Programmierung von FPGA-Prozessorsystemen mittels aktiver Komponenten

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Stephan Rühl
aus Heidelberg

März 2001

Dekan: Profesor Dr. Guido Moerkotte, Universität Mannheim

Referent: Prof. Dr. Reinhard Männer, Universität Mannheim

Korreferent: Prof. Dr.-Ing. Norbert Wehn, Universität Kaiserslautern

Tag der mündlichen Prüfung: 16.05.2001

ZUSAMMENFASSUNG

Die ständig steigenden Leistungsanforderungen an Computersysteme führen immer wieder dazu, daß für gewisse Anwendungsfälle die Rechenleistung universeller Mikroprozessor-Systeme nicht mehr ausreichend ist. Bekannte Beispiele hierfür sind die 3D-Bilderzeugung, industrielle Bildverarbeitung, aber auch die Hochenergiephysik mit ihren Trigger-Anwendungen. Einen Ausweg aus dieser Situation bieten FPGAs. Diese Bausteine können ähnlich wie ASICs digitale Schaltungen implementieren und verfügen über vergleichbare Leistungsmerkmale, können aber mehrmals umprogrammiert werden.
FPGAs treten in den relevanten Anwendungsgebieten nicht isoliert auf, sondern finden in Systemen Verwendung, die Zusatzbausteine wie Speicher, CPUs, DSPs und I/O-Bausteine verwenden. Um nicht für jeden Anwendungsfall ein eigenes System entwerfen zu müssen, werden universelle Systeme entwickelt, die für einen weiten Bereich von Anwendungen geeignet sind. Diese Systeme bezeichnet man auch als FPGA-Prozessoren.
Obwohl diese Systeme eine hohe Leistungsfähigkeit und Flexibilität aufweisen, gestaltet sich ihre Programmierung sehr aufwendig. Die Ursache hierfür liegt darin, daß für die verschiedenen Bestandteile des FPGA-Prozessors nicht nur verschiedene Entwurfssprachen, sondern vollständig verschiedene Programmieransätze verwendet werden. Zusätzlich müssen die Schnittstellen zwischen den einzelnen Systembestandteilen implementiert werden. Dies führt zu einem aufwendigen Entwurfsprozeß und setzt weitreichende Kenntnisse beim Programmierer über verschiedene Programmierverfahren und das verwendete Zielsystem voraus.
Ziel der vorliegenden Arbeit ist es, die Programmierung von FPGA-Prozessoren zu vereinfachen und somit kürzere Entwurfszeiten zu ermöglichen und die Programmierung dieser Systeme einem weiteren Anwenderkreis zugänglich zu machen.
Hierzu werden aktive Komponenten eingeführt, das darauf basierende Programmierverfahren vorgestellt, ein entsprechendes Programmiersystem entwickelt und die Leistungsfähigkeit dieses Systems an Hand von Problemlösungen im Bereich der Bildverarbeitung evaluiert.
Aktive Komponenten stellen Funktionen bereit, auf die über Ports zugegriffen werden kann. Ein Port transportiert hierbei die zur Ausführung der Funktion benötigten Daten bzw. Steuerinformationen. Anwendungen lassen sich einfach dadurch erstellen, daß die benötigten Komponenten ausgewählt und in geeigneter Weise verbunden werden. Die Komponenten handeln untereinander die Kommunikationsmechanismen für ihre Ports aus und beziehen hierin auch die Eigenschaften des gewählten FPGA-Prozessors ein. Jede aktive Komponente erstellt eine für den vorgegebenen Kontext und den FPGA-Prozessor optimierte Implementierung ihrer Funktion. In die Implementierung werden alle Bestandteile des FPGA-Prozessors einbezogen.
Die vorgelegten Ergebnisse führen zu dem Schluß, daß mit aktiven Komponenten der Entwicklungsaufwand für Anwendungen auf FPGA-Prozessoren deutlich reduziert werden kann, ohne daß hierdurch bedeutsame Nachteile im Bezug auf den Ressourcenbedarf und die Verarbeitungsleistung der erstellten Implementierung verursacht werden.

Inhalt

1

FPGA-Prozessoren Aufbau, Eigenschaften und Leistungsmerkmale

Bei FPGA-Prozessoren handelt es sich um eine relativ neue Rechnerarchitektur, die sich hauptsächlich dadurch auszeichnet, daß der Aufbau und die Funktionsweise der verwendeten Recheneinheit für einen gegebenen FPGA-Prozessor in weiten Grenzen frei wählbar ist. Dies wird dadurch erreicht, daß an Stelle einer fixen Implementierung des Rechenwerkes ein frei programmierbarer Baustein, das sogenannte FPGA (Field Programmable Gate Array), eingesetzt wird. Hieraus ergeben sich neue Möglichkeiten zur effizienten Implementierung von Problemlösungen, da eine geeignete Prozessorarchitektur für die jeweilige Aufgabenstellung gewählt werden kann.

Dieses Kapitel beginnt mit einer näheren Beschreibung von FPGAs (1.1), erläutert dann weitere Architekturmerkmale von FPGA-Prozessoren (1.2) und schließt mit einer Betrachtung der Aufgabenbereiche, für die diese Rechnerarchitektur sinnvoll eingesetzt werden kann (1.3) .

1.1 Aufbau von FPGAs

Wie in Bild 1.1 dargestellt, besteht ein FPGA im wesentlichen aus folgenden Teilen:

- Logik-Matrix
- I/O-Blöcke
- Vebindungsnetzwerk
- Konfigurationsspeicher

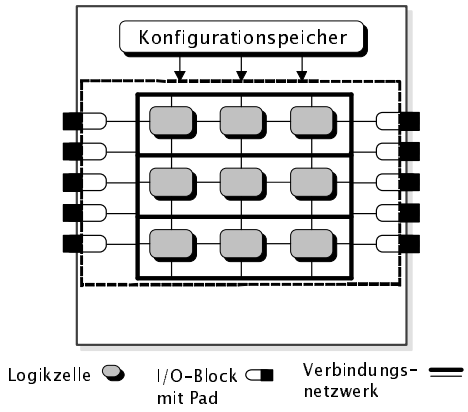

Bild 1.1. Prinzipieller Aufbau eines FPGA's

Die *Logik-Matrix* ist aus gleichartigen Zellen aufgebaut, die in der Lage sind, beliebige Logikgleichungen zu implementieren und Speicherfunktionen zu übernehmen. Die Anzahl der Signale, die innerhalb einer Logik-Zelle verknüpft werden können, ist typischerweise auf fünf beschränkt.

I/O-Blöcke stellen die Verbindung zu den externen Anschlüssen (Pins) dar und erlauben es, für diese gewisse Einstellungen vorzunehmen, die die Verwendung und Eigenschaften der Anschlüsse näher bestimmen. So kann festgelegt werden, ob ein Anschluß als Eingang, Ausgang oder bidirektionalen betrieben wird, welches Schaltverhalten er aufweist, ob das anliegende Signal als Taktsignal verwendet werden soll, etc. . Diese Einstellungsmöglichkeiten variieren allerdings je nach verwendetem FPGA-Typ.

Das *Vebindungsnetzwerk* besteht aus programmierbaren Verbindungen der Zellen der Logik-Matrix untereinander und mit den I/O-Blöcken.

Der *Konfigurationsspeicher* legen durch die, in ihm gespeicherten Daten sowohl das Verhalten der Zellen der Logik-Matrix, der I/O-Blöcke als auch des Verbindungsnetzwerkes fest und definiert somit die im FPGA zu implementierende Funktionalität . Dieser Speicher ist zumeist in SRAM- oder EEPROM-Technologie ausgeführt. Für FPGA-Prozessoren ist die SRAM-Variante am bedeutendsten, da sie die einfachste und schnellste Möglichkeit zur Neukonfiguration des Bausteines bietet.

Um eine Digitalschaltung auf einem FPGA zu implementieren wird diese auf die Logikzellen verteilt, geeignete Verbindungen zwischen den Logikzellen gewählt und daraus der Inhalt des Konfigurationsspeichers errechnet. Programme, die diese Aufgabe übernehmen, werden zumeist von den Herstellern der FPGAs bereitgestellt. Da es für die Verbindung zweier Punkte innerhalb des FPGAs mehrere Verbindungswege mit unterschiedlichem Zeitverhalten gibt und die geeigneten Pfade durch die Herstellersoftware ausgewählt werden, können über die Laufzeit von Signalen im voraus keine genauen Angaben gemacht werden. Aus diesem Grund werden auf FPGAs hauptsächlich taktsynchrone Schaltungen reali-

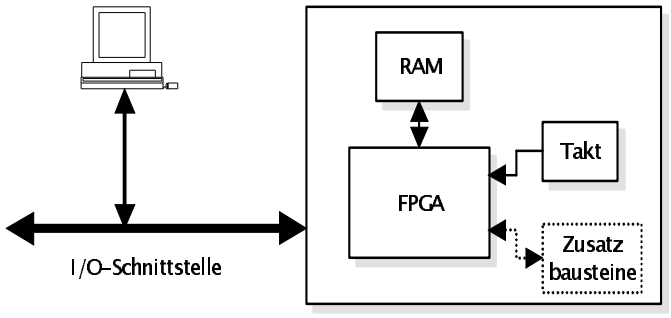

Bild 1.2. Prinzipieller Aufbau eines FPGA-Prozessors

siert, da hier, je nach maximaler Signallaufzeit, lediglich der Arbeitstakt angepaßt werden muß, um die korrekte Funktion der Implementierung zu gewährleisten.

Für weiter Informationen zu Aufbau und Funktion von FPGAs siehe [54] und [57].

1.2 Aufbau von FPGA-Prozessoren

Obwohl sich FPGA-Prozessoren je nach Anwendungsbereich sehr stark in ihrem Aufbau unterscheiden, weisen sie dennoch einige gemeinsame Grundmerkmale auf, die im Folgenden näher erläutert werden sollen.

Das eigentliche *Rechenwerk* eines FPGA-Prozessors besteht aus einem einzelnen oder mehreren untereinander verbunden FPGAs. Diese Verbindungen können festverdrahtet, oder über Crossbar-Switches frei verschaltbar ausgeführt sein. Die geeignete Topologie ist stark vom vorgesehenen Einsatzgebiet des FPGA-Prozessors abhängig und variiert somit von Modell zu Modell. Dem Rechenwerk zugeordnet sind zumeist noch RAM-Bausteine, die um Speichern größerer Datenmengen, oder aber als Lookup-Tabellen (Wertetabellen) eingesetzt werden. Während bei Systemen mit nur einem FPGA die RAM-Ausstattung lediglich in der Anzahl, Breite und Tiefe der RAM-Blöcke[1] variiert, stellt bei Systemen mit mehreren FPGAs, die Zuordnung der RAM-Blöcke zu den einzelnen FPGAs ein wesentliches Architekturmerkmal dar. Als zusätzliche Variante können hier die RAM-Böcke nicht nur einem, sondern mehreren FPGAs zugeordnet werden, wodurch sie als Bindeglied zur Datenkommunikation zwischen den FPGAs in Form von LUT's oder Zwischenspeichern genutzt werden können. Diese Anordnung läßt sich für viele Problemstellungen zu sehr effektiven Implementierungen nutzen [35].

[1] Mit RAM-Block ist hier ein einzelner oder mehrere RAM-Bausteine gemeint, die einen zusammenhängenden Speicherbereich implementieren und über einen vollständigen und exklusiv für diesen Block benutzten Satz an Steuer, Adress- und Datensignalen verfügen. Aus dieser Festlegung ergibt sich die Eigenschaft, daß auf verschiedene RAM-Blöcke parallel und unabhängig von einander zugegriffen werden kann.

Ein weiterer Bestandteil von FPGA-Prozessoren ist die *I/O-Schnittstelle*, die die Verbindung zur Außenwelt herstellt, die zu verarbeitenden Daten annimmt und die errechneten Ergebnisse weiterreicht. Ein eigenständiger Teil der I/O-Schnittstelle ist das *Host-Interface*, das die Verbindung zu einem Steuerrechner herstellt. Seine Hauptaufgabe besteht darin, den FPGA-Prozessor zu initialisieren, kann aber auch dazu verwendet werden, den aktuellen Zustand des FPGA-Prozessors zu überwachen oder Ergebnisse über den Hostrechner zu visualisieren oder weiterzuverarbeiten.

Zu der Basisausstattung eines FPGA-Prozessors gehören auch Bausteine zur Takterzeugung, die einen synchronen Betrieb der FPGAs ermöglichen. Diese Bausteine liefern meist mehrere unterschiedliche Taktsignale, deren Frequenz in den meisten Fällen programmierbar ist.

Erweitert wird diese Funktionalität bei einigen Systemen noch durch Standard-Bausteine wie diskrete FIFOs aber auch durch komplexere Bausteine wie Mikrocontroller und Signalprozessoren [21].

1.3 Anwendungsgebiete und geeignete Algorithmustypen

Während in den vorhergehenden Abschnitten der Aufbau von FPGA-Prozessoren dargelegt wurde, soll nun das Augenmerk auf mögliche Anwendungsgebiete für diese Rechnerarchitektur gelenkt werden. Denn obwohl sich auf einem FPGA-Prozessor, bei ausreichenden Logik-Ressourcen und genügend langsamen Takt, jede synchrone Schaltung realisieren läßt [53], gibt es nur eine eingeschränkte Klasse von Anwedungsbereichen, bei denen durch die Verwendung von FPGA-Prozessoren eine leistungsfähigere Realisierung als auf herkömmlichen Systemen möglich ist [39]. FPGA-Prozessoren treten hierbei je nach Anwedungsbereich in Konkurrenz zu Mikroprozessoren und DSPs[2]. Im Folgenden werden einige repräsentative Applikationen vorgestellt und daraus Kriterien abgeleitet, die Algorithmen erfüllen müssen, um auf FPGA-Prozessoren gewinnbringend realisiert werden zu können.

Im Bereich der *Kryptologie* wurde das RSA-Kryptosystems [42] erfolgreich auf FPGA-Prozessoren implementiert. Der Hauptberechnungsaufwand liegt bei diesem Algorithmus darin, daß mehrere Multiplikationen und Modulo-Operationen auf Binärzahlen mit großer Stellenanzahl auszuführen sind. Die in [53] beschriebene Realisierung benutzte hierbei Variablen mit bis zu 1000 Binärstellen und war dabei um eine Größenordnung schneller, als die schnellste bis dahin bekannte Implementierung.

Auch bei der *Strukturermittlung von Protein-Molekülen* wurden durch die Verwendung von FPGA-Prozessoren beachtliche Beschleunigungswerte erreicht [46].

[2] Natürlich treten FPGA-Prozessoren in diesem Bereich auch in Konkurrenz zu anwedungsspezifischen Bausteinen wie ASICS . Da im Rahmen dieser Arbeit aber nur rekonfigurierbare Systeme behandelt werden sollen, bleibt dieser Bereich unberücksichtigt.

Eine Möglichkeit, die geometrische Struktur eines solchen Moleküls zu ermitteln, besteht darin, für die möglichen räumlichen Anordnungen den jeweiligen Energiegehalt zu errechnen. Die Wahrscheinlichkeit, daß die aktuell betrachtete Anordnung der tatsächlichen Struktur des Moleküls entspricht, ist um so höher , je geringer die Energie der Molekülausrichtung ist. Solch ein Algorithmus konnte durch die Verwendung von FPGAs um mehr als den Faktor 30 beschleunigt werden.

Ein weiteres Anwendungsgebiet liegt im Bereich der *Hochenergiephysik*. In Elementarteilchenbeschleunigern werden Bewegungsdaten von Elementarteilchen gemessen. Aus diesen Informationen sollen die Flugbahnen der Teilchen bestimmt werden. Hierbei treten Datenraten auf, die bis in den Bereich von 60 TByte pro Sekunde reichen können [3]. Da diese Daten weder in Echtzeit bearbeitet, noch vollständig gespeichert werden können, müssen andere Wege beschritten werden. Meist sollen nur Spuren von Teilchen mit bestimmten Eigenschaften ermittelt werden. Da die zur Bestimmung dieser Flugbahnen notwendigen Daten nur einen sehr kleinen Teil des gesamten Datenvolumens ausmachen, versucht man anhand relativ einfacher Kriterien eine Vorauswahl der potenziell interessanten Daten zu treffen. Hierdurch kann die Anzahl der intensiv zu prüfenden Spuren drastisch reduziert werden, so daß die Anforderungen an die darauffolgenden Verarbeitungsstufe deutlich reduziert werden können. Für diese so genannten Trigger, liegen mehrere Implementierungen auf FPGA-Prozessoren vor [39][37]. Sie erreichen dabei eine Beschleunigung der hierfür verwendeten Algorithmen um ca. zwei Größenordnungen [37].

Der Bereich der *Bild- und Signalverarbeitung* stellt ein weiteres interessantes Anwendungsgebiet für FPGA-Prozessoren dar. So lassen sich unter anderem verschiedene Filter (FIR- und IIR-Filter [11][29][13], adaptive Filter [2], Median-Filter [52]), 1D-FFT [47], 2D-FFT[45], Bildsegmentierung[4], Bildkompression [53]und die Hough-Transformation[1] gewinnbringend auf FPGA-Prozessoren realisieren. Die hierbei angegebenen Beschleunigungswerte liegen meist zwischen 10 und 100, wobei zum Vergleich hauptsächlich DSPs und Workstations herangezogen werden.

Eine wesentliche Frage ist, welche Eigenschaften potentielle Algorithmen erfüllen müssen, um auf FPGA-Prozessoren die gewünschte Leistungssteigerung erreichen zu können. Eine wichtige Feststellung hierbei ist, daß die hier vorgestellten Bespiele meist mit einem Takt von 30MHz – 100MHz betrieben werden, wohingegen entsprechende Hochleistungs-CPUs mittlerweile schon im Bereich von 1 GHz und mehr arbeiten. Weshalb FPGA-Prozessoren dennoch in den oben aufgeführten Fällen eine größere Leistungsfähigkeit aufweisen, läßt sich an den folgenden Punkten festmachen:

1. Hohe Parallelisierbarkeit

   Eine bemerkenswerte Eigenschaft von FPGAs ist, daß das Maß der parallel ausführbaren Prozesse hauptsächlich durch deren Komplexität begrenzt

ist[3]. Ursächlich bedingt ist dies dadurch, daß innerhalb eines FPGAs alle Vorgänge parallel ablaufen und somit beliebig viele gleichzeitig aktive Rechenwerke realisiert werden können, solange die Logik-Ressourcen des FPGAs nicht erschöpft sind.

2. Pipeline orientierte Implementierung

   Innerhalb einer Pipeline durchlaufen die Daten hintereinander geschaltete Rechenstufen. Werden diese Operationen nacheinander auf alle Daten eines Datenstroms angewendet, sind idealerweise alle Rechenstufen gleichzeitig in Betrieb. Dies führt bei einer Pipeline mit n Stufen zu einer maximalen Beschleunigung von n, gegenüber einer einstufigen Implementierung[27]. Bei den oben aufgeführten Beispielen treten meist zwei Sonderfälle auf. Zum einen handelt es sich zumeist um einfache Pipelines, in dem Sinne, daß die Verarbeitungsschritte unabhängig von den verarbeiteten Daten sind und somit nur eine minimale Steuerlogik benötigt wird. Zum Anderen liegen in vielen Fällen systolische Pipelines vor [27]. Systolische Pipelines liefern in jedem Verarbeitungstakt ein neues Ergebnis und ermöglichen somit eine maximale Ausnutzung der Pipelinestufen und den größtmöglichen Datendurchsatz innerhalb der Pipeline. In vielen Fällen können auch mehrere identische Pipelines parallel betrieben werden, was den Datendurchsatz weiter erhöht.

3. Unübliche Datentypen bzw. Operationen

   FPGA-Prozessoren zeigen sich recht flexibel in der Verarbeitung unüblicher Datentypen. Hierbei kann es sich um eine ungewöhnliche Codierung der Information oder eine unübliche Anzahl der verwendeten Binärstellen handeln. Da die Rechenwerke und auch die Datenpfade speziell für den jeweiligen Datentyp entworfen werden können, werden hier weder Bandbreite noch Ressourcen vergeudet. Auch können Rechenwerke für spezielle Operationen entworfen werden, die bei Universal-CPUs durch mehrere Befehle realisiert werden müßten, was gerade mit entsprechenden Pipeline orientierten Implementierungen für hohe Datendurchsätze sorgt.

[3] Zusätzlich kommt in einigen Fällen noch eine Begrenzung durch die verfügbare I/O-Bandbreite der FPGAs hinzu, so daß die maximal erreichbare Parallelisierung in diesen Fällen dadurch eingeschränkt wird, daß die Rechenwerke nicht ausreichend mit Daten versorgt werden können.

2
Programmierung von FPGA-Prozessoren

Der Anspruch von FPGA-Prozessoren, nicht als rekonfigurierbare Hardware, sondern als eigenständige Rechnerarchitektur verstanden zu werden, beruht im wesentlichen darauf, daß sie wie gewöhnliche Computer benutzt und programmiert werden können.

Dieses Kapitel befaßt sich mit der Programmierung von FPGA-Prozessoren, wobei zuerst die Anforderungen an eine geeignete Programmierumgebung bestimmt und dann mit den Eigenschaften bereits existierender Ansätze verglichen werden.

2.1 Anforderungen an ein Programmiersystem

Die Gesamtheit aller Anforderungen an ein Programmiersystem laufen darauf hinaus, die Realisierung fehlerfreier, möglichst leistungsfähiger Applikationen mit geringem Aufwand realisierbar zu machen.

Der erste Punkt ist hierbei die Art der Problembeschreibung bzw. die Abstraktionsebene, auf der diese Beschreibung erfolgt. Während sich die ersten Programmiersprachen von ihrer Beschreibungsebene sehr stark an die Struktur des zu programmierenden Systems anlehnten, um damit eine möglichst leistungsfähige Implementierung zu erzielen, dies aber mit einem sehr hohen Implementierungsaufwand erkauften, setzt sich bis heute die Tendenz fort, die Beschreibungsebene immer weiter von der Implementierungsebene weg, zur Problemebene hin zu verschieben. Die Ideale Abstraktionsebene ist hierbei erreicht, wenn die Beschreibung auf der domänenspezifischen Abstraktion erfolgt. Dies ermöglicht es dem Entwickler, die Problemlösung mit den Ausdrucks- und Sprachmitteln der Disziplin zu beschreiben, die in dem Fachgebiet, zu dem die Problemstellung zählt,

üblich sind. Die Abstraktionsebene ist somit ein wichtiger Punkt, um die Problembeschreibung zu vereinfachen.

Für Anwendungsbereiche in denen Teillösungen in vielen Problemstellungen auftreten, wird die Wiederverwendbarkeit bereits erstellter Lösungsbeschreibungen relevant. Hierbei wird die Forderung gestellt, daß sich Problemlösungen so formulieren lassen, daß sie in möglichst vielen anderen Fällen verwendet, bzw. leicht an diese Fälle angepaßt werden können. Ein Punkt, der die Wiederverwendbarkeit zumindest im weiteren Sinne betrifft, ist die Portabilität. Sie bezeichnet die Möglichkeit, aus der Beschreibung einer Problemlösung, mit möglichst geringem Aufwand, Implementierungen für verschiedene Zielsysteme zu erstellen. Wünschenswert ist hierbei, daß bei einer Übertragung von einem System auf ein anderes leistungsfähigeres System automatisch auch die Leistungseigenschaften dieses Systems weitestgehend ausgenutzt werden.

Der zweite Punkt betrifft die Umwandlung der Beschreibung in die auf dem Zielsystem ausführbare Implementierung. Um eine möglichst leistungsfähige Implementierung zu ermöglichen, müssen Optimierungsschritte durchgeführt werden. Diese Optimierungen gliedern sich in die Ebenen der domänenorientierten Optimierungen, der Hochsprachen-Transformation ("High-Level Transformation"), und der zielmaschinenorientierten Optimierung.

Bei der domänenorientierten Optimierung werden Kenntnisse über die Struktur der Anwendungsdomäne benutzt, um eine möglichst effiziente Implementierung zu erhalten. Da hierbei ein geeigneter Algorithmus für das gegebene Problem ausgewählt wird, können hohe Effizienzsteigerungen erzielt werden. Die Aufnahme dieser Optimierungen in das Programmiersystem ist sehr aufwendig und auf die jeweilige Anwendungsdomäne beschränkt. Grundvoraussetzung für die Anwendung derartiger Optimierungen ist, daß die Lösungsbeschreibung auf ausreichend hoher Abstraktionsebene, der Domänenebene, vorliegt.

Hochsprachen-Transformationen setzen auf der Ebene der Beschreibungssprache auf und versuchen durch Restrukturierung der Beschreibung eine höhere Systemleistung zu erzielen. Hierbei werden z.B. unnötige Arbeitsschritte entfernt bzw. die Befehlsreihenfolge so modifiziert, daß die Abhängigkeit zwischen den einzelnen Operationen abnimmt und somit die Parallelisierbarkeit erhöht wird [56].

Der Schritt des Allocation/Scheduling [31]bestimmt, welche Verarbeitungsschritte gleichzeitig ausgeführt werden und legt die Anzahl und den Typ der benötigten Recheneinheiten fest.

Die Optimierung auf Maschinenebene hingegen, betrachtet, wie vorgegebene Verarbeitungsschritte möglichst effizient auf dem Zielsystem realisiert werden können.

Für inhomogene Systeme wie FPGA-Prozessoren tritt noch die Forderung hinzu, daß die Implementierung auch die unterschiedlichen Systembestandteile wie FPGA, DSP und Hostrechner zur Realisierung der Implementierung nutzt.

2.2 Existierende Ansätze

Zur Programmierung von FPGA-Prozessoren werden verschiedene Sprachen benutzt, die zum Teil speziell für diesen Anwendungsbereich entworfen bzw. aus dem Bereich der Hardwarebeschreibsprachen übernommen wurden.

Eine erste Gruppe umfaßt die *strukturorientierten HDLs* (Hardware Description Language). Die mit ihnen erstellbaren Beschreibungen benutzen Module fester Funktion und Schnittstellen. Zur Implementierung der gewünschten Funktionalität werden die geeigneten Module ausgewählt und so miteinander verbunden, daß sich eine strukturelle Beschreibung der Problemlösung ergibt.

Zur Realisierung von Verbindungen stehen einzelne Signale oder Signalbusse bereit. Basismodule sind Bausteine wie Logikgatter und Flip-Flops, die direkt auf dem FPGA implementiert werden können.

Aus diesem Zusammenhang ergibt sich, daß die Beschreibung faßt vollständig mit der Implementierung übereinstimmt, da sowohl die Module als auch die Verbindungen ohne weitere Umwandlungsschritte auf das FPGA abgebildet werden können. Dies hat für den Entwickler den Vorteil, daß er volle Kontrolle über die erzeugte Implementierung hat, erhöht aber gleichzeitig den Entwicklungsaufwand, da die Beschreibung unmittelbar auf der Implementierungsebene erfolgt und so alle wesentlichen Optimierungen vom Entwickler vorzunehmen sind.

Zur Strukturierung des Entwurfs und zur Erstellung weiterer Module sind meist Methoden zur hierarchischen Beschreibung von Modulen vorhanden, so daß neue Module aus bereits existierenden Modulen zusammengesetzt werden können, was ein gewisses Maß an Wiederverwendbarkeit ermöglicht.

Andererseits handelt es sich hauptsächlich bei den komplexeren Modulen um sehr spezielle Implementierungen, die meist nur im Bezug auf die benutzte Datenbreite parametrisierbar sind, so daß diese nur unter engen Randbedingungen eingesetzt werden können. Die Sicherstellung der Einhaltung dieser Bedingungen muß größtenteils durch den Entwickler erfolgen, da diese Programmiersysteme nicht in der Lage sind, die korrekte Bedienung der Schnittstellen zu überprüfen. Dies ist wenigverwunderlich, da diese Information nicht in den Modulen bereitgestellt wird, sondern nur implizit in der Modulimplementierung vorliegt. Die möglichen Prüfungen beinhalten so zumeist nur die Übereinstimmung der verwendeten Busbreite bzw. stellen sicher, daß ein Signal nicht von mehreren Datenquellen gleichzeitig genutzt werden.

Die Programmierumgebungen sind hierbei zumeist nur in der Lage, die Implementierung auf einem einzelnen FPGA vorzunehmen. Sollen mehrere FPGAs eines FPGA-Prozessors verwendet werden, muß die Beschreibung, gemäß der Anzahl der verwendeten FPGAs, von Hand in mehrere Teile gegliedert werden. Für die korrekte Verbindung zwischen den FPGAs und die Konfiguration des Gesamtsystems ist der Programmierer verantwortlich.

Im Gegensatz zum klassischen Ansatz, strukturelle Beschreibungen mittels Schaltplan zu erzeugen, finden bei den hier betrachteten Systemen hauptsächlich textuelle Beschreibungsverfahren ihre Anwendung. Die meisten Systeme sind

aus diesem Grund als Bibliothek auf Basis einer universellen, objektorientierten Sprache wie z.B. C++ realisiert, wobei sie sowohl Module als auch Schnittstellen auf Objekte und Verbindungen aber auf Verweise zwischen den Schnittstellenobjekten abbilden. Auf diese Weise steht die gesamte Funktionalität der zugrundeliegenden Sprache bereit, um die strukturelle Beschreibung der Problemlösung zu erstellen, was gerade bei regelmäßigen Strukturen den Erstellungsaufwand deutlich reduziert. Repräsentative Systeme sind z.B. CHDL [33] und CynLib [43].

Eine weitere Gruppe umfaßt die *verhaltensorientierten HDLs*. Bekannte Vertreter sind Verilog, VHDL [34] [31] und System C [51] [38]. Sie bieten neben strukturellen Beschreibungsverfahren auch die Möglichkeit der verhaltensorientierten Beschreibung.

Basiselement bei diesen Verfahren ist auch hier das Modul, definiert durch Schnittstellenbeschreibung (Entity) und Architektur (Architecture). Während die Schnittstellenbeschreibung die Verbindung des Moduls nach außen hin festlegt, beinhaltet die Architektur die Beschreibung der Funktionalität des Moduls in struktureller bzw. verhaltensorientierter Form. Somit ist es auch möglich, beide Beschreibungsmethoden auf Modulebene zu mischen.

Verhaltensbeschreibungen setzen sich aus Prozessen, Signalen und Variablen zusammen. Variablen dienen der Datenspeicherung und Signale zur Datenübertragung zwischen Modulen bzw. Prozessen. Mit Prozessen werden die parallelen Ausführungsstränge modelliert. Sie können verschiedene Operationen wie z.B. Zuweisungen, Schleifen, konditionelle Verzweigungen enthalten. Es sind aber auch Anweisungen vorhanden, die eine genaue Definition des Zeitverhaltens der einzelnen Operationen ermöglichen. Im allgemeinen reagieren Prozesse auf äußere Ereignisse. Zu diesem Zweck kann jedem Prozeß eine Liste von Signalen zugeordnet werden. Ändert sich der Wert eines dieser Signale wird der Prozeß aktiviert und kann die gewünschten Operationen durchführen.

Die Beschreibung auf funktionaler Ebene verringert den Implementierungsaufwand und läßt einen großen Spielraum von Optimierungen, der von Hochsprachentransformationen bishin zur Optimierung auf Maschinenebene reicht. Die domänenspezischen Optimierungen müssen aber auf Grund der verwendeten Beschreibungsebene dem Entwickler überlassen werden. Zu bemerken ist allerdings, daß die existierenden Implementierungen hauptsächlich dafür ausgelegt sind, einzelne FPGAs zu programmieren und keine Möglichkeit bieten, die anderen Systembestandteile des FPGA-Prozessors zu nutzen.

Ein anderer Ansatz versucht, *weitverbreitete Hochsprachen aus der Softwareentwicklung* zur Programmierung von FPGAs zu verwenden. Systeme wie Handel-C [14], ppC[58] und Transmogrifier-C[17] verwenden hierzu als Ausgangsbasis die weitverbreitete Programmiersprache ANSI-C.

Exemplarisch soll hier nun weiter auf Handel-C eingegangen werden. Verwendung findet hier eine sehr stark an C angelehnte Syntax, die an einigen Stellen gegenüber dem ANSI-Standard eingeschränkt und an einigen Stellen zwecks bessere Anwendbarkeit zur Hardwarebeschreibung erweitert wurde.

Die Einschränkungen betreffen zum einen die unterstützten Variablentypen. Es werden nur ganzzahlige Varianten unterstützt. Auch wurden Sprachelemente wie Zeiger, Funktionen und Aufzählungen aus dem unterstützen Sprachumfang gestrichen.

Hinzugefügt wurden Konzepte zur parallelen Ausführung von Anweisungen und die Kommunikation über sogenannte Channels. Letzter dienen dazu, Daten zwischen parallelen Ausführungssträngen auszutauschen, bzw. diese zu synchronisieren. Für die Kommunikation wird ein starres Protokoll verwendet, bei der eine Kommunikation erst zustande kommt, wenn beide Kommunikationspartner die entsprechende Sende- bzw. Empfangsanweisung aktiviert haben. Der Partner, der zuerst den Kommunikationsbefehl ausführt, wird dabei so lange blockiert, bis die Kommunikation durchgeführt werden kann.

Als weitere Einschränkung kommt hinzu, daß davon ausgegangen wird, daß für die Implementierung nur ein einziges Taktsignal verwendet wird, was spätestens bei der Ansteuerung von externen Geräten zu Problemen führt. Durch den Compiler werden Optimierungen in nur sehr geringem Maße vorgenommen, was zwar den Vorteil hat, daß der Entwickler die Art der erzeugten Implementierung sehr genau bestimmen kann, aber gleichzeitig auch die Verantwortung dafür trägt, eine Problembeschreibung zu wählen, die zu einer leistungsfähigen Implementierung führt.

Weitere Programmierverfahren stammen aus dem Bereich der *Hardware-Software Co-Synthese*. Obwohl sie von ihrem Grundansatz her, Hardware und Softwarebestandteile eines Systems einheitlich beschreiben können und erst in einem späteren Schritt die Entscheidung darüber treffen, welche Bestandteile des Programms als Hardware oder Software implementiert werden sollen, finden sie im Bereich der FPGA-Prozessoren wenig Beachtung. Dies resultiert daraus, daß die meisten dieser Programmiersysteme für Zielsysteme mit relativ eng eingegrenzten Architekturmerkmalen entworfen wurde. So ist das COSY-System [9] dafür ausgelegt SOC (System On Chip) Lösungen zu erstellen. LYCOS [36] und VULCAN [22] unterstützen lediglich Systeme, die aus einer CPU und einem ASIC aufgebaut sind. Das Cosyma-System [41] [28] gestattet zwar die Verwendung mehrere ASICs, setzt aber eine Speicherarchitektur voraus, bei der eine einzelne Speicherbank von der CPU und den ASICs gemeinsam genutzt wird. Ziel des Cosyma-Ansatzes ist es, die Leistungsfähigkeit der CPU durch eine Art Coprozessorarchitektur zu verbessern, wohingegen VULCAN daraufhin ausgelegt ist, die Größe des ASICs zu verringern, in dem geeignete Aufgaben auf die CPU ausgelagert werden. Andere Ansätze wie POLIS [10] unterstützen zwar die Aufteilung des Hardwareanteils auf mehrere FPGAs, gehen aber davon aus, daß die Verbindungsstrukturen zwischen den Bausteinen frei definiert werden können. Es werden nur Zusatzbausteine unterstützt, die direkt von der im System enthaltenen CPU angesteuert werden. Da die Ausnutzung der gegebenen Architekturmerkmale, wie die vorgegebene Verbindungsstruktur, die (verteilte) RAM-Architektur, Zusatzbausteine und I/O-Schnittstellen für die effektive Problemlösung auf FPGA-

Prozessoren unumgänglich ist, dies aber durch die Programmiersysteme nicht unterstützt wird, ist die Anwendbarkeit für FPGA-Prozessoren stark eingeschränkt.

Zusätzlich verwenden diese Systeme als Beschreibungsmittel Sprachen, die zwar die Beschreibung auf funktionaler Ebene ermöglichen, aber keine Anwendungsdomänen spezifische Beschreibung und Optimierung unterstützt. So verwenden Polis FSM-Charts (Finite State Machine) und LYCOS eine Untermenge der Sprache C.

Allen hier vorgestellten Systemen, die in der Praxis zur Programmierung von FPGA-Prozessoren eingesetzt werden, ist zu eigen, daß sie ihren Schwerpunkt auf die Programmierung von FPGAs legen und die anderen Systemkomponenten weitestgehend unberücksichtigt lassen. Da es nun aber für eine lauffähige Implementierung auf einem FPGA-Prozessor notwendig ist, alle benötigten Bestandteile des Zielsystems zu programmieren bzw. zu konfigurieren, muß dies mittels anderer Werkzeuge erfolgen. Für Mikroprozessoren existieren hierfür verschiedene Hochsprachencompiler, für andere Systemkomponenten wie Crossbar-Switches existieren entsprechende Konfigurationswerkzeuge der Bausteinhersteller. Für den Entwickler bedeutet dies nicht nur, daß er mehrere Entwicklungswerkzeuge und Beschreibungssprachen beherrschen muß, sondern resultiert auch in einem komplexen und fehlerträchtigen Entwurfszyklus (Bild 2.1).

Der Entwurfszyklus beginnt mit einer abstrakten Spezikation der zu realisierenden Problemlösung. Gebräuchliche Mittel sind hierbei formale Spezifikationssprachen, aber oft auch nur Blockdiagramme, mit denen die eigentliche Funktion des Lösungsansatzes beschrieben wird.

Im nächsten Schritt wird diese Beschreibung in einzelne Module aufgeteilt, die unabhängig voneinander implementiert werden können. Hinzu kommt noch die Festlegung, in welcher Form Daten bzw. Steuerinformationen zwischen den Modulen ausgetauscht werden. Für Module, die später in Software realisiert werden, müssen hierbei lediglich die benötigten Schnittstellenfunktionen festgelegt werden. Andere Module, die auf FPGAs implementiert werden solle, erfordern weitergehende Festlegungen. Hier muß die Schnittstelle auf Basis der verwendeten Daten- und Steuersignale und des zu verwendeten Protokolls spezifiziert werden.

Anschließend wird im Rahmen der Partitionierung für jedes Modul entschieden, auf welcher Verarbeitungseinheit des Zielsystems das jeweilige Modul realisiert werden soll. Zusätzlich muß geprüft werden, ob genügend Verbindungsressourcen bereitstehen, um die Kommunikation der Module zu ermöglichen. Gibt es hierbei Engpässe, kann entweder die Partitionierung überarbeitet werden, die Schnittstellendefinition modifiziert oder eine besser geeignete Modularisierung gewählt werden.

Nun werden die Module implementiert, wobei hierfür die Sprache Verwendung findet, die für die, dem Modul zugeordneten Verarbeitungseinheit vorgesehen ist. Stellt sich heraus, daß die Ressourcen der Verarbeitungseinheit für die Implementierung der zugeordneten Module nicht ausreichen, muß deren Implementie-

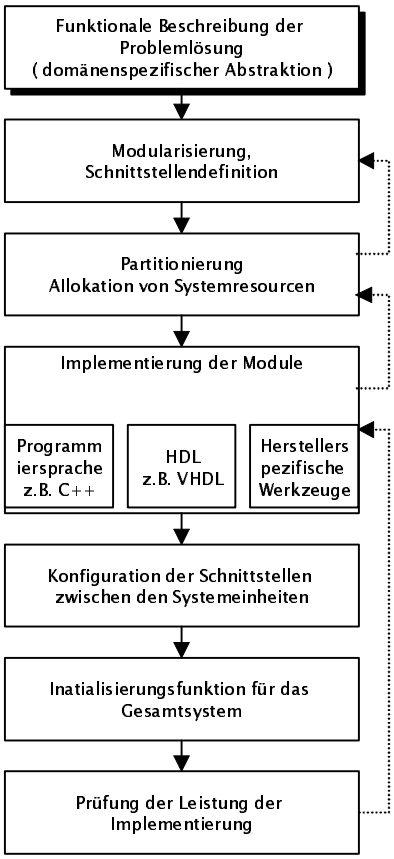

Bild 2.1. Herkömmlicher Entwurfszyklus

rung überarbeitet werden. Ist dies nicht ausreichend, muß der Entwurfszyklus in der Partitionierungs- oder sogar der Modularisierungsstufe von neuem beginnen. Wurde die Implementierung erfolgreich ausgeführt, wird die Konfiguration der Schnittstellen zwischen den Ausführungseinheiten erstellt und eine Funktion implementiert, die dazu dient das Gesamtsystem für die Ausführung der erstellten Systemlösung vorzubereiten. In diesem Stadium liegt dann die fertige Implementierung vor, für die dann noch überprüft werden muß, ob die gewünschte Systemleistung erreicht wird. Ist dies nicht der Fall, müssen einzelne Module, die Partitionierung oder sogar der ganze Entwurf überarbeitet werden.

Problematisch bei diesem Entwurfszyklus ist, daß schon in den frühen Stadien wesentliche Implementierungsmerkmale festgelegt werden, deren Eignung im Bezug auf das gewählte Zielsystem aber erst in einer wesentlich späteren Phase überprüft werden können. Dies führt wiederum dazu, daß einzelne Entwurfsschritte solange wiederholt werden müssen, bis ein geeignetes Ergebnis vorliegt. Da der Großteil dieses Aufwandes vom Entwickler selbst zu erbringen und nicht mittels der existierenden Entwicklungswerkzeuge automatisierbar ist, erhöht sich der Aufwand für die Produktentwicklung deutlich. Gleichzeitig ergeben sich hieraus

Probleme für die Portabilität des Entwurfs, da die Charakteristika des Zielsystems bereits bei der Modularisierung einbezogen werden. Die Portierung auf ein anderes Zielsystem erfordert dann meist eine aufwendige Neumodularisierung und eine erneute Durchführung der nachfolgenden Abbildungsschritte.

Auch die Wiederverwendbarkeit der Module ist gering. Für eine gegebene Funktion werden zumeist mehrere Module mit gleicher Funktionalität, aber unterschiedlichem Kommunikationsverhalten realisiert. Der Entwickler muß dann schon im Modularisierungsschritt eine Modulvariante wählen und dabei Kriterien wie verfügbare Routing- und Implementierungsressourcen und die Leistung der resultierenden Implementierung in Betracht ziehen. Stellt sich aber später heraus, daß eine ungeeignete Modulvariante gewählt wurde und wird stattdessen ein Modul gleicher Funktion aber mit anderem Kommunikationsverhalten verwendet, müssen alle mit dem ausgetauschten Modul verbundenen Module an das neue Kommunikationsverhalten angepaßt werden. Erzwingt dieser Anpassungsprozeß den Austausch weiterer Module, führt dies schnell dazu, daß ein Großteil des gesamten Entwurfs überarbeitet werden muß.

Gerade bei Funktionen, die auf FPGAs implementiert werden, können sich die Leistungsdaten und der Ressourcenbedarf für verschiedene Modulvarianten sehr stark unterscheiden, so daß die Wahl einer Modulvariante großen Einfluß auf die resultierende Implementierung hat. Diese Auswirkung potenziert sich noch dadurch, daß durch die Auswahl einer Modulvariante zur Implementierung einer Funktion auch Randbedingungen für die Implementierungen der anderen Funktionen definiert. Somit kann schon eine einzige ungünstige Entscheidung dazu führen, daß die resultierende Implementierung nicht auf das Zielsystem abgebildet werden kann, bzw. die geforderten Leistungsdaten nicht eingehalten werden können.

2.3 Bewertung

Für die Programmierung der einzelnen Bestandteile eines FPGA-Prozessors, insbesondere für die FPGAs selbst, stehen leistungsfähige Werkzeuge zur Verfügung. Dennoch gibt es Defizite bei der Wiedverwendbarkeit der Module, bei der Programmierung des FPGA-Prozessors als ganzes und bei der Portabilität.

