On Chip Signal Processing - RTL Implementierung RX

Daniel Ferrari
Nikolaus Haminger
Florian Kitzer
Matthäus Kücher
Alexander Traxler

24. Februar 2019

Inhaltsverzeichnis

1 Übersicht **2**
1.1 Spezifikation . . . 2
1.2 Implementierung . . . 2
1.3 Toplevel-Simulation . . . 2
1.3.1 Anleitung zur Toplevel-Simulation . . . 3
1.3.2 Simulations-Output . . . 3
1.3.3 Konstellationsdiagramm . . . 5
1.3.4 Synthese . . . 5

2 Komponenten **6**
2.1 Interpolation . . . 6
2.1.1 Architektur . . . 6
2.1.2 Simulation . . . 7
2.2 Coarse Alignment . . . 8
2.2.1 Architektur . . . 8
2.2.2 Simulation . . . 10
2.2.3 Synthese . . . 11
2.3 Cyclic Prefix Removal . . . 12
2.4 FFT Wrapper . . . 12
2.5 Fine Alignment . . . 13
2.5.1 Architektur . . . 13
2.5.2 Simulation . . . 14
2.5.3 Synthese . . . 14
2.6 Demodulation . . . 15

1 Übersicht

1.1 Spezifikation

In diesem Projekt sollte eine OFDM-RX-Architektur in VHDL implementiert werden. Ein grobes Blockdiagramm dieser RX-Komponente war vorgegeben und ist in Abbildung 1 sichtbar.

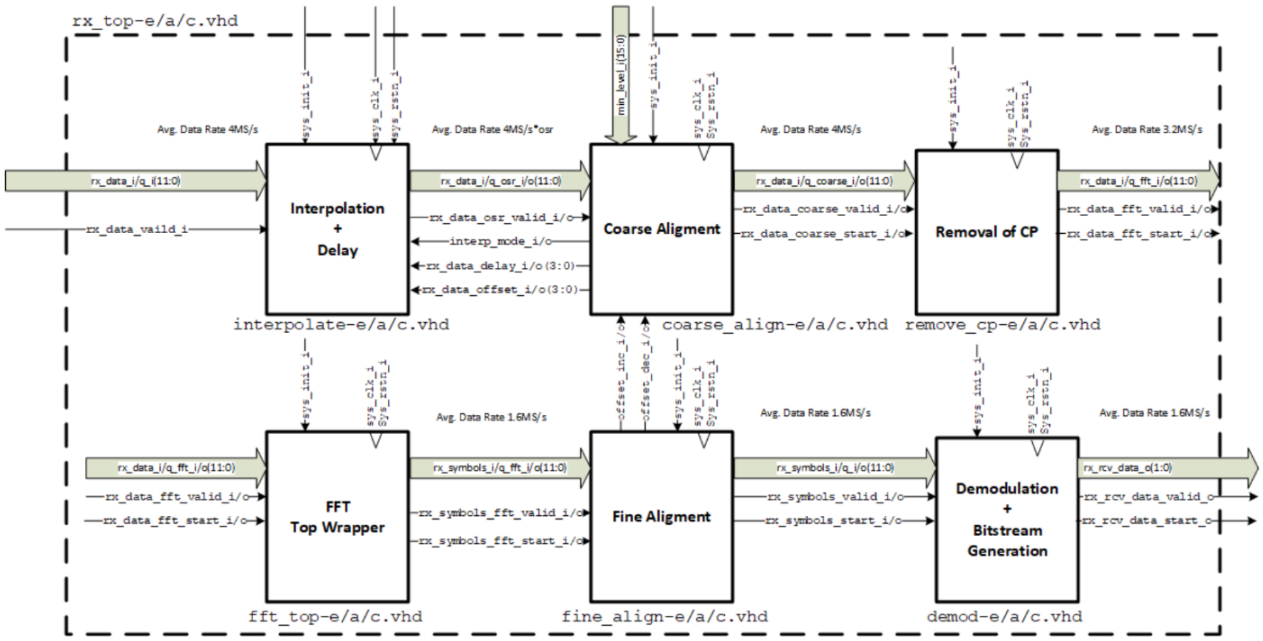

Abbildung 1: *Blockschaltbild der gesamten RX-Kette.*

1.2 Implementierung

Die Implementierung wurde weitgehend nach dem Vorbild des spezifizierten Blockschaltbildes vollzogen. Allerdings wird der Offset und das Delay in der Entity *Coarse Alignment* gehandhabt, weshalb die Leitungen `rx_data_delay`, `rx_data_offset` und `interp_mode` zwischen *Interpolation* und *Coarse Alignment* eliminiert wurden.

1.3 Toplevel-Simulation

Zur Verifikation der gesamten RX-Kette wurde eine Toplevel-Simulation erstellt. Für den Vergleich der Ergebnisse der Hardware-Kette wurde ein Golden Model in Python implementiert. Diese Implementierung berechnet die Input-Signale für die Hardware-Simulation (`rx_data_i`), speichert diese in CSV-Dateien ab und simuliert die gesamte RX-Kette in einer Hardware-ähnlichen Implementierung. Die Ergebnisse des Golden Models werden ebenfalls abgespeichert und am Ende der Hardware-Simulation mit eben jenen Ergebnissen verglichen. Außerdem werden *BER* und *EVM* berechnet sowie Konstellationdiagramme jeweils für beide Simulationen erstellt.

Achtung: Matlab wird nicht benötigt, da das gesamte Golden Model sowie die Verifikation in Python implementiert wurde (mit Fixed-Point-Repräsentation und Taylor-Approximation der

Interpolation). Die Gründe für diese Maßnahme sind die bessere Performance von Python, die einfachere Interaktion aus der Modelsim-Simulation und die bessere Strukturierungsmöglichkeit des Codes.

1.3.1 Anleitung zur Toplevel-Simulation

Eine ausführliche Anleitung befinden sich in der `README.md` des Repositories. Hier kurz noch einmal zusammengefasst:

Anforderungen:

- Modelsim oder Questasim
- Python3 (https://www.python.org/downloads/)
- numpy und matplotlib (`pip install numpy matplotlib`)

Ausführung:
Um die Simulation zu starten, einfach `start_rx_simulation.bat <cmd|gui|clean>` oder `start_rx_simulation.sh <cmd|gui|clean>` im Root-Folder des Repositories ausführen.

