Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

HAW HAMBURG

Masterarbeit

Robin Rolf Fröhlich

FPGA-gestützte latenzarme Mustererkennung für analoge Signale im Sub-Gigahertz-Bereich

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Robin Rolf Fröhlich

FPGA-gestützte latenzarme Mustererkennung für analoge Signale im Sub-Gigahertz-Bereich

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Studiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Robert Fitz
Zweitgutachter : Prof. Dr.-Ing. habil. Michael Berger

Abgegeben am 30.10.2018

Robin Rolf Fröhlich

Thema der Masterarbeit

FPGA-gestützte latenzarme Mustererkennung für analoge Signale im Sub-Gigahertz-Bereich

Stichworte

FPGA, IP-Core, SAD, Summe der absoluten Differenzen, Mustererkennung

Kurzzusammenfassung

Im Rahmen dieser Arbeit wird ein IP-Core vorgestellt, welcher basierend auf einem KC705 FPGA-Board als Coprozessor für Mustererkennung genutzt werden kann. Dabei erreicht der IP-Core eine vom FPGA begrenzten Signalabtastrate von 1 Gigasamplespro Sekunde bei einer Auflösung von 14 Bit. Er lässt sich über AXI zur Laufzeit konfigurieren. Es wird ferner ein Testsystem vorgestellt, welches diesen IP-Core verwendet und verifiziert. Für den IP-Core wird der Algorithmus Summe der Absoluten Differenzen zur Mustererkennung verwendet.

Robin Rolf Fröhlich

Title of the master thesis
FPGA-based low latency pattern recognition for analog signals below 1 Gigahertz

Keywords

FPGA, IP-Core, SAD, sum of absolute differences, pattern regocnition

Abstract

In this work a pattern recognition co-processor IP-Core is presented that shall be used on a KC705 FPGA board. It can recognize patterns in 14 bit wide signals with sampling rates up to 1 Gigasamples per second. It is configurable at runtime via AXI. Additionally, a test system is introduced that uses the presented IP-core which is also used for validation of such. For pattern recognition, sum of absolute differences is used as the algorithm of choice.

Inhaltsverzeichnis

Kapitel 1 - Einleitung

1.1 Hintergrund

Mustererkennung ist eine der wichtigsten Fähigkeiten intelligenter Lebewesen. So ist zum Beispiel zu erkennen, was Nahrung ist und was nicht, für die meisten essenziell. Das geschieht auf Basis von Erfahrungswerten, den Vergleichsmustern. Diese werden mit dem neuen Objekt verglichen und anhand dessen wird eine Ähnlichkeit festgelegt.

Allerdings ist die Fähigkeit, Muster zu erkennen nicht nur Lebewesen vorbehalten. Auch in technischen Prozessen ist diese Fähigkeit essenziell. Die Ähnlichkeit zweier Objekte oder Signale zu erfassen ist dabei eine Schlüsselfunktion. Wenn sie sich in entsprechenden Aspekten ähnlich genug sind, werden sie als gleich interpretiert.

Signale haben einen zeitlichen Ablauf, sie können sich wiederholen und lassen sich aufzeichnen, dadurch lassen sie sich sogar später mit einem anderen Signal oder sich selbst vergleichen. Wenn jetzt ein Muster im Signal relevant ist, das regelmäßig auftritt, dann lohnt es sich, dieses zu suchen.

Es ist natürlich keine neue Erfindung, Muster in Signalen zu erkennen und zu suchen. Das wird schon lange erfolgreich gemacht. Auch, das Ganze hochgradig flexibel in Hardware umzusetzen en, ist nicht neu. Auch nicht, analoge Signale in digitale zu wandeln und in diesen zu suchen. Allerdings reichen aufgrund von Taktratensteigerungen und überall steigendem Datenumsatz die bisherigen Entwicklungen nicht mehr aus. Es gibt mittlerweile Anwendungsfälle, die mit den bisherigen Entwicklungen nicht mehr zu machen sind. Daher lohnt es sich, auch das Bekannte neu aufzugreifen und weiter zu entwickeln.

1.2 Anforderungen und Stand der Technik

Gefordert ist die Entwicklung eines IP-Cores, also eines Hardwaremoduls, welches beliebig in weiteren Projekten eingesetzt werden kann. Dieser soll zur Mustererkennung in eindimensionalen Signalen eingesetzt werden und dabei eine höhere Samplerate und Auflösung bieten als bisher verfügbare, vergleichbare Produkte. Mit dem IP-Core sollen möglichst lange Vergleichsmuster möglichst latenzarm mit einem Eingangssignal verglichen und ab einem einstellbaren Schwellwert der Ähnlichkeit ein Signalpuls ausgegeben werden. Weiter ist gefordert, den IP-Core zur Laufzeit aus der Ferne konfigurieren zu können, beispielsweise um das Vergleichsmuster oder bei Verwendung von mehreren Eingangskanälen den Eingangskanal zu ändern.

Der IP-Core soll in ein FPGA-System (Field Programmable Gate Array) integrierbar sein, bei welchem der für die Verwendung von analogen Signalen notwendige ADC (Analog-Digital-Wandler) zum Zwecke der Flexibilität leicht austauschbar ist. Möglich ist dies bei Verwendung von FPGA-Boards, die das von Xilinx mitentwickelte FMC-Stecksystem (FPGA Mezzanine Card, [1]) für Erweiterungskarten verwenden. Für dieses Stecksystem ist eine breite Auswahl an ADC-Erweiterungskarten verfügbar. Als zusätzlicher Anreiz entfällt die Entwicklung eigener Peripherie und Treiber für den ADC bei Nutzung kommerziell verfügbarer ADC Erweiterungskarten, da diese jeweils vom Hersteller der Erweiterungskarten bereitgestellt werden.

Ein vergleichbares, bereits verfügbares Mustererkennungs-Produkt mitsamt ADC ist der icWaves von Riscure [2]. Ein icWaves kann Signale mit bis zu 200 MSpl / s (Mega-Samples pro Sekunde) bei einer Auflösung von 8 Bit verarbeiten. Als Mustererkennungs-Algorithmus verwendet der icWaves dabei die Summe der absoluten Differenzen (SAD). Es handelt sich allerdings um ein hochpreisiges, geschlossenes System.

Es sind ferner FMC-basierte ADC-Erweiterungskarten verfügbar, die eine Abtastrate von 1 GSpl/s bei einer Auflösung von 14 Bit bereitstellen können. Es ist daher das Ziel, Signale mit einer Abtastrate von bis zu 1 GSpl / s und einer Auflösung von 14 Bit verarbeiten zu können.

Als FPGA-System dient für dieses Projekt ein KC705 Evaluationskit [3]. Der auf dem Board montierte XC7K325T FPGA ist ausreichend dimensioniert und das Board verfügt über einen HPC-(High Pin Count)-FMC Steckanschluss, welcher bei der geforderten Samplerate als Anbindung für ADC-Erweiterungsboards notwendig ist. Zur Entwicklung wird die Software

Vivado [4] in Version 2016.4 genutzt. Ferner ist die spätere Nutzung des AD-FMCDAQ2-EBZ [5] von Analog Devices als FMC-basierte ADC-Erweiterungskarte angedacht. Die Schnittstellen des IP-Cores sollen daher auf die Verwendung mit diesem ADC-Board ausgelegt werden.

1.3 Überblick über die Arbeit

Zunächst wird in *Kapitel 2 - Auswahl des Algorithmus* der theoretische Hintergrund über mögliche Mustererkennungsalgorithmen erläutert und anhand des Ressourcenverbrauchs ein Algorithmus ausgewählt.

In *Kapitel 3 -Hardware Architektur* wird anschließend die grundlegende Architektur des entwickelten IP-Cores vorgestellt. Dazu werden die oberste Ebene und die Hauptkomponenten jeweils einzeln erläutert. Bei der obersten Ebene und den Hauptkomponenten handelt es sich um VHDL Module, in denen lediglich Routing und Instanziierung von Untermodulen stattfindet.

Als Einschub wird in *Kapitel 4 -Generelles Transport Protokoll GTP* das im Rahmen dieser Arbeit verwendete Transportprotokoll erläutert. Es dient der Paketierung von seriell, bzw. byteweise versendeten Nachrichten. Der Einschub ist notwendig, da sonst die im folgenden Kapitel vorgestellten Automaten unverständlich wären, da diese das Protokoll verwenden.

Anschließend folgen in *Kapitel 5 -Modulbeschreibungen SAD_CoProc* die Funktionsbeschreibungen der Untermodule, die die eigentlichen Funktionen ausführen.

Für die Hauptkomponenten werden in *Kapitel 6 -Funktionale Simulationen* die für die Verifikation der Modulfunktionen notwendigen, funktionalen Simulationen erläutert und deren Ergebnisse diskutiert. Diese dienen als Beleg für die Funktion des zu entwerfenden IP-Cores.

Da eine funktionale Simulation für die Verifikation nicht ausreicht, wird in den folgenden beiden Kapiteln ein Testsystem vorgestellt, welches den zu entwickelten IP-Core einbindet (*Kapitel 7 -Hardware-Testsystem*) und anschließend werden verschiedene Tests in Hardware durchgeführt (*Kapitel 8 -Hardwaretests*), die die Funktion belegen sollen.

Im letzten Kapitel, *Kapitel 9 -Fazit und folgende Arbeiten*, wird dann ein Fazit über die Arbeit gezogen und auf mögliche Folgearbeiten verwiesen.

Kapitel 2 - Auswahl des Algorithmus

2.1 Hintergrund

Aus der Forderung der hohen Signalabtastrate von 1 GSpl/s ergibt sich, dass ein Mustererkennungsalgorithmus verwendet werden muss, der, unter anderem aufgrund der Forderung nach kurzen Latzenzzeiten, mit wenig komplexer Logik realisierbar sein muss. Daher werden im Folgenden mögliche Algorithmen, die zur Mustererkennung verwendet werden können, gegeneinander abgewogen. Entscheidungsbasis für den zu wählenden Algorithmus wird dabei die Anzahl der auf dem FPGA benötigten Ressourcen sein, gefolgt von Latenzbetrachtungen, die allerdings selbst teilweise vom Ressourcenverbrauch abhängig sind. Die Auswahl der Algorithmen erfolgt hierbei aufgrund verschiedener, bereits durchgeführter Vergleiche, vgl. [6], [7], [8].

Mögliche Algorithmen sind die normierte Kreuzkorrelation (NCC), die Summe der absoluten Differenzen (SAD) und die Summe der quadrierten Differenzen (SSD). Diese werden nachfolgend vorgestellt und einer einfachen Komplexitätsanalyse unterzogen, um den Ressourcenverbrauch abschätzen zu können. Ein tieferer Vergleich wird nicht durchgeführt, da sich alle drei Algorithmen nach [6], [7] und [8] grundsätzlich gut für die Mustererkennung eignen und lediglich im Detail Unterschiede bestehen. Abschließend wird ein Fazit gezogen und ein Algorithmus ausgewählt.

2.2 Normierte Kreuzkorrelation

Die Kreuzkorrelation wird eingesetzt, um die Korrelation zweier ungleicher Signale zu beschreiben. Nachfolgende Erläuterungen sind an [9] angelehnt, allerdings für den Fall optimiert, dass nur ein eindimensionales Signal f der Länge N mit einem einzigen, gleichlangen Vergleichsmuster t korreliert wird.

Für die Aufgabe relevant ist nur die diskrete normierte Kreuzkorrelation, die den Korrelationskoeffizienten r berechnet, welcher zwischen -1 und 1 liegt. Bei zwei vollständig (negativ) linear zusammenhängenden Signalen nimmt dieser den Wert 1 (bzw. -1) an, bei linearer Unabhängigkeit den Wert 0. Er ist definiert über

$$r = \frac{\sum_x [f(x)-\bar{f}\,][t(x)-\bar{t}\,]}{\sqrt{\sum_x [f(x)-\bar{f}\,]^2 \sum_x [t(x)-\bar{t}\,]^2}} \qquad \text{(2-1), vgl. [9]}$$

Da das Vergleichsmuster und dessen Mittel bekannt sind, lassen sich die Terme $[t(x) - \bar{t}\,]$ im Zähler und $\sum_x [t(x) - \bar{t}\,]^2$ im Nenner aus (2-1) vorausberechnen. Diese können daher bei der Komplexitätsanalyse vernachlässigt werden.

Es folgt

$$r = \frac{\sum_x f'(x)t'(x)}{\sqrt{\sum_x f'(x)^2 \;\; t''(x)}} \;\; mit \qquad \begin{matrix} t'(x) = [t(x) - \bar{t}\,] \\ t''(x) = \sum_x [t(x) - \bar{t}\,]^2 \\ f'(x) = \left[f(x) - \bar{f}\,\right] \end{matrix} \qquad \text{(2-2)}$$

Nach (2-2) werden bei direkter Umsetzung zunächst N Additionen und eine Multiplikation für die Mittelwertbildung von f benötigt. Dazu kommen N Additionen für die Bildung von f'. Das Produkt in der Summe kostet für den Zähler N Additionen und N Multiplikationen, für den Nenner werden aufgrund der Quadrierung von f' N weitere Multiplikationen, also N Additionen und $2N$ Multiplikationen benötigt. Zusätzlich werden durch den Bruch eine weitere Multiplikation und eine Wurzelbildung benötigt. Insgesamt werden für die direkte Berechnung der NCC nach (2-2) also $4N$ Additionen, $(4N + 2)$ Multiplikationen und eine Wurzelbildung benötigt.

Andere Verfahren, die die notwendigen Operationen optimieren bzw. reduzieren, wie beispielsweise nach Lewis [9] oder Jea-Chem u.a. [10], sind für andere Anwendungsfälle gedacht. Der Anwendungsfall hier ist laufend ein eindimensionales Muster in einem eindimensionalen Signal zu suchen. Beide zitierten Werke verwenden NCC hingegen zum Erkennen von kleineren Musterbildern MxM in größeren Bildern NxN. Deren Optimierungsergebnisse lassen sich nicht auf den hier vorliegenden Fall übertragen. Daher wird nur die direkte Umsetzung von NCC betrachtet.

2.3 Summe der absoluten Differenzen

Die Summe der absoluten Differenzen ist ein Maß für die Gleichheit zweier Folgen. Es ist für ein Signal f und das Vergleichsmuster t jeweils der Länge N nachfolgend definiert

$$SAD = \sum_x |f(x) - t(x)| \tag{2-3}$$

Es wird die absolute Differenz zweier Folgenwertpaare gebildet und deren Betrag jeweils aufsummiert. Die Summe ist umso geringer, je ähnlicher sich f und t sind. Sind beide identisch, ist sie 0. Die beiden Signale f und t sind dabei gleichwertig, sie können also auch vertauscht werden.

Nachfolgend wird der Algorithmus nach [11] erläutert. Die Folgen f und t liegen im Algorithmus jeweils als n Bit breite Ganzzahlen ohne Vorzeichen vor. Der Algorithmus ist in mehrere Stufen aufgeteilt. Diese sind nachfolgend in Abbildung 1 dargestellt.

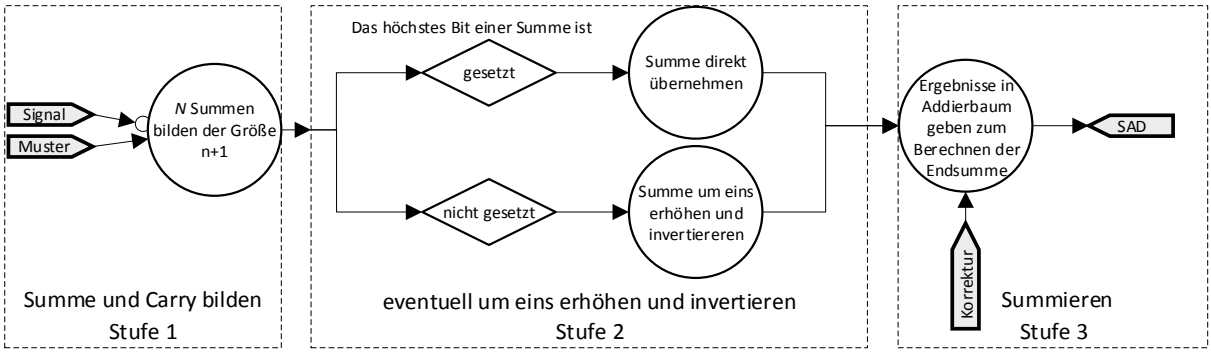

Abbildung 1: Berechnungsstufen in Summe der absoluten Differenzen

Zunächst wird in Stufe 1 für jedes Wertepaar $f(x)$ und $t(x)$ die Summe vom invertierten Signalwert f und dem Musterwert t gebildet, um den kleineren der beiden Werte zu ermitteln. Es gilt

$$|t - f|: \quad t - f > 0, \ wenn\ t > f \qquad \text{(2-4), vgl. [11](6) und (7)}$$

Wenn eine n Bit Ganzzahl ohne Vorzeichen zu ihrem Inversen addiert wird, ist dies stets ihr Maximalwert $2^n - 1$. Dies lässt sich umkehren, die Inverse einer Zahl ist immer die Differenz aus dem Maximum und der Zahl selbst.

Es gilt

$$f + \bar{f} = \sum_{i=0}^{n} 2^i = 2^n - 1 \qquad \text{(2-5) vgl. [11](8)}$$

$$\bar{f} = 2^n - 1 - f \qquad \text{(2-6) vgl. [11](9)}$$

Nach Addition von $2^n - 1$ in(2-4) folgt

$$t - f + 2^n - 1 > 2^n - 1 \qquad \text{(2-7) vgl. [11](12)}$$

Nach Einsetzen von (2-6) in (2-7) folgt

$$t + \bar{f} > 2^n - 1 \qquad \text{(2-8) vgl. [11](13)}$$

$$t + \bar{f} \geq 2^n \qquad \text{(2-9) vgl. [11](14)}$$

Der letzte Schritt ergibt sich aus der Verwendung von Ganzzahlen. Die Addition eines Wertes t mit dem invertierten Wert $\bar{f}$ erzeugt nach also nach (2-9) ein Carry, welches durch ein 2^n wertiges Bit dargestellt wird. Dies gilt jedoch nur, wenn t größer ist als f.

Für den Fall, dass kein Carry erzeugt wird, müssten die Werte getauscht werden. Es werden dann statt t und $\bar{f}$ jeweils $\bar{t}$ und f verwendet. Allerdings lässt sich der Wertetausch im Gegensatz zu den von Vassiliadis u.a. in [11] geschilderten Algorithmus vermeiden, denn es gilt für n Bit breite Ganzzahlen f und t sowie unter Vernachlässigung von Carry-Bits

$$t + \bar{f} = \overline{\bar{t} + f + 1} \qquad \text{(2-10)}$$

Beweis:

$$\overline{\bar{t} + f + 1} = 2^{n+1} - 1 - (\bar{t} + f + 1) \qquad \text{(2-11)}$$

$$= 2^{n+1} - 2 - \bar{t} - f \qquad \text{(2-12)}$$

$$= 2^{n+1} - 2 - (2^n - 1 - t) - \left(2^n - 1 - \bar{f}\right) \qquad \text{(2-13)}$$

$$= 2^{n+1} - 2^{n+1} - 2 + 2 + t + \bar{f} \qquad \text{(2-14)}$$

$$= t + \bar{f} \ \blacksquare \qquad \text{(2-15)}$$

Unter Verwendung der Beziehung nach (2-10) kann deshalb im Algorithmus auf den Wertetausch für Stufe 2 verzichtet werden. Stattdessen wird bei nichtgesetztem Carry zur Summe $t + \bar{f}$ zunächst mit eins addiert und anschließend invertiert.

In beiden Fällen ist das Ergebnis der Addition nun $2^n - 1 + |f - t|$. Da $|f - t|$ immer kleiner als 2^n ist, kann das Carry 2^n, welches durch Stufe 1 entstanden ist, für die weitere Berechnung vernachlässigt werden. Weiterhin folgt, dass jeder Summe der Korrekturterm 1 hinzugefügt werden muss.