Die Wiederverwendbarkeit ist dadurch eingeschränkt, daß für die verschiedenen Möglichkeiten der Implementierung einer Funktion verschiedene Module erstellt werden müssen. Der Entwickler, der nun eine bestimmte Funktion realisieren will, muß nun aber eines dieser Module auswählen und legt sich dadurch zwangsläufig auf eine Implementierung fest. Dies führt zu einigen unerwünschten Nebeneffekten. Die so erstellte Beschreibung ist nur bedingt optimierbar, da wesentliche Implementierungsdetails bereits festgelegt sind. Da aber die Einflußgrößen, wie die zu erwartende Leistung der Gesamtlösung und die Berücksichtigung der Beschränkungen der Zielplattform durch die gewählte Implementierungsvariante, zu diesem frühen Zeitpunkt nur sehr ungenau abzuschätzen sind, führt dies dazu, daß

die Entwurfschritte zumeist mehrmals durchlaufen werden müssen, bis eine angemessene Beschreibung gefunden ist. Auf Grund der mangelnden Optimierbarkeit muß dabei der Hauptaufwand vom Entwickler erbracht werden. Gleichzeitig wird hierdurch die Portabilität eingeschränkt, da bei den Entwurfsentscheidungen viele Eigenschaften des Zielsystems eingeflossen sind, die sich oft nicht auf andere Zielsysteme übertragen lassen.

Ziel einer Entwicklungsumgebung für FPGA-Prozessoren muß es also sein, diese Defizite zu beseitigen oder zumindest zu mildern, wobei eine Abstraktionsebene nötig ist, die im größeren Umfang, als dies von den hier vorgestellten Werkzeugen ermöglicht wird, vom Zielsystem und den zu treffenden Implementierungsentscheidungen abstrahiert.

3

Programmierung von FPGA-Prozessorsystemen mittels aktiver Komponenten

Um FPGA-Prozessor Systeme effizient programmieren zu können, muß ein einheitliches, das ganze System umfassende Programmierverfahren vorliegen. Hinzu kommt die Notwendigkeit, eine hohe Wiederverwendbarkeit und Portabilität von bereits erstellten Problemlösungen zu erreichen. Im folgenden werden, ausgehend von den für die Erstellung der Problembeschreibung benötigten Beschreibungselementen, aktive Komponenten eingeführt, die Grundlagen eines darauf basierenden Programmiersystems dargestellt und die Auswirkungen auf die Wiederverwendbarkeit, die Portabilität und den Entwurfszyklus untersucht.

3.1 Problembeschreibung mit aktiven Komponenten

3.1.1 Beschreibungsebenen

Bei der Umsetzung einer Problembeschreibung in eine ausführbare Implementierung müssen mehrere Beschreibungsebenen durchlaufen werden. Ausgehend von der Beschreibung der zu realisierenden Funktion werden bei jeder Transformation immer mehr Implementierungsdetails hinzugefügt, bis die Implementierungsebene erreicht wird. An die oberste Beschreibungsebene ergeben sich für FPGA-Prozessor Systeme die Forderung, daß die gewählte Abstraktionsebene möglichst hoch liegt, um eine möglichst effiziente Beschreibung der Problemlösung und domänenspezifische Optimierungen zu ermöglichen. Zum anderen muß die Beschreibung so gewählt werden, daß sie sich sinnvoll auf die möglichen Zielsysteme abbilden läßt.

An diese Beschreibungsebene schließen sich mehrere Zwischenebenen an. Auf diesen Ebenen liegt immer noch eine funktionale Beschreibung vor, die nun aber nicht mehr domänenspezifische Beschreibungsmittel, sondern allgemeinere Be-

schreibungselemente verwendet, da beim Übergang auf diese Ebene schon die domänenspezifischen Optimierungen durchgeführt und die geeigneten Algorithmen zur Implementierung der domänenspezifischen Beschreibungselemente bestimmt wurden.

Diese Verfeinerung schreitet zu den implementierungsspezifischen Ebenen weiter, in dem die für die Realisierung der Funktion benötigten Systemressourcen wie Speicher, I/O-Schnittstellen, Taktgeneratoren etc. in die Beschreibung aufgenommen werden. Ausgehend von dieser Verfeinerung kann nun die Implementierung auf den einzelnen Systemeinheiten erfolgen, wobei hierfür die gebräuchlichen Implementierungssprachen verwendet werden können.

Die Beschreibungsebenen können somit in die Klassen der domänenspezifischen Ebenen, der funktional orientierten Ebenen und den implementierungsspezifischen Ebenen eingeteilt werden.

3.1.2 Benötigte Beschreibungselemente

Auf allen oben dargestellten Beschreibungsebenen müssen geeignete Beschreibungselemente existieren, um auf der jeweiligen Abstraktionsebene die zu realisierende Funktionalität beschreiben zu können. Bei genauer Betrachtung läßt sich feststellen, daß die Beschreibungselemente über die Beschreibungsebenen hinweg gleichartig strukturiert sind und sich hauptsächlich durch ihren Abstraktionsgrad unterscheiden.

Die erste Gruppe von Beschreibungselementen umfaßt die Verarbeitungselemente. Mit ihnen lassen sich die einzelnen zu realisierende Arbeitsschritte definieren. Sie stellen verschiedene Funktionen bereit, die in ihrer allgemeinsten Form so aufgebaut sind, daß sie bestimmte Operationen auf Eingangsdaten ausführen und die Ergebnisse in Form von Ausgangsdaten liefern. Eine weitere Gruppe dient der Beschreibung des Datenaustauschs zwischen den Verarbeitungselementen. Während auf den höheren Abstraktionsebenen hauptsächlich die Kommunikationspartner und der Typ der zu transportierenden Information festgelegt werden, wird auf den unteren Beschreibungsebenen zusätzlich das Übertragungsverfahren spezifiziert. Des weiteren werden noch Beschreibungselemente benötigt, die es erlauben, die Reihenfolge, in der die Verarbeitungsschritte abgearbeitet werden sollen, festzulegen.

Viele der Beschreibungselemente, die auf den obersten Beschreibungsebenen verwendet werden, sind sehr spezifisch auf den jeweiligen Anwendungsbereich ausgelegt. Dies ist auch unumgänglich, will man eine domänenspezische Abstraktion erreichen. Um ein für beliebige Domänen geeignetes Beschreibungsverfahren zu erhalten, muß es möglich sein, ausgehend von einem beschränkten Satz von Basiselmenten, die für den jeweiligen Anwendungsbereich benötigten Beschreibungselemente zu definieren. Diese Definition muß sowohl eine für den Anwender verständliche Festlegung der durch das jeweilige Beschreibungselement bereitgestellten Funktionalität, als auch Transformationen enthalten, die die Abbildung des Beschreibungselementes auf tiefer gelegene Beschreibungsebenen ermöglicht.

Zusätzlich ist die Bereitstellung von Kriterien notwendig, die es erlauben, Prüfungen über die korrekte Verwendung des Beschreibungselementes innerhalb der Problembeschreibung durchzuführen.

3.1.3 Grundlegender Aufbau von aktiven Komponenten

Komponenten stellen in dem hier verwendeten Kontext Objekte dar, die eine oder mehrere Funktionen bereitstellen. Eine Funktion ist dadurch definiert, daß sie festgelegte Verarbeitungsschritte ausführt und Daten über eine festgelegte Schnittstelle entgegennimmt bzw. weiterleitet. Zusätzlich verfügt eine Komponente über Attribute, die das Verhalten der Funktionen näher festlegen bzw. beeinflussen können. Zur Schnittstellendefinition gehören nicht nur die Festlegung der Anzahl der zu übermittelnden Daten, sondern auch deren Typ. Die Schnittstellen selbst setzen sich aus Ports zusammen, wobei jeder Port für den Transport eines Funktionsparameters zuständig ist (Bild 3.1) .

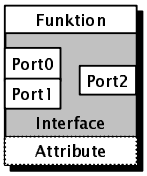

Bild 3.1. Prinzipieller Aufbau einer aktiven Komponente

Des weiteren bietet jede Schnittstelle Möglichkeiten zur Ablaufsteuerung. Hierbei stehen im wesentlichen zwei Verfahren, die datenflußorientierte bzw. kontrollflußorientierte Steuerung zur Verfügung. Bei der Datenflußsteuerung wird eine Funktion genau dann ausgeführt, wenn jeder Eingangsport Daten bereithält und jeder Ausgangsport Daten weiterleiten kann. Vorteilhaft ist dieser Ansatz hauptsächlich dann, wenn die Funktionsparameter aus Datenströmen stammen, bei denen auf jeden Datensatz gleiche Operationen auszuführen sind. Kontrollflußsteuerung wird hingegen hauptsächlich bei Aufgaben eingesetzt, bei denen die Ausführung von Verarbeitungsschritten explizit gesteuert werden soll. Diese Steuerfunktion übernimmt dann die so genannte Kontrolleinheit. Dieser Ansatz wird verwendet, wenn variable Befehlssequenzen ausgeführt oder ein bestimmter zeitlicher Ablauf realisiert werden soll. Beide Verfahren können von den Komponenten unterstützt werden, in dem die Funktionsschnittstelle entsprechend ausgelegt wird. Eine Schnittstelle, die Datenflußsteuerung unterstützt, enthält lediglich Ports, die die Parameter der Funktion transportieren. Bei der Kontrollflußsteuerung wird zusätzlich zu den Datenports ein Steuerport bereitgestellt. Ist über diesen Steuerport ein Datum verfügbar, wird die Funktion gestartet. Die unterschiedlichen Realisierungsmöglichkeiten sollen an Hand des Beispiels in Bild 3.2 näher erläutert werden.

Es sind zwei Beschreibungsmöglichkeiten der Funktion $z = (x + y)(e - f)$ dargestellt. Variante (a) ist hierbei die datenflußorientierte Beschreibungsvariante.

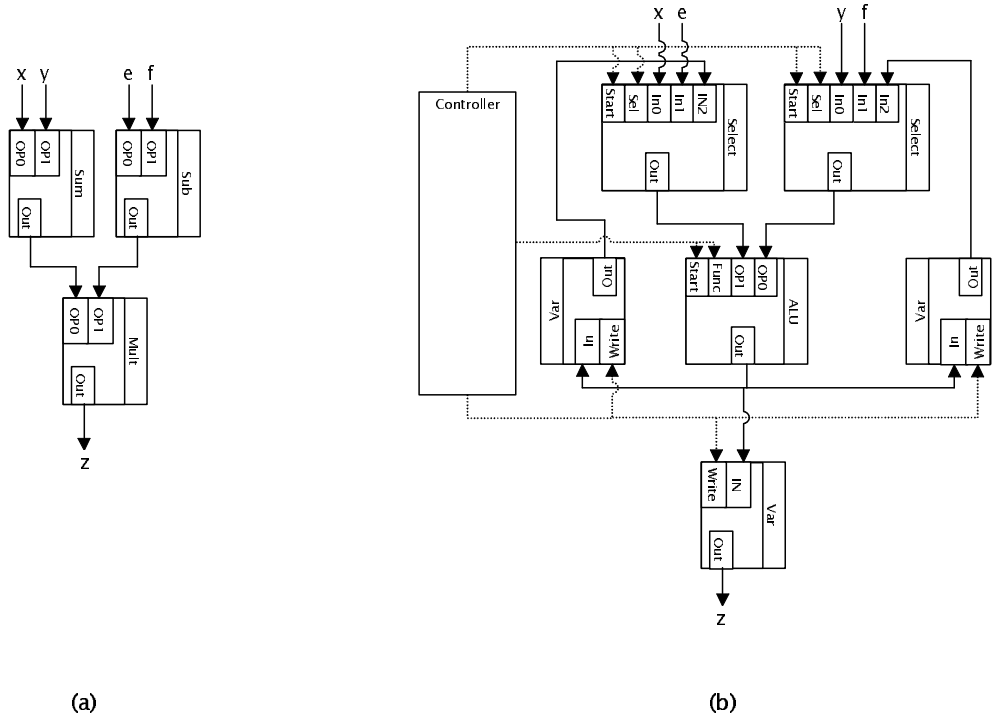

Bild 3.2. Datenflußorientierte (a) bzw. kontrollflußorientierte Implementierung (b) der Funktion $z = (x + y)(e - f)$

Für jeden Verarbeitungsschritt existiert eine separate Komponente, die das Ergebnis der Operation an die nachfolgende Verarbeitungsstufe weiterreicht. Die Beschreibung entspricht hierbei einer Pipelinestruktur, bei der im Idealfall alle Komponenten gleichzeitig aktiv sein können. Während in Variante (a) die Ausführbarkeit einer Funktion dardurch bestimmt ist, daß alle Ports der Funktion Daten bereitstellen bzw. entgegennehmen können, wird diese Steuerfunktion in der kontrollflußorientierten Variante (b) durch ein zentrales Steuerelement, den Controller[1] übernommen. Er kann durch entsprechende Steuerports den Zeitpunkt und die Art der durch die jeweilige Komponente auszuführende Funktion bestimmen. Hierzu wird die Funktion in mehrere sequentielle Verarbeitungsschritte aufgeteilt. In den ersten Arbeitsschritten werden die Werte x und y über die Datenselektoren an die ALU weitergereicht, dort addiert und anschließend zwischengespeichert. Auf gleiche Weise wird die Differenz von e und f bestimmt und gespeichert. Die beiden Zwischenergebnisse werden nach entsprechender Konfiguration der Datenselektoren durch die ALU multipliziert und das Ergebnis in der Ausgangsvariablen gespeichert.

[1] Obwohl der Controller auch als Komponente realisiert ist, wird er hier nur schematisch dargestellt, da ansonsten durch die große Anzahl von Ports die Übersichtlichkeit verloren gehen würde.

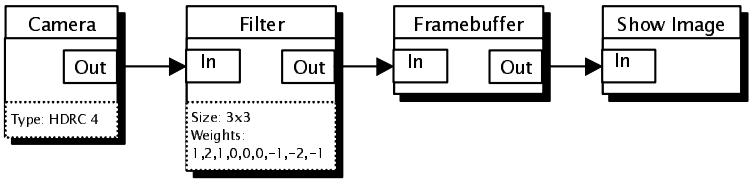

Bild 3.3. Beispielanwendung aus der Bildverarbeitung: Filtern und Anzeigen der Bilder eines Bildstromes.

Während das hier dargestellte Beispiel auf einer relativ niedrige Abstraktionsebene angesiedelt ist, lassen sich auf höheren Abstraktionsebenen auch komplexere Funktionen und Datentypen einsetzen. Ein weiteres Beispiel, das diesen Sachverhalt veranschaulichen soll, ist dem Bereich der Bildverarbeitung entnommen (Bild 3.3).

Hier wird ein System beschrieben, das die Daten einer Video-Kamera filtert, zwischenspeichert und dann die erzeugten Bilder visualisiert. Die Ports transportieren in dieser Beschreibungsform jedes Bild des Bildstroms als ein einziges Datum, auf das dann die Operationen filtern (Filter), speichern bzw. auslesen (Framebuffer) und anzeigen (Show Image) angewandt werden. Der Datentyp, der hier ein Bild definiert, enthält Informationen über die Anzahl der Pixel pro Zeile, die Anzahl der Zeilen und den Pixel-Typ. Dieser wiederum enthält Angaben über die Pixelbreite und das zugrundeliegende Basisformat wie z.B. RGB, Grauwert etc. .

Während mit diesen Beschreibungsmitteln schon einige Problembeschreibungen möglich sind, ergeben sich doch Probleme, zumeist auf Ebenen mittlerer Abstraktion, für die die hier vorgestellten Verfahren nicht ausreichend sind. Die hierfür benötigten Elemente sollen im folgenden Abschnitt näher beschrieben werden.

3.1.4 Kommunikationseigenschaften von Ports

Bisher wurde für Ports nur vorausgesetzt, daß sie in der Lage sind, Daten eines bestimmten Typs zu transportieren und anzuzeigen, ob neue Daten bereitstehen bzw. weitergeleitet werden können. Für die Beschreibung auf funktionaler Ebene ist dies auch ausreichend, muß aber für die Verwendung auf tiefer angesiedelten Beschreibungsebenen ergänzt werden.

Die Grundlage für diese Erweiterung besteht darin, jedem Port einen eigenen Porttyp zuzuordnen. Dieser Porttyp umfaßt alle bisher für Ports aufgezählten Eigenschaften, ergänzt diese aber um Kommunikationseigenschaften, in dem durch den jeweiligen Porttyp ein Kommunikationsprotokoll festgelegt wird.

Um ein derartiges Protokoll realisieren zu können, müssen zusätzlich zu den eigentlichen Daten noch Steuerinformationen übertragen werden. Die hierfür benötigten Datenkanäle werden wiederum durch Ports (Protokollports) repräsentiert und im Rahmen des Porttyps festgelegt. Das Protokoll selbst definiert dann

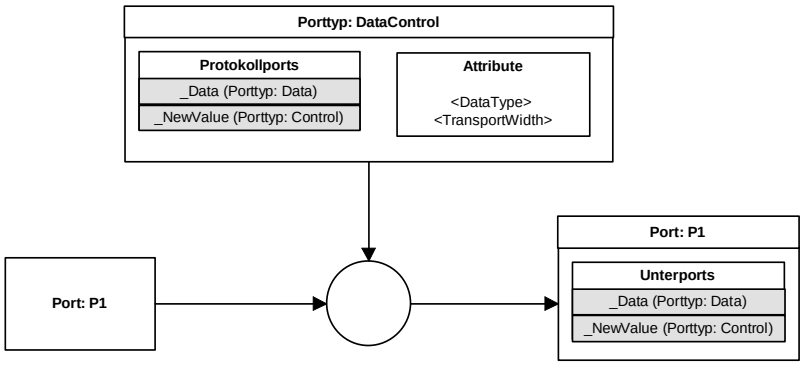

Bild 3.4. Zusammenwirken von Port und Porttyp

nur noch, wie diese Ports bedient werden müssen, um das festgelegte Kommunikationsverfahren zu unterstützen.

Wird einem Port ein Porttyp zugeordnet, werden die im Porttyp definierten Protokollports dem betreffenden Port als Unterports zugeordnet (Bild 3.4).

Porttypen können Attribute bereitstellen, die es erlauben, bestimmte Eigenschaften des vorgegebenen Protokolls zu beeinflussen. Ein verbindliches, für jeden Port anzugebendes Attribut ist das Master-Attribut, das angibt, ob der Port von sich aus Transaktionen initiieren kann. Weitere Attribute können Leistungsdaten wie z.B. den minimal erforderlichen Datendurchsatz spezifizieren.

Auch die Interpretation der Verbindung zwischen verschiedenen Ports muß dem hierarchischen Aufbau der Ports angepaßt werden. Dem wird dadurch Rechnung getragen, daß eine Verbindung von Ports gleichzeitig eine eins–zu–eins Verbindung der Unterports darstellt.

Mit diesen Mitteln lassen sich leicht aufeinander aufbauende Porttypen spezifizieren, was an einigen Beispielen gezeigt werden soll.

Der einfachste Porttyp ist eine 1-Bit breite Datenverbindung ohne jegliche Steuersignale, im folgenden als `DATA_1` bezeichnet. Darauf basiert der Typ `DATA`, der eine n-Bit Datenverbindung darstellt und hierzu n Ports vom Typ `DATA_1` als Protokollports definiert. Er verfügt über das Attribut DataType, das den Typ der transportierten Daten angibt .

Der Porttyp `Control` findet für Steuerports Verwendung. Der aktive Zustand wird durch das Attribut ActiveState, das die Werte High oder Low annehmen kann, spezifiziert.

Der Porttyp `DataNvRd`, ermöglicht eine gesteuerte Datenübertragung. Hierzu werden drei Protokollports, Data vom Typ `Data`, NewValue und Ready, beide vom Typ `Control`, verwendet. `DataNvRd` definiert die Attribute TransportWidth und DataType. Das TransportWidth-Attribut spezifiziert die Breite des Data-Ports und DataType den Typ der übertragenen Daten. Während der Data-Port die eigentlichen Daten überträgt, signalisiert der NewValue-Port, daß ein neues Datum auf den Datenleitungen anliegt. Mit Hilfe des Ready-Ports kann die Datensenke die Übertragung neuer Werte verhindern. Da die Bit-Breite der Daten getrennt

von der Breite des bei Datenübertragung verwendeten Daten-Ports spezifiziert werden kann, ist es mit diesen Porttypen möglich, eine serielle oder teilserielle Übertragung der Daten festzulegen. Verwendet man z.B. einen Port mit TransportWidth = 4, der aber Daten mit acht Binärstellen überträgt, muß ein Datum in zwei, vier Bit Hälften zerlegt, getrennt übertragen und beim Empfänger wieder zu einem acht Bit Wert zusammengesetzt werden. Weitere Porttypen sind in Abschnitt 9.3 beschrieben.

Um mit diesem Beschreibungsverfahren Ports auch auf den höchsten Beschreibungsebenen verwenden zu können, werden verschiedene Ansätze unterstützt.

Es ist möglich, Porttypen nicht vollständig zu spezifizieren. So kann z.B. beim Porttyp `DataNvRd` das TransportWidth-Attribut undefiniert bleiben, womit zwar das zugrundeliegende Kommunikationsverhalten festgelegt, die Implementierung aber offen bleibt. Im Extremfall ist es auch möglich, den Porttyp selbst vollständig undefiniert zu lassen.

Oft ist es aber wünschenswert, die grundlegenden Kommunikationseigenschaften zu spezifizieren, ohne aber bereits den konkreten Porttyp festzulegen. Erreicht wird dies durch abstrakte Porttypen. Sie stehen stellvertretend für eine Gruppe von verschiedenen Porttypen. Auf einer tiefer angesiedelten Beschreibungsebene wird dann der abstrakte Typ durch einen konkreten Typ ersetzt. Hierbei sind mehrere Umsetzungsschritte möglich, so daß ein abstrakter Typ auch erst durch weitere abstrakte aber spezifischere Porttypen ersetzt werden kann, bevor der für die Implementierung benötigte konkrete Porttyp ausgewählt wird.

3.1.5 Zusammenfassung

Es konnte gezeigt werden, daß mit aktiven Komponenten sowohl funktionale, auf domänenspezifischer Abstraktionsebene angesiedelte Beschreibungen, als auch Beschreibungen auf Implementierungsebene möglich sind. Zur Beschreibung werden Komponenten verwendet, die Verarbeitungfunktionen bereitstellen und über Ports die zu verarbeitenden Daten entgegennehmen bzw. über diese Ergebnisse weitergeben. Da die von Komponenten bereitgestellten Funktionen, die Typen der über die Ports transportierten Daten und auch der für die benutzten Porttypen beliebig festgelegt werden können, ist es möglich, für die jeweilige Anwendungsdomänen und verschiedenen Beschreibungsebenen geeignete Beschreibungselemente bereitzustellen. Da es auch möglich ist, abstrakte bzw. unvollständig definierte Porttypen zu verwenden, ist auch bei der Beschreibung des Kommunikationsverhaltens eine Beschreibungsebene erreichbar, die weitestgehend von Implementierungsentscheidungen abstrahiert und somit eine möglichst einfache und zielplattformunabhängige Beschreibung ermöglicht, die darüberhinaus noch weitgehend optimierbar ist.

Entwurfsentscheidung	**Entwurfskriterien**
Zielsystem	Verfügbarkeit, Kosten, Eignung
Partionierung	Implementierbarkeit, Leistung der Implementierung, ausreichende Systemressourcen
Porttypen	Resultierende Performance, verfügbare Systemresourcen, Unterstützung durch Komponente
Implementierung (einer Komponente)	Vorgegebene Porttypen, Attribute, Performance, Ressourcenbedarf, zugeordnete Systemeinheit

Tabelle 3.1. Entwurfsentscheidungen und die zugeordneten Entwurfskriterien

3.2 Von der Komponente zur Implementierung

Während zuvor die Lösungsbeschreibung mittels aktiver Komponenten behandelt wurde, soll hier die Umsetzung dieser Beschreibungen in eine ausführbare Implementierung behandelt werden. Ziel ist es, einen geeigneten Umsetzungsprozeß zu skizzieren und den Aufbau einer zur Realisierung dieses Umsetzungsprozesses benötigten Entwicklungsumgebung festzulegen. Im Laufe dieses Umsetzungsprozesses wird die Lösungsbeschreibung Schritt für Schritt immer weiter konkretisiert, in dem verschiedene Entwurfsentscheidungen getroffen werden. Tabelle 3.1 enthält die Aufstellung der relevanten Entwurfsentscheidungen und der zugrundeliegenden Entwurfskriterien.

Die erste Entscheidung betrifft die Auswahl des Zielsystems, wobei hier zwei verschiedene Ansätze verfolgt werden können. Im ersten Fall, der bei FPGA-Prozessoren am häufigsten auftritt, wird ein geeignetes System aus bereits existierenden Zielsystemen ausgewählt. Hierbei werden zum einen wirtschaftliche Aspekte aber auch die Eignung des Systems für die geplante Anwendung als Kriterien herangezogen. Im zweiten Fall wird die Erstellung einer Problemlösung mit dem Entwurf eines geeigneten Zielsystems verbunden. Die zu lösende Aufgabe besteht darin, ein Zielsystem zu entwerfen, so daß eine bestimmte Klasse von Problemstellungen auf diesem System möglichst effizient, mit möglichst guten Leistungsdaten, bei möglichst geringen Kosten für das Zielsystem implementiert werden kann. Das übliche Vorgesehen besteht dabei darin, mehrere hypothetische Systeme zu entwerfen, repräsentative Applikation darauf zu erstellen und dann die Leistungsfähigkeit des jeweiligen Systems zu bewerten und anhand dieser Kriterien das geeignete System auszuwählen. In beiden Fällen muß es möglich sein, eine Problemlösung auf verschiedenen Zielsystemen zu implementieren.

Eine weitere Entwurfsentscheidung betrifft die Partitionierung. Hier muß entschieden werden, welcher Teil der Problemlösung, also welche Komponente, auf welchem Element des Zielsystems implementiert werden soll. Das erste Kriterium zur Entscheidung dieser Frage ist, ob für die gegebene Komponente eine Imple-

mentierung auf der ausgewählten Systemeinheit möglich ist, bzw. unterstützt wird. Desweiteren ist zu klären, ob die geforderten Leistungsdaten bei einer Implementierung auf der ausgewählten Systemeinheit eingehalten werden können. Ist es das Ziel des Entwurfsprozesses, ein geeignetes Zielsystem für die gegebene Problemlösung zu entwerfen, muß eine Partitionierung gewählt werden, die die Verwendung möglichst preisgünstiger Bausteine ermöglicht. Ist das Ziel des Entwurfsprozesses hingegen eine Problemlösung auf ein gegebenes Zielsystem abzubilden, ändert sich die Aufgabenstellung dahingehend, daß eine Partitionierung gefunden werden muß, für die genügend Systemressourcen auf dem gewählten Zielsystem vorhanden sind. Zusätzlich muß bei der Partitionierung berücksichtigt werden, daß für die Kommunikationspfade zwischen den Komponenten ausreichende Verbindungsressourcen vorhanden sind. Spezifizieren die Komponenten abstrakte bzw. unterdefinierte Porttypen, sind nur grobe Schätzungen für dieses Kriterium möglich.

Bei der Entscheidung, welcher Porttyp für einen gegebenen Port ausgewählt werden soll, müssen mehrere Faktoren berücksichtigt werden. Der erste Gesichtspunkt ist hierbei, welche Porttypen für den gewählten Port durch die Komponente, zu der der Port gehört, unterstützt werden. Gleichzeitig muß berücksichtigt werden, daß nur diejenigen Porttypen verwendet werden können, die von allen mit dem ausgewählten Port verbundenen Ports bereitgestellt werden. Es ist also im eigentlichen Sinne nicht der Typ eines Ports, sondern der geeignete Porttyp für einen Knoten bestimmt werden. Es muß vermieden werden, daß ein Porttyp verwendet wird, der es unmöglich macht, für die anderen Knoten der Beschreibung eine geeignete Lösung zu finden. Dieses Problem resultiert daraus, daß Komponenten meist mehrere Ports enthalten. Obwohl für diese Ports mehrere Porttypen unterstützt werden, sind sie meist nicht unabhängig von einander wählbar. Dieser Sachverhalt ist in Bild 3.5 noch einmal dargestellt.

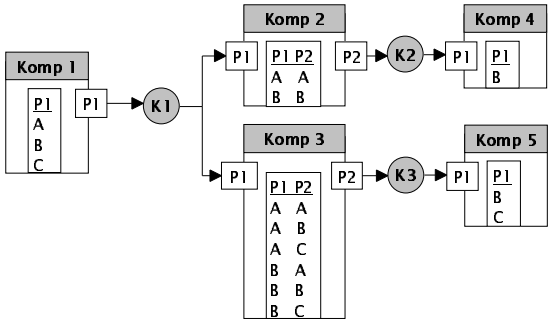

Bild 3.5. Kriterien bei der Auswahl von Knoten- bzw. Porttypen

Betrachtet man zu erst den Knoten K1, so stellt man fest, daß es bei einer lokalen Betrachtung für diesen Knoten zwei mögliche Porttypen (A, B) gibt. Würde man den Porttyp A auswählen, wäre dies zwar eine realisierbare Möglichkeit für den Knoten K1, ließe aber keine Möglichkeit offen, einen geeigneten Typ für den

Knoten K2 zu wählen. Dieses Problem tritt dadurch auf, daß die Komponente Komp2 für ihren Port P2 nur den Typ A unterstützt, falls ihr Port P1 mit dem Typ A vorbelegt ist. Die Komponente Komp 4 unterstützt für ihren Port P1 aber nur den Porttyp B, so daß für K2 kein geeigneter Porttyp verfügbar ist. Für den Knoten K3 ergibt sich aus dieser Konstellation kein Problem, da die Komponente Komp3 für ihren Port P2 die Typen A, B und C unterstützt, unabhängig davon, wie der Typ ihres Ports P1 vorbelegt ist. Um nun aber eine Umsetzung der Gesamtbeschreibung zu ermöglichen, ist es notwendig für K1 den Typ B auszuwählen, wodurch für K2 der Typ B und für K3 die Typen B oder C gewählt werden können. Die bisher genannten Kriterien waren notwendig, um eine Umsetzung der Beschreibung zu ermöglichen. Weitere Kriterien dienen dazu, die Performance der Implementierung zu erhöhen, den Ressourcenbedarf zu reduzieren oder die Beschränkungen der Zielplattform einzubeziehen. Die Wahl eines Porttyps beeinflußt die Leistungsfähigkeit der Implementierung und den Implementierungsaufwand in zweierlei Hinsicht. Unmittelbar legt der Porttyp die Leistungsfähigkeit der Kommunikation und den Bedarf an Verbindungsressourcen fest. Mittelbar nimmt diese Entscheidung aber auch auf die Implementierung der verbundenen Komponenten Einfluß, da der vorgegebene Kommunikationsmechanismus die möglichen Implementierungsvarianten einschränkt und somit auch die Performance und den Ressourcenbedarf der Komponentenimplementierung maßgeblich beeinflußt. Diese Beeinflussung betrifft aber, wie bereits gezeigt, nicht nur die direkt mit dem Knoten verbundenen Komponenten, sondern dehnt sich auf alle Komponenten aus, die nur mittelbar über andere Komponenten mit dem aktuell bearbeiteten Knoten verbunden sind. Auch die, durch den Porttyp belegten Verbindungsressourcen haben Auswirkungen auf die Implementierungsmöglichkeiten der anderen Knoten. Jedes Zielsystem verfügt nur über eine beschränkte Anzahl von Verbindungsressourcen. Jeder auf dem Zielsystem realisierte Knoten schränkt die frei verfügbaren Verbindungsressourcen weiter ein. Dies kann dazu führen, daß ein Knoten gewisse Porttypen nicht mehr nutzen kann, da für die Realisierung dieses Porttyps nicht mehr genügend Ressourcen auf dem Zielsystem bereitstehen.

Die Auswahl der geeigneten Implementierung für die jeweiligen Komponenten wird grundlegend von den bisher dargestellten Entscheidungen mitbestimmt. Durch die Partitionierung ist bereits festgelegt, auf welcher Systemeinheit die Komponente zu realisieren ist. Durch die bereits erfolgte Bestimmung der Porttypen ist die Funktion der Kommunikationsprozesse weitgehend festgelegt. Die verbleibenden Auswahlkriterien zielen darauf ab, die Komponente so zu Implementieren, daß die geforderten Leistungsdaten und andere durch die Attribute der Komponente festgelegten Eigenschaften eingehalten werden und gleichzeitig ein möglichst sparsamer Umgang mit den vorhandenen Implementierungsressourcen gepflegt wird.

Es bleibt festzuhalten, daß sich die hier beschriebenen Entwurfsentscheidungen gegenseitig stark beeinflussen. Dieser Zusammenhang muß beim Umsetzungsprozeß und den darin enthaltenen Optimierungsschritte berücksichtigt werden.

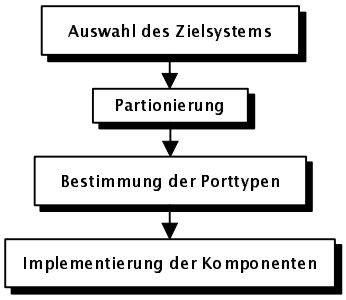

Bild 3.6. Reihenfolge der Entwurfsentscheidungen

Die eigentliche Aufgabe besteht darin, die oben genannten Entwurfsentscheidungen in einen Entwurfszyklus einzugliedern. Die Reihenfolge der Umsetzungsschritte ist so zu wählen, daß eine möglichst optimale Implementierung erstellt werden kann.

Auch müssen die Instanzen bestimmt werden, die die jeweilige Entwurfsentscheidung treffen bzw. die hierfür notwendigen Entscheidungskriterien liefern. Dies ist hauptsächlich im Bezug auf die Wartbarkeit und Erweiterbarkeit des Programmiersystems von Bedeutung. Anzustreben ist hierbei eine Konstellation, bei der die Instanz, die für die Erstellung der aus einer Entwurfsentscheidung resultierenden Implementierung zuständig ist, auch die Entwurfsentscheidung trifft bzw. die für die Entwurfsentscheidung notwendigen Kriterien liefert. Hierdurch wird eine sinnvolle Gruppierung der benötigten Informationen erreicht, die auf Grund der engen Lokalisierung sicherstellt, daß Änderungen an den bereitgestellten Informationen nur lokale Auswirkungen haben und keine weiteren Veränderungen am Gesamtsystem notwendig machen.

Was die Reihenfolge der Entwurfsentscheidungen betrifft, kann eine sinnvolle Ordnung dadurch gewonnen werden, in dem man die Abhängigkeiten zwischen den entsprechenden Entwurfsentscheidungen analysiert. Es läßt sich feststellen, daß einige Entwurfsentscheidungen aufeinander aufbauen. So muß z.B., um eine geeignete Implementierung einer Komponente zu bestimmen, die Partitionierung und die Bestimmung der Porttypen erfolgt sein, da nur hierdurch die Eigenschaften der Systemeinheit und das geforderte Kommunikationsverhalten in die Implementierungsentscheidung einbezogen werden können. Ähnlich gilt für die Auswahl der Porttypen. Hierfür muß die Entscheidung über die Partitionierung bereits erfolgt sein, da nur so die für die Realisierung der Verbindungen verfügbaren Verbindungsressourcen bekannt sind. Um wiederum die Partitionierung durchzuführen, muß zuvor das gewünschte Zielsystem bestimmt werden. Hieraus ergibt sich die in Bild 3.6 dargestellte Reihenfolge.

Die Durchführung dieser Entwurfsentscheidungen ist Aufgabe des zu erstellenden Entwicklungssystems. Es muß über Instanzen verfügen, die die benötigten Informationen bereitstellen bzw. Entwurfsentscheidungen selbst treffen. Die unverzichtbaren Elemente sind hierbei die Lösungsbeschreibung, die Zielsystembeschreibung und die Komponenten (Bild 3.7) .

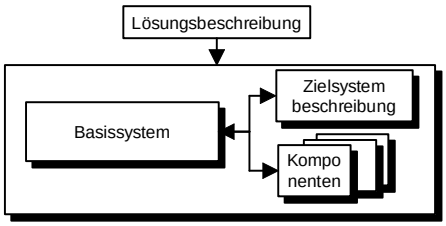

Bild 3.7. Aufbau des Entwicklungssystems

Während die Lösungsbeschreibung die zu implementierende Funktionalität vorgibt, enthält die Zielsystembeschreibung alle Informationen über das Zielsystem, die für die verschiedenen Abbildungsschritte und Entwurfsentscheidungen notwendig sind. Hierzu gehören die auf dem System verfügbaren Systemeinheiten, deren Typ und die auf ihnen verfügbaren Implementierungs- und Verbindungsressourcen. Des weiteren müssen Informationen darüber vorliegen, welche Verbindungsressourcen zwischen den Systemeinheiten verfügbar sind und wie sich auf diesen die Komponentenverbindungen realisieren lassen. Zusätzlich umfaßt die Systembeschreibung auch Elemente, die die Initialisierung der Systembestandteile durchführen bzw. definieren, wie diese Funktionen realisiert werden können.

Die Komponenten müssen in der Lage sein, Informationen über die von ihnen bereitgestellten Implementierungsvarianten und deren Eigenschaften bereitzustellen. Die Gesamtheit dieser Informationen wird als Konfigurationswissen bezeichnet. Es umfaßt die Implementierungsvarianten und deren Eigenschaften wie Ressourcenverbrauch, Performanceangaben, unterstützte Systemeinheiten und die für die verschiedenen Ports verfügbaren Porttypen einschließlich der möglichen Kombinationsmöglichkeiten, in denen diese für die eingesetzt werden können.

Auf Grund dieses Konfigurationswissens ist eine Komponente in der Lage eine geeignete Implementierung innerhalb eines vorgegebenen Kontextes zu wählen. Da eine Komponente beliebige Funktionen bereitstellen kann, ist es leicht ersichtlich, daß die Implementierung von der jeweiligen Komponente selbst durchzuführen ist, da nur sie in der Lage ist, die Bedeutung der Funktion zu interpretieren und dementsprechend die zur Realisierung der Funktion notwendigen Verarbeitungsschritte zu definieren.

Andererseits werden bei der Erstellung der Implementierungen verschiedene Schritte durchgeführt, die von allen Komponenten benötigt werden. Hierzu gehört z.B. ein Interface zur Verwendung von Implementierungssprachen, das von allen Komponenten zur Realisierung der jeweiligen Implementierung verwendet werden kann. Diese Aufgabe wird an das Basissystem übertragen.

Das Basissystem hat die Aufgabe, den gesamten Umsetzungsprozeß zu steuern und umfaßt die Instanzen, die für die Plazierung, die Bestimmung der Porttypen eines Knotens, das Routing und die Implementierungsunterstützung zuständig sind. Zusätzlich erstellt das Basissystem aus den Implementierungen der Komponenten die Implementierungsbeschreibung für die Systemeinheiten bzw. die Konfigurationsbeschreibungen für die übrigen Bestandteile des Zielsystems und

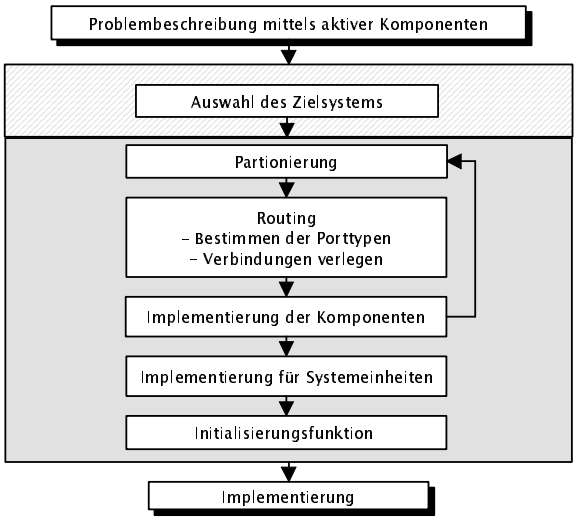

Bild 3.8. Umsetzungschritte von der Lösungsbeschreibung zur Implementierung

bindet hierin in Zusammenarbeit mit der Systembeschreibung auch die Funktionalität zur Initialisierung des gesamten Zielsystems ein.

Der resultierende Umsetzungszyklus ist noch einmal in Bild 3.8 dargestellt. Die grau hinterlegten Umsetzungsschritte werden automatisch vom Entwicklungssystem durchgeführt. Die Auswahl des Zielsystems soll in der weiteren Betrachtung als ein vom Entwickler selbst durchzuführender Arbeitsschritt betrachtet werden, da das Augenmerk hauptsächlich auf den eigentlichen Umsetzungsprozeß von der Lösungsbeschreibung zur Implementierung hin gelegt werden soll.

Erklärungsbedürftig ist die dargestellte Möglichkeit, vom Schritt der Implementierung der Komponenten, zurück zur Durchführung der Plazierung zu gelangen. Dieser Sachverhalt wird ausführlich in Abschnitt 4.3 behandelt, soll hier aber kurz angerissen werden, da die grundlegende Kenntnis dieses Sachverhaltes für das Verständnis der folgenden Passagen notwendig ist. Eine Komponente ist in der Lage, ihre Implementierung über die Verkettung anderer Komponenten zu beschreiben. Diese neu erzeugten Komponenten müssen auf dem Zielsystem abgebildet werden, so daß die Abbildungsschritte von der Partitionierung an, für diese Komponenten erneut zu durchlaufen sind.

Ansonsten ist zu beachten, daß der hier vorgestellte Umsetzungsprozeß bisher nur grob skiziert wurde. Vernachlässigt wurde in der Darstellung aus Bild 3.8 hauptsächlich, daß die Möglichkeit bestehen muß, einzelne Umsetzungsschritte zurückzunehmen. Dieser Fall kann auftreten, falls die Umsetzungsschritte auf Entwurfsentscheidungen basieren, die in Abhängigkeit zu anderen Entwurfsentscheidungen stehen, diese aber nicht gemeinsam, sondern nacheinander getroffen werden. Diese Problemstellung und die Realisierung der einzelnen Umsetzungsschritte wird in den nächsten Abschnitten behandelt.

4
Systemabbildung

In diesem Kapitel werden die einzelnen Abbildungsschritte (Bild 3.8) näher erläutert und in den Umsetzungsprozeß eingegliedert. Besondere Beachtung finden hierbei die Abhängigkeiten der einzelnen Umsetzungsschritte untereinander und die daraus resultierenden Auswirkungen auf den Umsetzungsprozeß. Die Verarbeitungsschritte werden gemäß ihrer zeitlichen Abfolge beschrieben, wobei die für die jeweiligen Implementierungsentscheidungen notwendigen Entscheidungsinstanzen und die Eigenschaften der Informationsinstanzen mit eingeschlossen sind.

4.1 Partitionierung

Bei der Partitionierung der Lösungsbeschreibung geht es darum, die innerhalb der Beschreibung verwendeten Komponenten den verschiedenen Systemeinheiten zuzuordnen. Hierbei sind verschiedene Punkte zu beachten. Die Grundbedingung besteht darin, daß die jeweilige Komponente auf der ausgewählten Systemeinheit implementierbar ist. Zusätzlich können Abhängigkeiten zwischen verschiedenen Komponenten existieren, die die Plazierungsvarianten weiter einschränken. Die Informationen, die für die Auswahl möglicher Systemeinheiten benötigt werden, müssen hierbei von den Komponenten und der Zielsystembeschreibung bereitgestellt werden. Während die Komponente über Informationen über die von ihr bereitgestellten Implementierungen und deren Anforderung an die verwendbaren Systemeinheiten verfügt, enthält die Zielsystembeschreibung die nötigen Aussagen über die verfügbaren Systemeinheiten und deren Eigenschaften. Sind die Anforderungen der Komponente mit den Eigenschaften einer Systemeinheit in Einklang zu bringen, so ist eine Implementierung der Komponente auf dieser

Systemeinheit prinzipiell möglich. In welcher Form diese Informationen bereitgestellt werden können, ist in Bild 4.1 dargestellt.

