Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Heiko Bordasch

VHDL-Modellierung einer Geschwindigkeitsregelung für ein autonomes Fahrzeug implementiert auf einer SoC - Plattform

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heiko Bordasch

VHDL-Modellierung einer Geschwindigkeitsregelung für ein autonomes Fahrzeug implementiert auf einer SoC - Plattform

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
Department Informatik
in der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Bernd Schwarz
Zweitgutachter : Prof. Dr.-Ing. Franz Korf

Abgegeben am 2. Juli 2009

Heiko Bordasch

Thema der Bachelorarbeit

VHDL-Modellierung einer Geschwindigkeitsregelung für ein autonomes Fahrzeug implementiert auf einer SoC-Plattform

Stichworte

VHDL, FPGA, FAUST, MicroBlaze,System on Chip, autonomes Fahrzeug, Geschwindigkeitsregelung, Hard Realtime System, PI-Regler, PT1-Glied, Quadratische Regelfläche

Kurzzusammenfassung

Diese Arbeit behandelt die VHDL-Modellierung einer FPGA basierten Geschwindigkeitsregelung für ein autonomes Fahrzeug. Bestehend aus einem Pulszähler, einem PI-Regler mit integriertem Abtastratentimer und einem PWM Modul, wird der V-Regler IP Core in ein MicroBlaze Bussystem integriert und auf einer SoC Plattform implementiert. Der PI-Regler wird in einer dreistufigen Pipeline implementiert und ist für die globale Zeitgebung des Systems zuständig. Ein Datenlogger speichert die Messdaten im SRAM und überträgt diese anschließend über die serielle Schnittstelle an den PC.

Heiko Bordasch

Title of the paper

VHDL-Modelling of a speed control unit for an autonomous vehicle implemented on a SoC platform

Keywords

VHDL, FPGA, FAUST, MicroBlaze,System on Chip, autonomous vehicle, speed control system, System-on-chip, hard real time system, control unit, quadratic control area

Abstract

This thesis describes the modelling of an FPGA based speed control unit for an autonomous vehicle. Containing a pulsecounter, a PI-Controller with an integrated periodtimer and a PWM modul, the IP Core is integrated into a microblaze bussystem and implemented on a system-on-chip platform. The PI-Controller is implemented in a three-stage pipeline and is responsible for the global time setting of the system. A datalogger saves the measured data in the SRAM and transfers them over the serial interface to the Pc.

Inhaltsverzeichnis

1. Einleitung

Zu Beginn des 20. Jahrhunderts war das Fahren mit dem PKW geprägt von zahlreichen Fahrzeugbedienungstätigkeiten (z.B. manuelle Scheibenwischerbetätigung oder manuelle Entriegelung). Parallel zur zunehmenden Automatisierung der Fahrtätigkeit wuchs die Verkehrsdichte und damit die Anforderung an die eigentliche Fahrzeugführung. Der Fahrer war zunehmend damit beschäftigt, das Fahrzeug sicher durch den Verkehr zu leiten. Forscher und Entwickler waren stetig darauf bedacht, dem Fahrer diese Aufgaben zu erleichtern. In den 70er Jahren wurden die ersten Fahrerassistenzsysteme (FAS), wie zum Beispiel das Antiblockiersystems (ABS) und das Elektonisches Stabilitätsprogramm (ESP), im Autombil eingesetzt. Beide Systeme sollen den Fahrer bei kritischen Situation unterstützen und bieten ein enormes Potenzial zur Verbesserung der aktiven Verkehrsicherheit, indem sie dem Fahrzeug Spursicherheit verleihen. Mit der Einführung von Abstandsregeltempomaten (Adpative Cruise Control ACC) zur Jahrtausendwende brach eine neue Ära der Manöver unterstützenden Fahrerassistenzsysteme an [10]. ACC ist ein modernes FAS und dient im ersten Funktionsteil, wie ein herkömmlicher Tempomat, dazu, eine vom Fahrer eingestellte Geschwindigkeit automatisch zu halten. Hinzukommt, dass das System Hindernisse im Fahrweg erkennt und durch gezieltes Abbremsen bzw. Beschleunigen dementsprechend reagieren kann [20].

In Zukunft werden weitere innovative Fahrerassistenzssysteme zum Einsatz kommen, die dem Fahrzeugführer elementare Aufgaben, wie das automatische Ausweichen in Gefahrensituationen, abnehmen sollen. Hierzu ist es von großer Bedeutung, dass das System einen guten Kenntnissstand seiner äußeren Umstände hat und in sehr kurzer Zeit geeignete Szenarien berechnen kann, um rechtzeitig reagieren zu können [22]. Das Fahrzeug muss, ähnlich wie ein Mensch mit zwei Augen, eine räumliche Vorstellungskraft besitzen und seine Umgebung detailiert und genau analysieren können. Für die Entwicklung von FAS sind nicht nur eine sehr präzisse Sensorik, sondern auch Embedded Systeme, die mit sogenannten „Hard Real Time“ Anforderungen Berechnungen richtig und vor allem rechtzeitig liefern, erforderlich. Ein weiterer Aspekt für die zukünftige Entwicklung von autonomen Systemen ist der Mangel an Platz. Aus diesem Grund können programmierbare elektronische Subsysteme eingesetzt werden, die oft nur an eine spezielle Aufgabe angepasst sind und aus Hard- und Softwarekomponenten bestehen. Solche „System on Chip Plattformen“ (SoC) werden auf FPGAs implementiert und können herkömmliche Mikrocontrollerboards, die nicht rekonfigurierbar sind, vermeiden. In der Autombilbranche werden aufrgund der großen Stückzahlen und der niedrigen Kosten anwendungspezifische integrierte Schaltungen, sogenannte ASICs eingesetzt. In dieser Arbeit wird die Aufgabenstellung aus der Automobiltechnik zur Erprobung von FPGA basierten Lösungen eingesetzt und ein Tempomat für ein autonomes Fahrzeug entworfen. Die Regelung von Systemeigenschaften wie Geschwindigkeit, Abstand oder Lenkwinkel hat bei der Entwicklung von autonomen Fahrzeugen eine große Bedeutung.

1.1. Autonome Fahrzeuge

Autonome Fahrzeuge stellen hochkomplexe, dynamische Systeme dar, die ihre Umgebung, ohne Eingriff des Menschen, analysieren und dementsprechend reagieren. Mit Hilfe von Kameras und Sensoren kann das Fahrzeug Objekte erkennen und Systemdaten erfassen (Drehzahl, Temperatur oder Abstand). Mit Einsatz der Regelungstechnik kann das autonome Fahrzeug diese Systemeigenschaften auf einen vorgegebenen Sollwert regeln und somit das gewünschte Verhalten erfüllen. Die Entwicklung von autonomen Systemen hat in den letzten Jahren deutliche Fortschritte gemacht. Sie werden in zahlreichen Bereichen der Technik eingesetzt. Ob als Transportsystem in der Industrie, als Marssonde im Weltall oder als Unterwasserfahrzeug, überall erfüllen sie schwierige, vom Menschen nur schwer erfüllbare Aufgaben.

- **DARPA URBAN Challenge**
  Die DARPA[1] Urban Challenge ist ein Wettbewerb für autonome Roboterfahrzeuge, bei dem die Fahrzeuge einen 60 Meilen langen bebauten Parcour innerhalb von sechs Stunden bewältigen müssen. Aufgeteilt in drei Missionen müssen die Fahrzeuge die kalifornische Straßenverkehrsordnung erfüllen [2]. Das Team Tartan Racing von der Carnegie Mellon University aus Pittsburgh gewann 2007 das Rennen mit Ihrem Chevy Tahoe „Boss“. Ausgestattet mit 10 Intel Core2Duo Blades mit jeweils 2,16 GHz ist es dem Fahrzeug möglich, sich vollkommen eigenständig im Verkehr zu bewegen. Zur Abstands- und Geschwindigkeitsmessung werden verschiedene, je nach Entfernung benötigte, Radar und Laser Sensoren eingesetzt [16].
- **Autonomous Underwater Vehicle AUV**
  Unbemannte Unterwasserfahrzeuge übernehmen in der Meeresforschung zunehmend Messaufgaben, die früher nur mit Einsatz von Forschungsschiffen erledigt werden konnten. Der „Explorer“ der Firma International Submarine Engineering ISE Ltd. kann mit einer Batterie eine Reichweite von 120 km bei 1,5 m/s zurücklegen. Ausgestattet mit einer sehr präzisen iXSea PHINS 6000 Inertial Navigations Einheit, kann er bis in Tiefen von 5000m vordringen und dort Messungen vornehmen [8].

Abbildung 1.1.: Tartan Racing [16]

Abbildung 1.2.: AUV Explorer [8]

[1] DARPA: Defense Advanced Research Projects Agency

1.2. Carolo Cup

Der Carolo Cup ist ein von der TU Braunschweig ins Leben gerufener Wettbewerb für autonome Modellfahrzeuge im Maßstab von 1:10. Die Fahrzeuge müssen drei verschiedene Szenarien bewältigen, dazu gehören das Fahren auf einer Rundstrecke mit und ohne Hindernisse und das parallele Einparken [1]. Die Hochschule für Angewandte Wissenschaften Hamburg (HAW) nimmt seit 2008 mit einem Leifahrzeug (nebula) der TU Braunschweig an dem Wettbewerb teil und konnte dort schon erste Erfolge feiern. Seit 2009 tritt das Team FAUST[2] der HAW mit einem selbstentworfenen Fahrzeug (onyx) an. Es basiert auf einem Ford F-350 Pickup Modell mit 3 Gang Getriebe. Für die Sensordatenerfassung sind drei ARM 7 Prozessoren zuständig. Ein Acer Aspire One Subnotebook mit einem Intel Atom 1,6 GHz Prozessor ist über USB mit der Kamera und den ARM Prozessoren verbunden. Ausgestattet mit zwei Inkremetalgebern an den Vorderrädern, jeweils vier Ultraschall- und Infarotsensoren vorne und hinten, einer Kamera zur Hindernisserkennung und einem Kompass zur Richtungsbestimmung können die Fahreigenschaften exakt bestimmt werden [6].

Abbildung 1.3.: Carolo Cup - Onyx [6]

Fahrzeugparameter	
Länge	466 mm
Breite	220 mm
Gewicht	8 kg
Radumfang	261 mm
Geschwindigkeit 1.Gang	1,45 m/s
Geschwindigkeit 2.Gang	2,63 m/s
Geschwindigkeit 3.Gang	3,80 m/s

Tabelle 1.1.: Fahrzeugdaten FAUST Onyx

[2]FAUST: Fahrerassistenz- und Autonome Systeme

1.3. Ziele der Arbeit

Im Rahmen dieser Bachelorarbeit wurde für das Carolo Cup Fahrzeug „Onyx“ eine Geschwindigkeitsregelung (vgl. Abb. 1.4) auf einem FPGA[3] entwickelt. Die Arbeit soll zum Aufbau einer SoC-basierten Fahrzeugsteuerungsplattform beitragen, die sukzessive zu einem Bahnführungssystem für Modellfahrzeuge weiterentwickelt wird. Zum ersten Einsatz kommt der V-Regler IP Core in einem automatischen Einparkassistenten (vgl. Abb. 2.1). Mit dem Einsatz von FPGA basierter Hardware können im Gegensatz zu herkömmlichen Prozessorsysteme, die nur wenige Operanden parallel verarbeiten können, alle Berechnungen völlig parallel stattfinden. Spezifische Aufgaben können mit einem FPGA in gleicher Zeit, aber mit wesentlich geringeren Taktraten erledigt werden. Die geforderten Hard-Realtime Bedingungen sind damit erfüllt. Die jetzigen sich parallel im Einsatz befindenen Mikrocontroller Boards sollen durch solche FPGA basierte Systeme ersetzt werden. Die Entwicklungsziele der Arbeit sind:

- Die messtechnische Analyse des Regelkreises mit anschließender Entwicklung und Implementierung eines V-Regler IP-Core in einem MicroBlaze Bussystem
- Die Geschwindigkeitsbestimmung mit einem Inkrementalgeber (vgl. Abb. 6.2)
- Die Ansteuerung des Motors über ein PWM Modul (vgl. Abb. 6.13)
- Eine Datenloggerfunktion zum Aufzeichnen von Messdaten und die vollständige Simulation des Regelsystems.

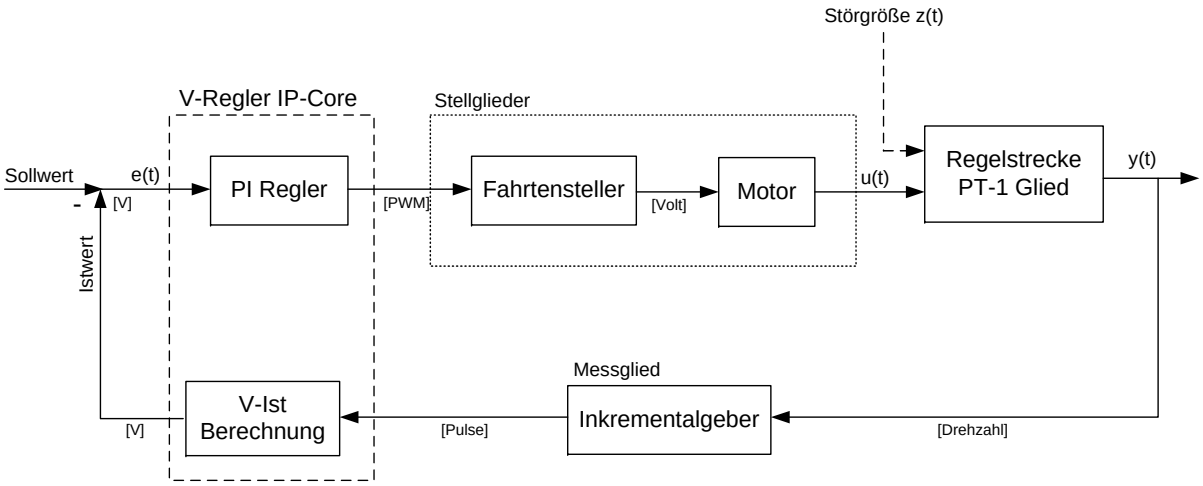

Abbildung 1.4.: Geschwindigkeitsregelkreis

Ein Pulszähler Modul berechnet für den PI Regler aus den Inkrementalgeberpulse die momentane Ist Geschwindigkeit. Der Geschwindigkeitsregler erzeugt periodisch für den Antrieb des Fahrzeuges die Stellgröße $u(t)$, die dann über einen PWM-gesteuerten Fahrtensteller und einen Bürstenmotor auf die Regelstrecke einwirkt. Durch die Rückführung der Ausgangsgröße $y(t)$ über das Messglied entsteht ein geschlossenes Regelsystem.

[3] Field Programmable Gate Array

Kapitelübersicht

Kapitel 1 dient als Einleitung. Es gibt einen Überblick über den Einsatz von Fahrerassistenzsystemen und soll einen Einstieg in das Thema der autonomen Fahrzeuge ermöglichen. Außerdem werden die Randbedingungen und die Ziele der Arbeit erläutert.

Kapitel 2 gibt eine Systemübersicht über die SoC basierte Fahrzeugsteuerungsplattform und das Geschwindigkeitsregelsystem. Die implementierten Module werden mit ihren jeweiligen Funktionalitäten vorgestellt.

Kapitel 3 stellt die verwendeten Technologien vor. Eine Einführung in „System on Chip“ Systeme soll einen Überblick über das Thema geben.

Kapitel 4 erläutert die Entwurfs- und Designmethoden, die bei der Modellierung des digitalen Systems eingesetzt werden. Der Entwurf von Regelsystemen und der RTL Modellierungsstil werden vorgestellt.

Kapitel 5 beschreibt sowohl die Identifikation der Fahrzeugregelstrecke als auch den analogen und digitalen Entwurf des Geschwindigkeitsreglers. Die Entwurfsschritte sowie die regelungstechnischen Grundlagen zum Entwurf des PI-Reglers werden erläutert und durch Diagramme und Zeichnungen vertieft.

Kapitel 6 konzentriert sich auf die VHDL-Modellierung und Implementierung des digitalen Geschwindigkeitsregelsystems. Die einzelnen Stufen der dreistufigen Reglerpipeline werden beschrieben und anhand von Simulationen getestet. Außerdem werden die Integration in das MicroBlaze Bussystem und die Entwicklung des Datenloggers vorgestellt.

Kapitel 7 stellt die messtechnische Analyse der Fahrzeugregelstrecke vor und wertet die gewonnenen Erkenntnisse der VHDL-Modellierung aus. Der FPGA Ressourcenverbrauch und eine Optimierung der zweiten Pipelinestufe werden betrachtet.

Kapitel 8 gibt eine Zusammenfassung der geschriebenen Arbeit.

2. Systemübersicht

Die Entwicklung der Geschwindigkeitsregelung ist ein Beitrag zum Aufbau eines Einparkassistenten (vgl. Abb. 2.1). Dieser soll ein autonom fahrendes Fahrzeug in eine vorgegebene Parklücke exakt einparken. Hierfür sind neben der Geschwindigkeitsregelung auch eine Abstandsregelung und der Kenntnissstand der Fahrzeugposition, die sogenannte Odometrie erforderlich, die mit dem Fahrzeug-Modell IP bestimmt wird. Für die Ermittlung der Systemdaten dienen Inkrementalgeber, Ultraschallsensoren und Infarotsensoren, die über Analog/Digital Wandler die IP Blöcke mit den entsprechenden Sensordaten versorgen. Ein WLAN und ein UART Modul dienen zum Übertragen von Messdaten, die während einer Testfahrt aufgezeichnet werden können.

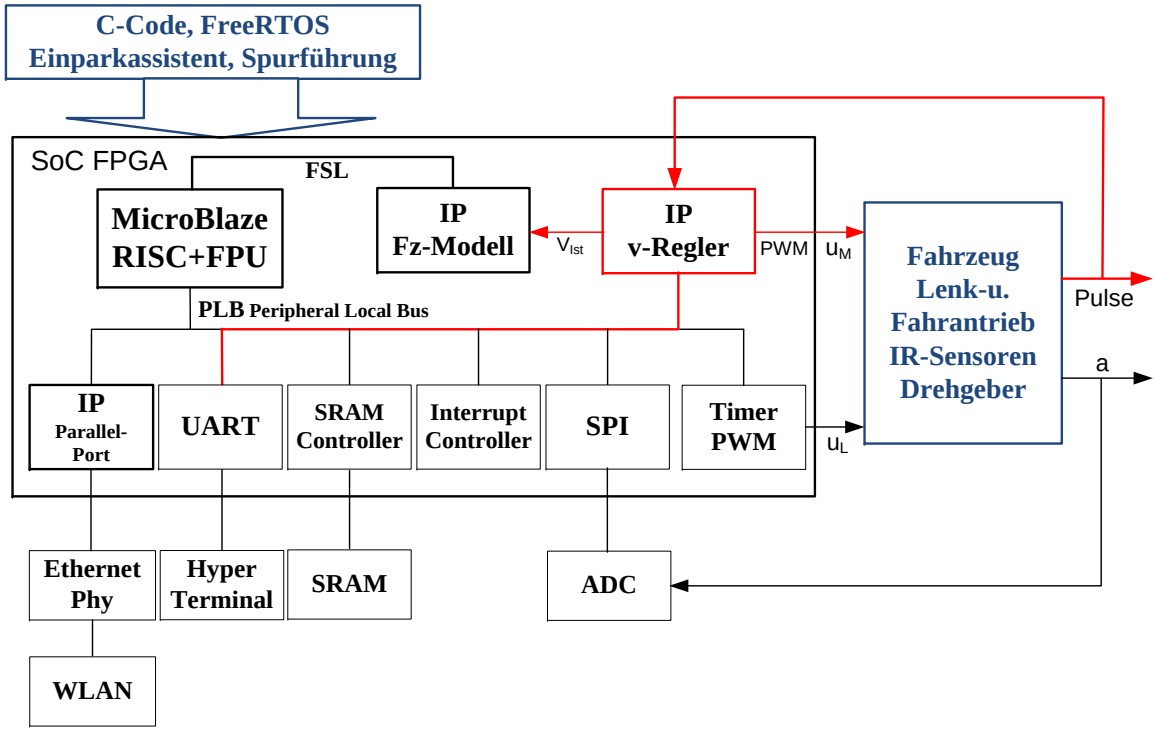

Abbildung 2.1.: SoC basierte Fahrzeugsteuerungsplattform

Die rot markierten Linien werden im Rahmen dieser Bachelorarbeit entwickelt. Über eine direkte Leitung zum Fahrzeug-Modell IP wird die momentane Ist Geschwindigkeit übertragen und zusätzlich für die Datenlogger Funktion in einem Software Register für den MicroBlazes bereitgestellt.

Das Geschwindigkeitsregelsystem besteht aus 3 Hauptkomponenten (vgl. Abb. 2.2): SoC Plattform mit einem Spartan-3E FPGA, Inkrementalgeber für die Geschwindigkeitsbestimmung und einem Bürstenmotor mit Fahrtensteller für den elektrischen Antrieb. Der V-Regler IP wird auf dem FPGA implementiert und besteht seinerseits wiederum aus 3 Komponenten:

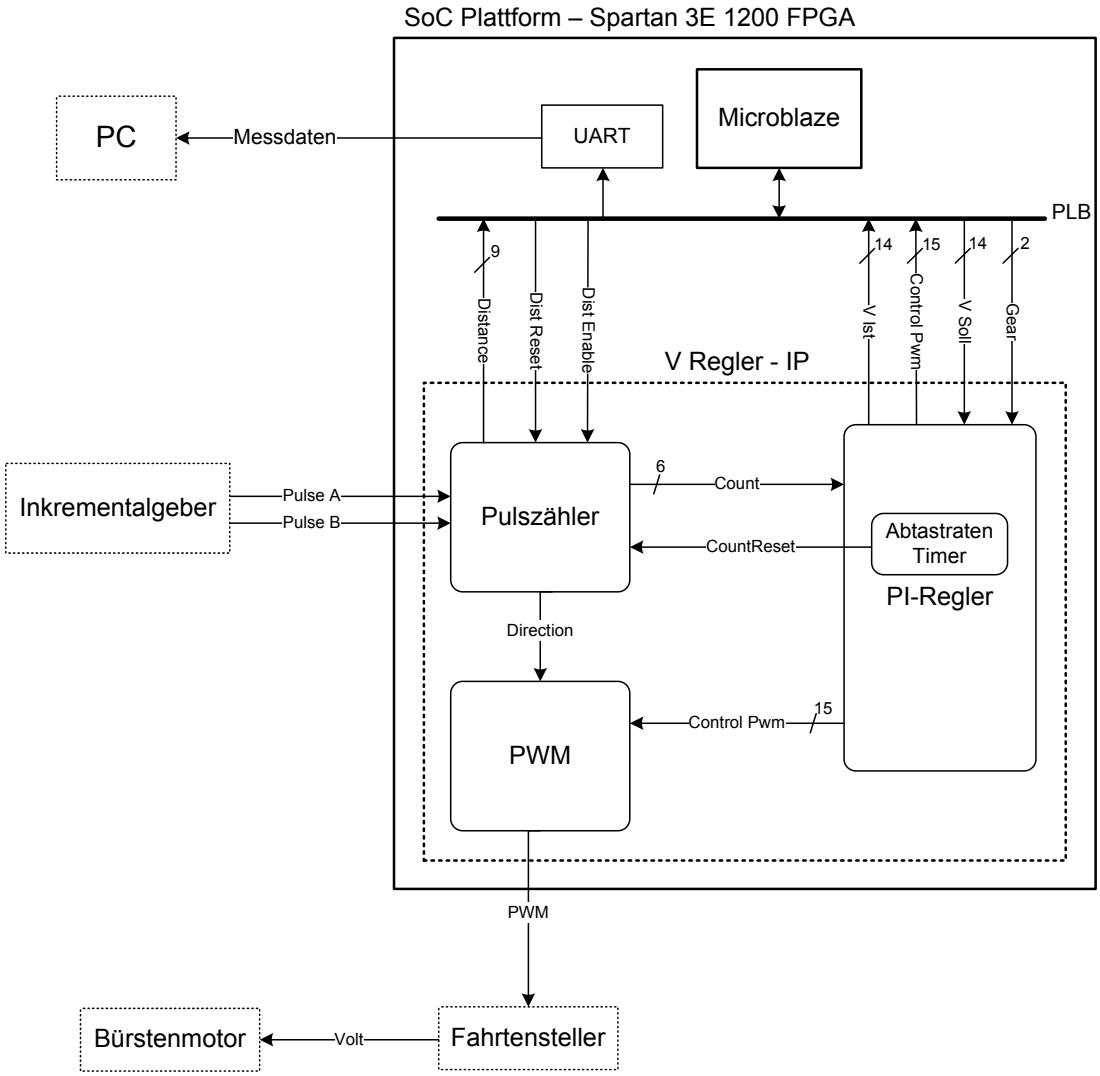

Abbildung 2.2.: Systemübersicht des Geschwindigkeitsregelsystems

- **MicroBlaze**
  Der MicroBlaze ist für die Vorgabe der Sollwerte für die Geschwindigkeit V_{Soll} und den Gang $Gear$ zuständig. Der Benutzer kann über Schalter am Board die gewünschten Werte einstellen und der MicroBlaze schreibt diese dann uber den PLB Bus in Software Register. Diese kann der V-Regler IP auslesen und seine Regelung dementsprechend anpassen. Weiterhin zeichnet der MicroBlaze Messdaten auf, die anschließend über den UART an den PC übertragen werden.

- **V Regler - IP-Core**
  Der V-Regler IP übernimmt die Hauptaufgabe der Regelung. Er berechnet die Ist Geschwindigkeit V_{Ist} des Fahrzeuges und schreibt diese in ein Software Register, welches anschließend vom MicroBlaze gelesen werden kann. Nachfolgend werden die einzelnen Teilmodule des V-Reglers erläutert:

  - **Pulszähler Modul**
    Das Pulszähler Modul zählt die vom Inkrementalgeber erzeugten Pulse $PulseA$ und $PulseB$ und bestimmt anhand der Flanken der Pulse (vgl. Abb. 2.3) die Drehrichtung $Direction$. Nach Ablauf einer Abtastperiode wird der Zählerstand $Count$ an den PI-Regler übertragen und anschließend zurückgesetzt. Für den Einparkassistenten wird zum Bemessen einer Parklücke ein zusätzlicher Counter implementiert, der über $DistEnable$ freigegeben wird und dann parallel zum Pulszähler die Pulse zählt bis $DistReset$ gesetzt wird (vgl. Abb. 6.2).

  - **PI-Regler Modul**
    In dem PI-Regler Modul wird aus dem Zählerstand $Count$ die momentane Ist Geschwindigkeit V_{Ist} berechnet und an den Fahrzeug-Modell IP übertragen. Aus Soll- und Ist Geschwindigkeit wird die Regelabweichung $e(t)$ gebildet und dem Regelalgorithmus zugeführt. Dieser berechnet daraus die Stellgröße $u(t)$ und sendet diese über $ControlPwm$ an das PWM Modul. Ein integrierter Abtastratentimer sorgt für die periodische Berechnung der Stellgröße $u(t)$ und erzeugt bei Erreichen des Endwertes einen Interrupt zur Signalisierung der Abtastperiode für weitere Berechnungen von anderen IP-Blöcke (vgl. Abb. 6.6).

  - **PWM Modul**
    Aus der Stellgröße $u(t)$ wird ein entsprechendes PWM Signal Pwm mit einer Periode von $20ms$ zur Ansteuerung des Fahrtenstellers erzeugt. Das PWM Modul modelliert ein Rechtecksignal mit einer konstanten Frequenz von 50 Hz und einer variablen Pulsbreite. Der Impuls hat, abhängig von Drehrichtung und Geschwindigkeit, ein Tastverhältnis zwischen $10\% - 20\%$.

- **Elektrischer Antrieb**
  Für den Antrieb des Fahrzeuges wird ein Bürstenmotor [15] mit Fahrtensteller eingesetzt. Der PWM-gesteuerte Fahrtensteller erzeugt aus dem PWM Signal Pwm eine Motorspannung $Volt$ im Spannungsbereich von $7,2V - 8,4V$ und steuert damit den Motor an. Durch den kugelgelagerten Antriebsstrang, kann die Kraft nahezu reibungslos auf die Räder übertragen werden.

Technische Details des Motors	
Nennspannung	7,2 V
Drehzahl bei max. Wirkungsgrad	22.000 UpM
Wirkungsgrad	78 %
Leerlaufdrehzahl	26.500 UpM

Tabelle 2.1.: Daten des Tamiya Super Stock TZ Motors [15]

- **Inkrementalgeber**
  Der Inkrementalgeber TCUT1300X01 [21] ist ein zweikanaliger optischer Sensor. Mit einem Lichtstrahl, der durch eine Lichtquelle erzeugt wird und durch eine Abtastplatte auf einen Phototransistor geleitet wird, wird die Geschwindigkeit und die Drehrichtung bestimmt. Die mit 120 Schlitzen versehenen Drehscheibe ist in der Felge des Fahrzeuges montiert und erzeugt bei Bewegung zwei um 90° Grad phasenverschobene Signale. Diese Pulse werden mit dem Pulszähler Modul gezählt und ausgewertet.

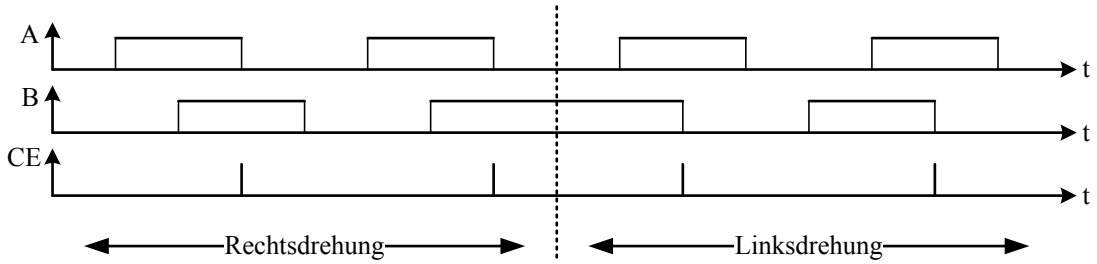

Abbildung 2.3.: Richtungsänderung des Inkrementalgebers

Pro Radumdrehung werden 120 Pulse erzeugt, die je nach Geschwindigkeit unterschiedlich breit sind. Ein Pulszähler zählt die Pulse wenn beide Pulskanäle eins sind und einer von beiden auf null geht (vgl. Abb. 2 und Abb. 6.2). Anhand der Anzahl von Pulsen $Count$ pro Abtastperiode und der Strecke pro Puls kann das Regler Modul die momentane Ist Geschwindigkeit berechnen. In Tabelle 2.2 werden weitere Daten für die Auswertung der Signale genannt.

Details Inkrementalgeber	
Radumfang	261 mm
Strecke pro Puls	2,175 mm
Max. Pulse pro Abtastperiode - 1.Gang	8 Pulse
Max. Pulse pro Abtastperiode - 2.Gang	10 Pulse
Max. Pulse pro Abtastperiode - 3.Gang	44 Pulse

Tabelle 2.2.: Daten des Inkrementalgebers

3. Technologieübersicht

3.1. System on Chip Design

Mit System on Chip (SoC) PLattformen können komplexe Hardware/Software Syteme auf einem Chip realisiert werden. Alle Funktionen eines Systems werden auf einem Integrated Circuit (IC) integriert und bestehen nicht mehr, wie ursprünglich üblich, aus einem Mikroprozessor und vielen weiteren ICs. Aufgebaut werden SoC Systeme meist aus einzelnen IP-Kernen (Intellectual Property), die jeweils eine eigene Funktionalität besitzen und über verschiedene Bussysteme miteinander verbunden sind. In zunehmeder Zahl werden SoCs auf FPGAs, die den klaren Vorteil der Programmierbarkeit bieten, implementiert. Hierbei kommen als Prozessoren entweder Hardcores (Power PC oder ARM) oder Softcores (Xilinx MicroBlaze oder Altera NiosII) zum Einsatz. Hardcores sind direkt im FPGA integriert und können nicht verändert werden, wobei Softcores als synthetisierbarer Quellcode vorliegen und vom Anwender auf seine Bedürfnisse angepasst werden können [24].
An die verschiedenen Busschnittstellen des MicroBlazes können nun die jeweilig IP-Cores angeschlossen werden, beispielsweise ein Ethernet Controller, ein UART oder aber ein selbst entwickelter IP (vgl. Abb. 3.1).

