Kapitel 3

Verilog Syntax Teil I

Die Hardware-Beschreibungssprache (Hardware Description Language - HDL) Verilog entspricht neben VHDL (Virtual Hardware Description Language) und eventuell AHDL dem aktuellen Standard. Für Einsteiger ist Verilog im Vergleich mit VHDL übersichtlicher und darum einfacher zu erlernen. Grundsätzlich ist mit Verilog und VHDL die gleiche digitale Schaltung in FPGAs realisierbar.

Eine HDL beschreibt die Anordnung von Komponenten und Verbindungsleitungen, wir sollten daher zunächst wissen welche Art von Signal überhaupt übertragen werden kann. Folgende Tabelle zeigt die möglichen Signalzustände einer 1-Bit Verbindung (wire) in Verilog:

0	eine logische Null also Zustand Low
1	eine logische Eins entspricht High
X	Zustand des Signals ist im Simulator undefiniert
Z	Signal ist hochohmig (z.B. für Tri-States)

Der Zustand 'X' tritt nur in der Simulation auf, falls Ausgänge noch ungewisse bzw. nicht initialisierte Zustände haben.
Um die Syntax näher zu bringen versuchen wir eine Auswahl an Quellcodes zu analysieren und diese weitgehend zu verstehen. Hierbei soll noch einmal daran erinnert werden, dass wir letzlich eine reale digitale Schaltung erstellen. Eine Realisierung des Projekts mit echten ICs bzw. auf unterster Ebene mit echten Transistor-Schaltungen wäre nach wie vor möglich - kostete uns jedoch zuviel Zeit, besonders wenn es um einfache Schaltungen mit vielen Bauteilen geht (Und es kommen sehr schnell viele Bauteile zusammen). Wie schon erwähnt ersetzt der FPGA die meisten Logik-Funktionen bevorzugt durch LUTs. Die Geschwindigkeit der kombinatorischen Logik wird dadurch verbessert. Der RTL Viewer stellt die

Schaltung aber mithilfe der alternativ zu verwendenden Gates dar, um eine bessere Übersicht zu geben.

3.1 Module

Beginnen wir nun mit dem Grundaufbau der Module in Verilog. Module können fertig zusammengebaute Schaltungen aus Komponenten wie ICs sein oder auch nur einzelne ICs selbst, die auf Gates basieren. Wir können also in einzelnen Modulen Bereiche einer digitalen Schaltung unterbringen, und diese 'Submodule' in einem weiteren Modul zusammenschalten - analog zu ICs, die auf dem Experimentierboard verdrahtet werden.

Module sind im Wesentlichen wie folgt aufgebaut:

syntax_test.v

```
1
2           // Zwei Slashes eröffnen eine Kommentar-Zeile
3           // Leerzeichen und Leerzeilen dienen allein der Übersichtlichkeit
4
5           // Der Name dieses Moduls ist 'syntax_test'
6           // Der Dateiname muss deshalb 'syntax_test.v' lauten
7
8           // In den Klammern hinter dem Modulnamen werden alle im
9           // Modul vorhandenen Ein- und Ausgänge als Argumente definiert
10
11  module syntax_test (clock, eingang_A, eingang_B, ausgang_X, ausgang_Y);
12
13          // Nun folgt die explizite Deklaration der Ein- und Ausgänge:
14
15  input clock;                              // 'clock' ist ein 1-Bit Eingang
16  input eingang_A;            // 'eingang_A' ist ein 1-Bit Eingang
17  input [3:0] eingang_B;  // 'eingang_B' ist ein 4-Bit Input
18
19  output [1:0] ausgang_Y;  // hier ein 2-Bit Ausgang
20
21          // Oft ist es nötig Outputs mittels 'reg' als Variable zu deklarieren
22          // weil das Signal innerhalb des Moduls als Variable verwendet wird.
23
24  output reg [1:0] ausgang_X;  // das 'reg' folgt direkt hinter 'output'
25
26          // Nur das Schlüsselwort 'reg' allein erzeugt eine Variable:
27
28  reg aktiv;          // hier eine 1-Bit Variable 'aktiv'
```

```
29
30  reg [7:0] data;  // eine weitere 8-Bit Variable 'data'
31
32          // Mit dem Schlüsselwort 'wire' wird eine Verbindung definiert
33          // 'wire' dienen zur Verbindung FPGA interner Komponenten
34
35  wire [7:0] kabel_A;  // ein 8-Bit Datenbus
36  wire kabel_B;                // hier eine 1-Bit Verbindung
37
38  wire [7:0] b1,b2,b3,b4;  // Vier 8-Bit Verbindungen
39
40          // Mit geschweiften Klammern können mehrere 'wire' zu
41          // einem gemeinsamen Bus zusammengefasst werden:
42
43  wire [31:0] databus = {b1,b2,b3,b4};
44
45          // Mithilfe des 'initial' Befehls können Variablen Startwerte erhalten:
46
47  initial
48  begin
49          ausgang_X[0] = 1;  // In eckigen Klammern steht der Index des Bits
50          ausgang_X[1] = 0;  // Das nullte Element ist das Bit ganz rechts (LSB)
51  end
52
53          // Dem 'always' Block schenken wir noch besondere Aufmerksamkeit:
54
55  always
56  begin
57          ausgang_X = eingang_A & aktiv; // Hier wird ein AND Gate erzeugt
58  end
59
60
61  endmodule  // Hiermit ist das Modul 'syntax_test' zu Ende
62
```

syntax_test.v

Dies ist im Wesentlichen die Struktur eines jeden Moduls. Weiterhin besteht die Möglichkeit Sub-Module zu erstellen und diese im sogenannten Top-Level-Modul zu verwenden. Zu Submodulen folgt weiter unten ein kurzer Abschnitt. Submodule können schon in einfachen Addier-Schaltungen sinnvoll sein, wie wir noch sehen werden.

