HAW HAMBURG

Bachelorarbeit

Manuel-André Böttger

Konzept und Implementierung eines Custom IP Cores für ein System-on-Chip auf einem FPGA zur Motorenansteuerung mittels PWM-Signalen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

Manuel-André Böttger

Konzept und Implementierung eines Custom IP Cores für ein System-on-Chip auf einem FPGA zur Motorenansteuerung mittels PWM-Signalen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Lutz Leutelt
Zweitgutachter: Prof. Dr. Marc Hensel

Eingereicht am: 23. Februar 2022

Manuel-André Böttger

Thema der Arbeit

Konzept und Implementierung eines Custom IP Cores für ein System-on-Chip auf einem FPGA zur Motorenansteuerung mittels PWM-Signalen

Stichworte

FPGA, System-on-Chip, MicroBlaze, AXI4-Lite, PWM , 2-FF-Synchronisierer

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Entwicklung eines IP Cores, welcher die Ansteuerung von Motoren mittels PWM-Signalen ermöglicht. Der entwickelte IP Core ermöglicht die autonome und manuelle Steuerung eines RC-Fahrzeugs und die Umstellung der Steuerungsart.

Manuel-André Böttger

Title of Thesis

Concept and implemantation of a custom IP Core for a system-on-a-chip on a FPGA for motor control by PWM signals

Keywords

FPGA, System-on-Chip, MicroBlaze, AXI4-Lite, PWM, 2-FF synchronizer

Abstract

This thesis includes the development of an IP Core, which enables the control of motors using PWM signals. The developed IP core enables autonomous and manual control of an RC vehicle and the conversion of the control mode.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das autonome Fahren gewinnt in der heutigen Zeit zunehmend an Bedeutung und entwickelt sich zu einer der Schlüsseltechnologien des 21. Jahrhunderts. Diese Schlüsseltechnologie hat das Potential maßgeblich das urbane Stadtbild zu verändern. Das autonome Fahren birgt somit große gesellschaftliche, wie wirtschaftliche Potentiale. So können durch diese Technologie neue Mobilitäts- und Verkehrskonzepte entwickelt werden, durch die das Verkehrsaufkommen erheblich verringert werden kann. Genauso können Verkehrsunfälle maßgeblich reduziert werden, da der Faktor Mensch und seine nicht vorhersehbaren möglichen Fehler raus gerechnet werden können. Ein weiterer Punkt ist die Weiterentwicklung von künstlicher Intelligenz für die Fahrzeugsteuerung. Jedoch gibt es auch viele offene Fragen bezüglich des autonomen Fahrens. Insbesondere sind ethisch-rechtliche Fragen zu klären. So ist zu klären, wie sich das Fahrzeug zu verhalten hat, wenn ein Unfall unvermeidbar ist. Auch ist zu klären, wer haftet, wenn es zu einem Unfall kommt, der durch einen technischen defekt des Fahrzeugs verursacht wird.
Dennoch liegen die vielen Vorteile des autonomen Fahrens auf der Hand und so sind schon heute viele Fahrassistenzsysteme in Fahrzeugen eingebaut. Hierzu wird modernste Sensortechnik und Software genutzt. Unter anderem wird hierbei auf Radar,- Laser,- Ultraschall- und Kameratechnik zurückgegriffen. Die daraus resultierenden großen Datenmengen müssen möglichst in Echtzeit und fehlerfrei verarbeitet werden. Um eine schnelle und sichere Datenverarbeitung zu gewährleisten, werden eingebettete System (englisch Embedded Systems) eingesetzt. Dabei kommen häufig Mikroprozessoren, FPGAs oder eine Kombination aus beidem zum Einsatz. Insbesondere in sicherheitsrelevanten Systemen, in denen eine schnelle, fehlerfreie Datenübertragung von Nöten ist, bieten sich FPGAs an, da diese eine wesentlich höhere Taktfrequenz als Mikrocontroller haben. Häufig werden auch beide Systeme kombiniert, wobei der Mikrocontroller als Steuereinheit dient und der FPGA die Sensorsignale verarbeitet. Im Rahmen dieser Arbeit wird ein solches Hardware-Software-Codesign weiterentwickelt.

1.1 Motivation und Zielsetzung

Kolja Gries hat in seiner Abschlussarbeit [9] ein System-on-Chip auf einem FPGA zur Multi-Sensor-Signalverarbeitung entwickelt. Dieses von ihm entwickelte System steuert und verarbeitet die Signale eines Ultraschallsensor-Arrays, welches zur Erkennung von Hindernissen in autonomen Fahrzeugen genutzt werden soll. Das System ist so konzipiert, dass es in RC-Fahrzeugen eingebaut werden kann und so im Rahmen von Bachelor-, Master- und Forschungsarbeiten genutzt und weiterentwickelt werden kann.
Ziel dieser Bachelorarbeit ist der Entwurf eines Custom IP-Cores zur Motorenansteuerung mittels PWM-Signalen. Dieser IP-Core soll die manuelle, wie autonome Steuerung eines RC-Fahrzeugs ermöglichen. Das System soll so ausgelegt sein, das die Umstellung der Steuerungsart ermöglicht wird. Hierfür wird zunächst eine Analyse des bestehenden Systems durchgeführt und daraus die Anforderungen an den zu entwerfenden Custom IP-Core abgeleitet. Auf Grundlage der Anforderungsanalyse erfolgt dann die Konzeption und das Design der Erweiterung. Nach erfolgreicher Umsetzung des Konzepts und des Designs, erfolgt eine Verifikation des Systems und eine Bewertung der Ergebnisse.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der Pulsweitenmodulation erläutert. Insbesondere wird hier auf die Ansteuerung des Lenkservos und Fahrtreglers mittels PWM-Signalen eingegangen. Außerdem wird der Soft-Core-Prozessor MicroBlaze vorgestellt, welcher zur Umsetzung der System-on-Chip-Architektur genutzt wird. Anschließend wird die Peripherie-Schnittstelle AXI4-Lite beschrieben, welche als Kommunikationsschnittstelle zwischen dem MicroBlaze und den Hardware-Komponenten genutzt wird.

2.1 Pulsweitenmodulation

Die Pulsweitenmodulation, kurz PWM, ist eine Modulationsart, bei der eine elektrische Spannung zwischen zwei Werten wechselt. Dabei wird das Verhältnis zwischen der Einschaltzeit und Periodendauer eines Rechtecksignals bei fester Grundfrequenz variiert. Das Verhältnis zwischen der Einschaltzeit t_{on} und der Periodendauer $T = t_{on} + t_{off}$ wird als Tastverhältnis p bezeichnet (engl. Duty Cycle, kurz DC). In Abbildung 2.1 ist der Verlauf eines PWM-Signals abgebildet. Der zeitliche Mittelwert der Spannung $U(t)$ innerhalb eines Intervalls $[0, T_s]$ lässt sich wie folgt beschreiben:

$$V_0 = V_{in} \cdot DC = V_{in} \cdot \frac{t_{on}}{T_s} = V_{in} \cdot \frac{t_{on}}{t_{on} + t_{off}} \tag{2.1}$$

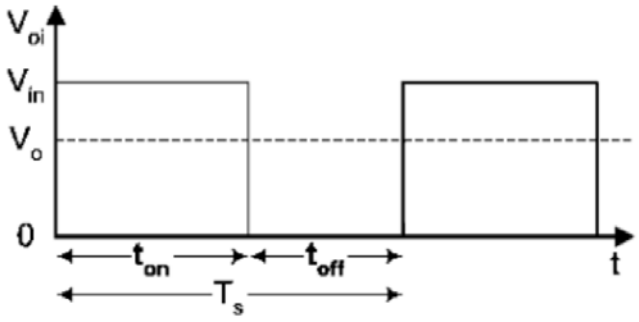
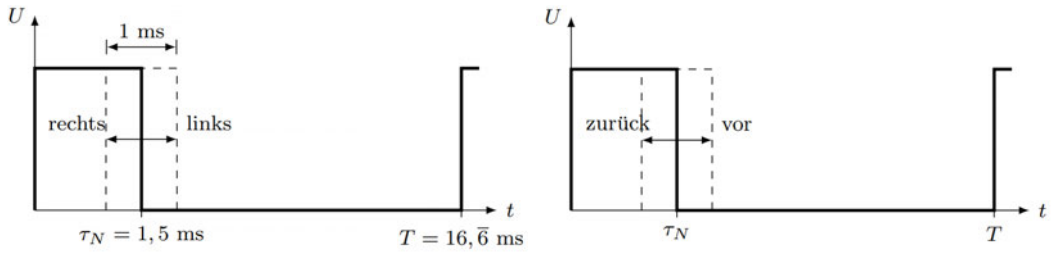


Abbildung 2.1: Allgemeiner Verlauf eines PWM-Signals [13]

2.2 Dart Truggy

Der Dart Truggy verfügt über einen Lenk-Servo und einen Fahrtregler. Über einen Empfänger werden auf zwei Kanälen PWM-Signale empfangen. Kanal 1 empfängt das PWM-Signal für den Lenk-Servo und Kanal 2 das Signal für den Fahrtregler. Der Fahrtregler leitet dann das Signal weiter an den Motor. Auf diese Weise wird der Dart Truggy durch einen Controller gesteuert. Abbildung 2.2 zeigt die PWM-Signale des Lenk-Servos und des Fahrtreglers.



(a) Servo (Kanal 1) (b) Fahrtregler (Kanal 2)

Abbildung 2.2: PWM-Signale Dart-Truggy [10]

2.2.1 Lenkung

Der Lenk-Servo sorgt für eine Lenk-Bewegung, in dem dieser ein PWM-Signal je nach Pulsbreite umsetzt. Der Servo benötigt eine Spannungsversorgung sowie ein PWM-Signal

gemäß Abbildung 2.2(a), welches den Auslenkwinkel des Servoarms festlegt. Die Eingangspannung des periodischen Rechtecksignals beträgt $V_{in} = 3,3\ V$. Die gemessene Periodendauer T beträgt etwa $T = 16,67\ ms$, was einer Frequenz von $f = 1/T \approx 60\ Hz$ entspricht. Die Pulsbreite τ liegt in einem Bereich von etwa 1 ms bis 2 ms mit der neutralen Stellung $\tau_N = 1,5\ ms$ [10]. Längere Pulse $\tau_L > \tau_N$ bedeuten eine Linkslenkung und kürzere Pulse $\tau_R < \tau_N$ eine Rechtslenkung, wie in Abbildung 2.2(a) zu erkennen ist.

2.2.2 Motor

Zur Steureung des Motors ist ein Fahrtregler notwendig, da der Motor keine PWM-Signale direkt umsetzen kann. Der Fahrtregler gibt eine gepulste Spannung beziehungsweise einen gepulsten Strom an den Motor ab. Der Fahrtregler verfügt über die gleichen Anschlüsse wie ein analoger Servo und wird auf die gleiche Weise angesteuert. Die Steuerung funktioniert daher genauso wie die Steuerung des Lenk-Servos über PWM-Signale. Die Frequenz beträgt ebenfalls $f = 60\ Hz$ und die Pulsdauer τ liegt ebenfalls zwischen 1 ms und 2 ms. Eine Pulsdauer von $\tau_N = 1,5\ ms$ entspricht der neutralen Stellung (Leerlauf). Längere Pulse $\tau_V > \tau_N$ erzeugen eine Vorwärtsbewegung und kürzere Pulse $\tau_R < \tau_N$ eine Rückwärtsbewegung [10]. Dieses Verhalten ist in Abbildung 2.2(b) dargestellt.

2.3 Soft-Core-Mikroprozessor MicroBlaze

Der MicroBlaze ist ein von FPGAs der Firma Xilinx verwendbarer Mikrocontroller. Dieser Microcontroller existiert nicht als physische Hardware, sondern ist nur als in Hardwarebeschreibungssprachen wie VHDL und Verilog verfasster Softcore verfügbar.

2.3.1 Architektur des MicroBlaze

Der Soft-Core-Mikroprozessor MicroBlaze ist ein RISC-Prozessor, welcher für Implementationen in Xilinx FPGAs optimiert ist [21, S.7]. Er kann als 32- oder 64-Bit RISC-Prozessor mit Harvard-Architektur implementiert werden. Der MicroBlaze nutzt die drei folgenden Schnittstellen:

- Local Memory Bus (LMB)
- Advanced eXtensible Interface (AXI4)
- AXI Coherency Extension (ACE)

Der LMB wird zur Anbindung des FPGA internen Block-RAMs (BRAM) genutzt. Die AXI4-Schnittstelle ermöglicht die Anbindung von Peripherie-Komponenten [21, S.61], um so die Erweiterung der System-Funktionalität zu ermöglichen. Über eine ACE- oder AXI4-Schnittstelle erfolgt der Cache-Zugriff [21, S.61]. Der Zugriff auf den Cache-Speicher ist bei der Nutzung von FPGA externen Speicher-Komponenten, wie Flash oder SRAM von großem Vorteil, da die Zugriffszeit auf diese Speicher-Elemente verringert wird. Der Cache-Speicher des MicroBlaze kann hierbei von 64 B bis 64 kB konfiguriert werden [21, S.94]. Der MicroBlaze bietet, je nach System-Anforderung, viele weitere optionale Konfigurationsmöglichkeiten und kann so durch weitere Komponenten erweitert werden [21, S8f]. Abbildung 2.3 zeigt das Block Diagramm des MicroBlaze. Die grauen Felder sind optionale Features.

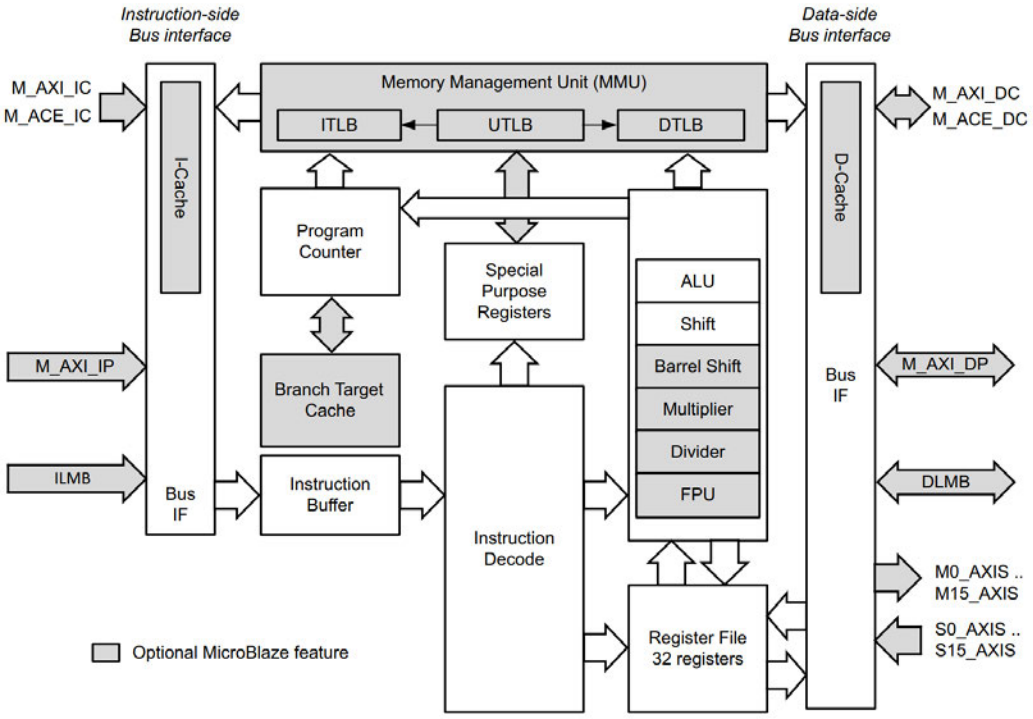


Abbildung 2.3: MicroBlaze Core Block Diagramm [21, S.7]

2.3.2 Local Memory Bus

Der LMB ist ein synchroner Bus, der hauptsächlich für den Zugriff auf den on-Chip BRAM verwendet wird. Er verwendet ein Minimum an Steuersignalen und ein einfaches Protokoll, um sicherzustellen, dass der BRAM in einem einzigen Taktzyklus aufgerufen wird. Alle Signale des LMB sind *active-High* [17, S.7]. Abbildung 2.4 zeigt das Blockschaltbild der eingesetzten IP-Funktionsblöcke. Die maximale nutzbare BRAM-Kapazität beträgt 256 kB [20, S.9] und ist abhängig von der verwendeten BUS-Datenbreite. Falls diese Kapazität nicht ausreicht, muss auf externe Speicher, wie dem SRAM, zurück gegriffen werden.

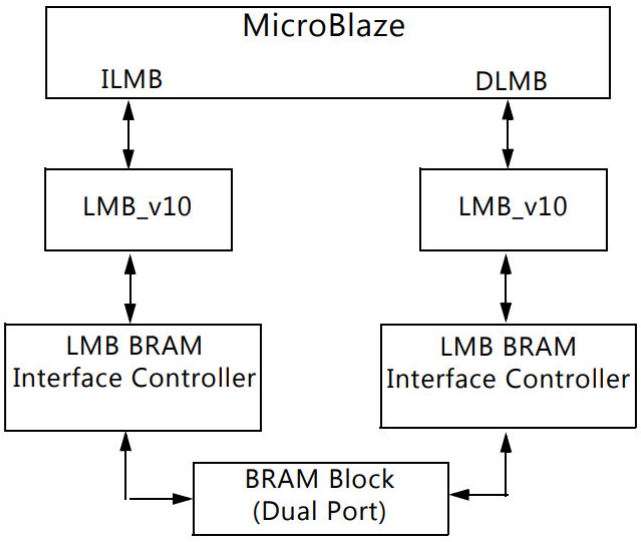


Abbildung 2.4: Anbindung des BRAM an den MicroBlaze über den LMB [17, S.9]

2.3.3 AXI4-Interfacestandard

Der AXI4-Interfacestandard ist Bestandteil des Advanced-Microcontroller-BUS-Architecture, kurz AMBA, Verbindungsstandards. Dieser wurde von der Firma ARM entwickelt und wird vorrangig zur Kommunikation zwischen digitalen Subsystemen und Prozessoren genutzt. Der Standard definiert die folgenden drei AXI4-Interfaces:

- **AXI4:** Hochgeschwindigkeitsinterface, bidirektional

- **AXI4-Lite:** Teilmenge von AXI4, reduzierte Übertragungsrate, bidirektional
- **AXI4-Stream:** Punkt-zu-Punkt Stream-Verbindung, unidirektional

In den folgenden Abschnitten wird die AXI4-Lite-Schnittstelle und die Bus-Topologie beschrieben.

2.3.4 AXI4-Interconnect

Die AXI4- bzw. AXI4-Lite-Schnittstelle ermöglicht eine Multi-Master-Multi-Slave-Kommunikation. Wenn mehr als zwei Bus-Teilnehmer in ein System eingebunden werden sollen, wird ein zusätzliches AXI-Interconnect-Modul benötigt. Dieses Modul ermöglicht die Konfiguration von bis zu 16 Master-Interfaces (MI) und 16 Slave-Interfaces (SI) [18, S.11]. Hierbei wird die Methode der speicherbezogenen Adressierung (Memory-Mapped I/O) genutzt. Abbildung 2.5 zeigt die physikalische Bus-Topologie.

