Einführung in

AHDL

(Altera Hardware Description Language)

A. Klindworth
Arbeitsbereich „Technische Grundlagen der Informatik"
Fachbereich Informatik der Universität Hamburg
September 1995

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Feldprogrammierbare ICs und Hardwarebeschreibungssprachen

Für den Entwurf anwendungsspezifischer integrierter Schaltungen (ASICs) steht häufig weniger ein optimales Entwurfs*ergebnis*, als vielmehr der schnelle, sichere und damit kostengünstige Entwurfs*vorgang* im Vordergrund. Hierin ist der wesentliche Grund für die rasche Verbreitung von FPGAs (**F**ield **P**rogrammable **G**ate **A**rrays) und CPLDs (**C**omplex **P**rogrammable **L**ogic **D**evices) zu sehen, von ICs also, deren Funktionalität ähnlich wie bei PROM-Speichern mit Hilfe einfacher Geräte vom Anwender konfiguriert werden kann. Der Entwerfer eines ICs erhält dadurch die Möglichkeit, unmittelbar nach Abschluß des Entwurfs selbst einen Prototypen herzustellen und in der Zielumgebung zu testen, wodurch das Auffinden und Korrigieren von Designfehlern stark vereinfacht wird.

Unter dem Aspekt der Vereinfachung des Entwurfs ist auch das Aufkommen automatischer und semi-automatischer Synthesewerkzeuge zu sehen. Sie nehmen dem Entwerfer reine Fleißarbeiten, wie etwa die Minimierung von Schaltnetzen und Schaltwerken, Laufzeitoptimierungen u.ä. ab. Die Aufgabe des Entwerfers verlagert sich dadurch von der Logik- auf die Register-Transfer-Ebene, d.h. eine Beschreibung, die das Ergebnis des Entwurfs einer digitalen Schaltung darstellt, ist nicht mehr eine Netzliste mit Logikgattern und Flipflops, sondern eine Struktur aus RT-Komponenten wie „Register", „Addierer", „Zähler", für die ledigich das Verhalten spezifiziert werden muß.

Die Verlagerung des Entwurfsgeschehens auf eine höhere Ebene der Abstraktion und die Forderung, Struktur und Verhalten gleichberechtigt nebeneinander spezifizieren zu können, zieht eine zunehmende Verbreitung textueller Schaltungsbeschreibungen als Eingabe für CAD-Werkzeuge im Bereich des IC-Designs nach sich. Der traditionelle *Schematic Entry*, also die Erstellung eines Schaltplans mit Hilfe eines Grafik-Editors, wird zwar von allen Tools nach wie vor unterstützt. Immer wichtiger werden jedoch formale Sprachen, die sog. Hardwarebeschreibungssprachen. Sie sind sehr viel besser geeignet, das Verhalten einer Schaltungskomponente zu spezifizieren, stehen den graphischen Methoden auch hinsichtlich der Möglichkeiten, Strukturen zu beschreiben, kaum nach und vereinfachen überdies den Datenaustausch zwischen verschiedenen Design-Tools. Letzteres setzt natürlich eine Standardisierung der Sprachen voraus, wie sie auch bei höheren Programmiersprachen vollzogen worden ist. Einen solchen, bereits etwas älteren Sprachstandard zur Beschreibung von Netzlisten stellt EDIF (**E**lectronic **D**esign **I**nterchange **F**ormat) dar. Neuere Sprachen mit der Möglichkeit, Verhalten einfach zu beschreiben, sind Verilog und vorallem VHDL (**V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage).

VHDL, deren Entwicklung 1983 vom US-Department of Defense initiiert wurde, ist seit 1987 als IEEE Standard 1076 genormt und kann als der quasi-Standard im Bereich der Hardwarebeschreibungssprachen angesehen werden. Ein Standard muß es (fast) jedem recht machen. um allgemeine Anerkennung zu finden. Dies gilt auch

VHDL, die alle wesentlichen Aspekte eines IC-Designs auf verschiedenen Ebenen der Abstraktion darzustellen erlaubt. Das Verhalten eines einzelnen Transistors kann in VHDL ebenso modelliert werden, wie das eines Prozessorkerns. Dem enormen Vorteil, ein konsistentes, über alle Ebenen des Entwurfs durchgängig verwendbares Beschreibungsmittel zur Verfügung zu haben, steht der Nachteil einer enormen Komplexität der Sprache gegenüber. Dieses stellt den Entwerfer nicht nur vor die schwierige Aufgabe, zwischen mehreren alternativen Formen der Beschreibung, die gleichwertig oder eben auch nicht gleichwertig sind, wählen zu müssen. VHDL-Beschreibungen sind zudem recht länglich, weil zum einen die Schlüsselwörter recht lang sind und zum anderen die strenge Typisierung der Sprache entsprechende Deklarationen erfordert und das Interfacing zwischen Strukturkomponenten erschwert.

Für ihr Entwurfssystem MAX+plusII hat die Firma Altera eine eigene Hardwarebeschreibungssprache mit Namen AHDL (**A**ltera **H**ardware **D**escription **L**anguage) zur Verfügung gestellt. AHDL ist eine Weiterentwicklung der Sprache ABEL, die für die Beschreibung von PLA- und GAL-Schaltungen weithin in Gebrauch ist. AHDL ist eine einfache Sprache, die aufgrund weniger impliziter Datentypen Beschreibungen von Struktur und Verhalten digitaler Schaltungen erlaubt, die sehr viel kompakter sind, als entsprechende Beschreibungen in VHDL.

In den nachfolgenden Kapiteln werden die wesentlichen Sprachkonstrukte von AHDL und ihre Verwendung vorgestellt. Der Leserin bzw. dem Leser mit Programmierkenntnissen wird dabei die Ähnlichkeit textueller Hardwarebeschreibungen mit hochsprachlichen Programmen auffallen. Er/sie wird daher das Konzept der textuellen Beschreibung als natürlich empfinden, sollte darüber jedoch die von Programmen abweichende Semantik einer Hardwarebeschreibung nicht vergessen.

> **AHDL ist, wie jede Hardwarebeschreibungssprache, eine konkurrente Sprache. Alle Anweisungen in allen Dateien eines Design-Projektes werden parallel „ausgeführt".**

Die folgende Einführung in die Benutzung der Sprache AHDL gliedert sich wie folgt: In den Kapiteln 2 und 3 werden die wichtigsten Sprachkonstrukte zusammenfassend erläutert. Neben der formalen Definition ihrer Syntax und Semantik wird ihre jeweilige Verwendung anhand von Anwendungsbeispielen demonstriert. Anschließend gibt Kapitel 4 Hinweise zur Gestaltung von AHDL-Beschreibungen, ehe in Kapitel 5 einige Beispiel-Designs diskutiert werden. Kapitel 6 schließlich befaßt sich mit den Möglichkeiten die vom MAX+plusII Compiler aus der AHDL-Beschreibung synthetisierte Struktur gezielt zu beeinflussen.

1.2 Hierarchischer Entwurf integrierter Schaltungen

Für den Entwurf integrierter Schaltungen hat sich eine hierarchische Vorgehensweise allgemein bewährt: Zunächst wird das Verhalten der Gesamtschaltung möglichst exakt definiert und anschließend eine Struktur aus einfacheren Komponenten entworfen, die das gewünschte Verhalten realisiert. Das Verhalten dieser Komponenten muß hierzu natürlich ebenfalls definiert sein und ist anschließend seinerseits durch eine Struktur aus noch einfacheren Komponenten zu implementieren. Diese schrittweise Verfeinerung erlaubt im Idealfall, die einzelnen Komponenten weitgehend unabhängig voneinander entwerfen zu können, so daß die Komplexität einer in einem Entwurfsschritt zu entwerfenden (Teil-)Schaltung auf ein handhabbares Maß reduziert wird. Der Entwurfsvorgang ist insgesamt abgeschlossen, wenn für jede der benutzten Komponenten eine Struktur aus elementaren Komponenten definiert worden ist. Bei Benutzung der Sprache AHDL werden diese elementaren Komponenten als *Primitive* bezeichnet. Zu ihnen gehören neben logischen Verknüpfungen wie UND, ODER und exklusiv-ODER verschiedene Typen von Flipflops und Buffern sowie High-Level-Operatoren wie „+" und „-".

Die hierarchische Gliederung einer Schaltungsbeschreibung wird von AHDL voll unterstützt. Der Benutzer erhält über geeignete Sprachkonstrukte die Möglichkeit, neben den Primitiven auch eigene Schaltungen oder Schaltungen aus Bibliotheken in einer AHDL-Beschreibung zu verwenden.

Die erste Entwurfsphase stellt sich demnach als ein Top-down - Vorgehen dar. Nachdem Strukturen auf den unteren Abstraktionsebenen entworfen worden sind, wird jedoch in der Regel die Änderung vorhergehender

Verhaltensspezifikationen als zweckmäßig erscheinen. Diese bottom-up - Effekte treten beim Entwurf jeder integrierten Schaltung auf, sind jedoch bei Schaltungen auf Basis von FPGAs und CPLDs besonders groß, da hier die bereits vorgegebene physikalische Struktur der Funktionsblöcke bestimmte Funktionen und/oder spezifische Formen der Realisierung gegenüber anderen begünstigt. Bei den meisten Typen feldprogrammierbarer ICs sind beispielsweise die speichernden Elemente (Flipflops) in den Funktionsblöcken **hinter** den Ressourcen zur Realisierung logischer Verknüpfungen angeordnet. D.h. vor speichernden Elementen einer Schaltung steht immer Logik zur Verfügung und ein guter Entwerfer wird deshalb darauf achten, daß nach Möglichkeit die in einem Datenpfad zwischen zwei Registern durchzuführenden logischen Verknüpfungen gerade so komplex sind, daß sie die vor dem Zielregister ohnehin verfügbare Logik gut ausnutzen.

> **Ohne genaue Kenntnis über die physikalischen Struktur der Funktionsblöcke und daraus resultierenden bottom-up - Einflüssen für den Entwurf auf höheren Ebenen kann eine qualitativ gute Schaltung auf Basis von FPGAs oder CPLDs nicht entworfen werden.**

1.3 Entwerfen komplexer Designs mit AHDL und MAX+plusII

Bevor im folgenden auf die Teile einer AHDL-Beschreibung im Einzelnen eingegangen wird, sollen an dieser Stelle einige Tips für das prinzipielle Vorgehen beim Entwurf mit AHDL gegeben werden. Dabei wird unterstellt, daß bei dem zu realsierenden Design der Bedarf an Ressourcen, gemessen an der Anzahl der benötigten Logikzellen, kritisch ist (Platzoptimalität). Bei Designs, die auf Geschwindigkeit hin zu optimieren sind, ist das beschriebene Entwurfsvorgehen in seinen Grundzügen ebenfalls zweckmäßig.

Die vom Autor gemachten Erfahrungen beim Entwurf mit MAX+plusII und AHDL lassen folgende Vorgehensweise sinnvoll erscheinen:

1. Bottom-up Entwurf der in dem Design-Projekt benötigten Komponenten.
2. Separates Compilieren jeder Komponente der Design-Hierarchie für die angestrebte Bausteinserie. Dabei einen Baustein als Ziel-Device auswählen, in dem die Komponente problemlos untergebracht werden kann.
3. Feststellen des Ressourcen-Bedarfs für jede Komponente. Dabei auf Optimalität der Realisierung unter Einbeziehung der voraussichtlichen Umgebung, in die die Komponente eingebettet ist, achten. Ggf. die Beschreibung der Komponente dahingehend optimieren.
4. Feststellen, ob bei Komponenten, die zuvor entworfene Komponenten instanziieren, letztere durch die Umgebung aufwendiger realisiert werden als bei der stand-alone Compilierung. Ist dieses der Fall, so können durch explizites Einfügen von in separaten und manuell allozierten Logikzellen gebildeten Zwischensignalen häufig Ressourcen eingespart werden.
5. Nach der Compilierung des gesamten Designs, detaillierte Analyse des Report-Files. Feststellen und Beseitigen von Ineffizienzen und Ermitteln des Ressourcen-Bedarfs.
6. Festlegung der zu verwendenden Devices und ggf. Steuerung der Plazierung von Schaltungsteilen und insbesondere der Anordnung der Pins.

Wichtigster Punkt hierbei ist, daß man sich zunächst auf das logische Design und die Abbildung auf die Funktionsblöcke der Devices konzentriert. Der Einfluß von realisierungsabhängigen Randbedingungen, die durch beschränkte Kapazitäten von Devices oder durch die festliegende physikalische Zuordnung von Pins zu einzelnen Gruppen von Logikzellen in den Devices entstehen, sollte dabei so gering wie möglich gehalten werden. Faustregel:

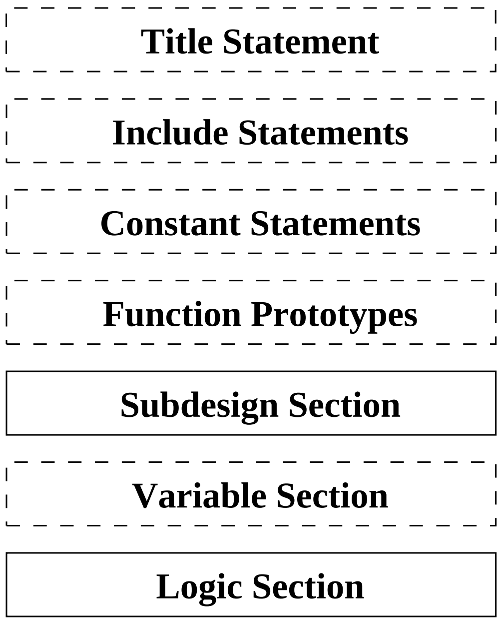
Zunächst dem Design-Projekt großzügig dimensionierte Devices zur Verfügung stellen und eine vollständig automatisch plazierte und ggf. partitionierte Realisierung erzielen. Anschließend dieses Ergebnis analysieren und, falls nötig, schrittweise solange modifizieren, bis das Resultat den Anforderungen genügt (Festlegen von Ziel-Device(s), manueller Eingriff in Plazierung und Logiksynthese, etc.).

