Digitale Signalverarbeitung mittels FPGA

Wahlmodul Entwicklung digitaler Systeme

von

Philip Ridder

Betreuer: Dipl. –Ing. (FH) Oliver Gießelmann

Dozent: Prof. Dr. Udo Jorczyk

Gelsenkirchen 21.Februar 2017

INHALT

1 Einleitung

Dieses Dokument stellt die ergänzende Dokumentation zum Projekt Digitale Signalverarbeitung mittels FPGA dar. Das Projekt ist Teil der Veranstaltung Entwurf digitaler Systeme an der Westfälischen Hochschule Gelsenkirchen aus dem Wintersemester 2016/2018 unter Leitung von Prof. Dr. –Ing. Udo Jorczyk.

Die Veranstaltung Entwurf digitaler Systeme soll den Studierenden einen Einstieg in die Hardwarebeschreibungssprache VHDL liefern, und praktische Fähigkeiten durch die Realisierung von Projekten vermitteln.

In dem Projekt Digitale Signalverarbeitung mittels FPGA wird die Filterung eines Audio-Signals mittels FIR-Filter realisiert. Die Entwicklung erfolgt auf einem Terasic DE1-SoC Entwicklungsboard. Dieses Board ist mit einem Cyclone V SoC von Altera bestückt. In diesem SoC sind sowohl eine FPGA-Struktur, als auch ein ARM Cortex Prozessor integriert. Der ARM Cortex wird in diesem Projekt zur Kommunikation mit einem PC und zur Konfiguration des FPGA während des Bootvorgangs verwendet. Der Prozessor wird mit einer angepassten Version von Yocto Linux betrieben.

2 Systembeschreibung

In dem Projekt werden hauptsächlich der Cyclone V und der Audio Codec des Entwicklungsboards verwendet. Sollten noch weitere Komponenten verwendet werden, werden diese im Lauf dieser Dokumentation erläutert.

Der Audio Codec dient dazu, ein eingehendes Audio Signal zu digitalisieren und dem FPGA über eine serielle Schnittstelle zur Verfügung zu stellen und ebenfalls aus einem seriellen Signal wieder ein analoges Signal zu erzeugen. Die Initialisierung und Konfiguration des Audio Codec erfolgt dabei über den ARM Cortex Prozessor (HPS).

Der FPGA nimmt die seriellen Daten des Audio Codec auf, verarbeitet diese und sendet das verarbeitete Signal seriell an den Audio Codec zurück.

Um eine korrekte Aussteuerung der ein- und ausgehenden Signale zu gewährleisten, wurde eine Aussteuerungsanzeige (Meterbridge) mit dem FPGA implementiert. Die Anzeige erfolgt mit den zehn integrierten LED des Entwicklungsboards. Über die Schalter SW0 und SW1 kann die Aussteuerungsanzeige auf den rechten beziehungsweise den linken Kanal geschaltet werden, als auch vor beziehungsweise hinter den FIR-Filter.

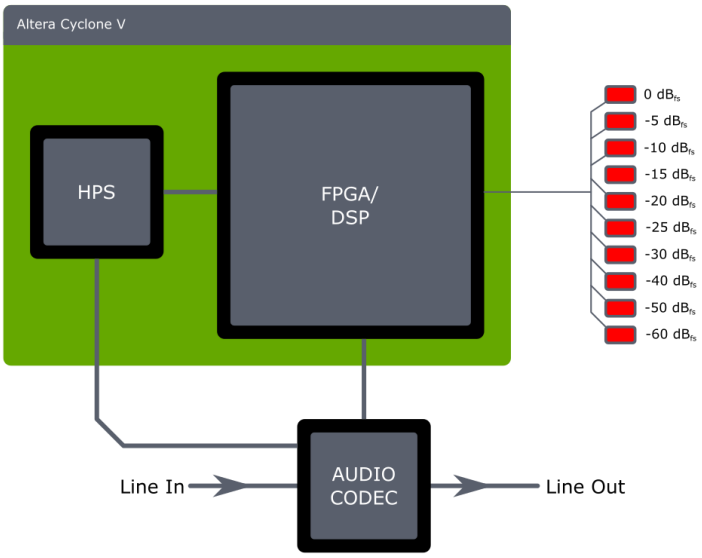


Abbildung 1: Systemaufbau

2.1 Signalverarbeitung

Wie bereits zuvor beschrieben erfolgt die Signalverarbeitung auf der FPGA-Struktur. Da der FIR-Filter die eingehenden Daten parallel verarbeitet, müssen die eingehenden Signale zunächst parallelisiert und nach Verlassen des Filters wieder serialisiert werden. Dies erfolgt über die Entitys *SERIAL_TO_PARALLEL* und *AUDIO_PARALLEL_SERIAL*. Das Routing der Signale auf die Aussteuerungsanzeige erfolgt mit Hilfe der Bus Multiplexer *BUS_MUX BM1* und *BM2*.

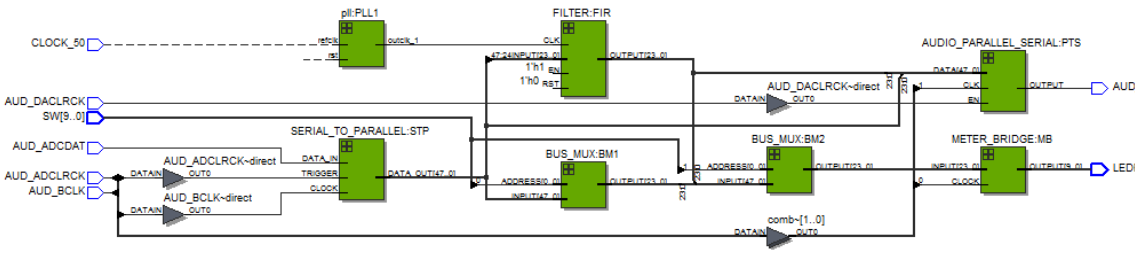


Abbildung 2: RTL Ansicht

3 Inbetriebnahme

Nachdem man das SD-Karten Image des Systems heruntergeladen hat, muss es zunächst mit dem Programm Win32DiskImager auf eine SD-Karte kopiert werden, welche mindestens über 8 GB Kapazität verfügt.

Wenn die SD-Karte ins Entwicklungsboard eingelegt ist, müssen auf der Unterseite des Entwicklungsboards die korrekten Einstellungen an den MSEL-Schaltern vorgenommen werden, damit der Prozessor direkt von der SD-Karte booten kann. Die korrekte Einstellung ist in Abbildung 3 zu sehen.

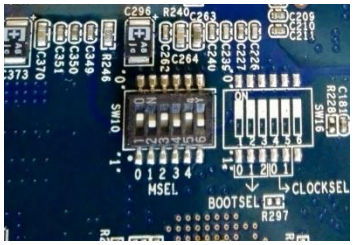


Abbildung 3: MSEL Einstellungen

Wenn die Einstellungen korrekt vorgenommen sind und das Board mit Strom versorgt wird, bootet der HPS automatisch von der SD-Karte und konfiguriert den FPGA.

4 Einrichten der Entwicklungsumgebung

Während der Bearbeitung des Projektes hat sich gezeigt, dass bei der Arbeit mit einem solch komplexen System bereits während der Startphase einige Probleme auftreten können. Um eine solide Basis zu schaffen folgt eine kurze Beschreibung der wichtigsten Einstellungen und Zusammenhänge. Auf dieser Grundlage kann das vorhandene Projekt gegebenenfalls den eigenen Wünschen angepasst werden.

4.1 Herstellen der Seriellen Kommunikation

Zur Übermittlung von Befehlen an den HPS wird eine serielle Verbindung benötigt. Dazu muss zunächst das Entwicklungsboard über USB mit dem PC verbunden werden. Hier ist nicht der USB BLASTER Anschluss (links) zu verwenden, sondern der UART TO USB Anschluss (oben rechts). Sobald das Board verbunden ist, sollte ein virtueller COM-Port am PC eingerichtet werden. Um eine serielle Verbindung herzustellen, muss man wissen, an welchem COM-Port das Board angeschlossen ist. Dies erfährt man am einfachsten über den Gerätemanager.

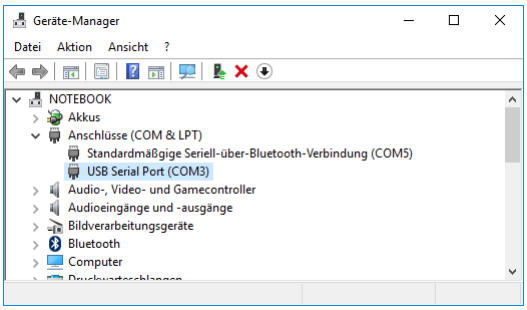


Abbildung 4: Gerätemanager

Startet man nun die Entwicklungsumgebung (Eclipse for DS-5), kann man über *Window → Show View → Other* ein neues Terminal Fenster öffnen.

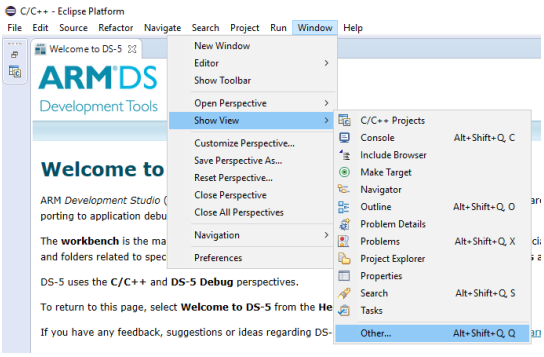


Abbildung 5: Show View

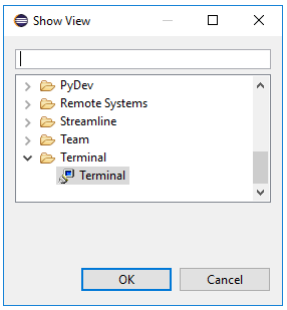


Abbildung 6: Terminal erstellen

Unter Settings nimmt man dann die Einstellungen entsprechend Abbildung 7 vor.

