VHDL-Synthese

Entwurf digitaler Schaltungen und Systeme

von
Prof. Dr. Jürgen Reichardt
Prof. Dr.-Ing. Bernd Schwarz

6., aktualisierte und erweiterte Auflage

Oldenbourg Verlag München

Prof. Dr. Jürgen Reichardt lehrt seit 1995 Informationstechnik an der Hochschule für Angewandte Wissenschaften Hamburg. Seine Arbeitsschwerpunkte sind Digitaler Systementwurf, Digitale Signalprozessoren, HW-SW-Codedesign, System-on-Chip(Soc)-Entwurf.

Prof. Dr.-Ing. Bernd Schwarz lehrt seit 1993 an der HAW Hamburg und vertritt die Themenbereiche Digitale Systeme und Computer Engineering.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

© 2013 Oldenbourg Wissenschaftsverlag GmbH
Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-verlag.de

Das Werk einschließlich aller Abbildungen ist urheberrechtlich geschützt. Jede Verwertung außerhalb der Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Bearbeitung in elektronischen Systemen.

Lektorat: Dr. Gerhard Pappert
Herstellung: Constanze Müller
Titelbild: Autor; Gestaltung: Irina Apetrei
Einbandgestaltung: hauser lacour
Gesamtherstellung: freiburger graphische betriebe GmbH & Co. KG, Freiburg

Dieses Papier ist alterungsbeständig nach DIN/ISO 9706.

ISBN 978-3-486-71677-1
eISBN 978-3-486-71678-8

Vorwort

Das vorliegende Buch entstand in seiner ersten Auflage vor nunmehr mehr als 12 Jahren aus einer Neubewertung der Lehrinhalte der Fächer „Digitaltechnik“ und „Entwurf Digitaler Systeme“ an den Departments Informations- und Elektrotechnik sowie Informatik der HAW-Hamburg. Diese hatte zum Ergebnis, die Hardwarebeschreibungssprache VHDL ((**V**ery High Speed Integrated Circuit) **H**ardware **D**escription **L**anguage) nicht als eigenständige Lehrveranstaltung sondern parallel zur Digitaltechnik zu vermitteln, um darauf aufbauend in den weiteren Lehrveranstaltungen Systementwürfe gestalten zu können. Ein besonderes Ziel dieser Lehrveranstaltungen ist es, den bekannten Beschreibungsformen für Elemente der Digitaltechnik einen Hochsprachenentwurf gegenüberzustellen. Der starke Hardwarebezug der genannten Lehrveranstaltungen setzt dabei voraus, dass überwiegend synthesefähige VHDL-Beschreibungen entwickelt werden. Besonderen Wert legen wir daher auf die Kombination aus Hardwarebeschreibung, Funktionsprüfung durch Simulation und Analyse der Syntheseergebnisse. Den synthesefähigen Hardwarebeschreibungen der ausgewählten Funktionselemente werden, soweit erforderlich, die digitaltechnischen bzw. bei den komplexeren Anwendungen der Kap. 9 und 10 auch die Grundlagen der systemtheoretischen Entwurfsprinzipien zur Seite gestellt.

Das Lehrbuch soll als Zielgruppe zunächst die Studierenden der Elektrotechnik und der Informatik an Hochschulen und Universitäten erreichen. Es ist gerade mit seinen Erweiterungen zur Modellierung komplexerer Systeme in den Kapiteln 8, 9 und 10 aber ebenso auf Entwicklungsingenieure[1] in der Praxis ausgerichtet, die erstmalig einen VHDL-basierten Digitalentwurf beabsichtigen.

Der Verkaufserfolg sowie die sehr positiven Rückmeldungen der Leser zu den bisherigen Auflagen der „VHDL-Synthese“ gab uns mit diesem Ansatz Recht und war uns Ansporn für eine kontinuierliche Weiterentwicklung der Inhalte bis zur nun vorliegenden 6. Auflage.

So wurden bereits in den letzten Auflagen unterschiedliche Systeme ergänzt bzw. überarbeitet, in denen die in den Kap. 1 bis 7 erlernten Grundlagen an konkreten Aufgabenstellungen vertieft werden. Diese reichen in der nun vorliegenden 6. Auflage vom Entwurf eines einfachen RISC-Prozessors in Kap. 8 über die Implementierung digitaler Filter in Kap. 9 bis hin zur Implementierung von Zustandsdifferenzengleichungen für Beobachter in Kap. 10, die für digitale Regelungen z.B. in der elektrischen Antriebstechnik benötigt werden.

---

[1] Wir bitten die Leserinnen um Verständnis, dass wir im Folgenden die männlichen Berufsbezeichnungen verwenden. Wir haben uns wegen der Kürze und der unseres Erachtens besseren Lesbarkeit für diese Formulierungen entschieden.

Weitere Änderungen bzw. Ergänzungen dieser Auflage sind:

- Die Einführung der standardisierten VHDL-Bibliothek numeric_std in Kap. 5 und deren Nutzung für alle nachfolgenden arithmetischen Anwendungen.
- Die Ergänzung des Kap. 3 um einen Abschnitt zur Modellierung von VHDL-Testumgebungen.
- Die Verwendung eines Dual-Port RAM-Bausteins als Speicher für die Instruktionen und Daten des RISC-Prozessors in Kap. 8.
- In Kap. 9 die Ergänzung um das Modell eines systolischen FIR-Filters sowie die Neubewertung der Implementierungsergebnisse durch neuere FPGA-Implementierungswerkzeuge für Xilinx FPGAs (ISE 13.4).

Wir danken unseren Studenten, die durch vielfältige Diplom-, Bachelor- und Master-Arbeiten in den letzten 12 Jahren wertvolle Beiträge zur systematischen Weiterentwicklung dieses Lehrbuches beigetragen haben. Weiterer Dank gilt den Lektoren des Oldenbourg-Verlages, die uns mit Rat und Tat bei der Erstellung des Buches geholfen haben.

