DIPLOMARBEIT

Fachgebiet der Diplomarbeit:
IC-Entwurf

Thema der Diplomarbeit:
Entwurf und Aufbau eines Rechnersystems zur hochgenauen Messung der Laufzeit von Impulsen als digitales „system on chip“ auf einem FPGA

Unternehmen, in dem die Diplomarbeit durchgeführt wurde:
Fresenius Medical Care GmbH, R&D, Gruppe Electronics & Software Engineering
in Bad Homburg v.d. Höhe

Diplomand:	Alexander Kühn
Referent:	Prof. Dipl.-Ing. Klaus Kief
Korreferent:	Prof. Dr.-Ing. Hans-Dieter Spindler
Betreuer seitens Fresenius Medical Care:	Dipl.-Ing. Dejan Nikolic

Wintersemester 2002/2003

Fachhochschule Gießen-Friedberg
Bereich Friedberg
Fachbereich E II
Fachrichtung Technische Informatik

Zusammenfassung

In dieser Diplomarbeit wird die Entwicklung eines Systems zur hochgenauen Messung von Laufzeiten von Ultraschall-Impulsen vorgestellt.

Die Messung der Ultraschall-Laufzeiten dient zur Bestimmung der Blutdichte während einer Dialysebehandlung.

Das Ziel dieser Diplomarbeit ist die Integration der digitalen Verarbeitungseinheit eines schon vorhandenen Systems als „system on chip“ auf einem FPGA. Im Zuge dieser Integration wird eine neue Leiterplatte entworfen, die zu dem bestehenden System elektrisch und mechanisch kompatibel ist.

Die digitale Verarbeitungseinheit des originalen Systems besteht aus einem DSP, der die Berechnung der Ultraschall-Laufzeit durchführt sowie einem FPGA, das diverse Steuerfunktionen u.a. zur Digitalisierung der Ultraschallsignale vornimmt.

Einleitend wird die Dialysebehandlung und das Messverfahren zur Ultraschall-Laufzeitmessung beschrieben. Darauf folgt eine Beschreibung der Eigenschaften des Ultraschallsignals und eine Vorstellung des zur Laufzeitmessung eingesetzten Algorithmus. Im weiteren wird der allgemeine Aufbau eines FPGA und der Entwicklungsprozess einer FPGA-Schaltung erörtert.

Der Schwerpunkt dieser Arbeit liegt auf der Implementierung der geforderten Funktionaliäten auf einen FPGA. Einen weiteren Schwerpunkt bildet der Aufbau einer neuen Leiterplatte, in der das neue digitale System integriert ist.

Abschließend folgen Erläuterungen zur Inbetriebnahme des Systems sowie eine zusammenfassende Bewertung der Ergebnisse.

Das System ist, mit dem in dieser Arbeit vorgestellten Lösungsansatz, in der Lage, Ultraschall-Laufzeiten mit einer zeitlichen Auflösung von mindestens $\pm$ 200 ps zu messen.

Eidesstattliche Erklärung

Hiermit erkläre ich, daß ich die Arbeit selbständig und nur unter Zuhilfenahme der aufgeführten Hilfsmittel sowie den Hinweisen meines Betreuers angefertigt habe.

Friedberg, den 15.03.2003

(Alexander Kühn)

Danksagung

An dieser Stelle möchte ich mich bei Herrn Dejan Nikolic, meinem Betreuer bei der Firma Fresenius Medical Care, für die Betreuung meiner Diplomarbeit bedanken.

Ebenso möchte ich mich bei Herrn Franz-Wilhelm Koerdt, dem Leiter der Gruppe Electronics & Software Engineering, bedanken, dass ich die Diplomarbeit in seiner Gruppe durchführen konnte.

Mein besonderer Dank gilt den Fresenius-Mitarbeitern Herrn Peter Scheunert, Herrn Walter Hansen und Herrn Andreas Röse für die wertvollen Hinweise zu schaltungstechnischen Fragen sowie Frau Birgit Hieronymi für die Erstellung des Platinenlayouts.

Ebenfalls bedanken möchte ich ich mich für die Betreuung meiner Diplomarbeit bei Herrn Prof. Dipl.-Ing. Klaus Kief als Referenten sowie Herrn Prof. Dr.-Ing. Hans-Dieter Spindler als Korreferenten.

Weiterhin möchte ich mich bei meinen Eltern Helga Kühn und Hans-Otto Kühn für die Unterstützung während meines Studiums bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

0. Glossar

ALU	Abkürzung für „Arithmetic and Logic Unit“ (Rechenwerk)
ASIC	Kundenspezifischer Schaltkreis
CMOS	Complementary-Metal-Oxide-Semiconductor
CPLD	Komplexer PLD (Complex PLD)
Dialyse	Blutwäsche
Dualport-RAM	Speicher, der über zwei unabhängige Kanäle gelesen und beschrieben werden kann
DSP	Digitaler Signalprozessor
FET	Feldeffekt-Transistor
FPGA	Field Programmable Gate Array (programmierbarer Logikbaustein)
GAL	Gate Array Logic
HDL	Hardware Definition Language
IC	Integrierte Schaltung (Integrated Circuit)
IP	Intellectual Property
Jitter	zeitliche Schwankungen eines Signals
JTAG	Standardschnittstelle zum Test von ingerierten Schaltungen (Joint Test Action Group)
LSB	Niederwertigstes Bit (Least Significant Bit)
MSB	Höchstwertiges Bit (Most Significant Bit)
OTP	einmal programmierbarer Baustein (One Time Programmable)
PAL	Programmable Array Logic
PLD	Programmierbare logische Schaltung (Programmable Logic Device)
PQFC	kleines vierreihiges IC-Gehäuse (Pin Quad Flat Package)
RAM	Speicher mit wahlfreiem Zugriff (Random Access Memory)
ROM	Festwertspeicher (Read Only Memory)
Sample	Abtastwert

SPI-Schnittstelle	Serial Peripheral Interface, synchrone serielle Schnittstelle
SRAM	statischer RAM
TTL	Transistor-Transistor-Logik
VHDL	Hardwarebeschreibungssprache (VHSIC Hardware Description Language)
VHSIC	Bezeichnung für hochintegrierte (schnelle) Schaltkreise (Very High Speed Integrated Circuits)

1. Einleitung

1.1. Dialysebehandlung

Die menschliche Niere erfüllt viele Funktionen im Körper.

Sie reinigt den Blutkreislauf von giftigen Stoffwechsel-Endprodukten, entfernt überschüssige Flüssigkeit, sie reguliert den Säure-Haushalt im Körper genauso wie den Elektrolyt-Haushalt, reguliert den Blutdruck und die Produktion roter Blutkörperchen sowie die Aufnahme von Kalzium.

Bei einer Fehlfunktion der Niere ist es unbedingt erforderlich, Therapiemaßnahmen zu treffen, da der Körper des Patienten nicht mehr fähig ist, giftige Stoffwechselprodukte sowie angesammeltes Wasser aus dem Blutkreislauf zu entfernen.

Mögliche Therapiemaßnahmen sind:

- Transplantation der Niere
- Dialyse

Weltweit leiden ca. eine Million Menschen an chronischem Nierenversagen[1]. Da Spenderorgane für die Nierentransplantation nicht in ausreichender Zahl verfügbar sind und nicht jeder Patient die Transplantation der Niere verträgt, kommt für viele Menschen derzeit nur die Dialyse als Therapiemaßnahme in Betracht.

In der Dialysetherapie werden zwei wichtige Grundverfahren eingesetzt :

- Peritonealdialyse
- Hämodialyse

Bei der Peritonealdialyse wird das Bauchfell als Dialyse-Membran verwendet. Da die Peritonealdialyse nicht Gegenstand dieser Diplomarbeit ist, wird hier jedoch nicht näher darauf eingegangen.[2]

Die Hämodialyse ist ein extrakorporales Blutreinigungsverfahren, d.h. das Blut des Patienten wird an eine externe Maschine geführt, dort gereinigt und wieder

[1] Stand 2001, vgl. dazu auch [1]

[2] für weitergehende Informationen wird auf [2] verwiesen

dem Patienten zugeführt. In Abbildung 1-1 ist das Verfahren der Hämodialyse aufgezeigt.

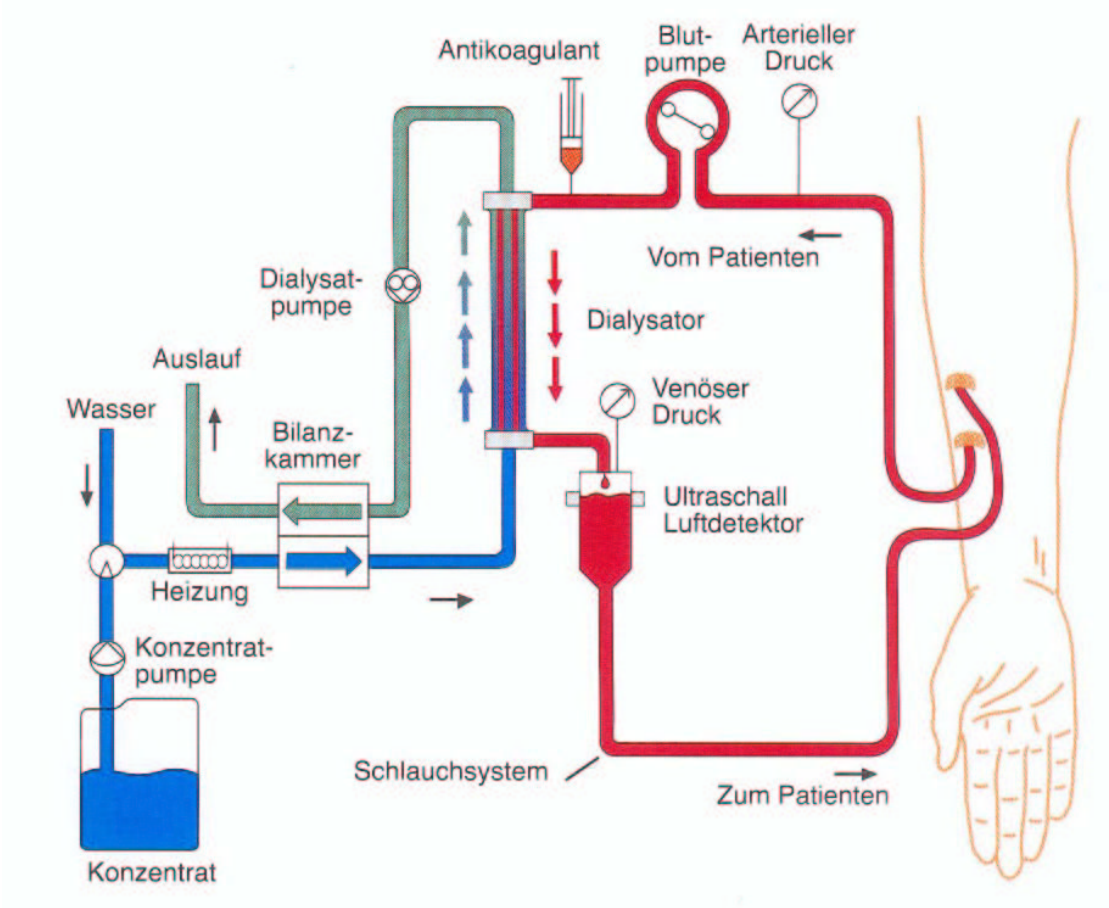


Abbildung 1-1: Prinzip der Hämodialyse

Der Patient erhält einen zentralen Gefäßzugang zwischen Arterie und Vene, den so genannten Shunt. Dieser Shunt ermöglicht es, hohe Blutflussraten (die bei der Dialyse zwischen 200-300 ml/min liegen) zu erzielen.

Das Blut wird mit Hilfe einer Blutpumpe über das arterielle Schlauchsystem aus einer arteriellen Blutentnahmekanüle in die Maschine befördert. Da das Blut außerhalb des Körpers schnell gerinnt, wird ein gerinnungshemmendes Mittel hinzugegeben, das so genannte Antikoagulant[3].

Das Blut durchströmt den Dialysator, in dem die Reinigung des Bluts stattfindet. Anschließend wird das gereinigte Blut über das venöse Schlauchsystem zu einer venösen Kanüle geleitet und dem Patienten wieder zugeführt.

[3]zumeist wird hierfür der Wirkstoff Heparin verwendet

Der Dialysator besteht aus ca. 10000 bis 15000 feinen Kapillaren, die von Blut durchströmt werden. Diese Kapillaren werden wiederum von der Dialysierflüssigkeit umspült. Somit bildet der Dialysator eine semipermeable Membran zwischen dem Blutkreislauf und dem Dialysatkreislauf.

Die Dialysemembran lässt nur Teilchen zur gegenüberliegende Seite passieren, wenn sie kleiner als die Membranporen sind. Größere Substanzen, wie z.B. die roten und weißen Blutkörperchen oder Bluteiweiße, können nicht durch die Membran in die Dialysierflüssigkeit übertreten. Kleine Partikel, wie z.B. Mineralien oder die giftigen Stoffwechselprodukte können die Membran jedoch durchdringen.

Die Dialysierflüssigkeit wird von der Maschine aus Wasser und Dialysekonzentrat (konzentrierte Salzlösung) gemischt. Nach dem Durchlaufen des Dialysators wird die Dialysierflüssigkeit in das Abwasser gegeben.

Im Dialysator sind zwei physikalische Effekte für die Reinigung des Blutes maßgeblich:

Diffusion: Hierbei wird die Tatsache ausgenutzt, dass Stoffe, die auf einer Seite der Membran in hoher Konzentration gelöst sind zu der Seite mit niedrigerer Konzentration übertreten, bis die Konzentration auf beiden Seiten ausgewogen ist. Die giftigen Stoffwechselprodukte werden von der Dialysierflüssigkeit aus dem Blut aufgenommen.

Ultrafiltration: Durch die Bildung von Unterdruck auf der Seite der Dialysierflüssigkeit und Überdruck auf der Seite des Blutes kann Wasser aus dem Blut entzogen werden.

Da immer nur ein kleiner Teil des gesamten Blutvolumens eines menschlichen Körpers die Dialysemaschine durchströmt, dauert eine Dialysebehandlung sehr lang. Eine durchschnittliche Dialysebehandlung dauert ca. vier bis fünf Stunden und muss ca. dreimal pro Woche durchgeführt werden.

1.2. Projektbeschreibung

Wie in Abbildung 1-1 zu sehen ist, müssen verschiedene Verfahrensparameter für eine erfolgreiche Dialysebehandlung überwacht werden. Verschiedenartige Senso-

ren messen ständig unter anderem die Drücke und Temperaturen auf der arteriellen und venösen Seite des extrakorporalen Blutkreislaufs.

Ein weiterer wichtiger Verfahrensparameter ist die Bestimmung der Blutdichte.

Die Dichte des Blutes verändert sich im Laufe einer Dialysebehandlung aufgrund des fortschreitenden Wasserentzugs durch die Ultrafiltration. Für die Qualität der Dialysebehandlung ist es wichtig, die Ultrafiltrationsrate so zu regeln, dass dem Patienten nicht zu viel Wasser in zu kurzer Zeit entzogen wird und dennoch am Ende der Dialysebehandlung genügend Wasser aus dem Körper des Patienten entfernt wurde.

Zur Bestimmung der Blutdichte wird eine Ultraschall-Laufzeitmessung vorgenommen. Die Laufzeit eines Ultraschallsignals durch ein Medium ist umgekehrt proportional zu der Schallgeschwindigkeit des Mediums. Die Schallgeschwindigkeit hängt wiederum ab von der Temperatur und der Dichte des Mediums.

Die Temperatur des Blutes wird über Temperatursensoren ermittelt. Dadurch ist es möglich, die Dichte des Blutes unter Einbeziehung der Bluttemperatur aus der Ultraschall-Laufzeit zu berechnen.

Die Ultraschall-Laufzeit wird mit Hilfe einer Ultraschallsensorik gewonnen. Diese Ultraschallsensorik führt zusätzlich eine Erkennung von Luftblasen im Blut durch.

Abbildung 1-2 zeigt den prinzipiellen Aufbau der Ultraschallsensorik.

Das vom Patienten ankommende Blut wird durch das arterielle Schlauchsystem in das Blutmodul geführt, in dem die Reinigung des Blutes erfolgt. Das Blut wird dabei durch eine Ultraschallmessstrecke geführt, die aus einem Ultraschallsender und einem gegenüberliegenden Ultraschallempfänger besteht (arterieller US-Sensor).

Das gereinigte Blut wird über das venöse Schlauchsystem aus dem Blutmodul herausgeführt. Hierbei passiert das Blut wiederum eine Ultraschallmessstrecke (venöser US-Sensor).

Die Ultraschallsender sind in einem bestimmten Abstand zu den Empfängern angeordnet (s_{art}, s_{ven}). Diese Abstände der Sensoren zueinander werden mit Abstandssensoren ermittelt und den übergeordneten Rechnersystemen zur Berechnung der Schallgeschwindigkeit zur Verfügung gestellt.

Die Ansteuerung der Ultraschallsensoren und die Bestimmung der Ultraschall-

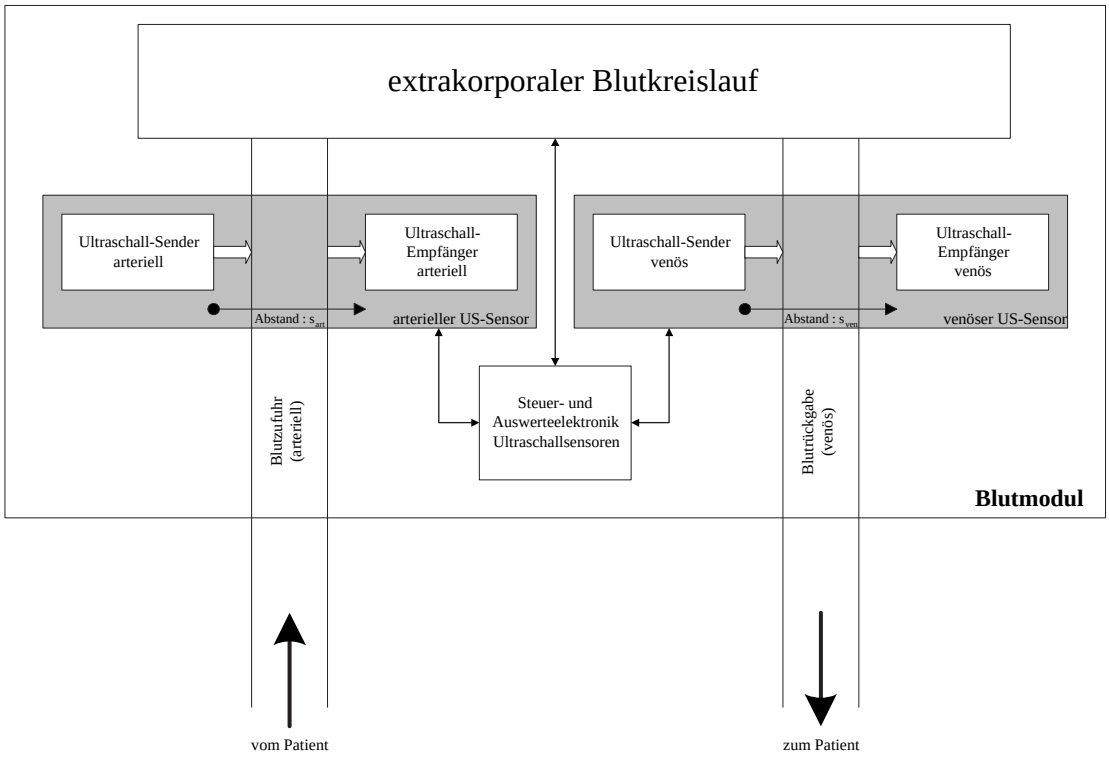


Abbildung 1-2: Aufbau der Ultraschallsensorik

Laufzeit wird über eine Steuer- und Auswerteelektronik durchgeführt.

Die Weiterentwicklung einer vorliegenden Steuer- und Auswerteelektronik ist das Ziel dieser Diplomarbeit.

In einer von Fresenius Medical Care bereits entwickelten Schaltung wird die Steuerung der Ultraschallsensoren von einem FPGA übernommen, während die Bestimmung der Ultraschall-Laufzeit von einem DSP durchgeführt wird.

Die Detektion der Luftblasen wird in dieser Arbeit nicht behandelt, da die sicherheitstechnischen Betrachtungen dieses Verfahrensparameters den Umfang dieser Arbeit übersteigen würden.

1.3. Anforderungen

Gegenstand dieser Arbeit ist die Integration der Steuerung und der Laufzeitauswertung auf einem einzigen FPGA sowie die Untersuchung, ob die Realisierung basierend auf FPGA-Technologie ökonomische und technische Vorteile gegenüber

der ursprünglichen Realisierung bietet.

Ebenfalls Gegenstand dieser Arbeit ist die Erörterung der Frage, ob es möglich und sinnvoll ist, auf dem Markt erhältliche IP-Kerne für diese Aufgabenstellung zu nutzen.

Die zu entwickelnde Steuer- und Auswerteelektronik muss verschiedenen Anforderungen genügen. Die Anforderungen an dieses System sind in Tabelle 1-2 zusammengefasst.

Generell ist hierbei zu sagen, dass die neu zu entwickelnde Steuer- und Auswerteelektronik in Bezug auf die elektrischen und mechanischen Eigenschaften vollständig kompatibel zu der bereits vorhandenen Elektronik sein muss.

Anforderungen an die Ultraschall-Laufzeitmessung	
Messbereich	6 μs bis 10 μs
Messgenauigkeit	$\pm$200 ps
Messart	nur relative Änderung der Laufzeit ab Beginn der Dialysebehandlung
maxmimale Aktualisierungszeit der Messwerte	100 ms

mechanische Anforderungen	
Platinenabmessungen (BxH)	110 mm x 60 mm
maximale Bauhöhe	10 mm
Steckverbindung zu den Ultraschallsensoren	MCX-Koaxsteckverbinder
Systemschnittstelle	SMC-Stecker 68pol.
Platzierung Systemschnittstelle	aus Originalplatinen-Layout

Versorgungsspannung	
Zur Verfügung stehende Versorgungsspannungen	2,5 V; 3,3 V; 5 V; 24 V
max. Belastbarkeit 2,5 V Versorgung	300 mA
max. Belastbarkeit 3,3 V Versorgung	300 mA
max. Belastbarkeit 5 V Versorgung	300 mA
max. Belastbarkeit 24 V Versorgung	100 mA

Systemschnittstelle	
Datenübertragung	SPI-Schnittstelle, Ultraschallplatine=SPI-Slave
FPGA-Konfiguration	per Jumper wählbar zwischen Onboard oder extern

Sonstiges	
Kosten	im Idealfall günstiger als bereits vorhandenes Modul

Tabelle 1-2: Anforderungsliste der Ultraschallelektronik

2. Grundlagen

2.1. Systemaufbau

Der Aufbau der Steuer- und Auswerteelektronik zur Ultraschall-Laufzeitmessung ist in Abbildung 2-3 dargestellt.

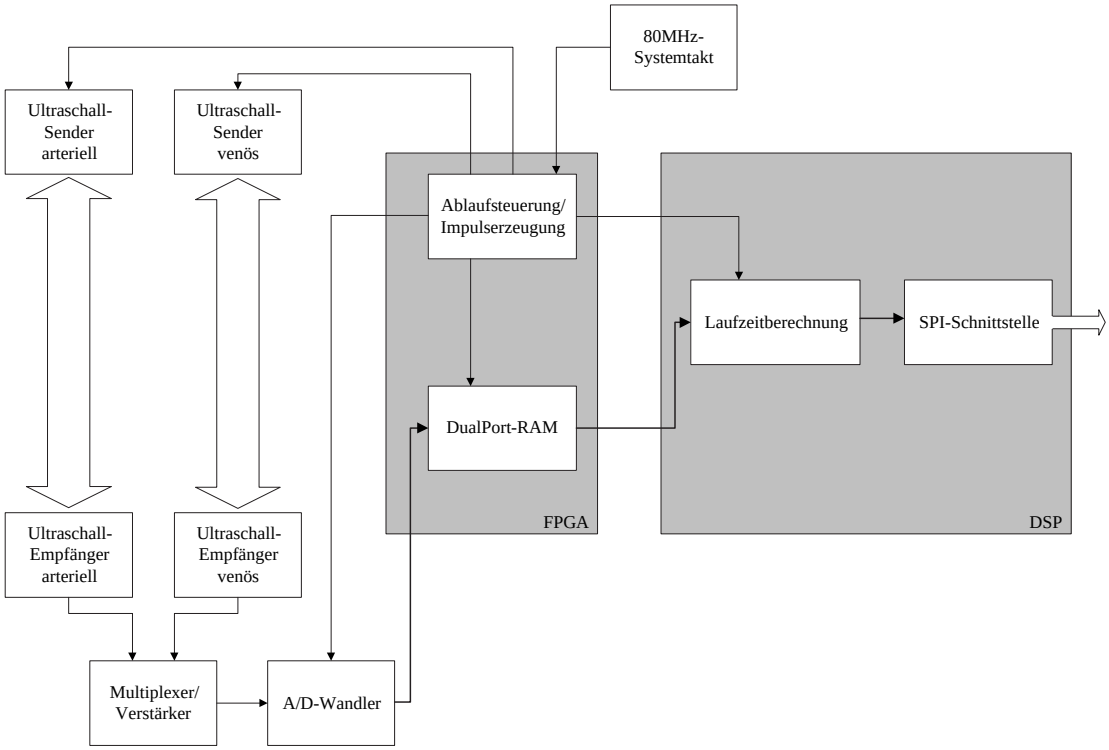


Abbildung 2-3: Systemaufbau

Ein temperaturkompensierter Quarzoszillator erzeugt einen 80 MHz Systemtakt. Da die angestrebte Genauigkeit der Ultraschall-Laufzeitmessung im Bereich von $\pm 200\ ps$ liegen soll, ist es erforderlich, hier einen jitterarmen Oszillator einzusetzen.

Die Ablaufsteuerung erzeugt Impulse, um abwechselnd den arteriellen und venösen Ultraschallsender anzuregen, den Eingansmultiplexer des A/D-Wandlers umzuschalten und den Wandlungsvorgang des A/D-Wandlers zu steuern.

Der A/D-Wandler liest die Signale aus der Ultraschall-Empfangsschaltung mit dem Systemtakt ein, daraus ergibt sich eine Abtastzeit von

$$t_{sample} = \frac{1}{f_{System}} = \frac{1}{80\ MHz} = 12.5\ ns \tag{2.1}$$

Die Wortbreite der vom A/D-Wandler gelieferten Abtastwerte liegt bei 8 bit. Die Abtastwerte werden in ein Dualport-RAM eingelesen.

Die Ultraschall-Laufzeit ergibt sich aus der Mittelung über mehrere einzelne Laufzeitmessungen.

Die gemittelte Laufzeit muss ca. alle 100 ms aktualisiert werden. Bei einer Mittelung von 100 Laufzeiten müssen beide Kanäle abwechselnd im Abstand von 500 μs zyklisch ausgelesen und die Laufzeit berechnet werden. Abbildung 2-4 zeigt den zeitlichen Ablauf für die Laufzeitmessung einer einzelnen arteriellen und venösen Laufzeit.

Dieser Ablauf wird im folgenden als Messzyklus bezeichnet.

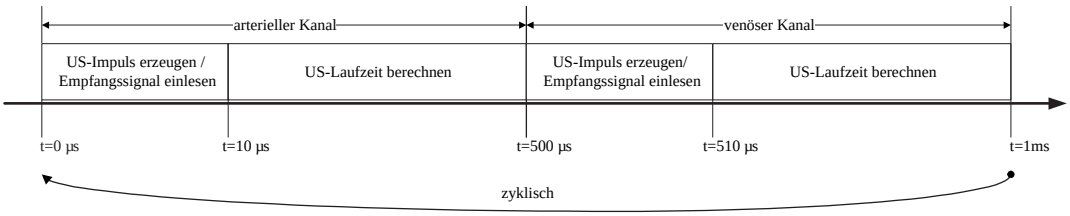


Abbildung 2-4: zeitlicher Ablauf der Ultraschallmessung

Nach $t = 0\ \mu s$ wird ein Anregungsimpuls für den arteriellen Ultraschallsender erzeugt und die A/D-Wandlung des arteriellen Empfangssignals gestartet.

Nach $t = 10\ \mu s$ wurden $10\ \mu s \times t_{sample} = 800$ Abtastwerte eingelesen. Die Laufzeitberechnung für den arteriellen Kanal wird gestartet.

Nach spätestens $t = 500\ \mu s$ ist eine neue arterielle Laufzeit fertig berechnet.

Nach $t = 500\ \mu s$ wird der Anregungsimpuls für den venösen Ultraschallsender erzeugt und die A/D-Wandlung des venösen Empfangssignals gestartet.

Nach $t = 510\ \mu s$ wurden 800 Abtastwerte aus dem venösen Empfänger eingelesen. Die Laufzeitberechnung für den venösen Kanal wird gestartet.

Nach spätestens $t = 1\ ms$ ist eine neue venöse Laufzeit fertig berechnet.

Wurden für jeden Kanal jeweils 100 Laufzeiten gemessen, so werden die 100 Messwerte für jeden Kanal gemittelt und an das SPI-Interface übermittelt.

Der Messzyklus beginnt wieder bei $t = 0\ \mu s$.

2.2. Eigenschaften des Ultraschallsignals

Zur Anregung der Ultraschallsender-Piezoelementes ist es erforderlich, einen Spannungssprung an dieses Piezoelement anzulegen. Dieser Spannungssprung beträgt -47 V gegenüber dem Ruhepotential. Eine Aufnahme des Sendeimpulses ist in Abbildung 2-5 zu sehen.

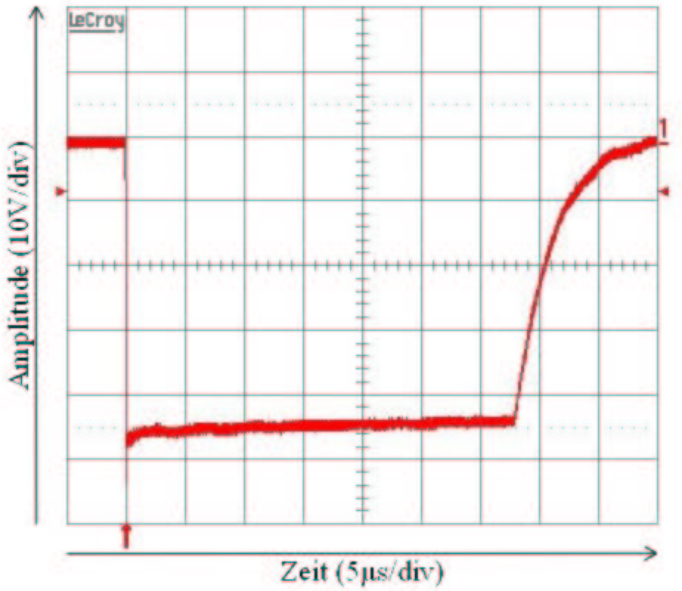


Abbildung 2-5: Ultraschall-Sendeimpuls

Die Aufnahme wurde mit einem Oszilloskop direkt am Senderausgang der Originalschaltung aufgenommen.

Es wird ein sehr steilflankiger Sprung auf $-47\ V$ erzeugt. Dieser Pegel wird über einen Zeitraum von ca. 30 μs gehalten, um die 10 μs dauernde Messung durch die nachfolgende steigende Flanke nicht zu stören.

Das vom Ultraschallempfänger-Piezoelement aufgenommene Signal ist in Abbildung 2-6 wiedergegeben.

Auf der Ordinate ist die Amplitude des Signals aufgetragen, auf der Abszisse die laufende Nummer der Abtastwerte (Samples). Die Daten wurden durch Auslesen des Samplespeichers aus der bestehenden Schaltung in einen PC übertragen und in der Plotsoftware GNUPLOT grafisch dargstellt.

Das Empfangssignal repräsentiert die Sprungantwort des aus den Komponenten

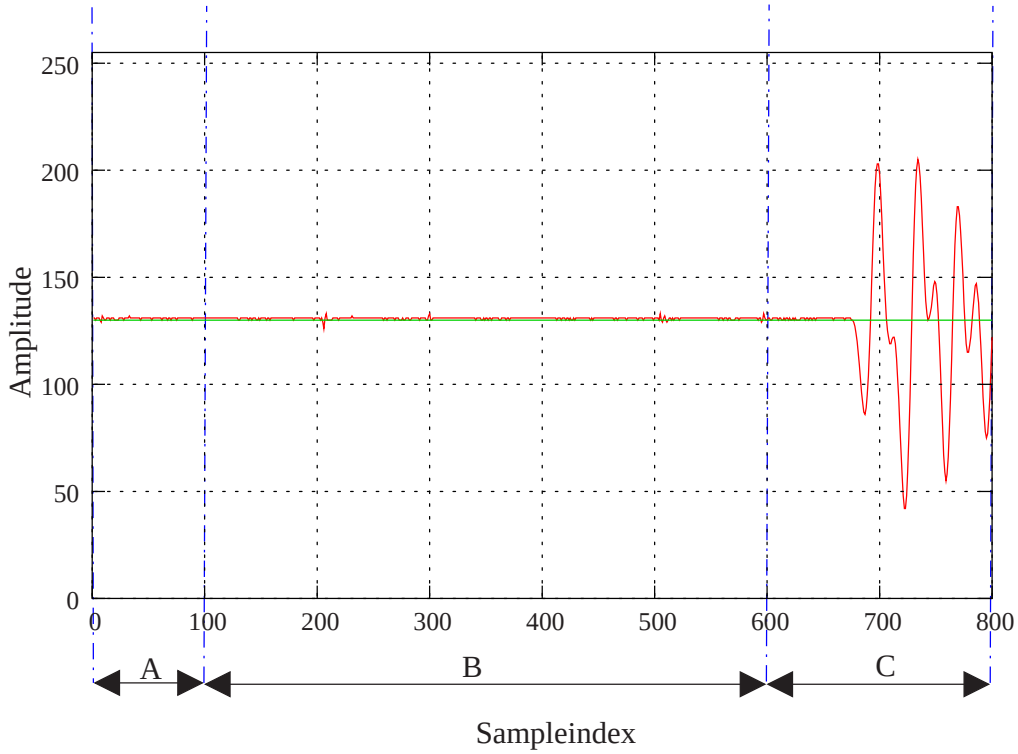


Abbildung 2-6: Ultraschall-Empfangssignal

- Ultraschallsende-Piezoelement,
- von Blut durchströmte Messzelle und
- Ultraschallempfänger-Piezoelement

gebildeten Systems.

Der Signalverlauf unterteilt sich in mehrere charakteristische Bereiche:

Bereich A: In diesem Bereich kann es unter Umständen zu einem Übersprechen des Sendesignals kommen, daher sollte dieser Bereich nicht in die Signalauswertung einfließen. Dieser Bereich erstreckt sich von Sampleindex 0 bis ca. 100.

Bereich B: Das Signal befindet sich danach in der Mitte des vom A/D-Wandler darstellbaren Wertebereichs (in Abbildung 2-6 grüne Linie) und ist von Rauschen und sporadischen Störungen überlagert.

Da der eingesetzte A/D-Wandler ein unipolarer Typ ist, entspricht dieser Mittelwert dem Nullwert des Empfangssignals, der so genannten *Baseline*. Dieser Bereich erstreckt sich von Sampleindex 100 bis ca. 400.

Bereich C: Nach einer bestimmten Zeit kommt der an den Sender angelegte Amplitudensprung an dem Empfänger an. Das Empfänger-Piezoelement schwingt dann auf seiner Resonanzfrequenz von ca. 2 MHz ein.

Die Zeit, die benötigt wird, bis der Empfänger auf seiner Resonanzfrequenz zu schwingen beginnt, entspricht der Laufzeit des Ultraschallsignals t_{us}.

Die Laufzeit kann in der hier eingesetzen Messanordnung im Bereich von sieben bis neun μs liegen. Während der Dialysebehandlung kann sich aufgrund des fortschreitenden Wasserentzugs die Laufzeit um 100 ns verändern.

Abbildung 2-7 zeigt das Empfangssignal im Bereich des Einsetzens der Resonanzschwingung des Empfängers.

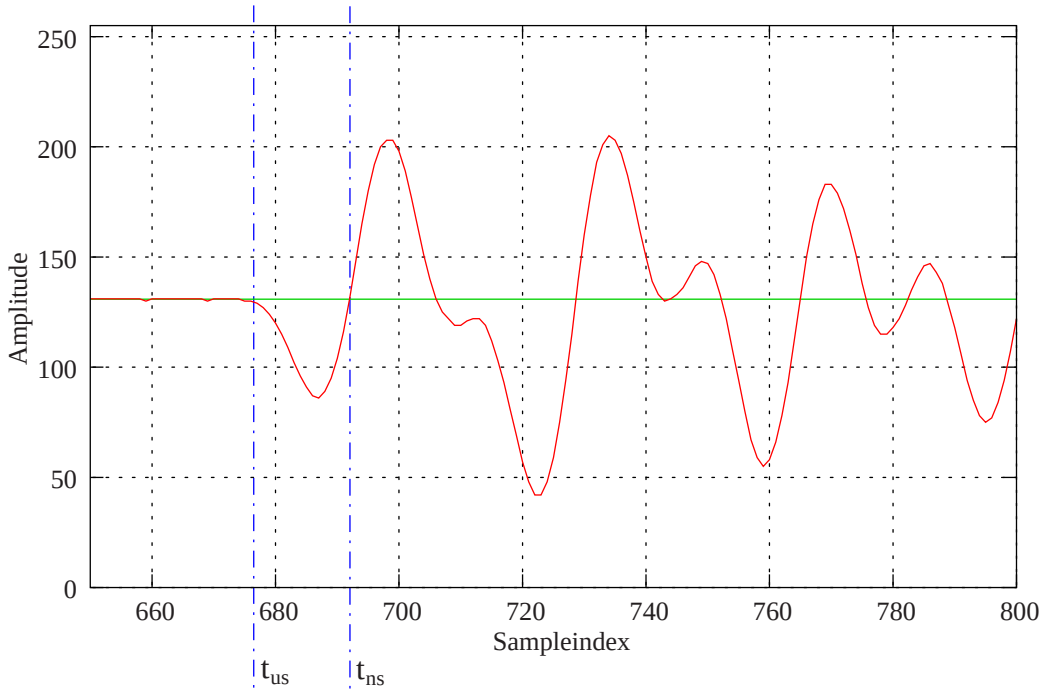


Abbildung 2-7: Ultraschall-Empfangssignal im Resonanzbereich

Da nur die Änderung der Blutdichte während einer Dialysebehandlung gemessen werden muss, ist es nicht notwendig die absolute Ultraschall-Laufzeit zu ermitteln.

Somit ist es nicht zwingend notwendig, den wahren Beginn der Resonanzschwingung t_{us} zu detektieren. Stattdessen kann man einen leichter zu erkennenden Zeitpunkt auswählen. Hier bietet sich der erste Nulldurchgang nach dem ersten negativen Wellenbauch an (t_{ns}). In diesem Punkt verläuft die Signalkurve sehr steil und der Schnittpunkt mit der Baseline ist sehr genau bestimmbar.

Das empfangene Signal besitzt einige Eigenschaften, welche die Erkennung des eingeschwungenen Zustands erleichtern:

- das Verweilen des Signals um die Baseline herum ist für eine gewisse Mindestzeit garantiert
- das Einschwingen beginnt bei der vorgegebenen Sprungform des Sendeimpulses grundsätzlich mit einem negativen Wellenbauch
- eine minimale Amplitude des eingeschwungenen Signals ist, außer im Fehlerfall, gegeben
- die Resonanzfrequenz ist relativ konstant.

2.3. Algorithmus zur Laufzeiterkennung

Zur Auswertung der Ultraschall-Laufzeit wurde ein Algorithmus entwickelt, der auf dem in der Originalschaltung implementierten Algorithmus basiert.

2.3.1. Überblick

Der Algorithmus zur Laufzeiterkenung untergliedert sich in mehrere Teilaufgaben. Das Zusammenspiel dieser Teilaufgaben ist in Abbildung 2-8 wiedergegeben.

1. Zunächst werden die Variablen, die zur Berechnung nötig sind, initialisiert.
2. Der Ultraschallsendeimpuls wird erzeugt und die Samples des Empfangssignals werden in einen Speicher eingelesen.

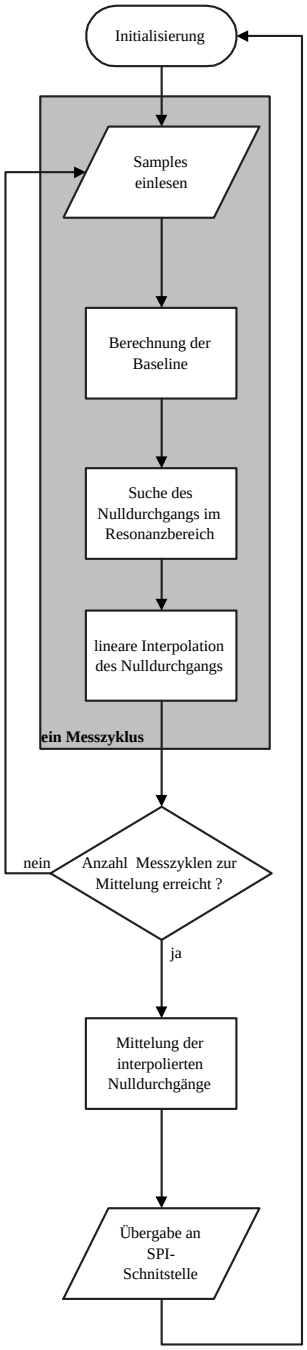


Abbildung 2-8: Algorithmus zur Laufzeiterkennung

3. Der Mittelwert des Empfangssignals im Ruhebereich wird gebildet; dieser Mittelwert entspricht der Baseline des Signals.

4. Der erste Durchgang des Signals durch die Baseline (Nullstelle) im Resonanzbereich wird gesucht und der untere und obere Sampleindex des Übergangs zwischengespeichert.

5. Anhand der Sampleindizes des Durchgangsbereiches wird eine lineare Interpolation des exakten Nulldurchgangs berechnet. Dieser Wert wird zwischengespeichert.

6. Wurde eine bestimmte Anzahl von Ultraschall-Laufzeiten gemessen, so wird darüber ein Mittelwert gebildet und an die übergeordneten Systeme via SPI weitergegeben. Danach beginnt der Algorithmus von vorne. Wurden jedoch noch nicht genügend Laufzeiten ermittelt, so wird sofort ein weiteres Ultraschallsignal eingelesen und berechnet.

Die Berechnung der Baseline findet im Bereich von Sampleindex 100 - 400 statt, da hier, wie im vorigen Abschnitt erwähnt, das Signal garantiert in Ruhe ist.

Die Suche der Nullstelle wird erst ab Sampleindex 400 begonnen, da der Bereich der Resonanz frühestens hier beginnen kann.

2.3.2. Teilalgorithmus Baselineberechnung

Um den Nulldurchgang im Resonanzbereich erkennen zu können, ist es nötig, zu wissen, in welchem Wertebereich die Baseline des Signals liegt. Der Wert der Baseline befindet sich im mittleren Feld des vom A/D-Wandler darstellbaren Bereichs. Der exakte Wert der Baseline ist jedoch abhängig von den Offseteinstellungen der analogen Verstärker und des A/D-Wandlers sowie über die Betriebszeit der Schaltung schwankenden Parametern wie z.B. der Betriebstemperatur und die Konstanz der Spannungsversorgung.

Daher ist es erforderlich, die Baseline für jeden Messzyklus erneut zu ermitteln.

Wie in Abbildung 2-6 zu sehen ist, befindet sich das Signal in einem langen Zeitraum (Abschnitt B in der Abbildung) in Ruhe.

Die Berechnung der Baseline erfolgt durch die Bildung des Mittelwerts der Abtastwerte über den Ruhebereich.

Der Mittelwert wird über den Samplebereich von Sampleindex $a = 100$ bis $b = 400$ gebildet. Somit ist garantiert, dass das anfängliche Übersprechen des Sendeimpulses und der Resonanzbereich des Signals nicht in die Berechnung einfliessen.

Der Mittelwert wird nach der Vorschrift

$$Baseline = \frac{1}{(b-a+1)} \sum_{i=a}^{b} x(i) \tag{2.2}$$

berechnet, wobei $x(i)$ dem Abtastwert von Sampleindex i entspricht.

2.3.3. Teilagorithmus Nulldurchgangssuche

Der Teilalgorithmus zur Suche des Nulldurchgangs versucht die Sampleindizes zu berechnen, zwischen denen der erste Nulldurchgang im Resonanzbereich stattfindet. Der Zeitpunkt des Nulldurchgangs ist in Abbildung 2-6 mit t_{ns} gekennzeichnet.

Zur Berechnung dieser Sampleindizes wird nur der Resonanzbereich des Signals von Sampleindex 400 bis 800 ausgewertet.

Der Algorithmus beginnt mit der Untersuchung des ersten Samples innerhalb des Resonanzbereichs und prüft, ob der Wert des Samples kleiner oder gleich dem Wert der Baseline ist.

Ist dies der Fall, so wird ein Merker gesetzt, der anzeigt, dass das Signal unterhalb der Baseline verläuft (das Signal ist negativ). Der aktuelle Samplewert wird in einem Summenregister aufsummiert. Der Algorithmus untersucht nun das nächste Sample. Ist dieser Samplewert größer als die Baseline, so wird überprüft, ob das Summenregister einen gewissen Wert überschritten hat. Ist der Wert des Summenregisters jedoch noch unterhalb einer Schwelle, so wird das Summenregister gelöscht und der zuvor gesetzte Merker zurückgesetzt.

Dieses Verhalten entspricht einer numerischen Integration.

Die Samples werden durchlaufen, bis das Summenregister eine Integrationsschwelle überschritten hat. Ist dies der Fall, wurde ein negativer Wellenbauch des Empfangssignals detektiert.

In Abbildung 2-9 sind wichtige Eigenschaften des auszuwertenden Signals hervorgehoben.