Kapitel 2

Aufbau eines AHDL Design Files

Ein AHDL Design (ein sog. Design-*Projekt*) besteht aus einem oder mehreren Files, wobei jedes Design-File die vollständige Beschreibung einer Schaltungskomponente (eines sog. *Subdesigns*) enthält. Der Name eines Design-Files muß dabei mit dem Namen des in ihm beschriebenen Subdesigns übereinstimmen und darf höchstens 8 Zeichen lang sein. Als File-Extension wird *.tdf* (für **T**ext **D**esign **F**ile) erwartet.

Ein Design-File ist dabei wie folgt aufgebaut, wobei bis auf Subdesign und Logic Section alle Teile optional sind:



Die Reihenfolge, in der die einzelnen Teile in einem AHDL-File erscheinen müssen, ist nicht strikt festgelegt und kann geringfügig variiert werden. Im Sinne eines guten Design-Stils wird die Beibehaltung der vorgeschlagenen Reihenfolge jedoch empfohlen.

2.1 Title Statement

Das Title Statement hat folgende BNF-Syntax

```
<title_statement> ::= TITLE <string>;
```

und erlaubt dem Benutzer, der nachfolgenden Schaltungsbeschreibung einen Titel zu geben. Dieser Titel ist ein Kommentar, der vom MAX+plusII-Compiler in das Report-File eingefügt wird.

Das Title Statement ist optional.

2.2 Constant Statement

Das Constant Statement hat folgende BNF-Syntax

```
<constant_statement> ::= CONSTANT <identifier> = <number>;
```

wobei `<number>` eine Dezimalzahl oder eine Zahl zu einer anderen Basis (Binär, Octal oder Hexadezimal) ist. Letztere wird angegeben, indem der Ziffernstring in doppelte Anführungszeichen gesetzt und die Basisangabe in Form eines Buchstaben vorangestellt wird, z.B. `B"00010"` für eine Binärzahl, `O"307"` für eine Octalzahl und `H"A3f0"` für eine Hexadezimalzahl.

Eine in einem Constant-Statement deklarierte Konstante kann in nachfolgenden Teilen des AHDL-Files anstelle einer Zahl benutzt werden. (Allerdings nicht in jedem Kontext!)

Ein AHDL-File kann beliebig viele Constant-Statements (also auch keines) enthalten, die jedoch alle **vor** der Subdesign, Variable und Logic Section stehen müssen.

2.3 Function Prototypes

Mit dem Prototype-Statement werden die externen Anschüsse (*Ports*) von jenen Subdesigns deklariert, die in dem AHDL-File benutzt (*instanziiert*) werden und deren Beschreibung in anderen Design-Files vorliegt. Solche Teilschaltungen werden als *Makrofunktionen* bezeichnet. Die BNF-Syntax des Function Prototype Statements ist

```
<function_prototype> ::= FUNCTION <design_name> (<input_list>)
                             RETURNS (<output_list>);
```

`<design_name>` ist hierbei der Name der Makrofunktion, `<input_list>` die Liste der Input-Ports und `<output_list>` die Liste der bidirektionalen und Ausgangs-Ports. Namen und Bitbreite der Ports im Prototypen müssen mit denen in der Subdesign Section des zu `<design_name>` gehörenden Design-Files übereinstimmen. Die Reihenfolge, in der die Ports in der Prototypdeklaration aufgeführt werden, darf jedoch von der im Design-File der Makrofunktion abweichen. Für Instanzen des Subdesigns `<design_name>` ist dann die Reihenfolge in der Prototypdeklaration ausschlaggebend, wobei im Hinblick auf einen guten Design-Stil diese mit der im Design-File übereinstimmen sollte.

Nur in einem Prototyp deklarierte Funktionen können später in der Variable und der Logic Section verwendet werden. Eine Ausnahme hiervon bilden die *Primitive*, also jene einfachen Teilschaltungen, die in AHDL bereits fest definiert als Bestandteil der Sprache zur Verfügung stehen.

Ein AHDL-File kann beliebig viele Prototypdeklarationen (also auch keine) enthalten, die jedoch **vor** der Subdesign, Variable und Logic Section stehen müssen.

2.4 Include-Statement

Das Include-Statement hat folgende BNF-Syntax

```
<include_statement> ::= INCLUDE "<filename>";      ,
```

kann nur außerhalb aller anderen Sektionen eines AHDL-Files benutzt werden und darf keine Subdesign Section enthalten. Include-Files können dementsprechend nicht ihrerseits Include Statements enthalten. Die Extension des Dateinamens muß *.inc* sein und wird ergänzt, wenn keine Extension angegeben ist. Ein Include-File wird zunächst in dem aktuellen Projekt-Directory gesucht. Ist es dort nicht vorhanden, wird in den Benutzer- und schließlich den MAXplus+II Standardbibliotheken nachgesehen.

Die engen Restriktionen für die Benutzung von Include-Files führt dazu, daß sie in der Regel nur Prototypen- und/oder Konstanten-Deklarationen enthalten.

2.5 Subdesign Section

Die Subdesign Section eines AHDL-Files enthält die Beschreibung der Teilschaltung, die in diesem File definiert wird. Hierzu gehört zunächst die Deklaration der Ein- und Ausgangssignale (*Ports*). Anschließend werden interne Knoten und Instanzen von Makrofunktionen und/oder Primitiven deklariert, worauf in einem Anweisungsteil die Funktionalität der Teilschaltung als eine Mischung aus Struktur- und Verhaltensbeschreibung definiert wird.

Die BNF-Syntax der Subdesign-Section ist:

```
<subdesign_section> ::=  SUBDESIGN <design_name> (
                           {<signal_list>;}
                           <signal_list> [;]
                         )

<signal_list> ::=  <port_list> : <port_type>

<port_list> ::=  <port> {,<port>}

<port> ::=  <port_name> [ "[" <decimal_number> [..<decimal_number>] "]" ]

<port_type> ::=    INPUT [ = VCC | =GND ]
                 | OUTPUT
                 | BIDIR [ = VCC | = GND ]
                 | MACHINE INPUT
                 | MACHINE OUTPUT                    .
```

Bei der Subdesign-Section sind folgende Punkte zu beachten:

- <design_name> muß mit dem Namen des AHDL-Files übereinstimmen.

- In einem Top-Level Design-File sind die in den Signallisten aufgeführten Ports Device Pins, die beispielsweise in der Design-Section bestimmten Pin-Nummern zugeordnet werden können. In Lower-Level Design-Files geben die Ports nur Ein- und Ausgangssignale der Teilschaltung an. Da bei der Logiksynthese auch über Dateigrenzen hinweg logische Funktionen aus der Beschreibung extrahiert und anschließend minimiert werden, kann es insbesondere dazu kommen, daß Portsignale von hierarchisch untergeorneten Teilschaltungen „wegoptimiert" werden. Sie können bei der Simulation mit dem MAXplus+II Simulator dann nicht mehr angesprochen werden, es sei denn, man hat das interessierende Signal zuvor mit einem Probe versehen.

- Die Port-Typen MACHINE_INPUT und MACHINE_OUTPUT dienen dazu, ganze endliche Automaten in ein Design-File zu importieren bzw. aus diesem zu exportieren. Ansonsten stehen als Ports nur Binärsignale zur Verfügung, wobei durch Angabe von Arraygrenzen im Anschluß an einen Port-Namen auch Busse von Binärsignalen definiert werden können.

- Für Ports vom Typ INPUT und BIDIR können default-Werte angegeben werden (VCC oder GND), die gültig sind, wenn durch die Umgebung der Teilschaltung kein anderer Wert eingestellt wird. Auf diese Weise können offene Eingänge mit definierten Signalpegeln versehen werden. Dieses gilt jedoch nicht für Device-Pins, da Altera EPLDs und FPGAs CMOS-Bausteine sind und somit nicht angeschlossene Input-Pins keinen definierten Signalpegel haben.

2.6 Variable Section

Variablen bezeichnen interne Knoten, endliche Automaten oder Instanzen von Makrofunktionen oder Primitiven. Sie werden in der Variable Section einer AHDL-Beschreibung deklariert, deren Syntax folgendermaßen definiert ist:

```
<variables_section> ::=  VARIABLE
                            <variable_declaration>;
                           {<variable_declaration>;}

<variable_declaration> ::=     <state_machine_declaration>
                             | <state_machine_alias_declaration>
                             | <variable_list> : <variable_type>

<variable_list> ::=  <variable> {,<variable>}

<variable> ::=  <variable_name> ["[" <decimal_number> [..<decimal_number>] "]"]

<variable_type> ::=    NODE
                     | <macrofunction_name>
                     | <primitive_name>

<state_machine_declaration> ::=  <state_machine_name> : <state_machine>

<state_machine_alias_declaration> ::=  <state_machine_name> : MACHINE;
```

2.6.1 Nodes

Variablen vom Typ `NODE` sind Binärsignale bzw. Busse von Binärsignalen, wenn es sich um Array-Variablen handelt. Sie dienen dazu, interne Verbindungen zwischen Strukturkomponenten (Instanzen von Makrofunktionen und Primitiven sowie State Machines) herzustellen oder Zwischensignale für komplexe Boolesche Ausdrücke bereitzustellen.

2.6.2 Instanzen

Um einen hierarchisch strukturiertes Entwurfsvorgehen zu ermöglich, können Subdesigns in anderen Design-Files benutzt werden. Man spricht in diesem Zusammenhang von der „Instanziierung von Makrofunktionen". Eine Möglichkeit der Instanziierung ist die Deklaration einer Variable, deren Typ die Makrofunktion ist. Der Variablenname bezeichnet dann den Namen der Instanz und die Ports dieser Instanz können über eine Punkt-Notation

```
<instance_name>.<port_name>
```

angesprochen werden.

Auch die Deklaration von Instanzen-Arrays (etwa für Register) ist möglich. Hierzu sind dem Variablennamen lediglich die Array-Grenzen hinzuzufügen. Eine einzelne Instanz des Arrays kann dann über eine entsprechende Indizierung der Form

<instance_array_name>"["<index>"]" oder einfach <instance_array_name><index>

selektiert werden.

Für die Instanziierung einer Makrofunktion per Variablendeklaration soll an dieser Stelle noch ein Beispiel angegeben werden, um die Zusammenhänge zu verdeutlichen.

Es sei eine Instanz der Makrofunktion `compare` zu erzeugen, die den Namen `comp` erhalten soll. Hierzu deklariert man einfach

```
VARIABLE comp: compare;
```

Für die Makrofunktion `compare` muß ein Design-File (mit Namen *compare.tdf*) existieren, entweder in dem Projekt-Directory oder einer der eingestellten Bibliotheken. Die Ports der Makrofunktion `compare`, müssen außerdem vor der Subdesign Section in einem Prototypen deklarariert worden sein, z.B. mit

```
FUNCTION compare (a[3..0], b[3..0]) RETURNS (less, equal, greater);
```

Die Ports der Instanz `comp` können nach diesen Deklarationen mit

```
comp.a[] , comp.b[] , comp.less , comp.equal , comp.greater
```

in der Logic Section angesprochen und in der gleichen Weise wie Nodes benutzt werden. Auch die Extraktion von Busteilen und die Gruppierung von Ports in der Form

`comp.a[2..1] , comp.b1` und `comp.(less, greater)`

sind zulässig.

In der gleichen Weise wie Makrofunktionen können auch Primitive instanziiert werden. Als Besonderheit kommt hier hinzu, daß alle Primitive nur einen einzigen Output-Port aufweisen. Deshalb kann auf der rechten Seite einer Zuweisung bzw. in einem Booleschen Ausdruck der Name einer Instanz eines Primitivs ohne hinzufügen eines Port-Namens benutzt werden, um den einzigen Output-Port anzusprechen. Darüberhinaus ist für alle Primitive außer `JKFF`, `JKFFE`, `SRFF` und `SRFFE` einer der Input-Ports implizit als primärer Input definiert. Auf der linken Seite einer Zuweisung kann daher der Name einer Instanz eines Primitivs ohne Port-Name angegeben werden, um diesen primären Input auszuwählen. Bei Flipflops ist dieser primäre Input der Dateneingang `d`.