Hamburg, Oktober 2012 J. Reichardt, B. Schwarz

Inhaltsverzeichnis

1 Einleitung und Übersicht 1
1.1 Motivation ... 1
1.2 Ziele und Organisation dieses Buches ... 2
1.3 Syntaxnotation ... 6
2 Synthese einfacher Schaltnetze 7
2.1 Entity, Architektur und Signale ... 7
2.1.1 Deklaration und Verwendung lokaler Signale ... 10
2.1.2 Richtungsmodi von Signalschnittstellen ... 11
2.1.3 Deklaration von Bussignalen ... 13
2.2 Simulation von VHDL-Entwürfen ... 13
2.3 Schaltnetze mit Boole'schen Gleichungen ... 16
2.4 Synthese selektiver und bedingter Signalzuweisungen ... 19
2.5 Übungsaufgaben ... 24
3 Entwurf digitaler Funktionselemente mit Prozessen 27
3.1 Prozesse ... 27
3.1.1 Deklaration und Ausführung von Prozessen ... 28
3.2 Schaltnetze mit sequentiellen Anweisungen ... 31
3.2.1 Die case-Anweisung ... 32
3.2.2 Die if-Anweisung ... 36
3.2.3 Schleifenkonstrukte ... 39
3.2.3.1 for loop ... 40
3.2.3.2 while loop ... 41
3.3 Einfache Schaltwerke ... 43
3.3.1 D-Flipflops und Register ... 44
3.3.1.1 VHDL-Beschreibung mit Signalen ... 44
3.3.1.2 Testbench zum D-Flipflop ... 46
3.3.1.3 Entwurf von Registern ... 48
3.3.1.4 Verwendung von Variablen in taktsynchronen Prozessen ... 48

3.3.2 Johnson-Zähler mit Taktteiler ... 50
3.3.3 Parametrisiertes Schieberegister ... 52
3.4 Erzeugung von Latches ... 55
3.5 Vermeidbare Synthesefehler ... 57
3.5.1 Kombinatorische Schleifen ... 57
3.5.2 Fehlverhalten durch unvollständige Empfindlichkeitslisten ... 59
3.6 RTL-Syntheserichtlinien ... 60
3.6.1 D-Flipflops und Register in getakteten Prozessen ... 62
3.6.2 D-Latches und kombinatorische Logik ... 62
3.7 VHDL-Testbenches ... 63
3.8 Übungsaufgaben ... 68

4 Tri-State- und Don't-Care-Modellierung 75
4.1 Die Datentypen std_ulogic und std_logic ... 75
4.2 Realisierung von Tri-State-Ausgangsstufen ... 78
4.3 Don't-Care-Werte als Ausgangssignal ... 83
4.4 Don't-Care-Werte als Eingangssignal ... 85
4.5 Konversion der Datentypen bit und bit_vector ... 88
4.6 Übungsaufgaben ... 90

5 Arithmetik und Synchronzähler 93
5.1 Arithmetik-Operatoren und zugehörige Datentypen ... 93
5.2 Komparator SN74xx85 ... 96
5.3 Entwurf von Synchronzählern ... 98
5.4 Arithmetik mit den Datentypen signed und unsigned ... 103
5.4.1 Entwurf eines kombinierten Addierers / Subtrahieres ... 105
5.5 Integer-Arithmetik ... 107
5.6 Konversionsfunktionen und indizierter Zugriff auf Feldelemente ... 110
5.7 Arithmetik mit Synopys Bibliotheken ... 113
5.8 Hinweise zur Verwendung der Datentypen ... 114
5.9 Übungsaufgaben ... 115

6 Entwurf von Zustandsautomaten 119
6.1 Automatenvarianten ... 119
6.2 Moore-Automat für eine Impulsfolgenerkennung ... 122
6.3 Entwurfsbeispiel für einen Mealy-Automaten ... 126

6.4 VHDL-Syntaxvarianten ... 132
6.4.1 Die Zwei-Prozess-Darstellung ... 133
6.4.2 Die Mehr-Prozess-Darstellung ... 135
6.4.2.1 Schnittstellensynchronisation ... 135
6.4.2.2 Maßnahmen zur Taktfrequenzerhöhung ... 137
6.4.2.3 Maßnahme zur Reduzierung der Latenzzeit ... 139
6.4.3 Die Ein-Prozess-Darstellung ... 141
6.4.4 Vergleich der Syntaxvarianten ... 142
6.5 Zustandscodierung ... 143
6.5.1 Strategien der Zustandscodierung ... 144
6.5.2 Umsetzung der Zustandscodierung in VHDL ... 145
6.5.3 Auswirkungen der Zustandscodierung auf die Syntheseergebnisse ... 146
6.6 Übungsaufgaben ... 152
7 Struktureller VHDL-Entwurf 155
7.1 Ziele und Methoden der Systempartitionierung ... 156
7.2 Struktureller Entwurf mit Komponenten ... 159
7.2.1 Struktureller Entwurf eines 4 zu 2 Prioritätsencoders ... 161
7.2.1.1 Komponentendeklaration ... 163
7.2.1.2 Komponenteninstanziierung und port map-Anweisung ... 163
7.2.1.3 Konfiguration zur Auswahl von Modellarchitekturen ... 165
7.2.1.4 Modellparametrisierung ... 166
7.2.1.5 Iterative Instanziierung ... 168
7.2.2 Entwurf einer skalierbaren Addier/Subtrahier-Einheit ... 170
7.2.3 Kopplung von Signalen in strukturellen VHDL-Beschreibungen ... 175
7.3 Strukturierung mit Unterprogrammen ... 178
7.3.1 Lokale Prozeduren und Funktionen ... 178
7.3.2 Definition und Einsatz von packages ... 183
7.4 Herstellerspezifische Komponenten und Komponentengeneratoren ... 186
7.4.1 Instanziierung von RAM- und ROM-Zellen mit Prozessen ... 186
7.4.2 Komponentengeneratoren ... 192
7.5 Unterstützung durch Synthesewerkzeuge ... 198
7.6 Übungsaufgaben ... 200

