Belegarbeit

Schaltkreisentwurf

FM-Demodulation auf einem Intel FPGA

David Lohner, Moritz Albert

Leipzig, den 18. März 2023

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Aufgabenstellung

Ziel dieses Belegs ist es einen FPGA-basierten FM-Radioempfänger zu bauen. Dazu sollen digitale komplexe Samples per UART an den FPGA gestreamt werden. Im FPGA soll das FM-Signal demoduliert und aufbereitet werden. Die Augangssamples erreichen den Host-PC wieder über eine UART-Schnittstelle. Die empfangenen Daten sollen dann validiert und hörbar gemacht werden. Verwendet wird ein Max1000 Entwicklungsboard der Firma „Arrow“.

Die Dokumentation der Belegarbeit untergliedert sich in drei Abschnitte. Zu Beginn wird das I/O-Management in Abschnitt 2 erläutert. Abschnitt 3 geht anschließend auf die Businesslogik ein. Dabei werden die zur Datenverarbeitung und Datenaufbereitung entwickelten Module beschrieben und im Signal-Tap-Analyzer der Software „Quartus-Prime“ betrachtet . Abschließend werden die Ausgangsdaten in Abschnitt 4 mit der Python-Implementierung verglichen.

2 I/O-Management

2.1 Datenvorbereitung

Der Real- und Imaginärteil der zu demodulierenden Samples ist in Form einer Txt-Datei zur Verfügung gestellt worden. Obwohl die Daten den 8-Bit signed Integer Wertebereich nicht überschreiten, wurden 16-Bit zum Streaming dieser Daten verwendet. Dies macht es möglich bei der konjugiert-komplexen Multiplikation den Wertebereich nicht auf 32-Bit erweitern zu müssen. Ein komplexes Sample wird als 32-Bit Zahl übertragen. Hierbei sind die Bits [31:16] der Realteil und die Bits [15:0] der Imaginärteil. Da die Daten über eine UART-Schnittstelle an den FPGA gestreamt werden, wird ein komplexes Sample von vier aufeinanderfolgenden Bytes im UART-Buffer repräsentiert. Das interne Zusammenfügen der Bytes wird weiter in Abschnitt 2.3 erläutert.

2.2 UART-Streaming

Die Architektur des UART-Streamings im FPGA ist aus den zur Verfügung gestellten Vorlesungsunterlagen übernommen worden. Der UART-Stream erfolgt mit einer Baudrate von 115200 Hz. Nach jedem gesendeten Byte wird ein Stopbit gesendet. Auf ein Paritybit wurde verzichtet. Um möglichen Fehldetektionen, die dem Charakter einer asynchronen Schnittstelle zu Grunde liegen, entgegenzuwirken, wird der Stream im FPGA mit einer Taktrate von 230400 Hz abgetastet. In den Modulen „uart_rx “ und „uart_tx “ werden Anfang und Ende jedes Streams detektiert, sowie die einzelnen Bits als Byte für die weitere Verarbeitung zur Verfügung gestellt. Zudem wird im Modul „uart_rx “ nach dem vollständigen Empfang eines Bytes ein Validsignal generiert, welches die Datenverabeitung taktet. Da mit Zählregistern gearbeitet wird musste das Modul leicht verändert werden. Zuvor ist durch ein Reset das Register „uart_r “ zu Null gesetzt worden. Da der Übergang

von High zu Low auch den Start der UART-Übertragung ankündigt, führte ein Reset zum ungewollten Detektieren eines UART-Bytes. Dies konnte jedoch leicht korrigiert werden, indem das Register im Reset-Fall auf logisch Eins gesetzt wird. Auf eine detailliertere Beschreibung dieser beiden Module wird jedoch verzichtet, da diese im Rahmen dieses Beleges nicht selbst entworfen wurden.

2.3 Merge & Split

Da immer vier aufeinander folgende Bytes ein komplexes Sample repräsentieren (siehe Abschnitt 2.1) müssen diese für die eigentliche Verarbeitung zusammengeführt werden. Zudem muss nach der Businesslogik das Ausgangssample wieder in einzelne Bytes aufgeteilt werden, um diese Taktgleich an den Rechner zurück zu schicken. Dies wird über die Module „merge_data “ und „split_data “ bewerkstelligt. Die Toplevel-Konfiguration der Module ist in Abbildung 1 dargestellt. Die Portliste der beiden Module ist in den Tabellen 1 und 2 ersichtlich.

