state = PING;
54                      end else begin
55                        next_state = READ_WRITE;
56                      end
57                    end else if (reg_cdc_rw_req_i && enum_fsm_done_i) begin
58                      reg_cdc_rw_ack_o  = 1'b1;
59                      enum_fsm_ack_o    = 1'b0;
60                      read_ping_data_o  = 1'b0;
61                      if (ping_slv) begin
62                        next_state = PING;
63                      end else begin
64                        reg_cdc_rw_ack_o = 1'b1;
65                        next_state       = PING;
66                      end
67                    end else begin
68                      next_state = PING;
69                    end
70                  end
71      default: begin
72                 ctrl_word_prw_o = 3'b000;
73                 next_state      = PING;
74               end
75    endcase
76  end
```

Enumeration-FSM

The enumeration process is turned off by default. Incoming slv_stat_zero unlike zero (line 5) enables the operation. As long as the slave_oo status is not zero, the enumeration process will stay in ON state. Conditional operator in line 26 delivers the information to the RPW-FSM 4.2.4. Section 28-39 is responsible for tracking the actual position in the enumeration process. Seven commands have to be fulfilled for a correct finalization. next_enum_step_trig_i triggeres a jump to the next position. A check if the command was acknowledged or not has already been done.

In total, 11 slaves could be enumerated. As we know that only one will be in use, this was reduced to avoid errors by a wrong device number assignment. Rows 42-50 only emit 0001 for numbering the slave. After accepting the device number by the slave a check, whether there are more slaves attached, is done in line 60. Meeting required conditions the signal enum_fsm_done_ff is set to a high level enabling further read or write commands in the RPW-FSM 4.2.4.

Two combinatorials at the lower part are responsible for assignment of input slave data to provided registers and sending the valid op-codes, including addresses for register handling. The six 8-bit words read from the slave containing basic information are delivered in reg_enum_read_data_i. Lines 97-102 show the allocation to corresponding register sets. Starting at 113, the address related to the registers to be read are allocated depending on the enumeration position. Further, the op-code read is delivered. Last command for finalization of the process is a write to slaves register 0046 (lines 142-146). Attached is the device number for naming (enum_write_data_o).

Listing 4.24: Enumeration-FSM

```
always_comb begin
  case(state)
        OFF: begin
               enum_stat = 1'b0;
               if (slv_stat_zero == 2'b01 | slv_stat_zero == 2'b10 | slv_stat_zero == 2'b11) begin
                 next_state <= ON;
               end else begin
                 next_state <= OFF;
               end
             end
         ON: begin
               enum_stat = 1'b1;
               if (slv_stat_zero == 2'b00 ) begin
                 next_state <= OFF;
               end else begin
                 next_state <= ON;
               end
             end
    default: begin
               next_state <= OFF;
               enum_stat  <= 1'b1;
             end
  endcase
end

assign enum_fsm_req_o = slv_ctrl_word_slv_stat_00_i == 2'b00 ? 1'b0 : enum_stat;

always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if(~sdw_reset_ni) begin
    enum_pos  <= 3'b000;
  end else begin
    if (next_enum_step_trig_i) begin
      enum_pos  <= enum_pos + 1'b1;
    end else begin
      if ( enum_pos == 3'b111 && next_enum_step_trig_i) begin
        enum_pos  <= 3'b000;
      end
    end
  end
end

always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if(~sdw_reset_ni) begin
    enum_write_device_num_next  <= 4'b0001;
  end else begin
    if (next_slv_id_i) begin
      enum_write_device_num_next  <= enum_write_device_num_next;
    end
  end
end

always_ff @(posedge sdw_data_clk or negedge sdw_reset_ni) begin
  if(~sdw_reset_ni) begin
    enum_fsm_done_ff <= 1'b0;
  end else begin
    if ((enum_write_device_num_next > 4'b0001) && ~state) begin
      enum_fsm_done_ff <= 1'b1;
    end else begin
      if (slv_ctrl_word_slv_stat_01_i[0] && ~slv_ctrl_word_slv_stat_00_i[0]) begin
        enum_fsm_done_ff <= 1'b1;
      end
    end
  end
end

assign enum_fsm_done_o = enum_fsm_done_ff;

always_comb begin
  if (~sdw_reset_ni) begin
    slv1_devid0_vers_d     = 8'h00;
    slv1_devid1_manido_d   = 8'h00;
    slv1_devid2_manid1_d   = 8'h00;
    slv1_devid3_partnum0_d = 8'h00;
    slv1_devid4_partnum1_d = 8'h00;
    slv1_devid5_class_d    = 8'h00;
    enum_irq = 1'b0;
    end else
      case (enum_pos)
                3'b001: begin
                          slv1_devid0_vers_d     = reg_enum_read_data_i;
                          enum_irq = 1'b0;
                        end
                3'b010: begin
                          slv1_devid1_manido_d   = reg_enum_read_data_i;
                          enum_irq = 1'b0;
                        end
                3'b011: begin
                          slv1_devid2_manid1_d   = reg_enum_read_data_i;
                          enum_irq = 1'b0;
                        end
                3'b100: begin
                          slv1_devid3_partnum0_d = reg_enum_read_data_i;
                          enum_irq = 1'b0;
```

```
94                      end
95                  3'b101: begin
96                            slv1_devid4_partnum1_d = reg_enum_read_data_i;
97                            enum_irq = 1'b0;
98                      end
99                  3'b110: begin
100                           slv1_devid5_class_d    = reg_enum_read_data_i;
101                           enum_irq = 1'b0;
102                     end
103                 default:begin
104                           enum_irq = 1'b1;
105                     end
106                 endcase
107 end
108
109 always_comb begin
110   if (~sdw_reset_ni) begin
111     enum_write_data_o = 8'h00;
112   end else
113     case (enum_pos)
114             3'b000: begin
115                       enum_rw_addr_o  = 16'h0000;
116                       enum_rw_o       = 3'b000;
117                     end
118             3'b001: begin
119                       enum_rw_addr_o  = 16'h0050;
120                       enum_rw_o       = 3'b010;
121                     end
122             3'b010: begin
123                       enum_rw_addr_o  = 16'h0051;
124                       enum_rw_o       = 3'b010;
125                     end
126             3'b011: begin
127                       enum_rw_addr_o  = 16'h0052;
128                       enum_rw_o       = 3'b010;
129                     end
130             3'b100: begin
131                       enum_rw_addr_o  = 16'h0053;
132                       enum_rw_o       = 3'b010;
133                     end
134             3'b101: begin
135                       enum_rw_addr_o  = 16'h0054;
136                       enum_rw_o       = 3'b010;
137                     end
138             3'b110: begin
139                       enum_rw_addr_o  = 16'h0055;
140                       enum_rw_o       = 3'b010;
141                     end
142             3'b111: begin
143                       enum_rw_addr_o    = 16'h0046;
144                       enum_rw_o         = 3'b011;
145                       enum_write_data_o = {4'b0000,enum_write_device_num_next};
146                     end
147     endcase
148 end
149
150 assign enum_pos_o = enum_pos;
```

Data Slot Enable

Each bit in a frame has to be driven by the controller or slave. Responsibilities to whom bits are related is decided according to the actual opcode. An example is given again for a ping request in the listing below. The result is a sequence of zeroes and ones to be converted into DDR, either enabling the controller signal output or rejecting it. If this is not properly handled, it's not comprehensible which device has to switch the level and which sent a bit.

Listing 4.25: Enable

```
1  always_comb begin
2    case (ctrl_word_prw)
3      3'b000: begin
4                ping_en           =  1'h0;
5                opcode_en         =  3'hf;
6                device_address_en =  4'hf;
7                reg_adress_en     = 16'h0000;
8                stat_phy_sync_en  =  9'hfff;
9                read_write_reg_en =  8'h00;
10               dyn_sync_par_en   =  5'hff;
11               nak_ack_en        =  2'h0;
12             end
13       ...
```

Due to DDR a special consideration had to made. This is captured in another section 4.3.2.

4.2.5 PHY

In PHY module a encoding following the NRZI principle is realized. Combined to that, the double data rate had to implemented. Both are important for reading information from the transmission line or writing to the slave. Calculation of even or odd parity is added here to facilitate a fast result generation. This is due to the fact that the protocol allows a maximum of one clock period using the 48x2 frame for this operation.

NRZI

Audio and control data is passed between framer and phy module on two connections per direction. Input to the ODDR are two signals created by the NRZI implementation (sdw_ddr_pipe_a_o and sdw_ddr_pipe_b_o). Following, sdw_phy_sig_o_eval_a is explained. The other is self-descriptive as the same method is applied. For both, a combinatorial decision is made based on three signals combined in a evaluation signal sdw_phy_sig_o_eval_a and sdw_phy_sig_o_eval_b (lines 2-3 in the listing below).

- sdw_phy_data_o_bit_one: This is the framer data passed trough to the output
- sdw_phy_o: Output of the ODDR register is captured in this signal
- sdw_pad_data: Is the actual level of the SoundWire data connection

Special behaviour of the modified NRZI encoding is based on the line switch leading to a transmitted one. Remaining on the same level, either high or low is related to zero transmission. If a zero is applied to the evaluation for output via sdw_phy_data_o_bit_one and sdw_phy_o, sdw_pad_data are low and additionally this leads to a low level sdw_ddr_pipe_a_o (lines 8-10). As mentioned before zero transmission has to take over the transmission line level. Thus, a high SDW_Data level requires a one for sdw_ddr_pipe_a_o as shown in rows 11-13. The next case is, in turn, an intended zero transmission combined with a high DDR output whereby the physical SDW_Data is low. Possible occurrence can be due to changes of the data line drivers. This is necessary to track the actual line behaviour. Consequential as from line 17 to 19 this is repeated. Case 3'b100 is the most primitive one where the transmission level has to be changed from zero to one for delivering a one to the slave. The lower three are flipped compared to case 2-4 as the one transmission is followed by the same behaviour of ODDR output and sdw_pad_data.

Listing 4.26: NRZI Implementation

```
1  always_comb begin
2    sdw_phy_sig_o_eval_a ={sdw_phy_data_o_bit_one, sdw_phy_o, sdw_pad_data};
3    sdw_phy_sig_o_eval_b ={sdw_phy_data_o_bit_two, sdw_phy_o, sdw_pad_data};
4  end
5
6  always_comb begin
7    case(sdw_phy_sig_o_eval_a)
8      3'b000: begin
9                sdw_ddr_pipe_a_o = 1'b0;
10             end
11     3'b001: begin
12               sdw_ddr_pipe_a_o = 1'b1;
13             end
14     3'b010: begin
15               sdw_ddr_pipe_a_o = 1'b0;
16             end
17     3'b011: begin
18               sdw_ddr_pipe_a_o = 1'b1;
19             end
20     3'b100: begin
21               sdw_ddr_pipe_a_o = 1'b1;
22             end
23     3'b101: begin
24               sdw_ddr_pipe_a_o = 1'b0;
25             end
26     3'b110: begin
27               sdw_ddr_pipe_a_o = 1'b1;
28             end
29     3'b111: begin
30               sdw_ddr_pipe_a_o = 1'b0;
31             end
32   endcase
33 end
```

Parity

Even or odd parity has to be calculated and set to the output within one clock period as the related window starts after the last dynamic synchronization bit. Due to predictable LFSR synchronization values, this can be achieved although the ODDR requires one cycle for setting the output. Again, an evaluation signal is implemented including the following.

- sdw_parity_ff: This is related to the calculated parity value up to bit 43 (ctrl_word_dyn_sync[1]) of the Frame
- sdw_pad_data_ff: Corresponds to actual SoundWire pad data
- ctrl_word_dyn_sync_i: Comprises the last dynamic synchronization bit

Listing 4.27 shows three proceedings out of the eight cases. The first one includes an even parity calculated over the frame by sdw_parity_ff. As sdw_pad_data_ff and ctrl_word_dyn_sync_i are low no further switch has to be applied for the new parity signal. Case two is different due to a dynamic synchronization pattern even parity is incremented to odd. Change of pad data occurs in the last displayed evaluation case. Again this leads to a switch from even to odd parity.

Listing 4.27: Parity Implementation

