Crashkurs VDHL

HM München, FK 06

1 Einleitung

Die Hardwarebeschreibungssprache VHDL hat sich (Stand 2002) zum weltweiten Standard für den Entwurf digitaler Schaltungen entwickelt. Kenntnisse in VHDL entsprechen in ihrer Bedeutung in der Hardwareentwicklung in etwa Kenntnissen in der Programmiersprache „C“ in der Softwareentwicklung.
In diesem Kurs werden Ihnen die grundlegenden Konzepte eines Schaltungsentwurfs in VHDL erläutert. Dabei geht es ausschließlich um die Schaltungssynthese, der gesamte Bereich der Simulation wird ausgespart. Mit den vermittelten Kenntnissen können Sie zwar weder elegante Beschreibungen für kleine Schaltungen noch parametrisierbare komplexe Entwürfe formulieren– sie können aber kleinere digitale Schaltungen nach „Schema F“ in VHDL beschreiben.

Kursziele:

- Verständnis für grundlegende Konzepte in VHDL
- Sprachreferenz (Syntax) für wichtige Befehle
- Schemata für grundlegende Aufgabenstellungen (Kombinatoriken, Automaten)

Theoretische Voraussetzungen:

- Boolesche Algebra
- Automatenentwurf (Moore-Automat)

Bitte beachten Sie, dass die Programmiersprache, die Ihnen das Nachdenken abnimmt, noch nicht erfunden worden ist. Sie werden im Verlauf des Kurses feststellen, dass Ihnen VHDL teilweise sehr einfache und eingängige Beschreibungsmöglichkeiten bietet – sofern Sie wissen, was beschrieben werden soll! Verwenden Sie also bei einem neuen Entwurf zunächst einen guten Teil Ihrer Zeit darauf, Ihr Problem günstig zu strukturieren und die einzelnen Teilaufgaben sauber zu definieren. Scheuen Sie sich nicht, zunächst ganz ohne VHDL Papier zu benutzen um darauf Blockdiagramme für die Grobstrukturierung oder Zustandsdiagramme für Automaten zu zeichnen. Erst wenn Sie sicher sind, das Problem in sinnvolle Teile zerlegt zu haben und für jeden Teil wissen, was er genau leisten soll, denken Sie an die Umsetzung in VHDL. Bei diesem Vorgehen werden Sie feststellen, dass die reine Programmierung in VHDL zum Kinderspiel wird, da die einzelnen Teile einfach aufgebaut sind (und damit mit Methoden aus „Schema F“ abgedeckt werden können) und leicht unabhängig voneinander ausgetestet werden können.
Wenn Sie stattdessen ohne klares Ziel vor Augen sofort mit einem VHDL-Programm beginnen, so werden Sie nach aller Erfahrung sehr bald den Überblick verlieren und ständig neue Variablen und Verzweigungen einführen, um „merkwürdige“ (weil nicht bedachte) Effekte in der Schaltung zu eliminieren.
Das Resultat, sofern es eines gibt, ähnelt dann einer Bauruine, die nur mit Stützen und Streben notdürftig aufrechterhalten wird und vermutlich beim ersten Unwetter (sprich Praxiseinsatz) einstürzt.

2 Signale, Typen und Vektoren

Für die folgenden Erläuterungen soll als Beispiel die Schaltung in Abbildung 1 verwendet werden.

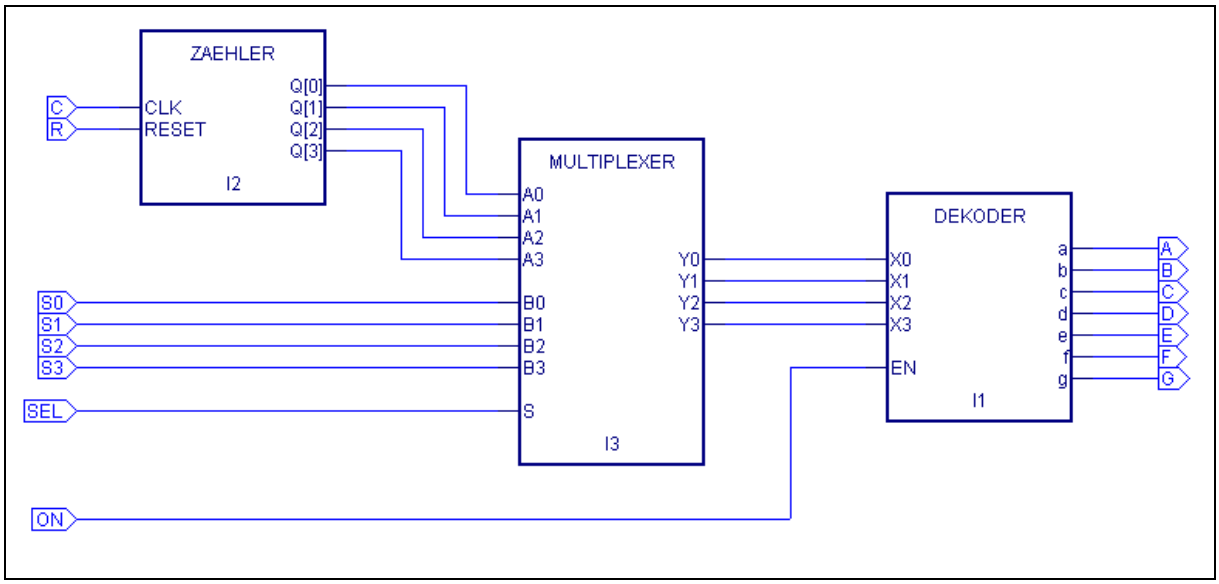

Abbildung 1: Blockschaltbild

Die Schaltung besteht aus drei **Blöcken** (Zaehler, Multiplexer, Dekoder), die untereinander durch **Signale** verbunden sind. Diese Darstellung entspricht dem Blockdiagramm auf Papier zur Aufteilung des Gesamtproblems in kleinere Teilprobleme. Die Gesamtschaltung hat die Aufgabe, entweder einen Zählerstand (0 bis 9) oder eine direkte Zahleneingabe (0 bis 9) an den Schaltern S0 bis S3 in Abhängigkeit von einem Wahlschalter SEL auf einer Siebensegmentanzeige auszugeben. Die Siebensegmentanzeige kann über einen Schalter ON ein- bzw. ausgeschaltet werden. Der Zähler kann mit zwei Tastern C und R entweder hochgezählt oder auf Null zurückgesetzt werden.
Diese Darstellung eignet sich für eine weiteren Entwurf in VHDL ideal, da in VHDL ebenfalls **Blöcke** beschrieben werden, die durch **Signale** untereinander verbunden sind.

Sofern in den Beispielen VDHL-Code verwendet wird, werden Schlüsselworte fett geschrieben oder farbig markiert.

2.1 Konventionen in VHDL

Vorab einige wichtige Regeln in VHDL, die Erläuterung der Beispiele folgt später.

2.1.1 Namensregeln

Für alle folgenden Ausführungen und natürlich den eignen Entwurf müssen Sie bei der Wahl von Bezeichnern (Namen von Signale etc.) beachten:

- Zwischen Groß- und Kleinschreibung wird in VHDL nicht unterschieden
- Namen müssen mit einem Buchstaben beginnen
- Schlüsselworte sind nicht als Namen erlaubt.

Da Sie nicht alle Schlüsselworte kennen können, sollten Sie entweder einen Editor verwenden, der VHDL-Schlüsselworte erkennt und von sich aus bereits hervorhebt oder in weiser Voraussicht verdächtige Namen wie „and“, „else“ oder „begin“ von sich aus vermeiden.

2.1.2 Kommentare

Kommentare werden durch einen doppelten Bindestrich eingeleitet und gelten bis zum Zeilenende (Abbildung 2).

```
-- Dieser Block enthält einen einfachen 2_zu_1 Multiplexer

entity multiplexer is
port
(
  S:        in bit;     -- Steuereingang zur Signalauswahl
  A, B:     in bit;     -- Dateneingänge
  Y:        out bit     -- Datenausgang
);
end;
```

Abbildung 2: Kommentare

2.1.3 Zuweisungen

Zuweisungen eines Signals an ein anders Signal bzw. eines konstanten Wertes an ein Signal werden in VHDL von rechts nach links vorgenommen und durch die Zeichenkombination <= dargestellt (Abbildung 2).

```
Y <= S;           -- S wird nach Y kopiert
Y <= A or B;      -- A und B werden mit “oder” verknüpft und das
                  -- Ergebnis wird Y zugewiesen
A => Y;           -- so rum geht es NICHT!
```

Abbildung 3: Zuweisungen

2.2 Typologie

VHDL ist eine streng typgebunde Sprache, d.h., jedem verwendete Signal bzw. Variable muss ein Typ zugewiesen werden. Eine implizite Typkonversion wie in „C“ oder einen Universaltyp wie „void“ gibt es nicht. Für den Entwurf genügt die Kenntnis der folgenden drei Typen: **boolean**, **bit** und **std_logic**.

Typ	**Wertevorrat**	**Verwendung**
boolean	true, false	logische Abfragen (if)
bit	0, 1	Entwurf
std_logic	0, 1, Z, -, U, L, H, X, W	Entwurf und Simulation

Tabelle 1: Standardtypen

Eine Deklaration eines Signals mit einem Typ sieht in VDHL wie in Abbildung 4 gezeigt aus.

