Das FPGA-Entwicklungssystem CHDL

Eine vollständige, C++-basierte Entwicklungsumgebung für FPGA-Koprozessoren

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

vorgelegt von

Diplom-Wirtschaftsinformatiker Klaus Kornmesser
aus Mannheim

Mannheim, 2004

Dekan: **Professor Dr. Jürgen Potthoff, Universität Mannheim**

Referent: **Professor Dr. Reinhard Männer, Universität Mannheim**

Korreferent: **Professor Dr. Peter Fischer, Universität Mannheim**

Tag der mündlichen Prüfung: 12. Oktober 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Einleitung und Motivation

Kapitel 1
Entwicklungssoftware für FPGAs und FPGA-Koprozessoren

1.1 Einführung

Field Programmable Gate Arrays (FPGAs) haben durch ihr breites Anwendungsgebiet und ihre immer weiter steigende Integrationsdichte die Welt der konfigurierbaren Logik bedeutend verändert. Die Erstellung von Designs für *Programmable Logic Devices* (PLDs) war noch überschaubar und mit einfachen Entwicklungswerkzeugen beherrschbar. FPGAs dagegen beinhalten weitaus mehr konfigurierbare Resourcen als PLDs und die Entwicklungssysteme zur Designerstellung für FPGAs sind deutlich komplexer geworden.

Insbesondere FPGA-basierte Koprozessoren, die zur Hardwarebeschleunigung von Algorithmen eingesetzt werden, stellen durch ihr enges Zusammenwirken mit Mikroprozessoren besondere Anforderungen an die Entwicklungssysteme. Die Anwendungen können einen erheblichen Umfang und Komplexitätsgrad erreichen. Daher werden höhere und abstraktere Beschreibungsebenen benötigt. Auch eine Simulation des Gesamtsystems ist unverzichtbar. Debugging-Verfahren müssen durch zusätzliche Funktionen moderner FPGAs, wie etwa das Readback-Verfahren [118] und die partielle Rekonfiguration [116] ergänzt werden. Zur optimalen Designerstellung müssen alle diese Bereiche in geeigneter Weise durch die Entwicklungssysteme unterstützt werden.

Neben den konventionellen *VHDL* [104]-basierten Systemen wie etwa *Leonardo Spectrum* [64], *Synplify* [93] oder *FPGA Express / FPGA Compiler II* [92] sind in den letzten Jahren auch neue Entwicklungssysteme entstanden. Die bedeutendsten sind *Handel-C* [20], *SystemC* [95], *JHDL* [9] und *PAM-Blox* [62], die in unterschiedlicher Form C, C++ oder auch *JAVA* zur Hardwarebeschreibung einsetzen. Einige davon sind als Klassenbibliotheken realisiert und besitzen keine eigenen, speziellen Kompiler zur Übersetzung der Hardwarebeschreibung, sondern verwenden handelsübliche C++ oder *JAVA*-Kompiler.

Jedoch berücksichtigen sowohl die konventionellen als auch die neuen Entwicklungssysteme die oben genannten Anforderungen der FPGA-Koprozessoren nicht ausreichend. Insbesondere existiert kein System, das alle diese Anforderungen in gleichem Maße erfüllt. Die vorhandenen Systeme weisen darüberhinaus heterogene Strukturen auf, die den Gesamtentwicklungsprozeß nicht optimal unterstützen können.

In der vorliegenden Arbeit wurde das Konzept der C++-basierten Hardwarebeschreibung mittels Klassenbibliotheken und handelsüblichen Kompilern weiterentwickelt und optimiert. Das Ergebnis ist ein homogenes System, das eine deutlich verbesserte Unterstützung für FPGA-Koprozessoren bietet: Das FPGA-Entwicklungssystem *CHDL*.

CHDL verfügt über mehrere parallel einsetzbare Beschreibungsebenen von der detaillierten strukturellen Spezifikation bis hin zur modernen Hochsprachenbeschreibung. Die Simulation von Gesamtsystemen wird ebenso unterstützt wie die Anwendung moderner Debugging-Verfahren.

Im folgenden Abschnitt sollen die oben erwähnten Problembereiche anhand eines einfachen FPGA-Koprozessors verdeutlicht werden.

1.2 Problembereiche

1.2.1 Struktur eines FPGA-Koprozessors

Abbildung 1.1 a zeigt die Grundstruktur eines PCI-basierten FPGA-Koprozessors. In der Regel sind solche Systeme in Form von Einsteckkarten für PCs realisiert. Sie enthalten einen oder mehrere FPGAs, in denen im Rahmen der vorhandenen Resourcen nahezu beliebige digitale Schaltungen implementiert werden können.

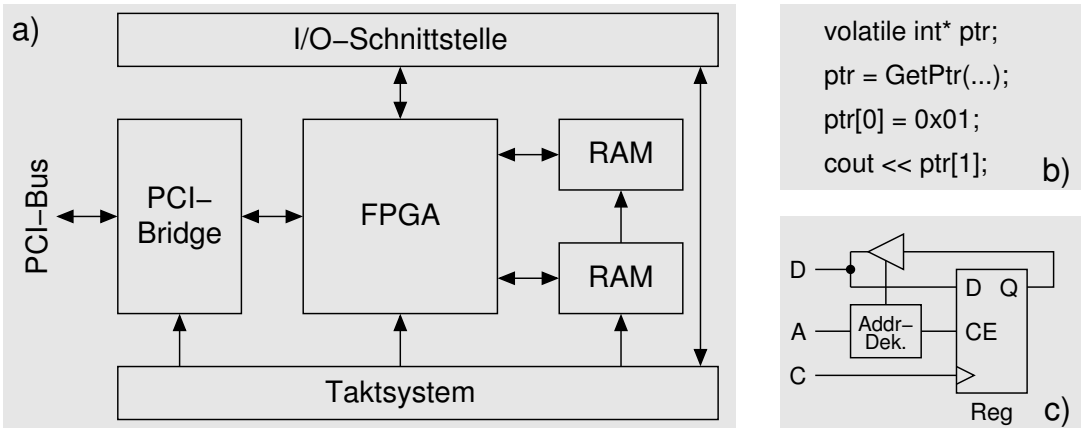


Abbildung 1.1: Komponenten eines FPGA-Koprozessors mit Übersicht (a), Software (b) und FPGA-Konfiguration (c)

1.2.2 Enge Kopplung zwischen Hardware- und Softwarebereich

Auf dem Hostrechner, meist einem handelsüblichen PC, wird ein Programm ausgeführt, das entsprechend der gewünschten Anwendung den Datenaustausch mit der Koprozessorkarte durchführt. Dieser Datenaustausch wird über Pseudoregister ("*special function registers*") realisiert. Sie bilden die Schnittstelle zwischen der Software (Abb. 1.1 b) und der Hardwareimplementierung im FPGA (Abb. 1.1 c). Das Verhalten der Software ist in hohem Maße abhängig von der Schaltung im FPGA. Umgekehrt wird das Verhalten des FPGAs vom Ablauf des Programmes gesteuert. Keine der beiden Komponenten kann in sinnvoller Weise ohne die andere arbeiten.

Obwohl beide Bereiche eine hohe Abhängigkeit aufweisen, müssen sie mit grundlegend verschiedenen Entwicklungssystemen und -sprachen implementiert werden:

- Host-Software ("*Applikation*").

  Diese umfaßt das Konfigurieren des FPGAs, den Datentransfer und alle weiteren Aufgaben, die nur vom Mikroprozessorsystem sinnvoll bearbeitet werden können (Dateizugriff, grafische Benutzerschnittstelle usw.). Die Implementierung erfolgt in der Regel mittels C oder C++.

- FPGA-Konfiguration ("*Design*").

  Hier wird die im FPGA zu realisierende digitale Schaltung festgelegt. Zur Hardwarebeschreibung stehen *VHDL* bzw. C-ähnliche Sprachen der neueren Entwicklungssysteme zur Verfügung.

Der Entwickler muß beide Sprachen und die jeweils dazugehörigen Entwicklungsumgebungen beherrschen, um Anwendungen für FPGA-Koprozessoren erstellen zu können.

1.2.3 Beherrschung von Umfang und Komplexität

Moderne FPGAs mit ihren enorm gestiegenen Resourcen (mehrere zehntausend Flip-Flop-Funktionen pro Baustein) ermöglichen es, umfangreiche und komplexe Implementierungen auf FPGA-Koprozessoren zu realisieren. Um jedoch diesen Umfang und die Komplexität beherrschen zu können, muß der Anwender von der Entwicklungssoftware auf geeignete Weise unterstützt werden.

So ist zur Implementierung effizienter FPGA-Designs notwendig, daß er verschieden hohe Abstraktionsebenen einsetzen kann. Den Hardwarebeschreibungssprachen *VHDL* und *Verilog* [103] sowie den Systemen *JHDL* und *PAM-Blox* fehlen höhere Ebenen, mit denen

z.B. das Erstellen komplexer Zustandsmaschinen erleichtert werden kann. Bei hochsprachenorientierten Systemen wie *Handel-C* dagegen ist keine direkte Integration niedriger Ebenen möglich.

Im Bereich der herkömmlichen Softwareentwicklung sind die Probleme von Umfang, Komplexität und Abstraktionsebenen schon seit langer Zeit bekannt. Hier wurden wirksame Lösungsmethoden entwickelt, so etwa der Übergang von Assembler- zu Hochsprachen oder das Konzept der Objektorientierung mit Vererbung und überladenen Methoden. An zeitkritischen Stellen kann jederzeit auf die Assemblerebene zurückgegriffen werden. In modernen Kompilern kann dies innerhalb des C/C++-Codes in Form von Inline-Assembler erfolgen. Auch die Source-Level-Debugger unterstützen diese Integration.

Auf den Bereich der Hardwarebeschreibungssprachen wurden diese Methoden bisher jedoch nicht ausreichend übertragen.

1.2.4 Realistische Simulation des Gesamtsystems

Nach der heute üblichen Vorgehensweise führt der Entwickler eine softwaremäßige Simulation der Hardwarebeschreibung durch, bevor er das Design auf der Zielhardware in Betrieb nimmt. Die Simulation bietet bessere Untersuchungsmöglichkeiten der einzelnen Signale und vermeidet darüberhinaus bei Fehlern Schäden an der Hardware.

Aufgrund der engen Kopplung zwischen Mikroprozessor und FPGA ist eine realistische Simulation nur zusammen mit der Applikation möglich. Werden Applikation und FPGA-Design jedoch mit verschiedenen Entwicklungssystemen erstellt, führt dies in der Praxis oft zu Schwierigkeiten, weil sich die Systeme zur Simulation nur schwer verbinden lassen (Abb. 1.2).

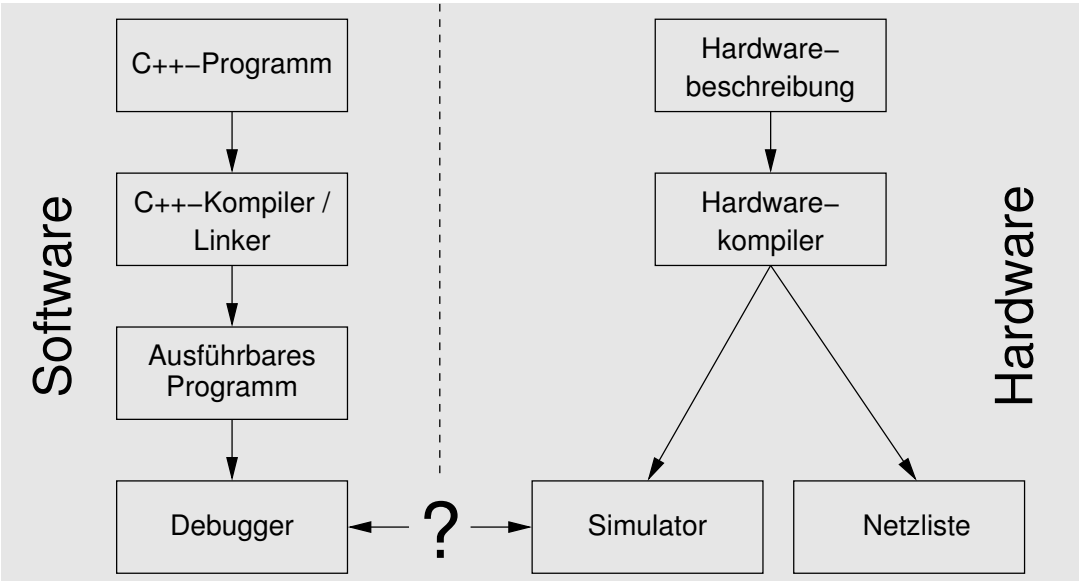


Abbildung 1.2: Probleme bei der Simulation des Gesamtsystems

Die Applikation kann zwar mit einem handelsüblichen Debugger schrittweise ausgeführt und getestet werden, bei Zugriffen auf Pseudoregister ist jedoch ein Zusammenwirken mit dem Hardwaresimulator erforderlich. Ohne die Simulation des Verhaltens dieser Pseudoregister kann die Applikation nicht realistisch fortgesetzt werden.

Andererseits sind zur Simulation der Hardwarebeschreibung externe Stimuli erforderlich. Die existierenden *VHDL*-basierten Systeme ermöglichen die Erzeugung solcher Stimuli mittels *VHDL*-Testbenches oder Testvektoren. Die Möglichkeiten sind jedoch eingeschränkt, da *VHDL* eine Hardwarebeschreibungssprache, aber keine vollständige und effiziente Programmiersprache darstellt. Bereits das Simulieren großer Speicherbausteine, etwa

SDRAMs, bereitet in der Praxis aufgrund der erforderlichen Rechenzeit und des Speicherbedarfes der *VHDL*-Simulatoren Schwierigkeiten. Die Simulation einer vollständigen FPGA-Koprozessoranwendung einschließlich Softwarekomponente ist nahezu unmöglich.

Moderne *VHDL*-Simulatoren beinhalten eine spezielle Schnittstelle, über die konventioneller Programmcode in Form von dynamischen Linkbibliotheken (DLLs) integriert werden kann. Diese Schnittstellen sind jedoch in der Praxis unhandlich und die Gesamtsimulation läuft nicht schnell genug ab, um komplexe Anwendungen in akzeptabler Zeit zu simulieren.

Das *SystemC*-Konzept erlaubt die Erstellung kompletter Koprozessoranwendungen mit C++, wobei auch ein Schwerpunkt auf die Simulation von Gesamtsystemen gelegt wird. Simulation und Synthese der Hardwarebeschreibung werden jedoch unterschiedlich behandelt: Die Simulation erfolgt mittels C++-Klassenbibliothek, die Synthese mittels speziellen Kompilern. Daher kann nicht jede simulierbare Beschreibung auch synthetisiert werden. Darüberhinaus ist das konkrete Syntheseergebnis kompilerabhängig.

Simulation und Synthese müssen jedoch auf derselben Datenbasis beruhen, um eine realistische Simulation von Gesamtsystemen zu ermöglichen.

1.2.5 Moderne Techniken zum Hardware-Debugging

Auch eine sorgfältig durchgeführte Simulation kann kein allgemein fehlerfreies Funktionieren eines FPGA-Designs garantieren. Dies gilt besonders im Zusammenhang mit Mikroprozessorsystemen, die aufgrund von Busarbitrierung und Multitasking-Betriebssystemen ein stark schwankendes und schwer vorhersehbares Zeitverhalten aufweisen. Hier kann eine Simulation immer nur eine beschränkte Anzahl von Ablaufkombinationen abdecken. Damit ist es zwar möglich, die Existenz eines Fehlers nachzuweisen, aber niemals die Fehlerfreiheit.

Tritt im Echtzeitbetrieb ein Fehler auf, der in der Simulation nicht reproduzierbar ist, wird ein Hardware-Debugging notwendig sein. Dazu stehen in der Praxis spezielle Geräte, wie etwa Logikanalyzer, zur Verfügung, mit denen das Verhalten des Systems überwacht und aufgezeichnet werden kann. Es ist jedoch auf diese Weise nicht ohne weiteres möglich, interne Signale des FPGA-Designs zu beobachten, was für die Fehlersuche von erheblicher Bedeutung ist.

Das Readback-Verfahren der FPGAs ermöglicht es, zu jedem beliebigen Zeitpunkt während des normalen Betriebes den Zustand aller internen Flip-Flops, Speicherblöcke und weiterer Signale auszulesen.

Durch partielle Rekonfiguration können am konfigurierten FPGA Änderungen vorgenommen werden. So sind die Zustände der Flip-Flops, Funktionsgeneratoren und Block-RAMs modifizierbar.

Diese beiden Verfahren erfordern jedoch eine Unterstützung durch das zur Hardwarebeschreibung verwendete Entwicklungssystem. Das größte Problem besteht darin, daß bei den herkömmlichen Systemen die Namen der Elemente in der Hardwarebeschreibung nicht unverändert in die Netzliste übernommen werden. Es ist daher schwierig, ein Symbol der Netzliste einem Bauteil der ursprünglichen Hardwarebeschreibung zuzuordnen. Ohne diese Zuordnung können jedoch Readback und partielle Rekonfiguration nicht automatisiert eingesetzt werden.

1.3 Zusammenfassung

Moderne FPGA-Koprozessoren stellen besondere Anforderungen an die Entwicklungssysteme bezüglich:

- Enger Kopplung zwischen Hardwarebereich (FPGA) und Softwarebereich (Mikroprozessor).
- Beherrschung von Umfang und Komplexität.
- Realistischer Simulation des Gesamtsystems.
- Moderner Techniken zum Hardware-Debugging.

Die konventionellen *VHDL*-basierten Systeme sind hierbei im wesentlichen durch die Eigenschaften der Sprache *VHDL* beschränkt. Diese ermöglicht keine höheren Abstraktionsebenen und keine vollständige Simulation von Koprozessoranwendungen. Bei einer Implementierung von Testbenches mittels *VHDL* sind zudem zwei Entwicklungssprachen notwendig. Weiterhin übernehmen *VHDL*-Kompiler Bauteilnamen nicht unverändert in die Netzliste, wodurch kein automatisierter Einsatz von Readback und partieller Rekonfiguration möglich ist.

Neuere C/C++-basierte Entwicklungssysteme überwinden die Sprachbarriere zwischen Hardware- und Softwarebereich. Jedoch werden Beherrschung von Umfang und Komplexität und Simulation komplexer externer Bausteine nicht ausreichend unterstützt. Darüberhinaus ist eine gleichzeitige Anwendung verschiedener Abstraktionsebenen sowie die Anwendung der Readback-Funktion bei diesen Systemen nicht vorgesehen.

SystemC hat zwar explizit die Simulationsmöglichkeit von Gesamtsystemen zum Ziel. Dieses Entwicklungssystem zeigt jedoch durch die unterschiedliche Behandlung von Simulation und Synthese Schwächen bei der Übereinstimmung dieser beiden Bereiche.

Das hochsprachenorientierte *Handel-C* stellt keine direkte Integration struktureller Ebenen zur Verfügung und besitzt keine ausreichende Unterstützung für die Simulation von FPGA-Koprozessoren. Wie auch bei *SystemC* ist keine automatisierte Anwendung von Readback und partieller Rekonfiguration möglich.

Zusammenfassend läßt sich feststellen, daß keines der bisher existierenden Systeme die Anforderungen erfüllt, die moderne FPGA-Koprozessoren an die Entwicklungssoftware stellen.

Kapitel 2

CHDL: Ein C++-basiertes Entwicklungssystem für FPGA-Koprozessoren

2.1 Einführung

Aufgrund der im vorigen Kapitel aufgezeigten Probleme herkömmlicher Systeme wurde im Rahmen dieser Arbeit ein homogenes C++-basiertes Entwicklungssystem implementiert, *CHDL* (**C**++-based **H**ardware **D**escription **L**anguage). Es setzt die konventionelle Programmiersprache C++ sowohl zur Hardwarebeschreibung als auch zur Simulation ein.

Ein Vorteil von *CHDL* besteht darin, daß zum Übersetzen der strukturellen Hardwarebeschreibung kein spezieller proprietärer Kompiler benötigt wird. *CHDL* beruht auf dem Prinzip einer C++-Klassenbibliothek, daher genügt ein handelsüblicher C++-Kompiler, um die Beschreibung zu übersetzen. Das resultierende Programm führt dann sowohl die Simulation als auch die Synthese durch.

CHDL kann prinzipiell alle verfügbaren FPGAs unterstützen. Implementiert sind zur Zeit alle FPGAs der Firma *XILINX* von *XC4000* bis *Virtex-II Pro*. Aufgrund des kompatiblen Netzlistenformates kann *CHDL* auch zur Entwicklung von Designs für *XILINX-XC9500* PLDs sowie FPGAs der Reihe *AT40K* von *Atmel* verwendet werden.

Die folgenden Abschnitte vermitteln einen kurzen Einblick in die Fähigkeiten des *CHDL*-Systems. Die Darstellung orientiert sich dabei an der üblichen Vorgehensweise beim Hardware-Design:

- Hardwarebeschreibung.
- Simulation.
- Synthese.
- Hardware-Debugging.

2.2 Hardwarebeschreibung

Zur Hardwarebeschreibung auf der untersten Ebene werden speziell definierte C++-Klassen und überladene Operatoren verwendet. Damit läßt sich eine strukturelle Beschreibung realisieren, die in ihrem Aussehen den klassischen strukturellen Hardwarebeschreibungssprachen wie etwa *ABEL* [117] stark ähnelt:

```
PadIn  A("A");
PadIn  CLK("CLK");
PadOut O("O");
DFF    FF1("FF1",CLK);

FF1 = (!FF1 & A) | (FF1 & !A);
O   = FF1;
```

Innerhalb der Hardwarebeschreibung können alle Mechanismen, die die Programmiersprache C++ zur Verfügung stellt, verwendet werden. So sind etwa bedingte Anweisungen, Parametrisierungen, Vererbung und virtuelle Methoden einsetzbar. Diese Kombinationsmöglichkeit stellt eine mächtige Methode dar, mit der sich selbst umfangreiche und komplexe FPGA-Designs erstellen und beherrschen lassen.

CHDL erlaubt weiterhin Beschreibungen auf höheren Abstraktionsebenen, um z.B. Zustandsmaschinen durch textuelle Beschreibung von Flußdiagrammen zu implementieren:

```
  BeginState();
    A = 0;
  EndState();

LABEL("Start");
  BeginState();
    A++;
  EndState();

  IF (A < 10,"Start");
```

Dieses Verfahren wird in Kapitel 9 detailliert beschrieben.

Eine noch höhere Ebene bietet die Hochsprachenbeschreibung, die in Teil IV erläutert wird:

```
  A = 0; S = 0;
  while (A < 10)
   {
     S = S + A;
     A = A + 1;
   }
```

Ein wichtiges Merkmal des *CHDL*-Systems liegt darin, daß alle diese verschiedenen Ebenen gleichberechigt nebeneinander in einer homogenen Entwicklungsumgebung eingesetzt werden können.

Der Entwickler kann bei der Arbeit mit *CHDL* alle für C++ verfügbaren Entwicklungswerkzeuge verwenden, so etwa auch Debugger, spezielle Quelltexteditoren oder Klassenbrowser.

2.3 Simulation

CHDL unterstützt ohne Einschränkungen die funktionale Simulation der Hardwarebeschreibung sowie der kompletten FPGA-Umgebung.

Das funktionale Verfahren ermöglicht durch den geringeren Rechenaufwand eine umfangreichere Simulation des Gesamtsystems als ein zeitbasiertes. Eine solche ist bei der Arbeit mit FPGA-Koprozessoren von größerer Bedeutung als eine zeitgenaue, jedoch rechenintensive Simulation eines Teilsystems. Zudem sind genaue Informationen über das zeitliche Verhalten eines FPGA-Designs erst nach Abschluß des Place&Route-Prozesses verfügbar. Zur Bestimmung der maximalen Taktfrequenz muß in jedem Fall der Timing-Analyzer von *XILINX* verwendet werden, da nur dieser alle internen Charakteristika der FPGAs berücksichtigt. Aus diesen Gründen wurde bei *CHDL* auf eine exakte zeitbasierte Simulation bewußt verzichtet.

Der Anwender kann beliebige externe Hardwarekomponenten in die Simulation einbeziehen. Das Verhalten dieser Komponenten wird dabei durch eine entsprechende C++-Funktion modelliert. Da *CHDL* als Klassenbibliothek konzipiert ist, können die Simulationsmodelle direkt ohne den Umweg über schwierig zu handhabende DLL-Schnittstellen integriert werden.

Auch interne Komponenten eines FPGA-Designs, die noch nicht im Detail implementiert wurden, können zunächst als angenäherte Simulationsmodelle einbezogen werden. Weiterhin ist zur Beschleunigung eine hierarchische Simulation möglich. Hierbei werden bereits erfolgreich getestete Module durch ihr Simulationsmodell ersetzt. Auf diese Weise lassen sich die Simulationszeiten erheblich verkürzen.

Durch eine spezielle Verbindung zwischen Hostsoftware und Hardwaresimulator können Zugriffe auf Pseudoregister realistisch simuliert werden. Die Hostsoftware läuft dabei in Echtzeit auf dem Prozessor und gegebenenfalls unter der Kontrolle eines konventionellen C++-Debuggers ab.

2.4 Synthese

CHDL unterstützt die direkte Ausgabe von Netzlisten im XNF- oder EDIF-Format. Die erzeugten Netzlisten können ohne weitere Bearbeitung direkt an die Place&Route-Software übergeben werden. Außer dieser ist keine weitere Synthesesoftware erforderlich, insbesondere keine *VHDL*-Kompiler.

Alle in der *CHDL*-Hardwarebeschreibung enthaltene Vorplazierungs- und Timinginformationen werden automatisch in die Netzliste integriert.

Die Synthese erfolgt wie die Simulation durch Ausführen der kompilierten Hardwarebeschreibung. Simulation und Synthese verwenden dieselbe Datenbasis, so daß eine nahezu vollständige Übereinstimmung erreicht wird.

2.5 Hardware-Debugging

CHDL unterstützt die Debugging-Verfahren, die von modernen FPGAs zur Verfügung gestellt werden.

Ermöglicht wird dies durch eine eindeutige und vorhersehbare Zuordnung der Bauteilnamen in der Netzliste. Mittels Readback können zu jedem beliebigen Zeitpunkt die aktuellen Zustände von Flip-Flops und RAMs ausgelesen werden.

Die partielle Rekonfiguration erlaubt weiterhin die schnelle Modifizierung dieser Zustände ohne erneuten Place&Route-Durchlauf. Damit lassen sich mächtige Verfahren zum Hardware-Debugging, wie etwa integrierte Logikanalyzer mit veränderbaren Triggerbedingungen, implementieren.

Durch die eindeutigen Netzlistennamen ergeben sich auch automatisierte Anwendungen zur nachträglichen schnellen Modifizierung von bereits gerouteten FPGA-Designs. Dies kann über eine Fernsteuerung des "fpga_editor"-Programmes mittels Batchdateien erfolgen.

2.6 Zusammenfassung

Es wurde ein erster Eindruck vermittelt, wie *CHDL* die Probleme moderner FPGA-Koprozessoren unterstützen kann. *CHDL* weist gegenüber anderen Entwicklungssystemen insbesondere folgende Vorteile auf:

- Die Kenntnis einer einzigen Programmiersprache (C++) ist ausreichend, um sowohl den Hardware- als auch den Softwarebereich eines FPGA-Koprozessors implementieren zu können.
- *CHDL* beruht vollständig auf handelsüblichem C++ ohne jegliche Modifikationen und ist als Klassenbibliothek realisiert. Jeder Softwareentwickler, der über Kenntnisse der wichtigsten Prinzipien der objektorientierten Programmierung und der digitalen Schaltungstechnik verfügt, ist damit in der Lage, Algorithmen auf FPGAs zu implementieren. Er kann dabei seine ihm vertraute C++-Entwicklungsumgebung, z.B. *Microsoft Visual Studio*, benutzen. Die Methoden, mit denen *CHDL*-Implementierungen erstellt werden, entsprechen denen der herkömmlichen Softwareentwicklung.
- Beim Erstellen komplexer und umfangreicher Implementierungen können die Vorteile der objektorientierten C++-Programmierung genutzt werden.
- Durch die Einbindung benutzerdefinierter Simulationsmodelle und der Simulation von Pseudoregistern werden FPGA-Koprozessoren optimal unterstützt.
- Außer einem handelsüblichen C++-Kompiler sowie der Place&Route-Software des FPGA-Herstellers sind keine weiteren Entwicklungswerkzeuge notwendig.
- Die Methoden, die moderne FPGAs zum Hardware-Debugging bereitstellen, werden vollständig unterstützt.

Teil II

Grundlagen und Stand der Technik

Kapitel 3
Digitale Schaltungstechnik

3.1 Einführung

Die digitale Schaltungstechnik bildet die Grundlage für die Realisierung von Schaltungen in FPGAs.

Die wichtigsten Grundelemente sind zunächst die elektronischen Verknüpfungselemente (Gatter), die einfache Schaltfunktionen ausführen. Aus diesen können Schaltnetze und Speicherelemente konstruiert werden, welche wiederum die Basis für Zustandsmaschinen darstellen.

Schaltnetze, Speicherelemente und Zustandsmaschinen bilden die Grundbausteine bei der Implementierung von Algorithmen in Hardware.

Diese Grundbausteine besitzen zeitliche Eigenschaften, die die maximal erreichbare Taktfrequenz der Hardwareimplementierung festlegen. Mit Hilfe verschiedener Optimierungsverfahren läßt sich die Taktfrequenz erhöhen.

3.2 Verknüpfungselemente

Elektronische Verknüpfungselemente, auch Gatter genannt, sind Schaltungen mit einem oder mehreren Eingängen sowie einem Ausgang, die einfache Schaltfunktionen ausführen. Eingänge und Ausgänge lassen sich durch binäre Schaltvariablen darstellen, die die Werte "0" bzw. "1" annehmen können. Eine Schaltfunktion ist eine eindeutige Zuordnungsvorschrift, die jeder Wertekombination ihrer Schaltvariablen den Ergebniswert "0" oder "1" zuordnet [49, 74, 11, 26].

Es gibt drei Grundverknüpfungen: "UND", "ODER" und "NICHT". Obwohl sich jede beliebige Schaltfunktion bereits mit diesen Grundverknüpfungen realisieren ließe, gibt es in der Praxis noch weitere, abgeleitete Verknüpfungen, mit denen Schaltfunktionen gelegentlich kompakter ausgedrückt werden können, z.B. "UND NICHT", "ODER NICHT" und "EXKLUSIV-ODER".

Die Abbildungen 3.1 und 3.2 zeigen einige Verknüpfungselemente mit ihrer grafischen Darstellung, wie sie im wissenschaftlichen Bereich verbreitet ist, und ihre Ergebniswerte im Überblick.

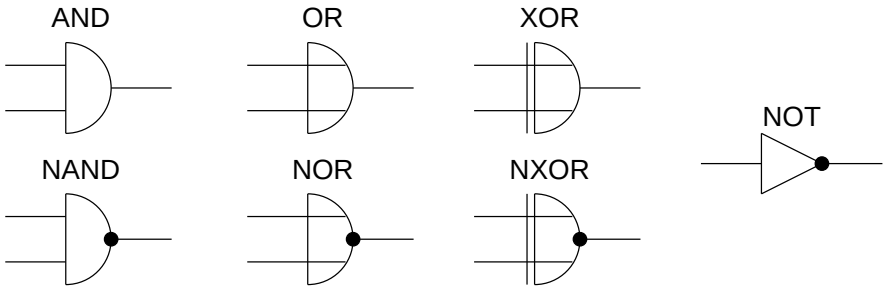


Abbildung 3.1: Darstellung einiger Verknüpfungselemente

3.3 Schaltnetze

Verknüpfungselemente lassen sich zu Schaltnetzen (Abb. 3.3) zusammenstellen, um beliebig komplexe Schaltfunktionen zu realisieren. In den folgenden Ausführungen werden als Schaltnetze nur solche Anordnungen bezeichnet, die keine Zyklen aufweisen. Sind Zyklen vorhanden, handelt es sich nicht um Schaltnetze, sondern bei sinnvoller Struktur um Speicherelemente.

A	B	NOT A	NOT B	A AND B	A OR B	A XOR B	A NAND B	A NOR B	A NXOR B
0	0	1	1	0	0	0	1	1	1
0	1	1	0	0	1	1	1	0	0
1	0	0	1	0	1	1	1	0	0
1	1	0	0	1	1	0	0	0	1

Abbildung 3.2: Ergebnisse einiger Schaltfunktionen

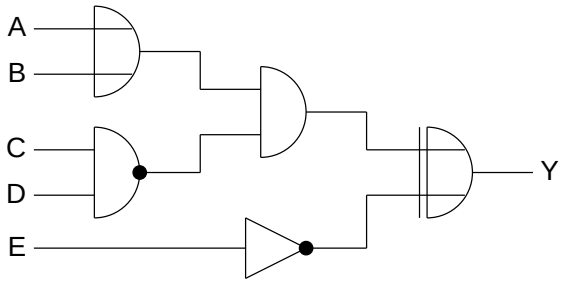


Abbildung 3.3: Ein Schaltnetz

Es stehen verschiedene Methoden zur Verfügung, um Schaltnetze in geeigneter Form zu beschreiben:

- Grafische Darstellung.

  Die grafische Darstellung eines Schaltnetzes in Form eines Schaltplanes (Abb. 3.3) ist für einfache Anordnungen zweckmäßig. Mit zunehmender Komplexität wird sie jedoch schnell unübersichtlich.

- Wertetabellen.

  Wertetabellen (Abb. 3.2) enthalten alle kombinatorisch möglichen Eingangszustände und den jeweiligen Ergebniswert. Sie können für Schaltnetze mit wenigen Eingängen schnell erstellt werden und lassen sich leicht verifizieren. n Schaltvariablen ermöglichen insgesamt 2^n verschiedene Kombinationen. Mit zunehmender Anzahl von Eingängen werden sie folglich sehr umfangreich. Zur kompakteren Darstellung können mehrere Zeilen mit gleichem Ergebniswert zusammengefaßt werden, wenn gemeinsame Eingänge innerhalb der Gruppierung unerheblich sind. Diese Eingänge werden dann durch ein "Don't-Care"-Symbol (X) gekennzeichnet [74].

- Verhaltensbeschreibungen.

  Dies sind verbale Formulierungen, mit denen das Verhalten eines Schaltnetzes in Abhängigkeit von seinen Schaltvariablen beschrieben wird. Die Formulierungen müssen eindeutig spezifiziert sein, um Mißverständnisse oder Mehrdeutigkeiten zu vermeiden. Verhaltensbeschreibungen von Schaltnetzen werden z.B. von der Hardwarebeschreibungssprache *VHDL* [104] unterstützt:

```
if (A = '1' or B = '1') then
    Y <= '1';
  else
    Y <= '0';
end if;
```

  oder:

```
Y <= '1' when A = '1' or B = '1' else '0';
```

Beide Formulierungen legen fest, daß das Schaltnetz den Ausgangswert "1" liefern soll, wenn eine der beiden Schaltvariablen den Wert "1" annimmt. Ansonsten soll der Ausgangswert den Zustand "0" besitzen.

- Formelmäßige Darstellung mit Schaltvariablen und Operatoren.

  Schaltnetze können durch Formeln dargestellt werden, in denen die Schaltvariablen mit Operatoren verknüpft sind. Dabei lassen sich Klammerungen verwenden, um die Prioritäten bei der Auswertung festzulegen. Auf diese Weise wird auch die Beschreibung komplexer Schaltfunktionen auf übersichtliche und kompakte Weise möglich. Solche formelmäßigen Darstellungen werden z.B. bei der Hardwarebeschreibungssprache *ABEL* [117] verwendet. Das Zeichen "&" repräsentiert dabei eine "UND"-Verknüpfung, ein "#" eine "ODER"-Verknüpfung und ein "!" eine "NICHT"-Operation.

  ```
  Y = ((A # B) & (C # D)) # !E
  ```

3.4 Speicherelemente

Anordnungen von Verknüpfungselementen können Zyklen enthalten. Stellen diese eine negative Rückkopplung dar, so wird die entstehende Konstruktion bei einigen oder allen Eingangskombinationen keinen stabilen Zustand einnehmen können. Sie gerät ins Schwingen und ist in der Digitaltechnik nicht sinnvoll einsetzbar.

Eine positive Rückkopplung dagegen kann einen erwünschten Speichereffekt verursachen. Bei geeigneter Verschaltung befindet sich die Anordnung stets stabil in einem von mehreren Zuständen. Nur bestimmte Eingangskombinationen können einen Zustandswechsel herbeiführen. Damit ist das Verhalten nicht, wie bei Schaltnetzen, nur vom Zustand der Eingänge abhängig, sondern zusätzlich noch vom aktuellen inneren Zustand.

Es lassen sich verschiedene Typen von Speicherelementen bilden, die sich in ihrem Verhalten unterscheiden [49, 74, 26]:

- RS-Flip-Flop (Abb. 3.4 a).

  Ein RS-Flip-Flop besitzt einen Reset- und einen Set-Eingang. Der Reset-Eingang versetzt das Flip-Flop in den Zustand "0", der Set-Eingang in den Zustand "1". Sind beide Eingänge inaktiv, bleibt der aktuelle Zustand erhalten.

- D-Flip-Flop (Abb. 3.4 b).

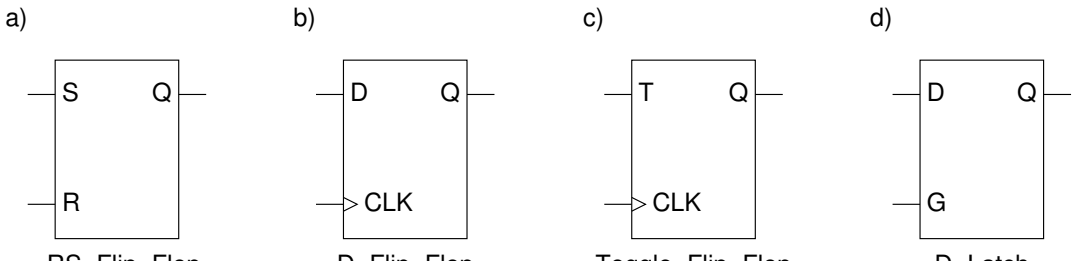
  Ein D-Flip-Flop ändert seinen aktuellen Zustand nur bei einer steigenden Signalflanke des Takteinganges. Es wird dann der Wert übernommen, der zu diesem Zeitpunkt am Dateneingang anliegt.

- Toggle-Flip-Flop (Abb. 3.4 c).

  Ein Toggle-Flip-Flop ändert wie das D-Flip-Flop seinen aktuellen Zustand nur bei einer steigenden Taktflanke. Hat zu diesem Zeitpunkt der T-Eingang den Wert "1", wechselt es in den anderen Zustand. Beim Wert "0" erfolgt kein Zustandswechsel.

- D-Latch (Abb. 3.4 d).

  Ein D-Latch übernimmt den Wert am Dateneingang, solange der Gate-Eingang aktiv ist. Sobald der Gate-Eingang inaktiv wird, bleibt der aktuelle Zustand fixiert.



RS–Flip–Flop

R	S	Q
0	0	Q
0	1	1
1	0	0
1	1	0

D–Flip–Flop

CLK	D	Q
0	X	Q
1	X	Q
⌜	0	0
⌜	1	1

Toggle–Flip–Flop

CLK	T	Q
0	X	Q
1	X	Q
⌜	0	Q
⌜	1	!Q

D–Latch

G	D	Q
0	X	Q
1	0	0
1	1	1

Abbildung 3.4: Verschiedene Typen von Speicherelementen

3.5 Zustandsmaschinen

Zustandsmaschinen bestehen aus Schaltnetzen und Speicherelementen. Üblicherweise werden D-Flip-Flops als Speicherelemente verwendet. Die damit entstehende Anordnung besitzt den Vorteil, daß sie innerhalb ihrer zulässigen Taktfrequenz unabhängig von der Durchlaufverzögerung des Schaltnetzes arbeiten kann.

In Abhängigkeit von der konkreten Anordnung der Schaltnetze und Speicherelemente können zwei Grundmodelle von Zustandsmaschinen unterschieden werden [11]:

- Die *Mealy*-Maschine (Abb. 3.5 a).

  Bei der *Mealy*-Maschine ändert sich der Ausgangsvektor Y, sobald sich ein Eingangssignal ändert. Es existiert ein direkter Logikpfad von X zu Y durch ein Schaltnetz.

- Die *Moore*-Maschine (Abb. 3.5 b).

  Bei der *Moore*-Maschine ändert sich der Ausgangsvektor Y erst mit dem nächsten Takt nach Änderung eines Eingangssignals. Die Ausgänge werden hier ausschließlich durch eine Verknüpfung der Speicherelemente gebildet. Es existiert kein direkter Logikpfad von X nach Y.

Die *Moore*-Maschine besitzt in konkreten Anwendungen den Nachteil, daß sie mit dem Ausgangsvektor Y nicht sofort auf ein Ereignis an den Eingangssignalen reagieren kann.

Ein Vorteil dieser Maschine ist jedoch in der zeitlichen Entkopplung der Ausgänge von den Eingängen zu sehen. Bei einer Aneinanderreihung mehrerer Zustandsmaschinen läßt sich damit eine Aufsummierung der Verzögerungszeiten mit den vor- und nachgeschalteten Maschinen vermeiden.

Das Verhalten von Zustandsmaschinen läßt sich durch Zustandsdiagramme (Abb. 3.6) beschreiben. Darin wird jeder Zustand durch einen Kreis repräsentiert. Ein Übergang von einem Zustand zu einem anderen wird durch einen Pfeil gekennzeichnet und muß eindeutig sein. Die Bezeichnung des Pfeils gibt dabei an, unter welchen Bedingungen der jeweilige Übergang stattfinden soll. Ein unbeschrifteter Pfeil kennzeichnet einen unbedingten Übergang. Führt für eine bestimmte Bedingung der Pfeil zum selben Zustand zurück, so muß er nicht eingezeichnet werden. Für jede Bedingung kann nur ein Pfeil zu einem anderen Zustand existieren. Es darf auch Zustände geben, die nie erreicht werden.

Zustandswechsel erfolgen nur bei steigenden Taktflanken. Kurzzeitige Änderungen der Eingangssignale zwischen den Taktflanken haben keine Auswirkungen auf die Zustände. Sie können bei *Mealy*-Maschinen jedoch eine entsprechende zeitweilige Änderung der Ausgangssignale bewirken.

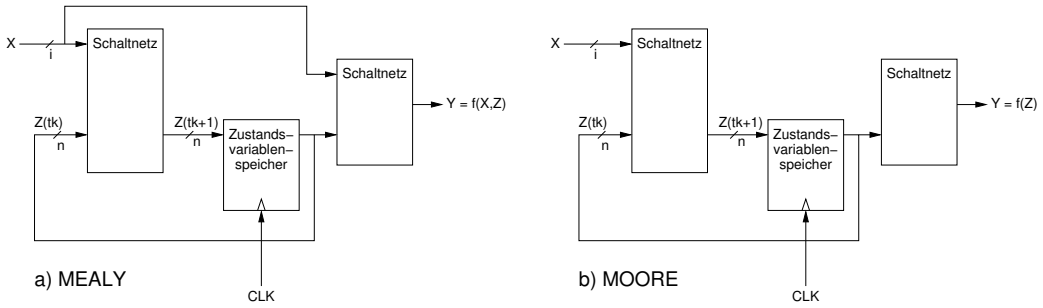


Abbildung 3.5: *Mealy*- und *Moore*-Zustandsmaschine

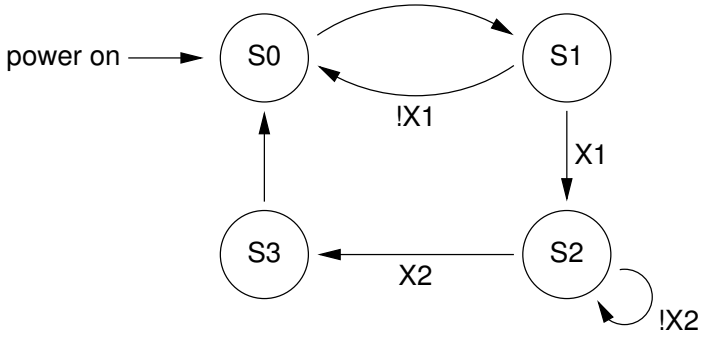


Abbildung 3.6: Ein Zustandsdiagramm

3.6 Zeitverhalten von Schaltnetzen, Speicherelementen und Zustandsmaschinen

3.6.1 Schaltnetze

Änderungen an den Eingängen eines Schaltnetzes wirken sich nicht sofort auf die Ausgänge aus. Vielmehr erfolgt die Änderung erst nach einer zeitlichen Verzögerung. Diese Zeit wird Durchlaufverzögerung (*propagation delay*) genannt. Sie ist abhängig vom Signalpfad durch das Schaltnetz und insbesondere von der Anzahl an Einzelgattern, die sich in diesem Pfad befinden. Somit kann die Verzögerung für jeden Signalpfad unterschiedlich sein (Abb. 3.7, 3.3). Auch die Richtung des Zustandswechsels kann Einfluß auf das Zeitverhalten haben [49, 11].

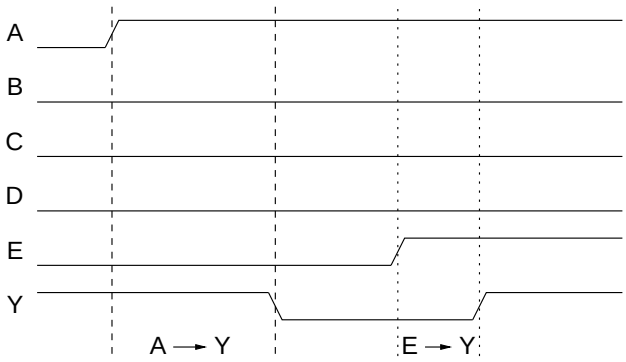


Abbildung 3.7: Zeitverhalten in Schaltnetzen

3.6.2 Speicherelemente

Bei den flankengesteuerten Flip-Flops sind folgende Zeiten von Bedeutung (Abb. 3.8 a):

- Clock-to-output-Zeit.

  Der Ausgang eines Flip-Flops ändert sich nicht sofort bei einer positiven Taktflanke, sondern erst nach einer bestimmten Verzögerungszeit.

- Setup-Zeit.

  Ein Eingangssignal wird von einem Flip-Flop nur dann bei der positiven Taktflanke zuverlässig übernommen, wenn es vorher für eine bestimmte Mindestzeit konstant angelegen hat.

- Hold-Zeit.

  Das Eingangssignal muß auch nach der positiven Taktflanke noch eine bestimmte Zeit lang stabil bleiben, um zuverlässig übernommen zu werden.

Dagegen ist bei den D-Latches anstelle der Clock-to-output-Zeit die Durchlaufverzögerung zu berücksichtigen (Abb. 3.8 b).

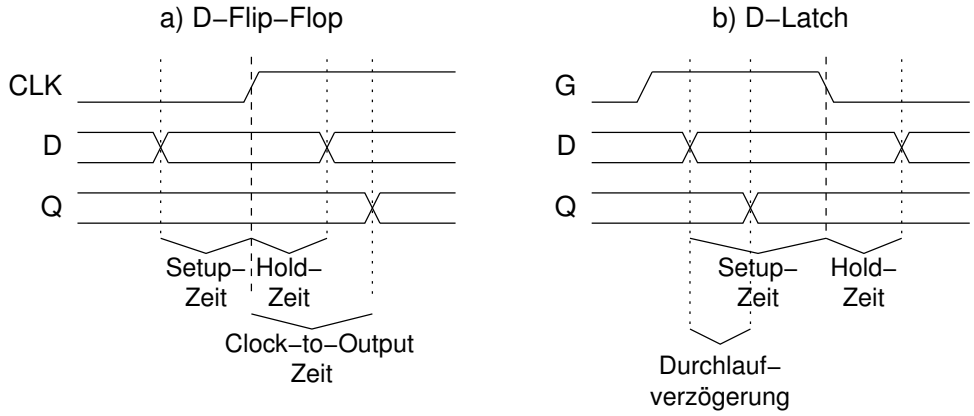


Abbildung 3.8: Zeitverhalten eines D-Flip-Flops und eines D-Latches

3.6.3 Zustandsmaschinen

Für die maximale Betriebsfrequenz einer Zustandsmaschine sind folgende Zeiten von Bedeutung:

- Die Clock-to-output-Zeit der Speicherelemente.
- Die Verzögerungszeit durch das Schaltnetz.
- Die Setup-Zeit der Speicherelemente.

Diese drei Werte ergeben addiert die minimale Zeit, die zwischen zwei positiven Taktflanken vergehen muß (Abb. 3.9). Ist aufgrund einer zu hohen Taktfrequenz die Periodenlänge zu kurz, führt dies zu einer Verletzung der Setup-Zeit am empfangenden Flip-Flop. Dadurch wird der nächste Zustand nicht zuverlässig übernommen und die Schaltung arbeitet fehlerhaft.

Für die Bestimmung der maximalen Taktfrequenz einer Gesamtschaltung muß für jeden Signalpfad die oben beschriebene Zeitsumme ermittelt werden. Der Maximalwert aller Pfade legt dann die Mindestperiodenlänge des Taktes fest.

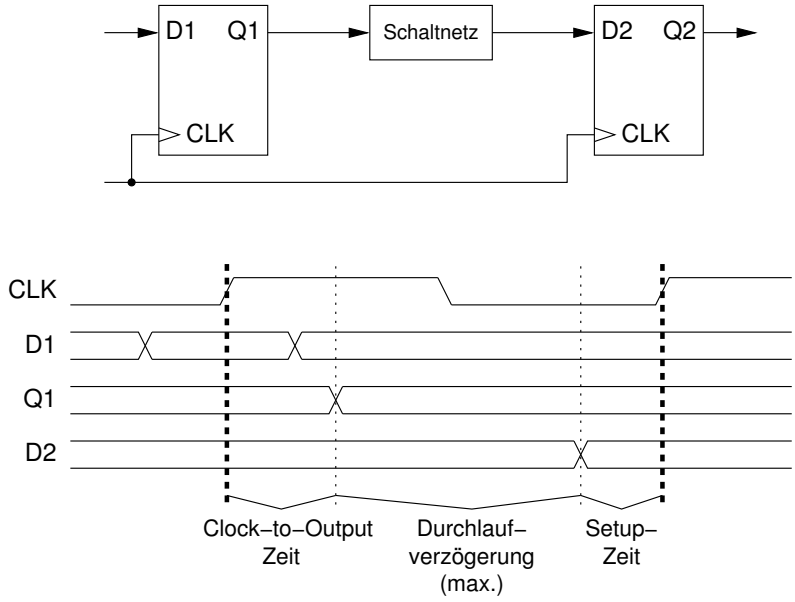


Abbildung 3.9: Zeitverhalten bei Zustandsmaschinen

3.7 Optimierung des Zeitverhaltens von Zustandsmaschinen

3.7.1 Verringerung der Logikstufen

Der Entwickler ist in der Regel bestrebt, eine möglichst hohe Taktfrequenz seiner Schaltung zu erreichen. Dazu muß er die Mindestperiodenlänge so gering wie möglich halten. Die Clock-to-output- sowie die Setup-Zeiten werden durch die eingesetzten Logikbausteine festgelegt und sind damit vorgegeben.

Die Verzögerungszeit durch das Schaltnetz wird durch die einzelnen Verzögerungszeiten der darin enthaltenen Elemente bestimmt. Diese addieren sich, wenn auf einem Signalpfad mehrere Elemente durchlaufen werden müssen.

Das in der Praxis gängigste Optimierungsverfahren besteht darin, die Schaltnetze so anzuordnen, daß möglichst wenig einzelne Logikstufen auf den Signalpfaden liegen. Bei vorgegebenen Hardwarekomponenten, so auch in FPGAs, stellt dies die einzige Einflußmöglichkeit dar, die der Entwickler auf die Durchlaufverzögerung hat.

Abbildung 3.10 zeigt zwei verschiedene Anordnungen eines Schaltnetzes, in dem eine "UND"-Verknüpfung von 10 Signalen implementiert werden soll. Dem Entwickler stehen jedoch nur einzelne "UND"-Gatter mit maximal 4 Eingängen zur Verfügung.

Anordnung a) realisiert die Schaltfunktion mit 3 Einzelkomponenten. Der längste Signalpfad von den Eingängen A, B, C und D zum Ausgang durchläuft 3 Komponenten. Angenommen, jede Komponente verursacht eine Verzögerung von 5 ns, beträgt die Gesamtverzögerung 15 ns.

Anordnung b) realisiert die Schaltfunktion mit 4 Einzelkomponenten. Der Signalpfad von allen Eingängen zum Ausgang durchläuft hier jedoch nur 2 Komponenten. Die Gesamtverzögerung beträgt 10 ns. Diese Anordnung benötigt jedoch mehr Einzelkomponenten als die vorige.

Bei einer Clock-to-output-Zeit von 2 ns und einer Setup-Zeit von 3 ns liegt die Maximalfrequenz von Anordnung a) bei 50 MHz, von Anordnung b) jedoch bei 67 MHz.

3.7.2 Sequentielle Bearbeitung von Operationen

Die Optimierung durch Verringerung der Logikstufen reicht insbesondere bei komplexen Schaltfunktionen oft nicht aus, um die gewünschte Taktfrequenz zu erreichen.

Es kann dann sinnvoll sein, die Schaltfunktion in mehrere Teilfunktionen zu zerlegen und in aufeinanderfolgenden Takten zu bearbeiten. Bei geeigneter Zerlegung weisen die einzelnen

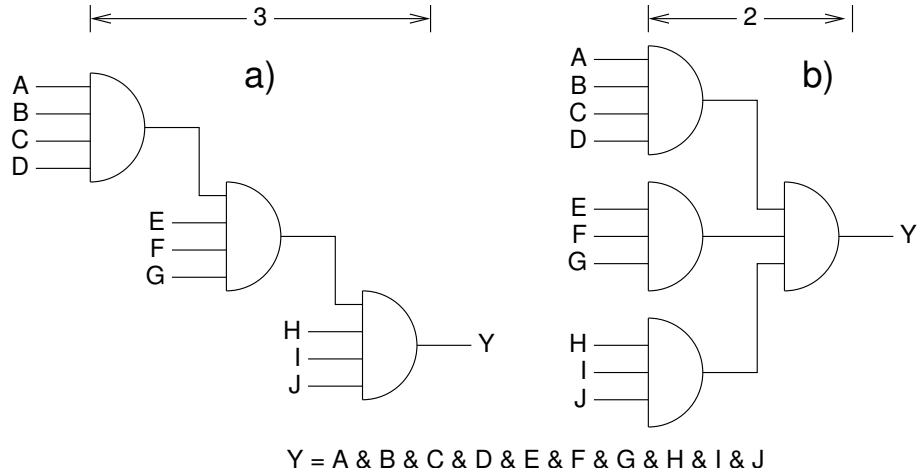


Abbildung 3.10: Optimierung von Schaltnetzen nach Logikstufen

Teilfunktionen eine geringere Durchlaufverzögerung auf.

Damit kann die mögliche Taktfrequenz erhöht werden. Dabei ist jedoch zu beachten, daß durch die zusätzlich notwendigen Takte wiederum die Gesamtausführungszeit steigt.

Angenommen, eine Schaltfunktion besitzt eine Durchlaufverzögerung von 20 ns. Clock-to-output- und Setup-Zeit betragen zusammen 5 ns. Damit liegt die maximale Taktfrequenz bei 40 MHz. Durch Zerlegung der Schaltfunktion in zwei Teile kann nun die Durchlaufverzögerung reduziert werden. Danach sind 2 Schritte für die Operation notwendig. Die Gesamtausführungszeit verkürzt sich nur, wenn durch die Zerlegung eine Taktfrequenz von mehr als 80 MHz erreicht werden kann. Dazu müsste sich die Durchlaufverzögerung jedoch auf weniger als 7.5 ns reduzieren lassen. In vielen Fällen ist der praktische Nutzen dieser Methode daher eingeschränkt.

Eine mehrtaktige Ausführung von Operationen kann jedoch auch aus anderen Gründen erforderlich sein:

- Ein sequentieller Eingangsdatenstrom zwingt die verarbeitende Logik, ebenfalls sequentiell vorzugehen.
- Resourcen, die nur in begrenzter Zahl vorhanden sind, müssen zur gemeinsamen Nutzung zeitlich aufgeteilt werden.
- Der auszuführende Algorithmus verlangt eine sequentielle Steuerung (z.B. Schleifen).
- Algorithmen können sich durch Mehrstufigkeit vereinfachen, da weniger kombinatorische Varianten notwendig sind.
- Beim Algorithmus ist der Umfang der Rechenoperation laufzeitabhängig (z.B. Prüfsummenberechnung von Datenpaketen unterschiedlicher Länge).

3.7.3 Pipelining

Wie oben erörtert, ist die Zerlegung einer Schaltfunktion in mehrere sequentielle Operationen nur bedingt geeignet, um die Gesamtausführungszeit zu reduzieren.

In vielen Fällen ist es jedoch möglich, die Aufteilung der Logik so vorzunehmen, daß die entstehenden einzelnen Teilstufen unabhängig voneinander arbeiten. Dann kann der Entwickler durch Überlappung dieser Teilstufen durchaus eine Verkürzung der Gesamtausführungszeit erreichen. Diese Methode wird als Pipelining bezeichnet [11, 48].

Pipelining stellt ein bedeutendes Verfahren dar, die maximale Taktfrequenz einer Schaltung durch Aufteilung von Schaltfunktionen zu erhöhen, ohne daß dadurch die Anzahl der notwendigen Takte durch die Zerlegung wesentlich ansteigt.

Angenommen, eine komplexe Schaltfunktion wird in 5 Teilstufen zerlegt, die nach dem Pipeline-Verfahren arbeiten können. Dann erhöht sich die Gesamtausführungszeit nicht auf das Fünffache, sondern nur um den absoluten Wert von 4 Takten. Diese sind notwendig, um am Ende die letzten 4 Ergebniswerte aus der Pipeline auszulesen (*pipeline flush*).

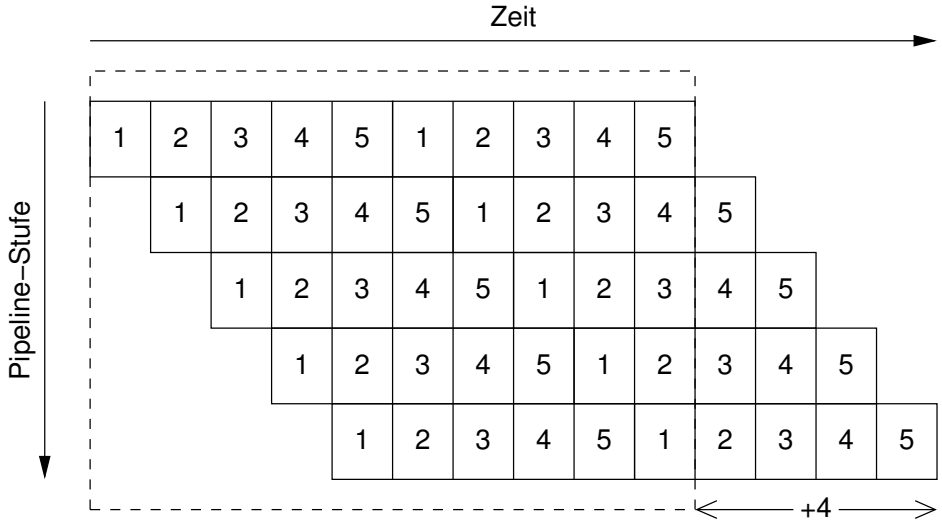


Abbildung 3.11: Pipeline-Verfahren

Voraussetzung für das Pipeline-Verfahren ist, daß sich die einzelnen Teilstufen unabhängig voneinander konstruieren lassen. Um dies zu erreichen, kann es notwendig sein, den auszuführenden Algorithmus deutlich umzustrukturieren.

3.7.4 Zerlegung von Zustandsmaschinen

Die Komplexität der Schaltnetze, mit denen Zustandsmaschinen realisiert werden, ergibt sich aus den implementierten Zustandsübergängen sowie den in den Bedingungen verwendeten Signalen. Sind die Zustände durch viele Übergangsbedingungen miteinander vernetzt, können sich Schaltnetze mit hohen Durchlaufverzögerungen ergeben. Aber auch komplexe Berechnungen, die in den einzelnen Übergangsbedingungen ausgeführt werden müssen, können die Komplexität erhöhen.

In solchen Fällen kann es sinnvoll sein, eine komplexe Zustandsmaschine in mehrere einfachere Maschinen zu zerlegen, die durch geeignete Kommunikation untereinander die Gesamtaufgabe ausführen (*statemachine decomposition*) [11, 4]. Dabei können einzelne dieser Maschinen so einfach werden, daß sie nur noch aus einem oder wenigen Flip-Flops bestehen und eventuell ausschließlich Verzögerungsfunktion haben.

Dieses Verfahren erfordert in der Regel eine genaue Analyse des Ablaufes bzw. geeignete Änderungen am Algorithmus. Die entstehende Anordnung kann dadurch unübersichtlicher werden, wodurch sich nachträgliche Änderungen schwieriger durchführen lassen.

Kapitel 4
Field Programmable Gate Arrays (FPGAs)

4.1 Einführung

***F**ield **P**rogrammable **G**ate **A**rrays* (FPGAs) sind durch den Anwender programmierbare Logikbausteine, die eine definierte Anzahl konfigurierbarer logischer Blöcke enthalten. Über allgemeine Verbindungsresourcen können diese Blöcke miteinander verschaltet werden [105, 15].

Als die Firma *XILINX* 1985 die ersten FPGAs auf den Markt brachte, wurden diese mit ihren damals unter 1000 Gatteräquivalenten von den Entwicklern zunächst als "Spielzeugbausteine" belächelt [105]. Sowohl in Preis, Geschwindigkeit und Stromverbrauch waren sie den bereits existierenden ***P**rogrammable **L**ogic **D**evices (PLDs)* eindeutig unterlegen.

Doch die Logikdichte und die Geschwindigkeit der FPGAs wurden ständig verbessert. Inzwischen stellen sie eines der am schnellsten wachsenden Marktsegmente der gesamten Halbleiterindustrie dar.

FPGAs beruhen auf dem Prinzip, daß sich aus den beiden Grundelementen Lookup-Tabellen und Flip-Flops nahezu jede beliebige digitale Schaltung realisieren läßt (Abb. 4.1).

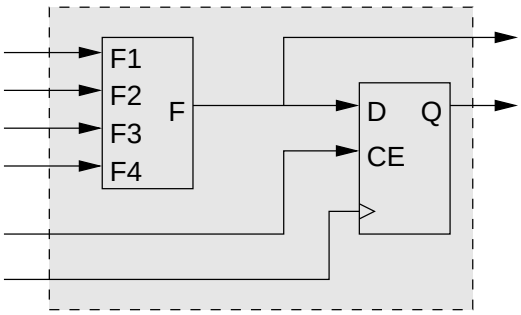


Abbildung 4.1: Grundprinzip der FPGAs

Der FPGA-Markt wird zur Zeit von zwei Firmen beherrscht: *XILINX* und *Altera*. Daneben gibt es noch einige Firmen, bei denen FPGAs nur einen Unterbereich der Produktpalette bilden: *Atmel*, *Lattice* (hat den FPGA-Bereich von *Lucent* übernommen) und *Actel*.

4.2 Unterschiede zwischen FPGAs und konventionellen PLDs

Die heute verfügbaren FPGAs unterscheiden sich in drei wesentlichen Punkten von konventionellen PLDs: Erstens in der Anordnung der einzelnen Logikblöcke, zweitens in der Art, wie kombinatorische Logik mit den vordefinierten Elementen realisiert wird, und drittens in der Technologie der Konfigurationszellen.

4.2.1 Anordnung der einzelnen Logikblöcke

Bei PLDs werden Verbindungen zwischen den Logikblöcken durch eine zentrale Schaltmatrix geführt. Jeder Ausgang eines Blockes kann mit nahezu jedem beliebigen Eingang eines anderen Blockes verbunden werden. Die geometrische Lage der beteiligten Blöcke zueinander auf dem Chip hat dabei weder für das Routing der Signale noch für das Zeitverhalten eine Bedeutung (Abb. 4.2 a). Das Problem einer zentralen Schaltmatrix besteht jedoch darin, daß ihre Größe mit zunehmender Anzahl von Logikblöcken quadratisch ansteigen muß, um die Verbindungsfähigkeit der Blöcke untereinander zu gewährleisten. Dies schränkt in der Praxis die mit PLDs erreichbare Logikdichte ein.

FPGAs dagegen verwenden eine grundsätzliche andere Anordnung. Hier existiert keine zentrale Schaltmatrix. Vielmehr sind die Blöcke gleichmäßig über die Chipfläche in Matrixanordnung verteilt. Zur Verschaltung untereinander existieren Verbindungsresourcen in den Bereichen zwischen den Blöcken. Über programmierbare Schalter können damit Verbindungswege zwischen Blöcken geschaltet werden (Abb. 4.2 b).

Im Gegensatz zu PLDs ist bei FPGAs die Lage der Logikblöcke zueinander relevant. Zum einen bestimmt sich durch die Lage der geometrische Verlauf der erforderlichen Verbindungswege. Ein Signal zwischen nicht direkt benachbarten Zellen muß über geeignete vorhandene Wege geführt werden.

Zum anderen hat diese Routing-Problematik Auswirkungen auf das Zeitverhalten der zu realisierenden Schaltung. Die Durchlaufverzögerung durch die Verbindungsresourcen ist umso größer, je größer der Abstand der zu verschaltenden Blöcke ist und mehr programmierbare Schalter dabei durchlaufen werden müssen. Konkrete Aussagen über das Zeitverhalten sind somit immer erst nach Abschluß des Place&Route-Verfahrens möglich.

Diese Abhängigkeit des zeitlichen Verhaltens von der Logikplazierung und dem Routing der Verbindungen stellt ein großes Problem der FPGA-Technologie dar. Es hat zur Folge, daß der Entwickler bei der Erstellung umfangreicher Designs immer auch das Zeitverhalten beachten muß. Die maximale Taktfrequenz, mit der die Schaltung letztendlich betrieben werden kann, ist dann oft von Einzelheiten der Implementierung abhängig. Ein einziger kombinatorischer Logikpfad mit langer Durchlaufverzögerung reicht aus, um die gesamte Schaltung "auszubremsen".

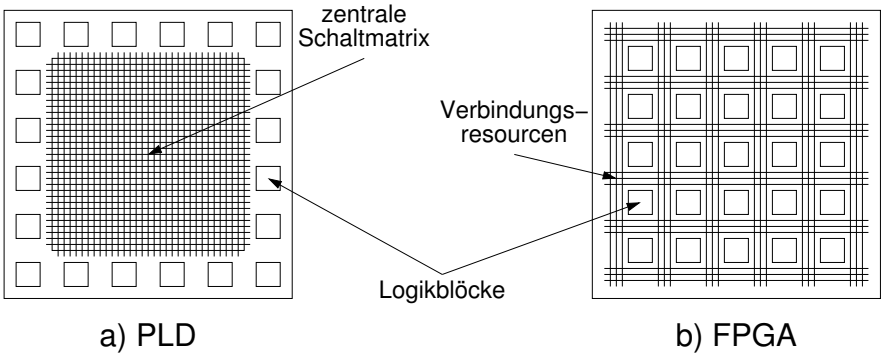


Abbildung 4.2: Anordnung der Logikblöcke bei PLDs und FPGAs

4.2.2 Realisierung kombinatorischer Logik

PLDs verwenden zur Implementierung kombinatorischer Logik konfigurierbare Produkttermanordnungen (Abb. 4.3). Dieses Verfahren beruht auf dem Prinzip, daß jede Schaltfunktion in eine ODER-Verknüpfung von UND-Termen umgeformt werden kann. Je komplexer die Funktion ist, desto mehr Produktterme werden dabei benötigt.

Sowohl die Anzahl der Eingangssignale als auch die Anzahl der ODER-kombinierbaren Produktterme pro Logikblock sind bei PLDs begrenzt. Werden diese Grenzen durch eine umfangreichere oder komplexere Funktion überschritten, muß sie geeignet aufgespalten und auf mehrere Logikblöcke verteilt werden. Dadurch summieren sich für das Zeitverhalten die Verzögerungszeiten der einzelnen Teilfunktionen. Aufgrund der zentralen Schaltmatrix sind die Auswirkungen jedoch leicht abschätzbar.

Die in FPGAs enthaltenen Lookup-Tabellen besitzen in der Regel vier Eingänge. Eine solche Lookup-Tabelle kann jede beliebig komplexe Schaltfunktion mit bis zu vier Signalen realisieren. Funktionen, die mehr als vier Signale besitzen, müssen auf mehrere Lookup-Tabellen verteilt werden. Bei einer solchen Verteilung ist dann wieder die geometrische Anordnung sowohl für das Routing als auch für das Zeitverhalten von Bedeutung.

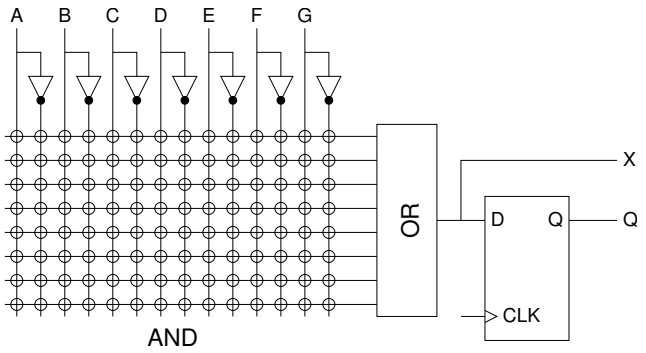


Abbildung 4.3: Aufbau einer PLD-Logikzelle

4.2.3 Technologie der Konfigurationszellen

PLDs speichern ihre Konfigurationsinformationen in EEPROM-Zellen. Diese gestatten technologiebedingt nur eine begrenzte Anzahl von Schreibvorgängen. Je nach Architektur können PLDs ca. 1000 bis 1000000 mal neu konfiguriert werden. Die Zeit, die zum Programmieren einer EEPROM-Zelle benötigt wird, liegt in der Größenordnung von zehn Millisekunden. Der Programmiervorgang muß zum Teil sequentiell ablaufen, so daß die Gesamtzeit für einen kompletten Baustein mehrere Sekunden beträgt.

FPGAs dagegen verfügen über SRAM-Zellen und damit über eine deutlich schnellere und vor allem unbegrenzte Wiederprogrammierbarkeit. Der Nachteil der SRAM-Zellen liegt vor allem im Verlust der gesamten Konfigurationsinformation beim Abschalten der Versorgungsspannung. Das bedeutet, daß FPGAs nach jedem erneuten Anlegen der Versorgungsspannung neu konfiguriert werden müssen.

4.2.4 Bedeutung für den Einsatzbereich von FPGAs

Für den Einsatzbereich von FPGAs ergeben sich aus den oben genannten Unterschieden zu den PLDs einige bedeutende Konsequenzen:

Durch das Fehlen einer zentralen Schaltmatrix unterliegen FPGAs nicht den Einschränkungen der PLDs bezüglich der erreichbaren Logikdichte. Die Struktur der FPGAs kann theoretisch beliebig in beiden Dimensionen ausgedehnt werden, ohne daß dabei einzelne Komponenten wie etwa eine zentrale Schaltmatrix überpropertional vergrößert werden müssen. Somit ist die erreichbare Logikdichte in der Praxis nur von der verwendeten Herstellungstechnologie abhängig. Da die Herstellungsprozesse von der Halbleiterindustrie ständig weiterentwickelt werden, ist absehbar, daß die Logikdichte der FPGAs ebenfalls weiter ansteigen wird. FPGAs können jede Verkleinerung der realisierbaren Halbleiterstrukturen durch ihre zweidimensionale Anordnung sofort quadratisch nutzen. Eine Halbierung der Strukturbreiten würde folglich zu einer Vervierfachung der Logikdichte führen.

Dem Anwender von FPGAs werden somit immer leistungsfähigere Bausteine zur Verfügung stehen, in denen sich immer umfangreichere und komplexere Schaltungen realisieren lassen. Anwendungsbereiche, die vor einigen Jahren im FPGA-Bereich aufgrund Anzahl der erforderlichen Logikresourcen noch undenkbar waren, sind inzwischen realisierbar geworden.

Durch die unbegrenzte Wiederprogrammierbarkeit wird zudem die Konstruktion sogenannter FPGA-Koprozessoren möglich. Dieselbe Hardware kann hier durch Rekonfiguration für verschiedenste Algorithmen beliebig oft eingesetzt werden. In Verbindung mit einem Mikroprozessor können FPGA-Koprozessoren zur Beschleunigung konventioneller Softwarealgorithmen verwendet werden. Auch dieses Anwendungsgebiet wird durch die steigende Logikdichte der FPGAs ständig erweitert.

Für den Anwender von FPGA-Koprozessoren bedeutet dies jedoch auch, daß Umfang und Komplexität seiner Schaltungen ständig zunehmen werden. Dies zu beherrschen, wird eine wichtige Aufgabe der Entwicklungssoftware sein.

Das zeitliche Verhalten der FPGA-Schaltung ist von der Logikplazierung und dem Routing der Verbindungen abhängig. Wie erwähnt, kann ein einziger langsamer Logikpfad das gesamte Design "ausbremsen". Gerade in dieser globalen Auswirkung zeigt sich ein besonderes Problem der FPGA-Programmierung gegenüber der konventionellen sequentiellen Ausführung von Softwarealgorithmen: Zeigt die Optimierung eines Softwarekompilers Schwächen, so beschränken sich diese auf die betreffenden Anweisungen, die entsprechend ineffizienter ausgeführt werden. Es ist aber nicht global das gesamte Programm betroffen. Schwächen eines Hardwarekompilers dagegen können ohne weiteres Auswirkungen auf das gesamte Design besitzen.

