AHDL Crash-Kurs

Version 1.2

Matthias Meyer
Institut für Nachrichtenvermittlung und Datenverarbeitung
Universität Stuttgart

Inhalt

1.	**Einleitung**	**2**
2.	**Boolesche Größen, Vektoren und Konstanten**	**3**
2.1.	Boolesche Größen	3
2.2.	Vektoren	3
2.3.	Konstanten	4
3.	**Aufbau eines AHDL Text Design Files**	**6**
3.1.	Constant Statements	7
3.2.	Function Prototype Statements	7
3.3.	Subdesign Section	8
3.4.	Variable Section	8
3.5.	Logic Section	9
4.	**Kombinatorische Logik**	**11**
4.1.	Boolesche Gleichungen	11
4.2.	Wahrheitstafeln	15
4.3.	IF Statements	16
4.4.	Case Statements	17
4.5.	Kombinatorische Funktionsprimitive	17
5.	**Sequentielle Logik**	**22**
5.1.	Flipflop-Primitive	22
5.2.	Register	24
5.3.	Zähler	25
5.4.	Automaten	25

1. Einleitung

Die Altera Hardware Description Language (AHDL) ist eine Hochsprache für die Beschreibung und Entwicklung komplexer kombinatorischer und sequentieller Logik. Es handelt sich dabei um eine proprietäre Hardwarebeschreibungssprache des amerikanischen Unternehmens ALTERA. Mit Hilfe von AHDL spezifizierte Logik-Designs können in eine Vielzahl verschiedener Altera-EPLDs (Erasable and Programmable Logic Devices) programmiert werden. Dabei ist AHDL unabhängig von der EPLD-Familie und dem speziellen EPLD-Typ, mit dessen Hilfe ein Logik-Design realisiert werden soll. Größere Logikdesigns können in mehrere Teildesigns (Subdesigns) zerlegt werden, die zusammengesetzt eine hierarchische Struktur ergeben. Dadurch ist es möglich, einmal entwickelte Subdesigns in Form von Makrofunktionen häufiger zu verwenden (modulares Design). Ein Simulator ermöglicht es, das logische Verhalten und das Zeitverhalten eines Logikdesigns vor dem Einsatz in der Schaltung am Rechner zu simulieren. Dabei werden bereits die technischen Daten des speziellen EPLDs berücksichtigt, mit dessen Hilfe das Logikdesign realisiert werden soll.

Das folgende einleitende Beispiel zeigt die Beschreibung eines Volladdierers in AHDL.

Beispiel:

```
SUBDESIGN volladd
    (
    a, b, carry_in: INPUT;
    carry_out, sum: OUTPUT;
    )

BEGIN

    sum =          a & !b & !carry_in
               # !a &  b & !carry_in
               # !a & !b &  carry_in
               #  a &  b &  carry_in;

    carry_out =  a & b
               # a & carry_in
               # b & carry_in;

END;
```

In der **Subdesign Section** werden die Eingangssignale und die Ausgangssignale des mit Hilfe des Subdesigns beschriebenen Logikblocks deklariert. Der Volladdierer im Beispiel verfügt über die beiden Eingänge a und b und den Übertragseingang carry_in. Die drei binären Variablen werden vom Volladdierer zu einem zweistelligen binären Ergebnis in Form eines Übertragsausgangs carry_out und eines Summenausgangs sum verarbeitet.

In der **Logic Section** zwischen den Schlüsselworten BEGIN und END werden die logischen Verknüpfungen des Subdesigns definiert, mit deren Hilfe der Volladdierer realisiert wird.

2. Boolesche Größen, Vektoren und Konstanten

Kombinatorische und sequentielle Logik basiert auf zweiwertigen Booleschen Variablen und der darauf definierten zweiwertigen Booleschen Algebra. Zweiwertige Boolesche Größen können nur die beiden Wahrheitswerte "wahr" und "nicht wahr" annehmen.[1]

Werden mit Hilfe logischer Operationen arithmetische Operationen wie z. B. die Addition realisiert, so erfolgt eine Uminterpretation der Booleschen Wahrheitswerte "wahr" und "nicht wahr" (logische Interpretation) in die binären Werte "0" und "1" (arithmetische Interpretation).

Durch das Aneinanderreihen von binären Größen und durch die Zuweisung von Stellenwerten ist es möglich, ganze Zahlen darzustellen. Aus diesem Grund ist es notwendig, zwischen den Wahrheitswerten "wahr" und "nicht wahr" und den ganzen Zahlen "0" und "1" zu unterscheiden.

Die folgenden Abschnitte beschreiben den Zusammenhang zwischen Booleschen Größen und ganzen Zahlen. Dieser Zusammenhang wird in AHDL durch Vektoren (Bitvektoren, Gruppen) hergestellt. Dabei werden auch numerische Konstanten als Vektoren interpretiert.

2.1 Boolesche Größen

Einzelne Signale an der Schnittstelle oder im Inneren eines Logikdesigns werden in AHDL stets als Boolesche Größen interpretiert. Dabei wird der Boolesche Wert "wahr" mit Hilfe des reservierten Wortes VCC und der Boolesche Wert "nicht wahr" mit Hilfe des reservierten Wortes GND dargestellt.

2.2 Vektoren

Boolesche Größen desselben Typs können zu Vektoren (Gruppen) zusammengefaßt werden. Ein Vektor kann bis zu 256 Elemente (Komponenten, Bits) enthalten. Vektoren können in AHDL als Feldvektoren und als Aufzählungsvektoren notiert werden:

Ein **Feldvektor** besteht aus einem Bezeichner, dem sich eine Bereichsangabe in eckigen Klammern anschließt. Die Bereichsangabe erfolgt in absteigender Reihenfolge. Die Grenzen sind ganze, nicht negative Zahlen.

Beispiel: `a[5..0]`

Wenn ein Feldvektor einmal definiert ist, können auch Teilvektoren des Vektors verwendet werden.

Beispiel: `a[4..2]`

1. In diesem Dokument werden die Begriffe "zweiwertige Boolesche Größe" und "Boolesche Größe" der Einfachkeit halber synonym verwendet.

Bei Teilvektoren mit nur einem Element können die eckigen Klammern weggelassen werden.

Beispiel: `a[0] = a0`

Falls der ganze Feldvektor gemeint ist, kann die Bereichsangabe zwischen den eckigen Klammern entfallen.

Beispiel: `a[] = a[5..0]`

Ein **Aufzählungsvektor** besteht aus einer in runden Klammern eingeschlossenen Liste von Größen, die durch Kommata voneinander getrennt werden. Ein Aufzählungsvektor kann auch Feldvektoren oder Teilvektoren enthalten.

Beispiel:
```
(a[4..2], b, c)
a[4..2] = (a4, a3, a2)
```

Feldvektoren und Aufzählungsvektoren können aufgefaßt werden als Gruppen von gleichartigen Booleschen Größen. Neben dieser logischen Interpretation können Vektoren (insbesondere Feldvektoren) auch als ganze Zahlen interpretiert werden. Dabei wird jedem Element des Vektors ein dualer Stellenwert zugewiesen (arithmetische Interpretation). Negative Zahlen werden dabei im Zweierkomplement dargestellt.