Syntax:
signal <signalnamen>: typ;

Beispiele:

```
signal X0, X1, X2, X3: bit;   -- vier Signale X0 bis X3 vom Typ bit
signal EN: std_logic;         -- signal EN hat den Typ std_logic
signal ein_aus: boolean;      -- ein boolesches Signal ein_aus
```

Abbildung 4: Signaldeklarationen

Bei Verknüpfungen müssen die einzelnen Signale vom gleichen Typ sein. Das Ergebnis kann natürlich auch nur einem Signal vom gleichen Typ zugewiesen werden. Einzelne konstante Werte der Typen bit und std_logic müssen in Hochkommas eingeschlossen werden (Abbildung 5).

```
X0 <= ´0´;              -- X0 ist konstant 0
EN <= ´Z´;              -- EN wird Z zugewiesen (hochohmig)
ein_aus <= true;        -- ein_aus ist wahr
X1 <= ´Z´;              -- geht nicht, weil der Typ bit den Wert
                        -- Z nicht kennt
ein_aus <= ´1´;         -- geht auch nicht, der Typ boolean den
                        -- Wert 1 nicht kennt (1 ist nicht true)

X0 <= (´1´xor X1) and X2 or not X3;      -- geht problemlos
X0 <= X1 or EN;         -- geht nicht, da verschiedene Typen
ein_aus <=(X0=X1);      -- das geht, weil zuerst X0 mit X1 verglichen
                        -- wird und das Ergebnis eines Vergleichs wahr
                        -- oder falsch ist. Dieser boolesche Wert kann
                        -- dann auch nur einem Signal vom typ boolean
                        -- zugewiesen werden!
```

Abbildung 5: Typprüfung bei Zuweisungen

Die neun Werte des Typs std_logic sind in Tabelle 2 kurz erklärt.

Wert	Bedeutung	Verwendung
0	“starke” log. Null	Synthese, Simulation (wie 0 im Typ bit), technisch Push/Pull
1	„starke“ log. Eins	Synthese, Simulation (wie 1 im Typ bit), technisch Push/Pull
Z	hochohmig	Synthese, Simulation, technisch Tristate-Buffer, z.b. Bus
-	don’t care	Synthese, der Wert ist egal; speziell bei Wertetabellen
U	unbekannt	Simulation, der Wert ist nicht bekannt
X	Konflikt	Simulation, Erkennung von 0 gegen 1 - Treiberkonflikten
L	„schwache“ 0	Synthese, technisch open source (offener Emitter)
H	„schwache“ 1	Synthese, technisch open drain (offener Kollektor)
W	„schwaches“ X	Simulation, Erkennung von L gegen H - Treiberkonflikten

Tabelle 2: Wertevorrat in std_logic

Für den Schaltungsentwurf sind neben den Werten ´0´ und ´1´ noch speziell die Werte ´Z´ und ´-´ interessant.

2.2.1 Wert ´Z´

Der Wert ´Z´ kann einem Signal zugewiesen werden.
In der technischen Schaltung entsteht bei der **Zuweisung** ein Tristate-Buffer. Solange der Wert ´Z´ zugewiesen ist, wird der Buffer in den hochohmigen Zustand geschaltet. Wird dagegen (zu einem anderen Zeitpunkt) diesem Signal der Wert ´0´ oder ´1´ zugewiesen, dann wird der Buffer automatisch wieder auf Gegentaktbetrieb umgestellt und der zugewiesene Wert ausgegeben. Damit können also Bustreiber, die einen Tristatebetrieb erfordern, modelliert werden.
Aber Achtung: Wenn die Zielschaltung, beispielsweise ein CPLD oder FPGA, keine Tristate-Buffer bereitstellt, kann diese Methode natürlich nicht verwendet werden!
Die **Abfrage** eines Signals auf den Wert ´Z´ ist zwar in der Simulation sinnvoll, in der Synthese jedoch nicht.

2.2.2 Wert ´-´

Der Wert ´-´ kann ebenfalls einem Signal zugewiesen werden.
Bei der **Zuweisung** sucht sich das Synthesewerkzeug selbständig einen Wert (´0´ oder ´1´) aus, den das Signal dann in der technischen Schaltung führt. Welcher von beiden Werten das ist, wissen Sie nicht – sie haben ja angegeben, dass Ihnen das auch egal ist.
Bei der **Abfrage** würde man sich erhoffen, dass das Synthesewerkzeug selbständig eine Minimierung vornimmt. Das ist jedoch nicht der Fall; vielmehr müsste ein technisch nicht vorhandener Wert ´-´ verarbeitet werden können. Die Abfrage dieses Wertes ist also zwar möglich, wird aber in aller Regel weder in der Simulation noch in der Synthese zu den erwarteten Ergebnissen führen!

2.3 Vektoren

Eine wesentliche Erleichterung, die VHDL bietet, ist die Zusammenfassung von Signalen gleichen Typs zu Vektoren. Als Vektor wird dabei ein eindimensionales Array (Feld) verstanden. Der Laufindex des Vektors ist eine Integerzahl. Obwohl es mit dem Kommando **array** in VHDL prinzipiell möglich ist, selber beliebige Felder zu definieren, sind für die Standardtypen bit und std_logic entsprechende Felder als Typen bereits vordefiniert. Sie sollten diese vordefinierten Typen verwenden, weil diese Typen als Dualzahlen interpretiert werden können und Berechnungen und Vergleiche damit durchgeführt werden können.

2.3.1 Deklaration

Eine Deklaration eines Vektors sieht wie in Abbildung 6 gezeigt aus:

Syntax:
signal <signalnamen>: typ(<lower> to <upper>);
signal <signalnamen>: typ(<upper> downto <lower>);

Beispiele:

```
signal x: bit_vector(0 to 7);          -- acht Signale vom Typ bit: x(0), x(1), x(2), …,x(7)
signal a: std_logic_vector (2 to 4);   -- drei Signale vom Typ std_logic: a(2), a(3), a(4)
signal r: bit_vector(3 downto 0);      -- vier Signale vom Typ bit: r(3), r(2), r(1), r(0)
```

Abbildung 6: Syntax Vektordeklaration

Die Namen der Einzelsignale werden aus dem Signalnamen und dem Laufindex in runden Klammern gebildet. Der Signalvektor ´signal ena_2: bit_vector(3 to 5)´ besteht also aus den Einzelsignalen ena_2(3), ena2_(4) und ena_2(5). Die Einzelsignale ´e1, e2 e3' können dagegen nicht als Vektor 'e': geschrieben werden, da hier die runden Klammern fehlen!

2.3.2 Zuweisungen und Verknüpfungen

Mit Vektoren ist es besonders einfach, mit nur einer Anweisung parallel eine Operation auf alle Einzelsignale (oder eine Teilmenge davon) auszuführen Einige Beispiele sind in Tabelle 3 gezeigt.

```
signal a,b,c: bit_vector(0 to 3);  -- Deklaration dreier Vektoren

c <= a or b;                   -- bitweises oder auf alle Elemente
c <= ('1','0','0','0');        -- Zuweisung als Aggregat
c <= "1000";                   -- Zuweisung als Bitstring
c <= x"A";                     -- Zuweisung als Hexadezimalzahl
c <= a(0) & "001";             -- Zusammensetzen mit &
c(1 to 2) <= "11";             -- Auswahl eines Teilvektors im Ziel
c <= a(0 to 1) & '1' & b(0);   -- Auswahl eines Teilvektors in der Quelle
(x,y,z,w) <= c;                -- Einzelsignalen wird ein Vektor zugewiesen
```

Ziel				VHDL-Kommando
c(0)	c(1)	c(2)	c(3)	
a(0) or b(0)	a(1) or b(1)	a(2) or b(2)	a(3) or b(3)	c <= a or b;
1	0	0	0	c <= ('1', '0', '0', '0');
1	0	0	0	c <= "1000";
1	0	1	0	c <= x"A";
a(0)	0	0	1	c <= a(0) & "001";
unverändert	1	1	unverändert	c(1 to 2) <= "11";
a(0)	a(1)	1	b(0)	c <= a(0 to 1) & '1' & b(0);

Tabelle 3: Arbeiten mit Vektoren

Beachten Sie dabei besonders, dass Sie sich sowohl aus dem Ziel als auch aus der Quelle jeweils Teile herausgreifen können und dass Sie Teile mit ‚&' zu breiteren Vektoren zusammensetzen können. Sie müssen lediglich darauf achten, dass die Gesamtbreite von Ziel und Quelle bzw. Quellen jeweils gleich ist. Speziell bei der Zuweisung konstanter Werte sehen Sie, dass Sie auch eine Darstellung als Hexadezimalzahl mit x"zahl" bzw. als Oktalzahl mit o"zahl" wählen können. Die angegebene Zahl wird dabei in einen Vektor mit binären Einzelwerten umgerechnet.
Sie können bei der Zuweisung konstanter Werte den Unterstrich zur besseren Kennzeichnung von zusammengehörenden Gruppen verwenden. Er hat aber sonst keine Bedeutung, d.h. er vertritt keine Position im Vektor. Falls Sie in einem Vektor alle bisher nicht belegten Positionen (d.h. Einzelsignale) mit demselben Wert belegen wollen, dann können Sie das Kommando (others => 'wert') verwenden (Abbildung 7).

```
signal c: bit_vector(0 to 7);

c <= "01001001";        -- diese und die beiden folgenden
c <= "0100_1001";       -- Zuweisungen haben gleiche Wirkung
c <= x"49";

c <= (others => '0');   -- allen Einzelsignalen von c wird '0'
-- zugewiesen
```

Abbildung 7: Zuweisung an Vektoren

2.3.3 Positionsbestimmung im Vektor

Die Anordnung der Einzelsignale im Vektor geschieht immer von links nach rechts. Bei der Zuweisung bzw. bitweisen Verknüpfung von Vektoren werden immer die Einzelsignale an gleicher Position miteinander verknüpft. Bei der Deklaration des Vektors kann mit Hilfe der Schlüsselworte to bzw. downto festgelegt werden, ob das am weitesten links stehende Einzelsignal die niedrigste bzw. höchste Nummer erhält.
Der Unterschied in beiden Deklarationsvarianten kommt vor allem dann zum Tragen, wenn der Vektor nicht mehr als ein Feld von Einzelsignalen sondern als Dualzahl interpretiert wird.

```
signal x: std_logic_vector(0 to 3);
signal y: std_logic_vector(3 downto 0);
x <= x"3";                  -- Zuweisung erfolgt nach Position
y <= x"3";                  -- Zuweisung erfolgt nach Position
r <= x+y;                   -- Arithmetik erfolgt nach Stellenwertigkeit
```

Vektor x				**Vektor y**				**Anweisung**
x(0)	x(1)	x(2)	x(3)	y(3)	y(2)	y(1)	y(0)	
0	0	1	1					x <= x ´´3´´ ;
				0	0	1	1	y <= x ´´3´´ ;
Stellenwertigkeit				**Stellenwertigkeit**				**Interpretation**
2^0	2^1	2^2	2^3	2^3	2^2	2^1	2^0	
0	0	1	1					x = 4+8 = 12
				0	0	1	1	y = 2+1 = 3

Tabelle 4: Position und Stellenwertigkeit

Das Beispiel nach Tabelle 4 zeigt, dass die Variante downto der gewohnten Stellenwertigkeit für Dualzahlen entspricht. Wenn Sie also Standardarithmetik auf Vektoren betreiben wollen, dann sollten Sie die Vektoren mit downto deklarieren.