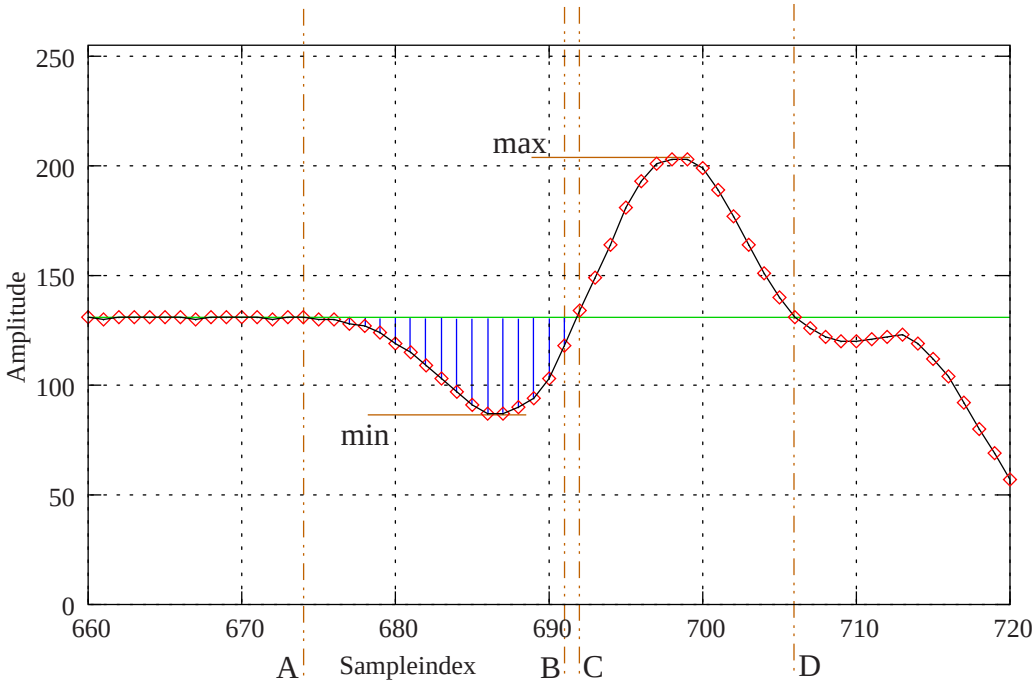


Abbildung 2-9: Empfangssignal mit hervorgehobenen Eigenschaften

Die blau schraffierte negative Fläche entspricht der negativen Summe, die dem Algorithmus als Integrationsschwelle genügt.

Wurde eine negativer Wellenbauch erkannt, so wird der Sampleindex des ersten positiven Samplewertes nach dem Wellenbauch in einer Variablen zwischengespeichert (Punkt C in Abbildung 2-9). Dieser Punkt ist gleichzeitig ein möglicher oberer Index des gesuchten Nulldurchgangs.

Nun werden die Samples solange weiter durchlaufen, bis wieder ein negativer Wert auftritt (Punkt D). Der Sampleindex des letzten positiven Werts wird ebenfalls in einer Variablen zwischengespeichert.

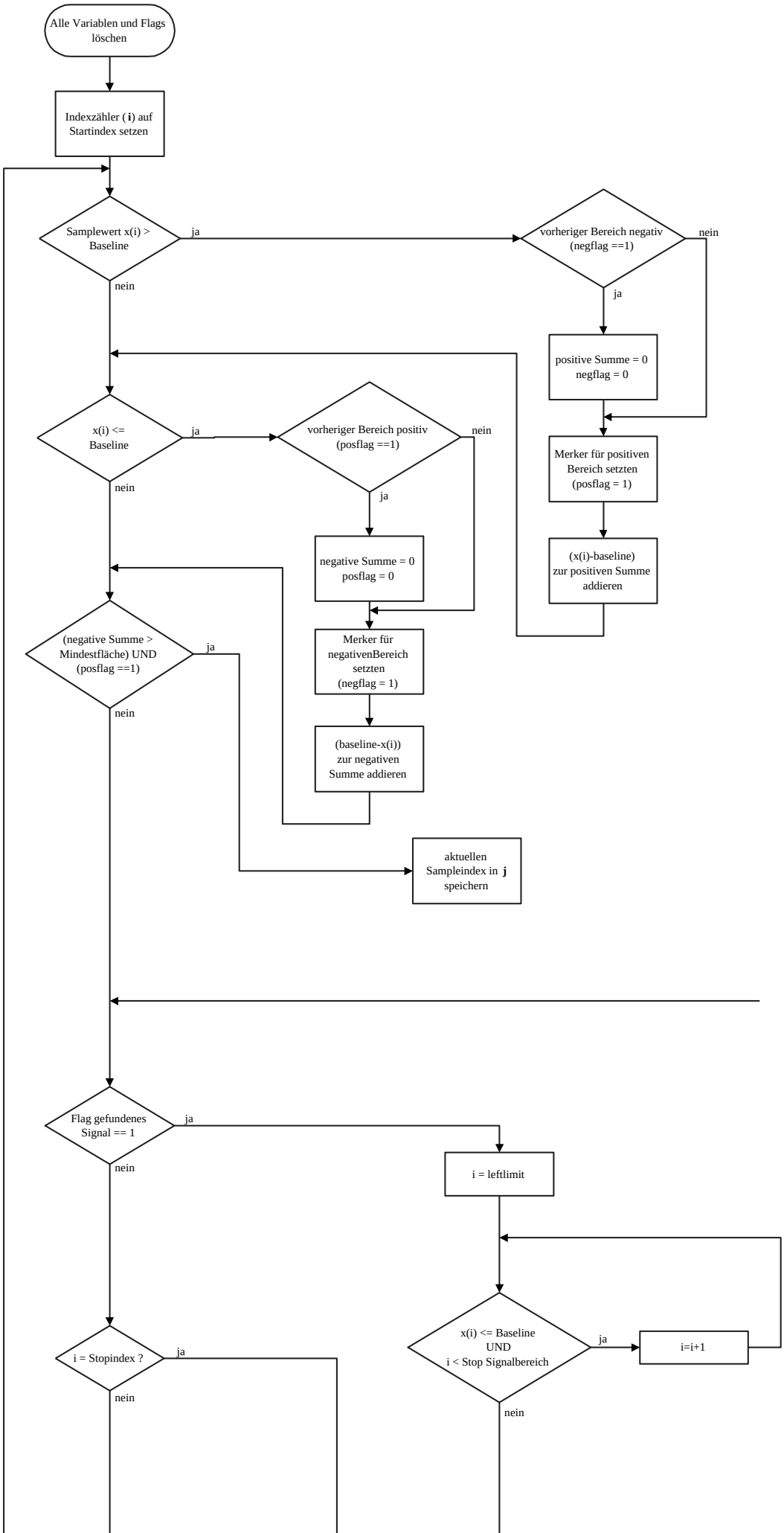
Nun läuft der Algorithmus wieder durch die Samples zurück, bis ein negativer Samplewert auftritt (Punkt B). Der Sampleindex auf dem der Punkt liegt, ist ein möglicher unterer Index des gesuchten Nulldurchgangs.

Im nächsten Schritt läuft der Algorithmus weiter zurück, bis wieder ein positiver Wert auftritt (entspricht Punkt A).

Nun wird überprüft, ob die Sampleindexabstände $B - A$ und $D - C$ ähnlich groß sind und in einem bestimmten Bereichsfenster liegen. Ist dies der Fall, wird ebenfalls überprüft, ob die maximale und minimale Amplitude des Signals (in der Abbildung mit min und max bezeichnet) ähnlich groß sind und innerhalb eines Bereichsfenster liegen.

Wurden die Kriterien erfüllt, so werden die Sampleindizes von Punkt B und C als Nulldurchgangsbereich gespeichert und dieser Teil des Algorithmus erfolgreich beendet. Wurden die Kriterien jedoch nicht erfüllt, so wird die Suche nach dem Nulldurchgang fortgesetzt.

Der Teilagorithmus zur Suche des Nulldurchgangs ist in Abbildung 2-10 als Flussdiagramm dargestellt.


Alle Variablen und Flags löschen
Indexzähler (i) auf Startindex setzen
Samplewert x(i) > Baseline
ja
nein
vorheriger Bereich negativ (negflag ==1)
nein
ja
positive Summe = 0
negflag = 0
Merker für positiven Bereich setzten (posflag = 1)
(x(i)-baseline) zur positiven Summe addieren
x(i) <= Baseline
ja
nein
vorheriger Bereich positiv (posflag ==1)
nein
ja
negative Summe = 0
posflag = 0
Merker für negativenBereich setzten (negflag = 1)
(baseline-x(i)) zur negativen Summe addieren
(negative Summe > Mindestfläche) UND (posflag ==1)
ja
nein
aktuellen Sampleindex in j speichern
Flag gefundenes Signal == 1
ja
nein
i = leftlimit
x(i) <= Baseline UND i < Stop Signalbereich
ja
i=i+1
nein
i = Stopindex ?
ja
nein

2.3.4. Teilalgorithmus Interpolation

Der Teilagorithmus zur Nulldurchgangssuche liefert zwei Werte zurück, die den Sampleindizes entsprechen, zwischen denen der Nulldurchgang stattfindet.

Der wahre Nulldurchgang kann beliebig zwischen diesen beiden Punkten liegen, die in Abbildung 2-9 als B und C bezeichnet sind.

Um den genauen Nulldurchgang zu bestimmen, ist es erforderlich, eine Interpolation um den Nulldurchgang herum durchzuführen. Somit kann auch die geforderte Genauigkeit von $\pm 200\ ps$ eingehalten werden, obwohl die Samples nur in einem zeitlichen Raster von 12.5 ns vorliegen.

Die Interpolationstiefe N bestimmt, wie viele Punkte der Kurve in die Interpolation einfließen. Bei einer Interpolationstiefe von $N = 1$, auf die sich die folgende Beschreibung des Algorithmus bezieht, wird ein Punkt überhalb und unterhalb des Nulldurchgangs zur Interpolation herangezogen.

In Abbildung 2-11 ist der Signalverlauf um den Nulldurchgang dargestellt.

Im hier behandelten Fall, in dem nur eine Interpolationstiefe von $N = 1$ aufgezeigt wird, können direkt die unteren und oberen Nulldurchgangsgrenzen als Interpolationspunkte verwendet werden (P_L und P_U). Wird jedoch eine höhere Interpolationstiefe benötigt, so müssen zunächst neue virtuelle Interpolationspunkte berechnet werden.

Hierzu ist es erforderlich, die Koordinaten der Punkte getrennt zu summieren und deren Mittelwert zu berechnen :

$$x_L = \frac{1}{N} \sum_{i=0}^{N-1} Px_{iL} \tag{2.3}$$

$$y_L = \frac{1}{N} \sum_{i=0}^{N-1} Py_{iL} \tag{2.4}$$

$$x_U = \frac{1}{N} \sum_{i=0}^{N-1} Px_{iU} \tag{2.5}$$

$$y_U = \frac{1}{N} \sum_{i=0}^{N-1} Py_{iU} \tag{2.6}$$

Somit setzt sich der interpolierte Punkt P_L aus x_L und y_L zusammen. Analog dazu setzt sich der Punkt P_U aus x_U und y_U zusammen.

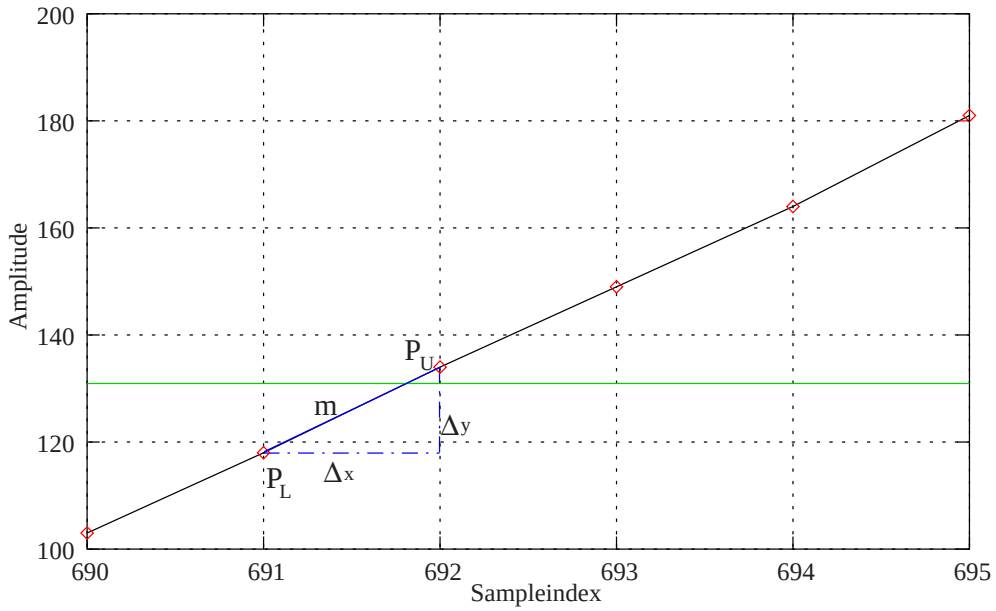


Abbildung 2-11: Signal mit Interpolationspunkten

Um den Nulldurchgang berechnen zu können, ist es notwendig, eine virtuelle Gerade zwischen die Interpolationspunkte zu legen. Die Steigung dieser Geraden wird durch folgende Formel ermittelt:

$$m = \frac{\Delta y}{\Delta x} = \frac{Py_U - Py_L}{Px_U - Px_L} \tag{2.7}$$

Der y-Achsenabschnitt dieser virtuellen Gerade entspricht dem Nulldurchgang des Ultraschallsignals.

Der y-Achsenabschnitt lässt sich durch Umstellung der Formel

$$y = mx + b \tag{2.8}$$

nach b berechnen. Als einzusetzender Punkt wird hier der interpolierte Punkt P_L gewählt.

Der interpolierte Nulldurchgang lässt sich aus folgender Gleichung ermitteln:

$$b = \frac{y}{m} + x = \frac{(Baseline - Py_L)}{m} + Px_L \tag{2.9}$$

Dieser Wert entspricht dem Sampleindex, an dem der Nulldurchgang auftritt.

Um die Laufzeit zu berechnen muss dieser Sampleindex noch mit der Samplezeit t_{Sample} multipliziert werden :

$$Laufzeit = b * t_{Sample} \tag{2.10}$$

2.3.5. Teilalgorithmus Mittelung der Laufzeit

Um ein akkurateres Ergebnis zu erhalten, wird die Laufzeit über mehrere Messzyklen gemittelt. Die Mittelung erfolgt über ca. $K = 100$ Messzyklen. Um die endgültige Laufzeit zu ermitteln, wird der Mittelwert über die einzelnen Laufzeiten berechnet:

$$Mittlere\ Laufzeit = \frac{1}{K} \sum_{i=0}^{K-1} Laufzeit(i) \tag{2.11}$$

2.4. Datenübertragung über SPI

Die Übermittlung der Ultraschall-Laufzeiten an die übergeordneten Systeme erfolgt über eine SPI-Schnittstelle.

Eine SPI-Schnittstelle stellt eine synchrone serielle Verbindung zwischen mehreren Endgeräten dar.

Eines der Endgeräte übernimmt die Rolle eines sogenannten *Masters*. Die übrigen Endgeräte werden als *Slaves* bezeichnet.

Die prinzipielle Verschaltung der SPI-Endgeräte ist in Abbildung 2-12 aufgezeigt.

Der SPI-Master erzeugt den Takt *clock*. Dieser wird parallel an alle Slaves geführt. Der Sendeausgang des Masters (tx) wird an alle Empfangs-Eingänge der Slaves (rx) gelegt. Ebenso werden alle Sendeausgänge der Slaves an den Empfangs-Eingang des Masters gelegt. Da die Sendeausgänge der Slaves parallel geschaltet sind, muss sichergestellt sein, dass nur ein Slave zur selben Zeit senden kann. Der Master wählt den Slave, mit dem er kommunizieren möchte über die *enable*-Ausgänge aus.

Die SPI-Schnittstelle der Ultraschallelektronik ist als Slave konfiguriert. Das übergeordnete System übernimmt die Rolle des Masters.

Der Kommunikationsablauf ist in Abbildung 2-13 aufgezeigt.

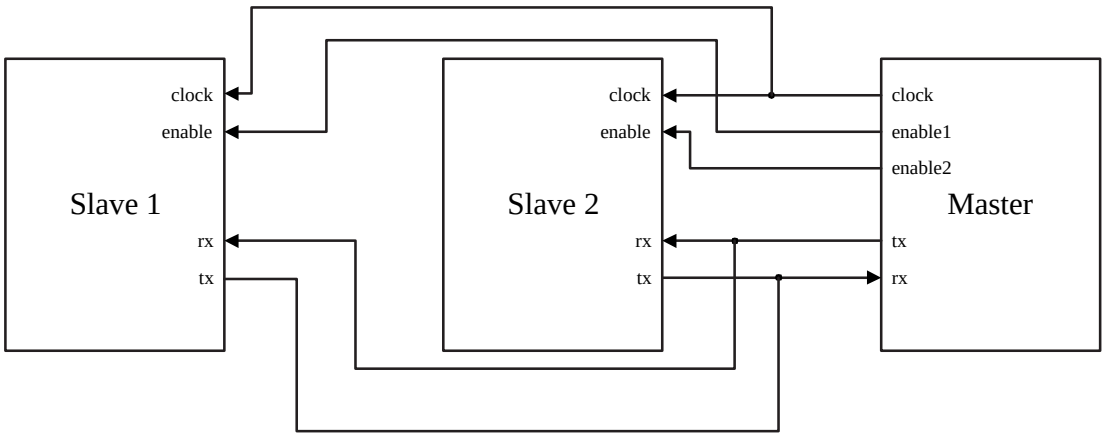


Abbildung 2-12: Verschaltung von SPI-Endgeräten

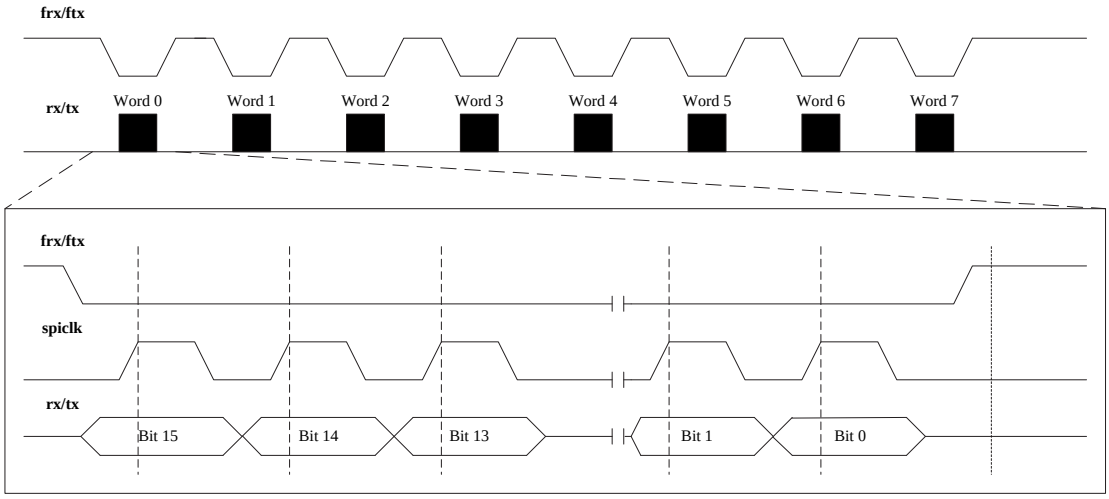


Abbildung 2-13: SPI-Kommunikationsablauf

Die Signale *ftx* und *frx* arbeiten als Enable-Signale. Die Signale sind zwar physikalisch getrennt vorhanden, führen jedoch den gleichen Pegel, daher ist es nicht erforderlich, beide Signale auszuwerten. Das Signal ist Low-Aktiv, d.h. die Ultraschallelektronik wird bei Low-Pegeln auf den SPI-Bus aufgeschaltet.

Bei jedem Zyklus des frx-Signals wird ein 16bit-Wort an das übergeordnete System übertragen. Zeitgleich sendet das übergeordnete System ebenfalls Daten an die Ultraschallelektronik.

Wird das frx-Signal auf L-Pegel gesetzt, werden die einzelnen Bits über die Empfangs- und Sendeleitungen übertragen. Die Bits werden synchron zur positiven Flanke

des Taktsignals *spiclk* übertragen und müssen daher zu diesem Zeitpunkt stabil sein. Die Wörter werden mit dem MSB zuerst übertragen.

Ein kompletter Datensatz besteht aus acht 16bit-Worten. Der Datensatz, den die Ultraschallelektronik verschickt, ist wie folgt aufgebaut:

0	9	10	15
0			
(Intensität arteriell)			
Laufzeit arteriell Vorkomma (in ns)			
Laufzeit arteriell Nachkomma		0	
0			
(Intensität venös)			
Laufzeit venös Vorkomma (in ns)			
Laufzeit venös Nachkomma		0	

Tabelle 2-3: SPI-Datensatz von der Ultraschallelektronik

Die Ultraschall-Intensitäten werden nur von der originalen Ultraschallelektronik übertragen.

Der Datensatz, den das übergeordnete System verschickt, ist nach folgendem Schema aufgebaut:

0	15
-	
-	
-	
-	
-	
-	
-	
Synchronisationswort (0xA5A5)	

Tabelle 2-4: SPI-Datensatz von den übergeordneten Systemen

Das Synchronisationswort wird von der Ultraschallelektronik genutzt, um fest-

zustellen, wann das übergeordnete System den Beginn eines neuen Datensatzes erwartet.

2.5. Einführung in die FPGA-Technologie

2.5.1. Allgemeines

FPGAs gehören zur Familie der programmierbaren logischen Schaltungen (kurz PLDs).

Im Gegensatz zur herkömmlichen Entwicklung von Digitalschaltungen durch Zusammenschalten mehrerer Standardbauelemente aus der TTL- und CMOS-Familie zu einem „Gattergrab“, wird die Funktionalität durch Programmierung bestimmt.

Die PLDs stellen hierbei eine Ansammlung von logischen Grundfunktionen auf einem IC dar, die durch Programmierung miteinander verknüpft werden.

Man klassifiziert grob zwei Typen von PLDs:

PLDs mit UND/ODER Struktur: Zu diesen PLDs gehören die klassischen, einfachen PAL und GAL-Bausteine sowie moderne, hochkomplexe und schnelle CPLD-Bausteine. Diese Typen sind jedoch nicht Gegenstand dieser Arbeit, für weitergehende Literatur zu diesem Thema wird auf [4] verwiesen.

PLDs mit Logikzellenstruktur: Zu diesen PLDs gehören die in dieser Arbeit verwendeten FPGAs. FPGAs bestehen aus Logikzellen, die matrixförmig angeordnet sind. Die Verbindung dieser Zellen untereinander erfolgt über frei programmierbare Verbindungsleitungen.

Je nach Art der Programmierung werden FPGAs in zwei Unterkategorien unterteilt:

Antifuse-FPGA: Die Programmierung der FPGAs erfolgt durch gezieltes Entfernen von Isolierungen und somit durch Herstellung von Verbindungen.

Diese Bausteine können nur einmal programmiert werden (OTP-Baustein).

SRAM-FPGA: Die Information, wie die Logik zu verknüpfen ist, wird bei Bausteinen dieses Typs in flüchtigen Speicherzellen abgelegt. Beim Ausschalten

der Versorgungsspannung geht die Information verloren, daher müssen diese Bausteine bei jedem Start des Systems neu programmiert werden.

Auf dem Markt der FPGA-ICs gibt es eine Vielzahl von Herstellern. Deren angebotene Baulemente unterscheiden sich in der Funktionalität kaum, manche Hersteller wie z.B. ATMEL haben sich jedoch auf Antifuse-FPGAs spezialisiert; andere Hersteller wie Altera oder Xilinx bieten SRAM-basierte FPGAs an. Die Kapazität der auf dem Markt erhältlichen FPGAs entspricht mehr als einer Million Gatterfunktionen.

2.5.2. Aufbau eines FPGA

Die in Abbildung 2-14 ersichtliche Skizze zeigt den inneren Aufbau eines FPGA[4].

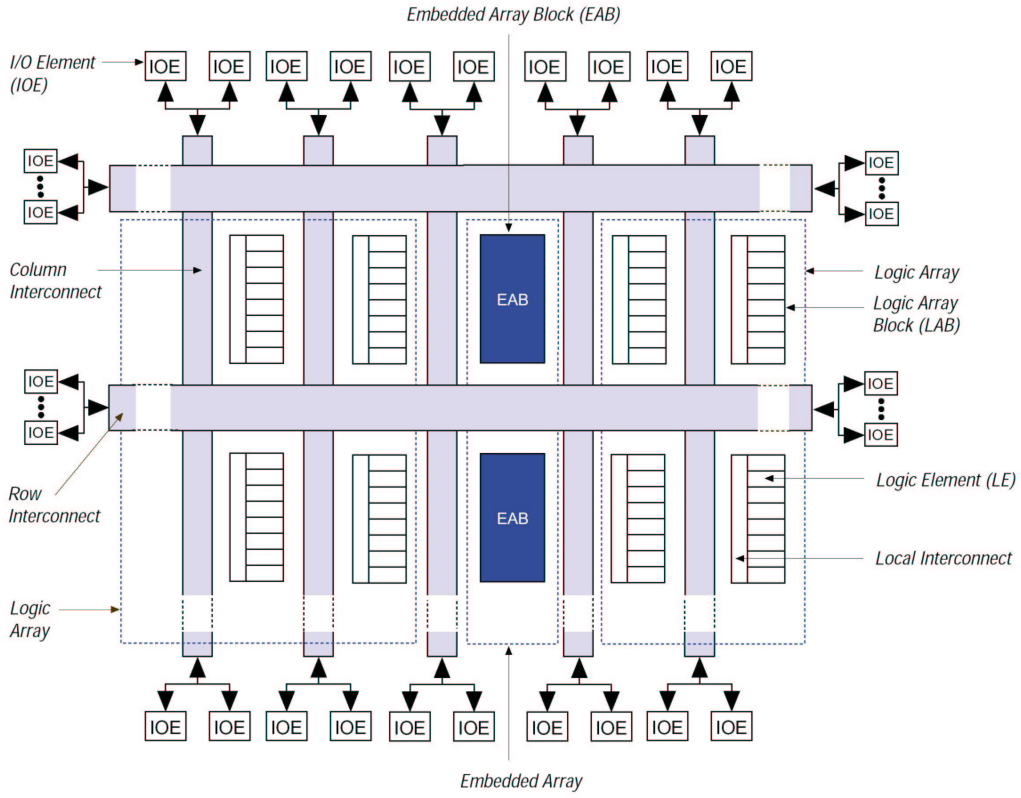


Abbildung 2-14: Struktur eines FPGA

[4]die hier wiedergegebene Struktur wurde aus [6] entnommen, der prinzipielle Aufbau ist jedoch bei anderen FPGA-Herstellern ähnlich

Das Grundelement von FPGA-Bausteinen sind kleine, programmierbare Logikzellen (LE, Logic Element), deren Funktionalität durch Programmierung festgelegt wird.

Diese Logikelemente sind gruppenweise als sogenannte Logic Array Blocks (LAB) angeordnet. Die Verbindungen innerhalb eines LAB (Local Interconnect) sind ebenfalls frei programmierbar.

Die Herstellung von Verbindungen zwischen den LABs erfolgt über zeilen- und spaltenweise Routing-Kanäle, die aus programmierbaren Verbindungspunkten aufgebaut sind. Diese Routing-Kanäle sind in der Abbildung als Row- und Column-Interconnect bezeichnet.

Die externe Ein- und Ausgabe erfolgt mittels IO-Blöcken (IOB), die ebenfalls an die Verbindungselemente angeschlossen sind, sie befinden sich am Rand des Chips. Diese IOBs können als Eingang, als Ausgang oder als bidirektionaler Anschluss programmiert werden; ebenso kann festgelegt werden, ob den Signalen ein Register vorgeschaltet wird.

Die Zellen sind innerhalb des FPGA in einer Matrix angeordnet.

Je nach Hersteller des FPGA, befindet sich eine bestimmte Menge frei nutzbarer RAM-Speicher auf dem Chip. Dieser RAM-Speicher ist in der Abbildung als EAB (Embedded Array Block) bezeichnet. Dieser Speicher hat ebenfalls Zugriff auf die Verbindungselemente.

Der Aufbau eines LE ist in Abbildung 2-15 wiedergegeben.

Ein LE verfügt über eine Lookup Table (LUT), ein programmierbares FlipFlop, eine Carry-Chain und eine Cascade Chain. Die LUT kann kombinatorische Gleichungen mit mehreren Eingängen und einem Ausgang implementieren, das FlipFLop kann als D-, T-, JK-, oder SR-FlipFlop konfiguriert werden. Als Steuereingänge besitzt das FlipFlop ein Clock-, ein Clear- und ein Preset-Signal, das entweder von speziell zugewiesenen Eingängen (dedicated inputs) oder von einer internen Logik bereitgestellt wird. Wenn das LE nur kombinatorische Logik implementieren soll, wird das FlipFlop mit dem RegisterBypass-Kanal übergangen.

Die Carry- und Cascade-Chain sind zwei schnelle Datenwege, die alle LEs in einem LAB und alle LABs innerhalb einer Spalte miteinander verketten. Sie sind dazu geeignet, arithmetische Funktionen wie z.B. Zähler und Addierer effizient zu im-

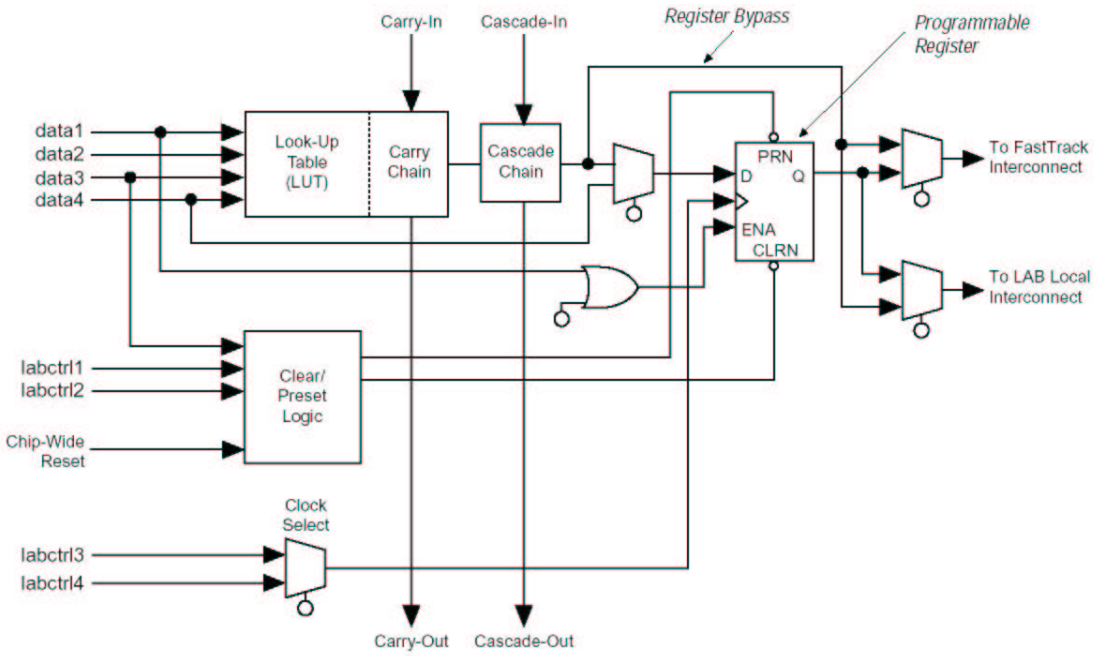


Abbildung 2-15: LE innerhalb eines FPGA

plementieren. Die LUT generiert für Zähl- und Addierfunktionen ein Carry-Signal, das direkt an das nächste LE weitergegeben werden kann. Das Carry des niederwertigeren Bits wird an die LUT des nächsthöheren Bits weitergegeben. Mit Hilfe der Cascade-Chain ist es möglich, Logikfunktionen mit vielen Eingängen effizient zu implementieren. Die benachbarten LEs können Teilausdrücke einer kombinatorischen Funktion auswerten. Die Cascade-Chain fasst diese Teilausdrücke entweder über eine UND- oder ODER-Verknüpfung zusammen.

Mehrere LEs sind zu einem LAB zusammengefasst, wie er in Abbildung 2-16 abgebildet ist.

Jeder LAB besteht aus mehreren LEs (hier 8 Stück); die Eingänge der LEs werden von der Local Interconnect Matrix gespeist. Diese Eingangssignale können entweder aus den Zeilenroutingkanälen, von speziell zugewiesenen Eingängen (wie z.B. dem Clock-Eingang) oder von den rückgekoppelten Ausgängen anderer LEs innerhalb eines LAB stammen.

Jeder LAB besitzt noch die bereits erwähnten Carry- und Cascade-Chain Eingänge, die von dem vorhergehenden LAB gespeist werden, sowie Carry- und Cascade-Chain-Ausgänge, die den nächsten LAB speisen.

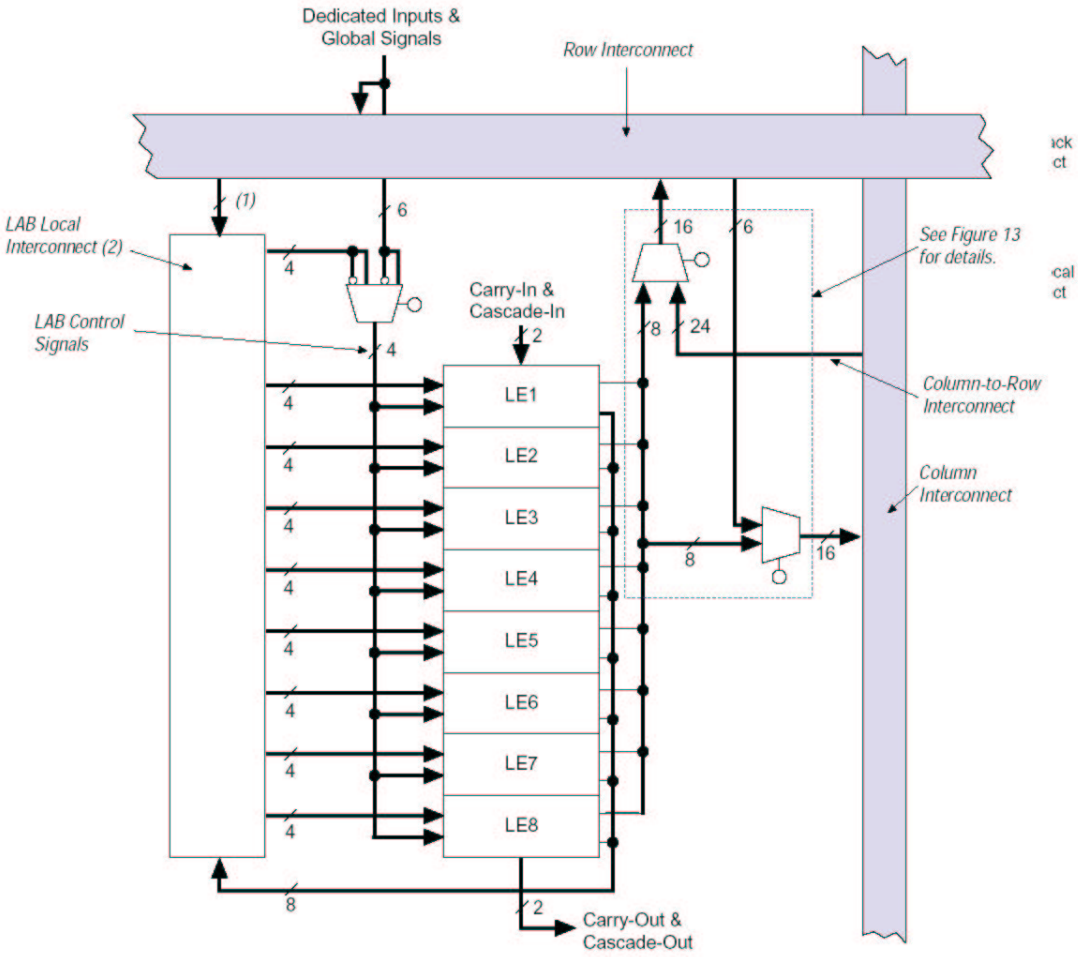


Abbildung 2-16: LAB innerhalb eines FPGA

2.5.3. Auswahl eines FPGA

Zur Auswahl eines für das Projekt geeignetem FPGA sind mehrere Kriterien zu beachten:

Anzahl der benötigten Logikelemente: Wieviele Logikelemente werden für das Design benötigt?

Größe der eingebetteten RAM-Blöcke: Wieviel kBit RAM sind nötig? Wird DualPort-RAM benötigt?

Speed-Grade: Mit welcher Taktfrequenz muss die Schaltung betrieben werden (abhängig vom Design)?

SRAM- oder OTP-Programmierung: Muss der FPGA rekonfigurierbar sein?

Bauform des IC: Wieviele IO-Pins sind nötig? Welche Bauformen sind im Layout nutzbar?

Stückkosten: Wie teuer sind die Bauelemente bei Abnahme einer bestimmten Menge?

Verfügbarkeit: Wird das Bauelement vom Hersteller noch längere Zeit produziert oder steht es kurz vor der Abkündigung?

Unterstützung durch Design-Werkzeuge: Wird der FPGA von der Design-Software und den Makrobibliotheken unterstützt oder bietet der Hersteller Design-Werkzeuge an?

Versorgungsspannung: Welche Versorgungsspannung ist nötig? Ist der FPGA TTL-kompatibel?

Die Frage, ob der FPGA SRAM- oder OTP-programmierbar ist, ließ sich für das vorliegende Projekt sehr einfach beantworten: Da das Projekt ein Prototyp ist, fiel hier die Entscheidung auf einen SRAM-basierten FPGA, denn dieser bietet die Möglichkeit, die FPGA-Software nach Korrekturen neu zu laden, ohne den Baustein austauschen zu müssen.

Als Hersteller für SRAM-basierte FPGAs kommen Altera und Xilinx in Frage, da diese Hersteller FPGAs in diversen Leistungsklassen anbieten.

Die Versorgungsspannungen sind in diesem Projekt ebenfalls unproblematisch. Es stehen 2.5 V und 3.3 V Versorgungen zur Verfügung. Diese Kombination aus Versorgungsspannungen hat sich als Quasi-Standard in der FPGA-Technologie etabliert.

Da das Projekt möglichst kostengünstig realisiert werden soll, war das Hauptaugenmerk bei der Auswahl auf die LowCost FPGA-Familien von Altera (ACEX1K) und Xilinx (Spartan II) gerichtet. Beide FPGA-Familien bieten Logikkapazitäten von 500 - 5000 Logikelementen und eingebettetes RAM in der Größe von 12 kBit bis 50 kBit.

Da schon Projekterfahrung mit der LowCost Familie von Altera vorhanden war, fiel die Entscheidung auf die *ACEX1K*-Serie des Herstellers. Altera liefert eine kostenfreie Entwicklungsumgebung aus (Quartus II). Mit dieser Entwicklungsum-

gebung ist es möglich, FPGA-Schaltungen sowohl grafisch als auch in Hardwarebeschreibungssprachen zu entwickeln.

Zur Auswahl des passenden Bausteins aus der ACEX1K-Familie spielt die Anzahl der benötigten Logikelemente eine große Rolle. Zunächst wurde daher der Algorithmus zur Laufzeitmessung untersucht und festgestellt, dass die mathematische Operation Division mehrmals Verwendung findet. Die Division ist sehr aufwendig als digitale Schaltung zu realisieren. Es wurde eine Dividierschaltung in der ALTERA-Software eingegeben, um festzustellen, welcher Logikelementverbrauch bei einem 32bit-Dividierer zu erwarten ist. Das Ergebnis von ca. 2000 Logikelementen überraschte bei einem 32bit-Dividierer nicht, daher wurde der größte Baustein aus der ACEX1K-Serie ausgewählt. Die Menge des darin zur Verfügung stehenden eingebetteten RAMs ist mehr als ausreichend.

Die Auswahl des Speed-Grades ist einerseits entscheidend für die maximale Taktfrequenz des Systems und andererseits für den Stückpreis des FPGA, denn ein FPGA mit dem höchsten SpeedGrade kostet den doppelten Stückpreis im Vergleich zu einem FPGA mit dem niedrigsten SpeedGrade. Aufgrund von Erfahrungen aus der Originalschaltung ist bekannt, dass der niedrigste SpeedGrade für den Systemtakt von 80 MHz ausreichend ist.

Die Wahl der Bauform fiel leicht, da schon die kleinste erhältliche Bauform genügend I/O-Leitungen zur Verfügung stellt. Als für das Projekt geeigneter Baustein erwies sich daher der Altera EP1K100QC208-3. Dieser Baustein bietet:

- 5000 Logikelemente
- 50 kBit eingebetteter RAM
- Speed Grade 3 (niedrigster SpeedGrade)
- 2.5 V Kernspannung / 3.3 V I/O Spannung
- 208 Pin PQFC Bauform (147 Pins als I/O nutzbar)

2.5.4. Designmethodik von FPGA-Schaltungen

Zur Entwicklung einer FPGA-Schaltung sind mehrere Schritte nötig. Diese Entwicklungsschritte (Design-Flow) sind in Abbildung 2-17 aufgeführt.

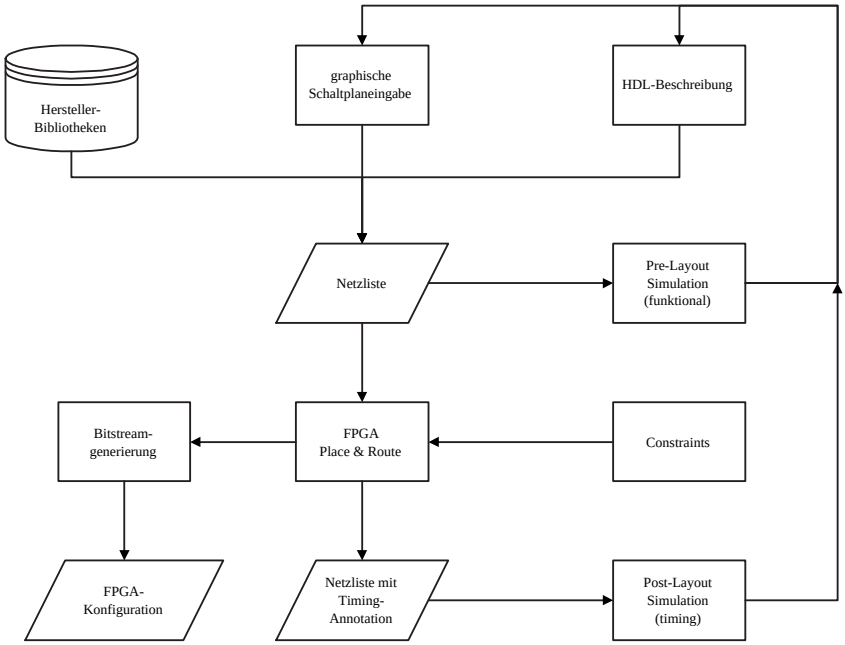


Abbildung 2-17: Design-Flow zur FPGA-Entwicklung

Zunächst erfolgt die Beschreibung der zu implementierenden Funktionalität. Die Beschreibung kann über eine graphische Schaltplaneingabe oder über eine HDL-Sprache (VHDL, Verilog, Abel) erfolgen. Auch eine gemischte Eingabe, in der Teilmodule in HDL-Beschreibung und andere Module als Schaltplan vorliegen, ist möglich. Zusätzlich können Hersteller-Bibliotheken und IP-Kerne eingebunden werden, die Funktionen auf höherer Ebene bereitstellen (z.B. Addierer, Zähler, Mikrocontroller). Die erstellten Hardwarebeschreibungen werden von einem FPGA-Compiler zusammen mit eingebundenen Hersteller-Bibliotheken und IP-Kernen verarbeitet. Der Compiler erstellt eine Netzliste der Schaltung. Eine Netzliste repräsentiert die Funktionalität der Schaltung auf Gatterebene.

Anhand der Netzliste kann bereits eine funktionale Simulation der Schaltung durchgeführt werden.

Um die Netzliste in ein konkretes FPGA zu überführen, muss der FPGA-Compiler ein so genanntes Place & Route durchführen. Durch das Place & Route werden die Funktionen aus der Netzliste auf die Logikelementstruktur des FPGA überführt (Place) und die Verdrahtung der Elemente untereinander durchgeführt (Route). Einen starken Einfluss auf das Verhalten der Place & Route Prozedur haben die Constraints. Constraints sind Vorgaben bezüglich des Timings (minimale Taktfre-

quenz) und z.B. der Belegung der I/O-Pins.

War dieser Prozess erfolgreich, so erhält man eine Netzliste, welche die konkrete Struktur des FPGA enthält. Da nun die Strukur bekannt ist, kann man Rückschlüsse auf das Timing der Schaltung ziehen (Timing-Annotation). Anhand dieser Netzliste ist es möglich, in einer Simulation auch das zeitliche Verhalten zu berücksichtigen.

Wenn die Schaltung in der Simulation nicht die geforderte Funktionalität aufweist, so kann man jederzeit den Designflow wieder von vorne beginnen.

Entsprechen die Ergebnisse der Simulation dem gewünschten Ergebniss, kann man einen Bitstream generieren, in dem die FPGA-Programmierdaten enthalten sind.

2.5.5. Hardwarebeschreibung mit VHDL

Die Entwicklung komplexer Systeme in programmierbaren Logikelementen kann mit einer reinen Schaltplaneingabe sehr mühselig und unübersichtlich werden.

Gerade die Implementierung von komplexen sequentiellen Abläufen und Zustandsautomaten sind über eine textuelle Beschreibung wesentlich einfacher zu beherrschen.

Eine Sprache, welche die textuelle Beschreibung elektronischer Systeme zur Aufgabe hat, ist VHDL[5].

Der Aufbau einer VHDL-Beschreibung ist in zwei wesentliche Teile untergliedert:

Schnittstellenbeschreibung (Entity): In dem Entity-Abschnitt werden die Ein- und Ausgänge des zu implementierenden Systems definiert.

Funktionale Beschreibung (Architecture): In dem Architecture-Abschnitt wird die Funktionalität des Systems beschrieben.

Die Schnittstellenbeschreibung fasst alle Ein- und Ausgänge des Systems in einer Sektion zusammen. Die funktionale Beschreibung ist abgekoppelt von der Schnittstellenbeschreibung. Es ist sogar möglich, dass mehrere funktionale Beschreibungen zu einer Schnittstellenbeschreibung existieren. Über eine VHDL-Anweisung

[5] für tiefergehende Informationen empfiehlt sich die Lektüre von [5]

kann dann eine der funktionalen Beschreibungen als gültige Beschreibung ausgewählt werden.

Die funktionale Beschreibung kann auf zwei Arten erfolgen:

Strukturale Beschreibung: Bei der strukturalen Beschreibung wird der Aufbau des Systems durch Verknüpfen von Unterkomponenten beschrieben. Die Unterkomponenten liegen als Bibliotheken vor oder werden ebenfalls als VHDL-Beschreibung implementiert.

Verhaltensbeschreibung In der Verhaltensbeschreibung wird direkt das Verhalten von Systemen auf Änderungen von Eingangssignalen beschrieben.

Es ist ebenfalls möglich, die beiden Beschreibungsverfahren zu mischen.

Es folgt ein einfaches Beispiel, das eine logische Verknüpfung dreier Eingänge als strukturale VHDL-Beschreibung darstellt:

```
-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY undoder IS
    PORT(
            eingang1 : IN       STD_LOGIC;
            eingang2 : IN       STD_LOGIC;
            eingang3 : IN       STD_LOGIC;
            ausgang  : OUT      STD_LOGIC
        );
END undoder;



ARCHITECTURE a OF undoder IS

COMPONENT AND2 IS
    PORT(
                in1 : IN STD_LOGIC;
                in2 : IN STD_LOGIC;
                out : OUT STD_LOGIC;
        );
END COMPONENT;

COMPONENT OR2 IS
    PORT(
                in1 : IN STD_LOGIC;
                in2 : IN STD_LOGIC;
                out : OUT STD_LOGIC;
        );
END COMPONENT;

SIGNAL zwischenergebnis : STD_LOGIC;
```

```
BEGIN

and_inst: AND2
    PORT MAP(
        in1     => eingang1,
        in2     => eingang2,
        out     => zwischenergebnis
        );

or_inst: OR2
    PORT MAP(
        in1     => eingang3,
        in2     => zwischenergebnis,
        out     => ausgang
        );


END a;
```

Zunächt werden die Herstellerbibliotheken eingebunden.