In Stufe 3 werden die entstandenen Summen nun mithilfe eines Addierbaumes zu einem einzigen Term summiert. In diesem fließt auch der Korrekturterm gebündelt ein.

Falls vorzeichenbehaftete Signale verwendet werden sollen, so müssen diese vorher mit einem Offset belegt werden, da bei vorzeichenbehafteten Zahlen der Wertebereich gegenüber vorzeichenfreien Zahlen der gleichen Bitbreite um die höchste Stelle verschoben ist. Dies ist in Tabelle 1 anhand von Zahlen mit vier Bit Breite anschaulich dargestellt.

Tabelle 1: Veranschaulichung des notwendigen Offsets bei vorzeichenbehafteten Zahlen

Mit Vorzeichen		Ohne Vorzeichen		Mit Vorzeichen + Offset als vorzeichenfrei	
7	0111	15	1111	15	1111
0	0000	8	1000	8	1000
-8	1000	0	0000	0	0000

Die Differenz zwischen Maximum und Minimum ist bei beiden Zahlenarten identisch, sie beträgt stets $2^n - 1$. Wenn die vorzeichenbehafteten Zahlen mit einem Offset belegt werden (rechte Spalte), so verhalten sie sich ihrem Wertebereich gegenüber identisch zu vorzeichenfreien Zahlen (mittlere Spalte). Da der Offset durch die Subtraktion in Stufe 1 herausfällt, muss er in folgenden Stufen nicht weiter berücksichtigt werden.

Zur Komplexitätsanalyse lässt sich die direkte Betragsbildung des ersten Schrittes nach [11] vermeiden, falls, wie in dieser Arbeit, integrale Datentypen verwendet werden. Dadurch beträgt die Anzahl an notwendigen Additionen $2N$. Im schlimmsten Fall, falls alle Wertepaare kein Carry bilden, sind damit $3N$ Additionen nötig.

2.4 Summe der quadrierten Differenzen

Anders als bei der SAD, werden bei der Summe der quadrierten Differenzen die Differenzen nicht betragsmäßig, sondern quadriert zusammengezählt. Es gilt

$$SAD = \sum_x \big(f(x) - t(x)\big)^2 \tag{2-16}$$

Es sind $2N$ Additionen und verglichen mit SAD anstatt der Betragsbildung nun N zusätzliche Multiplikationen für die Quadratur nötig. Da in dieser Arbeit die Betragsbildung, wie unter 2.3 geschildert, als Addition durchgeführt werden kann, ist die SAD hier klar im Vorteil.

2.5 Zusammenfassung und Fazit

Anders als NCC, dass $4N$ Additionen, $(4N + 2)$ Multiplikationen und einer Wurzelbildung bedarf und SSD, für welches $2N$ Additionen und N Multiplikationen benötigt; bedarf SAD mit nur maximal $3N$ Additionen deutlich weniger Operationen und ist damit für die latenzkritische Echtzeitanwendung auf einem FPGA besser geeignet.

Für die Mustererkennung wird im Rahmen dieser Arbeit daher SAD verwendet, da dieser die geringste Anzahl an Operationen benötigt und damit wie gefordert am wenigsten Ressourcen verbraucht.

Kapitel 3 - Hardware Architektur

3.1 Grundlegendes

Das Rechenwerk, welches für die Eingangsdaten die SAD berechnen soll, wird zusammen mit der nötigen Konfigurationshardware als eigenständiges Paket (IP-Core) entwickelt, welches durch andere Projekte leicht eingebunden und damit leicht wiederverwertet werden kann. Die Konfiguration wird dabei mittels des von Xilinx mitentwickelten AXI4 Busses realisiert.

Später kann der IP Core dann in andere Projekte integriert werden, sofern eine AXI Schnittstelle vorhanden ist. Beispielsweise stellt Analog Devices für das AD_FMCDAQ2-EBZ Board ein Projekt für das KC705 zur Verfügung, welches sich dafür verwenden ließe. Die Dateneingänge und deren Taktrate werden aufgrund dessen an die im AD_FMCDAQ2-EBZ Projekt verwendeten angepasst. Diese sind so ausgelegt, dass 4 Samples auf einmal transportiert werden, so dass die Taktrate von 1 GHz auf 250 MHz reduziert ist. Zur Standardisierung des Interfaces liegen diese Samples in 16 Bit breite vor, von denen 14 Bit effektiv genutzt werden.

Diese Taktrate ist für die Signalverarbeitung der einzelnen Samples zu niedrig. Eine Erhöhung auf 1 GHz ist mit dem verwendeten FPGA nicht möglich, da dessen maximale Taktfrequenz bei 700 MHz liegt. Damit zumindest jedes zweite Sample prozessiert werden kann, wird der Datentakt auf 500 MHz verdoppelt, was die maximal erkennbare Frequenz auf 250 MHz begrenzt.

Der zu entwickelnde IP Core hat ein AXI4 Lite Interface für die Konfiguration (*s0_axi*, *s0_axi_clk* und *s0_axi_resetn*), je einen Datentakt- (*data_clk*) und Reset-Eingang (*data_nres*), zwei Dateneingänge in 64 Bit (4 mal 16) Breite (*new_data_I* und *new_data_Q*) und einen Triggerausgang (*trigger*). Er trägt den Namen *SAD_CoProc*. Sein Blockschaltbild ist nachfolgend in Abbildung 2 dargestellt.

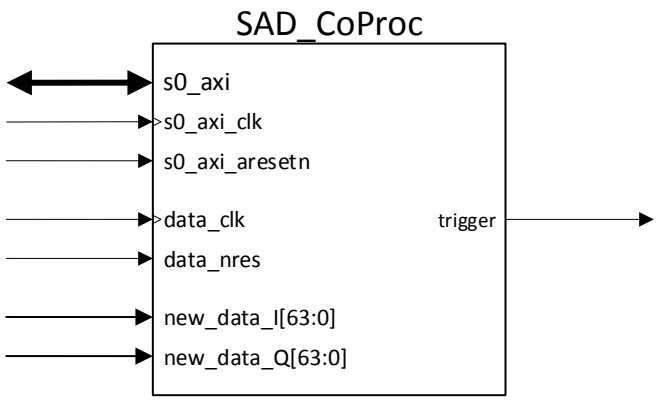

Abbildung 2: Blockschaltbild des *SAD_CoProc* IP Cores.
Der Bus *s0_axi* ist ein AXI4 Lite Port mit vier 32 Bit Registern.

Das Gesamtsystem kann dann beispielsweise mit Hilfe eines MicroBlaze [12] Softcore Prozessors und zusätzlicher Peripherie angelegt werden. Es wird zunächst ein Testsystem fürden Test des IP Core in Hardwareentwickelt. Dieses Testsystem verfügt über eine UART als Konfigurationsschnittstelle, die als COM-Port an einem Windows-PC erreichbar ist, und nutzt die auf dem KC705 vorhandenen DIP Schalter, Knöpfe und LEDs als zusätzliche Benutzerschnittstellen. Das Testsystem wird der Übersicht halber in Kapitel 7 - Hardware-Testsystem erläutert.

Die AXI4 Lite Verbindung zur CPU wird mittels eines Assistenten von Vivado realisiert und konfiguriert. So wird das weiter unten vorgestellte System mit der obersten Ebene eingebunden und die Signale durchgereicht. Es werden dabei vier Register à 32 Bit eingesetzt, wobei von diesen jeweils nur die untersten 8 Bit unidirektional verwendet werden. Es wird daher je ein Register für Datenein- (*axi_rx*) und ausgabe (*axi_tx*), und je eins für ein-(*axi_in_conf*) und ausgehende (*axi_out_conf*) Verbindungskonfiguration genutzt. Die mittels Assistenten entstandenen Module, die für die Verwendung des AXI-Busses benötigt sind, werden im Rahmen dieser Arbeit nicht weiter vorgestellt.

Nachfolgend werden in 3.2 die Hauptkomponenten des *SAD_CoProc* vorgestellt. Die Untermodule werden hingegen in Kapitel 5 -Modulbeschreibungen *SAD_CoProc* genauer erläutert. Die Kapiteltrennung dient hierbei der Übersicht und Abgrenzung von Routing- und Logikmodulen. Ferner wird zwischenzeitlich in Kapitel 4 -Generelles Transport Protokoll GTP noch das notwendige Transportprotokoll für die Konfigurationsdaten erläutert, welches für das Verständnis der Funktion unbedingt notwendig ist.

3.2 Hauptkomponenten des *SAD_CoProc*

3.2.1 Oberste Ebene *app_main*

Um einen modularen Aufbau zu gewährleisten, wird das Rechenwerk von der Konfiguration getrennt. Dadurch kann es unabhängig von diesem getestet werden und umgekehrt. Der IP-Core verfügt daher über drei Komponenten: das SAD Rechenwerk *sad_main*, der Applikationssteuerung *app_fsm* und der Abwicklung der seriellen Schnittstelle über AXI, hier *uart_main.* Ferner werden noch ein Taktverdoppler-Modul (*clk_wiz_0*) als IP-Core, welches den Datentakt von 250 MHz auf 500 MHz erhöht, mit dazu notwendiger Reset-Synchronisation *app_reset_sync* eingesetzt. Zusammengefasst werden diese Komponenten im Modul *app_main*, welches die oberste Ebene bildet. Das Blockschaltbild ist in Abbildung 3 detailliert dargestellt. Wichtig zu erwähnen ist *cpu_axi_res*, dieser Reset dient dem Rücksetzen im Fehlerfall. Speichernde Register wie *samples* bleiben hierbei gesetzt.

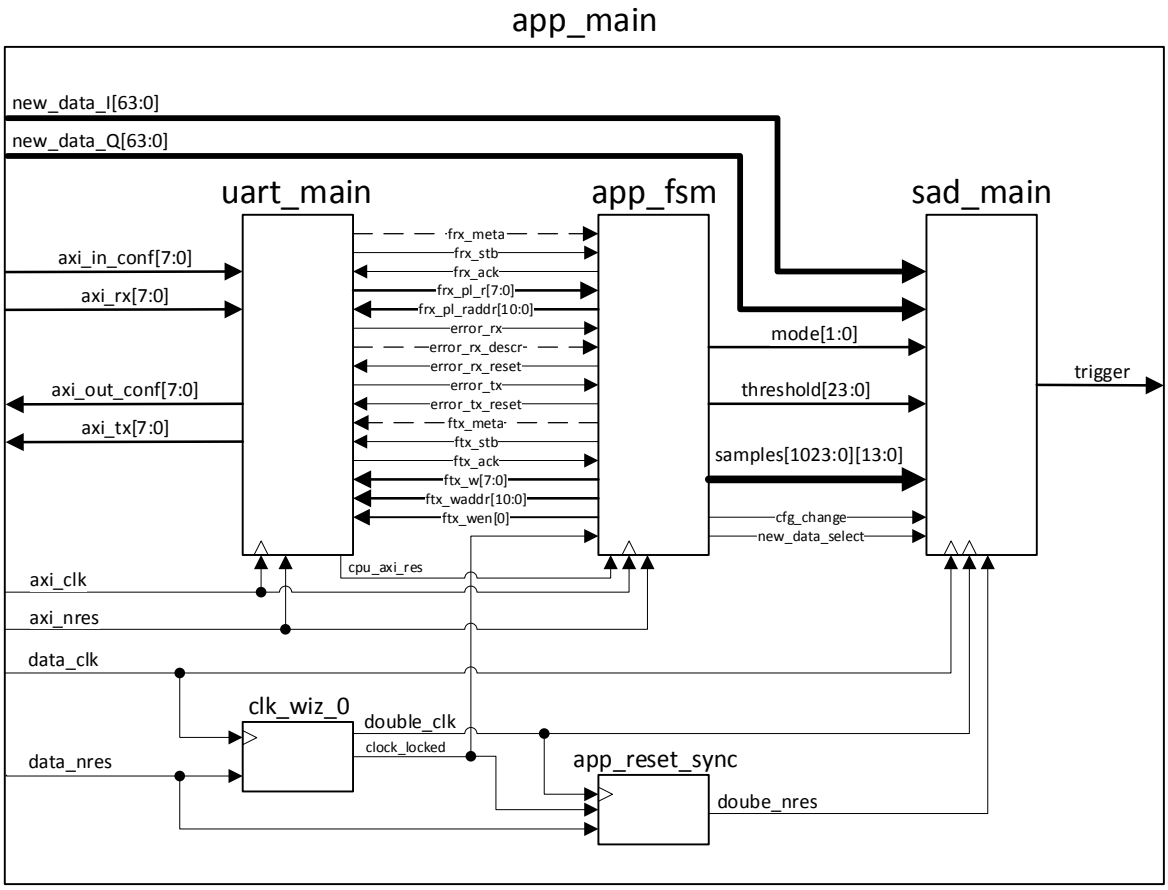

Abbildung 3: Blockschaltbild des Hauptmoduls *app_main*.
Gestrichelte Linien zeigen eigene Datentypen.

3.2.2 Rechenwerk *sad_main*

Das Rechenwerk hat mehrere Aufgaben. Es soll zum einen die Eingangsdaten aufbereiten, sodass diese weiterverwendet werden können. Zum anderen soll es die SAD zwischen Eingangsdaten und einem Vergleichsmuster berechnen. Anschließend soll die SAD dann mit einem Schwellwert verglichen werden, wobei bei Unterschreiten des Schwellwertes ein Trigger-Signal gesetzt werden soll. Die erste und letzte Aufgabe wird im Modul *sad_ctrl* durchgeführt, die Berechnung der SAD in *sad_core*. Zur Überprüfung der Rechenwerte in der Hardware wird ein ILA (*ila_0,* Integrated Logic Analyser, IP-Core, mit dessen Hilfe sich interne Signale im FPGA aufzeichnen/anzeigen lassen) eingebunden, welcher einige Eingangs- und Ausgangswerte des Moduls *sad_core* überwacht. Dieser ist optional für das Debugging und den Test der Hardware. Er wird daher im finalen Design fehlen. Das Blockschaltbild von *sad_main* ist in Abbildung 4 dargestellt.

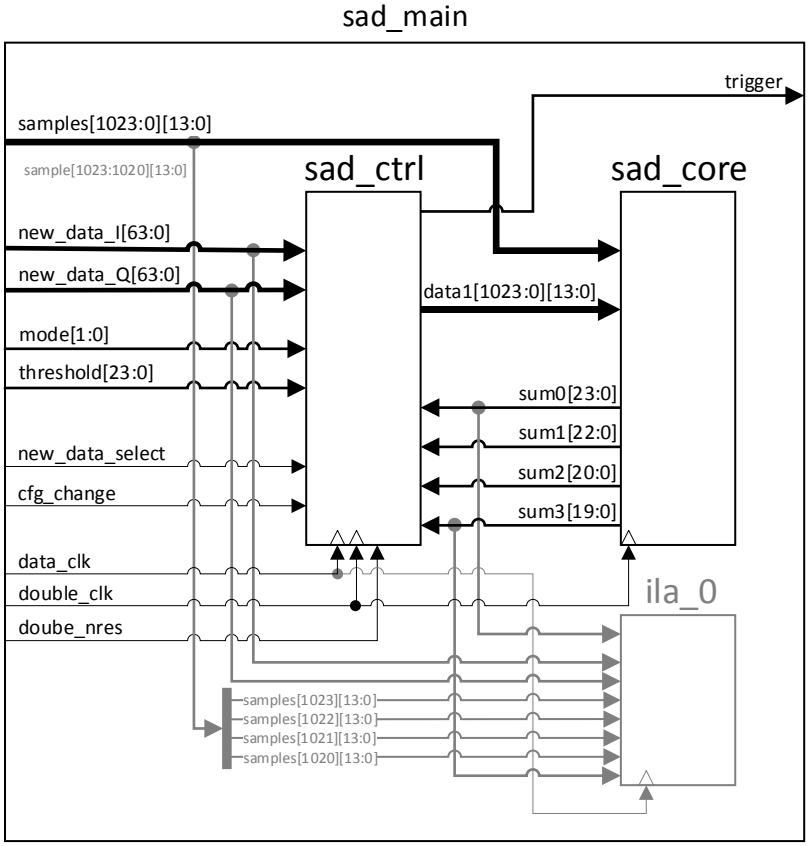

Abbildung 4: Blockschaltbild *sad_main*.
Die Grau dargestellten Elemente sind für das Debugging und im Enddesign nicht enthalten.

3.2.3 Applikationssteuerung *app_fsm*

Die Applikationssteuerung und Konfiguration wird mit Hilfe des Moduls *app_fsm* durchgeführt. Dieses ist als endlicher Automat (FSM)konzipiert. Mit Hilfe des Automaten werden eingehende Datenpakete verarbeitet. Die Funktion des Automaten in Hardware wird mittels eines ILA (*ila_app*) überprüft. Hierzu werden unter anderem die Statusvariable des Automaten (*st*) und weitere interne Signale überwacht. Der ILA wird im Enddesign nicht vorhanden sein, da er nur dem Debugging und Test dient.

Die genauere Funktionsweise des Automaten wird in 5.9 beschrieben, das Blockschaltbild ist in Abbildung 3 enthalten. Es wird an dieser Stelle daher nicht weiter auf die Funktion eingegangen.

Datenpakete bestehen aus einem zwei Byte langen Operationscode (OP-Code) und von diesem abhängig einer bestimmten Anzahl an Nutzdaten. Dies ist angedacht, damit leicht weitere Funktionen zu den bereits Bestehenden hinzugefügt werden können. Es sind OP-Codes zum Wechseln des Vergleichsmusters mit 1024, 512, 128 oder 64 Samples Länge, zum Wechseln des Kanals und Einstellen des Schwellwertes verfügbar. Weitere OP-Codes können bei Bedarf leicht hinzugefügt werden. Ein Abfragen der aktuellen Werte ist, bis auf den aktuellen Kanal, ebenfalls vorgesehen und möglich. Dies wird ferner in Kapitel 4 -Generelles Transport Protokoll GTP erläutert.

3.2.4 Serielle Schnittstelle *uart_main*

Die Abwicklung der ein- und ausgehenden Daten über AXI übernimmt das Modul *uart_main*. Dessen Blockschaltbild ist nachfolgend in Abbildung 5 dargestellt.

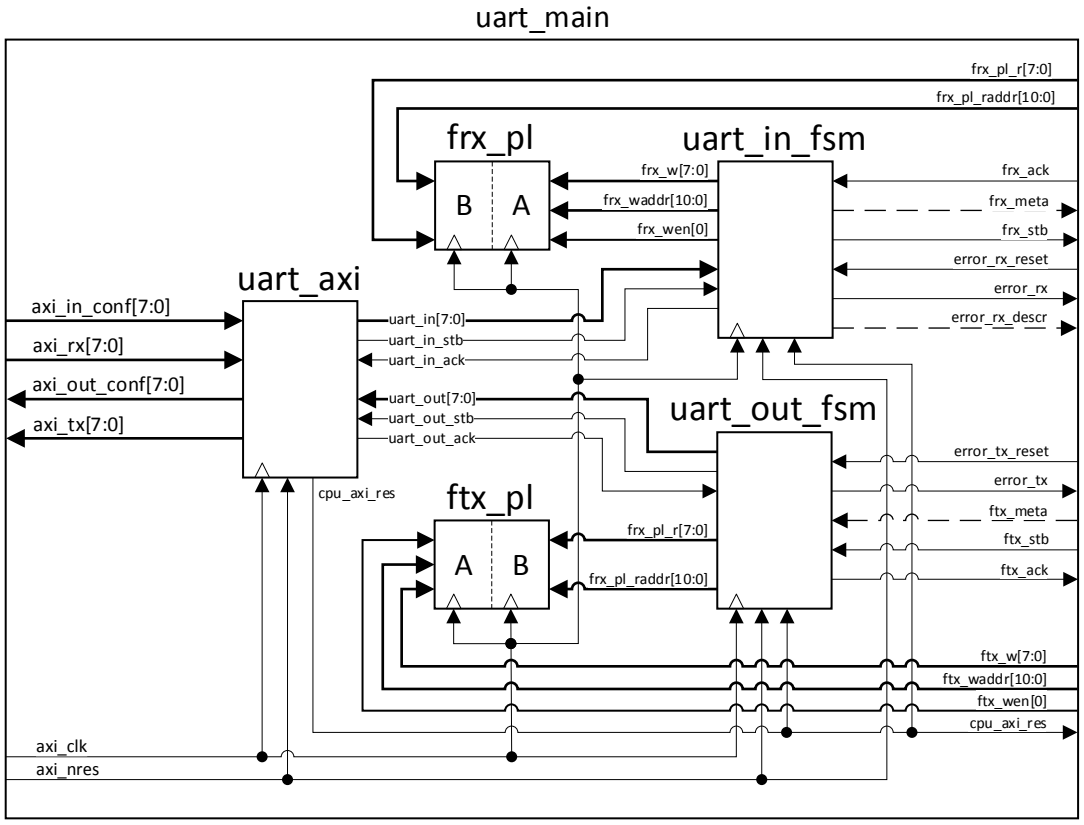

Abbildung 5: Blockschaltbild *uart_main*.