3 Aufbau einer Schaltungsbeschreibung

Sie kennen jetzt die Bedeutung von Typen, können Signale in VHDL deklarieren und sie entweder einzeln oder als Vektoren bearbeiten. In diesem Kapitel lernen Sie den grundsätzlichen Aufbau einer Schaltungsbeschreibung, speziell die Beschreibung eines einzelnen Blocks, kennen. Verwenden Sie am besten für jeden Block eine eigene Datei und benennen Sie die Datei so, dass Sie die darin enthaltene Schaltung auch wiedererkennen. Die Datei besteht aus drei getrennten Abschnitten

- Header mit Bibliotheks- und Packageeinbindungen
- Entity für die Schnittstellendefinition
- Architecture für die Funktionsbeschreibung.

3.1 Header

In diesem Teil legen Sie die Definitionen fest, die für die folgende Schaltungsbeschreibung gelten sollen. Da diese Definitionen nur für die unmittelbar folgende Schaltung (definiert durch ihre Schnittstelle) gilt, müssten Sie den Header vor der nächsten Schaltung in derselben Datei wiederholen. Da Sie ja in jeder Datei nur eine Schaltung beschreiben, stellt sich das Problem für Sie nicht ...
Der Header sieht so gut wie immer wie in Abbildung 8 gezeigt aus.

```
library ieee;                          -- die verwendete Bibliothek
use ieee.std_logic_1164.all;           -- die Definition der Grundtypen und Werte
use ieee.std_logic_unsigned.all;       -- die Definition einer unsigned-Arithmetik
```

Abbildung 8: Typischer Header

Sofern Sie keine Arithmetik auf Dualzahlen betreiben, können Sie die dritte Zeile weglassen. Diese Art der Arithmetik funktioniert nur auf Vektoren des Typs std_logic und interpretiert die Zahlen als vorzeichenlos. Es gibt auch andere Arithmetiken, die teilweise mehr leisten, aber auch herstellerspezifisch sind (speziell Synopsys). Die angegebene Arithmetik ist standardisiert und funktioniert mit allen standardkonformen Syntheseprogrammen.
Der Header ist in „C“ vergleichbar mit #include-Anweisungen zu Beginn eines Programms.

3.2 Die Schnittstelle – Entity

Der nächste Abschnitt ist die Definition der Schnittstelle der Schaltung. An dieser Stelle wird der Name der Blockes, der die Schaltung repräsentiert, definiert sowie sämtliche Signale, die an diesen Block angeschlossen sind. Dieser Block ist in „C“ vergleichbar mit der Deklaration einer Funktion. Auch hier wird der Funktionsname sowie die Typen und Zahl der übergebenen Parameter definiert.
An einem Block gibt es Eingangssignale, Ausgangssignale sowie Signale, die ihre Richtung wechseln können.

Signalrichtung	VHDL-Schlüsselwort	Bemerkung
Eingang	in	kann nur gelesen werden (Quelle)
Ausgang	out	kann nur geschrieben werden (Ziel)
Bidirektional	inout	kann sowohl gelesen als auch beschrieben werden

Tabelle 5: Signalrichtungen

Die formale Syntax für die einfache Deklaration einer Schnittstelle (entity) zeigt Abbildung 9.

```
entity <blockname> is
port
(
  <signalnamen>: <richtung> <typ>;  -- bei weiteren Signalen: ´;´
  <signalnamen>: <richtung> <typ>   -- beim letzten Signal KEIN ´;´
);                                  -- dafür nach der Klammer der ´;´
end;
```

Abbildung 9: Rahmen einer Entity

Mit dieser Kenntnis können Sie jetzt die Schnittstelle des Multiplexers aus Abbildung 1 wie in Abbildung 10 gezeigt deklarieren.

```
entity MULTIPLEXER is
port
(
  A0, A1, A2, A3: in bit;      -- Datenwort A
  B0, B1, B2, B3: in bit;      -- Datenwort B
  S:              in bit;      -- Steuereingang zur Auswahl (0: A, 1: B)
  Y0, Y1, Y2, Y3: out bit      -- Datenausgang
);
end;
```

Abbildung 10: Deklaration einer Entity

Bitte beachten Sie zunächst, dass der Blockname exakt dem Namen im Schaltplan entsprechen muss. Das betrifft auch die Groß-/Kleinschreibung. Innerhalb einer reinen VHDL-Beschreibung wäre dies zwar nicht erforderlich; wenn Sie aber den Block in einer anderen Umgebung (wie dem Schaltplan) wiedererkennen wollen, dann kommt es ja auch darauf an, ob das andere Programm eine Unterscheidung zwischen Groß- und Kleinschreibung vornimmt. Um mögliche Fehler auszuschließen verwenden Sie daher gleiche Schreibweisen. Dies betrifft natürlich auch die Signalnamen!
Weiterhin sehen Sie, dass Sie mehrere Signale vom gleichen Typ auch zusammen deklarieren können. Es empfiehlt sich, die Signale bei der Deklaration sinnfällig zu gruppieren.
Des weiteren fördert es die Lesbarkeit, wenn Sie Einrückungen verwenden, um den Gültigkeitsbereich eines Konstrukts (hier ‚port') zu verdeutlichen.
In diesem Beispiel können Sie die Signale nicht als Vektor deklarieren, weil ja die runden Klammern nicht im Signalnamen auftauchen. Für den Block „Zaehler" ist das aufgrund der gewählten Signalnamen aber möglich (Abbildung 11).

```
entity ZAEHLER is
port
(
  CLK:   in bit;                        -- Taktsignal
  RESET: in bit;                        -- asynchroner Reset auf Null
  Q:     out bit_vector(3 downto 0)     -- Ausgabe
);
end;
```

Abbildung 11: Entitydeklaration mit Vektor

Die in der entity deklarierten Signale sind sowohl außerhalb des Blockes sichtbar (das ist für die Verbindung zu anderen Blöcken im Schaltplan wichtig) als auch innerhalb der nun folgenden Schaltungsbeschreibung. Auch in „C" sind ja die übergebenen Parameter unmittelbar, d.h. ohne weitere Deklaration, in der Funktion verwendbar. Im Beispiel (Abbildung 1) sehen Sie, dass das Ausgangsignal Q(0), das in der entity ZAEHLER deklariert ist, mit dem Eingangsignal A0 der entity MULTIPLEXER im Schaltplan verbunden ist. Die beiden entities in den beiden VHDL-Dateien brauchen dazu nichts voneinander zu wissen.

3.3 Die Funktion - Architecture

Der letzte Abschnitt ist die Beschreibung der Funktion des eben definierten Blocks. Es kann durchaus für eine entity mehrere verschiedene Funktionsbeschreibungen geben. Für einen Zähler gäbe es die Möglichkeit, ihn als Synchron- bzw. Asynchronzähler auszuführen. Je nach Bedarf könnte dann eine der beiden Möglichkeiten für einen bestimmten Entwurf verwendet werden. Zur Identifizierung muss daher auch die Funktionsbeschreibung einen eindeutigen Namen erhalten. In diesem Kurs wird immer nur eine Beschreibung pro entity verwendet und sie heißt immer 'verhalten'.
Die formale Syntax für die einfache Deklaration einer Funktionsbeschreibung (architecture) sieht wie folgt aus:

```
architecture <beschreibungsname> of <blockname> is
 -- hier können lokale Signale deklariert werden
begin

 -- hier steht die Funktionsbeschreibung (nebenläufig)

end;
```

Abbildung 12: Rahmen einer Architecture

Damit ist der größte Teil der Funktionsbeschreibung des Multiplexers bereits verständlich (Abbildung 13).

```
architecture verhalten of MULTIPLEXER is
  signal a,b,y: bit_vector (0 to 3);  -- Hilfssignal als Vektor

begin
  a <= (a0,a1,a2,a3);          -- Zuweisung der Eingangssignale an den Vektor A
  b <= (b0,b1,b2,b3);          -- Zuweisung der Eingangssignale an den Vektor B

  y <= a when (s='0') else b; -- Auswahl durch S

  y0 <= y(0); y1 <= y(1);      -- Zuweisung des Ergebnisses an die Ausgangssignale
  y2 <= y(2); y3 <= y(3);
end verhalten;
```

Abbildung 13: Funktion des Multiplexers

Zunächst sehen Sie, dass der Blockname genau wie in der entity deklariert lautet. In der zweiten Zeile werden lokale Hilfssignale definiert. Da der Multiplexer ja vier Signalpaare jeweils völlig identisch behandelt, ist eine Vektordarstellung dieser Signale angebracht.
In den ersten beiden Zeilen nach dem 'begin' werden zunächst die Vektoren a und b mit den entsprechenden Signalen aus der Schnittstellendeklaration belegt.
Anschließend erfolgt in Abhängigkeit vom Wert von s die bedingte Zuweisung an den internen Ergebnisvektor y. Dieses Kommando wird später erklärt.
Zuletzt werden die Signale y0 bis y3 aus der Schnittstellendeklaration wieder aus dem internen Vektor y gebildet.
Natürlich hätte der Multiplexer auch gleich mit Signalnamen a(0), .. a(3), die sich in VHDL als Vektor darstellen lassen, definiert werden können. In diesem Fall hätten die Deklaration der internen Vektoren a, b und y sowie die einleitenden und abschließenden Umkopieraktionen vollständig entfallen können. Dies ist hier nur aus Demonstrationsgründen nicht erfolgt.