Im Entity-Abschnitt werden die Ein- und Ausgänge definiert. Als Datentyp der Ein- und Ausgänge wird der Typ STD_LOGIC verwendet. Dieser Datentyp entspricht einem Bit, dass nur die Logikpegel High und Low darstellen kann.

Im Architecture-Abschnitt werden zunächst die Schnittstellen der Unterkomponenten AND2 und OR2 deklariert. Die Komponenten selbst sind in den Herstellerbibliotheken vorhanden.

Als nächstes wird das lokale Signal *zwischenergebnis* definiert. Signaldefinitionen sind Verbindungen innerhalb eines VHDL-Moduls.

Danach beginnt die eigentliche Beschreibung des Moduls. Die Unterkomponenten AND2 und OR2 werden instantiiert und mit den Ein- und Ausgängen sowie den lokalen Signalen verdrahtet.

Das Modul verknüpft die Eingangssignale *eingang1* und *eingang2* mit einer UND-Logik. Das Ergebnis der UND-Logik wird über das lokale Signal *zwischenergebnis* mit dem Eingangssignal *eingang3* über eine ODER-Logik verknüpft. Das Ergebnis wird auf den Ausgang *ausgang* gegeben.

Im folgenden Beispiel wird die gleiche Funktionalität über eine Verhaltensbeschreibung implementiert. Zusätzlich wird das Beispiel um den Ausgang *ausgang2* erweitert, dieser Ausgang repräsentiert das Ergebnis der ODER-Verknüpfung zwischen *eingang1* und *eingang2*.

```
ARCHITECTURE a OF undoder IS

SIGNAL zwischenergebnis : STD_LOGIC;

BEGIN
 ausgang  <= (eingang1 AND eingang2) OR eingang3;
```

```
 ausgang2 <= eingang1 OR eingang2

END a;
```

Diese Beschreibung des Verhaltens des Systems ist sehr kompakt und auch leicht nachvollziehbar. Eine Besonderheit ist, dass die beiden Anweisungen für die Ausgänge unabhängig voneinander und parallel abgearbeitet werden. Bei herkömmlichen Programmiersprachen laufen zwei untereinanderstehende Anweisungen sequentiell ab. Dieser Verhaltensweise ist große Beachtung zu schenken, besonders wenn man vorher in herkömmlichen Programmiersprachen gearbeitet hat.

Um sequentielle Abläufe in VHDL zu beschreiben, ist es nötig, die Anweisungen in einen Prozess zu packen. Ein Prozess wird immer dann ausgeführt, wenn sich ein von dem Prozess überwachtes Eingangssignal verändert.

Abschliessend folgt ein Beispiel, dass die Verwendung eines Prozesses anhand eines einfachen 8bit-Zählers aufzeigt.

```
-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY zaehler IS
    PORT(
            takt      : IN       STD_LOGIC;
            ausgang   : OUT      UNSIGNED(7 DOWNTO 0);
        );
END zaehler;

ARCHITECTURE a OF zaehler IS

BEGIN

counter: PROCESS(clk)
        VARIABLE zaehlerstand : UNSIGNED(7 DOWNTO 0);
        BEGIN
        IF takt'EVENT AND takt = '1' THEN
                zaehlerstand := zaehlerstand + 1;
                ausgang <= zaehlerstand;
        END IF;

        END PROCESS;

END a;
```

Der Zähler erhält ein Taktsignal über den Eingang *takt*. Der Zählerstand wird auf den Ausgang gegeben.

Es wird ein Prozess definiert, der das Taktsignal überwacht. Innerhalb des Prozesses existiert die 8bit Variable *zaehlerstand*. Variablen werden in Registern abgelegt. Wird der Prozess durch eine Änderung des Taktsignals aktiviert, so wird über-

prüft, ob es sich um eine positive Taktflanke handelt. Danach wird der Wert in dem Register *zaehlerstand* um eins erhöht und der Wert des Zählerstand auf den Ausgang gegeben.

Danach wartet der Prozess wieder auf eine Änderung des Taktsignals.

Es ist in VHDL möglich, mehrere Prozesse zu definieren, die unabhängig voneinander ablaufen.

3. Simulation des Algorithmus zur Laufzeiterkennung

3.1. Allgemeines

Um sicherzustellen, dass der in Abschnitt 2.3 vorgestellte Algorithmus prinzipiell lauffähig ist, war es zweckmäßig, den Algorithmus zunächst auf einem PC zu implementieren und zu simulieren.

Die Prototypenimplementierung wurde mit der Rapid-Prototyping Sprache OCTAVE durchgeführt. OCTAVE ist eine zu MATLAB kompatible Entwicklungsumgebung, die frei erhältlich ist. Die Sprache ist sehr gut geeignet, um Algorithmen zur Signalverarbeitung effizient zu implementieren. Jede Variable in OCTAVE ist eine Matrix. Die angebotenen mathematischen Funktionen können direkt Matrizen verarbeiten.

Um den Algorithmus mit echten Daten zu simulieren, wurden die Ultraschallempfangssignale der originalen Ultraschallhardware aufgezeichnet. Die aufgezeichneten Daten von 100 Messzyklen liegen in einer Textdatei vor, in der die Abtastwerte des Ultraschallsignals abgelegt sind.

Die aufgezeichneten Abtastwerte können mit einem einfachen *load()*-Befehl[6] in die OCTAVE-Umgebung geladen werden. Die Daten liegen dann jedoch in einer einspaltigen Matrix vor, in deren Zeilen alle Messwerte hintereinander abgelegt sind. Damit der Algorithmus Zugriff auf die Daten der einzelnen Messzyklen hat, war es zunächst erforderlich, ein Skript zu schreiben, das die einspaltige Matrix in eine Matrix aufsplittet, in der jede Spalte einem Messzyklus entspricht.

Dieses Skript ist in Anhang B.1 aufgeführt.

[6] eine komplette Übersicht über die OCTAVE-Programmiersprache ist in [7] aufgeführt

3.2. Implementierung

Der Algorithmus ist in dem OCTAVE-Skript *usplotintra* implementiert, der Quellcode ist in Anhang B.2 wiedergegeben.

Zunächst werden einige Konstanten definiert, welche die Signalcharakteristika beschreiben (Zeile 3 - 30):

SAMPLEANZ legt die Anzahl der Abtaswerte pro Messzyklus fest

FSAMPLE gibt die Frequenz in Hz an, mit der die Abtastwerte aufgenommen wurden

BASESTART/BASEND legt fest, in welchem Signalbereich die Berechnung der Baseline erfolgen soll

SIGSTART/SIGEND legt fest, in welchem Signalbereich die Suche des Nulldurchgangs erfolgen soll

AREAMIN legt die Mindestsumme einer negativen Signalschwingung fest

WAVEMINPERIOD/WAVEMAXPERIOD legt fest, in welchem Abtastbereich eine Signalschwingung liegen darf

WAVEMINAMPL legt fest, wie groß die Schwingungsamplitude mindestens sein muss

INTERPOLCOUNT legt fest, wieviele Interpolationsschritte zur Berechnung des exakten Nulldurchgangs angewendet werden.

Das Skript gibt für jeden Messzyklus eine grafische Darstellung des Ultraschallsignals mitsamt Markierungen für den gefundenen Nulldurchgang aus. Um diese Funktionalität bei Bedarf abzuschalten, ist es möglich, die Konstante $DOPLOT$ (Zeile 32) auf den Wert 0 zu setzen.

Als nächstes wird festgelegt, dass die Plotausgabe mit Koordinatengitter gezeichnet wird und welcher Wertebereich im Plot dargestellt wird (Zeile 34 - 36).

Nachfolgend wird die Datei, welche die in Matrixform vorliegenden Ultraschalldaten enthält, in die OCTAVE-Umgebung geladen. Die Daten existieren nun in der Variablen *impulses* (Zeile 41). Die Hilfsvariablen werden gelöscht (Zeile 43 - 50).

Nun werden in einer Schleife alle Messzyklen (= Spalten der Matrix *impulses*) durchlaufen (Zeile 55 - 280). Der Zähler z zählt hierbei den aktuellen Messzyklus durch. Innerhalb der Schleife werden zunächst die Abtastwerte des aktuellen Messzyklus in die Variable x kopiert (Zeile 58). Die Anzahl der Abtastwerte in x wird ermittelt und in der Variablen *datasize* gespeichert (Zeile 61). Es wird die eindimensionale Matrix y angelegt, deren Spaltenanzahl der Anzahl der Abtastwerte entspricht. Jedes Element dieser Matrix wird mit dem Wert 1 vorbesetzt (Zeile 62). Im nächsten Schritt wird die Baseline des Ultraschallsignals gemäß Formel 2.2 mit Hilfe der in OCTAVE eingebauten Mittelwertfunktion *mean()* berechnet. Der errechnete Mittelwert wird in der Variablen *mval* gespeichert (Zeile 67).

Jedes einzelne Element der Matrix y wird nun mit dem Skalar *mval* multipliziert, so dass in jedem Element von y der Wert von *mval* steht. Dies ist nötig, damit im späteren Plot des Ultraschallsignals eine durchgängige Linie als Markierung für die Baseline eingezeichnet werden kann (Zeile 71).

Im folgenden werden die Hilfsvariablen, die zur Berechnung eines Nulldurchgangs notwendig sind, gelöscht (Zeile 74 - 82), der Nulldurchgang des Signals gemäß dem in Abschnitt 2.3.3 beschriebenen Algorithmus berechnet (Zeile 83 - 212) und die lineare Interpolation nach Abschnitt 2.3.4 (Zeile 213 - 244) durchgeführt. Das auf die Zeitbasis normierte Ergebnis der Nulldurchgangssuche eines Messzyklus wird in das Datenfeld *time* (Zeile 244 - 250) zur späteren Mittelwertbildung eingefügt.

Falls die Konstante *DOPLOT* den Wert 1 besitzt, so wird ein Plot des Ultraschallempfangssignals mitsamt Markierungen für die Grenzen des gefundenen Nulldurchgangs in einem Plotfenster ausgegeben (Zeile 262 - 282).

Nachdem alle Messzyklen durchlaufen wurden (Ende der Schleife in Zeile 285) wird zu Testzwecken die Standardabweichung sowie das Maximum und Minimum der zuvor ermittelten Ultraschalllaufzeiten berechnet (Zeile 281 - 285).

Im nächsten Schritt wird ein Mittelwert der Laufzeit-Ergebnisse von jeweils zehn Messzyklen berechnet (gemäß Formel 2.11) und in in dem Datenfeld *means* gespeichert (Zeile 294 - 307). Die Berechnung der Mittelwerte erfolgt hierbei im Gegensatz zur Algorithmusbeschreibung nicht mit 100 Messzyklen, da nicht genügend Messzyklen als Rohdaten vorliegen.

Abschließend wird die Standardabweichung der ermittelten Laufzeiten berechnet

sowie ein Histogramm zur Visualisierung ausgegeben (Zeile 309 - 312).

3.3. Ergebnisse der Simulation

Die Simulation konnte die ordnungsgemäße Funktionsfähigkeit des Algorithmus aufzeigen.

Bei einer sinnvollen Wahl der oben genannten Konstanten zur Beschreibung der Signalcharakteristika, konnte für jeden Datensatz aus den Messreihen eine gültige Laufzeit ermittelt werden.

Sinnvolle Werte für die Konstanten sind in den Konstantendefinitionen in dem OCTAVE-Skript aufgeführt.

Mit den vorliegenden Messdaten konnte eine Standardabweichung der Laufzeiten von $\pm$ 500 *ps* erreicht werden. Anzumerken ist hierbei jedoch, dass die Mittelung der Einzellaufzeiten nur über 10 Werte durchgeführt wurde.

4. Realisierung der FPGA-Software

4.1. Allgemeines

Die vorgestellte OCTAVE-Simulation des Algorithmus ist in der Lage, die aufgezeichneten Ultraschall-Empfangssignale auszuwerten und eine interpolierte Ultraschall-Laufzeit zu liefern.

In diesem Kapitel wird die Umsetzung dieses Algorithmus auf die FPGA-Architektur beschrieben. Ebenfalls wird hier die Implementierung der peripheren Teilmodule erörtert. Zu diesen Modulen gehören das SPI-Interface zur Datekommunikation, die Erzeugung der Ultraschall-Sendeimpulse sowie die Synchronisation des Ablaufs der Laufzeitmessung.

Ein weiterer Aspekt bei der Implementierung der FPGA-Software war die Untersuchung auf die Verwendbarkeit von IP-Kernen.

4.2. Auswahl geeigneter IP-Kerne

Vor der Implementierung der Teilmodule der FPGA-Software wurde untersucht, ob die Möglichkeit besteht, einen Teil der geforderten Funktionen mit Hilfe von IP-Kernen zu realisieren.

IP-Kerne sind Makromodule, die als Netzliste oder auch als vollständige VHDL- oder Verilog-Hardwarebeschreibung vorliegen. Diese Module können in ein Projekt eingebunden und dort verwendet werden. Im weiteren Sinne sind die bei den FPGA-Compiler mitgelieferten Bibliotheken ebenfalls IP-Kerne, denn sie stellen auch komplexere Funktionen wie z.B. Zähler, Addierer als Module zur Verfügung.

Ein erster Ansatz zur Verwendung von IP-Kernen wäre es, einen DSP-Kern zu finden, der den in der Originalschaltung eingesetzten TexasInstruments-DSP komplett ersetzt. Auf dem Markt ist jedoch keine solche Lösung erhältlich.

Daraufhin wurde untersucht, ob andere DSP-Kerne verwendet werden könnten, die sich nicht an die TexasInstruments-Architektur anlehnen. Es gibt eine Vielzahl von DSP-Kernen. Diese sind teilweise jedoch so komplex, dass sie nur noch in sehr großen FPGAs und ASICs synthetisiert werden können.

Die Suche nach möglichen IP-Kernen wurde auf die Angebote der Firma Altera[7] beschränkt, denn die dort angebotenen IP-Kerne sind direkt mit dem verwendeten FPGA-Compiler und den Altera Bausteinen nutzbar.

Altera bietet keine IP-Kerne an, die komplette DSPs implementieren, sondern nur Teilfunktionen, die in Systemen der digitalen Signalverarbeitung häufig zu finden sind, wie z.B. FIR- und IIR-Filter, Modulatoren und Demodulatoren. Diese Funktionen kommen jedoch in dem vorliegenden Algorithmus nicht zur Anwendung.

Ein weiterer Ansatz wäre die Verwendung eines Mikrocontroller-Kerns. Hier wird von Altera der NIOS-Prozessor angeboten. Dieser IP-Kern stellt einen kompletten Mikrocontroller bereit. Der Algorithmus könnte für diese Prozessorarchitektur in der Hochsprache C implementiert werden. Die Verwendung des Prozessors hat jedoch den Nachteil, dass die Ausführungsgeschwindigkeit sehr langsam wird und dass wiederum ein externer Speicher vorhanden sein muss, der die Software für den Prozessor enthält. Theoretisch wäre es möglich, die Software in den eingebetteten RAM-Bereich einzugliedern, da dieser Bereich auch als ROM konfigurierbar ist. Dieser Speicherbereich ist allerdings zu klein, um die Daten aus der Laufzeitmessung und die Software abzuspeichern.

Aufgrund der Komplexität des Algorithmus bot es sich an, die Funktionalität direkt in der Hardwarebeschreibungssprache VHDL zu realisieren.

4.3. Struktur der FPGA-Software

Zunächst wurde eine modulare Struktur ausgearbeitet, die es ermöglicht, die Problemstellung in Teilmodulen zu implementieren und diese Teilmodule getrennt voneinander zu simulieren und zu testen.

Eine grobe Struktur ergibt sich aus der Tatsache, dass einerseits ein algorithmischer Teil und andererseits periphere Teilaufgaben in einer bestimmten Reihenfolge zu bearbeiten sind. Daher ergibt sich die in Abbildung 4-18 aufgeführte Struktur.

Die Struktur besteht aus folgenden Teilblöcken:

Ultraschallimpulserzeugung: Die Ultraschallimpulserzeugung erzeugt periodisch den zur Anregung des Ultraschallsenders notwendigen Signalsprung und ver-

[7] siehe dazu die Webseite: http://www.altera.com/ip/ipm-index.html

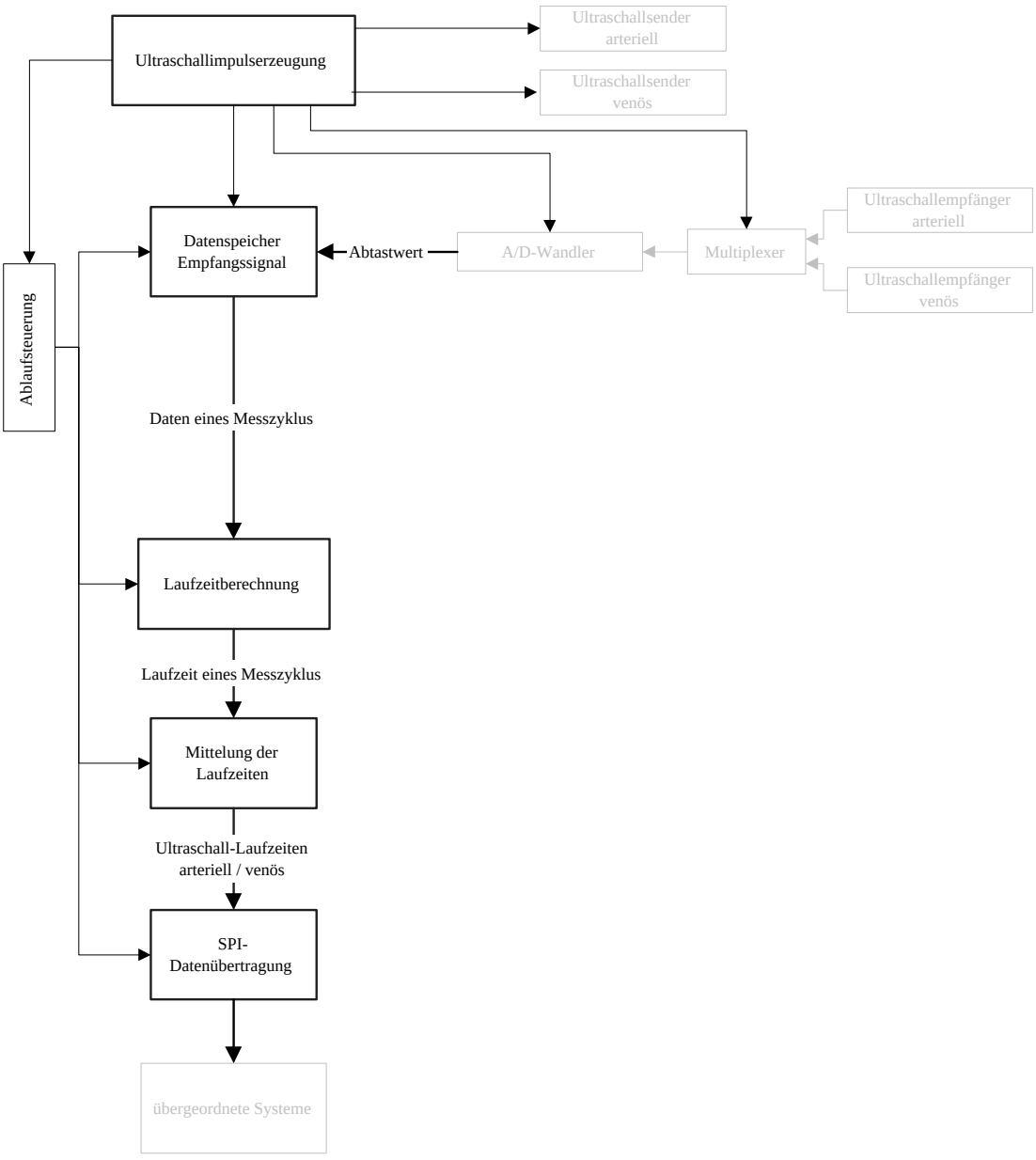


Abbildung 4-18: Struktur der FPGA-Blöcke

anlasst die A/D-Wandlersteuerung zum Einlesen des Ultraschallempfangssignals. Des weiteren wird die Ablaufsteuerung benachrichtigt, wenn ein kompletter Messzyklus in den Datenspeicher eingelesen wurde.

A/D-Wandlersteuerung: Die A/D-Wandlersteuerung schaltet den A/D-Wandler für das Abtasten des Ultraschallempfangssignals frei. Der Datenspeicher, in dem die Abtastwerte abgelegt werden, wird ebenfalls von der A/D-Wandlersteuerung für das Beschreiben des Speichers freigeschaltet.

Ablaufsteuerung: Die Ablaufsteuerung wird von der Ultraschallimpulserzeugung nach dem Einlesen eines Messzyklus benachrichtigt. Sie koordiniert den Ablauf der Signalverarbeitungskette zur Laufzeitberechnung und benachrichtigt das Modul zur SPI-Datenübertragung, wenn eine neu berechnete Ultraschall-Laufzeit bereit steht.

Datenspeicher Empfangssignal: Die Abtastwerte des Ultraschallempfangssignals werden in den Datenspeicher geschrieben. Die Signalverarbeitungskette greift lesend auf den Speicher zu.

Laufzeitberechnung: Die Laufzeitberechnung liest die Abtastwerte aus dem Datenspeicher aus und berechnet die Ultraschall-Laufzeit eines einzelnen Messzyklus.

Mittelung der Laufzeit: Die Laufzeiten aus dem Teilmodul Laufzeitberechnung werden getrennt für den arteriellen und venösen Kanal zwischengespeichert. Nachdem eine bestimmte Anzahl Messzyklen verarbeitet wurden, wird der Mittelwert für die Kanäle berechnet.

SPI-Datenübertragung: Der errechnete Mittelwert für den arteriellen und den venösen Kanal wird in einem Pufferspeicher abgelegt und kann über das SPI-Protokoll von den übergeordneten Systemen abgefragt werden.

4.3.1. Modulkonzept

Das Hauptaugenmerk bei der Entwicklung der FPGA-Softwarestruktur lag bei folgenden Punkten:

- Das Design soll synchron sein[8].
- Jedes Teilmodul soll selbstständig funktionsfähig sein.
- Jedes Teilmodul soll eine einheitliche Schnittstelle bieten, um die Teilmodule einfach zu integrieren.

Aus diesen aufgeführten Punkten erwies sich die in Abbildung 4-19 gezeigte Struktur als zweckmäßig.

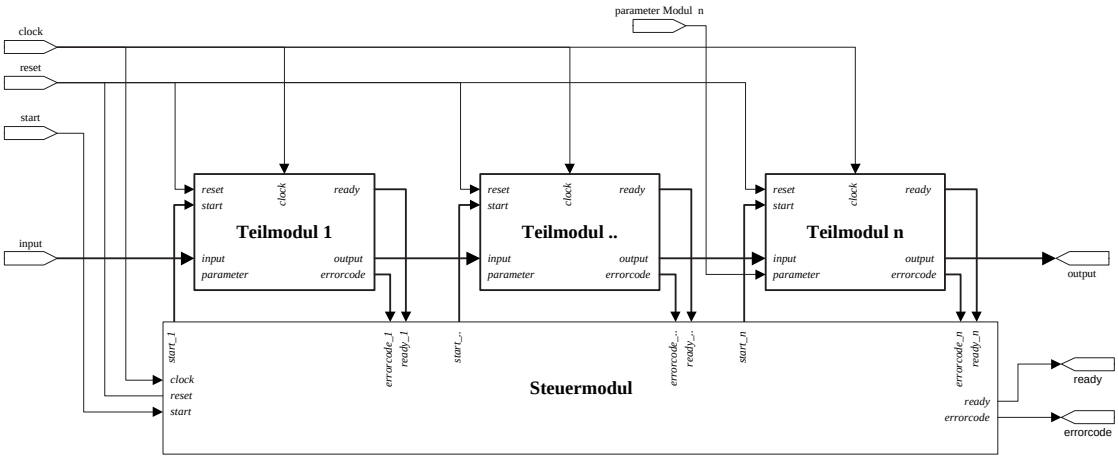


Abbildung 4-19: Struktur der Teilmodule

Jedes Teilmodul wird mit der gleichen Systemtaktfrequenz *clock* betrieben. Ebenso steht ein globales Reset-Signal zur Verfügung, das die Teilmodule asynchron zu dem Systemtakt in einen initialen Zustand versetzen kann.

Die Teilmodule besitzen zwei Schnittstellentypen:

- Steuerschnittstelle
- Datenschnittstelle

Die Steuerschnittstelle enthält Signale zur Ablaufsteuerung des Teilmoduls. Zunächst verweilt das Modul in einem Ruhezustand. Wird das Start-Signal gesetzt, so beginnt das Modul seinen Prozess abzuarbeiten. Die Beendigung des Prozesses signalisiert das Teilmodul mit dem Ready-Signal, wobei eventuell aufgetretene

[8] einige Hinweise zum Design von synchronen digitalen Schaltungen sind in [3], Kapitel 10, S. 232ff. gegeben

Fehler über ein *Errorcode*-Signal mitgeteilt werden. Das Errorcode-Signal kann von den übergeordneten Steuermodulen ausgewertet werden, um Fehlerbehandlungsmaßnahmen durchzuführen.

Die im Prozess zu bearbeitenden Daten erhält das Modul über eine *Input*-Schnittstelle, ebenso können dem Modul konstante Parameter übergeben werden. Nach Beendigung des Prozesses stehen die verarbeiteten Daten am *Output*-Signal des Moduls zur Verfügung.

Um die Verarbeitungskette zu koordinieren, wird für mehrere Teilmodule ein Steuermodul bereitgestellt. Dieses Steuermodul besitzt eine wie oben beschriebene Steuerschnittstelle. Das Steuermodul wartet auf das Start-Signal und startet danach den Prozess des ersten Teilmoduls. Danach wartet das Steuermodul auf die Beendigung des Teilmoduls. Wurde das Teilmodul beendet, so wird der Fehlercode ausgewertet und bei Erfolg das nächste Teilmodul gestartet. Nachdem das letzte Teilmodul ausgeführt wurde oder in einem der vorherigen Teilmodule ein Fehler aufgetreten ist, wird das Ready-Signal gesetzt und bei Bedarf ein Fehlercode ausgegeben. In einem Steuermodul kann eine Zeitüberschreitungsüberwachung realisiert werden, welche die Prozesskette abbricht, wenn ein Teilmodul zur Beendigung seines Prozesses längere Zeit als erwartet benötigt.

Jedes Modul ist im Kern als ein endlicher Zustandsautomat implementiert, der grundsätzlich nach dem in Abbildung 4-20 gezeigten Schema arbeitet.

Das Teilmodul verweilt zunächst in dem Zustand *state_init*. Wenn das Start-Signal gesetzt wird, so wird das Ready-Signal zurückgesetzt und die weiteren modulspezifischen Zustände abgearbeitet. Der letzte modulspezifische Zustand geht in den Zustand *state_complete* über. Dieser Zustand setzt das Ready-Signal und springt wieder in den Zustand *state_init*.

4.4. Implementierung der Module

Nachdem die beschriebene Struktur der Software festgelegt wurde, konnte die Implementierung der Teilmodule beginnen.

Als Entwicklungsmethode wurde das Bottom-Up Verfahren angewendet, d.h. es

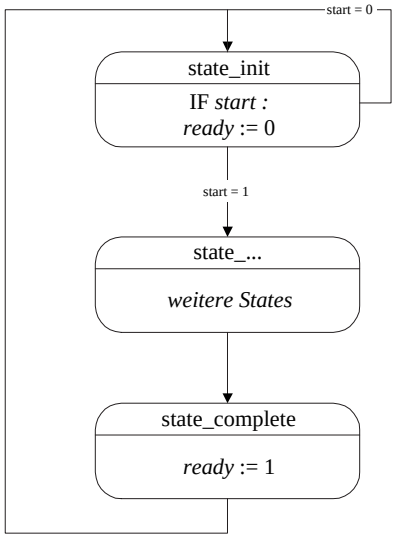


Abbildung 4-20: grundlegende Struktur des Zustandsautomaten

wurden die einzelnen Funktionsmodule entworfen und in einer Testumgebung getestet. Nach der Fertigstellung der Teilmodule wurden diese über Steuermodule miteinander verknüpft.

Zunächst wurde die gesamte Problemstellung analysiert und in einzelne Teilmodule zerlegt. Als ersten Strukturierungsschritt bot sich die in Abschnitt 4.3 ausgearbeitete grobe Struktur an. Die dort vorgestellten Module wurden noch einmal in kleinere Teilmodule aufgegliedert.

Die gesamte ausgearbeitete Modulstruktur ist in Abbildung 4-21 ersichtlich.

Eine Besonderheit dieser Struktur stellen die Teilmodule für den Datenspeicher und die ALU dar. Diese Module werden von den Teilmodulen aus dem Berechnungsmodul verwendet. Über Multiplexer werden die ALU und der Datenspeicher auf das gerade aktive Modul geschaltet.

4.4.1. Teilmodul Baselineberechnung (meanproc)

Die Entwicklung der Teilmodule begann mit dem Modul *Baselineberechnung*. Im Zuge der Entwicklung dieses Teilmoduls wurde auch die ALU implementiert, die arithmetische Funktionalitäten den Modulen zur Verfügung stellt.

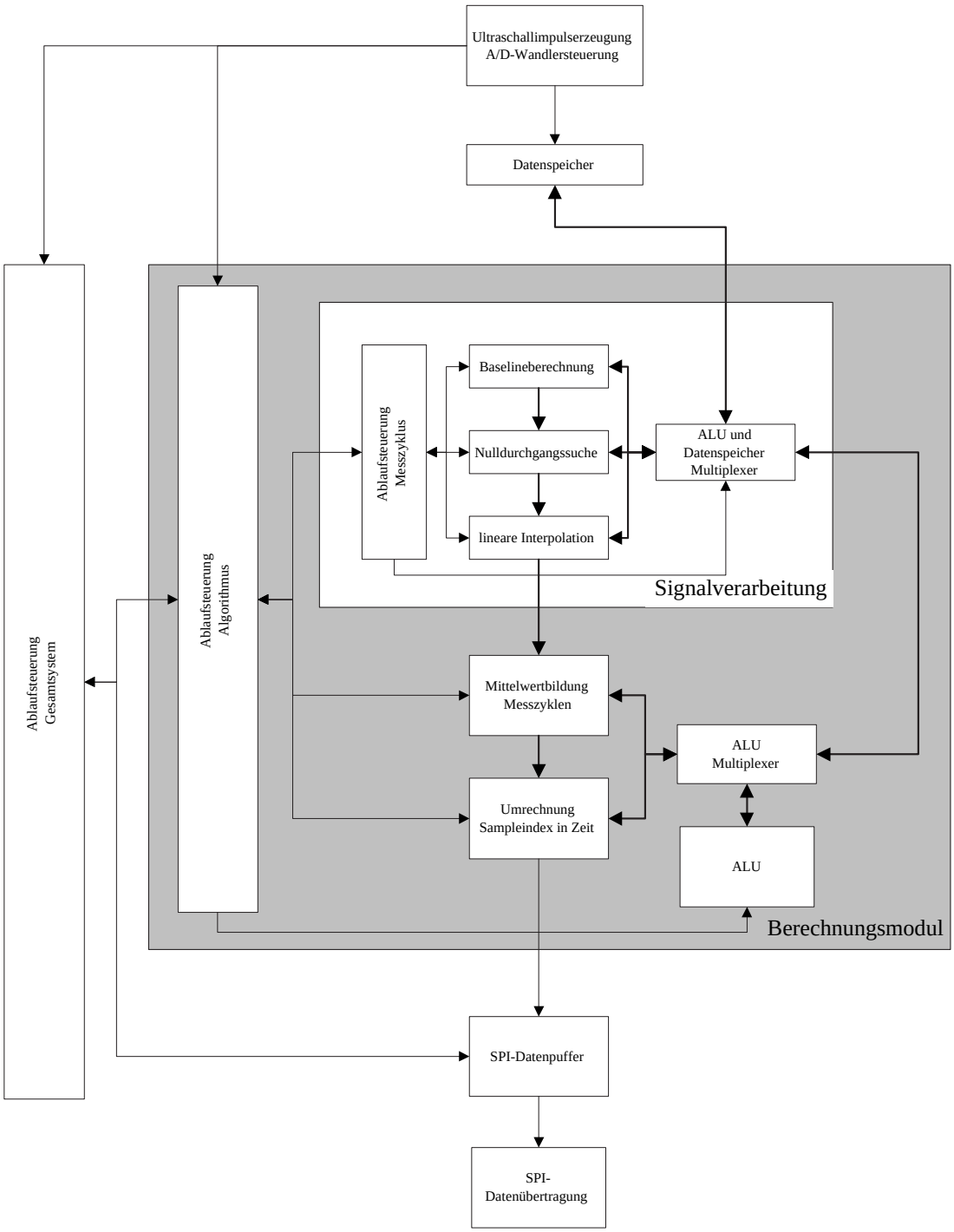


Abbildung 4-21: gesamte Modulstruktur der FPGA-Software

Das Teilmodul Baselineberechnung wurde in der Hardwarebeschreibungssprache VHDL entworfen; der Quelltext ist in Anhang B.3.1 aufgeführt.

Das Modul wird in der VHDL-Beschreibung als *meanproc* bezeichnet.

Zunächst wird die Schnittstelle des Moduls in dem Entity-Abschnitt definiert (Zeile 50 - 82).

Zusätzlich zu den Ein- und Ausgängen für die Modulsteuerung, die in Abschnitt 4.3.1 beschrieben wurden, sind hier der Adress- und Datenbus zum Zugriff auf den Sampledatenspeicher und die ALU definiert. Es werden zwei Parameter an die Baselineberechnung übergeben, welche die Sampleindizes angeben, zwischen denen die Baseline berechnet werden soll (*meanstart_i*,*meanstop_i*). Diese Parameter sind 10bit lange vorzeichenlose Ganzzahlen.

Das Modul stellt den Wert der berechneten Baseline in zwei Datenformaten bereit. Der Ausgang *mean_o* stellt den Wert der Baseline als 8bit-Ganzzahl zur Verfügung, während der Ausgang *mean16_o* die Baseline als 16bit-Festkommawert (8bit Vorkomma, 8bit Nachkomma) zur Verfügung stellt.

Die Funktionalität des Moduls ist in dem Architecture-Abschnitt der VHDL-Beschreibung aufgeführt.

Wurde das Start-Signal gesetzt, so werden die Parameter in Registern gespeichert, das Ready-Signal und der Fehlercode werden zurückgesetzt.

Im folgenden State *state_checkparam* wird überprüft, ob der Parameter für den Beginn der Baselineberechnung nicht größer als der Parameter für das Ende der Berechnung ist. Falls der Beginn größer ist, wird die Abarbeitung des Moduls mit einem Fehlercode abgebrochen.

In den folgenden States wird die Baseline gemäß dem vorgestellten Algorithmus nach Abschnitt 2.3.2 berechnet. Die arithmetischen Operationen werden hierbei mit Hilfe der ALU berechnet. Da das ALU-Modul mit Parametern vorgeladen und gestartet wird und das Ende einer ALU-Operation ausgewertet werden muss, wurde jeder arithmetische Schritt in mehreren States untergebracht. Diese Art der Ansteuerung der ALU wird hier für den ersten arithmetischen Schritt aufgezeigt.

Die weiteren arithmetischen Schritte arbeiten analog dazu.

Um den Mittelwert der Samplewerte berechnen zu können, muss zunächst die Anzahl der Samples ermittelt werden. Hierzu wird die Differenz aus den Parametern *meanstart* und *meanstop* gebildet und der Wert 1 addiert.

Im State *state_prepsamplediff* werden die ALU-Operandenregister mit den Operanden *meanstart* und *meanstop* vorgeladen. Ebenso wird das ALU-Opcoderegister mit dem auszuführenden Operator vorgeladen.

Danach wird im State *state_samplediff* das ALU-Modul gestartet.

Im State *state_ssamplediff* wird das Start-Signal wieder zurückgesetzt.

Im State *state_postsamplediff* wird solange gewartet, bis das ALU-Modul seine Ausführung beendet hat und sein Ready-Signal setzt. Die Ergebnisse aus den ALU-Ergebnisregistern werden in die lokalen Register transferiert (hier samplediff).

In den folgenden Schritten wird zu der Sampledifferenz der Wert 1 addiert, um die Anzahl der zu mittelnden Werte zu bestimmen (Zeile 229 - 257); danach wird die Mittelwertbildung ausgeführt (Zeile 259 - 337).

Die abschließende Division der Mittelwertbildung liefert einerseits den 8bit langen ganzzahligen Anteil des Mittelwerts und andererseits einen 8bit langen Restanteil zurück.

Der 8bit lange ganzzahlige Wert der Baseline wird direkt an den Ausgang übertragen (Zeile 396). Zur Durchführung weiterer Berechnungen wird jedoch ein genauerer Wert der Baseline benötigt. Dieser Wert wird als 16bit-Festkommazahl dargestellt. Die oberen 8bit entsprechen hierbei dem ganzzahligen Anteil.

Um den Nachkommaanteil zu ermitteln, wird der 8bit Restanteil zunächst auf 16bit erweitert (Zeile 398 - 400). Dieser 16bit Wert wird durch die Anzahl der Samples dividiert (Zeile 339 - 374). Das Ergebnis dieser Division entspricht nun einem 8bit Wert, der dem Nachkommaanteil des Baselinewertes repräsentiert. Der Ganzzahl- und der Nachkommaanteil werden zusammengeführt auf den Ausgang gelegt (Zeile 402 - 406).

Das Modul setzt sein Ready-Signal und springt wieder in den State *state_init*, um auf das Start-Signal zu warten.

4.4.2. Teilmodul ALU (alucore)

Dieses Modul führt festgelegte arithmetische Funktionen aus. Der Quelltext ist in Anhang B.3.2 aufgeführt.

Um eine ausreichend große Genauigkeit der arithmetischen Operationen zu erzielen, wurde die Wortbreite der Parameter auf 24bit festgelegt.

Die Operanden werden über die Parameter *regop1_i* und *regop2_i* übergeben. Der Operator wird über den Parameter *opcode_i* ausgewählt. Die ALU wäre also theoretisch in der Lage 2^8 verschiedene Operationen auf die Parameter durchzuführen.

Die ALU hat die Möglichkeit, den Inhalt zweier Ergebnisregister auf die Ausgänge *rege1_o* und *rege2_o* zu legen. Zusätzlich kann die ALU noch Fehlercodes auf den Ausgang *regstat_o* ausgeben.

In der vorliegenden Implementierung wurden drei Operatoren vorgesehen:

Operatoren der ALU				
Opcode	Operation	Ergebnis 1	Ergebnis 2	Fehlercode
01	Addition	$op1 + op2$	-	-
02	Subtraktion	$op1 - op2$	-	-
04	Division	$op1/op2$	$op1\ mod\ op2$	0001=Divison durch 0

Tabelle 4-5: Operatoren der ALU

Zunächst befindet sich die ALU im State *state_init* und wartet auf das Start-Signal. Wurde das Start-Signal gesetzt, so werden die Parameter in lokalen Registern zwischengespeichert und der Fehlercode- und Ready-Ausgang zurückgesetzt (Zeile 136 - 161).

Im folgenden State *state_decodeop* wird der auszuführende Operator dekodiert und die entsprechende Aktion durchgeführt (Zeile 164 - 191). Die Operatoren Addition und Subtraktion werden innerhalb dieses States direkt ausgeführt und das Ergebnis in den Ergebnisregistern gespeichert. Ist der Opcode nicht bekannt, so wird ein Fehlercode gesetzt und die ALU-Berechnung beendet. Für den Operator Division sind weitere Schritte erforderlich. Die Division ist nicht direkt in der VHDL-Beschreibung zugänglich. Daher wurde aus der Herstellerbibliothek das Dividiermodul *lpm_divider* eingebunden (Zeile 250 - 258). Dieses Dividiermodul

arbeitet nach dem Prinzip des *Pipelining*. Durch das Pipelining ist die Rechenzeit des Dividiermoduls vorhersagbar. Der Dividierer wurde auf eine Pipelinetiefe von elf Takten eingestellt, d.h. das Ergebnis der Divison ist nach elf Takten am *clock*-Eingang verfügbar.

Wenn die Divison als Operator ausgeführt wird, wird im State *state_decodeop* zunächst überprüft, ob der Divisor *regop2* den Wert 0 hat. Ist dies der Fall, wird die ALU-Operation mit einem Fehlercode abgebrochen. Ist der Divisor nicht 0, so wird die Division gestartet. Eine Zählschleife bestehend aus den States *state_divide* und *state_dividewait* wartet die elf Takte ab und speichert danach die Ergebnisse der Division in den Ergebnisregistern.

Nachdem die ALU-Operation beendet wurde, wird das Ready-Signal gesetzt und die ALU springt wieder in den State *state_init*.

4.4.3. Teilmodul Nulldurchgangssuche (findsig)

Der Teilalgorithmus zur Nulldurchgangssuche (siehe Abschnitt 2.3.3) wurde in das Modul *findsig* implementiert. Der Quelltext ist in Abschnitt B.3.3 aufgeführt.

Das Modul hat Zugriff auf den Adress- und Datenbus des Sampledatenspeichers und auf das ALU-Modul.

Der derzeitige Stand des Moduls nutzt momentan nicht die ALU-Funktionalitäten, da dieses Modul sehr viele Berechnungen in kurzer Zeit durchführen muss. Die ALU benötigt jedoch zum Starten und Ausdekodieren des Operators selbst sehr viel Zeit.

Der Algorithmus ist auf den ganzzahligen Wert der Baseline angewiesen. Dieser Wert wird von dem Teilmodul Baselineberechnung an den Eingang *mean_i* gelegt.

Das Modul erhält noch zusätzliche Parameter, die den Ablauf des Algorithmus beeinflussen. Diese Parameter stimmen mit den in Abschnitt 2.3.3 besprochenen Parametern überein.

Nach der Beendigung der Berechnung stellt das Modul zwei 10bit lange Werte zur Verfügung, die den unteren (*lower_o*) und oberen (*upper_o*) Sampleindizes entsprechen, zwischen denen der Nulldurchgang auftritt.

Wurde das Start-Signal gesetzt, so werden die Hilfsregister gelöscht, das Ready-Signal und der Fehlercode zurückgesetzt und die Parameter des Moduls in lokalen Registern gespeichert.

Im State *state_checkparam* wird überprüft, ob der Parameter *findsigstart* größer als der Parameter *findsigstop* ist. Ist dies der Fall, so wird der Ablauf des Moduls mit der Ausgabe eines Fehlercodes beendet.

War die Überprüfung der Parameter erfolgreich, beginnt der Ablauf des Algorithmus (Zeile 233 - 464). Die Implementierung unterscheidet sich hierbei nicht von der OCTAVE-Implementierung.

Wurde das Ende des Algorithmus erreicht, so wird überprüft, ob in dem Bereich der Nulldurchgangssuche ein Nulldurchgang gefunden wurde (Zeile 470). Wurde kein Nulldurchgang gefunden, so beendet sich das Modul mit einem Fehlercode.

Das Ready-Signal wird gesetzt und die Sampleindizes (bzw. der Wert 0, wenn kein Nulldurchgang gefunden wurde) werden auf die Ausgänge gelegt.

Die Laufzeit dieses Moduls ist abhängig von der Form des Ultraschallempfangssignals. Daher ist es notwendig, in das übergeordnete Steuermodul eine Überwachung der Laufzeit des Moduls einzufügen um die Messzykluszeit von $500\mu s$ nicht zu überschreiten. Wurde diese Messzykluszeit überschritten, so muß das Steuermodul dafür sorgen, dass eine Fehlerbehandlung durchgeführt wird.

4.4.4. Teilmodul Interpolation (interpolate)

Das Teilmodul Interpolation implementiert den in Abschnitt 2.3.4 beschriebenen Algorithmus zur linearen Interpolation des Nulldurchgangs. Der VHDL-Quelltext ist in Abschnitt B.3.4 aufgeführt.

Das Modul hat Zugriff auf den Adress- und Datenbus des Sampledatenspeichers und auf das ALU-Modul. Die Parameter für das Modul sind die 10bit Sampleindizes, zwischen denen der Nulldruchgang auftritt (*lower_i* und *upper_i*), der 16bit Festkommawert der Baseline (*mean16_i*) und der 4bit Wert *polcount_i*, der die Anzahl der Interpolationsschritte bestimmt.

Das Modul liefert den interpolierten Sampleindex, an dem der Nulldurchgang auftritt, in Form eines Festkommawertes mit 20bit Länge zurück. Die oberen 10bit dieses Wertes entsprechen dem ganzzahligen Anteil, während die unteren 10bit dem Nachkommaanteil des Sampleindex entsprechen.

Auch in diesem Modul werden nach Setzen des Start-Signals die Hilfsregister gelöscht, das Ready-Signal und der Fehlercode zurückgesetzt und die Parameter in lokale Register übertragen.

Zunächst werden je nach Interpolationstiefe die Komponenten der zu interpolierenden Punkte zur späteren Mittelung aufsummiert und gespeichert (Zeile 252 - 372).

Die aufsummierten Punkte werden durch die Interpolationstiefe dividiert (Zeile 375 - 433). Die vollständige Mittelung mit Hilfe der Division durch die Interpolationstiefe ist jedoch nur bei den Komponenten des unteren Punktes nötig, da der obere Punkt nur in die Berechnung der Steigungsgerade einfließt und somit nur das Verhältnis der x- und y-Komponente des oberen Punktes von Bedeutung ist. Liefert die ALU bei der Divison einen Fehlercode zurück, so wird die Abarbeitung des Moduls mit einem Fehlercode abgebrochen.

In den folgenden Schritten werden die Differenzen zwischen den x- und y-Komponenten der virtuellen Punkte gebildet und in den Registern *dval* und *dindex* gespeichert (Zeile 436 - 482).

Die Steigung zwischen den virtuellen Punkten wird durch Division von *dval* durch *dindex* berechnet. Die errechnete Steigung wird in dem Register *dm* gespeichert (Zeile 484 - 512). Tritt bei der Division ein Fehler auf, so wird die Abarbeitung des Moduls ebenfalls mit einem Fehlercode abgebrochen.

Im folgenden wird der y-Achsenabschnitt der zwischen den virtuellen Punkten konstruierten Geraden berechnet. Hierzu wird die y-Komponente des unteren virtuellen Punktes von dem Wert der Baseline subtrahiert und das Ergebnis in dem Register *deltamin* gespeichert (Zeile 515 - 539).

Nun wird durch Division des y-Achsenabschnitts *deltamin* und der Steigung *dm* der Nulldurchgang dieser Geraden bestimmt und in dem Register *dns* zwischengespeichert (Zeile 541 - 569).