3.2 Variablen

Verilog bietet die Möglichkeit unterschiedliche Typen von Variablen und auch Konstanten zu definieren (Wie wir solche Variablen als echte Hardware verstehen können, wird im Laufe der nächsten Verilog Beispiele klarer). Weiterhin gibt es mehrere Möglichkeiten Variablen mit Werten zu versehen. Die folgenden Code-Abschnitte zeigen die meisten dieser Möglichkeiten:

```
1
2           // Das Schlüsselwort 'reg' haben wir schon kennengelernt,
3           // im Folgenden noch einmal die Zuweisung von Start-Werten:
4
5           // folgende Anweisungen bedeuten: 'A' ist eine 4-Bit Variable,
6           // zu Beginn bekommt 'A' vier Bits zugewiesen: 4'
7           // diese werden als einzelne Bits geschrieben: b 0101
8
9   reg [3:0] A;
10
11  initial A = 4'b0101 ;
12
13          // Um diese Schreibweise plausibel zu machen, noch ein Beispiel
14          // 'B' soll eine 8-Bit Variable sein
15          // Zu Beginn bekommt 'B' 8 Bits zugewiesen: 8'
16          // diese werden aber als Hexadezimal-Werte geschrieben: h FF
17          // Der Wert FF = 15*16+15 = 255, binär also '11111111'
18
19  reg [7:0] B;
20
21  initial B = 8'hFF ;
22
23          // Im Folgenden ein 2-Bit Signal, das zu Beginn hochohmig gesetzt wird
24          // Dies geschieht über die Zuweisung des Wertes 'z' (High Impedance)
25
26  reg [1:0] status;
27
28  initial status = 2'bZZ;
29
30          // Mittels 'parameter' wird eine Konstante erzeugt
31          // Diese ist hier in Dezimalschreibweise geschrieben
32          // Die Bitbreite wird automatisch entsprechend zugewiesen
33
34  parameter zahl = 314;  // Das wären also relevante 9-Bit
35
36          // Ein Feature: Unterstriche in Zahlenangaben werden ignoriert
37
38  parameter langezahl = 16'b1000_0100_1000_0001; // zur besseren Übersicht
39
```

```
40          // Auch bei Konstanten kann 'X' und 'Z' verwendet werden:
41
42  parameter vergleich = 10'bZZZZ_XXXX_01;
43
44          // Der Typ 'integer' erzeugt eine maximal 32-Bit lange Zahl:
45
46  integer i;  // Integerzahlen werden nur FPGA intern verwendet
47
48          // Weiterhin soll der ASCII-Code erwähnt werden,
49          // Pro Symbol bzw. Zeichen werden Sieben Bits beansprucht:
50
51  parameter wort = 'ABC'  // Dies sind also 21-Bit
52
53          // Der folgende Abschnitt befasst sich mit der ASCII Codierung
54
55  endmodule
56
```

Beim ASCII-Code handelt es sich um eine international abgeglichene Tabelle mit Sonderzeichen-, Ziffern- und Buchstaben-Codes. Pro Symbol wird als Codierung ein 7-Bit langer Code, verwendet.

Dez	Hex	Char
0	0x00	NUL
1	0x01	SOH
2	0x02	STX
3	0x03	ETX
4	0x04	EOT
5	0x05	ENQ
6	0x06	ACK
7	0x07	BEL
8	0x08	BS
9	0x09	TAB
10	0x0A	LF
11	0x0B	VT
12	0x0C	FF
13	0x0D	CR
14	0x0E	SO
15	0x0F	SI
16	0x10	DLE
17	0x11	DC1
18	0x12	DC2
19	0x13	DC3
20	0x14	DC4
21	0x15	NAK
22	0x16	SYN
23	0x17	ETB
24	0x18	CAN
25	0x19	EM
26	0x1A	SUB
27	0x1B	ESC
28	0x1C	FS
29	0x1D	GS
30	0x1E	RS
31	0x1F	US

Dez	Hex	Char
32	0x20	SP
33	0x21	!
34	0x22	"
35	0x23	#
36	0x24	$
37	0x25	%
38	0x26	&
39	0x27	'
40	0x28	(
41	0x29	)
42	0x2A	*
43	0x2B	+
44	0x2C	,
45	0x2D	-
46	0x2E	.
47	0x2F	/
48	0x30	0
49	0x31	1
50	0x32	2
51	0x33	3
52	0x34	4
53	0x35	5
54	0x36	6
55	0x37	7
56	0x38	8
57	0x39	9
58	0x3A	:
59	0x3B	;
60	0x3C	<
61	0x3D	=
62	0x3E	>
63	0x3F	?