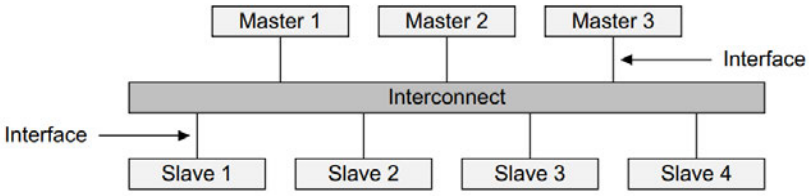


Abbildung 2.5: Bus-Topologie der AXI4-Schnittstelle [2, S.A1-28]

Abbildung 2.6 zeigt das Blockdiagramm des IP-Cores. Dieser verfügt, entsprechend der Anzahl der angeschlossenen Bus-Teilnehmer, über Master- und Slave-Schnittstellen. Intern besitzt dieses Modul Koppelelemente, welche in der Abbildung als *Coupler* bezeichnet werden. Diese bestehen aus den folgenden Hardware-Komponenten und ermöglichen so die Kommunikation zwischen den Bus-Teilnehmern [18, S.5]:

- **AXI Crossbar:** Verbindet einen oder mehrere Master mit einem oder mehreren Slaves
- **AXI Data Width Converter:** Ermöglicht die Konvertierung der Bus Datenreihe
- **AXI Clock Converter:** Ermöglicht die Konvertierung des genutzten Bus-Taktes

- **AXI Protocol Converter:** Ermöglicht die Konvertierung des Schnittstellentypen
- **AXI Data FIFO:** Ermöglicht die Pufferung von Daten
- **AXI Register Slice:** Ermöglicht die Nutzung von Pipeline-Registern, typischerweise um kritische Timing-Pfade aufzulösen
- **AXI MMU:** Ermöglicht die Adresskodierung und Adressneuzuordnung

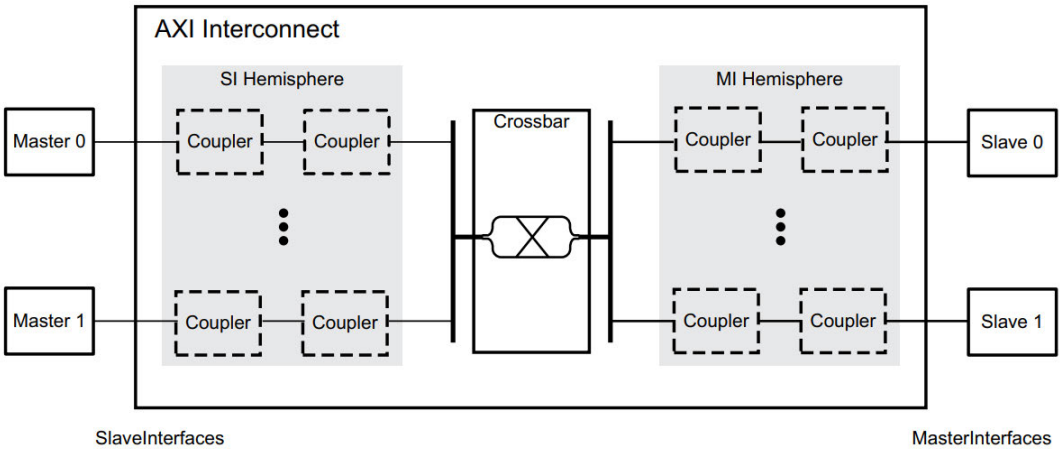


Abbildung 2.6: AXI-Interconnect Core Diagramm [18, S.11]

2.3.5 AXI4-Protokoll

Das AXI4 und AXI4-Lite-Interface nutzt insgesamt fünf voneinander unabhängige Kanäle [19, S.8]. Zwei Adress-Kanäle, zwei Daten-Kanäle und einen Schreib-Antwort-Kanal. In Abbildung 2.7 ist der Master-Lesevorgang dargestellt. Über den *Read adress channel* teilt der Master dem Slave mit, welche Daten übertragen werden sollen. Nachdem der Slave diese Information erhalten hat, überträgt der Slave über den *Read data channel* die Datenpakete an den Master. Abbildung 2.8 zeigt den Master-Schreibvorgang. Über den *Write adress channel* sendet der Master Informationen über die Art der zu übertragenden Datenpakete. Daraufhin werden die Datenpakete über den *Write data channel* an den Slave gesendet. Über den *Write response channel* gibt der Slave den Erhalt der Datenpakete an den Master zurück. Die Abbildungen zeigen, dass bei Verwendung einer AXI4-Schnittstelle das Übertragen mehrerer Datenpakete hintereinander möglich ist.

Dabei muss nur die Startadresse angegeben werden. Bei einer AXI4-Lite-Schnittstelle ist dies nicht möglich. Hier ist die Paketanzahl je Adressierung auf eins begrenzt.

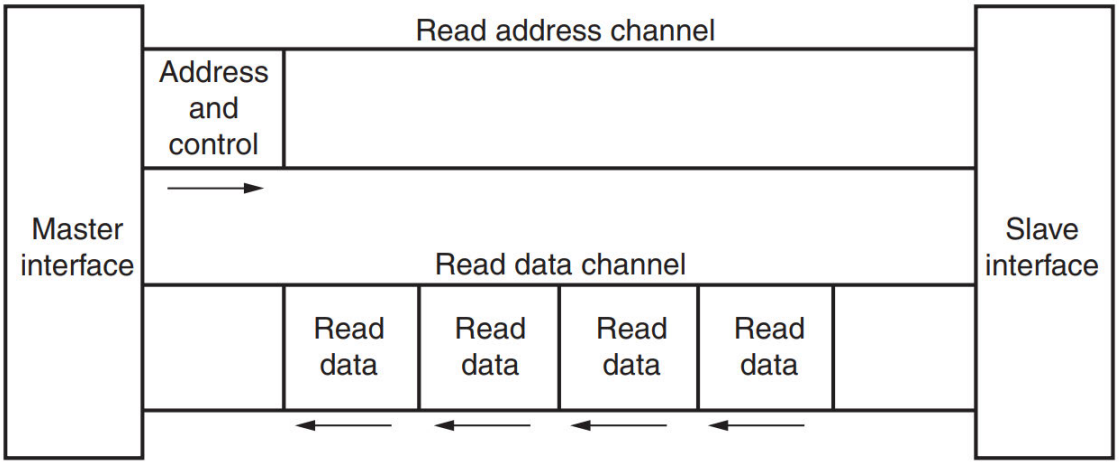


Abbildung 2.7: Kanal-Architektur des Lesevorgangs unter Verwendung einer AXI4-Schnittstelle [19, S.7]

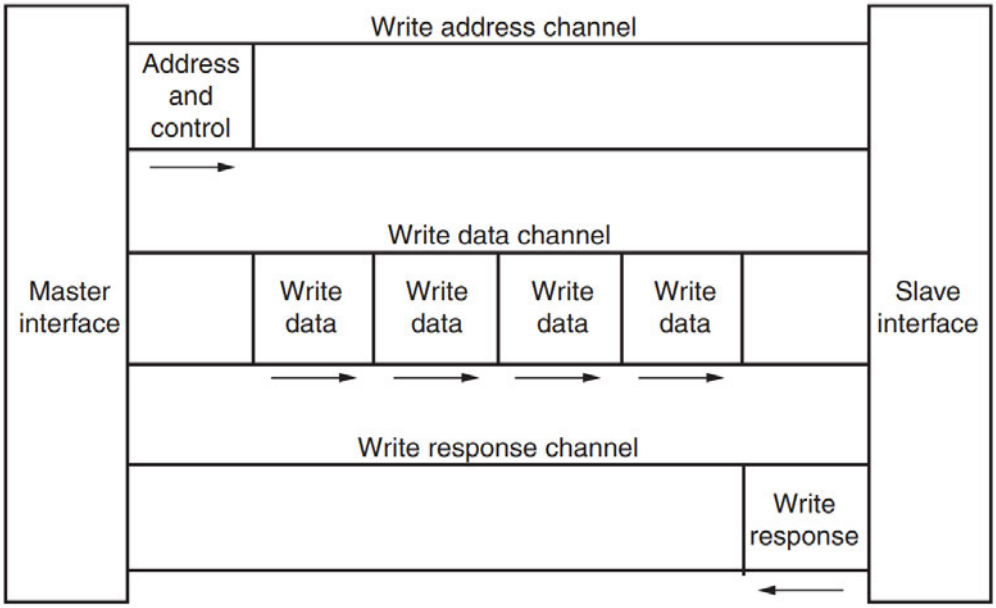


Abbildung 2.8: Kanal-Architektur des Schreibvorgangs unter Verwendung einer AXI4-Schnittstelle [19, S.8]

Alle fünf Übertragungskanäle nutzen das Zwei-Wege-Handshake-Verfahren zur Übertragung von Daten, Adressen und Steuerinformation [2, S.A3-41]. Je Kanal wird ein Valid- und Ready-Signal verwendet. Das Valid-Signal wird von der Datenquelle erzeugt und zeigt an, dass Adressinformationen, Steuerinformationen oder Daten verfügbar sind (T1). Die Zielquelle erzeugt, sobald sie bereit zum Datentransfer ist, das Ready-Signal (T2). Sind beide Signale aktiv, können die Daten übertragen werden. Die Datenquelle muss hierbei sicherstellen, dass die Informationen während der Übertragungszeit stabil bleiben. Mit der nächsten Taktflanke erfolgt dann die Übertragung (T3) [9, S.14]. Abbildung 2.9 zeigt den Ablauf des Zwei-Wege-Handshake-Verfahrens.

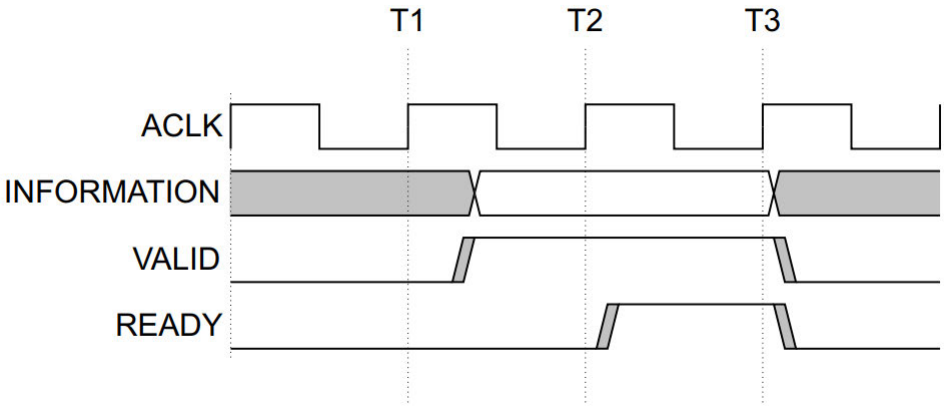


Abbildung 2.9: Prinzip des Zwei-Wege-Handshake-Verfahrens einer AXI4-Schnittstelle [2, S.A3-41]

3 Analyse des bestehenden Coprozessor-Systems

In diesem Abschnitt wird das bestehende Coprozessor-System analysiert. Dabei wird zunächst auf die ausgewählte Datenbus-Architektur und die Hardware-Plattform des Coprozessors eingegangen. Anschließend wird die Coprozessor-Architektur erläutert und die Entwicklung des Demonstrator-Systems analysiert. Insbesondere wird auch auf die ausgewählte Hardware zur Realisierung des Demonstrator-Systems eingegangen. Außerdem wird die Software- und Hardware-Architektur des System-on-Chip analysiert. Ziel dieser Analyse ist es, die Stärken und Schwächen des bestehenden Systems zu erkennen und daraus die Anforderungen für das PWM-Modul abzuleiten.

3.1 Aufbau des bestehenden Demsonstrator-Systems

Das System ist als Demonstrator-System auf einem Breadboard aufgebaut. In den folgenden Abschnitten wird die verwendete Hardware dieses Systems analysiert. Abbildung 3.1 zeigt das Blockschaltbild des Demonstrator-Systems.

3.1.1 Arduino Nano V3.0

Als zentrale Steuerungseinheit wird ein Arduino Nano V3.0 [1] verwendet. Der Arduino Nano ist eine auf dem ATmega328P [12] basierende Microcontroller-Plattform. Die Programmierung erfolgt in der Programmiersprache $C++$ und wird über eine USB- oder ICSP-Schnittstelle geladen. Der ATmega328P ist ein 8-Bit AVR RISC Mikrocontroller der Firma Atmel und wird mi einer maximalen Taktfrequenz von 16 MHz betrieben. Der Arduino Nano wird mit einer Betriebsspannung von 5 V betrieben und über die USB-Schnittstelle oder einen ausgeführten Pin erfolgen. Der auf dem Board eingebaute

Spannungsregler regelt die zulässige Eingangsspannung von $7 - 20\ V$ auf die benötigten $5\ V$ runter. Weitere technische Daten sind in Tabelle 3.1 gelistet.

Tabelle 3.1: Technische Kenndaten des Arduino Nano [1][12]

Eigenschaft	**Wert**
Mikrocontroller	ATmega328P
Architektur	8-Bit AVR RISC
Flash	32 kB
SRAM	2 kB
EEPROM	1 kB
Taktrate	16 MHz
Zähler	Zwei 8-Bit Counter
	Einen 16-Bit Counter
	Einen Real-Time Counter
ADC	8 Kanäle, 10 Bit
Schnittstellen	UART/USART
	TWI (I2C-kompatibel)
	SPI (Master/Slave)
Betriebsspannung	$5\ V$ / $7 - 20\ V$
Ausgangsspannung	$5\ V$ und $3,3\ V$
Digital I/O Spannung	$5\ V$
Stromverbrauch	19 mA
Ausgangsstrom I/O	max. 20 mA
Ausgangsstrom $3,3\ V$-Pin	max. 50 mA
Built-in LED Pin	13
Anzahl digitale I/O-Pins	14
Anzahl analoge I/O-Pins	8
PWM-Pins	6

3.1.2 FPGA-Modul Cmod A7-35T

Als FPGA für das Coprozessor-System wird der Cmod A7-35T [7] der Firma Digilent eingesetzt. Das Cmod A7-35T Board ist ein kleines, 48-Pin DIP Formfaktor-Board, das um einen Xilinx Artix-7 FPGA herum aufgebaut ist [6]. Das Board besitzt einen Quad-SPI Flash zum programmieren, genauso wie eine USB-JTAG-Programmierschaltung, ei-

ne USB-UART Bridge, eine Clock Source, einen Pmod Host Connector, einen externen SRAM, einen Quad-SPI Flash und Standard I/O Geräte [6]. Diese Komponenten machen das Board zu einer kompakten Plattform für digitale Logikschaltungen und eignet sich auch hervorragend für eingebettete MicroBlaze Soft-Core Coprozessor-Systeme unter Verwendung der Entwicklungssoftware Vivado der Firma Xilinx. Die genauen technische Daten des FPGA-Moduls sind in Tabelle 3.2 gelistet.

Tabelle 3.2: Technische Kenndaten des Cmod A7-35T [7]

Eigenschaft	**Wert**
FPGA	Artix-7 (XC7A35T-1CPG236C)
LUTs	20800
Flip-Flops	41600
Block Ram	255 kB
DSP Slices	90
Clock Management Tiles	5
ADC	1 $MSPS$
FPGA Betriebsspannung	$3,3\ V$
Oszillator (extern)	12 MHz
SRAM (extern)	512 kB
Quad-SPI-Flash (extern)	4 MB
Digital I/O	44
Analog I/O	2
Digital I/O Spannung I/O	max. $3,3\ V$
Betriebsspannung (extern)	$3,5 - 5,5\ V$
Abmessungen PCB	$1,778\ cm\ x\ 6,985\ cm$
Programmier-Schnittstelle	QSPI-Flash / JTAG
Sonstiges	USB-UART Bridge
	USB-JTAG Bridge
	Pmod-Schnittstelle
	2 LED
	1 RGB-LED
	2 Taster

3.1.3 Level-Shifter-Modul PCA9306 und BSS138

Das Level-Shifter-Modul PCA9306 wird verwendet, um sicher zu stellen ,dass, ungeachtet einer Pegelanpassung, die Übertragungsstrecke innerhalb der I2C-Bus-Spezifikation betrieben wird. Als Wandler dient der IC vom Typ PCA9306 [14] der Firma Texas Instruments. Dieser IC ist speziell für die Anwendung bei Inter-Chip Bus-Systemen, wie I2C oder SMBus ausgelegt. Die Pegel-Konvertierung erfolgt dual-bidirektional.
Für die Ansteuerung der Ultraschallsensoren werden drei Level-Shifter-Module vom Typ BSS138 verwendet. Diese sind wesentlich einfacher aufgebaut. Vier N-Kanal Feldeffekttransistoren ermöglichen die Pegel-Konvertierung von vier bidirektionalen Kanälen [8].

3.1.4 Spannungsregler-Modul LM2596S

Der Spannungsregler LM2596S [5] dient der sicheren Spannungsversorgung des Demonstrator-Systems. Es regelt die Eingangsspannung des Akkupacks des Dart Truggy von $7,2\ V$ auf $5\ V$ runter. Hierbei ist zu beachten, dass der maximale Ausgangsstrom des Spannungsreglers ohne Kühlkörper $2\ A$ beträgt. Die Schaltfrequenz der asynchronen Gleichrichtung beträgt $150\ kHz$, was für den Betrieb des FPGAs, der Level-Shifter und der Ultraschallsensoren ausreicht.

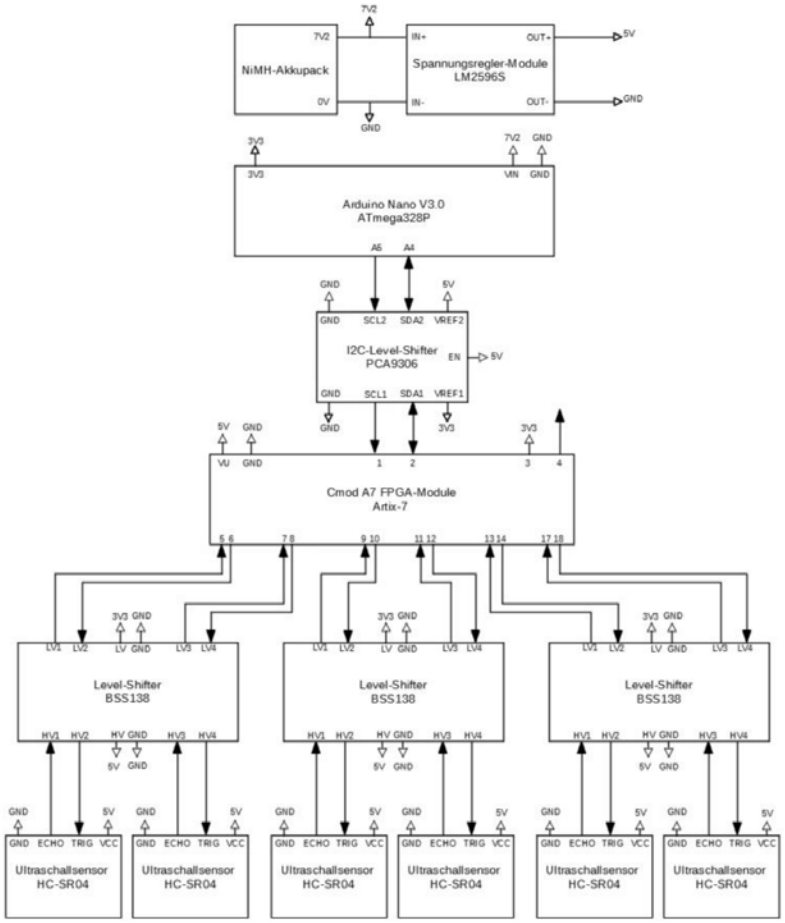


Abbildung 3.1: Blockschaltbild des Demonstrator-Systems zur Multi-Sensor-Signalverarbeitung [9, S.41]

3.2 Software-Design des System-on-Chip

In diesem Abschnitt wird das vorhandene Software-Design analysiert. Dabei wird auf das I2C-Paket-Layout und die Typisierung der AXI4-Slave-Register für IP-Funktionsblöcke eingegangen.

3.2.1 Software-Architektur des System-on-Chip

Die Kommunikation zwischen dem Arduino Nano und dem Soft-Core-Prozessor wurde mit ein er I2C-Schnittstelle umgesetzt. Abbildung 3.2 zeigt das Flussdiagramm, welches die Umsetzung der I2C-Kommunikation zeigt. Nach dem Programmstart wird zunächst die Initialisierung der UART-Schnittstelle, des Cache-Zugriffsspeichers und des GPIO-Moduls durchgeführt. Anschließend wird die I2C-Schnittstelle initialisiert. Das I2C-Hardware-Modul wird hierbei mit der festgelegten Slave-Adresse konfiguriert, sowie die Funktionalität eines Hardware-Interrupts durch dieses hergestellt. Nachfolgend findet eine Initialisierung des Interrupt-Systems statt. Nach der Freigabe von globalen Interrupts, wird eine *while(1)*-Schleife betreten, die *busy-waiting-Loop*. Erfolgt eine Lese- oder Schreibanfrage durch den I2C-Master, wird ein Interrupt-Request (IRQ) ausgelöst und eine Interrupt-Service-Routine (ISR) aufgerufen, welche den funktionalen Programmteil beinhaltet. Erfolgt kein IRQ durch die I2C-Hardware-Komponente, befindet sich das Programm kontinuierlich in einer *busy-waiting-Loop* [9, S.47].