Es ist aufgeteilt in *uart_axi*, welches einzelne Bytes von bzw. zu der übergeordneten AXI Schnittstelle weiterreicht; *uart_in_fsm*, welches die eintreffenden Bytes wieder in Pakete zusammenfasst; *uart_out_fsm*, welches die ausgehenden Pakete in einzelne Bytes aufteilt. Ferner verfügt die Komponente über zwei Block-RAM Module, in denen Daten von ein- (*frx_pl*) bzw. ausgehenden (*ftx_pl*) Konfigurationspaketen - sofern nötig - zwischengespeichert werden. Die Konfigurationspakete werden im nachfolgenden Kapitel 4 -Generelles Transport Protokoll GTP weiter erläutert.

3.3 Notwendige Timing- und Routing-Constraints

Einige Verbindungen überschreiten Taktdomänen. Es muss daher sichergestellt werden, dass diese Taktdomänenüberschreitungen von Vivado funktional korrekt umgesetzt werden können. Dazu werden so genannte Timing-Contraints genutzt. Ferner kann es durch hohe Auslastung des FPGA zu langen Signalpfaden kommen, die nicht mehr innerhalb einer Taktperiode zurückgelegtwerden können. Um den Place &Route Algorithmus bei der Synthese zu entlasten, gibt es spezielle Timing-Constraints, zum Beispiel False- und Multicylce-Paths. False-Paths werden beispielsweise bei zeitlich unkritischen Konfigurationsregistern eingesetzt, die sich im laufenden Betrieb nicht häufig oder gar nicht ändern. Multicycle-Paths hingegen werden eingesetzt, wenn genauer definiert werden soll, wie viele Taktzyklen die Setup- und Hold-Zeit jeweils betragen sollen.

Nachfolgend sind in Listing 1 die für SAD_CoProc notwendigen internen Constraints aufgelistet, sie werden im Folgenden nicht weiter erläutert. Die als Kommentar stehenden externen Constraints binden den Ausgang *trigger* zu dem USER_SMA_GPIO_N Ausgang des KC705, dies muss das übergeordnete Modul übernehmen, welches den IP-Core einbindet.

Listing 1: Timing-Constraints des SAD_CoProc, *sad_coproc.xdc*.

```
#multiple paths that are not relevant in timing, as mostly configuration registers, that may
#be changed, but that change is rare

set_false_path -from [get_pins -hierarchical \
        -filter { NAME =~  "*samples_ff_reg*/C*" && NAME !~      "*CE*" }]\
        -to [get_pins -hierarchical *s1_sum_reg*/D*]

set_multicycle_path 4 -from [get_pins -hierarchical \
        -filter { NAME =~  "*channel_select_reg/C" && NAME !~  "*CE*" }] \
        -to [get_pins -hierarchical *data_reg_reg*/D*]
set_multicycle_path 1 -hold -from [get_pins -hierarchical \
        -filter { NAME =~  "*channel_select_reg/C" && NAME !~  "*CE*" }] \
        -to [get_pins -hierarchical *data_reg_reg*/D*]

set_false_path -from [get_pins -hierarchical \
        -filter { NAME =~  "*threshold_ff_reg*/C*" && NAME !~  "*CE*" }]

set_multicycle_path 4 -from [get_pins -hierarchical \
        -filter { NAME =~  "*mode_reg*/C*" && NAME !~  "*CE*" }] \
        -to [get_pins -hierarchical *diff_reg*/D*]
set_multicycle_path 1 -hold -from [get_pins -hierarchical \
                -filter { NAME =~  "*mode_reg*/C*" && NAME !~  "*CE*" }] \
                -to [get_pins -hierarchical *diff_reg*/D*]

set_multicycle_path 4 -from [get_pins -hierarchical \
        -filter { NAME =~  "*cfg_changed_reg/C" && NAME !~      "*CE*" }] \
        -to [get_pins -hierarchical *trigger_change_shrg_reg*/D*]
set_multicycle_path 1 -hold -from [get_pins -hierarchical \
        -filter { NAME =~  "*cfg_changed_reg/C" && NAME !~      "*CE*" }] \
        -to [get_pins -hierarchical *trigger_change_shrg_reg*/D*]

#to be added in higher level constraint
#USER_SMA_GPIO_N
#set_property PACKAGE_PIN Y24 [get_ports trigger]
#set_property IOSTANDARD LVCMOS25 [get_ports trigger]
```

Kapitel 4 - Generelles Transport Protokoll GTP

4.1 Grundlegendes

Um für die Übertragung ein standardisiertes Protokoll Transportprotokoll zu verwenden, was auch andere Applikationen nutzen können, wird das „Generelle Transport Protokoll“ (GTP) verwendet. Es ist ein schlankes Protokoll, welches nachfolgendend vorgestellt wird.

Das Protokoll verwendet ein Byte als Header, das so genannte Config-Byte (CB), in der Regel gefolgt von einem Feld LF, welches die Länge N der Nutzdaten, im folgenden Payload genannt, enthält. Danach folgen N Nutzdaten und eventuell ein Feld LEF über die Länge M der vom Gerät zu sendenden Daten und eventuell eine Prüfsumme XOR1. Im Protokoll wird als Byte-reihenfolge Big Endian verwendet.

4.2 Config-Byte

Die Bits des Config-Byte sind nachfolgend in Tabelle 2 definiert.

Tabelle 2: Bitbeschreibungen GTP Config-Byte. Verwendet wird die in C typische Schreibweise zur Bitklassifikation.

Bit	Name	Funktion, wenn gesetzt
0x80	RFU	Reserviert
0x40	ASCII/BIN	Protokoll in ASCII, sonst binär
0x20	RFU	Reserviert
0x10	NAK	Kommunikationsfehler, nochmal senden/ Fehlercode folgt, wenn ACK auch gesetzt ist.
0x08	ACK	Bestätigung, Nachricht erfolgreich gesendet / Fehlercode folgt, wenn NAK auch gesetzt ist
0x04	SERVICE	Service Nachricht folgt.
0x02	LEX	Ein LE Feld ist angehängt, dieses beschreibt die Länge der geforderten Antwort
0x01	XOR1X	Eine XOR1 Prüfsumme ist angehängt.

Da in dieser Arbeit nur binär kommuniziert wird, wird die ASCII Funktionalität nicht erläutert und auch nicht seitens des entworfenen Systems unterstützt. Falls ein Fehlercode gesendet werden soll, kann das Config-Byte nur den Wert 18_{16} annehmen, bei einer Service-Nachricht nur 04_{16}. In beiden Fällen entfallen Längenfelder und Prüfsumme. Im Falle eines ACK oder NAK wird das Config-Byte alleine gesendet.

4.3 Längenfelder LF und LEF

Die Längenfelder LF und LEF sind so konzipiert, dass, sofern die Länge N beziehungsweise M kleiner als 127 ist, sie ein Byte lang sind und diese Länge enthalten. Dieses erste Byte trägt dabei den Namen L0 bzw. LE0. Falls N bzw. M größer sind, enthält das erste Byte die Länge J bzw. K, und es folgen J bzw. K Bytes, die dann die Länge N bzw. M darstellen. Die Unterscheidung zwischen beiden Stellen wird mittels des hochwertigsten Bits vorgenommen. Dies ist nachfolgend in Tabelle 3 dargestellt.

Tabelle 3: Inhalt der Längenfelder LF bzw. LEF.

Bedingung	Wert von 0x80	Restliche Bits	Folgend
N / $M < 127$	0	N / M	-
N / $M \leq 127$	1	J / K	J/K Bytes, die N / M darstellen

4.4 Prüfsumme XOR1

Die Prüfsumme XOR1 wird mittels XOR von allen Bytes eines Paketes bis auf die mitgesendete Prüfsumme berechnet und anschließend um eins verringert. Das Ergebnis wird dann angehängt bzw. mit der mitgesendeten Prüfsumme verglichen. Falls diese nicht identisch sein sollten, ist das gesamte Paket ungültig und muss erneut gesendet werden. Dies wird mittels Fehlercode dem jeweiligen Sender angezeigt.

4.5 Verwendung innerhalb von *SAD_CoProc* und OP-Codes

Innerhalb von *SAD_CoProc* wird das Protokoll normal verwendet, es werden aber weder die ASCII Funktionalität noch Service-Nachrichten unterstützt. Das System initiiert von sich aus nie eine Nachricht. Es wartet stets, bis es Nachrichten in Form von sogenannten OP Codes erhält und quittiert diese mit einem ACK. Ausgaben seitens des Systems werden mittels passendem OP-Code und passender geforderter Länge M unterstützt.

Die Länge des Payloads N ist immer mindestens 2, um einen OP-Code unterzubringen, plus die Anzahl der zum OP-Code gehörenden Daten. Die verfügbaren OP-Codes sind nachfolgend in Tabelle 4 aufgelistet. Wie bereits weiter oben erwähnt, ließen sich zum Zwecke der Erweiterung leicht neue hinzufügen.

Tabelle 4: Verwendbare OP-Codes zur Konfiguration von *SAD_CoProc*

OP-Code	Name	Beschreibung	N	M
00 10	SAD Mode 00 - 1024	1024 Samples einstellen/abfragen	2050	2048
00 11	SAD Mode 01 - 512	512 Samples einstellen/abfragen	1026	1024
00 12	SAD Mode 10 - 128	128 Samples einstellen/abfragen	258	256
00 13	SAD Mode 11 - 64	64 Samples einstellen/abfragen	130	128
00 20	threshold	Schwellwert einstellen/abfragen	5	3
00 40	channel I	Kanal I verwenden	2	X
00 41	channel Q	Kanal Q verwenden	2	X

Kapitel 5 - Modulbeschreibungen *SAD_CoProc*

In diesem Kapitel werden die tieferliegenden VHDL Module innerhalb des entwickelten SAD_CoProc IP-Cores genauer beschrieben. Es wird der Übersicht halber allerdings zumeist nur eine sinngemäße Dokumentation verwendet, so dass zum Beispiel Automatengraphen nicht vollständig beschrieben werden, stattdessen wird sinngemäß dokumentiert, was in welchen Zustand passiert. Es werden die im vorangegangen Kapitel eingeführten Größen bezüglich GTP eingesetzt.

5.1 Modul *sad_ctrl*

Innerhalb des Modul *sad_ctrl* werden die eingehenden Signaldaten synchronisiert, sortiert, mit dem notwendigem Offset belegt und parallelisiert. Gleichzeitig wird auch das ausgehende *trigger* Signal aus verschiedenen SAD der jeweils gerade durch *mode* eingestellten Länge durch Subtraktion mit dem Schwellwert *threshold* berechnet. Ein Trigger wird ausgelöst, sobald die Differenz, *diff*, kleiner oder gleich 0 ist. Das Blockschaltbild des Moduls ist in Abbildung 6 dargestellt. Der obere Pfad zeigt dabei die Datenverarbeitung und der untere die Trigger-Berechnung.

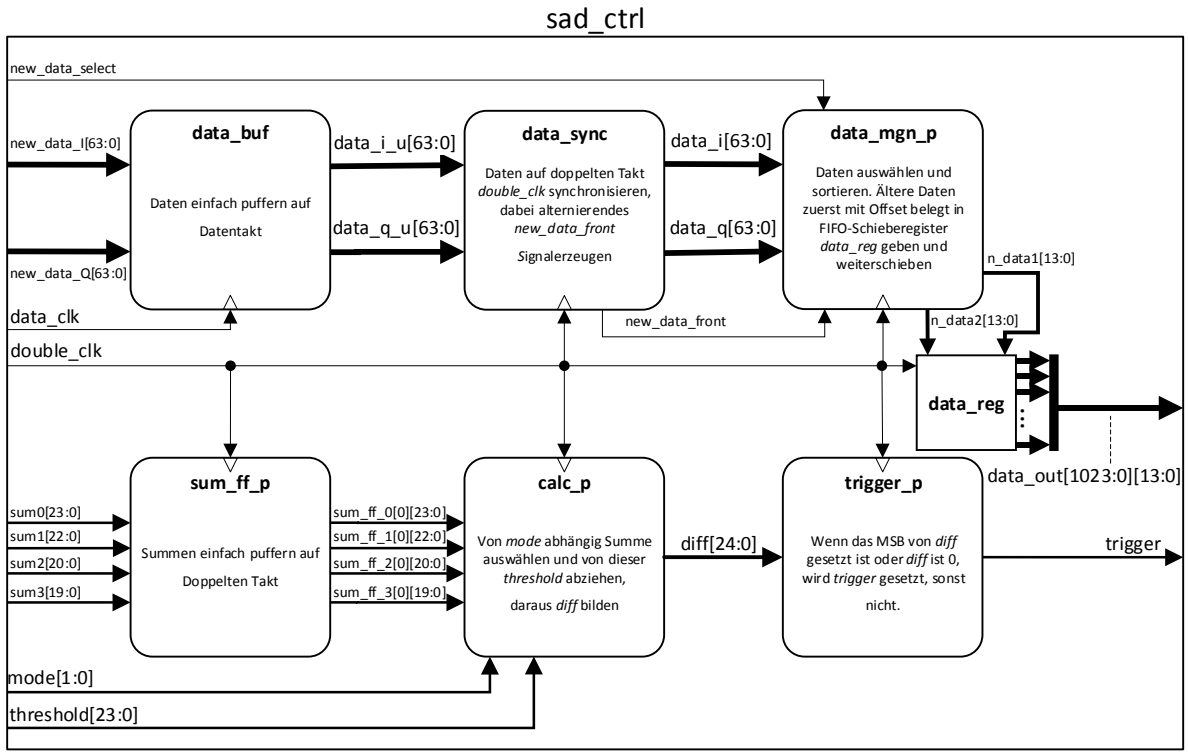

Abbildung 6: Blockschaltbild *sad_ctrl* inklusive beinhalteten Prozessen (abgerundete Ecken).

5.2 Modul *sad_core*

Im Modul *sad_core* findet die eigentliche Berechnung der SAD statt. Diese ist in drei Stufen aufgebaut. Die Stufen 1 und 2 entsprechen dabei der Differenz aus Datum und Musterdatum mit anschließender Betragsbildung, welche für die Bildung der SAD notwendig ist. Stufe 3 beinhaltet die Summation der einzelnen Differenzen für die finale Berechnung der SAD. Die Funktionsweise ist in Abbildung 7 dargestellt.

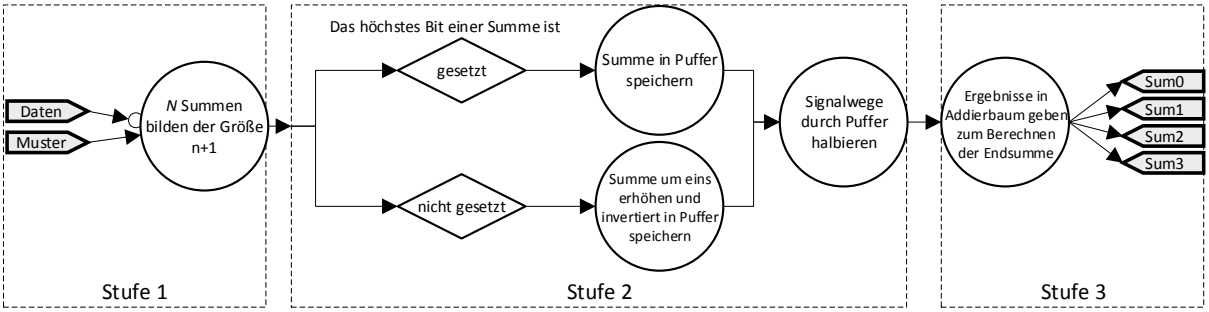

Abbildung 7: Funktionsweise des Moduls *sad_core*, Stufe 1 bis 3.

Es werden in Stufe 1 jeweils die zueinander gehörenden invertierten Eingangs- und nicht invertierten Vergleichsdatenwerte zu einer ein Bit breiteren Summe addiert, da ja, wie in 2.3 erläutert, ein Carry gebildet werden soll In Stufe 2 wird dann geprüft, ob das höchstwertige Bit der jeweiligen Summe gesetzt ist. Falls dies so ist, wird die entstandene Summe invertiert und anschließend um eins erhöht übernommen; falls nicht, wird sie direkt übernommen. Die übernommenen Summen werden zwischengepuffert.

Der Puffer ist notwendig, um der starken Ausdehnung der einzelnen Logikzellen auf dem FPGA Rechnung zu tragen, da bei der verwendeten Sampleanzahl von 1024 Samples eine hohe Auslastung der benötigten Logikzellen erreicht wird. Ein pünktliches Erreichen der Werte am nächsten Register ist bei langen Strecken aufgrund der endlichen Signallaufzeit nicht innerhalb eines Taktes möglich. Stattdessen kann die Strecke halbiert werden, indem einmal zwischengespeichert wird. Dies kostet zwar einen zusätzlichen Takt und ein zusätzliches Register pro Signal, was aber dank der Zellbelegung im KC705 FPGA kein Problem darstellt. Jede Zelle verfügt über zwei unabhängige Flipflops, damit sind ausreichend Flipflops vorhanden.

Weiter werden die Daten dann in Stufe 3 auf Addierbäume gegeben. Dafür werden zunächst 16 Addierbaummodule, *csa_65to1* (siehe 5.4), genutzt. Diese reduzieren die zu summierenden Terme von 64 plus einem Korrekturkern auf einen Summenwert. Für Musterlängen von 64 Samples reicht diese Reduzierung bereits (*sum3*). Ansonsten werden noch eine 16 zu eins, *csa_16to1*, für 1024 Sample lange Muster (*sum0*); eine 8 zu eins, *csa_8to1*, für 512 Sample lange Muster (*sum1*); und eine Addition der beiden hochwertigsten Summenterme, also eine 2 zu 1 Reduktion (*sum2*) vorgenommen. Basis der Addierbäume, die in 5.4 erläutert werden, sind die Volladdierermodule *csa_fa* und *csa_fa_limited*. Diese werden nachfolgend in 5.3 erläutert.

Da die Samples an dieser Stelle nicht mit einem Offset belegt sind, muss dies bereits vor der Übertragung passieren. Dies wird durch die Konfigurationssoftware gewährleistet.

5.3 Module *csa_fa* und *csa_fa_limited*

Das Modul *csa_fa* reduziert drei k Bit breite Signale zu zwei k + 1 breiten Signalen als Carry-Save-Addierer [13]. Der Wert k ist hierbei variabel, so dass das Modul mehrfach verwendet werden kann. Das Modul *csa_fa_limited* unterscheidet sich insoweit, dass Ein- und Ausgangssignale jeweils beide k Bit breit sind. Dies wird genutzt, da die maximale Summe bekannt ist (siehe 5.4) und weitere Stellen nicht von Nöten sind.