Dieses Problem ist besonders in den Bereichen relevant, in denen FPGAs mit ***A**pplication **S**pecific **I**ntegrated **C**ircuits* (ASICs) konkurrieren. FPGAs haben gegenüber der ASIC-Technologie sowohl in den erreichbaren Taktfrequenzen als auch in der erreichbaren Logikdichte Nachteile. Die programmierbaren Schalter der FPGAs besitzen einen Widerstand und eine Kapazität. Dadurch weisen die Verbindungen in FPGAs eine etwa um den Faktor 3 höhere Verzögerungszeit auf als die metallisierten Verbindungen in ASICs. Da hinter jedem programmierbaren Schalter eine SRAM-basierte Konfigurationszelle liegt, ist die Dichte der für den Anwender verfügbaren Logik geringer. So erreichen ASICs gegenüber FPGAs eine 8- bis 12-fach höhere Dichte [15].

Um die praktischen Auswirkungen dieser technologischen Nachteile in Grenzen zu halten, müssen dem FPGA-Anwender Möglichkeiten zur Erstellung effizienter Designs zur Verfügung stehen. Die zu implementierenden Schaltungen müssen sowohl hinsichtlich Resourcenverbrauch und Zeitverhalten effizient optimiert werden können.

4.3 Architektur der *XILINX*-FPGAs

4.3.1 Allgemeines

Der FPGA-Marktführer *XILINX* bietet FPGAs unterschiedlicher, aber dennoch ähnlicher Architektur an. In der folgenden Darstellung wird die Struktur der *Virtex*-FPGAs vorgestellt. Diese Bausteine, zusammen mit der Low-Cost-Variante *Spartan-II*, stellen die zur Zeit vorherrschende FPGA-Architektur dar. Weitere Architekturen sind: *XC4000E*, *Spartan*, *Virtex-II* und *Virtex-II PRO* [122, 114, 115, 119, 120, 121].

Die wesentlichen konfigurierbaren Elemente sind die ***C**onfigurable **L**ogic **B**locks* (CLBs), die ***I**nput-**O**utput **B**locks* (IOBs) sowie die separaten Block-RAMs (*SelectRAM*):

- Die CLBs beinhalten die Elemente, mit denen der Anwender seine Logik implementieren kann: Lookup-Tabellen und Speicherelemente.
- Die IOBs stellen die Schnittstelle zwischen den internen Routing-Resourcen und den Gehäusepins dar.
- Die Block-RAMs sind spezielle große Speicherelemente innerhalb der CLB-Matrix. Sie können als RAMs, ROMs oder als Lookup-Tabellen verwendet werden und besitzen eine konfigurierbare Datenbreite.

Die CLBs sind im Hauptbereich des Chips in Matrixform angeordnet, die IOBs befinden sich am Rand der Matrix (Abb. 4.4). Flexible Verbindungsresourcen ermöglichen die Verschaltung von CLBs, IOBs und Block-RAMs zu der vom Anwender gewünschten Implementierung.

4.3.2 Configurable Logic Blocks (CLBs) und Slices

Ein CLB besteht aus zwei identischen Elementen (*Slices*) und enthält insgesamt vier Lookup-Tabellen sowie vier Speicherelemente (Abb. 4.5).

Die Lookup-Tabellen können als unabhängige Funktionsgeneratoren mit jeweils bis zu vier Eingängen betrieben werden. Durch schnelle interne Verbindungsresourcen lassen sich diese auch kombinieren, um Lookup-Tabellen mit 5 oder 6 Eingängen zu bilden. Darüberhinaus können sie als RAM, ROM oder Dual-Port-RAM verwendet werden (*Distributed RAM*, Abb. 4.6).

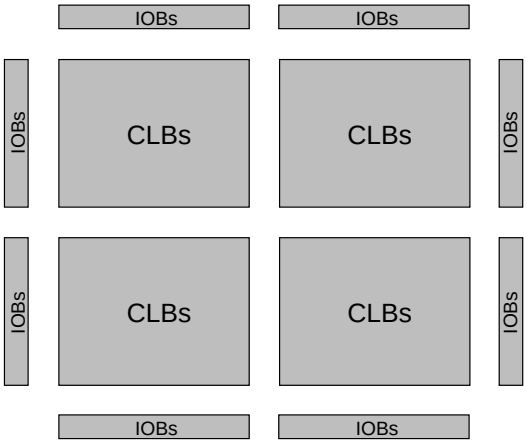


Abbildung 4.4: *Virtex* Blockdiagramm

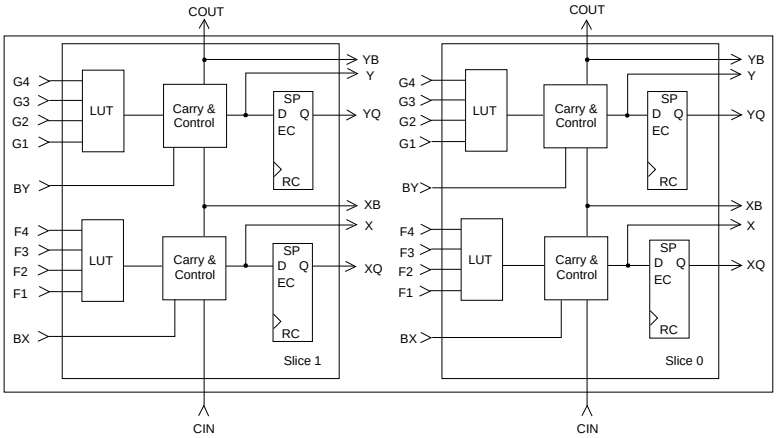


Abbildung 4.5: *Virtex* / *Spartan-II*: Configurable Logic Block

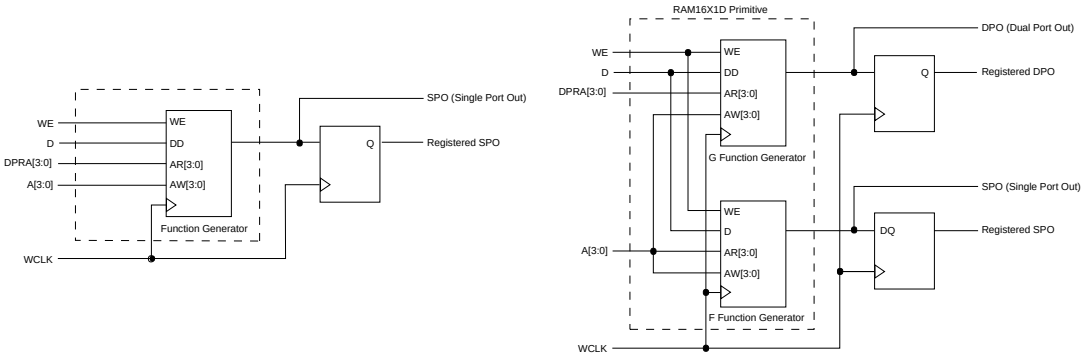


Abbildung 4.6: *Virtex* Distributed RAM

Die Speicherelemente lassen sich als D-Flip-Flops oder D-Latches konfigurieren.

Weiterhin enthalten die CLBs eine Carry-Chain-Logik zum Aufbau von breiten Arithmetikfunktionen, wie etwa Addierern und Zählern. Sie verfügt über eigene Verbindungsresourcen, die geringe Verzögerungszeiten im Carry-Pfad ermöglichen.

4.3.3 I/O Blocks (IOBs)

Die IOBs befinden sich am Rand der CLB-Matrix und stellen die Schnittstelle zwischen den internen Routing-Resourcen und den Gehäusepins dar. Jeder IOB enthält einen Eingangstrei-

ber, einen Ausgangstreiber sowie ein Output-Enable-Signal zum Abschalten des Ausgangstreibers. Weiterhin sind drei Speicherelemente vorhanden, die in den Datenpfad geschaltet werden können (Abb. 4.7).

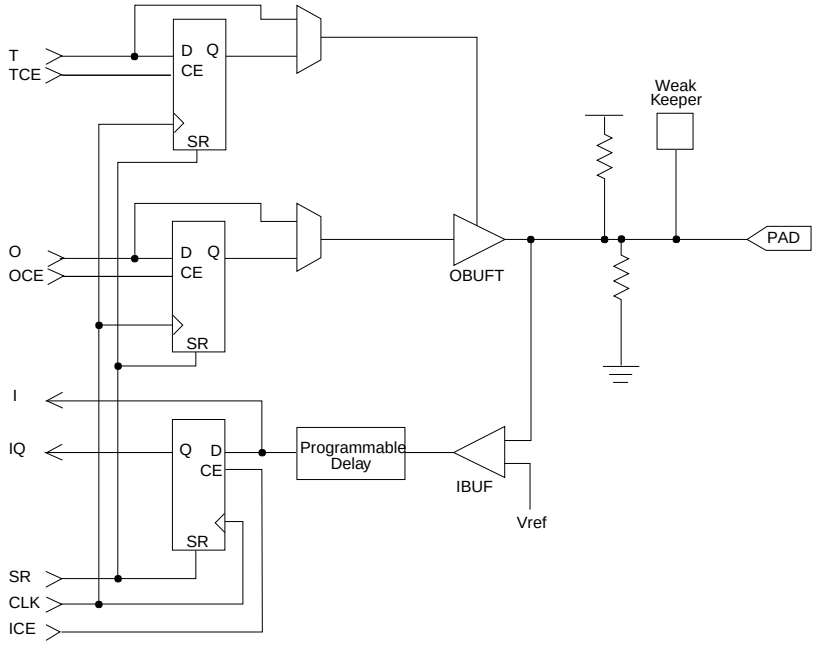


Abbildung 4.7: *Virtex* I/O Block

4.3.4 Block-RAMs (*SelectRAM*)

In die CLB-Matrix sind spezielle Block-RAMs integriert (Abb. 4.8). Diese können in verschiedenen Konfigurationen (256x16 bis 4096x1) als Single-Port- oder Dual-Port-RAM genutzt werden.

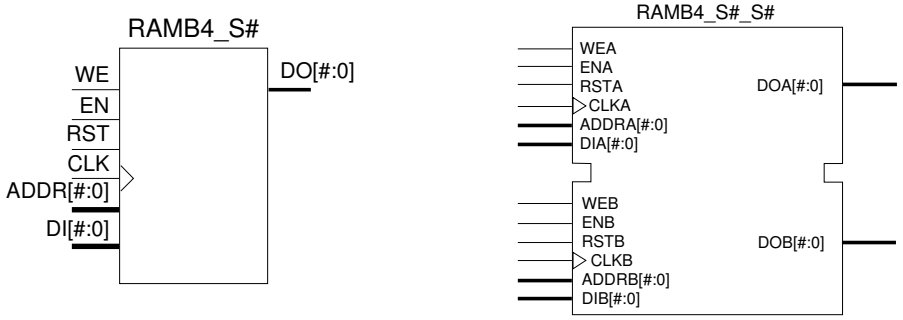


Abbildung 4.8: *Virtex SelectRAM*

Diese RAMs stellen eigene Elemente dar, die keine Resourcen der CLB-Logik belegen. Sie besitzen den Vorteil, daß sie große Speicherkapazitäten zur Verfügung stellen, die ansonsten durch Zusammenschaltung vieler einzelner Lookup-Tabellen der CLBs gebildet werden müssten.

4.4 Anforderungen an die Entwicklungswerkzeuge

Aus den vorhandenen Komponenten und den zahlreichen Konfigurationsmöglichkeiten ergibt sich eine große Flexibilität der FPGAs. Es wird aber auch deutlich, welche Fülle von Detailinformationen bei der Designerstellung festgelegt werden muß. Einerseits bieten FPGAs durch die direkte Kontrolle der Resourcen die Möglichkeit, effiziente handoptimierte Schaltungen

zu realisieren. Andererseits werden umfangreiche und komplexe Schaltungen durch die erforderliche Menge von Details schnell unübersichtlich.

Es sind daher Entwicklungswerkzeuge gefordert, die eine Implementierung von Details auch automatisch vorzunehmen können. Daneben muß jedoch der direkte Zugriff auf die Resourcen weiterhin möglich bleiben, um in Problemfällen auch die effizientere manuelle Konfiguration zu erlauben.

Deshalb ist es sehr wichtig, daß die Entwicklungswerkzeuge die Hardwareumsetzung mit nachvollziehbaren Verfahren vornehmen. Nur so kann der Entwickler bei der Planung und Erstellung von FPGA-Designs den Überblick über die Resourcenverwendung behalten.

Damit stehen die Entwicklungswerkzeuge vor der Aufgabe, einerseits in ihrer Beschreibungsform dem Anwender die Möglichkeit einer direkten Einflußnahme auf die FPGA-Resourcen bieten. Andererseits müssen sie Hardwarebeschreibungen auf hohen Abstraktionsebenen erlauben, um Umfang und Komplexität zu beherrschen und den Anwender von Details zu entlasten. Die dabei angewandten automatischen Verfahren sollen effizient und nachvollziehbar sein.

Die sich hieraus ergebenden Probleme und Lösungsmöglichkeiten werden später in eigenen Kapiteln behandelt.

Kapitel 5
FPGA-Koprozessoren

5.1 Einführung

Ein hochaktuelles Einsatzgebiet für FPGAs sind die FPGA-Koprozessoren. Hierbei handelt es sich um FPGA-basierte Komponenten, die eng mit einem konventionellen Mikroprozessor zusammenarbeiten und beispielsweise als PCI-Einsteckkarten realisiert sind.

Die Grundidee der FPGA-Koprozessoren besteht darin, Teile eines Gesamtalgorithmus, die sich effizienter in einer direkten Hardwareimplementierung lösen lassen, auf dem FPGA zu realisieren. Der restliche Algorithmus wird konventionell vom Mikroprozessor bearbeitet. Die unbegrenzte Wiederprogrammierbarkeit der FPGAs ermöglicht dabei, dieselbe Hardware für die verschiedensten Anwendungen beliebig oft einzusetzen.

Vergleicht man die erreichbaren Taktfrequenzen moderner Mikroprozessoren (zur Zeit zwischen 2 und 3 GHz) und FPGAs (je nach konkreter Implementierung 60 - 120 MHz), so stellt sich zunächst die Frage, warum FPGAs so ideal zur Beschleunigung von Algorithmen eingesetzt werden können. Die Mikroprozessoren scheinen durch ihre hohe Taktfrequenz und ihre intensive Weiterentwicklung der letzten Jahre den FPGAs weit überlegen zu sein.

Im Gegensatz zu Mikroprozessoren, die eine universelle Innenschaltung besitzen müssen, kann bei FPGAs die Hardware den einzelnen Algorithmen speziell angepaßt werden. Dadurch können FPGAs schneller und effizienter sein als Mikroprozessoren mit ihrer durchschnittlichen Leitungsfähigkeit.

Da sich FPGAs beliebig oft neu konfigurieren lassen, besitzen sie dennoch die gleiche Flexibilität wie Mikroprozessoren.

Das Prinzip, einen universellen Prozessor durch spezielle Hardware zu unterstützen, wird bereits bei numerischen Koprozessoren oder den 3D-Grafikprozessoren moderner PC-Grafikkarten genutzt.

Aktuelle Einsatzgebiete von FPGA-Koprozessoren sind z.B. Bildverarbeitung, Verschlüsselung, Datenkompression, schnelle Mustererkennung sowie schnelle Realisierung von Prototypen [15].

5.2 Das Prinzip der FPGA-Koprozessoren

Ein typischer rekonfigurierbarer Koprozessor, wie ihn Abbildung 5.1 zeigt, beinhaltet [58]:

- Einen oder mehrere FPGAs.
- Für jeden FPGA einen lokalen Speicher (RAM).
- Ein flexibles und konfigurierbares Taktsystem.
- Ein flexibles und schnelles Interface zur Rekonfiguration der FPGAs.
- Eine flexible und erweiterbare externe I/O-Schnittstelle.
- Ein schnelles Mikroprozessorinterface mit der Möglichkeit, direkte Schreib- und Lesezugriffe auf den FPGA über Pseudoregister oder DMA-Kanäle vorzunehmen, die sich im Adreßbereich des Mikroprozessors befinden.

FPGA-Koprozessoren lassen sich auf sehr unterschiedliche Weise realisieren. Insbesondere die Art und die Größe der lokalen Speicherbausteine (SRAM oder SDRAM) sowie die Implementierung des Taktsystems variieren stark bei den verfügbaren Systemen.

Die Grundstrukturen sind jedoch ähnlich. Sie ergeben sich aus der begrenzten Menge existierender Bussysteme sowie der Notwendigkeit eines schnellen Datentransfers.

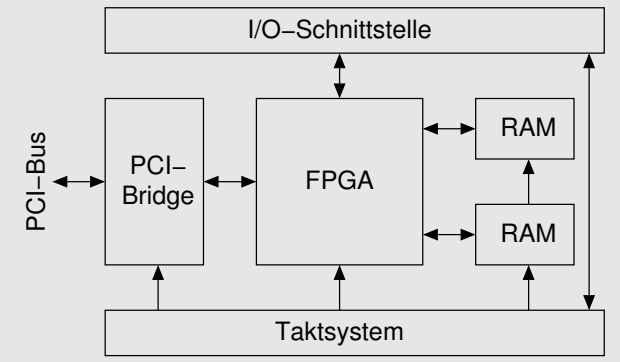


Abbildung 5.1: Struktur eines FPGA-Koprozessors

Die konventionelle Mikroprozessortechnik basiert auf der *von-Neumann*-Architektur. Diese besitzt einige kritische Engstellen, die den Nutzen der hohen Taktfrequenz in der Praxis einschränken (*von-Neumann-bottleneck*) [101, 29]. So arbeiten Mikroprozessoren zwar intern mit Takten von mehreren GHz und nutzen zusätzlich Verfahren wie Superpipelining oder paralleler Verarbeitung [48]. Die Datenschnittstelle des Prozessors nach außen, vor allem zum Hauptspeicher, erlaubt jedoch nur Frequenzen von einigen Hundert MHz. Zudem müssen sich alle Recheneinheiten zum Lesen und Schreiben der Daten diese Schnittstelle teilen.

Um diese Engstellen zu überwinden, wurden Prozessorarchitekturen vorgeschlagen, die eine rekonfigurierbare Erweiterung auf dem Chip besitzen [78].

FPGA-Koprozessoren können durch ihre vollständige Rekonfigurierbarkeit jedoch eine weitaus höhere Leistungssteigerung erreichen. Hier ist es möglich, zeitkritische Teile eines Algorithmus unabhängig von der Universalhardware eines Mikroprozessors völlig selbständig durch eine synchrone Schaltung auszuführen. Dadurch können spezielle Recheneinheiten konstruiert, Berechnungen parallelisiert und zusätzliche Verfahren wie etwa Pipelining eingesetzt werden. Außerdem lassen sich mehrere unabhängige I/O-Schnittstellen realisieren.

Auf diese Weise kann eine FPGA-Implementierung trotz ihrer niedrigeren Taktfrequenz schneller sein als ein Mikroprozessor.

FPGA-Koprozessoren werden sowohl von universitären Einrichtungen (z.B. Universität Mannheim: *RACE-1* [102], FZI Darmstadt: *Spyder* [33]) als auch von kommerziellen Unternehmen (z.B. Silicon Software GmbH: *microEnable* [85], Annapolis Microsystems Inc: *WILDFORCE* [3]) entwickelt.

5.3 Hardware-Software-Codesign

Während bei der konventionellen Programmierung ein Algorithmus nur durch Softwareimplementierung realisiert wird, erfordert ein FPGA-Koprozessor zusätzlich die Implementierung der Hardware. Ein FPGA-Entwickler muß neben der Software- und Hardwareprogrammierung auch die Schnittstelle zwischen den beiden Bereichen beherrschen.

Einer der ersten Schritte bei der Implementierung eines Algorithmus für einen FPGA-Koprozessor besteht in der Regel darin, aus der Software diejenigen Teilbereiche zu extrahieren, die sich für eine Implementierung im FPGA eignen. Dieser Vorgang wird als "Partitionierung" bezeichnet. Es kann dabei eventuell auch notwendig sein, die Software zu modifizieren.

Weil die Implementierung von Hardware wesentlich aufwendiger und schwieriger zu warten ist, wird üblicherweise nicht der gesamte Algorithmus im FPGA umgesetzt. Teile, die nicht zeitkritisch sind, können weiterhin durch Software realisiert werden.

Das Ziel des Hardware-Software-Codesigns ist nicht, den Prozessor durch eine eigene Hardware zu ersetzen, sondern dessen universelle Architektur gezielt zu ergänzen.

Eines der Hauptprobleme bei diesem Konzept stellt das effiziente Zusammenwirken zwischen Prozessor und FPGA dar. Dies gilt besonders, wenn große Datenmengen ausgetauscht werden müssen.

5.3.1 Pseudoregister (*Special Function Registers, SFRs*)

Ein Standardverfahren zum Datenaustausch zwischen einem Mikroprozessor und einem externen Hardwarebaustein stellen Pseudoregister dar. Diese befinden sich im externen Baustein und bilden die Schnittstelle zwischen dem Bussystem und den internen Signalen des Bausteins [74].

Aus Sicht eines Mikroprozessors ist ein Pseudoregister ein Element, das sich auf einer bestimmten Adresse des Adressraums befindet und wie ein Teil des Hauptspeichers beschrieben und gelesen werden kann.

Pseudoregister realisieren jedoch kein Speicherelement, sie können vielmehr einen Datenkanal darstellen. Deshalb ist in der Regel das Ergebnis eines Lesevorganges nicht mit einem zuvor geschriebenen Wert identisch. Ebenso werden zwei aufeinanderfolgende Lesezugriffe nicht in jedem Fall den gleichen Wert zurückliefern.

Aus Sicht der Software und des Mikroprozessors ergibt sich zunächst kein Unterschied zwischen einem Zugriff auf eine Adresse des Hauptspeichers und einem Zugriff auf ein Pseudoregister:

```
char* ptr = SFR_ADDR;

*ptr = 0x12;     // Schreiben auf SFR
cout << *ptr;    // Lesen von SFR
```

Allerdings unterscheiden sich die konkreten Auswirkungen von Zugriffen auf Pseudoregister deutlich von denen auf normale Speicherelemente:

- Die Zugriffsreihenfolge kann relevant sein.

  Beim Zugriff auf unabhängige Hauptspeicheradressen ist die Reihenfolge unerheblich. Die Reihenfolge bei Zugriffen auf verschiedene Pseudoregister kann, abhängig von der konkreten Implementierung, jedoch erhebliche Bedeutung haben.

- Wiederholte Zugriffe können relevant sein.

  Bei jedem Schreibzugriff auf eine Hauptspeicheradresse geht der dort zuvor gespeicherte Wert verloren. Bei mehreren aufeinanderfolgenden Schreibvorgängen ohne zwischenzeitlichem Auslesen ist somit nur der letzte Vorgang relevant. Schreibzugriffe auf ein Pseudoregister können jedoch alle relevant sein, insbesondere, wenn dieses einen Datenkanal darstellt. Ähnliches gilt für Lesezugriffe: Aufeinanderfolgendes Lesen einer Hauptspeicheradresse liefert stets den gleichen Wert, nicht aber Zugriffe auf Pseudoregister.

- `memcpy()` und `memmove()` können Fehler verursachen.

  Beim Schreiben und Lesen unabhängiger Hauptspeicheradressen ist die Reihenfolge unerheblich. In manchen Laufzeitbibliotheken wird die Reihenfolge bewußt abgeändert, um eine schnellere Ausführung zu erreichen. Bei einer `memmove()`-Operation mit sich überschneidenden Speicherbereichen kann zudem die gesamte Operation von hinten nach vorne ablaufen. Aus diesen Gründen kann die Verwendung von `memcpy()` und `memmove()` bei Pseudoregistern Fehler verursachen.

- Deaktivierung des Cache ist erforderlich.

  Für den Bereich des Addressraumes, in dem sich die Pseudoregister befinden, muß der Cache deaktiviert werden. Die Cache-Funktion steht im Widerspruch zum Prinzip der Pseudoregister und würde deren Funktion stören.

Diese Besonderheiten der Pseudoregister stehen im Gegensatz zu den Voraussetzungen, auf denen optimierende Kompiler aufbauen. Diese versuchen, Code zu optimieren, indem nichtrelevante Zugriffe vermieden, Zugriffsreihenfolgen angepasst und häufig verwendete Werte temporär in Prozessorregistern gehalten und daher nicht erneut geladen werden.

Im obigen Programmbeispiel würde daher bei aktivierter Optimierung kein Lesezugriff auf das Pseudoregister erfolgen, sondern der noch vorhandene temporäre Wert direkt weiterverwendet werden. Beim Einsatz von Pseudoregistern muß folglich zwingend die Optimierung abgeschaltet werden. Nahezu alle Kompiler ermöglichen dies selektiv mit dem Schlüsselwort `volatile`.

Der korrekte Code für den Zugriff auf Pseudoregister lautet daher:

```
volatile char* ptr = SFR_ADDR;

*ptr = 0x12;     // Schreiben auf SFR
cout << *ptr;    // Lesen von SFR
```

5.3.2 Direct Memory Access (DMA)

Der DMA-Transfer stellt ein Verfahren dar, um effizient große Datenmengen über ein Bussystem ohne Mitwirkung des Prozessors zu übertragen. Rechenzeit des Prozessors wird nur zum Starten des Transfers benötigt, nicht aber zu dessen Durchführung. Während der Datenübertragung kann der Prozessor somit andere Aufgaben ausführen [6].

Es können zwei Arten des DMA-Transfers unterschieden werden [72]:

- DMA ohne zusätzliche Flußsteuerung.

  Der Transfer wird mit maximaler Geschwindigkeit durchgeführt. Unterbrechungen können nur durch Steuerleitungen des Bussystems (Bus-Ready Signale) ausgelöst werden. Diese Transferart kann mit einer `memcpy()`-Operation ohne Prozessorbeteiligung verglichen werden. Der Vorteil im Zusammenhang mit Pseudoregistern besteht darin, daß beim DMA die Zugriffsreihenfolge eindeutig definiert ist.

- DMA-on-demand.

  Hier existiert eine zusätzliche Flußsteuerung über das `DREQ`-Signal. Ein Transfer erfolgt nur bei aktivem Signal. Dies stellt ein sehr effizientes Verfahren dar, um den Transferablauf von der externen Hardware zu kontrollieren, ohne daß Rechenzeit des Prozessors etwa durch ein kontinuierliches Abfragen von Statussignalen (*Polling*) oder durch Interrupts benötigt wird.

5.4 Anwendungstypen von FPGA-Koprozessoren

Abhängig von der Herkunft der zu verarbeitenden Daten lassen sich bei FPGA-Koprozessoren zwei Anwendungstypen unterscheiden:

- Die Daten stammen von einem externen Gerät.

  Der Koprozessor erhält die zu verarbeitenden Daten von einem externen Gerät, z.B. einer Kamera (Abb. 5.2). Im FPGA werden in Echtzeit, also mindestens mit der Geschwindigkeit der eingehenden Daten, verschiedene Rechenoperationen ausgeführt. Solche Operationen können etwa Bildformatkonvertierung, Änderung des Wertebereiches, Auswahl von Bildausschnitten oder eine Bildkomprimierung sein.

  Die bearbeiteten Bilddaten werden vom FPGA in den Hostrechner transferiert. Dies stellt die Hauptdatenrichtung bei diesem Anwendungstyp dar.

  Diese Methode erlaubt es, durch Nutzung paralleler Strukturen und Pipelining wesentliche Rechenoperationen bereits auf dem Weg in den Hostrechner durchzuführen. Der Mikroprozessor wird mit diesen Operationen nicht mehr belastet und kann somit für andere Berechnungen eingesetzt werden.

  Das auf dem Host ausgeführte Anwendungsprogramm führt die Konfiguration des FPGAs, die Vorgabe von Parametern, die Gesamtsteuerung des Bildverarbeitungsvorganges (Starten, Stoppen, Unterbrechen) sowie die Kontrolle von Statussignalen durch.

Abbildung 5.2: FPGA-Koprozessor mit externer Datenquelle

Dies sind keine zeitkritischen Vorgänge, sie sollten deshalb zweckmäßigerweise im Softwarebereich implementiert werden.

- Daten werden vom Hostrechner geliefert.

  Der Hostrechner stellt die zu verarbeitenden Daten bereit und nimmt die erhaltenen Ergebnisse auf (Abb. 5.3). Hier sind keine externen Geräte am Datentransfer beteiligt.

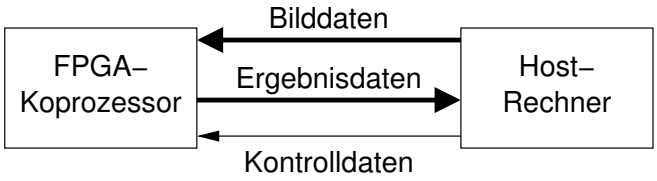


Abbildung 5.3: Hostrechner liefert und nimmt Daten ab

Dies ist der typische Anwendungsfall, wenn der FPGA-Koprozessor die auf dem Host ausgeführte Software durch Hardwarestrukturen unterstützen soll.

Die Software überträgt die zu verarbeitenden Daten zum Koprozessor. Dort werden sie von der im FPGA enthaltenen Logik bearbeitet und die Ergebnisse zum Hostrechner zurücktransferiert.

Vorteilhaft ist dieses Verfahren, wenn Algorithmen durch parallele Strukturen und Pipelining beschleunigt werden können. Voraussetzung ist jedoch, daß der Zeitaufwand für den nötigen Datentransfer in beide Richtungen im Vergleich zur Algorithmusbeschleunigung vernachlässigbar ist.

Kapitel 6
Software zur Designentwicklung

6.1 Einführung

Zur Erstellung von FPGA-Designs sind verschiedene Entwicklungssysteme verfügbar. Der prinzipielle Erstellungsvorgang von der Eingabe des Schaltungsentwurfes bis hin zur Inbetriebnahme läuft bei all diesen Systemen nach einem ähnlichen Schema ab:

Zunächst gibt der Entwickler seinen Schaltungsentwurf in das Entwicklungssystem ein. Danach kann er mit Hilfe des Systems eine softwaremäßige Simulation seiner Schaltung vornehmen. Entdeckt er dabei Fehler, so korrigiert er den Entwurf. Sobald die Simulation fehlerfrei abläuft, kann er die Synthese durchführen. Diese erstellt eine Netzliste, die von den Place&Route-Werkzeugen des FPGA-Herstellers weiterverarbeitet werden muß, um einen Konfigurationsbitstrom zu erhalten. Mit diesem wird der FPGA konfiguriert. Zuletzt überprüft er, ob seine Schaltung auch im Echtzeitbetrieb fehlerfrei funktioniert. Treten noch Fehler auf, muß der Entwurf wieder entsprechend korrigiert werden. Andernfalls ist der Erstellungsprozeß abgeschlossen.

6.1.1 Eingabe des Schaltungsentwurfes

Die Eingabe des Schaltungsentwurfes, die auch Hardwarebeschreibung genannt wird, kann in grafischer oder textueller Form vorgenommen werden.

Die existierenden grafischen Schaltplaneditoren gehen dabei weit über die üblichen Funktionen von Zeichenprogrammen hinaus. Es stehen fortgeschrittene Methoden zur Verfügung, mit denen Hierarchien gebildet, automatische Überprüfungen vorgenommen und Bibliotheken aufgebaut werden können. Aus diesen Gründen kann eine grafische Entwurfseingabe immer noch durchgeführt werden, obwohl sie bei umfangreichen Designs schnell unübersichtlich und damit schwer wartbar zu werden droht.

Textuelle Formen der Schaltungseingabe bedienen sich direkter Hardwarebeschreibungssprachen. Bei diesen wird nach einer vorgegebenen Syntax und Semantik die zu implementierende Schaltung formuliert.

Nach der Methode, wie diese textuelle Beschreibung erfolgt, lassen sich drei Grundformen unterscheiden:

- Strukturelle Beschreibung.

  Hierbei handelt es sich prinzipiell um die textuelle Beschreibung eines Schaltplans. Es werden die eingesetzten Bauteile und deren Verbindungsstruktur spezifiziert.

  Es müssen jedoch nicht alle Bauteile explizit aufgezählt werden. Oft sind Formulierungen möglich, die implizit Bauteile anlegen und verschalten. Aus Signalen und Operatoren gebildete Schaltfunktionen sind Beispiele für solche Formulierungen. So erzeugt etwa in der strukturellen Beschreibungssprache *ABEL* [117] die Formulierung

  ```
  D = (A & B) # C
  ```

  automatisch ein UND-, ein ODER-Gatter sowie die dazugehörigen Verbindungen. Ein wichtiges Merkmal solcher struktureller Beschreibungen ist, daß die spezifizierte Struktur dauerhaft, also unabhängig von einem bestimmten Ausführungszeitpunkt, ist.

- Verhaltensorientierte Beschreibung.

  Bei dieser Form der Beschreibung steht nicht die Verbindungsstruktur im Vordergrund, sondern vielmehr das Verhalten der Komponenten. Die Beschreibung erfolgt auf einer höheren Abstraktionsebene, auf der nicht mehr alle Details angegeben werden müssen. Die Formulierung in der Sprache *VHDL* [104]

```
Y <= '1' when A = '0' and B = '0' else '0';
wait until clock'event and clock = '1';
Y <= '1' when A = '1' and B = '1' else '0';
wait until clock'event and clock = '1';
```

spezifiziert, wie sich der Zustand des Signals Y in Abhängigkeit von den Signalen A und B verändert. Zusätzlich ist hier noch die Information über eine aktuelle Ausführungsposition enthalten: Nach der ersten steigenden Taktflanke folgt Y einer anderen Bedingung. Damit geht diese Beschreibungsform weit über die Möglichkeiten der strukturellen Beschreibung hinaus.

In der strukturellen Form müßte die Zeitabhängigkeit explizit mittels spezieller Zustandsvariablen in die Struktur integriert werden. Dies erfolgt bei der verhaltensorientierten Beschreibung automatisch, indem das Entwicklungssystem die notwendigen zusätzlichen Strukturen erzeugt. Der Entwickler ist in dieser Hinsicht von zahlreichen Implementierungsdetails entlastet.

- Hochsprachenorientierte Hardwarebeschreibung.

  Hierbei handelt es sich um eine aktuelle Forschungsrichtung [27, 32, 36, 37, 38, 59, 88, 89, 123, 20, 95]. Sie hat zum Ziel, in konventionellen Programmiersprachen, wie etwa C, formulierte Algorithmen direkt in eine effiziente Hardwarestruktur zu kompilieren. Dazu sind spezielle Hardwarekompiler erforderlich, wobei das Ausführungsmodell der konventionellen Softwareprogrammierung entsprechen sollte. Im Idealfall könnte dann ein in C für einen Mikroprozessor formulierter Algorithmus direkt und ohne Anpassungen vom Hardwarekompiler bearbeitet und für eine Hardwarestruktur optimiert werden.

  Einige Beispiele für hochsprachenorientierte Entwicklungssysteme sind *Handel-C* [20], *Streams-C* [38], *PPC* [123] und *SystemC* [95].

Die herkömmlichen kommerziellen Entwicklungssysteme, die auf *ABEL*, *VHDL*, *Verilog* oder C-ähnlichen Hochsprachen basieren, verwenden eigene Kompiler zur Übersetzung der Hardwarebeschreibung.

Es sind jedoch in den letzten Jahren auch einige Systeme entstanden, die zur strukturellen Hardwarebeschreibung konventionelle Programmiersprachen wie etwa C++ oder *JAVA* einsetzen. Zur Übersetzung können dann handelsübliche Kompiler verwendet werden. Durch die Beschränkung auf Standardkompiler lassen sich mit solchen Systemen allerdings nicht alle der drei oben genannten Grundformen der Hardwarebeschreibung realisieren.

Es gibt unterschiedliche Methoden, wie C++ oder *JAVA* konkret zur Hardwarebeschreibung genutzt werden kann. Bei allen wird die notwendige Funktionalität in speziellen Bauteilbibliotheken implementiert. Die Form der Hardwarebeschreibung muß sich prinzipiell an der grundsätzlichen Struktur von C++ bzw. *JAVA* orientieren. Sowohl das Ausführungsmodell als auch die Bedeutung einzelner Formulierungen weichen daher bei diesen Beschreibungsformen deutlich vom eigentlichen C++/*JAVA*-Modell ab.

Die spezielle Problematik sowie Lösungsansätze dieser Entwicklungssysteme werden später in eigenen Kapiteln diskutiert.

Beispiele für C++/*JAVA*-orientierte Entwicklungssysteme sind *PamDC* [98], *JHDL* [9] und *JERC* [52].

6.1.2 Simulation

Nach der Beschreibung der zu implementierenden Schaltung mit einer der oben genannten Methoden könnte der Entwickler direkt zur Realisierung übergehen und das Ergebnis seines Entwurfes in der Hardware testen. Insbesondere Schaltungsentwürfe im frühen Entwicklungsstadium können jedoch noch Fehler enthalten. Eine frühe Fehlersuche direkt an der Hardware ist aus folgenden Gründen nicht empfehlenswert:

- Die Fehlersuche im Echtzeitbetrieb ist oft wegen der hohen Schaltgeschwindigkeiten mit einem erheblichen Aufwand verbunden. So sind spezielle Logikanalyzer notwendig, die jedoch nur einen kleinen zeitlichen Ausschnitt darstellen können. Die Auswahl der korrekten Signale und Triggerbedingungen kann ein langwieriger Prozeß bei der Lokalisierung von Fehlern sein.

- Selbst beim Einsatz moderner Meßgeräte sind nicht alle Signale der Schaltung von außen zugänglich. Der Entwickler kann bei der Fehlersuche somit nicht alle Signalverläufe überwachen.

- Ein Fehler in der implementierten Logik kann die Funktion der Umgebungshardware so stark beeinträchtigen, daß eine Fehlersuche am laufenden System unmöglich wird oder sogar ein Schaden entsteht.

Daher bieten fast alle Entwicklungssysteme eine softwarebasierte Simulation der Hardwarebeschreibung an. Diese hat den Vorteil, daß der zeitliche Verlauf der Signale leicht überwacht und aufgezeichnet werden kann. Es sind sämtliche Signale verfügbar und es kann kein Schaden an der Hardware entstehen.

Abhängig vom Entwicklungssystem stehen verschiedene Methoden der Simulation zur Verfügung:

- Testvektoren [11].

  Hier werden in textueller Form eine Reihe von Werten für die Eingangssignale und die entsprechenden Werte der erwarteten Ausgangssignale angegeben. Die Hardwarebeschreibung wird dann unter Verwendung dieser Eingabewerte simuliert und es wird geprüft, ob die tatsächlichen Ausgangswerte mit den erwarteten übereinstimmen. Auch eine grafische Ausgabe aller Signalverläufe (*Waveform*) ist möglich.

- Testbenches.

  Bei komplexeren Simulationen erfolgt die Erzeugung der Eingangssignale nicht mit Testvektoren, sondern mittels Testbenches. Hierbei wird mithilfe einer in der Regel verhaltensorientierten Beschreibung eine virtuelle Komponente generiert, die dann die Stimuli für die Simulation liefert.

  Dieses Verfahren hat insbesondere den Vorteil, daß die Stimuli auch als Reaktion auf die Signale des gerade simulierten Designs erzeugt werden können. Vor allem, wenn die Design-Signale nicht zu exakt festgelegten Zeitpunkten agieren müssen, sondern einen zeitlichen Spielraum haben, sind Testbenches besser geeignet als Testvektoren mit fester Zeitvorgabe.

Der Simulationsprozeß kann geschätzte Zeitwerte für Durchlaufverzögerungen, Clock-to-output- und Setup-Zeiten berücksichtigen. Damit können Reaktionen des Designs auf Verletzungen bestimmter Zeitkriterien wie Setup-Zeiten an den Eingangssignalen oder maximale Taktfrequenz untersucht werden.

Es ist jedoch auch möglich, eine Simulation ohne solche Zeitinformationen durchzuführen. Es werden dann Standardwerte für interne Verzögerungen angenommen. Eine solche *funktionale* Simulation ist immer dann sinnvoll, wenn keine exakten Zeitinformationen vorliegen oder garantiert ist, daß die Schaltung später immer innerhalb ihrer zeitlichen Spezifikationen betrieben wird.

Insbesondere kann eine funktionale Simulation bei FPGA-Koprozessoren ausreichend sein. Vor dem Place&Route-Prozeß liegen keine detaillierten Zeitinformationen vor. Diese können nur mit dem Timing-Analyzer des FPGA-Herstellers ermittelt werden. Dieser berechnet für ein plaziertes Design die maximale Taktfrequenz und liefert einen Bericht über die Einhaltung der vorgegebenen Zeitkriterien. Bei einem FPGA-Koprozessor werden nur

Änderungen am FPGA-Design vorgenommen, nicht aber an der Umgebungshardware. Daher kann die Einhaltung der zeitlichen Spezifikationen in der Regel garantiert werden.

Moderne Entwicklungssysteme erlauben nicht nur die Simulation einzelner Komponenten, wie etwa FPGAs. Vielmehr ist es möglich, ein gesamtes System miteinander interagierender Bausteine zu simulieren. Auf diese Weise kann durch die Simulation die Funktionsfähigkeit des Gesamtsystems bereits vor dem Aufbau eines Prototypen überprüft werden.

6.1.3 Synthese

Bei der Synthese werden die in der Hardwarebeschreibung verwendeten Elemente auf die prinzipiell vorhandenen Resourcen des Zielbausteins abgebildet. Das Ergebnis ist eine Netzliste, in der alle verwendeten Resourcen, ihre Parameter sowie die Verschaltungsstruktur enthalten sind.

Die Netzliste beinhaltet darüberhinaus Angaben über einzuhaltende Zeitkriterien, Plazierungsinformationen für einzelne Elemente, Startzustände von Speicherelementen und die Zuordnung der nach außen gerichteten Signale zu den Gehäusepins.

Beim nachfolgenden Place&Route-Prozeß, den nicht mehr das Entwicklungssystem, sondern eine vom FPGA-Hersteller gelieferte Software durchführt, werden die in der Netzliste enthaltenen Resourcen auf die programmierbaren Elemente des Bausteins abgebildet und die vorhandenen Verbindungsresourcen verwendet, um die Elemente miteinander zu verschalten.

Dies ist ein sehr rechenintensiver Vorgang, der bei Designs für moderne FPGAs abhängig von den vorgebenen Zeitkriterien mehrere Stunden benötigen kann.

Das Ergebnis ist ein Konfigurationsbitstrom, der zur Inbetriebnahme in den FPGA geladen werden muß.

Die Place&Route-Software wird grundsätzlich vom FPGA-Hersteller geliefert. Sie nimmt als Eingabe Netzlisten in den Formaten EDIF bzw. XNF (nur *XILINX*) an. Da die FPGA-Hersteller den genauen Aufbau ihres Konfigurationsbitstromes unter Verschluß halten, ist diese Place&Route-Software die unterste Ebene, an der herstellerfremde Entwicklungssysteme ansetzen können.

6.1.4 Hardware-Debugging

Auch bei einer sorgfältig durchgeführten Simulation ist noch kein allgemein fehlerfreies Funktionieren eines FPGA-Designs garantiert. Eine Simulation kann immer nur eine beschränkte Anzahl von Ablaufkombinationen abdecken.

Insbesondere Mikroprozessorsysteme weisen aufgrund von Busarbitrierung und Multitasking-Betriebssystemen ein stark schwankendes Zeitverhalten auf. Durch die Vielzahl der möglichen Varianten im zeitlichen Ablauf ist eine vollständige Simulation in der Praxis kaum möglich. Daher kann mit einer Simulation zwar die Existenz eines Fehlers nachgewiesen werden, aber nicht die Fehlerfreiheit.

In der Praxis kann es weiterhin vorkommen, daß eine reale Hardwarekomponente nicht im Detail dem bei der Simulation verwendeten Modell entspricht.

Eine Simulation sehr umfangreicher Designs über viele Taktzyklen kann zu extrem langen Simulationszeiten führen. Die Aufzeichnung der Signalverläufe produziert dann Datenmengen, die nicht mehr sinnvoll handhabbar sind.

In solchen Fällen ist es notwendig, ein Hardware-Debugging durchzuführen. Dies erfolgt in der Regel mit speziellen Meß- und Anzeigegeräten, z.B. Logikanalyzern [34].

Ziel ist es, durch Überwachung und Aufzeichnung von Signalen im Echtzeitbetrieb unter realen Bedingungen schrittweise die Fehlerquelle zu lokalisieren.

6.2 Anforderungen an ein optimales FPGA-Entwicklungssystem

6.2.1 Einführung

In den letzten Jahren verstärkte sich zunehmend die Auffassung, daß *VHDL* nicht die optimale Unterstützung für komplexe FPGA-Koprozessoranwendungen bietet [9, 13, 58, 14, 54, 23, 43].

Daher verwenden einige Systeme [9, 62, 23] weit verbreitete Programmiersprachen, wie etwa C++ oder *JAVA*, ohne Veränderung ihrer Syntax zur Hardwarebeschreibung. Die Übersetzung erfolgt dabei mit einem handelsüblichen Kompiler (z.B. *Microsoft Visual C++*). Andere Forschungsrichtungen beschäftigen sich mit der direkten Übersetzung von Hochsprachen in Hardwarestrukturen.

Weiterhin wurde eine Reihe von Anforderungen aufgestellt, die ein optimales FPGA-Entwicklungssystem erfüllen sollte. Diese Anforderungen werden nachfolgend, aufgeteilt in die Bereiche Hardwarebeschreibung, Simulation, Synthese und Hardware-Debugging, dargestellt und näher erläutert.

6.2.2 Hardwarebeschreibung

An ein ideales FPGA-Entwicklungssystem werden folgende Anforderungen gestellt [9, 13, 58, 14, 54, 23]:

- Die Hardwarebeschreibung sollte intuitiv und leicht erlernbar sein. Sie sollte auf einer leicht verständlichen universellen Programmiersprache basieren. Existierende Kompiler und Debugger sollten so weit wie möglich genutzt werden können.
- Das System sollte intensiven Gebrauch moderner objektorientierter Techniken machen.
- Es muß eine hierarchische Designbeschreibung möglich sein.
- Es sollte keinen Unterschied zwischen den vorgegebenen Primitiven und den benutzerdefinierten Komponenten geben.
- Die Beschreibung sollte in weitem Umfang Parametrisierungen unterstützen.
- Es müssen unterschiedliche Abstraktionsebenen unterstützt werden.

In der Praxis hat sich insbesondere bei den FPGA-Koprozessoren eine Verschiebung der Entwicklungsarbeit von der Hardware- auf die Softwareebene ergeben. Echte Hardwareentwicklung ist in der Regel nur noch beim Aufbau von Erweiterungsmodulen notwendig.

Die Entwicklung der FPGA-Designs erfolgt zunehmend auf der Softwareebene. Hier stellt sich das Problem, daß die Schaltungsentwürfe für moderne FPGAs sowohl umfangreicher als auch in ihrer Struktur komplexer sind als früher. Um diesen Umfang und die Komplexität beherrschen zu können, müssen die Hardwarebeschreibungssprachen geeignete Mechanismen zur Verfügung stellen.

Im Bereich der konventionellen Softwareentwicklung sind diese schon seit langer Zeit bekannt. Hier wurden wirksame Lösungsmethoden entwickelt, so etwa der Übergang von Assembler- zu Hochsprachen oder das Konzept der Objektorientierung. Weiterhin stehen dem Softwareentwickler heute Code-Analyzer, Source-Level-Debugger sowie eine Reihe weiterer Werkzeuge zur Verifikation und zur Erleichterung der Fehlersuche zur Verfügung.

Die Entwicklung von Anwendungen für FPGA-Koprozessoren ist eng verbunden mit der Erstellung von Software für Mikroprozessoren. Daher liegt es nahe, die für die Softwareentwicklung eingesetzten, bewährten und dem Entwickler vertrauten Werkzeuge und Prinzipien auch für die Hardwarebeschreibung zu verwenden.

So tragen etwa die modernen Techniken der Objektorientierung zu einer Erhöhung der Abstraktionsebene bei. Im wesentlichen sind es die Konzepte der Vererbung, Datenkapselung, virtuellen Funktionen sowie der automatischen Ausführung von Konstruktoren und Destruktoren, die zu deutlich mächtigeren Beschreibungen führen.

Sowohl der Umfang als auch die Komplexität können durch Bildung von Hierarchien beherrscht werden. Solche Hierarchien lassen sich unter Verwendung des Klassenkonzeptes aufbauen.

Sind vordefinierte und nachträglich erstellte Komponenten in gleicher Weise anwendbar, können beliebig tief geschachtelte Hierarchien gebildet werden.

Die neu erstellten Komponenten können zur leichteren Anwendung in Bibliotheken gesammelt werden. Wenn ähnliche Elemente, mit Parametern versehen, zu einem gemeinsamen Programmcode zusammengefaßt werden, ergibt sich eine deutlich reduzierte Anzahl zu wartender Komponenten.

Das wesentliche Ziel beim Einsatz von FPGAs besteht darin, durch die Hardwareimplementierung von Algorithmen Geschwindigkeitsvorteile gegenüber einer herkömmlichen Softwareimplementierung zu erreichen.

Dazu muß der Entwickler einer FPGA-Schaltung einerseits in der Lage sein, zeitkritische Teile seines Designs auf der untersten Hardwareebene zu beschreiben, auf der er maximalen Einfluß auf den Einsatz der speziellen FPGA-Resourcen besitzt. Dies wird in der Regel eine strukturelle Beschreibung mit detaillierter Anordnung einzelner Flip-Flops, Gatter, Speicherblöcke usw. sein. Andererseits benötigt er für weniger zeitkritische Teile eine höhere Ebene, auf der er schnell und unkompliziert Designbeschreibungen realisieren kann.

Dieses Nebeneinander verschiedener Abstraktionsebenen ist in der konventionellen Softwareentwicklung ebenfalls bekannt: An zeitkritischen Stellen kann jederzeit auf Assemblercode zurückgegriffen werden. In modernen Kompilern kann dies innerhalb des C/C++-Codes in Form von Inline-Assembler erfolgen. Auch Source-Level-Debugger unterstützen diese Integration.

6.2.3 Simulation

Die Simulation sollte folgende Voraussetzungen erfüllen [9, 13, 58, 14, 54, 23]:

- Die verwendeten Objekte sollten nicht nur einfache Netzlistenobjekte repräsentieren, sondern auch die Simulation unterstützen.
- Das System muß eine Simulation auf der jeweiligen Architektur unterstützen.
- Die Simulation sollte in allen Phasen des Designerstellungsprozesses unterstützt sein. Mixed-Mode-Simulation muß möglich sein.
- FPGA-Mikroprozessorkopplungen müssen sowohl in Simulation als auch im Echtzeitbetrieb unterstützt werden. Dies sollte auf eine einheitliche Weise erfolgen können. Die Hardwarebeschreibung muß beide Bereiche ohne Änderungen unterstützen können.
- Es sollte Unterstützung für die Rekonfiguration zur Laufzeit besitzen.

Aufgrund der komplexen Interaktion der in einem System vorhandenen konfigurierbaren und nichtkonfigurierbaren Komponenten muß eine Simulation des gesamten Systems möglich sein. Es reicht heute nicht mehr aus, nur einzelne Logikbausteine zu simulieren. Ebenfalls nicht mehr ausreichend ist das Konzept der Testvektoren, die an einen simulierten Baustein angelegt werden, um anschließend den tatsächlich erzeugten Ergebnisvektor mit dem erwarteten zu vergleichen.

Vielmehr müssen alle Komponenten in die Simulation einbezogen werden, wobei externe Komponenten, wie etwa Speicherbausteine, durch entsprechende Modelle zu simulieren sind. Damit ergibt sich eine interaktive Simulation. Dieses Verfahren bietet eine wesentlich realistischere Simulation als der Einsatz von Testvektoren. Auch lassen sich bedeutend mehr Simulationsfälle abdecken.

Eine Mixed-Mode-Simulation ermöglicht eine Zusammenstellung des Systems aus synthetisierbaren und nichtsynthetisierbaren Beschreibungen. Damit wird es möglich, FPGA-Designs zu simulieren, in denen noch nicht alle Module synthetisierbar implementiert sind. So kann bereits in diesem Stadium das prinzipielle Funktionieren der gesamten Anordnung getestet werden. Im Weg der schrittweisen Verfeinerung werden danach die übrigen Module im Detail implementiert.

Dieses Verfahren kann auch dazu beitragen, die Simulationsgeschwindigkeit zu erhöhen, da ein nichtsynthetisierbares Modell in der Regel schneller simuliert werden kann als eine detaillierte strukturelle Implementierung.

Bei FPGA-Koprozessoren muß auch die auf dem Hostrechner ablaufende Software, die mit dem FPGA kommuniziert, in die Simulation einbezogen werden können. Vorteilhaft wäre hier eine Simulation, bei der diese Software in Echtzeit auf dem Hostrechner ausgeführt werden kann, wobei der Hardwaresimulator zu den Zeitpunkten der FPGA-Kommunikation auf geeignete Weise eingreift. Es ist auch denkbar, die Software unter Kontrolle eines herkömmlichen Debuggers auszuführen. Notwendig ist dazu die Simulation von Zugriffen auf Pseudoregister um die erforderliche Verbindung zwischen der Software auf dem Hostrechner und dem Hardwaresimulator herzustellen.

Erfolgt bei einer Koprozessoranwendung eine Rekonfiguration des FPGAs zur Laufzeit, so muß diese Änderung der Hardware auch während der Simulation berücksichtigt werden können. Der Simulator muß folglich in der Lage sein, während eines Simulationslaufes die zugrundeliegende Hardwarebeschreibung auszutauschen.

Die Simulation eines FPGA-Koprozessors kann sehr komplex sein. Ein Simulationsmodell für einen Speicherchip oder ein Taktsystem kann noch mit wenig Aufwand realisiert werden, einen Mikroprozessor durch Software zu simulieren, benötigt erheblich mehr Aufwand.

Diese Probleme wurden im *Riley-2*-Projekt erkannt [58]. Eine exakte Simulation des ganzen Systems wäre sehr langsam und speicherintensiv, so daß jeweils nur Teile des Systems zusammen simuliert werden könnten. Deshalb wurden einige Lockerungen der exakten Simulation zugelassen:

- Es wird nur das funktionale Verhalten der Umgebung simuliert. Eine exakte zeitliche Simulation benötigt erheblich mehr Rechenaufwand, führt aber zum gleichen Ergebnis, vorausgesetzt, das System wird innerhalb der zulässigen Frequenzen betrieben. Oft sind zeitliche Angaben der Hersteller auch nicht exakt garantiert, sondern beschreiben nur den worst-case Fall.

- Die Simulation muß nicht das gesamte Prozessormodell bzw. nicht alle Instruktionen beinhalten.

- Die Simulation des Bustransfers sollte jedoch möglichst exakt vorgenommen werden.

Obwohl sie in manchen Punkten eingeschränkt ist, wird eine funktionale, schnelle Simulation, die das gesamte System umfaßt, besser sein, als eine, die zwar zeitlich exakt, aber mit unvollständigen oder nicht garantierten Zeitinformationen nur einen Teil des Systems simulieren kann.

Renner et al. [75] vertreten die Ansicht, daß eine Hardware-Software-Kosimulation nicht ausreichend ist. Sie schlagen den Einsatz einer architekturspezifischen Simulation mit einer Echtzeitprototypenumgebung wie etwa *REPLICA* vor, um besser mit den Problemen der Echtzeitdatenflußsteuerung umgehen zu können.

6.2.4 Synthese

Die Synthese muß folgende Kriterien erfüllen [9, 13, 58, 14, 54, 23]:

- Es muß eine effiziente Nutzung der FPGA-Resourcen gewährleistet sein. Detaillierte strukturelle Formulierungen sowie Vorplazierungen und Timing-Vorgaben müssen möglich sein.

- Das System muß eine ausreichende Portabilität zwischen den FPGA-Familien unterstützen. Dies betrifft insbesondere die Gestaltung des Interfaces zwischen FPGA und Anwendungssoftware.

Das Entwicklungssystem muß über geeignete Optimierungsverfahren verfügen, um FPGA-Resourcen effizient nutzen zu können. So sollten Schaltfunktionen minimiert und redundante Signale entfernt werden können. Der Entwickler muß weiterhin die Möglichkeit besitzen,

kritische Teile seines Designs so exakt wie möglich auf den FPGA zu übertragen. Oft können durch Vorplazierungen einzelner Resourcen und durch detaillierte Timing-Vorgaben bessere Ergebnisse erreicht werden. Das Entwicklungssystem muß diese Angaben in die Netzliste übernehmen.

6.2.5 Hardware-Debugging

Die Synthese sollte folgenden Anforderungen genügen:

- Die generierten Netzlisten müssen eindeutige Netznamen besitzen, die auf das ursprüngliche Design zurückzugeordnet werden können. Dies ist eine wichtige Anforderung für neue Methoden des Hardware-Debugging.
- Es muß eine Möglichkeit zur absoluten Vorplazierung bestehen.

Das Readback-Verfahren ermöglicht es, zu jedem beliebigen Zeitpunkt und unabhängig vom Echtzeitbetrieb den aktuellen Zustand der internen Flip-Flops, Speicherblöcke sowie einiger weiterer Signale auszulesen. Um diese Informationen den Bauteilen der Hardwarebeschreibung zuordnen zu können, ist es erforderlich, daß die Namensgebung des Syntheseprozesses dies unterstützt. Insbesondere dürfen während der Synthese keine automatischen Namen vergeben werden, über die keine Rückzuordnungsinformationen verfügbar sind.

Für einige Primitive, etwa Block-RAMs, ist es notwendig, eine Vorplazierung vornehmen zu können, da deren Position nicht über den automatischen Mechanismus (*logic allocation file*), der später vorgestellt wird, ermittelt werden kann.

6.3 *VHDL*-basierte Entwicklungssysteme

6.3.1 Allgemeines

Es existiert eine Vielzahl kommerzieller Entwicklungssysteme, die die Sprache *VHDL* zur Designeingabe benutzen. *VHDL* gilt als Industriestandard und wird insbesondere auch zur Implementierung von Schaltungen für ASICs eingesetzt.

Solche Systeme sind z.B.

- Aldec: *Active-CAD* [2].
- Exemplar: *Leonardo Spectrum* [30].
- Accolade: *PeakVHDL* und *PeakFPGA DesignSuite* [1].
- Synplicity: *Synplify* [93].
- Synopsys: *FPGA Compiler II* (früher: *FPGA Express*) [92, 91].

Diese Systeme unterstützen den kompletten Entwicklungszyklus von der Designeingabe bis zur Erstellung der architekturspezifischen Netzliste. Dies umfaßt auch die Verwaltung von Bibliotheken, Integration von Modulen von Drittanbietern sowie die Simulation der entwickelten Schaltung.

Einige Systeme unterstützen nur die Simulation von *VHDL*-Code, nicht die Synthese:

- Aldec: *Active HDL* / *Riviera* [2].
- Model Technology: *ModelSim* [65].

6.3.2 Hardwarebeschreibung

Die Hardwarebeschreibung in *VHDL* wird mittels *entities* vorgenommen. Jede *entity* definiert dabei ein Interface (*port*) sowie eine *architecture*, in der das jeweilige Verhalten der *entity* beschrieben wird.

Innerhalb der *port*-Definition werden die Signalnamen sowie die Richtung (`in`, `out` bzw. `inout`) angegeben:

```
entity dff is
 port ( clock : in  std_logic;
        din   : in  std_logic;
        dout  : out std_logic );
end dff;
```

Die *architecture* kann die Definitionen interner Signale, Komponenten sowie von Prozessen enthalten.

```
architecture rtl of dff is

signal intclk,enable : std_logic;

component BUFGS
  port ( I : in std_logic;
         O : out std_logic );
end component;

begin
  the_intclk : BUFGS port map (I => clock, O => intclk);

  enable <= '1';

  process(intclk,enable,din)
  begin
    wait until intclk'event and intclk = '1';
    if (enable = 1) then
       dout <= din;
    end if;
  end process;
end rtl;

configuration cfg_dff of dff is
   for rtl
   end for;
end cfg_dff;
```

Die Verwendung der Komponente `the_intclk` zeigt, daß *VHDL* neben verhaltensorientierten Prozessen auch strukturelle Beschreibungen unterstützt. Es ist möglich, auch große Designs ausschließlich mit solchen strukturellen Elementen zu formulieren.

VHDL verfügt weiterhin über Anweisungen, die kompakte Schreibweisen für wiederholte Formulierungen bieten. Die folgende Implementierung eines 8-Bit breiten 1-aus-4 Multiplexers demonstriert dies:

```
read_sigs: block
begin
  mux_dbus0: for i in 0 to 7 generate
```

```
        databus(i) <=      reg1(i) when sel = "00"
                      else reg2(i) when sel = "01"
                      else reg3(i) when sel = "10"
                      else reg4(i) when sel = "11"
                      else '0';
  end generate mux_dbus0;
end block read_sigs;
```

Auch die Definition von Zustandsmaschinen ist möglich:

```
type STATE_TYPE_A is (S0, S1);
attribute ENUM_ENCODING of STATE_TYPE_A : type is "0 1";
signal atc_state : STATE_TYPE_A;

mystates: process
begin
  wait until clk'event and clk = '1';

  case mystate is
    when S0 =>
             if (enable = '0') then
                 mystate <= S0;
               else
                 if (go = '1') then
                     mystate <= S1;
                   else
                     mystate <= S0;
                 end if;
             end if;
    when S1 =>
             mystate <= S0;
  end case;
end process mystates;
```

6.3.3 Simulation

Die Simulation einer *VHDL*-Hardwarebeschreibung erfolgt mittels Testbenches. Dies sind Formulierungen in *VHDL*, die die notwendigen Stimuli erzeugen.

Die Kontrolle und grafische Anzeige der Simulationsergebnisse erfolgt mit Werkzeugen, die in die jeweiligen *VHDL*-Entwicklungssysteme integriert sind.

Da der Testbench-Code nicht synthetisierbar sein muß, stehen hier erweiterte Anweisungen zur Verfügung, die bei synthetisierbarem Code nicht zulässig sind. Dies sind etwa Befehle zur Ausgabe von Kontrollmeldungen, zum Durchführen von Dateizugriffen und zum Allokieren und Freigeben von Speicher. Mithilfe dieser Erweiterungen ist es möglich, Testbenches zu implementieren, die Testvektoren aus Dateien lesen, Ergebnisse in Dateien schreiben oder auch externe RAM-Bausteine einschließlich Inhalt simulieren können.

Moderne *VHDL*-Simulatoren enthalten darüberhinaus Schnittstellen zur Anbindung von dynamischen Link-Bibliotheken (DLLs). Damit kann nahezu beliebiger C/C++-Code in die Simulation integriert werden.

6.3.4 Synthese

Die Synthese von *VHDL*-Code erfolgt durch spezielle Kompiler, die direkt Netzlisten erzeugen.

Zusatzinformationen wie etwa Vorplazierungen, Zuordnungen zu den Gehäusepins oder Timing-Kriterien werden, abhängig vom *VHDL*-Kompiler, direkt in den *VHDL*-Code integriert oder der Place&Route-Software in separaten Dateien übergeben.

6.4 *PamDC*

6.4.1 Allgemeines

Bei *PamDC* [98] handelt es sich um eine C++-Klassenbibliothek, die von der Firma *Digital Equipments Corporation* für die *XC4000*-FPGAs entwickelt wurde. Es ist das Nachfolgesystem von *Perle1DC* [10]. Die Zielplattform für *PamDC* war der FPGA-Prozessor *DecPerle*.

6.4.2 Hardwarebeschreibung

Der Datentyp `Bool` wird verwendet, um Signale zu definieren. Damit können Schaltfunktionen entsprechend der C-Syntax spezifiziert werden. Der Operator = dient dabei zum Zuweisen des Ergebnisses an ein neues Netz:

```
Bool  a, b, c;

c = a ^ b;
```

Zusätzlich zu den Signalen können weiterhin die Konstanten `ZERO` und `ONE` eingesetzt werden.
Speicherelemente werden mittels der Primitive `reg` erzeugt (Abb. 6.1).

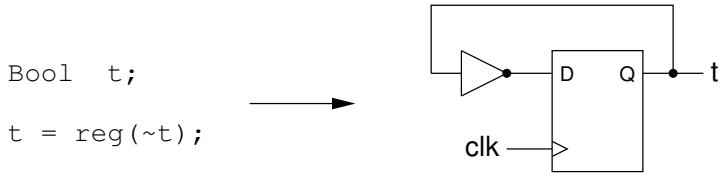


Abbildung 6.1: Erzeugen von Speicherelementen

Es können Busse definiert werden, z.B.

```
WireVector<Bool,12> data;
```

Diese Busse können jedoch nicht direkt in Schaltfunktionen eingesetzt werden. Stattdessen ist die Implementierung von Schleifen erforderlich:

```
for (int i = 0; i < 12; i++)
   data[i] = reg(data[i+1]);
```

Der Anwender kann eigene Klassen erstellen und diese später wiederum in neuen Klassen verwenden. Auf diese Weise können modulare Designs erzeugt werden.

```
class FullAdder : public Node
{
public:
  FullAdder() : Node("FullAdder") {}
  void logic (Bool& a, Bool& b, Bool& cin,
              Bool& sum, Bool& cout)
  {
    input(a);
    input(b);
    input(cin);
    output(sum);
    output(cout);

    sum = a ^ b ^ cin;
```

```
    cout = (a & b) | (b & cin) | (cin & a);
  }
};
```

Bei den vordefinierten Elementen wie Registern, Multiplexern usw. erfolgt die Verschaltung über die Parameterliste beim Konstruktoraufruf. Die Anordnung der Signale wird dabei durch die Position der Parameter festgelegt (Abb. 6.2).

```
Bool  a, b, s, y;

y = mux(s,a,b);
```

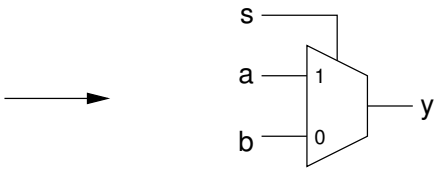


Abbildung 6.2: Verschalten von vordefinierten Elementen

Bei den vom Anwender erstellten Modulen erfolgt die Verschaltung über die `logic()`-Funktion:

```
FullAdder  adder;

adder.logic(a,b,cin,sum,cout);
```

In die Hardwarebeschreibung können Vorplazierungsanweisungen und Timingkriterien integriert werden:

```
a <<= LOC(12,11);

a <<= TIMESPEC(40);
```

Die Vorgabe von Startwerten für die Speicherelemente ist nur zusammen mit einer Negation und einer Vorplazierung möglich. Dadurch wird die Negation in das Speicherelement verlagert und der Startzustand "1" erreicht.

```
out = ~reg(in);
out <<= LOC(2,2);
```

Die Zuordnung der internen Signale zu den Gehäusepins kann wie folgt vorgenommen werden:

```
Bool  pad;

assign_iob(100,pad);
assign_input(100);
```

Dies ordnet dem Signal `pad` den IOB mit Pinnummer 100 zu und legt fest, daß es sich nur um einen Eingang handelt.

6.4.3 Simulation

Um eine Simulation durchzuführen, müssen die entsprechenden Anweisungen zur Erzeugung der Stimuli in die `main()`-Funktion des Gesamtprogrammes aufgenommen werden:

```
main()
{
  FullAdder* node = new FullAdder;
```

```
    ...
    node->logic(...);
    node->simul_setup();

    for (int i = 0; i < 10; i++)
     {
       reset.set_value(0);
       node->compute_outputs();
       cout << "out = " << out.get_value() << endl;
       node->tick();
     }
  }
```

Es können externe Hardwarekomponenten in die Simulation einbezogen werden. Die entsprechenden Module besitzen keine Hardwarebeschreibung innerhalb der `logic()`-Funktion. Stattdessen wird das Verhalten der Komponente in der Funktion `compute_outputs()` durch C++-Code modelliert. Dieser Code hat im wesentlichen die Aufgabe, den neuen Zustand der Ausgangssignale in Abhängigkeit von den Eingangssignalen zu berechnen. Auf diese Weise kann z.B. externer Speicher in die Simulation integriert werden.

Die Simulation besitzt jedoch einige Einschränkungen. So kann etwa keine asynchrone Logik simuliert werden. Auch kombinatorische Logik in Taktpfaden (*gated clocks*) wird nicht unterstützt.

6.4.4 Synthese

Für die Synthese hat die `main()`-Funktion folgende Struktur:

```
main()
{
  FullAdder* node = new FullAdder;
  ...
  node->logic(...);
  ...
  node->write_module("add.dcf");
}
```

Die erzeugte DCF-Datei kann danach mit dem Programm `dcf2xnf` in eine Netzliste konvertiert werden.

6.5 *JHDL*

6.5.1 Allgemeines

JHDL [9, 13] wurde an der *Brigham Young University* entwickelt. Ziel ist unter anderem die Unterstützung von mikroprozessorgekoppelten rekonfigurierbaren FPGA-Systemen sowohl in der Simulation als auch im Echtzeitbetrieb. Für beide Bereiche kann die gleiche Hardwarebeschreibung verwendet werden.

6.5.2 Hardwarebeschreibung

JHDL verwendet *JAVA* über eine Klassenbibliothek als Beschreibungssprache und ist dadurch plattformunabhängig.

Jedes Bauteil wird durch ein *JAVA*-Objekt repräsentiert. Der Anwender kann eigene Klassen erstellen, die von der Basisklasse `Logic` abgeleitet sind.

Die Basisklasse stellt Methoden zur Verfügung, mit denen kombinatorische Logik erzeugt werden kann, z.B. `and()`, `or()`, `xor()`, `and_o()`, `or_o()` und `xor_o()`.

Bei den `_o()`-Varianten wird das Ausgangssignal des entsprechenden Gatters mit dem letzten Signal der Parameterliste verbunden. Die übrigen Methoden erzeugen dagegen ein

neues Signal als Ausgang. Ihnen wird kein Ausgangssignal als Parameter übergeben. Auf diese Weise können Gatter verschachtelt werden. Dabei ist darauf zu achten, daß das äußerste Gatter immer die `_o()`-Version ist:

```
or_o(and(a,b),and(a,cin),and(b,cin),cout);
xor_o(a,b,cin,sum);
```

Die Signale `cout` und `sum` sind die Ausgänge der beiden Gatteranordnungen.

Das Interface der Klasse wird über eine spezielle Struktur definiert:

```
public static CellInterface() cell_interface =
{
  in("a",1);
  in("b",1);
  in("cin",1);
  out("sum",1);
  out("cout",1);
};
```

Die Verbindung des Interfaces mit den Signalen der Parameterliste erfolgt über `connect()`-Anweisungen. Die Methode `super()` dient dazu, die Klassenhierarchie festzulegen, indem sie die Vaterklasse spezifiziert:

```
public class FullAdder extends Logic
{
  public FullAdder ( Node parent,
                     Wire a,
                     Wire b,
                     Wire cin,
                     Wire sum,
                     Wire cout )
  {
    super(parent);
    connect("a", a);
    connect("b", b);
    connect("cin", cin);
    connect("sum", sum);
    connect("cout", cout);

    or_o(and(a,b),and(a,cin),and(b,cin),cout);
    xor_o(a,b,cin,sum);
  }
};
```

Mehrere erzeugte Objekte können über *Wires* und Parameterlisten miteinander verbunden werden.

JHDL unterstützt weiterhin Vorplazierungsinformationen:

```
place(sum,0,0);
place(cout,0,1);
```

Die Ankopplung von mikroprozessorgekoppelten FPGAs erfolgt über *Inports* und *OutPorts* (Abb. 6.3). Jeder Datentransfer zwischen Mikroprozessor und FPGA läuft über diese Schnittstelle. Sie enthält Datenpuffer, um die Synchronisierung zum globalen Designtakt vorzunehmen.

Der Zugriff auf die Ports erfolgt im Echtzeitbetrieb aus einem *JAVA*-Programm mittels Funktionen wie `port.write()` und `port.read()`.

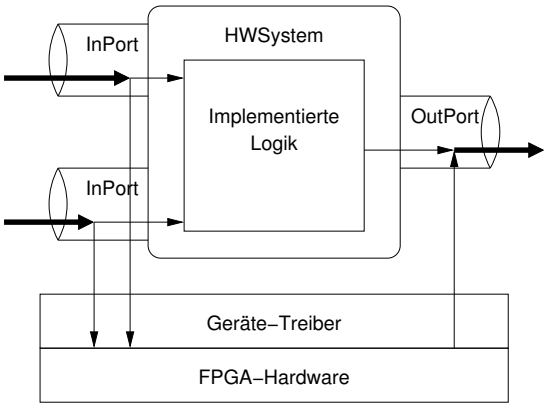


Abbildung 6.3: Systemschnittstelle von *JHDL*

6.5.3 Simulation

JHDL verwendet zur Simulation Testbenches. Dies sind Klassen, die die zu simulierende Schaltung auf oben erläuterte Weise generieren, explizit Signalwerte an die Eingänge dieser Schaltung anlegen sowie die Zustände der Ausgänge auslesen. Eine solche Klasse besitzt kein Interface.

Jede Testbench muß eine `reset()`- und eine `clock`-Methode implementieren:

```
public void reset()
{
  a.put(this,0);
  b.put(this,0);
}

public void clock()
{
  a.put(this,value1);
  b.put(this,value2);
  System.out.println("sum = " + sum.get(this));
}
```

Die Simulation von FPGA-Mikroprozessorkopplungen erfolgt ebenfalls über die *InPorts* und *OutPorts* (Abb. 6.3). Jedoch wird der Datentransfer bei der Simulation nicht über den Gerätetreiber, sondern über den *JHDL*-Simulator geführt.