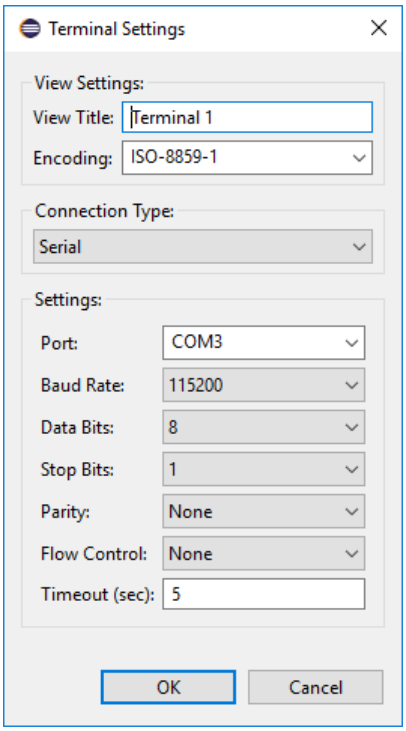


Abbildung 7: Terminal Settings

Stellt man nun eine Verbindung über Connect mit dem Board her und startet das Board neu, sollte das Entwicklungsboard folgenden Output liefern. Das Passwort zum System lautet *root*.

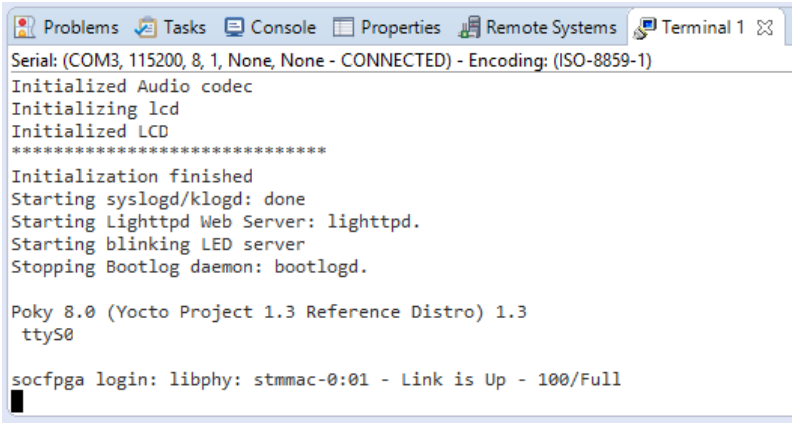


Abbildung 8: Serieller Output nach Bootvorgang

4.2 Daten per SSH übertragen

Während der Entwicklung kann es hilfreich sein, Daten per Drag and Drop auf die SD-Karte schieben zu können. Dies hat den Vorteil, dass die SD-Karte nicht ständig entfernt werden muss. Dazu wird eine SSH-Verbindung benötigt, welche zunächst eingerichtet werden muss. Eine SSH-Verbindung erfolgt über eine Netzwerkverbindung. Wird das Board direkt mit dem PC verbunden, ist darauf zu achten, dass ein Crossover-Kabel verwendet wird.

4.2.1 SSH einrichten

Um ein SSH dienst einzurichten müssen die nachfolgenden Befehle ausgeführt werden.

```
root@socfpga:cd /etc/ssh
root@socfpga:/etc/ssh# cp sshd_config sshd_config.orig
root@socfpga:/etc/ssh# echo PermitEmptyPasswords yes >> sshd_config
root@socfpga:/etc/ssh# start-stop-daemon -K -x /usr/sbin/sshd
root@socfpga:/etc/ssh# start-stop-daemon -S -x /usr/sbin/sshd
```

Listing 1: SSH einrichten

Mit dem ersten Befehl (*change directory*) wechselt man in das Verzeichnis */etc/ssh/*, wo die Konfigurationsdatei des SSH-Dienstes liegt.

Durch den zweiten Befehl (*copy*) wird die Konfigurationsdatei *sshd_config* kopiert und als *sshd_config.orig* gespeichert.

Der dritte Befehl (*echo*) setzt den Parameter *PermitEmptyPassword* in der *sshd_config* Datei auf *yes*. Somit ist ein Zugriff auf die SD-Karte über SSH ohne Passwort möglich. Ist dies nicht gewünscht, muss dieser Parameter auf *no* gesetzt werden.

Um die Konfiguration zu übernehmen, muss der SSH-Dienst neugestartet werden. Dazu dienen die Befehle vier und fünf. Die Option *–K* steht dabei für *kill* und *–S* für *start*.

4.2.2 Passwort festlegen

Mit dem Befehl *passwd* wird das Passwort festgelegt. Durch die vorige Einstellung ist es nun möglich ein leeres Passwort zu vergeben.

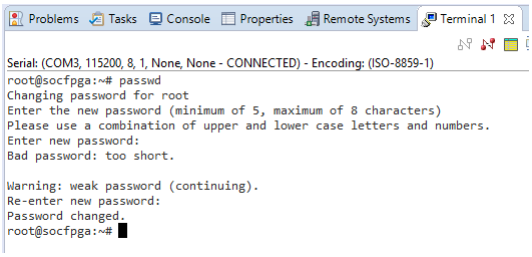


Abbildung 9:Ändern des Passworts

4.2.3 Remote Verbindung einrichten

Zunächst öffnet man die Remote Systems Ansicht über *Window* → *Show View* → *Others*.

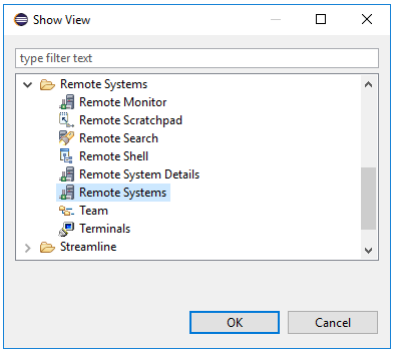


Abbildung 10: Remote Systems Ansicht öffnen

Über den Button *Define a new connection to a remote System* erstellt man eine neue *SSH Only* Verbindung. Im nachfolgenden Fenster wird die IP-Adresse des Entwicklungsboards eingetragen.

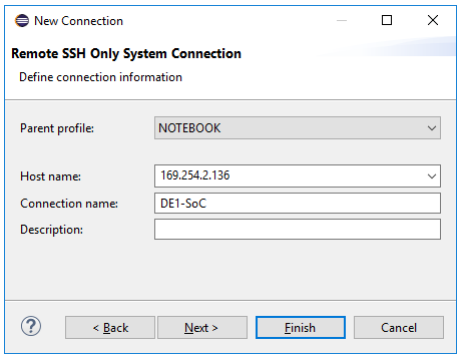


Abbildung 11: SSH Verbindung einrichten

Ist die IP-Adresse nicht bekannt, erhält man nähere Informationen über die Serielle Verbindung, indem man den Befehl *ifconfig eth0* eingibt. Die Antwort des Boards sollte ähnlich der in Abbildung 12 aussehen.

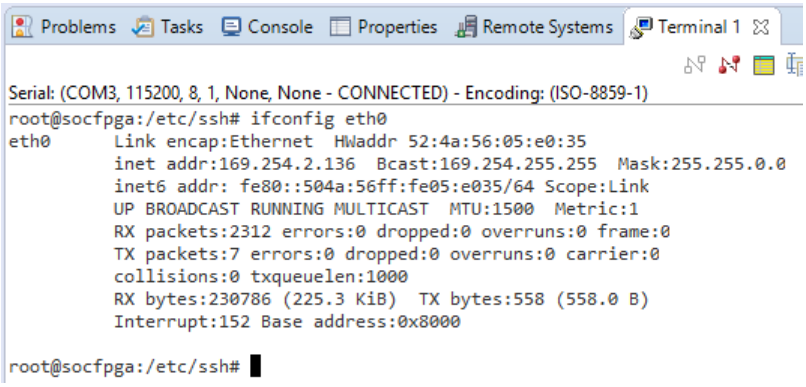


Abbildung 12:IP Adresse des Entwicklungsboards

Nach Bestätigen mit *Finish* wird eine neue Verbindung angelegt. Damit diese problemlos funktioniert, muss noch der Benutzer geändert werden. Über die Eigenschaften der Verbindung kann der Standardbenutzer eingestellt werden.

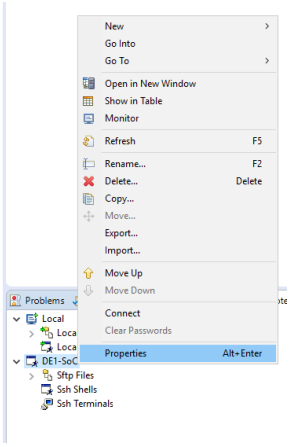
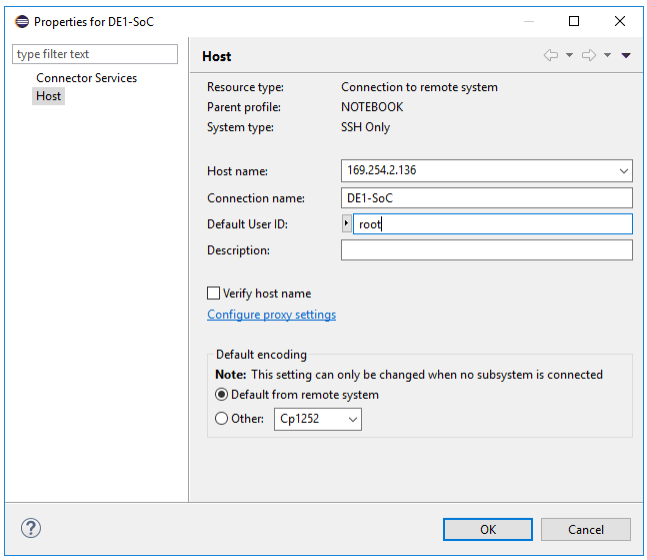


Abbildung 13: Verbindungseigenschaften

Der Standardbenutzer sollte hier *root* lauten.



Eine Verbindung zum Entwicklungsboard ist nun möglich und man sollte vollen Zugriff auf das Dateisystem der SD-Karte erhalten.