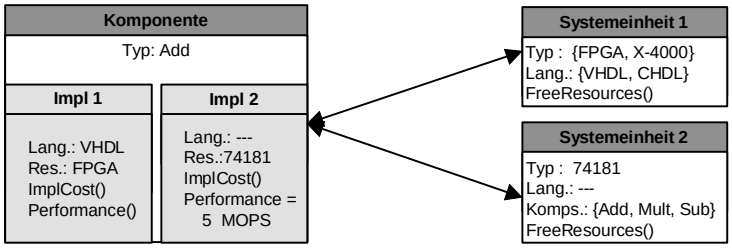

Bild 4.1. Auswahlkriterien bei der Zuordnung von Komponenten zu Systemeinheiten

Die Komponente spezifiziert ihren Typ, der die Komponente eindeutig identifiziert und auch die verfügbaren Implementierungsvarianten. Für jede Implementierungsvariante wird die verwendete Implementierungssprache (Lang.) und die verwendbaren Systemressourcen bekanntgegeben. Bei der Spezifikation der möglichen Systemressourcen (Res.) kann ein eindeutiger Bausteintyp (Impl2), eine Bausteinklasse (Impl 1) oder ein bestimmter Baustein auf einem bestimmten Zielsystem vorgegeben werden, was dann sinnvoll ist, wenn die Implementierung auf ein spezielles Zielsystem ausgelegt ist und bereits Annahmen über dessen Eigenschaften in die Implementierung eingeflossen sind. Von Seiten der Systemeinheiten bestehen die bereitgestellten Informationen in deren Typ bzw. der Bausteinklasse, den für die Implementierung unterstützten Implementierungssprachen und gegebenenfalls auch einer Liste der unterstützten Komponenten. Um also eine Komponente auf einer gegebenen Systemeinheit implementieren zu können, muß eine Implementierungsvariante der Komponente vorliegen, bei der gilt:

$$Komponete.Impl.Res \cap Systemeinheit.Typ \neq \{\} \quad \wedge$$
$$Komponete.Impl.Lang \cap Systemeinheit.Lang \neq \{\} \quad \wedge$$
$$(Komponete.Type \cap Systemeinheit.Komp \neq \{\} \vee Systemeinheit.Komp = \{\})$$

Diese Bedingungen sind aber nicht ausreichend, um die geforderte Implementierung erstellen zu können. Es muß zusätzlich in Betracht gezogen werden, ob für die schlußendlich erzeugte Implementierung der Komponente ausreichend Ressourcen bereitstehen, genügend Verbindungsressourcen zu den verbundenen Komponenten verfügbar sind und ob die geforderten Leistungsdaten eingehalten werden. Da aber viele dieser Kriterien im jetzigen Stadium nicht oder nur mittels sehr ungenauer Schätzungen ermittelt werden können, basieren die hier getroffenen Entscheidungen auf ungenauen Daten. Dies kann dazu führen, daß in späteren Umsetzungsschritten festgestellt wird, daß keine Implementierung mit den geforderten Eigenschaften möglich ist. So kann z.B. im Routing-Schritt ein Engpaß an Routing Ressourcen auftreten, der sich nur durch eine Änderung der Partitionierung auflösen läßt. Gleiches trifft auf den Schritt der Implementierung der Komponenten zu. Hier kann es hauptsächlich zu Engpässen bei den verfügbaren

Implementierungsressourcen der zugeordneten Systemressourcen komme. In diesen Fällen muß eine neue Partitionierung erfolgen, die die Erfahrungswerte aus dem vorhergehenden Umsetzungszyklus mit aufnimmt und versucht, die festgestellten Engpässe zu vermeiden.

Ein bisher nur angedeutetes Problem betrifft die Plazierung von Komponenten, zwischen denen Abhängigkeiten existieren, die die Plazierungsvarianten weiter einschränken. Dieses Problem soll exemplarisch an der in Bild 4.2 dargestellten Konstellation erläutert werden.

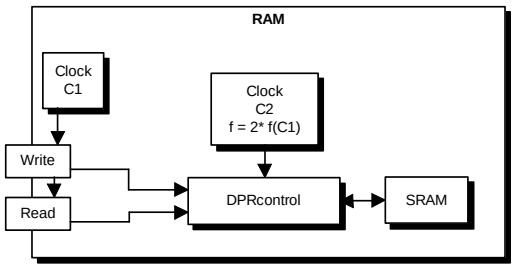

Bild 4.2. Definition von Abhängigkeiten zwischen Komponenten am Beispiel gekoppelter Taktgeneratoren

Die dargestellte Komponente stellt die Funktionalität eines RAM-Bausteins bereit, der es erlaubt, gleichzeitige Lese- und Schreibzugriffe unabhängig voneinander durchzuführen. Um diese Funktionalität zu beschreiben, verwendet diese Komponente vier weitere Komponenten, wobei die eigentliche Speicherfunktion durch die DPRcontrol-Komponente wahrgenommen wird. Sie benutzt für die Realisierung dieser Funktion SRAM-Speicher, der nur eine zeitliche Staffelung der Lese- und Schreibzugriffe gestattet. Aus diesem Grund ist die DPRcontrol-Komponente so ausgelegt, daß sie mit einem Takt arbeitet, der doppelt so hoch wie die Zugriffsfrequenz auf die Lese- bzw. Schreibfunktion der RAM-Komponente ist. Somit können in der Zeit eines Lese- bzw. Schreibzugriffes zwei getrennte Zugriffe auf das SRAM durchgeführt werden, was nach außen hin den Eindruck vermittelt, daß die Lese- und Schreibzugriffe zeitgleich erfolgen. Um die benötigten Taktsignale anfordern zu können, werden Komponenten vom Typ Clock verwendet, deren Aufgabe es ist, Taktsignale definierter Frequenz bereitzustellen. Die gewünschte Frequenz wird dabei als Attribut spezifiziert. C1 ist dabei so parametrisiert, daß es den gleichen Takt aufweist, wie er für den Read- und WritePort der RAM-Komponente verwendet wird. Bei C2 hingegen, kann man sehen, daß die Frequenz dieser Komponente in Abhängigkeit von der Frequenz von C1 spezifiziert ist. Aufgabe des Partitionierungsschrittes ist es nun, die abhängigen Komponenten zu finden und so zu plazieren, daß die festgelegten Eigenschaften durch die Plazierung erfüllt werden. Für derartige Aufgaben müssen dem Partitionierungsalgorithmus Funktionen zugeordnet werden, die es erlauben, die oben genannten Abhängigkeiten zu erkennen und deren Einhaltung zu prüfen. Der eigentliche Plazierungsalgorithmus bleibt dabei unverändert.

Wie bereits erläutert, kann die Implementierung einer Komponente auch dadurch erfolgen, daß für deren Beschreibung die Verkettung anderer Komponenten verwendet wird. Diese müssen in einem separaten Partitionierungsschritt auf das Zielsystem übertragen werden. Meistens ist es sinnvoll, diese Komponenten auf der selben Systemeinheit wie die übergeordneten Komponenten zu implementieren, da Verbindungen innerhalb einer Systemeinheit meist kostengünstiger und leistungsfähiger als Verbindungen zwischen verschiedenen Systemeinheiten zu realisieren sind.

Dennoch gibt es auch Situationen, in denen untergeordnete Komponenten auf anderen Systemeinheiten implementiert werden müssen. Als Beispiel hierfür soll eine Komponente dienen, die eine große, in einem RAM-Baustein abgelegte, mit konstanten Werten vorbelegte Tabelle enthält. Da die Komponente als solche auf dem RAM-Baustein implementiert wird, aber die für ihre Funktion notwendige Initialisierung nicht selbst vornehmen kann, wird sich ihre Realisierung aus zwei Unterkomponenten zusammensetzen. Eine Komponente reserviert den benötigten Speicherraum innerhalb des RAM-Bausteins, die andere ist für die Initialisierung des Speicherbereichs zuständig und wird deshalb auf eine mit dem RAM-Baustein verbundene Systemeinheit abgebildet, die in der Lage ist, die gewünschte Initialisierungsfunktion durchzuführen.

Eine Komponente kann aber auch, um z.B. die geforderte Performance zu erzielen, festlegen, daß alle Unterkomponenten auf der selben Systemeinheit abgebildet werden.

Um Ports auf dem Zielsystem verbinden zu können, müssen sie Elementen des Zielsystems zugeordnet werden. Dies wird dadurch erreicht, daß beim Plazieren einer Komponente auf einer Systemeinheit, jedem Port der Komponente (und auch jedem Unterport) ein Systemport zugeordnet wird. Bei einem Systemport kann es sich um einen physikalischen oder einen symbolischen Anschluß handeln. Physikalische Anschlüsse werden hauptsächlich dann verwendet, wenn es sich bei der Systemeinheit um einen Baustein mit fest vedrahteter Funktion handelt, wie dies bei Addierern und RAM-Bausteinen der Fall ist. Hier muß gezwungenermaßen der Port der Addierer-Komponente, der das Ergebnis transportiert, mit dem Anschluß (Pin) des Addierer-Bausteins übereinstimmen, der die Ergebnisse der Addition liefert. Anders verhält es sich bei den meisten programmierbaren Systemeinheiten wie z.B. den FPGAs. Hier besteht kein vorgegebener Zusammenhang zwischen implementierter Funktion und den zu verwendeten Pins des Bausteins. In diesem Fall werden symbolische Systemports angelegt, die zum Ausdruck bringen, daß der zugeordnete Port der Komponente innerhalb der Systemeinheit realisiert ist und von dort aus mit weiteren internen Systemports aber auch mit den Systemports verbunden werden kann, die die physikalischen Anschlüsse der Systemeinheit repräsentieren.

Die Implementierung der Komponente muß dann so ausgelegt sein, daß sie die dem jeweiligen Port zugeordneten Daten auch am festgelegten Systemport bereitstellt.

Systemports bilden die Grundlage für das im nächsten Abschnitt behandelte Routing.

4.2 Routing

Das Routing teilt sich in die Bestimmung der Porttypen eines Knoten und das eigentliche Verlegen der Verbindungen auf dem Zielsystem. Zusätzlich muß definiert werden, in welcher Reihenfolge diese Verarbeitungsschritte auf die Gesamtheit aller Knoten innerhalb der Lösungsbeschreibung angewendet werden.

4.2.1 Bestimmung der Porttypen eines Knoten

Zur Bestimmung eines geeigneten Typs für die über einen Knoten verbundenen Ports, müssen sowohl die Eigenschaften des gewählten Porttyps, die Eigenschaften der Komponenten und auch die Eigenschaften des Zielsystems berücksichtigt werden.

Um nun die Porttypen für einen Knoten zu bestimmen, werden alle Komponenten, die mit mindestens einem Port an diesem Knoten teilhaben, über die verwendbaren Porttypen befragt. Dies wurde bereits in Abschnitt 3.2 beschrieben. Das dort geschilderte Vorgehen stellt sicher, daß für alle Knoten ein Porttyp bestimmt werden kann. Vernachlässigt werden aber andere Eigenschaften, wie die Auswirkung einer Typenwahl auf die resultierende Leistungsmerkmale des Knotens, die Leistungsfähigkeit des Gesamtsystems und die dadurch bedingten Implementierungskosten.

Um diese Anforderungen in den Entscheidungsprozeß einbeziehen zu können, müssen die von den Porttypen bereitgestellten Informationen erweitert werden. Bisher liefern die Porttypen nur Informationen, die sich auf die grundlegenden Kommunikationseigenschaften beziehen und notwendig sind, um die Übereinstimmung der Typen der an einer Kommunikation beteiligten Ports sicherzustellen. Andere Eigenschaften betreffen nicht den Kommunikationsmechanismus als solchen, sondern bezeichnen Eigenschaften des konkreten Ports, die hauptsächlich durch die Implementierung der Komponente bestimmt sind.

Bei diesen Informationen handelt es sich z.B. um den über diesen Port erreichbaren Datendurchsatz und die resultierenden Implementierungskosten. Der maximale Datendurchsatz wird zwar vom Kommunikationsverfahren her festgelegt, der effektive Datendurchsatz ist aber weiterhin von der Implementierung der Komponente und deren Verarbeitungsleistung abhängig. Weitere zu berücksichtigende Einschränkungen ergeben sich bei Komponenten, die Daten entgegennehmen, verarbeiten und die Ergebnisse über Ports an andere Komponenten weiterleiten. Hier kann die erzielbare Datenrate auf den Eingangsports dadurch vermindert werden, daß die Ergebnisse nicht schnell genug über die Ausgangsports weitergereicht werden können.

Ähnlich ist der Sachverhalt im Bezug auf die durch die Wahl eines Porttypen verursachten Implementierungskosten. Die unmittelbaren Auswirkungen bestehen darin, daß für einen gewählten Porttyp eine definierte Menge an Verbindungsressourcen benötigt wird. Mittelbare Effekte ergeben sich daraus, daß die Festlegung eines Porttyps die Anzahl der möglichen Implementierungsvarianten der am Knoten teilhabenden Komponenten einschränkt. Werden dadurch Implementierungsvarianten mit niedrigen Implementierungskosten ausgeschlossen, erhöht sich der minimale Implementierungsaufwand für diese Komponenten und das Gesamtsystem.

Um nun die benötigte Information in den Porttyp mit aufnehmen zu können, wird eine Zweiteilung der Attribute eines Porttyps in Typ-Attribute und Port-Attribute vorgenommen (Bild 4.3).

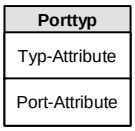

Bild 4.3. Attributklassen von Porttypen

Um die möglichen Porttypen eines Knotens zu bestimmen,wird für jeden mit dem Knoten verbundene Port erfragt, welche Typen für ihn realisiert werden können. Jeder Porttyp, der für alle Ports des Knotens verfügbar ist, wird in eine Liste, die Typ-Liste übernommen. Hierbei werden die Typ-Attribute kopiert, für die Port-Attribute hingegen sind andere Operationen notwendig. Dies soll an dem Beispiel aus Bild 4.4 näher erläutert werden.

Es sollen die für einen gegeben Knoten möglichen Porttypen ermittelt werden. Alle Ports unterstützen den Porttyp A, spezifizieren aber zusätzlich die Portattribute Cost und Datarate. Das Attribut Cost ist ein Maß für die Implementierungskosten, das Attribut Datarate hingegen gibt den Datendurchsatz an, wobei ein größerer Wert einen höheren Datendurchsatz darstellt. Ein geeignetes Maß für die Implementierungskosten ist die Summe der Cost Werte der Porttypen aller angeschlossenen Ports, woraus sich im dargestellten Beispiel ein Wert von 10 ergibt. Anders verhält es sich bei der Bewertung des erreichbaren Datendurchsatzes. Hier wäre das Minimum, im gegebenen Fall also der Wert 5, der gelieferten Datarate-Werte als Kriterium anzunehmen. Um diesen Vorgang für alle verfügbaren Porttypen zu automatisieren, wird für jedes Kommunikationsverfahren eine Porttypen-Definition erstellt, die Angaben zu allen kommunikationsrelevanten Eigenschaften wie das verwendete Kommunikationsverfahren, die verwendeten Unterports und Angaben zu den verfügbaren Typ- und Port-Attributen enthält. Für jedes Port-Attribut muß festgelegt werden, welche Operationen bei der Zusammenfassung von Porttypen auf das jeweilige Port-Attribut auszuführen sind. Ein konkreter Porttyp setzt sich dann aus einem Verweis auf die zugrundeliegende Porttypendefinition, den Typ-Attributen und den Port-Attributen zusammen (Bild 4.5).

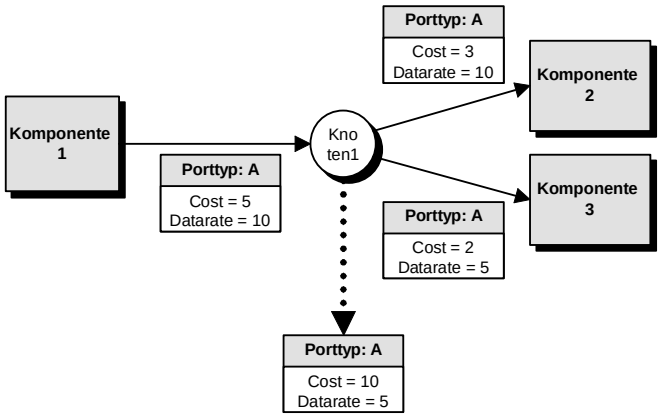

Bild 4.4. Verarbeitung der Port-Attribute bei der Bestimmung der möglichen Porttypen für einen Knoten

Zwei Porttypen sind kompatibel, wenn sie ihn ihrer Typdefinition und ihren Typ-Attributen übereinstimmen.

Bei dem hier vorgestellten Vorgehen zur Ermittlung der Porttypen eines Knotens müssen nicht nur die Auswirkungen auf die unmittelbar verbundene Ports und Komponenten berücksichtigt werden, sondern auch die Auswirkungen auf das Gesamtsystem. Dies wird dadurch erreicht, daß eine Komponente, um die möglichen Porttypen eines Ports ermitteln zu können, die möglichen Porttypen aller anderen Ports der selben Komponente einbeziehen muß. In Bild 4.6 ist dieser Vorgang dargestellt.

Ausgangspunkt ist die Anfrage an Komponente0, die möglichen Porttypen für ihren Port P1 zu bestimmen. Weiterhin sei vorrausgesetzt, daß die für Port P1 verfügbaren Porttypen durch die Wahl des Typs für Port P2 beeinflußt werden, da die verfügbaren Implementierungsvarianten nur eine beschränkte Anzahl der Kombinationsmöglichkeiten der Porttypen zulassen. Da für P2 noch kein Porttyp festgelegt wurde, wird der zugehörige Knoten zur Ermittlung der von den angeschlossenen Ports unterstützten Typen herangezogen. Hierbei wird natürlich der Port P2, von der Anfrage durch den Knoten ausgeschlossen, da ja zur Bestimmung der Typen von P2, die Typen von P1 zu bestimmen wären. Dies ist aber gerade der Ausgangspunkt unserer Berechnung und würde somit zu einer Endlosschleife führen.

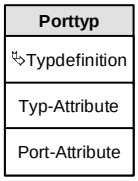

Bild 4.5. Konkreter Porttyp mit Porttypendefinition und Attributen

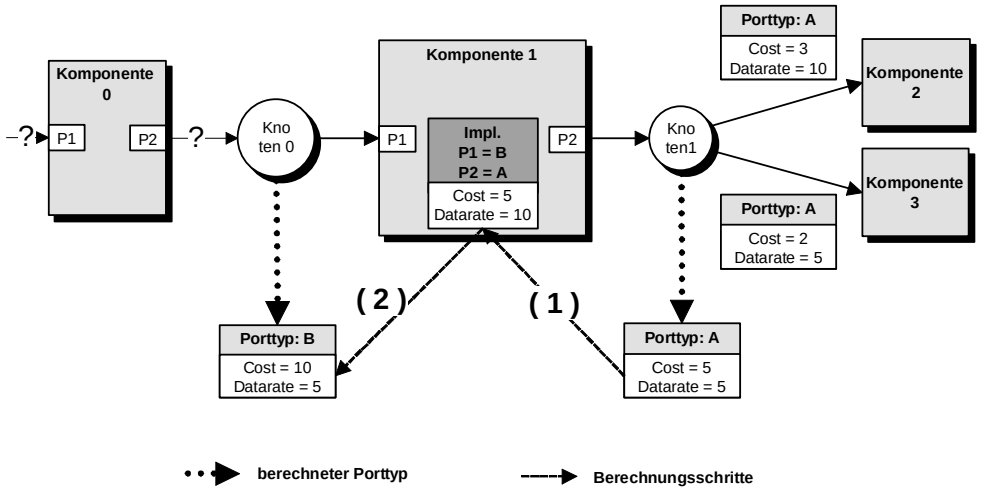

Bild 4.6. Einbeziehung des Gesamtkontextes in die Bestimmung der Porttypen

Der Knoten 0 führt also eine Anfrage für Komponente1.P1 durch. Da die Implementierungsvarianten von Komponente1 Abhängigkeiten zwischen den Typen der Ports P1 und P2 definieren, bestimmt die Komponente zuerst die möglichen Typen von P2. Auch hier sind nur die Vorgaben der anderen an den Knoten (Knoten 1) angeschlossenen Ports von Bedeutung, so daß der Port Komponente1.P2 von der Berechnung des Porttyps ausgenommen wird. Hierdurch werden die für den Knoten 1 möglichen Typen und somit auch die für den Port Komponente1.P2 möglichen Typen bestimmt. Komponente1 überprüft nun, welche Implementierungsvarianten die für P2 ermittelten Typen unterstützen und bestimmt daraus die möglichen Typen für P1. Die Port-Attribute für P1 ergeben sich aus der Verknüpfung der Port-Attribute von P2 und den Eigenschaften der passenden Implementierungsvariante. Im gegeben Fall sind hiermit auch die möglichen Porttypen für Knoten 0 und damit auch für Komponente0.P2 bestimmt. Dies ermöglicht Komponente0 die Typen für Komponente0.P1 zu ermitteln.

Der hier dargelegte Ansatz ermöglicht es, die Problematik der Porttypenbestimmung lokal für einen Knoten zu betrachten und dabei doch eine globale Optimierung in Bezug auf die Implementierungskosten und Leistungsmerkmale des Gesamtsystems zu erreichen. Nach dem der für einen Knoten am besten geeignete Porttyp ausgewählt wurde, wird dieser allen angeschlossenen Ports als aktueller Porttyp zugewiesen und damit auch gleichzeitig die dem Porttyp zugeordneten Unterports erzeugt. Die Bestimmung der Porttypen wird nun ebenfalls für alle verbundenen Unterports durchgeführt. Führt dies bei einem Unterport zu keiner Lösung, muß für den übergeordneten Knoten ein anderer Porttyp gewählt werden und die Bestimmung der Typen der Unterports von neuem gestartet werden. War dieser Vorgang erfolgreich, kann mit dem Verlegen der Verbindungen auf dem Zielsystem begonnen werden.

4.2.2 Verlegen der Verbindung auf dem Zielsystem

Für die Verlegung der Verbindungen müssen sowohl die Art und Weise, wie die Verbindungsmöglichkeiten auf dem Zielsystem innerhalb der Zielsystembeschreibung spezifiziert, als auch das eigentliche Vorgehen bei der Bestimmung des Verbindungspfades festgelegt werden.

4.2.2.1 Beschreibung der Verbindungsressourcen

Die Zielsystembeschreibung setzt sich aus den Systemeinheiten zusammen. Diese sind in der Lage, die Implementierung der ihnen zugeordneten Komponenten zu realisieren. Zusätzlich müssen sie die benötigten Portverbindungen zwischen den Komponenten erstellen.

Je nach Systemkomponente existieren sehr unterschiedliche Verbindungsmöglichkeiten, die sich von ihrem Umfang her, von den verwendbaren Kommunikationsverfahren, den über die realisierten Verbindungen transportierbaren Datentypen und in der Art der Beschreibung der Verbindungen mittels der gewählten Implementierungssprache unterscheiden.

So stellt ein FPGA hauptsächlich ein Bit breite, gerichtete Verbindungen zur Verfügung. Auf einer CPU hingegen können die Daten meist mit höherer Bitbreite und variablen Datentypen übertragen werden. Zusätzlich ist keine feste Datenrichtung vorgegeben und es werden deutlich komplexere Kommunikationsverfahren bereitgestellt.

Die erstellten Verbindungen müssen in die Implementierungsbeschreibung der jeweilige Systemeinheit aufgenommen und somit in der jeweiligen Implementierungssprache beschrieben werden.

Über die Verbindungen innerhalb einer Systemeinheit hinaus muß die Zielsystembeschreibung auch Informationen über die Verbindungen zwischen den einzelnen Systemeinheiten enthalten. Gerade bei Verbindungen zwischen Systemeinheiten, die sich in ihrer Grundstruktur und ihrem Kommunikationsverhalten stark unterscheiden, wie dies z.B. bei der Kopplung von FPGAs und CPUs der Fall ist, werfen besondere Probleme auf.

Für die weitere Diskusion dieser Problemfelder sei ein exemplarisches Zielsystem vorausgesetzt, das in Bild 4.7 dargestellt ist.Dieses System besteht aus

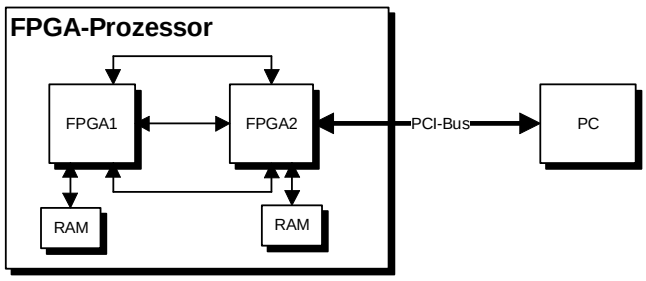

Bild 4.7. Grundlegender Aufbau eines Zielsystems

einem PC, der über einen PCI-Bus mit einem FPGA-Prozessor verbunden ist. Der FPGA-Prozessor selbst besteht aus zwei untereinander verbundenen FPGAs, die jeweils über einen getrennten Satz an Speicherbausteinen verfügen. FPGA2 ist zusätzlich noch für die Ansteuerung des PCI-Interfaces zuständig.

Um das Routing durchführen zu können, muß die Systembeschreibung um Informationen zu den verfügbaren Verbindungsmöglichkeiten erweitert werden, was in einem ersten Schritt dadurch erreicht wird, daß jeder Systemeinheit ein Routingobjekt zur Seite gestellt wird (Bild 4.8).

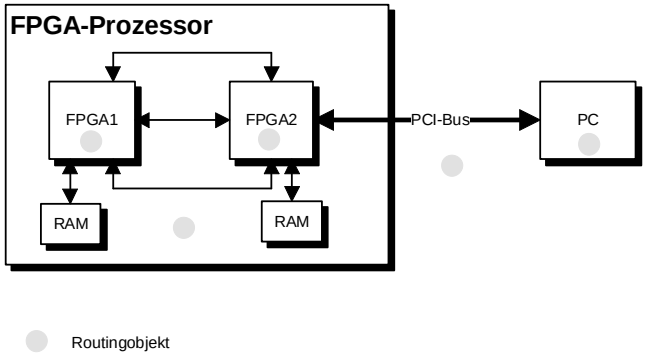

Bild 4.8. Zielsystem mit Routingobjekten

Innerhalb dieser Routingobjekte kann nun die Routinginformation für jede Systemeinheit getrennt hinterlegt werden. Das Routingobjekt ist für das Auffinden und das Verlegen von Verbindungen auf der jeweiligen Systemeinheit zuständig.

Um nun Verbindungen über die Grenzen der verschiedenen Systemeinheiten hinweg verlegen zu können, wird noch ein globaler Router benötigt, der auf die Routingobjekte aufsetzt und mit deren Hilfe die einzelnen Teilverbindungen realisiert. Da die eigentliche Funktion des globalen Routers in Abschnitt 4.2.2.3 behandelt wird, soll hier nur die Schnittstelle zwischen den Routingobjekten und dem globalen Router näher betrachtet werden.

Ein Routingobjekt muß angeben, ob es eine Verbindung zwischen zwei Systemports herstellen kann, welche verschiedene Pfade es hierfür gibt, welche Implementierungskosten die jeweilige Verbindung verursacht und wie hoch die erreichbare Übertragungsrate über diesen Pfad ist.

Anhand dieser Merkmale kann dann der globale Router entscheiden, welche der möglichen Verbindungen verwendet werden sollen. Die Aufgabe,die eigentliche Verbindung zu realisieren, wird wiederum vom Routingobjekt wahrgenommen.

Da die Zuständigkeit eines Routingobjektes auf eine Systemeinheit beschränkt ist, kann es nur dann zwei Systemports verbinden, wenn sich diese auf der gleichen Systemeinheit befinden. Diese Forderung ist auch erfüllt, falls sich zwei Systemports auf verschieden Systemeinheiten befinden, aber bereits eine Verbindung des einen Systemports zur Systemeinheit des andern Systemports existiert.

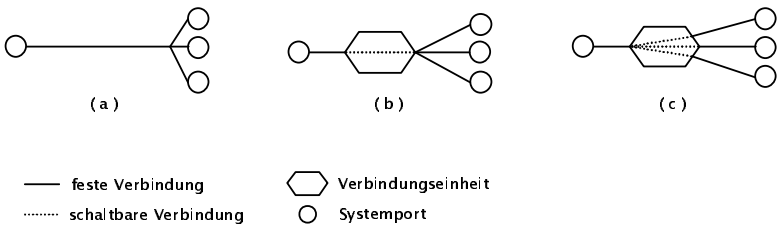

Bild 4.9. Verschiedene Arten von Verbindungsmöglichkeiten zwischen Systemports (a) feste Verbindung, (b) schaltbare Punkt zu Punkt Verbindung, (c) schaltbare Mehrfacherbindung

In allen anderen Fällen ist das direkte Verlegen der Verbindung nicht möglich, so daß das Routingobjekt lediglich angeben kann, welche Systemports vom Startport aus erreichbar sind. Um diese Aufgaben zu erfüllen zu können, muß das Routingobjekt über eine Liste der möglichen Verbindungen auf der Systemeinheit verfügen, wobei für jede Verbindung folgende Angaben bereitgestellt werden müssen:

- Verbindung belegt/frei
- erreichbare Systemports
- unterstützte Porttypen
- Kosten
- Übertragungsleistung

Im wesentlichen sind hierbei drei verschiedene Arten von Verbindungen zu unterscheiden (Bild 4.9).

Die erste Klasse besteht aus allen fest vorgegebenen, statischen Verbindungen, wie sie z.B. auf einer Leiterplatte mittels Leiterbahnen realisiert sind.

Die andere Klasse besteht aus schaltbaren Verbindungen, die mittels einer Verbindungseinheit hergestellt werden können. Hierbei muß die Verbindungseinheit so konfiguriert werden, daß sie die gewünschte Verbindung realisiert und den durch die verbundenen Ports vorgegebenen Porttyp unterstützt. Derartige Verbindungen findet man z.B. in programmierbaren Logikbausteinen aber auch in konfigurierbaren Verbindungsbausteinen wie Crossbarswitches.

Bei FPGAs besteht z.B. die Möglichkeit einen beliebigen internen Systemport mit einem externen Systemport[1], der einen I/O-Pin des FPGA repräsentiert, zu verbinden. Hier ist es nicht ausreichend, die Verbindung zu verlegen, sondern es

[1]Interne Systemports sind Systemports, die nicht direkt mit einem Systemport anderer Systemeinheiten verbunden werden können, wie dies z.B. bei Signalen auf einem FPGA gegeben ist, die erst mit einem I/O-Pin verbunden werden müssen, um zu anderen Systemeinheiten verbunden werden zu können. Bei FPGAs würden also die I/O-Pins externen Systemports und die internen Signale internen Systemports zugeordnet werden.

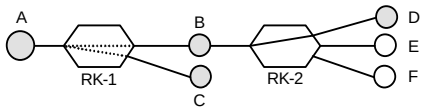

Bild 4.10. Behandlung von bereits verwendeten Komponenten bei der Bestimmung der erreichbaren Systemports

muß auch noch der I/O-Pin dahingehend konfiguriert werden, daß er als Eingang oder Ausgang fungiert. Fungiert der interne Systemport z.B. als Datenquelle, muß der I/O-Pin über den externen Systemport als Ausgang konfiguriert werden. Um nun diese Verbindungsmöglichkeiten den Routingobjekten bekannt zu machen, werden Routingkomponenten erzeugt, die im wesentlichen die Eigenschaften der Verbindungseinheiten modellieren.

Eine Routingkomponente verfügt über Informationen welche Systemports sie verbinden kann, welche Porttypen für die Verbindung zulässig sind, welche Leistungsdaten die Verbindung aufweist, wie hoch die Implementierungskosten für diese Verbindung sind und wie die jeweilige Verbindungseinheit zu programmieren ist. Auch muß die Routingkomponente protokollieren, ob sie bereits eine Verbindung realisiert (geschaltete Verbindung) bzw. ob die repräsentierte Verbindung bereits genutzt wird (feste Verbindung).

Zusätzlich verfügt die Routingkomponente über Schnittstellen, mit denen ein Routingobjekt feststellen kann, ob eine Routingkomponente bereits benutzt wird und welche anderen Systemports für einen gegeben Systemport über diese Routingkomponente erreichbar sind. Eine Routingkomponente, über die bereits eine Verbindung realisiert ist, muß unter den erreichbaren Systemports auch diejenigen aufführen, die an realisierten Verbindungen teilhaben (Bild 4.10). Auf die Anfrage, welche Systemports für den Systemport A über die Routingkomponente RK-1 erreichbar sind, muß diese die Systemports B, C, D nennen. Ausgangspunkt hierzu ist, daß RK-1 über die Information verfügt, daß es selbst den Systemport A mit den Systemports B und C verbinden kann. Da nun aber der Systemport B über RK-2 auch mit dem Systemport D verbunden ist[2], kann RK-1 natürlich auch die Systemports A und C verbinden, in dem sie eine Verbindung zwischen A und B herstellt.

Wie in Abschnitt 4.2.2.3 gezeigt wird, muß es möglich sein, einzelne bereits verlegte Verbindungen wieder aufzulösen. Um dies zu ermöglichen, muß jede Routingkomponente für die durch sie realisierten Verbindungen einen Zähler führen, der angibt, wie oft diese Verbindung verwendet wird. Wird eine neue Verbindung hergestellt, oder eine bereits bestehende Verbindung erneut genutzt, wird dieser Zähler um eins erhöht. Beim Freigeben einer Verbindung wird dieser Wert um eins erniedrigt. Erst wenn der Zähler wieder seinen Ausgangswert erreicht hat, darf die Verbindung aufgelöst werden. Ausgehend von der Konfiguration in Bild

[2] Um diese Berechnung durchführen zu können, ist es notwendig, daß für jeden Systemport die ihm zugeordneten Routingkomponenten bestimmt werden können.

4.10, wobei davon ausgegangen wird, daß bisher noch keine Verbindung verlegt wurde, also die Verbindung von A nach B noch nicht existiert, sollen folgende Schritte ausgeführt werden:

1. Verbinden von B und D
2. Verbindung von A nach D herstellen
3. Verbindung von A nach D freigeben
4. Verbindung von B nach D frei geben

Nach der Durchführung der entsprechenden Schritte ergeben sich folgende Zählerwerte(Ausganswerte: $\text{Zähler}_{AB} = 0$, $\text{Zähler}_{BD} = 0$):

1. $\text{Zähler}_{AB} = 0$, $\text{Zähler}_{BD} = 1$
2. $\text{Zähler}_{AB} = 1$, $\text{Zähler}_{BD} = 2$
3. $\text{Zähler}_{AB} = 0$, $\text{Zähler}_{BD} = 1$
4. $\text{Zähler}_{AB} = 0$, $\text{Zähler}_{BD} = 0$

In Schritt 3 wird die Verbindung zwischen A und B, in Schritt 4 die Verbindung zwischen B und D aufgelöst, da hier die zugeordneten Zähler ihren Ausgangswert erreichen.

Während sich mit dem bisher dargestellten Lösungsansatz schon die meisten Routingaufgaben lösen lassen, gibt es eine besondere Gruppe von Verbindungsmöglichkeiten, die getrennt betrachtet werden müssen. Hierbei handelt es sich um Verbindungen, die nicht wie die bisher betrachteten Verbindungen eine leitungsorientierte Kommunikation ermöglichen, sondern Kommunikationsvarianten implementieren, bei denen Quelle und Ziel durch eine Kennung innerhalb der transportierten Information repräsentiert werden. Solch eine Konstellation findet sich in dem hier exemplarisch betrachteten Zielsystem bei der Kopplung des FPGA 2 mit dem PC über den PCI-Bus (Bild 4.7). Das übliche Vorgehen ist hierbei, die Daten über den PCI-Datenbus im Rahmen eines Schreib- oder Lesezugriffes zu transferieren, wobei der Kommunikationspartner durch die auf dem PCI-Adressbus angelegte Adresse eindeutig identifiziert wird. Um auch bei solchen Verbindungen leitungsorientierte Kommunikationsvarianten unterstützen zu können, wird eine neue Klasse von Routingkomponenten eingeführt, die es ermöglichen, virtuelle Verbindungskanäle zu erzeugen. Deren grundlegende Struktur ist in Bild 4.11 dargestellt.

Eine wichtige Eigenschaft dieser Routingkomponenten besteht darin, daß sie mehreren Routingobjekten und damit auch mehreren Systemeinheiten zugeordnet sein können. Der wesentliche Schritt zur Erstellung der virtuellen Kanäle besteht nun darin, daß der nicht kanalorientierte Kommunikationspfad mittels zweier aktiver Komponenten gekapselt wird.

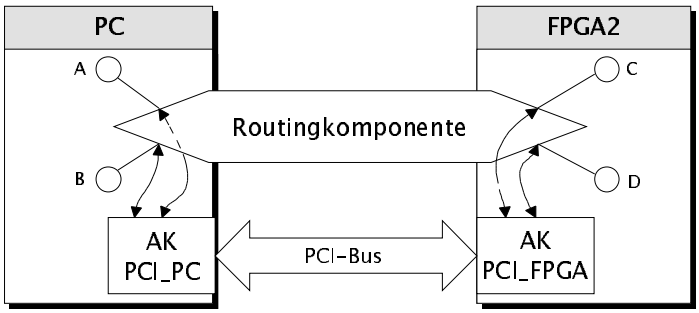

Bild 4.11. Virtuelle Verbindungskanäle, realisiert durch aktive Komponenten (AK)

An einem Ende des Pfades findet eine Abbildung der kanalorientierten Kommunikation auf den zu kapselnden Kommunikationsmechanismus statt und am anderen Ende wird diese Abbildung wieder rückgängig gemacht. Für die Komponenten, die über den so erzeugten Pfad kommunizieren, ist dieses Vorgehen vollständig transparent und auch durch das Routingobjekt können derartige Verbindungen wie gewöhnliche Verbindungen gehandhabt werden. In den meisten Fällen lassen sich über einen Verbindungspfad mehrere virtuelle Verbindungskanäle aufbauen, was in dem hier aufgeführten Beispiel dadurch erreicht werden kann, daß jedem Kanal eine eigene Adresse (oder Adressbereich) zugeordnet wird, so daß bei jedem Zugriff auf den PCI-Bus anhand dieser Adresse die jeweiligen Adressaten bestimmt werden können.

Mit Hilfe der Routingkomponenten läßt sich sehr leicht ein universelles Routingobjekt erstellen, bei dem zur Anpassung an die jeweilige Systemeinheit nur noch die geeigneten Routingkomponenten erstellt werden müssen. Während dieser Ansatz zur Beschreibung von Verbindungen, die mindestens einen physikalischen Systemport umfassen, ausreichend ist, müssen Verbindungen zwischen dynamisch erzeugten, symbolischen Systemports anders behandelt werden. Diese Art von Systemports findet z.B. bei FPGAs ihre Anwendung, wenn eine aktive Komponente auf diesem plaziert wird. Hiebei wird für jeden Port der aktiven Komponente ein entsprechender Systemport erzeugt. Diese Systemports können aber keiner konkreten Position innerhalb des FPGAs zugeordnet werden, sondern es wird lediglich ein symbolischer Bezeichner vergeben[3]. Die meisten FPGAs sind in der Lage, diese Systemports untereinander frei zu verbinden. Um diese Eigenschaft zu unterstützen, werden die Routingobjekte durch die Möglichkeit ergänzt, bei Bedarf die benötigten Routingobjekte zur Verbindung dieser Systemports zu

[3]Die Zuordnung des Systemports zu den real vorhandenen Implementierungsressourcen des FPGAs wird erst bei der unmittelbaren Abbildung der Implementierung auf das FPGA durchgeführt, wobei die hierfür benötigte Software in den meisten Fällen vom FPGA-Hersteller bereitgestellt wird. Einige Hersteller ermöglichen es dem Programmierer, die Zuordnung der Implementierung zu den Systemressourcen zu beeinflussen bzw. auch exakt vorzugeben. Diese Möglichkeit wird nur in wenigen Fällen angewendet, da zum einen eine genaue Kenntnisse über den internen Aufbau des FPGAs vorliegen muß und dieses Vorgehen für größere Implementierungen sehr aufwendig ist. Auch geht hierdurch die Portierbarkeit verloren.

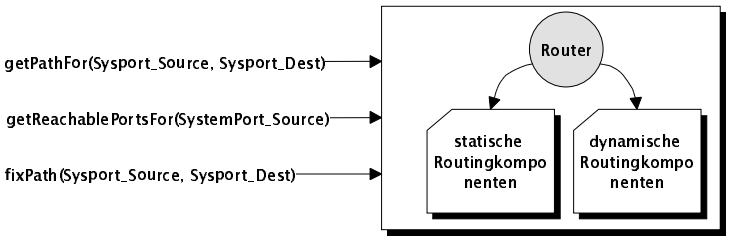

Bild 4.12. Aufbau der Routingobjekte

erzeugen. Der hieraus resultierende Aufbau der Routingobjekte ist noch einmal in Bild 4.12 dargestellt.

Mit dem hier dargestellten Verfahren ist es möglich, Punkt zu Punkt Verbindungen innerhalb einer Systemeinheit und zwischen angrenzenden Systemeinheiten zu verlegen. Die zusätzliche Funktionalität, die zur Verlegung ganzer Knoten mit Verbindungen zu beliebigen Systemeinheiten notwendig ist, soll im nächsten Abschnitt näher erläutert werden.

4.2.2.2 Routing der Knoten

Um ein universelles Routingsystem aufzubauen, müssen Mechanismen gefunden werden, die folgende Aufgaben erfüllen:

- Verlegen von Knoten mit mehr als zwei Kommunikationspartnern
- Verlegen von Verbindungen über mehrere Systemeinheiten hinweg
- Behandlung hierarchisch aufgebauter Porttypen
- Verlegung von Verbindungen bei hierarchisch aufgebauten aktive Komponenten
- Verlegbarkeit aller Knoten muß sichergestellt werden

Wie bereits dargelegt, unterstützen Routingobjekte nur eins-zu-eins Verbindungen. Um auch Knoten mit mehr als zwei Kommunikationspartnern behandeln zu können, müssen diese auf eins zu eins Verbindungen heruntergebrochen werden.

Hierzu teilt man die zu verbindenden Systemsports in Gruppen ein. Zu einer Gruppe gehören alle Systemports, die bereits untereinander verbunden sind. Nun wählt man aus jeder Gruppe einen Repräsentanten und erhält so eine Menge von Systemports, die noch miteinander verbunden werden müssen, um den Knotens zu realisieren.

Aus dieser Menge wählt man wiederum einen Port aus und versucht eine Verbindung zu einem beliebigen anderen Systemport aus dieser Menge herzustellen. Nach erfolgreichem Verlegen dieser Verbindung, wird die Gruppeneinteilung den neuen Gegebenheiten angepaßt und die zuvor genannten Arbeitschritte so oft

durchgeführt, bis nur noch eine Gruppe vorhanden ist und somit alle für den Knoten notwendigen Verbindungen erzeugt wurden.

Ähnlich muß beim Verlegen von Verbindungen über mehrere Systemeinheiten hinweg vorgegangen werden. Auch hier kann wieder von der Aufgabenstellung der Verbindung zweier Systemports ausgegangen werden. Beginnend von einem der Systemports, ermittelt man für diesen die zugehörige Systemeinheit und damit auch das entsprechende Routingobjekt. Das Routingobjekt liefert die Aussage, ob es die gewünschte Verbindung verlegen kann bzw. welche externen Systemports für den angegebenen Systemport erreichbar sind. Kann die Verbindung hergestellt werden, ist die Aufgabe erfüllt.