Es werden als Eingänge die Signale *A, B* und *C* und als Ausgänge *P* und *S* definiert. P und S berechnen sich folgend in *csa_fa:*

$$\begin{aligned} S(i = k \dots 0): \quad & S(i) = A(i) \oplus B(i) \oplus C(i) \\ & S(k+1) = 0 \\ P(i = k \dots 1): \quad & P(i+1) = \big(A(i) \wedge B(i)\big) \vee \big(A(i) \wedge C(i)\big) \vee \big(B(i) \wedge C(i)\big) \\ & P(0) = 0 \end{aligned} \tag{5-1}$$

Entsprechend in *csa_fa_limited*:

$$\begin{aligned} S(i = k \dots 0): \quad & S(i) = A(i) \oplus B(i) \oplus C(i) \\ P(i = k \dots 1): \quad & P(i+1) = \big(A(i) \wedge B(i)\big) \vee \big(A(i) \wedge C(i)\big) \vee \big(B(i) \wedge C(i)\big) \\ & P(0) = 0 \end{aligned} \tag{5-2}$$

5.4 Module *csa_65to1*, *csa_16to1* und *csa_8to1*

Aus der Kaskadierung von Carry-Save-Addierern lassen sich Addierbäume konstruieren. Es lässt sich bis auf maximal zwei Signale, welche die Stufe unbearbeitet passieren oder anderweitig addiert werden, eine Reduktion pro Stufe von 3 : 2 erreichen. Mit ausreichend Stufen bleiben am Ende 2 Summenterme übrig, die mit Hilfe regulärer Addierer zu einem Summenterm zusammengefasst werden. Ein solcher Addierbaum ist beispielhaft anhand von *csa_16to1* in Abbildung 8 dargestellt.

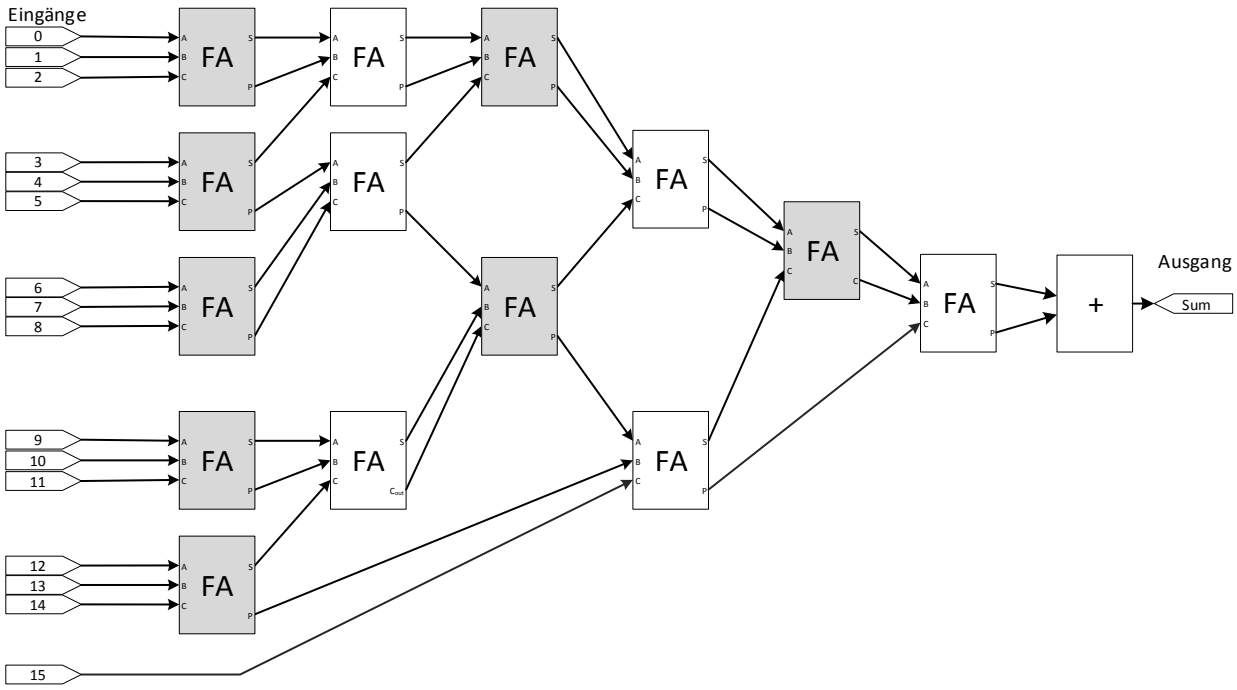

Abbildung 8: Modulbeschreibung csa_16to1. Es werden hier ausschließlich csa_fa_limited eingesetzt, da bereits die größtmögliche Bitbreite erreicht ist. Diese werden zwecks Übersicht nicht gesondert gekennzeichnet.

Wie zu erkennen, werden die Eingänge stufenhaft addiert und in der Anzahl reduziert. Die überzähligen Werte überspringen dabei falls nötig einige Stufen, zum Beispiel bei Eingang 15. In jeder Stufe steigt die Bitbreite um eins, bis die maximale Bitbreite von 24 Bit erreicht worden ist. Danach steigt diese nicht weiter, da die maximal mögliche SAD von 1024 Wertepaaren 14 Bit breiter Zahlen nur $2^{10} \cdot (2^{14}\text{-}1) = 2^{24} - 2^{10}$ betragen kann, was sich mit 24 Bit vollständig darstellen lässt.

Bei der 65 zu 1 Summation in *csa_65to1* werden zusätzlich in der zweiten Stufe die notwendigen Korrekturterme addiert, die durch den Schritt der Betragsbildung notwendig sind.

5.5 Modul *uart_axi*

Das Modul *uart_axi* verfügt über zwei Prozesse, einen für die eingehenden Daten, den anderen für die ausgehenden Daten. Diese sind jeweils in Abbildung 9 und in Abbildung 10 vollständig dargestellt. Der Eingangsprozess *input* sorgt dabei für den Datenverkehr vom AXI-Bus in Richtung *uart_in_fsm*, der Ausgangsprozess *output* schreibt den Datenverkehr von *uart_out_fsm* auf den AXI-Bus. Zusätzlich wird der Wert *conf_i.res* als CPU basierter Reset an die anderen Module weitergegeben als *cpu_axi_res*.

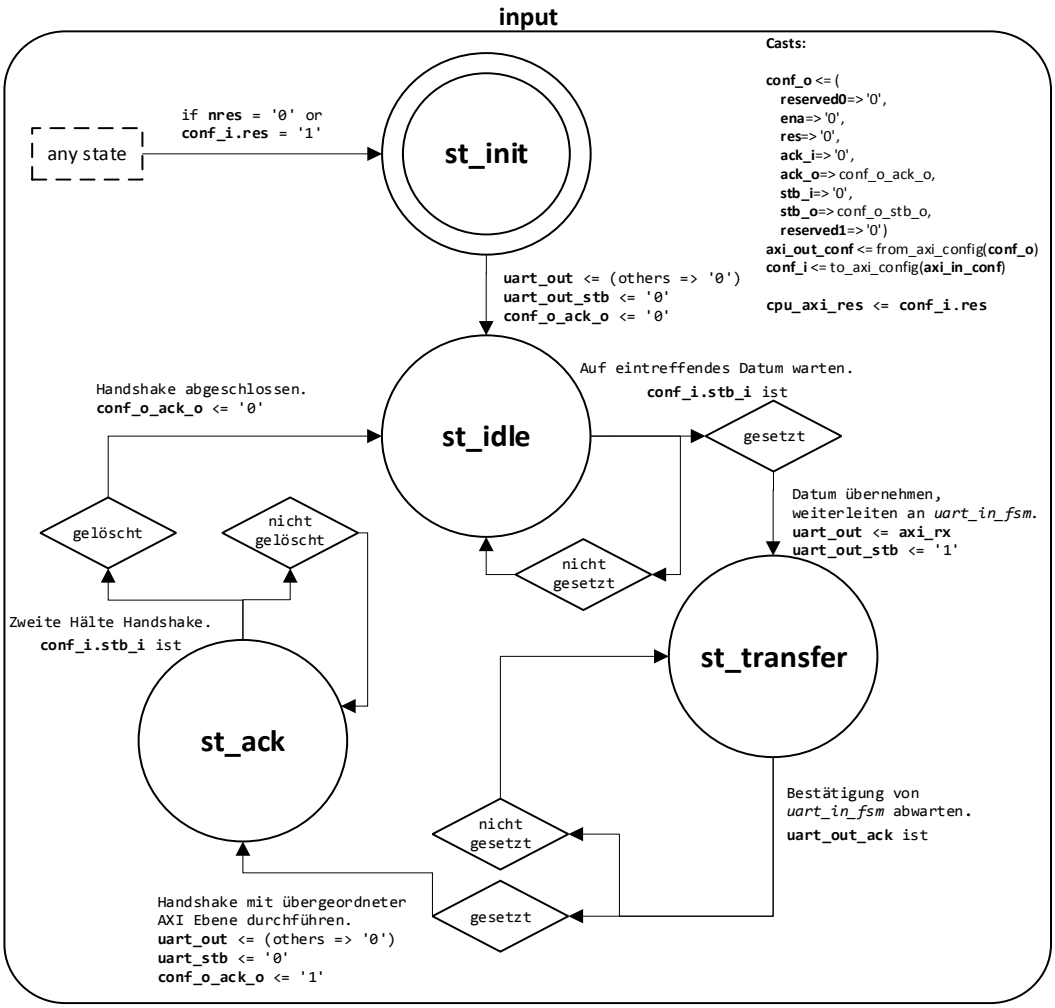

Abbildung 9: Automatengraph für den Prozess *input* des Moduls *uart_axi*.

Der Prozess *input* folgt dabei einen Kreislauf. Es wird gewartet, bis ein Datum von AXI aus anliegt, welches seitens der übergeordneten AXI Stelle mittels setzen des Bits *std_i* im *conf_i* Register geschieht. Wenn dem so ist, so wird dieses Datum an das Modul *uart_in_fsm* weitergeleitet, indem *axi_rx* auf das Register *uart_out* weitergeleitet und *uart_out_stb* gesetzt wird.

Anschließend wird auf die Bestätigung der Verarbeitung seitens *uart_in_fsm* gewartet, was dieses durch Setzen von *uart_out_ack* kennzeichnet. Im Anschluss wird ein doppelter Handshake mit der übergeordneten AXI Stelle durchgeführt, indem zunächst *conf_o.ack_i* gesetzt und geprüft wird, ob *conf_i.stb_i* gelöscht worden ist. Wenn es dann gelöscht wurde, wird *conf_o.ack_i* ebenfalls gelöscht und es kann ein weiteres Datum empfangen werden.

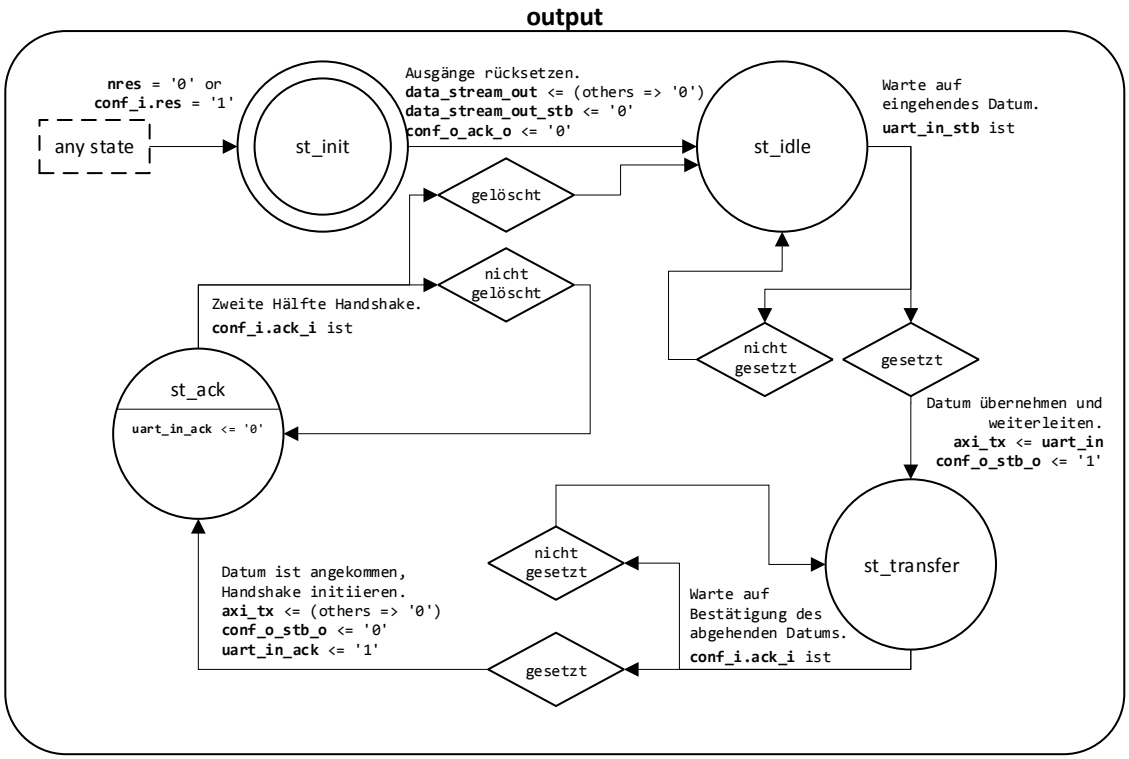

Abbildung 10: Automatengraph für den Prozess *output* des Moduls *uart_axi*.

- Auch der Prozess *output* folgt einem Kreislauf. Hier werden die Daten des Moduls *uart_out_fsm* weitergeleitet. Dies wird signalisiert durch setzen von *uart_in_stb*. Sobald dies gesetzt ist, werden die Daten durch Schreiben vom Register *uart_in* auf den AXI-Bus *axi_tx* und das Setzen von *conf_o._stb_o* and die übergeordnete AXI-Stelle weitergeleitet. Anschließend wird auf Beginn des doppelten Handshakes seitens der AXI Stelle gewartet, was mit gesetztem *conf_i._ack_i* signalisiert wird. Bei erfolgreichem erstem Handshake werden *axi_tx* und *conf_o.stb_o* wieder gelöscht und *uart_out_fsm* wird mittels Setzen von *uart_in_ack* die erfolgreiche Prozessierung mitgeteilt. Danach wird *uart_in_ack* wieder gelöscht, da dieses nur einen Takt lang gesetzt sein darf. Ferner wird auf die zweite Hälfte des Handshakes seitens der AXI-Stelle gewartet, hierfür muss *conf_i._ack_i* von dieser gelöscht werden. Im Anschluss kann ein neues Datum an AXI weitergeleitet werden.

5.6 Modul *uart_in_fsm*

Im Modul *uart_in* werden die eingehenden Daten von der übergeordneten AXI-Stelle, die bereits durch *uart_axi* empfangen wurden, in GTP Pakete verpackt, damit diese durch *app_fsm* verarbeitet werden können. Die Funktionalität von *uart_in* ist nachfolgend vereinfacht in Abbildung 11 dargestellt, da ein vollständiger Automatengraph an dieser Stelle aufgrund der Komplexität des Automaten zu unübersichtlich wäre. Es werden ferner die in Kapitel 4 -Generelles Transport Protokoll GTP eingeführten Größen genutzt.

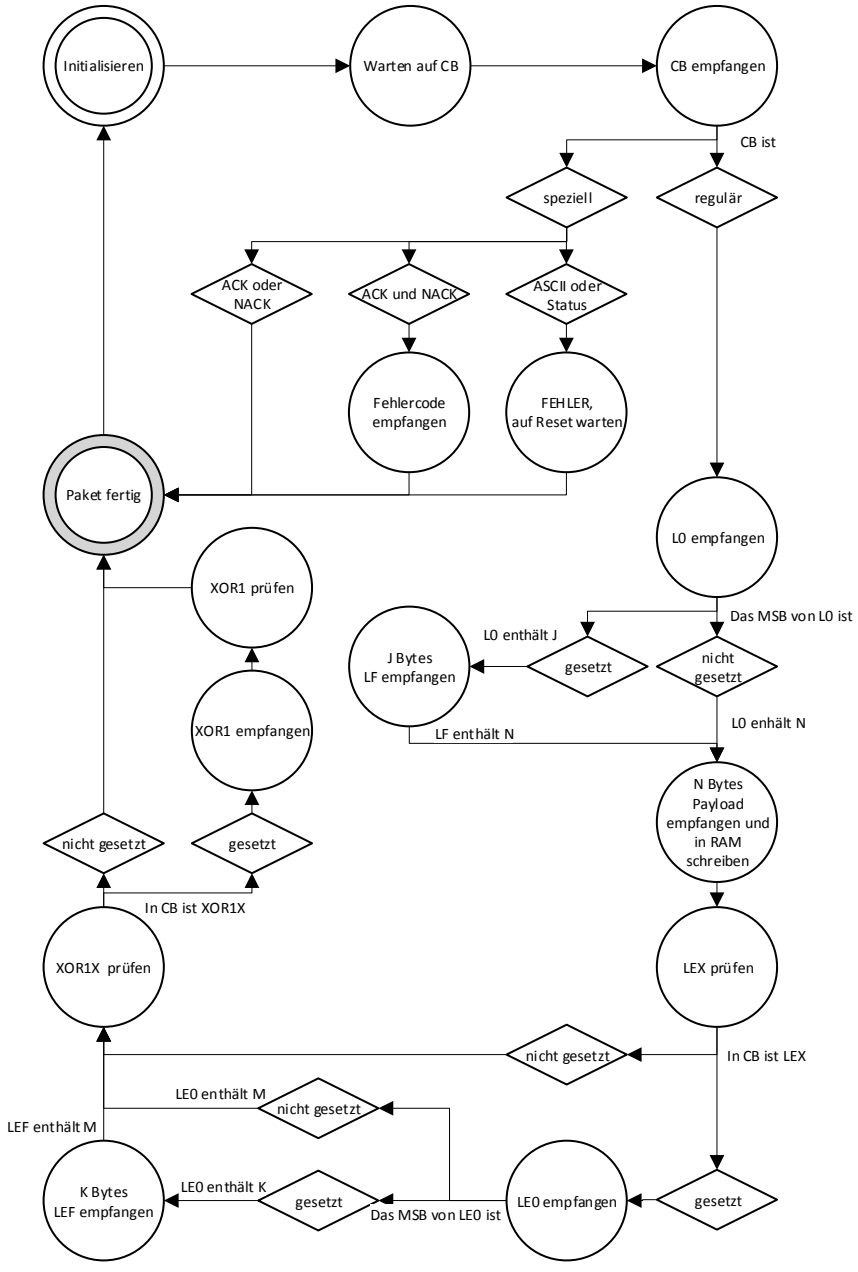

Abbildung 11: Vereinfachter Automatengraph des Moduls *uart_in*.

Es wird zunächst auf ein einkommendes, initiales Byte gewartet. Dies geschieht durch Prüfen des Registers *uart_out_stb* (hier *out* genannt, da es *uart_axi* verlässt). Ist dieses gesetzt, signalisiert *uart_axi* hiermit ein empfangbares Byte. Dieses wird übernommen und durch einen Takt langes Setzen von *uart_out_ack* quittiert. Dies gilt ebenfalls für alle nachfolgenden Bytes. Ferner werden alle Informationen, wie das Config-Byte CB, die Nachrichtenlänge N, die Länge der angeforderten Nachricht M und der OP-Code in *fout_meta* (hier *out* genannt, da es *uart* verlässt) gespeichert, sobald sie bekannt sind. Ebenso wird jedes eintreffende Byte eines Paketes zur Berechnung der internen Prüfsumme genutzt.

Dieses empfangene, erste Byte wird als Config-Byte interpretiert. Für den Fall, dass es sich um einen Fehlercode, eine Statusmeldung, ein ACK oder ein NAK handelt oder ASCII gesetzt ist (siehe Kapitel 4 - GTP), gilt das Paket als speziell, ansonsten gilt es als regulär. ASCII und Status dürfen hierbei nicht gesetzt sein, da dieses System beides nicht unterstützt. Es wird bei Empfang beider jeweils ein Fehlerzustand signalisiert, der bis zum externen Rücksetzen des Fehlers gehalten wird. Ein GTP-Fehlercode, also ein CB mit gesetztem ACK und NAK, wird noch zu Ende empfangen. Ansonsten, also bei ACK oder NAK oder empfangenen Fehlercode, ist das Paket fertig empfangen.