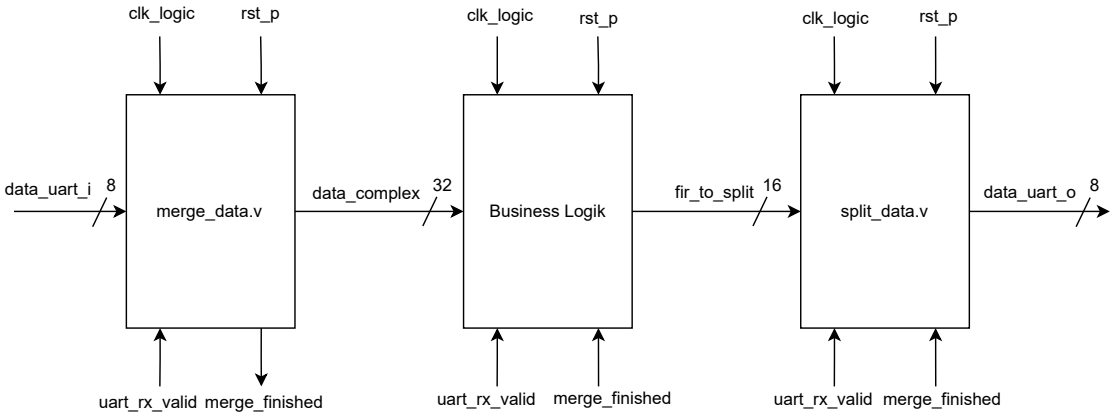

Abbildung 1: Toplevel-Konfiguration der Module „merge_data.v“ und „split_data.v“

Beide Module werden mit dem UART-Streaming Takt „clk_logic“ betrieben. Das Reset wird über den User-Button an PIN E6 ausgelöst. Das Startsignal wird von dem Modul „uart_rx “ generiert. Ohne dieses könnte das zum Streaming synchrone Zusammenführen und Aufteilen der Daten nicht durchgeführt werden. Jedes Mal, wenn ein neues Byte des Streams zur Verfügung steht, wird das Schieberegister im Modul „merge_data.v“ mit dem aktuellen Ausgangssample des Moduls „uart_rx.v“ beschrieben. Zeitgleich wird ein Zählerregister inkrementiert. Erreicht dieses den Wert Drei ist ein vollständiges komplexes Sample empfangen worden. Ein erfolgreiches Zusammenführen wird den anderen Modulen über das Steuersignal „merge_finished_o“ mitgeteilt. Damit das Zählerregister synchron zum Datenempfang arbeitet, muss das Reset vor Betrieb des FM-Demodulators betätigt werden.

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
start_i	Kündigt den erfolgreichen Empfang eines Datenbytes über UART an	1-Bit	I
data_uart_i	Eingangsdaten	8-Bit	I
merge_finished_o	Steuersignal um das erfolgreiche Zusammenführen der Bytes zu bestätigen	1-Bit	O
data_o	Komplexes Ausgangssample bestehend aus Real- [31:16] und Imanginärteil [15:0]	32-Bit	O

Tabelle 1: Portliste des Moduls „merge_data.v“

Nach der Verarbeitung in der Business-Logik ist das Ausgangssample in Form eines 16-Bit signed Integers verfügbar. Damit das Senden der Daten in der selben Rate wie das Empfangen erfolgen kann, wird das Ausgangssignal wieder in ein 32-Bit Format gebracht. Wird das Steuersignal „merge_finished_o“ detektiert, folgt das Beschreiben des Speichers mit dem aktuellen Ausgangssample der Businesslogik. Dabei werden die Bits [31:16] mit dem Wert Null beschrieben und die Bits[15:0] mit den Nutzdatenbytes. Ein internes Zählerregister wird synchron zum Startsignal inkrementiert und der Speicherwert an der Adresse des Zählerregisters an den Ausgang gelegt. So wird zuerst das höchste Byte über UART übertragen. Dies sowie die zwei informationslosen „Nullbytes“ werden bei der Interpretation der Daten im Python-Code berücksichtigt (siehe Anhang 1, Zeilen 41 bis 49).

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
start_i	Bedingung für die Zähler Inkrementierung	1-Bit	I
merge_finished_i	Bedingung für das Updaten des internen Speichers	1-Bit	I
data_i	Ausgangssignal der Datenverarbeitungslogik	signed 16-Bit	I
data_uart_o	Ausgangsbyte für die UART-Übertragung an den PC	8-Bit	O

Tabelle 2: Portliste des Moduls „split_data.v“

3 Datenverarbeitung

Die Datenverarbeitungslogik setzt sich aus den Modulen „conj_c_mult.v“, „avg_128.v“ und „fir_17.v“ zusammen. So teilt sich die Businesslogik in ein Demodulationsmodul und zwei Datenaufbereitungsmodule (DC-Offset Entfernung und Filterung) auf. Deren Funktionsweise wird in den Abschnitten 3.1 und 3.2 noch genauer erläutert. Da für die Implementierung des FM-Demodulators keine Schleifen benötigt werden, wurde das Prinzip der „Pipeline“-Datenverarbeitung angewendet. Dies bedeutet, dass die Module seriell miteinander verschaltet sind und somit nach einem modulspezifischen Delay in jedem Takt ein valides Sample am Ausgang der Logik anliegt. Die Toplevel-Konfiguration der Datenverarbeitungslogik ist in Abbildung 2 ersichtlich. Die Portlisten der einzelnen Module sind in den Tabellen 3, 4 und 5 zu finden.