Zur Demonstration der Besonderheiten im Zusammenhang mit Instanzen von AHDL-Primitiven folgt wiederum ein einfaches Beispiel:

Der Prototyp des D-Flipflops ist

```
FUNCTION DFF (d, clk, clrn, prn) RETURNS (q);
```

und ist, da es sich um ein Primitiv handelt, implizit deklariert. Ein entsprechendes Prototyp-Statement wird also in dem Design-File, indem `DFF` instanziiert werden soll, nicht benötigt. Der folgende Ausschnitt aus einem Design-File zeigt die Verwendung impliziter Instanzen-Ports, worauf insbesondere auf die Verhältnisse bei dem Instanzen-Array `counter` hingewiesen sei.

```
VARIABLE
  a, b: DFF;
  counter[2..0]: DFF;
BEGIN
  ...

  a = b;  % entspricht a.d = b.q %

  counter[].clk = takt;  % Verbindet den Takteingang 'clk' aller drei    %
                         % Instanzen von 'counter' mit dem Signal 'takt' %

  counter[] = counter[] +1;
    % entspricht  (counter[2].d, counter[1].d, counter[0].d) =           %
    %              (counter[2].q, counter[1].q, counter[0].q) + B"001"; %
  ...

END;
```

2.6.3 State Machines

In AHDL können endliche Automaten (State Machines) beschrieben werden. Hierzu werden in der Variable Section zunächst der Name des Automaten sowie Namen für die einzelnen Zustände deklariert. Diese Deklarationen können durch die Angabe von Zustandscodes ergänzt werden. Ansonsten ermittelt der Compiler eine günstige Zustandscodierung unter Berücksichtigung der in der Logic Section befindlichen Beschreibung von Übergangs- und Ausgangsfunktion des Automaten, wobei jeder Zustand durch einen Binärvektor der Länge $\lceil ld \text{<number\_of\_states>} \rceil$ repräsentiert wird.

Die Syntax für die Deklaration einer State Machine ist

```
<state_machine_declaration> ::= <state_machine_name> : <state_machine>

<state_machine> ::=  MACHINE  [<state_machine_bits>]  <states>

<state_machine_bits> ::=  OF BITS ( <state_machine_port_list> )

<states> ::=  WITH STATES ( <state>, {,<state>} )

<state> ::=  <state_name> [ = <state_value> ]

<state_value> ::=  <number> | <constant_name>

<state_machine_port_list> ::=  <state_machine_port> {,<state_machine_port>}

<state_machine_port> ::=
  <port_name> ["[" <decimal_number> [..<decimal_number>] "]" ]
```

Anhand eines exemplarischen Ausschnitts aus einem Design-File soll die Deklaration von State Machines in AHDL illustriert werden.

```
VARIABLE

  fsm1 : MACHINE OF BITS (q2, q1, q0)
           WITH STATES (
             init = B"000",
             state1 = B"001",
             state2 = B"110"
           );

  fsm2 : MACHINE WITH STATES ( init,
                               do_something,
                               do_something_else,
                               do_some_more,
                               waiting
                             );
```

Die erste Deklaration erzeugt einen endlichen Automaten mit Namen `fsm1` und drei Zustandsbits (`q2`, `q1` und `q0`). Der Automat hat die drei Zustände `init`, `state1` und `state2` mit den angegebenen Zustandscodes. Die Zustandsbits können in der Logic Section wie gewöhnliche Knoten benutzt werden.

Die Deklaration von `fsm2` erzeugt einen Automaten mit fünf Zuständen, für die keine Zustandscodierung vorgegeben ist. Der Compiler wird deshalb im Zuge der Logiksynthese $\lceil ld\,5 \rceil = 3$ Zustandsflipflops erzeugen und eine günstige Zustandscodierung ermitteln.

Für beide Automaten sind in der Logic Section durch diese Deklarationen einige Machine Ports definiert, mit deren Hilfe die Automaten angesteuert werden können. Es sind dieses

clk: ein Takteingang, bei dessen Vorderflanke Zustandsübergänge in dem Automaten erfolgen.

reset: ein high-aktiver Input-Port, mit dem der Automat asynchron initialisiert, d.h. in den ersten Zustand in der Liste der Zustände, gebracht werden kann.

ena: ein high-aktiver Enable-Eingang, der bei Anliegen eines low-Pegels Zustandsübergänge mit Ausnahme des asynchronen Resets unterbindet.

Die Syntax zur Auswahl eines dieser Kontroll-Ports folgt der Punkt-Notation bei Instanzen, also

```
<state_machine_name>.<state_machine_port_name>    .
```

Der Zustand, in dem sich ein Automat befindet, kann durch Vergleich mit dem Zustandsnamen („==" oder „!=") festgestellt werden. Der als nächstes von dem Automaten einzunehmende Zustand wird durch Zuweisung des entprechenden Zustandsnamens an den Zustand des Automaten spezifiziert. In beiden Fällen wird zur Referenzierung des Zustands einfach der Name des Automaten angegeben. Die Anweisung

```
fsm2 = do_something_else;
```

bewirkt beispielsweise, daß der Automat `fsm2` bei der nächsten aktiven Vorderflanke des Taktes in den Zustand `do_something_else` übergeht. Durch die Anweisung

```
is_initialized = (fsm1 == init);
```

ist der Wert des Nodes `is_initialized` undefiniert, wenn der Zustand von `fsm1` undefiniert ist, `VCC`, wenn der Zustand `init` ist und `GND` für jeden anderen Zustand von `fsm1`.

Von den drei Kontrollsignalen eines Automaten (*clk*, *reset* und *ena*) muß nur *clk* in jedem Fall mit einem Signal verbunden werden. Für *reset* und *ena* sind `GND` bzw. `VCC` als default-Werte gesetzt. Lediglich in dem Fall, daß für den Initialisierungszustand ein Zustandscode ungleich Null spezifiziert wurde, muß auch der *reset*-Eingang des Automaten angesteuert werden, um den Automaten in diesen Anfangszustand bringen zu können. Zustandsflipflops werden, wie alle Flipflops in den Altera EPLDs, beim Einschalten der Versorgungsspannung zurückgesetzt, sodaß sich jeder Automat zunächst in dem Zustand mit dem Zustandscode Null befindet.

2.6.4 Machine Alias

Mit Hilfe einer Machine Alias Deklaration kann einer State Machine ein anderer Name gegeben werden, der an Stelle des in der State Machine Deklaration aufgeführten Namens benutzt werden soll. Dieses kann insbesondere dann sinnvoll sein, wenn die State Machine in einem anderen Design-File deklariert ist und von diesem über einen Output-Port vom Typ `MACHINE` exportiert wird. In dem folgenden Beispiel wird von der Makrofunktion `ss_def` eine State Machine `ss_out` exportiert. Mit der auf das Prototype Statement folgenden Machine Alias Deklaration und der Instanziierung des Automaten über einen inline-Funktionsaufruf, ist `ss` der Name für diese Instanz des Automaten. Dieser Name kann verwendet werden, als ob `ss` in diesem Design-File deklariert worden wäre.

```
FUNCTION ss_def (clock, reset, count) RETURNS (MACHINE ss_out);
...
```

```
VARIABLE ss: MACHINE;
...

BEGIN
  ...

  ss = ss_def (sys_clk, reset, !hold);
  is_in_state_s0 = (ss == s0);
  ...

END;
```

2.7 Logic Section

In der Logic Section eines AHDL Design-Files wird die Funktionalität des Subdesigns in einer Mischung aus Struktur- und Verhaltensbeschreibung spezifiziert. Hierzu stehen fünf Typen von Anweisungen sowie die Möglichkeit der Instanziierung von anderen Subdesigns zur Verfügung. Im einzelnen sind dies

- Zuweisungen,
- Defaults Statement,
- If Statements,
- Case Statements,
- Wertetabellen (Table Statements)
- und inline-Funktionsaufrufe.

Syntax und Semantik jedes dieser AHDL Konstrukte werden in den nachfolgenden Abschnitten erläutert. Sie ähneln entsprechenden Konstrukten in höheren Programmiersprachen, semantisch besteht jedoch ein wesentlicher Unterschied:

> **AHDL ist, wie jede Hardwarebeschreibungssprache, eine konkurrente Sprache. Alle Anweisungen in allen Logic Sections eines Design-Projektes werden parallel ausgeführt.**

Insbesondere sind alle Zuweisungen an ein und denselben Knoten gleichzeitig aktiv und werden logisch verodert, d.h. die beiden Zuweisungen

```
a = b;
a = c & d;
```

werden interpretiert als

```
a = b # (c & d);
```

Existiert für einen Knoten keine Zuweisung, so hat er den Wert `GND`. Dieser default-Wert eines Knotens kann aber mit Hilfe des Defaults Statements in `VCC` geändert werden, wodurch bei Mehrfachzuweisungen eine logische UND-Verknüpfung der zugewiesenen Werte erfolgt.

> **Zur besseren Übersichtlichkeit sollte man in einer AHDL-Beschreibung jedoch darauf achten, daß für einen Knoten stets genau eine Zuweisung wirksam ist.**

2.7.1 Zuweisungen

Zuweisungen werden in der Logic Section zur Repräsentation von Signalverbindungen innerhalb des Designs sowie zur Spezifikation der Übergangsfunktion einer State Machine benutzt. Die Syntax einer solchen Zuweisung lautet

```
<assignment> ::=    <binary_assignment>
                  | <state_machine_assignment>

<binary_assignment> ::=  <lgroup> = <rgroup>;

<state_machine_assignment> ::=    <state_machine_next_state_assignment>
                                | <state_machine_control_assignment>

<state_machine_next_state_assignment> ::=
  <state_machine_name> = <state_machine_state_name>;

<state_machine_control_assignment> ::=
    <state_machine_name>.clk = <rgroup>;
  | <state_machine_name>.reset = <rgroup>;
  | <state_machine_name>.ena = <rgroup>;
```

`<lgroup>` ist dabei ein einzelnes Binärsignal, ein Port des Subdesign bzw. einer Instanz oder eine Gruppe hiervon. `<rgroup>` ist ein Ausdruck, wobei die Wortbreite des Ergebnisses mit der von `<lgroup>` übereinstimmen muß. Ausnahmen: Ist `<rgroup>` eine Zahl, so wird die Wortbreite ihrer Binärdarstellung vor der Zuweisung an `<lgroup>` angepaßt. Ist `<rgroup>` hingegen ein einzelnes Binärsignal und `<lgroup>` eine Gruppe, so wird ersteres jedem Mitglied der Gruppe zugewiesen.

Bei Zuweisungen an Gruppen können in diesen Kommas als Platzhalter für nicht zugewiesene Gruppenmitglieder benutzt werden. In der Zuweisung

```
(a, , c, ) = (e, f, g, h)
```

erhält beispielsweise `a` den Wert von `e` und `c` den von `g`. Die Verwendung von Platzhaltern ist insbesondere dann sinnvoll, wenn `<rgroup>` ein inline-Funktionsaufruf ist. Hierzu ein weiteres Beispiel: Mit dem Namen `adder` sei eine Makrofunktion definiert, deren Prototyp

```
FUNCTION adder (a[3..0], b[3..0], c_in) RETURNS (s[3..0], c_out);
```

lautet. Soll bei einer Instanziierung von `adder` der Übertragsausgang `c_out` ignoriert werden, so kann dies durch eine Zuweisung mit einem inline-Funktionsaufruf erfolgen, wobei auf der linken Seite für den nicht zuzuweisenden Ausgang von `adder` ein Platzhalter angegeben wird:

```
(sum[3..0], ) = adder(arg1[], arg2[], GND);
```

Eine Zuweisung der Form

```
sum[3..0] = adder(arg1[], arg2[], GND);  % FALSCH %
```

hingegen verursacht eine Fehlermeldung, da die Wortbreiten auf der linken und auf der rechten Seite nicht übereinstimmen.

Für die Zuweisung von Folgezuständen und Kontrollsignalen sei im übrigen auch auf die Beispiele in Abschnitt 5.4 verwiesen.

2.7.2 Case Statement

Das Case Statement spezifiziert eine Liste von Alternativen, von denen jeweils eine in Abhängigkeit von dem Wert eines Ausdrucks ausgewählt wird. Die Syntax des Case Statements lautet:

```
<case_statement> ::=  CASE <rgroup> IS
                        WHEN <choices> => <statements>
                        {WHEN <choices> => <statements>}
                        [WHEN OTHERS => <statements>]
                      END CASE;

<choices> ::=  <constant_group> {, <constant_group>}
```

Stimmt der Wert von <rgroup> mit einer <constant_group> einer Alternative überein, so werden die Anweisungen aktiviert, die auf den Pfeil (`=>`) folgen. Die auf das optionale `WHEN OTHERS =>` folgenden Anweisungen kommen zur Ausführung, wenn keine der vorhergehenden Alternativen durch <rgroup> selektiert worden ist.

Beispiel für die Verwendung eines Case Statements:

```
CASE f[].q IS
  WHEN H"00"  => adr[] =  0;
                 s = a & b;
  WHEN H"01"  => count[] = count[] + 1;
  WHEN H"02",
       H"03",
       H"04"  => f[3..0] = adr[4..1];
  WHEN OTHERS => f[] = f[];
END CASE;
```

Im Zusammenhang mit State Machines wird zweckmäßigerweise meist ein Case Statement verwendet, um Übergangs- und/oder Ausgangsfunktion des Automaten zu spezifizieren. Zur Illustration diene hier folgendes Beispiel für einen einfachen Moore-Automaten:

```
SUBDESIGN moore (
  clk, reset: INPUT;
  d, q: OUTPUT;
)

VARIABLE ss: MACHINE WITH STATES (s0, s1);
BEGIN
  ss.clk = clk;
  ss.reset = reset;
```

```
    CASE ss IS    % Abhaengig vom aktuellen Zustand des Automaten ... %

      WHEN s0 => q = GND;
                 IF d             % Falls d = VCC              %
                   THEN ss = s1;  % in den Zustand s1 wechseln %
                 END IF;          % sonst in s0 bleiben        %

      WHEN s1 => q = VCC;
                 IF !d            % Falls d = GND              %
                   THEN ss = s0;  % in den Zustand s0 wechseln %
                 END IF;          % sonst in s1 bleiben        %
    END CASE;
  END;
```

Der Automat `ss` ist offensichtlich funktional äquivalent zu einem D-Flipflop.