8 Entwurf eines RISC-Prozessors 203
8.1 Spezifikation der Prozessorfunktionen 204
8.1.1 Das Programmiermodell 204
8.1.2 Der Instruktionssatz 205
8.1.3 Instruktionsformate 206
8.2 Prozessorarchitektur und -programmierung 207
8.3 VHDL-Realisierung 210
8.3.1 Registerfile 211
8.3.2 Instruktions- und Datenspeicher 214
8.3.3 Instruction Fetch (IF) 217
8.3.4 Instruction Decode (ID) 218
8.3.5 Execute (EX) 221
8.3.6 Memory Access (MEM) 223
8.3.7 Top-Level-Architektur des RISC-Prozessors 226
8.3.8 Einfache Testbench für den RISC-Prozessor 229
8.4 Entwurfsverifikation 230
8.4.1 Vermeidung von RW-Hazards 232
8.4.2 Vermeidung von C-Hazards 233
8.4.3 Speichern und Lesen des Datenspeichers 235
8.5 FPGA-Implementierung 236
9 Modellierung digitaler Filter 239
9.1 FIR-Filter 241
9.1.1 Parallele FIR-Filterstrukturen 241
9.1.2 Zahlendarstellung im Q-Format 246
9.1.2.1 Addition mit vorzeichenrichtiger Erweiterung der Summanden 248
9.1.2.2 Binäre Multiplikation 249
9.1.3 Filterskalierung und Beispielfilter 250
9.1.4 VHDL-Modelle paralleler FIR-Filterstrukturen 254
9.1.4.1 Kopplung von unterschiedlichen Taktbereichen 254
9.1.4.2 Basismodell der Direktform 257
9.1.4.3 Direktform mit symmetrisch balanciertem Addiererbaum 265
9.1.4.4 Direktform mit unsymmetrisch balanciertem Addiererbaum 269
9.1.4.5 Linear-Phasen-Struktur 279
9.1.4.6 Transponierte Form mit reduzierter Anzahl der Multiplizierer 289

9.1.4.7 Systolische FIR Filter ... 294
9.1.4.8 Vergleich der Implementierungsergebnisse und Timinganalysen ... 298
9.1.5 Sequentielle FIR-Struktur mit MAC-Einheit ... 300
9.1.5.1 Multiplizierer-Akkumulatoreinheit ... 303
9.1.5.2 Ringpuffer für die Abtastwerte ... 305
9.1.5.3 Koeffizienten-ROM ... 307
9.1.5.4 Sequenzsteuerung mit einem Zustandsautomaten ... 308
9.1.5.5 Simulations- und Syntheseergebnisse ... 312
9.1.6 Taktschemata der Filterdatenpfade ... 316
9.1.6.1 Audiodatenrate ... 317
9.1.6.2 Hohe Datenraten ... 318
9.2 IIR-Filter ... 319
9.2.1 Koeffizientenberechnung und Beispielfilter ... 319
9.2.2 Parallele Strukturen für IIR-Filter 2. Ordnung ... 324
9.2.3 VHDL-Modelle für Varianten der Direktform II ... 327
9.2.3.1 Transponierte Struktur der Direktform II ... 327
9.2.3.2 Modifizierte Direktform II ... 335
9.2.3.3 Implementierungsergebnisse und Timinganalyse ... 338
9.2.4 Sinusgenerator als synthesefähige Testbench ... 339
9.2.5 Entwurfsschritte für IIR-Filter ... 346
9.3 FPGA-Entwurf mit Simulink ... 347
10 Zustandsdifferenzengleichungen für Beobachter 351
10.1 Zeitinvariante und lineare Systeme ... 354
10.1.1 Zeitliche Diskretisierung ... 355
10.1.2 VHDL-Modell für die Zustandsdifferenzengleichungen ... 359
10.2 Numerische Integration von gekoppelten Differentialgleichungen ... 367
10.2.1 Rechteck- und Trapezintegration ... 368
10.2.2 VHDL-Modell der kombinierten numerischen Integration ... 372
10.2.3 Prozessorelement mit Mehrzyklus-Datenpfad ... 380
10.2.3.1 Algorithmisches Zustandsdiagramm ... 381
10.2.3.2 Datenpfad mit Mehrfach-Ressourcennutzung ... 383
10.2.3.3 VHDL-Modellierung des Prozessorelementes ... 388

10.3 Vergleich der Implementierungsergebnisse ... 397
10.4 Nichtlineare, gekoppelte Integralgleichungen ... 402
10.4.1 Odometrie mit einem Einspur-Fahrzeugmodell ... 403
10.4.2 VHDL-Modell des nichtlinearen Doppelintegrators ... 407
11 Anhang 421
11.1 VHDL-Codierungsempfehlungen ... 421
11.2 Checkliste zum VHDL-basierten Entwurf digitaler Systeme ... 425
11.3 Liste der VHDL-Schlüsselworte ... 429
11.4 Lösungen zu den Übungsaufgaben ... 429
11.4.1 Lösungen zu den Aufgaben in Kap. 2.5 ... 430
11.4.2 Lösungen zu den Aufgaben in Kap. 3.7 ... 432
11.4.3 Lösungen zu den Aufgaben in Kap. 4.6 ... 438
11.4.4 Lösungen zu den Aufgaben in Kap. 5.7 ... 441
11.4.5 Lösungen zu den Aufgaben in Kap. 6.6 ... 446
11.4.6 Lösungen zu den Aufgaben in Kap. 7.7 ... 451
12 VHDL-Syntaxübersicht und Bibliotheken 457
13 Literaturverzeichnis 467
14 Sachregister 473

1 Einleitung und Übersicht

1.1 Motivation

Die Entwicklung von mikroelektronischen Schaltungen auf hohem Abstraktionsniveau wird durch die beiden nachfolgenden Trends bestimmt:

- Starker Anstieg der Entwurfskomplexität
- bei drastisch reduzierten Entwicklungszeiten (Time to market).

Zugleich entwickelt sich der Markt für mikroelektronische Produkte von Computeranwendungen immer weiter hin zu Kommunikations- und Consumer-Anwendungen. Exemplarisch sollen hier nur das rasante Wachstum des Mobilfunkmarktes sowie die neuartigen Applikationen, die sich aus dem Zusammenwachsen der Computer und digitaler Fernsehsysteme im Zusammenhang mit der Internet-Entwicklung ergeben, genannt werden.