Der *JHDL*-Simulationskernel implementiert folgenden Ablauf:

1. Jeder *InPort* wird veranlaßt, das nächste Datenwort aus dem Puffer zu lesen.
2. Ein Taktpuls wird ausgeführt.
3. Alle *Wires*, die von der synchronen Logik im vorigen Schritt verändert wurden, werden aktualisiert.
4. Alle betroffenen kombinatorischen Logikelemente, deren Eingänge sich geändert haben, werden aktualisiert.
5. Die Schritte 3 und 4 werden solange wiederholt, bis keine Änderungen mehr auftreten.

6. Jeder *OutPort* wird veranlaßt, den aktuellen Zustand an seinen Signalleitungen als Datenwort in seinen Puffer zu schreiben.

7. Die Schritte 1 bis 6 werden wiederholt, bis die komplette Anzahl auszuführender Taktpulse bearbeitet wurde.

Die Simulation hat jedoch einige wesentliche Beschränkungen. So wird ein global synchrones Design vorausgesetzt, in dem nur ein einziger Takt existiert. Auch asynchrone Schaltungsschleifen werden nicht unterstützt.

6.5.4 Synthese

Für die Synthese muß eine Klasse erstellt werden, die im Konstruktor folgenden Code enthält:

```
tmapper = new XC4000TechMapper(true);

Logic.setDefaultTechMapper(tmapper);

// Hier die Logik erzeugen
...

tmapper.netlist(adder,"FullAdder.edn");
```

Die generierte Netzliste enthält keine Informationen über den Typ des Ziel-FPGAs, so daß dieser dem Place&Route-Prozeß explizit übergeben werden muß.

6.6 *SystemC*

6.6.1 Allgemeines

SystemC [95] ist als C++-Klassenbibliothek konzipiert. Es kann zusammen mit handelsüblichen C++-Entwicklungswerkzeugen eingesetzt werden, um ausführbare Modellspezifikationen auf Systemebene zu erstellen. Bei dieser ausführbaren Spezifikation handelt es sich um ein C++-Programm, das das Verhalten des Systems simuliert. C++ wurde als Beschreibungssprache für diese Spezifikationen gewählt, da sie diejenigen Kontroll- und Datenabstraktionen bereitstellt, die für das Erstellen von kompakten und effizienten Systembeschreibungen notwendig sind. Außerdem wird diese Sprache von den meisten Entwicklern beherrscht und es existiert eine Vielzahl von Entwicklungswerkzeugen.

SystemC in Form der frei erhältlichen C++-Klassenbibliothek ermöglicht jedoch nur die Simulation der Systembeschreibung. Zur Synthese sind spezielle kommerzielle Kompiler erforderlich, z.B. der *CoCentric SystemC Compiler* [24] von *Synposys*.

Bei der bisher üblichen Designmethode zur Spezifikation ausführbarer Systeme erstellt der Entwickler zunächst ein C++-Modell des gesamten Systems. Damit kann er das Gesamtkonzept überprüfen. Danach erfolgt die Entscheidung, welche Teile des Systems in Hardware implementiert werden sollen. Diese werden dann manuell in den entsprechenden *VHDL*-Code übertragen (Abb. 6.4).

Dieses Verfahren besitzt jedoch einige Nachteile:

- Die manuelle Übertragung ist fehleranfällig. Es ist schwer, zu überprüfen, ob die *VHDL*-Implementierung wirklich der ursprünglichen C++-Systemspezifikation entspricht.

- Durch die manuelle Konvertierung entsteht eine Trennung zwischen der Systemebene und der Hardwarebeschreibungsebene. Änderungen in der Hardwarebeschreibung müssen manuell in die C++-Spezifikation zurückübertragen werden und umgekehrt. Dies kann leicht zu Inkonsistenzen führen.

- Testprogramme, die die C++-Spezifikation testen, können nicht ohne weiteres auf die Hardwareebene übertragen werden. Das bedeutet, daß nicht nur die C++-Spezifikation selbst, sondern auch die entsprechenden Testprogramme manuell konvertiert werden müssen.

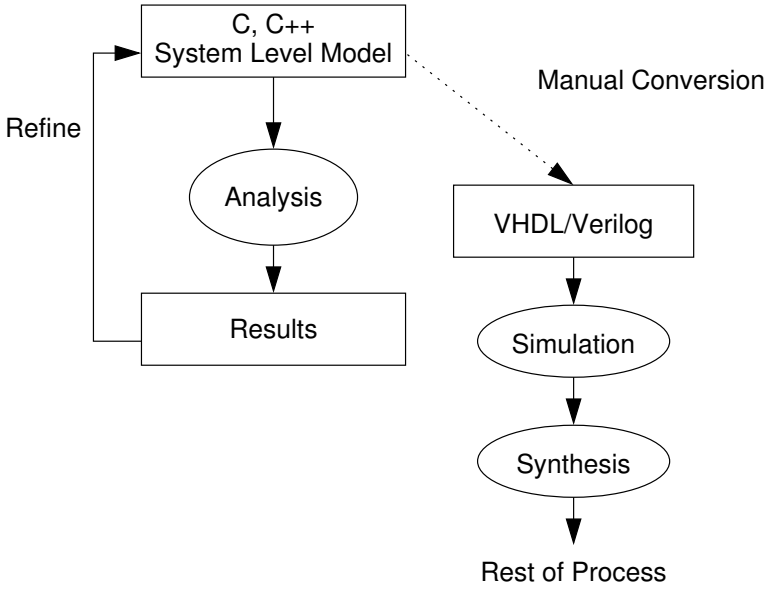


Abbildung 6.4: Konventionelle Designmethode

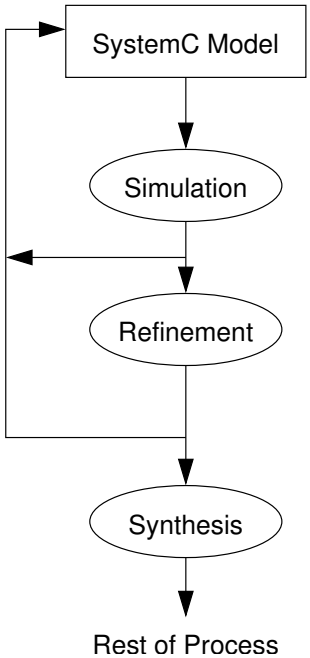


Abbildung 6.5: *SystemC*-Designmethode

Bei *SystemC* dagegen wird die Systemspezifikation von Anfang an in einer Sprachumgebung erstellt, die sowohl die Simulation als auch die spätere Synthese erlaubt (Abb. 6.5).

Dadurch ergeben sich folgende Vorteile:

- Da *SystemC* unterschiedliche Abstraktionsebenen unterstützt, kann die Systembeschrei-

bung schrittweise verfeinert werden. Es sind keine manuellen Konvertierungen notwendig, es können keine Inkonsistenzen entstehen.

- Der Entwickler muß nur eine Sprache, C++, beherrschen. Es ist zwar immer eine gewisse Detailkenntnis der verwendeten Sprache erforderlich, um effizient Designs implementieren zu können. Bei *SystemC* beschränkt sich dies jedoch auf eine einzige Sprache, im Gegensatz zur konventionellen Methode, bei der die Entwickler sowohl C++ als auch *VHDL* im Detail kennen müssen.

- Die Testprogramme können während der gesamten Phase der Designverfeinerung unverändert eingesetzt werden.

6.6.2 Hardwarebeschreibung

Ähnlich wie bei *VHDL* wird die Hardwarebeschreibung unter Verwendung von Prozessen vorgenommen. Es können beliebig viele Prozesse spezifiziert werden, die alle parallel nebeneinander existieren können [96, 94].

SystemC unterscheidet drei Arten von Prozessen:

- Methods (`SC_METHOD`).

  Dieser Prozeßtyp wird komplett ausgeführt, sobald er aktiviert wird. Danach gibt er die Kontrolle an den Simulator zurück. Er darf keine Endlosschleife enthalten. Dieser Prozeßtyp hat keinen eigenen Ausführungspfad. Es ist kein Aufruf von `wait()` zulässig.

- Threads (`SC_THREAD`).

  Dieser Prozeßtyp kann aktiviert und inaktiviert werden. Er enthält in der Regel eine Endlosschleife und gibt nach mehreren Anweisungen durch einen Aufruf von `wait()` die Kontrolle an den Simulator zurück. Er besitzt einen eigenen Ausführungspfad.

- Clocked Threads (`SC_CTHREAD`).

  Dies ist eine spezielle Variante von `SC_THREAD`. Er enthält nur eine Taktflanke in seiner Aktivierungsliste. Veränderungen der Ausgangssignale werden erst zur nächsten Taktflanke aktiv.

Das folgende Implementierungsbeispiel zeigt die Realisierung eines D-Flip-Flops zunächst in *VHDL*, danach in *SystemC*.

VHDL-Implementierung:

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
 port(clock : in  std_logic;
      din   : in  std_logic;
      dout  : out std_logic);
end dff;
architecture rtl of dff is
begin
  process
  begin
    wait until clock'event and clock = '1';
    dout <= din;
  end process;
end rtl;
```

SystemC-Implementierung:

```
#include "systemc.h"

SC_MODULE(dff)
{
  sc_in<bool> din;
  sc_in<bool> clock;
  sc_out<bool> dout;

  void doit()
   {
     dout = din;
   };

  SC_CTOR(dff)
   {
     SC_METHOD(doit);
     sensitive_pos << clock;
   }
};
```

6.6.3 Simulation

SystemC verwendet ein spezielles Ausführungsmodell, um parallele Prozesse simulieren zu können. Die einzelnen Prozesse haben einen eigenen Ausführungspfad und einen eigenen Stack. Der Wechsel zwischen den Prozessen erfolgt an genau definierten Punkten. Dies sind Aufrufe der `wait()`-Funktion. Dies entspricht einem nicht preemptiven Multitasking. Dadurch ist sichergestellt, daß die Prozesse nicht an anderen Punkten unterbrochen werden können. Prozesse rufen niemals andere Prozesse auf. Sie können jedoch Funktionen aufrufen, die keine Prozesse darstellen. Prozesse können andere Prozesse auslösen, indem sie Signale verändern, die sich in der Sensitivity-Liste dieser anderen Prozesse befinden. Von dem Code, der sich zwischen zwei `wait()`-Aufrufen befindet, wird angenommen, daß er für die Simulation parallel abläuft. Um dies zu erreichen, muß sichergestellt sein, daß die Reihenfolge der Prozesse keine Auswirkung hat. *SystemC* verwendet bei der Simulation sogenannte Delta Cycles, die jeweils aus einer Evaluierungs- und einer Updatephase bestehen. In einer bestimmten simulierten Zeit können mehrere dieser Delta Cycles durchlaufen werden. Die Reihenfolge der Prozeßausführung in der Evaluierungsphase ist nicht definiert, aber auch nicht relevant.

Der Simulator führt folgende Arbeitsschritte aus:

1. Initialisierungsphase.

   Alle Prozesse außer `SC_CTHREAD`s werden ausgeführt. Die Reihenfolge der Ausführung ist nicht definiert.

2. Evaluierungsphase.

   Alle Prozesse, die die Bedingungen für eine Aktivierung erfüllen, werden aktiviert. Ändern diese Prozesse sofort Ausgangssignale, können weitere Prozesse aktiviert werden.

3. Schritt 2 wird solange wiederholt, bis keine Prozesse mehr aktiviert werden können.

4. Update-Phase.

   Alle Änderungen von Ausgangssignalen, die während Schritt 2 angefordert wurden, werden ausgeführt.

5. Verzögerte Benachrichtigungen.

   Alle Prozesse, die verzögert benachrichtigt wurden, werden ausgeführt. Danach setzt die Bearbeitung mit Schritt 2 fort.

6. Wenn keine Zeitbenachrichtigungen vorliegen, ist die Simulation beendet.

7. Die Simulationszeit wird bis nur nächsten Zeitbenachrichtigung vorgestellt.

8. Feststellen, für welche Prozesse Zeitbenachrichtigungen für die aktuelle Simulationszeit vorliegen. Weiter mit Schritt 2.

6.6.4 Synthese

Die Synthese von *SystemC*-Hardwarebeschreibungen erfolgt nicht über die C++-Klassenbibliothek, sondern mittels speziellen *SystemC*-Kompilern.
Diese bearbeiten die zu synthetisierenden Module und generieren die Netzliste.

6.7 *Handel-C* (Version 3)

6.7.1 Allgemeines

Handel-C [69, 40] wurde ursprünglich an der *Oxford University* entwickelt und später von der neu gegründeten Firma *Celoxica* [20] übernommen.

Es handelt sich um eine im wesentlichen auf C aufbauende Hochsprache, die eine direkte Synthese in eine Netzliste erlaubt.

Integriert in die *DK1 Design Suite* von *Celoxica* ist *Handel-C* das zur Zeit am weitesten entwickelte kommerziell verfügbare Produkt dieser Art.

6.7.2 Hardwarebeschreibung

Handel-C verwendet zur Hardwarebeschreibung eine eingeschränkte C-Syntax, die um einige Zusatzoperatoren und -schlüsselworte erweitert wurde [21, 18, 19].

Im Gegensatz zu den C++- bzw. *JAVA*-basierten Systemen *PamDC*, *JHDL* und *SystemC* wird hier C nicht lediglich benutzt, um Hardwarestrukturen zu beschreiben. Die ursprüngliche Bedeutung der Formulierungen bleibt vielmehr erhalten und wird unter Beibehaltung des Ausführungsmodells in eine entsprechende Hardwarestruktur übersetzt.

Handel-C stellt die üblichen Konstrukte zur Angabe des Kontrollflusses bereits, so etwa `if...else`, `for`, `while`, `switch` usw.

Das zeitliche Verhalten eines *Handel-C*-Programmes basiert auf der Grundregel, daß jede Anweisung einen Taktzyklus benötigt. Ausgenommen sind die oben genannten Kontrollanweisungen, die direkt in kombinatorische Logik umgesetzt werden und daher keinen eigenen Taktzyklus benötigen. Eine weitere Ausnahme bilden Funktionsaufrufe und Channel-Zugriffe, die mehrere Takte benötigen können.

Um die Vorteile von Hardware nutzen zu können, muß es möglich sein, Anweisungen parallel auszuführen. *Handel-C* ermöglicht die Implementierung von parallelen Anweisungen. Diese Parallelität muß jedoch vom Entwickler mittels `par{...}`-Anweisungen explizit angegeben werden. Es existiert keine Optimierung, bei der unabhängige Anweisungen automatisch parallelisiert werden.

Spaltet sich der Kontrollfluß, um solche parallelen Anweisungen auszuführen, und benötigen die einzelnen Teilpfade eine unterschiedliche Anzahl von Takten, so warten alle kürzeren, bis der längste Pfad abgearbeitet ist (Abb. 6.6).

In einem FPGA-System ist es oft erforderlich, daß mehrere parallel ablaufende Prozesse auf eine gemeinsame Resource, etwa einen SDRAM-Kontroller, zugreifen müssen. Diese gemeinsame Resource kann zu jedem Zeitpunkt nur von einem Prozeß belegt sein. Solange sie einen Auftrag bearbeitet, kann kein anderer Prozeß auf sie zugreifen. Ist sie frei und liegen zum gleichen Zeitpunkt mehrere Zugriffsanforderungen vor, muß eine Prioritätslogik entscheiden, welcher Prozeß den Zugriff erhält.

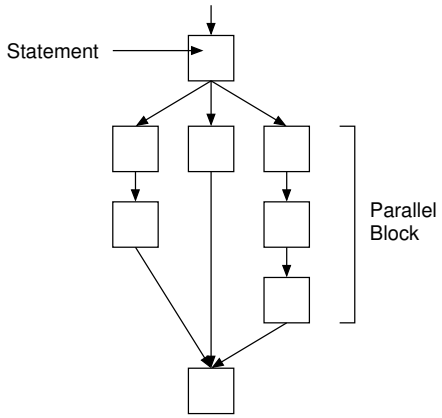


Abbildung 6.6: *Handel-C*: Branching und Re-Joining

Handel-C stellt keine automatische Unterstützung für diese Problematik bereit. Der Entwickler muß manuell Methoden einfügen, um zu verhindern, daß ein gemeinsamer Prozeß zeitgleich von mehreren anderen Prozessen aufgerufen wird.

Dies kann durch Semaphoren, die von *Handel-C* bereitgestellt werden, erfolgen. Sie sichern die jeweiligen Prozeßaufrufe entsprechend ab:

```
sema  s1;

while (...)
 {
   trysema(s1);
   ...
   releasesema(s1);
 }
```

Eines der zentralen Konzepte von *Handel-C* sind *Channels*. Diese werden als Link zwischen parallelen Prozessen eingesetzt. Ein Prozeß schreibt Daten in den Channel, ein anderer liest sie aus. Neben dem eigentlichen Datentransfer implementieren Channels eine Synchronisation zwischen den beiden beteiligten Prozessen, so daß der Prozeß, der früher die betreffende Stelle im Ablauf erreicht, solange wartet, bis auch der andere Prozeß zum Transfer bereit ist. Dieses Verfahren bildet eine Alternative zum konventionellen Unterprogrammaufruf.

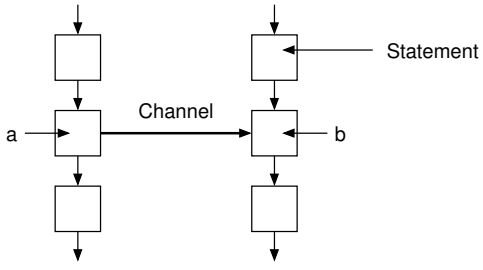


Abbildung 6.7: *Handel-C*: Channels

Die `prialt`-Anweisung stellt eine spezielle Unterstützung für Channels bereit. Sie ermöglicht die ständige Überwachung mehrerer Channels, ob diese zum Schreiben oder Lesen bereit sind. Ist ein Channel bereit, werden die entsprechenden Anweisungen ausgeführt. Für den Fall, daß zu einem Zeitpunkt mehrere Channels bereit werden, sorgt eine Prioritätslogik dafür, daß die Bearbeitung in der angegebenen Reihenfolge abläuft:

```
prialt
 {
   case channel1 ? value1:
      ...
     break;
   case channel2 ? value2:
      ...
     break;
   default:
      ...
     break;
 }
```

Mit dieser Anweisung läßt sich ein Prozeß implementieren, der über Channels Daten von mehreren anderen Prozessen annehmen kann, wobei die Prioritäten der einzelnen Channels klar definiert sind. Dies ist immer dann notwendig, wenn mehrere Prozesse auf eine gemeinsame Resource zugreifen müssen (Abb. 6.8).

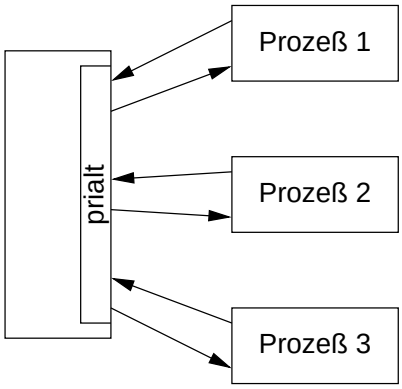


Abbildung 6.8: *Handel-C*: Zugriff auf gemeinsame Resourcen

Diese Methode gleicht das Fehlen eines Arbitrationsmechanismus für den Aufruf gemeinsamer Funktionen aus, gibt der entstehenden Hardwarebeschreibung jedoch ein strukturelles Aussehen, das stark an *VHDL*-basierte Designs erinnert.

Zusätzlich zu den üblichen C-Operatoren stehen folgende spezielle Operatoren und Anweisungen zur Verfügung:

- *Ausdruck* `<-` *Anzahl*

  Übernimmt die angegebene Anzahl niederwertiger Bits von *Ausdruck*.

- *Ausdruck* `\\` *Anzahl*

  Verwirft die angegebene Anzahl niederwertiger Bits von *Ausdruck*.

- *Ausdruck1* `@` *Ausdruck2*

  Verwirft die durch *Ausdruck2* ermittelte Anzahl niederwertiger Bits von *Ausdruck1*.

- *Channel* `!` *Expression*

  Schreibt das Ergebnis von *Expression* in den Channel.

- *Channel* `?` *Variable*

  Liest den Channel und schreibt den Wert in die Variable.

- `prialt` {*Anweisungen*}

  Testet verschiedene Channels auf Verfügbarkeit und führt entsprechende Anweisungen aus.

- `trysema`

  Testet, ob eine Semaphore angefordert werden kann.

- `releasesema`

  Gibt eine Semaphore wieder frei.

Weiterhin sind folgende zusätzliche Datentypen vorhanden:

- `char`, `short`, `int`, `long`, `int` n, jeweils *signed* bzw. *unsigned*.

- `chan`, `chanin`, `chanout`.

  Channels für Synthese und Simulation.

- `ram`, `rom`, `wom`, `mpram`.

  Verschiedene Arten von Speicher (RAM, ROM, Write-Only-Memory, Multiport-RAM).

- `sema`.

  Semaphore.

Die Schnittstelle von Modulen zu ihrer Umgebung kann durch verschiedene Interfaces implementiert werden:

- `bus_in`, `bus_out`, `bus_ts`.

  Diese Interfaces implementieren Schnittstellen innerhalb des FPGAs. Damit lassen sich unterschiedliche Module miteinander verbinden.

- `port_in`, `port_out`.

  Diese repräsentieren die Schnittstellen des FPGAs zu seiner Umgebung.

- Benutzerdefinierte Interfaces.

  Hiermit lassen sich DLL-Funktionen in die Simulation einbinden oder Module integrieren, die mit einer anderen Hardwarebeschreibungssprache implementiert wurden (z.B. EDIF-Netzlisten aus *VHDL*-Beschreibungen).

Das folgende Beispielprogramm zeigt die Implementierung von zwei parallel ablaufenden Prozessen `process1` und `process2`. Der erste Prozeß sendet nacheinander die Werte 0 bis 9 in einen Channel. Er wartet dabei automatisch immer solange, bis der Wert vom anderen Prozeß gelesen wurde. Die verwendeten Variablen sind 8 Bit breite Integer-Werte.

```
clock...

chanout log;

void  process1 ( chanout& channel )
{
  int 8 i;

  i = 0;
  while (i < 10)
   {
```

```
      channel ! i;
      i++;
    }
}

void  process2 ( chanin& channel )
{
  int 8 data;

  while (1)
   {
     channel ? data;
     log ! data;
   }
}

void  main()
{
  chan int 8 channel;

  par
   {
     process1(&channel);
     process2(&channel);
   }
}
```

6.7.3 Simulation

Die Simulation erfolgt über die `chanin`- und `chanout`-Schnittstellen.

Die `chanin`-Elemente können für die Simulation so konfiguriert werden, daß sie Datenworte aus einer externen Datei lesen und als Stimuli für die Simulation verwenden. Entsprechend können die `chanout`-Elemente die an ihnen auflaufenden Daten in eine Datei schreiben.

Weiterhin ist eine DLL-Schnitstelle vorhanden, über die durch externen C-Code beliebige Funktionalität an `chanin` und `chanout` angekoppelt werden kann.

Die *DK1 Suite* enthält verschiedene Hilfsprogramme zum Durchführen der Simulation und zum Visualisieren der Simulationsergebnisse.

6.7.4 Synthese

Aus der Hardwarebeschreibung kann direkt eine EDIF-Netzliste erzeugt werden. Auch der Export von automatisch erzeugtem *VHDL*-Code ist möglich.

6.8 Analyse der existierenden Systeme

6.8.1 Allgemeines

Es existieren zur Zeit vier Typen von Entwicklungssystemen, mit denen Designs für FPGAs erstellt werden können:

- *VHDL*-basierte Entwicklungssysteme.
- Systeme, die eine strukturelle Hardwarebeschreibung mit konventionellem C++ bzw. *JAVA* realisieren (*PamDC*, *JHDL*).
- Systeme, die C++ zum Erstellen von ausführbaren Spezifikationen einsetzen (*SystemC*).
- Hochsprachenorientierte Entwicklungssysteme mit eigenem Kompiler (*Handel-C*).

In den folgenden Ausführungen wird untersucht, inwieweit diese Systeme die Anwendung von FPGA-Koprozessoren unterstützen.

6.8.2 Hardwarebeschreibung

Im Bereich der Hardwarebeschreibung muß das Entwicklungssystem den Anwender in erster Linie bei der Beherrschung von Umfang und Komplexität unterstützen. Dies gilt besonders für die Detailstrukturen, die bei der Erstellung zeitkritischer Designs notwendig sind.

VHDL-basierte Systeme weisen bei der Entwicklung größerer Projekte Schwächen durch den Sprachaufbau von *VHDL* auf. So sind die Möglichkeiten zur Parametrisierung begrenzt und es fehlen Methoden zur Bildung höherer Beschreibungsebenen. In umfangreichen, komplexen Anwendungen bietet diese Sprache wenig Unterstützung die Komplexität mit kompakten Beschreibungen beherrschen zu können. Auch die Spezifikation von Detailinformationen, etwa des Startzustandes von Speicherelementen oder Plazierungsanweisungen, werden nicht von allen Kompilern unterstützt.

Obwohl *VHDL* speziell zur Beschreibung von Hardwarestrukturen entworfen wurde und als Industriestandard gilt, weisen universelle Programmiersprachen wie C/C++ oder *JAVA* einige wichtige Vorteile auf [14, 43]:

- Universelle Programmiersprachen besitzen eine größere Mächtigkeit. Der Entwickler kann Funktionen wie Konsolen-I/O oder Zugriffe auf Dateien einsetzen, um Programme einfacher zu schreiben und zu testen. Durch Einsatz objektorientierter Techniken wie Kapselung, Vererbung, virtuelle Methoden und überladene Operatoren können die Vorteile der modernen Softwareentwicklung genutzt werden. Die objektorientierte Struktur der modernen Programmiersprachen ist optimal für die Hardwarebeschreibung geeignet. Durch hierarchische Beschreibung kann das Erstellen von komplexen Beschreibungen deutlich vereinfacht werden.
- Die meisten Entwickler, die mit Koprozessoren oder Embedded Systems arbeiten, besitzen bereits sehr gute Kenntnisse von C oder C++. Sie müssen auf diese Weise keine zusätzliche Sprache beherrschen.
- Es existieren viele bewährte Entwicklungsumgebungen für C/C++. Sie umfassen auch Source-Level-Debugger und sind zum Teil frei verfügbar (z.B. gcc für *Linux*).
- Sowohl für die Hardwarebeschreibung als auch für die Softwareentwicklung kann dieselbe Programmiersprache verwendet werden. Dies schafft die Möglichkeit für viele neue Anwendungen in den Bereichen Hardware-Software-Codesign und -Cosimulation.

Besonders die Tatsache, daß mit C/C++ dieselbe Programmiersprache sowohl für den Hardware- als auch für den Softwarebereich eingesetzt werden kann, ist beim Einsatz von FPGA-Koprozessoren von großer Bedeutung: Für beide Bereiche können dieselben Entwicklungswerkzeuge verwendet werden. Durch die vertraute Sprache wird auch den Softwareentwicklern der Hardwarebereich eher zugänglich.
Die C++ bzw. *JAVA*-basierten Systeme zeigen jedoch in der Struktur ihrer Hardwarebeschreibung deutliche Schwächen. So werden teilweise Anweisungen für Busse nicht unterstützt (*PamDC*) oder die Verbindung der einzelnen Komponenten untereinander erfolgt umständlich prozedural über Parameterlisten (*JHDL*). Die Möglichkeiten, die sich aus der Verwendung einer mächtigen universellen Programmiersprache ergeben, werden nicht konsequent genutzt. Insbesondere setzen diese Systeme die Konzepte der Objektorientierung nicht konsequent ein. Diese stellen jedoch eine wertvolle und bewährte Methode zur Beherrschung von Umfang und Komplexität dar.

Weiterhin fehlen integrierte Verfahren zur Bildung höherer Beschreibungsebenen, so etwa Zustandsmaschinen.

Die erwähnten Beschränkungen dieser Systeme beruhen vermutlich auf der Problematik, daß das zugrundeliegende Ausführungsmodell von C/C++ bzw. *JAVA* an der konventionellen Softwareentwicklung orientiert ist. Für eine Hardwarebeschreibung ist dieses nicht direkt einsetzbar. Je nachdem, wie die Modifizierung des Ausführungsmodells erfolgt, entstehen in den unterschiedlichen Systemen funktionale Einschränkungen. Es wäre zu untersuchen, inwieweit es dennoch möglich wäre, solche Einschränkungen zu vermeiden.

Das System *Handel-C* bietet nur die hochsprachenorientierte Beschreibungsform. Es sind keine Möglichkeiten für eine integrierte Einbindung von effizientem strukturellen Code vorhanden. Teilimplementierungen auf anderen Beschreibungsebenen sind nur Einsatz externer *VHDL*-Module realisierbar.

In vielen aktuellen FPGA-Anwendungen spielen Geschwindigkeit und effiziente Resourcennutzung eine große Rolle. Daher ist der Entwickler oft darauf angewiesen, besonders kritische Komponenten seines Designs auf unterster Ebene zu implementieren. Nur so kann er die Resourcen der FPGAs optimal ausnutzen.

Der Hochsprachenkompiler selbst kann einige effiziente Strukturen nicht automatisch bilden. So wird etwa Pipelining nicht automatisch unterstützt. Auch jede Form von Parallelität muß explizit in die Beschreibung eingefügt werden.

Pipeline-Strukturen können zwar mittels Arrays aus Variablen und den entsprechenden Zuweisungen implementiert werden, dabei muß der Entwickler jedoch auch die gesamte Flußsteuerung (z.B. für einen Pipeline-Stall) selbst realisieren. Die erforderlichen Formulierungen können dann umfangreicher sein als in einer entsprechenden strukturellen Implementierung.

Einige Einschränkungen von *Handel-C* können sich insbesondere bei Designs, in denen die Ausführungsgeschwindigkeit eine große Rolle spielt, negativ auswirken. So kann etwa ein Channel nur direkt in eine Variable eingelesen werden. Es ist nicht möglich, den Wert aus dem Channel im gleichen Takt mit einem anderen Wert zu verrechnen. Auf diese Weise sind im Ablauf zusätzliche Takte erforderlich. Besonders in Anwendungen, die Online-Daten verarbeiten, bei denen kein Anhalten des Datenstroms möglich ist, kann dies problematisch sein.

Durch die Spracheinschränkungen, die zusätzliche Taktzyklen erfordern, werden auch die Möglichkeiten zum Aufbau von Pipelines beschränkt.

SystemC benutzt C++ zur Modellierung verhaltensorientierter Beschreibungen, wobei durch Kompilierung mit einem handelsüblichen Kompiler eine ausführbare Spezifikation des Gesamtsystems entsteht. Diese kann direkt eine Simulation durchführen.

Die Form der verhaltensorientierten Beschreibung ist stark an *VHDL* angelehnt. Dadurch besteht auch hier wie bei den Systemen *PamDC* und *JHDL* die Problematik des Ausführungsmodells. Die bei *SystemC* gewählte Anpassungsmethode hat zur Folge, daß die Hardwarebeschreibung zwar simulierbar, jedoch nicht synthetisierbar ist. Auch führt die enge Anlehnung

an *VHDL* zu Einschränkungen, so daß die Mächtigkeit der Sprache C++ nicht mehr vollständig genutzt werden kann.

Zur Synthese der Hardwarebeschreibung muß ein separater Kompiler verwendet werden. Nicht alle Konstrukte der Beschreibung sind auch synthetisierbar.

Durch diese Situation entsteht eine Trennung zwischen Simulation und Synthese, da nicht jedes sprachliche Konstrukt, das für die Simulation zulässig ist, auch synthetisierbar sein wird. Auch werden die Syntheseergebnisse der *SystemC*-Kompiler ähnlich wie bei den *VHDL*-Kompilern im Detail schwer vorhersehbar sein und von Produkt zu Produkt variieren.

Prinzipiell haben die C/C++/*JAVA*-basierten Systeme durch ihre mächtige Beschreibungssprache das größte Potential für eine optimale Unterstützung der Hardwarebeschreibung für FPGA-Koprozessoren. Die konkrete Realisierung der Beschreibungsform weist jedoch Einschränkungen auf, durch die nicht alle zur Verfügung stehenden Möglichkeiten genutzt werden können.

6.8.3 Simulation

Für die Simulation von FPGA-Koprozessoranwendungen ist die Integration der FPGA-Umgebung von erheblicher Bedeutung.

VHDL benutzt Testbenches zur Emulation der Umgebungshardware. Die Implementierung dieser Testbenches erweist sich bei der Modellierung komplexer Bausteine, wie etwa Mikrokontrollern oder Bus-Bridges als schwierig, da *VHDL* dazu weniger mächtig ist als konventionelle Programmiersprachen. Umfangreiche Testbenches führen darüberhinaus zu langen Simulationszeiten. Probleme verursachen insbesondere Komponenten mit hohem Speicherbedarf, etwa bei der Integration großer SDRAM-Bausteine in die Simulation.

Einige *VHDL*-Simulatoren verfügen zwar über DLL-basierte Schnittstellen, um externe C/C++-Funktionen in die Simulation einzubinden. Dies führt jedoch bei komplexen Systemen mit einer Vielzahl externer Komponenten zu einer sehr heterogenen Anordnung, die schwer handhabbar werden kann.

Eine ähnliche Problematik der Schnittstellen existiert bei *Handel-C*. Hier müssen komplexe Simulationen fast immer über DLL-Schnittstellen durchgeführt werden, da *Handel-C* selbst keine Erstellung von Testbenches unterstützt.

Die C/C++/*JAVA*-basierten Systeme mit universellen Programmiersprachen als Grundlage bieten dagegen effiziente Möglichkeiten, die FPGA-Umgebung zu emulieren. Der Programmcode, der die Emulation vornimmt, kann leicht integriert werden. Es sind keine DLL-Schnittstellen erforderlich.

Die vorhandenen Implementierungen von *PamDC* und *JHDL* weisen jedoch einige Einschränkungen auf, die eine Simulation umfangreicher und komplexer FPGA-Koprozessoranwendungen erschweren oder sogar unmöglich machen. So beschränkt *JHDL* die Simulation auf einen einzigen globalen Takt. Asynchrone Logik kann nicht simuliert werden. *PamDC* besitzt ähnliche Einschränkungen. Hier ist die Simulation von asynchroner Logik allgemein sowie von kombinatorischer Logik im Taktpfad nicht unterstützt.

SystemC ermöglicht ebenfalls eine einfache Integration des Emulationscodes. Dieses System besitzt jedoch den großen Nachteil, daß die spätere Synthese nicht mithilfe der Klassenbibliothek, sondern mit einem separaten Kompiler erfolgt. Dadurch beruhen Simulation und Synthese nicht auf der gleichen Datenbasis. Bei einer unterschiedlichen Umsetzung der Hardwarebeschreibung können sich Abweichungen zwischen Simulation und Echtzeitbetrieb ergeben.

Ein weiteres wichtiges Kriterium ist die Simulation der engen Mikroprozessorankopplung von FPGA-Koprozessoren, um eine hohe Übereinstimmung von Simulation und Synthese zu erreichen.

Hier sind *VHDL*-basierte Systeme schon aufgrund der sprachlichen Trennung zwischen Hardware- und Softwarebereich ungeeignet. Da die beiden Bereiche mit unterschiedlichen Entwicklungssystemen implementiert werden, sind die Möglichkeiten zu einer integrierten Simulation stark eingeschränkt.

Die besten Möglichkeiten bieten hier die C/C++-basierten Systeme, da sie dieselbe Sprache für die Hardwarebeschreibung und die Steuersoftware einsetzen.

Die Steuersoftware kann mit minimalen Änderungen sowohl für die Simulation als auch für den Echtzeitbetrieb verwendet werden. Es entfällt somit die sonst notwendige Doppelentwicklung etwa für *VHDL*-Testbenches.

Auch die Simulation von Zugriffen auf Pseudoregister läßt sich am einfachsten mit C/C++ realisieren, da diese Sprache im Gegensatz etwa zu *JAVA* die notwendigen Zugriffe mittels Zeigern direkt unterstützen.

Bei *Handel-C* ist eine integrierte Simulation von Hardware- und Softwarebereich nur mit einer heterogenen Anordnung unter Verwendung von DLL-Schnittstellen möglich. Es wird zwar für beide Bereiche annähernd die gleiche Beschreibungssprache verwendet, jedoch sind unterschiedliche Entwicklungssysteme erforderlich. Daher besteht hier eine ähnliche Problematik wie bei den *VHDL*-basierten Systemen.

Aufgrund der möglichen Abweichungen zwischen Simulation und Synthese kann bei *SystemC* eine hohe Übereinstimmung von Simulation und Echtzeitbetrieb prinzipiell nicht erreicht werden.

Für die Simulation von FPGA-Koprozessoren bieten C/C++-basierte Systeme die beste Unterstützung. Die enge Kopplung zwischen Mikroprozessor und FPGA kann aufgrund der einheitlichen Sprache für Hardware- und Softwarebereich direkt und ohne Umwege über DLL-Schnittstellen simuliert werden. Die gesamte Simulation läßt sich aus einer einzigen Entwicklungsumgebung heraus realisieren.

6.8.4 Synthese und Hardware-Debugging

Trotz hoher Übereinstimmung von Simulation und Synthese kann bei FPGA-Koprozessoren ein Hardware-Debugging erforderlich sein. Daher sollte das Entwicklungssystem die spätere Anwendung moderner Debugging-Mechanismen wie Readback oder partieller Rekonfiguration unterstützen.

Das zentrale Problem stellt dabei die Übernahme der im Design vergebenen Namen in die Netzliste dar.

VHDL, *SystemC* und *Handel-C* nehmen Veränderungen an den Namen vor, bzw. ermöglichen keine direkte Vergabe von Namen an einzelne Elemente. Es ist daher schwierig, einem Namen in der Netzliste eindeutig sein entsprechendes Element in der ursprünglichen Hardwarebeschreibung zuzuordnen. Gerade diese Zuordnung ist aber für eine automatisierte Anwendung moderner Debugging-Verfahren wie etwa Readback unverzichtbar.

Auch bei den C/C++/*JAVA*-basierten Systemen bestehen nur eingeschränkte Möglichkeiten, gezielt Einfluß auf die Benennung der Komponenten auszuüben.

Um eine möglichst hohe Übereinstimmung von Simulation und Echtzeitbetrieb zu erreichen, wäre es wünschenswert, auch eine Simulation der Readback-Funktion zu besitzen. Damit können die Soll-Werte der Readback-Testpunkte mit den Ist-Werten des Echtzeitbetriebs verglichen werden, um Fehler schnell lokalisieren zu können.

Eine solche direkte Unterstützung von Verfahren zum Hardware-Debugging ist zur Zeit in keinem der verfügbaren Entwicklungssysteme vorhanden.

Abbildung 6.9 zeigt abschließend eine kompakte Übersicht über die wesentlichen Eigenschaften der untersuchten Entwicklungssysteme.

System	Verwendete Sprache	Ebenen der Hardwarebeschreibung	Übersetzungsprozeß	Unterstützung für Zustandsmaschinen	Simulationsmöglichkeit	Sprache der Testbenches	Synthese	Unterstützung für Readback
VHDL-basierte Systeme	*VHDL*	strukturell, verhaltensorientiert	*VHDL*-Kompiler	Ja	Ja	*VHDL* bzw. C++ über DLL-Schnittstelle	integriert	Nein
PamDC	C++	strukturell	C++-Klassenbibliothek	Nein	eingeschränkt	C++	integriert	Nein
JHDL	*JAVA*	strukturell	*JAVA*-Klassenbibliothek	Nein	eingeschränkt	*JAVA*	integriert	Nein
SystemC	C++	verhaltensorientiert, Hochsprache	Simulation: C++-Klassenbibliothek, Synthese: *SystemC*-Kompiler	Prozesse	Ja	C++	separater Kompiler	Nein
Handel-C	erweitertes C	Hochsprache	*Handel-C*-Kompiler	Prozesse	Ja	C++ über DLL-Schnittstelle	integriert	Nein

Abbildung 6.9: Übersicht der Entwicklungssysteme

Kapitel 7
Zusammenfassung

Im vorausgegangenen Teil II wurden zunächst Grundprinzipien der digitalen Logik sowie Methoden zur Optimierung digitaler Schaltungen vorgestellt. Die Darstellung orientierte sich dabei an denjenigen speziellen Eigenschaften der Logikbausteine, die der Entwickler berücksichtigen muß, um bei der Umsetzung von Algorithmen auf Hardware eine optimale Ausführungsgeschwindigkeit zu erreichen. Wesentliche Faktoren sind die maximale Taktfrequenz sowie die Anzahl der notwendigen Takte. Die Optimierung der Hardwarestruktur kann mit den Verfahren der Parallelisierung und des Pipelinings erfolgen.

Nachfolgend wurden das Grundkonzept der FPGAs sowie die wesentlichen Unterschiede dieser Bausteine zu den konventionellen PLDs erörtert. Der bedeutendste Unterschied liegt in der Anordnung der einzelnen Logikblöcke. Diese sind bei PLDs eindimensional an eine zentrale Schaltmatrix gekoppelt, die gleichzeitig den begrenzenden Faktor der erreichbaren Logikdichte darstellt. Bei FPGAs befinden sich dagegen die Logikblöcke selbst in einer zweidimensionalen Matrixanordnung. Die Verbindungsresourcen sind nicht zentralisiert, sondern verteilt. Diese Architektur ist beliebig skalierbar. FPGAs können jede Verbesserung der Herstellungsprozesse quadratisch nutzen, wodurch sich die Logikdichte immer weiter erhöhen wird.

Als Folge der Dezentralisierung der Verbindungsresourcen geht jedoch das von PLDs bekannte vorhersehbare Zeitverhalten verloren. Die Verzögerungszeiten der Verbindungen sind bei FPGAs in hohem Maße von der geometrischen Zuordnung der Logik auf die Logikblöcke abhängig. Die Place&Route-Software verfügt zwar über Algorithmen, die diese Zuordnung optimieren, jedoch ist deren Leistungsfähigkeit insbesondere bei zeitkritischen Schaltungen begrenzt. Die Berücksichtigung der Verzögerungszeiten stellt in der Praxis eines der zentralen Probleme bei der Anwendung von FPGAs dar.

Der konkrete Aufbau moderner FPGAs wurde am Beispiel der *Virtex*-Architektur erläutert. Die Beschreibung orientierte sich an den vorhandenen Resourcen und deren prinzipiellen Konfigurationsmöglichkeiten. Es wurde gezeigt, daß die Logikblöcke über leistungsfähige Strukturen verfügen, mit denen sich effiziente Schaltungen realisieren lassen. Dazu kann es allerdings notwendig sein, eine Vielzahl von Details zu spezifizieren. Da die vorhandenen Optimierungsalgorithmen bei zeitkritischen Designs nicht ausreichen, benötigt der FPGA-Anwender bei umfangreichen Schaltungen eine Hilfestellung durch geeignete Software. Diese muß ihn dabei unterstützen, trotz notwendiger Detailangaben Umfang und Komplexität der Gesamtschaltung zu beherrschen.

Das Konzept der FPGA-Koprozessoren kann dazu beitragen, die Schwächen und Engpässe der konventionellen Mikroprozessortechnik zu überwinden. Es wurde verdeutlicht, welche Vorteile, aber auch Probleme des Hardware-Software-Codesigns sich hinter der engen Kopplung zwischen FPGA und Mikroprozessor verbergen. Durch die steigende Logikdichte der FPGAs vergrößert sich auch das Anwendungsgebiet der FPGA-Koprozessoren laufend. Der Entwickler wird daher mit immer umfangreicheren und komplexeren Designs konfrontiert.

Zur Erstellung von Schaltungen für FPGAs stehen dem Entwickler mehrere Softwarepakete zur Verfügung. Diese bieten verschiedene Methoden, die zu realisierende Schaltung in Textform zu spezifizieren. Die entstandene Hardwarebeschreibung kann dann simuliert und schließlich zu einer Netzliste synthetisiert werden. Die Darstellung der wichtigsten zur Zeit verfügbaren FPGA-Entwicklungssysteme orientierte sich an diesem Erstellungsprozeß. Neben den *VHDL*-basierten Systemen wurden *PamDC*, *JHDL*, *SystemC* und *Handel-C* vorgestellt.

Im Rahmen einer Analyse dieser Entwicklungssysteme wurde gezeigt, daß sie bei der Verwendung mit modernen FPGA-Koprozessoren in allen Bereichen des Designerstellungsprozesses Schwächen aufweisen.

Die *VHDL*-basierten Entwicklungssysteme bieten dem Anwender keine ausreichende Unterstützung für die Beherrschung umfangreicher und komplexer Schaltungen. Eine effiziente Simulation kompletter Systeme ist nur mit heterogenen Kombinationen mehrerer Softwarepakete und DLL-Schnittstellen möglich. *VHDL*-Konstrukte zeigen eine hohe Abhängigkeit vom eingesetzten Kompiler und ein direkter Zugriff auf die FPGA-Resourcen ist nur eingeschränkt realisierbar.

Mit *SystemC* kann zwar der Einsatz von *VHDL* vermieden werden, es verwendet jedoch eine eng an *VHDL* angelehnte verhaltensorientierte Hardwarebeschreibung. Dadurch kann die *SystemC*-Klassenbibliothek nur zur Simulation, nicht aber zur Synthese verwendet werden. Zur Synthese ist ein spezieller Kompiler erforderlich, nicht alle bei der Simulation zulässigen Konstrukte sind damit auch synthetisierbar. Somit kann bei *SystemC* keine Übereinstimmung von Simulation und Synthese garantiert werden.

Das *Handel-C*-System ermöglicht nur eine Hochsprachenbeschreibung. Diese stellt jedoch keine ausreichenden Konstrukte für die Erstellung effizienter Designs bereit. Für Teilimplementierungen auf niedriger Ebene muß dann auf *VHDL*-basierte Werkzeuge zurückgegriffen werden. Der integrierte Einsatz verschiedener Abstraktionsebenen ist nicht möglich.

Das größte Potential bieten C/C++- bzw. *JAVA*-basierte Systeme, die als Klassenbibliothek realisiert sind. Für die Designerstellung können die herkömmlichen Entwicklungswerkzeuge verwendet werden. C/C++ besitzt gegenüber *JAVA* den Vorteil, daß es die gängige Implementierungssprache für FPGA-Koprozessoren ist und über alle für das Hardware-Software-Codesign notwendigen Konstrukte (etwa Zeiger) verfügt. Die vorgestellten Systeme *PamDC* und *JHDL* sind solche Bibliotheksimplementierungen. Sie nutzen jedoch die Möglichkeiten, die sich aus der Verwendung der mächtigen Sprachen C++ bzw *JAVA* ergeben, bei weitem nicht aus. Die Struktur der Hardwarebeschreibung wirkt umständlich und besitzt einen eher prozeduralen als objektorientierten Charakter. Die Simulation weist Einschränkungen auf, die das Simulieren komplexer Schaltungen erschwert oder sogar unmöglich macht.

Ein Entwicklungssystem, das in der Lage wäre, Anwendungen für FPGA-Koprozessoren optimal zu unterstützen, müsste folgende Eigenschaften aufweisen:

- Einsatz einer mächtigen konventionellen Programmiersprache zur Hardwarebeschreibung. Dabei sollte es sich bevorzugt um C/C++ handeln, da damit auch überwiegend die Softwareanwendungen für FPGA-Koprozessoren implementiert werden.
- Nutzung handelsüblicher Kompiler und C/C++-Entwicklungsumgebungen.
- Unterstützung hierarchischer und parametrisierbarer Hardwarebeschreibungen.
- Bereitstellung mehrerer Abstraktionsebenen auf homogene Art und Weise.
- Uneingeschränkte Simulationsmöglichkeiten, einschließlich effizienter Simulation gesamter Systeme.
- Nutzung derselben Datenbasis für Simulation und Synthese.
- Unterstützung enger FPGA-Mikroprozessorkopplungen in der Simulation.
- Synthese von Netzlisten, die Readback und partielle Rekonfiguration ermöglichen.
- Insgesamt muß es sich um ein homogenes System handeln, das Hardwarebeschreibung, Simulation, externe Simulationsmodelle und Echtzeitbetrieb mit derselben Sprache und übereinstimmenden Methoden realisiert.

Im folgenden Teil III wird das FPGA-Entwicklungssystem *CHDL* vorgestellt, das im Rahmen dieser Arbeit implementiert wurde und die oben genannten Anforderungen erfüllt.

Teil III

Das FPGA-Entwicklungssystem *CHDL*

Kapitel 8
Einführung

Das FPGA-Entwicklungssystem *CHDL* ist in Form einer C++-Klassenbibliothek implementiert und ermöglicht eine Hardwarebeschreibung mittels C++ auf mehreren Abstraktionsebenen. Die unterste Ebene ist rein strukturell orientiert. Es können in hierarchischer Anordnung Bauteile erzeugt und miteinander verschaltet werden. Darauf aufbauend ist ein Verfahren zur Beschreibung von Zustandsmaschinen implementiert. Beide Beschreibungsmethoden verwenden C++ in unveränderter Syntax und können daher von jedem C++-Kompiler verarbeitet werden. Durch Nutzung üblicher Methoden der Sprache C++, wie Parametrisierung, Klassenbildung, Vererbung usw., kann der Anwender nahezu beliebige weitere Abstraktionsebenen hinzufügen.

Beim *CHDL*-System wurde das Konzept des Systementwurfes mit C++ so weiterentwickelt, daß die Prinzipien der Objektorientierung optimal genutzt werden können. *CHDL* stellt dem Entwickler die Möglichkeiten, die C++ zur Beherrschung umfangreicher und komplexer Anwendungen bietet, auch bei der Hardwarebeschreibung zur Verfügung.

Es wird das gleiche Grundprinzip verwendet, auf dem auch andere Systeme wie z.B. *PamDC* oder *JHDL* beruhen: Logische Grundbausteine (Flip-Flops, Gatter usw.) werden durch Klassen repräsentiert. Das Erzeugen eines Grundelements erfolgt durch Instanziierung eines Objektes der betreffenden Klasse. Das Verbinden der Objekte untereinander kann mittels speziellen Funktionen oder Operatoren durchgeführt werden.

Die Hardwarebeschreibung wird wie ein übliches C++-Programm kompiliert. Beim Ausführen erfolgt die Simulation bzw. die Erzeugung der Netzliste. Die Realisierbarkeit dieses Prinzips wurde bereits von den oben genannten Systemen demonstriert.

CHDL bietet durch die konsequente, objektorientierte Struktur wesentlich mehr Unterstützung für alle Phasen des Designerstellungsprozesses als andere Systeme.

Zunächst erfolgt eine Diskussion über die Wahl der optimalen universellen Programmiersprache für die Beschreibung von Hardware. Es wird ausführlich dargelegt, warum für *CHDL* die Sprache C++ gewählt wurde. Danach wird die Notwendigkeit mehrerer Abstraktionsebenen begründet sowie die mit C++ realisierbaren Ausführungsmodelle erörtert.

Schließlich wird dargestellt, wie die Designentwicklung mit *CHDL* konkret realisiert wurde.

Im Rahmen der Entwurfseingabe wird eine Gegenüberstellung von *CHDL* mit anderen existierenden Systemen, die ebenfalls C++ bzw. *JAVA* zur Hardwarebeschreibung einsetzen, vorgenommen. Es werden die grundlegenden Unterschiede sowie Vor- und Nachteile von *CHDL* und diesen Systemen diskutiert.

Die Beschreibung der Simulation behandelt den Aufbau des *CHDL*-Simulatorkerns, das Erstellen von Simulationsmodellen sowie die spezielle Unterstützung für FPGA-Koprozessoren.

Beim Hardware-Debugging werden Verfahren vorgestellt, die erst durch die Unterstützung von *CHDL* ihre volle Mächtigkeit entfalten können: Readback und partielle Rekonfiguration.

Kapitel 9
Entwurfseingabe

9.1 Die Wahl der optimalen universellen Programmiersprache für die Beschreibung von Hardware

In den letzten Jahren entstanden sowohl Entwicklungssysteme, die C++ verwenden, als auch solche, die *JAVA* zur Hardwarebeschreibung einsetzen. Es gab zahlreiche Diskussionen darüber, welche Sprache besser geeignet sei.

Ein Projekt [23] vertrat die Ansicht, daß nur Programmiersprachen wie *JAVA*, Smalltalk oder CommonLisp die Anforderungen an ein optimales Entwicklungssystem erfüllten, weil die anderen, wie etwa C++, über kein Metadaten-Interface verfügen. Dieses sei aber notwendig, um zur Laufzeit die Struktur der existierenden Objekte erkennen zu können. Es würde drastische Änderungen am Kompiler erfordern, um diese Laufzeitanalysen zu ermöglichen, oder aber erheblichen zusätzlichen Programmieraufwand für die Datenstrukturen, die eine entsprechende Designanalyse durchführen.

In der Tat fehlt der Sprache C++ ein solches Metadaten-Interface. Es sind jedoch zahlreiche Möglichkeiten vorhanden, mit denen die dadurch fehlenden Funktionen dennoch implementiert werden können.

Durch das Fehlen eines Metadaten-Interfaces ergeben sich speziell für die Hardwarebeschreibung folgende Hauptprobleme:

- Im Programm müssen zur Laufzeit die Namen der Objektinstanzen bekannt sein. Wenn der Entwickler ein Objekt "A1" der Klasse "A" erzeugt, muß dieses Objekt wissen, daß sein Name "A1" ist. Dies ist sowohl für die Simulation als auch für die Synthese der Netzliste notwendig. Auch in Projekten, die auf die C++-Hardwarebeschreibung aufbauen, z.B. Hochsprachenkompilern, sind die Namen der zu verschaltenden Bauteile und Pins oft erst zur Laufzeit bekannt. In den existierenden C++ Kompilern ist lediglich die Funktionalität enthalten, zur Laufzeit den Namen der Klasse zu ermitteln (*Runtime-Type-Information*, RTTI). Der Entwickler muß also den Objektnamen als zusätzlichen Parameter übergeben. Dieses Problem könnte aber durch Präprozessorfunktionen beseitigt werden.

- Um die kompletten hierarchischen Namen zur Laufzeit zu ermitteln, müssen die Punkte bekannt sein, an denen Hierarchieebenen betreten und verlassen werden. Dies kann mit einem speziellen lokalen Objekt erreicht werden, das zu Beginn einer neuen Hierarchieebene erzeugt und beim Verlassen der Ebene vom Kompiler automatisch gelöscht wird. Das ist unproblematisch, wenn das Ende der Hierarchieebene mit dem Ende eines lokalen Geltungsbereiches identisch ist. Das entsprechende Objekt kann dann die jeweils notwendigen Aktionen in seinem Konstruktor und Destruktor durchführen.

- Wenn ein Objekt in einem anderen enthalten ist, müssen beide über ihr Verhältnis zueinander informiert sein. Dies ist notwendig, um etwa zu ermitteln, zu welchem Bauteil ein Pin gehört. Von den existierenden C++-Kompilern wird dies nicht unterstützt. Es stehen jedoch Programmiertechniken zur Verfügung, um diese Informationen mit vernünftigem Aufwand und auf verdeckte Weise zu erhalten.

Die Verwendung von C++ zur Hardwarebeschreibung hat einige interessante Vorteile gegenüber *JAVA*:

- Bei der Arbeit mit Koprozessoren, Mikrokontrollern und Embedded Systems ist die eingesetzte Programmiersprache in der Regel C/C++, nicht *JAVA*. *JAVA* zur Hardwarebeschreibung würde somit wieder zwei verschiedene Sprachen erfordern. Insbesondere für das Modellieren auf der Systemebene hat sich inzwischen C++ durchgesetzt.

Wird C++ zur Hardwarebeschreibung eingesetzt, können zudem die bereits vorhandenen und dem Entwickler vertrauten C++-Entwicklungsumgebungen und Source-Level-Debugger verwendet werden.

- *JAVA* unterstützt keine überladenen Operatoren. Dies kann jedoch eine sehr mächtige Fähigkeit zur Hardwarebeschreibung darstellen. Insbesondere zur kompakten Darstellung von logischen Funktionen wie etwa

  ```
  FF = (A & B) | C;
  ```

  ist die Überladung von Operatoren zwingend notwendig.

- C++ unterstützt effiziente DLL-Schnittstellen zur dynamischen Einbindung von Bibliotheken in laufende Prozesse. Dies kann sehr hilfreich zur Implementierung neuer, flexibler Simulationsmethoden sein.

- *JAVA* wurde ursprünglich für plattformunabhängige Programmierung vor allem von Netzwerkanwendungen konzipiert, nicht für hardwarenahe Applikationen. C und C++ bieten dem Anwender deutlich mehr Optimierungsmöglichkeiten inbezug auf Ausführungsgeschwindigkeit.

Jedoch sind auch einige Nachteile von C++ gegenüber *JAVA* zu berücksichtigen:

- C++ benötigt durch das fehlende Metadaten-Interface umfangreiche Maßnahmen, um etwa die bei *JAVA* vorhandenen Funktionen wie

  ```
  .getClass()
  .getDeclaredFields()
  .getDeclaredMethods()
  .getDeclaredConstructors()
  ```

  ersetzen zu können.

- C++ ist nicht wie *JAVA* von Natur aus plattformunabhängig.

Ziel beim Entwurf des *CHDL*-Systems war in erster Linie, umfangreiche und komplexe Anwendungen für FPGA-Koprozessoren bestmöglich zu unterstützen. Diese Unterstützung sollte sowohl die Entwurfseingabe als auch die Simulation und den Echtzeitbetrieb umfassen.

Bei der Entwurfseingabe waren zum einen die Vorteile, die das Überladen von Operatoren zur kompakten Designbeschreibung bietet, mitentscheidend.

Zum anderen ist die übliche Programmiersprache bei der Anwendung von FPGA-Koprozessoren C/C++, nicht *JAVA*. Dies kann damit erklärt werden, daß C/C++ gerade durch seine Plattformabhängigkeit effizientere Schnittstellen zur Hardware ermöglicht. Auch die in vielen Fällen zu implementierenden Gerätetreiber sind nur in C bzw. C++ realisierbar.

Die sich damit bietende Gelegenheit, die Hardwarebeschreibung in derselben Programmiersprache vornehmen zu können wie die Applikationsprogramme des Koprozessors, stellt sich als äußerst interessant dar. Damit würde sich auch ein großer einmaliger Mehraufwand bei der Implementierung des Entwicklungssystems rechtfertigen lassen, der durch das fehlende Metadaten-Interface entsteht.

In der Simulation muß gerade bei umfangreichen Koprozessoranwendungen ein Schwerpunkt auf die Ausführungsgeschwindigkeit gesetzt werden. Insbesondere die Simulation von Gesamtsystemen kann sehr rechenintensiv sein. Die Ausführungsgeschwindigkeit von *JAVA*-Programmen konnte in den letzten Jahren durch Einführung von Native-Code-Kompilern verbessert werden. Dies sind Kompiler, die keinen *JAVA*-Bytecode, sondern architekturspezifischen Maschinencode generieren. Dieser ist nicht mehr plattformunabhängig, jedoch deutlich schneller ausführbar als der zu interpretierende Bytecode. Dennoch kann damit nicht

die Geschwindigkeit der inzwischen technisch ausgereiften C++-Kompiler erreicht werden. Diese bieten zudem noch die Möglichkeit, extrem zeitkritische Teile des Codes in Form von Assemblerprogrammen zu integrieren.

Die Funktionalität, die durch das fehlende Metadaten-Interface zwingend implementiert werden muß, stellte sich bei einem ersten Prototyp als einmaliger und vertretbarer Aufwand dar.

Es ist gelungen, die notwendigen Maßnahmen zur Lösung der oben aufgezählten Hauptprobleme zum größten Teil in den Basisklassen zu verbergen. Der Anwender von *CHDL* hat nur wenige Punkte bei der Programmierung zu beachten, so daß er mit den Nachteilen, die C++ gegenüber *JAVA* bei der Hardwarebeschreibung besitzt, kaum in Kontakt kommt.

Die Vorteile einer Entscheidung zugunsten von C++ überwiegen deutlich gegenüber dem Mehraufwand bei der Implementierung. Daher fiel die Wahl auf C++.

9.2 Die Notwendigkeit mehrerer Abstraktionsebenen

Von den existierenden Entwicklungssystemen sind als verschiedene zur Verfügung stehende Abstraktionsebenen zunächst die strukturelle sowie die verhaltensorientierte Hardwarebeschreibung bekannt.

Durch die immer umfangreicher und komplexer werdenden Anwendungen gestaltet sich die Beherrschbarkeit strukturell orientierter Hardwarebeschreibungen zunehmend schwieriger.

Die verhaltensorientierten Beschreibungen stellen eine interessante Alternative zu den strukturellen dar. Sie können den Entwickler von Implementierungsdetails entlasten.

Verhaltensorientierte Sprachen setzen allgemeine Modelle ein, etwa die der Zustandsmaschinen, die der Beschreibung implizit zugrundegelegt werden. Dadurch muß der Entwickler nicht mehr die gesamte Funktionalität beschreiben, sondern nur noch das allgemeine zugrundeliegende Modell für seine Anwendung konkretisieren.

So kann er etwa sequentielle Anweisungen formulieren, ohne dabei die nötigen Speicherelemente für den jeweils aktuellen Zustand verschalten zu müssen. Diese Aufgabe wird nach einem festgelegten Ausführungsmodell vom verhaltensorientierten Kompiler automatisch übernommen.

Damit stellt sich die Frage, inwieweit die strukturelle Form der Beschreibung überhaupt noch notwendig ist.

In FPGA-Anwendungen, bei denen der Resourcenverbrauch und die Taktfrequenz kritische Faktoren darstellen, kann jedoch eine strukturelle Beschreibung der einzige Weg sein, die strengen Anforderungen zu erfüllen.

Der Nachteil einer verhaltensorientierten Beschreibung liegt darin, daß keine präzise Kontrolle über die zu implementierende Schaltung besteht. Das zugrundeliegende Modell kann zwar konkretisiert, aber nicht mehr im Detail verändert werden.

Dies gilt insbesondere, wenn der zu implementierende Algorithmus an sich einen strukturellen Charakter aufweist. Das ist oft bei datenflußorientierten Anwendungen der Fall. Hier kann die Erstellung einer strukturellen Beschreibung effizienter sein als eine verhaltensorientierte.

Der Entwickler hat in solchen Fällen bereits eine eher strukturelle Vorstellung der Implementierung. Er muß nun diese Vorstellung in das Ausführungsmodell und in die Semantik der verhaltensorientierten Sprache umsetzen. Dies kann dazu führen, daß er Formulierungen verwenden muß, die deutlich von seiner inneren Implementierungsvorstellung abweichen, eventuell sogar umständlicher sind.

Der Optimierungsalgorithmus des Entwicklungssystems steht nun vor einer praktisch unlösbaren Aufgabe: Er soll die wahre Intention des Anwenders ermitteln und die verhaltensorientierte Beschreibung effizient in die zur Verfügung stehenden strukturellen Grundelemente umsetzen.

Die Folge ist, daß die resultierende Implementierung in der Regel weniger effizient ist, als bei einer strukturellen Vorgabe.

Durch strukturelle Designbeschreibung und der Möglichkeit, Vorplazierungsinformationen zu übergeben, wird der Entwickler immer bessere Ergebnisse erzielen können als mit einer verhaltensorientierten.

Allerdings bestehen auch komplexe Designs selten ausschließlich aus zeitkritischen Komponenten. Nicht für alle Teile ist die präzise Kontrolle über die Resourcen mittels struktureller Beschreibung notwendig.

In vielen Anwendungsfällen wird der Nachteil einer weniger effizienten Implementierung gegenüber dem Vorteil der einfacheren Beschreibbarkeit in den Hintergrund treten.

Insbesondere bei sequentiellen Algorithmen kann es sehr aufwendig sein, sie rein strukturell zu beschreiben. Nicht nur die Erstellung ist dabei schwieriger, sondern auch die Wartung des erstellten Codes.

Aus den genannten Gründen bietet es sich an, die verschiedenen Beschreibungsebenen miteinander zu kombinieren. Die strukturelle und die verhaltensorientierte Ebene sollten nicht zueinander in Konkurrenz stehen, sondern sich ergänzen, um FPGAs mit allen ihren mächtigen Fähigkeiten optimal einsetzen zu können.

Sowohl die strukturelle Beschreibung als auch alle abstrakteren Formen setzen zu ihrem Verständnis die Kenntnis der zugrundliegenden Modelle voraus. Bei der Strukturbeschreibung etwa besteht dieses Modell aus der Annahme, daß alle definierten Bauteile und Verbindungen parallel nebeneinander existieren. Daraus ergibt sich unter anderem, daß die Reihenfolge der einzelnen Definitionen keine Bedeutung besitzt.

Universelle Programmiersprachen benutzen ebenfalls zugrundeliegende Modelle, die sich naturgemäß an der sequentiellen Ausführbarkeit auf Mikroprozessoren orientieren.

Soll nun zur Hardwarebeschreibung eine universelle Programmiersprache eingesetzt werden, ist zunächst zu prüfen, inwieweit die hier eingesetzten Modelle zu denen der Hardwarebeschreibung kompatibel sind.

In der folgenden Darstellung wird zunächst das Ausführungsmodell der Programmiersprache C++ näher erläutert. Danach erfolgt eine Analyse der Modelle, die für die Hardwarebeschreibung notwendig sind und Lösungsvorschläge für die Realisierung mit C++.

9.3 Realisierbare Ausführungsmodelle

9.3.1 Das Ausführungsmodell von C++

Das Prinzip universeller Programmiersprachen, wie C++, beruht auf einer sequentiellen Ablaufreihenfolge. Dies ist dadurch bedingt, daß Mikroprozessoren nur über eine geringe Anzahl von Recheneinheiten verfügen und daher den größten Teil der Anweisungen nacheinander ausführen müssen. Auch der Datentransfer mit der Umgebung ist in der Regel nur über eine einzige Schnittstelle möglich.

Dem folgenden Programmcode liegt daher die Annahme zugrunde, daß die einzelnen Anweisungen, auch soweit sie unabhängig voneinander sind, immer in der angegebenen Reihenfolge abgearbeitet werden:

```
void  Function1 ( void )
{
  A = 1;
  B = 2;
  B = B + 1;
  C = A;
}
```

Moderne Mikroprozessoren mit mehreren Recheneinheiten verfügen über die Möglichkeit, diese Reihenfolge begrenzt abzuändern, um unabhängige Anweisungen parallel auszuführen. Dies hat jedoch für die sequentielle Beschreibung keine Konsequenzen. Jede Optimierung, die die Kompiler und die Scheduler in der Prozessoren vornehmen, orientiert sich stets an der vorgegebenen sequentiellen Beschreibung.

9.3.2 Strukturelle Hardwarebeschreibung

Eine strukturelle Hardwarebeschreibung beruht auf einer Vielzahl parallel nebeneinander existierenden Elementen und deren Verbindungen untereinander.

Soll zur Beschreibung eine konventionelle Programmiersprache eingesetzt werden, läßt sich dies realisieren, indem Bauteile durch Variablen und Verbindungen durch Zuweisungen repräsentiert werden. Betrachtet man jedoch diese Zuweisungen näher, ergeben sich durch die Annahme der Parallelität folgende Unterschiede zum obigen Code-Beispiel:

- Die Anweisungen `B = 2` und `B = B + 1` dürfen nicht mehr beide angegeben werden, da sie im Widerspruch zueinander stehen. Vorher wurden sie zu unterschiedlichen Zeitpunkten ausgeführt, jetzt sind sie parallel. Es gilt die *single-assignment*-Regel, nach der jeder Variablen nur einmal ein Wert zugewiesen werden darf. Bedingte Anweisungen sind davon nicht betroffen, soweit sich die einzelnen Pfade gegenseitig ausschließen.

- Da die Ausführung nicht mehr sequentiell sondern parallel erfolgt, muß nun auf andere Weise spezifiziert werden, zu welchen Zeitpunkten die Operationen durchgeführt werden sollen. Dies wird besonders für die Anweisung `B = B + 1` deutlich. Während die anderen Anweisungen durch statische Verbindungen implementiert werden können, ist `B = B + 1` nur mit einem flankengesteuerten Register realisierbar. Jede andere Implementierung würde zu einer kombinatorischen Schleife und damit zu einem nicht beabsichtigten Verhalten führen.

- Um die Flexibilität der Beschreibung zu erhöhen, können die Arbeitstakte für jede einzelne Variable getrennt festgelegt werden. Außerdem können spezielle Signale wie etwa ein Clock-Enable-Signal existieren. Damit wird erreicht, daß zwar alle Anweisungen parallel ausgeführt werden können, aber nicht zwangsläufig ausgeführt werden müssen.

Eine solche strukturelle Beschreibung kann durch Implementierung spezieller C++-Klassen und überladener Operatoren mit einem handelsüblichen C++-Kompiler übersetzt werden, da sie keine Modifikationen an der Sprache C++ selbst erfordert. Während der Ausführung des kompilierten Programmes können durch geeignete objektorientierte Maßnahmen die einzelnen Anweisungen intern registriert, überprüft und später zu einer Netzliste synthetisiert werden.

Die hierzu bei *CHDL* verwendeten Verfahren werden später vorgestellt.

9.3.3 Verhaltensbeschreibung von Zustandsmaschinen

Bei der Beschreibung von Zustandsmaschinen muß über die Spezifikation der Variablen und der Verbindungen hinaus auch der sequentielle Ablauf angegeben werden. Es ist zwar denkbar, Zustandsmaschinen auf strukturelle Weise zu beschreiben, eine verhaltensorientierte Formulierung, wie sie *VHDL* oder *SystemC* bereitstellen, wird jedoch deutlich leichter verständlich sein.

Im Gegensatz zur strukturellen Beschreibung werden bei der verhaltensorientierten nicht global alle Operationen parallel ausgeführt, sondern jeweils nur diejenigen, die im momentanen Zustand aktiv sind. Es existiert zusätzlich zu den Anweisungen ein Kontrollpfad, der die Zustandsübergänge festlegt. Weiterhin kann dieser Kontrollpfad abhängig vom Zustand

interner oder externer Signale sein, was sich durch bedingte Anweisungen ausdrücken läßt:

```
while (1)
 {
   wait();
   if (A == 1)
     {
       B = 0;
       wait();
       B = 2;
     }
    else
     {
       B = 1;
     }
 }
```

Bei diesem Ausführungsmodell ist folgendes zu beachten:

- Es werden nur die Anweisungen im jeweils aktuellen Zustand ausgeführt. Innerhalb dieses Zustandes erfolgt die Ausführung parallel.
- Die *single-assignment*-Regel gilt nur für Anweisungen im selben Zustand, da die einzelnen Zustände sich gegenseitig ausschließen.
- Die Zustände müssen eindeutig gegeneinander abgegrenzt sein, um festzulegen, welche Anweisungen parallel ausgeführt werden sollen und welche nicht. Im Beispiel erfolgt dies mit der `wait()`-Anweisung.
- Die Zustandsübergänge können bedingte Anweisungen enthalten. Ist kein expliziter Übergang spezifiziert, wird implizit ein Übergang zum direkt nachfolgenden Zustand angenommen.

Bei der Realisierung dieses Modells mit C++ stellen insbesondere die bedingten Anweisungen `if`, `switch`, `for`, `do` und `while` ein Problem dar.

Diese können sowohl in strukturellen Beschreibungen als auch in verhaltensorientierten sinnvoll eingesetzt werden. Dies wird am folgenden Beispiel des `if` demonstriert.

Auf struktureller Ebene ist es mit `if` möglich, Ausdrücke zu formulieren, die ansonsten einen expliziten Multiplexer benötigen würden (Abb. 9.1).

```
if (A == 1)
   B = 0;
 else
   B = 1;
```

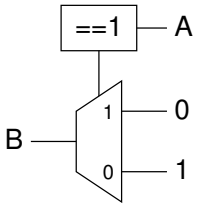


Abbildung 9.1: `if` realisiert einen Multiplexer

In einer verhaltensorientierten Beschreibung wird mit `if` festgelegt, ob eine Anweisung mit dem nächsten Takt ausgeführt wird oder nicht.

In strukturellen Beschreibungen sind bedingte Anweisungen jedoch nicht zwingend erforderlich. Ihre Funktionalität kann auch auf andere Weise erreicht werden, z.B. durch Multiplexer oder Clock-Enable-Signale (Abb. 9.2).