Dieser errechnete Nulldurchgang bezieht sich jedoch nur auf den unteren virtuellen Punkt. Um den absoluten Nulldurchgang in Bezug auf den Sampleindex 0 zu errechnen, ist es noch erforderlich, die x-Komponente des unteren virtuellen Punktes auf das Ergebnis aufzuaddieren (Zeile 571 - 594). Das endgültige Ergebnis wird in dem Register *ustime* gespeichert.

Nach der Berechnung wird das Ready-Signal gesetzt. Das Modul kehrt wieder in den initialen Zustand zurück und wartet auf das Start-Signal.

4.4.5. Integration der Teilmodule Signalverarbeitung

Die beschriebenen Teilmodule

- *meanproc*
- *findsig*
- *interpolate*

wurden mitsamt dem Steuermodul *sigcontroller* zu einem Gesamtmodul *sigproc* zusammengefasst.

Die Struktur des Moduls *sigproc* ist in Abbildung 4-22 aufgeführt. Der Quelltext wird hier nicht aufgeführt, da er, abgesehen von den ALU- und Datenspeichermultiplexern, nur aus Signalverdrahtungen der Teilmodule besteht.

Die Teilmodule sind in einer Kette angeordnet. Das von *meanproc* bereitgestellte Ergebnis wird als Parameter an das Teilmodul *findsig* weitergegeben. Das Ergebnis von *findsig* wird wiederum als Parameter von *interpolate* weiterverwendet.

Das Ergebnis des Teilmoduls *interpolate* entspricht dem Ergebnisausgang des Gesamtmoduls *sigproc*.

Die Parameter des Gesamtmoduls *sigproc* werden an die entsprechenden Teilmodule weitergeleitet.

Den Zugriff auf die von allen Teilmodulen verwendeten Module ALU und Sampledatenspeicher werden von den Multiplexern *alumux* und *datmux* geregelt. Das

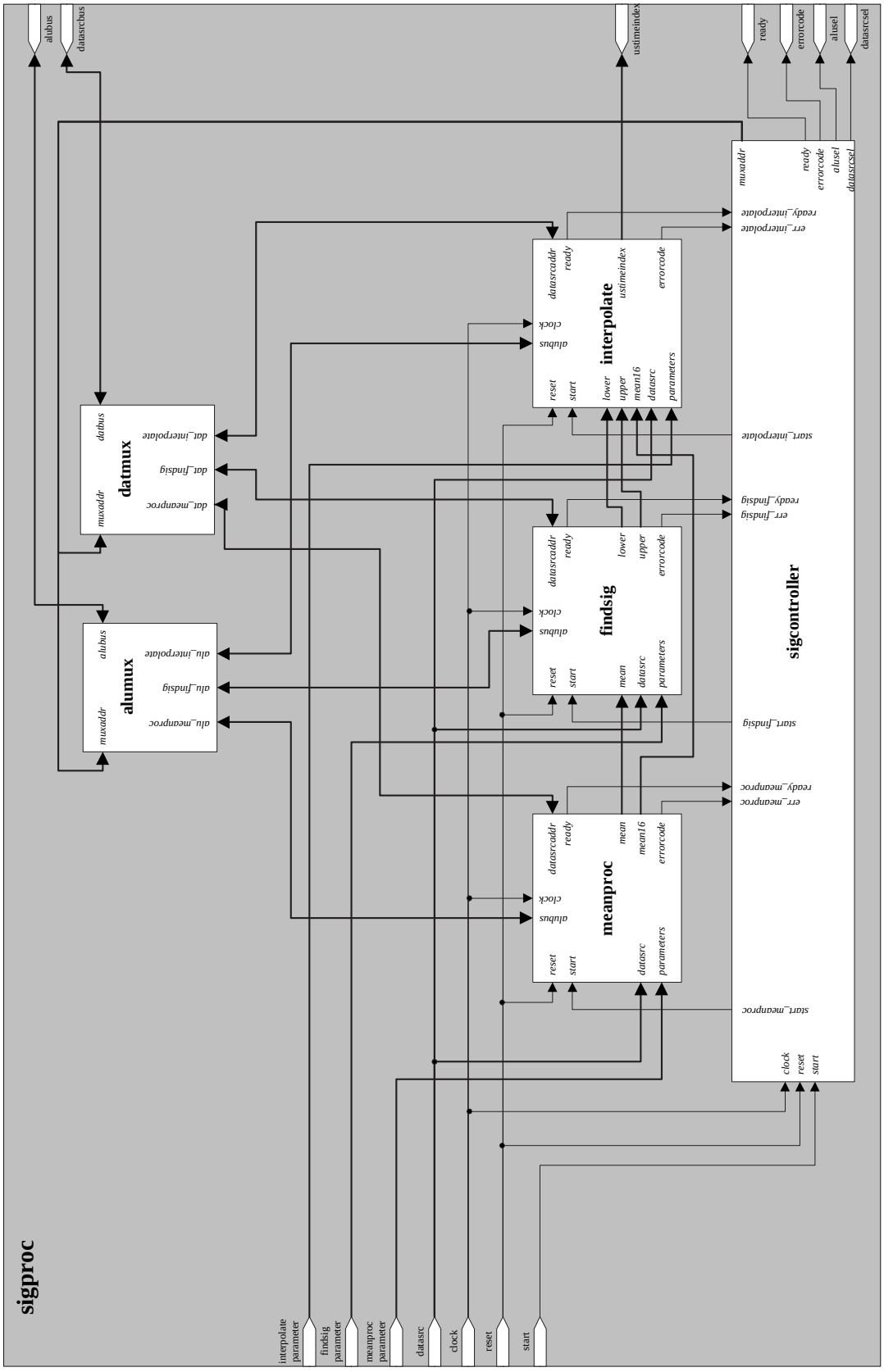


Abbildung 4-22: Struktur des Moduls Signalverarbeitung

Modul *sigcontroller* schaltet über diese Multiplexer die ALU und den Sampledatenspeicher auf das Teilmodul, das gerade die Berechnung durchführt.

Die Koordination der Verarbeitungskette wird von dem Modul *sigcontroller* übernommen. Der Quelltext des Moduls ist in Anhang B.3.5 aufgeführt.

Das Start-Signal von *sigproc* wird an das Modul sigcontroller weitergeleitet.

Ist das Start-Signal gesetzt, so werden alle Hilfsregister sowie das Ready-Signal und der Fehlercode zurückgesetzt.

Zunächst werden die ALU- und Datenspeichermultiplexer auf das Modul *meanproc* geschaltet und das Modul gestartet. Es wird solange gewartet, bis *meanproc* die Beendigung seiner Rechnenoperationen durch Setzen seines Ready-Signals mitteilt. Trat bei der Berechnung ein Fehler auf, so bricht *sigcontroller* die Verarbeitungskette ab und setzt einen entsprechenden Fehlercode (Zeile 164 - 194).

Verlief die Verarbeitung von *meanproc* fehlerfrei, wird der ALU- und Datenspeichermultiplexer auf das Modul *findsig* geschaltet und dieses Modul gestartet. Im gleichen Schritt wird der Zähler *timeoutcount* gestartet.

Nun wartet *sigcontroller* auf die Beendigung des Moduls *findsig*.

Da die Laufzeit des Moduls *findsig* abhängig von der Signalform des Ultraschall-Empfangssignals ist, wird in jedem Taktzyklus der Warteschleife der Zählerstand von *timeoutcount* überprüft. Hat *timeoutcount* einen bestimmten Wert überschritten, so wird die weitere Verarbeitung abgebrochen und ein Fehlercode gesetzt.

Konnte das Modul *findsig* seine Berechnungen innerhalb des gültigen Zeitfensters ausführen, so wird wiederum überprüft, ob das Modul einen Fehlercode zurückgeliefert hat. In diesem Falle bricht *sigcontroller* wiederum die Verarbeitungskette ab (Zeile 196 - 232).

Konnte *findsig* fehlerfrei ausgeführt werden, so werden die Multiplexer auf das Modul *interpolate* geschaltet und dieses Modul gestartet. Wurde die Ausführung des Moduls *interpolate* fehlerhaft beendet, so bricht auch hier *sigcontroller* die Verarbeitungskette ab und setzt einen entsprechenden Fehlercode.

Wurde *interpolate* fehlerfrei ausgeführt, so setzt *sigcontroller* keinen Fehlercode und kehrt in den Wartezustand *state_init* zurück.

4.4.6. Teilmodul Mittelwertbildung (meanshots)

Das Teilmodul zur Mittelwertbildung *meanshots* berechnet den Mittelwert aus mehreren von dem Modul *sigproc* gelieferten Nulldurchgängen. Hier wird auch die Unterscheidung getroffen, ob es sich um ein Messzyklus aus dem arteriellen oder dem venösen Kanal handelt.

Die Implementierung des Moduls wurde in VHDL durchgeführt. Der Quelltext ist in Anhang B.3.6 aufgeführt.

Das Modul erwartet als Parameter den von *sigproc* berechneten Nulldurchgang als 20bit Festkommawert. Über den Parameter *meancount_i* wird festgelegt, wie viele Messzyklen gemittelt werden. Die Anzahl der zu mittelnden Messzyklen errechnet sich folgendermaßen:

$$Anzahl\ Messzyklen = 2^{meancount\_i}$$

Um unterscheiden zu können, von welchem Kanal der aktuelle Messzyklus kommt, wird der Parameter *artvenflag_i* verwendet. Dieses Signal wird von der Ultraschallimpulserzeugung generiert.

Als Ergebnis werden die gemittelten Sampleindizes des Nullstellendurchgangs als 20bit Festkommawert getrennt für den arteriellen und den venösen Kanal ausgegeben.

Wird das Start-Signal gegeben, so werden die Ready-Signale für das Modul und für einen einzelnen Messzyklus zurückgesetzt. Ebenfalls wird der Fehlercode des Moduls zurückgesetzt und die Summenregister gelöscht. Auch die Zähler*meanstepart* und *meanstepven* werden zurückgesetzt. Diese Zähler geben an, wie viele Messzyklen für den jeweiligen Kanal bereits in die Mittelung aufgenommen wurden.

Anhand des Parameters *meancount_i* wird gemäß der oben genannten Formel die Anzahl der zu mittelnden Messzyklen bestimmt und in dem Register *sumendval* gespeichert (Zeile 170 - 212).

Nun wartet das Modul im State *state_waitsample* auf das *Samplestart*-Signal. Dieses Signal wird von dem übergeordneten Steuermodul gesetzt, wenn die Laufzeit eines Messzyklus fertig berechnet wurde.

Wurde das Samplestart-Signal gesetzt, so werden die Parameter in lokalen Registern gespeichert.

Im State *state_chkerr* wird überprüft, ob bei der Berechnung der Ultraschall-Laufzeit ein Fehler aufgetreten ist. Hierzu wird das Signal *errsignal* ausgewertet. Dieses Signal ist eine logische ODER-Verknüpfung über alle Bits des Fehlercode-Wortes aus dem Modul *sigproc* und führt somit immer dann High-Pegel, wenn ein Fehler auftrat (Zeile 112). Wurde die Ultraschall-Laufzeit fehlerhaft berechnet, so wird die aktuelle Laufzeit auf den Wert 0 gesetzt.

Im nächsten State *state_sumup* wird abhängig von dem gerade zu bearbeitenden Kanal das Ergebnis der Ultraschall-Laufzeitmessung in das Register *timeartsum*(arterieller Kanal) oder *timevensum* (venöser Kanal) aufsummiert. Wurde die Anzahl der zu mittelnden Werte für einen Kanal schon überschritten, so wird für diesen Kanal kein Wert aufsummiert.

Im folgenden State *state_checkend* wird überprüft, ob für den arteriellen und den venösen Kanal die Anzahl der zu mittelnden Messzyklen erreicht wurde. Wurde noch nicht die notwendige Anzahl erreicht, so wird der State *state_readysample* angesprungen. In diesem State wird der Ausgang *readysample_o* gesetzt und benachrichtigt somit das übergeordnete Steuermodul, dass ein Laufzeitwert erfolgreich zwischengespeichert wurde. Danach beginnt der Zyklus des Wartens auf einen neuen Laufzeitwert wieder von vorne, indem in den State *state_waitsample* gesprungen wird.

Wurde die notwendige Anzahl der Laufzeiten erreicht, werden in den aufeinanderfolgenden States *state_divideart* und *state_divideven* die aufsummierten Laufzeiten durch die Anzahl der zu mittelnden Messzyklen dividiert. Diese Divison wird durch Schieben der Bits in den Summenregistern um den Parameter *meancount* erreicht.

Danach geht das Modul in den State *state_complete* über und setzt sein Ready-Signal. Hiernach wartet das Modul in dem initialen State *state_init* auf das Start-Signal.

Die Mittelung der Laufzeiten erfolgt in der vorliegenden Implementierung über

64 einzelne Laufzeiten. Jeder Kanal liefert eine neue Laufzeit im Abstand von einer Millisekunde. Da die Mittelung durch Schieben von Bits realisiert ist, kann der Mittelwert nur über 2^x Werte gebildet werden. In den Anforderungen ist eine minimale Aktualisierungsrate der Laufzeiten von $100ms$ gefordert. Eine Mittelung über 128 Werte würde diese Anforderung verletzen, die nächst kleinere Anzahl der Mittelungen beträgt 64. Daher wurde dieser Wert als Anzahl der Mittelungen gewählt.

4.4.7. Teilmodul Laufzeitumrechnung (index2time)

Die bisherigen Berechnungen berücksichtigen nicht die Abtastzeit, in der die Samples vorliegen. Die Ergebnisse beziehen sich auf den Sampleindex, an dem der Nulldurchgang auftritt. Um die endgültige Ultraschall-Laufzeit zu errechnen, muss der Sampleindex mit der Abtastzeit multipliziert werden.

Das Modul *index2time* rechnet die Sampleindizes in einen Zeitwert um. Als Parameter nimmt das Modul die gemittelten arteriellen (*timeindexart_i*) und venösen (*timeindexven_i*) Sampleindizes entgegen. Zur Umrechnung wird der Parameter *timeconst_i* benötigt. Dieser Parameter bestimmt die Zeitkonstante, mit der die Sampleindizes multipliziert werden. Das Format der Zeitkonstante ist ein Festkommawert mit 6bit Länge, wovon 5bit dem ganzzahligen Anteil und 1bit dem Nachkommaanteil entsprechen. In den weiteren Verarbeitungseinheiten wird der ganzzahlige Anteil als Zeitwert mit der Basis 1 ns interpretiert. Es ist also möglich, die Sampleindizes mit Zeitkonstanten von 0 ns bis $31,5$ ns in $0,5$ ns Schritten umzurechnen.

Das Modul stellt die errechneten arteriellen und venösen Laufzeiten aufgeteilt in einen 16bit ganzzahligen Anteil und einen 10bit Nachkommaanteil zur Verfügung.

Das Modul besitzt Schnittstellen zur Verwendung der ALU. Diese werden in dem aktuellen Softwarestand jedoch nicht genutzt, da die Worbreite der ALU für die hier genutzten Operationen nicht ausreicht.

Das Modul wartet zunächst im State *state_init* auf das Start-Signal. Wurde das Start-Signal gegeben, so wird das Ready-Signal zurückgesetzt und die temporären Summenregister gelöscht. Danach werden im State *state_reg* die Parameter in lo-

kalen Registern zwischengespeichert und der Zähler *bitcount* mit der Anzahl der Vorkommastellen von *timeconst* vorgeladen.

Danach wird im State *state_shiftloop* überprüft, ob der Zähler *bitcount* den Wert 0 erreicht hat. Hat der Zähler noch nicht 0 erreicht, so wird das Register *bcount* mit dem um den Wert 1 dekrementierten Zählerstand geladen und in den State *state_shiftleft* gesprungen.

In dem State *state_shiftleft* wird überprüft, ob die Stelle von *timeconst*, auf die der Zähler zeigt, eine logische 1 enthält. Ist dies der Fall, so werden die Werte der Sampleindizes um *bcount* Bits nach links geschoben .Dies entspricht einer Multiplikation mit 2^{bcount}. Das Ergebnis der Multiplikation wird in den Registern *artsum* und *vensum* aufsummiert. In dem State *state_checkloop* wird überprüft, ob der Bitzähler am Ende angelangt ist. Sofern dies zutrifft, springt das Modul in den State *state_complete* und setzt sein Ready-Signal. Ist der Zähler noch nicht am Ende angelangt, wird der Zähler um den Wert 1 dekrementiert und springt wieder in den State *state_shiftloop*.

Diese Operation entspricht der schrittweisen Multiplikation der Vorkommastellen von *timeconst* mit den Sampleindizes.

Hat der Bitzähler *bitcount* im State *state_shiftloop* den Wert 0 erreicht, so wird der State *state_shiftright* angesprungen.

In diesem State wird überprüft, ob das Nachkommabit von *timeconst* gesetzt ist. Ist dies der Fall, so werden die Sampleindizes um 1bit nach rechts geschoben und auf die Summenregister addiert. Danach wird der State *state_checkloop* angesprungen.

Diese Operation entspricht der Multiplikation der Sampleindizes mit dem Faktor 0,5.

4.4.8. Integration der Teilmodule Berechnung

Die oben beschriebenen Teilmodule

- *sigproc*
- *meanshots*

- *index2time*

wurden zu einem Berechnungsmodul *calculationmain* zusammengefasst. Das Teilmodul *alucore* wurde ebenfalls in dieses Modul integriert.

Die Koordination der Teilmodule übernimmt das Steuermodul *sigsequencecontroller*.

Auch hier wurde auf eine Darstellung des Quelltextes verzichtet, da es sich um die reine Verknüpfung der Elemente handelt. Eine Strukturskizze des Gesamtmoduls ist in Abbildung 4-23 aufgeführt.

Auch in diesem Modul sind die Teilmodule wiederum in einer Verarbeitungskette angeordnet. Die Ergebnisse des Teilmoduls *sigproc* werden an das Teilmodul *meanshots* als Eingangswerte weitergegeben. Die Ergebnisse aus *meanshots* werden wiederum an das Teilmodul *index2time* weitergeleitet. Das Ergebnis von *index2time* entspricht dem Gesamtergebnis der Berechnung und wird als Ausgang des Gesamtmoduls *calculationmain* an die nächsthöhere Modulschicht weitergegeben.

Das Modul *sigproc* hat als einziges Modul in der Verarbeitungskette Zugriff auf den Sampledatenspeicher. Daher ist der Adress- und Datenbus des Sampledatenspeichers direkt auf dieses Modul geführt.

Damit jedes Teilmodul auf die ALU zugreifen kann, wird das Teilmodul *alucore* über den Multiplexer *alumux* auf das gerade arbeitende Teilmodul geschaltet.

Die Teilmodule benötigen jeweils eigene konstante Parameter. Diese Parameter sind in dem Modul *algocfg* definiert (siehe dazu den Quelltext in Anhang B.3.9). Die Ausgänge dieses Moduls sind an die entsprechenden Parametereingänge der Teilmodule geführt.

Neben den allgemeinen Steuersignalen werden die Signale *newsample* und *artven* von dem Berechnungsmodul verwendet. Das Signal *newsample* wird von der Ultraschallimpulserzeugung gesetzt, wenn neue Abtastwerte vorliegen. Ob es sich dabei um die Daten des arteriellen oder venösen Kanals handelt, wird über das Signal *artven* angezeigt.

Da in dem Teilmodul *meanshots* die Mittelung der Laufzeiten getrennt nach arteriellem und venösem Kanal durchgeführt wird, benötigt dieses Modul die Infor-

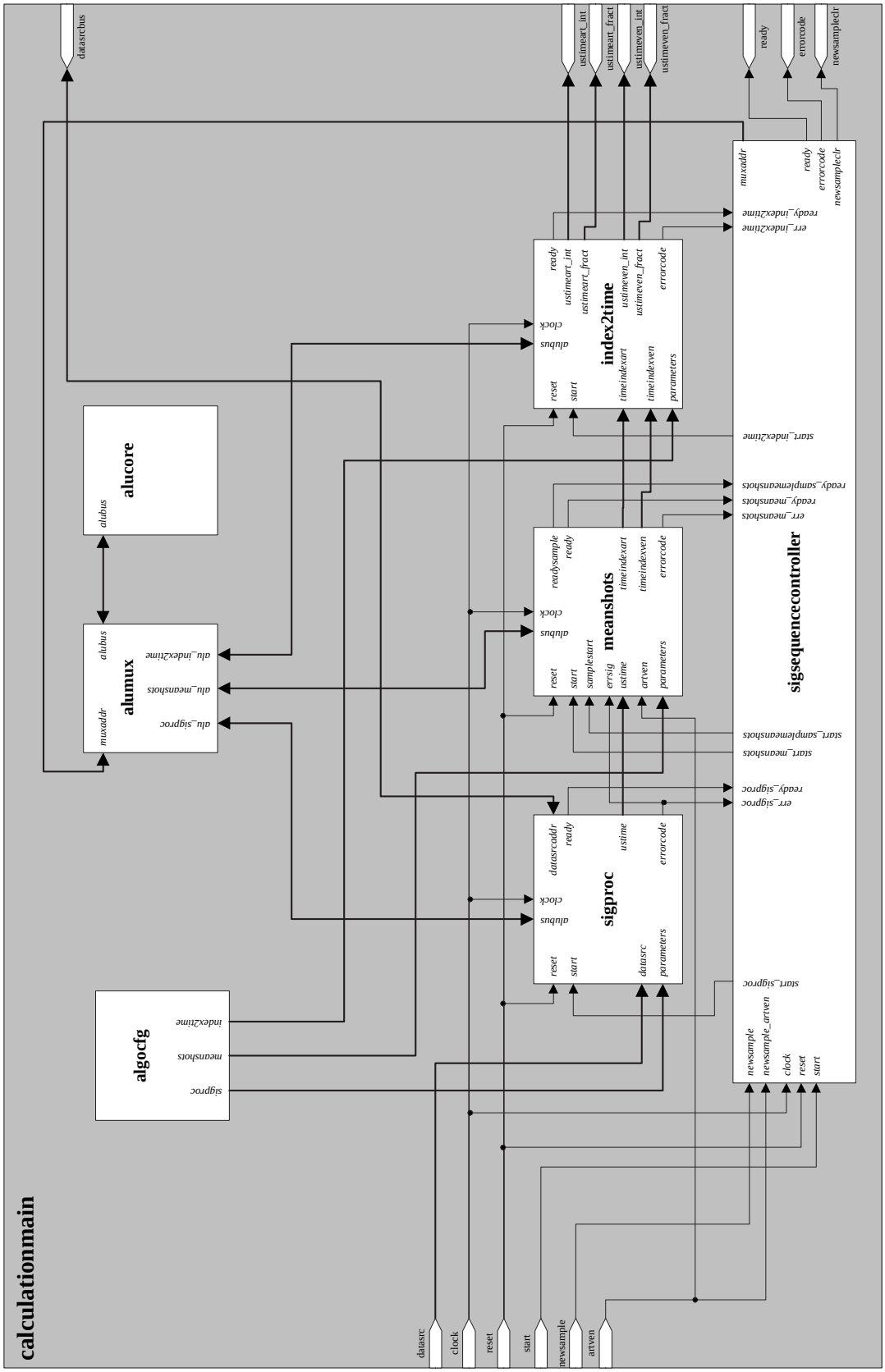


Abbildung 4-23: Struktur des Moduls *calculationmain*

mationen aus dem Signal *artven*. Daher wird dieses Signal direkt in das Modul *meanshots* geführt.

Die Verarbeitungskette wird von dem Steuermodul *sigsequencecontroller* koordiniert. Der Quelltext des Steuermoduls ist in Anhang B.3.8 aufgeführt.

Bekommt das Steuermodul das Start-Signal, so werden alle Start-Signale für die Teilmodule sowie der Fehlercode zurückgesetzt. Da die ALU nur aus dem Modul *sigproc* verwendet wird, wird der Multiplexer bei der Initialisierung einmalig auf dieses Teilmodul geschaltet.

Zunächst wird das Teilmodul *meanshots* gestartet und solange gewartet, bis neue Ultraschall-Empfangsdaten vorliegen (Zeile 159 - 177).

Sind neue Daten vorhanden, so wird das Modul *sigproc* gestartet und so lange gewartet, bis die Ausführung des Moduls beendet wurde (Zeile 179 - 197).

Der Fehlercode des Moduls *sigproc* wird gespeichert und auf den Fehlercodeausgang des Gesamtmoduls ausgegeben. Danach wird über das Signal *start_meansample* das Teilmodul *meanshots* angewiesen, den aus dem Modul *sigproc* berechneten Sampleindex zu bearbeiten (Zeile 199 - 210).

Es wird solange gewartet, bis das Modul *meanshots* das Signal *readymeansample* setzt. Wurde das Signal gesetzt, so wird überprüft, ob das Modul *meanshots* genügend Messzyklen verarbeitet hat, um einen Mittelwert zu bilden. Wurden genügend Messzyklen verarbeitet, setzt *meanshots* ebenfalls das Signal *readymean*. Sollte dies nicht der Fall sein, so wird wieder in den State *state_waitforsignal* zurückgesprungen und wieder auf neue Ultraschall-Empfangsdaten gewartet (Zeile 212 - 227).

Wenn das Signal *readymean* gesetzt wurde, wird das Modul *index2time* gestartet und auf das Ende der Berechnungen dieses Moduls gewartet (Zeile 229 - 247). Nach dem Ende der Berechnungen durch *index2time* wird das Ready-Signal des Gesamtmoduls gesetzt und das Modul springt wieder in den State *state_init* und wartet auf sein Start-Signal.

4.4.9. Teilmodul Ultraschallimpulserzeugung

Die Aufgabe der Utraschallimpulserzeugung ist es, die Sendeimpulse im Zeitraster von 500 μs zu generieren, den A/D-Wandler auf den arteriellen oder den venösen Kanal umzuschalten und die Daten aus dem A/D-Wandler in den Sampledatenspeicher zu schreiben.

Die Ultraschallimpulserzeugung wurde bereits in gleicher Form in dem FPGA der Originalschaltung implementiert und getestet.

Der Schaltplan des Moduls wurde in die zwei Teilmodule

- *shotcontroller* (siehe Anhang B.3.10)
- *shotsync* (siehe Anhang B.3.11)

aufgeteilt.

Das Teilmodul *shotsync* sorgt dafür, dass der 80 MHz-Systemtakt und die Signale für den arteriellen und venösen Sendeimpuls über gleiche Signalwege geführt werden. Die Logikfunktionen wurden in der gleichen Logikzelle innerhalb des FPGA platziert. Somit ergibt sich für die Signale ein annähernd gleiches Laufzeitverhalten.

Der durch die Logik des Moduls *shotsync* geleitete Systemtakt wird auf einen Ausgangspin des FPGA gelegt und dem A/D-Wandler als Taktsignal zugeführt. Zusätzlich wird der Takt wieder über einen Eingangspin dem FPGA zugeführt. Mit diesem Taktsignal wird auch das Modul *shotcontroller* getaktet.

Innerhalb des Moduls *shotcontroller* wird der 80 MHz-Takt mit Hilfe des T-FlipFlops *inst* auf 40 MHz heruntergetaktet. Dieser 40 MHz Takt betreibt den 16bit Zähler *inst21*. Dieser Zähler ist als Modulo-Zähler ausgeführt und beginnt bei dem Zählerstand 40100 wieder von vorne. Über den globalen Reset-Eingang kann der Zähler asynchron zurückgesetzt werden.

Der Ausgang dieses Zählers ist an die Komparatoren *inst6* bis *inst11* angeschlossen. Diese Komparatoren vergleichen den aktuellen Zählerstand mit einem Vergleichswert. Der Zählerstand entspricht dabei verschiedenen Zeitpunkten.

Zum Zeitpunkt $t = 0\mu s$ (bzw. $t = 1ms$ nach dem ersten Einschalten) spricht der Komparator *inst6* an und erzeugt den Sendeimpuls für den arteriellen Ultraschallsender. Der Sendeimpuls wird durch das D-FlipFlop *inst22* auf Low-Pegel gehalten. *inst6* sorgt über das ODER-Glied *inst15* dafür, dass die FlipFlops *inst16* und *inst17* durchschalten und den 10bit-Zähler *inst24* starten. Dieser Zähler generiert direkt die Adressdaten für den Sampledatenspeicher und läuft mit dem Systemtakt von 80 MHz. Das 80 MHz-Taktsignal wird über das NICHT-Glied *inst14* invertiert und steuert die Schreibfreigabe für den Sampledatenspeicher an.

Nach $t = 10\mu s$ spricht der Komparator *inst10* an und setzt über die ODER-Verknüpfung *inst30* die FlipFlops *inst16* und *inst17* zurück, die wiederum den Adresszähler *inst24* stoppen. Ebenso wird das FlipFlop *inst29* gesetzt. Dieses FlipFlop benachrichtigt über das Signal *newsample* die übrigen Teilmodule, dass ein neues Ultraschall-Empfangssignal aufgenommen wurde. Zurückgesetzt wird dieses FlipFlop von dem Modul *calculationmain*.

Nach $t = 30\mu s$ wird über den Komparator *inst7* das D-FlipFlop *inst22* und somit auch der Sendeimpuls zurückgesetzt. Ebenfalls wird der Adresszähler *inst24* zurückgesetzt und das T-FlipFlop *inst3* umgeschaltet. Dieses FlipFlop generiert das Signal *artvensel*. Dieses Signal wird über einen Ausgangspin des FPGA herausgeführt und steuert direkt den Eingangsmultiplexer des A/D-Wandlers an. Das Modul *calculationmain* nutzt ebenfalls dieses Signal um zu bestimmen, welcher Kanal ausgelesen wurde.

Der Ablauf ab $t = 500\mu s$ ist ähnlich zu dem vorgestellten Ablauf, weswegen an dieser Stelle nicht näher darauf eingegangen wird.

4.4.10. Teilmodul Sampledatenspeicher (samplemem)

Der Sampledatenspeicher wurde aus der Herstellerbibliothek eingebunden. Es wurde das RAM-Objekt *lpm_ram* verwendet. Dieses Modul wurde auf eine Wortbreite von 16bit und eine Größe von 1024 Worten parametrisiert. Der Speicher arbeitet asynchron, daher werden die Takteingänge *rdclk* und *wrclk* nicht verwendet.

4.4.11. Teilmodul SPI-Datenübertragung (spi)

Das Modul *spi* implementiert die Schnittstelle zwischen dem SPI-Datenpuffer *dprbuf* und der physikalischen SPI-Schnittstelle.

Das Modul wurde in VHDL implementiert. Der Quelltext ist im Anhang B.3.12 aufgeführt.

Der Ablauf der SPI-Kommunikation verläuft asynchron zur Ultraschall-Laufzeitberechnung, daher verfügt das Modul nicht über die beschriebene Zustandsautomatenstruktur.

Nach außen hin stellt sich das Modul als ein acht 16bit-Worte großer Sende- und Empfangspufferspeicher dar.

Da in der vorliegenden Implementierung keine Daten außer dem Synchronisationswort aus dem SPI-Interface in die Ultraschallelektronik eingelesen werden, wird nur der Sendepuffer genutzt.

Der Sende- und Empfangspuffer besteht aus zwei RAM-Blöcken, die von der Bibliotheksfunktion *lpm_ram* abgeleitet wurden. Der Speicher wurde auf eine Größe von acht 16bit Worten parametrisiert. Der Speicher arbeitet synchron.

Die Puffer für die beiden Datenrichtungen wurden in dem Modul *dprbuf* zusammengefasst und in das Modul *spi* als Komponente eingebunden.

Die von den übergeordneten Systemen empfangenen Daten werden in das 16bit-Schieberegister *spirxshift* mit jeder positiven Taktflanke des SPI-Taktsignals hineingeschoben (Zeile 191 - 199). Das Schieberegister ist jedoch nur aktiviert, wenn das Enable-Signal *frx* Low-Pegel führt. Ein Löschen des Inhalts des Schieberegisters ist über das Signal *reset_spirxshift* möglich.

Der Inhalt des Schieberegisters wird auf das 16bit-Register *spirxreg* geführt, das bei einer positiven Flanke des Enable-Signals den Wert aus dem Schieberegister übernimmt. Zu dem Zeitpunkt der positiven Flanke von *frx* wurde ein komplettes Datenwort eingelesen.

Die zu sendenden Daten werden aus dem Sendepuffer bei jeder negativen Flanke des *ftx*-Signals in das Sendeschieberegister *spitxshift* übernommen. Die Daten werden mit jeder positiven Flanke der SPI Taktleitung seriell aus dem Schieberegister transferiert.

Der Wortzähler *spiwordcounter* zählt mit jeder positiven Flanke des *frx*-Signals einen Schritt weiter und adressiert damit das Datenwort im Sendepuffer, das als nächstes übertragen wird (Zeile 279 - 287).

Da die SPI-Datenübertragung nicht sicherstellt, wann das übergeordnete System Datenpakete erwartet, wird das Synchronisationswort des übergeordneten Systems ausgenutzt, um den Wortzähler zurückzusetzen. Hierzu wird der Ausgang des Empfangsregisters auf das Datenwort 0xA5A5 überprüft (Zeile 237). Wurde das Datenwort 0xA5A5 empfangen, wird das Signal *reset_spirxshift* gesetzt. Dieses Signal setzt den Wortzähler und die Schieberegister zurück.

4.4.12. Teilmodul SPI-Datenpuffer (spicntrl)

Das Teilmodul SPI-Datenpuffer schreibt die ermittelten Ultraschall-Laufzeiten in den SPI-Datenpuffer hinein.

Das im vorigen Abschnitt beschriebene Modul *spi* wurde eingebunden und die Steuer- und Datenleitungen der SPI-Schnittstelle herausgeführt.

Der Quelltext des Moduls ist in Anhang B.3.13 aufgeführt.

Wird die Abarbeitung des Moduls durch das Start-Signal im State *state_init* gestartet, so werden zunächst das Ready-Signal und der Fehlercode zurückgesetzt. Ebenfalls wird der Zähler *spiaddrcount* zurückgesetzt. Dieser Zähler adressiert den SPI-Datenpuffer.

Im nächsten State *state_setwrdata* wird der 16bit-Datenbus des SPI-Datenpuffers je nach Stand des Adresszählers mit den zu sendenden Daten vorbesetzt und die Schreibfreigabe *spiwren* für den Datenpuffer gesetzt.

Daraufhin wird im folgenden State *state_checkwriteend* die Schreibfreigabe wieder zurückgenommen. Hat der Adresszähler noch nicht alle Datenpufferadressen durchgezählt, wird er inkrementiert und erneut der State *state_setwrdata* angesprungen. Wurde jedoch das Ende das Datenpuffers erreicht, springt das Modul in den State *state_complete* und setzt das Ready-Signal. Daraufhin verweilt das Modul wieder in dem initialen State *state_init*.

4.4.13. Gesamtstruktur der FPGA-Software (usdsp)

Die beschriebenen Module wurden zu einer Gesamtstruktur zusammengefasst. Die Struktur wurde als Schaltbild entworfen. Sie ist im Anhang B.3.14 aufgeführt.

Die Teilmodule

- Ultraschallimpulserzeugung (bestehend aus *shotcontroller* und *shotsync*)
- Sampledatenspeicher (*samplemem*)
- Berechnung der Laufzeit (*calculationmain*)
- SPI-Datenübertragung (*spicntrl*)

wurden in einer Verarbeitungskette angeordnet. Ebenfalls wurden alle externen Schnittstellen in diesem Modul an entsprechende Ein- und Ausgangspins des FPGAs geführt.

Das Berechnungsmodul ist nur fähig, mit einer Taktfrequenz von 20 MHz zu arbeiten. Daher wurde der Taktteiler *clockdiv* eingefügt, der den 80 MHz Systemtakt auf 20 MHz herunterteilt.

Die Koordination der Module untereinander erfolgt über das Steuermodul *maincontroller*.

Der Quelltext des Steuermoduls ist in Anhang B.3.15 aufgeführt.

Das Modul setzt zunächst im State *state_init* alle Start-Signale zurück. Danach wird im State *state_startcalc* das Start-Signal für das Berechnungsmodul *calculationmain* gestartet.

Im State *state_waitcalc* wird solange gewartet, bis *calculationmain* sein Ready-Signal setzt.

Danach wird im State *state_startspi* das SPI-Datenübertragungsmodul *spinctrl* gestartet. Hiernach wird wiederum im State *state_waitspi* auf das Ready-Flag des Modul *spicntrl* gewartet. Nachdem das Ready-Signal gesetzt wurde, beginnt der gesamte Zyklus erneut durch Springen in den State *state_init*.

4.5. Test der Teilmodule

Durch die beschriebene Architektur der Teilmodule war es möglich, diese Module unabhängig voneinander in einer Testumgebung zu testen.

Der Test der Module erfolgte hierbei in der Simulationsumgebung der Designsoftware Quartus II. Diese Simulationsumgebung arbeitet auf der Signalebene, d.h. es werden Signalmuster auf die Eingänge gelegt. Der Simulator simuliert dann die Antwort der Ausgänge auf die angelegten Signalmuster.

Zum Test jedes Teilmoduls wurde das jeweilige Modul in ein eigenständiges Projekt geladen und die Modulanschlüsse direkt an Ein- und Ausgänge gelegt.

Um die Teilmodule der Signalverarbeitungskette zu testen, war es nötig, diesen Modulen das ALU-Modul und den Sampledatenspeicher zur Verfügung zu stellen.

Damit die Module mit echten Signaldaten getestet werden konnten, wurde eine spezielle Version des Sampledatenspeichers generiert. Diese Version wurde als ROM-Speicher deklariert und der ROM-Inhalt mit den Sampledaten aus der OCTAVE-Simulation vorbelegt.

Die Modulparameter sowie die Steuersignale wurden direkt in der Simulation als Signalmuster an die Eingänge gelegt.

Durch die Simulation war es möglich, ein Fehlverhalten der Teilmodule zu erkennen. Da jedoch nur Ein- und Ausgänge in der Simulation berücksichtigt werden, konnten keine Signale innerhalb eines Moduls beobachtet werden.

Um dennoch Signale innerhalb eines Moduls beobachten zu können, wurde aus jedem Teilmodul ein *Debug*-Bus herausgeführt. Die inneren Signale konnten wahlweise auf den Debug-Bus gelegt werden und waren somit auch in der Simulation zugänglich.

5. Realisierung der Hardware

Ein großer Teil des Hardwareentwurfs konnte aus dem bereits vorhandenen System wiederverwendet werden. Die Schaltpläne wurden mit Hilfe der Designsoftware INTEGRA erstellt. INTEGRA ermöglicht es, einerseits die Eingabe von Schaltplänen und andererseits das komplette Layout der Platine durchzuführen.

Nach der Eingabe der Schaltpläne wurden die Daten an eine Mitarbeiterin von Fresenius Medical Care weitergegeben, die das Layout für die Platine erstellte und zur Fertigung der Platinen an eine externe Firma weitergab.

Der grundlegende Aufbau der Hardware ist in Abbildung 5-24 wiedergegeben.

Im folgenden werden die Komponenten und ihre Funktion vorgestellt.

5.1. Versorgung Sendeimpulserzeugung

Zur Anregung der sendeseitigen Piezoelemente ist eine Impulsamplitude von $-47\,V$ erforderlich. Diese Spannung wird mit Hilfe des Schaltreglers IC3 (Schaltbild 1/ D,E,F-1,2,3) vom Typ LM3478 und seiner Peripherie aus dem 24 V-Versorgungszweig erzeugt.

Die 24 V werden dem Schaltregler jedoch nicht direkt zugeführt, sondern über eine Aktivierungslogik bestehend aus dem NOR-Gatter IC7 sowie den Transistoren T4 und T5 (Schaltbild 1/ C,D-1,2,3). Diese Aktivierungslogik sorgt dafür, dass die 24 V Versorgungsspannung nur an den Schaltregler geführt wird, wenn die externen Signale *US_IMPOFF_AR* sowie *US_IMPOFF_SR* logisch 0 entsprechen. Durch diese Aktivierungslogik ist es möglich, dass die übergeordneten Systeme den Ultraschallsender außer Kraft setzen können und keine Energie aus dem Sender zu dem Patienten zugeführt wird. Die 24 V Versorgung wird über den Pufferkondensator C18 (Schaltbild 3/ A,B-1,2) von der Dialysemaschine zugeführt.

Die über C16 gepufferte Sendeimpulsspannung wird über die Widerstände R37 und R61 an die arterielle und venöse Sendeimpulserzeugung geleitet.

Zusätzlich wird die Sendeimpulsspannung an den Doppel-Operationsverstärker IC5 (Schaltbild 1/ D,E,F-7,8) geführt. Der OP ist als Komparator geschaltet und gibt einen logischen 1-Pegel ab, wenn die Schaltreglerspannung nicht im Bereichs-

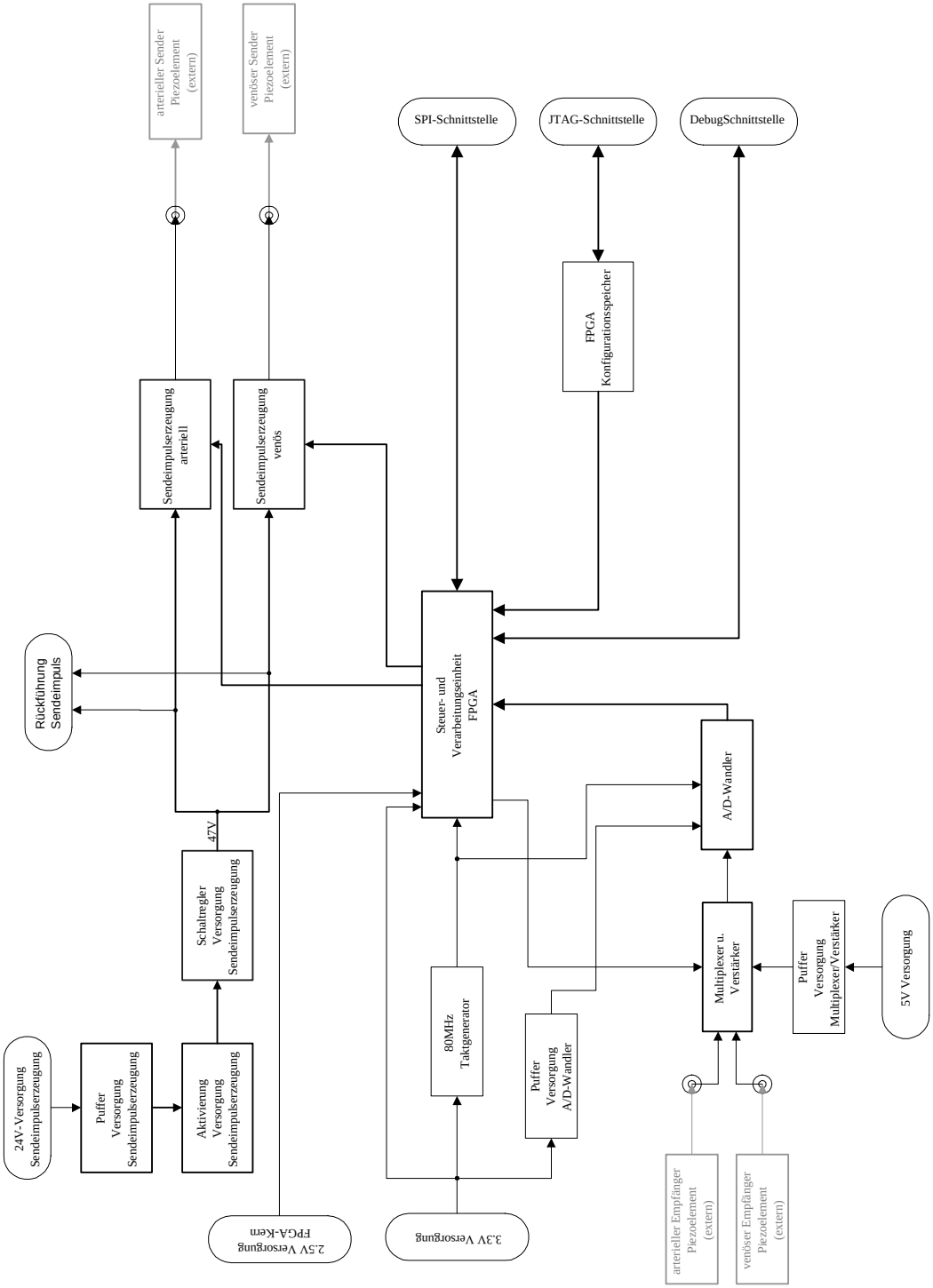


Abbildung 5-24: Komponenten der Hardware

fenster von ca. -47 V liegt. Die Ausgangssignale des OP werden auf das NOR-Glied IC12 (Schaltbild 1/B,C-2,3) geführt, um die OP-Signale TTL-kompatibel zu machen und um das Signal zu invertieren. Die Signale *US_VENIMP_AR(SR)* sowie *US_ARTIMP_AR(SR)* ermöglichen den übergeordneten Systemen die Überwachung der korrekten Funktion des Schaltreglers.

5.2. Sendeimpulserzeugung

Die Sendeimpulserzeugung ist für die arterielle und venöse Seite identisch aufgebaut, weswegen hier nur die arterielle Seite beschrieben wird.

Das Ansteuersignal *fpga_imp_art* aus dem FPGA wird über den Transistor T8 (Schaltbild 1 / A-1,2) invertiert und auf TTL-Pegel gebracht.

Dieses Signal *send_imp_art* wird auf das Gate des FET T10 (Schaltbild 1/ F-4) geleitet. Ist *fpga_imp_art* logisch 1, so wird *send_imp_art* logisch 0. Der FET T10 sperrt und die -47 V-Spannung aus dem Schaltregler wird über eine Umverschaltung aus Zenerdioden, die sicherstellen, dass die Spannung -47 V nicht überschreitet, auf die Ausgangsbuchse X7 und damit dem Sendepiezoelement zugeführt. Ist *fpga_imp_art* logisch 0, so zieht T10 die -47 V-Spannung nach Masse und das Piezoelement wird nicht mehr versorgt.

Eine Abschwächung des Sendeimpulses auf -33 V kann über das Signal *fpga_test_amp_art* erreicht werden. Das Signal wird über das NOT-Gatter IC12 (Schaltbild 1/C-1) auf TTL-Pegel gebracht und invertiert. Das invertierte Signal *send_test_amp_art* wird dem FET T3 zugeführt. Ist der FET T3 durchgeschaltet (*fpga_test_amp_art* ist logisch 0), so greift die Zenerdiode D10 und begrenzt die Ausgangsspannung.

5.3. Eingangssignalaufbereitung und A/D-Wandler

Die von den Piezoelementen empfangenen Ultraschallsignale werden über Koaxleitungen an die Eingangsbuchsen X5 und X6 geführt. Die Signaleingänge sind über Anpassungsglieder an den Multiplexer IC1 (Schaltbild 1/ A,B,C-4,5,6) vom Typ MAX4310 angeschlossen. Dieser Multiplexer beinhaltet ebenfalls eine Verstärkerstufe, die das schwache Eingangsignal von ca. $U_{SS} = 200\ mV$ auf $U_{SS} = 1\ V$ ver-

stärkt. Die Auswahl des Eingangskanals erfolgt über die Signalleitung *rec_chsel*. Dieses Signal wird aus dem vom FPGA kommenden Signal *fpga_rec_chsel* über das NOT-Gatter IC12 abgeleitet. IC12 sorgt hierbei wieder für die Pegelanpassung von der 3.3 V IO-Spannung des FPGA auf den benötigten TTL-kompatiblen 5 V-Pegel. Führt *fpga_rec_chsel* H-Pegel, so wird der arterielle Kanal ausgewählt, anderenfalls der venöse Kanal.