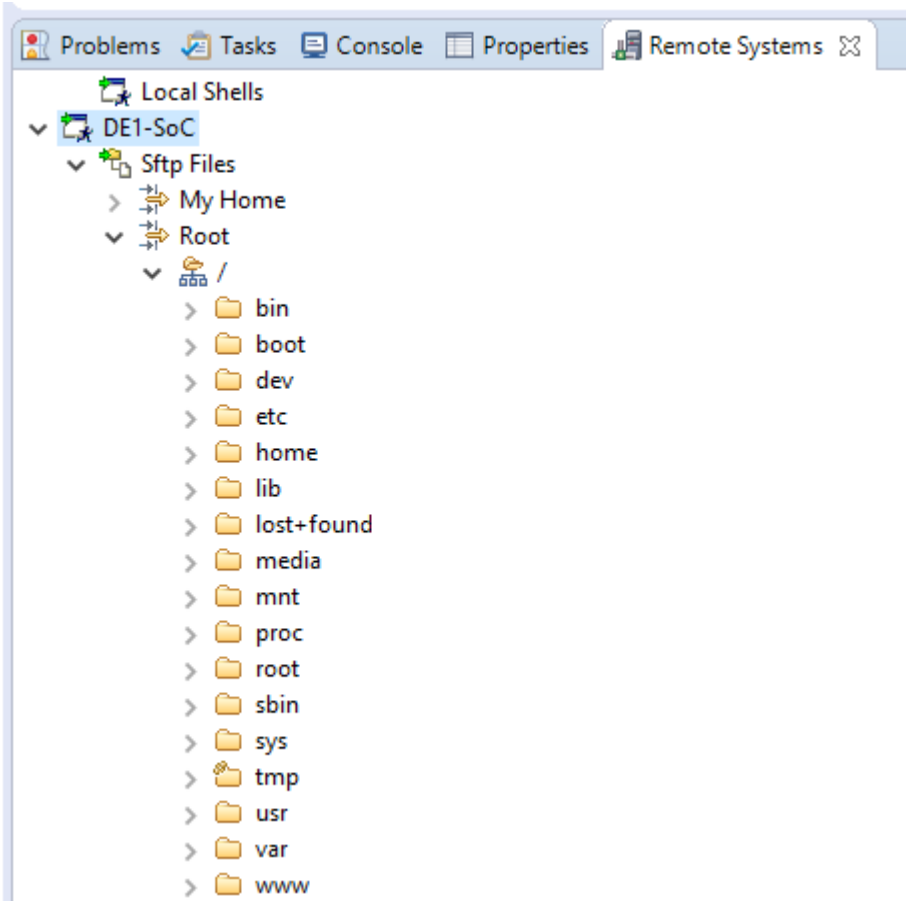


Abbildung 14: Remote Zugriff auf SD-Karte

4.3 IP-Adresse Festlegen

Möchte man regelmäßig per Remote Zugriff auf Dateien zugreifen ist es sinnvoll, eine feste IP-Adresse für das Board einzurichten. Um dies zu erreichen, muss die Konfiguration in der Datei *interfaces* geändert werden. Diese befindet sich im Verzeichnis */etc/network/*.

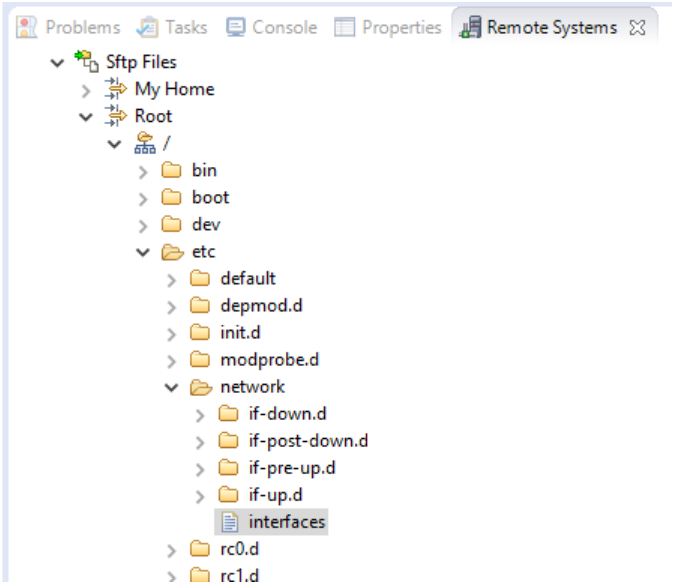


Abbildung 15: Interfaces

Hier müssen die Einträge folgendermaßen geändert werden. Die Zeile

iface eth0 inet dhcp

muss mit einer Raute auskommentiert werden. Dies verhindert, dass das Board eine IP-Adresse über einen dhcp Dienst bezieht. Nun muss die Datei um

auto eth0

iface eth0 inet static

address 169.254.2.136

netmask 255.255.0.0

erweitert werden. In diesem Teil wird festgelegt, dass die eingetragene Adresse eine statische IP ist.

```
16 # Wired or wireless interfaces
17 #auto eth0
18 #iface eth0 inet dhcp
19 iface eth1 inet dhcp
20
21 auto eth0
22 iface eth0 inet static
23     address 169.254.2.136
24     netmask 255.255.0.0
25
```

Abbildung 16: Interfaces Einträge

4.4 FPGA beim Booten konfigurieren

Wird die Stromversorgung eines FPGA getrennt, verliert dieser auch die gesamte Konfiguration. Damit beim Bootvorgang der FPGA konfiguriert wird, sind einige Schritte nötig um dies zu einzurichten. Dazu benötigt man ein Programm, welches die Konfigurationsdaten in den FPGA lädt und ein Skript, welches während des Bootvorgangs ausgeführt wird.

Das Programm befindet sich auf der SD-Karte im Verzeichnis

/home/root/fpga_config/

Dort befindet sich ebenfalls die Konfigurationsdatei (.rbf) für dieses Projekt. Möchte man den FPGA manuell konfigurieren, erreicht man dies über das serielle Terminal mit dem Befehl

./hps_config_fpga soc_system.rbf

Soll die Konfiguration automatisch erfolgen benötigt man zunächst eine Skript-Datei, welche genau diesen Befehl ausführt. Skript-Dateien, welche beim Booten ausgeführt werden befinden sich in der Regel im Verzeichnis

/etc/init.d

Die Skript-Datei zu diesem Projekt ist die startup.sh Datei. Der Aufbau ist in

```
#!/bin/bash

echo "Starting SoC initialization"
echo "*****************************"

cd /home/root/fpga_config/
echo "Configuring FPGA fabric"
./hps_config_fpga "soc_system.rbf"

cd /home/root/
echo "Routing I2C multiplexer"
./i2cmux "hps"

echo "Initializing audio codec"
./audioinitialize

echo "*****************************"
echo "Initialization finished"


exit 0
```

Listing 2 zu sehen. Besonders wichtig bei einer Skript-Datei ist die erste Zeile. Diese beinhaltet eine sogenannte *Magic Line* (*Shebang*). Die Zeichenkombination *#!* gefolgt von einem Parameter legt fest, um welchen Dateityp es sich handelt. In diesem Fall ist ein Bash-Skript gemeint. Fehlt diese Zeile, wird das Skript nicht korrekt ausgeführt.

Das Keyword *echo* gefolgt von einem String sorgt während der Ausführung dafür, dass der angegebene String über die Konsole ausgegeben wird.

Um nun die Konfigurationsdatei zu laden, wird exakt wie beim manuellen Laden vorgegangen. Zuerst wird das Verzeichnis mit *cd* (*change directory*) gewechselt und danach das Konfigurationstool ausgeführt. Wichtig dabei ist, dass die Programmausführung mit der Kombination *./* gestartet wird und der Parameter als String, also in Anführungszeichen, übergeben wird.

Ist das Skript abgearbeitet, wird dies mit dem Befehl *exit 0* verlassen. Sollten während der Ausführung des Skriptes Fehler auftreten, ist der Exitcode von Null verschieden.

```
#!/bin/bash

echo "Starting SoC initialization"
echo "*****************************"

cd /home/root/fpga_config/
echo "Configuring FPGA fabric"
./hps_config_fpga "soc_system.rbf"

cd /home/root/
echo "Routing I2C multiplexer"
./i2cmux "hps"

echo "Initializing audio codec"
./audioinitialize

echo "*****************************"
echo "Initialization finished"


exit 0
```

Listing 2: startup.sh

Damit dieses Skript beim Booten ausgeführt wird, muss dem System mitgeteilt werden, dass ein weiteres Skript geladen wird. Dazu führt man den Befehl

update-rc.d startup.sh defaults

aus. Beim nächsten Neustart wird nun das Skript automatisch ausgeführt und die FPGA Konfiguration geladen.

5 Wichtige Zusammenhänge

Gerade während der ersten Gehversuche fehlen Informationen über fundamentale Zusammenhänge, welche nicht hinreichend dokumentiert sind. Dieser Abschnitt soll eine knappe Übersicht über die wichtigsten Zusammenhänge liefern, um nicht zu Beginn schon die Motivation zu verlieren.

5.1 QSYS und Eclipse

Um die Funktionalität des HPS zu erweitern verwendet man QSYS. QSYS ist ein integriertes Tool in der Quartus Entwicklungsumgebung. Hier werden zum Beispiel Schnittstellen zwischen dem HPS und der FPGA-Struktur definiert. Diese kommunizieren dann später über einen gemeinsamen Speicher. Die Start (Base) und End-Adressen des Speichers sind in Abbildung 17 zu sehen.

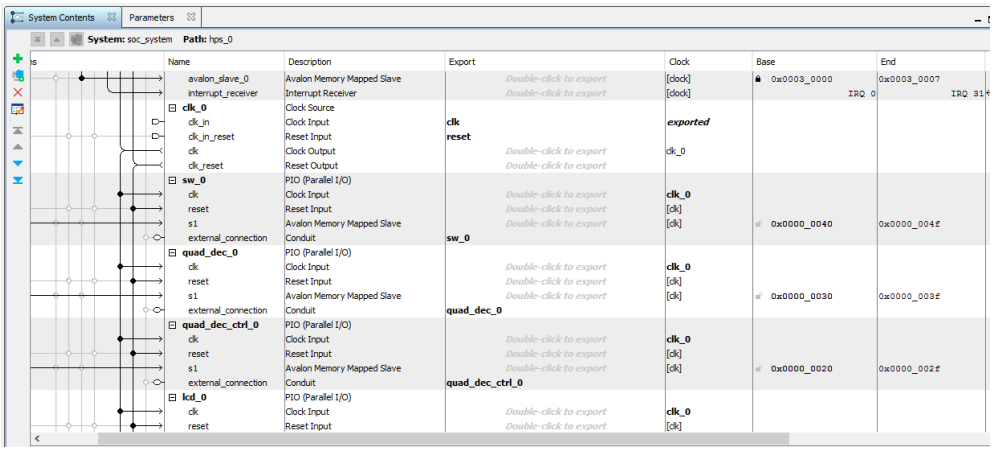


Abbildung 17: QSYS

Wenn nun über den HPS auf diesen Speicher zugegriffen werden soll, kann man diese Adressen händisch in eine Header-Datei übertragen, oder ein speziell dafür geschriebenes Tool verwenden. Dieses Tool ist ein Kommandozeilenprogramm, welches man über die *Nios II Command Shell erreicht*.