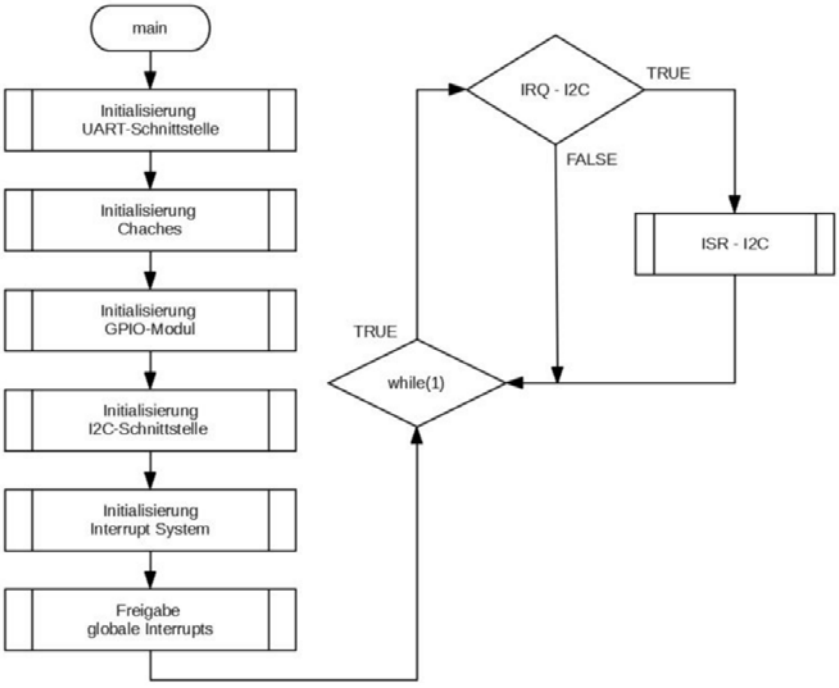


Abbildung 3.2: Flussdiagramm der Hauptfunktion zur Initialisierung des System-on-Chip [9, S.48]

Abbildung 3.3 zeigt das Flussdiagramm der Interrupt-Service-Routine. Zunächst wird abgefragt, ob es sich um eine Schreibanfrage handelt. Ist dies der Fall, wird zunächst eine Rx-LED auf dem FPGA eingeschaltet. Nachdem alle Datenpakete des Masters erfolgreich empfangen wurden, wird das I2C-Empfangsregister ausgelesen. Je nach Paketinhalt werden die Daten einem AXI4-Slave-Register eines bestimmten IP-Funktionsblock zugewiesen. Abschließend wird die Rx-LED wieder ausgeschaltet und die Interrupt-Service-Routine verlassen. Bei einer Leseanfrage, also wenn die Abfrage *EVENT == MASTER WRITE* negativ ausfällt, wird zunächst eine Tx-LED auf dem FPGA eingeschaltet. Analog zur Schreibanfrage wird anschließend ein AXI4-Slave-Register ausgelesen. Anschließend erfolgt der Aufbau des I2C-Paket-Layouts und die Daten werden an den Master übertragen. Danach wird die Tx-LED wieder ausgeschaltet und die Interrupt-Service-Routine verlassen.

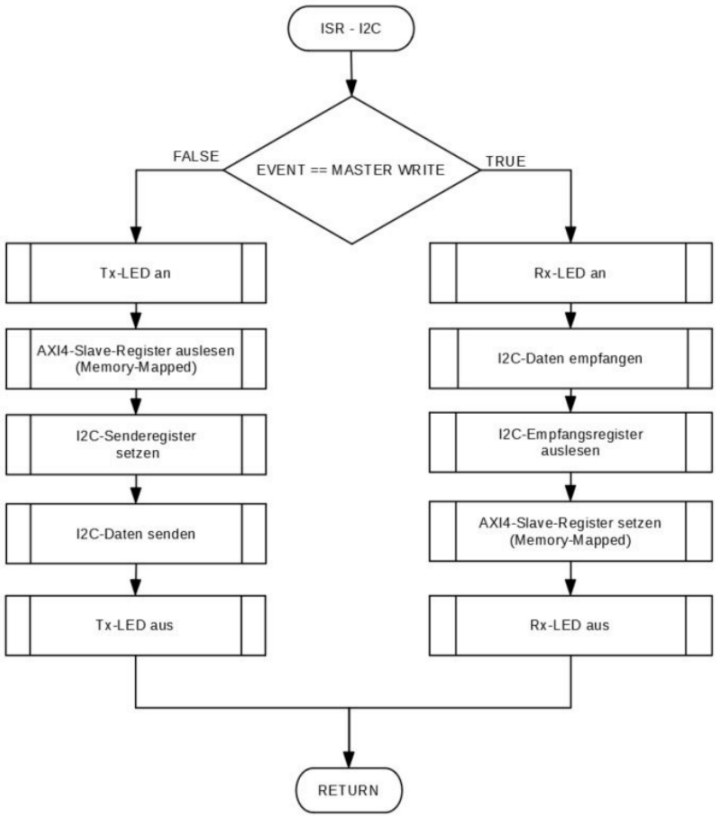


Abbildung 3.3: Flussdiagramm der Interrupt-Service-Routine zur I2C-Kommunikation und Ausführung von Speicherzugriffsroutinen des System-on-Chip [9, S.49]

3.2.2 Definition des I2C-Paket-Layouts

Abbildung 3.4 zeigt den Aufbau des I2C-Layouts. Zum Auslesen und Beschreiben der AXI-Slave-Register wird das I2C-TX/RX-Layout verwendet. Das legt die Modul-Adresse des Slaves und den Befehls-Typ der I2C-Übertragung fest. Das Byte 1 definiert das entsprechende Register des Moduls und besitzt ein Reserve-Bereich für mögliche Erweiterungen. Die nachfolgenden Bytes beinhalten die Daten. Das Byte 0 des I2C-Select-Layouts beinhaltet eine Slect-ID, durch die der Slave erkennt, dass es sich um eine Register-Auswahl durch den Master handelt. Byte 1 definiert die Modul-Adresse und den Befehlstypen und Byte 2 das Modul-Register. Anschließend kann ein Registerwert eines bestimmten IP-Moduls in den Schreib-Speicher geladen werden. Die nachfolgenden Bytes beinhalten keine Funktion. Tabelle 3.3 listet die genaue Definition des I2C-Paket-Layouts.

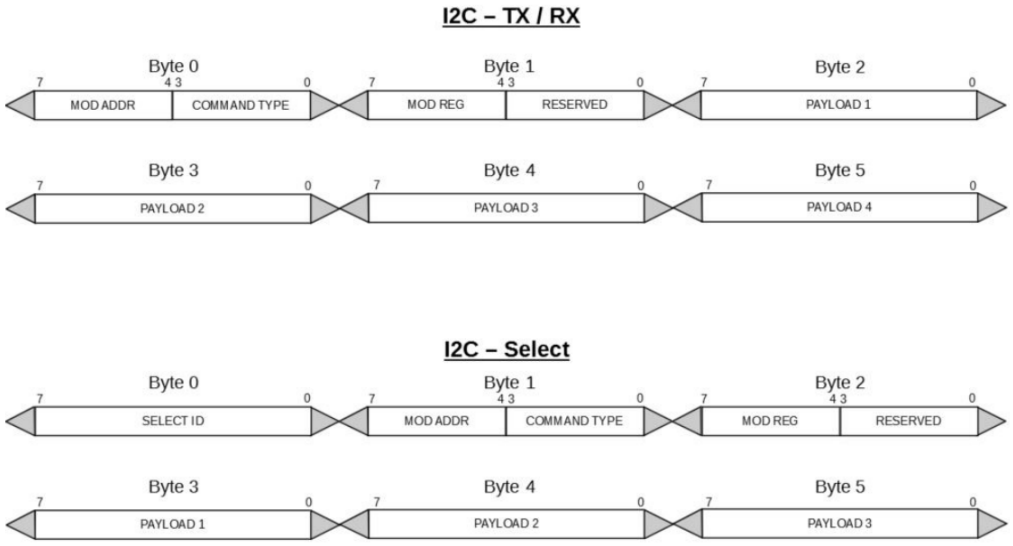


Abbildung 3.4: Aufbau des I2C-Layouts [9, S.50]

Tabelle 3.3: Definition des I2C-Paket-Layouts [9, S.51]

Bitfeld-Name	**Bedeutung**	**Inhalt**	**Bit-Anzahl**
MOD ADDR	IP-Modul Adresse	0xA - Sensor-Modul	4
		0xB - PWM-RX-Modul	
		0xC - PWM-TX-Modul	
		0xD - Gateway-Modul	
COMMAND TYPE	Befehlstyp (Debug)	0x0 - Select	4
		0x1 - Konfiguration	
		0x2 - Status	
		0x3 - Steuerung	
		0x4 - Daten	
MOD REG	Module-Register	0x0 - Register 0	4
		0x1 - Register 1	
		0x2 - Register 2	
		0x3 - Register 3	
		0x4 - Register 4	
		0x5 - Register 5	
		0x6 - Register 6	
		0x7 - Register 7	
Payload	Nutzdaten	0x000000 bei I2C-Select	24
		Nutzdaten bei I2C-TX/RX	32
Select ID	Select-ID	0x00	8
RESERVED	Reserviert	0x00	4

3.2.3 Typisierung der AXI4-Lite-Slave-Register

Zur leichteren Zuordnung des Register-Inhalts, wurde eine Typisierung der AXI4-Slave-Register vorgenommen. In Tabelle 3.4 sind die festgelegten Register-Bezeichnungen sowie die vorgesehenen Anwendungsfunktionen gelistet.

Tabelle 3.4: Typisierung des AXI-Slave-Registers [9, S.51]

Register-Bezeichnung	**Anwendungsfunktion**
CONFIG	Register konfigurieren das AXI4-Slave-Modul (z.B. vorgegebene Zählerwerte)
STATUS	Zustandsregister des AXI4-Slave-Moduls (z.B. Modul aktiv)
DATA	Datenregister des AXI4-Slave-Moduls (z.B. Sensorwerte, Steuerwerte)

3.3 Hardware-Design des System-on-Chip

In diesem Abschnitt wird das Hardware-Design des System-on-Chip analysiert. Insbesondere die AXI-Hardware-Module, welche nicht als IP-Funktionsblöcke in der Entwicklungsumgebung von Vivado zur Verfügung stehen, werden in ihrer Konfiguration und Funktion erläutert.

3.3.1 Hardware-Architektur des System-on-Chip

Die Hardware-Architektur setzt sich aus dem Soft-Prozessor-Core MicroBlaze und insgesamt 10 Peripherie-Modulen zusammen. Abbildung 3.5 zeigt den Aufbau der Hardware-Architektur. Die Kommunikation zwischen dem Mikroprozessor und den Peripherie-Module erfolgt über den On-Chip-Bus AXI4-Lite. Der AXI UART Lite LogicCore dient als Schnittstelle für eine serielle Datenübertragung zur UART/USB-Bridge, welche beim Software-Debugging genutzt wird. Das AXI I2C-Interface dient als Kommunikationsschnittstelle mit dem Mikrocontroller ATmega328P des Arduino Nano. Der AXI QSPI LogicCore ermöglicht die Anbindung des externen Flash-Speichers, welcher zum Laden der FPGA-Konfiguration und des ausführbaren Programms benötigt wird. Der AXI External Memory Controller (EMC) ermöglicht die Anbindung des externen SRAM-Speichers. Auf diesem Speicher können Teile des ausführbaren Programms ausgeführt werden und dient als Erweiterung des FPGA-internen Block-RAMs. In dem bestehenden System wird der SRAM nicht verwendet und dient als Möglichkeit in Folgeprojekten größere Software-Architekturen zu implementieren. Der AXI-GPIO LogicCore dient als Schnittstelle zur Ansteuerung von zwei LEDs, welche als optische Anzeige dienen, um

die Abfrageart des I2C-Masters anzuzeigen. Diese IP-Blöcke sind als Standard in der Vivado-Design-Suite vorhanden und mussten nicht erstellt werden. Daher wird in der Analyse nicht weiter auf diese eingegangen.
Das Sensor-Modul dient als Ansteuerung für das Sensor-Array. Außerdem wird in dem Modul die Multi-Sensor-Signalverarbeitung durchgeführt. Die Ansteuerung des Moduls und die Abfrage der Sensor-Werte erfolgt durch I2C-Master. Das Fail-Save-Data-Monitoring-Modul (FSDM) dient als Überwachung der Ausgangsspannung des Arduino Nano von $3,3\ V$. Fällt diese Spannung ab, wird ein Trigger-Signal als Flag gesetzt und auf einen GPIO-Pin geführt. Außerdem steuert das Modul, je nachdem welche I2C-Paket-Art gesendet wird eine RGB-LED an.
In den nächsten Abschnitten erfolgt eine genauere Betrachtung der beiden Module. Insbesondere wird dabei auf die Konfiguration der AXI-Slave-Register und die Funktion der Module eingegangen.

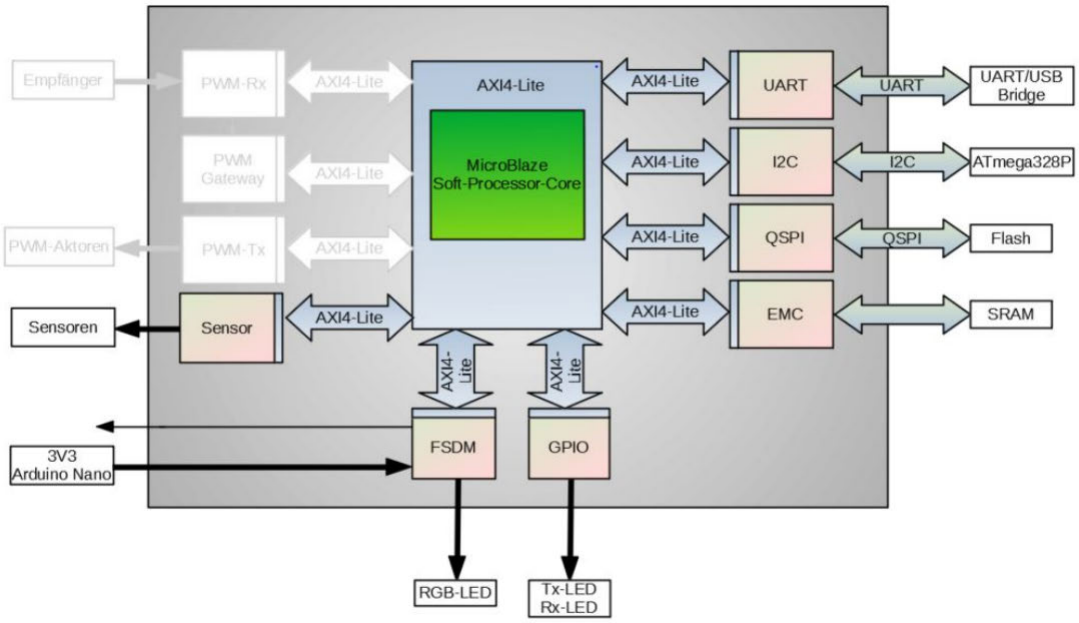


Abbildung 3.5: Hardware-Architektur des System-on-Chip zur Implementierung der Multi-Sensor-Signalverarbeitung [9, S.54]

3.3.2 Hardware-Design des Fail-Safe-Data-Monitoring

Abbildung 3.6 zeigt das Blockschaltbild des FSDM-Moduls. Das Modul ist über eine AXI4-Lite-Schnittstelle mit dem Mikroprozessor verbunden und besitzt drei Eingänge. Einen Eingang für den 100 MHz Systemtakt, einen für den Low-aktiven Reset und einen für die $3,3$ V des Arduino Nano. Über drei Ausgangssignale werden die RGB-LEDs angesteuert. Die Ansteuerung einer RGB-LED erfolgt in Abhängigkeit der gesendeten und empfangenen I2C-Paket-Art. In Tabelle 3.5 ist die Farbcodierung aufgeführt.

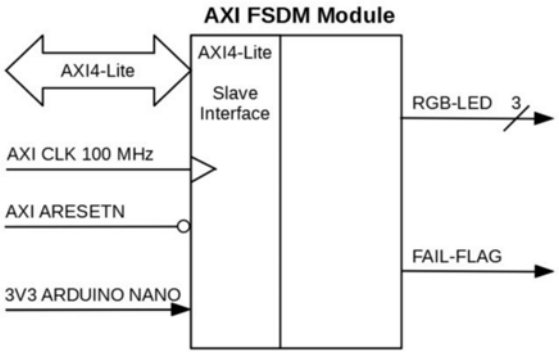


Abbildung 3.6: Blockschaltbild des FSDM-Moduls [9, S.55]

Tabelle 3.5: Farbcodierung der RGB-LEDs [9, S. 55]

I2C-Paket-Art	**LED-Farbe**
Select	aus
Konfiguration	weiß
Status	grün
Steuerung	hellblau
Daten	dunkelblau
Sonderfall	**LED-Farbe**
Fehlerfall Arduino Nano	rot

Die Befehlstypen des I2C-Frames werden in das AXI4-Lite-Slave-Register, welches für dieses Modul angelegt wurde, abgespeichert. Das Modul ist mit insgesamt vier Registern ausgestattet, wobei nur das Register 0 genutzt wird. Das liegt an der minimal möglichen Register-Anzahl, die durch die Entwicklungsumgebung festgelegt ist. Die Register-

Beschreibung ist in Abbildung 3.7 dargestellt. Es handelt sich hierbei um das Konfigurationsregister 0 des FSDM-Moduls.

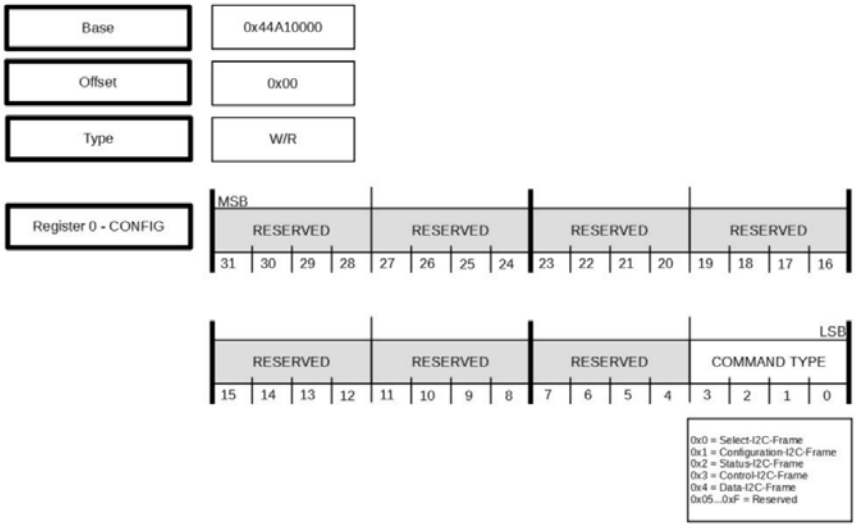


Abbildung 3.7: Register-Beschreibung FSDM-Modul [9, S.56]

Das Modul besitzt zwei Zustände und ist als Mealy-Automat entworfen. In Abbildung 3.8 ist der Zustandsautomat abgebildet. Die Zustände des Moduls sind RUN und FAIL. Im Zustand RUN liegen die $3,3\ V$ des Arduino Nano an und abhängig vom Registerinhalt wird die RGB-LED angesteuert. Die $3,3\ V$ sind dabei an einen digitalen Input-Pin angeschlossen. Sobald die Ausgangsspannung abfällt und somit ein Low-Pegel erkannt wird, erfolgt der Zustandswechsel. Im Zustand FAIL liegt vermutlich ein Fehler im ATmega238P vor und das FAIL-FLAG-Ausgangssignal wird gesetzt. Außerdem wird die rote RGB-LED angesteuert. Der Zustand kann nur durch einen System-Reset verlassen werden.

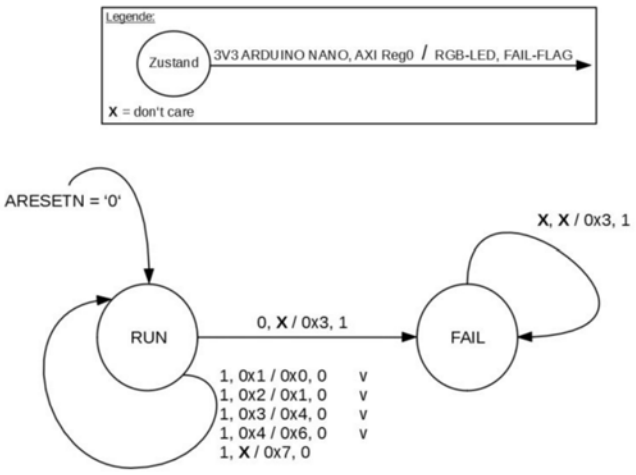


Abbildung 3.8: Zustandsdiagramm FSDM-Modul [9, S.57]

3.3.3 Hardware-Design der Multi-Sensor-Signalverarbeitung

Das AXI-Sensor-Modul ist für die Ansteuerung der Sensoren verantwortlich. Der I2C-Master gibt die Ansteuerung vor und das Sensor-Modul gibt die ermittelten Entfernungsinformationen der Sensoren zurück. Das Sensor-Modul besteht aus mehreren Modulen, welche in diesem Abschnitt genauer analysiert werden.