Beispiel: logische Interpretation:
```
a[5..0] = (VCC, VCC, GND, GND, VCC, VCC);
```

arithmetische Interpretation:
```
a[5..0] = 51;       oder
a[5..0] = -13;
```

2.3 Konstanten

Numerische Konstanten sind positive oder negative ganze Zahlen, die als arithmetische Vektoren interpretiert werden. Die einzelnen Elemente (oder Komponenten) des Vektors entsprechen dabei den Ziffern der Konstante in Binärdarstellung. Negative Zahlen werden im Zweierkomplement dargestellt. Die Anzahl der Elemente des Konstantenvektors richtet sich nach der Größe des Vektors, mit dem die Konstante verknüpft, verglichen oder dem die Konstante zugewiesen wird.

Beispiel:
```
19 = (GND, VCC, GND, GND, VCC, VCC)
-3 = (VCC, VCC, VCC, VCC, GND, VCC)
```

Neben der dezimalen Schreibweise können Konstanten in binärer, in oktaler und in hexadezimaler Schreibweise angegeben werden.

Tabelle 1: Zahlendarstellungen

Radix	Beispiel
dezimal	`239`
binär	`B"11101111"`
oktal	`Q"357"` oder `O"357"`
hexadezimal	`X"EF"` oder `H"EF"`

3. Aufbau eines AHDL Text Design Files

Subdesigns werden in sogenannten Text Design Files (TDF) abgelegt. Ein Text Design File setzt sich aus mehreren Abschnitten (Sections) zusammen. Die nachfolgende Tabelle enthält alle wichtigen Sections eines Text Design Files in ihrer üblichen Reihenfolge. Jedes Text Design File muß eine Subdesign Section und eine Logic Section enthalten. Die Definition von Konstanten und Funktionsprototypen sowie die Deklaration von Variablen ist optional.

Tabelle 2: Abschnitte (Sections) in einem AHDL Text Design File

Section	Beschreibung
Constant Statements	Definition von Konstanten
Function Prototype Statements	Definition von Funktionsprototypen für Makrofunktionen
Subdesign Section	Deklaration der Schnittstelle des Subdesigns (Ein- und Ausgangssignale)
Variable Section	Deklaration von logischen Elementen (Instanzen von Funktionsprimitiven oder Makrofunktionen)
Logic Section	Definition der logischen Operationen

Beispiel:

```
CONSTANT DISP = 3;

FUNCTION volladd (a, b, carry_in)
    RETURNS (carry_out, sum);

SUBDESIGN add_disp
    (
    a[3..0]:     INPUT;
    s[3..0]:     OUTPUT;
    c:           OUTPUT;
    )

VARIABLE
    add[3..0]:  volladd;

BEGIN
    add[].a = a[];
    add[].b = DISP;
    add[0].carry_in = GND;
    add[1].carry_in = add[0].carry_out;
    add[2].carry_in = add[1].carry_out;
    add[3].carry_in = add[2].carry_out;
    c = add[3].carry_out;
    s[] = add[].sum;
END;
```

3.1 Constant Statements

Mit Hilfe von Constant Statements werden numerischen Konstanten symbolische Namen zugewiesen.

Die Verwendung von Konstanten erhöht die Lesbarkeit eines Text Design Files. Symbolische Konstanten sollten vor allem dann verwendet werden, wenn numerische Konstanten in einem Design öfter auftauchen. Sollte sich die Konstante im Lauf der Entwicklungsphase ändern, dann muß nur die entsprechende Konstantendefinition abgeändert werden.

Beispiel:
```
CONSTANT DISP = 3;
```

Durch die obenstehende Anweisung wird der symbolischen Konstanten DISP die numerische Konstante 3 zugewiesen.

3.2 Function Prototype Statements

Logikdesigns können in mehrere Teildesigns zerlegt werden (Subdesigns). Jedes Subdesign kann als "Black Box" betrachtet werden, von dem von außen nur die Eingänge und die Ausgänge sichtbar sind. Eine solche "Black Box" wird als Makrofunktion bezeichnet.

Soll eine Makrofunktion in einem Design verwendet werden, dann muß zuvor der Name und die Schnittstelle der Makrofunktion mit Hilfe eines Function Prototype Statements eingeführt werden. Die Schnittstellensignale von Makrofunktionen werden als **Ports** bezeichnet.

Beispiel:
```
FUNCTION volladd (a, b, carry_in)
    RETURNS (carry_out, sum);
```

Im Beispiel wird die Makrofunktion volladd eingeführt. Sie realisiere einen Volladdierer mit den Eingangsports a, b und carry_in und den Ausgangsports carry_out und sum (vgl. einleitendes Beispiel). Das Function Prototype Statement definiert nur die Schnittstelle des Volladdierers, seine tatsächliche Realisierung bleibt an dieser Stelle verborgen ("Black Box", vgl. Bild 1).

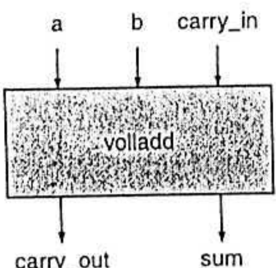

Bild 1: Die Schnittstelle der Makrofunktion "volladd"

Bevor die Makrofunktion verwendet werden kann, müssen zuvor eine oder mehrere Instanzen der Makrofunktion in der Variable Section deklariert werden (siehe Abschnitt 3.4).

3.3 Subdesign Section

In der Subdesign Section werden die Eingänge und die Ausgänge eines Subdesigns deklariert. Zusätzlich wird das Subdesign mit einem Namen versehen, über den das Subdesign als Makrofunktion in anderen Subdesigns referenziert werden kann. Schnittstellensignale des gleichen Typs können in Gruppen zusammengefaßt und als Feldvektoren deklariert werden.

Beispiel:
```
SUBDESIGN add_disp
    (
    a[3..0]:    INPUT;
    s[3..0]:    OUTPUT;
    c:          OUTPUT;
    )
```

Das Subdesign im Beispiel trägt den Namen add_disp. Es werden vier Eingänge (a3, a2, a1 und a0) und 5 Ausgänge (s3, s2, s1, s0 und c) deklariert, wobei die 4 Eingänge zum Vektor a[] und 4 der 5 Ausgänge zum Vektor s[] zusammengefaßt sind.

