Entwurf und Implementierung rekonfigurierbarer Controller für mechatronische Systeme

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur

(Dr.-Ing.)

von Dipl.-Ing. Steffen Toscher

geb. am 8. Juli 1979 in Schwerin

genehmigt durch die Fakultät für Maschinenbau der Otto-von-Guericke-Universität Magdeburg

Gutachter:

Prof. Dr.-Ing. Roland Kasper

Prof. Dr. rer. nat. Georg Rose

Promotionskolloquium am 15. Dezember 2008

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Mobile Systeme der Otto-von-Guericke-Universität Magdeburg. Ich möchte allen danken, die zum Gelingen beigetragen haben. Mein besonderer Dank gilt Prof. Dr.-Ing. Roland Kasper. Er gab mir die Gelegenheit, diese Arbeit am Lehrstuhl für Mechatronik durchzuführen und hat sie durch anregende Diskussionen und wichtige Ratschläge begleitet. Mein Dank gilt auch Dr.-Ing. Thomas Reinemann für Unterstützung, Ratschläge und Anregungen.

Prof. Dr. rer. nat. Georg Rose danke ich für wertvolle Hinweise und die Erstellung des Zweitgutachtens.

Zusammenfassung

Heutige Standardimplementierungsplattformen für digitale Steuer- und Regelfunktionalitäten mechatronischer Systeme, wie Mikrocontroller und Signalprozessoren, basieren auf einer festen Hardwarearchitektur und der sequentiellen oder eingeschränkt parallelen Abarbeitung von Software. Moderne rekonfigurierbare Hardware stellt jedoch eine hohe Rechenleistung, harte Echtzeitfähigkeiten und eine allgemeine Hardwareprogrammierbarkeit zur Verfügung. Die grundlegende Strukturvariabilität rekonfigurierbarer Hardware ermöglicht darüber hinaus eine flexible Umsetzung digitaler Steuer- und Regelfunktionalitäten.

Die vorliegende Arbeit behandelt den Entwurf und die Implementierung rekonfigurierbarer Controller für mechatronische Systeme auf der Grundlage moderner rekonfigurierbarer FPGAs. Dabei wird die partielle Rekonfiguration als Möglichkeit genutzt, die Funktionalität der entworfenen Controller dynamisch an veränderte Hardwareanforderungen anzupassen. Die Arbeit stellt eine Entwurfsmethodik vor, die eine fachnahe Spezifikation und Umsetzung der partiellen Rekonfiguration ermöglicht. Der Entwurf strukturvariabler Funktionen wird weiterhin durch ein implizites und echtzeitfähiges verteiltes Rekonfigurationsmanagement unterstützt. Darüber hinaus werden Infrastrukturkomponenten und Funktionsgruppen entwickelt, die den Einsatz rekonfigurierbarer Hardware in mechatronischen Systemen unterstützen und vereinfachen. Die Anwendbarkeit und Leistungsfähigkeit der vorgestellten Methoden und Funktionsgruppen wird mit ausgewählten Beispielanwendungen nachgewiesen.

Inhaltsverzeichnis

1 Einleitung

Die Anforderungen an moderne Elektronikbausteine und den daraus aufgebauten informationsverarbeitenden Komponenten mechatronischer Systeme haben in den letzten Jahren und Jahrzehnten stark zugenommen. Anwendungen, zum Beispiel im Fahrzeug, erfordern eine hohe Rechenleistung und harte Echtzeitfähigkeit bei gleichzeitig kostengünstigen Implementierungsmöglichkeiten. Heutige Standardimplementierungsplattformen für digitale Steuer- und Regelfunktionalitäten mechatronischer Systeme, wie Mikrocontroller und Signalprozessoren, basieren auf einer festen Hardwarearchitektur und der sequentiellen oder eingeschränkt parallelen Abarbeitung von Software. Dieser Ansatz ermöglicht eine hohe Flexibilität in Bezug auf in Software implementierte Funktionen, da diese jederzeit an neue Anforderungen angepasst und verändert werden können. Aufgrund ihrer festen Hardwarearchitektur sind Mikrocontroller und Signalprozessoren jedoch nicht an sich verändernde Hardwareanforderungen anpassbar, zum Beispiel bei verändertem Rechen-, Kommunikations- oder I/O-Bedarf. Weiterhin wird die Rechenleistung und das Echtzeitverhalten von Mikrocontrollern, Universal- und Signalprozessoren durch die sequentielle Arbeitsweise limitiert, da nur eine begrenzte Anzahl von Verarbeitungselementen zur Verfügung steht. Bei mechatronischen Systemen, die hochdynamische Signalverarbeitung beinhalten, ist so die Anzahl der pro Tastperiode durchführbaren Operationen eingeschränkt.

Der Einsatz rekonfigurierbarer Hardware, insbesondere von *Field Programmable Gate Arrays* (FPGAs), für die Implementierung von Steuer- und Regelfunktionen in mechatronischer Systemen stellt einen Lösungsansatz für diese Probleme dar. Rekonfigurierbare Hardware ermöglicht eine parallele Informationsverarbeitung direkt in der Hardware und so eine sehr hohe Rechenleistung in Zusammenhang mit harter Echtzeitfähigkeit. Neben der hohen Rechenleistung stellen programmierbare Logikbausteine wie FPGAs sehr große Kommunikations- und I/O-Leistungen zur Verfügung. Eine weitere wesentliche Eigenschaft rekonfigurierbarer Hardware ist die Anpassung an veränderliche Hardwareanforderungen eines mechatronischen

Systems. Durch eine Rekonfiguration der Hardware kann so neuen Anforderungen an Funktionalität, Rechenleistung und Schnittstellen im Entwicklungsprozess oder Produktlebenszyklus entsprochen werden. Eine solche Anpassung erfolgt im Normalfall durch eine Umprogrammierung der kompletten Hardware eines Bausteins außerhalb des regulären Betriebs. Während der Laufzeit eines auf diese Weise implementierten Controllers sind alle benötigten Hardwarekomponenten nebeneinander auf dem Baustein realisiert, auch wenn diese nie gleichzeitig benutzt werden. Eine wesentlich größere Flexibilität und eine bessere Ausnutzung der Hardwareressourcen bieten auf modernen FPGAs implementierte partiell und dynamisch rekonfigurierbare Controller. Sie ermöglichen eine dynamische Anpassung von Rechen- und Kommunikationsleistung, Steuerungs- und Regelungsfunktionen sowie Schnittstellen an den jeweiligen Betriebszustand während der Laufzeit des Controllers. Für die Echtzeitfähigkeit eines so implementierten Controllers ist entscheidend, dass die partiellen Konfigurationsänderungen der Hardware unter Einhaltung von Echtzeitbedingungen ablaufen.

Der Entwurf und die Implementierung solcher partiell und dynamisch rekonfigurierbaren Controller für mechatronische Systeme ist Gegenstand der vorliegenden Arbeit. Sie gliedert sich wie folgt:

Kapitel 1 gibt neben der allgemeinen Einleitung einen Überblick über rekonfigurierbare Hardware und geht speziell auf FPGAs ein. *Kapitel 2* stellt aktuelle Entwurfsverfahren für Steuerungs- und Regelungseinrichtungen im Bereich der Mechatronik vor und erläutert den Einsatz rekonfigurierbarer Hardware in der Mechatronik. Ein weiterer Schwerpunkt sind Grundlagen, Werkzeuge und Anwendungen der partiellen Rekonfiguration moderner FPGAs. Die der vorliegenden Arbeit zugrunde liegende neue Entwurfsmethodik für rekonfigurierbare Controller in mechatronischen Systemen wird in *Kapitel 3* beschrieben. Dabei wird auf Entwurfsverfahren, Partitionierung und ein verteiltes echtzeitfähiges Rekonfigurationsmanagement eingegangen. In *Kapitel 4* werden die entwickelte Struktur und Funktionsgruppen für rekonfigurierbare Controller in der Mechatronik dargestellt. Ausgehend von der Struktur rekonfigurierbarer Controller beschreibt *Kapitel 5* die Implementierung auf der Zielhardware und geht insbesondere auf die entwickelten echtzeitfähigen Rekonfigurationsmechanismen ein. Schließlich werden in *Kapitel 6* Anwendungen rekonfigurierbarer Controller in mechatronischen Systemen vorgestellt, die auf den Ergebnissen der Arbeit basierend entworfen und implementiert wurden. Schwerpunkte sind ein rekonfigurierbarer Antriebscontroller für DC-Antriebe und ein rekonfigurierbarer Controller für die Ansteuerung piezo-elektrischer Aktoren.

1.1 Grundlagen rekonfigurierbarer Hardware

Die Realisierung komplexer Algorithmen in der Digitaltechnik kann grundsätzlich auf vielen verschiedenen Architekturen erfolgen. Wesentliche Auswahlkriterien hierbei sind die für die Anwendung benötigte Rechenleistung und die zugrunde liegende Flexibilität der Zielarchitektur. Abbildung 1 gibt qualitativ eine Übersicht über die subjektive Flexibilität verschiedener Architekturen gegenüber ihrer Rechenleistung.

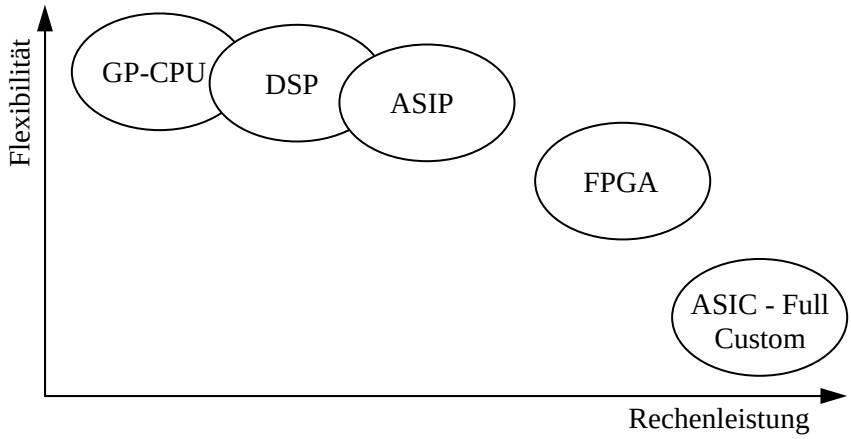


Abbildung 1: Vergleich verschiedener Architekturen nach Flexibilität und Rechenleistung

Größtmögliche Flexibilität bieten Universalprozessoren (*General Purpose CPUs*, GP-CPUs). Sie zeichnen sich durch eine feste Hardwarearchitektur aus und basieren auf der sequentiellen oder eingeschränkt parallelen Abarbeitung von Instruktionen. Die zu implementierende Funktionalität wird durch die Instruktionen in Software abgebildet und durch die Ausführung der Instruktionen auf der universellen und festen Prozessorhardware umgesetzt [1]. Die softwareseitige Programmierung ermöglicht es, Universalprozessoren für verschiedenste Anwendungsgebiete flexibel einzusetzen. Eine Änderung der implementierten Funktionalität erfolgt durch eine Anpassung der Software, die Hardware ist nicht veränderlich. Allerdings ist die Rechenleistung von Universalprozessoren durch die feste Hardware und die sequentielle Programmausführung begrenzt. Weitere durch Software programmierbare Architekturen sind domänenspezifische Prozessoren wie digitale Signalprozessoren (DSPs) und Prozessoren mit anwendungsspezifischem Instruktionssatz (*Application Specific Instruction Set Processors*, ASIPs). Diese Architekturen basieren ebenfalls auf einer festen Hardwarearchitektur und der sequentiellen sowie eingeschränkt parallelen Abarbeitung von Software, sind jedoch auf bestimmte Algorithmenklassen und Anwendungen spezialisiert [2]. Die Spezialisierung der Hardware führt zu einer erhöhten Rechenleistung und verbesserten Ressourcennutzung, aber auch einer verringerten Flexibilität gegenüber Universalprozessoren.

Den flexiblen durch Software programmierbaren Architekturen entgegengesetzt sind kunden- und anwendungsspezifische integrierte Schaltungen (*Application Specific Integrated Circuits*, ASICs), die bis in die Transistorebene auf maximale Rechenleistung und Datendurchsatz optimiert sind. Der Begriff ASIC beschreibt eine große Gruppe von Bausteinen, die in ihrer Hardwarestruktur für eine bestimmte Anwendung optimiert sind [3]. Diese Architekturen implementieren sämtliche Funktionalität direkt in Hardware und können nach der Fertigung des Bausteins nicht mehr geändert werden. Die Flexibilität von ASICs ist so stark eingeschränkt. Eine gewisse Anpassungsfähigkeit kann jedoch durch im Baustein eingebettete Prozessoren erreicht werden (*System on Chip*, SoCs). Die am weitesten spezialisierte und vollkundenspezifische Form von ASICs sind Full Custom Makros.

Einen Kompromiss zwischen der Flexibilität softwareprogrammierbarer Lösungen und der hohen Rechenleistung anwendungsspezifischer Schaltungen bieten *rekonfigurierbare Architekturen*. Ihre Rechenleistung kommt in die Nähe anwendungsspezifischer Bausteine, jedoch kann die implementierte Funktionalität auch nach der Fertigstellung der Komponente an die jeweilige Anwendung angepasst (*rekonfiguriert*) werden. Die Anpassung der Hardwarestruktur an neue Anforderungen stellt den wesentlichen Aspekt rekonfigurierbarer Hardware dar. Der Begriff *Rekonfiguration* kann nach [4] als Prozess der Anpassung und Veränderung der Hardwarestruktur einer Komponente beschrieben werden. Eine weitere Differenzierung ergibt sich aus der Art der Rekonfiguration einer Hardwarekomponente. Erfolgt eine Anpassung und Veränderung der Hardwarestrukturen lediglich außerhalb des Betriebs und der Laufzeit einer Hardwarekomponente, wird dies als *statische Rekonfiguration* oder Konfiguration bezeichnet. Die *dynamische Rekonfiguration* einer Hardwarekomponente wird dagegen durch die Anpassung und Veränderung der Hardwarestrukturen während des laufenden Betriebs charakterisiert. Eine weitergehende Differenzierung der dynamischen Rekonfiguration ergibt sich aus der Art der Hardwareanpassung [5]. Kann im laufenden Betrieb nur die gesamte Hardwarestruktur verändert werden, so bedingt dies in einem System mit nur einer Konfigurationsebene (*Einzelkontextsystem*) den zeitkritischen Austausch sämtlicher Konfigurationsdaten. Durch die Einführung mehrerer Konfigurationsebenen (*Multikontextsystem*) können verschiedene Hardwarekonfigurationen gleichzeitig vorgehalten werden, zwischen denen zur Laufzeit gewechselt werden kann. Problematisch sind der wesentlich höhere Speicherbedarf und die schlechte Ressourcennutzung gegenüber Systemen mit nur einer Konfigurationsebene. Einen deutlich flexibleren Ansatz verfolgt die *partielle Rekonfiguration* von Systemen mit nur einer Konfigurationsebene: während ein Teil der Hardware in Betrieb bleibt, wird ein anderer Teil umkonfiguriert. Bedingung ist hierbei, dass Teile der Konfigurationsebene einzeln angespro-

chen werden können. Die partielle Rekonfiguration erhöht auf diese Weise die Flexibilität und Ressourceneffizienz rekonfigurierbarer Architekturen. Zusätzlich können auf Grundlage der partiellen Rekonfiguration Prinzipien und Methoden der Softwaretechnik auf rekonfigurierbare Hardware übertragen werden. Systeme, die von einer partiellen Rekonfiguration profitieren und diese einsetzen, finden sich bereits im Bereich der Kommunikationstechnik und Videoverarbeitung [4]. Hier unterstützt die partielle Rekonfiguration den dynamischen Zugriff auf Funktionalitäten zur Laufzeit des Systems ohne Erhöhung des Ressourcenbedarfs. Vergleichbare Ansätze für die Steuerung und Regelung mechatronischer Systeme existieren jedoch kaum.

Ein weiteres wesentliches Unterscheidungsmerkmal rekonfigurierbarer Architekturen ist ihre *Granularität* [4]. Der Granularitätsgrad ergibt sich aus den Konfigurationsmöglichkeiten einer Komponente: die Funktionalität feingranularer Architekturen kann bis zur Bitebene modifiziert werden, die Funktionalität grobgranularer Architekturen jedoch nur in Blöcken mit einer bestimmten Wortbreite wie 8,16 oder 32 Bit. Eine kurze Beschreibung grobgranularer Architekturen erfolgt in Abschnitt 1.3. Die Gruppe der feingranularen Architekturen wird hauptsächlich von programmierbaren Logikbausteinen (*Programmable Logic Devices*, PLDs), insbesondere *Complex Programmable Logic Devices* (CPLDs) und *Field Programmable Gate Arrays* (FPGAs), gebildet.

1.2 Field Programmable Gate Arrays

FPGAs wurden im Jahr 1985 von der US-amerikanischen Firma Xilinx auf den Markt gebracht und stellen heute die bedeutendste Gruppe rekonfigurierbarer Hardware dar. Während die frühen FPGAs vor allem für die Realisierung dichter Verbindungen (Glue Logic) zwischen digitalen Bausteinen eingesetzt wurden, sind heutige FPGAs wesentlich leistungsfähiger und können zur Implementierung kompletter digitaler Systeme verwendet werden. Wichtige Einsatzgebiete sind digitale Signalverarbeitung und eingebettete Systeme. Die Bezeichnung *Field Programmable Gate Array* bezieht sich auf die grundlegende Eigenschaft von FPGAs, nach der Herstellung und damit „im Feld“ oder Einsatzgebiet programmierbar zu sein. Weiterführende Informationen zu Grundlagen und Einsatzgebieten von FPGAs sind in [6], [7] und [8] zu finden.

1.2.1 Technologien und Hersteller

Zur Herstellung von FPGAs werden je nach Produzent und gewünschten Eigenschaften unterschiedliche Technologien eingesetzt. Moderne FPGAs lassen sich so in SRAM-, Flash-EEPROM- und Antifuse-basierte Bausteine einteilen. Die nur einmal programmierbaren Antifuse-FPGAs sind für den flexiblen Einsatz als rekonfigurierbare Hardwarekomponente ungeeignet. SRAM und Flash-basierte FPGAs können dagegen mehrfach programmiert werden und sind rekonfigurierbare Hardwarekomponenten. Die große Mehrheit der heute eingesetzten FPGAs nutzt die SRAM-Technologie, da diese ein unbegrenztes und flexibles Programmieren der Hardware erlaubt. Allerdings ist SRAM im Gegensatz zu Flash-EEPROM eine flüchtige Speichertechnologie, so dass SRAM-basierte FPGAs bei jedem Systemstart neu konfiguriert werden müssen. Hierfür wird meist ein externer nicht flüchtiger Speicher eingesetzt. Aufgrund der großen Verbreitung und Marktdominanz SRAM-basierter Architekturen bezieht sich der Begriff FPGA in der vorliegenden Arbeit vorwiegend auf SRAM-basierte FPGAs.

Marktführender Hersteller von FPGAs ist die US-amerikanische Firma Xilinx. Sie stellt SRAM-basierte FPGAs sowohl für den Hochleistungsbereich als auch für kostensensitive Anwendungen her. Der Hochleistungsbereich wird von Xilinx Virtex FPGAs abgedeckt, die eine hohe Logikdichte mit integrierten dedizierten Funktionsblöcken kombinieren. Die aktuellen Vertreter dieser Familie wie die Virtex-4 und Virtex-5 FPGAs verfügen je nach Ausstattung unter anderem über eingebettete Multiplizierer, DSP-Blöcke und Prozessoren [9]. Gegenwärtige Vertreter der kostengünstigeren und kleineren Xilinx Spartan FPGAs sind die Spartan-3 Bausteine, von denen auch eine nicht flüchtige Flash-basierte Variante erhältlich ist [10]. Ein weiterer führender Hersteller von FPGAs ist die Firma Altera, die ähnlich wie Xilinx Hochleistungsbausteine und kostengünstige einfachere FPGAs auf SRAM-Basis anbietet. Aktuelle Hochleistungsbausteine sind die Stratix II und III FPGAs [11]. Die kostengünstigen Bausteine werden als Cyclone vermarktet. Weitere wichtige Hersteller von FPGAs sind unter anderem Lattice und Actel. Lattice stellt sowohl FPGAs auf Basis von SRAM (LatticeSC, LatticeECP2) als auch auf einer kombinierten Basis von Flash und SRAM (LatticeXP2 [12]) her. Actel produziert vor allem Flash-basierte FPGAs. Eine Besonderheit stellen die Actel Fusion Mixed-Signal FPGAs dar, die über einen integrierten Analog/Digital-Wandler sowie weitere analoge Funktionen verfügen [13].

1.2.2 Architektur

Die grundlegende Architektur eines FPGAs entspricht einem regelmäßigen zweidimensionalen Feld von programmierbaren Logikblöcken, das mit einem programmierbaren Verbindungsnetzwerk versehen ist und über programmierbare I/O-Blöcke mit der Außenwelt kommuniziert. Abbildung 2 zeigt den prinzipiellen Aufbau eines FPGAs aus diesen drei Grundelementen. Die dargestellte Architektur entspricht einem symmetrischen Array und wird auch als *Island-Style*-Topologie bezeichnet, da die Logikblöcke wie Inseln im Verbindungsnetzwerk integriert sind. Vor allem FPGAs der marktführenden Firmen Xilinx und Altera nutzen diese Struktur. Altera FPGAs sind dabei auch hierarchisch strukturiert, indem Logikblöcke zu übergeordneten Gruppen zusammengefasst werden. Weitere mögliche FPGA-Strukturen umfassen unter anderem reihenbasierte Lösungen [4]. Die folgenden Absätze geben eine kurze Beschreibung der drei genannten Grundelemente von FPGAs.

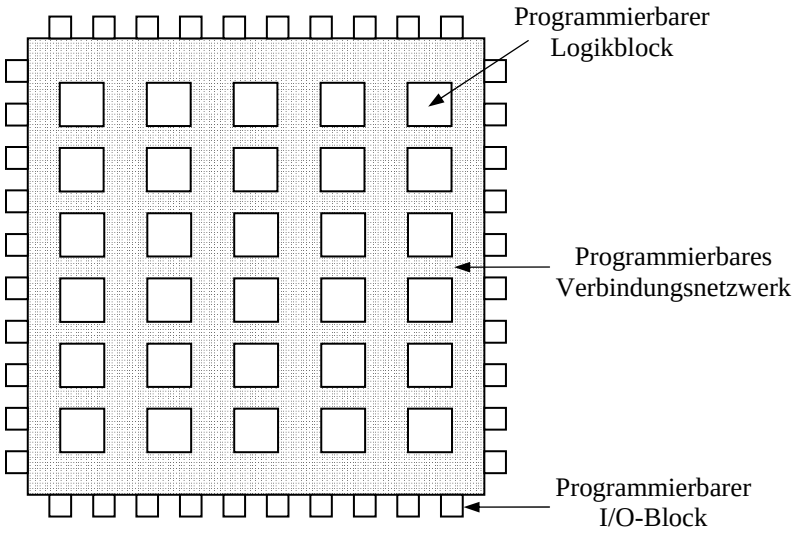


Abbildung 2: Grundlegende Architektur eines FPGAs [8]

Die Basis für die Implementierung von logischen Funktionen auf einem FPGA bilden programmierbare Logikblöcke. Sie bestehen heute herstellerübergreifend meist aus programmierbarer kombinatorischer Logik, Flipflops und schneller Übertragslogik (Carry Logic). Abbildung 3 zeigt den prinzipiellen Aufbau eines solchen programmierbaren Logikblocks. Die in heutigen FPGAs eingesetzten Logikblöcke sind oft sehr flexibel konfigurierbar. So können die Flipflops takt- oder zustandsgesteuert (Latch) eingesetzt und mit synchronen und asynchronen Setz- und Rücksetzsignalen beaufschlagt werden. Weiterhin erfüllt die Übertragslogik häufig zusätzliche Funktionen, wie weitergehende Unterstützung bei Multiplikation und Addition. Die programmierbare kombinatorische Logik innerhalb eines Logikblocks wird

allgemein mit Look-Up-Tables (LUTs) realisiert. Dabei handelt es sich um programmierbare SRAM-Speicherzellen, die beliebige logische Funktionen abbilden. Für einen LUT mit vier Eingängen, wie er in modernen FPGAs aus Flächengründen meist eingesetzt wird, wird beispielsweise ein 16x1 Bit Speicher benötigt. Eine Alternative zu LUTs stellen Kombinationen von Multiplexern und Logikgattern dar. In vielen modernen FPGAs werden die einzelnen programmierbaren Logikblöcke in Gruppen zusammengefasst, um so größere Funktionen effektiver implementieren zu können. Diese Bereiche bestehen in einem Xilinx Virtex-4 FPGA aus acht Blöcken mit vier Eingängen und in einem Altera Stratix-II FPGA aus 24 Blöcken mit drei Eingängen und 16 Blöcken mit vier Eingängen ([9] [11]). Hier ist deutlich der hierarchische Ansatz in Altera FPGAs zu erkennen.

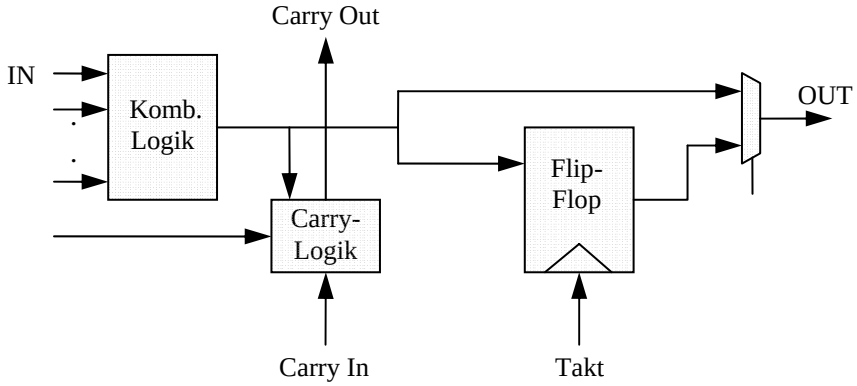


Abbildung 3: Prinzipielle Struktur eines programmierbaren Logikblocks [8]

Die programmierbaren Verbindungsnetzwerke in FPGAs sind für die lokale Verbindung von Logikblöcken innerhalb der übergeordneten Gruppen und die globale Verbindung der Gruppen und Blöcke verantwortlich. Für die komplexeren globalen Verbindungen haben sich insbesondere zwei grundlegende Routing-Architekturen durchgesetzt: die oben bereits genannte Island-Style-Architektur und die Long-Line-Architektur. Die Island-Style-Architektur basiert auf Schaltmatritzen (Switch Boxes), die einzelne horizontale und vertikale Segmente der globalen Verbindungsleitungen flexibel verbinden. Logikblöcke bzw. Gruppen von Blöcken sind über Verbindungsstellen (Connection Boxes) an die globalen Verbindungsleitungen angekoppelt. Die Long-Line-Architektur basiert dagegen auf Leitungen, die die gesamte Länge oder Breite des FPGAs einschließen. Auf diese Weise können Gruppen und Logikblöcke über die Zusammenschaltung von zwei oder mehr Leitungen verbunden werden. Die beiden Routing-Architekturen werden in modernen FPGAs oft kombiniert eingesetzt. Eine detaillierte Darstellung und Klassifizierung der verschiedenen Verbindungsnetzwerke findet sich in [8].

Die Verbindung der inneren Strukturen mit den Anschlusskontakten eines FPGAs wird über programmierbare I/O-Blöcke realisiert. Grundlegende Bestandteile der I/O-Blöcke sind Input-Buffer für die Eingänge und Tri-State-Buffer für die Ausgänge. Zusätzlich können die verschiedenen Signale eines I/O-Blocks, wie das Freigabesignal für den Tri-State sowie Eingangs- und Ausgangssignale, je nach Programmierung über Register geführt werden. Diese elementare Struktur der I/O-Blöcke ist bei den meisten Architekturen und Herstellern sehr ähnlich. Die I/O-Blöcke heutiger FPGAs verfügen darüber hinaus über eine Vielzahl zusätzlicher Funktionen, die den Einsatz von FPGAs in vielen Anwendungen erleichtern. So können verschiedenste I/O-Standards eingestellt und Verzögerungszeiten für Eingänge vorgegeben werden.

1.2.3 Dedizierte Funktionsblöcke

Moderne FPGAs enthalten neben den oben beschriebenen grundlegenden Architekturkomponenten oft weitere spezialisierte Funktionsblöcke, die die Funktionalität für verschiedene Anwendungsgebiete erweitern. Dies führt zu deutlich heterogeneren Strukturen heutiger FPGA-Architekturen. In fast allen heute auf dem Markt befindlichen FPGAs sind eingebettete RAM-Blöcke vorhanden, die eine effiziente Implementierung von Speicherkomponenten ermöglichen ohne auf die Flipflops und LUTs der Grundarchitektur zurückzugreifen. Die Größe und Anzahl der dedizierten Speicherblöcke ist je nach Architektur, Hersteller und Produktfamilie unterschiedlich. Die größeren heute erhältlichen FPGAs enthalten in der Summe der verteilten Blöcke bis zu 1 MB dedizierten Speicher, der über die große Zahl der Ports sehr flexibel ansprechbar ist. Eine weitere ergänzende Komponente moderner FPGAs sind eingebettete Blöcke zur direkten und effizienten Umsetzung arithmetischer Operationen. Häufig werden einzelne dedizierte Multiplizierer oder konfigurierbare DSP-Blöcke eingesetzt. Die DSP-Blöcke können unter anderem Addition, Subtraktion, Multiplikation implementieren und so die Realisierung von Signalverarbeitungsalgorithmen wesentlich effizienter ermöglichen.

Eine besondere Stellung nehmen moderne FPGAs mit eingebetteten Mikroprozessoren ein. Diese erweitern die hohe Rechenleistung und hardwarebezogene Flexibilität eines FPGAs um die Flexibilität softwareprogrammierbarer Systeme. So ist es möglich, ganze eingebettete Systeme auf nur einem FPGA zu implementieren. Beispiele sind die älteren ARM-basierten Altera Excalibur FPGAs und die neueren PowerPC-basierten Xilinx Virtex-II Pro, Virtex-4 FX und Virtex-5 FX FPGAs [14]. An dieser Stelle sollen außerdem die so genannten Softcore-Mikroprozessoren erwähnt werden. Im Gegensatz zu den oben genannten Prozessoren sind sie nicht als dedizierte Funktionsblöcke fest auf den FPGAs vorhanden, sondern werden flexibel

mit den grundlegenden Ressourcen eines FPGAs implementiert. Beispiele für diese Klasse von eingebetteten Mikroprozessoren sind der MicroBlaze von Xilinx [15], der NIOS II von Altera [16] und der LatticeMICO32 [17].

1.2.4 Programmierung

Die Programmierung SRAM-basierter FPGAs beschreibt die Art und Weise in der Konfigurationsdaten (*Bitstream*) auf den Baustein geladen werden und die grundlegenden SRAM-Zellen konfigurieren. Die Konfigurationsdaten werden vorher von den Implementierungswerkzeugen des FPGA-Herstellers aus einer vom Anwender erstellten Spezifikation erzeugt. Für die initiale Konfiguration werden die erforderlichen Daten meist von einer übergeordneten Ebene wie PC, Mikrocontroller oder externem Programmierspeicher (PROM) an eine der Programmierschnittstellen des FPGAs geschickt und konfigurieren den gesamten Baustein. Die Programmierschnittstellen basieren dabei auf serieller oder 8 bzw. 32 Bit Datenübertragung und laufen mit Geschwindigkeiten von 1-100 MHz. Übliche Konfigurationszeiten liegen bei einigen hundert Mikrosekunden bis Sekunden. Die bekannteste serielle Schnittstelle ist das Boundary Scan Interface (Joint Test Action Group, JTAG). Die partielle Rekonfiguration von FPGAs, also die Umprogrammierung von Hardwarebereichen im laufenden Betrieb, setzt insbesondere eine Möglichkeit zur getrennten Konfiguration einzelner Bereiche des FPGAs voraus. Eine der wenigen FPGA-Architekturen, die über eine entsprechende Granularität der Konfiguration verfügen, sind Xilinx FPGAs. Abschnitt 2.3 dieser Arbeit geht näher auf die partielle Rekonfiguration von FPGAs ein.

1.3 Weitere rekonfigurierbare Architekturen

Die Gruppe programmierbarer Logikbausteine umfasst neben FPGAs weiterhin die *Complex Programmable Logic Devices* (CPLDs). CPLDs bestehen aus größeren Blöcken von programmierbaren UND- und ODER-Feldern (*Programmable Logic Arrays*, PALs) und haben einen komplexeren Aufbau als FPGAs. Sie erreichen jedoch nicht deren Leistungsfähigkeit und werden vor allem als Verbindungslogik eingesetzt. Eine Besonderheit stellt der Altera MAX II CPLD dar: er ist als Multikontextsystem realisiert und kann so im laufenden Betrieb zwischen verschiedenen Hardwarekonfigurationen wechseln [18]. Als mit programmierbaren Logikbausteinen verwandte Architekturen sind weiterhin Mikrocontroller mit eingebetteten FPGAs zu erwähnen. Hier wird ein Mikroprozessor mit festen Peripheriekomponenten um ein kleineres meist SRAM-basiertes FPGA erweitert. Als Beispiel sei die Atmel FPSLIC Archi-

tektur genannt, die ebenfalls die dynamische Rekonfiguration des eingebetteten FPGAs unterstützt ([19] [20]).

Eine konsequente Weiterentwicklung stark heterogener FPGAs oder Mikroprozessoren mit eingebetteten FPGAs sind die *Coarse-Grained Reconfigurable Devices* (CGRDs). Viele CGRDs bestehen aus einer Anordnung von grobgranularen programmierbaren Funktionsblöcken und einem programmierbaren Verbindungsnetzwerk. Die Funktionsblöcke sind meist arithmetisch-logische Einheiten (ALUs) für die Ausführung von Additions-, Subtraktions- und Multiplikationsoperationen. Die Wortbreite der Funktionsblöcke ist im Gegensatz zu feingranularen Architekturen fest und beträgt meist 8, 16 oder 32 Bit. Für spezifische Anwendungen bieten CGRDs daher eine bessere Flächen- und Energieeffizienz als feingranulare Strukturen, jedoch auf Kosten der Flexibilität. Als Beispiel sei hier die PACT XPP-III Architektur erwähnt, die auf dem genannten Prinzip von grobgranularen Funktionsblöcken und konfigurierbaren Verbindungsnetzwerken basiert [21]. Ein weiteres Beispiel grobgranularer Architekturen sind die software-konfigurierbaren Prozessoren der Firma Stretch, die eine Weiterentwicklung von Mikroprozessoren mit eingebettetem FPGA darstellen [22]. Besonderes Merkmal dieser Architektur ist eine konfigurierbare Erweiterung des Instruktionssatzes in Hardware. Weitergehende Informationen zu grobgranularen Architekturen sind in [4], [8] und [23] zu finden.

1.4 Ziele der Arbeit

Das Ziel der vorliegenden Arbeit ist es, Methoden und Vorgehensweisen für den Entwurf und die Implementierung flexibler Controller in mechatronischen Systemen auf der Basis moderner rekonfigurierbarer Hardware zu entwickeln. Dabei wird insbesondere die partielle Rekonfiguration als Möglichkeit genutzt, die Funktionalität der entworfenen Controller dynamisch an veränderte Hardwareanforderungen anzupassen. Ein Teilziel der Arbeit ist daher die Entwicklung einer Entwurfsmethodik, die sich in bestehende Entwurfskonzepte mechatronischer Systeme integriert und den Entwurf strukturvariabler Funktionalitäten durch ein implizites Rekonfigurationsmanagement unterstützt. Weiterhin werden essentielle Funktionsgruppen rekonfigurierbarer Controller entwickelt und in ein durchgängiges Spezifikationsframework eingebunden. Ziel ist es, mit den entwickelten Funktionsgruppen eine Infrastruktur zu schaffen, die den Einsatz partiell rekonfigurierbarer Hardware für mechatronische Systeme unterstützt und vereinfacht. Von besonderer Bedeutung ist dabei die Entwicklung eines hardwarenahen und echtzeitfähigen Rekonfigurationsmechanismus. Als Zielhardware für die Control-

ler und deren Funktionsgruppen werden FPGAs eingesetzt. Die Implementierung aller Funktionalitäten erfolgt direkt in Hardware und gewährleistet die harte Echtzeitfähigkeit der Controller. Abschließendes Ziel der Arbeit ist es, die Anwendbarkeit und Leistungsfähigkeit der entwickelten Methoden und Funktionsgruppen mit ausgewählten Beispielimplementierungen aus dem Bereich der Mechatronik aufzuzeigen.

2 Stand der Technik

Dieses Kapitel der Arbeit stellt den Stand der Forschung und Technik im Bereich rekonfigurierbarer Hardware und deren Einsatz in der Mechatronik vor. Dabei wird zuerst auf gebräuchliche Entwurfsmethoden für Steuer- und Regelfunktionalitäten in mechatronischen Systemen eingegangen, da diese grundlegend für die zu entwickelnden rekonfigurierbaren Controller sind. Es folgt eine Übersicht über den Einsatz rekonfigurierbarer Hardware in verschiedenen Anwendungsgebieten der Mechatronik mit entsprechenden Beispielen. Der dritte Abschnitt dieses Kapitels beschreibt Hintergründe und Möglichkeiten der partiellen Rekonfiguration heutiger FPGAs. Das Kapitel schließt mit den Grundlagen bitserieller Signalverarbeitung, die für die Implementierung rekonfigurierbarer Funktionalitäten in der vorliegenden Arbeit eingesetzt wird.

2.1 Entwurfsmethoden

Heutige mechatronische Systeme, z.B. im Fahrzeug, sind durch einen hohen Anteil von Elektronik und Software geprägt. Der Entwurf und die Funktionsentwicklung der darin implementierten Steuer- und Regelfunktionalitäten erfordern Methoden, die sich in übergeordnete Entwicklungsprozesse einordnen und so eine durchgängige System- und Produktentwicklung unterstützen.

Von besonderer Bedeutung für die Entwicklung mechatronischer Systeme und insbesondere den Entwurf von Steuer- und Regelfunktionalitäten ist das ursprünglich aus der Softwareentwicklung stammende V-Modell [24]. Dieses Vorgehensmodell trennt die Systementwicklung in eine absteigende und eine aufsteigende Entwicklungsphase. Die erste Phase beginnt mit der Definition der Benutzeranforderungen und geht über die Systemspezifikation und den eigentlichen Systementwurf bis zur Systemimplementierung. Die zweite Entwicklungsphase führt von der Systemintegration, also dem Zusammenführen entwickelter Komponenten, über Sys-

temtests bis zur Abnahme des Gesamtsystems. Eine gedankliche Zusammensetzung beider Entwicklungsphasen führt zu dem namensgebenden Gesamtmodell des Vorgehens. Die Systementwicklung nach dem V-Modell findet sich in einer angepassten Form auch in der VDI-Richtlinie 2206 „Entwicklungsmethodik für mechatronische Systeme“ [25].

Als exemplarische Anwendung des V-Modells für den Entwurf von Steuer- und Regelfunktionalitäten mechatronischer Systeme kann die Systementwicklung eingebetteter Systeme (Steuergeräte) im Kraftfahrzeug betrachtet werden. Ein beispielhaftes V-Modell für diesen Anwendungsfall wird in [26] vorgestellt und ausführlich beschrieben. Der dort dargestellte Systementwicklungsprozess lässt sich grob in einen übergeordneten Teil der Systementwicklung und einen untergeordneten Teil der Komponentenentwicklung aufteilen. Der Entwurf von Steuer- und Regelfunktionalitäten erfolgt dabei vor allem im Kontext der Softwareentwicklung, die einen Zweig der Komponentenentwicklung bildet. Abbildung 4 zeigt den auf die Softwareentwicklung bezogenen Teil des zugrunde liegenden V-Modells.

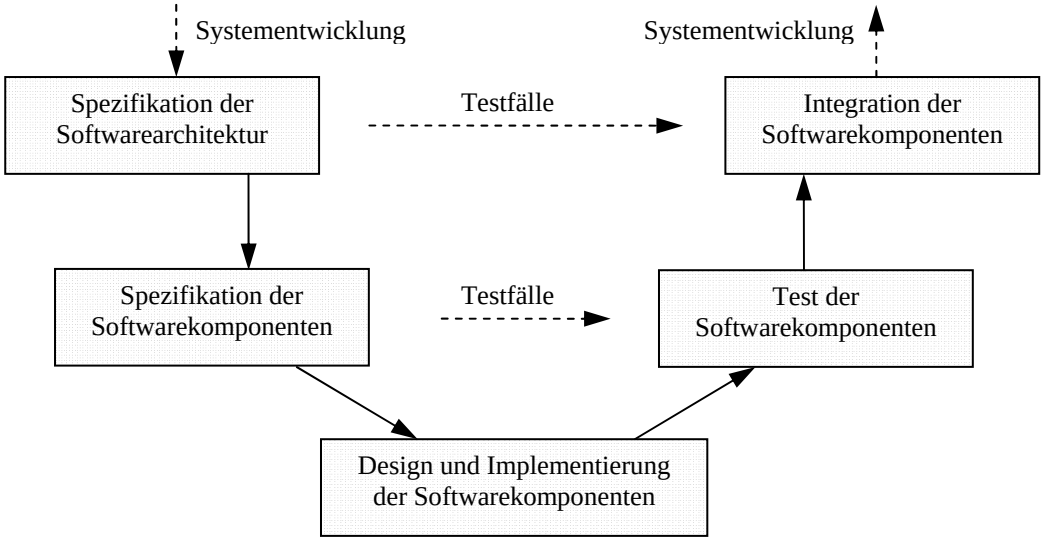


Abbildung 4: Ausschnitt aus einem V-Modell zur Systementwicklung eingebetteter Systeme im Kraftfahrzeug [26]

Besondere Bedeutung im Entwicklungsprozess kommt den Spezifikationsmitteln für Steuer- und Regelfunktionalitäten zu. Dies bezieht sich in der Entwicklung mechatronischer Systeme vor allem auf die Spezifikation der Softwarekomponenten. Übliche Spezifikationsmittel sind hier überwiegend grafische Methoden wie Blockdiagramme für den Datenfluss und flache oder hierarchische Zustandsautomaten für den Kontrollfluss. Weiterhin werden Entscheidungstabellen und Programmiersprachen wie C oder C++ direkt eingesetzt. Für eine ausführliche Beschreibung der genannten Spezifikationsmittel und deren Einordnung in den Entwicklungsprozess sei auf [27] verwiesen. Werkzeuge für die Funktionsentwicklung, die diese Spe-

zifikationsmittel einsetzen, sind unter anderem Matlab/Simulink der Firma The Mathworks, LabVIEW von National Instruments und ASCET der ETAS GmbH.

Die Entwicklung von Funktionen und Komponenten, die zur Implementierung in rekonfigurierbarer Hardware wie FPGAs bestimmt sind, kann analog zu der Entwicklung von Softwarekomponenten in das V-Modell eingegliedert werden. Für den Entwurf dieser Funktionalitäten werden grundlegend ähnliche Spezifikationsmethoden wie in der Softwareentwicklung angewandt. Da die entsprechenden Werkzeuge allerdings aus der Digitalelektronik und Hardwareentwicklung stammen, sind sie meist nur eingeschränkt in mechatronische Entwicklungsprozesse integrierbar. Dies trifft speziell auf die eingesetzten Hardwarebeschreibungssprachen wie VHDL und Verilog zu [28].

Eine systemübergreifende Alternative zu den genannten Spezifikationsmitteln bietet die *Unified Modeling Language* (UML). Die standardisierte Modellierungssprache stammt aus der Softwareentwicklung betrieblicher Anwendungssysteme und verfügt über komplexe grafische Beschreibungsformen für die Modellierung von Prozessen, Software und Systemen. Der Einsatz von UML für den Entwurf mechatronischer Systeme und deren Steuer- und Regelfunktionalitäten ist jedoch aufgrund der primären Ausrichtung auf betriebliche Softwaresysteme sehr komplex und nur eingeschränkt möglich. Entsprechende Ansätze, Methoden und Werkzeuge werden unter anderem in [29] vorgestellt. Weitere Einsatzmöglichkeiten der UML in verwandten Domänen sind der Entwurf eingebetteter Systeme und Echtzeitsysteme und das Hardware/Software-Codesign für rekonfigurierbare Architekturen ([30] [31]). Auch hier sind UML-basierte Entwurfsmethoden sehr komplex und nur eingeschränkt in bestehende Verfahren integrierbar. Aus den genannten Gründen wird die UML in der vorliegenden Arbeit nicht als Spezifikationsmittel verwendet.

2.2 Rekonfigurierbare Hardware in der Mechatronik

Die Standardimplementierungsplattform für digitale Steuer- und Regelfunktionalitäten in mechatronischen Systemen stellen heute Mikrocontroller dar. Rekonfigurierbare Hardware wird dagegen selten zur Umsetzung von Steuer- und Regelfunktionalitäten genutzt und dient in vielen Systemen vorwiegend zur Realisierung peripherer und zeitkritischer I/O-Funktionen. Moderne FPGAs bieten jedoch flexible Hardwareplattformen mit hoher Rechenleistung und harten Echtzeitfähigkeiten, die in vielen mechatronischen Systemen vorteilhaft eingesetzt werden können. Beispiele für mechatronische Systeme mit harten Echtzeitanforderungen sind unter anderem Kfz-Steuergeräte und sicherheitskritische Steuersysteme in der Luftfahrt. Steu-

ergeräte für die Ansteuerung von Verbrennungsmotoren (Motormanagement, Einspritzung, Zündung) erfordern so die garantierte Ausführung spezifischer Funktionen im unteren Mikrosekundenbereich [27]. Im Vergleich zu Mikrocontrollern gewährleisten FPGAs dabei die Funktionsausführung in allen Betriebszuständen eines Steuergerätes unter Beachtung der harten Echtzeitanforderungen. Auch können deutlich geringere Ausführungszeiten einzelner Funktionen realisiert werden.

Grafische Spezifikationsmethoden und damit einhergehende automatisierte Codegenerierung für Hardwarebeschreibungssprachen erleichtern die Integration rekonfigurierbarer Hardware in mechatronische Entwicklungsprozesse. Allerdings ist der Einsatz von FPGAs in mechatronischen Systemen meist kostenintensiver und aufwendiger als die Nutzung der Standardimplementierungsplattform Mikrocontroller. Ihre Vorteile wie Rechenleistung, Echtzeitfähigkeit und Hardwareflexibilität machen FPGAs dennoch zu einer günstigen Implementierungsplattform für die Steuer- und Regelfunktionalitäten vieler mechatronischer Systeme. Im Folgenden wird der Einsatz von FPGAs in verschiedenen Anwendungsgebieten an ausgewählten Beispielen erläutert. Die beschriebenen Systeme setzen keine partielle Rekonfiguration der FPGAs zur Abbildung von Funktionalitäten ein. Entsprechende Ansätze sind sehr selten und werden in Abschnitt 2.3 dieser Arbeit vorgestellt.

Eines der wichtigsten Anwendungsgebiete für FPGAs und zum Teil auch CPLDs in der Mechatronik ist die Regelung und Steuerung elektrischer Antriebe. Hier können mit FPGAs hohe Abtastraten und harte Echtzeitfähigkeit für digitale Regelungen elektrischer Maschinen realisiert werden. Abbildung 5 zeigt den schematischen Aufbau und die Komponenten eines einfachen FPGA-basierten drehzahlgeregelten Antriebssystems am Beispiel eines DC-Motors. Der Motor ist dabei mit einem Encoder zur Positions- und Drehzahlerfassung ausgestattet. Zum Einsatz kommen meist Inkrementalgeber, die über einen oder mehrere Kanäle jeweils ein periodisches Impulssignal übermitteln. Die Auswertung der Impulssignale und deren Umsetzung in Positions- und Drehzahlinformationen erfolgt in einem Decoder, der auf einer Auswertung der Flanken der Impulssignale basiert. Der Decoder kann im FPGA direkt in Hardware implementiert werden. Damit ergibt sich eine große Flexibilität bezüglich möglicher Abtastraten, Auflösungen und Einsatzfälle. Innerhalb des FPGAs erfolgt nun die für Steuerung und Regelung des Systems implementierte Signalverarbeitung, z.B. eine PI-Drehzahlregelung, und gibt die entsprechenden Stellsignale für die Drehzahl des Motors in digitaler Form aus. Diese werden dann über eine Pulweitenmodulation (PWM) auf ein Signal zur Ansteuerung der Leistungselektronik aufgeprägt. Die PWM-Komponente kann ebenfalls

sehr flexibel in Bezug auf Periodendauer (Auflösung) und Anwendungsfall direkt im FPGA in Hardware implementiert werden. Die Leistungselektronik außerhalb des FPGAs ist im Fall eines DC-Motors meist eine H-Brücke (Vierquadrantensteller) mit vier Schaltern, die angesteuert durch das PWM-Signal die mittlere Ankerspannung des DC-Motors und damit dessen Drehmoment und Drehzahl stellt. Für weitere Informationen zu elektrischen Antriebssystemen sei auf [32] verwiesen.