```
1  assign parity_eval_sig = {sdw_parity_ff,sdw_pad_data_ff,ctrl_word_dyn_sync_i};
2
3  always_comb begin
4    case(parity_eval_sig)
5      3'b000: begin
6                sdw_parity_new = 1'b0;
7              end
8      3'b001: begin
9                sdw_parity_new = 1'b1;
10             end
11     3'b010: begin
12               sdw_parity_new = 1'b1;
13             end
14 ...
```

4.2.6 Generated Modules

Previous modules have all been self implemented. Additional ones like APB interface including all registers, audio data FIFO and the 12.288MHz clock were generated by the help of different tools.

APB Interface

APB interface is in charge of handling all registers. Module sdw_apb_reg is generated including a package sdw_apb_reg_p. As this interface is already present on the FPGA, there was a simple expansion by another slave, having a different address for communication, necessary. A tool for generating register APB linkage has been written using programming language Pearl. All automatic produced code can be recognized by surrounding as shown below.

Listing 4.28: Register Generation

```
1  //marker_template_start
2  //[% PROCESS template/general.template%]
3  //[% PROCESS template/apb.template%] [%
4  // SET RegList = data.sdw_reg.data;
5  // INCLUDE apb_reg_ff_reset;
6  // %]
7  //marker_template_code
```

Input to the Pearl script is an Excel table. In appendix E the used register Excel list is shown. Each register is instantiated following rules depending on values like MSB, LSB, Reset etc. set in the table. E.g. placement of values in a register require different parameters. Thus sdw_apb_reg_pkg is supplied. Code snippet in listing 4.29 displays register MCP_CTRL_ADDR on address 8'h10. On bit related to this is an enable EN and the width of it is one. As the bit is placed on the first position of the 8-bit register, no shift operation is needed. As a register write would change all bits of it this, can be bypassed using a mask (line 7 in the listing). Setting values after reset conditions is implemented by the last parameter.

Listing 4.29: Register Package

```
1  // Register MCP_CTRL
2  parameter bit [7:0] MCP_CTRL_ADDR = 8'h10;
3  // Bitfield EN
4  parameter int          MCP_CTRL_EN_WIDTH                = 1;
5  parameter int          MCP_CTRL_EN_SHFT                 = 0;
6  parameter bit [31:0]   MCP_CTRL_EN_MSK                  = (2**1)-1 << 0;
7  parameter bit          MCP_CTRL_EN_RESET                = 1'b0;
```

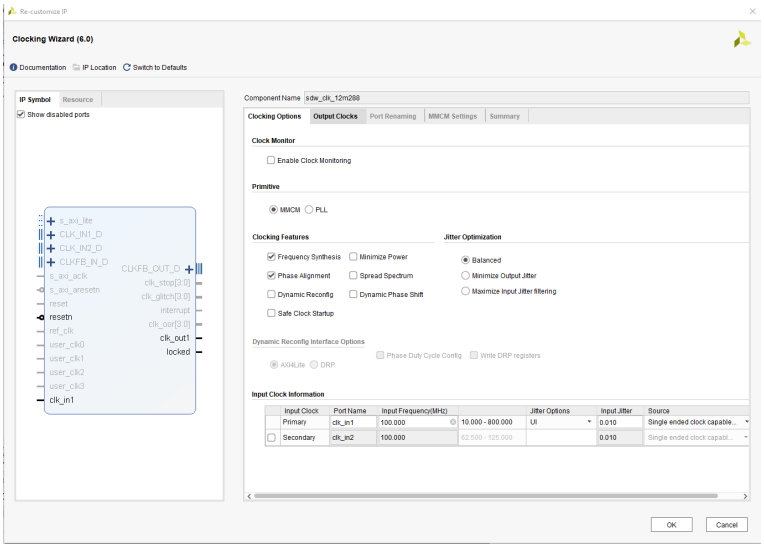


Figure 4.3: Clock Module Generation

FIFO

Vivado Software is delivered with an IP-Catalogue that includes FIFO IP's . AXI streaming interface can be chosen for related data transport in the wizard to customize the IP. Listing 4.17 shows the generated FIFO sdw_fifo_w32_d8_A. According to value w32 in the naming, this points out the width of input data up to 32-bit. Depth in turn is related to d8. Address size is needed to reproduce placing and state of data in the FIFO. By d8, the address is defined as 8-bit wide thus a 64 values deep FIFO is generated.

Accompanied by data transport connections are two signals fifo_if_empty_n and fifo_if_full_n. In case of too fast or slow data passing, those signals show extraordinary FIFO data filling.

12.288MHz Clock

The most important module for functional SoundWire operation is the generated 12.288MHz clock. Again, this IP is being delivered by Vivado's design suite. The set input clock is 100MHz. Mixed-Mode Clock Manager (MMCM) can be considered as a superset of PLL enabling phase shifting. Naming is due to the analogue PLL part and digital implementation of Delay Locked Loop (DCM) part. In another tab, the desired 12.288MHz clock can be configured. If the wizard is capable of producing this, it can be seen in a slot displaying actual generated clock. As duty cycles were already set to 50%, no change related to this had been done.

Instantiation of generated clock module according to listing 4.30 is done on the same level as previous described Leda-Control Module. Input okClk is the

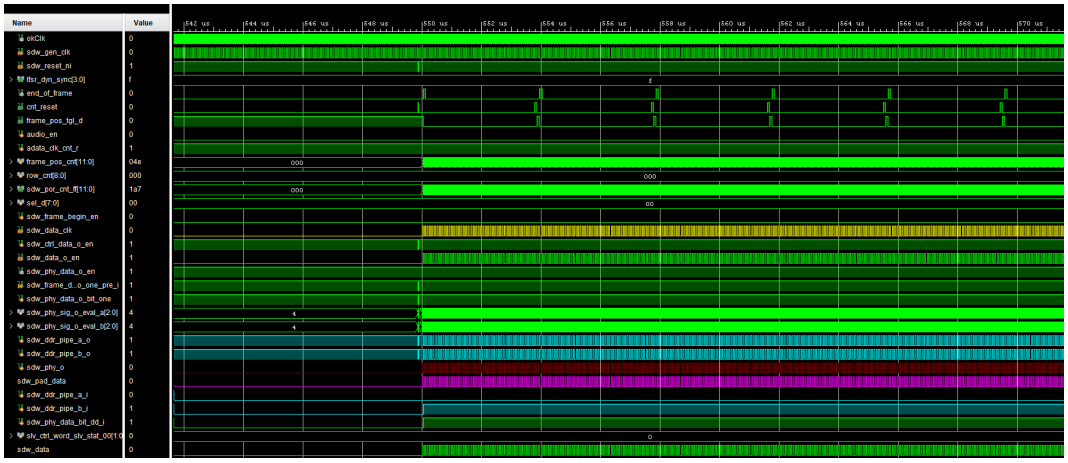


Figure 4.4: Behavioural Simulation

100MHz clock from the Opal Kelly FPGA board to be converted into desired 12.288MHz sdw_gen_clk.

Listing 4.30: Clock Instantiation

```
1 // SDW CLOCK-------------------------------------------------------------------
2 sdw_clk_12m288 u_sdw_clk_12m288 (
3   .clk_out1   (sdw_gen_clk), // O: SDW 12,288 MHz CLK
4   .resetn     (por_reset_n), // I: Reset
5   .locked     (),            // indicates stable output clk
6   .clk_in1    (okClk)        // Input Clock
7 );
```

4.3 Testing

Error detection and improvement of implemented SoundWire controller follows a sequence beginning with a behavioural simulation. In general, all signal edges will be perfect in time thus no statement about real behaviour can be made at this point. Setting delays to simulate special behaving or standard detention of units is possible for all pins in the provided constraints file. Specifically, clock signals are configured there. Figure 4.4 shows an example of simulation results. On the left side, signals concerning desired test cases are displayed. okCLK is the first signal in the picture which is used to generate the SoundWire clock responsible for all further signals along with Leda's 41.472MHz clock. The purple line on the lower side is the controller input and output SDW_DATA signal. Zooming in to single edges can be done for exact treatment.

Processing a behavioural simulation of a SoundWire interface requires a controller and a slave as both are driving different bit slots of the protocol. Consequently a test-slave had to be implemented discussed in the following section.

4.3.1 Test Slave

ACK or NAK is a small selection of required responses to test the SDW protocol. As the final design on the FPGA does not include the same easier implementation regarding the variables can be done. Nevertheless, proportional a huge amount of the test-slave is copy and paste of previous implemented controller PHY and Framer module. Input and output related DDR transmission is adopted here. Tiny configuration changes are made regarding the output enable signal due to different bit responsibility. The automated response to the different opcodes read, ping and write is shown in listing 4.31.

Simulation of the controller enumeration process requires a slave answer to register read accesses. Starting by line 19 in the listing, a response to such a read request is given. The standard reaction due to pings are shown from 8 to 18. Depending on enumeration status, slave_stat_00 and slave_stat_01 are treated differently.

Listing 4.31: Test Slave Implemenation

```
1  always_comb begin
2    if (~sdw_reset_ni) begin
3      ctrl_word_slv_stat_00 = 2'b00;
4      sample_interval_test  = 8'h00;
5      ctrl_word_nak = 1'b0;
6    end else begin
7    case (slv_ctrl_word_opcode)
8      3'b000: begin
9                if (ctrl_word_dev_addr != 4'h0) begin
10                 ctrl_word_slv_stat_00 = 2'b00;
11                 ctrl_word_slv_stat_01 = 2'b01;
12                 ctrl_word_nak = 1'b0;
13               end else begin
14                 ctrl_word_slv_stat_00 = 2'b11;
15                 ctrl_word_slv_stat_01 = 2'b00;
16                 ctrl_word_nak         = 1'b0;
17               end
18             end
19     3'b010: begin
20               if (slv_ctrl_word_reg_addr == 16'h0050) begin
21                 ctrl_word_slv_stat_03 = 2'b00;
22                 ctrl_word_slv_stat_02 = 2'b10;
23                 ctrl_word_slv_stat_01 = 2'b00;
24                 ctrl_word_slv_stat_00 = 2'b01;
25               ...
```

Together, test-slave implementation is required to check basic functionality but no further SoundWire operations were done here.

4.3.2 Synthesis and Implementation

Synthesization of System Verilog code converts RTL-code into the Gate-level net-lists whereas the implementation takes the net-lists and checks placement, routing and possibilities of optimization in respect to FPGA design.

A single error popped up when synthesis was tested due to wrong reset condition of a flip-flop. One has to ensure all flip-flops are reset by the same sdw_reset_ni signal.

However, running the implementation was associated with more changes to be done. Most relevant change was due to the ODDR. At the beginning this was not only used for the actual output signal, but also the enable signal in the

Framer module was created using a ODDR. As those specific blocks can only be accessed at latest point of pin description, an instantiation within the Framer module was impossible. Despite dis-functional FPGA structure regarding the enable ODDR. It was taken as model to implement DDR enable signal as shown in following listing 4.32. Two FF's are necessary to comply a consistent enable in each case of the signal edge. Lines 1 to 16 are related to the flip-flop in charge of falling edge whereas the other from rows 18 to 30 is enabling the audio signal on rising edge. An inverted clock signal can cause confusion here.

Last in the list is putting together both of the signals generating a DDR enable.

Listing 4.32: DDR Enable generation