In der Industrie reichen herkömmliche Wege über grafische Schaltplaneingaben auf Logikelementebene schon seit längerem nicht mehr aus, um in den kurzen Entwicklungszyklen wettbewerbsfähig zu bleiben. Ein neuer Ansatz beim Schaltungsentwurf ist die Hardwarebeschreibung durch Hochsprachen, der durch eine zunehmende Zahl von CAE-Werkzeugen zur Schaltungssimulation und -synthese unterstützt wird. Ein besonderer Vorteil dieses Konzepts ist die gemeinsame Nutzung einer einheitlichen Modellierung für Simulation und Synthese. Dabei dient die Simulation der Verifikation der Entwurfsidee und die Synthesewerkzeuge setzen die Hardwarebeschreibung automatisch in Schaltpläne und Logikgatter-Netzlisten um. Letztere werden für die herstellerspezifische Implementierung der Schaltung auf Hardwareplattformen wie ASICs (Application Specific Integrated Circuits), FPGAs und CPLDs benötigt. Zusammen mit dem Einsatz der CAE-Werkzeuge führt die Hardwareentwicklung auf Basis von Hochsprachen dazu, dass die Vielfalt der implementierungsspezifischen Entwurfsdetails stark reduziert wird. Der Entwickler kann sich so mehr dem Systementwurf widmen und wird in die Lage versetzt, in kürzerer Zeit komplexere Schaltungen zu entwerfen. Die Realisierung digitaler Systeme mit FPGAs, die bis zu zwei Millionen Logikzellen[1] bieten [55], erfordert eine Beschreibung auf Basis komplexer Komponenten, wie z.B. ALUs (Arithmetic Logic Units), Steuerwerken und On-Chip RAM/ROM-Speicher.

Durch Nutzung von Top-Down bzw. Bottom-Up Entwurfsstilen sowie durch Einsatz wiederverwendbarer Komponenten lässt sich eine im Vergleich zum Schaltplanentwurf höhere Abstraktionsebene des Systementwurfs erreichen [16], [42].

---

[1] Logikzelle: Kombination aus einer Look-Up-Tabelle mit vier Eingängen und einem Daten-Flipflop

Als Modellierungssprachen für den beschriebenen Entwurfsstil haben sich weltweit die beiden Sprachen Verilog und VHDL etabliert, die den Markt in der Vergangenheit etwa zu gleichen Teilen dominierten [56]. Verilog war überwiegend im amerikanischen Markt vertreten, hat sich jedoch in Europa gegenüber VHDL weniger durchsetzen können. Seit Mitte der 90er Jahre ist auch in den USA ein erheblicher Anstieg der VHDL-Nutzung zu beob-achten [57]. Seit wenigen Jahren wird insbesondere für Anwendungen, bei denen es auf eine gemeinsame Hardware-Software-Modellierung ankommt, auch SystemC [68] zur Simulation bzw. Synthese digitaler Hardware eingesetzt. Dabei handelt es sich um eine standardisierte C++-Klassenbibliothek, für die es erst seit kurzem Synthesewerkzeuge gibt [69].

Aus den oben genannten Gründen und aufgrund der IEEE-Standardisierung sowie der daraus resultierenden Portierbarkeit der Entwicklungsergebnisse zwischen unterschiedlichen CAE-Umgebungen bzw. Bausteinen wurde die Sprache VHDL für die Lehre und das vorliegende Buch ausgewählt. Die in allen Hardwarebeschreibungssprachen verwendeten Konzepte sind jedoch recht ähnlich, womit das Erlernen einer weiteren Sprache nach Studium dieses Buches recht leicht fallen sollte. Einen Vergleich findet der Leser z.B. in [2].

VHDL wurde erstmalig 1987 und nachfolgend in den Jahren 1993, 2002 und 2008 durch die IEEE-Norm 1076 standardisiert [3]. Der synthesefähige Syntaxanteil von VHDL, der nur eine Teilmenge des gesamten Sprachumfangs darstellt, wurde 1999 durch die Norm 1076.6 standardisiert [20]. Die in den Synthesewerkzeugen implementierten Möglichkeiten gehen jedoch teilweise über diesen Standard hinaus. Aus diesem Grund wurde im vorliegenden Buch besonderer Wert darauf gelegt, nur eine sicher synthesefähige Untermenge zur Hardwarebeschreibung einzusetzen. Als Erprobungsgrundlage dienten überwiegend die PC-basierten Werkzeuge Aurora der Fa. Viewlogic [10], FPGA-Express der Fa. Synopsys [5] und XST der Fa. Xilinx [81].

1.2 Ziele und Organisation dieses Buches

Die besondere Aufgabe, die wir uns mit diesem Buch gestellt haben, besteht darin, die digitaltechnischen Entwurfselemente mit den VHDL-Modellierungselementen zusammenzuführen. Dabei werden folgende immer wiederkehrende Entwurfsschritte bearbeitet:

- Digitaltechnische Problembeschreibung
- Hardwarebeschreibung mit VHDL
- Verifikation des VHDL-Codes durch Simulation
- Analyse des Syntheseergebnisses.

Schrittweise erfolgt eine Steigerung der zu vermittelnden Syntaxkomplexität. Dies geschieht parallel zum wachsenden Funktionsumfang der modellierten Komponenten. So wird die Hardware-Modellierung mit VHDL aus Sicht der Digitaltechnik von einfachen kombinatorischen Logikfunktionen, Flipflops, gesteuerten Zählern und Zustandsautomaten bis zu komplexen Systemen wie einem RISC-Prozessor bzw. verschiedenen Implementierungsvarianten digitaler Filter dargestellt.

Aufgrund der Vielzahl möglicher VHDL-Beschreibungen für eine Problemstellung haben wir es angestrebt, jeweils wiederverwendbare und erweiterbare VHDL-Module vorzustellen. Dazu gehört auch, dass die Syntheseergebnisse als technologieunabhängige Schaltpläne sorgfältig aufbereitet und erläutert werden. Für den Leser soll sich dadurch als Lernziel ein geschulter Blick ergeben, mit dem er die plausible Übereinstimmung der synthetisierten Flipflops, Latches und Schaltnetze mit seiner Entwurfsvorstellung prüfen kann.