1.3.2 Simulations-Output

Nachfolgend ist der Output einer Simulation von 3 unterschiedlichen RX-Sequenzen zu sehen.

```
# ========================================
# Test the OFDM RX path
# ========================================
# [      0.00 us] INFO    : Working in folder src/grpRx/unitTopLevel/sim
#
# Testing RX chain with symbol sequence #0
# ----------------------------------------
# [      0.12 us] INFO    : Generating input data and expected bitstream by calling ../src/
    golden_model.py
# [      0.12 us] INFO    : Loading input data and expected bitstream from files rx_in_signal0
    .csv and result_bits0.csv
# [      0.12 us] INFO    : Feeding system with RX symbols, this might take a while
# [    243.38 us] INFO    : Received  256 of 5120 output bits ( 1 of 20 chips)
# [    323.37 us] INFO    : Received  512 of 5120 output bits ( 2 of 20 chips)
# [    403.26 us] INFO    : Received  768 of 5120 output bits ( 3 of 20 chips)
# [    483.26 us] INFO    : Received 1024 of 5120 output bits ( 4 of 20 chips)
# [    563.26 us] INFO    : Received 1280 of 5120 output bits ( 5 of 20 chips)
# [    643.37 us] INFO    : Received 1536 of 5120 output bits ( 6 of 20 chips)
# [    723.38 us] INFO    : Received 1792 of 5120 output bits ( 7 of 20 chips)
# [    803.37 us] INFO    : Received 2048 of 5120 output bits ( 8 of 20 chips)
# [    883.26 us] INFO    : Received 2304 of 5120 output bits ( 9 of 20 chips)
# [    963.26 us] INFO    : Received 2560 of 5120 output bits (10 of 20 chips)
# [   1043.27 us] INFO    : Received 2816 of 5120 output bits (11 of 20 chips)
# [   1123.37 us] INFO    : Received 3072 of 5120 output bits (12 of 20 chips)
# [   1203.38 us] INFO    : Received 3328 of 5120 output bits (13 of 20 chips)
# [   1283.37 us] INFO    : Received 3584 of 5120 output bits (14 of 20 chips)
# [   1363.27 us] INFO    : Received 3840 of 5120 output bits (15 of 20 chips)
# [   1443.26 us] INFO    : Received 4096 of 5120 output bits (16 of 20 chips)
# [   1523.27 us] INFO    : Received 4352 of 5120 output bits (17 of 20 chips)
# [   1603.37 us] INFO    : Received 4608 of 5120 output bits (18 of 20 chips)
# [   1680.11 us] INFO    : Received enough output bits
# [   1680.11 us] INFO    : Verifying RX chain output
# [   1680.11 us] SUCCESS : Validation successful: BER=0.000, EVM=20.895db
#
# [   1680.11 us] INFO    : Scatter plot file written: scatter_plot0.png
#
# Testing RX chain with symbol sequence #1
# ----------------------------------------
# [   1680.13 us] INFO    : Generating input data and expected bitstream by calling ../src/
    golden_model.py
```

```
# [   1680.13 us] INFO    : Loading input data and expected bitstream from files rx_in_signal1
    .csv and result_bits1.csv
# [   1680.13 us] INFO    : Feeding system with RX symbols, this might take a while
# [   1923.38 us] INFO    : Received  256 of 5120 output bits ( 1 of 20 chips)
# [   2003.37 us] INFO    : Received  512 of 5120 output bits ( 2 of 20 chips)
# [   2083.26 us] INFO    : Received  768 of 5120 output bits ( 3 of 20 chips)
# [   2163.26 us] INFO    : Received 1024 of 5120 output bits ( 4 of 20 chips)
# [   2243.26 us] INFO    : Received 1280 of 5120 output bits ( 5 of 20 chips)
# [   2323.36 us] INFO    : Received 1536 of 5120 output bits ( 6 of 20 chips)
# [   2403.38 us] INFO    : Received 1792 of 5120 output bits ( 7 of 20 chips)
# [   2483.36 us] INFO    : Received 2048 of 5120 output bits ( 8 of 20 chips)
# [   2563.26 us] INFO    : Received 2304 of 5120 output bits ( 9 of 20 chips)
# [   2643.26 us] INFO    : Received 2560 of 5120 output bits (10 of 20 chips)
# [   2723.26 us] INFO    : Received 2816 of 5120 output bits (11 of 20 chips)
# [   2803.36 us] INFO    : Received 3072 of 5120 output bits (12 of 20 chips)
# [   2883.38 us] INFO    : Received 3328 of 5120 output bits (13 of 20 chips)
# [   2963.36 us] INFO    : Received 3584 of 5120 output bits (14 of 20 chips)
# [   3043.26 us] INFO    : Received 3840 of 5120 output bits (15 of 20 chips)
# [   3123.26 us] INFO    : Received 4096 of 5120 output bits (16 of 20 chips)
# [   3203.26 us] INFO    : Received 4352 of 5120 output bits (17 of 20 chips)
# [   3283.36 us] INFO    : Received 4608 of 5120 output bits (18 of 20 chips)
# [   3360.11 us] INFO    : Received enough output bits
# [   3360.11 us] INFO    : Verifying RX chain output
# [   3360.11 us] SUCCESS : Validation successful: BER=0.000, EVM=21.016db
#
# [   3360.11 us] INFO    : Scatter plot file written: scatter_plot1.png
#
# Testing RX chain with symbol sequence #2
# ----------------------------------------
# [   3360.13 us] INFO    : Generating input data and expected bitstream by calling ../src/
    golden_model.py
# [   3360.13 us] INFO    : Loading input data and expected bitstream from files rx_in_signal2
    .csv and result_bits2.csv
# [   3360.13 us] INFO    : Feeding system with RX symbols, this might take a while
# [   3603.24 us] INFO    : Received  256 of 5120 output bits ( 1 of 20 chips)
# [   3683.24 us] INFO    : Received  512 of 5120 output bits ( 2 of 20 chips)
# [   3763.26 us] INFO    : Received  768 of 5120 output bits ( 3 of 20 chips)
# [   3843.26 us] INFO    : Received 1024 of 5120 output bits ( 4 of 20 chips)
# [   3923.36 us] INFO    : Received 1280 of 5120 output bits ( 5 of 20 chips)
# [   4003.38 us] INFO    : Received 1536 of 5120 output bits ( 6 of 20 chips)
# [   4083.36 us] INFO    : Received 1792 of 5120 output bits ( 7 of 20 chips)
# [   4163.27 us] INFO    : Received 2048 of 5120 output bits ( 8 of 20 chips)
# [   4243.26 us] INFO    : Received 2304 of 5120 output bits ( 9 of 20 chips)
# [   4323.27 us] INFO    : Received 2560 of 5120 output bits (10 of 20 chips)
# [   4403.36 us] INFO    : Received 2816 of 5120 output bits (11 of 20 chips)
# [   4483.38 us] INFO    : Received 3072 of 5120 output bits (12 of 20 chips)
# [   4563.36 us] INFO    : Received 3328 of 5120 output bits (13 of 20 chips)
# [   4643.27 us] INFO    : Received 3584 of 5120 output bits (14 of 20 chips)
# [   4723.26 us] INFO    : Received 3840 of 5120 output bits (15 of 20 chips)
# [   4803.27 us] INFO    : Received 4096 of 5120 output bits (16 of 20 chips)
# [   4883.36 us] INFO    : Received 4352 of 5120 output bits (17 of 20 chips)
# [   4963.38 us] INFO    : Received 4608 of 5120 output bits (18 of 20 chips)
# [   5040.10 us] INFO    : Received enough output bits
# [   5040.10 us] INFO    : Verifying RX chain output
# [   5040.10 us] SUCCESS : Validation successful: BER=0.000, EVM=22.675db
#
# [   5040.10 us] INFO    : Scatter plot file written: scatter_plot2.png
#
# ========================================
# Simulation end @5040.10 us
# ========================================
# Number of checks :           3
# Successful       :           3
# Warnings         :           0
# Errors           :           0
# ========================================
# SIMULATION SUCCESS
# ========================================
```