Die Versorgung des Multiplexers wird über die Pufferschaltung DR2 und C14 (Schaltbild 3/ B-2) aus der 5 V-Versorgung des Systems abgeleitet.

Das ausgewählte und verstärkte Eingangssignal wird an den A/D-Wandler IC4 (Schaltbild 1/ B,C-7,8) vom Typ AD9283 geführt.

Der A/D-Wandler wandelt das Eingangssignal mit einer Abtastrate von 80 MHz in ein digitales Signal mit einer Wortbreite von 8 bit. Das Taktsignal wird von dem FPGA bereitgestellt.

Der Datenbus des A/D-Wandlers wird direkt an den FPGA geführt.

Der A/D-Wandler benötigt zwei getrennte Versorgungsspannungen von jeweils 3.3 V. Die Versorgungsspannung für den Analogteil wird mit den Pufferelementen DR1 und C17 (Schaltbild 3/ D-2) aus der 3.3 V-Versorgung des Systems abgeleitet.

Die Versorgungsspannung für den digitalen Teil wird direkt aus der über den Kondensator C38 (Schaltbild 3/ D-1) gepufferten 3.3 V-Versorgung des Systems bereitgestellt.

5.4. Systemtaktgenerator

Der Takt des Systems beträgt 80 MHz. Um die eingangs erwähnte Jitterabweichung von ± 200 *ps* zu erreichen, ist es notwendig, dass ein spezieller Quarzoszillator zum Einsatz kommt. Dieser Quarzoszillator Q1 (Schaltbild 2/ F-2) der Firma FOQ wird aus der 3.3 V Versorgungsspannung des Systems gespeist und erzeugt ein 80 MHz-Rechtecksignal. Dieser Takt wird zum einen an den Takteingang des FPGA und zum anderen an den A/D-Wandler geführt. Q2 ist ein Dummy-Bauelement wleches die spätere Verwendung eines Quarzes mit anderer Bauform in der Schaltung erlaubt.

5.5. Steuer- und Verarbeitungseinheit

Die bisher genannten Schaltungskomponenten konnten direkt in das neue System übernommen werden, der folgende Teil musste jedoch neu entworfen werden.

Aufgrund des Bedarfs an Logikelementen der FPGA-Software von ca. 4000 LEs wurde das FPGA EP1K100QC208-3 von Altera ausgewählt, das eine Kapazität von 5000 LEs besitzt.

Dieses FPGA ersetzt den in der Originalhardware vorhandenen FPGA EP1K10 sowie den DSP mitsamt seiner Peripherieschaltung wie Speicher und Busdekoder.

Der FPGA besitzt die Möglichkeit, die IO-Signalanschlüsse, bis auf wenige für Sonderfunktionen reservierte Anschlüsse, frei zu belegen. Die Belegung der Anschlüsse wurde so gewählt, dass die Leiterbahnführung möglichst kurz zu den anderen Komponenten verlaufen kann.

Die Belegung der frei programmierbaren Anschlüsse des FPGA (IC 8) ist in Schaltbild 5 zu sehen. Das Taktsignal des 80 MHz Oszillators wird an den reservierten Takteingang Pin 79 geführt.

Das Reset-Signal des übergeordneten Systems ist an den nur als Eingang nutzbaren Pin 78 angeschlossen.

Die LEDs D11-D13 (Schaltbild 5 / F-1) können von dem FPGA zu Kontrollzwecken angesteuert werden. Ebenso wurden 17 Datenleitungen von dem FPGA auf die Steckerleiste X1 geführt, die für Debugging-Zwecke frei verwendbar sind.

Das FPGA kommuniziert über SPI-Schnittstellen mit dem übergeordneten System. Es wurden drei unabhängie SPI-Kanäle vorgesehen (*US_SPI*, *SPI2* sowie *SPI3*). Diese SPI-Signalleitungen werden auf den Systemsteckverbinder X2 geführt.

Der FPGA wird mit zwei Versorgungsspannungen betrieben. Der Logikkern benötigt eine Versorgungsspannung von 2.5 V, während die Spannung für die IO-Leitungen 3.3 V beträgt. Die Versorgung wird über mehrere Punkte in den FPGA eingespeist. Jeder Einspeisungspunkt muss dabei mit Pufferkondensatoren von jeweils 10 nF und 100 nF gepuffert[9] werden, da Schaltvorgänge im FPGA kurze Spannungseinbrüche auf den Versorgungsleitungen zu Folge haben. Die Verschal-

[9] einige Hinweise zur korrekten Pufferung sind in [8] aufgeführt

tung der Pufferkondensatoren an den FPGA-Versorgungseingängen ist im Schaltbild 2/ C,D,E,F-4,5,6,7,8 zu sehen.

5.6. FPGA-Konfigurationsspeicher

Die Konfiguration des FPGA[10] wird in dem Konfigurationsspeicher IC11 (Schaltbild 2/ C-2,3) vom Typ EPC2 gespeichert.

Der Speicher wird über die seriellen Konfigurationsleitungen des FPGA ausgelesen und über eine externe JTAG-Schnittstelle, die ebenfalls auf den Systemsteckverbinder X2 geführt ist, programmiert.

Optional ist es möglich, dass das FPGA direkt von den übergeordneten Systemen programmiert werden kann. Um die Konfigurationsquelle zwischen dem Konfigurationsspeicher und den externen Konfigurationsleitungen umzuschalten, wurden die Jumper J2-J6 vorgesehen.

Die JTAG-Schnittstelle ist als Kette zwischen FPGA, dem Konfigurationsspeicher und dem übergeordneten System geschaltet. Somit ist es auch möglich per JTAG mit dem FPGA zu kommunizieren, um z.B. direkt den Pegel von einzelnen IO-Anschlüssen abzufragen. Damit die Kette nicht unterbrochen wird, falls der Konfigurationsspeicher fehlt, dient der Jumper J1 als Brücke.

5.7. Platinenlayout

Die Bauelemente wurden zunächst mit Hilfe des Layout-Moduls der Software INTEGRA auf der vorgegebenen Platinengröße von 110 mm x 60 mm vorplatziert.

Besonderes Augenmerk wurde hierbei auf die exakte Platzierung des Systemschnittstellensteckers X2 gelegt. Die Bauelemente wurden nach ihren Funktionsgruppen angeordnet.

Die Platzierung der Elemente Quarzoszillator, FPGA und A/D-Wandler wurde so gewählt, dass die Verbindungsstrecken der Elemente Oszillator - FPGA und FPGA - A/D-Wandler sehr kurz gehalten werden konnten. Der Debugstecker X1 wurde so platziert, dass er auch im eingebauten Zustand der Platine leicht zugänglich

[10] weitergehende Information zur Konfiguration sind in [9] verzeichnet

ist.

Nachdem die Vorplatzierung vollzogen wurde, konnte die Platine von einer FMC-Mitarbeiterin layoutet werden. Es stellte sich heraus, dass ein vierlagiger Aufbau der Platine genügt, um alle Leiterbahnen zu berücksichtigen.

Nachdem das Layout fertiggestellt wurde, konnte die Platine an einen externen Fertiger zur Fertigung und auch Bestückung der Bauelemente übergeben werden.

6. Inbetriebnahme der Soft- und Hardware

6.1. Inbetriebnahme der Hardware

Es wurden fünf Platinen von dem externen Fertiger erstellt. Das Ergebnis ist in Abbildung 6-25 zu sehen.

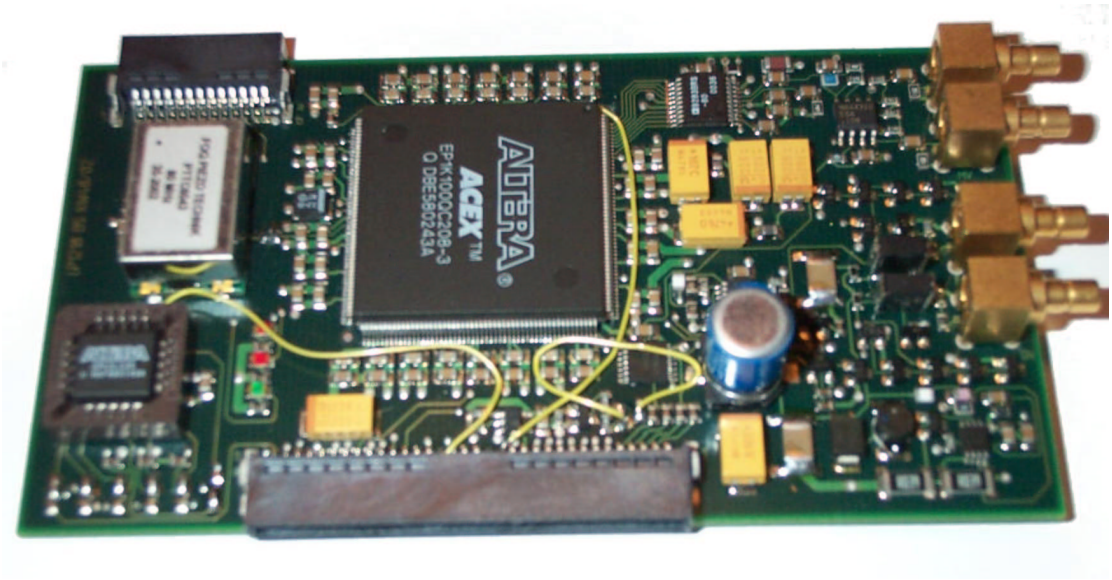


Abbildung 6-25: bestückte Platine

Zunächst wurden die Platinen visuell auf ihre korrekte Bestückung und auf eventuell fehlerhafte Lötstellen überprüft. Diese visuelle Überprüfung verlief erfolgreich.

Es erschien nicht sinnvoll, die Platine sofort in einem Dialysemaschinenaufbau in Betrieb zu nehmen, da hierbei viele externe Komponenten Einfluss auf den Betrieb der Schaltung nehmen könnten. Daher wurde eine Laborplatine aufgebaut, die es ermöglicht, die Ultraschallplatinen ohne externe Systeme in Berieb zu nehmen.

Die Laborplatine ist in Abbildung 6-26 abgebildet.

Eine 68polige SMC-Sockelleiste wurde auf die Laborplatine gelötet, in der die Ultraschallplatine eingesteckt wird. Mehrere Spannungsregler auf der Laborplatine erzeugen aus einer 7.5 V Versorgungsspannung, die von einem Labornetzteil zur Verfügung gestellt wird, die Versorgungsspannungen für die Ultraschallplatine (2.5 V; 3.3 V und 5 V). Zusätzlich stellt das Labornetzteil eine 24 V Spannung für die Sendeimpulserzeugung bereit.

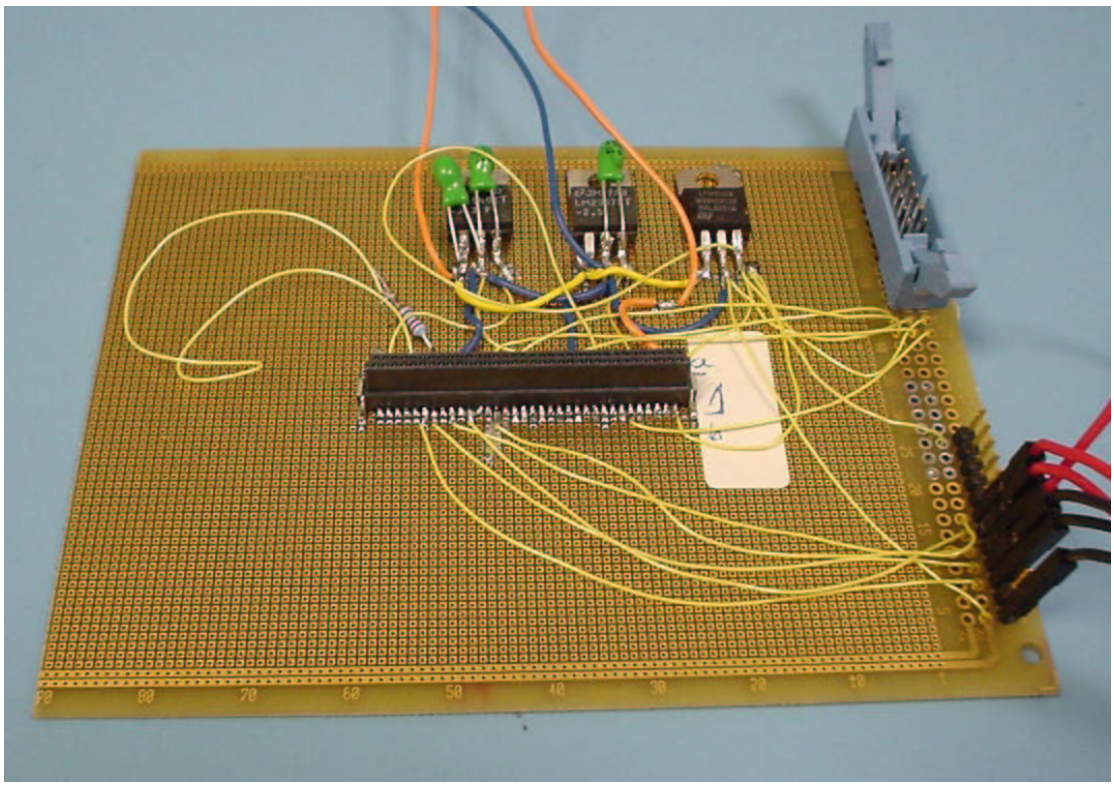


Abbildung 6-26: Aufbau der Laborplatine

Weiterhin wurden die JTAG-Konfigurationsleitungen und das SPI-Interface auf Pfostensteckerleisten geführt. Ein mit dem PC verbundenes JTAG-Konfigurationskabel kann direkt auf die JTAG-Pfostenleiste aufgesteckt werden.

Die Konfiguration des FPGAs auf der Ultraschallplatine erfolgt über einen EPC2-Konfigurationsspeicher, der über die externe JTAG-Schnittstelle programmiert wird. Daher wurden die Jumper J1 - J6 in die entsprechenden Positionen verbunden.

Zunächst wurde überprüft, ob das System über die JTAG-Schnittstelle konfigurierbar ist und der FPGA anläuft. Es wurde eine minimale FPGA-Software erstellt, welche die drei Debug-LEDs D11 - D13 ansteuert.

Mit einer in der Software Quartus II integrierten JTAG-Programmiersoftware wurde versucht, die FPGA-Software in den EPC2-Konfigurationsspeicher zu laden. Dieser Vorgang schlug zunächst fehl, da die JTAG-Programmiersoftware keine Verbindung mit dem JTAG-Bus herstellen konnte.

Eine naheliegende Fehlstellung der Jumper J1 - J6 konnte nach einer Überprüfung

ausgeschlossen werden.

Mit Hilfe eines Oszilloskops wurde der Signalweg auf dem JTAG-Bus verfolgt und dabei festgestellt, dass sich im Schaltplan der Ultraschallplatine ein Fehler eingeschlichen hatte und keine Verbindung der JTAG-Steuerleitungen zu dem Systemschnittstellenstecker bestand.

Die Verbindungen wurden nachträglich mit WireWrap-Draht nachgezogen. Die Korrektur war erfolgreich und der EPC2-Baustein konnte über JTAG von dem PC programmiert werden. Ebenfalls war der FPGA in der Lage, die Konfiguration aus dem Konfigurationsspeicher zu laden. Das in der Testsoftware einprogrammierte Verhalten der Debug-LEDs wurde ausgeführt.

Im nächsten Testschritt wurde die in Abschnitt 4.4 entwickelte Software in das FPGA geladen. In der Software ist LED D11 ist so programmiert, dass sie ständig leuchtet; die LED D12 leuchtet, wenn der Algorithmus einen Fehlercode zurückgibt und LED D13 schaltet ihren Zustand um, wenn eine neue Ultraschalllaufzeit in das SPI-Interface übertragen wurde (regelmäßiges Blinken).

Die Software lief jedoch nicht an. Da die LED D11 aufleuchtete, konnte davon ausgegangen werden, dass die Konfiguration erfolgreich verlief.

Mit dem Oszilloskop wurde gemessen, ob der Ultraschallsendeimpuls erzeugt wurde. Auch der Sendeimpuls blieb aus, der Schaltregler generierte jedoch aus der 24 V Versorgung die -47 V Sendeimpulsspannung.

Nun lag die Vermutung nahe, dass der Quarzoszillator kein Taktsignal erzeugt. Daher wurde am Ausgang des Oszillators das Taktsignal gemessen. Das Signal war vorhanden; daraufhin wurde das Taktsignal direkt am Takteingang des FPGA gemessen. Hier erschien das Taktsignal nicht mehr. Das einzige Bauteil auf dem Signalpfad war der Widerstand R31. Nach dem Überbrücken des Widerstandes begann der FPGA, das Programm abzuarbeiten. Eine Untersuchung des Widerstandes ergab, dass entgegen dem Bestückungsplan ein Widerstand mit dem Wert 22 KOhm anstatt eines Exemplars mit dem Wert 220 Ohm eingesetzt wurde.

Eine Begutachtung der erzeugten Ultraschallsendeimpulse ergab, dass die Längen und Abstände der Sendeimpulse den Anforderungen entsprachen.

6.2. Inbetriebnahme der FPGA-Software

Zur Durchführung weiterer Untersuchungen war es notwendig, eine Ultraschallsensorstrecke zur Verfügung zu haben. Hierzu wurde ein bereits vorhandener experimenteller Aufbau verwendet, welcher aus einem Blutschlauchsystem mit den arteriellen und venösen Sensorpaaren besteht.

Der komplette Versuchsaufbau ist in Abbildung 6-27 ersichtlich.

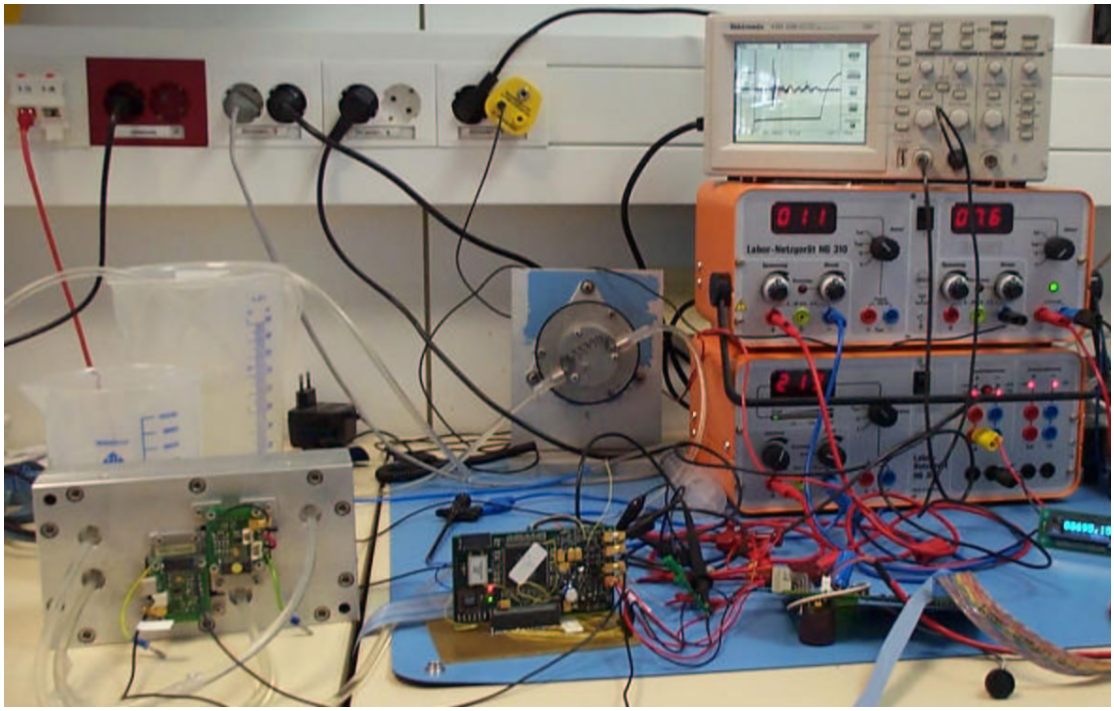


Abbildung 6-27: Versuchsaufbau

Auf der linken Seite ist das Blutschlauchsystem mit der Ultraschallsensorstrecke zu sehen. In der Mitte befindet sich die Ultraschallplatine und auf der rechten Seite die Labornetzteile zur Spannungsversorgung.

Die Ultraschallsensorstrecke wurde mit Koaxialkabeln verbunden. Die Funktion der Ultraschallsender konnte leicht überprüft werden, da im Betrieb ein schwacher Pfeifton zu vernehmen ist.

Das Blutschlauchsystem wurde mit Wasser befüllt. Die Ultraschallauswertung erkannte kein gültiges Signal (LED D12 leuchtete permanent auf). Eine Messung des am Empfänger anliegenden Ultraschallsignals ergab jedoch ein sauber ausgesteuertes Signal. Dieses Signal lag aber nicht mehr hinter dem Multiplexer an. Eine

gleichzeitige Messung des Kanalauswahlsignals zeigte, dass dieses Signal invertiert war. Während der Messung des arteriellen Kanals wurde der venöse Kanal vom Multiplexer durchgeschaltet. Nach dem Einfügen eines Inverters in den Kanalauswahlzweig der FPGA-Software wurde der Fehler behoben und ein gültiges Signal erkannt.

Nun folgte der Test der SPI-Schnittstelle. Um diesen Test durchzuführen, wurde die Ultraschallplatine in einen Dialysemaschinenaufbau eingebaut. Hier zeigte sich der Fehler, dass die Ultraschallplatine zwar Daten lieferte, die gemessenen Laufzeiten jedoch im Bereich von mehreren Nanosekunden schwankten.

Da nur wenige Dialysemaschinenaufbauten vorhanden waren und die Analyse der Fehler sehr langwierig sein kann, wurde der oben beschriebene Laboraufbau mit einer C167 CPU-Karte erweitert. Es wurde ein kleines C-Programm geschrieben, das im zeitlichen Abstand von 100 ms die Ultraschalllaufzeiten aus der Ultraschallplatine ausliest, diese Daten in einen Speicher schreibt und zusätzlich die Laufzeiten vorformatiert auf einem Display ausgibt.

Die in den Speicher geschriebenen Ultraschalllaufzeiten konnten mit dem Debugger aus der Tasking-C167 Entwicklungsumgebung ausgelesen und grafisch dargestellt werden.

Laut Anforderungsliste werden die Laufzeiten in zwei 16-bit-Worten übertragen, ein 16-bit-Wort repräsentiert die Vorkommastellen der Laufzeit in Nanosekunden, das zweite 16-bit-Wort überträgt die Nachkommastellen. Von den 16-bit Nachkommastellen werden jedoch nur 10-bit genutzt. Diese 10-bit wurden allerdings in die Bitpositionen 15 - 6 hineinkodiert und nicht in die Bitpositionen 9 - 0. Nach der Korrektur der Kodierung der Nachkommastellen wurde festgestellt, dass die Laufzeiten innerhalb der geforderten Messgenauigkeit lagen.

Nachem die Messgenauigkeiten die Anforderungen erfüllten, wurden Versuche durchgeführt, die das Verhalten der Ultraschallauswertung im Fehlerfalle untersuchen sollten. Hierzu wurden mehrere Szenarien untersucht:

- Es wurde Luft in das Schlauchsystem hineingepumpt
- Die Ultraschallempfänger (Sender) wurden abgekoppelt
- Die Ultraschallempfänger (Sender) wurden vertauscht

Bei allen Szenarien war die Schaltung in der Lage, den Fehler zu erkennen. Über das SPI-Interface wurden jedoch gültige Ultraschall-Laufzeiten übertragen, die aus der letzten fehlerfreien Messung stammen. Daher wurde die FPGA-Software so angepasst, dass bei einer fehlerhaften Messung die zu übertragende Laufzeit auf den Wert Null gesetzt wurde.

Weiterhin zeigte die Schaltung das Verhalten, dass, sobald eine Fehlersituation eintrat, dieser nicht mehr aufgehoben wurde. Nach einer Untersuchung der Fehlerbehandlungsroutine in der FPGA-Software wurde festgestellt, dass der Fehlercode, der den Fehlerzustand beschreibt, nicht wieder zurückgesetzt wurde. Nachdem der Fehlercode vor jedem neuen Messzyklus zurückgesetzt wurde, war die Schaltung in der Lage, nach dem Auftreten und Beheben eines Fehlerfalls wieder gültige Messungen zu liefern.

6.3. Untersuchung der geforderten Messgenauigkeit

Zur Bestimmung der Messgenauigkeit des Systems wurde der oben beschriebene Laboraufbau verwendet. Die Ultraschallsensorstrecke wurde mit Leitungswasser bei Zimmertemperatur durchströmt. Es wurden sowohl Messungen mit konstantem Wasserfluss als auch mit stehendem Wasser durchgeführt.

Die Ultraschall-Messwerte wurden im Abstand von 100 ms von der C167-CPU ausgelesen und in ein Datenfeld geschrieben, das 8000 Werte umfasste. Somit wurden die Ultraschall-Messwerte über einen Zeitraum von

$$8000 \times 100\ ms = 800\ s = 13\ min$$

erfasst.

Die in den Datenfeldern gespeicherten Messwerte wurden über den C167-Debugger CrossView in den PC ausgelesen und grafisch ausgewertet.

Betrachtet wird hier nur die Messung des arteriellen Kanals. Eine hier nicht näher erläuterte Überprüfung der Messergebnisse des venösen Kanals zeigten ein ähnliches Verhalten.

Zunächst wurde die Messung ohne Pumpfluss durchgeführt.

Abbildung 6-28 zeigt die grafische Auswertung der Messung über die komplette Messdauer von ca. 13 Minuten.

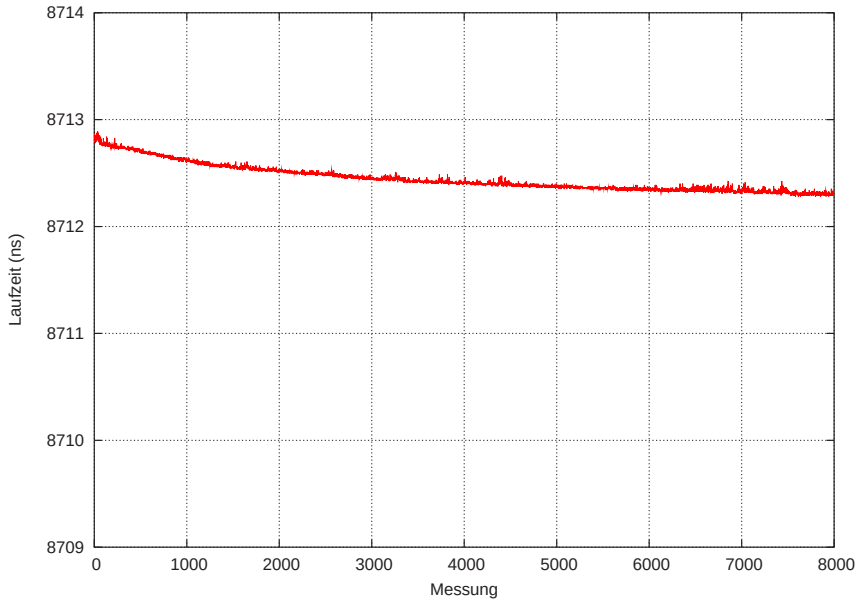


Abbildung 6-28: Laufzeit ohne Pumpfluss über 13 Minuten

Zu sehen ist hier ein annähernd lineares Absinken der Ultraschall-Laufzeit innerhalb des Messzeitraums.

Dieses lineare Absinken läßt sich dadurch erklären, dass das System nicht auf konstanter Temperatur gehalten wird. Das Leitungswasser erwärmt sich im Laufe der Zeit auf die Umgebungstemperatur. Das Wasser hat die Eigenschaft, dass mit zunehmender Temperatur ebenfalls die Schallgeschwindigkeit zunimmt.

Um die Messgenauigkeit der Laufzeitmessung zu ermitteln, war es daher nötig, nur einen kurzen Zeitbereich der aufgenommenen Messung zu betrachten. Zur Untersuchung wurden die Messwerte im Bereich von Messung 0 bis Messung 150 herangezogen. Der Zeitbereich dieser Messung entspricht $15 \times 100\ ms = 15\ s$.

Der untersuchte Ausschnitt der Messung ist in Abbildung 6-29 grafisch dargestellt.

Um ein Maß für die Genauigkeit der Laufzeitmessung zu bekommen, wurde die Standardabweichung der Messwerte über einen Zeitraum von 15 Sekunden gebildet. Zur Berechnung dieser Abweichung wurde OCTAVE eingesetzt. Die Messwerte wurden in OCTAVE geladen und die Standardabweichung mit Hilfe des Befehls *std()* ermittelt.

Die Funktion *std()* ermittelt die Standardabweichung über mehrere Datensätze.

Der Wert dieser Standardabweichung beträgt in dem kurzen Messzeitraum von

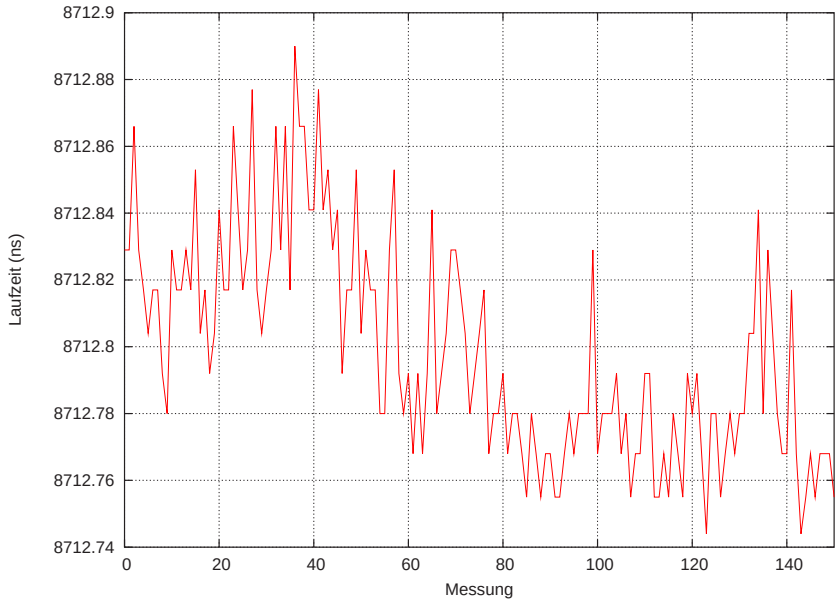


Abbildung 6-29: Laufzeit ohne Pumpfluss über 15 Sekunden

15 Sekunden einem Wert von ± 35 *ps*. Dies zeigt, dass die Ultraschallelektronik deutlich genauer misst, als in den Anforderungen spezifiziert wurde.

Die nächste Messung wurde mit eingeschalteter Pumpe durchgeführt (also mit konstantem Fluss). Abbildung 6-30 zeigt das Ergebnis der Messung über den Zeitraum von 13 Minuten.

Hier ist zunächst ein stärkerer Abfall der Ultraschall-Laufzeit über den Zeitraum zu sehen. Der stärkere Abfall der Laufzeit ist bedingt durch den Austausch der Umgebungswärme mit dem Wasserbehälter.

Zur Ermittlung der Genauigkeit der Messung wurde ebenfalls der oben genannte kurze Zeitraum betrachtet. Die grafische Darstellung dieses Messbereichs ist in Abbildung 6-31 zu sehen.

Eine Analyse der Standardabweichung der Messwerte in diesem Bereich ergab eine Standardabweichung von ± 52 *ps*.

Mit Pumpfluss ist auch die Genauigkeit der Messung in kurzen Zeitbereichen eingeschränkt, dies geht jedoch nicht zu Lasten der Ultraschallelektronik. Ursachen für die Erhöhung der Ungenauigkeit sind z.B. lokale Verwirbelungen des Wassers innerhalb der Messkammer sowie Veränderungen der Umgebungsbedingungen wie der Temperatur.

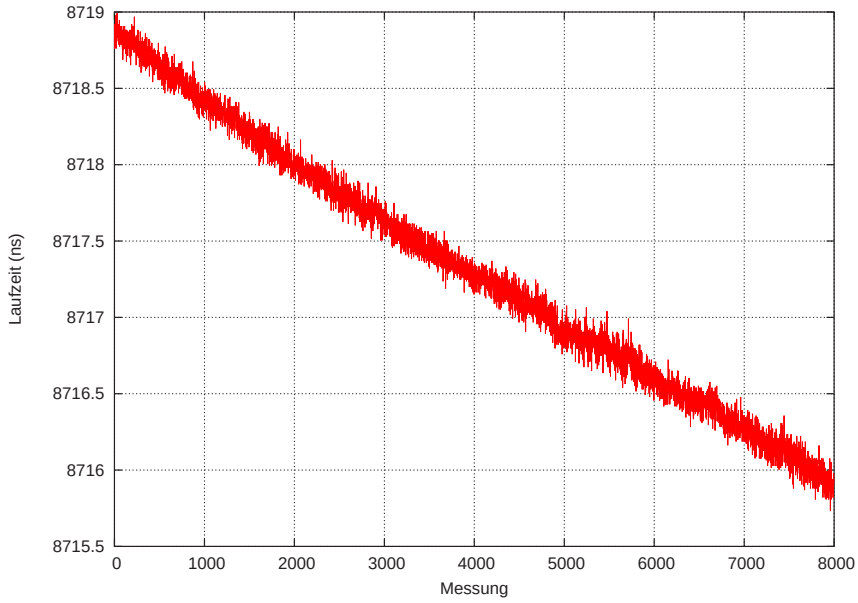


Abbildung 6-30: Laufzeit mit Pumpfluss über 13 Minuten

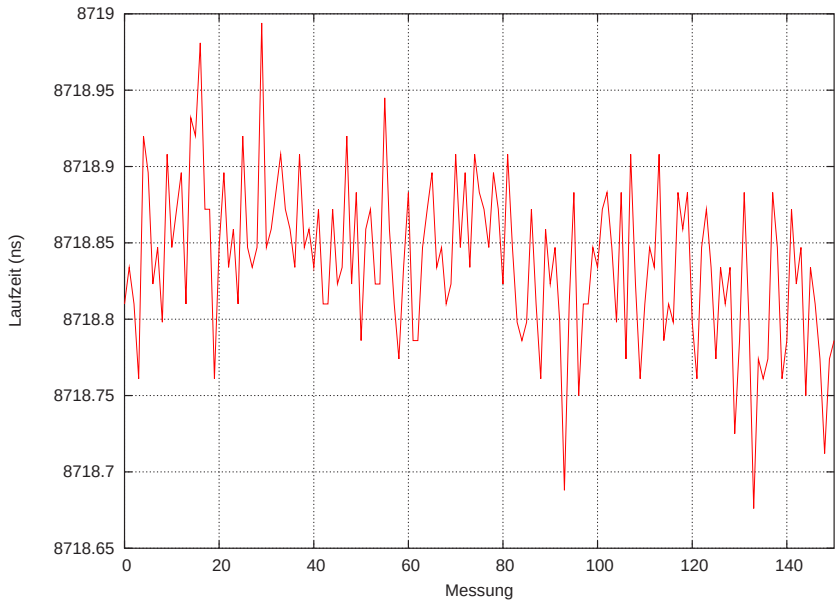


Abbildung 6-31: Laufzeit mit Pumpfluss über 15 Sekunden

7. Schlussbetrachtung

Die Inbetriebnahme der neuen Ultraschallsteuer- und Auswerteelektronik konnte erfolgreich beendet werden.

Abschließend werden nun die Ergebnisse der vorliegenden Diplomarbeit diskutiert.

7.1. Technische Aspekte

Die in den Anforderungen festgehaltenen technischen Eigenschaften wurden mit dem in dieser Diplomarbeit vorgestellten Ansatz erfüllt.

Das System ist in der Lage, die Steuerung und die Auswertung einer Ultraschall-Laufzeitmessung durchzuführen. Die Genauigkeit der Messung wurde in Abschnitt 6.3 nachgewiesen.

Die Kompatibilität mit dem Originalsystem ist gewährleistet, lediglich die Software der übergeordneten Systeme ist leicht zu modifizieren, da der Nachkommastellen-Anteil der Laufzeit in einem geringfügig abgeänderten Format übergeben wird.

Das System könnte in der Dialysebehandlung Einsatz finden. Hierzu wären jedoch noch zusätzliche sicherheitstechnische Funktionalitäten erforderlich.

7.2. Wirtschaftliche Aspekte

Das Platinenlayout der neuen Ultraschallelektronik konnte im Vergleich zu der Originalelektronik stark vereinfacht werden. Die Originalplatine ist doppelseitig bestückt und besteht aus acht Lagen während die neue Elektronik nur einseitig bestückt ist und aus vier Lagen besteht. Hier zeigt sich der Vorteil der Integration der kompletten digitalen Verarbeitungseinheit als System on Chip auf einem FPGA.

Der Hauptkostenfaktor der neuen Elektronik ist das FPGA und der Konfigurationsspeicher.

Die Realisierung der FPGA-Software erforderte den größten Baustein aus der LowCost-Familie von Altera. Der niedrigste SpeedGrade des Bausteins ist jedoch für die Schaltung ausreichend. Somit belaufen sich die Kosten für das FPGA auf

ca. 25 Euro pro Stück bei einer Menge von 100 Stück.

Die Kosten für den Altera EPC2-Konfigurationsspeicher belaufen sich auf 20 Euro pro Stück bei einer Menge von 100 Stück.

Der Konfigurationsspeicher ist jedoch nicht zwingend notwendig; es wurde eine Möglichkeit vorgesehen, das FPGA mit Hilfe der übergeordneten Syteme zu konfigurieren.

7.3. Vergleich der FPGA-Variante mit der DSP-Variante

Die Variante, die digitale Verarbeitungseinheit als System on Chip auf einem FPGA zu realisieren, bietet mehrere Vorteile, aber auch Nachteile.

Die Vorteile der FPGA-Lösung liegen eindeutig in dem vereinfachten Aufbau der Leiterplatte. Die Anzahl der Lagen konnte stark reduziert werden; ebenso war bei gleicher Baugröße keine doppelseitige Bestückung der Platine notwendig.

Das komplette DSP-System mitsamt seinen Peripheriekomponenten konnte eingespart werden. Der hardwareseitige Aufbau des digitalen Schaltungsteils gestaltet sich aufgrund der wenigen Bauelemente wesentlich einfacher.

Die Kosten für den FPGA-Baustein liegen in der vorliegenden Schaltung in ähnlichen Größenordnungen wie das DSP-System mitsamt seiner Peripherie. Diese Kosten können jedoch das DSP-System übersteigen, wenn weitere Funktionalität gefordert wird, da ein größerer Baustein eingesetzt werden müsste.

Ein gravierender Nachteil der FPGA-Lösung ist die geringere softwareseitige Flexibilität. Eine Änderung des Algorithmus erfordert in einem DSP-System nur die Änderung des C-Quellcodes.

Änderungen des Algorithmus auf der FPGA-Ebene erfordern jedoch größeren Aufwand, da der Algorithmus als Hardwarebeschreibung vorliegt. Somit muss sich der Entwickler auf Hardwareebene mit dem System beschäftigen. Dies erfordert unter anderem einen erhöhten Zeitaufwand für den Entwicklungsprozess. Weiterhin kann die Anzahl der zur Verfügung stehenden Logikelemente durch Änderungen des Algorithmus schnell überschritten werden. In einem solchen Fall müsste ein neuer FPGA-Baustein mit höheren Logikelementkapazitäten eingesetzt werden.

Ebenfalls kann der Fall eintreten, dass Timing-Vorgaben für das System durch

geringfügige Änderungen nicht mehr eingehalten werden können. Diese Gefahr läßt sich durch ein sauberes synchrones Design jedoch weitgehend vermeiden.

7.4. Mögliche Optimierungen

Das vorliegende System bietet noch einen gewissen Spielraum für Optimierungen, die jedoch in dieser Arbeit aus zeitlichen Gründen nicht mehr in das System einfliessen konnten.

Ein Hauptziel der Optimierungen wäre die Reduktion der benötigten Logikelemente. Wären signifikant weniger Logikelemente nötig, könnten sicherheitsrelevante Funktionen in das vorhandene FPGA integriert werden.

Ein Ansatz zur Optimierung des Systems könnte sein, weitere Funktionen in die ALU zu integrieren. In der Implementierung werden oft Vergleiche durchgeführt, die direkt zu Logikbausteinen synthetisiert werden. Ein Auslagern der Vergleichsfunktionen in die ALU könnte Logikelemente einsparen. Das Auslagern in die ALU hat jedoch den Nachteil, dass die Verarbeitungszeit ansteigt, da zur Anwendung der ALU eine Wertübergabe in Registern durchgeführt werden muss.

Weiterhin wäre ein Gegenstand der Optimierung die Untersuchung, ob der Aufbau des Dividierers innerhalb der ALU platzsparender zu realisieren ist.

A. Literaturverzeichnis

Literatur

[1] Fresenius Medical Care: *Die Fresenius Medical Care AG 2001/02 auf einen Blick.* Stand März 2003.
Internet-Link : http://www.fresenius.de/1/pdf/01_3b_fmc.pdf

[2] Pschyrembel, W.: *Pschyrembel Klinisches Wörterbuch.* 256. Auflage. de Gruyter Verlag, Berlin (1994).

[3] Tietze, U./ Schenk, Ch.: *Halbleiter-Schaltungstechnik.* 10. Auflage. Springer Verlag, Berlin (1993).

[4] Heusinger/ Ronge/ Stock: *Handbuch der PLDs und FPGAs.* 1. Auflage. Franzis Verlag, Poing (1994).

[5] Lehmann/ Wunder/ Selz: *Schaltungsdesign mit VHDL.* 1. Auflage. Franzis Verlag, Poing (1994).