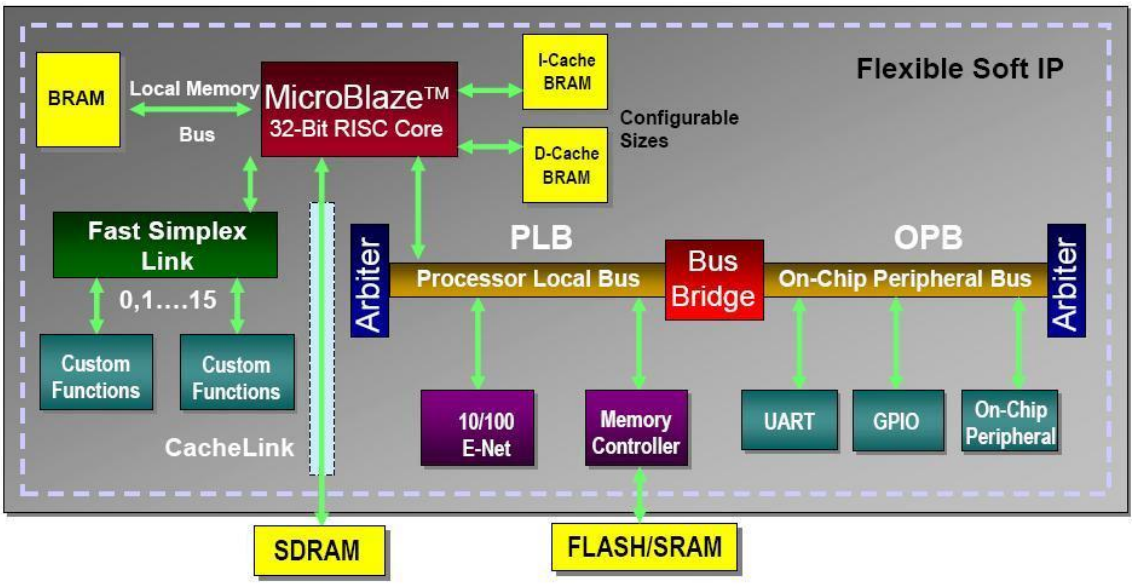

Abbildung 3.1.: Typische Implementierung eines SoC Systems mit MicroBlaze [27]

3.1.1. MicroBlaze

Der Xilinx MicroBlaze ist ein Softcore Prozessor mit einer 32bit Harvard RISC Architektur. Mit Hilfe des Embedded Development Studio (EDK) der Firma Xilinx, kann der MicroBlaze in ein SoC System mit FPGA integriert werden. Über den Local Memory Bus (LMB) werden die getrennten Adress- und Datenspeicher im BRAM mit dem MicroBlaze verbunden. Je nach Größe des FPGAs, kann der Prozessor ab Version 6.00 mit einer 3- bis 5-stufigen Pipeline implementiert werden. Um die Performance des Systems zu erhöhen können ebenfalls separate Instruktions- und Datencaches, die über den Xilinx Cache Link (XCL) angeschlossen werden können, verwendet werden. Je nach Einsatzzweck, kann der Entwickler den Prozessor mit weiteren konfigurierbaren Features ausstatten. Beispielsweise mit einer Floating Point Einheit oder einer Debug Logik [27]. Die grau hinterlegten Komponenten in Abb. 3.2 zeigen die optionalen Features des MicroBlazes. Durch den Anschluss von umfangreicher Peripherie und externen Speicherbausteinen an die Bussysteme des MicroBlaze, ensteht ein „System on Chip Design“ (vgl. Abb. 3.1).

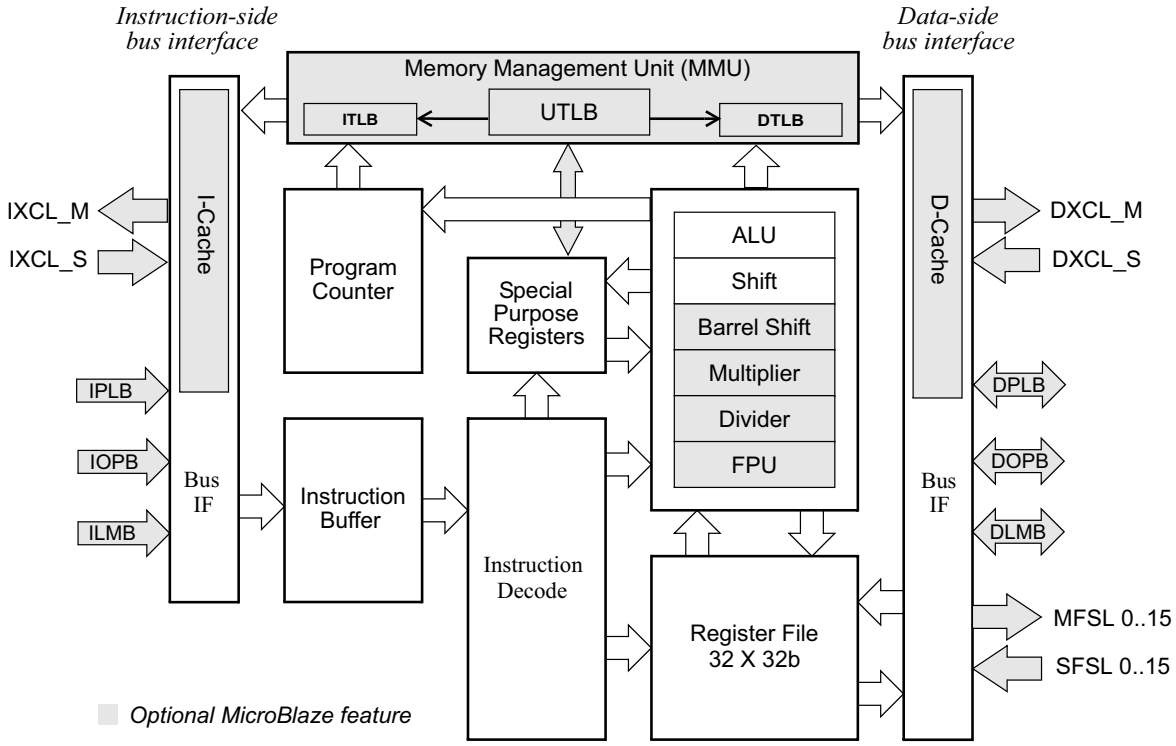

Abbildung 3.2.: MicroBlaze Block Diagramm [27]

3.1.2. Processor Local Bus - PLB

Der Processor Local Bus (PLB) ermöglicht eine schnelle Verbindung mit hoher Bandbreite und wenig Latenz zwischen zwei IP-Blöcken. In erster Linie dient er der Kommunikation zwischen MicroBlaze und schneller Peripherie, beispielsweise mit einem Ethernet- oder Memory Controller (vgl. Abb. 3.1).

Jeder PLB Transfer besteht aus zwei Phasen: Der Adressphase mit asynchronem Protokoll und der Datenphase.

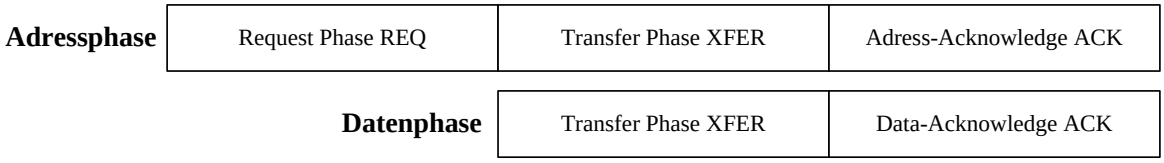

Abbildung 3.3.: Ablauf eines PLB Transfers

Durch Adresspipelining und entkoppelte Schreib- und Lesebusse mit unterschiedlichen Adressbussen ist es möglich, innerhalb eines Taktzyklus, Schreib- und Leseoperationen nebenläufig auszuführen und die Übertragungsrate zu erhöhen. An einem PLB Bussystem können bis zu 16 Bus-Master und beliebig viele Slaves angeschlossen werden. Die Master arbitrieren sich während der Adressphase innerhalb von drei Taktzyklen. Welcher Master die Kontrolle über den Bus erhält, wird entweder über Prioritäten oder über das Round-Robin-Verfahren festgelegt[28].

3.1.3. Fast Simplex Link - FSL

Der Fast Simplex Link (FSL) ist eine schnelle unidirektionale Punkt zu Punkt Verbindung zwischen MicroBlaze und IP Block (vgl. Abb. 3.1). Der Vorteil des Einsatzes des FSL ist der direkte Zugriff auf das Registerfile des Prozessors. Da die Kommunikation über asynchrone oder synchrone FIFOs erfolgt können die Busteilnehmer, der Master und der Slave, mit unterschiedlichen Taktraten betrieben werden. Der Master kann mit putfsl() Daten in die FIFO schreiben und der Slave holt sie mit dem Befehl getfsl() ab. Damit sowhol derMicroBlaze als auch der IP-Block lesen und schreiben können, müssen auf Grund der unidirektionalen Verbindung zwei FSL-Busse implementiert werden [26].

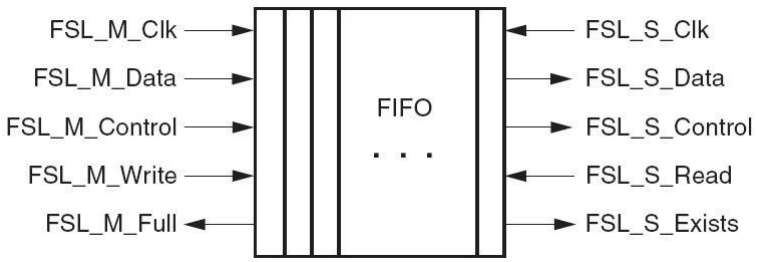

Abbildung 3.4.: FSL - Block Diagramm[26]

3.2. Nexys 2 Board

Als Zielhardware für das SoC System und für die Implementierung der Geschwindigkeitsregelung wurde ein Nexys 2 Board (Abb. 3.5) der Firma Digilent gewählt [3].

.

- Xilinx Spartan 3E XC3S1200E FPGA mit 19.512 Logikelementen (1)
- 16MB SDRAM (2) und 16MB Flash ROM (3)
- USB2 (4), VGA (5), 4 x 12Pin PMOD (6), 40 nutzbare FPGA I/O (7) und serielle Schnittstelle (8)
- 50 MHz Oszillator (9)
- 8 Schiebeschalter (10) und 4 Taster (11)
- 7-Segementanzeige mit 4 Ziffern (12) und 8 LEDs (13)

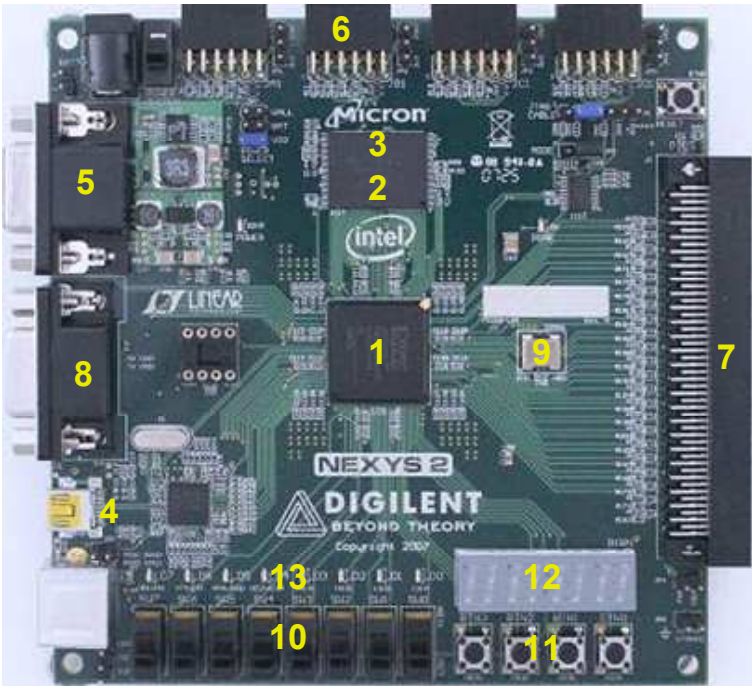

Abbildung 3.5.: Nexys 2 Board [3]

Über je einen Pin am PMOD Anschluss werden die Pulskanäle des Inkrementalgebers und die PWM Leitung des Fahrtenstellers angeschlossen. Die Schiebeschalter werden genutzt, um den gewünschten Gang einzustellen. Zur Visualisierung wird der gewählte Gang auf der sieben Segmentanzeige angezeigt. Nach einer Testfahrt können über die serielle Schnittstelle per UART die Messdaten (Geschwindigkeit und Stellgröße) auf einen PC übertragen werden.

4. Design und Entwurfsmethodik

Der strukturelle Entwurf von komplexen digitalen Systemen erfordert eine Reihe von verschiedenen durchzuarbeitenden Phasen. Zu Beginn erstellt der Entwickler ein High-Level Blockschaltbild, das eine Gesamtübersicht des Systems enthält. Jeder Block enthält seine konkrete Funktionalität, die mit mathematischen Formeln oder Algorithmen spezifiziert wird. Die Schnittstellen zwischen den Blöcken und I/O Ports, die nach außen führen, werden definiert und ein, für die Kommunikation notwendiges, Format wird gewählt.
Nachdem eine grobe Übersicht des Systems besteht, werden die einzelnen Funktionalitäten in Komponenten zerlegt und mit verschiedenen Entwurfsmethoden entwickelt. Jede Komponente erfüllt eine Teilaufgabe des kompletten Systems und wird durch lokale Signale mit anderen Komponenten verbunden. So ergibt sich ein strukturiertes Gesamtsystem. Bei der Strukturierung von komplexen digitalen System hat sich bewährt, Datenpfad und Steuerpfad voneinander zu trennen [4].

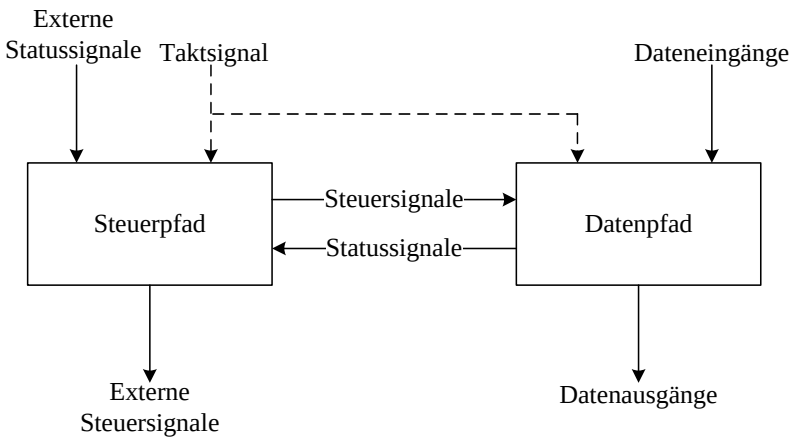

Abbildung 4.1.: Trennung von Datenpfad und Steuerpfad [4]

Der Datenpfad besteht aus kombinatorischen und getakteten Schaltungselementen. Er beschreibt die vom Algorithmus auszuführenden arithmetischen Operationen. Über den Dateneingang gelangen Daten von außen in das System und werden mit Hilfe von Steuersignalen verarbeitet und ausgegeben. Statussignale teilen dem Steuerpfad mit, in welchem Zustand er sich gerade befindet.

Der Steuerpfad kann als endlicher Automat (vgl. 4.1) dargestellt werden. Er kontrolliert den Ablauf des Algorithmus indem er Steuersignale an den Datenpfad oder an externe Komponenten sendet. Bei der Geschwindigkeitsregelung übernimmt der Abtastratentimer die Aufgabe des Steuerwerks. Er liefert die Zeitpunkte für die Brechnung der neuen Stellgröße u(t).

4.1. Endlicher Automat - Finite State Machine (FSM)

Ein Endlicher Automat ist ein Verhaltensmodell eines Systems mit einer endlichen Anzahl von Zuständen Z. Mit periodisch getakteten Zustandsautomaten können zyklische Funktionsabläufe realisiert werden. Sie dienen als Grundlage zum Entwurf von komplexen digitalen Systemen und sind definiert durch das Quintupel {Z,E,A,f,h}, wobei E und A die Menge der Ein- und Ausgangssignalen darstellen. Die Zustandsübergangsfunktion f wird durch eine logische Bedingung beschrieben, die erfüllt sein muss, damit der Automat in den nächsten Zustand wechselt. Hierfür muss der zurückliegende Zustand, also die Information über die Vergangheit des Systems, gespeichert werden. Bei einem Zustandswechsel wird die Ausgabefunktion h ausgelöst. Je nach Art des Automaten, ist h entweder nur abhängig von dem aktuellen Zustand oder von Zustand und der Eingabe [4].

4.1.1. Mealy Automat

Im Gegensatz zum Moore Automaten, hängt die Ausgabe bei einem Mealy Automaten von dem aktuellen Zustand und der Eingabe ab. Die blaue Linie in Abb. 4.2 zeigt den Unterschied. Es existiert ein direkter Pfad vom Eingang zum Ausgangschaltnetz. Der Einsatz von Mealy Automaten führt oft zur Verringerung der zu implementierenten Zustände [13].

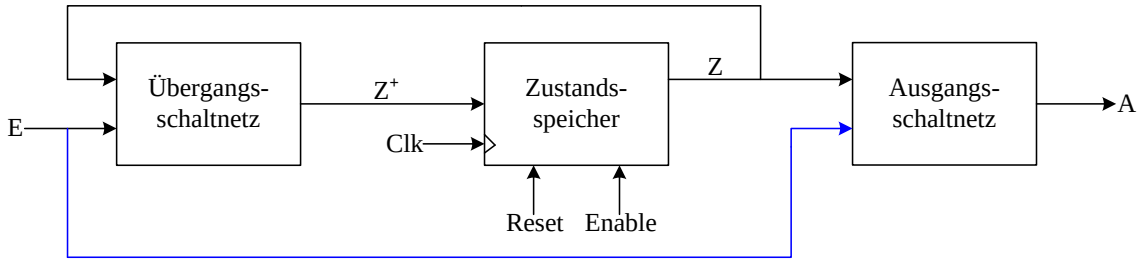

Abbildung 4.2.: Mealy Automat [13]

- Übergangsschaltnetz:
  Ist für die Folgezustandsberechnung zuständig. Es besteht aus einer rein kombinatorischen Logik, die in einem eigenständigen VHDL-Prozess modelliert wird und mit case when Statements aufgebaut ist.
- Zustandsspeicher
  Eine Anzahl n von synchron getakteten Flip-Flops, mit denen 2^n Zustände binär codiert werden können, ist für die Speicherung des Folgezustandes zuständig. Der Zustandsspeicher wird mit einem getakteten Prozess modelliert.
- Ausgangsschaltnetz
  Erzeugt die zu einem Zustand entsprechenden Ausgangssignale. Die Aktualisierung der Ausgänge erfolgt entweder durch einen Zustandswechsel oder durch eine Änderung der Eingangssignale. Dieser Prozess besteht wiederum nur aus rein kombinatorischer Logik.

4.2. RTL-Modellierungsstil

Die Register Transfer Ebene ist ein Abstraktionslevel, das beim Entwurf von digitalen Hardwareschaltungen, den Signalfluss der Daten zwischen zwei Registern beschreibt. Jede Schaltung besteht aus kombinatorischer Logik und aus Speicherelementen, die parallel und taktsynchron die Ausgangsdaten der Logik speichern (vgl. Abb. 4.3). Bei der Synthese wird aus dem RTL Modell eine logische Schaltung generiert und durch „Place and Route“ ein Layout erzeugt.

Bei der VHDL Modellierung des PI Reglers werden beide Blöcke in einem VHDL Prozesse modelliert. Die kombinatorische Logik, aufgebaut aus einer Anzahl an Gattern, berechnet das Ausgangssignal der Pipelinestufe und speichert dieses parallel zum Takt in einem Register. Die gespeicherten Ausgangsdaten der Logik bestimmen die Eingangsdaten der folgenden Logik. Das Zeitschema auf RTL Ebene wird durch das Zählen von Taktzyklen gegeben, die je nach Implementierung auf steigende oder fallende Flanke reagieren.

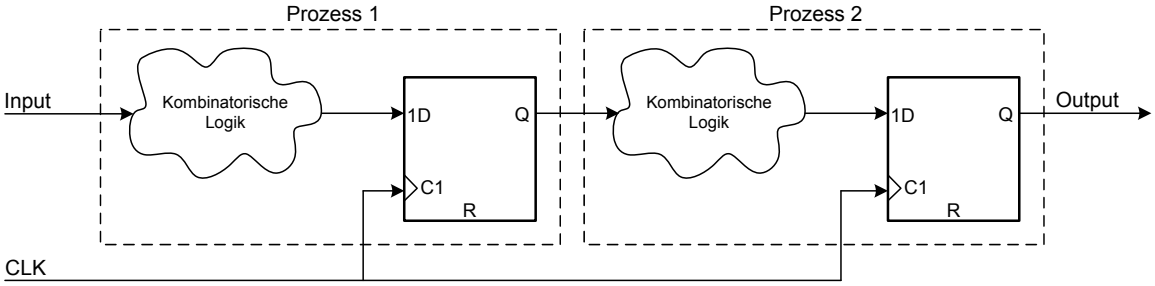

Abbildung 4.3.: Register Transfer Ebene mit Gliederung in Prozessen

Die RTL Modellierung wird so aufgebaut, dass am Ausgang von kombinatorischer Logik stets ein Register die Daten speichert und das Ergebniss nicht direkt in eine weitere kombinatorische Logik gelangt. Anderfalls können kombinatorische Schleifen entstehen und Signallaufzeiten nicht eingehalten werden. Für den Datenpfad entsteht eine Pipeline, die die Daten parallel verarbeitet und speichert. Damit die Register stets den aktuellen Wert speichern und der längste Signallaufzeitpfad nicht überschritten wird, hat die Logik maximal eine Taktperiode Zeit um die Ausgangssignale zu berechnen. Zur Modellierung der Arithmetik werden Variablen zur Wiederverwendung in getaktteten Prozessen mit sequentiellen Anweisungen eingesetzt. Dies hat den Vorteil, dass während einer Taktperiode mehrere arithmetischen Operationen ausgeführt werden können, ohne das Ergebniss zu speichern. Der Entwickler legt die Architektur des Designs fest und bestimmt welche Datensignale in Registern gespeichert werden sollen. Außerdem definiert er alle Takt-, Steuer- und Rücksetzsignale, die für die Implementierung notwendig sind [17].

Damit der VHDL Code besser lesbar ist und alle Funktionen klar definiert sind, werden alle Teilaufgaben in einzelnen Komponenten *component* modelliert. So ist es möglich, dass auch einzelne Module für weitere Aufgaben wiederverwendbar sind. Die Eingangs- und Ausgangssignale der Komponenten werden bei der Instanziierung miteinander verdrahtet und können somit miteinander kommunizieren und ihre Daten austauschen. Bei der Geschwindigkeitsregelung werden drei Komponenten deklariert und in einer Top Entity zusammengefasst: Der Pulszähler, der PI Regler und das PWM Modul.

4.3. Entwurf eines Regelsystems

Von der Aufgabenstellung bis zur Implementierung eines Regelsystems erfolgen viele Arbeitsschritte (vgl. Abb. 4.4). Im ersten Schritt werden die Aufgaben des Reglers und alle technischen Randbedingung, wie die Auslegung der Sensorik und die Messverfahren, festgelegt. Anschließend werden die Sprungantworten gemessen und das Übertragungsglied der Regelstrecke wird eindeutig identifiziert. Aus dieser Identifikation werden die Systemparameter bestimmt und ein mathematisches Modell in Abhängigkeit von den Randbedingungen erarbeitet. Im vierten und fünften Schritt wird eine geeignete Reglerstruktur festgelegt und die passenden Einstellwerte der Reglerparameter berechnet. Mit geeigneten Testszenarien können sowohl Regler als auch Regelstrecke auf korrekte Funktionsweise getestet werden. Lässt sich ein gewünschtes Verhalten nicht einstellen, so können die Reglerparameter optimiert werden.

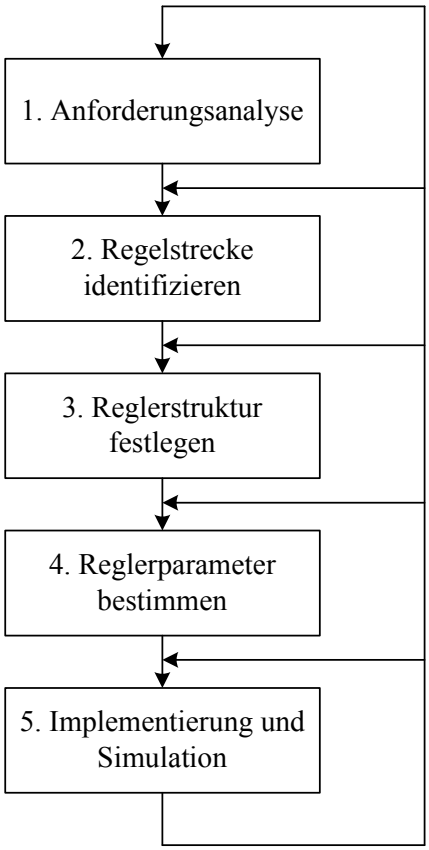

Abbildung 4.4.: Vorgehensmodell zum Reglerentwurf

Reglerentwürfe erfordern teilweise das mehrmalige Durcharbeiten aller Phasen, da bestimmte Systemzustände oder Randbedingungen erst bei Inbetriebnahme bekannt werden. Beispielsweise können bestimmte Störgrößen oder Fehlverhalten erst zur Laufzeit erkannt werden [12].

4.3.1. Timing eines digitalen Regelkreises

Jede digitale Regelung setzt einen Abtastvorgang (vgl. 5.4.1) voraus, der diskrete Zeitwerte aus einem kontinuierlichen Zeitverlauf entnimmt. Hierfür wird ein Timer implementiert, der die benötigten Abtastzeitpunkte T für die Berechnung des Stellsignals liefert. Nach Ablauf einer Abtastperiode werden die Messwerte eingelesen und die Regelabweichung $e(t)$ gebildet. Der Regelalgorithmus berechnet aus $e(t)$ die Stellgröße $u(t)$ und lässt diese auf die Regelstrecke einwirken. Da der Algorithmus zurückliegende Werte mit in die Berechnung einbezieht müssen sowohl die Stellgröße $u(t)$ als auch die Regelabweichung $e(t)$ gespeichert werden.

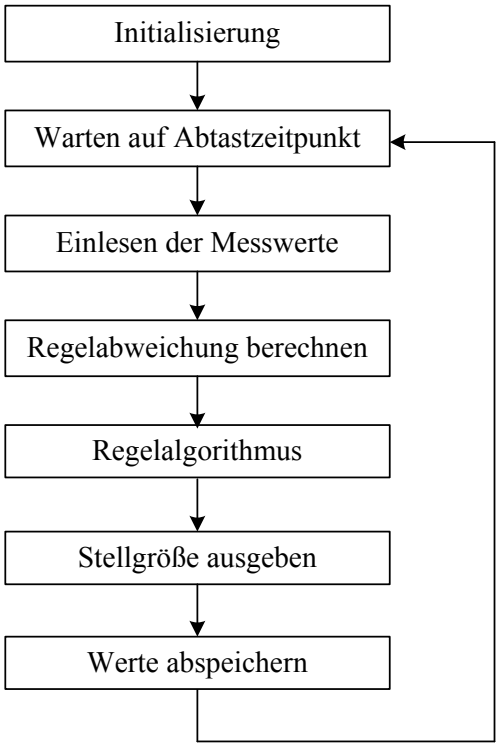

Abbildung 4.5.: Arbeitsschritte eines digitalen Reglers

In der Regelungstechnik werden Regler meist so entworfen, dass die Regelstrecke entweder ein gutes Führungs- oder Störverhalten hat. Der Regler soll bei einer Sollwertänderung möglichst schnell und präzise reagieren können und den Istwert auf den Sollwert bringen, wobei die Stellgröße $u(t)$ einen bestimmten Grenzwert nicht überschreiten darf. Bei einer Störung $z(t)$ des Regelkreises soll der Regler die Störgröße sofort kompensieren können, sodass sie keine Auswirkung auf die Regelgröße hat [14].

4.4. Zahlendarstellung im Q-Format

Zur Modellierung der Integeraritmetik mit dem RTL Coding Style wird das Q-Format $Q_{n.m}$ verwendet. Dies hat den Vorteil, dass keine externen Floating Point Bibliotheken eingebunden werden müssen. Das Q-Format besteht aus einer festen Anzahl von Fractionalbits F_m, einer variablen Anzahl n von Integerbits G_n und einem Vorzeichenbit S. Für die Wahl des Datentyps empfiehlt sich der Vektordatentyp signed oder unsigned, da hierfür die arithmetischen Operatoren +,- und * in den entsprechenden Bibliotheken definiert sind. Für ein $Q_{n.m}$- Format werden $m + n + 1$ Bits benötigt. Die Zahl im Q-Format wird vom Synthesewerkzeug als normale Binärdarstellung eines Integerstrings verarbeitet und nur der Entwickler hat die normale Betrachtungsweise mit Binärpunkt: [13]

$$S\, G_{\mathrm{n}} G_{\mathrm{n\text{-}1}} ... G_0 \bullet F_{\mathrm{m}} F_{\mathrm{m\text{-}1}} ... F_0$$

Um eine Dezimalzahl in das Q-Format zu konvertieren muss die Zahl mit 2^m multipliziert werden und anschließend auf die nächstmögliche Ganzzahl gerundet werden. m gibt die Anzahl der Nachkommastellen an. Dieses Ergebnis in binär repräsentiert die Zahl im Q-Format. Durch das Auf- bzw. Abrunden enstehen Ungenauigkeiten, sogenannte Quantisierungsfehler. Bei der Rücktransformation werden die Q-Format Zahl direkt von binär in dezimal gewandelt und anschließend durch 2^m geteilt.

- **Addition von Zahlen im Q-Format**
  Bei der Addition muss der Ergebnisvektor für eine überlauffreie Realisierung eine Bitstelle breiter sein als der größte Summand:

$$SGG \bullet 0...9 \; + \; SGG \bullet 0...9 \; = \; SGGG \bullet 0...9$$

- **Multiplikation von Zahlen im Q-Format**
  Bei der Multiplikation muss der Ergebnissvektor so breit gewählt werden, wie die Summe der Vektorbreiten der Summanden:

$$SGG \bullet 0...9 \; * \; SGG \bullet 0...9 \; = \; SGGGGG \bullet 0...19$$

Einige Kennwerte für Zahlen im Q-Format genannt werden in Tabelle 4.1 gezeigt. Als Beispiel dient ein Vektor mit der Breite von 10 Bit. Die Reglerparameter in der V-Regelung werden ebenfalls mit 10 Fractionalbits implementiert. Dies bietet für die Geschwindigkeitsregelung ein genügendes Maß an Genauigkeit

Vektorlänge	B + 1	10 Bit
Größte positive Zahl	$1 - 2^{-B}$	0.998047
Kleinste negative Zahl	-1	-1
Quantisierungsstufe	$D = 2^{-B}$	0.001953
Maximaler Quantisierungsfehler	$D/2 = 2^{-(B+1)}$	0.000977

Tabelle 4.1.: Formatkennwerte für Q10 - Zahlendarstellung [13]

5. Regelungstechnische Analyse

Zu Beginn des digitalen Reglerentwurfs wird zuerst ein analoger Regler, spezifiziert durch eine Übertragungsfunktion oder Differentialgleichung, für eine analoge Regelstrecke entworfen. Diese Differentialgleichung wird zur Implementierung auf einem FPGA in eine zeitdiskrete Differenzengleichung transformiert. Da ein Digitalrechner Messwerte nicht kontinuierlich verarbeiten kann, spricht man bei einer digitalen Regelung von einem zeitdiskreten oder quasikontinuierlichen Modell. In Kapitel 4.3 wurde die prinzipielle Vorgehensweise für den Entwurf eines Regelsystems vorgestellt. Nachfolgend werden die einzelnen Schritte, die für einen erfolgreichen Geschwindigkeitsreglerentwurf notwendig sind, erläutert (vgl. Abb. 1.4).

5.1. Identifikation der Fahrzeugregelstrecke

Als Regelstrecke wird das System bezeichnet, dessen Verhalten durch eine Regelung zielgerichtet verändert werden soll. Bei der Geschwindigkeitsregelung stellt das Carolo Cup Fahrzeug die Regelstrecke dar. Das Übertragungsverhalten einer linearen Regelstrecke lässt sich in den meisten Fällen mit genügender Genauigkeit aus der Sprungantwort ermitteln. Diese ist definiert als die Reaktion $y(t)$ des Systems auf eine sprungförmige Veränderung der Eingangsgröße $u(t)$ [18]. Die Sprungantwortmethode setzt voraus, dass sich die Strecke zu Beginn $t = 0$ in einem stationären Zustand befindet. Zur Identifikation des Übertragungsgliedes, wurde das Fahrzeug im aufgebockten Zustand in jedem Gang auf die maximale Geschwindigkeit beschleunigt, dies entspricht einer Stellgröße $u(t)$ von 100%. Abb. 5.1 zeigt exemplarisch die gemessene und die interpolierte Sprungantwort für das Fahrzeug im 1. Gang mit maximaler Stellgröße $u(t)$.

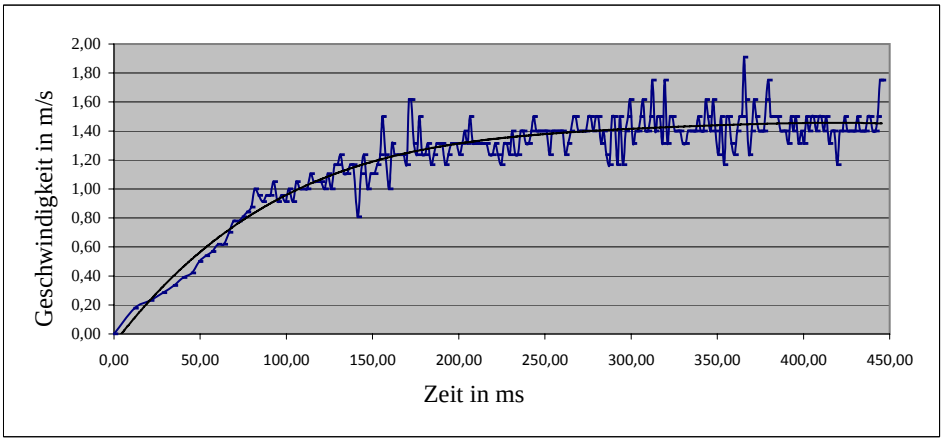

Abbildung 5.1.: Sprungantwort der Fahrzeugregelstrecke bei 100% PWM im 1.Gang

Die Sprungantwort zeigt einen ähnlichen Verlauf wie eine e-Funktion und kann beschrieben werden durch:

$$h(t) = K_S(1 - e^{-\frac{t}{T_1}}) \tag{5.1}$$

Dieser $h(t)$ Verlauf beschreibt die Sprungantwort eines Verzögerungsgliedes 1. Ordnung, ein sogenantes PT_1 Glied ohne Verzugszeit. Die Übertragungsfunktion eines PT_1 Gliedes lautet:

$$G_s(s) = \frac{K_S}{(1 + T_1 s)} \tag{5.2}$$

Wobei K_S der Proportionalbeiwert oder Verstärkungsfaktor der Strecke ist. Die Zeitkonstante T_1 gibt den Zeitraum an, den der exponentiell steigende Prozess braucht, um auf 63,2 % des Endwertes anzusteigen [18].