1.3.3 Konstellationsdiagramm

Abbildung 2 zeigt die Konstellationsdiagramme einer Symbol-Sequenz für die Golden-Model- und die Hardware-Simulation. Die Plots sind nach einer Simulation im Ordner `src/grpRx/unitTopLevel/sim` zu finden.

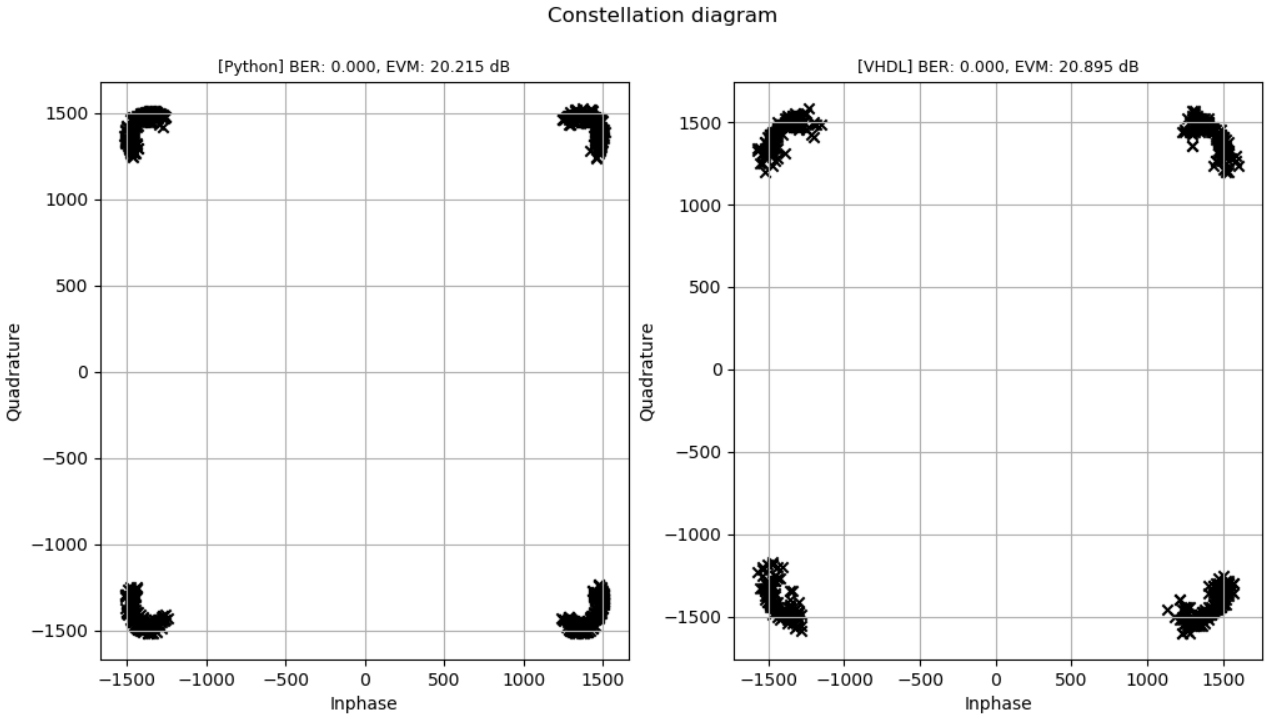

Abbildung 2: *Konstellationsdiagramme der Golden-Model- und Hardware-Simulationen*

1.3.4 Synthese

Für das implementierte Gesamtsystem wurde eine Synthese mit `Quartus` durchgeführt. Als Zielplattform wurde der FPGA 5CSXFC6D6F31C6 des DE-10 Standard-Boards verwendet. Folgende Ressourcen werden für das Gesamtsystem benötigt bei einer Symbollänge von 320 Samples und einer Überabtastung von 16. Diese sind in der nachstehenden Tabelle 2 zu sehen.

Ressource	**Menge**
Register	15,737
ALMs	7,912 / 41,910
M10K-Blöcke	42 / 553
Block Memory Bits	134,896 / 5,662,720
Total Block Memory Implementation Bits	430,080 / 5,662,720
DSP-Blöcke	17 / 112
Pins (theoretisch)	48 / 499

Tabelle 1: Tabelle mit benötigten Ressourcen.

Die Timing-Analyse hat ergeben, dass das Gesamtsystem im 1100mV-85C-Model mit einer

maximalen Taktfrequenz von 79,8 MHz betrieben werden kann. Als Systemtakt sind 100 MHz vorgesehen.

2 Komponenten

2.1 Interpolation

Die Aufgabe des Blockes `Interpolation` ist das Upsampling der einzelnen Symbole um 2^n Symbole. Als Interpolationsalgorithmus wird die Taylor-Reihe verwendet mit

$$f(x) = f(x_0) + f^{'}(x_0)(x - x_0) + \frac{f^{''}(x_0)}{2!}(x - x_0)^2,$$

wobei die erste und zweite Ableitung aus den ursprünglichen Samples berechnet wird

2.1.1 Architektur

Für die Berechnung der Interpolationssymbole durch die Taylor-Reihe werden jeweils 3 Symbole benötigt. Zu Begin wird deshalb so lange gewartet, bis 3 Symbole gespeichert werden. Danach kann nun die erste und zweite Abbleitung durch

$f^{'}(x_0) = f(x_1) - f(x_0)$
$f^{''}(x_0) = f(x_2) - 2f(x_1) + f(x_0)$

berechnet werden und somit mit jedem Takt ein neues interpoliertes Symbol ausgegeben werden. Wie viele Symbole interpoliert werden sollten, wird als generic Parameter mitgegeben.