In den anderen Fällen geht man so vor, daß von den erreichbaren externen Systemports aus, weitere Versuche unternommen werden, den Zielport zu erreichen. Man nutzt dabei aus, daß ein externer Systemport zwei Systemeinheiten zugeordnet ist, wobei die eine der anderen hierarchisch übergeordnet ist.

Mittels eines externen Systemports gelangt man also von einer untergeordneten Systemkomponente wie FPGA2 (Bild 4.8) zur übergeordneten Systemeinheit,dem FPGA-Prozessor. Auf dieser Ebene kann man dann eine Verbindung zu anderen Systemeinheiten auf dieser Ebene herstellen und von dort aus wiederum über einen erreichbaren externen Systemport auf die Ebene der gesuchten Systemeinheit hinabsteigen, bzw. eine Ebene aufsteigen, wenn die gewünschte Systemeinheit sich nicht auf dieser oder einer untergeordneten Ebenen befindet.

Auf diese Weise kann man alle möglichen Verbindungen bestimmen. Nachdem die gewünschte Verbindung ausgewählt wurde, werden alle beteiligten Routingobjekte angewiesen, den ihnen zugeordneten Teil der Gesamtverbindung zu realisieren.

Wie bereits dargelegt, können Porttypen und damit auch die zugeordneten Ports hierarchisch aufgebaut sein. Bedeutsam wird dies dann, wenn es nicht möglich ist eine Verbindung zwischen Ports eines gegeben Typs herzustellen.

Definitionsgemäß ist die Verbindung zwischen Ports äquivalent zu der eins-zu-eins Verbindung ihrer Unterports. Dies eröffnet die Möglichkeit, die gewünschten Verbindungen durch eine Verbindung der Unterports herzustellen. Konnte also eine Verbindung auf direktem Wege nicht realisiert werden, kann ein weiterer Routingversuch unter Zuhilfenahme der Unterports durchgeführt werden.

Wie in Abschnitt 4.3 näher erläutert, kann die Implementierung einer aktiven Komponente auch mittels eines Netzes bestehend aus andern aktiven Komponenten beschrieben werden, woraus sich dann eine hierarchische Beschreibung der Komponenten ergibt.

Eine Komponente kann erst dann ihre Implementierung erzeugen, wenn alle ihre Porttypen bestimmt und die entsprechenden Verbindungen verlegt wurden, das Routing also bereits abgeschlossen ist. Die Komponente fügt dann, neben den zur Implementierungsbeschreibung benötigten Komponenten, auch weitere Verbindungen und Knoten hinzu, wobei einige von ihnen Verbindungen zu Ports der übergeordneten aktiven Komponente repräsentieren.

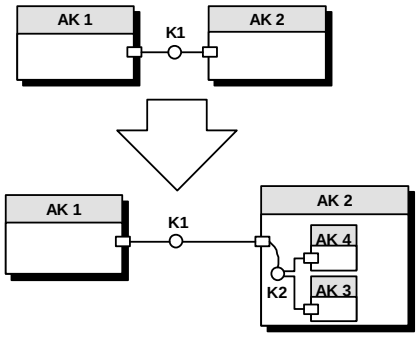

Bild 4.13. Neu erzeugte Knoten bei der hierarchischen Implementierung von aktiven Komponenten

Dieser Zusammenhang ist in Bild 4.13 dargestellt. Bei der Durchführung der Implementierung von AK2 wird ein neuer Knoten K2 erzeugt.

Bei der Verlegung von K2 müssen einige Sonderfälle beachtet werden. Die Problematik besteht darin, daß von Beginn an nicht die endgültige Verbindung verlegt wird, sondern dies in mehreren Zwischenschritten erfolgt. Im obigen Beispiel führt die endgültige Verbindung von AK1 nach AK3 und AK4. Verlegt wird aber eine Verbindung von AK1 nach AK2 und erst im Anschluß daran eine Verbindung von AK2 nach AK3 und AK4. Dieses Vorgehen kann in einigen Fällen zu nicht optimalen Verbindungen führen.

Als Beispiel soll angenommen werden, daß AK1 auf FPGA1 und AK2 auf FPGA2 plaziert wurden (hier wird das Zielsystem aus Bild 4.7 vorausgesetzt). Bei der Realisierung von K1 wird eine Verbindung von FPGA1 zu FPGA2 belegt. Nach dem nun die Implementierung von AK2 durchgeführt wurde, sollen AK3 auf FPGA2 und AK4 auf FPGA1 plaziert werden. Die nach dem Verlegen von K2 resultierende Konstellation ist in Bild 4.14 dargestellt.

Das resultierende Routingergebnis ist nicht optimal, da eine überflüssige Verbindung von FPGA1 zu FPGA2 erzeugt wird. Daß für den Port von AK2 ein symbolischer Systemport angelegt wird, ist hingegen ohne Bedeutung, da er keine realen Ressourcen belegt.

Um hier zu einem besseren Ergebnis zu kommen, müssen die Verbindungen, die zur Realisierung von K1 verwendet wurden, aufgelöst und aus den beiden Knoten K1 und K2 ein neuer temporärer Knoten erzeugt werden, der die tatsächlich zu verbindenden Systemsports enthält. Wenn nun dieser Knoten neu verlegt wird, kann ein optimales Routingergebnis erzielt werden.

Diese Fälle sollten nicht die Regel sein, da die zur Implementierung verwendeten aktiven Komponente aus Gründen der besseren Performance und des geringeren Verbrauches an Verbindungsressourcen meist auf der gleichen Systemeinheit wie ihre übergeordnete Komponente implementiert werden.

Dennoch gibt es Systemeinheiten, bei denen auch in diesem Fall ein Neuverlegen sinnvoll sein kann. Da hierbei gegebenenfalls ein anderer externer Systemport als Zugang zur Systemeinheit gewählt wird, führt dies zu einem anderen

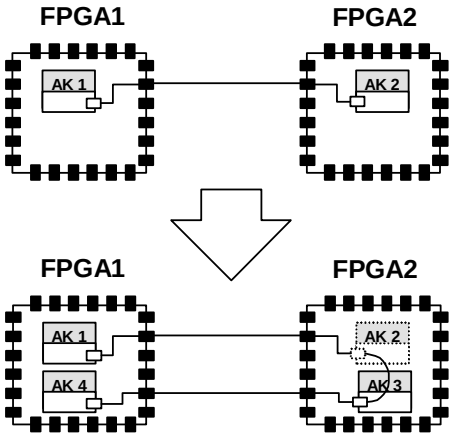

Bild 4.14. Probleme beim Routing von aktiven Komponenten mit mehreren Implementierungsebenen

Verbindungspfad auf der Systemeinheit, der möglicherweise geringere Implementierungskosten verursacht. Dieser Fall tritt aber nur bei Systemeinheiten auf, bei denen Ports fixe, physikalische, interne Systemports zugeordnet werden.

Es bleibt festzuhalten, daß durch die Möglichkeit der hierarchischen Implementierung der aktiven Komponenten ein an den Hierarchieebenen orientierter Routingprozeß notwendig ist. Er kann zwar dazu führen, daß bereits verlegte Verbindungen wieder aufgelöst werden müssen, stellt aber immerhin sichert, daß die benötigten Routingressourcen reserviert werden, bevor die Implementierung der entsprechenden Komponenten erfolgt. Dies gewährleistet, daß Engpässe bei den Routingressourcen so früh wie möglich erkannt werden.

4.2.2.3 Gesamtablauf des Routings

Nachdem die benötigten Schritte zur Realisierung eines Knotens auf dem Zielsystem dargelegt wurden, sollen diese nun in einen Gesamtablauf eingegliedert werden, der dafür Sorge trägt, daß alle Knoten der Lösungsbeschreibung erfolgreich verlegt werden können.

Hierzu wird zuerst eine Liste der vorhandenen Knoten erstellt und die folgendenen Schritte ausgeführt:

1. Auswahl des nächsten Knotens in der Knotenliste
2. Bestimmen der Liste der möglichen Porttypen für den ausgewählten Knoten
3. Auswählen des nächsten, möglichen Porttyps
4. Setzen des Porttyps für alle Ports des Knotens
5. Implementierung des Knotens
    - falls Routing nicht möglich, Routing über Unterports

  - für die verschiedenen Unterports des Porttyps wird jeweils ein Knoten erzeugt, der die jeweiligen Unterports der über den Knoten verbundenen Ports verbindet
  - Routing aller Unterport-Knoten
    ▶ Konnten alle Unterport-Knoten erzeugt werden, ist somit auch der übergeordnete Knoten implementiert
- Konnte der Knoten realisiert werden, Entnahme aus der Knotenliste und weiter bei Punkt 1, ansonsten weiter bei Punkt 3

6. Konnte der Knoten nicht realisiert werden, wird geprüft, welche Routingressourcen unbenutzt sein müßten, um ein Verlegen des Knoten zu ermöglichen. Die Knoten, die diese Routingressourcen verwenden, werden aufgelöst und so parametrisiert, daß sie mit diesen Ressourcen sparsamer umgehen oder ein anderer Pfad für ihre Verlegung gewählt wird. Die aufgelösten Knoten werden am Anfang der Knotenliste erneut hinzugefügt. Weiter bei Punkt 1.
7. Konnte ein Knoten nicht verlegt werden, muß der Engpaß erneut analysiert und gegebenefalls eine andere Plazierung der Komponenten gewählt werden.

Die Maßnahme in Schritt 6, bereits bestehende Verbindungen aufzulösen, wird dadurch notwendig, daß beim Verlegen eines Knotens immer nur darauf geachtet wird, wie dieser eine Knoten am effizientesten realisiert werden kann. Daß dadurch natürlich Verbindungspfade für andere Knoten verbaut werden, bleibt zunächst unberücksichtigt. Erst wenn ein Engpaß auftritt, wird diese Gegebenheit ananalysiert und Gegenmaßnahmen eingeleitet.

Die Maßnahme besteht darin, die benötigten Routingressourcen frei zu halten. Dies kann dadurch geschehen, daß beim erneuten Verlegen der Knoten, die den Engpaß hervorgerufen haben, die relevanten Routingressourcen gesperrt oder zumindest der Kostenfaktor für die Verwendung dieser Ressourcen erhöht und somit andere Pfade zum Verlegen der Knoten erzwungen oder zumindest ein sparsamerer Umgang mit den Routingressourcen motiviert wird.

4.3 Implementierung der einzelnen Komponenten

Die zuvor geschilderten Abbildungsschritte dienen dazu, die Randbedingungen unter denen die jeweilige aktive Komponente operieren muß, festzustellen und die entsprechenden Entwurfsentscheidungen zu treffen.

Für eine aktive Komponente ist also bekannt, auf welcher Systemeinheit sie implementiert werden soll und welche Verfahren zur Kommunikation mit den verbundenen Komponenten zu verwenden ist. Auch ist bereits sichergestellt, daß diese Randbedingungen zu einem auf dem gewählten Zielsystem realisierbaren Implementierung führen.

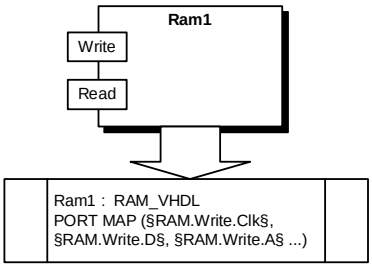

Bild 4.15. Implementierung einer aktiven Komponente in VHDL

Die Komponente muß also nur anhand der vergebenen Randbedingungen eine geeignete Implementierung treffen, was vollständig unabhängig von den anderen Komponenten geschehen kann, da deren Eigenschaften ja schon bei der Bestimmung der Randbedingungen berücksichtigt wurden.

Der Komponente stehen im Wesentlichen zwei Vorgehensweisen zur Erzeugung ihrer eigenen Implementierung zur Verfügung. Sie können hierzu eine geeignete Implementierungssprache wählen oder aber die Implementierung durch die Verkettung anderer Komponenten beschreiben.

In beiden Fällen besteht das Hauptproblem darin, wie eine Verbindung zwischen den Ports der Komponenten bzw. deren Systemports und der eigentlichen Implementierung hergestellt werden kann. Wird die Implementierung mittels einer Implementierungssprache realisiert, ist es meist so, daß eine Verbindung dadurch beschrieben wird, daß auf ein gemeinsames Objekt mit einem eindeutigen Bezeichner zugegriffen wird. In VHDL oder CHDL werden Kommunikationspartner dadurch verbunden, daß sie einem gemeinsamen Signal mit eindeutigem Namen zugewiesen werden. In C++ kann die gewünschte Verbindung so beschrieben werden, daß die Kommunikationspartner einer gemeinsamen Instanz eines für Kommunikationszwecke geeigneten Objektes zugeordnet werden. Eine Verbindung zwischen Implementierung und Systemport kann also dadurch erfolgen, daß Implementierung und Systemport dem gleichen Kommunikationsobjekt (C++-Objekt oder CHDL-Signal) zugeordnet werden.

Da zum gegebenen Zeitpunkt noch keine festen Kommunikationsobjekte vorhanden sind, wird stattdessen für die Verbindung zu einem Port ein symbolischer Namen verwendet, der sich vom Namen des entsprechenden Ports der aktiven Komponente ableitet. Dies ist in Bild 4.15 dargestellt, wobei als Implementierungssprache VHDL Verwendung findet.

In dem hier aufgeführten Beispiel wird die Implementierung einer aktiven Komponente RAM betrachtet. Diese stellt die zwei Ports Write für Schreibzugriffe und Read für die Lesezugriff zur Verfügung. Diese beiden Ports enthalten wiederum Unterports wie CLK, A, D etc.. Zur Realisierung der Komponente steht das VHDL-Modul mit RAM_VHDL zur Verfügung. In dem für die Komponente RAM erzeugten VHDL-Beschreibung wird zuerst eine Instanz des Moduls RAM_VHDL erstellt (1. Zeile). Im Anschluß daran werden die Signale des

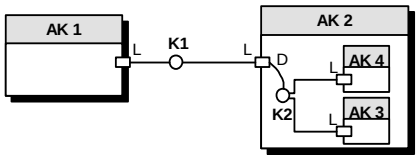

Bild 4.16. Verschiedene Verbindungsarten

VHDL-Moduls mit den entsprechenden Anschlüssen der Komponente assoziiert, was mittels der PORT MAP Anweisung erfolgt. Innerhalb dieser Anweisung sind die einzelnen symbolischen Namen aufgeführt. Jeder dieser Bezeichner wird durch einen je nach verwendeter Implementierungssprache unterschiedlichen Begrenzer eingerahmt (hier §). Der eigentliche Bezeichner entspricht dem Portnamen. Bei Ports, die einen oder mehrere übergeordnete Ports haben, wird der Name so gewählt, daß er sich aus den Namen der übergeordneten Ports und des Ports selbst zusammensetzt. Hierbei wird bei den Ports auf höchster Ebene begonnen und die einzelnen Namen durch '.' getrennt. Um nun diesen Bezeichner eindeutig zu machen, ist es noch notwendig, den Namen der Komponente voran zu stellen. Dieser wird wiederum durch '.' vom Rest des Bezeichners getrennt.

Wie die Kommunikationsobjekte angelegt und mit den Systemports verbunden werden, ist in Abschnitt 4.4 näher erläutert.

Verwendet man hingegen die Möglichkeit der hierarchischen Beschreibung von Komponenten, so muß es möglich sein, zwischen zwei Verbindungstypen zu unterscheiden.

Der eine Verbindungstyp verbindet Ports, die zu Komponenten auf gleicher Hierarchieebene gehören und somit einen Kommunikationspfad darstellen. Der andere Typ umfaßt Verbindungen von Ports, die zu Komponenten gehören, die sich auf verschiedenen Hierarchieebenen befinden. Sie stellen keinen eigentlichen Kommunikationspfad dar, sondern dienen nur dazu, eine Zuordnung der Ports auf den verschiedenen Hierarchieebenen zu ermöglichen.

Um diese Beschreibungsmöglichkeit bereitzustellen, verfügt jeder Port über die Möglichkeit, an zwei Verbindungen teilzuhaben. Die Verbindung auf gleicher Hierarchieebene wird hierbei als Level-Connection (L) und die Verbindung zwischen Hierarchieebenen als Down-Connection (D) bezeichnet (Bild 4.16).

Anhand dieser Definition ist es auch leicht möglich, einen Algorithmus zu definieren, der für einen gegeben Port einen Knoten erstellt, der alle tatsächlich verbundenen Ports enthält und alle Ports, die als Bindeglied zwischen den Ebenen dienen, entfernt. Dieser Algorithmus ist, wie bereits in Abschnitt 4.2.2.2 beschrieben, für ein optimiertes Routingergebnis notwendig. Zur Erstellung dieses optimierten Knotens sind folgende Schritte notwendig:

1. Bestimme den Knoten für den aktuellen Port (Level-Connnection)

2. Erstelle eine Liste aller an diesen Knoten angeschlossenen Ports

3. Jeder Port ohne Down-Connection wird in die Ergebnisliste aufgenommen und als besucht markiert

4. Für jeden Port mit Down-Connection, der nicht als besucht markiert ist

   - Port wird zum aktuellen Port
   - Port als besucht markieren
   - ▶ Schritt1

Dieser Algorithmus muß allerdings noch modifiziert werden, da ein Port auch als verbunden gilt,wenn eine Verbindung über einen übergeordneten Port realisiert wurde, wie das in Bild 4.17 dargestellt ist.

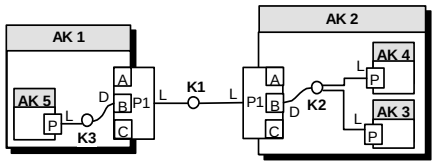

Bild 4.17. Behandlung von hierarchisch gegliederten Ports bei der Bestimmung eines optimierten Knotens

Die Modifikation besteht darin, daß zuerst ein vollständiger Knoten erzeugt wird, der alle direkt oder über mehrere Knoten hinweg miteinander verbundenen Ports enthält, wobei auch die Ports enthalten sind, die als Bindeglied zwischen den Implementierungsebenen dienen. In das Ergebnis werden natürlich nur die Ports aufgenommen, die keine Down-Connection enthalten.

Für jeden der ermittelten Ports wird gerprüft, ob ein übergeordneter Port existiert und für diesen erneut der vollständige Knoten ermittelt. Daraus werden dann die für die gesuchte Verbindung relevanten Unterports extrahiert und zum endgültigen Ergebnis hinzugefügt, falls diese keine Down-Connection aufweisen.

Von hier geht es in zwei verschiedene Richtungen weiter. Für die Ports des vollständigen Knotens und die neu dem Ergebnis hinzugefügten Ports wird wiederum geprüft, ob Verbindungen durch übergeordnete Ports realisiert sind. Daraus ergibt sich der folgende Algorithmus:

Initialisierung

- Ergebnisliste und Zwischenergebnisliste erzeugen
- ▶ Hauptalgorithmus(StartPort, Ergebnisliste, Zwischenergebnisliste)
- Erstellen des optimierten Knotens aus den Ports in der Ergebnisliste

Hauptalgorithmus(_Port_, _Ergebnisliste_ ,_Zwischenergenisliste_)

- Knoten = Berechne den vollständigen Knoten(_Port_)
- jeden Port des Knotens ohne Down-Connection in die Ergebnisliste aufnehmen, der nicht als besucht gekennzeichnet ist
- alle Ports des Knotens als besucht kennzeichnen
- ▶ Suche Verbindungen über übergeordnete Ports(Knoten, _Zwischenergebnisliste_)
- für alle Ports aus der Zwischenergebnisliste, die nicht als besucht markiert sind ▶ Hauptalgorithmus(Port, _Ergebnisliste_, _Zwischenergebnisliste_)
- Entfernen aller besucht- bzw. bearbeitet Markierungen

Suche Verbindungen über übergeordnete Ports(_Knoten_,_Zwischenergebnisliste_)

1. für jeden Port aus _Knoten_, der nicht als bearbeitet markiert ist und für den ein übergeordneter Port existiert, der ebenfalls nicht als bearbeitet markiert ist
   (a) den Port als bearbeitet markieren
   (b) Knoten = Berechne den vollständigen Knoten(übergeordneter Port)
      i. ▶ Suche Verbindungen über übergeordnete Ports(Knoten, _Zwischenergebnisliste_)
      ii. alle Ports von Knoten als bearbeitet markieren
      iii. alle betreffenden Unterports in Zwischenergebnisliste speichern

Berechne den vollständigen Knoten(_Port_)

1. Bestimme die Knoten für den aktuellen Port(Level- und Down-Connection)
2. Erstelle eine Liste aller an diesen Knoten angeschlossenen Ports
3. Jeder Port wird in den Knoten aufgenommen
4. Jeder Port ohne Down-Connection wird als erledigt markiert
5. Für jeden Port mit Down-Connection, der nicht als erledigt markiert ist
   - Port wird zum aktuellen Port
   - Port als erledigt markieren
   - ▶ Schritt1
6. Entfernen aller erledigt Markerungen

Wendet man diesen Algorithmus an um im Bild 4.17 für den Port AK4.P den optimierten Knoten zu ermitteln, so werden folgende Schritte ausgeführt :

1. Bestimmen des vollständigen Knoten (VK) für AK4.P
   $VK(AK4.P) = \{AK4.P, AK3.P, AK2.P1.B\}$
   $Ergebnisliste = \{AK4.P, AK3.P\}$

2. Der einzige Port aus $VK(AK4.P)$ der einen übergeordneten Port hat, ist AK2.P1.B, so daß für AK2.P1 der vollständige Knoten berechnet wird
   $VK(AK2.P) = \{AK1.P1, AK2.P1\}$
   Der vollständige Knoten enthält keine Ports mit übergeordneten Ports, so daß der Algorithmus für diese keine weiteren Werte liefert. Die relevanten Ports sind $AK1.P1.B$ und $AK2.P1.B$. $AK2.P1.B$ ist als besucht markiert und wird nicht weiter bearbeitet.

3. Auf $AK1.P1.B$ wird der Hauptalgorithmus angewendet. Es wird
   $VK(AK1.P1.B)$ berechnet.
   $VK(AK1.P1.B) = \{AK1.P1.B, AK5.P\}$
   Es existieren keine weiteren übergeordneten Ports, so daß die Suche über übergeordnete Ports beendet wird. $AK1.P1.B$ wurde bereits besucht und wird nicht weiter behandelt.

4. Auf $AK5.P$ hingegen wird erneut der Hauptalgorithmus angewendet, wobei für $AK5.P$ der vollständige Knoten ermittelt wird.
   $VK(AK5.P) = \{AK1.P1.B, AK5.P\}$
   $AK1.P1.B$ wird nicht weiter behandelt, da $AK1.P1.B$ als besucht und $AK1.P1$ bereits als bearbeitet markiert ist. $AK5.P$ wird in die Ergebnisliste aufgenommen
   $Ergebnisliste = \{AK4.P, AK3.P3, AK5.P\}$
   Da $AK5.P$ keine übergeordneten Ports hat, bricht der Algorithmus hier ab und liefert den optimierten Knoten (OK)
   $OK(AK4.P) = \{AK4.P, AK3.P3, AK5.P\}$

Mit der Definition des Algorithmus zur Bestimmung eines optimierten Knotens sind alle Voraussetzungen geschaffen, um das Routing bei hierarchisch implementierten Komponenten durchführen zu können. Wie die zur Implementierung einer aktiven Komponente erzeugten Komponenten in den Abbildungsprozeß aufgenommen werden, wird in Abschnitt 4.6 näher behandelt.

4.4 Erstellung der Implementierungbeschreibung der Systemeinheiten

Ziel dieser Verarbeitungsstufe ist es, eine in sich geschlossene Implementierungsbeschreibung für die jeweilige Systemeinheit zu erstellen, die folgende Punkte umfaßt:

- Beschreibung der implementierten Funktionalität
- Beschreibung der realisierten Verbindungen
- Beschreibung der Grundkonfiguration der Systemeinheit

Das anzuwendende Vorgehen ist je nach Art der Systemeinheit unterschiedlich. Es gibt Systemeinheiten, auf die alle oben genannten Punkte zutreffen, wie es bei FPGAs und CPUs der Fall ist. Andere Systemeinheiten hingegen unterstützen nur die Möglichkeit zum Verlegen von Verbindungen. Zu dieser Klasse gehören z.B. Crossbarswitches. Da der erst genannte Typ von Systemeinheiten alle Aufgaben umfaßt und die Behandlung der anderen Typen durch Auslassen der nicht benötigten Schritte möglich ist, sollen hier hauptsächlich auf Bausteine wie FPGAs eingegangen werden.

Die realisierte Funktionalität setzt sich aus den Implementierungen der einzelnen aktiven Komponenten zusammen, die auf die Systemeinheit abgebildet wurden. Ziel muß es hierbei sein, aus den vielen Einzelimplementierungen der aktiven Komponenten eine gemeinsame Beschreibung zu erstellen. Um dies zu erreichen, muß zuerst eine gemeinsame Beschreibungsbasis, in Form einer Implementierungssprache gewählt werden.

Jede auf der Systemeinheit abgebildete aktive Komponente wird aufgefordert, ihre Implementierung mittels dieser Implementierungssprache zu beschreiben. Bevor nun aber diese Teilimplementierungen zur Gesamtimplementierung zusammengesetzt werden können, ist es notwendig, die verwendeten symbolischen Verbindungsnamen aufzulösen. Für jede Verbindung muß ein Objekt in der Implementierungssprache erzeugt werden, das eine Verbindung repräsentiert. Jeder Systemport, der an dieser Verbindung teilnimmt, wird diesem Objekt zugewiesen. Für VHDL und CHDL sind diese Objekte Signale, bei C++ geeignete Objekte[4].

Es muß also bestimmt werden, welche Systemports durch die Systemeinheit miteinander verbunden sind. Die auf der Systemeinheit vorhandenen Routingkomponenten werden so gruppiert, daß alle Routingkomponenten einer Gruppe direkt oder indirekt miteinander verbunden sind. Für alle Routingkomponenten einer Gruppe bestimmt man dann die verwendeten Systemports. Die so erstellte Liste aller verbundenen Systemports wird in die Liste der verbundenen Ports konvertiert, in dem für jeden Systemport der zugeordnete Port bestimmt wird.

[4]Die einfachste Realisierung einer geeigneten Klasse würde eine read() und write() Funktion bereitstellen. Die Datenquelle würde über write() die Daten an das Objekt übergeben, die Datenquelle(n) mit read() auslesen.

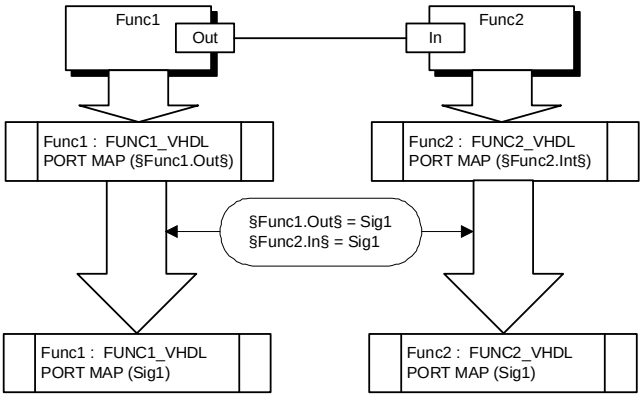

Bild 4.18. Zuweisung von Signalnamen

Nun kann für diese Verbindung ein eindeutiger Name gewählt werden, und alle symbolischen Bezeichner, die einen Port aus der Liste der verbundenen Ports darstellen, durch diesen Namen ersetzen (Bild 4.18).

Zusätzlich wird für jede Gruppe von Routingkomponenten eine Implementierung erzeugt, die dazu dient, das für diese Verbindung benötigte Signal mittels der gewählten Implementierungssprache zu erzeugen. Der zu verwendende Signaltyp ergibt sich aus dem Typ der verbundenen Ports.

Routingkomponenten, die physikalische Systemports enthalten, erzeugen für diese eine Implementierung, die Systemports in angemessener Weise konfiguriert und zusätzlich mit den entsprechenden Signalen verbindet. Danach können die einzelnen Implementierungsbeschreibungen der aktiven Komponenten und Verbindungen zusammengefaßt werden.

In den meisten Fällen ist es aber nicht ausreichend, die Implementierungen einfach aneinander zu ketten, sondern bestimmte Teile der jeweiligen Implementierung müssen an bestimmte Positionen in der Beschreibung der Gesamtimplementierung eingegliedert werden. So muß sichergestellt werden, daß Objekte, bevor sie referenziert werden, auch deklariert und initialisiert wurden. Ebenso müssen die Signale erst deklariert werden, bevor über sie Verbindungen hergestellt werden können.

Aus diesem Grund besteht die Möglichkeit, Teile der Implementierungsbeschreibung der Komponenten Segmenten innerhalb der Beschreibung der Gesamtimplementierung zuzuordnen.

Diese Zuordnung erfolgt dadurch, daß dem jeweiligen Codefragment ein Segmentselektor vorangestellt wird. Der Code bis zum nächsten Segmentselektor wird dann in das entsprechende Segment der Gesamtbeschreibung eingefügt. Bild 4.19 zeigt hierzu ein Beispiel, wobei als Implementierungssprache C++ verwendet wird.

Für einige Systemeinheiten müssen noch weitere Konfigurationsschritte implementiert werden. Diese Konfigurationsinformation kann in den meisten Fällen

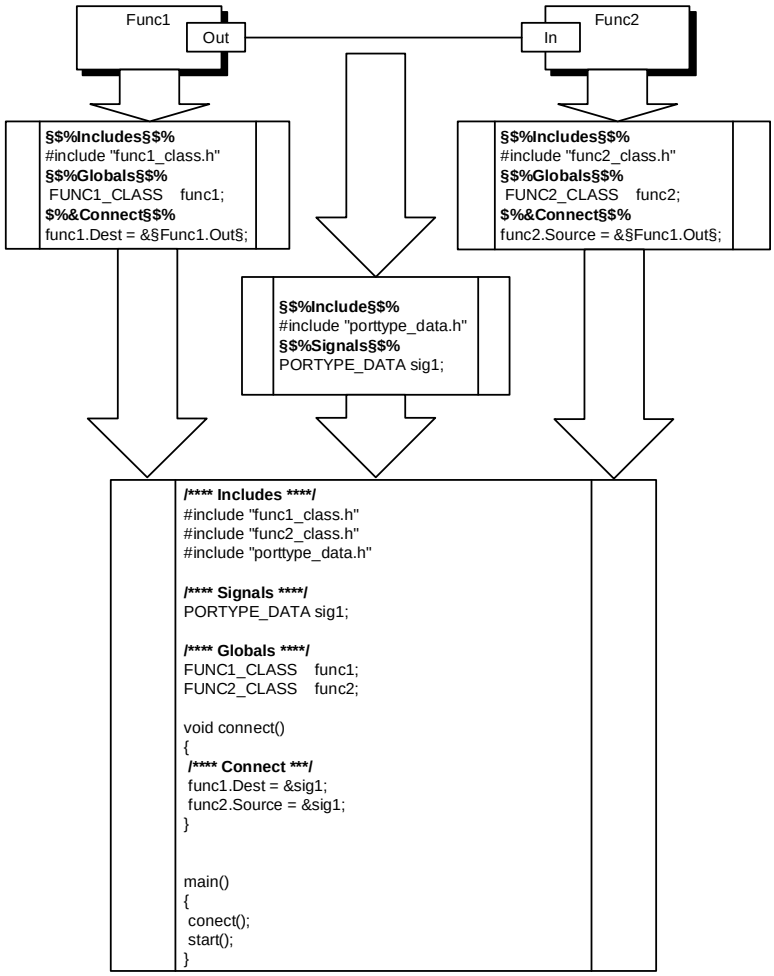

Bild 4.19. Erzeugung der Implementierung einer Systemeinheit aus den Implementierungen der aktiven Komponenten und der Implementierung der Verbindungen

in die Implementierungsbeschreibung aufgenommen werden. Diese Aufgabe wird durch die Systemeinheit selbst vorgenommen.

4.5 Initialisierung des Gesamtsystems

Eine weitere wichtige Aufgabe besteht darin, das gesamte Zielsystem zu initialisieren und die implementierte Anwendung zu starten. Das hierzu notwendige Vorgehen ist sehr stark vom jeweiligen System abhängig, da die meisten Systemeinheiten nicht in der Lage sind, ihre Gesamtinitialisierung selbst vorzunehmen, sondern dies mittels anderer Systemeinheiten erfolgen muß, die mit der entsprechenden Systemeinheit verbunden sind. Dies trifft z.B. auf FPGAs zu. Sie ver-

fügen über eine Konfigurationsschnittstelle, auf die die Initialisierungsdaten zu übertragen sind.

Um die Anwendung leicht bedienbar zu machen, ist es erstrebenswert, diese Initialisierung von einer zentralen Stelle aus durchführen. Um hierbei Systemeigenheiten berücksichtigen zu können, muß die Information über das Vorgehen bei der Initialisierung des Gesamtsystems Bestandteil der Systembeschreibung sein.

Die offensichtliche Möglichkeit ist hierbei, dezidierte aktive Komponenten zu erstellen, die diese Aufgabe erfüllen. Da sie aber nicht Teil der Lösungsbeschreibung sind, müssen sie schon im Rahmen der Systembeschreibungen auf die jeweiligen Systemeinheiten abgebildet und dann automatisch bei der Erstellung der Implementierung für die entsprechende Systemeinheit berücksichtigt werden.

Will man hier ein flexibleres Vorgehen erreichen, bieten sich die in Abschnitt 6.2 besprochenen virtuellen Systemeinheiten an.

Oft lassen sich nicht alle Systemeinheiten von einer zentralen Stelle aus direkt initialisieren und steuern, so daß man in diesen Fällen eine hierarchische Initialisierungsstruktur vorsehen muß.

Hierbei gibt es dann mehrere Systemeinheiten, die jeweils eine separate Gruppe von Systemeinheiten initialisieren, selbst aber wieder von einer zentralen Stelle aus angesteuert werden können.

4.6 Gesamtablauf

Nachdem bisher die zur Umsetzung notwendigen Einzelschritte besprochen wurden, sollen diese nun in den gesamten Umsetzungsprozeß (Bild 4.20) eingegliedert werden. Die im Bild grau hinterlegten Verarbeitungsschritte werden vom Programmiersystem automatisch ausgeführt und setzen sich aus den zuvor besprochenen Arbeitsschritten zusammen.

Aus der Darstellung ist auch zu erkennen, daß der Umsetzungsprozeß nicht vollständig geradlinig fortschreitet, sondern einige Verarbeitungsschritte auch gegebenenfalls iterativ durchlaufen werden. Dies betrifft zum einen das Routing, bei dem, wie in Abschnitt 4.2.2.3 bereits dargelegt, einzelne Verbindungen aufgelöst oder auch die Zuordnung der aktiven Komponenten zu den Systemeinheiten geändert werden müssen.

Auch die Möglichkeit, aktive Komponenten durch andere aktive Komponenten zu implementieren, erzwingt ein iteratives Vorgehen, da bei der Implementierung neue Komponenten erzeugt werden, die dann wieder plaziert, verbunden und implementiert werden müssen.

Aus diesem Grund ist es notwendig, daß nachdem der Implementierung der Komponenten geprüft wird, ob in diesem Schritt neue aktive Komponenten erzeugt wurden. Für sie müssen dann die erforderlichen Abbildungsschritte erneut durchgeführt werden. Dieser Prozeß bricht erst ab, wenn alle aktiven Komponenten auf dem Zielsystem implementiert wurden.

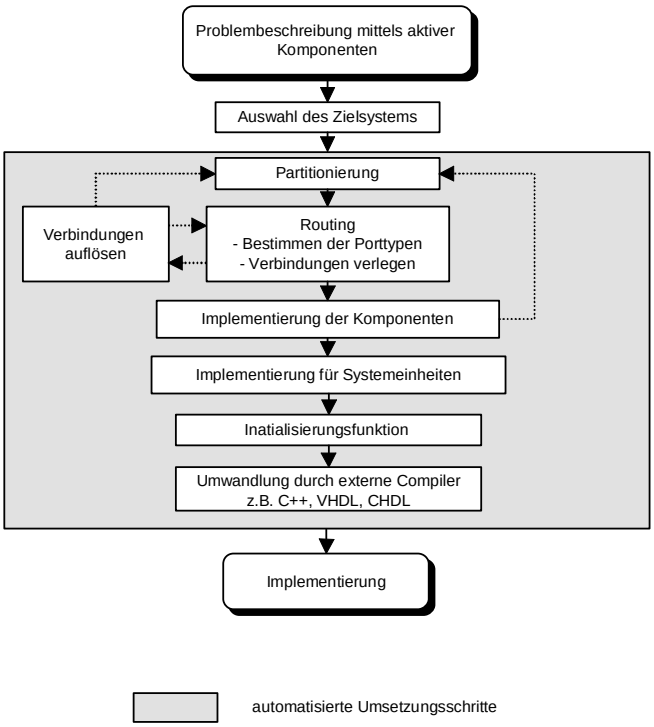

Bild 4.20. Gesamtablauf

Für einige Komponenten ist es notwendig, die Durchführung ihrer Implementierung zum frühestmöglichen Zeitpunkt zu verhindern und stattdessen einen späteren Zeitpunkt wählen zu können.

Unmittelbar einsichtig ist dies bei aktiven Komponenten, die der Realisierung virtueller Verbindungskanäle dienen (Abschnitt 4.2.2.1). Diese sind zwar schon beim ersten Durchlauf vorhanden, sollten aber erst implementiert werden, wenn alle anderen aktiven Komponenten erzeugt und verbunden wurden. Eine Implementierung zu einem früheren Zeitpunkt würde die Möglichkeit ausschließen, daß später erzeugte Komponenten Verbindungen über diese virtuellen Kanäle nutzen könnten.

Ähnlich verhält es sich mit Komponenten deren Implementierung durch Äbhängigkeiten zu anderen Komponenten beeinflußt werden, wie dies z.B. bei aktiven Komponenten, die die Funktion von Taktgeneratoren bereitstellen, der Fall ist (Bild 4.2). Hier können auf jeder Implementierungsebene neue Abhängigkeiten erzeugt werden, so daß eine Implementierung erst dann sinnvoll ist, wenn alle Abhängigkeiten bekannt sind. Dies wird durch den vorgestellten Gesamtablauf unterstützt.

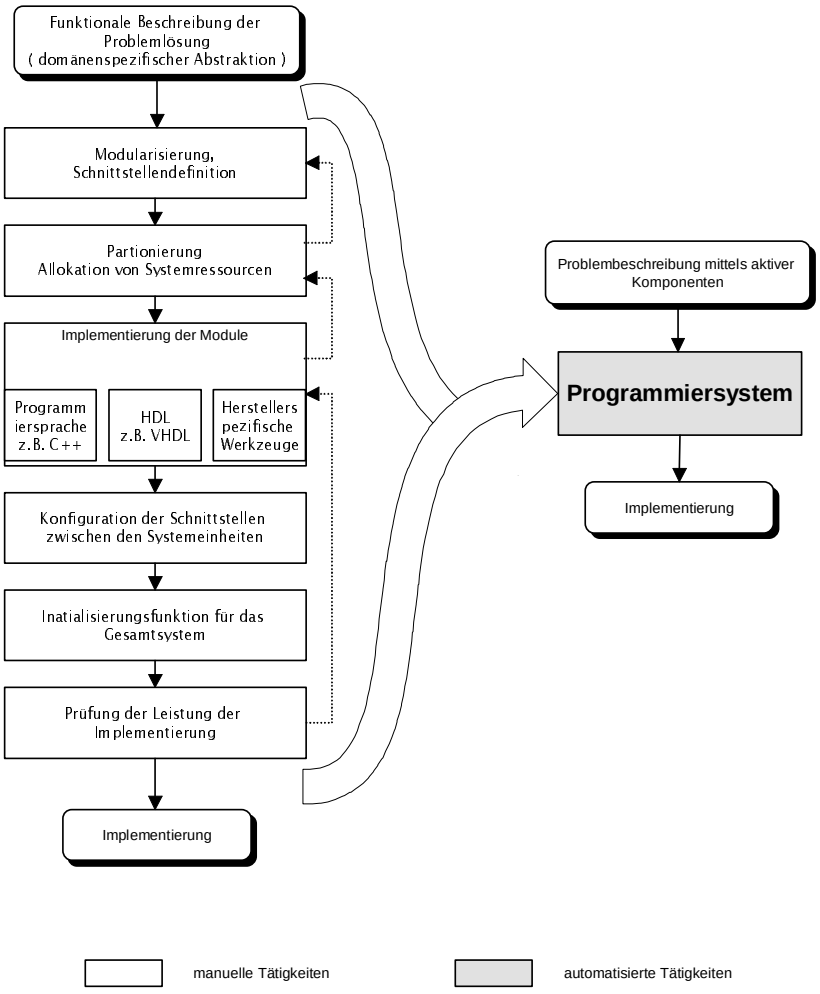

Bild 4.21. Gegenüberstellung des klassischen Entwurfszyklus (links) und des Entwurfszyklus mit aktiven Komponenten (rechts)

4.7 Zusammenfassung

Es konnte gezeigt werden, daß mittels aktiver Komponenten Lösungsbeschreibungen auf hoher Abstraktionsebene erstellt werden und durch den vorgestellten Umsetzungsprozeß auf verschieden heterogen Systemen implementiert werden können.

Hervorzuheben ist hierbei, daß Abstraktionsebenen bishin zur domänenspezifischen Ebene erreichbar sind und domänenspezifische Optimierungen unterstützt werden. Die erstellte Beschreibung ist weitgehend ohne Änderung auf verschiedene Zielsysteme übertragbar und gestattet die Programmierung des gesamten Zielsystems.

In Bild 4.21 wird der klassische Entwurfszyklus (siehe Abschnitt 2.2) zur Programmierung heterogener Systeme dem Entwurfszyklus mit aktiven Komponen-

ten gegenübergestellt. Hieraus läßt sich erkennen, daß sich bei der Variante mit aktiven Komponenten der Entwurfszyklus deutlich vereinfacht, was hauptsächlich dadurch erreicht wird, daß nur noch eine Beschreibungssprache von Nöten ist und die Systemabbildung vollständig automatisiert ist.

Durch die Einführung von Porttypen, die sowohl die zu transportierenden Daten, als auch die verwendeten Kommunikationsverfahren einschließen, sind weitreichende Schnittstellenüberprüfungen möglich, so daß weitgehend sichergestellt werden kann, daß die Schnittstellen der Komponenten gemäß ihrer Spezifikation angesteuert werden. Dies führt zu einer deutlichen Reduktion der potentiellen Fehlerquellen.

Während bisher die prinzipielle Anwendbarkeit dieses Ansatzes im Vordergrund stand, liegt in den folgenden Kapiteln der Schwerpunkt darauf, wie dieser Ansatz effizient umgesetzt werden kann, wobei besonders Fragen zur Portabilität und der effizienten Definition von aktiven Komponenten behandelt werden.

5

Vereinfachung des Entwurfs aktiver Komponenten

Um möglichst flexible und universell einsetzbare aktive Komponenten zu erhalten, müssen diese möglichst viele verschiedene Kommunikationsmechanismen und Implementierungsvarianten bereitstellen. Die Erstellung der hierzu notwendigen Komponentenimplementierungen und die Definition des Konfigurationswissens können schnell sehr aufwendig werden. Im vorliegenden Kapitel sollen einige Verfahren dargestellt werden, die diesen Punkten Rechnung tragen.

5.1 Vorabimplementierungen

Jede aktive Komponente muß in der Lage sein, die unterstützten Porttypen für alle von ihr bereitgestellten Ports bestimmen zu können. Dies wird in einigen Fällen dadurch erschwert, daß zur Implementierung einer Komponente eine Verkettung anderer Komponenten eingesetzt werden kann. Um nun die Frage über die unterstützten Porttypen beantworten zu können, müßten Informationen über die von den zur Implementierung verwendeten aktiven Komponenten ermittelt und je nach Art ihrer Verschaltung ausgewertet werden. Hierbei lassen sich zwei wesentliche Fälle unterscheiden (Bild 5.1).