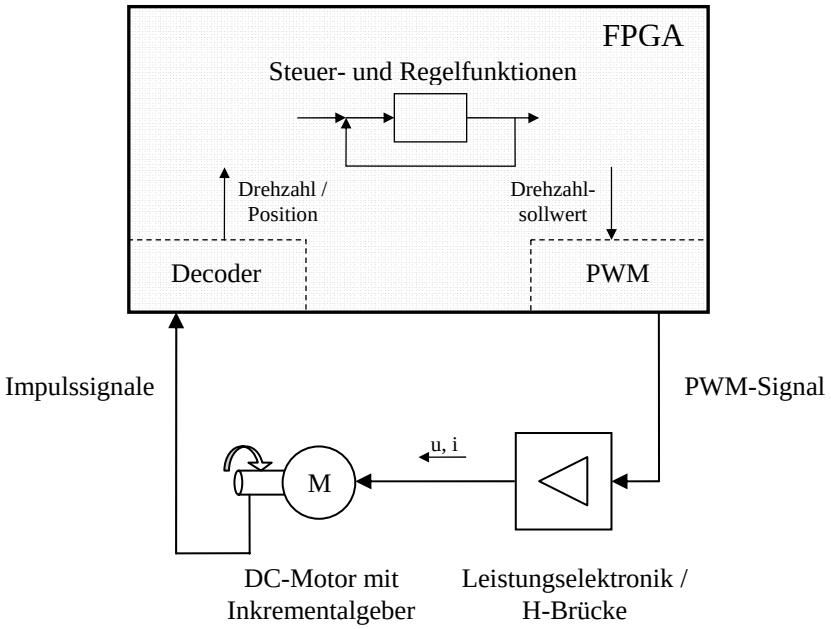


Abbildung 5: Schematischer Aufbau eines FPGA-basierten Antriebssystems mit DC-Motor

Ein Beispiel für die Implementierung von Decoder und PWM-Komponenten in FPGAs für die Ansteuerung von DC-Motoren wird in [33] beschrieben. Dort wird das eingebettete FPGA eines Atmel FPSLIC Mikrocontrollers für die flexible Hardwareumsetzung von mehreren Decoder- und PWM-Einheiten eingesetzt. Eine vollständig in FPGA-Hardware implementierte Antriebssteuerung für Asynchronmotoren auf der Basis von Xilinx FPGAs wird in [34] vorgestellt. Der Vorteil einer direkten FPGA-Implementierung ist hier die einfache Erweiterbarkeit des Systems um Komponenten wie Fehlererkennung und Benutzerschnittstellen. In modernen Antriebssystemen werden neben Asynchronmotoren und DC-Motoren häufig auch elektronisch kommutierte Synchronmaschinen, die so genannten bürstenlosen DC-Motoren (BLDC-Motoren), eingesetzt. Die komplexen Ansteuerungsalgorithmen dieser Antriebe können ebenfalls sehr gut auf FPGAs implementiert werden. So werden unter anderem in [35] und [36] Implementierungen von Controllern für BLDC-Motoren auf FPGAs beschrieben. Beide Quellen nutzen dazu Xilinx FPGAs und den eingebetteten Softcore-Mikroprozessor

MicroBlaze von Xilinx. Eine besondere Bedeutung im Kontext der Antriebssysteme kommt den Actel Fusion Mixed-Signal FPGAs zu. Diese verfügen über integrierte Analog-Digital-Wandler, die eine Messung von Motorstrom und Motortemperatur auch ohne zusätzliche externe Wandler ermöglichen [37]. Weiterhin bieten Actel Fusion FPGAs integrierte Treiberschaltungen für leistungselektronische Stellglieder.

Ein weiteres Anwendungsgebiet von FPGAs in mechatronischen Systemen stellen die Robotik und verwandte Domänen dar. Hier werden FPGAs vor allem für die direkte Implementierung untergeordneter Steuer- und Regelfunktionalitäten sowie die Kommunikation mit übergeordneten Systembestandteilen eingesetzt. Ein bekanntes Beispiel hierfür ist die mechatronische Hand des Deutschen Zentrums für Luft- und Raumfahrt [38]. Die Steuerung und Regelung dieses Systems erfolgt auf drei Ebenen. Auf der obersten Ebene befindet sich eine Kombination aus digitalem Signalprozessor als zentrale Recheneinheit und FPGA, das hier für die Kommunikation zuständig ist. Auf der mittleren Ebene, physisch durch ein Board in der Handfläche repräsentiert, übernimmt ein FPGA die komplexe Kommunikation und den Datentransfer zwischen der untersten und der oberen Ebene. Diese unterste Ebene der Hand sind die einzelnen Finger inklusive Sensorik und BLDC-Motoren als Antriebe. Auf dieser Ebene werden FPGAs in jedem einzelnen Finger für die Kommunikation mit den oberen Ebenen aber auch für die direkte Ansteuerung der Antriebe eingesetzt. Ein weiteres Beispiel für den Einsatz von FPGAs in der Robotik ist in [39] zu finden. Dort wird die Implementierung eines Fuzzy-Reglers für die adaptive Positionierung eines Gelenkes mit harten Nichtlinearitäten auf einem Xilinx FPGA beschrieben.

Ein drittes wichtiges Anwendungsgebiet für FPGAs in der Mechatronik ist das Rapid Control Prototyping, also die Entwicklung von Steuer- und Regelfunktionalitäten innerhalb einer Produktentwicklung. FPGAs bieten hier die Möglichkeit, eine konsistente Entwicklungsplattform für digitale Hardware in allen Entwicklungsphasen einzusetzen. Die implementierten Funktionen können durch eine statische Rekonfiguration der Hardware an verschiedene Entwicklungsschritte angepasst und beliebig erweitert werden. Der Einsatz eingebetteter Prozessorsysteme ergänzt diesen Ansatz um die Flexibilität von Software. So wird in [40] die Implementierung eines FPGA-basierten Rapid-Control-Prototyping-Systems für elektrische Antriebe beschrieben. Hier wird zusätzlich zur direkten Implementierung von Funktionen in Hardware ein Softcore-Mikroprozessor auf dem FPGA genutzt. Besonders interessant ist der Einsatz von FPGAs für die Funktionsentwicklung von Steuergeräten im Kraftfahrzeug (*Electronic Control Units*, ECUs). Ein Beispiel aus diesem Bereich wird in [41] vorgestellt. Das dort

beschriebene Entwicklungssteuergerät für die Motorsteuerung eines Motorrades basiert auf der CompactRIO-Plattform von National Instruments [42]. Die Hardwareplattform besteht aus einem Fließkommaprozessor für übergeordnete Regelfunktionen und flexibler FPGA-Hardware, die für die direkte Implementierung von aktornahen Regelfunktionen und für schnelle I/O-Funktionen eingesetzt wird. Die Verwendung der FPGA-Hardware ermöglicht den Funktionsentwicklern sehr hohe Abtastraten und eine hochdynamische Regelung der Verbrennungsvorgänge im Motor. Für die grafische Programmierung des Systems wird LabVIEW und das LabVIEW FPGA Modul von National Instruments genutzt [43].

2.3 Partielle Rekonfiguration von FPGAs

Die partielle Rekonfiguration von FPGAs ermöglicht den Austausch bzw. die Umprogrammierung von Teilen der Hardwarestruktur im laufenden Betrieb. Die Strukturvariabilität partiell rekonfigurierbarer Hardware bietet dabei völlig neue Ansätze für die Gestaltung und den Einsatz digitaler Hardware. Im Bereich der Mechatronik können so Controller realisiert werden, die ihre Hardwarestruktur und Parameter dynamisch an Betriebspunkte eines Systems anpassen, ohne alle benötigten Algorithmen parallel zu implementieren. Dadurch sinkt der Ressourcenbedarf der Implementierung. Weiterhin wird die Flexibilität der hardwarebasierten Controller stark erhöht, so dass aus der Softwaretechnik bekannte Prinzipien für den Entwurf angewendet werden können.

Das Potential der partiellen Rekonfiguration SRAM-basierter FPGAs wird bereits in [6] aus dem Jahr 1995 beschrieben. Die dort behandelten Architekturen waren sehr feingranular, so dass die Hardwarestruktur anhand von einzelnen Konfigurationsbits geändert werden konnte. Unter den heutigen auf dem Markt befindlichen Architekturen unterstützen insbesondere FPGAs von Xilinx die partielle Rekonfiguration. Xilinx bietet im Gegensatz zu anderen Herstellern Werkzeuge sowie Hardwareunterstützung für die Umsetzung partiell rekonfigurierbarer Funktionalitäten [44]. Aus diesem Grund befasst sich die überwiegende Mehrzahl der aktuellen Forschungs- und Entwicklungsprojekte zur partiellen Rekonfiguration von FPGAs mit Xilinx Bausteinen. Xilinx FPGAs sind auch die Zielplattform der in dieser Arbeit entwickelten rekonfigurierbaren Controller. Die folgenden Ausführungen zu Grundlagen, Werkzeugen und Umsetzung der partiellen Rekonfiguration beziehen sich daher direkt auf Xilinx FPGAs.

Auf weitere nach Herstellerangaben partiell rekonfigurierbare FPGA-Architekturen wie den Atmel FPSLIC/AT40K sowie die LatticeSC und Orca Bausteinen von Lattice sei an dieser Stelle nur der Vollständigkeit halber verwiesen ([20] [45] [46]).

2.3.1 Hardwaregrundlagen

Die partielle Rekonfiguration von FPGAs setzt auf der Hardwareebene die Möglichkeit voraus, einzelne Bereiche der Architektur getrennt von anderen Bereichen umzuprogrammieren. Ausgehend von der Granularität ihrer Hardwarestruktur könnten SRAM-basierte FPGAs auf Bitebene konfiguriert werden. Allerdings bedingen die spezifischen Architekturen der Hersteller, dass der Konfigurationszugriff meist nur auf das ganze FPGA oder größere Teile davon möglich ist. Letzteres gilt für Xilinx FPGAs, die auf diese Weise partiell rekonfigurierbar sind. Die kleinsten einzeln programmierbaren Hardwareeinheiten auf Xilinx FPGAs werden als Frames bezeichnet. Die Größe und Anzahl der Frames ist dabei von der Bausteinfamilie und teilweise auch der Bausteingröße abhängig. So umfassen Frames der Virtex-II und Spartan-3 FPGAs immer Spalten über die ganze Höhe des FPGAs ([47] [48]). Die daraus abgeleiteten kleinsten praktisch rekonfigurierbaren Bereiche setzen sich aus mehreren Frames zusammen und werden durch einzelne Spalten oder Blöcke der Architektur repräsentiert. Die minimale Breite der Bereiche beträgt einen *Configurable Logic Block* (CLB), der wiederum aus vier *Slices*, den grundlegenden Logikblöcken von Xilinx FPGAs mit je zwei LUT und zwei Flipflops, besteht. Die kleinste einzeln rekonfigurierbare Einheit ist für Virtex-II und Spartan-3 FPGAs daher von der Größe des zugrunde liegenden Bausteins abhängig. Bei den moderneren Xilinx Virtex-4 FPGAs gilt die Beschränkung der kleinsten rekonfigurierbaren Bereiche auf Spalten der ganzen Länge des Bausteins nicht mehr. Hier können rechteckige Blöcke mit einer minimalen Höhe von 16 CLBs und einer minimalen Breite von einem CLB rekonfiguriert werden [49].

2.3.2 Werkzeuge und Umsetzung

Für die Implementierung partiell rekonfigurierbarer Designs werden von Xilinx verschiedene Werkzeugketten und Verfahren bereitgestellt. Nur für kleine Änderungen eines Systems ist der *Difference Based Partial Reconfiguration Design Flow* geeignet [50]. Dabei werden die Datenstände zweier existierender Konfigurationsdatensätze (Bitstreams) verglichen und die Differenz als partieller Bitstream für die Rekonfiguration verwendet. Für den Austausch ganzer Hardwarebereiche kann der *Early Access Partial Reconfiguration Design Flow* (EAPR Design Flow) eingesetzt werden [51]. Dieser wurde aus dem älteren *Module Based Partial Reconfiguration Design Flow* entwickelt und basiert auf einer Aufteilung des Designs in einen statischen sowie einen oder mehrere partiell rekonfigurierbare Bereiche [52]. Die Kommunikation zwischen rekonfigurierbaren und statischen Bereichen eines Designs wird dabei über so genannte Bus-Makros realisiert. Sie sind mit Hilfe von LUTs und Routingressourcen

implementierte persistente Verbindungen, die von einer Rekonfiguration nicht betroffen sind. Für den EAPR Design Flow kommen modifizierte Implementierungswerkzeuge von Xilinx zum Einsatz, die initiale Konfigurationsdatensätze für den ganzen FPGA und partielle Bitstreams für die rekonfigurierbaren Bereiche erzeugen. Die Vorgehensweise ist skriptbasiert, eine ergänzende grafische Unterstützung wird in [53] vorgestellt.

Auf der Hardwareebene erfolgt die Umsetzung der partiellen Rekonfiguration für den Nutzer durch die Übermittlung der erzeugten partiellen Bitstreams an die Konfigurationsschnittstellen des FPGAs. Der Großteil der Konfigurationsschnittstellen muss von externen Quellen gesteuert werden. Eine Ausnahme ist der in Xilinx FPGAs eingebettete *Internal Configuration Access Port* (ICAP), der die Selbstrekonfiguration eines Systems ermöglicht [54]. Weitere Informationen zu Vorgehensweisen und Werkzeugen finden sich unter anderem in [4]. Kapitel 5 dieser Arbeit geht näher auf Hardwaregrundlagen, Implementierungsverfahren, Werkzeuge und Konfigurationsschnittstellen ein.

2.3.3 Anwendungen

Ein wichtiges Anwendungsgebiet für die partielle Rekonfiguration von FPGAs ist die Bild- und Videoverarbeitung. So wird in [55] mit der Sonic-on-a-Chip-Architektur eine modulare Plattform für die Videoverarbeitung beschrieben, die unter Ausnutzung der partiellen Rekonfiguration auf Xilinx FPGAs implementiert wurde. In [56] wird dagegen ein video-basiertes Fahrerassistenzsystems für Kraftfahrzeuge vorgestellt. Während die obere Ebene des Systems durch einen im FPGA eingebetteten Prozessor realisiert wird, werden auf untere Ebene Hardwarebeschleuniger für die Bildverarbeitung mittels partieller Rekonfiguration ausgetauscht. Ein ebenfalls wichtiger Anwendungsbereich ist *Software-defined Radio* (SDR) [53]. Die partielle Rekonfiguration von FPGAs ermöglicht in SDR-Systemen eine flexible Anpassung der Hardware an verschiedene drahtlose Datenübertragungsverfahren [57]. Für diesen Anwendungsbereich wird von Xilinx und iSR Technologies bereits eine Entwicklungsplattform auf der Basis von Xilinx Virtex-4 FPGAs angeboten [58]. Weitere Anwendungsgebiete, wie adaptive Kryptographiesysteme und Hochleistungsrechner, werden unter anderem in [4] dargestellt.

Im Bereich der Mechatronik gibt es bisher nur wenige Ansätze zur Ausnutzung der partiellen Rekonfiguration von FPGAs. Bereits im Jahr 2002 wurde in [59] ein Fail-Safe-System für Steuergeräte in Kraftfahrzeugen vorgestellt, das auf der partiellen Rekonfiguration von älteren Xilinx XC6200 FPGAs basiert. Diese Architekturen verfügten nur über beschränkte Logikres-

sourcen und waren sehr feingranular aufgebaut. In [60] und [4] wird weiterhin ein theoretischer Ansatz für die Implementierung adaptiver Controller auf partiell rekonfigurierbarer Hardware beschrieben. Besonderes Merkmal ist hier der Einsatz zweier rekonfigurierbarer Controllermodule, von denen jeweils nur eines aktiv ist. Eine Hauptsteuereinheit realisiert das Umschalten und die Rekonfiguration der Module.

2.3.4 Weiterführende Ansätze

Der Einsatz der partiellen Rekonfiguration von FPGAs wird durch Randbedingungen und Voraussetzungen heutiger Implementierungssoftware und FPGA-Hardware teilweise eingeschränkt. So gibt es bei aktuellen Implementierungsverfahren immer einen gewissen Überhang an Konfigurationsdaten und -aufwand, der sich aus einer unnötigen Umprogrammierung von Teilen der Hardware ergibt. Ansätze zur Minimierung des Rekonfigurationsüberhangs und damit zu effizienteren Rekonfigurationsmethoden werden unter anderem in [61], [62] und [63] vorgestellt. Die Verteilung der I/O-Pins und deren Zuordnung zu bestimmten Schnittstellen auf aktuellen FPGAs und FPGA-Entwicklungsboards sind ebenfalls problematisch. Hier setzt die in [64] und [4] beschriebene ESM-Plattform an, auf der die Verbindung von rekonfigurierbaren Bereichen zu I/O-Pins durch einen speziellen Interfacecontroller realisiert wird. Weiterhin wird eine freie Umplatzierung (Reallokation) von rekonfigurierbaren Funktionen auf dem FPGA durch heutige Hardware und Implementierungsverfahren noch nicht unterstützt und ist daher Gegenstand aktueller Forschung und Entwicklung. Für weiterführende Informationen zu diesem und anderen aktuellen Forschungsthemen wie temporaler Platzierung und Schedulingalgorithmen sei auf [4] verwiesen.

2.4 Bitserielle Signalverarbeitung

Für die Implementierung von Steuer- und Regelfunktionen in rekonfigurierbarer FPGA-Hardware wird in dieser Arbeit auch bitserielle Signalverarbeitung nach [65] eingesetzt. Gegenüber herkömmlichen bitparallelen Ansätzen verfügt die bitserielle Signalverarbeitung über eine bessere Skalierbarkeit und einen geringeren Logikaufwand. Grundlegender Gedanke der bitseriellen Informationsverarbeitung ist die Übertragung eines Operanden im Zeitmultiplex auf nur einer Leitung. Die so sehr niedrige Komplexitätsebene ermöglicht eine intensive Nutzung von Pipelineeffekten. Während bei bitparallelen Ansätzen eine Erhöhung der Verarbeitungswortbreite immer eine Steigerung des Gatterbedarfs bedeutet, führt dies bei bitseriellen Methoden lediglich zu einer Vergrößerung der Bearbeitungs- und Übertragungszeit. Dabei

hängt die mögliche Abtastfrequenz f_S direkt von der Wortbreite n ab und begrenzt sie nach oben entsprechend der Taktfrequenz f_C. Es gilt:

$$f_{S\max} = \frac{f_C}{n} \qquad \text{Gleichung 2-1}$$

Allerdings erlauben moderne FPGAs Taktfrequenzen von mehreren 100 MHz, so dass sich für die Implementierung mechatronischer Steuer- und Regelfunktionen kein Nachteil durch die Reduzierung der Abtastfrequenz ergibt.

Da arithmetische Algorithmen immer von einer bekannten Gewichtung der Operandenbits ausgehen, wird für die bitserielle Signalverarbeitung nach [65] eine Kennzeichnung aller Bits durch ein getrennt zu übertragendes Synchronsignal implementiert. Das Synchronsignal wird durch einen Bus mit einer der Verarbeitungswortbreite entsprechenden Breite realisiert. Dieses Prinzip wird in Abbildung 6 verdeutlicht. Dabei muss zusätzlich jedem Operator innerhalb der bitseriellen Signalverarbeitungskette die durch vorhergehende Operatoren verursachte Verzögerung des Datenstroms bekannt gegeben werden. Die Verzögerung wird manuell oder automatisiert mit einer Strukturanalyse vor der Implementierung ermittelt [66].

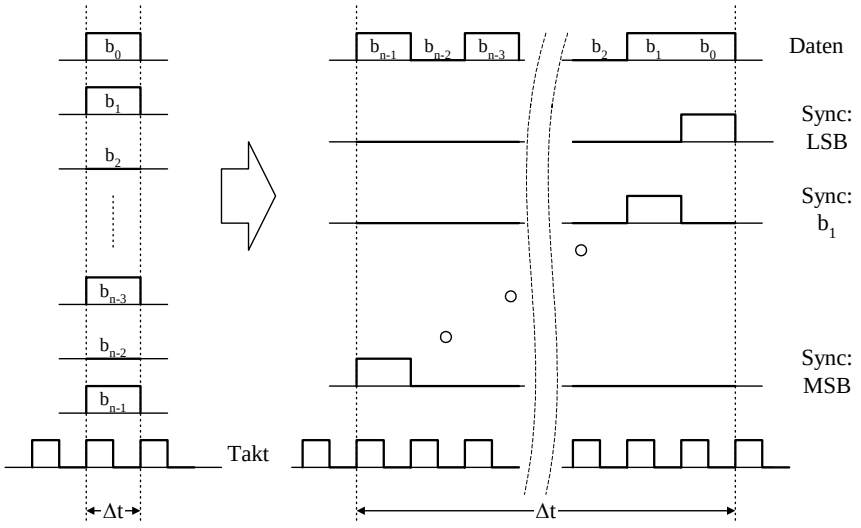


Abbildung 6: Synchronisationssignale bitserieller Signalverarbeitung nach [65]

Für weitere Informationen zu Grundlagen bitserieller Signalverarbeitung sei auf [65] verwiesen. Abschnitt 5.1.2 dieser Arbeit geht auf die Einsatzmöglichkeiten und Vorteile bitserieller Signalverarbeitung in partiell rekonfigurierbaren Systemen ein.

3 Entwurfsmethodik

Dieses Kapitel der Arbeit stellt eine universelle Entwurfsmethodik für rekonfigurierbare Controller in mechatronischen Systemen vor. Ausgehend von allgemeinen Entwurfsmethoden für Steuer- und Regelfunktionalitäten mechatronischer Systeme wird eine Methodik entwickelt, die sich in bestehende Vorgehensweisen integriert und diese um Abbildungsmöglichkeiten für partiell rekonfigurierbare Hardwarefunktionen ergänzt. Dabei werden allgemeingültige Spezifikationsmittel und Werkzeuge eingesetzt. Besonderes Merkmal der hier vorgestellten Entwurfsmethodik ist die implizite Definition und Integration eines Rekonfigurationsmanagements. Dieses ergibt sich aus einer Partitionierung von Funktionalitäten in statische und rekonfigurierbare Hardwareanteile und ermöglicht eine hardwareunabhängige Berücksichtigung von Rekonfigurationsvorgängen schon zu frühen Entwicklungszeitpunkten. Die eingesetzten Spezifikationsmittel unterstützen hierbei eine verteilte Implementierung des Rekonfigurationsmanagements sowie eine echtzeitfähige Lade- und Aktivierungstrategie für rekonfigurierbare Hardwarefunktionen.

Der erste Abschnitt des Kapitels beschreibt die grundlegenden Methoden und Werkzeuge der Entwurfsmethodik und deren Einbindung in bestehende Vorgehensweisen. Daraufhin wird im zweiten Abschnitt die Partitionierung von rekonfigurierbaren Controllern und Hardwarefunktionen auf Funktions- und Hardwareebene erläutert. Der dritte Teil des Kapitels geht näher auf das verteilte Rekonfigurationsmanagement, dessen implizite Spezifikation und abschließend auf Lade- und Aktivierungsstrategien für Hardwarefunktionen ein.

3.1 Entwurfsgrundlagen

Der Entwurf von Steuer- und Regelfunktionalitäten für mechatronische Systeme erfolgt heute zumeist im Kontext übergeordneter Vorgehensweisen für die System- und Produktentwicklung. Insbesondere das aus der Softwareentwicklung stammende V-Modell wird für die Be-

schreibung und Modellierung der Entwurfsprozesse eingesetzt. Mit einem der Systementwicklung untergeordnetem V-Modell können daher die Entwurfsschritte für rekonfigurierbare Controller in die Entwicklungsprozesse mechatronischer Systeme eingebunden werden. Ein entsprechender Ansatz wird in diesem Abschnitt beschrieben. Ein weiterer Schwerpunkt der hier vorgestellten Entwurfsgrundlagen sind die innerhalb der Entwicklungsprozesse angewandten Spezifikationsmittel und Spezifikationswerkzeuge für rekonfigurierbare Controller und deren Funktionalitäten. Dabei werden ausgehend von bestehenden Softwarewerkzeugen grafische Spezifikationsmittel für Steuer- und Regelfunktionen eingesetzt, die sowohl in der Software- als auch der digitalen Hardwareentwicklung Anwendung finden. Dieser Ansatz unterstützt die Integration partiell rekonfigurierbarer Hardware in Entwicklungsprozesse mechatronischer Systeme.

3.1.1 Vorgehensmodell

Die Vorgehensweise für den Entwurf rekonfigurierbarer Controller in der Mechatronik kann durch einen Unterzweig eines V-Modells der Systementwicklung beschrieben werden. Ausgehend von dem in Abschnitt 2.1 vorgestellten Teil-V-Modell für die Softwareentwicklung von Kraftfahrzeugsteuergeräten nach [26] wird hier beispielhaft ein Teil-V-Modell für rekonfigurierbare Controller mit Fokus auf die partiell rekonfigurierbaren Funktionalitäten vorgestellt.

Wie in Abbildung 7 dargestellt schließt sich der Unterzweig für rekonfigurierbare Controller an die übergeordnete Ebene der Systementwicklung mit dem Teilschritt der Spezifikation der logischen Controllerstruktur an. Dabei wird ein abstraktes Funktionsmodell des Controllers entworfen, das die Struktur und die Vernetzung der einzelnen Teilfunktionen beschreibt. Dieser Schritt ist unabhängig von der späteren Zielplattform und Hardware. Im nächsten Schritt des Teil-V-Modells erfolgt die Partitionierung der einzelnen Controllerfunktionen in statische und partiell rekonfigurierbare Bestandteile. Die Partitionierung auf Funktionsebene bedingt bereits eine Berücksichtigung der partiellen Rekonfigurierbarkeit der Zielplattform, ist jedoch ansonsten unabhängig von der realen Ausprägung der Hardware. Im Anschluss an die Partitionierung werden getrennte Entwurfsschritte für statische und rekonfigurierbare Funktionsgruppen, die Kommunikation zwischen den Controllerbestandteilen und die Abbildung der technischen Controllerstruktur auf eine bestimmte Hardware durchlaufen. In Abbildung 7 werden aus Übersichtsgründen nur die Entwurfsschritte für rekonfigurierbare Funktionalitäten gezeigt. Diese umfassen Spezifikation, Design, Implementierung und Test der rekonfigurierbaren Funktionalitäten. Daraufhin erfolgt die Zusammenführung der einzelnen Funktions-

gruppen in der Implementierung des Gesamtsystems rekonfigurierbarer Controller und abschließend dessen Test. Die Teilschritte der Partitionierung des Controllers sowie die Spezifikation der rekonfigurierbaren Funktionalitäten ermöglichen dabei die implizite Integration eines verteilten Rekonfigurationsmanagements, das in Abschnitt 3.3 vorgestellt wird.

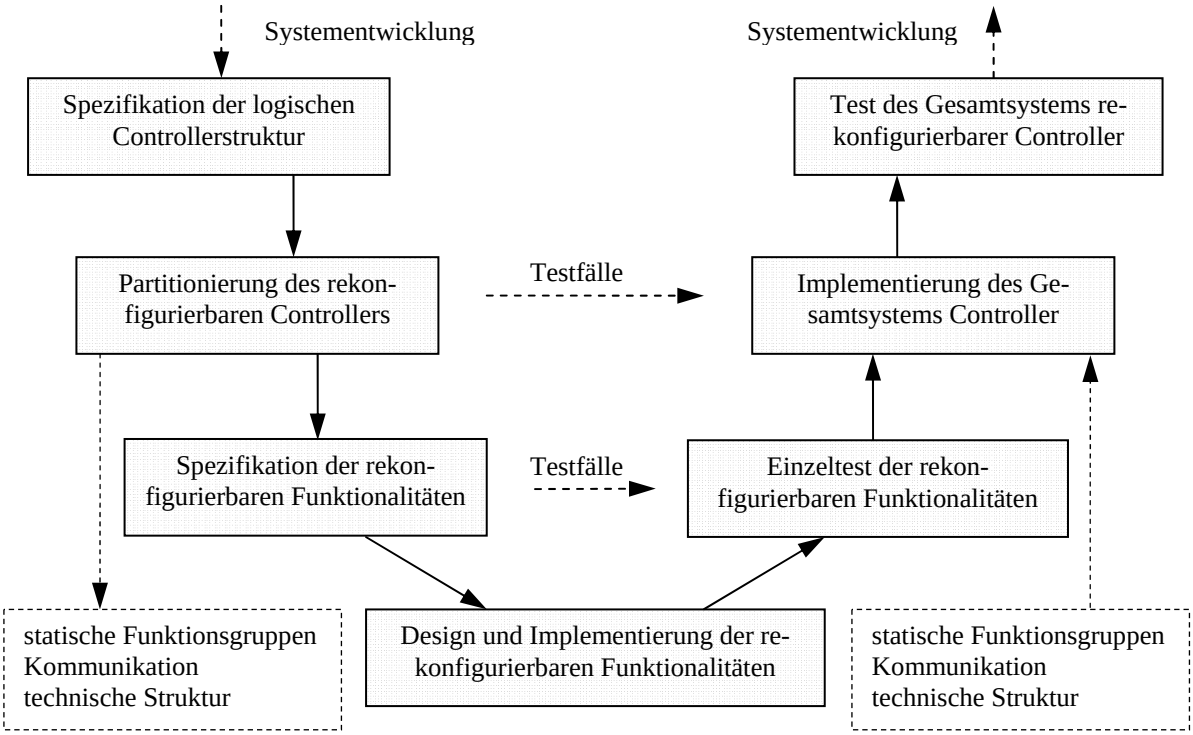


Abbildung 7: Untergeordnetes V-Modell für den Entwurf rekonfigurierbarer Controller

3.1.2 Spezifikationswerkzeuge

Die Entwicklung von Steuer- und Regelfunktionalitäten für mechatronische Systeme erfolgt zumeist mit Softwarewerkzeugen, die eine getrennte grafische Spezifikation von Kontroll- und Datenfluss unterstützen. Zum Einsatz kommen unter anderem Werkzeuge wie Matlab/Simulink der Firma The Mathworks, LabVIEW von National Instruments und ASCET der ETAS GmbH. Der Kontrollfluss wird dabei oft durch Zustandsautomaten abgebildet, deren Steuersignale einen durch Blockdiagramme repräsentierten Datenfluss beeinflussen. Die Spezifikationsmittel der verschiedenen Werkzeuge ermöglichen so eine implementierungsunabhängige Beschreibung von Steuer- und Regelfunktionen und erlauben zum Teil eine Überführung der erstellten Spezifikationsmodelle in Produktionssoftware. Ausgehend davon werden in dieser Arbeit ebenfalls allgemeingültige und hardwareunabhängige grafische Spezifikationsmittel für die Abbildung von Struktur und Verhalten rekonfigurierbarer Funktionalitäten genutzt.

Der Datenfluss rekonfigurierbarer Steuer- und Regelfunktionen in mechatronischen Systemen wird in der vorliegenden Arbeit vorwiegend mit Blockdiagrammen spezifiziert. Blockdiagramme bestehen aus Signalverbindungen, die den eigentlichen Datenfluss repräsentieren, und Blöcken, die Funktionen abbilden. Die Funktionsblöcke eines Blockdiagramms können neben grundlegenden Signalverarbeitungselementen auch weitere Blockdiagramme beinhalten. Weiterhin können Funktionsblöcke andere Spezifikationsformen wie Wahrheitstabellen, Zustandsautomaten und Programmiersprachen integrieren. Dabei verfügen alle Funktionsblöcke über definierte Schnittstellen, die sich in Eingänge und Ausgänge unterscheiden. Auf diese Weise können rekonfigurierbare Funktionalitäten hierarchisch gegliedert und so übersichtlich strukturiert werden. Die oberste Ebene (Top-Level) definiert dann die Schnittstellen eines rekonfigurierbaren Controllers oder einer rekonfigurierbaren Funktion innerhalb eines Controllers, während die niedrigste Ebene elementare Algorithmen abbildet.

Der Kontrollfluss rekonfigurierbarer Steuer- und Regelfunktionen in mechatronischen Systemen wird für die vorliegende Arbeit dagegen mit Zustandsautomaten spezifiziert. Zustandsautomaten bestehen aus Zuständen und Transitionen, den Übergängen zwischen Zuständen. Die Transition zwischen zwei Zuständen erfolgt abhängig von bestimmten Bedingungen, die dem Ausgangszustand zugeordnet sind. Die einzelnen Zustände enthalten spezifische Funktionen, die durch ihre Zustandsaktionen festgelegt sind. Zustandsaktionen können dabei unter anderem mit Blockdiagrammen spezifiziert werden. Auch den Transitionen kann eine spezifische Funktion, die Übergangsaktion, zugeordnet werden. Weiterhin können einzelne Zustände untergeordnete Zustandsautomaten beinhalten, so dass hierarchische Automaten entstehen.

Die Spezifikationsmodelle für Kontroll- und Datenfluss werden in dieser Arbeit mit Softwarewerkzeugen erstellt, die eine Überführung in Hardwarebeschreibungssprachen wie VHDL oder Verilog gestatten. Die abstrakte Beschreibung mit grafischen Mitteln und nachfolgender Codegenerierung erleichtert den Zugang zu programmierbarer Hardware. Primäres Spezifikations- und Codegenerierungswerkzeug ist der HDL Designer von Mentor Graphics [67], der Blockdiagramme, Zustandsautomaten sowie weitere Spezifikationsformen unterstützt ([68] [69]). Der HDL Designer wird darüber hinaus als Plattform für ein übergreifendes Spezifikationsframework genutzt, das in Abschnitt 4.3 näher beschrieben wird. Weiterhin wird Matlab/Simulink in Verbindung mit dem Simulink HDL Coder für die Spezifikation, Simulation und Codegenerierung einzelner Controllerbestandteile eingesetzt [70]. Die aktuelle Version des Simulink HDL Coders unterstützt die Codegenerierung jedoch nur für eine Untergruppe elementarer Funktionsblöcke. Hier nicht verwendete Alternativen zu den ge-

nannten Spezifikations- und Codegenerierungswerkzeugen sind der ebenfalls in Matlab/Simulink integrierte System Generator for DSP von Xilinx [71], das LabVIEW FPGA Modul von National Instruments [43] und spezifische grafische Editoren einzelner FPGA-Hersteller.

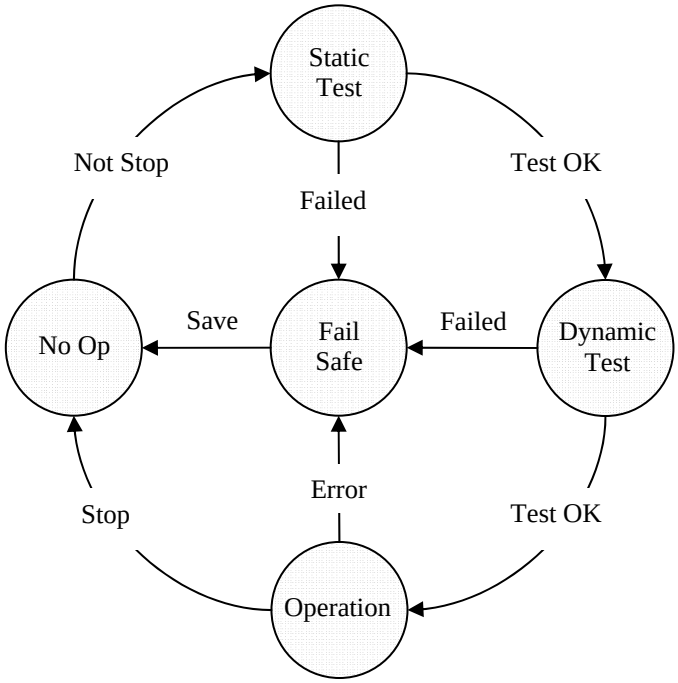


Abbildung 8: Zustandsautomat für Hochlauf und Betrieb eines mechatronischen Systems

3.1.3 Spezifikation der logischen Controllerstruktur mit Zustandsautomaten

Der Einsatz von Zustandsautomaten für die Spezifikation mechatronischer Steuer- und Regelfunktionen basiert auf einer engen Verknüpfung zwischen Kontroll- und Datenfluss. Die logische Struktur eines mechatronischen Controllers mit Betriebszuständen, Ablaufsteuerungen und Funktionswechseln kann durch die Zuordnung von Blockdiagrammen und anderen Funktionsbeschreibungen zu den Zuständen eines Automaten sehr effizient abgebildet werden. Abbildung 8 zeigt einen beispielhaften Zustandsautomaten, der den Hochlauf und den Betrieb eines mechatronischen Systems kontrolliert und so dessen logische Controllerstruktur definiert. Ausgehend von statischen und dynamischen Tests, die auch Aktorik und Sensorik betreffen können, wird das System in den Hauptbetriebszustand gebracht. Im Fehlerfall werden die Testvorgänge und der Hauptbetriebszustand verlassen, um das System in einen sicheren Zustand (Fail Safe) zu bringen. Der Neustart des Systems bedingt den Wechsel in einen passiven Zustand (No Op). Die Spezifikation der einzelnen Betriebszustände erfolgt mit Datenflussbeschreibungen wie Blockdiagrammen, während die Spezifikation des übergeordneten

Systemverhaltens auf einem Zustandsautomaten basiert. Eine Erweiterung dieser Methodik auf rekonfigurierbare Hardware ermöglicht den Einsatz von Zustandsautomaten zur Spezifikation von Lade- und Aktivierungsvorgängen für rekonfigurierbare Steuer- und Regelfunktionalitäten. Ein entsprechender Ansatz zur Verbindung von Spezifikation, Rekonfigurationsmanagement und Strukturvariabilität wird in den folgenden Abschnitten vorgestellt.

3.2 Partitionierung

Die Partitionierung rekonfigurierbarer Controller umfasst die Einteilung der Controllerfunktionen in statische und zur Laufzeit austauschbare Bestandteile sowie deren Abbildung auf generische rekonfigurierbare Hardware. Ausgangspunkt ist ein abstraktes Funktionsmodell, das die logische Struktur und Vernetzung der Teilfunktionen abbildet. Die Partitionierungsmethodik setzt auf der Funktionsebene an und bricht die Controllerfunktionen auf rekonfigurierbare Module und Hardwaretasks herunter. Ein Bezug auf die reale Zielplattform ist für die Partitionierung auf Funktionsebene nicht notwendig, nur die partielle Rekonfigurierbarkeit wird vorausgesetzt.

3.2.1 Partitionierung auf Funktionsebene

Die Partitionierung auf Funktionsebene beschreibt die Aufteilung der Controllerfunktionen in statische und rekonfigurierbare Teilfunktionen. Wurde die logische Controllerstruktur und -funktionalität wie in dem Beispiel aus Abbildung 8 mit Zustandsautomaten spezifiziert, kann die Partitionierung davon ausgehend erfolgen. Jeder der abgebildeten Betriebszustände beschreibt eine spezifische Konfiguration des Controllers, die sich aus verschiedenen Teilfunktionen zusammensetzt. Der Übergang zum folgenden Betriebszustand bewirkt eine Änderung der Konfiguration und führt zum Einsatz neuer Teilfunktionen. Die einzelnen Betriebszustände nutzen jedoch eine Untermenge der Teilfunktionen des Controllers gemeinsam, so dass diese unabhängig vom aktuellen Betriebszustand immer aktiv sind. Wird die Aufteilung der Teilfunktionen in immer aktive und betriebszustandsabhängig aktive Funktionen auf rekonfigurierbare Hardware übertragen, liegt implizit eine Partitionierung in statische und partiell rekonfigurierbare Controllerbestandteile vor.

Statische und immer aktive Teilfunktionen werden unter anderem durch Kommunikationsschnittstellen, Prozessschnittstellen, Speicher- und Rekonfigurationsverwaltung und übergeordnete Controllerfunktionen repräsentiert. Partiell rekonfigurierbare und somit zur Laufzeit des Controllers austauschbare Teilfunktionen sind dagegen betriebszustandsabhängige Steuer-

und Regelalgorithmen, Testabläufe oder Fail-Safe-Funktionen. Das Konzept partiell rekonfigurierbarer Teilfunktionen kann um rekonfigurierbare Schnittstellen erweitert werden.

Das Ergebnis der Partitionierung auf Funktionsebene sind die Teilmengen statischer und partiell rekonfigurierbarer Controllerfunktionen. Die Spezifikation der statischen Funktionsgruppen erfolgt im Anschluss mit den genannten Werkzeugen und wird in Abschnitt 4.2 näher beschrieben. Die Teilmenge der partiell rekonfigurierbaren Controllerfunktionen wird nach der Partitionierung in abstrakte Zustandsautomaten überführt, die Betriebszustände des Controllers unabhängig von statischen Funktionen abbilden. Abschnitt 3.3 beschreibt diesen Automaten und seinen Einsatz für ein verteiltes Rekonfigurationsmanagement, nachdem im Folgenden auf ein Abbildungskonzept für partiell rekonfigurierbare Funktionen auf rekonfigurierbarer Hardware eingegangen wird.

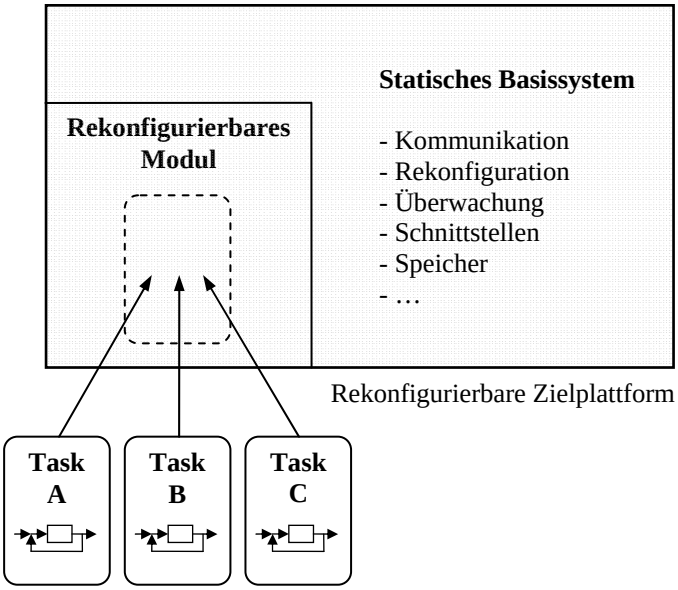


Abbildung 9: Rekonfigurierbare Module und Hardwaretasks

3.2.2 Partitionierung auf Hardwareebene

Die Partitionierung auf Hardwareebene erfolgt im ersten Schritt in Anlehnung an den in [51] beschriebenen *Early Access Partial Reconfiguration Design Flow* (EAPR Design Flow) von Xilinx. Ausgangspunkt sind die auf Funktionsebene ermittelten Teilmengen statischer und partiell rekonfigurierbarer Controllerfunktionen. Statische Teilfunktionen des Controllers werden einem statischen, nicht rekonfigurierbaren Bereich der Zielplattform zugewiesen. Der statische Bereich wird analog zum EAPR Design Flow als Basissystem bezeichnet und umfasst neben den statischen Teilfunktionen sämtliche I/O-Verbindungen und globale Ressourcen des Controllers.

Die Teilmenge der zur Laufzeit des Controllers austauschbaren Algorithmen wird dagegen mit Hilfe rekonfigurierbarer Hardwarebereiche und dort gekapselter Hardwarefunktionen auf der Zielplattform abgebildet. Die Hardwarebereiche werden hier abweichend vom EAPR Design Flow als *rekonfigurierbare Module* und die Hardwarefunktionen als *Hardwaretasks* bezeichnet. Rekonfigurierbare Module stellen feste Hardwarebereiche der Zielplattform dar, die als Platzhalter für verschiedene ihnen zugeordnete Hardwaretasks dienen. Hardwaretasks sind Funktionen, die zur Laufzeit des Controllers mit Hilfe der partiellen Rekonfiguration in rekonfigurierbare Module geladen und aktiviert werden können. Sie stellen die eigentliche rekonfigurierbare Controllerfunktionalität dar und implementieren betriebszustandsabhängige Steuer- und Regelalgorithmen. In einem rekonfigurierbaren Modul ist zu einem Zeitpunkt immer nur ein Hardwaretask aktiv. Ein rekonfigurierbarer Controller kann jedoch über mehrere rekonfigurierbare Module verfügen. Abbildung 9 illustriert das Prinzip rekonfigurierbarer Module und Hardwaretasks.

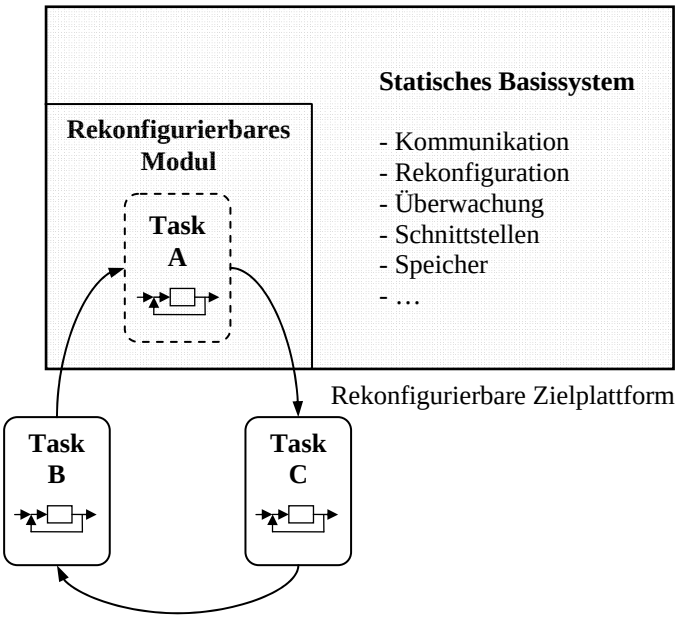


Abbildung 10: Abbildung von Hardwaretasks auf Module mit Zustandsautomaten

Der zweite Schritt der Partitionierung auf Hardwareebene ist die Zuordnung von Hardwaretasks zu einem rekonfigurierbaren Modul. Die Zuordnung erfolgt ausgehend von der Spezifikation der logischen Controllerstruktur und ist fest. Zeitlich aufeinander folgende Funktionen, die abhängig vom jeweiligen Betriebszustand des Controllers sind, können einem gemeinsamen rekonfigurierbaren Modul zugewiesen werden. Zeitlich unabhängig voneinander aktive Funktionen erfordern getrennte rekonfigurierbare Module. Eine implizite Zuordnung von Hardwaretasks zu rekonfigurierbaren Modulen ergibt sich aus dem Einsatz von Zustandsau-

tomaten für die Spezifikation der logischen Controllerstruktur und -funktionalität. Hier können Betriebszustände und ihre spezifischen Controllerfunktionen direkt auf rekonfigurierbare Module und Hardwaretasks abgebildet werden. Abbildung 10 verdeutlicht diesen Ansatz. Die Erweiterung der Vorgehensweise auf ein verteiltes Rekonfigurationsmanagement wird im folgenden Abschnitt vorgestellt. Schnittstellen und Ressourcenbedarf rekonfigurierbarer Module und der von ihnen implementierten Hardwaretasks werden in Kapitel 5 diskutiert.

3.3 Spezifikation rekonfigurierbarer Funktionalitäten

Ausgehend von der Partitionierung rekonfigurierbarer Controller in Module und zugehörige Hardwaretasks wird in diesem Abschnitt der Arbeit eine Spezifikationsmethodik für rekonfigurierbare Funktionalitäten vorgestellt. Die Methodik unterstützt implizit ein verteiltes Rekonfigurationsmanagement, das aus einer funktionalen Beschreibung rekonfigurierbarer Controller abgeleitet wird. Das verteilte Rekonfigurationsmanagement kann direkt in Hardware überführt werden, benötigt kein externes Kontrollsystem und ermöglicht die Selbstrekonfiguration. Basis der Spezifikationsmethodik ist die konsequente Anwendung von Zustandsautomaten für die Abbildung der Strukturvariabilität rekonfigurierbarer Hardware und die Aktivierung einzelner Teilfunktionen. Der Abschnitt schließt mit der Erweiterung der Methodik um eine echtzeitfähige Aktivierungsstrategie für Hardwaretasks.

3.3.1 Spezifikation rekonfigurierbarer Module

Die Spezifikation rekonfigurierbarer Module erfolgt auf der Basis hierarchischer Zustandsautomaten, die direkt in Hardware abgebildet werden. Der Einsatz von Zustandsautomaten ermöglicht eine implizite Berücksichtigung der Lade- und Aktivierungsvorgänge von Hardwaretasks, indem Tasks als Zustände eines Moduls interpretiert werden. Jedes rekonfigurierbare Modul eines Controllers wird im Rahmen der Spezifikationsmethodik mit einem eigenen Zustandsautomaten beschrieben, dessen Zustände die je nach Betriebszustand zu ladenden und zu aktivierenden Hardwaretasks repräsentieren. Die Bedingungen für die Transitionen zwischen den Zuständen hängen typischerweise von Prozessgrößen, wie z.B. der Drehzahl eines Antriebssystems, und der Erfüllung bestimmter Test- oder Fehlerkriterien ab. Der das Modul auf diese Weise beschreibende Zustandsautomat wird als *Modul-FSM* bezeichnet. Die Abkürzung FSM steht für den englischen Begriff für Zustandsautomat, Finite State Machine.