Dez	Hex	Char
64	0x40	@
65	0x41	A
66	0x42	B
67	0x43	C
68	0x44	D
69	0x45	E
70	0x46	F
71	0x47	G
72	0x48	H
73	0x49	I
74	0x4A	J
75	0x4B	K
76	0x4C	L
77	0x4D	M
78	0x4E	N
79	0x4F	O
80	0x50	P
81	0x51	Q
82	0x52	R
83	0x53	S
84	0x54	T
85	0x55	U
86	0x56	V
87	0x57	W
88	0x58	X
89	0x59	Y
90	0x5A	Z
91	0x5B	[
92	0x5C	\
93	0x5D	]
94	0x5E	^
95	0x5F	_

Dez	Hex	Char
96	0x60	`
97	0x61	a
98	0x62	b
99	0x63	c
100	0x64	d
101	0x65	e
102	0x66	f
103	0x67	g
104	0x68	h
105	0x69	i
106	0x6A	j
107	0x6B	k
108	0x6C	l
109	0x6D	m
110	0x6E	n
111	0x6F	o
112	0x70	p
113	0x71	q
114	0x72	r
115	0x73	s
116	0x74	t
117	0x75	u
118	0x76	v
119	0x77	w
120	0x78	x
121	0x79	y
122	0x7A	z
123	0x7B	{
124	0x7C	|
125	0x7D	}
126	0x7E	~
127	0x7F	DEL

3.3 Operatoren

Die Verknüpfung von Bits oder Bit-Folgen erfolgt in Verilog durch Operatoren. George Boole veröffentlichte 1849 ein Schema zur algebraischen Beschreibung von Prozessen, die beim logischen Denken und bei Gedankenschlüssen beteiligt sind. Dieses Schema wurde erweitert und trägt heute den Namen Boolsche Alegbra. In den späten 1930ern erkannte Claude Shannon, dass die Boolsche Algebra ein effektives Werkzeug darstellt um Schaltnetzwerke - bestehend aus vielen Schaltern - mathematisch zu beschreiben. Diese Algebra ist demnach ideal zur Entwicklung, Beschreibung oder Analyse digitaler Schaltungen geeignet. Beispielsweise lässt sich aus einer Karnaugh Tabelle die Funktion der kombinatorischen Logik eines Gates ablesen und anschliessend vereinfachen. Da wir die Boolesche Algebra möglicherweise noch benutzen werden, seien im Folgenden ein paar Theoreme und Eigenschaften wiederholt. Das Plus Zeichen + entspricht einem ODER, während zwei durch den Punkt · verknüpfte Variablen per UND

verknüpft sind:

UND / ODER Logik:

$$a \cdot 0 = 0$$
$$a \cdot 1 = a$$
$$a \cdot a = a$$
$$a + 1 = 1$$
$$a + 0 = a$$
$$a + a = a$$

Komplementärgesetze:

$$a + \overline{a} = 1$$
$$a \cdot \overline{a} = 0$$

Kommutativgesetze:

$$a + b = b + a$$
$$a \cdot b = b \cdot a$$

Assoziativgesetze:

$$(a + b) + c = a + (b + c)$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Distributivgesetze:

$$a \cdot (b + c) = a \cdot b + a \cdot c$$
$$a + b \cdot c = (a + b) \cdot (a + c)$$

Absorptionsgesetze:

$$a + a \cdot b = a$$
$$a \cdot (a + b) = a$$

Gesetze von De Morgan:

$$\overline{a \cdot b} = \overline{a} + \overline{b}$$
$$\overline{a + b} = \overline{a} \cdot \overline{b}$$
$$a + \overline{a} \cdot b = a + b$$
$$a \cdot (\overline{a} + b) = a \cdot b$$

Das Vereinfachen logischer Funktionen hat kann den Vorteil haben, dass die Schaltung einfacher wird und schneller entwickelt werden kann. Wie wir noch sehen werden, kann auch das Umformen von Ausdrücken manchmal sinnvoll sein.

Wir können drei wichtige Typen von Operatoren unterscheiden: Bitweise Operatoren, Reduktions- sowie Logische Operatoren.
Wir betrachten zunächst die unterschiedlichen Typen der Booleschen Operatoren:

Bitweise Operatoren	
~a	not
a & b	and
a | b	or
a ^ b	xor
a ~^ b	xnor

Reduktions Operatoren	
&a	and
~&a	nand
|a	or
~|a	nor
^a	xor
~^a	xnor

Logische Operatoren	
!a	not
a&&b	and
a||b	or
a==b	nor
a!=b	xor

Bitweise Operatoren wirken auf oder zwischen Vektoren, und zwar linear in jeder Komponente:

~(4'b0101) = {~0,~1,~0,~1} = 4'b1010
4'b0101 & 4'b0011 = {0&0, 1&0, 0&1, 1&1} = 4'b0001

Reduktions Operatoren wirken zwischen allen Komponenten des nachfolgenden Vektors:

&(4'b0101) = 0 & 1 & 0 & 1 = 1'b0
^(4'b0100) = 0 ^ 1 ^ 0 ^ 0 = 1'b1

Logische Operatoren liefern ein Wahr bzw. Falsch also ein einzelnes Bit als Ergebnis:

!(4'b0100 == 4'b0101) = 1'b0
(2'b01 == 2'b01) || (4'b0100 == 4'b0010) = 1 || 0 = 1'b1

Sonstige Operatoren in Verilog: Auch wenn wir diese Operatoren in den hier vorgestellten Beispiel Quelltexten kaum verwenden, sollten sie hier vorgestellt werden. Durch sie können wir sehen, wie flexibel die Hardware-Beschreibungssprache Verilog bzw. die Synthetisierungssoftware ist. Weiterhin können wir erahnen, wie komplex FPGA Projekte werden können und wie leistungsfähig heutige FPGAs

sind. Wie wir sehen werden, können wir den FPGA mit den hier vorgestellten Projekten nur zu wenigen Prozent auslasten.

Relations Operatoren

a < b	kleiner als
a <= b	kleiner gleich
a > b	größer als
a >= b	größer gleich

Bedingungs Operatoren

(a) ? b : c	wenn a dann b sonst c

Arithmetische Operatoren

-a	negative Zahl
a+b	Addition
a-b	Subtraktion
a*b	Multiplikation
a/b	Division
a%b	Modulus
a**b	Potenz
a<<b	Links Shift
a>>b	Rechts Shift

Wie können wir nun diese Operatoren in Verilog verwenden?

3.4 Wire

Ein 'wire' definiert in Verilog wie wir schon erwähnt haben **ein FPGA internes Kabel**, was zur Verbindung von FPGA internen Komponenten wie z.B. Gates, Decodern, Flip-Flops etc. verwendet wird. Dabei wird zur allgemeinen Definition folgende Syntax verwendet:

```
1
2  wire A;        // ein 1-Bit wire
3  wire [1:0] B;  // ein 2-Bit wire
4  wire [15:0] C;  // ein 16-Bit 'Bus'
5
```

Weiterhin ist folgende Syntax möglich um direkt bei der Definition eines 'wire' eine kombinatorische Logik einzubauen:

```
1
2  // Zunächst zwei Beispiele inklusive kombinatorischer Logik,
3  // die in einem 'wire D' bzw. 'wire E' als Ausgangsleitung enden:
4
5  wire D = X | Y;
6  wire E = ((X==1) && Y==1));
7
```

Die Schreibweise in obiger Syntax bei (X==1) und (Y==1) soll zeigen, dass die Quartus Software Zahlen, die nicht im Format 4'bXXXX oder 8'hFF geschrieben werden, als Dezimalzahlen betrachtet. Diese werden automatisch in Binärzahlen umwandelt, und die niedrigsten Bits (hier 4 bzw. 8 Bits) als Eingabewert verwendet. Eine 1 oder eine 0 im Dezimalsystem wird demnach umgewandelt in einen 1-Bit Wert und spart Schreibarbeit (der Wert 1'b0 bzw. 1'b1 kann als 1 bzw. 0 geschrieben werden).

3.5 assign-Anweisung

Mit der assign-Anweisung wird eine kontinuierliche Zuweisung **für Output Signale** gemacht, sie reagiert nicht edge-sensitiv auf andere Signale. Sie ist deshalb für kombinatorische Zwecke gedacht, beispielsweise für sogenannte Buffer hinter Signalen mit hohem Fan-Out, oder auch für Multiplexer. Mit einer abkürzenden Schreibweise für die if-Anweisung (die analog auch einen Multiplexer realisieren kann) betrachten wir nun folgenden Code für eine besondere Art von Buffer:

```
1
2  assign data_out[] = (enabled) ? data[] : 8'bz ;
3
```

Dies ist ein Tri-State-Buffer. Ist 'enabled' ein wahrer Ausdruck bzw. auf dem Logik Level High, so wird dem Ausgang 'data_out' der Eingang 'data' zugewiesen, ansonsten bekommt 'data_out' acht Bits vom Typ 'z' zugewiesen und wird damit hochohmig gesetzt (englischer Ausdruck hierfür: High Impedance). Wird der Bereich in den eckigen Klammern leergelassen, so ist die gesamte Bitbreite des Signals gemeint. Noch einfacher ist es allerdings die eckigen Klammern einfach ganz weg zu lassen - auch so ist die gesamte Bitbreite gemeint.

Im Folgenden noch ein simpler 2-Bit 2-to-1 Multiplexer, einmal direkt in der Definition eines wire:

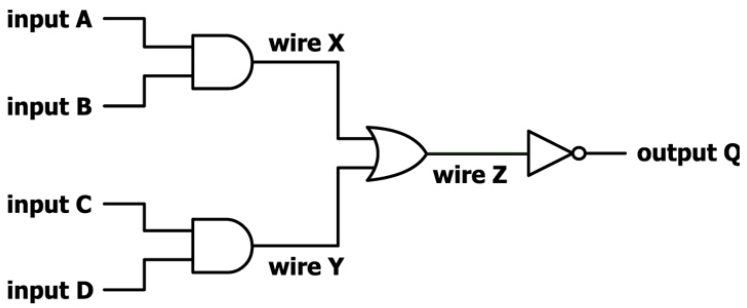
```
1
2  wire [1:0] data = (select) ? data_a[1:0] : data_b[1:0] ;
3
```

und die Variante mit der assign-Zuweisung:

```
1
2  wire [1:0] data;
3
4  assign data = (select) ? data_a[1:0] : data_b[1:0] ;
5
```

Im Folgenden wollen wir nun einige Beispiele untersuchen.