Bild 5.1. Implementierungsvarianten von aktiven Komponenten

Bei der aktiven Komponente AK10 wird die gesamte Implementierung über untergeordnete aktive Komponenten realisiert. Bei Ak20 hingegen wird nur ein Teil über Unterkomponenten realisiert, der andere Teil ist variabel.

Um nun auch in diesen Fällen eine sinnvolle Bestimmung der Porttypen durchführen zu können, wird das Verfahren der Vorabimplementierung eingeführt. Hierbei können Komponenten Unterkomponenten hinzugefügt und die benötigten Verbindungen hergestellt werden. Die Unterkomponenten und Verbindungen werden nicht implementiert, weshalb diese Komponenten als virtuelle Komponenten und die Verbindungen als virtuelle Verbindungen bezeichnet werden.

Ihre eigentliche Funktion besteht darin, die Bestimmung der Porttypen zu erleichtern. Dies wird dadurch erreicht, daß bei der Bestimmung des Porttyps eines Ports, der über eine virtuelle Down-Connection verbunden ist, die Anfrage an die verbundenen untergeordneten Komponenten weitergeleitet wird. Virtuelle Level-Connections werden für die Porttypbestimmung wie gewöhnliche Verbindungen gehandhabt.

Bei der Porttypbestimmung von AK10.A wird also die Anfrage an Port AK2.A weitergeleitet. AK2 bestimmt dann die möglichen Typen für AK2.B über die Verbindung mit AK4. Entsprechendes gilt für die Ports AK10.B und A10.C. AK10 muß also keine weitere Funktionalität zur Porttypbestimmung bereitstellen.

Ähnlich verhält es sich bei AK20. Hier ist aber zu berücksichtigen, daß die Ports AK2.B und AK3.B nicht verbunden sind und somit deren Typen nicht unmittelbar bestimmt werden können. Tritt nun eine Anfrage für diese Ports auf, wird sie an AK20 weitergeleitet. Da AK20 über Kenntnisse der Eigenschaften der Implementierung des variablen Teils verfügt, kann es auch die von diesem Implementierungsteil unterstützten Porttypen für die Verbindungen zu den Ports AK2.B und AK3.B ermitteln.

AK20 macht also nur Angaben über Ports, die sie durch eine selbst generierte Implementierung ansteuert und muß nicht über die von den zur Implementierung verwendeten Komponenten unterstützen Porttypvarianten informiert sein. Dies hat den zusätzlichen Vorteil, daß bei einer Erweiterung der Flexibilität der für die Implementierung verwendeten Komponenten die implementierte Komponente davon profitiert, ohne daß an ihr Änderungen vorzunehmen sind. Um eine endgültige Implementierung zu erzeugen, muß die Komponente ihre virtuellen Bestandteile in reale Einheiten umwandeln.

5.2 Portadapter

In diesem Abschnitt soll es hauptsächlich darum gehen, die Handhabung verschiedener Kommunikationsverfahren zu vereinheitlichen und damit die Unterstützung dieser Kommunikationsvarianten durch die Komponenten zu vereinfachen. Hierzu werden zuerst die Eigenschaften der Porttypen klassifiziert und in verschiedene Gruppen eingeteilt. Ausgehend hiervon werden Methoden ermittelt, die es erlauben, Übergänge zwischen den verschiedenen Typen herzustellen.

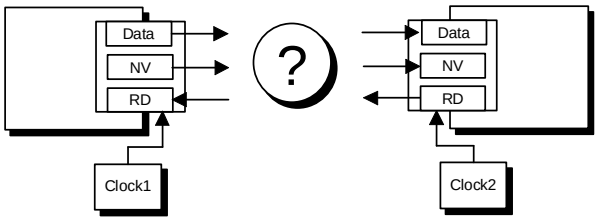

Bild 5.2. Kommunikation zwischen zwei Komponenten, deren Porttypen nur durch den verwendenten Takt unterscheiden

Die von einer aktiven Komponente unterstützbaren Porttypen lassen sich in verschiedene Gruppen einteilen:

- Kommunikationsverfahren mit gleichem (Basis-)Datentyp
  - Typen mit gleichem Übertragungsverfahren aber unterschiedlicher Realisierung
  - Typen mit gleichen Steuerungsmöglichkeiten
  - Typen mit verschiedenen Steuerungsmöglichkeiten
- Unterschiedliche Datentypen
  - gleicher Basistyp, gleicher Wertebereich, aber anderes Format
  - gleicher Basistyp, unterschiedlicher Wertebereich
  - unterschiedlicher Basistyp

Die erste Gruppe bilden Porttypen, bei denen übertragen Daten den gleichen Typ aufweisen und sich somit lediglich durch das verwendete Kommunikationsverfahren unterscheiden. Die zu betrachtenden Eigenschaften des Kommunikationsverfahrens sind hierbei das Übertragungsverhalten und die Steuermöglichkeiten, die durch das verwendete Protokoll bereitgestellt werden.

Das Übertragungsverhalten umfaßt die Bereiche des Kommunikationsverhaltens, die die Art und Weise der Datenübertragung festlegen. Hierzu gehören unter anderem Eigenschaften wie parallele, serielle, blockorientierte, taktsynchrone, asynchrone Übertragung. Die Steuerungsmöglichkeiten hingegen geben an, in welcher Weise auf den Ablauf der Datenübertragung Einfluß genommen werden kann. Hierzu gehören die Art und Weise der Qualifizierung der Daten, die Möglichkeit, die Datenübertragung anzuhalten, aber auch, ob ein Transfer von der Datenquelle oder Datensenke initiiert wird.

Eine Erste Klasse von Porttypen läßt sich dadurch abgrenzen, daß sie zwar über gleiche Übertragungsverfahren und Steuermöglichkeiten verfügen, sich aber dennoch in der konkreten Implementierung des Kommunikatinsverfahrens unterscheiden. In Bild 5.2 ist ein solcher Fall dargestellt.

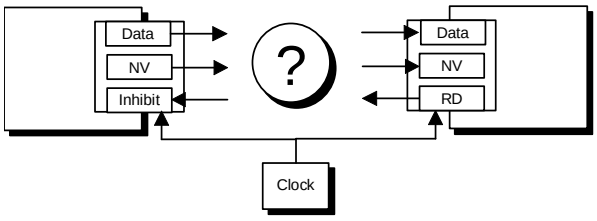

Bild 5.3. Kommunikation zwischen Ports gleicher Eigenschaften aber unterschiedlicher Implementierung

Es werden zwei Komponenten gezeigt, die jeweils über einen Port Daten austauschen sollen. Der Porttyp verwendet hierzu drei taktsynchrone Unterports. Data dient hierbei zur Übertragung eines vollständigen Datums, NV (NewValue) qualifiziert die Daten und RD (Ready) gibt an, daß weitere Daten übertragen werden dürfen. Während diese Merkmale beiden Kommunikationspartnern gemeinsam sind, unterscheiden sie sich in dem Merkmal des verwendeten Taktes.

Um nun eine Kommunikation zu ermöglichen, muß einer der Kommunikationsteilnehmer eine Synchronisation auf den Takt der Gegenstelle vornehmen. Wichtig ist hierbei, daß die Synchronisationsstufe (z.B. ein asynchroner FIFO-Speicher) keine Änderung an der Implementierung der jeweiligen Komponente hervorruft, sondern lediglich zur bestehenden Implementierung hinzugefügt werden kann. Dies führt zu keinen Leistungseinbusen.

Ein ähnliches Problem ist in Bild 5.3 dargestellt.Hier sollen wiederum zwei synchrone Ports miteinander kommunizieren. Während der eine Porttyp (im Bild rechts) dem zuvor dargestellten Typ entspricht, unterscheidet sich der zweite Port (im Bild links) von diesem Typ dadurch, daß er andere Unterports verwendet.

Die Unterports Date und NV erfüllen auch hier die Aufgabe, jeweils ein Datum zu transportieren, bzw. die Gültigkeit der Daten anzuzeigen. Aber anstatt mit RD anzuzeigen, daß neue Daten übertragen werden dürfen, wird das Inhibit-Signal verwendet. Durch die Aktivierung dieses Signales wird die Übertragung weiterer Daten unterbunden. Es verwendet somit gegenüber dem RD-Signal die entgegengesetzte Steuerlogik. Diese Typen unterscheiden sich also weder im Übertragungsverhalten noch in den Steuermöglichkeiten, sondern haben nur unterschiedliche Implementierungsvarianten gewählt.

Dementsprechend einfach ist es auch, eine Anpassung zwischen den beiden Typen herzustellen, da beim Verbinden der RD- und Inhibit-Signale lediglich ein Inverter eingefügt werden muß. Auch hier ist keine Anpassung der Komponentenimplementierung notwendig und auch die Performance der Implementierung bleibt unbeeinflußt.

Eine weitere Gruppe umfaßt Porttypen mit unterschiedlichem Übertragungsverhalten aber gleichen Steuerungsmöglichkeiten. Ein Beispiel hierzu ist in Bild 5.4 dargestellt.

Auch hier findet wieder ein Porttyp Verwendung, der die gleichen Unterports nutzt, wie sie schon aus dem ersten Beispiel bekannt sind. Obwohl die Steuer-

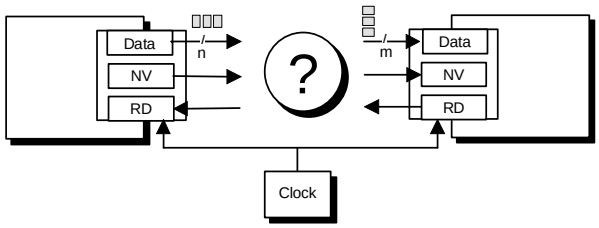

Bild 5.4. Porttypen mit gleichen Steuerungsmöglichkeiten aber unterschiedlichem Übertragungsverhalten

möglichkeiten beider Ports identisch sind, wird dennoch eine andere Art der Datenübertragung gewählt. Die Datenquelle überträgt ein Datum nicht am Stück, sondern zerlegt diese in maximal n-Bit große Datenpakete, die zeitlich nacheinander übertragen werden. Die Datensenke hingegen ist so ausgelegt, daß sie in jedem Übertragungsschritt ein vollständiges Datum entgegennimmt.

In diesem Fall ist eine Anpassung des Kommunikationsverhaltens dadurch möglich, daß die Datenpakete auf Seiten der Datensenke wieder zu einem vollständigen Datum zusammengesetzt werden, bevor sie zur Implementierung der Datensenke weitergeleitet werden. Während hierdurch zwar keine Einbusen im Bezug auf die Leistung der Implementierung entstehen, sind doch einige Punkte im Bezug auf die Implementierungskosten näher zu betrachten.

Bei entsprechend hoher Bit-Breite der zu übertragenden Daten, werden durch die Notwendige Adaption merkliche Implementierungskosten verursacht. Eine in Bezug auf die Implementierungskosten und Verarbeitungsleistung optimale Implementierung ist aber in diesem Fall über das Verfahren der Kommunikationsadaption meist nicht zu erreichen. Das beruht darauf, daß in vielen Fällen für den in der Datensenke verwendeten Algorithmus auch eine serielle Variante existiert, die es ermöglicht, die Datenpakete einzeln zu verarbeiten. Da diese Implementierungsvariante in den meisten Fällen deutlich geringere Implementierungskosten verursacht, als die parallele Variante,wäre dies für das gegebene Problem die ideale Lösung. Dies ließe sich aber nur durch eine Überarbeitung der Implementierung der Datensenke realisieren.

Komplizierter wird der Zusammenhang bei Porttypen, die sich zusätzlich noch in den bereitgestellten Steuermöglichkeiten unterscheiden. Ein Beispiel hierzu ist in Bild 5.5 dargestellt.

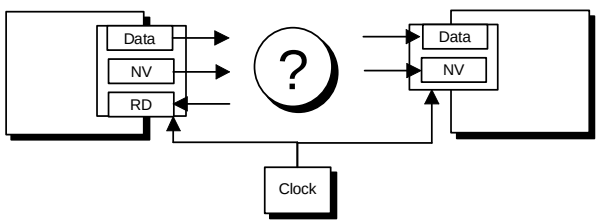

Bild 5.5. Porttypen mit unterschiedlichen Steuermöglichkeiten

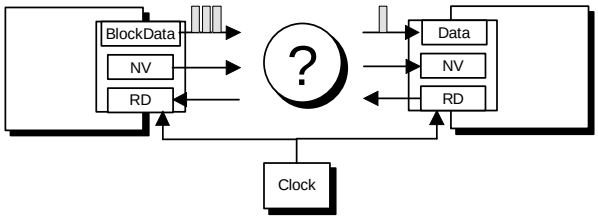

Bild 5.6. Übergang von blockorientierter zu zeichenorientierter Übertragung

Die beiden verwendeten Porttypen unterscheiden sich lediglich dadurch, daß die Datenquelle die Steuerung über das RD-Signal unterstützt, wohingegen die Datensenke diese Möglichkeit nicht vorsieht. In diesem Fall ist die anzuwendende Adaption leicht durchzuführen, da ein Verzicht der Datensenke auf die Möglichkeit, die Übertragung neuer Daten zu unterbinden, gleichbedeutend damit ist, auf der Seite der Datenquelle das RD-Signal konstant auf aktiven Pegel zu setzen. In einer Konstellation, in der die Verhältnisse zwischen Datensenke und Datenquelle genau umgekehrt sind, also die Datensenke das RD-Signal unterstützt, die Datenquelle hingen nicht, ist keine sinnvolle Adaption möglich.

Ähnliche Problemstellungen ergeben sich beim Übergang von blockorientierter Übertragung zu zeichenorientierter Übertragung (Bild 5.6).

Obwohl beide Porttypen die gleiche Steuersignale verwenden, beziehen sich diese doch auf verschiedene Datenquantitäten. Während die Datensenke in jedem Übertragunsschritt ein Datum entgegennimmt, versendet die Datenquelle in jedem Überdatenschritt einen Block von n Daten. Während die Datensenke durch ein aktives RD-Signal ein Datum anfordert, wird die Aktivierung des RD-Signal von der Datenquelle als Aufforderung dazu verstanden, einen vollständigen Datenblock zu versenden.

Um hier eine Adaption durchführen zu können, muß ein Zwischenspeicher (FIFO-Speicher) eingerichtet werden, dessen Größe ausreichend ist, einen ganzen Datenblock aufzunehmen. Dieser Zwischenspeicher gibt dann Datum für Datum an die Datensenke weiter und fordert, falls der Speicher wieder ausreichend geleert ist, einen neuen Datenblock bei der Datenquelle an.

Ein weiteres Merkmal, in dem sich Porttypen unterscheiden können, ist der Typ der transportierten Daten. Die Datentypen lassen sich anhand der Kriterien Basistyp, Wertebereich und Format klassifizieren.

Der Basistyp gibt an, welche Form von Information durch ein Datum dargestellt werden kann. Dies können z.B. Fließkommazahlen, Integerwerte, aber auch Zeichenfolgen sein. Der Wertebereich gibt an, welche Werte von einem Datum angenommen werden können. Das Format wiederum legt fest, in welcher Weise die Information kodiert ist. Zahlen können z.B. im BCD-Format, Zweierkomplement und Zeichen im ASCII- oder UNICODE-Format dargestellt werden.

Daten, die den gleichen Basistyp, den gleichen Wertebereich aber ein unterschiedliches Format aufweisen, können verlustfrei ineinander überführt werden, so daß hier eine einfache Anpassung der Porttypen möglich ist, in dem einfach in

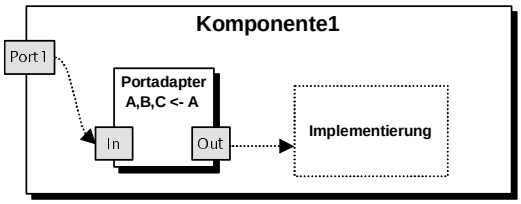

Bild 5.7. Komponente mit Portadapter

den entsprechenden Kommunikationspfad der geeignete Codekonverter eingebaut wird.

Gleiches gilt für die Fälle, bei denen zusätzlich der Wertebereich variiert, wenn der Wertebereich der Datensenke größer als der der Datensenke ist und somit bei der Konvertierung keine Information verlorengeht. Der umgekehrte Fall, bei der eine Konvertierung von einem größeren in einen kleineren Wertebereich notwendig ist, kann auch realisiert werden. Hierbei gehen aber Informationen verloren.

Aus den obigen Ausführungen ist ersichtlich, daß es in vielen Fällen möglich ist, durch Adaptionsmechanismen die von einer Komponente unterstützten Porttypen deutlich zu erweitern, ohne dabei in die Implementierung der Komponente selbst eingreifen zu müssen.

Dies legt die Definition von Portadaptern nahe, deren Aufgabe darin besteht, einen gegebenen Porttyp an verschiedene andere Porttypen anzupassen. Unschön hieran ist, daß diese Adapter manuell einzufügen sind. Wünschenswert wäre es aber, wenn für jeden Port die relevanten Portadapter je nach Bedarf in einer Art und Weise automatisch eingefügt werden würden, die für die Komponente vollständig transparent wäre.

Um dies zu erreichen, werden die verschiedenen Porttypen in eine Hierarchie eingegliedert. Porttypen, zwischen denen eine Adaption möglich ist, erhalten hierbei einen gemeinsamen Basisporttyp. Jedem Porttyp kann dann ein geeigneter Portadapter zugeordnet werden, der die geforderten Anpassungen durchführt. Wird jetzt einem Port ein Porttyp zugewiesen, wird der entsprechende Portadapter automatisch als Unterobjekt der aktiven Komponente eingeordnet, zu der der Port gehört[1]. In Bild 5.7 ist die sich daraus ergebende Situation dargestellt.

Wenn die vorhandene Implementierung von Komponnete1 nur den Porttyp A unterstützt, wird dies durch den eingefügten Portadapter dahingehend ausgedehnt, daß zusätzlich die Typen B und C bereitgestellt werden und hierdurch die Flexibilität der Komponente deutlich erweitert wird.

Störend ist aber dennoch, daß die Integration des Portadapters nicht vollständig transparent ist. Im Gegensatz zu der Variante ohne Portadapter, muß nun die Komponente nicht die Typen von Port1 und dessen Unterports auf die ent-

[1] Die aktive Komponente kann dies allerdings auch unterbinden. Dies ist notwendig, da die Portadapter auch als aktive Komponenten implementiert werden und zumindest einen Port besitzen, der den Porttyp trägt, für den der Portadapter zuständig ist. Würde man an dieser Stelle die automatische Erzeugung des Portadapters nicht unterbinden, würden endlos neue Portadapter eingefügt.

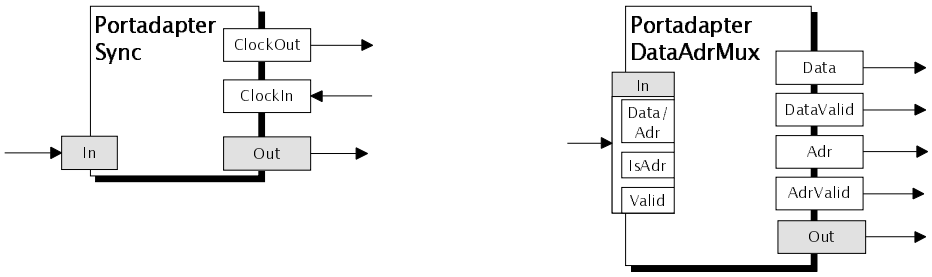

Bild 5.8. Vereinfachung der Handhabung von Kommunikationsmechanismen durch portadapter

sprechende Anfrage hin angeben, sondern muß dies für den Port Komponente1.Portadapter.Out tun.

Um diesen Unterschied aufheben zu können, wird die Benennung von Ports dahingehend geändert, daß beim Ansprechen eines Ports durch dessen vollständigen Namen zusätzlich ".Port" anzufügen ist. Aus Komponente1.Port1 wird dementsprechend Komponente1.Port1.Port. Für einen Port ohne Portadapter entspricht der symbolische Unterport "Port" dem Port selbst. Für einen Port hingegen der über einen Portadapter verfügt, erreicht man über diesen Bezeichner den Ausgangsport des Portadapters, so daß Komponente1.Port1.Port mit dem Port Komponente1.Portadapter.Out gleichzusetzen ist. Durch diese Konvention können nun Ports mit Portadapter auf die gleiche Weise gehandhabt werden wie Ports ohne Portadapter.

Zusätzlich kann ein Portadapter noch dazu dienen, die Ansteuerbarkeit der Kommunikationsmechanismen zu vereinfachen. In Bild 5.8 sind hierzu zwei Beispiele aufgeführt.Das erste Beispiel (im Bild links) zeigt einen Portadapter, der taktsynchrone Porttypen adaptiert. Zu der bisher bekannten Funktionalität der Porttypanpassung stellt dieser Adapter zusätzliche Ports bereit. Hierbei handelt es sich um die Ports ClockIn und ClockOut. Der Entwickler kann über diese Ports nun leicht den Takt, der zur Kommunikation verwendet wird, an ClockOut abgreifen bzw. über ClockIn angeben, welchen Takt er selbst zur Kommunikation verwenden will.

Eine Beschreibung dieses Sachverhaltes wäre zwar auch ohne die Unterstützung des Portadapters möglich, dennoch vereinfacht dieser Ansatz die Anwendbarkeit des bereitgestellten Kommunikationsmechanismus, da die Implementierungsdetails vom Entwickler verborgen bleiben. So ist es für die Bedienung dieses Ports unerheblich, ob das Taktsignal als eigenes Signal zu den Unterports des Porttyps gehört, oder ob es lediglich in Form eines Attributes Bestandteil des Porttyps ist.

Das zweite Beispiel (im Bild rechts) benutzt einen Porttyp, der auf einem Unterport Daten- und Adressinformation in zeitlicher Abfolge überträgt, wobei ein Unterport die Gültigkeit der Daten anzeigt (Valid) und ein anderer (IsAdr) zur Unterscheidung zwischen Daten- bzw. Adressinformation herangezogen wird. Um diesen Kommunikationsmechanismus einfacher anwenden zu können, ist es

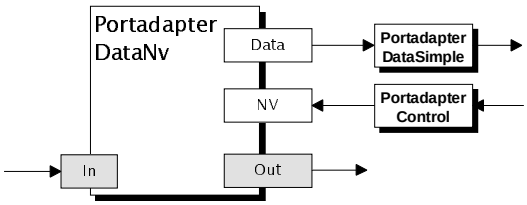

Bild 5.9. Hierarchische Verwendung von Portadaptern

sinnvoll, die Daten- und Adressinformation über getrennte Ports bereitzustellen, wobei die jeweiligen Daten auch durch unterschiedliche Ports qualifiziert werden.

Dies wird durch die Portpaare Data/DataValid und Adr/AdrValid realisiert. Wird der Portadapter an einem Ausgansport betrieben, muß er die Aufspaltung der Daten- und Adressinformation rückgängig machen und die Steuersignale entsprechend anpassen.

Um auch hier den Einsatz der Portadapter transparenter zu gestalten, können die entkoppelten Ports und die dazugehörigen Steuerports als symbolische Unterports in den mit dem Portadapter verbundenen Port eingetragen werden. Somit wären die Daten, die über einen Port mit Namen P übertragen werden, unter P.Data.Port zu erreichen.

Der Entwickler kann also auf einer abstrahierten Beschreibung des Kommunikationsmechanismus aufsetzen und ist somit weitgehend von der eigentlichen Implementierung der Kommunikation unabhängig, wodurch sich ganze Klassen verschiedener Kommunikationsmechanismen in gleicher Weise bedienen lassen und so aktive Komponenten ohne zusätzlichen Aufwand eine wesentlich größere Zahl von Kommunikationsmechanismen unterstützen können.

Eine weitere praktische Eigenschaft bei der Verwendung von Portadaptern, die zur einfacheren Bedienung weitere Ports definieren, besteht darin, daß hierdurch auch Portadapter die Funktionalität andere Portadapter nutzen können. Dies wird in Bild 5.9 anhand eines Porttyps (DataNv) gezeigt, der eine Übertragung von Daten inklusive eines Gültigkeitssignals (NV) ermöglicht.

Während der Portadapter selbst die Synchronisation zwischen Kommunikationspartnern durchführt, falls für die Übertragung verschiedene Taktsignale verwendet werden, kann er die Anpassung des Datenformates einem hierfür geeigneten Portadapter übertragen, in dem er für den Port Data einen entsprechenden Basisporttyp vorgibt.

Der Basisporttyp ist entweder ein abstrakter oder unterdefinierter Porttyp. Er legt somit keine konkrete Implementierung fest, sondern definiert nur grundlegende Eigenschaften der möglichen Porttypen.

Im vorliegenden Beispiel erreicht man dies dadurch, daß der Porttyp DataSimpel verwendet wird, der die Übertragung von Daten mit beliebiger Bitbreite, in verschiedenen Datenformaten ermöglicht. Der entsprechende Portadapter kann dann z.B. die Umwandlung zwischen verschiedenen Zahlendarstellungen durch-

führen. Bei Steuersignalen vom Typ Control, wie z.B. dem Port NV, kann diese Anpassung zwischen Eins-Aktiven und Null-Aktiven Porttypen erfolgen.

Die Möglichkeit, Portadapter hierarchisch aufzubauen, erhöht die Wiederverwendbarkeit und vereinfacht so den Entwurf eines neuen Portadapters deutlich.

6
Verbesserung der Portierbarkeit

Bei der Abbildung von Problemlösungen auf verschiedene Zielsysteme kann es leicht zu Engpässen bei den verfügbaren Systemeinheiten kommen. Problematisch sind hierbei besonders Systemeinheiten, die von sich aus nur in der Lage sind, eine Komponente aufzunehmen. Diese Systemeinheiten werden im folgenden als Systemressource bezeichnet.

Um diesem Problem besser begegnen zu können, werden Verfahren eingeführt, die die gemeinsame Nutzung von Systemressourcen durch mehrere Komponenten erlauben, auch wenn dies von den Systemressourcen selbst nicht unterstützt wird (Abschnitt 6.1).

Um die Portierbarkeit zu ermöglichen, wurden bereits Verfahren zur zielplattformunabhängigen Anforderung von Systemeinheiten mittels aktiver Komponenten bereitgestellt. Allerdings stellen nicht alle Zielsysteme die gleichen Typen von Systemeinheiten zur Verfügung, so daß die Anforderung von bestimmten Systemressourcen auf gewissen Zielsystemen nicht möglich ist. Um diese Einschränkung aufzuheben oder zumindest zu verringern, werden virtuelle Systemeinheiten eingeführt (Abschnitt 6.2).

6.1 Resourcesharing

6.1.1 Teilbarkeit von Ressourcen

Die Möglichkeiten der gemeinsamen Nutzung von Systemressourcen durch verschiedene Teile der Lösungsimplementierung sind im großen Maße vom Typ der Systemressourcen abhängig.

Die erste Klasse sind die real teilbaren Systemressourcen, bei denen eine, in sich geschlossene Systemressource, in der Lage ist, sich in verschiedene logische

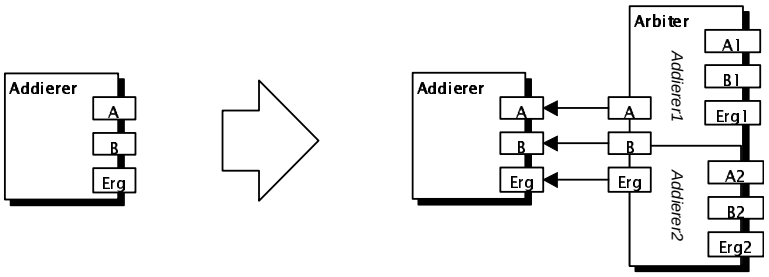

Bild 6.1. Teilen einer Ressource durch zeitliche Schachtelung der Zugriffe

Teile zu trennen. Ein Repräsentant dieser Klasse ist z.B. ein Speicherbaustein. Dieser ist in der Lage, Daten in einem definierten Umfang zu speichern. Er läßt sich aber durchaus in mehrere logische Teile trennen, in dem der Speicher in verschiedene Adressbereiche aufgeteilt wird. Jeder dieser Teile stellt die gleiche grundlegende Funktion wie die gesamte Ressource bereit und jede Teilressource kann ihre Funktion unabhängig von den anderen Teilressourcen erfüllen[1]. Gleiches gilt z.B. für einen DMA-Controller, der verschiedene unabhängige DMA-Kanäle bereitstellt.

Eine zweite Klasse von Ressourcen lassen sich nicht in dieser Form aufteilen. Bei ihnen kann aber in vielen Fällen die Verfügbarkeit der Funktionalität zeitlich zwischen den verschiedenen potentiellen Nutzern der Ressource aufgeteilt werden, wie dies in Bild 6.1 dargestellt ist.

Um die Zugriffe der verschiedenen Nutzer zu Trennen, wird der zu teilenden Ressource ein Arbiter vorgeschaltet. Der Arbiter stellt jedem Teilnehmer einen separaten Zugang zur gewünschten Funktionalität zur Verfügung und leitet diese Anfragen an die Ressource weiter, die dann die eigentliche Verarbeitung ausführt. Gleichzeitig auftretende Anfragen müssen zeitlich geschachtelt werden, wobei es vom gewählten Arbitrierungsalgorithmus anhängig ist, in welcher Reihenfolge die Zugriffe bearbeitet werden. Dieses Verfahren ist aber nur auf Ressourcen anwendbar, deren Ausganszustand nach Ausführung der Funktion wiederhergestellt wird.

Oft treten beide Arten der Teilbarkeit gemeinsam auf. Betrachtet man z.B. den obengenannten Fall eines Speicherbausteines noch einmal genauer, stellt man fest, daß zwar der Speicher als solches real teilbar, der Zugang zu den Speicherfunktion aber nur zeitlich getrennt werden kann, da nur eine Schnittstelle für den Zugriff bereitsteht.

Um solche Bausteine teilen zu können, müssen sowohl der Adressraum geteilt, als auch der Zugriff auf die Schnittstelle zeitlich gemultiplext werden (Bild 6.2).

[1] Hier wird vorerst nur die Funktion des Speicherbausteins betrachtet, Daten zu Speichern. Um diese Funktion auslösen zu können, müssen natürlich die entsprechenden Bausteinschnittstellen bedient werden. Wie diese zu teilen sind, wird noch zu erläutern sein.

Um auch auf diesen nicht vollständig real teilbaren Systemressourcen mehrere aktive Komponenten implementieren zu können, müssen die Beschreibungsmöglichkeiten der aktiven Komponenten erweitert werden.

6.1.2 Accessports

Ein wichtiger Punkt bei der Teilung nicht vollständig real teilbarer Systemeinheiten besteht darin, mehreren Teilnehmern ein eigenes Interface für die Ansteuerung der gewünschten Funktion bereitzustellen.

Um dies zu unterstützen, werden aktive Komponenten um Accessports erweitert. Ein Accessport umfaßt hierbei alle für die Ansteuerung einer bestimmten Funktion notwendigen Ports. Um einen neuen Accessport zu erhalten, muß dieser bei der aktiven Komponente unter Angabe der gewünschten Funktion, die über diesen Accessport erreichbar sein soll, angefordert werden. Die Komponente muß hierzu überprüfen, ob sie über eine Implementierung verfügt, die die gewünschte Anzahl von Accessports unterstützt. Ist eine geeignete Implementierung nicht vorhanden, kann kein neuer Accessport erzeugt werden.

Aufgabe der aktiven Komponente ist es nun, eine Implementierung zu erzeugen, die für die zeitliche Trennung der nicht real teilbaren Bestandteile und für die Aufteilung der real teilbaren Bestandteile der Systemressource sorgt. Hierzu können jedem Accessport Attribute zugeordnet werden, die angeben, welcher Bereich des real teilbaren Ressourcenteil über diesen Accessport anzusteuern und wie der Accessport bei der Arbitrierung zu handhaben ist. Über Attribute die der aktiven Komponente zugeordnet sind, kann die Auswahl des Arbitrierugsalgorithmus beeinflußt werden.

Mit diesen Beschreibungsmethoden kann der Entwickler im Programm explizit die Möglichkeit des Resourcesharings vorsehen. Anwendung wird dies aber hauptsächlich in solchen Fällen finden, in denen der Programmierer den Sachverhalt beschreiben will, daß bei der gegebenen Programmlogik die gemeinsame Nutzung der Ressource vorteilhaft ist. Ein Beispiel hierfür ist in Bild 6.3 dargestellt.

Hier soll jeder der zwei Teilnehmer auf einen eigenen Speicherbereich zugreifen können. Da die Teilnehmer aber so verschaltet sind, daß jeder Teilnehmer nur

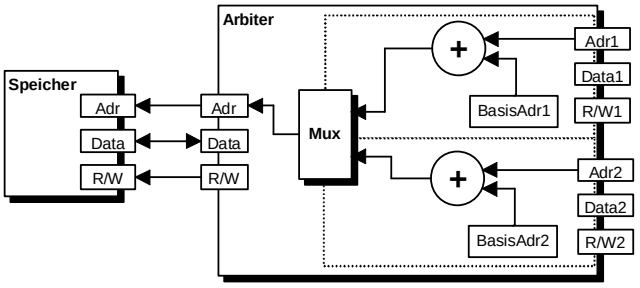

Bild 6.2. Teilen eines Speicherbausteins

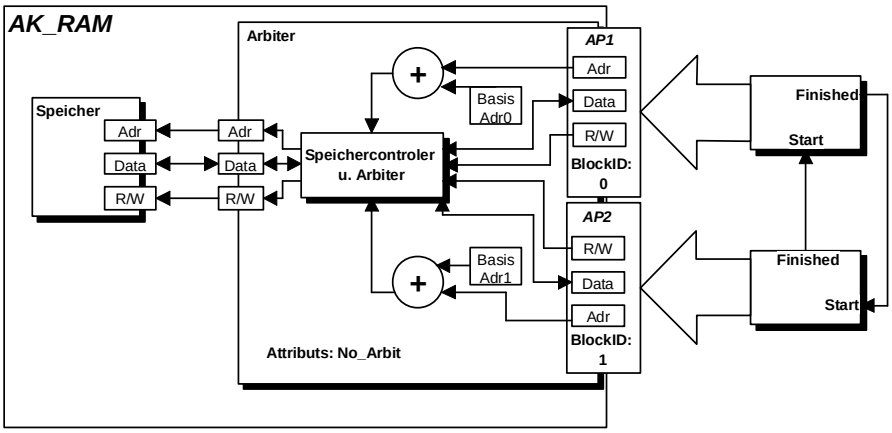

Bild 6.3. Expliziete Verwendung des Resourcesharings

dann auf den Speicher zugreifen kann, wen der andere einen Zugriff abgeschlossen hat, treten keine gemeinsamen Zugriffe auf.

Man kann nun zur Realisierung der Speicherfunktion eine Komponente verwenden (AK_RAM), für sie zwei Accessports erzeugen und jeden der Teilnehmer über einen Accessport auf die Speicherfunktion zugreifen lassen. Um den Speicherbereich zu definieren, auf den ein Accessport zugreift, wird jedem Accessport eine entsprechende BlockID als Attribut zugeordnet. Die BlockID referenziert eine Blockdeskriptor, der genauere Informationen über die Eigenschaften des angeforderten Speicherbereiches enthält. Da gemeinsame Zugriffe durch beide Teilnehmer ausgeschlossen sind, wird für AK_RAM ein Attribut (NO_ARBIT) erzeugt, das diesen Sachverhalt anzeigt. Die Komponente AK_RAM kann nun ihre Implementierung dadurch erzeugen, daß sie zwei Unterkomponenten anlegt. Die eine dient der Reservierung eines Speicherbausteins, der die zu speichernden Daten aufnimmt, die andere realisiert den Arbiter, wobei sich dieser im gegeben Fall darauf beschränken kann, die Signale zwischen den Teilnehmern und dem Speicherbaustein zu multiplexen.

In vielen anderen Fällen wird man aber die Verwendung unabhängiger Komponenten vorziehen, da hiermit oft eine weitaus intuitivere Beschreibung der Sachverhalte möglich ist. Aus diesem Grund müssen auch Fälle unterstützt werden, bei denen mehr Ressourcen in Form von Komponenten angefordert werden, als auf dem Zielsystem real vorhanden sind.

6.1.3 Automatisches Resourcesharing

Der grundlegende Ansatz beim automatischen Resourcesharing besteht darin, mehrere aktive Komponenten des gleichen Typs zu einer einzigen aktiven Komponente zu verschmelzen. Bei diesem Vorgehen dürfen aber weder die eigentliche Lösungsbeschreibung, noch deren Funktionalität geändert werden.

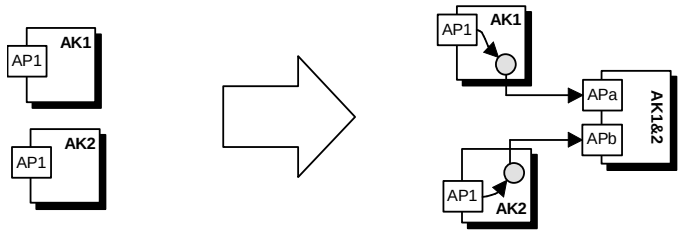

Bild 6.4. Verschmelzen von Komponenten

Um dies zu ereichen, wird eine neue aktive Komponente des gleichen Typs erzeugt. Für jede Funktion werden Accessports in dem Umfang angelegt, wie sie der Summe der von den zu verschmelzenden Komponenten benutzten Accessports für die jeweilige Funktion entsprechen. Zusätzlich wird die neue erzeugte Komponente für alle zu verschmolzenen Komponenten als Unterkomponente eingetragen und somit als Implementierung der jeweiligen Komponente behandelt, was dadurch vervollständigt wird, daß die Accessports der zu verschmelzenden Komponente über eine Down-Connection mit dem entsprechenden Accessport der verschmolzenen Komponente verbunden werden (Bild 6.4).Bei Funktionen, für die eine nicht ausreichende Zahl von Accessports bereitgestellt werden kann, ist dieses Verfahren allerdings nicht ausreichend.

Geht man aber zurück zum Ausgangsproblem, nicht vollständig teilbare Ressourcen teilbar zu gestalten, wird man feststellen, daß diese Ressourcen nicht direkt teilbar sind, aber oft durch Komponenten angesteuert werden, die mehrere Accessports und damit auch das Resourcesharing unterstützen.

In Bild 6.3 wurde eine Speicherkomponente dargestellt, die sich durch die Verwendung zweier weiterer Komponenten implementiert. Die eine Komponente repräsentiert den nicht vollständig teilbaren Speicherbaustein, wohingegen die zweite Komponente, die den Zugriff auf den Speicherbaustein steuert, mehrere Accessports für den Zugriff auf die Speicherfunktionen bereit stellt. Wie bei derartigen Konstellation ein Resourcesharing möglich ist, zeigt Bild 6.5.

Hier wird die Speichercontroller-Komponente wie bisher, durch Zuammenfassen der Accessports in eine einzelne Komponente überführt. Bei der Verschmelzung der Speicherkomponenten wurde jedoch so vorgegangen, daß nicht die Anzahl der Accessports, sondern der erforderliche Ressourcenbedarf angepaßt wurde. Diese Anordnung kann nun auf ein System mit nur einem Speicherbaustein abgebildet werden. Es bleibt nun die Frage zu klären, in welchen Fällen dieses Verfahren anwendbar ist und wie bestimmt werden soll, wie die Zusammenfassung der jeweiligen Komponente erfolgen kann.

Um hier zu anwendbaren Regeln zu gelangen, müssen zwei verschiedene Arten von Ports unterschieden werden. Die bisher am häufigsten verwendeten Ports dienen dazu, den Zugriff auf die Funktionen der Komponente zu ermöglichen, weshalb sie auch bei der Erzeugung des entsprechenden Accessports dupliziert werden.

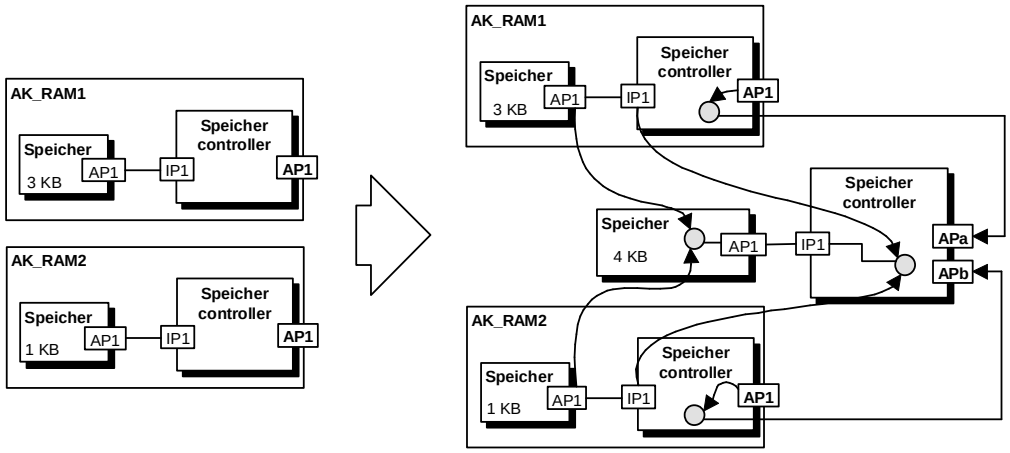

Bild 6.5. Automatisches Resourcesharing mittels Implementierungsports

Im Gegensatz zu diesen Funktionsports gibt es noch die Gruppe der Implementierungsports. Sie dienen einer Komponente dazu, andere Komponenten anzusprechen, um mit ihrer Hilfe, die eigene Funktionalität bereitstellen zu können. Ihre Anzahl ist für einen gegebenen Typ von Komponente unveränderlich und damit auch unabhängig von der Anzahl der Accessports. Ports, die über Implementierungsports verbunden sind, müssen beim Verschmelzen nicht vervielfältigt werden, da die verschmolzene Komponente vom gleichen Typ wie die zu verschmelzenden Komponenten ist und somit auch die gleiche Anzahl von Implementierungsports aufweist.

Es kann aber notwendig sein, den Umfang der Implementierungsressourcen anzupassen, die für die Komponente, die mit ihrem Port mit dem Implementierungsport verbunden ist, zur Realisierung der gewünschten Funktion benötigt wird.

Bevor diese Verschmelzungsvorschriften angewendet werden können, muß geprüft werden, ob eine Verschmelzung überhaupt möglich ist. Über den folgenden Algorithmus kann dieser Sachverhalt ermittelt werden:

1. Sind die zu verschmelzenden Komponenten vom gleichen Typ ?
   Nein ▶ *Verschmelzen nicht möglich*

2. Kann die verschmolzene Komponente genügend Accessports erzeugen ?
   Nein ▶ *Verschmelzen nicht möglich*

3. Sind ausreichend Implementierungsressourcen (z.B. Speicherbereich oder Logikzellen) für die verschmolzene Komponente verfügbar?
   Nein ▶ *Verschmelzen nicht möglich*

4. Für Komponenten, die über einen Implementierungsport mit der aktuellen Komponente verbunden sind
   ▶ Schritt *1*

Dieses Vorgehen berücksichtigt, daß die Verschmelzung von Komponenten nur dann erfolgen kann, wenn dies durch alle mit ihr direkt oder indirekt verbunden relevanten Komponenten unterstützt wird.

Mit dem hier vorgestellten Verfahren ist es nun möglich, bei zu geringer Anzahl der Systemressourcen, die von einer Lösungsbeschreibung angeforderten Systemressourcen in vielen Fällen soweit zu reduzieren, daß eine Implementierung auf dem gewünschten Zielsystem möglich wird. Durch dieses Vorgehen kann der Entwickler ermutigt werden, bei der Lösungsbeschreibung den größtmöglichen Parallelisierungsgrad bei der Erstellung der Lösungsbeschreibung zu verwenden, da dies bei Systemen mit ausreichenden Systemressourcen zur bestmöglichen Performance führt, für kleinere Systeme aber immer noch eine sinnvolle Realisierung erlaubt.