```
1  always_ff @(posedge ddr_en_inv_clk or negedge sdw_reset_ni) begin
2    if (~sdw_reset_ni) begin
3      q0_out <= 1'b0;
4      qd2_posedge_int <= 1'b0;
5    end else begin
6      if (sdw_reset_ni == 1'b1 && s_in == 1'b1) begin
7        q0_out <= 1'b1;
8        qd2_posedge_int <= 1'b1;
9      end else begin
10       if (ce_in == 1'b1 && sdw_reset_ni == 1'b1 && s_in == 1'b0) begin
11         q0_out <= sdw_ctrl_data_o_en;
12         qd2_posedge_int <= sdw_audio_data_o_en;
13       end
14     end
15   end
16 end
17
18 always_ff @(negedge ddr_en_inv_clk or negedge sdw_reset_ni) begin
19   if (~sdw_reset_ni) begin
20     q1_out <= 1'b0;
21   end else begin
22     if (sdw_reset_ni == 1'b1 && s_in == 1'b1) begin
23       q1_out <= 1'b1;
24     end else begin
25       if (ce_in == 1'b1 && sdw_reset_ni == 1'b1 && s_in == 1'b0) begin
26         q1_out <= sdw_audio_data_o_en;
27       end
28     end
29   end
30 end
31
32 assign sdw_data_o_en =  ddr_en_inv_clk ?  q0_out : q1_out;
```

After finishing the implementation-run, transferring the code via a generated bitstream to the FPGA is done by the SDK-FrontPanel and testing real behaviour including the MAX98374 is possible.

4.3.3 Implementation Testing Setup

The board setup used for testing can be seen in figure 4.5. On the left side, the speaker is connected to outputs of MAX98374. SoundWire is then the link between digital amp and the FPGA. As previously mentioned, a third wire is used to avoid floating ground potential. Blue board is used after functional SDW setup to transfer an I²S audio signal from the jack socket. Power supply for all boards and the USB Laptop connection used to set registers are not shown for a clear sight.

Two problems caused major circumstances during testing phase. The first one detected has been a huge ringing 4.3.4 on data an clock wire. The other challenge was caused by a too little data signal to clock delay 4.3.5 causing unstable sampling of data.

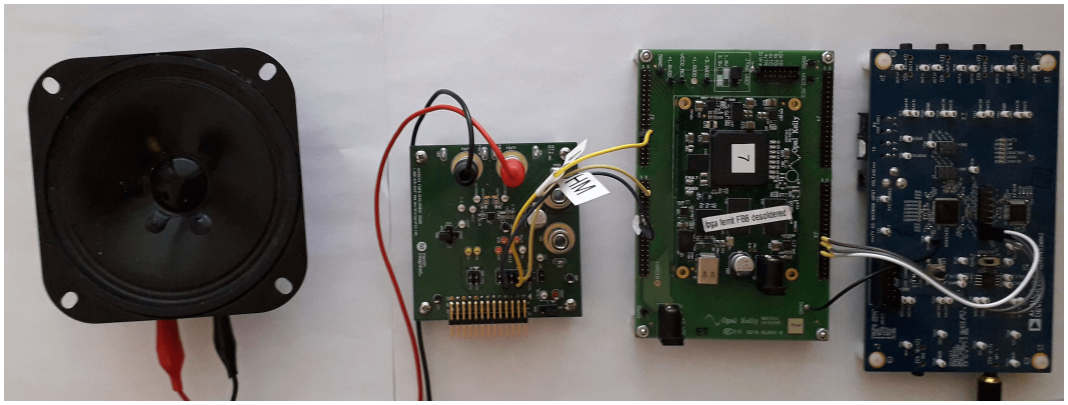


Figure 4.5: Implementation Testing Setup

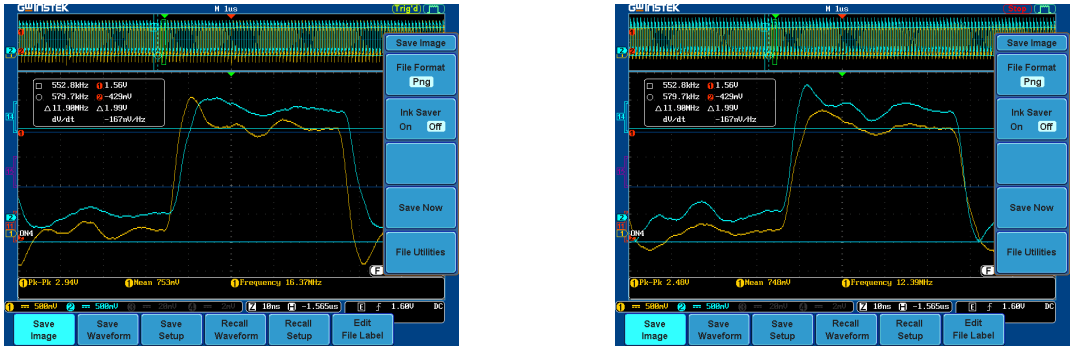
Figure 4.6: Ringing on SoundWire connection

4.3.4 Ringing

Ringing is related to the overshooting of signals. A rapid level change does not achieve desired state at the first moment, but overshoots it depending on wire capacitance as well as inductance. In case 4.6, a oscillation of the SDW Clock voltage is shown in blue. The yellow signal displays the data line after applying a 100Ω resistor in between. The value was chosen by an iteration including 50Ω and 200Ω resistance. An acceptable amount of reduced ringing and little slope change using the medium version emphasise usage of this. Thus, both connections were expanded by a 100Ω resistor.

4.3.5 Signal Delay

Taking a look at the signals in figure 4.7 b), one recognizes almost no time delay between data signal (yellow) and clock signal (blue). This little difference



(a) Signals including a delay

(b) Signals without delay

Figure 4.7: SoundWire Signal Delay

implies a limited time for a correct decision threshold sampling the data. Either too fast or slow data acquisition can occur, delivering wrong data.

The first attempt to resolve this was again done including resistors, leading to further problems due to a too flat slope. The idea to redirect the data signal back into the FPGA and out again brought success without a significant change of signal behaviour. The difference of a few nano seconds on a signal period of about 80ns enabled the proper functionality of the protocol. Redirection has been included into the PHY module as displayed in the listing. The output of ODDR is sdw_phy_o. This signal is led by sdw_pad_data_del_o to a FPGA pin and on a pin besides returned by sdw_pad_data_del_i. Actual SoundWire data in-out line is the sdw_pad_data driven including the enable signal sdw_phy_data_o_en

Listing 4.33: Signal Delay