Abbildung 11 zeigt ein verallgemeinertes Beispiel einer Modul-FSM mit drei Zuständen. Ausgehend vom Startzustand des Moduls, der durch Task 1 repräsentiert wird, wird bei Erfül-

lung der jeweiligen Transitionsbedingung der nächstfolgende Task in das Modul geladen und aktiviert. Eine an das Beispiel für Hochlauf und Betrieb eines mechatronischen Systems aus Abschnitt 3.1.3 angelehnte Umsetzung dieses Prinzips der Rekonfigurationssteuerung ist in Abbildung 12 dargestellt. Startzustand ist hier ein Hardwaretask, der statische Systemtests implementiert. Sind diese Tests abgeschlossen, wird ein Hardwaretask für dynamische Systemtests geladen und aktiviert. Sind auch diese Tests vollzogen, wechselt das System durch Laden und Aktivieren des nächsten Hardwaretasks in den Hauptbetriebszustand. Im Fehlerfall wird der Task für den Hauptbetriebszustand durch einen Fail-Safe-Task ersetzt und das System geht in einen sicheren Zustand. Die dargestellte Controllerstruktur wurde gegenüber Abschnitt 3.1.3 leicht vereinfacht, indem der Fail-Safe-Zustand nur aus dem Hauptbetriebszustand zu erreichen ist und im Falle eines sicheren Systemzustands direkt in den Neustart des Hochlaufs übergeht.

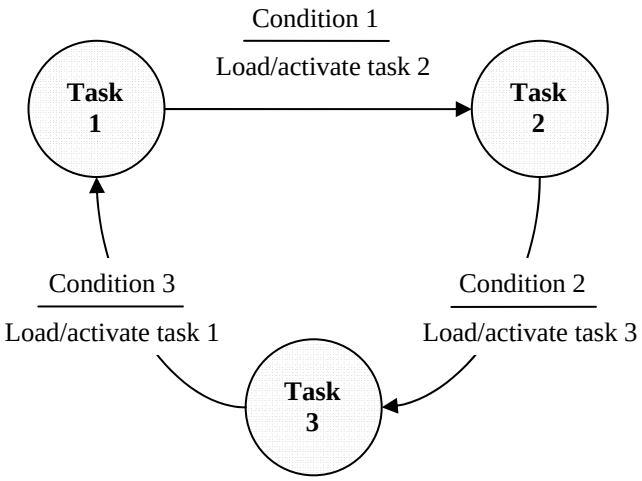


Abbildung 11: Beispiel einer Modul-FSM

Da die Struktur einer Modul-FSM bereits zu frühen Entwurfszeitpunkten festgelegt wird, kann sie in ihre einzelnen Zustände herunter gebrochen und in den Hardwaretasks des betreffenden Moduls direkt implementiert werden. Die Hardwaretasks stellen damit nicht nur die Zustände der Modul-FSM dar, sondern implementieren zusätzlich die ausgehende Transition und initiieren die Lade- und Aktivierungsvorgänge für nachfolgende Tasks. Eine Modul-FSM ist daher zu keinem Zeitpunkt vollständig auf der Zielplattform vorhanden, sondern wird nur teilweise durch den jeweilig aktiven Hardwaretask implementiert. Die Modul-FSM als abstraktes Spezifikationsmittel ermöglicht so ein implizites und verteiltes Rekonfigurationsmanagement, das basierend auf der funktionalen Spezifikation bereits zu frühen Entwurfszeitpunkten integriert wird.

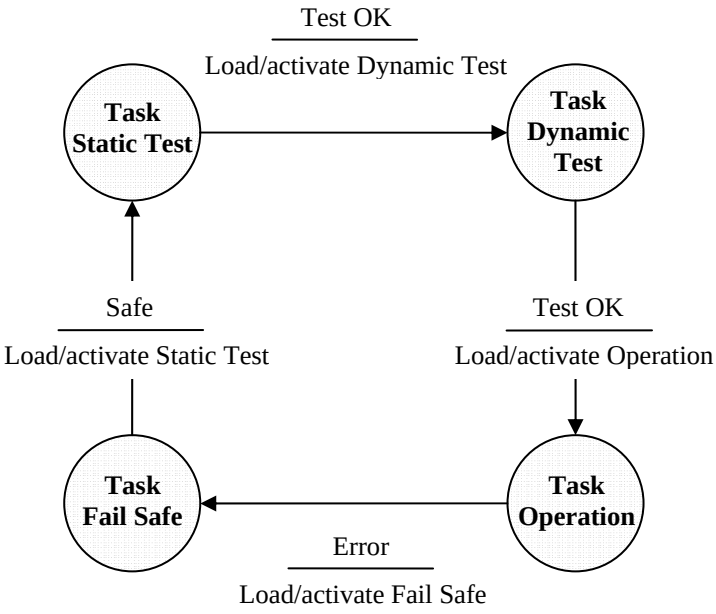


Abbildung 12: Modul-FSM für Hochlauf und Betrieb eines mechatronischen Systems

3.3.2 Spezifikation von Hardwaretasks

Die Spezifikation von Hardwaretasks erfolgt primär mit den in Abschnitt 3.1.2 beschriebenen Methoden und dient der Abbildung von Steuer- und Regelfunktionen auf rekonfigurierbare Hardware. Eingesetzt werden vor allem Blockdiagramme, die den Datenfluss der Steuer- und Regelalgorithmen beschreiben und mit Codegenerierungswerkzeugen in Hardwarebeschreibungssprachen überführt werden. Beispiele für die Spezifikation von Steuer- und Regelfunktionen in Hardwaretasks werden in Kapitel 6 vorgestellt.

Im Kontext der im vorherigen Abschnitt beschriebenen verteilten Rekonfigurationssteuerung muss die Spezifikation der Hardwaretasks weiterhin die Transitionen der übergeordneten Modul-FSM und die Lade- und Aktivierungsvorgänge für nachfolgende Tasks berücksichtigen. Für diese Aufgabe wird in jedem Hardwaretask eines Moduls ein zusätzlicher Zustandsautomat implementiert, der die Zustände des Tasks kontrolliert und Teile der Modul-FSM abbildet. Der Zustandsautomat wird als *Task-FSM* bezeichnet und ist für alle Hardwaretasks identisch. Er kann mit dem jeweiligen Task direkt auf die Hardware der Zielplattform übertragen werden. Zusätzlich wird ein globaler Bezeichner, der *Task-Marker*, für jeden Hardwaretask eingeführt. Mit dem Task-Marker wird der aktuell aktive Task gekennzeichnet und der Aktivierungsvorgang für zeitlich folgende Hardwaretasks initiiert. Der Austausch des Task-Markers zwischen Hardwaretasks und übergeordneten Ebenen erfolgt mit einem dedizierten

Kommunikationssystem, das in Kapitel 4 beschrieben wird und die Anbindung mehrerer rekonfigurierbarer Module unterstützt.

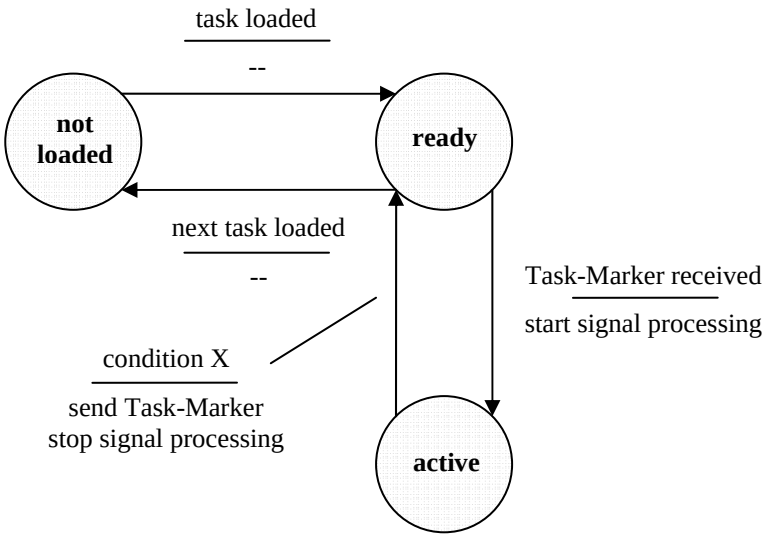


Abbildung 13: Task-FSM

Der Aufbau der Task-FSM ist an das Basis-Zustandsmodell für Softwaretasks in Echtzeitbetriebssystemen nach OSEK/VDX-OS [72] angelehnt, das unter anderem in [27] näher beschrieben wird. Abbildung 13 zeigt die Task-FSM und die Einbindung des Task-Markers zur Steuerung der Zustände eines Hardwaretasks. Die Task-FSM verfügt über die drei Zustände **not loaded**, **ready** und **active**. Im Zustand **not loaded** ist der Task nicht geladen und damit inaktiv. Ein Hardwaretask in diesem Zustand wird durch Konfigurationsdaten in einer Speichereinheit des Controllers repräsentiert. Im Zustand **ready** ist der Hardwaretask in sein übergeordnetes Modul geladen, nimmt jedoch nicht aktiv an der Signalverarbeitung teil. Der Übergang in den Zustand **active** und damit die aktive Signalverarbeitung erfolgt mit dem Empfang des Task-Markers des spezifischen Hardwaretasks. Die Abfrage des Task-Markers kann in dem einfachen Fall eines Controllers mit nur einem rekonfigurierbaren Modul vernachlässigt werden, da laden und aktivieren eines Hardwaretasks hier identisch sind. Wird im Zustand **active** eine bestimmte Bedingung wie das Eintreten eines Fehlerfalls oder das Ende einer Testfunktion erfüllt, sendet die Task-FSM den Task-Marker für den nächsten zu aktivierenden Task, stoppt die Signalverarbeitung und geht in den Zustand **ready** zurück. Die Transition entspricht dabei dem Zustandsübergang der übergeordneten Modul-FSM. Ist der nächstfolgende Task in das Modul geladen, befindet sich der vorige Task wieder im Zustand **not loaded**.

3.3.3 Aktivierungsstrategien

Für echtzeitfähige Controller ist innerhalb des Rekonfigurationsmanagements die Zeit zu berücksichtigen, die für den Ladevorgang der Hardwaretasks benötigt wird. Diese Zeit wird als Rekonfigurationszeit T_R bezeichnet und ist von der Größe des zu rekonfigurierenden Moduls, den Implementierungsmethoden und spezifischen Eigenschaften der Zielhardware abhängig. Typische Rekonfigurationszeiten moderner FPGAs, wie den in dieser Arbeit eingesetzten Xilinx Virtex-II und Virtex-4 Bausteinen, liegen im Bereich von Millisekunden. Für kleine rekonfigurierbare Module und mit spezialisierten Rekonfigurationslösungen können auch Rekonfigurationszeiten im Mikrosekundenbereich realisiert werden. Abschnitt 5.2 vertieft diesen Ansatz.

Der Einfluss der Rekonfigurationszeit T_R auf die Rekonfigurationssteuerung ergibt sich aus der Relation zur Abtastzeit T_S und Signalverarbeitungszeit T_P. Die Abtastzeit T_S resultiert aus dem Shannon-Theorem und damit aus dem zugrunde liegenden mechatronischen System. Die Signalverarbeitungszeit T_P ist dagegen von der eingesetzten Logik und Taktung eines Hardwaretasks abhängig. Ist die Summe aus T_R und T_P gleich oder kleiner als T_S, kann die Rekonfiguration eines Moduls und so der Ladevorgang eines Hardwaretasks innerhalb einer Abtastperiode ohne Datenverlust vorgenommen werden. Die Randbedingungen moderner mechatronischer Systeme lassen diese Aktivierungsstrategie nur für sehr kleine Rekonfigurationszeiten zu. Überschreitet die Summe aus T_R und T_P jedoch die Abtastzeit T_S, gehen ein oder mehrere Abtastschritte während des Ladevorganges eines Hardwaretasks verloren. Dieses Verhalten ist für Echtzeitsysteme nicht vertretbar, kann aber durch das Vorabladen von Hardwaretasks verhindert werden.

Das Vorabladen von Hardwaretasks wird auf der Zielplattform mit einer Erweiterung der Hardwareressourcen eines rekonfigurierbaren Moduls umgesetzt. So kann während der Laufzeit eines Hardwaretasks bereits der folgende Task geladen werden, ohne sofort aktiviert werden zu müssen. Wird der laufende Task beendet, kann ohne Verzögerung auf den bereits geladenen Task umgeschaltet werden. Die Spezifikation dieser echtzeitfähigen Vorablade- und Aktivierungsstrategie erfolgt mit der Task-FSM nach Abschnitt 3.3.2 sowie einer erweiterten Modul-FSM und ist vollständig in die vorgestellte Entwicklungsmethodik integrierbar. Abbildung 14 zeigt eine beispielhafte Modul-FSM, die ein rekonfigurierbares Modul mit je einem aktiven und einem vorab geladenen Task beschreibt. Im Gegensatz zu einer Modul-FSM nach Abschnitt 3.3.1 wird hier der Ladevorgang für Hardwaretasks bereits einen Zustand früher initiiert, so dass zum Aktivierungszeitpunkt lediglich eine Umschaltung erfolgt.

Da im Beispiel ein Hardwaretask vorab geladen wird, ergibt sich ein verdoppelter Ressourcenbedarf für das rekonfigurierbare Modul auf der Zielplattform.

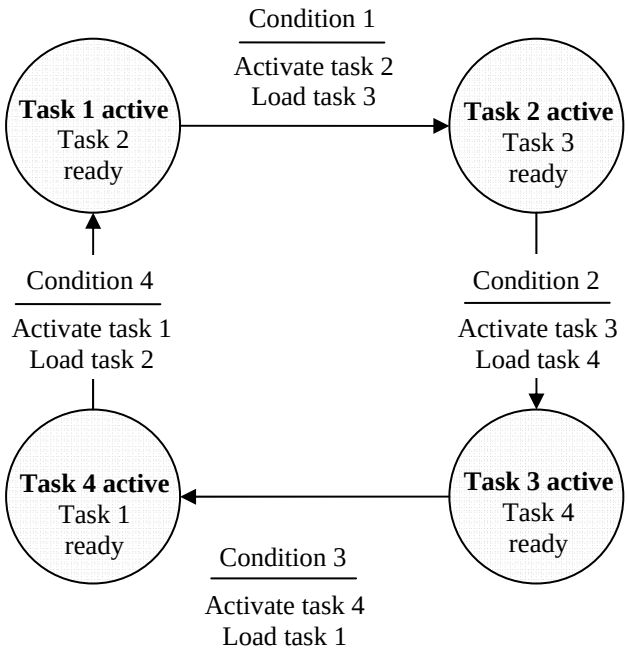


Abbildung 14: Erweiterte Modul-FSM für Vorabladen von Hardwaretasks

Der Ressourcenbedarf eines rekonfigurierbaren Moduls vergrößert sich proportional mit jedem vorab zu ladenden Hardwaretask. Die auf Vorabladen basierende Aktivierungsstrategie ist daher nur sinnvoll mit rekonfigurierbaren Modulen nutzbar, für deren Gesamtzahl möglicher Betriebszustände N_S und Anzahl der durch Betriebszustandswechsel erreichbarer Zustände N_R gilt:

$$N_R + 1 < N_S \qquad \text{Gleichung 3-1}$$

Ein iteratives Durchlaufen von Teilen des V-Modells nach Abschnitt 3.1.1 bietet darüber hinaus die Möglichkeit, die Partitionierung und Spezifikation der rekonfigurierbaren Funktionalitäten in Bezug auf den Ressourcenbedarf der resultierenden Module und Tasks zu optimieren.

4 Struktur und Funktionsgruppen

Das vierte Kapitel der Arbeit erläutert Struktur und Funktionsgruppen rekonfigurierbarer Controller für mechatronische Systeme. Schwerpunkt des Kapitels ist der Entwurf statischer Komponenten. Ausgehend von der in Kapitel 3 vorgestellten Entwurfsmethodik wird eine grundlegende technische Struktur FPGA-basierter rekonfigurierbarer Controller entwickelt, die rekonfigurierbare Module und Hardwaretasks abbildet und um statische Infrastrukturkomponenten ergänzt. Die entwickelten statischen Funktionsgruppen umfassen eine dedizierte Kommunikationslösung für rekonfigurationsrelevante Daten, Komponenten für das Rekonfigurations- und Speichermanagement sowie für die Zustandssicherung und Zustandswiederherstellung rekonfigurierbarer Steuer- und Regelfunktionen. Weiterhin werden Prozessschnittstellen und Schnittstellen für die Kommunikation mit übergeordneten Systemen und ihre Integration in rekonfigurierbare Controller vorgestellt. Das Kapitel schließt mit der Beschreibung eines Spezifikationsframeworks für rekonfigurierbare und statische Komponenten rekonfigurierbarer Controller. Grundlage sind hier Funktionsbibliotheken und Softwarewerkzeuge für Entwurf, funktionale Verifikation und Umsetzung der Komponenten.

Der erste Abschnitt des Kapitels beschreibt die Struktur rekonfigurierbarer Controller für mechatronische Systeme und die grundlegende Abbildung der Funktionalitäten auf FPGA-Hardware. Im zweiten Abschnitt des Kapitels werden statische Funktionsgruppen und Infrastrukturkomponenten FPGA-basierter rekofigurierbarer Controller vorgestellt. Der dritte Abschnitt beschreibt das Spezifikationsframework für Komponenten und Funktionen rekonfigurierbarer Controller.

4.1 Struktur rekonfigurierbarer Controller

Die technische Struktur rekonfigurierbarer Controller bestimmt die grundlegende Abbildung von Controllerkomponenten und deren Vernetzung auf der Zielplattform, definiert Schnittstel-

len und gibt Rahmenbedingungen für die Implementierung der Controller vor. Sie stellt damit den Zusammenhang zwischen den abstrakten Spezifikationsmethoden nach Kapitel 3 und den herstellerabhängigen Implementierungsmethoden nach Kapitel 5 her.

Der Entwurf der technischen Struktur rekonfigurierbarer Controller für mechatronische Systeme erfolgt ausgehend von der gewählten Zielhardware und den in Kapitel 3 beschriebenen Partitionierungs- und Spezifikationsmethoden. So wird die Strukturvariabilität partiell rekonfigurierbarer FPGAs eingesetzt, um rekonfigurierbare Module und die zugehörigen Hardwaretasks auf der Zielhardware abzubilden. Ergänzt werden die rekonfigurierbaren Funktionalitäten durch das nicht rekonfigurierbare Basissystem und übergeordnete Kommunikations- und Speicherlösungen. Abbildung 15 illustriert diesen Ansatz mit der beispielhaften Struktur eines rekonfigurierbaren Controllers, der über zwei rekonfigurierbare Module verfügt.

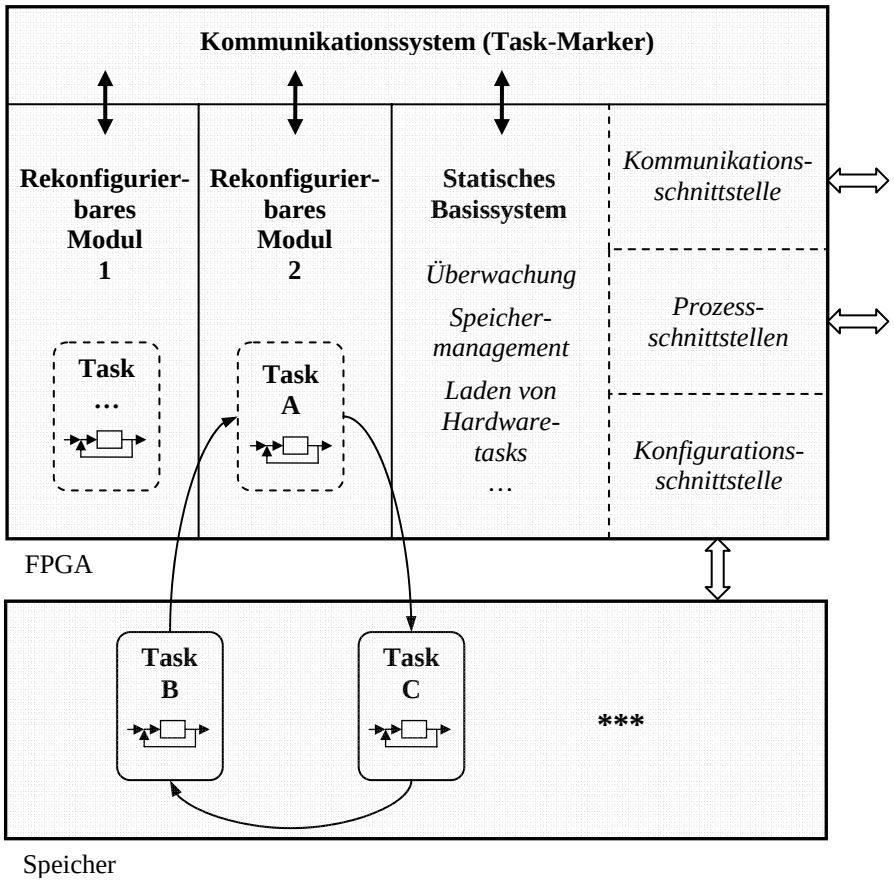


Abbildung 15: Struktur rekonfigurierbarer Controller für mechatronische Systeme

Die im Beispiel dargestellte Controllerstruktur setzt sich auf der obersten Ebene aus der Zielplattform FPGA und einer Speichereinheit zusammen. Die Speichereinheit ist für die Speicherung der Konfigurationsdaten der einzelnen Hardwaretasks zuständig. Sie kann mit externen Speicherbausteinen realisiert oder auch in dedizierten Speicherblöcken des FPGAs implemen-

tiert werden. Die Umsetzung in dedizierten Speicherblöcken des FPGAs ist von der Größe und Anzahl der zu speichernden Konfigurationsdaten abhängig. Ausreichende Speicherressourcen können in den meisten Fällen nur von großen und entsprechend kostenintensiven Bausteinen bereitgestellt werden.

Der Hauptteil rekonfigurierbarer Controller für mechatronische Systeme wird auf die Zielhardware FPGA abgebildet. Form, Größe, Schnittstellen und Anzahl der rekonfigurierbaren Module eines Controllers sind dabei von der konkreten Zielhardware und dem Anwendungsfall abhängig. Kapitel 5 geht näher auf diese Problematik ein und beschreibt den aufwandsreduzierenden Einsatz bitserieller Algorithmen und Signalübertragungsverfahren. Soll darüber hinaus die in Abschnitt 3.3.3 beschriebene und auf Vorabladen basierende Aktivierungsstrategie für Hardwaretasks eingesetzt werden, muss dies bei der Spezifikation der technischen Struktur eines Controllers berücksichtigt werden. Ausgehend vom erforderlichen Ressourcenbedarf werden einzelne Module zusammengefasst und als Supermodul interpretiert. Im Falle des Beispiels nach Abbildung 15 kann so durch Zusammenfassen von Modul 1 und Modul 2 zu einem Supermodul ein Hardwaretask vorab geladen werden. Dieser Ansatz ist jedoch nur für einfache Modul-FSMs mit einem angemessenen Ressourcenaufwand realisierbar. Daher wurde in der vorliegenden Arbeit eine echtzeitfähige und schnelle Rekonfigurationslösung entwickelt, die eine direkte Lade- und Aktivierungsstrategie ohne das Vorabladen von Hardwaretasks unterstützt (vergleiche Abschnitt 5.2).

Das statische Basissystem ergänzt den Controller um Schnittstellen und beinhaltet wesentliche Teile des Rekonfigurations- und Speichermanagements. Die Controllerschnittstellen setzen sich dabei aus einer Kommunikationsschnittstelle und einer oder mehreren Prozessschnittstellen zusammen. Die Kommunikationsschnittstelle realisiert die Anbindung des Controllers an übergeordnete Steuer- und Regelstrukturen und kann für den Transfer von Konfigurationsdaten eingesetzt werden. Die Anbindung von für die Steuerung und Regelung des Systems wichtigen Prozesssignalen, wie Istwerten und Stellwerten, erfolgt über die Prozessschnittstellen, deren Ausprägung damit vom konkreten Anwendungsfall abhängig ist. Beide Schnittstellentypen können zusätzlich über FPGA-externe Bestandteile verfügen, die analoge und physikalische Schnittstellenebenen umsetzen. Da die Controllerschnittstellen als Teil des statischen Basissystems implementiert werden, können sie nur durch die Umprogrammierung des gesamten FPGAs ausgetauscht werden. Das in Kapitel 3 vorgestellte Konzept von rekonfigurierbaren Modulen und Hardwaretasks kann jedoch unter Beachtung der dann sehr komplexen Modulschnittstellen auf die Controllerschnittstellen ausgedehnt werden.

Rekonfigurations- und Speichermanagement implementieren innerhalb des statischen Basissystems die Steuerung, Überwachung und Umsetzung der partiellen Rekonfiguration. Das Speichermanagement realisiert dabei die Verwaltung und den Zugriff auf die Konfigurationsdaten der Hardwaretasks, während das Rekonfigurationsmanagement die Ladevorgänge umsetzt. Dies umfasst auch den Zugriff auf die interne Konfigurationsschnittstelle des FPGAs und ermöglicht die Selbstrekonfiguration eines Controllers. Der Begriff Selbstrekonfiguration umschreibt in diesem Zusammenhang die Fähigkeit des Systems, den Zeitpunkt der partiellen Rekonfiguration selbst zu bestimmen und diese eigenständig und unabhängig von externen Konfigurationslösungen auszuführen. Für die Initiierung eines Ladevorganges relevante Daten wie der Task-Marker werden zwischen den Modulen und dem statischen Basissystem eines Controllers über ein übergeordnetes und dediziertes Kommunikationsystem ausgetauscht.

Die einzelnen Funktionsgruppen des statischen Basissystems sowie das Kommunikationssystem werden im folgenden Abschnitt der Arbeit detailliert beschrieben und die Spezifikation der technischen Struktur rekonfigurierbarer Controller für mechatronische Systeme damit vervollständigt.

4.2 Funktionsgruppen

Die Funktionsgruppen rekonfigurierbarer Controller für mechatronische Systeme sind Bestandteil der technischen Controllerstruktur und umfassen die nicht rekonfigurierbaren Funktionalitäten. Diese statischen Infrastrukturkomponenten werden im Basissystem des Controllers implementiert und durch ein übergeordnetes Kommunikationssystem für rekonfigurationsbezogene Daten ergänzt. Die Entwicklung der Funktionsgruppen kann durch einen eigenen Unterzweig des V-Modells für den Entwurf rekonfigurierbarer Controller nach Abbildung 7 beschrieben werden. Dabei werden die Entwurfsschritte Spezifikation, Design und Implementierung sowie Test der Komponenten analog zu den rekonfigurierbaren Funktionalitäten durchlaufen. Es werden die in Abschnitt 3.1 vorgestellten Spezifikationswerkzeuge verwendet. Da es sich hier jedoch um nicht rekonfigurierbare Funktionalitäten handelt, werden Zustandsautomaten vorrangig zur direkten Beschreibung von Abläufen sowie Spezifikation von Protokollen und nicht zur Abbildung von Strukturvariabilitäten eingesetzt. Der Entwurf der Funktionsgruppen ist zum Teil von der konkreten Zielhardware abhängig und bezieht sich hier auf Xilinx FPGAs. Weiterhin wird die Zielplattform FPGA in Einzelfällen durch FPGA-externe Bausteine ergänzt.

4.2.1 Kommunikationssystem

Das dedizierte Kommunikationssystem verbindet die rekonfigurierbaren Module und das statische Basissystem eines rekonfigurierbaren Controllers. Seine Hauptaufgabe ist der sichere und fehlerfreie Transfer des Task-Markers und anderer rekonfigurationsrelevanter Daten zwischen dem Basissystem und den Task-FSMs der in den Modulen geladenen Hardwaretasks. Die grundlegende Struktur des Kommunikationssystems ist in Abbildung 16 dargestellt.

Das Kommunikationssystem setzt sich aus einem persistenten Kommunikationsmakro und den Teilnehmeranschlüssen für die rekonfigurierbaren Module/Hardwaretasks sowie das Basissystem zusammen. Das persistente Makro wird dabei über fest definierte Leitungen und Elemente im FPGA realisiert, die nicht von der partiellen Rekonfiguration beeinflusst werden. Auf diese Weise kann die Kommunikation während einer Rekonfiguration auch über Modulgrenzen hinaus gewährleistet werden. Kapitel 5 geht näher auf diese auch als Bus-Makros bezeichneten Leitungen und Elemente und deren Grundlagen ein. Neben Kommunikationsleitungen umfasst das persistente Makro je ein Register und einen Multiplexer pro Teilnehmer. Das Kommunikationsmakro wird in den einzelnen Teilnehmern durch lokale Zustandsautomaten (*Communication System FSM*, CS-FSM) ergänzt, die zusammen mit den Registern den Teilnehmeranschluss repräsentieren.

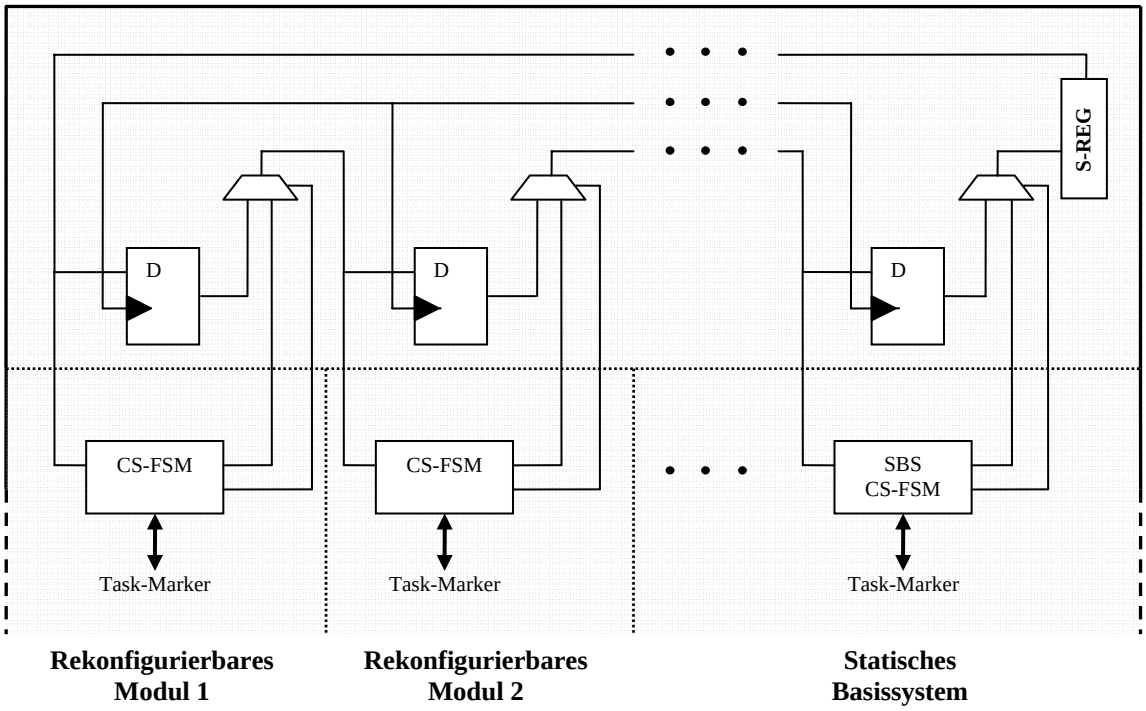


Abbildung 16: Aufbau des Kommunikationssystems

Die funktionale Struktur des Kommunikationssystems wird durch die Kommunikationsleitungen und die Teilnehmerregister bestimmt. Diese bilden gemeinsam ein Schieberegister, dessen Speichereinheiten auf die einzelnen Teilnehmer aufgeteilt sind. Jeder Teilnehmer speichert so ein Bit der auszutauschenden Daten und schiebt den Speicherinhalt mit jedem Systemtaktzyklus zum nächsten Teilnehmer weiter. Enthalten die auszutauschenden Daten mehr Bits als Teilnehmer am Kommunikationssystem teilnehmen, kann im statischen Basissystem ein zusätzliches Schieberegister (S-REG) vorgesehen werden, das diesen Unterschied ausgleicht. Der Zugriff eines Teilnehmers auf das Kommunikationssystem wird durch den lokalen Multiplexer gesteuert. Die Interaktion zwischen Teilnehmeranschluss und Kommunikationsmakro während der Rekonfiguration eines Moduls wird verhindert, indem das lokale Reset-Signal des Moduls durch den Multiplexer ausgewertet wird. Die Reset-Signale werden durch das Rekonfigurationsmanagement im Basissystem kontrolliert, so dass der fehlerfreie Betrieb des Kommunikationssystems auch während der Rekonfiguration der einzelnen Teilnehmer gewährleistet werden kann.

Die mit dem Kommunikationssystem zu übertragenden Daten sind applikationsabhängig und werden mit den lokalen Zustandsautomaten der Teilnehmer während der Spezifikation der Module und Hardwaretasks festgelegt. Im einfachsten Fall sind lediglich die Task-Marker der zu aktivierenden und ladenden Hardwaretasks auszutauschen, so dass die um Schiebefunktionen erweiterten Task-FSMs direkt an das Kommunikationsmakro angeschlossen werden können. Die Erweiterung dieses Konzepts auf die Vorabladestrategie nach Abschnitt 3.3.3 und mehrere rekonfigurierbare Module bedingt jedoch ein komplexeres Kommunikationsprotokoll. Die Ankopplung der Module an das Kommunikationssystem wird in diesen Fällen mit dedizierten Zustandsautomaten realisiert, die lesend und schreibend auf die Teilnehmerregister des Kommunikationsmakros zugreifen.

Ein im Rahmen der Arbeit umgesetztes beispielhaftes Protokoll besteht aus mehreren Startbits, einem Kennzeichner für die Nutzung des Datenbereichs und dem von der konkreten Implementierung abhängigen Datenbereich. Der Datenbereich beinhaltet dabei immer den aktuellen Task-Marker und kann durch weitere applikationsspezifische Daten ergänzt werden. Die Startbits dienen der Synchronisierung der Teilnehmer auf das Protokoll. Die Schreibberechtigung eines Teilnehmers für den Datenbereich wird mit dem Kennzeichner gegeben. Ein Teilnehmer kann nur dann schreibend zugreifen, wenn dieses Bit nicht gesetzt ist. Ist der Kennzeichner gesetzt, wird lesend auf die Daten zugegriffen und diese an die Task-FSM weitergegeben. Das Rücksetzen des Kennzeichnerbits geschieht durch den schreibenden Teilnehmer

nach erfolgtem Transport der Daten durch das von allen Teilnehmern gebildete Schieberegister. Eine höhere Priorisierung der Task-Marker bestimmter Hardwaretasks wie Fail-Safe-Funktionen kann hier durch Vergleich und Überschreiben des Datenbereichs trotz gesetztem Kennzeichnerbit erreicht werden.

Eine alternative Implementierung des Kommunikationssystems auf Basis einer direkten bitparallelen Übertragung der Task-Marker zwischen den rekonfigurierbaren Modulen und dem statischen Basissystem ist ebenfalls möglich. Grundlage einer solchen Implementierung sind bitparallele fest definierte Leitungen (Bus-Makros), die einen Bus bilden. Der Bus umfasst dabei Leitungen für die zu transportierenden Daten sowie mindestens eine Steuerleitung, die den Zugriff auf den Bus analog zum Kennzeichnerbit der Schieberegisterimplementierung steuert. Die elementaren Komponenten des Kommunikationssystems wie die CS-FSMs und die lokalen Multiplexer werden weiterhin wie oben beschrieben für die Ankopplung der Teilnehmer eingesetzt. Die bitparallele Variante des Kommunikationssystems ist aufgrund der komplexen Struktur der Bus-Makros jedoch nur mit erhöhtem Implementierungs- und Ressourcenaufwand umzusetzen.

In allen hier genannten Fällen verfügt der Teilnehmeranschluss des statischen Basissystems zusätzlich über Funktionen, die eine Initiierung und so auch den Start und Neustart des Kommunikationssystems realisieren. Dies beinhaltet das Senden des initialen Task-Markers, der den ersten Hardwaretask aktiviert und damit die Signalverarbeitung und das Kommunikationssystem startet. Weiterhin verfügt der Teilnehmeranschluss des Basissystems über eine Verbindung zum Rekonfigurations- und Speichermanagement des Controllers und übermittelt diesem die empfangenen Daten einschließlich der Task-Marker.

4.2.2 Rekonfigurations- und Speichermanagement

Das Rekonfigurations- und Speichermanagement rekonfigurierbarer Controller realisiert die Handhabung der Konfigurationsdaten von Hardwaretasks sowie die Steuerung und Ausführung der partiellen Rekonfiguration. Die Aufgaben der Funktionsgruppe umfassen die Verwaltung von Konfigurationsdatensätzen, den Zugriff auf die Konfigurationsdaten zu ladender Hardwaretasks und die Ansteuerung der internen Konfigurationsschnittstelle der Zielhardware. Die Implementierung des Rekonfigurations- und Speichermanagements erfolgt im statischen Basissystem der rekonfigurierbaren Controller und basiert auf direkt in FPGA-Hardware umgesetzten Zustandsautomaten. Die grundlegende Struktur der Funktionsgruppe im Kontext eines rekonfigurierbaren Controllers ist in Abbildung 17 dargestellt.

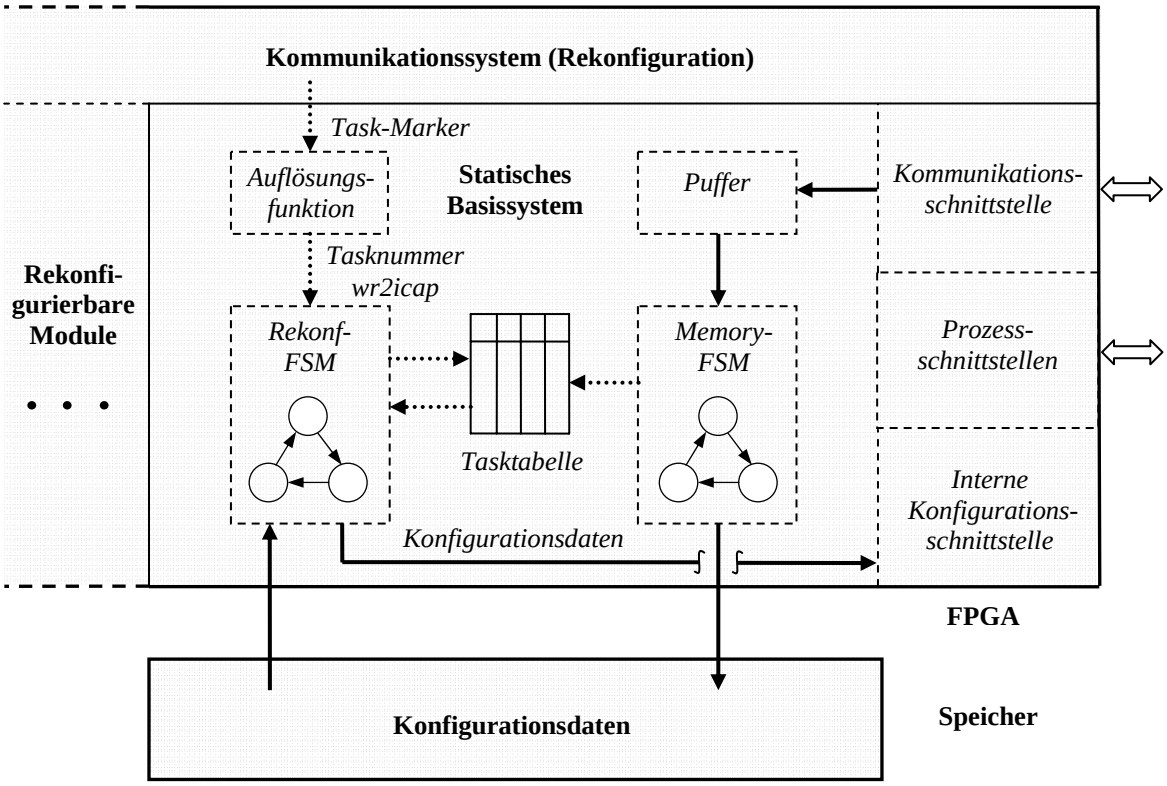


Abbildung 17: Rekonfigurations- und Speichermanagement

Die Konfigurationsdaten für Hardwaretasks werden mit den in Kapitel 5 beschriebenen herstellerspezifischen Implementierungsmethoden erzeugt und liegen als so genannte Bitstreams in Form von Binärdateien vor. Diese werden von einem übergeordneten System zu einem beliebigen Zeitpunkt an die Kommunikationsschnittstelle eines rekonfigurierbaren Controllers gesendet und dort in einen als Dual-Port-RAM ausgeführten Pufferspeicher geschrieben, der die Schnittstelle von anderen Bestandteilen des Basissystems entkoppelt. Die zentrale Komponente des Speichermanagements ist ein Zustandsautomat (Memory-FSM), der auf den Pufferspeicher zugreift und die empfangenen Konfigurationsdaten in den bereits in Abschnitt 4.1 beschriebenen Hauptspeicher des Controllers schreibt. Dabei werden Startadresse und Anzahl der gespeicherten Bytes pro Konfigurationsdatensatz in eine spezielle Tasktabelle eingetragen und einer festen Tasknummer zugeordnet. Um diese Art der Speicherung zu ermöglichen, werden die Konfigurationsdaten vor dem Senden mit einem Nachrichtenkopf (Header) versehen. Der Kopf enthält Informationen über die Art der folgenden Daten, die Tasknummer, die Anzahl der gesendeten Tasks und die Anzahl der folgenden Datenbytes. Tabelle 1 illustriert den Aufbau des Nachrichtenkopfes. Die Informationen werden durch die Memory-FSM beim Zugriff auf den Pufferspeicher ausgewertet. So werden Konfigurationsdaten unter Auswertung der Tasknummer in Hauptspeicher und Tasktabelle abgelegt und sonstige Daten an andere Komponenten des Controllers weitergeleitet. Ein Überschreiben der gespeicherten Konfi-

gurationsdaten eines Hardwaretasks ist auch zur Laufzeit des Controllers möglich. Weiterhin kann die Memory-FSM für einzelne Anwendungen so modifiziert werden, dass eintreffende Konfigurationsdaten sofort für die direkte Rekonfiguration eines Moduls verwendet werden.

Bytenummer	Inhalt/Daten	Anmerkung
0	0000.000X	X = 1 → Konfigurationsdaten X = 0 → sonstige Daten
1	YYYY.ZZZZ	YYYY = Tasknummer ZZZZ = Gesamtanzahl der Tasks (0…ZZZZ)
2	Konfigurationsdatenbytes I	Bereich für die Anzahl gesendeter Konfigurationsdatenbytes
3	Konfigurationsdatenbytes II	
4	Konfigurationsdatenbytes III	
5	Konfigurationsdaten	Bereich für übermittelte Konfigurationsdaten …
6	…	
7	…	
…	…	

Tabelle 1: Aufbau des Nachrichtenkopfes für die Übermittlung von Konfigurationsdaten

Neben Speicherung und Verwaltung der Konfigurationsdaten implementiert die Funktionsgruppe weiterhin den Zugriff auf die Daten im Falle einer Rekonfiguration. Das Laden eines Hardwaretasks wird durch den Empfang eines Task-Markers über das im vorhergehenden Abschnitt beschriebene Kommunikationssystem initiiert. Der Teilnehmeranschluss des statischen Basissystems übermittelt den empfangenen Task-Marker an eine Auflösungsfunktion, die den Zusammenhang zwischen Task-Marker und zu ladendem Hardwaretask herstellt. Die Auflösung ist notwendig, um bei Einsatz der Vorabladestrategie nach Abschnitt 3.3.3 zwischen zu aktivierenden und zu ladenden Hardwaretasks zu unterscheiden. Die Auflösungsfunktion kann dabei als einfache Wahrheitstabelle oder Zustandsautomat beschrieben werden. Im nächsten Schritt werden die resultierende Tasknummer und ein Schreibbefehl (*wr2icap*) an den Rekonfigurationsmechanismus übergeben oder gehalten bis dieser frei ist. Der Rekonfigurationsmechanismus wird durch einen Zustandsautomaten (Rekonf-FSM) repräsentiert, der nach Erhalt der Tasknummer sowie des Schreibbefehls auf die Tasktabelle zugreift und den Speicherbereich des zu ladenden Hardwaretasks ermittelt. Daraufhin liest der Rekonfigurationsmechanismus die Konfigurationsdaten aus dem Hauptspeicher des Controllers und schreibt diese direkt an die interne Konfigurationsschnittstelle des FPGAs. Eine detaillierte

Beschreibung des Rekonfigurationsmechanismus und der Ansteuerung der internen Konfigurationsschnittstelle ICAP in Xilinx FPGAs wird in Kapitel 5 gegeben.

4.2.3 Zustandssicherung und Zustandswiederherstellung

Die Funktionsgruppen für Zustandssicherung und Zustandswiederherstellung in rekonfigurierbaren Controllern realisieren die Erfassung von veränderlichen Speicherinhalten in Hardwaretasks und deren Wiederherstellung nach einer partiellen Rekonfiguration.

Relevante Komponenten für eine Zustandssicherung und -wiederherstellung in Hardwaretasks sind vor allem Integrations- und Differentiationsglieder der dort implementierten Reglerstrukturen. Diese verfügen über einen internen Speicher, dessen Inhalt sich mit jedem Abtastschritt des Controllers ändern kann und die Ausgangswerte der Komponenten direkt beeinflusst. Im Fall einer partiellen Rekonfiguration des übergeordneten rekonfigurierbaren Moduls gehen die Speicherwerte verloren und stehen dem folgenden Hardwaretask oder späteren Hardwaretasks nicht mehr zur Verfügung. Falls der neue Hardwaretask ebenfalls über Komponenten mit internem Speicher verfügt, bewirkt der Verlust der Speicherwerte ein eingeschränktes Regelverhalten bis diese wiederhergestellt sind. Ein Beispiel stellt ein Hardwaretask mit aktivem Proportional-Integral-Drehzahlregler für ein elektrisches Antriebssystem dar, der durch einen Hardwaretask mit neu parametriertem Regler oder einer erweiterter Reglerstruktur ersetzt wird. Das Wiedererreichen der Speicherwerte im neuen Hardwaretask benötigt daraufhin eine bestimmte Anzahl von Abtastschritten, während der das betreffende Integrationsglied nur eingeschränkt an der Generierung des Drehzahlstellwertes beteiligt ist.

Die in dieser Arbeit entwickelten Funktionen für Zustandssicherung und Zustandswiederherstellung implementieren eine automatisierte Wiederherstellung der Speicherinhalte einzelner Komponenten und ermöglichen so den Übergang zwischen Hardwaretasks ohne die oben beschriebene negative Beeinflussung des Systemverhaltens. Die grundlegende Struktur der Funktionsgruppen für Zustandssicherung und -wiederherstellung ist in Abbildung 18 am Beispiel eines Integrationsgliedes dargestellt. In den Hardwaretasks werden die Komponenten, deren Zustände gesichert und wiederhergestellt werden sollen, um Schnittstellen für das Auslesen (*mem_out*) und Schreiben (*mem_in*) der Speicherinhalte erweitert. Zusätzlich verfügen diese Komponenten über weitere Schnittstellensignale für die Steuerung der Lese- und Schreibvorgänge. Auf der Systemebene implementiert das statische Basissystem des Controllers eine übergeordnete Überwachungseinheit für die Zustandswiederherstellung und eine zentrale Funktionsgruppe zur Speicherung der Zustände (MEM_RECON) jedes rekonfigu-

rierbaren Moduls, dessen Hardwaretasks eine Zustandssicherung und -wiederherstellung erfordern. Die Verbindung zwischen den Elementen der Komponentenebene und der Systemebene erfolgt mit fest definierten und nicht von der partiellen Rekonfiguration beeinflussten Leitungen, den so genannten Bus-Makros (siehe auch Kapitel 5). Die Signale für die Zustandssicherung und -wiederherstellung sind damit Bestandteil der Schnittstellen des übergeordneten rekonfigurierbaren Moduls und können an das persistente Kommunikationsmakro nach Abschnitt 4.2.1 angegliedert werden.