Um die Möglichkeit des Resourcesharings automatisch durchführen zu können, muß sie in den Umsetzungsprozeß eingegliedert werden. Sie kann sinnvoll bei der Plazierung der Komponenten angewandt werden und hier die möglichen Plazierungsvarianten erweitern. Kann eine Komponente nicht plaziert werden, weil die relevanten Systemressourcen jeweils nur eine Komponente aufnehmen können und keine unbenutzte Systemressource zu Verfügung steht, oder eine geeignete Systemressource bereits durch eine andere Komponente belegt ist, kann der Versuch unternommen werden, eine Verschmelzung der Komponenten vorzunehmen und so eine geeignete Plazierung der Komponente zu erreichen.

6.2 Virtuelle Systemeinheiten

In einigen Fällen können Zielsysteme die von einer Lösungsbeschreibung angeforderten Typen von Systemeinheiten nicht bereitstellen, da sie nicht in expliziter Form auf dem Zielsystem vorhanden sind. Werden diese Arten von Systemeinheiten von vielen Programmen benötigt, schränkt dies die Anwendbarkeit des Zielsystems deutlich ein.

Ein typischer Fall hierfür sind Systemeinheiten, die es ermöglichen, das Gesamtsystem zurückzusetzen (Global Reset) oder für das Gesamtsystem die Ausführung der gesamten Applikation zu starten (Global Start). Diese Funktionalität wird bei der Initialisierung des Gesamtsystems verwendet und kann von einer Komponente dazu benutzt werden, einen definierten Anfangszustand herzustellen bzw. die Ausführung der Applikation zu starten. Diese Funktionalität könnte zwar fest in die Lösungsbeschreibung kodiert werden, müßte aber dann feste Annahmen über die Struktur des Zielsystems machen. Diese Beschreibung ist nur sehr schlecht portierbar.

Wählt man hingegen den Ansatz über die Anforderung einer entsprechenden Systemeinheit, kann der Initiator der Aktion und die Art und Weise, wie die Information innerhalb des Systems verteilt wird, durch das Zielsystem selbst definiert werden. Hierzu muß innerhalb der Zielsystembeschreibung eine Systemeinheit definiert sein, die diese Funktion übernimmt. In Bild 6.6 ist ein Beispiel aufgeführt,

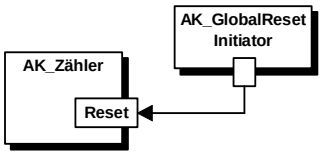

Bild 6.6. Anforderung einer Verbindung zum globalen Resetsignal

bei der eine Komponente, AK_Zähler, mittels einer weiteren Komponente den Zugang zum GlobalReset-Signal anfordert.

Einige Zielsysteme sehen hierfür keine eigenen fest definierten Steuerressourcen vor und überlassen es dem Entwickler, geeignete Implementierungen zu erzeugen. Aus Gründen der Wiederverwendbarkeit und der Portabilität ist es aber vorteilhaft, die hierfür zu implementierenden Mechanismen als Bestandteil der Systembeschreibung aufzunehmen.

Um dies zu unterstützen, werden virtuelle Systemeinheiten eingeführt. Jeder virtuellen Systemeinheit ist eine aktive Komponente zugeordnet, die die eigentliche Funktion der Ressource realisiert. Eine virtuelle Systemeinheit muß aus diesem Grunde einer realen Systemeinheit zugeordnet sein, auf der die ihr zugeordnete aktive Komponente realisiert werden kann.

Bei der Partitionierung wird eine virtuelle Systemeinheit wie jede andere Systemeinheit behandelt. Wenn aber eine Komponente auf der virtuellen Systemeinheit plaziert wird, hat dies zur Folge, daß die zugeordnete Komponente angewiesen wird, die entsprechende Funktionalität zu implementieren. In den weiteren Abbildungsschritten werden die Komponente zur Implementierung von virtuellen Systemeinheiten in gleicher Weise wie die restlichen aktiven Komponenten behandelt. Hierbei verursachen unbenutzte virtuelle Systemeinheiten keine Implementierungskosten.

Dieses Vorgehen soll anhand des Beispiels aus Bild 6.7 näher erläutert werden. Im dargestellten Zielsystem finden zwei unterschiedliche Arten von virtuellen Systemeinheiten Verwendung. Die Systemeinheiten vom Typ "Global Rest Initiator" (GRI) erzeugen ein Reset-Signal, wenn das Gesamtsystem zurückgesetzt wird und können somit von Komponenten angefordert werden, die ein Reset-Signal benötigen, um einen definierten Anfangszustand herzustellen.

Des weiteren existiert eine virtuelle Systemeinheit vom Typ "Global Reset Master" (GRM), mit deren Hilfe das Gesamtsystem zurückgesetzt werden kann. Auf dem Zielsystem sind mehrere GRIs vorhanden, was dazu führt, daß eine Komponente, die auf FPGA1 implementiert wird, diese virtuelle Systemeinheit auf FPGA1 benutzt, da beim Plazieren der GRI-Komponente die am nächsten gelegene Systemeinheit verwendet wird.

Für FPGA2 würde das bedeuten, daß, falls hier eine Komponente das Global Reset-Signal benötigt, es dieses über die auf FPGA1 enthaltene Ressource anfordern kann und dann die Verbindungen zwischen beiden FPGAs dazu benutzt werden, das Reset-Signal vom GRI auf FPGA1 zum FPGA2 zu transportieren. Gleichzeitig muß ein Verbindungsnetz vom GRM zu den GRI bestimmt werden,

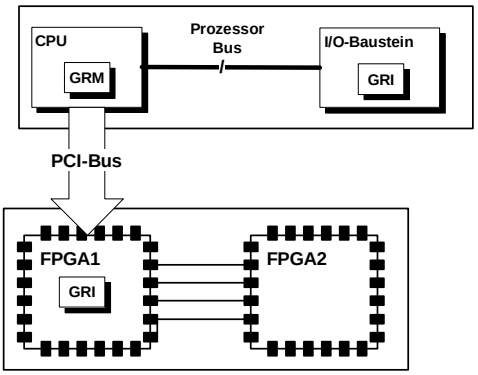

Bild 6.7. Zielsystem mit virtuellen Systemressourcen (GRM = Global Reset Master, GRI = Global Reset Initiator)

damit der GRM über die GRIs das Gesamtsystem zurücksetzen kann. Das führt auch dazu, daß für die verschiedenen GRIs verschiedene Implementierungen vorgesehen werden müssen.

Der Reset-Vorgang für Komponenten auf den FPGAs kann z.B. durch einen Schreibzyklus auf den PCI-Bus ausgelöst werden. Komponenten die auf dem I/O-Baustein realisiert sind, können zurückgesetzt werden, indem die CPU einen Reset-Zyklus auf dem Prozessorbus initiiert. Der GRI auf dem I/O-Baustein muß aus diesem Grunde keine Implementierung erzeugen, da der I/O-Baustein selbst über die entsprechende Reset-Logik verfügt. Auch müssen keine zusätzlichen Verbindungspfade aufgebaut werden, da die Verbindung über den Prozessorbus bereits besteht.

Für FPGA1 ergibt sich eine etwas andere Konstellation. Der GRI muß so ausgelegt sein, daß der GRM Daten an ihn senden kann. Es wird also eine Verbindung zwischen GRM und GRI verlegt, die ein Datum mit Gültigkeitssignal übertragen kann. Empfängt der GRI über diese Verbindung ein Datum, erzeugt er ein Reset-Signal für die Komponenten, die ihm zugeordnet wurden.

Der GRM ist so aufgebaut, daß er bei Aktivierung den Reset-Zyklus auf dem Prozessorbus anstößt und auf der Verbindung zum GRI auf FPGA1 ein Datum erzeugt, womit dann alle Komponenten auf dem Zielsystem, die das GlobalReset-Signal angefordert haben, zurückgesetzt werden.

Aus den hier dargelegten Beispielen ist ersichtlich, daß virtuelle Systemeinheiten dazu beitragen, die Portabilität von Problemlösungen zu erleichtern, da mit ihrer Hilfe zielsystemspezifische Aspekte aus der Problembeschreibung in die Zielsystembeschreibung übernommen werden können, was gleichzeitig zu einer höheren Wiederverwendbarkeit führt.

7
Sonderfälle bei Knoten und Routing

7.1 Knoten mit mehreren Initiatoren

Nicht behandelt wurden bisher Knoten mit mehreren Kommunikationsinitiatoren. Im einfachsten Fall stellt dies eine ungültige Anordnung dar und führt dazu, daß die Umsetzung der Lösungsbeschreibung abgebrochen wird. Dennoch gibt es Fälle, bei denen derartige Anordnungen erlaubt sind.

Hierbei muß berücksichtigt werden, daß es zu Zugriffskonflikten kommen kann, wenn mehrere Initiatoren am gleichen Knoten einen Kommunikationsvorgang einleiten. Dies verhindert in vielen Fällen eine erfolgreiche Kommunikation, kann aber auch zusätzlich dazu führen, daß es zu Schäden auf dem Zielsystem kommt. Bild 7.1 zeigt eine derartige Konstellation.

Hier sind drei Ports über einen gemeinsamen Knoten miteinander verbunden, wobei jeder Port als Initiator auftreten kann. Der verwendete Porttyp stellt eine Tri-State Verbindung dar. Ein Port kann hierbei die drei Zustände High, Low und hochohmig einnehmen. Die ersten beiden Zustände übermitteln die gewünschte Information, wohingen der dritte Zustand dazu dient, daß der entsprechende Port lediglich als Eingang betrieben wird.

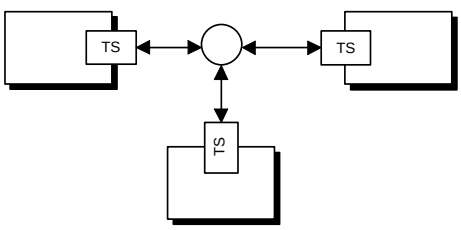

Bild 7.1. Knoten mit mehreren Initiatoren

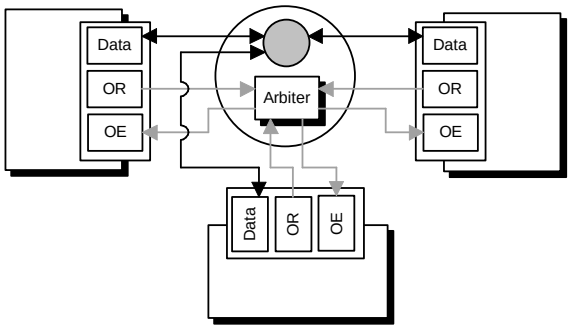

Bild 7.2. Knoten mit Implementierung zur Realisierung der Arbitrierung

Diese Anordnung stellt eine korrekte Verschaltung der Ports dar und ist so lange unkritisch, so lange nur ein Teilnehmer nicht im hochohmigen Zustand ist. Überträgt aber ein Teilnehmer einen Low-Pegel und ein anderer zum gleichen Zeitpunkt einen High-Pegel, schlägt der Kommunikatinsvorgang fehl. Zusätzlich fließen noch hohe Ausgleichströme, die zu lokaler Erwärmung und z.B. bei FPGAs zur Zerstörung des Bausteins führen können.

Will man derartige Konstellationen nicht grundsätzlich verbieten, kann man eine sichere Implementierung dadurch erzielen, daß man den Porttyp modifiziert und die Funktion des Knotens erweitert. Der Porttyp muß so erweitert werden, daß ein Port nur aktiv wird, wenn er hierfür eine Freigabe erhalten hat. Der Knoten muß zusätzlich zu seiner Verbindungsfunktion noch eine Kontrollfunktion einnehmen. Wie in Bild 7.2 dargestellt, setzt sich der Knoten jetzt aus einer aktiven Komponente und einem Knoten zusammen.

Dieser Knoten hat jetzt aber tatsächlich nur seine ursprüngliche Funktion, die angeschlossenen Ports zu verbinden und wird deshalb als reiner Knoten bezeichnet. Die für den Knoten neu erzeugte Komponente führt die Arbitrierung durch. Jeder Port, der ein Datum übertragen will, aktiviert zuerst seinen OR (Output Request) Unterport. Indem der Arbiter sicherstellt, daß zu jedem Zeitpunkt nur ein OE (Output Enable) Unterport aktiv ist, kann er verhindern, daß Zugriffskonflikte entstehen.

Für jeden Porttyp muß also festgelegt werden, ob mehrere Initiatoren an einem Knoten erlaubt sind und mit welcher aktiven Komponente in diesem Fall der Knoten realisiert werden kann. Diese Information wird in der Porttypdefinition hinterlegt. Die bei einer Knotenimplementierung erzeugten reinen Knoten und Komponenten werden in der bekannten Weise weiterverarbeitet.

7.2 Routing von abstrakten Typen

Wie in Abschnitt 4.2.2.1 dargelegt, können einzelne Routingkomponenten verschiedene Porttypen unterstützen. Zusätzlich gibt es Konstellationen, in denen Routingkomponenten Typumwandlungen vornehmen können.

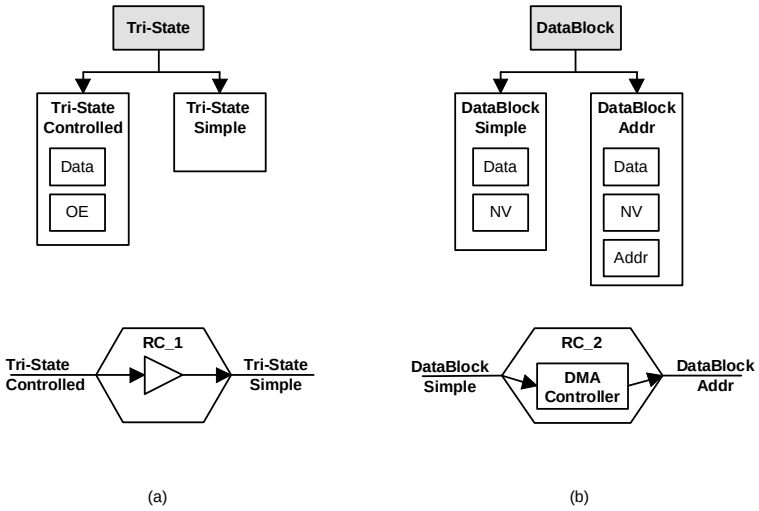

Bild 7.3. Verwendung abstrakter Porttypen beim Routing

Wie dieser Sachverhalt zur Erweiterung der Verbindungsmöglichkeiten genutzt werden kann, soll an den folgenden Beispielen näher erläutert werden.

In Bild 7.3 sind zwei abstakte Porttypen mit jeweils zwei zugeordneten konkreten Typen dargestellt. Zusätzlich sind zwei Routingkomponenten vorhanden, die eine Konvertierung zwischen den verschiedenen konkreten Formen der abstrakten Typen durchführen können.

Beim Verlegen einer Verbindung zwischen zwei Ports kann zuerst der abstrakte Typ vereinbart werden. Es werden dann die relevanten Routingressourcen und für die Teilverbindung vom Port zur Routingressource ein geeigneter konkreter Typ ausgewählt. Von dieser Routingkomponente aus wird dann die Verbindung zum Ziel oder einer weiteren Routingkomponente verlegt.

Trifft man hierbei auf eine Routingkomponente, die den aktuellen Typ in einen anderen Typ wandeln kann, ist es möglich hier eine Typübergang zu realisieren. Auch für den letzten Verbindungsabschnitt zum Zielport hin, wird dann ein konkreter Porttyp für den Zielport ausgewählt.

Dieses Vorgehen ermöglicht es Verbindungen zwischen Ports herzustellen, obwohl sie verschiedene Porttypen realisieren.Die Porttypen müssen die gleiche grundlegende Funktion breitstellen, was dadurch sichergestellt wird, daß sie einen gemeinsamen abstrakten Basistyp haben. Dies erhöht die Flexibilität beim Routing und kann in einigen Fällen auch zur Reduzierung des Ressourcenbedarfs beitragen. Dies soll an zwei Beispielen näher erläutert werden.

Als Ausgangsbasis für das erste Beispiel soll der abstrakte Porttyp Tri-State dienen (Bild 7.3a). Dieser Porttyp repräsentiert eine Tri-State Verbindung und bietet hierfür zwei Realisierungsvarianten in Form konkreter Porttypen an. Die eine Variante (Tri-StateSimple) stellt einen Port dar, der die drei Zustände der Tri-State Verbindung annehmen kann.

Die Zweite Variante (Tri-StateControlled) hingegen, verwendet zwei digitale Signale. Data legt die zu übertragende Information fest und OE zeigt mit einem aktiven Pegel an, daß diese Daten übertragen werden sollen. Dieser Porttyp ist so ausgelegt, daß mittels eines Tri-State Treibers eine Umwandlung in ein echtes Tri-State Signal vom Typ Tri-StateSimple erfolgen kann.

Für das Routing ist dieser Zusammenhang dadurch von Bedeutung, daß nur wenige Systemeinheiten dazu in der Lage sind, Tri-State Signale direkt zu verlegen, sondern meist nur digitale Signale unterstützen. Dennoch stellen diese Bausteine in vielen Fällen Tri-State Teiber für ausgehende Signale bereit. Benötigt bei einer Kommunikation die Datensenke den Porttyp Tri-StateSimple, führt dies dazu, daß die Verbindung auch mit diesem Typ verlegt werden müßte, weshalb es in den oben geschilderten Fällen dazu führt, daß die Verbindung nicht verlegt werden kann.

Benutzt man aber zum Verlegen den abstrakten Basistyp Tri-State und wird von der Datenquelle der reale Porttyp Tri-StateControlled unterstützt, kann die Verbindung bis zum letzten Verbindungsschritt mit dem Typ Tri-StateControlled verlegt werden und erst in der Letzten Verbindungsstrecke (falls ein entsprechender Tri-State Teiber bereitgestellt wird) in den für die Datenquelle benötigten Porttyp Tri-StateSimple konvertiert werden.

Auch für den zweiten in Bild 7.3 dargestellten abstrakten Porttyp DataBlock ergeben sich ähnliche Anwendungsfälle.

Hier soll der Fall betrachtet werden, bei dem von einer Komponente aus, Daten direkt in einen Speicher übertragen werden. Für die Übertragung des Datenblocks stehen zwei Verfahren bereit. Die einfachere Variante stellt der Porttyp DataBlockSimple dar. Er enthält zwei Unterports, wobei Data die Daten überträgt und NV die Gültigkeit der übertragenen Daten anzeigt. Der zweite Typ, DataBlockAddr, benutzt hingegen noch einen zusätzlichen Unterport Addr, der die Position des Datums innerhalb des Datenblocks angibt.

Da der Speicher eine Adresse zur Speicherung der Daten benötigt, müßte die Datenquelle um eine Funktion erweitert werden, die diese erzeugt. Findet aber die Datenübertragung auf dem Zielsystem mittels eines DMA-Controllers statt, kann dieser die von der Datensenke benötigten Adressignale automatisch, ohne weitere Implementierungskosten realisieren. Wird nun beim Routing der abstrakte Basistyp DataBlock verwendet, kann die Verbindung von der Datenquelle bis hin zum DMA-Controller den Typ DataBlockSimple verwenden, wird dort dann in den Typ DataBlockAddr gewandelt und in dieser Form bis zum Speicher verbunden.

8
Ansätze für Simulation und Monitoring

Nachdem eine Problemlösungsbeschreibung erstellt wurde, muß überprüft werden, ob die so beschriebene Funktionalität die gestellten Aufgaben tatsächlich erfüllt. Hier sollen zwei mögliche Varianten angesprochen werden, wobei lediglich betrachtet werden soll, welche Besonderheiten sich durch das hier vorgestellte Programmierverfahren ergeben und welches prinzipielle Vorgehen dadurch nahegelegt wird.

8.1 Gemeinsame Grundlagen

Das übliche Vorgehen, die Korrektheit einer Lösungsbeschreibung zu prüfen, besteht darin, diese in eine Form zu überführen, die es ermöglicht, aus vorgegebenen Eingabewerten Ergebnisse bestimmen zu können, wobei diese dann wiederum mit den Erwartungswerten überprüft werden.

Diese ausführbare Form läßt sich nun in zweierlei Hinsicht klassifizieren. Das erste Kriterium ist die Beschreibungsebene, die dem Ausführungsmodell zugrundeliegt. Die Lösungsbeschreibung wird auf ihrem Weg zur eigentlichen Implementierung in mehrere andere Beschreibungsvarianten gewandelt, wobei in jedem Schritt immer mehr Implementierungsdetails festgelegt und somit ein stetiges Vorschreiten von einer abstrakten, funktionalen Beschreibung hin zu einer konkreten, auf dem Zielsystem ausführbaren Beschreibung vollzogen wird.

Um so näher ein Ausführungsmodell der Implementierungsebene liegt, um so exakter und umfangreicher sind die Ergebnisse, die sich daraus gewinnen lassen. Gleichzeitig ergibt sich aber das Problem, daß die gewonnene Ergebnisse auch interpretiert werden müssen. Dies wird dadurch erschwert, daß der Entwickler der Problemlösung nur die oberste Beschreibungsebene erstellt hat und nun eine

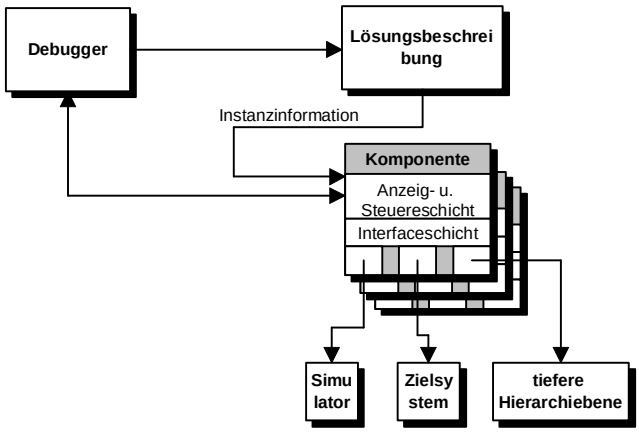

Bild 8.1. Prinzipieller Aufbau für Simulation und Monitoring

Zuordnung der Ergebnisse in diese Beschreibungsebene vornehmen muß. Hierfür sind weitreichende Kenntnisse des zugrundeliegenden Zielsystem und der dort verwendeten Bausteine notwendig.

Ein Aufführungsmodell auf höherer Ebene hingegen, erlaubt zwar eine einfachere Interpretation der Ergebnisse, beschreibt aber auch nur Sachverhalte, die sich auf der gegebenen Abstraktionsebene abspielen.

Ein weiteres Unterscheidungsmerkmal besteht darin, in welcher Form das Ausführungsmodell vorliegt. Hierbei verwendet man entweder die eigentliche Implementierung auf dem Zielsystem und muß dann entsprechende Verfahren bereitstellen, die es gestatten, die gewünschte Information aus dem aktuellen Zustand des Zielsystems zu extrahieren, oder man erstellt ein eigenes Ausführungsmodell, das mittels eines Computers ausgeführt werden kann. Die gewünschte Information wird hierbei den internen Zuständen der Simulationsmodelle entnommen.

Obwohl bei beiden Ansätzen die Ausführungsergebnisse auf verschiedene Weise gewonnen werden, sind die Schritte, die für die Auswertung und Visualisierung der gewonnenen Daten notwendig sind, im wesentlichen gleich (Bild 8.1).

Der als Debugger bezeichnete Teil stellt das zentrale Bedienelement für den Benutzer dar. Hier kann er in den Ablauf der Programmausführung eingreifen und die Ergebnisse visualisieren.

Als Basiseinheit dienen auch hier wieder die aktiven Komponenten, da sie ja die Hauptbestandteile der Lösungsbeschreibung darstellen und für die Veranschaulichung des aktuellen Ausführungszustandes der Problemlösung maßgebend sind. Der Debugger muß also den Zustand der Komponenten auf allen Beschreibungsebenen feststellen, beeinflussen und anzeigen können, wobei beides sehr stark vom jeweiligen Typ der Komponente abhängt.

Aus diesem Grund wird die Definition der aktiven Komplementen um eine Anzeige- und Steuerschicht erweitert. Diese Anordnung ist notwendig, da nur die Komponenten über ihren internen Aufbau und ihr Ausführungsverhalten Kenntnis haben. Nur sie verfügt über ausreichende Informationen, wie der aktuelle Aus-

führungszustand zu interpretieren, zu ändern oder sinnvoll anzuzeigen ist. Auch muß bei diesen Ansatz bei der Erstellung neuer Komponenten keine Änderung am Debugger vorgenommen werden.

Eine weitere Schicht, die Interfaceschicht, macht die Anzeige- und Steuerschicht unabhängig vom zugrundeliegenden Ausführungsmodell. Sie muß die geeigneten Verfahren wählen, um aus dem Ausführungsmodell die gewünschten Daten zu extrahieren bzw. zu setzen, wobei ein einheitliches, für Anzeige- und Steuerschicht verwertbares Format verwendet wird.

Neben der Möglichkeit, den aktuellen Ausführungszustand direkt aus dem Ausführungsmodell zu bestimmen, kann man bei hierarchisch aufgebauten Komponenten deren Zustand auch aus dem Zustand der für die Implementierung verwendeten untergeordneten Komponenten ermitteln.

Mit dem gleichen Verfahren ist es möglich, bei Ausführungsmodellen auf tiefergelegenen Beschreibungsebenen auch eine Darstellung der Ereignisse in Form der in den übergeordneten Beschreibungsebenen verwendeten Komponenten zu ermöglichen. Um dies zu erreichen, sind alle Umsetzungsschritte von der Lösungsbeschreibung bis hin zur Implementierung so ausgelegt, daß eine eindeutige Zuordnung der Elemente der verschiedenen Beschreibungsebenen zueinander gewährleistet ist. Die Abbildung der Zustandsdaten zwischen den Beschreibungsebenen ist Aufgabe der Komponenten.

8.2 Monitoring

Beim Monitoring besteht die Aufgabe darin, den Ausführungszustand der einzelnen Komponenten über den Zustand der Systemeinheiten des Zielsystems zu ermitteln.

Einige dieser Bausteine verfügen über spezielle Schnittstellen, die diesen Zweck erfüllen[1]. So besteht bei FPGAs oft die Möglichkeit, den aktuellen Zustand des gesamten Bausteins über eine entsprechende Readback-Schnittstelle auszulesen. Aus dieser Datenmenge müssen die für die betrachtete Komponente relevanten Informationen extrahiert und anschließend interpretiert werden.

Für Bausteine, die diese Funktionalität nicht bereitstellen, kann die Komponente ihre Implementierung dahingehend erweitern, daß sie eigene Schnittstellen und Zugriffspfade erzeugt, so daß die Komponenten der Lösungsbeschreibung in der Lage sind, die Zustandsinformation auszulesen.

Um hier einen eindeutigen Zugangspunkt zu errichten, wird in der Zielsystembeschreibung eine eigene virtuelle Systemeinheit angelegt, über die alle Monitoring-Schnittstellen erreichbar sind. Hierbei wird die entsprechende Komponentenimplementierung, bzw. die Schnittstelle zum Monitoring-Interface dadurch aus-

[1] Für die Nutzung derartiger Schnittstellen existieren besondere Randbedingungen, die es z.B. notwendig machen können, die Ausführung der auf dem entsprechenden Baustein realisierten Funktionalität anzuhalten. Da hier nur ein prinzipieller Überblick gegeben werden soll, wird auf diese Details nicht eingegangen.

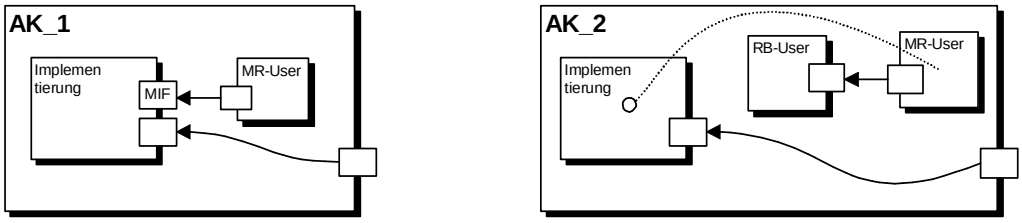

Bild 8.2. Implementierungsvarianten zur unterstützung des Monitoringinterfaces
(MIF = Monitoringinterface, RB = Readbach, MR = Monitoringresource)

gewählt, daß sie durch eine eindeutige Kennung identifiziert wird. Um diese Möglichkeit zu unterstützen, kann eine Komponente in zwei Varianten aufgebaut sein (Bild 8.2).

Die aktive Komponente AK_1 implementiert ein eigenes Monitoring-Interface und beschreibt aus diesem Grunde ihre Implementierung durch zwei Komponenten. Eine von ihnen realisiert die Funktionalität der Komponente inklusive der Monitoring-Schnittstelle. Die zweite Komponente, MR-User, dient dazu, die Monitoring-Schnittstelle anzusteuern und wird auf die Monitoring-Ressource abgebildet.

Eine zweite Variante AK_2 benutzt die Readback-Schnittstelle der Ausführungseinheit, auf der sie abgebildet wurde. Hierzu werden zwei Komponenten verwendet, wobei die eine, RB_User, den Zugang zur Readback-Schnittstelle herstellt und die andere, MR-User, wiederum die Verbindung zur Monitoring-Ressource realisiert.

Zusätzlich enthält MR-User einen Verweis auf die Implementierungskomponente von AK_2, da hier die Informationen bereitgestellt werden, mit denen die relevanten Zustandsdaten extrahiert werden können. Für die Gesamtimplementierung auf dem Zielsystem ergibt sich dann die in Bild 8.3 dargestellte Konstellation.

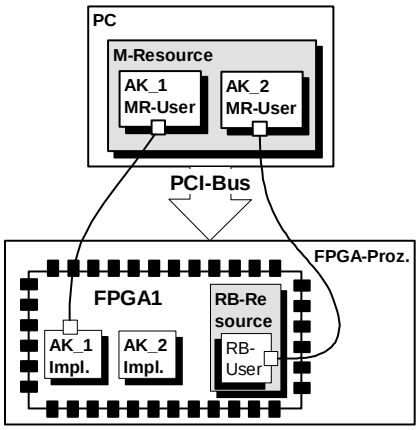

Bild 8.3. Realisierung des Monitorings auf dem Zielsystem

8.3 Simulation

Bei der Simulation besteht das Hauptproblem darin, geeignete Simulationsmodelle für die einzelnen Komponenten bereitzustellen. Diese Aufgabe muß der Komponente selbst zugeordnet werden, da nur sie genügend Informationen über ihren internen Aufbau verfügt.

Eine erste Möglichkeit besteht darin, ein eigenes Simulationsmodell bereitzustellen. Sie kann ihren Zustand aber auch aus den Simulationsmodellen der für die Implementierung verwendeten Komponenten berechnen. Eine weitere Möglichkeit ergibt sich für Komponenten, deren Implementierung in einer Implementierungssprache vorliegt. Hier kann ein externer Simulator für diese Sprache verwendet werden, um den Zustand der Komponente mittels ihrer Implementierungsbeschreibung zu bestimmen.

9
Implementierung des Basissystems

Im folgenden wird ein Demonstrationssystem vorgestellt, das die Programmierung FPGA-Basierter Systeme mittels aktiver Komponenten ermöglicht. Zu Beginn wird die Abbildung der Beschreibungsmittel auf eine universelle Programmiersprache behandelt. Im Anschluß daran, werden die realisierten Eigenschaften der einzelnen Umsetzungsschritte, die verfügbaren Porttypen, die dazugehörigen Portadapter, Basiskomponenten und die unterstützten Zielsysteme behandelt.

9.1 Abbildung der Beschreibungsmerkmale in Java

Als Sprache zur Implementierung des Prototypen wurde Java verwendet. Da es sich hierbei um eine objektorientierte Sprache handelt, werden die für die Programmierung mit aktiven Komponenten notwendigen Beschreibungselementen als Objekte bzw. Objektklassen abgebildet. Ein Klassendiagramm hierfür ist in Bild 9.1 dargestellt.

Ausgangsbasis für die Abbildung der Lösungsbeschreibung ist die Klasse der aktiven Komponenten (AK). Sie stellen mehrere Komponenten-Funktionen bereit, auf die mittels Accessports zugegriffen werden kann. Jeder Accessport enthält die für die Ansteuerung der zugeordneten Funktion benötigten Ports.

Zusätzlich stellen die aktiven Komponenten Funktionen zur Erzeugung von Accessports, zur Bestimmung der möglichen Porttypen, dem Hinzufügen von untergeordneten aktiven Komponenten und zur Durchführung der Implementierung der Komponente auf dem Zielsystem bereit. Attribute erlauben es, die Eigenschaften der Komponenten näher zu spezifizieren.

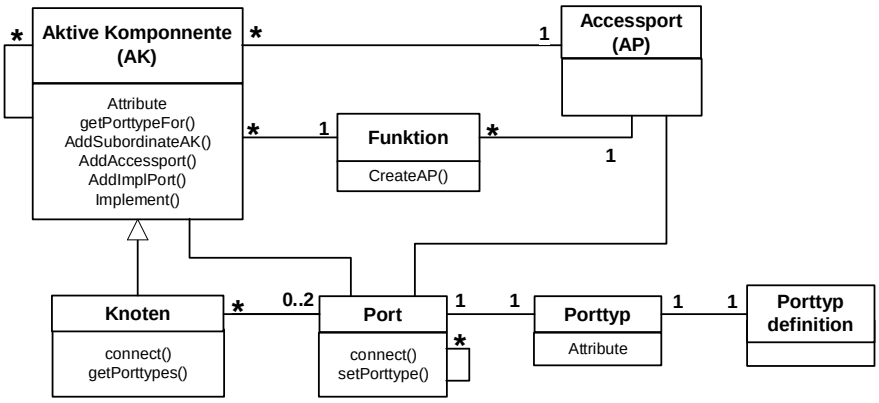

Bild 9.1. Klassendiagramm der zur Abbildung der Lösungsbeschreibung benötigten Klassen

Ports, über die die aktiven Komponenten Informationen austauschen, verfügen über einen Porttyp, der sich wiederum aus einer Porttypdefinition und Attributen zusammensetzt.

Während die Porttypendefinition die grundlegende Funktionsweise des Ports definiert, beeinflussen die Attribute die Auswahl der Realisierungsvarianten und Leistungsmerkmale.

Von der Klasse der aktiven Komponenten abgeleitet, ist die Klasse der Knoten. Knoten ermöglichen die Verbindung mehrere Ports, können aber auch eine Implementierung für den Knoten erzeugen, wenn dies vom verwendeten Porttyp gefordert wird. Zusätzlich sind sie in der Lage, die Schnittmenge der von den angeschlossenen Ports unterstützten Porttypen zu bilden.

Die Erstellung einer Lösungsbeschreibung kann also dadurch erfolgen, daß die benötigten AK-Klassen instanziert, mit den benötigten Accessports ausgestattet, die relevanten Attribute gesetzt und Ports mittels der bereitgestellten Funktionen untereinander verbunden werden.

Die für die Abbildung auf das Zielsystem benötigten Klassen zeigt Bild 9.2.

Das Zielsystem setzt sich aus Systemeinheiten zusammen, die hierarchisch gegliedert sein können. Beim Plazieren wird für jede aktive Komponente eine Systemkomponente und für jeden Port ein Systemport erzeugt bzw. zugewiesen. Sowohl die Systemkomponenten als auch die Systemports repräsentieren einzelne Implementierungsressourcen der Systemeinheit.

Durch die Zuordnung einer aktiven Komponente zu einer Systemkomponente wird die entsprechende Implementierungsressource reserviert. Gleiches gilt für Ports und Systemports. Zusätzlich wird durch die Zielsystemklasse eine Funktion bereitgestellt, die die Implementierungsbeschreibung für die gesamte Systemeinheit erstellt.

Jeder Systemeinheit ist ein Router zugeordnet, der Verbindungen innerhalb der Systemeinheit verlegen kann. Dieser verwendet mehrere Routingkomponenten, die die auf der Systemeinheit möglichen Verbindungen repräsentieren. Jeder

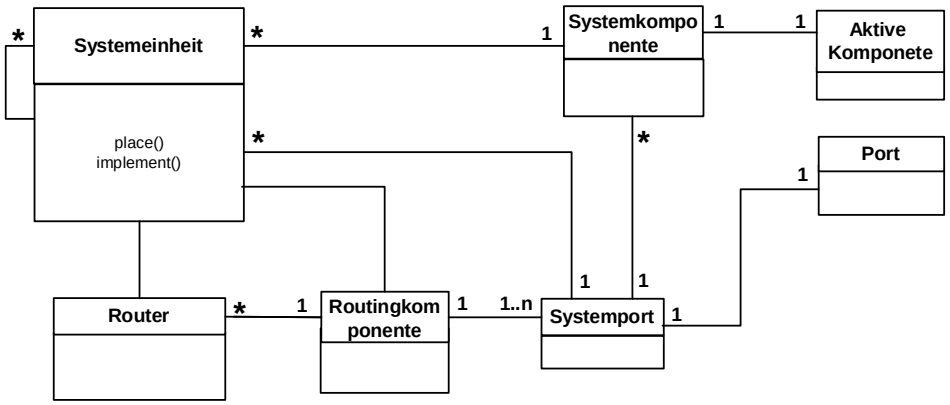

Bild 9.2. Klassendiagramm der zur Abbildung auf das Zielsystem benötigten Klassen

Routingkomponente sind die Systemports zugeordnet, die über sie verbunden werden können.

9.2 Realisierte Eigenschaften der Umsetzungsschritte

9.2.1 Plazieren

Das Plazieren der für die Lösungsbeschreibung vorgesehenen Komponenten kann prinzipiell automatisch erfolgen. Hierbei wird aber für jede Komponente die erste Systemeinheit als Ziel ausgewählt, auf der eine Abbildung der entsprechenden Komponente möglich ist. Somit bietet es sich an, die Komponenten auf oberster Ebene mit Hand zu plazieren.

Erzeugt eine Komponente zur Erstellung ihrer Implementierung weitere Komponenten, wird versucht, diese auf die gleiche Systemeinheit wie die übergeordnete Komponente zu plazieren. Ist dies nicht möglich, wird eine Systemeinheit ausgewählt, die der Systemeinheit der übergeordneten Komponente möglichst nahe liegt.

Beim Plazieren werden Komponenten, die Taktgeneratoren verwenden, gesondert behandelt. Es wird sichergestellt, daß die von der Lösungsbeschreibung vorgegebenen Eigenschaften und Abhängigkeiten durch die zugewiesenen Taktgeneratoren erfüllt wird.

Für derartige Komponenten, die zusätzlich keine expliziten Einschränkungen für die verwendbaren Taktgeneratoren machen, wird der zu verwendende Taktgenerator dadurch bestimmt, daß man zuerst die Systemeinheit ermittelt, auf der das Vaterobjekt der betrachteten Komponente plaziert ist. Von dieser Systemeinheit wird der Standard Taktgenerator erfragt und dieser dann der Taktgeneratorkomponente zugewiesen. Hierdurch erreicht man, daß für Komponenten auf einer Systemeinheit alle unspezifizierten Taktsignale einem einzigen Taktgenerator entnommen werden. Dies vermeidet unnötige Synchronisatonsstufen.

Außerdem werden diese Komponenten nicht in den ersten Plazierungsdurchläufen behandelt. Die Plazierung erfolgt erst, nach dem alle Komponenten plaziert wurden, deren Implementierung unabhängig vom gewählten Takt ist. Dies ist dadurch möglich, daß eine Komponente, die dazu aufgefordert wurde, ihre Implementierung zu erstellen, dies ablehnt und einen späteren Implementierungszeitpunkt anfordert.

Nachdem die Taktgeneratoren zugewiesen wurden, werden die Komponenten plaziert, deren Implementierung von der Wahl der Taktsignale abhängt. Zu dieser Klasse von Komponenten gehören z.B. Synchronisationskomponenten. Um hier eine optimale Implementierung durchführen zu können, muß festgestellt werden, ob tatsächlich unterschiedliche Taktsignale Verwendung finden. Ist dies nämlich nicht der Fall, kann eine wesentlich ressourcensparendere Implementierungsvariante gewählt werden.

9.2.2 Routing

Der verwendete globale Router unterstützt das Verlegen beliebiger Porttypen. Auch können Ports über ihre Unterports verbunden werden. Das Verlegen von Verbindungen mit abstrakten Porttypen wird unterstützt.

Obwohl die benötigten Mechanismen zum Auflösen und Neuverlegen bereits verlegter Verbindungen implementiert sind, macht der Routing-Algorithmus in der realisierten Form davon keinen Gebrauch.

9.2.3 Implementierung der Komponenten

Bei der Erstellung der Implementierung der Komponenten werden die Verfahren der hierarchischen Beschreibung und der Verwendung von Implementierungssprachen unterstützt.

Wird für die Implementierung eine Implementierungssprache benutzt, so muß die Komponente eine textuelle Beschreibung ihrer Funktion mit den Mitteln der ausgewählten Implementierungssprache bereitstellen. Bei Komponenten, die lediglich über feste Implementierungen verfügen, kann diese Beschreibung in Form einer Datei abgelegt sein. Die Komponente muß dann für ihre Implementierung nur noch die geeignete Date referenzieren.

In vielen Fällen muß aber eine Implementierung erstellt werden, die auf die vorgegebenen Randbedingungen wie die verwendeten Porttypen, Anzahl der Accessports und gesetzten Attribute in Betracht zieht. Hier stehen der Komponente, die als Objekt in der Java definiert ist, alle Sprachmittel dieser Sprache zur Verfügung, um eine geeignete textuelle Beschreibung zu erstellen.

Unterstützt werden in der vorliegenden Realisierung die zwei Implementierungssprachen C++ für den Softwarebereich und CHDL für den Bereich der Hardwarebeschreibung.

9.3 Porttypen und Portadapter

Die vom Basissystem bereitgestelltem Porttypen zeigt Bild 9.3. Wie bereits erläutert, verfügt jeder Port über das Attribut IsMaster, das angibt, ob der jeweilige Port als Initiator einer Informationsübertragung auftritt. Das führt dazu, daß es für jeden Porttyp zwei Varianten gibt. Die eine stellt den Initiator, die andere die Gegenstelle dar. Handelt es sich hierbei um Porttypen, die weitere Unterports spezifizieren, ist die Zuordnung der Initiatorfunktion bei beiden Varianten genau entgegengesetzt.

In Bild 9.3 ist die Darstellung so gewählt, daß die Ausprägung der Unterports so angegeben ist, wie sie der Initiatorvariante des Portstyps entspricht, der diese Unterports erzeugt. Alle hier verwendeten Portadapter stellen Zusatzports für alle Unterports der jeweiligen Porttypen bereit und erlauben somit die Kaskadierung der Portadapter.

Die erste Gruppe der unterstützten Porttypen, die Basisporttypen, realisieren die einfachsten Möglichkeiten, Informationen zu übertragen. Der Porttyp Control transportiert eine binäre Steuerinformation, wobei deren Interpretation durch das Attribut ActiveState festgelegt wird. Der Wert dieses Attributes gibt an, welcher über den Port übertragene Wert als aktiver Zustand zu interpretieren ist. Es können die Werte High, Low und Edge[1] angegeben werden. Der bereitgestellte Portadapter ermöglicht es, eine Anpassung zwischen Porttypen durchzuführen, die für ActiveState unterschiedliche Werte aufweisen. Dies ist allerdings nur möglich, falls keiner der anzupassenden Typen den Wert Edge als Attributwert verwendet.

ControlPulsed stellt ebenfalls ein binäres Steuersignal dar, wobei ein aktiver Zustand dadurch angezeigt wird, daß der Port beim Auftreten einer positiven Flanke des zugeordneten Taktes einen High-Pegel führt. Ein aktives Signal wird bei jeder positiven Flanke als neue Information gewertet, so daß ein einzelnes Datum maximal während einer Taktperiode anliegen darf. Die Zuordnung des benötigten Taktes geschieht dadurch, daß im ClockMaster Attribut, die für die Erzeugung des Taktsignales zuständige Taktgenerator-Komponente angegeben wird.