[6] Altera Digital Library CD: *Datenblatt ACEX1K-Serie.* siehe auch beiliegende CD-ROM im Verzeichnis /datasheets/acex.pdf

[7] Eaton, John W.: *GNU Octave Manual* 3. Auflage. enthalten im OCTAVE-Programmpaket (Version 2.1)

[8] Altera Digital Library CD: *Application Note 75, High-Speed Board Designs.* siehe auch beiliegende CD-ROM im Verzeichnis /datasheets/an075.pdf

[9] Altera Digital Library CD: *Application Note 116, Configuring SRAM-bases LUT Devices.* siehe auch beiliegende CD-ROM im Verzeichnis /datasheets/an116.pdf

B. Programmquellen

B.1. Octave-Skript zur Konvertierung der Messdaten

```
1  function convdat(filenamein, filenameout, sampleperpacket)
2  % liest eine Datei mit aufeinanderfolgenden Messzyklen ein
3  % und ordnet die Daten in einer neuen Matrix an.
4  % Die Spalten der neuen Matrix entsprechen den einzelnen Messzyklen
5
6  % filenamein - Dateiname der Datei mit Rohdaten
7  % filenameout - Dateiname der zu schreibenden Datei
8  % sampleperpacket - Länge eines einzelnen Messzyklus
9
10     % lade die Rohdaten in die Variable rawdata
11     load filenamein rawdata;
12
13     % ermittle die Anzahl der Datensätze
14     [numberofsamples temp] = size(rawdata);
15
16     % berechne die Anzahl der Messzyklen
17     packets = numberofsamples/sampleperpacket;
18
19     % schreibe ersten Messzyklus in erste Spalte von Variable impulses
20     impulses = rawdata(1:sampleperpacket);
21
22     % extrahiere Messzyklen
23     for i=1:(packets-1)
24         samplestart = (i*sampleperpacket)+1;
25
26         % hänge die extrahierten Messzyklen als neue Spalten an die
27         % Variable impulses an
28         impulses = [impulses
29                     rawdata(samplestart:(i*sampleperpacket)+sampleperpacket)];
30     endfor;
31
32     % speichere die neue Matrix in filenameout
33     save filenameout impulses;
34 endfunction
```

B.2. Octave-Simulation des Algorithmus

```
function usplot

    % Anzahl der Samples im Datensatz
    SAMPLEANZ = 800;
    % Abtastfrequenz
    FSAMPLE = 80E6;
    % Startindex der Baseline-Berechnung
    BASESTART = 100;
    % Stopindex der Baseline-Berechnung
    BASEEND = 400;

    % Startindex der Signalsuche
    SIGSTART = 400;
    % Stopindex der Signalsuche
    SIGEND = SAMPLEANZ;

    % minimale Fläche einer Signalperiode
    AREAMIN = 100;
    % minimale Signal Periode
    WAVEMINPERIOD = 20;
    % maximale Signal Periode
    WAVEMAXPERIOD = 90;

    % minimale Amplitude
    WAVEMINAMPL = 50;

    % Anzahl der Interpolationsschritte
    INTERPOLCOUNT = 2;

    % flag ffür Plot
    DOPLOT = 1;

    % plotparameters
    grid("on");
    axis([690,(695),100,200]);

    %load impuls2.dat

    % lade Sample-Datei
    load pulses2.dat

    % Variablen initialiseren
    meantime = 0;
    means = 0;
    time = 0;

    xisum = 0;
    yisum = 0;
    % x = ussignal();

    % laufe durch Sample-Datensätze

    %for z=1:400
    for z=1:168

        % extrahiere aktuellen Datensatz aus Sample-Datei
        x = impulses(:,z);

        % ermittle die Größe des Sample-Datensatzes
        [datasize temp] = size(x);
        y= ones(datasize,1);


        % berechne die Baseline anhand von Mittelwert
        % zwischen BASESTART und BASEEND
        mval = mean(x(BASESTART:BASEEND))
        mvali = floor(mval)
        z
```

```
70          % führe Variable mit Wert=Baseline ein -> für Plot
            y = y .* mval;
72

74          % initialisiere Zählregister
            posfl = 0;
76          negfl = 0;
            poscount = 0;
78          negcount = 0;
            xlsignal = 0;
80          xrsignal = 0;

82          foundwave = 0;

84          % laufe durch den Resonanzbereich des Ultraschallsignals
            for i = SIGSTART:SIGEND
86
                % prüfe, ob das Signal überhalb der Baseline ist
88              if( x(i) > mvali )
                    % wenn vorheriges Sample negativ war
90                  % setzte Summe von positiver Fläche zurück
                    if( negcount == 1)
92                      % printf("found negative area : %d\n", negfl);
                        posfl = 0;
94                      negcount = 0;
                    endif
96                  % setzte Flag für positive Samplewerte
                    poscount = 1;
98                  % summiere die zur Baseline relative Signalamplitude
                    % in Summenregister für positive Wert
100                 posfl = posfl + (x(i)-mvali);
                endif
102
                % prüfe ob das Signal unterhalb der Baseline ist
104             if( x(i) <= mvali )
                    % wenn vorheriges Sample positiv war
106                 % setzte Summe von negativer Fläche zurück
                    if( poscount == 1 )
108                     % printf("found positive area : %d\n", posfl);
                        negfl = 0;
110                     poscount = 0;
                    endif
112                 % setze Flag für negative Samplewerte
                    negcount = 1;
114                 % summiere die zur Baseline relative Signalamplitude
                    % in Summenregister für negative Werte
116                 negfl = negfl + (mvali-x(i));
                endif
118
                % prüfe, ob wir uns in einem positiven Signalbereich befinden
120             % und der vorherige negative Bereich eine Mindestfläche
                % erreicht hatte
122             if( (negfl > AREAMIN) & (poscount==1))

124                 % finde das obere Ende des Signalverlaufs

126                 j = i % setzte Zählregister mit aktueller Sampleposition

128                 % inkrementiere den Zähler, solange das Signal noch
                    % positiv ist und noch nicht das Ende des Resonanzbereichs
130                 % erreicht ist
                    while( (x(j) > mvali) & (j < SIGEND) )
132                     j = j + 1;
                    endwhile
134
                    % der Zähler steht nun auf dem Index des ersten negativen
136                 % Wertes nach dem positiven Bereich
                    % dekrementiere den Zähler um 1 Position, somit entspricht
138                 % der Zählerstand dem letzten positiven Sample
                    j=j-1;
140                 rightposlimit = j  % speichere oberes Ende des
```

```
                                % Signalverlaufs
142
                % finde das untere Ende des Signalverlaufs
144
                % laufe vom oberen Ende zurück, solange bis das Signal
146             % wieder negativ wird oder das Ende des Resonanzbereichs
                % erreicht ist
148             while( (x(j) >= mvali) & (j > SIGSTART) )
                    j = j -1;
150             endwhile

152             jt = j

154             % laufe von dem Übergangspunkt negativ/positiv zurück,
                % solange bis das Signal positiv wird oder das Ende des
156             % Resonanzbereichs erreicht ist
                while( (x(j) < mvali) & (j > SIGSTART) )
158                 j = j -1;
                endwhile
160
                js = j
162             % der Zähler steht nun auf dem index des letzten positiven
                % Wertes vor dem negativen Bereich
164
                % inkrementiere den Zähler um 1, damit er am unteren
166             % Ende des interessanten Bereichs steht

168             leftposlimit = j+1; % speichere unteres Ende des
                                    % Signalverlaufs
170
                % berechne Signalmetriken
172             % berechne die Periode aus Abstand oberes /unteres Ende
                waveperiod = rightposlimit - leftposlimit;
174             % finde Maxima im interessanten Bereich
                wavemin = min(x(leftposlimit:rightposlimit));
176             % finde Minima im interessanten Bereich
                wavemax = max(x(leftposlimit:rightposlimit));
178
                % berechne Signalamplitude (Spitze-Spitze) aus
180             % Minima und Maxima
                waveampl = (wavemax-mvali)+(mvali-wavemin);
182
                % prüfe, ob der erkannte Schwingungsbereich die geforderten
184             % Charakteristika erfüllt
                % wenn ja, setzte Flag für gefundenes Signal
186             if( (waveampl >= WAVEMINAMPL) & (waveperiod<=WAVEMAXPERIOD)
                    & ( waveperiod >= WAVEMINPERIOD) )
188                 xlsignal = [xlsignal leftposlimit];
                    xrsignal = [xrsignal rightposlimit];
190                 foundwave = 1;
                endif
192         endif

194         % wenn ein Signal gefunden wurde, speichere den Übergang
            % zwischen negativen und positiven Bereich = Bereich
196         % der gesuchten Nullstelle
            % und breche die weitere Untersuchung des Signals ab
198         % ansonsten untersuche das Signal weiter
            if( foundwave == 1)
200             i = leftposlimit;

202             while( (x(i) <= mvali) & (i < SIGEND))
                    i++;
204             endwhile

206             uppertrans = i
                lowertrans = i-1
208             break;
            endif
210
        endfor  % interesting region loop
```

```
212

214         [temp siganz] = size(xlsignal);
            % initialisiere Hilfsvariablen zur Interpolation
216         m = 0;
            ns = 0;
218
            upper = 0;
220         lower = 0;
            xupper = 0;
222         xlower = 0;

224         % summiere obere und untere Punkte auf
            for i=0:(INTERPOLCOUNT-1)
226             upper = upper + (uppertrans + i);
                lower = lower + (lowertrans - i);
228
                xupper = xupper + x(uppertrans+i);
230             xlower = xlower + x(lowertrans-i);
            endfor
232
            % berechne den Mittelwert der aufsummierten Punkte
234         upper = upper / INTERPOLCOUNT;
            lower = lower / INTERPOLCOUNT;
236         xupper = xupper / INTERPOLCOUNT;
            xlower = xlower / INTERPOLCOUNT;
238
            % berechne die Steigung
240         m = (xupper - xlower) / (upper-lower);
            % berechne den y-Achsenabschnitt
242         ns =(mval-xlower)/m + lower;

244         % berechne die Laufzeit aus interpoliertem Sampleindex
            % und speichere die Laufzeit ab
246         if( z == 1 )
                time = ns*12.5;
248         else
                time = [time ns*12.5];
250         endif

252 %         if mod(z, 10 ) == 0
    %             printf("mittlere Laufzeit: %f ns\n", meantime);
254 %             means = [means meantime];
    %             meantime = 0;
256 %         else
    %             meantime = meantime + (ns*12.5);
258 %         endif
            printf("Laufzeit: %f ns\n", ns * 12.5);
260

262         if (DOPLOT == 1)
                graw("unset arrow\n");
264             plot(x,"-@",y);

266             for i = 2:siganz
                    gnuplotstring = sprintf(
268                     "set arrow from %d,0 to %d,255 nohead lt 3\n",
                        xlsignal(1,i), xlsignal(1,i));
270                 graw(gnuplotstring);
                    gnuplotstring = sprintf(
272                     "set arrow from %d,0 to %d,255 nohead lt 3\n",
                        xrsignal(1,i), xrsignal(1,i));
274                 graw(gnuplotstring);

276                 gnuplotstring = sprintf(
                        "set arrow from %f,0 to %f,255 nohead lt 5\n", ns, ns);
278                 graw(gnuplotstring);
                endfor
280             graw("replot\n");
                pause;
282         endif
```

```
284
        endfor % nächster Datensatz
286     % Berechne Signalstatistiken
        printf("Standardabweichung : %f Mittel : %f\n",std(time), mean(time));
288
        printf("max() : %f maxabweichung() : %f\n",
290             max(time), max(time) - mean(time) );
        printf("min() : %f minabweichung() : %f\n",
292             min(time), mean(time) - min(time) );

294
        GLOBALMEANSTEP = 10
296
        erst = 1;
298     meanshot = 0;
        % berechne den Mittelwert aus den einzelnen Laufzeiten
300     for i=1:160
            if( mod(i, GLOBALMEANSTEP) == 0 )
302             if( erst == 1)
                    erst = 0;
304                 means = meanshot / (GLOBALMEANSTEP-1);
                else
306                 means = [means (meanshot/GLOBALMEANSTEP)];
                endif
308             meanshot = time(i);
            else
310             meanshot = meanshot + time(i);
            endif
312     endfor

314     means
        std(means)
316     axis();
        hist(time,100);
318
    endfunction
```

B.3. FPGA-Sofware

B.3.1. Teilmodul Baselineberechnung

```
2  ----------------------------------------------------------------------
   --  Projekt: USDSP                                                  --
   --           Ultraschalllaufzeitmessung                             --
4  --                                                                  --
   --  Modul:   MEANPROC                                               --
6  --           berechnet den Mittelwert von einer Anzahl von Samples  --
   --                                                                  --
8  --  Inputs:                                                         --
   --    clk             - System Takt                                 --
10 --    reset           - System Reset                                --
   --    start_i         - starten den Berechnungsvorgang              --
12 --    datasrc_i       - 8bit Datenbus von Samplequelle              --
   --    meanstart_i     - 10bit startindex der Berechnung             --
14 --    meanstop_i      - 10bit stopindex der Berechnung              --
   --    alurege1_i      - 24bit Ergebnisregister 1 von ALU            --
16 --    alurege2_i      - 24bit Ergebnisregister 2 von ALU            --
   --    alustatus_i     - 4bit Statusregister von ALU                 --
18 --    aluready_i      - Ready Flag von ALU                          --
   --  Outputs:                                                        --
20 --    ready_o           - zeigt Ende der Mittelwertbildung an       --
   --    datasrcaddr_o     - 10bit Adressbus zur Samplequelle          --
22 --    dataaccess_o      - zeigt Benutzung der Samplequelle an       --
   --    errorcode_o       - 4bit Fehlercode                           --
24 --    alurega_o         - 24bit ALU Operand Register 1              --
   --    aluregb_o         - 24bit ALU Operand Register 2              --
26 --    aluop_o           - 8bit ALu Opcode                           --
   --    alusel_o          - zeigt Benutzung der ALU an                --
28 --    alustart_o        - startet eine ALU Operation                --
   --    mean_o            - 8bit Ganzzahl Ergebnis der Mittelwertbildung --
30 --    mean16_o          - 16bit Fixedpoint Ergebnis der Mittelwertbildung --
   --                                                                  --
32 --  Autor: A. Kühn                                                  --
   --  Datum: 24.09.2002                                               --
34 --       Revision : a - 24.10.2002                                  --
   ----------------------------------------------------------------------
36


38
   -- include standard libraries
40 LIBRARY ieee;
   USE ieee.std_logic_1164.ALL;
42 USE ieee.std_logic_arith.all;

44
   LIBRARY lpm;
46 USE lpm.lpm_components.ALL;

48
   -- Moduldefinition
50 ENTITY meanproc IS
       PORT(
52 --DEBUG
           dbgport        : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
54 --ENDDEBUG

56         clk            : IN    STD_LOGIC;
           reset          : IN    STD_LOGIC;
58         start_i        : IN    STD_LOGIC;
           datasrc_i      : IN    UNSIGNED(7 DOWNTO 0);
60         meanstart_i    : IN    UNSIGNED(9 DOWNTO 0);
           meanstop_i     : IN    UNSIGNED(9 DOWNTO 0);
62         alurege1_i     : IN    UNSIGNED(23 DOWNTO 0);
           alurege2_i     : IN    UNSIGNED(23 DOWNTO 0);
64         alustatus_i    : IN    UNSIGNED(3 DOWNTO 0);
           aluready_i     : IN    STD_LOGIC;
```

```
66
        ready_o        : OUT   STD_LOGIC;
68      datasrcaddr_o  : OUT   STD_LOGIC_VECTOR(9 DOWNTO 0);
        dataaccess_o   : OUT   STD_LOGIC;
70      errorcode_o    : OUT   UNSIGNED(3 DOWNTO 0);

72      alurega_o      : OUT   UNSIGNED(23 DOWNTO 0);
        aluregb_o      : OUT   UNSIGNED(23 DOWNTO 0);
74      aluop_o        : OUT   UNSIGNED(7 DOWNTO 0);
        alusel_o       : OUT   STD_LOGIC;
76      alustart_o     : OUT   STD_LOGIC;

78      mean_o         : OUT   UNSIGNED(7 DOWNTO 0);
        mean16_o       : OUT   UNSIGNED(15 DOWNTO 0)
80      );
END meanproc;
82

84  ARCHITECTURE a OF meanproc IS

86      -- Deklaration der States
        TYPE STATE_TYPE IS ( state_init, state_checkparams,
88                           state_prepsamplediff, state_samplediff,
                             state_ssamplediff, state_postsamplediff,
90                           state_prepsamplecount, state_samplecount,
                             state_ssamplecount, state_postsamplecount,
92                           state_prepsum, state_ssum,state_sum,
                             state_postsum, state_sumloop,
94                           state_prepdivide, state_divide, state_sdivide,
                             state_postdivide,
96                           state_prepscale, state_scale, state_sscale,
                             state_postscale,
98                           state_complete);

100 -- signals

102     SIGNAL state: STATE_TYPE;                       -- Statevariable

104
        -- auf 16bit erweitertes Signal für Rest der Division
106     SIGNAL rem16:       UNSIGNED(15 DOWNTO 0);

108 BEGIN

110     -- Statemachine
        PROCESS (clk, reset)
112
        -- Register für Paramter
114     VARIABLE meanstart : UNSIGNED(9 DOWNTO 0);
        VARIABLE meanstop  : UNSIGNED(9 DOWNTO 0);
116
        -- Zählregister für Samplequellenadressindex
118     VARIABLE datasrcaddress : UNSIGNED(9 DOWNTO 0);

120     -- Register für Differenz zwischen Stop- und Startsampleindex
        VARIABLE samplediff : UNSIGNED(9 DOWNTO 0);
122
        -- Register für Anzahl Samples -> 1bit größer als samplediff wg.Overflow
124     VARIABLE samplecount : UNSIGNED(10 DOWNTO 0);

126
        -- Register zum Aufsummieren der Werte (2bit Reserve wg. Overflow)
128     VARIABLE sumval : UNSIGNED(17 DOWNTO 0);

130     VARIABLE shiftrem : UNSIGNED(15 DOWNTO 0);

132     -- Register für Mittelwert
        VARIABLE mean    : UNSIGNED(7 DOWNTO 0);
134     -- Register für Rest des Mittelwertes
        VARIABLE meanrem : UNSIGNED(7 DOWNTO 0);
136
```

```
    -- Register für Fixedpoint-wert des Mittelwertes
138 VARIABLE mean16 : UNSIGNED(15 DOWNTO 0);

140 VARIABLE errorcode : UNSIGNED(3 DOWNTO 0);

142 BEGIN
        -- asynchroner Reset
144     IF reset = '1' THEN

146         -- setze Ready-Flag zurück
            ready_o <= '0';
148
            -- Setze Zugriffs-Flags für ALU und Samplequelle zurück
150         dataaccess_o <= '0';
            alusel_o <= '0';
152
            -- initialer State
154         state <= state_init;

156     -- positive Taktflanke
        ELSIF clk'EVENT AND clk = '1' THEN
158
            CASE state IS
160
                -- Init State
162             WHEN state_init =>

164                 -- warte auf Start-Signal
                    IF start_i = '1' THEN
166                     -- lade Adressindex mit Startsampleindex vor
                        datasrcaddress := meanstart_i;
168
                        meanstart   := meanstart_i;
170                     meanstop := meanstop_i;

172                     -- setze Ready-Flag zurück
                        ready_o <= '0';
174                     -- Lösche Summenregister
                        sumval := CONV_UNSIGNED(0,18);
176
                        errorcode := "0000";
178
                        -- Setze Zugriffs-Flags für ALU und Samplequelle
180                     dataaccess_o <= '1';
                        alusel_o <= '1';
182
                        -- nächsten State setzen
184                     state <= state_checkparams;
                    ELSE
186                     -- in diesem State verweilen
                        state <= state_init;
188                 END IF;

190
                WHEN state_checkparams =>
192                 -- prüfe, ob die Paramter logisch sind
                    IF meanstart > meanstop THEN
194                     errorcode := "0001";
                        state <= state_complete;
196                 ELSE
                        state <= state_prepsamplediff;
198                 END IF;

200             -- Samplediff State
                WHEN state_prepsamplediff =>
202                 -- bilde Differenz zwischen Stop- und Startsampleindex
                    aluregа_o <= CONV_UNSIGNED( meanstop, 24 );
204                 aluregb_o <= CONV_UNSIGNED( meanstart, 24 );

206                 aluop_o   <= CONV_UNSIGNED( 02, 8);
                    state <= state_sampledIff;
```

```
208
                WHEN state_samplediff =>
210                 -- starte ALU operation
                    alustart_o <= '1';
212                 state <= state_ssamplediff;

214             WHEN state_ssamplediff =>
                    -- ALU start Flag zurücksetzen
216                 alustart_o <= '0';
                    state <= state_postsamplediff;
218
                WHEN state_postsamplediff =>
220                 -- wenn ALU fertig , speichere
                    -- Ergebnis in samplediff-Register
222                 IF aluready_i = '1' THEN
                        samplediff := CONV_UNSIGNED( alurege1_i, 10 );
224                     state <= state_prepsamplecount;
                    ELSE
226                     state <= state_postsamplediff;
                    END IF;
228
                -- Samplecount State
230             WHEN state_prepsamplecount =>
                    -- addiere 1 zu samplediff um Anzahl der Samples
232                 -- zu ermitteln
                    alurega_o <= CONV_UNSIGNED( samplediff, 24 );
234                 aluregb_o <= CONV_UNSIGNED( 1, 24 );

236                 aluop_o   <= CONV_UNSIGNED( 01, 8);
                    state <= state_samplecount;
238
                WHEN state_samplecount =>
240                 -- starte ALU operation
                    alustart_o <= '1';
242                 state <= state_ssamplecount;

244             WHEN state_ssamplecount =>
                    -- ALU start Flag zurücksetzen
246                 alustart_o <= '0';
                    state <= state_postsamplecount;
248
                WHEN state_postsamplecount =>
250                 -- wenn ALU fertig ist, speichere
                    -- Ergebnis in samplecount-Register
252                 IF aluready_i = '1' THEN
                        samplecount := CONV_UNSIGNED( alurege1_i, 11 );
254                     state <= state_prepsum;
                    ELSE
256                     state <= state_postsamplecount;
                    END IF;
258
                -- Sum State
260             WHEN state_prepsum =>
                    -- Summiere den aktuellen Samplewert auf
262                 -- das Summenregister auf
                    alurega_o <= CONV_UNSIGNED( sumval, 24 );
264                 aluregb_o <= CONV_UNSIGNED( datasrc_i, 24 );

266                 aluop_o   <= CONV_UNSIGNED( 01, 8);
                    state <= state_sum;
268
                WHEN state_sum =>
270                 -- starte ALU operation
                    alustart_o <= '1';
272                 state <= state_ssum;

274             WHEN state_ssum =>
                    -- ALU start Flag zurücksetzen
276                 alustart_o <= '0';
                    state <= state_postsum;
278
```

```
                    WHEN state_postsum =>
280                     -- wenn ALU fertig ist, speichere
                        -- Ergebnis in Summenregister
282                     IF aluready_i = '1' THEN
                            sumval := CONV_UNSIGNED( alurege1_i, 18 );
284                         state <= state_sumloop;
                        ELSE
286                         state <= state_postsum;
                        END IF;
288
                    WHEN state_sumloop =>
290                     -- wenn der obere Samplequellenindex erreicht ist,
                        -- wird in
292                     -- die Division gesprungen, ansonsten wird der
                        -- Adressindex inkrementiert und die Summation
294                     -- erneut ausgeführt

296                     IF datasrcaddress = meanstop THEN
                            state <= state_prepdivide;
298                     ELSE
                            datasrcaddress :=datasrcaddress + 1;
300                         state <= state_prepsum;
                        END IF;
302
                    -- Divide State
304                 WHEN state_prepdivide =>
                        -- dividiere den Summenwert durch die Anzahl der Samples
306                     aluregа_o <= CONV_UNSIGNED( sumval, 24 );
                        aluregb_o <= CONV_UNSIGNED( samplecount, 24 );
308
                        aluop_o <= CONV_UNSIGNED(4,8);
310                     state <= state_divide;

312                 WHEN state_divide =>
                        -- starte ALU operation
314                     alustart_o <= '1';
                        state <= state_sdivide;
316
                    WHEN state_sdivide =>
318                     -- ALU start Flag zurücksetzen
                        alustart_o <= '0';
320                     state <= state_postdivide;

322                 WHEN state_postdivide =>
                        -- wenn ALU fertig ist, speichere 8bit
324                     -- Ergebnis in mean-Register
                        -- und 8bit-Rest in meanrem Register
326                     IF aluready_i = '1' THEN
                            IF alustatus_i = CONV_UNSIGNED(0,4) THEN
328                             mean := CONV_UNSIGNED( alurege1_i, 8);
                                meanrem := CONV_UNSIGNED( alurege2_i, 8);
330                             state <= state_prepscale;
                            ELSE
332                             errorcode := "0010";
                                state <= state_complete;
334                         END IF;
                        ELSE
336                         state <= state_postdivide;
                        END IF;
338
                    -- Scale State
340                 WHEN state_prepscale =>
                        -- dividiere den auf 16bit erweiterten Rest
342                     -- der Mittelwertdivision
                        -- durch Anzahl der Samples um 8bit Nachkommastellen
344                     -- zu erhalten
                        aluregа_o <= CONV_UNSIGNED( rem16, 24 );
346                     aluregb_o <= CONV_UNSIGNED( samplecount, 24 );

348                     aluop_o <= CONV_UNSIGNED(4,8);
                        state <= state_scale;
```

```
350
                WHEN state_scale =>
352                 -- starte ALU operation
                    alustart_o <= '1';
354                 state <= state_sscale;

356             WHEN state_sscale =>
                    -- ALU start Flag zurücksetzen
358                 alustart_o <= '0';
                    state <= state_postscale;
360
                WHEN state_postscale =>
362                 -- wenn ALU fertig ist, speichere 16bit
                    -- Ergebnis in mean16-Register
364
                    IF alureadY_i = '1' THEN
366                     IF alustatus_i = CONV_UNSIGNED(0,4) THEN
                            mean16 := CONV_UNSIGNED( alurege1_i, 16 );
368                     ELSE
                            errorcode := "0011";
370                     END IF;
                        state <= state_complete;
372                 ELSE
                        state <= state_postscale;
374                 END IF;

376
                -- Complete-State
378             WHEN state_complete =>
                    -- setze Ready-Flag
380                 ready_o <= '1';
                    -- Setze Zugriffs-Flags für ALU und Samplequelle zurück
382                 alusel_o <= '0';
                    dataaccess_o <= '0';
384                 -- in Init-State zurückspringen
                    state <= state_init;
386
                WHEN others =>
388                 state <= state_init;

390
            END CASE;
392     END IF;
        -- Ende Statemachine
394
        -- verbinde 8bit Ganzzahl Mittelwert-Ausgang mit Register
396     mean_o <= mean;

398     -- erweitere den 8bit Rest auf 16bit
        rem16(15 DOWNTO 8) <= meanrem;
400     rem16(7 DOWNTO 0) <= "00000000";

402     -- verbinde 8bit Ganzzahl und 8bit Nachkomma-Teil
        -- des Mittelwertes zu 16bit auf Ausgang
404     mean16_o(15 DOWNTO 8) <= mean;
        mean16_o(7 DOWNTO 0) <= mean16(7 DOWNTO 0);
406     -- verbinde Sampleadressindexzähler mit
        -- Adressausgang zur Samplequelle
408     datasrcaddr_o <= CONV_STD_LOGIC_VECTOR(datasrcaddress,10 );

410     errorcode_o <= errorcode;

412             dbgport(7 DOWNTO 0) <= CONV_STD_LOGIC_VECTOR(datasrc_i,8);
                dbgport(15 DOWNTO 8) <= CONV_STD_LOGIC_VECTOR(mean,8);
414             dbgport(16) <= start_i;

416 END PROCESS;

418

420 END a;
```

B.3.2. Teilmodul ALU

```
--------------------------------------------------------------------------
--  Projekt: USDSP                                                      --
--           Ultraschall-Laufzeitmessung                                --
--                                                                      --
--  Modul:   ALUCORE                                                    --
--           stellt arithmetische Dienste zur Verfügung                 --
--                                                                      --
--  Inputs:                                                             --
--    clk            - Taktsignal                                       --
--    reset          - bringt das Modul in einen definierten Zustand    --
--    start_i        - ist dieses Signal H, wird der Vorgang gestartet  --
--    regop1_i       - 24bit Operanden Register 1                       --
--    regop2_i       - 24bit Operanden Register 2                       --
--    opcode_i       - 8bit Opcode Register                             --
--  Outputs:                                                            --
--    ready_o        - wird H, wenn die Berechnung beendet wurde        --
--    rege1_o        - 24bit Ergebnis Register 1                        --
--    rege2_o        - 24bit Ergebnis Register 2                        --
--    regstat_o      - 8bit Statusregister                              --
--                                                                      --
--  Autor: A. Kühn                                                      --
--  Datum:  21.10.2002                                                  --
--  Revision :  a - 28.11.2002                                          --
--------------------------------------------------------------------------


-- Einbinden der Standardbibliotheken
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

-- ALU Mnemonics einbinden


-- Moduldefinition
ENTITY alucore IS
    PORT(
        clk                         : IN    STD_LOGIC;
        reset                       : IN    STD_LOGIC;
        start_i                     : IN    STD_LOGIC;
        regop1_i                    : IN    UNSIGNED(23 DOWNTO 0);
        regop2_i                    : IN    UNSIGNED(23 DOWNTO 0);
        opcode_i                    : IN    UNSIGNED(7 DOWNTO 0);

        ready_o                     : OUT   STD_LOGIC;
        rege1_o                     : OUT   UNSIGNED(23 DOWNTO 0);
        rege2_o                     : OUT   UNSIGNED(23 DOWNTO 0);
        regstat_o                   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
END alucore;


ARCHITECTURE a OF alucore IS

-- Komponenten

-- 24bit Addierer/Subtrahierer
component lpm_addsub_24_24
    PORT
    (
        add_sub     : IN STD_LOGIC ;
        dataa       : IN STD_LOGIC_VECTOR (23 DOWNTO 0);
        datab       : IN STD_LOGIC_VECTOR (23 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (23 DOWNTO 0);
        overflow    : OUT STD_LOGIC
```

```
70      );
    end component;
72
    -- 24bit Dividierer
74  component lpm_divide_24_24p
        PORT
76      (
            numer       : IN STD_LOGIC_VECTOR (23 DOWNTO 0);
78          denom       : IN STD_LOGIC_VECTOR (23 DOWNTO 0);
            clock       : IN STD_LOGIC ;
80          quotient        : OUT STD_LOGIC_VECTOR (23 DOWNTO 0);
            remain      : OUT STD_LOGIC_VECTOR (23 DOWNTO 0)
82      );
    end component;
84
        -- Deklaration der States
86      TYPE STATE_TYPE IS ( state_init,
                        state_decodeop,state_divide,
88                      state_divideready, state_dividewait,
                        state_complete );
90
        SIGNAL state: STATE_TYPE;                       -- Statevariable
92
        SIGNAL addsub: STD_LOGIC;
94
        SIGNAL aval : UNSIGNED(23 DOWNTO 0);
96      SIGNAL bval : UNSIGNED(23 DOWNTO 0);
        SIGNAL addsubval : STD_LOGIC_VECTOR(23 DOWNTO 0);
98      SIGNAL addsubov  : STD_LOGIC;
        SIGNAL divval    : STD_LOGIC_VECTOR(23 DOWNTO 0);
100     SIGNAL divrem    : STD_LOGIC_VECTOR(23 DOWNTO 0);

102


104
    BEGIN
106
        -- Statemachine
108     PROCESS (clk, reset)

110     VARIABLE rega : UNSIGNED(23 DOWNTO 0);
        VARIABLE regb : UNSIGNED(23 DOWNTO 0);
112     VARIABLE regc : UNSIGNED(23 DOWNTO 0);
        VARIABLE regd : UNSIGNED(23 DOWNTO 0);
114     VARIABLE opcode : UNSIGNED(7 DOWNTO 0);
        VARIABLE regstat : UNSIGNED(3 DOWNTO 0);
116     VARIABLE waitcount : INTEGER;

118


120     BEGIN
            -- Statemachine im RESET-Zustand
122         IF reset = '1' THEN

124             -- Ready-Ausgang auf L setzen
                ready_o <= '0';
126
                -- Variablen initialisieren
128             regstat := CONV_UNSIGNED(0, 4);
                waitcount := 0;
130
                -- Stateeinsprung festlegen
132             state <= state_init;

134         ELSIF clk'EVENT AND clk = '1' THEN

136             CASE state IS
                    -- Init State
138                 WHEN state_init =>

140                     -- warten auf Start-Signal
```

```
                    IF start_i = '1' THEN
142                     -- wenn Startsignal gegeben ->
                        --  Ready-Ausgang auf L setzen
144                     waitcount := 0;

146                     -- Input-Werte in Register speichern
                        rega := regop1_i;
148                     regb := regop2_i;
                        opcode := opcode_i;
150
                                                regstat := "0000";
152
                        -- Ready-Flag zurücksetzen
154                     ready_o <= '0';

156                     -- im nächsten State fortfahren
                        state <= state_decodeop;
158                 ELSE
                        -- in diesem State verweilen
160                     state <= state_init;
                    END IF;
162

164             WHEN state_decodeop =>

166                 CASE opcode IS
                        -- Addition
168                     WHEN CONV_UNSIGNED(01,8) =>
                            regc := rega + regb;
170                         state <= state_complete;

172                     -- Subtraktion
                        WHEN CONV_UNSIGNED(02,8) =>
174                         regc := rega - regb;
                            state <= state_complete;
176
                        -- Division
178                     WHEN CONV_UNSIGNED(04,8) =>
                            -- prüfen auf Division durch 0
180                         IF regb = 0 THEN
                                regstat := "0001";
182                             state <= state_complete;
                            ELSE
184                             state <= state_divide;
                            END IF;
186
                        -- undefinierter Opcode
188                     WHEN OTHERS =>
                            regstat := "1111";
190                         state <= state_complete;
                    END CASE;
192


194
                WHEN state_divide =>
196                 -- Verzögerungsschleife für Dividierer,
                    -- da er in einer Pipeline-Architektur realisiert ist
198                 IF waitcount = 10 THEN
                        state <= state_divideready;
200                 ELSE
                        state <= state_dividewait;
202                 END IF;

204             WHEN state_dividewait =>
                    waitcount := waitcount + 1;
206                 state <= state_divide;

208

210             WHEN state_divideready =>
                    -- Ergebnisse von Dividerer in Register speichern
```

```
212                     regc := UNSIGNED(divval);
                        regd := UNSIGNED(divrem);
214                     state <= state_complete;

216                 -- Complete-State
                    WHEN state_complete =>
218
                        -- Ready-Signal auf H setzen
220                     ready_o <= '1';
                        -- wieder in den Init-State zurückkehren
222                     state <= state_init;
                    WHEN others =>
224                     state <= state_init;

226             END CASE;
            END IF;
228
            -- Registerinhalte nach aussen führen
230         aval <= rega;
            bval <= regb;
232         rege1_o <= regc;
            rege2_o <= regd;
234         regstat_o <= CONV_STD_LOGIC_VECTOR(regstat,4);

236         -- Ende Statemachine

238     END PROCESS;

240
    -- Addierer/Subtrahierer verdrahten
242 lpm_addsub_24_24_inst : lpm_addsub_24_24 PORT MAP (
            add_sub   => addsub,
244         dataa     => CONV_STD_LOGIC_VECTOR(aval,24),
            datab     => CONV_STD_LOGIC_VECTOR(bval,24),
246         result    => addsubval,
            overflow      => addsubov
248     );

250 -- Dividierer verdrahten
    lpm_divide_24_24p_inst : lpm_divide_24_24p PORT MAP (
252         numer     => CONV_STD_LOGIC_VECTOR(aval,24),
            denom     => CONV_STD_LOGIC_VECTOR(bval,24),
254         clock     => clk,
            quotient      => divval,
256         remain    => divrem
        );
258
    END a;
```

B.3.3. Teilmodul Nulldurchgangssuche

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   FINDSIG                                                          --
--           versucht den ersten Nulldurchgang in dem Ultraschallsignal       --
--           zu finden                                                        --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - System Takt                                           --
--    reset           - system reset                                          --
--    start_i         - startet den Algorithmus                               --
--    datasrc_i       - 8bit Datenbus von Samplequelle                        --
--    findsigstart_i  - 10bit Startindex des relevanten Bereichs              --
--    findsigstop_i   - 10bit stopindex des relevanten Bereichs               --
--    mean_i          - 8bit Ganzzahl Mittelwert des Signals                  --
--                                                                            --
--    minarea_i       - 10bit Mindestfläche des Signals                       --
--    minperiod_i     - 8bit Mindestperiode des Signals                       --
--    maxperiod_i     - 8bit maximalperiode des Signals                       --
--    minampl_i       - 8bit Mindestamplitude des Signals                     --
--    alurege1_i      - 24bit Ergebnis Register 1 von ALU                     --
--    alurege2_i      - 24bit Ergebnis Register 2 von ALU                     --
--    alustatus_i     - 4bit status register von ALU                          --
--    aluready_i      - Ready Flag von ALU                                    --
--  Outputs:                                                                  --
--    ready_o         - zeigt das Ende des Algorithmus an                     --
--    dataaccess_o    - Flag zur Reservierung der Samplequelle                --
--    datasrcaddr_o   - 10bit Adressausgang zur Samplequelle                  --
--    errorcode_o     - 4bit Fehlercode                                       --
--    aluregop1_o     - 24bit ALU Operand Register 1                          --
--    aluregop2_o     - 24bit ALU Operand Register 2                          --
--    aluop_o         - 8bit ALU opcode register                              --
--    alusel_o        - Flag zur Reservierung der ALU                         --
--    alustart_o      - Flag zum Starten der ALU                              --
--    lower_o         - 10bit unterer Sampleindex Nullstellenübergang         --
--    upper_o         - 10bit oberer Sampleindex Nullstellenübergang          --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 24.09.2002                                                         --
--      Revision : a - 24.10.2002                                             --
--------------------------------------------------------------------------------



-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;


LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduledefinition
ENTITY findsig IS
    PORT(

--DEBUG
        dbgport         : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG
        clk             : IN    STD_LOGIC;
        reset           : IN    STD_LOGIC;
        start_i         : IN    STD_LOGIC;
        datasrc_i       : IN    UNSIGNED(7 DOWNTO 0);
        findsigstart_i  : IN    UNSIGNED(9 DOWNTO 0);
        findsigstop_i   : IN    UNSIGNED(9 DOWNTO 0);
        mean_i          : IN    UNSIGNED(7 DOWNTO 0);
```

```
70          minarea_i       : IN    UNSIGNED(9 DOWNTO 0);
            minperiod_i     : IN    UNSIGNED(7 DOWNTO 0);
72          maxperiod_i     : IN    UNSIGNED(7 DOWNTO 0);
            minampl_i       : IN    UNSIGNED(7 DOWNTO 0);
74          aluregel_i      : IN    UNSIGNED(23 DOWNTO 0);
            aluregel2_i     : IN    UNSIGNED(23 DOWNTO 0);
76          alustatus_i     : IN    UNSIGNED(3 DOWNTO 0);
            alueready_i     : IN    STD_LOGIC;
78
            ready_o         : OUT   STD_LOGIC;
80          dataaccess_o    : OUT   STD_LOGIC;
            datasrcaddr_o   : OUT   STD_LOGIC_VECTOR(9 DOWNTO 0);
82          errorcode_o     : OUT   UNSIGNED(3 DOWNTO 0);
            aluregop1_o     : OUT   UNSIGNED(23 DOWNTO 0);
84          aluregop2_o     : OUT   UNSIGNED(23 DOWNTO 0);
            aluop_o         : OUT   UNSIGNED(7 DOWNTO 0);
86          alusel_o        : OUT   STD_LOGIC;
            alustart_o      : OUT   STD_LOGIC;
88
            lower_o         : OUT   UNSIGNED(9 DOWNTO 0);
90          upper_o         : OUT   UNSIGNED(9 DOWNTO 0)
            );
92  END findsig;

94
    ARCHITECTURE a OF findsig IS
96
        -- Deklaration der States
98      TYPE STATE_TYPE IS ( state_init,
                             state_registerparam,state_checkparam,
100
                             state_integrate, state_checkarea,
102                          state_regfindleft,
                             state_prepfindleft, state_findright,
104                          state_findleft,
                             state_incright, state_decleft, state_decfindleft,
106                          state_checkwaveparam, state_compwaveparam,
                             state_minmaxiterate,
108                          state_prepfindminmax, state_findminmax,
                             state_iterate,
110                          state_calclowerupper,
                             state_complete);
112
        SIGNAL state: STATE_TYPE;                       -- Statevariable
114
    BEGIN
116
        -- Statemachine
118     PROCESS (clk, reset)

120     -- Sampleindexadresszählregister
        VARIABLE datasrcaddress : UNSIGNED(9 DOWNTO 0);
122
        -- temporärer Sampleindexspeicher
124     VARIABLE srcaddrtemp    : UNSIGNED(9 DOWNTO 0);

126     VARIABLE rightlimit     : UNSIGNED(9 DOWNTO 0);
        VARIABLE leftlimit      : UNSIGNED(9 DOWNTO 0);
128     VARIABLE waveperiod     : UNSIGNED(9 DOWNTO 0);

130     VARIABLE wavemax        : UNSIGNED(7 DOWNTO 0);
        VARIABLE wavemin        : UNSIGNED(7 DOWNTO 0);
132     VARIABLE waveampl       : UNSIGNED(7 DOWNTO 0);

134     -- Flag, das eine gültige Nullstelle gefunden wurde
        VARIABLE foundwave      : STD_LOGIC;
136
        -- Summationsregister für positiven Signalbereich
138     VARIABLE posarea        : UNSIGNED(17 DOWNTO 0);
        -- Summationsregister für negativen Signalbereich
140     VARIABLE negarea        : UNSIGNED(17 DOWNTO 0);
```

```
        -- Flag für positiven Signalbereich (> Mittelwert)
142     VARIABLE posflag        : STD_LOGIC;
        -- Flag für negativen Signalbereich (<= Mittelwert)
144     VARIABLE negflag        : STD_LOGIC;
        VARIABLE lower          : UNSIGNED(9 DOWNTO 0);
146     VARIABLE upper          : UNSIGNED(9 DOWNTO 0);

148     -- Register für Paramter
        VARIABLE mean           : UNSIGNED(7 DOWNTO 0);
150     VARIABLE minarea        : UNSIGNED(9 DOWNTO 0);
        VARIABLE minperiod      : UNSIGNED(7 DOWNTO 0);
152     VARIABLE maxperiod      : UNSIGNED(7 DOWNTO 0);
        VARIABLE minampl        : UNSIGNED(7 DOWNTO 0);
154     VARIABLE findsigstart   : UNSIGNED(9 DOWNTO 0);
        VARIABLE findsigstop    : UNSIGNED(9 DOWNTO 0);
156
        VARIABLE errorcode      : UNSIGNED(3 DOWNTO 0);
158

160     BEGIN
            -- asynchroner reset
162         IF reset = '1' THEN

164             -- setze Ready-Flag zurück
                ready_o <= '0';
166             -- setze Samplequellen- und ALU  Reservierung zurück
                dataaccess_o <= '0';
168             alusel_o <= '0';

170             -- initialer+ State setzen
                state <= state_init;
172
            -- positive Taktflanke
174         ELSIF clk'EVENT AND clk = '1' THEN

176             CASE state IS

178                 -- Init State
                    WHEN state_init =>
180
                        -- warte auf Start-Signal
182                     IF start_i = '1' THEN

184                         -- setze Ready-Flag zurück
                            ready_o <= '0';
186                         -- setze Fehlercode zurück
                            errorcode := "0000";
188                         -- setze Samplequellen- und ALU-Reservierung zurück
                            dataaccess_o <= '1';
190                         alusel_o <= '1';

192                         -- lösche alle temporären Berechnungsregister
                            foundwave := '0';
194                         posarea := CONV_UNSIGNED( 0, 18 );
                            negarea := CONV_UNSIGNED( 0, 18 );
196                         posflag := '0';
                            negflag := '0';
198

200                         -- mit Integrate State fortsetzen
                            state <= state_registerparam;
202                     ELSE
                            -- in diesem State verweilen
204                         state <= state_init;
                        END IF;
206
                    WHEN state_registerparam =>
208
                         -- lade Sampleindexzäähler mit Startwert vor
210                       datasrcaddress := findsigstart_i;
```

```
212                     -- Paramter in Register zwischenspeichern
                        mean     :=  mean_i;
214                     minarea :=  minarea_i;
                        minperiod := minperiod_i;
216                     maxperiod := maxperiod_i;
                        minampl := minampl_i;
218                     findsigstart := findsigstart_i;
                        findsigstop := findsigstop_i;
220
                        state <= state_checkparam;
222
                    WHEN state_checkparam =>
224
                        -- prüft, ob die Parameterwerte einen Sinn ergeben
226                     IF findsigstart > findsigstop THEN
                            errorcode := "0010";
228                         state <= state_complete;
                        ELSE
230                         state <= state_integrate;
                        END IF;
232
                    -- Integrate State
234                 WHEN state_integrate =>
                        -- prüfen, ob der aktuelle Samplewert größer als
236                     -- der Mittelwert ist,
                        -- wenn ja und der vorhergehende Samplewert kleiner
238                     -- als der Mittelwert war
                        -- (negativ-Flag gesetzt)-> lade das Register für
240                     -- Summation der positiven
                        -- Werte mit dem akutellen Wert vor und
242                     -- setzte die positiv-Flag

244                     -- wenn schon vorher im positiven
                        -- Bereich (positiv Flag gesetzt) ->
246                     -- aktueller Samplewert zur positiven Summe addieren
                        IF datasrc_i > mean THEN
248                         IF negflag = '1' THEN
                                -- Differenzbildung, weil relativ
250                             -- zum Mittelwert betrachtet wird
                                posarea := CONV_UNSIGNED(datasrc_i - mean,18);
252                             negflag := '0';

254                         ELSE

256                             -- Differenzbildung, weil relativ
                                -- zum Mittelwert betrachtet wird
258                             posarea := posarea + (datasrc_i - mean);
                            END IF;
260                         posflag := '1';

262
                        -- wenn kleiner und der vorhergehende Samplewert
264                     -- größer als der Mittelwert war
                        -- -> lade das Register für Summation der negativen
266                     -- Werte mit dem akutellen Wert vor und setzte
                        -- die negativ-Flag
268
                        -- wenn schon vorher im negativen
270                     -- Bereich (negativ Flag gesetzt) ->
                        -- aktueller Samplewert zur negativen Summe addieren
272
                        ELSE
274                         IF posflag = '1' THEN
                                -- Differenzbildung, weil relativ zum
276                             -- Mittelwert betrachtet wird
                                negarea := CONV_UNSIGNED(mean - datasrc_i,18);
278                             posflag := '0';

280                         ELSE
                                -- Differenzbildung, weil relativ zum
282                             -- Mittelwert betrachtet wird
```

```
                        negarea := negarea + (mean - datasrc_i);
284                 END IF;
                    negflag := '1';
286
                END IF;
288
                -- mit Checkarea State fortsetzen
290             state <= state_checkarea;

292         -- Checkarea-State
            WHEN state_checkarea =>
294
                -- wenn der aktuelle Samplewert größer
296             -- als der Mittelwert ist,
                -- prüfen ob, die vorhergehende negative Fläche einen
298             -- Mindestwert überschreitet,
                -- wenn ja, den aktuellen Sampleindex als vermeintlichen
300             -- Nullstellenbereich zwischenspeichern
                -- ansonsten mit nächstem Sample fortsetzen
302             IF (negarea > minarea) AND posflag = '1' THEN
                    srcaddrtemp := datasrcaddress;
304                 state <= state_findright;
                ELSE
306                 state <= state_iterate;
                END IF;
308
            WHEN state_findright =>
310
                -- findet die rechte Begrenzung des
312             -- interessanten Bereichs:
                -- solange das Sample positiv ist wird der
314             -- Indexzähler inkrementiert,
                -- bis entweder das Sample negativ wird oder der
316             -- gültige Bereich
                -- verlassen wurde
318             -- die rechte Begrenzung wird in
                -- rightlimit zwischengespeichert
320
                IF (datasrc_i > mean) AND
322                (datasrcaddress < findsigstop) THEN
                    state <= state_incright;
324             ELSE
                    rightlimit := datasrcaddress - 1;
326                 state <= state_regfindleft;
                END IF;
328
            WHEN state_incright =>
330             -- inkrementiert den Indexzähler und springt zurück
                -- in den vergleichenden State
332             datasrcaddress := datasrcaddress + 1;
                state <= state_findright;
334
            WHEN state_regfindleft =>
336             -- da der Adresszähler schon wieder
                -- im negativen Bereich steht,
338             -- wird er um 1 dekrementiert, um im
                -- positiven Bereich zu stehen
340             datasrcaddress := datasrcaddress - 1;
                state <= state_prepfindleft;
342
            WHEN state_prepfindleft =>
344
                -- Der Indexzähler wird solange dekrementiert, bis die
346             -- negative Fläche erreicht wurde
                IF( datasrc_i >= mean) AND
348               ( datasrcaddress > findsigstart) THEN
                    state <= state_decleft;
350             ELSE
                    state <= state_findleft;
352             END IF;
```

```
354                 WHEN state_decleft =>
                        datasrcaddress := datasrcaddress - 1;
356                     state <= state_prepfindleft;

358                 WHEN state_findleft =>
                        -- analog zu state_findright, es wird jedoch die
360                     -- linke Begrenzung gesucht und in leftlimit gespeichert

362                     -- die Register für wavemin und
                        -- wavemax werden initialisiert
364
                        IF (datasrc_i < mean) AND
366                         ( datasrcaddress > findsigstart) THEN
                            state <= state_decfindleft;
368                     ELSE
                            leftlimit := datasrcaddress + 1;
370                         wavemin := CONV_UNSIGNED( 255, 8 );
                            wavemax := CONV_UNSIGNED(   0, 8 );
372                         state <= state_prepfindminmax;
                        END IF;
374
                    WHEN state_decfindleft =>
376                     datasrcaddress := datasrcaddress - 1;
                        state <= state_findleft;
378
                    WHEN state_prepfindminmax =>
380                     -- starte die Suche nach Minimum und Maximum an der
                        -- unteren Grenze des interessanten Bereichs
382                     datasrcaddress := leftlimit;
                        state <= state_findminmax;
384
                    WHEN state_findminmax =>
386
                        -- der interessante Bereich wird durchiteriert
388                     -- wenn das Sample kleiner als wavemin, dieses Sample
                        -- als wavemin speichern
390                     -- wenn das Sample größer als wavemax ist, dieses Sample
                        -- als wavemax speichern
392                     IF datasrcaddress <= rightlimit THEN
                            IF datasrc_i < wavemin THEN
394                             wavemin := datasrc_i;
                            END IF;
396
                            IF datasrc_i > wavemax THEN
398                             wavemax := datasrc_i;
                            END IF;
400

402                         state <= state_minmaxiterate;
                        ELSE
404                         state <= state_checkwaveparam;
                        END IF;
406
                    WHEN state_minmaxiterate =>
408
                        datasrcaddress := datasrcaddress + 1;
410                     state <= state_findminmax;

412
                    WHEN state_checkwaveparam =>
414                     -- berechne die Wellenperiode durch Differenzbildung
                        -- von oberer und unterer Grenze des
416                     -- interessanten Bereichs
                        waveperiod := rightlimit - leftlimit;
418                     -- Spitze-Spitze Amplitude aus Maximum und Minimum
                        waveampl := (wavemax-mean)+(mean-wavemin);
420                     state <= state_compwaveparam;

422                 WHEN state_compwaveparam =>
                        -- prüfe, ob die Wellenformcharakteristika zutreffen,
424                     -- wenn ja, setzte foundwave und berechne die Nullstelle
```

```
                    -- ansonsten, setze das obere Ende des zuletzt
426                 -- untersuchten Bereichs als neue Startposition für
                    -- die weitere Suche nach Nullstellen
428
                    IF (waveperiod <= maxperiod) AND
430                     (waveperiod >= minperiod)
                        AND (waveampl >= minampl ) THEN
432                     foundwave := '1';
                        state <= state_calclowerupper;
434                 ELSE
                        datasrcaddress := rightlimit;
436                     negflag := '1';
                        posflag := '1';
438                     negarea := CONV_UNSIGNED(0,18);
                        posarea := CONV_UNSIGNED(0,18);
440                     state <= state_integrate;
                    END IF;
442
                WHEN state_iterate =>
444
                    -- wenn der obere Index des zu untersuchenden Bereichs
446                 -- erreicht wurde, in Complete State wechseln
                    -- ansonsten den Sampleindexzähler inkrementieren
448                 -- und wieder mit der Integration anfangen
                    IF datasrcaddress = findsigstop THEN
450                     state <= state_complete;
                    ELSE
452                     datasrcaddress :=datasrcaddress + 1;
                        state <= state_integrate;
454                 END IF;