```
1  assign sdw_pad_data       = sdw_phy_data_o_en     ? sdw_pad_data_del_i : 1'bz;
2  assign sdw_pad_data_del_o = sdw_phy_o;
```

Solving these issues enabled a successful protocol transport. Thus, enumeration could be done including the finishing device ID naming. Furthermore, register read and write operations were handled properly.

4.4 Results

By implementation of the interface, an effective reduction of wiring was shown. Although a lot of possible functionality had been omitted, a successful exchange of data was possible. Nevertheless, the goal to control the MAX98374 had been achieved.

Due to measurement restrictions, an actual energy consumption of the SoundWire module related to the extended FPGA expense couldn't be shown. For the complete setup, depending on turned on parts, between 3mA in idle state up to 8mA, driving additional speaker and audio I²S transmission, were drawn by the circuitry. This can not be handled as a valid statement for on-chip energy consumption .

Instead of this area expenditure and expense on flip-flops is discussed in the following being a valuable output of this implementation followed by a measurement of the audio signal.

4.4.1 Area Expenditure

Turning RTL code into actual hardware components is the main feature of the GENUS synthesis tool. It's part of CADENCE design tools and is only accessible having licences. Inputs are a timing constraints file to define required signal behaviour, especially related to the clock, and a synthesis script including library cells, effort to be exposed, input files (RTL modules) etc. Listing 4.34 shows some of instantiated hardware.

Listing 4.34: Genus Report

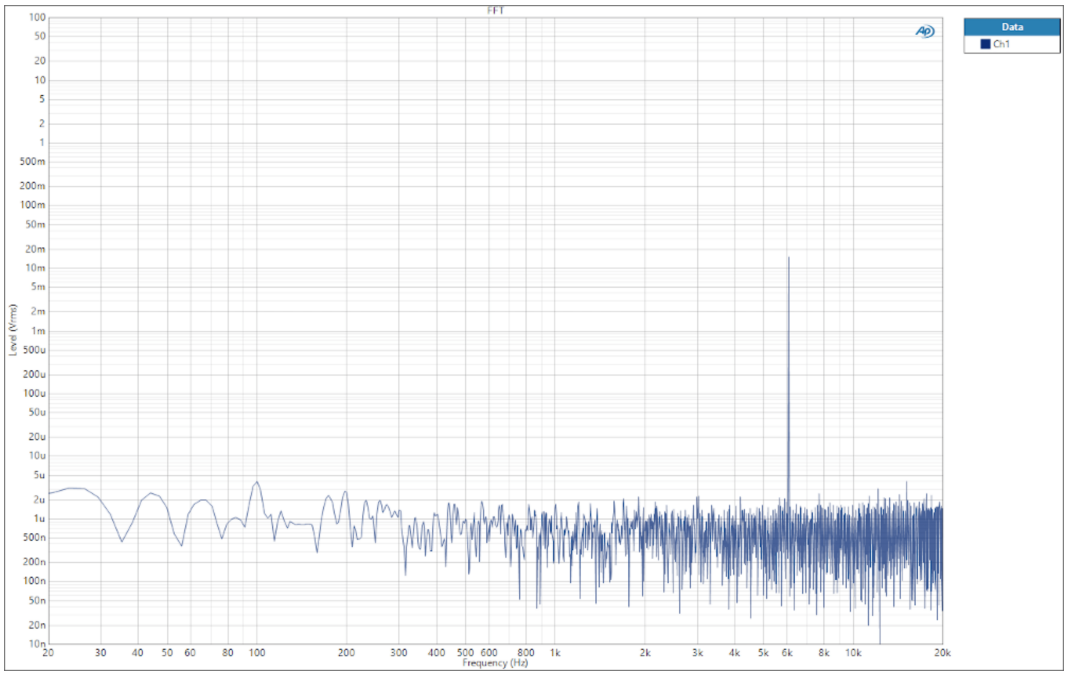
```
1                                                                    Instantiated/
2  Register                                                Inferred  Type
3  ------------------------------------------------------------------------------------------------
4  u_apb_reg/dp1_bo_blockctrl3_block_pkg_mode_ff_reg                 inferred      flip-flop asynchronous reset
5  u_apb_reg/dp1_bo_chanen_chan1_en_ff_reg                           inferred      flip-flop asynchronous reset
6  u_apb_reg/dp1_bo_hctrl_start_ff_reg[4]                            inferred      flip-flop asynchronous reset
7  ...
```

Overall, 1481 FF's have been implemented causing an area expenditure of about 0.15 square millimetres. Key result picked out of this file is a majority of flip-flops related to the generated FIFO register. 1024 can be assigned to this, opening up possibilities for improvement. As stated before, only 16 bit of audio data will be used. Thus, a change from 32-bit FIFO to 16-bit can be done. Furthermore, the address-width thus depth of the register can be minimized by a lot. As a view at least the half of related flip-flops can be obviated decreasing in turn the area expense.

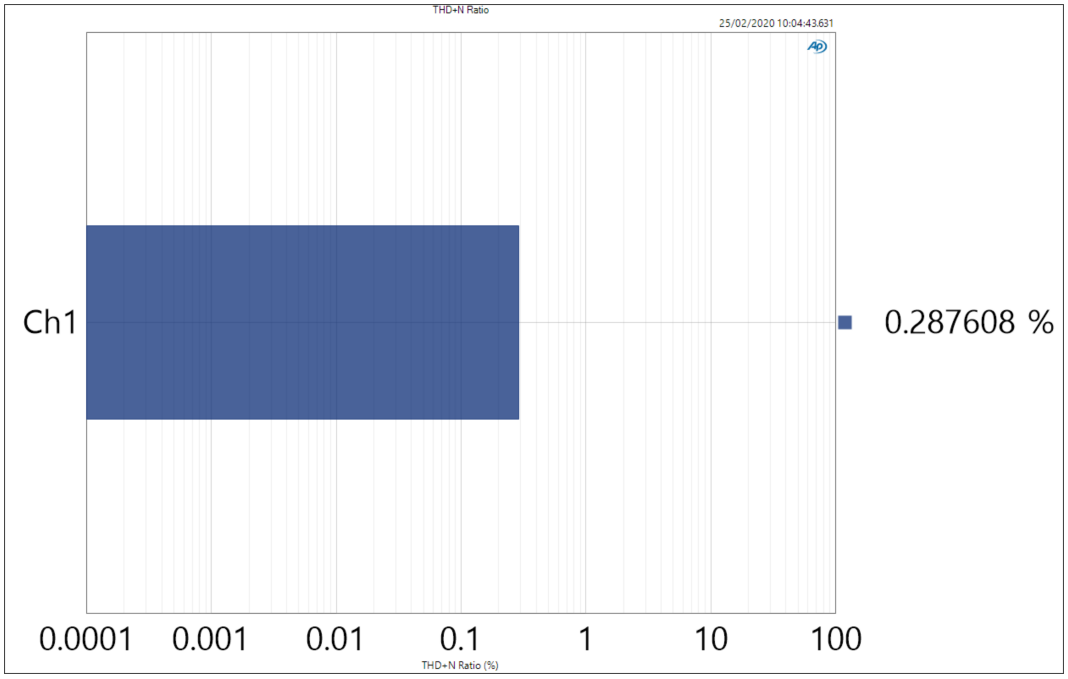
Neglecting the FIFO register, 457 flip-flops remain as the basis of the SoundWire interface. In relation this is about one third of the total amount of flip-flops as well as the area (about 0.05 square millilitres). Optimizing the register for audio input will lead to a successful improvement of the implementation and expenditure. Related to Leda's digital core area, the generated module consumes about 5.4%. Reducing to one third, neglecting the FIFO this scales down to 1.8%. In comparison to integrated I²S and I²C interfaces (4.2%), this is less area. Nevertheless, one has to be aware that SoundWire has been implemented in a reduced functionality.

4.4.2 Audio Signal Measurement

For audio measurement, an Audio Precision work station was available in the laboratory. MAX98374's tone-generator is able to emit audio sine signals in fractions of sampling rate. As a division of 48kHz by a set fraction-rate register of 8 leads to 6kHz, this can be seen in the Fast Fourier Transformation (FFT) of figure 4.8. A THD+N ratio of 0.287% has been computed as an acceptable result.



(a) FFT



(b) THD+N

Figure 4.8: Audio Precision Measurement

5 Conclusion

An investigation of different improvement possibilities for an MEMS driver ASIC was captured through the work of this thesis. Reducing energy consumption, wiring or area-count were desired results. Overall, two sections are delivered covering the behaviour of audio data encoding and a possible interface for reducing wiring.

The encoding of audio data showed to be of little influence on the early stage of investigation. Although statistically a reduced toggle rate could be reached, this moved in a very limited area, strongly depending on applied signal-form and level. Four encoding mechanisms were implemented, whereas only three were applied to either serial or parallel wiring systems. A self designed pattern encoding showed switching reduction for serial data transmission using a sine plus noise signal. Full-scale related signals could be reduced in toggling by this. For serial encodings, no other option was relevant due to a disproportional expense. Regarding parallel data wiring, OEFNSC promised to be an effective application which couldn't be underpinned. The reason for this lies in a limited useful operational capability. For an extremely optimized ASIC the energy consumption of capacitive loads, in-between wires, have to be taken into account (which is not corresponding to this case of application). For noise transmission a switching reduction was detected. Overall, applying encoding techniques for audio data is not recommended in respect to the methods tested.

Success in optimizing the ASIC was found by an investigation of available interfaces for audio and control data transmission. SoundWire is a novel protocol developed by the MIPI Alliance aware of transmitting both data types. Thus, a reduction in wiring can be achieved. Previously, 5 wire connections due to the I²S and I²C could be reduced to a two line connection. As the market doesn't display a huge increase of applied SoundWire interfaces, this is a draw back right now and could change at any moment. As there is a need for a subwoofer, regarding some applications of the MEMS speakers, another interface could be needed. Such a case decreases an additional high amount of wiring to a little. Implementing the interface showed a variety of configuration possibilities. A decision for a minimum implementation was made, ensuring a normal register interaction and audio transmission between an, amplifier MAX98374, slave and the controller integrated on the FPGA responsible for control setup of the Leda ASIC.

Appendix

A SoundWire Frame

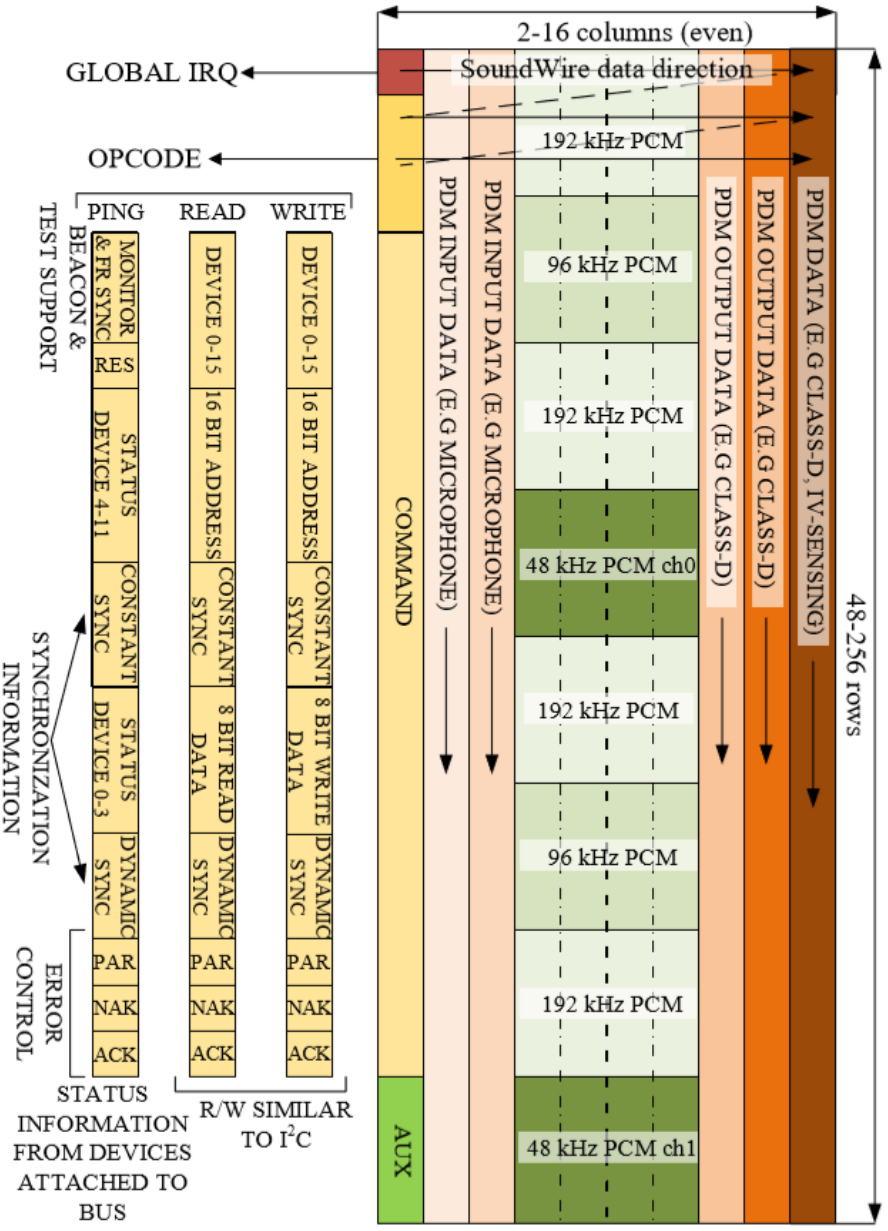


Figure .1: Soundwire Frame Structure (see Pierre-Louis Bossart, 2014)

B Encoding Tables

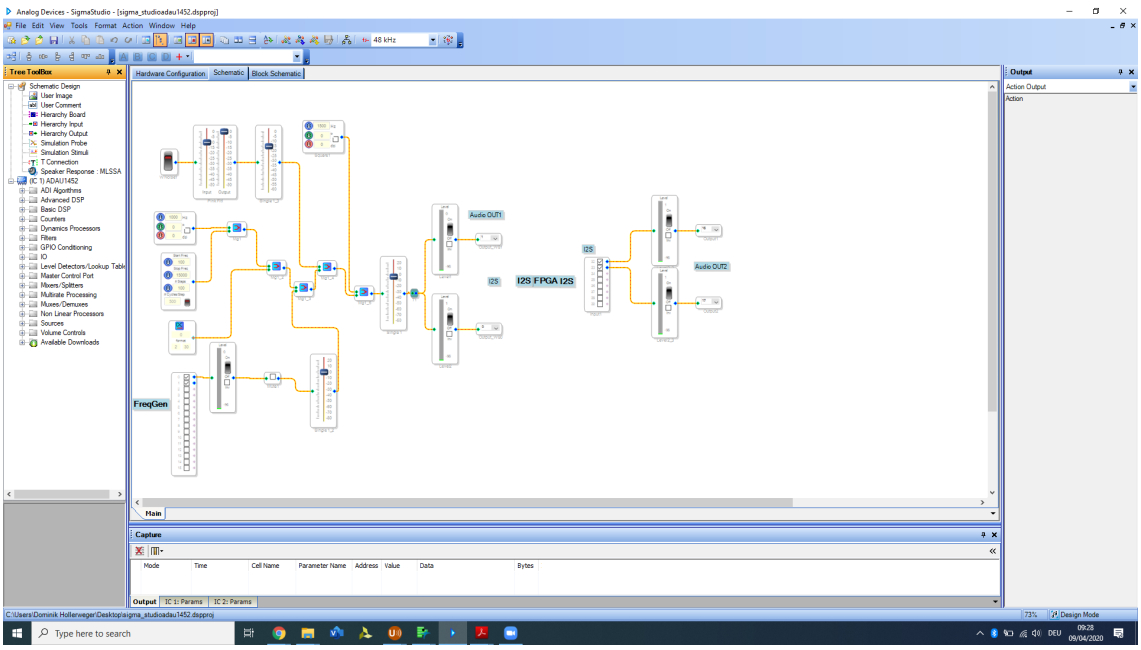
Signal Lenght=48000 Samples			Serial			Parallel		
			Self-Transitions			Self-Transitions		Self&Coupling-Transitions
			BINV	TINV	Patterns	BINV	TINV	OEFNSC
NOISE	0 dBFS	ORG	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
		Encoded	99,80%	91,21%	73,85%	80,37%	89,31%	75,16%
		Encoded inc. Overhead	106,36%	94,74%	86,48%	86,38%	95,30%	87,51%
	-20 dBFS	ORG	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
		Encoded	99,4%	90,5%	82,4%	71,8%	78,7%	70,9%
		Encoded inc. Overhead	107,66%	96,64%	100,14%	78,00%	84,90%	83,35%
Sine+Noise	0 dBFS	ORG	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