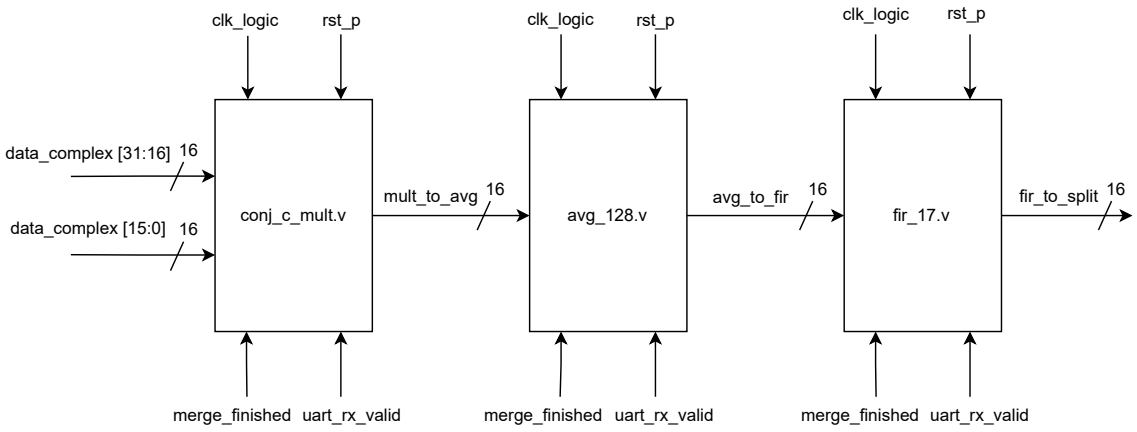

Abbildung 2: Toplevel-Konfiguration der Businesslogik

auch hier werden alle Module mit der Taktleitung „clk_logic“ betrieben. Das Resetsignal wird über den User-Button ausgelöst. Das Signal „merge_finished“ wird im Modul „merge_data_.v“ generiert. Das Startsignal „uart_rx_valid“ entsteht im Modul „uart_rx.v“ nach der Übertragung des Stopbits eines jeden UART-Transfers.

3.1 Demodulation

Das zu demodulierende komplexe FM-Empfangssignal ist bereits in das Basisband gemischt worden. Die Abtastfrequenz des Signals beträgt $200kHz$.

Um die informationstragende Phasenfunktion des Signals zu gewinnen kann folgende Gleichung 1 verwendet werden.

$$s_{BB}[k] = Im\{s_{FM,BB}[k] \cdot *s_{FM,BB}[k-1]\} \tag{1}$$

Diese approximiert die Phasenfunktion unter der Annahme, dass der Frequenzfehler f_0 als der Phasenhub zwischen zwei Samples hinreichend klein ist. Gleichung 1 bildet somit die Grundlage des Moduls „conj_c_mult.v“. Da im Allgemeinen die Multiplikation zweier komplexer Zahlen als $(a + jb) \cdot (c + jd)$ ausgedrückt werden kann, lässt sich der Imaginärteil mithilfe der Summe aus den beiden Konstanten K1 und K3 berechnen. Deren Berechnung im Modul ist der Gleichung 2 zu entnehmen.

$$\begin{aligned} K_1 &= real[k] \cdot (real[k-1] - imag[k-1]) \\ K_3 &= real[k-1] \cdot (imag[k] - real[k]) \\ s_{BB}[k] &= K_1 + K_3 \end{aligned} \tag{2}$$

Da die Multiplikation mit dem konjugiert-komplexen letzten Eingangssample durchgeführt werden muss, ist dies bei der Berechnung der Konstante K_1 in Gleichung 2 berücksichtigt worden. So wird der letzte Imaginärteil vom letzten Realteil subtrahiert

und nicht auf diesen aufaddiert. Im Modul wird dieser Schritt in das Beschreiben der Variable „last_in_imag_r“ verlagert. Des Weiteren arbeitet das Modul synchron zum Steuersignal „merge_finished_i“ und beschreibt die Speicherregister bei der Detektion einer High-Flanke. Da die Eingangswerte den Wertebereich einer 8-Bit signed Integerzahl nicht überschreiten ($max\{s_{FM,BB}\} = 67$), reichen 16-Bit, um das Ergebnis auf der Ausgangsleitung „demod_o“ vollständig zu repräsentieren. Um die korrekte Funktionsweise des Moduls zu bestätigen, ist ein Ausschnitt aus dem Signal-Tap-Analyzer in Abbildung 3 dargestellt. Die Portliste des Moduls ist in Tabelle 3 zu sehen. Der Signal-Tap-Analyzer wird für alle folgenden Analysen mit dem Startsignal „uart_rx_valid“ getriggert und der Clock „clk_logic“ getaktet. Betrachtet wird hier der Zeitabschnitt zwischen 400 bis 560 Takten, welcher die Verarbeitung von zwei komplexen Eingangssamples abbildet (Sample 5 und 6 der Eingangsdaten). Dies hat den Hintergrund, die Analyse des Signal-Taps so übersichtlich wie möglich zu gestalten.