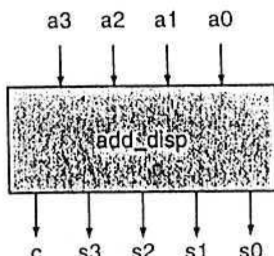

Bild 2: Die Schnittstelle des Subdesigns "add_disp"

3.4 Variable Section

In der Variable Section werden Instanzen von **Makrofunktionen** und von **Funktionsprimitiven** deklariert. Nach ihrer Deklaration in der Variable Section stehen diese Instanzen als **logische Elemente** zur Verfügung.
Mit Hilfe von **Makrofunktionen** können bereits bestehende Subdesigns in das aktuelle Subdesign integriert werden. Über **Funktionsprimitive** stehen logische Basiselemente wie z. B. Flipflops (Speicherelemente) zur Verfügung. Funktionsprimitive sind vordefinierte Makrofunktionen, deren Schnittstelle implizit ohne vorherige Definition bekannt ist.

Die Signale an der Schnittstelle eines logischen Elements werden als Ports bezeichnet. Den Eingangsports von logischen Elementen können Werte zugewiesen werden, während Ausgangsports logischer Elemente in booleschen Ausdrücken verwendet werden können.

Makrofunktionen müssen vor der Deklaration von Instanzen mit Hilfe eines Function Prototype Statements eingeführt werden (siehe Abschnitt 3.2). Funktionsprimitive können direkt instanziiert werden.

Logische Elemente können in Gruppen zusammengefaßt und als Feldvektoren deklariert werden. Im folgenden Beispiel werden vier Instanzen des Volladdierers volladd als Vektor mit vier Elementen deklariert.

Beispiel:
```
VARIABLE
    add[3..0]:  volladd;
```

Entsprechend können Instanzen von Funktionsprimitiven als Vektoren deklariert werden. So realisiert man z. B. Register als Vektoren von Flipflops (siehe Abschnitt 5.2).

Als besondere Form von Flipflop-Vektoren (Speichervektoren) ist es möglich, in der Variable Section eine Gruppe von Speicherelementen als Zustandsvektor eines Automaten (Finite State Machine) zu deklarieren (siehe Abschnitt 5.4).

Boolesche Grundoperationen (z. B. das logische UND) und einige andere Grundoperationen können mit Hilfe von Operatoren realisiert werden. Es ist nicht notwendig, logische Grundgatter als Makrofunktionen einzuführen und in der Variable Section als Instanzen zu deklarieren.

3.5 Logic Section

In der Logic Section werden Zuweisungen und logische Verknüpfungen definiert. Logische Verknüpfungen können als Boolesche Funktionen, als Wahrheitstabellen oder mit Hilfe von IF und CASE Anweisungen definiert werden (siehe Kapitel 4). Zuweisungen können als triviale Boolesche Gleichungen aufgefaßt werden.

In der Logic Section kann auf die Ports von logischen Elementen zugegriffen werden, indem man hinter dem Namen der Instanz den Namen des entsprechenden Ports angibt. Der Name des Ports wird dabei durch einen Punkt vom Namen der Instanz getrennt.

Beispiel:
```
BEGIN

    add[3..0].a = a[3..0];
    add[3..0].b = DISP;

    add0.carry_in = GND;
    add1.carry_in = add0.carry_out;
    add2.carry_in = add1.carry_out;
    add3.carry_in = add2.carry_out;

    c = add3.carry_out;

    s[3..0] = add[3..0].sum;

END;
```

Die im Beispiel definierten Zuweisungen sind in zur Veranschaulichung zusätzlich in Form eines Blockschaltbildes dargestellt (Bild 3).

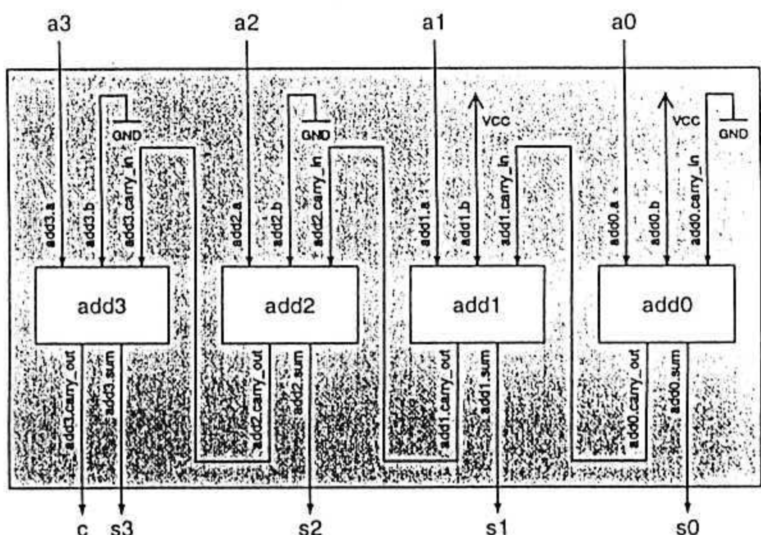

Bild 3: Grafische Darstellung der in "add_disp" definierten Zuweisungen

Die Anweisungen in der Logic Section beschreiben die **Struktur** eines Logikdesigns und legen **keine zeitliche Reihenfolge** fest. Alle Ausdrücke werden gleichzeitig ausgewertet. Die Reihenfolge der Anweisungen in der Logic Section ist beliebig. Diese Eigenschaft unterscheidet Hardwarebeschreibungssprachen von den meisten konventionellen Programmiersprachen, bei denen Anweisungen sequentiell, d. h. nacheinander ausgeführt werden.

4. Kombinatorische Logik

Kombinatorische Logik zeichnet sich dadurch aus, daß die Ausgangssignale kombinatorischer Logik zu jedem Zeitpunkt nur von den Eingangssignalen zu diesem Zeitpunkt abhängen. Kombinatorische Logik kann mit Hilfe von Booleschen Gleichungen, mit Wahrheitstabellen oder mit Hilfe von IF und CASE Statements implementiert werden.

4.1 Boolesche Gleichungen

4.1.1 Logische Operatoren

Boolesche Gleichungen setzen sich aus Booleschen Größen und den logischen Grundoperatoren UND, ODER und NICHT und aus daraus ableitbaren Operatoren wie z. B. dem NICHT UND oder dem EXKLUSIV ODER zusammen.

Die in AHDL zur Verfügung stehenden logischen Operatoren sind in der folgenden Tabelle zusammengefaßt. Jeder Operator kann wahlweise mit Hilfe eines Symbols oder mit Hilfe eines reservierten Wortes dargestellt werden.

Tabelle 3: Logische Operatoren

Symbol	res. Wort	Beschreibung
!	NOT	logisches NICHT
&	AND	logisches UND
#	OR	logisches ODER
$	XOR	logisches EXKLUSIV ODER
!&	NAND	logisches NICHT UND
!#	NOR	logisches NICHT ODER
!$	XNOR	logisches EXKLUSIV NICHT ODER

Beispiel: `u = a & b # a & c # b & c;`

Der "&"-Operator hat Vorrang vor dem "#"-Operator. Eine vollständige Zusammenstellung aller Operatoren und ihrer Prioritäten folgt im Abschnitt 4.1.4. Die Reihenfolge der Auswertung kann durch Klammern verändert werden.