3.3.3.1 Hardware-Design des Sensor-IP-Cores

Abbildung 3.9 zeigt das Blockschaltbild des AXI-Sensor-Moduls. Das Modul ist, wie das FSDM-Modul, über eine AXI4-Lite-Schnittstelle mit dem Mikroprozessor verbunden. Diese Schnittstelle verfügt über sechs Slave-Register, über die das Modul konfiguriert und gesteuert wird. Außerdem werden die ermittelten Sensordaten in den Registern gespeichert. Der IP-Core nutzt zwei Taktsignale, einmal die 100 MHz als Systemtakt und zusätzlich noch einen $5,2$ MHz-Takt. Warum ein asynchroner Takt genutzt wird, ist in Abschnitt 3.3.3.3 genauer erläutert. Der IP-Core besteht im wesentlichen aus zwei Modulen. Einmal dem Sensor Array Modul und dem Sensor Modul. Für jeden Ultraschallsensor wir ein Sensor Modul benötigt, weshalb insgesamt sechs Sensor Module implementiert sind. Die Sensor Module sind alle identisch aufgebaut. Das Sensor Modul generiert den des zum Start eines Messvorgangs benötigten Trigger-Impuls und ermittelt aus dem

eingehenden ECHO-Signal die Entfernungsinformation. Nach jeweils einem Messdurchlauf werden die Entfernungsinformationen an das Sensor Array Modul übermittelt. Die Register-Beschreibungen sind in den Abbildungen 3.10-3.15 dargestellt.

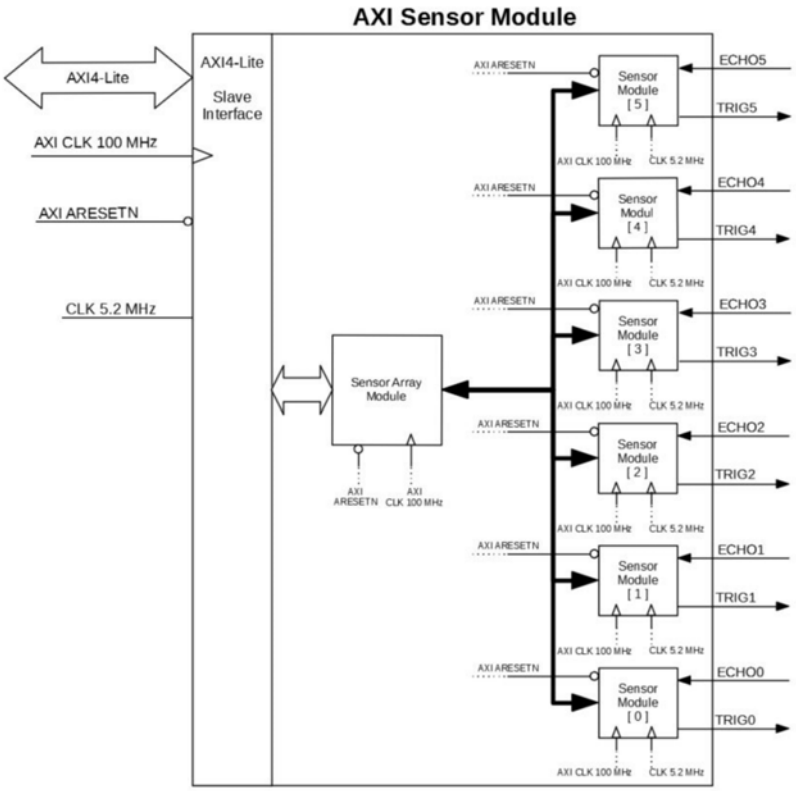


Abbildung 3.9: Blockschaltbild des AXI-Sensor-Moduls [9, S.67]

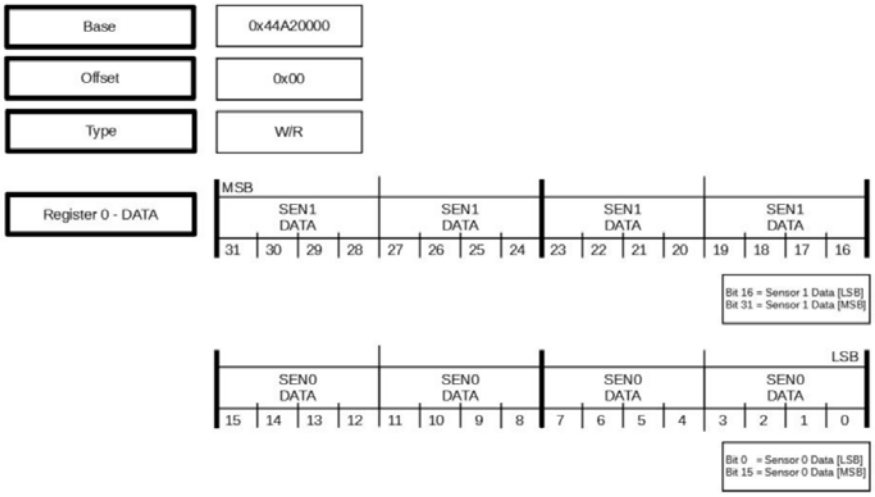


Abbildung 3.10: Register-Beschreibung AXI Sensor Modul (Register 0) [9, S.68]

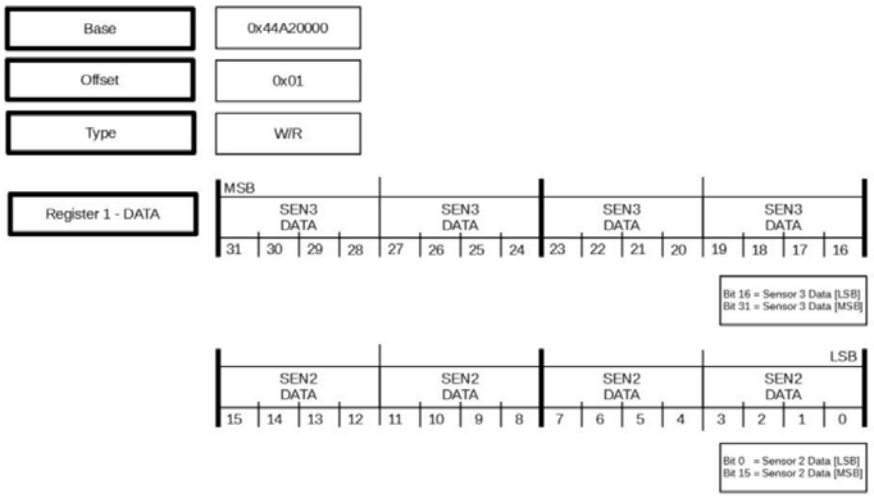


Abbildung 3.11: Register-Beschreibung AXI Sensor Modul (Register 1) [9, S.68]

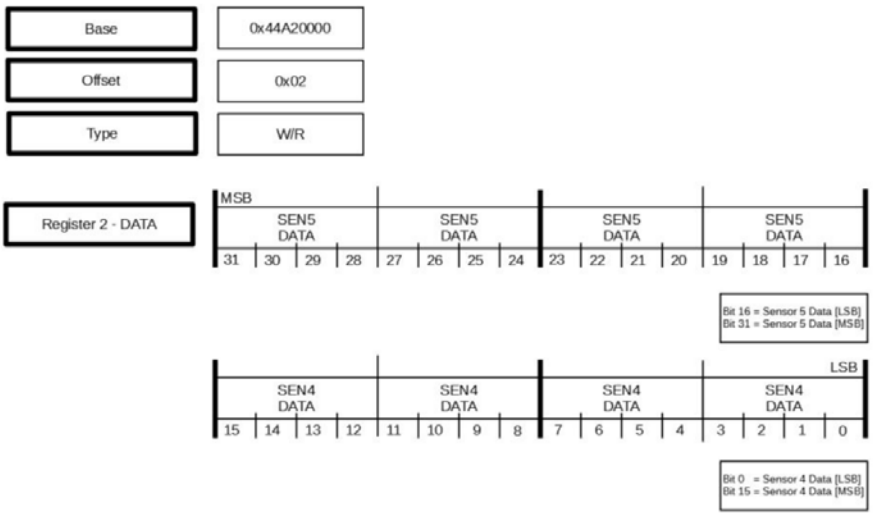


Abbildung 3.12: Register-Beschreibung AXI Sensor Modul (Register 2) [9, S.69]

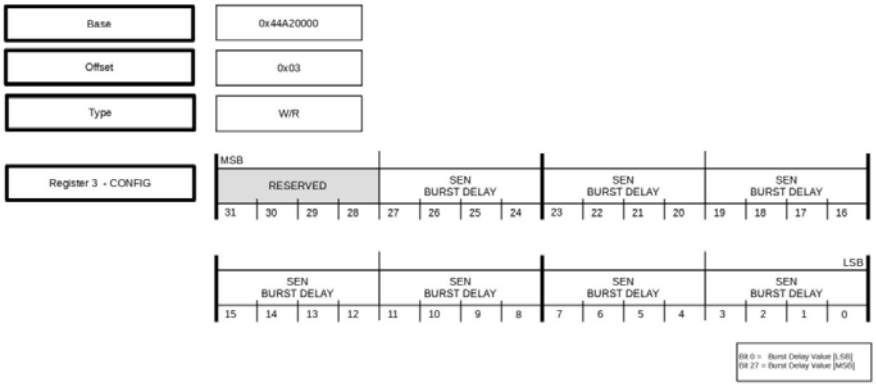


Abbildung 3.13: Register-Beschreibung AXI Sensor Modul (Register 3) [9, S.69]

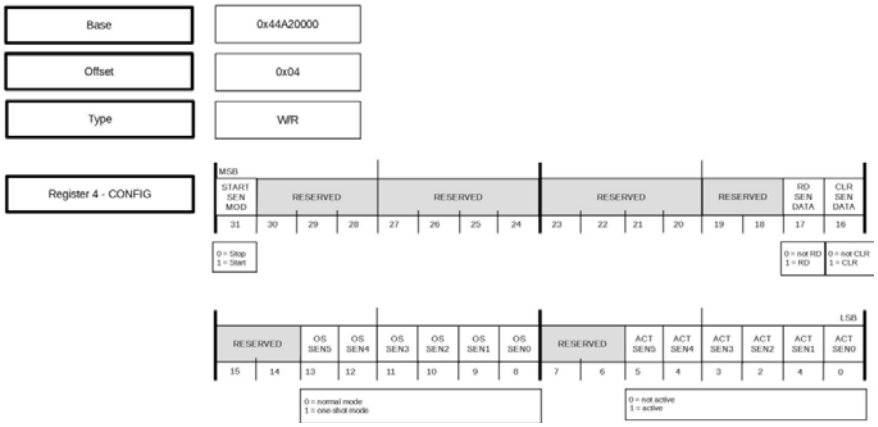


Abbildung 3.14: Register-Beschreibung AXI Sensor Modul (Register 4) [9, S.70]

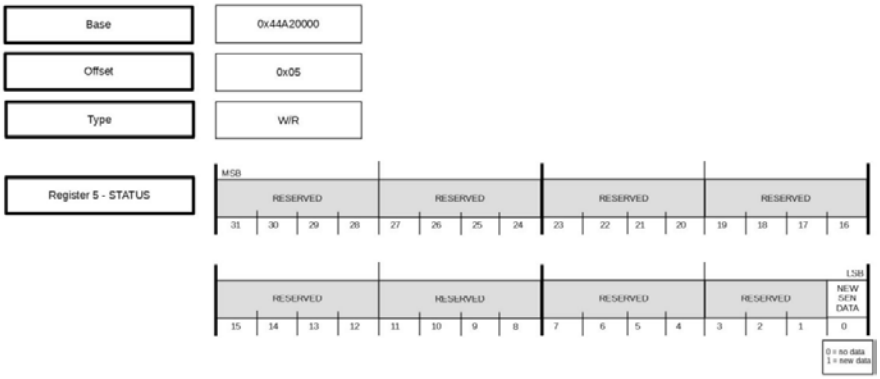


Abbildung 3.15: Register-Beschreibung AXI Sensor Modul (Register 5) [9, S.70]

3.3.3.2 Hardware-Design des Sensor-Array-Moduls

Das Sensor-Array-Modul ist die zentrale Steuereinheit des IP-Blocks und ist direkt mit den AXI-Registern verbunden. Das Modul steuert die Sensor-Module und wird mit einer Taktfrequenz von 100 MHz betrieben. Abbildung 3.16 zeigt das Blockschaltbild des Sensor-Array-Moduls. Ein Messvorgang wird gestartet, wenn das Bit 31 in Register 4 durch den I2C-Master gesetzt wird. Daraufhin erfolgt die sequenzielle Ansteuerung der Sensor Module und zwar von Sensor 0 aufsteigend. Das Aktivierungsintervall innerhalb eines Messvorgangs ist dabei zeitlich nicht konstant, sondern erfolgt jeweils nach Vorliegen eines gültigen Messwertes des aktivierten Sensors [9, S.58]. Dieses Vorgehen soll die

gegenseitige Beeinflussung der Ultraschallsensoren reduzieren. Wenn von allen Sensoren Messwerte vorliegen, wird Bit 0 in Register 5 gesetzt. Die Quittierung und Löschung des Bits erfolgt ausschließlich über den I2C-Master. Der Messvorgang wird so lange fortgesetzt, bis Bit 31 in Register 4 wieder gelöscht wird [9, S.58].
Um die Datenintegrität zu erhalten, werden die Sensorwerte in den AXI-Registern nach jeweils einem Messdurchlauf aktualisiert. Somit ist gewährleistet, dass die Sensorwerte untereinander konsistent sind. Diese werden nach jedem Messdurchlauf durch die nachfolgenden aktuellen Sensorwerte ersetzt. Nach setzen des Bit 17 in Register 4, wird das Ersetzen unterbrochen und die Übertragung der Sensordaten über die I2C-Schnittstelle erfolgt. Da je I2C-Frame nur 32-Bit gesendet werden können, sind für einen vollständigen Lesezugriff drei Leseanfragen des Masters nötig. Das Setzen des Bit 16 in Register 4 führt zum Löschen der abgerufenen Sensordaten in den Registern 0-2. Anschließend werden Bit 16 und 17 in Register 4 durch den I2C-Master zurückgesetzt [9, S.58]. Damit ist der Auslesevorgang abgeschlossen und Bit 0 in Register 5 wird quittiert.

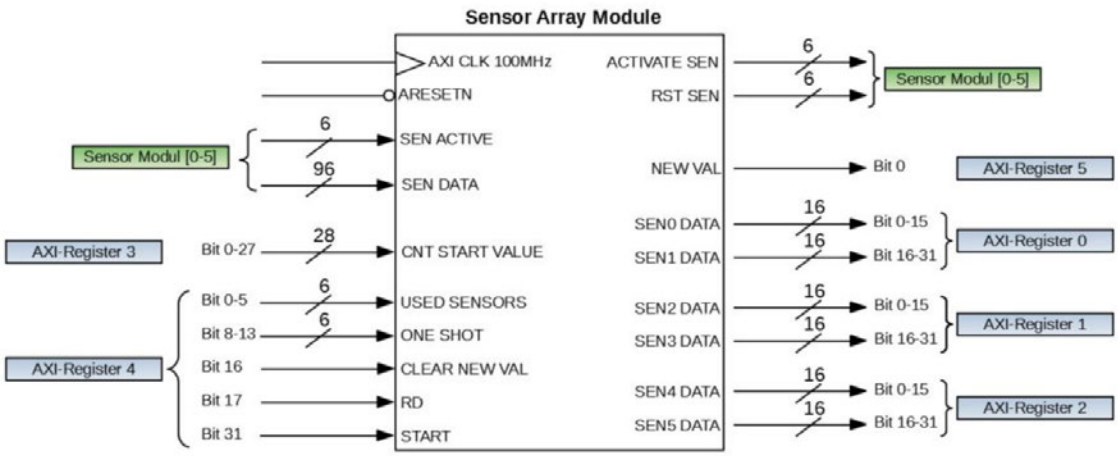


Abbildung 3.16: Blockschaltbild des Sensor-Array-Moduls [9, S.70]

3.3.3.3 Hardware-Design des Sensor-Moduls

Das Sensor-Modul setzt sich aus insgesamt vier Modulen zusammen. Eine Stereinheit, einen Zähler, einen Flankendetektor und einem Taktteiler. Das Hardware-Module wird mit zwei unterschiedlichen externen Taktfrequenzen betrieben. Während die Steuereinheit mit dem Systemtakt 100 MHz arbeitet, arbeitet der Taktteiler mit einem $5,2$ MHz

Takt. Der Taktteiler reduziert die Frequenz von $5,2\ MHz$ auf 325 kHz, die wiederum von dem Zähler und dem Flankendetektor genutzt werden. Für den Taktteiler wurde die Methode des Clock-Gating genutzt [9, S.59]. Die Clock-Domäne $5,2\ MHz$ wurde gewählt, da sich dadurch eine geringere Anzahl an benötigten Flip-Flops ergibt. Diese Clock-Domäne ist nicht nötig, da der Frequenzteiler auch mit dem Systemtakt gebaut werden kann. Insbesondere die Clock-Verbindungen zum Counter und Edge-Dtektor sind nicht nötig, da diese sowieso nur mit einem Clock-Enable-Signal geschaltet werden.
Wenn das Sensor-Modul vom Sensor-Array-Modul aktiviert wurde, sendet dieses ein Trigger-Signal an den angeschlossenen Ultraschallsensor, welcher wiederum einen Messvorgang startet. Danach entsperrt die Steuereinheit den Zähler. Sobald eine steigende Flanke durch den Flankendetektor des eingehenden ECHO-Signals erkannt wird, beginnt der Zählvorgang. Durch eine fallende Flanke wird der Zählvorgang gestoppt. Dadurch liegt die Pulsbreite des ECHO-Signals als Zählwert vor und wird über ein Vier-Wege-Handshake-Verfahren von der Steuereinheit eingelesen und in ein internes Register geschrieben. Dieses Register wird anschließend vom übergeordneten Sensor-Array-Modul ausgelesen. Die Steuereinheit und der Zähler sind als Finite-State-Machines entworfen worden [9, S.59]. Abbildung 3.17 zeigt das Blockschaltbild des Sensor-Moduls.

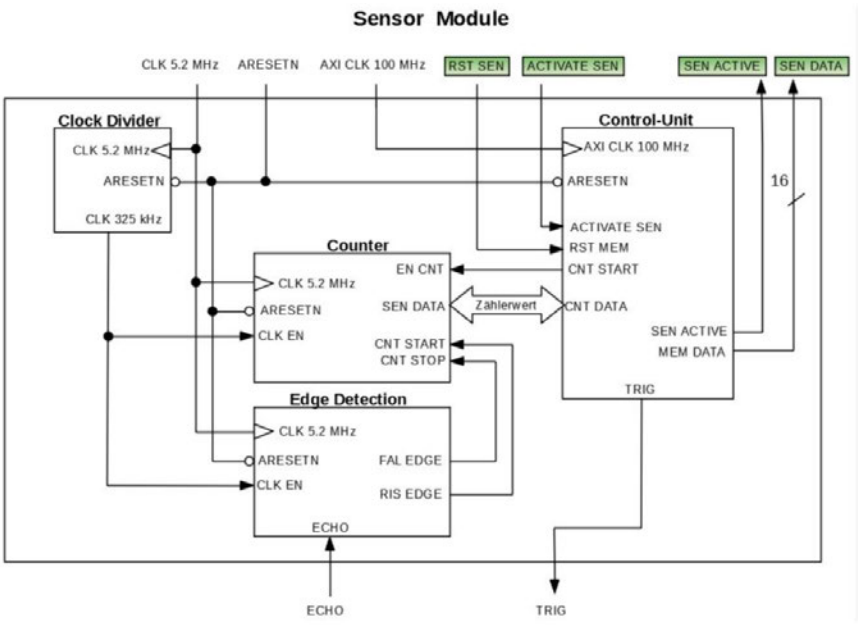


Abbildung 3.17: Blockschaltbild des Sensor-Moduls [9, S.71]

4 Anforderungsanalyse

Auf Grundlage der Analyse des vorigen Kapitels erfolgt die Anforderungsanalyse für das zu entwerfende PWM-Modul zur Motorenansteuerung. Aus Treffen mit Professor Dr. Marc Hensel als Auftraggeber und aus der Anforderungsanalyse werden die funktionalen und nicht-funktionalen Anforderungen definiert.