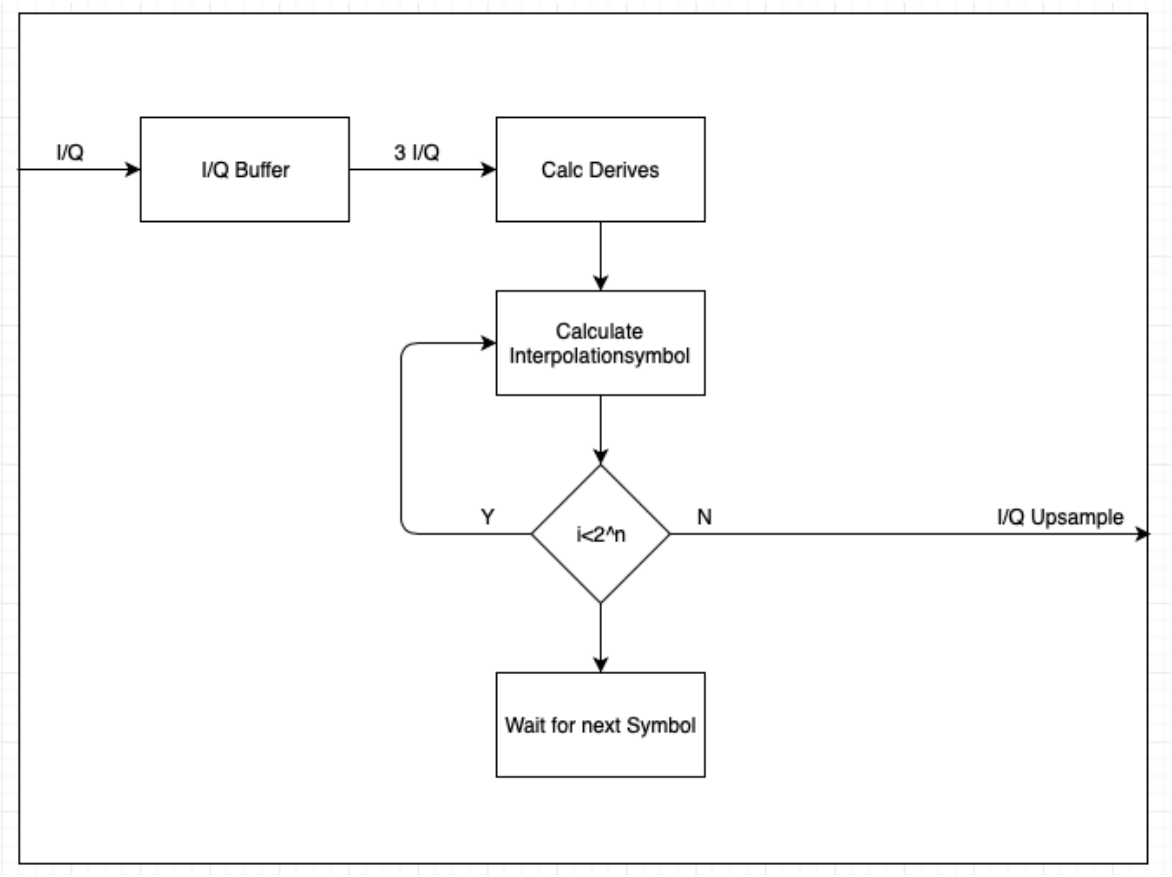

Abbildung 3: *Architektur von* `Interpolation`

.

2.1.2 Simulation

Für die korrekte Funktionalität des Interpolationsblockes wurde Matlab und Modelsim für die Simulation verwendet. Dazu wurden zufällige Symbole in Matlab generiert. Diese Symbole wurden einmal mit der Matlab Funktion und einmal mit vhdl interpoliert, geplotet und verglichen.

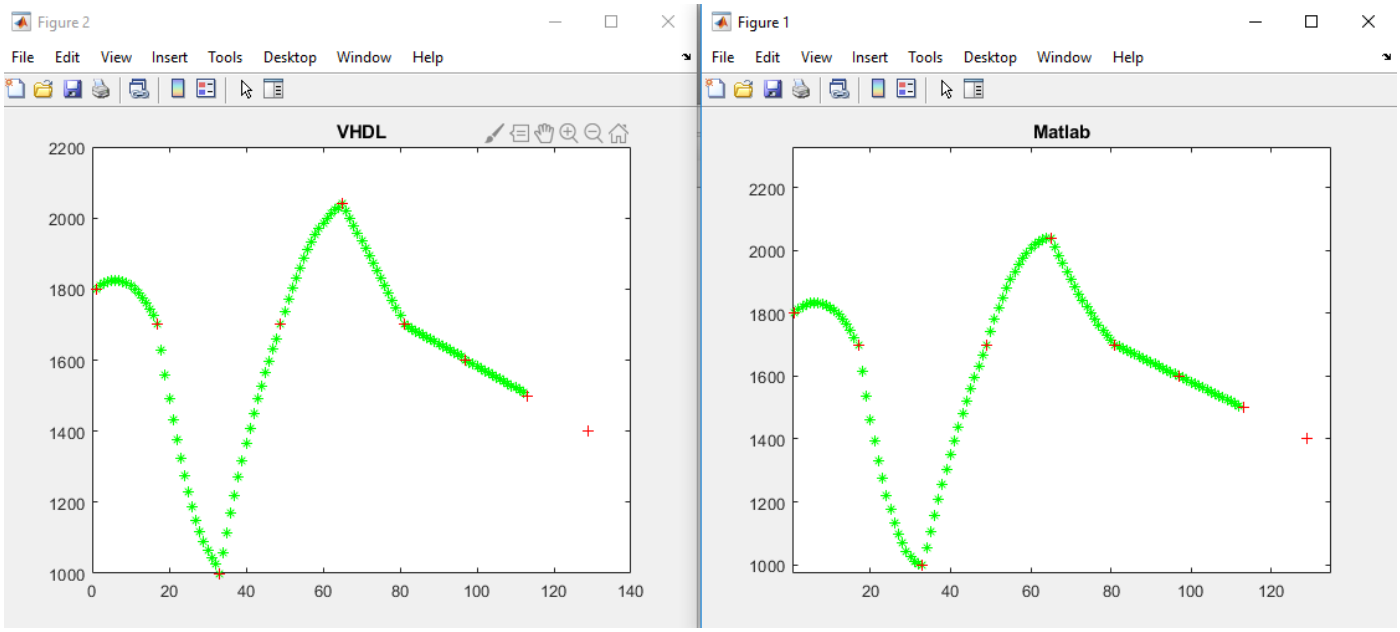

Abbildung 4: *Testausgabe von* ***`Interpolation`***

.

2.2 Coarse Alignment

Die Aufgabe des `Coarse Alignment` ist das Finden des Beginns einer OFDM-Übertragung. Für das Finden der Beginns wird die Methode nach Schmidl und Cox verwendet. Dazu wird in Hardware die Korrelation über zwei identische Halbsymbole (= Trainingssymbol) berechnet und das Maximum detektiert. Danach werden fortlaufend die Steuersignale `start_of_symbol` und `data_valid` des Datenstromes gesetzt und nach den Vorgaben des `Fine Alignment` korrigiert.