Wir betrachten ein einfaches Beispiel, um die Verwendung von 'wire' und 'assign' zu wiederholen:



Und direkt im Anschluss eine mögliche Umsetzung in Verilog:

```
1
2  module aoi (A, B, C, D, Q)
3
4  input A, B, C, D;
5  output Q;
6
7  wire X = A & B;
8  wire Y = C & D;
9
10 wire Z = X | Y;
11
12 assign Q = ~Z;
13
14 endmodule
15
```

Hier ist zu sehen, dass Inputs natürlicherweise vom Typ 'wire' sind, genauso wie Outputs. Um FPGA interne Komponenten, Gates bzw. Module zu verbinden sind

ebenfalls 'wire' notwendig.
Bei Outputs ist jedoch zu unterscheiden, ob der Wert rein kombinatorisch oder mithilfe von Flip-Flops (sequentiell) ermittelt wird. Handelt es sich um eine kombinatorische Funktion, so lässt sich der Ausgang mit einer 'wire' bzw. einer 'assign' Zuweisung verknüpfen. Wird für die Ermittlung des Ausgangs Wertes ein Flip-Flop benötigt, so muss aus der Definition 'output Q' ein 'output reg Q' werden. Dieser Ausgang kann dann nicht mehr direkt kombinatorisch (mittels 'wire' oder 'assign') verknüpft werden, sondern die Zuweisung muss in einem 'always-Block' erfolgen. Dazu kommen wir später noch im Detail.

Der folgende Verilog Code demonstriert die Verknüpfung von Inputs durch Operatoren:

```
1
2  module gates (A, B, C,
3          AND_gate, NAND_gate, OR_gate, NOR_gate, XOR_gate, XNOR_gate, NOT_gate);
4
5  input A, B, C;
6  output AND_gate, NAND_gate, OR_gate, NOR_gate, XOR_gate, XNOR_gate, NOT_gate;
7
8  assign        AND_gate  = A & B;
9  assign        NAND_gate = ~(A & B);
10 assign        OR_gate   = A | B;
11 assign        NOR_gate  =  ~(A | B);
12 assign        XOR_gate  = A ^ B ^ C;    // Drei Signale durch 'XOR' verbunden
13 assign        XNOR_gate = ~(A ^ B);
14 assign        NOT_gate  = ~A ;
15
16 endmodule
17
```

Die Outputs werden hier mit der 'assign' Anweisung verbunden.

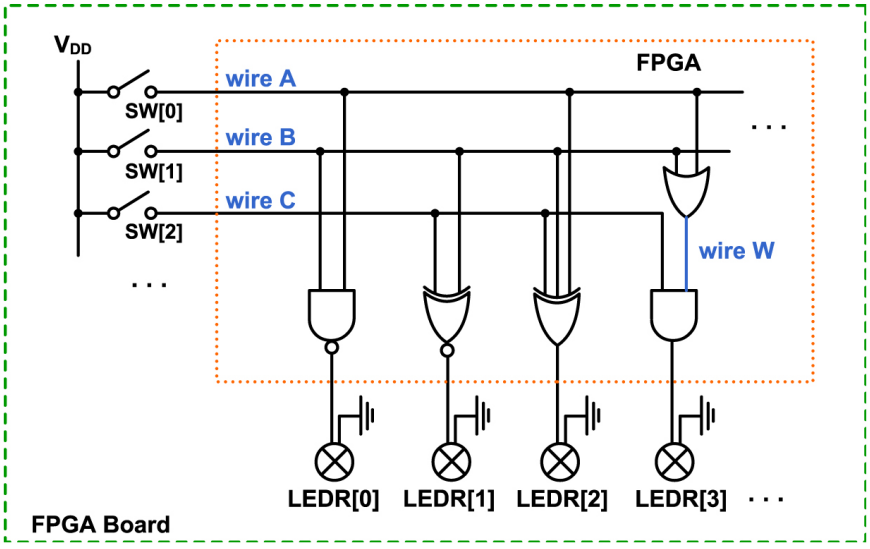
Hierzu eine Alternative, es werden vorgefertigte Konstrukte für Gates in Verilog verwendet:

```
1
2  module gates (A, B, C,
3          AND_gate, NAND_gate, OR_gate, NOR_gate, XOR_gate, XNOR_gate);
4
5  input A, B, C;
6  output AND_gate, NAND_gate, OR_gate, NOR_gate, XOR_gate, XNOR_gate;
```

```
7
8           // Im Folgenden ist das erste Argument in Klammern immer der Ausgang
9           // alle weiteren Argumente sind die Eingänge:
10          // Vor den Klammern links steht der Name des Gates:
11
12  and   and1  (AND_gate, A,B) ;   // Hier ein AND-Gate namens and1
13  nand  nand1 (NAND_gate, A,B);
14  or    or1   (OR_gate, A,B);     // Reihenfolge: (Ausgang, Eingang, Eingang,...)
15  nor   nor1  (NOR_gate, A,B);
16  xor   xor1  (XOR_gate, A,B,C); // Das Gate kann auch mehr als Zwei Inputs haben
17  xnor  xnor1  (XNOR_gate, A,B);
18
19  endmodule
20
```

Beide Möglichkeiten liefern eine rein kombinatorische Logik.