Zur Bestimmung der Parameter K_S und T_1, wird das Wendetangentenverfahren angewandt. Hier wird eine Tangente zum Zeitpunkt $t = 0$ durch den Ursprung gelegt. Der stationäre Endwert bestimmt mit seiner Höhe y den Proportionalbeiwert K_S. Dies ist im Falle der Geschwindigkeitsregelung die maximale Endgeschwindigkeit V_{max} pro Gang. Der Schnittpunkt der Tangente mit K_S auf der Zeitachse definiert die Zeitkonstante T_1 [12].

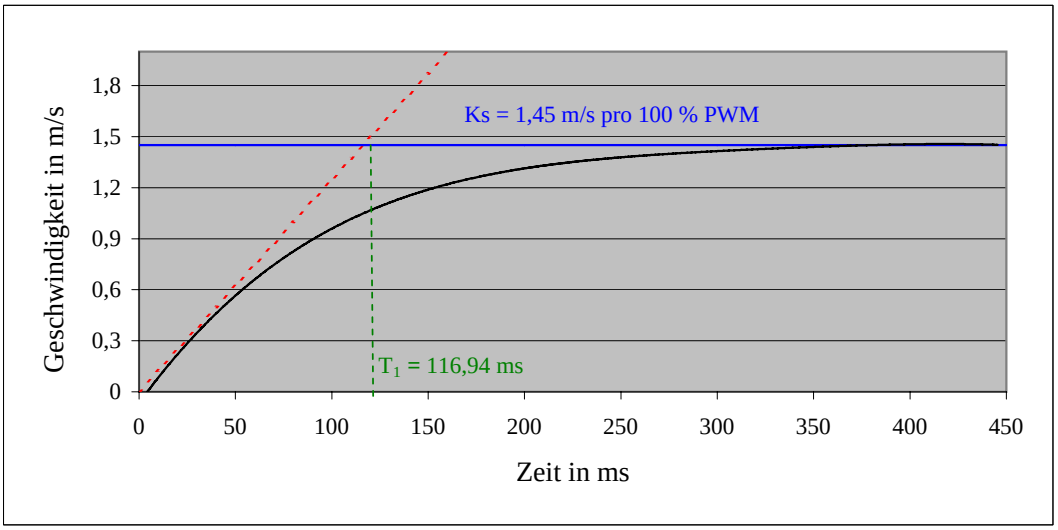

Abbildung 5.2.: Berechnung der Regelstreckenparameter des 1. Ganges

Die Berechnung und die Darstellung der Sprungantworten des zweiten und dritten Ganges werden in Kapitel 7.1 dargestellt.

	1.Gang	2.Gang	3.Gang
$K_S = V_{max}$	1,45 m/s	2,63 m/s	3,80 m/s
T_1	116,94 ms	80,92 ms	245,16 ms

Tabelle 5.1.: Verstärkungsfaktor K_S und Zeitkonstante T_1 der jeweiligen Gänge

5.2. PI-Regler

Der Regler hat die Aufgabe, die Regelgröße mit dem Sollwert zu vergleichen und bei Abweichungen die Stellgröße $u(t)$ so zu verändern, dass im Ideafall die Regelabweichung $e(t)$ gegen Null geht. Für die Geschwindigkeitsregelung wurde ein PI Regler gewählt. Dieser ist die Kombination aus P- und I-Regler und besteht aus einem proportionalwirkenden P und einem integralwirkenden I Anteil. Auf einen differenzierenden D-Anteil kann verzichtet werden, da die zu regelnde Größe, die Geschwindigkeit, nicht träge ist. Der D-Anteil wirkt sich nicht auf $e(t)$ aus, sondern nur auf dessen Änderungsgeschwindigkeit. Der Anteil des P-Teils an der berechneten Stellgröße $u(t)$ ist immer proportional der Regelabweichung $e(t)$ [18].

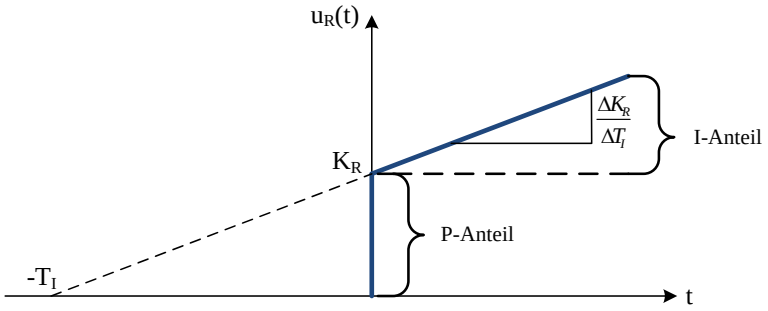

Abbildung 5.3.: Sprungantwort eines PI Reglers bei Sprung der Eingangsgröße $e(t)$

Der P-Anteil reagiert aufgrund des fehlenden Zeitanteils unmittelbar auf eine Regelabweichung, jedoch kann er diese nicht vollständig beseitigen. Dies liegt daran, dass wenn $e(t) = 0$ wird, dann wird auch $u(t) = 0$ und somit wird keine neue Stellgröße erzeugt. Der I-Anteil kann diese bleibende Regeldifferenz vollständig eliminieren, indem er $e(t)$ integriert und damit auf die Stellgröße $u(t)$ einwirkt. Jedoch neigt der I-Regler zu Schwingungen und ist im Vergleich zum P-Regler langsamer, da er die Vergangenheit von $e(t)$ mit in die Berechnung von $u(t)$ einbezieht. Der PI Regler kombiniert den Vorteil des P-Reglers, nämlich schnelle Reaktion, mit dem Vorteil des I-Reglers, der stationären Ausregelung. Der kontinuierliche zeitliche Verlauf der Stellgröße $u(t)$ ergibt sich:

$$u_R(t) = K_R e(t) + \frac{K_R}{T_I} \int_0^t e(\tau) d\tau \tag{5.3}$$

Aus der Gleichung 5.3 folgt die Übertragungsfunktion $G_R(s)$ des PI Reglers:

$$G_R(s) = \frac{U(s)}{E(s)} = K_R + \frac{K_R}{(T_I * s)} \tag{5.4}$$

Der Verstärkungsfaktor K_R und die Zeitkonstante oder Integrationsbeiwert T_I werden als Einstellwerte des Reglers bezeichnet. Durch geeignete Wahl dieser Werte, lässt sich der Regler so einstellen, dass ein möglichst günstiges Regelverhalten entsteht und der Regler die Regeldifferenz schnell und vollständig beseitigen kann. Nachfolgend werden die unbekannten Parameter K_R und T_I mit der quadratischen Regelfläche berechnet.

5.3. Quadratische Regelfläche

Die quadratische Regelfläche ist ein Integralkriterium zur Bestimmung der Regelgüte und zur Optimierung der Reglerparameter. Vorrausgesetzt die Übertragungsfunktion der Regelstrecke liegt vor (vgl. Gleichung 5.2), dann kann dem Regelkreis eine Maßzahl I zugeordnet werden, die sogenannte Regelgüte. Bei Integralkriterien wird die Fläche zwischen der 100% Linie, dem Sollwert und der Führungsübergangsfunktion $h(t)$ gebildet und als Gütemaß I genommen [18].

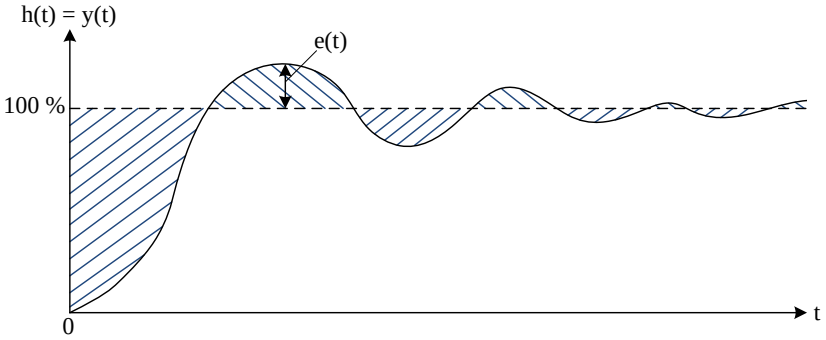

Abbildung 5.4.: Antwort eines Regelkreises mit eingezeichneter Regelfläche

Die quadratische Regelfläche I_q bestimmt diese Fläche durch Integration der quadratischen Regelabweichung e^2 über der Zeit [11].

$$I_q = \int_0^\infty e^2(t)\mathrm{d}t = Min \tag{5.5}$$

Ein günstiges dynamisches Regelverhalten liegt vor, wenn I minimal ist. Abbildung 5.5 zeigt die quadrierte Regelabweichung $e^2(t)$ aus Abb. 5.5. Die optimalen Reglerparameter K_R und T_I sind nun diejenigen, durch die I minimal wird.

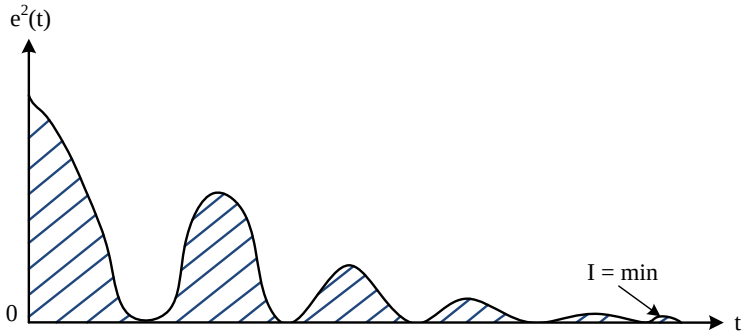

Abbildung 5.5.: Darstellung der quadratischen Regelfläche

Das quadratische Gütekriterium wird in der Regelungstechnik oft eingesetzt, wobei sich in Folge der Quadrierung von $e(t)$ große Anfangsabweichungen ungünstig auswirken. Für die Geschwindigkeitsregelung ist dieses Kriterium aber hinreichend geeignet, da es nur mässige Überschwingungen und kurze Anregelzeiten gibt.

Da die Berechnung von optimalen Reglerparameter nach dem quadratischen Gütekriterium recht aufwendig ist, wurden für die Kombination von Regelstrecken mit Standardreglertypen die optimalen Einstellwerte in allgemein normierter Form berechnet [18]. Die normierte quadratische Regelfäche I_{qN} wird in dimensionsloser Form angegeben:

$$I_{qN} = \frac{I_q}{(z_0 K_S)^2 * T_1} = \frac{T_{IN}}{2K(K+1)} \tag{5.6}$$

Wobei z_0 die Störung am Eingang der Regelstrecke darstellt (vgl. Abb. 1.4). Mit der Beziehung $K = K_R K_S$ und $T_{\text{IN}} = T_I/T_1$ kann Gleichung 5.6 nach T_I aufgelöst werden:

$$T_I = I_q * \frac{2K_R(K_R K_S + 1)}{{z_0}^2 K_S} \tag{5.7}$$

Diese Gleichung hat neben den gesuchten Einstellwerten T_I und K_R auch I_q als unbekannten Parameter. Aus diesem Grund wird im ersten Schritt der Verlauf der Regelabweichung $e(t) = V_{\text{Soll}} - V_{\text{Ist}}$ zur Berechnung von I_q abgeschätzt. Die Sprungantwort aus Abb. 5.1 zeigt, dass V_{Ist} bei einem Sprung von 0 m/s auf V_{Max} einen exponentiellen Verlauf hat. Daraus ergibt sich für $e(t)$ die Beziehung:

$$e(t) = V_{Max} - V_{Max}(1 - e^{-t/T_1}) = V_{Max} e^{-t/T_1} \tag{5.8}$$

Die geschätzte quadratische Regelfläche I_q^* lässt sich nun, mit der allgemeinen Gleichung 5.5, aus dem Verlauf von $e(t)$ berechnen:

$$I_q^* = V_{Max}^2 \int_0^\infty e^{-2t/T_1} \mathrm{d}t = \frac{V_{Max}^2}{2} * T_1 \tag{5.9}$$

Nimmt man nun an, dass z_0 gleich der maximal möglichen Regelabweichung $z_0 = e_{Max} = U_{PwmMax} = 1$ ist, dann kann T_I in Abhängigkeit von K_R dargestellt werden und K_S ist gleich V_{Max}, da $U_{PwmMax} = 1$ ist:

$$T_I = T_1 * K_S K_R(K_R K_S + 1) \tag{5.10}$$

Die Zeitkonstante T_I gibt an, wie schnell der Regler die Regelabweichung e integriert. K_R hingegen bestimmt, wie stark der Regler auf eine Reglerabweichung reagieren soll, indem er zu Beginn der Regelung einen Sprung auf die Höhe K_R erzeugt. Um die optimalen Einstellwerte für T_I zu finden, muss K_R ebenfalls geschätzt werden. Da der Regler schnell reagieren und nicht zu sehr überschwingen soll, darf K_R nicht zu groß und nicht zu klein gewählt werden. Wenn $K_R > K_S$ wäre, dann würde die Stellgröße $u(t)$ bei jeder Abweichung über den Sollwert V_{Soll} springen.

Aus diesem Grund wurde für K_R folgende Annahmen getroffen:

$$K_{R1} = \frac{U_{PwmMax}}{V_{Max}} = \frac{1}{K_S} \tag{5.11}$$

$$K_{R2} = \frac{1}{2}\frac{U_{PwmMax}}{V_{Max}} = \frac{1}{2K_S} \tag{5.12}$$

Daraus ergibt sich für T_I:

$$T_{I1} = 2 * T_1 \tag{5.13}$$

$$T_{I2} = T_1 * \frac{3}{4} \tag{5.14}$$

Mit dem zuvor beschriebenen Verfahren können für den PI-Regler jeweils zwei Wertepaare für K_R und T_I berechnet werden. Durch Simulation des Regelsystems werden anschließend die Einstellparameter, mit denen ein günstiges Regelverhalten erreicht werden kann, korrigiert. Generell geht man bei einem Regelkreisentwurf mittels Simulation von anfangs festgelegten bzw. berechneten Regelparametern und Gütemaßen aus. Es wird dann versucht K_R und T_I so einzustellen, dass das System ein günstiges Regelverhalten aufweist und das Gütemaß der Regelung erfüllt ist. Wenn das gewünschte Verhalten durch die berechneten Parameter nicht erfüllt werden kann, muss gegebenenfalls die Abschätzungen für K_R wiederholt werden oder man wählt eine andere Reglerstruktur [18].

	1.Gang	2.Gang	3.Gang
K_{R1}	0,68966 s/m	0,38023 s/m	0,26316 s/m
T_{I1}	232,98 ms	161,84 ms	490,32 ms
K_{R2}	0,34483 s/m	0,19011 s/m	0,13158 s/m
T_{I2}	87,37 ms	60,69 ms	183,87 ms

Tabelle 5.2.: Berechnung des Integrationsbeiwerts T_I und des Verstärkungsfaktors K_R

Mit dieser Vorgehensweise kann der Regelkreis mit einem analogen zeitkontinuierlichen PI-Regler entworfen werden. Für die Implementierung auf einem FPGA muss dieses System in ein digitales zeitdiskretes System transformiert werden. Nachfolgend wird die Diskretisierung des Systems dargestellt.

5.4. Digitale Realisierung eines analogen Reglers

Im vorherigen Abschnitt wurde für den Geschwindigkeitsregler ein analoger PI-Regler entworfen. Bei der digitalen Regelung muss der Digitalrechner, der in endlicher Geschwindigkeit binär vorliegende Daten verarbeiten kann, die analogen Messwerte der Regelstrecke zu bestimmtenm Zeitpunkten abtasten und digitalisieren (vgl. Abb. 5.6). Um dies zu Erreichen, muss das analoge zeitkontinuierliche System in ein digitales zeitdiskretes System transformiert werden. Im Gegensatz zum zeitkontinuierlichen System, bei dem das Übertragungsverhalten anhand von Differentialgleichungen beschrieben wird, wird bei einem zeitdiskreten System der Zusammenhang von Ein- und Ausgangsgröße mit Hilfe von Differenzengleichungen beschrieben.

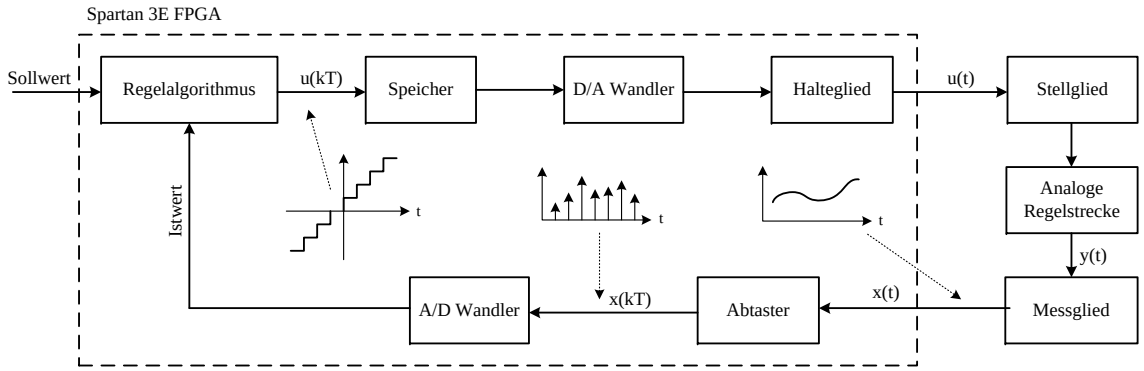

Abbildung 5.6.: Aufbau eines digitalen Regelkreises [14]

- **D/A und A/D Wandlung**
  Die abgetasteten Werte werden mit dem A/D Wandler in eine digitale binäre Zahl mit einer festen Anzahl von Stellen gewandelt. Durch diese endliche Wortlänge entstehen Quantisierungsfehler, die in einem digitalen System nicht vermieden werden können. Der Regelalgorithmus verarbeitet dieses digitale Signal und erzeugt das Stellsignal, dass durch den D/A Wandler in eine analoge Spannung zurück gewandelt wird.

- **Abtaster**
  Der Abtaster wird genutzt, um aus einem analogen kontinuierlichen Signal zu festen Abtastzeitpunkte zeitdiskrete Werte zu entnehmen. Bei der Geschwindigkeitsregelung wird dieser Abtaster mit Hilfe eines Timers realisiert. Dieser gibt die Zeitpunkte für die Berechnung der Stellgröße vor.

- **Halteglied**
  Das Halteglied dient dazu, dass das Stellsignal während einer Abtastperiode konstant anliegt.

5.4.1. Abtastvorgang

Bei einem kontinuierlichem System kann dem Signal zu jedem Zeitpunkt t in einem bestimmten Intervall ein eindeutiger Wert zugewiesen werden. Bei der digitalen Regelung können nur zeitdiskrete Signale verarbeitet werden, da der Digitalrechner zusätzlich Rechenzeit für andere Berechnungen benötigt. Deshalb wird das kontinuierliche Signal zu konstanten diskreten Zeitpunkten T abgetastet. Dies geschieht ähnlich wie ein Schalter, der zu bestimmten Zeitpunkten das kontinuierliche Signal liest und speichert.

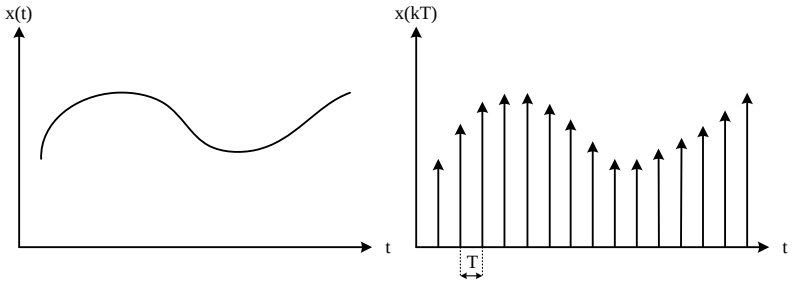

Abbildung 5.7.: Diskretisierung eines kontinuierlichen Signals

Das Halteglied hinter dem PI-Regler sorgt dafür, dass das zeitdiskrete Stellsignal konstant über eine Abtastperiode T an der Regelstrecke anliegt. In einem Digitalspeicher wird dieses Signal bis zur nächsten Abtastung abgelegt. Durch die Zahlenfolge $1T, 2T, 3T, ...$ entsteht eine Treppenfunktion, deren Werte bis auf die Quantisierungsfehler den Werten der Messwerte entsprechen.

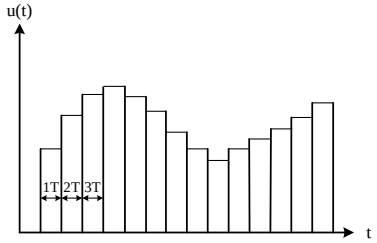

Abbildung 5.8.: Treppenfunktion eines zeitdiskreten Signals

Die Größe der Abtastperiode T ist abhängig von den Zeitkonstanten T_1 der Regelstrecke und den Randbedingungen wie Anforderungen an Genauigkeit und Rechenkapazität. Bei der Geschwindigkeitsregelung darf T nicht zu groß sein, da sonst starke Änderungen der Geschwindigkeit nicht schnell genug bemerkt werden können. Als Faustformel für die Wahl der Abtastperiode gilt [5]:

$$T \leq 0.1 * T_1 \Rightarrow T \approx \frac{T_1}{10} \tag{5.15}$$

Für die Abtastperiode T der Geschwindigkeitsregelung ergeben sich mit den berechneten Zeitkonstanten der Regelstrecke (vgl. Abb. 5.2) folgende Zeiten:

$$T_{1.Gang} = \frac{T_1}{10} = \frac{116,94ms}{10} \approx 11ms \tag{5.16}$$

$$T_{2.Gang} = \frac{T_1}{10} = \frac{80,920ms}{10} \approx 8ms \tag{5.17}$$

$$T_{3.Gang} = \frac{T_1}{10} = \frac{245,16ms}{10} \approx 24ms \tag{5.18}$$

Diese Werte wären die optimalen Abtastperioden für die Regelung. Da der Motor jedoch über einen PWM gesteuerten Fahrtensteller, der mit einer konstanten PWM Periode betrieben wird, angesteuert wird, können diese Abtastraten nicht eingehalten werden. Die PWM Periode des Fahrtenstellers beträgt $20ms$. Dies bedeutet, dass der Motor nur alle $20ms$ mit der neuen Stellgröße u(t) angetrieben werden kann. Daraus folgt, dass für den ersten und zweiten Gang weniger Messpunkte bis zum Erreichen des stationären Endwerts vorhanden sind und somit die Stellgröße seltener berechnet wird.

	Messpunkte $T_1/20ms$
1.Gang	5
2.Gang	4
3.Gang	12

Tabelle 5.3.: Anzahl der Messpunkte bis zum Erreichen von T_1

5.5. Diskretisierung des PI-Reglers

Zur Realisierung des digitalen PI-Reglers muss die zeitkontinuierliche Übertragungsfunktion $G_R(s)$ des analogen Reglers (vgl. Gleichung 5.4) in eine zeitdiskrete Übertragungsfunktion $G_R(z)$ transformiert werden. Zeitkontinuierliche Systeme werden in der s-Ebene beschrieben und durch die Laplace Transformation analysiert. Bei zeitdiskreten Systemen erfolgt die Analyse mit Hilfe der z-Transformation in der z-Ebene. Der Zusammenhang der z-transformierten Ausgangsgröße $X_a(z)$ zur z-transformierten Eingangsgröße $X_e(z)$ wird definiert durch die allgemeine z-Übertragungsfunktion $G_R(z)$ [19]:

$$G(z) = \frac{X_a(z)}{X_e(z)} = \frac{b_0 + b_1 z^{-1} + ... + b_m z^{-m}}{a_0 + a_1 z^{-1} + ... + a_n z^{-n}} \tag{5.19}$$

Mit Einsatz von verschiedenen Verfahren lässt sich ein System von einer zeitkontinuierlichen Form in eine zeitdiskrete Form transformieren. Die tranformierte Rekursionsgleichung oder auch Differenzengleichung kann nach der Transformation in einem Digitalrechner implementiert werden [14].

5.5.1. Z-Transformation

Die Transformation von der s-Ebene in die z-Ebene erfordert eine Äquivalenzbeziehung, die den Zusammenhang zwischen s und z darstellt. Die exakte Z-Transformation verwendet die Beziehung:

$$z = e^{T*s} \quad \Rightarrow \quad s = \frac{1}{T} ln(z) \tag{5.20}$$

Für den Entwurf des Geschwindigkeitsreglers wurden zwei Entwurfsverfahren angewandt, die jeweils unterschiedliche Beziehungen haben. Beide Methoden beschreiben ein mathematisches Verfahren, wie man das Integral der kontinuierlichen Übertragungsfunktion in der Zeit T numerisch annähern kann und somit die Fläche unter der Kurve bestimmen kann.

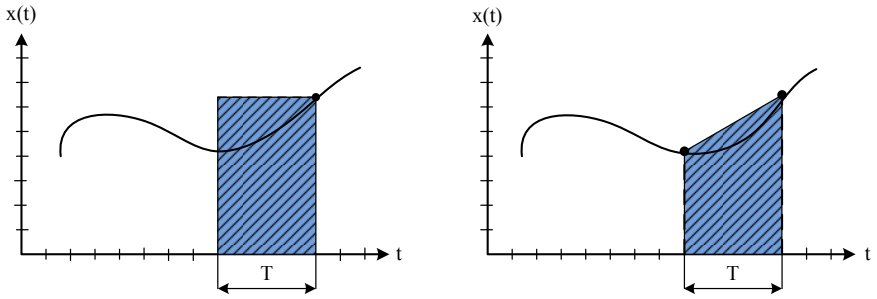

Abbildung 5.9.: Integration durch rechte Rechteck- und Trapezregel

- **Integration mit Euler Verfahren**
  Das Euler Verfahren hat für die numerische Integration der Differentialgleichung sowohl eine Äquivalenzbeziehung für den Rückwärtsdifferenzenquotient als auch für den Vorwärtsdifferenzenquotient. Je nach Wahl wird die Rechtecksfläche entweder vom rechten Rand oder vom linken Rand der Kurve berechnet (vgl. Abb. 5.9 links). Die jeweilige Rechteckregel lautet [14]:

  $$\textit{Rückwärts} : s = \frac{z-1}{T*z} \qquad \textit{Vorwärts} : s = \frac{z-1}{T} \tag{5.21}$$

  Die Rechteckregel stellt nur eine grobe Näherung des kontinuierlichen Systems dar und wird nur bei sehr kleinen Abtastraten empfohlen .

- **Bilineare Transformation mit Tustin Formel**
  Eine genauere Approximation bietet die Integration nach der Trapezregel, die sogenannte Bilineare Transformation oder auch Tustin Formel (vgl. Abb. 5.9 rechts). Diese ordnet jedem Punkt in der s-Ebene eindeutig einen Punkt in der komplexen z-Ebene zu. Aus der Entwicklung der Potenzreihe von $ln(z)$ und dem Abbruch nach dem ersten Glied erhält man die Beziehung [14]:

  $$s = \frac{2}{T} * \frac{z-1}{z+1} \tag{5.22}$$

  Die Bilineare Transformation hat den Vorteil, dass sie die Fläche unter der Kurve besser bestimmen kann. [9].

Der analoge Geschwindigkeitsregler mit der Übertragungsfunktion $G_R(s)$ kann nun mit den zuvor genannten Transformationsbeziehungen digitalisiert werden (vgl. 5.21 und 5.22). Die Berechnung des I-Anteils und der diskreten Übertragungsfunktion $G_R(z)$ für den PI-Regler wird exemplarisch mit der Bilinearen Transformation durchgeführt. Durch Einsetzen der Äquivalenzbeziehung in die kontinuierliche Übertragungsfunktion 5.4 erhält man die Z-Übertragungsfunktion $G_R(z)$:

$$\begin{aligned} G_R(z) &= K_R + \frac{K_R}{T_I * \frac{2}{T}\frac{z-1}{z+1}} \\ &= K_R * \left(\frac{2T_I * (z-1) + T * (z+1)}{2T_I * (z-1)}\right) * \left(\frac{z^{-1}}{z^{-1}}\right) \end{aligned} \tag{5.23}$$

Als nächstes erweitert man die Gleichung mit der Zustandsvariablen z^{-1}. Diese stellt in einem digitalen System einen Zustandsspeicher dar, also den Bezug auf die Vergangenheit des Wertes.

$$G_R(z) = K_R * \left(\frac{2T_I - 2T_I z^{-1} + T + Tz^{-1}}{2T_I - 2T_I z^{-1}}\right) = \frac{u(z)}{e(z)}$$

Multipliziert man nun den Ausgangswert $u(z)$ des Reglers mit dem Nenner und den Eingangswert $e(z)$ mit dem Zähler, dann erhält man die zugehörige Differenzengleichung für die Stellgröße $u(k)$:

$$\begin{aligned} u(z) * \underbrace{(2T_I)}_{a_0} - u(z) * \underbrace{(2T_I z^{-1})}_{a_1} &= K_R e(z) * \underbrace{(2T_I + T)}_{b_0} - K_R e(z) * \underbrace{(2T_I z^{-1} - Tz^{-1})}_{b_1} \\ a_0 u(k) - a_1 u(k-1) &= b_0 K_R e(k) - b_1 K_R e(k-1) \\ u(k) &= \frac{b_0 K_R}{a_0} e(k) - \frac{b_1 K_R}{a_0} e(k-1) + \frac{a_1}{a_0} u(k-1) \end{aligned} \tag{5.24}$$

Diese Gleichung des Geschwindigkeitsregelalgorithmus enthält die Parameter $u(k-1)$ und $e(k-1)$. Sie stellen die jeweils in der Vergangenheit liegenden Werte für die Stellgröße u und die Regelabweichung e dar. Abb. 5.10 verdeutlicht anhand eines Blockschaltbildes die Struktur des Reglers. Die Koeffizienten a_0, a_1, b_0 und b_1 sind alle durch die zuvor bestimmten Parametern berechenbar.

$$a_0 = a_1 = 2T_I \qquad b_0 = 2T_I + T \qquad b_1 = 2T_I - T \tag{5.25}$$

Für die Berechnung der Einstellwerte des Geschwindigkeitsreglers wurden sowohl die Parameter K_{R1} und T_{I1} als auch K_{R2} und T_{I2} berücksichtigt. Da die Abtastperiode T ebenso in die Berechnung mit einfließt, wird auch der Fall der Stellgliedseinschränkung durch die konstante PWM Periode mit beachtet.

Parameterbestimmung nach Trapezregel							
		1.Gang		2.Gang		3.Gang	
		$T = 11ms$	$T = 20ms$	$T = 8ms$	$T = 20ms$	$T = 24ms$	$T = 20ms$
		p_1	p_2	p_1	p_2	p_1	p_2
e(k)	K_{R1}/T_{I1}	0,70594	0,71926	0,38963	0,40372	0,26960	0,26853
	K_{R2}/T_{I2}	0,36654	0,38430	0,20264	0,22143	0,14017	0,13874
e(k-1)	K_{R1}/T_{I1}	0,67338	0,66006	0,37083	0,35674	0,25672	0,25780
	K_{R2}/T_{I2}	0,32312	0,30536	0,17758	0,15879	0,12299	0,12442

Tabelle 5.4.: PI-Reglerparameter nach Bilinearer Transformation

Für den Geschwindigkeitsregler kann folgende Differenzengleichung aufgestellt werden (vgl. Gleichung 5.24).

$$u(k) = p_1 * e(k) - p_2 * e(k-1) + u(k-1) \tag{5.26}$$

Bei der VHDL Implementierung dieser Gleichung werden 2 Addierer und 2 Multiplizierer inferriert. Abb. 5.10 zeigt zwei in Reihe geschaltete Verzögerungsglieder erster Ordnung mit einem Delayregister $e(k-1)$ zur Verzögerung der Regelabweichung und einem Rückführungsregister $u(k-1)$ für die Stellgröße. Sowohl der Eingang $e(k)$ als auch der Ausgang $u(k)$ werden zur Entkopplung von anderen Stufen in Registern gespeichert.

Der längste Signallaufzeitpfad, bestehend aus einer Multiplikation, einer Subtraktion und einer Addition (vgl. Abb. 5.10), beträgt ca. 24 ns und ist deutlich größer als die Taktperiode von 20 ns. In Kapitel 6.1.2 wird die Modifikation dieses Signalflussgraphen vorgestellt und die konkrete Implementierung erläutert.