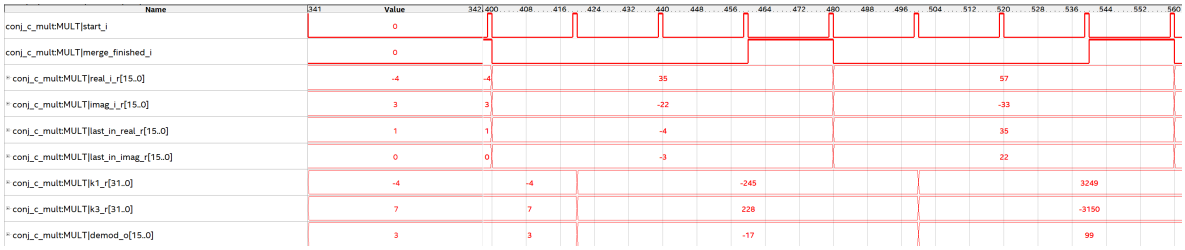

Abbildung 3: Ausschnitt aus dem Signal-Tap-Analyzer für das Modul „conj_c_mult.v“

Die Abbildung bestätigt zum einen, dass in den Registern „last_in_imag_r“ und „last_in_real_r“ das konjugiert komplexe letzte Eingangssample gespeichert ist (Zeitabschnitte: 400-480, 480-560). Zum anderen ist klar ersichtlich, wie die Signale „uart_rx_valid“(start_i) und „merge_finished_i“ die interne Verarbeitung takten. Das Aktualisieren der Buffer erfolgt immer, wenn beide Signale high sind. Das darauf folgende Ausgangssample wird bei der Detektion des nächsten Startsignals berechnet (Zeitpunkte 419, 499) und ist am Ausgang bis zum Aktualisierungszeitpunkt verfügbar.

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
start_i	Bedingung für Berechnung des Ausgangssamples und Speicheraktualisierung	1-Bit	I
merge_finished_i	Bedingung für die Speicheraktualisierung	1-Bit	I
real_i	Realteil des aktuellen Eingangssamples	signed 16-Bit	I
imag_i	Imaginärteil des aktuellen Eingangssamples	signed 16-Bit	I
demod_o	demoduliertes Basisbandsample	signed 16-Bit	O

Tabelle 3: Portliste des Moduls „conj_c_mult.v“

3.2 Aufbereitung

Nach der Demodulation werden das DC-Offset sowie die störenden Frequenzen aus dem Signal entfernt. Dies wird in den folgenden Unterpunkten 3.2.1 und 3.2.2 näher erläutert.

3.2.1 Averaging

Dadurch, dass das Signal nicht perfekt in die Nulllage gemischt wurde, entsteht ein DC-Offset im sonst mittelwertfreien Signal. Um dieses aus dem demodulierten Signal zu entfernen, wird im Modul „avg_128.v“ der Mittelwert der letzten 128 Eingangssamples berechnet und vom aktuellen Sample abgezogen. Das Register „sum“ berechnet sich aus der Addition des Registers „sum_r“ und dem Register „data_i_r“. Davon wird dann der Wert der zuletzt an der Speicherstelle des Zählregisters stand abgezogen. So wird die Summe in jedem Takt aktualisiert. Der gemittelte Wert des Summenregisters „sum_r“ bildet das Ausgangssignal des Moduls. Das Teilen durch die Bufferlänge wird hierbei von einem Rechtsshift um sieben der Variable „sum_r“ umgesetzt. Um Fehler, die durch das Shiften einer signed Integerzahl entstehen, entgegenzuwirken (bsp: $-13 >>> 7 = -1 != 0$), wird das MSB des Summenregisters abgefragt und der Fehler im Fall einer negativen Zahl korrigiert. Dies ist in dem folgenden Codeauschnitt (Gleichung 3) zu sehen.

$$data\_o = (sum\_r[19])\ ?\ (data\_i\_r - (sum\_r >>> 7) - 1) : (data\_i\_r - (sum\_r >> 7)) \quad (3)$$

Abbildung 4 zeigt nun die Analyse des Moduls im Signal-Tap-Analyzer. Da es aufgrund der Speichergröße nicht möglich ist, den gesamten Bufferinhalt im Signal-Tap anzuzeigen, wurde sich hier auf dessen ersten acht Einträge beschränkt.