Beispiel: `v = a & (b # a) & (c # b) & c;`

Logische Operatoren können auch auf Vektoren und Teilvektoren angewendet werden. Alle beteiligten Vektoren oder Teilvektoren müssen dieselbe Anzahl von Elementen (Größe) aufweisen. Tauchen in einer Gleichung Vektoren und einzelne Boolesche Größen auf, so wird jedes Element des Vektors mit der entsprechenden Booleschen Größe verknüpft.

Beispiel: `y[3..0] = a[3..0] & !sel # a[7..4] & sel;`

entspricht
```
y3 = a3 & !sel # a7 & sel;
y2 = a2 & !sel # a6 & sel;
y1 = a1 & !sel # a5 & sel;
y0 = a0 & !sel # a4 & sel;
```

Der unäre NICHT-Operator wird auf Vektoren komponentenweise angewendet.

Beispiel:
```
!a[3..0] = (!a3, !a2, !a1, !a0)
!9 = !(1,0,0,1) = (!1,!0,!0,!1) = (0,1,1,0) = 6
```

Im letzten Beispiel wird davon ausgegangen, daß die Konstante "!9" im Kontext mit Vektoren der Größe 4 (d. h. Vektoren mit je 4 Elementen) verwendet wird.

Einzelsignale werden ausschließlich als Boolesche Größen interpretiert. Ihnen dürfen deshalb lediglich die logischen Konstanten VCC und GND zugewiesen werden. Einzelsignalen dürfen keine numerischen Konstanten zugewiesen werden.

Beispiel: falsch: `y = 0;`
richtig: `y = GND;`

4.1.2 Arithmetische Operatoren

Die arithmetischen Operatoren "+" und "-" werden verwendet, um arithmetische Additionen und Subtraktionen auf Vektoren und Konstanten auszuführen. Dabei müssen alle beteiligten Vektoren oder Teilvektoren die gleiche Größe aufweisen. Konstanten in arithmetischen Ausdrücken werden gegebenenfalls durch vorzeichenrichtige Erweiterung an die Größe der beteiligten Vektoren oder Teilvektoren angepaßt.

Beispiel: `a[3..0] + b[7..4] - 3;`

entspricht `(a3, a2, a1, a0) + (b7, b6, b5, b4) + (1, 1, 0, 1);`

Im obigen Beispiel ist zu beachten, daß die Addition nicht komponentenweise erfolgt wie bei Vektoren der linearen Algebra. Zwar werden die Bits entsprechender Stellen jeweils aufaddiert, jedoch wird zusätzlich ein eventuell in der Stelle mit dem nächstgeringeren Stellenwert entstandener Übertrag mitaddiert.

Der unäre "-"-Operator kann verwendet werden, um das Zweierkomplement eines Vektors oder einer Konstante zu bilden.

Beispiel: `a[3..0] = -b[3..0];`

entspricht `a[3..0] = !b[3..0] + 1;`

Die arithmetischen Operatoren sind mit Vorsicht einzusetzen, da sie wegen der Übertragsberechnung nach dem "Carry Look Ahead"-Verfahren insbesondere bei großen Vektoren zu recht komplexen kombinatorischen Ausdrücken führen.

4.1.3 Vergleichsoperatoren

Das Ergebnis eines Vergleichsoperators kann entweder "wahr" (logische Konstante VCC) oder "nicht wahr" (logische Konstante GND) sein. Infolge dieser Eigenschaft können Vergleiche als Boolesche Größen in Booleschen Gleichungen verwendet werden. In der nachfolgenden Tabelle sind alle in AHDL zur Verfügung stehenden Vergleichsoperatoren zusammengefaßt.

Tabelle 4: Vergleichsoperatoren

Symbol	Typ	Beschreibung
==	arithmetisch, logisch	gleich
!=	arithmetisch, logisch	ungleich
<	arithmetisch	vorzeichenloses kleiner als
<=	arithmetisch	vorzeichenloses kleiner oder gleich als
>	arithmetisch	vorzeichenloses größer als
>=	arithmetisch	vorzeichenloses größer oder gleich als

Die Vergleichsoperatoren "gleich" und "ungleich" können auf logisch interpretierte und auf arithmetisch interpretierte Vektoren angewendet werden. Die restlichen Vergleichsoperatoren sind nur für arithmetisch interpretierte Vektoren sinnvoll anwendbar. Dabei werden die Vektoren als vorzeichenlose Binärzahlen interpretiert. Operatoren für den Vergleich vorzeichenbehafteter Vektoren sind nicht vorgesehen.

Beispiel: `zero = a[3..0] == 0;`

entspricht `zero = !(a3 # a2 # a1 # a0);`

oder `zero = !a3 & !a2 & !a1 & !a0;` (Theorem von de Morgan)

4.1.4 Priorität der Operatoren

Die Operanden in logischen und arithmetischen Ausdrücken werden in der Reihenfolge ihrer Priorität ausgewertet, wobei Priorität 1 die höchste Priorität darstellt (siehe Tabelle 5).

Operanden gleicher Priorität werden von links nach rechts ausgewertet. Die Reihenfolge der Auswertung kann durch runde Klammern beeinflußt werden.

In der folgenden Tabelle sind die Prioritäten aller Operanden zusammengestellt.

Tabelle 5: Priorität der Operanden

Priorität	Operator	Typ	Beschreibung
1	-	unär	arithmetische NEGATION
1	!	unär	logisches NICHT
2	+	binär	arithmetisches PLUS
2	-	binär	arithmetisches MINUS
3	==	binär	arithmetisches/logisches GLEICH
3	!=	binär	arithmetisches/logisches UNGLEICH
3	<	binär	arithmetisches KLEINER
3	<=	binär	arithmetisches KLEINER ODER GLEICH
3	>	binär	arithmetisches GRÖSSER
3	>=	binär	arithmetisches GRÖSSER ODER GLEICH
4	&	binär	logisches UND
4	!&	binär	logisches NICHT UND
5	$	binär	logisches EXKLUSIV ODER
5	!$	binär	logisches EXKLUSIV NICHT ODER
6	#	binär	logisches ODER
6	!#	binär	logisches NICHT ODER

4.2 Wahrheitstafeln

Wahrheitstabellen (Truth Tables) können in AHDL verwendet werden, um kombinatorische Logik zu beschreiben. Darüberhinaus können mit Hilfe von Wahrheitstabellen Zustandsübergänge von Automaten definiert werden (siehe Abschnitt 5.4).

Beispiel:

```
TABLE

    a,  b,  c[4..1] =>  s[1..0], t;

    0,  0,  B"0XXX" =>   1,      1;
    0,  0,  B"1XXX" =>   3,      0;
    0,  1,  B"XX00" =>   2,      0;
    0,  1,  B"XX01" =>   0,      0;
    0,  1,  B"XX10" =>   0,      1;
    0,  1,  B"XX11" =>   1,      X;
    1,  X,  B"XX0X" =>   3,      X;
    1,  X,  B"XX1X" =>   0,      X;

END TABLE;
```

Wahrheitstabellen werden durch die Schlüsselworte TABLE und END TABLE eingeschlossen. Eine Wahrheitstabelle besteht aus einer Kopfzeile und mehreren Hauptzeilen. Der linke und der rechte Teil einer Wahrheitstabelle wird in allen Zeilen durch das Symbol "=>" getrennt. Die einzelne Spalten innerhalb eines Teils werden durch Kommata getrennt. Die Kopfzeile und jede Hauptzeile werden durch einen Semikolon abgeschlossen.