		Encoded	99,98%	94,38%	80,19%	92,78%	94,23%	89,59%
		Encoded inc. Overhead	106,97%	98,35%	94,58%	96,19%	98,99%	101,31%
	-20 dBFS	ORG	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
		Encoded	98,8%	93,3%	77,6%	97,1%	96,4%	97,1%
		Encoded inc. Overhead	106,06%	99,39%	93,94%	97,81%	97,08%	102,19%
Sweep	0 dBFS	ORG	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
		Encoded	99,99%	92,84%	74,93%	95,43%	105,38%	94,77%
		Encoded inc. Overhead	106,48%	96,27%	87,03%	97,18%	109,47%	99,66%
	-20 dBFS	ORG	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
		Encoded	99,6%	91,5%	82,9%	97,9%	110,4%	98,0%
		Encoded inc. Overhead	106,71%	96,76%	99,65%	98,52%	112,10%	99,88%

Figure .2: Swithing Activity of Encoding Methods in %

Signal Lenght=48000 Samples			Serial			Parallel		
			Self-Transitions			Self-Transitions		Self&Coupling-Transitions
			BINV tgl/s	TINV tgl/s	Patterns tgl/s	BINV tgl/s	TINV tgl/s	OEFNSC tgl/s
NOISE	0 dBFS	ORG	383446	383446	383446	383817	383820	383817
		Encoded	382690	349732	283186	308479	342785	288467
		Encoded inc. Overhead	407826	363274	331606	331540	365792	335869
	-20 dBFS	ORG	312503	312503	312503	384102	384108	384102
		Encoded	310560	282797	257621	275894	302359	272324
		Encoded inc. Overhead	336448	301995	312941	299583	326106	320135
Sine+Noise	0 dBFS	ORG	334333	334333	334333	323567	323567	323567
		Encoded	334281	315558	268115	300193	304882	289899
		Encoded inc. Overhead	357643	328802	316213	311249	320307	327807
	-20 dBFS	ORG	330000	330000	330000	273990	273990	273990
		Encoded	325991	307992	255999	265989	263994	265989
		Encoded inc. Overhead	349991	327992	309999	267989	265998	279988
Sweep	0 dBFS	ORG	372770	372770	372770	254125	254141	254125
		Encoded	372734	346078	279299	242522	267803	240824
		Encoded inc. Overhead	396943	358863	324439	246946	278205	253264
	-20 dBFS	ORG	322215	322215	322215	175160	175160	175160
		Encoded	321045	294776	266991	171395	193366	171731
		Encoded inc. Overhead	343827	311778	321103	172562	196348	174945

Figure .3: Swithing Activity of Encoding Methods in Total

C ADAU1452 Configuration



D MAX98374 Register Map

Device Register Map

The general control registers in the address space 0x2000 and above are accessible through both the SoundWire compatible and I2C interface and are used for configuring the device (except the SoundWire compatible interface). The registers in the address space 0x0040 to 0x0337 are only accessible through the SoundWire compatible interface and are used for configuring the SoundWire slave interface settings (Figure 9). Register bit fields come in several varieties that are summarized and color coded in Table 1.

Read-only bit fields (R) are used to indicate an internal device state and cannot be changed directly by the host. Writing to these registers has no effect. Read-only status bit fields are the same, however, writing a 1 to these registers clears the status while writing a 0 has no effect.

Write-only bit fields (W) are single-bit push-button controls. Writing a 1 to these bit fields performs an action (i.e., software reset, interrupt clear, etc.). Writing a 0 has no effect and readback always returns a 0.

Read/write bit fields (RW) can be both read and written by the host, and the last written value is the value returned on read back.

Reserved bit fields (indicated by a "—" in Table 2) are not used to program or control the device. When writing a register that contains reserved bit fields, always write a 0 to the reserved bit field segments.

Table 1. Register Map Bit Field Color Coding

BIT FIELD TYPE	FUNCTIONAL DESCRIPTION
Read-Only Bit Field	A write to a read-only bit field has no effect.
Read-Only Status Bit Field (Clear on Write)	Write a 1 to clear a read-only status bit field. A write of 0 has no effect.
Write-Only Bit Field	Write a 1 to activate the function of a write-only bit field. A write of 0 has no effect, and readback always returns 0.
Read/Write Bit Field	Read and write commands function normally. There can be write access restrictions.
Reserved Bit Field	Always write a 0 to a reserved bit field.

Table 2. General Control Register Map

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	REGISTER NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SOFTWARE RESET REGISTER										
0x2000	0x00	Software Reset	—	—	—	—	—	—	—	RST
INTERRUPT REGISTERS										
0x2001	0x00	Interrupt Raw 1	THERMSHDN_BGN_RAW	THERMSHDN_END_RAW	THERMWARN_BGN_RAW	THERMWARN_END_RAW	BDE_L4_RAW	BDE_LEVEL_RAW	BDE_ACTIVE_BGN_RAW	BDE_ACTIVE_END_RAW
0x2002	0x00	Interrupt Raw 2	PWRUP_DONE_RAW	PWRDN_DONE_RAW	OTP_FAIL_RAW	SPK_OVC_RAW	CLKSTART_RAW	CLKSTOP_RAW	ICC_SYNC_ERR_RAW	ICC_DATA_ERR_RAW
0x2003	0x00	Interrupt Raw 3	PVDD_UVLO_SHDN_RAW	—	THERMFB_BGN_RAW	THERMFB_END_RAW	DHT_ACTIVE_BGN_RAW	DHT_ACTIVE_END_RAW	LMTR_ACTIVE_BGN_RAW	LMTR_ACTIVE_END_RAW
0x2004	0x00	Interrupt State 1	THERMSHDN_BGN_STATE	THERMSHDN_END_STATE	THERMWARN_BGN_STATE	THERMWARN_END_STATE	BDE_L4_STATE	BDE_LEVEL_STATE	BDE_ACTIVE_BGN_STATE	BDE_ACTIVE_END_STATE
0x2005	0x00	Interrupt State 2	PWRUP_DONE_STATE	PWRDN_DONE_STATE	OTP_FAIL_STATE	SPK_OVC_STATE	CLKSTART_STATE	CLKSTOP_STATE	ICC_SYNC_ERR_STATE	ICC_DATA_ERR_STATE
0x2006	0x00	Interrupt State 3	PVDD_UVLO_SHDN_STATE	—	THERMFB_BGN_STATE	THERMFB_END_STATE	DHT_ACTIVE_BGN_STATE	DHT_ACTIVE_END_STATE	LMTR_ACTIVE_BGN_STATE	LMTR_ACTIVE_END_STATE
0x2007	0x00	Interrupt Flag 1	THERMSHDN_BGN_FLAG	THERMSHDN_END_FLAG	THERMWARN_BGN_FLAG	THERMWARN_END_FLAG	BDE_L4_FLAG	BDE_LEVEL_FLAG	BDE_ACTIVE_BGN_FLAG	BDE_ACTIVE_END_FLAG
0x2008	0x00	Interrupt Flag 2	PWRUP_DONE_FLAG	PWRDN_DONE_FLAG	OTP_FAIL_FLAG	SPK_OVC_FLAG	CLKSTART_FLAG	CLKSTOP_FLAG	ICC_SYNC_ERR_FLAG	ICC_DATA_ERR_FLAG
0x2009	0x00	Interrupt Flag 3	PVDD_UVLO_SHDN_FLAG	—	THERMFB_BGN_FLAG	THERMFB_END_FLAG	DHT_ACTIVE_BGN_FLAG	DHT_ACTIVE_END_FLAG	LMTR_ACTIVE_BGN_FLAG	LMTR_ACTIVE_END_FLAG
0x200A	0x00	Interrupt Enable 1	THERMSHDN_BGN_EN	THERMSHDN_END_EN	THERMWARN_BGN_EN	THERMWARN_END_EN	BDE_L4_EN	BDE_LEVEL_EN	BDE_ACTIVE_BGN_EN	BDE_ACTIVE_END_EN
0x200B	0x00	Interrupt Enable 2	PWRUP_DONE_EN	PWRDN_DONE_EN	OTP_FAIL_EN	SPK_OVC_EN	CLKSTART_EN	CLKSTOP_EN	ICC_SYNC_ERR_EN	ICC_DATA_ERR_EN
0x200C	0x00	Interrupt Enable 3	PVDD_UVLO_SHDN_EN	—	THERMFB_BGN_EN	THERMFB_END_EN	DHT_ACTIVE_BGN_EN	DHT_ACTIVE_END_EN	LMTR_ACTIVE_BGN_EN	LMTR_ACTIVE_END_EN
0x200D	0x00	Interrupt Flag Clear 1	THERMSHDN_BGN_CLR	THERMSHDN_END_CLR	THERMWARN_BGN_CLR	THERMWARN_END_CLR	BDE_L4_CLR	BDE_LEVEL_CLR	BDE_ACTIVE_BGN_CLR	BDE_ACTIVE_END_CLR
0x200E	0x00	Interrupt Flag Clear 2	PWRUP_DONE_CLR	PWRDN_DONE_CLR	OTP_FAIL_CLR	SPK_OVC_CLR	CLKSTART_CLR	CLKSTOP_CLR	ICC_SYNC_ERR_CLR	ICC_DATA_ERR_CLR
0x200F	0x00	Interrupt Flag Clear 3	PVDD_UVLO_SHDN_CLR	—	THERMFB_BGN_CLR	THERMFB_END_CLR	DHT_ACTIVE_BGN_CLR	DHT_ACTIVE_END_CLR	LMTR_ACTIVE_BGN_CLR	LMTR_ACTIVE_END_CLR
0x2010	0x00	IRQ Bus Configuration	—	—	—	—	—	IRQ_MODE	IRQ_POL	IRQ_EN
THERMAL PROTECTION REGISTERS										
0x2014	0x10	Thermal-Warning Threshold Configuration	—	—	THERMWARN_THRESH[5:0]					
0x2015	0x27	Thermal-Shutdown Threshold Configuration	—	—	THERMSHDN_THRESH[5:0]					

Table 2. General Control Register Map (continued)

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	REGISTER NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x2016	0x01	Thermal Hysteresis Configuration	—	—	—	—	—	—	THERM_HYST[1:0]	
0x2017	0xC0	Thermal-Foldback Settings	THERMFB_HOLD[1:0]		—	—	THERMFB_RLS[1:0]		THERMFB_SLOPE[1:0]	
0x2018	0x00	Thermal-Foldback Enable	—	—	—	—	—	—	—	THERMFB_EN