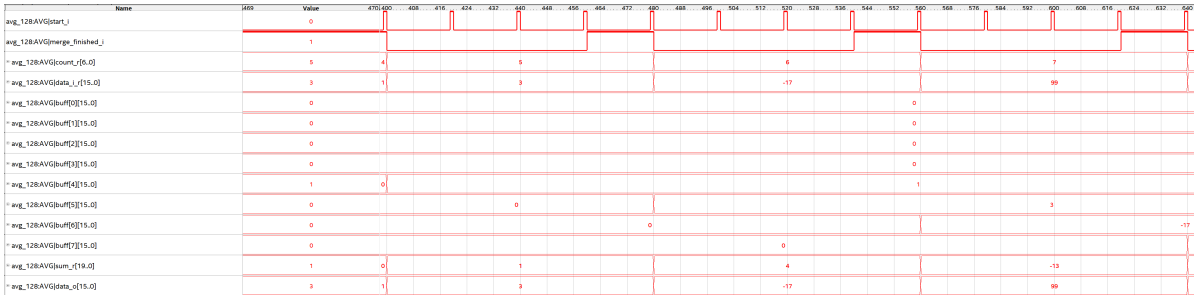

Abbildung 4: Ausschnitt aus dem Signal-Tap-Analyzer für das Modul „avg_128.v“

Die Abbildung zeigt den Zeitabschnitt zwischen 400 und 640 Takten. Es wird deutlich, dass das Schreiben der Eingangswerte in den Buffer, das Inkrementieren des Zählregisters „count_r“ sowie die Summierung synchron zu den bereits erwähnten Steuersignalen ausgeführt wird. Nach 480 Takten hat das aktuelle Eingangssample den Wert -17. Die Summe der letzten Eingangswerte beträgt 4. Da $int16\{\frac{4}{128}\} = 0$ gilt, wird von dem aktuellen Ausgangssample der Wert Null abgezogen, weshalb sich dieser zu -17 berechnet. Die Portliste dieses Moduls ist in Tabelle 4 auf der nächsten Seite zu sehen.

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
start_i	Bedingung für das Updaten der internen Register	1-Bit	I
merge_finished_i	Bedingung für das Updaten der internen Register	1-Bit	I
data_i	Demoliertes Eingangssample	signed 16-Bit	I
data_o	Mittelwertfreies Ausgangssample	signed 16-Bit	O

Tabelle 4: Portliste des Moduls „avg_128.v“

3.2.2 Filterung

Da das Signal mit einer Abtastrate von 200 kHz digitalisiert wurde, können Frequenzen bis 100 kHz repräsentiert werden. Da der Mensch jedoch nur in der Lage ist, Frequenzen bis maximal 20 kHz wahrzunehmen, erscheint es sinnvoll, eine Bandbegrenzung des Ausgangssignals vorzunehmen. Dafür wurde ein FIR-Filter mit einer Impulsantwortlänge von 17 Samples verwendet. Die Grenzfrequenz des Filters beträgt 10 kHz. Dies hat den Hintergrund, dass Frequenzen zwischen $10kHz$ und $20kHz$ die Verständlichkeit von menschlicher Sprache kaum beeinflussen. Zudem ist die Flankensteilheit eines FIR-Filters mit 17 Filtertaps nicht hoch genug, weshalb die niedrigere Grenzfrequenz zur besseren Unterdrückung der Signalanteile über 20 kHz beiträgt (siehe Abbildung 5). Da das Ergebnis der Demodulation ein Nachrichtenbeitrag ist (nur Sprache, keine Musik), ist die Einstellung des Filters legitim. Die Filterkoeffizienten sind mithilfe der Funktion „firwin()“ aus der Python-Bibliothek „Scipy“ berechnet worden. Die halblogarithmische Darstellung der Übertragungsfunktion ist in der folgenden Abbildung 5 veranschaulicht.