2.2.1 Architektur

In Abbildung 5 ist ein grober Überblick über die Implementierung gegeben. Die Implementierung beginnt links oben mit dem Eintreffen der I- und Q-Komponente des Empfangssignals. Beide Komponenten haben Datenbreite von 12 Bit signed. Die eintreffenden Daten werden in einem Ringbuffer mit der halben Symbollänge gespeichert, wodurch die Daten um die halbe Symboldauer verzögert werden. Die darauffolgenden Multiplizierer und Addierer stellen die komplexe Multiplikation der Korrelationsberechnung dar.

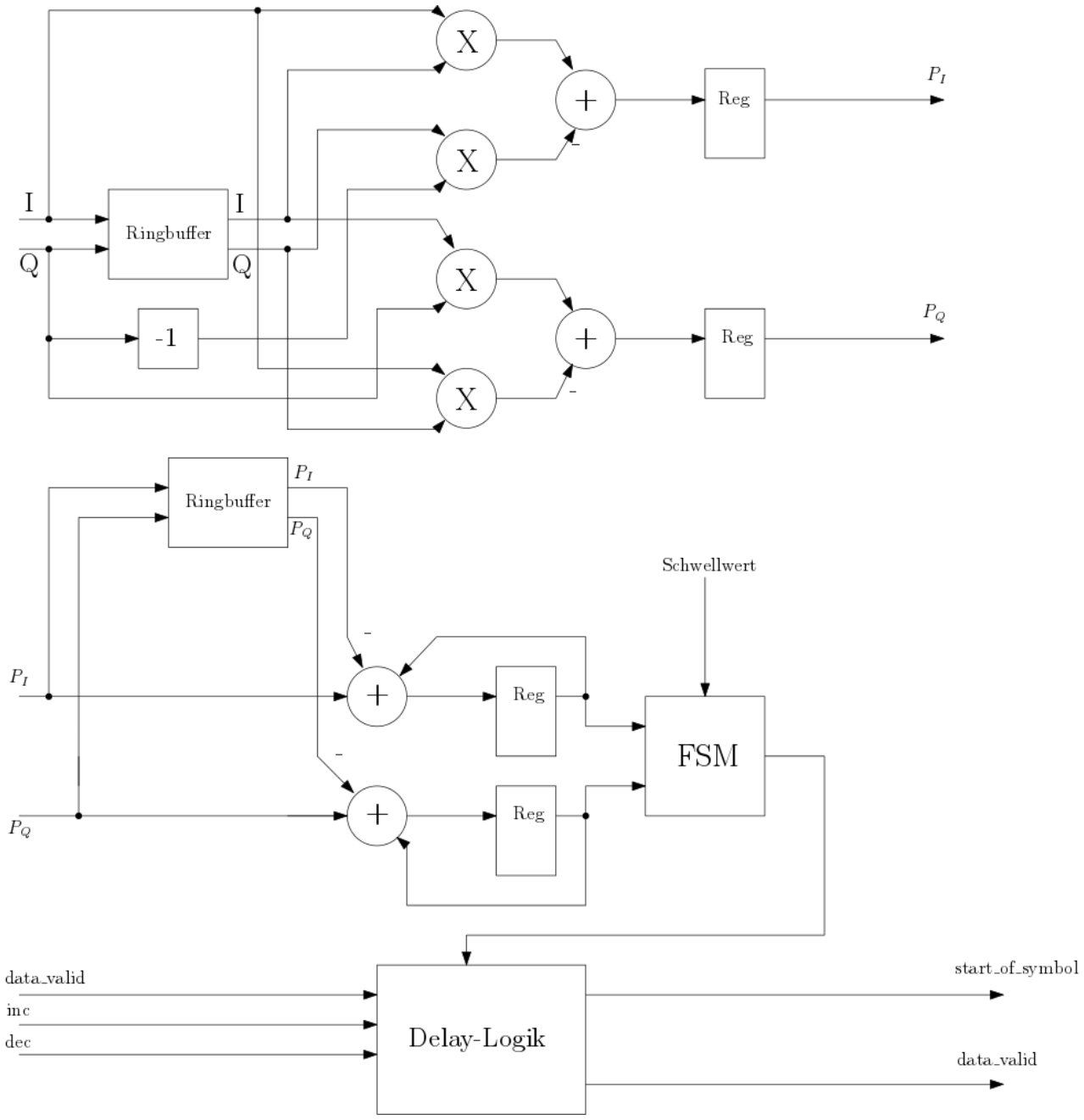

Abbildung 5: *Architektur des* `Coarse Alignment`*. Links oben sind die Eingangsdaten als I und Q Anteil zu sehen. Darauf folgt die Korrelationsberechnung, gefolgt von der FSM zum Erkennen des Maximums. Unten im Bild ist die Logik zur Steuersignalgenerierung zu sehen.*

Die Ergebnisse der komplexen Multiplikation werden in Register gespeichert, bevor diese im Zielregister aufsummiert werden. Nach der komplexen Multiplikation haben sowohl I- als auch Q-Komponente eine Datenbreite von 25 Bit signed. Auch die Ergebnisse der komplexen Multiplikation werden in einem Ringbuffer abgelegt, damit diese nach der halben Symboldauer wieder subtrahiert werden können. Dadurch entsteht der Effekt eines gleitenden Korrelationsfensters über dem Empfangssignal. Für das Ergebnis der Korrelation wird eine Datenbreite von 35 Bit signed verwendet.

Das berechnete Korrelationssignal wird einer FSM zugeführt. Diese FSM beginnt ein Maximum auf dem Korrelationssignal zu suchen, wenn dieses einen gegebenen Schwellwert überschreitet. Ein Maximum wurde dann gefunden, wenn der aktuelle Signalwert im Vergleich zum vorhergehenden Signalwert beginnt zu sinken. Ab diesen Zeitpunkt werden die Daten vom `Coarse Alignment` vom Eingang zum Ausgang weitergegeben. Für die Auswertung wird nur die I-Komponente des Korrelationssignals verwendet, da diese das Maximum ausbildet. Die FSM steuert danach die Delay-Logik, die für die Generierung der Steuersignale `start_of_symbol` und `data_valid` zuständig ist. Diese besteht intern aus zwei Zählern. Der erste Zähler ist für die Generierung des `data_valid`-Signals zuständig. Wenn dieser Zähler einen Überlauf hat, wird das Signal für eine Taktperiode auf 1 gesetzt und der zweite Zähler ebenfalls erhöht. Hat der zweite Zähler einen Überlauf wird das `start_of_symbol`-Signal für eine Taktperiode auf 1 gesetzt. Die Korrektur des Delays wird über die `inc`- und `dec`-Leitungen durch das `Fine Alignment` gesteuert. Ist eine der beiden Leitungen 1 wird der erste Zähler der Delay-Logik entsprechend korrigiert, wodurch die Steuersignale um ein Datum nach vorne oder nach hinten verschoben werden. Die Auswertung der Leitungen wird immer zu Beginn eines neuen OFDM-Symbols durchgeführt.