4.1 Stakeholder

In diesem Abschnitt werden die Stakeholder ermittelt. Stakeholder sind Interessensgruppen innerhalb eines Projekts. Sowohl interne, als auch externe Personen, Personengruppen oder auch Institutionen können damit gemeint sein. Diese Interessensgruppen haben ein Interesse an dem System oder sind direkt oder indirekt durch das Projekt betroffen. In dieser Arbeit haben sich die gleichen Stakeholder herausgestellt, wie in Kolja Gries Arbeit [9]. In Tabelle 4.1 sind die Stakeholder definiert.

Tabelle 4.1: Funktionale Anforderungen an das System

Gewerk	Profil
Projektleiter	Seltener Anwender des Systems. Zuweisung von Teilprojekten an die Entwickler. Ünterstützt bei der Konzeption von Softwaremodulen. Keine Kenntnisse in der Hardwarebeschreibungssprache VHDL.
Entwickler - Level 1	Anwender des Systems in Teilprojekten. Häufig Student im Grundstudium mit Grundkenntnissen in der Programmiersprache C. Keine Kenntnisse in der Hardwarebeschreibungssprache VHDL.
Entwickler - Level 2	Anwender des Systems in Teilprojekten. Häufig ein Student im Hauptstudium. Fundierte Kenntnisse in der Programmiersprache C. Grundkenntnisse in der Hardwarebeschreibungssprache VHDL
Entwickler - Level 3	Anwender des Systems in Teilprojekten. Person, die ein Projekt im Rahmen einer Abschlussarbeit übernimmt. Fundierte Kenntnisse und Erfahrungen in der Programmiersprache C. Fundierte Kenntnisse in der Hardwarebeschreibungssprache VHDL.
Fahrer	Anwender des Systems als Kunde. Steuert das RC-Fahrzeug.

4.2 Anwendungsfälle

Die Anwendungsfälle stellen die Funktionalitäten und das von außen sichtbare Verhalten des Systems aus Sicht von Nutzern oder anderen Systemen dar. Die Anwendungsfälle wurden mit den Stakeholdern, insbesondere dem Projektleiter Marc Hensel erarbeitet. Die Anwendungsfälle der Fahrzeugsteuerung sind in Abbildung 4.1 dargestellt.

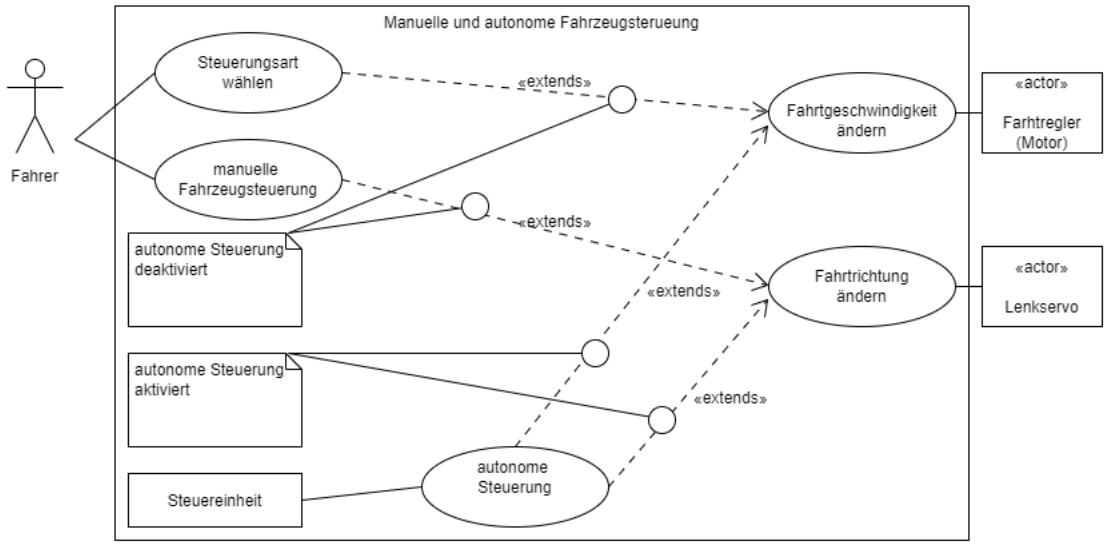


Abbildung 4.1: Anwendungsfälle der Fahrzeugsteuerung

4.3 Anforderungen

In Absprache mit dem Auftraggeber Marc Hensel werden funktionale und nicht-funktionale Anforderungen an das System definiert. Die funktionalen Anforderungen legen fest, was das System an Funktion beinhalten soll. Die nicht-funktionalen Anforderungen ´sind Systemspezifikationen, die prinzipiell keine Auswirkung auf die Funktionalität haben. In Tabelle 4.2 sind die funktionalen Anforderungen gelistet und in Tabelle 4.3 die nicht-funktionalen Anforderungen.

4.3.1 Funktionale Anforderungen an das System

Tabelle 4.2: Funktionale Anforderungen an das System

Nr.	Anforderung	Priorität
F1	Die System-Hardware soll so ausgelegt sein, dass es die manuelle und autonome Fahrzeugsteuerung durch den Fahrer ermöglicht.	Hoch
F2	Das System soll so ausgelegt sein, das es über vorhandene Arduino-Bibliotheken angesteuert werden kann.	Hoch
F3	Das System soll die Umstellung der Steuerungsart (manuelle oder autonome Steuerung), welche vom Fahrer initialisiert wird, ermöglichen.	Hoch
F4	Das System soll die PWM-Signale des Empfängers auslesen.	Hoch
F5	Das System stellt dem Mikrocontroller die Empfängerwerte über eine Schnittstelle zur Verfügung.	Niedrig
F6	Die PWM-Kanäle sollen separat durch den Mikrocontroller ein und ausgeschaltet werden können.	Niedrig
F7	Das System soll die Konfiguration der PWM-Signale zur Ansteuerung der Aktoren über eine Schnittstelle ermöglichen.	Hoch
F8	Das System soll die PWM-Aktoren gemäß der Konfiguration durch den Mikrocontroller ansteuern.	Hoch

4.3.2 Nicht-funktionale Anforderungen an das System

Tabelle 4.3: Nicht-funktionale Anforderungen an das System

Nr.	Anforderung	Priorität
N1	Das System soll den Arduino Nano als zentrale Regel- und Steuereinheit nicht ersetzen.	Hoch
N2	In der Entwicklungsphase soll eine hohe Flexibilität von Hard- und Software gegeben sein.	Hoch
N3	Das System soll kompatibel für die Plattformen Arduino und Raspberry Pi sein.	Hoch
N4	Die Slave-Adresse des Systems soll softwareseitig konfigurierbar sein.	Hoch
N5	Der Duty-Cycle des PWM-Moduls für das autonome Fahren soll softwareseitig konfigurierbar sein	Hoch

5 Konzept und Design

Auf Grundlage der Anforderungsanalyse, wird in diesem Kapitel der Lösungsansatz für das System entworfen. Anschließend wird das Hardware-Design, insbesondere die Umsetzung in VHDL, genauer beschrieben.

5.1 Konzept

In diesem Abschnitt wird das Konzept des Lösungsansatzes für das System beschrieben. Hierbei wird auf das Konzept für die manuelle und autonome Fahrzeugsteuerung eingegangen.

5.1.1 Manuelle Fahrzeugsteuerung

Für die manuelle Fahrzeugsteuerung gibt es mehrere Lösungsansätze. Zum einen kann das PWM-Signal vom Empfänger abgegriffen und durch einen Zähler die Pulsbreite des Signals ermittelt werden. Der Zählwert der ermittelten Pulsbreite kann in ein internes Register geschrieben werden und mit diesem Zählwert wäre es möglich einen Duty Cycle zu entwerfen und über diesen Fahrtregler und Lenk-Servo anzusteuern. Eine weitere Möglichkeit ist das PWM-Signal vom Empfänger abzugreifen und direkt an den Fahrtregler und Lenk-Servo über entsprechende Ausgangs-Pins durch zu schleifen. Dieser Lösungsansatz ist einfacher und schneller umzusetzen, da bei diesem Ansatz kein Zähler und zusätzlicher Duty Cycle nötig ist und erfüllt die Systemanforderungen für das manuelle Fahren. Daher wird dieser Lösungsansatz für das System umgesetzt.

5.1.1.1 Synchronisationsfehler und MTBF

Zur Umsetzung des manuellen Fahrens muss das eingehende PWM-Signal zunächst synchronisiert werden. Dies ist notwendig, da das eingehende Signal asynchron zur Taktfrequenz seinen Zustand ändert. Dies wiederum kann zu einem Synchronisationsfehler führen. Ein Sysnchronisationsfehler tritt genau dann auf, wenn es zu einer Flankenänderung des Eingangssignals eines D-Flip-Flops, im weiteren Verlauf D-FF genannt,

während der *Setup*-Zeit t_{setup} oder *Hold*-Zeit t_{hold} kommt. Wenn dies geschieht, kann das Ausgangssignal des D-FF die folgenden drei Zustände annehmen:

1 Das Ausgangssignal wird zu '1'

2 Das Ausgangssignal wird zu '0'

3 Das D-FF gelangt in einen *metastabilen* Zustand. Das Ausgangssignal ist weder '1', noch '0'. Dieser Zustand löst sich nach der Zeit T_r und es wird einer der beiden oberen Zustände eingenommen, wobei nicht vorherzusagen ist welcher.

In Abbildung 5.1 ist dieses Verhalten abgebildet. Sollte das Ausgangssignal den Zustand 2 annehmen, würde dieser, nach dem Beispiel in dieser Abbildung, mit der nächsten Taktflanke Zustand 1 annehmen.

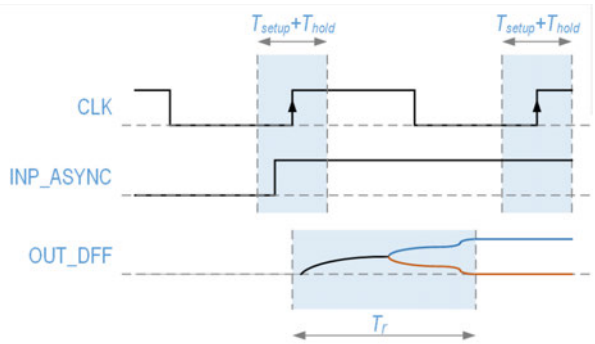


Abbildung 5.1: Synchronisationsfehler [11]

Die Zeit T_r ist die Zeit, die der D-FF braucht, um sich aus dem *metastabilen* Zustand zu lösen. Die Wahrscheinlichkeit, dass sich der D-FF noch nicht aus dem *metastabilen* Zustand gelöst hat lässt sich mit

$$P(T_r) = e^{-\frac{T_r}{\tau}} \tag{5.1}$$

bestimmen. τ ist die Abklingkonstante und ein technischer Parameter des D-FFs. Die durchschnittliche Zeit zwischen zwei Synchronisationsfehlern lässt sich durch die *MEAN TIME BETWEEN FAILURE* (MTBF) bestimmen. Die MTBF ist definiert durch

$$MTBF(T_r) = \frac{1}{R_{meta} \cdot P(T_r)} = \frac{e^{\frac{T_r}{\tau}}}{w \cdot f_{clk} \cdot f_d} \tag{5.2}$$

mit $R_{meta} = w \cdot f_{clk} \cdot f_d$, der durchschnittlichen Fehlerrate. Die Parameter von R_{meta} sind in Tabelle 5.1 gelistet.

Tabelle 5.1: Parameter R_{meta} [11]

f_{clk}	Taktfrequenz
f_d	Wechselrate des Eingangssignals (steigende Flanke pro Sekunde)
w	empfindliches Zeitfenster, technischer Parameter des D-FF

Tabelle 5.2 zeigt, dass die MTBF extrem stark von der Zeit T_r, die einem D-FF gegeben wird, um in einen stabilen Zustand nach der Metastabilität zu gelangen, abhängt. T_r ist der einzige Parameter der veränderbar ist, die anderen hängen von anderen System-Anforderungen ab. Daher wird die MTBF als Funktion von T_r ausgedrückt.

T_r	MTBF
0.0 ns	$4.00 * 10^{-05}$ sec (0.04 msec)
2.5 ns	$5.94 * 10^{-03}$ sec (5.94 msec)
5.0 ns	$8.81 * 10^{-01}$ sec (0.88 sec)
7.5 ns	$1.31 * 10^{+02}$ sec (131 sec)
10.0 ns	$1.94 * 10^{+04}$ sec (5.39 hours)
12.5 ns	$2.88 * 10^{+06}$ sec (3.33 days)
15.0 ns	$4.27 * 10^{+08}$ sec (1.36 years)
17.5 ns	$6.34 * 10^{+10}$ sec ($2.01 * 10^3$ years)
20.0 ns	$9.42 * 10^{+12}$ sec ($2.99 * 10^5$ years)
22.5 ns	$1.40 * 10^{+15}$ sec ($4.43 * 10^7$ years)
25.0 ns	$2.07 * 10^{+17}$ sec ($6.58 * 10^9$ years)
27.5 ns	$3.08 * 10^{+19}$ sec ($9.76 * 10^{11}$ years)
30.0 ns	$4.57 * 10^{+21}$ sec ($1.45 * 10^{14}$ years)
32.5 ns	$6.78 * 10^{+23}$ sec ($2.15 * 10^{16}$ years)
35.0 ns	$1.01 * 10^{+26}$ sec ($3.19 * 10^{18}$ years)

Abbildung 5.2: Beispielberechnungen für die MTBF in Abhängigkeit von T_r [4]

5.1.1.2 2-FF-Synchronisierer

Seien T_c, T_{setup} und T_{comb} die Taktfrequenz, die Setup-Zeit und die Verzögerungszeit der kombinatorischen Logik. So lässt sich T_r wie folgt bestimmen:

$$T_r = T_c - T_{comb} - T_{setup} \tag{5.3}$$

Ziel ist es, ein möglichst großes T_r zu erhalten. Dies gelingt, in dem man die Zeitverzögerung der kombinatorischen Logik $T_{comb} = 0$ bekommt. Um dies zu erreichen, wird ein 2-FF Synchronisierer eingesetzt. Bei einem 2-FF-Synchronisierer werden zwei D-FFs hintereinander geschaltet. Diese Schaltung ist in Abbildung 5.3 abgebildet. So lässt sich

die Zeitverzögerung der kombinatorischen Logik T_{comb} raus rechnen und man erhält eine maximale Auflösungszeit T_r. Die Auflösungszeit bei einem 2-FF-Synchronisierer ist durch

$$T_r = T_c - T_{setup} \tag{5.4}$$

gegeben.

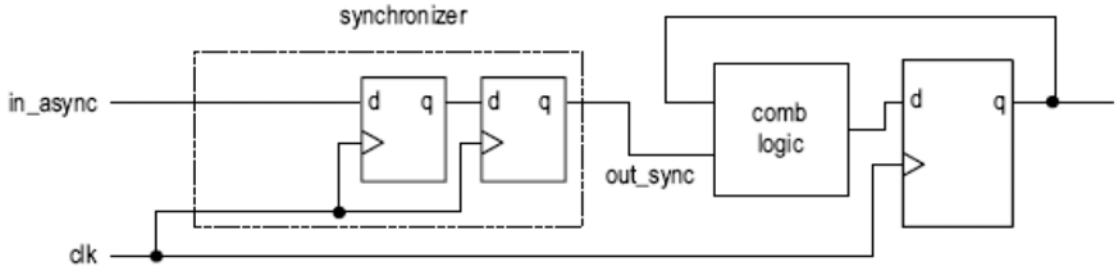


Abbildung 5.3: 2-FF-Synchronisierer [4]

5.1.2 Autonome Fahrzeugsteuerung

Für die autonome Fahrzeugsteuerung gibt es mehrere mögliche Lösungsansätze. In jedem Fall muss ein Duty Cycle entwickelt werden. Um diesen umzusetzen, gibt es einerseits die Möglickeit, über ein *Clock Enable* einen Zähler anzusteuern. Für *Clock Enable* wird ein Frequenzteiler benötigt, welcher wiederum Ressourcen kostet, da mehr Flip-Flops nötig sind. Daher und der wird auf ein *Clock Enable* verzichtet und ein großer Zähler gebaut. Dieser ist auch einfacher umzusetzen und erfüllt genauso die Anforderungen an das autonome Fahren. Das Konzept des Duty Cycle wird im nächsten Abschnitt genauer erläutert.

5.1.2.1 Duty Cycle

Das Prinzip des Zählers folgt einem einfachen und gut umzusetzenden Konzept. Es braucht hierfür lediglich drei Register. Eine Zählvariable N, einen Endwert N_{max} als feste Konstante und eine variabel durch den Entwickler vorgebbare Variable N_{duty}. Über N_{max} lässt sich die Frequenz des Duty Cycle einstellen. Seien f_{clk} und f_{duty} die Taktfrequenz und die einzustellende Duty Cycle-Frequenz, so kann N_{max} mit

$$N_{max} = \frac{f_{clk}}{f_{duty}} \tag{5.5}$$

bestimmt werden. Mit der Annahme $0 < N_{duty} < N_{max}$ lässt sich nun der Duty Cycle bauen. Zunächst werden N_{duty} und N_{max} die zuvor ermittelten Werte zugewiesen. Diese Werte müssen je nach Anforderung an den Duty Cycle entsprechend bestimmt werden. Der Zählvariable N wird der Wert 0 zugewiesen. Der Duty Cycle funktioniert dann auf folgende Weise:

- Mit Betreten des Duty Cycle, bekommt der Output eine '1'.
- Solange $N < N_{max} - 1$ wahr ist, wird N um 1 inkrementiert ($N <= N + 1$).
- Solange $N < N_{duty} - 1$ wahr ist, bekommt der Output eine '1' zugewiesen.
- Wenn $N < N_{duty} - 1$ falsch ist, bekommt der Output eine '0' zugewiesen.
- Wenn $N < N_{duty} - 1$ falsch ist, springt der Algorithmus aus dem Duty Cycle.

Bevor der Algorithmus wieder den Duty Cycle durchläuft, werden die Register neu initialisiert.

5.2 Design

In diesem Kapitel wird das Design des Systems beschrieben. Dabei wird auf das Hardware-Design des PWM-Moduls und die Definition des AXI4-Interface eingegangen. Der Algorithmus des VHDL-Codes wird anhand eines ASMD-Charts beschrieben. Außerdem wird auf das von Marc Hensel erstellte Platinen-Design kurz eingegangen.

5.2.1 Platinen-Design

Da das Coprozessor-System zunächst auf einem Breadboard aufgebaut war, ist in Absprache mit Kolja Gries und Marc Hensel ein Platinen-Design entwickelt wurden. Bei dem Design werden das PCA9306-Board und die drei BBS138 Boards durch das TXS0108E [3] ersetzt, da dieses über 8 Kanäle verfügt und I2C fähig ist. Die Trigger-Signal-Pins werden statt über den Level-Shifter, direkt mit den Ultraschallsensoren verbunden. Das Design für die Platine stammt von Marc Hensel und ist in Abbildung 5.4 abgebildet.