```
if (A == 1)
   B = 0;
```

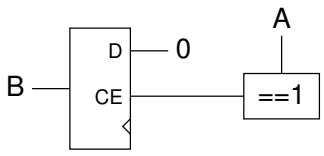


Abbildung 9.2: `if` ersetzt Clock-Enable

In verhaltensorientierten Beschreibungen sind sie jedoch grundsätzlich notwendig, um mehrere Ablaufpfade einer Prozedur festzulegen. Bei einem Aufruf der Prozedur wird aber immer nur eine einzige dieser verschiedenen Möglichkeiten durchlaufen. Die Ablaufpfade werden dann in bedingte Zustandsübergänge umgesetzt.

Das folgende Beispiel stellt diese Problemsituation konkreter dar:

```
void  Process_Mux ( void )
{
  if (sel == 0)
     output = input0;
   else
     output = input1;
}
```

Unabhängig davon, wie im Entwicklungssystem die Informationen über die zu verschaltenden Netze erzeugt werden, ist es unmöglich, aus dieser Beschreibung in nur einem Durchlauf der Prozedur die komplette Schaltung zu synthetisieren. Denn abhängig von der Entscheidung `sel == 0` wird immer nur einer der beiden möglichen Ablaufpfade durchlaufen. Der jeweils andere kann bei der Synthese nicht berücksichtigt werden.

Lösbar wäre dieses Problem auf zwei Wegen:

- Mehrfaches Durchlaufen der Prozedur für alle relevanten Kombinationen der Eingangsvariablen, um alle Pfade abdecken zu können. Mit den existierenden C++-Kompilern ist es nicht möglich, zur Laufzeit die Existenz bedingter Anweisungen zu erkennen, ohne diese selbst zu durchlaufen. Es bleibt nur die Brute-Force Methode, alle möglichen Kombinationen der Eingangsvariablen zu durchlaufen. Dies hätte wegen der Vielzahl der zu testenden Kombinationen in der Praxis lange Laufzeiten zur Folge.

- Implementierung eines eigenen C++-Parsers, um eine Analyse der Ablaufpfade zu ermöglichen.

Ziel des *CHDL*-Projektes war, auf der untersten Ebene der Hardwarebeschreibung die prinzipiellen Probleme einer Programmiersprache zur Netzlistenerzeugung zu bewältigen, dabei aber auf effiziente Ausführung zu achten und den Einsatz handelsüblicher C++-Kompiler zu ermöglichen. Diese unterste Ebene sollte als Basis für weitere, abstraktere Ebenen dienen und daher bewußt einfach und effizient gehalten werden.

Aus diesen Gründen und den oben dargestellten Problemen werden die bedingten C++-Anweisungen nicht in Form einer verhaltensorientierten Beschreibung genutzt.

Bedingte Anweisungen können jedoch vom Entwickler eingesetzt werden, um den Ablauf der Hardwarebeschreibung selbst zu beeinflussen. In der Praxis hat sich dies in Verbindung mit Parametrisierung als sehr mächtiges Instrument herausgestellt.

Die C++-Anweisungen `if`, `for`, `while`, `do` und `switch` sind in einer strukturellen Hardwarebeschreibung mit Makro- oder Präprozessoranweisungen vergleichbar. Mit ihnen können zur Laufzeit Teile der eigentlichen Hardware ausgelassen, selektiert oder wiederholt werden.

Das folgende Beispiel

```
for (i = 0; i < 6; i++)
 {
   if (i & 0x01)
       A[i] = B[i/2];
     else
       A[i] = C[i/2]
 }
```

ist äquivalent zu:

```
A[0] = C[0];
A[1] = B[0];
A[2] = C[1];
A[3] = B[1];
A[4] = C[2];
A[5] = B[2];
```

Die resultierende Hardwarebeschreibung ist auf diese Weise eine effiziente Kombination aus echten Hardwarebeschreibungsanweisungen und zusätzlichen Makroanweisungen.

9.4 Strukturelle Hardwarebeschreibung mit C++

9.4.1 Überblick

Die Programmiersprache C++ besitzt im Gegensatz zu einigen anderen Sprachen, wie etwa *JAVA*, kein Metadaten-Interface. Sowohl dies als auch die oben erörterte Problematik der Ausführungsmodelle machen es notwendig, zunächst eine Reihe von Verwaltungsfunktionen zu implementieren. Erst damit kann C++ optimal zur Hardwarebeschreibung eingesetzt werden.

Die nötigen Funktionen umfassen:

- Eine Objektverwaltung.

  Diese verwaltet Listen aller bestehenden Objekte. Sie realisiert weiterhin eine Objekthierarchie, in der alle beteiligten Objekte ihre Beziehung zueinander kennen.

- Eine Verwaltung für die Namenshierarchie.

  Zur Verwaltung verschachtelter Module müssen die Namen der einzelnen Ebenen zu einem eindeutigen Gesamtnamen kombiniert werden können.

- Eine Verlängerung der Objektlebensdauer.

  Lokale Objekte, die als automatische Variablen auf dem Stack angelegt sind, werden am Ende ihres Geltungsbereiches vom Kompiler automatisch gelöscht. Da sie als Bauteile aber erhalten bleiben sollen, muß ein Mechanismus existieren, der diese Situation erkennt und das Objekt erhält.

- Eine eindeutige und nachvollziehbare Benennung der Objekte zur Laufzeit.

  Ein C++-Objekt kann zur Laufzeit nicht automatisch den Namen ermitteln, der ihm in der Programmdatei bei seiner Instanziierung zugewiesen wurde. Zur späteren Erzeugung von Netzlisten ist jedoch ein eindeutiger Name notwendig.

- Eine Netzlistenverwaltung.

  Die Netzlistenverwaltung beinhaltet alle bestehenden Verbindungen zwischen den Pins von Bauteilen. Sie ist in der Lage, eine Konsistenzprüfung vorzunehmen und unzulässige Situationen, etwa mehrere treibende Ausgänge im selben Netz, zu erkennen.

- Einen Mechanismus, um logische Ausdrücke formulieren zu können.

  Die Hardwarebeschreibung sollte nicht nur einfache Zuweisungen zum Erstellen von Verbindungen unterstützen, sondern auch logische Ausdrücke. Dies sollte sowohl mit einzelnen Pins als auch mit Busstrukturen möglich sein.

- Maßnahmen zur Vereinfachung der Schreibweise.

  Durch geeigneten Einsatz von C++-Techniken läßt sich eine kompakte Schreibweise erreichen.

- Möglichst frühzeitige und präzise Fehlermeldungen.

  Fehler in der Hardwarebeschreibung sollten möglichst früh erkannt und präzise gemeldet werden, um dem Entwickler optimal zu unterstützen.

9.4.2 Die Objektverwaltung

Es muß ein Mechanismus implementiert werden, der in der Lage ist, Listen aller bestehenden Objekte selbständig zu verwalten. Weiterhin ist eine Objekthierarchie erforderlich, in der alle beteiligten Objekte ihre Beziehung untereinander ermitteln können. So muß etwa ein Bauteilobjekt in der Lage sein, alle in ihm enthaltenen Pins zu ermitteln und jedes Pinobjekt muß einen Verweis auf sein zugehöriges Bauteilelement besitzen. Die Objektverwaltung stellt außerdem sicher, daß nicht mehr benötigte Elemente automatisch gelöscht werden.

Um die Hardwarebeschreibung von diesen Aufgaben zu enlasten, sollte die Verwaltung soweit wie möglich im Verborgenen ablaufen.

Es bietet sich an, diese Funktionalität innerhalb der Konstruktoren und Destruktoren der jeweiligen Basisklassen zu implementieren. Später abgeleitete Klassen sind dann von dieser Aufgabe entlastet. C++-Bibliotheken zur Erstellung grafischer Benutzeroberflächen, wie etwa *QT* [100] von *Trolltech*, wenden ebenfalls solche Verfahren an.

9.4.3 Die Hierarchieverwaltung

Zum Erstellen komplexer modularer Beschreibungen werden bereits definierte Bauteile innerhalb der Definition neuer Bauteile verwendet. Die Namen der einzelnen Hierarchieebenen müssen zu einem eindeutigen Gesamtnamen zusammengesetzt werden, der später in der Netzliste verwendet wird.

Um während der Laufzeit zu erkennen, auf welcher Hierarchieebene eine gerade ausgeführte Anweisung einzuordnen ist, muß die aktuelle Ebene jeweils am Ende des Konstruktors verlassen werden. Dazu ist es notwendig, daß an diesem Ende möglichst automatisch eine interne Verwaltungsfunktion aufgerufen wird. Diese Situation wird von den C++-Kompilern nicht unterstützt.

Es kann jedoch ein explizites Hilfsobjekt verwendet werden, um das gewünschte Verhalten zu erreichen:

```
MyPart::MyPart ( const char* Name )
  : BasePart(Name)
{
  ProcessName _N(this);

  ...
}
```

Die Erzeugung des Hilfsobjektes muß die erste Anweisung innerhalb des Konstruktors sein. Der Aufruf des Destruktors dieses Hilfsobjektes erfolgt automatisch am Ende des Konstruktors, womit im Verborgenen die nötigen Schritte zum Verlassen der aktuellen Hierarchieebene durchgeführt werden können.

9.4.4 Die Lebensdauer von Objekten

Objekte, die als automatische Variable angelegt werden, besitzen nur eine begrenzte Lebensdauer. Am Ende ihres Geltungsbereiches, also spätestens am Ende der Funktion wird automatisch ihr Destruktor aufgerufen und der zugeordnete Speicherplatz auf dem Stack ungültig.

Werden Objekte zur Repräsentation von Bauteilen eingesetzt, sollte ihre Lebensdauer jedoch über die Funktion hinaus bestehen.

Dies wäre dann zu erreichen, wenn alle Objekte dynamisch mittels `new` angelegt werden. Solche Objekte sind in der Schreibweise jedoch deutlich unhandlicher als die normale Variante. Außerdem sind die später beschriebenen Verfahren zur Reduzierung redundanter Pinangaben in diesem Fall nicht möglich.

```
DFF*  FF1 = new DFF;
DFF*  FF2 = new DFF;

FF1->D = FF1->Q & FF2->Q;
```

Im Vergleich zu:

```
DFF FF1;
DFF FF2;

FF1 = FF1 & FF2;
```

Die Methode, Bauelemente dynamisch zu erzeugen, ist jedoch in den Fällen zwingend erforderlich, in denen erst zur Laufzeit der Typ oder die Anzahl der zu erzeugenden Bauteile bekannt wird. Hier ist die statische Methode nicht anwendbar.

Ein Verfahren, um die Lebensdauer eines Objektes über seinen begrenzten Geltungsbereich hinaus zu verlängern, besteht darin, das Objekt innerhalb des Destruktors zu kopieren und dabei dynamisch zu erzeugen. Damit erhält es eine beliebig lange Lebensdauer:

```
DFF::~DFF()
{
  if (!CtrlFlag)
   {
     new DFF(this);
     CtrlFlag = 1;
   }
}
```

Die Variable `CtrlFlag` ist dabei notwendig, um unter bestimmten Bedingungen, z.B. am Ende des *CHDL*-Programmes, trotz dieses speziellen Destruktors doch alle Objekte löschen zu können.

9.4.5 Die eindeutige und nachvollziehbare Benennung der Objekte zur Laufzeit

Sowohl für die Simulation als auch für die spätere Erzeugung der Netzlisten ist es notwendig, Objekte mit einem eindeutigen Namen zu versehen. Es bietet sich an, den Namen zu verwenden, den das Objekt auch im Programmcode besitzt. Die verfügbaren C++-Kompiler sehen jedoch keinen Mechanismus vor, mit dem dieser Name aus der Compile-Zeit in die Laufzeit übertragen werden kann. Lediglich der Klassenname kann über die Runtime-Type-Information (RTTI) erhalten werden, nicht aber der Name von einzelnen Objekten. Existierende C++-Klassenbibliotheken, die eine derartige Funktionalität benötigen, etwa *QT*, setzen hierzu einen proprietären Meta-Kompiler ein, der ähnlich einem Präprozessor vor dem C++-Kompiler abläuft.

Bei *CHDL* wurde bewußt auf einen solchen Meta-Kompiler verzichtet, um die unterste Ebene der Hardwarebeschreibung völlig kompatibel zu konventioneller C++-Entwicklung zu halten.

Es kann auch wünschenswert sein, die Namen von Objekten abweichend vom Namen im Programmcode zu wählen, etwa wenn Objekte innerhalb einer Schleife mit einem Index im Namen erzeugt werden. Auch ein Meta-Kompiler könnte diese erst zur Laufzeit entstehenden Namen nicht ermitteln.

Die Namensangabe von *CHDL*-Objekten erfolgt daher explizit im Konstruktor. Die dabei entstehende Redundanz ist der Preis, der für die völlige C++-Kompatibilität gezahlt werden muß:

```
DFF  FF1("FF1");
DFF  FF2("FF2");
```

Die Angabe von Objektnamen ist für die Hardwarebeschreibung nicht zwingend erforderlich. Wird kein Name angegeben, erhält das Objekt einen automatischen Namen in der Form `PARTxxxxx`, wobei die letzten Ziffern eine intern weitergezählte Nummer darstellen. Auf solche nicht explizit benannten Objekte kann später in der Simulation nicht zugegriffen werden. Auch stehen sie für das Readback-Verfahren nicht zur Verfügung.

9.4.6 Die Verwaltung von Netzen

Die Netzlistenverwaltung verfolgt die Verbindungen der Pins untereinander. Die Erzeugung von solchen Verbindungen erfolgt mit dem Zuweisungsoperator:

```
FF1.D = FF1.Q;
```

Um dies zu realisieren, wird der Zuweisungsoperator der Klasse `BasePin` mit entsprechender Funktionalität überladen.

Entsprechend den Konventionen der üblichen Hardwarebeschreibungs- und Programmiersprachen befindet sich das Ziel der Zuweisung auf der linken und die Quelle auf der rechten Seite des Gleichheitszeichens.

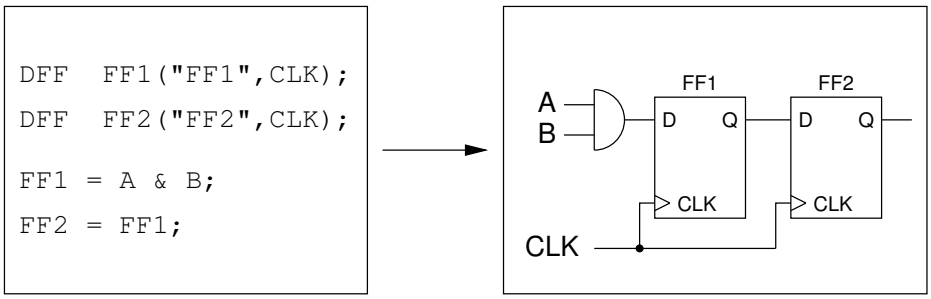


Abbildung 9.3: Strukturelle Beschreibung

9.4.7 Die Auswertung von Schaltfunktionen

Um eine kompakte Hardwarebeschreibung zu realisieren, müssen Schaltfunktionen direkt formuliert werden können. Dies ist durch Überladen von Operatoren realisierbar.

Es können beliebig komplexe Ausdrücke dargestellt werden:

```
FF1 = A & B | (C & D);
FF2 = A & (B == 0);
```

Die korrekte Behandlung der Operatorprioritäten und Klammerungen wird dabei bereits durch den C++-Kompiler sichergestellt. Die einzelnen Operatoren werden in der richtigen Reihenfolge abgearbeitet.

Die sofortige Erzeugung von Gatterobjekten wäre die am einfachsten zu implementierende Methode. Gerade bei FPGAs bietet es sich jedoch an, bereits in dieser Phase ein Mapping auf die Lookup-Tabellen vorzunehmen. Dies ermöglicht eine genauere Vorhersage des Resourcenverbrauchs.

Die Information über den anzuwendenden Operator und die beteiligten Pins werden daher nicht sofort umgesetzt, sondern in einem Hilfsobjekt aufgezeichnet. Erst bei der endgültigen Verwendung seines Ausgangspins wird dieses Hilfsobjekt nach einer Logikoptimierung implementiert.

Um dieses Verfahren zu realisieren, müssen die Hilfsobjekte in die Operatorüberladungen aufgenommen werden. Die Ergebnisse von Operatoren sind somit nicht Pins, sondern Hilfsobjekte. Auch diese Hilfsobjekte können wieder mit Operatoren verknüpft werden.

Das Beispiel

```
FF1.D = FF1.Q & !FF2.Q;
```

führt zu folgender Ausführungsreihenfolge:

1. Operator "!" mit FF2.Q als Argument. Ergebnis ist ein Hilfsobjekt.
2. Operator "&" mit FF1.Q und dem Hilfsobjekt aus 1.
3. Zuweisungsoperator mit FF1.D und dem Hilfsobjekt aus 2.

Es läßt sich folgender Auswertungsbaum aufbauen:

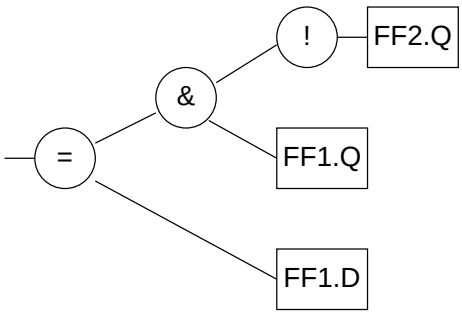


Abbildung 9.4: Auswertungsbaum

Der Entwickler wird in der Praxis logische Ausdrücke in einer Form eingeben, die für ihn leicht verständlich ist. Dies wird in der Regel nicht die optimalste und minimalste Form sein. So kann es sein, daß Variablen mehrfach vorkommen, redundante Variablen verwendet werden oder sich ganze Teilausdrücke zu ’0’ oder ’1’ reduzieren lassen.

Das Entwicklungssystem sollte in der Lage sein, solche suboptimal formulierten Ausdrücke zu optimieren. Eine einfach zu implementierende aber wirkungsvolle Optimierung kann z.B. nach dem Quine-McCluskey-Verfahren [49] erfolgen.

Wie oben bereits erwähnt, ist es sinnvoll, auch ein Mapping der Logik auf die Lookup-Tabellen vorzunehmen.

Die Produktterme der aufgezeichneten Logik müssen hierfür so angeordnet und zerlegt werden, daß Teilausdrücke mit maximal 4 Schaltvariablen entstehen. Ein einfaches Verfahren besteht darin, die einzelnen Produktterme so in Gruppen anzuordnen, daß in jeder Gruppe maximal 4 verschiedene Variablen verwendet sind. Terme, die für sich alleine schon mehr als 4 Variablen beinhalten, werden so angeordnet, daß gemeinsame Variablen möglichst wirksam ausgeklammert werden können.

Für die folgende Beispiellogik

```
(!A &  B &  C & !D &  E) |
(!A &  B &  C &  D & !E) |
(!A & !B &           !E) |
( A &       C &  D & !E) |
( A &       C & !D &  E) |
( A & !B & !C &       E) |
(      B & !C &       E)
```

könnte die Aufteilung etwa so aussehen:

```
(!A &  B &  C & !D &  E) |
(!A &  B &  C &  D & !E) |
( A &       C &  D & !E) |
( A &       C & !D &  E) |

(!A & !B &           !E) |
( A & !B & !C &       E) |
(      B & !C &       E)
```

Im der oberen Gruppe kann nun die Variable C ausgeklammert werden:

```
(((!A &  B &  !D &  E) |
  (!A &  B &   D & !E) |
  ( A &        D & !E) |
  ( A &       !D &  E)) & C) |

(!A & !B &      !E) |
( A & !B & !C &  E) |
(      B & !C &  E)
```

Damit kann die Logik gemäß Abbildung 9.5 auf die LUTs verteilt werden.

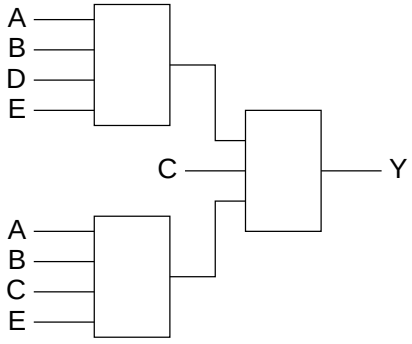


Abbildung 9.5: Partitionierung der Logik auf LUTs

Für dieses rechenintensive Problem der optimalen Verteilung der Logik auf die LUTs wurden zahlreiche Algorithmen vorgeschlagen [77].

Das *CHDL*-System in seiner aktuellen Form enthält nur einen trivialen Algorithmus für die Logikpartitionierung. Die Logikhilfsobjekte wären die geeigneten Stellen, an denen effizientere Algorithmen, z.B. aus [77], integriert werden könnten.

9.4.8 Vereinfachung der Schreibweise

Um die Schreibweise zu vereinfachen, können auch spezielle Operatoren der Bauteilklassen eingesetzt werden:

```
BasePin& DFF::operator = ( const BasePin& Pin )
{
  D = Pin;
}

DFF::operator BasePin& ()
{
  return (Q);
}
```

Auf diese Weise läßt sich folgende Schreibweise realisieren:

```
DFF  FF1;
DFF  FF2;

FF1 = FF1 & !FF2;
```

Damit ist eine Ausdrucksweise erreicht, die mit der von konventionellen strukturellen Hardwarebeschreibungssprachen, wie etwa *ABEL* [117], vergleichbar ist.

9.4.9 Frühzeitige und präzise Erkennung von Fehlern

Fehler, die dem Anwender bei der Eingabe der Hardwarebeschreibung unterlaufen, sollten so detailliert wie möglich gemeldet werden.

Der früheste Zeitpunkt, zu dem ein Eingabefehler erkannt werden kann, liegt beim Kompilieren der Beschreibung. Hier können syntaktische Fehler, z.B. einfache Tippfehler, erkannt werden. Es ist aber auch möglich, komplexere Fehler zu entdecken, indem die Prüfungsmöglichkeiten des Kompilers genutzt werden. So kann etwa die unzulässige Verwendung für in diesem Fall nicht definierte Funktionen oder Operatoren erkannt werden. Auch die Angabe von falschen Datentypen wird gemeldet.

Der Vorteil der Fehlererkennung beim Kompilieren liegt darin, daß der Anwender vom Kompiler die genaue Position erhält, an der der Fehler auftritt.

Das unzulässige Verschalten eines Ausgangspins mit einem anderen Ausgangspin könnte durch eine geeignete Definition der Zuweisungsoperatoren bereits beim Kompilieren erkannt werden: Die Klasse der Ausgangspins erhält keinen Zuweisungsoperator zu anderen Pinklassen. Nimmt der Anwender dennoch eine solche Zuweisung vor, erhält er eine Fehlermeldung des Kompilers an genau dieser Stelle.

Es hat sich jedoch als sinnvoll herausgestellt, einige Fehlersituationen nicht beim Kompilieren zu behandeln, damit die Konstruktion der Klassen und Operatoren nicht zu komplex wird. Der C++-Standard erlaubt den Kompilern die automatische Anwendung von Umwandlungsoperatoren. Dadurch kann es bei nicht vorhandenen direkten Operatoren zu unerwünschten indirekten Operatoranwendungen kommen.

Weiterhin existieren Pins, deren Eigenschaft "Eingang" bzw. "Ausgang" davon abhängig ist, ob sich die betreffende Zuweisung innerhalb oder außerhalb eines Moduls befindet: Der Ausgangspin eines Modules ist zwar außerhalb ein Ausgang, aber innerhalb des Modules ist er ein Eingang, der von einem anderen Signal gespeist werden muß.

Während der Laufzeit besteht eine Vielzahl von Möglichkeiten, Fehler zu erkennen, so etwa mehrfach getriebene Netze, Netze ohne Signalquelle, unbenutzte Eingangspins, unzulässige Busbreiten, unzulässige Modulparameter usw. Es bereitet in manchen Fällen jedoch Schwierigkeiten, dem Anwender einen sinnvollen Hinweis auf die Position des Fehlers in seinem Quellcode zu geben. Da sich die Programmstelle, in der der Fehler festgestellt wird, in der Regel in einem Unterprogramm befindet, kann die eigentliche Fehlerposition nur über einen Debugger und die Aufrufliste festgestellt werden.

9.4.10 Implementierung von architekturunabhängigen Grundelementen

Die verfügbaren FPGAs weisen in ihren Architekturen große Ähnlichkeiten auf. So verwenden alle zur Implementierung von kombinatorischer Logik Funktionsgeneratoren mit vier Eingängen. Die konfigurierbaren Blöcke enthalten Anordnungen aus einem Funktionsgenerator und einem Speicherelement. Es gibt nur geringe Unterschiede.

Bei *CHDL* wird das Ziel verfolgt, die zentralen Grundelemente wie Flip-Flops, Gatter, arithmetische Funktionen sowie die Interfaces zu den Gehäusepins architekturunabhängig zu gestalten.

Elemente, die spezielle Resourcen der jeweiligen Architekturen darstellen, sollten architekturspezifisch bleiben. Sie werden dazu in eigenen Bibliotheken für jede Architektur zusammengefaßt.

Hardwarebeschreibungen, die ausschließlich architekturunabhängige Grundelemente verwenden, können auf diese Weise ohne Modifikation auf jeder FPGA-Architektur simuliert und synthetisiert werden.

Nachfolgend werden die wichtigsten Unterschiede der Architekturen erörtert und dargestellt, wie jeweils architekturunabhängige Elemente konstruiert werden können.

- Vorhandensein expliziter Clock- bzw. Gate-Enable-Eingänge an den Speicherelementen.

  Einige Architekturen besitzen einen Clock-Enable-Eingäng (CE) an den Flip-Flops bzw. einen Gate-Enable-Eingang (GE) an Latches. Andere Architekturen unterstützen diese Funktion nicht explizit.

  Die CE- bzw. GE-Funktionalität ist in ihrer expliziten Version nicht unbedingt notwendig. Sie kann innerhalb des Datenpfades emuliert werden:

  ```
  FF.D  = A;
  FF.CE = B;
  FF.C  = C;
  ```

  Ist gleichwertig mit

  ```
  FF.D  = (B & A) | (!B & FF.Q);
  FF.C  = C;
  ```

  Verfügt die Zielarchitektur über diese Eingänge, können sich bei deren Nutzung deutliche Vorteile durch die vereinfachten Datenpfade ergeben. Daher sollten sie in einem architekturunabhängigen Element vorhanden sein.

  Bei den Architekturen, die die CE- bzw. GE-Eingänge nicht explizit unterstützen, können die Eingänge dennoch in der Hardwarebeschreibung vorteilhaft sein, da sie kompaktere und übersichtlichere Formulierungen ermöglichen. Die nötige Funktionalität kann problemlos, wie oben beschrieben, automatisch in den Datenpfad integriert werden.

  Werden die CE- bzw. GE-Eingänge vom Design nicht belegt, ergeben sich keine Nachteile durch dieses Verfahren.

- Konfigurierbarkeit der Speicherelemente zu einer Latch-Funktion.

  Einige Architekturen verfügen in ihren Speicherelementen sowohl über eine Flip-Flop- als auch über eine Latch-Funktion. Andere erlauben nur die Verwendung als Flip-Flop.

  Eine Latch-Funktion kann auch durch kombinatorische Logik konstruiert werden:

  ```
  Q = ( G &  GE & D) |
      (!G       & Q) |
      (     !GE & Q);
  ```

Der Nachteil dieser Anordnung besteht darin, daß sie eine kombinatorische Schleife enthält, die von den existierenden Place&Route-Werkzeugen nicht bei den Timing-Kriterien berücksichtigt wird. Solche Latches können daher im Echtzeitbetrieb Probleme verursachen.

Andererseits können in unterstützenden Architekturen als Latch konfigurierbare Speicherelemente vorteilhaft sein. Sie sparen kombinatorische Resourcen und ihr Timing wird berücksichtigt.

Das Verhalten von Latches kann mit flankengesteuerten Flip-Flops nicht erreicht werden. Eine Latch-Funktionalität kann in Designs aber durchaus wünschenswert sein.

Daher sollten Latches als architekturunabhängige Grundelemente zur Verfügung stehen. Bei nicht unterstützenden Architekturen können sie nach oben beschriebener Gleichung kombinatorisch realisiert werden.

- Asynchrone Set- bzw. Reset-Signale an den Speicherelementen.

  Die *XILINX*-Architekturen erlauben an einem Speicherelement jeweils entweder ein asynchrones Set- oder ein Reset-Signal, jedoch nicht beide gleichzeitig. Zudem besteht eine Abhängigkeit vom Startwert: Ein Speicherelement, das nach der Konfiguration auf den Zustand "1" initialisiert sein soll, kann nur einen asynchronen Set-Eingang besitzen. Für den Startzustand "0" ist analog nur ein Reset-Eingang zulässig.

  Die asynchrone Funktionalität kann bei Flip-Flops nicht emuliert werden, wenn sie nicht explizit vorhanden ist.

  Aus diesem Grund ist es nicht möglich, ein architekturunabhängiges Speicherelement zu konstruieren, das asynchrone Eingänge vollständig unterstützt.

  Bei *CHDL* wird dieses Problem so gelöst, daß Speicherelemente zwar architekturunabhängig sind, jedoch nicht hinsichtlich der asynchronen Set- und Reset-Eingänge.

  Bei der Umsetzung auf eine bestimmte Zielarchitektur wird automatisch die Verwendung dieser Eingänge überprüft. Ist die konkrete Verschaltung nicht zulässig, erhält der Anwender eine entsprechende Fehlermeldung.

  Dies bedeutet, daß Designs, die solche asynchronen Eingänge benutzen, eventuell nicht auf alle Architekturen umgesetzt werden können. Da der Einsatz asynchroner Signale bei FPGAs aufgrund der dadurch möglichen Timingprobleme ohnehin nicht empfohlen ist, stellt die hier gewählte Entscheidung keine ernsthafte Einschränkung dar.

- Unterschiedlicher Aufbau der Fast-Carry-Logik.

  Dieser Bereich ist vor allem für die arithmetischen Grundelemente relevant, die in beliebigen Bitbreiten Einsatz finden, so etwa Zähler, Inkrementer, Addierer usw.

  Die Fast-Carry-Logik ist innerhalb der *Virtex*-Architektur grundlegend anders implementiert als in der *XC4000E*-Reihe.

  Im äußeren Interface ergeben sich jedoch keine Unterschiede.

  Daher bietet es sich an, diese arithmetischen Bauteile nach außen hin architekturunabhängig zu gestalten. Die interne Implementierung erfolgt in Abhängigkeit von einem Architektur-Flag speziell für die jeweilige Fast-Carry-Struktur.

- Unterschiedlicher Aufbau der Pin-Interfaces (IOBs).

  Abhängig von der Architektur werden Flip-Flops in den IOBs unterschiedlich unterstützt.

  Bei einigen wenigen Architekturvarianten (z.B. *XC4000H*) sind in den IOBs keine Flip-Flops vorhanden.

IOB-Flip-Flops können für das Timing eines FPGA-Designs nach außen von erheblicher Bedeutung sein. Die Verzögerungszeit der Ausgänge ist wesentlich kürzer, als dies selbst ein optimal geführtes Signal aus der CLB-Matrix bieten könnte.

Daher sollte ein architekturunabhängiges Pin-Interface integrierte Flip-Flops ermöglichen. Bei den nicht unterstützenden Architekturen kann die prinzipielle Funktionalität, nicht aber der Timing-Vorteil durch Verwendung eines CLB-Flip-Flops erreicht werden.

Komplexer ist die Situation bei Architekturen, die sowohl Ausgangs- als auch Eingangs-Flips-Flops in den IOBs unterstützen. Teilweise werden hier unterschiedliche Clocks ermöglicht.

Dieser Fall läßt sich in einer nicht unterstützenden Architektur nicht emulieren. Ähnlich verhält es sich mit Clock-Enable (CE) Signalen, die teilweise unterstützt werden. Auch diese lassen sich nicht emulieren, da in den IOBs die erforderliche Logik fehlt, bzw. bei einer Implementierung mit CLB-Logik der Timing-Vorteil nicht produzierbar ist.

Analog zu der Problematik der asynchronen Signale bei Speicherelementen wird bei *CHDL* folgende Lösung gewählt:

Die Pin-Interfaces enthalten ein einziges, entweder im Eingangs- oder im Ausgangspfad verfügbares Flip-Flop ohne Clock-Enable-Eingang. Wird in Designs eine weitergehende Funktionalität benötigt, muß auf spezielle Elemente der jeweiligen Architektur zurückgegriffen werden.

9.4.11 Übersicht über die Pin- und Bauteilklassen

Die *CHDL*-Verwaltungsklassen (Abb. 9.6) stellen die grundsätzliche Funktionalität bereit, die zur Hardwarebeschreibung mit C++ erforderlich ist.

Die *CHDL*-Pin-Klassen (Abb. 9.7) bilden die Interfaces der vordefinierten Grundelemente sowie der Simulationsmodelle.

Die *CHDL*-Node-Klassen (Abb. 9.8) stellen entsprechend die Interfaces der anwenderdefinierten Module dar.

Die *CHDL*-Pad-Klassen (Abb. 9.9) repräsentieren die Schnittstellen zu den Gehäusepins.

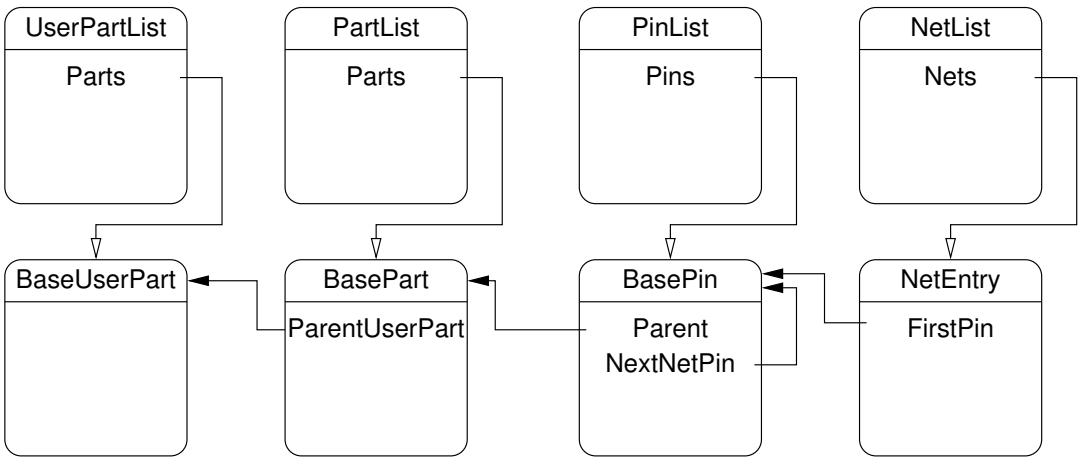


Abbildung 9.6: *CHDL*-Verwaltungsklassen

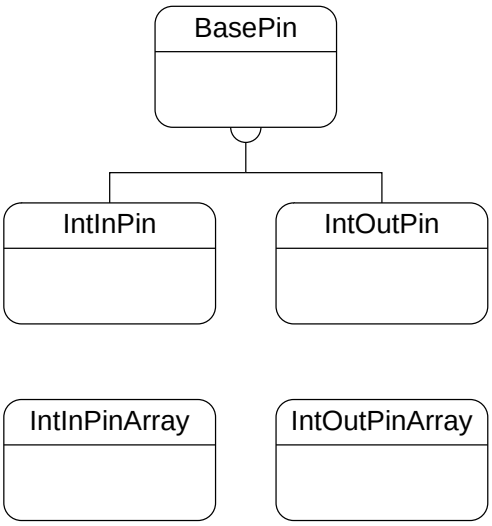


Abbildung 9.7: *CHDL*-Pin-Klassen

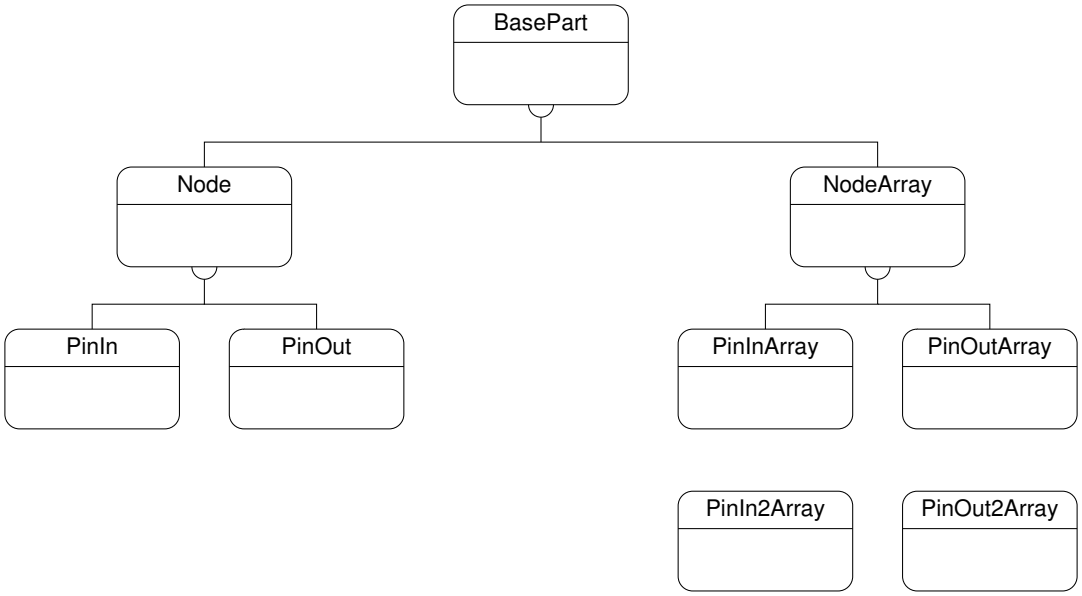


Abbildung 9.8: *CHDL*-Node-Klassen

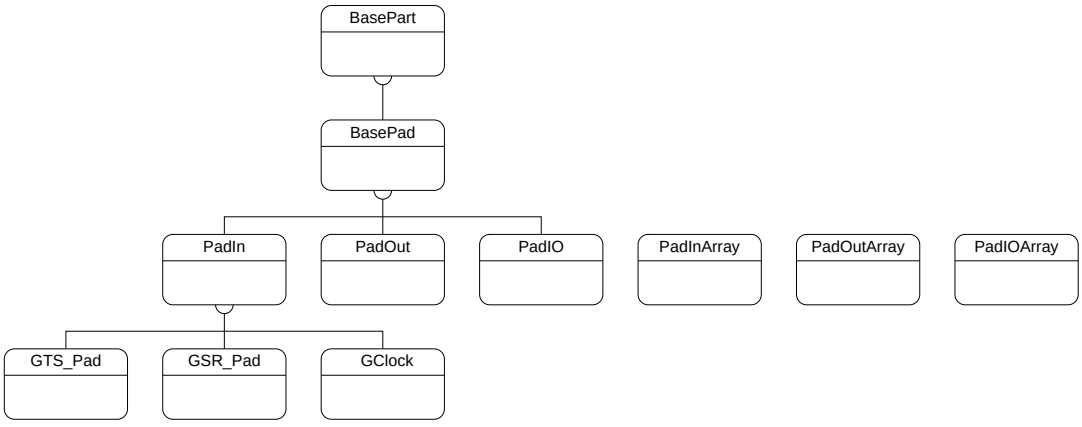


Abbildung 9.9: *CHDL*-Pad-Klassen

9.5 Modularisierung

9.5.1 Bedeutung der Modularisierung

Die im vorigen Abschnitt erläuterte strukturelle Hardwarebeschreibung stößt ohne weitere Maßnahmen schnell an die Grenzen der Übersichtlichkeit.

Es ist notwendig, daß sich die Gesamtbeschreibung in kleinere Einheiten (Module) zerlegen läßt, die getrennt entwickelt und gewartet werden können. Diese Zerlegung sollte in beliebiger Tiefe möglich sein, d.h. aus kleinen Einheiten lassen sich größere bilden, die wiederum zu größeren zusammengeschaltet werden können.

Dieses Vorgehen hat außer der Verbesserung der Übersichtlichkeit noch weitere Vorteile:

- Wiederverwendung.

  Bei geeigneter Struktur können bereits existierende Module direkt in neuen Schaltungen eingesetzt werden. Dies beschleunigt die Designerstellung erheblich, da die Module nicht jedesmal neu implementiert werden müssen.

- Sukzessive Elimination von Fehlern.

  Je öfter ein Modul in neuen Schaltungen wiederverwendet wird, desto schneller werden eventuelle Fehler entdeckt. Die entdeckten Fehler werden beseitigt und das fehlerhafte Modul durch die korrigierte Version ersetzt. Auf diese Weise erhöht sich die Qualität der wiederverwendeten Module kontinuierlich.

- Die Komplexität der erstellbaren Anwendungen wird deutlich erhöht, da der Entwickler nur die wichtigsten Verhaltenseigenschaften, nicht aber alle Details der wiederverwendeten Module kennen muß.

- Module können nach Funktionsgruppen getrennt in Anwenderbibliotheken gesammelt und von mehreren Arbeitsgruppen gewartet und genutzt werden. Auf diese Weise kann jede Arbeitsgruppe vom speziellen Know-How der anderen Gruppen profitieren.

9.5.2 Erstellen von *CHDL*-Modulen

Interface

CHDL unterstützt die Modularisierung durch die Möglichkeit, Teile des Designs in eigenen Klassen zusammenzufassen. Die Verschaltung mit anderen Modulen bzw. Bauteilen erfolgt über ein frei definierbares Interface.

Die Interface-Definition kann die Datentypen

- `PinIn`.

  Ein einzelnes Eingangssignal in das Modul hinein.

- `PinOut`.

  Ein einzelnes Ausgangssignal aus dem Modul heraus.

- `PinInArray`, `PinIn2Array`.

  Ein Signalvektor (eindimensional bzw. zweidimensional) in das Modul hinein.

- `PinOutArray`, `PinOut2Array`.

  Ein Signalvektor (eindimensional bzw. zweidimensional) aus dem Modul heraus.

enthalten.

Um dem Konzept der Datenkapselung zu folgen, sollte die innere Schaltung des Modules nur über dieses Interface mit der äußeren Umgebung verbunden werden.

```
class MyModule : public BaseUserPart
{
public:
  PinIn   A;
  PinIn   B;
  PinIn   CLK;
  PinOut  D;

  MyModule ( const char* Name );
};
```

Implementierung

Die Implementierung des Modules erfolgt innerhalb des Konstruktors. Der darin enthaltene Code wird dadurch direkt beim Anlegen eines neuen Objektes dieser Klasse ausgeführt. Um die Hierarchie der Modulnamen korrekt aufzubauen, muß, wie bereits erwähnt, sofort zu Beginn des Konstruktors eine Instanz von `ProcessName` erzeugt werden. Weiterhin müssen die einzelnen Konstruktoren der Interface-Objekte explizit aufgerufen werden, damit diese ihre Laufzeitnamen erfahren. Der Modulname wird direkt dem Konstruktor der Basisklasse `BaseUserPart` übergeben. Innerhalb der Implementierung müssen alle Laufzeitnamen, die an erzeugte Module oder Bauteile vergeben werden, eindeutig sein. Um die innerhalb des Modules existierenden Elemente global eindeutig zu kennzeichnen, erhalten sie den Modulnamen als Präfix. Wird z.B. später ein Modul "M1" instanziert, erhält das darin enthaltene Flip-Flop "FF1" den globalen Namen "M1/FF1".

```
MyModule::MyModule ( const char* Name )
  : BaseUserPart(Name),
    A(this,"A"),
    B(this,"B"),
    CLK(this,"CLK"),
    D(this,"D")
{
  ProcessName _N(this);

  DFF  FF1("FF1",CLK);

  FF1    = A;
  FF1.CE = B;

  D = FF1;
}
```

Parametrisierung

Der Code im Konstruktor kann beliebig parametrisiert werden. Auf diese Weise lassen sich allgemeinere Module gestalten, die für die einzelnen Parameter jeweils speziell angepasste Implementierungen erzeugen.

Es ist auch möglich, die Parametrisierung bereits auf das Interface anzuwenden, um etwa die Breite von Bussen anzupassen. Dabei können zur Angabe der Busbreite alle an dieser Position zulässigen C++-Ausdrücke eingesetzt werden. Eventuell notwendige bedingte Anweisungen können mit dem Konditional-Operator realisiert werden.

```
MyModule::MyModule ( const char* Name, int DataWidth )
  : BaseUserPart(Name),
    DataIn(this,DataWidth,"DataIn"),
    CLK(this,"CLK"),
```

```
    DataOut(this,DataWidth ? DataWidth/2 : 1,"DataOut")
...
```

Es können auch mehrere Konstruktoren mit jeweils unterschiedlichen Parameterkombinationen implementiert werden. Dazu ist es weiterhin zulässig, gemeinsame Codeabschnitte in eigene Methoden auszulagern, die dann aus den Konstruktoren aufgerufen werden. Die Erzeugung des `ProcessName`-Objektes sollte jedoch nur in den Konstruktoren erfolgen.

Instanziierung

Die Erzeugung eines neuen Objektes eines Anwendungsmodules erfolgt genau wie bei den Primitiven. Jedes neue Objekt muß einen eigenen Namen erhalten. Es können sowohl automatische Objekte als auch dynamische mittels `new` angelegt werden.

```
MyModule* M1;

MyModule  M2("M2",8);

M1 = new MyModule("M1",16);
```

9.5.3 Aufbau von Anwenderbibliotheken

Analog zu den Verfahren, die die C++-Werkzeuge für die Erstellung konventioneller Anwendungsprogramme bieten, können Anwendermodule in Bibliotheken zusammengefaßt werden. Auf diese Weise entstehen Programm-Module, die getrennt kompiliert und erst in der eigentlichen Endanwendung zusammengelinkt werden.

Komplexe *CHDL*-Projekte können so unter Verwendung moderner C++-Entwicklungssysteme (z.B. Microsoft Visual Studio) wie herkömmliche Programmprojekte verwaltet werden. Auch die Verwendung von Versionskontrollsystemen (z.B. CVS) ist problemlos möglich.

9.6 Implementierung von Zustandsmaschinen

9.6.1 Modifizierte *Moore*-Maschine

Die maximale Taktfrequenz von FPGAs wird im wesentlichen durch die kombinatorischen Logikpfade bestimmt. Bei der Aneinanderreihung mehrerer Module können sich diese Pfade verlängern. Sollen hohe Taktfrequenzen erreicht werde, muß der Entwickler dieses Anwachsen der kombinatorischen Logik im Auge behalten können.

Zur Implementierung von Zustandsmaschinen empfiehlt sich folgende Modifikation der *Moore*-Maschine, um ein kontrollierbareres Zeitmodell zu erhalten (Abb. 9.10):

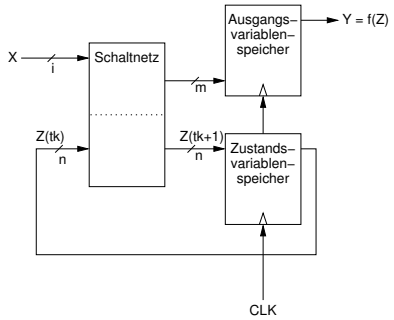


Abbildung 9.10: Modifizierte *Moore*-Zustandsmaschine

Der bisherige Zustandsvariablenspeicher wird aufgeteilt in Zustandsvariablen und Ausgangsvariablen. Dadurch zerfällt die kombinatorische Logik in einen Teil, der die Zustandsübergänge bestimmt, und einen anderen, der das Verhalten der Ausgangsvariablen festlegt.

Eingangs- und Zustandssignale liegen jeweils an beiden Teilen an. Die Ausgangssignale werden direkt aus dem Ausgangsvariablenspeicher, also aus Flip-Flops, geführt und stellen Startpunkte für kombinatorische Logikpfade dar.

Dies hat den Vorteil, daß die gesamte Maschine im Detail aus einzelnen Flip-Flops mit kombinatorischer Logik an den Dateneingängen gesehen werden kann, wodurch die Durchlaufverzögerungen besser abschätzbar sind.

Nachteil an diesem Modell ist die höhere Anzahl benötigter Flip-Flops, da jedes Ausgangssignal durch ein eigenes Flip-Flop gebildet wird, während bei der ursprünglichen *Moore*-Maschine durch die zusätzliche Logik mehrere Ausgangssignale aus dem codierten Zustand generiert werden könnten.

9.6.2 *One-Hot*-Encoding

Bei FPGAs ist die Anzahl der Eingangssignale an den Lookup-Tabellen auf vier begrenzt. Daher kann es vorteilhaft sein, den aktuellen Zustand nicht komprimiert in den Zustandsvariablen zu kodieren, sondern für jeden möglichen Zustand eine Variable vorzusehen ("*One-Hot*-Encoding" [74]). Hier kann mit einem einzigen Signal geprüft werden, ob sich die Maschine in einem bestimmten Zustand befindet, während bei der konventionellen Kodierung immer alle Zustandsvariablen geprüft werden müssen.

Dieses Verfahren kann sich jedoch auch nachteilig auswirken, wenn geprüft werden muß, ob sich die Maschine in einem von mehreren bestimmten Zuständen befindet. Hier könnte durch eine optimierte Kodierung der Zustände eventuell eine einfachere Logik erreichbar sein. Jedoch führt auch dies nicht generell zum besseren Ergebnis. Oft müssen mehrere der oben genannten Bedingungen geprüft werden. Es wird nicht immer möglich sein, eine Kodierung zu finden, die für alle Bedingungen ein optimales Ergebnis darstellt.

Es existiert jedoch noch ein anderer Grund, weshalb für das Zustandsmaschinenmodell von *CHDL* das *One-Hot*-Verfahren gewählt wurde: Mit einer geringen Modifikation erlaubt dieses im Gegensatz zu den anderen Modellen, daß sich eine Maschine gleichzeitig in mehreren Zuständen befindet. Dies scheint zunächst im Widerspruch zu den Grundsätzen des Zustandsmaschinendesigns zu stehen. Es ermöglicht jedoch, wie später demonstriert wird, einige mächtige Verfahren zur Beschreibung mehrerer Threads in Zustandsmaschinen sowie zur Implementierung von Pipeline-Kontrollern.

9.6.3 Flußdiagramme zur Beschreibung von Zustandsmaschinen

Flußdiagramme (Abb. 9.11) können eingesetzt werden, um den Ablauf einer Zustandsmaschine darzustellen. Es gibt Anweisungsblöcke, Bedingungen und Übergangspfeile. Die Anweisungsblöcke definieren die einzelnen Zustände der Zustandsmaschine, die Übergangspfeile zusammen mit den Bedingungen die Zustandsübergänge.

Die Zustandsübergänge werden mit den Zustandsvariablen und der kombinatorischen Übergangslogik realisiert.

Die Anweisungen in den Anweisungsblöcken werden ausgeführt, wenn der entsprechende Zustand aktiv wird. Dies läßt sich implementieren, indem die Clock-Enable-Signale der Ausgangsvariablen mitverwendet werden. An der Übergangslogik liegt der jeweils nächste Zustand an. Dieses Signal wird eingesetzt, um Aktionen außerhalb der Zustandsmaschine zeitgleich mit dem Übergang in den nächsten Zustand auszuführen.

9.6.4 Automatische Erzeugung von Zustandsmaschinen aus Flußdiagrammen

Eine Zustandsmaschine, die aus einem Flußdiagramm gebildet wird, stellt eine Kombination aus einem *One-Hot*-Controller und einem Datenpfad dar (Abb. 9.12).

Es müssen folgende Komponenten gebildet werden:

- Start-Logik.

  Es gibt einen impliziten Startzustand `S0`, der direkt nach der Konfiguration oder einem Reset aktiv ist. Nach dem ersten Takt wird dieser Startzustand inaktiv und bleibt im inaktiven Zustand.

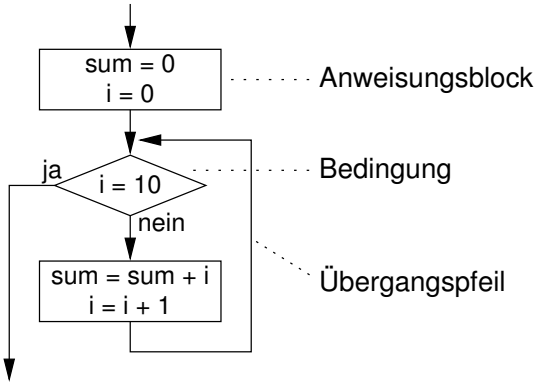


Abbildung 9.11: Darstellung als Flußdiagramm

- Instanziierung der State-Flip-Flops.

  Jedem Anweisungsblock des Flußdiagramms wird ein Zustand und damit ein Flip-Flop zugeordnet. Im späteren Ablauf ist zu jedem Zeitpunkt immer nur eines dieser Flip-Flops aktiv.

- Zustandsübergangsgleichungen.

  Die einzelnen Zustands-Flip-Flops werden durch Übergangsgleichungen zusammengeschaltet. Dadurch ergibt sich der im Flußdiagramm definierte Ablauf der Anweisungsblöcke.

- Daten-Gleichungen des Datenpfades.

  Es ist möglich, einer Variable in mehreren Anweisungsblöcken Werte zuzuweisen. Das bedeutet, daß sich am Dateneingang der Variable eine Logik befinden muß, die in Abhängigkeit vom aktuellen Zustand den korrekten Wert auswählt.

- Enable-Gleichungen des Datenpfades.

  In allen Anweisungsblöcken, in denen eine Variable nicht erwähnt wird, behält sie ihren aktuellen Zustand. Die Zeitpunkte, in denen sie einen neuen Wert erhält, werden durch die Enable-Gleichungen bestimmt.

Am komplexesten ist die Bildung der Übergangsgleichungen. Ein bestimmtes Zustands-Flip-Flop wird beim nächsten Takt aktiv, wenn folgende Bedingungen erfüllt sind:

- Eines der Zustands-Flip-Flops, die einem direkten Vorgängerzustand im Flußdiagramm zugeordnet sind, ist aktiv.
- Alle Bedingungen, die sich auf dem Übergangsweg von diesem Vorgängerzustand befinden, sind erfüllt.

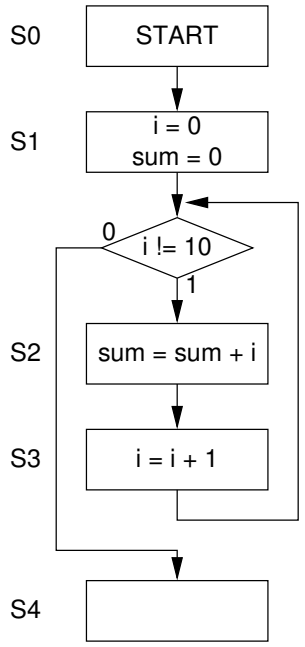
Für jeden Zustand ist also der direkte Vorgängerzustand zu ermitteln. Außerdem sind alle Bedingungen in die Gleichung zu integrieren. Eine Übergangsgleichung ist ein disjunktiver Term bestehend aus den Teilgleichungen für jeden möglichen Weg zu diesem Zustand. Jede Teilgleichung ist ein Produktterm aus dem direkten Vorgängerzustand und allen Gleichungen, die auf dem Weg zum aktuellen Zustand liegen.

Zustand `S1` folgt immer auf `S0`. Der Zustand `S2` ist von `S1` aus zu erreichen, wenn `i != 10`. Oder von `S3` aus, ebenfalls, wenn `i != 10`. Zustand `S3` folgt immer auf `S2`. Der Zustand `S4` ist von `S1` aus zu erreichen, wenn `i == 10`. Oder von `S3` aus, ebenfalls, wenn `i == 10`.

Gruppiert man alle Produktterme, die von einem bestimmten Zustand ausgehen, so ist bei aktivem Zustand für alle beliebigen Kombinationen der enthaltenen Bedingungen immer

genau einer aktiv. Das bedeutet, von einem Zustand aus gibt es immer genau einen Weg zum nächsten. Das folgt aus dem Prinzip des *One-Hot*-Encoding.

```
NS2 = S1 &  (i != 10)
NS4 = S1 & !(i != 10)
```




```
NS1 = S0
NS2 = (S1 & (i != 10)) | (S3 & (i != 10))
NS3 = S2
NS4 = (S1 & !(i != 10)) | (S3 & !(i != 10))

S1 = NS1
S2 = NS2
S3 = NS3
S4 = NS4

i = (NS1 & 0) | (NS3 & (i+1))
i.CE = NS1 | NS3
sum = (NS1 & 0) | (NS2 & (sum+i))
sum.CE = NS1 | NS2
```

Abbildung 9.12: Bildung der Gleichungen

S_i ist dabei der aktuelle Zustand, NS_i der nächste Zustand.

Dies ermöglicht eine Kombination aus Kontroller und Datenpfad, wobei die NS_i die Steuersignale für den Datenpfad sind (Auswahl der Daten bei mehreren möglichen und Bestimmung des Übernahmezeitpunktes über Clock-Enable-Signale).

9.6.5 Optimierungsmöglichkeiten

Da die Übergangsgleichungen immer nach dem gleichen Prinzip gebildet werden, ergeben sich auch stets ähnliche Möglichkeiten zur Logikoptimierung. Auf den verschiedenen Ebenen der Gleichungen entstehen unterschiedliche Optimierungssituationen:

Produktterme

Bei der Bildung der Produktterme können in Abhängigkeit von den einzelnen Bedingungen auf dem Übergangspfad konstante Teilausdrücke entstehen. Dies ist dann der Fall, wenn im Flußdiagramm Abhängigkeiten zwischen den einzelnen aufeinanderfolgenden Bedingungen existieren. Dann kann es vorkommen, daß manche Übergangspfade niemals verwendet werden.

```
NS2 = S1 & A & B & !A;  --> NS2 = 0
```

Es kann auch vorkommen, daß ein Produktterm Teilausdrücke doppelt enthält. Dies ist dann der Fall, wenn im Übergangspfad mehrmals die gleiche Bedingung enthalten ist.

```
NS2 = S1 & A & B & A;  --> NS2 = S1 & A & B
```

Übergangsgleichungen

Durch die disjunktive Anordnung der Gesamtübergangsgleichung kann es zu redundanten Variablen kommen, wenn zwei parallele Übergangspfade so beschaffen sind, daß immer einer von beiden gültig ist.

```
NS2 = S1 & A & !B;
NS2 = S1 & A &  B;  --> NS2 = S1 & A
```

Die oben genannten Fälle sind allerdings Spezialfälle, in denen die Bedingungen des Flußdiagramms unsauber angegeben sind.

Größere Bedeutung hat die nachfolgend erläuterte Optimierung der Enable-Gleichungen sowie der Dateneingangs-Gleichungen.

Enable-Gleichungen

Durch die disjunktive Anordnung der Enable-Gleichungen kommt es oft zur Reduzierung der Menge von Produkttermen. Dies ist dann der Fall, wenn einer Variable in mehreren Zuständen ein Wert zugewiesen wird und sich dabei die Bedingungen der einzelnen Übergangsgleichungen reduzieren.

```
NS1 =  S0
NS2 = (S1 & !A & !B) |
      (S2 & !A & !B)
NS3 = (S1 &  A) |
      (S2 & !A &  B)

i.CE = NS1 | NS2 | NS3
```

reduziert sich zu

```
NS1   = S0
NS2_3 = (S1 & !A & !B) |
        (S1 &  A) |
        (S2 & !A)

i.CE = NS1 | NS2_3
```

Zusätzlich kann es vorkommen, daß sich in der Enable-Gleichung alle Bedingungen reduzieren und alle Zustands-Flip-Flops enthalten sind. Dies ist dann der Fall, wenn einer Variable in jedem Zustand ein Wert zugewiesen wird. Die Enable-Gleichung reduziert sich dann auf "1".

Die oben genannten Logikoptimierungen können die Enable-Gleichungen deutlich reduzieren.

Datenpfad-Gleichungen

Die Datenpfad-Gleichungen sind disjunktive Verknüpfungen von Produkttermen. Die Produktterme enthalten einen Übergangsterm und einen Datenterm.

Bei Steuersignalen in Flußdiagrammen besteht der Datenterm oft aus der Konstanten "0" oder "1". Dies reduziert den Produktterm auf "0" bzw. den Übergangsterm. Bei diesen Übergangstermen ist analog zu oben eine Reduzierung der enthaltenen Bedingungen möglich.

9.6.6 Mehrere Threads in Zustandsmaschinen

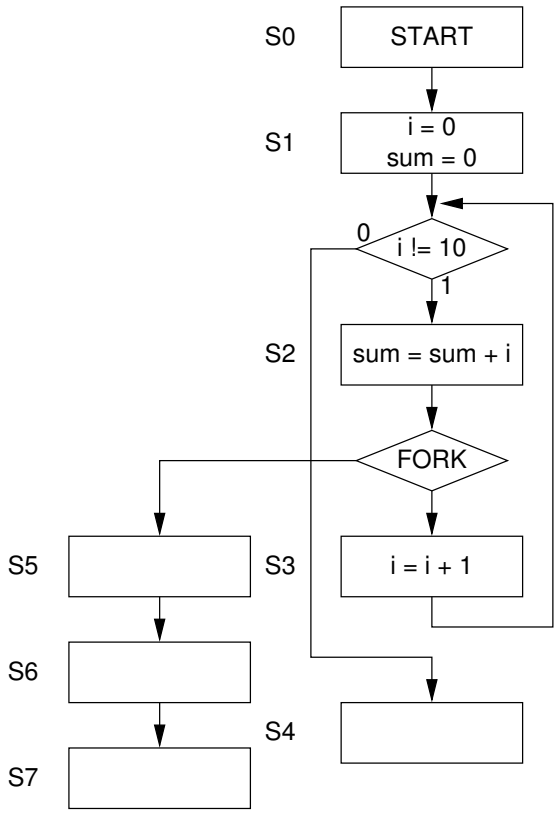
Wie bereits erwähnt, ermöglicht eine Modifizierung des *One-Hot*-Verfahrens mehrere aktive Zustände in einer Zustandsmaschine: Die Restriktion, daß zu jedem Zeitpunkt genau ein State-Bit aktiv ist, wird aufgegeben. Die Zulässigkeit mehrerer aktiver Zustände kann jetzt genutzt werden, um in einer Zustandsmaschine mehrere Threads zu realisieren (Abb. 9.13).

Ein solches Vorgehen bietet folgende Vorteile:

- Zustandsmaschinen, die eine zusammengehörige Aufgabe bearbeiten, jedoch unabhängige Ablaufpfade erfordern, können in einer einzigen Maschine mit einem gemeinsamen Interface implementiert werden. Dies kann die Übersichtlichkeit und die Datenkapselung des Designs verbessern.

- Oft steht der Entwickler vor dem Problem, daß eine Zustandsmaschine Kontrollsignale erzeugt, die dann mit einer bestimmten Verzögerung weitergeleitet werden müssen. Ein Beispiel sind Signale, die die Gültigkeit von Lesedaten bei der Ansteuerung von SDRAMs mit einer CAS-Latency anzeigen. Solche Signale können nicht innerhalb der normalen Zustandsbeschreibung erzeugt werden, da sie unabhängig vom Hauptausführungspfad sind. Eine Implementierungsmöglichkeit besteht darin, die betreffenden Signale durch eine nachfolgende strukturelle Beschreibung in Form einer Verzögerungskette zu realisieren.

  Die Zulässigkeit mehrerer Threads erlaubt in diesen Fällen die homogene Beschreibung als Zustandsmaschine ohne eine separate strukturelle Logik.




```
NS1 = S0
NS2 = (S1 & (i != 10))
      (S3 & (i != 10))
NS3 = S2
NS4 = (S1 & !(i != 10))
      (S3 & !(i != 10))

NS5 = S2
NS6 = S5
NS7 = S6

S1 = NS1      S5 = NS5
S2 = NS2      S6 = NS6
S3 = NS3      S7 = NS7
S4 = NS4

i = (NS1 & 0) |
    (NS3 & (i+1))
i.CE = NS1 | NS3
sum = (NS1 & 0) |
      (NS2 & (sum+i))
sum.CE = NS1 | NS2
```

Abbildung 9.13: Zustandsmaschine mit mehreren Threads

9.6.7 Beschreibung von Zustandsmaschinen in *CHDL*

CHDL stellt eine Methode zur Verfügung, mit der Flußdiagramme mittels C++-Code beschrieben werden können (Abb. 9.14). Die Implementierung erfolgt dann automatisch nach den zuvor erläuterten Methoden.

Es wird zunächst eine Klasse von `BaseSM` abgeleitet und das gewünschte Interface definiert. Zur Beschreibung des Flußdiagrammes stehen nun folgende Funktionen zur Verfügung:

- `BeginState();`

  Diese Funktion kennzeichnet den Beginn eines Anweisungsblocks. Zuweisungen zu Variablen können nur innerhalb eines solchen Blocks angegeben werden. Jeder Block wird später als eigener Zustand implementiert.

- `EndState();`

  Diese Funktion kennzeichnet das Ende eines Anweisungsblocks.

- `LABEL ( const char* Name );`

  Kennzeichnet ein Sprungziel. Dem Ziel kann ein beliebiger String zugeordnet werden. Der Name muß innerhalb der Beschreibung eindeutig sein.

- `GOTO ( const char* Name );`

  Kennzeichnet einen unbedingten Sprung zum angegebenen Ziel. Das Ziel muß in dieser Beschreibung existieren.

- `IF ( BasePin&, const char* Name1, [const char* Name2] );`

  Kennzeichnet einen bedingten Sprung zum angegebenen Ziel. Der Sprung wird ausgeführt, wenn der angegebene Pin den Wert "1" besitzt. Optional kann ein zweites Sprungziel angegeben werden, das im Fall "0" ausgeführt wird. Die Ziele müssen in dieser Beschreibung existieren. Statt einer direkten Pin-Angabe kann auch ein Ausdruck angegeben werden, der als Ergebnis einen Pin liefert, z.B. `(i == 0)`.

- `IFNOT ( BasePin&, const char* Name1, [const char* Name2] );`

  Analog zu `IF`, aber der Sprung wird ausgeführt, wenn der Pin den Wert "0" besitzt.

- `FORK ( const char* Name );`

  Erzeugt eine Verzweigung zum angebenen Ziel, wobei hier das Ausführungstoken verdoppelt wird. Diese Anweisung entspricht dem Starten eines neuen Threads. Der bisherige Thread setzt seine Ausführung normal fort.

- `FORKIF ( BasePin&, const char* Name );`

  Analog zu `FORK`, aber der neue Thread wird nur erzeugt, wenn der angebene Pin den Wert "1" besitzt.

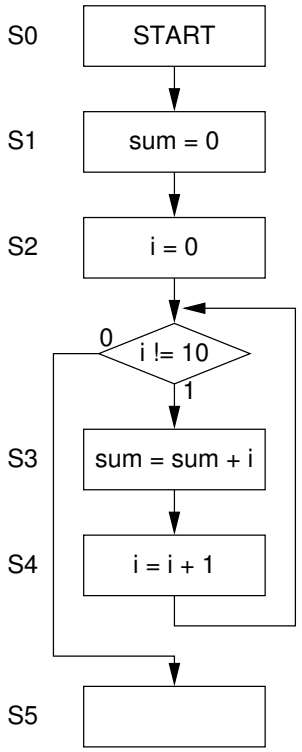
- `END ( void );`

  Kennzeichnet das Ende eines Threads. Das Ausführungstoken wird gelöscht.

- `FLOWCONTROL ( BasePin& );`

  Fügt bis zur nächsten `END`-Anweisung das angegebene Signal zur Flußkontrolle von Pipelines ein.

Innerhalb eines Anweisungsblocks sind keine Sprunganweisungen zulässig. Es können beliebig viele Sprunganweisungen hintereinander angegeben werden. Sprungziele sind außerhalb von Anweisungsblöcken überall zulässig, auch zwischen einzelnen Sprunganweisungen.




```
    BeginState();
      sum = 0;
    EndState();

    BeginState();
      i = 0;
    EndState();

LABEL("Loop");

    IFNOT (i != 10,"Ende")

    BeginState();
      sum = sum + i;
    EndState();

    BeginState();
      i = i + 1;
    EndState();

    GOTO("Loop");

LABEL("Ende");

    BeginState();
    EndState();
```

Abbildung 9.14: *CHDL*-Beschreibung eines Flußdiagrammes

9.6.8 Pipeline-Kontroller

Eine Pipeline besteht aus mehreren hintereinander angeordneten Registerstufen, zwischen denen mittels kombinatorischer Logik Rechenoperationen ausgeführt werden. Mit jedem Takt werden die in Registern enthaltenen Daten über diese Logik in die nachfolgende Stufe weitergeschoben.

Eine solche Anordnung ist immer dann vorteilhaft, wenn

- auf eine Folge von Datenworten jeweils gleiche oder ähnliche Rechenoperationen angewendet werden müssen,
- diese Operationen als einzelne kombinatorische Logik zu komplex wäre,
- sich die Operationen in einzelne, voneinander unabhängige Teiloperationen zerlegen lassen

und

- die Aneinanderreihung der Teiloperationen eine statische Struktur aufweist, d.h. die Reihenfolge und die Anzahl der Teiloperationen nicht abhängig von den Datenworten ist.

Das Pipeline-Verfahren ist dabei nicht beschränkt auf Rechenoperationen, die nur jeweils ein einzelnes Datenwort betreffen. Es können durchaus auch vorangehende oder nachfolgende Datenworte in die Berechnung eingehen. Voraussetzung ist lediglich, daß eine statische Struktur vorliegt.

Der Vorteil von Pipeline-Anordnungen liegt darin, daß alle Teilrechenoperationen parallel ausgeführt werden und dadurch mit jedem Takt ein Datenwort auf seiner jeweiligen Stufe bearbeitet wird. Dies führt dazu, daß, abgesehen von der Zeit, die die Pipeline zum Füllen bzw. Leeren benötigt, mit jedem Takt ein Datenwort komplett verarbeitet aus der Pipeline herausgeschoben wird.

Ein Datenwort benötigt für seine komplette Verarbeitung mehrere Takte und es können sich mehrere Datenworte gleichzeitig in der Pipeline befinden. Daher bietet es sich an, die einzelnen Stufen mit einem *Valid*-Flag zu versehen, das mit den Daten mitgeschoben wird. Am Flag der letzten Stufe lassen sich dann leicht die vollständigen Worte erkennen.

Kann nicht mit jedem Takt ein Datenwort in die Pipeline eingefügt werden, sind vier Anordnungen denkbar:

1. Weiterschieben aller Daten und Flags mit jedem Takt.

   Die einzelnen Stufen werden hier so aneinandergereiht, daß sie mit jedem Takt die Daten und das *Valid*-Flag der vorangehenden Stufe übernehmen.

   Wird in einem Takt keine neues Datenwort eingefügt, ist diese Situation durch ein inaktives Flag gekennzeichnet. Die entstehende Lücke wird gemeinsam mit den Daten weitergeschoben.

   Am Ende der Pipeline bewirken die Lücken dann, daß in diesem Takt kein gültiges Ergebniswort zur Verfügung steht.

   Eine solche Anordnung kann mit einem kontinuierlich laufenden Förderband verglichen werden. Auf das Band gelegte Elemente werden mit gleichmäßiger Geschwindigkeit zum anderen Ende befördert, Lücken ebenso.

   Ungeeignet ist diese Methode jedoch, wenn innerhalb einer Stufe mehrere Datenworte benötigt werden und die Pipeline durch nicht kontinuierliches Füllen Lücken enthalten kann.

2. Weiterschieben der Daten und Flags mit globaler Flußkontrolle auf der Ausgangsseite.

   Hier existiert zusätzlich ein Kontrollsignal, mit dem ein Weiterschieben von der Ausgangsseite verhindert werden kann. Dieses wirkt gleichermaßen auf alle Stufen.

   Hier muß beachtet werden, daß bei aktivem Kontrollsignal kein neues Datenwort in die Pipeline eingefügt werden kann. Die Eingangsseite muß diesen Fall behandeln können.

   Vergleichbar ist diese Anordnung mit einer Variante von Fall 1, in der die Ausgangsseite das Förderband anhalten kann.

3. Weiterschieben der Daten und Flags mit globaler Flußkontrolle auf der Eingangsseite.

   Hier existiert zusätzlich ein Kontrollsignal, mit dem ein Weiterschieben von der Eingangsseite verhindert werden kann. Dieses wirkt gleichermaßen auf alle Stufen.

   Diese Anordnung ist vergleichbar mit einer Röhre, in die zu beliebigen Zeitpunkten (Takten) Elemente eingeschoben werden. Diese schieben die vor ihnen befindlichen Elemente weiter. Sobald die Röhre vollständig gefüllt ist, verläßt bei jedem neuen Einschieben ein Element am Ende die Röhre.

   Zu beachten ist hierbei, daß das Kontrollsignal auch Auswirkungen auf das Entstehen von gültigen Ergebniswerten hat. Solange keine neuen Daten in die Pipeline geschoben werden, stehen auch keine Ergebnisse zur Verfügung.

   Dies erfordert zusätzliche Maßnahmen am Ende des Prozesses, wenn die Pipeline geleert werden muß, um die restlichen darin enthaltenen Ergebnisse auszulesen.

   Entsprechend dem obigen Röhrenvergleich müssen die restlichen Elemente auf andere Weise zum Ende geschoben werden.

4. Dynamisches Weiterschieben der Daten.

   Die Stufen werden über eine zusätzliche Logik verbunden, die ein lokal gesteuertes Weiterschieben der Daten und *Valid*-Flags erlaubt. Daten und *Valid*-Flag werden von der Stufe S_n in die Stufe S_{n+1} weitergeschoben, wenn das Flag aktiv ist und die Stufe

S_{n+1} frei ist bzw. im nächsten Takt frei würde. Eine Stufe wird frei, wenn sie ihr Datenwort und aktives Flag erfolgreich in die nachfolgende Stufe schieben kann.

Gemäß dem obigen Vergleich mit einer Röhre entspricht dieses Verfahren der Situation, daß die Röhre ein Gefälle aufweist. Dadurch bewegen sich eingefügte Elemente mit einer bestimmten (gleichmäßigen) Geschwindigkeit durch die Röhre. Werden am Ende Elemente nicht rechtzeitig entnommen, kann dies zu einem Rückstau führen, sobald neue Elemente nachfolgen. Lücken wie im Fall 1 können hier nicht entstehen.

Den größten Implementierungsaufwand erfordert Fall 4. Es ist eine Flußkontrolle zwischen allen Stufen der Pipeline erforderlich. Die Stufe S_n kann ihre Daten nur weiterschieben, wenn Stufe S_{n+1} frei ist bzw. frei würde. S_{n+1} würde frei, wenn S_{n+2} frei ist bzw. frei würde usw. Dadurch zieht sich ein kombinatorischer Logikpfad durch alle Pipeline-Stufen, der für das Timing des Gesamtdesigns zum Problem werden kann. Es ist zwar möglich, diesen Pfad durch Flip-Flops zu entkoppeln, jedoch benötigt diese Methode Register, die kurzfristig in den Datenpfad geschaltet werden können. Das erfordert weitere Resourcen an Logik und Speicherelementen.

Der Vorteil von Anordnung 4 liegt in der gleichzeitig vorliegenden Pufferfunktion der Pipeline. Es wäre zulässig, an der Ausgangsseite den Datenfluß kurzzeitig zu unterbrechen, ohne daß dies zu einer Störung auf der Eingangsseite führen würde.

Die Fälle 1, 2 und 3 sind mit wenig Aufwand realisierbar und ermöglichen eine starke Zeitentkopplung der Logikpfade und somit hohe Taktfrequenzen.

Am flexibelsten ist eine Kombination von Fall 2 und 3. Sowohl die Eingangs- als auch die Ausgangsseite können die Pipeline anhalten. Ein Stoppen von der Ausgangsseite aus bewirkt zugleich ein Stoppen der Datenquelle, aus der die Pipeline gespeist wird.

Fall 4 hat zwar den Vorteil, daß er gleichzeitig eine Pufferfunktion zur Verfügung stellt. Jedoch dürfte es vom Designprinzip her sinnvoller sein, die Pufferproblematik dort zu behandeln, wo sie entsteht, als sie auf andere Teile des Designs auszudehnen. An der Stelle, an der die Notwendigkeit einer Pufferfunktion begründet liegt, ist auch am ehesten eine Abschätzung über dessen erforderliche Größe möglich.

9.6.9 Unterstützung von Pipeline-Kontrollern in *CHDL*

CHDL stellt mit den Anweisungen `FORK`, `FORKIF` und `FLOWCONTROL` eine spezielle Unterstützung zur Realisierung von Pipeline-Kontrollern zur Verfügung:

Mit `FORK` bzw. `FORKIF` lassen sich jederzeit neue zusätzliche Ausführungstokens erzeugen. Diese neuen Tokens können sich alle im selben Ausführungspfad befinden, solange sie nicht kollidieren. Auf diese Weise entstehen zahlreiche Threads, die alle Variablen in diesem Ablaufpfad gemeinsam verwenden. Die aktiven Ausführungstokens werden mit jedem Takt um eine Anweisung weitergeschoben und am Ende des Pfades mittels `END` gelöscht. Anweisungen werden nur ausgeführt, wenn sich an ihrer Position ein aktives Token befindet.

Dieses Verhalten entspricht genau dem einer Pipeline-Anordnung: Mit jedem Takt können Daten in die Pipeline eintreten. Sind in einem Taktzyklus keine Daten verfügbar, wird ein "Loch" in Form eines nicht aktiven Ausführungstokens eingefügt. Nach kompletter Bearbeitung tritt das Ergebnisdatum aus der Pipeline aus.

Das Einfügen von nicht aktiven Ausführungstokens ist problematisch, wenn eine Stufe der Pipeline Daten einer vorangehenden oder nachfolgenden Stufe benötigt. Zu diesem Zweck ist es möglich, die Zustandsvariablen in Bedingungen einzusetzen, um Wartezustände zu implementieren.

Können die Daten am Ausgang der Pipeline nicht abgenommen werden, ermöglicht die Anweisung `FLOWCONTROL` ein Stoppen der Ausführungstokens. Die Anweisungen der Pipeline werden in diesem Fall nur ausgeführt, wenn das entsprechende Ausführungstoken und das spezifizierte Kontrollsignal aktiv sind.

9.7 Anwendung der Hardwarebeschreibung

9.7.1 Allgemeines

In diesem Abschnitt werden einige Anwendungsbeispiele für die C++-basierte Hardwarebeschreibung mit *CHDL* vorgestellt.

Zunächst erfolgt eine Einführung in die wichtigsten Grundelemente: Pads, Speicherelemente, arithmetische Funktionen, Single- und Dual-Port-RAM, Multiplexer und Vergleicher. Die Beschreibung enthält jeweils eine grafische Darstellung und das entsprechende *CHDL*-Interface mit den wesentlichen Elementen.

Danach folgen einige strukturelle Schaltungen: Ein Binärzahler, ein Signalflankendetektor, ein Parity-Generator sowie eine einfache Recheneinheit (ALU).

Weiterhin werden die Einsatzmöglichkeiten von Vererbung und Polymorphismus demonstriert.

Abschließend folgen einige Implementierungsbeispiele für Zustandsmaschinen.

9.7.2 *CHDL*-Grundelemente

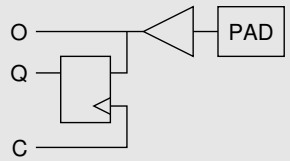
Pads

Die Pad-Objekte (Abb. 9.15) stellen das Interface zwischen dem FPGA-Design und den Gehäusepins (*Pads*) dar. Es existieren Eingangs-, Ausgangs- und bidirektionale Pads, die dementsprechend Eingangs- bzw. Ausgangstreiber sowie ein optionales Flip-Flop enthalten. Bei den Ausgangs-Pads ist nur einer der Eingänge `I` und `D` nutzbar, bei den Eingangs-Pads können beide Signale `O` und `Q` genutzt werden.

Die `Lock`-Funktion kann verwendet werden, um den Pads eindeutige Gehäusepins zuzuweisen.