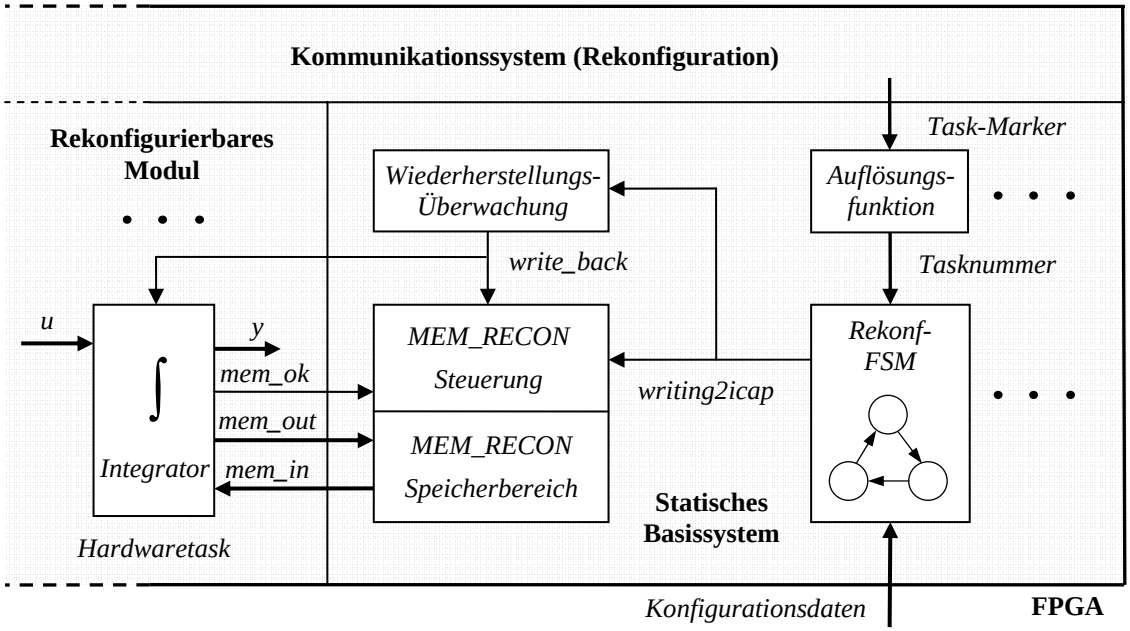


Abbildung 18: Zustandssicherung und Zustandswiederherstellung

Die Zustandssicherung für Komponenten mit der beschriebenen Schnittstellenerweiterung erfolgt mit jedem Abtastschritt des Controllers und wird von der Komponente durch Senden des Signals *mem_ok* initiiert. Die zu speichernden Daten liegen dabei am Eingang der Funktionsgruppe für die Speicherung der Zustände im statischen Basissystem an und werden in einem Register abgelegt. Im Falle einer partiellen Rekonfiguration des betreffenden Moduls sendet das Rekonfigurationsmanagement nach Abschnitt 4.2.2 das Signal *writing2icap* an die Funktionsgruppe für die Speicherung der Zustände, die daraufhin den zuletzt erfassten Zustand dauerhaft speichert. Der Zustand des neu geladenen Hardwaretasks wird danach analog mit jedem Abtastschritt und *mem_ok*-Signal in ein neues Register geschrieben. Für Modul-FSMs mit unverzweigter Taskreihenfolge ist eine sukzessive Speicherung der Zustände hinreichend. Basierend auf einer Übermittlung der Tasknummer des aktiven Hardwaretasks kann die Speicherung der Zustände darüber hinaus auch mit tabellarischer Zuordnung umgesetzt

werden. In beiden Fällen kann ein Hardwaretask über mehrere zu sichernde Zustände verfügen, die von ihnen zugeordneten Instanzen der Speicherfunktion erfasst werden.

Die Zustandswiederherstellung einer Komponente wird ebenfalls durch das Rekonfigurationsmanagement nach Abschnitt 4.2.2 veranlasst und durch die Funktionsgruppe für Wiederherstellungsüberwachung umgesetzt. Diese wertet das Rekonfigurationssignal *writing2icap* sowie für komplexe Modul-FSMs zusätzlich die Tasknummer aus und generiert nach erfolgtem Ladevorgang eines Hardwaretasks ein Signal für das Zurückschreiben eines Zustandes *(write_back)*. Die Auswertung und Signalgenerierung kann alternativ direkt in der Auflösungsfunktion des Rekonfigurationsmanagements erfolgen. Das Rückschreibesignal wird daraufhin an die Funktionsgruppe für die Speicherung der Zustände und die wiederherzustellende Komponente gesendet. Innerhalb der Funktionsgruppe für die Speicherung der Zustände wird im einfachsten Fall ein sukzessiver Zugriff auf die Register und deren Speicherinhalte analog zum Speichervorgang implementiert. Der wiederherzustellende Zustand wird dann an die Komponente gesendet und dort bei anliegendem *write_back*-Signal in den internen Speicher übernommen.

Die hier vorgestellte Zustandssicherung und Zustandswiederherstellung wurde sowohl für bitserielle als auch bitparallele Integrations- und Differentiationsglieder umgesetzt und ist auf weitere Komponenten übertragbar. Der Einsatz bitserieller Übertragungsmethoden zwischen Komponenten- und Systemebene senkt den Implementierungsaufwand deutlich (vergleiche Abschnitt 5.1).

4.2.4 Controllerschnittstellen

Die Schnittstellen rekonfigurierbarer Controller stellen die Verbindung zu übergeordneten Komponenten und Prozesssignalen her. Dabei wird ausgehend von der in Abschnitt 4.1 vorgestellten Struktur rekonfigurierbarer Controller in Kommunikations- und Prozessschnittstellen unterschieden. Die Anbindung eines Controllers an übergeordnete Steuer- und Regelsysteme erfolgt mit der Kommunikationsschnittstelle. Die Prozessschnittstellen realisieren dagegen die Anbindung von Prozesssignalen der zu steuernden oder regelnden Systeme an den rekonfigurierbaren Controller. Die Implementierung der Schnittstellen erfolgt innerhalb des statischen Basissystems eines Controllers und wird um externe Bestandteile für analoge und physikalische Schnittstellenebenen ergänzt. Da wesentliche Teile der Schnittstellen direkt in FPGA-Hardware umgesetzt werden, kann durch die Neukonfiguration der Zielhardware außerhalb des regulären Betriebs eine Anpassung der Schnittstellen an veränderte Einsatzbedin-

gungen und Entwicklungsstände stattfinden. Darüber hinaus können die auf dem FPGA implementierten Schnittstellenbestandteile als Hardwaretasks und rekonfigurierbare Module nach Kapitel 3 interpretiert und auf diese Weise partiell rekonfiguriert werden. Grundlage dafür ist eine umfassende Analyse der beteiligten Modulschnittstellen. Für die Steuerung und Regelung mechatronischer Systeme ist die partielle Rekonfiguration von digitalen Schnittstellenbestandteilen jedoch von geringer Bedeutung und wird hier nicht weiter betrachtet.

Kommunikationsschnittstelle

Die Kommunikationsschnittstelle eines rekonfigurierbaren Controllers ermöglicht die Kommunikation und den Datenaustausch mit übergeordneten Steuer- und Regelstrukturen. Dafür kann grundsätzlich auf eine Vielzahl von Kommunikationsstandards zurückgegriffen werden, deren digitale Bestandteile im FPGA implementiert und durch externe Komponenten für physikalische Schnittstellenebenen ergänzt werden. Dieses Vorgehen wird durch vorkonfigurierte Lösungen verschiedener FPGA-Hersteller und Drittentwickler unterstützt. Beispiele für so implementierbare Kommunikationsstandards umfassen unter anderem verschiedene Ethernetprotokolle, den Universal Serial Bus (USB), das Controller Area Network (CAN-Bus) und FlexRay aus dem Automotivebereich sowie industrielle Feldbussysteme wie Interbus und den Process Field Bus (Profibus).

Für die im Rahmen der Arbeit umgesetzten rekonfigurierbaren Controller wurde exemplarisch der Universal Serial Bus nach dem aktuellen Standard USB 2.0 [73] als Kommunikationsschnittstelle implementiert. Der Universal Serial Bus bietet eine schnelle Anbindung der rekonfigurierbaren Controller an einen USB-Host, der z.B. durch ein PC-System repräsentiert werden kann. Der Datentransfer wird dabei immer durch den USB-Host initiiert und im Fall eines PC-Systems durch entsprechende Softwareprogramme und Treiberzugriffe auf die USB-Hardware umgesetzt. Mögliche Einsatzgebiete einer USB-Schnittstelle für rekonfigurierbare Controller sind unter anderem die Entwicklung rekonfigurierbarer Funktionalitäten für konkrete Anwendungen, das Rapid Control Prototyping sowie alle Anwendungen, die von der Flexibilität und Portabilität einer USB-basierten Kommunikationslösung profitieren.

Grundsätzlich kann der digitale Teil der USB-Schnittstelle dabei direkt im FPGA oder extern umgesetzt werden. Eine Umsetzung im FPGA erfolgt auf Basis von vorkonfigurierten Funktionsblöcken, so genannten Softcores, die in ein bestehendes Design eingebunden und mit diesem auf dem FPGA implementiert werden können. Die physikalische Anbindung an den Bus ist dann jedoch zusätzlich zu den belegten FPGA-Ressourcen mit externen Bausteinen zu rea-

lisieren. Alternativ zu dieser Lösung können externe USB-Controller eingesetzt werden, die sowohl den digitalen Teil der Schnittstelle als auch die physikalische Anbindung an den Bus in einem Baustein umsetzen und so Ressourcen auf der Zielhardware einsparen. Für die im Rahmen der Arbeit erstellte USB-Schnittstelle wurde daher ein externer USB-Controller der Firma Cypress, der EZ-USB FX2 USB Mikrocontroller [74], eingesetzt. Dieser Baustein basiert auf einem eingebetteten Mikrocontroller der 8051-Familie und übernimmt neben der physikalischen Busanbindung zusätzlich Aufgaben wie Serialisierung, Prüfung und Zwischenspeicherung der Daten. Die Konfiguration des USB-Controllers erfolgt mit extern zu erstellender Firmware und wird durch den 8051-Mikrocontroller umgesetzt. So können USB-Betriebsmodi und die Zuweisung von USB-Verbindungsendpunkten (Endpoints) über die Firmware festgelegt werden.

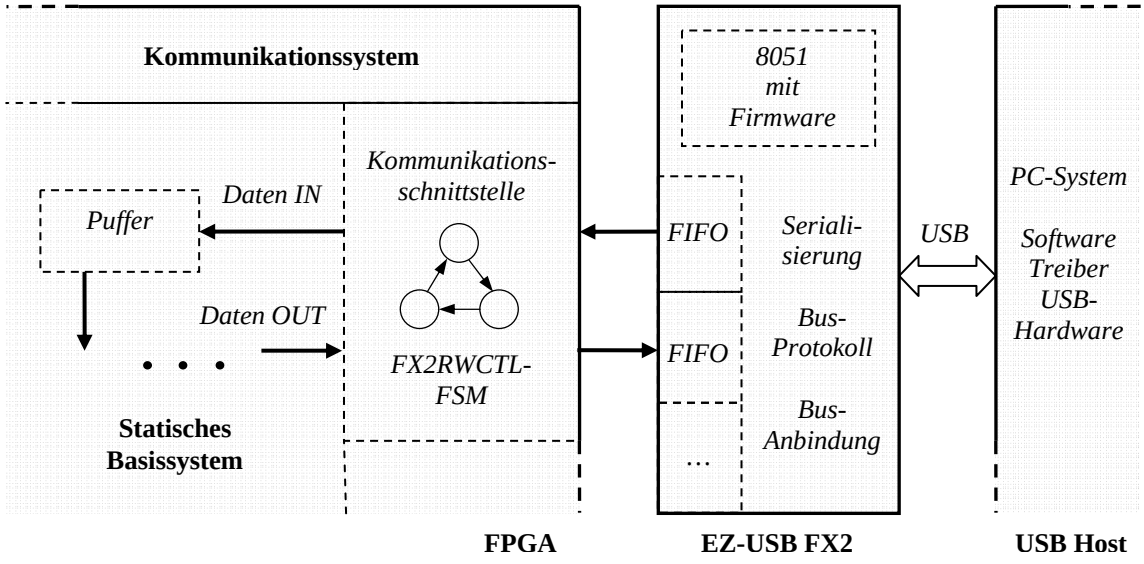


Abbildung 19: USB-Schnittstelle

Die Anbindung des USB-Controllers an das FPGA erfolgt über dedizierte FIFO-Speicher des EZ-USB FX2, die USB-Verbindungsendpunkte repräsentieren und von der Kommunikationsschnittstelle innerhalb des FPGAs kontrolliert werden. Hier setzt ein direkt in Hardware implementierter Zustandsautomat (FX2RWCTL-FSM) das Protokoll für die Schreib- und Lesesteuerung der FIFO-Speicher des USB-Controllers um und verbindet so Funktionsgruppen innerhalb des FPGAs mit dem externen Baustein. Abbildung 19 illustriert dieses Prinzip. Der Zugriff auf die FIFO-Speicher des USB-Controllers kann dabei synchron oder asynchron in Bezug auf den Systemtakt des FPGAs erfolgen. Eine synchrone Lösung ermöglicht einen höheren Datendurchsatz während eine asynchrone Lösung ressourcensparender implementiert werden kann. Durch eine Anpassung der Firmware des USB-Controllers und des FPGA-

seitigen Zustandsautomaten kann zwischen beiden Betriebsarten gewechselt werden. Der FPGA-interne datenbezogene Anschluss von Funktionsgruppen an den Zustandsautomaten erfolgt über Pufferspeicher, die als Dual-Port-RAM ausgelegt sind (vergleiche Abschnitt 4.2.2) und so die Kommunikationsschnittstelle von den Funktionsgruppen entkoppeln. Zusätzlich sind die Kommunikationsschnittstelle und die Funktionsgruppen eines rekonfigurierbaren Controllers über Steuerleitungen verbunden, die den Empfang von Daten signalisieren und das Schreiben von zu sendenden Daten in die FIFO-Speicher des USB-Controllers initiieren.

Prozessschnittstellen

Die Prozessschnittstellen rekonfigurierbarer Controller ermöglichen die Anbindung von Prozesssignalen der zu steuernden oder regelnden Systeme. Die Ausprägung und Umsetzung der Prozessschnittstellen ist daher stark von den einzelnen Anwendungen und den zu übermittelnden Signalen abhängig. Grundsätzlich kann dabei in rein digitale Schnittstellen und Schnittstellen mit Digital/Analog- oder Analog/Digital-Wandlung unterschieden werden. Wie im Fall der Kommunikationsschnittstelle können digitale Bestandteile von Prozessschnittstellen im FPGA implementiert und durch externe Komponenten für die Umsetzung der Wandlungen ergänzt werden.

Die Ansteuerung von Stellgliedern und Aktoren mechatronischer Systeme erfolgt über die Ausgabe von Stellgrößen. Für viele Anwendungen rekonfigurierbarer Controller in mechatronischen Systemen kann dabei auf die Pulsweitenmodulation als Übertragungsmethode zurückgegriffen werde. Diese Modulationsart basiert auf der Variation des Tastverhältnisses eines Signals und ist sehr einfach direkt in FPGA-Hardware umsetzbar. Ein vom FPGA ausgegebenes pulsweitenmoduliertes Signal kann über die Verwendung von externen Treiberbausteinen unmittelbar für die Ansteuerung von leistungselektronischen Stellgliedern eingesetzt werden. Eine Beispielimplementierung der Pulsweitenmodulation für die Ansteuerung elektrischer Antriebe ist in Kapitel 6 beschrieben. Weiterhin kann die Ansteuerung leistungselektronischer Stellglieder über Treiberbausteine mit alternativen Methoden wie Zweipunktreglern direkt in FPGA-Hardware realisiert werden. Eine entsprechende Umsetzung für das Laden und Entladen von Piezo-Aktoren wird ebenfalls in Kapitel 6 vorgestellt. Die Digital/Analog-Umsetzung erfolgt in beiden genannten Fällen über die Demodulation der digitalen Stellsignale durch das Tiefpassverhalten der angeschlossenen Aktoren. Eine voll digitale Ausgabe von Prozesssignalen kann über die FPGA-seitige Implementierung von typischen

Datenbusschnittstellen wie SPI oder I^2C erreicht werden und ermöglicht den Anschluss externer Wandlerbausteine sowie Stellglieder mit digitalen Schnittstellen.

Das Einlesen von digital abgebildeten Prozesssignalen kann ebenfalls mit den genannten Datenbusschnittstellen oder ähnlichen allgemeingültigen FPGA-seitig implementierten Schnittstellen erfolgen. Darüber hinaus können spezifische digitale Prozesssignale auch durch dedizierte Schnittstellen auf dem FPGA aufbereitet werden. Ein Beispiel für diese Art der Anbindung ist die Auswertung der binären Ausgangssignale von Inkrementalgebern für die Lage- und Drehzahlerfassung elektrischer Antriebe. Eine einfache Beispielimplementierung dieses Prinzips wird in Kapitel 6 beschrieben. Ein generelles Problem von FPGAs im Bereich der Mechatronik stellen jedoch die fehlenden analogen Eingänge dar. Üblicherweise wird das FPGA daher zur Anbindung analoger Prozesssignale um einen oder mehrere externe Analog/Digital-Umsetzer (ADUs) ergänzt, die analoge Signale wandeln und in digitaler Form an die Datenbusschnittstellen eines FPGA weiterleiten. Vor dem Hintergrund dieser Problematik wurde der Einsatz von Delta-Sigma-Analog/Digital-Umsetzern für FPGA-basierte rekonfigurierbare Controller untersucht. Diese können zu einem wesentlichen Teil direkt in FPGA-Hardware realisiert werden und benötigen nur geringe externe und I/O-Ressourcen. Die grundlegende Struktur eines Delta-Sigma-ADUs ist in Abbildung 20 dargestellt.

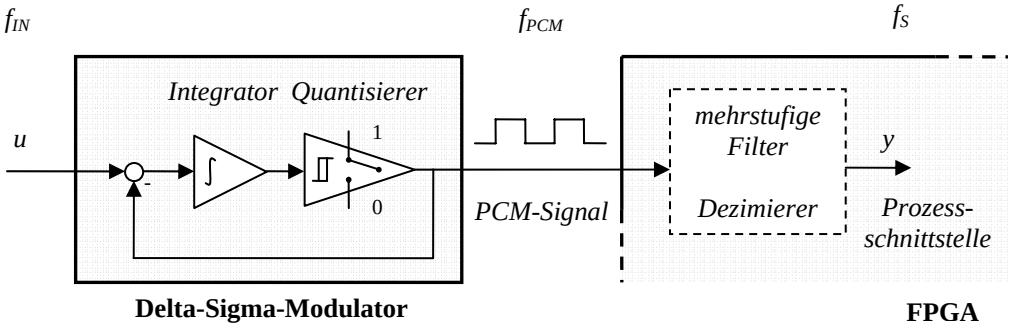


Abbildung 20: Struktur eines Delta-Sigma-Analog/Digital-Wandlers

Delta-Sigma-ADUs basieren auf der Wandlung eines analogen Eingangssignals in ein pulscodemoduliertes binäres Signal (PCM-Signal), dessen arithmetischer Mittelwert dem des analogen Signals entspricht. Die Wandlung wird durch einen relativ einfachen FPGA-externen Baustein, den Delta-Sigma-Modulator, umgesetzt. Das PCM-Signal wird daraufhin an das FPGA weitergeleitet und dort weiterverarbeitet. Die Abtastfrequenz des PCM-Signals f_{PCM} liegt deutlich über der Abtastfrequenz der auf dem FPGA implementierten Signalverarbeitung f_S eines rekonfigurierbaren Controllers. Innerhalb des FPGAs erfolgt die Bildung eines reprä-

sentativen Signals durch eine mehrstufige digitale Filterung und Dezimierung der Abtastfrequenz des PCM-Signals. Die Auflösung des Ausgangssignals ist dabei logarithmisch von dem Verhältnis zwischen Abtastfrequenz des PCM-Signals und Abtastfrequenz des Ausgangssignals (*Over Sampling Ratio*, OSR) abhängig. Daher kann die Auflösung durch eine Anpassung der Filter- und Dezimationsalgorithmen variiert werden ohne Änderungen am FPGA-externen Teil des Wandlers vorzunehmen. Dieser großen Flexibilität FPGA-basierter Delta-Sigma-ADUs steht jedoch ein hoher Ressourcenbedarf und Implementierungsaufwand der mehrstufigen Filter- und Dezimierungsalgorithmen gegenüber. Der Einsatz von Delta-Sigma-ADUs in rekonfigurierbaren Controllern für mechatronische Systeme ist aus diesem Grund nicht für jeden Anwendungsbereich uneingeschränkt möglich. Die diesem Teilabschnitt zugrunde liegende Umsetzung eines FPGA-basierten Delta-Sigma-ADUs ist in [75] beschrieben. Für Grundlagen und weiterführende Informationen zu Delta-Sigma-Wandlern sei auf [76] verwiesen.

4.3 Spezifikationsframework

Der folgende Abschnitt der Arbeit stellt ein Spezifikationsframework vor, das die Entwurfsschritte für FPGA-basierte Komponenten und Funktionsgruppen rekonfigurierbarer Controller verallgemeinert und das V-Modell nach Abschnitt 3.1.1 um praktische Aspekte ergänzt. Das Spezifikationsframework basiert auf Funktionsbibliotheken und einer Werkzeugkette für den Entwurf, die funktionale Verifikation und die initiale Hardwareumsetzung statischer und rekonfigurierbarer Controllerbestandteile. Abbildung 21 zeigt die dem Spezifikationsframework zugrunde liegenden Entwurfsschritte und die zugehörige Werkzeugkette. Die Werkzeugkette umfasst dabei zu großen Teilen allgemeingültige Softwarewerkzeuge der FPGA-basierten digitalen Hardwareentwicklung. Ausgehend von einzelnen Komponenten und Funktionsgruppen rekonfigurierbarer Controller erfasst das Spezifikationsframework auch die Entwurfsschritte des Gesamtsystems.

Ausgangspunkt der Werkzeugkette ist die abstrakte Spezifikation von Funktionen und Komponenten sowie der logischen und technischen Controllerstruktur. Dies erfolgt mit den in Abschnitt 3.1.2 vorgestellten generischen Spezifikationswerkzeugen wie Blockdiagrammen und Zustandsautomaten. Die abstrakte Spezifikation wird im folgenden Entwurfsschritt mit grafischen Softwarewerkzeugen in ein konkretes Design überführt. Dabei kann auf Funktionsbibliotheken zurückgegriffen werden, die im Rahmen der Arbeit erstellt wurden. Die Bibliotheken umfassen:

- bitserielle und bitparallele Steuer- und Regelalgorithmen für Hardwaretasks
- ein konfigurierbares Kommunikationssystem nach Abschnitt 4.2.1 mit den FSM-basierten Teilnehmeranschlüssen und Task-FSMs
- das Rekonfigurations- und Speichermanagement nach 4.2.2 mit der Zustandssicherung und -wiederherstellung nach 4.2.3
- sowie digitale Prozessschnittstellen und die FPGA-seitige Implementierung der USB-Schnittstelle nach 4.2.4.

Ergänzt werden die Bibliotheken durch vorkonfigurierte Modulschnittstellen für rekonfigurierbare Module und deren Hardwaretasks. Die Funktionsbibliotheken wurden vorwiegend mit dem HDL Designer von Mentor Graphics erstellt und können mit diesem für das Komponentendesign auf grafischer Ebene eingesetzt werden. Zusätzlich wurden für einzelne Anwendungen Funktionsbibliotheken in Matlab/Simulink angelegt. Der grafische Entwurf von Komponenten und Funktionsgruppen mit Softwarewerkzeugen wie dem HDL Designer oder Matlab/Simulink ermöglicht in vielen Fällen eine direkte Abbildung der generischen Spezifikation und vereinfacht so den Entwurfsprozess. Darüber hinaus kann auch direkt auf Hardwarebeschreibungssprachen wie VHDL oder Verilog zurückgegriffen werden.

Die grafische Beschreibung von Komponenten und Funktionsgruppen wird im folgenden Entwurfsschritt mittels einer Codegenerierung innerhalb der genannten Softwarewerkzeuge in eine Hardwarebeschreibungssprache überführt. Gegenüber der direkten Umsetzung von Komponenten und Funktionsgruppen mit Hardwarebeschreibungssprachen abstrahiert diese Vorgehensweise den Zugang zu rekonfigurierbarer Hardware und vereinfacht den Entwurfsvorgang deutlich. Auf der Basis des generierten HDL-Quelltextes können die erstellten Funktionsbibliotheken für rekonfigurierbare Controller auch außerhalb der Entwurfswerkzeuge genutzt werden. Die funktionale Verifikation der Komponenten und Funktionsgruppen erfolgt durch eine Verhaltenssimulation des generierten HDL-Quelltextes mit einem HDL-Simulator wie Mentor Graphics' ModelSim unter Verwendung von zugeschnittenen Testszenarien [77]. ModelSim kann dabei in die grafische Oberfläche des HDL Designers eingebunden und von dort gesteuert werden. Im Fehlerfall müssen einzelne oder sämtliche bereits durchlaufene Entwurfsschritte modifiziert und wiederholt werden. Die Verhaltenssimulation repräsentiert damit einen Testfall aus dem V-Modell nach Abschnitt 3.1.1. Anzumerken ist, dass bei alleiniger Verwendung von Matlab/Simulink für das Komponentendesign eine initiale Verhaltens-

simulation bereits zum Entwurfszeitpunkt innerhalb der Entwicklungsumgebung ohne eine Codegenerierung möglich ist.

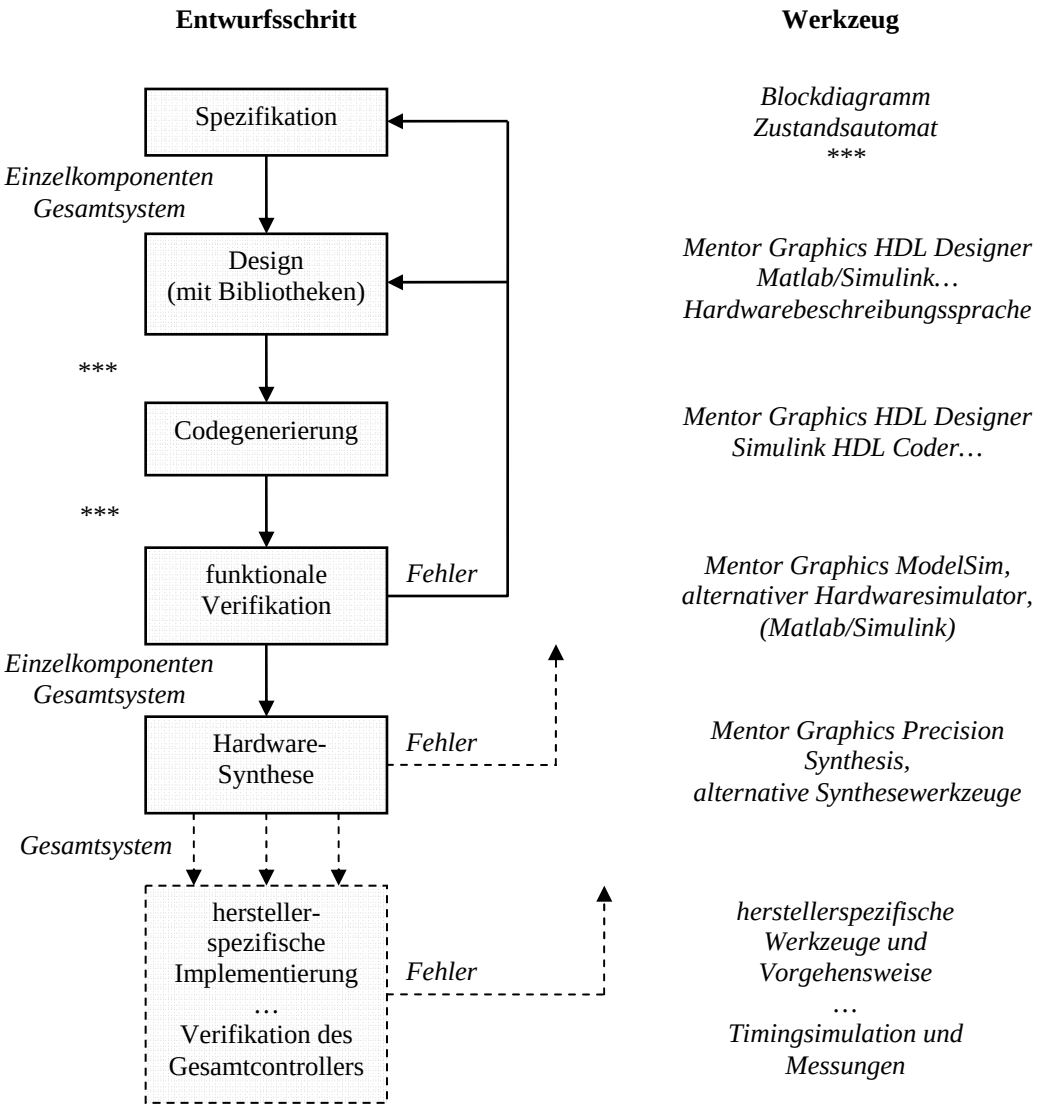


Abbildung 21: Entwurfsschritte und Softwarewerkzeuge

Nach der funktionalen Verifikation des generierten HDL-Quelltextes folgt mit der Hardwaresynthese der erste auf die Hardwareumsetzung der Komponenten bezogene Entwurfsschritt. Die Hardwaresynthese überführt den HDL-Quelltext einer Komponente in eine Beschreibungsform auf Registerebene und nimmt erste Optimierungen sowie zielhardwarespezifische Anpassungen vor. Das Ergebnis der Synthese ist eine Netzliste, die eine erste Abschätzung des Ressourcenbedarfs der synthetisierten Komponenten und Funktionsgruppen erlaubt. Im Fehlerfall ist auch hier eine weitere Iteration der Entwurfsschritte durchzuführen. Werkzeuge für die Hardwaresynthese sind unter anderem in den Entwicklungsumgebungen einzelner FPGA-Hersteller enthalten. In dieser Arbeit wurde das herstellerunabhängige Synthesewerk-

zeug Precision Synthesis von Mentor Graphics eingesetzt, da es wie ModelSim in die grafische Oberfläche des HDL Designers eingebunden werden kann [78]. Diese Vorgehensweise unterstützt so die Durchgängigkeit des Entwurfsprozesses.

An die Hardwaresynthese der Komponenten und Funktionsgruppen rekonfigurierbarer Controller schließt sich die eigentliche Implementierung des Systems mit herstellerspezifischen Werkzeugen an. Die grundlegenden Implementierungsschritte, synthesebezogenen Voraussetzungen und hardwarespezifischen Randbedingungen sowie die Verifikation des Gesamtsystems werden im folgenden Kapitel 5 der Arbeit beschrieben.

5 Implementierung

Das fünfte Kapitel der Arbeit behandelt die hardwarebezogene Implementierung rekonfigurierbarer Controller für mechatronische Systeme. Ausgehend von den spezifischen Eigenschaften der gewählten Zielhardware erfolgt eine Analyse der Hardwareressourcen für die Umsetzung statischer und rekonfigurierbarer Funktionalitäten. Der Einsatz bitserieller Signalverarbeitung sowie die Skalierung und Portierbarkeit der Controller werden in diesem Kontext ebenfalls untersucht. Weiterer Schwerpunkt des Kapitels ist eine hardwarebasierte Rekonfigurationslösung, die mit sehr geringem Hardwareaufwand flexibel implementiert werden kann. Der echtzeitfähige Rekonfigurationsmechanismus wurde für die interne Konfigurationsschnittstelle von Xilinx FPGAs ausgelegt und ermöglicht aufgrund geringer Rekonfigurationszeiten schnelle Ladevorgänge für Hardwaretasks. Auf Basis des im vorhergehenden Kapitel vorgestellten Rekonfigurations- und Speichermanagements realisiert die Funktionsgruppe darüber hinaus die Selbstrekonfiguration der Controller. Weiterhin werden in diesem Kapitel die herstellerspezifischen Implementierungsschritte für partiell rekonfigurierbare Systeme auf Xilinx FPGAs beschrieben und auf rekonfigurierbare Controller für mechatronische Systeme angewendet. Das Kapitel schließt mit der Beschreibung hardwarebezogener Verifikationsmethoden für rekonfigurierbare Controller auf Grundlage der Implementierungsergebnisse. Die vorgestellten Methoden umfassen dabei Timingsimulationen sowie Messungen auf der Zielplattform.

Der erste Abschnitt des Kapitels beschreibt die Zielhardware und Hardwareressourcen für die Implementierung rekonfigurierbarer Controller. Im zweiten Abschnitt wird die hier entwickelte echtzeitfähige Rekonfigurationslösung vorgestellt. Daraufhin werden im dritten Abschnitt die Vorgehensweise bei der Hardwareimplementierung und die Anbindung an das Spezifikationsframework erläutert. Der vierte und letzte Abschnitt des Kapitels geht auf die Verifikation der implementierten Controller ein.

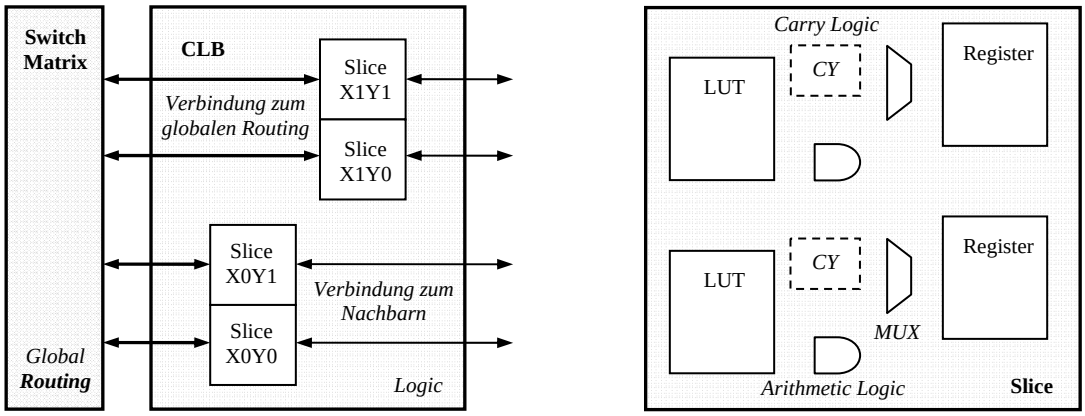
5.1 Zielhardware und Ressourcen

Die Implementierung der rekonfigurierbaren Controller für mechatronische Systeme und zugehöriger Funktionsgruppen erfolgt mit partiell rekonfigurierbaren FPGAs der Firma Xilinx. Alle Funktionalitäten werden dabei direkt in FPGA-Hardware umgesetzt, ohne auf eingebettete Mikroprozessoren zurückzugreifen. Auf diese Weise kann der Ressourcenverbrauch auf Systemebene gering gehalten werden. Darüber hinaus unterstützt die direkte Hardwareimplementierung der Algorithmen die Echtzeitfähigkeiten der Controller.

5.1.1 Rekonfigurierbare Controller auf Xilinx FPGAs

Die SRAM-basierten FPGAs der Firma Xilinx können in Bausteine für den Hochleistungsbereich und für einen kostensensitiven Einsatz unterschieden werden. Die Bausteine für den Hochleistungsbereich werden durch die Xilinx Virtex-FPGAs repräsentiert. Aktuelle Bausteine innerhalb der Virtex-Familie sind Xilinx Virtex-II, Virtex-4 und Virtex-5 FPGAs, wobei der letztere die jüngste Generation darstellt. Für den kostensensitiven Einsatz sind Xilinx Spartan FPGAs geeignet, mit dem Spartan-3 als jüngster Generation. Die Hardwareressourcen der Xilinx Virtex-FPGAs sind dabei allgemein größer als die der Spartan-FPGAs. Als Zielhardware für die in dieser Arbeit entworfenen rekonfigurierbaren Controller wurden konkret Virtex-II und Virtex-4 FPGAs eingesetzt ([79] [80]). Die folgenden Ausführungen beziehen sich daher auf die sehr ähnliche Hardwarearchitektur dieser Bausteine, sind aber auch auf die ebenfalls verwandte Architektur der Spartan-3 FPGAs anwendbar [81].

Die grundlegende Hardwareressource der genannten Xilinx FPGA-Typen sind die matrixartig angeordneten *Configurable Logic Blocks* (CLBs), die je vier als *Slices* bezeichnete Subeinheiten umfassen. Jede der Subeinheiten beinhaltet zwei kombinatorische Funktionsgeneratoren und zwei synchrone Speicher. Die Funktionsgeneratoren entsprechen dabei Look-Up-Tables (LUTs) mit vier Eingängen. Die Speicher können als flanken- oder zustandsgesteuerte Flipflops konfiguriert werden. Eine Slice repräsentiert damit einen in Abschnitt 1.2.2 beschriebenen elementaren programmierbaren Logikblock der Architektur. Zusätzlich enthalten die Slices verschiedene Multiplexer und elementare Logikgatter. Abbildung 22 illustriert den Aufbau der grundlegenden Hardwareressourcen. Neben den CLBs werden für die Implementierung der rekonfigurierbaren Controller weitere Hardwarekomponenten wie digitale Clockmanager (DCMs) und eingebettete RAM-Blöcke sowie für einzelne Anwendungsfälle dedizierte Multiplizierer eingesetzt. Tabelle 2 zeigt beispielhaft die verfügbaren und hier relevanten elementaren Hardwareressourcen zweier in dieser Arbeit eingesetzter FPGA-Bausteine.



a) CLB b) CLB Slice

Abbildung 22: Struktur von CLBs und CLB Slices eines Xilinx Virtex-II FPGA nach [79]

Die Verbindung der elementaren Hardwarekomponenten untereinander erfolgt mit dem programmierbaren Verbindungsnetzwerk (Routingressourcen) der FPGAs, während die Verbindung mit der Außenwelt mit programmierbaren I/O-Blöcken umgesetzt wird. Für eine detaillierte Beschreibung der Architektur der genannten FPGA-Typen sei auf die oben angeführte Literatur verwiesen. Details zu einzelnen elementaren Hardwarekomponenten und deren Einsatz für die Implementierung rekonfigurierbarer Controller werden auch in den folgenden Unterabschnitten aufgeführt.

FPGA	CLB-Matrix	CLB Slices	Block RAM	DCMs	MULTs
Virtex-II XC2V1000	40 x 32 = 1280	1280 x 4 = 5120	40 x 18 Kbit = 720 Kbit	8	40 x 18 bit * 18 bit
Virtex-4 XC4VLX60	128 x 52 = 6656	6656 x 4 = 26624	160 x 18 Kbit = 2880 Kbit	8	64 x 18 bit * 18 bit

Tabelle 2: Beispiele verfügbarer Hardwareressourcen in Xilinx Virtex-II und Virtex-4 FPGAs

Statisches Basissystem und globale Komponenten

Die statischen Funktionsgruppen der rekonfigurierbaren Controller werden im Basissystem implementiert und beanspruchen elementare sowie dedizierte Hardwarekomponenten. Das statische Basissystem umfasst dabei einen Großteil der verfügbaren Hardwareressourcen der Zielplattform. Im Basissystem werden die elementaren CLBs des FPGAs für die grundlegende Umsetzung von Registern, Logikfunktionen und Zustandsautomaten eingesetzt. Die dafür beanspruchten Hardwareressourcen sind stark vom konkreten Anwendungsfall und dessen spezifischen Anforderungen abhängig. Entsprechend detaillierte Angaben finden sich daher in den Implementierungsergebnissen der Anwendungen nach Kapitel 6.

Neben den CLBs kommen für einzelne Funktionsgruppen dedizierte Funktionsblöcke zum Einsatz. So werden eingebettete RAM-Blöcke der Xilinx FPGAs für die Implementierung von kleineren Speichern wie im Rekonfigurations- und Speichermanagement oder in den Funktionsgruppen der Zustandssicherung und -wiederherstellung verwendet. Die eingebetteten RAM-Blöcke der Xilinx FPGAs unterstützen sowohl den Single-Port- als auch den Dual-Port-Betrieb. Der Hauptspeicher des Systems für die Speicherung der Konfigurationsdaten der Hardwaretasks kann ebenfalls mit eingebetteten RAM-Blöcken implementiert werden, wenn größere FPGAs wie die in Tabelle 2 aufgeführten Bausteine eingesetzt werden. Einschränkend wirkt sich die Verteilung der einzelnen RAM-Blöcke auf dem FPGAs aus, die zusätzliche Ressourcen für die Adressdekodierung und Datenzuweisung erfordert. Ein Dual-Port-Speicher mit einer Speicherkapazität von 256 KB und 32 Bit breiten Datenschnittstellen kann so auf dem Virtex-4 XC4VLX60 Baustein mit 116 RAM-Blöcken (je 18 Kbit) und zusätzlichen 435 CLB Slices (20 DFFs, 830 LUTs) umgesetzt werden.

Weiterhin werden für die Hardwareumsetzung der Funktionalitäten des Basissystems globale dedizierte Hardwarekomponenten verwendet. Diese werden im Rahmen der herstellerspezifischen Implementierungsschritte jedoch nicht dem Basissystem sondern einer übergeordneten Ebene, dem so genannten Top-Level, zugeordnet (vergleiche Abschnitt 5.3.1). Betroffen ist globale Logik wie I/O-Hardware und digitale Clockmanager.

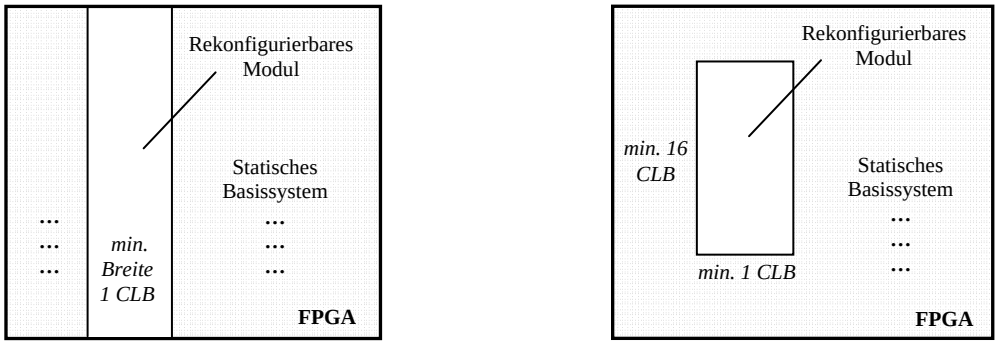
Ein digitaler Clockmanager wird unter anderem für die Taktung des Rekonfigurationsmechanismus und damit für die Ansteuerung der internen Konfigurationsschnittstelle der Zielplattform verwendet. Der digitale Clockmanager erlaubt dabei eine weit reichende Modifikation der Taktfrequenz. Ausgenutzt wird hier eine deutliche Erhöhung des Taktes, um kürzere Ladezeiten für Hardwaretasks zu erzielen. Eine nähere Beschreibung dieser Zusammenhänge findet sich in Abschnitt 5.2.3. Ein zusätzlicher Einsatz von digitalen Clockmanagern kann für die synchrone Kommunikation zwischen der Kommunikationsschnittstelle auf dem FPGA und dem externen Cypress EZ-USB FX2 Baustein erfolgen.

Ebenso zählt auch die interne Konfigurationsschnittstelle ICAP (*Internal Configuration Access Port*) der Xilinx FPGAs zur globalen Logik und ist daher für die Hardwareimplementierung einer übergeordneten Ebene zuzuordnen. Für eine detaillierte Beschreibung der Schnittstelle und ihrer Einbettung in das statische Basissystem durch den Rekonfigurationsmechanismus sei auf Abschnitt 5.2 verwiesen.

Rekonfigurierbare Module und Hardwaretasks

Die rekonfigurierbaren Module und Hardwaretasks der Controller basieren primär auf direkt in den CLBs der Ziel-FPGAs implementierten Steuer- und Regelfunktionen. Darüber hinaus können eingebettete Hardwarekomponenten wie einzelne RAM-Blöcke und Multiplizierer lokal in den Hardwaretasks genutzt werden. Die in rekonfigurierbaren Modulen verfügbaren Hardwareressourcen sind durch die Form und Größe der Module begrenzt.

Die Größe eines rekonfigurierbaren Moduls ist direkt abhängig vom Ressourcenbedarf des größten aufzunehmenden Hardwaretasks, der durch ein iteratives Durchlaufen des Vorgehensmodells nach Abschnitt 3.1.1 ermittelt wird. Die Form der Module ist dagegen von der gewählten Zielhardware abhängig. Rekonfigurierbare Bereiche auf Xilinx Virtex-II und Spartan-3 FPGAs müssen so immer die volle Höhe des Bausteins einnehmen und mindestens einen CLB breit sein. Diese Einschränkung gilt für die neueren Xilinx Virtex-4 FPGAs nicht mehr, hier beträgt die architekturbedingte Mindesthöhe 16 CLBs bei einer Mindestbreite von einem CLB [49]. Form und Größe eines rekonfigurierbaren Moduls sind für beide Architekturen fest und können nur durch eine Neuimplementierung des zugrunde liegenden rekonfigurierbaren Controllers geändert werden. Abbildung 23 illustriert die Zusammenhänge.



a) Virtex-II / Spartan-3 b) Virtex-4

Abbildung 23: Form und Größe rekonfigurierbarer Module auf Xilinx FPGAs

Die rekonfigurierbaren Module verfügen im Gegensatz zum statischen Basissystem der Controller über keine direkte Verbindung mit FPGA-externen Systemen. Daher erfolgt der Transfer von prozessrelevanten Größen wie Soll-, Ist- und Stellwerten über Schnittstellen zwischen Modul und Basissystem. Die Modulschnittstellen müssen während des Entwurfsprozesses passend für alle Hardwaretasks des betreffenden Moduls ausgelegt werden, da eine Änderung der Schnittstellenkonfiguration zur Laufzeit des Controllers nicht unterstützt wird. Die von den Schnittstellen übertragenen Signale werden abhängig von den Hardwaretasks eines Moduls ermittelt und basieren auf einer Vereinigung der Schnittstellensignale aller Hardware-

tasks eines Moduls. Dabei können für einzelne Hardwaretasks Signale der Modulschnittstelle ungenutzt verbleiben. Die Hardwareumsetzung der Modulschnittstellen und die Signalübertragung zwischen rekonfigurierbaren Modulen und dem statischen Basisdesign eines Controllers wird im folgenden Unterabschnitt beschrieben.

Bus-Makros

Als Bus-Makros bezeichnet Xilinx nicht von der partiellen Rekonfiguration beeinflussbare Verbindungen zwischen rekonfigurierbaren und statischen Bereichen eines FPGAs [51]. Alle Signale, die in einen rekonfigurierbaren Bereich hinein oder heraus führen, müssen mit Bus-Makros umgesetzt werden. Ausnahmen sind globale Taktsignale, die über dedizierte Verbindungsressourcen verfügen. Signale, die einen rekonfigurierbaren Bereich durchqueren und über keine Verbindung zu der Signalverarbeitung innerhalb des Bereiches verfügen, benötigen für den aktuellen Early Access Partial Reconfiguration Design Flow (EAPR Design Flow) von Xilinx keine Bus-Makros.

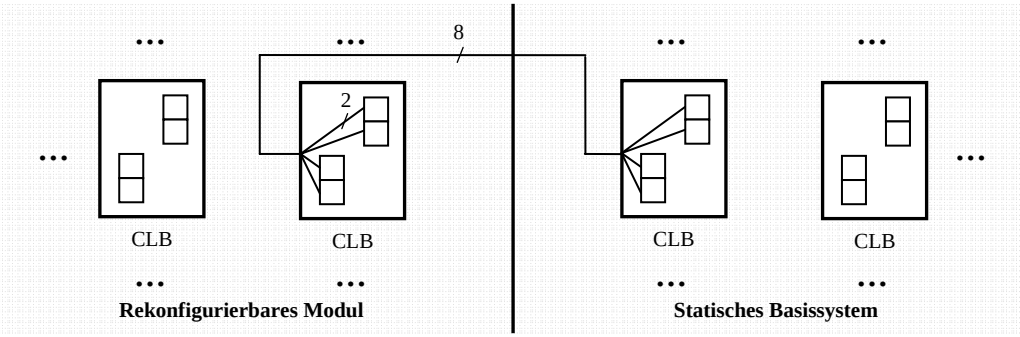


Abbildung 24: Struktur LUT-basierter Bus-Makros (nach [4])

Die Hardwaregrundlage für Bus-Makros sind elementare Hardwarekomponenten der CLBs, die den Anfangs- und Endpunkt der Signalübertragung bilden. Für ältere Versionen der Bus-Makros wurden Tri-State-Puffer eingesetzt, neuere Makros basieren dagegen auf den Look-Up-Tables der elementaren CLBs. Jeder der acht Look-Up-Tables eines CLBs dient dabei als Anfangs- oder Endpunkt einer Einzelleitung, so dass ein Bus-Makro mit einer Übertragungsbreite von 8 Bit zwei CLBs verbraucht. Die Hardwarekomponenten und Leitungen der Makros werden relativ auf der Zielhardware platziert und als Einheit fest implementiert. Die Übertragungsrichtung der Makros ist aufgrund der festen Hardwareimplementierung fixiert. Abbildung 24 illustriert das Prinzip LUT-basierter Bus-Makros.