Das vorliegende Buch soll kein vollständiges VHDL-Syntaxlexikon darstellen, stattdessen wird ein synthesefähiger VHDL-Syntaxvorrat aus Sicht der digitaltechnischen Zielvorstellungen erlernt. Nicht synthesefähige Syntax wird als solche ausdrücklich erläutert und nur im Zusammenhang mit rein simulationstechnischen Aspekten genutzt. Dazu gehören zum einen die Formulierung von Stimuligeneratoren in VHDL-Testumgebungen und zum anderen die Modelle für nicht VHDL-basierte Schaltungsmakros. Die FPGA-Hersteller liefern diese optimierten Hardwaremakros, die bei der Synthese in die Netzlisten eingebunden werden können, in der Regel zusammen mit VHDL-Modellen, die das Verhalten dieser Schaltungsfunktionen nachbilden. Dabei reicht die Komplexität der Makros von einfachen RAM- und ROM-Speichern über spezielle mathematische Operationen bis hin zu IP-Blöcken (IP = Intellectual Property), die z.B. Bus-Interfaces oder parametrisierbare Filterstrukturen darstellen.

Die Simulation einer Hardwarebeschreibung ist im Lernprozess sowie in der praktischen Entwicklungstätigkeit ein entscheidender Schritt zur Verifikation eines Entwurfes. Zum Nachvollziehen der Beispielentwürfe des Buches sowie zur Überprüfung der eigenen Lösungen zu den Übungsaufgaben wird dringend empfohlen, einen VHDL-Simulator bzw. ein Synthesewerkzeug einzusetzen. Eine Einführung in die Benutzung von ModelSim bzw. ISE/Webpack findet der Leser auf der Homepage der Autoren.

Die einzelnen Kapitel dieses Lehrbuches lassen sich grob gefasst drei Hardware-Syntheseschwerpunkten zuordnen:

- Logik-Synthese,
- Schaltwerk-Synthese,
- Struktur-Synthese hierarchischer Systeme.

Die in der Literatur dafür benutzten Beschreibungsformen sind mit folgenden Begriffen klassifizierbar [13], [15], [42], [43]:

- Verhaltensbeschreibung: Modellierung des Ein-/Ausgangsverhaltens von digitaltechnischen Komponenten. Prinzipiell ist zu unterscheiden, ob die Modelle synthesefähig sind (Datenflussdarstellung), oder ob sie nur für die Simulation geeignet sind (algorithmische Darstellung).
- Strukturbeschreibung: Mit hierarchischen Strukturmodellen werden die Signalkopplungen zwischen den Komponenten beschrieben.

Im vorliegenden Lehrbuch bilden Hardwarebeschreibungen mit der Datenflussdarstellung den Kern der Ausführungen. Diese Hardware-Verhaltensbeschreibung wird in den IEEE-Standards als Darstellung auf der Register Transfer Ebene (RTL = Register Transfer Level) präzisiert [42], [43]. Darin werden die Schaltungen mit ihren Registern und der dazwischen liegenden kombinatorischen Logik beschrieben. Der VHDL-IEEE-Synthesestandard [20] wurde für RTL-Beschreibungen entwickelt.

Das Buch ist neben dieser Einführung in 9 weitere Kapitel und einen Anhang gegliedert, dabei werden die drei oben genannten Themenbereiche wie folgt behandelt: In den Kapiteln 2 bis 4 sollen mit elementaren VHDL-Konstrukten digitale Grundkomponenten realisiert werden. Darauf aufbauend erfolgt in zwei weiteren Kapiteln eine Darstellung des Entwurfs komplexerer Komponenten, wie z.B. von Komparatoren, Zählern und Automaten. Nach einer Vorstellung der Strukturierungsmethoden digitaler Systeme mit VHDL werden die bisher erlernten Konzepte an zwei völlig unterschiedlichen Beispielen umgesetzt: Einem einfachen RISC-Prozessor sowie Implementierungsvarianten digitaler Filter.

Der erste Themenbereich beginnt im Kapitel 2 mit der Vorstellung der wesentlichen syntaktischen Grundstrukturen eines VHDL-Codes zur Beschreibung einfacher Schaltnetze mit Entwurfseinheiten (Entities) und Architekturen. Dazu gehören auch Signale und deren nebenläufige Wertzuweisung. Schon in diesem einführenden Kapitel wird der Leser auf Möglichkeiten hingewiesen, den eigenen Entwurf mittels geeigneter VHDL-Testumgebungen zu simulieren.

Zur Erweiterung der allein mit nebenläufigen Anweisungen stark eingeschränkten Modellierungsmöglichkeiten wird in Kapitel 3 die Beschreibung von Schaltnetzen und Schaltwerkelementen mit Prozessen und den zugehörigen sequentiellen Anweisungen erläutert. Diese erlauben zunächst Verzweigungen, die entweder Multiplexer oder Prioritätsencoder generieren. Schleifenkonstrukte können entweder der Erzeugung einer endlichen Anzahl regelmäßiger Hardwareelemente oder aber der Abarbeitung getakteter Vorgänge dienen. Dazu werden Paritätsgeneratoren vorgestellt, bei denen auch auf den Unterschied zwischen VHDL-Signalen und -Variablen eingegangen wird. Als Beispiele einfacher Schaltwerke werden D-Flipflops und Register sowie ein Zähler mit Schaltwerktabelle und ein parametrisiertes Schieberegister entworfen. Die speziellen Möglichkeiten, die der Einsatz von Prozessen in Testumgebungen liefert, werden ebenfalls in diesem Kapitel aufgezeigt.

In Kapitel 4 werden mehrwertige Datentypen eingeführt, mit denen sich über die einfache 0-1-Darstellung digitaler Signale hinaus, unter anderem hochohmige Buszustände (Tri-State-Treiber) sowie Don't-Care-Einträge in Wahrheitstabellen umsetzen lassen.