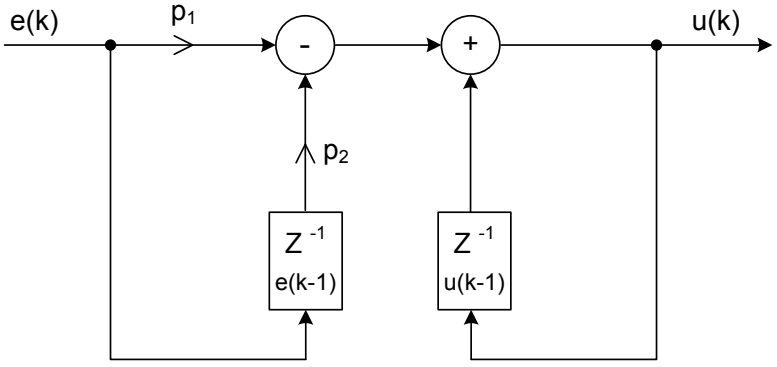

Abbildung 5.10.: Signalflussgraph des Geschwindigkeitsreglers mit Delaypfad und Akkumulatorrückführung

Mit Einsatz der rechtsseitigen Rechteckregel anstatt der Trapezregel ändern sich nur die Einstellwerte p_1 und p_2 des Reglers. Die Struktur der Gleichung bleibt gleich.

$$u(k) = \frac{K_R b_0}{a_0} e(k) - \frac{K_R b_1}{a_0} e(k-1) + \frac{a_1}{a_0} u(k-1) \tag{5.27}$$

$$a_0 = a_1 = b_1 = T_I \qquad b_0 = T_I + T \tag{5.28}$$

Nachfolgend werden die berechneten Werte in Tabelle 5.5 gezeigt. Die vollständige Berechnung und die zugehörige Z-Transformation wird hier nicht weiter erläutert.

Parameterbestimmung nach Rechteckregel							
		1.Gang		2.Gang		3.Gang	
		$T = 11ms$	$T = 20ms$	$T = 8ms$	$T = 20ms$	$T = 24ms$	$T = 20ms$
		p_1	p_2	p_1	p_2	p_1	p_2
e(k)	K_{R1}/T_{I1}	0,72222	0,74886	0,39903	0,42722	0,27604	0,27390
	K_{R2}/T_{I2}	0,38824	0,42377	0,21517	0,25276	0,14875	0,14589
e(k-1)	K_{R1}/T_{I1}	0,68966	0,68966	0,38023	0,38023	0,26316	0,26316
	K_{R2}/T_{I2}	0,34483	0,34483	0,19011	0,19011	0,13158	0,13158

Tabelle 5.5.: PI-Reglerparameter nach Integration mit Rechteckregel

5.6. Diskretisierung der Regelstrecke

Zum Überprüfen des Regelalgorithmuses wird der geschlossene Regelkreise simuliert. Hierfür wird die Übertragungsfunktion des PT-1 Gliedes (vgl. Gleichung 5.2) ebenfalls mit der Bilinearen Tranformation in eine z-Übertragungsfunktion transformiert. Die Differenzengleichung des zeitdiskreten PT-1 Gliedes lautet:

$$y(k) = \frac{K_S}{a_0} u(k) + \frac{K_S}{a_0} u(k-1) - \frac{a_1}{a_0} y(k-1) \tag{5.29}$$

$$a_0 = 1 + \frac{2T_1}{T} \qquad a_1 = 1 - \frac{2T_1}{T} \tag{5.30}$$

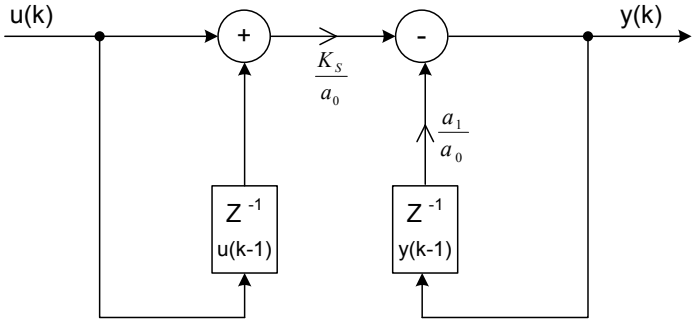

Abbildung 5.11.: Signalflussgraph für die Fahrzeugregelstrecke

6. Modellierung und Implementierung

In diesem Kapitel wird die Entwicklung des V-Regler IP-Cores und die Integration in das MicroBlaze Bussystem dargestellt (vgl. Abb. 2.1). Das SoC System wurde mit dem „Xilinx Embedded Development Kit EDK“ [25] entworfen und mit dem „Integrated Software Environment ISE“ [29] wurde aus dem VHDL Code ein FPGA Design erstellt. Für die Modellierung werden die in Kapitel 3 und 4 vorgestellten Designmethoden verwendet.

6.1. Top Entity des Geschwindigkeitsreglers

Der Geschwindigkeitsregler besteht aus drei hierarchisch strukturierten Komponenten, die über Schnittstellensignale miteinander kommunizieren. In einer Top Entity *V_Regler_Top_Entity* (vgl. Abb. 6.1) werden die einzelnen Komponenten zusammengefasst und bei der Instanziierung werden die Ein- und Ausgangssignale verdrahtet. Die externen Signale *Pulse_A_Top* und *Pulse_B_Top* werden an die Pulskanälen des Inkrementalgebers angeschlossen. Der Fahrtensteller wird über das Signal *PWM_Top* angesteuert.

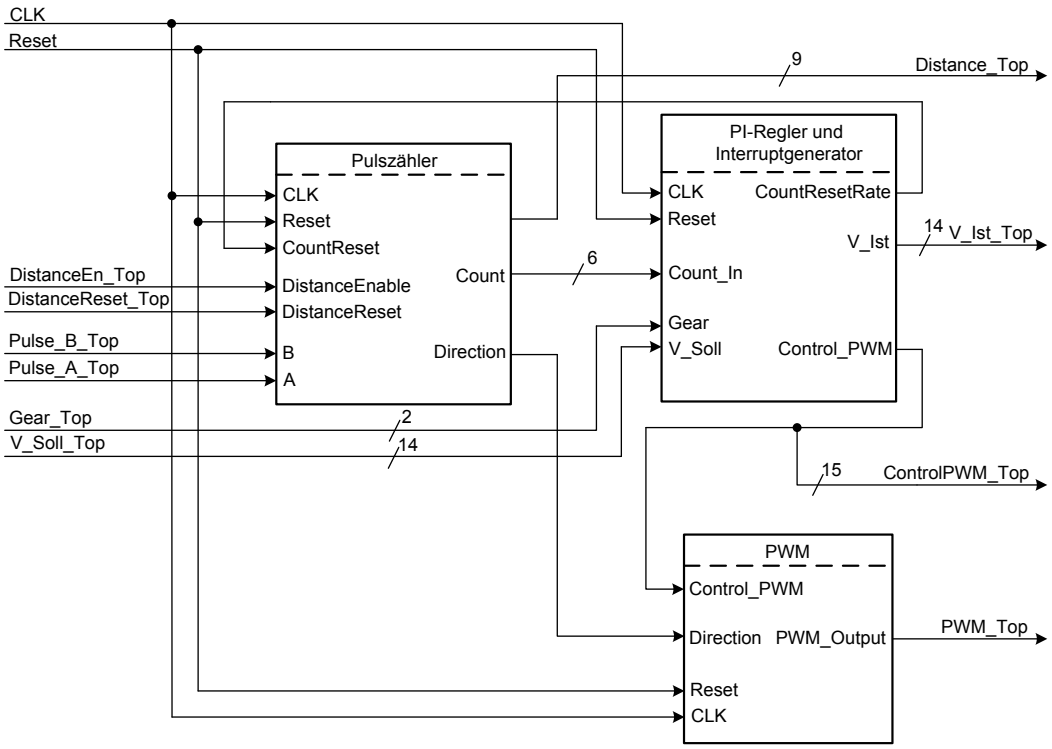

Abbildung 6.1.: Top Entity *V_Regler_Top_Entity* des Geschwindigkeitsregler IPs

Signalname	Beschreibung
CLK	Taktsignal für den Systemtakt von 50 MHz
Reset	Globaler Reset zum Zurücksetzen aller Module
DistanceEn_Top	Freigabe des *DistanceCounter* zum Zählen der Pulse parallel zum *PulseCounter* durch den Fahrzeugmodell IP
DistanceReset_Top	Zurücksetzen des *DistanceCounter* nach Bemessung der Parklücke durch Fahrzeugmodell IP
Gear_Top	Vorgabe des gewünschten Ganges vom MicroBlaze an PI Regler zum Berechnen der Stellgröße
V_Soll_Top	Vorgabe der Soll Geschwindigkeit vom MicroBlaze an PI Regler zur Berechnung der Regelabweichung
V_Ist_Top	Übergabe der berechneten Ist Geschwindigkeit an Fahrzeugmodell IP und an MicroBlaze für den Datenlogger
Direction	Übergabe der Drehrichtung an PWM Modul zur Erzeugung des entsprechenden PWM Signals
Distance_Top	Übergabe der gezählten Pulse nach der Bemessung einer Parklücke an Fahrzeugmodell IP
ControlPWM_Top	Weitergabe der Stellgröße an PWM Modul und MicroBlaze zur Erzeugung eines PWM Signal und zur Aufzeichnung der Daten durch Datenlogger
Count	Weitergabe des *PulseCounter* Zählerstandes an PI Regler zur Berechnung der Ist Geschwindigkeit
CountResetRate	Zurücksetzen des *PulseCounter* nach Ablauf einer Abtastperiode und Erzeugung eines Interrupts für die globale Zeit durch den PI Regler.

Tabelle 6.1.: Schnittstellensignale der Top Entity

6.1.1. Pulszähler Modul

Der Pulszähler zählt innerhalb der Abtastperiode *T*, die vom Inkrementalgeber erzeugten Pulse und bestimmt die Drehrichtung für das PWM Modul. Die Auswertung der um 90° versetzten Pulskanäle und die Steuerung der Counter werden in einer FSM (vgl. Abschnitt 4.1) mit fünf Zuständen modelliert. Die FSM tastet bei einer Frequenz von 50 MHz die Pulskanäle *Pulse_A* und *Pulse_B* ab und erzeugt das Freigabesignal *CountEnable* für den *PulseCounter*. Dieses wird gesetzt, wenn *Pulse A* und *Pulse B* einen logischen *high* Pegel besitzen und einer der beiden Kanäle auf *low* geht (vgl. Abb. 2). Die FSM übernimmt die Aufgabe des Steuerpfades indem sie über *CountEnable* die Freigabe für die Counter erzeugt.
Der *PulseCounter* inkrementiert seinen Zählerstand *Count* taktsynchron und bei Freigabe *CountEnable* durch die FSM. Anschließend wird *Count*, bei Ablauf der Abtastperiode, an den PI-Regler übergeben und über *CountReset* vom PI-Regler zurückgesetzt (vgl. Abb. 6.2).

Für die Bemessung der Parklücke wird innerhalb des Einparkassistenten ein zusätzlicher 9 Bit *DistCounter* eingesetzt. Zu Beginn des Einparkmanövers wird das Steuersignal *DistanceEnable* vom Fahrzeugmodell IP gesetzt und sobald die FSM einen Puls erkannt hat, wird der Counter parallel zum *PulseCounter* inkrementiert. Bei Erreichen des Parklückenendes wird *DistanceEnable* vom Fahrzeugmodell IP zurück genommen und der Zählerstand durch *DistanceReset* zurückgesetzt. Es wird davon ausgegangen, dass die Parklücke maximal einen Meter lang ist.

$$DistCounter = \frac{1\ m}{StreckeproPuls} = \frac{1\ m}{0,002175\ m} \approx 459\ Pulse \Rightarrow 9\ Bit \qquad (6.1)$$

Die gefahrene Strecke kann vom Fahrzeugmodell IP mit den gezählten Pulsen *Distance* berechnet werden (vgl. Abb. 6.2). Bei einer Größe von 9 Bit kann der *DistCounter* 511 Pulse zählen, dies entspricht einer maximalen Strecke von 1,11 m. Es muss beachtet werden, dass die gemessene Strecke in einer Anzahl von Pulsen vorliegt und gegebenenfalls noch mit der Strecke pro Puls multipliziert werden muss, um so die tatsächliche Strecke in Metern zu bekommen.

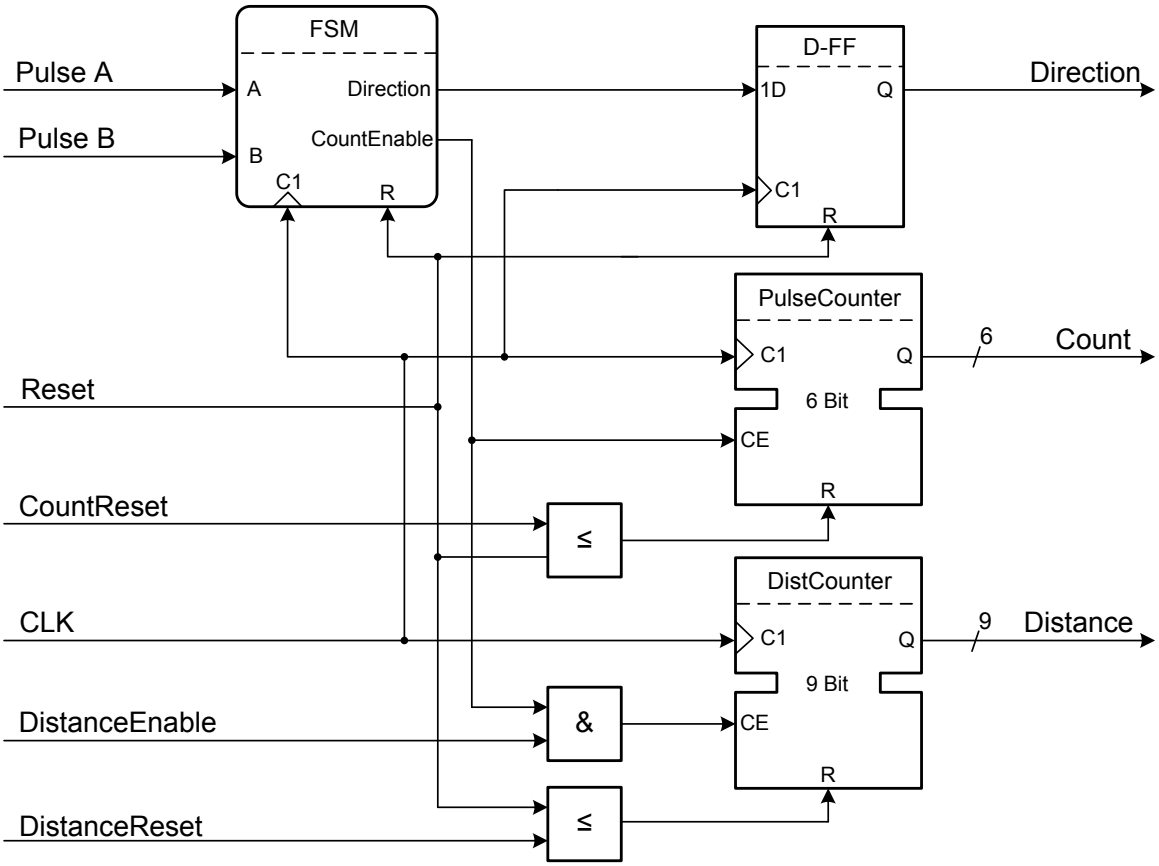

Abbildung 6.2.: Pulszähler mit FSM Steuerung für *PulseCounter* und *DistCounter*

Die Bitbreite des *PulseCounter* ergibt sich aus der maximalen Anzahl an Pulsen bei maximaler Geschwindigkeit im dritten Gang pro Abtastperiode T. Die maximale Pulsanzahl und die daraus folgende Bitbreite des Counters wird mit der Geschwindigkeit $V_{Max} \approx 4m/s$ und der Strecke pro Puls $s_p = 2,175mm$ berechnet:

$$V = \frac{s}{T} \Rightarrow s = V * T \Rightarrow s = 4\ m/s * 0,02\ s = 80\ mm \tag{6.2}$$

$$PulseCounter = \frac{s}{s_p} = \frac{80\ mm}{2,175\ mm} = 36,78 \Rightarrow 6\ Bit \tag{6.3}$$

Bei einer Größe von 6 Bit kann der *PulseCounter* innerhalb von T maximal 63 Pulse zählen bevor er einen Overflow erzeugt und der Zählerstand ungültig wird. Für das Carolo Cup Fahrzeug ist dies aber ausreichend, da bei maximaler Geschwindigkeit maximal 40 Pulse erzeugt werden.

Modelliert werden der *PulseCounter* und der *DistCounter* in einem gemeinsamen Prozess mit einem synchronen Reset. Die aktuellen Zählerstände *count_sig* und *distance_count_sig* werden in einem 6 Bit bzw. 9 Bit großen Register gespeichert. Am Ende des Prozesses wird der Zählerstand des Pulszählers über den Ausgang *Count* an das PI-Regler Modul übermittelt (vgl. Code 6.1). Die gemessene Strecke in Pulsen wird über *Distance* in ein Software Register geschrieben und kann von dort aus für weitere Berechnungen gelesen werden.

```
0  Count_P: process(CLK)
   begin
       if(CLK'event and CLK = '1') then
           if(Reset = '1') then
               count_sig          <= (others => '0');
5              distance_count_sig  <= (others => '0');
           elsif(CountReset = '1') then
               count_sig       <= (others => '0');
           end if;
           if(DistanceReset = '1') then
10             distance_count_sig  <= (others => '0');
           elsif(count_enable = '1') then
               count_sig <= count_sig + 1;
               if(DistanceEnable = '1') then
                   distance_count_sig  <= distance_count_sig + 1;
15             end if;
           end if;
       end if;
   end process Count_P;

20 Count    <= count_sig;           -- lokales Count Signal wird an das PI-Regler Modul übertragen
   Distance <= distance_count_sig;  -- die gemessene Strecke in Pulsen wird auf den Ausgang gelegt
```

Listing 6.1: VHDL Implementierung zweier Counter zum Zählen der Pulse

Der *DistCounter* arbeitet unabhängig vom *PulseCounter* und kann nur durch einen globalen Reset oder einen explizit gewünschten *DistanceReset* zurückgesetzt werden. Somit ist gewährleistet, dass während einer Parklücken Bemessung, die länger als 20 ms dauert, der Zählerstand des *DistCounter* nicht nach Ablauf einer Abtastperiode durch *CountReset* zurückgesetzt wird. Der integrierte Abtastratentimer im PI-Regler Modul setzt ausschließlich über *CountReset* den *PulseCounter* für die Geschwindigkeitsregelung zurück (vgl. Abb. 6.2).

Die Decodierung der Eingangssignale *Pulse A* und *Pulse B* erfolgt über eine FSM, die die Pulskanäle abtastet und auswertet. Zunächst werden die Eingangssignale synchronisiert und die internen synchronisierten Signale *pulse_a_sig* und *pulse_b_sig* dienen als Eingangssignale der FSM (vgl. Code 6.2). Der endliche Automat besteht aus einem Quintupel an Zuständen:

$$Z = \{Idle, A\_One, B\_One, AB\_One, AB\_Null\} \tag{6.4}$$

Die Zustände ergeben sich aus den Pegeln der Pulskanäle und setzen sich aus der Konkadination von *pulse_a_sig* und *pulse_b_sig* zusammen. Zur besseren Veranschaulichung werden sie mit einer symbolischen Zustandsbeschreibung beschrieben (vgl. Abb. 6.3).

Listing 6.2: Eingangsynchronisation von Pulse A und Pulse B

```
0  Input_Synch: process(CLK)
   begin
       if(CLK'event and CLK = '1') then
           if(Reset = '1') then
               pulse_a_sig <= '0';
5              pulse_b_sig <= '0';
           else
               pulse_a_sig <= A;       -- Eingangssynchronisation von Puls A
               pulse_b_sig <= B;       -- Eingangssynchronisation von Puls B
           end if;
10     end if;
   end process Input_Synch;
```

Der Inkrementalgeber übermittelt seine Zustände im Graycode. Dies bedeutet, dass sich die direkt benachbarte Binärwerte in nur einem Bit unterscheiden. Ein Zustandsübergang von $AB\_Null \rightarrow AB\_One$ oder $A\_One \rightarrow B\_One$ ist beispielsweise nicht möglich und wird in Abb. 6.3 nicht dargestellt. Erkennt die FSM das der aktuelle Zustand gleich dem alten Zustand ist, dann verbleibt sie im aktuellen Zustand bis einer der Pulskanäle sich ändert. Bei einem Zustandswechsel werden die Ausgänge der FSM *CountEnable* und *Direction* gesetzt.

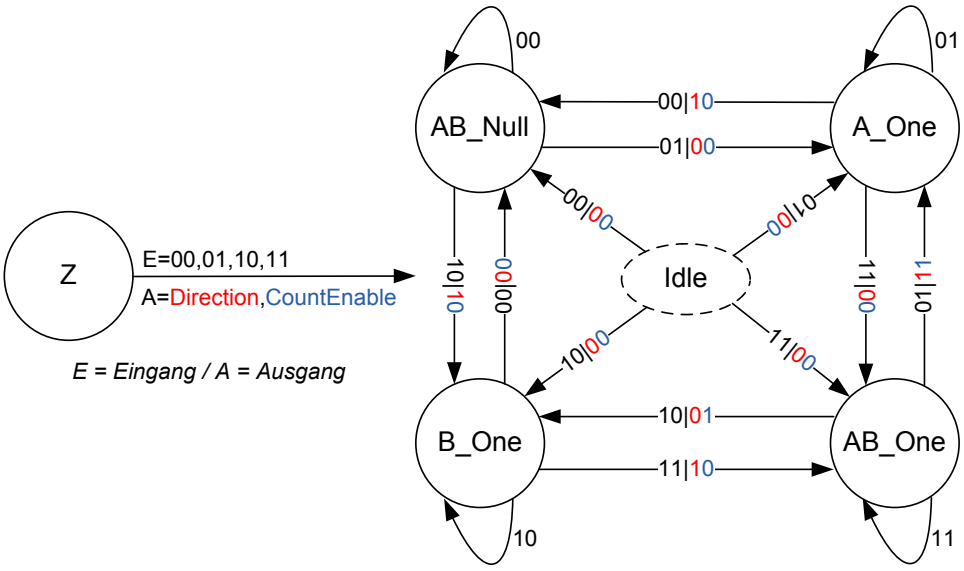

Abbildung 6.3.: Zustandsdiagramm der Pulszählermoduls FSM

Bei Start und Reset beginnt der Automat im *Idle* Zustand, da zu Beginn einer Messung nicht feststellbar ist, wie die Drehscheibe des Inkrementalgebers steht. Der *Idle* Zustand erzeugt keinerlei Ausgaben, sondern bestimmt nur den Folgezustand *next_state*. Dieser wird von kombinatorischer Logik bestimmt und in einem getakteten Prozess gespeichert. Der kombinatorische Prozess enthält sowohl das Ausgangsschaltnetz als auch das Übergangsschaltnetz der FSM. Solch eine Implementation wird Huffman-Normalform genannt [7] und hat den Vorteil, dass kein zusätzlicher kombinatorischer Prozess, der die Ausgänge berechnet, benötigt wird. Die Drehrichtung wird bei einem Zustandsübergang direkt aus der FSM, über das Signal *Direction*, in einem Register gespeichert.

Bei der sequentiellen Zustandscodierung durch das Synthesewerkzeug werden den symbolisch beschriebenen Zuständen (*A_One*, *B_One*...) aufsteigende Bitkombinationen zugeordnet. Für die Codierung der fünf Zustände durch das Synthesewerkzeug werden 3 Bit benötigt. Die illegalen Zustandsübergänge, die aufgrund der Graycode Übermittlung nicht vorkommen können, sind in Code 6.3 trotzdem codiert. Bei Auswertung des Fehlers kann bei diesen Zustandsübergänge ein Errorflag gesetzt werden.

Listing 6.3: Übergangs- und Ausgangsschaltnetz der FSM

```
0  Direction_P: process(pulse_a_sig, pulse_b_sig, state)
   variable state_intern: std_logic_vector(1 downto 0); -- Zustandsvariable für Pulskanäle
   begin
       state_intern   :=  pulse_b_sig & pulse_a_sig;     -- Verknüpfen der synchronisierten Eingänge Puls A und Puls B
       direction_sig  <=  direction_reg;                 -- Standard Initialisierung für Direction
5      count_enable   <=  '0';                           -- Setzen des Standard Count Enable für den Pulszähler

       case state is
           when Idle    => case state_intern is                    -- Startzustand der FSM
                             when "00" => next_state <= AB_null;
10                           when "01" => next_state <= A_one;
                             when "10" => next_state <= B_one;
                             when "11" => next_state <= AB_one;
                             when others => next_state <= Idle;
                           end case;
15         when A_One   => case state_intern is
                             when "00" => next_state <= AB_null; direction_sig <= '1';
                             when "01" => next_state <= A_one;
                             when "10" => next_state <= B_one;       -- illegaler Zustandsübergang / Graycode
                             when "11" => next_state <= AB_one; direction_sig <= '0';
20                           when others => next_state <= Idle;
                           end case;
           when B_One   => case state_intern is
                             when "00" => next_state <= AB_null; direction_sig <= '0';
                             when "01" => next_state <= A_one;
25                           when "10" => next_state <= B_one;
                             when "11" => next_state <= AB_one; direction_sig <= '1';
                             when others => next_state <= Idle;
                           end case;
           when AB_One  => case state_intern is
30                           when "00" => next_state <= AB_null;
                             when "01" => next_state <= A_one; count_enable <= '1'; direction_sig <= '1';
                             when "10" => next_state <= B_one; count_enable <= '1'; direction_sig <= '0';
                             when "11" => next_state <= AB_one;
                             when others => next_state <= Idle;
35                         end case;
           when AB_Null => case state_intern is
                             when "00" => next_state <= AB_null;
                             when "01" => next_state <= A_one; direction_sig <= '0';
                             when "10" => next_state <= B_one; direction_sig <= '1';
40                           when "11" => next_state <= AB_one;     -- illegaler Zustandsübergang / Graycode
                             when others => next_state <= Idle;
                           end case;
           when others  => next_state <= Idle;
       end case;
45 end process Direction_P;
```

Die Abtastfrequenz muss so hoch sein, dass bei maximaler Geschwindigkeit zwischen zwei Zustandswechseln mindestens eine Abtastung stattgefunden hat. Bei maximaler Geschwindigkeit beträgt die Pulsbreite der Inkrementalgeberpulse ca. 643 μs (vgl. Kap. 7.2). Dies bedeutet, dass bei einer Taktperiode von 20 ns (bei 50 MHz) genügend Abtastungen erfolgt sind, um die Pulsfolge auszuwerten.

Zum Testen der Funktionalität des Pulszähler Moduls und für die Veranschaulichung des Timings wurde eine Pulsfolge bestehend aus jeweils 10 Pulsen modelliert. Bei einer Pegeländerung der Eingangssignale *Pulse A* oder *Pulse B* wird der aktuelle Zustand *state* berechnet und konstant in einem Register gespeichert, bis einer der Pulskanäle sich ändert oder ein globaler Reset auftritt.

Sobald *Pulse A* und *Pulse B* einen logischen *high* Pegel besitzen und einer der beiden Kanäle auf *low* geht, wird das Freigabesignal *count_enable* gesetzt und *Count* inkrementiert (vgl. Abb. 2). Nach 20 ms wird der *PulseCounter* vom PI-Regler Modul durch das Eingangssignal *CountReset* zurückgesetzt.
Zu Beginn der Simulation ist das Ausgangssignal *Direction* gleich 0 und signalisiert damit dem PWM Modul, dass das Fahrzeug vorwärts fährt. Nach 5 ms findet eine Drehrichtungsänderung statt, indem ein Zustandswechsel von $A\_One \rightarrow AB\_Null$ und nicht wie zuvor von $A\_One \rightarrow AB\_One$ stattfindet. Bei der nächsten Taktflanke wird *Direction = 1* und das PWM Modul kann eine entsprechende PWM erzeugen, sodass das Fahrzeug rückwärts fährt.

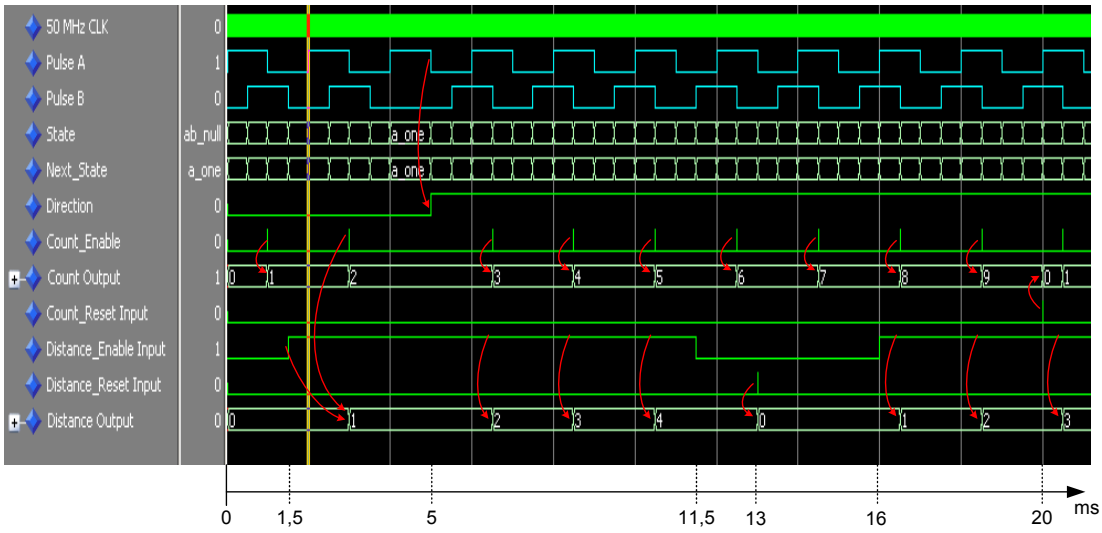

Abbildung 6.4.: Timingdiagramm des Pulszählers mit einer Pulsfolge bestehend aus 10 Pulsen ($V \approx 1\ m/s$) und einer Drehrichtungsänderung

Die Testbench setzt nach 1.5 ms das Freigabesignal *DistanceEnable* für den *DistCounter*. Ab diesem Zeitpunkt zählt der Counter für eine Zeit von 10 ms parallel zum *PulseCounter* bei *count_enable = 1* die Pulse bis entweder *Distance_Reset* gesetzt wird oder *DistanceEnable* auf *low* geht (vgl. Abb. 6.4). Das Enable Signal wird nach 11,5 ms vom Fahrzeugmodell IP des Einparkassistenten zurückgenommen. Die gezählten Pulse *Distance* werden in ein Software Register des MicroBlazes geschrieben.

Da die Software Register des MicroBlazes jeweils 32 Bit groß sind, teilen sich die Steuersignale *DistanceEnable* und *DistanceReset* gemeinsam mit dem Steuersignal für den Gang ein Register. Die zwei niederwertigen Bits sind für den jeweiligen Gang reserviert und werdem vom MicroBlaze beschrieben. An dritter und vierter Stelle stehen *DistanceReset* und *DistanceEnable*. Diese Bitstellen werden über die Adresse des Registers über den PLB Bus von einem weiteren integrierten IP Core beschrieben (vgl. Kapitel 6.2.

6.1.2. PI-Regler Modul mit integriertem Abtastratentimer

Das PI-Regler Modul implementiert die in Kapitel 5.5 hergeleitete Differenzengleichung des PI-Reglers (vgl. Gleichung 5.26). Diese wird in in mehrere Teilaufgaben zerlegt und in vier getrennt getakteten Prozessen bearbeitet (vgl. Abb. 6.5). Über Steuersignale, die im ersten Prozess vom Abtastratentimer erzeugt und verzögert werden, signaliseren sich die Prozesse gegenseitig die Fertigstellung ihrer Berechnung. Abb. 6.5 soll einen Überblick über den Ablauf des PI-Reglers geben und die in Abb. 6.6 dargestellte Pipeline verdeutlichen. Nach vier Takten steht die Stellgröße zur Weiterverarbeitung bereit.