Der linke Teil der Kopfzeile enthält einzelne Signale oder Vektoren, die einen Wert repräsentieren. Dabei handelt es sich um die Eingangssignale des Subdesigns oder um Ausgangsports von logischen Elementen innerhalb des Subdesigns.

Der rechte Teil der Kopfzeile enthält einzelne Signale oder Vektoren, denen ein Wert zugewiesen werden kann. Dabei handelt es sich um die Ausgangssignale des Subdesigns oder um Eingangsports von logischen Elementen innerhalb des Subdesigns.

Der Hauptteil der Wahrheitstabelle darf ausschließlich Konstanten enthalten. Mit Vektoren bezeichnete Spalten enthalten im Hauptteil numerische Konstanten, mit Einzelsignalen bezeichnete Spalten die logischen Konstanten VCC und GND. Zur besseren Lesbarkeit dürfen die logischen Konstanten VCC und GND für Einzelsignale (Boolesche Größen!) in Wahrheitstabellen ausnahmsweise durch "1" und "0" ersetzt werden.

Im Hauptteil der Wahrheitstabelle können sowohl für einzelne Signale als auch für Teilvektoren "don't cares" verwendet werden. "Don't cares" für einzelne Signale werden mit dem reservierten Wort "X" bezeichnet. Sollen einzelne Elemente (Bits) eines Vektors als "don't cares" verwendet werden, so ist die entsprechende numerische Konstante in binärer Schreibweise anzugeben, wobei die "don't care"-Stellen durch ein "X" gekennzeichnet werden.

"Don't cares" können auch im linken Hauptteil der Tabelle verwendet werden. Dabei ist darauf zu achten, daß es keine Eingangs-Bitmuster gibt, die auf mehr als eine Zeile zutreffen.

Im folgenden ist die im Beispiel dieses Abschnitts realisierte Wahrheitstabelle wiedergegeben.

Tabelle 6: Im Beispiel realisierte Wahrheitstabelle

a	b	c4	c3	c2	c1	s1	s0	t
0	0	0	X	X	X	0	1	1
0	0	1	X	X	X	1	1	0
0	1	X	X	0	0	1	0	0
0	1	X	X	0	1	0	0	0
0	1	X	X	1	0	0	0	1
0	1	X	X	1	1	0	1	X
1	X	X	X	0	X	1	1	X
1	X	X	X	1	X	0	0	X

4.3 IF Statements

Kombinatorische Logik kann mit Hilfe von IF Statements in Form "bedingter Logik" formuliert werden. Dabei werden boolesche Ausdrücke in Abhängigkeit von Bedingungen ausgewertet.

Beispiel:

```
IF a[] == b[] THEN
    equal = VCC;
ELSIF a[] < b[] THEN
    equal = GND;
    smaller = VCC;
ELSE
    equal = GND;
    smaller = GND;
END IF;
```

Bedingte Logik kann immer durch unbedingte Boolesche Gleichungen ersetzt werden. So läßt sich das oben angegebene Beispiel ohne die Verwendung bedingter Logik wie folgt formulieren:

Beispiel:

```
equal =   a[] == b[];
smaller = a[] <  b[];
```

Sinnvoll eingesetzt können IF Statements die Lesbarkeit eines Designs erhöhen. Ein weiteres sinnvolles Einsatzgebiet von IF Statements ist die Definition von Zustandsübergängen von Automaten (siehe Abschnitt 5.4).

Wenn möglich, sollte jedoch auf die Verwendung bedingter Logik verzichtet werden, da die Vollständigkeit mit Hilfe von IF Statements definierter bedingter Logik nur schwer überschaubar ist. So enthält die in Form bedingter Logik formulierte Fassung im Beispiel oben keine Aussage darüber, welcher Wert dem Signal smaller im Fall a[] = b[] zugewiesen werden soll. Ein weiterer Nachteil von IF Statements ist, daß ihre Verwendung oft zu relativ komplexer kombinatorischer Logik führt.

Besonders bei IF Statements ist darauf zu achten, daß mit Hilfe bedingter Logik die Struktur eines Logikdesigns und keine zeitliche Reihenfolge definiert wird. Alle Ausdrücke in IF und ELSIF Statements werden **gleichzeitig** ausgewertet. In konventionellen Programmiersprachen werden vergleichbare Statements **nacheinander** ausgewertet.

4.4 Case Statements

Mit dem CASE-Statement steht eine weitere Anweisung für die Formulierung bedingter Logik zur Verfügung. Bei einem CASE Statement handelt es sich um eine eingeschränkte Fassung des IF bzw. ELSIF Statements.

Beispiel:

```
CASE a[] IS
    WHEN 1 =>
        y = c & d;
    WHEN 2 =>
        y = e & f;
    WHEN 4 =>
        y = g & h;
    WHEN 8 =>
        y = i;
    WHEN OTHERS
        y = GND;
END CASE;
```

Mit Hilfe eines CASE Statements wird ein einziger Vektor oder Teilvektor in jedem WHEN Zweig mit einer Konstante verglichen. Der WHEN OTHERS Zweig deckt alle nicht explizit aufgeführten Fälle ab.

Zum Vergleich: In einem IF Statement kann als Bedingung jeder beliebige Boolesche Ausdruck angegeben werden. Die Bedingungen in aufeinanderfolgenden IF und ELSIF Statements müssen in keiner Beziehung zueinander stehen.

Will man auf bedingte Logik nicht verzichten, so ist das CASE Statement dem IF Statement vorzuziehen, da es übersichtlicher ist und weniger aufwendige Logik zur Folge hat.

4.5 Kombinatorische Funktionsprimitive

Die nachfolgende Tabelle enthält eine Zusammenstellung der wichtigsten kombinatorischen Funktionsprimitive und ihre impliziten Funktionsprototypen. Mit Hilfe kombinatorischer Funktionsprimitive werden logische Basiselemente ohne Speicherfähigkeit

instanziiert. Einige der Basiselemente haben keine logische Funktion im eigentlichen Sinne. Sie dienen dazu, Logikdesigns zu gliedern oder dem Compiler die Verwendung bestimmter Ressourcen vorzuschreiben.