```
class PadIn
{
 IntOutPin O;
 IntOutPin Q;
 IntInPin  C;

 PadIn ( const char* Name );
 Lock ( uint value );
};
```




```
class PadOut
{
 IntInPin  I;
 IntInPin  D;
 IntInPin  C;

 PadOut ( const char* Name );
 Lock ( uint value );
};
```

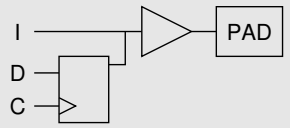


Abbildung 9.15: Eingangs- und Ausgangs-Pads

Speicherelemente (D-Flip-Flops und D-Latches)

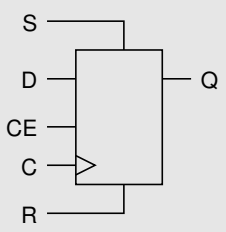
Die D-Flip-Flops (Abb. 9.16) besitzen neben den Pins `D`, `C` und `Q` noch optionale Pins `CE` (Clock-Enable), `S` (asynchrones Setzen) und `R` (asynchrones Zurücksetzen).

Bei den D-Latches (Abb. 9.16) existieren statt der Pins `C` und `CE` entsprechende Pins `G` und `GE`.

Die `Init`-Funktion kann genutzt werden, um Speicherelementen einen bestimmten Startzustand zuzuweisen. Erfolgt keine Zuweisung, wird der Startzustand "0" implementiert.

```
class DFF
{
  IntInPin  D;
  IntInPin  CE;
  IntInPin  C;
  IntInPin  R;
  IntInPin  S;
  IntOutPin Q;

  DFF ( const char* Name );
  Init ( uint Value );
};
```




```
class DLATCH
{
  IntInPin  D;
  IntInPin  GE;
  IntInPin  G;
  IntInPin  R;
  IntInPin  S;
  IntOutPin Q;

  DLATCH ( const char* Name );
  Init ( uint Value );
};
```

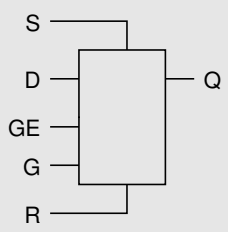


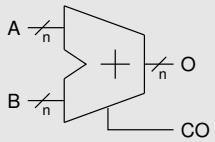
Abbildung 9.16: D-Flip-Flops und D-Latches

Addierer und Subtrahierer

Die Addierer- und Subtrahierer-Elemente (Abb. 9.17) können in beliebigen Bitbreiten erzeugt werden. Sie führen eine Addition bzw. Subtraktion von vorzeichenlosen Binärwerten durch.

```
class ADDER
{
 IntInPinArray  A;
 IntInPinArray  B;
 IntOutPinArray O;
 IntOutPin      CO;

 ADDER ( uint Width, const char* Name );
};
```




```
class SUBTRACT
{
 IntInPinArray  A;
 IntInPinArray  B;
 IntOutPinArray O;
 IntOutPin      CO;

 SUBTRACT ( uint Width, const char* Name );
};
```

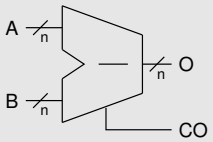


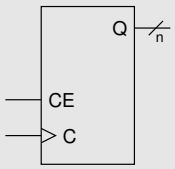
Abbildung 9.17: n-Bit Addierer / Subtrahierer

Zähler

Die binären Aufwärts- und Abwärts-Zähler (Abb. 9.18) können in beliebigen Bitbreiten erzeugt werden. Sie verfügen über einen Clock-Enable-Eingang `CE`.

```
class COUNTUP
{
 IntInPin       CE;
 IntInPin       C;
 IntOutPinArray Q;

 COUNTUP ( uint Width, const char* Name );
};
```




```
class COUNTDN
{
 IntInPin       CE;
 IntInPin       C;
 IntOutPinArray Q;

 COUNTDN ( uint Width, const char* Name );
};
```

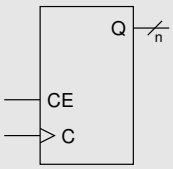


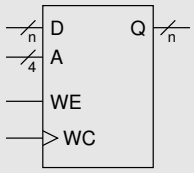
Abbildung 9.18: n-Bit binärer Aufwärts-/Abwärtszähler

Speicherelemente

Die Single- und Dual-Port-Speicher (Abb. 9.19) besitzen vier Adressleitungen. Die Datenbreite kann beliebig gewählt werden. Das Schreiben in den Speicher erfolgt immer synchron zu `WC`, das Auslesen verläuft asynchron.

```
class RAM16
{
 IntInPinArray  D;
 IntInPinArray  WAD;
 IntInPinArray  WE;
 IntInPinArray  WC;
 IntOutPinArray Q;

 RAMD16 ( uint Width, const char* Name );
};
```




```
class RAMD16
{
 IntInPinArray  D;
 IntInPinArray  WAD;
 IntInPinArray  RAD;
 IntInPinArray  WE;
 IntInPinArray  WC;
 IntOutPinArray Q;

 RAMD16 ( uint Width, const char* Name );
};
```

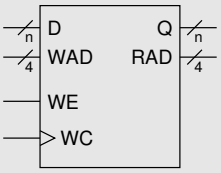


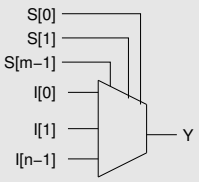
Abbildung 9.19: 16 x n Bit Single- / Dual-Port-Speicher

Multiplexer

Multiplexer (Abb. 9.20) sind in zwei Varianten verfügbar. Das Element `MUX` verwendet ein binär kodiertes Auswahlsignal. Die Variante `DMUX` dagegen besitzt jeweils ein Auswahlsignal für jeden Datenkanal. Hier ist zu beachten, daß zu jedem Zeitpunkt immer nur maximal eines dieser Signale aktiviert sein darf.

```
class MUX
{
 IntInPin2Array I;
 IntInPinArray  S;
 IntOutPinArray Y;

 MUX ( uint Lines,
       uint Width,
       const char* name );
};
```




```
class DMUX
{
 IntInPin2Array I;
 IntInPinArray  S;
 IntOutPinArray Y;

 DMUX ( uint Lines,
        uint Width,
        const char* name );
};
```

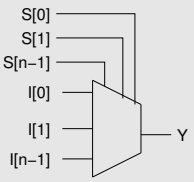


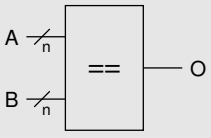
Abbildung 9.20: n-Bit Multiplexer

Vergleicher

Vergleicher (Abb. 9.21) existieren in zwei Varianten. Das Element `COMP` vergleicht zwei Signalvektoren, während `EQUAL` einen Signalvektor mit einer Konstante vergleicht.

```
class COMP
{
 IntInPinArray  A;
 IntInPinArray  B;
 IntOutPin      O;

 COMP ( uint Width, const char* Name );
};
```




```
class EQUAL
{
 IntInPinArray  A;
 IntOutPin      O;

 COMP ( uint Width, uint Value,
        const char* Name );
};
```

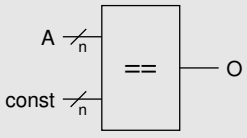


Abbildung 9.21: n-Bit Vergleicher

9.7.3 Strukturelle Schaltungen

Binärzähler

Der folgende Code implementiert einen binären 16-Bit Aufwärtszähler und führt die Signale an die Gehäusepins 10 bis 25. Der Takteingang ist mit Pin 1 verbunden.

```
PadIn       CLK("CLK");
PadOutArray Q(16,"Q");

CLK.Lock(1);

int  Q_LockValues[] = { 10,11,12,13,14,15,16,17,
                        18,19,20,21,22,23,24,25 };
Q.Lock(Q_LockValues);

COUNTUP  Cnt(16,"Cnt",CLK);

Q = Cnt;
```

Digitaler Flankendetektor

Dieses Code-Beispiel realisiert einen digitalen Flankendetektor in einem eigenen Anwendermodul. Beim Auftreten eines Low-High Übergangs wird das Ausgangssignal für die Dauer von einem Takt aktiviert.

```
class RisingEdgeDetect : public BaseUserPart
{
public:
  PinIn  IN;
  PinOut OUT;
  PinIn  CLK;

  RisingEdgeDetect ( const char* Name );
};

RisingEdgeDetect::RisingEdgeDetect ( const char* Name )
  : BaseUserPart(Name),
    IN("IN"),
    OUT("OUT"),
    CLK("CLK")
{
  DFF  IN1("IN1",CLK);

  IN1 = IN;
  OUT = IN & !IN1;
}
```

Parity-Generator

Dieses Beispiel demonstriert den Einsatz der Parametrisierungsmöglichkeiten. Der Parameter `Width` wird bereits im Interface eingesetzt, um die Breite des Eingangsvektors `IN` zu spezifizieren. Innerhalb des Moduls legt er die Weite des XOR-Gatters fest.

```
class ParityGen : public BaseUserPart
{
public:
```

```
  PinInArray IN;
  PinOut     PAR;

  ParityGen ( const char* Name );
};

ParityGen::ParityGen ( int Width, const char* Name )
  : BaseUserPart(Name),
    IN(Width,"IN"),
    PAR("PAR")
{
  XOR  Xor(Width,"Xor");

  Xor.I = IN;
  PAR = Xor;
}
```

Einfache ALU

Dieses Code-Beispiel implementiert eine einfache ALU. Abhängig von den Signalen an `SEL` werden folgende Operationen ausgeführt:

SEL	Bezeichnung	Operation
0	Null	0
1	Eins	1
2	A	A
3	B	B
4	Addition	$A + B$
5	Subtraktion	$A - B$
6	Negierung	$-A$
7	isNull	$A == 0$

```
class ALU : public BaseUserPart
{
public:
  PinInArray  A;
  PinInArray  B;
  PinInArray  SEL;
  PinOutArray O;

  ParityGen ( const char* Name );
};

ALU::ALU ( int Width, const char* Name )
  : BaseUserPart(Name),
    A(Width,"A"),
    B(Width,"B"),
    SEL(3,"SEL"),
    O(Width,"O")
```

```
{
  ADDER  Add(Width,"Add");
  SUB    Sub(Width,"Sub");
  NEG    Neg(Width,"Neg");
  EQUAL  Equal0(Width,0,"Equal0");
  DMUX   Mux(8,Width,"Mux");

  Add.A  = A;
  Add.B  = B;
  Sub.A  = A;
  Sub.B  = B;
  Neg    = A;
  Equal0 = A;

  Mux.I[0] = 0;
  Mux.I[1] = 1;
  Mux.I[2] = A;
  Mux.I[3] = B;
  Mux.I[4] = Add;
  Mux.I[5] = Sub;
  Mux.I[6] = Neg;
  Mux.I[7][0] = Equal0;
  Mux.I[7][Bus(1,Width-1)] = 0;

  for (int i = 0; i < 8; i++)
     Mux.S[i] = (SEL == i);

  O = Mux;
}
```

9.7.4 Vererbung und Polymorphismus

Vererbung und Polymorphismus sind zwei sehr mächtige Konzepte der Programmiersprache C++. Beim *CHDL*-System können diese auch innerhalb der Hardwarebeschreibung verwendet werden.

Wird eine Bauteilklasse von einer anderen abgeleitet, erbt sie deren Eigenschaften. Die Vererbung umfaßt zum einen die Klassenattribute, also die Interface-Pins, und zum anderen die Methoden der Klasse.

Sinnvoll einsetzbar ist dieses Konzept immer, wenn mehrere Bauteile über viele Gemeinsamkeiten verfügen. Die gemeinsam vorhandenen Eigenschaften werden dann in der Basisklasse implementiert. Die abgeleiteten Bauteile implementieren nur noch die jeweils fehlende Funktionalität.

Das folgende Beispiel zeigt die Implementierung von zwei Bauteilen, die über eine gemeinsame Basisklasse verfügen. Gemeinsam genutzt sind die Pins `D`, `Q0` und `CLK`, weiterhin die Implementierung einer Verzögerungsstufe mit `FF1`.

Bauteil `Part_A` ergänzt die Basisfunktionalität mit einem weiteren Ausgangspin (`Q1`) und einer weiteren Verzögerungsstufe (`FF2`).

Bauteil `Part_B` dagegen fügt einen weiteren Pin (`NQ0`) sowie eine Invertierung hinzu.

```
class  CommonPart : public BaseUserPart
{
public:
  PinIn  D;
  PinIn  CLK;
  PinOut Q0;
```

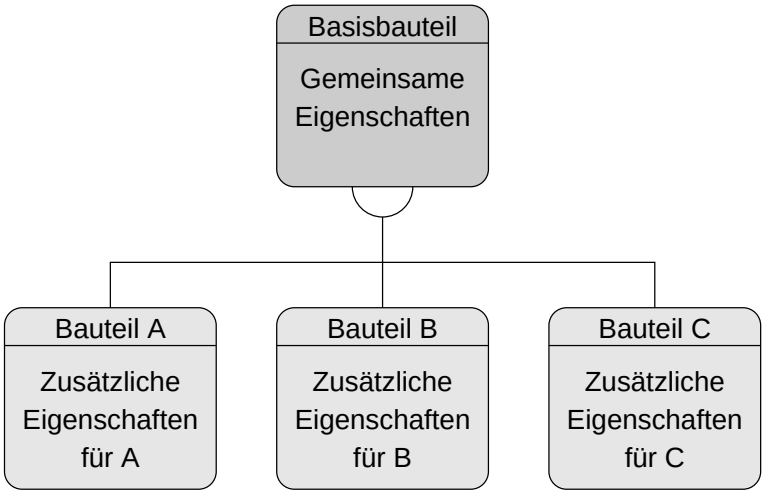


Abbildung 9.22: Vererbung bei *CHDL*-Bauteilen

```
  CommonPart ( const char* Name );
};

CommonPart::CommonPart ( const char* Name )
  : BaseUserPart(Name),
    D("D"),
    CLK("CLK"),
    Q0("Q0")
{
  ProcessName _N(this);

  DFF  FF1("FF1",CLK);
  FF1 = D;
  Q0  = FF1;
}

class Part_A : public CommonPart
{
public:
  PinOut Q1;

  Part_A ( const char* Name );
};

Part_A::Part_A ( const char* Name )
  : CommonPart(Name),
    Q1("Q1")
{
  ProcessName _N(this);

  DFF  FF2("FF2",CLK);
  FF2 = Q0;
  Q1  = FF2;
}
```

```
class Part_B : public CommonPart
{
public:
  PinOut NQ;

  Part_B ( const char* Name );
};

Part_B::Part_B ( const char* Name )
  : CommonPart(Name),
    NQ("NQ")
{
  ProcessName _N(this);

  NQ = !Q0;
}
```

Zu beachten ist dabei, daß die Hierarchieebenen zur Namensbildung der Bauteile in der Basisklasse und den abgeleiteten Klassen gleich sind. Das hat zur Folge, daß keine gleichnamigen Bauteile innerhalb der Konstruktoren zulässig sind. Vom C++-Kompiler wird dies zwar akzeptiert, das *CHDL*-Laufzeitsystem wird diesen Fall jedoch als Fehler behandeln.

Eine abgeleitete Bauteilklasse kann nicht nur Funktionalität hinzufügen, sondern auch Teile der Basisklasse durch neue ersetzen. Möglich ist dies durch Einsatz virtueller Methoden. Interface-Pins der Basisklasse können nicht ersetzt werden.

Das folgende Code-Beispiel demonstriert dieses Verfahren. Die Implementierung des Basiselements erfolgt hier nicht direkt im Konstruktor, sondern mittels einer eigenen Methode (`Build`). Diese wird nach dem Anlegen des Bauteils explizit aufgerufen.

```
class  CommonPart : public BaseUserPart
{
public:
  PinIn  D;
  PinIn  Q0;
  PinIn  CLK;

  CommonPart ( const char* Name );

  void  Build ( void );
  virtual void  IntBuild ( void );
};

CommonPart::CommonPart ( const char* Name )
  : BaseUserPart(Name),
    D("D"),
    CLK("CLK"),
    Q0("Q0")
{ }

void  CommonPart::Build ( void )
{
  ProcessName _N(this);
  ...
  IntBuild();
```

```
  ...
}

void  CommonPart::IntBuild ( void )
{
  DFF  FF1("FF1",CLK);
  FF1 = D;
  Q0  = FF1;
}
```

Das abgeleitete Bauteil `Part_A` fügt einen weiteren Pin `Q1` hinzu und modifiziert gleichzeitig die Basisimplementierung, wodurch jetzt zwei Verzögerungsstufen realisiert werden.

```
class Part_A : public CommonPart
{
public:
  PinOut Q1;

  Part_A ( const char* Name );

  void  IntBuild ( void );
};

Part_A::Part_A ( const char* Name )
  : CommonPart(Name),
    Q1("Q1")
{ }

void  Part_A::IntBuild ( void )
{
  DFF  FF1("FF1",CLK);
  DFF  FF2("FF2",CLK);
  FF1 = D;
  FF2 = FF1;
  Q0  = FF2;
}
```

Durch den Mechanismus der virtuellen Methoden wird nun im folgenden Code nicht mehr `IntBuild` der Basisklasse, sondern der abgeleiteten Klasse ausgeführt.

```
Part_A  A("A");
A.Build();
```

Zu beachten ist dabei, daß der Aufruf virtueller Methoden aus einem Konstruktor heraus nicht den oben beschriebenen Effekt hätte. Daher ist der explizite Aufruf der Funktion `Build` erforderlich. Die *virtual table* eines Objektes wird erst am Ende des Konstruktors auf die virtuellen Methoden der aktuellen Klasse gelegt. Ein Aufruf einer solchen Methode im Konstruktor hätte die Ausführung der entsprechenden Methode der nächsten Basisklasse zur Folge.

9.7.5 Zustandsmaschinen

Einfacher Dezimalzähler

Die folgende Zustandsmaschinenimplementierung stellt einen Dezimalzähler dar. Die erzeugte Zahlenfolge beginnt mit "0". Ist der Wert "9" erreicht, wird er nicht weiter inkrementiert, sondern zurück auf den Startwert "0" gesetzt.

```
class DecCounter : public BaseSM
{
public:
  SMDRegOut  Q;

  DecCounter ( const char* Name );
};

DecCounter::DecCounter ( const char* Name )
  : BaseSM(Name),
    Q(4,"Q")
{
  ProcessName _N(this);

LABEL("START");
  BeginState();         // Initialisierung: Q = 0
    Q = 0;
  EndState();

LABEL("LOOP");
  IF (Q == 9,"START");  // auf '9' folgt '0'

  BeginState();
    Q = Q + 1;          // nächster Wert
  EndState();

  GOTO ("LOOP");        // zurück in die Schleife
}
```

Summenberechnung

Diese Zustandsmaschine berechnet die Summe aller Zahlen von 1 bis zu einem übergebenen Wert. Die Berechnung wird über das externe Signal START gestartet. Das Signal RDY wird aktiviert, sobald die Berechnung komplett ist. Die Summe ist dann solange an S verfügbar, bis das START-Signal wieder inaktiviert wird.

```
class ComputeSUM : public BaseSM
{
public:
  PinInArray N;
  PinIn      START;
  SMDRegOut  S;
  SMDFFOut   READY;

  ComputeSUM ( const char* Name );
};

ComputeSUM::ComputeSUM ( const char* Name )
  : BaseSM(Name),
    N(4,"N"),
    START("START"),
    S(8,"S"),
    READY("READY")
{
  ProcessName _N(this);
```

```
    SMDReg  SUM(8,"SUM");  // lokale Variablen
    SMDReg  I(4,"I");

  LABEL("WAIT");
    BeginState();
      SUM   = 0;           // Warten, bis START = 1
      I     = 0;
      READY = 0;
    EndState();

    IF (!START,"WAIT");

  LABEL("LOOP");
    IF (I == N,"READY");   // Endwert erreicht ?

    BeginState();
      SUM = SUM + I;       // aktuellen Wert addieren
      I = I + 1;           // nächster Wert
    EndState();

    GOTO ("LOOP");         // zurück in die Schleife

  LABEL("READY");
    BeginState();
      S     = SUM;         // Summe ausgeben
      READY = 1;           // READY-Flag setzen
    EndState();

    IF (START,"READY,"WAIT"); // Warten, bis Summe gelesen
  }
```

9.8 Vergleich der *CHDL*-Beschreibung zu anderen Systemen

In den letzten Abschnitten wurden die Möglichkeiten der *CHDL*-Hardwarebeschreibung ausführlich dargestellt.

Diese soll nun am Beispiel eines 1-Bit Volladdierers mit verschiedenen existierenden Systemen verglichen und diskutiert werden. Das Addierer-Beispiel soll dabei nur die Syntax demonstrieren. In einem realen Design könnte in jedem System ein vordefiniertes Addierer-Element verwendet werden.

Es werden hier nur strukturelle Beschreibungen verglichen, daher sind *SystemC* sowie *Handel-C* nicht aufgeführt.

9.8.1 *SL - Structured Design Language*

```
ADD1 ( IN cin, a, b; OUT cout, sum):
  VAR temp;
  X1 = XOR2(temp, a, b);
  X2 = XOR2(sum, cin, temp);
  M1 = M2_1(cout, cin, b, temp);
```

Das Prinzip, mit dem bei *SL* [42] Verbindungen zwischen den Bauteilen hergestellt werden, hat einen prozeduralen Charakter. Es werden keine objektorientierten Methoden eingesetzt. Die Pins der Gatter werden nicht durch Pin-Objekte, sondern durch die Positionen in der Parameterliste repräsentiert. So ist in der ersten `XOR2`-Anweisung `temp` der Ausgang des

Gatters, `a` und `b` sind die Eingänge. Diese Methode ist deutlich unübersichtlicher und erfordert mehr Schreibaufwand als die Angabe von zwei logischen Ausdrücken `sum = a ^ b ^ cin` und `cout = (a&b)|(b&cin)|(b&cin)`, die die gleiche Funktion erfüllen.

Es handelt sich bei *SL* um eine proprietäre Sprache, nicht um eine universelle Programmiersprache.

9.8.2 *Pebble*

```
BLOCK hadd [a,b:WIRE] [cout,sum:WIRE]
BEGIN
  xor2 [a,b] [sum];
  and2 [a,b] [cout];
END;
```

Pebble [54] ist in seiner Struktur vergleichbar zu *SL*. Die Verbindungen werden ebenfalls auf prozedurale Weise erstellt. Die Beschreibung ist allerdings kompakter und die Ein- und Ausgänge sind leichter erkennbar.

9.8.3 Codegenerator nach *Chu/Weaver/Sulimma*

```
public class FullAdder extends GenComponent
{
  public InputWire     a,b,cin;
  public OutputWire    cout,sum;
  public LogicFunction summation;
  public LogicFunction carry;

  public FullAdder()
  {
    summation = new LogicFunction("a^b^cin");
    carry = new LogicFunction("(a&b)|(a&cin)"
                            +"|(b&cin)");
    cout = carry.o;
    sum  = summation.o;
  }
  public void attachWires()
  {
    summation.o = sum;
    carry.o     = cout;
  }
}
```

Der Codegenerator nach *Chu/Weaver/Sulimma* [23] verwendet *JAVA* als universelle Programmiersprache. Es benötigt die Deklaration der lokalen Funktionen `summation` und `carry` im public-Interface. Diese werden wiederum von der zusätzlichen `attachWires` Funktion verwendet, was streng genommen eine Verletzung des objektorientierten Prinzip der Kapselung darstellt.

Logische Ausdrücke werden mittels `LogicFunction()` implementiert. Diese Art der Auswertung ist hier möglich, da auf das Metadaten-Interface von *JAVA* zurückgegriffen werden kann. Sie ist jedoch nicht sehr anwenderfreundlich. Es wäre angenehmer, die Ausdrücke direkt als Programmcode formulieren zu können.

Elemente, die Busse verwenden, benötigen eine aufwendige Methode mit expliziten `for`-Schleifen, um die entsprechenden Verbindungen zu realisieren [23].

9.8.4 *JHDL*

```
public class FullAdder extends Logic
{
  public FullAdder ( Node parent,
                     Wire a,
                     Wire b,
                     Wire cin,
                     Wire sum,
                     Wire cout )
  { super(parent);
    connect("a", a);
    connect("b", b);
    connect("cin", cin);
    connect("sum", sum);
    connect("cout", cout);

    or_o(and(a,b),and(a,cin),and(b,cin),cout);
    xor_o(a,b,cin,sum);
  }
}
```

Die strukturelle Beschreibung von *JHDL* [9] weist Ähnlichkeiten zu *CHDL* auf. Die Verbindung der Elemente zeigt jedoch wiederum einen prozeduralen Charakter wie bei *SL* und *Pebble*. Die objektorientierten Prinzipien sind nicht konsequent realisiert. Obwohl JHDL ebenfalls wie der zuvor erörterte Codegenerator nach *Chu/Weaver/Sulimma* eine *JAVA*-Implementierung ist, werden in der `connect`-Funktion die Objektnamen als zusätzliche Parameter übergeben.

9.8.5 *PamDC*

```
class FullAdder : public Node
{
public:
  FullAdder() : Node("FullAdder") {}
  void logic (Bool& a, Bool& b, Bool& cin,
              Bool& sum, Bool& cout)
  { input(a);
    input(b);
    input(cin);
    output(sum);
    output(cout);

    sum = a ^ b ^ cin;
    cout = (a & b) | (b & cin) | (cin & a);
  }
};
```

Dieses C++-basierte System [98] zeigt ebenfalls strukturelle Ähnlichkeiten mit *CHDL*. Die Hardwarebeschreibung benutzt jedoch keine Implementierung im Konstruktor, sondern eine zusätzliche `logic` Funktion. Logische Gleichungen können wie bei *CHDL* direkt angegeben werden. Auch diese Sprache benötigt die explizite Angabe des Objektnamens (hier: `FullAdder`) als zusätzlichen Parameter. Wie bei *JHDL* werden die Ein- und Ausgangspins nicht als Pin-Objekte, sondern als Parameter dargestellt, was dem Erstellen von Verbindungen wiederum einen prozeduralen Charakter verleiht. Die Sprache macht keinen intensiven Gebrauch von objektorientierten Methoden.

9.8.6 *CHDL*

```
class FullAdder : public BaseUserPart
{
public:
  PinIn  a,b,cin;
  PinOut cout,sum;

  FullAdder ( const char* Name )
    : BaseUserPart(Name),
      a("a"),
      b("b"),
      cin("cin"),
      cout("cout"),
      sum("sum")
  {
    ProcessName _N(this);

    sum  = a ^ b ^ cin;
    cout = (a & b) | (a & cin) | (b & cin);
  }
};
```

CHDL folgt einem strengeren objektorientierten Ansatz als die zuvor beschriebenen Systeme. Pins sind Teile von Bauelementen, keine Parameter. Aufgrund der C++-Implementierung kann die zusätzliche Angabe des Objektnamens bei Pins und Bauelementen nicht vermieden werden, wenn diese Namen zur Laufzeit bekannt sein sollen.

9.9 Zusammenfassung

Im vorangegangenen Kapitel wurde zunächst erläutert, weshalb bei *CHDL* die Programmiersprache C++ als Basis für die Hardwarebeschreibung gewählt wurde und nicht eine der anderen wie etwa *JAVA*.

Dabei waren folgende Kriterien entscheidend:

- C++ ist die üblicherweise bei FPGA-Koprozessoren eingesetzte Sprache für das Erstellen der Softwarekomponente. Die gleichzeitige Verwendung von C++ zur Hardwarebeschreibung bietet zahlreiche Vorteile. So können etwa bereits vorhandene und dem Entwickler vertraute C++-Entwicklungsumgebungen und Source-Level-Debugger eingesetzt werden.

- Die oft im Zusammenhang mit C++ genannten Probleme wie das Fehlen eines Metadaten-Interfaces oder die Plattformabhängigkeit können mit vertretbaren Mitteln bewältigt werden. Die notwendigen Maßnahmen verursachen für den Anwender der Hardwarebeschreibung nur einen minimalen Mehraufwand und bedeuten keine Einschränkung in der Flexibilität.

- C++ bietet gegenüber *JAVA* einige Vorteile, die sich sowohl durch eine kompaktere Hardwarebeschreibung als auch durch höhere Ausführungsgeschwindigkeiten auswirken. So verfügt *JAVA* nicht über die Möglichkeit, Operatoren zu überladen. Gerade diese Überladung ist für eine kompakte Darstellung von Schaltfunktionen erforderlich. *JAVA* wurde in erster Linie auf Plattformunabhängigkeit entwickelt, die verfügbaren Kompiler erreichen nicht die hohen Ausführungsgeschwindigkeiten der C++-Werkzeuge. Weiterhin bietet C++ effiziente DLL-Schnittstellen zur Einbindung dynamischer Bibliotheken.

Die nachfolgenden Abschnitte diskutierten die Notwendigkeit mehrerer Abstraktionsebenen sowie die mit C++ realisierbaren Ausführungsmodelle.

Es wurde dargestellt, welche Mechanismen implementiert werden müssen, um eine Hardwarebeschreibung mit unverändertem C++ und handelsüblichen Kompilern zu ermöglichen. Dabei wurde das Ziel verfolgt, den späteren Anwender möglichst wenig mit diesen Mechanismen zu belasten.

Nach der Implementierung aller notwendigen Maßnahmen konnten zunächst zwei Ausführungsmodelle auf verschiedenen Abstraktionsebenen implementiert werden:

- Eine strukturelle Hardwarebeschreibung.
- Eine Methode zur Beschreibung von Zustandsmaschinen mittels Flußdiagrammen.

Die strukturelle Hardwarebeschreibung ähnelt der Hardwarebeschreibungssprache *ABEL* und ermöglicht kompakte Beschreibungen mit direkter Formulierung von Schaltfunktionen. Sie erlaubt eine detaillierte Kontrolle über die FPGA-Resourcen und damit die Erstellung effizienter Schaltungen. Weiterhin enthält sie architekturunabhängige Versionen von Elementen wie etwa D-Flip-Flops oder I/O-Pads, um soweit wie möglich portable Beschreibungen zu unterstützen.

Die strukturelle Ebene weist zunächst einen niedrigen Abstraktionsgrad auf. In Verbindung mit den üblichen C/C++-Mechanismen wie Bedingungen, Schleifen oder Parametrisierung kann der Abstraktionsgrad jedoch deutlich erhöht werden.

Die Beschreibung ermöglicht die Erstellung neuer Klassen, mit denen eine Modularisierung bei umfangreichen Designs vorgenommen werden kann. Dies unterstützt auf flexible Art den Aufbau komplexer Bibliotheken, auch die Wiederverwendbarkeit einmal erstellter Bauteile wird damit erleichtert.

Die Beschreibung von Zustandsmaschinen ermöglicht die Implementierung von sequentiellen Kontrollern. Bedingt durch die Realisierung mittels C++ ist diese Form der Beschreibung etwas umfangreicher als bei anderen Hardwarebeschreibungssprachen. Der Vorteil besteht jedoch darin, daß auch hier zur Synthese kein spezieller Kompiler erforderlich ist. Die Zustandsmaschinenbeschreibung kann in normalen C++-Code eingebettet werden.

Das Verfahren des modifizierten *One-Hot*-Encodings stellt ein flexibles Ausführungsmodell dar, in dem sich auch komplexe nebenläufige Ablaufpfade realisieren lassen.

Dabei bleibt das Verfahren, nachdem die Synthese dieser Zustandsmaschinen erfolgt, stets nachvollziehbar. Dies kann insbesondere für die Abschätzung des Zeitverhaltens von großer Bedeutung sein.

Weiterhin wurde erläutert, wie die objektorientierten Konzepte der Datenkapselung, der Vererbung und des Polymorphismus in Hardwarebeschreibungen genutzt werden können. Zusammen mit der Fähigkeit zur Modularisierung können auf diese Weise leistungsfähige erweiterbare Anwenderbibliotheken aufgebaut werden.

In einem Vergleich mit anderen Systemen wurde das konsequente objektorientierte Design von *CHDL* sowie die kompakte und übersichtliche Form der Hardwarebeschreibung verdeutlicht. Es wurde gezeigt, daß die Methode, die Pins von Bauteilen nicht über Prozedurparameter, sondern über direkte Zuweisungen miteinander zu verbinden, zu einer deutlich übersichtlicheren Schreibweise führt.

Die Code-Beispiele demonstrieren weiterhin, daß die anderen Systeme deutlich weniger Unterstützung für die Erstellung umfangreicher und komplexer Hardwarebeschreibungen bieten. Dies ist im wesentlichen darin begründet, daß sie die objektorientierten Konzepte nicht so konsequent realisieren wie *CHDL*.

Damit verfügt *CHDL* bereits auf der strukturellen Ebene der Entwurfseingabe über zahlreiche Vorteile gegenüber anderen Entwicklungssystemen.

Auf der strukturellen Ebene aufbauende Verfahren zur Beschreibung von Zustandsmaschinen sind bei den anderen Systemen nicht vorhanden.

Kapitel 10
Simulation

10.1 Einsatz von C++ zur Simulation von Hardwarebeschreibungen

Im vorigen Kapitel wurden die Möglichkeiten der Hardwarebeschreibung mittels C++ diskutiert. Es wurde deutlich, daß dies in erster Linie durch den Aufbau statischer Strukturen erfolgt. Die Bedeutung der Anweisungen weicht hier deutlich vom üblichen C++-Ausführungsmodell ab.

Soll nun C++ nicht nur zur Hardwarebeschreibung, sondern auch zu deren Simulation eingesetzt werden, muß mit dieser Sprache auch das Verhalten der einzelnen Elemente, wie etwa Gatter und Flip-Flops, implementiert werden können.

Betrachtet man die Verhaltensbeschreibung zur Simulation eines Bauteils am Beispiel eines UND-Gatters, so ist erkennbar, daß das konventionelle Ausführungsmodell von C++ dazu direkt verwendet werden kann:

```
if ((Input1 == 1) && (Input2 == 1))
    Output = 1;
  else
    Output = 0;
```

Die Problematik der bedingten Anweisungen stellt sich hier nicht, da in einem Simulationsdurchgang nur der gerade aktuelle Pfad zur Auswertung durchlaufen werden muß.

Da C++ eine vollständige universelle Programmiersprache darstellt, läßt sich das Verhalten beliebig komplexer Bauteile auf diese Weise beschreiben.

In einer Gesamtanordnung mehrerer solcher Beschreibungen muß sichergestellt sein, daß diese auf effiziente Weise zusammenwirken. Der oben gezeigte Beispielcode für ein UND-Gatter muß zur fortlaufenden Simulation periodisch ausgeführt werden. Beim Simulieren vieler Gatter kann dies erhebliche Rechenzeiten erfordern. Es muß also ein Mechanismus existieren, der die Ausführung der einzelnen Simulationsprozeduren nur dann anstößt, wenn dies erforderlich ist. Nur so können vertretbare Simulationszeiten erreicht werden. Das hierzu bei *CHDL* eingesetzte Verfahren wird im nächsten Abschnitt vorgestellt.

Jedes synthetisierbare Grundelement erhält seine eigene simulierbare Verhaltensbeschreibung. Damit kann jede beliebige Schaltung, die aus diesen Elementen zusammengesetzt wird, ebenfalls simuliert werden.

Zusätzlich können auch Elemente existieren, die nur eine Simulationsbeschreibung, aber keine synthetisierbare Hardwarebeschreibung enthalten. Dies ist sinnvoll, um externe Bausteine der FPGA-Umgebung in die Simulation einzubeziehen, so etwa Speicherbausteine, Taktgeneratoren oder Bus-Bridges.

Auch die aus dem *VHDL*-Bereich bekannten Simulationsmethoden der Testvektoren und Testbenches lassen sich mit solchen C++-Funktionen realisieren.

Die Verhaltensbeschreibung zur Simulation ist nicht an die Einschränkungen der synthetisierbaren Hardwarebeschreibung gebunden. Daher sind hier auch weit mehr Anweisungen zulässig, z.B. Einsatz von dynamisch allokiertem Speicher oder Dateiein- und -ausgabe.

Durch die Modellierung des Verhaltens mit C++ können selbst hochkomplexe Bausteine, wie etwa Mikroprozessoren, in die Simulation integriert werden.

10.2 Das Ausführungsmodell der *CHDL*-Simulation

Wie oben bereits erwähnt, müssen die einzelnen Simulationsprozeduren auf effiziente Weise zusammenwirken können, um viele parallel arbeitende Elemente zu simulieren.

Ein auf einem Mikroprozessor ablaufender Prozeß kann Anweisungen jedoch nur sequentiell ausführen. Die zu simulierende Parallelität könnte mithilfe von parallelen Prozessen bzw. Threads realisiert werden. Jedoch stellen auch diese keine echte Parallelität dar. Vielmehr wird jeder Prozeß bzw. Thread nach einer von einem Scheduler festgelegten Zeit unterbrochen und die Weiterverarbeitung eines anderen Prozesses/Threads fortgesetzt. Die Verwaltung dieser Prozesse/Threads durch das Betriebssystem verursacht einen Zeitaufwand, der die Effizienz des Gesamtsystems mit zunehmender Anzahl verringert. In einem Mikroprozessorsystem mit wenigen Prozessen ist diese Effizienzeinbuße noch vertretbar.

Soll jedoch auf diese Weise eine Vielzahl von einzelnen Logikelementen simuliert werden, deren Anzahl leicht mehrere Tausend betragen kann, würde dies zur Überlastung des Gesamtsystems führen.

Weiterhin ist der Simulationscode für die meisten dieser Logikelemente sehr einfach, so daß eine Implementierung als jeweils eigener Prozeß/Thread als übertrieben aufwendig erscheint.

Ein effizienteres Verfahren besteht darin, den Simulationsablauf in kleine, nicht mehr weiter unterteilbare Simulationsschritte (*Steps*) zu zerlegen. In jedem Simulationsschritt werden alle notwendigen Bauteile simuliert, indem ihre Simulationsfunktionen ausgeführt werden. Ist sichergestellt, daß die Reihenfolge der Bauteile bei dieser Auswertung keine Rolle spielt, erreicht man eine scheinbare Parallelität innerhalb der Simulationsschritte. Die Auswertungsreihenfolge spielt dann keine Rolle, wenn die einzelnen Ergebnisse die Auswertung der nachfolgenden Bauteile nicht beeinflussen. Dies kann durch ein "Einfrieren" der Signale erfolgen. Erst, wenn alle Bauteile simuliert sind, werden die während dieser Phase berechneten neuen Ausgangssignale übernommen.

Desweiteren kann ein Optimierungsmechanismus sicherstellen, daß in jedem Simulationsschritt nur diejenigen Bauteile simuliert werden, an deren Eingängen Zustandswechsel aufgetreten sind. Dies verkürzt die erforderliche Simulationszeit.

Es existieren jedoch auch Bauteile, deren Verhalten nicht oder nicht ausschließlich von externen Signalen bestimmt wird. So besitzt etwa ein Taktgenerator nur Ausgänge. Für solche Bauteile müssen Zeitpunkte festgelegt werden können, zu denen ihre Simulationsfunktion aktiviert wird. Es wäre ineffizient, diese Funktion in jedem Simulationsschritt auszuführen.

Das folgende Verfahren zur Simulation erfüllt die oben genannten Anforderungen:

1. Initialisierungsphase.

   Alle Bauteile werden mit ihren Startwerten initialisiert. Dies kann für jedes Bauteil separat und unabhängig von den anderen erfolgen, daher ist die Reihenfolge der Bearbeitung nicht relevant. Weiterhin werden alle initialisierten Bauteile zur späteren Bearbeitung in Schritt 3 markiert.

2. Update-Phase.

   Alle Netzzustände werden kopiert. Diese Kopien sind für die nächste Evaluierungsphase relevant. In dieser Phase werden auch die Netzzustände der aktuellen Simulationszeit für die grafische Ausgabe abgespeichert.

3. Evaluierungsphase.

   Die Simulationsfunktionen aller Bauteile, die zur Bearbeitung markiert sind, werden ausgeführt. Sie übernehmen die Berechnung der neuen Ausgangswerte. Dabei werden die Netzzustände des letzten Durchlaufs verwendet. Diese verändern sich während der aktuellen Phase nicht. Dadurch wird erreicht, daß die Reihenfolge der Bearbeitung der einzelnen Bauteile unerheblich ist. Verändert ein Bauteil in dieser Phase die Zustände seiner Ausgänge, so werden alle Bauteile, die an den betreffenden Ausgängen angeschlossen sind, für die Bearbeitung im nächsten Schritt 3 markiert.

4. *Wakeup*-Liste abarbeiten.

Alle Simulationsfunktionen der Bauteile, die auf die aktuelle Simulationszeit programmiert sind (*Wakeup*-Liste), werden ausgeführt. Auch hier ist die konkrete Reihenfolge unerheblich. Verändert ein Bauteil in dieser Phase die Zustände seiner Ausgänge, so werden alle Bauteile, die an den betreffenden Ausgängen angeschlossen sind, für die Bearbeitung im nächsten Schritt 3 markiert.

5. Simulationszeit erhöhen.

   Wurden in den Phasen 3 oder 4 Operationen durchgeführt, wird die Simulationszeit um einen *Step* erhöht. Ansonsten bestimmt sich die aktuelle Simulationszeit aus der frühesten folgenden Zeit der *Wakeup*-Liste.

   Danach wird die Ausführung mit Schritt 2 fortgesetzt, bis der zu simulierende Zeitraum komplett bearbeitet ist.

10.3 Die Implementierung des *CHDL*-Simulators

10.3.1 Simulierte Logikzustände

Es werden nur die Zustände "0", "1" und "Z" (hochohmig) simuliert. Dabei kann der Zustand "Z" nur an den Gehäusepins oder an Ausgängen interner TriState-Buffer auftreten. Auf die Simulation von weiteren Zuständen wie etwa starken bzw. schwachen Logikpegeln oder undefinierten Pegeln wurde bewußt verzichtet, um höhere Simulationsgeschwindigkeiten erreichen zu können.

Diese Einschränkungen sind mit dem internen Aufbau von FPGAs vereinbar. Hochohmige Signale können dort nur an Gehäusepins und TriState-Buffern auftreten. Undefinierte Pegel sowie das Zusammentreffen von starken oder schwachen Pegeln sind in FPGA-Designs nicht zulässig.

Ein Gegeneinandertreiben mehrerer Ausgänge ist im realen Betrieb nur an den Gehäusepins und den TriState-Buffern möglich. Solche Situationen können vom *CHDL*-Simulator, ähnlich wie Setup-Zeit-Verletzungen, erkannt und durch entsprechende Warnungen gemeldet werden. Der Simulator wird nicht versuchen, eine realitätsnahe Simulation solcher Situationen durchzuführen.

10.3.2 Initialisierung der Simulation

Jedes simulierbare Element besitzt eine Funktion `InitEvaluate()`, die zu Beginn der Simulation vom Simulatorkern aufgerufen wird. In dieser Funktion können die Bauteile den Startwert ihrer Ausgänge festlegen sowie interne Variablen initialisieren. Es ist garantiert, daß `InitEvaluate()` für jedes Bauteil genau einmal ausgeführt wird. Bauteile, die die *Wakeup*-Liste verwenden, können hier auch den Zeitpunkt bestimmen, zu dem die Simulationsfunktion `Evaluate()` zum ersten Mal ausgeführt werden soll.

10.3.3 Simulation der Teilschritte

Die gesamte Simulation wird in einzelnen Teilschritten ausgeführt. Jeder Teilschritt umfaßt die Schritte 2 bis 5 des Ablaufes, der oben erläutert wurde.

Es wird eine funktionale Simulation durchgeführt. Kombinatorische Elemente sowie die Gehäusepins und Latches besitzen dabei eine angenommene Durchlaufverzögerung von einem Simulationsschritt. Die Setup- und Clock-to-Output-Zeiten bei flankengesteuerten Speicherelementen betragen ebenfalls einen Schritt.

Jedes simulierbare Element besitzt eine Funktion `Evaluate()`, die die Verhaltenssimulation für einen Teilschritt durchführt. In Abhängigkeit von den aktuellen Eingangssignalen und interner Variablen ermittelt diese Funktion die nächsten Werte der Ausgänge.

Die Simulationsfunktionen werden unter der Kontrolle des Simulatorkerns ausgeführt. Die Aktivierung erfolgt nur für die jeweils markierten Elemente oder diejenigen mit einem aktuellen Eintrag in der *Wakeup*-Liste. An den übrigen Elementen ist keine Veränderung der Eingangssignale aufgetreten bzw. momentan keine Aktion notwendig.

Durch dieses Verfahren sind keine parallelen Simulationsprozesse (Abb. 10.1) notwendig. Stattdessen können innerhalb eines Simulationsschrittes die einzelnen Bauteile in beliebiger Reihenfolge bearbeitet werden (Abb. 10.2).

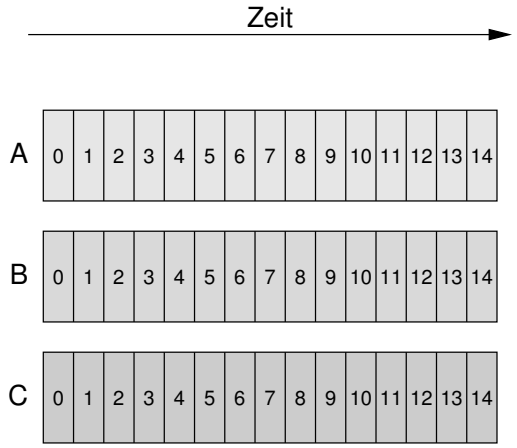


Abbildung 10.1: Simulation mit parallelen Prozessen

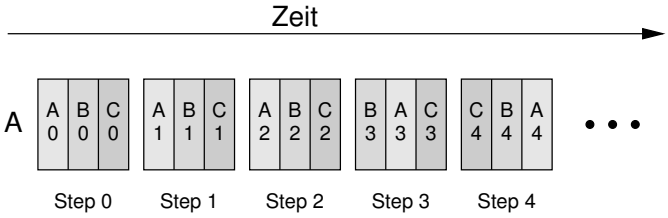


Abbildung 10.2: Eventbasierte Simulation

Abbildung 10.3 zeigt die wesentlichen Datenstrukturen des Simulatorkerns.

Die aktuellen Netzzustände müssen in jedem Durchlauf von Schritt 2 kopiert werden. Daher werden diese nicht innerhalb der Netzobjekte gespeichert, sondern in einem externen zusammenhängenden Bereich. Dies ermöglich ein effizientes Kopieren.

Das Markieren der Bauteile für die Bearbeitung in Schritt 3 erfolgt nicht mithilfe von Flags, sondern durch Aufnahme in eine globale Update-Liste. Für jedes Netz ist eine Liste aller angeschlossenen Bauteile vorhanden. Ändert sich der Zustand eines Netzes, werden alle Bauteile in dieser Liste in die Update-Liste übernommen. Da sich Bauteile in mehreren Netzen befinden können, verhindern entsprechende Flags eine Mehrfachaufnahme.

Der Nachteil dieses Verfahrens besteht darin, daß zwischen dem Aufbau der internen Bauteil-, Pin- und Netzlisten und der eigentlichen Simulation ein Zwischenschritt eingefügt werden muß, der die oben beschriebenen Datenstrukturen aufbaut.

Ändert sich während der Simulation die Struktur der Netzliste, etwa bei einer Rekonfiguration von FPGAs, muß dieser Zwischenschritt wiederholt werden. Dies stellt jedoch kein zeitliches Problem dar, da in der Praxis solche Rekonfigurationen im Verhältnis zur Anzahl der Simulationsschritte kaum ins Gewicht fallen werden.

10.4 Simulationsmethoden

Das *CHDL*-System bietet verschiedene Möglichkeiten, eine Hardwarebeschreibung softwaremäßig zu simulieren:

- Anlegen von Testvektoren.

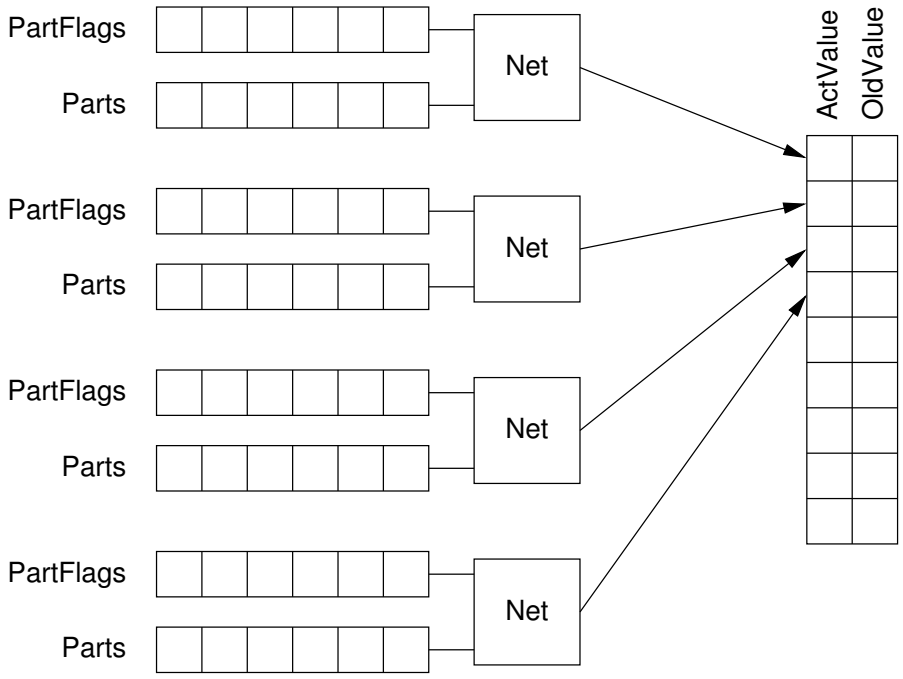


Abbildung 10.3: Datenstrukturen des Simulatorkerns

Mit den Funktionen `SetState()`, `GetState()` und `Steps()` kann der Anwender Testvektoren an die Gehäusepins des zu simulierenden FPGAs anlegen und die Ergebnisse überprüfen. Da dies innerhalb einer C++-Funktion erfolgt, hat er nahezu beliebige Möglichkeiten, dies zu gestalten. Bei rein kombinatorischen Designs kann etwa eine Verifikation durch automatisiertes Anlegen aller kombinatorisch zulässigen Eingangsvektoren erfolgen. Möglich ist auch eine reaktive Simulation, bei der der nächste Testvektor vom Ergebnis des vorigen Schrittes abhängt.

- Implementierung von Testbenches.

  Um externe Hardware, etwa Speicherbausteine oder Mikroprozessoren, in die Simulation zu integrieren, können für jeden dieser Bausteine eigene Simulationsklassen definiert werden. Diese übernehmen dann selbständig die Emulation der betreffenden Bausteine, indem sie mittels `SetState()` und `GetState()` die Zustände der Gehäusepins ermitteln und beeinflussen können.

- Hierarchische Simulation innerhalb des FPGAs.

  Der Anwender kann ein FPGA-Design bereits in Verbindung mit externer Hardware simulieren, obwohl noch nicht alle internen Module synthetisierbar implementiert sind. *CHDL* bietet die Möglichkeit, interne Module zu implementieren, die nur über eine Simulationsfunktion verfügen. Dies nutzt den Umstand, daß simulierbare Verhaltensbeschreibungen für komplexe Elemente in der Regel leichter zu implementieren sind als die synthetisierbare Hardwarebeschreibung. So kann der Entwickler z.B. ein komplexes Cache-Modul bereits vor der konkreten Hardwarebeschreibung in seiner Simulation verwenden.

In den folgenden Ausführungen werden die genannten Simulationsmethoden näher beschrieben.

10.4.1 Anlegen von Testvektoren

Zugriffe auf die Zustände der Gehäusepins

Die in der Hardwarebeschreibung verwendeten Pad-Objekte besitzen spezielle Simulationsfunktionen, über die während der Simulation der jeweils aktuelle Zustand am Gehäusepin ermittelt und verändert werden kann.

- `uchar GetState ( void )`

  Ermittelt den aktuellen Zustand am Pad und liefert einen der Werte "0", "1" oder "255" für "low", "high" bzw. "hochohmig" zurück. Durch den Aufruf dieser Funktion wird der aktuelle Pad-Zustand nicht verändert. `GetState()` kann daher innerhalb der Simulationsfunktion beliebig oft aufgerufen werden.

- `void SetState ( uchar value )`

  Setzt den Pad-Zustand für den nächsten Simulationsschritt auf den gewünschten Wert "0" oder "1". Dabei ist zu beachten, daß diese Änderung erst für den nächsten Simulationsschritt gültig wird, d.h. ein Aufruf von `GetState()` nach dieser Funktion liefert weiterhin den ursprünglichen Zustand zurück. Der Simulatorkern überprüft, ob durch einen Aufruf von `SetState()` der aktuelle Zustand tatsächlich verändert wird. Daher darf diese Funktion während eines Durchlaufs der Simulationsfunktion für jeden Pad nur höchstens einmal aufgerufen werden.

- `void SetTriState ( void )`

  Setzt den Pad-Zustand für den nächsten Simulationsschritt auf "hochohmig". Auch diese Funktion darf während eines Durchlaufs der Simulationsfunktion für jeden Pad nur höchstens einmal aufgerufen werden.

Für Pad-Arrays existieren entsprechende Funktionen, die auf das gesamte Feld wirken:

- `uint64 GetState ( void )`

  Ermittelt den aktuellen Zustand am Pad-Array und liefert den entsprechenden Wert zurück.

- `void SetState ( uint64 value )`

  Setzt den Zustand am Pad-Array für den nächsten Simulationsschritt auf den gewünschten Wert.

- `void SetTriState ( void )`

  Setzt den Zustand am Pad-Array für den nächsten Simulationsschritt auf "hochohmig".

Die konkrete Reihenfolge, in der die beschriebenen Funktionen aufgerufen werden, ist unerheblich, da der Simulatorkern in der Zeit zwischen den Simulationsschritten inaktiv ist.

Ausführen von Simulationsschritten

Die Funktion `Steps()` führt die angegebene Anzahl von Simulationsschritten durch.

Bei der Erzeugung des Taktes für eine zu simulierende synchrone Schaltung ist darauf zu achten, daß die Periodenlänge ausreichend lang gewählt wird. Umfaßt sie weniger Schritte als zur Auswertung mehrstufiger Logikanordnungen und zur Einhaltung der Setup-Zeiten benötigt werden, wird sich die Simulation nicht korrekt verhalten. Die in diesem Fall zu beobachtenden Effekte entsprechen grob etwa denen eines Designs im Echtzeitbetrieb bei zu hoher Taktfrequenz.

Simulation rein kombinatorischer Schaltungen

Die Simulation eines rein kombinatorischen Designs wird durchgeführt, indem zunächst die Pad-Zustände mittels `SetState()` bzw. `SetTriState()` auf die gewünschten Werte gesetzt werden. Da jedes kombinatorische Element im Design eine Durchlaufverzögerung von einem Schritt besitzt, sollten danach mehrere Simulationsschritte ausgeführt werden. Dann können mittels `GetState()` die vom Design verursachten Änderungen der Ausgangspads ermittelt und geprüft werden.

```
A.SetState(0);
B.SetState(1);

Steps(10);

cout << "Pad C hat Zustand" << C.GetState() << endl;
```

Simulation synchroner Schaltungen

Zur Simulation synchroner Schaltungen wird ein periodisch wechselnder Takt benötigt. Zusätzlich ist zu beachten, daß bei den angelegten Testvektoren eine Setup-Zeit von mindestens einem Simulationsschritt eingehalten werden muß. Die internen Simulationsfunktionen der implementierten Speicherelemente prüfen diese Bedingung und reagieren bei der Verletzung der Setup-Zeit mit einer entsprechenden Warnung.

```
D.SetState(0);
CE.SetState(1);
Steps(10);

D.SetState(1);

CLK.SetState(0);
Steps(10);
CLK.SetState(1);
Steps(10);

cout << "Pad Q hat Zustand" << Q.GetState() << endl;
```

Schaltungen mit mehreren Takten

Es können ohne Einschränkung auch Schaltungen mit mehreren, zueinander asynchronen Takten simuliert werden. Um für eine realitätsnahe Simulation annähernd unabhängige Takte zu erhalten, muß die entsprechende Funktionalität etwas komplexer gestaltet werden:

```
cnt1 = 0;  act1 = 0;
cnt2 = 0;  act2 = 0;

for (i = 0; i < 1000; i++)
 {
   if (cnt1++ == 10) // Wechsel alle 10 Schritte
    {
      act1 = !act1;
      CLK1.SetState(act1);
      cnt1 = 0;
    }
   if (cnt2++ == 13) // Wechsel alle 13 Schritte
    {
      act2 = !act2;
```

```
      CLK2.SetState(act2);
      cnt2 = 0;
    }
  }
```

Da ein Simulationsschritt die kleinste Zeiteinheit für den Simulator darstellt, ist zu beachten, daß der oben gezeigte Code keine wirklich asynchronen Takte erzeugen kann. Wechselt Takt 1 alle 10 Schritte und Takt2 alle 13 Schritte seinen Zustand, wird sich die entstehende Folge alle $10 * 13 = 130$ Schritte wiederholen. Die Simulation asynchroner Takte läuft folglich immer mit einer Regelmäßigkeit ab, wie sie im Echtzeitbetrieb nicht zu finden ist.

Die obigen Beispiele für getaktete Schaltungen machen bereits deutlich, daß die Simulationsfunktionen eine komplexe Struktur besitzen können, wenn das Anlegen der eigentlichen Testvektoren und die Takterzeugung in derselben Funktion vorgenommen werden. Dies könnte durch eine Aufteilung dieser Funktion in mehrere Module, die aus Sicht des Simulators parallel zueinander ablaufen, vermieden werden. Eine solche Methode wird nachfolgend vorgestellt.

10.4.2 Implementierung von Testbenches

In einer konkreten Anwendung wird der FPGA in der Regel von mehreren externen Komponenten umgeben sein.

CHDL bietet die Möglichkeit, beliebig viele Testbenches in Form von Simulationsklassen zu bilden, deren Simulationsfunktionen aus der Sicht des Simulators parallel zueinander ablaufen.

Die einzelnen Simulationsfunktionen können entweder vollständig unabhängig voneinander arbeiten, etwa zur Erzeugung zweier unabhängiger Takte, oder auf Signaländerungen anderer Module reagieren. So kann in einem Gesamtsystem beispielsweise die Erzeugung des globalen Taktes mit einem eigenständigen Modul vorgenommen werden. Ein weiteres Modul liefert dann synchron zu diesem Takt Testvektoren.

Aufbau einer Simulationsklasse

Eine Simulationsklasse ist von der Basisklasse `BaseSimModel` abgeleitet. Ihre Attributliste umfaßt Zeiger auf alle Pads bzw. Pad-Arrays, die für das Verhalten der Klasse benötigt werden. Diese Zeiger werden innerhalb der `Connect()`-Funktion mit den entsprechenden Pads des zu simulierenden FPGAs initialisiert. Weiterhin sind in der Attributliste alle internen Variablen enthalten, die von der `Evaluate()`-Funktion benötigt werden.

```
class ClockSimulator : public BaseSimModel
{
private:
  GClock* CLK;
  uint    ActCnt;
  uint    ActState;

public:
  ClockSimulator ( const char* Name );

  void  Connect ( GClock& CLK );

  void  Evaluate ( void );
};

void  ClockSimulator::Connect ( GClock& CLK1 )
{
  CLK = &CLK1;
}
```

Die Simulationsfunktion

Das eigentliche Verhalten der Simulationsklasse wird in der Funktion `Evaluate()` festgelegt. Diese wird während der Simulation in jedem Simulationsschritt aufgerufen und kann über die oben genannten Pad-Zeiger die Pad-Zustände ermitteln und verändern. So kann etwa die Simulationsfunktion eines Speicherbausteins die Steuersignale Chip-Select (CS), Write-Enable (WE) und Output-Enable (OE) überwachen und bei einer Aktivierung das entsprechende Verhalten des Speichers nachbilden.

Die folgende Funktion wechselt den Zustand des Taktausgangs alle 10 Simulationsschritte:

```
void  ClockSimulator::Evaluate ( void )
{
  if (ActCnt == 10)
    {
      ActState = !ActState;
      CLK->SetState(ActState);
      ActCnt = 0;
    }
   else
    {
      ActCnt++;
    }
}
```

Instanziierung und Anbindung

Für jedes in die Simulation zu integrierende Bauteil wird ein Objekt der entsprechenden Klasse erzeugt. Über die Methode `Connect()` erfolgt die Anbindung an die jeweils gewünschten Pads.

Anschließend muß das Simulationsobjekt mittels `AddSimModel` beim Simulator registriert werden:

```
ClockSimulator* ClockGen = new ClockSimulator("ClockGen");
ClockGen->Connect(Design->CLK);
AddSimModel(ClockGen);
```

Parametrisierung und Sonderfunktionen

Die Simulationsklassen können beliebig parametrisiert werden, um einzelne Objekte mit jeweils spezifischen Arbeitseinstellungen zu erzeugen.

Das Verhalten der Simulationsklassen kann auch durch beliebige Sonderfunktionen beeinflußt werden, die vor oder während der Simulation ausgeführt werden. So kann etwa die Frequenz eines Taktgenerators vor oder während der Simulation festgelegt werden.

```
ClockGen->SetFrequency(40);
```

10.4.3 Hierarchische Simulation

Die Simulation von Anwendermodulen (*BaseUserParts*) erfolgt durch die Simulationsfunktionen der enthaltenen Grundelemente.

Es kann jedoch aus folgenden Gründen auch vorteilhaft sein, in das Design Elemente zu integrieren, die die Simulation durch eine eigene `Evaluate()`-Funktion selbst übernehmen:

- Erhöhung der Simulationsgeschwindigkeit.

  Bei komplexen Designs kann die Simulation aufgrund der großen Anzahl der zu bearbeitenden Grundelemente sehr zeitaufwendig werden. Dies führt besonders dann, wenn

viele Taktzyklen bis zum Zeitpunkt des Fehlers simuliert werden müssen, zu unangenehm langen Wartezeiten.

Eine Lösung dieses Problems besteht darin, in denjenigen Teilen des Designs, die bereits ausreichend verifiziert sind, auf eine detaillierte Simulation zu verzichten. Stattdessen wird ein nicht synthetisierbares Simulationsmodul eingebunden, dessen Funktion durch eine verhaltensorientierte C++-Funktion emuliert wird. Eine solche Funktion kann das Verhalten auf abstrakterer Ebene behandeln und wird in der Regel deutlich schneller ausgeführt als eine detaillierte Hardwarebeschreibung.

Diese Methode birgt allerdings das Risiko nicht erkannter Abweichungen zwischen der detaillierten, synthetisierbaren Version und der verhaltensorientierten Beschreibung. Dies kann dazu führen, daß sich Designs in der Simulation korrekt verhalten, im Echtzeitbetrieb jedoch Fehler aufweisen, die dann nur schwer zu lokalisieren sind.

Werden zur Simulationsbeschleunigung jedoch nur zuverlässig geprüfte Module verwendet, kann diese Methode von großem Nutzen sein.

- Simulation im frühen Entwicklungsstadium.

  Bei einem normalen Simulationsverfahren kann ein Design erst simuliert werden, wenn es komplett implementiert ist.

  In der Praxis kann es jedoch vorteilhaft sein, wenn der Entwickler eine Schaltung bereits näherungsweise simulieren kann, obwohl ein oder mehrere komplexe Module noch nicht im Detail implementiert sind.

  Eine nur simulierbare Verhaltensbeschreibung in Form von C++-Code ist oft einfacher und schneller zu implementieren als die konkrete Hardwarebeschreibung.

  Durch eine frühe Simulation des Gesamtsystems, auch wenn sie nur näherungsweise erfolgt, können prinzipielle Probleme bereits frühzeitig erkannt werden. Dies kann verhindern, daß viel Entwicklungszeit in die Detailimplementierung von Modulen investiert wird, die sich später als ungeeignet erweisen.

CHDL bietet die Möglichkeit, spezielle interne Simulationsklassen zu implementieren, die genau wie Anwendermodule oder Primitive im Design verwendet werden können. Diese sind jedoch nicht synthesefähig, sondern unterstützen nur eine Verhaltensbeschreibung für die Simulation.

Die hierarchische Simulation ermöglicht ein Mischen von synthetisierbaren Elementen mit solchen, die nur simulierbar sind. Das Einfügen dieser Simulationsmodule kann dabei auf verschiedenen Ebenen der Hierarchie erfolgen. In der Regel wird eine Simulation umso detaillierter ablaufen, je niedriger diese Ebene ist. Auf einer hohen Ebene kann die Simulation mehr Funktionalität umfassen und wird schneller ausgeführt.

Im folgenden Beispiel wird der Aufbau eines solchen Simulationsmodells erläutert. Es handelt sich um die Simulation einer Zustandsmaschine mit vier Zuständen:

```
class Controller : public BasePart
{
public:
  IntInPin      CLK;
  IntInPin      Start;
  IntInPin      Ready;
  IntInPin      Valid;
  IntInPinArray Data;

  uint  OldCLK, State;

  Controller ( const char* Name );
```

```
  Controller ( const Controller& );
  ~Controller();

  void  InitEvaluate ( void );
  void  Evaluate ( void );
};

Controller::Controller ( const char* Name )
  : BasePart(Ctrl),
    CLK("CLK"),
    Start("Start"),
    Valid("Valid"),
    Data(8,"Data"),
    Ready("Ready")
{ }

void  Controller::InitEvaluate ( void )
{
  OldCLK = 0;  State = 0;
}

void  Controller::Evaluate ( void )
{
  if (CLK.GetLogicValue() == OldCLK)
     return;

  switch (State)
   {
     case 0:
       if (Start.GetLogicValue())
          State = 1;
       break;
     case 1:
       if (Valid.GetLogicValue() &&
          (Data.GetLogicValue() == 3))
          State = 2;
       break;
     case 2:
       if (Valid.GetLogicValue() &&
          (Data.GetLogicValue() == 5))
           State = 3;
         else
           State = 1;
      break;
     case 3:
       Ready.SetLogicValue(1);
       State = 0;
       break;
   }
  OldCLK = CLK.GetLogicValue();
}
```

Alle Variablen, die zwischen den einzelnen Simulationsschritten erhalten bleiben sollen, müssen im Klasseninterface definiert werden. Im Beispiel sind dies die Variablen `OldCLK` und

`State`.

Diese Art von Simulationsmodulen wird vom *CHDL*-Kernel genauso wie die synthetisierbaren Grundelemente verwaltet. Daher müssen im Gegensatz zu einer `BaseUserPart`-Implementierung zusätzlich der Copy-Konstruktor und ein spezieller Destruktor implementiert werden:

```
Controller::Controller ( const Controller& Ctrl )
  : BasePart(Ctrl),
    CLK(Ctrl.CLK),
    Start(Ctrl.Start),
    Valid(Ctrl.Valid),
    Data(Ctrl.Data),
    Ready(Ctrl.Ready)
{
  OldCLK = Ctrl.OldClock;  State = Clock.State;
}

Controller::~Controller()
{
  if (!CtrlFlag)
   {
     NewPart = new Controller(*this);
     CtrlFlag = 1;
   }
}
```

10.4.4 Auswertung der Simulationsergebnisse

Grafische Darstellung

Die klassische Methode der Auswertung von Simulationsergebnissen besteht in der grafischen Darstellung der Signalverläufe. *CHDL* unterstützt dieses Verfahren, indem während der Simulation eine spezielle Datei erzeugt wird. Diese enthält die Zeitverläufe ausgewählter Signale in kompakter Form. Mithilfe eines Anzeigeprogrammes können diese Verläufe angezeigt, untersucht oder auch ausgedruckt werden (Abb. 10.4).

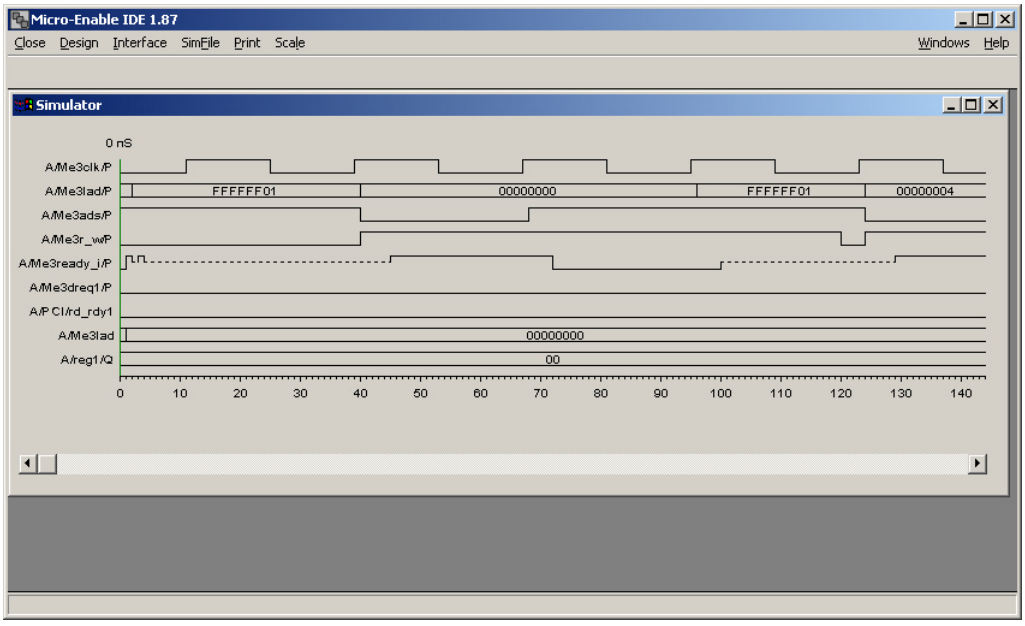


Abbildung 10.4: Grafische Darstellung der Signalverläufe

Ausführung der Applikation

Neben der grafischen Anzeige kann eine Auswertung beim *CHDL*-System auch innerhalb der Applikation erfolgen. Durch die Simulation der Pseudoregister wird die Applikation genauso ausgeführt wie im realen Betrieb. Dies bedeutet, daß dem Entwickler hier dieselben Methoden zum Debugging zur Verfügung stehen wie bei der konventionellen Softwareentwicklung. Die grafische Anzeige eignet sich am besten zur näheren Untersuchung eines bestimmten Zeitpunktes. Die Integration in die Applikation dagegen zeigt ihre Stärken besonders bei der Simulation großer Zeiträume. Hier wird die grafische Auswertungsmethode schnell sehr unübersichtlich.

10.5 Spezielle Simulationsverfahren

10.5.1 Zugriffe auf Special-Function-Register

Zugriffe auf Special-Function-Register werden innerhalb der laufenden Anwendung mittels Zeigern vorgenommen. Hier stellt sich das Problem, wie diese Zugriffe durch den Simulator abgefangen und behandelt werden können.

Es gibt prinzipiell zwei Möglichkeiten:

- Abfangen des Zugriffes durch eine Hilfsklasse und geeignet überladene Operatoren.

  Für die Simulation wird der direkte Zeiger auf einen Integer-Typ durch eine Hilfsklasse ersetzt. Innerhalb dieser können nun Schreib- und Lesezugriffe durch die überladenen Operatoren abgefangen werden. Diese Methode hat vor allem den Nachteil, daß sie einen Unterschied zwischen der simulierten und der Echtzeitanwendung einführt. Es ist denkbar, Zugriffe auf Special-Function-Register auch im Echtzeitdurchlauf mit diesen Hilfsklassen durchzuführen. Dies ermöglicht zusätzliche Debugging-Ausgaben oder Sammeln von statistischen Informationen, führt jedoch durch den zusätzlichen Code zu einer Verlangsamung der Zugriffe.