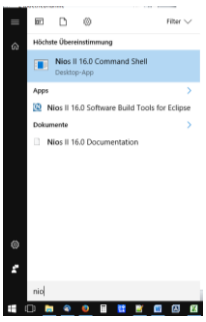


Abbildung 18: Nios II Command Shell

Zunächst wechselt man in das Verzeichnis, wo die *.qsys*-Datei zu dem HPS-System abgelegt ist. Danach führt man den Befehl

./generate_hps_qsys_header.sh

aus. Nach erfolgter Erstellung einer Header Datei sollte die Ausgabe der Konsole wie folgt aussehen.

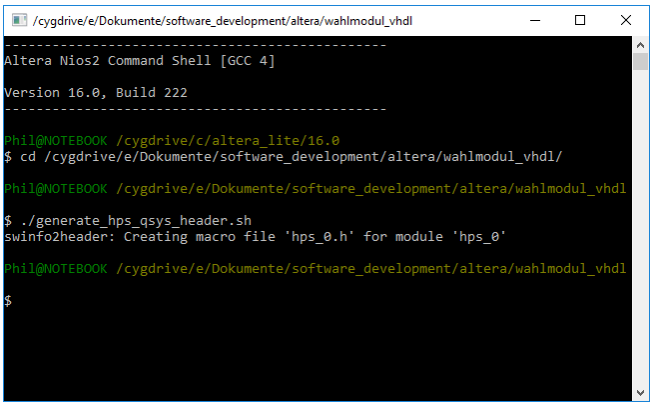


Abbildung 19:Header Erstellung

In dem Verzeichnis sollte sich nun eine Datei mit dem angegebenen Namen befinden, welche Compilerdirektiven mit den entsprechenden Speicherbereichen beinhalten.

```
153  * Macros for device 'sw_0', class 'altera_avalon_pio'
154  * The macros are prefixed with 'SW_0_'.
155  * The prefix is the slave descriptor.
156  */
157 #define SW_0_COMPONENT_TYPE altera_avalon_pio
158 #define SW_0_COMPONENT_NAME sw_0
159 #define SW_0_BASE 0x40
160 #define SW_0_SPAN 16
161 #define SW_0_END 0x4f
162 #define SW_0_BIT_CLEARING_EDGE_REGISTER 0
163 #define SW_0_BIT_MODIFYING_OUTPUT_REGISTER 0
164 #define SW_0_CAPTURE 0
165 #define SW_0_DATA_WIDTH 10
```

Abbildung 20: Compilerdirektiven mit Speicherbereichen

6 Software

Dieser Abschnitt soll einen Einblick in die Software geben, welche benötigt wird, um die gewünschte Funktionalität herzustellen. Der Fokus wird dabei auf die Programme gelegt, welche im Startskript aufgerufen werden. Da das Konfigurationstool *hps_config_fpga* von Terasic übernommen wurde und für beliebige .rbf-Dateien verwendet werden kann, wird darauf nicht weiter eingegangen. Vielmehr sind es *i2cmux* und *audioinitialize,* welche bis hier noch keine Aufmerksamkeit bekommen haben.

Das Entwicklungsboard verfügt über einen Analogmultiplexer. Dieser ermöglicht es, periphere Bausteine, wie den Audio Codec, sowohl vom FPGA, als auch vom HPS über I2C anzusprechen. *i2cmux* ermöglicht es, diesen Multiplexer zwischen beiden Strukturen umzuschalten.

audioinitialize dient dazu, den Audio Codec nach Benutzervorgaben zu initialisieren. Dies könnte auch mit Hilfe eines IP-Cores von Altera erfolgen, es existiert aber ein ausschlaggebendes Argument dies nicht zu machen. FPGA-Ressourcen sind immer begrenzt und im Vergleich zum Speicher sehr wertvoll. Erfolgt die Konfiguration über den HPS, wird lediglich Speicher auf der SD-Karte belegt und die FPGA-Struktur bleibt unberührt. Ein weiterer Grund ist die Flexibilität. Während Änderungen in der FPGA-Struktur immer eine komplette Synthese mit sich zieht (Quartus Lite), muss eine Änderung am Programm lediglich neu kompiliert werden.

6.1 i2cmux

Der I2C Multiplexer wird über einen bestimmten Pin des Cyclone V angesteuert. Dieser wiederum kann über das Schreiben eines bestimmten Registers angesprochen werden. In Unix basierten Betriebssystemen muss dafür nicht extra ein neuer Treiber geschrieben werden. Man kann direkt über ein Speicherabbild auf den entsprechenden Speicherbereich zugreifen. Dazu befindet sich im Verzeichnis */dev/* die Datei *mem*. Diese Datei stellt quasi eine Kopie der Hardwareregister dar. Das Angenehme daran ist, dass man auf sie zugreifen kann, wie auf eine Textdatei. Man sollte nur wissen, in welchen Bereich man hineinschreiben möchte/darf und in welche nicht. Der Zugriff darauf erfolgt immer auf die gleiche Weise. Zunächst wird die Datei mit dem Befehl *open()* geöffnet und ein Handle angelegt, welches später verwendet wird. *MEMORY_MAP_FILE_NAME* ist dabei die Adresse der Speicherabbilddatei. *O_RDWR* und *O_SYNC* sind Flags, welche Zugriffseigenschaften der Funktion steuern.

```
fd = open(MEMORY_MAP_FILE_NAME, (O_RDWR | O_SYNC));
```

Anschließend wird über den Befehl *mmap()* ein bestimmter Bereich dieser Datei in den Speicher gemappt. Der Befehl gibt ein Handle zurück, mit dem später der Speicher angesprochen wird.

```
virtual_base = mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE),
MAP_SHARED, fd, HW_REGS_BASE);
```

**i2c_switch_0_addr* ist hier ein Pointer, der die Adresse von *virtual_base* enthält. HPS_I2C_CONTROL ist das Bit im Speicher, welches den Pin des I2C Multiplexers steuert. In dem unten zu sehenden Beispiel wird dieser Speicherbereich mit einer 1 beschrieben.

```
*i2c_switch_0_addr |= HPS_I2C_CONTROL;
```

Möchte man den Speicherbereich zurücksetzen, muss die Zuweisung wie folgt aussehen.

```
*i2c_switch_0_addr &= ~(HPS_I2C_CONTROL);
```

Möchte man zu Testzwecken auf das Programm zugreifen wurden die Befehle *hps* und *fpga* implementiert. Man kann den Multiplexer also über das serielle Terminal mit den Befehlen

i2xmux fpga

und

i2cmux hps

auf den FPGA beziehungsweise den HPS routen.

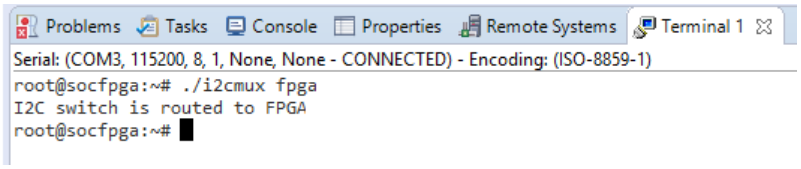


Abbildung 21: I2C Multiplexer

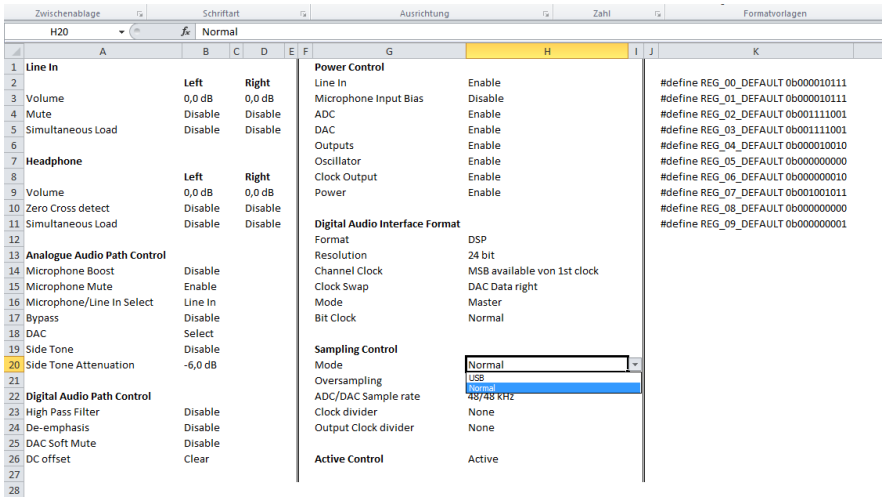
6.2 audioinitialize

Der Zugriff auf die I2C verantwortlichen Register erfolgt auf die gleiche Weise wie in 6.1 beschrieben. Zur Initialisierung wird eine Reihe von Befehlen an den Audio Codec übermittelt. Da der Audio Codec ein Write Only Device ist, bietet es sich an, alle Register auf einen bekannten Wert zu initialisieren. Dazu werden in der Headerdatei *audio_codec.h* diese Register definiert und an den Coden übertragen.

```
#define REG_00_DEFAULT 0b000010111
#define REG_01_DEFAULT 0b000010111
#define REG_02_DEFAULT 0b001111001
#define REG_03_DEFAULT 0b001111001
#define REG_04_DEFAULT 0b000010010
#define REG_05_DEFAULT 0b000000000
#define REG_06_DEFAULT 0b000000010
#define REG_07_DEFAULT 0b001001011
#define REG_08_DEFAULT 0b000000000
#define REG_09_DEFAULT 0b000000001
```

Listing 3: audio_codec.h

Damit man sich nicht jedes Mal mit dem Datenblatt quälen muss, wurde eine Excel-Datei erstellt, welche die Konfiguration vereinfachen soll.