Xilinx stellt im Rahmen des EAPR Design Flows vorkonfigurierte Bus-Makros mit einer Übertragungsbreite von 8 Bit bereit. Diese sind an die spezifischen FPGA-Baureihen angepasst und bieten optional eine synchrone Übertragung und einen Enable-Eingang, um unerwünschte Zustände während einer Rekonfiguration zu unterbinden. Weiterhin liegen Makros verschiedener Länge vor. Während für Xilinx Virtex-II FPGAs architekturbedingt nur horizontale Makros existieren, sind für Virtex-4 Bausteine auch vertikale Übertragungsrichtungen vorhanden. Zusätzlich besteht die Möglichkeit, über den Einsatz der Implementierungswerkzeuge von Xilinx selbst definierte Makros zu erstellen oder die bereitgestellten Makros zu modifizieren. Das in Abschnitt 4.2.1 beschriebene Kommunikationssystem ist ein Beispiel für eine alternative Makroimplementierung.

5.1.2 Rekonfigurierbare Controller und bitserielle Signalverarbeitung

Die in dieser Arbeit eingesetzten bitseriellen Algorithmen stellen eine Variante der in Abschnitt 2.4 beschriebenen bitseriellen Signalverarbeitung nach [65] dar. So wird den im Zeitmultiplex auf einer Leitung übertragenen Operanden ein Startbit vorangestellt und gleichzeitig der Synchronisationsbus nach Abbildung 6, Abschnitt 2.4, zusätzlich zur Verarbeitungswortbreite um ein Bit erweitert. Damit können die Operanden mit einer festen Abtastzeit übertragen und synchronisiert werden. Die Implementierung der bitseriellen Funktionen erfolgt dabei ausschließlich in den elementaren CLBs der Zielplattform. Unabhängig von der Verarbeitungswortbreite der Algorithmen ist für die Übertragung der Operanden nur eine Leitung der Routingressourcen des FPGAs erforderlich. Auf diese Weise können Steuer- und Regelfunktionen in Hardwaretasks sehr effizient umgesetzt werden.

Neben der direkten Implementierung von Steuer- und Regelfunktionen wird die bitserielle Signalübertragung vorrangig für eine Ressourcen und Aufwand minimierende Umsetzung der Schnittstellen rekonfigurierbarer Module eingesetzt. Diese Schnittstellen realisieren die Verbindung der in den Modulen geladenen Hardwaretasks mit dem statischen Basissystem und übermitteln vor allem Prozesssignale wie Soll-, Ist- und Stellwerten. Auf der Grundlage bitserieller Übertragungsmethoden können die Schnittstellen effizient mit nur einer Leitung pro Signal umgesetzt werden, so dass der Bedarf an den im vorigen Abschnitt beschriebenen Bus-Makros minimiert wird. Bitseriell ausgelegte Modulschnittstellen sind über die Einsparung von Routingressourcen hinaus unabhängig von der Verarbeitungswortbreite der eigentlichen Signalverarbeitung. Eine Änderung der Verarbeitungswortbreite erfordert daher keine Anpassung der Schnittstelle des Moduls.

Der Einsatz bitserieller Übertragungsmethoden bedingt jedoch zusätzliche Komponenten für Serialisierung, Deserialierung und Synchronisation der Signalverarbeitung im statischen Basisdesign und den Hardwaretasks. Die Serialisierung und Deserialisierung der zu übertragenden Signale wird mit auf Schieberegistern basierenden Parallel-Seriell- (*p2s*) und Seriell-Parallelwandlern (*s2p*) umgesetzt. Diese werden direkt in den CLBs der Zielhardware implementiert. Die Abtastfrequenz f_S der bitseriellen Signalübertragung und folgenden Verarbeitung wird dabei über ein bereitzustellendes Signal (*op_new*) bei der Parallel-Seriellwandlung vorgegeben. Sie kann frei gewählt werden und ist nur durch das Verhältnis von Systemtakt f_C zu Verarbeitungswortbreite *n* begrenzt. Für die hier eingesetzte Variante der bitseriellen Signalverarbeitung nach [65] gilt:

$$f_{S\max} = \frac{f_C}{n+1} \qquad \text{Gleichung 5-1}$$

Auf Basis der festen Abtastzeit der bitseriellen Signalverarbeitung können bei hinreichend kleinen Rekonfigurationszeiten die Ladevorgänge für Hardwaretasks in den so vorhandenen Zeitfenstern ohne Signalübertragung umgesetzt werden. Ein Vorabladen von Hardwaretasks nach Abschnitt 3.3.3 ist dann nicht mehr notwendig. Die Synchronisation der bitseriellen Signalverarbeitung erfolgt mit dem erwähnten dedizierten Synchronisationsbus (*chk*, check), der im Basisdesign generiert und an die rekonfigurierbaren Module weitergeleitet wird. Die Übermittlung des Synchronisationsbusses an die Module kann mit einem Einzelsignal umgesetzt werden, aus dem in den Hardwaretasks ein lokaler Synchronisationsbus generiert wird.

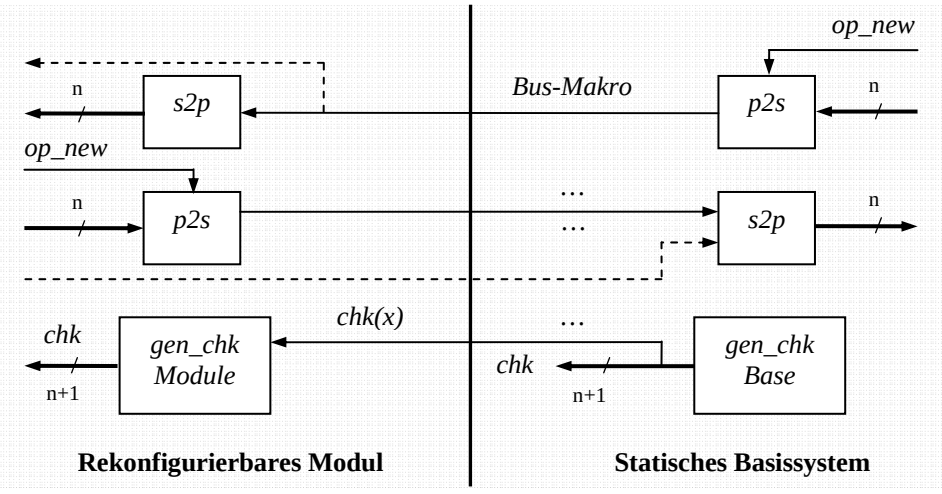


Abbildung 25: Bitserielle Signalübertragung in rekonfigurierbaren Controllern

Abbildung 25 illustriert die aufgeführten Zusammenhänge von Serialisierung, Deserialierung und Synchronisation bitserieller Signalübertragung und -verarbeitung in rekonfigurierbaren

Controllern. Ein Beispiel für den Einsatz bitserieller Algorithmen in rekonfigurierbaren Controllern für mechatronische Systeme wird mit einem rekonfigurierbaren Antriebscontroller in Abschnitt 6.1 der Arbeit gegeben. Grundsätzlich kann bei den hier vorgestellten rekonfigurierbaren Controllern jedoch anwendungsspezifisch und frei zwischen bitseriellen und bitparallelen Signalverarbeitungs- und Übertragungsmethoden gewählt werden. Es werden reine bitserielle oder bitparallele Systeme sowie Mischformen unterstützt.

5.1.3 Skalierung und Portierbarkeit

Die in dieser Arbeit entwickelten rekonfigurierbaren Controller unterstützen eine einfache Anpassung ihrer Komponenten an neue Systemfunktionen. Auf Basis der modularen Systemarchitektur und der Funktionsbibliotheken nach Abschnitt 4.3 kann die Erweiterung oder Modifikation der Steuer- und Regelfunktionen sowie der Controllerschnittstellen erfolgen. Darüber hinaus ermöglicht die Hardwareimplementierung der Steuer- und Regelfunktionen der Controller eine freie Einstellung der Abtastzeiten der Signalverarbeitung, die nur durch die Eigenschaften der Zielhardware und der Prozessschnittstellen begrenzt sind. Im Falle einer bitseriellen Umsetzung von Steuer- und Regelfunktionen ist hier zusätzlich der oben genannte Zusammenhang zwischen Systemtakt und Verarbeitungswortbreite anzuführen. Da heutige FPGAs Systemfrequenzen bis zu mehreren 100 MHz unterstützen, stellt dies jedoch keine Einschränkung für mechatronische Systeme dar.

Im Gegensatz zu einer Umsetzung mit Mikrocontrollern ermöglicht die direkte Hardwareimplementierung von Steuer- und Regelfunktionen eine freie Anpassung der Verarbeitungswortbreite. Der Ressourcenverbrauch für eine bitparallele Hardwareimplementierung ist dabei deutlich von der gewählten Wortbreite abhängig. Im Falle bitserieller Methoden bedeutet eine Erhöhung der Verarbeitungswortbreite dagegen eine Vergrößerung der Bearbeitungs- und Übertragungszeit sowie eine moderate Erhöhung des Ressourcenbedarfs an elementaren Logikblöcken aufgrund längerer Registerketten in Operatoren mit Speicherfunktion. Tabelle 3 zeigt die Skalierung und den synthesebasierten Ressourcenverbrauch bitserieller Algorithmen am Beispiel eines Proportional-Integral-Gliedes (PI-Regler) für verschiedene Verarbeitungswortbreiten inklusive Startbit. Die Reglerparameter wurden hier exemplarisch mit 5 Bit aufgelöst. Die eingesetzten bitseriellen Verstärker (Multiplikation mit konstantem Faktor) realisieren auch nicht ganzzahlige Verstärkungsfaktoren und skalieren das Ergebnis automatisch [65]. Routingressourcen für die bitserielle Signalübertragung sind unabhängig von Änderungen der Verarbeitungswortbreite.

Die rekonfigurierbaren Controller unterstützen weiterhin eine einfache Portierung auf verschiedenen FPGA-Baureihen von Xilinx. Grundlage hierfür sind die hohe Abstraktionsebene der Entwurfsmethodik, die HDL-basierten Funktionsbibliotheken und die direkte Hardwareimplementierung der Controller sowie ihrer Komponenten ohne Einsatz dedizierter Hardwareelemente. Eine Ausnahme stellen interne Speicherblöcke, dedizierte Multiplizierer, digitale Clockmanager und die interne Konfigurationschnittstelle ICAP dar. Sowohl interne Speicherblöcke als auch dedizierte Multiplizierer und digitale Clockmanager sind jedoch auf allen Xilinx FPGAs der aktuellen Spartan- und Virtex-Baureihen vorhanden. Die interne Konfigurationsschnittstelle ICAP ist ebenfalls in allen aktuellen Bausteinen der Virtex-Baureihe präsent. Dagegen verfügen nur spezifische Bausteine der Xilinx Spartan-FPGAs (Spartan-3A) über die ICAP-Schnittstelle. Eine Portierung der rekonfigurierbaren Controller auf Xilinx FPGAs ohne interne Konfigurationsschnittstelle kann ohne weiteres unter Einbeziehung externer Konfigurationsschnittstellen vorgenommen werden. Abschnitt 5.2 geht näher auf die Konfigurationsschnittstellen von Xilinx FPGAs ein.

Verarbeitungs-wortbreite	**CLB Slices**	**DFFs / Latches**	**LUTs**
8 Bit	20	38	39
16 Bit	23	46	39
32 Bit	31	62	39

Tabelle 3: Ressourcenverbrauch und Skalierung eines bitseriellen Proportional-Integral-Gliedes am Beispiel eines Xilinx Virtex-II FPGAs (XC2V1000)

Eine Portierung der rekonfigurierbaren Controller auf FPGAs anderer Hersteller kann gleichfalls auf der Grundlage der HDL-basierten Funktionsbibliotheken und einer direkten Hardwareumsetzung erfolgen, bedingt aber eine Unterstützung der partiellen Rekonfiguration durch die Zielhardware. Die Portierung einzelner, dann statisch ausgelegter, Steuer- und Regelfunktionen ist für einen Großteil moderner FPGA-Architekturen möglich. Die Umsetzung der Controllerkomponenten mit dedizierten Hardwareelementen muss in beiden Fällen mit adäquater dedizierter Hardware der Zielplattform oder alternativ mit elementaren Logikblöcken vorgenommen werden.

5.2 Rekonfigurationslösung

Die Umsetzung der partiellen Rekonfiguration von Xilinx FPGAs erfolgt für die hier entwickelten rekonfigurierbaren Controller auf Basis einer internen Konfigurationsschnittstelle. Die

Ansteuerung der Konfigurationsschnittstelle und ihre Einbindung in das Rekonfigurations- und Speichermanagement nach Abschnitt 4.2.2 bilden die Grundlage für das Laden von Hardwaretasks sowie die Realisierung der Selbstrekonfiguration des Systems.

5.2.1 Rekonfigurationsschnittstellen

Die in dieser Arbeit als Zielhardware gewählten Xilinx FPGAs verfügen über verschiedene Konfigurationsschnittstellen, die zur Umsetzung der partiellen Rekonfiguration eingesetzt werden können. Die grundlegende Funktion aller Konfigurationsschnittstellen ist dabei die Entgegennahme und Interpretation von Konfigurationsdaten, die zu verändernde Hardwarebereiche repräsentieren. Die Wahl einer bestimmten Konfigurationsschnittstelle beruht auf der spezifischen Anwendung, der konkreten Zielhardware und weiteren äußeren Rahmenbedingungen.

Grundsätzlich können die Konfigurationsschnittstellen von Xilinx FPGAs in externe und interne Schnittstellen unterschieden werden. Die externen Konfigurationsschnittstellen umfassen:

- eine serielle Konfigurationsschnittstelle mit einem Slave- Modus für den Einsatz externer Konfigurationscontroller wie Mikroprozessoren oder CPLDs und einem Master-Modus für den Einsatz spezieller serieller Konfigurationsdatenspeicher; Master-Modus nur für initiale Konfiguration

- eine als *SelectMAP* bezeichnete parallele Konfigurationsschnittstelle mit einem 8 Bit oder 32 Bit breiten Datenbus; ebenfalls im Slave- oder Master-Modus

- eine JTAG-Schnittstelle nach IEEE-Standard 1149.1/1532 (*Joint Test Action Group,* richtig: *Standard Test Access Port and Boundary-Scan Architecture* und *In-System Configuration*), vorrangig für die Konfiguration von Entwicklungsboards über ein PC-System, verfügt über dedizierte I/O-Pins.

Die aufgeführten externen Konfigurationsschnittstellen der Zielhardware können für die Umsetzung der partiellen Rekonfiguration eines Xilinx FPGAs eingesetzt werden [51], bedingen jedoch aufgrund ihrer Struktur eine übergeordnete externe Rekonfigurationssteuerung und in jedem Fall einen externen Konfigurationsdatenspeicher. Die übergeordnete externe Rekonfigurationssteuerung muss so mindestens die Ansteuerung der Konfigurationsschnittstellen und damit die Durchführung der Ladevorgänge für Hardwaretasks implementieren. Mögliche

Grundlagen für die externe Umsetzung einer Rekonfigurationssteuerung umfassen CPLDs, kleinere FPGAs und Mikroprozessoren. Im einfachsten Fall kann die Ansteuerung einer Konfigurationsschnittstelle automatisiert mit einem PC-System, Programmiersoftware und spezieller Konfigurationshardware (Programmierkabel) vorgenommen werden. Eine Selbstrekonfiguration der Zielhardware ist mit den aufgeführten externen Konfigurationsschnittstellen nicht oder nur umständlich möglich. Für weiterführende Informationen zu den Konfigurationsschnittstellen der Xilinx FPGAs sei auf die Datenblätter des Herstellers verwiesen ([82] [47] [83]).

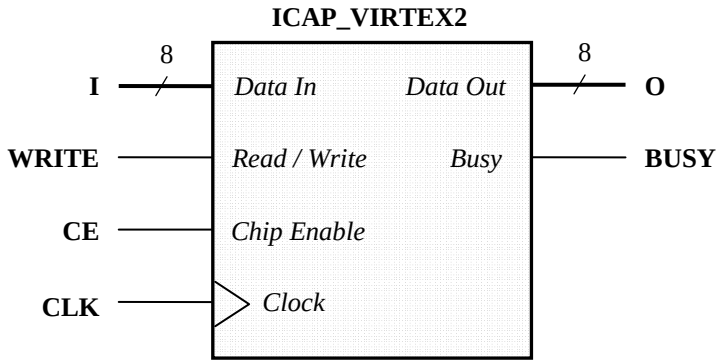


Abbildung 26: Interne Konfigurationsschnittstelle eines Xilinx Virtex-II FPGAs [47]

Zusätzlich zu den externen Konfigurationsschnittstellen verfügen Xilinx FPGAs der Virtex-Baureihen und teilweise auch der Spartan-Baureihen (Spartan-3A) über eine interne Konfigurationsschnittstelle. Diese wird als *Internal Configuration Access Port*, kurz ICAP, bezeichnet und dient den hier entwickelten rekonfigurierbaren Controllern für mechatronische Systeme als primäre Schnittstelle für die Umsetzung der partiellen Rekonfiguration. Die ICAP-Schnittstelle basiert auf der externen *SelectMAP*-Konfigurationslösung und stellt wie diese einen parallelen Datenbus für die Übermittlung der Konfigurationsdaten bereit. Dabei beträgt die Breite des Datenbusses analog zur externen *SelectMAP*-Schnittstelle für Spartan-3A und Virtex-II FPGAs 8 Bit und für Virtex-4 FPGAs wahlweise 8 oder 32 Bit. Für die jüngsten Bausteine der Virtex-Baureihe, Virtex-5 FPGAs, ist eine Busbreite von 8, 16 oder 32 Bit möglich. Der Datenbus ist unidirektional als Eingang und Ausgang vorhanden. Während der eingehende Bus für die Konfiguration genutzt werden kann, dient der ausgehende Datenbus dem Lesen von Statusregistern und der konfigurationsbezogenen Verifikation einzelner Hardwarebereiche. Neben dem Datenbus umfasst ICAP analog zur *SelectMAP*-Schnittstelle verschiedene Ansteuersignale sowie ein Kontrollsignal. Die Ansteuerung der ICAP-Schnittstelle folgt dem Protokoll für die externe *SelectMAP*-Schnittstelle und wird im folgen-

den Abschnitt näher beschrieben. Abbildung 26 zeigt den Aufbau von ICAP für ein Xilinx Virtex-II FPGA.

Für die Umsetzung der Ansteuerung und damit der Selbstrekonfiguration wird von Xilinx der Einsatz eingebetteter Mikroprozessorsysteme empfohlen [54]. Dafür werden von Xilinx vorkonfigurierte Peripheriekomponenten für den *On-Chip Peripheral Bus* (OPB) und den *Processor Local Bus* (PLB) zur Verfügung gestellt, die eine Ansteuerung von ICAP mit auf den eingebetteten Mikroprozessorsystemen ausgeführter Software ermöglichen. Der Konfigurationstakt ist dabei an den Takt des eingesetzten Bussystems gekoppelt. Die unterstützten Mikroprozessoren umfassen den in FPGA-Hardware implementierten MicroBlaze und den auf dedizierter Hardware basierenden PowerPC (vergleiche Abschnitt 1.2.3). Eine genauere Beschreibung sowie Details zum Ressourcenbedarf der als HWICAP bezeichneten Peripheriekomponenten findet sich in [84] und [85].

5.2.2 Rekonfigurationsmechanismus

Die Umsetzung der partiellen Rekonfiguration der Zielhardware erfolgt für die hier entwickelten Controller mit der internen Konfigurationsschnittstelle ICAP. Im Gegensatz zu den genannten Lösungen für eine Ansteuerung der Schnittstelle wird jedoch kein eingebettetes Mikroprozessorsystem eingesetzt. Statt einer Kombination aus Software und Buskomponenten basiert der Rekonfigurationsmechanismus ausschließlich auf einer direkten Hardwareimplementierung der Ansteuerung. Ausgangspunkt der Hardwareumsetzung ist das in Abbildung 27 dargestellte Protokoll für das Laden von Konfigurationsdaten durch ICAP. Das Protokoll beruht auf der Ansteuerung der externen *SelectMAP*-Schnittstelle und ist hier für einen 8 Bit Datenbus dargestellt [83].

Wie in Abbildung 27 zu erkennen, wird die ICAP-Schnittstelle durch das Setzen der Low-aktiven Ansteuersignale WRITE und CE in den Betriebszustand für das Schreiben von Konfigurationsdaten versetzt. Sollen nur Schreibvorgänge ausgeführt werden, können beide Signale auch dauerhaft gesetzt werden. Andernfalls ist WRITE immer vor CE zu setzen, da es sonst zum Abbruch des Schreibvorgangs kommt. Die Konfigurationsdaten werden mit jeder steigenden Flanke des Taktsignals CLK übernommen. Ihr Umfang richtet sich nach der möglichen und gewählten Datenbusbreite der Schnittstelle. Bei einer Datenbusbreite von 8 Bit ist zu beachten, dass die Konfigurationsbits innerhalb der einzelnen Bytes in reversierter Reihenfolge bereitzustellen sind. Dies gilt nicht für die anderen Datenbusbreiten. Ein Ladevorgang kann über Rücksetzen des CE-Signals oder Anhalten des Taktsignals pausiert werden. Das in

Abbildung 27 ebenfalls aufgeführte BUSY-Signal signalisiert die Annahmebereitschaft der Schnittstelle für neue Konfigurationsdaten und wird initial von der Schnittstelle bei aktivem CE-Signal zurückgesetzt. Eine Überwachung des BUSY-Signals ist für Spartan-3A und Virtex-II FPGAs bei einem Konfigurationstakt unter 50 MHz sowie allgemein für Virtex-4 FPGAs und neuere Bausteine nicht erforderlich.

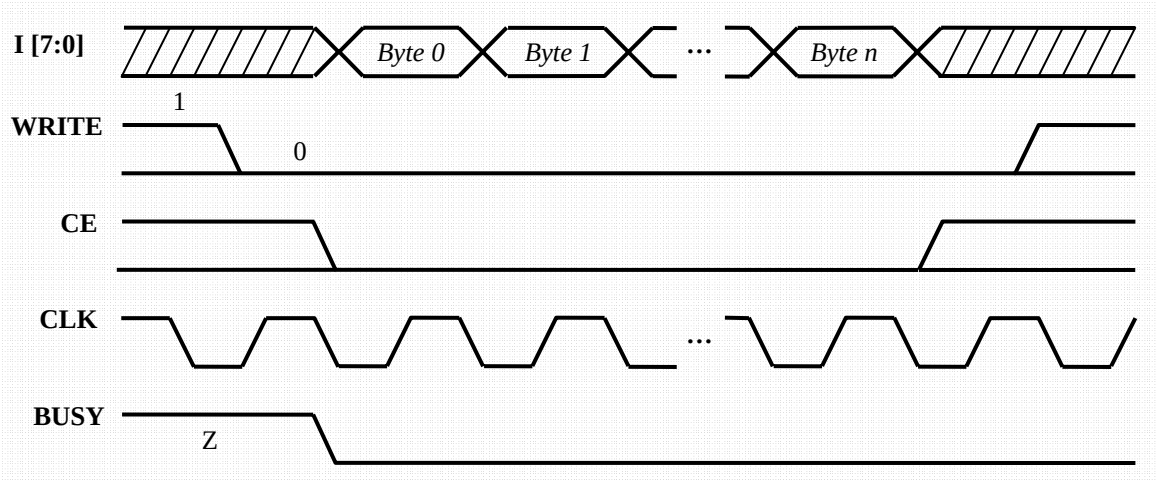


Abbildung 27: Ansteuersignale für ICAP, Schreiben von Konfigurationsdaten

Anzumerken ist, dass für eine korrekte Funktion der ICAP-Schnittstelle der hier eingesetzten Virtex-II und Virtex-4 FPGAs das Protokoll geringfügig modifiziert werden muss. Im Falle des Xilinx Virtex-II XC2V1000 FPGAs sind so die WRITE und CE-Signale dauerhaft aktiv zu setzen oder zwischen den Ladevorgängen mehrfach zu setzen und rückzusetzen. Ein einfaches Schalten der Signale bewirkt die Inaktivität der ICAP-Schnittstelle bei jedem zweitem Ladevorgang. Im Falle des Xilinx Virtex-4 XC4VLX60 FPGAs ist das CE-Signal für jeden Ladevorgang im 32 Bit Modus zu setzen und danach rückzusetzen. Ein dauerhaftes Setzen führt ebenfalls zur Inaktivität der ICAP-Schnittstelle.

Die Hardwareumsetzung des oben dargestellten Protokolls erfolgt in den hier entwickelten rekonfigurierbaren Controllern mit einem direkt in elementarer FPGA-Hardware implementierten Zustandsautomaten (Rekonf-FSM). Der Zustandsautomat ist an das in Abschnitt 4.2.2 beschriebene Rekonfigurations- und Speichermanagement angebunden. Dieses übermittelt im Rekonfigurationsfall die Tasknummer eines zu ladenden Hardwaretasks an die Rekonf-FSM und initiert mit einem Schreibbefehl (*wr2icap*) die partielle Rekonfiguration des zugehörigen Moduls. Der Automat greift mit der Tasknummer auf die Tasktabelle des Rekonfigurations- und Speichermanagements zu und ermittelt so die Größe und Speicheradresse des zu ladenden Hardwaretasks. Daraufhin erfolgt ein direkter Lesezugriff auf den Hauptspeicher des Control-

lers und die Konfigurationsdaten des zu ladenden Hardwaretasks werden umgehend mit dem oben gezeigten Protokoll an die ICAP-Schnittstelle des FPGAs geschrieben. Der Datenausgang des als Dual-Port-RAM ausgelegten Hauptspeichers ist dabei direkt mit dem Dateneingang der Konfigurationsschnittstelle verbunden. Nach Beendigung des Ladevorganges für einen Hardwaretasks befindet sich der Zustandsautomat wieder in seinem Ausgangszustand und startet bei Übermittlung einer Tasknummer den nächsten Ladevorgang. Abbildung 28 zeigt eine abstrahierte Variante des Zustandsautomaten. Reale Umsetzungen des Automaten generieren zusätzlich Statussignale, die den Fortschritt der Ladevorgänge anzeigen.

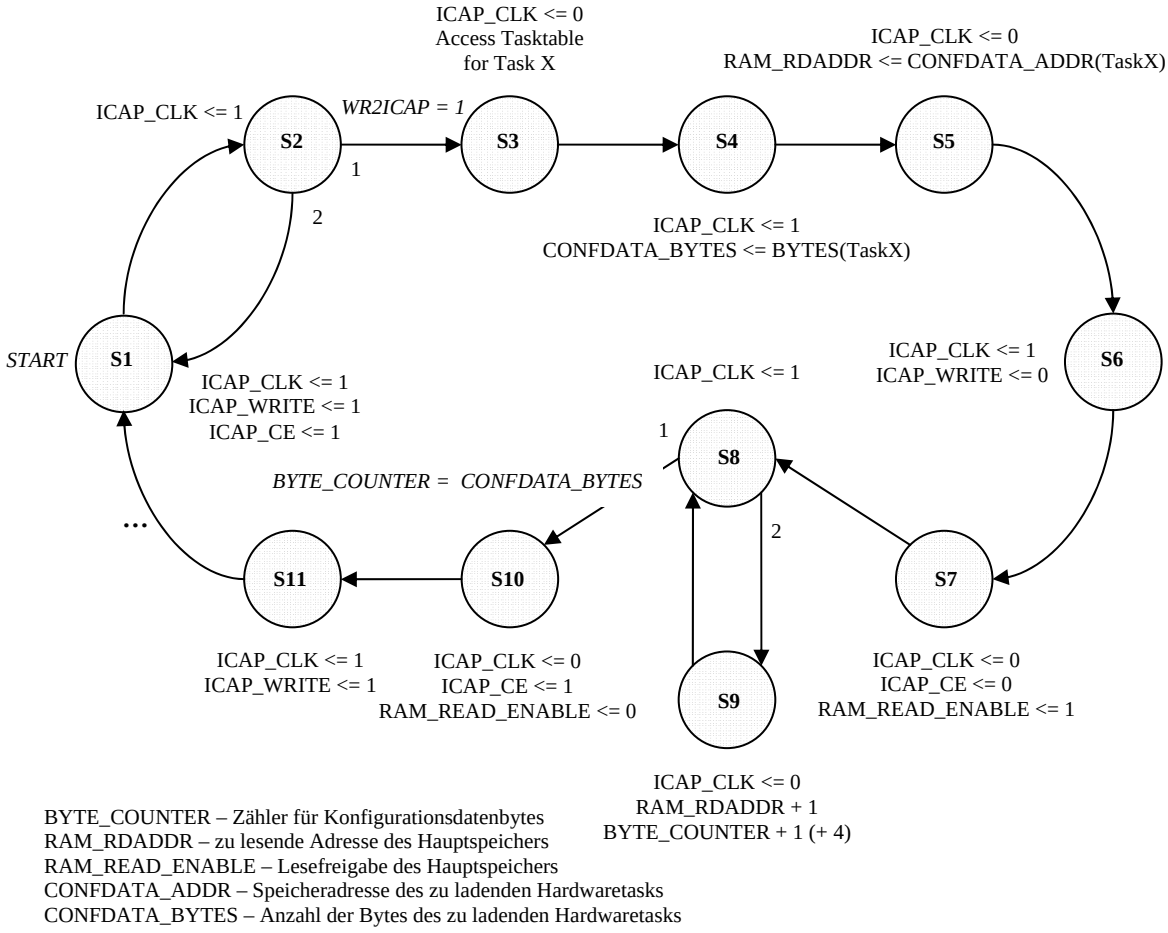


Abbildung 28: Zustandsautomat für die Ansteuerung von ICAP (Rekonf-FSM)

Die direkte Hardwareimplementierung des Zustandsautomaten für die Ansteuerung der internen Konfigurationsschnittstelle ermöglicht einen frei einstellbaren Konfigurationstakt und damit eine Beeinflussung der Rekonfigurationszeiten. Grundlage für die Einstellung des Konfigurationstaktes ist die Variation des Automatentaktes mit einem digitalen Clockmanager. Alternativ kann der Konfigurationstakt auch direkt mit einem DCM erzeugt werden. Weiterhin unterstützt der direkte Speicherzugriff des Zustandsautomaten eine Ausführung der Lade-

vorgänge für Hardwaretasks unter harten Echtzeitbedingungen, da die benötigte Rekonfigurationszeit im Voraus bekannt ist und die Rekonfiguration sicher in dieser Zeit vollzogen wird. Darüber hinaus erlaubt die direkte Hardwareimplementierung eine effektive und Ressourcen schonende Abbildung des Automaten auf der Zielhardware. So wurde eine Beispielimplementierung für den 32 Bit Modus der ICAP-Schnittstelle eines Xilinx Virtex-4 XC4VLX60 FPGAs und einen 256 KB RAM als Hauptspeicher des Systems mit einem synthesebasierten Ressourcenverbrauch von 82 CLB Slices (163 DFFs, 157 LUTs) umgesetzt.

Der Zustandsautomat für die Ansteuerung der ICAP-Schnittstelle ist vorkonfiguriert für die verschiedenen Ausprägungen der Schnittstelle Bestandteil der Funktionsbiliotheken und damit Teil des Spezifikationsframeworks nach Abschnitt 4.3. Für die Simulation und messtechnische Überprüfung des Rekonfigurationsmechanismus sei auf Kapitel 6 verwiesen.

5.2.3 Rekonfigurationszeiten

Da mit dem vorgestellten Rekonfigurationsmechanismus ein direkter Zugriff auf die Konfigurationsdaten und die interne Konfigurationsschnittstelle ICAP erfolgt, sind die erzielbaren Rekonfigurationszeiten unmittelbar vom Umfang der zu ladenden Konfigurationsdaten, der möglichen Konfigurationstaktrate und der Datenbusbreite der Schnittstelle abhängig. Für den 8 Bit Modus der Schnittstelle gilt so folgender Zusammenhang:

$$T_R = \frac{1}{f_{ICAP\_CLK}} * x_{Bytes} \qquad \text{Gleichung 5-2}$$

Für den 32 Bit Modus gilt analog:

$$T_R = \frac{1}{f_{ICAP\_CLK}} * \frac{x_{Bytes}}{4} \qquad \text{Gleichung 5-3}$$

Dabei ist x_{Bytes} die Anzahl der an ICAP zu schreibenden Konfigurationsdatenbytes, f_{ICAP_CLK} die Taktfrequenz der Schnittstelle und T_R die resultierende Rekonfigurationsdauer. Die Anzahl der zu schreibenden Konfigurationsdatenbytes ist von der Größe des zu rekonfigurierenden Moduls und den Implementierungsoptionen (vergleiche Abschnitt 5.3) abhängig. Die möglichen Maximalfrequenzen für das Schreiben von Konfigurationsdaten an die ICAP-Schnittstelle sind bausteinspezifisch und in Tabelle 4 aufgeführt. Die Maximalfrequenz für Virtex-II und Spartan-3 FPGAs kann bei Beobachtung des BUSY-Signals erhöht werden. Bei gesetztem BUSY-Signal nimmt die Konfigurationschnittstelle jedoch keine Daten entgegen, so dass sich die Rekonfigurationszeit um die Haltetakte vergrößert.

Die Maximalfrequenz der ICAP-Schnittstelle von Virtex-4 FPGAs kann ebenfalls abweichend von den Angaben in den Datenblättern erhöht werden. So gibt Xilinx eine Maximalfrequenz von 128 MHz für eine Beispielimplementierung der erwähnten PLB-HWICAP-Peripheriekomponente an [85]. Die tatsächlich erreichbare Frequenz variiert jedoch und ist vom konkreten Anwendungsfall abhängig.

FPGA	$\mathbf{f_{ICAP\_CLK\_MAX}}$	$\mathbf{f_{ICAP\_CLK\_MAX}}$ **(BUSY)**
Spartan-3	50 MHz	66 MHz
Virtex-II	50 MHz	66 MHz
Virtex-4	100 MHz	entfällt

Tabelle 4: Maximalfrequenzen für ICAP nach [79], [80] und [81]

Als Beispiel für mit dem vorgestellten Rekonfigurationsmechanismus erzielbare Rekonfigurationszeiten sei hier ein Modul von 4 mal 20 CLBs auf einem Virtex-4 XC4VLX60 FPGA angeführt. Die 35824 Konfigurationsdatenbytes eines zugehörigen Hardwaretasks können im 32 Bit Modus mit 100 MHz innerhalb von 89,56 µs geladen werden. Die Anzahl der Konfigurationsdatenbytes für andere Hardwaretasks des Moduls kann aufgrund von Implementierungsoptionen variieren. Weitere Beispiele für konkrete Anwendungen sind in Kapitel 6 der Arbeit zu finden.

5.3 Implementierungsschritte

Die Implementierung der rekonfigurierbaren Controller erfolgt auf der Basis herstellerspezifischer Vorgehensweisen und deren Anbindung an das Spezifikationsframework nach Abschnitt 4.3. Grundlage für die Implementierung sind dabei Softwarewerkzeuge von Xilinx und ihre Anwendung auf die Funktionsgruppen der Controller. Das Ergebnis der Hardwareimplementierung sind Konfigurationsdaten für das Gesamtsystem und einzelne Hardwaretasks.

5.3.1 Herstellerspezifische Implementierungsschritte

Die herstellerspezifischen Implementierungsschritte zur Umsetzung von partiell rekonfigurierbaren Systemen auf Xilinx FPGAs sind im *Early Access Partial Reconfiguration Design Flow* (EAPR Design Flow) zusammengefasst [51]. Der EAPR Design Flow basiert auf dem

älteren *Module Based Partial Reconfiguration Design Flow* [52]. Er erweitert diesen um die partielle Rekonfiguration von Virtex-4 FPGAs sowie um rechteckige, nicht die ganze Höhe des Bausteins umfassende, rekonfigurierbare Module. Darüber hinaus bietet der EAPR Design Flow eine vereinfachte Behandlung von Signalen, die rekonfigurierbare Hardwarebereiche durchqueren. Beide Implementierungsmethoden nutzen die Standardimplementierungswerkzeuge von Xilinx [86], die im Fall des EAPR Design Flows in einer modifizierten Version vorliegen müssen.

Im Folgenden wird eine Übersicht über den EAPR Design Flow gegeben und dessen Anwendung auf die in dieser Arbeit entwickelten rekonfigurierbaren Controller beschrieben. Abbildung 29 illustriert die Vorgehensweise. Für eine detaillierte Darstellung der Implementierungsschritte sei auf [51] und [49] verwiesen.

Vorbereitung

Ausgangspunkt für die Implementierung eines partiell rekonfigurierbaren Controllers mit dem EAPR Design Flow sind die Syntheseergebnisse des statischen Basissystems und der einzelnen Hardwaretasks. Die Netzliste des statischen Basissystems enthält dabei alle statischen, nicht rekonfigurierbaren Bestandteile des Controllers. Ergänzt werden die Syntheseergebnisse des Basissystems und der Hardwaretasks durch eine übergeordnete Ebene, das Top-Level. Das Top-Level instanziiert die Controllerbestandteile als Blackbox und beinhaltet zusätzlich dedizierte Hardwarekomponenten auf Systemebene, da die Implementierungswerkzeuge keine globale Logik in untergeordneten Ebenen unterstützen. Daher sind I/O-Pins, digitale Clockmanager, die ICAP-Schnittstelle und die Bus-Makros dem Top-Level zuzuordnen. Die Netzliste des Top-Levels kann mit einem beliebigen Synthesewerkzeug aus einer einfachen strukturellen HDL-Beschreibung erzeugt werden.

Der nächste Schritt der Implementierungsvorbereitungen ist die Festlegung hardwarebezogener Randbedingungen, der so genannten *Constraints*. Die Randbedingungen weisen dem Basissystem und den als Blackbox für Hardwaretasks repräsentierten rekonfigurierbaren Modulen feste Bereiche auf der konkreten Zielhardware zu. Die Wahl und Ausprägung rekonfigurierbarer Bereiche ist von den Möglichkeiten des verwendeten FPGA-Typs abhängig (vergleiche Abschnitt 5.1.1). Rekonfigurierbare Bereiche des Systems werden zusätzlich mit einem speziellen Attribut gekennzeichnet. Darüber hinaus umfassen die Randbedingungen Angaben zur Platzierung aller im Top-Level aufgeführten Hardwarekomponenten, wie I/Os, digitale Clockmanager, ICAP und Bus-Makros. Die Festlegung der Randbedingungen kann mit grafi-

schen Werkzeugen von Xilinx oder manuell mit einem Texteditor erfolgen. In beiden Fällen werden die Randbedingungen in einer als *User Constraint File* (*.ucf) bezeichneten Textdatei abgebildet und an die Implementierungswerkzeuge übergeben.

Vor der Anwendung des EAPR Design Flows wird von Xilinx weiterhin eine statische Implementierung des Gesamtsystems empfohlen. Diese wird mit den Standardimplementierungswerkzeugen durchgeführt und folgt den herkömmlichen Implementierungsschritten für Xilinx FPGAs [86]. Die statische Umsetzung sollte für jede Kombination rekonfigurierbarer Funktionalitäten vorgenommen werden. Auf der Basis der statischen Implementierung können Analysen des Zeitverhaltens und auftretender Implementierungsfehler vorgenommen sowie die Randbedingungen der Hardwareumsetzung angepasst werden.

Durchführung

Die Umsetzung eines partiell rekonfigurierbaren Systems mit dem EAPR Design Flow beginnt mit der Implementierung des statischen Basissystems. Die Netzliste des Basissystems und des Top-Levels werden dabei durch die Implementierungswerkzeuge in eine native Beschreibungsform übersetzt (*ngdbuild*), auf die Zielhardware abgebildet (*map*) sowie auf der Elementarebene platziert und verbunden (*place and route*). Die rekonfigurierbaren Bereiche bleiben wie in den Randbedingungen festgelegt erhalten, sind jedoch leer. Das Ergebnis des ersten Implementierungsschritts ist eine native, mit grafischen Werkzeugen von Xilinx lesbare Datei für das implementierte Teilsystem. Zusätzlich wird eine als *static.used* bezeichnete Datei angelegt, die Informationen über rekonfigurierbare Bereiche durchquerende Signale enthält und als Vorgabe für nachfolgende Implementierungsschritte dient.

Die Implementierung der rekonfigurierbaren Bereiche erfolgt daraufhin in gleicher Weise (*ngdbuild, map, place and route*) zusammen mit dem Top-Level des Systems. Das statische Basissystem wird hier mit der Datei *static.used* berücksichtigt und die dort erfassten Signalleitungen nicht für die Implementierung rekonfigurierbarer Funktionalitäten verwendet. Dieser Implementierungsschritt muss für jeden Hardwaretask eines rekonfigurierbaren Bereiches gesondert durchlaufen werden. Soll ein Task in mehr als einem rekonfigurierbaren Modul implementiert werden, ist der Implementierungsschritt ebenfalls für jedes Modul gesondert zu durchlaufen. Das Ergebnis der Teilimplementierung sind native Beschreibungen der einzelnen Hardwaretasks, die deren Funktionalität ortsfest innerhalb der zugehörigen rekonfigurierbaren Bereiche auf der Elementarebene der Zielhardware abbilden.

Anschließend werden die Ergebnisse der Einzelimplementierungen des Basissystems und der Hardwaretasks im finalen Implementierungsschritt (*Merge*) vereinigt und Konfigurationsdaten für das Gesamtsystem und die einzelnen Hardwaretasks generiert. Dabei kommen die Werkzeuge *PR_verifydesign* und *PR_assemble* zum Einsatz, die im Rahmen des EAPR Design Flows bereitgestellt werden. Die Werkzeuge integrieren Überprüfungen der Ausgangsdaten, die Vereinigung der Teilimplementierungen und die Generierung der finalen Konfigurationsdaten. Der Implementierungsschritt muss ebenfalls für jeden Hardwaretask separat durchgeführt werden. Sind mehrere rekonfigurierbare Module im System vorhanden, ist die finale Implementierung iterativ für alle Kombinationen der Hardwaretasks vorzunehmen.

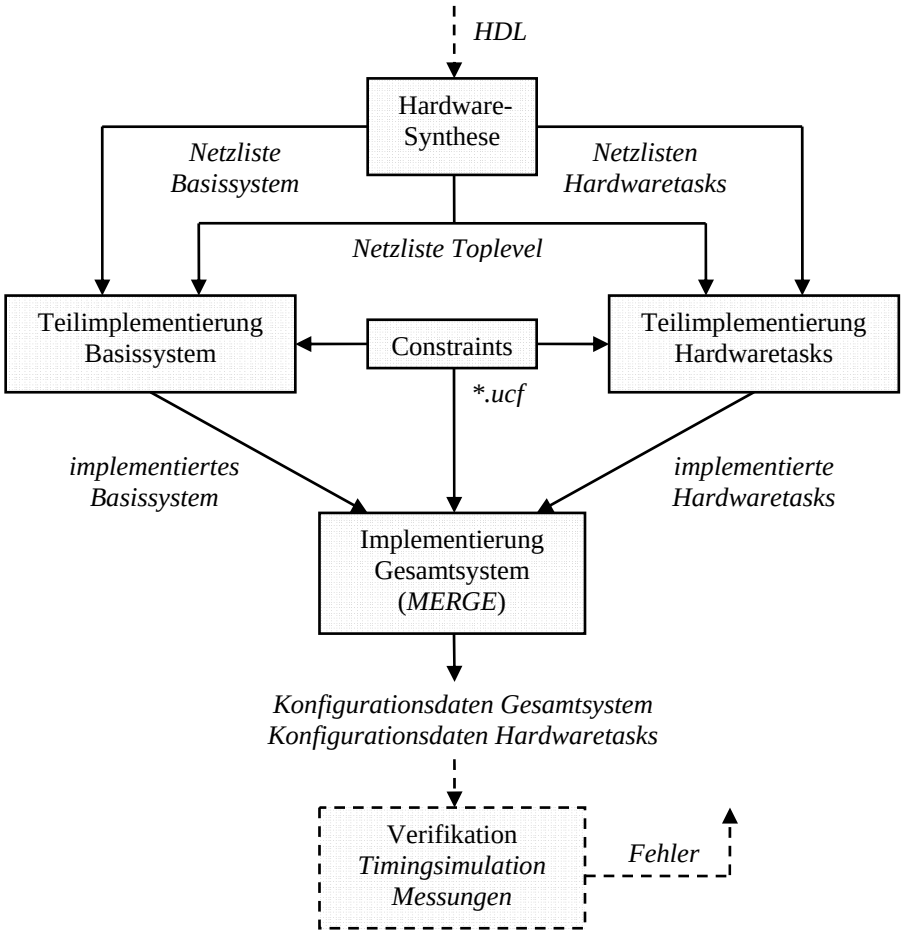


Abbildung 29: Implementierungsschritte des EAPR Design Flows

An dieser Stelle sei darauf hingewiesen, dass die finalen Konfigurationsdaten innerhalb von *PR_assemble* durch das Werkzeug *BitGen* der Standardimplementierungswerkzeuge von Xilinx erzeugt werden [86]. Dies geschieht für die Hardwaretasks standardmäßig unter Nutzung einer Option zur Komprimierung der Konfigurationsdaten, die als *Multiple-Frame-Write*-Verfahren bezeichnet wird [47]. Hierbei werden identische Konfigurationsdaten für die kleinsten programmierbaren Bereiche der Zielhardware zusammengefasst und beim Schreiben

an die Konfigurationsschnittstelle für die Konfiguration mehrerer Bereiche genutzt. Daher kann der Umfang der Konfigurationsdatensätze für ein rekonfigurierbares Modul unabhängig von der Modulgröße variieren.

5.3.2 Anbindung an das Spezifikationsframework

Die Anbindung der herstellerspezifischen Implementierungsschritte an das Spezifikationsframework nach 4.3 erfolgt durch die Hardwaresynthese sowie die Umsetzung des EAPR Design Flows für rekonfigurierbare Controller. Die Steuerung des hier eingesetzten Synthesewerkzeugs Precision Synthesis von Mentor Graphics ist dabei aus der grafischen Oberfläche des Entwurfswerkzeugs HDL Designer möglich. Vorraussetzung der Hardwaresynthese ist hier die hierarchische Zuordnung der Controllerkomponenten zu Top-Level, Basissystem und Hardwaretasks. Die auf die Hardwaresynthese folgende Aufstellung der zielhardwarespezifischen Randbedingungen wird dagegen textbasiert oder mit grafischen Werkzeugen durchgeführt. Entsprechende Werkzeuge sind der Pinout and Area Constraints Editor PACE sowie der so genannte Floorplanner, beide sind in den Standardentwicklungswerkzeugen von Xilinx enthalten.

Die Umsetzung des EAPR Design Flows für rekonfigurierbare Controller erfolgt schließlich mit anwendungsspezifischen Skripten. Die Skripte koordinieren den Ablauf der Implementierung, rufen die herstellerspezifischen Softwarewerkzeuge auf und verwalten anfallende Daten. Dabei folgt die skriptbasierte Implementierung der rekonfigurierbaren Controller weitgehend den Vorgaben in [51]. Im Anschluss an den EAPR Design Flow werden Konfigurationsdaten, die über die USB-Schnittstelle eines rekonfigurierbaren Controllers gesendet werden sollen, zusätzlich mit dem in Abschnitt 4.2.2 aufgeführten Nachrichtenkopf versehen.

5.4 Verifikation

Die Verifikation des implementierten Gesamtsystems schließt die Implementierungsschritte ab und erfolgt sowohl auf digitaler Hardwareebene als auch auf Funktionsebene der rekonfigurierbaren Controller. Die eingesetzten Verifikationsmethoden umfassen schwerpunktmäßig Timingsimulationen der Controller und werden durch Messungen auf Hardware- und Funktionsebene ergänzt.

5.4.1 Timingsimulationen

Timingsimulationen finden im Gegensatz zu Verhaltenssimulationen nach der Implementierung eines FPGA-Designs statt und ermöglichen eine sehr detaillierte Analyse des Systemverhaltens unter Berücksichtigung von spezifischen Eigenschaften der Zielhardware und Signallaufzeiten. Ausgangspunkt der Timingsimulationen ist die Überführung eines vollständig implementierten rekonfigurierbaren Controllers in eine HDL-basierte Netzliste, die dessen Funktionalität und Struktur auf elementarer Hardwareebene der Zielplattform wiedergibt. Hierfür wird das Werkzeug *NetGen* der Standardentwicklungswerkzeuge von Xilinx eingesetzt [86]. Zusätzlich zu der HDL-basierten Netzliste generiert *NetGen* ausgehend von Timinganalysen ausführliche Informationen zu Signallaufzeiten und dem Zeitverhalten elementarer Hardwarekomponenten. Die Verbindung von HDL-Netzliste und zugehörigen Zeitinformationen erlaubt eine hochgenaue Simulation der Funktionalität und des Zeitverhaltens eines Controllers, die das reale Verhalten der Hardware weitestgehend abbildet.