PCM INTERFACE REGISTERS										
0x201E	0x55	Digital Output Pin Drive Strength Configuration	ICC_DRV[1:0]		LRCLK_DRV[1:0]		IRQ_DRV[1:0]		DOUT_DRV[1:0]	
0x2020	0xFE	PCM Interface Data Output Channel Configuration 1	PCM_TX_CH7_HIZ	PCM_TX_CH6_HIZ	PCM_TX_CH5_HIZ	PCM_TX_CH4_HIZ	PCM_TX_CH3_HIZ	PCM_TX_CH2_HIZ	PCM_TX_CH1_HIZ	PCM_TX_CH0_HIZ
0x2021	0xFF	PCM Interface Data Output Channel Configuration 2	PCM_TX_CH15_HIZ	PCM_TX_CH14_HIZ	PCM_TX_CH13_HIZ	PCM_TX_CH12_HIZ	PCM_TX_CH11_HIZ	PCM_TX_CH10_HIZ	PCM_TX_CH9_HIZ	PCM_TX_CH8_HIZ
0x2023	0x00	PCM Interface Data Output Speaker DSP Feedback Channel Source	—	—	—	—	PCM_SPK_FB_DEST[3:0]			
0x2024	0xC0	PCM Interface Data Format Configuration	PCM_DATA_WIDTH[1:0]		PCM_FORMAT[2:0]			PCM_TX_INTERLEAVE	PCM_CHANSEL	PCM_TX_EXTRA_HIZ
0x2025	0x00	Audio Interface Mode Configuration	—	—	—	—	—	—	INTERFACE_MODE[1:0]	
0x2026	0x04	PCM Interface Clock Ratio Configuration	—	—	—	PCM_BCLKEDGE	PCM_BSEL[3:0]			
0x2027	0x08	PCM Interface Sample Rate	—	—	—	—	PCM_SR[3:0]			
0x2028	0x88	Speaker Path Sample Rate	SPK_SR[3:0]				—	—	—	—
0x2029	0x00	PCM Interface Digital Mono Mixer Configuration 1	DMMIX_CFG[1:0]		—	—	DMMIX_CH0_SOURCE[3:0]			
0x202A	0x00	PCM Interface Digital Mono Mixer Configuration 2	—	—	—	—	DMMIX_CH1_SOURCE[3:0]			
0x202B	0x00	PCM Interface Data Input (DIN) Enable	—	—	—	—	—	—	—	PCM_RX_EN
0x202C	0x00	PCM Interface Data Output (DOUT) Enable	—	—	—	—	—	—	—	PCM_TX_EN

Table 2. General Control Register Map (continued)

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	REGISTER NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
ICC REGISTERS										
0x202E	0x00	ICC Receiver Enables 1	ICC_RX_CH7_EN	ICC_RX_CH6_EN	ICC_RX_CH5_EN	ICC_RX_CH4_EN	ICC_RX_CH3_EN	ICC_RX_CH2_EN	ICC_RX_CH1_EN	ICC_RX_CH0_EN
0x202F	0x00	ICC Receiver Enables 2	ICC_RX_CH15_EN	ICC_RX_CH14_EN	ICC_RX_CH13_EN	ICC_RX_CH12_EN	ICC_RX_CH11_EN	ICC_RX_CH10_EN	ICC_RX_CH9_EN	ICC_RX_CH8_EN
0x2030	0xFF	ICC Transmitter Channel Configuration 1	ICC_TX_CH7_HIZ	ICC_TX_CH6_HIZ	ICC_TX_CH5_HIZ	ICC_TX_CH4_HIZ	ICC_TX_CH3_HIZ	ICC_TX_CH2_HIZ	ICC_TX_CH1_HIZ	ICC_TX_CH0_HIZ
0x2031	0xFF	ICC Transmitter Channel Configuration 2	ICC_TX_CH15_HIZ	ICC_TX_CH14_HIZ	ICC_TX_CH13_HIZ	ICC_TX_CH12_HIZ	ICC_TX_CH11_HIZ	ICC_TX_CH10_HIZ	ICC_TX_CH9_HIZ	ICC_TX_CH8_HIZ
0x2032	0x30	ICC Data Link Configuration	ICC_DHT_CONFIG	ICC_LIM_CONFIG	ICC_BDE_CONFIG	ICC_THERM_CONFIG	ICC_DHT_LINK_EN	ICC_LIM_LINK_EN	ICC_BDE_LINK_EN	ICC_THERM_LINK_EN
0x2034	0x00	ICC Transmitter Configuration	ICC_SYNC_SLOT[3:0]				ICC_TX_DEST[3:0]			
0x2035	0x00	ICC Transmitter Enable	–	–	–	–	–	–	–	ICC_TX_EN
SOUNDWIRE SLAVE INTERFACE REGISTERS										
0x2036	0x05	SoundWire Input Clock Configuration	–	–	–	SWIRE_CLK_RATE[1:0]		SWIRE_CLK_SEL[2:0]		
SPEAKER PATH CONTROL REGISTERS										
0x203D	0x00	Speaker Path Digital Volume Control	–	SPK_VOL[6:0]						
0x203E	0x08	Speaker Path Output Level Scaling	SPK_GAIN[3:0]				SPK_GAIN_MAX[3:0]			
0x203F	0x02	Speaker Path DSP Configuration	SPK_VOL_SEL	–	SPK_INVERT	–	SPK_VOL_RMPDN_BYPASS	SPK_VOL_RMPUP_BYPASS	SPK_DITH_EN	SPK_DCBLK_EN
0x2040	0x00	Tone Generator Configuration	–	–	–	–	TONE_CONFIG[3:0]			
0x2041	0x03	Speaker Amplifier Configuration	SPK_SSM_EN	SPK_OSC_SEL	–	–	SPK_FSW_SEL	SPK_SSM_MOD_INDEX[2:0]		
0x2042	0x00	Speaker Amplifier Switching Edge Rate Configuration	–	–	–	–	–	SPK_EDGE_CTRL[1:0]		–
0x2043	0x00	Speaker Path and Speaker DSP Data Feedback Path Enables	–	–	–	–	–	–	SPK_FB_EN	SPK_EN

Table 2. General Control Register Map (continued)

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	REGISTER NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
MEASURMENT ADC REGISTERS										
0x2051	0x00	Measurement ADC Sample Rate Control	—	—	—	—	—	—	MEAS_ADC_SR[1:0]	
0x2052	0x00	Measurement ADC PVDD Channel Filter Configuration	—	—	—	MEAS_ADC_PVDD_FILT_EN	—	—	MEAS_ADC_PVDD_FILT_COEFF[1:0]	
0x2053	0x00	Measurement ADC Thermal Channel Filter Configuration	—	—	—	MEAS_ADC_TEMP_FILT_EN	—	—	MEAS_ADC_TEMP_FILT_COEFF[1:0]	
0x2054	0x00	Measurement ADC PVDD Channel Readback	MEAS_ADC_PVDD_DATA[7:0]							
0x2055	0x00	Measurement ADC Thermal Channel Readback	MEAS_ADC_THERM_DATA[7:0]							
0x2056	0x00	Measurement ADC PVDD Channel Enable	—	—	—	—	—	—	—	MEAS_ADC_PVDD_EN
BDE REGISTERS										
0x2090	0x00	BDE Level Hold Time	BDE_HLD[7:0]							
0x2091	0x00	BDE Gain Reduction Attack/ Release Rates	BDE_GAIN_ATK[3:0]				BDE_GAIN_RLS[3:0]			
0x2092	0x00	BDE Clipper Mode	—	—	—	—	—	—	—	BDE_CLIP_MODE
0x2097	0x00	BDE Level 1 Threshold	BDE_L1_VTHRESH[7:0]							
0x2098	0x00	BDE Level 2 Threshold	BDE_L2_VTHRESH[7:0]							
0x2099	0x00	BDE Level 3 Threshold	BDE_L3_VTHRESH[7:0]							
0x209A	0x00	BDE Level 4 Threshold	BDE_L4_VTHRESH[7:0]							
0x209B	0x00	BDE Level Threshold Hysteresis	BDE_VTHRESH_HYST[7:0]							
0x20A8	0x00	BDE Level 1 Limiter Configuration	—	—	—	—	BDE_L1_LIM[3:0]			
0x20A9	0x00	BDE Level 1 Clipper Configuration	—	—	BDE_L1_CLIP[5:0]					
0x20AA	0x00	BDE Level 1 Gain Reduction Configuration	—	—	BDE_L1_GAIN[5:0]					
0x20AB	0x00	BDE Level 2 Limiter Configuration	—	—	—	—	BDE_L2_LIM[3:0]			

Table 2. General Control Register Map (continued)

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	REGISTER NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x20AC	0x00	BDE Level 2 Clipper Configuration	—	—	BDE_L2_CLIP[5:0]					
0x20AD	0x00	BDE Level 2 Gain Reduction Configuration	—	—	BDE_L2_GAIN[5:0]					
0x20AE	0x00	BDE Level 3 Limiter Configuration	—	—	—	—	BDE_L3_LIM[3:0]			
0x20AF	0x00	BDE Level 3 Clipper Configuration	—	—	BDE_L3_CLIP[5:0]					
0x20B0	0x00	BDE Level 3 Gain Reduction Configuration	—	—	BDE_L3_GAIN[5:0]					
0x20B1	0x00	BDE Level 4 Limiter Configuration	—	—	—	—	BDE_L4_LIM[3:0]			
0x20B2	0x00	BDE Level 4 Clipper and State Configuration	BDE_L4_MUTE	BDE_L4_INF_HLD	BDE_L4_CLIP[5:0]					
0x20B3	0x00	BDE Level 4 Gain Reduction Configuration	—	—	BDE_L4_GAIN[5:0]					
0x20B4	0x00	BDE Level 4 Infinite Hold Clear	BDE_L4_HLD_RLS	—	—	—	—	—	—	—
0x20B5	0x00	BDE Enable	—	—	—	—	—	—	—	BDE_EN
0x20B6	0x00	BDE Current State	—	—	—	—	BDE_STATE[3:0]			
DHT REGISTERS										
0x20D1	0x01	DHT Configuration	DHT_SPK_GAIN_MIN[3:0]				DHT_VROT_PNT[3:0]			
0x20D2	0x02	DHT Attack Rate Settings	—	—	—	DHT_ATK_STEP[1:0]		DHT_ATK_RATE[2:0]		
0x20D3	0x03	DHT Release Rate Settings	—	—	—	DHT_RLS_STEP[1:0]		DHT_RLS_RATE[2:0]		
0x20D4	0x00	DHT Enable	—	—	—	—	—	—	—	DHT_EN
LIMITER REGISTERS										
0x20E0	0x00	Limiter Threshold Configuration	—	LIM_THRESH[4:0]					—	—
0x20E1	0x00	Limiter Attack and Release Rate Configuration	LIM_ATK_RATE[3:0]				LIM_RLS_RATE[3:0]			
0x20E2	0x00	Limiter Enable	—	—	—	—	—	—	—	LIM_EN
ENABLE REGISTERS										
0x20FE	0x00	Device Auto-Restart Configuration	—	—	—	—	OVC_AUTO RESTART_EN	THERM_AUTO RESTART_EN	CMON_AUTO RESTART_EN	CMON_EN
0x20FF	0x00	Global Enable	—	—	—	—	—	—	—	EN
REVISION IDENTIFICATION REGISTER										
0x21FF	0x43	Device Revision Identification Number	REV_ID[7:0]							

Table 3. SoundWire Slave Interface Register Map

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SLAVE CONTROL PORT (SCP) REGISTERS										
0x0040	0x00	SCP_IntStat_1	SCP2 Cascade	Port 3 Cascade	Port 2 Cascade	Port 1 Cascade	—	IntStat ImpDef1	IntStat Port Ready	IntStat Parity
0x0041	0x00	SCP_IntMask_1	—	—	—	—	—	IntMask ImpDef1	IntMask Bus Clash	intMask Parity
0x0042	0x00	SCP_IntStat_2	—	—	—	—	—	—	—	Port 4 Cascade
0x0044	0x00	SCP_Ctrl	ForceReset	CurrentBank	—	—	—	—	ClockStop Now	ClockStop_ NotFinished
0x0045	0x00	SCP_SystemCtrl	—	—	—	—	—	—	—	ClockStop Prepare