Wichtig bei Case Statements für State Machines ist, daß `WHEN OTHERS` nicht benutzt werden kann, um aus illegalen Zuständen [1] heraus zu kommen. Die Anweisungen zu `WHEN OTHERS` werden nur für die definierten Zustände des Automaten ausgeführt, die nicht in einer der anderen Alternativen abgefragt wurden. Wird eine „Erholung" von illegalen Zuständen gewünscht, so muß ld(nr_of_states) gleich der Anzahl der Zustandsflipflops sein, d.h. man fügt der Deklaration des Automaten solange dummy-Zustände hinzu, bis diese Bedingung erfüllt ist.

Die beiden abschließenden Beispiele dieses Abschnitts demonstrieren dieses Vorgehen. In der ersten Version des Automaten findet keine „Erholung" von illegalen Zuständen statt. Für den Fall, daß der Automat in einen illegalen Zustand gerät (z.B. durch ungünstiges Timing an den Kontrolleingängen), ist der Folgezustand nicht definiert.

```
  SUBDESIGN insecure (
    clk, go: INPUT;
    ok: OUTPUT;
  )
  VARIABLE fsm: MACHINE WITH STATES (idle, one, two, three, four);
  BEGIN
    fsm.clk = clk;
    CASE fsm IS
      WHEN idle   =>  IF go THEN fsm = one; END IF;
      WHEN one    =>  fsm = two;
      WHEN two    =>  fsm = three;
      WHEN three  =>  fsm = four;
      WHEN OTHERS =>  fsm = idle;   % Dieser Folgezustand wird nur gesetzt, %
                                    % falls der aktuelle Zustand four ist   %
    END CASE;
  END;
```

Eine höhere Sicherheit (bei höherem Realisierungsaufwand) erhält der Automat, wenn eine „Erholung" von illegalen Zuständen vorgesehen wird. In der folgenden Realisierung von `fsm` ist der Automat längstens für die Dauer einer Taktperiode an `clk` in einem illegalen Zustand, ehe er wider in einen definierten Zustand wechselt.

[1] Der Zustand eines Automaten wird als *illegal* bezeichnet, wenn die zugehörige Wertebelegung der Zustandsflipflops keinem definierten Zustand entspricht. Man beachte, daß illegale Zustände nur auftreten können, wenn ld(nr_of_states) ungleich der Anzahl der Zustandsflipflops ist.

```
SUBDESIGN secure (
  clk, go: INPUT;
  ok: OUTPUT;
)
VARIABLE fsm: MACHINE WITH STATES (
                idle, one, two, three, four,
                dummy1, dummy2, dummy        % 3 dummy-Zust"ande f"ur die %
              );                             % illegalen Wertebelegungen  %
BEGIN                                        % der Zustandsflipflops      %
  fsm.clk = clk;
  CASE fsm IS
    WHEN idle   =>  IF go THEN fsm = one; END IF;
    WHEN one    =>  fsm = two;
    WHEN two    =>  fsm = three;
    WHEN three  =>  fsm = four;
    WHEN OTHERS =>  fsm = idle;   % aktiviert f"ur Zustand four und   %
                                  % alle vormals illegalen Zust"ande  %
  END CASE;
END;
```

2.7.3 Defaults Statement

Mit Hilfe des Defaults Statements können default-Werte für Variablen definiert werden. Die Syntax ist

```
<defaults_statement> ::=  DEFAULTS
                            <lgroup> = <constant_group>;
                            {<lgroup> = <constant_group>;}
                          END DEFAULTS;
```

Das Defaults Statement bewirkt, das den Knoten in der `<lgroup>` die Werte der `<constant_group>` zugewiesen werden, wenn in den übrigen Anweisungen der Logic Section keine anderen Zuweisungen erfolgen. Wird für einen Knoten kein anderer default-Wert spezifiziert, so ist dieser `GND`. Bei gleichzeitiger Zuweisung mehrerer Werte an einen Knoten, werden diese logisch verodert, wenn der default-Wert des Knotens `GND` ist, und verundet, wenn der default-Wert `VCC` ist.

Beachten Sie: Der über ein Defaults Statement spezfizierte default-Wert ist etwas anderes als der default-Wert eines Subdesign-Ports!

Die Verwendung des Defaults Statements wird abschließend anhand eines Beispiels illustriert:

```
BEGIN
  DEFAULTS
    a = GND;  % kann entfallen, da Voreinstellung %
    b = VCC;
  END DEFAULTS;

  IF c1 THEN
    a = a1;
    b = b1;
  END IF;
```

```
    IF c2 THEN
      a = a2;
      b = b2;
    END IF;
  END;
```

Diese Logic Section ist äquivalent zu

```
  BEGIN
    a = (c1 & a1) # (c2 & a2);
    b = (!c1 # b1) & (!c2 # b2);
  END;
```

2.7.4 If Statement

Das If Statement spezifiziert Folgen von Anweisungen, die in Abhängigkeit des Wertes eines Ausdrucks ausgeführt werden. Die Syntax ist

```
<if_statement> ::=  IF <rgroup> THEN
                      <statements>
                    {ELSIF <rgroup> THEN
                       <statements>}
                    [ELSE <rgroup> THEN
                       <statements>]
                    END IF;
```

`<rgroup>` ist hierbei ein Ausdruck, der zu einem einzelnen Binärwert auswertet. Ist dieser Wert `VCC`, so werden die Statements des THEN-Teils ausgeführt, anderenfalls die des (optionalen) ELSE- bzw. ELSIF-Teils. Wichtig: **Alle Bedingungen in einer IF-ELSIF-Kette werden** *parallel* **ausgewertet.** Das folgende Beispiel zeigt hierzu, wie der Compiler ein If Statement interpretiert.

```
IF    a THEN c = d;
ELSIF b THEN c = e;
ELSE         c = f;
END IF;
```

wird zunächst interpretiert als

```
IF a THEN c = d;
END IF;

IF !a & b THEN c = e;
END IF;

IF !a & !b THEN c = f;
END IF;
```

und daraus bei der Logiksynthese schließlich die logische Funktion

```
c = (a & d) # (!a & b & e) # (!a & !b & f)
```

abgeleitet.

2.7.5 Wertetabellen (Table Statements)

Das Table Statement gibt dem Benutzer die Möglichkeit, beliebige logische Funktionen sowie die Übergangs- und/oder Ausgangsfunktion von State Machines in Form einer Wertetabelle zu spezifizieren. Die Syntax hierzu lautet:

```
<table_statement> ::=  TABLE <inputs> => <outputs> ;
                         <input_values> => <output_values> ;
                         {<input_values> => <output_values> ;}
                       END TABLE;

<inputs> ::=  <rgroup> {, <rgroup>}

<outputs> ::=  <lgroup> {, <lgroup>}

<input_values> ::=  <constant_group> {, <constant_group>}

<output_values> ::=  <constant_group> {, <constant_group>}
```

Für die korrekte Verwendung des Table Statements sind folgende Punkte zu beachten:

- Für `<inputs>` und `<outputs>` ist jeweils nur eine Liste von Knoten, Subdesign-Ports und Gruppen von beiden, sowie Namen von State Machines zugelassen.
- Bei der Zuordnung von Eingangs- zu Ausgangswerten dürfen in `<input_values>` und `<output_values>` nur Konstanten benutzt werden. Insbesondere sind Ausdrücke oder Namen von Knoten oder Ports nicht erlaubt.
- Zwischen den Listen `<inputs>` und `<input_values>` besteht eine 1-zu-1-Korrespondenz, d.h. sie müssen gleichviele Elemente mit identischem Typ aufweisen. Gleiches gilt für die Beziehung zwischen `<outputs>` und `<output_values>`.
- Eingangswerte in `<input_values>` können auf den Wert `X` für „don't care" gesetzt werden, um anzuzeigen, daß die Ausgabe unabhängig von dem zugehörigen Eingang ist.
- Trifft keine der in dem Table Statement aufgeführten Eingangsbelegungen für die `<inputs>` zu, so wird den `<outputs>` auch kein Wert zugewiesen. Sie behalten dann den an anderer Stelle zugewiesenen bzw. den default-Wert.

Ein wichtiges Anwendungsfeld für Wertetabellen ist die Beschreibung von Decodern. Das folgende Subdesign zeigt hierzu die Beschreibung eines Adreßdecoders für ein 16-Bit Mikroprozessor System.

```
SUBDESIGN decoder (
  adr[15..0], m/io: INPUT;
  rom, ram, printer, io_port[1..0]: OUTPUT;
)
```

```
BEGIN
  TABLE
     m/io, adr[]              =>  rom, ram, printer, io_port[];
   %--------------------------------------------------------------%
      1, B"00XXXXXXXXXXXXXX"  =>   1,   0,      0,       0 ;
      1, B"100XXXXXXXXXXXXX"  =>   0,   1,      0,       0 ;
      0, B"0000001010101110"  =>   0,   0,      1,       0 ;
      0, B"0000001011011110"  =>   0,   0,      0,       1 ;
      0, B"0000001101110000"  =>   0,   0,      0,       2 ;
  END TABLE;
END;
```

Wie man die Übergangs- und/oder Ausgangsfunktion einer State Machine mit Hilfe eines Table Statements beschreiben kann, wird in Abschnitt 5.4 exemplarisch demonstriert.

2.7.6 In-Line Funktionsaufrufe

Neben der Möglichkeit, benannte Instanzen von Makrofunktionen und Primitiven in der Variable Section einer AHDL-Beschreibung per Deklaration zu instanziieren, können anonyme Instanzen auch über inline-Funktionsaufrufe in der Logic Section erzeugt werden. Für die Instanziierung einer Makrofunktion muß deren Prototyp wie gewohnt über ein Prototyp Statement bekannt gemacht worden sein. Der Funktionsaufruf kann dann überall dort erfolgen, wo ein Ausdruck stehen darf. Sein Wert ist die zu einer Gruppe zusammengefaßte Menge der bidirektionalen und Ausgangs-Ports, wobei diese innerhalb der Gruppe in der gleichen Reihenfolge angeordnet sind, wie es die gültige Prototypdeklaration angibt.

Das folgende Beispiel zeigt einen inline-Funktionsaufruf in einer Zuweisung:

```
FUNCTION makrofkt (a, b[2..0], c] RETURNS (d[2..1], e[0..2]);
 ...
  f[5..0] = (makrofkt (g[0..1], , B"11") , GND)
 ...
```

Die Anweisung bewirkt zunächst, daß eine anonyme Instanz von `makrofkt` erzeugt wird. Anschließend wird der Gruppe aller Input-Ports dieser Instanz die Gruppe `(g[0..1], , B"11")` zugewiesen, wodurch `a` mit `g0`, `b2` mit `g1`, sowie `b0` und `c` jeweils mit VCC verbunden werden. An `b1` wird bei der Instanziierung kein Wert zugewiesen, sodaß der in der Definition von `makrofkt` für diesen Input vermerkte default-Wert für `b1` gesetzt wird. Die Signale an den Ausgängen der erzeugten Instanz werden schließlich mit `GND` zu einer Gruppe zusammengefaßt und der Gruppe `f[5..0]` zugewiesen, wodurch `f5` mit `d2`, `f4` mit `d1`, `f3` mit `e0`, `f2` mit `e1`, `f1` mit `e2` und `f0` mit `GND` verbunden wird.

Kapitel 3

Elemente

3.1 Reservierte Worte

Im folgenden sind alle reservierten AHDL-Schlüsselwörter aufgeführt. Ob Groß- oder Kleinbuchstaben benutzt werden, ist dabei unerheblich. AHDL ist nicht case-sensitiv. Mit dem Ziel einer höheren Übersichtlichkeit der AHDL-Beschreibungen wird jedoch empfohlen, für alle Schlüsselwörter Großbuchstaben, für alle vom Benutzer vergebenen Namen Kleinbuchstaben zu verwenden.

AND	DEVICE	IS	OPTIONS	THEN
BEGIN	DFF	JKFF	OR	TITLE
BIDIR	DFFE	JKFFE	OTHERS	TRI
BITS	ELSE	LATCH	OUTPUT	VARIABLE
BURIED	ELSIF	LCELL	RETURNS	VCC
CARRY	END	MACHINE	SRFF	WHEN
CASCADE	EXP	MACRO	SRFFE	WITH
CASE	FUNCTION	MCELL	SOFT	X
CLIQUE	GLOBAL	NAND	STATES	XNOR
CONNECTED_PINS	GND	NODE	SUBDESIGN	XOR
CONSTANT	IF	NOR	TABLE	
DEFAULTS	INCLUDE	NOT	TFF	
DESIGN	INPUT	OF	TFFE	

3.2 Namen

Im Zuge der Erstellung einer Harwdarebeschreibung in AHDL vergibt der Benutzer Namen für

- Konstanten,
- die externen Anschlüsse des Designs,
- Makrofunktionen,
- Instanzen,

- State Machines, Zustandsnamen und -bits
- sowie für interne Knoten.