	A	B	D	G	H	K
1	**Line In**			**Power Control**		
2		**Left**	**Right**	Line In	Enable	#define REG_00_DEFAULT 0b000010111
3	Volume	0,0 dB	0,0 dB	Microphone Input Bias	Disable	#define REG_01_DEFAULT 0b000010111
4	Mute	Disable	Disable	ADC	Enable	#define REG_02_DEFAULT 0b001111001
5	Simultaneous Load	Disable	Disable	DAC	Enable	#define REG_03_DEFAULT 0b001111001
6				Outputs	Enable	#define REG_04_DEFAULT 0b000010010
7	**Headphone**			Oscillator	Enable	#define REG_05_DEFAULT 0b000000000
8		**Left**	**Right**	Clock Output	Enable	#define REG_06_DEFAULT 0b000000010
9	Volume	0,0 dB	0,0 dB	Power	Enable	#define REG_07_DEFAULT 0b001001011
10	Zero Cross detect	Disable	Disable			#define REG_08_DEFAULT 0b000000000
11	Simultaneous Load	Disable	Disable	**Digital Audio Interface Format**		#define REG_09_DEFAULT 0b000000001
12				Format	DSP	
13	**Analogue Audio Path Control**			Resolution	24 bit	
14	Microphone Boost	Disable		Channel Clock	MSB available von 1st clock	
15	Microphone Mute	Enable		Clock Swap	DAC Data right	
16	Microphone/Line In Select	Line In		Mode	Master	
17	Bypass	Disable		Bit Clock	Normal	
18	DAC	Select				
19	Side Tone	Disable		**Sampling Control**		
20	Side Tone Attenuation	-6,0 dB		Mode	Normal	
21				Oversampling	USB / Normal	
22	**Digital Audio Path Control**			ADC/DAC Sample rate	48/48 KHz	
23	High Pass Filter	Disable		Clock divider	None	
24	De-emphasis	Disable		Output Clock divider	None	
25	DAC Soft Mute	Disable				
26	DC offset	Clear		**Active Control**	Active	
27						
28						

Abbildung 22:Audio Codec Excel-Tool

Alle Einstellmöglichkeiten der vorhandenen Register wurden mit Drop Down Listen realisiert. Wird eine Einstellung geändert, findet die entsprechende Konfiguration der Register statt. Ist die gewünschte Konfiguration vorgenommen, müssen die Definitionen (oben rechts) in die Headerdatei kopiert werden. Nach erneuter Kompilierung wird der Codec mit den gewünschten Einstellungen initialisiert.

7 VHDL

Dieser Teil der Dokumentation soll den Aufbau und die Funktion des DSP-Teils detaillierter beleuchten. Dabei wird ein besonderer Fokus auf synthetisierbaren VHDL-Code gelegt, welcher frei von Latches ist. Des Weiteren werden Entitys, wo es sinnvoll ist, generisch gestaltet. Dadurch sind diese für andere Anwendungen skalierbar.

7.1 Bus Multiplexer

Ein generischer Bus Multiplexer lässt sich relativ leicht in VHDL realisieren. Die Ein- und Ausgänge werden über Vektoren realisiert *INPUT* und *OUTPUT*. Dem Ausgangsvektor wird ein Bereich des Eingangsvektors in Abhängigkeit von der anliegenden Adresse zugewiesen.

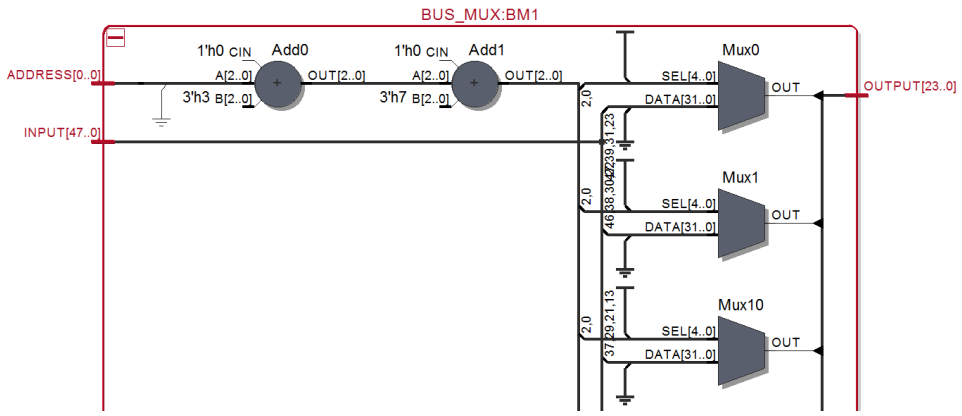
```
architecture BEHAVIOUR of BUS_MUX is

begin
      OUTPUT <= INPUT(
                  BUS_WIDTH + to_integer(unsigned(ADDRESS)) * BUS_WIDTH - 1
                  downto
                  to_integer(unsigned(ADDRESS)) * BUS_WIDTH);

      end BEHAVIOUR;
```

Listing 4: Bus Multiplexer Architektur

Die Synthese liefert das nachfolgende Ergebnis. Wie gewünscht, werden mehrere Multiplexer parallel geschaltet, welche einen gemeinsamen Adressbus besitzen.



Listing 5: Bus Multiplexer Registertransferebene

7.2 Aussteuerungsanzeige (Meterbridge)

Die Aussteuerungsanzeige unterscheidet zunächst, ob es sich um einen positiven, oder einen negativen Wert handelt. Diese Unterscheidung wird anhand des höchstwertigsten Bits getroffen. Danach wird der Wert der niederwertigen Bits zu

einem Integer Wert gewandelt und mit einem Grenzwert verglichen. Wird dieser Grenzwert überschritten, werden die Ausgänge entsprechend gesetzt.

```
architecture BEHAVIOUR of METER_BRIDGE is

begin

UPDATE : process (CLOCK)
begin
      if CLOCK'event and CLOCK = '1' then
            if(INPUT(INPUT_WIDTH-1)='0') then
                  if unsigned(INPUT(INPUT_WIDTH-2 downto 0)) > 8388606 then
                        OUTPUT <= "1111111111";
                  elsif unsigned(INPUT(INPUT_WIDTH-2 downto 0)) > 4717260
then
                        OUTPUT <= "0111111111";
                  ...
                  end if;
            else
                  if unsigned(INPUT(INPUT_WIDTH-2 downto 0)) > 8380219 then
                        OUTPUT <= "0000000000";
                  elsif unsigned(INPUT(INPUT_WIDTH-2 downto 0)) > 8362081
then
                        OUTPUT <= "0000000001";
                  ...
                  end if;
            end if;
      end if;
end process UPDATE;

end BEHAVIOUR;
```

Listing 6: Aussteuerungsanzeige Architektur

Mit der Testbench kann man erkennen, dass mit dem Überschreiten des jeweiligen Grenzwertes je ein zusätzlicher Ausgang auf high gezogen wird. Auf die Darstellung der Registertransferebene wird hier verzichtet, da diese zu viel Platz einnehmen würde.

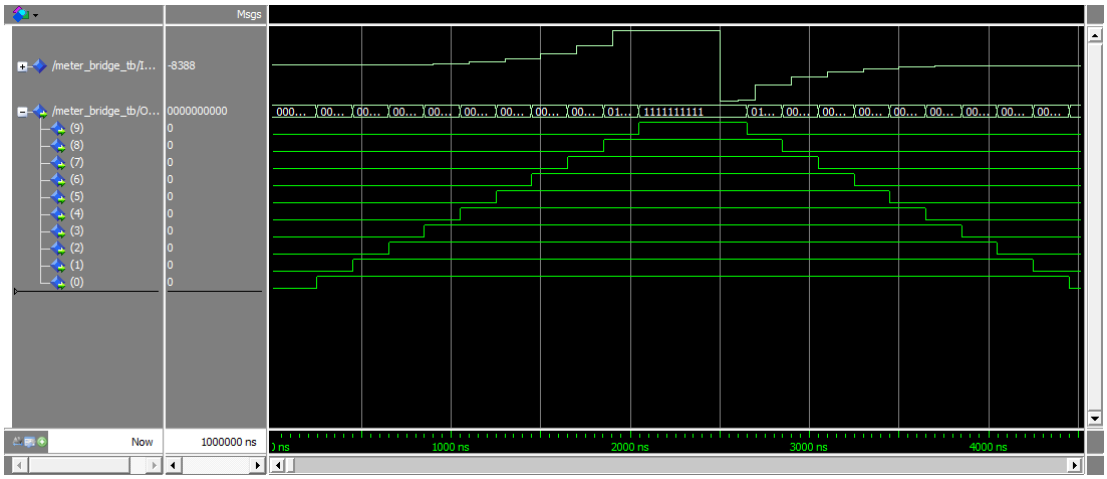


Abbildung 23: Aussteuerungsanzeige Testbench

7.3 Seriell zu Parallel Wandlung

Die Seriell zu Parallel Wandlung erfolgt mit Hilfe eines Schieberegisters und eines Bitzählers. Das Schieberegister ist so lange aktiv, wie die *CLOCK_ENABLE* Leitung high ist. Der Bitzähler zählt eine bestimmte Anzahl von Bits, nachdem dieser einen Trigger Impuls erhalten hat. Solange der Bitzähler zählt, befindet sich sein Ausgang *WORKING* auf high.

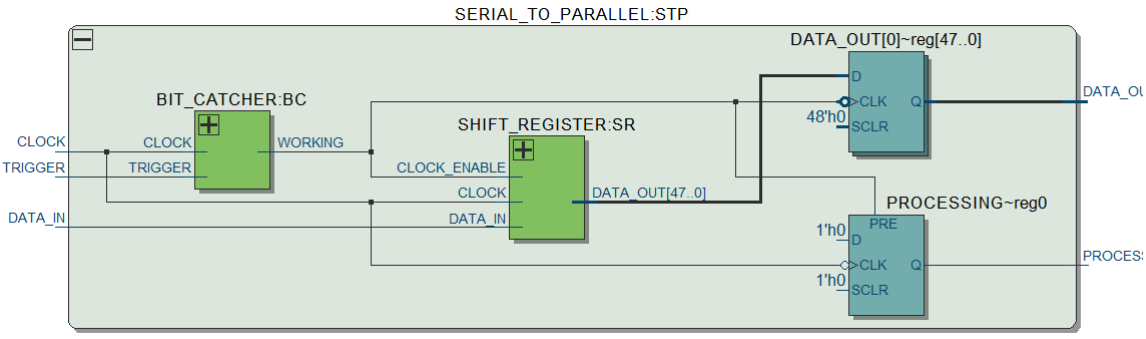


Abbildung 24: Seriell zu Parallel Registertransferebene

Das Verhalten des Wandlers ist Abbildung 25 zu sehen. Der Trigger Impuls kann asynchron erfolgen. Sobald der Trigger auftritt, wird der Ausgang *PROCESSING* auf high gesetzt. Daten am Eingang werden mit steigender Taktflanke ins Schieberegister übernommen. Sobald die gewünschte Anzahl an Bits eingegangen ist, wird der Ausgang *DATA_OUT* mit dem Wert des Schieberegisters gesetzt. Der *PROCESSING* Ausgang wird mit der nächsten fallenden Taktflanke zurückgesetzt.