4 Nebenläufige und sequentielle Umgebungen

Ein wesentlicher Unterschied zwischen VHDL und einer normaler rein sequentiellen Programmiersprache wie „C“ besteht darin, dass Anweisungen in VHDL in der Regel nebenläufig, d.h. parallel abgearbeitet werden. Das liegt daran, dass aus der Beschreibung eine Schaltung erzeugt wird, deren einzelne Bestandteile (Gatter) ebenfalls ständig parallel arbeiten. Sie müssen sich bei der Beschreibung in VHDL zu jedem Zeitpunkt darüber klar sein, ob Sie sich derzeit in einer nebenläufigen oder einer sequentiellen Umgebung befinden.
Eine Umgebung ist dabei zwischen einem ’begin’ und dem zugehörigen ’end’ definiert. Die erste, von der architecture bereits eröffnete, Umgebung ist nebenläufig.
VHDL unterstützt Sie bei der Unterscheidung zwischen nebenläufigen und parallelen Umgebungen dadurch, dass die meisten Kommandos nur in einer der beiden Umgebungen erlaubt und bei illegaler Verwendung solcher Kommandos Fehlermeldungen erzeugt werden.

4.1 Nebenläufige Umgebung

In dieser Umgebung werden alle Kommandos bzw. zu Blöcken zusammengefassten Kommandos parallel ausgeführt. Die Reihenfolge, in der Anweisungen in der Umgebung stehen, hat keinen Einfluss auf das Ergebnis! Sehen Sie sich dazu den Code in Abbildung 14 an, bei der die Reihenfolge der Zeilen gegenüber Abbildung 13 umgestellt worden ist.

```
architecture verhalten of MULTIPLEXER is
  signal a,b,y: bit_vector (0 to 3);  -- Hilfssignal als Vektor

begin
  a <= (a0,a1,a2,a3);          -- Zuweisung der Eingangssignale an den Vektor A
  b <= (b0,b1,b2,b3);          -- Zuweisung der Eingangssignale an den Vektor B
  y0 <= y(0); y1 <= y(1);      -- Zuweisung des Ergebnisses an die Ausgangssignale
  y2 <= y(2); y3 <= y(3);

  y <= a when (s='0') else b; -- Auswahl durch S
end verhalten;
```

Abbildung 14: Nebenläufigkeit I (erlaubt)

Da die erste Umgebung in einer Architecture nebenläufig ist, spielt die Reihenfolge, in der Anweisungen aufgeschrieben werden, keine Rolle für die Funktion. In einer sequentiellen Programmiersprache wäre die Berechnung von y (letzte Zeile) **nach** der Verwendung in der dritten und vierten Zeile doch etwas merkwürdig.
Die Nebenläufigkeit führt allerdings auch dazu, dass die Zuweisung verschiedener Werte an ein- und dasselbe interne Signal illegal ist.

	VHDL	C
1	`signal x0,x1,a,b,y1,y2: bit;`	`int a,b,c;`
2	`c  <= a and b;`	`c = a + b;`
3	`y1  <= x0 when (c = ‘0’) else x1;`	`if (c>1) then …`
4	`c  <= a or b;`	`c = a-b;`
5	`y2  <= x0 when (c = ‘0’) else x1;`	`if (c>0) then …`

Abbildung 15: Nebenläufigkeit II (verboten)

Die Idee ist in beiden Fällen, ein Hilfssignal ’c’ zur leichteren Lesbarkeit vor der Auswertung zu berechnen (Zeilen 2 und 4). In „C“ ist das auch kein Problem, da die Anweisungen ja nacheinander abgearbeitet werden. In einer nebenläufigen VHDL-Umgebung werden aber alle Zeilen, speziell also auch die beiden Zuweisungen an dasselbe Signal ’c’, gleichzeitig ausgeführt. Damit entsteht ein Konflikt, da ’c’ nicht zur gleichen Zeit das Ergebnis der beiden Verknüpfungen von ’a’ und ’b’ sein kann.

4.2 Sequentielle Umgebung

4.2.1 Der Prozess

Die einzige sequentielle Umgebung, die in VHDL existiert, wird durch einen Prozess definiert (Abbildung 16).

```
process <empfindlichkeitsliste>
 -- hier können lokale Signale oder Variablen deklariert werden
begin
 -- hier ist eine sequentielle Umgebung
end process;
```

Abbildung 16: Prozess, Syntax

Die Anweisungen werden in der sequentiellen Umgebung nacheinander abgearbeitet. Wenn das Prozessende erreicht ist, startet die Ausführung sofort wieder am Prozessbeginn.
Eine parallele Umgebung kann auch mehrere Prozesse beinhalten. Diese Prozesse werden dann alle parallel ausgeführt, wobei die Anweisungen innerhalb eines Prozesses nacheinander abgearbeitet werden. Prozesse werden sehr häufig für die Beschreibung sequentieller Schaltungen (Automaten) verwendet. Ein Moore-Automat besteht ja definitionsgemäß aus zwei Schaltnetzen (Zustandsübergangsfunktion und Ausgabefunktion) sowie dem Zustandsspeicher. Hier bietet es sich geradezu an, die beiden Schaltnetze nebenläufig zu beschreiben und den sequentiell arbeitenden Speicher innerhalb derselben architecture mit einem Prozess zu beschreiben.
Die Empfindlichkeitsliste soll alle Signale enthalten, die innerhalb des Prozesses zu einer Änderung führen können.
Die lokal definierten Signale und Variablen sind grundsätzlich nur innerhalb des Prozesses sichtbar. Natürlich muss der Prozess auch Ausgangssignale an seine Umgebung zur weiteren Auswertung weiterleiten können. Im Beispiel des Moore-Automaten wäre das der aktuelle Zustand. Dies kann nur über Signale erfolgen, die außerhalb des Prozesses deklariert sind. Dabei muss beachtet werden, dass dann Zuweisungen an diese Signale nur im Prozess erlaubt sind. Das wird offensichtlich, wenn man sich überlegt, dass die Signalzuweisungen zwar in einem Prozess nacheinander ablaufen, diese aber mit allen nebenläufigen Anweisungen außerhalb des Prozesses konkurrieren. Betrachten Sie das Beispiel in Abbildung 17.

```
architecture verhalten of ZAEHLER is
  signal qint: std_logic_vector(3 downto 0);

begin

  process (reset, clk)                -- Prozess hängt von clk und reset ab
  begin
    if (reset='0') then qint <= x"0"; -- Asynchrones Rücksetzen auf Null
    elsif (clk='1') and clk'event     -- sonst warten auf Taktflanke
    then
      qint <= qint+1;                 -- Zählerstand hochzählen
    end if;
  end process;

  q <= qint;                          -- Nebenläufige Ausgabe

end;
```

Abbildung 17: Zähler als Prozess

Es handelt sich um eine Funktionsbeschreibung des Zählers aus Abbildung 1. Innerhalb der nebenläufigen Umgebung der Architecture laufen ein Prozess und eine Zuweisung (letzte Zeile) parallel. Sie können den Zähler als einen Moore-Automaten betrachten, hier wäre die Ausgabefunktion der Zustandsvektor ’qint’ selbst. Der Prozess selbst reagiert sowohl auf den Takt ’clk’ als auch das asynchrone Rücksetzsignal ’reset’. Daher stehen diese beiden Signale

in der Empfindlichkeitsliste. Innerhalb des Prozesses wird eine bedingte Zuweisung (Syntax wird später erklärt) ausgeführt. Falls 'reset' aktiv ist wird der Zählerstand sofort zurückgesetzt. Ansonsten wird auf eine steigende Taktflanke gewartet. Trifft diese ein, dann wird der Zählerstand um eins erhöht. Da hier das Prozessende erreicht ist, beginnt die Ausführung wieder am Prozessbeginn. Damit wird wieder das Signal 'reset' abgefragt bzw. auf die nächste steigende Taktflanke gewartet.

4.2.2 Signalzuweisung innerhalb und außerhalb des Prozesses

Betrachten Sie jetzt den Code in Abbildung 18.

```
architecture verhalten of ZAEHLER is
  signal qint: std_logic_vector(3 downto 0);

begin

  qint <= x"0" when (reset='0');      -- Asynchrones Rücksetzen auf Null

  process (clk)                       -- Prozess hängt von clk ab
  begin
    if (clk='1') and clk'event        -- warten auf Taktflanke
    then
      qint <= qint+1;                 -- Zählerstand hochzählen
    end if;
  end process;

  q <= qint;                          -- Nebenläufige Ausgabe

end;
```

Abbildung 18: Zuweisung außerhalb des Prozesses

Die beschriebene Funktion ist prinzipiell dieselbe, nur wird hier das asynchrone Rücksetzsignal 'reset' außerhalb der Prozessumgebung abgefragt. Das erscheint eigentlich als sinnvoll, denn der Rücksetzvorgang ist ja nicht an den Takt gebunden. Leider kommt es dabei aber zu zwei nebenläufigen Zuweisungen zu dem Signal 'qint', einmal außerhalb des Prozesses und einmal innerhalb des Prozesses. Damit ist diese Beschreibung nicht möglich.