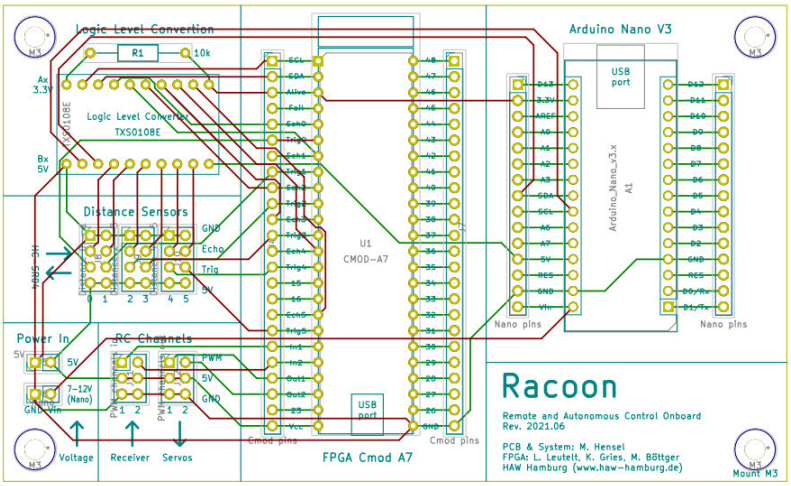


Abbildung 5.4: PCB-Design des Coprozessor-Systems

5.2.2 Hardware-Design des PWM IP Cores

Abbildung 5.5 zeigt das Blockschaltbild des PWM-Moduls. Das Modul besitzt vier Eingänge. Einmal den AXI-Systemtakt von 100 MHz, den AXI-Reset und die beiden PWM-Signale für den Fahrtregler und Lenk-Servo. Die Eingänge sind mit PWMIN_DR (DR für Drive) und PWMIN_DIR (DIR für Direction) bezeichnet und sind die PWM-Signale des Controllers. Das Modul ist außerdem über eine AXI4-Lite-Schnittstelle mit dem Mikroprozessor verbunden. Diese Schnittstelle verfügt über vier Slave-Register über die das Modul konfiguriert und gesteuert wird. Die Register werden vom Master, dem Arduino Nano, beschrieben. In der Steuerungsart des manuellen Fahrens, werden die eingehenden PWM-Signale durch die beiden PWM-Sync Module synchronisiert und durch die PWM-Ctrl-Module direkt an die entsprechenden Ausgangspins durchgeschliffen.

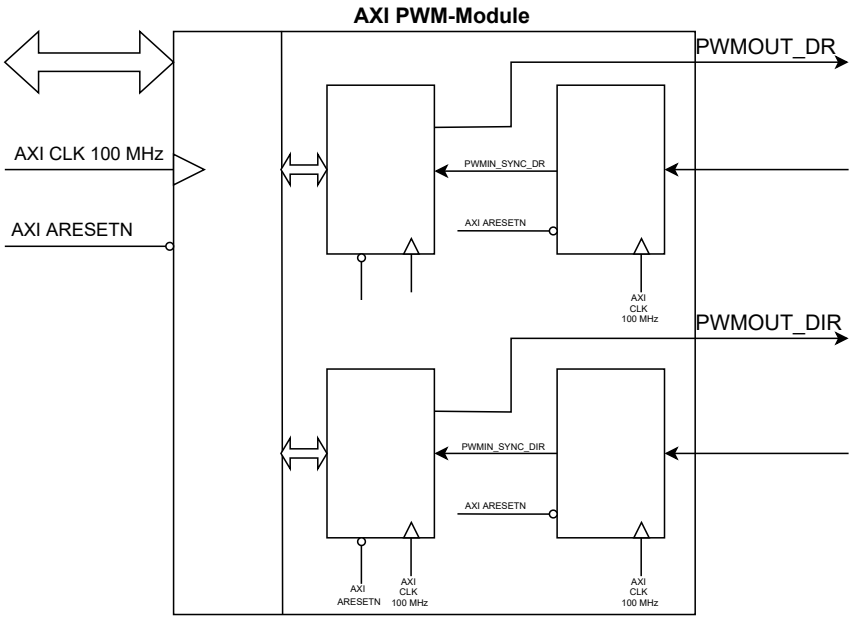


Abbildung 5.5: Blockschaltbild des AXI-PWM-Moduls

5.2.3 AXI4-Lite Slave Interface

Das PWM-Modul besitzt insgesamt vier AXI-Register. Dies ist dem geschuldet, dass ein Custom IP Core mit mindestens vier AXI-Registern konfiguriert werden muss. Dies ist von der Entwicklungsumgebung so festgelegt. Ein AXI-Register ist 32-Bit groß. In Tabelle 5.2 ist die Konfiguration der Register gelistet.

Tabelle 5.2: Konfiguration des AXI4-Lite Slave Interface

Slave-Register (Bit)	Register-Wert	Funktion
Slave-Register 0 (0-24)	NMAX	24-Bit-Vektor zur Einstellung der Frequenz des Duty Cycle
Slave-Register 1 (0-24)	NDUTY_DR	24-Bit-Vektor zur Steuerung des Duty Cycle (Fahrtregler)
Slave-Register 2 (0-24)	NDUTY_DIR	24-Bit-Vektor zur Steuerung des Duty Cycle (Lenk-Servo)
Slave-Rgister 3 (0)	RUN	Bit zum starten des PWM-Moduls
Slave-Rgister 3 (1)	AUTO	Bit zur Auswahl der Steuerungsart '0' für die manuelle Fahrzeugsteuerung '1' für die autonome Fahrzeugsteuerung

Die in der Tabelle aufgeführten Register-Werte werden von dem Master beschrieben. Die 24-Bit für die die Register-Werte NMAX, NDUTY_DR und NDUTY_DIR erklären sich aus den Systemspezifikationen. Der Systemtakt beträgt $f_{clk} = 100\ MHz$ und die einzustellende Frequenz für den Duty Cycle beträgt $f_{duty} = 60\ Hz$, was der PWM-Frequenz des Dart Truggy entspricht, wie aus den Grundlagen zu entnehmen ist. Somit lässt sich NMAX nach Formel 5.1.2.1 bestimmen.

$$NMAX = \frac{100\ MHz}{60\ Hz} \approx 1666667$$

Mit NMAX lässt sich nun die nötige Bit-Breite des Vektors bestimmen.

$$2^n = NMAX \tag{5.6}$$

$$\Rightarrow n = log_2(NMAX) \approx 20,67 \tag{5.7}$$

Da das I2C-Layout so aufgebaut ist, dass Byte-weise über Payloads gelesen und geschrieben wird, werden für die Bit-Vektoren 24 Bit verwendet. So werden drei Byte in die jeweiligen Register geschrieben.

5.2.4 Hardware-Design des PWM-CTRL-Moduls

Die Module sind als *Finite-State-Machines* mit Daten-Pfad (FSMD) entworfen. Abbildung 5.6 zeigt das ASMD-Chart für die PWM-Module. Der Algorithmus wird nachfolgend exemplarisch für das $PWM - Ctrl - DR - Modul$ erläutert und gilt genauso für das $PWM - Ctrl - DIR - Modul$, da diese gleich aufgebaut sind.

IDLE

Ausgangspunkt des Algorithmus ist der Zustand IDLE. Dieser Zustand wird solange nicht verlassen, bis RUN durch den Master gesetzt wird. Ist RUN gesetzt, wird abgefragt, welche Steuerungsart gewählt ist. Das geschieht über AUTO. Ist AUTO gesetzt, wird der Zustand IDLE verlassen und der Zustand PWMCTRL_INIT wird betreten. Ist AUTO nicht gesetzt, wird IDLE ebenfalls verlassen und der Zustand PWMFWD betreten.

PWMFWD

Der Zustand PWMFWD ist für die manuelle Fahrzeugsteuerung zuständig und schleift das synchronisierte PWM-Signal durch. Beim Betreten des Zustandes, wird PWMOUT_DR das synchronisierte PWM-Signal PWMIN_SYNC_DR zugewiesen. Anschließend erfolgt die Abfrage, ob RUN nach wie vor gesetzt ist. Sollte dies

nicht der Fall sein, wird der Zustand PWMFWD verlassen und Zustand IDLE erneut betreten. Wenn RUN noch gesetzt ist, folgt die Abfrage, ob AUTO gesetzt ist oder nicht. Sollte AUTO gesetzt sein, wird der Zustand PWMFWD verlassen und der Zustand PWMCTRL_INIT wird betreten. Sollte AUTO nach wie vor nicht gesetzt sein, wird PWMFWD verlassen und mit der nächsten Taktflanke wieder betreten.

PWMCTRL_INIT

Beim Betreten des Zustands werden die internen Register initialisiert. NMAXINT und NDUTYINT_DR sind interne Register und bekommen die Werte aus dem AXI4-Lite-Slave-Register zugewiesen. N is die Zählvariable für den Duty Cycle und wird mit 0 initialisiert. PWMOUT_DR wird ein '1'-Signal zugewiesen. Anschließend erfolgt die Abfrage RUN='1'. Wenn diese wahr ist, wird der Zustand PWMCTRL_INIT verlassen und PWMCTRL_DUTY betreten. Wenn RUN='1' falsch ist, wird PWMCTRL_INIT verlassen und der Zustand IDLE betreten.

PWMCTRL_DUTY

Im Zustand PWMCTRL_DUTY wird der Duty Cycle für das autonome Fahren, wie in Kapitel 5.1.2.1 beschrieben, umgesetzt. Solange die Bedingung $N < NMAX - 1$ erfüllt ist wird N inkrementiert. Mit der nächsten Abfrage $N < NDUTY_DR - 1$ wird PWMOUT_DR '1' gesetzt, wenn diese Bedingung erfüllt ist. Ist diese Bedingung nicht erfüllt, wird PWMOUT_DR '0' gesetzt. Anschließend wird in beiden Fällen der Zustand PWMCTRL_DUTY verlassen und mit der nächsten Taktflanke wieder betreten. Sobald die Bedingung $N < NMAX - 1$ nicht mehr erfüllt ist, wird abgefragt, ob AUTO=1 ist. Ist dies wahr, wird der Zustand PWMCTRL_DUTY verlassen und PWMCTRL_INIT wird betreten. Ist die Bedingung falsch, wird der Zustand ebenfalls verlassen und der Zustand PWMFWD wird betreten.

Beim Duty Cycle ist darauf zu achten, dass die Pulsdauer τ, wie im Grundlagen-Kapitel beschrieben, zwischen 1 ms und 2 ms liegt. Mit $NMAX = 1666667$ lassen sich die Werte für die obere und untere Grenze, sowie für die neutrale Stellung, der Pulsdauer

bestimmen.

$$\begin{aligned} \frac{T}{\tau_R} &= \frac{NMAX}{NDUTY} \\ \Rightarrow NDUTY &= NMAX \cdot \frac{\tau_R}{T} \\ \Rightarrow NDUTY_{min} &= 1666667 \cdot \frac{1\ ms}{16,67\ ms} \approx 99980 \\ \Rightarrow NDUTY_N &= 1666667 \cdot \frac{1,5\ ms}{16,67\ ms} \approx 149970 \\ \Rightarrow NDUTY_{max} &= 1666667 \cdot \frac{2\ ms}{16,67\ ms} \approx 199960 \end{aligned} \tag{5.8}$$

Mit $99980 < NDUTY < 199960$ ist der Wertebereich, den NDUTY annehmen darf, klar definiert. Diese Grenzen werden nicht auf Hardware-Ebene abgefragt, sondern müssen auf Software-Seite durch zum Beispiel *if-Abfragen* eingehalten werden.

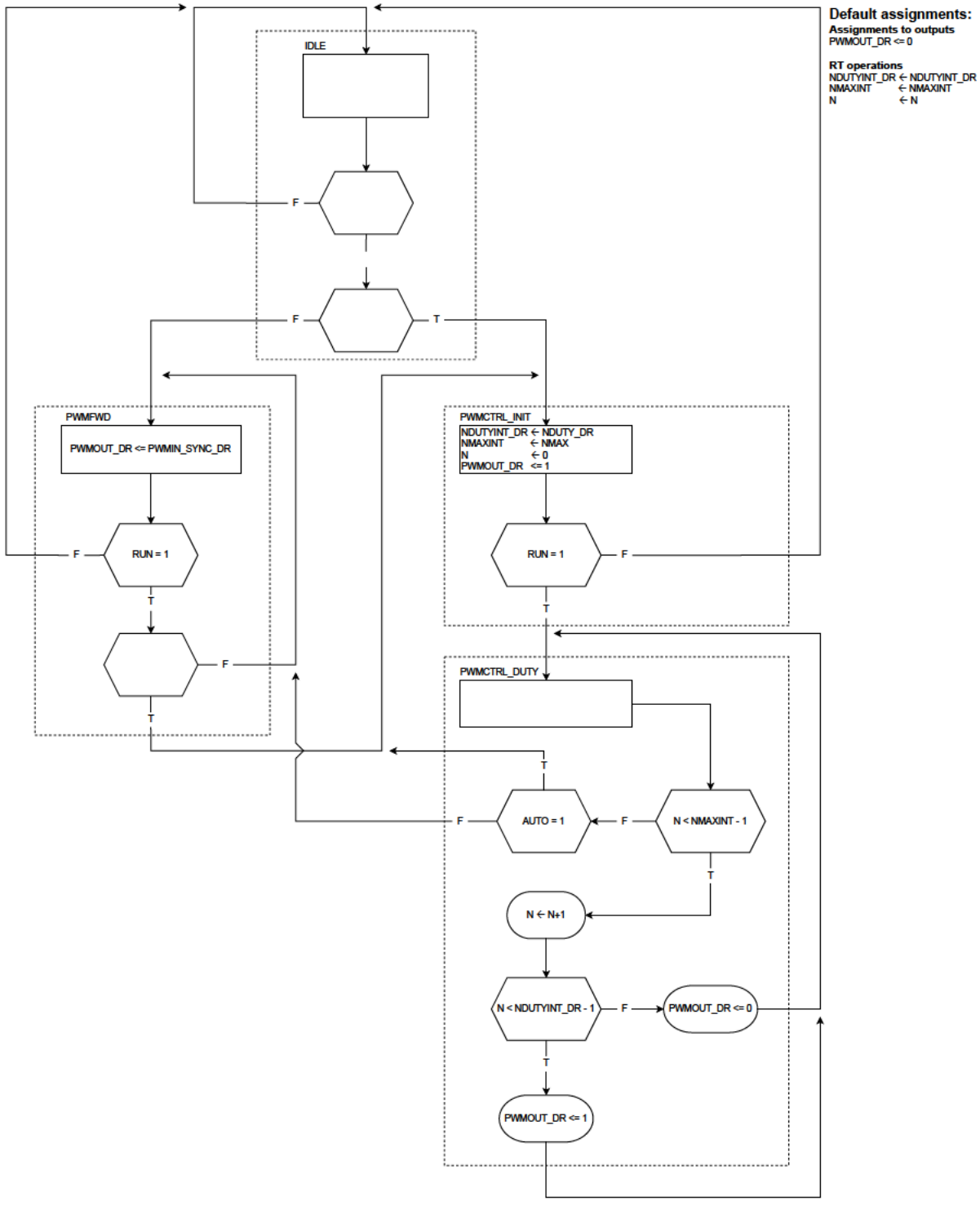


Abbildung 5.6: ASMD-Chart für des PWM-Ctrl-Dr-Modul

6 Umsetzung

In diesem Kapitel wird die Umsetzung des Konzepts dargestellt. Es wird hierbei auf die Inbetriebnahme des bestehenden Coprozessor-Systems, die Umsetzung des PWM-Moduls und die Einbindung des PWM-Moduls in das bestehende System eingegangen.

6.1 Inbetriebnahme des Coprozessor-Systems

In diesem Abschnitt erfolgt die Inbetriebnahme des bestehenden Coprozessor-Systems. Das System wurde mit dem Software-Tool Vivado 2019.1 von Xilinx entwickelt. Für die Umsetzung des PWM IP Cores wird die Version Vivado 2021.1 verwendet. Daher ist ein Upgrade des Projektes in die neue Version von Nöten, damit der PWM IP Core in das bestehende System eingebunden werden kann. Hierbei sind ein paar Schritte zu beachten. Das System wird nach [16, S.115-121] in Betrieb genommen. Folgende Schritte sind für die Inbetriebnahme notwendig:

1 Öffnen des Projekts im *read-only*-Modus

2 Sichern des Projektes als Absicherung.

3 Über *Report IP Status* kontrollieren, welche Änderungen seit dem letzten Release vorgenommen wurden und nur die IPs upgraden, bei denen ein Upgrade notwendig ist. Oder, wenn man sich sicher ist, dass die Veränderungen kein Einfluss auf das bestehende System haben, auch die upgraden.

4 Design des Systems erneut validieren. Ausführen über Button *Validate Design*.

5 Output-Produkte über *Generate Block Design* neu generieren.

6 *HDL Wrapper neu erstellen*.

6.2 Umsetzung des PWM IP Cores

In diesem Abschnitt wird die Umsetzung des VHDL-Modells der einzelnen Module des PWM IP Cores erläutert. Die Umsetzung wird anhand von ausgewählten Code-Beispielen dargestellt. In Abbildung 6.2 ist das umgesetzte PWM-Modul abgebildet.

6.2.1 Umsetzung des PWM-Sync-Moduls mittels 2-FF Synchronisierer

Die Umsetzung des PWM-Sync-Moduls erfolgt nach Chu [4]. Das Modul ist als 1-Prozess-Modell umgesetzt und verfügt über einen synchronen Reset, welcher high-aktiv ist. Die Umsetzung ist in Listing 6.2.1 für das PWM-Sync-Dr-Modul abgebildet. *meta_reg* ist hierbei das Ausgangssignal des ersten D-FF und *sync_reg* der Ausgang des zweiten D-FF. *meta_next* und *sync_next* sind jeweils die Eingänge der D-FFs. Mit jeder Taktflanke werden den Ausgängen *meta_reg* und *sync_reg* die anstehenden Signale der Eingänge *meta_next* und *sync_next* zugewiesen. Dies ist in den Zeilen 24 und 25 zu erkennen. In den Zeilen 30 und 31 erfolgt die *next-state logic*. Dabei wird den Eingängen der D-FFs mit jeder Taktflanke das eingehende Eingangssignal zugewiesen. Wenn also ein PWM-Signal eingeht, wird dieses *meta_next* zugewiesen. Das ausgehende Signal vom ersten D-FF *meta_reg* wird *sync_next* zugewiesen. Und dem Output-Signal PWM_SYNC_-DR_OUT wird schließlich das Ausgangssignal vom zweiten D-FF *sync_reg* zugewiesen. Die Umsetzung für das PWM-Sync-Dir-Modul erfolgt analog zu der hier abgebildeten Umsetzung. Als Eingangssignal wird lediglich das PWMIN_DIR, statt PWMIN_DR genutzt. Und als Ausgang dementsprechend PWM_SYNC_DIR_OUT, statt PWM_-SYNC_DR_OUT.