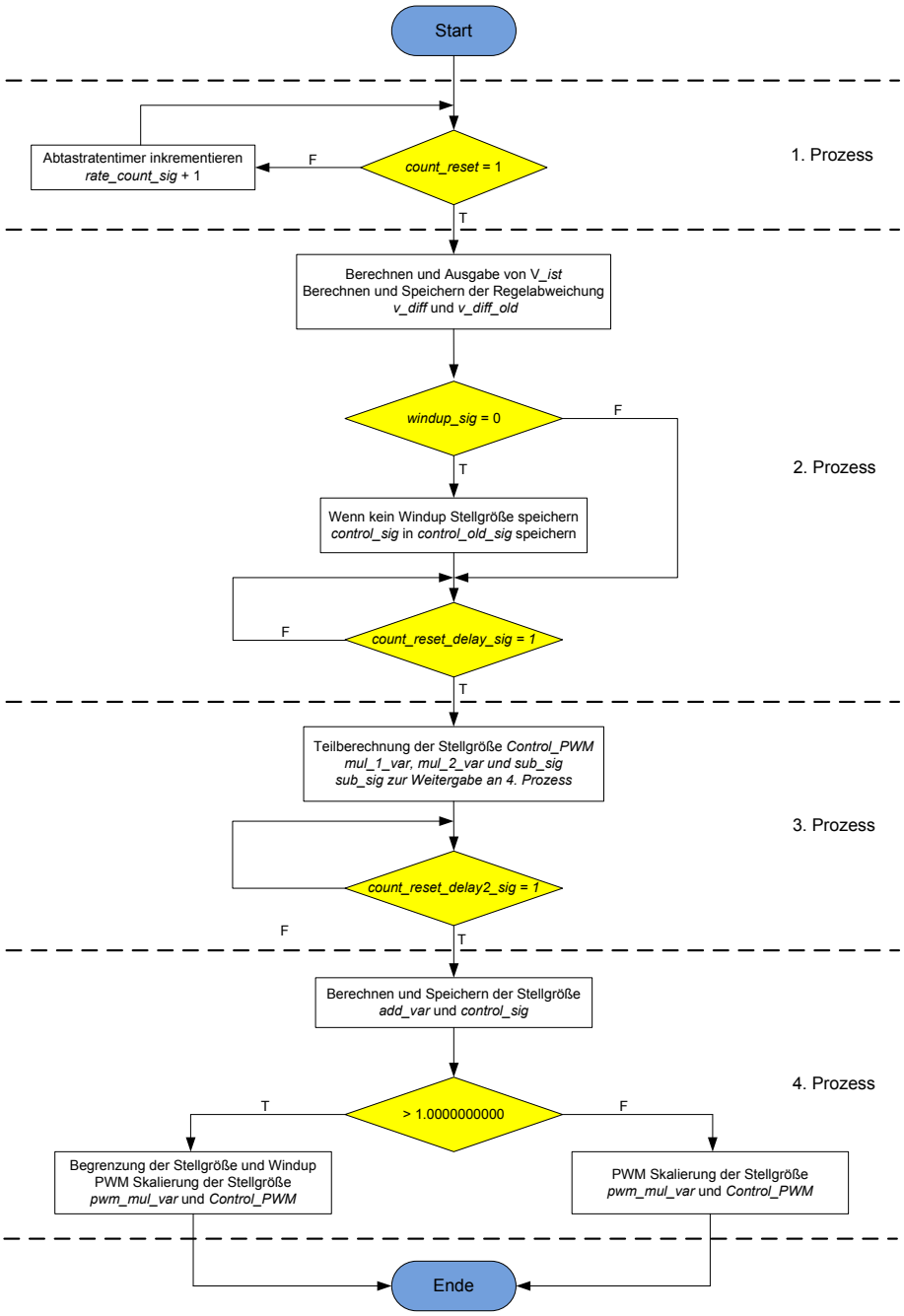

Abbildung 6.5.: Programmablaufdiagramm des PI-Regler Moduls

Die Berechnung der Stellgröße beinhaltet drei Multiplikationen, zwei Subtraktionen und eine Addition. Diese können bei einer Frequenz von 50 MHz nicht innerhalb eines Taktes berechnet werden. Der längste Signallaufzeitpfad wäre ca. 24 ns lang und somit wären die Timinganforderungen nicht erfüllt, da die Taktperiode 20 ns groß ist. Aus diesem Grund wird eine 3-stufige Pipeline implementiert, die den größten Signallaufzeitpfad verkürzt, indem sie die Zwischenergebnisse in Register speichert (vgl. Abb. 6.6).

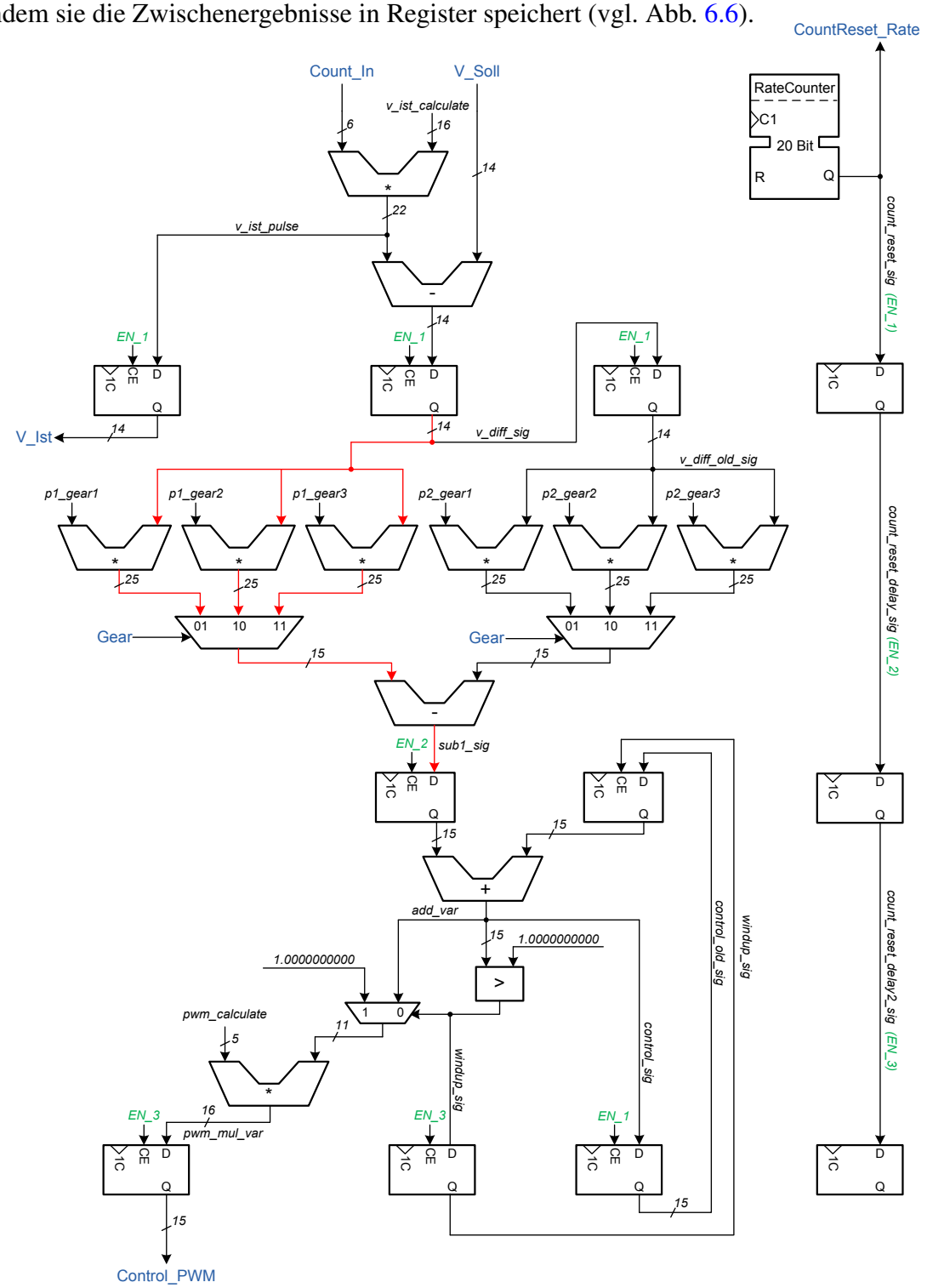

Abbildung 6.6.: 3-stufige Pipeline des PI-Reglers zur Berechnung der Stellgröße und zur Erzeugung der globalen Zeit durch Interruptgenerierung

Die rot markierte Linie in Abb. 6.6 signalisiert den längsten Signallaufzeitpfad der dreistufigen Pipeline des Geschwindigkeitsreglers. Die maximale Signallaufzeit beträgt 11,288 ns. Innerhalb dieser Zeit wird das Zwischenergebniss *sub_sig* nach einer Multiplikation und einer gesteuerten Subtraktion in der zweiten Pipelinestufe in einem Pipelineregister gespeichert. Die minimale Periode bestimmt die maximale Taktfrequenz:

$$f_{max} = \frac{1}{11,288ns} = 88,59\ MHz \tag{6.5}$$

Der Abtastratentimer *RateCounter* (Abb. 6.6 rechts) dient als Steuerpfad für die Pipeline. Er erzeugt die Freigabesignale für das Delayregister $e(k-1)$ und für die Pipeline Register, die für die Schnittstellensynchronisation zwischen zwei Pipelinestufen zuständig sind. Das initiale Steuersignal *count_reset_sig* wird vom *RateCounter* gesetzt und anschließend für die weiteren Pipelinestufen in Delayregistern verzögert (vgl. Abb. 6.6 rechts). *Count_reset_sig* wird über den Ausgang *Count_Reset_Rate* nach außen geführt und sowohl zum Zurücksetzen des Pulszählerstandes benutzt als auch zur Interruptgenerierung für die globale Zeit des Systems.

Aufgrund der realisierten Pipelinestruktur muss der Signalflussgraph aus Abb. 5.10, der direkt aus der Differenzengleichung gewonnen worden ist, mit einem zusätzlichen Trennregister *sub_sig* ergänzt werden (vgl. Abb. 6.7). Dieses dient zur Entkopplung der zweiten von der dritten Pipelinestufe. Der Komparator am Ende des Signalflussgraphes stoppt bei Überschreitung der Stellgrößenbegrenzung die Akkumulation der Stellgröße im Rückführungsregister $u(k-1)$.

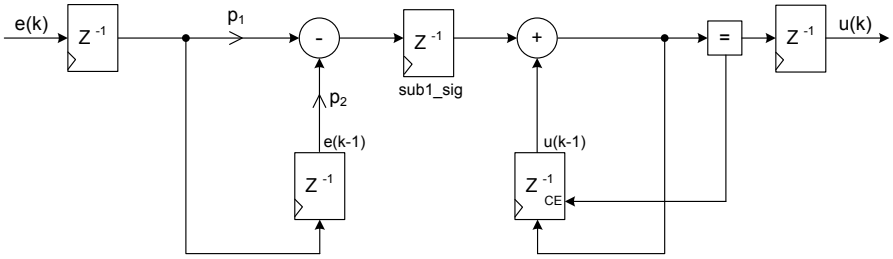

Abbildung 6.7.: Signalflussgraph des Geschwindigkeitsreglers mit zusätzlichem Trennregister zum Erfüllen der Timinganforderungen (vgl. Abb. 6.6)

Anti Windup Effekt

Bei der Geschwindigkeitsregelung ist die Stellgröße auf 1 begrenzt. Um zu Vermeiden, dass bei einem Erreichen der Stellgrößenbegrenzung der I-Anteil die Regelabweichung weiter aufintegriert, wird ein „Anti Windup“ realisiert. Wenn u(k) größer eins ist, wird das Steuersignal *windup_sig* gesetzt und das gesteuerte Rückkopplungsregister u(k-1) speichert die zu große Stellgröße *control_sig* nicht (vgl. Abb. 6.7). Die Begrenzung wird durch einen Komparator, der das Additionsergebnis in der Variablen *add_var* vergleicht, realisiert. Die zuvor berechnete Stellgröße, die kleiner eins war, bleibt in Register u(k-1) gespeichert. Damit ist gewährleistet, dass der PI-Regler nicht übersteuert und das Rückführungssignal *control_old_sig* so begrenzt ist, dass der Regelkreis stets stabil ist.

Der integrierte Abtastratentimer

Der Abtastratentimer steuert die Pipeline und ist als globaler Zeitgeber für die IP Blöcke und das Software Konzept des Einparkassistenten ausgelegt. Bei Erreichen der Abtastperiode wird ein Interrupt erzeugt und der Zählerstand des *PulseCounter* über *Count_Reset* zurückgesetzt. Wie in Abschnitt 5.4.1 beschrieben, können die berechneten Abtastraten bei der Implementierung nicht realisiert werden. Aus diesem Grund wird für jeden Gang des Fahrzeuges eine einheitliche Abtastperiode von 20 ms verwendet. Diese bestimmt durch Delayregister die zeitliche Abfolge der Berechnungen der folgenden Prozesse.

Der Abtastratentimer wird in einem getakteten Prozess modelliert und übernimmt die Aufgabe des Steuerpfades. Für die Implementierung der 20 ms Periode wird bei einer Systemfrequenz von 50 MHz ein 20 Bit großer taktflankengesteuerter Counter eingesetzt (vgl. Abb. 6.8). Dieser inkrementiert bei jeder positiven Taktflanke den Zählerstand *rate_count_sig*. Der maximale Zählerstand *rate_20ms* beträgt 1.000.000 Ticks und wird in einem VHDL Generic abgelegt (vgl. Code 6.4).

```
0  Periode_P: process(CLK)
   begin
       if(CLK'event and CLK = '1') then
           if(Reset = '1') then
               rate_count_sig <= (others => '0');
5          elsif(rate_count_sig = rate_20ms) then   -- rate_20ms = "11110100001001000000" = 1.000.000 Ticks
               count_reset_sig <= '1';
               rate_count_sig <= (others => '0');
           else
               rate_count_sig <= rate_count_sig + 1;
10             count_reset_sig <= '0';
           end if;
       end if;
   end process Periode_P;
```

Listing 6.4: Abtastratentimer als globaler Zeitgeber und zur Steuerung der Pipeline

Bei Erreichen von *rate_20ms* wird das Steuersignal *count_reset_sig* gespeichert. Für einen Takt lang ist *count_reset_sig* gesetzt und signalisiert der 1. Pipelinestufe, dass sie ihre Berechnung beginnen kann. Anschließend wird das Signal für die restlichen Pipelinestufen in Delayregister verzögert. Ein Reset des Zählerstandes erfolgt nur bei einem synchronen Systemreset oder bei Erreichen des Endwertes.

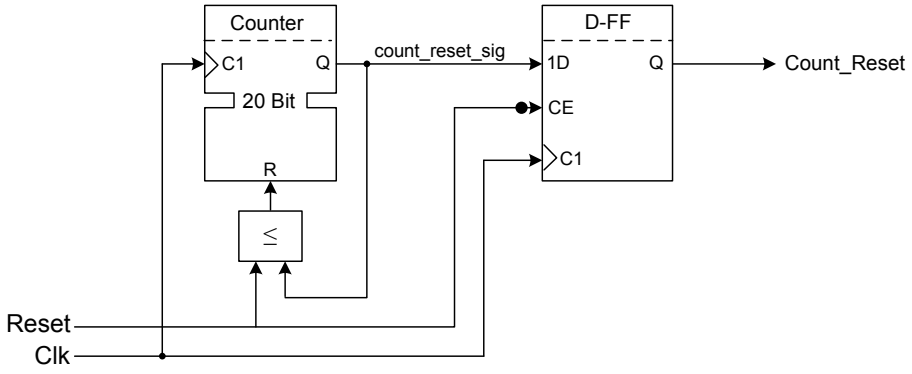

Abbildung 6.8.: 20 Bit großer Abtastratentimer zur Generierung der globalen Zeit

Da der Counter denselben Endwert wie der Counter für die PWM Periode hat, kann bei knapp werdenden FPGA Ressourcen die Funktion des Abtastratentimers in das PWM Modul ausgelagert werden (vgl. Abschnitt 6.1.3).

1. Pipelinestufe: Berechnung der Ist Geschwindigkeit und der Regelabweichung und Speicherung der Delay- und Rückkoppelsignale

In der ersten Pipelinestufe wird bei Freigabe des Abtastratentimers durch das Steuersignal *count_reset_sig*, die momentane Ist Geschwindigkeit *v_ist_pulse* berechnet und über ein Übergaberegister auf den Ausgang *V_Ist* gelegt. Der Vorteil der Speicherung des Signals ist die Ausgangsynchronisation von *v_ist_pulse*, die die Verarbeitungsstufen und IPs entkoppelt. *V_Ist* wird für die direkte Weitergabe an den Fahrzeugmodell IP verwendet und zur Speicherung in einem Software Register für die Datenlogger Funktion. Die momentane Ist Geschwindigkeit ergibt sich aus den gezählten Pulsen *Count_In* während einer Abtastperiode und der Strecke pro Puls s_p.

$$V\_Ist = \frac{Count\_In * s_p}{20\ ms} = Count\_In * \frac{2,175\ mm}{20\ ms} = Count\_In\ *\ 0,10875 \quad (6.6)$$

Da die Abtastperiode und die Strecke s_p konstant sind, wird der Quotient aus $\frac{s_p}{20\ ms}$ in einem VHDL Generic *vist_calculate_20ms* abgelegt. Somit wird nur ein Multiplizierer zur Berechnung von *V_Ist* benötigt.
Die gesamte Modellierung wird mit dem RTL Coding Style des Q-Formats realisiert. Da der Faktor 0,10875 relativ klein ist, wird ein Q-Format mit einem Guard Bit und 15 Fractionalbits gewählt. Der entsprechende Faktor im $Q_{1.15}$ Format wird wie folgt konvertiert (vgl. Abschnitt 4.4).

$$vist\_calculate\_20ms = 0,10875 * 2^{15} \approx 3563 \Rightarrow 0.000110111101011 \quad (6.7)$$

Mit 15 Fractionalbits wird ein maximaler Quantisierungsfehler von $2^{-(15+1)} = 0,000015259$ erreicht. Für die Berechnung von *V_Ist* ergibt sich daraus eine prozentuale Abweichung von 0,019 %.

Die Variable *v_ist_pulse* stellt das Ergebniss aus der Multiplikation von *vist_calculate_20ms* und *Count_In* für die Weiterberechnung der Regelabweichung bereit. Bei der Ausgangssynchronisation wird *V_Ist* von ursprünglich 20 Bit auf 14 Bit gekürzt (vgl. Code 6.5).

Listing 6.5: Erste Pipelinestufe zur Berechnung von e(k) und V_Ist

```
V_Ist_Difference_P: process(CLK)
variable v_ist_pulse     :     unsigned(21 downto 0);
begin
    if(CLK'event and CLK = '1') then
        if (Reset = '1') then
            v_diff_sig       <= (others => '0');
            v_diff_old_sig   <= (others => '0');
            control_old_sig  <= (others => '0');

        elsif(count_reset_sig = '1') then
            v_ist_pulse           :=  unsigned(Count_In) * vist_calculate_20ms;
            v_diff_sig            <=  signed(V_Soll) - signed(v_ist_pulse(18 downto 5));
            v_diff_old_sig        <=  v_diff_sig;                          -- Delayregister von e(k)
            count_reset_delay_sig <=  count_reset_sig;                     -- Delaysignal zur Freigabeverzögerung

            if(windup_sig = '0') then
                control_old_sig <= control_sig;                            -- Rückführungsregister von u(k)
            end if;

            V_Ist       <= std_logic_vector(v_ist_pulse(18 downto 5));     -- Ausgangssynchronisation
        else
            count_reset_delay_sig <= '0';
        end if;
    end if;
end process V_Ist_Difference_P;
```

Die Regelabweichung *v_diff_sig* wird aus der Differenz von *V_Soll* und den ersten 14 Bits von *v_ist_pulse* gebildet. Die gewünschte Soll Geschwindigkeit wird dem Prozess vom MicroBlaze über ein Software Register zur Verfügung gestellt. Die Vektorbreiten der Geschwindigkeitsvektoren im Q-Format ergeben sich aus der maximal erreichbaren Geschwindigkeit des Carolo Cup Fahrzeuges ($\approx$ 4 m/s) und einer Anzahl von 10 Fractionalbits zur Erreichung einer ausreichenden Genauigkeit bei geringen Geschwindigkeiten. Dazu wird ein *sign* Bit implementiert:

$$SG_3G_2G_1 \bullet F_9...F_0 \quad \Rightarrow \quad 14\ Bit$$

Die größte aus diesem Darstellungsbereich resultierende Geschwindigkeit beträgt $7 + (1 - 2^{-10}) = 7,99902\ m/s$. Abhängig vom Gang beträgt die maximale Abweichung der Geschwindigkeiten zwischen 0,013 % und 0,034 %.

Da bei der Implementierung sowohl vorzeichenlose als auch vorzeichenbehaftete Operationen vorkommen, werden alle VHDL Berechnungen auf dem Datentyp *signed* oder *unsigned* aus dem Package *numeric_std* durchgeführt. Beide Datentypen basieren auf dem Datentyp *std_logic_vector* und können durch Typ-Casting gemeinsam verwendet werden (vgl. Code 6.5).

Das Delaysignal *v_diff_old_sig* und das Rückführungssignal *control_old_sig* übernehmen zur nächsten Taktflanke die Werte von *control_sig* und *v_diff_sig* (vgl. Code 6.5). Diese wurden bei der letzten Abtastperiode berechnet und liegen in einem Register bereit. *control_sig* wird aufgrund der Begrenzung des I-Anteils nur gespeichert wenn *windup_sig* = 0 ist (vgl. Abb. 6.7).

Zur Schnittstellensynchronisationzwischen den Pipelinestufen werden alle Register am Ende des Prozesses aktualisiert. Das Steuersignal *count_reset_sig* wird durch Setzen des Signals *count_reset_delay_sig* verzögert (vgl. Abb. 6.9). Mit der Verzögerung der Freigabesignale ist gewährleistet, dass die nachfolgende Stufe immer mit den neusten Ergebnissen rechnet und die Berechnung der Stellgröße nicht durch den Gebrauch von alten Werten verfälscht.

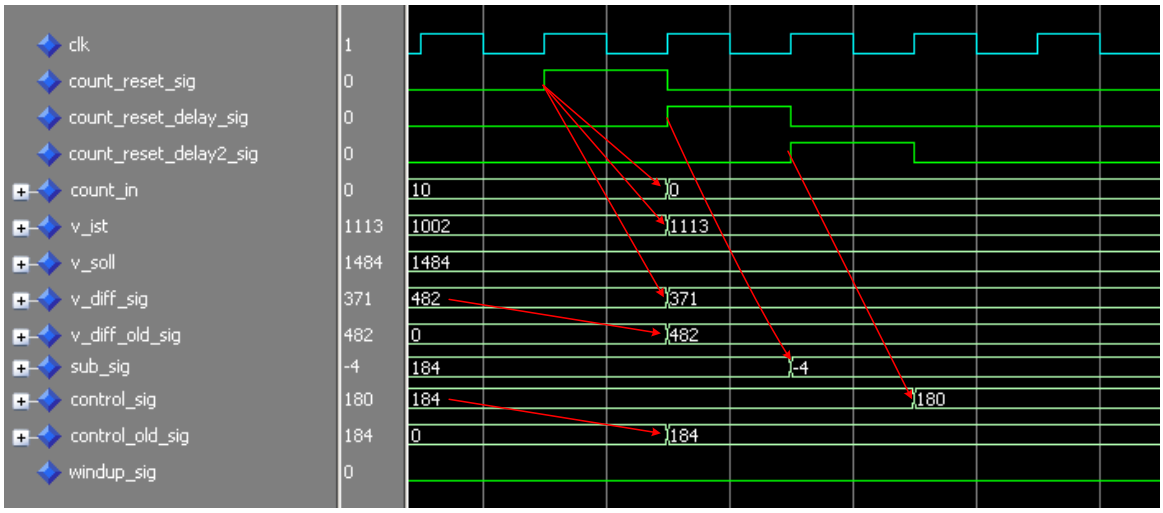

Abbildung 6.9.: Timing des Regler Moduls mit Verzögerung der Steuersignale

2. Pipelinestufe: Berechnung der Zwischenergebnisse für die Stellgröße

Die zweite Pipelinestufe berechnet das Zwischenergebniss *sub_sig* bei Freigabe von der ersten Pipelinestufe durch *count_rest_delay_sig*. Bei der Implementierung nach Code 6.7 werden vom Synthesewerkzeug sechs 11x14 Bit Multiplizierer, die parallel ihre Ergebnisse liefern, inferiert. Über ein geteiltes Software Register wird der Gang über 2 Bit vom MicroBlaze vorgegeben. Die aktuelle Regelabweichung *v_diff_sig* und die alte Regelabweichung *v_diff_old_sig* werden, abhängig vom Gang, mit dem ersten und zweiten Einstellparameter des PI-Reglers multipliziert (vgl. Tabelle 5.4 und Abb. 6.7). Hierfür werden vom Synthesewerkzeug zwei Multiplexer am Ausgang, die die entsprechenden Multiplikationen für die anschließende Subtraktion bestimmem, inferiert. Die Multiplexer und die große Anzahl an benötigten ALUs gibt den Ausschlag, dass der längste Signallaufzeitpfad von 11,288 ns sich in diesem Prozess befindet.

Durch „Resource Sharing“ kann die Anzahl der 11x14 Bit Multiplizierer auf zwei minimiert werden. Anstatt die Ausgänge zu multiplexen, werden dabei die Eingänge des Multiplizierers selektiert und anschließend mit den entsprechenden Parametern multipliziert (vgl. Abschnitt 7.5).

Die berechneten Einstellparameter aus Kapitel 5.5 werden in ein entsprechendes Q-Format konvertiert. Alle Parameter befinden sich im Bereich von $0 < Param > 1$ und benötigen somit keine spezifische Integeranteile. Die Anzahl der Fractionalbits oder die Größe des Bruchteils hängt von der gewünschten Genauigkeit der Parameter ab. Für die Einstellwerte des PI-Reglers wurden 10 Fractional Bits mit einem zusätzlichen *sign* Bit gewählt. Dadurch wird eine Präzision von $2^{-(10+1)} = 0,0004883$ erreicht und sowohl kleine als auch große Parameter nahezu 1:1 konvertiert.

$$Parameter : S \bullet F_9 ... F_0 \quad \Rightarrow \quad 11\ Bit$$

Um eine möglichst einfache Konfiguration der Einstellparameter zu gewährleisten, werden alle Parameter in der VHDL Entity des Regler Moduls als Generics deklariert (vgl. Code 6.6). Dadurch können die Werte der Generics bei der Komponenteninstanziierung angepasst und gegebenenfalls zu verändert werden.

Listing 6.6: Deklaration der Einstellparameter in Generics

```
0  param1_gear1  : signed(10 downto 0) := "00110001001"; -- 1. Parameter für 1.Gang mit KR2 = 0,3838  Q1.10 Format
   param2_gear1  : signed(10 downto 0) := "00100111000"; -- 2. Parameter für 1.Gang mit KR2 = 0,3047  Q1.10 Format
   param1_gear2  : signed(10 downto 0) := "00011100010"; -- 1. Parameter für 2.Gang mit KR2 = 0,2207  Q1.10 Format
   param2_gear2  : signed(10 downto 0) := "00010100010"; -- 2. Parameter für 2.Gang mit KR2 = 0,1582  Q1.10 Format
   param1_gear3  : signed(10 downto 0) := "00010001110"; -- 1. Parameter für 3.Gang mit KR2 = 0,1387  Q1.10 Format
5  param2_gear3  : signed(10 downto 0) := "00001111111"; -- 2. Parameter für 3.Gang mit KR2 = 0,1240  Q1.10 Format
```

In Code 6.6 werden die Deklarationen für die Einstellparameter gezeigt, die mit K_{R2} und T_{I2} berechnet worden sind. Man erkennt, im Vergleich zu Tabelle 5.4, dass die Konvertierung in das Q-Format durch das Abschneiden der niederwertigen Bits Ungenauigkeiten hervorbringt. Jedoch sind diese Quantisierungsfehler bei der Geschwindigkeitsregelung vernachlässigbar, da die Erhöhung der Vektorbreiten größere Multiplizierer zur Folge hätte.

Die Vektorbreiten der internen Variablen *mul_1_var* und *mul_2_var* ergeben sich aus der Summe der Faktoren. Sowohl die aktuelle Regelabweichung *v_diff_sig* als auch die alte *v_diff_old_sig* haben eine Breite von 14 Bit und werden mit einem 11 Bit breiten Parameter multipliziert. Daraus ergibt sich für die Variablen eine Vektorbreite von 25 Bit und eine Q-Fomat Darstellung wie folgt:

$$mul\_var : SGGGG \bullet F_{19}...F_0 \quad \Rightarrow \quad 25\ Bit$$

Bei der Multiplikation der zwei Q-Format Zahlen wird der Binärpunkt um 10 Bit nach links geschoben, sodass der ursprüngliche 10 Bit große Fractionalteil auf 20 Bit anwächst. Das doppelte *sign* Bit wird als Guard Bit interpretiert. Die Produkte werden in Variablen abgelegt, damit die nachfolgende Subtraktion die aktuell berechneten Zwischenergebnisse benutzen kann (vgl. Code 6.7). Die erste Multiplikation *mul_1_var* berechnet den P-Anteil des Reglers und sorgt somit für die schnelle Reaktion auf eine Regelabweichung.

Listing 6.7: Zweite Pipelinestufe zur Zwischenberechnung der Stellgröße

```
0  Control_P: process(CLK)
   variable mul_1_var, mul_2_var  :  signed(24 downto 0);
   begin
       if(CLK'event and CLK = '1') then
           if(Reset = '1') then
5              sub_sig   <=   (others => '0');
           elsif (count_reset_delay_sig = '1') then
               case Gear is
                   when "01" => mul_1_var := param1_gear1 * v_diff_sig;
                                mul_2_var := param2_gear1 * v_diff_old_sig;
10                              sub_sig <= mul_1_var(24 downto 10) - mul_2_var(24 downto 10);

                   when "10" => mul_1_var := param1_gear2 * v_diff_sig;
                                mul_2_var := param2_gear2 * v_diff_old_sig;
                                sub_sig <= mul_1_var(24 downto 10) - mul_2_var(24 downto 10);
15
                   when "11" => mul_1_var := param1_gear3 * v_diff_sig;
                                mul_2_var := param2_gear3 * v_diff_old_sig;
                                sub_sig <= mul_1_var(24 downto 10) - mul_2_var(24 downto 10);
                   when others => null;
20             end case;
               count_reset_delay2_sig <= count_reset_delay_sig;        --Verzögerung des Delaysignals
           else
               count_reset_delay2_sig <= '0';
       end if;
25 end if;
   end process Control_P;
```

In Abbildung 6.6 wird der Subtrahierer, der von zwei 3 x 1 Multiplexern gesteuert wird, dargestellt. Dieser berechnet die Differenz aus *mul_1_var* und *mul_2_var* und speichert durch eine nebenläufige Signalzuweisung das Ergebniss *sub_sig* in einem 15 Bit großen Pipelineregister. Da die Anzahl von 10 Fractionalbits eine ausreichende Genauigkeit mit sich bringt, werden die 10 niederwertigen Fractionalbits der Produkte abgeschnitten.

$$sub\_sig : SGGGG \bullet F_9...F_0 \quad \Rightarrow \quad 15\ Bit$$

Das Signal *sub_sig* wird und zur Entkopplung und zur Weitergabe des Zwischenergebnisses an die dritte Pipelinestufe benutzt und kann sowohl positiv als auch negativ sein. Dies ist der Fall, wenn die alte Regelabweichung wesentlich größer als die aktuelle ist und bei der Subtraktion das *sign* Bit gesetzt wird. Das Delaysignal *count_reset_delay_sig* wird zur Freigabe der dritten Pipelinestufe durch *count_reset_delay2_sig* verzögert und in einem Delayregister gespeichert.

3. Pipelinestufe: Berechnung, Begrenzung und Skalierung der Stellgröße für Weitergabe an PWM Modul

Die dritte Piplinestufe wird durch *count_reset_delay2_sig* freigegeben und berechnet, durch das in der zweiten Stufe bereitgestellte Zwischenergebniss *sub_sig*, die Stellgröße. Zur Berechnung der aktuellen Stellgröße *control_sig* wird *sub_sig* mit der alten Stellgröße *control_old_sig*, die bei der vorherigen Abtastperiode berechnet und gespeichert worden ist, addiert und in einer 15 Bit breiten vorzeichenbehafteten Variablen *add_var* abgelegt. Die Vektorbreite und das Q-Format des Additionsergebnisses ergeben sich aus dem größtem Summanden der Addition, dem Signal *control_old_sig*:

$$add\_var : SGGGG \bullet F_9 ... F_0 \quad \Rightarrow \quad 15\ Bit$$

Die Variable wird am Prozessende durch eine Ausgangssynchronisation in *control_sig* gespeichert (vgl. Code 6.8). Die Zwischenberechnung auf der Variablen ist notwendig, damit die aktuelle Stellgröße sofort zur Skalierung für das PWM Modul vorhanden ist. Bei direkter Benutzung von *control_sig* würde die Skalierung immer mit der alten Stellgröße rechnen und somit eine Abtastperiode verzögert sein.

```
0  PWM_Scale_P: process(CLK)
   variable add_var         : signed(14 downto 0);
   variable pwm_mul_var     : unsigned(15 downto 0);
   begin
       if(CLK'event and CLK = '1') then
5          if(Reset = '1') then
               control_sig      <= (others => '0');
               Control_PWM      <= (others => '0');
           elsif (count_reset_delay2_sig = '1') then
                   add_var := sub_sig + control_old_sig;                    -- Stellgröße Berechnung
10             if(add_var > "0000010000000000") then                        -- Stellgröße Begrenzung
                   pwm_mul_var := "10000000000" *  pwm_calculate;           -- Stellgröße Skalierung
                   windup_sig  <= '1';                                      -- Stellgröße Anti-Windup
               else
                   pwm_mul_var := unsigned(add_var(10 downto 0)) * pwm_calculate;
15                 windup_sig  <= '0';
               end if;
           control_sig <= add_var;
           Control_PWM <= std_logic_vector(pwm_mul_var(14 downto 0)) after 5ns;
           end if;
20     end if;
   end process PWM_Scale_P;
```

Listing 6.8: Berechnung der skalierten Stellgröße *Control_PWM* mit anschließender Ausgangssynchronisation

Aufgrund der Stellgrößenbegrenzung und der Begrenzung des I-Anteils durch das Anti-Windup darf die Stellgrößenvariable *add_var* nicht > 1 sein. Mit Einsatz des in Abb. 6.6 dargestellten Komparators wird *add_var* zur Laufzeit überprüft und gegebenenfalls wird das Signal *windup_sig* gesetzt (vgl. Code 6.8 - Zeile 10). Dieses wird als Freigabesignal für das Rückführungsregister *control_old_sig* in der ersten Pipelinestufe genutzt. Die aktuelle Stellgröße *control_sig* wird erst dann wieder gespeichert, wenn *windup_sig* null wird. So ist sichergestellt, dass der I-Anteil des Reglers die kommende Regelabweichung nicht weiter aufintegriert und er somit begrenzt ist (vgl. Abb. 6.7). Die alte Stellgröße *control_old_sig* hat bei jeder Berechnung von *add_var* einen Wert kleiner eins und der Regler kann die Stellgröße bei der nächsten Abtastperiode verkleinern. Die zu große Stellgröße, die den Sollwert überschritten hat, wird jedoch in *control_sig* gespeichert.