Im regulären Fall hingegen wird auf ein L0 Byte gewartet. Anschließend wird dieses interpretiert, falls dieses auf ein LF Feld hindeutet, welches N beinhaltet, wird dieses empfangen und interpretiert. Danach wird der Payload der Länge N erwartet und empfangen. Dabei wird jedes Byte einzeln empfangen und mittels *fout_pl*, *fout_pl_addr* und *fout_pl_wen* in den BRAM *frx_pl* geschrieben, woraus es durch *app_fsm* später wieder gelesen wird. Dazu wird ein Zähler für die Addresse *fout_pl_addr* eingesetzt, der von N runterzählt. Anschließend werden noch eventuelle LE0 bzw. LEF Bytes empfangen analog zu L0 und LF, falls LEX im Config-Byte gesetzt war. Diese beinhalten *M*. Dies gilt ebenfalls für die Prüfsumme XOR1, die nur empfangen wird, wenn XOR1X gesetzt war. Falls diese empfangen wird, wird die intern berechnete Prüfsumme über das Paket mit der Empfangenen verglichen und bei Ungleichheit ein Fehler weitergeleitet.

Nach Abschluss des Empfangs wird das Paket an *app_fsm* weitergereicht. Dies geschieht indem *fout_stb* gesetzt und so lange gehalten wird, bis es von *app_fsm* mittels Setzen von *fout_ack* bestätigt wurde. Anschließend wird neu initialisiert und es kann ein neues Paket empfangen werden.

5.7 Modul *uart_out_fsm*

Das Modul *uart_out_fsm* erfüllt eine ähnliche Aufgabe wie das Modul *uart_in_fsm*, allerdings in die andere Richtung. So werden ausgehende Pakete seitens *app_fsm* in einzelne Bytes zerlegt und an *uart_axi* gesendet. Die Funktionalität ist auch hier vereinfacht in Abbildung 12 dargestellt, da auch hier die Komplexität des Automaten einen vollständigen Automatengraphen unübersichtlich machen würde.

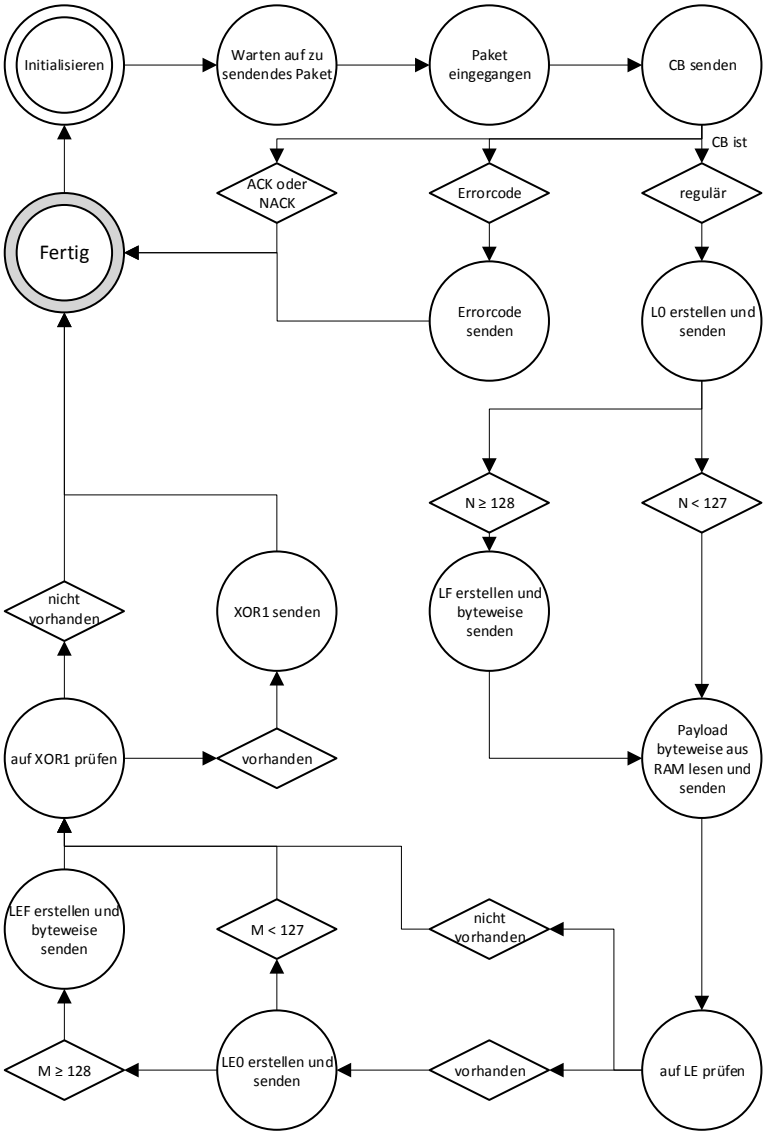

Abbildung 12: Vereinfachter Automatengraph des Moduls *uart_out_fsm*.

Das Eintreffen eines Paketes wird von *app_fsm* durch Setzen von *fin_stb* (*in*, da es in *uart_main* ein Eingang ist) signalisiert. Aus *fin_meta* gehen dabei das CB, der OP-Code, N und M hervor.

Eine Prüfsumme wird immer über alle gesendeten Bytes eines Paketes gebildet, jedoch wird diese nicht immer versendet.

Auch hier wird zwischen regulären und ACK bzw. NAK Paketen und Fehlercodes unterschieden. Im Falle von ACK und NAK ist das Paket abgeschlossen nach Senden des CB abgeschlossen, im Falle eines Fehlercodes wird noch der Fehlercode gesendet. Zum Senden wird das jeweilige Datum auf das Register *uart_in* gelegt, *uart_in_stb* gesetzt und gehalten, bis seitens *uart_axi* mit einem *uart_in_ack* quittiert wird. Da Statusnachrichten nicht von *app_fsm* versendet werden können, werden diese hier nicht berücksichtigt.

Nach dem CB wird bei regulären Paketen L0 erstellt und gesendet. Falls nötig, also insofern N größer oder gleich 128 ist, wird ein LF erstellt und gesendet. Ansonsten werden die Daten des Payloads einzeln mittels *fin_pl_r* und *fin_pl_raddr* aus dem BRAM *ftx_pl* gelesen und gesendet. Hierbei wird ein Zähler genutzt, der bei N beginnt und bis 0 runterzählt. Es muss, nachdem die Adresse *fin_pl_raddr* eingestellt worden, ist ein Takt gewartet werden, bis das Datum an *fin_pl_r* anliegt. Dies wird entsprechend berücksichtigt.

Falls notwendig, werden ein LE0 Byte, ein LEF Feld und ein XOR1 Byte noch erstellt und ebenfalls versendet, bevor das Paket abgeschlossen ist und ein neues nach erneuter Initialisierung versendet werden kann.

5.8 Modul *app_reset_sync*

Das Modul *app_reset_sync* ist notwendig, um den Reset vom Datentakt *data_clk* (250 MHz) auf den doppelten Takt *double_clk* (500 MHz) zu synchronisieren. Dazu wird ein mit *double_clk* getaktetes Register genutzt, um den eingehenden *data_nres* Reset zwischenzuspeichern und damit zu synchronisieren. Dies ist aber nur zulässig, solange das *locked* Signal, welches der Taktsignalgeber *clk_wiz_0* erzeugt, gesetzt ist, da dieses die Gültigkeit der *double_clk* anzeigt. Der zwischengespeicherte, synchrone Reset *double_nres* wird ausgegeben.

5.9 Modul *app_fsm*

Das Modul *app_fsm* ist die Schaltzentrale für die Konfiguration des SAD_CoProc. Es dient der Abwicklung der Kommunikationspakete seitens der übergeordneten AXI-Stelle. Es beinhaltet unter anderem auch den Speicher für das Vergleichsmuster, hier *samples* genannt. Die Funktionalität ist auch hier vereinfacht in Abbildung 13 dargestellt, da ein vollständiger Automatengraph zu unübersichtlich würde.

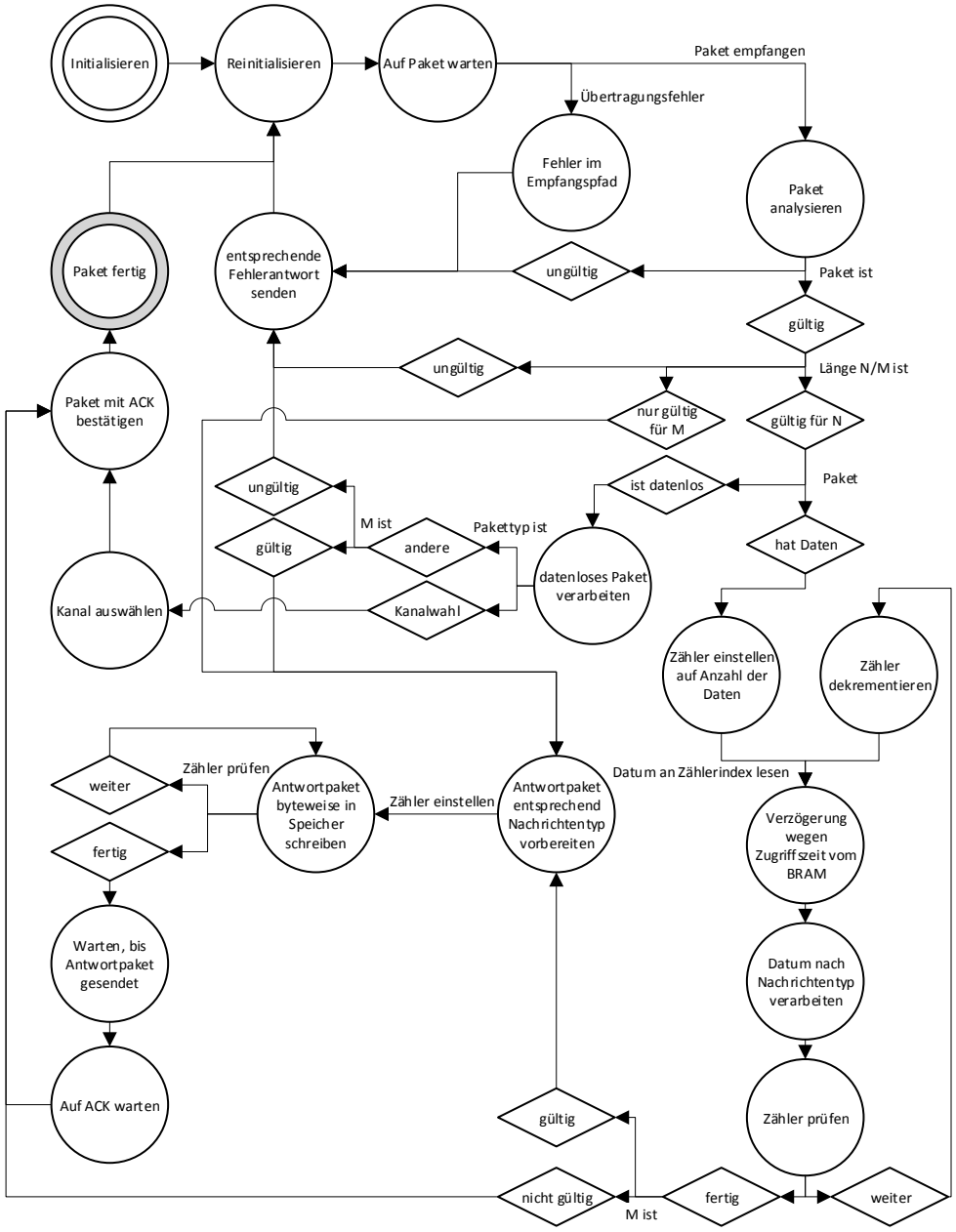

Abbildung 13: Vereinfachter Automatengraph des Moduls *app_fsm*.

Bei Reset werden zunächst alle Register initialisiert und Statussignale zurückgesetzt. Von hier ausgehend wird auf ein eintreffendes Paket gewartet, welches seitens *uart_in_fsm* durch Setzen von *frx_stb* signalisiert wird. Bei einem Übertragungsfehler wird dieser mit einer entsprechenden Fehlerantwort, bzw. einem zu sendenden Fehlercode, quittiert.

Ein eingetroffenes Paket wird zunächst analysiert. Dazu wird *frx_meta* analysiert, hieraus gehen die Metainformationen N, M, der OP-Code und das CB hervor. Falls es direkt ungültig ist, weil beispielsweise der OP-Code fehlerhaft ist, wird ebenfalls eine entsprechende Fehlerantwort gesendet. Ein gültiges Paket hingegen wird auf die Länge $N - 2$ geprüft, dabei werden die zwei bereits für den OP Code benötigten Byte abgezogen. Ist diese Länge gültig, wird auf ein datenbehaftetes oder datenloses Paket geprüft. Ist diese ungültig, wird M geprüft. Ist M gültig, wird ein Antwortpaket erstellt und gesendet, ist auch M ungültig, so ist das eingetroffene Paket ungültig und es wird eine Fehlerantwort gesendet.

Ein datenloses Paket kann nur einen Kanalwechsel OP-Code beinhalten oder muss ein gültiges M aufweisen, damit es gültig ist. Ansonsten ist es ungültig und wird mit entsprechender Fehlerantwort quittiert. Im Falle eines gültigen M wird ein Antwortpaket gesendet.

Ein datenbehaftetes Paket wird byteweise aus dem BRAM *frx_pl* gelesen und entsprechend dem Nachrichtentyp gespeichert. So werden Vergleichsmuster in *samples* gespeichert, und ein Schwellwert in *threshold*. Nach vollständiger Bearbeitung wird anschließend auf ein gültiges M geprüft: Bei ungültigem M wird das Paket mit einem ACK quittiert; auf ein gültiges M folgt ein Antwortpaket.

Für ein Antwortpaket werden zunächst die Metainformationen in *ftx_meta* erstellt und dann der Payload byteweise in den BRAM ftx_pl geschrieben. Nach Abschluss des Schreibvorgangs wird das Paket an *uart_out_fsm* mittels setzen von *ftx_stb* weitergereicht. Ein Antwortpaket erfordert eine Quittierung seitens der AXI-Stelle mittels eines ACK. Es wird daher auf einkommendes ACK gewartet.

Im Anschluss auf ein gültiges Paket wird ein ACK versendet. Bei vorhandenem Antwortpaket wird auf ein eintreffendes ACK ebenfalls mit einem ACK geantwortet. Anschließend wird reinitialisiert, dabei werden die Register, die zum Beispiel das Vergleichsmuster, die aktuelle Samplelänge und den Schwellwert speichern, nicht zurückgesetzt.

Kapitel 6 - Funktionale Simulationen

Um die Funktionalität der in den vorangegangenen Kapiteln vorgestellten Modulen zu gewährleisten, werden für die Bereiche des Rechenwerks und die der Kommunikation im Gesamtsystem Simulationen durchgeführt. Für diese Simulationen wird ModelSim [14] von Mentor verwendet. Dazu sind zwei Testbenches erstellt worden, eine für das SAD Rechenwerk (*sad_main_tb*) und eine weitere (*app_main_tb*), die das Gesamtsystem und dessen Grundfunktionen testet. Die Validierung erfolgt dabei händisch durch Auswertung der Signalverläufe.

6.1 Simulation Rechenwerk, *sad_main_tb*

6.1.1 Vorstellung der Testbench

Das Rechenwerk wird mittels *sad_main_tb* getestet. Die dafür verwendete Testbench ist im nachfolgenden Listing aufgeführt.

Listing 2: sad_main_tb.vhd

```
library work;
    use work.csa_type_pack.all;
    use work.tb_pack.all;

entity sad_main_tb is
end entity;

architecture rtl_sim of sad_main_tb is

    constant DATA_CLK_SIM_IT : natural := 50000;
    signal DATA_CLK_SIM_IT_LEFT : natural := DATA_CLK_SIM_IT;


    signal data_clk: std_logic;
    signal double_clk: std_logic;
    signal double_nres: std_logic;
    signal new_data_I: std_logic_vector(63 downto 0);
    signal new_data_Q: std_logic_vector(63 downto 0);
    signal new_data_select: std_logic;
    signal samples: ArrN00(1023 downto 0);
    signal threshold: unsigned(bitwidth_main + 9 downto 0);
    signal trigger: std_logic;
    signal mode: std_logic_vector(1 downto 0);


    constant in_test : natural := 2024;
    type test_arr is array(in_test-1 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    signal test_data_i : test_Arr;
    subtype index_type is natural range in_test-1 downto 0;
    signal test_data_i_index : natural := in_test-1;
begin
```

Fortsetzung(2/3) Listing 2: sad_main_tb.vhd

```
sad_main_inst: entity work.sad_main
    port map (
        data_clk        => data_clk,
        double_clk      => double_clk,
        double_nres     => double_nres,
        new_data_I      => new_data_I,
        new_data_Q      => new_data_Q,
        new_data_select => new_data_select,
        samples         => samples,
        threshold       => threshold,
        trigger         => trigger,
        mode            => mode
    );

stimuli_p: process is
begin
    new_data_Q <= x"0001_0001_0001_0001";
    new_data_select <= '0';
    --for static tests with Q channel:
    --for i in samples'range loop
    --  samples(i) <= "10000000000001";
    --end loop;
    threshold <= x"000000";
    mode <= "00";
    wait;
end process;

data_feed_p: process is
begin
    if (double_nres = '1') then
        new_data_i (63 downto 48) <=
            test_data_i(test_data_i_index-3)(15 downto 0);--newest
        new_data_i (47 downto 32) <=
            test_data_i(test_data_i_index-2)(15 downto 0);
        new_data_i (31 downto 16) <=
            test_data_i(test_data_i_index-1)(15 downto 0);
        new_data_i (15 downto  0) <=
            test_data_i(test_data_i_index-0)(15 downto 0);--oldest
        if test_data_i_index = 3 then
            test_data_i_index <= in_test-1;
        else
            test_data_i_index <= test_data_i_index - 4;
        end if;
        if (DATA_CLK_SIM_IT_LEFT= 0) then
            wait;
        end if;
    end if;
    wait for DATA_CLK_PERIOD;
end process;

data_init_p: process is
begin
    for i in test_data_i'range loop
        test_data_i(i) <= std_logic_vector(to_unsigned(i,16));
    end loop;
    for i in 1023 downto 0 loop
        ---samples need offset because data is signed and also offset
        --eg: 1000(-8)-> 0000(0), 0000->1000 (0->8), 0111->1111 (7->15)
        samples(i) <= std_logic_vector(to_unsigned(8192 + 1023 - i,14));
    end loop;
    wait until double_nres <= '1';
    wait;
end process;
```

Fortsetzung(3/3) Listing 2: sad_main_tb.vhd

```
    data_clock_p: process is
    begin
        data_clk <= '0';
        wait for DATA_CLK_PERIOD / 2;
        data_clk <= '1';
        wait for DATA_CLK_PERIOD / 2;
        if DATA_CLK_SIM_IT_LEFT = 0 then
            wait;
        else
            DATA_CLK_SIM_IT_LEFT <= DATA_CLK_SIM_IT_LEFT - 1;
        end if;

    end process;
    dbl_clock_p: process is
    begin
        double_clk <= '1';
        wait for DBL_CLK_PERIOD / 2;
        double_clk <= '0';
        wait for DBL_CLK_PERIOD / 2;
    end process;

    reset_p: process is
    begin
        double_nres <= '0';
        wait for RST_HOLD_DURATION;
        wait until rising_edge(data_clk);
        double_nres <= '1';
        wait;
    end process;

end architecture;
```

Ende Listing 2: sad_main_tb.vhd

Innerhalb der Testbench wird die Hauptkomponente *sad_main* instanziiert (*sad_main_inst*) und mit Daten versorgt. Die Prozesse *data_clock_p* und *dbl_clock_p* sind hierbei für die Taktgenerierung mittels *data_clk* bzw. *double_clk* zuständig. Ferner ist der Prozess *reset_p* für den low-aktiven Reset *double_nres* zuständig. Dieser wird zu Beginn aktiviert, für mindestens eine Taktperiode gehalten und zu einer steigenden Taktflanke des Datentaktes wieder aufgehoben.