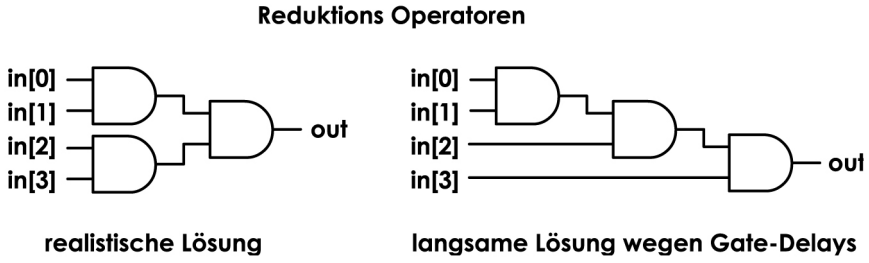
Wir betrachten nun ein erstes Projekt, um den FPGA und die Verschaltung mit externen Komponenten kennenzulernen. Dieses Projekt wird in allen Schritten zu Praktikumsbeginn vorgeführt. Dabei soll vor Allem der Work-Flow (also ein effektiver Arbeitsablauf bei den Projekten) demonstriert werden. Die folgende Skizze zeigt, welche Art von Schaltung hergestellt werden soll:



Offensichtlich handelt es sich hierbei um ein sehr einfaches Projekt, welches nach Belieben mit mehr Gates, Inputs und Outputs versehen werden kann. Es kön-

nen auch mehrere Gates hintereinander geschaltet werden, indem die Ausgangs-Signale einiger Gates mit den entsprechenden Eingängen eines weiteren Gates - hier beispielsweise durch ein 'wire' namens 'W' - verbunden werden.

Kommen wir nun zur Reduktions Logik, folgende Abbildung zeigt Möglichkeiten zur Realisierung von reduzierenden AND Operatoren:



Die meisten Operatoren sind unter der gleichen Bezeichnung übrigens auch in der Programmiersprache C enthalten.

Betrachten wir nun einige spezielle Syntax-Konstrukte, die ebenfalls relativ C ähnlich sind:

3.6 if-else-Anweisung

Um einen bestimmten Zustand einer Variable abzufragen können wir ein Konstrukt der folgenden Art benutzen:

```
1
2  // Der 'always' Block ist notwendig, um die Inputs festzulegen,
3  // auf deren Änderung hin reagiert werden soll:
4
5  always @ (enable)
6  begin
7          if (enable == 1)  // Wenn 'enable' gleich Eins ist
8                  begin
9                          address = 4'b0000; // 'address' auf '0000' setzen
10                         data = 8'hF0;        // 'data' auf '11110000' setzen
11                         write = 1'b1;        // und 'write' auf Eins
12                 end
13         else  // Wenn 'enable' nicht gleich Eins ist, also Null
14                 begin
```

```
15                          write = 1'b0 ;
16                          data_out = 8'h00 ;  // 'write' und 'data_out' zuweisen
17                          address = 4'b1010 ; // und 'address' auf '1010' setzen
18                  end
19  end
20
```

Es könnten beispielsweise Multiplexer verwendet werden um diese Art von Schaltung zu realisieren.

3.7 case-Anweisung

Soll eine Variable auf mehrere Werte überprüft werden, ist folgende Syntax zur Fallunterscheidung hilfreich, diese entspricht im Grunde ebenfalls nur einer Anordnung von Multiplexern:

```
1
2  // Zunächst wird wieder die Variable angegeben,
3  // auf die der always-Block sensitiv reagieren soll:
4
5  always @ (bus_address)
6  begin
7          case(bus_address)
8                  0 : data[1:0] = data_bus[1:0] ;
9                  1 : data[1:0] = data_bus[3:2] ;
10                 2 : data[1:0] = data_bus[5:4] ;
11                 3 : data[1:0] = 2'b00 ;
12         endcase        // 'case' muss mit 'endcase' beendet werden
13 end
14
```

Das Signal 'bus_address' hätte hier demnach idealerweise 2-Bit, in Dezimalschreibweise können wir die Fälle einfach abfragen.