4.2.3 Zeitpunkt der Signalaktualisierung

Weiterhin müssen Sie in einer sequentiellen Umgebung beachten, dass Signalzuweisungen, die innerhalb eines Prozesses vorgenommen werden (in Abbildung 17 die Zeile 'qint <= qint +1') erst mit der nächsten Taktflanke wirksam werden. Wenn Sie also ein Signal in einem Prozess abfragen, dann wird der Wert **vor** der Taktflanke verwendet und nicht der gerade eben erst neu berechnete Wert. Sie können sich auch dies erklären, wenn Sie an den Speicher des Automaten denken. Auch hier wird der eben mit der Zustandsübergangsfunktion berechnete neue Zustand erst mit der nächsten Taktflanke in den Zustandsspeicher übernommen. Bis dahin steht im Speicher der alte (also bisherige) Wert zur Auswertung zur Verfügung.

4.2.4 Unerwünschte Latches

Die letzte Falle, die in einer sequentiellen Umgebung lauert, ist die Abfrage eines Signals vor einer Zuweisung. In einer nebenläufigen Umgebung hatte das ja keine weiteren Folgen, da die Reihenfolge der Anweisungen ohnehin keine Bedeutung hat. Wenn aber in einer sequentiellen Umgebung auf ein Signal zugegriffen wird, dem bisher kein Wert zugewiesen wurde, dann muss offensichtlich auf einen älteren, d.h. gespeicherten, Wert zurückgegriffen werden. Das Synthesewerkzeug fügt deshalb ein speicherndes Element, meist ein Latch, ein. Genau das wünschen Sie aber in der Regel nicht. Achten Sie also darauf, dass Sie in einer sequentiellen Umgebung einem Signal zuerst einen Wert zuweisen, ehe Sie es verwenden.

4.2.5 Variablen im Prozess

Wie in Kapitel 4.2.3 dargelegt, werden Signale grundsätzlich erst mit dem nächsten Prozessdurchlauf aktualisiert. Da es manchmal wünschenswert ist, auf berechnete Ergebnisse sofort zugreifen zu können, können innerhalb eines Prozesses Variablen deklariert werden. Für Variablen gelten die gleichen Konventionen (Typen, Vektoren) wie für Signale, nur wird statt des Schlüsselwortes 'signal' das Schlüsselwort 'variable' verwendet und Zuweisungen werden nicht mit '<=' gekennzeichnet sondern mit ':='.

```
14  architecture verhalten of ZAEHLER is
15    signal qint: std_logic_vector(3 downto 0);
16
17  begin
18
19    process (reset, clk)                -- Prozess hängt von reset und clk ab
20      variable V: std_logic_vector(3 downto 0);
21    begin
22      if reset='0'                      -- Asynchrones Rücksetzen auf Null
23      then
24        qint <= x"0";
25        V := x"0";
26      elsif (clk='1') and clk'event     -- warten auf Taktflanke
27      then
28        qint <= qint+1;                 -- Zählerstand hochzählen
29        V := V+1;                       -- Hilfsvariable hochzählen
30      end if;
31    end process;
32
33    q <= qint;                          -- Nebenläufige Ausgabe
34
35  end;
```

Abbildung 19: Variablen in Prozessen

In Abbildung 19 ist der bereits bekannte Zähler um die Deklaration und Verwendung einer Variablen 'V' erweitert. Beachten Sie zunächst die lokale Deklaration in Zeile 20. Die Variable ist damit nur innerhalb des Prozesses sichtbar.
In Zeile 25 wird die Variable bei einem Reset ebenfalls auf Null gesetzt; beachten Sie hier die geänderte Syntax für die Zuweisung.
In Zeile 29 wird die Variable synchron mit dem Signal 'qint' um eins erhöht. Der Unterschied ist aber jetzt, dass in weiteren Anweisungen im 'then'-Zweig (Zeilen 27 bis 30) bereits der neue Wert zur Verfügung steht. Wenn also vor der Taktflanke sowohl 'qint' als auch 'V' die Hexadezimalzahl 8 enthalten haben sollten, dann enthält nach Zeile 29 das Signal 'qint' **weiterhin** die Zahl 8 während die Variable 'V' **bereits** die Zahl 9 enthält.
Erst mit der nächsten Taktflanke wird dem Signal 'qint' die neue Zahl 9 zugewiesen, so dass Signal und Variable wieder synchron laufen.

5 Anweisungen

5.1 Einfache Verknüpfungen

Signale und Variablen können miteinander verknüpft werden und das Ergebnis kann einem Signal oder einer Variablen zugewiesen werden. Die Verknüpfung und Zuweisung ist sowohl in nebenläufigen als auch sequentiellen Umgebungen zulässig. Die verfügbaren Verknüpfungen sind in Abbildung 20 gezeigt.

Schlüsselwort	Operation	Bemerkung
not	Negation	hat Vorrang vor allen anderen Operatoren
and	und	Bei „Serienschaltung“ kann die Klammerung entfallen, ’a **and** b **and** c’ ist zulässig
or	oder	
nand	negiertes und	„Serienschaltung“ ohne Klammerung ist nicht zulässig, ’a **nor** b **nor** c’ ist also verboten!
nor	negiertes oder	
xor	Antivalenz (exklusiv oder)	
xnor	Äquivalenz (neg. xor)	

Abbildung 20: Boolesche Operatoren

Grundsätzlich ist es empfehlenswert, Klammern zu setzen, um die Reihenfolge der Verknüpfungen unmittelbar einsichtig zu machen.
Damit können Sie den Multiplexer aus Abbildung 1 in VHDL wie in Abbildung 21 gezeigt beschreiben.

```
library ieee;                           -- die verwendete Bibliothek
use ieee.std_logic_1164.all;            -- die Definition der Grundtypen und Werte


entity MULTIPLEXER is
port
(
  A0, A1, A2, A3: in bit;      -- Datenwort A
  B0, B1, B2, B3: in bit;      -- Datenwort B
  S:              in bit;      -- Steuereingang zur Auswahl (0: A, 1: B)
  Y0, Y1, Y2, Y3: out bit      -- Datenausgang
);
end;


architecture verhalten of MULTIPLEXER is
begin
  y0 <= (a0 and not s) or (b0 and s);  -- Multiplexergleichung
  y1 <= (a1 and not s) or (b1 and s);
  y2 <= (a2 and not s) or (b2 and s);
  y3 <= (a3 and not s) or (b3 and s);
end verhalten;
```

Abbildung 21: Beispiel für Verknüpfung

5.2 Arithmetische Operatoren

In VHDL sind auch komplexere Operationen möglich. Für den Schaltungsentwurf kommen dabei aber meist nur Addition und Subtraktion in Frage, da Operationen wie Multiplikation oder Division in der Regel zu unverhältnismäßig aufwendigen Schaltungen führen würden. Für die Anwendung arithmetischer Operationen muss bekannt sein, auf welchem Typ die Operation arbeitet und wie eine Dualzahl interpretiert wird. Dies wird durch das verwendete Package (das ist die 'use'-Anweisung im Header) festgelegt. Leider stehen nicht in allen Entwurfswerkzeugen alle Möglichkeiten zur Verfügung.

Am sichersten fahren Sie, wenn Sie die im Standard definierte unsigned-Arithmetik auf den Typ std_logic_vector verwenden. Dabei müssen die Signale oder Variablen vom Typ std_logic_vector sein und die Interpretation des Vektors erfolgt als vorzeichenlose Dualzahl.

Falls Sie einen anderen Typ bearbeiten wollen, müssen Sie eine explizite Typkonversion vorsehen. Ebenso können Sie eine vorzeichenbehaftete Interpretation der Zahl, z.B. im Zweierkomplement, selbst ableiten.

Schlüsselwort	**Operation**	**Bemerkung**
+	Addition	
-	Subtraktion	
=	gleich	unabhängig vom Typ, geht ohne Arithmetikpackage
/=	ungleich	erzeugt, wie der Vergleich auf Gleichheit, einen booleschen Wert (Typ **boolean**) Anwendung in Abfragen wie **if/elsif/when**
<	kleiner	
<=	kleiner gleich	
>	größer	
>=	größer gleich	

Abbildung 22: Vergleiche und Arithmetik

Sehen Sie sich den Code in Abbildung 23 an. Es handelt sich um eine vollständige Beschreibung des Zählers aus Abbildung 1.