0x0046	0x00	SCP_DevNumber	—	—	Group_ID[1:0]		Device Number[3:0]			
0x0050	0x21	SCP_DevId_0	Device_ID[47:40]							
0x0051	0x01	SCP_DevId_1	Device ID[39:32]							
0x0052	0x9F	SCP_DevId_2	Device ID[31:24]							
0x0053	0x87	SCP_DevId_3	Device ID[23:16]							
0x0054	0x08	SCP_DevId_4	Device ID[15:8]							
0x0055	0x00	SCP_DevId_5	Device ID[7:0]							
0x0060	0x00	SCP_FrameCtrl	RowControl[4:0]					ColumnControl[2:0]		
0x0070	0x00	SCP_FrameCtrl	RowControl[4:0]					ColumnControl[2:0]		
DATA PORT 1 REGISTERS										
0x0100	0x00	DP1_IntStat	—	—	—	—	—	—	IntStat Port Ready	IntStat Test Fail
0x0101	0x00	DP1_IntMask	—	—	—	—	—	—	IntMask Port Ready	IntMask Test Fail
0x0102	0x20	DP1_PortCtrl	—	—	Port Direction	Next InvertBank	PortDataMode[1:0]		PortFlowMode[1:0]	
0x0103	0x00	DP1_BlockCtrl1	—	—	—	WordLength[4:0]				
0x0104	0x00	DP1_PrepareStatus	—	—	—	—	—	—	N-Finished Channel 2	N-Finished Channel 1
0x0105	0x00	DP1_PrepareCtrl	—	—	—	—	—	—	Prepare Channel 2	Prepare Channel 1
DATA PORT 1 - BANK 0 REGISTERS										
0x0120	0x00	DP1_ChannelEn	—	—	—	—	—	—	Enable Channel 2	Enable Channel 1
0x0122	0x00	DP1_SampleCtrl1	SampleIntervalLow[7:0]							
0x0123	0x00	DP1_SampleCtrl2	SampleIntervalHigh[7:0]							
0x0124	0x00	DP1_OffsetCtrl1	Offset1[7:0]							
0x0125	0x00	DP1_OffsetCtrl2	Offset2[7:0]							
0x0126	0x00	DP1_HCtrl	HStart[3:0]				HStop[3:0]			
0x0127	0x00	DP1_BlockCtrl3	—	—	—	—	—	—	—	BlockPacking Mode

Table 3. SoundWire Slave Interface Register Map (continued)

REGISTER PROPERTIES			REGISTER CONTENTS							
ADDR	POR	NAME	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
DATA PORT 1 - BANK 1 REGISTERS										
0x0130	0x00	DP1_ChannelEn	—	—	—	—	—	—	Enable Channel 2	Enable Channel 1
0x0132	0x00	DP1_SampleCtrl1	SampleIntervalLow[7:0]							
0x0133	0x00	DP1_SampleCtrl2	SampleIntervalHigh[7:0]							
0x0134	0x00	DP1_OffsetCtrl1	Offset1[7:0]							
0x0135	0x00	DP1_OffsetCtrl2	Offset2[7:0]							
0x0136	0x00	DP1_HCtrl	HStart[3:0]				HStop[3:0]			
0x0137	0x00	DP1_BlockCtrl3	—	—	—	—	—	—	—	BlockPacking Mode
DATA PORT 3 REGISTERS										
0x0300	0x00	DP3_IntStat	—	—	—	—	—	—	IntStat Port Ready	IntStat Test Fail
0x0301	0x00	DP3_IntMask	—	—	—	—	—	—	IntMask Port Ready	IntMask Test Fail
0x0302	0x20	DP3_PortCtrl	—	—	Port Direction	Next InvertBank	PortDataMode[1:0]		PortFlowMode[1:0]	
0x0303	0x00	DP3_BlockCtrl1	—	—	—	WordLength[4:0]				
0x0304	0x00	DP3_PrepareStatus	—	—	—	—	—	—	N-Finished Channel 2	N-Finished Channel 1
0x0305	0x00	DP3_PrepareCtrl	—	—	—	—	—	—	Prepare Channel 2	Prepare Channel 1
DATA PORT 3 - BANK 0 REGISTERS										
0x0320	0x00	DP3_ChannelEn	—	—	—	—	—	—	Enable Channel 2	Enable Channel 1
0x0322	0x00	DP3_SampleCtrl1	SampleIntervalLow[7:0]							
0x0323	0x00	DP3_SampleCtrl2	SampleIntervalHigh[7:0]							
0x0324	0x00	DP3_OffsetCtrl1	Offset1[7:0]							
0x0325	0x00	DP3_OffsetCtrl2	Offset2[7:0]							
0x0326	0x00	DP3_HCtrl	HStart[3:0]				HStop[3:0]			
0x0327	0x00	DP3_BlockCtrl3	—	—	—	—	—	—	—	BlockPacking Mode
DATA PORT 3 - BANK 1 REGISTERS										
0x0330	0x00	DP3_ChannelEn	—	—	—	—	—	—	Enable Channel 2	Enable Channel 1
0x0332	0x00	DP3_SampleCtrl1	SampleIntervalLow[7:0]							
0x0333	0x00	DP3_SampleCtrl2	SampleIntervalHigh[7:0]							
0x0334	0x00	DP3_OffsetCtrl1	Offset1[7:0]							
0x0335	0x00	DP3_OffsetCtrl2	Offset2[7:0]							
0x0336	0x00	DP3_HCtrl	HStart[3:0]				HStop[3:0]			
0x0337	0x00	DP3_BlockCtrl3	—	—	—	—	—	—	—	BlockPacking Mode

SDW register list

Address	RegName	BitName	Access	HW	HWTrig	Privilege	MSB	LSB	Reset	Descr Description
8'h00	MCP_INTSTAT									Master Control Port Interrupt Status Registers
		TX_EMPTY	r	data		0	0	0	1'b0	Transmitter empty
		RX_FULL	r	data		0	1	1	1'b0	Receiver full
		TX_UR	r	data		0	2	2	1'b0	Transmitter underrun
		RX_OR	r	data		0	3	3	1'b0	Receiver overrun
		SLVRW	r	data		0	4	4	1'b0	Interrupt Status Parity
		BUSCLASH	r	data		0	5	5	1'b0	Interrupt Status Busclash
		PORTRDY	r	data		0	6	6	1'b0	Interrupt Status Port ready
		PORT1CASCADE	r	data		0	7	7	1'b0	Interrupt Status Port 1 Cascade
8'h04	MCP_INTMASK									Master Control Port Interrupt Status Registers
		MASK_TX_EMPTY	rw	cfg		0	0	0	1'b1	Mask Transmitter empty
		MASK_RX_FULL	rw	cfg		0	1	1	1'b1	Mask Receiver full
		MASK_TX_UR	rw	cfg		0	2	2	1'b1	Mask Transmitter underrun
		MASK_RX_OR	rw	cfg		0	3	3	1'b1	Mask Receiver overrun
		MASK_SLVRW	rw	cfg		0	4	4	1'b1	Mask Interrupt Status Parity
		MASK_BUSCLASH	rw	cfg		0	5	5	1'b1	Mask Interrupt Status Busclash
		MASK_PORTRDY	rw	cfg		0	6	6	1'b1	Mask Interrupt Status Port ready
		MASK_PORT1CASCADE	rw	cfg		0	7	7	1'b1	Mask Interrupt Status Port 1 Cascade
8'h08	MCP_INTSTATMASKED									Master Control Port Interrupt Mask Registers
		TX_EMPTY	r	const		0	0	0	1'b0	Masked Status Transmitter empty
		RX_FULL	r	const		0	1	1	1'b0	Masked Status Receiver full
		TX_UR	r	const		0	2	2	1'b0	Masked Status Transmitter underrun
		RX_OR	r	const		0	3	3	1'b0	Masked Status Receiver overrun
		SLVRW	r	const		0	4	4	1'b0	Masked Status Interrupt Status Parity
		BUSCLASH	r	const		0	5	5	1'b0	Masked Status Interrupt Status Busclash
		PORTRDY	r	const	r	0	6	6	1'b0	Masked Status Interrupt Status Port ready
		PORT1CASCADE	r	const		0	7	7	1'b0	Masked Status Interrupt Status Port 1 Cascade
8'h0C	MCP_INT_CLR									Master Control Port Clear Interrupt Status Registers
		TX_UR	rw	pw1		0	2	2	1'b0	Clear transmitter underrun interrupt by writting 1
		RX_OR	rw	pw1		0	3	3	1'b0	Clear receiver overrun interrupt by writting 1
8'h10	MCP_CTRL									Master Control Port Register
		EN	rw	cfg	w	0	0	0	1'b0	Soundwire Master Enable
		CLKOFF	rw	cfg		0	1	1	1'b0	Soundwire Master Clock Input Off
		CLKSTOP_NOTFINNISHED	r	data		0	2	2	1'b0	Soundwire Master Clock Output Stop Not finnished

		ENUMEN	rw	cfg		0	3	3	1'b0	Soundwire Master Clock Output Stop Now/ used as enumeration off/on
		RESET	rw	data	w	0	4	4	1'b0	Soundwire Master Reset
		PHYSYNC	rw	cfg		1	5	5	1'b0	Soundwire PHY configuration
8'h11	MCP_CLKCFGO									Soundwire Clock Configuration Output
		SDWCLKSEL	rw	cfg		0	2	0	3'b000	Soundwire Clock Gear Select 0x0: 12,288 Mhz 0x1: 12,288 Mhz 0x2: 12,288 Mhz 0x3: 12,288 Mhz 0x4: 12,288 Mhz 0x5-0x7: 12,288 Mhz
		SDWCLKRATE	rw	cfg		0	4	3	2'b00	Soundwire Clock Rate 0x0: 0x1: 0x2: 0x3: 0x4: 0x5-0x7:
8'h12	MCP_DPCTRL									
		EN	rw	cfg	w	0	0	0	1'b0	Data Port Enable
		WORD_SIZE	rw	cfg		0	3	1	3'b101	Word size for transmitting data 0x0: 16bit 0x1: 24bit 0x2: 32bit 0x3-0x7: 32bit
8'h13	MCP_BANKCTRL									
		CURRENTBANK	rw	cfg		0	4	4	1'b0	
8'h14	MCP_TEST									
		MUX	rw	cfg		0	7	0	8'b00000000	
8'h60	MCP_FRAMECTRL									Master Control Port Frame Register
		COLUMN	rw	cfg		0	2	0	3'b000	Column Control
		ROW	rw	cfg		0	7	3	5'b00000	Row Control
8'h70	MCP_NEXTFRAME									Master Control Port Frame Register
		COLUMN	rw	cfg		0	2	0	3'b000	Column Control
		ROW	rw	cfg		0	7	3	5'b00000	Row Control
8'h80	DP1_INTSTAT									Data Port 1 Registers
		TEST_FAIL	r	data		0	0	0	1'b0	Interrupt Status Test Failed
		PORT_RDY	r	data		0	1	1	1'b0	Interrupt Status Port Ready

8'h81	DP1_INTMASK								
		TEST_FAIL	rw	cfg	0	0	0	1'b0	Interrupt Mask Test Failed
		PORT_RDY	rw	cfg	0	1	1	1'b0	Interrupt Mask Port Ready
8'h82	DP1_PORTCTRL								
		FLOW_MODE	rw	cfg	0	1	0	2'b01	Port flow mode
		DATA_MODE	rw	cfg	0	3	2	2'b00	Port data mode
		NEXT_INVBANK	rw	cfg	0	4	4	1'b0	Next invert bank
		DIR	rw	cfg	0	5	5	1'b0	Port direction
8'h83	DP1_BLOCKCTRL1								
		WORD_LEN	rw	cfg	0	5	0	6'b000000	Word length
8'h84	DP1_PREPSTAT								