Der zweite Themenbereich, der die Modellierung komplexerer Komponenten vorstellt, beginnt in Kapitel 5 mit der Einführung arithmetischer Operatoren, die für die mehrwertigen Datentypen definiert sind. Darauf aufbauend wird der Entwurf von Komparatoren, Addierern und Subtrahierern erläutert. Auch einfache Schaltwerke wie gesteuerte Zähler werden hier behandelt. Ergänzend wird in diesem Kapitel der Integer-Datentyp eingeführt, mit dem sich besonders vorteilhaft indizierte Feldzugriffe, wie sie z.B. bei der Speicheradressierung erforderlich sind, beschreiben lassen.

Die Umsetzung der aus der Digitaltechnik bekannten Automatenstrukturen behandelt Kapitel 6. Dabei wird am Beispiel eines Automaten zur seriellen Sequenzerkennung auf geeignete Abbildungsvarianten der Automatenkomponenten auf VHDL-Prozesse und die Darstellung der Zustandscodierung eingegangen. Die Realisierung von Signalsynchronisationen und des Komponentenpipelinings zur Taktfrequenzerhöhung in Systemen mit gekoppelten Automaten wird ebenfalls vorgestellt.

Im dritten Schwerpunkt werden in Kapitel 7 die wesentlichen Schritte zum VHDL-Entwurf digitaler Systeme vorgestellt. Dazu gehört neben der Vorstellung einer Entwurfsmethodik die Abbildung auf Strukturmodelle, die durch gekoppelte Komponenten und Netzlisten

repräsentiert werden. Als weitere Strukturierungsmittel werden in diesem Kapitel Unterprogramme erläutert. Als Ausblick wird die Einbindung herstellerspezifischer, parametrisierbarer Schaltungsmakros in den VHDL-Code diskutiert.

In den Kapiteln 8 und 9 werden zwei Beispiele für digitale Systeme mit unterschiedlichem Schwerpunkt vorgestellt: Während beim schrittweisen Entwurf eines RISC-Prozessors mit vierstufiger Pipeline im Kapitel 8 die Methodik des Systementwurfs im Vordergrund steht, wird im Kapitel 9 besonderer Wert auf die Simulations- und Synthesesemantik von Entwurfsvarianten digitaler FIR- und IIR-Filter gelegt. Dabei erfolgt eine vergleichende Bewertung des Implementierungsaufwands dieser Strukturen.

Das Kapitel 10 behandelt die Modellierung von gekoppelten Zustandsdifferenzengleichungen, wie sie in Beobachtersystemen der Regelungstechnik genutzt werden. Die Grundlagen digitaler Integrationsverfahren für lineare und nichtlineare Systembeschreibungen sowie die Architektursynthese von Einzyklus-, Mehrzyklus- und von Pipeline-Modellen sind Gegenstand dieses weiterführenden Kapitels.

Insbesondere diese drei Kapitel gehen über die üblicherweise in der Digitaltechnik gelehrten Lehrinhalte hinaus und richten sich an Studierende höherer Semester.

Am Ende der Kapitel 2 bis 7 ist jeweils eine Reihe von Übungsaufgaben zur Entwicklung digitaler Grundkomponenten zusammengestellt.

Der Anhang enthält neben einer Musterlösung zu den Übungsaufgaben auch komprimierte Hinweise mit Codierungsempfehlungen, die einen synthesegerechten VHDL-Entwurf ermöglichen, eine Checkliste zur Systematik des Entwurfs digitaler Systeme unter Verwendung von VHDL sowie eine Liste der VHDL-Schlüsselworte.

Dateien mit Quellcodes für Beispiellösungen sowie weitere Informationen zu diesem Buch finden Sie unter dem URL: http://users.etech.haw-hamburg.de/users/reichardt/ und auf den Internetseiten des Verlags (www.oldenbourg-wissenschaftsverlag.de > VHDL-Synthese > Zusatzmaterial).

1.3 Syntaxnotation

Anhand zweier exemplarischer VHDL-Syntaxbeschreibungen soll kurz auf die in diesem Buch verwendete Bezeichnungsweise eingegangen werden:

```
entity <Entityname> is
        [generic(<Deklaration von Parametern>);]
port(<Deklaration der Ein- und Ausgänge>);
        [< Entitydeklarationen >;]
end <Entityname>;

architecture <Architekturname> of <Entityname> is
        [< Architekturdeklarationen >;]
begin
        {<VHDL-Anweisungen>;}
end <Architekturname>;
```

- Grundsätzlich wird VHDL-Code bzw. die VHDL-Syntaxbeschreibung auch im Fließtext als `Courier Font` dargestellt.
- VHDL-Schlüsselworte bzw. -Trennzeichen sind **fett** gedruckt.
- Elemente in spitzen Klammern stellen Bezeichner oder andere, untergeordnete Elemente der Syntaxbeschreibung dar (z.B. <Entitydeklarationen>).
- Die in eckigen Klammern angegebenen Syntaxelemente sind optional und werden in den einführenden Beispielen weggelassen (z.B. [**generic**( );])
- In geschweiften Klammern { } angegebene Syntaxkonstrukte können beliebig oft wiederholt werden.
- Elemente einer Liste alternativer Möglichkeiten von Syntaxkonstrukten werden durch den Vertikalstrich | getrennt.

Während die ersten fünf Notationselemente im oben angegebenen Beispiel wieder zu finden sind, stellt die nachfolgende Syntaxbeschreibung ein Beispiel für das letzte Element dar:

```
for <Marke>| others | all:<Komponenten-Name>
use entity <Bibliothek>.<Entityname>(<Architekturname>);
```

Eine vollständige Syntaxübersicht findet der Leser im Kap. 12, in dem eine Zusammenstellung der Fa. Qualis [58] reproduziert ist. Unter dem URL: http://users.etech.haw-hamburg.de/users/schubert/vorles.html lässt sich eine deutschsprachige VHDL-Syntaxbeschreibung im PDF-Format über das Internet herunter laden.