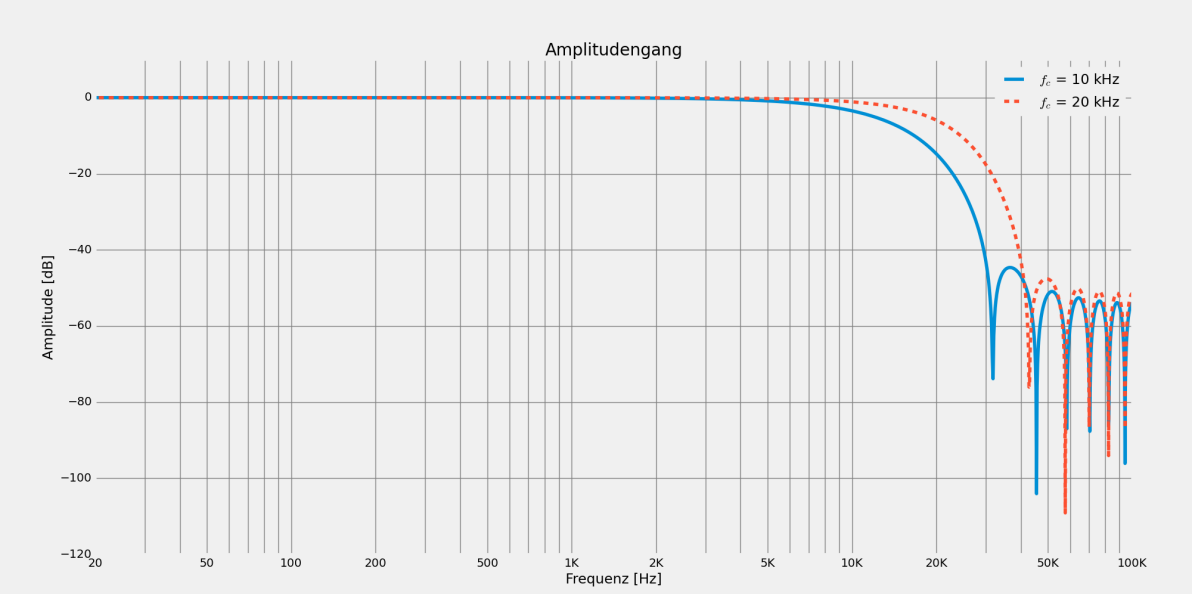

Abbildung 5: Amplitudengänge des FIR-Filters für die Grenzfrequenzen $10kHz$ und $20kHz$

Die rote Kurve zeigt hierbei den Frequenzgang des FIR-Filters mit einer Grenzfrequenz von 20 kHz und die blaue Kurve den FIR-Filter mit der Grenzfrequenz von 10 kHz. Um den Filter auf dem FPGA zu realisieren, mussten die Filterkoeffizienten in ein Festkommaformat gewandelt werden. So wurde das Format 0.16 zum Konvertieren der Filtertaps verwendet. Da alle Filterkoeffizienten positiv und kleiner als eins sind, konnte auf ein Vorzeichen, sowie Integerbit verzichtet werden. Um nach der Filterung den Ausgang des Moduls wieder als 16-Bit Integer zu formatieren, wird das Ergebnis der Faltung um 16 Bits nach rechts verschoben. Damit, wie in Abschnitt 3.2.1 bereits erwähnt, keine Fehler durch das Verschieben vorzeichenbehafteter Ganzzahlen entstehen, wurde das Prinzip aus Gleichung 3 hier ebenfalls angewandt. Die Faltung wird im Modul über die Multiplikation der letzten 17 Samples mit den Filtertaps realisiert. Die Summe daraus bildet dann den Ausgang des Filters. Abbildung 6 zeigt das Verhalten des Filters zwischen den Takten 960 und 1200. Aus Speichergründen musste auf die Darstellung des Buffers und der 17 Akkumulationsregister verzichtet werden.

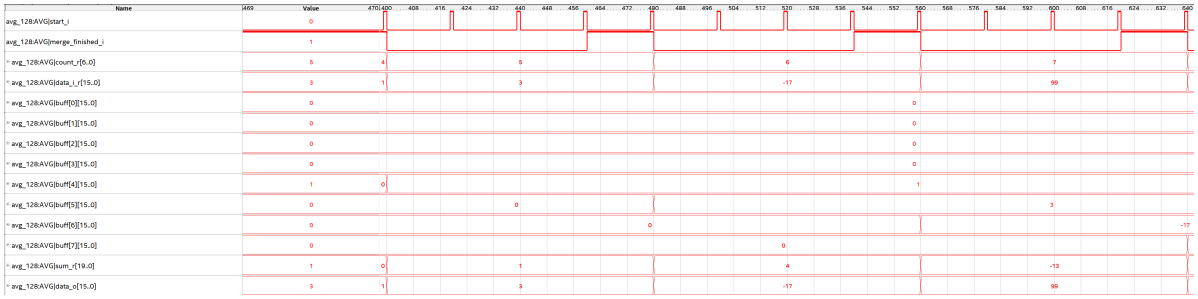

Abbildung 6: Ausschnitt aus dem Signal-Tap-Analyzer für das Modul „fir_17.v“

Es zeigt sich nun auch hier, dass die Verarbeitung der Eingangssamples synchron zu den Steuersignalen erfolgt. Zudem bestätigt die Abbildung 6 die korrekte Formatierung des Ergebnisses zu einer signed 16-Bit Ganzzahl (beispielhaft gilt: $\frac{sum\_r}{2^{16}} ! = 7 =$ *data_o* zwischen Takt 960 und 1040). Die Portliste des Filtermoduls ist der Tabelle 5 zu entnehmen.