Nochmals:

> **AHDL ist nicht case-sensitiv, d.h. es wird nicht zwischen Groß- und Kleinschreibung unterschieden.**

Neben den von höheren Programmiersprachen bekannten syntaktischen Anforderungen, wonach ein Bezeichner mit einem Buchstaben beginnen muß, auf den eine beliebige Kombination von Buchstaben, Ziffern und Unterstrichen folgen kann, erlaubt AHDL kontextabhängig auch Namen, die mit einem Slash („/”) beginnen (sinnvoll zur Kennzeichnung low-aktiver Signale). In einfache Anführungszeichen gesetzte Namen dürfen darüberhinaus auch mit Ziffern beginnen oder Bindestriche enthalten.

Die Länge eines Namens darf 32 Zeichen nicht übersteigen. Namen von Makrofunktionen und Subdesigns sogar höchstens 8 Zeichen lang sein. (MS-DOS läßt grüßen!)

Um zur funktionalen Simulation den *Synopsys* VHDL Simulator benutzen zu können, wird die AHDL-Beschreibung eines Designs hierarchieerhaltend in eine VHDL-Beschreibung konvertiert. Daraus resultieren eine Reihe zusätzlicher Einschränkungen für die Menge der erlaubten Bezeichner. In VHDL müssen Bezeichner mit einem Buchstaben beginnen und dürfen nur aus Buchstaben, Ziffern und Unterstrichen bestehen. Auch darf ein Name weder mit einem Unterstrich beginnen oder enden, noch zwei aufeinanderfolgende Unterstriche enthalten. Nicht verwendet werden dürfen selbstverständlich außerdem alle VHDL-Schlüsselwörter, ebenso Bezeichner der für die Simulation benötigten Datentypen und Packages. Bei der Konvertierung von AHDL nach VHDL werden darüberhinaus zusätzliche Namen für Knoten und Instanzen erzeugt, die mit „VHDL_” bzw. „N_” beginnen. Auch Namen dieser Form müssen daher für den AHDL-Benutzer ausgeschlossen werden.

Die Syntax erlaubter Bezeichner ist demnach

```
<name> ::=  <letter> { [_] <letter>|<digit> }     .
```

Neben den AHDL Schlüsselwörtern (siehe vorhergehenden Abschnitt) sind folgende Bezeichner nicht zugelassen:

```
alle Bezeichner der Form  VHDL_<name>  und  N_<name>

ABS                       FILE             RANGE
ACCESS                    FOR              RECORD
AFTER                     GENERATE         REGISTER
ALIAS                     GENERIC          REM
ALL                       GUARDED          REPORT
ARCHITECTURE              IN               RETURN
ARRAY                     INOUT            SELECT
ASSERT                    LABEL            SEVERITY
ATTRIBUTE                 LIBRARY          SIGNAL
BLOCK                     LOOP             SIGNED
BODY                      MAP              STD_LOGIC
BUFFER                    MOD              STD_LOGIC_VECTOR
BUS                       NEW              SUBTYPE
COMPONENT                 NEXT             TO
```

CONFIGURATION	NULL	TRANSPORT
CONV_SIGNED	ON	TYPE
CONV_STD_LOGIC_VECTOR	OPEN	UNITS
CONV_UNSIGNED	OUT	UNSIGNED
DISCONNECT	PACKAGE	UNTIL
DOWNTO	PORT	USE
ENTITY	PROCEDURE	WAIT
EXIT	PROCESS	WHILE

3.3 Kommentare

Kommentare werden in AHDL durch ein Prozentzeichen „%" eingeleitet und abgeschlossen und können überall dort in eine Beschreibung eingefügt werden, wo ein Leerzeichen, Tabulator oder Zeilenumbruch erlaubt ist.

Der Text hinter zwei aufeinanderfolgenden Minuszeichen bis zum Zeilenende wird ebenfalls als Kommentar interpretiert.

3.4 Gruppen

In AHDL gibt es zwei Möglichkeiten, mehrere Knoten zu Bussen zusammenzufassen:

1. Sogenannte *Decimal Groups* werden deklariert, indem hinter dem Namen der Gruppe in eckigen Klammern ein Indexbereich angegeben wird, z.B. `a[1..4]`. Es kann nur ein Indexbereich angegeben werden (eindimensionales Array). Die gesamte Gruppe kann nach einer derartigen Deklaration mit `a[1..4]` oder kurz `a[]` angesprochen werden. Teile der Gruppe werden durch Angabe von Unterbereichen spezifiziert, z.B. `a[2..3]` oder `a[3]`. Die eckigen Klammern können bei der Selektion eines einzelnen Gruppenmitglieds auch entfallen (`a3` statt `a[3]`).
2. *Sequential Groups* werden gebildet, indem eine Liste von Knoten, Ports und/oder Konstanten in runde Klammern eingeschlossen wird, also z.B. `(a, b[3..0], c)`. Eine solche Gruppe wird als ein Binärvektor interpretiert, dessen Länge gleich der aufsummierten Längen der einzelnen Gruppenmitglieder ist. *Sequential Groups* können auch zur Zusammenfassung von Ports einer Instanz benutzt werden. Ist z.B. `reg` eine Instanz des Primitivs `DFF`, so können mit der Zuweisung

   ```
   reg.(d, clk, clrn, prn) = (data_in, sys_clk, clear, VCC)
   ```

   allen vier Input-Ports von `reg` Werte zugewiesen werden. Auf der linken Seite von Zuweisungen können in *Sequential Groups* außerdem Kommas als Platzhalter für nicht zuzuweisende Bits verwendet werden. Beispielsweise erhält in der Zuweisung

   ```
   (a, , b) = c[1..3]
   ```

   `a` den Wert von `c[1]` und `b` den Wert von `c[3]` zugewiesen.

Wichtig: Namen von State Machines oder Zustandsnamen dürfen nicht Mitglied in Gruppen sein. In diesem Zusammenhang ist Vorsicht geboten, da der Compiler bei einem Verstoß gegen diese Regel nicht immer eine entsprechende Fehlermeldung ausgibt. Das Synthese-Ergebnis ist dann jedoch fast immer falsch, d.h. es entspricht in keiner Weise dem (vermeindlich) spezifizierten Verhalten.

Zum Abschluß der Erläuterungen zu den Gruppen sei noch einmal darauf hingewiesen, daß die folgenden drei Notationen ein und dieselbe Gruppe bezeichnen:

```
b[5..0]
(b[5], b[4..2], b1, b[0])
(b5, b4, b3, b2, b1, b0);
```

3.5 Zahlen

Ganze Zahlen können in AHDL als Dezimal-, Binär-, Oktal- oder Hexadezimalzahlen angegeben werden. Wird keine andere Basis angegeben, so interpretiert der Compiler eine Ziffernfolge als Dezimalzahl. Zahlen zu anderen Basen werden durch voranstellen eines Basisinidikators und Einschließen des Werts in doppelte Anführungszeichen spezifiziert. Binärzahlen erhalten als Basisindikator den Buchstaben `B` bzw. `b`, Oktalzahlen `Q`, `q`, `O` oder `o` und Hexadezimalzahlen `H`, `h`, `X` oder `x` vorangestellt. Gültige Zahlen sind beispielsweise

```
143
B"01001"
q"715"
H"A2d"          .
```

Außer bei der Angabe von Array-Indices, wo nur Dezimalzahlen zugelassen sind, können überall Zahlen zu einer beliebigen Basis benutzt werden. Bei der Logiksynthese werden sie vom Compiler in vorzeichenbehaftete Binärzahlen der im jeweiligen Kontext benötigten Wortbreite umgewandelt.

3.6 Operatoren und Ausdrücke

Ausdrücke in AHDL bestehen aus einem oder mehreren Operanden, die durch logische oder arithmetische Operatoren voneinander getrennt sind. Sie können in einer AHDL-Beschreibung überall dort stehen, wo Binärsignale bzw. Binärworte erwartet werden, also auf der rechten Seite von Zuweisungen, als Bedingung in If-Statements und als Selektor in Case-Statements.

Operanden können sein:

- Konstanten,
- Zahlen,
- Knoten,
- Subdesign-Ports,
- Instanzen-Ports,
- inline-Funktionsaufrufe,
- Gruppen,
- Ausdrücke,
- geklammerte Ausdrücke.

Tabelle 3.1 enthält eine Zusammenstellung aller verfügbaren Operatoren.

In Tabelle 3.2 sind die Operatoren nach absteigender Priorität geordnet aufgeführt. Ein Operator niedrigerer Priorität bindet stärker als ein Operator höherer Priorität. Es ist jedoch ein guter Design-Stil und dient in hohem

Operator	**Beispiel**	**Funktion**
`!` `NOT`	`!a` `NOT a[3..0]`	logische Invertierung
`&` `AND`	`tom & jerry` `a[] AND O"15"`	logisches UND
`!&` `NAND`	`a[3..1] !& b[5..3]` `a NAND b`	logisches NICHT-UND
`#` `OR`	`tee # kaffee` `a[] OR b[]`	logisches ODER
`!#` `NOR`	`a !# b` `(a, b) NOR c[1..0]`	logisches NICHT-ODER
`$` `XOR`	`tot $ lebendig` `c[] XOR B"'1011"`	logisches exklusiv-ODER
`!$` `XNOR`	`x !$ y` `(a, b, c) !$ d[]`	logische Äquivalenz
+ (unär)	`+1`	positives Vorzeichen
- (unär)	`-a[]`	negatives Vorzeichen (2er-Komplement)
+	`counter[7..0] + 14`	Addition
-	`a[] - (b[3..0], c, GND)`	Subtraktion
`==`	`adr[15..0] == H"C800"`	Test auf Gleichheit
`!=`	`a != b`	Test auf Ungleichheit
`<`	`sieben[] < acht[]`	Test auf „kleiner"
`<=`	`wert[] <= 17`	Test auf „kleiner oder gleich"
`>`	`wert[] > B"101101"`	Test auf „größer"
`>=`	`delta[] >= 0`	Test auf „größer oder gleich"

Tabelle 3.1: AHDL Operatoren

Priorität	**Operatoren**
1	`!` (`NOT`)
1	`+` (unär)
1	`-` (unär)
2	`+` (Addition)
2	`-` (Subtraktion)
3	`==`
3	`!=`
3	`<`
3	`<=`
3	`>`
3	`>=`
4	`&` (`AND`)
4	`!&` (`NAND`)
5	`$` (`XOR`)
5	`!$` (`XNOR`)
6	`#` (`OR`)
6	`!#` (`NOR`)

Tabelle 3.2: Prioritäten der AHDL Operatoren

Maße der Übersichtlichkeit einer AHDL-Beschreibung, wenn in einem Ausdruck die Auswertungsreihenfolge durch Klammersetzung hervorgehoben wird, selbst wenn Klammern angesichts der Prioritäten der Operatoren nicht erforderlich sind.

In den folgenden Abschnitten werden noch einige wichtige Eigenschaften der Operatoren kurz erläutert.

3.6.1 NOT

Der `NOT`-Operator kann in drei unterschiedlichen Typen verwendet werden:

- Ist der Operand ein einfaches Binärsignal, so wird es invertiert.
- Ist der Operand eine Gruppe von Binärsignalen, so wird jedes Mitglied der Gruppe invertiert.
- Ist der Operand eine Zahl oder Konstante, so wird jedes Bit der binären Repräsentation der Zahl invertiert, also das Einskomplement gebildet, wobei die Wortbreite dem Kontext angepaßt wird. Beispiel: `!9 = !B"01001" = B"10110"`.

3.6.2 AND, NAND, OR, NOR, XOR, XNOR

Für jeden der sechs Booleschen Operatoren existieren fünf Operanden-Kombinationen, in denen die Operatoren unterschiedlich interpretiert werden:

- Sind beide Operanden einfache Binärsignale, so bewirkt der Operator die Boolesche Verknüpfung der beiden Signale.
- Sind beide Operanden Gruppen von Binärsignalen, so müssen ihre Wortbreiten übereinstimmen. Die Boolesche Verknüpfung wird dann komponentenweise durchgeführt.
- Ist einer der Operanden eine Gruppe und der andere ein einfaches Binärsignal, so wird jedes Bit der Gruppe mit dem einzelnen Binärsignal verknüpft.
  Beispiel: `a & b[1..3]` wird interpretiert als `(a & b1, a & b2, a & b3)`.
- Sind beide Operanden Zahlen, so werden beide Zahlen als Binärzahlen interpretiert, wobei die Zahl mit der kürzeren Binärdarstellung vorzeichenrichtig an die Wortbreite der längeren Zahl angepaßt wird. Beispiel: `(3 # 8) = (B"0011" # B"1000") = B"1011"`.
- Ist einer der Operanden eine Zahl und der andere ein einfaches Binärsignal oder eine Gruppe, so wird die Binärdarstellung der Zahl an die Wortbreite des anderen Operanden (vorzeichenrichtig) angepaßt. Ist hierzu eine Verkürzung der Binärzahl erforderlich, so gibt der Compiler eine Warnung aus. Die Wirkung des Operators entspricht dann der, wie sie bei der Verknüpfung einer Gruppe mit einem Binärsignal bzw. einer Gruppe erläutert wurde. Beispiel: `(a, b, c) & 1 = (a, b, c) & B"001" = (0, 0, c)`. Man beachte insbesondere, daß sich ein Ausdruck, der den Operanden `VCC` verwendet, von einem mit dem Operanden `1` unterscheidet. Es gilt nämlich z.B. `(a, b, c) & VCC = (a & VCC, b & VCC, c & VCC) = (a, b, c)`.