Im Folgenden noch eine andere praktische Variante:

```
1
2  always @ (a or b or c or d)  // a,b,c und d sind in der 'Sensitivity Liste'
3  begin
4          case( {a,b,c,d} )
5                  4'b0010 : data[1:0] = data_bus[1:0] // 1.Fall;
6                  4'b0101 : data[1:0] = data_bus[3:2] // 2.Fall;
7                  default : data[1:0] = 2'b00 ; // alle andere Fälle
8          endcase
9  end
10
```

Hier wird das aus Vier Bits 'a', 'b', 'c' und 'd' zusammengesetzte 4-Bit Wort - man könnte es auch als 4-Bit Buskabel bezeichnen - 'abcd' überprüft. Dies lässt sich als Umsetzung einer Wahrheits-Tabelle auffassen. Mit 'default' werden alle Fälle abgehandelt, die nicht explizit untersucht werden. Sind dies mehrere, nicht explizit untersuchte Fälle, so wird eine State-Maschine impliziert. Wir werden dafür Flip-Flops benötigen wie wir in einem späteren Kapitel noch diskutieren wollen.

3.8 for-Schleife

Um eine regelmäßig auf- oder abwärts zählende Folge von ganzen Zahlen zu erhalten - beispielsweise als Bit Index einer Mehr-Bit Variablen - kann die for-Schleife hilfreich sein:

```
1
2  integer i; // Die Laufvariable muss definiert werden
3
4  always @ (data)
5  begin
6          for (i=0; i<3; i=i+1) // Startwert für 'i' ist Null, Endwert ist Zwei
7          begin
8                  shift[i] = data[i+1]; // drei parallele Anweisungen
9          end
10         shift[3] = data[0]; // Parallel dazu eine Anweisung unabhängig von 'i'
11 end
12
13
```

Zwischen 'begin' und 'end' werden also in diesem Fall Vier parallele Zuweisungen gemacht, Drei von der Art:

'shift[0]' bekommt den Wert von 'data[1]'
'shift[1]' bekommt den Wert von 'data[2]'
'shift[2]' bekommt den Wert von 'data[3]'

darunter folgt noch ein einzelnes - weil von 'i' unabhängiges - ebenfalls dazu parallel ausgeführtes

'shift[3]' bekommt den Wert von 'data[0]'.

Genaugenommen dient die for-Schleife also nur dazu, dem Entwickler Schreibarbeit zu ersparen. Alle in dieser for-Schleife möglichen Zuweisungen werden in der Anwendung parallel ausgeführt.

3.9 Submodule benutzen

Die Verwendung von bereits geschriebenen Modulen kann ebenfalls Arbeit ersparen, aber auch zu einer besseren Übersichtlichkeit des Projekts beitragen.

Angenommen wir haben ein Modul erstellt namens 'adder' und ein Modul namens 'comparator' und diese wären etwa wie folgt aufgebaut:

```

module adder (a, b, sum);

input [1:0] a,b;

output [2:0] sum;

// etwas Code

endmodule

```

sowie:

```

module comparator (a, b, equal, smaller);

input [1:0] a,b;

output equal, smaller;

// etwas Code

endmodule

```

dann würden wir diese beiden Sub-Module in unserem Top-Level Modul wie folgt verwenden:

```

module top_level (a, b, sum, equal, smaller);

input a,b;

output sum, equal, smaller;

// Im Folgenden das Implementieren der Sub-Module

adder adder1 (a, b, sum);  // 'adder1' und 'comp1' sind beliebige
comparator comp1 (a, b, equal, smaller);  // aber jeweils einzigartige Namen

endmodule

```

Am besten erlernt sich Verilog durch praktische Anwendung. Die folgenden Kapitel bieten einige Beispiele, um die unterschiedlichen Anweisungen zu verstehen. Mit dem RTL-Viewer und durch die Simulation lassen sich das Prinzip der Synthetisierung von Verilog zur fertigen Schaltung und die Funktionsweise der digitalen Schaltung nachvollziehen.

Beginnen wir mit etwas kombinatorischer Logik, an der das weiter oben diskutierte Gate Delay gut beobachtet werden kann.

3.10 Lab 1

Um die für uns vorerst wichtigsten Komponenten kennenzulernen, erstellen wir nun unser erstes Projekt.

1. Wir wollen eine Schaltung erstellen, welche die LEDs mit den Schaltern verknüpft:
   Dazu erstellen wir also zunächst ein neues Projekt und verwenden die korrekten Einstellungen für unser FPGA Board. Sobald das Projekt erstellt wurde, ist es ratsam die DE2 Pin Tabelle zu importieren. Jetzt kann der Standard Rahmen für ein Verilog Modul eingefügt werden:

```
1
2  module my_name (a, b, c, ...);
3
4
5  endmodule
6
```

   Unser FPGA Board besitzt sehr viele LEDs und Schalter, es ist daher sinnvoll, die Variablen als Vektoren zu formulieren:

```
1
2  module my_name (SW, KEY, LEDR);
3
4  input [9:0] SW;  // Switches
5  input [3:0] KEY;  // Keys = Push-Buttons
6  output [9:0] LEDR;  // Red LEDs
7
8  endmodule
9
```

Unsere Ausgänge LEDR können direkt per assign-Anweisung mit Eingängen verknüpft werden:

```
1
2  assign LEDR[0] = SW[0];
3  assign LEDR[1] = SW[1];
4  assign LEDR[2] = SW[2];
5  ...
6
```

Alternativ können wir auch mehrere LEDR in einem Schritt zuweisen:

```
1
2  assign LEDR[3:0] = SW[3:0];
3
```

Ebenso werden die Push-Buttons eingesetzt:

```
1
2  assign LEDR[3:0] = ~KEY[3:0];
3
```

Hierbei ist zu beachten, dass die Push-Buttons active low betrieben werden, daher verwenden wir üblicherweise den invertierten Wert.