Listing 6.1: Umsetzung des 2-FF-Synchronisierers

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity PWM_SYNC_DR_MODULE is
5      port (
6          CLK_100MHZ      :      in std_logic;
7          ARESETN         :      in std_logic;
8          PWMIN_DR        :      in std_logic;
9          PWM_SYNC_DR_OUT:       out std_logic
10         );
11 end PWM_SYNC_DR_MODULE;
12 architecture Behavioral of PWM_SYNC_DR_MODULE is
13     signal meta_reg, sync_reg: std_logic;
14     signal meta_next, sync_next: std_logic;
15 begin
16     -- Two D-FFs
17     twoDFF: process(CLK_100MHZ)
18     begin
19         if (rising_edge(CLK_100MHZ)) then
20             if ARESETN = '1' then
21                 meta_reg <= '0' after 2 ns;
22                 sync_reg <= '0' after 2ns;
23             else
24                 meta_reg <= meta_next after 2 ns;
25                 sync_reg <= sync_next after 2 ns;
26             end if;
27         end if;
28     end process;
29     -- next state logic
30     meta_next <= PWMIN_DR after 2 ns;
31     sync_next <= meta_reg after 2 ns;
32     -- output
33     PWM_SYNC_DR_OUT <= sync_reg after 2 ns;
34 end Behavioral;
```

6.2.2 Umsetzung des PWM-CTRL-Moduls

Die Umsetzung des PWM-Ctrl-Moduls erfolgt nach dem in Kapitel 5.2.4 erläuterten Konzept. Das Modul ist als 2-Prozess-Modell umgesetzt und verfügt genauso wie der 2-FF-Synchronisierer über einen synchronen Reset, der high-aktiv ist. In Abbildung 5.2.4 ist das Blockschaltbild des PWM-Ctrl-Dr-Moduls abgebildet. Die Werte für NMAX und NDUTY_DR werden durch den Master in das jeweilige AXI-Register geschrieben. Durch Setzen des Bit RUN wird das Modul gestartet. Anschließend erfolgt die Abfrage der Steuerungsart über das Bit AUTO. Ist AUTO gesetzt, wird die autonome Fahrzeugsteuerung gestartet und der Duty Cycle wird nach der Konfiguration des Masters durchlaufen. Ist AUTO nicht gesetzt, wird die manuelle Fahrzeugsteuerung gestartet und das eingehende synchronisierte PWM-Signal PWM_SYNC_DR_IN wird direkt dem Ausgangssignal PWMOU_DR zugewiesen. Der Ablauf ist genau wie in Kapitel 5.2.4 beschrieben nach dem ASMD-Chart 5.6 in VHDL umgesetzt. Analog zum PWM-Ctrl-Dr-Modul ist das PWM-Ctrl-Dir-Modul umgesetzt. Statt der Eingänge NDUTY_DR und PWM_SYNC_DR_IN werden NDUTY_DIR und PWM_SYNC_DIR_IN genutzt. Als Ausgang wird dementsprechend statt des PWMOUT_DR, der Ausgang PWMOUT_DIR genutzt.

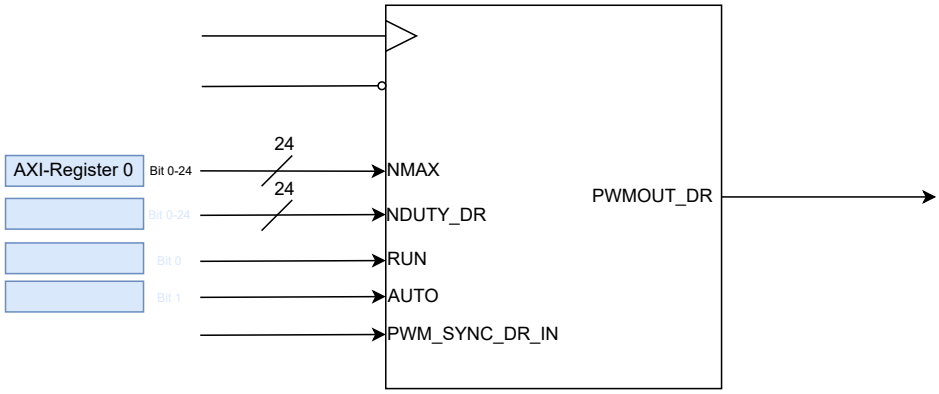


Abbildung 6.1: Blockschaltbild des PWM-Ctrl-Dr-Moduls

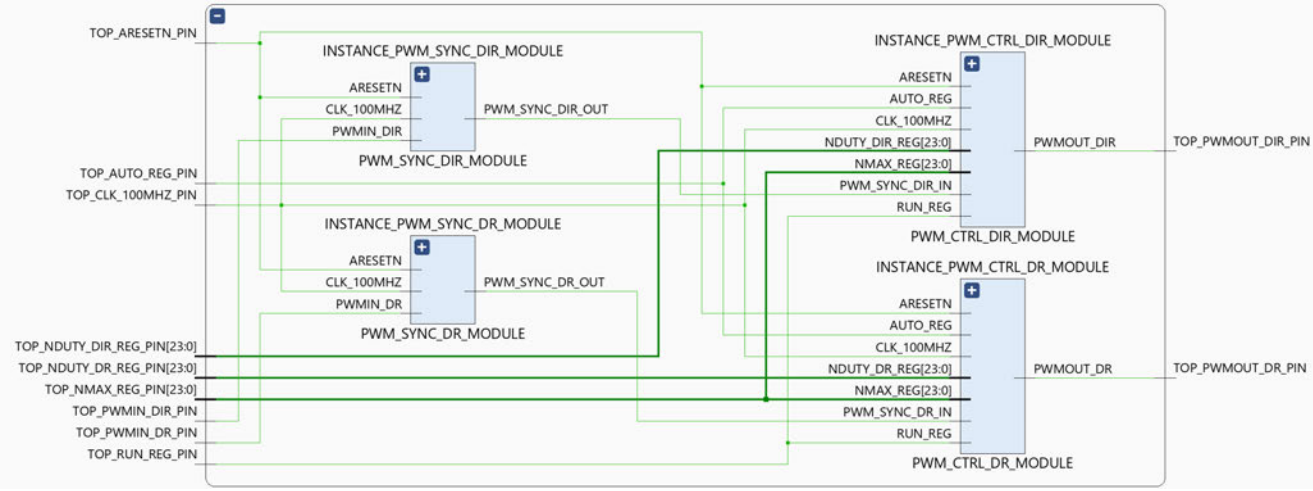


Abbildung 6.2: Umgestztes PWM-Module

6.3 Implementierung des PWM-Moduls in das Coprozessor-System

Nach der Umsetzung des PWM IP Cores, kann dieser nun in das bestehende Block Design eingebunden werden. Nach der erfolgten Implementation, kann die Synthese erfolgen. Die Synthese von VHDL-Code ist ein Prozess der Realisierung der VHDL-Beschreibung, in dem die primitiven *logic-cells* von der FPGA Bibliothek genutzt werden [4]. Bei diesem Prozess wird die VHDL-Beschreibung in eine Netzliste umgesetzt. Nach erfolgreicher Synthese, wird die Vivado Implementation gestartet. Die Implementation umfasst alle Entwurfsschritte, die erforderlich sind, um die Netzliste auf dem FPGA zu platzieren und zu Routen [15]. Nach erfolgter Implementation, kann der Bitstream erstellt werden. Ein FPGA Bitstream ist ein File, welches die Progrmmier-Informationen für einen FPGA enthält. Ein Xilinx FPGA kann nur mit einem spezifischen Bitstream programmiert werden. Nach erfolgreichem erstellen des Bitstreams kann der FPGA programmiert werden. Anschließend erfolgt in der Vitis-Umgebung, welche Eclipse-basiert ist, die Programmierung des MicroBlaze in der Programmiersprache C. Dazu wird die implementierte Hardware, welche das Bitstream inkludiert, in eine *.xsa*-Datei exportiert. Auf Grundlage dieser Datei wird ein *Platform-Project* in Vitis erstellt. In dieser Platform sind automatisch generierte C-Projekte für die Xilinx IP Cores. Mit *'Build project'* wird die erstellte Plattform debugged, was zu Fehlern bei den Custom IP Cores, also den selbst erstellten IP Cores, führt. Dieser Fehler ist in Abbildung 6.3 abgebildet. Dieser Fehler tritt bei allen drei Custom IP Cores auf. Um diesen zu beheben, muss man nach [22] die drei Zeilen

Code in Abbildung 6.4 aus dem jeweiligen Makefile durch die drei Zeilen aus Abbildung 6.5 ersetzen. Anschließend wird *'Build project'* ohne Fehler ausgeführt.

```
microblaze-xilinx-elf-gcc.exe: error: *.c: Invalid argument
make[2]: *** [Makefile:18: libs] Error 1
make[1]: *** [Makefile:46: microblaze_0/libsrc/axi_fsdm_module_v1_0/src/make.libs] Error 2
make: *** [Makefile:18: all] Error 2
Failed to build  the bsp sources for domain - standalone_domain
Failed to generate the platform.
```

Abbildung 6.3: Blockschaltbild Gesamtsystem

```
12 INCLUDEFILES=*.h
13 LIBSOURCES=*.c
14 OUTS = *.o
```

Abbildung 6.4: Blockschaltbild Gesamtsystem