Zusätzlich zur Begrenzung wird der Komparator zur Skalierung der Stellgröße für das PWM-Modul benutzt. Das PWM Signal hat einen Duty Cycle zwischen 10% und 20% und eine Periode von 20 ms (vgl. Abschnitt 6.1.3). Innerhalb von 0,5 ms muss die Stellgröße auf alle Geschwindigkeitswerte abgebildet werden. Dies entspricht für den 20 Bit großen PWM Counter, bei einer Frequenz von 50 MHz, 25.000 Zählerticks.

$$25.000\ Ticks \quad = \quad 110000110101000 \quad \Rightarrow \quad 15\ Bit \tag{6.8}$$

Da die Stellgröße einen maximalen Wert von eins hat und mit 10 Fractional Bits im Q-Format dargestellt wird, muss eine Skalierung von 11 Bit auf 15 Bit implementiert werden. Hierfür wird ein Faktor *pwm_calculate* als VHDL Generic deklariert

$$pwm\_calculate \quad = \frac{25.000\ Ticks}{max.\ Stellgrösse} = \frac{110000110101000}{1.0000000000} = 11000 = 24 \tag{6.9}$$

Mit Hilfe von *pwm_calculate* kann die Stellgröße auf jedes Bit des Counters abgebildet werden und ermöglicht somit eine höhere Auflösung der Geschwindigkeiten, die durch das PWM Signal erzeugt werden. Insgesamt können 25.000 verschiedene Geschwindigkeiten dargestellt werden.

Die 15 Bit große Stellgröße *add_var* wird aufgrund des maximalen Wertes von eins auf 11 Bit gekürzt und bei $add\_var < 1$ direkt mit *pwm_calculate* multipliziert. Bei Überschreiten der Begrenzung wird *add_var* zuerst auf eins gesetzt und anschließend multipliziert (vgl. Code 6.8). Das Ergebniss wird in der Variablen *pwm_mul_var* abgelegt und nach Begrenzung und Skalierung werden die unteren 15 Bit, die nun die Anzahl an Zählerticks für die Berechnung des Duty Cycles darstellen, auf den Ausgang *Control_PWM* gelegt. Nach der Ausgangssynchronisation kann das PWM Modul die skalierte Stellgröße, in ein PWM Signal umwandeln.

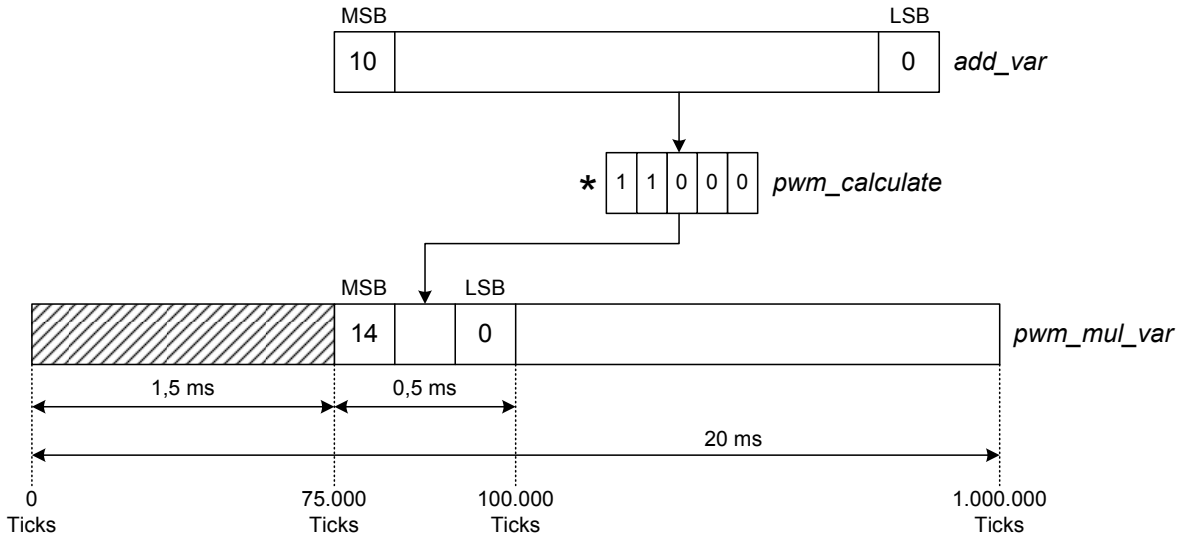

Abbildung 6.10.: Skalierung der Stellgröße für das PWM Signal *Control_PWM*

Durch die Multiplikation mit *pwm_calculate* entsteht bei der Skalierung eine Ungenauigkeit, die Auswirkung auf den maximalen PWM Duty Cylce hat und somit auch auf die maximal erreichbare Geschwindigkeit des Fahrzeuges. Bei einer Stellgröße von 1 ergibt sich durch die Multiplikation ein maximaler Wert von:

$$Control\_PWM_{Max} = 1.0000000000 * 11000 = 24.576\ Ticks \tag{6.10}$$

Zur Erreichung des maximalen Duty Cycles von 2 ms müsste *Control_PWM* einen maximalen Wert von 25.000 haben (vgl. Abschnitt 6.1.3). Durch die Abweichung von 424 Ticks ergibt sich bei einer Frequenz von 50 MHz der neue Duty Cycle:

$$\begin{aligned} Duty\ Cycle_{\ Max} &= 424\ Ticks * 20\ ns = 0,00848\ ms && (6.11)\\ &\Rightarrow\ 2\ ms - 0,00848\ ms && (6.12)\\ &= 1,99152\ ms && (6.13) \end{aligned}$$

Das Fahrzeug kann aufgrund dieser Abweichung die maximale Endgeschwindigkeit nicht vollständig erreichen.

Das Timing Diagramm 6.11 zeigt den Ablauf des Regelalgorithmuses mit der Pipelinestruktur. Die skalierte Stellgröße *Control_PWM* liegt nach etwa vier Takten am Ausgang an. Jede Signalzuweisung wurde in der Simulation mit dem VHDL Schlüsselwort *after* um 5 ns verzögert und dient dazu, die Signallaufzeit zu symbolisieren.

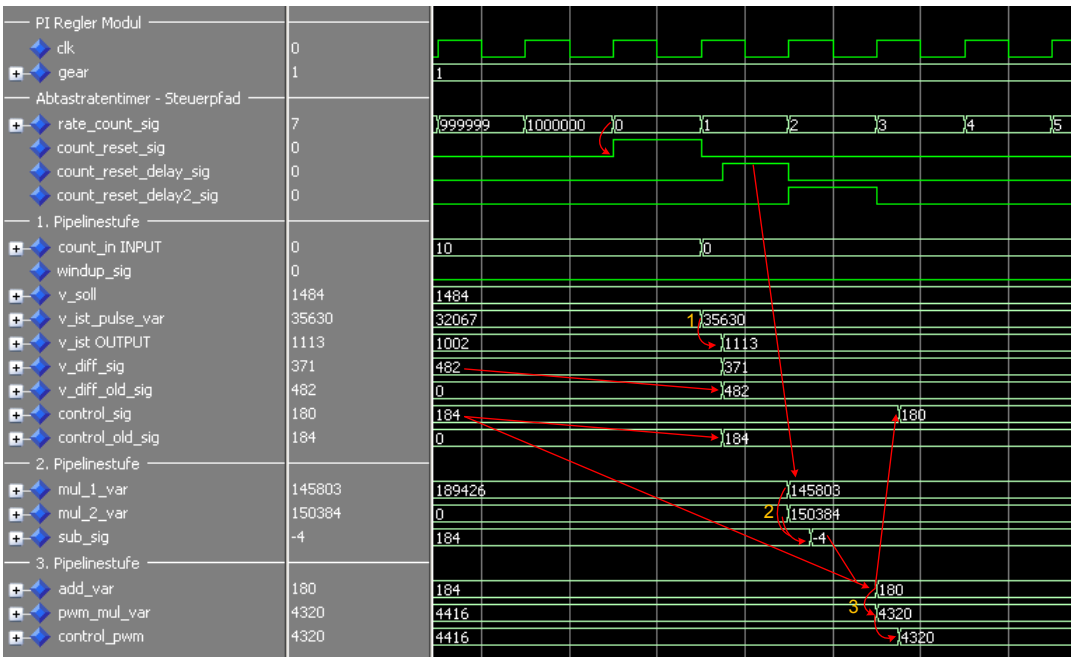

Abbildung 6.11.: Timingsimulation der dreistufigen Pipeline im ersten Gang bei $V_{Soll} = 1,45 m/s$

6.1.3. PWM Modul

Zur Ansteuerung des Fahrtenstellers erzeugt das PWM Modul das Signal *PWM_Output* mit einer Periode von 20 ms und einem Tastverhältniss (Duty Cycle) zwischen 10%-20%. Bei einem Stillstand des Motors *motor_holdup* ist PMOD JA1-L15 am Board für 1,5 ms (vgl. Abb. 6.12) gesetzt. Abhängig von der Drehrichtung wird diese Holdup Zeit entweder größer oder kleiner.

V	Duty Cycle
-100 %	1,0 ms
Stillstand	1,5 ms
+100 %	2,0 ms

Tabelle 6.2.: Daten der PWM Modulation

Für die Modulation von *PWM_Output* wird ein Counter, der den Zählerstand *pwm_count_sig* bei jedem Takt inkrementiert, implementiert. Bei einer einer PWM Periode von 20 ms muss der Counter 20 Bit breit sein. Dies ergibt sich aus der Taktperiode des Systemtakts und dem maximalen Zählerstand $Count_{Max}$ zur Modellierung der Periode.

$$Count_{Max} = \frac{20ms}{20ns} = 1.000.000Ticks \Rightarrow 20\ Bit \tag{6.14}$$

Bei Erreichen des maximalen Zählerstandes *periode_20ms* wird der Zählerstand *pwm_count_sig* des Counters zurückgesetzt und der synchronisierte Ausgang *PWM_Output* gesetzt (vgl. Abb. 6.12). Dieser bleibt solange auf logisch 1 bis die gewünschte skalierte Stellgröße *Control_PWM* erreicht ist.

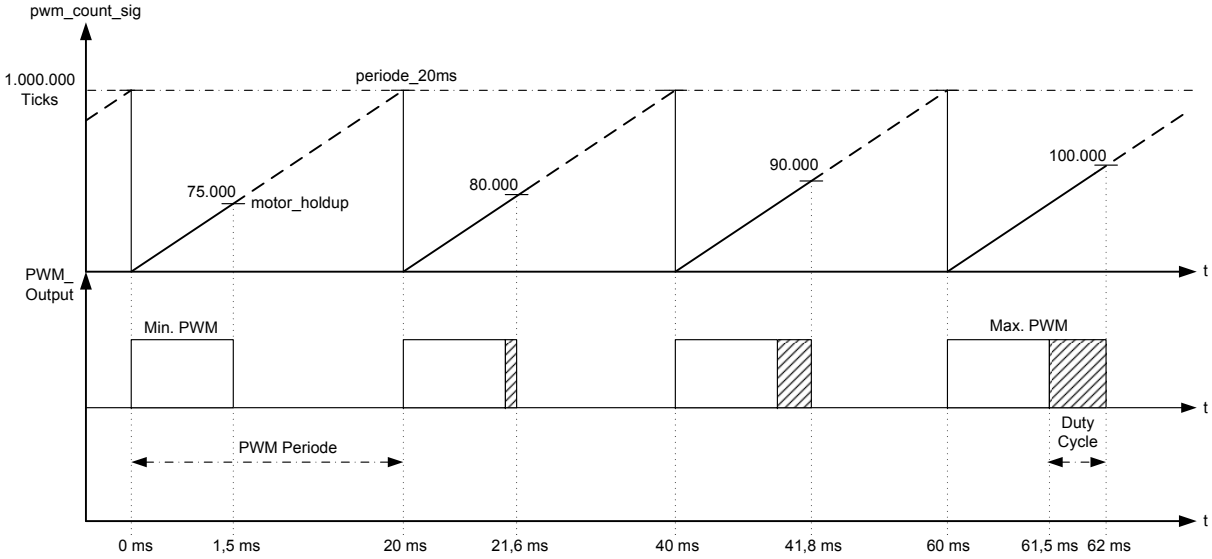

Abbildung 6.12.: PWM Erzeugung mit einem 20 Bit großen Counter für die Periode und den Duty Cycle

Da der Counter taktflankengesteuert ist wird das gesamte PWM Modul in einem getaktetem VHDL Prozess implementiert. Für die Berechnung des Duty Cycles wird kein weiterer Counter inferiert, sondern der bereits implementierte PWM Perioden Counter benutzt. Ausgehend von der *motor_holdup* Zeit wird die Stellgröße, die am Eingang *Control_PWM* anliegt, je nach Drehrichtung *Direction* entweder addiert oder subtrahiert und anschließend mit dem Zählerstand *pwm_coun_sig* verglichen (vgl. Zeile 11-14 in Code 6.9). Bei Übereinstimmung wird der Ausgang *PWM_Output* auf 0 gesetzt und bis zur nächsten Übereinstimmung gewartet.

Listing 6.9: VHDL Prozess zur Modellierung der PWM Periode und des Duty Cycles

```
0  PWM_Periode: process(CLK)
   begin
       if(CLK'event and CLK = '1') then
           if(Reset = '1') then
                 pwm_count_sig  <=  (others => '0');
5                PWM_Output     <=  '0';
           elsif(pwm_count_sig = periode_20ms) then
                 pwm_count_sig <= (others => '0');
                 PWM_Output <= '1';
           else
10             pwm_count_sig <= pwm_count_sig + 1;
               if((pwm_count_sig = motor_holdup + unsigned(Control_PWM)) and Direction = '0') then      --Vorwärtsfahrt
                   PWM_Output <= '0';
               elsif((pwm_count_sig = motor_holdup - unsigned(Control_PWM)) and Direction = '1') then  -- Rückwärtsfahrt
                   PWM_Output <= '0';
15         end if;
           end if;
       end if;
   end process PWM_Periode;
```

Am Ausgang des PWM Moduls wird ein D-Flip Flop mit synchronen Reset inferiert (vgl. Abb. 6.13). Dieses wird bei einem Overflow des Counters gesetzt und der Ausgang *PWM_Output* wird auf High gezogen. Der Dateneingang des Flip Flops ist an Ground angeschlossen. Er setzt *PWM_Output* beim Erreichen des Duty Cycles (CE = 1) auf Null.

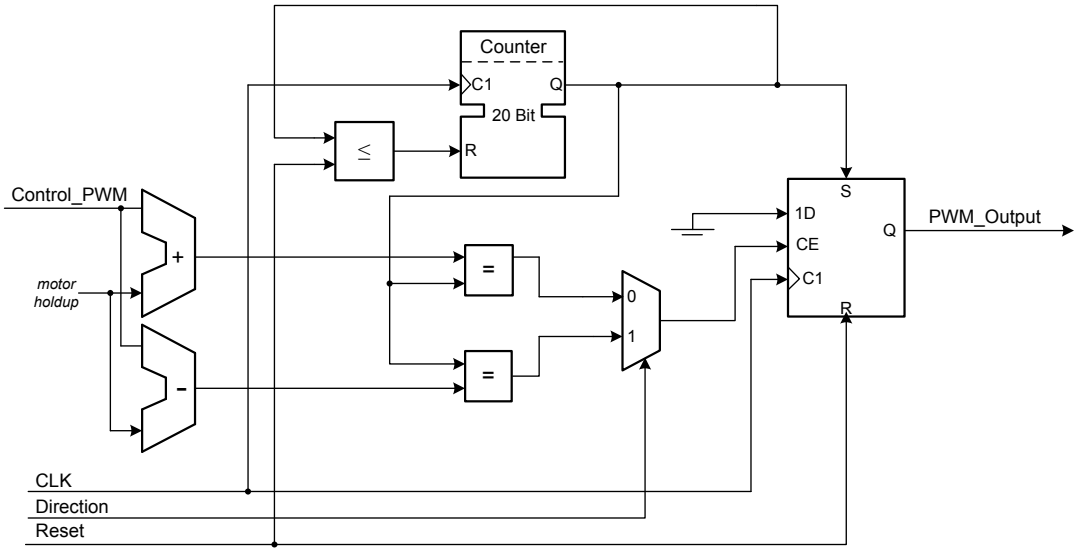

Abbildung 6.13.: RTL Schaltbild des PWM Moduls

Da zu Systemstart keine Stellgröße vorhanden ist und somit *Control_PWM = 0* ist, wird das Fahrzeug erst ab der zweiten Abtastperiode mit einer PWM angesteuert. Dies entspricht einer Verzögerung von 20 ms.

6.1.4. Simulation der Top Entity *V_Regler_Top_Entity*

Zum Test der Top Entity wurde eine VHDL Testbench, die eine konstante Pulsfolge, den ersten Gang und die Soll Geschwindigkeit vorgibt, erstellt. Abb. 6.14 zeigt einen Ausschnitt der VHDL Simulation. Es erfolgt keine Rückführung der Stellgröße auf den Eingang des Pulszählers und somit ist der Regelkreis offen. Die Ist Geschwindigkeit bleibt aufgrund der Pulsfolge konstant. Es wird ausschließlich die Funktion der einzelnen Module, die in der Top Entity zusammengefasst sind, getestet.

- Eine Pulsfolge bestehend aus 10 Pulsen mit einer Pulsbreite von 2 ms und keiner Drehrichtungsänderung wurde generiert
- Als Soll Geschwindigkeit *V_Soll* wurde die maximale Geschwindigkeit im ersten Gang von 1,45 m/s gewählt
- Das Freigabesignal *Distance_Enable* wird nach 1,5 ms gesetzt und bleibt für 10 ms konstant auf eins. In dieser Zeit zählt der *DistCounter* parallel zum *PulseCounter*
- Nach 13 ms wird über *Distance_Reset* der Zählerstand des *DistCounters* zurückgesetzt und bis zum nächsten *Distance_Enable* gewartet

Da zu Beginn der Simulation *V_Ist* gleich null ist, erzeugt der Regler eine Stellgröße, die nahezu dem Maximum entspricht. Nach der zweiten Abtastperiode erreicht er diesen maximalen Wert von 24.576 Zählerticks. Dies entspricht exakt dem berechneten Wert aus Abschnitt 6.10. Die PWM, die sich daraus ergibt, hat einen Duty Cycle von 1,99154 ms und entspricht bis auf eine Abweichung von 20 ns exakt dem Wert aus Abschnitt 6.11. Aufgrund der Ausgangssynchronisation von *Control_PWM* hat die PWM eine Abweichung von einem Takt (vgl. Abb. 6.13). Die resultierende Geschwindigkeit von 1,4438 m/s hat eine Abweichung von 0,43% von der Soll Geschwindigkeit, die 1,45 m/s beträgt.

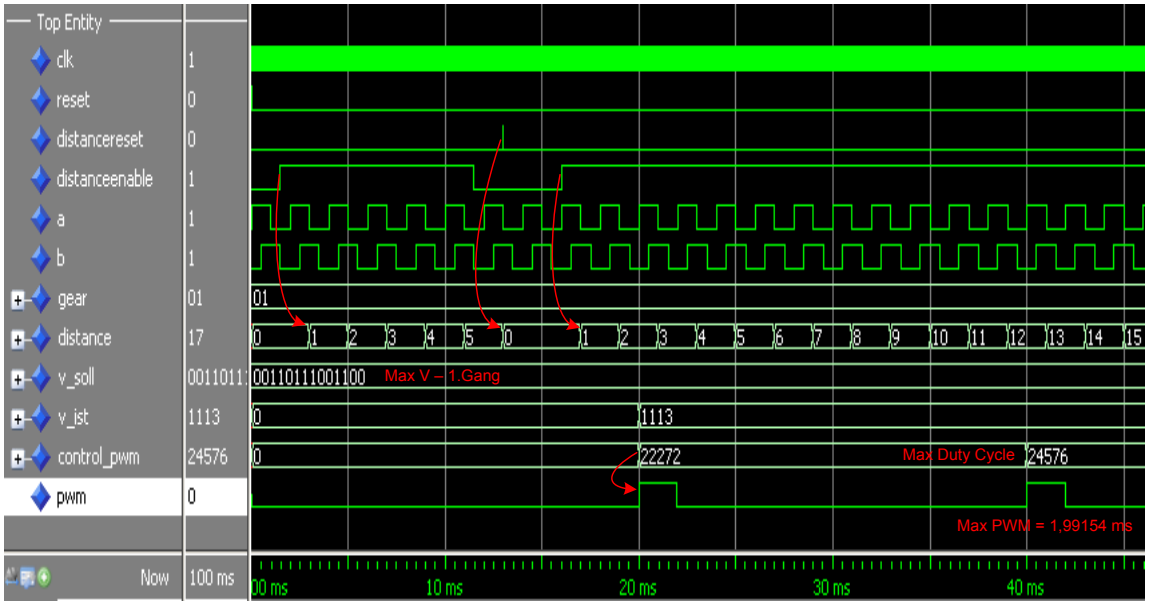

Abbildung 6.14.: Simulation der Top Entity *V_Regler_Top_Entity*

6.2. V-Regler IP Core

Bei der Erstellung des IP-Cores mit dem „Create and Import Peripheral Wizard“ des EDK Tools werden zwei VHDL Template Dateien erzeugt: Die *user_logic.vhd*, die für die Kommunikation mit dem PLB Bus über Software Register verantwortlich ist und die Top Entity *pi_controlspeed.vhd* (vgl. Abb. 6.15).

- **user_logic.vhd**
  - Instantiierung der Top Entity als Komponente und Verdrahtung der internen Signale über eine *portmap* (vgl. Code 6.10)
  - Deklaration von fünf 32 Bit großen adressierbaren Software-Registern (*SW_REG*) zur Kommunikation mit dem MicroBlaze über den PLB Bus
  - Portliste mit Schnittstellensignalen (*Pulse_A, Pulse_B, PWM*), die außerhalb des FPGAs an die PINs angeschlossen werden
  - Lokale Signale zum taktsynchronen Beschreiben der Software Register (vgl. Code 6.11)
  - Setzen des Interrupt Status Registers für den Abtastperioden Interrupt über *IP2Bus_IntrEvent*
- **pi_controlspeed.vhd**
  - Instantiierung der User_Logic und des PLB IPIF Bus Interfaces
  - Instantiierung des Interruptcontrollers und Konfiguration des Capture Modus (*registered level - non inverting*)
  - Schnittstellensignale der User_Logic werden über eine *portmap* an die Portliste weitergeleitet und so nach außen geführt

Listing 6.10: Top Entity Portmap der User_Logic

```
TOP_Instantiate: V_Regler_Top_Entitiy
  port map
  (
    CLK_Top             =>  Bus2IP_Clk,              -- System Takt von 50 MHz
    Reset_Top           =>  Bus2IP_Reset,            -- System Reset
    DistanceReset_Top   =>  slv_reg1(29),            -- Bit 2 von Reg 1 für DistanceReset von MB
    DistanceEnable_Top  =>  slv_reg1(28),            -- Bit 3 von Reg 1 für DistanceEnable von MB
    Puls_A_Top          =>  Puls_A_UserLogic,        -- externes Signal von Inkrementalgeber
    Puls_B_Top          =>  Puls_B_UserLogic,        -- externes Signal von Inkrementalgeber
    Gear_Top            =>  slv_reg1(30 to 31),      -- Bit 0-1 von Reg 1 für Gang von MB
    V_Soll_Top          =>  slv_reg0(18 to 31),      -- Register 0 für V Soll von MB
    Distance_Top        =>  distance_UserLogic_sig,  -- lokales Signal für MB
    TimePeriod_Top      =>  timeperiod_UserLogic_sig, -- Generierung des INterrupts aus Abtastperiode
        V_Ist_Top       =>  v_ist_UserLogic_sig,     -- Register 2 für V Ist von MB
    Control_PWM_Top     =>  control_pwm_UserLogic_sig, -- lokales Signal für MB
    PWM_Top             =>  PWM_UserLogic            -- externes Signal
  );
```

Listing 6.11: Beschreiben der Software Register zur Kommunikation mit MicroBlaze

```
slv_reg2(18 to 31)  <=  v_ist_UserLogic_sig;         -- Beschreiben von SW 2 mit V_Ist zur Übertragung an MB
slv_reg3(23 to 31)  <=  distance_UserLogic_sig;      -- Beschreiben von SW 3 mit Distancezur Übertragung an MB
slv_reg4(17 to 31)  <=  control_pwm_UserLogic_sig;   -- Beschreiben von SW 4 mit PWM zur Übertragung an MB
```

Der V-Regler IP Core wird an den PLB Bus angeschlossen und kommuniziert über fünf Software Register, die jeweils eine eigene Adresse besitzen, mit dem MicroBlaze und anderen IP Blöcken (vgl. Abb. 6.15). Alle Register sind mit *Bus2IP_Clk* getaktetet und verfügen über jeweils zwei Steuersignale *slv_reg_write_sel* und *slv_reg_read_sel*. Über *Bus2IP_Data* werden die Register vom MicroBlazes über den PLB Bus mit Daten beschrieben. Register 0 wird vom MicroBlaze mit der Soll Geschwindigkeit beschrieben und vom IP Core ausgelesen. Das geteilte Steuerregister 1 wird vom MB mit dem gewünschten Gang beschrieben. Über die zugehörige Adresse kann der ebenfalls am PLB abgeschlossene IP Core des Einparkassistenten *DistanceEnable* und *DistanceReset* für die Parklückenberechnung an Bitstelle 2 und 3 schreiben. Die restlichen Register werden vom IP mit Messwerten und Berechnungen beschrieben und anschließend vom MB über die Software gelesen. Der Fahrzeugmodell IP kann über die Adresse die Register auslesen und weiterverarbeiten.

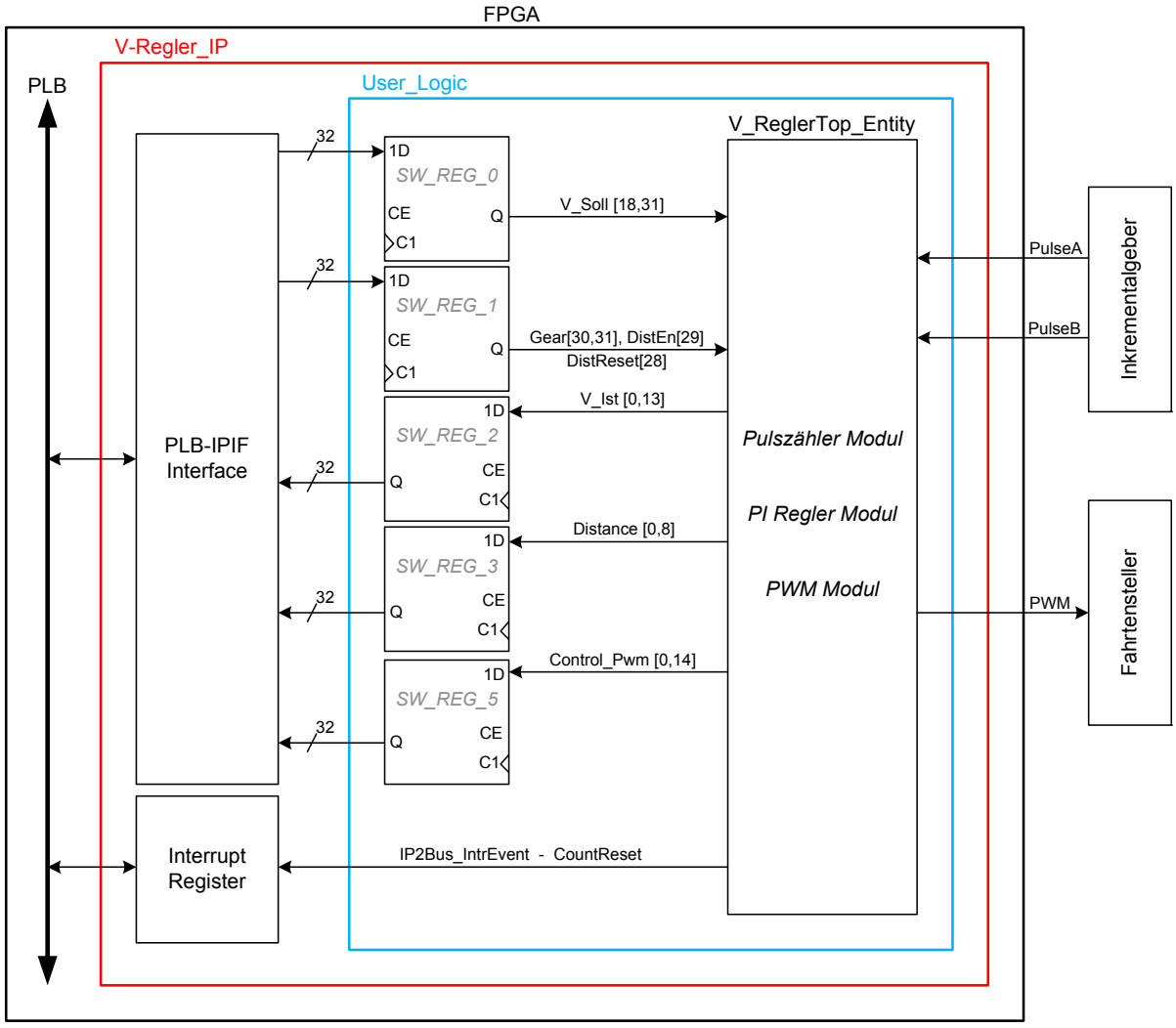

Abbildung 6.15.: User Logic Implementation des V-Regler IP Cores

Die Softwareregister 2,3 und 5 werden nicht wie gewöhnlich über einen Multiplexer kombinatorisch beschrieben, sondern werden als Ausgangssynchronisationsregister direkt vom IP beschrieben. Der MicroBlaze kann von diesen Registern nur lesen, aber sie nicht beschreiben (vgl. Code 6.11).

Die Signale *PulseA*, *PulseB* und *PWM* werden als extern deklariert und in der *.ucf* Datei auf die PMOD Anschlüsse des Boards gelegt (vgl. Code 6.12).

Listing 6.12: UCF Datei des V-Regler IP Cores

```
Net pi_controlspeed_0_PWM_VRegler_pin      LOC = L15;
Net pi_controlspeed_0_Puls_A_VRegler_pin   LOC = K12;
Net Puls_B_VRegler_pin                     LOC = L17;
```

Da der V-Regler IP Core aus mehreren VHDL Dateien aufgebaut ist, wird die „Peripheral Analyze Order (PAO)“ Datei für die Synthese angepasst. Diese Datei bestimmt die Reihenfolge der Compilierung der VHDL Dateien. Aus diesem Grund muss darauf geachtet werden, dass die Einträge in der PAO Datei stets die richtige hierarchische Struktur aufweisen (vgl. Code 6.13).

Listing 6.13: PAO Datei des V-Regler IP Cores

```
lib pi_controlspeed_v1_00_a PI_Regler vhdl
lib pi_controlspeed_v1_00_a Pulszaehler vhdl
lib pi_controlspeed_v1_00_a PWM vhdl
lib pi_controlspeed_v1_00_a Top_Entity vhdl
lib pi_controlspeed_v1_00_a user_logic vhdl
lib pi_controlspeed_v1_00_a pi_controlspeed vhdl
```

Damit das EDK Tool die externen Ports *Pulse_A_VRegler*, *Pulse_B_VRegler* und *PWM_VRegler* erkennt, wird die „Microprocessor Peripheral Definition (MPD)“ Datei erweitert. Das Signal *V_Ist_VRegler*, das direkt zum Odometrie IP weitergeleitet wird, wird ebenfalls eingetragen. Diese Datei definiert das Interface des V-Regler IP Cores und enthält alle Signale des Hardware Moduls. Die Signalnamen in der MPD Datei müssen mit den Portdeklarationen, den Vektorbreiten und der Datenrichtung in *pi_controlspeed.vhd* übereinstimmen.

Listing 6.14: MPD Datei des V-Regler IP Cores

```
PORT Puls_A_VRegler = Puls_A_VRegler, DIR = I
PORT Puls_B_VRegler = Puls_B_VRegler, DIR = I
PORT PWM_VRegler = PWM_VRegler, DIR = O
PORT V_Ist_VRegler = V_Ist_VRegler, DIR = O
```

Über das EDK wurden zusätzlich zum MicroBlaze Mikrocontroller die für die Implementierung und den Test notwendigen IP Blöcke:

- Micron_RAM (16 MB großer SDRAM)
- Debug_Module
- LED_7SEGMENT, LEDs_8Bit, Push_Buttons_3Bit und Switches_8Bit
- RS232_PORT

hinzugefügt. Alle IP Blöcke werden genutzt, um die auf dem Board vorhandenen Peripheriegeräte zu nutzen (vgl. Abschnitt 3.2). Der SDRAM und die RS232 Schnittstellen werden für die Datenlogger Funktion verwendet. Die im RAM gespeicherten Messwerte werden über die serielle Schnittstelle an den PC übertragen (vgl. Abschnitt 6.4). Alle weiteren IPs werden für die Visualisierung und die Konfiguration des Systems verwendet (vgl. Abschnitt 6.3).

6.3. MicroBlaze Software Implementierung

Das C Programm, das auf dem MicroBlaze läuft, wurde mit dem „Software Development Kit (SDK)“ erstellt und wird genutzt, um die Soll Geschwindigkeit und den Gang für den V-Regler IP Core vorzugeben. Für die Initialisierung der GPIOs wird die Methode *XGpio_Initialize*, die als Übergabeparameter die Adresse einer XGpio Variablen und die Device ID des Gerätes erhält, aufgerufen. Anschließend wird die Datenrichtung mit *XGpio_SetDataDirection* gesetzt(vgl. Code 6.15).