3.6.3 Arithmetische Operatoren

Für arithmetische Operationen stehen die Operatoren „`+`” und „`--`” zur Verfügung. Sie sind nur für Gruppen und Zahlen bzw. Konstanten definiert. Die Operationen werden in binärer Zahldarstellung mit 2er-Komplement-Darstellung negativer Zahlen durchgeführt, wobei Zahlen und Konstanten in ihrer Wortbreite an den Kontext angepaßt werden. Bei der Addition bzw. Subtraktion von Gruppen müssen diese die gleiche Wortbreite aufweisen.

3.6.4 Komparatoren

Mit Hilfe der Komparatoren „`==`“, „`!=`“, „<“, „<=“, „>“ und „>=“ können einzelne Binärsignale bzw. Gruppen miteinander verglichen werden. Als Ergebnis des Vergleichs erhält man einen Binärwert (`VCC`, falls der Vergleich `true` liefert, `GND`, falls er `false` liefert und `X` (für undefiniert), falls einer der Operanden undefinierte Werte enthält).

Für den Vergleich mit einzelnen Binärsignalen stehen nur der Test auf Gleichheit bzw. Ungleichheit zur Verfügung. Beim Vergleich von Gruppen werden deren Binärworte als Darstellung **vorzeichenloser** (positiver) Binärzahlen interpretiert.

3.7 Primitive

Primitive sind definierte Teilschaltungen, die AHDL dem Benutzer zur Verfügung stellt. Sie bilden die Basiselemente, aus denen jede Schaltung komponiert wird und sind also beispielsweise vergleichbar mit den Standardzellen beim Entwurf eines Standardzellen-ICs.

AHDL stellt, neben VCC und GND, den Booleschen Verknüpfungen AND, OR, XOR, NAND, NOR, XNOR und NOT sowie den Pin-Primitiven INPUT, OUTPUT und BIDIR, insgesamt 16 verschiedene Primitive zur Verfügung, die in der gleichen Weise benutzt werden können, wie vom Benutzer definierte Subdesigns oder Makrofunktionen aus Bibliotheken, also sowohl als Objekt-Typ in einer Variablendeklaration als auch in Form eines inline-Funktionsaufrufs in der Logic Section eines Design-Files.

Im folgenden werden die Primitive und ihre (implizite) Prototypen-Deklaration aufgelistet und die jeweils realisierte Funktion kurz erläutert. Einige der Eingangssignale sind mit default-Werten versehen (z.B. die asynchronen *Preset-* und *Clear-* Signale *prn* und *clrn* der Flipflops). Diese Eingänge können bei der Verwendung des Primitivs offen gelassen werden.

CARRY (in) RETURNS (out)

Carry Buffer; weist die Logiksynthese an, für die Bildung des Signals `out`, die spezielle *carry-out-* Logik in den Devices der FLEX8000-Familie zu verwenden. Bei inkorrektem Gebrauch des Carry-Buffers wird dieser ignoriert.

CASCADE (in) RETURNS (out)

Cascade Buffer; weist die Logiksynthese an, für die Bildung des Signals `out`, die spezielle *cascade-out-* Funktion in den Devices der FLEX8000-Familie zu verwenden. Bei inkorrektem Gebrauch des Cascade-Buffers wird dieser ignoriert.

DFF (d, clk, clrn := VCC, prn := VCC) RETURNS (q)

Vorderflankengesteuertes D-Flipflop mit asynchronem *Preset* (`prn`) und asynchronem *Clear* (`clrn`) (beide low-aktiv und per default inaktiv). `q` ist der im Flipflop gespeicherte Wert.

DFFE (d, clk, clrn := VCC, prn := VCC, ena := VCC) RETURNS (q)

Wie DFF, jedoch mit zusätzlichem high-aktivem *Enable* (`ena`).

EXP (in) RETURNS (out)

Expander Buffer; weist den Synthese-Algorithmus an, den logischen Wert von `in` über einen *Shared Expander* zu führen, wobei `in` invertiert wird. Der Expander Buffer sollte nur von erfahrenen Entwerfern und auch nur in Ausnahmefällen benutzt werden.

GLOBAL (in) RETURNS (out)

Global Buffer; weist den Synthese-Algorithmus an, das Eingangssignal `in` auf eine globale Leitung (*Clock*, *Clear*, *Preset* oder *Output Enable*) zu legen. `in` muß dabei ein Input-Pin des Designs sein, wobei für *Output Enable* eine Invertierung des Pin-Signals zugelassen ist. Nicht alle Device-Familien verfügen über alle vier globalen Leitungen. Bei Fehlen des entsprechenden Netzes wird der Global Buffer bei der Compilation einfach ignoriert.

JKFF (j, k, clk, clrn := VCC, prn := VCC) RETURNS (q)

Vorderflankengetaktetes JK-Flipflop mit asynchronem *Preset* (`prn`) und asynchronem *Clear* (`clrn`) (beide low-aktiv und per default inaktiv). Die Übergangsfunktion ist die eines gewöhnlichen JK-Flipflops:

J	K	Q_{neu}
0	0	Q_{alt}
0	1	1
1	0	0
1	1	$\neg Q_{\mathrm{alt}}$

`q` ist der im Flipflop gespeicherte Wert.

JKFFE (j, k, clk, clrn := VCC, prn := VCC, ena := VCC) RETURNS (q)

Wie JKFF, jedoch mit zusätzlichem high-aktivem *Enable* (`ena`).

LATCH (d, ena := VCC) RETURNS (q)

Pegelgesteuertes Flipflop; solange `ena = VCC` ist, wird `d` nach `q` durchgeschaltet, während `ena = GND` bleibt der zuletzt an q zugewiesene Wert unverändert.

LCELL (in) RETURNS (out)

LCELL Buffer; weist den Synthese-Algorithmus an, für die Bildung des Ausgangssignals `out` in jedem Fall eine eigene Logikzelle vorzusehen, auch wenn `in` primärer Input oder seinerseits Ausgang einer Logikzelle ist. LCELL Buffer können benutzt werden, um das Mapping der Logik auf die Logikzellen manuell zu beeinflussen und (hoffentlich) effizientere Realisierungen zu erreichen. In der Regel ist der SOFT Buffer dem LCELL Buffer vorzuziehen, da hier eine Extra-Logikzelle nur alloziert wird, wenn dieses auch sinnvoll ist.

MCELL (in) RETURNS (out)

MCELL Buffer; alte Bezeichnung für LCELL-Buffer. Der MCELL-Buffer wird zwecks backward-Kompatibilität weiterhin unterstützt, sollte in neuen Designs jedoch nicht mehr benutzt werden.

SOFT (in) RETURNS (out)

SOFT Buffer; wirkt ähnlich dem LCELL Buffer, mit der Ausnahme, daß nicht in jedem Fall eine zusätzliche Logikzelle alloziert wird. Beispielsweise wird ein SOFT-Buffer ignoriert, wenn das Signal `in` bereits das Ausgangssignal einer Logikzelle oder einen Input-Pin bezeichnet. Der SOFT Buffer ist daher in der Regel dem LCELL Buffer vorzuziehen, wenn es darum geht, das Mapping der Logik auf die Logikzellen manuell zu beeinflussen.

SRFF (r, s, clk, prn := VCC, clrn := VCC) RETURNS (q)

Vorderflankengetaktetes RS-Flipflop mit asynchronem *Preset* (`prn`) und asynchronem *Clear* (`clrn`) (beide low-aktiv und per default inaktiv). Das Flipflop wird bei `r = VCC` synchron zurückgesetzt, bei `s = VCC` synchron gesetzt. `q` ist der im Flipflop gespeicherte Wert.

SRFFE (r, s, clk, clrn := VCC, prn := VCC, ena := VCC) RETURNS (q)

Wie SRFF, jedoch mit zusätzlichem high-aktivem *Enable* (`ena`).

TFF (t, clk, clrn := VCC, prn := VCC) RETURNS (q)

Vorderflankengesteuertes T-Flipflop mit asynchronem *Preset* (`prn`) und asynchronem *Clear* (`clrn`) (beide low-aktiv und per default inaktiv). Das Flipflop toggelt bei einer Vorderflanke an `clk`, falls `t = VCC`. `q` ist der im Flipflop gespeicherte Wert.

TFFE (t, clk, clrn := VCC, prn := VCC, ena := VCC) RETURNS (q)

Wie TFF, jedoch mit zusätzlichem high-aktivem *Enable* (`ena`).

TRI (in, oe := VCC) RETURNS (out)

Tristate Buffer mit Dateneingang, *Output Enable* und Datenausgang. Bei `oe = GND` ist `out` hochohmig, ansonsten ist `out = in`. Ist `oe` dauerhaft mit `VCC` verbunden, also der `TRI`-Buffer stets aktiviert, so wird der `TRI`-Buffer während der Logiksynthese durch einen SOFT-Buffer ersetzt. Bei der Verwendung von Tristate-Buffern ist zu beachten, daß sie immer auf einen bidirektionalen Pin geführt werden müssen. In den Altera Devices können keine internen tristate-Signale benutzt werden.

3.8 Makrofunktionen

Für die hierarchische Gliederung eines Design Projekts bietet AHDL die Möglichkeit, neben den Primitiven auch Makrofunktionen in einem Design-File zu instanziieren. Makrofunktionen sind Subdesigns, die in separaten Design-Files beschrieben wurden.

In einem Prototypen-Statement sind hierzu zunächst der Name der Makrofunktion und die Namen der Ports zu deklararieren. Die Makrofunktion kann dann entweder über eine Variablendeklaration oder durch einen inline-Funktionsaufruf instanziiert werden.

Hierzu ein Beispiel: In einem Design-File *binadd4.tdf* wurde das Subdesign `binadd4` zur Additionen zweier 4-Bit Binärworte definiert und soll nun in dem Design `binadd8` instanziiert werden, um zwei 8-Bit Binärworte zu addieren. In dem Design-File *binadd8.tdf* werden daher zuächst der Prototyp der Makrofunktion deklariert und anschließend zwei Instanzen von `binadd4` erzeugt:

```
FUNCTION binadd4 (a[3..0], b[3..0], carry_in) RETURNS (s[3..0], carry_out);

SUBDESIGN binadd8 (
  arg1[7..0], arg2[7..0]: INPUT;
  sum[7..0], carry_out: OUTPUT;
)

VARIABLE
  low_nibble_adder: binadd4;
  ripple_carry: NODE;
BEGIN
  low_nibble_adder.(a[], b[], carry_in) = (arg1[3..0], arg2[3..0], GND);
  (sum[3..0], ripple_carry) = low_nibble_adder.(s[], carry_out);
  (sum[7..4], carry_out) = binadd4(arg1[7..4], arg2[7..4], ripple_carry);
END;
```

Die erste Instanz wird in der Variable Section deklariert und erhält den Namen `low_nibble_adder`. Sie berechnet die niederwertigen vier Bits der Summe. `ripple_carry` nimmt den dabei möglicherweise entstehenden Übertrag auf, um ihn an den Addierer für die oberen vier Bits weiterzuleiten. Dessen Instanziierung erfolgt in der Logic Section über einen inline-Funktionsaufruf. Diese Instanz von `binadd4` erhält dadurch keinen in der AHDL-Beschreibung verfügbaren Namen. Der Compiler erzeugt für diese Instanz einen künstlichen Namen, in dem auf den Makrofunktionsnamen ein Doppelpunkt und eine eindeutige Instanznummer folgen.

Makrofunktionen ermöglichen es, Teilschaltungen, die in unterschiedlichen Design-Projekten immer wieder benötigt werden, in Bibliotheken bereit zu stellen. Die zum Lieferumfang des Entwurfssystem MAX+plusII gehörende Standard-Bibliothek enthält über 300 Makrofunktionen, überwiegend funktional identisch mit Funktionen aus der weitverbreiteten 74er TTL- und CMOS-Bausteinreihe. Auch der Benutzer kann von ihm erstellte Subdesigns als Makrofunktionen in eigenen Bibliotheken zusammenfassen und für seine Design-Projekte verfügbar halten. Die Prototypen von Bibliotheksfunktionen sollten, wie es bei der MAX+plusII-Standardbibliothek der Fall ist, in Include-Files `<funktionsname>`*.inc* bereitgestellt werden.

Kapitel 4

Gestaltung von AHDL-Beschreibungen

In diesem Kapitel sollen einige Hinweise für die Gestaltung von AHDL-Beschreibungen gegeben werden. Ihre durchgängige Beachtung erhöht in hohem Maße Übersichtlichkeit und intuitive Verständlichkeit des AHDL-Codes.