Statische Dateneingänge werden in *stimuli_p* definiert. Das eingespielte Vergleichsmuster, eine fallende Zahlenfolge, beginnend bei 1023 bis 0, und die Testdaten, ebenfalls eine fallende Zahlenfolge von 2023 bis 0, werden im Prozess *data_init_p* definiert. Die am Kanal I anliegenden Daten werden im Prozess *data_feed_p* in jedem Datentakt gewechselt bzw. es wird die zugrundeliegende Zahlenfolge fortgesetzt. Bei Gleichheit von Vergleichsmuster und Eingang soll der Trigger-Ausgang *trigger* gesetzt werden, *threshold* wird dazu auf 0 eingestellt.

Die Anzahl der maximalen Taktzyklen ist innerhalb der Testbench durch DATA_CLK_SIM_IT begrenzt, damit sich das Simulationsprogramm nicht aufhängt.

6.1.2 Ergebnisse der Simulation

Das Wave-Ausgabefenster von ModelSim für *sad_main_tb* ist nachfolgend in Abbildung 14 dargestellt.

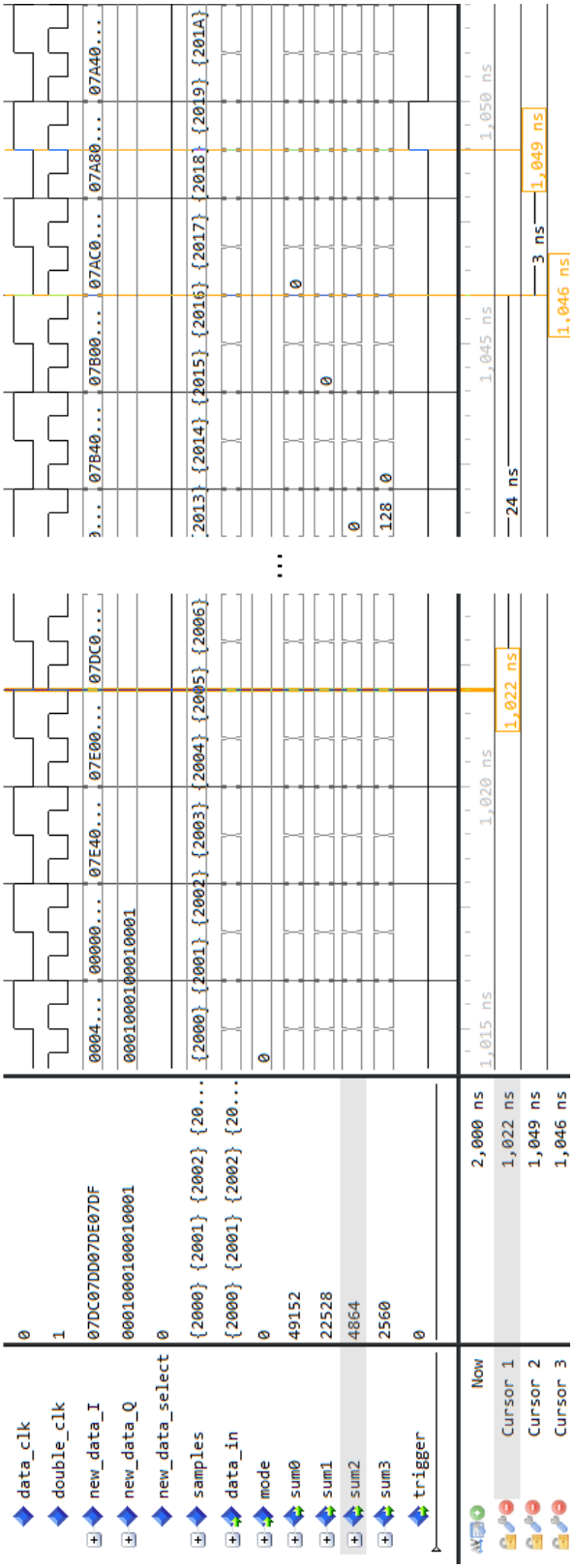

Abbildung 14: Ergebnis der Simulation von *sad_main_tb*.
Dargestellt sind alle wesentlichen Ein- und Ausgänge. Die Cursor zeigen die wichtigsten Zeitpunkte.

Wie zu erkennen ist, wird die Summe nach Durchlaufen aller Addierbaumstufen zu null (*Cursor 3, sum0*) wenn die Eingangsfolge gleich der Vergleichsfolge ist (*Cursor 1, data_in* und *samples*). Bei einem wie hier eingestelltem Schwellwert *threshold* von 0 wird dann der Trigger ausgelöst (*Cursor 2*, *trigger*). Die Funktionalität der getesteten Komponente ist in der Simulation gegeben, denn die SAD aus beiden Folgen wird jederzeit korrekt berechnet, die Eingangsdaten werden korrekt einsortiert und es wird korrekterweise ein Trigger-Signal bei Unterschreiten des Schwellwertes *threshold* ausgelöst.

6.2 Simulation Gesamtsystem, *app_main_tb*

6.2.1 Vorstellung der Testbench

Das Gesamtsystem des SAD_CoProc wird mittels app_*main_tb* getestet. Diese Testbench ist im nachfolgenden Listing aufgeführt.

Listing 3: app_main_tb.vhd

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.NUMERIC_STD.all;

library std;
    use std.textio.all;

    use work.tb_pack.all;
    use work.uart_pack.all;

entity app_main_tb is
end entity;

architecture rtl_sim of app_main_tb is

    constant DATA_CLK_SIM_IT : natural := 100000;
    signal DATA_CLK_SIM_IT_LEFT : natural := DATA_CLK_SIM_IT;
    constant AXI_CLK_SIM_IT : natural := 40000;
    signal AXI_CLK_SIM_IT_LEFT : natural := AXI_CLK_SIM_IT;


    signal data_clk: STD_LOGIC;
    signal data_nres: std_logic;
    signal axi_clk: STD_LOGIC;
    signal axi_nres: std_logic;
    signal axi_out_conf: std_logic_vector(7 downto 0);
    signal axi_rx: std_logic_vector(7 downto 0) := (others => '0');
    signal axi_in_conf: std_logic_vector(7 downto 0) := (others => '0');
    signal conf_i, conf_o:axi_config;
```

Fortsetzung(2/4) Listing 3: app_main_tb.vhd

```
    signal conf_i_stb_i , conf_i_ack_i : std_logic;
    signal axi_tx: std_logic_vector(7 downto 0);
    signal new_data_I: std_logic_vector(63 downto 0) := (others => '0');
    --signal new_data_Q: std_logic_vector(63 downto 0);
    signal trigger: std_logic;

    constant bitwidth_main : integer := 14;
    type ArrN02 is array(natural range<>) of std_logic_vector(bitwidth_main + 1 downto 0);
    signal in_samples: ArrN02(1023 downto 0);

    signal axi_done : boolean := false;

    constant in_test : natural := 2024;
    type test_arr is array(in_test-1 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    signal test_data_i : test_Arr;
    subtype index_type is natural range in_test-1 downto 0;
    signal test_data_i_index : natural := in_test-1;

    --X"0000000000000000";
    constant new_data_Q : std_logic_vector(63 downto 0) :=(others => '0');
    constant q_const_top : std_logic_vector(7 downto 0) := x"20";
    constant q_const_bot : std_logic_vector(7 downto 0) := x"00";

begin

    app_main_inst: entity work.app_main
        port map (
            data_clk        => data_clk,
            data_nres       => data_nres,
            axi_clk         => axi_clk,
            axi_nres        => axi_nres,
            axi_out_conf    => axi_out_conf,
            axi_rx          => axi_rx,
            axi_in_conf     => axi_in_conf,
            axi_tx          => axi_tx,
            new_data_I      => new_data_I,
            new_data_Q      => new_data_Q,
            trigger         => trigger
        );

    conf_b: block
    begin
    conf_i <= (
        reserved0   => '0',
        ena         => '0',
        res         => '0',
        ack_i       => conf_i_ack_i,
        ack_o       => '0',
        stb_i       => conf_i_stb_i,
        stb_o       => '0',
        reserved1   => '0');
    axi_in_conf <= from_axi_config(conf_i);
    conf_o <= to_axi_config(axi_out_conf);
    end block conf_b;

    axi_feep_p: process is
        procedure axi_inp_tb (
                input : std_logic_vector(7 downto 0);
                wait_cycles : natural
            ) is
        begin
            axi_rx <= input;
            conf_i_stb_i <= '1';
            wait until conf_o.ack_o <= '1';
            wait for AXI_CLK_PERIOD;
            conf_i_stb_i <= '0';
            axi_rx <= (others => '0');
            wait until conf_o.ack_o <= '0';
            wait for wait_cycles * AXI_CLK_PERIOD;
        end procedure axi_inp_tb;
```

Fortsetzung(3/4) Listing 3: app_main_tb.vhd

```
begin ------------------------------------------
    axi_rx <= (others => '0');
    conf_i_stb_i <= '0';
    wait until axi_nres = '1';
    wait until AXI_CLK_SIM_IT-AXI_CLK_SIM_IT_LEFT = 20; --reset is cleared

    for i in ch_change_q'range loop --set to q
        axi_inp_tb(ch_change_I(i),random);
    end loop;
    wait for 10 * AXI_CLK_PERIOD;

    for i in write_samples_64'range loop --set to i
        axi_inp_tb(write_samples_64(i),random);
    end loop;
    for i in 0 to 63 loop
        --axi_inp_tb(q_const_top,random);
        --axi_inp_tb(q_const_bot,random);
        axi_inp_tb(in_samples(1023-i)(15 downto 8),random);
        axi_inp_tb(in_samples(1023-i)( 7 downto 0),random);
    end loop;
    wait for 20 * AXI_CLK_PERIOD;

    --axi_done <= true;
    wait;-----------------------------------------
end process axi_feep_p;

axi_reply_p:process is
begin
    conf_i_ack_i <= '0';
    loop
        --positve handshake
        wait until conf_o.stb_o = '1' or AXI_CLK_SIM_IT_LEFT = 0;
        if AXI_CLK_SIM_IT_LEFT = 0 then
            wait;
        end if;
        wait for AXI_CLK_PERIOD;
        conf_i_ack_i <= '1';
        --negative handshake
        wait until conf_o.stb_o = '0' or AXI_CLK_SIM_IT_LEFT = 0;
        if AXI_CLK_SIM_IT_LEFT = 0 then
            wait;
        end if;
        wait for AXI_CLK_PERIOD;
        conf_i_ack_i <= '0';
    end loop;
end process axi_reply_p;

data_feed_p: process is
begin
    if (data_nres = '1') then
        new_data_i (63 downto 48) <=
            test_data_i(test_data_i_index-3)(15 downto 0);--newest
        new_data_i (47 downto 32) <=
            test_data_i(test_data_i_index-2)(15 downto 0);
        new_data_i (31 downto 16) <=
            test_data_i(test_data_i_index-1)(15 downto 0);
        new_data_i (15 downto  0) <=
            test_data_i(test_data_i_index-0)(15 downto 0);--oldest
        if test_data_i_index = 3 then
            test_data_i_index <= in_test-1;
        else
            test_data_i_index <= test_data_i_index - 4;
        end if;
        if (DATA_CLK_SIM_IT_LEFT= 0) then
            wait;
        end if;
    end if;
    wait for DATA_CLK_PERIOD;
end process;
```

Fortsetzung(4/4) Listing 3: app_main_tb.vhd

```
    data_init_p: process is
    begin
        for i in test_data_i'range loop
            test_data_i(i) <= std_logic_vector(to_unsigned(i,16));
        end loop;
        for i in 1023 downto 0 loop
            ---samples need offset because data is signed and also offset
            --eg: 1000(-8)-> 0000(0), 0000->1000 (0->8), 0111->1111 (7->15)
            in_samples(i) <= std_logic_vector(to_unsigned(8192 + 1023 - i,16));
        end loop;
        wait until data_nres <= '1';
        wait;
    end process;

    data_clock_p: process is
    begin
        data_clk <= '0';
        wait for DATA_CLK_PERIOD / 2;
        data_clk <= '1';
        wait for DATA_CLK_PERIOD / 2;
        if DATA_CLK_SIM_IT_LEFT = 0 then
            wait;
        else
            DATA_CLK_SIM_IT_LEFT <= DATA_CLK_SIM_IT_LEFT - 1;
        end if;

    end process;

    axi_clock_p: process is
    begin

        axi_clk <= '0';
        wait for AXI_CLK_PERIOD / 2;
        axi_clk <= '1';
        wait for AXI_CLK_PERIOD / 2;
        if AXI_CLK_SIM_IT_LEFT = 0 or axi_done then
            wait;
        else
            AXI_CLK_SIM_IT_LEFT <= AXI_CLK_SIM_IT_LEFT - 1;
        end if;
    end process;

    reset_p: process is
    begin
        data_nres <= '0';
        axi_nres <= '0';
        wait for RST_HOLD_DURATION;
        wait until rising_edge(data_clk);
        data_nres <= '1';
        axi_nres <= '1';
        wait;
    end process;