```
FPGAHelper::FPGAHelper ( uint* addr )
{ }

FPGAHelper::operator uint ()
{
  uint data = FPGA_ReadAccess(addr);
  return (data);
}

FPGAHelper::operator = ( uint data )
{
  FPGA_WriteAccess(addr,data);
}

FPGAPtr::FPGAPtr ( uint* base )
{ }

FPGAHelper FPGAPtr::operator [] ( uint indx )
{
  return (FPGAHelper(base+indx));
}
```

  Der Zugriff erfolgt nicht über einen Zeiger, sondern über die Klasse `FPGAPtr`:

```
FPGAPtr  ptr(GetFPGABaseAddress());

ptr[0] = 0x00;
a = ptr[0];
```

- Einsetzen des Exception-Mechanismus, den das Betriebssystem zur Verfügung stellt.

  Bei modernen Betriebssystemen wie *Microsoft Windows* oder *Linux* kann eine benutzerdefinierte Exception-Behandlungsroutine eingebunden werden, die bei Zugriffen auf nicht definierte Speicherbereiche aufgerufen wird. Diese Behandlungsroutine erhält beim Aufruf ausreichende Informationen vom Betriebssystem über die genaue Position der Ursache. Der Simulator kann nun die Adresse ermitteln, auf die zugegriffen werden sollte, den Zugriff softwaremäßig nachbilden und das Ergebnis in das entsprechende Prozessorregister schreiben. Danach wird die Kontrolle wieder an das ursprüngliche Programm übergeben. Der große Vorteil dieser Methode besteht darin, daß die Durchführung der Registerzugriffe bei Simulation und Echtzeit identisch ist. Dadurch können bei der Simulation auch compilerbedingte Effekte erkannt werden, die etwa durch ein vergessenes `volatile`-Schlüsselwort verursacht werden. Diese Methode erfordert eine wesentlich aufwendigere Implementierung als die zuvor genannte. Alle relevanten Assembleranweisungen, mit denen der C++-Kompiler einen Zugriff auf ein Special-Function-Register realisieren kann, müssen per Software nachgebildet werden, einschließlich aller vorhandenen Adressierungsarten. Bei den zur Zeit vorhandenen Debugger-Werkzeugen ist es jedoch nicht möglich, ein Anwendungsprogramm, daß diese Methode verwendet, unter Kontrolle des Debuggers auszuführen. Der Debugger bricht beim ersten Auftreten einer Ausnahme ab und kann die Bearbeitung nicht fortsetzen.

```
struct sigaction sa;

sa.sa_handler  = ExceptionHandler;
sa.sa_mask     = 0;
sa.sa_flags    = 0;
sa.sa_restorer = NULL;

sigaction(SIGSEGV,&sa,NULL);

void   ExceptionHandler ( int  nr )
{
  CONTEXT*  ContextRecord;
  uint      retcode;

  ContextRecord = (CONTEXT*) &nr;

  retcode = HandleException(ContextRecord);

  if (retcode != EXCEPTION_CONTINUE_EXECUTION)
     signal(SIGSEGV,SIG_DFL);
}
```

  Die Behandlungsroutine erhält vom Betriebssystem außer der Nummer der Ausnahme noch eine Reihe weiterer Informationen. Diese befinden sich in einer Struktur, von der die Ausnahmenummer nur den ersten Eintrag darstellt. Sie enthält die Inhalte aller Prozessorregister zum Zeitpunkt der Ausnahme:

```
struct CONTEXT
{
  uint   SigNr;
  uint   SegGs;
  uint   SegFs;
  uint   SegEs;
  uint   SegDs;
  uint   Edi;
  uint   Esi;
  uint   Ebp;
  uint   Esp;
  uint   Ebx;
  uint   Edx;
  uint   Ecx;
  uint   Eax;
  uint   TrapNr;
  uint   ErrorCode;
  uint   Eip;
  ushort SegCs,_SegCs;
  uint   EFlags;
  uint   _Esp;
  ushort SegSs,_SegSs;
};
```

Mit diesen Informationen ist es nun möglich, die Assembleranweisung zu ermitteln, die die Ausnahme ausgelöst hat. Abbildung 10.5 zeigt den Aufbau eines x86-Befehls. Das erste Byte enthält den Operationscode, die folgenden Bytes die notwendigen Operanden sowie Angaben zur verwendeten Adressierungsart [45, 46].

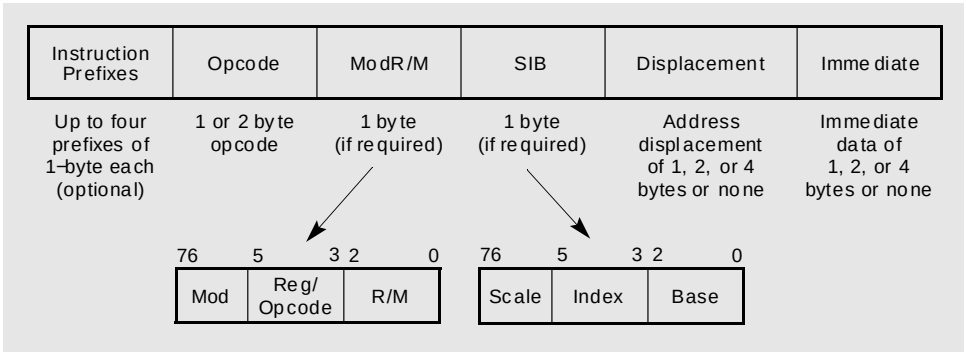


Abbildung 10.5: Aufbau eines Intel x86-Befehls

```
uchar  ReadOpcode ( CONTEXT* CRec )
{
  uchar  opcode;

  register uint  eax asm("ax");
  eax = CRec->Eip;
  __asm("movb  (%eax),%al");
  opcode = (uchar) eax;

  CRec->Eip++;

  return (opcode);
}
```

Durch eine vollständige Dekodierung dieses Befehls, insbesondere der Adressierungsart kann bei Schreibzugriffen festgestellt werden, welches Register den Quelloperanden und welches die Zieladresse beinhaltet. Dann führt der Simulator den entsprechenden Schreibzugriff auf das Funktionsregister durch. Bei Lesezugriffen wird die Quelladresse und das Zielregister für das Ergebnis ermittelt. Der Simulator führt den Lesezugriff aus und schreibt das Ergebnis in den entsprechenden Eintrag der CONTEXT-Struktur. Beim Lesen und Dekodieren des Befehls wird der Befehlszeiger (EIP) jeweils entsprechend erhöht, so daß dieser am Ende der Behandlungsroutine auf den nächsten Befehl zeigt. Danach wird die Kontrolle wieder an das Betriebssystem übergeben, das den unterbrochenen Prozess mit den modifizierten CONTEXT-Einträgen wieder aktiviert. Für den Prozeß selbst läuft dieses Verfahren vollständig transparent ab.

Wie oben dargelegt, besitzt jede dieser Methoden Vorteile und Nachteile. Keine kann alle Erfordernisse ohne Einschränkung unterstützen. Im *CHDL*-System wurden daher beide Methoden implementiert, zwischen denen der Anwender je nach vorliegendem Problemfall wählen kann.

10.5.2 Rekonfiguration von FPGAs

FPGA-Koprozessoren verwenden gelegentlich mehrere eigenständige FPGA-Konfigurationen, um nacheinander einzelne Teilprobleme einer Gesamtaufgabe zu lösen. Dies kann dann sinnvoll sein, wenn die Gesamtaufgabe die Resourcen des FPGAs übersteigen würde oder wenn mit mehreren kleineren Designs eine höhere Taktfrequenz erreichbar ist.

In eine realistische Simulation der Gesamtaufgabe muß dann auch dieses Umkonfigurieren des FPGAs einbezogen werden. Dies wird erreicht durch ein völliges Löschen und Wiederaufbauen der zu simulierenden Netzliste, während der interne Zustand aller Simulationsmodelle für die externe Hardware erhalten bleibt.

```
BaseDesign* D;

D = new Design1("D1");
ClockGen->Connect(D);
Simulation1();
delete(D);

D = new Design2("D2");
ClockGen->Connect(D);
Simulation2();
delete(D);
```

10.5.3 Simulation mehrerer FPGAs

Sind in einem FPGA-Koprozessorsystem mehrere FPGAs vorhanden, können diese gleichzeitig simuliert werden. Dabei sind auch Interaktionen zwischen den FPGAs simulierbar.

Jeder vorhandene FPGA erhält einen eindeutigen Namen, der in die Namensgebung aller Bauteile einbezogen wird. Somit können Bauteile mit gleichem Namen in mehreren FPGAs unterschieden werden.

Die Simulationsmodelle können durch die Verwendung von Zeigern auf die Gehäusepins an einen FPGA gekoppelt werden. Es lassen sich auch Verbindungen zwischen einem Simulationsmodell und mehreren FPGAs herstellen, etwa bei einem gemeinsamen Taktgenerator. Es ist jedoch nicht auf direkte Weise möglich, zwei FPGA-Pins miteinander zu verbinden. Dazu wird ein spezielles Simulationsmodell für Verbindungen benötigt, das dann an alle beteiligten FPGAs gekoppelt wird.

```
BaseDesign*     D1;
BaseDesign*     D2;
ClockSimulator* ClockGen;
BaseSimModel*   C;

D1 = new Design1("D1");
D2 = new Design2("D2");
ClockGen->Connect(D1,D2);

C = new Connections("C");
C->AddConnection(D1->DataIn,D2->DataOut);

Simulation();
```

10.5.4 Dynamisches Einbinden von Simulationsfunktionen

Funktionen, die in einer dynamischen Link-Bibliothek (DLL) enthalten sind, lassen sich während der Laufzeit eines Prozesses einbinden und aufrufen. Dadurch wird es möglich, in einem Prozeß zunächst die Hardwarebeschreibung auszuführen und die internen Bauteil- und Netzlisten aufzubauen, die Simulationsfunktionen aber erst später aus einer solchen DLL einzubinden. Es können auch mehrere Simulationsfunktionen nacheinander ausgeführt werden.

Dazu wird der *CHDL*-Kernel selbst als DLL implementiert. Beim Starten der Anwendung wird die *CHDL*-DLL automatisch geladen und mit dem Prozeß verbunden. Dieser öffnet nun dynamisch die gewünschte Simulations-DLL und führt die Simulationsfunktion aus (Abb. 10.6). Die *CHDL*-DLL wird dabei gemeinsam verwendet.

Weiterhin können innerhalb der Simulation Sicherungspunkte gesetzt werden, zu denen der aktuelle Zustand aller Netze und Bauteile auf einen Stack gelegt und zu einem späteren Zeitpunkt wiederhergestellt oder verworfen werden kann. Durch Setzen eines Sicherungspunktes zu Beginn jeder Simulation und Wiederherstellen am Ende können mit einer einmal aufgebauten Hardwarebeschreibung beliebig viele Simulationsläufe durchgeführt werden.

Insbesondere ist es möglich, nur einen Teil der Simulation zu wiederholen und dabei auf dem Ergebnis einer vorher durchgeführten Simulation aufzubauen. Dies ist dann von großem Nutzen, wenn der Fehlerzeitpunkt relativ spät liegt und der davor liegende, korrekt arbeitende Teil nicht ständig neu simuliert werden soll. Nach dem korrekt arbeitenden Teil wird ein Sicherungspunkt gesetzt. Nach dem Ablauf der weiteren Simulation wird dieser wiederhergestellt. Jetzt kann der zweite Simulationsdurchlauf gestartet werden, ohne daß der bisherige vordere Teil wiederholt werden muß (Abb. 10.7). Besonders bei komplexen Designs mit großem Datentransfer kann dies die Simulationszeiten erheblich verkürzen.

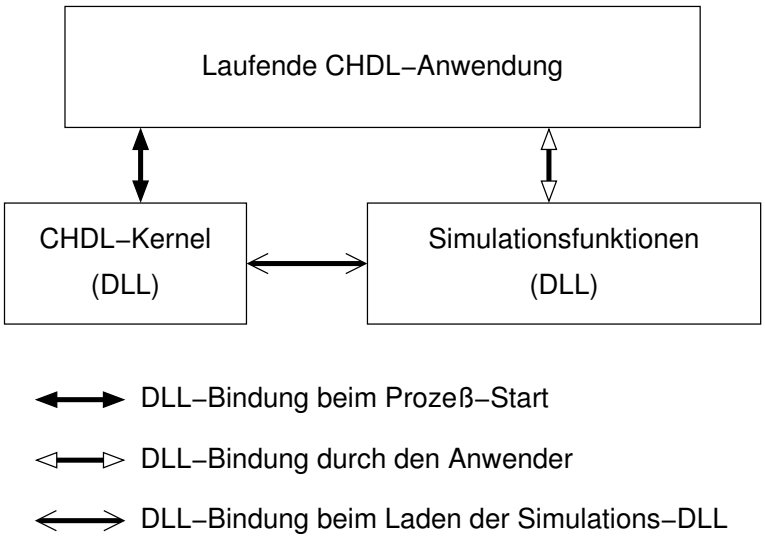


Abbildung 10.6: Einbinden von Simulationsfunktionen aus DLLs

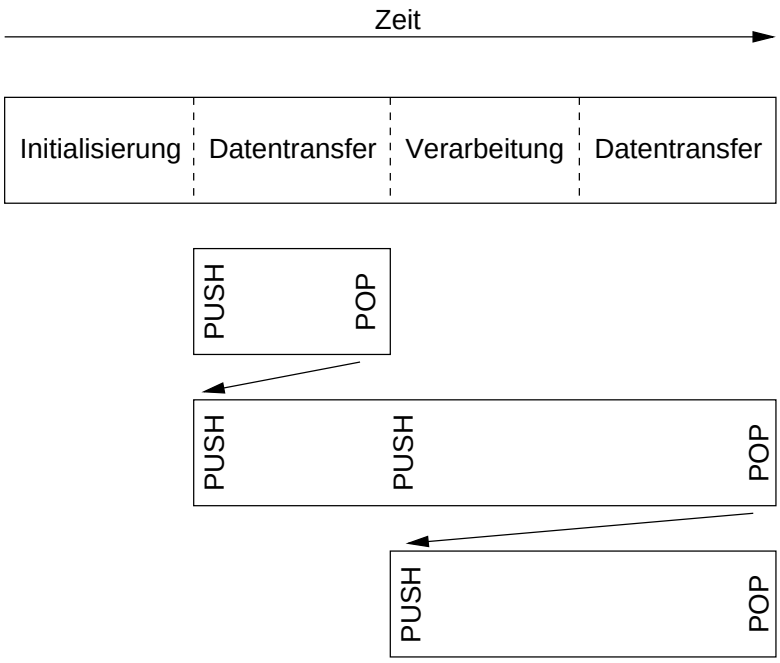


Abbildung 10.7: Verwendung von Sicherungspunkten

10.6 Zusammenfassung

CHDL ermöglicht eine vollständige Simulation von FPGA-Designs einschließlich externer Hardware.

Die Simulation wird realisiert, indem jedes Grundelement eine C++-Funktion zur Modellierung seines Verhaltens erhält. Diese Funktion ermittelt jeweils den aktuellen Zustand der Eingangssignale und berechnet die entsprechenden neuen Werte der Ausgangssignale. Über die implementierten Netze wirken die Ausgänge auf die Eingänge der angeschlossenen Bauteile, wodurch die Simulation der Gesamtschaltung möglich wird.

Im Gegensatz zu anderen Entwicklungssystemen besitzt *CHDL* keine Einschränkungen in den Simulationsmöglichkeiten. Es können sowohl synchrone als auch asynchrone Konstruktionen simuliert werden. Kombinatorische Schleifen sind ebenso zulässig wie mehrere asynchron zueinander verlaufende Takte.

Der Simulatorkernel integriert zahlreiche Optimierungsmethoden, die eine effiziente und zeitsparende Simulation erlauben. So sind die einzelnen Simulationsfunktionen nicht permanent aktiv, sondern werden explizit aufgerufen. Aufrufe erfolgen nur, wenn sich relevante Eingangssignale an dem entsprechenden Bauteil verändert haben oder dieses Bauteil die Aktivierung zu diesem Zeitpunkt selbst angefordert hat.

Weiterhin beschränkt sich die Simulation auf funktionale Verfahren mit Standardwerten für die wesentlichen Verzögerungszeiten. Relevante Logikzustände sind "0", "1" und "hochohmig". Diese Annahmen sind für die Arbeit mit FPGAs zweckmäßig.

Die Erzeugung der Simulationsstimuli kann durch Anlegen von Testvektoren oder durch Implementierung spezieller Simulationsklassen, die mit Testbenches vergleichbar sind, erfolgen.

Diese Simulationsklassen erlauben die Integration externer Komponenten, deren Verhalten hierzu mittels C++-Funktionen modelliert wird.

Auch innerhalb eines FPGA-Designs können Elemente existieren, die nur eine Simulationsfunktion, aber keine Hardwareimplementierung enthalten. Dies ist zweckmäßig, um die Simulation zu beschleunigen oder frühzeitig eine Gesamtsimulation zu erhalten, bevor alle Module synthetisierbar implementiert sind.

Es wurden spezielle Simulationsverfahren erläutert. Bei Zugriffen auf Special-Function-Register kann der Anwender zwischen zwei Methoden wählen, abhängig davon, ob eine schnellere oder eine präzisere Simulation gewünscht ist.

Die Rekonfiguration von FPGAs wird unterstützt, indem das zugrundeliegende Design während der Simulation gelöscht und neu aufgebaut werden kann.

Zur Simulation von mehreren FPGAs existieren spezielle Modelle, mit denen die Verbindungen zwischen den FPGAs realisiert werden.

Zur Fehlersuche in umfangreichen Simulationsläufen können DLL-Schnittstellen eingesetzt und Sicherungspunkte verwendet werden.

Im Bereich der Simulation zeigen sich die bedeutenden Vorteile, die die Verwendung der Programmiersprache C++ bietet.

Sowohl bei der Erstellung von Testvektoren als auch bei der Implementierung von Simulationsklassen ist die universelle Sprache C++ den Möglichkeiten der Sprache *VHDL* zur Beschreibung von Testbenches weit überlegen. Diese Überlegenheit umfaßt nicht nur sprachliche Konstrukte, sondern auch Ausführungsgeschwindigkeit sowie die Erweiterbarkeit durch DLL-Schnittstellen.

Kapitel 11
Synthese

11.1 Allgemeines

CHDL ermöglicht neben der Simulation auch die Synthese der implementierten Schaltung. Dazu werden die im Design verwendeten Elemente auf die Grundelemente abgebildet, die die Architektur des Ziel-FPGAs zur Verfügung stellt.

Die hierbei entstehende Anordnung wird in Form einer Netzliste im XNF- oder EDIF-Format exportiert. Diese Netzliste kann ohne weitere Modifikationen direkt als Eingabe für die herstellerspezifischen Place&Route-Werkzeuge verwendet werden.

Als Schnittstelle zu *VHDL*-basierten Werkzeugen kann es wünschenswert sein, anstelle einer Netzliste automatisch generierten *VHDL*-Code zu exportieren. *CHDL* verfügt über eine solche Exportfunktion, mit der auch eine Weiterverarbeitung von *CHDL*-Designs im ASIC-Bereich realisierbar wäre. Hier muß jedoch in der Regel noch eine separate Simulation des erzeugten *VHDL*-Codes mit den Technologiezellen des ASIC-Herstellers erfolgen.

Die nächsten Abschnitte vermitteln einen Überblick über die Struktur der Netzlistenformate XNF und EDIF sowie über das Aussehen von automatisch generiertem *VHDL*-Code. Abbildung 11.1 zeigt die Beispielschaltung, die der Darstellung zugrundeliegt. Sie enthält ein Flip-Flop, ein UND-Gatter sowie einige Treiber von und zu den Gehäusepins.

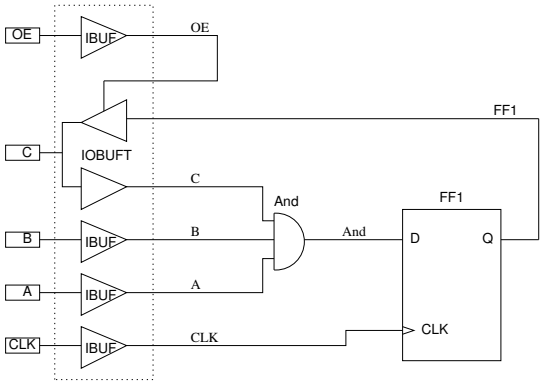


Abbildung 11.1: Beispielschaltung für die Netzlistenformate

11.2 XNF-Netzlisten

Das XNF-Format wurde von *XILINX* für die Familien bis *XC4000E* und *Spartan* eingesetzt. Für die neueren Familien *Virtex* und *Spartan-II* kann XNF nur noch sehr eingeschränkt verwendet werden, da nicht alle Grundelemente unterstützt werden.

Die Struktur von XNF orientiert sich in erster Linie an der Instanziierung der verwendeten Elemente. Jedes Element erhält dabei einen eindeutigen Namen. Der Namenszuordnung folgt eine Liste der Interface-Pins mit einer Richtungsangabe "I" (input), "O" (output) oder "B" (bidirektional), zusammen mit dem Namen des angeschlossenen Netzes.

Die komplette Netzliste beginnt mit einem Header, der die Versionsnummer des XNF-Formates und den genauen Typ des Ziel-FPGAs einschließlich Gehäuseform und Speedgrade enthält.

Diesem Header folgen eine Auflistung der Netze zu den Gehäusepins und die einzelnen Instanziierungen der eingesetzten Grundelemente.

Abgeschlossen wird die XNF-Netzliste durch die Anweisung `EOF`.

XNF-Header:

```
LCANET, 6
PART, 4028EXHQ240-3
```

Angabe des Chip-Interfaces:

```
EXT, A_EXT,   I, , LOC=P1
EXT, B_EXT,   I, , LOC=P2
EXT, C_EXT,   B, , LOC=P3
EXT, OE_EXT,  I, , LOC=P4
EXT, CLK_EXT, I, , LOC=P5
```

Instanziierung der Treiber zu den Gehäusepins:

```
SYM, A_PAD, IBUF
PIN, I, I, A_EXT
PIN, O, O, A
END
SYM, B_PAD, IBUF
PIN, I, I, B_EXT
PIN, O, O, B
END
SYM, C_PADI, IBUF
PIN, I, I, C_EXT
PIN, O, O, C
END
SYM, C_PADO, OBUFT, FAST
PIN, I, I, FF1
PIN, T, I, NET0001
PIN, O, O, C_EXT
END
SYM, OE_PAD, IBUF
PIN, I, I, OE_EXT
PIN, O, O, OE
END
SYM, CLK_PAD, IBUF
PIN, I, I, CLK_EXT
PIN, O, O, CLK
END
```

Instanziierung aller übrigen benötigten Zellen:

```
SYM, PART0001, INV
PIN, O, O, NET0001
PIN, I, I, OE
END
SYM, FF1, DFF, INIT=R
PIN, Q, O, FF1
PIN, D, I, NET0002
PIN, C, I, CLK
END
SYM, PART0002, AND
PIN, O, O, NET0002
PIN, I0, I, A
```

```
PIN, I1, I, B
PIN, I2, I, C
END
```

EDIF-Abschluß:

```
EOF
```

11.3 EDIF-Netzlisten

EDIF-Netzlisten [44, 112] stellen das Standardformat nahezu aller FPGA-Hersteller dar. Es wird für alle Architekturen unterstützt. Die Struktur von EDIF unterscheidet sich deutlich von XNF.

Eine EDIF-Netzliste enthält drei Hauptbereiche:

- Die Bibliothek externer Zellen.

  Hier sind die Zellen aufgelistet, die von der Place&Route-Software bereitgestellt und innerhalb des FPGA-Designs verwendet werden.

- Die Bibliothek interner Zellen.

  Sie umfaßt alle Zellen, die im Rahmen des hierarchischen Aufbaus des Designs im Hauptmodul oder in anderen internen Zellen verwendet werden. Ein interne Zelle kann externe Zellen oder zuvor definierte interne Zellen enthalten.

- Die Referenzierung des Hauptmoduls.

  Durch sie wird festgelegt, welche Zelle der internen Bibliothek die oberste hierarchische Ebene des Designs darstellt.

Jede Zelle umfaßt folgende Abschnitte:

- Das Zellen-Interface.

  Hier sind alle Signale einschließlich einer Richtungsangabe enthalten, die die Zelle nach außen zur Verfügung stellt.

- Die Instanzen aller enthaltenen Elemente.

  Alle Elemente, die innerhalb der Zelle verwendet werden, sind hier zusammen mit eventuellen Attributen ausgelistet.

- Die Liste aller Netze.

  Im Gegensatz zu XNF stellen bei EDIF die Netze eigene Objekte der Beschreibung dar. Jedes Netz hat einen innerhalb der Zelle eindeutigen Namen und enthält eine Auflistung aller beteiligten Pins.

Die komplette Netzliste beginnt mit einem Header, der die Versionsnummer des EDIF-Formates, das Datum der Erstellung sowie eventuell weitere Kommentare enthält.

Diesem Header folgen die Zellendefinitionen der externen und der internen Bibliothek sowie die Referenzierung des Hauptmoduls.

EDIF-Header:

```
(edif edif
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (status
    (written
      (timestamp 1 7 8 12 36 20)
```

Deklaration der externen Zellen:

```
(external PRIM
  (edifLevel 0)
  (technology (numberDefinition))
  (cell IPAD
    (cellType GENERIC)
    (view NetList
      (viewType NETLIST)
      (interface
        (port IPAD (direction INPUT))
    )))
  (cell IOPAD
    (cellType GENERIC)
    (view NetList
      (viewType NETLIST)
      (interface
        (port IOPAD (direction INOUT))
    )))
  (cell IBUF
    (cellType GENERIC)
    (view NetList
      (viewType NETLIST)
      (interface
        (port I (direction INPUT))
        (port O (direction OUTPUT))
    )))
  (cell OBUFT
    (cellType GENERIC)
    (view NetList
      (viewType NETLIST)
      (interface
        (port O (direction OUTPUT))
        (port T (direction INPUT))
        (port I (direction INPUT))
    )))
  (cell IBUFG
    (cellType GENERIC)
    (view NetList
      (viewType NETLIST)
      (interface
        (port I (direction INPUT))
        (port O (direction OUTPUT))
    )))
  (cell AND3
    (cellType GENERIC)
    (view NetList
      (viewType NETLIST)
      (interface
        (port I0 (direction INPUT))
        (port I1 (direction INPUT))
        (port I2 (direction INPUT))
        (port O  (direction OUTPUT))
    )))
```

```
    (cell DFF
      (cellType GENERIC)
      (view NetList
        (viewType NETLIST)
        (interface
          (port D   (direction INPUT))
          (port C   (direction INPUT))
          (port Q   (direction OUTPUT))
      )))
  )
```

Definition der internen Zellen:

```
  (library DESIGN
    (edifLevel 0)
    (technology (numberDefinition))

    (cell IOBUFT
      (cellType GENERIC)
      (view NetList
        (viewType NETLIST)
        (interface
          (port B  (direction INOUT))
          (port I  (direction INPUT))
          (port OE (direction INPUT))
          (port O  (direction OUTPUT))
        )
        (contents
          (instance InBuffer
            (viewRef NetList (cellRef IBUF (libraryRef PRIM))))
          (instance OutBuffer
            (viewRef NetList (cellRef OBUFT (libraryRef PRIM))))

          (net I (joined
              (portRef I)
              (portRef I (instanceRef OutBuffer))))
          (net OE (joined
              (portRef OE)
              (portRef OE (instanceRef OutBuffer))))
          (net O (joined
              (portRef O)
              (portRef O (instanceRef InBuffer))))
          (net B (joined
              (portRef B)
              (portRef I (instanceRef InBuffer))))
              (portRef O (instanceRef OutBuffer))))
      )))
```

Definition des Hauptmoduls:

```
    (cell edif
      (cellType GENERIC)
      (view view_1
        (viewType NETLIST)
```

```
(interface
)
(contents
  (instance A_pad
    (viewref view_1 (cellRef IPAD (libraryRef PRIM)))
    (property LOC (string "P1"))
  )
  (instance B_pad
    (viewref view_1 (cellRef IPAD (libraryRef PRIM)))
    (property LOC (string "P2"))
  )
  (instance C_pad
    (viewref view_1 (cellRef IOPAD (libraryRef PRIM)))
    (property LOC (string "P3"))
  )
  (instance OE_pad
    (viewref view_1 (cellRef IPAD (libraryRef PRIM)))
    (property LOC (string "P4"))
  )
  (instance CLK_pad
    (viewref view_1 (cellRef IPAD (libraryRef PRIM)))
    (property LOC (string "P5"))
  )
  (instance A)
    (viewRef view_1 (cellRef IBUF (libraryRef PRIM)))
  )
  (instance B)
    (viewRef view_1 (cellRef IBUF (libraryRef PRIM)))
  )
  (instance C)
    (viewRef view_1 (cellRef IOBUFT (libraryRef DESIGN)))
  )
  (instance OE)
    (viewRef view_1 (cellRef IBUF (libraryRef PRIM)))
  )
  (instance CLK)
    (viewRef view_1 (cellRef IBUFG (libraryRef PRIM)))
  )
  (instance And)
    (viewRef view_1 (cellRef AND3 (libraryRef PRIM)))
  )
  (instance FF1)
    (viewRef view_1 (cellRef FD (libraryRef PRIM)))
  )

  (net A_PAD (joined
    (portRef IPAD (instanceRef A_pad)))
    (portRef I (instanceRef A)))
  (net B_PAD (joined
    (portRef IPAD (instanceRef B_pad)))
    (portRef I (instanceRef B)))
  (net C_PAD (joined
    (portRef IOPAD (instanceRef A_pad)))
```

```
            (portRef B (instanceRef C)))
          (net OE_PAD (joined
            (portRef IPAD (instanceRef OE_pad)))
            (portRef I (instanceRef OE))
          (net CLK_PAD (joined
            (portRef IPAD (instanceRef CLK_pad)))
            (portRef I (instanceRef CLK))

          (net A (joined
            (portRef O (instanceRef A)))
            (portRef I0 (instanceRef And))
          )
          (net B (joined
            (portRef O (instanceRef B)))
            (portRef I1 (instanceRef And))
          )
          (net C (joined
            (portRef O (instanceRef C)))
            (portRef I2 (instanceRef And))
          )
          (net OE (joined
            (portRef O (instanceRef OE)))
            (portRef OE (instanceRef C))
          )
          (net And (joined
            (portRef O (instanceRef And)))
            (portRef D (instanceRef FF1))
          )
          (net FF1 (joined
            (portRef O (instanceRef FF1)))
            (portRef I (instanceRef C))
          )
          (net CLK (joined
            (portRef O (instanceRef CLK)))
            (portRef C (instanceRef FF1))
          )
        )
    ))
  )
```

Referenzierung des Hauptmoduls:

```
  (design edif
    (cellRef edif
      (libraryRef DESIGN))
    (property PART (string "4028EX-3-HQ240"))
  )
)
```

11.4 *VHDL*-Export

Es wird nur der Export von strukturellem *VHDL*-Code unterstützt. Dieser Code ist für Simulation und Synthese geeignet, er ist jedoch weder verständlich lesbar noch dokumentiert. Die Angabe von Bussen erfolgt nicht in der für *VHDL* eigentlich üblichen kompakten Form, sondern mittels einzelnen Signalen.

Der Code beginnt mit der Definition der *entity*, die das Interface nach außen enthält. Danach folgt eine Liste aller später verwendeten internen Netze und die Instanzen der strukturellen Komponenten mit ihren Verbindungen. Aus der Sicht des *CHDL*-Systems stellt strukturelles *VHDL* lediglich ein weiteres Netzlistenformat dar.

```
entity example is
  port ( A_PAD : in  std_logic;
         B_PAD : in  std_logic;
         C_PAD : inout std_logic;
         OE_PAD : in  std_logic;
         CLK_PAD : in  std_logic );
end example;

architecture rtl of example is

signal A_EXT : std_logic;
signal B_EXT : std_logic;
signal C_EXT : std_logic;

signal A : std_logic;
signal B : std_logic;
signal C : std_logic;
signal OE : std_logic;
signal FF1 : std_logic;
signal CLK : std_logic;
signal NET0001 : std_logic;

component PadIn
  port ( P : in  std_logic;
         O : out std_logic );
  end component;
component PadIOOE
  port ( P : inout std_logic;
         O : out std_logic;
         I : in  std_logic;
         OE : in  std_logic );
  end component;
component AND3
  port ( I1 : in  std_logic;
         I2 : in  std_logic;
         I3 : in  std_logic;
         O  : out std_logic );
  end component;
component DFF
  port ( D : in  std_logic;
         Q : out std_logic;
         C : in  std_logic );
  end component;

begin
  A_EXT <= A_PAD;
  B_EXT <= B_PAD;
  C_EXT <= C_PAD;
  OE_EXT <= OE_PAD;
```

```
  CLK_EXT <= CLK_PAD;

  A_PAD : PadIn port map (P => A_EXT,O => A);
  B_PAD : PadIn port map (P => B_EXT,O => B);
  C_PAD : PadIOOE port map (P => C_EXT,O => C,I => FF1,OE => OE);
  OE_PAD : PadIn port map (P => OE_EXT,O => OE);
  CLK_PAD : PadIn port map (P => CLK_EXT,O => CLK);
  FF1 : DFF port map (D => NET0001,Q => FF1,C => CLK);
  PART0001 : AND3 port map (I1 => A,I2 => B,I3 => C,O => NET0001);
end rtl;
```

Kapitel 12
Hardware-Debugging

12.1 Notwendigkeit und Probleme des Hardware-Debugging

Beim Hardware-Debugging versucht der Entwickler, das korrekte Verhalten eines FPGA-Designs im Echtzeitbetrieb zu verifizieren und eventuelle Fehler zu lokalisieren.

Man könnte zunächst vermuten, daß eine detailgetreue Softwaresimulation, wie sie von *CHDL* zur Verfügung gestellt wird, ausreicht, um fehlerfrei arbeitende FPGA-Designs zu erhalten. Dies trifft jedoch aus folgenden Gründen nur eingeschränkt zu:

- Fehler der Hardware oder der Simulationsmodelle.

  Die Hardware der FPGA-Karte kann fehlerhaft sein. Die möglichen Fehlerursachen reichen von Kontaktschwierigkeiten an Steckverbindungen bis hin zu defekten Leiterbahnen oder mangelhaften Lötstellen. In FPGA-Systemen, die sich noch in der Prototypphase befinden, kann weiterhin nicht ausgeschlossen werden, daß neu erstellte Simulationsmodelle für einzelne Komponenten dieses Systems noch nicht korrekt arbeiten. Hier kann die Simulation alleine keine Hilfe bieten, da eine wesentliche Abweichung zwischen den Simulationsmodellen und der Realität besteht. Im Fall eines Hardwaredefektes ist das implementierte Design korrekt, kann jedoch auf der defekten Hardware nicht korrekt ablaufen. Im Fall des fehlerhaften Simulationsmodells arbeitet das Design in der Simulation zwar korrekt, ist aber in Wirklichkeit fehlerhaft, da sich der Entwickler beim Verifizieren an der fehlerhaften Simulation orientiert hat.

- Nichtdeterministisches Verhalten durch Multitasking-Betriebsysteme und Arbitration in Bussystemen.

  Bei FPGA-Designs, die mit einer Applikation Daten austauschen, ist der genaue zeitliche Ablauf nicht vorhersehbar. Insbesondere bei Multitasking-Betriebssystemen treten Taskswitches auf, die zu Pausen unterschiedlicher Länge zwischen den FPGA-Zugriffen führen können. In Bussystemen, in denen Anforderungen mehrerer unabhängiger Busmaster durch einen Arbiter geregelt werden, können auch DMA-Zugriffe in unregelmäßiger Weise unterbrochen werden. Weiterhin entstehen Unregelmäßigkeiten durch notwendige Umsetzungen zwischen den Taktdomänen von Prozessor, Bussystem und FPGA-Design.

  Bei jedem Echtzeittestlauf liegen somit andere zeitliche Bedingungen vor, die erheblich von den Simulationsbedingungen abweichen können. Die Simulation kann immer nur eine kleine Teilmenge dieser möglichen Ablaufkombinationen umfassen. Liegt ein Designfehler vor, der nur unter bestimmten zeitlichen Bedingungen auftritt, etwa eine fehlerhafte Behandlung von Signalen der Flußsteuerung, ist dieser eventuell bei Simulationsläufen nie zu erkennen, tritt aber im Echtzeitbetrieb in unregelmäßigen und schwer zu reproduzierenden Situationen auf.

Aus diesen Gründen kann auch eine sorgfältig durchgeführte Simulation nicht garantieren, daß ein Design fehlerfrei ist.

Wenn nun ein Hardware-Debugging nicht prinzipiell vermeidbar ist, sollte es vom Entwicklungssystem optimal unterstützt werden, um dem Entwickler das Lokalisieren von Fehlern zu erleichtern.

Dazu sollen nachfolgend zunächst die spezifischen Probleme des Hardware-Debugging näher erläutert werden.

Bei der Fehlersuche im Echtzeitbetrieb unterscheidet sich die Vorgehensweise grundlegend von der Softwaresimulation, da der Designbetrieb unter realen Bedingungen stattfindet und die Untersuchung des zeitlichen Ablaufes nur mit Einschränkungen möglich ist.

Um das zeitliche Verhalten des Designs untersuchen zu können, müssen Datenverläufe aufgezeichnet werden. Dazu werden in der Regel Logikanalyzer eingesetzt. Aber selbst moderne Geräte verfügen nur über eine begrenzte Anzahl von Datenkanälen und Speicherkapazität. Es kann daher nur ein Ausschnitt aus dem Gesamtablauf aufgezeichnet werden, der durch geschickte Auswahl von Daten und Triggerbedingungen festgelegt werden muß.

Soll nun ein weiterer Ausschnitt aufgezeichnet werden, muß ein neuer Gesamtablauf durchgeführt werden. Aufgrund der bereits erwähnten Unregelmäßigkeiten in Mikroprozessorsystemen kann jedoch dieser neue Ablauf ein anderes Verhalten aufweisen. Es kann schwierig und zeitaufwendig sein, gerade die interessierenden Verläufe zu reproduzieren.

Bei der Arbeit mit FPGAs sind Verfahren denkbar, die unter bestimmten Bedingungen oder zu festgelegten Zeitpunkten den Takt anhalten. Es ist dann auch ohne Logikanalyzer möglich, gezielt den Zustand bestimmter Signale zu überprüfen.

In modernen FPGA-Koprozessoren kann dieses Verfahren jedoch aus verschiedenen Gründen oft nicht eingesetzt werden. So enthalten neuere FPGAs Elemente zur Synthese oder Phasenverschiebung von Takten. Diese Elemente verlieren bei einer Veränderung des Arbeitstaktes ihren eingeschwungenen Zustand. Wird der Takt zwischenzeitlich angehalten, ist es unmöglich, den Ablauf des Designs so fortzusetzen, als hätte es keine Unterbrechung gegeben.

Auch manche externe Komponenten, etwa SDRAMs, erlauben kein Anhalten des Arbeitstaktes.

Es gibt Ansätze, eine Debugging-Unterstützung durch Integration bestimmter Mechanismen in ein Design zu realisieren. So wurde etwa das Einfügen einer zusätzlichen Clock-Enable-Logik in das gesamte Design vorgeschlagen. Dies kann entweder synchron unter Verwendung der Clock-Enable-Eingänge von Flip-Flops oder durch Einfügen einer Logik direkt in die Taktleitung (*Gated Clock*) erfolgen. Letzteres führt jedoch meistens zu einer Verzögerung des Taktsignales und damit zu Timing-Problemen.

Ferner ist das Verfahren nicht einsetzbar, wenn mehrere asynchrone Takte vorhanden sind oder externe Bausteine kein Anhalten des Taktes erlauben.

Es gibt auch Anordnungen, in denen der Arbeitstakt von einem externen Gerät, etwa einer Kamera eingespeist wird. Hier ist ebenfalls das Anhalten dieses Taktes unmöglich.

Aber selbst dann, wenn Datenausschnitte über ausreichende Zeiträume aufgezeichnet werden können oder Verfahren zum Anhalten des Arbeitstaktes vorhanden sind, können oft nicht alle interessierenden Signale untersucht werden. Der Entwickler hat nicht ohne weiteres Zugriff auf die internen Signale des Designs. Für den Einsatz von Logikanalyzern oder sonstigen Meßgeräten sind nur die Signale an den Gehäusepins der einzelnen Komponenten zugänglich.

Um interne Signale an diese äußeren Pins zu führen, können bei FPGAs entsprechende Netze in das Design eingefügt werden. Dazu ist jedoch jedesmal der Durchlauf eines neuen Place&Route-Prozesses erforderlich, was dieses Verfahren sehr zeitaufwendig machen kann.

Eine wichtige Ausnahme von diesen Einschränkungen ist das Readback-Verfahren für FPGAs, das im nächsten Abschnitt erläutert wird.

12.2 Das Readback-Verfahren bei FPGAs

12.2.1 Allgemeines

Das Readback-Verfahren der *XILINX*-FPGAs ermöglicht es, zu jedem beliebigen Zeitpunkt den Zustand interner Signale aus dem Chip auszulesen.

Dazu werden über die Konfigurationsschnittstelle die Konfigurationsdaten zurückgelesen. Bei diesen Daten kann zwischen folgenden Bereichen unterschieden werden:

- Routing-Wege zwischen und innerhalb der CLBs.

  Diese Daten bleiben während des Betriebs konstant und sind nicht dokumentiert. Für das Readback-Verfahren sind sie nur von Bedeutung, wenn das korrekte Laden der Konfigurationsdaten überprüft werden soll.

- Flip-Flops.

  Jedem Flip-Flop sind zwei Bits im Konfigurations- bzw. Readback-Bitstrom zugeordnet:

  Ein Bit enthält beim Readback den eingefrorenen aktuellen Zustand des Flip-Flops nach dem Capture-Vorgang. Dies stellt die interessanteste Information im Readback-Datenstrom dar.

  Das andere Bit legt den Startzustand des Flip-Flops nach dem Deaktivieren des Global-Reset-Signales fest. Es wird während des Betriebs nicht verändert und beim Readback im gleichen Zustand ausgelesen. Bei einer partiellen Rekonfiguration können über dieses Bit Zustände von Flip-Flops verändert werden. Die Änderung tritt jedoch erst nach einem globalen Reset in Kraft. Es ist nicht möglich, einzelne Flip-Flops während des normalen Betriebes zu verändern.

- IOB-Eingänge.

  Diese Daten werden wie die Zustände der Flip-Flops vor dem Readback durch den Capture-Vorgang eingefroren.

- CLB-RAMs.

  Diese Bits enthalten den aktuellen Zustand der CLB-RAMs. Sie ermöglichen jeweils das Auslesen und Verändern der RAM-Inhalte. Bei einer partiellen Rekonfiguration werden die Änderungen sofort wirksam. CLB-RAMs werden nicht vom globalen Reset beeinflußt. Die *XILINX*-Software ermöglicht bisher keine Zuordnung der CLB-RAMs zu den Netznamen. Die Auswertung der Readback-Daten ist nur möglich, wenn die Position im Design bekannnt ist. Zuverlässig kann dies nur erreicht werden, wenn die für den Readback interessierenden RAMs im Design auf bestimmte Positionen vorplaziert werden. Dadurch kann jedoch bei großen Designs der Place&Route-Prozeß negativ beeinflußt werden.

- Block-RAMs.

  Diese Daten repräsentieren den aktuellen Zustand der Block-RAMs. Die Bits ermöglichen wie bei den CLB-RAMs jeweils das Auslesen und Verändern der RAM-Inhalte. Bei einer partiellen Rekonfiguration werden die Änderungen sofort wirksam. Die Block-RAMs werden nicht vom globalen Reset beeinflußt, jedoch ist während des Resets ein Auslesen nicht möglich. Zum korrekten Auslesen oder Neukonfigurieren der Block-RAMs ist ein Herunterfahren (Shutdown) des FPGAs erforderlich. Wie bei den CLB-RAMs ist bisher keine Zuordnung zu den Netznamen möglich. Dieses Problem kann auch hier nur durch eine Vorplazierung gelöst werden.

Beim Place&Route-Durchlauf wird außer dem Konfigurationbitstrom eine zusätzliche Datei (mit Endung *.ll) erzeugt, die Readback-Informationen enthält (Es sind nur die relevanten Zeilen abgedruckt):

```
Revision 3
; Created by bitgen E.37 at Fri May 31 21:08:05 2002
; Bit lines have the following form:
; <offset> <frame number> <frame offset> <information>
; <information> may be zero or more <kw>=<value> pairs
; Block=<blockname     specifies the block associated with
;                      this memory cell.
;
; Latch=<name>         specifies the latch associated with
;                      this memory cell.
;
```

```
; Net=<netname>        specifies the user net associated
;                      with this memory cell.
;
; COMPARE=[YES | NO]   specifies whether or not it is
;                      appropriate to compare this bit
;                      position between a "program" and a
;                      "readback" bitstream.
;                      If not present the default is NO.
;
; Ram=<ram id>:<bit>   This is used in cases where a CLB
; Rom=<ram id>:<bit>   function generator is used as RAM
;                      (or ROM).  <Ram id> will be either
;                      'F', 'G', or 'M', indicating
;                      that it is part of a single F or G
;                      function generator used as RAM, or
;                      as a single RAM (or ROM) built from
;                      both F and G.  <Bit> is a decimal
;                      number.
;
; Info lines have the following form:
; Info <name>=<value>  specifies a bit associated with the
;                      LCA configuration options, and the
;                      value of that bit.  The names of
;                      these bits may have special meaning
;                      to software reading the .ll file.
;
Bit     21     1   240 Block=P124 Latch=I1
...
Bit  97460   374   154 Block=CLB_R15C15 Latch=YQ Net=reg1_3
Bit  97470   374   144 Block=CLB_R14C15 Latch=YQ Net=reg1_5
Bit  97491   374   123 Block=CLB_R12C15 Latch=YQ Net=reg1_7
Bit  98505   378   153 Block=CLB_R15C15 Latch=XQ Net=reg1_2
Bit  98515   378   143 Block=CLB_R14C15 Latch=XQ Net=reg1_4
Bit  98536   378   122 Block=CLB_R12C15 Latch=XQ Net=reg1_6
Bit 106856   410   154 Block=CLB_R15C14 Latch=YQ Net=reg1_1
Bit 107901   414   153 Block=CLB_R15C14 Latch=XQ Net=reg1_0
...
Info ReadCaptureEnabled=1
Info STARTSEL0=1
```

Dieser Datei kann entnommen werden, daß sich z.B. das Bit 0 des Registers `reg1` am Ausgang `XQ` der CLB in Reihe 15, Spalte 14 befindet. Weiterhin kann aus dem Offset, der Frame-Nummer und dem Frame-Offset die Bitnummer im Readback-Datenstrom berechnet werden, die den momentanen Zustand dieses Registerbits wiederspiegelt.

Bei der Arbeit mit *CHDL* wird der Entwickler jedoch nicht mit den Details dieser Datei konfrontiert. Es sind zahlreiche Funktionen vorhanden, die die notwendigen Berechnungen automatisch vornehmen.

12.2.2 Anwendungsbeispiel

Im folgenden Anwendungsbeispiel wird demonstriert, wie der aktuelle Wert eines Registers mittels Readback aus dem FPGA gelesen werden kann. Das dazu verwendete Verfahren ist für Simulation und Echtzeitbetrieb identisch.

Zunächst wird durch den Aufruf der Funktion `Readback` der Readback-Datenstrom eingelesen. Danach kann über die Funktionen `GetRegisterState` und `GetPinState` der

aktuelle Wert von Registern und Pins ermittelt werden. Der entscheidende Zeitpunkt ist immer der Aufruf von `Readback`. Danach können beliebig viele Register und Pins abgefragt werden, die Reihenfolge ist unerheblich.

```
volatile unsigned long* volatile ptr = GetAccessPointer(board);

ptr[0] = 0x30;

printf("%08X\n",(int)ptr[0]);

Readback();
printf("reg1 = %08X\n",GetRegisterState("reg1",0,7));

printf("%08X\n",(int)ptr[0]);
printf("Status = %08X\n\n",(int)ptr[0x0100]);

Readback();
printf("reg1 = %08X\n",GetRegisterState("reg1",0,7));
```

12.2.3 Realisierung eines Logikanalyzers im Design

Zum Auffinden komplexerer Fehler können kleine Logikanalyzer, die in das Design integriert werden, von großem Nutzen sein. Die Block-RAMs der *Spartan-II*- und *Virtex*-FPGAs ermöglichen einen einfachen Aufbau solcher Analyzer mit bis zu 16 Eingangskanälen pro Block-RAM bei einer Aufnahmekapazität von 256 Samples.

Logikanalyzer benötigen zur Aufzeichnung nach dem konventionellen Verfahren erhebliche Mengen an Speicher ($Samples * Channels$ Bit). Dieser stellt in FPGAs jedoch eine begrenzte Resource dar, vor allem, wenn diese Logikanalyzer zusätzlich zum eigentlichen Design implementiert werden sollen. Da zum Auffinden komplexer Fehler in der Regel 8 bis 16 Kanäle notwendig sind, ist die Anzahl der realisierbaren Samples eingeschränkt.

Es sind einige Möglichkeiten zur Reduzierung dieses Speicherbedarfs denkbar:

- Reduzierung der gleichzeitig notwendigen Kanäle durch mehrfaches Ausführen und Zusammensetzen der einzelnen Diagramme.

  Dies setzt jedoch voraus, daß die einzelnen Durchläufe völlig identisch ablaufen, damit die Einzeldiagramme unter identischen Bedingungen zustandekommen. Aufgrund des nichtdeterministischen Verhaltens des Bussystems in einem PC ist dies in der Praxis nur schwer realisierbar.

- Komprimierung der entstehenden Daten in Echtzeit.

  Man könnte existierende Komprimierungsverfahren (z.B. RLE-Codierung) einsetzen, um die entstehende Datenmenge zu reduzieren. Dies würde jedoch die Implementierung erheblich aufwendiger gestalten. So müssten etwa zur Realisierung unterschiedlicher Wortbreiten zahlreiche Multiplexer und Barrel-Shifter integriert werden. Eine mit vertretbarem Aufwand realisierbare Lösung wäre eine RLE-Codierung mit 16 Bits pro Datenwort, bei dem das oberste Bit keinen Datenkanal darstellt, sondern folgende Bedeutung hat: Ist das Bit 0, handelt es sich um ein reguläres Sample-Wort. Ist es 1, enthalten die restlichen Bits die Anzahl der Wiederholungen des letzten regulären Wortes. Wichtig ist, daß der verwendete Algorithmus zu keinem Zeitpunkt mehr als 1 Datenwort pro Takt erzeugt, da pro Takt nur 1 Wort in den Speicher geschrieben werden kann.

- Ein flexibles Triggersystem, um den Zeitpunkt des Aufnahmestarts möglichst präzise festlegen zu können.

  Auf diese Weise werden nur relevante Daten aufgezeichnet und die zur Verfügung stehenden Samplepunkte (typischerweise 256-512 pro Block-RAM bei 8 bzw. 16 Kanälen) enthalten ausreichend Informationen, um den Fehler erkennen und beseitigen zu können. Dazu muß das Triggersystem auch die Möglichkeit bieten, Zeitpunkte vor dem Triggerpunkt aufzunehmen und zusätzliche Wartezeiten nach dem Triggerpunkt angeben zu können.

Da keine großen Zeiträume abgedeckt werden können, ist die Fähigkeit des Analyzers, auf den relevanten Zeitpunkt triggern zu können, von erheblicher Bedeutung.

In der Praxis wird sich der Entwickler bei der Fehlersuche schrittweise an den eigentlichen Fehlerzeitpunkt herantasten müssen. Dabei wird es erforderlich sein, die Triggerbedingung häufig zu ändern.

Ist diese Triggerbedingung fest im Design enthalten, muß zu jeder Änderung ein neuer Place&Route-Prozeß durchlaufen werden, was einen erheblichen Zeitaufwand bedeuten kann.

Wird der Triggermechanismus jedoch so realisiert, daß er durch Änderung von Konfigurationsbits angepaßt werden kann, ist kein neues Place&Route erforderlich. Im Konfigurationsbitstrom können nur Startwerte von Flip-Flops und CLB- bzw. Block-RAMs manipuliert werden, keine Routing-Konfigurationen.

Das Triggersystem muß also so flexibel aufgebaut sein, daß keine Routing-Änderungen notwendig sind.

Es bietet sich eine Implementierung einer Zustandsmaschine mittels synchronem Block-RAM an (Abb. 12.1). Die Ausgänge des RAMs stellen den aktuellen Zustand dar und werden an einige der Adresseingänge zurückgekoppelt. Die restlichen Adresseingänge werden als externe Signale verwendet, auf die die Zustandsmaschine mit entsprechenden Zustandsänderungen reagieren kann.

In einem Block-RAM in der Konfiguration 1024x4 lassen sich auf diese Weise 16 Zustände mit 6 externen Signalen realisieren. Die Ausführung beginnt nach Deaktivieren des globalen Resets im Zustand "0". Um auch einen Zeitraum vor dem Triggerpunkt realisieren zu können, läuft die Aufnahme der Signale permanent. Sobald das Triggersystem den Zustand "15" erreicht, startet ein Zähler, der bis zur Anzahl gewünschter Samples nach dem Triggerpunkt hochzählt und dann die Aufnahme stoppt. Später muß der aktuelle Zustand des Adresszählers ausgelesen werden, um die korrekte zeitliche Lage der Daten zum Triggerpunkt zu erhalten.

Die Zustandsübergänge vom Startzustand "0" bis zum Triggerzustand "15" sind frei programmierbar. Durch die Implementierung als RAM existieren keine Einschränkungen in der Komplexität der Zustandsübergänge.

Der spezielle Zustand "14" implementiert eine zusätzliche Funktion, bei der nach dem Triggerpunkt die Aufnahme zunächst gestoppt und erst nach Ablauf einer programmierten Wartezeit wiederaufgenommen wird. Auf diese Weise können Zeiträume mit einem zeitlichen Abstand zum Triggerpunkt aufgezeichnet werden, wenn die zur Verfügung stehenden Samples nicht ausreichen.

Abbildung 12.2 zeigt ein Beispiel für die Programmierung der Triggereinheit. Zunächst wird eine 0-1-0 Flanke des Signals T0 abgwartet. Danach muß das Signal T1 den Zustand "1" und der Signalvektor T[2:5] den Zustand "4" annehmen, um den Trigger auszulösen.

Die Programmierung erfolgt durch Angabe der gewünschten Zustandsübergänge in Abhängigkeit vom jeweils aktuellen Zustand und der Trigger-Eingangssignale. Dabei muß für jeden benutzten Zustand eine eindeutige Zuordnung zu einem Nachfolgezustand für alle Kombinationsmöglichkeiten der Eingangssignale existieren. An der Definition der Übergänge von Zustand "2" nach "15" bzw. "2" nach "2" wird deutlich, daß die notwendigen Angaben zum Verweilen in einem Zustand recht aufwendig werden können. Es bietet sich daher an, überlappende Zustandsübergänge zuzulassen, wobei immer die zuletzt angegebene Variante die

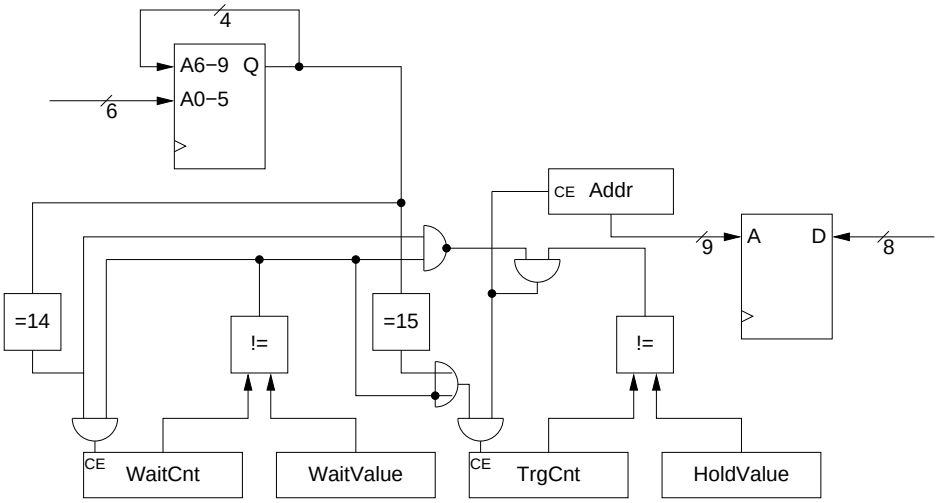
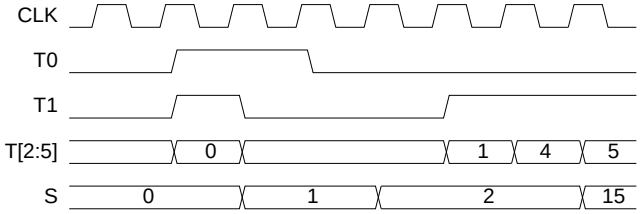


Abbildung 12.1: Integrierter Logic Analyzer

ActState	T5	T4	T3	T2	T1	T0	NextState
0	X	X	X	X	X	0	0
0	X	X	X	X	X	1	1
1	X	X	X	X	X	1	1
1	X	X	X	X	X	0	2
2	0	1	0	0	1	X	15
2	X	X	X	X	0	X	2
2	X	X	X	1	X	X	2
2	X	X	1	X	X	X	2
2	X	0	X	X	X	X	2
2	1	X	X	X	X	X	2
15	X	X	X	X	X	X	15



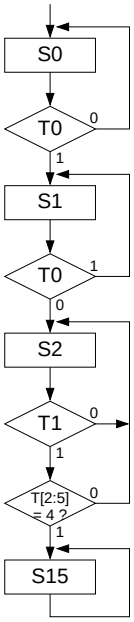


Abbildung 12.2: Triggereinheit

gültige ist. So kann zunächst ein Standardübergang von "2" nach "2" definiert werden, der dann durch spätere speziellere Übergänge überschrieben wird. Nach dem Erreichen des Zustands "15" muß dieser beibehalten werden, d.h. der Nachfolgezustand von "15" muß stets "15" sein. Das gleiche gilt für den speziellen Zustand "14".

```
SetState(Data,0,"XXXXXX",0);   // 0 -> 0
SetState(Data,0,"XXXXX1",1);   // 0 -> 1

SetState(Data,1,"XXXXXX",1);   // 1 -> 1
SetState(Data,1,"XXXXX0",2);   // 1 -> 2