```
12 INCLUDEFILES=$(wildcard *.h)
13 LIBSOURCES=$(wildcard *.c)
14 OUTS = $(wildcard *.o)
```

Abbildung 6.5: Blockschaltbild Gesamtsystem

Aus zeitlichen Gründen konnte keine Umsetzung auf Hardware stattfinden. In den nächsten Schritten muss ein Applikations-Projekt (Software-Projekt) erstellt werden und die C-Dateien aus dem bestehenden Coprozessor-System müssen in dieses geladen werden. Anschließend muss der C-Code, der das Lesen und Beschreiben der Register des PWM-Moduls ermöglicht, geschrieben werden. Danach kann das gesamte Coprozessor-System debugged werden und wenn dies erfolgreich ist, kann das System in die Hardware geladen werden. Anschließend muss der Arduino-Code, der das PWM-Modul steuern soll, geschrieben werden. Dies alles wurde in dieser Arbeit nicht umgesetzt.

In Abbildung 6.6 ist das Gesamtsystem mit dem erstellten PWM-Modul abgebildet.

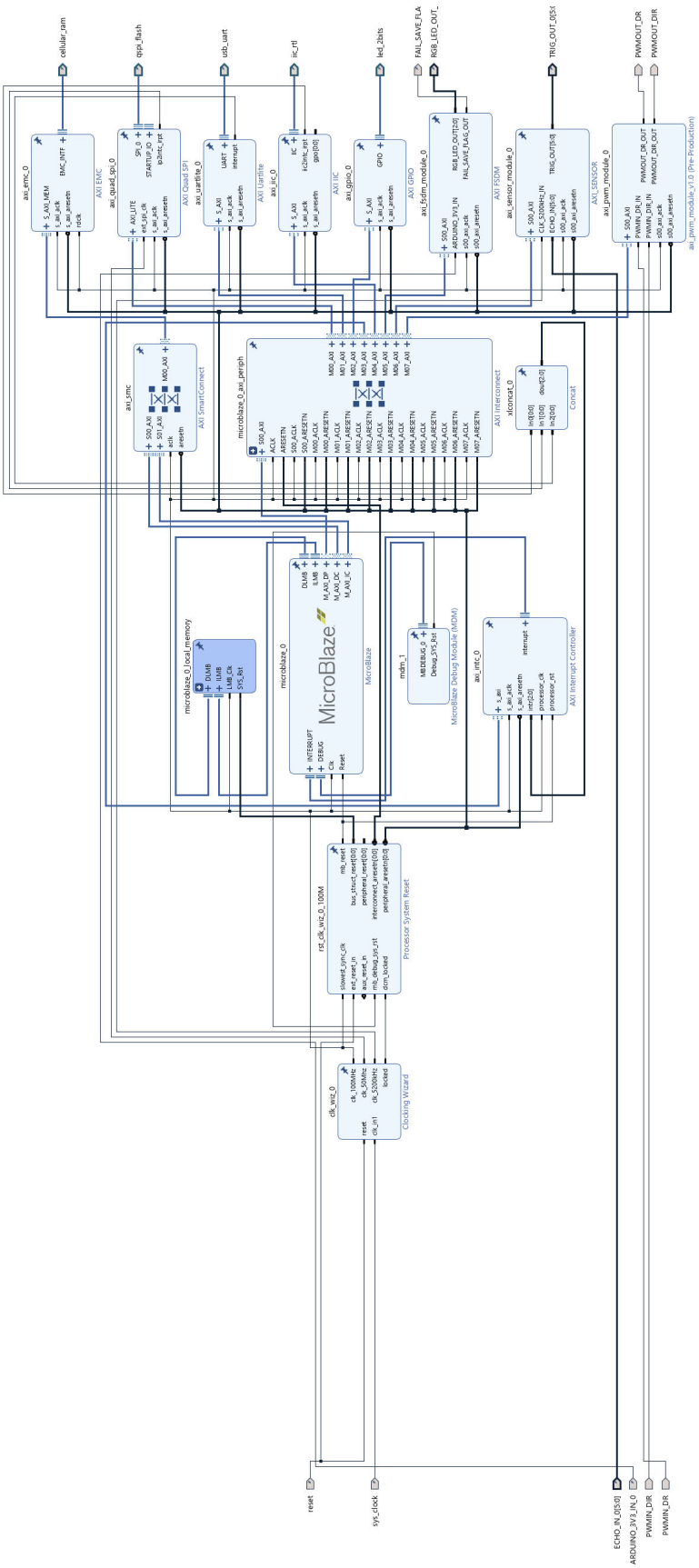


Abbildung 6.6: Blockschaltbild Gesamtsystem

7 Test

In diesem Kapitel wird das konzipierte PWM-Modul auf seine Funktion durch verschiedene Simulationen in der Vivado Simulationsumgebung verifiziert. Es wird die autonome Fahrzeugsteuerung simuliert, die manuelle Fahrzeugsteuerung und das Umschaltverhalten bei einem Wechsel der Steuerungsart.

7.1 Simulation der autonomen Fahrzeugsteuerung

In diesem Abschnitt wird die autonome Fahrzeugsteuerung simuliert. Die Simulationen werden für das PWM-Ctrl-Dr-Modul durchgeführt.Da das PWM-Ctrl-Dir-Modul identisch aufgebaut ist, gelten die Simulationsergebnisse für dieses genauso. Die Simulation für das autonome Fahren werden für $\tau = 1\ ms$, $\tau = 1,5\ ms$ und $\tau = 2\ ms$ durchgeführt. Es werden also die obere und untere Grenze, sowie die neutrale Stellung getestet. Eine genaue Analyse erfolgt für die untere Grenze des PWM-Signals mit $\tau = 1\ ms$. Für den Duty Cycle sind dementsprechend $NMAX = 1666667$ und $NDUTY\_DR = 99980$ einzustellen. Da in eine Testbench keine Dezimalwerte zugewiesen werden können, müssen diese Werte entsprechend in Binär-Werte umgerechnet werden:

$$NMAX = \text{"}000110010110111001101011\text{"}$$
$$NDUTY\_DR = \text{"}000000011000011010001100\text{"}$$

Die vorne weg stehenden Nullen ergeben sich daraus, da es sich bei $NMAX$ und $NDUTY\_DR$ um 24-Bit Vektoren handelt. Daher müssen die nicht benötigten Bits mit Nullen gefüllt werden. In Abbildung 7.1 ist das Ergebnis der Simulation für $NDUTY\_DR = 99980$ zu sehen. In der linken Spalte sieht man die Signale, die in der Simulation abgebildet werden. Man erkennt, dass RUN und $AUTO$ gesetzt sind und dadurch der Duty Cycle gestartet wird. Mit setzen dieser Bits, werden die Werte von $NMAX$ und $NDUTY\_DR$ geladen und der Duty Cycle startet. Wie in der Abbildung zu erkennen ist, steht das PWM-Signal für ca. 1 ms an und die Periodendauer beträgt

ca 16, 67 ms. Das zeigt, dass der Duty Cycle funktioniert.

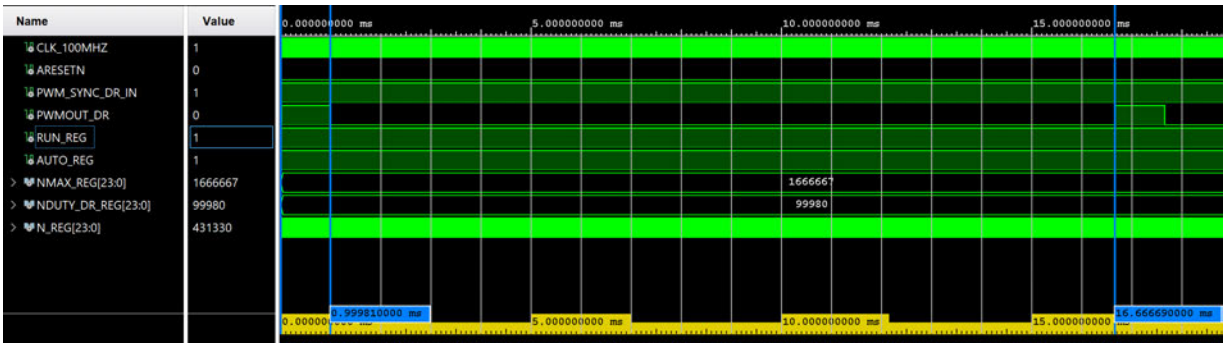


Abbildung 7.1: Simulation des Duty Cycles der autonomen Steuerung

In Abbildung 7.2 ist die Bedingung für das Rücksetzen des PWM-Signals zu erkennen. Sobald in den internen Zähler N_REG der Wert 99979 geschrieben wird, ist die Bedingung $N\_REG < NDUTY\_DR - 1$ nicht mehr erfüllt und $PWMOUT\_DR$ wird auf 0 gesetzt.

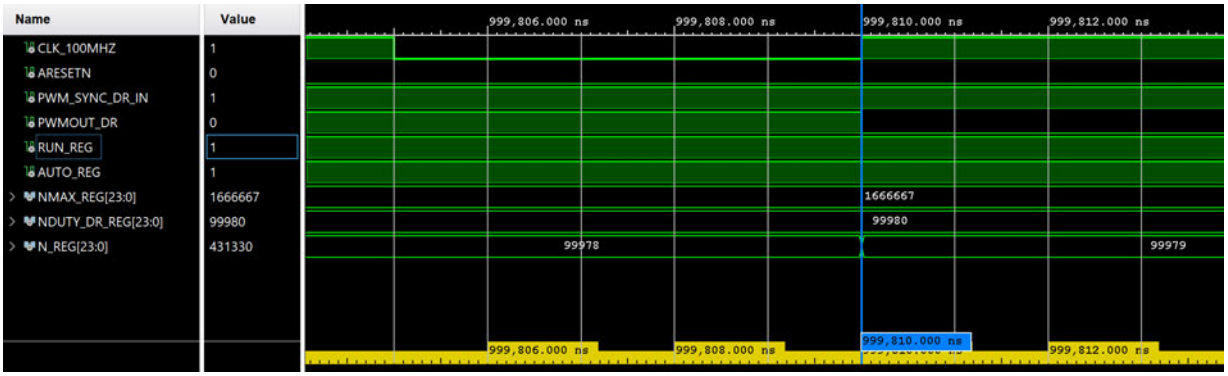


Abbildung 7.2: Bedingung für das Zurücksetzen von PWMOUT

In Abbildung 7.3 ist die Abbruchbedingung für den Duty Cycle abgebildet. Der Duty Cycle wird verlassen, sobald $N\_REG < NMAX - 1$ nicht mehr erfüllt ist. Dies ist in der Abbildung gut zu erkennen. Sobald der Duty Cycle verlassen wird und $AUTO$ weiterhin gesetzt ist, wird der Zustand PWMCTRL_INIT betreten und $PWMOUT\_DR$ wird wieder gesetzt, genauso wie N_REG mit der nächsten Taktflanke wieder auf 0 gesetzt wird.

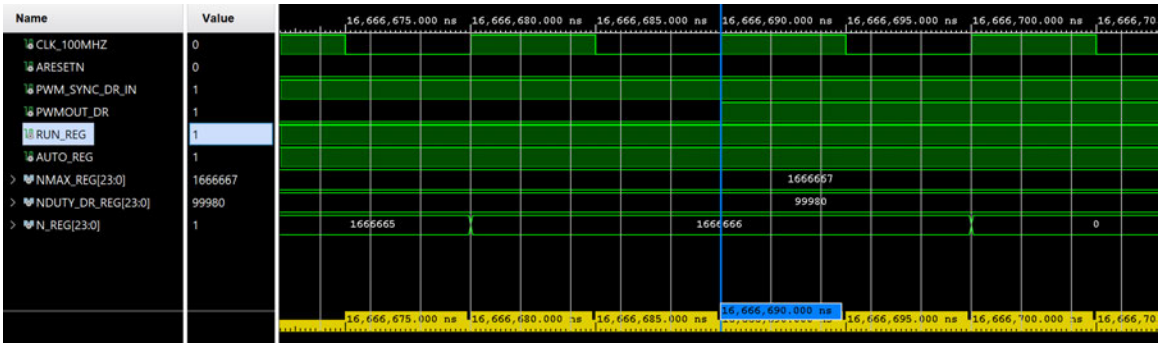


Abbildung 7.3: Abbruchbedingung des Duty Cycles

in den Abbildungen 7.4, 7.5 und 7.6 sind die Simulationsergebnisse für das autonome Fahren über eine Simulationszeit von ca. 100 *ms* abgebildet.

7.1.1 NMAX=1666667, NDUTY=99980

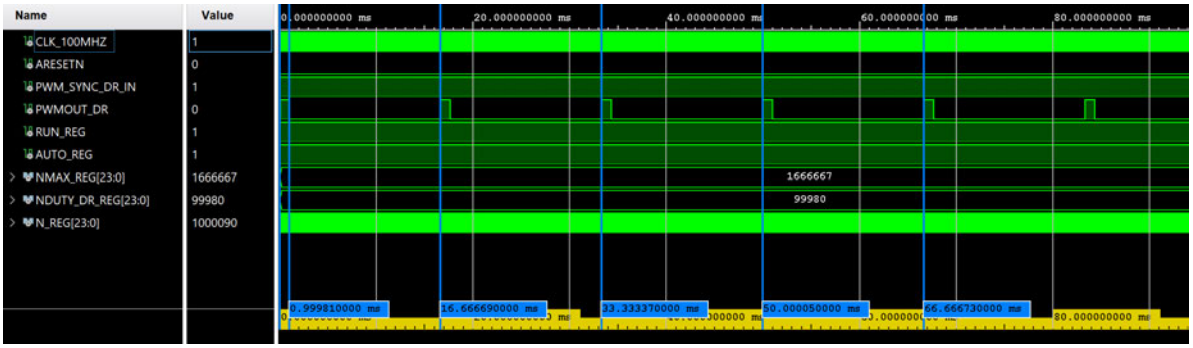


Abbildung 7.4: Simulation der unteren Grenze des PWM Duty Cycles

7.1.2 NMAX=1666667, NDUTY=149970

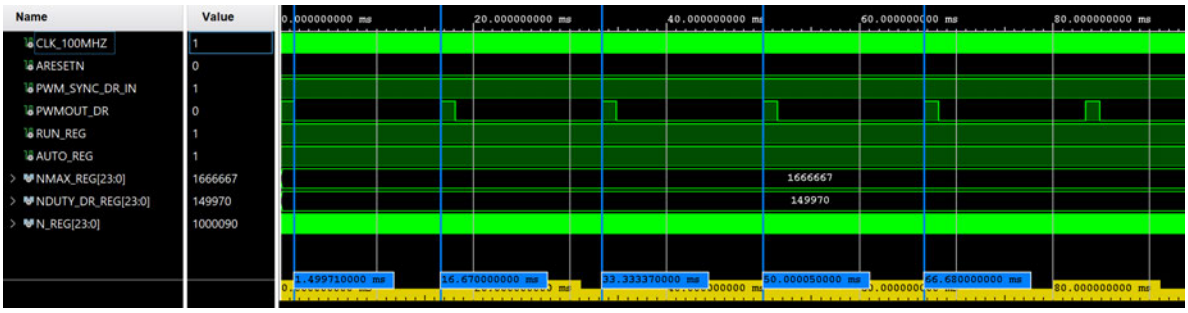


Abbildung 7.5: Simulation der neutralen Position des PWM Duty Cycles

7.1.3 NMAX=1666667, NDUTY=199960

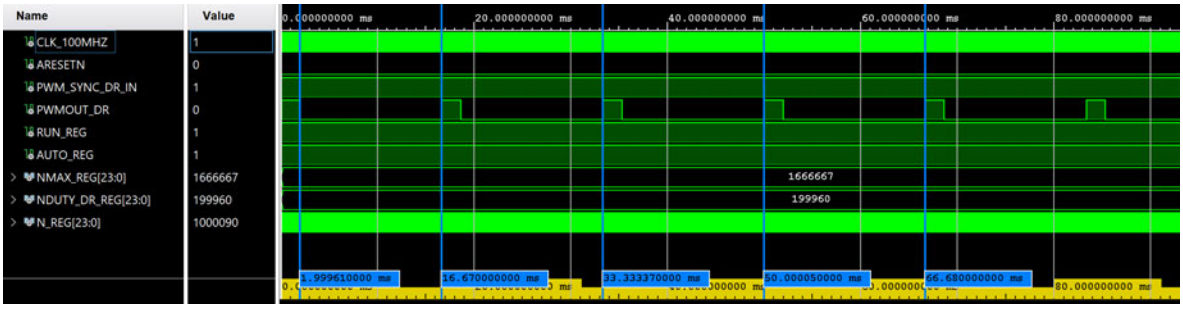


Abbildung 7.6: Simulation der oberen Grenze des PWM Duty Cycles

7.2 Simulation der manuellen Fahrzeugsteuerung

In diesem Abschnitt erfolgt die Simulation des manuellen Fahrens. Um das manuelle Fahren zu simulieren wird eine *for-Schleife* in der Testbench durchlaufen. Dabei ist die Schleife so konzipiert worden, dass das $PWM\_SYNC\_DR\_IN$ genau $1,5\ ms$ anliegt und die Periodenzeit $16,67\ ms$ beträgt. So kann die Simulation realitätsnah erfolgen. Abbildung 7.7 zeigt das Ergebnis. Es ist zu erkennen, dass RUN gesetzt ist und $AUTO$ nicht. Dadurch wird die manuelle Steuerung gestartet und $PWMOUT\_DR$ ist immer genau dann 1, wenn das Eingangssignal $PWM\_SYNC\_DR\_IN$ ansteht. Das Simulationsergebnis zeigt, dass die manuelle Fahrzeugsteuerung sehr gut funktioniert.

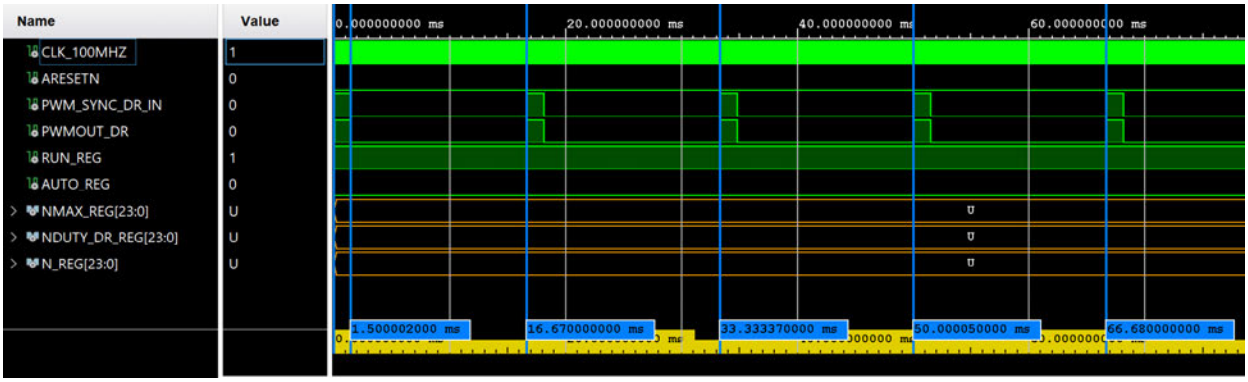


Abbildung 7.7: Simulation der manuellen Steuerung

7.3 Simulation des Umschaltverhaltens

In diesem Abschnitt wird das Umschaltverhalten des Moduls simuliert. Es erfolgt einmal die Simulation des Umschaltens der manuellen zur autonomen Steuerung und umgekehrt.

7.3.1 Umschalten von manueller zu autonomer Fahrzeugsteuerung

Abbildung 7.8 zeigt das Umschalten der manuellen Steuerung zur autonomen Steuerung. Zunächst werden drei Perioden der manuellen Steuerung durchlaufen, bis dann $AUTO = 1$ gesetzt wird. Anschließend wird der Zustand $PWMCTRL\_INIT$ betreten und wie zu erkennen ist, werden die Register mit den vorgegebenen Werten initialisiert. Anschließend wird der Duty Cycle durchlaufen. Das Ergebnis zeigt, dass das Umschalten von der manuellen zur autonomen Steuerung sehr gut funktioniert.

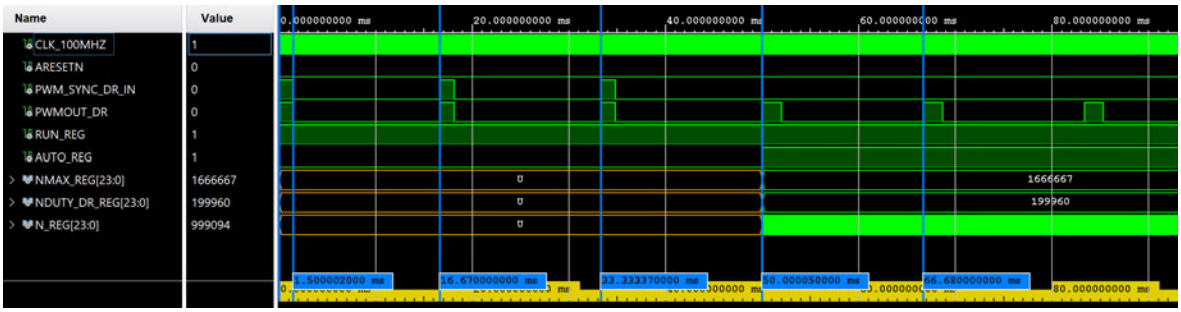


Abbildung 7.8: Simulation des Umschaltens von manueller zu autonomer Steuerung

7.3.2 Umschalten von autonomer zu manueller Fahrzeugsteuerung

Abbildung 7.9 zeigt das Umschalten der autonomen Steuerung zur manuellen Steuerung. Zunächst wird ganz normal der Duty Cycle durchlaufen. Nach 40 ms wird dann $AUTO = 0$ gesetzt. Gleichzeitig steht ein $PWM\_SYNC\_DR\_IN$-Signal an. Dennoch wird $PWMOUT\_DR$ nicht gesetzt, da der Duty Cycle so konzipiert ist, dass dieser erst nach einem kompletten Durchlauf verlassen werden kann. Das ist daran zu erkennen, dass der Zähler N_REG noch hoch zählt. Erst nach ca. 50 ms ist der Duty Cycle durchlaufen un die Bedingung $N\_REG < NMAX - 1$ nicht mehr erfüllt. Und da die Abfrage $AUTO = 1$ ebenfalls nicht mehr erfüllt ist, wird der Zustand $PWMFWD$ betreten und die manuelle Fahrzeugsteuerung ist möglich. Das ist daran zu erkennen, da $PWMOUT\_DR$ immer nur dann gesetzt wird, wenn $PWM\_SYNC\_DR\_IN = 1$ ist. Außerdem wird N_REG nicht wieder auf 0 gesetzt, da dies nur im Zustand

$PWMCTRL\_INIT$ geschieht. Das zeigt, dass das Umschalten zur manuellen Fahrzeugsteuerung sehr gut funktioniert.

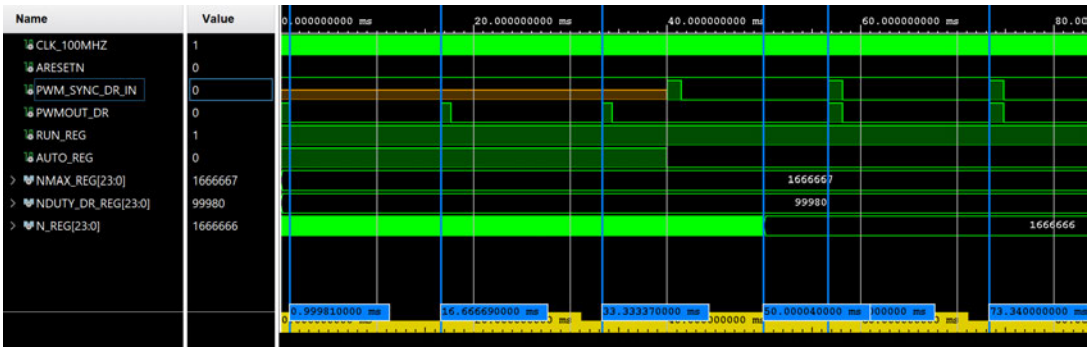


Abbildung 7.9: Simulation des Umschaltens von autonomer zu manueller Steuerung

7.4 Simulation des Abbruchverhaltens

In diesem Abschnitt wird das Abbruch-Verhalten simuliert. Das heißt, das Modul wird auf sein Verhalten bei einem System-Reset getestet. Außerdem wird das Verhalten getestet, wenn RUN während des Programmablaufs 0 gesetzt wird. In Abbildung 7.10 ist dieses Verhalten simuliert dargestellt. Die Simulation erfolgte für das autonome Fahren, was daran zu erkennen ist, das RUN und $AUTO$ gesetzt sind. Nach 35 ms wird RUN dann gleich 0 gesetzt. Zunächst läuft der Duty Cycle ganz normal weiter, da der Duty Cycle nur dann verlassen wird, wenn der Duty Cycle eine komplette Periode durchlaufen ist und $N\_REG < NMAX - 1$ nicht mehr erfüllt ist. Daher erfolgt erst nach ca. 50 ms der Wechsel in den Zustand $PWMCTRL\_INIT$, was dafür sorgt das $PWMOUT\_DR$ mit dem nächsten Taksignal gesetzt wird und mit dem darauffolgenden Taktsignal wieder zurückgesetzt wird. Das bedeutet, dass $PWMOUT\_DR$ für 20 ns gesetzt wird, was eigentlich nicht sein sollte. Da erst nach der Initialisierung die Abfrage $RUN = 1$ erfolgt, wird aber $PWMOUT\_DR$ für eine sehr kurze Zeit gesetzt. Da Abfrage $RUN = 1$ nun nicht mehr wahr ist, wird der Zustand $IDLE$ betreten. Erst wenn RUN wieder gesetzt ist, wird der Duty Cycle wieder durchlaufen, was nach 55 ms der Fall ist. Nach 70 ms dann erfolgt ein Reset, was zum sofortigen Zurücksetzen Register führt. Diese Simulation hat eine Schwäche des umgesetzten Modul gezeigt. Es sollte eine zusätzliche Abfrage von RUN nach der Abbruchbedingung des Duty Cycles erfolgen. Dadurch wird vermieden, dass erst der Zustand $PWMCTRL\_INIT$ betreten wird und $PWMOUT\_DR$ für eine sehr kurze Zeit gesetzt wird. Dennoch zeigt die Simulation, dass die Abbruchbedingungen sehr gut funktionieren.

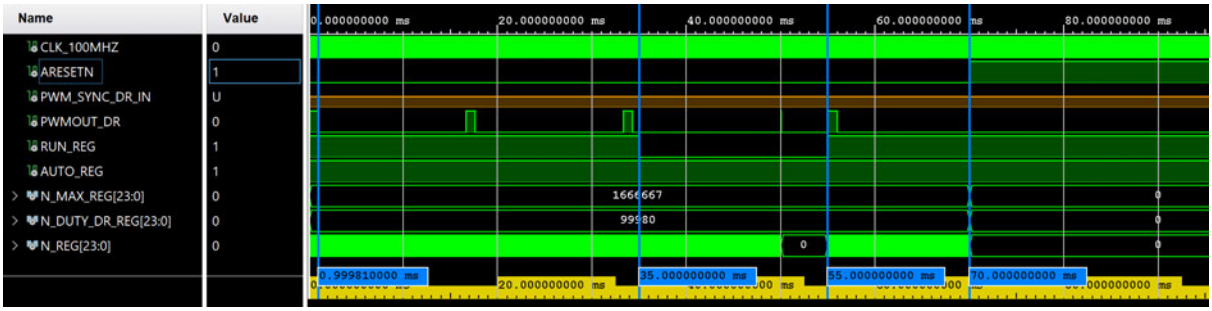


Abbildung 7.10: Simulation Abbruchverhalten

7.5 Auswertung bezüglich der Anforderungen

Da die Implementation des Systems aus zeitlichen Gründen nicht mehr möglich war, konnte kein Test auf Hardware durchgeführt werden. Daher konnte keine der gestellten Anforderungen erfüllt werden. Die Simulationen zeigen allerdings, dass sowohl die manuelle Steuerung, als auch die autonome Steuerung theoretisch funktionieren. Außerdem ist es möglich, die Steuerungsart einzustellen und während des Betriebs zu ändern. Es ist zudem möglich, das PWM-Modul über die Slave-Register zu konfigurieren. Das Modul kann auch separat über *RUN* ein- und ausgeschaltet werden.

8 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde ein AXI IP Core PWM-Modul zur Motorenansteuerung mittels PWM-Signalen umgesetzt. Dieser IP Core ermöglicht es, ein RC-Fahrzeug manuell und autonom zu fahren. Das PWM-Modul ist die Erweiterung eines bestehenden Systems, dass die Multi-Sensorsignalverarbeitung ermöglicht. Im ersten Schritt wurde das bestehende Coprozessor-System analysiert und daraus die Anforderungen für das entwickelte PWM-Modul abgeleitet. Anschließend erfolgte die Entwicklung eines Konzepts und das daraus resultierende Design wurde in VHDL umgesetzt. Die Umsetzung des VHDL-Codes auf Hardware und somit ein Test des gesamten Systems mit der Multi-Sensor-Signalverarbeitung war aus zeitlichen Gründen nicht möglich. Somit wurde die Funktionalität des PWM-Moduls durch Simulationen verifiziert. Die Simulationsergebnisse haben gezeigt, dass das PWM-Modul das autonome Fahren und das manuelle Fahren ermöglicht. Beim autonomen Fahren ist es zudem möglich den Duty Cycle durch entsprechende Konfiguration der Register vorzugeben. Ebenfalls ist ein Umschalten zwischen den Steuerungsarten möglich.

Im nächsten Schritt sollte das Gesamtsystem auf Hardware umgesetzt und der MicroBlaze programmiert werden. Anschließend muss der Arduino-Code für die Ansteuerung des PWM-Moduls entwickelt werden. Wenn das Coprozessor-System dann einwandfrei läuft, kann dieses beliebig erweitert werden. So kann das System durch weitere Sensoren und damit weitere IP Cores erweitert werden oder man verbessert zunächst das bestehende System. Da sollte der asynchrone Takt von $5,2\ MHz$ des Sensor-Moduls möglichst durch den Systemtakt ersetzt werden und dieser das Clock-Enable über einen Frequenzteiler erzeugen. Das PWM-Modul kann ebenfalls noch durch einige Features erweitert werden. So sollte auf jeden Fall das Fail-Save-Flag vom FSDM-Modul an das PWM-Modul gekoppelt werden, sodass dieses im Falle ienes Fehlers gestoppt wird. Eine weitere Möglichkeit zur Erweiterung wäre eine Ausfallüberwachung der Ultraschallsensoren.

Literaturverzeichnis

[1] Arduino: *Arduino Nano.* – URL https://docs.arduino.cc/hardware/nano

[2] ARM: *AMBA AXI and ACE Protocol Specification.* 2020

[3] Az Delivery: *TXS0108E.* – URL https://cdn.shopify.com/s/files/1/1509/1638/files/TXS0108_Logiklevel_Converter_Datenblatt_AZ-Delivery_Vertriebs_GmbH.pdf?v=1608239085

[4] Chu, Pong P.: *RTL Hardware Design using VHDL.* John Wiley and Sons, 2006

[5] Conrad, Electronic: *Voltage Step Down Module.* 2019. – URL https://asset.conrad.com/media10/add/160267/c1/-/gl/002134134ML00/bedienungsanleitung-2134134-makerfactory-spannungsregler-mf-6402402-1-st.pdf

[6] Digilent, Inc: *Cmod A7: Breadboardable Artix-7 FPGA Module.* – URL https://digilent.com/shop/cmod-a7-breadboardable-artix-7-fpga-module/

[7] Digilent, Inc: *Cmod A7 Reference Manual.* Oktober 2019. – URL https://digilent.com/reference/_media/reference/programmable-logic/cmod-a7/cmod_a7_rm.pdf

[8] Fairchild, Semiconductor: *BSS138.* Oktober 2005. – URL http://cdn.sparkfun.com/datasheets/BreakoutBoards/BSS138.pdf

[9] Gries, Kolja: *Konzept und Implementierung eines System-on-Chip auf einem FPGA für die Multi-Sensor-Signalverarbeitung in autonomen Fahrzeugen*, HAW Hamburg, Bachelor Thesis, 2020

[10] Hensel, Marc: *RC-Projekte Arduino.* 2018

[11] Leutelt, Lutz: *Multiple Clock Systems.* 2021

[12] Microchip: *megaAVR Data Sheet*. 2018. – URL http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf

[13] Tamboli, Jakirhusen I. ; Jagtap, Satyawan R. ; Sutar, Amol R.: Pulse width modulation implementation using FPGA and CPLD ICs. In: *International Journal of Scientific & Engineering Research* (2012)

[14] TI: *PCA9306 Dual Bidirectional I2C Bus and SMBus Voltage-Level Translator*. Oktober 2021. – URL https://www.ti.com/lit/ds/symlink/pca9306.pdf

[15] Xilinx, Inc: *Implementation*. August 2012. – URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug904-vivado-implementation.pdf#:~:text=Vivadoimplementationencompassesallofthedesignsteps,logical,physical,andtimingconstraintsofthedesign.

[16] Xilinx, Inc: *Vivado Design Suite User Guide*. April 2015. – URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug994-vivado-ip-subsystems.pdf

[17] Xilinx, Inc: *Local Memory Bus (LMB) v3.0*. April 2016. – URL https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/lmb_v10/v3_0/pg113-lmb-v10.pdf

[18] Xilinx, Inc: *AXI Interconnect v2.1*. Dezember 2017. – URL https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[19] Xilinx, Inc: *AXI Reference Guide*. Juli 2017. – URL https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[20] Xilinx, Inc: *AXI Block RAM (BRAM) Controller v4,1*. Mai 2019. – URL https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_1/pg078-axi-bram-ctrl.pdf

[21] Xilinx, Inc: *MicroBlaze Processor Reference Guide*. Juni 2020. – URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug984-vivado-microblaze-ref.pdf

[22] Xilinx, Inc: *Xilinx Forum*. September 2021. – URL https://support.xilinx.com/s/question/0D52E00006hpYgWSAU/vitis-ide-20211-custom-axi-ip-core-compile-error?language=en_US

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original