Ist nun das Modul fertig geschrieben, so kann das Projekt compiliert werden. Das Modul wird dabei automatisch gespeichert und weiterhin findet eine Syntax Überprüfung statt, welche umgehend auf Fehler aufmerksam macht.

Bei einem fehlerfreien Ablauf, sollten im 'Compilation Report' nur noch normale Warnungen, keine 'Critical Warnings' auftauchen. Unter den normalen Warnungen können in vielen Fällen dennoch wichtige Informationen enthalten sein, beispielsweise werden nicht-verbundene Ein- und Ausgänge aufgelistet.

Der nächste Schritt ist eine vollständige Simulation des Projekts. Dazu wird ein 'Vector Waveform File' erstellt, und als Erstes per Doppelklick im weissen Bereich unter der 'Name' Spalte der Dialog zum Hinzufügen neuer Signale aufgerufen. Hier ist die Auswahl 'Show Pins: All' sinnvoll, im linken

Teil des Dialogfensters stehen dann die vorhandenen Signale zur Auswahl und können mit dem '»' Symbol der Simulation hinzugefügt werden.

Ist dieser Schritt erledigt, so lässt sich nun mit dem 'Waveform Editing Tool' der Signalverlauf für jeden Eingang 'zeichnen'. Sind alle Eingänge in dieser Weise behandelt worden, so kann nun die Simulation gestartet werden. Um diesen Prozess etwas individueller gestalten zu können, ist das 'Simulator Tool' aus dem Menü hilfreich. Hier kann mit den Einstellungen in gewissem Rahmen experimentiert werden. Weiterhin sollte auch die 'Functional Simulation' einmal ausprobiert werden, um die Ergebnisse mit der 'Timing' Simulation vergleichen zu können.

Wenn die Simulation nun erfolgreich war, und im Idealfall die 'Simulation Coverage' bei 100 Prozent liegt, so folgt der letzte Schritt: Die Konfiguration des FPGAs.

Dazu starten wir den 'Programmer' und überprüfen die Standardeinstellungen. Als USB-Hardware muss der 'USB-Blaster' ausgewählt sein, dies sollte automatisch der Fall sein, sofern das FPGA-Board bereits VOR dem Starten der Quartus Software angeschlossen und angeschaltet wurde.

Ist dieser Vorgang beendet, kann die konstruierte Schaltung getestet werden.

2. Das hiermit erstellte Projekt kann als Ausgangsbasis für die folgenden Schaltungen dienen. Wir können nun zwischen die Schalter und die LEDs auch Gates setzen, und weiterhin die assign-Anweisung benutzen:

```
1
2 assign LEDR[5] = SW[5] & SW[6];
3 assign LEDR[7] = SW[7] ^ SW[8];
4
```

Wollen wir mehrere Gates hintereinander schalten so können wir wie folgt vorgehen:

```
1
2  wire X = SW[13] & SW[14];
3  wire Y = SW[11] | SW[12];
4
5  assign LEDR[7] = X ^ Y;
6
```

Es soll nun eigenständig ein Multiplexer aus Gates aufgebaut werden, und dieser dann verknüpft mit LED und Schalter, um die Schaltung zu demonstrieren. Dazu ist natürlich vorweg eine Simulation zu erstellen, anschliessend kommt der Programmer zum Einsatz.

3. Da wir nun die Switches, die Keys und die LEDs kennengelernt haben, widmen wir uns nun noch den 7-Segment HEX Anzeigen. Wir wollen zunächst nur verstehen, wie wir die einzelnen Segmente ansteuern können. In unserem ersten Projekt verwenden wir nur die Ausgänge HEX0 und HEX1. Wie der Name schon andeutet können 7 Segmente angesteuert werden, diese Ausgänge würden wir demnach wie folgt in unser Projekt einfügen:

```
1
2  module my_name (SW, KEY, LEDR, HEX0, HEX1);
3
4  ...
5
6  output [6:0] HEX0;
7  output [6:0] HEX1;
8
9  endmodule
10
```

Insgesamt haben wir 8 derartige Anzeigen mit den Namen HEX0 bis HEX7 zur Verfügung.

Es soll nun eigenständig eine Schaltung entwickelt werden, um mit den Switches verschiedene HEX Segmente aktivieren zu können.

Für spätere Projekte kann es manchmal sinnvoll sein, eine Sicherungskopie der relevanten Verilog-Dateien zu erstellen. Bei komplexeren Projekten kann schon mal die Übersicht verloren gehen, welcher Schritt zu welchem

Fehler geführt hat (Dies betrifft vor Allem Timing Probleme, denen wir unter Umständen in späteren Kapiteln begegnen können).

4. Abschliessend wollen wir nun noch eine Schaltung konstruieren, die mit einem Schalter zwischen zwei Zuständen eines HEX Displays wechseln kann. Der Schalter soll in der einen Position auf der Anzeige HEX7 die obere Hälfte der Segmente leuchten lassen und in der anderen Position die untere Hälfte der Segmente.