SetState(Data,2,"XXXXXX",2);   // 2 -> 2
```

```
SetState(Data,2,"01001X",15);  // 2 -> 15

SetState(Data,15,"XXXXXX",15); // 15 -> 15
```

Abbildung 12.3 zeigt die verschiedenen Anordnungen der Aufnahmezeiträume vor und nach dem Triggerpunkt.

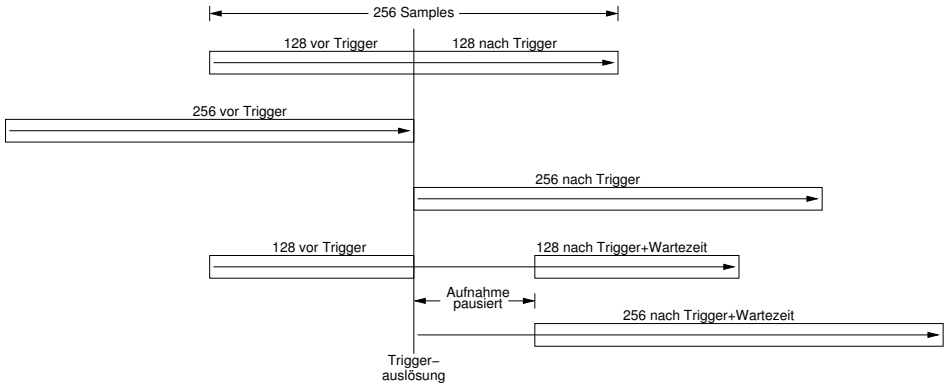


Abbildung 12.3: Triggerzeiträume

Ein ähnliches System wurde in [73] in einem Zusammenwirken von *JHDL* mit *JBits* [111] und *Chipscope* [110] von *XILINX* vorgestellt.

Das hier erläuterte *CHDL*-Verfahren weist demgegenüber jedoch eine deutlich engere Kopplung der beteiligten Komponenten sowie eine größere Flexibilität auf. Von großem Vorteil ist insbesondere die eindeutige Benennung der Netznamen. Dies erlaubt eine Automatisierung der Vorgehensweise sowie gezielte Änderungen am Design ohne ein erneutes Place&Route, wie es im nächsten Abschnitt erläutert wird.

12.3 Designänderungen ohne erneutes Place&Route

Beim Hardware-Debugging entsteht oft ein großer Zeitaufwand durch die wiederholte Änderung des Designs und den danach erforderlichen Place&Route-Prozeß.

Es gibt jedoch einige Methoden, mit denen bereits geroutete Designs in beschränktem Umfang ohne neues Place&Route verändert werden können.

12.3.1 Einsatz von LUTs und partieller Rekonfiguration

Der Entwickler kann in seinem Design mehrere alternative Teilschaltungen realisieren, von denen dann eine durch den Einsatz von Lookup-Tabellen als Multiplexer selektiert werden kann. Die Lookup-Tabellen können durch partielle Rekonfiguration ohne Place&Route verändert werden. Bei diesem Verfahren müssen jedoch alle Teilschaltungen und Multiplexer im Design integriert werden. Dadurch ist der Umfang der möglichen Designänderungen begrenzt.

12.3.2 Fernsteuerung des *FPGA-Editor*

An einem bereits gerouteten Design können mittels *FPGA-Editor* nahezu beliebige Änderungen vorgenommen werden. So können etwa Netze gelöscht und andere hinzugefügt werden. Dadurch wird es möglich, z.B. an einen oben beschriebenen Onchip-Logikanalyzer nachträglich beliebige Signale anzukoppeln, ohne daß dazu im Design Multiplexer vorgesehen werden müssen. Voraussetzung für dieses Verfahren ist jedoch, daß die Namen der Netze eindeutig bekannt sind.

Die Designänderungen lassen sich in einer Batchdatei zusammenfassen, die vom *FPGA-Editor* automatisch nach dem Start ausgeführt wird. Diese Batchdatei kann auch automatisch generiert werden.

Der Zeitaufwand für dieses Verfahren setzt sich zusammen aus den Zeiten, die der *FPGA-Editor* für das Routen der neu hinzugefügten Netze benötigt, sowie aus dem Zeitaufwand für das Neuerzeugen des Bitstroms mittels `bitgen`.

Der laufzeitintensive komplette Place&Route-Prozeß kann auf diese Weise vermieden werden.

12.4 Zusammenfassung

In der Praxis sind Abweichungen zwischen Simulation und Echtzeitbetrieb nicht immer zu vermeiden. Als Gründe dafür wurden zum einen Fehler der Hardware bzw. der Simulationsmodelle genannt, zum anderen das nicht deterministische Verhalten von Mikroprozessorsystemen.

Da die Verfahren zum Hardware-Debugging deutlich von denen der Simulation abweichen und eigene spezifische Probleme aufweisen, muß es vom eingesetzten Entwicklungssystem gezielt unterstützt werden.

Bei *CHDL* wurde der Schwerpunkt auf die Unterstützung des Readback-Verfahrens und der partiellen Rekonfiguration gelegt.

Das Readback-Verfahren wurde durch Implementierung spezieller Funktionen wie etwa `GetRegisterState()` anwenderfreundlich gestaltet. Die zum Teil komplexen Berechnungen der Bitpositionen im Readback-Datenstrom konnten komplett hinter diesen Funktionen verborgen werden.

Im Zusammenhang mit den Readback-Funktionen zeigen sich die deutlichen Vorteile, die *CHDL* gegenüber anderen Systemen besitzt.

Die eindeutige und vorhersehbare Benennung der Bauteile in der Netzliste, die auch beliebig tiefe Hierarchien berücksichtigt, macht die unkomplizierte Anwendung des Readback-Verfahrens erst möglich. Auch die Entscheidung, die Benennung der Netze am Namen des treibenden Bauteils auszurichten, zeigt hier ihre Vorteile. Die Zuordnung der Speicherelemente zu den Positionen im Readback-Datenstrom erfolgt nämlich nicht über die Bauteilnamen sondern über die Netznamen.

An einem Beispiel wurde die einfache Handhabbarkeit der Readback-Funktionen mit *CHDL* demonstriert.

Die Anwendung der partiellen Rekonfiguration wurde anhand eines im FPGA-Design implementierten Logikanalyzers erläutert. Durch die Rekonfiguration können gezielt Zustände einzelner Speicherelemente verändert werden, ohne einen erneuten zeitaufwendigen Place&Route-Prozeß starten zu müssen. Im Beispiel wurde diese Fähigkeit dazu verwendet, die Triggerbedingungen für den Logikanalyzer festzulegen.

CHDL trägt durch seine spezielle Unterstützung nicht nur dazu bei, manche Funktionalitäten, wie etwa das Auslesen interner Signale, überhaupt erst zu ermöglichen. Vielmehr erlaubt es auch eine einfache Handhabbarkeit der nötigen Maßnahmen sowie eine deutliche Reduzierung des Zeitaufwandes, der zum Lokalisieren von Fehlern benötigt wird.

Teil IV

Hochsprachenorientierte Hardwarebeschreibung mit *CHDL*

Kapitel 13
Einführung

13.1 Die Problematik hochsprachenorientierter Beschreibungen

In den letzten Jahren sind einige Systeme [95, 20, 38, 123] entstanden, die Hardwarebeschreibungen mittels einer Hochsprache ermöglichen. Damit kann die Beschreibung auf einer weit höheren Abstraktionsebene erfolgen, als dies zur Zeit mit den strukturellen und verhaltensorientierten Sprachen erreichbar ist.

Im Idealfall sollte ein Algorithmus, der ursprünglich in einer universellen Programmiersprache wie C für ein Mikroprozessorsystem erstellt wurde, ohne wesentliche Änderungen in eine effiziente Hardwarestruktur umgesetzt werden können. Damit wären auch Entwickler, die keine Hardwarekenntnisse besitzen, in der Lage, innerhalb kurzer Zeit Designs für FPGAs zu erstellen.

Dieser Idealfall konnte bisher jedoch noch von keinem der existierenden Systeme erreicht werden. Die notwendigen Änderungen, die der Entwickler bei der Portierung bereits implementierter Algorithmen von einem Mikroprozessorsystem auf den Hardwarekompiler vornehmen muß, sind zum Teil erheblich. Auch die Effizienz der Umsetzung, sowohl bei den sprachlichen Konstrukten als auch beim Resourcenbedarf und den erreichbaren Taktfrequenzen, ist noch nicht ausreichend.

Die Problematik ist vergleichbar mit der Portierung von Programmcode eines universellen Mikroprozessors auf einen Mikrokontroller mit eingeschränkten Ressourcen.

Im Gegensatz zu den gängigen Mikroprozessoren verfügen Mikrokontroller nur über sehr wenig Arbeitsspeicher für Variablen und Stack. Zeiger sind nur eingeschränkt einsetzbar und automatische Variablen auf dem Stack werden nur wenig unterstützt.

Diese Einschränkungen können zur Folge haben, daß der ursprüngliche Programmcode nicht effizient auf dem Mikrocontroller ausgeführt werden kann. Die dann vorzunehmenden Änderungen gehen weit über die geringen Anpassungen hinaus, die üblicherweise bei der Portierung auf einen anderen Kompiler nötig sind. So kann eine komplette Umstellung des Codes, oft verbunden mit der Implementierung völlig anderer Algorithmen und Optimierungsmethoden, erforderlich sein.

Der Unterschied zwischen einem universellen Mikroprozessor und einer FPGA-basierten Hardwarestruktur ist noch viel grundlegender als zwischen einem Mikroprozessor und einem Mikrokontroller:

In einem FPGA steht eine Vielzahl von Funktionalitäten zur Verfügung, die in Mikroprozessoren bzw. -kontrollern nicht vorhanden sind. Beispiele sind echte Dual-Port-RAMs, zahlreiche völlig voneinander unabhängige Speicherelemente sowie die Fähigkeit, mehrere Schnittstellen zum Datentransfer zu realisieren.

Umgekehrt stellen FPGAs manche Funktionen nur eingeschränkt oder überhaupt nicht bereit, die in Mikroprozessoren regelrecht im Überfluß existieren: Große Speicherbereiche, die eine Vielzahl von Variablen zulassen, große Stackbereiche, mit denen rekursive Algorithmen realisiert werden können sowie praktisch unbegrenzte Möglichkeiten, Daten zwischen verschiedenen Variablen zu transferieren. In FPGAs sind rekursive Algorithmen schwer zu realisieren und jede Transfermöglichkeit von Daten von einem Register zu einem anderen erfordert die Implementierung von Signalleitungen und Multiplexern.

Bei den konkreten sprachlichen Konstrukten besteht die Problematik, daß diese einerseits in ihrer Bedeutung möglichst nah an den universellen Programmiersprachen liegen, jedoch andererseits auch die Vorteile der Hardwareprogrammierung wie etwa Parallelität und Pipelining unterstützen sollen. Es besteht eine semantische Lücke [88] zwischen den universellen Programmiersprachen und der konfigurierbaren Hardware, die sich auf die Effizienz der Umsetzung auswirkt.

Daher stellt sich die prinzipielle Frage, inwieweit die Ansätze, die Hardwarebeschreibung komplett auf die Ebene der Hochsprachen zu verlagern, ihre Ziele erreichen können.

Das Ausführungsmodell einer Hochsprache, das sich zwecks Kompatibilität an universellen Programmiersprachen orientiert, wird immer von den Modellen der Hardwareprogrammierung abweichen. Der Entwickler muß sich bei der Implementierung nach diesem konventionellen Ausführungsmodell richten, um seinen Algorithmus in der betreffenden Hochsprache formulieren zu können. Eventuell ist sogar erforderlich, daß er Teile der zu realisierenden Schaltung etwas umständlicher als nötig ausdrücken muß, um dem Modell zu entsprechen. Der automatische Optimierer des Entwicklungssystems steht dann vor der nahezu unlösbaren Aufgabe, die eigentliche Intention des Entwicklers ermitteln zu müssen, um die optimale Hardwarestruktur zu generieren.

Die Tatsache, daß sich Hochsprachen aus diesen Gründen nicht effizient umsetzen lassen, bedeutet jedoch nicht, daß sie prinzipiell ungeeignet für die Designerstellung sind.

Die Probleme existierender Systeme beruhen im wesentlichen darauf, daß sie die komplette Designerstellung auf der Ebene der Hochsprachen vornehmen wollen. Hochsprachen haben zweifellos Vorteile bei der Beschreibung komplexer Algorithmen. Wäre es möglich, einzelne zeitkritische Teile eines Designs auf einer niedrigeren und effizienteren Beschreibungsebene zu formulieren und die Hochsprache für die Steuerung des Gesamtablaufes einzusetzen, entfiele die Effizienzproblematik weitgehend.

13.2 Anforderungen an eine Hochsprache

Eine Hochsprache zur Hardwarebeschreibung soll analog zur konventionellen Softwareentwicklung eine Ebene mit höherem Abstraktionsgrad realisieren. Diese kann den Entwickler von zahlreichen Implementierungsdetails entlasten und den Programmcode leichter überschaubar und wartbar gestalten.

Eine Hochsprache zur Hardwarebeschreibung sollte insbesondere

- eine Loslösung von einzelnen Taktschritten erlauben.

  Für spezielle Situationen, etwa die Implementierung von Protokollen, sollte dennoch die explizite Angabe von Takten zulässig sein.

- Möglichkeiten zur Modularisierung bieten.

  Dies ist notwendig, um Komplexität zu beherrschen und Wiederverwendung zu vereinfachen.

- eine effiziente Umsetzung in die Hardware vornehmen.

  Die Hauptvorteile beim Einsatz von FPGAs bestehen in der Ausnutzung paralleler Strukturen sowie von Pipelining durch mehrfach vorhandene Rechenwerke und Daten-Interfaces. Bei der Realisierung einer Hochsprache muß darauf geachtet werden, daß diese Vorteile auch genutzt werden können und nicht durch Ineffizienzen in der konkreten Hardwareumsetzung wieder neutralisiert werden.

- die gemeinsame Nutzung beschränkter Resourcen durch mehrere Programmteile unterstützen.

  In FPGA-Systemen existieren oft Komponenten, etwa SDRAMs, die gemeinsam genutzt werden müssen. Die Hochsprache sollte den Entwickler hierbei insbesondere von Implementierungsdetails der Arbitration entlasten.

- Unterstützung für die Programmierung mit nebenläufigen Funktionen bieten.

  Damit kann eine Entsprechung zur konventionellen Programmierung mit Threads hergestellt werden. Der Algorithmus wird auf mehrere Ausführungsstränge verteilt, die die jeweils lokal vorhandenen Rechenwerke ansteuern.

- Source-Level-Debugging ermöglichen.

  Während der Softwaresimulation und auch beim Hardware-Debugging sollte dem Entwickler die aktuelle Ausführungsposition sowie der Inhalt von Variablen angezeigt werden können. Da die Zuordnung zu einzelnen Takten vom Hochsprachenkompiler im wesentlichen automatisch vorgenommen wird, müssen zusätzliche Informationen über diese Zuordnung zur Verfügung gestellt werden. Dies entspricht den Debug-Informationen in der konventionellen Programmierung.

13.3 Das Konzept der hochsprachenorientierten Hardwarebeschreibung mit *CHDL*

Aus den im vorigen Abschnitt genannten Gründen wurde beim *CHDL*-System bewußt darauf verzichtet, eine umfassende Hochsprachenebene für die *komplette* Designerstellung zu realisieren.

Vielmehr verfolgt *CHDL* das Ziel, die Hochsprachenebene gleichberechtigt neben die Ebenen der strukturellen Beschreibung und der Definition von Zustandsmaschinen zu stellen.

Die Hochsprache bildet eine erweiterte Form der Zustandsmaschinenbeschreibung, indem die notwendigen Sprunganweisungen durch Elemente der strukturierten Programmierung (`for`, `while`, `switch` usw.) ersetzt werden. Dies erlaubt eine kompaktere Beschreibung komplexer sequentieller Abläufe.

Wird in einem Gesamtdesign für unterschiedlich zeitkritische Teile jeweils die am besten geeignete Form der Hardwarebeschreibung gewählt, können sich diese Ebenen in optimaler Form ergänzen. Voraussetzung ist dazu, daß diese Kombination unterschiedlicher Beschreibungsformen möglichst umfassend vom Entwicklungssystem unterstützt wird.

Die zur Zeit existierenden Systeme berücksichtigen gerade diese Anforderung nicht ausreichend. Dies liegt darin begründet, daß sie im Gegensatz zur hier vertretenen Auffassung die komplette Hardwarebeschreibung auf die Hochsprachenebene verlagern wollen. Teilimplementierungen auf niedrigeren Ebenen sind entweder überhaupt nicht oder nur durch heterogene Einbindung von *VHDL*-Modulen möglich.

Diese Vorgehensweise ist vergleichbar mit der Fähigkeit moderner C++-Kompiler, an zeitkritischen Stellen Assemblercode zu integrieren, um effizientere Implementierungen zu erreichen.

Die Hochsprachenunterstützung von *CHDL* folgt dem Konzept, daß einzelne Module des Designs (entsprechend den *BaseUserParts*) mittels der Hochsprachenbeschreibung implementiert und beliebig mit Modulen anderer Beschreibungsebenen kombiniert werden können.

Der wesentliche Unterschied zu anderen hochsprachenorientierten Systemen besteht darin, daß bei *CHDL* der notwendige Kompiler direkt auf der *CHDL*-Klassenbibliothek aufbaut. Er besitzt damit ein direktes Interface zu den Grundelementen. Auf diese Weise läßt sich eine optimale Integration der verschiedenen Ebenen der Hardwarebeschreibung realisieren. Die Hochsprachenbeschreibung kann zusammen mit dem strukturellen *CHDL*-Code aus einer einzigen Entwicklungsumgebung heraus bearbeitet, simuliert und synthetisiert werden.

Kapitel 14
Realisierung des *CHDL*-Hochsprachenkompilers

14.1 Überblick

Bei der Übersetzung einer hochsprachenorientierten Hardwarebeschreibung in eine synthetisierbare strukturelle Anordnung bestehen folgende drei Problembereiche:

- Implementierung von Variablen.

  Die in der Hochsprachenbeschreibung verwendeten Variablen müssen in geeigneter Weise auf die Speicherelemente abgebildet werden, die in FPGAs zur Verfügung stehen.

- Auswertung arithmetischer Ausdrücke.

  Jede Hochsprache erlaubt die direkte Angabe arithmetischer Ausdrücke. Diese können beliebig komplex sein. Außerdem können sie Elemente unterschiedlicher Datentypen enthalten, die angeglichen werden müssen. Zudem befinden sich Zuweisungen an vorgegebenen Ausführungspositionen, dürfen also nur zu bestimmten Zeitpunkten aktiviert werden.

- Umsetzung der Kontrollanweisungen.

  Aus Kontrollanweisungen wie `if`, `while`, `for` usw. müssen entsprechende Zustandsübergänge der zugrundeliegenden Zustandsmaschine gebildet werden.

Zur Auswertung arithmetischer Ausdrücke und Zuweisungen sowie zur Umsetzung der Kontrollanweisungen verwendet das *CHDL*-System ein virtuelles Prozessormodell. Dieses wird in den folgenden Ausführungen *Hardware Virtual Machine* (HVM) genannt. Der Hochsprachenkompiler erzeugt als Zwischenstufe einen Bytecode, der Steuerungsanweisungen für den virtuellen Prozessor darstellt. Insoweit besitzt das Konzept Ähnlichkeiten mit der *JAVA virtual machine* (JVM) [90]. Im Gegensatz zur JVM ist die HVM jedoch nicht auf die direkte Ausführung von Algorithmen optimiert, sondern zunächst auf die Bildung von Hardwarestrukturen, die später den Algorithmus ausführen. Während beispielsweise die JVM während des Abarbeitens des Bytecodes eine Addition direkt ausführt, generiert die HVM einen Hardwareaddierer.

Die HVM unterstützt weiterhin spezielle Anweisungen für die explizite Parallelisierung von Operationen. Insofern ist das hier eingesetzte Verfahren flexibler als die Kompilierung echten *JAVA*-Bytecodes nach Hardware, wie sie von [17] realisiert wurde.

14.2 Implementierung von Variablen

Variablen stellen grundlegende Elemente jeder Hochsprache dar. In ihnen können berechnete Werte gespeichert werden. Der aktuelle Wert bleibt solange erhalten, bis der Variablen ein neuer Wert zugewiesen wird. Das Auslesen erfolgt zerstörungsfrei.

In FPGAs lassen sich Variablen auf unterschiedliche Weise realisieren:

- Diskrete Speicherelemente (Flip-Flops).

  Für jedes Bit der Variablen wird ein Flip-Flop verwendet. Auf diese Weise kann die Implementierung genau an die angeforderte Datenbreite der Variable angepaßt werden. Der aktuelle Wert ist permanent verfügbar. Er kann von beliebig vielen anderen Elementen verwendet werden. Es ist möglich, der Variablen in jedem Takt einen neuen

Wert zuzuweisen. Erhält sie diesen aus verschiedenen Datenquellen, so müssen diese jedoch mit einem Multiplexer entkoppelt werden. Zu welchen Zeitpunkten und aus welcher Quelle sie einen neuen Wert erhält, wird von der aktuellen Ausführungsposition bestimmt.

- LUT-basiertes CLB-RAM.

  Für jedes Bit der Variablen wird ein Bit eines CLB-RAMs verwendet. Diese Methode besitzt den Vorteil, daß keine diskreten Flip-Flops belegt werden. Jedes Bit des CLB-RAM besitzt jedoch nur eine Eingangs- und eine Ausgangsleitung, die Werte sind also nicht separat nach außen geführt. Nur die jeweils addressierten Bits sind beschreibbar bzw. lesbar. Diese Form der Implementierung kann für Variablen sinnvoll sein, deren Werte nur selten benötigt werden. Sind in einem CLB-RAM Bits von mehreren Variablen enthalten, kann in einem Takt nur einer davon ein neuer Wert zugewiesen werden.

- Block-RAM.

  Dies ist zunächst vergleichbar mit der Implementierung im CLB-RAM. Block-RAMs sind jedoch weniger flexibel, da sie nur wenige Kombinationen aus Daten- und Adressbreiten unterstützen. Zudem benötigen sie zum Auslesen einen zusätzlichen Taktzyklus.

Im *CHDL*-System werden Variablen mittels der zuerst genannten Methode in diskreten Flip-Flops implementiert. Diese stellt im Zuammenhang mit Parallelisierung und Pipelining die flexibelste Lösung dar. Die beiden anderen Methoden sind wegen der fehlenden parallelen Addressierbarkeit nicht geeignet.

Jede Variable der Hardwarebeschreibung existiert als eigenständige Implementierung. Dies stellt einen wesentlichen Unterschied zur konventionellen Programmierung dar, bei der Variablen nicht nur in Prozessorregistern, sondern auch in Hauptspeicheraddressen existieren können.

Bei der Übersetzung der *CHDL*-Hochsprachenbeschreibung stehen die Möglichkeiten zur Parallelisierung und zum Pipelining im Vordergrund, nicht die unbedingte Entsprechung zu klassischem C/C++. Daher werden einige Einschränkungen in Kauf genommen, die sich durch die gewählte Methode der eigenständigen Implementierung ergeben:

So unterstützt die Beschreibung keine direkten rekursiven Strukturen. Diese würde eine Implementierung lokaler Variablen in einem Stack sowie eine aufwendigere Verwaltung der Rücksprungaddressen erfordern.

Weiterhin existieren keine Zeiger auf Variablen. Die Einführung von Zeigern hätte bei der Umsetzung in eine Hardwarestruktur einen großen Implementierungsaufwand durch auswählende Multiplexer zur Folge, der den Resourcenverbrauch deutlich erhöhen und das Zeitverhalten verschlechtern würde. Die indirekte Addressierung von Variablen über Zeiger stellt in der konventionellen Programmierung ein mächtiges Verfahren dar. Im Zusammenhang mit Parallelisierung und Pipelining führt sie jedoch zu wenig effizienten Strukturen.

14.3 Auswertung arithmetischer Ausdrücke

14.3.1 Die Hardware Virtual Machine (HVM)

Eine Hochsprache muß die Angabe arithmetischer Ausdrücken unterstützen. Dazu ist ein automatisches Schema notwendig, nach dem beliebig komplexe Ausdrücke unter Berücksichtigung der Operatorprioritäten und -Assozitäten ausgewertet werden können. Abbildung 14.1 zeigt die für C/C++ relevanten Operatoren und deren Prioritäten [79].

Die Aufgabe der HVM besteht darin, eine standardisierte Struktur bereitzustellen, in der die Auswertung beliebig komplexer Ausdrücke, Zuweisungen und Kontrollstrukturen erfolgen kann. Damit liefert sie ein Interface zwischen der lexikalischen Analyse der Hochsprachenbeschreibung und der konkreten Erzeugung der entsprechenden Hardwarestrukturen.

Prior.	/Assoz.	Operator	Funktion
16	L	−>, .	Elementauswahl
16	L	[]	Indexoperator
16	L	()	Funktionsaufruf
16	L	()	Klammerung in Ausdrücken
15	R	++, −−	Inkrement, Dekrement
15	R	~	bitweise Negation
15	R	!	logische Negation
15	R	+, −	Vorzeichen
15	R	*, &	Inhaltsoperator, Adreßoperator
15	R	()	Typkonversion
14	L	−>*, .*	Elementauswahl
13	L	*, /, %	Multiplikative Operatoren
12	L	+, −	Additive Operatoren
11	L	<<, >>	Shift−Operatoren
10	L	==, !=	Gleichheitsoperatoren
9	L	<, <=, >, >=	Relationale Operatoren
8	L	&	bitweises UND
7	L	^	bitweises Exklusiv−ODER
6	L	|	bitweises ODER
5	L	&&	Logisches UND
4	L	||	Logisches ODER
3	R	?:	Konditional−Operator
2	R	=, *=, /=, %=, +=, −= >>=, <<=, &=, ^=, |=	Zuweisungsoperatoren
1	L	,	Kommaoperator

Abbildung 14.1: Operatorprioritäten von C/C++

Die HVM besitzt ein Ausführungsmodell, das einerseits allgemein genug ist, um beliebige Algorithmen umsetzen zu können, andererseits jedoch konkret genug, um effiziente Ergebnisse bei der Umsetzung zu erhalten.

Das grundlegende Prinzip, nach dem die sequentielle Struktur der Hochsprachenbeschreibung implementiert wird, ist das Modell der modifizierten *One-Hot*-Zustandsmaschine, wie sie vom strukturellen *CHDL*-System bekannt ist. Die Verteilung der zu implementierenden Zuweisungen auf die einzelnen Zustände erfolgt nach den Datenabhängigkeiten, die von der lexikalischen Analyse ermittelt werden.

Das Analysemodul des Hochsprachenkompilers erzeugt bei der Verarbeitung der Hochsprachenbeschreibung elementare Anweisungen wie `PUSH`, `LABEL`, `JMP` oder `OP`. Die Ausgabe erfolgt in Form eines Bytecodes.

Die HVM arbeitet diesen Bytecode ab und implementiert dabei Schaltfunktionen mittels eines Arithmetikstacks sowie einzelne Zustände mittels des zugrundeliegenden Zustandsmaschinenmodells.

14.3.2 Der Arithmetikstack

Der Arithmetikstack dient zur Implementierung kombinatorischer Logik. Zusammen mit der rekursiven Vorgehensweise des Hochsprachenkompilers können damit auf einfache Weise Ausdrücke auswertet und implementiert werden (Abb. 14.2).

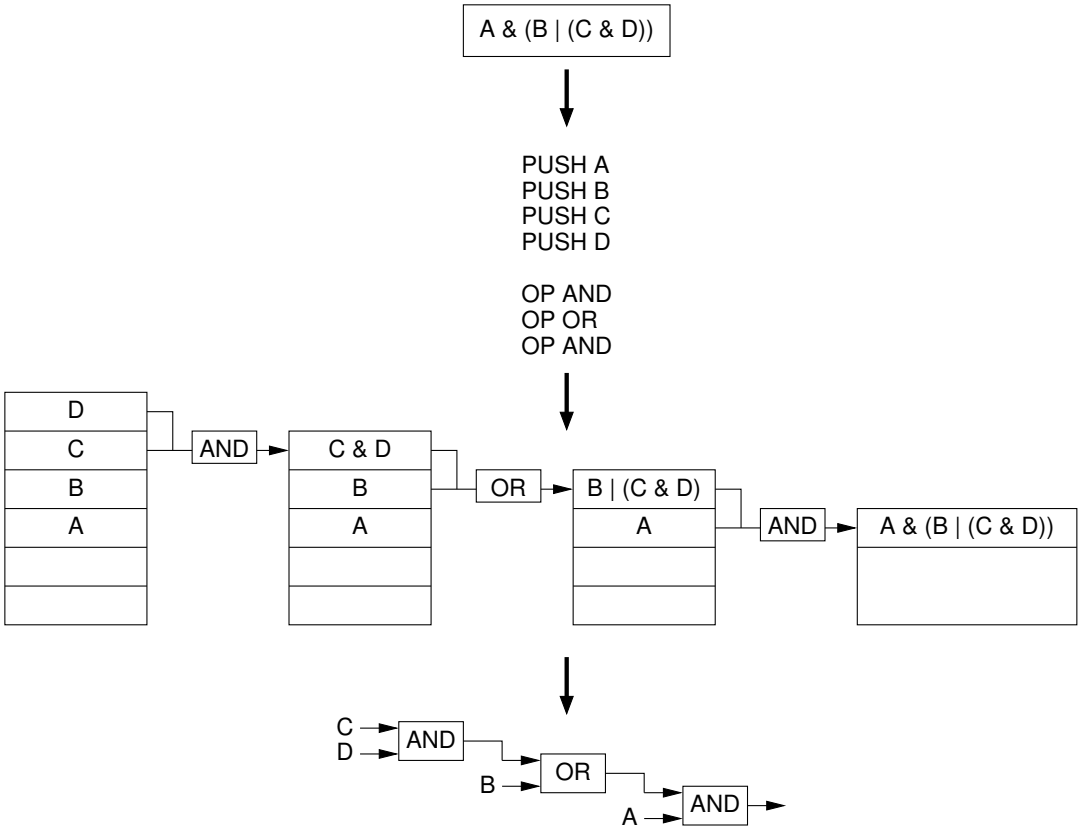


Abbildung 14.2: Die Auswertung von Ausdrücken mittels Stack

14.3.3 Die Anweisungen des Bytecodes

Anweisungen zur Definition von Variablen

- `DEFINE`

  Instanziiert eine Variable des angegebenen Datentyps.

Anweisungen zur Auswertung von Ausdrücken

- `PUSH`

  Die Anweisung `PUSH` legt das Argument auf den Arithmetikstack. Das Argument kann eine Konstante oder eine Variable sein. Bei einer Konstanten wird der entsprechende Wert, bei Variablen die entsprechende Referenz, die einen L- oder R-Wert darstellen kann, auf der obersten Stackposition gespeichert.

- `CLEAR`

  Hiermit wird nach der kompletten Verarbeitung eines Ausdrucks der zuletzt auf dem Stack verbleibende Wert entfernt.

Operatoranweisungen

Eine Operatoranweisung entfernt die benötigten Argumente vom Stack und speichert die Referenz des Ergebnisses an der obersten Stackposition.

Es existieren folgende Operatoranweisungen:

- `OP AND`

  Erzeugt eine UND-Verknüpfung der obersten beiden Stackwerte.

- `OP OR`

  Erzeugt eine ODER-Verknüpfung der obersten beiden Stackwerte.

- `OP ADD`

  Erzeugt eine Addition der obersten beiden Stackwerte.

- `OP SUB`

  Erzeugt eine Subtraktion der obersten beiden Stackwerte.

- `OP MUL`

  Erzeugt eine Multiplikation der obersten beiden Stackwerte.

- `OP DIV`

  Erzeugt eine Division der obersten beiden Stackwerte.

- `OP NEG`

  Erzeugt eine Negierung des obersten Stackwertes.

- `OP NOT`

  Erzeugt eine logische Negierung des obersten Stackwertes.

- `OP LOGAND`

  Erzeugt eine logische UND-Verknüpfung der obersten Stackwerte.

- `OP LOGOR`

  Erzeugt eine logische ODER-Verknüpfung der obersten beiden Stackwerte.

Anweisungen zur Ablaufsteuerung

- `LABEL`

  Die Anweisung `LABEL` definiert eine Sprungmarke zur Ablaufsteuerung.

- `JMP`

  Die Anweisung `JMP` bewirkt einen Abschluß des aktuellen Zustandes und einen unbedingten Zustandsübergang zur angegebenen Sprungmarke.

- `JMPZ` und `JMPNZ`

  Diese Anweisungen bewirken einen Abschluß des aktuellen Zustandes. Sie entfernen den obersten Wert vom Stack, der ein Signal vom Datentyp `bool` darstellt. Ist die Bedingung erfüllt bzw. nicht erfüllt, erfolgt ein Zustandsübergang zur angegebenen Sprungmarke, ansonsten ein Übergang zum nachfolgenden Zustand.

- `BEGINPAR` und `ENDPAR`

  Kennzeichnen den Anfang und den Ende eines explizit parallelen Anweisungsblocks.

14.4 Kontrollanweisungen

14.4.1 Allgemeines

In den folgenden Codebeispielen wird die konkrete Übersetzung der Flußdiagramme in den Bytecode der HVM erläutert. Die prinzipielle Vorgehensweise orientiert sich dabei an den Verfahren, die konventionelle Kompiler anwenden [26, 108].

14.4.2 Die *if*-Anweisung

Mittels `if` lassen sich alternative Ablaufpfade festlegen. Abhängig von der spezifizierten Bedingung wird entweder der erste oder der zweite Pfad ausgeführt. Wird kein zweiter Pfad angegeben, wird bei nicht erfüllter Bedingung keine Anweisung ausgeführt.

```
if (a == b)
  {
    c = 1;
  }
 else
  {
    c = 2;
    d = 1;
  }
```

Für das Prüfen der Bedingung ist im allgemeinen kein eigener Taktschritt erforderlich. Nur, wenn die Bedingung komplexere Anweisungen enthält, die bereits mehrere Takte benötigen, oder der Wert einer der Variablen in einem CLB- oder Block-RAM gespeichert ist, steht das Ergebnis nicht sofort zur Verfügung. Die beiden alternativen Ablaufpfade können unterschiedlich viele Anweisungen enthalten.

Existiert kein zweiter Pfad, so ist festzulegen, ob bei Nichterfüllung der Bedingung ein eigener Taktzyklus benötigt wird oder nicht. Wenn ja, handelt es sich um einen leeren Takt, der lediglich eine Ausführungspause bewirkt. Ein solcher kann sinnvoll sein, um eine Symmetrie der Ablaufpfade zu erreichen. Damit ist die Ausführungszeit der Gesamtanweisung unabhängig von der Bedingung. Um die Gesamtausführungszeit zu verringern, kann es jedoch auch sinnvoll sein, keinen leeren Takt zu implementieren, sondern bei Nichterfüllung direkt mit der nächsten Anweisung fortzufahren.

```
if (a == b)
   c = 1;
```

In einer Hochsprachenimplementierung bietet es sich an, beide Methoden zu unterstützen, z.B. mit einer besonderen Formulierung, wenn ein leerer Takt gewünscht ist:

```
if (a == b)
  { c = 1; }
 else
  { }
```

Abbildung 14.3 zeigt die Übersetzung einer *if*-Anweisung ohne Alternativpfad. Zunächst wird der Ausdruck der Bedingung ausgewertet. Ist das Ergebnis Null, wird der bedingte Pfad übersprungen.

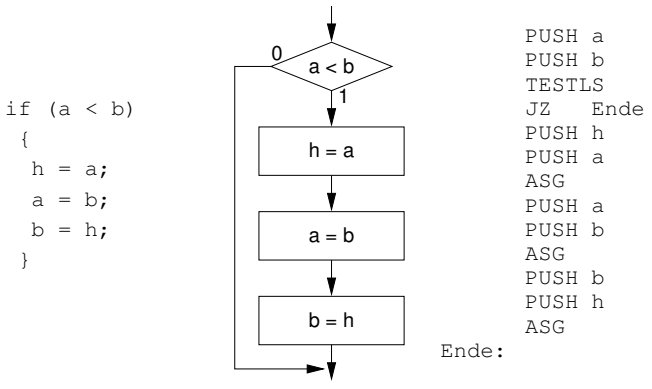


Abbildung 14.3: IF-Anweisung

Abbildung 14.4 zeigt die Übersetzung einer *if*-Anweisung mit Alternativpfad. Zunächst wird wieder der Ausdruck der Bedingung ausgewertet. Ist das Ergebnis Null, wird zum Alternativpfad gesprungen. Am Ende des bedingten Pfades erfolgt ein Sprung zum Ende der Anweisung.

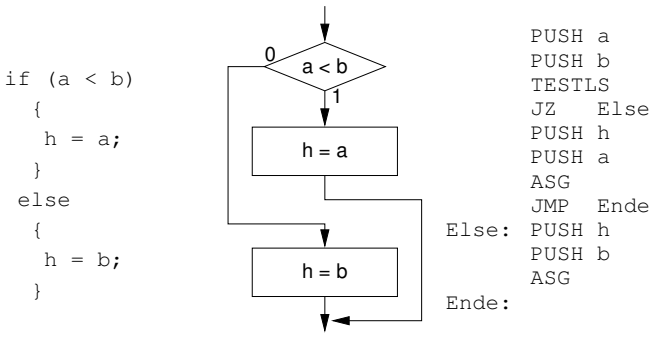


Abbildung 14.4: IF...ELSE-Anweisung

14.4.3 Die *while*-Anweisung

Mit `while` können Schleifen implementiert werden. Der nachfolgende Block von Anweisungen wird wiederholt, solange die Bedingung erfüllt ist.

```
while (a < 10)
 {
   b++;
   a--;
 }
```

Ist die Bedingung bereits zu Beginn nicht erfüllt, kann ohne zusätzlichen Takt direkt mit der nachfolgenden Anweisung fortsetzt werden.

Abbildung 14.5 zeigt die Übersetzung einer *while*-Schleife. Zunächst wird der Ausdruck der Bedingung ausgewertet. Ist das Ergebnis Null, wird der Schleifeninhalt übersprungen. Am Ende der Schleife erfolgt der Rücksprung zur Auswertung der Bedingung (`Loop`).

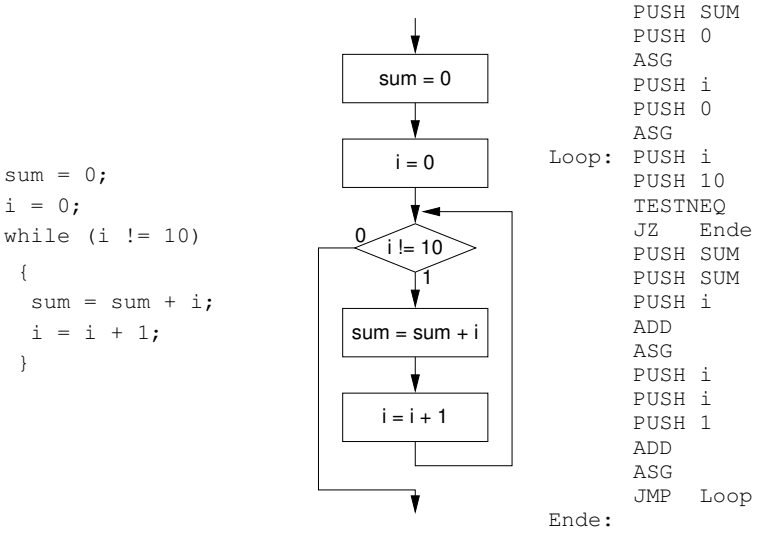


Abbildung 14.5: WHILE-Schleife

14.4.4 Die *do...while*-Anweisung

Auch mit `do...while` lassen sich Schleifen realisieren. Der Unterschied zu `while` besteht darin, daß der nachfolgende Block von Anweisungen in jedem Fall mindestens einmal ausgeführt wird, auch wenn die Bedingung bereits zu Beginn nicht erfüllt ist.

```
do
 {
   b++;
   a--;
 } while (a < 10);
```

Abbildung 14.6 zeigt die Übersetzung einer *do*-Schleife. Zunächst erfolgt die Ausführung des Schleifeninhaltes. Danach wird der Ausdruck der Bedingung ausgewertet. Ist das Ergebnis ungleich Null, erfolgt der Rücksprung zum Beginn der Schleife (`Loop`).

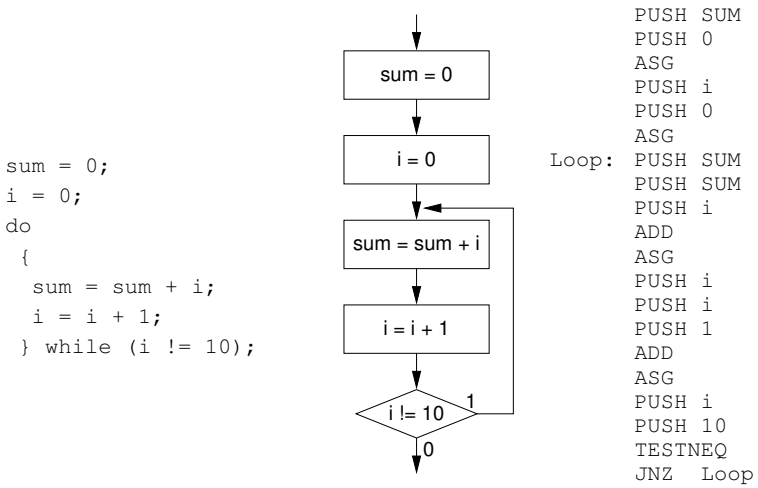


Abbildung 14.6: DO...WHILE-Schleife

14.4.5 Die *for*-Anweisung

Mit `for` lassen sich ebenfalls Schleifen realisieren.

Die Formulierung

```
for (i = 0; i < 10; i++)
 {
   ...
 }
```

ist äquivalent zu folgender Konstruktion, jedoch kompakter:

```
i = 0;
while (i < 10)
 {
   ...
   i++;
 }
```

Abbildung 14.7 zeigt die Übersetzung einer *for*-Schleife. Zunächst wird die Initialisierung ausgeführt. Danach erfolgt die Auswertung der Bedingung. Ist das Ergebnis Null, wird zum

Ende der Anweisung gesprungen, ansonsten der Schleifeninhalt mit anschließender Inkrementierung ausgeführt. Am Ende der Schleife erfolgt der Rücksprung zur Auswertung der Bedingung (`Loop`).

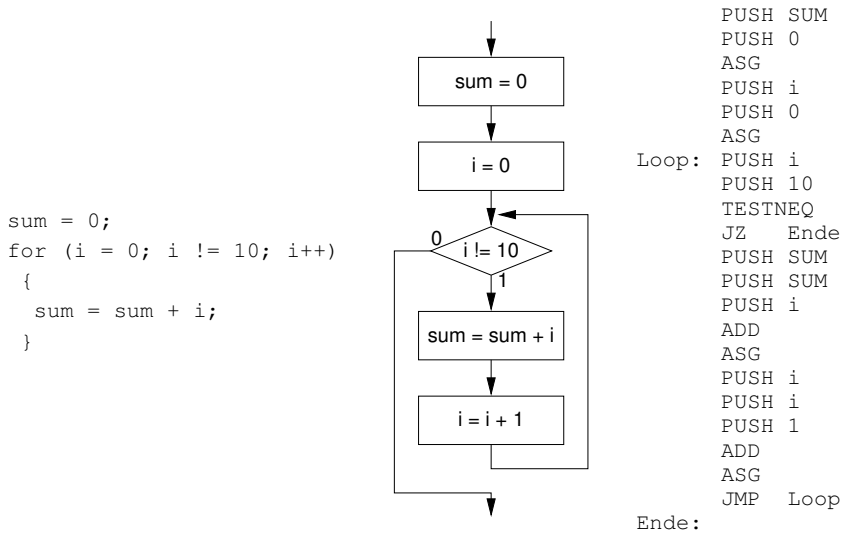


Abbildung 14.7: FOR-Schleife

14.4.6 Die *switch*-Anweisung

`switch` ermöglicht die Angabe einer Vielzahl alternativer Ausführungspfade in Abhängigkeit von einer Bedingung. Im Gegensatz zu den zuvor erläuterten Kontrollanweisungen sind hier jedoch die Bedingungen in der Form eingeschränkt, daß nur der Vergleich eines Ausdrucks mit einem konstanten Wert zulässig ist.

```
switch (a)
 {
   case 0:
     b = 1;
     break;
   case 1:
     b = 2;
     break;
   default:
     b = 0;
 }
```

Die einzelnen Pfade sind jedoch nicht streng alternativ, da die `break`-Anweisungen auch entfallen können. In diesem Fall setzt die Ausführung mit der nachfolgenden Anweisung fort, auch wenn diese zu einem anderen Pfad gehört.

Abbildung 14.8 zeigt die Übersetzung einer *switch*-Anweisung. Zunächst wird geprüft, ob die Bedingung für den ersten Pfad erfüllt ist. Wenn ja, wird dieser ausgeführt, ansonsten erfolgt der Test für den zweiten Pfad. Ist auch dieser nicht erfüllt, wird der *default*-Pfad ausgeführt. *break*-Anweisungen führen zur Beendigung der *switch*-Anweisung.

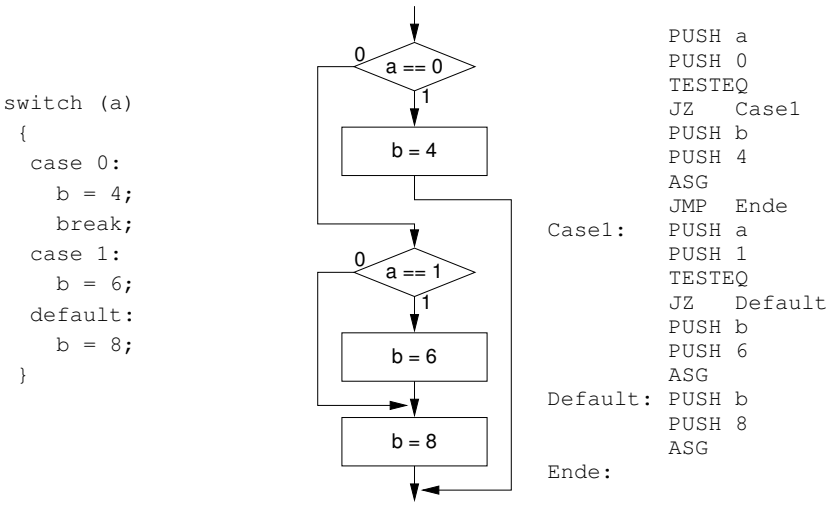


Abbildung 14.8: SWITCH-Anweisung

14.5 Implementierung von Funktionsaufrufen

14.5.1 Allgemeines

Ein zentrales Konzept jeder Hochsprache sind Funktionsaufrufe. In der konventionellen Programmierung ist eine Funktion ein Abschnitt des Programmcodes, der von beliebigen anderen Programmpunkten aus aufgerufen werden kann. Funktionen ermöglichen auf diese Weise, einen Algorithmus, der mehrmals benötigt wird, nur einmal implementieren zu müssen. Außerdem lassen sich Funktionen parametrisieren, d.h. der konkrete Ablauf kann abhängig von den jeweils übergebenen Parametern bei jedem einzelnen Aufruf unterschiedlich sein. Es ist dadurch möglich, mehrere ähnliche Algorithmen einmal durch eine einzige Funktion zu implementieren. Will man das Konzept der Funktionsaufrufe auf die Hardwareebene übertragen, so sind folgende Probleme zu lösen:

- Für jeden Aufruf einer Funktion müssen Verbindungsresourcen angelegt werden, um die Parameter sowie die Handshake-Signale für Aufruf und Fertigstellung zu übertragen. In der konventionellen Programmierung sind diese Verbindungen dagegen virtueller Art.
- Es muß eine Logik bereitgestellt werden, die aus den möglichen Parametern die aktuell benötigten auswählt.
- Es müssen Vorkehrungen für den Fall getroffen werden, daß zwei oder mehrere Kontroller gleichzeitig auf die Funktion zugreifen wollen.
- Nach der Abarbeitung der Funktion muß der korrekte Kontroller von der Fertigstellung benachrichtigt werden.

Die Problematik der Funktionsaufrufe läßt sich in drei Fallgruppen unterteilen, die jeweils unterschiedliche Anforderungen stellen. Um eine effiziente Implementierung zu erreichen, sollte nicht für jeden Funktionsaufruf die komplexeste Implementierung vorgenommen werden.

Denkbar sind folgende Situationen:

- Einfacher Funktionsaufruf.

  Hier wird eine Funktion nur aus einer einzigen anderen Funktion aufgerufen. Der Aufruf kann beliebig oft erfolgen, es ist jedoch immer nur ein einziger Aufruf gleichzeitig aktiv.

- Paralleler Funktionsaufruf.

  Hier ruft eine Funktion zwei oder mehrere andere Funktionen parallel auf. Da nicht garantiert ist, daß beide Funktionen die gleiche Zeit für ihre Ausführung benötigen, muß die aufrufende Funktion warten, bis beide beendet sind. Hierbei ist zu beachten, daß auch die Reihenfolge der Beendigung nicht bekannt ist.

- Aufrufe von gemeinsam genutzten Funktionen.

  Bei diesem Fall kann eine Funktion aus mehreren anderen Funktionen aufgerufen werden, die Aufrufe können auch gleichzeitig erfolgen.

14.5.2 Einfacher Funktionsaufruf

Dieser Fall wird in der Praxis vor allem zur Zerlegung komplexer Funktionen oder zur Auslagerung von mehrfach benötigtem Programmcode eingesetzt.

Im ersten Unterfall wird die Funktion nur ein einziges Mal aufgerufen, hier sind keine besonderen Maßnahmen für die Parameter erforderlich. Diese können durch normale Verbindungen realisiert werden.

Im zweiten Unterfall wird die Funktion mehrfach aufgerufen. Die einzelnen Aufrufe können mit unterschiedlichen Parametern erfolgen. Daher ist zusätzlich ein Multiplexer notwendig, um die jeweils korrekten Parameter auszuwählen.

Abbildung 14.9 zeigt einen einfachen Funktionsaufruf der ersten Kategorie.

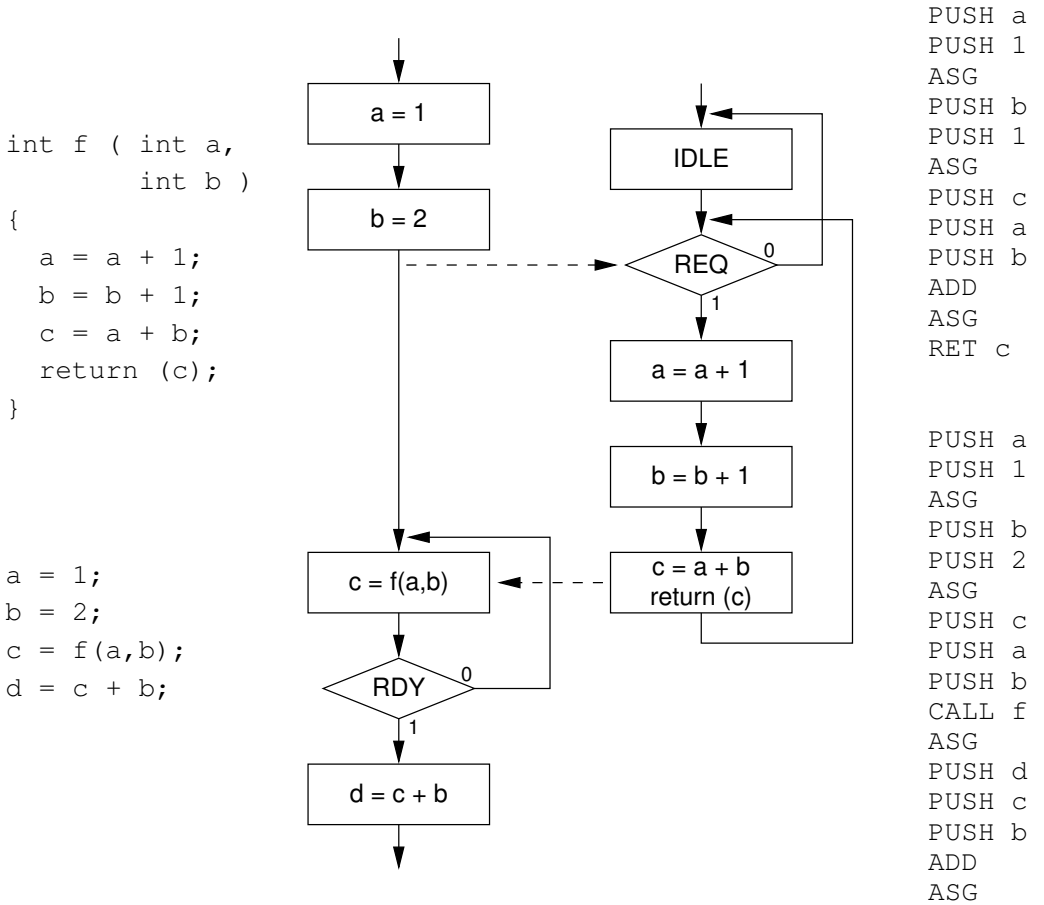


Abbildung 14.9: Funktionsaufruf

14.5.3 Parallele Funktionsaufrufe

Hier werden parallel zwei oder mehrere Funktionen aufgerufen. Der Aufruf und die Parameterauswahl erfolgen wie bei den einfachen Funktionsaufrufen.

Eine besondere Behandlung erfordert die Erkennung der Beendigung. Sowohl die Zeit, die die Funktionen benötigen als auch die Reihenfolge der Beendigung sind nicht bekannt. Daher ist für jede Funktion ein Ende-Flag zu implementieren. Die ausführende Funktion verbleibt in einem Wartezustand, bis alle Flags aktiv sind.
Abbildung 14.10 zeigt einen parallelen Funktionsaufruf mit zwei aufgerufenen Funktionen.

```
a = 1;
b = 2;
par {
  c1 = f1(a,b);
  c2 = f2(a,b);
}
d = c + b;
```

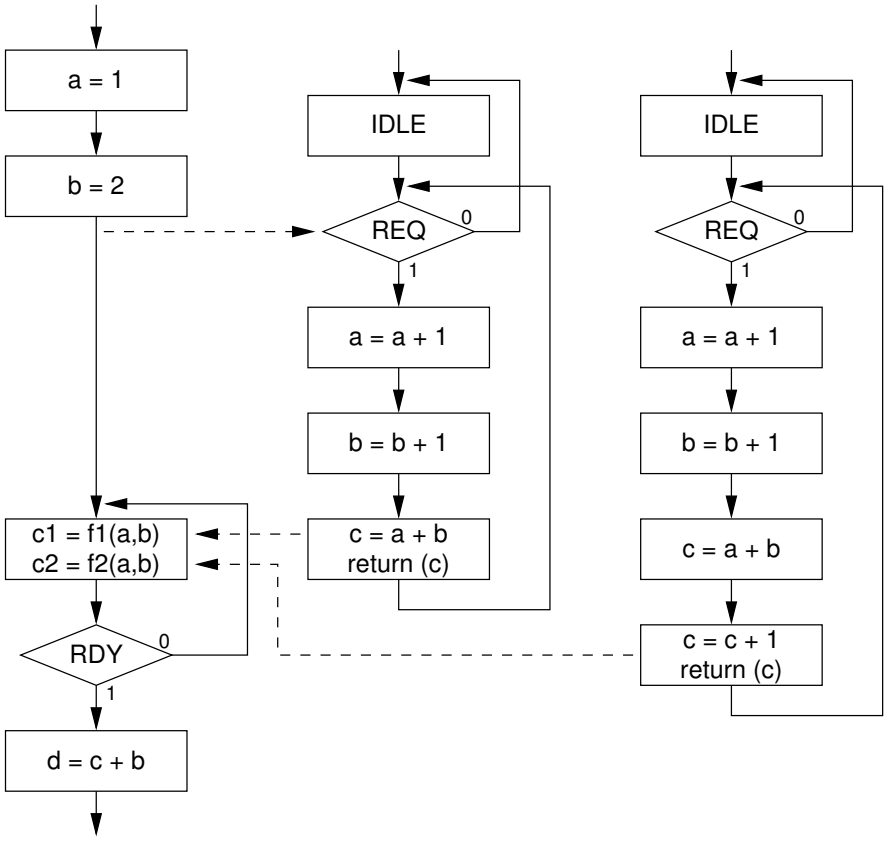


Abbildung 14.10: Parallele Funktionsaufrufe

14.5.4 Aufrufe gemeinsam verwendeter Funktionen

Solange eine Funktion einmal oder mehrmals als einzige eine andere Funktion aufruft, kann vorausgesetzt werden, daß die aufgerufene Funktion zu den erforderlichen Zeitpunkten auch verfügbar ist.

Versuchen jedoch mehrere Funktionen unabhängig voneinander, dieselbe andere Funktion aufzurufen, sind zwei Fälle zu unterscheiden:

- Zugriff auf eine zur Zeit laufende Funktion.

  Die aufgerufene Funktion ist noch durch den Aufruf einer anderen Funktion beschäftigt. Der neue Aufrufer muß warten, bis die Funktion den laufenden Aufruf beendet

hat.

- Gleichzeitiger Zugriff auf die Funktion.

  Hier muß eine Prioritätslogik entscheiden, welche Funktion den Zugriff bekommt, die anderen müssen warten.

Abbildung 14.11 zeigt den vom *CHDL*-System verwendeten Kontroller, der den Aufruf gemeinsamer Funktionen steuert.

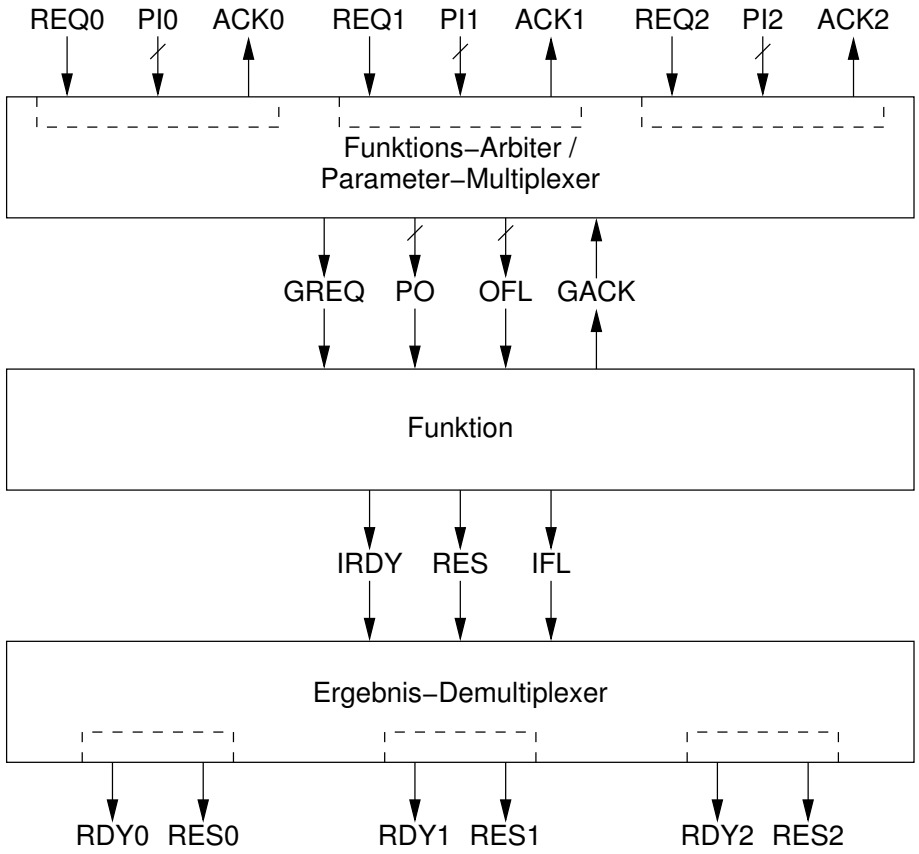


Abbildung 14.11: Kontroller für Funktionsaufrufe

Der Ablauf bei einem Funktionsaufruf ist folgender:

- Der Kontroller der aufrufenden Funktion stellt die Parameter an `PI` bereit und aktiviert für einen Takt sein `REQ`-Signal.
- Der Funktionsarbiter ermittelt, ob die Funktion bereit ist, einen Aufruf anzunehmen. Er prüft weiterhin, ob mehrere `REQ`-Signale aktiviert sind und wählt nach einem festgelegten Prioritätsschema einen Kanal aus.
- Entsprechend dem gewählten Kanal werden über einen Multiplexer die entsprechenden Parameter ausgewählt.
- Das `ACK`-Signal des gewählten Kanals wird für einen Takt aktiviert.
- Das `GREQ`-Signal wird aktiviert, um die Funktion zu starten.
- Die kodierte Kanalnummer wird als `OFL` an die Funktion übergeben.
- Eine nicht pipelinefähige Funktion deaktiviert daraufhin das `GACK`-Signal, um anzuzeigen, daß sie keine weiteren Aufrufe annehmen kann.

- Nach Beendigung aktiviert die Funktion das `IRDY`-Signal und stellt die kodierte Kanalnummer des entsprechenden Aufrufes als `IFL` und das Ergebnis an `RES` bereit.
- Der Ergebnisdemultiplexer leitet in Abhängigkeit von der Kanalnummer das `IRDY`-Signal und das Ergebnis an den entsprechenden Ausgabekanal weiter.
- Die entsprechende aufrufende Funktion übernimmt mit dem `RDY`-Signal die Ergebnisdaten. Wurden gleichzeitig mehrere Funktionen aufgerufen, wartet sie, bis alle aufgerufenen Funktionen ihr `RDY`-Signal aktiviert haben. Dann wird die Ausführung mit dem nächsten Zustand fortgesetzt.

Das Prinzip der gemeinsamen Nutzung von Funktionen weist folgende Vorteile auf:

- Beschränkt vorhandene interne oder externe Resourcen können von mehreren Modulen genutzt werden.

  Die Nutzung der Resourcen kann dabei global oder lokal erfolgen. Bei globaler Nutzung greifen alle Kanäle auf die gesamten Resourcen zu. Bei lokaler Nutzung ist jedem Kanal ein Teil der Resourcen zugeordnet. Auf diese Weise ist es möglich, etwa ein SDRAM so auf mehrere Kontroller zu verteilen, daß jedem ein eigener Adreßraum zugeteilt ist.

- Es sind keine zusätzlichen Synchronisationsmaßnahmen, wie etwa Semaphoren notwendig.

  Alle notwendigen Sicherungsmaßnahmen sind bereits im beschriebenen Kontroller vorhanden, der Entwickler wird insoweit von den notwendigen Implementierungsdetails entlastet.

14.6 Unterstützung für Parallelität

14.6.1 Parallele Anweisungen

Mithilfe der zuvor erläuterten Variablen und Kontrollanweisungen lassen sich bereits beliebige Algorithmen realisieren. Jedoch wird eine ausschließlich hierauf basierende Hardwareimplementierung die Vorteile von FPGAs nicht nutzen können. Die erzeugte Hardwarestruktur folgt dem sequentiellen Konzept der Mikroprozessoren, ist diesen aber aufgrund der geringeren Taktfrequenzen deutlich unterlegen.

Geschwindigkeitsvorteile lassen sich erst erzielen, wenn Anweisungen parallel ausgeführt werden. Dazu müssen die betreffenden Anweisungen jedoch unabhängig voneinander sein.

So können etwa die Anweisungen

```
a = a + 1;
b = b + 1;
```

problemlos parallelisiert werden, nicht jedoch

```
a = a + 1;
b = a * 2;
```

Diese Anweisungen sind nicht unabhängig, insbesondere ist hier die Ausführungsreihenfolge relevant.

14.6.2 Parallele Ablaufpfade

Es können nicht nur einzelne Anweisungen parallelisiert werden, sondern auch komplette Ablaufpfade. Das bedeutet nicht, daß diese Pfade synchron nebeneinander ablaufen müssen, sie können vielmehr völlig unabhängig voneinander ausgeführt werden. Damit lassen sich komplexe Berechnungen parallelisieren, deren Ablauf im Detail stark abweicht.

Diese Form der Parallelisierung läßt sich mit dem Konzept der *Threads* in konventionellen Mehrprozessorsystemen vergleichen.

14.6.3 Synchronisierung paralleler Ablaufpfade

Auch, wenn mehrere Ablaufpfade prinzipiell völlig unabhängig voneinander ausgeführt werden, kann es erforderlich sein, daß sie zu bestimmten Zeitpunkten eine Synchronisierung vornehmen. Notwendig ist dies immer dann, wenn sie auf gemeinsame Resourcen zugreifen oder Daten austauschen wollen. Die Synchronisierung kann mittels gemeinsamen Variablen (*shared memory*) oder Mutex-Elementen erfolgen.

Synchronisierung mittels gemeinsamen Variablen

Im untenstehenden Beispiel verwendet Pfad 2 zur Ausführung Daten, die von Pfad 1 bereitgestellt werden. Hat Pfad 2 die Position erreicht, an dem diese Daten benötigt werden, muß sichergestellt sein, daß Pfad 1 diese bereits zur Verfügung gestellt hat.

Der notwendige Mechanismus läßt sich realisieren, indem Pfad 2 eine Kontrollvariable prüft, die von Pfad 1 beschrieben werden kann. Solange diese Variable nicht einen vereinbarten Wert, z.B. "1" enthält, muß Pfad 2 warten. Pfad 1 setzt diesen Wert, sobald er die Daten bereitgestellt hat. Entsprechend existiert eine gemeinsame Variable in umgekehrter Richtung, mit der Pfad 2 die erfolgte Abnahme der Daten anzeigt.

```
      PFAD 1

while (cnt)
 {
   DIN = cnt;
   DataAvail = 1;
   while (!DataRead);
   DataAvail = 0;
   while (DataRead);
   cnt--;
 }
```

```
      PFAD 2

while (cnt)
 {
   while (!DataAvail);
   sum = sum + DIN;
   DataRead = 1;
   while (DataAvail);
   DataRead = 0;
   cnt--;
 }
```

An diesem Beispiel zeigt sich ein mögliches Problem von Hochsprachenimplementierungen: Durch das notwendige Rücksetzen der Kontrollvariablen ist ein zusätzlicher Taktzyklus erforderlich. Dies reduziert die maximale Datentransferrate auf die Hälfte.

Synchronisierung mittels Mutex-Elementen

Mutex-Elemente stellen erweiterte Funktionen zur Verfügung, um den Zugriff auf gemeinsame Resourcen abzusichern.

Jeder zu synchronisierende Prozeß besitzt einen kritischen Bereich, der nur jeweils von einem Prozeß ausgeführt werden darf. Das zuvor erläuterte Modell der gemeinsamen Variablen kann hier nicht angewendet werden, denn bei diesem kann eine Kontrollvariable nur von einem einzigen Prozeß beschrieben werden.

Mit Mutex-Elementen läßt sich einfach ermitteln, ob sich bereits ein anderer Prozeß im kritischen Bereich befindet. Außerdem können sie mittels einer Prioritätsregelung die Situation auflösen, daß mehrere Prozesse zeitgleich in den kritischen Bereich eintreten wollen.

```
      PFAD 1                              PFAD 2

while (cnt)                          while (cnt)
 {                                    {
   ...                                  ...
   tryLock();                           tryLock();
   ...                                  ...
   ...                                  releaseLock();
   releaseLock();                       ...
   cnt--;                               cnt--;
 }                                    }
```

14.6.4 Spezifizierung von Parallelität

Insbesondere bei der Implementierung von Protokollen kann es erforderlich sein, die Zuordnung der Anweisungen zu den Taktzyklen nicht einer automatischen Optimierung zu überlassen, sondern explizit anzugeben.

Zum Zusammenfassen mehrerer Anweisungen zu einem parallelen Block sind verschiedene Schreibweisen denkbar, so etwa:

```
par { a = a + 1;          oder:        [ a = a + 1;
      b = b * 5;                         b = b * 5;
      c = c / 2; }                       c = c / 2; ]
```

Der Hochsprachenkompiler sollte mit einer entsprechenden Warnung reagieren, wenn Anweisungen innerhalb eines solchen Blocks nicht parallelisierbar sind.

14.6.5 Implizite Parallelität

Bei Nutzung der impliziten Parallelität werden unabhängige Anweisungen automatisch parallel ausgeführt. Sobald eine Anweisung folgt, die eine Datenabhängigkeit zu den vorigen Anweisungen besitzt, wird der aktuelle Zustand geschlossen und ein neuer geöffnet. Eine Datenabhängigkeit liegt vor, wenn eine Variable verwendet wird, die in einem der vorigen Zustände verändert wurde oder eine weitere Zuweisung an eine gleiche Variable erfolgt.

14.6.6 Explizite Parallelität

Bei Nutzung der expliziten Parallelität kann der Anwender mittels eckiger Klammern selbst festlegen, welche Anweisungen parallel ausgeführt werden sollen. Es sind nur aufeinanderfolgende Anweisungen parallelisierbar. Auf diese Weise kann das Zeitverhalten der Funktion genau festgelegt werden um z.B. synchrone Protokolle zu implementieren.

```
  A = 1;
[ B = 2;
  C = 2; ]
  D = 3;
```

14.6.7 Realisierung von Mutex-Elementen

Abbildung 14.12 zeigt die Implementierung von Mutex-Elementen. Solange eine andere Funktion die Sperre hält, verbleibt der Kontroller im Wait-Zustand.

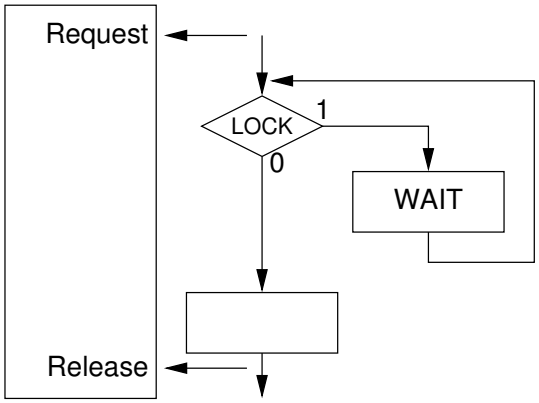


Abbildung 14.12: Mutex-Element

14.7 Unterstützung für Pipelining

14.7.1 Implementierung von Pipelines

Pipelining stellt neben der Parallelisierung ein weiteres Verfahren dar, die Ausführungsgeschwindigkeit von Algorithmen zu erhöhen [11, 48, 101].

Hierzu wird der auszuführende Algorithmus in mehrere Stufen zerlegt. Eingehende Daten müssen zur kompletten Verarbeitung alle Stufen durchlaufen. Mit jedem Taktzyklus kann ein neues Datenwort in die Anordnung eintreten, alle Stufen arbeiten dabei parallel. Nach einer Anfangsverzögerung kann mit jedem Takt ein komplett bearbeitetes Datum abgegeben werden.

In der konventionellen Programmierung gibt es keine Entsprechung zum Pipelining. Es läßt sich nur in speziellen Hardwarestrukturen realisieren.

Das Pipeline-Konzept ist mit konventionellen Programmiersprachen nur schwer darstellbar. Der Grund liegt darin, daß dort jeder Thread eine eindeutige aktuelle Ablaufposition besitzt, die durch den Programmzähler festgelegt wird. Beim Pipelining existieren jedoch mehrere solche aktiven Ablaufpositionen. Eine gepipelinte Funktion kann bereits neue Daten aufnehmen, bevor die vorhergehenden komplett bearbeitet wurden. Dies ist mit dem Prinzip der *von-Neumann*-Architektur nicht vereinbar.

Dennoch ist das Pipeline-Verfahren für die Beschleunigung von Algorithmen von zu großer Bedeutung, um es vom Konzept der Hochsprachen auszuschließen. Es stellt sich jedoch die Frage, wie Pipelining in sinnvoller Weise in einer Hochsprachenbeschreibung angewendet werden kann. Die folgenden Ausführungen versuchen dies zu verdeutlichen.

14.7.2 Bedeutung der Datenabhängigkeiten

Zur Parallelisierung von Anweisungen ist im Normalfall erforderlich, daß diese voneinander unabhängig sind. Anweisungen sind dann abhängig, wenn eine nachfolgende Anweisung das Ergebnis einer vorangehenden verwendet.

Pipelining erlaubt eine Parallelisierung auch in den Fällen, in denen die Anweisungen voneinander abhängig sind. Es müssen jedoch zusätzliche Maßnahmen getroffen werden. Die parallele Ausführung kann ermöglicht werden, indem die Ergebnisse der einzelnen Anweisungen in spezielle lokale Variablen der jeweiligen Pipeline-Stufe geschrieben werden. Die nachfolgenden Anweisungen verwenden dann nicht die Originalvariablen, sondern die lokalen Variablen der vorangehenden Pipeline-Stufe.

Anzahl und die Anordnung der Pipeline-Stufen werden durch die Ausführungsreihenfolge der abhängigen Anweisungen festgelegt.

14.7.3 Pipelining in Hochsprachenbeschreibungen

Im Zusammenhang mit Hochsprachen existieren mehrere Möglichkeiten, Pipeline-Anordnungen zu erzeugen:

- Strukturelle Darstellung.

  Die Pipeline-Struktur wird mittels Variablenfeldern und parallelen Anweisungen realisiert. Diese Form wird im System *Handel-C* eingesetzt.

```
for (i = 1; i < nr; i++)
 {
   par
    { A[0] = D[i];
      A[1] = A[0];
      A[2] = A[1] + A[0];
      if (A[2] > 100)
          A[3] = A[2] - 100;
        else
          A[3] = A[2];
      S[i] = A[3]; }
 }
```

  Diese Darstellung ist mit dem konventionellen Modell vereinbar, da sie nur eine aktive Ausführungsposition besitzt, auch wenn dabei mehrere Anweisungen zeitgleich bearbeitet werden. Nachteil dieses Verfahrens ist jedoch, daß der Entwickler die Pipeline-Stufen und die Datenabhängigkeiten manuell implementieren muß.

- Datenfluß-Methode.

  Das konventionelle Modell wird aufgegeben und die Schaltung nach dem Datenflußmodell konstruiert. Hier existiert keine aktuelle Ausführungsposition, sondern die Daten durchlaufen die Anordnung selbstständig.

  Ein Beispiel für eine datenflußorientierte Hochsprache ist *ppC* [123].

  Beim Datenflußmodell erhält die Beschreibung einen strukturellen Charakter. Es werden Berechnungen durchgeführt und die Ergebnisse an Variablen zugewiesen. Zusätzlich wird vom Hardwarekompiler eine Flußkontrolle implementiert. Diese stellt zum einen sicher, daß Berechnungen erst durchgeführt werden, wenn alle Operanden verfügbar sind. Zum anderen erhält jede Variable eine Markierung, die anzeigt, ob sie gültige Daten enthält.

  Eine solche Anordnung kann auf flexible Weise auch auf unregelmäßig eintreffende Daten reagieren. Eingehende Daten werden verarbeitet, sobald alle zusätzlich notwendigen Daten ebenfalls verfügbar sind und die nachfolgenden Stufen neue Daten aufnehmen können.

  Da ein Gesamtdesign in der Regel nicht nur datenflußorientierte, sondern auch konventionelle Problembereiche enthält, muß die Hochsprache beide Konstruktionsmöglichkeiten zur Verfügung stellen. Dies kann beispielsweise getrennt nach Prozessen mit speziellen Schlüsselwörtern erfolgen.

```
process _dataflow func1()
{
  A0 = D[i];
  A1 = A0 + D[i];
  if (A1 > 100)
      S[i] = A1 - 100;
    else
      S[i] = A1;
}
```

Das Problem bei diesem Verfahren besteht darin, daß die Implementierung der flexiblen Kontroll-Logik aufwendig sein kann und mit zunehmender Anzahl von Stufen lange kombinatorische Pfade verursacht.

- Automatisierte Berücksichtigung der Datenabhängigkeiten.

  In einer Hochsprachenbeschreibung, die eine Wiederholungsanweisung enthält

```
while (...)
 {
   Anweisung_A;
   Anweisung_B;
   Anweisung_C;
 }
```

  wird zunächst ermittelt, welche Anweisungen aufgrund fehlender Datenabhängigkeiten parallelisiert werden können. Dadurch ergeben sich mehrere Blöcke, die jeweils aus einer oder mehreren parallel ausführbaren Anweisungen bestehen. Die Anzahl dieser Blöcke legt direkt die notwendigen Taktzyklen für einen Schleifendurchlauf fest.

  Die Pipeline-Struktur wird gebildet, indem diese Blöcke parallel ausgeführt werden, die zu verarbeitenden Daten jedoch jeweils um einen Taktzyklus versetzt werden. Dazu müssen alle Variablen, die in den späteren Stufen verwendet werden, durch Verzögerungsregister geführt werden. Dieses Einfügen von Verzögerungsregistern bewirkt zusätzlich eine Verkürzung der Logikpfade.

  Abbildung 14.13 zeigt das Schema dieses Verfahrens:

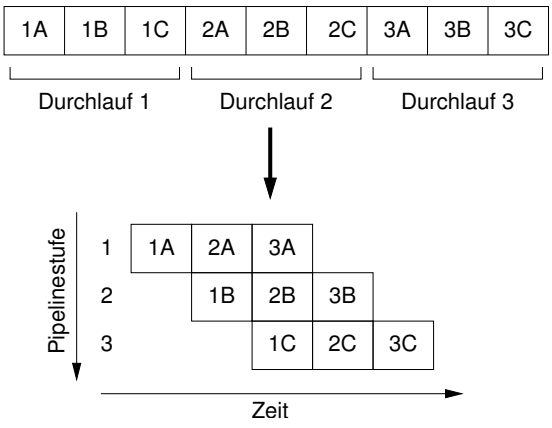


Abbildung 14.13: Aufbau der Pipeline-Struktur

Die konkrete Realisierung dieses Verfahrens kann auf verschiedene Arten erfolgen. Die bei *CHDL* eingesetzte Methode wird später im Detail erläutert.

14.7.4 Nicht pipelinefähige Algorithmen

Es gibt Algorithmen, die nicht als Pipeline implementiert werden können. Bei dem folgenden Programmcode einer vereinfachten Division (Abb. 14.14) ist die Anzahl der Takte, die für die Bearbeitung eines Datums erforderlich ist, vom Datum selbst abhängig.
Bei Verwendung der bereits erläuterten erweiterten Flußdiagramme würden konkret folgende Probleme auftreten:

- Neu eintretende Daten würden die Werte von a und d verändern, die von den zuvor eingetretenen Daten noch benötigt werden.

```
int Func2 (int a, int b )
{
 int d;

 d = 0;
 while (a >= 0)
  {
   a = a - b;
   d = d + 1;
  }
 d = d - 1;
 return (d)
}
```

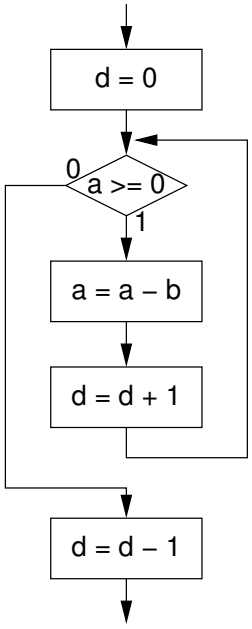


Abbildung 14.14: Nicht pipelinefähige Funktion

- Das Ausführungstoken der neu eintretenden Daten würde mit dem Token des vorigen Datums, das noch in der Schleife rotiert, verschmelzen. Dadurch ginge die Information, daß zwei Tokens aktiv sind, verloren.

Da die Anzahl der Takte, die die Funktion in der Schleife verbringt, von den konkret eintretenden Daten abhängt, kann die Entscheidung, wann neue Daten eintreten können, erst zur Laufzeit getroffen werden. Um die Probleme zu vermeiden, müsste ein Mechanismus implementiert werden, der die Verfügbarkeit der Funktion für neue Daten regelt. Im vorliegenden Algorithmus könnten jedoch dadurch keine Vorteile durch Pipelining erreicht werden.

Eine Möglichkeit für einen solchen Regelmechanismus zeigt Abbildung 14.15. Beim Eintritt in den nicht pipelinefähigen Teil der Funktion wird das Signal ACK gesperrt. Die Freigabe erfolgt erst, wenn der Ablauf den kritischen Teil wieder verläßt. Dies entspricht dem Einsatz eines Mutex-Elementes in der konventionellen Thread-Programmierung.

Um einen Algorithmus sinnvoll in eine Pipeline-Struktur umsetzen zu können, müssen folgende Voraussetzungen vorliegen:

- Neu eintretende Daten dürfen keine Werte überschreiben, die von nachfolgenden Pipeline-Stufen noch benötigt werden.
- Aktuelle Daten, die später noch benötigt werden, werden erhalten, indem sie als Kopien durch die folgenden Stufen geschoben werden.
- Die Ausführungszeit in Taktzyklen muß für alle Ausführungstokens gleich sein. Ist dies nicht der Fall kommt es zu einem versuchten "Überholen" einzelner Daten, was zur Folge hat, das Ausführungstokens miteinander verschmelzen und damit Daten verloren werden.
- Die möglichen Ablaufpfade der Ausführungstokens müssen so gestaltet sein, daß es an keiner Stelle zu einem Verschmelzen von Tokens kommen kann. Kritisch sind insbesondere Wiederholungsanweisungen.

In den folgenden Ausführungen wird ein Verfahren vorgestellt, nach dem innerhalb des *CHDL*-Systems automatisch Pipeline-Strukturen erzeugt werden können.

In einer Hochsprachenbeschreibung, die eine Wiederholungsanweisung enthält

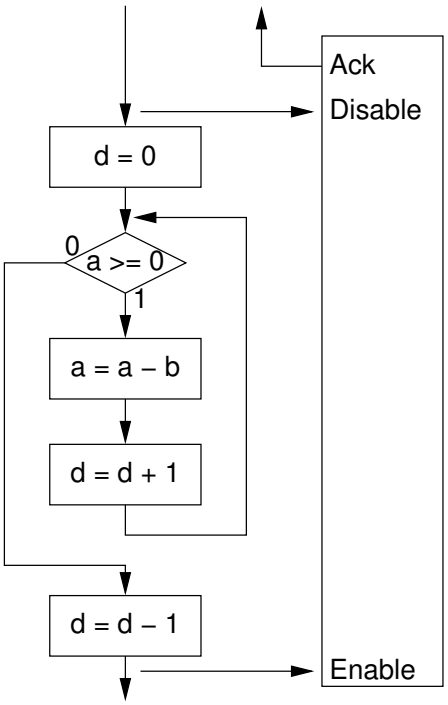


Abbildung 14.15: Regelmechanismus für `Func2`

```
while (...)
 {
   Anweisung1;
   Anweisung2;
   Anweisung3;
 }
```

wird zunächst ermittelt, welche Anweisungen aufgrund fehlender Datenabhängigkeiten parallelisiert werden können. Dadurch ergeben sich mehrere Blöcke, die jeweils aus einer oder mehreren parallel ausführbaren Anweisungen bestehen. Die Anzahl dieser Blöcke legt direkt die notwendigen Taktzyklen für einen Schleifendurchlauf fest.

Ziel des Pipelining ist, diese Anzahl Taktzyklen auf Eins zu reduzieren.

Die Pipeline-Struktur wird gebildet, indem die einzelnen Blöcke parallel ausgeführt werden, die zu verarbeitenden Daten jedoch jeweils um einen Taktzyklus versetzt werden. Dazu müssen alle Variablen, die in den späteren Stufen verwendet werden, durch Verzögerungsregister geführt werden. Genau dieses Einfügen von Verzögerungsregistern bewirkt die erwünschte Verkürzung der Logikpfade.

Im Beispielcode

```
while (i < 100)
 {
   b = a + 1;
   c = a * 2;
   d = b + c;
   e[i] = d * a + b;
   i++;
 }
```

werden zunächst die Blöcke gebildet. Deren Anzahl bestimmt die später nötige Zahl der Pipeline-Stufen.

```
while (i < 100)
 {
   parallel { b = a + 1; c = a * 2; }
   parallel { d = b + c; }
   parallel { e[i] = d * a + b; i++; }
 }
```

Danach wird die zeitliche Verschiebung durch die Verzögerungsregister implementiert:

```
while (i < 100)
 {
   parallel { b = a + 1; c = a * 2; a1 = a; i1 = i; }
   parallel { d = b + c;            a2 = a1; b1 = b; i2 = i1;}
   parallel { e[i2] = d * a2 + b1; i++; }
 }
```

Die entstandenen Anweisungen können nun alle parallel ausgeführt werden:

```
while (i < 100)
 {
   parallel
    { b = a + 1; c = a * 2; a1 = a; i1 = i;
      d = b + c;            a2 = a1; b1 = b; i2 = i1;
      e[i2] = d * a2 + b1; i++; }
 }
```

Zum Schluß muß noch das nachträgliche Leeren der Pipeline berücksichtigt werden:

```
while (i < 100)
 {
   parallel
    { b = a + 1; c = a * 2; a1 = a; i1 = i;
      d = b + c;            a2 = a1; b1 = b; i2 = i1;
      e[i2] = d * a2 + b1; i++; }
 }
parallel
 { d = b + c;            a2 = a1; b1 = b; i2 = i1;
   e[i2] = d * a2 + b1; i++; }
parallel
 { e[i2] = d * a2 + b1; i++; }
```

Bei diesem Verfahren bestehen noch folgende Schwierigkeiten:

- Bei n Pipeline-Stufen steht der erste Ergebniswert erst nach dem n-ten Taktzyklus zur Verfügung, die ersten $n-1$ Durchläufe sind ungültig. Beim obigen Beispiel wirkt sich dies in der noch nicht initialisierten Indexvariablen `i2` aus.
- Variablen, die innerhalb der Pipeline-Stufe 2 oder höher verändert werden, können nicht ohne weiteres in der Abbruchbedingung eingesetzt werden.

Diese Probleme können durch Einfügen geeigneter Kontrollvariablen gelöst werden. Jeder Pipeline-Stufe wird eine Variable zugeordnet, die angibt, ob die in dieser Stufe verwendeten Werte gültig sind. Die erste Stufe wird nur ausgeführt, wenn die `while`-Bedingung erfüllt ist. Das Ergebnis dieser Bedingung wird dann durch die Kette der Kontrollvariablen geführt. Auf diese Weise kann im Beispiel die Ausführung der Anweisung `e[i2] = d * a2 + b1`

in den ersten beiden Taktzyklen verhindert werden. Weiterhin sorgt die Konstruktion dafür, daß die `while`-Schleife erst nach komplettem Leeren der Pipeline verlassen wird. Auch der Inhalt der Zählvariablen `i` hat am Ende der Schleife den erwarteten Wert 100, was in den obigen Versionen des Beispielcodes noch nicht der Fall wäre.