Tabelle 7: Kombinatorische Funktionsprimitive

Primitiv	Impliziter Funktionsprototyp	Beschreibung
`NODE`	`FUNCTION NODE (in) RETURNS (out);`	Internes Signal für Zwischenergebnisse (Knoten)
`SOFT`	`FUNCTION SOFT (in) RETURNS (out);`	Internes Signal für Zwischenergebnisse (Soft Buffer)
`LCELL`	`FUNCTION LCELL (in) RETURNS (out);`	Internes Signal für Zwischenergebnisse (Logic Cell)
`TRI`	`FUNCTION TRI (in, oe) RETURNS (out)`	Datentor (Tristate Buffer)
`GLOBAL`	`FUNCTION GLOBAL (in) RETURNS (out);`	Vorschrift zur Verwendung globaler Eingangssignale

4.5.1 Knoten ("Nodes")

Knoten können verwendet werden, um Zwischenergebnisse Boolescher Gleichungen mit einem Namen zu versehen. Die Deklaration von Knoten ist sinnvoll, wenn ein Boolescher Ausdruck in einem Design wiederholt auftritt. Der Ausdruck kann durch einen ausagekräftigen Knotennamen ersetzt werden, wodurch die Lesbarkeit und die Wartbarkeit eines Design-Files erhöht wird.

Knoten gehören zu den logischen Basiselementen, die in Form eines Funktionsprimitivs zur Verfügung gestellt werden. Aus diesem Grund muß ein Knoten vor seiner Verwendung in der Variable Section als Instanz deklariert werden. Im folgenden Beispiel wird ein Knoten (NODE) mit dem Namen "zero" deklariert:

Beispiel:

```
VARIABLE
    zero:   NODE;
```

In der Logic Section kann einem Knoten über seinen Eingangsport "in" ein Wert zugewiesen werden:

Beispiel:

```
zero.in = a[7..0] == 0;
```

Der Wert eines Knotens kann über seinen Ausgangsport "out" in Booleschen Gleichungen verwendet werden:

Beispiel:

```
f = zero.out & d # !zero.out & e;
```

Da aus dem Kontext immer zweifelsfrei hervorgeht, ob der Eingangsport oder der Ausgangsport des Knotens gemeint ist, kann auf die Angabe des Portnamens verzichtet werden. Aus diesem Grund kann das obige Beispiel auch wie folgt formuliert werden:

Beispiel:

```
zero = a[7..0] == 0;
f = zero & d # !zero & e;
```

Knoten werden vom Compiler während des Übersetzungsvorgangs bei jedem Auftreten durch ihre Definition ersetzt. Mit Hilfe von Node-Funktionsprimitiven ist es daher nicht möglich, den Compiler anzuweisen, den entsprechenden Ausdruck nur ein Mal zu realisieren. Wird ein Knoten mehrmals verwendet, so wird auch die entsprechende Logik auf dem Bauelement mehrmals realisiert. So wird der Ausdruck aus dem Beispiel vom Compiler in folgenden Ausdruck umgesetzt:

Beispiel:

```
f = a[7..0] == 0 & d # a[7..0] != 0 & e;
```

4.5.2 Die Funktionsprimitive "LCELL" und "SOFT"

Das Funktionsprimitiv LCELL wird genauso verwendet wie das NODE-Primitiv. Der Unterschied zum Knoten besteht darin, daß der entsprechende Boolesche Ausdruck vom Compiler nur ein Mal realisiert wird. Aus diesem Grund eignet sich das LCELL-Primitiv, um Ressourcen auf dem Bauelement zu sparen und den Compiler bei der Zerlegung komplexer logischer Ausdrücke zu unterstützen.

Mit Hilfe von LCELL-Primitiven untergliederte Boolesche Gleichungen werden mehrstufig realisiert, d. h. die entsprechenden Signale weisen größere Verzögerungszeiten auf.

Beim SOFT-Primitiv handelt es sich um ein "optionales" LCELL-Primitiv. Wenn der Compiler die mit Hilfe von SOFT-Primitiven untergliederte Logik ohne Zwischenstufen erzeugen kann, dann wird das LCELL-Primitiv wie ein NODE-Primitiv behandelt (einstufige Realisierung). Ist die Logik für eine einstufige Realisierung zu komplex, dann werden die entsprechenden SOFT-Primitive durch LCELL-Primitive ersetzt (mehrstufige Realisierung).

4.5.3 Tristate-Buffer

Tristate-Ausgänge werden mit Hilfe des Funktionsprimitivs TRI realisiert. Die Ports eines Tristate-Buffer-Primitivs sind im Bild des nachfolgenden Beispiels dargestellt.

Beispiel: Deklaration in der Variable Section

```
VARIABLE
    a:        TRI;
```

Bezeichnung der Ports in der Logic Section

`a.in`	Eingang
`a.oe`	Freigabeeingang
`a.out`	Ausgang

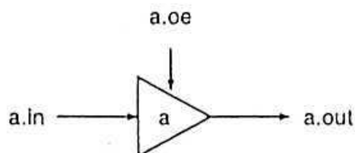

Bild 4: Die Ports eines Tristate-Buffers mit dem Bezeichner "a"

Der Tristate-Buffer ist aktiv, wenn der Freigabeeingang oe == VCC ist. Für oe == GND ist der Ausgang des Tristate-Buffers hochohmig.

Bei der Verwendung der Datenports "in" und "out" kann auf die Angabe der Portnamen verzichtet werden.

Der Ausgang eines Tristate-Buffers darf nur einem Ausgangssignal zugewiesen werden. Bausteininterne Tristate-Buffer sind nicht möglich. Der Ausgang eines Tristate-Buffers wird automatisch einem Ausgang zugewiesen, wenn der Tristate-Buffer und das entsprechende Ausgangssignal mit identischen Namen versehen werden.

Tristate-Buffer ermöglichen die Realisierung bidirektionaler Leitungen. Dazu muß das entsprechende Schnittstellensignal in der Subdesign Section mit Hilfe des reservierten Wortes BIDIR deklariert werden.

Im nachfolgenden Beispiel wird der interne Vektor a[] auf den bidirektionalen Leitungen d[15..0] ausgegeben, wenn das Eingangssignal /read == GND ist. Für /read == VCC können von außen Daten angelegt werden. In beiden Fällen kann über d[] auf die Daten der bidirektionalen Leitungen zugegriffen werden.

Hinweis: Laut Konvention werden low-aktive Signale mit einem führenden "/"-Symbol dargestellt. Dabei ist das "/"-Symbol Bestandteil des Signalnamens und hat keine logische Bedeutung. Zur Negation muß der NICHT-Operator "!" verwendet werden.

Beispiel:

```
SUBDESIGN
    ...
    /read:        INPUT;
    d[15..0]:     BIDIR;
    ...

VARIABLE
    ...
    d[15..0]:     TRI;
    ...

BEGIN
    ...
    d[].oe = !/read;
    d[].in = a[];
    ...
END;
```

4.5.4 Das Funktionsprimitiv "GLOBAL"

Neben frei programmierbaren Ein-/Ausgabeleitungen verfügen viele EPLDs über dedizierte Eingangspins für bestimmte Aufgaben (Dedicated Inputs). Dazu gehören Takteingänge, Löscheingänge und Freigabeeingänge. Die Verwendung der dedizierten Eingänge bietet den Vorteil kürzerer Laufzeiten. Art und Anzahl der dedizierten Eingänge sind bausteinabhängig.