Bevor das Programm die Endlosschleife betritt, wird die Soll Geschwindigkeit in Software Register 0 geschrieben. Der V-Regler IP erwartet einen 14 Bit großen Wert, der an Stelle slv_reg0(18 to 31) steht (vgl. Code 6.10). Hierfür wurde vom EDK eine Makro Methode erstellt, der die Basis Adresse, der Software Register Offset und die zu schreibenden Daten übergeben werden (vgl. Code 6.15). Die *User_Logic* kann anschließend das Register auslesen und die Daten an das PI Regler Modul weiterleiten.

Für den Test des Systems mit dem Nexys 2 Board werden Dipswitch SW0 und SW1 genutzt, um den gewünschten Gang einzustellen. Je nach Schalterstellung schreibt der MicroBlaze den Gang in *Slv_Reg1* und sendet diesen an die *User_Logic*. Zur Kontrolle wird der aktuelle Gang auf der 7 Segment Anzeige ausgegeben. Zusätzlich leuchtet die entsprechende LED. Ist kein Schalter geschaltet, dann wird als Defaultzuweisung Gang eins gesendet (vgl. Code 6.15). Die Software liest über *XGpio_DiscreteRead*, der der Pointer auf die Adresse übergeben wird, den Status aller Schalter ein und entscheidet anhand der Codierung welcher Schalter geschaltet ist.

Listing 6.15: Abfrage der Dipswitches und Senden des Ganges

```
0  XGpio led_one;                  -- Pointer auf die Basisadresse der Leds
   XGpio dipswitch_one;            -- Pointer auf die Basisadresse der Schalter
   XGpio seven_segment;            -- Pointer auf die Basisadresse der 7 Segmentanzeige
   XGpio button_one;               -- Pointer auf die Basisadresse der Taster

5  XGpio_Initialize(&led_one, XPAR_LEDS_8BIT_DEVICE_ID);              -- Initialisierung der Leds
   XGpio_Initialize(&button_one, XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID);   -- Initialisierung der Taster
   XGpio_Initialize(&dipswitch_one, XPAR_SWITCHES_8BIT_DEVICE_ID);    -- Initialisierung der Schalter
   XGpio_Initialize(&seven_segment, XPAR_LED_7SEGMENT_DEVICE_ID);     -- Initialisierung der Anzeige

10 XGpio_SetDataDirection(&led_one, 1, 0x00000000);                   -- Leds als Output deklarieren
   XGpio_SetDataDirection(&button_one, 1, 0x11111111);                -- Taster als Input deklarieren
   XGpio_SetDataDirection(&dipswitch_one, 1, 0x11111111);             -- Schalter als Inout deklarieren
   XGpio_SetDataDirection(&seven_segment, 1, 0x00000000);             -- 7 Segmentanzeige als Output deklarieren

15 PI_CONTROLSPEED_mWriteReg(XPAR_PI_CONTROLSPEED_0_BASEADDR,PI_CONTROLSPEED_SLV_REG0_OFFSET, 0x400);

   while(1){

   dipswitch_status = XGpio_DiscreteRead(&dipswitch_one, 1);          -- Auslesen der Schalterstellung
20
   switch(dipswitch_status)
   {
       case 1:
           XGpio_DiscreteWrite(&led_one, 1, 0x01);
25         XGpio_DiscreteWrite(&seven_segment, 1, 0x4F);
           PI_CONTROLSPEED_mWriteReg(XPAR_PI_CONTROLSPEED_0_BASEADDR,PI_CONTROLSPEED_SLV_REG1_OFFSET, 0x1);
           break;
       case 2:
           XGpio_DiscreteWrite(&led_one, 1, 0x02);
30         XGpio_DiscreteWrite(&seven_segment, 1, 0x24);
           PI_CONTROLSPEED_mWriteReg(XPAR_PI_CONTROLSPEED_0_BASEADDR,PI_CONTROLSPEED_SLV_REG1_OFFSET, 0x2);
           break;
       case 3:
           XGpio_DiscreteWrite(&led_one, 1, 0x03);
35         XGpio_DiscreteWrite(&seven_segment, 1, 0x30);
           PI_CONTROLSPEED_mWriteReg(XPAR_PI_CONTROLSPEED_0_BASEADDR,PI_CONTROLSPEED_SLV_REG1_OFFSET, 0x3);
           break;
       default:
           XGpio_DiscreteWrite(&led_one, 1, 0x00);
40         XGpio_DiscreteWrite(&seven_segment, 1, 0xFF);
           PI_CONTROLSPEED_mWriteReg(XPAR_PI_CONTROLSPEED_0_BASEADDR,PI_CONTROLSPEED_SLV_REG1_OFFSET, 0x1);
           }
```

Methode	Funktion
XGpio_Initialize(...)	Initialisierung eines GPIOs bei Übergabe der Geräte ID und eines Pointers auf die Basisadresse
XGpio_SetDataDirection(...)	Setzen der Datenrichtung eines GPIOs bei Übergabe der Geräte ID, des Operationsmodus und der Basisadresse
XGpio_DiscreteRead(...)	Auslesen eines GPIO Registers, das spezifiziert ist durch die Basisadresse und den Operationsmodus
XGpio_DiscreteWrite(...)	Beschreiben bzw. Setzen eines GPIO Registers durch die Basisadresse, den Channel und die Daten
PI_Controlspeed_mWriteReg(...)	Makro Methode des V-Regler IP Cores zum Beschreiben eines Softwareregisters, spezifiziert durch die Basisadresse, den Softwareregister Offset und die zu schreibenden Daten
PI_Controlspeed_mReadSlaveReg3	Makro Methode des V-Regler IP Cores zum direkten Auslesen eines Softwareregisters bei Übergabe der IP Basisadresse
PI_Controlspeed_EnableInterrupt	Freigabe des V-Regler IP Interrupts bei Übergabe der Basisadresse des IPs
xil_printf(...)	UART Funktion zur byteweisen Beschreibung der seriellen Schnittstelle
microblaze_enable_interrupts()	Freigabe der Interrupts des MicroBlaze Systems. Bei Start des Prozessor sind standardmässig alle Interrupts deaktiviert
microblaze_register_handler(...)	Registrierung der ISR für den MicroBlaze Interrupt

Tabelle 6.3.: Methoden des MicroBlaze Softwaresystems

Interruptgenerierung für die globale Zeitgebung

Der Interrupt für die globale Zeitgebung wird direkt in der User Logic erzeugt. Der Ausgang *CountResetRate* des Abtastratentimers wird in der *user_logic* an den Ausgang *IP2Bus_IntrEvent* des instantiierten Interrupt Standalone Controllers angeschlossen. Der Capture Modus „Registered Level (non inverting)“ wird in der Top Entity *pi_controlspeed* in der Konstanten *USER_INTR_CAPTURE_MODE* deklariert. Ein zusätzlicher Interrupt Controller *XPS_INTC*, der als extra IP Core integriert wird, wird nicht benötigt. Der MicroBlaze Prozessor besitzt einen Interrupteingang, der direkt mit einem IP Core verbunden werden kann. Im „System Assembly View“ des EDK wird die IRQ Leitung des V-Regler IPs an den Interrupteingang des MicroBlazes angeschlossen.

Eine „Interrupt Service Routine (ISR)“, die den Interrupt verarbeitet, wird mit *microblaze_register_handler(...)* an den IRQ gebunden. Die ISR liest das Interrupt Status Register des V-Regler IP Cores aus und nimmt anschließend den IRQ wieder zurück. Sowohl die Interrupts des MicroBlazes als auch die Interrupts des V-Regler IPs müssen explizit freigegeben werden (vgl. Tabelle 6.3).

6.4. Datenlogger Funktion

Zur Aufzeichung von Messwerten während einer Testfahrt wurde eine Datenlogger Funktion implementiert. Diese verarbeitet die vom PI Regler Modul berechneten und in Software Registern abgespeicherte Werte für Ist Geschwindigkeit und Stellgröße (vgl. Abb. 6.15). In einer Endlosschleife werden die Register drei und vier ausgelesen und in einer Xuint32 Variablen gespeichert. Da sowohl Ist Geschwindigkeit als auch Stellgröße nach einer bestimmten Zeit einen konstanten Wert annehmen, speichert der Datenlogger nur unterschiedliche Werte (vgl. Code 6.16). Für den Vergleich des aktuellen Wertes mit dem vorherigen Wert muss der alte Wert ebenso gespeichert werden. Bei nicht Übereinstimmung der Werte wird der aktuelle Wert in ein Array der Größe 100 geschrieben. Sobald die Arraygrenze erreicht wird, wird bis zum Reset kein weiterer Wert mehr gespeichert.

Listing 6.16: Datenloggerfunktion zum Auslesen von V_Ist und Control_PWM

```
0  //######################## Auslesen der Software Register SW3 und SW4 ##########################

   control_pwm = PI_CONTROLSPEED_mReadSlaveReg4(XPAR_PI_CONTROLSPEED_0_BASEADDR, 0);
   v_ist       = PI_CONTROLSPEED_mReadSlaveReg3(XPAR_PI_CONTROLSPEED_0_BASEADDR, 0);

5  //################ Abfrage der Taster BTN2 und BTN3 und Ausgabe auf UART ####################

   buttons_status = XGpio_DiscreteRead(&button_one, 1);
   switch(buttons_status){
   case 2:
10     for(count = 0; count < 100; count ++){
           xil_printf("V_Ist: 0x%08X\r\n",v_ist_array[count]);
       }
       break;
   case 4:
15     for(count = 0; count < 100; count ++){
           xil_printf("Control_PWM: 0x%08X\r\n",control_pwm_array[count]);
       }
       break;
   }
20
   //################ Speichern der Messwerte V_Ist und Control_PWM im RAM ####################

   if(control_pwm != control_pwm_old)
   {
25     control_pwm_array[array_count_control_pwm] = control_pwm;
       array_count_control_pwm ++;
       if(array_count_control_pwm == 99){array_count_control_pwm = 98;}
   }
   if(v_ist != v_ist_old){
30     v_ist_array[array_count_v_ist] = v_ist;
       array_count_v_ist ++;
       if(array_count_v_ist == 99){ array_count_v_ist = 98;}
   }
   v_ist_old = v_ist;
35 control_pwm_old = control_pwm;
```

Der Anwender kann nach einer Testfahrt die gespeicherten Ist Geschwindigkeiten und Stellgrößen über die serielle Schnittstelle RS232 mit einer Baudrate von 9600 auslesen. Hierfür wurden die Taster BTN2 für die Geschwindigkeiten und BTN3 für die Stellgrößen konfiguriert. Nach Drücken eines dieser Taster liest die Software das Array aus dem RAM aus und übergibt alle Werte an den PC. Man beachte, dass es sich bei der Stellgröße um die PWM skalierte Stellgröße handelt und diese gegebenenfalls durch 24 geteilt werden muss um die tatsächliche Stellgröße zu bekommen.

Das komplette C Programm wird anstatt im BRAM im Micron SRAM abgelegt. Dieser hat eine Größe von 16 MB und bietet genügend Platz für die komplette Software. Hierfür muss das Linker Script des Programms, das angibt wo die .text, .rodata, .bss und .stack Section gespeichert werden, verändert werden.

7. Messtechnische Analyse und Simulation des Gesamtsystems

Für die Geschwindigkeitsregelung müssen das Carolo Cup Fahrzeug und die Fahrzeugregelstrecke exakt analysiert und ausgewertet werden. Die Berechnung der Fahrzeug spezifischen Daten, die für die Berechnung der Einstellparameter und zur Modellierung benötigt werden, werden in diesem Kapitel vorgestellt. Anschließend wird das gesamte Geschwindigkeitsregelsystem simuliert und das Testszenario beschrieben.

7.1. Aufzeichnung und Approximation der Sprungantwort

Zur Identifikation der Regelstrecke und zur Bestimmung der maximalen Geschwindigkeiten des Fahrzeuges wurde der erste Pulskanal des Inkrementalgebers an ein Speicheroszilloskop angeschlossen und die Sprungantwort der jeweiligen Gänge gemessen. Hierfür wurde das Fahrzeug, im aufgebockten Zustand, mit der Fernbedienung auf die maximale Geschwindigkeit beschleunigt.

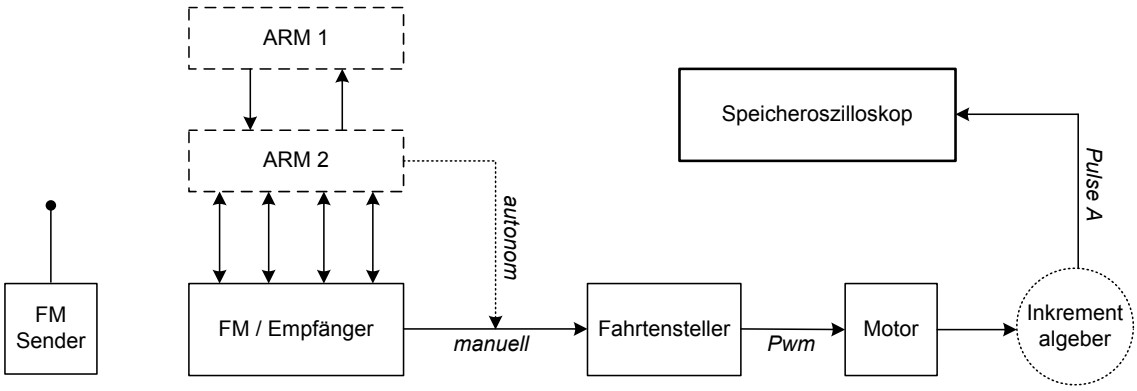

Abbildung 7.1.: Messaufbau zum Aufzeichnen der Sprungantworten

Bei einer Samplerate von 100 μs wurden 10.000 Messwerte innerhalb von einer Sekunde aufgezeichnet. Diese wurden vom Oszilloskop in einer CSV Datei abgelegt und gespeichert. Die Messdaten haben entweder den Wert von $\approx$ 5 V oder von $\approx$ 0 V. Je nachdem wie der Lichtstrahl des Inkrementalgebers reflektiert wird. Ein Java Programms liest die CSV Datei ein und berechnet aus den Messwerten die Pulsbreiten des Inkrementalgeberpulses (vgl. Abb. 7.2), die aus einer Anzahl von Samples bestehen. Aus diesen Zeitwerten werden mit der allgemeinen Geschwindigkeitsformel die zugehörigen Geschwindigkeiten pro Puls ermittelt und in einer Excel Datei gespeichert.

Die Strecke pro Puls, die für die Geschwindigkeitsberechnung verwendet wird, ergibt sich aus dem Radumfang und der Anzahl an Schlitzen auf der Drehscheibe:

$$s_{pulse} = \frac{261\ mm}{120\ Schlitze} = 2,175\ mm \tag{7.1}$$

In Abb. 7.2 sind die Pulsbreiten des ersten Ganges dargestellt. Man erkennt, anhand der immer kleiner werdenden Pulsbreiten, die Beschleunigung des Fahrzeuges.

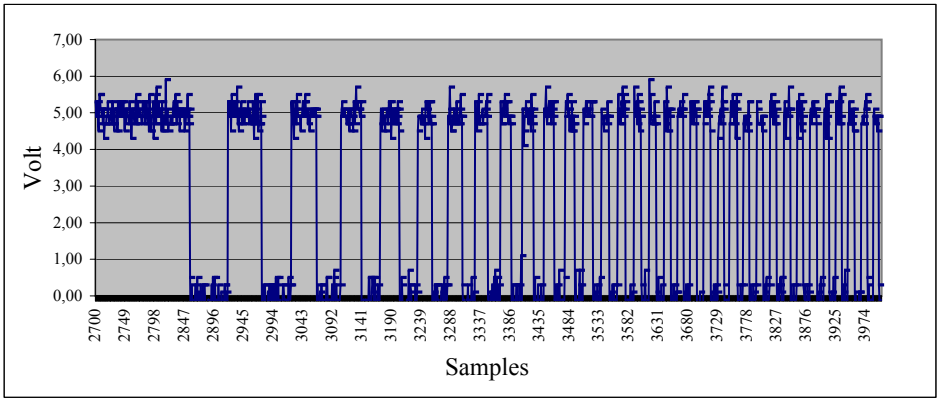

Abbildung 7.2.: Pulsbreiten der Inkrementalgeberpulse

Da sich die in der Excel Datei abgespeicherten Geschwindigkeitswerte nicht in einer geradlinigen Kurve darstellen lassen, wird die Sprungantwort approximiert und ebenfalls in einem Diagramm dargestellt. Abb. 7.3 und Abb. 7.4 zeigen, dass bei den Geschwindigkeiten Schwingungen auftreten und somit keine konstante Endgeschwindigkeit erreicht werden kann. Aus diesem Grund wird bei der Approximation der Sprungantwort ein Durchschnittswert genommen, der alle Messwerte möglichst genau abbildet. Die maximale Durchschnittsgeschwindigkeit ergibt sich aus dem stationären Endwert der Approximationskurve. Dieser wird im zweiten Gang nach ca. 300 ms erreicht und kann durch eine Punktprobe berechnet werden (vgl. Abb. 7.3).

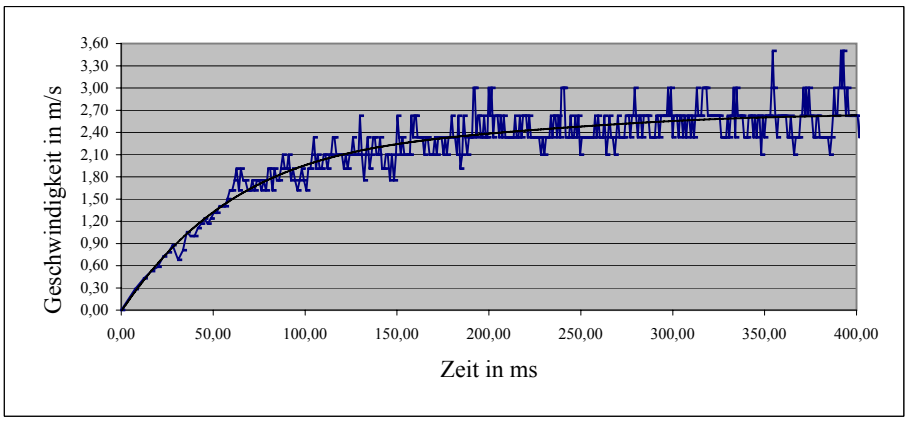

Abbildung 7.3.: Sprungantwort und Approximationskurve des Fahrzeuges im 2. Gang

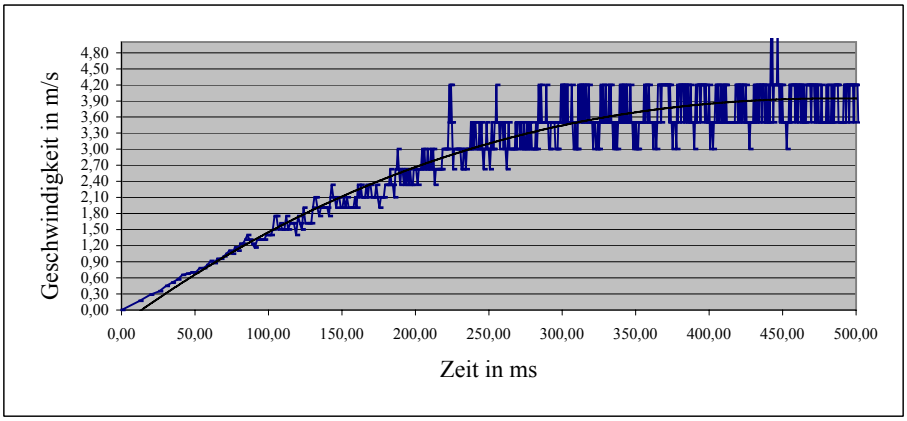

Abbildung 7.4.: Sprungantwort und Approximationskurve des Fahrzeuges im 3. Gang

Anhand der Approximationskurve werden die Systemparameter K_S und T_1 mit Hilfe des Tangentenverfahrens berechnet. Hierfür wird eine Tangente durch den Ursprung der Kurve gelegt. Der Schnittpunkt mit dem stationären Endwert ergibt die Zeitkonstante T_1. Nähere Informationen und die genaue Berechnung der Tangenten und der Schnittpunkte befinden sich auf der beiligenden CD.

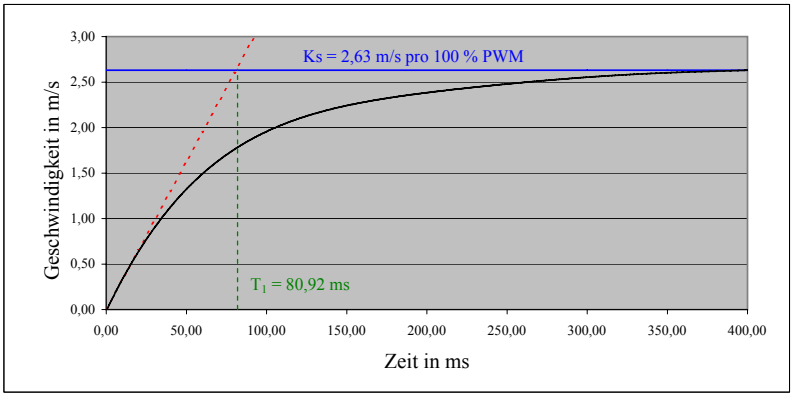

Abbildung 7.5.: Approximierte Sprungantwort mit Tangentenverfahren im 2. Gang

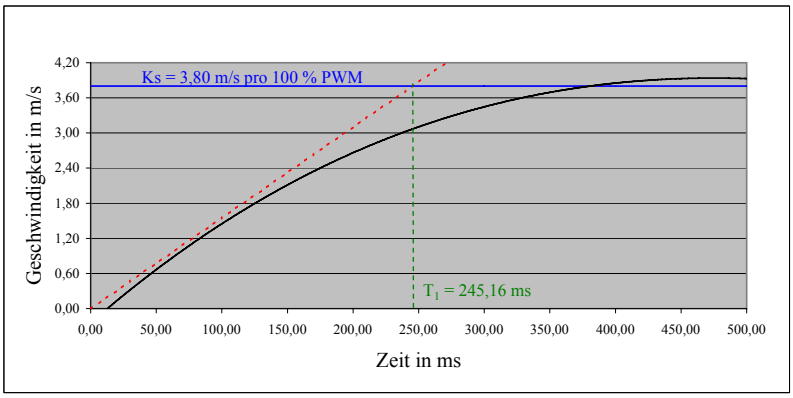

Abbildung 7.6.: Approximierte Sprungantwort mit Tangentenverfahren im 3. Gang

7.2. Maximale Pulsbreiten pro Gang

Zur Bestimmung der maximalen Pulsbreiten der Inkrementalgeberpulse und der daraus-folgenden maximalen Geschwindigkeit wurde bei der Aufnahme der Sprungantwort der stationäre Endwert gemessen. Wie in Kapitel 7.1 beschrieben, ist die Endgeschwindigkeit nicht konstant, sondern die Geschwindigkeitswerte schwingen. Aus diesem Grund wurde für die Berechnung der Einstellparameter des Reglers die Durchschnittsgeschwindigkeit benutzt und nicht die aus Abb.pulsbreite1Gang berechnete Geschwindigkeit.

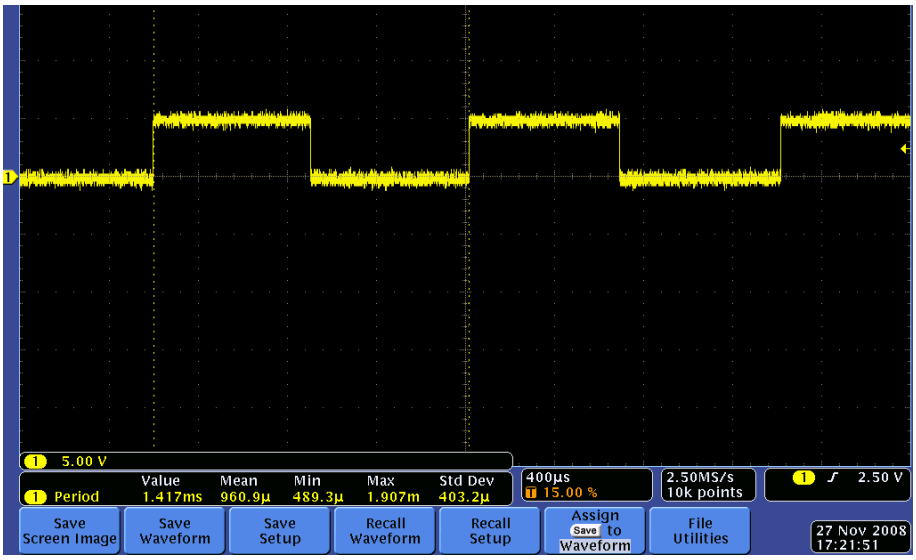

Abbildung 7.7.: Gemessene Pulsbreite des Inkrementalgebers im ersten Gang

Im ersten Gang beträgt die maximale Pulsbreite 1,417 ms (vgl. Abb. 7.7). Daraus ergibt sich mit der Strecke pro Puls eine Geschwindigkeit von;

$$V_{Max1} = \frac{Strecke\ pro\ Puls}{Pulsbreite} = \frac{2,175\ mm}{1,417\ ms} = 1,53 m/s \tag{7.2}$$

Durch die Schwingungen bei den Geschwindigkeiten kann nicht sichergestellt werden, dass die Messung die tatsächliche maximale Geschwindigkeit zeigt. Die Abweichung zur maximalen Durchschnittsgeschwindigkeit von 1,45 m/s, die zur Berechnung der Einstell-parameter genutzt wurde, beträgt lediglich 0,02 m/s. In Tabelle 7.2 werden die Pulsbreiten mit den darausfolgenden Geschwindigkeiten der restlichen Gänge gezeigt.

	Pulsbreite	**Max V**	**⌀ V**	**Abweichung**
1. Gang	1,417 ms	1,53 m/s	1,45 m/s	0,02 m/s
2. Gang	782,8 μ s	2,77 m/s	2,63 m/s	0,14 m/s
3. Gang	643,2 μ s	3,38 m/s	3,80 m/s	0,42 m/s

Tabelle 7.1.: Pulsbreiten der Inkrementalgeberpulskanäle

7.3. Simulation des geschlossenen Regelkreises

Zum Nachweisen der Modellierung wurde die Fahrzeugregelstrecke, die ebenfalls in ein zeitdiskretes System transformiert wurde (vgl. Abschnitt 5.6), in ein VHDL Modul integriert. Parallel zum PI-Regler Modul wurden die Parameter der Regelstrecke in eine Q-Format Darstellung konvertiert und als VHDL Generic abgelegt. Die Differenzengleichung wurde in einem getakteten Prozess modelliert, der am Ende die Ausgangsgröße *V_Speed* auf die maximale Geschwindigkeit pro Gang begrenzt (vgl. Code 7.1).

Listing 7.1: Begrenzung von V_Speed auf die maximale Geschwindigkeit

```
if(Gear = "01" and sub_1 >= "00010111001100") then
    sub_1 := "00010111001100";                    -- entspricht 1,45 m/s
elsif(Gear = "10" and sub_1 >= "00101010000101") then
    sub_1 := "00101010000101";                    -- entspricht 2,63 m/s
elsif(Gear = "11" and sub_1 >= "00111100110011") then
    sub_1 := "00111100110011";                    -- entspricht 3,80 m/s
end if;

v_speed_sig   <=   sub_1;
V_Speed       <=   std_logic_vector(v_speed_sig);
```

Bei diesem Test kamen das PWM Modul und der Pulszähler nicht zum Einsatz. Das PI-Regler Modul wurde so verändert, dass der Stellgrößenausgang *Control_PWM* nicht skaliert, sondern direkt mit 11 Bit an die Regelstrecke übergegeben wird. Diese berechnet, abhängig vom Gang, mit Hilfe der z-transformierten Übertragunsfunktion die entsprechende Geschwindigkeit *V_Speed* und sendet diese an den Reglereingang, der normalerweise den Zählerstand des Pulszählers empfängt. Anstatt die Geschwindigkeit zu berechnen, bekommt der Regler die Ist Geschwindigkeit von der Regelstrecke und bildet direkt die Regelabweichung (vgl. Abb. 7.8).

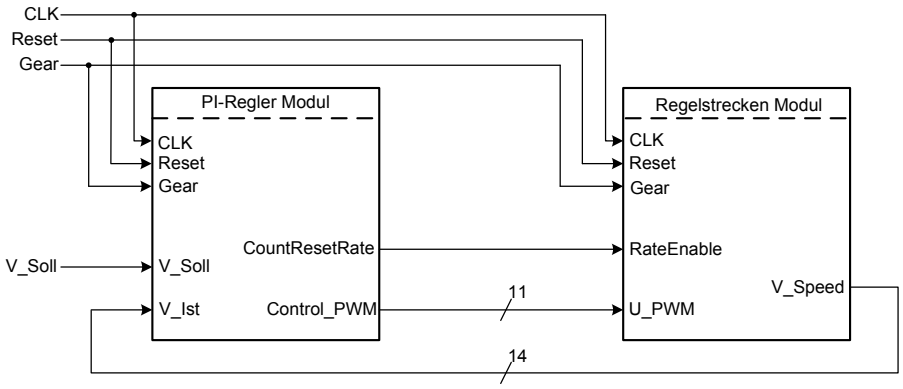

Abbildung 7.8.: Blockschaltbild für Simulation des geschlossenen Regelkreises

Die Simulation wurde zum Test der Einstellparametern sowohl mit K_{R1} und T_{I1} als auch mit K_{R2} und T_{I2} durchgeführt. Es ergab sich, dass bei allen Gängen die Wertepaare K_{R2} und T_{I2} die bessere Reaktion auf eine Regelabweichung aufweisen und die Soll Geschwindigkeit *V_Soll* von 1 m/s (entspricht im Q-Format = 1024 dezimal) exakt erreicht werden konnte (vgl. Abb. 7.9 unten). Mit den Wertepaare K_{R1} und T_{I1} konnte *V_Soll* in keinem Gang erreicht werden. Der Ausgang der Regelstrecke *V_Speed* hat jeweils eine Abweichung von 0,02 m/s (vgl. Abb. 7.9 oben). Dies hat zur Folge das die Regelabweichung *V_Diff* mit den Parametern K_{R1} und T_{I1} nicht vollständig beseitigt werden kann.

Der P-Anteil des Reglers, die sprungförmige Änderung auf eine Regelabweichung, ist zu Beginn in beiden Bilder gut zu erkennen. Da K_{R1} doppelt so groß ist wie K_{R2} erreicht der P-Anteil im oberen Bild die Begrenzung sofort und der Regler muss dies korrigieren. Dadurch kommt es zu einer größeren Ausregelungszeit als im unteren Bild.

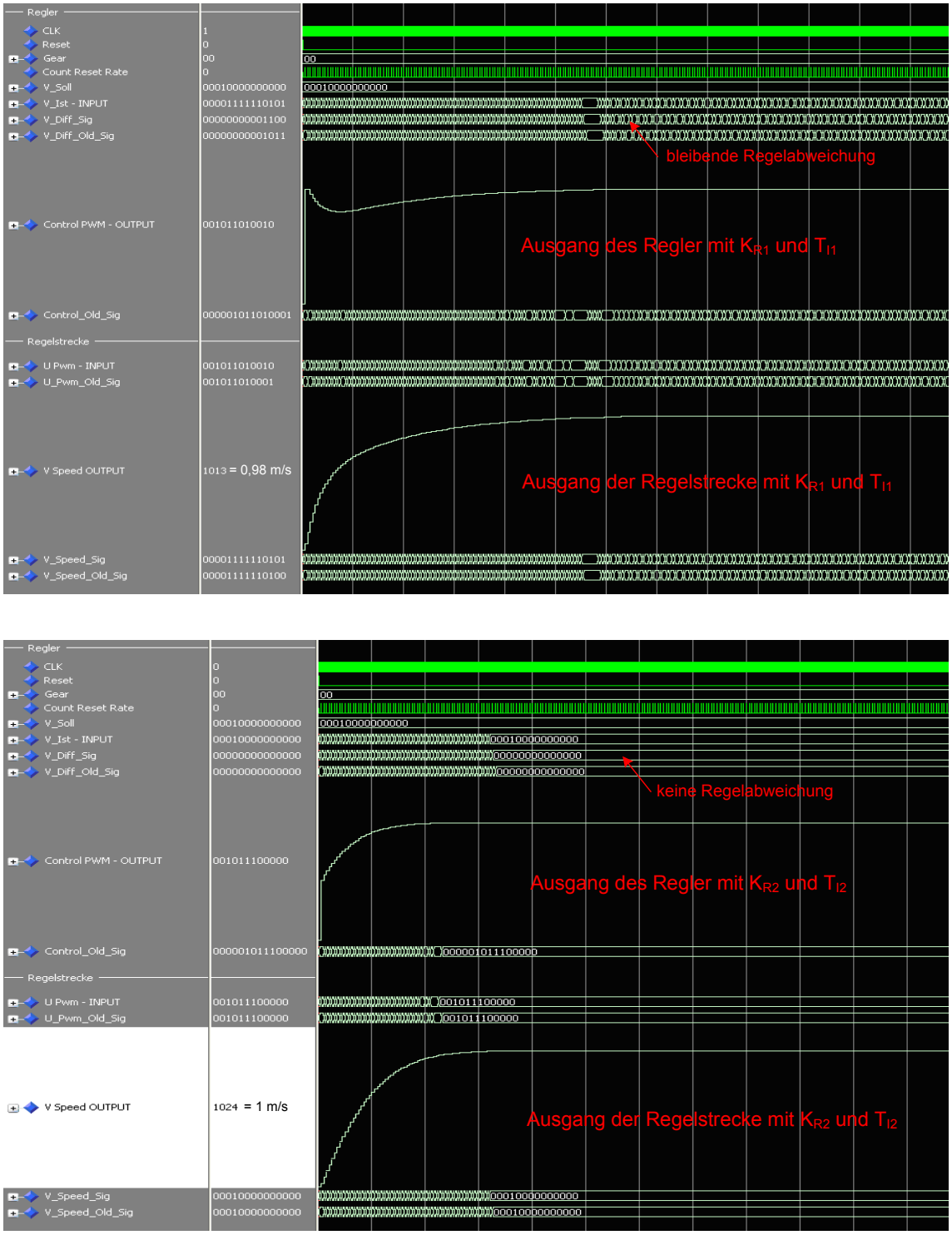

Abbildung 7.9.: Simulation des geschlossenen Regelkreises im 1. Gang. Oben mit K_{R1} und T_{I1} und unten mit K_{R2} und T_{I2}

7.4. Nachweis über die Funktionalität des SoC Systems

Eine Testfahrt mit dem Carolo Cup Fahrzeug war aus bautechnischen Veränderungen nicht möglich. Der zu Beginn dieser Arbeit eingesetzte Bürstenmotor mit 22.000 Umdrehungen wurde in der Zwischenzeit durch einen stärkeren und schnelleren Brushless Motor ausgetauscht. Außerdem wurde der gesamte Antriebsstrang umgebaut. Aus diesem Grund ist die in der messtechnischen Analyse gewonnene Erkenntniss, über die Fahrzeugregelstrecke, auf das alte Modell bezogen. Die Einstellparameter und die Sprungantworten des PT-1 Gliedes müssen neu berechnet und gegebenenfalls, bei Änderung des Verzögerungsgliedes, muss die Wahl des Reglers korrigiert werden.