456             WHEN state_calclowerupper =>
                    -- berechne die obere und untere Grenze
458                 -- des Nullstellenübergangs
                    -- anhand der zwischengespeicherten vermeintlichen
460                 -- Nullstelle
                    lower := srcaddrtemp - CONV_UNSIGNED(1,10);
462                 upper := srcaddrtemp;
                    state <= state_complete;
464
                -- Complete-State
466             WHEN state_complete =>

468                 -- wenn im gesamten Bereich keine Nullstelle gefunden
                    -- wurde, setze den Fehlercode
470                 IF foundwave = '0' THEN
                                            lower := CONV_UNSIGNED(0,10);
472                                         upper := CONV_UNSIGNED(0,10);
                                        errorcode := "0001";
474                 END IF;

476                 -- setze Ready-Flag
                    ready_o <= '1';
478                 -- setze Samplequellen- und ALU Zugriff zurück
                    dataaccess_o <= '0';
480                 alusel_o <= '0';

482                 -- die State-Machine neu starten
                    state <= state_init;
484
                WHEN others =>
486                 state <= state_init;

488         END CASE;
        END IF;
490     -- Ende Statemachine

492     -- verbinde sampleindexzähler mit Adressausgang für Samplequelle
        datasrcaddr_o <= CONV_STD_LOGIC_VECTOR(datasrcaddress,10 );
494     -- verbinde die Ergebnisregister mit den Ergebnisausgängen
        lower_o <= lower;
```

```
496             upper_o <= upper;
                errorcode_o <= errorcode;
498
        END PROCESS;
500
    END a;
```

B.3.4. Teilmodul Interpolation

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   INTERPOLATE                                                      --
--           interpoliert die wahre nullstelle aus der gelieferten oberen-----
--           und unteren Grenze                                               --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - system takt                                           --
--    reset           - system reset                                          --
--    start_i         - Start-Signal                                          --
--    datasrc_i       - 8bit Datenbus von Samplequelle                        --
--    lower_i         - 10bit unterer Index der Nullstelle                    --
--    upper_i         - 10bit oberer Index der Nullstelle                     --
--    mean16_i        - 16bit fixedpoint Mittelwert des Signals               --
--    polcount_i      - 4bit Anzahl der Interpolationstiefe                   --
--    aluregе1_i      - 24bit Ergebnis Register 1 von ALU                     --
--    aluregе2_i      - 24bit Ergebnis Register 2 von ALU                     --
--    aluready_i      - Ready-Signal von ALU                                  --
--    austatus_i      - Status-Register von ALU                               --
--  Outputs:                                                                  --
--    ready_o         - Ready-Flag, wenn Prozess beendet ist                  --
--    errorcode_o     - 4bit Fehlercode                                       --
--    dataaccess_o    - zeigt die Benutzung der Samplequelle an               --
--    datasrcaddr_o   - 10bit Adressindex der Samplequelle                    --
--    alurega_o       - 24bit Operand Register 1 für ALU                      --
--    aluregb_o       - 24bit Operand Register 2 für ALU                      --
--    alustart_o      - Starte ALU Berechnung                                 --
--    aluop_o         - 8bit ALU opcode                                       --
--    alusel_o        - zeigt die Benutzung der ALU an                        --
--    ustimeindex_o   - 20bit fixedpoint index der intepolierten              --
--                      Nullstelle                                            --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 27.09.2002                                                         --
--      Revision : a - 25.10.2002                                             --
--------------------------------------------------------------------------------



-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduldefinition
ENTITY interpolate IS
    PORT(
--DEBUG
        dbgport          : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG

        clk              : IN    STD_LOGIC;
        reset            : IN    STD_LOGIC;
        start_i          : IN    STD_LOGIC;
        datasrc_i        : IN    UNSIGNED(7 DOWNTO 0);
        lower_i          : IN    UNSIGNED(9 DOWNTO 0);
        upper_i          : IN    UNSIGNED(9 DOWNTO 0);
        mean16_i         : IN    UNSIGNED(15 DOWNTO 0);
        polcount_i       : IN    UNSIGNED(3 DOWNTO 0);

        aluregе1_i       : IN    UNSIGNED(23 DOWNTO 0);
        aluregе2_i       : IN    UNSIGNED(23 DOWNTO 0);
        alustatus_i      : IN    UNSIGNED(3 DOWNTO 0);
```

```
70          aluready_i          : IN    STD_LOGIC;

72          alurega_o           : OUT   UNSIGNED(23 DOWNTO 0);
            aluregb_o           : OUT   UNSIGNED(23 DOWNTO 0);
74          alustart_o          : OUT   STD_LOGIC;
            aluop_o             : OUT   UNSIGNED(7 DOWNTO 0);
76          alusel_o            : OUT   STD_LOGIC;

78
            dataaccess_o        : OUT   STD_LOGIC;
80          ustimeindex_o       : OUT   UNSIGNED(19 DOWNTO 0);
            datasrcaddr_o       : OUT   STD_LOGIC_VECTOR(9 DOWNTO 0);
82          errorcode_o         : OUT   UNSIGNED(3 DOWNTO 0);
            ready_o             : OUT   STD_LOGIC
84          );
END interpolate;
86

88  ARCHITECTURE a OF interpolate IS

90
    -- Deklaration der States
92  TYPE STATE_TYPE IS ( state_init,
                         state_reg,
94                       state_prepsumlower, state_sumlower,
                         state_ssumlower, state_postsumlower,
96                       state_prepsumvallower, state_sumvallower,
                         state_ssumvallower, state_postsumvallower,
98                       state_prepsumupper, state_sumupper,
                         state_ssumupper, state_postsumupper,
100                      state_prepsumvalupper, state_sumvalupper,
                         state_ssumvalupper, state_postsumvalupper,
102                      state_checksumend,
                         state_prepcalcindexmean, state_calcindexmean,
104                      state_scalcindexmean, state_postcalcindexmean,
                         state_prepcalcvalmean, state_calcvalmean,
106                      state_scalcvalmean, state_postcalcvalmean,
                         state_prepcalcdeltaval, state_calcdeltaval,
108                      state_scalcdeltaval, state_postcalcdeltaval,
                         state_prepcalcdeltaindex, state_calcdeltaindex,
110                      state_scalcdeltaindex, state_postcalcdeltaindex,
                         state_prepcalcm, state_calcm, state_scalcm,
112                      state_postcalcm,
                         state_prepcalcb, state_calcb, state_scalcb,
114                      state_postcalcb,
                         state_prepcalcns, state_calcns, state_scalcns,
116                      state_postcalcns,
                         state_prepcalctime, state_calctime,
118                      state_scalctime, state_postcalctime,
                         state_complete );
120
    SIGNAL state: STATE_TYPE;                    -- Statevariable
122

124 -- auf 16bit erweiterter Samplewert
    SIGNAL datasrc16 : UNSIGNED(15 DOWNTO 0);
126 -- auf 20bit erweiterter Nullstellenbereich
    SIGNAL upper20 : UNSIGNED(19 DOWNTO 0);
128 SIGNAL lower20 : UNSIGNED(19 DOWNTO 0);

130 -- auf 20bit erweiterte Interpolation
    SIGNAL interpol20 : UNSIGNED(19 DOWNTO 0);
132

134 BEGIN

136 sigs:

138 -- erweitert den aktuellen Samplewert von 8bit auf 16bit
    -- 8bit Ganzzahl | 8bit Nachkomma
140 datasrc16(15 DOWNTO 8) <= datasrc_i(7 DOWNTO 0);
```

```
        datasrc16(7 DOWNTO 0) <= "00000000";
142
        -- erweitert den Nullstellenindex von 10bit auf 20bit
144     -- 10bit Ganzzahl | 10bit Nachkomma
        upper20(19 DOWNTO 10) <= upper_i(9 DOWNTO 0);
146     upper20(9 DOWNTO 0) <= "0000000000";
        lower20(19 DOWNTO 10) <= lower_i(9 DOWNTO 0);
148     lower20(9 DOWNTO 0) <= "0000000000";

150     -- auf 20bit erweiterter aktueller Interpolationsschritt
        interpol20(19 DOWNTO 14) <= "000000";
152     interpol20(9 DOWNTO 0) <= "0000000000";

154
        -- Statemachine
156     PROCESS (clk,reset)

158     -- Register für aktuellen Interpolationsschritt
        VARIABLE interpolstep : UNSIGNED(3 DOWNTO 0);
160     -- 24bit Summenregister für untere und obere Sampleindizies
        -- 24bit -> von 10 auf 20bit erweiterte Länge + 4bit um Überlauf
162     -- zu vermeiden
        VARIABLE lowersum     : UNSIGNED(23 DOWNTO 0);
164     VARIABLE uppersum     : UNSIGNED(23 DOWNTO 0);
        -- 20bit Summenregister für untere und obere Samplewerte
166     -- 20bit -> von 8 auf 16bit erweiterte Länge + 4bit um Überlauf
        -- zu vermeiden
168     VARIABLE valuppersum  : UNSIGNED(19 DOWNTO 0);
        VARIABLE vallowersum  : UNSIGNED(19 DOWNTO 0);
170     -- Register für aktuellen Sampleindex
        VARIABLE datasrcaddress : UNSIGNED(9 DOWNTO 0);
172     -- Register für mittleren negativen Sampleindex
        VARIABLE indexmean    : UNSIGNED(23 DOWNTO 0);
174     -- Register für mittleren negativen Samplewert
        VARIABLE valmean      : UNSIGNED(19 DOWNTO 0);
176     -- Register für Wertedifferenz obere und untere Grenze
        VARIABLE dval         : UNSIGNED(19 DOWNTO 0);
178     -- Register für Indexdifferenz obere und untere Grenze
        VARIABLE dindex       : UNSIGNED(23 DOWNTO 0);
180     -- Register für Steigung
        VARIABLE dm           : UNSIGNED(19 DOWNTO 0);
182     -- Register für y-Achsenabschnitt
        VARIABLE deltamin     : UNSIGNED(15 DOWNTO 0);
184     -- Register für relativen Nulldurchgang
        VARIABLE dns          : UNSIGNED(15 DOWNTO 0);
186     -- Register für interpolierten Nullstellenindex
        VARIABLE ustime       : UNSIGNED(19 DOWNTO 0);
188
        -- Register für Parameter
190     VARIABLE polcount     : UNSIGNED(3 DOWNTO 0);
        VARIABLE lower        : UNSIGNED(9 DOWNTO 0);
192     VARIABLE upper        : UNSIGNED(9 DOWNTO 0);
        -- Register für Fehlercode
194     VARIABLE errorcode    : UNSIGNED(3 DOWNTO 0);

196     BEGIN
            -- asynchroner reset
198         IF reset = '1' THEN

200
                -- setze Ready-FLag zurück
202             ready_o <= '0';
                -- setzte Samplequellen- und ALU Benutzungsflag zurück
204             dataaccess_o <= '0';
                alusel_o <= '0';
206
                errorcode:= "0000";
208
                -- initialer State setzen
210             state <= state_init;
```

```
212         -- positive Taktflanke
            ELSIF clk'EVENT AND clk = '1' THEN
214
                CASE state IS
216
                    -- Init State
218                 WHEN state_init =>
                        -- warte auf Startsignal
220                     IF start_i = '1' THEN

222                         -- setze Ready-Flag zurück
                            ready_o <= '0';
224
                            -- setze Samplequellen- und ALU Benutzungsflags
226                         dataaccess_o <= '1';
                            alusel_o <= '1';
228
                            -- lösche alle temporären Berechnungsregister
230                         interpolstep := CONV_UNSIGNED(0,4);
                            lowersum := CONV_UNSIGNED(0, 24 );
232                         uppersum := CONV_UNSIGNED(0, 24 );
                            valuppersum := CONV_UNSIGNED(0, 20 );
234                         vallowersum := CONV_UNSIGNED(0, 20 );

236                                                 errorcode := "0000";
                            -- starte die Berechnung
238                         state <= state_reg;
                        ELSE
240                         -- verweile in diesem State
                            state <= state_init;
242                     END IF;

244
                    WHEN state_reg =>
246                     -- speichere alle Parameter in Register
                        polcount:= polcount_i;
248                     lower := lower_i;
                        upper := upper_i;
250                     state <= state_prepsumlower;

252                 WHEN state_prepsumlower =>
                        -- setze aktuellen Sampleindex auf
254                     -- (untere Nullstellengrenze - Interpolationsschritt)
                        -- addiere den auf 20bit erweiterten Index auf die Summe
256                     -- des negativen Index
                        datasrcaddress := lower - interpolstep;
258                     alurega_o <= CONV_UNSIGNED(lowersum, 24);
                        aluregb_o <= CONV_UNSIGNED( (lower20-interpol20), 24 );
260                     aluop_o <= CONV_UNSIGNED( 01,8);
                        state <= state_sumlower;
262

264                 WHEN state_sumlower =>
                        alustart_o <= '1';
266                     state <= state_ssumlower;

268                 WHEN state_ssumlower =>
                        alustart_o <= '0';
270                     state <= state_postsumlower;

272                 WHEN state_postsumlower =>
                        -- wenn ALU bereit ist, speichere das Ergebnis im
274                     -- negativen Summenregister
                        IF aluready_i = '1' THEN
276                         lowersum := CONV_UNSIGNED( aluregel_i, 24 );
                            state <= state_prepsumvallower;
278                     END IF;

280                 WHEN state_prepsumvallower =>
                        -- addiere den auf 16bit erweiterten Samplewert auf die
282                     -- Summe der negativen Werte
```

```
                    aluregа_o <= CONV_UNSIGNED(vallowersum, 24);
284                 aluregb_o <= CONV_UNSIGNED( datasrc16, 24 );
                    aluop_o <= CONV_UNSIGNED( 01,8);
286
                    state <= state_sumvallower;
288
                WHEN state_sumvallower =>
290                 alustart_o <= '1';
                    state <= state_ssumvallower;
292
                WHEN state_ssumvallower =>
294                 alustart_o <= '0';
                    state <= state_postsumvallower;
296
                WHEN state_postsumvallower =>
298                 IF alureadу_i = '1' THEN
                        -- wenn ALU bereit ist, speichere das Ergebnis im
300                     -- negativen Werteregister
                        vallowersum := CONV_UNSIGNED( alurege1_i, 20 );
302                     state <= state_prepsumupper;
                    END IF;
304
                WHEN state_prepsumupper =>
306                 -- setze aktuellen Sampleindex auf
                    -- (obere Nullstellengrenze + Interpolationsschritt)
308                 -- addiere den auf 20bit erweiterten Index auf die
                    -- Summe des positiven Index
310                 datasrcaddress := upper + interpolstep;
                    aluregа_o <= CONV_UNSIGNED(uppersum, 24);
312                 aluregb_o <= CONV_UNSIGNED( (upper20+interpol20), 24 );
                    aluop_o <= CONV_UNSIGNED( 01,8);
314
                    state <= state_sumupper;
316
                WHEN state_sumupper =>
318                 alustart_o <= '1';
                    state <= state_ssumupper;
320
                WHEN state_ssumupper =>
322                 alustart_o <= '0';
                    state <= state_postsumupper;
324
                WHEN state_postsumupper =>
326                 -- wenn ALU bereit ist, speichere das Ergebnis im
                    -- positiven Summenindexregister
328                 IF alureadу_i = '1' THEN
                        uppersum := CONV_UNSIGNED( alurege1_i, 24 );
330                     state <= state_prepsumvalupper;
                    END IF;
332
                WHEN state_prepsumvalupper =>
334                 -- addiere den auf 16bit erweiterten Samplewert auf die
                    -- Summe der positiven Werte
336                 aluregа_o <= CONV_UNSIGNED(valuppersum, 24);
                    aluregb_o <= CONV_UNSIGNED( datasrc16, 24 );
338                 aluop_o <= CONV_UNSIGNED( 01,8);

340                 state <= state_sumvalupper;

342             WHEN state_sumvalupper =>
                    alustart_o <= '1';
344                 state <= state_ssumvalupper;

346             WHEN state_ssumvalupper =>
                    alustart_o <= '0';
348                 state <= state_postsumvalupper;

350             WHEN state_postsumvalupper =>
                    -- wenn ALU bereit ist, speichere das Ergebnis im
352                 -- positiven Werteregister
                    IF alureadу_i = '1' THEN
```

```
354                         valuppersum := CONV_UNSIGNED( alurege1_i, 20 );
                            state <= state_checksumend;
356                     END IF;

358
                WHEN state_checksumend =>
360                 -- wenn die Anzahl der Interpolationsschritte erreicht
                    -- wurde, fahre mit Mittelwertbildung fort
362                 -- ansonsten inkrementiere den
                    -- Interpolationsschrittzähler und
364                 -- summiere erneut auf
                    IF interpolstep = polcount THEN
366                     interpolstep := CONV_UNSIGNED(0,4);
                        polcount := polcount +1;
368                     state <= state_prepcalcindexmean;
                    ELSE
370                     interpolstep := interpolstep + 1;
                        state <= state_prepsumlower;
372                 END IF;

374
                WHEN state_prepcalcindexmean =>
376                 -- berechne den Mittelwert der negativen Sampleindizies
                    -- lowersum/polcount
378                 aluega_o <= CONV_UNSIGNED(lowersum,24);
                    alurega_o <= CONV_UNSIGNED(polcount, 24 );
380                 aluop_o   <= CONV_UNSIGNED(04,8 );
                    state <= state_calcindexmean;
382
                WHEN state_calcindexmean =>
384                 alustart_o <= '1';
                    state <= state_scalcindexmean;
386
                WHEN state_scalcindexmean =>
388                 alustart_o <= '0';
                    state <= state_postcalcindexmean;
390
                WHEN state_postcalcindexmean =>
392                 -- wenn ALU bereit, speichere Ergebnis in Register für
                    -- mittleren negativen Sampleindex
394                 IF aluready_i = '1' THEN
                        IF alustatus_i = CONV_UNSIGNED(0,4) THEN
396                         indexmean := CONV_UNSIGNED(alurege1_i, 24 );
                            state <= state_prepcalcvalmean;
398                     ELSE
                            errorcode := "0001";
400                         state <= state_complete;
                        END IF;
402
                    END IF;
404
                WHEN state_prepcalcvalmean =>
406                 -- berechne den Mittelwert der negativen Samplewerte
                    -- vallowersum/polcount
408                 aluega_o <= CONV_UNSIGNED(vallowersum,24);
                    alurega_o <= CONV_UNSIGNED(polcount, 24 );
410                 aluop_o   <= CONV_UNSIGNED(04,8 );
                    state <= state_calcvalmean;
412
                WHEN state_calcvalmean =>
414                 alustart_o <= '1';
                    state <= state_scalcvalmean;
416
                WHEN state_scalcvalmean =>
418                 alustart_o <= '0';
                    state <= state_postcalcvalmean;
420
                WHEN state_postcalcvalmean =>
422                 -- wenn ALU bereit, speichere Ergebnis in Register für
                    -- mittlere negative Samplewert
424                 IF aluready_i = '1' THEN
```

```
                    IF alustatus_i = CONV_UNSIGNED(0,4) THEN
426                     valmean := CONV_UNSIGNED(alurege1_i, 20 );
                        state <= state_prepcalcdeltaval;
428                 ELSE
                        errorcode := "0010";
430                     state <= state_complete;
                    END IF;
432
                END IF;
434

436             WHEN state_prepcalcdeltaval =>
                    -- bilde die Differenz aus oberer und unterer Wertesumme
438                 aluregа_o <= CONV_UNSIGNED(valuppersum, 24);
                    aluregb_o <= CONV_UNSIGNED(vallowersum, 24 );
440                 aluop_o <= CONV_UNSIGNED( 02,8);

442                 state <= state_calcdeltaval;

444             WHEN state_calcdeltaval =>
                    alustart_o <= '1';
446                 state <= state_scalcdeltaval;

448             WHEN state_scalcdeltaval =>
                    alustart_o <= '0';
450                 state <= state_postcalcdeltaval;

452             WHEN state_postcalcdeltaval =>
                    -- wenn ALU bereit, speichere Ergebnis in Register
454                 -- für Wertedifferenz
                    IF alureadу_i = '1' THEN
456                     dval := CONV_UNSIGNED( alurege1_i, 20 );
                        state <= state_prepcalcdeltaindex;
458                 END IF;

460             WHEN state_prepcalcdeltaindex =>
                    -- berechne die Differenz aus oberer unterer Indexsumme
462                 aluregа_o <= CONV_UNSIGNED(uppersum, 24);
                    aluregb_o <= CONV_UNSIGNED(lowersum, 24 );
464                 aluop_o <= CONV_UNSIGNED( 02,8);

466                 state <= state_calcdeltaindex;

468             WHEN state_calcdeltaindex =>
                    alustart_o <= '1';
470                 state <= state_scalcdeltaindex;

472             WHEN state_scalcdeltaindex =>
                    alustart_o <= '0';
474                 state <= state_postcalcdeltaindex;

476             WHEN state_postcalcdeltaindex =>
                    -- wenn ALU bereit, speichere Ergebnis in Register für
478                 -- Indexdifferenz
                    IF alureadу_i = '1' THEN
480                     dindex := CONV_UNSIGNED( alurege1_i, 24 );
                        state <= state_prepcalcm;
482                 END IF;

484             WHEN state_prepcalcm =>
                    -- berechne die Steigung aus Wertedifferenz(y)
486                 -- und Indexdifferenz(x)
                    -- Wertedifferenz/Indexdifferenz
488                 aluregа_o <= CONV_UNSIGNED(dval,24);
                    aluregb_o <= CONV_UNSIGNED(dindex, 24 );
490                 aluop_o   <= CONV_UNSIGNED(04,8 );
                    state <= state_calcm;
492
                WHEN state_calcm =>
494                 alustart_o <= '1';
                    state <= state_scalcm;
```

```
496
                WHEN state_scalcm =>
498                 alustart_o <= '0';
                    state <= state_postcalcm;
500
                WHEN state_postcalcm =>
502                 -- wenn ALU bereit, speichere Ergebnis in Register
                    -- für Steigung
504                 IF alureadY_i = '1' THEN
                        IF alustatus_i = CONV_UNSIGNED(0,4) THEN
506                         dm := CONV_UNSIGNED(alurege1_i, 20 );
                            state <= state_prepcalcb;
508                     ELSE
                            errorcode := "0011";
510                         state <= state_complete;
                        END IF;
512                 END IF;


514
                WHEN state_prepcalcb =>
516                 -- berechne den y-Achsenabschnitt durch Subtraktion der
                    -- interpolierten
518                 -- negativen Summe vom Signalmittelwert (Nullpunkt)

520                 aluega_o <= CONV_UNSIGNED(mean16_i,24);
                    aluregb_o <= CONV_UNSIGNED(valmean, 24 );
522                 aluop_o   <= CONV_UNSIGNED(02,8 );
                    state <= state_calcb;
524
                WHEN state_calcb =>
526                 alustart_o <= '1';
                    state <= state_scalcb;
528
                WHEN state_scalcb =>
530                 alustart_o <= '0';
                    state <= state_postcalcb;
532
                WHEN state_postcalcb =>
534                 -- wenn ALU bereit, speichere Ergebnis in Register
                    -- für y-Achsenabschnitt
536                 IF alureadY_i = '1' THEN
                        deltamin := CONV_UNSIGNED(alurege1_i, 16 );
538                     state <= state_prepcalcns;
                    END IF;
540
                WHEN state_prepcalcns =>
542                 -- berechnet den interpolierten relativen Nulldurchgang
                    -- des Signals durch Division von y-Achsenabschnitt
544                 -- durch Steigung
                    aluega_o <= CONV_UNSIGNED(deltamin,24);
546                 aluregb_o <= CONV_UNSIGNED(dm, 24 );
                    aluop_o   <= CONV_UNSIGNED(04,8 );
548                 state <= state_calcns;

550             WHEN state_calcns =>
                    alustart_o <= '1';
552                 state <= state_scalcns;

554             WHEN state_scalcns =>
                    alustart_o <= '0';
556                 state <= state_postcalcns;

558             WHEN state_postcalcns =>
                    -- wenn ALU bereit, speichere Ergebnis in Register
560                 -- für relativen Nulldurchgang
                    IF alureadY_i = '1' THEN
562                     IF alustatus_i = CONV_UNSIGNED(0,4) THEN
                            dns := CONV_UNSIGNED(alurege1_i, 16 );
564                         state <= state_prepcalctime;
                        ELSE
566                         errorcode := "0100";
```

```
                                state <= state_complete;
568                         END IF;
                        END IF;
570
                    WHEN state_prepcalctime =>
572                     -- addiere den intepolierten Nullstellenindex zum
                        -- relativen Nulldurchgang
574                     -- um absoluten Nulldurchgang zu berechnen
                        aluregа_o <= CONV_UNSIGNED(dns,24);
576                     aluregb_o <= CONV_UNSIGNED(indexmean, 24 );
                        aluop_o   <= CONV_UNSIGNED(01,8 );
578                     state <= state_calctime;

580                 WHEN state_calctime =>
                        alustart_o <= '1';
582                     state <= state_scalctime;

584                 WHEN state_scalctime =>
                        alustart_o <= '0';
586                     state <= state_postcalctime;

588                 WHEN state_postcalctime =>
                        -- wenn ALU bereit, speichere Ergebnis in Register
590                     -- für interpolierten Nullstellenindex
                        IF alueready_i = '1' THEN
592                         ustime := CONV_UNSIGNED(alurege1_i, 20 );
                            state <= state_complete;
594                     END IF;

596
                    -- Complete-State
598                 WHEN state_complete =>

600                     -- setze Ready-Flag
                        ready_o <= '1';
602                     -- setze Samplequellen- und ALU Benutzungsflags zurück
                        dataaccess_o <= '0';
604                     alusel_o <= '0';
                        -- wieder in den Init-State zurückkehren
606                     state <= state_init;

608                 WHEN others =>
                        state <= state_init;
610

612             END CASE;
            END IF;
614

616
            -- auf 20bit erweiterter aktueller Interpolationsschritt
618         interpol20(13 DOWNTO 10) <= interpolstep;

620         -- Ausgabe der interpolierten Nullstelle
            ustimeindex_o <= ustime;
622         -- Ausgabe des Sampleindex auf Adressausgang
            datasrcaddr_o <= CONV_STD_LOGIC_VECTOR(datasrcaddress, 10);
624         -- Ausgabe des Fehlercodes
            errorcode_o <= errorcode;
626
            -- End Statemachine
628
        END PROCESS;
630

632 END a;
```

B.3.5. Steuermodul Signalverarbeitung

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   SIGCONTROLLER                                                    --
--           steuert den Ablauf der Signalverarbeitungskette                  --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - System Takt                                           --
--    reset           - System Reset                                          --
--    start_i         - Startflag um die Signalverarbeitung zu starten        --
--    readymean_i     - Ready-Flag von Mittelwertbildung                      --
--    errmean_i       - 4bit Fehlercode von Mittelwertbildung                 --
--    readyfindsig_i  - Ready-Flag von Nullstellensuche                       --
--    errfindsig_i    - 4bit Fehlercode von Nullstellensuche                  --
--    readyinterpol_i - Ready-Flag von Interpolation                          --
--    errinterpol_i   - 4bit Fehlercode von Interpolation                     --
--  Outputs:                                                                  --
--    ready_o         - Ready-Flag, zeigt Ende der Signalverarbeitung         --
--    running_o       - aktiv, solange Signalverarbeitung arbeitet            --
--    errorcode_o     - 4bit Fehlercode                                       --
--    datasrcsel_o    - zeigt Benutzung der Samplequelle an                   --
--    rammux_o        - 2bit Multiplexeradresse für Samplequelle              --
--    alusel_o        - zeigt Benutzung der externen ALU an                   --
--    alumux_o        - 2bit Multiplexeradresse für externe ALU               --
--    startmean_o     - Start-Flag für Mittelwertbildung                      --
--    startfindsig_o  - Start-Flag für Nullstellensuche                       --
--    startinterpol_o - Start-Flag für Interpolation                          --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 11.10.2002                                                         --
--      Revision: a - 24.10.2002                                              --
--                b - 02.12.2002                                              --
--                                                                            --
--------------------------------------------------------------------------------



-- Load standard Libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduldefinition
ENTITY sigcontroller IS
    PORT(
--DEBUG
        dbgport          : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG

        clk              : IN    STD_LOGIC;
        reset            : IN    STD_LOGIC;
        start_i          : IN    STD_LOGIC;
        readymean_i      : IN    STD_LOGIC;
        errmean_i        : IN    UNSIGNED(3 DOWNTO 0);
        readyfindsig_i   : IN    STD_LOGIC;
        errfindsig_i     : IN    UNSIGNED(3 DOWNTO 0);
        readyinterpol_i  : IN    STD_LOGIC;
        errinterpol_i    : IN    UNSIGNED(3 DOWNTO 0);

        ready_o          : OUT   STD_LOGIC;
        running_o        : OUT   STD_LOGIC;
        errorcode_o      : OUT   UNSIGNED(3 DOWNTO 0);
        datasrcsel_o     : OUT   STD_LOGIC;
        rammux_o         : OUT   UNSIGNED(1 DOWNTO 0);
```

```
70          alusel_o          : OUT   STD_LOGIC;
            alumux_o          : OUT   UNSIGNED(1 DOWNTO 0);
72          startmean_o       : OUT   STD_LOGIC;
            startfindsig_o    : OUT   STD_LOGIC;
74          startinterpol_o   : OUT   STD_LOGIC
            );
76  END sigcontroller;

78
    ARCHITECTURE a OF sigcontroller IS
80

82      -- Deklaration der States
        TYPE STATE_TYPE IS ( state_init,
84                           state_startmean, state_poststartmean,
                             state_waitmean,
86                           state_startfindsig, state_poststartfindsig,
                             state_waitfindsig,
88                           state_startinterpol, state_poststartinterpol,
                             state_waitinterpol,
90                           state_complete );

92      SIGNAL state: STATE_TYPE;                          -- Statevariable

94
    BEGIN
96
        -- Statemachine
98      PROCESS (clk, reset)

100     -- Fehlercode Register
        VARIABLE errorcode  : UNSIGNED(3 DOWNTO 0);
102     -- Zähler für Timeout
        VARIABLE timeoutcount : UNSIGNED(15 DOWNTO 0);
104         VARIABLE alumux          : UNSIGNED(1 DOWNTO 0);

106     BEGIN
            -- asynchroner Reset
108         IF reset = '1' THEN

110             -- setze Ready- und Running-Flags zurück
                ready_o <= '0';
112             running_o <= '0';

114             -- setze alle Start-Flags zurück
                startmean_o <= '0';
116             startfindsig_o <= '0';
                startinterpol_o <= '0';
118
                -- setze Samplequellen- und ALU Benutzungsflags zurück
120             datasrcsel_o <= '0';
                alusel_o <= '0';
122
                 -- initialer State setzen
124             state <= state_init;

126         -- positive Taktflanke
            ELSIF clk'EVENT AND clk = '1' THEN
128
                CASE state IS
130
                    -- Init State
132                 WHEN state_init =>
                        -- warte auf Start-Flag
134                     IF start_i = '1' THEN
                            -- setze Ready-Flag zurück
136                         ready_o <= '0';

138                         -- setze Fehlercode zurück
                            errorcode := "0000";
140                         -- setzte Timeout-Zähler zurück
```

```
                    timeoutcount := CONV_UNSIGNED(0,16);
142
                    -- setze alle Start-Flags zurück
144                 startmean_o <= '0';
                    startfindsig_o <= '0';
146                 startinterpol_o <= '0';

148                 -- setze running flag
                    running_o <= '1';
150

152                     alumux := "00";
                    -- setze Samplequellen- und ALU Benutzungsflags
154                 datasrcsel_o <= '1';
                    alusel_o <= '0';
156
                    -- beginne mit startmean State
158                 state <= state_startmean;
                ELSE
160                 -- verweile in Init-State
                    state <= state_init;
162             END IF;

164         -- Startmean State
            WHEN state_startmean =>
166             -- schalte Samplequellen- und ALU Multiplexer
                -- auf das Mittelwert-Modul
168             rammux_o <= "00";
                alumux := "00";
170             -- setze Start-Flag für Mittelwert-Modul
                startmean_o <= '1';
172
                state <= state_poststartmean;
174
            -- poststartmean State
176         WHEN state_poststartmean =>
                startmean_o <= '0';
178             state <= state_waitmean;

180         -- waitmean-State
            WHEN state_waitmean =>
182             -- warte auf Ready-Signal von Mittelwert-Modul
                -- wenn kein Fehler aufgetreten ist, fahre mit
184             -- Nullstellensuche fort, ansonsten setze Fehlercode
                IF readymean_i = '1' THEN
186                 IF errmean_i = 0 THEN
                        state <= state_startfindsig;
188                 ELSE
                        errorcode := "0001";
190                     state <= state_complete;
                    END IF;
192             ELSE
                    state <= state_waitmean;
194             END IF;

196         WHEN state_startfindsig =>
                -- schalte Samplequellen- und ALU Multiplexer
198             -- auf das Nullstellensuche-Modul
                rammux_o <= "01";
200             alumux := "01";
                startfindsig_o <= '1';
202             state <= state_poststartfindsig;

204         WHEN state_poststartfindsig =>
                startfindsig_o <= '0';
206             state <= state_waitfindsig;

208         WHEN state_waitfindsig =>
                -- warte auf Ready-Signal von Nullstellensuche-Modul
210             -- wenn kein Fehler aufgetreten ist, fahre mit
                -- Interpolation fort, ansonsten setze Fehlercode
```

```
212                     -- wenn keine Nullstelle innerhalb 300us gefunden
                        -- wurde, setzte Fehlercode auf Timeout
214
                        IF readyfindsig_i = '1' THEN
216
                            IF errfindsig_i = 0 THEN
218                             state <= state_startinterpol;
                            ELSE
220                             errorcode := "0010";
                                state <= state_complete;
222                         END IF;
                        ELSE
224                         -- 300us Timeout
                            IF timeoutcount = 6000 THEN
226                             errorcode := "0011";
                                state <= state_complete;
228                         ELSE
                                timeoutcount := timeoutcount + 1;
230                             state <= state_waitfindsig;
                            END IF;
232                     END IF;

234                 WHEN state_startinterpol =>
                        -- schalte Samplequellen- und ALU Multiplexer
236                     -- auf Interpolations-Modul
                        rammux_o <= "10";
238                     alumux := "10";
                        startinterpol_o <= '1';
240                     state <= state_poststartinterpol;

242                 WHEN state_poststartinterpol =>
                        startinterpol_o <= '0';
244                     state <= state_waitinterpol;

246                 WHEN state_waitinterpol =>
                        -- warte auf Ready-Signal von Interpolations-Modul
248                     -- wenn kein Fehler aufgetreten ist, springe
                        -- in complete-State
250                     -- ansonsten setzte Fehlercode
                        IF readyinterpol_i = '1' THEN
252                         IF errinterpol_i = 0 THEN
                                state <= state_complete;
254                         ELSE
                                errorcode := "0100";
256                             state <= state_complete;
                            END IF;
258                     ELSE
                            state <= state_waitinterpol;
260                     END IF;


262
                    -- Complete-State
264                 WHEN state_complete =>

266                     -- setze Ready-Flag
                        ready_o <= '1';
268                     -- setze Samplequellen- und ALU Benutzungsflags zurück
                        datasrcsel_o <= '0';
270                     alusel_o <= '0';
                        -- setzte Running-Flag zurück
272                     running_o <= '0';

274                     -- wieder in den Init-State zurückkehren
                        state <= state_init;
276
                    WHEN others =>
278                     state <= state_init;

280             END CASE;
            END IF;
282
```

```
            -- gebe Inhalt von errorcode-Register aus
284         errorcode_o <= errorcode;
            -- Ende Statemachine
286                 alumux_o <= alumux;

288     END PROCESS;

290 END a;
```

B.3.6. Teilmodul Mittelwertbildung

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   MEANSHOTS                                                        --
--           summiert die arteriellen und venösen "Schüsse" und bildet        --
--           nach einer vorgegebenen Anzahl den Mittelwert der Schüsse        --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - System Takt                                           --
--    reset           - system reset                                          --
--    start_i         - Start-Flag um die Prozedur zu starten                 --
--    restart_i       - startet die Mittelwertbildung erneut                  --
--    samplestart_i   - startet die Summierung für einen Schuss               --
--    ustimeindex_i   - 20bit fixedpoint Sampleindex                          --
--    meancount_i     - 8bit Anzahl der Schüsse zur Mittelung                 --
--    artvenflag_i    - zeigt an, dass das aktuelle Sample vom                --
--                      venösen Kanal anliegt                                 --
--    aluregе1_i      - 24bit Result Register 1 für  ALU                      --
--    aluregе2_i      - 24bit Result Register 2 für  ALU                      --
--    alustatus_i     - Statusregister von ALU                                --
--    aluready_i      - Ready-Signal von ALU                                  --
--  Outputs:                                                                  --
--    ready_o          - Ready-FLag, wenn die Schüsse gemittelt wurden        --
--    readysample_o    - Ready-Flag, wenn ein Sample aufsummiert wurde        --
--    errorcode_o      - 4bit Fehlercode                                      --
--    aluregа_o        - 24bit Operand Register 1 for ALU                     --
--    aluregb_o        - 24bit Operand Register 2 for ALU                     --
--    alustart_o       - Starte ALU Operation                                 --
--    aluop_o          - 8bit ALU opcode                                      --
--    alusel_o         - Zeigt die Benutzung der ALU an                       --
--    usmeantimeart_o - 20bit fixedpoint Ergebnis Mittelung arteriell         --
--    usmeantimeven_o - 20bit fixedpoint Ergebnis Mittelung venös             --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 28.10.2002                                                         --
--      Revision : a - 02.12.2002                                             --
--------------------------------------------------------------------------------



-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduledefinition
ENTITY meanshots IS
    PORT(
--DEBUG
        dbgport             : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG
        clk                 : IN    STD_LOGIC;
        reset               : IN    STD_LOGIC;
        start_i             : IN    STD_LOGIC;
        samplestart_i       : IN    STD_LOGIC;
        restart_i           : IN    STD_LOGIC;
        ustimeindex_i       : IN    UNSIGNED(19 DOWNTO 0);
        meancount_i         : IN    UNSIGNED(7 DOWNTO 0);
        artvenflag_i        : IN    STD_LOGIC;
        errsig_i            : IN    UNSIGNED(3 DOWNTO 0);
        aluregе1_i          : IN    UNSIGNED(23 DOWNTO 0);
        aluregе2_i          : IN    UNSIGNED(23 DOWNTO 0);
        aluready_i          : IN    STD_LOGIC;
        alustatus_i         : IN    UNSIGNED(3 DOWNTO 0);
```

```
70
          alurega_o           : OUT   UNSIGNED(23 DOWNTO 0);
72        aluregb_o           : OUT   UNSIGNED(23 DOWNTO 0);
          alustart_o          : OUT   STD_LOGIC;
74        aluop_o             : OUT   UNSIGNED(7 DOWNTO 0);
          alusel_o            : OUT   STD_LOGIC;
76
          readysample_o       : OUT   STD_LOGIC;
78
          usmeantimeart_o     : OUT   UNSIGNED(19 DOWNTO 0);
80        usmeantimeven_o     : OUT   UNSIGNED(19 DOWNTO 0);

82        errorcode_o         : OUT   UNSIGNED(3 DOWNTO 0);
          ready_o             : OUT   STD_LOGIC
84        );
  END meanshots;
86

88  ARCHITECTURE a OF meanshots IS

90
      -- Deklaration der States
92    TYPE STATE_TYPE IS ( state_init,
                           state_reg,state_waitsample, state_chkerr,
94                         state_sumup,
                           state_checkend,
96                         state_divideart, state_saveart,
                           state_divideven, state_saveven,
98                         state_readysample,
                           state_complete );
100
      SIGNAL state: STATE_TYPE;                          -- Statevariable
102
      SIGNAL meanresult : UNSIGNED(26 DOWNTO 0);
104

106   SIGNAL errsig    : STD_LOGIC;

108 BEGIN

110
  errsighandler:
112     errsig <= errsig_i(0) OR errsig_i(1) OR errsig_i(2) OR errsig_i(3);

114
      -- Statemachine
116   PROCESS (clk,reset)

118   -- Register für artven-Flag
      VARIABLE artven          : STD_LOGIC;
120
      -- Register für aktuellen Index
122   VARIABLE ustimeindex     : UNSIGNED(19 DOWNTO 0);
      -- Register für Anzahl der zu mittelnden Werte (2^)
124   VARIABLE meancount       : UNSIGNED(7 DOWNTO 0);

126   -- Zähler für bereits aufsummierte Werte
      VARIABLE meanstepart     : UNSIGNED(7 DOWNTO 0);
128   VARIABLE meanstepven     : UNSIGNED(7 DOWNTO 0);

130   -- arterielles und venöses Summenregister
      -- 26bit -> 20bit + 6bit Reserve um Überlauf zu vermeiden
132   VARIABLE timeartsum      : UNSIGNED(26 DOWNTO 0);
      VARIABLE timevensum      : UNSIGNED(26 DOWNTO 0);
134
      -- Register für max. Anzahl der Summierungen (2^meancount)
136   VARIABLE sumendval       : UNSIGNED(7 DOWNTO 0);

138   -- Register für gemitteltes Ergebnis
      VARIABLE timeart         : UNSIGNED(19 DOWNTO 0);
140   VARIABLE timeven         : UNSIGNED(19 DOWNTO 0);
```

```
142
144     -- Register für Fehlercode
        VARIABLE errorcode      : UNSIGNED(3 DOWNTO 0);
146
        VARIABLE errsignal      : STD_LOGIC;
148
        BEGIN
150         -- asynchroner Reset
            IF reset = '1' THEN
152
                -- setzte Ready-Flag zurück
154             ready_o <= '0';
                -- setzte ReadySample-Flag zurück
156             readysample_o <= '0';

158             -- setzte Flag für ALU-Benutzung zurück
                alusel_o <= '0';
160
                -- initialer State
162             state <= state_init;

164         -- positive Taktflanke
            ELSIF clk'EVENT AND clk = '1' THEN
166
                CASE state IS
168
                    -- Init State
170                 WHEN state_init =>
                        -- warte auf Start-Signal
172                     IF start_i = '1' THEN

174                         -- setzte Ready-Flag zurück
                            ready_o <= '0';
176                         -- setzte ReadySample-Flag zurück
                            readysample_o <= '0';
178
                            -- setzte Flag für ALU-Benutzung zurück
180                         alusel_o <= '0';

182                         -- setzte Fehlercode zurück
                            errorcode := "0000";
184                         -- beginne mit Berechnung
                            state <= state_reg;
186                     ELSE
                            -- verweile in diesem State
188                         state <= state_init;
                        END IF;
190

192                 WHEN state_reg =>
                        -- speichere Paramter in Register
194                     meancount := meancount_i;

196                     -- die Anzahl der zu mittelnden Werte MUSS 2^n
                        -- entsprechen, meancount_i entspricht n
198                     -- somit errechnet sich der Endwert der
                        -- Aufsummierschritte
200                     -- aus dem Nach links schieben von 1 um
                        -- meancount_i Stellen
202                     sumendval := SHL("00000001",
                                         CONV_UNSIGNED(meancount_i,3));
204
                        -- lösche alle temporären Register
206                     meanstepart := CONV_UNSIGNED(0,8);
                        meanstepven := CONV_UNSIGNED(0,8);
208                     timeartsum := CONV_UNSIGNED(0,27);
                        timevensum := CONV_UNSIGNED(0,27);
210
                        state <= state_waitsample;
```

```
212
                WHEN state_waitsample =>
214                 -- warte auf nächstes Sample
                    IF samplestart_i = '1' THEN
216                     -- setzte ReadySample-Flag zurück
                        readysample_o <= '0';
218                     -- setzte Flag für ALU Benutzung
                        alusel_o <= '1';
220
                        -- speichere Eingänge in Register
222                     artven := artvenflag_i;
                        ustimeindex := ustimeindex_i;
224                     errsignal := errsig;

226                     state <= state_chkerr;
                    ELSE
228                     -- verweile in diesem State
                        state <= state_waitsample;
230                 END IF;

232             WHEN state_chkerr =>
                    IF errsignal = '1' THEN
234                     ustimeindex := CONV_UNSIGNED(0,20);
                    END IF;
236                 state <= state_sumup;

238             WHEN state_sumup =>
                    -- wenn artven-Flag nicht gesetzt ist, addiere
240                 -- den aktuellen Wert zu dem arteriellen Summenregister
                    -- ansonsten addiere den aktuellen Wert zu dem venösen
242                 -- Summenregister
                    IF artven = '0' THEN
244                     -- nur addieren, wenn Anzahl der zu mittelnden
                        -- Samples
246                     -- noch nicht erreicht ist
                        IF  meanstepart = sumendval THEN
248                         null;
                        ELSE
250                         timeartsum := timeartsum + ustimeindex;
                            meanstepart := meanstepart + 1;
252                     END IF;
                    ELSE
254                     -- nur addieren, wenn Anzahl der zu
                        -- mittelnden Samples
256                     -- noch nicht erreicht ist
                        IF meanstepven = sumendval THEN
258                         null;
                        ELSE
260                         timevensum := timevensum + ustimeindex;
                            meanstepven := meanstepven + 1;
262                     END IF;
                    END IF;
264
                    state <= state_checkend;
266
                WHEN state_checkend =>
268                 -- wenn Anzahl der zu mittelnden Samples auf
                    -- arterieller UND
270                 -- venöser Seite erreicht ist, zur Mittelwertbildung
                    -- springen, ansonsten weitere Samples verarbeiten
272                 IF (meanstepart = sumendval) AND
                       (meanstepven = sumendval) THEN
274                     state <= state_divideart;
                    ELSE
276                     state <= state_readysample;
                    END IF;
278
                WHEN state_divideart =>
280                 -- bilde den Mittelwert der arteriellen Schüsse
                    -- durch Rechtsschieben um meancount
282                 -- (Division durch 2^meancount)
```

```
                    timeart := CONV_UNSIGNED(
284                         SHR(timeartsum, CONV_UNSIGNED(meancount,3)),20);
                    state <= state_saveart;
286
                WHEN state_saveart =>
288                 state <= state_divideven;

290             WHEN state_divideven =>
                    -- bilde den Mittelwert der venösen Schüsse
292                 -- durch Rechtsschieben um meancount
                    -- (Division durch 2^meancount)
294                 timeven := CONV_UNSIGNED(
                            SHR(timevensum, CONV_UNSIGNED(meancount,3)),20);
296                 state <= state_saveven;

298             WHEN state_saveven =>
                    state <= state_complete;
300

302             WHEN state_readysample =>
                    -- setzte Flag für ALU Benutzung zurück
304                 alusel_o <= '0';
                    -- setzte ReadySample-FLag
306                 readysample_o <= '1';
                    -- springe in "Warten auf nächstes Sample" State
308                 state <= state_waitsample;

310
                -- Complete-State
312             WHEN state_complete =>

314                 -- setze Ready-Flag
                    readysample_o <= '1';
316                 ready_o <= '1';
                    -- setzte Flag für ALU Benutzung zurück
318                 alusel_o <= '0';
                    -- wieder in den Init-State zurückkehren
320                 state <= state_init;

322             WHEN others =>
                    state <= state_init;
324         END CASE;
        END IF;
326
        -- gebe Ergebnisse auf Ausgang
328     usmeantimeart_o <= timeart;
        usmeantimeven_o <= timeven;
330
        -- gebe Fehlercoderegister auf Ausgang
332     errorcode_o <= errorcode;
        -- End Statemachine
334
    END PROCESS;
336
END a;
```