2.2.2 Simulation

Für die Simulation wurde sowohl Matlab als auch Modelsim verwendet. Mit Hilfe von Matlab wurde ein Empfangssignal mit der Länge von 100 Symbolen generiert. In diesem Empfangssignal ist das Trainingssymbol an der dritten Stelle. Der Beginn der Korrelation des generierten Empfangssignals ist in Abbildung 6 zu sehen. Der grün umrandete Signalteil stellt das Trainingssymbol dar. Das obere Diagramm zeigt das Ergebnis der iterativen Implementierung. Das untere Diagramm zeigt das Ergebnis der formalen Implementierung. Es ist zu erkennen, dass beide Diagramme bis auf eine Verschiebung den gleichen Kurvenverlauf darstellen.

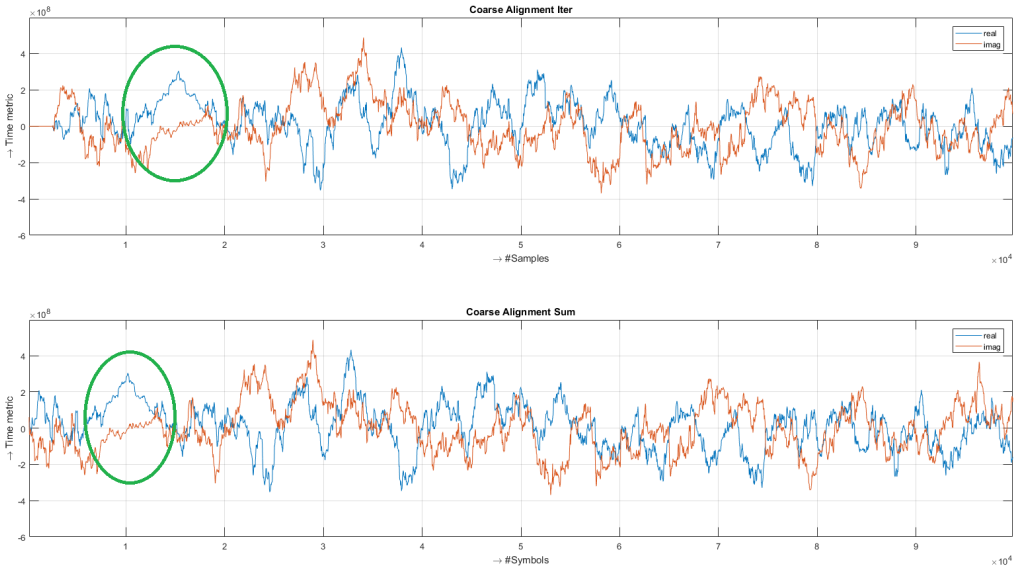

Abbildung 6: *Beginn des Empfangssignals. Oben ist die iterative und unten die formale Implementierung dargestellt. Der grün umrandete Signalteil stellt das Trainingssymbol dar.*

Für die Simulation wurde der Simulator `Modelsim` verwendet. Als Testframework wurde UVVM verwendet. Dadurch war es möglich ein besser strukturierte VHDL-Testbench zu entwerfen. In der Simulation wird das mit Matlab generierte Empfangssignal an den `Coarse Alignment`-Block angelegt und auf die Detektion des Trainingssymbols gewartet. Nach der Detektion wird das Timing der Steuersignale überprüft. Darauf wird überprüft, ob die Daten korrekt vom Block weiter gegeben werden. Am Ende wird das Verhalten im Zusammenhang mit den Steuerleitungen vom `Fine Alignment` und dem `init`-Signal überprüft. In Abbildung 7 ist ein Teil der Waveform der Simulation zu sehen. Die berechnete Korrelation in der Simulation stimmt mit der Berechnung in Matlab überein.

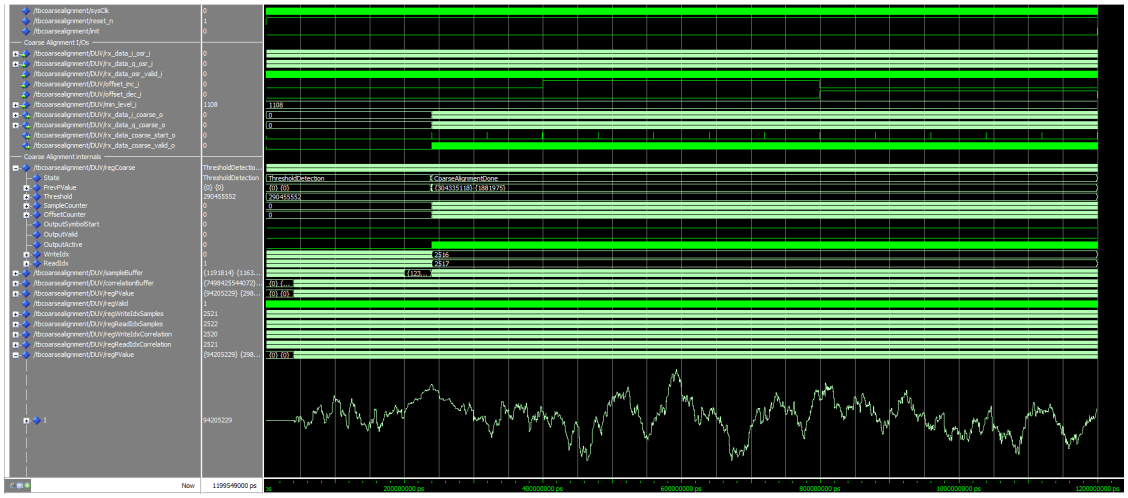

Abbildung 7: *Waveform der Simulation des* ***`Coarse Alignment`****. Im unteren Teil der Bildes ist die Korrelation des Empfangssignales zu erkennen.*

Um die Simulation starten zu können, müssen die Skripen `compile_uvvm.do`, `compile.do` und `run_simulation.do` in dieser Reihenfolge ausgeführt werden.

2.2.3 Synthese

Für das implementierte `Coarse Alignment` wurde eine Synthese mit `Quartus` durchgeführt. Als Zielplattform wurde der FPGA 5CSXFC6D6F31C6 des DE-10 Standard-Boards verwendet. Folgende Ressourcen werden für das `Coarse Alignment` benötigt bei einer Symbollänge von 320 Samples und einer Überabtastung von 16. Diese sind in der nachstehenden Tabelle 2 zu sehen.