Port	Semantik	Format	I/O
clk	Taktsignal	1-Bit	I
rst	Resetsignal	1-Bit	I
start_i	Bedingung für das Updaten der internen Register	1-Bit	I
merge_finished_i	Bedingung für das Updaten der internen Register	1-Bit	I
data_i	Mittelwertfreies Eingangssample	signed 16-Bit	I
data_o	Gefiltertes Ausgangssample	signed 16-Bit	O

Tabelle 5: Portliste des Moduls „fir_17.v“

4 Vergleich & Fazit

Um einen Vergleich für die im FPGA berechneten Ausgangssamples anführen zu können, wurde die gesamte Verarbeitungskette in Python implementiert. Diese ist auf Github im Ordner „Python“ unter dem Namen „process_data.ipynb“ zu finden. Der Quellcode zum Streamen der Daten in den FPGA ist dem Listing 1 zu entnehmen. Um den UART-Buffer des MAX1000 nicht zu überlasten, werden immer 500 Bytes in jedem Schleifendurchlauf gesendet und ausgelesen (siehe Zeile 26 in Listing 1). Beim Vergleich der Ausgangsdaten hat sich gezeigt, dass 1.42% der Samples um den Wert Eins von der Python Implementierung abweichen. In Bezug auf den höchsten Wert der Ausgangsdaten entspricht dies einer prozentualen Abweichung von 0.02%. Durch den Vergleich der Daten nach den einzelnen Modulen zeigte sich, dass die Abweichungen erst nach der Filterung entstehen. Da die Berechnungen im Filtermodul sowohl in der Simulation (ModelSim), als auch im Signal-Tap vollständig nachvollziehbar waren, sind die Abweichungen durch die Wandlung der Filtertaps in ein Festkommaformat erklärbar. Aufgrund des geringen prozentualen Fehlers in Bezug zur Signalamplitude sind diese jedoch nicht hörbar und somit auch vernachlässigbar. Abbildung 7 zeigt nun das gesamte im FPGA demodulierte und aufbereitete FM-Basisbandsignal im Zeit- und Frequenzbereich. Das Signal ist hierfür auf dessen maximale Amplitude normiert worden. Die Berechnung der STFT erfolgte mit einer Segmentlänge von 512 Samples und die Farbcodierung der Amplituden liegt zwischen 0dB und -90dB.

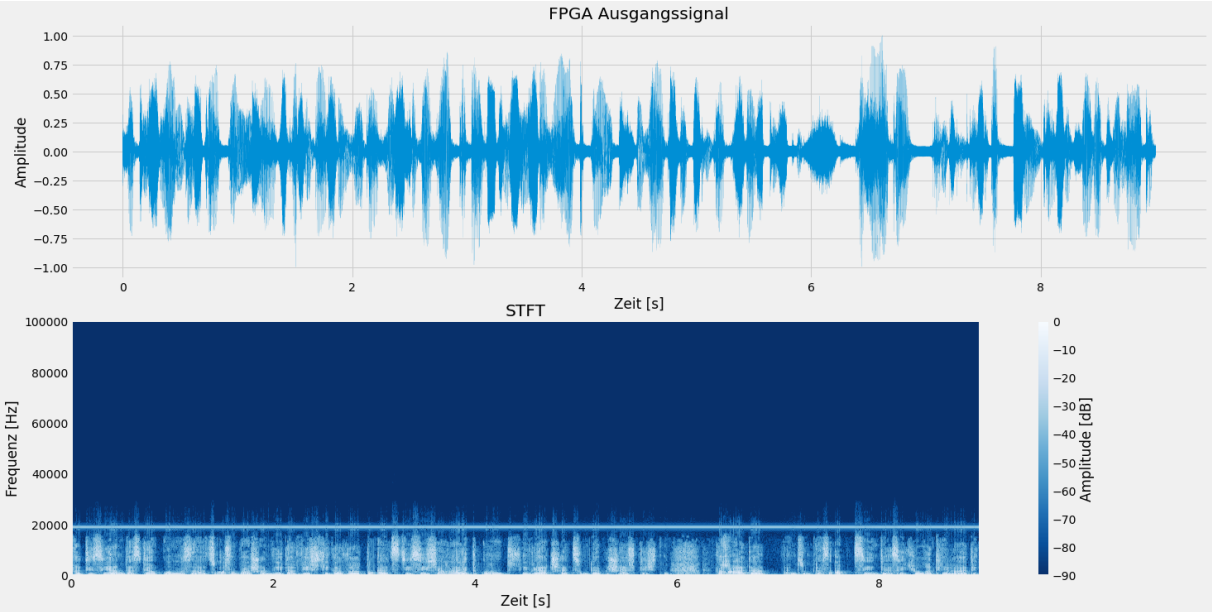

Abbildung 7: Zeit- und Frequenzbereich des FPGA-Ausgangssignals

Im Vergleich zu dem demodulierten Signal ohne Offset-Ausgleich und Filterung (siehe Abbildung 8 im Anhang) wird die korrekte Funktionsweise der Module „avg_128.v“ und „fir_17.v“ noch einmal deutlich.

Abschließend lässt sich sagen, dass der FM-Demodulator erfolgreich implementiert werden konnte. In Bezug auf die verfügbaren Ressourcen wurden 85 % der Logikelemente verbraucht. Zudem wurden vier der 48 Hardwaremultiplizierer benötigt. Die durch die Pipeline enstehende Verarbeitungsverzögerung beträgt 24 Bytes bzw. sechs Ausgangssamples. Das Quartus-Prime-Projekt, sowie alle zur Entwicklung benötigten Textdateien und Python-Programme sind im GitHub-Repository verfügbar. Die finalen Wave-Dateien sind im Ordner „Python“ unter den Namen „Signal_Python“ und „Signal_FPGA“ hinterlegt.