Mit Hilfe des GLOBAL-Primitivs wird der Compiler angewiesen, den entsprechenden dedizierten Eingang zu verwenden. Dabei wird dem Eingangsport "in" das entsprechende Eingangssignal zugewiesen. Der Ausgangsport "out" muß je nach Art des dedizierten Einganges einem Takt-, Lösch- oder Freigabeeingang zugewiesen werden. Da aus dem Kontext hervorgeht, ob der Eingangsport oder der Ausgangsport gemeint ist, kann auf die Angabe der Portnamen "in" bzw. "out" verzichtet werden.

4.5.5 In-Line Referenzen

Mit Hilfe von In-Line Referenzen können Funktionsprimitive ohne vorherige Deklaration in der Variable Section verwendet werden. Dabei wird die Funktionsschreibweise verwendet, wobei die Eingangsports die Rolle der Funktionsargumente übernehmen. Der Ausgangsport steht als Funktionswert zur Verfügung.

Beispiel:

```
carry_out = lcell(a & b #  a & carry_in
                         #  b & carry_in);
d[].oe = global(!/read);
```

Auch Makrofunktionen können als In-Line Referenzen verwendet werden. Hat die Makrofunktion mehr als einen Ausgangsport, so wird der Funktionswert als Aufzählungsvektor notiert.

Beispiel:

```
(cl, s) = volladd(a, b, c);
```


5. Sequentielle Logik

Sequentielle Logik zeichnet sich im allgemeinen dadurch aus, daß die Ausgangssignale sequentieller Logik zu jedem Zeitpunkt von den Eingangssignalen zu diesem Zeitpunkt und von den Eingangssignalen zu einigen oder allen vorhergegangenen Zeitpunkten abhängen. Diese Eigenschaft setzt ein Gedächtnis voraus, das mit Hilfe von Speicherelementen realisiert wird.

5.1 Flipflop-Primitive

In AHDL stehen verschiedene Arten von Speicherelementen (Flipflops) in Form vordefinierter Funktionsprimitive zur Verfügung. Tabelle 8 zeigt alle zur Verfügung stehenden Primitive zusammen mit ihrem impliziten Funktionsprototyp.

Tabelle 8: Flipflop-Primitive

Primitiv	Impliziter Funktionsprototyp	Flipflop-Typ
DFF	FUNCTION DFF (d, clk, clrn, prn) RETURNS (q);	D-Flipflop
TFF	FUNCTION TFF (t, clk, clrn, prn) RETURNS (q);	T-Flipflop
SRFF	FUNCTION SRFF (s, r, clk, clrn, prn) RETURNS (q);	SR-Flipflop
JKFF	FUNCTION JKFF (j, k, clk, clrn, prn) RETURNS (q);	JK-Flipflop
DFFE	FUNCTION DFFE (d, clk, clrn, prn, ena) RETURNS (q);	D-Flipflop mit Freigabe
TFFE	FUNCTION TFFE (t, clk, clrn, prn, ena) RETURNS (q);	T-Flipflop mit Freigabe
SRFFE	FUNCTION SRFFE (s, r, clk, clrn, prn, ena) RETURNS (q);	SR-Flipflop mit Freigabe
JKFFE	FUNCTION JKFFE (j, k, clk, clrn, prn, ena) RETURNS (q);	JK-Flipflop mit Freigabe

Jeder Flipflop-Typ besitzt einen Takteingang clk, einen asynchronen Rücksetzeingang clrn (clear negated) und einen asynchronen Setzeingang prn (preset negated). Das "n" am Ende der Portbezeichnung des Setz- und des Rücksetzeingangs weist darauf hin, daß es sich dabei um low-aktive Signale handelt. Die aktive Taktflanke ist jeweils die positive Taktflanke (Übergang von GND nach VCC).

Darüberhinaus besitzt jedes Flipflop die für seinen Typ charakteristischen Eingänge: Das D-Flipflop den Dateneingang D, das Toggle-Flipflop den Toggeleingang T, das SR-Flipflop den Setzeingang S und den Rücksetzeingang R und das JK-Flipflop die beiden Vorbereitungseingänge J und K.

Alle vier Flipfloptypen stehen in einer Standardversion und in einer Version mit Freigabeeingang zur Verfügung. Mit Hilfe des Freigabeeingangs ena kann festgelegt werden, ob sich die aktive Taktflanke auf das Flipflop auswirkt oder nicht. Bei ena == VCC verhält sich das Flipflop wie der entsprechende Standardtyp, während bei ena == GND das Flipflop seinen Wert auf jeden Fall beibehält.

Alle Ein- und Ausgangssignale der Flipfloptypen stehen bei den Instanzen der entsprechenden Funktionsprimitive als Ports zur Verfügung.

Beispiel: Deklaration in der Variable Section

```
VARIABLE
    a:        DFF;
```

Bezeichnung der Ports in der Logic Section

`a.d`	Setzeingang D (Eingangsport)
`a.clk`	Takteingang CLK (Eingangsport)
`a.prn`	asynchroner Setzeingang $\overline{\text{SET}}$ (Eingangsport)
`a.clrn`	asynchroner Rücksetzeingang $\overline{\text{RESET}}$ (Eingangsport)
`a.q`	Ausgang Q (Ausgangsport)

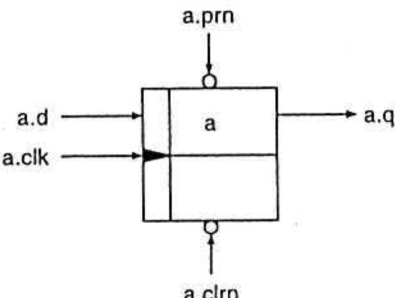

Bild 5: Die Ports eines D-Flipflops mit dem Bezeichner "a"

Da alle Flipflops nur einen Ausgang besitzen, kann man den Namen der Instanz ohne angehängtes ".q" verwenden, um den Ausgangsport der Instanz auf der rechten Seite einer Booleschen Gleichung zu bezeichnen.

Wenn ein Primitiv nur einen Primäreingang besitzt, so kann der Name der Instanz in gleicher Weise ohne den entsprechenden Portnamen verwendet werden, um den Eingangsport auf der linken Seite einer Booleschen Gleichung zu bezeichnen. Das trifft auf das D-Flipflop und das T-Flipflop in beiden Varianten zu (a = a.d bzw. a = a.t).

Wird den asynchronen Setz- und Rücksetzports kein Wert zugewiesen, so liegen die clrn- und prn-Ports standardmäßig auf VCC.

Wenn Instanzen von Flipflops und Ausgangssignale des Subdesigns mit identischen Namen versehen werden, dann werden die Ausgänge der Flipflops direkt mit den zugehörigen Ausgängen des Subdesigns verbunden ("Registered Outputs").

5.2 Register

Register werden durch Flipflop-Vektoren realisiert. Das folgende Beispiel beschreibt ein 16 Bit breites Register, das den Eingangsvektor d[] bei der positiven Taktflanke von clock nur dann übernimmt, wenn load == VCC ist.