Ressource	Menge
Register	232
ALMs	182 / 41,910
M10K-Blöcke	12 / 553
Block Memory Bits	122,880 / 5,662,720
Total Block Memory Implementation Bits	245,760 / 5,662,720
DSP-Blöcke	2 / 112
Pins (theoretisch)	72 / 499

Tabelle 2: Tabelle mit benötigten Ressourcen.

Die Timing-Analyse hat ergeben, dass das `Coarse Alignment` im 1100mV-85C-Model mit einer maximalen Taktfrequenz von 111,68 MHz betrieben werden kann. Als Systemtakt sind 100 MHz vorgesehen.

2.3 Cyclic Prefix Removal

2.4 FFT Wrapper

Für den FFT-Block wurde ein Wrapper für die Altera-FFT implementierung gebaut. Dazu werden 256 I/Q Symbole in einem Buffer gespeichert. Dieser Buffer wird der Altera FFT implemssentierung übergeben, die Steuersignale Start und Valid gesetzt und die Berechnung der Altera FFT beginnt. Wenn die Altera FFT-Implementierung die Berechnung fertig hat, werden die Steuerung des Ausganges Valid und StartOfPacket gesetzt. Somit kann das FFT-Ergebnis wieder in den internen Buffer gespeichert werden. Die unbenutzen Frequenzen werden herausgefiltert und die Daten an das FineAlignment weitergegeben.

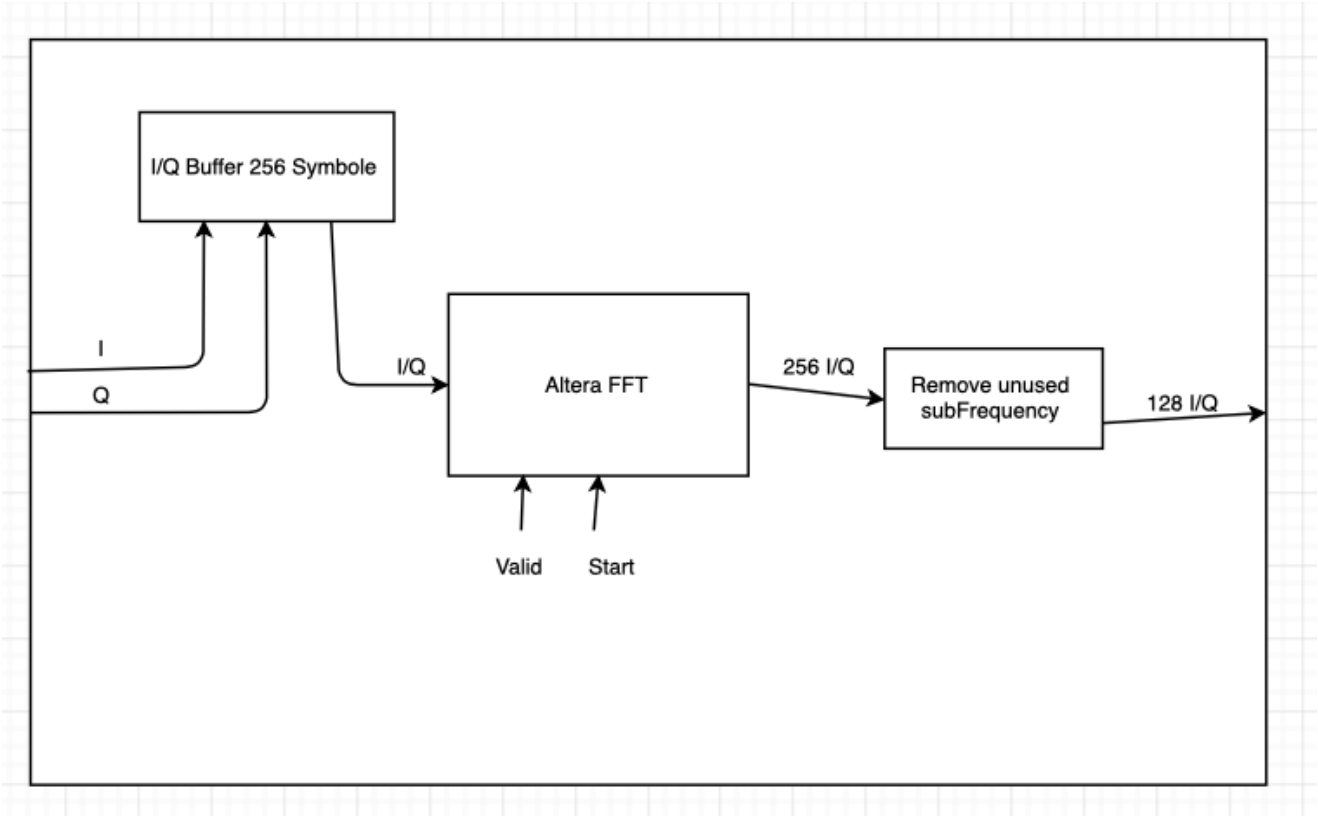

Abbildung 8: *Architektur von* ***FFT Wrapper***

.

2.5 Fine Alignment

Das `Fine Alignment` kümmert sich um die Feinjustierung der Symbole über die Phase. Eine Phasenverschiebung im Frequenzbereich ist gleichbedeutend mit einer zeitlichen Verschiebung im Zeitbereich. Nach der groben Korrektur (Coarse Alignment) besteht immer noch ein Phasenfehler, der über mehrere Symbole hinweg korrigiert werden kann.

2.5.1 Architektur

Das Blockdiagramm in Abbildung 9 verdeutlicht den Aufbau des Fine Alignment. Zur Berechnung des Phasenfehlers werden, aufgrund der Linearphasigkeit des Filters in diesem Bereich, nur die ersten 32 Symbole verwendet. Die Symbole werden zu Beginn in den 1. Quadranten (First Quadrant) gebracht. Die Inphase- und Quadratur-Komponenten werden separat aufsummiert. Der Phasenfehler ergibt sich durch die Subtraktion der Inphase- von der Quadratur-Komponente (Q-I). Anders ausgedrückt Imaginärteil - Realteil. Dieser Ansatz ist möglich, da wir QPSK verwenden, wo nur ein Symbol pro Quadrant vorhanden ist. Ergibt die Subtraktion 0 so ist kein Phasenfehler gegeben. Andernfalls ist nur das Vorzeichen interessant, welches die Richtung für die Verschiebung im Zeitbereich angibt.