end architecture;
```

Ende Listing 3: app_main_tb.vhd

Wie auch in 6.1, wird die zu testende Komponente, *app_main*, instanziiert (*app_main_inst*). Ferner sind die Prozesse *data_clock_p*, *data_feed_p* und *data_init_p* in ihrer Funktion identisch. Der Prozess *reset_p* ist zusätzlich noch für den weiteren Reset *axi_nres* zuständig, dieser ist mit dem *data_nres* allerdings vom Verhalten her identisch.

Zusätzlich sind die Prozesse *axi_feed_p* und *axi_reply_p,* diese Simulieren die übergeordnete AXI-Stelle. In axi_feed_p werden Konfigurationspakete an den SAD_CoProc geleitet, in axi_reply_p wird die Antwort der AXI-Stelle auf eingehende Nachrichten simuliert. Es wird der Kanal I ausgewählt und anschließend ein 64 Sample langes Vergleichsmuster übertragen. Die Konfiguration sollte *mode* auf 11_2 umschalten und das Vergleichsmuster sollte in *samples* gespeichert werden.

6.2.2 Ergebnisse der Simulation

Das Wave-Ausgabefenster von ModelSim für *app_main_tb* ist nachfolgend in Abbildung 15 dargestellt.

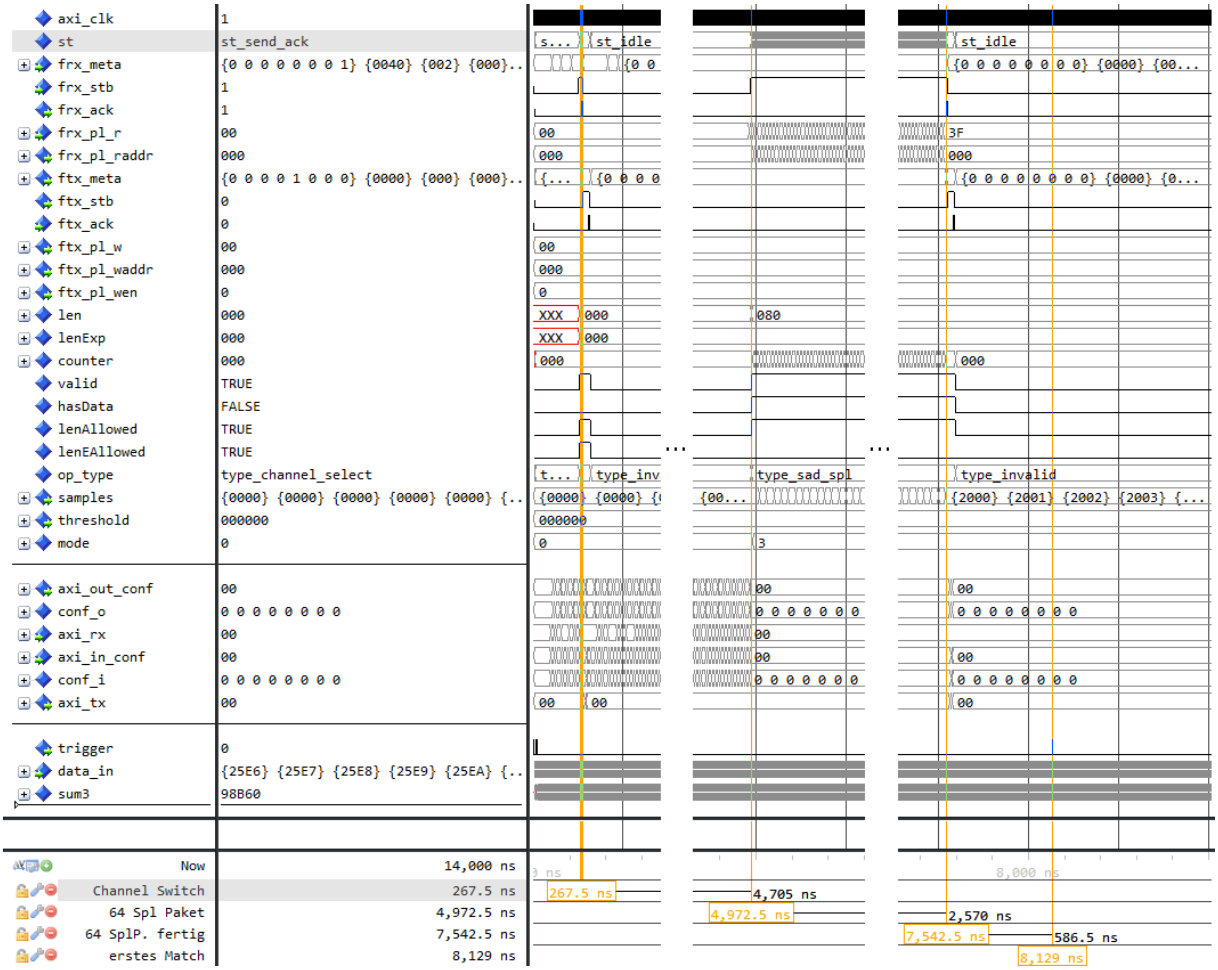

Abbildung 15: Ergebnisse der Simulation von *app_main_tb*.
Dargestellt sind alle wesentlichen Ein- und Ausgänge. Die vier Cursor zeigen die wesentlichen Zeitpunkte.

Es wird zunächst eine Kanalwechselnachricht übertragen, die korrekt empfangen und bestätigt wird (Cursor C*hannel Switch*). Der OP-Code wird korrekt erkannt (*op_type, valid, lenAllowed*). Die Bestätigung mittels ACK (*st* wird zu `st_send_ack`) kann auch in *ftx_meta* mit gesetztem ACK Bit und *ftx_stb* erkannt werden.Anschließend beginnt die Übertragung des 64 Samples langen Vergleichsmusters. Dies lässt sich in der regen Aktivität der Signale *axi_out_conf*, *axi_in_conf* und *axi_rx* im Bereich zwischen erstem und zweitem Cursor erkennen. Das Modul uart_in_fsm schreibt hierbei die eintreffenden Vergleichsmusterwerte in den BRAM *frx_pl*. Beim zweiten Cursor (*64 Spl Paket*) ist diese Übertragung abgeschlossen. Es wird seitens des Moduls *uart_in_fsm* an *app_fsm* mitgeteilt, dass ein Paket angekommen ist, indem *frx_stb* gesetzt wird. Nun werden die Vergleichsmusterwerte des Paketes byteweise aus dem BRAM *frx_pl* gelesen und in *samples* gespeichert. Dies ist ferner zu daran zu erkennen, dass sich *st, frx_pl_r, frx_pl_raddr* und der Zähler *counter* stark ändern.Am dritten Cursor (*64 SplP. fertig*) ist dieser Vorgang abgeschlossen. Es kann erneut ein gesendetes ACK-Paket erkannt werden. Ferner ist *samples* mit den korrekten Werten belegt. Der vierte Cursor zeigt einen Ausschlag des Trigger-Ausgangs. Dieser ist gesetzt, da Vergleichsmuster- und Eingangsdaten zu einem Zeitpunkt kurz zuvor gleich gewesen sind und *threshold* nach Anfangs erfolgtem Reset einen Wert von 0 besitzt.

Anhand dieser Beobachtungen lässt sich die Funktion des Hauptmoduls bestätigen. Es können verschiedene Konfigurationspakete empfangen werden, die korrekt bearbeitet und beantwortet werden. Ferner funktioniert auch das miteingebundene Rechenwerk weiterhin korrekt.

Kapitel 7 - Hardware-Testsystem

7.1 Gesamtübersicht Testsystem

Um den entwickelten SAD_CoProc IP Core in Hardware zu testen, ist ein System on Chip (SoC) basiertes Testsystem entwickelt worden. Das Testsystem wird mit Hilfe des anvisierten KC705 Boards realisiert. Dieses wird mittels USB-UART per USB an einen Windows PC angeschlossen, auf dem eine Konfigurationssoftware läuft. Für möglichst einfache Entwicklung und Verwendung werden möglichst alle Benutzerschnittstellen des KC705 verwendet bzw. deren Verwendung vorbereitet.

Nachfolgend werden die SoC Hardware, die für das SoC System notwendige Firmware und die Konfigurationssoftware auf dem PC beschrieben, eine Übersicht ist in Abbildung 16 dargestellt. Das zur Aufzeichnung des Trigger-Ausgangs verwendete Oszilloskop, ein PicoScope [15], ist in dieser Übersicht ebenfalls dargestellt, wird aber nicht weiter erläutert.

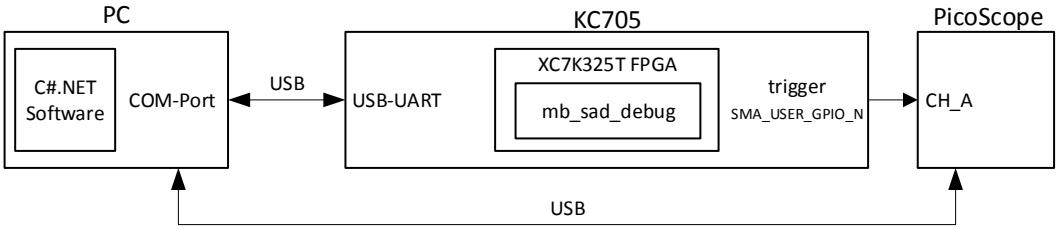

Abbildung 16: Übersicht über das Testsystem.

7.2 FPGA Blockschaltbild *mb_sad_debug*

Das beschriebene SoC beinhaltet einen MicroBlaze Softcore-Prozessor [12] (MB, *microblaze_0*) mit angeschlossener Peripherie. Es wird *mb_sad_debug* genannt. Das Testsystem wird hierbei mit Hilfe des in Vivado befindlichen IP Integrators, den dort vorhandenen Assistenten und den von Xilinx bereitgestellten IP Cores zusammengeschaltet. Das Blockschaltbild ist nachfolgend in Abbildung 17 dargestellt.

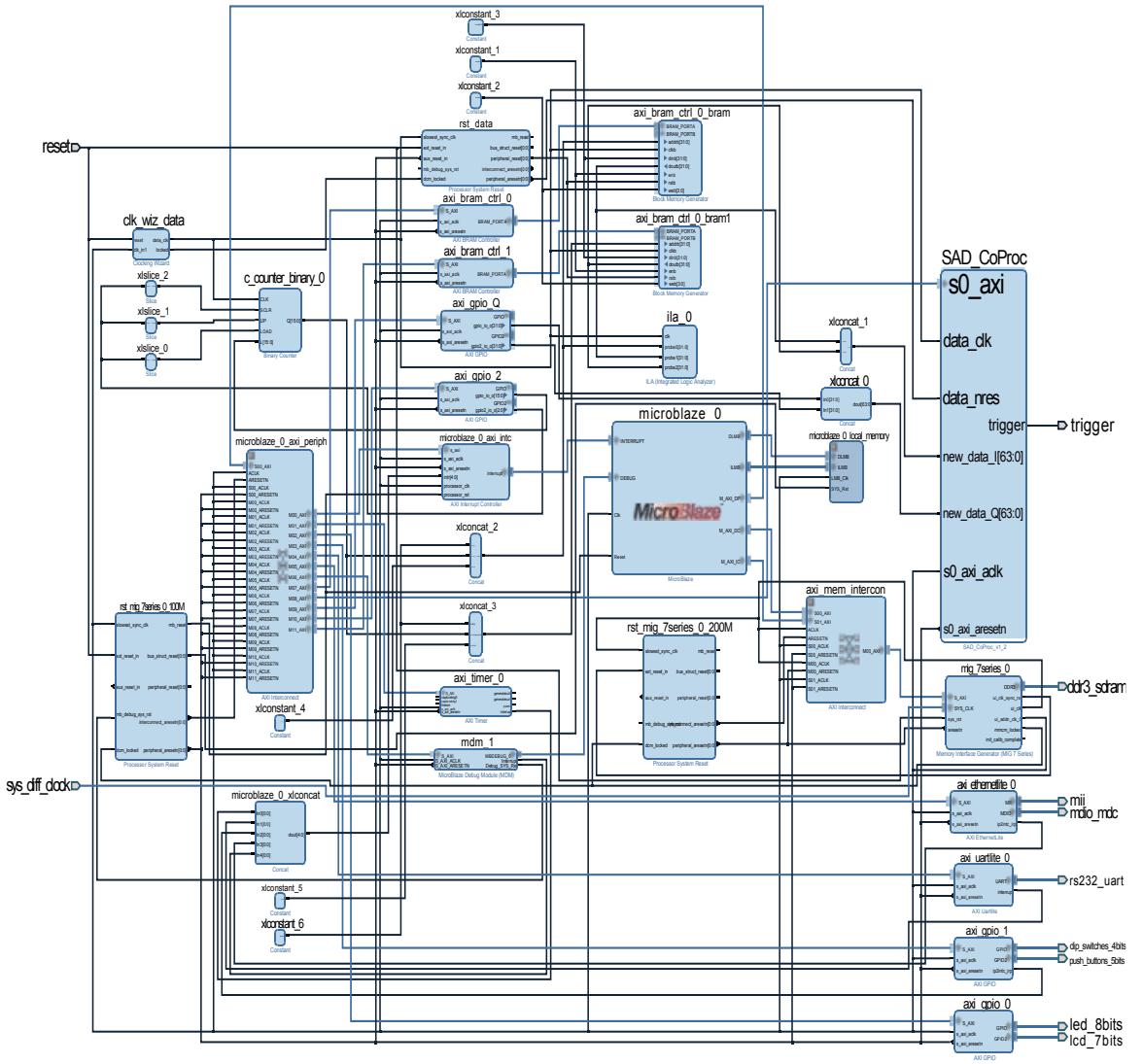

Abbildung 17: Blockschaltbild Testsystem *mb_sad_debug*.

Die Peripherie beinhaltet mehrere über AXI verbundene Komponenten, darunter der zu testende *SAD_CoProc*. Unter anderem werden die Taster, Schalter, LEDs und das Display des KC705 als Benutzerschnittstellen verbunden (*axi_gpio_0*, *axi_gpio_1*). Ferner wird auch der auf dem KC705 vorhandene DDR3 Speicher über AXI angeschlossen (*mig_7series_0*). Als UART wird der als COM Port verfügbare USB-UART Anschluss des KC705 genutzt (*axi_uartlite_0*). Auch eine Verbindung mittels TCP/IP und den Ethernet-Anschluss ist vorbereitet (*axi_ethernetlite_0*). Um später ein Betriebssystem verwenden zu können, wird zusätzlich noch je ein AXI Timer (*axi_timer_0*) und ein Interrupt-Controller (*microblaze_0_axi_intc*) eingebunden.

Der MB verwendet eine Taktfrequenz von 100 MHz, die durch den KC705 internen Oszillator erzeugt wird (*sys_diff_clock*). Diese Frequenz wird auch für den AXI-Bus (*XX_axi_aclk*) verwendet. Es wird daher (*clk_wiz_data*) noch ein zusätzlicher 250 MHz Takt für die Signaldaten

erzeugt. Weiter wird der System-Reset (*reset*) noch für jede Taktfrequenz synchronisiert (*rst_mig_7series_0_100M*, *rst_mig_7series_0_200M*, *rst_data*).

Für Kanal Q werden zwei 32 Bit AXI GPIO Register (*axi_gpio_Q*) für statische Eingangswerte verwendet. Für Kanal I, über welchen selbstgewählte Datenfolgen getestet werden sollen, werden zwei 64 KiB große, 32 Bit breite BRAM-Blöcke über AXI verbunden (*axi_bram_ctrl_0_bram*, und *axi_bram_ctrl_0_bram1*). Diese sind jeweils als Zweitor BRAM ausgeführt, wobei das B Tor mit dem Datentakt von 250 MHz betrieben wird und das A Tor über AXI (*axi_bram_ctrl_0*, *axi_bram_ctrl_1*). Als Adresse dient hierbei ein Zähler (*c_counter_binary_0*), welcher ebenfalls mit dem Datentakt läuft. Der Zähler wird mittels AXI_GPIO kontrolliert (*axi_gpio_2*). Zur Überwachung der korrekten Signalabfolge ist ein ILA (*ila_0*) eingebettet. Dieser überwacht die Adressierung und den Ausgang beider BRAMs, die für Kanal I eingesetzt werden.

Ferner sind noch Steuerblöcke (*xlslice_X, xlconcat_X*) sowie Konstanten (*constant_X*) vorhanden, auf diese wird jedoch der Übersicht halber nicht weiter eingegangen. Es gilt für die Adressierung der 64 Bit breiten Eingänge des *SAD_CoProc* allgemein, dass die höherwertigen Eingänge jeweils mit den namentlich höherwertigen Ausgängen verbunden werden.

7.3 SoC Firmware

Die für den Betrieb des Testsystems notwendige SoC Firmware (*sad_debug_app*) läuft auf dem in *mb_sad_debug* verwendeten MB und hat Zugriff auf alle über AXI angebundenen Elemente: Zur Entwicklung der Software ist Xilinx SDK verwendet worden. Dieses erlaubt die Integration des Betriebssystems freeRTOS [16] ohne weitere eigene Konfiguration. Als Sprache ist C++ gewählt worden, um objektorientierte Programmierung einsetzen zu können. Nachfolgend wird in Abbildung 18 die Klassenübersicht innerhalb der Firmware dargestellt.

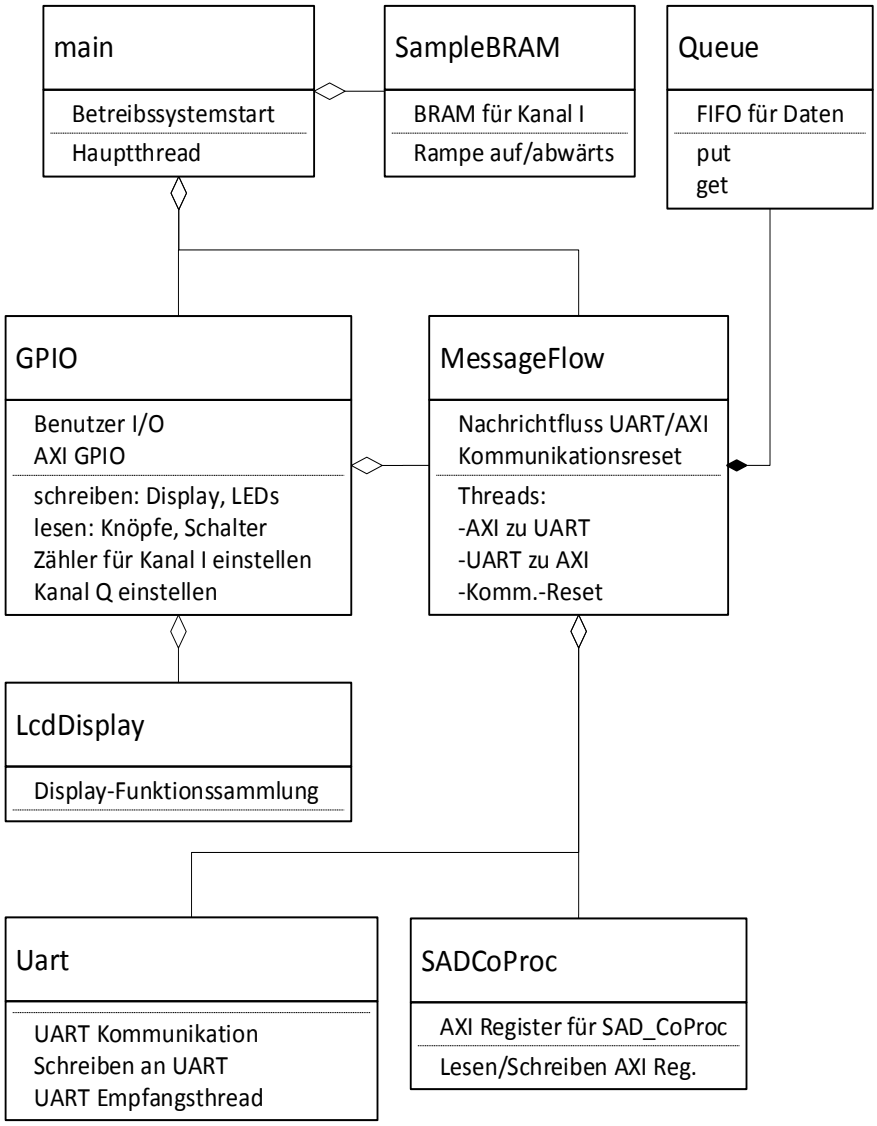

Abbildung 18: UML-Klassendiagramm der SoC Firmware *sad_debug_app*.

Zunächst startet die Hauptmethode *main* das Betriebssystem und den Hauptthread. Dann werden die Module *SampleBRAM*, *GPIO* und *MessageFlow* vom Hauptthread erzeugt. Hiernach ist die Firmware vollständig lauffähig.

SampleBRAM dient dem Befüllen der beiden BRAMs für Kanal I. GPIO dient der Darstellung des Displays, wofür mit *LcdDisplay* eine eigene Funktionssammlung vorhanden ist, und der LEDs, sowie der Annahme der Eingabe der Knöpfe und Schalter des KC705. Ferner können ein statischer Wert für Kanal Q eingestellt und der Zähler, der die Adressen des BRAMs für Kanal I hoch-/runterzählt, konfiguriert werden. *MessageFlow* verwaltet den Fluss von Nachrichtenbytes des PCs an den *SAD_CoProc* und umgekehrt. Der PC ist dabei über *UART* verbunden, der *SAD_CoProc* über *SADCoProc*.

MessageFlow verfügt über drei Threads. Der Thread AXI zu UART leitet dabei vom SAD_CoProc ausgehende Bytes weiter an den PC, der Thread UART zu AXI empfängt Bytes in der anderen Richtung und speichert diese in einer Queue zwischen, damit diese blockweise schnell empfangen und nachgehend einzeln vom *SAD_CoProc* verarbeitet werden können. Der Thread Komm.-Reset überwacht die mittlere Taste des KC705 und verwendet diese als Reset für die Kommunikationsmodule im *SAD_CoProc*. Das heißt, durch Betätigen der Taste wird innerhalb von *SAD_CoProc* der *cpu_axi_res* ausgelöst und alle an der Kommunikation beteiligten Automaten zurückgesetzt.

7.4 PC basierte Konfigurationssoftware

Die Konfigurationssoftware dient dem Versenden von Konfigurationspaketen im GTP Protokoll. Diese werden von der Software erstellt und über USB-UART an das KC705 gesendet. Die Software ist mit Hilfe von Visual Studio [17] und der Sprache bzw. dem Framework C#.NET entwickelt worden. Diese bietet leichteEntwicklung und Erweiterbarkeit gerade in Bezug auf mögliche spätere Nutzung grafischer Benutzeroberflächen.

Die Software besteht hierbei aus einer Klasse *Uart*, die die Verbindung zum COM Port aufbaut, der vom KC705 belegt wird. Ferner ist die Klasse in der Lage, die verschiedenen Nachrichtentypen nach Kapitel 4 zu versenden. Die verschiedenen Nachrichtentypen werden dabei mittels den Methoden *createPacket* erstellt und mittels *send* versendet. Ein zu übertragenes Mustersignal wird hierbei automatisch mit dem notwendigen Offset bedacht.

Zur Ausführung der Testfunktionen muss die Klasse lediglich von einem Programm instanziiert werden, da die Methodenaufrufe dafür vorerst im Konstruktor der Klasse sind. Dies ließe sich jedoch auch leicht in eine aufrufbare Testmethode auslagern, die für Testframeworks genutzt werden kann.

Kapitel 8 - Hardwaretests

Die Funktionalität des Testsystems ist mit verschiedenen Konfigurationspaketen und Eingabewerten getestet worden. Die Ergebnisse, welche mit Hilfe der in das Design eingebetteten ILA und dem an den Trigger-Ausgang angeschlossenen PicoScope (vgl. Abbildung 16, S.42) erzeugt worden sind, werden nachfolgend dargestellt.

8.1 Test der Konfiguration mit 64 Samples

Der erste Test ist, ob die Konfiguration über AXI und den angeschlossenen PC funktioniert. Dazu wird ein 64 Zeichen langes Vergleichspaket versendet. Dieses stellt eine von 63 bis 0 fallende Rampe dar. Die 0 ist dabei das aktuellste Sample. Dieses sollte mit dem notwendigen Offset in das höchstwertigste Sampleregister gespeichert werden. Die Ausgabe des ILA, welcher in *app_fsm* eingebettet ist, ist nachfolgend in Abbildung 19 dargestellt.

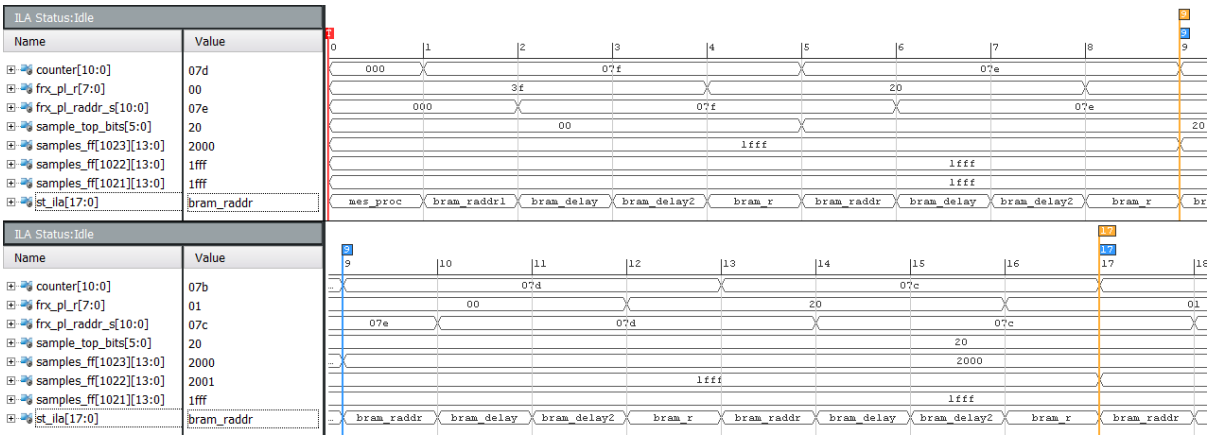

Abbildung 19: Ergebnisse Hardwaretest, 64 Sample Paket, ILA Ausgabefenster. Der Trigger für den ILA ist darauf eingestellt, dass aufgezeichnet wird, sobald *st_ila* den Wert *mes_proc* annimmt. Beim ersten Cursor, Zeitpunkt 9, ist zu erkennen wie das erste Sample übernommen wird, beim zweiten, Zeitpunkt 17, das zweite.