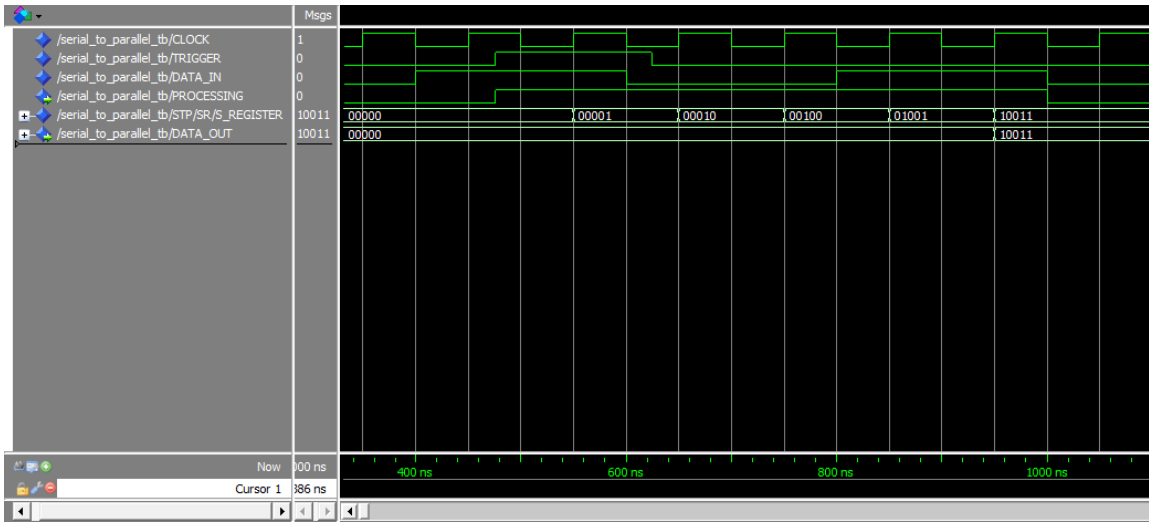


Abbildung 25: Seriell zu Parallel Testbench

Die in der Simulation gezeigte Funktionalität wird zum einen durch die Signalführung und zum anderen durch zwei Prozesse in der Architektur realisiert. Der *OUTPUT_DELAY* Prozess dient als Verzögerung für den *PROCESSING* Ausgang, damit dieser erst mit fallender Taktflanke zurückgesetzt wird. Solange das *BC_WORKING* Signal auf high ist, befindet sich der *PROCESSING* Ausgang ebenfalls auf high. Erst, wenn *BC_WORKING* auf low ist und eine fallende Taktflanke auftritt, wird der Ausgang zurückgesetzt. Der *UPDATE* Prozess dient zur Aktualisierung des *DATA_OUT* Ausgangs mit dem Wert des Schieberegisters. Der Prozess reagiert auf die fallende Flanke des *BC_WORKING* Signals, welches vom Bitzähler geliefert wird.

```
architecture BEHAVIOUR of SERIAL_TO_PARALLEL is

begin

	UPDATE : process(BC_WORKING)
	begin
		if(BC_WORKING'event and BC_WORKING='0') then
			DATA_OUT  <= SR_DATA_OUT;
		end if;
	end process UPDATE;

	OUTPUT_DELAY : process(CLOCK,BC_WORKING)
	begin
		if(BC_WORKING = '1') then
			PROCESSING <= '1';
		elsif(CLOCK'event and CLOCK='0') then
			PROCESSING <= '0';
		end if;
	end process OUTPUT_DELAY;


	end BEHAVIOUR;
```

Listing 7: Seriell zu Parallel Architektur

7.3.1 Schieberegister

In diskreter Schaltungstechnik besteht ein Schieberegister aus kaskadierten D-Flip-Flops. In VHDL lässt sich dies über ein Signal *S_REGISTER* erzeugen, welches innerhalb eines Prozesses *SHIFT* jedes Element des Registers um eine Zelle weiterschiebt und das erste Bit mit dem Eingangswert belegt. In diesem Fall reagiert der Prozess auf eine positive Taktflanke des *CLOCK* Eingangs.

```
architecture BEHAVIOUR of SHIFT_REGISTER is

signal S_REGISTER : std_logic_vector (BIT_WIDTH-1 downto 0) := (others =>
'0');

begin

      SHIFT : process(CLOCK)
      begin
            if(CLOCK'event and CLOCK = '1' and CLOCK_ENABLE='1') then
                  for I in 0 to BIT_WIDTH-2 loop
                        S_REGISTER(I+1) <= S_REGISTER(I);
                  end loop;
                  S_REGISTER(0) <= DATA_IN;
            end if;

      end process SHIFT;

DATA_OUT <= S_REGISTER;


      end BEHAVIOUR;
```

Listing 8: Schieberegister Architektur

Die Synthese liefert das gewünschte Ergebnis.

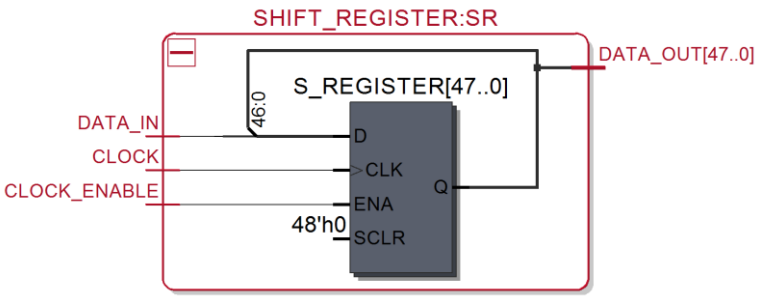


Abbildung 26: Schieberegister Registertransferebene

Und die Simulation zeigt ebenfalls das gewünschte Verhalten. Solange der *CLOCK_ENABLE* Eingang auf high liegt, nimmt das Schieberegister Daten an. Sobald dieser Eingang auf low ist, wird der *CLOCK* Eingang ignoriert und keine weiteren Daten akzeptiert.

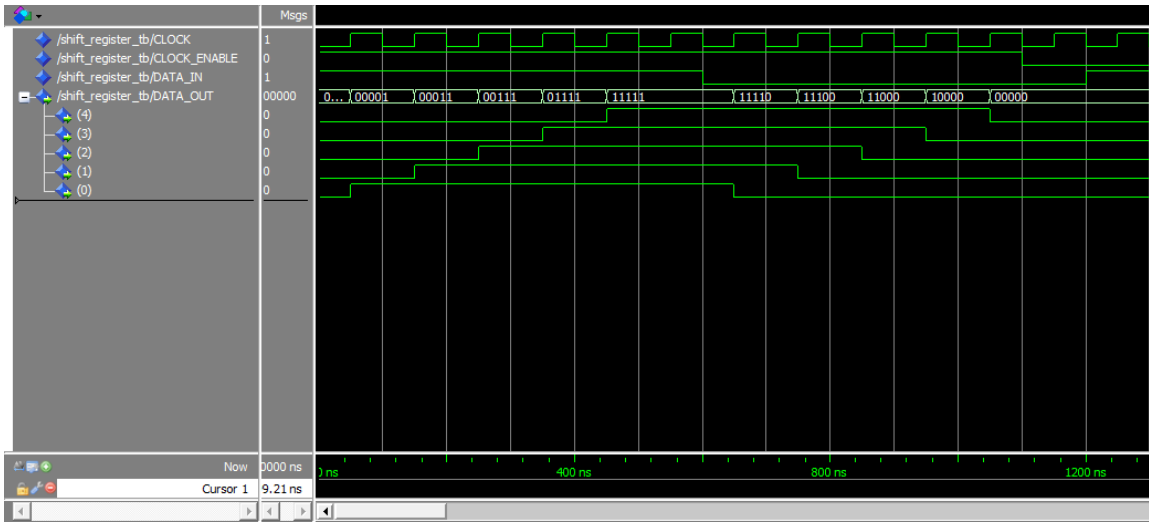


Abbildung 27: Schieberegister Testbench

7.3.2 Bitzähler

Der Bitzähler wird benötigt, da der Audio Codec während der seriellen Übertragung das Taktsignal kontinuierlich mitlaufen lässt. Zudem gibt der Codec ein Synchronisierungssignal aus, welches das erste Bit einer Übertragung markiert. Der Bitzähler reagiert auf dieses Synchronisierungssignal und zählt eine zuvor definierte Anzahl von Bits. Sobald das Synchronisierungssignal auftritt, ist der Ausgang des Bitzählers auf high, bis dieser die gewünschte Anzahl an Bits detektiert hat. Danach wird der Ausgang wieder zurückgesetzt. Somit kann der Bitzähler als *CLOCK_ENABLE* Quelle für ein Schieberegister dienen.

Die Architektur beinhaltet einen Prozess, welcher sensitiv auf das *TRIGGER* und das *COUNTER_OUT* Signal reagiert. Sobald eine Änderung am *TRIGGER* Signal auftritt und *TRIGGER* auf high liegt, wird der Zähler freigegeben und das *WORKING* Flag auf high gesetzt. Sobald *COUNTER_OUT* eine positive Flanke erzeugt, wird der Zähler gestoppt und das *WORKING* Flag zurückgesetzt.

```
architecture BEHAVIOUR of BIT_CATCHER is

begin

	COUNT : process(TRIGGER,COUNTER_OUT)
	begin
		if(TRIGGER='1') then
			COUNTER_RESET <= '0';
			WORKING <= '1';
		elsif(COUNTER_OUT'event and COUNTER_OUT='1') then
			COUNTER_RESET <= '1';
			WORKING <= '0';
		end if;

	end process COUNT;


end BEHAVIOUR;
```

Listing 9: Bitzähler Architektur

Die Simulation zeigt das gewünschte Verhalten. Der Trigger kann asynchron eintreffen und der Zähler zählt positive Taktflanken. Sobald die letzte positive Taktflanke auftritt, wird der *WORKING* Ausgang unmittelbar zurückgesetzt.