Hat ein komplettes Symbol das Fine Alignment durchlaufen, erwartet sich das Coarse Alignment eine Entscheidung ob *increment* oder *decrement*. Keines von beiden ist in der Implementierung nicht möglich. Die inc und dec Leitungen sind direkt verknüpft mit dem Vorzeichen des Ergebnisse der Phasenberechnung. Das Ergebnis schwankt somit bei der Berechnung (innerhalb der 32 Symbole) stark und ist danach fix auf einem Wert für den Rest des Symbols (bis zum Start des nächsten). Dadurch schwankt das Fine Alignment immer zwischen dem besten und zweitbesten Wert für Korrektur der Phase. Wichtig für die Implementierung sind die Bitbreiten für die Summen von Imaginär- und Realteil (Reg Imaginär, Reg Real). Diese berechnen sich aus 32 * 12 Bit = 17 Bit.

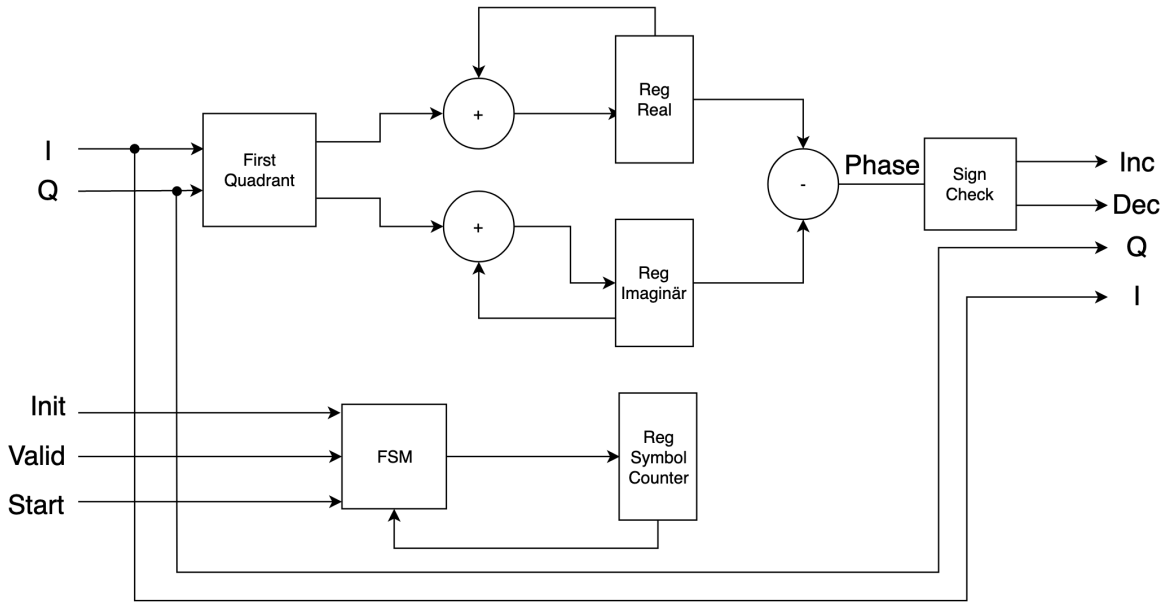

Abbildung 9: Blockdiagramm des Fine Alignment

2.5.2 Simulation

Getestet wurde mit UVVM. Die Testbench vom Coarse Alignment wurde wiederverwendet und adaptiert für das Fine Alignment. Geprüft werden Reset-Condition, Ergebnis von inc und dec nach den ersten 32 Werten des Symbols und die Init-Condition nach dem Symbol. Aufgrund der Implementierung ist im Reset und im Init die dec Leitung aktiv und die inc Leitung inaktiv. Die Leitungen werden aber erst beim Übergang von Init auf Phase in diesen Zustand versetzt. Ansonsten würde das eigentliche Ergebnis nach den 32 Symbolen verloren gehen (vgl. Abbildung 10).

Um Werte für die Simulation zu generieren wurde ein Matlab-Modell des Fine Alignment erstellt. In jedem Schritt wo korrigiert wird können die Daten aus dem Skript für die Simulation gespeichert werden. Lediglich die Vergleiche in der Testbench der inc und dec Leitungen müssen entsprechend angepasst werden.

2.5.3 Synthese

Die Synthese wurde ebenfalls für den Device 5CSXFC6D6F31C6 durchgeführt. Eingestellt bei den generics wurden eine Symbollänge von 128 und eine Bitbreite von 12 Bit. Die Implemen-

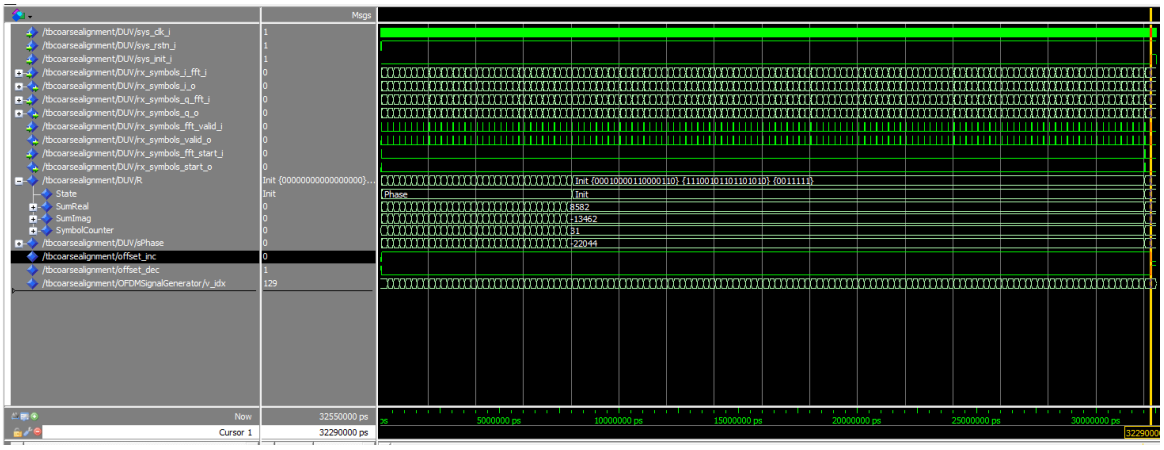

Abbildung 10: Simualtion mit Init-/Reset-Zustand beim Cursor.

tierung könnte für die Synthese noch verbessert werden. Die größer/kleiner Vergleiche für die Quadranten können auf ein gleich abgeändert werden, wenn man auf die Vorzeichen prüft. Register können ebenfalls beim *SymbolCounter* gespart werden, wenn man ihn auf die 32 Symbole beschränkt. Diese Optimierungen wurden nicht mehr durchgeführt. Tabelle 3 zeigt den Ressourcenverbrauch. Die fmax beträgt für das Modell 1100mV 85C (Slow) 253,04 MHz. Das ist ausreichend für die geforderten 100 MHz.

Ressource	**Menge**
Register	42
ALMs	118 / 41,910
Pins (theoretisch)	52 / 499

Tabelle 3: Tabelle mit benötigten Ressourcen.

2.6 Demodulation