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_unsigned.all;
 4
 5  entity ZAEHLER is
 6  port
 7  (
 8    CLK:   in bit;                        -- Taktsignal
 9    RESET: in bit;                        -- asynchroner Reset auf Null
10    Q:     out bit_vector(3 downto 0)     -- Ausgabe
11  );
12  end;
13
14  architecture verhalten of ZAEHLER is
15    signal qint: std_logic_vector(3 downto 0); -- internes Hilfssignal
16
17  begin
18
19    process (reset, clk)                  -- Prozess hängt von reset und clk ab
20    begin
21      if (reset='0') then qint <= x"0";   -- Asynchrones Rücksetzen auf Null
22      elsif (clk='1') and clk'event       -- warten auf Taktflanke
23      then
24        if (qint > x"8")                  -- Abfrage auf >8, da Signal noch den
25        then                              -- vorherigen Zählerstand enthält
26          qint <= x"0";
27        else
28          qint <= qint+1;                 -- Zählerstand hochzählen
29        end if;
30      end if;
31    end process;
32
33    q <= to_bitvector(qint);              -- Nebenläufige Ausgabe mit Typkonversion
34  end;
35
```

Abbildung 23: Arithmetik und Vergleich im Zähler

Zunächst einmal wird in Zeile 3 das Package für vorzeichenlose Arithmetik auf den Typ 'std_logic_vector' eingebunden. Sie benötigen dies sowohl für den Vergleich in Zeile 24 als auch die Addition in Zeile 28.
In Zeile 15 wird ein lokales Hilfssignal von diesem Typ deklariert. Selbst wenn das Ausgabesignal (hier 'Q' in Zeile 10) bereits als 'std_logic_vector' deklariert worden wäre, hätten Sie dennoch ein internes Hilfssignal deklarieren müssen. Der Grund ist, dass das Signal 'Q' ja ein reines Ausgabesignal mit der Richtungsangabe 'out' ist . Diese Signale können aber nicht gelesen werden! Also könnten Sie auch nicht (wie in Zeile 28) auf den letzten Zählerstand zugreifen und damit letztlich nicht zählen.
Da der Zähler nur bis 9 zählen soll, wird in Zeile 24 wird auf Zählerüberlauf abgefragt. Es wäre natürlich auch möglich, auf 'qint = 9' abzufragen. In diesem Fall würde aber der Zähler bei den fehlerhaften bzw. beim Einschalten zufällig entstandenen Zuständen 10 bis 15 nicht sofort zurückgesetzt.
Die Abfrage erfolgt auf größer als 8, da das Signal 'qint' ja wie in Abschnitt 4.2.3 beschrieben um einen Takt „nachhinkt". Wenn der Zähler vor dem Takt auf 8 stand, dann wird zwar mit der Zuweisung in Zeile 28 der Zählerstand auf 9 erhöht. Für Abfragen wird dies aber erst **nach dem nächsten Takt** wirksam.
Zuletzt wird der aktuelle Zählerstand in Zeile 33 ausgegeben. Hier ist eine explizite Typkonversion erforderlich, da ja das interne Signal einen anderen Typ als das Ausgabesignal hat.

5.3 with/select

Mit dieser **nebenläufigen** Anweisung kann **ein** Auswahlsignal **mehrfach** abgefragt werden und in Abhängigkeit von dem gewählten Zweig können **einem** Signal beliebige Verknüpfungen zugewiesen werden. Die Anweisung führt in der Schaltung häufig zu einem Multiplexer, bei dem das Auswahlsignal an den Steuereingängen anliegt und die einzelnen Verknüpfungen an den Dateneingängen des Multiplexers anliegen. Das Ergebnis kann dann am Ausgang des Multiplexers abgeholt werden.

with <auswahlsignal> **select**
ergebnis <= <Verknüpfung_1> **when** <auswahlwert_1>,
<Verknüpfung_2> **when** <auswahlwert_2>,
<Verknüpfung_n> **when others**;

Abbildung 24: Syntax with/select

Die Syntax der Anweisung zeigt Abbildung 24. Zu beachten ist zunächst, dass das Auswahlsignal zwar ein Vektor sein, der aber bereits vorher aus Einzelsignalen zusammengesetzt worden sein muss. Deklarieren Sie, wenn notwendig, einfach ein lokales Hilfssignal. Natürlich muss der Typ des Ergebnisses mit dem Typ der einzelnen Verknüpfungen jeweils übereinstimmen. Ebenso muss der Typ des Auswahlsignals mit den Typen der Auswahlwerte übereinstimmen.
Wenn Sie nicht alle Möglichkeiten erschöpfend behandeln, dann sollten Sie noch eine Defaultzuweisung mit der Anweisung 'when others' vornehmen. Speziell beim Typ 'std_logic' haben Sie ja neun Werte. Bei einer erschöpfenden Aufzählung müssten Sie auch alle Werte außer '0' und '1' abfragen – was Sie natürlich nicht tun. Statt dessen verwenden Sie für die letzte Auswahl einfach 'others'. Der Code für den Dekoder aus Abbildung 1 in Abbildung 25 ist damit selbsterklärend.

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

entity DEKODER is
port
(
  X0,X1,X2,X3:   in  bit;
  EN:            in  bit;
  a,b,c,d,e,f,g: out bit
);
end;

architecture behaviour of DEKODER is
signal y: bit_vector(0 to 6);
signal sel: bit_vector(3 downto 0);

begin
  sel <= (x3,x2,x1,x0);                -- Zusammensetzung erforderlich
  with sel select                      -- Umsetzung Dualzahl in
  y <= "1111110" when x"0",            -- Siebensegmentcode
       "0110000" when x"1",
       "1101101" when x"2",
       "1111001" when x"3",
       "0110011" when x"4",
       "1011011" when x"5",
       "1011111" when x"6",
       "1110000" when x"7",
       "1111111" when x"8",
       "1111011" when x"9",
       "0000000" when others;          -- hier bleibt die Anzeige dunkel

  a <= y(0) and en; b <= y(1) and en;  -- Enable wirkt auf alle Ausgänge
  c <= y(2) and en; d <= y(3) and en;
  e <= y(4) and en; f <= y(5) and en;
  g <= y(6) and en;

end;
```

Abbildung 25: Dekoder mit with/select

5.4 when/else

Auch diese Anweisung ist wie with/select nur in **nebenläufigen** Umgebungen zulässig. Mit dieser Anweisung können jedoch wechselnde Signale auf zutreffende Bedingungen geprüft werden und in Abhängigkeit davon einem Ergebnissignal verschiedene Verknüpfungen zugewiesen werden. Außerdem ist es möglich, dass sich die Bedingungen überlappen – in diesem Fall wird die erste zutreffende Bedingung ausgewählt. Dies war bei with/select nicht möglich, da dort alle Verzweigungen disjunkt sein mussten.

```
<ergebnis>  <=     < Verknüpfung_1> when <Bedingung_1>
                   else  < Verknüpfung_2> when <Bedingung_2>
                   else  <Verknüpfung_n>;
```

Abbildung 26: Syntax when/else

Die in Abbildung 26 gezeigte Syntax zeigt, dass hier beliebige Bedingungen verwendet werden können. Der letzte Zweig ohne Bedingung entspricht der Defaultzuweisung 'when others'.

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

entity DEKODER is
port
(
  X0,X1,X2,X3:   in  bit;
  EN:            in  bit;
  a,b,c,d,e,f,g: out bit
);
end;

architecture behaviour of DEKODER is
signal y: bit_vector(0 to 6);
signal sel: bit_vector(3 downto 0);

begin

  sel <= "1111" when (EN='0')          -- Enable hier am Eingang
         else (x3,x2,x1,x0);

  with sel select                      -- Umsetzung Dualzahl in
  y <= "1111110" when x"0",            -- Siebensegmentcode
       "0110000" when x"1",
       "1101101" when x"2",
       "1111001" when x"3",
       "0110011" when x"4",
       "1011011" when x"5",
       "1011111" when x"6",
       "1110000" when x"7",
       "1111111" when x"8",
       "1111011" when x"9",
       "0000000" when others;          -- hier bleibt die Anzeige dunkel

  a <= y(0); b <= y(1); c <= y(2); d <= y(3);
  e <= y(4); f <= y(5); g <= y(6);

end;
```

Abbildung 27: Beispiel when/else

In diesem Beispiel wird die Enable-Funktion dadurch realisiert, dass in einer when/else-Anweisung das interne Auswahlsignal 'sel' entweder mit dem nicht vorkommenden Wert '1111'" belegt wird oder eben mit dem anzuzeigenden Wert. Die nachfolgende Codeumsetzung sorgt dafür, dass bei der Auswahl '1111' die Anzeige dunkel bleibt.

5.5 if/then

Diese Anweisung entspricht der when/else-Anweisung, ist jedoch nur in **sequentiellen** Umgebung zulässig. Mit dieser Anweisungen können beliebige Blöcke von sequentiellen Anweisungen in Abhängigkeit von beliebigen Bedingungen ausgeführt werden.
Die Syntax zeigt Abbildung 28.

```
if      <Bedingung_1>     then  <Sequentielle Anweisungen 1>;
elsif   <Bedingung_2>     then  <Sequentielle Anweisungen 2>;
elsif   <Bedingung_2>     then  <Sequentielle Anweisungen 3>;
else                            <Sequentielle Anweisungen n>;
end if;
```

Abbildung 28: Syntax if/then

Beachten Sie die unterschiedliche Schreibweise von ’elsif’ und ’end if’. Die Bedingungen müssen vom Typ boolean sein. Das können entweder Signale/Variablen von diesem Typ sein oder aber Vergleiche zweier Ausdrücke gleichen Typs.
Da in jedem Zweig ein ganzer Block von Anweisungen stehen kann, können if/then-Anweisungen beliebig tief geschachtelt werden.

```
process (reset, clk)                    -- Prozess hängt von reset und clk ab
begin
  if (reset='0') then qint <= x"0";     -- Asynchrones Rücksetzen auf Null
  elsif (clk='1') and clk'event         -- warten auf Taktflanke
  then
    if (qint > x"8")                    -- Abfrage auf >8, da Signal noch den
    then                                -- vorherigen Zählerstand enthält
      qint <= x"0";
    else
      qint <= qint+1;                   -- Zählerstand hochzählen
    end if;
  end if;
end process;
```

Abbildung 29: Beispiel if/then

Der Code in Abbildung 29 ist ein Ausschnitt aus dem Code für den Zähler (Abbildung 23). Dabei wird eine geschachtelte if/then-Konstruktion verwendet, denn in dem then-Zweig der ersten elsif-Bedingung ist eine weitere if/then-Anweisung enthalten.
An diesem Beispiel wird auch deutlich, dass mit der if/then-Anweisung eine Priorisierung bei mehreren gleichzeitig wahren Bedingungen erreicht wird.
Angenommen, eine steigende Taktflanke trifft ein während das Signal ’reset’ den Wert ’0’ hat. Dann wird dennoch nur der then-Zweig der ersten if/then-Anweisung ausgeführt, d.h. der Zähler bleibt zurückgesetzt. So soll es bei einem asynchronen Reset auch sein.

5.6 case/is

Diese Anweisung entspricht der with/select-Anweisung, ist jedoch nur in **sequentiellen** Umgebung zulässig. Mit dieser Anweisungen können ebenfalls beliebige Blöcke von sequentiellen Anweisungen in Abhängigkeit von **einem** Testsignal ausgewählt werden.