5 Anhang

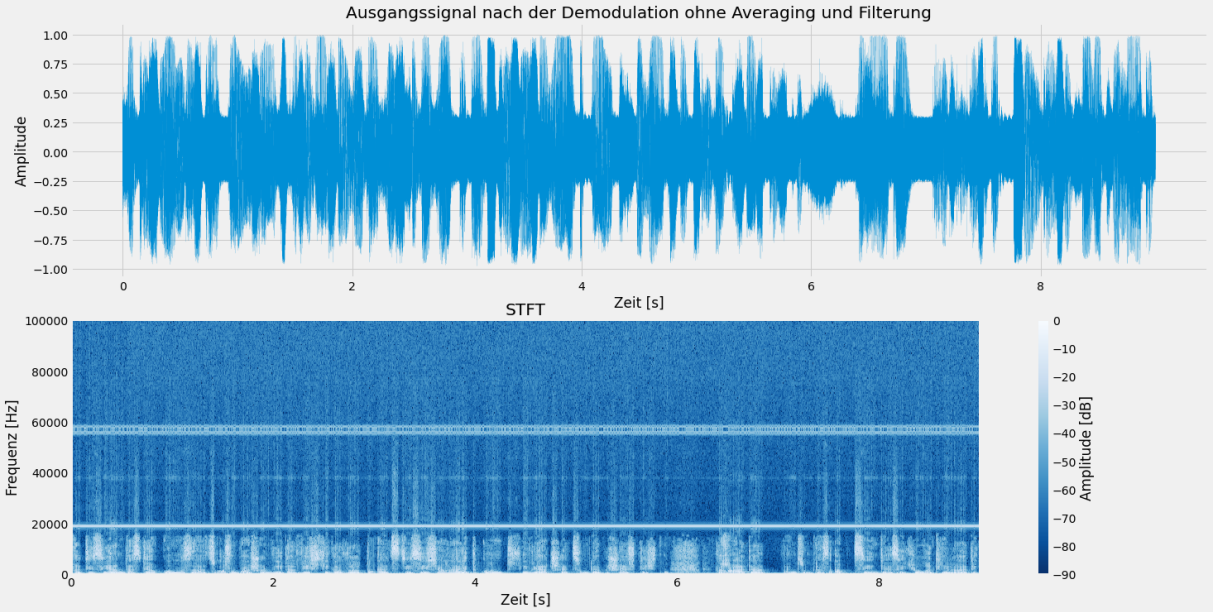

Abbildung 8: Zeit- und Frequenzbereich des FPGA-Ausgangssignals nach der Demodulation ohne Averaging und Filterung

```python
import serial
import numpy as np
import serial.tools.list_ports
print('try to open COM Port')

#COM-Schnittstelle detektieren
l = list(serial.tools.list_ports.comports())
print(*l, sep="\n")

#Lesen der formatierten Eingangsbytes
data = np.loadtxt('fm_bytes.txt', dtype='int')

#Länge auf 7.2 Millionen Samples beschränken und als Liste formatieren
data_in = data[0:7200000].tolist()

#Konfiguration der Schnittstelle
ser = serial.Serial('COM4', 115200, serial.EIGHTBITS, serial.PARITY_NONE,
    serial.STOPBITS_ONE, 0.1)

#Anzahl der Schleifendurchläufe
N = int(len(data_in)/500)

output = np.zeros((N, int(len(data_in)/N)))

print('try to read from COM Port')

#Sende- und Leseschleife von 500 Bytes pro Durchlauf
for i in range (N):
    print(i,'von',(N))
    ser.write(data_in[int(i*len(data_in)/N): int((i+1)*len(data_in)/N)])
    r = ser.read(len(data_in))

    mv = memoryview(r).cast('B')
    data_o = (np.array(mv))
    output[i, :] = data_o

print('data transfer successfull \n\rformatting output array ...')
x = output.reshape(1, len(data_in))

#Ersten vier Elemente Löschen (Delay durch Merge & Split)
x_mod = np.delete(x, [0,1,2,3])

#Formatierung als 8-Bit uint
x_mod = np.array(x_mod, dtype=np.uint8)

#Zusammenfassen als 32-Bit int
fm_demod = np.zeros((len(x_mod)//4,), dtype=np.int32)

for i in range(0, len(x_mod), 4):
    val = x_mod[i] << np.int32(24) | x_mod[i+1] << np.int32(16) | x_mod[i+2] <<
        np.int32(8) | x_mod[i+3]
    fm_demod[i//4] = np.int16(val)

#Speichern der formatierten Ausgangsdaten
np.savetxt('output_data.txt', X=fm_demod[5:], fmt='%d', delimiter= ' \n')

print('done')

```

Listing 1: Python-Code zum Senden aus Auslesen der Daten über UART