4.1 Formatierungshinweise

Die Lesbarkeit von AHDL-Code wird deutlich verbessert, wenn die Beschreibung in folgender Weise formatiert wird:

- Geben Sie alle Schlüsselwörter, Device-Namen und Konstanten in Großbuchstaben, den übrige Text einschließlich Filenamen und Makrofunktionen in Kleinbuchstaben ein.
- Benuzten Sie innerhalb der einzelnen Teile der Beschreibung Einrückungen, um zusammengehörige Teile optisch hervorzuheben. Achten Sie insbesondere darauf, daß Sie **immer** und **überall** in derselben Weise einrücken.
- Fügen Sie keine Leerzeichen vor Kommas, Semikolons, Anführungszeichen oder schließenden Klammern ein. Verzichten Sie ebenso auf Leerzeichen hinter öffnenden Klammern oder Anführungszeichen. Beispiel:

  `DFF(d, clk, clrn, prn);` statt `DFF ( d , clk , clrn , prn ) ;`

- Fügen Sie vor und hinter dyadischen Operatoren ein Leerzeichen ein, also

  `c = !a3 & a2 & a1 & a0;` statt `c = !a3&a2&a1&a0;`

- Setzen Sie bei Verwendung verschiedener Operatoren in ein und demselben Ausdruck Klammern, um die Auswertungsreihenfolge hervorzuheben und zwar auch dann, wenn aufgrund der Prioritäten der Operatoren Klammern nicht erforderlich sind, also

  `c = a3 # (a2 & a1);` statt `c = a3 # a2 & a1;`

- Trennen Sie ein Schlüsselwort nach vorn und nach hinten durch ein Leerzeichen oder einen Zeilenumbruch vom umgebenden Text, also

  `OF BITS (q[3..0])` statt `OF BITS(q[3..0])`

4.2 Kommentierung

Kommentare sind in einer AHDL-Beschreibung überall dort erlaubt, wo auch ein Leerzeichen stehen kann. Zweckmäßigerweise sollte man aber Kommentare in der Regel nur am Zeilenende oder in separaten Zeilen in den Text einfügen. Dabei sollten Sie insbesondere darauf achten, daß ein Kommentar nicht lediglich das kommentierte Statement dupliziert, wie es im folgenden schlechten Beispiel geschieht:

```
IF clear THEN
  % load q[] with 0 %
  q[] = 0;
END IF;
```

Ist damit zu rechnen, daß ein Design-File nach längerer Zeit oder von anderen Personen benutzt oder bearbeitet werden muß, sollten in einem längeren natürlichsprachlichen Kommentar am Anfang jedes Design-Files die Funktionalität des in ihm beschriebenen Subdesigns kurz, aber vollständig beschrieben werden.

4.3 Namen

Die meisten „Fehler" hinsichtlich der Lesbarkeit und Verständlichkeit einer AHDL-Beschreibungen können bei der Namensvergabe gemacht werden.

Ein Name sollte sinnvollerweise so gewählt werden, daß er den Zweck oder die Aktion, für die er benutzt wird, unmißverständlich widerspiegelt. Beispielsweise ist dem Knotennamen `direction` der Name `up` vorzuziehen, weil dadurch nicht nur ausgesagt wird, daß eine Richtung angegeben wird, sondern auch welche Richtung im Falle eines High-Pegels an `up` selektiert ist. Ein ähnliches Beispiel ist etwa das Namenspaar `access_mode` und `access_memory`.

Desweiteren sollten Namen nur abgekürzt werden, wenn dadurch die bezeichnete Funktion nicht unkenntlich gemacht wird. Beispielsweise ist die Abkürzung `clk` für `clock` naheliegend, während `rdrm` den ungekürzten und sprechenden Namen `refresh_dram` kaum noch erkennen läßt.

Sehr hilfreich für die Verständlichkeit ist es auch, wenn man durchgängig eine einheitliche Notation zur Kennzeichnung von low-aktiven Signalen verwendet, also beispielsweise

`/write`[1]
`n_write`[1]
`nWrite`
`write_n`.

[1]Nicht erlaubt, falls VHDL zur Simulation benutzt wird

Kapitel 5

Beispiel-Designs

In diesem Kapitel wird die Verwendung der in den beiden vorhergehenden Kapiteln im Detail erläuterten AHDL-Konstrukte anhand einiger Beispiel-Designs demonstriert.

Ein Design kann in AHDL auf ganz unterschiedliche Weise semantisch äquivalent beschrieben werden. Letztlich wird jede AHDL-Beschreibung zunächst in eine Menge von logischen Verknüpfungen übersetzt, deren Ergebnisse entweder die Eingänge von Flipflops ansteuern oder über einen primären Ausgang des Designs extern verfügbar gemacht werden. Beispielsweise sind die beiden AHDL-Anweisungen

```
c = a & select_a # b & !select_a;
```

```
IF select_a THEN
  c = a;
ELSE
  c = b;
END IF;
```

völlig gleichwertig. Die linke Form der Beschreibung hebt den logischen Zusammenhang zwischen den beteiligten Signalen hervor, während die Verwendung des If Statements eher den Multiplexer- Charakter betont, indem das Steuersignal `select_a` auch in der Beschreibung eine (den „Programmfluß“) steuernde Funktion erhält.

Die Wahl einer bestimmten Form der Beschreibung dokumentiert damit stets auch die Sicht, die der Entwerfer von seinem Design hat. Die im folgenden vorgestellten Beispiel-Designs können deshalb auch gänzlich anders beschrieben werden. Die hier gewählte Form der Beschreibung betont funktionale Zusammenhänge auf der RT-Ebene, woraus sehr kompakter AHDL-Code resultiert.

5.1 8 Bit Schieberegister

Der folgende AHDL-Code beschreibt ein parallel und seriell ladbares 8-Bit Schieberegister mit low-aktivem Reset.

```
SUBDESIGN shifter (
  resetn,                   % low-aktives asynchrones reset-Signal      %
  clk,                      % Taktung des Registers                     %
  shift_en,                 % Signal zur Aktivierung des Schiebe-Modus' %
  load_en,                  % Signal zur Aktivierung des Lade-Modus'    %
  serial_in,                % Serieller Dateneingang                    %
  parallel_in[7..0]: INPUT; % Paralleler Dateineingang                  %

  data_out: OUTPUT;         % Serieller Datenausgang                    %
)

VARIABLE
  reg[7..0]: DFFE;    % Instanziiert Gruppe von 8 D-Flipflops mit Enable %

BEGIN

  reg[].clrn = resetn;        % Asynchrones Zuruecksetzen des Registers, %
                              % falls resetn = GND (low-aktiv)           %

  reg[].clk = clk;                % Taktung des Registers mit Port 'clk' %

  reg[].ena = shift_en # load_en;     % Register ist nur enabled, falls  %
                                      % shift_en oder load_en = VCC gilt %

  IF shift THEN
    reg[] = (reg[6..0], serial_in);       % Falls Schieben enabled ist,  %
                                          % reg[7].d = reg[6].q ,        %
                                          % reg[6].d = reg[5].q ,        %
                                          % ...                          %
                                          % reg[1].d = reg[0].q ,        %
                                          % reg[0].d = serial_in         %
  ELSE
    reg[] = parallel_in[];                % Falls Schieben disabled ist, %
                                          % reg[7].d = parallel_in[7] ,  %
                                          % ...                          %
                                          % reg[0].d = parallel_in[0]    %
  END IF;

  data_out = reg7;        % Ausgabe des h"ochstwertigsten Bits reg[7].q  %

END;
```

5.2 Modulo-60 Zähler

In dem folgenden Subdesign wird ein Modulo-60 Zähler mit Enable und synchronem high-aktivem Reset beschrieben. Man beachte insbesondere die unmittelbare Umsetzung der Funktionalität in AHDL-Code durch Verwendung der Operatoren „==" und „+". Die zur Realisierung dieser Operatoren benötigten logischen Funktionen werden vom Compiler automatisch generiert.

```
SUBDESIGN mod60cnt (
  reset:  INPUT;        % synchrones, high-aktives reset-Signal %
  clk:    INPUT;        % Z"ahltakt                             %
  enable: INPUT;        % high-aktives enable-Signal            %

  value[5..0]: OUTPUT;  % Z"ahlerstand %
)

VARIABLE
  counter[5..0]: DFFE;    % Instanziiert Gruppe von 6 D-Flipflops %

BEGIN

  counter[].clk = clk;    % Taktung des Z"ahlers; Wertewechsel erfolgen  %
                          % mit der Vorderflanke des Subdesign-Ports clk %

  counter[].ena = enable;         % Takt nur wirksam, falls enable = VCC %

  IF reset # (counter[] == 59) THEN  % Falls reset = VCC oder momentaner %
    counter[] = 0;                   % Wert ist 59, bei n"achster Takt-  %
                                     % vorderflanke Z"ahler auf 0 setzen %
  ELSE
    counter[] = counter[] +1;        % sonst Z"ahler um 1 erh"ohen.      %

  END IF;

  value[] = counter[];        % Z"ahlerstand ausgeben "uber Port value[] %

END;
```

5.3 Einfache Uhr

In diesem Abschnitt wird anhand des Beispiels einer einfachen Uhr die Realisierung hierarschischer Designs in AHDL demonstriert. Das top-level Subdesign trägt den Namen `clock` und verfügt über einen synchronen high-aktiven Eingang `reset`, um die Uhrzeit auf Null zu setzen, sowie einen Takteingang `clk_1Hz`, an den ein 1Hz Taktsignal zur sekundenweisen Fortschaltung der Uhrzeit angeschlossen wird. Die Zeit wird in Stunden (Output `hours`), Minuten (Output `minutes`) und Sekunden (Output `seconds`) ausgegeben. Zum Mitzählen von Sekunden und Minuten wird innerhalb des Subdesigns `clock` die Makrofunktion `mod60cnt` aus Abschnitt 5.2 zweimal instanziiert. Für das Zählen der Stunden wird ein Modulo-24 Zähler instanziiert, dessen AHDL-Beschreibung mit Ausnahme der Wortbreite des Zählers und des Zahlenwert für den Vergleich mit der von `mod60cnt` übereinstimmt:

```
SUBDESIGN mod24cnt (
  reset, clk, enable: INPUT;
  value[4..0]: OUTPUT;
)

VARIABLE
  counter[4..0]: DFFE;  % Wortbreite des Z"ahlers 5 Bits %

BEGIN
  counter[].clk = clk;
  counter[].ena = enable;
  IF reset # (counter[] == 23) THEN
    counter[] = 0;
  ELSE
    counter[] = counter[] +1;
  END IF;
  value[] = counter[];
END;
```

Um die beiden alternativen Möglichkeiten zur Instanziierung von Makrofunktionen zu demonstrieren, werden die beiden Modulo-60 Zähler `sec_counter` und `min_counter` per Deklaration in der Variable Section des Subdesigns `clock` instanziert, während der Modulo-24 Zähler über einen inline-Funktionsaufruf in der Logic Section instanziiert wird. Man beachte darüberhinaus die unterschiedliche Vorgehensweise bei der Herstellung von Verbindungen von und zu den Ports der instanziierten Makrofunktionen.

```
FUNCTION mod60cnt (reset, clk, enable) RETURNS (value[5..0]);
FUNCTION mod24cnt (reset, clk, enable) RETURNS (value[4..0]);

SUBDESIGN clock (
  reset, clk_1Hz: INPUT;
  hours[4..0], minutes[5..0], seconds[5..0]: OUTPUT;
)

VARIABLE
  sec_counter, min_counter: mod60cnt;
  last_second, last_minute: NODE;

BEGIN

  sec_counter.clk = clk_1Hz;
  sec_counter.enable = VCC;             % Sekundenz"ahler immer aktiviert %
  sec_counter.reset = reset;
  seconds[] = sec_counter.value[];
  last_second = (seconds[] == 59);      % Falls 59. Sekunde, bei          %
                                        % n"achstem Takt Minuten erh"ohen %

  min_counter.(clk, enable, reset) = (clk_1Hz, last_second, reset);
  minutes[] = min_counter.value[];
  last_minute = (minutes[] == 59);

  hours[] = mod24cnt(reset, clk_1Hz, last_second & last_minute);

END;
```

5.4 Sequence Analyzer

Zur Demonstration der Möglichkeiten, die AHDL zur Beschreibung von endlichen Automaten (State Machines) bietet, wird in diesem Abschnitt das Design-File eines einfachen Bitsequenz-Analysators vorgestellt. Dieser Analysator hat die Aufgabe, in einer unendlichen Sequenz von Bits jedes Vorkommen der Bitfolge „10110" durch Aktivieren einer Ausgangsleitung `recognized` anzuzeigen. Hierbei sollen auch sich überlappende Bitfolgen erkannt werden. Die zu untersuchende Sequenz wird dem Automaten bitweise zugeführt, wobei das jeweils nächste Bit an den Eingang `d` angelegt wird und anschließend mit einer Vorderflanke am Takteingang `clk` ein Zustandsübergang in dem Analysator ausgelöst wird. Über einen asynchronen high-aktiven Eingang `reset` kann der Automat darüberhinaus jederzeit für die Untersuchungen einer neuen Sequenz initialisiert werden.

```
SUBDESIGN analyzer (
  reset, clk, d:  INPUT;
  recognized:     OUTPUT;
)

VARIABLE
  fsm: MACHINE WITH STATES ( init,
                             read1,
                             read10,
                             read101,
                             read1011,
                             read10110 );
BEGIN

  fsm.reset = reset;
  fsm.clk = clk;
  CASE fsm IS
    WHEN init       =>  IF d THEN fsm = read1;
                        END IF;

    WHEN read1      =>  IF !d THEN
                          fsm = read10;
                        ELSE
                          fsm = init;
                        END IF;

    WHEN read10     =>  IF d THEN
                          fsm = read101;
                        ELSE
                          fsm = init;
                        END IF;

    WHEN read101    =>  IF d THEN
                          fsm = read1011;
                        ELSE
                          fsm = read10;
                        END IF;

    WHEN read1011   =>  IF !d THEN
                          fsm = read10110;
                        ELSE
                          fsm = read1;
```

```
                          END IF;

          WHEN read10110  =>  IF d THEN
                                fsm = read101;
                              ELSE
                                fsm = init;
                              END IF;
        END CASE;

        recognized = (fsm == read10110);   % Ausgangsfunktion %

      END;
```