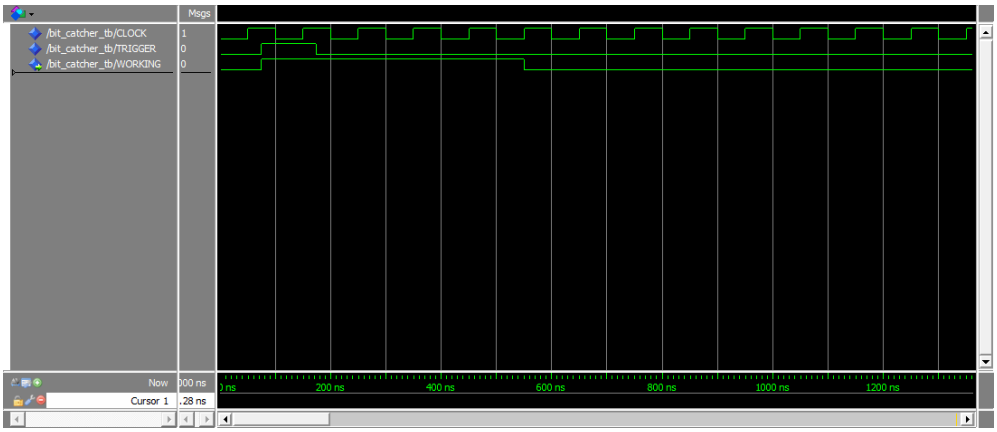


Abbildung 28: Bitzähler Testbench

Die Synthese liefert das nachfolgende Ergebnis. Das Zähler Objekt wird im nachfolgenden Abschnitt genauer beschrieben.

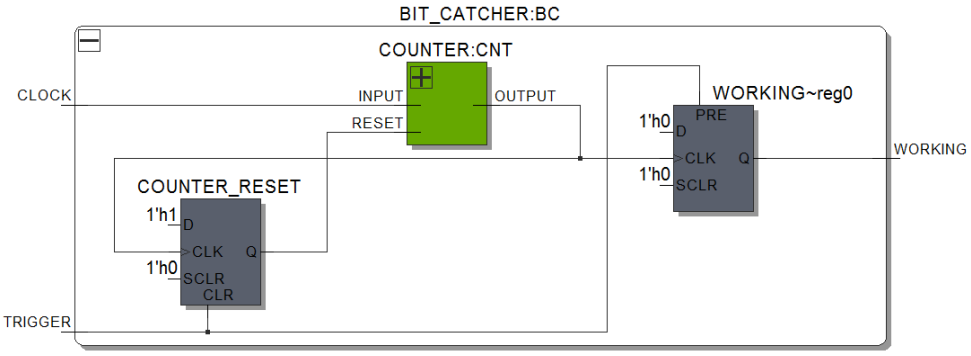


Abbildung 29: Bitzähler Registertransferebene

7.3.2.1 Zähler

Der Zähler zählt eine bestimmte Anzahl von positiven Taktflanken an seinem *INPUT* Eingang und invertiert den *OUTPUT* Ausgang, sobald die Anzahl erreicht wurde. Nach der Invertierung beginnt dieser sofort wieder mit dem Zählen und invertiert den Ausgang erneut bei Erreichen der Anzahl an Taktflanken. Der Zähler kann somit als Taktteiler angesehen werden. In VHDL wird dieser mit einem Prozess realisiert, welcher auf das *RESET* und das *INPUT* Signal reagiert. Solange sich *RESET* auf high befindet, ist der Ausgang *OUTPUT* low und der Zähler auf null gesetzt. Ist der *RESET* Eingang auf low und tritt eine positive Taktflanke an *INPUT* auf, wird der Zähler inkrementiert. Hat der Zähler sein Limit erreicht, wird der Ausgang invertiert und der Zähler auf null zurückgesetzt.

```
architecture BEHAVIOUR of COUNTER is

begin

	CNT : process(RESET, INPUT)
	begin
		if(RESET='1') then
			STATE   <= '0';
			COUNTER <= 0;
		elsif INPUT'event and INPUT = '1' then
			if COUNTER=LIMIT-1 then
				STATE   <= not STATE;
				COUNTER <= 0;
			else
				COUNTER <= COUNTER+1;
			end if;
		end if;

	end process CNT;

OUTPUT <= STATE;


	end BEHAVIOUR;
```

Listing 10: Zähler Architektur

Die Simulation bestätigt das gewünschte Verhalten. Das *RESET* Signal kann asynchron erfolgen und setzt den Zähler unmittelbar zurück.

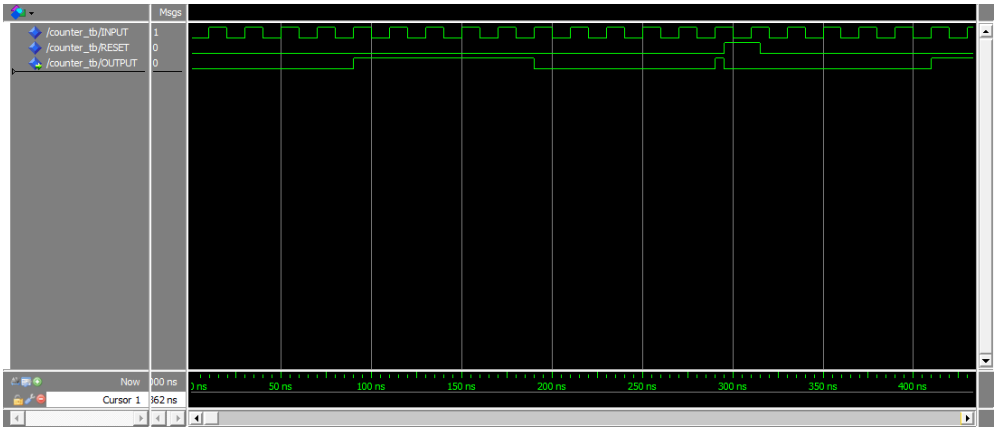


Abbildung 30: Zähler Testbench

Das Syntheseergebnis ist ebenfalls zufriedenstellend, da der Zähler ohne Latches synthetisiert worden ist.

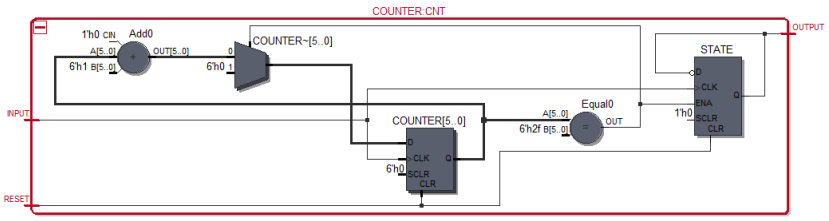


Abbildung 31: Zähler Registertransferebene

7.4 Parallel zu Seriell Wandlung

Die Parallel zu Seriell Wandlung ist relativ einfach zu realisieren. Da die Daten Parallel vorliegen, muss lediglich über einen Multiplexer bestimmt werden, welches Element des Vektors am Ausgang ausgegeben werden soll. Da der Aufbau eines Multiplexers zuvor bereits beschrieben worden ist, wird hier lediglich die Registertransferebene betrachtet.

Wie bei der Seriell zu Parallel Wandlung wird der Takt und ein Synchronisierungsimpuls vom Audio Codec vorgegeben. Damit zum richtigen Zeitpunkt die korrekte Adresse am Multiplexer anliegt, wird wieder ein Bitzähler verwendet. Dieser gibt am Ausgang den aktuellen Zählerstand am Ausgang aus.

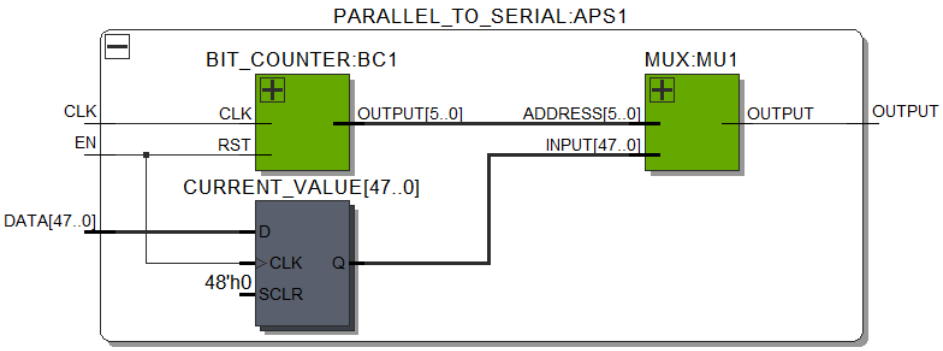


Abbildung 32: Parallel zu Seriell Wandlung Registertransferebene

8 Filterdesign mit Matlab

Um die Erstellung des FIR-Filters nicht unnötig komplex zu gestalten wird dieser mittels Matlab erstellt. Worauf man dabei achten sollte wird in diesem Abschnitt behandelt. Der Filter Designer ist Teil der Signal Processing Toolbox, welche für das Design vorhanden sein muss.

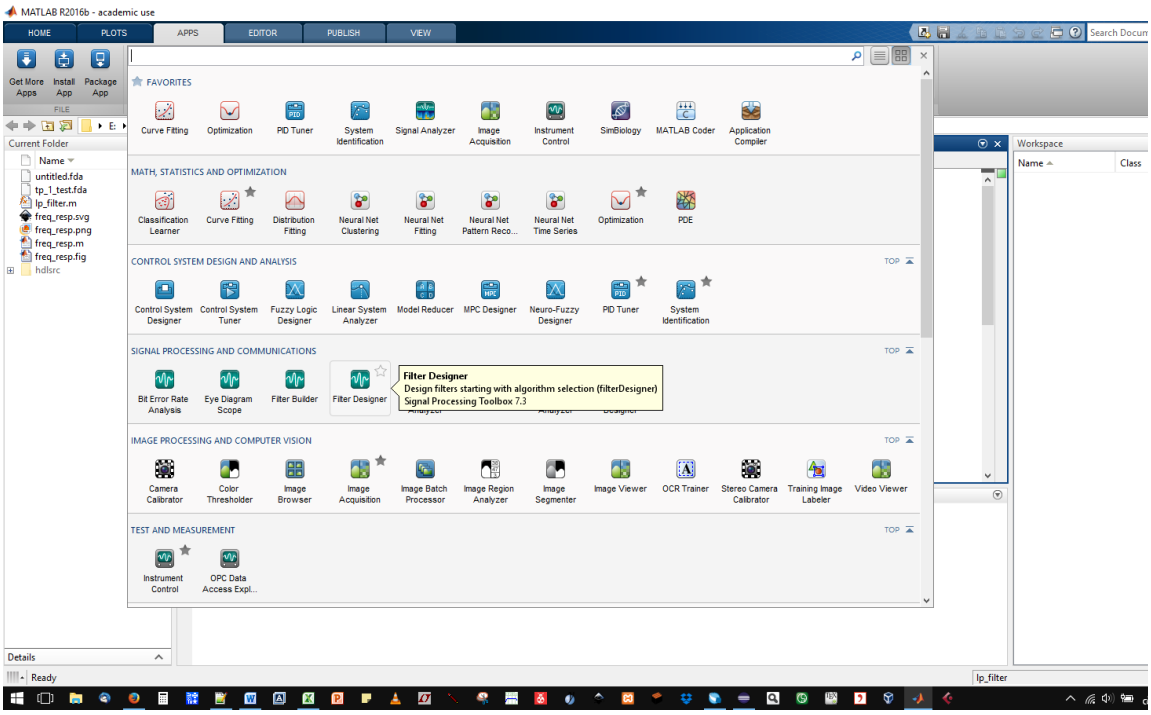


Abbildung 33: Matlab Filter Designer

Nach Eingabe der gewünschten Filtereigenschaften und Klick auf *Design Filter* erhält man zunächst den Frequenzgang. Dieser stellt den idealen Frequenzgang dar, welcher sich ergibt, wenn keine Rundungsfehler gemacht werden. Da wir allerdings in unserer Auflösung beschränkt sind, werden sich Rundungsfehler ergeben. Um die Auswirkungen dieser Fehler abschätzen zu können, muss der Filter quantisiert werden.