Beispiel:

```
CONSTANT WIDTH = 16;

SUBDESIGN register
    (
    clock:          INPUT;
    load:           INPUT;
    d[WIDTH..1]:    INPUT;
    q[WIDTH..1]:    OUTPUT;
    )

VARIABLE
    reg[WIDTH..1]:  DFFE;

BEGIN
    reg[].clk = clock;
    reg[].ena = load;
    reg[].prn = VCC;
    reg[].clrn = VCC;
    reg[].d = d[];
    q[]  = reg[].q;
END;
```

Mit den Vereinfachungsregeln vom Ende des letzten Abschnitts (Abschnitt 5.1) läßt sich die Variable Section und die Logic Section des obigen Beispiels wie folgt vereinfacht darstellen:

```
VARIABLE
    q[WIDTH..1]:     DFFE;

BEGIN
    q[].clk = clock;
    q[].ena = load;
    q[] = d[];
END;
```

5.3 Zähler

Mit Hilfe von Flipflop-Vektoren und den arithmetischen Operatoren "+" und "-" lassen sich auf einfache Weise Zählerschaltungen realisieren. Das folgende Beispiel beschreibt einen 8-Bit-Aufwärts-/Abwärts-Zähler, der mit dem synchronen Takt clock getaktet und mit Hilfe der Eingangssignale up und down gesteuert wird. Dabei wird der Zähler nur dann freigegeben, wenn entweder up oder down aktiv ist.

Beispiel:

```
CONSTANT WIDTH = 8;

SUBDESIGN updown
    (
    clock:          INPUT;
    up:             INPUT;
    down:           INPUT;
    q[WIDTH..1]:    OUTPUT;
    )

VARIABLE
    q[WIDTH..1]:    DFFE;

BEGIN
    q[].clk = clock;
    q[].ena = up $ down;
    q[] = up & q[] + 1 # down & q[] - 1;
END;
```

5.4 Automaten

Neben der Deklaration von Flipflop-Vektoren ist es möglich, in der Variable Section Zustandsvektoren und Zustände von Automaten zu deklarieren. Dabei wird ein Automat wie jedes logische Element mit einem Namen versehen.

Beispiel:

```
simple: MACHINE WITH STATES (idle, wait, active);
```

Das Beispiel deklariert einen Automaten mit dem Namen "simple", der die drei Zustände "idle", "wait" und "active" besitzt. Der Compiler wählt die notwendige Anzahl von Flipflops und die Zustandscodierung automatisch.

Es ist möglich, den Zustandsvektor oder einen Teil des Zustandsvektors vorzugeben. Diese Art der Deklaration hat den Vorteil, daß die Ausgänge der Zustandsflipflops direkt verwendet werden können.

Beispiel:

```
simple: MACHINE OF BITS (q1, q0)
        WITH STATES (idle, wait, active);
```

Wenn die Anzahl der vorgegebenen Zustandsflipflops zur Unterscheidung der Zustände nicht ausreicht, so fügt der Compiler automatisch weitere Flipflops hinzu.

Den größtmöglichen Einfluß auf die Synthese eines Automaten übt man aus, indem man zusätzlich die Codierung der Zustände vorschreibt.

Beispiel:

```
simple: MACHINE OF BITS (q1, q0)
        WITH STATES
            (
            idle    = B"00",
            wait    = B"01",
            active  = B"11"
            );
```

Definiert man zwei oder mehrere Zustände mit derselben Codierung, so fügt der Compiler zur Unterscheidung dieser Zustände automatisch die notwendige Anzahl von Zustandsbits hinzu.

Jeder Automat besitzt die drei Standard-Eingabeports "clk", "reset" und "ena". Über den "clk"-Eingabeport (Clock) wird das Taktsignal für den Automaten festgelegt. Beim "reset"-Port handelt es sich um einen high-aktiven Rücksetzeingang, mit dessen Hilfe der Automat asynchron in den zuerst definierten Zustand zurückgesetzt werden kann. Der "ena"-Port (Enable) kann verwendet werden, um das Taktsignal für den Automaten zu sperren (ena == GND) oder freizugeben (ena = VCC).

Dem "clk"-Port muß in jedem Fall ein Wert zugewiesen werden. Die Beschaltung des "reset"-Ports ist optional, wenn im Ausgangszustand des Automaten alle Zustandsbits als "0" definiert sind. In allen anderen Fällen ist es notwendig, den Automaten vor dem Betrieb mit Hilfe des "reset"-Ports in den Ausgangszustand zurückzusetzen (Power-On Reset). Die Beschaltung des "ena"-Ports ist optional.

Beispiel:

```
simple.clk = clock;
simple.reset = !/reset;
```

Die Zustandsübergänge eines Automaten werden in der Logic-Section definiert. Dazu kann man einem Automaten jeden seiner Zustände zuweisen und einen Automaten mit jedem seiner Zustände vergleichen.

Beispiel:

```
IF simple == idle & !start THEN
    simple = wait;
END IF;
```

Das Beispiel definiert einen Zustandsübergang vom Zustand IDLE in den Zustand WAIT, wenn start == GND ist.

Zur Definition der Zustandsübergänge eines Automaten bieten sich CASE-Statements oder Wahrheitstabellen an (siehe Abschnitt 4.4 bzw. Abschnitt 4.2).

Das folgende Beispiel zeigt die Definition der Zustandsübergänge mit Hilfe eines CASE-Statements. In allen nicht explizit aufgeführten Fällen verbleibt der Automat in seinem jeweiligen Zustand.

Beispiel:

```
CASE simple IS
    WHEN idle =>
        IF !start THEN simple = wait; END IF;
    WHEN wait =>
        IF start THEN simple = active; END IF;
    WHEN active =>
        simple = idle;
END CASE;
```

Das nächste Beispiel zeigt die Definition der Zustandsübergänge in Form einer Wahrheitstabelle.

Beispiel:

```
TABLE

    simple, start   =>  simple;

    idle,   0       =>  wait;
    idle,   1       =>  idle;
    wait,   0       =>  wait;
    wait,   1       =>  active;
    active, X       =>  idle;

END TABLE;
```

Je nach Art der Erzeugung der Ausgangssignale werden MEALY-, MOORE- und MEDWEDEW-Automaten unterschieden.

Beim **MEALY-Automaten** hängen die Ausgangssignale vom Zustand des Automaten und von den Eingangssignalen des Automaten ab.

Beispiel: `y = simple == wait & start;`

Beim **MOORE-Automaten** hängen die Ausgangssignale nur vom Zustand des Automaten ab.

Beispiel: `y = simple == active;`

Beim **MEDWEDEW-Automaten** schließlich sind die Ausgangssignale direkt mit den Ausgängen der Zustandsflipflops verbunden. Dazu ist es notwendig, bei der Deklaration des Automaten den Zustandsvektor und die Zustandscodierung zumindest teilweise zu definieren.

Beispiel: `y = q1;`