B.3.7. Teilmodul Laufzeitumrechnung

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   INDEX2TIME                                                       --
--           multipliziert die gemittelten arteriellen und venösen            --
--           Sampleindizies mit einer Zeitkonstanten, um die                  --
--           Ultraschallaufzeit zu errechnen                                  --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - system Takt                                           --
--    reset           - system reset                                          --
--    start_i         - startet den Prozess                                   --
--    timeindexart_i  - 20bit fixedpoint Sampleindex arteriell                --
--    timeindexven_i  - 20bit fixedpoint Sampleindex venös                    --
--    timeconst_i     - 6bit Zeitbasis -> 5bit Ganzzahl | 1bit nachkomma      --
--    alurege1_i      - 24bit Result Register 1 von  ALU                      --
--    alurege2_i      - 24bit Result Register 2 von  ALU                      --
--    alustatus_i     - 4bit Statusregister von ALU                           --
--    aluready_i      - Ready-Signal von  ALU                                 --
--  Outputs:                                                                  --
--    ready_o           - Ready-Flag für das Ende der Berechnung              --
--    errorcode_o       - 4bit Fehlercode                                     --
--    alurega_o         - 24bit Operand Register 1 für ALU                    --
--    aluregb_o         - 24bit Operand Register 2 für ALU                    --
--    alustart_o        - Starte ALU Berechnung                               --
--    aluop_o           - 8bit ALU opcode                                     --
--    alusel_o          - zeigt die Benutzung der ALU an                      --
--    ustimeartint_o    - 16bit integer Laufzeit arteriell                    --
--    ustimeartfract_o  - 10bit nachkomma Laufzeit arteriell                  --
--    ustimevenint_o    - 16bit integer Laufzeit venös                        --
--    ustimevenfract_o  - 10bit nachkomma Laufzeit venös                      --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 30.10.2002                                                         --
--      Revision : a - 02.12.2002                                             --
--------------------------------------------------------------------------------


-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

-- Moduledefinition
ENTITY index2time IS
    PORT(
--DEBUG
        dbgport             : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG

        clk                 : IN    STD_LOGIC;
        reset               : IN    STD_LOGIC;
        start_i             : IN    STD_LOGIC;
        timeindexart_i      : IN    UNSIGNED(19 DOWNTO 0);
        timeindexven_i      : IN    UNSIGNED(19 DOWNTO 0);
        timeconst_i         : IN    UNSIGNED(5 DOWNTO 0);

        alurege1_i          : IN    UNSIGNED(23 DOWNTO 0);
        alurege2_i          : IN    UNSIGNED(23 DOWNTO 0);
        alustatus_i         : IN    UNSIGNED(3 DOWNTO 0);
        aluready_i          : IN    STD_LOGIC;

        alurega_o           : OUT   UNSIGNED(23 DOWNTO 0);
```

```
70          aluregb_o               : OUT   UNSIGNED(23 DOWNTO 0);
            alustart_o              : OUT   STD_LOGIC;
72          aluop_o                 : OUT   UNSIGNED(7 DOWNTO 0);
            alusel_o                : OUT   STD_LOGIC;
74

76          ustimeartint_o          : OUT   UNSIGNED(15 DOWNTO 0);
            ustimeartfract_o        : OUT   UNSIGNED(9 DOWNTO 0);
78
            ustimevenint_o          : OUT   UNSIGNED(15 DOWNTO 0);
80          ustimevenfract_o        : OUT   UNSIGNED(9 DOWNTO 0);

82          errorcode_o             : OUT   UNSIGNED(3 DOWNTO 0);
            ready_o                 : OUT   STD_LOGIC
84          );
    END index2time;
86

88  ARCHITECTURE a OF index2time IS

90
        -- Deklaration der States
92      TYPE STATE_TYPE IS ( state_init,
                             state_reg,
94                           state_shiftloop,
                             state_shiftleft, state_shiftright,
96                           state_checkloop,
                             state_complete );
98
        SIGNAL state: STATE_TYPE;                       -- Statevariable
100

102 BEGIN

104

106     -- Statemachine
        PROCESS (clk,reset)
108
        -- Zähler für Ganzzahl-Bits
110     VARIABLE bitcount : UNSIGNED(3 DOWNTO 0);
        -- temporärer Bitzähler
112     VARIABLE bcount    : UNSIGNED(3 DOWNTO 0);

114
        -- temporäre Summenregister für Zwischenergebnisse
116     VARIABLE artsum : UNSIGNED(25 DOWNTO 0);
        VARIABLE vensum : UNSIGNED(25 DOWNTO 0);
118
        -- Register für Parameter
120     VARIABLE timeindexart : UNSIGNED(19 DOWNTO 0);
        VARIABLE timeindexven : UNSIGNED(19 DOWNTO 0);
122     VARIABLE timeconst : UNSIGNED(5 DOWNTO 0);

124     -- Register für Fehlercode
        VARIABLE errorcode : UNSIGNED(3 DOWNTO 0);
126
        BEGIN
128         -- asynchroner reset
            IF reset = '1' THEN
130
                -- setze Ready-Flag zurück
132             ready_o <= '0';

134             -- setzte Flag für ALU Benutzung zurück
                alusel_o <= '0';
136
                -- initialer State
138             state <= state_init;

140         -- positiver Takt
```

```
        ELSIF clk'EVENT AND clk = '1' THEN
142
            CASE state IS
144
                -- Init State
146             WHEN state_init =>
                    -- warte auf Start-Signal
148                 IF start_i = '1' THEN
                        -- setzte Ready-Flag zurück
150                     ready_o <= '0';
                        -- setze Flag für ALU-Benutzung
152                     alusel_o <= '1';
                        -- setze Zähler auf Anzahl der Ganzzahl-Bits (5)
154
                        -- lösche temporäre Register
156                     artsum := CONV_UNSIGNED(0, 26);
                        vensum := CONV_UNSIGNED(0, 26);
158
                        -- springe zur Berechnung
160                     state <= state_reg;
                    ELSE
162                     -- verweile in diesem State
                        state <= state_init;
164                 END IF;

166             WHEN state_reg =>
                    -- speichere die Eingangswerte
168                 timeindexart := timeindexart_i;
                    timeindexven := timeindexven_i;
170                 timeconst := timeconst_i;
                    bitcount := CONV_UNSIGNED(5, 4);
172                 state <= state_shiftloop;

174             WHEN state_shiftloop =>
                    -- wenn der Bitzähler 0 erreicht hat ->
176                 --  springe zu shiftright
                    -- ansonsten dekrementiere den temporären Bitzähler
178                 --      und springe in shiftleft
                    IF bitcount = 0 THEN
180                     state <= state_shiftright;
                    ELSE
182                     bcount := bitcount - 1;
                        state <= state_shiftleft;
184                 END IF;

186             WHEN state_shiftright =>
                    -- wenn das Nachkommabit von timeconst_i gesetzt ist,
188                 -- schiebe um ein Bit nach rechts (Divison durch 2)
                    IF timeconst(CONV_INTEGER(bitcount)) = '1' THEN
190                     artsum := artsum +
                            SHR(CONV_UNSIGNED(timeindexart,26),
192                             CONV_UNSIGNED(1,8) );
                        vensum := vensum +
194                         SHR(CONV_UNSIGNED(timeindexven,26),
                                CONV_UNSIGNED(1,8) );
196                     state <= state_checkloop;
                    ELSE
198                     state <= state_checkloop;
                    END IF;
200
                WHEN state_shiftleft =>
202                 -- wenn das Bit an der Stelle des aktuellen Bitzähler
                    --  gesetzt ist
204                 -- schiebe um die Zählstelle nach links
                    --  (Multiplikation mit Bitposition)
206                 IF timeconst(CONV_INTEGER(bitcount)) = '1' THEN
                        artsum := artsum +
208                         SHL(CONV_UNSIGNED(timeindexart,26), bcount);
                        vensum := vensum +
210                         SHL(CONV_UNSIGNED(timeindexven,26), bcount);
                        state <= state_checkloop;
```

```
212                     ELSE
                            state <= state_checkloop;
214                     END IF;

216                 WHEN state_checkloop =>
                        -- wenn Bitzähler auf 0 steht ist die Berechnung beendet
218                     -- ansonsten dekrementiere Bitzähler und berechne weiter
                        IF bitcount = 0 THEN
220                         state <= state_complete;
                        ELSE
222                         bitcount := bitcount - 1;
                            state <= state_shiftloop;
224                     END IF;

226
                    -- Complete-State
228                 WHEN state_complete =>

230                     -- setze Ready-Flag
                        ready_o <= '1';
232                     -- setzte Flag für ALU-benutzung zurück
                        alusel_o <= '0';
234                     -- wieder in den Init-State zurückkehren
                        state <= state_init;
236
                    WHEN others =>
238                     state <= state_init;

240             END CASE;
            END IF;
242
            -- Ende Statemachine
244
            -- gebe Ergebnisse an Ausgang
246         ustimeartint_o <= artsum(25 DOWNTO 10);
            ustimeartfract_o <= artsum(9 DOWNTO 0);
248         ustimevenint_o <= vensum(25 DOWNTO 10);
            ustimevenfract_o <= vensum(9 DOWNTO 0);
250         -- gebe Fehlercode an Ausgang
            errorcode_o <= errorcode;
252
    END PROCESS;
254
END a;
```

B.3.8. Steuermodul Berechnung

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   SIGSEQUENCECONTROLLER                                            --
--           steuert die Berechnung der Ultraschalllaufzeit und               --
--           koodiniert die Teilmodule                                        --
--                                                                            --
--  Inputs:                                                                   --
--    clk              - system Takt                                          --
--    reset            - system reset                                         --
--    start_i          - startet den Berechnungsprozess                       --
--    newsample_i      - zeigt an, dass ein neuer Datensatz bereitsteht       --
--    newsampleartven_i - zeigt an, welcher Kanal aktiv ist                   --
--    readysig_i       - Ready-Flag von sigproc-Modul                         --
--    errsig_i         - 4bit Fehlercode von sigproc-Modul                    --
--    readymean_i      - Ready-Flag von meanshots-Modul, dass neuer           --
--                       Mittelwert errechnet wurde                           --
--    readymeansample_i - Ready-Flag von meanshots-Modul, dass ein            --
--                        Wert übernommen wurde                               --
--    errmean_i        - 4bit Fehlercode von meanshots-Modul                  --
--    readytime_i      - Ready-Flag von index2time-Modul                      --
--    errtime_i        - 4bit-Fehlercode von index2time-Modul                 --
--  Outputs:                                                                  --
--    ready_o          - zeigt an, dass eine neue Ultraschalllaufzeit         --
--                       ermittelt wurde                                      --
--    errorcode_o      - 4bit Fehlercode                                      --
--    newsampleclr_o   - Clear-Flag für newsample Signal                      --
--    alusel_o         - 2bit ALU Auswahladresse                              --
--    startsig_o       - Startsignal für sigproc-Modul                        --
--    startmean_o      - Startsignal für meanshots-Modul                      --
--    startmeansample_o - Startsignal für Sample meanshots-Modul              --
--    starttime_o      - Startsignal für index2time-Modul                     --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 11.10.2002                                                         --
--       Revision: a - 24.10.2002                                             --
--       Revision: b - 03.12.2002                                             --
--                                                                            --
--------------------------------------------------------------------------------



-- Load standard Libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduldefinition
ENTITY sigsequencecontroller IS
    PORT(
--DEBUG
        dbgport             : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG

        clk                 : IN    STD_LOGIC;
        reset               : IN    STD_LOGIC;
        start_i             : IN    STD_LOGIC;
        newsample_i         : IN    STD_LOGIC;
        newsampleartven_i   : IN    STD_LOGIC;
        readysig_i          : IN    STD_LOGIC;
        errsig_i            : IN    UNSIGNED(3 DOWNTO 0);
        readymean_i         : IN    STD_LOGIC;
        readymeansample_i   : IN    STD_LOGIC;
        errmean_i           : IN    UNSIGNED(3 DOWNTO 0);
```

```
70          readytime_i         : IN    STD_LOGIC;
            errtime_i           : IN    UNSIGNED(3 DOWNTO 0);
72
            ready_o             : OUT   STD_LOGIC;
74          errorcode_o         : OUT   UNSIGNED(3 DOWNTO 0);
            newsampleclr_o      : OUT   STD_LOGIC;
76          alusel_o            : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
            startsig_o          : OUT   STD_LOGIC;
78          startmean_o         : OUT   STD_LOGIC;
            startmeansample_o   : OUT   STD_LOGIC;
80          starttime_o         : OUT   STD_LOGIC

82
        );
84  END sigsequencecontroller;

86
    ARCHITECTURE a OF sigsequencecontroller IS
88

90      -- Deklaration der States
        TYPE STATE_TYPE IS ( state_init,
92                           state_startmeanshotsproc, state_waitforsignal,
                             state_poststartmeanshotsproc,
94                           state_startsigproc, state_poststartsigproc,
                             state_waitsigproc,
96                           state_startmeanshotssampleproc,
                             state_poststartmeanshotssampleproc,
98                           state_waitmeanshotssampleproc,
                             state_startindex2timeproc,
100                          state_poststartindex2timeproc,
                             state_waitindex2timeproc,
102                          state_complete
                            );
104
        SIGNAL state: STATE_TYPE;                       -- Statevariable
106

108 BEGIN

110     -- Statemachine
        PROCESS (clk, reset)
112
        VARIABLE errorcode  : UNSIGNED(3 DOWNTO 0);
114         VARIABLE alusel         : STD_LOGIC_VECTOR(1 DOWNTO 0);
        BEGIN
116         -- asynchroner Reset
            IF reset = '1' THEN
118
                -- setzte alle Start_Flags zurück
120             startmean_o <= '0';
                startmeansample_o <= '0';
122             startsig_o <= '0';
                starttime_o <= '0';
124             -- setze Ready-Flag zurück
                ready_o <= '0';
126                         alusel := "00";

128              -- initialer State
                state <= state_init;
130
            -- positive Taktflanke
132         ELSIF clk'EVENT AND clk = '1' THEN

134             CASE state IS

136                 -- Init State
                    WHEN state_init =>
138                     -- setzte alle Start_Flags zurück
                        startmean_o <= '0';
140                     startmeansample_o <= '0';
```

```
                    startsig_o <= '0';
142                 starttime_o <= '0';
                    -- schalte ALU auf sigproc-Modul
144                 alusel := "00";

146                 -- wenn Start-Signal
                    IF start_i = '1' THEN
148                     -- Fehlercode zurücksetzen
                        errorcode := "0000";
150                     -- setze Ready-Flag zurück
                        ready_o <= '0';
152                     state <= state_startmeanshotsproc;
                    ELSE
154                     -- verweile in diesem State
                        state <= state_init;
156                 END IF;

158
                WHEN state_startmeanshotsproc =>
160                 -- starte den meanshots-Prozess
                    startmean_o <= '1';
162                 state <= state_poststartmeanshotsproc;

164             WHEN state_poststartmeanshotsproc =>
                    startmean_o <= '0';
166                 state <= state_waitforsignal;

168             WHEN state_waitforsignal =>
                    -- schalte ALU auf sigproc-Modul
170                 -- warte auf neues Sample,
                    -- wenn neues  Sample vorhanden -> starte sigproc
172              --   alusel := "00";
                    IF newsample_i = '1' THEN
174                     state <= state_startsigproc;
                    ELSE
176                     state <= state_waitforsignal;
                    END IF;
178
                WHEN state_startsigproc =>
180                                 newsampleclr_o <= '1';
                    -- starte den sigproc-Prozess
182                 startsig_o <= '1';
                    state <= state_poststartsigproc;
184
                WHEN state_poststartsigproc =>
186                 startsig_o <= '0';
                                    newsampleclr_o <= '0';
188                 state <= state_waitsigproc;

190             WHEN state_waitsigproc =>
                    -- wenn der sigproc-Prozess beenden wurde, mache
192                 -- weiter mit meanshotssample
                    IF readysig_i = '1' THEN
194                     state <= state_startmeanshotssampleproc;
                    ELSE
196                     state <= state_waitsigproc;
                    END IF;
198
                WHEN state_startmeanshotssampleproc =>
200                 -- schalte ALU auf meanshots-Modul
                   -- alusel := "01";
202                 -- Fehlercode dem Fehlercode aus sigproc-Modul zuweisen
                    errorcode := errsig_i;
204                 -- starte den meanshotssample-Prozess
                    startmeansample_o <= '1';
206                 state <= state_poststartmeanshotssampleproc;

208             WHEN state_poststartmeanshotssampleproc =>
                    startmeansample_o <= '0';
210                 state <= state_waitmeanshotssampleproc;
```

```
212                 WHEN state_waitmeanshotssampleproc =>
                        -- wenn der meanshotssample-Prozess beendet wurde
214                     -- prüfen, ob alle Samples zur Mittelwertbildung
                        -- vorhanden
216                     -- wenn ja, mit index->Zeitumwandlung fortfahren
                        -- anonsten weiteres Signal aufnehmen
218                     IF readymeansample_i = '1' THEN
                            IF readymean_i = '1' THEN
220                             state <= state_startindex2timeproc;
                            ELSE
222                             state <= state_waitforsignal;
                                                            --alusel := "00";
224                         END IF;
                        ELSE
226                         state <= state_waitmeanshotssampleproc;
                        END IF;
228
                    WHEN state_startindex2timeproc =>
230                     -- schalte ALU auf index2time-Modul
                        -- alusel := "10";
232                     -- starte index2time-Prozess
                        starttime_o <= '1';
234                     state <= state_poststartindex2timeproc;

236                 WHEN state_poststartindex2timeproc =>
                        starttime_o <= '0';
238                     state <= state_waitindex2timeproc;

240                 WHEN state_waitindex2timeproc =>
                        -- wenn der idnex2time-Prozess beendet wurde,
242                     -- in den Complete-State wechseln
                        IF readytime_i = '1' THEN
244                         state <= state_complete;
                        ELSE
246                         state <= state_waitindex2timeproc;
                        END IF;
248
                    WHEN state_complete =>
250                     -- schalte ALU auf sigproc-Modul
                        -- alusel := "00";
252                     -- setzte Ready-Flag
                        ready_o <= '1';
254                     -- gehe wieder in Init-State
                        state <= state_init;
256
                    WHEN others =>
258                     state <= state_init;
                END CASE;
260         END IF;

262         errorcode_o <= errorcode;
            -- End Statemachine
264                 alusel_o <= alusel;

266     END PROCESS;

268 END a;
```

B.3.9. Parameterdefinitionen

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   ALGOCFG                                                          --
--           enthält Konstanten, die das Verhalten der Signalverarbeitung --
--           beeinflussen                                                     --
--                                                                            --
--  Inputs:                                                                   --
--  Outputs:                                                                  --
--      meanstart_const_o      - 10bit sample mean calculation start index --
--      meanstop_const_o       - 10bit sample mean calculation stop index  --
--      findsigstart_const_o   - 10bit findsig start index                 --
--      findsigstop_const_o    - 10bit findsig stop index                  --
--      minarea_const_o        - 10bit minimal area for signal detection   --
--      minperiod_const_o      - 8bit minimal signal period for detection  --
--      maxperiod_const_o      - 8bit maximal signal period for detection  --
--      minampl_const_o        - 8bit minimal amplitude for detection      --
--      interpolsteps_const_o - 4bit number of interpolation steps         --
--      meancount_const_o      - 8bit number of shots to average           --
--      timeconst_const_o      - 5bit int/1bit fract of sample time const  --
--  Autor: A. Kühn                                                            --
--  Datum: 22.11.2002                                                         --
--------------------------------------------------------------------------------

-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Modulschnittstelle
ENTITY algocfg IS
    PORT(
            meanstart_const_o      : OUT UNSIGNED(9 DOWNTO 0);
            meanstop_const_o       : OUT UNSIGNED(9 DOWNTO 0);
            findsigstart_const_o   : OUT UNSIGNED(9 DOWNTO 0);
            findsigstop_const_o    : OUT UNSIGNED(9 DOWNTO 0);
            minarea_const_o        : OUT UNSIGNED(9 DOWNTO 0);
            minperiod_const_o      : OUT UNSIGNED(7 DOWNTO 0);
            maxperiod_const_o      : OUT UNSIGNED(7 DOWNTO 0);
            minampl_const_o        : OUT UNSIGNED(7 DOWNTO 0);
            interpolsteps_const_o  : OUT UNSIGNED(3 DOWNTO 0);
            meancount_const_o      : OUT UNSIGNED(7 DOWNTO 0);
            timeconst_const_o      : OUT UNSIGNED(5 DOWNTO 0)

        );
END algocfg;

ARCHITECTURE a OF algocfg IS
BEGIN

constants:
    -- Startindex der Mittelwertberechnung
    meanstart_const_o      <= CONV_UNSIGNED( 100, 10 );
    -- Stopindex der Mittelwertberechnung
    meanstop_const_o       <= CONV_UNSIGNED( 400, 10);
    -- Startindex der Signalsuche
    findsigstart_const_o   <= CONV_UNSIGNED( 400, 10);
    -- Stopindex der Signalsuche
    findsigstop_const_o    <= CONV_UNSIGNED(800, 10);
    -- Signalsuche: Mindestfläche negativer Schwingungsbauch
    minarea_const_o        <= CONV_UNSIGNED( 180, 10);
    -- Signalsuche: Mindestperiode Signalschwingung (Samples/s)
    minperiod_const_o      <= CONV_UNSIGNED(  20, 8);
    -- Signalsuche: Maximale Periode Signalschwingung (Samples/s)
```

```
70      maxperiod_const_o       <= CONV_UNSIGNED(  70, 8);
        -- Signalsuche: Mindestamplitude im Signalbereich
72      minampl_const_o         <= CONV_UNSIGNED( 100, 8);
        -- Anzahl Interpolationsschritte im Nullstellenbereich
74      interpolsteps_const_o <= CONV_UNSIGNED( 2, 4);
        -- Anzahl Laufzeitmessungen zur Mittelung der Gesamtlaufzeit 2^n
76      meancount_const_o       <= CONV_UNSIGNED( 6, 8);
        -- Zeitkonstante Umrechnung Sampleindex -> Zeit 5Vorkomma,1Nachkomma
78      timeconst_const_o       <= CONV_UNSIGNED( 25, 6);
    END a;
```

B.3.10. Schaltplan Ultraschallimpulserzeugung

B.3.11. Schaltplan Ultraschallimpuls-Synchronisation

B.3.12. Teilmodul SPI-Datenübertragung

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   SPI                                                              --
--           implementiert ein SPI-Slave Interface mit 8x16bit Datenpuffer    --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - System Takt                                           --
--    reset           - System Reset                                          --
--    wren            - Schreibfreigabe SPI Sendepuffer                       --
--    wraddress       - 2bit (8 Adressen) Adresseingang Sendepuffer           --
--    wrdata          - 16bit Dateneingang Sendepuffer                        --
--    rden            - Lesefreigabe SPI Empfangspuffer                       --
--    rdaddress       - 2bit (8 Adressen) Adresseingang Empfangspuffer        --
--    spiclk          - SPI-Bus clock                                         --
--    spirx           - SPI Receive Signal                                    --
--    spifrx          - SPI Receive Enable Signal                             --
--    spiftx          - SPI Transmit Enable Signal                            --
--  Outputs:                                                                  --
--    rddata          - 16bit Datenausgang vom Empfangspuffer                 --
--    spitx           - SPI Transmit Signal                                   --
--    dataavail       - Flag, neue Daten im Empfangspuffer                    --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 12.11.2002                                                         --
--------------------------------------------------------------------------------



-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduldefinition
ENTITY spi IS
    PORT(
--DEBUG
        dbgport         : INOUT STD_LOGIC_VECTOR(19 DOWNTO 0);
--ENDDEBUG

        clk             : IN    STD_LOGIC;
        reset           : IN    STD_LOGIC;

        wren            : IN    STD_LOGIC;
        wraddress       : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
        wrdata          : IN    STD_LOGIC_VECTOR(15 DOWNTO 0);

        rden            : IN    STD_LOGIC;
        rdaddress       : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);

        spiclk          : IN    STD_LOGIC;
        spirx           : IN    STD_LOGIC;
        spifrx          : IN    STD_LOGIC;
        spiftx          : IN    STD_LOGIC;

        rddata          : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0);
        spitx           : OUT   STD_LOGIC;
        dataavail       : OUT   STD_LOGIC
        );
END spi;

```

```
70
72  ARCHITECTURE a OF spi IS

74  -- Komponenten
    -- DualPort-RAM für Sende/Empfangspuffer
76  COMPONENT dprbuf IS
        PORT(
78          clk             : IN    STD_LOGIC;
            reset           : IN    STD_LOGIC;
80
            wren            : IN    STD_LOGIC;
82          wraddress       : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
            wrdata          : IN    STD_LOGIC_VECTOR(15 DOWNTO 0);
84
            rden            : IN    STD_LOGIC;
86          rdaddress       : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);

88          spiwren         : IN    STD_LOGIC;
            spiwraddress    : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
90          spiwrdata       : IN    STD_LOGIC_VECTOR(15 DOWNTO 0);

92          spirden         : IN    STD_LOGIC;
            spirdaddress    : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
94
            r0wren          : OUT   STD_LOGIC;
96          r1wren          : OUT   STD_LOGIC;
            r0rden          : OUT   STD_LOGIC;
98          r1rden          : OUT   STD_LOGIC;

100         rddata          : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0);
            spirddata       : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0)
102         );
    END COMPONENT;
104
    -- Schieberegister seriell in / parallel out
106 component lpm_shiftreg_1to16
        PORT
108     (
            clock       : IN STD_LOGIC ;
110         enable      : IN STD_LOGIC ;
            shiftin     : IN STD_LOGIC ;
112         aclr        : IN STD_LOGIC ;
            q           : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
114     );
    end component;
116
    -- Schieberegister parallel in / seriell out
118 component shiftreg_16to1
        port
120     (
            loadclk     :  IN  STD_LOGIC;
122         clock       :  IN  STD_LOGIC;
            enable      :  IN  STD_LOGIC;
124         aclr        :  IN  STD_LOGIC;
            data        :  IN  STD_LOGIC_VECTOR(15 downto 0);
126         shiftout    :  OUT  STD_LOGIC
        );
128 end component;

130 -- 16bit Register
    component lpm_register16
132     PORT
        (
134         clock       : IN STD_LOGIC ;
            aclr        : IN STD_LOGIC ;
136         data        : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            q           : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
138     );
    end component;
140
```

```
--D-FlipFLop
COMPONENT DFF
    PORT (d    : IN STD_LOGIC;
        clk    : IN STD_LOGIC := '0';
        clrn   : IN STD_LOGIC ;
        prn    : IN STD_LOGIC ;
        q      : OUT STD_LOGIC );
END COMPONENT;

-- 2bit Zähler
component lpm_counter8
    PORT
    (
        clock       : IN STD_LOGIC ;
        clk_en      : IN STD_LOGIC ;
        aclr        : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
end component;



SIGNAL  spisync          : STD_LOGIC;
SIGNAL  reset_spirxshift : STD_LOGIC;
SIGNAL  spirxword        : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL  spirxwordreg     : STD_LOGIC_VECTOR(15 DOWNTO 0);

SIGNAL memrden           : STD_LOGIC;
SIGNAL memrdaddress      : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL memrddata         : STD_LOGIC_VECTOR(15 DOWNTO 0);

SIGNAL memwren           : STD_LOGIC;
SIGNAL memwraddress      : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL memwrdata         : STD_LOGIC_VECTOR(15 DOWNTO 0);

SIGNAL r0 : STD_LOGIC;
SIGNAL r1 : STD_LOGIC;
SIGNAL r2 : STD_LOGIC;
SIGNAL r3 : STD_LOGIC;
SIGNAL rddatamem         : STD_LOGIC_VECTOR(15 DOWNTO 0);


BEGIN

-- Das Empfangsschieberegister übernimmt an jeder positiven SPI-Taktflanke
-- den Wert an der Empfangsleitung,
-- sofern das Enable-Signal(spifrx) aktiv ist

-- SPI-Empfangsschieberegister verdrahten
spirxshift : lpm_shiftreg_1to16
    PORT MAP(
        clock   => spiclk,
        enable  => NOT spifrx,
        shiftin => spirx,
        aclr    => reset_spirxshift,
        q       => spirxword
    );

-- Nach Übertragung eines 16bit Datenblocks (spifrx) wird H,
-- wird der 16bit Wert aus dem Empfangsschieberegister in
-- das Empfangsregister übertragen

-- SPI-Empfangsregister verdrahten
spirxreg : lpm_register16
    PORT MAP(
        clock   => spifrx,
        aclr    => reset_spirxshift,
        data    => spirxword,
        q       => spirxwordreg
```

```
212       );

214
    -- Das SPi-Sendeschieberegister wird nach der Übertragung eines Datenblocks
216 -- mit einem neuen Datenwert aus dem Sendepuffer geladen,
    -- diese Daten werden mit jeder positiven SPI-Taktflanke seriell an die SPI-
218 -- Sendeleitung gelegt

220 -- SPI-Sendeschieberegister verdrahten
    spitxshift : shiftreg_16to1
222     PORT MAP(
            loadclk  => NOT spifrx,
224         clock    => spiclk,
            enable   => NOT spifrx,
226         aclr     => reset_spirxshift,
            data     => memrddata,
228         shiftout => spitx
        );
230

232
    syncspi :
234     -- spisync wird H, wenn das festgelegte
        -- Synchronisationswort "A5A5" empfange wurde
236     -- damit werden alle SPI-Datenblockzähler zurückgesetzt
        spisync <= '1' WHEN spirxwordreg = CONV_STD_LOGIC_VECTOR(16#A5A5#,16)
238                   ELSE '0';
        reset_spirxshift <= spisync;
240
        rddata <= memrddata;
242     memrden <= '1';

244

246 -- Sende/empfangspufferspeicher verdrahten
    spibuffers : dprbuf
248     PORT MAP(
            clk => clk,
250         reset => reset,

252         wren => wren,
            wraddress => wraddress,
254         wrdata => wrdata,

256         rden => rden,
            rdaddress => rdaddress,
258
            spiwren => memwren,
260         spiwraddress => memwraddress,
            spiwrdata => memwrdata,
262
            spirden => memrden,
264         spirdaddress => memrdaddress,

266         r0wren => r0,
            r1wren => r1,
268         r0rden => r2,
            r1rden => r3,
270
            rddata => rddatamem,
272         spirddata => memrddata
        );
274
    -- Der SPI Wortzähler wird nach Übertragung eines Datenblocks inkrementiert
276 -- und zeigt auf den nächsten zu übertragenden Datenblock
    -- im Sende/Empfangspuffer
278
    -- SPI Wortzähler verdrahten
280 spiwordcounter : lpm_counter8
        PORT MAP (
282         clock    => spifrx,
```

```
            clk_en    => '1',
284         aclr      => reset_spirxshift,
            q         => memrdaddress
286     );

288
END a;
```

B.3.13. Teilmodul SPI-Datenpuffer

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   SPICNTRL                                                         --
--           kopiert die ermittelten Ultraschalllaufzeiten                    --
--           in den SPI Puffer                                                --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - System Takt                                           --
--    reset           - System Rreset                                         --
--    start_i         - startet den Kopiervorgang                             --
--    errorcode_i     - 4bit Fehlercode des Systems                           --
--    usmeantimeartint_i- 16bit Ganzzahl Laufzeit arteriell                   --
--    usmeantimeartfract_i - 10bit Nachkomma Laufzeit arteriell               --
--    usmeantimevenint_i- 16bit Ganzzahl Laufzeit venös                       --
--    usmeantimevenfract_i - 10bit Nachkomma Laufzeit venös                   --
--    spiclk          - SPI-Bus clock                                         --
--    spirx           - SPI Receive Signal                                    --
--    spifrx          - SPI Receive Enable Signal                             --
--    spiftx          - SPI Transmit Enable Signal                            --
--  Outputs:                                                                  --
--    ready_o         - zeigt das Ende des Kopiervorgangs an                  --
--    errorcode_o     - 4bit Fehlercode des SPI-Subsystems                    --
--    spitx           - SPI Transmit Signal                                   --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 15.11.2002                                                         --
--------------------------------------------------------------------------------


-- include standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduldefinition
ENTITY spicntrl IS
    PORT(

--DEBUG
        dbgport                 : INOUT STD_LOGIC_VECTOR(19 DOWNTO 0);
--ENDDEBUG

        clk                     : IN    STD_LOGIC;
        reset                   : IN    STD_LOGIC;
        start_i                 : IN    STD_LOGIC;
        errorcode_i             : IN    UNSIGNED(3 DOWNTO 0);
        usmeantimeartint_i      : IN    UNSIGNED(15 DOWNTO 0);
        usmeantimeartfract_i    : IN    UNSIGNED(9 DOWNTO 0);
        usmeantimevenint_i      : IN    UNSIGNED(15 DOWNTO 0);
        usmeantimevenfract_i    : IN    UNSIGNED(9 DOWNTO 0);

        spiclk                  : IN    STD_LOGIC;
        spirx                   : IN    STD_LOGIC;
        spifrx                  : IN    STD_LOGIC;
        spiftx                  : IN    STD_LOGIC;

        ready_o                 : OUT   STD_LOGIC;
        errorcode_o             : OUT   UNSIGNED(3 DOWNTO 0);

        spitx                   : OUT   STD_LOGIC

        );
END spicntrl;
```

```
70
72  ARCHITECTURE a OF spicntrl IS
74
    -- Komponenten
76
    -- SPI link
78  COMPONENT spi IS
        PORT(
80  --DEBUG
            dbgport             : INOUT STD_LOGIC_VECTOR(19 DOWNTO 0);
82  --ENDDEBUG
84          clk                 : IN    STD_LOGIC;
            reset               : IN    STD_LOGIC;
86
            wren                : IN    STD_LOGIC;
88          wraddress           : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
            wrdata              : IN    STD_LOGIC_VECTOR(15 DOWNTO 0);
90
            rden                : IN    STD_LOGIC;
92          rdaddress           : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
94          spiclk              : IN    STD_LOGIC;
            spirx               : IN    STD_LOGIC;
96          spifrx              : IN    STD_LOGIC;
            spiftx              : IN    STD_LOGIC;
98
            rddata              : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0);
100         spitx               : OUT   STD_LOGIC;
            dataavail           : OUT   STD_LOGIC
102         );
    END COMPONENT;
104
106
108     -- Deklaration der States
        TYPE STATE_TYPE IS ( state_init,
110                          state_setwrdata, state_checkwriteend,
                             state_complete );
112
        SIGNAL state: STATE_TYPE;                       -- Statevariable
114
116     SIGNAL spiwrdata            : UNSIGNED(15 DOWNTO 0);
        SIGNAL spiwren              : STD_LOGIC;
118     SIGNAL spiwraddress         : UNSIGNED(2 DOWNTO 0);
120     SIGNAL spirddata            : STD_LOGIC_VECTOR(15 DOWNTO 0);
        SIGNAL spirden              : STD_LOGIC;
122     SIGNAL spirdaddress         : UNSIGNED(2 DOWNTO 0);
        SIGNAL spidataavail         : STD_LOGIC;
124
    --DEBUG
126     SIGNAL spidbg               : STD_LOGIC_VECTOR(19 DOWNTO 0);
    --ENDDEBUG
128
130 BEGIN
132
            -- Statemachine
134     PROCESS (clk,reset)
136     -- Zählregister für SPI-Pufferadresse
        VARIABLE spiaddrcount : UNSIGNED(2 DOWNTO 0);
138     -- Register für Fehlercode
        VARIABLE errorcode    : UNSIGNED(3 DOWNTO 0);
140
```

```
        VARIABLE errcalc            : UNSIGNED(3 DOWNTO 0);
142 BEGIN
        -- asynchrone reset
144     IF reset = '1' THEN

146         -- setzte Ready_Flag zurück
            ready_o <= '0';
148
            -- setzte Fehlercoderegister zurück
150         errorcode := "0000";
            -- setze Zähler für SPI-Pufferadresse zurück
152         spiaddrcount := CONV_UNSIGNED(0,3);
            -- deaktiviere SPI-Pufferspeicherzugriff
154         spiwren <= '0';

156         -- Initialer State
            state <= state_init;
158
        -- positive Taktflanke
160     ELSIF clk'EVENT AND clk = '1' THEN

162         CASE state IS

164             -- Init State
                WHEN state_init =>
166                 -- warte auf Start-Signal
                    IF start_i = '1' THEN
168
                        -- setze Ready-Flag zurück
170                     ready_o <= '0';
                        -- setzte Fehlercoderegister zurück
172                     errorcode := "0000";

174                                         -- übernehme Fehlercode aus Berechnungsmodul
                                            errcalc := errorcode_i;
176                     -- setze Zähler für SPI-Pufferadresse zurück
                        spiaddrcount := CONV_UNSIGNED(0,3);
178                     -- deaktiviere SPI-Pufferspeicherzugriff
                        spiwren <= '0';
180
                        -- Folgestate setzen
182                     state <= state_setwrdata;
                    ELSE
184                     -- in aktuellem State verweilen
                        state <= state_init;
186                 END IF;

188             -- setwrdata State
                WHEN state_setwrdata =>
190                 -- setzt SPI-Pufferspeicherdatenbus in Abhängigkeit von
                    -- dem akutell zu beschreibenden Puffer
192
                    CASE CONV_INTEGER(spiaddrcount) IS
194                     WHEN 0 =>
                            -- schreibe Nullen an Pufferstelle 0
196                         spiwrdata <= CONV_UNSIGNED(16#00AA#,16);
                        WHEN 1 =>
198                         -- schreibe Nullen an Pufferstelle 1
                            spiwrdata <= CONV_UNSIGNED(16#0055#,16);
200                     WHEN 2 =>
                            -- schreibe Ganzzahl aus
202                         -- arterieller Ultraschalllaufzeit
                            -- an Pufferstelle 2
204
                                spiwrdata <= usmeantimeartint_i;
206

208                     WHEN 3 =>
                            -- schreibe Nachkomma aus
210                         -- arterieller Ultraschalllaufzeit
                            -- an Pufferstelle 3
```

```
212
                                    spiwrdata(9 DOWNTO 0) <= usmeantimeartfract_i;
214                                 spiwrdata(15 DOWNTO 10) <= "000000";
                                WHEN 4 =>
216                                 -- schreibe Nullen an Pufferstelle 4
                                    spiwrdata <= CONV_UNSIGNED(16#0055#,16);
218                             WHEN 5 =>
                                    -- schreibe Nullen an Pufferstelle 5
220                                 spiwrdata <= CONV_UNSIGNED(16#00AA#,16);
                                WHEN 6 =>
222                                 -- schreibe Ganzzahl aus
                                    -- venöser Ultraschalllaufzeit
224                                 -- an Pufferstelle 6

226                                 spiwrdata <= usmeantimevenint_i;

228                             WHEN 7 =>
                                    -- schreibe Nachkomma aus
230                                 -- venöser Ultraschalllaufzeit
                                    -- an Pufferstelle 7
232
                                    spiwrdata(9 DOWNTO 0) <= usmeantimevenfract_i;
234                                 spiwrdata(15 DOWNTO 10) <= "000000";
                                WHEN others =>
236                                 -- Abfangen von ungültigem
                                    -- Pufferspeicheradresszähler
238                                 null;
                            END CASE;
240
                            -- aktiviere das Schreiben in SPI-Pufferspeicher
242                         spiwren <= '1';

244                         -- Folgestate setzen
                            state <= state_checkwriteend;
246
                        -- Checkwriteend State
248                     WHEN state_checkwriteend =>
                            -- deaktiviere das Schreiben in SPI-Pufferspeicher
250                         spiwren <= '0';

252                         -- prüfen, ob das Ende des SPI-Pufferspeichers
                            -- erreicht wurde
254                         IF spiaddrcount = 7 THEN
                                -- wenn das Ende des Pufferspeichers erreicht wurde,
256                             -- den Adresszähler zurücksetzen und
                                -- in Complete-State wechseln
258                             spiaddrcount := CONV_UNSIGNED(0,3);
                                state <= state_complete;
260                         ELSE
                                -- wenn das Ende noch nicht erreicht wurde,
262                             -- den Adresszähler inkrementieren und im
                                -- setwrdata State fortsetzen
264                             spiaddrcount := spiaddrcount + 1;
                                state <= state_setwrdata;
266                         END IF;

268
                        -- Complete State
270                     WHEN state_complete =>

272                         -- setze Ready-Flag
                            ready_o <= '1';
274                         -- wieder in den Init State zurückkehren
                            state <= state_init;
276
                    END CASE;
278             END IF;
                -- End Statemachine
280
                -- SPI-Pufferspeicheradresszähler auf SPI-Adressausgang legen
282             spiwraddress <= spiaddrcount;
```

```
        -- Fehlercoderegister auf Fehlercode-Ausgang legen
284     errorcode_o  <= errorcode;
    END PROCESS;
286

288 -- SPI-Teilmodul einbinden
    spiinst : spi
290     PORT MAP(
    --DEBUG
292     dbgport    => spidbg,
    --ENDDEBUG
294
        clk        => clk,
296     reset      => reset,

298     wren       => spiwren,
        wraddress => CONV_STD_LOGIC_VECTOR(spiwraddress,3),
300     wrdata     => CONV_STD_LOGIC_VECTOR(spiwrdata,16),

302     rden       => spirden,
        rdaddress => CONV_STD_LOGIC_VECTOR(spirdaddress,3),
304
        spiclk     => spiclk,
306     spirx      => spirx,
        spifrx     => spifrx,
308     spiftx     => spiftx,

310     rddata     => spirddata,
        spitx      => spitx,
312     dataavail => spidataavail
        );
314

316 END a;
```

B.3.14. Schaltpläne Gesamtstruktur

B.3.15. Steuermodul Gesamtsystem

```
--------------------------------------------------------------------------------
--  Projekt: USDSP                                                            --
--           Ultraschalllaufzeitmessung                                       --
--                                                                            --
--  Modul:   maincontroller                                                   --
--           koordiniert die Teilmodule zur Berechnung und zur                --
--           SPI Übertragung                                                  --
--                                                                            --
--  Inputs:                                                                   --
--    clk             - system Takt                                           --
--    reset           - system reset                                          --
--    readycalc_i     - Ready-Flag von Berechnungsmodul                       --
--    errcalc         - 4bit Fehlercode von Berechnungsmodul                  --
--    readyspi        - Ready-Flag von SPI-Modul                              --
--    errspi_i        - 4bit-Fehlercode von SPI-Modul                         --
--  Outputs:                                                                  --
--    errorcode_o     - 4bit Fehlercode                                       --
--    startcalc_o     - Startsignal für BerechnungsModul                      --
--    startspi_o      - Startsignal für SPI-Modul                             --
--                                                                            --
--  Autor: A. Kühn                                                            --
--  Datum: 10.12.2002                                                         --
--                                                                            --
--------------------------------------------------------------------------------


-- Load standard Libraries
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

LIBRARY lpm;
USE lpm.lpm_components.ALL;


-- Moduldefinition
ENTITY maincontroller IS
    PORT(
--DEBUG
        dbgport           : INOUT STD_LOGIC_VECTOR(16 DOWNTO 0);
--ENDDEBUG

        clk               : IN    STD_LOGIC;
        reset             : IN    STD_LOGIC;
        readycalc_i       : IN    STD_LOGIC;
        errcalc_i         : IN    UNSIGNED(3 DOWNTO 0);
        readyspi_i        : IN    STD_LOGIC;
        errspi_i          : IN    UNSIGNED(3 DOWNTO 0);

        errorcode_o       : OUT   UNSIGNED(3 DOWNTO 0);
        startcalc_o       : OUT   STD_LOGIC;
        startspi_o        : OUT   STD_LOGIC


        );
END maincontroller;


ARCHITECTURE a OF maincontroller IS


    -- Deklaration der States
    TYPE STATE_TYPE IS ( state_init,
                         state_startcalc, state_poststartcalc,
                         state_waitcalc,
                         state_startspi, state_poststartspi,
                         state_waitspi
                       );
```

```
70
    SIGNAL state: STATE_TYPE;                          -- Statevariable
72
74  BEGIN

76      -- Statemachine
        PROCESS (clk, reset)
78
        VARIABLE errorcode   : UNSIGNED(3 DOWNTO 0);
80
        BEGIN
82          -- asynchroner Reset
            IF reset = '1' THEN
84
                -- setzte alle Start_Flags zurück
86              startcalc_o <= '0';
                startspi_o <= '0';
88
                 -- initialer State
90              state <= state_init;

92          -- positive Taktflanke
            ELSIF clk'EVENT AND clk = '1' THEN
94
                CASE state IS
96
                    -- Init State
98                  WHEN state_init =>
                        -- setzte alle Start_Flags zurück
100                     startcalc_o <= '0';
                        startspi_o <= '0';
102
                        state <= state_startcalc;
104

106                 WHEN state_startcalc =>
                        -- starte Laufzeitberechnungsprozess
108                     startcalc_o <= '1';
                        state <= state_poststartcalc;
110
                    WHEN state_poststartcalc =>
112                     startcalc_o <= '0';
                        state <= state_waitcalc;
114
                    WHEN state_waitcalc =>
116                     -- wenn der Berechnungsprozess beendet wurde,
                        --  starte SPI-Übertragung
118                     IF readycalc_i = '1' THEN
                            state <= state_startspi;
120                     ELSE
                            state <= state_waitcalc;
122                     END IF;

124                 WHEN state_startspi =>
                        -- starte SPI-Modul
126                     startspi_o <= '1';
                        state <= state_poststartspi;
128
                    WHEN state_poststartspi =>
130                     startspi_o <= '0';
                        state <= state_waitspi;
132
                    WHEN state_waitspi =>
134                     -- wenn die SPI-Übertragung beendet wurde,
                        --  zurück zum Anfang springen
136                     IF readyspi_i = '1' THEN
                            state <= state_init;
138                     ELSE
                            state <= state_waitspi;
140                     END IF;
```

```
142
                    WHEN others =>
144                     state <= state_init;
                END CASE;
146         END IF;

148         errorcode_o <= errorcode;
            -- End Statemachine
150
    END PROCESS;
152
END a;
```

C. Schaltpläne der Hardware

D. Layout der Platine

E. beiliegende CD-ROM

Folgender Inhalt befindet sich auf der beiliegenden CD-ROM:

/datasheets	Datenblätter und Application Notes
/dipltext	diese Diplomarbeit im PDF-Format
/hardware	Schaltpläne der Hardware im INTEGRA-Format
/src/sim	Quellen der OCTAVE-Simulation
/src/usdsp	Quellen der FPGA-Software