Der zugehörige Portadapter gestattet die Anpassungen zwischen Ports, die zu unterschiedlichen Takten synchron sind. Der Portadapter stellt zusätzliche Ports bereit. Über ClockOut kann auf das Taktsignal zugegriffen werden, mit dem die Information angeliefert bzw. über ClockIn ein Takt vorgegeben werden, mit dem intern auf die übertragene Information zugegriffen werden soll. Die Adaption und die Behandlung des ClockMaster Attributs werden dann durch den Portadapter vorgenommen.

Der letzte Porttyp aus dieser Gruppe ist Data1. Hier wird lediglich eine einzelne binäre Information transportiert.

[1] Nimmt das ActiveState Attribut den Wert Edge an, wird das Signal als Taktsignal behandelt, wobei die positive Flanke als aktive Flanke benutzt wird.

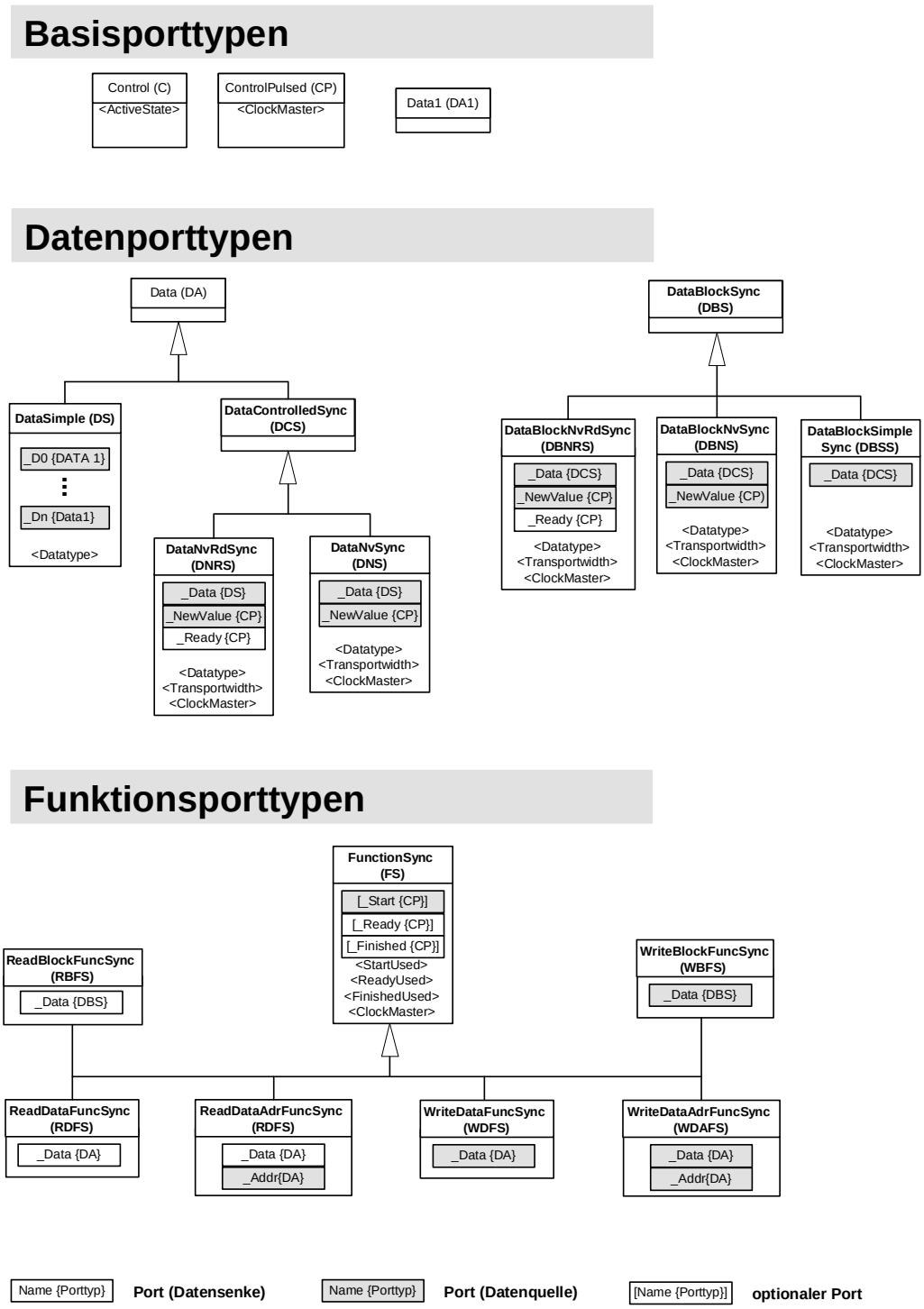

Bild 9.3. Bereitgestellte Porttypen

Die zweite Gruppe dient der Übertragung von Daten und Datenblöcken. Ausgangsbasis ist hierbei der Porttyp DataSimple. Er übermittelt Daten, wobei deren Typ im Attribut Datatype anzugeben ist. Alle Bits eines Datums werden gleichzeitig übertragen, weshalb als Unterports für jedes Bit des Datums ein eigener Port, _Dx [2], vom Typ Data1 verwendet wird.

Porttypen, die die kontrollierte Übertragung einzelner Daten synchron zu einem vorgegebenen Takt ermöglichen, werden im abstrakten Porttyp DataControlledSync zusammengefaßt. Konkretisierungen dieser Klasse liegen in den Typen DataNvRdSync und DataNvSync vor.

Beide benutzen zum Transport der Daten einen eigenen Datenport und zeigen über _NewValue an, daß ein neues Datum vorliegt. Der Datenkanal muß hierbei nicht mit der Bitbreite des Datums übereinstimmen, da die Möglichkeit besteht, mehrere Daten gleichzeitig bzw. nur einen Teil eines Datums zu transportieren. Die Bitbreite, mit der die Übertragung stattfindet, wird über das Attribut Transportwidth festgelegt. DataNvRdSync stellt zusätzlich noch das Steuersignal _Ready bereit, mit dem die Datensenke anzeigt, ob sie ein neues Datum entgegen nehmen kann. Ist _Ready inaktiv, dürfen keine weiteren Daten übertragen werden. Die verfügbaren Portadapter unterstützen die Synchronisierung von Ports, die unterschiedliche Taktsignale verwenden und gestatten die Festlegung der verwendeten Taktsignale durch die Zusatzports ClockIn und ClockOut (siehe ControlPulsed).

Zusätzlich werden Anpassungen zwischen Ports, die zwar den gleichen Datentyp aber unterschiedliche Transportbreiten verwenden, durchgeführt. Somit kann die Implementierung von Komponenten unabhängig von der jeweiligen Transportbreite der Daten gestaltet werden. Das Verhältnis zwischen Transport- und Datenbreite muß hierbei ganzzahlig sein und dem Wert 2^n *oder* $\frac{1}{2^n}$ $(n \in \mathcal{Z}, n \geq 0)$ entsprechen bzw. die Transportbreite kann auch ein Bit sein.

Ein weiterer abstrakter Porttyp, DataBlockSync faßt die taktsynchron Porttypen zusammen, die einen Datenblock übertragen können. Die konkreten Typen sind prinzipiell genauso aufgebaut wie DataNvRdSynv und DataRdSync, nur daß sich hier die Steuersignale nicht auf ein einzelnes Datum, sondern auf einen Datenblock beziehen. Der Datenkanal ist so ausgelegt, daß er die einzelnen Daten des Datenblocks kontrolliert übertragen kann, weshalb hier der Porttyp DataControlledSync vorgesehen ist.

Der verfügbare Portadapter kann zwar lediglich verschiedene Takte synchronisieren, der Portadapter des Data-Zusatzports stellt aber die gesamte Flexibilität des DataControlledSync Porttyps zur Übertragung der einzelnen Daten bereit. Der angegebene Datentyp definiert hier sowohl den Aufbau des Datenblocks, wie auch den Typ der Basiselemente des Blocks.

Die letzte Gruppe dient der Realisierung ganzer Funktionen. Sie verfügen über Datenports, die die Parameter der Funktion liefern bzw. die Ergebnisse weiterge-

[2] Die Namen der Unterports beginnen mit einem '_', um Namenskonflikte mit den durch die Portadapter angelegten Zusatzports zu vermeiden.

ben. Zusätzlich werden Steuersignale bereitgestellt die die Ausführung der Funktion starten, das erneute Starten der Funktion erlauben oder das Ende der Funktion und das Vorliegen der Ergebnisse anzeigen können. Alle diese Steuersignale sind optional, wobei durch entsprechende Attribute festgelegt wird, ob sie benutzt werden sollen oder nicht.

Eine Funktion kann dann ausgeführt werden, wenn für alle Parameter neue Daten vorliegen und alle Ergebnisports neue Daten entgegennehmen können. Wird das Start-Signal für einen Port nicht benutzt, StartUsed = false, dann wird anhand des oben beschriebenen Kriteriums die Funktion automatisch gestartet Ist hingegen das Start-Signal Bestandteil des Ports, so wird die Funktion durch die Aktivierung dieses Signals gestartet. Dies vereinfacht den Aufwand zur Auswertung des Startkriteriums der Funktion. Die Komponente, die das Start-Signal bedient, muß aber sicherstellen, daß die benötigten Daten gültig anstehen, bzw. die Ergebnisse weitergegeben werden können.

Die _Ready und _Finished Signale ermöglichen die effiziente Ansteuerung von Funktionen, deren Implementierung pipelineorientiert ist. Über _Ready kann angezeigt werden, daß bereits ein neuer Funktionsaufruf gestartet werden kann, obwohl die Ergebisse des zuvor gestarteten Funktionsaufrufes noch nicht vorliegen. Die Ergebnisse der Funktionsaufrufe können dann nacheinander bei aktiviertem _Finished-Signal entgegengenommen werden. Wird kein _Finished-Signal benutzt, muß jeder Datenport die Gültigkeit seiner Daten selbst anzeigen.

Der Portadapter führt Taktsynchronisierungen durch und ist in der Lage, Porttypen die kein _Start- bzw. _Finished-Signal verwenden, in Porttypen mit diesen Signalen umzuwandeln. Unterstützt werden zusätzlich Funktionen zum Lesen oder Schreiben eines einzelnen Datums bzw. eines Datums mit Adressinformation.

Die Funktionsporttypen werden auch für Softwareimplementierungen bereitgestellt.

9.4 Basiskomponenten

Es werden zwei universelle aktive Komponenten bereitgestellt. Die erste dient lediglich dazu, boolesche Gleichungen leichter realisieren zu können. Die zu implementierende Funktion wird ihr in Form einer Gleichung übergeben, wobei die zu verknüpfenden Ports einfach mit ihrem Namen angesprochen werden können. Die Komponente erzeugt dann automatisch die benötigte Implementierung.

Eine zweite Komponente realisiert eine allgemeine Grundstruktur zur Ansteuerung von Speicherbausteinen (Bild 9.4).

Diese Basiskomponente ist AK_Memory. Sie unterstützt Lese– und Schreibfunktionen, wobei die Möglichkeit besteht, auf ein einzelnes Datum, auf ein Datum innerhalb eines Adressbereiches oder auf einen ganzen Datenblock zuzugreifen.

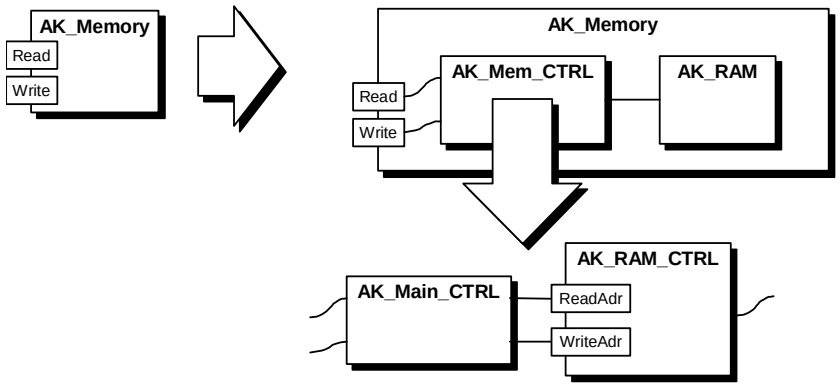

Bild 9.4. Aktive Komponente zur Realisierung von RAM-Zugriffen

Beim Anlegen des entsprechenden Accessports muß der Typ der zu speichernden Daten und eine eindeutige ID für den bereitzustellenden Datenbereich angegeben werden. Accessports mit gleicher ID greifen auf den gleichen Datenbereich zu.

In einem ersten Schritt implementiert sich die Komponente durch zwei andere Komponenten. AK_RAM dient dazu, die benötigten Speicherbausteine auf dem Zielsystem anzufordern. AK_Mem_CTRL übernimmt die Ansteuerung des Speichers.

AK_Mem_Ctrl implementiert sich wiederum mittels anderer Komponenten, wobei AK_RAM_CTRL den vom Typ der verwendeten Speicherbausteins abhängigen Teil übernimmt. Da AK_RAM bereits auf dem Zielsystem abgebildet wurde, ist auch der Typ der anzusteuernden Speicherbausteine bekannt, so daß eine geeignete Variante für AK_RAM_CTRL gewählt werden kann.

Für den eigentlichen Speicherzugriff werden Ports bereitgestellt, die den Typ ReadBlockAdrFunc oder WriteBlockAdrFunc unterstützen. Zusätzlich wird vorausgesetzt, daß auf jeden dieser Ports gleichzeitig, unabhängig von eventuellen Zugriffen auf anderen Ports, zugegriffen werden kann. Die aktuelle Implementierung unterstützt momentan nur einen Lese- und Schreibport pro Speicherbaustein.

AK_MAIN_CTRL führt dann Aufgaben wie Arbitrierung, Adressvergabe, Anordnung der Daten im Speicher und die Unterstützung verschiedener Porttypen durch. In der vorliegenden Implementierung wird eine freie Zahl von Accessports und verschiedenen Speicherbereichen unterstützt. Für jeden Speicherbereich wird ein Adressraum bestimmt und eine Anordnung der Daten gewählt, die eine möglichst effiziente Nutzung des Speichers ermöglichen.

Als Beispiel soll hier die Speicherung eines Datenblocks mit 32 8-Bit Daten dienen, wobei die Datenquelle in der Lage sein soll, die Daten mit 8- oder 16-Bit Breite zu transportieren. Weiterhin sei vorausgesetzt, daß der anzusprechende Speicher 16-Bit breit ist, wobei auf jedes Byte eines Speicherwortes auch separat zugegriffen werden kann.

In dieser Konstellation kann die Datenübertragung mit 8 oder 16 Bit erfolgen. Bei gleichzeitiger Übertragung von 16-Bit, können diese direkt in ein Speicherwort übernommen werden. Werden hingegen zur Übertragung nur 8 Bit verwendet, gibt es zwei Realisierungsvarianten. Die erste schreibt jedes übertragene Datum separat in eine Speicherworthälfte. Die zweite Variante faßt vor dem Schreiben zwei 8-Bit Daten zusammen und legt diese in einem Zugriff in einem Speicherwort ab. Die erste Variante verursacht im allgemeinen den geringeren Implementierungsaufwand, hat aber den Nachteil, daß die maximal nutzbare Speicherbandbreite reduziert wird, was gerade bei mehreren gleichzeitigen Zugriffen auf einen Speicherbaustein zu Leistungseinbusen führen kann. Variante 2 hat diesen Nachteil nicht, verursacht aber einen höheren Implementierungsaufwand.

Als Porttypen werden ReadDataFunkSync, WriteDataFuncSync, ReadDataAdrFuncSync, WriteDataAdrFuncSync, ReadBlockFuncSync und WriteBlockFuncSync unterstützt. Für diese Typen werden automatisch die zur Ansteuerung von AK_RAM_CTRL benötigten Adressignale erzeugt und auch eine Arbitrierung bei zeitgleichem Zugriff durchgeführt.

In einer Applikation kann also z.B. ein ganzer Datenblock in den Speicher geschrieben werden, ohne dabei irgendwelche Angaben über die Anordnung der Daten im Speicher oder der Ansteuerung der Speicherbausteine zu machen.

9.5 Zielsysteme

Als Zielsystem ist ein PC im Verbund mit einem FPGA-Prozessor vorgesehen. Bei dem verwendeten FPGA-Prozessor handelt es sich um eine PCI-Einsteckkarte mit einem FPGA der XILINX-4000er Reihe. Über das PCI-Interface findet die Kommunikation zwischen PC und FPGA-Prozessor statt. Zusätzlich sind auf dieser Karte eine SRAM Speicherbank, programmierbare Taktgeneratoren und über Aufsteckplatinen anpaßbare I/O-Verbindungen vorhanden.

Für das Gesamtsystem wird von der Zielsystembeschreibung ein Initialisierungsverfahren bereitgestellt, das vom PC aus die Initialisierung durchführt. Es existieren virtuelle Routingressourcen, die die Kommunikation von Daten und Datenblöcken zwischen PC und FPGA ermöglichen, wobei zur Blockübertragung auch die Übertragung mittels des auf der Karte vorhanden DMA-Controllers unterstützt wird.

Auf dem FPGA wird sichergestellt, daß Taktsignale nur auf den speziell vom Hersteller vorgesehenen Pfaden verlegt werden. An virtuellen Ressourcen stehen noch Implementierungen für globale Start- und Reset-Signale bereit. Die fertige Implementierung besteht aus einem auf dem PC ausführbaren Programm und einer Datei, die die Konfiguration des FPGAs beschreibt. Zur Ausführung der Gesamtapplikation ist es ausreichend, das Programm auf dem PC zu starten.

10

Ein Programmiersystem für Bildverarbeitungsanwendungen

Nachdem nun die Eigenschaften der Basissystemimplementierung beschrieben wurden, soll in diesem Kapitel das Potential der damit realisierbaren Programmiersysteme anhand einiger Beispiele aufgezeigt werden. Diese Beispiele sind aus dem Bereich der Bildverarbeitung entnommen. Dies ist dadurch motiviert, daß es sich bei diesem Anwendungsbereich um einen typischen Anwendungsfall für FGA-Prozessoren handelt und zum anderen in diesem Bereich im Rahmen des OpenEye-Projektes schon einige Problemlösungen mit herkömmlichen Programmierverfahren erstellt wurden.

Auf Grund dieser Voraussetzungen lassen sich der jeweilige Programmieraufwand bzw. die resultierenden Ergebnisse beider Ansätze vergleichen und zusätzlich der Aufwand abschätzen, der notwendig ist, um bereits bestehende, in herkömmlichen Implementierungssprachen erstellte Module in eine Programmierumgebung mit aktiven Komponenten aufzunehmen.

10.1 Anforderungen

Mit dem zu erstellende Programmiersystem sollen Applikationen entwickelt werden können, die die Bilder einer Progressiv–Scan-Kamera in Echtzeit verarbeiten und anzeigen. Es soll die Möglichkeit bestehen, verschiedene Operationen auf den Bildstrom anwenden zu können. Hierbei sollen arithmetische Operationen wie Addition, Subtraktion , Betragsbildung, trigonometrische Funktion, beliebige 2D-Filter und Farbraumkonvertierungen auf den Bildstrom anwendbar sein. Es soll eine möglichst hohe, problembezogene Beschreibungsebene ermöglicht werden.

Aus diesem Grund müssen Datentypen bereitgestellt werden, die ein ganzes Bild bzw. einen Bildpunkt beschreiben. Auch soll bei Bedarf die Bitbreite eines Bildpunktes automatisch angepaßt werden, wobei möglichst lange die maximal genutzte Farbtiefe erhalten bleiben soll. Es müssen Porttypen erstellt werden, die ganze Bilder bzw. Bildströme transportieren und auch die Bildsynchronisationsinformation bei Bedarf bereitstellen.

Die bereits im OpenEye-Projekt erstellten Module sollen mit möglichst geringem Aufwand in aktive Komponenten umgesetzt werden. Hierbei soll die Eigenschaft der für die Implementierung auf FPGAs vorgesehenen Module ausgenutzt werden, daß alle in jedem Takt ein neues Ergebnis liefern können.

Als Zielsystem soll ein PC mit einer microEnable FPGA-Prozessorkarte unterstützt werden und als Implementierungssprachen CHDL und C++ verwendet werden.

10.2 Realisierung

10.2.1 Datentypen

Zuerst müssen die benötigten Datentypen definiert werden. Der Typ, der ein Bild repräsentiert, hat Attribute, die die Anzahl der Zeilen bzw. Spalten und den Pixeltyp angeben. Zusätzlich ist ein Attribut vorgesehen, das festlegt, in welcher Reihenfolge die Pixel übertragen werden. In der aktuellen Implementierung wird dieses Attribut jedoch ignoriert.

Der Pixeltyp hat Attribute, die die Bitbreite der Pixel und deren Basistyp spezifizieren. Der Basistyp gibt an, wie die Pixelinformation zu interpretieren ist. Hierbei kann es sich um Grauwertpixel, RGB-Pixel oder gepackte RGB-Pixel (RGBP) handeln.

Bei den Farbformaten spezifiziert der Basistyp auch, wie die Aufteilung der verfügbaren Bits auf die Farbkanäle erfolgt und bei Grauwerten kann angegeben werden, ob die Pixelwerte als vorzeichenbehaftet oder vorzeichenlos behandelt werden sollen. Das gepackte RGB-Format wird nur für RGB-Pixel mit einer Breite von 24-Bit unterstützt und führt dazu, daß in einem 32-Bit Word ein vollständiger Pixel und ein Farbkanal des nachfolgenden Pixels übertragen wird. Sinnvoll ist dieses Format hauptsächlich, um die Bilddaten effizient in einem Speicher ablegen zu können.

Ein weiteres Attribut spezifiziert, welche Bildsynchronisationsinformation zusammen mit den Pixeldaten übertragen werden. Hierbei kann bei der Übertragung der Pixeldaten angezeigt werden, ob eine neue Zeile beginnt bzw. endet oder ob gerade ein neues Bild beginnt bzw. die Übertragung eines Bildes abgeschlossen wird.

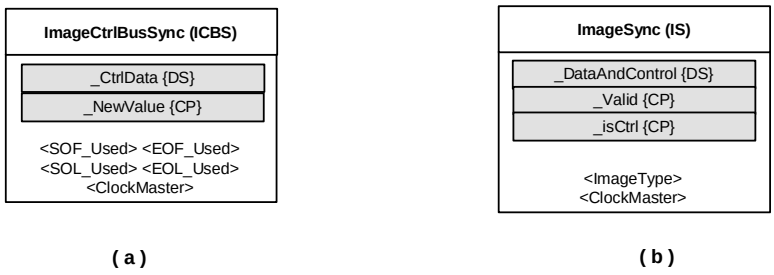

Bild 10.1. Porttypen zur Übertragung der Bild und Synchronisierungsinformation

10.2.2 Porttypen

Um die benötigten Porttypen zu bestimmen, müssen sowohl die Beschreibungs- als auch die Implementierungsebenen betrachtet werden. Auf der Beschreibungsebene muß ein Porttyp existieren, der die Übertragung der Bildinformation inklusive der Bildsynchronisationsdaten ermöglicht. Auf der Implementierungsebene ist es sinnvoll, das Kommunikationsverhalten der bereits existierenden Module zu betrachten. Sie verarbeiten hauptsächlich einen Pixelstrom bzw. einzelne Pixel, wobei zusätzliche Steuersignale die Gültigkeit der Daten anzeigen.

Da die Übertragung zusätzlich taktsynchron erfolgt, können hierfür die vom Basissystem bereitgestellten Porttypen DataNvSync bzw. DataBlockSync verwendet werden. Die Möglichkeit, die Übertragung von seiten der Datensenke zu verzögern, wird nicht genutzt, da die auf dem FPGA implementierten Module so ausgelegt sind, daß sie in jedem Takt neue Daten aufnehmen bzw. Ergebnisse liefern können. Sollte diese Funktionalität aber später dennoch einmal benötigt werden, könnte dies durch die Verwendung des Porttyps DataNvRdSync bzw. einer geeigneten Variante von DataBlockSync aus dem Basissystem realisiert werden.

Zusätzlich greifen einige Modulimplementierungen auf die Bildsynchronisationsinformationen zu. Während die einzelnen Synchronisationssignale als ein Port vom Typ ControlPulsed abgebildet werden können, ist es jedoch sinnvoll, die Synchronisationsinformation in ihrer Gesamtheit über einen Port übertragen zu können (Abschnitt 10.2.3). Der hierzu verwendete Porttyp ImageCtrlBus ist in Bild 10.1a dargestellt.

Der Port _CtrlData transportiert hier die eigentliche Information, während _NewValue wie bisher die Gültigkeit der Daten anzeigt. Über die bereitgestellten Attribute kann angegeben werden, welche Synchronisationsinformation benötigt wird. Die Breite von _CtrlData wird entsprechend angepaßt. SOF_Used bzw. EOF_Used legen fest, ob die Information über den Beginn eines neuen Bildes bzw. das Ende des aktuellen Bildes benötigt wird. Für SOL_Used und EOL_Used gilt das selbe, wobei sich die Angaben jedoch auf den Anfang bzw. das Ende einer Bildzeile beziehen.

Um die gesamte Bildinformation zu übertragen, wird der Porttyp ImageSync (Bild 10.1b) bereitgestellt. Über _DataAndControl werden sowohl die Pixeldaten als auch die Synchronisationsinformation zeitversetzt übertragen, wobei _Valid

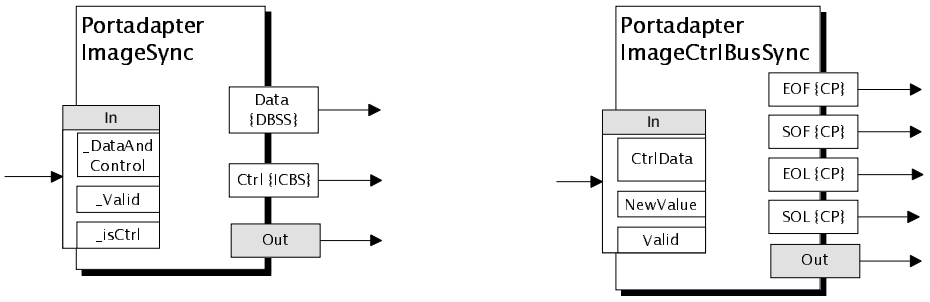

Bild 10.2. Verfügbare Portadapter

die Gültigkeit der Daten anzeigt und _isCtrl mit einem aktiven Signal anzeigt, daß es sich bei den übertragenen Daten um Synchronisationsinformationen handelt.

Die vorhandenen Module zur Ansteuerung der Videokamera verwenden das gleiche Verfahren und können somit leicht in die aktiven Komponenten eingebunden werden.

10.2.3 Portadapter

Aufgabe der Portadapter ist es, die Handhabung der Kommunikationsverfahren zu vereinfachen und Kommunikationsverfahren bzw. Porttypen untereinander anzupassen. Für die oben vorgestellten Porttypen werden die Portadapter aus Bild 10.2 bereitgestellt.

Der Portadapter für den Porttyp ImageSync spaltet den ankommenden Datenstrom in Pixeldaten (Data) und Synchronisationsinformation (Ctrl) auf, so daß die Implementierung der Komponente auf diese Daten über getrennte Ports zugreifen kann, wobei die Pixelinformation wie ein gewöhnlicher Datenblock behandelt wird. Für abgehende Daten muß der Portadapter aus den zwei Datenströmen, Pixeldaten und Synchronisationsdaten wieder einen gemeinsamen Datenstrom erzeugen. Er stellt auch sicher, daß der für den In-Port verwendete Bild- bzw. Pixeltyp so gewählt wird, daß er den Erfordernissen der mit dem Data- bzw. Ctrl-Port verbundenen Komponenten entspricht.

Zusätzlich ist der Portadapter in der Lage, Anpassungen der Pixelbreite bei Grauwertbildern vorzunehmen. Die Vorgaben sind hier, die maximal genutzte Farbtiefe weitestgehend zu erhalten, ohne dabei Implementierungsressourcen durch eine unnötig hohe Datenbreite zu vergeuden.

Um dies zu erreichen, wird eine Kostenfunktion verwendet, die für eine Veränderung der Pixelbreite Werte liefert, die eine möglichst kleine Abweichung von der Originalgröße bevorzugt und bei einer gleich großen Abweichung der größeren Bitbreite den Vorzug gibt.

Dies ist allerdings nicht ausreichend, da in vielen Fällen die Implementierungskosten nicht durch die Pixelbreitenadaption dominiert werden, sondern haupt-

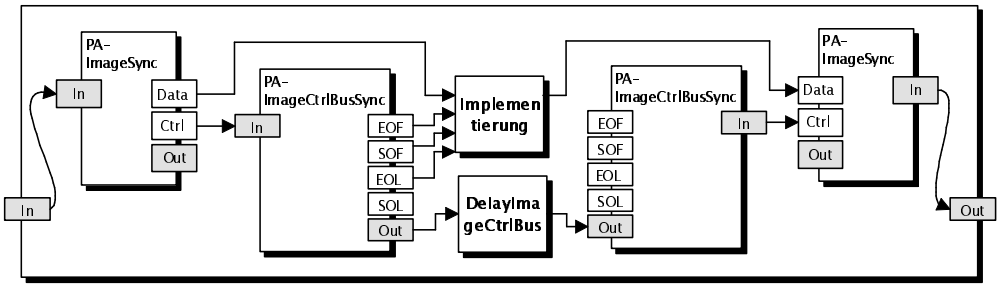

Bild 10.3. Einbindung bestehender Bildvearbeitungsmodule in aktive Komponenten

sächlich von den Implementierungskosten der angeschlossenen Komponenten bestimmt wird, so daß eine möglichst geringe Pixelbreite gewählt werden würde.

Aus diesem Grunde wird diese Anpassung nur bei der Datensenke vorgenommen. Nur Datensenken, die keine beliebigen Pixelbreiten unterstützen, nehmen diese Adaption vor.

Um hierbei aber den Verbrauch unnötiger Routingressourcen zu vermeiden, falls eine Reduktion der Pixelbreite unumgänglich ist, werden die ungenutzten, an der Datensenke ankommenden Unterports des Datenports als ungenutzt markiert und somit beim Routing nicht berücksichtigt.

Ähnliche Aufgaben übernimmt der Portadapter für Ports vom ImageCtrlBusSync. Er stellt die Synchronisationssignale als einzelne Steuersignale bereit und bestimmt den Typ des In-Ports, indem er die verbundenen Steuersignale und den Typ des Ausgansports Out ermittelt.

Wird hierbei bei einem Portadapter für eingehende Daten eine der vier vorhandenen Synchronisationsinformationen entweder über die Steuersignale oder den Out-Port benutzt, wird diese für den In-Port als benutzt markiert. Für ausgehende Daten wird hingegen durch den Portadapter die Möglichkeit bereitgestellt, über den In-Port weniger Steuerinformationen zu übertragen, als intern bereitgestellt werden, falls diese von den verbundenen Komponenten nicht benötigt werden.

Bild 10.3 zeigt, wie mit dieser Konstruktion bestehende Module leicht in aktive Komponenten eingebettet werden können. Über den ImageSync-Portadapter des In-Ports gelangen die Pixeldaten des ankommenden Bildstromes zu der durch das existierende Modul bereitgestellten Implementierung. Über den ImageCtrlBusSync-Portadapter können die benötigten Synchronisationssignale abgegriffen werden. Der durch die Implementierung des Moduls erzeugte Pixelstrom wird an den ImageSync-Portadapter des Out-Ports weitergereicht. Mit den von ImageCtrlBusSync-Portadapter gelieferten Synchronisationsinformationen wird daraus der Bildstrom erzeugt.

Da das verwendete Modul zumeist eine zeitliche Verzögerung hervorruft, müssen auch die Synchronisationssignale entsprechend verzögert werden, damit auch beim ausgehenden Bildstrom eine Übereinstimmung zwischen Pixelposition und Synchronisationsinformation vorliegt. Hierzu wird die aktive Komponente Delay-

ImageCtrlBus verwendet. Sie verzögert die eingehende Synchronisationsinformation im gleichen Maße, in dem das Modul die Pixeldaten verzögert. Hiervon sind nur diejenigen Synchronisationssignale betroffen, die von den nachfolgenden Komponenten tatsächlich benötigt werden.

10.2.4 Ausgleich unterschiedlicher Pfadverzögerungen

Bei Funktionen, die zwei Bilder als Eingabeparameter erwarten und die korrespondierenden Pixel dieser Bilder verknüpfen, müssen Maßnahmen ergriffen werden, die sicherstellen, daß die entsprechenden Pixel beider Bilder der Funktion gleichzeitig bereitgestellt werden (Bild 10.4).

Wie bereits in den Anforderungen festgelegt, ist eine Videokamera der primäre Bildlieferant. In der im Bild dargestellten Anordnung gelangt der von der Kamera erzeugte Bildstrom über zwei verschiedene Pfade zur Add-Komponente, wo die Bilder dann pixelweise addiert werden. Nun werden aber die Pixeldaten auf dem Pfad von der Kamera zum Port Add.P1 um 10 Zeiteinheiten und von der Kamera zu Port Add.P2 um 20 Zeiteinheiten verzögert, so daß die entsprechenden Pixel der beiden Bilder zeitversetzt bei der Add-Komponente ankommen. Da diese Problematik bei vielen Komponenten auftritt, ist es sinnvoll, eine allgemeine, wiederverwendbare Lösung zu erstellen.

Ein erster Ansatz wäre einen FIFO-Speicher vor den Port ADD.P1 zu schalten, der in der Lage ist, so viele Pixel zwischenzuspeichern, wie in der Differenz der Laufzeiten der beiden Pfade übertragen werden können. Obwohl dies zu einer funktionsfähigen Lösung führt, kann diese, falls große Laufzeitunterschiede auftreten, einen recht hohen Bedarf an Implementierungsressourcen verursachen.

Eine in bezug auf die Implementierungskosten günstigere Lösung macht sich die, in den Anforderungen vorgegebene Eigenschaft zu nutze, daß jede Komponente in jedem Verarbeitungsschritt eine neue Pixelinformation bereitstellt. In diesem Fall kann der FIFO-Speicher durch ein Schieberegister ersetzt werden, das mit wesentlich geringerem Bedarf an Implementierungsressourcen verwirklicht werden kann.

Um diesen Lösungsansatz allen Komponenten bereitstellen zu können, wird der ImageSync-Portadapter so erweitert, daß er in der Lage ist, die ankommenden Daten zusätzlich zu verzögern. Die Komponente gibt dazu an, für welche Ports ein Laufzeitausgleich erfolgen soll, wobei eine beliebige Anzahl abhängiger Ports

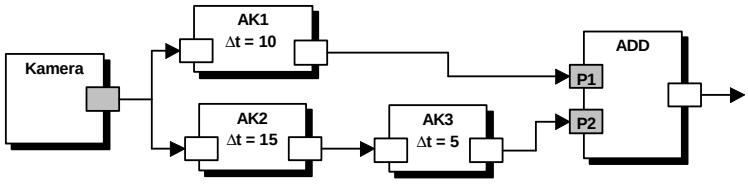

Bild 10.4. Abweichende Verzögerungszeiten auf den Datenpfaden

unterstützt wird. Aus dieser Information kann dann der Portadapter bestimmen, in welchem Pfad und in welchem Umfang Verzögerungen zu integrieren sind.

Um die Bestimmung der Laufzeit der Daten von der Kamera zu einem gegebenen Port bestimmen zu können, muß die Verzögerungszeit zwischen ankommenden und abgehenden Daten für jede Komponente bzw. Funktion bekannt sein.

Um die Laufzeit der Daten zu einem Eingangsport bestimmen zu können, ermittelt man zu erst den verbundenen Ausgangsport und erfragt für diesen die Laufzeit. Die Verzögerungszeit eines Ausgangsports setzt sich aus der Verzögerungszeit der Funktion und der Laufzeit der Daten von der Kamera zu den Eingangsports der Funktion zusammen. Verfügt eine Funktion über mehrere Eingangsports, wird das Maximum der Laufzeiten aller Eingangsports der Funktion verwendet. Mit Erreichen der Kamera ist die Berechnung abgeschlossen.

10.2.5 Verfügbare Komponenten

Die verfügbaren Komponenten sind in Bild 10.5 dargestellt. Alle Komponenten mit mehr als zwei Eingängen unterstützen das im vorherigen Abschnitt vorgestellte Verfahren zur Anpassung der Pfadverzögerung.

10.2.5.1 Camera

Die Camera-Komponente basiert auf einem Modul, das in der Lage ist, verschiedene Kameratypen anzusteuern. Es erzeugt aus den Pixeldaten und der Synchronisationsinformation einen gemeinsamen Datenstrom, der dem Porttyp ImageSync entspricht. Zusätzlich sind für verschiedene Kameras Module vorhanden, die die Konfiguration des jeweiligen Kameratyps ermöglichen. Hier können Werte wie z.B. das zu wählende Bildformat, Helligkeit und Kontrast eingestellt werden. Diese Konfigurationsmodule stellen aber lediglich die zur Konfiguration notwendige Schnittstelle bereit, führen diese aber nicht selbst durch. Beide Module sind für die Implementierung auf FPGAs ausgelegt.

Die Camera-Komponente muß neben der Implementierung noch dafür sorgen, daß für den Port, an dem sie die Bildinformation bereitstellt, der Datentyp so festgelegt wird, daß dieser dem von der Kamera gelieferten Bild entspricht. Hierzu müssen sowohl die durch den Benutzer gesetzten Attribute, die Parameter wie Bildgröße und Pixeltiefe beeinflussen, als auch die Eigenschaften der zugrundeliegenden Kamera beachtet werden.

Der so berechnete Datentyp wird dann dem Ausgangsport der Camera-Komponente zugewiesen. Wie in den später aufgeführten Beispielen gezeigt wird, entbindet dies den Anwender in den meisten Fällen davon, für die verwendeten Komponenten explizite Angaben über das zu verwendende Bildformat zu machen. Dies beruht darauf, daß an die Kamera angeschlossene Komponenten den bereitgestellten Datentyp übernehmen, den Typ ihrer Ausgansports entsprechend festlegen und somit die Information über das Bildformat an die verbundenen Komponenten weiterreichen.

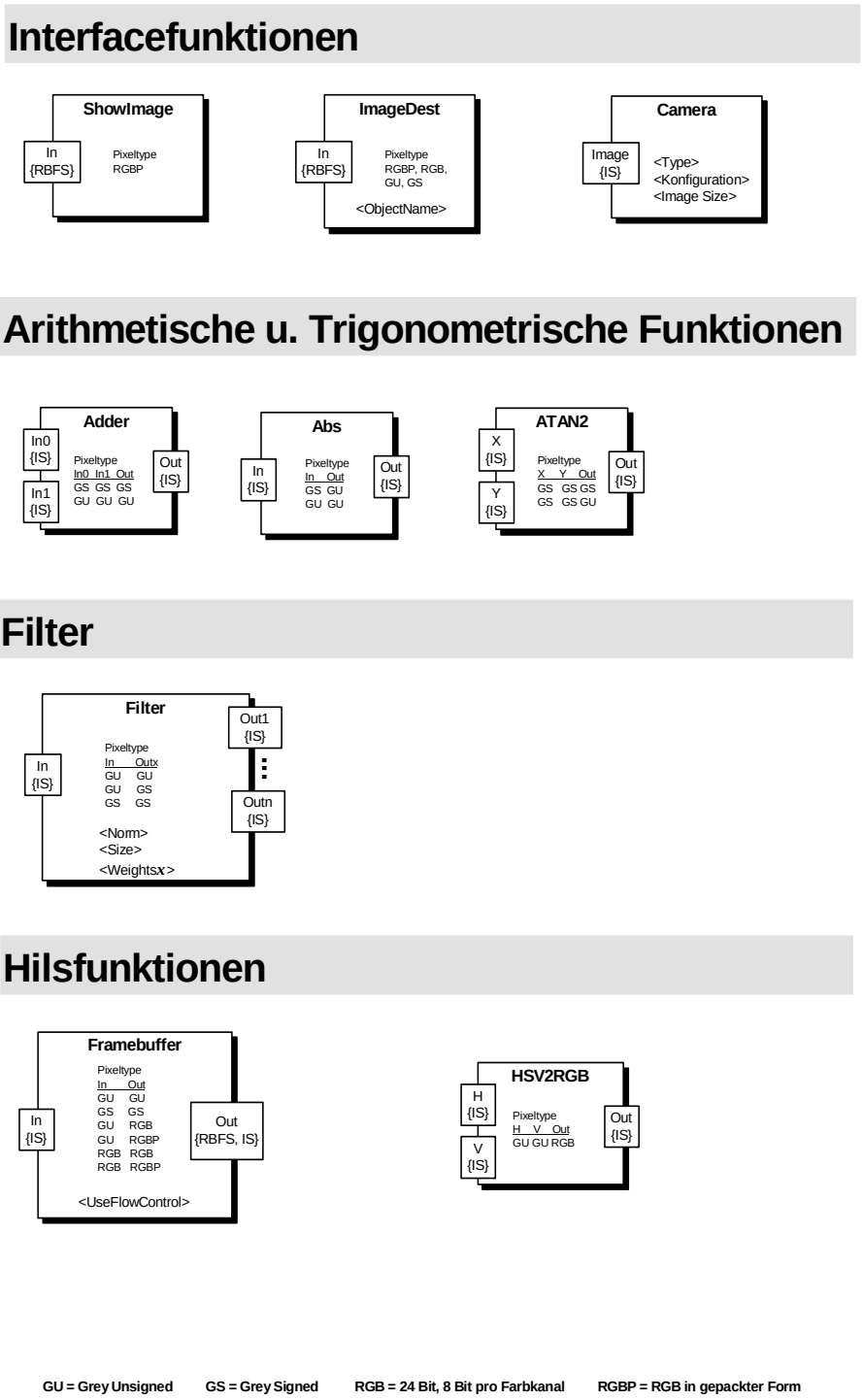

Bild 10.5. Verfügbare Komponenten

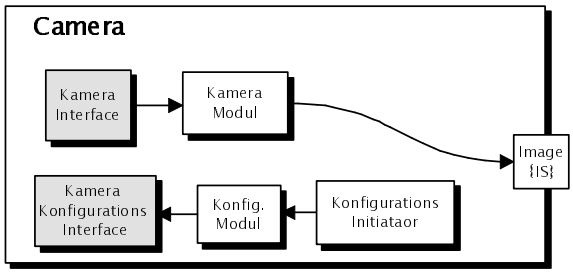

Bild 10.6. Aufbau der Camera-Komponente

Bei der Implementierung müssen verschiedene Teile betrachtet werden. Das Camera-Modul realisiert zwar die Ansteuerlogik, muß aber noch mit der entsprechenden Schnittstelle der Kamera verbunden werden. Gleiches gilt auch für das Konfigurationsmodul, wobei hier noch eine Implementierung erzeugt werden muß, die die Konfigurationsdaten zum Konfigurationsmodul überträgt und somit die eigentliche Initialisierung durchführt. Dies wird mit der in Bild 10.6 dargestellten Anordnung erreicht.

Die bereitgestellten Module müssen hierbei als aktive Komponenten gekapselt sein. Die grau hinterlegten Komponenten sind so ausgelegt, daß sie nur auf Systemeinheiten (hier meistens Steckverbinder) plaziert werden können, die mit den entsprechenden Schnittstellen der Kamera verbunden sind, wodurch die Verbindung zur Kamera für beliebige Zielsysteme hergestellt werden kann. Der Konfigurations-Initiator führt die Konfiguration der Kamera durch, in dem er die Konfigurationsdaten über das Konfigurationsmodul überträgt.

Oft ist der Konfigurationsvorgang etwas aufwendiger, so daß es sinnvoll ist, die Konfigurationskomponente so auszulegen, daß sie als Software realisiert werden kann. Dies vermeidet unnötig hohe Implementierungskosten.