In dieser ersten AHDL-Beschreibung des Analysators wird die Übergangsfunktion des Automaten mit Hilfe eines Case Statements [1] spezifiziert, während die Ausgangsfunktion in diesem Beispiel so einfach ist, daß sie mit Hilfe einer einzigen Zuweisung angegeben werden kann. Da es sich bei dieser Realisierung insgesamt um einen Moore-Automaten handelt, hätte die zu jedem Zustand gehörende Belegung des Ausgangswerts auch in das Case Statement integriert werden können. In allen Alternativen bis auf `WHEN read10110` wäre hierzu `recognized = GND;` als zusätzliche Anweisung aufzunehmen, für den Zustand `read10110` entsprechend `recognized = VCC`.[2] Eine andere Möglichkeit, Übergangs- und/oder Ausgangsfunktion des Automaten zu spezifizieren, stellt die Verwendung einer Wertetabelle (Table Statement) dar, wie sie in der folgenden AHDL-Beschreibung gewählt wurde:

```
      ...
      BEGIN

        TABLE
            fsm,     d  =>  fsm,    recognized;
         %--------------------------------------%
          init,      0  =>  init,       0 ;
          init,      1  =>  read1,      0 ;
          read1,     0  =>  read10,     0 ;
          read1,     1  =>  init,       0 ;
          read10,    0  =>  init,       0 ;
          read10,    1  =>  read101,    0 ;
          read101,   0  =>  read10,     0 ;
          read101,   1  =>  read1011,   0 ;
          read1011,  0  =>  read10110,  0 ;
          read1011,  1  =>  read1,      0 ;
          read10110, 0  =>  init,       1 ;
          read10110, 1  =>  read101,    1 ;

        END TABLE;

      END;
```

Bei der Auswahl einer bestimmten Form der Beschreibung spielen persönliche Vorlieben eine wesentliche Rolle. Man sollte jedoch auf eine Lokalität der Beschreibungen von Übergangs- und Ausgangsfunktion achten, also beispielsweise nicht Teile des Übergangsverhaltens in einem Case Statement und andere Teile davon getrennt mit anderen Anweisungen spezifizieren.

[1] Zur Verwendung des Case Statements zur Beschreibung des Übergangs- und Ausgabeverhaltens einer State Machine siehe auch den Abschnitt 2.7.2

[2] Die Zuweisungen `recognized = GND` sind aufgrund des default-Werts `GND` für alle Knoten eines Designs redundant.

Die beiden bisher vorgestellten Beschreibungen waren Moore-Automaten. Nach Auffassung des Autors sind Moore-Automaten in ihrem Verhalten überschaubarer und damit leichter zu handhaben als Mealy-Automaten. Dennoch kann eine Mealy-Realisierung sinnvoll sein, weil sie entweder mit einem kleineren Realisierungsaufwand verbunden ist oder aber der unmittelbare Einfluß von Eingangsbelegungen auf die Ausgabe des Automaten erwünscht oder gar erforderlich ist. Im folgenden wird der Sequenz-Analysator daher alternativ als Mealy-Automat beschrieben, wobei wiederum der Formulierung mit Hilfe des Case Statements die Verwendung des Table Statements gegenüber gestellt wird.

```
...

BEGIN

  fsm.reset = reset;
  fsm.clk = clk;
  CASE fsm IS
    WHEN init       =>  IF d THEN fsm = read1;
                        END IF;

    WHEN read1      =>  IF !d THEN
                          fsm = read10;
                        ELSE
                          fsm = init;
                        END IF;

    WHEN read10     =>  IF d THEN
                          fsm = read101;
                        ELSE
                          fsm = init;
                        END IF;

    WHEN read101    =>  IF d THEN
                          fsm = read1011;
                        ELSE
                          fsm = read10;
                        END IF;

    WHEN read1011   =>  IF !d THEN
                          fsm = read10;
                        ELSE
                          fsm = read1;
                        END IF;
  END CASE;

  recognized = (fsm == read1011) & !d;

END;
```

Eine Mealy-Realisierung mit Hilfe des Table Statements könnte dann so aussehen:

```
...

BEGIN

  TABLE
      fsm,      d  =>  fsm,    recognized;
   %-------------------------------------%
    init,       0  =>  init,       0 ;
    init,       1  =>  read1,      0 ;
    read1,      0  =>  read10,     0 ;
    read1,      1  =>  init,       0 ;
    read10,     0  =>  init,       0 ;
    read10,     1  =>  read101,    0 ;
    read101,    0  =>  read10,     0 ;
    read101,    1  =>  read1011,   0 ;
    read1011,   0  =>  read10,     1 ;
    read1011,   1  =>  read1,      0 ;
  END TABLE;

END;
```

In dem Beispiel des Sequenz-Analysators kann man bei der Beschreibung auch ganz auf die Verwendung einer State Machine verzichten, wie die nachfolgende Realisierung mit Hilfe eines Schieberegisters zeigt:

```
CONSTANT sequence_to_detect = B"10110";

SUBDESIGN analyze5 (
  reset, clk, d: INPUT;
  recognized: OUTPUT;
)

VARIABLE buffer[4..0]: DFF;

BEGIN
  buffer[].clrn = !reset;
  buffer[].clk = clk;
  buffer[] = (buffer[3..0], d);

  recognized = (buffer[] == sequence_to_detect);

END;
```

Auch hierzu existiert wieder eine Mealy-Realisierung:

```
CONSTANT prefix_to_detect = B"1011";
CONSTANT suffix = 0;

SUBDESIGN analyze6 (
  reset, clk, d: INPUT;
  recognized: OUTPUT;
)

VARIABLE buffer[3..0]: DFF;

BEGIN
  buffer[].clrn = !reset;
  buffer[].clk = clk;
  buffer[] = (buffer[2..0], d);

  recognized = (buffer[] == prefix_to_detect) & (d == suffix);

END;
```

Welche der in diesem Abschnitt vorgestellten Alternativen für die Beschreibung von Automaten nun die geeignetste ist, hängt von vielen Faktoren ab. Zunächst muß festgestellt werden, ob das unterschiedliche Zeitverhalten von Moore- und Mealy-Realisierungen sowie andere Abweichungen im Verhalten für die zu lösende Aufgabenstellung relevant sind oder nicht. Anschließend ist zu überprüfen, welche der verbleibenden Alternativen den Anforderungen an Geschwindigkeit und Ressourcenbedarf genügen. In dem Beispiel des Sequenz-Analysators stellen beispielsweise die vier Beschreibungen mit Hilfe von State Machines, gemessen an der Anzahl der belegten Logikzellen, die kostengünstigsten Realisierungen dar.[3]

[3] Logiksynthese mit den Standard-Optionen und Bausteinserie MAX7000

Kapitel 6

Steuerung der Logiksynthese in MAX+plusII

Die Steuerung von Logiksynthese und Plazierung erfolgt in der MAX+plusII Oberfläche zum einen über den Menüpunkt „Assign“. Als wichtigste Festlegung wird über den Unterpunkt der Zielbaustein ausgewählt („Assign-Device“), wo entweder nur die Baustein-Serie (CLASSIC, MAX5000, MAX7000, FLEX8000, u.a) oder aber auch ein ganz bestimmter Zielbaustein (z.B. EPM7128LC84-20) spezifiziert werden kann. Über weitere Untermenüpunkte kann der Logiksynthese-Algorithmus variiert werden - sowohl für das Design insgesamt („Global Logic Synthesis“) als auch für einzelne Teile („Logic Options“). Als eine weitere recht nützliche Möglichkeit ist die Bildung von sogenannten *Cliquen* zu nennen. In einer Clique können Teile eines Designs (z.B. eine Menge von Makrofunktionen) als zusammengehörig deklariert werden. Bei der Plazierung versucht der MAX+plusII Compiler dann die Mitglieder der Clique in benachbarten Logikzellen zu plazieren, so daß möglichst kurze Verbindungen innerhalb der Clique resultieren.

Für eine manuelle Plazierung einzelner Logikzellen sowie für die Festlegung, wo welche Pins lokalisiert sein sollen, gibt es seit der Version 5 einen speziellen *Floorplan Editor*, mit dessen Hilfe die entsprechenden Zuweisungen mittels Drag-and-drop mit der Maus erfolgen können.

Für nähere Informationen zu den vorgenannten Features siehe insebesondere die umfangreiche Online-Hilfe in MAX+plusII.

6.1 Abschätzen des Realisierungsaufwands

Hinsichtlich der vom Compiler aus einer AHDL-Beschreibung synthetisierten Realisierung muß sich der Entwerfer stets vergegenwärtigen, welche Logikfunktionen er mit seiner Beschreibung spezifiziert, und die hierfür benötigte Anzahl von Logikzellen abschätzen. In vielen Fällen wird die Logiksynthese eine optimale oder ausreichend gute Abbildung auf die Logikzellen des Ziel-Devices ermitteln. Dieses ist jedoch nicht immer und insbesondere dann nicht der Fall, wenn die zu realisierenden Logikfunktionen sehr komplex werden. In Extremfällen bricht die Logiksynthese mit der Fehlermeldung „Logic too complex.” ab. Aber auch wenn die Logiksynthese erfolgreich ist, sollte man anhand des Report-Files überprüfen, ob die erstellte Realisierung nicht unnötig Ressourcen verschwendet.

Ein wichtiger Anhaltspunkt für die Anzahl der von einem Design mindestens benötigten Logikzellen ergibt sich aus deren Struktur: jede Logikzelle verfügt über Ressourcen zur Realisierung **einer** logischen Funktion, deren

Ergebnis auf das Flipflop der Logikzelle geführt wird.[1] Für jedes Flipflop eines Designs, also für jede Instanz eines Flipflop-Primitivs (`DFF`, `DFFE`, `JKFF`, `JKFFE`, `SRFF`, `SRFFE`, `TFF` oder `TFFE`) wird somit eine Logikzelle benötigt. Ebenso belegt jeder primäre Ausgang des Designs eine Logikzelle, es sei denn, der auszugebende Wert ist das Ausgangssignal eines Flipflops, für das ja ohnehin bereits eine eigene Logikzelle belegt werden mußte.

Eine untere Schranke für die Anzahl der von einem Design benötigten Logikzellen ist deshalb gegeben durch

```
<Anzahl Flipflops> + <Anzahl nicht Flipflops entstammender Outputs>
```

Wie groß die Anzahl der darüberhinaus benötigten Logikzellen ist, hängt von Art und Komplexität der zu implementierenden logischen Funktionen ab. Für Designs mit einem im Vergleich zum Bedarf an logischen Verknüpfungen großen Anteil an speichernden Elementen wird diese untere Schranke häufig erreicht. Sind viele verschiedene logische Funktionen mit zahlreichen Argumenten oder Funktionen, die sich nur schlecht auf die physikalische Struktur der Logikzellen abbilden lassen (z.B. XOR-reiche Verknüpfungen) zu implementieren, so kann selbst ein gutes Syntheseergebnis einen deutlich höheren Realisierungsaufwand haben, als es die untere Schranke angibt. Es bleibt dem Entwerfer überlassen, schlechte Ergebnisse zu erkennen und durch manuelle Intervention (etwa durch explizite Allokation von Logikzellen für Zwischensignale) das Synthesergebnis für seine Bedürfnisse zu optimieren.

Für den Fall, daß nicht die Anzahl der Logikzellen sondern die Geschwindigkeit des Designs optimiert werden soll, gilt sinngemäß das gleiche: In vielen Fällen liefert die vollautomatische Logiksynthese geschwindigkeitsoptimale Realisierungen. Nicht selten können jedoch durch manuelle Steuerung der Synthese und durch Einfügen zusätzlicher Komponenten in die AHDL-Beschreibung schnellere Lösungen erzielt werden, für die nicht immer ein „Preis" in Form zusätzlicher Logikzellen zu zahlen ist.

6.2 Synthese- und Device-Optionen

MAX+plusII verfügt über eine Reihe von Optionen, mit denen die Logiksynthese gesteuert und Device-Parameter gesetzt werden können. Die Vielzahl der Optionen im einzelnen zu erläutern, würde den Rahmen dieser kurzen Einführung sprengen, zumal man mit den voreingestellten Optionen in den meisten Fällen akzeptable Ergebnisse erzielt. Bei nicht befriedigendem Synthese-Ergebnis ist es stark von dem zu implementierenden Design-Projekt abhängig, ob durch Verändern von Optionen eine Verbesserung erreicht werden kann, so daß generelle Aussagen hierzu nicht getroffen werden können. Die Erfahrungen des Autors sind hier überwiegend negativ, d.h. Verändern von Optionen führte in den meisten Fällen eher zu einer Verschlechterung denn zu einer Verbesserung der Resultate.

Über die Wirkung der einzelnen Optionen stehen unter MAX+plusII im Bedarfsfall online-Hilfetexte zur Verfügung, die ausreichend Informationen geben, um einen gezielten Einsatz zu ermöglichen. An dieser Stelle wird daher auf nähere Erläuterungen verzichtet.

[1] Diese Aussage gilt so nur für die Altera-Devices der Serien CLASSIC, MAX5000 und MAX7000. Die Logikzellen in den FPGAs der FLEX-Serie verfügen über einige zusätzliche Einrichtungen, für die die nachfolgenden Abschätzungen nur bedingt zutreffen.