```
v2 = 0; v3 = 0;
while ((i < 100) || (v2 || v3))
 {
   parallel
    {
      v2 = (i < 100); v3 = v2;
      a2 = a1; b1 = b; i2 = i1;
      if (i < 100)
         b = a + 1; c = a * 2; a1 = a; i1 = i;
      if (v2)
         d = b + c; a2 = a1; b1 = b; i2 = i;
      if (v3)
         e[i2] = d * a2 + b1;
      if (i < 100)
         i++; }
 }
```

Anweisungen, die keine Abhängigkeiten von Ergebnissen der vorangehenden Stufen enthalten, werden unter der `while`-Bedingung ausgeführt. Dies bewirkt eine faktische Verlagerung in die erste Pipeline-Stufe.

Der entstandene Code weist nach dem Einfügen der Kontrollvariablen sowohl Elemente von Parallelität als auch von sequentieller Ausführung auf: Prinzipiell werden alle Anweisungen zeitgleich ausgeführt. In den Phasen des Füllens und des Leerens der Pipeline wird die Ausführung zusätzlich durch die weitergeschobenen Zustände der Kontrollvariablen gesteuert.

Durch die Kette der Kontrollvariablen erhält man de facto eine sequentielle Anordnung mit mehreren aktiven Ausführungspositionen. Durch eine Erweiterung des konventionellen Ausführungsmodells, das nur eine aktive Ausführungsposition besitzt, kann eine vereinfachte Ausdrucksweise erreicht werden. Abbildung 14.16 zeigt den ursprünglichen Programmcode sowie die resultierende Implementierung in Form eines erweiterten Flußdiagramms.

Das Ausführungsmodell dieses Diagramms weist folgende Besonderheiten auf, die in konventionellen Flußdiagrammen nicht zulässig sind:

- Nach dem Zustand S2 findet eine Aufspaltung des Ausführungstokens statt. Nach S2 wird sowohl S3 aktiviert als auch S2, solange die `while`-Bedingung erfüllt ist. Dadurch können gleichzeitig mehrere der Zustände S2, S3 und S4 aktiv sein. Dies realisiert die Pipeline-Funktion.

- Nach dem Zustand S4 wird das betreffende Ausführungstoken gelöscht. S4 besitzt keinen Nachfolgezustand.

- Sobald die `while`-Bedingung nicht mehr aktiv ist, verbleibt die Anordnung noch solange in Zustand S5, bis keiner der Zustände S3 und S4 mehr aktiv ist. Dies stellt die Phase des Leerens der Pipeline dar. Um die notwendigen Bedingungen zu realisieren, müssen die Zustände der Ausführungstokens verwendet werden können.

14.8 Die Schnittstelle zur strukturellen *CHDL*-Hardwarebeschreibung

14.8.1 Allgemeines

Die hochsprachenorientierte Hardwarebeschreibung kann auf einfache Weise mit der strukturellen Ebene von *CHDL* kombiniert werden. Jeder Prozeß der Hochsprachenbeschreibung

```
i = 0;
while (i < 100)
 {
  b = a + 1;
  c = a * 2;
  d = b + c;
  e[i] = d * a + b;
  i = i + 1;
 }
```

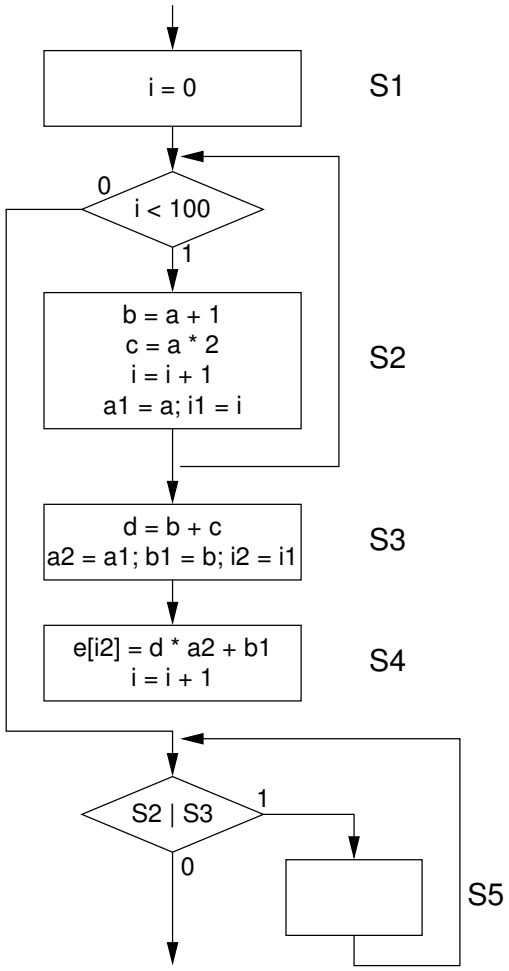


Abbildung 14.16: Ablauf in Pipeline-Form

legt die Implementierung eines Moduls fest. Bei der Verschaltung der Module ist dann unerheblich, ob diese mittels struktureller Beschreibung, Zustandsmaschinenbeschreibung oder Hochsprache realisiert wurden. Damit stehen im *CHDL*-System alle Ebene der Hardwarebeschreibung gleichberechtigt nebeneinander.

Zum Datenaustausch mit anderen Modulen werden Streams verwendet. Die nötige Flußkontrolle zum Lesen und Schreiben dieser Streams wird von der HVM automatisch erzeugt.

Im folgenden Anwendungsbeispiel wird ein Anwendungsbeispiel für die Einbindung eines Hochsprachenmoduls vorgestellt.

14.8.2 Anwendungsbeispiel

Der untenstehende Prozeß liest ein Datenwort vom Eingabe-Stream `I` ein. Innerhalb einer Schleife wird die Summe aller Zahlen von Eins bis zu diesem Wert berechnet. Der Ausgabe-Stream `O` erhält dieses Ergebnis.

Diese Hochsprachenbeschreibung befindet sich in einer eigenen Datei (hier: `func1.c`), die vom *CHDL*-Hochsprachenkompiler bearbeitet wird.

```
process1 ( istream I:10, ostream O:10 )
{
  reg  A:10,B:10,SUM:10;

  SUM = 0;
  A   = I;
```

```
  while (A)
   {
     SUM = SUM + A;
     A = A - 1;
   }

  O = SUM;
}
```

Die Einbindung der Datei `func1.c` erfolgt innerhalb eines *CHDL*-Moduls, das den Hochsprachenkompiler explizit startet. Besonderes Merkmal bei *CHDL* ist, daß dieser Kompiler keine eigenständige Anwendung, sondern Teil der *CHDL*-Klassenbibliothek ist.

Die Bearbeitung der Hochsprachenbeschreibung erfolgt in zwei Stufen. Zunächst wird der Hochsprachenparser gestartet, der den Zwischencode für die HVM generiert. Im nächsten Schritt wird die HVM ausgeführt. Sie erzeugt direkt die entsprechenden strukturellen *CHDL*-Bauteile.

```
class Part1 : public BaseSM
{
public:
  DynInterface  Port;

  Part1 ( const char* Name );
};

Part1::Part1 ( const char* Name )
  : BaseSM(Name),
    Port(this)
{
  HParser     P;
  uint32      retcode;
  Interpreter I;

  ProcessName  _N(this);

  retcode = P.Parse("func1.c");
  if (retcode)
   {
     printf("Fehler: <%s>\n",P.ErrorText(retcode));
     exit(1);
   }

  I.Execute(P.GetCodeBufferAdr(),P.GetCodeBufferLen(),
            stdout,this,&Port,C);

  Generate();
}
```

Das Modul besitzt ein dynamisches Pin-Interface, das von der Hochsprachenbeschreibung implementiert wird. Über dieses Interface kann das Modul später mit dem Gesamtdesign verschaltet werden. Die Anbindung erfolgt dabei über die Funktionen `ConnectToIStream` und `ConnectToOStream`:

```
  NodeArray  CIN(10,"CIN");
```

```
  Node       CIN_DIV("CIN_DIV");
  Node       CIN_DIA("CIN_DIA");
  NodeArray  COUT(10,"COUT");
  Node       COUT_DOV("COUT_DOV");
  Node       COUT_DOA("COUT_DOA");
  Part1      P1("P1");

  P1.Port.ConnectToIStream("I",CIN,CIN_DIV,CIN_DIA);
  P1.Port.ConnectToOStream("O",COUT,COUT_DOV,COUT_DOA);
}
```

In einem Gesamtdesign verhält sich dieses Modul genau wie andere Module, die direkt mittels strukturellem *CHDL*-Code implementiert sind. Es kann somit auf die gleiche Weise simuliert und synthetisiert werden.

Bei der Synthese des Gesamtdesigns kann der Hochsprachenkompiler detaillierte Informationen über die Implementierung des bearbeiteten Prozesses ausgeben. Dies umfasst auch die Anweisungen, die der HVM zur Realisierung der Hardwarestruktur übergeben werden.

```
 LABEL-TABELLE
---------------
>process1<        00000020  00000000 0  int process1(istream I:10,
                                                    ostream O:10);

 TYP-TABELLE
-------------

 Typ-Definitionen
--------------------
0020 : FUNCTION,  Typ : 00000002   Elemente :   0  int(istream I:10,
                                                       ostream O:10);

 Definierte Funktionen
-----------------------
int process1(istream I:10,ostream O:10)
 Parameter
>I<               40000013  00000000 0  istream I:10;
>O<               40000014  00000001 0  ostream O:10;

 Lokale Variablen
>A<               40000011  00000002 0  reg A:10;
>B<               40000011  00000003 0  reg B:10;
>SUM<             40000011  00000004 0  reg SUM:10;


PC_SEQUENTIAL
PC_VARIABLE <0>
PC_NAME = <I>
PC_VARIABLE <1>
PC_NAME = <O>
PC_VARIABLE <2>
PC_NAME = <A>
PC_VARIABLE <3>
PC_NAME = <B>
PC_VARIABLE <4>
```

```
PC_NAME = <SUM>
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000004 (SUM)
PC_PUSH   Type = 00000002  TypeExt = 00000000 Value = 00000000 (#0)
PO_ASG    Type = 00000002
PC_CLEAR
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000002 (A)
PC_PUSH   Type = C0000013  TypeExt = 0000000A Value = 00000000 (I)
PO_ASG    Type = 00000002
PC_CLEAR
PC_LABEL  Dest = 00000000
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000002 (A)
PC_PUSH   Type = 00000002  TypeExt = 00000000 Value = 00000000 (#0)
PO_EQUAL  Type = 00000000
PC_JMPNZ  Dest = 00000001
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000004 (SUM)
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000004 (SUM)
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000002 (A)
PO_ADD    Type = 00000002
PO_ASG    Type = 00000002
PC_CLEAR
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000002 (A)
PC_PUSH   Type = C0000011  TypeExt = 0000000A Value = 00000002 (A)
PC_PUSH   Type = 00000002  TypeExt = 00000000 Value = 00000001 (#1)
PO_SUB    Type = 00000002
PO_ASG    Type = 00000002
PC_CLEAR
PC_JMP    Dest = 00000000
PC_LABEL  Dest = 00000001
```

Kapitel 15
Vergleich mit anderen Hochsprachensystemen

15.1 *SystemC*

Die Hardwarebeschreibung von *SystemC* ist so beschaffen, daß sie mit einem handelsüblichen Kompiler und der *SystemC*-Bibliothek simuliert werden kann. Daraus ergeben sich sowohl die Grundstruktur als auch einige Einschränkungen.

Das gewünschte Verhalten der einzelnen Komponenten kann entweder mittels Methoden (`SC_METHOD`) oder Threads (`SC_THREAD`, bzw. `SC_CTHREAD`) modelliert werden. Diese enthalten das Verhalten für einen Takt (Methoden) oder für mehrere Takte (Threads). In Methoden sind daher keine Endlosschleifen zulässig, in Threads wird die Funktion `wait` zur Aufteilung in einzelne Takte verwendet.

Daraus ergibt sich, daß in *SystemC* Formulierungen wie

```
s = 0; i = 0;
while (i < 10)
 {
   s = s + i; i = i + 1;
 }
```

nicht zulässig sind. Die korrekte Hardwarebeschreibung für diese Summenbildung in einem Thread würde lauten:

```
s = 0; i = 0;
wait(...);
while (i < 10)
 {
   s = s + i; i = i + 1;
   wait(...);
 }
```

Der Anwender muß die Aufteilung auf die einzelnen Taktzyklen manuell vornehmen.

Nur die Threads enthalten einen eigenen Ausführungspfad und sind somit für hochsprachenorientierte Beschreibungen geeignet. Durch die Notwendigkeit der `wait`-Funktion ist die Beschreibung jedoch umfangreicher als eine entsprechende in *CHDL*.

Das Verfahren, mit dem diese Beschreibung konkret umgesetzt wird, bleibt dem Entwickler jedoch verborgen. Er kann dadurch nur schwer abschätzen, welche Komplexität und welches Zeitverhalten eine von ihm erstellte Zustandsmaschine später haben wird.

Ein Vorteil von *SystemC* besteht darin, daß für die Simulation der Hardwarebeschreibungen ein konventioneller C++-Kompiler ausreicht, während *CHDL* für Beschreibungen auf dieser Ebene bereits einen speziellen Hochsprachenkompiler benötigt.

Zur Synthese ist jedoch auch bei *SystemC* ein spezieller Kompiler erforderlich.

Durch die Notwendigkeit der `wait`-Funktion kann *SystemC* nur eingeschränkt als Hochsprache bezeichnet werden. Es existiert kein Mechanismus zur automatischen Zuordnung auf die einzelnen Taktzyklen oder zur Optimierung mittels Pipelining.

Die Möglichkeit, auf der Ebene vpn *SystemC* Module mit eigenem Ablaufpfad zu beschreiben, steht bei *CHDL* prinzipiell schon mit der C++-basierten Zustandsmaschinenbeschreibung zur Verfügung. Diese besitzt lediglich die Einschränkung, daß einige Anweisungen der strukturierten Programmierung wie etwa `while` oder `for` nicht direkt verwendet werden können.

15.2 *Handel-C*

Die Hochsprachenbeschreibung von *Handel-C* benötigt im Gegensatz zu *SystemC* keine Hilfskonstrukte wie die `wait`-Funktion. Die Zuordnung auf die einzelnen Taktzyklen wird automatisch vorgenommen. Dabei gilt die Grundregel, daß jede Anweisung einen Takt benötigt. Mithilfe der Anweisung `par` können mehrere Anweisungen oder auch Folgen von Anweisungen parallelisiert werden.

Im Gegensatz zu *CHDL* ist jedoch keine automatische Optimierung vorgesehen, die unabhängige Anweisungen in einem Takt zusammenfassen kann.

Im Beispiel

```
s = 0; i = 0;
while (i < 10)
 {
   s = s + i; i = i + 1;
 }
```

würden die Anweisungen `s = s + i` und `i = i + 1` in zwei getrennten Taktzyklen bearbeitet. Sie sind aber bei der hier vorliegenden Reihenfolge unabhängig voneinander und könnten ohne weiteres im gleichen Takt ausgeführt werden.

Bei gleicher Formulierung wird daher eine Implementierung mit *Handel-C* in der Regel mehr Taktzyklen benötigen als mit *CHDL*. Auch die automatische Anwendung von Pipeline-Verfahren ist bei *Handel-C* nicht vorgesehen. Durch manuelles Einfügen von Anweisungen zur Parallelisierung kann bei *Handel-C*-Beschreibungen die Anzahl notwendiger Takte reduziert werden, auch Pipeline-Konstruktionen sind möglich. Die resultierende Hardwarebeschreibung wird jedoch umfangreicher sein als bei *CHDL*:

```
s = 0; i = 0;
while (i < 10)
 {
   par { s = s + i;
         i = i + 1} ;
 }
```

Bei der Anwendung der `par`-Anweisung trägt der Entwickler selbst die Verantwortung für die Einhaltung der Datenabhängigkeiten.

Handel-C verwendet *Channels* zur Kommunikation zwischen parallelen Prozessen. Diese sind jedoch nicht auf die Einsparung von Taktzyklen optimiert. Der Zugriff auf einen Channel benötigt immer einen Taktzyklus, eine Parallelisierung mit anderen Anweisungen ist nicht möglich. Werden über Channels Daten übertragen, kann der empfangende Prozeß nur in jedem zweiten Takt Daten annehmen, da er im Takt nach der Annahme zunächst das gerade empfangende Datum verarbeiten muß. Die effiziente Konstruktion pipelineähnlicher Verfahren ist mit Channels somit nicht möglich.

Will der Entwickler nun, um effiziente Teilimplementierungen zu erreichen, einzelne Module seiner Gesamtschaltung auf einer niedrigeren Beschreibungsebene implementieren, muß er auf *VHDL*-Code zurückgreifen. Dazu muß er ein zweites Entwicklungssystem nutzen, was insbesondere die Simulation der Gesamtschaltung erschwert.

CHDL bietet für diese Fälle eine deutlich bessere Integration. Strukturelle und hochsprachenorientierte Hardwarebeschreibung kann nahezu beliebig gemischt werden. Die einzige Einschränkung besteht darin, daß das Mischen nicht innerhalb eines Moduls erfolgen kann. Jedes Modul ist entweder komplett strukturell oder aber komplett hochsprachenorientiert beschrieben.

Kapitel **16**
Zusammenfassung

Die größten Schwierigkeiten bei der Umsetzung hochsprachenorientierter Beschreibungen bestehen in der effizienten Synthese der Hardwarebeschreibung sowie in der Integration unterschiedlicher Abstraktionsebenen.

Das *CHDL*-System versucht, diese Probleme zu lösen, indem die Hochsprachenbeschreibung gleichberechtigt neben die strukturelle Beschreibung gestellt wird. Zeitkritische Teile des Designs werden strukturell implementiert, weniger zeitkritische, aber komplexe durch eine Hochsprachenbeschreibung.

Eine solche Vorgehensweise vermeidet von Anfang an die Problematik, daß der Entwickler eine strukturelle Vorstellung der zu implementierenden Schaltung in eine konventionell orientierte Hochsprache umsetzen muß. Denn dies verursacht erhebliche Probleme bei der späteren automatischen Optimierung der Beschreibung, in der die Informationen über die ursprüngliche strukturelle Absicht nicht mehr enthalten ist.

Die Übersetzung der Hochsprachenbeschreibung erfolgt durch einen in C++ implementierten C-Parser. Dieser generiert aus der analysierten Beschreibung ein Zwischenformat in einer Bytecodedarstellung. Er enthält die durchzuführenden Berechnungen, Kontrollanweisungen sowie Informationen für späteres Source-Level-Debugging.

Der Bytecode wird von der *Hardware Virtual Machine* (HVM) in eine Hardwarestruktur umgesetzt.

Diese ist damit für den größten Teil der vorzunehmenden Optimierungen zuständig. Folgende Optimierungen können hier vorgenommen werden:

- Automatische Ermittlung der Anweisungen, die parallelisiert werden können. Dadurch bestimmt sich die Aufteilung der Anweisungen auf die einzelnen Zustände der zu bildenden Zustandsmaschine.

  Bei diesem Vorgang kann die HVM auf die Methode der Zustandsmaschinenbeschreibung zugreifen, die das *CHDL*-Grundsystem zur Verfügung stellt.

- Automatische Konstruktion der Pipeline-Kontroller nach dem erläuterten Verfahren.

  Auch hier kann auf bereits existierende Funktionen des *CHDL*-Grundsystems zurückgegriffen werden, so etwa das modifizierte *One-Hot*-Verfahren, das Pipeline-Kontroller explizit unterstützt.

Die Effizienz der Umsetzung einer Hochsprache in Hardware ist grundsätzlich durch die Struktur der verwendeten Sprache eingeschränkt. Um mit einer nahe an C/C++ liegenden Hochsprache dennoch akzeptable Optimierungsergebnisse zu erzielen, werden die Verfahren der Parallelisierung und des Pipelining unterstützt.

Parallelisierbare Anweisungen können automatisch ermittelt oder vom Entwickler explizit vorgegeben werden. Innerhalb von Schleifen kann Pipelining eingesetzt werden. Dadurch überlagern sich die einzelnen Teilanweisungen und es wird eine parallele Ausführung voneinander abhängiger Operationen ermöglicht.

Die Hochsprachenbeschreibung ist direkt in das *CHDL*-System integriert. Der Entwickler kann für jedes einzelne Modul entscheiden, ob er eine Implementierung mittels struktureller Beschreibung, Zustandsmaschinendarstellung oder Hochsprachenbeschreibung vornimmt.

Der gesamte Vorgang kann mithilfe der *CHDL*-Bibliothek in einem einzige Prozeß im gleichen Adressraum vorgenommen werden. Dadurch ergibt sich eine wesentlich engere Kopplung der beteiligten Komponenten als bei herkömmlichen Systemen erreichbar ist.

Die vorgestellte Hochsprache enthält alle relevanten Kontrollanweisungen und ermöglicht die Beschreibung sowohl taktgenauer als auch vom Takt losgelöster Algorithmen.

Teil V

Gesamtbewertung und Ausblick

Kapitel 17
Erreichte Ziele

17.1 Gesamtkonzept

Mit *CHDL* konnte ein FPGA-Entwicklungssystem implementiert werden, das vollständig auf der universellen Programmiersprache C++ basiert. C++ wird für die Hardwarebeschreibung, für die Simulationsmodelle der externen Bausteine sowie für den Softwarebereich von FPGA-Koprozessoren eingesetzt.

CHDL ist in Form einer C++-Klassenbibliothek realisiert, die auf einfache Weise eingebunden sowie mit beliebigen weiteren Bibliotheken kombiniert werden kann. Lediglich die Hochsprachenbeschreibung wird mit einem speziell dazu entwickelten Kompiler übersetzt. Auch dieser ist Bestandteil der Klassenbibliothek und damit vollständig in das *CHDL*-Konzept integriert.

Die vordefinierten Klassen des *CHDL*-Systems werden über die bei C++ üblichen Methoden mittels Include- und Bibliotheksdateien eingebunden. Das Kompilieren erfolgt mit einem handelsüblichen C++-Kompiler.

Durch Ausführen der erstellten Hardwarebeschreibung wird eine Simulation bzw. die Synthese durchgeführt. Die Synthese liefert direkt eine Netzliste.

Ähnliche Konzepte zur Hardwarebeschreibung mittels C/C++ wurden bereits von anderen Systemen realisiert. Diese weisen jedoch wesentliche Einschränkungen in ihrer Funktionalität auf und unterstützen FPGA-Koprozessoren nicht in optimaler Weise. So bieten sie etwa keine ausreichende Unterstützung für die Beherrschung umfangreicher und komplexer Designs und ermöglichen keine vollständige Simulation von FPGA-Koprozessoren.

CHDL wurde nicht auf einem bestehenden System aufgebaut, sondern stellt eine vollständige Neuentwicklung dar. So konnten solche Einschränkungen weitgehend vermieden werden.

Auch handelt es sich bei *CHDL* nicht primär um eine hochsprachenorientierte Umsetzung und Optimierung von C/C++ in eine Hardwarestruktur, sondern vielmehr um ein System, das mehrere aufeinander aufbauende Abstraktionsebenen zur Verfügung stellt.

Herausragendes Merkmal ist dabei die vollständige Integration all dieser Abstraktionsebenen in das Gesamtkonzept von *CHDL*. Es bestehen nicht wie bei anderen Systemen getrennte Anwendungen für strukturelle Hardwarebeschreibung, Definition von Zustandsmaschinen und Hochsprachenbeschreibung. Alle Ebenen sind in der *CHDL*-Klassenbibliothek enthalten und können zusammen mit der Simulation aus einer einzigen Entwicklungsumgebung heraus genutzt werden.

Die daraus resultierende enge Kopplung aller Komponenten erlaubt weitreichende und effiziente Simulationsverfahren, die deutlich über die Fähigkeiten anderer Systeme hinausgehen.

Die automatisierbaren Verfahren, die *CHDL* zum Hardware-Debugging mittels Readback und zur partiellen Rekonfiguration bereitstellt, konnten bisher von keinem anderen Entwicklungssystem in diesem Umfang realisiert werden.

17.2 Hardwarebeschreibung

17.2.1 Allgemeines

Die Sprache C++ bewährt sich seit Jahren bei der Erstellung umfangreicher und komplexer Softwareanwendungen. Es handelt sich um eine vollständige und mächtige Programmiersprache. Durch Anwendung objektorientierter Techniken ist es möglich, Implementierungen auf hohem Abstraktionsgrad zu erstellen und den Entwickler an vielen Stellen von Details zu entlasten.

Im Rahmen dieser Arbeit ist es gelungen, die genannten Fähigkeiten von C++ zur Beherrschung von Umfang und Komplexität auch auf die Hardwarebeschreibung zu übertragen.

Mit derselben Sprache wird in der Regel auch der Softwarebereich von FPGA-Koprozessoren implementiert. So muß der Entwickler bei der Verwendung von *CHDL* nur noch eine einzige Sprache beherrschen: C++. Die enge Kopplung zwischen FPGA und Mikroprozessor ist dadurch deutlich besser beherrschbar als beim Einsatz verschiedener Sprachen.

Zur Hardwarebeschreibung können weiterhin alle dem Softwareentwickler bereits vertrauten Werkzeuge und Programmiertechniken eingesetzt werden. Das Fachwissen und die Erfahrung einer Sprache reichen nun aus, um den gesamten Designentwicklungszyklus für FPGA-Koprozessoren vollständig zu beherrschen.

Mittels C++-Klassen und -Objekten kann ein hierarchischer Designentwurf erfolgen. Analog zur konventionellen Softwareentwicklung sind alle gängigen Verfahren zum Aufbau wiederverwendbarer Bibliotheken einsetzbar.

Die mit *CHDL* erreichbare Form der Hardwarebeschreibung ist kompakt und konsequent objektorientiert gehalten. Dadurch wird die Übersichtlichkeit der Beschreibung deutlich erhöht.

Die Reduktion der Komplexität und des Umfangs führen dabei jedoch nicht zu einem Verlust an Effizienz. Im Gegenteil, durch spezielle Anweisungen hat der Entwickler vollständigen Einfluß auf die einzelnen Resourcen des FPGAs.

Zudem zwingt die Hardwarebeschreibung von *CHDL* den Entwickler auch nicht, auf einer bestimmten Abstraktionsebene zu arbeiten. Er hat vielmehr die Freiheit, jedes Modul seines Gesamtdesigns auf der Ebene zu implementieren, die ihm am geeignetsten dafür erscheint.

17.2.2 Strukturelle Ebene

Die strukturelle Ebene ermöglicht eine Hardwarebeschreibung, die der textuellen Beschreibung eines Schaltplans entspricht. Mit ihr lassen sich die effizientesten Implementierungen erzielen, sie erfordert jedoch auch den größten Aufwand. Die zu realisierende Schaltung muß im Detail spezifiziert werden. C++ besitzt zahlreiche programmtechnische Möglichkeiten, diesen Umfang und diese Komplexität etwa durch objektorientierte Methoden oder weitgehende Parametrisierungen zu beherrschen.

CHDL stellt mittels vordefinierter Klassen grundlegende Elemente bereit, beispielsweise Flip-Flops, Gatter, Multiplexer und Zähler. Alle in FPGAs enthaltenen Resourcen sind als Klassen verfügbar. Die Elemente können durch Instanziierung der entsprechenden Klassen erzeugt und durch Operatoren miteinander verschaltet werden. Der Entwickler kann die Menge der Elemente beliebig erweitern, indem er neue Klassen definiert, die in ihrem Konstruktor die gewünschte Funktionalität implementieren. Auf diese Weise lassen sich beliebig komplexe strukturelle Hardwarebeschreibungen spezifizieren.

Die vordefinierten Elemente sind überwiegend architekturunabhängig. Dies erleichtert die spätere Portierung auf andere, auch zukünftige, FPGAs, selbst wenn eine solche bei der Designerstellung noch nicht geplant war.

Zahlreiche Funktionen erlauben es, die einzelnen FPGA-Resourcen zu kontrollieren. So ist es etwa möglich, gezielt Elemente vorzuplazieren oder Parameter zu spezifizieren und somit direkten Einfluß auf die zu implementierenden Hardwarestrukturen auszuüben. Auf diese Weise lassen sich resourcensparende Designs mit hohen Taktfrequenzen erstellen.

Bereits auf dieser Ebene ist *CHDL* anderen Systemen deutlich überlegen. *VHDL* beispielsweise bietet durch die eingeschränkte sprachliche Mächtigkeit eine schwächere Unterstützung bei der Erstellung umfangreicher und komplexer Schaltungen als C++. Auch die Kontrolle über die einzelnen FPGA-Resourcen, die zur Erstellung hocheffizienter Strukturen unbedingt erforderlich ist, wird von *VHDL* nur eingeschränkt ermöglicht.

Systeme wie *PamDC* oder *JHDL* verfolgen einen ähnlichen Ansatz wie *CHDL*, realisieren diesen jedoch nicht konsequent genug. Die Hardwarebeschreibung wirkt umständlich und führt bei der Erstellung umfangreicherer Schaltungen schnell zur Unübersichtlichkeit. Es

fehlen Verfahren, die kompakte, aber dennoch leistungsfähige Beschreibungen erlauben.

Strukturelle Hardwarebeschreibungen mit *SystemC* wirken ebenfalls umständlich und neigen zur Unübersichtlichkeit. Die Form der Beschreibung ist eng an *VHDL* angelehnt und zudem werden Konstrukte verwendet, die eine Synthese mit einem handelsüblichen C++-Kompiler unmöglich machen. Daher benötigt dieses System im Gegensatz zu allen anderen C/C++-basierten einen speziellen Kompiler für die Synthese.

Handel-C wiederum verfügt über keine direkte Unterstützung der strukturellen Ebene. Zur Einbindung struktureller Teilimplementierungen muß auf *VHDL*-Code zurückgegriffen werden.

17.2.3 Zustandsmaschinenbeschreibung

Die Ebene der Zustandsmaschinenbeschreibung erlaubt es, Algorithmen in Form von Flußdiagrammen direkt in eine Hardwarebeschreibung umzusetzen. Innerhalb von Zuständen können Anweisungen für arithmetische Operationen spezifiziert werden. Bedingte oder unbedingte Zustandsübergänge legen dann die Ausführungsreihenfolge der Zustände fest.

Diese Form der Hardwarebeschreibung ist am besten geeignet für die Teile eines Gesamtdesigns, die in ihrer Grundstruktur einen sequentiellen Ablauf aufweisen und bei denen das Timing und die Menge der benötigten Resourcen weniger kritisch sind.

Die bei *CHDL* realisierte Form der Zustandsmaschinenbeschreibung geht dabei weit über die Möglichkeiten anderer Systeme hinaus. Das zugrundeliegende Ausführungsmodell des modifizierten *One-Hot*-Encodings läßt mehrere aktive Ausführungstokens und damit mehrere aktive Threads in einer Zustandsmaschinenbeschreibung zu.

Die Beschreibung kann sogar auf eine dynamische Weise erfolgen, bei der sich die konkrete Implementierung erst zur Laufzeit bestimmt. Dies ermöglicht flexible Verfahren zur automatischen Erzeugung von Zustandsmaschinen z.B. durch hochsprachenorientierte Werkzeuge.

Zwar verfügt *VHDL* ebenfalls über eine Methode zur Spezifizierung von Zustandsmaschinen, diese unterliegt jedoch den gleichen sprachlichen Einschränkungen, die von der strukturellen Ebene bereits bekannt sind.

Die Systeme *PamDC* und *JHDL* bieten keine spezielle Unterstützung für die Erstellung von Zustandsmaschinen. Sie stellen lediglich eine einfache strukturelle Hardwarebeschreibung zur Verfügung, die für komplexe Schaltungen nicht ausreichend ist.

Bei *SystemC* erfolgt die Zustandsmaschinenbeschreibung in Form von Prozessen, die jedoch stark an *VHDL* angelehnt sind und keine direkte und kompakte Umsetzung von Flußdiagrammen ermöglichen. Das Verfahren, mit dem diese Beschreibung konkret umgesetzt wird, bleibt dem Entwickler verborgen. Er kann dadurch nur schwer abschätzen, welche Komplexität und welches Zeitverhalten ein von ihm erstellter Prozeß später haben wird.

Ein ähnliches Problem ist bei *Handel-C* gegeben. Hier lassen sich auch einfache Zustandsmaschinen nur über die Hochsprachenbeschreibung realisieren.

17.2.4 Hochsprachenebene

Die Ebene der Hochsprachen, die sich bei *CHDL* noch im Prototypstadium befindet, eignet sich für umfangreiche sequentielle Algorithmen, die weniger resourcen- und zeitkritisch sind.

CHDL unterstützt eine auf der Sprache C basierende Hochsprachenbeschreibung. Die Parallelisierung von Anweisungen kann manuell vorgegeben oder automatisch durchgeführt werden. Die Sprache umfaßt die üblichen Kontrollanweisungen wie etwa `if`, `while`, `do` oder `for`.

Die Umsetzung in eine Hardwarestruktur erfolgt intern mithilfe der darunterliegenden Ebene der Zustandsmaschinenbeschreibung. Es handelt sich dabei um eine leicht nachvollziehbares Verfahren, mit der der Anwender stets in der Lage ist, das Ergebnis seiner Beschreibung abzuschätzen.

Die vorhandene Implementierung des Hochsprachenkompilers enthält das automatische Pipelining noch nicht. Auch die Parallelisierung ist noch nicht vollständig unterstützt. Den-

noch ist bereits jetzt erkennbar, daß der zugrundeliegende Ansatz ein deutlich höheres Potential besitzt als andere hochsprachenorientierte Systeme.

So ist bei *Handel-C* ein automatisches Pipelining prinzipiell nicht vorgesehen. Auch muß jede Form von Parallelität explizit angegeben werden, es existiert kein Mechanismus, der aufgrund einer Analyse der Datenabhängigkeiten eine Parallelisierung selbständig vornimmt. Durch das implementierte Umsetzungsverfahren werden daher mehr Takte benötigt als eigentlich erforderlich sind.

17.3 Simulation

Die bei *CHDL* eingesetzte gemeinsame Sprache C++ erleichtert die Simulation von Gesamtsystemen. Hardwarebeschreibung, externe Simulationsmodelle sowie der Softwarebereich von FPGA-Koprozessoren können mit derselben Sprache implementiert werden.

Das *CHDL*-System verfügt über einen universellen, funktionalen Simulationsmechanismus, der ohne Einschränkung mehrere Taktsignale, asynchrone Anordnungen sowie kombinatorische Schleifen bearbeiten kann.

Die Simulation der einzelnen Elemente sowie externer Komponenten erfolgt über C++-Funktionen, mit denen die Reaktion der Elemente auf Veränderungen ihrer Eingangssignale sowie auf vorgegebene zeitliche Ereignisse festgelegt wird.

Diese Funktionen haben gegenüber *VHDL*-Testbenches den Vorteil, daß sie sehr effizient implementiert werden können. Damit wird auch die Simulation großer externer Speicher, wie etwa SDRAMs, mit nur geringem Bedarf an Rechenzeit möglich.

Die enge Kopplung von Hardware- und Softwarebereich bei FPGA-Koprozessoren wird durch ein exception-basiertes Verfahren zur Simulation von Pseudoregistern unterstützt. Dies erlaubt die Verwendung des nahezu unveränderten Softwarebereiches sowohl zur Simulation als auch zum realen Betrieb.

Hardwarebeschreibung, externe Bauteile und Softwarebereich können so innerhalb desselben Prozesses ausgeführt werden. Dies erlaubt hohe Simulationsgeschwindigkeiten ohne effizienzmindernde Verfahren zur Interprozeßkommunikation. Auf diese Weise können auch komplexe Interaktionen zwischen dem FPGA und seiner Umgebung in die Simulation integriert werden.

Die meisten bisherigen Entwicklungssysteme ermöglichen nur die Erzeugung eines Signalverlaufes während der Simulation, der schnell unübersichtlich werden kann. *CHDL* dagegen erlaubt zusätzlich die vollständige Simulation kompletter Koprozessoranwendungen mit realen Arbeitsdaten. So kann etwa bei einem Bildverarbeitungsprojekt auch das entstandene Ergebnisbild überprüft werden.

Da *CHDL* im Gegensatz zu separaten *VHDL*-Simulatoren oder *SystemC* für Simulation und Synthese dieselbe strukturelle Datenbasis verwendet, sind Inkonsistenzen weitgehend ausgeschlossen.

17.4 Synthese und Hardware-Debugging

Die Synthese des *CHDL*-Systems erzeugt Netzlisten, die von der Place&Route-Software direkt weiterverarbeitet werden können.

Alle Bauteilnamen der Hardwarebeschreibung werden unverändert in die Netzliste übernommen. Damit ist es möglich, alle Symbole der Netzliste eindeutig zu identifizieren, was eine weitgehende Automatisierung von Verfahren zum Readback und zur partiellen Rekonfiguration erlaubt.

Auf diese Weise kann *CHDL* Methoden zum Hardware-Debugging bereitstellen, die deutlich über die Fähigkeiten anderer Systeme hinausgehen. Neben dem einfachen Auslesen des aktuellen Designzustandes zu jedem beliebigen Zeitpunkt kann dieser Zustand auch jederzeit modifiziert werden. So wird es beispielsweise möglich, Startpunkte für die Simulation zu definieren oder integrierte Logikanalyzer automatisiert zu betreiben.

Kapitel **18**

Bisherige Einsatzbereiche und Entwicklungsstand des *CHDL*-Systems

18.1 Bisherige Einsatzbereiche von *CHDL*

Die Ebene der strukturellen Hardwarebeschreibung ist vollständig implementiert und wurde bereits in mehreren universitären Projekten zu Entwicklung realer Anwendungen eingesetzt.

Das erste umfangreiche Projekt, das komplett mithilfe des *CHDL*-Systems entwickelt wurde, war ein Videokompressionssystem [53] entsprechend dem ITU H.263-Standard. Das Hauptproblem bestand darin, die komplexe Kompressionslogik im damals größten zur Verfügung stehenden FPGA (XC4085XLA) zu realisieren. *CHDL* ermöglichte eine effiziente Implementierung durch volle Kontrolle über die FPGA-Resourcen auf niedriger Ebene. Die komplette Schaltung benötigte 89 Prozent der vorhandenen Resourcen und erreichte eine maximale Taktfrequenz von 35 MHz. Der begrenzende Faktor war hierbei allerdings die Peripherie des FPGAs.

Im Projekt "Blob-Analyse" [99] wurden die Möglichkeiten, die *CHDL* zur Parametrisierung bietet, intensiv genutzt. Es entstand eine Bibliothek von Modulen, die in ihrer konkreten Implementierung speziell an die jeweiligen Anforderungen angepasst werden konnten und auf diese Weise resourcensparende Implementierungen erlaubten.

In [82] wurde demonstriert, wie mit *CHDL* der Abstraktionsgrad struktureller Hardwarebeschreibungen erhöht werden kann. Die Aufgabe bestand darin, ein allgemeines Interface zur Anbindung von Speicherelementen zu implementieren. Bei den Speicherelementen selbst konnte es sich um internes CLB-RAM, Block-RAM oder auch um externe Bausteine wie statische oder dynamische Speicher handeln. Der Anwender dieses allgemeinen RAM-Interfaces spezifiziert nur noch seine Anforderungen, wie etwa die Größe des benötigten Speicherbereiches und die erforderlichen Zugriffszeiten. Das Interface teilt dann eine passende Speicherresource zu oder informiert den Anwender, daß eine solche nicht zur Verfügung steht. Mithilfe der objektorientierten Konzepte von C++ kann diese Funktionalität weitgehend gekapselt und verborgen werden.

Die Firma *Silicon Software GmbH* [85] führt seit 1999 eine permanente Evaluierung des *CHDL*-Systems durch und setzt dieses inzwischen als hauptsächliches Entwicklungswerkzeug bei der Programmierung des FPGA-Koprozessors *microEnable* ein.

18.2 Zustandsmaschinenbeschreibung

Die Implementierung der Zustandsmaschinen ist im Rahmen dieser Arbeit ebenfalls vollständig abgeschlossen worden.

Eine erste komplexe Anwendung dieser Beschreibungsebene wurde mit der Implementierung eines SDRAM-Kontrollers vorgenommen.

Er wird zur Ansteuerung handelsüblicher SDRAM-Bausteine eingesetzt und enthält die notwendige Logik wie etwa Refresh-Zähler und die Bedingungen zum Seitenwechsel und zur Überwachung der maximalen Aktivierungszeit einer Seite. Weiterhin unterstützt er effiziente Burst-Zugriffe auch großer Datenmengen und zwei gleichberechtigte Schnittstellen für Lese- und Schreibzugriffe.

Die realisierte Zustandsmaschine umfaßt etwa 80 Zustände, einschließlich notwendiger Wartetakte, die zur Einhaltung der SDRAM-Spezifikationen erforderlich sind.

Die FPGA-Implementierung dieses Kontrollers mit einer minimalen Umgebungslogik, die zum Betrieb in einem PCI-System erforderlich ist, erreicht laut *XILINX*-Zeitanalyse eine Taktfrequenz von 133 MHz. Dies entspricht der maximalen Frequenz der verfügbaren (nicht DDR-) SDRAM-Bausteine.

Ein Kontroller-Modul für SDRAMs wird auch von *XILINX* als IP-Core angeboten. Solche Cores enthalten die entsprechende Funktionalität in hochoptimierter Form und werden als EDIF-Netzliste geliefert. Das Modul *OPB Synchronous SDRAM Controller* [113] besitzt eine ähnliche Funktionalität wie der implementierte *CHDL*-Kontroller. Die maximale Betriebsfrequenz bei einer Datenbreite von 32 Bit ist mit 137 MHz angegeben.

Dies zeigt, daß die Zustandsmaschinenumsetzung des *CHDL*-Systems effizient genug arbeitet, so daß der erzeugte SDRAM-Kontroller in seinem Zeitverhalten mit dem hochoptimierten IP-Core vergleichbar ist.

18.3 Hochsprachenorientierte Beschreibung

18.3.1 Hochsprachenparser

Das Parsermodul zur Analyse der Hochsprachenbeschreibung wurde prinzipiell mit allen Kontrollanweisungen und Operatoren, die in der Sprache C zulässig sind, implementiert.

Aufgrund des großen Umfanges einer solchen Parser-Implementierung, die die Behandlung einer Vielzahl von Detailfällen erfordert, konnte die konkrete Umsetzung einiger Sprachkonstrukte bisher nicht vollständig implementiert werden. So sind etwa die Definition von Datenstrukturen mittels `struct` sowie der Aufruf von Unterprogrammen noch nicht integriert.

Die aktuelle Implementierung erlaubt aber bereits die Übersetzung von Hochsprachenprozeduren mit ihren Parameterlisten. Als zulässige Parameter werden bisher nur die Datentypen `istream` und `ostream` unterstützt.

18.3.2 *Hardware Virtual Machine* (HVM)

Die *Hardware Virtual Machine* erfordert insbesondere in der Implementierung der einzelnen Operatoren eine Vielzahl von Fallunterscheidungen. Diese Notwendigkeit ergibt sich aus der jeweils speziellen Behandlung verschiedener Datentypen. Es wurden bisher nur die wichtigsten Kombinationen realisiert, um das prinzipielle Funktionieren des Konzepts zeigen zu können.

Die Analyse der Datenabhängigkeiten zur automatischen Parallelisierung der Anweisungen erfolgt zur Zeit nach einem vereinfachten Schema: Anweisungen werden solange in denselben Zustand aufgenommen, bis eine Zuweisung an eine zuvor verwendete Variable erkannt wird. Dies schließt den aktuellen Zustand ab, die aktuelle Anweisung wird in den neuen Zustand übernommen. Dies stellt eine sehr einfache Prüfung der Datenabhängigkeiten dar, die in Teil IV diskutierten weitergehende Algorithmen wurden bisher noch nicht implementiert.

Die automatische Generierung der Pipeline-Kontroller wäre ebenfalls Bestandteil der HVM, ist in der vorhandenen Implementierung bisher ebenfalls noch nicht enthalten.

Die HVM ermöglicht aber bereits die Umsetzung von Hochsprachenprozeduren über den beschriebenen Zwischencode in eine Hardwarestruktur, die Anbindung an die *CHDL*-Umgebung ist insoweit auch funktionsfähig.

Kapitel 19
Ausblick

19.1 Akzeptanzprobleme kommerzieller Anwender

Trotz der unbestreitbaren Vorteile von *CHDL* konnte sich dieses Entwicklungssystem bislang noch nicht im kommerziellen Bereich durchsetzen. Dies kann durch die nachfolgend beschriebenen Faktoren erklärt werden.

19.1.1 Umfangreiche existierende *VHDL*-Bibliotheken

In Betrieben, die schon lange Zeit im FPGA-Bereich tätig sind, bestehen umfangreiche und weitgehend verifizierte *VHDL*-basierte Bibliotheken. Die Weiterverwendbarkeit dieses Codes ist für diese Anwender ein entscheidendes Kriterium bei einem Wechsel des Entwicklungssystems.

19.1.2 Kommerzielle Anwender sind skeptisch gegenüber dem, was nicht verbreiteter Standard ist

Es gibt nur wenige Referenzanwendungen von *CHDL*, die zudem überwiegend aus dem universitären Bereich stammen.

Es erfordert viel Zeit, sich in ein neues System einzuarbeiten. Einfache Designimplementierungen reichen in der Regel nicht aus, um die Eignung eines Entwicklungssystems für komplexe Anwendungen zu überprüfen. Das Risiko, nach aufwendiger Einarbeitung und Implementierung eines realen Projektes auf unerwartete Schwierigkeiten zu stoßen, wird bei proprietären Systemen höher eingestuft als bei Standardanwendungen.

19.1.3 Weiterentwicklung

Viele Projekte kommerzieller Anwender sind langfristig orientiert. Auch inbezug auf Know-How und Einarbeitungszeiten der Mitarbeiter herrscht ein langfristiges und eher konservatives Denken vor. Gerade unter diesen Gesichtspunkten besitzt die Frage der Zukunftssicherheit eine große Bedeutung. Kein kommerzieller Anwender wird bereit sein, die Zeit seiner Mitarbeiter in ein System zu investieren, dessen zukünftige Entwicklung nicht abschätzbar ist. Universitäten sind aufgrund befristeter Forschungsprojekte und oft wechselnder Mitarbeiter selten in der Lage, eine langfristige Weiterentwicklung zu garantieren.

In diesem Zusammenhang ist insbesondere die Unterstützung zukünftiger FPGA-Architekturen wichtig. Zur Zeit unterstützt *CHDL* die *XILINX*-Architekturen bis *Virtex-II Pro*. Es ist zur Zeit jedoch nicht absehbar, in welcher Weise sich diese Architekturen weiterentwickeln und mit welchem Aufwand die notwendige Unterstützung realisiert werden kann.

19.1.4 Schnelle Fehlerbeseitigung

Wird ein Entwicklungssystem in kommerziellen Projekten eingesetzt, können Softwarefehler, die nicht rechtzeitig erkannt und behoben werden, hohe Kosten verursachen. Bei der Entscheidung über ein neues System wird der Anwender Wert darauf legen, daß erkannte Fehler schnell beseitigt werden, um solche Kosten zu vermeiden.

19.2 Mögliche Weiterentwicklungen

Das beschriebene Akzeptanzproblem des *CHDL*-System kann durch geeignete Weiterentwicklung gelöst werden. Abschließend werden daher einige mögliche Maßnahmen erläutert.

19.2.1 Verbesserte Schnittstellen zu klassischen Systemen

CHDL verfügt über die Möglichkeit, neu erstellte Module in Form von *VHDL*-Code zu exportieren. Dadurch ist bereits eine Integration in eine *VHDL*-Umgebung realisierbar.

Umgekehrt können Module, die mit beliebigen anderen Systemen erstellt wurden, in ein *CHDL*-Design integriert werden. Dies ist bisher jedoch nur auf der Ebene von Netzlisten möglich.

Solche externen Module können nicht ohne weiteres innerhalb der *CHDL*-Umgebung simuliert werden. Der Entwickler muß zunächst eine separate Simulationsfunktion implementieren, was eine Quelle für mögliche Inkonsistenzen darstellt.

Lösbar wäre dieses Problem mit einer Importfunktion für das EDIF-Netzlistenformat. Unter Verwendung eines EDIF-Parsers könnten aus den einzelnen EDIF-Primitiven *CHDL*-Konstrukte erzeugt werden, die dann ohne separate Simulationsfunktion sofort simulierbar wären.

Auf diese Weise wäre bei einer Einführung von *CHDL* die Weiterverwendung bestehender *VHDL*-Bibliotheken möglich. Durch den direkten Import von EDIF-Code sind alle Module sowohl simulierbar als auch synthetisierbar. *CHDL* garantiert durch die Verwendung derselben Datenbasis auch die Übereinstimmung zwischen Simulation und Synthese.

19.2.2 Kombination der *CHDL*-Simulation mit anderen Systemen

Mit dem oben erwähnten EDIF-Parser wäre es weiterhin möglich, komplette FPGA-Designs aus anderen Systemen in die *CHDL*-Umgebung zu importieren.

Dadurch könnten die Vorteile der *CHDL*-Simulation insbesondere für FPGA-Koprozessoren auch dann genutzt werden, wenn die zu simulierenden Designs nicht mit *CHDL*, sondern der Hardwarebeschreibungssprache eines anderen Systems erstellt wurden.

Die vorhandene Hardwarebeschreibung wird zunächst vom entsprechenden Kompiler synthetisiert und eine EDIF-Netzliste erzeugt. Diese Netzliste wird über die EDIF-Importfunktion in das *CHDL*-System eingelesen. Die Simulation kann dann auf die gleiche Weise durchgeführt werden wie bei normalen *CHDL*-Designs.

Im Simulationsbereich für FPGA-Koprozessoren besitzen die herkömmlichen Entwicklungssysteme eindeutige Schwächen, so daß das Interesse hier höher sein dürfte als auf anderen Gebieten. Voraussetzung ist allerdings, daß der Anwender seinen bisherigen Designzyklus beibehalten kann und die *CHDL*-Simulation nur zusätzlich einsetzt.

19.2.3 Kombination des *CHDL*-Hardware-Debuggings mit anderen Systemen

CHDL stellt einige mächtige Verfahren für ein Hardware-Debugging bereit.

Die Anwendbarkeit in Verbindung mit anderen Systemen zur Hardwarebeschreibung ist jedoch durch das Problem der nicht identifizierbaren Netzlistensymbole eingeschränkt. So ist etwa eine automatische Unterstützung in *VHDL*-basierten Designs nur schwer zu realisieren.

Mit *CHDL* erstellte integrierte Logikanalyzer können trotz dieser Problematik eingesetzt werden, um die Fehlersuche zu beschleunigen. Die betreffenden Module besitzen durch ihre Implementierung in *CHDL* bereits eindeutige Netznamen. Sie können als Netzlistenmodule in beliebige andere Entwicklungssysteme integriert werden und bleiben dabei für den Readback-Mechanismus automatisch erkennbar.

19.2.4 Gestaltung offener Schnittstellen

Durch die Implementierung als C++-Klassenbibliothek bietet *CHDL* dem Anwender prinzipiell die Möglichkeit, eigene Funktionalität hinzuzufügen oder vorhandene zu ersetzen. Diese Eigenschaft ist bei vielen anderen Entwicklungssystemen nicht vorhanden.

Der Anwender ist bei *CHDL* in der Lage, eventuell notwendige kleinere Änderungen oder Korrekturen selbst dann einzufügen, wenn ihm der *CHDL*-Quellcode nicht vorliegt. Mit einigen Grundkenntnissen der EDIF-Struktur ist es ihm so etwa möglich, Resourcen einer erweiterten FPGA-Architektur selbst hinzuzufügen oder eine als fehlerhaft erkannte zu ersetzen.

Voraussetzung ist jedoch, daß die Schnittstellen zum *CHDL*-Kernsystem geeignet gestaltet sind. Die Kenntnis dieser Schnittstellen alleine muß ausreichen, um Änderungen vornehmen zu können. In der vorhandenen Implementierung ist diese Voraussetzung noch nicht erfüllt.

Bei der Weiterentwicklung von *CHDL* könnten die Schnittstellen entsprechend gestaltet werden, um durch die dann vorhandene Offenheit des Systems bei den Anwendern Vertrauen zu schaffen.

19.2.5 Optimierung des Gesamtentwicklungsprozesses

Die erforderlichen Entwicklungszeiten bei der Erstellung von FPGA-Designs sind nicht nur von der Mächtigkeit und den Abstraktionsebenen der Hardwarebeschreibungssprache abhängig, sondern auch von der Handhabbarkeit der gesamten Entwicklungssoftware. *CHDL* bietet aufgrund seiner Implementierung als C++-Klassenbibliothek deutlich mehr Möglichkeiten zur Automatisierung als klassische Systeme, die in der Regel als eigenständige Anwendungen realisiert sind. So lassen sich mit *CHDL* leicht Anordnungen erstellen, die die Synthese und den nachfolgenden Place&Route-Prozeß automatisieren. Auch der Aufbau umfangreicher und komplexer Testcases, die selbständig ablaufen, ist denkbar. Die notwendige Unterstützung, die *CHDL* für solche Anwendungen zur Verfügung stellen müsste, wäre mit geringem Aufwand implementierbar.

Eine so erreichbare Reduzierung der Gesamtentwicklungszeit könnte insbesondere für kleinere Unternehmen mit begrenzten Personalresourcen von Interesse sein. Der Einsatz von *CHDL* für den kompletten Designentwicklungszyklus könnte hier konkret mit dem Ziel erfolgen, die Produktivität einer kleinen Entwicklergruppe zu erhöhen.

19.2.6 Erweiterungen der *Hardware Virtual Machine* (HVM)

Das Prinzip der HVM stellt eine sehr flexible Schicht zwischen dem Hochsprachenparser und der strukturellen Hardwareebene dar. So könnte die HVM anstelle direkter Hardwarestrukturen auch Objektcode für konventionelle Mikroprozessoren generieren (Abb. 19.1). Es wäre weiterhin möglich, dies für jeden zu implementierenden Prozeß separat zu entscheiden. Damit könnten komplette Algorithmen in der Hochsprache spezifiziert werden, wobei später ein Teil davon direkt in Hardware und der restliche Teil als Objektcode für Mikroprozessoren implementiert wird.

Ein zukünftiges System könnte somit bei Verfügbarkeit geeigneter Algorithmen eine automatische Partition des Hochsprachencodes vornehmen.

Alternativ wäre bei der HVM auch analog zur *JAVA Virtual Machine* eine direkte Ausführung möglich. Damit liesse sich eine schnelle Simulation auf der Ebene der Hochsprache realisieren. Dazu müsste jedoch noch eine Schnittstelle zu niedrigeren Beschreibungsebenen in das Konzept der HVM integriert werden.

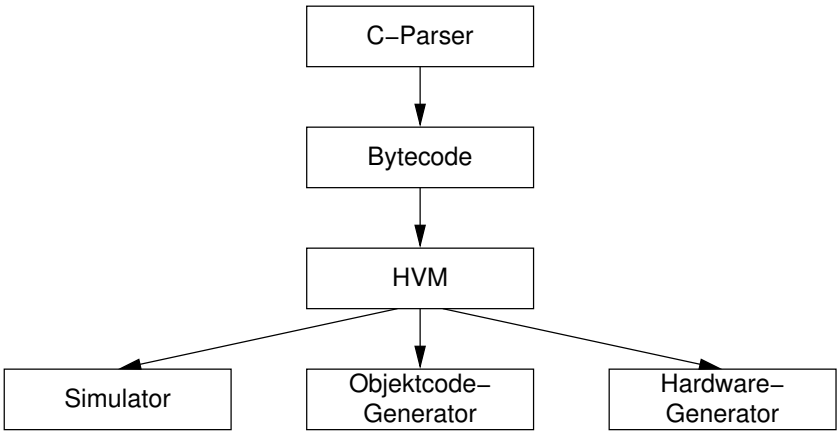


Abbildung 19.1: Erweiterter Einsatz der HVM

Literaturverzeichnis

[1] Accolade Design Automation.
`http://www.acc-eda.com`

[2] Aldec.
`http://www.aldec.com`

[3] Annapolis Microsystems Inc. *Overview of the WILDFORCE, WILDCHILD and WILDSTAR Reconfigurable Computing Engines*.
`http://www.annapmicro.com`

[4] P. Ashar, S. Devadas, A. R. Newton. *Sequential Logic Synthesis*. Kluwer, Boston, 1992.

[5] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua,S. Amarasinghe. *Parallelizing Applications into Silicon*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 70-80, Napa, USA, April 1999.

[6] S. R. Ball. *Embedded Microprocessor Systems*. Newnes, Boston, 1996.

[7] S. Baranov. *CAD System for ASM and FSM Synthesis*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 119-128, August 1998.

[8] J. Becker, A. Kirschbaum, F. -M. Renner, M. Glesner. *Perspectives of Reconfigurable Computing in Research, Industry and Education*. In *Field Programmable Logic and Applications (FPL)*, S. 39-48, Tallin, Estonia, September 1998.

[9] P. Bellows, B. Hutchings. *JHDL - An HDL for Reconfigurable Systems*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 175-184, Napa, USA, April 1998.

[10] P. Bertin, H. Touati. *PAM Programming Environments: Practice and Experience*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, Napa, USA, April 1994.

[11] M. Bolton. *Digital Systems Design with Programmable Logic*. Addison-Wesley, 1991.

[12] G. Brebner. *Field-Programmable Logic: Catalyst for New Computing Paradigms*. In *Field Programmable Logic and Applications (FPL)*, S. 49-58, Tallin, Estonia, September 1998.

[13] Brigham Young University. *JHDL Documentation*.
`http://jhdl.ee.byu.edu`

[14] O. Brosch, P. Dillinger, K. Kornmesser, A. Kugel, R. Männer, R. Rissmann, S. Rühl, H. Simmler, H. Singpiel, R. Lay, K.-H. Noffz. *Simulating FPGA-Coprocessors using the FPGA Development System CHDL*. In *Proceedings of PACT'98 Workshop on Reconfigurable Computing*, Paris, 1998.

[15] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic. *Field-Programmable Gate Arrays*. 2. Aufl., Kluwer, Boston, 1993.

[16] M. Budiu, S. C. Goldstein. *Fast Compilation for Pipelined Reconfigurable Fabrics*, In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, S. 195-205, 1999.

[17] J. M. P. Cardoso, H. C. Neto. *Macro-Based Hardware Compilation of Java Bytecodes into a Dynamic Reconfigurable Computing System*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 2-11, Napa, USA, April 1999.

[18] Celoxica. *DK1 Design Suite User Manual*. Version 2.1, 2001.

[19] Celoxica. *Floating Point and Fixed Point Libraries Manual*. Version 1.0, 2001.

[20] Celoxica. *Handel-C*.
`http://www.celoxica.com`

[21] Celoxica. *Handel-C Language Reference Manual*. Version 2.1, 2001.

[22] K. S. Chatha, R. Vemuri. *Hardware-Software Codesign for Dynamically Reconfigurable Architectures*. In *Field Programmable Logic and Applications (FPL)*, S. 175-184, Glasgow, GB, September 1999.

[23] M. Chu, N. Weaver, K. Sulimma, A. DeHon, J. Wawrzynek. *Object Oriented Circuit-Generators in Java*. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, S. 158-166, Napa, USA, April 1998.

[24] CoCentric SystemC Compiler.
`http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC_ds.html`

[25] D. C. Cronquist, P. Franklin, S. G. Berg, C. Ebeling. *Specifying and Compiling Applications for RaPiD*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 116-125, Napa, USA, April 1998.

[26] R. D. Dowsing, F. W. D. Woodhams. *Computers: from logic to architecture*. Van Nostrand Reinhold (International), London, 1990.

[27] A. A. Duncan, D. C. Hendry, P. Gray. *An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 106-115, Napa, USA, April 1998.

[28] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, S. G. Berg. *Mapping Applications to the RaPiD Configurable Architecture*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 106-115, Napa, USA, April 1997.

[29] M. Eisenring, J. Teich. *Interfacing Hardware and Software*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 520-524, August 1998.

[30] Exemplar.
`http://www.exemplar.com`

[31] J. Fischer, C. Müller, H. Kurz. *A Co-simulation Concept for an Efficient Analysis of Complex Logic Designs*. In *Field Programmable Logic and Applications (FPL)*, S. 495-499, Tallin, Estonia, September 1998.

[32] J. Frigo, M. Gokhale, D. Lavenier. *Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective*, In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, S. 134-140, Monterey, USA, 2001.

[33] FZI Darmstadt. *Spyder*-Koprozessor.
`http://www.fzi-darmstadt.de`

[34] J. G. Ganssle. *The Art of Designing Embedded Systems*. Newnes, Boston, 1999.

[35] M. B. Gokhale, J. Stone. *Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 63-69, Napa, USA, April 1999.

[36] M. B. Gokhale, E. Gomersall. *High Level Compilation for Fine Grained FPGAs*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 165-173, Napa, USA, April 1997.

[37] M. B. Gokhale, J. M. Stone. *NAPA C: Compiling for a Hybrid RISC/FPGA Architecture*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 126-135, Napa, USA, April 1998.

[38] M. B. Gokhale, J. M. Stone. *Stream-Oriented FPGA Computing in the Streams-C High Level Language*. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, Napa, April 2000.

[39] J. P. Hammes, R. Rinker, W. Böhm, W. A. Najjar, B. Draper. *A High Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems*. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, S. 135-141, Las Vegas, USA, Juni 2000.

[40] Handel-C homepage.
`http://oldwww.comlab.ox.ac.uk/oucl/groups/hwcweb/handel/index.html`

[41] E. Hering, J. Gutekunst, U. Dyllong. *Handbuch der praktischen und technischen Informatik*. 2. Aufl., Springer, Berlin Heidelberg, 2000.

[42] S. Holmström. *SL - A Structural Design Language*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 371-376, Glasgow, GB, August 1999.

[43] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting. *A CAD Suite for High-Performance FPGA Design*. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, S. 12-23, Napa, USA, April 1999.

[44] IEEE. *EDIF Specification*.
`http://www.ieee.com`

[45] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Order Number 243190, 1997.

[46] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Order Number 243191, 1997.

[47] J. A. Jacob, P. Chow. *Memory Interfacing and Instruction Specification for Reconfigurable Processors*, In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, S. 145-154, 1999.

[48] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.

[49] J. Keller, W. J. Paul. *Hardware Design*. B. G. Teubner, Stuttgart, 1995.

[50] K. Kennedy. *Telescoping Languages: A Compiler Strategy for Implementation of High-Level Domain-Specific Programming Systems*. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, S. 297-304, Cancun, Mexico, Mai 2000.

[51] A. Koch. *Enabling Automatic Module Generation for FCCM Compilers*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 274-275, Napa, USA, April 1999.

[52] E. Lechner, S. A. Guccione. *The Java Environment for Reconfigurable Computing*. In *Field Programmable Logic and Applications (FPL)*, S. 284-293, London, GB, September 1997.

[53] G. Lienhart. *Entwicklung eines Videokompressionssystems für Videokonferenzanwendungen entsprechend dem ITU H.263-Standard auf dem FPGA-Prozessor microEnable*. Diplomarbeit im Studiengang Physik, Ruprecht-Karls-Universität Heidelberg, August 1999.

[54] W. Luk, S. McKeever. *Pebble: A Language for Parameterised and Reconfigurable Hardware Design*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 9-18, Tallin, Estonia, August 1998.

[55] W. Luk, N. Shirazi, P.Y.K. Cheung. *Modelling and Optimising Run-Time Reconfigurable Systems*. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, April 1996.

[56] P. Lysaght, J. Stockwood. *A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays*. In *IEEE Transactions on VLSI Systems*, September 1996.

[57] R. Macketanz, W. Karl. *JVX - A Rapid Prototyping System Based on Java and FPGAs*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 99-108, August 1998.

[58] P. Mackinlay, P. Cheung, W. Luk, R. Sandiford. *Riley-2: A Flexible Platform for Codesign and Dynamic Reconfigurable Computing Research*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 91-100, London, GB, September 1997.

[59] T. Maruyama, M. Takagi, T. Hoshino. *Hardware Implementation Techniques for Recursive Calls and Loops*. In *Field Programmable Logic and Applications (FPL)*, S. 450-455, Glasgow, GB, September 1999.

[60] R. B. Maunder, Z. A. Salcic, G. G. Coghill. *High-Level Hierarchical HDL Synthesis of Pipelined FPGA-Based Circuits Using Synchronous Modules*. In *Field Programmable Logic and Applications (FPL)*, S. 377-384, Glasgow, GB, September 1999.

[61] G. McGregor, P. Lysaght. *Extending Dynamic Circuit Switching to Meet the Challenges of New FPGA Architectures*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 31-40, September 1997.

[62] O. Mencer, M. Morf, M. J. Flynn. *PAM-Blox: High Performance FPGA Design for Adaptive Computing*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 167-174, Napa, USA, April 1998.

[63] O. Mencer, H. Hübert, M. Morf, M. J. Flynn. *StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox*. In *Field Programmable Logic and Applications (FPL)*, S. 595-604, Villach, Österreich, August 2000.

[64] Mentor Graphics. *Leonardo Spectrum*.
`http://www.mentor.com/leonardospectrum/datasheet.pdf`

[65] Model Technology.
`http://www.modelsim.com`

[66] P. Moisset, P. Diniz, J. Park. *Matching and Searching Analysis for Parallel Hardware Implementation on FPGAs*, In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, S. 125-133, Monterey, USA, 2001.

[67] L. Moll, M. Shand. *Systems Performance Measurement on PCI Pamette*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 125-133, Napa, USA, April 1997.

[68] A. Oliveira, A. Melo, V. Sklyarov. *Specification, Implementation and Testing of HFSMs in Dynamically Reconfigurable FPGAs*. In *Field Programmable Logic and Applications (FPL)*, S. 313-322, Glasgow, GB, September 1999.

[69] OXFORD hardware compilation group. *The Handel Language. Technical Report*. Oxford University, 1997.

[70] I. Page. *Hardware Compilation, Configurable Platforms and ASICs for Self-Validating Sensors*. In *Field Programmable Logic and Applications (FPL)*, S. 418-427, London, GB, September 1997.

[71] R. Paul. *Elektrotechnik und Elektronik für Informatiker, Band 2: Grundgebiete der Elektronik*. B. G. Teubner, Stuttgart, 1995.

[72] PLX Technology. *PCI 9080 Datasheet*. `http://www.plxtech.com/download/9080/databook/9080db-106.pdf`

[73] T. Price, C. Patterson. *Reconfigurable Breakpoints for Co-debug*. In *Field Programmable Logic and Applications (FPL)*, S. 473-482, Belfast, Nordirland, August 2001.

[74] F. P. Prosser, D. E. Winkel. *The Art of Digital Design*. 2. Aufl., Prentice Hall, Englewood Cliffs, 1987.

[75] F. Renner, J. Becker, M. Glesner. *Field Programmable Communication Emulation and Optimization for Embedded System Design*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 58-67, Villach, Österreich, August 2000.

[76] D. Robinson, G. McGregor, P. Lysaght. *New CAD Framework Extends Simulation of Dynamically Reconfigurable Logic*. In *Proceedings of the Workshop on Field-Programmable Logic and Applications (FPL)*, S. 1-8, August 1998.

[77] T. Sasao (ed.). *Logic Synthesis and Optimization*. 4. Aufl., Kluwer, Boston, 1998.

[78] S. Sawitzki, A. Gratz, R. Spallek. *Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays*. In *Field Programmable Logic and Applications (FPL)*, S. 411-415, Tallin, Estonia, September 1998.

[79] M. Schader. *Die Programmiersprache C++*. Springer, Berlin Heidelberg, 2002.

[80] W. Schiffmann, R. Schmitz. *Technische Informatik 1*. 2. Aufl., Springer, Berlin Heidelberg, 1993.

[81] W. Schiffmann, R. Schmitz. *Technische Informatik 2*. 2. Aufl., Springer, Berlin Heidelberg, 1994.

[82] T. Schmitz. *Entwicklung eines neuen RAM-Managements für FPGA-Prozessoren*. Diplomarbeit im Studiengang Physik, Ruprecht-Karls-Universität Heidelberg, Oktober 2000.

[83] M. Shand. *A Case Study of Algorithm Implementation in Reconfigurable Hardware and Software*. In *Field Programmable Logic and Applications (FPL)*, S. 333-343, London, GB, September 1997.

[84] J. Šilc, B. Robič, T. Ungerer. *Processor Architecture*. Springer, Berlin Heidelberg, 1999.

[85] Silicon Software GmbH. *microEnable-Koprozessor*.
`http://www.silicon-software.com`

[86] V. Sklyarov, J. Fonseca, R. Monteiro, A. Oliveira, A. Melo, N. Lau, I. Skliarova, P. Neves, A. Ferrari. *Development System for FPGA-Based Digital Circuits*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 266-267, Napa, USA, April 1999.

[87] V. Sklyarov, R. S. Monteiro, N. Lau, A. Melo, A. Oliveira, K. Kondratjuk. *Integrated Development Environment for Logic Synthesis Based on Dynamically Reconfigurable FPGAs*. In *Field Programmable Logic and Applications (FPL)*, S. 19-28, Tallin, Estonia, September 1998.

[88] G. Snider, B. Shackleford, R. J. Carter. *Attacking the Semantic Gap Between Application Programming Languages and Configurable Hardware*, In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, S. 115-124, Monterey, USA, 2001.

[89] D. Soderman, Y. Panchul. *Implementing C Algorithms in Reconfigurable Hardware using C2Verilog*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 339-342, Napa, USA, April 1998.

[90] Sun MicroSystems. *Specification of the JAVA Virtual Machine*.
`http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

[91] Synopsys.
`http://www.synopsys.com`

[92] Synopsys. *FPGA Express / FPGA Compiler II*.
`http://www.synopsys.com/products/fpga/fpga_compilerII_ds.html`

[93] Synplicity.
`http://www.synplicity.com`

[94] SystemC. *Functional Specification for SystemC 2.0, Version 2.0-P*. 2001.
`http://www.systemc.org`

[95] SystemC. *SystemC homepage*.
`http://www.systemc.org`.

[96] SystemC. *SystemC: User Guide, Version 2.0, 2001*.
`http://www.systemc.org`

[97] U. Tietze, C. Schenk. *Halbleiter-Schaltungstechnik*. 11. Aufl., Springer, Berlin Heidelberg, 1999.

[98] H. Touati, M. Shand. *PamDC: A C++ Library for the Simulation and Generation of XILINX FPGA Designs*.
`http://www.research.compaq.com/SRC/pamette/PamDC.pdf`

[99] T. Trenschel. *Blob-Analyse - Bestimmung von Formparametern beliebig geformter Objekte auf FPGAs in Echtzeit*. Diplomarbeit im Studiengang Physik, Ruprecht-Karls-Universität Heidelberg, August 2000.

[100] Trolltech. *Die QT-Bibliothek*.
`http://www.trolltech.com`

[101] T. Ungerer. *Datenflußrechner*. Teubner, Stuttgart, 1993.

[102] Universität Mannheim. *RACE-1-Koprozessor*.
`http://www.ti.uni-mannheim.de`

[103] Verilog Resources.
`http://www.verilog.com`

[104] VHDL Resources.
`http://www.vhdl.org`

[105] M. Wannemacher. *Das FPGA-Kochbuch*. Thomson, Bonn, 1998.

[106] M. Weinhardt, W. Luk. *Pipeline Vectorization for Reconfigurable Systems*. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, S. 52-62, Napa, USA, April 1999.

[107] K. Weiß, T. Steckstor, G. Koch, W. Rosenstiel. *Exploiting FPGA-Features during the Emulation of a Fast Reactive Embedded System*. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, S. 235-242, 1999.

[108] R. Wilhelm, D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer, Berlin Heidelberg, 1992.

[109] H. Wupper, U. Niemeyer. *Elektronische Schaltungen 2*. Springer, Berlin Heidelberg, 1996.

[110] XILINX Inc. *Chipscope*.
`http://www.xilinx.com/xilinxonline/chipscope.htm`

[111] XILINX Inc. *JBits*.
`http://www.xilinx.com/xilinxonline/jbits.htm`

[112] XILINX Inc. *Libraries Guide (EDIF Library)*.
`http://www.xilinx.com/support/sw_manuals/xilinx5/download/lib.zip`

[113] XILINX Inc. *OPB Synchronous SDRAM Controller*.
`http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb_sdram.pdf`

[114] XILINX Inc. *Spartan and Spartan-XL Families Field Programmable Gate Arrays*.
`http://direct.xilinx.com/bvdocs/publications/ds060.pdf`

[115] XILINX Inc. *Spartan-II 2.5V FPGA Family: Functional Description*.
`http://direct.xilinx.com/bvdocs/publications/ds001_2.pdf`

[116] XILINX Inc. *Spartan-II FPGA Family Configuration and Readback*.
`http://direct.xilinx.com/bvdocs/appnotes/xapp176.pdf`

[117] XILINX Inc. *The ABEL-HDL Language*.
`http://toolbox.xilinx.com/docsan/xilinx5/help/state/html/abel_hdllanguage.htm`

[118] XILINX Inc. *Using the XC4000 Readback Capability*.
`http://www.xilinx.com/bvdocs/appnotes/xapp015.pdf`

[119] XILINX Inc. *Virtex 2.5 V Field Programmable Gate Arrays*.
`http://direct.xilinx.com/bvdocs/publications/ds003-2.pdf`

[120] XILINX Inc. *Virtex-II Platform FPGAs: Detailed Description*.
`http://direct.xilinx.com/bvdocs/publications/ds031-2.pdf`

[121] XILINX Inc. *Virtex-II Pro Platform FPGAs: Functional Description*.
`http://direct.xilinx.com/bvdocs/publications/ds083-2.pdf`

[122] XILINX Inc. *XC4000E and XC4000X Series Field Programmable Gate Arrays*.
`http://direct.xilinx.com/bvdocs/publications/4000.pdf`

[123] R. Zoz. *Eine Hochsprachen-Programmierumgebung für FPGA-Prozessoren*.
Inaugural-Dissertation, Ruprecht-Karls-Universität Heidelberg, April 1997.