10.2.5.2 ShowImage und ImageDest

ShowImage und ImageDest sind beides Komponenten, die lediglich als Software implementiert werden können.

ShowImage ist zusätzlich auf Ausführungseinheiten beschränkt, die in der Lage sind, Bilder zu visualisieren. Sie kann auf einem PC implementiert werden und zeigt dort die über den In-Port angelieferte Bildinformation als Bildfolge an. Die Bilder müssen in Form eines Datenblocks übertragen werden, weshalb der Porttyp ReadBlockFunctionSync verwendet wird.

ImageDest verhält sich ähnlich, nur daß hier die Bildinformation nicht angezeigt, sondern nur ein Objekt erzeugt wird, das den Zugriff auf die Bilddaten erlaubt. Da der Name dieses Objektes durch das ObjectName Attribut festgelegt werden kann, können somit Implementierungen, die nicht als Komponenten verfügbar sind, hinzugelinkt werden und können über das erzeugte Objekt die Bilddaten weiterverarbeiten.

10.2.5.3 Adder, ABS und ATAN2

Addition und Betragsbildung werden durch die Komponenten Adder und ABS realisiert und sind für die Implementierung auf FPGAs vorgesehen.

ATAN2 berechnet die Funktion $\arctan(y/x)$ und kann auf FPGAs abgebildet werden. Das Ergebnis wird im Bogenmaß angegeben und liegt zwischen $-\pi$ und π. Für $x = 0$ ist das Ergebnis $-\pi$ oder π, je nach dem, ob y positiv oder negativ ist.

Die Komponenten nehmen ganze Grauwertbilder entgegen, wobei die Bildgröße an allen Ports identisch sein muß. In einigen Fällen, z.B. ABS und ATAN2, ist es allerdings erlaubt, daß sich das Eingangs- und Ausgangsbild darin unterscheiden, daß in einem Fall ein vorzeichenbehafteter Pixeltyp und im anderen Fall ein vorzeichenloser Pixeltyp verwendet wird. Die möglichen Kombinationen sind in Bild 10.5 als Tabelle in der Komponentendarstellung enthalten.

Der Addierer unterstützt beliebige Pixelbreiten für die Summanden und sorgt zusätzlich dafür, daß die Pixelbreite für das Ergebnis so gewählt wird, daß kein Verlust an Genauigkeit auftritt. Dies ist notwendig, da das Ergebnis einer Addition einen größeren Wertebereich als die Summanden hat. Bei der Addition von einem n-Bit breiten Pixel mit einem m-Bit breiten Pixel ist für die Darstellung des Ergebnisses ein Pixeltyp der Breite $\max(n, m) + 1$ notwendig.

Beim Betragsbilder wird aus einem Bild mit vorzeichenbehafteten Pixeln ein Bild mit vorzeichenlosen Pixeln erzeugt. Tritt der Fall ein, daß schon am Eingang vorzeichenlose Pixel vorliegen, wird der Betragsbilder dadurch realisiert, daß der In-Port mit dem Out-Port verbunden wird und somit keine Implementierungsressourcen belegt werden.

Die ATAN2 -Komponente erwartet zwar an den Eingängen Bilder mit vorzeichenbehafteten Pixeln, kann aber als Ergebnis vorzeichenlose bzw. vorzeichenbehaftete Werte liefern. Erwähnenswert ist dennoch, daß zur Realisierung der ATAN2 Funktion auf einem FPGA nur ein Modul existiert, das 8-Bit Pixeldaten unterstützt. Die Komponente kann aber durch die bereitgestellten Portadapter dennoch beliebige Pixelbreiten unterstützen.[1]

10.2.5.4 Filter

Die Filter-Komponente verfügt ebenfalls über eine Implementierungsvariante für FPGAs und unterstützt Grauwertbilder und Filtermasken beliebiger Größe, wobei diese durch das Attribut <Size> anzugeben ist.

Die Bildtypen am Eingang und den Ausgängen des Filters sind weitgehend identisch. Für vorzeichenlose Eingangsdaten können allerdings sowohl vorzeichenlose als auch vorzeichenbehaftete Ergebnisse erzeugt werden.

Die Bildgröße und die Pixelbreite bleiben erhalten, wobei gegebenenfalls eine automatische Normalisierung der Ergebnispixelwerte durchgeführt wird. Wird

[1] Bei höheren Pixelbreiten führt dies natürlich zu Rundungsfehlern und somit zu einer nicht optimalen Implementierung.

allerdings vom Benutzer das Attribut <Norm> benutzt, wird der hier angegebene Wert als Normalisierungsfaktor verwendet.

Die Filter-Komponente unterstützt die Möglichkeit, gleichzeitig mehrere Filter mit gleichgroßer Filtermaske auf ein Bild anzuwenden. Der Anwender muß hierzu lediglich für jeden Filter die benötigte Filtermaske über die Parameter <Weights$x$> bereitstellen. Die Ergebnisse der verschiedenen Filteroperationen werden dann an unterschiedlichen Ports bereitgestellt.

10.2.5.5 Framebuffer und HSV2RGB

Die Framebuffer-Komponente nimmt ein Bild aus einem Bildstrom entgegen und speichert dieses. Über den Out-Port kann die gespeicherte Information als Datenblock (ReadBlockFuncSync) und als Bild (ImageSync) ausgelesen werden, wobei die Verfügbarkeit neuer Bilddaten explizit signalisiert wird. Über das Attribut UseFlowControl kann festgelegt werden, daß erst mit dem Schreiben neuer Bilddaten begonnen werden darf, wenn die alten Daten vollständig gelesen wurden. Während die Bildgröße erhalten bleibt, kann die Framebuffer-Komponente Konvertierungen zwischen verschiedenen Pixelformaten durchführen (siehe Tabelle in der Komponentendarstellung Bild 10.5)

Die Komponente HSV2RGB nimmt zwei Grauwertbilder gleicher Größe entgegen, interpretiert diese als die H und V Kanäle des HSV-Farbraums und erzeugt daraus ein Bild gleicher Größe, dessen Pixel aber als Farbdarstellung das RGB Format verwendet. Für den S-Kanal wird der größtmögliche positive Wert angenommen.

Das für die Realisierung auf FPGAs zugrundeliegende Modul unterstützt an den Eingängen nur 8-Bit Pixel. Für die Komponente wird diese Beschränkung dadurch aufgehoben, daß die eventuell notwendige Bitbreitenanpassung automatisch durch die vorhandenen Portadapter durchgeführt wird.

Beide Komponenten können auf FPGAs implementiert werden.

10.3 Erstellung der Lösungsbeschreibung

Um die Erstellung der Lösungsbeschreibung zu erleichtern, wurde das vorgestellte Programmiersystem um eine Möglichkeit zur grafischen Eingabe ergänzt (Bild 10.7). Der Benutzer kann Komponenten aus einer Bibliothek auswählen, die entsprechenden Attributwerte festlegen und die Komponenten untereinander verbinden. Nach der Erstellung der Lösungsbeschreibung wird der Übersetzungsprozeß angestoßen, wobei für die Erstellung der ausführbaren Implementierung auch die benötigten externen Compiler für C++ und CHDL automatisch aufgerufen werden.

Obwohl diese Umgebung viele Möglichkeiten des zugrundeliegenden Programmiersystems wie die Erstellung neuer Komponenten und die Verwaltung von Accessports nicht unterstützt, lassen sich doch mit den vorhandenen Komponenten

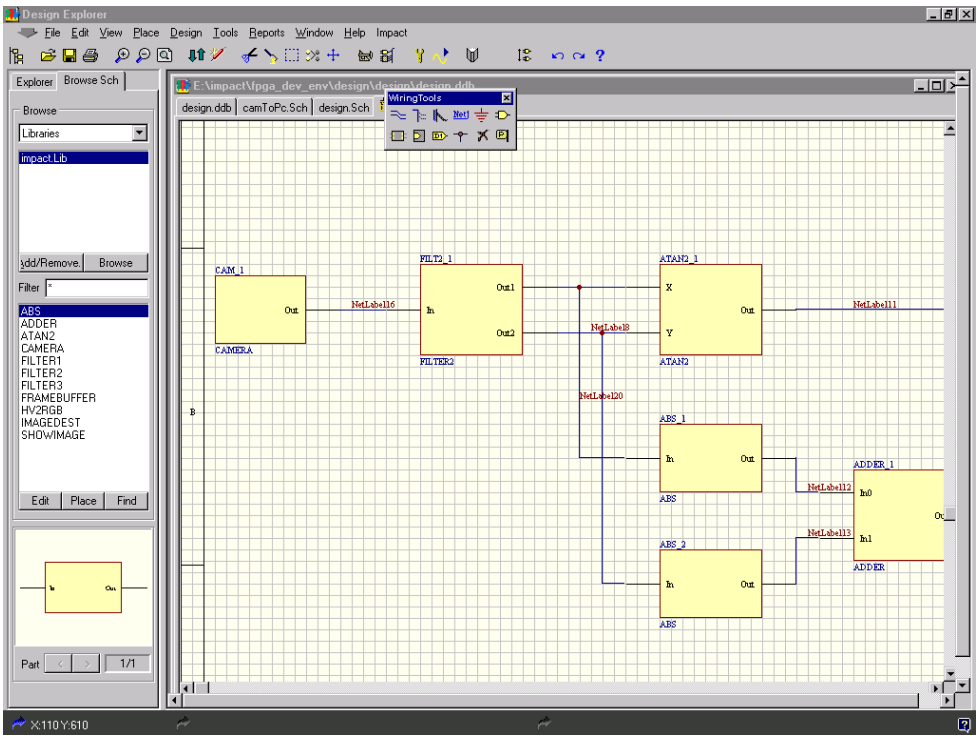

Bild 10.7. Grafische Eingabe der Lösungsbeschreibung

einige nützliche Implementierungen erstellen, die im folgenden Abschnitt näher erläutert werden sollen.

10.4 Realisierte Anwendungen

Im Folgenden werden die, mit dem zuvor vorgestellten Programmiersystem realisierten Anwendungen vorgestellt und ihre Leistungsfähigkeit mit Implementierungen verglichen, die auf konventionelle Weise erstellt wurden.

10.4.1 Frame Grabber

Ein Frame Grabber realisiert die grundlegenden Aufgaben eines Bildverarbeitungssystems. Er nimmt die Bilder einer Kamera entgegen, legt diese in einem Zwischenspeicher ab und überträgt sie dann zur Visualisierungseinheit. Eine Beschreibung dieser Funktionalität mit den verfügbaren Komponenten zeigt Bild 10.8.

Die über die Camera-Komponente bereitgestellten Bilder werden durch die Komponente Framebuffer zwischengespeichert und mit der ShowImage-Komponente angezeigt. Die Attribute der Camera-Komponente definieren den zu ver-

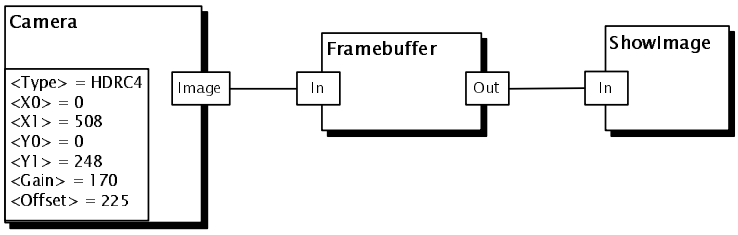

Bild 10.8. Beschreibung eines Frame Grabbers

wendenden Kameratyp, die gewünschte Bildgröße und die einzustellenden Werte für Kontrast und Helligkeit (Gain, Offset).

Die in diesem und in den folgenden Beispielen benutzte Kamera verfügt über einen CMOS-Sensor und ermöglicht es, einen beliebigen Ausschnitt aus dem Sensorbild zu definieren, der dann am Ausgang der Kamera bereitgestellt wird. Die Bildinformation besteht aus 10-Bit breiten vorzeichenlosen Grauwertbildpunkten. Für die Konfiguration der Kamera stellt diese eine eigene Schnittstelle bereit.

Die Camera- und Framebuffer-Komponente werden auf dem FPGA realisiert, die ShowImage-Komponente auf dem PC. Bei der Erstellung der Implementierung führen die Komponenten einige automatische Optimierungs- bzw. Anpassungsschritte durch.

Alle Komponenten bestimmen den zu transportierenden Bildtyp automatisch. Die Camera-Komponente stellt den Bildtyp bereit, führt über die Portadapter eine Synchronisierung der Information von dem auf der Kamera verwendeten Takt, auf den Takt des FPGAs durch und erzeugt die für die Konfiguration der Kamera notwendigen Komponenten.

Der Konfigurations-Initiator (siehe Abschnitt 10.2.5.1) ist hierbei als Software ausgelegt und wird auf dem PC implementiert. Die Framebuffer-Komponente wählt einen geeigneten Speicherbaustein aus, erzeugt die benötigten Adresssignale und implementiert die Ansteuerlogik. Da die ShowImage Komponente nur Bilder im gepackten RGB Format unterstützt, wird zusätzlich eine Wandlung zwischen Grauwertdarstellung und RGBP Format von der Framebuffer-Komponente durchgeführt.

Mit Hilfe der Zielsystembeschreibung wird ein DMA-Transfer für die Übertragung der Bilddaten eingerichtet. Für die Ansteuerung des Interfaces zur Kamerainitialisierung wird ein Speicherbereich im PCI-Adressraum reserviert, die Ansteuersoftware erzeugt. Auf dem FPGA werden die Konfigurationsdaten anhand der beim PCI-Buszugriff benutzten Adresse extrahiert und an das Konfigurationsmodul weitergereicht.

In der hier verwendeten Konstellation überträgt die Kamara die Bilder mit einer Rate von 60Hz, die auch alle vollständig in Echtzeit angezeigt werden.

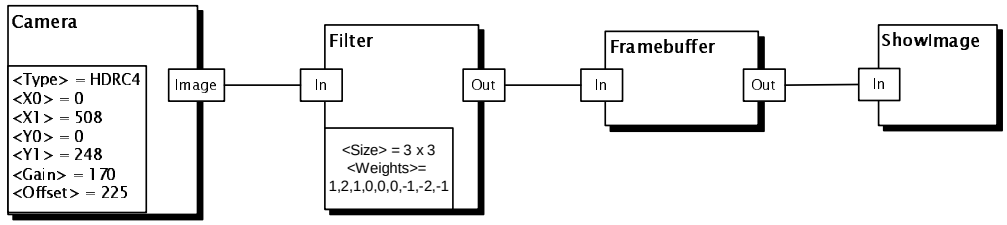

Bild 10.9. Programm zur Erzeugung eines Kantenbildes

10.4.2 Kantenbild

Die in Bild 10.9 dargestellte Beschreibung fügt zum Frame Grabber Entwurf noch einen Filter hinzu. Die Filtermaske realisiert einen Sobel-X Filter[23] , der im Ergebnisbild horizontale Kanten besonders hervorhebt.

Die durchgeführten Optimierungen und Anpassungsschritte entsprechen denen des Frame Grabber Entwurfs, wobei der Filter noch die vom Framebuffer benutzten Steuersignale soweit verzögern muß, daß sie die richtigen Pixel im Ergebnisbild kennzeichnen. Von der Framebuffer-Komponente wird das Zeilenende-Signal bei der Konvertierung vom RGB- ins RGBP-Format und das Bildende-Signal zur Unterscheidung von aufeinanderfolgenden Bildern benötigt.

Auch hier werden die von der Kamera gelieferten Bilder in Echtzeit verarbeitet und angezeigt.

10.4.3 Lokale Orientierung

Die Lokale Orientierung erzeugt ein Gradientenbild, wobei jeder Bildpunkt die Richtung und den Betrag der Änderung der Grauwertpunkte des Ausgangsbildes anzeigt. Um diese Information anschaulich darzustellen, kann die Richtung des Gradienten durch eine Farbe und der Betrag durch die Intensität des Bildpunktes im Ergebnisbild dargestellt werden. Zur Berechnung dieser Funktion sind folgende Schritte durchzuführen:

1. Erzeuge zwei Kantenbilder für Kanten in x- bzw. y-Richtung

$$g(x,y) \mapsto (dx,dy) = (\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y})$$

2. Berechne die Richtung und den Betrag des Vektorbildes

$$\begin{aligned} \varphi &= \arctan \frac{dy}{dx} \\ l &= \sqrt{dx^2 + dy^2} \end{aligned}$$

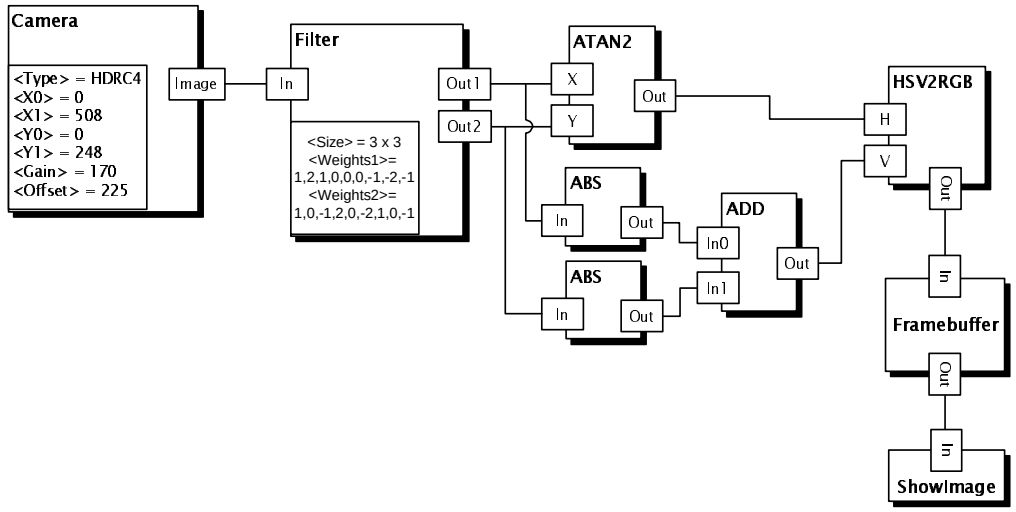

Bild 10.10. Beschreibung der Lokalen Orientierung

3. Stelle das Vektor Bild im HSV-Bildraum dar

$$\begin{aligned} H &= \varphi \\ S &= \infty \\ V &= l \end{aligned}$$

Die Umsetzung dieses Algorithmus ist in Bild 10.10 dargestellt. Die benötigten Kantenbilder werden durch den Filter bereitgestellt. Statt der Betrag des Vektors wird die Summe der Beträge der Vektorkomponenten gebildet, was zwar mathematisch nicht exakt ist, aber für Visualisierungszwecke vollständig ausreicht.

Auch hier müssen die benötigten Steuersignale von der Kamera bis hin zum Framebuffer durchgeschleift und entsprechend verzögert werden. Zusätzlich müssen mehrere Pixelbreitenapassungen durchgeführt werden. Für die Eingänge von ATAN2 von 10-Bit auf 8-Bit und am V-Eingang von HSV2RGB von 11-Bit auf 8-Bit. Auch diese Anpassungsschritte werden automatisch durchgeführt.

Auch hier läuft die Verarbeitung der Bilder in Echtzeit bei voller Framerate ab.

10.4.4 Leistungsmerkmale und Bewertung

Um die Leistungsfähigkeit des vorgestellten Beschreibungsverfahrens abschätzen zu können, müssen zwei Faktoren berücksichtigt werden. Dies betrifft den Aufwand zur Erstellung der Lösungsbeschreibung und die Leistungsfähigkeit der resultierenden Implementierung in bezug auf Verarbeitungsleistung und Ressourcenbedarf. Die mittels aktiver Komponenten erstellten Problemlösungen sollen

mit Entwürfen verglichen werden, die direkt mit C++ und CHDL entwickelt wurden.

10.4.4.1 Lösungsbeschreibung

Zum Vergleich der Beschreibungsverfahren soll die Lösungsbeschreibung der Lokalen Orientierung herangezogen werden. In der Variante mit aktiven Komponenten werden jeweils neun Komponenten und Verbindungsknoten benötigt. Bei einer Beschreibung in CHDL/C++ sind hingegen ca. 200 Code-Zeilen notwendig, um die gleiche Funktionalität zu beschreiben. Zusätzlich muß der Entwickler dabei über fundierte Kenntnisse der Zielplattform und der Programmierung von FPGAs verfügen. Auch ist die Übertragung der Beschreibung auf ein anderes Zielsystem mit großem Aufwand verbunden. Die Lösungsbeschreibung fällt mit aktiven Komponenten deutlich kürzer aus, erfolgt auf höherer Abstraktionsebene und ist leicht auf andere Zielsysteme portierbar, wodurch der Entwurfsaufwand deutlich Reduziert wird. Dadurch, daß sich Komponenten automatisch auf den Kontext anpassen, in dem sie verwendet werden und eventuelle Fehlersituationen erkennen, wird die Entwicklungszeit weiter verkürzt.

Obwohl die hier vorgestellte Implementierung des Programmiersystems nicht auf hohe Übersetzungsleistung ausgelegt ist, wird die Übersetzungszeit von der Beschreibung bis hin zur fertigen Implementierung nur unwesentlich um ca. 15% verlängert, so daß hieraus keine besondere Beeinträchtigung entsteht.

10.4.4.2 Leistungsdaten der resultierenden Implementierungen

Die Leistungsdaten[2] der resultierenden Implementierungen bei Verwendung von aktiven Komponenten bzw. der Programmierung mit CHDL/C++[3] sind in Tabelle 10.1 dargestellt.

Hierbei wird nur der Teil der Applikation betrachtet, der auf FPGAs implementiert ist, da er zum einen das Leistungsbestimmende Element darstellt und sich zusätzlich für die, auf dem PC realisierten Teile, bei beiden Ansätzen identische Leistungsdaten ergeben.

Zur Bestimmung der Implementierungskosten wird die Anzahl der auf dem FPGA belegten CLBS herangezogen. Die Beurteilung der Verarbeitungsleistung erfolgt über den maximalen Takt, mit dem die Implementierung betrieben werden kann. Da bei allen hier vorgestellten Implementierungen in jedem Verarbeitungsschritt ein Datum bearbeitet wird, entspricht der in der Spalte Takt angegebene Wert dem maximal erreichbaren Datendurchsatz.

---

[2] Der hier angegebene Takt ist der von den Herstellerwerkzeugen des FPGA-Herstellers angegebene Maximalwert, der erreichbar ist, wenn die Verbindungen auf dem FPGA ideal verlegbar wären, also nur minimale Verzögerungszeiten verursachen würden. Dieses Vorgehen wurde gewählt, da die Abbildungssoftware bei identischen Ausgangsdaten bei mehreren Durchläufen verschiedene Ergebnisse liefert und auch eine vom FPGA-Typ unabhängige Bewertung durchgeführt werden soll.

[3] Die mit CHDL/C++ erstellten Vergleichimplementierungen wurden dankender Weise von Herrn Dipl. Informatiker Peter Dillinger zur Verfügung gestellt.

	Aktive Komponenten		Konvent. Entwurf	
	Kosten	**Takt**	**Kosten**	**Takt**
Frame Grabber	297 CLBS	76 MHZ	295 CLBS	52 MHz
Kantenbild	846 CLBS	51 MHz	842 CLBS	45 MHz
Lokale Orientierung	2841 CLBS	39 MHz	2679 CLBS	39 MHz

Tabelle 10.1. Vergleich der Leistungdaten der Implementierungen

Wie aus der Tabelle zu entnehmen ist, wird in allen Beispielen mit aktiven Komponenten ein Implementierungsergebnis erzielt, das zumindest den gleichen Takt und damit auch den gleichen Datendurchsatz wie die entsprechenden konventionellen Entwürfe aufweist. Im schlechtesten Fall werden hierbei um ca. 6% höhere Implementierungskosten verursacht.

Besonders beim Frame Grabber-Entwurf fällt auf, daß die Verarbeitungsleistung bei der Verwendung von aktiven Komponenten deutlich höher liegt als beim konventionellen Entwurf. Dies beruht darauf, daß die verwendeten aktiven Komponenten gezielt Registerstufen zu den verwendeten Implementierungsmodulen hinzufügen, um eine möglichst hohe Verarbeitunsgeschwindigkeit zu erreichen. Ein konventioneller Entwurf hingegen, wird üblicher Weise nur so weit optimiert, bis der für die Anwendung erforderliche Datendurchsatz gerade erreicht wird.

Dieser Effekt nimmt bei den vorgestellten Beispielen mit zunehmender Komplexität der Entwürfe ab. Das beruht darauf, daß in diesen Fällen der Verarbeitungstakt durch die maximale Betriebsfrequenz der Implementierungsmodule begrenzt wird. Da in beiden Ansätzen die gleichen Implementierungsmodule verwendet werden, erfolgt somit eine Angleichung der Verarbeitungsleistung.

Bei der lokalen Orientierung begrenzt z.B. die Implementierung des ATAN2-Moduls den Betriebstakt auf max. 39 MHz. Um hier eine Verbesserung zu erzielen, müßte die ATAN2-Komponente um eine leistungsfähigere Implementierung ergänzt werden.

Zusammenfassend läßt sich sagen, daß mit beiden Ansätzen gleichwertige Implementierungsergebnisse erzielt werden können. Der etwas höhere Ressourcenbedarf bei der Verwendung von aktiver Komponenten fällt nur unwesentlich ins Gewicht. Gerade in Hinblick auf die stetig wachsende Komplexität von FPGAs wird dieser Aspekt weiter an Bedeutung verlieren.

11
Bewertung und Ausblick

Ziel der vorliegenden Arbeit ist es, die Programmierung von FPGA-Prozessorsystemen zu vereinfachen. Die hier betrachteten Systeme bestehen aus konfigurierbarer Hardware und einem Mikroprozessor-basierten Teil.

Es sollte ein Verfahren entwickelt werden, das es ermöglicht, Problemlösungen auf möglichst hoher Abstraktionsebene zu beschreiben. Hierzu müssen Schritte vorgesehen werden, die eine Abbildung dieser Beschreibung auf das Gesamtsystem ermöglichen.

Die Wahl des Zielsystems soll möglichst frei erfolgen können, ohne dabei die Lösungsbeschreibung anpassen zu müssen. Die resultierende Implementierung soll das Gesamtsystem in einer Weise nutzen, das eine möglichst hohe Leistungsfähigkeit erzielt wird. Hierbei ist es besonders wichtig, die Architekturmerkmale des Zielsystems und die Ausführungseigenschaften der einzelnen Systembestandteile auszunutzen und geeignete Optimierungsschritte durchzuführen.

Im Rahmen dieser Arbeit wurden aktive Komponenten eingeführt und ein darauf basierendes Beschreibungsverfahren vorgestellt. Aktive Komponenten repräsentieren Funktionen. Sie sind zusätzlich in der Lage, an Hand der gegebenen Randbedingungen eine geeignete Implementierung für diese Funktion zu wählen.

Im folgenden sollen die Beschreibungs- und Implementierungseigenschaften von aktiven Komponenten und die Eigenschaften des vorgestellten Abbildungsprozesses an Hand der oben aufgeführten Kriterien bewertet werden.

11.1 Beschreibungseigenschaften aktiver Komponenten

Die Erstellung der Lösungsbeschreibung mittels aktiver Komponenten erfolgt dadurch, daß geeignete Komponenten anhand der von ihnen bereitgestellten Funk-

tionen ausgewählt und in geeigneter Form über ihre Schnittstellen miteinander verbunden werden. Schnittstellen transportieren die von der Funktion benötigten bzw. erzeugten Daten.

Da eine Komponente beliebige Funktionen definieren kann und auch der Typ der Daten, die über die Schnittstellen transportiert werden, je nach Bedarf festgelegt werden kann, ist eine sehr gut an den Aufgabenbereich angepaßte Beschreibung möglich. Bei geeigneter Wahl der Funktionen und Datentypen ist somit die Ebene der domänenspezifischen Beschreibung für beliebige Anwendungsbereiche erreichbar und ermöglicht somit auch domänenspezifische Optimierungen.

Die Schnittstellen der Komponenten setzen sich aus Ports zusammen. Jedem Port ist ein Porttyp zugewiesen, der den Typ der zu übertragenden Daten und das benutzten Kommunikationsverfahren beschreibt. Um eine möglichst einfache Beschreibung zu ermöglichen, kann der Porttyp aber auch unter- bzw. unspezifiziert bleiben. Während des Umsetzungsprozesses wird dann automatisch ein geeigneter, vollständig definierter Porttyp gewählt.

Die Verbindung von Datentyp und Kommunikationsmechanismus ermöglicht weitreichende Überprüfungen, ob Verbindungen zwischen verschiedenen Ports zu einer funktionsfähigen Implementierung führen können. Während bei aktiven Komponenten eine unsachgemäße Verwendung der Schnittstellen auch auf Protokollebene erkannt und entsprechend behandelt werden kann, ist dies bei herkömmlichen HDLs nicht der Fall, da hier Verbindungen nur über den verwendeten Datentyp spezifiziert sind.

Nachteilig an diesem Verfahren ist, daß für jede Anwendungsdomäne ein eigener Satz von aktiven Komponenten und Daten- bzw. Porttypen zu entwerfen ist, um eine entsprechend hohe Abstraktionsebene zu erreichen. Da die Definition der Komponente als solches nicht sonderlich aufwendig ist, besteht das Hauptproblem darin, das Konfigurationswissen der Komponenten zu erstellen. Das Konfigurationswissen umfaßt die Implementierungsvarianten und Auswahlkriterien, die es der Komponente ermöglichen, eine geeignete Implementierung zu wählen. Dies wird in Abschnitt11.2 näher behandelt.

Während die Definition neuer Datentypen meist unumgänglich ist, wird die Porttypendefinition dadurch vereinfacht, daß Kommunikationsverfahren oft domänenübergreifend verwendet werden können. Da Porttypen und damit auch Ports hierarchisch aufgebaut sind, können neue Porttypen auch aus bereits bestehenden zusammengesetzt werden.

Da aber bei der jetzigen Implementierung des Programmiersystems für jede Komponente, Port- oder Datentyp eine eigene Java-Klasse zu erzeugen ist, sollte das System dahingehend weiterentwickelt werden, daß hierfür eine eigene, besser geeignete Beschreibungsmöglichkeit vorgesehen wird.

Zusätzlich muß die grafische Benutzeroberfläche so erweitert werden, daß sie den vollen Funktionsumfang des zugrundeliegenden Programmiersystems unterstützt.

11.2 Implementierungseigenschaften aktiver Komponenten

Jede Komponente verfügt über ein Konfigurationswissen. Dieses umfaßt die möglichen Implementierungsvarianten und Kriterien, die festlegen, unter welchen Randbedingungen die jeweilige Implementierung die beste Alternative darstellt. Aktive Komponenten ermöglichen somit nicht nur die Wiederverwendbarkeit von Implementierungen, sondern auch der Implementierungsentscheidungen, die zu der jeweiligen Implementierung geführt haben.

Da eine Komponente mehrere Implementierungsvarianten unterstützt, ist ihre Wiederverwendbarkeit deutlich höher als die der zugrundeliegenden Einzelimplementierungen. Zusätzlich können andere oder weniger erfahrene Entwickler von den Kenntnissen anderer Entwickler profitieren, indem sie deren Komponenten verwenden. Somit sind auch unerfahrene Entwickler in der Lage, leistungsfähige Problemlösungen auf FPGA-Prozessorsystemen zu erstellen, wodurch der Anwenderkreis deutlich erhöht werden kann.

Problematisch ist hierbei, daß für jede Komponente das geeignete Konfigurationswissen bereitzustellen ist. Dieser Vorgang kann recht aufwendig sein. Gemildert wird dies dadurch, daß Komponenten hierarchisch beschrieben werden können. Komponenten können somit ihr Konfigurationswissen über die Verkettung anderer Komponenten beschreiben. Zusätzlich ermöglichen automatisch eingefügte Portadapter die Erstellung von flexiblen Komponenten. Portadapter sind in der Lage, zwischen verschiedenen Port- und Datentypen zu adaptieren. Das Kommunikationsverhalten von Komponenten kann hierdurch deutlich flexibler gestaltet werden.

Dessen ungeachtet ist es notwendig, die Beschreibung des Konfigurationswissen zu vereinfachen, indem entsprechende Beschreibungsmittel bereitgestellt werden. Besonderes Augenmerk sollte hierbei auf ein möglichst automatisches Einbinden bestehender Implementierungen gelegt werden. Im Softwarebereich wäre besonders die Unterstützung eines standardisierten Komponentenmodells interessant.

11.3 Systemabbildung

Ziel der Systemabbildung ist es, die Lösungsbeschreibung möglichst optimal auf ein gegebenes Zielsystem abzubilden. Hierzu müssen die Komponenten den Einheiten des Zielsystems zugeordnet, die Porttypen der Ports bestimmt und die Verbindungen zwischen den Komponenten auf dem Zielsystem realisiert werden. Die Porttypen werden dabei so gewählt, daß sie eine Implementierung auf dem Zielsystem erlauben und gleichzeitig die Leistungsanforderungen durch die Implementierung erfüllt werden. In diese Entscheidung werden sowohl die Eigenschaften des Zielsystems, als auch die Implementierungseigenschaften der Komponenten einbezogen.

Die Eigenschaften des Zielsystems sind dabei in der Zielsystembeschreibung zusammengefaßt. Hierzu gehören die vorhandenen Systemeinheiten, die möglichen

Verbindungspfade und die Vorgehensweise bei der Systeminitialisierung. Zusätzlich sind Beschreibungselemente enthalten, die die Kommunikation auch zwischen unterschiedlichen Einheiten wie FPGA und CPU unterstützen. Um Engpässe bei Systemressourcen zu vermeiden, wird das Resourcesharing unterstützt. Hierdurch können Systemeinheiten, die von sich aus nur eine Komponente aufnehmen können, dennoch von mehreren Komponenten gemeinsam genutzt werden.

Nachdem die Verbindungen verlegt wurden, wird die Implementierung der Komponenten durchgeführt und die Gesamtimplementierung einschließlich der notwendigen Konfiguration- und Initialisierungsfunktionen erzeugt.

Dieses Vorgehen ermöglicht eine hohe Portabilität und resultiert in leistungsfähigen Implementierungen. Die im Rahmen dieser Arbeit durchgeführten Beispielimplementierungen zeigen durchgängig die gleiche Leistungsfähigkeit wie Handentwürfe. Dies betrifft sowohl die Verarbeitungsleistung als auch den Ressourcenbedarf.

Verbesserungen sind dadurch möglich, daß weitere globale Optimierungsstrategien unterstützt werden, die dann die Randbedingungen für die Komponentenimplementierung genauer vorgeben. Dies wäre z.B. durch Optimierung der kritischen Pfade und Allocation/Scheduling Schritte möglich[1]. Zusätzlich muß die Implementierung der Plazierung- und Routing-Funktion erweitert werden. Das Plazieren muß weitestgehend automatisch erfolgen. Beim Routing müssen Schritte eingefügt werden, die bestehende Verbindungen auflösen und neu verlegen können. Auch müssen Möglichkeiten vorgesehen werden, Plazierungen zu verwerfen und neu zu gestalten, falls für eine gegebene Plazierung keine Implementierung möglich ist.

Die Programmierung mit aktiven Komponenten stellt eine interessante Alternative zu herkömmlichen Programmierverfahren dar und ermöglicht eine effiziente Programmierung von FPGA-Prozessorsystemen. Schon mit der hier vorgestellten rudimentären Implementierung eines solchen Programmiersystems konnten gute Ergebnisse erzielt werden. Obwohl hierbei nur ein einfach strukturiertes Zielsystem zur Implementierung genutzt wurden, ist bei der Verwendung eines weiter ausgebauten Programmiersystems auch auf komplexeren Zielsystemen mit ähnlichen Ergebnissen zu rechnen.

[1] Bisher wird Allocation und Scheduling nur auf der Implementierungsebene unterstützt, falls die Implementierungssprache dieses Vorgehen bereitstellt. Bei VHDL ist dieser Fall gegeben.

Literaturverzeichnis

[1] A. L. Abbot, P. M. Athanas, L. Chen, and R. L. Elliott. Finding lines and building pyramids with splash. *FPGAs for Custom Computing Machines Workshop*, pages 155–163, 1994.

[2] B. Allaire and B. Fischer. Block adaptive filter. *Xilinx Inc. and Application Notes*, 1997.

[3] W. W. Amstrong. Atlas - technical proposal for a general-purpose pp experiment at the large hadron collider at cern. Technical report, CERN/LHCC/94-43, 1994.

[4] R. V. R. M. Athanus and A. L. Abbot. High-speed region detection and labeling using an FPGA-based custom computing plattform. *5th International workshop on Field Programmable Logic Applications*, 1995.

[5] V. Betz. Directional bias and non-university in FPGA global routing architectures. *ICCAD 1996.*

[6] V. Betz. VPR user's manual. *University of Toronto*, 1996.

[7] C. J. Block. A model for representing ruby circuits. *Functional Programming, Glasgow 1996.*

[8] O. Bringmann and W. Rosenstiel. Resource sharing in hierarchical synthesis. Technical report, Universität Karlsruhe, 1997.

[9] Y. Brunel, A. Sangiovanni-Vincentelli, and R. Kress. COSY: A methodology for system design based on reusable hardware and software IP's. 1998.

[10] M. Chido and D. Engels. A case study in computer-aided co-design of embedded controllers. *Design Automation for Embedded Systems*, 1, 1996.

[11] C.-J. Chou, S. Mohanakrishnan, J. B, and Evans. FPGA implementation of digital filters. *ICSPAT'*, 1993.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison Wesley, 2000.

[13] P. Dillinger, S. Hezel, and H. Lauer. FPGAs zur Echtzeit-Bildverarbeitung mit 1D/2D-FIR-Filteroperationen. In *Image Processing and Machine Vision*, volume 1572, pages 213–218, Düsseldorf, 2000. VDI Berichte.

[14] Embedded Solutions Ltd. *Handel-C Language Reference Manual (Version 2.1)*.

[15] R. Englander. *Developing JAVA Beans*. O'Reilley, 1997.

[16] I. Foster. *Designing and Buliding Parallel Programs*. Addison Wesley, 1994.

[17] D. Galloway. *Transmogrifier*. University of Toronto, 1996.

[18] M. Grand. *JAVA Language Reference*. O'Reilley, 1997.

[19] M. Greenberg. HDL techniques for faster synthesized counters. Motorola Inc., 1997.

[20] M. Gschwind. Optimizing VHDL code for FPGA targets. *Technische Universität Wien, 1996*.

[21] S. Guccione. *List of FPGA-Based Computing Machines*. `http://www.io.com/~guccione/HW_list.html`, 2000.

[22] R. K. Gupta and G. D. Micheli. System synthesis via hardware-software codesign. Technical report, Computer System Laboratory, Stanford University, 1992.

[23] P. Haberäcker. *Digitale Bildverarbeitung*. Hanser Verlag, 1991.

[24] R. Harper. Introduction to standart ML. *Carnegie Mellon University, 1993*.

[25] S. Hauck. Multi-FPGA systems. *Dissertation, University of Washington, 1995*.

[26] T. Heath. Automating the compilation of software into hardware. *Dissertation, University of Oxford, 1993*.

[27] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[28] D. Herrmann, J.Henkel, and R.Ernst. An approach to the adaptation of estimated cost parameters in the COSYMA system. In *CODES '94*, 1994.

[29] S. Hezel and R. Männer. Schnelle Berechnung von 2D-FIR-Filteroperationen mittles FPGA-Koprozessor microEnable. In *21. DAGM-Symposium, Mustererkennung*, page 250pp. Springer Verlag, 1999.

[30] C. Iseli. Spider a reconfigurable processor develepment system. *Dissertation, Universität Lausanne*, 1996.

[31] A. A. Jerraya, H. Ding, P. Kission, and M. Rahmouni. *Behavioral Synthesis and Componet Reuse with VHDL.* Kluwer Academic Publisher, 1997.

[32] G. . Knudsen. *JAVA Fundamental Classes Reference.* O'Reilly, 1997.

[33] K. Kornmesser. *CHDL : Ein C++-Basierte Hardwarebescheribsprache für FPGAs und PLDs.* Universität Mannheim, 1998.

[34] Lehmann, Wunder, and Salz. *Schaltungsdesign mit VHDL.* Franzis Verlag, 1994.

[35] J. Ludvig. *Enable++: Ein Universeller FPGA-Triggerprozessor Für Das ATLAS-Experiment.* Dissertation. Universität Heidelberg, 1988.

[36] J. Madsen. LYCOS: The Lyngby Co-Systhesis System. *Design Automation for Embedded Systems, vol.2, nr.2*, 1997.

[37] R. Männer, M. Sessler, and H. Simmler. *Pattern Recognition and Reconstruction on an FPGA Coprozessor Board.* Universität Mannheim, 2000.

[38] B. Niemann, R. Buettner, and M. Speitel. "Brücke" zwischen IC- und system-entwickler. *Elektronik 8/2000*, pages 92–101.

[39] K.-H. Noffz. *Ein FPGA-Prozessor als 2nd-Level-Trigger für ATLAS.* Dissertation. Universität Heidelberg, 1995.

[40] P. R. Panda. High level synthesis design repository. *University of Carlifornia*, 1995.

[41] R.Ernst, J.Henkel, and T. Benner. Hardware-software codesign of embedded controllers based on hardware-extraction. In *International Workshop on Hardware-Software Codesign*, 1992.

[42] R. L. Rivest, A. Shamir, and L. Adleman. Public key cryptography. *CACM*, 21:120–126, 1979.

[43] J. Sanguinetti. *Bridging the Design Gap with Cynthesis.* www.cynapps.com.

[44] P. Shaw. A highly parallel FPGA-based machine and its formal verification. *University of Strathclyde*, 1993.

[45] N. Shirazi, P. M. Athanas, and A. L. Abbot. Implementation of a 2-D fast fourier transform on a FPGA-based custom computing machine. 1995.

[46] H. Simmler, E. Bindewald, and R. Männer. *Acceleration of Protein Energy Calculation by FPGAs.* Universität Mannheim, 2000.

[47] H. So and P. Husted. Tradeoff between DSP and FPGA-implementation. *Berkeley University*, 1999.

[48] M. Spivey. How to programm in handel. Technical report, 1995.

[49] B. Stroustrup. *Die C++ Programmiersprache.* Addison-Wesley, 1992.

[50] B. Stroustrup. *Design und Entwicklung von C++.* Addison-Wesley, 1994.

[51] Synopsys Inc. Overview of the open SystemC initiative. Technical report, 1999.

[52] A. Tarmaster, P. M. Athanas, and A. L. Abbott. Accelerating image filters using a custom computing machine. 1995.

[53] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: The coming of age. 1994.

[54] M. Wannemacher. *Das FPGA Kochbuch.* MITP Bonn, 1998.

[55] M. J. Wirthlin. The nano processor: A low resource reconfigurable processor. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 23–30, 1994.

[56] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Inc., 1996.

[57] Xilinx Inc. *The Programmable Logic Data Book.* 1994.

[58] R. Zoz. *Eine Hochsprachen-Programmierumgebung für FPGA-Prozessoren.* Dissertation, Universität Heidelberg, 1997.

Danksagung

Zum Schluß möchte ich allen danken, die zum Gelingen dieser Arbeit beigetragen haben.

Herrn Prof. Dr. Männer dafür, daß ich die Promotion an seinem Lehrstuhl durchführen konnte und für seine ausgezeichnete Betreuung.

Jozsef Ludvig und Jürgen Hesser für die interessanten und ausgiebigen Diskussionen.

Peter Dillinger und Stefan Hezel für ihre Unterstützung bei der Implementierung der Bildverarbeitungsfunktionen.

Andrea Seeger, die mir in allen verwaltungstechnischen Belangen zur Seite stand.

Meinen Eltern und meinem Bruder, für ihre Geduld und den wertvollen Rückhalt, den sie mir gegeben haben.