		CHAN1_NOTFIN	r	const	0	0	0	1'b0	N finnished Channel 1
8'h85	DP1_PREPCTRL								
		CHAN1_PREP	rw	cfg	0	0	0	1'b0	Prepare Channel 1
8'hA0	DP1_B0_CHANEN								Data Port 1 -Bank 0 Registers
		CHAN1_EN	rw	cfg	0	0	0	1'b0	Channel Enable
8'hA2	DP1_B0_SAMPLECTRL1								
		INTERVAL_LOW	rw	cfg	0	7	0	1'b0	Sample Interval Low
8'hA3	DP1_B0_SAMPLECTRL2								
		INTERVAL_HIGH	rw	cfg	0	7	0	1'b0	Sample Interval High
8'hA4	DP1_B0_OFFSETCTRL1								
		OFFSET_1	rw	cfg	0	7	0	1'b0	Offset
8'hA5	DP1_B0_OFFSETCTRL2								
		OFFSET_2	rw	cfg	0	7	0	1'b0	Offset
8'hA6	DP1_B0_HCTRL								
		STOP	rw	cfg	0	3	0	1'b0	H Stop
		START	rw	cfg	0	7	4	1'b0	H Start
8'hA7	DP1_B0_BLOCKCTRL3								
		BLOCK_PKG_MODE	rw	cfg	0	0	0	1'b0	Block Packing Mode
8'hB0	DP1_B1_CHANEN								Data Port 1 -Bank 1 Registers
		CHAN1_EN	rw	cfg	0	0	0	1'b0	Channel Enable
8'hB2	DP1_B1_SAMPLECTRL1								
		INTERVAL_LOW	rw	cfg	0	7	0	1'b0	Sample Interval Low
8'hB3	DP1_B1_SAMPLECTRL2								
		INTERVAL_HIGH	rw	cfg	0	7	0	1'b0	Sample Interval High
8'hB4	DP1_B1_OFFSETCTRL1								
		OFFSET_1	rw	cfg	0	7	0	1'b0	Offset
8'hB5	DP1_B1_OFFSETCTRL2								
		OFFSET_2	rw	cfg	0	7	0	1'b0	Offset

8'hB6	DP1_B1_HCTRL								
		STOP	rw	cfg	0	3	0	1'b0	H Stop
		START	rw	cfg	0	7	4	1'b0	H Start
8'hB7	DP1_B1_BLOCKCTRL3								
		BLOCK_PKG_MODE	rw	cfg	0	0	0	1'b0	Block Packing Mode
8'hC0	SLV1_DEVID0								
		UNIQUEIDVERS	r	const	0	7	0	8'b00000000	Slave Device Unique ID
8'hC1	SLV1_DEVID1								
		MANID0	r	const	0	7	0	8'b00000000	Manufacturer ID
8'hC2	SLV1_DEVID2								
		MANID1	r	const	0	7	0	8'b00000000	Manufacturer ID
8'hC3	SLV1_DEVID3								
		PARTNUM0	r	const	0	7	0	8'b00000000	Audio Part Number
8'hC4	SLV1_DEVID4								
		PARTNUM1	r	const	0	7	0	8'b00000000	Audio Part Number
8'hC5	SLV1_DEVID5								
		Class	r	const	0	7	0	8'b00000000	Class Mipi Reserved
8'hD0	SDW_SLV_OP								
		READ	rw	data w	0	0	0	1'b0	SDW Slave Register Read
		WRITE	rw	data w	0	1	1	1'b0	SDW Slave Register Write
8'hD1	SDW_SLVDEV								
		ADDR	rw	cfg	0	3	0	4'b0000	SDW Slave Device ID
8'hD2	SDW_SLVLOW								
		ADDR	rw	cfg	0	7	0	8'b00000000	SDW Slave Register Address lower part
8'hD3	SDW_SLVHIGH								
		ADDR	rw	cfg	0	7	0	8'b00000000	SDW Slave Register Address higher part
8'hD4	SDW_SLV								
		DATA	rw	data	0	7	0	8'b00000000	SDW Slave Register DATA Write
8'hD5	SDW_SLVRDATA								
		DATA	r	data	0	7	0	8'b00000000	SDW Slave Register DATA Read
8'hF8	REGINFO								Register file information register
		DATE	r	const	0	17	0	18'dYY05DD	Date
8'hFC	ID								ID register
		PERIPHERAL_TYPE	r	const	0	7	0	8'h20	Peripheral ID
		VERSION	r	const	0	15	8	8'h01	Version
Last line of pin list - insert new lines above									

Bibliography

Abdellatif Bellaouar, Mohammed Ibrahim Elmasry (June 30, 1995). *Low-Power Digital VLSI Design Circuits and Systems* (cit. on p. 2).

Alexander Khazin, Lior Amarillio (2018). "Power reduction through clock management." US20180032307A1 (cit. on pp. 19, 22).

AnalogDevices (Jan. 2014). *SigmaDSP Digital Audio Processor ADAU 1452*. Ed. by AnalogDevices. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ADAU1452.pdf (cit. on p. 34).

Andrea Rusconi Clerici, Ferruccio Bottoni (May 22, 2018). "MEMS Loudspeaker having an Actuator Structure and a Diaphragm spaced apart therefrom." 9980051B2 (cit. on p. 1).

Arasan (2020). *MIPI SoundWire Master Controller 1.1*. Ed. by Arasan. URL: https://www.arasan.com/products/mipi/soundwire/soundwire-master/#sec2 (cit. on pp. 35, 37).

ARM (Aug. 17, 2004). *AMBA 3 APB Protocol*. Ed. by ARM (cit. on p. 43).

Chennakesavulu, M., T. .Jayachandra Prasad, and V. Sumalatha (Dec. 12, 2018). "Data encoding techniques to improve the performance of System on Chip." In: *Computer and Information Sciences* (cit. on p. 26).

CirrusLogic (Nov. 2019). *Low Power Audio Codec with SoundWire and Audio Processing*. Ed. by CirrusLogic. URL: https://www.cirrus.com/products/cs42142/ (cit. on p. 30).

Cutress, Dr. Ian (Dec. 11, 2019). *Intel's Manufacturing Roadmap from 2019 to 2029: Back Porting, 7nm, 3nm, 2nm, and 1.4nm*. URL: https://www.anandtech.com/show/15217/intels-manufacturing-roadmap-from-2019-to-2029 (cit. on p. 4).

Freescale Semiconductor, Inc. (1994). *Synchronous Serial Interface*. Ed. by Inc. Freescale Semiconductor. URL: http://notes-application.abcelectronique.com/314/314-68633.pdf (cit. on p. 17).

FreescaleSemiconductor (2000). *Enhanced Serial Audio Interface*. Tech. rep. Freescale Semiconductor (cit. on p. 17).

Inc., AnalogDevices (2016). *ADSP-SC5373 EZ-KIT Manual*. Ed. by AnalogDevices Inc. (cit. on p. 19).

Inc., MIPI Alliance (2008). *MIPI Alliance Specification for Serial Low-power Inter-chip Media Bus (SLIMbus)*. Tech. rep. MIPI Alliance Inc. (cit. on p. 20).

Inc., Motorola (2001). *DSP56301 User Manual*. Ed. by Motorola Inc. (cit. on p. 17).

Intel (2002). *Audio Codec '97*. Tech. rep. Intel Corporation (cit. on p. 15).

Intel (2010). *High Definition Audio Specification*. Tech. rep. Intel Corporation (cit. on p. 16).

Intel (Sept. 2019). *Intel 300 Series Chipset Family On-Package Platform Controller Hub*. Tech. rep. Intel (cit. on pp. 20, 30).

Jafarzadeh, N. et al. (2014). "Data Encoding Techniques for Reducing Energy Consumption in Network-on-Chip." In: *Transactions on Very Large Scale Integration (VLSI), Systems22 22(3) 675-685* (cit. on p. 26).

MaximIntegrated (Mar. 2018). *MAX98374 Digital Input Class D Speaker Amplifier with DHT*. Ed. by MaximIntegrated. URL: https://datasheets.maximintegrated.com/en/ds/MAX98374.pdf (cit. on pp. 30, 34).

MIPI Alliance, Inc. (Jan. 23, 2019). *Specification for SoundWire*. Ed. by Inc. MIPI Alliance (cit. on pp. 20, 43).

Nawrocki, W. (2011). "Physical Limits for Scaling of Electronic Devices in Integrated Circuits." In: *Physical Properties of Nanosystems* (cit. on p. 4).

NXP (Apr. 4, 2014). *UM10204 I²C-bus specification and user manual*. Ed. by NXP (cit. on p. 11).

Panda Preeti Silpa, Shrivastava Aviral (Jan. 1, 2010). *Power-efficient System Design*. ISBN: 9781441963888. DOI: 10.1007/978-1-4419-6388-8 (cit. on pp. 2, 6).

Pavel Bohacik, Automotive and Industrial Solutions Group (2012). *MPC5604 Serial Audio Interface*. Tech. rep. Freescale Semiconductor (cit. on p. 17).

PhilipsSemiconductors (1996). *I2S bus specification*. Tech. rep. PhilipsSemiconductors (cit. on p. 14).

Pierre-Louis Bossart Juha Backman, Jens Kristian Poulsen (2014). *SoundWire: a new MIPI standard audio interface*. Convention e-Brief 172. Audio Engineering Society (cit. on pp. 20, 79).

Rakesh Chadha, Jayaram Bhasker (Dec. 5, 2012). *An ASIC Low Power Primer: Analysis, Techniques and Specification* (cit. on p. 3).

Sanchez, Zeke van (Mar. 2018). *XEM7310*. Ed. by Zeke van Sanchez. URL: https://docs.opalkelly.com/display/XEM7310/XEM7310 (cit. on p. 31).

SMSC (2010). *Media Local Bus Specification*. Tech. rep. SMSC (cit. on p. 18).

TDK (Jan. 8, 2020). *TDK announces world's first MIPI standard SoundWire® microphone*. Ed. by TDK. URL: https://invensense.tdk.com/news-media/tdk-invensense-announces-t5808-worlds-first-mipi-standard-soundwire-microphone/ (cit. on p. 30).

TexasInstruments (2001). *PCM CODEC*. Tech. rep. TexasInstruments (cit. on p. 12).

Xilinx (Mar. 7, 2011). *AXI Reference Guide*. Ed. by Xilinx. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf (cit. on p. 43).

Xilinx (May 8, 2018). *7 Series FPGAs SelectIO Resources*. Ed. by Xilinx (cit. on p. 47).

CMOS Scaling into the Nanometer Regime (Apr. 4, 1997) (cit. on p. 5).