Die Timingsimulationen der Controller werden mit einem HDL-Simulator wie ModelSim von Mentor Graphics durchgeführt [77]. Die generierte HDL-Netzliste des implementierten Controllers wird hierfür in einer zugeschnittenen Testumgebung instanziiert und zusammen mit den Zeitinformationen an den HDL-Simulator übergeben. Die Testumgebung ist ebenfalls in einer Hardwarebeschreibungssprache zu erstellen und beinhaltet nicht synthesefähige Verhaltensmodelle für FPGA-externe Controllerbestandteile (Kommunikation, Wandler, …) und Prozesskomponenten (Stellglieder, Regelstrecke, …). Die Funktionsbibliotheken des Spezifikationsframeworks nach Abschnitt 4.3 stellen vorkonfigurierte Testumgebungen und Verhaltensmodelle zur Verfügung. Weiterhin ist dem HDL-Simulator die so genannte *simprim*-Bibliothek von Xilinx zu übergeben, die Verhaltensmodelle für die elementaren Hardwareelemente enthält. Die Bibliothek wird von Xilinx im Rahmen der Standardentwicklungswerkzeuge bereitgestellt. Abbildung 30 illustriert die Vorgehensweise für die Erstellung einer Timingsimulation der rekonfigurierbaren Controller.

Die Durchführung einer Timingsimulation ermöglicht im Gegensatz zu Messungen eine genaue Analyse auch interner Signale und Komponenten. So kann die Funktion und das Zeitverhalten von Steuer- und Regelalgorithmen und Controllerbestandteilen, wie dem Rekonfigurations- und Speichermanagement oder dem Kommunikationssystem, auf Funktionsebene und Hardwareebene bitgenau untersucht werden. Dedizierte Hardwarekomponenten wie RAM-Blöcke und digitale Clockmanager werden durch die *simprim*-Bibliothek abgebildet und können so ebenfalls berücksichtigt werden. Nicht durch eine Timingsimulation unter-

stützt wird die Ausführung der partiellen Rekonfiguration mit ICAP. Simuliert wird daher ein Gesamtcontroller mit fest implementierten Hardwaretasks, wie er für die initiale Konfiguration der Zielhardware eingesetzt wird. Die funktionale Überprüfung der Ansteuerung der ICAP-Schnittstelle und der übergeordneten Rekonfigurationssteuerung erfolgt unabhängig von einer simulativen Umsetzung der Ladevorgänge.

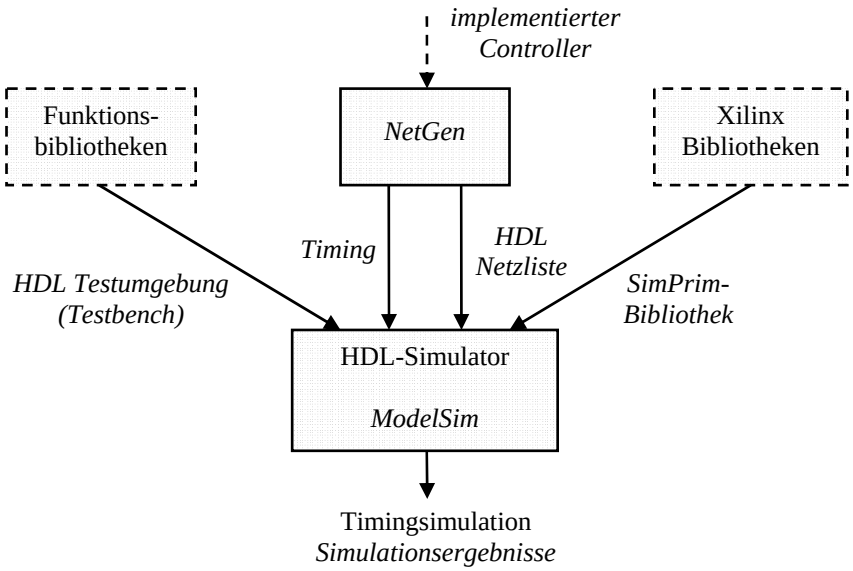


Abbildung 30: Vorgehensweise für die Erstellung von Timingsimulationen

Bedingt durch die Größenordung der generierten Zeitinformationen für einzelne Hardwarekomponenten ist die zeitliche Auflösung der Timingsimulation sehr hoch einzustellen. Typische Auflösungen bewegen sich im Pikosekundenbereich (10^{-12} s) und bewirken zusammen mit der Komplexität der zugrunde liegenden Netzliste einen relativ hohen Rechenaufwand. Repräsentative Timingsimulationen der hier entwickelten rekonfigurierbaren Controller mit einer simulierten Funktionsdauer im zwei- und dreistelligen Millisekundenbereich erfordern mit heutiger PC-Hardware Rechenzeiten von mehreren Stunden.

5.4.2 Messungen

Die Verifikation rekonfigurierbarer Controller mit Messungen erfolgt nach der Inbetriebnahme der Zielhardware auf Hardware- und Funktionsebene. Anders als bei Timingsimulationen kann hier die Detektion und Überwachung von Rekonfigurationsvorgängen vorgenommen werden. Weiterhin ermöglichen Messungen auf Funktionsebene eine Überprüfung der implementierten Steuer- und Regelalgorithmen im Kontext eines bestehenden mechatronischen Systems.

Die Messungen auf digitaler Hardwareebene werden vorrangig mit einem Logikanalysator umgesetzt. Logikanalysatoren können die Signalverläufe digitaler Ausgangssignale eines FPGAs bildlich darstellen und erlauben damit eine weitgehende Analyse der Funktionalitäten rekonfigurierbarer Controller. Die Erfassung interner Größen durch einen Logikanalysator ist bereits im Entwurfsprozess zu berücksichtigen, da die betreffenden Signale an physikalische Ausgänge der Zielhardware geführt werden müssen. Diese Vorgehensweise erhöht jedoch die Komplexität eines Controllers und kann eine Beeinflussung der Signallaufzeiten bewirken.

Alternativ zu Logikanalysatoren können FPGA-interne Datenerfassungsmethoden eingesetzt werden. Diese Vorgehensweise basiert auf zusätzlich in der Zielhardware implementierten Datenerfassungs- und Kommunikationskomponenten, die ausgewählte Signalverläufe aufzeichnen und einer übergeordneten Ebene übermitteln. Die Methode kann um das Auslesen von Statusregistern der Zielhardware und Konfigurationsinformationen (*Readback*) durch die ICAP-Schnittstelle ergänzt werden. Neben anwendungsspezifischen Eigenentwicklungen kommt vor allem das Werkzeug *ChipScope* von Xilinx zur Anwendung. *ChipScope* unterstützt die Generierung der Datenerfassungs- und Kommunikationsmodule sowie deren Integration in einen bestehenden Hardwareentwurf. Aufgezeichnete Daten können über die JTAG-Schnittstelle eines FPGAs an einen PC übermittelt und dort grafisch analysiert werden. Wie beim Einsatz eines Logikanalysators kann es auch hier zu einer Beeinflussung der Signallaufzeiten kommen. Darüber hinaus belegen die Datenerfassungs- und Kommunikationsmodule zusätzliche Ressourcen der Zielhardware.

Die Messungen auf der Funktionsebene rekonfigurierbarer Controller beziehen sich vorrangig auf die implementierten Steuer- und Regelalgorithmen im Kontext eines mechatronischen Systems. Die digitale Funktionalität der Steuer- und Regelalgorithmen wird dabei mit den oben vorgestellten Methoden der Hardwareebene überprüft. Dies geschieht vor allem über die Erfassung einzelner digitaler Prozess- und Regelgrößen. Die Einsatzmöglichkeiten von Logikanalysatoren und FPGA-interner Datenerfassung im Kontext eines realen mechatronischen Systems sind jedoch durch die unterschiedlichen Zeithorizonte und begrenzte Speicherkapazitäten der Datenerfassung stark eingeschränkt. Die weiterführende Verifikation der Funktionalität der Steuer- und Regelalgorithmen muss daher mit anwendungsspezifischen externen Messwerkzeugen und Methoden erfolgen.

6 Anwendungen

Dieses Kapitel der Arbeit stellt Anwendungen rekonfigurierbarer Controller in mechatronischen Systemen vor. Ausgehend von den entwickelten Methoden und Funktionsgruppen werden rekonfigurierbare Controller für konkrete Anwendungen entworfen und exemplarisch auf ausgewählten FPGAs implementiert. Schwerpunkte des Kapitels sind ein rekonfigurierbarer Antriebscontroller für DC-Antriebe und ein rekonfigurierbarer Controller für die Ansteuerung piezo-elektrischer Aktoren. Die Controller umfassen neben Infrastrukturkomponenten anwendungsspezifische Schnittstellen und Hardwaretasks, die mit Elementen aus Funktionsbibliotheken des Spezifikationsframeworks umgesetzt werden. Ergänzend zu anwendungsbezogenen Implementierungen diskutiert das Kapitel den Einsatz rekonfigurierbarer Controller für die Steuergeräteentwicklung und das Rapid Control Prototyping.

Der erste Abschnitt des Kapitels beschreibt Spezifikation, Struktur und Komponenten eines rekonfigurierbaren Antriebscontrollers sowie dessen Umsetzung und Test. Im zweiten Abschnitt wird ein rekonfigurierbarer Controller für die Ansteuerung piezo-elektrischer Aktoren vorgestellt. Der Controllerentwurf umfasst hier ebenfalls Spezifikation, Struktur und Komponenten und wird durch Implementierung und Verifikation vervollständigt. Der dritte und letzte Abschnitt des Kapitels geht auf die Einsatzmöglichkeiten rekonfigurierbarer Controller in der Funktionsentwicklung von Steuergeräten ein.

6.1 Rekonfigurierbarer Antriebscontroller

Der rekonfigurierbare Antriebscontroller basiert auf Steuer- und Regelfunktionalitäten für DC-Antriebssysteme, die allgemein in der Literatur beschrieben [32] und gewöhnlich mit Mikrocontrollern umgesetzt werden. Grundlage des Controllers ist hier jedoch eine direkte Hardwareumsetzung der Steuerungs- und Regelungsalgorithmen auf FPGAs und der Austausch dieser Algorithmen zur Laufzeit. Beispiele für Steuer- und Regelfunktionalitäten sind

einschleifige Positions- und Drehzahlregelungen sowie Kaskadenstrukturen mit unterlagerter Stromregelung. Ergänzt werden die rekonfigurierbaren Steuerungs- und Regelungsalgorithmen durch antriebsspezifische Schnittstellen. Interne und externe Kommunikation, die Überwachung und Ausführung der partiellen Rekonfiguration sowie die Speicherverwaltung werden mit den in Kapitel 4 vorgestellten Funktionsgruppen realisiert.

Im Gegensatz zu einer Umsetzung der Steuerungs- und Regelungsalgorithmen mit einem Mikrocontroller sind Verarbeitungswortbreite und Abtastrate hier basierend auf der direkten Hardwareimplementierung frei einstellbar. Der Einsatz partieller Rekonfiguration und die so erzielbare funktionsbezogene Strukturvariabilität der Zielplattform erhöhen zusätzlich die Flexibilität der Hardwareimplementierung.

6.1.1 Spezifikation

Die Spezifikation des rekonfigurierbaren Antriebscontrollers erfolgt auf der Grundlage der logischen Controllerstruktur. Diese erfasst die umzusetzenden Funktionalitäten des Controllers und kann abstrahierend durch einen Zustandsautomaten abgebildet werden. So wird der rekonfigurierbare Antriebscontroller hier ausgehend vom in Abbildung 8 (Abschnitt 3.1.3) aufgeführten Zustandsautomaten für Hochlauf und Betrieb mechatronischer Systeme entworfen. Die Entwurfsschritte sind jedoch ohne weiteres auf andere Problemstellungen der Mechatronik und elektrischen Antriebstechnik anwendbar. Die vorliegende logische Controllerstruktur wird einer Partitionierung unterzogen und die Funktionalitäten des Controllers so in statische und rekonfigurierbare Komponenten zerlegt. Die statischen Komponenten des Antriebscontrollers umfassen hierbei Schnittstellen und Infrastrukturkomponenten. Die rekonfigurierbaren Controllerkomponenten werden durch die Steuer- und Regelfunktionen der einzelnen Betriebszustände gestellt. Aus der Zuordnung der Controllerkomponenten zu einer konkreten Zielplattform resultiert die technische Controllerstruktur, die zusammen mit den statischen Komponenten im nachfolgenden Abschnitt näher beschrieben wird.

Die rekonfigurierbaren Steuer- und Regelfunktionen der Betriebszustände des Antriebscontrollers werden im nächsten Entwurfsschritt in eine Modul-FSM und zugehörige Hardwaretasks überführt. Die in Abbildung 12 (Abschnitt 3.3.1) dargestellte einfache Modul-FSM wird dazu um die Vorablade- und Aktivierungsstrategie nach Abschnitt 3.3.3 erweitert, um so den Wechsel zwischen Betriebszuständen ohne einen Verlust von Abtastschritten zu ermöglichen. Weiterhin wird der dort nur generisch aufgeführte Hauptbetriebszustand beispielhaft durch einen PI-Drehzahlregler ersetzt. Die Modul-FSM repräsentiert damit ein Task-Set von vier

Hardwaretasks, die neben dem PI-Drehzahlregler statische und dynamische Systemtests sowie einen sicheren Systemzustand für den Fehlerfall umfassen. Auf der Grundlage der Vorabladestrategie werden zwei zu einem Supermodul zusammengefasste rekonfigurierbare Module für die Abbildung der Hardwaretasks auf der Zielhardware benötigt. Die resultierende Modul-FSM ist in Abbildung 31 dargestellt. Bei Nichterfüllung der statischen und dynamischen Tests wird der jeweilige Testvorgang wiederholt. Die Erweiterung der Modul-FSM um Transitionen und Zustände für eine ergänzende Testauswertung und -reaktion ist unter Beachtung der bestehenden Hardwaretasks und vorhergehender Entwurfschritte möglich.

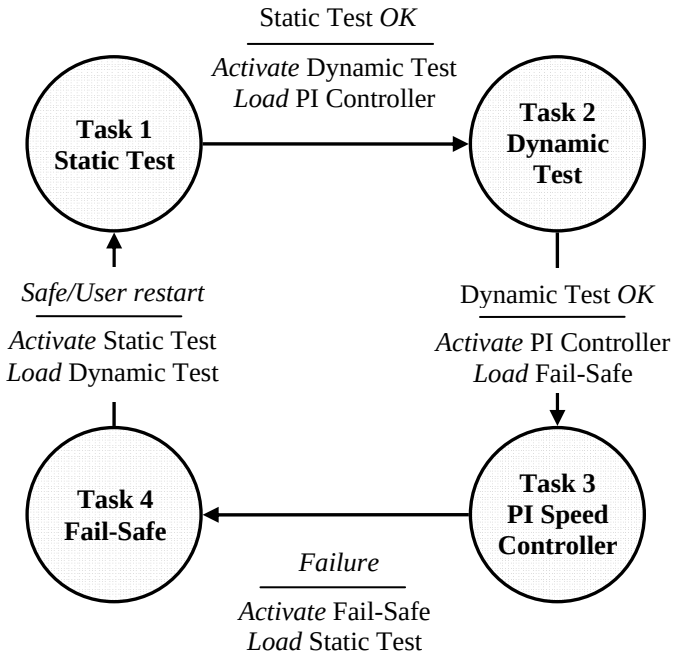


Abbildung 31: Modul-FSM des rekonfigurierbaren Antriebscontrollers

6.1.2 Struktur und Komponenten

Die Struktur des rekonfigurierbaren Antriebscontrollers basiert auf dem in Kapitel 4 entwickelten Konzept und ist in Abbildung 32 dargestellt. Der Antriebscontroller umfasst FPGA-seitig zwei rekonfigurierbare Module, ein Kommunikationssystem und ein statisches Basissystem, das durch externe Schnittstellenkomponenten ergänzt wird. Das zu regelnde Antriebssystem ist schematisch ebenfalls in Abbildung 32 dargestellt und besteht aus DC-Motor, Inkrementalgeber (Encoder) zur Positions- und Drehzahlerfassung sowie leistungselektronischen Stellgliedern (H-Brücke mit leistungselektronischen Schaltern). Im Anwendungsfall sind zusätzlich Getriebe und mechanische Lasten zu beachten.

Die Kommunikation zwischen den rekonfigurierbaren Modulen und dem statischen Basissystem des Controllers wird mit Bus-Makros und bitseriellen Übertragungsmethoden realisiert. Neben dem dedizierten Kommunikationssystem für rekonfigurationsbezogene Daten nach Abschnitt 4.2.1 werden so Sollwerte, Istwerte und Stellwerte für die Drehzahl des Antriebssystems sowie weitere relevante Größen übertragen. Das Prinzip der bitseriellen Signalübertragung auf Controllerebene wird in Abbildung 33 für Ist- und Stellwerte illustriert. Die eingesetzten Seriell-Parallel- (s2p) und Parallel-Seriell-Wandler (p2s) in den Hardwaretasks werden nicht benötigt, wenn auf bitserielle Algorithmen für die Umsetzung der Steuerungs- und Regelungsfunktionen zurückgegriffen wird.

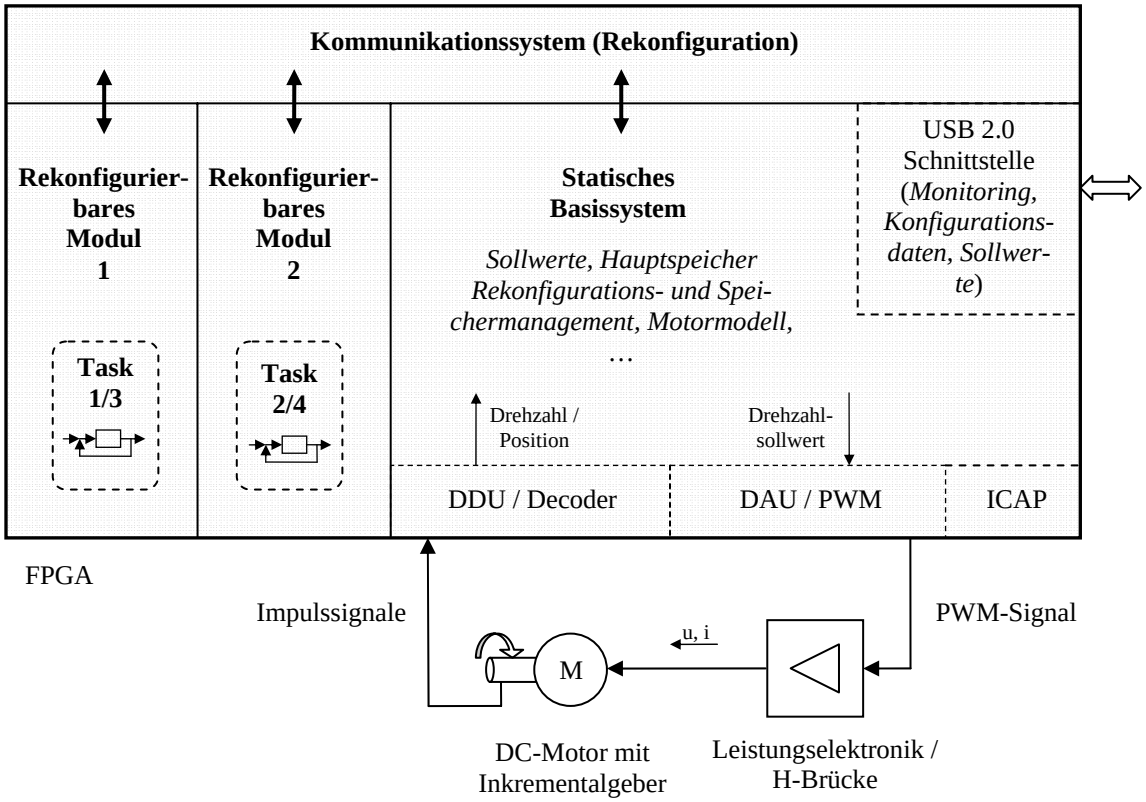


Abbildung 32: Struktur des rekonfigurierbaren Antriebscontrollers

Statisches Basissystem

Das statische Basissystem des rekonfigurierbaren Antriebscontrollers umfasst Kommunikations- und Prozessschnittstellen, Infrastrukturkomponenten und den Hauptspeicher des Systems. Damit realisiert das Basissystem die Anbindung des Controllers an übergeordnete Strukturen sowie das Antriebssystem, verwaltet Konfigurationsdaten und führt die Ladevorgänge für Hardwaretasks aus. Weiterhin kann im Basissystem die Sollwertgenerierung für die Drehzahlregelung implementiert werden. Darüber hinaus wurde im Rahmen der Arbeit ein einfaches synthesefähiges DC-Motormodell eingesetzt, das als Grundlage für die Fehlerdetek-

tion oder einen Beobachter verwendet werden kann. Die Umsetzung im statischen Basissystem ermöglicht den Zugriff auf das Modell in allen Betriebszuständen des Controllers. Das Motormodell kann alternativ direkt in einem Hardwaretask implementiert werden. Das statische Basissystem wurde mit dem HDL Designer von Mentor Graphics spezifiziert und durch eine Codegenerierung in HDL-Quelltext überführt.

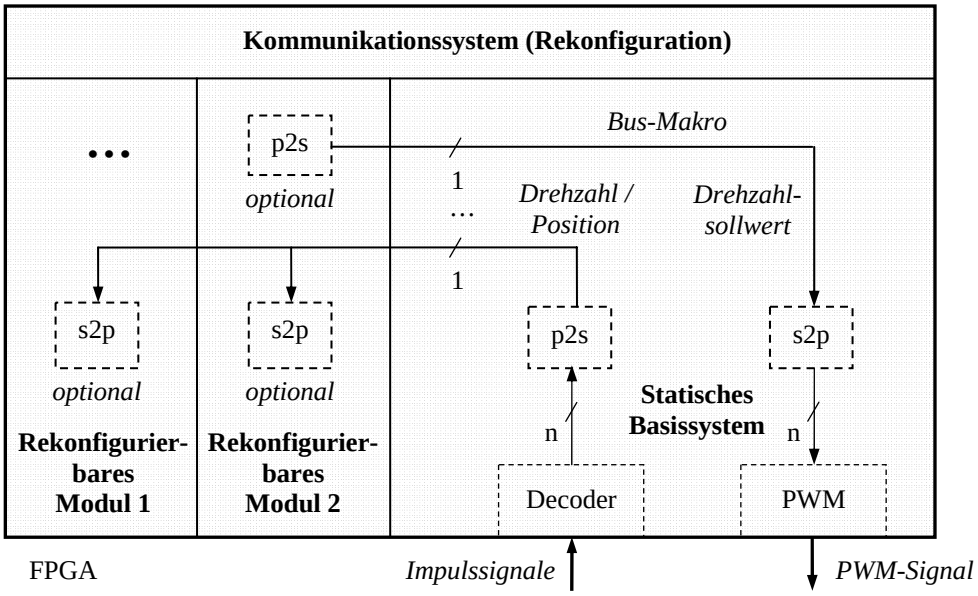


Abbildung 33: Bitserielle Signalübertragung im rekonfigurierbaren Antriebscontroller

Die Kommunikation des Controllers mit übergeordneten Systemen wird mit der USB-2.0-Schnittstelle nach Abschnitt 4.2.4 realisiert. Diese dient hier vorrangig dem Empfang von Konfigurationsdaten und Sollwerten für die Drehzahlregelung, kann aber auch für die Übermittlung einzelner Messwerte eingesetzt werden. Die Anbindung des Controllers an das Antriebssystem erfolgt mit Prozessschnittstellen für die Erfassung der Drehzahl sowie die Ansteuerung der leistungselektronischen Stellglieder. Die Drehzahlerfassung als Vorraussetzung für eine Drehzahlregelung wird dabei durch einen Decoder umgesetzt, der die Impulssignale des Inkrementalgebers auswertet. Grundlage der Auswertung sind direkt in FPGA-Hardware implementierte Zähler, die Impulse in festgelegten Zeitintervallen erfassen und mit der bekannten Impulszahl pro Umdrehung des Gebers in Drehzahlistwerte überführen. Der Decoder legt dabei die Abtastzeit und Verarbeitungswortbreite *n* der nachfolgenden Signalverarbeitung fest. Die Ansteuerung der leistungselektronischen Stellglieder erfolgt über die Aufprägung des Drehzahlstellwertes auf ein pulsweitenmoduliertes Signal (PWM). Grundlage für die PWM-Generierung sind ebenfalls direkt in FPGA-Hardware implementierte Zähler, die das PWM-Signal in Abhängigkeit der Relation zwischen Drehzahlstellwert und Maximaldrehzahl des Antriebssystems schalten. Die Periodendauer des PWM-Signals kann flexibel eingestellt

und so dem anzusteuernden Antriebssystem angepasst werden. Auf der Basis der Taktfrequenz des FPGAs sind hier auch sehr hohe Frequenzen realisierbar.

Die Verwaltung von Konfigurationsdaten und die Ausführung der partiellen Rekonfiguration werden im statischen Basissystem durch das in Abschnitt 4.2.2 vorgestellte Rekonfigurations- und Speichermanagement sowie den integrierten Rekonfigurationsmechanismus nach Abschnitt 5.2.2 umgesetzt. Das Rekonfigurations- und Speichermanagement ist dabei mit der USB-Schnittstelle verbunden und legt so empfangene Konfigurationsdaten im Hauptspeicher des Systems ab. Damit können einzelne Hardwaretasks zur Laufzeit des Controllers auch mehrfach empfangen und modifiziert werden. Die Ausführung der Ladevorgänge für Hardwaretasks erfolgt unter Einbeziehung der internen Konfigurationsschnittstelle ICAP.

Hardwaretasks

Die Hardwaretasks des rekonfigurierbaren Antriebscontrollers umfassen ausgehend von der Modul-FSM nach Abbildung 31 vier Tasks, die nacheinander in die zwei rekonfigurierbaren Module des Controllers geladen werden. Die Tasks stellen elementare Steuer- und Regelfunktionen für den Hochlauf und Betrieb eines Antriebs dar und können auf der Grundlage der Funktionsbibliotheken des Spezifikationsframeworks um zusätzliche Funktionalitäten wie erweiterte Regelstrukturen ergänzt werden. Die nachfolgend vorgestellten Hardwaretasks wurden unter Einbeziehung bitserieller Methoden mit dem HDL Designer von Mentor Graphics spezifiziert und durch eine Codegenerierung in HDL-Quelltext überführt. Die eingesetzten Steuer- und Regelalgorithmen liegen in bitparaleller Form auch für eine Codegenerierung aus Matlab/Simulink heraus vor.

Task 1 des Task-Sets wird während der Inbetriebnahme des Antriebssystems ausgeführt und beinhaltet statische Tests der Controllerfunktionalitäten. Der Schwerpunkt der Tests liegt auf dem bitseriellen Datentransfer zwischen dem statischen Basissystem und den rekonfigurierbaren Modulen. Der Testalgorithmus überprüft dabei die Übertragung von Sollwerten und vom Decoder generierten Istwerten, indem die Startbits der Signale überwacht und auf Synchronität kontrolliert werden. Startbits sind jeder bitseriellen Übertragung vorangestellt und liegen damit auch für das unbewegte Antriebssystem vor. Der Task kann um weitere Testalgorithmen für das statische Systemverhalten erweitert werden. Beispiele hierfür umfassen Tests der Kommunikationschnittstelle und USB-Funktionalität.

Task 2 des Task-Sets wird ebenfalls während der Inbetriebnahme des Antriebssystems ausgeführt und beinhaltet dynamische Tests. Der Task verfügt hierzu über einen Funktionsgenerator und einen Testalgorithmus für den vom Decoder generierten und parallel-seriell gewandelten Istwert der Drehzahl. Der Funktionsgenerator umfasst eine oder mehrere Testsequenzen für den Drehzahlstellwert des Antriebs. Beispielhaft sei eine parametrierbare Rampenfunktion genannt. Mit Hilfe des resultierenden Istwertes der Drehzahl wird die Funktion des Inkrementalgebers und des nachfolgenden Decoder kontrolliert. Eine Erweiterung der Überprüfung um einen Vergleich mit Drehzahlwerten des mitlaufenden einfachen Motormodells im Basisdesign wird ebenso unterstützt. Darüber hinaus können weitere Test der externen Komponenten des Antriebssystems implementiert werden.

Task 3 des Task-Sets stellt den Hauptbetriebszustand des Antriebssystems dar und realisiert eine PI-Drehzahlregelung auf der Basis eines bitseriellen PI-Algorithmus und eines bitseriellen Subtraktionsgliedes. Der PI-Algorithmus setzt sich aus grundlegenden bitseriellen Signalverarbeitungselementen wie Integrator, Addierer und Verstärkungsgliedern zusammen. Die Parameter des PI-Algorithmus K_P und K_I werden außerhalb der Laufzeit des Controllers abhängig vom konkreten DC-Antrieb mit einschlägigen Verfahren bestimmt und in der Spezifikation des Hardwaretasks als Konstanten festgelegt [32]. Eine Veränderung der Parameter kann durch das Laden eines neu parametrierten Hardwaretasks mit PI-Algorithmus realisiert werden. Ebenfalls mittels partieller Rekonfiguration kann die Regelstruktur des Hauptbetriebszustands verändert werden. Die Funktionsbibliotheken stellen hierfür einen PID-Algorithmus und Kaskadenstrukturen zur Verfügung. Die Spezifikation des Controllers ist in diesen Fällen anzupassen und die Modul-FSM um die neuen Hardwaretasks zu erweitern.

Task 4 des Task-Sets stellt eine Sicherung des Antriebssystems im Fehlerfall (Fail-Safe) dar und wird im Fehlerfall aus dem Hauptbetriebszustand heraus aktiviert. Die Fehlerdetektion ist im Hauptbetriebszustand (Task 3) implementiert und basiert auf einem Vergleich zwischen dem Drehzahlistwert und dem Drehzahlwert des mitlaufenden einfachen Motormodells im Basisdesign. Fehlerbedingungen auf der Grundlage von Eingangswerten werden außerhalb der Laufzeit des Controllers festgelegt. Wird eine Fehlerbedingung wie ein plötzlicher Ausfall der Drehzahlwerte detektiert, erfolgt die Aktivierung des Fail-Safe-Tasks. Das Antriebssystem wird daraufhin gestoppt und die Signalverarbeitung neu initialisiert. Der Fail-Safe-Task kann für die Implementierung weiterer Sicherheits- und Diagnosefunktionen genutzt werden.

6.1.3 Implementierung

Der rekonfigurierbare Antriebscontroller wurde mit dem *Module Based Partial Reconfiguration Design Flow* für einen Xilinx Virtex-II XC2V1000 FPGA und mit dem *Early Access Partial Reconfiguration Design Flow* für einen Xilinx Virtex-4 XC4VLX60 FPGA umgesetzt. Referenzversion der eingesetzten Implementierungswerkzeuge ist Xilinx ISE 6.3 respektive Xilinx ISE 9.1. Die Hardwaregrundlage der Implementierung wird durch Entwicklungsboards der Firma Avnet gestellt, die neben dem FPGA auch den EZ-USB FX2 USB-Controller von Cypress umfassen.

Randbedingungen und Parameter

Die rekonfigurierbaren Module des Controllers auf dem Virtex-II XC2V1000 FPGA nehmen die volle Höhe des Bausteins ein und verfügen bei einer Breite von 2 CLBs über elementare Hardwareressourcen von 80 CLBs und damit 320 CLB Slices. Die rekonfigurierbaren Module auf dem Virtex-4 XC4VLX60 FPGA sind dagegen nicht auf die volle Höhe des Bausteins beschränkt und wurden als rechteckige Bereiche von 4 mal 20 CLBs in die CLB-Matrix eingebettet. Damit verfügen diese Module ebenfalls über elementare Hardwareressourcen von 80 CLBs. Die nicht von den Modulen erfassten Bereiche der FPGAs können für das statische Basissystem des Controllers genutzt werden. Der Hauptspeicher des Controllers kann dabei auf dem Virtex-II XC2V1000 FPGA mit bis zu 90 KB Block-RAM implementiert werden, darüber hinaus wird hier externer SRAM eingesetzt. Der Hauptspeicher des Virtex-4 XC4VLX60 FPGAs wird mit 256 KB Block-RAM vollständig auf dem Baustein umgesetzt. Die interne Konfigurationsschnittstelle ICAP und damit auch der Hauptspeicher werden auf dem Virtex-II XC2V1000 FPGA mit 8 Bit und auf dem Virtex-4 XC4VLX60 FPGA mit 32 Bit Datenbusbreite betrieben.

Die Taktfrequenz der FPGAs wird auf den Entwicklungsboards generiert und kann in den Bausteinen mit digitalen Clockmanagern (DCMs) angepasst werden. Der Virtex-II XC2V100 FPGA wird so mit einer Taktfrequenz von 40 MHz betrieben, der Virtex-4 XC4VLX60 FPGA mit 100 MHz. Davon ausgehend wird die interne Konfigurationsschnittstelle des Virtex-II FPGAs von der Rekonf-FSM mit 20 MHz angesteuert. Die Maximalfrequenz von 50 MHz kann über den Einsatz eines digitalen Clockmanagers erreicht werden. Dies wurde für den Virtex-4 FPGA standardmäßig umgesetzt: hier wird die Rekonf-FSM durch einen DCM mit 200 MHz getaktet und schreibt so mit der Maximalfrequenz von 100 MHz Konfi-

gurationsdaten an ICAP. Darüber hinaus erzeugt der DCM einen 40 MHz Takt, mit dem aus Kompatibilitätsgründen das Basissystem und die Hardwaretasks getaktet werden können.

Verarbeitungswortbreite und Abtastzeit der Steuer- und Regelalgorithmen des Controllers wurden für die Beispielimplementierung mit 16 Bit inklusive bitseriellem Startbit und 1 ms gewählt. Die Auflösung der Drehzahlwerte beträgt damit 15 Bit. Abtastzeiten von 100 µs und kleiner wurden ebenfalls getestet. Beide Werte werden in der Spezifikation des Decoders zugewiesen und auf nachfolgenden Steuer- und Regelalgorithmen bezogen. Alle Prozessgrößen werden controllerintern skaliert und innerhalb der Verarbeitungswortbreite als ganzzahlige Werte abgebildet. Die Reglerparameter K_P und K_I des Drehzahlreglers werden abhängig vom Antriebssystem in der Spezifikation der Hardwaretasks festgelegt und wurden hier exemplarisch mit 5 Bit aufgelöst. Auf der Grundlage skalierender bitserieller Verstärker nach [65] wurden nicht ganzzahlige Verstärkungsfaktoren realisiert.

Ergebnisse

Der Ressourcenverbrauch des implementierten Antriebscontrollers ist für beide gewählte Ziel-FPGAs aufgrund der ähnlichen Hardwarestruktur und der nahen Verwandtschaft der Implementierungsschritte vergleichbar. Da die grundlegenden Register und Funktionsgeneratoren durch die Implementierungswerkzeuge verteilt auf der Zielhardware platziert werden, weicht die Anzahl belegter CLB Slices in den nachfolgenden Ergebnissen vom Idealwert ab.

So kann der Hardwaretasks für statische Tests mit rund 120 CLB Slices (120 DFFs, 170 LUTs) und der Hardwaretasks für dynamische Tests mit rund 270 CLB Slices (275 DFFs, 430 LUTs) umgesetzt werden. Der Hardwaretask für den Hauptbetriebszustand des Controllers mit bitseriellem Subtraktionsglied und PI-Regler erfordert ebenfalls etwa 120 CLB Slices (135 DFFs, 140 LUTs), während der Hardwaretasks für den Fail-Safe-Zustand mit etwa 165 CLB Slices (185 DFFs, 195 LUTs) implementiert werden kann. Der Ressourcenbedarf des statischen Basissystems liegt bei rund 780 CLB Slices (770 DFFs, 1090 LUTs), zwei dedizierten RAM-Blöcken und einem dedizierten Multiplizier für das Motormodell. Wird der Hauptspeicher des Controllers direkt im Basissystem implementiert, erhöht sich der Ressourcenbedarf im Fall des 256 KB Speichers für den Virtex-4 XC4VLX60 FPGA um 116 dedizierte RAM-Blöcke und 435 CLB Slices (20 DFFs, 830 LUTs). Der Teilnehmeranschluss an das Kommunikationssystem belegt in den Hardwaretasks und dem Basissystem jeweils rund 20 CLB Slices.

Die erzielbaren Rekonfigurationszeiten des Antriebscontrollers sind für die eingesetzten FPGA-Typen aufgrund von Hardwareeigenschaften und Implementierungsmethoden deutlich verschieden. Die Konfigurationsdaten der Hardwaretasks des Virtex-II XC2V1000 FPGAs wurden ohne eine Komprimierung durch das Multiple-Frame-Write-Verfahren erzeugt und umfassen konstant 19580 Bytes. Diese können bei einem Systemtakt von 40 MHz mit 20 MHz byteweise an die interne Konfigurationsschnittstelle geschrieben werden. Damit ergibt sich eine Rekonfigurationszeit von 979 µs. Die Konfigurationsdaten der Hardwaretasks des Virtex-4 XC4VLX60 FPGA wurden mit einer Komprimierung durch das Multiple-Frame-Write-Verfahren erzeugt und umfassen so zwischen 35784 und 41180 Bytes. Bei einer Frequenz von 100 MHz für das Schreiben von Konfigurationsdaten an ICAP im 32 Bit Modus ergeben sich damit Rekonfigurationszeiten zwischen 89,46 µs und 102,95 µs. Der Mehrbedarf an Konfigurationsdaten gegenüber den unkomprimierten Hardwaretasks des Virtex-II FPGAs beruht auf der Konfigurationsarchitektur des Virtex-4 FPGAs sowie der Anwendung des EAPR Design Flows. So werden im EAPR Design Flow Signale, die rekonfigurierbare Bereiche durchqueren, unterstützt und in den erzeugten Konfigurationsdaten berücksichtigt.

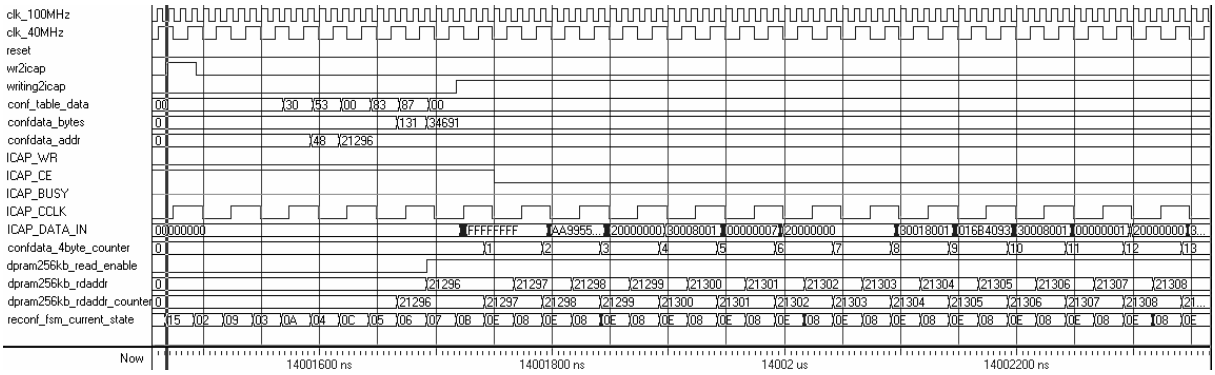


Abbildung 34: Rekonfigurationsvorgang des Antriebscontrollers (Timingsimulation, ModelSim)

6.1.4 Verifikation

Die Verifikation des rekonfigurierbaren Antriebscontrollers wurde entwurfsbegleitend mit Verhaltenssimulationen des generierten HDL-Quelltextes vorgenommen und nach erfolgter Implementierung durch Timingsimulationen ergänzt. Grundlage beider Verifikationsmethoden sind zugeschnittene Testumgebungen mit nicht synthesefähigen Verhaltensmodellen von FPGA-externen Bestandteilen des Antriebssystems. Die Funktionsbibliotheken des Spezifikationsframeworks wurden hierzu um VHDL-basierte Verhaltensmodelle von Digital/Analog-Wandlung, DC-Motor (Verzögerungsglied 1. und 2. Ordnung) und Inkrementalgeber ergänzt.

Weiterhin wurde der rekonfigurierbare Antriebscontroller mit einem realen DC-Kleinantrieb getestet und erfolgreich in Betrieb genommen. Eingesetzt wurde der DC-Kleinmotor 3557 K012 CS mit Planetengetriebe G38/1S der Firma Faulhaber. Als Encoder wurde der optische Impulsgeber HEDS5500A ebenfalls von Faulhaber verwendet. Der Geber verfügt über 2 Kanäle und eine Auflösung von 500 Impulsen pro Umdrehung. Die leistungselektronische Ansteuerung des Antriebs wurde mit dem Motortreiberbaustein VNH2SP30-E von STMicroelectronics umgesetzt. Der Treiberbaustein für Anwendungen im automotiven Bereich und der Robotik integriert leistungselektronische Stellglieder in Form einer H-Brücke und besitzt einen dedizierten PWM-Eingang. Die Auslegung des digitalen Drehzahlreglers für das Antriebssystem wurde quasikontinuierlich vorgenommen und die Frequenz des PWM-Signals auf rund 20 kHz eingestellt. Mit diesen Daten wurden auch die Modelle und Komponenten der Verhaltens- und Timingsimulationen parametriert.

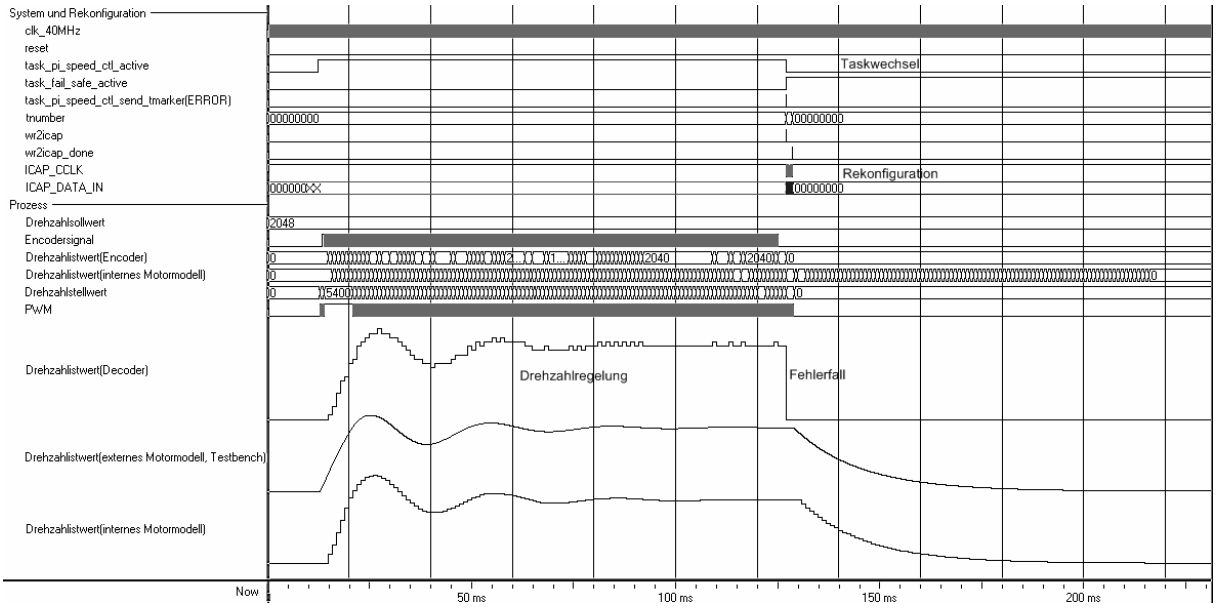


Abbildung 35: Taskwechsel des Antriebscontrollers mit Rekonfigurationsvorgang und Prozessgrößen (Verhaltenssimulation, ModelSim)

Im Ergebnis der Verifikation konnte mit den Verhaltens- und Timingsimulationen die Funktion des Controllers bitgenau in Bezug auf die partielle Rekonfiguration (Laden und Aktivieren von Hardwaretasks), Speichermanagement, Schnittstellen und Kommunikation nachgewiesen werden. Abbildung 34 zeigt den Beginn eines Rekonfigurationsvorgangs in ModelSim im Rahmen einer Timingsimulation. Für eine bessere Darstellbarkeit wurde hier und in den folgenden Abbildungen ICAP im 32 Bit Modus mit einer Rekonfigurationsfrequenz von 20 MHz betrieben. Mit den Verhaltensmodellen der externen Controllerbestandteile sowie des Antriebssystems konnte darüber hinaus die Funktion der Hardwaretasks und die exemplarische

regelungstechnische Auslegung des Controllers in einem größerem Zeithorizont simulativ untersucht und belegt werden. So zeigt Abbildung 35 die Verhaltenssimulation eines Taskwechsels inklusive Rekonfigurationsvorgang und Prozessgrößen.

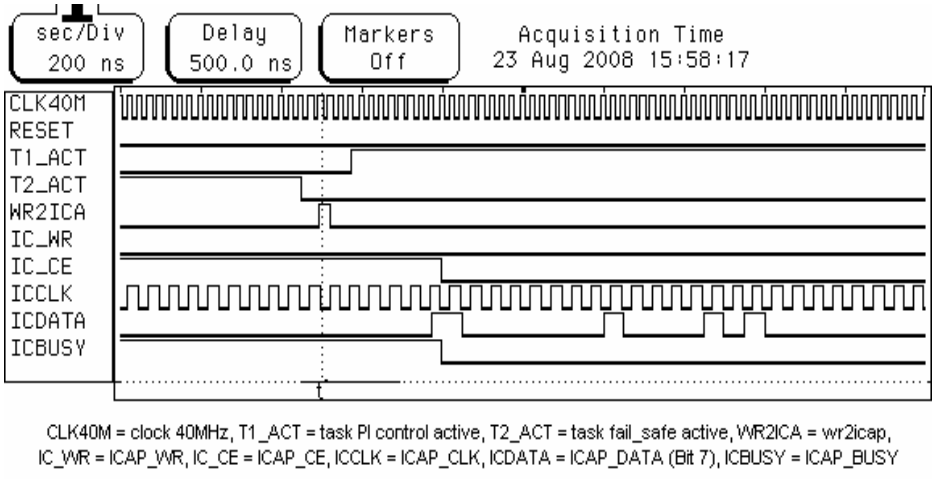


Abbildung 36: Taskwechsel des Antriebscontrollers mit Rekonfigurationsvorgang (Messung, Logikanalysator)

Auf der Grundlage des DC-Kleinantriebs und der Avnet FPGA Boards wurde weiterhin mit einem Logikanalysator die digitale Funktionalität des Antriebscontrollers im realen Betrieb überprüft. Hierbei konnte die echtzeitfähige Ausführung der Lade- und Aktivierungsvorgänge für Hardwaretasks nachgewiesen werden. Abbildung 36 stellt einen mit dem Logikanalysator erfassten Taskwechsel und den Beginn des Rekonfigurationsvorgangs dar. Der Taskwechsel und zugehörige Prozesssignale auf Funktionsebene wurden zusätzlich mit einem Oszilloskop aufgezeichnet (Abbildung 37).

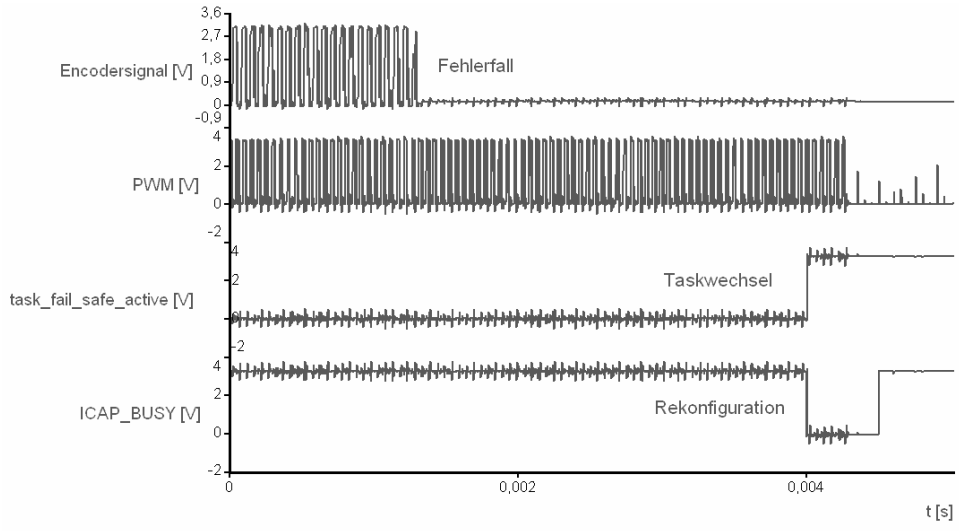


Abbildung 37: Taskwechsel des Antriebscontrollers mit Rekonfigurationsvorgang (Messung, Oszilloskop)

6.2 Rekonfigurierbarer Controller für piezo-elektrische Aktoren

Der rekonfigurierbare Controller für die Ansteuerung piezo-elektrischer Aktoren basiert auf einer in [87] vorgestellten Leistungsverstärkertopologie und einem FPGA für die Umsetzung aktornaher Steuerungs- und Regelungsalgorithmen sowie Diagnosefunktionen. Abbildung 38 zeigt die Leistungsverstärkertopologie nach [87]. Der Aktor wird hier über die Betätigung der leistungselektronischen Schalter spannungs- und stromgeregelt aus Speicherkondensatoren geladen und in diese wieder entladen. Die Versorgungsspannung der Kondensatoren wird durch einen DC/DC-Wandler bereitgestellt und ist größer oder kleiner als die positive respektive negative Maximalspannung des Aktors. Auf diese Weise können sehr schnelle Anstiegszeiten der Aktorspannung erzielt und der Aktor dynamisch geladen und entladen werden. Die in Reihe zum Aktor geschaltete Spule dient als temporärer Energiespeicher und wirkt strombegrenzend. Der ebenfalls in Reihe geschaltete Widerstand wird für die Strommessung eingesetzt.