Der Trigger, welcher darauf eingestellt ist, die Aufzeichnung zu beginnen, sobald ein Paket in *app_fsm* anliegt (*st* bzw. *st_ila* wird zu *mes_proc*) wird ausgelöst und das Paket wird analysiert. Nach zwei BRAM Lesezyklen, die jeweils 4 Takte andauern, wird in Takt 9 der höchstwertigste Wert in das Sampleregister übernommen. In Takt 17 folgt dann das nächste Sample. Zusätzlich erfolgt nach senden der Nachricht in der PC Software der Empfang und die Ausgabe eines ACK.

Die Samples werden in der richtigen Reihenfolge in die Samplespeicher übernommen. Dies belegt, neben dergesendeten Antwort, welche von der PC basierten Konfigurationssoftware empfangen wird, die Funktion der Konfigurationsautomaten.

8.2 Test des Rechenwerkes mit statischen Eingang

Ein weiterer Test ist die Funktion des Rechenwerkes. Dieses muss bei konstanten Eingängen jeweils richtige Ergebnis liefern. Die Ausgabe des ILA in *sad_core* für vier beispielhafte Fälle ist in Abbildung 20 dargestellt. Es werden 64 Sample lange Vergleichsmuster und Kanal Q als Eingang verwendet, daher ist nur Summe *sum3* relevant.

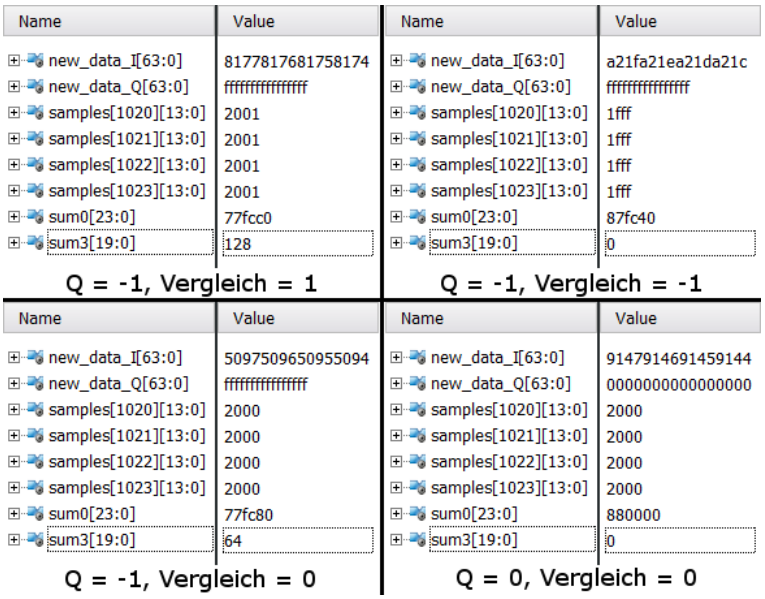

Abbildung 20: Ergebnisse des Tests des Rechenwerkes bei statischen Eingang (Kanal Q), ILA Ausgabefenster. Die Summe *sum3* ist dezimal dargestellt, alles andere hexadezimal.

Bei einem Vergleichsmuster mit konstantem Wert 1 und einem statischen Eingang, der auf -1 liegt, entsteht ein Ergebnis von 128. Bei gleichen Werten, also beide entweder -1 oder 0, ist das Ergebnis jeweils 0. Bei einem Wertepaar von -1 und 0 ist das Ergebnis 64.

Die absolute Differenz von -1 und 0 beträgt 1, bei 64 Samplen ergibt dies 64. Analog verhält es sich bei -1 und 1, dort beträgt die absolute Differenz 2 und bei 64 Samplen ergibt dies 128. Die Differenzen sind damit jeweils korrekt berechnet worden. Auch vollständige Gleichheit ist korrekterweise erkannt. Das Rechenwerk berechnet also unabhängig von den Eingangswerten die korrekte Summe bei statischen Eingang.

8.3 Test des Rechenwerkes mit veränderlichen Eingang

Das Rechenwerk muss auch dahingehend geprüft werden, ob die eingehenden Daten korrekt einsortiert und in den entsprechenden Registern weitergeschoben werden. Auch muss die Funktion des Trigger-Ausgangs bestätigt werden. Dazu wird ein steigendes Rampensignal an Kanal I angelegt und ein 1024 Sample langes Vergleichsmuster, welches ebenfalls eine steigende Rampe darstellt, eingespielt. Als Schwellwert wird 0 verwendet, es wird daher nur bei Gleichheit ein Trigger-Signal gesetzt. Das an Kanal I eingespielte Muster wiederholt sich nach 2048 Samples, dies hat erwartungsgemäß einen regelmäßigen Trigger-Puls zur Folge. In Abbildung 21 wird die Ausgabe des ILA im Rechenwerk dargestellt, in Abbildung 22 der mit dem PicoScope aufgenommene Trigger-Ausgang.

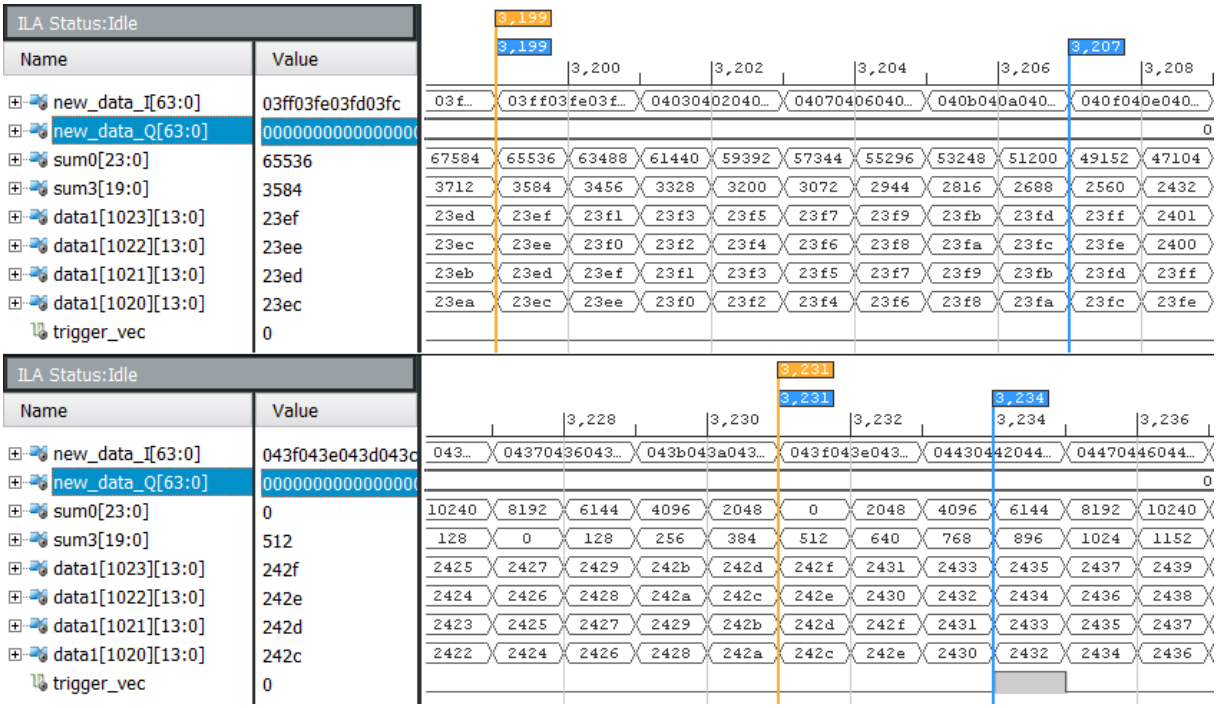

Abbildung 21: Ergebnisse des Tests des Rechenwerkes bei veränderlichen Eingang (Kanal I), ILA Ausgabefenster. Die Summen *sum0* und *sum3* sind dezimal dargestellt, alles andere hexadezimal.

In Abbildung 21 ist beim ersten Cursor, Zeitpunkt 3199, das Anliegen der letzten Werte des Vergleichsmusters, konkret 1020 ($3FC_{16}$), 1021, 1022 und 1023 ($3FF_{16}$), an Kanal I zu erkennen. Diese werden beim zweiten Cursor, Zeitpunkt 3207, in der richtigen Reihenfolge inklusive Offset in *data1* übernommen. Am dritten Cursor, Zeitpunkt 3231, ist sum0 gleich 0. Dies erzeugt am vierten Cursor, Zeitpunkt 3234, einen Trigger. Die Verzögerung von Eingang am IP-Core zu ausgelöstem Trigger beträgt daher 35 Takte.

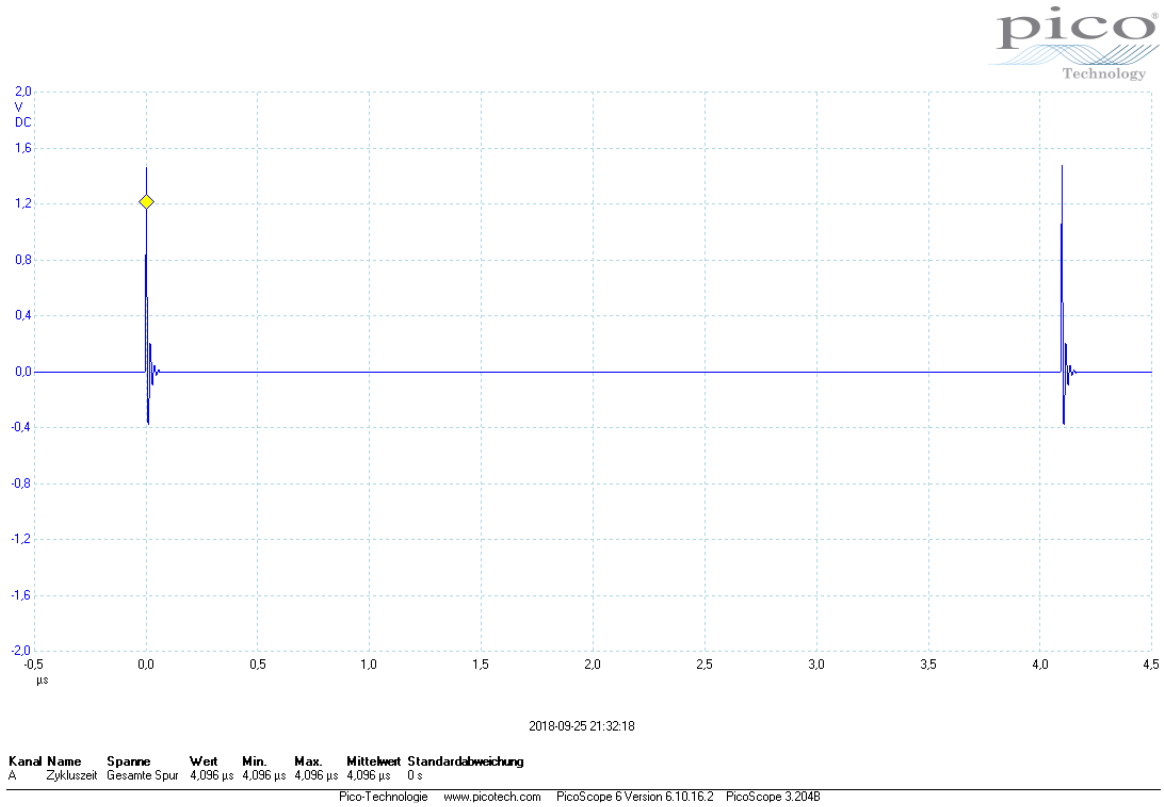

Abbildung 22: Aufnahme des Trigger-Ausgangs, Aufnahme durch PicoScope.

Wie in Abbildung 22 zu erkennen wird regelmäßig ein Trigger gesetzt. Der Abstand der Trigger-Impulse beträgt 4096 ns. Dies entspricht 2048 Takten bei einer Taktrate von 500 MHz, bzw. einer Taktperiode von 2 ns.

Die Eingangsdaten werden korrekt einsortiert und das Rechenwerk arbeitet korrekt. Der Trigger-Ausgang wird gesetzt. Bei sich wiederholenden Signalen wird auch der Trigger im gleichen Maße wiederholt ausgelöst. Die Konfiguration mittels der Konfigurationssoftware funktioniert.

Weitere Tests sind im Zuge der Entwicklung zwar durchgeführt worden, diese werden im Rahmen dieser Arbeit allerdings nicht aufgegriffen, da Sie für das Verständnis und den Beleg nicht von Relevanz sind.

Kapitel 9 - Fazit und folgende Arbeiten

Der im Rahmen dieser Arbeit entwickelte IP Core zur Mustererkennung, welcher über AXI konfiguriert werden kann, funktioniert wie vorgesehen. Es ist mit ihm möglich, 14 Bit breite Signale zu verarbeiten und bis zu 1024 Werte lange Vergleichsmuster zu erkennen. Da durch die hohe Auslastung auf dem verwendeten FPGA und dessen Taktbegrenzung nur alle zwei Sample eine Bearbeitung erfolgen kann, sinkt die effektive Samplingrate auf 500 MSpl/s. Dies ist jedoch trotzdem besser als bei vorherigen Implementationen. Weiter kann durch Verwendung eines potenteren FPGA diese Beschränkung umgangen werden, indem ein zweites Rechenwerk zusätzlich eingesetzt wird. Die Verzögerung für die Mustererkennung durch den Algorithmus beträgt 35 Take was bei 500 MHz 70 ns entspricht.

Im Vergleich mit dem icWaves bietet der entwickelte IP-Core ein offenes System, welches sich zusätzlich noch skalieren lässt. So kann das Design auch leicht auf andere FPGA/ADC Kombinationen angepasst werden, da keinerlei spezifische Eigenschaften des KC705 genutzt werden. Ferner lassen sich auch Konfigurationsoptionen einfach hinzufügen und damit der Funktionsumfang erweitern. Durch die Skalierbarkeit ist der Preis des eingesetzten FPGA-Boards gegenüber dem recht teuren icWaves angemessen.

In Zukunft kann dieser IP Core in ein vollständiges, FPGA basiertes SoC-ADC System integriert werden, beispielsweise in eines, welchen den AD-FMCDAQ2-EBZ als FMC basierten ADC verwendet. Allerdings ist der entwickelte IP-Core nicht auf diesen ADC beschränkt. Er kann ohne Modifikationen auch mit anderen ADC kombiniert werden, sofern deren Ausgangsdaten im FPGA als 4 aneinandergereihte, vorzeichenkorrekt in 16 Bit gepackte, 14 Bit Samples bei maximal 250 MHz vorliegen.

I. Abkürzungsverzeichnis

FPGA	Field Programmable Gate Array
ADC	Analog Digital Umsetzer
Spl / s	Abtastwerte pro Sekunde
FMC	FPGA Mezzanine Card
KC705	Xilinx Kintex-7 FPGA KC705 Evaluation Kit
SAD	Summe der absoluten Differenzen
NCC	Normierte Kreuzkorrelation
HPC	High Pin Count, Bauform des FMC Stecksystems
IP-Core	Intellectual property core; vielfach einsetzbarer, vorgefertigter Funktionsblock eines Chipdesigns
ILA	Integrated logic analyser; Logikanalysator-Modul als innerhalb von Vivado einsetzbarer IP-Core, mit dessen Hilfe interne Signale aufgezeichnet, angezeigt und debuggt werden können-
SSD	Summer der quadrierten Differenzen
OP-Code	Operationscode
CB	Config-Byte
MB	Microblaze Softcore-Prozessor
SoC	System on chip; ein vorständiges Computersystem mit zusätzlicher Peripherie auf einem ASIC oder FPGA
ASIC	Application specific integrated circuit; speziell entwickelter integrierter Schaltkreis.
GPIO	General purpose input/output; Allgemeiner, meistens bidirektional nutzbarer Pin in einem Elektronikumfeld
COM-Port	Unter Windows die Bezeichnung für eine serielle Kommunikationsstelle
UART	Universal asynchronous receiver transmitter, serielle Kommunikationsschnittstelle zum Senden und Empfangen von Daten über eine Datenleitung

II. Abbildungs-, Tabellen- und Listingverzeichnis

Abbildungen:

Tabellen:

Listings:

III. Quellenverzeichnis

[1] Xilinx, „FPGA Mezzanine Card (FMC) Standard,“ [Online]. Available: https://web.archive.org/web/20101228012045/http://www.xilinx.com/products/boards_kits/fmc.htm. [Zugriff am 23 Juli 2018].

[2] Riscure, „icWaves,“ [Online]. Available: https://www.riscure.com/product/icwaves/. [Zugriff am 23 Juli 2018].

[3] Xilinx, „Xilinx Kintex-7 FPGA KC705 Evaluation Kit,“ [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html. [Zugriff am 23 Juli 2018].

[4] Xilinx, „Vivado Design Suite,“ 29 August 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html. [Zugriff am 29 August 2018].

[5] Analog Devices, „AD-FMCDAQ2-EBZ,“ 29 August 2018. [Online]. Available: http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-ad-fmcdaq2-ebz.html#eb-gettingstarted. [Zugriff am 29 Juli 2018].

[6] S. Patil, J. S. Nadar, J. Gada, S. Mothare und S. S. Nair, „Comparison of Various Stereo Vision Cost,“ *International Journal of Engineering and Innovative Technology (IJEIT),* Bd. 2, Nr. 8, pp. 222-226, Februar 2013.

[7] N. Roma, J. Santos-Victor und J. Tomé, „A Comparative Analysis of Cross-Correlation Matching Algorithms Using a Pyramidal Resolution Approach,“ *Empirical Evaluation Methods in Computer Vision,* Nr. 50, pp. 117-142, 2002.

[8] A. Giachetti, „Matching techniques to compute image motion,“ *Image and Vision Computing,* Bd. 3, Nr. 18, pp. 247-260, Februar 2000.

[9] J. P. Lewis, „Fast Normalized Cross-Correlation,“ 1995. [Online]. Available: http://scribblethink.org/Work/nvisionInterface/nip.html#foot263. [Zugriff am 24 August 2018].

[10] Y. Jea-Chem, C. Byoung Deog und C. Hyoung-Kee, „1-D fast normalized cross-correlation using additions," *Digital Signal Processing,* Nr. 5, pp. 1482-1493, 01 September 2010.

[11] S. Vassiliadis, E. Hakkennes, J. Wong und G. Pechanek, „The sum-absolute-difference motion estimation accelerator," 27 August 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.5427&rep=rep1&type=pdf. [Zugriff am 20 August 2018].

[12] Xilinx, „MicroBlaze Soft Processor Core," 29 August 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html. [Zugriff am 29 August 2018].

[13] J. G. Earle, „Latched Carry Save Adder Circuit for Multipliers". USA Patent 3,340,388, 12 Juli 1965.

[14] Mentor, „Modelsim," [Online]. Available: https://www.mentor.com/products/fv/modelsim/. [Zugriff am 15 Oktober 2018].

[15] Pico Technology, „Pico Oscilloscopes," [Online]. Available: https://www.picotech.com/products/oscilloscope. [Zugriff am 24 Oktober 2018].

[16] freeRTOS, „The FreeRTOS™ Kernel," [Online]. Available: https://www.freertos.org/index.html. [Zugriff am 24 Oktober 2018].

[17] Microsoft, „Visual Studio," [Online]. Available: https://visualstudio.microsoft.com/de/. [Zugriff am 24 Oktober 2018].

[18] Y. M. Fouda, „One-Dimensional Vector based Pattern Matching," 10 September 1994. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1409/1409.3024.pdf. [Zugriff am 24 August 2018].

IV. Anhangsverzeichnis

Quellcodes können bei den beiden Prüfern:
Herrn Prof. Dr.-Ing. Fitz
und
Herrn Prof. Dr.-Ing. habil. Michael Berger
eingesehen werden.

Nachfolgend dargestellt ist das kommentierte Hauptverzeichnis der angefügten CD

Name	**Beschreibung**
Masterarbeit.pdf	Dieses Dokument
Konfigurationssoftware	Verwendete C#.NET Quellcodes für die Konfigurations-software
Originalprojekte	Verwendete Projekte für Vivado, Xilinx SDK, Visual Studio
Quellen	Einige verwendete Quellen
SoC Firmware Quelltexte	Verwendete C++-Quellcodes für die Firmware
VHDL Module	Verwendete VHDL Quellcodes

Eigenständigkeitserklärung

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken oder dem Internet wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht. Die vorliegende Arbeit wurde keinem anderen Prüfungsgremium vorgelegt.

Hamburg, 30.10.2018

_____