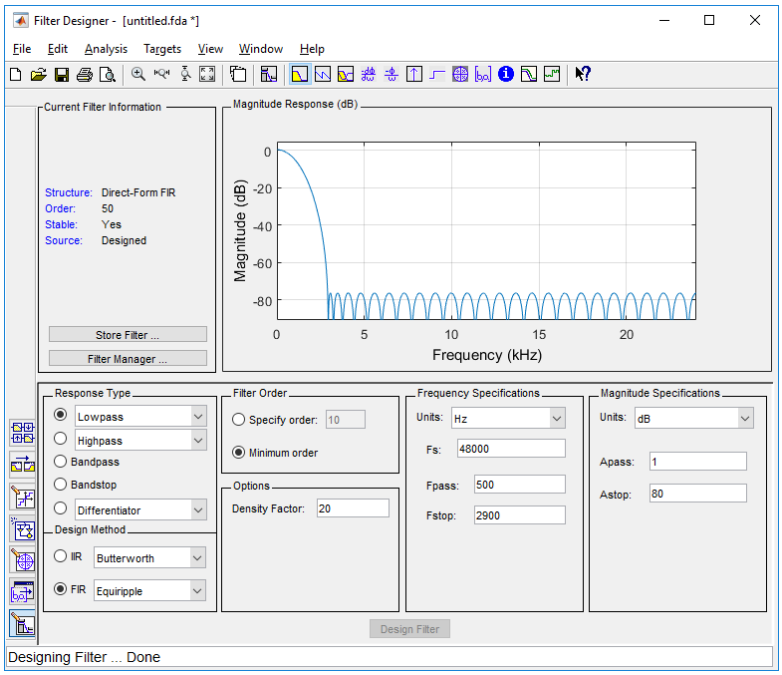


Abbildung 34: Filter Designer Tiefpass

Am linken Rand befindet sich der Button *Set quantization parameters*, hier können beliebige Einstellungen vorgenommen werden um einen Vergleich zwischen einem Referenzfilter und einem quantisierten Filter zu betrachten. Hier sollte darauf geachtet werden, dass die Wortlänge des Eingangs- und Ausgangssignals der Wortlänge der Zielarchitektur entspricht. Im Fall des Projektes arbeitet der Audio Codec mit 24 Bit Wörtern. Ist das Ergebnis zufriedenstellend, kann über *Targets → Generate HDL…* ein VHDL Filter mit den gewünschten Parametern erzeugt werden.

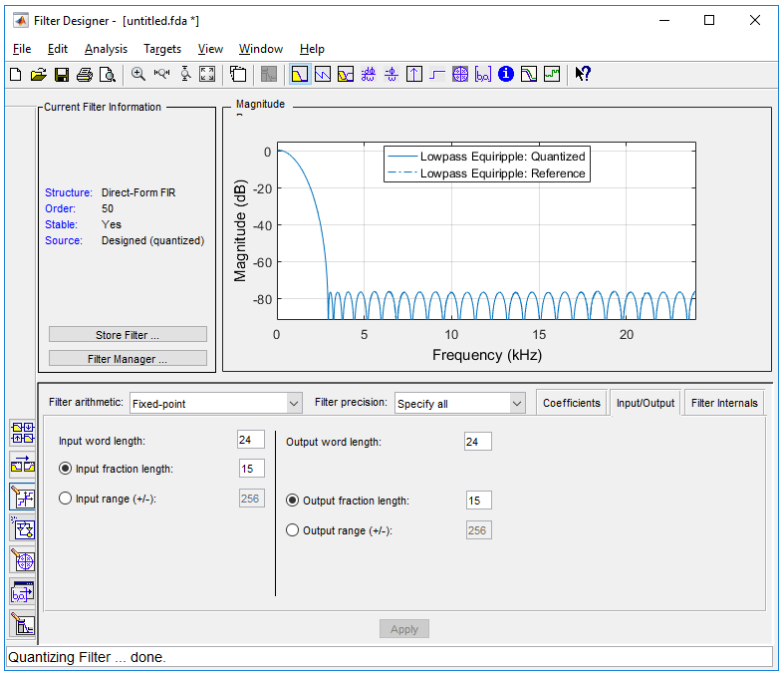


Abbildung 35: Quantisierungsparameter

Zuerst wird die Filterarchitektur festgelegt. Da ein FPGA eine begrenzte Anzahl an Multiplikatoren vorhält ist es sinnvoll, diese sparsam einzusetzen. Daher wird hier die Architektur auf *Fully serial* gesetzt. Dies bewirkt, dass nur ein Multiplikator verwendet wird und die Ergebnisse jeweils zwischengespeichert werden. Zudem empfiehlt es sich, die Option *Optimize for HDL* zu wählen.

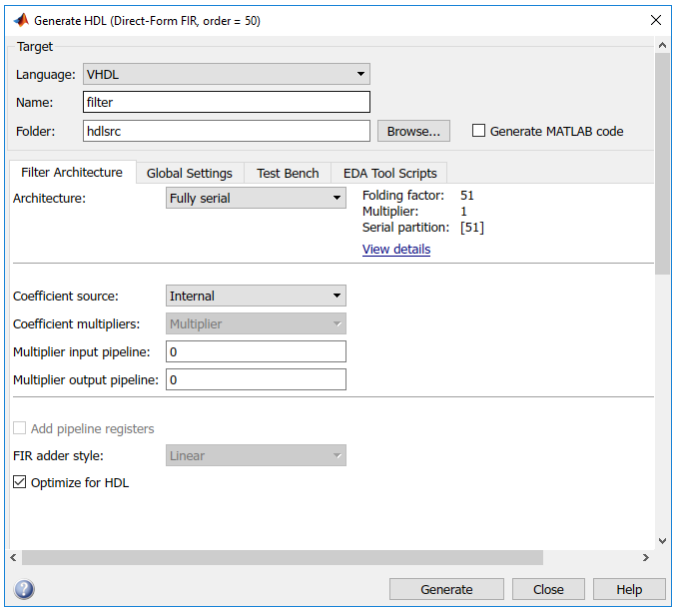


Abbildung 36: Filterarchitektur

Unter *Global Settings → Ports* kann unter *Input data type* und *Output data type* das Zahlenformat eingestellt werden. Da der Audio Codec ausschließlich mit Vorzeichenbehafteten Datentypen arbeitet, wird hier *signed/unsigned* gewählt.

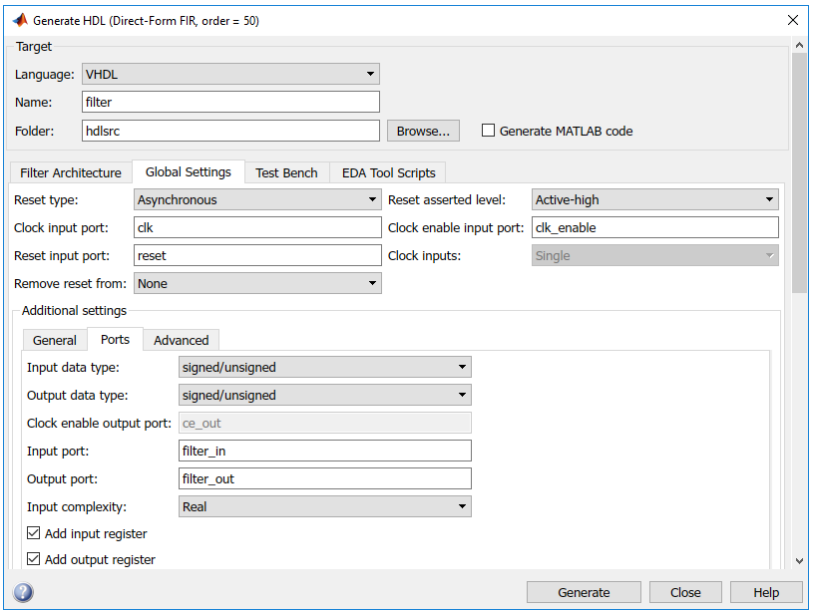


Abbildung 37: Datentypen

Mit einem Klick auf Generate wird der VHDL Code erzeugt. Standardmäßig wird zum Filter eine Testbench erzeugt. Mit Hilfe dieser kann man sich die Verwendung des Filters und dessen Funktionsweise in ModelSim visualisieren lassen.

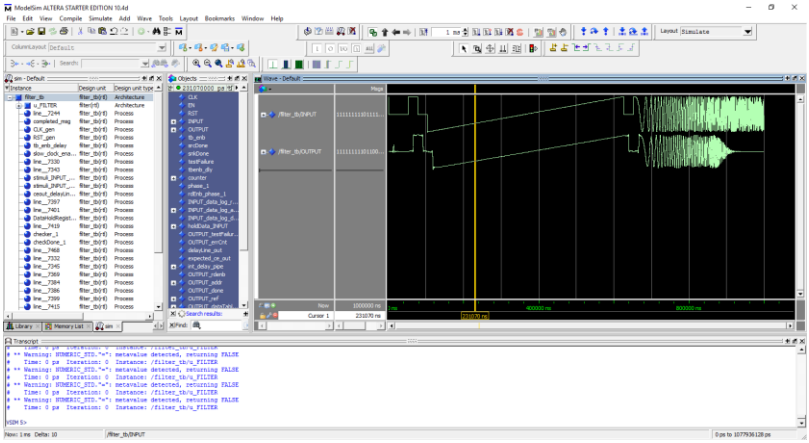


Abbildung 38: Filter Testbench in ModelSim