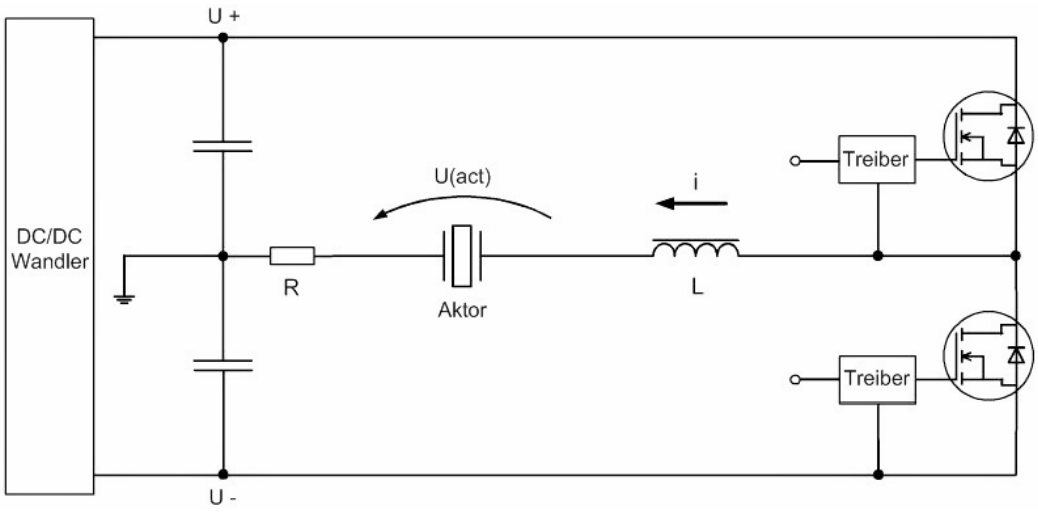


Abbildung 38: Leistungsverstärkertopologie für die Ansteuerung piezo-elektrischer Aktoren nach [87]

Die Spannungs- und Stromregelung des Aktors wird mit Messungen der Aktorspannung und des Aktorstroms realisiert. Diese Größen werden in aufbereiteter Form dem FPGA-basierten Controller übermittelt und dort durch geeignete Regelalgorithmen in Ansteuersignale für die leistungselektronischen Schalter umgesetzt. Für den hier entworfenen Controller werden Zweipunktregler eingesetzt, die auf einer übergeordneten Ebene aus Ist- und Sollwert der Aktorspannung ein Signal für Laden oder Entladen generieren und dann auf einer untergeordneten Ebene die leistungselektronischen Schalter abhängig vom Strom öffnen und schließen. Beide Regelvorgänge sind mit einer Hysterese in Bezug auf die Regelgrößen versehen.

Der rekonfigurierbare Controller für die Ansteuerung piezo-elektrischer Aktoren stellt eine beispielhafte Umsetzung von FPGA-seitigen Steuerungs-, Regelungs- und Diagnosefunktionen für die Leistungsverstärkertopologie nach [87] dar. Zusätzlich können weitere Funktionen wie die Regelung der Speicherkondensatorspannung über den DC/DC-Wandler FPGA-seitig implementiert werden. Die komplexe FPGA-externe Umsetzung der Piezo-Ansteuerung wird als gegeben vorausgesetzt.

Einsatzgebiet der grundlegenden Leistungsverstärkertopologie ist unter anderem die Kraftstoffeinspritzung in Verbrennungsmotoren mit piezo-basierten Injektoren. Für weiterführende Informationen zu Implementierung und Dimensionierung des Leistungsverstärkers sei auf [87] verwiesen. Entwurf, Auslegung und Beispielumsetzungen von Systemen mit piezo-elektrischem Aktor, Leistungsverstärker und FPGA-basierter Ansteuerung werden darüber hinaus in [88] beschrieben. Die Grundlagen und Einsatzmöglichkeiten piezo-elektrischer Aktoren werden in [89] und [90] dargestellt.

6.2.1 Spezifikation

Ausgangspunkt der Spezifikation des rekonfigurierbaren Controllers für piezo-elektrische Aktoren ist die logische Controllerstruktur. Diese wird angelehnt an den Antriebscontroller nach Abschnitt 6.1 festgelegt und basiert auf dem Zustandsautomaten für Hochlauf und Betrieb mechatronischer Systeme nach Abbildung 8 (Abschnitt 3.1.3). Die so beschriebenen Controllerfunktionalitäten werden mit einer Partitionierung in statische und rekonfigurierbare Bestandteile zerlegt. Schnittstellen, Infrastrukturkomponenten sowie die aktornahe Stromregelung werden dabei den statischen Bestandteilen zugeordnet. Die rekonfigurierbaren Controllerkomponenten umfassen Diagnosefunktionen sowie Steuer- und Regelalgorithmen für die Aktorspannung.

Die Modul-FSM des Controllers wird ebenfalls analog zum Antriebscontroller nach Abschnitt 6.1 spezifiziert und implementiert die Vorablade- und Aktivierungsstrategie nach Abschnitt 3.3.3. Damit sind zwei rekonfigurierbare Module auf der Zielplattform erforderlich, die zu einem Supermodul zusammengefasst werden. Das Task-Set umfasst hier Hardwaretasks für System- und Aktortests, die Spannungsregelung als Hauptbetriebszustand sowie einen sicheren Systemzustand im Fehlerfall. Abbildung 39 zeigt die Modul-FSM des rekonfigurierbaren Controllers für piezo-elektrische Aktoren. Eine Erweiterung der Modul-FSM um zusätzliche Hardwaretasks ist unter Beachtung vorhergehender Entwurfschritte möglich.

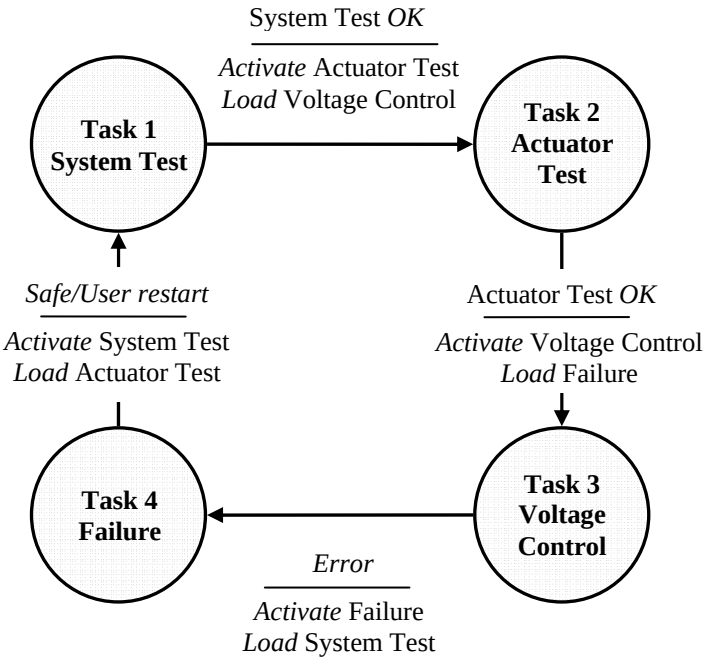


Abbildung 39: Modul-FSM des rekonfigurierbaren Controllers für piezo-elektrische Aktoren

6.2.2 Struktur und Komponenten

Die Struktur des rekonfigurierbaren Controllers für piezo-elektrische Aktoren umfasst FPGA-seitig zwei rekonfigurierbare Module, ein statisches Basissystem und ein Kommunikationssystem, das Module und Basissystem verbindet. Sie wird durch FPGA-externe Schnittstellenkomponenten ergänzt und ist in Abbildung 40 dargestellt. Der Controller generiert die Ansteuersignale *Gate_1* und *Gate_2* für die leistungselektronischen Schalter der externen Verstärkerschaltung und steuert damit den Energiefluss in den angeschlossenen piezo-elektrischen Aktor. Die resultierende Aktorspannung (U_{act_ist}) wird über Spannungsteiler und Analog/Digital-Umsetzer in digitalisierter Form an den Controller übermittelt. Der Aktorstrom wird mittels Messwiderstand in eine Spannung überführt und daraufhin mit Spannungsteilern und Analog-Komparatoren gegen externe Referenzwerte verglichen. Auf diese Weise werden vier logische Signale generiert, die den Zustand des Aktorstroms abbilden. I_{up} steht hier für ein Überschreiten des Referenzwertes, I_{down} für ein Unterschreiten. I_{up} und I_{down} werden für den positiven (I_{up}/I_{down} *Gate 1*) und negativen Aktorstrom (I_{up}/I_{down} *Gate 2*) erfasst und an den Controller übermittelt.

Die Datenübertragung zwischen den rekonfigurierbaren Modulen des Controllers und dem statischen Basissystem wurde mit Ausnahme des dedizierten Kommunikationssystems für rekonfigurationsbezogene Daten und logischer Signale bitparallel umgesetzt. Die Hardware-

grundlage für die Datenübertragung sind auch hier Bus-Makros. Das Prinzip der controllerinternen Kommunikation wird ebenfalls in Abbildung 40 illustriert.

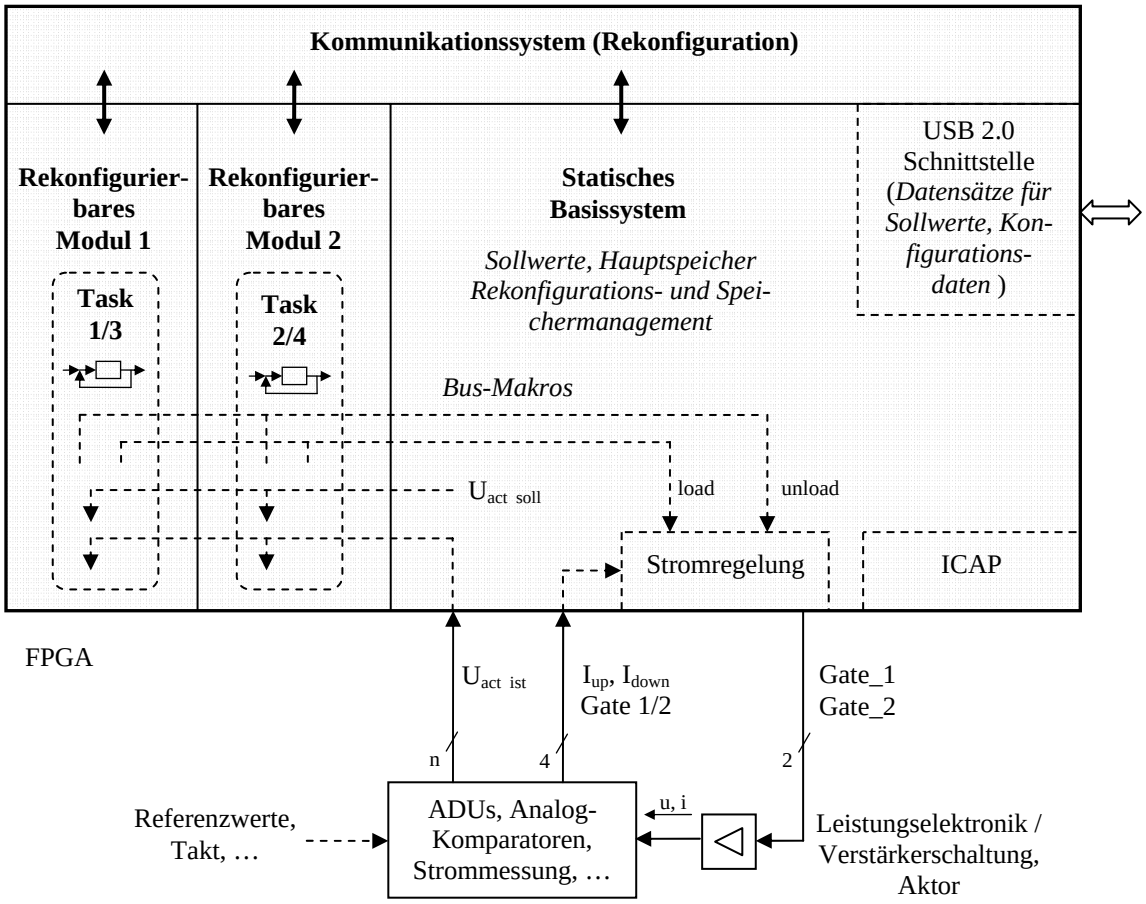


Abbildung 40: Struktur des rekonfigurierbaren Controllers für piezo-elektrische Aktoren

Statisches Basissystem

Das statische Basissystem implementiert Schnittstellen, Infrastrukturkomponenten und den Hauptspeicher des Controllers. Die Kommunikation mit übergeordneten Strukturen wird mit der USB-2.0-Schnittstelle nach Abschnitt 4.2.4 umgesetzt, die hier insbesondere dem Empfang von Konfigurationsdaten und Sollwertdatensätze für die Aktorspannung dient. Für Diagnosezwecke können Prozessgrößen über die USB-Schnittstelle an übergeordnete Strukturen übermittelt werden. Die Anbindung der Spannungs- und Stromwerte des Aktors an den Controller erfolgt digital. So werden die Spannungswerte von externen Analog/Digital-Umsetzern bereitgestellt, während der Aktorstrom über die binären Signale der Analog-Komparatoren erfasst wird. Die Ansteuerung der leistungselektronischen Schalter erfolgt ebenfalls diskret mit binären Stellsignalen. Die Infrastrukturkomponenten des Controllers werden vergleichbar

zum Antriebscontroller nach Abschnitt 6.1 durch das Rekonfigurations- und Speichermanagement nach Abschnitt 4.2.2 gestellt.

Darüber hinaus implementiert das statische Basissystem des Controllers die aktornahe Stromregelung. Diese ist als Zweipunktregler ausgeführt und basiert auf in [87] und [88] vorgestellten Methoden. Die Stromregelung wertet die von der übergeordneten Spannungssteuerung/-regelung generierten Signale für Laden (*load*) oder Entladen (*unload*) des Aktors aus und setzt in Abhängigkeit vom Aktorstrom die diskreten Ansteuersignale für den Leistungsverstärker. Die Ausgangssignale der Analog-Komparatoren können im FPGA digital gefiltert werden, um den Einfluss hochfrequenter Störungen zu verringern. Schaltvorgänge werden vorgenommen, wenn der Aktorstrom bei gesetztem Lade-/Entladesignal einen durch die Referenzwerte der Analog-Komparatoren vorgegebenen Wert über- oder unterschreitet. Die Schaltfrequenz richtet sich nach der Dimensionierung des Leistungsverstärkers und Aktors.

Das statische Basissystem wurde mit dem HDL Designer von Mentor Graphics spezifiziert und in HDL-Quelltext überführt. Grundlage sind die Funktionsbibliotheken des Spezifikationsframeworks sowie bestehende Implementierungen nach [87] und [88].

Hardwaretasks

Die rekonfigurierbaren Komponenten des Controllers umfassen vier durch die Modul-FSM repräsentierte Hardwaretasks. Die Tasks werden mittels partieller Rekonfiguration in die rekonfigurierbaren Module des Controllers geladen und stellen beispielhafte Diagnose-, Steuer- und Regelfunktionen für piezo-elektrische Aktoren auf der Grundlage der angeführten Leistungsverstärkertopologie dar. Der Entwurf der Hardwaretasks wurde ebenfalls mit dem HDL Designer von Mentor Graphics vorgenommen und basiert auf bestehenden Implementierungen nach [87] und [88] sowie den Funktionsbibliotheken des Spezifikationsframeworks. Dabei wurden vorrangig bitparallele Methoden eingesetzt und durch eine Codegenerierung in HDL-Quelltext überführt. Die Hardwaretasks sind auf dieser Grundlage flexibel erweiterbar. Der Algorithmus für die Regelung der Aktorspannung wurde im Zusammenhang mit der aktornahen Stromregelung auch für eine Codegenerierung aus Matlab/Simulink spezifiziert.

Task 1 des Task-Sets führt während des Systemstarts einfache Tests des Gesamtsystems aus. So wird beispielhaft der Initialwert der Aktorspannung überprüft, indem digitalisierte Messwerte auf Einhaltung von Ober- und Untergrenzen kontrolliert werden. Die Signale für Laden (*load*) und Entladen (*unload*) werden hier nicht gesetzt. Der Hardwaretask ist um weitere sys-

tembezogene Testalgorithmen erweiterbar. Diese sind für die konkrete Anwendung auszulegen und können Tests der Analog/Digital-Umsetzung der Aktorspannung, der Anbindung und Referenzwerte der Analog-Komparatoren sowie der internen und externen Kommunikation umfassen. Darüber hinaus können die Spannungsbereiche der Speicherkondensatoren überprüft und über den grundlegenden DC/DC-Wandler angepasst werden.

Task 2 des Task-Sets wird nach den Systemtests aktiviert und beinhaltet Tests des Aktors. Mit einem einfachen Testalgorithmus wird der Aktor spannungsgesteuert und stromgeregelt geladen und entladen. Die resultierende Aktorspannung wird dabei erfasst und mit gegebenen Ober- und Untergrenzen abgeglichen. Der Hardwaretask ist um weitere aktorbezogene Testalgorithmen erweiterbar. Beispiele umfassen die Parametrierung der Hysterese von Strom- und Spannungsregelung, die Ermittlung von Spannungsgrenzen des angeschlossenen Aktors über eine Detektion mechanischen Aufsetzens und die daraus abgeleitete Anpassung von Sollwertdatensätzen.

Task 3 des Task-Sets repräsentiert den Hauptbetriebszustand des Controllers und realisiert die Spannungsregelung des angeschlossenen Aktors. Die Spannungsregelung basiert auf einem Vergleich der digitalisierten Istwerte der Aktorspannung mit Sollwerten, die im statischen Basissystem generiert oder empfangen wurden. Die Istwerte werden hierzu mit einer konstanten Hysterese beaufschlagt, um Schwingungen zu vermeiden. Der direkte Vergleich der Spannungswerte wird mit digitalen Komparatoren vorgenommen, die so als Zweipunktregler die Signale für Laden (*load*) und Entladen (*unload*) des Aktors erzeugen. Der Wechsel zwischen Laden und Entladen kann zusätzlich mit einer einstellbaren Verzögerung versehen werden. Der Hardwaretask ist um weitere Funktionen für den Hauptbetriebszustand erweiterbar.

Task 4 des Task-Sets wird im Fehlerfall aktiviert und stellt einen sicheren Systemzustand her. Die Fehlererkennung ist im Hauptbetriebszustand implementiert und detektiert beispielhaft das Überschreiten von voreingestellten Spannungsgrenzen. Der Aktor wird daraufhin durch den aktivierten sicheren Hardwaretask spannungs- und stromgeregelt in einen spannungsneutralen Zustand gebracht. Die Dynamik dieses Vorgangs ist über die Parametrierung der Spannungs- und Stromregelung beeinflussbar. Der Hardwaretask kann zusätzlich um Funktionen für einen sicheren Betrieb des Aktors erweitert werden. Dafür sind im Fehlerfall die Sollwerte der Aktorspannung innerhalb des Tasks zu modifizieren sowie die Hystereseparameter und Schaltzeiten der Spannungs- und Stromregelung anzupassen.

6.2.3 Implementierung

Der rekonfigurierbare Controller für piezo-elektrische Aktoren wurde mit dem *Early Access Partial Reconfiguration Design Flow* für einen Xilinx Virtex-4 XC4VLX60 FPGA implementiert. Dabei kam die Version 9.1 der Xilinx ISE Implementierungswerkzeuge zum Einsatz. Die Hardwaregrundlage der Implementierung ist ein Virtex-4 Entwicklungsboard der Firma Avnet, das den Xilinx Virtex-4 XC4VLX60 FPGA und den EZ-USB FX2 USB-Controller von Cypress umfasst.

Randbedingungen und Parameter

Die rekonfigurierbaren Module des Controllers sind angelehnt an den Antriebscontroller nach Abschnitt 6.1 als rechteckige Bereiche von 4 mal 20 CLBs in die CLB-Matrix des Virtex-4 XC4VLX60 FPGAs integriert. Damit stehen für die Implementierung der Hardwaretasks je 80 CLBs mit 320 CLB Slices zur Verfügung. Das statische Basissystem nimmt den verbleibenden Teil des FPGAs ein. Der Hauptspeicher des Controllers wird als Bestandteil des Basissystems mit einer Speicherkapazität 256 KB und einem 32 Bit breiten Datenbus direkt auf dem FPGA umgesetzt. Die interne Konfigurationsschnittstelle ICAP wird ebenfalls mit einem 32 Bit breiten Datenbus betrieben.

Der Xilinx Virtex-4 XC4VLX60 FPGA wird auf dem Entwicklungsboard mit einer Taktfrequenz von 100 MHz angesteuert. FPGA-intern wird der Takt mit einem digitalen Clockmanager in 40 MHz für den allgemeinen Betrieb des Controllers und 200 MHz für die Ausführung der partiellen Rekonfiguration umgewandelt. Damit können Konfigurationsdaten mit der Maximalfrequenz von 100 MHz an die interne Konfigurationsschnittstelle ICAP geschrieben werden.

Die Verarbeitungswortbreite der bitparallelen Signalverarbeitung wurde für die Implementierung beispielhaft auf 8 Bit festgelegt. Dies betrifft vorrangig die Verarbeitung der digitalisierten Spannungswerte des Aktors. Eine Anpassung der Verarbeitungswortbreite kann während der Spezifikation des Controllers erfolgen. Prozessignale wie die Aktorspannung werden controllerextern auf die Verarbeitungswortbreite skaliert und für die interne Signalverarbeitung ganzzahlig interpretiert. Weitere aktor- und schaltungsbezogene Randbedingungen wurden aus [87] übernommen und Parameter der Steuer- und Regelalgorithmen wie Hysterese, Schaltzeiten und Begrenzungen danach eingestellt.

Ergebnisse

Das statische Basissystem des Controllers kann mit einem Ressourcenverbrauch von rund 915 CLB Slices (460 DFFs, 1520 LUTs) und 118 dedizierten RAM-Blöcken umgesetzt werden. Darin enthalten sind 435 CLB Slices (20 DFFs, 830 LUTs) für die Ansteuerung des Hauptspeichers und 116 dedizierte RAM-Blöcke für dessen Implementierung. Der Hardwaretask für die Systemtests kann mit rund 75 CLB Slices (75 DFFs, 120 LUTs) und der Hardwaretask für die Aktortests mit rund 85 CLB Slices (80 DFFs, 135 LUTs) umgesetzt werden. Der Hardwaretask für den Hauptbetriebszustand des Controllers mit der Spannungsregelung des Aktors erfordert etwa 70 CLB Slices (60 DFFs, 105 LUTs), während der Hardwaretask für den sicheren Systemzustand mit etwa 90 CLB Slices (80 CLBs, 145 LUTs) implementiert werden kann. Der Teilnehmeranschluss an das Kommunikationssystem belegt in Hardwaretasks und Basissystem jeweils rund 20 CLB Slices.

Die Rekonfigurationszeiten des Controllers für piezo-elektrische Aktoren liegen leicht über denen des Antriebscontrollers nach Abschnitt 6.1, da die Konfigurationsdaten der Hardwaretasks aufgrund der größeren Anzahl eingesetzter Bus-Makros etwas umfangreicher ausfallen. So umfassen die unter Verwendung der Multiple-Frame-Write-Option erzeugten Konfigurationsdaten zwischen 36233 und 44486 Bytes. Mit einer Frequenz von 100 MHz für das Schreiben von Konfigurationsdaten an ICAP im 32 Bit Modus sind Rekonfigurationszeiten zwischen 90,58 µs und 111,22 µs erzielbar. Die Rekonfigurationszeiten können durch die Verkleinerung der rekonfigurierbaren Module auf der Zielhardware noch weiter gesenkt werden. Davon wurde hier jedoch Abstand genommen, um die Erweiterung der in den Hardwaretasks implementierten Funktionalitäten ohne erneute Skalierung der Hardwareressourcen zu ermöglichen.

Neben der Implementierung des Controllers mit zwei rekonfigurierbaren Modulen und der Vorablade- und Aktivierungsstrategie nach Abschnitt 3.3.3 wurde auch eine Implementierung mit nur einem Modul und einer direkten Aktivierung der Tasks vorgenommen. Die Hardwaretasks werden dabei sukzessive in das Modul geladen und aktiviert, ein Vorabladen findet nicht statt. Der Ressourcenverbrauch dieser Implementierung liegt leicht unterhalb der oben angeführten Werte, da das Kommunikationssystem und dessen Teilnehmeranschlüsse hier stark vereinfacht sind. Der Umfang der Konfigurationsdaten der einzelnen Hardwaretasks ist ebenfalls etwas geringer. Ursache hierfür ist die kleinere Zahl von Bus-Makros und den rekonfigurierbaren Bereich durchquerenden Signalen sowie die Komprimierung durch das Mul-

tiple-Frame-Write-Verfahren. Die Rekonfigurationszeiten der Implementierung des Controllers mit nur einem Modul liegen damit zwischen 77,27 µs und 82,56 µs.

6.2.4 Verifikation

Die Verifikation des rekonfigurierbaren Controllers für piezo-elektrische Aktoren wurde mit Verhaltenssimulationen des generierten HDL-Quelltextes und Timingsimulationen des implementierten Systems durchgeführt. Die Simulationen basieren auf zugeschnittenen Testumgebungen, in denen der Controller instanziiert und um nicht synthesefähige Verhaltensmodelle FPGA-externer Komponenten ergänzt wird. Hierzu wurden die Funktionsbibliotheken des Spezifikationsframeworks um VHDL-basierte Verhaltensmodelle des Leistungsverstärkers, eines piezo-elektrischen Aktors sowie der Strom- und Spannungserfassung durch Analog-Komparatoren und Analog/Digital-Wandler erweitert.

Das Verhaltensmodell des Leistungsverstärkers realisiert dabei eine einfache Zuordnung der Versorgungsspannung zum Aktor in Abhängigkeit von Aktorstrom und Ansteuersignalen. Die Analog-Komparatoren und Analog/Digital-Wandler wurden dagegen mit Vergleichsoperationen, Skalierungen und Datentypkonvertierungen umgesetzt. Da hier nicht synthesefähige VHDL verwendet wird, können Zeitinformationen und zeitabhängige Effekte wie Schaltzeiten direkt in die Verhaltensmodelle integriert werden. Das Verhaltensmodell des piezo-elektrischen Aktors wurde nach einer in [91] beschriebenen technischen Modellierung erstellt und mit nicht synthesefähigen Integrationsalgorithmen in VHDL implementiert. Der elektrische Teil des Aktors ist abweichend von [91] zusammen mit dem Messwiderstand und der strombegrenzenden Induktivität des Leistungsverstärkers als einfache RLC-Reihenschaltung modelliert. Grundlage ist ein entsprechend parametriertes Verzögerungsglied zweiter Ordnung. Der mechanische Teil des Aktors wurde ebenfalls mit einem Verzögerungsglied zweiter Ordnung beschrieben, das hier auf der Bewegungsgleichung eines elastischen Körpers basiert. Die Kopplung des elektrischen und mechanischen Teils erfolgt über den Piezoeffekt und den inversen Piezoeffekt. Hystereseeffekte werden dabei nicht berücksichtigt, können aber im Rahmen einer erweiterten Modellierung hinzugefügt werden. Die Parametrierung des Aktormodells und des Verstärkers folgt den Angaben in [87].

Mit den Verhaltens- und Timingsimulationen konnte die Funktionalität des Controllers auf digitaler Ebene bitgenau in Bezug auf die partielle Rekonfiguration (Laden und Aktivieren von Hardwaretasks), Speichermanagement, Schnittstellen und Kommunikation nachgewiesen werden. Das Aktormodell sowie die Verhaltensmodelle der weiteren externen Komponenten

wurden für eine Überprüfung der aktorbezogenen Steuer- und Regelfunktionalität des Controllers eingesetzt. Abbildung 41 zeigt die Verhaltenssimulation eines Taskwechsels inklusive Rekonfigurationsvorgang und Prozessgrößen. Im Vergleich zu den in [87] und [88] beschriebenen Ergebnissen aus Mixed-Mode-Simulationen (Analog- und Digitaldomäne) mit Berücksichtigung von Hystereseeffekten sind die hier erzielbaren Resultate jedoch deutlich vereinfacht. Daher sind für eine Inbetriebnahme des Controllers mit einem piezo-elektrischen Aktor und entsprechenden externen Komponenten weitere Untersuchungen der analogen Bestandteile der Ansteuerung sowie des Aktors vorbereitend vorzunehmen.

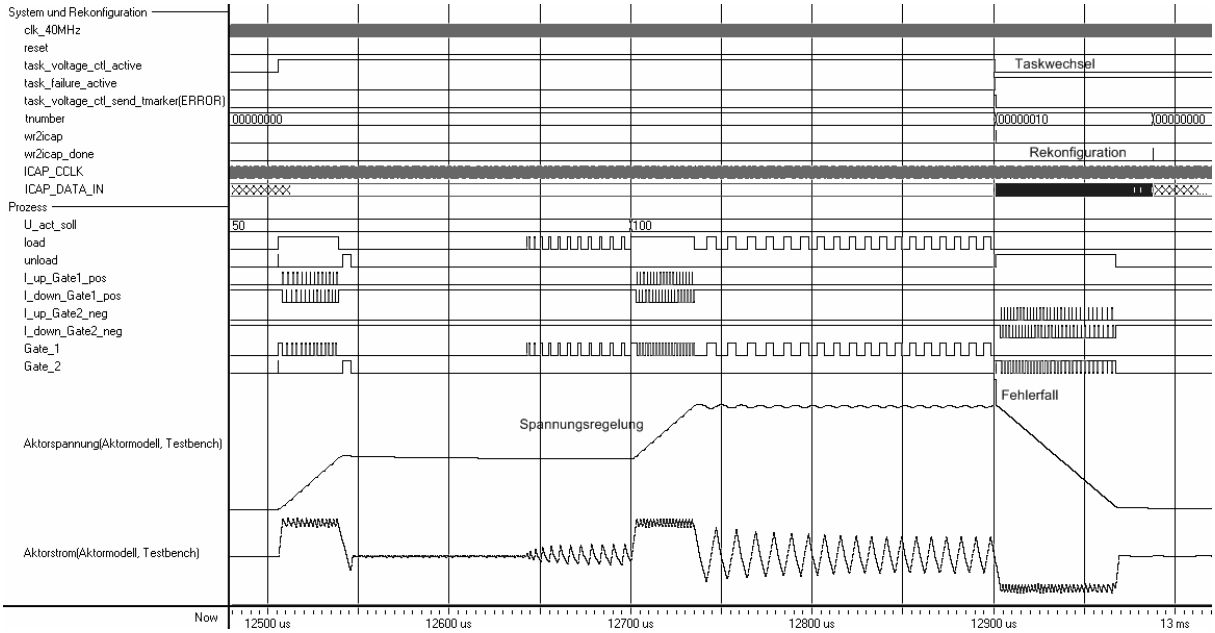


Abbildung 41: Taskwechsel des Piezocontrollers mit Rekonfigurationsvorgang und Prozessgrößen (Verhaltenssimulation, ModelSim)

6.3 Rapid Prototyping und Steuergeräteentwicklung

Die in dieser Arbeit vorgestellten Methoden und Funktionsgruppen können neben Einzelanwendungen auch für das Rapid Prototyping in der Steuergeräteentwicklung eingesetzt werden. Der Begriff Rapid Prototyping beschreibt hier die prototypische Umsetzung und Validation von Steuer- und Regelfunktionalitäten im Kontext einer konkreten Anwendung. In der Softwareentwicklung kraftfahrzeugbezogener Systeme stellt das Rapid Prototyping mit speziellen Entwicklungssteuergeräten einen wichtigen Teil des Entwicklungsprozesses dar [27]. Der Einsatz von FPGAs in der Entwicklung hardwarebasierter Funktionalitäten kraftfahrzeugbezogener Systeme kann auf eine ähnliche Weise erfolgen [41]. Die Verbindung hardware- und softwarebasierter Ansätze führt dabei zu neuen und flexiblen Methoden in der Entwicklung mechatronischer Fahrzeugsysteme [92].

Allgemein werden FPGAs in aktuellen Rapid-Prototyping-Systemen vor allem für die Realisierung zeitkritischer und rechenintensiver I/O-Funktionen eingesetzt, da sie eine hohe Rechenleistung unter Echtzeitbedingungen mit hohen Abtastzeiten verbinden. Die grundlegende Programmierbarkeit der Bausteine ermöglicht dabei die Beibehaltung der Hardwareplattform über den Entwicklungszeitraum. Schnittstellen und Funktionalitäten werden durch die Neukonfiguration der Bausteine angepasst. Auf diese Weise kann die Bewertung verschiedener I/O-Konzepte sowie Aktor- und Sensorschnittstellen und die Fehlerkorrektur auf der Grundlage einer durchgängigen Hardwareplattform erfolgen. Die Umprogrammierung dieser klassischen Systeme muss jedoch außerhalb der Laufzeit der Prototyping-Umgebung stattfinden, so dass alle zur Laufzeit benötigten Funktionalitäten parallel in Hardware zu implementieren sind.

Auf der Grundlage der in dieser Arbeit entwickelten Methoden und Funktionsgruppen können FPGAs im Rahmen von Rapid-Prototyping-Systemen wesentlich flexibler eingesetzt werden. Die partielle Rekonfiguration der Bausteine ermöglicht dabei die Portierung von aus der Softwareentwicklung bekannten Prototyping-Funktionen in den Bereich der digitalen Hardwareentwicklung. So können hardwarebasierte Parameter, Steuer- und Regelstrukturen, einzelne Funktionen und ganze Funktionsmodule während der Laufzeit des Prototyping-Systems dynamisch modifiziert und angepasst werden. Ausgangspunkt dafür ist die Spezifikation und Implementierung dieser Parameter, Strukturen und Funktionen als Hardwaretasks. Die Größe der zugehörigen rekonfigurierbaren Module und damit die Granularität der partiellen Rekonfiguration ist innerhalb der zielhardwarebedingten Grenzen (vergleiche Abschnitt 5.1.1) variabel und kann an den Ressourcenbedarf der zu modifizierenden Funktionalität angeglichen werden. Mit sehr kleinen Modulen kann so der Zugriff auf Parametersätze und untergeordnete Bestandteile von Steuer- und Regelstrukturen erfolgen, während größere Module den Austausch ganzer Funktionen und Strukturen unterstützen.

Der Einsatz rekonfigurierbarer Controller zur externen prototypischen Realisierung von hardwarebasierten Funktionalitäten eines existierenden Steuergerätes wird hier in Analogie zur Softwareentwicklung als *Bypass*-Verfahren bezeichnet [27]. Dabei werden Funktionen aus dem Steuergerät ausgekoppelt und extern auf einem FPGA-basierten rekonfigurierbaren Controller umgesetzt. Dieses Vorgehen bedingt das Freischneiden der extern umzusetzenden Funktionalität und das Vorhandensein einer Bypass-Schnittstelle auf dem Steuergerät. Die Bypass-Schnittstelle sendet die Eingangsdaten der freigeschnittenen Funktion an den rekonfigurierbaren Controller und leitet dessen Ausgangsdaten wieder in den Datenfluss des Steuer-

gerätes ein. Die auf dem rekonfigurierbaren Controller umgesetzten Algorithmen können mit den oben beschriebenen Prototyping-Funktionen zur Laufzeit des Steuergerätes modifiziert werden. Der Controller wird dazu über seine USB-Schnittstelle mit einem übergeordneten PC-System verbunden und empfängt auf diesem Weg die erforderlichen Konfigurationsdaten.

Die Erweiterung eines rekonfigurierbaren Controllers um ein übergeordnetes mikroprozessorbasiertes System ermöglicht die prototypische Implementierung von bedeutend komplexeren Steuergerätefunktionen. Diese Vorgehensweise wird hier in Analogie zur Softwareentwicklung als *Fullpass*-Verfahren bezeichnet [27]. Dabei bilden FPGA, Mikroprozessorsystem und erforderliche Aktor-, Sensor- und Kommunikationsschnittstellen ein eigenständiges Entwicklungssteuergerät, das flexibel software- und hardwarebasierte Funktionalitäten umsetzen kann. Das Mikroprozessorsystem kann sowohl FPGA-intern als auch extern implementiert werden und realisiert neben überordneten Steuer- und Regelfunktionen auch die Bereitstellung von Hardwaretasks für den untergeordneten rekonfigurierbaren Controller. Dieser setzt rechenintensive und zeitkritische Steuer-, Regel- und I/O-Funktionen um, die zur Laufzeit des Systems parametriert und modifiziert werden können, und entlastet so den Mikroprozessor.

Ein beispielhaftes Szenario einer Fullpass-Anwendung mit rekonfigurierbarem Controller kann auf der Grundlage der in Abschnitt 6.2 vorgestellten Ansteuerung für piezo-elektrische Aktoren entworfen werden. Die Leistungsverstärkertopologie und der rekonfigurierbare Controller werden dabei um ein Mikroprozessorsystem ergänzt und als Entwicklungssteuergerät für die piezo-basierte Kraftstoffeinspritzung eingesetzt. Die Einspritzmengenberechnung und Generierung der Einspritzimpulse wird auf einer softwarebasierten Funktionsebene mit dem Mikroprozessorsystem realisiert. Im Kontext piezo-elektrischer Injektoren beruht die Impulsvorgabe auf zeitabhängigen Sollwertverläufen der Aktorspannung, die in Pulsbreite, Pulsform sowie Spannungsanstiegen variieren können. Die Umsetzung der Impulsvorgabe erfolgt wie in Abschnitt 6.2rgabe erfolgt wie in Abschnitt 6.2ntroller mit hochdynamischen Algorithmen für die Strom- und Spannungsregelung. Darüber hinaus implementiert der rekonfigurierbare Controller aktornahe Diagnosefunktionen, die eine Analyse der elektromechanischen Eigenschaften angeschlossener Aktoren durchführen und so die Nachparametrierung des Systems ermöglichen. Auf der Grundlage der partiellen Rekonfiguration können die Diagnosefunktionen für die Inbetriebnahme piezo-elektrischer Injektoren geladen und danach durch die Strom- und Spannungsregelung ersetzt werden. Weiterhin können die einzelnen hardwarebasierten Funktionen im laufenden Betrieb des Entwicklungssteuergerätes dynamisch parametriert und modifiziert werden.

7 Fazit und Ausblick

In der vorliegenden Arbeit wurden der Entwurf und die Umsetzung rekonfigurierbarer Controller für mechatronische Systeme auf der Basis moderner partiell rekonfigurierbarer FPGAs beschrieben. Ausgehend von bestehenden Entwurfsmethoden und Spezifikationswerkzeugen der Mechatronik wurde eine Entwurfsmethodik für rekonfigurierbare Controller entwickelt, die eine fachnahe Spezifikation und Umsetzung der partiellen Rekonfiguration ermöglicht. Die Methodik umfasst in Anlehnung an die Softwareentwicklung kraftfahrzeugbezogener Systeme ein Vorgehensmodell, grafische Spezifikationswerkzeuge sowie abstrahierende System- und Komponentenspezifikationen. Weiterhin beinhaltet die Entwurfsmethodik ein echtzeitfähiges verteiltes Rekonfigurationsmanagement, das implizit während der System- und Komponentenspezifikation berücksichtigt wird. Die Spezifikation der rekonfigurierbaren Funktionalitäten der Controller erfolgt dabei mit Zustandsautomaten, die die Strukturvariabilität der Zielhardware abbilden. So werden Betriebszustände und Funktionen eines Controllers festen Hardwarebereichen, den rekonfigurierbaren Modulen, zugeordnet und mittels partieller Rekonfiguration als Hardwaretasks in diese geladen. Die Anwendung einer Vorablade- und Aktivierungsstrategie ermöglicht einen echtzeitfähigen Wechsel zwischen Hardwaretasks.

Es wurde eine prototypische technische Struktur rekonfigurierbarer Controller entwickelt und die rekonfigurierbaren Funktionalitäten um statische, nicht rekonfigurierbare, Infrastrukturkomponenten ergänzt. Im Rahmen der Arbeit wurden so ein dediziertes Kommunikationssystem, Funktionsgruppen eines Rekonfigurations- und Speichermanagements sowie Komponenten für die Zustandssicherung und Zustandswiederherstellung von Hardwaretasks entwickelt. Weiterhin wurden Kommunikations- und Prozessschnittstellen für den Einsatz rekonfigurierbarer Controller in mechatronischen Systemen spezifiziert und umgesetzt. Die statischen Infrastrukturkomponenten und rekonfigurierbaren Steuer- und Regelfunktionen werden in Form von Funktionsbibliotheken im Rahmen eines übergeordneten Spezifikationsframework bereitgestellt. Das Spezifikationsframework umfasst darüber hinaus eine Werkzeugkette für den

Entwurf, die funktionale Verifikation und eine initiale Hardwareumsetzung der Controllerkomponenten. Die Implementierung der Controller erfolgt mit herstellerspezifischen Implementierungswerkzeugen direkt in Hardware auf partiell rekonfigurierbaren FPGAs der Firma Xilinx. Neben bitparallelen Algorithmen kommen auch bitserielle Methoden zum Einsatz, mit denen rekonfigurierbare Funktionalitäten sehr effizient und gut skalierend auf die Zielhardware abgebildet werden können. Die Implementierung wird um eine echtzeitfähige Rekonfigurationslösung für die interne Konfigurationsschnittstelle von Xilinx FPGAs ergänzt. Diese beinhaltet einen direkten Speicherzugriff auf die Konfigurationsdaten der Hardwaretasks und eine hardwarebasierte Ansteuerung der internen Konfigurationsschnittstelle. Damit wird die Selbstrekonfiguration der Controller ermöglicht.

Abschließend wurde die Anwendbarkeit und Leistungsfähigkeit der entwickelten Methoden und Konzepte mit Beispielimplementierungen konkreter Anwendungen aus dem Bereich der Mechatronik nachgewiesen. Weiterhin wurde der Einsatz rekonfigurierbarer Controller für die Steuergeräteentwicklung in modernen Rapid-Prototyping-Systemen diskutiert. Mit einer verbesserten Herstellerunterstützung der partiellen Rekonfiguration kann der Einsatz rekonfigurierbarer Controller in mechatronischen Systemen weiter vereinfacht und attraktiver gestaltet werden. So zählen die Reduzierung des Rekonfigurationsüberhangs durch effizientere Konfigurationsdaten sowie die freie Platzierung rekonfigurierbarer Funktionen auf der Zielhardware zu aktuellen Forschungsthemen. Darüber hinaus wird die Codegenerierung für hardwarebasierte Funktionen aus gängigen Spezifikationsumgebungen heraus zunehmend unterstützt. Besonders in der Steuergeräteentwicklung ermöglicht die Kombination aus softwarebasierten und rekonfigurierbaren hardwarebasierten Prototyping-Werkzeugen innovative Ansätze und durchgängige Entwurfsumgebungen. Die in dieser Arbeit vorgestellten Methoden und Konzepte sind flexibel erweiterbar und bilden eine Grundlage für den Entwurf und die Implementierung rekonfigurierbarer Controller mit zukünftigen Hardwaregenerationen.

Bezeichnungen und Formelzeichen

Abkürzungen und Bezeichnungen:

ADU	Analog/Digital-Umsetzer
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
BLDC-Motor	Brushless DC Motor, bürstenloser Gleichstrommotor
CAN	Controller Area Network
CGRD	Coarse Grained Reconfigurable Device
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CS-FSM	Communication System FSM
CY	Carry Logic, Übertragslogik
DC/DC-Wandler	Gleichstromwandler
DC-Antrieb/Motor	Gleichstromantrieb, Gleichstrommotor
DCM	Digital Clock Manager
DFF	D-Flipflop
DSP	Digital Signal Processing, digitale Signalverarbeitung
EAPR Design Flow	Early Access Partial Reconfiguration Design Flow
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIFO	First In – First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine, endlicher Zustandsautomat
GP-CPU	General Purpose Central Processing Unit
HDL	Hardware Description Language
I/O	Input/Output
I^2C	Inter-Integrated Circuit, serieller Datenbus
ICAP	Internal Configuration Access Port
JTAG	Joint Test Action Group, IEEE-Standard 1149.1
LSB	Least Significant Bit, niedrigstwertiges Bit
LUT	Look-Up Table

MSB	Most Significant Bit, höchstwertiges Bit
MULT	Multiplizierer
MUX	Multiplexer
OPB	On-Chip Peripheral Bus
OSEK/VDX-OS	Spezifikation für Echtzeitbetriebssysteme des OSEK/VDX-Gremiums
OSR	Over Sampling Ratio
p2s	Parallel/Seriell-Wandlung
PAL	Programmable Array Logic
PCM	Puls Code Modulation
PI	Proportional-Integral(-Glied/Regler)
PID	Proportional-Integral-Differential(-Glied/Regler)
PLB	Processor Local Bus
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
PWM	Pulsweitenmodulation
RAM	Random Access Memory
s2p	Seriell/Parallel-Wandlung
SBS	statisches Basissystem
SDR	Software Defined Radio
SoC	System on Chip
SPI	Serial Peripheral Interface, serieller Datenbus
SRAM	Static RAM
S-REG	Schieberegister
UML	Unified Modelling Language
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits

Formelzeichen und Signale:

b_n	n-tes Bit
chk	check, Synchronisationssignal
f_C	Taktfrequenz der Signalverarbeitung
$f_{ICAP\_CLK(\_MAX)}$	(maximale) Taktfrequenz der ICAP-Ansteuerung

f_{IN}	Frequenz der Eingangssignale
f_{PCM}	Frequenz des pulscodemodulierten Signals
$f_{S(max)}$	(maximale) Abtastfrequenz
i	elektrischer Strom
K_I	Verstärkungsfaktor des Integralanteils
K_P	Verstärkungsfaktor des Proportionalanteils
L	Induktivität
n	Verarbeitungswortbreite
N_R	Anzahl durch Betriebszustandswechsel erreichbarer Betriebszustände
N_S	Anzahl möglicher Betriebszustände
op_new	neuer Operand für Parallel/Seriell-Wandlung
R	ohmscher Widerstand
t	Zeit
T_P	Signalverarbeitungszeit
T_R	Rekonfigurationszeit
T_S	Abtastzeit der Signalverarbeitung
u	elektrische Spannung
U(act)	Aktorspannung
U +, -	Versorgungsspannung
wr2icap	Signal für die Initiierung der Rekonfiguration über ICAP
x_{Bytes}	Anzahl von Bytes eines Konfigurationsdatensatzes

Bezeichnungen der Funktionsgruppe für Zustandssicherung und Zustandswiederherstellung:

mem_in	Datenleitung Basissystem - Komponente
mem_ok	komponenteninterner Speicher auslesebereit
mem_out	Datenleitung Komponente - Basissystem
write_back	initiiert Zustandswiederherstellung
writing2icap	initiiert finalen Speichervorgang des Komponentenzustands im Rekonfigurationsfall

Bezeichnungen der internen Konfigurationsschnittstelle ICAP:

BUSY	Aktivitätssignal
CE	Freigabesignal

CLK	Takt
I	Dateneingang
O	Datenausgang
WRITE	Schreibfreigabe

Bezeichnungen des Zustandsautomaten für die ICAP-Ansteuerung (Rekonf-FSM):

BYTE_COUNTER	Zähler für Konfigurationsdatenbytes
CONFDATA_ADDR	Speicheradresse des zu ladenden Hardwaretasks
CONFDATA_BYTES	Anzahl der Bytes des zu ladenden Hardwaretasks
ICAP_CE	Freigabesignal für ICAP
ICAP_CLK	Taktsignal für ICAP
ICAP_WRITE	Schreibfreigabe für ICAP
RAM_RDADDR	zu lesende Adresse des Hauptspeichers
RAM_READ_ENABLE	Lesefreigabe des Hauptspeichers
WR2ICAP	Signal für die Initiierung der Rekonfiguration über ICAP

Bezeichnungen des rekonfigurierbaren Controllers für piezo-elektrische Aktoren:

Gate_1	Ansteuersignal für leistungselektronischen Schalter 1
Gate_2	Ansteuersignal für leistungselektronischen Schalter 2
$I_{down\ Gate\ 1}$	Komparatorsignal für oberen Grenzwert des positiven Aktorstroms
$I_{down\ Gate\ 2}$	Komparatorsignal für unteren Grenzwert des positiven Aktorstroms
$I_{up\ Gate\ 1}$	Komparatorsignal für oberen Grenzwert des negativen Aktorstroms
$I_{up\ Gate\ 2}$	Komparatorsignal für unteren Grenzwert des negativen Aktorstroms
load	initiiert Laden des Aktors
U_{act_ist}	Istwert der Aktorspannung
U_{act_soll}	Sollwert derAktorspannung
unload	initiiert Entladen des Aktors

Literaturverzeichnis

[1] Schiffmann, Wolfram: *Technische Informatik Band 2: Grundlagen der Computertechnik*. Springer, 2005

[2] Doblinger, Gerhard: *Signalprozessoren: Architekturen, Algorithmen, Anwendungen*. J. Schlembach Fachverlag, Wilburgstetten, 2004

[3] Schiffmann, Wolfram ; Schmitz, Robert: *Technische Informatik Band 1: Grundlagen der digitalen Elektronik*. Springer, 2004

[4] Bobda, Christophe: *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*. Springer, 2007

[5] Compton, Katherine ; Hauck, Scott: Reconfigurable computing: a survey of systems and software. In: *ACM Comput. Surv.* 34 (2002), Nr. 2, S. 171–210.