Auf Grund dieser Gegebenheit wurde das Gesamtsystem mit einem PicoScope und einem manuellen Drehgeber getestet, der vor und zurück bewegt wurde. Der Drehgeber erzeugt zwei um 90° Grad versetzte Pulse, die an PMOD K12 und PMOD L17 angeschlossen werden und vom Pulszähler Modul ausgewertet werden. Über den implementierten Datenlogger werden die abgespeicherten Ist Geschwindigkeiten über die serielle Schnittstelle an den PC übertragen (vgl. Abb. 7.10).

```
V Ist: 0x00000000  →  0,00 m/s
V Ist: 0x0000029C  →  0,65 m/s
V Ist: 0x000000DE  →  0,22 m/s
V Ist: 0x00000459  →  1,08 m/s
V Ist: 0x0000029C  →  0,65 m/s
V Ist: 0x000001BD  →  0,43 m/s
V Ist: 0x00000B4E  →  2,82 m/s
        .....
```

Abbildung 7.10.: Ausgabe der Ist Geschwindigkeit über den Datenlogger

Der Datenlogger liest das gesamte Software Register 2, das 32 Bit groß ist, aus und überträgt byteweise den Geschwindigkeitswert an den PC. Dort können, beispielsweise über das Microsoft Hyperterminal, die Daten empfangen und dargestellt werden. Die Werte in Abb. 7.10 wurden vom manuellen Drehgeber erzeugt und dienen als Funktionstest für den Datenlogger und den Pulszähler. Da keine Rückführung der Stellgröße stattfindet, ist der Regelkreis nicht geschlossen. Die Pulse wurden manuell von Hand und nicht über das PWM Signal erzeugt.

Das an PMOD L15 angeschlossene PWM Signal wurde mit einem Picoscope 3204 getestet. Um den maximalen und den minimalen Duty Cycle des Signals zu messen, wurde über die MicroBlaze Software eine Soll Geschwindigkeit von 0 m/s und 1 m/s vorgegeben. Das Messergebniss ergab, für das Maximum, eine Zeit von 1,98 ms und für das Minimum eine Zeit von 1,516 ms (vgl. Abb. 7.11 und Abb. 7.12). Dies entspricht bis auf kleinste Abweichungen, den berechneten Werten für den Duty Cycle.

Der Einparkassistent, der im Rahmen des Projekts „High Performance Embedded Computing“ auf einem autonomen Fahrzeug, das auf FPGA-basierte System on Chip Plattformen (SoC) ausgerichtet ist, implementiert wird, wird durch einen Brushless Motor mit Hall Sensor angetrieben. Für den Motor ist eine messtechnische Analyse zur Identifikation der Regelstrecke vorzunehmen.

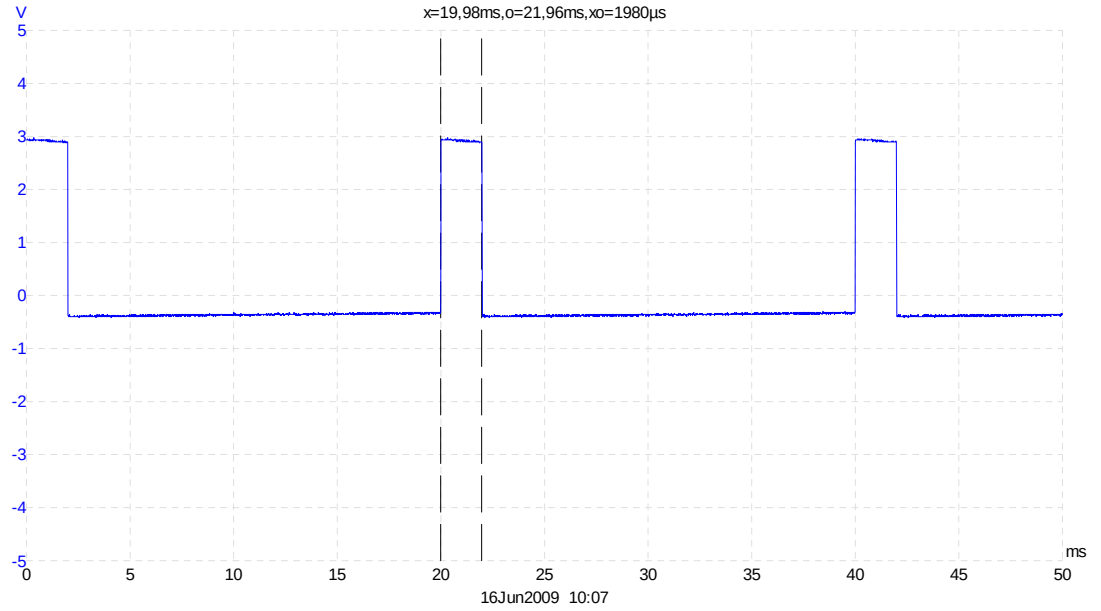

Abbildung 7.11.: Messung des maximalen PWM Signals mit einem Duty Cycle von 2 ms und einer Periode von 20 ms an PMOD L15 mit einem PicoScope

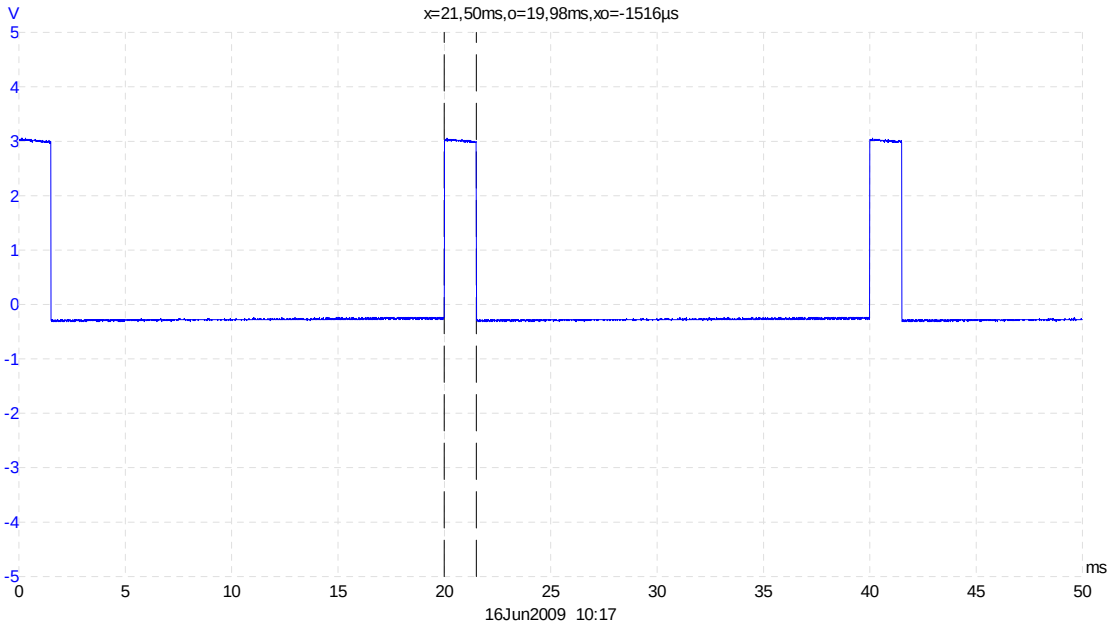

Abbildung 7.12.: Messung des minimalen PWM Signals mit einem Duty Cycle von 1,5 ms und einer Periode von 20 ms an PMOD L15 mit einem PicoScope

7.5. Optimierung der zweiten Pipelinestufe durch Ressource Sharing

Durch die sechs Multiplizierer in der zweiten Pipelinestufe können die FPGA Ressourcen des Spartan 3Es schnell knapp werden (vgl. Abb. 6.6). Durch Umstrukturierung von Code 6.7 kann die Anzahl von sechs auf zwei reduziert werden. Dabei werden die Multiplikationen von *v_diff_sig* und *v_diff_old_sig* mit den Einstellparametern erst nach den Multiplexern durchgeführt. Diese bestimmen, abhängig vom Gang, den Multiplikator für *mul1* und *mul2*. Zwei neu hinzugekommene Variablen *param1* und *param2* stellen die Daten bereit (vgl. Code 7.2). Somit werden nicht die Ausgänge sondern die Eingänge der Multiplizierer gemultiplexed.

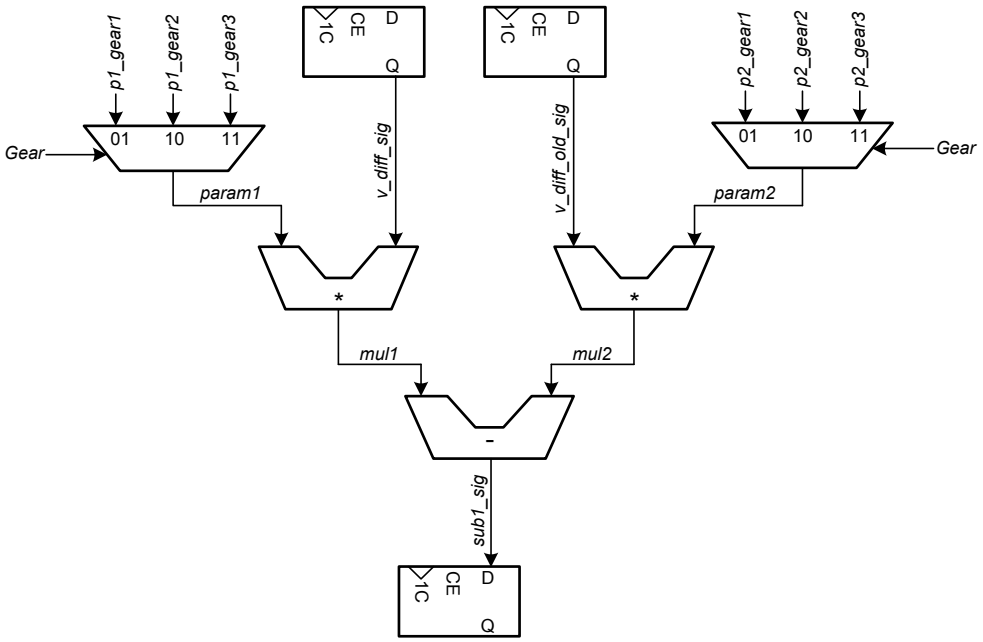

Abbildung 7.13.: Optimierung der zweiten Pipelinestufe durch Ressource Sharing

Im Vergleich zu Abb. 6.6 zeigt Abb. 7.13 ledeglich zwei Multiplizierer und zwei Multiplexer. Das Synthesewerkzeug inferriert bei eingeschaltener „Speed“ oder „Area“ Optimierung ein ROM, welches die Parameter als Daten enthält und den codierten Gang als Adresse verwendet.

Listing 7.2: Reduzierung der Multiplizierer der zweiten Pipelinestufe

```
case Gear is
    when "01"   => param1 := param1_gear1;
                   param2 := param2_gear1;
    when "10"   => param1 := param1_gear2;
                   param2 := param2_gear2;
    when "11"   => param1 := param1_gear3;
                   param2 := param2_gear3;
    when others => param1 := (others => '0');
                   param2 := (others => '0');
end case;

mul_1_var := param1 * v_diff_sig;
mul_2_var := param2 * v_diff_old_sig;
sub_sig <= mul_1_var(24 downto 10) - mul_2_var(24 downto 10);
```

7.6. Ressourcenbedarf

Bedarf an FPGA Ressourcen für das Gesamtsystem

Die verwendeten FPGA Ressourcen beziehen sich auf das Spartan 3E FPGA der Version 3s1200efg320-4 und wurde vom EDK berechnet. Bei der Synthese war die Optimierung auf Geschwindigkeit aktiviert. Der längste Signallaufzeitpfad beträgt 12,456 ns benötigt. Diese minimale Taktperiode weicht von der in Abschnitt 6.5 berechneten Zeit ab. Der Grund hierfür ist, dass der längste Signallaufzeitpfad sich bei dem Gesamtsystem im Debug Modul befindet und nicht in dem implementierten Geschwindigkeitsregler. Die maximale Taktfrequenz des Systems beträgt somit 80,283 MHz.

	Verwendet	**Vorhanden**	**Prozent**
Anzahl an Slices	2582	8672	29%
Anzahl an Flip Flops	3199	17344	18%
Anzahl an LUTs	3716	17344	21%
Anzahl an IOBs	87	250	34%
Anzahl an BRAMs	4	28	14%
Anzahl an MULT	11	28	29%

Tabelle 7.2.: Ressourcenbedarf des Gesamtsystems in einem Spartan 3E 1200

Bei Optimierung der zweiten Pipelinestufe durch Ressource Sharing minimiert sich die Anzahl der Multiplizierer MULT von elf auf sieben. Ebenfalls wird die Anzahl der Lookup Tabellen LUTS um 55 Stück reduziert.

Größe des Software Modules *V_Regler_App*

Da die Software komplett im 16 MB großen Micron Ram auf dem Board gespeichert wird, stehen hier ausreichend Ressourcen zur Verfügung. Für die Weiterentwicklung oder der Verbesserungen der Software ist noch ausreichend Platz vorhanden.

Größe der Software *V_Regler_App*		
.text	6910 Bytes	ausführbarer Programmcode
.data	364 Bytes	initialisierte Variablen
.bss	2090 Bytes	nichtinitialisierte globale Variablen
Gesamt	9364 Bytes	

Tabelle 7.3.: Größe der MicroBlaze Software *V_Regler_App*

Die Software würde aufgrund der Größe ebenso in das BRAM passen. Jedoch wurde davon abgesehen, damit der Datenlogger nahezu beliebig viele Messwerte speichern und ausgeben kann. Damit das Programm, das im externen SRAM liegt, schneller ausgeführt werden kann wurde ein zusätzlicher Instruktions Cache *microblaze_0_IXCL* verwendet. Diese verbindet über den „Xilinx Cache Link (XCL)“ den SRAM mit dem MicroBlaze.

8. Zusammenfassung

Im Rahmen des Projekts Fahrerassistenz- und Autonome Systeme (FAUST) wurde in dieser Arbeit eine FPGA basierte Geschwindigkeitsregelung für ein autonomes Fahrzeug auf einer System on Chip Platform implementiert. Die Regelung soll ein unterlagertes Modul der Fahrzeug-Bahnführung darstellen und kommt in einem Einparkassistenten zum ersten Einsatz. Integriert in einem MicroBlaze Bussystem wird der V-Regler IP Core an den PLB Bus angeschlossen. Über Software Register tauscht der V-Regler Daten mit dem Prozessor und dem Fahrzeugmodell IP aus.

Bei einer messtechnischen Analyse des Carolo Cup Fahrzeuges, wurden die Sprungantworten der Fahrzeugregelstrecke mit einem Speicheroszilloskop und einer Java-Applikation aufgezeichnet und approximiert. Die Identifikation ergab ein System erster Ordnung, ein PT-1 Glied. Mit dem Wendetangentenverfahren wurden die Zeitkonstante T_1 und der Verstärkungsfaktor K_S berechnet. Als Reglerstruktur wurde ein PI-Regler, dessen Zeitkonstante T_I und Verstärkungsfaktor K_R mit dem Quadratischen Gütekriterium bestimmt worden ist, gewählt. Die zeitkontinuierliche Übertragungsfunktion $G_R(s)$ des PI-Reglers wurde mit der z-Transformation in eine zeitdiskrete Differenzengleichung transformiert. Hierbei kam als Äquivalenzbeziehung sowohl die Bilinearen Transformation als auch das Euler Verfahren zum Einsatz.

Zur Bestimmung der Ist Geschwindigkeit erzeugt ein in der Vorderradfelge des Fahrzeuges angebrachter Inkrementalgeber zwei um 90° versetzte Pulse. Diese werden in einem Pulszähler mit einer FSM gezählt und ausgewertet. Durch die Phasenverschiebung der Pulse wird in der FSM die Drehrichtung erkannt und die Freigabe für den Counter gesetzt. Der Zählerstand wird dem PI-Regler übermittelt und bei Erreichen der Abtastperiode zurückgesetzt. Parallel zum Pulszähler wurde ein zweiter Counter implementiert, der bei Freigabe des Fahrzeugmodell IPs, die Pulse zur Parklückenbemessung mitzählt.

Ein im PI-Regler Modul integrierter Abtastratentimer erzeugt alle 20 ms einen Interrupt und setzt das initiale Freigabesignal für die Reglerpipeline. In Delayregistern wird das Freigabesignal für die Folgestufen gespeichert und verzögert. Der Interrupt wird zur globalen Zeitgebung des Gesamtsystems genutzt und vom MicroBlaze verarbeitet.

Der in einer dreistufigen Pipeline implementierte PI-Regler berechnet, bei Freigabe durch den Abtastratentimer, aus den gezählten Pulsen die Ist Geschwindigkeit und die Regelabweichung. Über Pipelineregister, die die Stufen entkoppeln, wird das Zwischenergebniss an die Folgestufe weitergereicht und verarbeitet. Für die Modellierung der Differenzengleichung wird die Regelabweichung $e(k)$ in einem Delayregister und die Stellgröße $u(k)$ in einem Rückführungsregister gespeichert. Ein Anti-Windup, der den I-Anteil des Reglers begrenzt, erzeugt die Freigabe für das Rückführungsregister. Die letzte Pipelinestufe begrenzt und skaliert die Stellgröße und gibt diese über einen synchonisierten Ausgang

an das PWM Modul. Zum Ansteuern des Fahrtenstellers erzeugt das PWM Modul eine 20 ms große Periode mit einem Duty Cycle zwischen 10% und 20%.

Zum Aufzeichnen von Messdaten während einer Testfahrt wurde eine Datenlogger Funktion implementiert, die die Ist Geschwindigkeit und die Stellgröße im Micron SRAM speichert und anschließend über die serielle Schnittstelle an den PC überträgt.

Die Simulation des geschlossenen Regelkreises ergab, dass bei passender Wahl der Einstellparameter, die Regelabweichung vollständig beseitigt werden kann und die Soll Geschwindigkeit erreicht wird. Zum Nachweis der Funktionalität des Gesamtsystems wurde das Geschwindigkeitsregelystem mit einem manuellen Drehgeber und einem Picoscope getestet. Über den Datenlogger wurden die aufgezeichneten Messdaten an den PC übertragen und mit dem Hyperterminal ausgewertet.

Glossar

Ausregelzeit

Ist die Zeitspanne, wenn nach einer sprungförmigen Änderung der Eingangsgröße die Regelgröße einen vorgegebenen Toleranzbereich verlässt und diesen wieder zum dauernden Verbleib betritt. Für die Regelung sollte diese Zeit möglichst gering sein [5].

Kontinuierliches Signal

Kontinuierliche Signale sind analoge Signale, die jeden Wert zwischen einem Minimum und einem Maximum annehmen können. Zu jedem bestimmtem Zeitpunkt kann dem Signal ein eindeutiger Wert zugewiesen werden, die sogenannte Zeitkontinuität.

Quantisierungsfehler

Bei der Umwandlung von analogen Signalen in digitale Signalen entstehen Quantisierungsfehler. Diese entstehen durch Rundungsfehler, die durch die endliche Genauigkeit von Dezimalbrüchen bei der Darstellung von digitalen Signalen hervorgerufen werden.

Regelgüte

Die Regelgüte beschreibt das Verhalten der Regelgröße, die durch den Sollwert und durch Störgrößen beeinflusst wird, im Regelkreis. Anhand des Gütemasses, welches den zeitlichen Verlauf der Regelgröße und der Stellgröße beschreibt, kann eine Aussage über die Qualität der Regelung getroffen werden. Für eine optimale Regelung muss das Gütemaß minimiert werden. Hierfür gibt es verschiedene Kriterien bzw. Normen, wie z.B. das Betragskriterium, das ITAE Kriterium oder das Quadratisches Gütekriterium [23].

Überschwingweite

Die Überschwingweite bezeichnet die größte vorübergehende Abweichung der Regelgröße vom Sollwert nach einer sprungförmigen Änderung der Eingangsgröße / Führungsgröße. Bei der Regelung darf sie eine bestimmte Toleranzgrenze nicht überschreiten.

Übertragungsfunktion

Übertragungsfunktionen beschreiben in einem Regelsystem das Verhalten von Eingangsgröße zu Ausgangsgröße. Der zeitliche Verlauf der Ausgangsgröße kann eindeutig bestimmt werden [5].

Verzugszeit

Die Verzugszeit ist die Zeit, die vergeht, bis die Stellgröße nach einer sprungförmigen Änderung der Eingangsgröße, beginnt sich merklich zu ändern. Bei Systemen erster Ordnung (PT-1 Glied) ist die Verzugszeit gleich null.

Wendetangente

Eine Wendetangente ist eine Tangente, die durch den Wendepunkt einer Funktion geht. Sie wird beschrieben durch die allgemeine Geradengleichung:

$$y(x) \; = \; mx \; + \; b \tag{8.1}$$

Zur Bestimmung der Wendetangente muss der Wendepunkt der Funktion bestimmt werden. Anschließend berechnet man durch die erste Ableitung die Steigung *m* im Wendepunkt. Die Formel für die Wendetangente lautet:

$$t(x) \; = \; f'(x_0) \; * \; (x - x_0) \; + \; f(x_0) \tag{8.2}$$

Der Schnittpunkt mit der y-Achse wird in der Geradengleichung durch die Variable *b* dargestellt.

Zeitdiskretes Signal

Ein zeitdiskretes Signal besteht aus einer periodischen Folge von einzelnen Impulsen, die den Verlauf über die Zeit darstellen. Der Wert des Signals wird nur zu bestimmten Zeitpunkten betrachtet, den sogenannten Abtastwerten. Zwischen zwei Abtastwerte kann der Signalwert nur mit bestimmten Vorrausetzungen, die in einem Abtasttheorem beschrieben sind, berechnet werden [9]. Wenn die Zeitspanne zwischen zwei Abtastwerten sehr gering ist, spricht man von einem quasikontinuierlichem Signal.

Abkürzungsverzeichnis

FPGA	Field Programmable Gate Array
ACC	Adaptive Cruise Control
ASIC	Application Specific Integrated Circuit
CLK	Clock (Taktsignal)
FAS	Fahrerassistenzsysteme
FAUST	Fahrerassistenz- und Autonome Systeme
FIFO	First In First Out
FSM	Finite State Machine
GPIO	General Purpose Input/Output
IP	Intellectual Property
IRQ	Interrupt Request
LUT	Look Up Table
PWM	Pulsweiten Modulation
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SoC	System on Chip
UART	Universal Asynchronous Receiver Transmitter
VHDL	Very High Speed Integrated Circuit Hardware Description Language

$e(t)$	Regelabweichung
f_{max}	maximale Taktfrequenz des Systems
$G_R(s)$	kontinuierliche Übertragungsfunktion des Reglers
$G_R(z)$	zeitdiskrete Übertragungsfunktion des Reglers
$G_S(s)$	kontinuierliche Übertragungsfunktion der Regelstrecke
$G_S(s)$	zeitdiskrete Übertragungsfunktion der Regelstrecke
I_q	Quadratische Regelfläche
K_P	Verstärkungsfaktor des Reglers
K_S	Proportionalbeiwert oder Verstärkungsfaktor der Regelstrecke
s_{Pulse}	Strecke pro Puls
T	Abtastperiode
T_1	Zeitkonstante der Regelstrecke
T_I	Zeitkonstante oder Integrationsbeiwert des Reglers
$u(t)$	Stellgröße (Eingang der Regelstrecke)
V_{Ist}	Ist Geschwindigkeit des Fahrzeuges
V_{Soll}	Soll Geschwindigkeit
$x(t)$	Messwert nach dem Messglied
$y(t)$	Regelgröße (Ausgang der Regelstrecke)
$z(t)$	Störgröße am Eingang der Regelstrecke

Literaturverzeichnis

[1] CUP, Carolo: *Carolo Cup - Studenten entwickeln autonome Modellfahrzeuge*. 2008. – URL http://www.carolo-cup.de/. – abgerufen 03.04.09

[2] DARPA: *DARPA Urban Challenge*. 2007. – URL http://www.darpa.mil/grandchallenge/index.asp. – abgerufen 03.04.09

[3] DIGILENT, Inc.: *Nexys-2*. 2008. – URL http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=451&Prod=NEXYS2. – abgerufen 07.04.09

[4] GAJSKI, Daniel D.: *Principles of Digital Design*. Prentice Hall, 1997. – ISBN 0-13-301144-5

[5] GROSSE, Norbert ; SCHORN, Wolfgang: *Taschenbuch der praktischen Regelungstechnik*. Fachbuchverlag Leipzig, 2006. – ISBN 3-446-40302-7

[6] HAW, Hamburg: *FAUST - Autonomes Fahren - Carolo Cup*. 2009. – URL http://www.informatik.haw-hamburg.de/2141.html. – abgerufen 03.04.09

[7] HOFFMANN, Dirk W.: *Theoretische Informatik*. Hanser Verlag, 2009. – ISBN 3-446-41511-4

[8] ISE, International Submarine Engineering L.: *Explorer Autonomous Underwater Vehicle*. 2008. – URL http://www.ise.bc.ca/Explorer.html. – abgerufen 30.03.09

[9] KIENCKE, Uwe ; JÄKEL, Holger: *Signale und Systeme*. Bd. Edition 4. Oldenbourg Verlag, 2008. – ISBN 978-3-486-58734-0

[10] KOPF, Matthias: *Fahrerassistenzsysteme mit maschineller Wahrnehmung*. Springer Berlin Heidelberg, 2005. – ISBN 978-3-540-23296-4

[11] ORLOWSKI, Peter F.: *Praktische Regeltechnik - Anwendungsorientierte Einführung für Maschinenbauer und Elektrotechniker*. Bd. Edition 6. Springer, 2007. – ISBN 3-540-68358-5

[12] PHILIPPSEN, Hans-Werner: *Einstieg in die Regelungstechnik - Vorgehensmodell für den praktischen Reglerentwurf*. Fachbuchverlag Leipzig, 2004. – ISBN 3-446-22377-0

[13] REICHARDT, Jürgen ; SCHWARZ, Bernd: *VHDL-Synthese - Entwurf digitaler Schaltungen und Systeme*. Bd. 4. Auflage. Oldenbourg Verlag, 2007. – ISBN 978-3-486-58192-8

[14] SCHULZ, Gerd: *Regelungstechnik 2 - Mehrgößenregelung, Digitale Regelungstechnik, Fuzzy Regelung*. Bd. 2. Auflage. Oldenbourg Verlag, 2008. – ISBN 978-3-486-58318-2

[15] TAMIYA: *Super Stock Motor TZ*. 2008. – URL http://www.tamiya.de/de/produkte/ersatztuningteilezubehoer/motorenundregler/produktdetails.htm?sArtNr=53696. – abgerufen 07.04.09

[16] TARTAN, Racing: *Carnegie Mellon - Tartan Racing*. 2007. – URL http://www.tartanracing.org/. – abgerufen 03.04.09

[17] THOMAS, D.E. ; LAGNESE, E.D. ; WALKER, R.A. ; NESTOR, J.A. ; RAJAN, J.V. ; BLACKBURN, R.L.: *Algorithmic and Register-Transfer Level Synthesis: The System Architect´s Workbench*. Springer Verlag, 1990. – ISBN 978-0-7923-9053-4

[18] UNBEHAUEN, Heinz: *Regelungstechnik 1 - Klassische Verfahren zur Analyse und Synthese linearer kontinuierlicher Regelsysteme*. Bd. 13. Auflage. Vieweg, 2005. – ISBN 3-528-21332-9

[19] UNBEHAUEN, Heinz: *Regelungstechnik 2 - Zustandsregelungen, digitale und nichtlineare Regelsysteme*. Bd. 9. Auflage. Vieweg, 2007. – ISBN 978-3-528-83348-0

[20] VDI, Wissensforum: *Integrierte Sicherheit und Fahrerassistenzsysteme*. Bd. 24. VDI/VW - Gemeinschaftstagung. VDI Verlag GmbH, 2008. – ISBN 973-3-18-092048-1

[21] VISHAY: Subminiature Dual Channel Transmissive Optical Sensor with Phototransistor Outputs. In: *Datasheet TCUT1300X01* (2007), Nr. 2.3

[22] WIKIPEDIA: *Fahrerassistenzsystem — Wikipedia, Die freie Enzyklopädie*. 2009. – URL http://de.wikipedia.org/wiki/Fahrerassistenzsystem. – abgerufen 11. April 2009

[23] WIKIPEDIA: *Regelgüte — Wikipedia, Die freie Enzyklopädie*. 2009. – URL http://de.wikipedia.org/wiki/Regelg%C3%BCte. – abgerufen 22. Juni 2009

[24] WIKIPEDIA: *System on a Chip — Wikipedia, Die freie Enzyklopädie*. 2009. – URL http://de.wikipedia.org/wiki/SystemonaChip. – abgerufen 09. April 2009

[25] XILINX: Embedded System Tools Reference Manual. In: *Embedded Development Kit* (2002), Nr. EDK 10.1, Servie Pack 3

[26] XILINX: Fast Simplex Link (FSL) Bus. In: *Product Specification* (2008), Nr. 2.11a

[27] XILINX: MicroBlaze Processor Reference Guide. In: *Embedded Development Kit EDK 10.1i* (2008), Nr. 9.0

[28] XILINX: Processor Local Bus (PLB) v4.6. In: *Product Specification* (2008), Nr. 1.03a

[29] XILINX: Xilinx ISE 10.1 Design Suite Software Manuals and Help. In: *PDF Collection* (2008)

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

A. Fahrzeugregelstrecke als digitales Modell

Die kontinuierliche Übertragungsfunktion $G_S(s)$ des PT-1 Gliedes (vgl. Gleichung 5.2) wurde für die Simulation des geschlossenen Regelkreises mit der z-Transformation in eine Differenzengleichung transformiert (vgl. Abschnitt 5.6). Hierfür wurde die Bilineare Transformation (Trapezintegration) verwendet.

$$\frac{Y}{U} = G_S(s) = \frac{K_S}{1 + T_1 s} \quad \Rightarrow \quad s = \frac{2}{T}\frac{z-1}{z+1} \quad \Rightarrow \quad G_S(z) = \frac{K_S}{1 + \frac{2T_1}{T}\frac{z-1}{z+1}}$$

Die Herleitung der Differenzengleichung für die Fahrzeugregelstrecke mit der zeitdiskreten Übertragungsfunktion $G_S(z)$ lautet wie folgt:

$$G_S(z) = K_S * \frac{z+1}{z\left(1 + \frac{2T_1}{T}\right) + \left(1 - \frac{2T_1}{T}\right)} * \left(\frac{z^{-1}}{z^{-1}}\right)$$

$$G_S(z) = K_S * \frac{1 + z^{-1}}{\left(1 + \frac{2T_1}{T}\right) + \left(1 - \frac{2T_1}{T}\right) z^{-1}} = \frac{y(z)}{u(z)}$$

$$y(z)\left(\left(1 + \frac{2T_1}{T}\right) + \left(1 - \frac{2T_1}{T}\right) z^{-1}\right) = K_S u(z)(1 + z^{-1})$$

$$a_0 y(k) + a_1 y(k-1) = K_s u(k) + K_S u(k-1)$$

$$y(k) = -\frac{a_0}{a_1} y(k-1) + K_S \frac{1}{a_0} u(k) + K_S \frac{1}{a_0} u(k-1)$$

$$a_0 = 1 + \frac{2T_1}{T} \quad a_1 = 1 - \frac{2T_1}{T} \quad b_0 = b_1 = 1$$

B. CD: VHDL-, C- und Java-Quellcode sowie EDK und ModelSim Projektdaten

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 2. Juli 2009	
Ort, Datum	Unterschrift