```
case   <testsignal> is
       when  <Wert_1>    => <Sequentielle Anweisungen 1>;
       when  <Wert_2>    => <Sequentielle Anweisungen 2>;
       when  <Wert_2>    => <Sequentielle Anweisungen 3>;
       when others       => <Sequentielle Anweisungen n>;
end case;
```

Abbildung 30: Syntax case/is

Das zu testende Signal und die Testwerte müssen natürlich vom gleichen Typ sein. Wie bei der with/select-Anweisung sollten Sie eine Defaultzuweisung in einem Zweig 'when others' vorsehen. Dies ist hier besonders wichtig, weil die case/is-Anweisung fast immer bei der Beschreibung einer Zustandsübergangsfunktion benutzt wird. Ist diese Funktion aber nicht vollständig definiert, dann besteht die Gefahr, dass der Automat in einen ungewollten Zustand übergeht und evtl. dort bleibt.
Da ganze Blöcke aufgrund einer Bedingung ausgeführt werden können, sind auch hier beliebig tiefe Schachtelungen möglich.
Ein ausführliches Beispiel zur case/is-Anweisung folgt in der Automatenbeschreibung.

```
architecture verhalten of MULTIPLEXER is
  signal a,b,y: bit_vector(0 to 3);   -- Hilfsvektoren

begin
  a <= (a0,a1,a2,a3);                 -- Kopieraktionen
  b <= (b0,b1,b2,b3);
  y0 <= y(0); y1 <= y(1); y2 <= y(2); y3 <= y(3);

  process (s,a,b)                     -- Empfindlichkeitsliste
  begin
    case s is
      when '0' => y <= a;
      when '1' => y <= b;
    end case;
  end process;

end;
```

Abbildung 31: Beispiel case/is

Im Code aus Abbildung 31 für den Multiplexer musste zunächst mit dem Prozess eine sequentielle Umgebung geschaffen werden, da case/is in nebenläufigen Umgebungen unzulässig ist. Als Besonderheit kann hier die Defaultzuweisung entfallen, weil das Testsignal vom Typ 'bit' ist und damit tatsächlich nur die Werte '0' und x'1' annehmen kann. Diese beiden Möglichkeiten werden auch explizit abgefragt, so dass keine weitere Möglichkeit mehr übrig bleibt.

6 Automatenbeschreibung

Automaten können in VHDL auf vielfältige Weise beschrieben werden. Die Art der Beschreibung hat teilweise Einfluss auf das Syntheseergebnis; manche Entwurfsprogramme können auch nicht alle möglichen Beschreibungsformen umsetzen. Aus diesem Grund wird hier nur eine einzige Möglichkeit zur Beschreibung eines Moore-Automaten vorgestellt. Die Beschreibung kann von allen bekannten Entwurfswerkzeugen umgesetzt werden und ist gleichzeitig übersichtlich. Die Erweiterung auf den Mealy-Automaten ist trivial.
Ein Moore-Automat enthält drei voneinander getrennte Blöcke:

- Zustandsspeicher
- Zustandsübergangsfunktion
- Ausgabefunktion

Im diesem „Schema F"-Entwurf werden diese drei Blöcke in einer nebenläufigen Umgebung als zwei parallel ausgeführte Prozesse modelliert. Der erste Prozess beschreibt lediglich den Zustandsspeicher, während der zweite Prozess sowohl die Zustandsübergangsfunktion als auch die Ausgabefunktion beschreibt.
Für die Zustandsübergangsfunktion und die Ausgabefunktion wären auch nebenläufige Anweisungen denkbar. Die Darstellung als Prozess hat aber den Vorteil, dass die case/is-Anweisung verwendet werden kann, die für jede Bedingung wieder einen ganzen Block mit Anweisungen ermöglicht.
Das Grundgerüst hat in diesem Schema die in Abbildung 32 gezeigte Struktur:

```
architecture verhalten of automat is
  signal z_alt, z_neu: zustaende;                  -- symbolische zustaende
begin

  process (clk, init)                              -- zustandsspeicher
  begin
    if (init) then zustand <= resetzustand;        -- hier die asynchrone initialisierung
    elsif (clk = '1') and clk'event                -- warten auf steigende taktflanke
    then
      z_neu <= z_alt;                              -- zustandsaktualisierung
    end if;
  end process;

  process (z_alt, eingaenge)                       -- ausgabe-/uebergangsfunktion
  begin
    case z_alt is
      when z0 => z_neu <= f_u(z0,eingaenge);       -- uebergaenge vom zustand z0
                 aus   <= f_a(z0, eingaenge);      -- ausgabe im zustand z_0
      when z1 => z_neu <= f_u(z1,eingaenge);       -- uebergaenge vom zustand z1
                 aus   <= f_a(z1, eingaenge);      -- ausgabe im zustand z_1
      ....
    end case;
  end process;

end architecture;
```

Abbildung 32: Schema für den Moore-Automat

6.1 Spezielle Konstrukte

Für eine gut lesbare und erfolgreich synthetisierbare Beschreibung nach diesem Schema benötigen Sie noch zwei bisher nicht vermittelte Kenntnisse.

6.1.1 Symbolische Zustandskodierung

Bei Ihrem Papierautomaten haben Sie sinnvollerweise symbolische Zustandsnamen wie ein, aus, vorwaerts und rueckwaerts gewählt. Dafür gibt es in VHDL keinen geeigneten Typ, obwohl es sicherlich günstig wäre, diese Namen zunächst beizubehalten. Mit der in Abbildung 33 gezeigten Anweisungsfolge können Sie sich aber in VHDL einen eigenen Aufzählungstyp definieren.

Syntax:
type <typname> **is** (wert1, wert2, ..., wert_n)**;**

Beispiel:
type gaenge **is** (leerlauf, vorwaerts, rueckwaerts);
signal z_alt, z_neu: gaenge;

Abbildung 33: Syntax Definition eines Aufzählungstyps

Sie haben damit zwei Signale 'z_alt' und 'z_neu' zur Verfügung, die jeweils die Werte 'leerlauf', 'vorwaerts' und 'rueckwaerts' annehmen können. Sie können mit diesen Werten zwar keine Verknüpfungen bilden – Sie können aber die symbolischen Namen einem geeigneten Signal zuweisen und ein Signal mit einem Wert auf Gleichheit prüfen. Mehr benötigen Sie in der Automatenbeschreibung auch nicht!
Wenn Sie den fertig beschriebenen Automaten synthetisieren wollen, dann müssen Sie natürlich den symbolischen Namen Dualzahlen zuordnen. Wenn Sie nichts weiter tun, dann wird das Entwurfswerkzeug automatisch eine Zuordnung durchführen. Bei manchen Werkzeugen können Sie auch noch als Option auswählen, nach welchen Verfahren das geschehen soll. Übliche Möglichkeiten sind binär, one-hot und gray.
Sie können aber auch selbst eine Kodierung vornehmen, und zwar ohne dass Sie in der gesamten VHDL-Beschreibung die symbolischen Namen durch Dualzahlen ersetzen. Erweitern Sie die Definition aus Abbildung 33 um eine Zuordnung nach Abbildung 34.

Syntax:
type <typname> **is** (wert1, wert2, ..., wert_n)**;**
attribute enum_encoding: string**;**
attribute enum_encoding of <typname> **: type is "**zahl_fuer_wert1,zahl_fuer_wert2,…";

Beispiel:
type gaenge **is** (leerlauf, vorwaerts, rueckwaerts);
attribute enum_encoding: string;
attribute enum_encoding of gaenge: type is "00 11 10";
signal z_alt, z_neu: gaenge;

Abbildung 34: Beispiel Zustandskodierung

In diesem Beispiel haben Sie festgelegt, dass der Zustand 'leerlauf' mit "00" kodiert wird, der Zustand 'vorwaerts' mit "11" und der Zustand 'rueckwaerts' mit '10'.

6.1.2 Prozessstruktur

Die meisten Synthesewerkzeuge können eine sequentielle Schaltung nur dann fehlerfrei abbilden, wenn sie in einer „Normaldarstellung“ beschrieben sind. Alle bekannten Synthesewerkzeuge können Beschreibung nach Abbildung 35 erkennen und erfolgreich synthetisieren.