[6] Oldfield, John V. ; Dorf, Richard C.: *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. Wiley & Sons, New York, 1995

[7] Meyer-Baese, Uwe: *Digital Signal Processing with Field Programmable Gate Arrays*. 2. Auflage. Springer, 2004

[8] Gokhale, Maya ; Graham, Paul S.: *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005

[9] Xilinx: *Virtex-4 User Guide*. (v2.3), 2007

[10] Xilinx: *Spartan-3 Generation FPGA User Guide*. (v1.4), 2008

[11] Altera: *Stratix II Device Family*. http://www.altera.com/products/devices/stratix2/ [Juni 2008]

[12] Lattice: *LatticeXP2 FPGA*. http://www.latticesemi.com/products/fpga/xp2/ [Juni 2008]

[13] Actel: *Actel Fusion Programmable System Chips Data Sheet*. (v1.2). 2008

[14] Xilinx: *PowerPC 405 Processor Block Reference Guide*. (v2.2), 2007

[15] Xilinx: *MicroBlaze Processor Reference Guide*. (v8.0), 2007

[16] Altera: *Nios II Embedded Processor*. http://www.altera.com/products/ip/processors/nios2/ [Juni 2008]

[17] Lattice: *LatticeMico32 Soft Processor*. http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/ [Juni 2008]

[18] Altera: *MAX II Device Handbook*. (MII5V1-3.0), 2007

[19] Atmel: *Field Programmable System Level Integrated Circuits*. http://www.atmel.com/products/FPSLIC/ [Juni 2008]

[20] Atmel: *Performing Dynamic Reconfiguration in FPSLIC™ Devices - A Scrolling Message Display*. Atmel Application Note (3515A-FPSLI-9/04), 2007

[21] PACT XPP Technologies: *XPP-III Processor Overview*. White Paper Version 2.0.1, 2006

[22] Stretch: *S6000 Family Software Configurable Processors*. http://www.stretchinc.com/products/s6000.php [Juni 2008]

[23] Lange, Hendrik: *Modellbasierte Effizienzanalyse grobgranularer rekonfigurierbarer Prozessorarchitekturen*, Universität Dortmund, Diss., 2007

[24] Bundesministerium des Inneren: *V-Modell XT*. http://www.v-modell-xt.de/ [Juni 2008]

[25] *VDI 2206: Entwicklungsmethodik für mechatronische Systeme*. Beuth Verlag, Berlin, 2004

[26] Reif, Konrad: *Automobilelektronik*. 2. Auflage. Vieweg, Wiesbaden, 2007

[27] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering*. 3. Auflage. Vieweg, 2006

[28] Reichardt, Jürgen ; Schwarz, Bernd: *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*. Oldenbourg Verlag München, 2000

[29] Burmester, Sven: *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. Logos-Verlag, Berlin, 2006

[30] Duc, Bui M.: *Real-Time Object Uniform Design Methodology with UML*. Springer, 2007

[31] Steinbach, Bernd ; Fröhlich, Dominik ; Beierlein, Thomas: Hardware/Software Codesign of Reconfigurable Architectures Using UML. In: Martin, Grant (Hrsg.) ; Müller, Wolfgang (Hrsg.): *UML for SOC Design*. Springer, 2005, S. 89–117

[32] Riefenstahl, Ulrich: *Elektrische Antriebstechnik*. B.G. Teubner Stuttgart, 2000

[33] Atmel: *Motor Control using FPSLIC™/FPGA*. Atmel Application Note (Rev. 3023A–FPSLI–06/02), 2007

[34] Xilinx: *Logic-Based AC Induction Motor Controller*. Xilinx Application Note XAPP448 (v1.0), 2005

[35] Vélez, D. ; Shanblatt, M.: FPGA Implementation of a Brushless Motor Control. In: *Proceedings of Circuits, Signals, and Systems (CSS 2006)*, 2006

[36] Xilinx: *FPGA Motor Control Reference Design*. Xilinx Application Note XAPP808 (v1.0), 2005

[37] Actel: *Actel Fusion Application Solutions: Motor Control*. http://www.actel.com/products/solutions/motorcontrol/ [Juni 2008]

[38] He, P. ; Jin, M.H. ; Yang, L. ; Wei, R. ; Liu, Y.W. ; Cai, H.G. ; Liu, H. ; Seitz, N. ; Butterfass, J. ; Hirzinger, G.: High performance DSP/FPGA controller for implementation of HIT/DLR dexterous robot hand. In: *Proc. IEEE International Conference on Robotics and Automation ICRA '04*, 2004. S. 3397–3402

[39] Chaoui, H. ; Yagoub, M.C.E. ; Sicard, P.: FPGA Implementation of a Fuzzy Controller for Neural Network Based Adaptive Control of a Flexible Joint with Hard Nonlinearities. In: *Proc. IEEE International Symposium on Industrial Electronics*, 2006, S. 3124–3129

[40] Toscher, Steffen: *Entwurf und Realisierung einer durchgängigen Entwicklungsumgebung zur flexiblen Umsetzung von Steuergerätefunktionalität für elektronisch kommutierte Elektromotoren in Hardware und Software unter Ausnutzung der Rekonfigurierbarkeit von programmierbarer Logik*, Institut für Mechatronik und Antriebstechnik, Otto-von-Guericke-Universität Magdeburg, Diplomarbeit, 2004

[41] Dase, C. ; Falcon, J.S. ; Maccleery, B.: Motorcycle control prototyping using an FPGA-based embedded control system. In: *IEEE Control Systems Magazine* 26 (2006), Nr. 5, S. 17–21.

[42] National Instruments: *NI CompactRIO*. http://www.ni.com/compactrio/ [Juni 2008]

[43] National Instruments: *NI LabVIEW FPGA*. http://www.ni.com/fpga/d/ [Juni 2008]

[44] Kao, Cindy: Benefits of Partial Reconfiguration. In: *Xilinx Xcell Journal* 55 (2005), S. 65 – 67

[45] Megacz, Adam: A Library and Platform for FPGA Bitstream Manipulation. In: *Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM 2007*, 2007, S. 45–54

[46] Lattice: *LatticeSC sysCONFIG Usage Guide*. Technical Note TN1080, 2008

[47] Xilinx: *Virtex-II Platform FPGA User Guide*. (v2.2), 2007

[48] Xilinx: *Spartan-3 Advanced Configuration Architecture*. Xilinx Application Note XAPP452 (v1.0), 2004

[49] Lysaght, P. ; Blodget, B. ; Mason, J. ; Young, J. ; Bridgford, B.: Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In: *Proc. International Conference on Field Programmable Logic and Applications FPL '06*, 2006

[50] Xilinx: *Difference-Based Partial Reconfiguration.* Xilinx Application Note XAPP290 (v2.0), 2007

[51] Xilinx: *Early Access Partial Reconfiguration User Guide*. (v1.1), 2006

[52] Xilinx: *Two Flows for Partial Reconfiguration: Module Based or Difference Based*. Xilinx Application Note XAPP290 (v1.2), 2004

[53] Dorairaj, Nij ; Shiflet, Eric ; Goosman, Mark: PlanAhead Software as a Platform for Partial Reconfiguration. In: *Xilinx Xcell Journal* 55 (2005), S. 68 – 71

[54] Blodget, Brandon ; James-Roxby, Philip ; Keller, Eric ; McMillan, Scott ; Sundararajan, Prasanna: A Self-reconfiguring Platform. In: *Lecture Notes in Computer Science Volume 2778/2003*. Springer, 2003, S. 565-574

[55] Sedcole, Nicholas P.: *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing*, Imperial College of Science, Technology and Medicine, University of London, Diss., 2006

[56] Claus, Christopher ; Zeppenfeld, Johannes ; Müller, Florian ; Stechele, Walter: Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In: *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, 2007. S. 498–503

[57] Delahaye, Jean P. ; Gogniat, Guy ; Roland, Christian ; Bomel, Pierre: Software radio and dynamic reconfiguration on a DSP/FPGA platform. In: *frequenz, journal of telecommunications* 58 (2004), S. 152-159.

[58] ISR Technologies: *JTRS SDR KIT*. http://www.isr-t.com/JTRS_SDR_eng.shtml und http://www.xilinx.com/prs_rls/dsp/0626_sdr.htm [Juni 2008]

[59] Chujo, Naoya: Fail-safe ECU System Using Dynamic Reconfiguration of FPGA. In: *R&D Review of Toyota CRDL* 37 (2002), S. 54–60

[60] Danne, Klaus ; Bobda, Christophe ; Kalte, Heiko: Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration. In: *Lecture Notes in Computer Science Volume 2778/2003*. Springer, 2003, S. 272–281

[61] Rullmann, M. ; Merker, R.: A Reconfiguration Aware Circuit Mapper for FPGAs. In: *Proc. IEEE International Parallel and Distributed Processing Symposium IPDPS 2007*, 2007

[62] Tan, Heng ; DeMara, R.F.: A Physical Resource Management Approach to Minimizing FPGA Partial Reconfiguration Overhead. In: *Proc. IEEE International Conference on Reconfigurable Computing and FPGA's ReConFig 2006*, 2006

[63] Claus, Christopher ; Müller, Florian H. ; Stechele, Walter: Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro. In: *Workshop on reconfigurable computing Proceedings (ARCS 06)*, 2006, S. 122–131

[64] Majer, Mateusz ; Teich, Jürgen ; Ahmadinia, Ali ; Bobda, Christophe: The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. In: *The Journal of VLSI Signal Processing* Volume 47 (2007), Nr. 2, S. 15–31

[65] Reinemann, Thomas: *Verfahren zur direkten Implementierung von Algorithmen auf Gatterebene*, Otto-von-Guericke-Universität Magdeburg, Diss., 2003

[66] Reinemann, Thomas ; Kasper, Roland: Delay optimization for bit serial algorithms. In: *Proceedings of International Signal Processing Conference (GSPx 2003)*, 2003

[67] Mentor Graphics: *HDL Designer Series User Manual*. (v2007.1). 2007

[68] Mentor Graphics: *Graphical Editors User Manual for the HDL Designer Series*. (v2007.1), 2007

[69] Mentor Graphics: *State Machine Editors User Manual for the HDL Designer Series*. (v2007.1), 2007

[70] The Mathworks: *Simulink® HDL Coder™ 1.2*. http://www.mathworks.com/products/slhdlcoder/ [Januar 2008]

[71] Xilinx: *System Generator for DSP*. http://www.xilinx.com/ise/optional_prod/system_generator.htm [Juni 2008]

[72] *OSEK/VDX Operating System Specification 2.2.3*. http://www.osek-vdx.org/. [Juni 2008]

[73] *Universal Serial Bus Specification*. (Revision 2.0). http://www.usb.org. [Juni 2008]

[74] Cypress: *CY7C68013 EZ-USB FX2™ USB Microcontroller Data Sheet*. (Rev. *F), 2005

[75] Toscher, S. ; Reinemann, T. ; Kasper, R. ; Hartmann, M.: A Reconfigurable Delta-Sigma ADC. In: *Proc. IEEE International Symposium on Industrial Electronics*, 2006, S. 495–499

[76] Norsworthy, Steven R. ; Schreier, Richard ; Temes, Gabor C.: *Delta-Sigma Data Converters: Theory, Design, and Simulation*. IEEE Press, Piscataway, NJ, 1997

[77] Mentor Graphics: *ModelSim*. http://www.model.com/ [Juni 2008]

[78] Mentor Graphics: *Precision RTL*. http://www.mentor.com/products/fpga_pld/synthesis/precision_rtl/ [Juni 2008]

[79] Xilinx: *Virtex-II Platform FPGAs: Complete Data Sheet*. (v3.5), 2007

[80] Xilinx: *Virtex-4 Data Sheet: DC and Switching Characteristics*. (v3.1), 2007

[81] Xilinx: *Spartan-3 FPGA Family: Complete Data Sheet*. (v2.3), 2007

[82] Xilinx: *Spartan-3 Generation Configuration User Guide*. (v1.3), 2007

[83] Xilinx: *Virtex-4 Configuration Guide*. (v1.9), 2007

[84] Xilinx: *LogiCore OPB HWICAP Data Sheet*. (v1.3), 2004

[85] Xilinx: *LogiCore XPS HWICAP Data Sheet*. (v1.00a), 2007

[86] Xilinx: *Development System Reference Guide 9.1i*. 2007

[87] Gnad, Gunnar ; Kasper, Roland: Power Drive Circuits for Piezo-Electric Actuators in Automotive Applications. In: *Proc. IEEE International Conference on Industrial Technology ICIT 2006*, 2006, S. 1597–1600

[88] Gnad, Gunnar: *Ansteuerkonzept für piezoelektrische Aktoren*, Otto-von-Guericke-Universität Magdeburg, Diss., 2005

[89] Ruschmeyer, Karl ; Bartz, Wilfried J. (Hrsg.): *Piezokeramik : Grundlagen, Werkstoffe, Applikationen*. expert-Verlag, 1995

[90] Ikeda, Takuro: *Fundamentals of piezoelectricity*. Oxford University Press, 1996 (Oxford Science Publications)

[91] Kasper, Roland: *Mechatronik III - Mechatronische Aktoren und Sensoren*. Vorlesung an der Otto-von-Guericke Universität Magdeburg, 2007

[92] Kasper, Markus: Flexible Ansteuerung für magnetventilbasierte Komponenten. In: *atp* 06 (2004), S. 71–77

Anhang

Der Anhang beinhaltet Quelltexte ausgewählter Funktionsgruppen und Anwendungen der in dieser Arbeit entwickelten rekonfigurierbaren Controller. Die Quelltexte basieren zum Teil auf Codegenerierung und liegen in VHDL vor. Sie werden durch ein beispielhaftes Implementierungsskript ergänzt.

Aus Gründen der Übersichtlichkeit werden die Quelltexte und Skripte gekürzt abgebildet.

Anhang A Beispiele für Funktionsgruppen

A.1 Zustandssicherung und Zustandswiederherstellung

VHDL-Quelltext der Komponente für Zustandssicherung und Zustandswiederherstellung eines bitseriellen Integrators (MEM_RECON):

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

entity mem_recon_i_ser is
generic (
  word_length  : integer :=0 );
port (
  clk          : in  std_logic;
  reset        : in  std_logic;
  writing2icap : in  std_logic;
  write_back   : in  std_logic;
  mem_complete : in  std_logic;
  mem_in       : in  std_logic;
  mem_save_out : out std_logic );
end mem_recon_i_ser;

architecture arch_mem_recon of mem_recon_i_ser is

  signal task_counter    : integer := 0;
  signal mem_position    : integer := word_length - 2;
  signal write_back_hold : std_logic;
  signal mem1_save       : std_logic_vector (word_length - 2 downto 0);
  signal mem2_save       : std_logic_vector (word_length - 2 downto 0);
  signal mem_in_par      : std_logic_vector (word_length - 2 downto 0);

begin  -- arch_mem_recon
  task_event: process (writing2icap, reset)
  begin  -- process task_event
    if reset = '1' then
      task_counter <= 0;
    elsif writing2icap'event and writing2icap = '1' then
      if  task_counter = 15 then
        task_counter <= 0;
      else
        task_counter <= task_counter + 1 ;
      end if;
    end if;
  end process task_event;

  mem_position_process : process (clk, write_back) is
  begin  -- process  mem_position
      if write_back = '1' then
      mem_position <= 0;
      write_back_hold <= '1';
       elsif clk'event and clk = '1' then
         if mem_position = word_length - 2 then
             write_back_hold <= '0';
          else
             mem_position <= mem_position + 1 ;
             write_back_hold <= '1';
```

```
        end if;
      end if;
   end process mem_position_process;

   memory_in_par_proc: process (clk, reset, mem_complete)
      variable mem_var : std_logic_vector (word_length - 2 downto 0);
    begin
      if reset = '1' then
        mem_var := (others => '0');
        mem_in_par <= (others => '0');
      elsif clk'event and clk = '1' then
        mem_var := mem_in & mem_var (mem_var'high  downto 1);
        if mem_complete = '1' then
          mem_in_par <= mem_var;
        end if;
      end if;
    end process memory_in_par_proc;

  memory_save_proc: process (clk, reset, mem_complete)
  begin  -- process memory_save_out
    if reset = '1' then
      mem1_save <= (others => '0');
      mem2_save <= (others => '0');
      mem_save_out <= '0';
    elsif mem_complete'event and mem_complete = '0' then
      if task_counter mod 2=0 then
        mem1_save <= mem_in_par;
      end if;
      if task_counter mod 2=1 then
        mem2_save <= mem_in_par;
      end if;
    elsif write_back_hold = '1' and clk'event and clk = '0' then
      if task_counter mod 2= 1 then
        mem_save_out <= mem1_save(mem_position);
      end if;
      if task_counter mod 2= 0 then
        mem_save_out <= mem2_save(mem_position);
      end if;
    end if;
  end process memory_save_proc;
end arch_mem_recon;
```

A.2 Rekonfigurationsmechanismus

VHDL-Quelltext des Zustandsautomaten für die Ansteuerung von ICAP eines Xilinx Virtex-4 FPGAs im 32 Bit Modus (Rekonf-FSM):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;

ENTITY rekon_fsm_v4_32 IS
   PORT(
      clk               : IN     std_logic;
      conf_t_number     : IN     std_logic_vector ( 7 DOWNTO 0 );
      conf_table_data   : IN     std_logic_vector ( 7 DOWNTO 0 );
      reset             : IN     std_logic;
      wr2icap           : IN     std_logic;
```

```
      ICAP_CCLK         : OUT    std_logic;
      ICAP_CE           : OUT    std_logic;
      ICAP_WR           : OUT    std_logic;
      conf_table_rdaddr : OUT    std_logic_vector ( 4 DOWNTO 0 );
      dpram2_rdaddr     : OUT    std_logic_vector ( 23 DOWNTO 0 );
      dpram2_re         : OUT    std_logic;
      wr2icap_done      : OUT    std_logic;
      writing2icap      : OUT    std_logic);
END rekon_fsm_v4_32 ;

ARCHITECTURE fsm OF rekon_fsm_v4_32 IS

   -- Internal signal declarations
   -- ...
   TYPE STATE_TYPE IS (
      s0,
      s1_idle1,
      s1_idle2,
      s2_table,
      -- ...
      s9_table,
      -- ...
      s18_icap);
   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";
   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE;
   SIGNAL next_state : STATE_TYPE;
   -- Declare any pre-registered internal signals
   SIGNAL ICAP_CCLK_int : std_logic ;
   SIGNAL ICAP_CE_int : std_logic ;
   SIGNAL ICAP_WR_int : std_logic ;
   -- ...

BEGIN
   clocked_proc : PROCESS (clk, reset)
   BEGIN
      IF (reset = '1') THEN
         current_state <= s0;
         -- Default Reset Values
         ICAP_CCLK <= '1';
         ICAP_CE <= '1';
         ICAP_WR <= '0';
         -- ...
      ELSIF (clk'EVENT AND clk = '1') THEN
         current_state <= next_state;
         -- Registered output assignments
         ICAP_CCLK <= ICAP_CCLK_int;
         ICAP_CE <= ICAP_CE_int;
         ICAP_WR <= ICAP_WR_int;
         -- ...
         -- Combined Actions
         CASE current_state IS
            WHEN s0 =>
               dpram2_rdaddr_counter <= 0 ;
               conf_table_rdaddr_counter <= 0 ;
               confdata_byte_counter <= 0;
               confdata_addr <= (OTHERS => '0');
            WHEN s1_idle1 =>
            -- ditto ...
            WHEN s2_table =>
```

```
               conf_table_rdaddr_counter <=
(to_integer(unsigned(conf_t_number)) * 6);
            WHEN s4_table =>
               conf_table_rdaddr_counter <= conf_table_rdaddr_counter + 1 ;
            WHEN s6_table =>
               confdata_addr (15 downto 8) <= conf_table_data ;
               conf_table_rdaddr_counter <= conf_table_rdaddr_counter + 1 ;
            WHEN s7_table =>
               confdata_bytes (7 downto 0) <= conf_table_data ;
               dpram2_rdaddr_counter<=to_integer(unsigned(confdata_addr)) ;
            WHEN s8_table =>
               confdata_bytes (15 downto 8) <= conf_table_data ;
            WHEN s9_table =>
               confdata_bytes (23 downto 16) <= conf_table_data ;
            WHEN s10_icap =>
               dpram2_rdaddr_counter <= dpram2_rdaddr_counter + 1;
               confdata_byte_counter <= confdata_byte_counter + 4;
            WHEN s3_table =>
               conf_table_rdaddr_counter <= conf_table_rdaddr_counter + 1 ;
            WHEN s5_table =>
               confdata_addr (7 downto 0) <= conf_table_data ;
               conf_table_rdaddr_counter <= conf_table_rdaddr_counter + 1 ;
            WHEN s9a_sync =>
               dpram2_rdaddr_counter <= dpram2_rdaddr_counter + 1;
               confdata_byte_counter <= confdata_byte_counter + 4;
            WHEN s6a_table =>
               confdata_addr (23 downto 16) <= conf_table_data ;
               conf_table_rdaddr_counter <= conf_table_rdaddr_counter + 1 ;
            WHEN OTHERS =>
               NULL;
         END CASE;
      END IF;
   END PROCESS clocked_proc;

   nextstate_proc : PROCESS (confdata_byte_counter, confdata_bytes,
current_state, reset, wr2icap)
   BEGIN
      CASE current_state IS
         WHEN s0 =>
            IF (reset = '0') THEN
               next_state <= s1_idle1;
            ELSE
               next_state <= s0;
            END IF;
         WHEN s1_idle1 =>
            next_state <= s1_idle2;
         WHEN s2_table =>
            next_state <= s3_table;
         WHEN s4_table =>
            next_state <= s5_table;
         WHEN s6_table =>
            next_state <= s6a_table;
         WHEN s7_table =>
            next_state <= s8_table;
         WHEN s8_table =>
            next_state <= s9_table;
         WHEN s9_table =>
            next_state <= s9a_sync;
         WHEN s14_icap =>
            next_state <= s15_icap;
         WHEN s10_icap =>
            next_state <= s11_icap;
         WHEN s11_icap =>
```

```
            IF (confdata_byte_counter =
to_integer(unsigned(confdata_bytes)) + 1) THEN
               next_state <= s12_icap;
            ELSE
               next_state <= s10_icap;
            END IF;
         WHEN s3_table =>
            next_state <= s4_table;
         WHEN s5_table =>
            next_state <= s6_table;
         WHEN s12_icap =>
            next_state <= s13_icap;
         WHEN s13_icap =>
            next_state <= s14_icap;
         WHEN s15_icap =>
            next_state <= s16_icap;
         WHEN s9a_sync =>
            next_state <= s11_icap;
         WHEN s6a_table =>
            next_state <= s7_table;
         WHEN s16_icap =>
            next_state <= s17_icap;
         WHEN s17_icap =>
            next_state <= s18_icap;
         WHEN s18_icap =>
            next_state <= s1_idle1;
         WHEN s1_idle2 =>
            IF (wr2icap = '1') THEN
               next_state <= s2_table;
            ELSE
               next_state <= s1_idle1;
            END IF;
         WHEN OTHERS =>
            next_state <= s0;
      END CASE;
   END PROCESS nextstate_proc;

   output_proc : PROCESS (current_state)
   BEGIN
      -- Default Assignment
      ICAP_CCLK_int <= '1';
      ICAP_CE_int <= '1';
      ICAP_WR_int <= '0';
      -- ...
      -- Combined Actions
      CASE current_state IS
         WHEN s0 =>
            ICAP_CCLK_int <= '1';
         WHEN s1_idle1 =>
            ICAP_CCLK_int <= '1';
         WHEN s2_table =>
            ICAP_CCLK_int <= '1';
         WHEN s4_table =>
            ICAP_CCLK_int <= '1';
         WHEN s6_table =>
            ICAP_CCLK_int <= '1';
         WHEN s7_table =>
            ICAP_CCLK_int <= '1';
         WHEN s8_table =>
            dpram2_re_int <= '1' ;
            ICAP_CCLK_int <= '0';
         WHEN s9_table =>
            ICAP_CCLK_int <= '1';
```

```
            writing2icap_int <= '1';
         WHEN s14_icap =>
            ICAP_CCLK_int <= '0';
            wr2icap_done_int <= '1';
         WHEN s10_icap =>
            ICAP_CCLK_int <= '0';
            ICAP_CE_int <= '0';
         WHEN s11_icap =>
            ICAP_CCLK_int <= '1';
            ICAP_CE_int <= '0';
         WHEN s3_table =>
            ICAP_CCLK_int <= '0';
         WHEN s5_table =>
            ICAP_CCLK_int <= '0';
         WHEN s12_icap =>
            ICAP_CCLK_int <= '0';
         WHEN s13_icap =>
            ICAP_CCLK_int <= '1';
            dpram2_re_int <= '0' ;
         WHEN s15_icap =>
            ICAP_CCLK_int <= '1';
            writing2icap_int <= '0';
         WHEN s9a_sync =>
            ICAP_CCLK_int <= '0';
            ICAP_CE_int <= '0';
         WHEN s6a_table =>
            ICAP_CCLK_int <= '0';
         WHEN s16_icap =>
            ICAP_CCLK_int <= '0';
         WHEN s17_icap =>
            ICAP_CCLK_int <= '1';
         WHEN s18_icap =>
            ICAP_CCLK_int <= '0';
            wr2icap_done_int <= '0';
         WHEN s1_idle2 =>
            ICAP_CCLK_int <= '0';
         WHEN OTHERS =>
            NULL;
      END CASE;
   END PROCESS output_proc;
   -- Concurrent Statements
   dpram2_rdaddr_int <= std_logic_vector (to_unsigned
(dpram2_rdaddr_counter, dpram2_rdaddr_int'length));
   conf_table_rdaddr_int <= std_logic_vector (to_unsigned
(conf_table_rdaddr_counter, conf_table_rdaddr_int'length));
END fsm;
```

Anhang B Beispiel für Implementierungsskripte

Implementierungsskript des Anwendungsbeispiels "Rekonfigurierbarer Antriebscontroller" nach Abschnitt 6.1 für den EAPR Design Flow und einen Xilinx XC4VLX60 FPGA:

```
#!  /bin/bash
cd drivectl_v4_tl_struct/drivectl_v4_tl_struct
# translate the top level design
ngdbuild -p 4vlx60ff668-10 -uc ../../drivectl_v4_tl.ucf -modular initial
drivectl_v4_tl.edf drivectl_v4_tl.ngd
cd ../../
# implement the PR base design
cd drivectl_v4_base_struct/drivectl_v4_base_struct
cp ../../busmacros/*.nmc .
cp ../../dpram256KB_coregen/dpram256KB_coregen.ngc .
cp
../../comm_sys_wrapper_struct/comm_sys_wrapper_struct/comm_sys_wrapper.edf
.
ngdbuild -uc ../../drivectl_v4_tl.ucf -modular initial
../../drivectl_v4_tl_struct/drivectl_v4_tl_struct/drivectl_v4_tl.edf
drivectl_v4_tl.ngd
map drivectl_v4_tl.ngd
par -w drivectl_v4_tl.ncd drivectl_v4_tl_base_routed.ncd
cd ../../
# implement task static_test for slot0
cd drivectl_v4_slot0_static_test/drivectl_v4_slot0_static_test
cp ../../drivectl_v4_base_struct/drivectl_v4_base_struct/static.used
./arcs.exclude
cp ../../busmacros/*.nmc .
ngdbuild -uc ../../drivectl_v4_tl.ucf -modular module -active
drivectl_v4_slot0
../../drivectl_v4_tl_struct/drivectl_v4_tl_struct/drivectl_v4_tl.edf
map drivectl_v4_tl.ngd
par -w drivectl_v4_tl.ncd task_static_test_routed.ncd
cd ../../
# implement task pi_ctl for slot0
# ditto ...
# implement task run_up for slot1
# ditto ...
# implement task fail_safe for slot1
# ditto
# MERGES
#we have 6 folders/pairs: static_test_base, pi_ctl_base,
run_up_static_test_base, run_up_pi_ctl_base, fail_safe_static_test_base,
fail_safe_pi_ctl_base
# merge base design with task static_test
cd drivectl_v4_merges/static_test_base
cp
../../drivectl_v4_base_struct/drivectl_v4_base_struct/drivectl_v4_tl_base_r
outed.ncd .
cp
../../drivectl_v4_slot0_static_test/drivectl_v4_slot0_static_test/task_stat
ic_test_routed.ncd .
PR_verifydesign.bat drivectl_v4_tl_base_routed.ncd
task_static_test_routed.ncd
PR_assemble.bat drivectl_v4_tl_base_routed.ncd task_static_test_routed.ncd
cd ../../
# merge base design with task pi_ctl
# ditto ...
```

```
# merge base design + task static_test with task run_up
cd drivectl_v4_merges/run_up_static_test_base
cp ../static_test_base/drivectl_v4_tl_base_routed_full.ncd .
cp
../../drivectl_v4_slot1_run_up/drivectl_v4_slot1_run_up/task_run_up_routed.
ncd .
PR_verifydesign.bat drivectl_v4_tl_base_routed_full.ncd
task_run_up_routed.ncd
PR_assemble.bat drivectl_v4_tl_base_routed_full.ncd task_run_up_routed.ncd
cd ../../
# merge base design + task pi_ctl with task run_up
# ditto ...
# merge base design + task static_test with task fail_safe
# ditto ...
# merge base design + task pi_ctl with task fail_safe
# ditto ...
exit 0
```

Anhang C Beispiel für Anwendungen

C.1 Top-Level des rekonfigurierbaren Antriebscontrollers nach Abschnitt 6.1

VHDL-Quelltext für einen Xilinx Virtex-4 FPGA:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;
-- ...

ENTITY drivectl_v4_tl IS
   PORT(
      EMPTY             : IN     std_logic;      -- USB
      FULL              : IN     std_logic;
      FA                : OUT    std_logic_vector (1 DOWNTO 0);
      PKTEND            : OUT    std_logic;
      SLOE              : OUT    std_logic;
      SLRD              : OUT    std_logic;
      SLWR              : OUT    std_logic;
      FD                : INOUT  std_logic_vector (7 DOWNTO 0)
      USB_RESET         : OUT    std_logic;
      act_val_enc       : IN     std_logic;      --Process
      clk_in            : IN     std_logic;
      reset             : IN     std_logic;
      set_val           : IN     std_logic_vector (1 DOWNTO 0);
      clk_out           : OUT    std_logic;
      ctl_val_bpd       : OUT    std_logic_vector (word_length-2 DOWNTO 0);
      ctl_val_pwm       : OUT    std_logic;);
END drivectl_v4_tl ;

ARCHITECTURE struct OF drivectl_v4_tl IS

   -- Constant declarations ...
   -- Internal signal declarations ...

attribute nopad : boolean;
attribute nopad of ctl_val_bpd : signal is true;
attribute nopad of clk_out : signal is true;

   -- Component Declarations
   COMPONENT comm_sys_wrapper
   -- ...
   COMPONENT busmacro_xc4v_b2t_async_wide_4bit
   -- ...
   COMPONENT busmacro_xc4v_t2b_async_wide_2bit
   -- ...
   COMPONENT drivectl_v4_base
   -- ...
   COMPONENT drivectl_v4_slot0
   -- ...
   COMPONENT drivectl_v4_slot1
   -- ...
   COMPONENT BUFG
   -- ...
   COMPONENT BUFGMUX
   -- ...
   COMPONENT DCM_BASE
   -- ...
```

```
   COMPONENT ICAP_VIRTEX4
   -- ...
BEGIN
   -- Instance port mappings.
   COMM_SYS_I : comm_sys_wrapper
      GENERIC MAP (
         subscribers => subscribers,
         Data_length => Data_length,
         gc_slot     => gc_slot)
      PORT MAP (
         slot_data    => slot_data,
         slot_control => slot_control,
         reset        => reset_comm_sys,
         clk          => clk,
         PSIN         => PSI0,
         Start_SP     => Start_SP,
         T_O          => T_O,
         reset_O      => reset_O,
         PSOUT        => PSO0);
   bm_commsys_out_slot0 : busmacro_xc4v_b2t_async_wide_4bit
      PORT MAP (
         input0  => PSI0_0,
         input1  => S0_SC,
         input2  => S0_I,
         input3  => Start_SP_0,
         output0 => PSI0(0),
         output1 => slot_control(0),
         output2 => slot_data(0),
         output3 => Start_SP(0));
   bm_commsys_out_slot1 : busmacro_xc4v_b2t_async_wide_4bit
      PORT MAP (
         input0  => PSI0_1,
         input1  => S1_SC,
         input2  => S1_I,
         input3  => Start_SP_1,
         output0 => PSI0(1),
         output1 => slot_control(1),
         output2 => slot_data(1),
         output3 => Start_SP(1));
   bm_process_in_slot0 : busmacro_xc4v_b2t_async_wide_4bit
      PORT MAP (
         input0  => act_val_bsd,
         input1  => set_val_bsd,
         input2  => act_val_ref_bsd,
         input3  => chk_X,
         output0 => act_val_bsd_s0,
         output1 => set_val_bsd_s0,
         output2 => act_val_ref_bsd_s0,
         output3 => chk_X_s0);
   bm_process_in_slot1 : busmacro_xc4v_b2t_async_wide_4bit
      PORT MAP (
         input0  => act_val_bsd,
         input1  => set_val_bsd,
         input2  => act_val_ref_bsd,
         input3  => chk_X,
         output0 => act_val_bsd_s1,
         output1 => set_val_bsd_s1,
         output2 => act_val_ref_bsd_s1,
         output3 => chk_X_s1);
   bm_commsys_in_slot0 : busmacro_xc4v_t2b_async_wide_2bit
      PORT MAP (
         input0  => reset_O(0),
         input1  => T_O(0),
```

```
      output0 => reset_O_0,
      output1 => T_O_0);
   bm_commsys_in_slot1 : busmacro_xc4v_t2b_async_wide_2bit
      PORT MAP (
      input0  => reset_O(1),
      input1  => T_O(1),
      output0 => reset_O_1,
      output1 => T_O_1);
   drivectl_v4_base_0 : drivectl_v4_base
      PORT MAP (
      EMPTY                  => EMPTY,
      FD_in                  => FD,
      FULL                   => FULL,
      T_I                    => T_O(2),
      act_val_enc            => act_val_enc,
      clk                    => clk,
      clk_rec                => clk_200MHz,
      ctl_val_bsd            => PSO0,
      reset                  => reset,
      set_val                => set_val,
      FA                     => FA,
      FD_out                 => FD_out,
      H_fixed_bufgmux        => H_fixed_bufgmux,
      ICAP_CCLK              => ICAP_CCLK,
      ICAP_CE                => ICAP_CE,
      ICAP_DATA_IN           => ICAP_DATA_IN,
      ICAP_WR                => ICAP_WR,
      L_fixed_bufgmux        => L_fixed_bufgmux,
      PKTEND                 => PKTEND,
      PSO                    => PSI0(2),
      S0_Reset               => s0_reset,
      S1_Reset               => s1_reset,
      SC                     => GC_SC,
      SLOE                   => SLOE,
      SLRD                   => SLRD,
      SLWR                   => SLWR,
      Start_SP               => Start_SP(2),
      T_O                    => GC_I,
      USB_RESET              => USB_RESET,
      act_val_bsd            => act_val_bsd,
      act_val_ref_bsd        => act_val_ref_bsd,
      chk_X                  => chk_X,
      ctl_val_bpd            => ctl_val_bpd,
      ctl_val_pwm            => ctl_val_pwm,
      fifodata_out_enable_inv => fifodata_out_enable_inv,
      set_val_bsd            => set_val_bsd);
   drivectl_v4_slot0_0 : drivectl_v4_slot0
      GENERIC MAP (
      subscribers => subscribers,
      Data_Length => Data_Length)
      PORT MAP (
      Reset_O_0   => reset_O_0,
      T_O_0       => T_O_0,
      act_val     => act_val_bsd_s0,
      act_val_ref => act_val_ref_bsd_s0,
      chk_X       => chk_X_s0,
      clk         => clk,
      set_val     => set_val_bsd_s0,
      PSI0        => PSI0_0,
      S_I         => S0_I,
      S_SC        => S0_SC,
      Start_SP    => Start_SP_0);
   drivectl_v4_slot1_0 : drivectl_v4_slot1
```

```
      GENERIC MAP (
         subscribers => subscribers,
         Data_Length => Data_Length)
      PORT MAP (
         Reset_O_1   => reset_O_1,
         T_O_1       => T_O_1,
         act_val     => act_val_bsd_s1,
         act_val_ref => act_val_ref_bsd_s1,
         chk_X       => chk_X_s1,
         clk         => clk,
         set_val     => set_val_bsd_s1,
         PSI0        => PSI0_1,
         S_I         => S1_I,
         S_SC        => S1_SC,
         Start_SP    => Start_SP_1);
   bufg_0 : BUFG
      PORT MAP (
         O => clk_in_int,
         I => clk_in);
   bufg_1 : BUFG
      PORT MAP (
         O => clk,
         I => clk_dv);
   bufgmux_0 : BUFGMUX
      PORT MAP (
         O  => clk_200MHz,
         I0 => L_fixed_bufgmux,
         I1 => clk_2x,
         S  => LOCKED);
   DCM_1 : DCM_BASE
      GENERIC MAP (
         CLKDV_DIVIDE          => 2.5,
         CLKFX_DIVIDE          => 1,
         CLKFX_MULTIPLY        => 4,
         CLKIN_DIVIDE_BY_2     => FALSE,
         CLKIN_PERIOD          => 10.0,
         CLKOUT_PHASE_SHIFT    => "NONE",
         CLK_FEEDBACK          => "1X",
         DCM_PERFORMANCE_MODE  => "MAX_RANGE",
         DESKEW_ADJUST         => "SYSTEM_SYNCHRONOUS",
         DFS_FREQUENCY_MODE    => "LOW",
         DLL_FREQUENCY_MODE    => "LOW",
         DUTY_CYCLE_CORRECTION => TRUE,
         FACTORY_JF            => X"C080",
         PHASE_SHIFT           => 0,
         STARTUP_WAIT          => false)
      PORT MAP (
         CLK0     => CLK0,
         CLK180   => OPEN,
         CLK270   => OPEN,
         CLK2X    => clk_2x,
         CLK2X180 => OPEN,
         CLK90    => OPEN,
         CLKDV    => clk_dv,
         CLKFX    => OPEN,
         CLKFX180 => OPEN,
         LOCKED   => LOCKED,
         CLKFB    => CLK0,
         CLKIN    => clk_in_int,
         RST      => reset);
   ICAP_VIRTEX4_0 : ICAP_VIRTEX4
      GENERIC MAP (
         ICAP_WIDTH => "X32")
```

```
      PORT MAP (
         BUSY  => ICAP_BUSY,
         O     => OPEN,
         CE    => ICAP_CE,
         CLK   => ICAP_CCLK,
         I     => ICAP_DATA_IN,
         WRITE => ICAP_WR);

   slot_data(2) <= GC_I;
   T_O2 <= T_O(2);
   TokenNew <= GC_I;
   slot_control(2) <= GC_SC;
   Reset_comm_sys(0) <= s0_reset;
   Reset_comm_sys(1) <= s1_reset;
   Reset_comm_sys(2) <= reset;
   clk_out <= clk_dv;
   FD <= FD_out when fifodata_out_enable_inv = '0' else (others => 'Z');
END struct;
```

C.2 Hardwaretask des rekonfigurierbaren Antriebscontrollers nach Abschnitt 6.1

VHDL-Quelltext des Hardwaretasks für den Hauptbetriebszustand mit bitseriellem Subtraktionsglied und PI-Regler (ohne Teilnehmeranschluss für das Kommunikationssystem):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;
-- ...

ENTITY drivectl_v4_task IS
   PORT(
      clk         : IN     std_logic;
      reset       : IN     std_logic;
      act_val     : IN     std_logic;    -- process
      act_val_ref : IN     std_logic;
      chk_X       : IN     std_logic;
      set_val     : IN     std_logic;
      DataSent    : IN     std_logic;    -- Communication System
      StartSP     : IN     std_logic;
      DataOut     : OUT    std_logic_vector (data_length - 1 DOWNTO 0);
      PSO         : OUT    std_logic;
      PendData    : OUT    std_logic;
      Tmarker     : OUT    std_logic_vector (data_length - 1 DOWNTO 0));
END drivectl_v4_task ;

ARCHITECTURE pi_ctl OF drivectl_v4_task IS

   constant TMLength : integer := Tmarker'length;
   constant denom : integer := 5;
   constant K_P : integer := integer(round(1.5*2.0**denom));
   constant K_I : integer := integer(round(0.6*2.0**denom));
   constant diff_d : integer := DelaySub (word_length, 0, 0, 0);
   constant pi_d : integer := DelayPI (word_length,diff_d,denom,K_P,K_I);
   -- Internal signal declarations
   -- ...

   -- Component Declarations
   COMPONENT genchk_reconf
```

```
   -- ...
   COMPONENT pi
   -- ...
   COMPONENT s2p
   -- ...
   COMPONENT sub
   -- ...
   COMPONENT bs_end
   -- ...

BEGIN
   -- Instance port mappings.
   genchk_reconf_0 : genchk_reconf
      PORT MAP (
         Clk       => clk,
         ChkIn_0   => chk_X,
         Reset     => reset,
         BM_enable => OPEN,
         ChkOut    => chk);
   pi_0 : pi
      GENERIC MAP (
         K_P      => K_P,
         K_I      => K_I,
         denom    => denom,
         datain_d => diff_d)
      PORT MAP (
         chk         => chk,
         clk         => clk,
         datain_bsd  => diff_bsd,
         reset       => not_startSP,
         dataout_bsd => ctl_bsd);
   s2p_0 : s2p
      GENERIC MAP (
         datain_d => 0,
         Teiler   => 1)
      PORT MAP (
         datain_bsd => act_val_ref,
         Clk        => clk,
         Reset      => reset,
         chk        => chk,
         ov_out     => OPEN,
         DataParOut => act_val_ref_par);
   sub_0 : sub
      GENERIC MAP (
         minuend_d   => 0,
         subtr_d     => 0,
         SIGND       => 1,
         ERROR_CHECK => 0)
      PORT MAP (
         minuend_bsd => set_val,
         subtr_bsd   => act_val_ref,
         Clk         => clk,
         Reset       => not_startSP,
         chk         => chk,
         diff_bsd    => diff_bsd);
   bs_end_0 : bs_end
      GENERIC MAP (
         word_length => word_length,
         datain_d    => pi_d,
         to_sync     => 0)
      PORT MAP (
         DataIn_bsd => ctl_bsd,
         Clk        => clk,
```

```
        Reset      => reset,
        dataout    => PSO);

   TMarker <= std_logic_vector (to_unsigned(TMT0,TMLength));
   DataOut <= std_logic_vector (to_unsigned(TMT1,TMLength));
   not_startSP <= not startSP;

   finish : process (clk, reset)
   begin  -- process finish
     if reset = '1' then
       finished    <= '0';
     elsif clk'event and clk = '1' then
         if StartSP = '1' and act_val = '1'
            and act_val_ref_par >= limit then
           finished <= '1';
         else
           finished <= '0';
         end if;
     end if;
   end process finish;

   send_tmarker: process (clk, reset)
   begin  -- process send_tmarker
     if reset = '1' then
       PendData <= '0';
     elsif clk'event and clk = '1' then
       if finished = '1' then
         PendData <= '1';
       end if;
       if DataSent = '1' then
         PendData <= '0';
       end if;
     end if;
   end process send_tmarker;
END pi_ctl;
```

C.3 Testbench des rekonfigurierbaren Antriebscontrollers nach Abschnitt 6.1

VHDL-Quelltext der Testbench für Verhaltens- und Timingsimulationen:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;
-- ...

ENTITY drivectl_v4_tb IS

END drivectl_v4_tb ;

ARCHITECTURE struct OF drivectl_v4_tb IS

   -- file input reading
   constant FileName : string := "configuration_data.txt";
   file InFile       : text open read_mode is FileName;
   constant clk_time_40MHz : time := 12.5 ns;
   -- Internal signal declarations
   -- ...
   SIGNAL ctl_val_dig       : std_logic_vector(word_length - 2 DOWNTO 0);
   SIGNAL ctl_val_real      : real;
```

```
   SIGNAL rpm               : real; -- revolutions per minute

   -- Component Declarations
   COMPONENT drivectl_v4_tl
   -- ...
   END COMPONENT;
   COMPONENT real2enc
   -- ...
   COMPONENT ClockSource
   -- ...
   COMPONENT ResetSource
   -- ...
   COMPONENT dac
   -- ...
   COMPONENT pt1
   -- ...

BEGIN
   -- Instance port mappings.
   drivectl_v4_tl_0 : drivectl_v4_tl
      PORT MAP (
         EMPTY             => EMPTY,
         FULL              => FULL,
         act_val_enc       => act_val_enc,
         clk_in            => clk_100MHz,
         reset             => reset,
         set_val           => set_val,
         FA                => FA,
         PKTEND            => PKTEND,
         SLOE              => SLOE,
         SLRD              => SLRD,
         SLWR              => SLWR,
         USB_RESET         => USB_RESET,
         clk_out           => clk_40MHz,
         ctl_val_bpd       => ctl_val_bpd,
         ctl_val_pwm       => ctl_val_pwm,
         FD                => FD);
   enc_0 : real2enc
      PORT MAP (
         clk         => clk_40MHz,
         reset       => reset,
         rpm         => rpm,
         act_val_enc => act_val_enc);
   clk_0 : ClockSource
      GENERIC MAP (
         CLOCK => 100 MHz)
      PORT MAP (
         CLk => clk_100MHz);
   rst_0 : ResetSource
      GENERIC MAP (
         INIT_TIME => 100 ns)
      PORT MAP (
         reset => reset);
   dac_0 : dac
      GENERIC MAP (
         par_ww => word_length - 1,
         umin   => -16384,
         umax   => 16383,
         ymin   => -16384.0,
         ymax   => 16383.0,
         init   => 0.0)
      PORT MAP (
         u => ctl_val_dig,
```

```
      y => ctl_val_real);
pt1_0 : pt1
   GENERIC MAP (
      T    => 16 ms,
      init => 0.0,
      K    => 1.0)
   PORT MAP (
      u => ctl_val_real,
      y => rpm);

handshakes : process
begin  -- process handshakes: emulates Cypress EZ USB FX2
--task 0
  wait for 1012.5 ns;           --FA changes
  FULL  <= '0';                 --fifo read
  EMPTY <= '1';
  wait until SLRD = '0';        --wait until start of read sequence
  wait for (41092+3)*100 ns;    --read
  wait for 60 ns;               --assert fifo flags after last SLRD
  FULL  <= '1';                 --fifo empty, data stored in BRAMs
  EMPTY <= '0';
  wait for 1024*100*2 ns;       --wait for ram2ram swap to complete
--task 1 ditto
--task 2 ditto
--task 3 ditto ...
--receive 1KB data and set slot_enable
  wait for 1000 ns;
  FULL  <= '0';                 --fifo read
  EMPTY <= '1';
  wait until SLRD = '0';        --wait until start of read sequence
  wait for 1023*100 ns;         --read 1KB of data
  wait for 60 ns;               --assert fifo flags after last SLRD
  FULL  <= '1';                 --fifo empty, data stored in BRAMs
  EMPTY <= '0';
  -- receive more data ...
end process;

ReadFile : process (SLRD, clk_40MHz)
  constant ERROR_MSG : string := "Error reading file: " & FileName;
  variable bv        : bit_vector (7 downto 0);
  variable lin       : line;
  variable good      : boolean;
begin
  if reset = '1' then
    FD <= (others => 'Z');
  elsif (SLRD'event and SLRD = '0') then
    readline (InFile, lin);
    read (lin, bv, good);
    assert good report ERROR_MSG severity failure;
    FD <= To_StdLogicVector(bv) after 14.3 ns;
  elsif FA = "10" then
    FD <= (others => 'Z');
  end if;
end process;

setval: process is
begin  -- process setval
   -- set set_vals here
end process setval;

ctl2ctl : process (clk_40MHz, reset) is
begin
   if clk_40MHz'event and clk_40MHz = '1' then
```

```
       if ctl_val_bpd(14) = '1' then
         ctl_val_dig <= (others => '0');
       else
         ctl_val_dig <= ctl_val_bpd;
       end if;
     end if;
   end process ctl2ctl;
END struct;
```