```
process (<empfindlichkeitsliste>)
-- hier lokale Signale und Variablen deklarieren
begin
  if (<asynchrone bedingung>)
    then
      <z_neu> <= initialzustand;
    elsif <takt> = 'wert' and <takt>'event
      then
        -- hier beginnt die sequentielle umgebung
  end if;
end process;
```

Abbildung 35: Synthetisierbarer taktsynchroner Prozess

Beachten Sie zunächst, dass Sie alle notwendigen Signale auch wirklich in die Empfindlichkeitsliste des Prozesses eintragen.
Falls Ihr Automat mit einem asynchronen Signal in einen initialen Zustand gebracht werden kann (Beispiel: Reset), dann beginnen Sie den Prozess sofort mit einer entsprechenden if/then-Anweisung.
In den elsif-Bedingung dieser Anweisung (bzw. die if-Bedingung, wenn Sie keine asynchrone Initialisierung benötigen) schreiben Sie für einen Übergang bei

- **steigender** Taktflanke: (takt = '1') and takt'event
- **fallender** Taktflanke: (takt = '0') and takt'event

Versuchen Sie nicht, einen Automaten zu kreieren, der auf beide Taktflanken reagiert.
Versuchen Sie nicht, die Taktflankenabfrage an eine andere Stelle zu verschieben.
Versuchen Sie nicht, die asynchrone Bedingungsabfrage an eine andere Stelle zu verschieben.
Versuchen Sie nicht, mehrere Taktflankenabfragen einzubauen.

Sollten Sie in einer nicht von Ihnen verfassten Beschreibung eine 'wait'-Anweisung am Ende des Prozesses sehen, dann denken Sie sich statt dieser Anweisung die entsprechende Taktflankenabfrage an den Beginn des Prozesses (wie in Abbildung 35). Dies ist eine weitere übliche Methode, die jedoch (Stand 2002) an Bedeutung verliert.

6.2 Beispiel Frag-O-Mat

In diesem Kapitel wird mit den bisherigen Kenntnissen ein Moore-Automat in VHDL entworfen. Die bisherige Erfahrung hat gezeigt, dass sich professorales Frageverhalten (leider) einigermaßen zuverlässig mit einem Automaten namens „Frag-O-Mat“ nachbilden lässt. Es seien in der Vorlesung noch ein harter Kern von vier Hörern A,E,L und S übriggeblieben. Diese werden normalerweise der Reihe nach befragt. Kann die Frage beantwortet werden (Bedingung w=weiß es), dann kommt als nächstes der alphabetisch folgende Kandidat dran. Dies ist im Zustandsübergangsgraph (Abbildung 36) die rechte Schleife A -> E -> L -> S -> A.

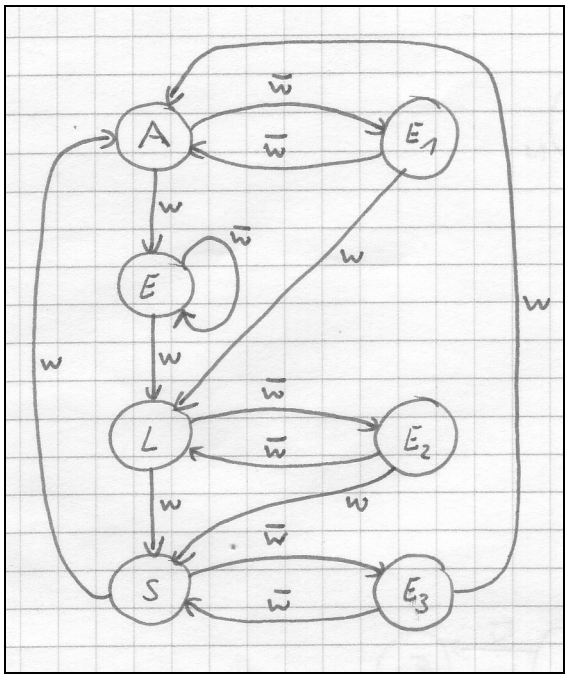

Abbildung 36: Zustandsübergänge „Frag-O-Mat“

Wenn die Frage nicht beantwortet werden kann (w_bar = weiß es nicht), dann kommt zunächst E an die Reihe, da hier die größte Wahrscheinlichkeit besteht, eine zufriedenstellende Antwort zu erhalten. Im Zustandsübergangsgraph sind dies die Übergänge nach rechts in die Zustände E1, E2 und E3 sowie der Sonderfall E -> E.
Weiß es E auch nicht, dann war die Frage zu schwer oder schlecht formuliert. In diesem Fall erhält der zuerst befragte Hörer eine weitere Chance mit einer besser angepassten Frage (Übergänge von E1, E2 und E3 nach links sowie der Sonderfall E -> E).
Konnte E dagegen die Frage beantworten, dann kommt als nächstes der ohnehin als „normaler“ Nachfolger vorgesehene Hörer an die Reihe (Übergänge von E1 und E2 nach links unten bzw. von E3 nach A).
Eine kurze Prüfung ergibt, dass der Graph sowohl vollständig als auch widerspruchsfrei ist.
Die Ausgabefunktion ist besonders einfach, es wird einfach der einem Zustand zugeordnete Hörer angezeigt (Abbildung 37)

Zustand	A	L	S	E	E1	E2	E3
Ausgabe	A	L	S	E			

Abbildung 37: Ausgabefunktion Frag-O-Mat

Der Automat ist, zumindest nach außen, sehr genügsam (Abbildung 38)

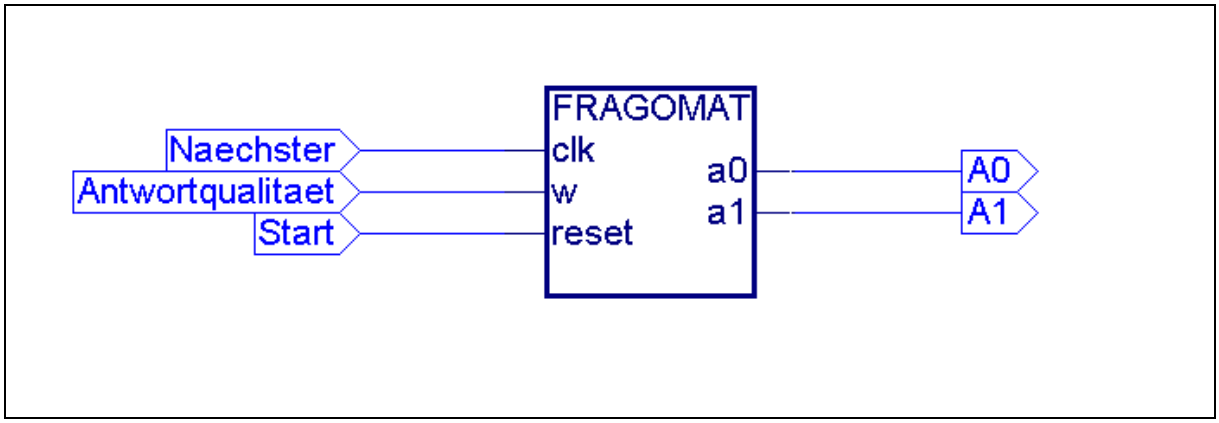

Abbildung 38: Blocksymbol des Frag-O-Mat

Die ersten beiden Abschnitte der dazupassenden VHDL-Beschreibung sind daher kaum mehr als eine Fingerübung (Abbildung 39).

```
 1
 2  library ieee;
 3  use ieee.std_logic_1164.all;
 4
 5
 6  entity fragomat is
 7  port
 8  (
 9    clk, w, reset: in std_logic;
10    a0, a1:        out std_logic
11  );
12  end;
13
14
```

Abbildung 39: Header und Entity des Frag-O-Mat

```
15  architecture verhalten of fragomat is
16   type hoerer is (Z_A,Z_E,Z_L,Z_S,Z_E1,Z_E2,Z_E3);
17   attribute enum_encoding: string;
18   attribute enum_encoding of hoerer: type is "000 100 001 010 101 110 111";
19   signal zalt, zneu: hoerer;
20   signal ausgabe: std_logic_vector(0 to 1);
21
22  begin
23
24   process (clk, reset)
25   begin
26     if (reset = '1')
27       then zalt <= Z_A;
28       elsif (clk = '1') and clk'event
29         then zalt <= zneu;
30     end if;
31    end process;
32
```

Abbildung 40: Deklarationen und Speicherprozess

In Zeile 15 beginnt die Funktionsbeschreibung. Zunächst wird ein eigener Aufzählungstyp 'hoerer' definiert, damit die symbolischen Zustandsnamen weiterverwendet werden können. Außerdem wird - nur zu Demonstrationszwecken - eine Zustandskodierung vorgegeben.
Der erste Prozess für den Zustandsspeicher beginnt in Zeile 24. Bei einem Reset wird asynchron der Zustand Z_A eingestellt, ansonsten mit der steigenden Flanke am Takt der neue Zustand übernommen.
Die Beschreibung hält sich genau an das in Abbildung 35 angegebene Schema!

```
33
34   process (w, zalt)
35     variable x: boolean;
36   begin
37     if (w = '1')
38       then x := true;
39       else x := false;
40     end if;
41
42     case zalt is
43       when Z_A  => if x
44                      then zneu <= Z_E;
45                      else zneu <= Z_E1;
46                    end if;
47                    ausgabe <= "00";
48       when Z_E  => if x
49                      then zneu <= Z_L;
50                      else zneu <= Z_E;
51                    end if;
52                    ausgabe <= "11";
53       when Z_L  => if x
54                      then zneu <= Z_S;
55                      else zneu <= Z_E2;
56                    end if;
57                    ausgabe <= "01";
58       when Z_S  => if x
59                      then zneu <= Z_A;
60                      else zneu <= Z_E3;
61                    end if;
62                    ausgabe <= "10";
63       when Z_E1 => if x
64                      then zneu <= Z_L;
65                      else zneu <= Z_A;
66                    end if;
67                    ausgabe <= "11";
68       when Z_E2 => if x
69                      then zneu <= Z_S;
70                      else zneu <= Z_L;
71                    end if;
72                    ausgabe <= "11";
73       when Z_E3 => if x
74                      then zneu <= Z_A;
75                      else zneu <= Z_S;
76                    end if;
77                    ausgabe <= "11";
78     end case;
79   end process;
```

Abbildung 41: Zustandsübergangsfunktion und Ausgabefunktion

Die Beschreibung wird jetzt mit dem zweiten Prozess für die beiden Schaltnetze fortgesetzt. Beachten Sie zunächst, dass die Empfindlichkeitsliste dieses Prozesses ganz anders als die des ersten ist: Der Prozess hängt vom Zustand und den Eingangssignalen ab, nicht aber vom Takt oder dem Reset.

Nur zu Demonstrationszwecken wird ein lokales boolesches Signal 'x' deklariert, das mit der if/then-Anweisung in den Zeilen 37 bis 40 mit einem später direkt abfragbaren Wert für die Übergangsbedingungen „w" und „w_bar" belegt wird.

In den Zeilen 42 bis 78 sind nun in einer einzigen case/is-Anweisung alle notwendigen Übergänge und Ausgaben beschrieben. Beachten Sie, dass zunächst mit der case/is-Anweisung die einzelnen Zustände unterschieden werden können. Die Zeilen 53 bis 57 gelten zum Beispiel, wenn sich der Automat gerade im Zustand Z_L befindet. Die if/then-Anweisung für diesen Fall (Zeilen 53 bis 56) berechnet die beiden hier möglichen Folgezustände. Dabei wird schlicht aus dem Papierentwurf abgeschrieben!

Ebenso einfach erfolgt in jedem Zustand die direkte Zuweisung an den Ausgabevektor in Zeile 57.

Die Beschreibung endet mit dem Umkopieren an die Ausgangssignale (Abbildung 42).

```
80
81
82   a0 <= ausgabe(0);
83   a1 <= ausgabe(1);
84 end;
85
```

Abbildung 42: Ausgabe des internen Ausgabesignals