2 Synthese einfacher Schaltnetze

In diesem Kapitel werden die grundlegenden VHDL-Konzepte erläutert. Ausgehend von vollständigen Beispielen wird der VHDL-Sprachschatz schrittweise, problemorientiert erweitert. Dabei werden zunächst die Begriffe `entity` und `architecture` erklärt, die die Schnittstellen einer Logikeinheit bzw. deren Funktion beschreiben. Es werden Signale definiert, denen ein digitaler Wert mit nebenläufigen Anweisungen zugewiesen wird. Am Ende des Kapitels wird erläutert, wie die VHDL-Entwürfe mittels geeigneter Testumgebungen simuliert werden können.

Nach dem Durcharbeiten soll der Leser in der Lage sein, synthesefähige VHDL-Beschreibungen für einfache kombinatorische Schaltungen auf Gatterebene entwerfen zu können. Dazu gehört zunächst nur die korrekte Verwendung der Datentypen `bit` und `bit_vector` sowie der problemorientierte Umgang mit den `port`-Modi `in, out` und `buffer`. Die syntaktischen und semantischen Unterschiede der bedingten und selektiven Signalzuweisungen sollen verstanden sein.

Die Funktion des `inout port`-Modus wird später im Kapitel 4 erläutert, da zugehörige praktische Anwendungen den Datentyp `std_logic_vector` erfordern, der erst in diesem Kapitel eingeführt wird.

2.1 Entity, Architektur und Signale

Die in der nachfolgenden Liste aufgeführten VHDL-Elemente sind als grundlegende Strukturelemente jeder VHDL-Beschreibung anzusehen:

- In der mit `entity` bezeichneten Entwurfseinheit werden die Schnittstellen eines VHDL-Funktionsblocks nach außen beschrieben. In einem Vergleich eines Board-Designs mit einem VHDL-Quellcode stellt eine `entity` den zu bestückenden IC-Gehäusetyp dar, der durch die Anzahl und die Bezeichnung der Anschlüsse eindeutig definiert ist. Die Deklaration der Anschlüsse innerhalb der `entity` erfolgt mit Hilfe einer `port`-Anweisung.
- Die `architecture` beschreibt das Innenleben, d.h. die Funktionalität des VHDL-Codes. Jeder `entity` muss (mindestens) eine `architecture` zugeordnet sein. In obiger Vorstellung beschreibt also die Architektur, welche Funktion bzw. welcher Chip sich in dem Gehäuse befindet.
- Ein `port`-Signal beschreibt die Kommunikation einer `entity` nach außen. In unserem Modell werden `port`-Signale verwendet, um verschiedene integrierte Schaltkreise

(ICs) auf einem Board (Platine) miteinander zu verknüpfen. Innerhalb einer Architektur werden lokale Signale erforderlich, wenn verschiedene Funktionsblöcke innerhalb eines ICs miteinander verbunden werden sollen. Signale können also auf unterschiedlichen Hierarchieebenen definiert werden.

Jedes Signal ist von einem eindeutig zu definierenden Typ. Zur Einführung soll hier zunächst nur der Datentyp `bit` bzw. `bit_vector` benutzt werden. Der Wertevorrat des Datentyps `bit` besteht aus den logischen Werten '0' und '1'. Ein `bit_vector` stellt einen aus mehreren `bit`-Signalen bestehenden Bus dar. Dieser Bus kann entweder aufsteigend, z.B. als `bit_vector(0 to 7)`, oder aber abfallend als `bit_vector (7 downto 0)` bezeichnet werden.

Alle Signale müssen einen eindeutigen Namen besitzen, der entweder in der `entity` oder in der weiter unten erläuterten Signaldeklarationsanweisung (Abschnitt 2.1.1) festgelegt wird. Die Schnittstellen der `entity` werden als `port` bezeichnet. Dessen Signalrichtung ist bei der Deklaration anzugeben. Die Syntax eines `entity`/`architecture`-Paars besteht aus den beiden Rahmen:

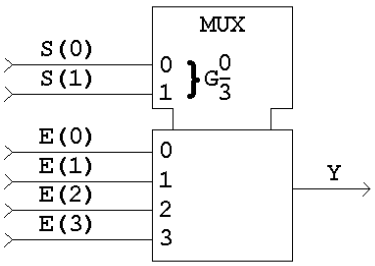
```
entity <Entityname> is
       [generic(<Deklaration von Parametern>);]
       port(<Deklaration der Ein- und Ausgänge>);
[< Entitydeklarationen >;]
end <Entityname>;

architecture <Architekturname> of <Entityname> is
[< Architekturdeklarationen >;]
begin
       {<VHDL-Anweisungen>;}
end  <Architekturname>;
```

Als erstes Beispiel zeigt Code 2-1 die vollständige VHDL-Beschreibung eines Multiplexers mit vier Dateneingängen nach Bild 2-1.

```
-- mux4x1.vhd
-- Selektive Signalzuweisung
----------------------------
entity MUX4X1 is
       port(  S: in bit_vector(1 downto 0);
              E: in bit_vector(3 downto 0);
              Y: out bit);
end MUX4X1;

architecture VERHALTEN of MUX4X1 is
begin
       with S select  -- Auswahlsignal
       Y <=   E(0) when "00",
              E(1) when "01",
              E(2) when "10",
              E(3) when "11";
end VERHALTEN;
```



Code 2-1: Beschreibung eines 4 zu 1 Multiplexers

Bild 2-1: Symbol des 4 zu 1 Multiplexers

An diesem Code 2-1 sollen die ersten typischen Eigenschaften von VHDL-Codes erläutert werden:

- Kommentare beginnen an beliebiger Stelle einer Zeile mit „ -- " und enden am Ende der Zeile.
- Groß-/Kleinschreibung wird prinzipiell ignoriert. Allerdings wird in Anlehnung an [1] folgende Konvention verwendet, die sich insbesondere in der neueren VHDL-Literatur mehr und mehr durchsetzt:
- VHDL-Schlüsselworte werden klein geschrieben
- Eigendefinitionen wie z.B. Signale etc. werden groß geschrieben.
- Namen und Bezeichner (identifier) müssen mit einem Buchstaben beginnen. Die nachfolgenden Zeichen können Buchstaben, Ziffern oder aber der Unterstrich „ _ “ sein. Die Länge des Namens ist prinzipiell beliebig. Eine eindeutige Identifizierung erfolgt jedoch bei den meisten CAE-Werkzeugen über eine Länge von maximal 32 Zeichen.
- Vollständige VHDL-Anweisungen werden mit einem „ ; “ abgeschlossen. Zum Trennen von Teilen einer VHDL-Anweisung dienen entweder spezielle Schlüsselworte (z.B. `with select when`) oder aber, in einigen Syntaxkonstrukten, auch ein Komma bzw. ein Doppelpunkt. Anweisungen können über mehr als eine Zeile verteilt werden.
- Die Zuweisung eines Signalwerts erfolgt durch den Operator „<=“ . Dem Signal Y wird abhängig vom Auswahlsignal S eins der einzelnen Bits E(0), E(1), E(2) oder E(3) zugewiesen (selektive Signalzuweisung). Man spricht von einer unbedingten Signalzuweisung (unconditional signal assignment), wenn die Zuweisung unabhängig vom Zustand anderer Signale erfolgt. Ein Beispiel für eine unbedingte Signalzuweisung ist:

  ```
  Y <= E(0);
  ```

- Die Zuweisung bzw. Abfrage von Signalwerten vom Typ `bit` erfolgt durch Klammerung in Apostroph. So wird z.B. in nachfolgender Anweisung dem Signal Y ein Low-Pegel ('0') zugewiesen wenn E(3) einen High-Pegel ('1') besitzt:

  ```
  Y <= '0' when E(3)='1' else ....;
  ```

- Hingegen erfolgt die Zuweisung von `bit_vector`-Konstanten durch Einbettung in Anführungszeichen: Dem Signal E wird durch

  ```
  E <= "1010";
  ```

  die angegebene Bitfolge zugewiesen, die auch als Bit-String (bit string literal) bezeichnet wird. Identisch dazu ist die folgende alternative Form der Signalzuweisung, in der den einzelnen `bit_vector`-Elementen ein `bit`-Signalwert zugewiesen wird.

  ```
  E <= ('1','0','1','0');
  ```

  Diese Art der bitweisen Zusammenfassung von Signalen oder Konstanten wird als Aggregat bezeichnet.

Die Angabe der Signalflussrichtung in der `port`-Anweisung der `entity` ist für die Verwendung von Signalen der `architecture` entscheidend: Signale, die mit dem `port`-Modus `in` gekennzeichnet sind, dürfen in den VHDL-Anweisungen der Architektur grundsätzlich nur abgefragt bzw. mit anderen Signalen verknüpft werden, d.h. in einer Signalzu-

weisung nur auf der rechten Seite stehen. Hingegen müssen Signale, die mit dem Schlüsselwort `out` gekennzeichnet sind, bei einer Signalzuweisung auf der linken Seite stehen. Derartige Signale dürfen nicht auf Ihren Wert abgefragt werden. Der nachfolgende Code 2-2 enthält entsprechende Fehler:

```
-- Beispiel mit Fehlern
---------------------------
entity MUX4X1 is
       port(   S: in bit;
               E: in bit_vector(1 downto 0);
               Y: out bit);
end MUX4X1;

architecture VERHALTEN of MUX4X1 is
begin
       with Y select          -- Fehler: Abfrage auf out-Signal
       S <=    E(0) when '0', -- Fehler: Zuweisung auf in-Signal
               E(1) when '1';
end VERHALTEN;
```

Code 2-2: Fehlerhafte Verwendung von `port`-Signalen

2.1.1 Deklaration und Verwendung lokaler Signale

Innerhalb einer Architektur können Signale auch lokal definiert werden. Diese internen Signale dienen zum Informationsaustausch zwischen Funktionselementen und Blöcken innerhalb einer `architecture`, sie bieten jedoch nicht die Möglichkeit, mit anderen Entities Informationen auszutauschen. Lokale Signale sind innerhalb der `architecture` vor dem `begin` mit Hilfe des Schlüsselworts `signal` zu deklarieren. In einer solchen Deklarationszeile interner Signale lassen sich auch mehrere Signale gleichen Typs deklarieren (z.B. die Signale B und Y in Code 2-3).

Bei Signalzuweisungen ist grundsätzlich zu beachten, dass in der Regel nur eine Quelle einem Signal einen Wert zuweisen darf. Der nachfolgende VHDL-Codeauszug ist somit in dieser Hinsicht fehlerhaft:

```
architecture VERHALTEN of TEST is
signal A: bit_vector(0 to 3);
signal B, Y: bit_vector( 3 downto 0);

begin
       A <= "1010";
       B <= "0101";
       Y <= A after 100 ns;  -- Erste Signalquelle
       Y <= B after 200 ns;  -- Fehler: 2. Signalquelle
end VERHALTEN;
```

Code 2-3: Fehlerhafte VHDL-Beschreibung mit zwei Signalquellen für das Signal y

In Hardware würde eine derartige Zuweisung eine verdrahtete (wired-) Verknüpfung bedeuten, die nur für Open-Collector- bzw. Tri-State-Signale erlaubt ist. Derartige Signaltypen werden in Kap. 4.1 näher erläutert.

2.1.2 Richtungsmodi von Signalschnittstellen

Einfache Ein- bzw. Ausgangssignale sind zur Beschreibung digitaler Systeme in vielen Fällen nicht ausreichend. Zur Realisierung von Signalrückführungen ist es erforderlich, ein nach außen zu führendes Signal auch intern weiter zu verwenden. So einen Fall stellt z.B. die Kreuzkopplung zweier NOR-Gatter in einem RS-Latch dar. Die intern wieder verwendeten Signale müssen mit dem Schlüsselwort `buffer` bezeichnet werden.

Als weitere Variante kennt die Digitaltechnik bidirektionale Signale, wie z.B. die Datensignale eines RAM-Speichers, die gelesen und geschrieben werden. Für derartige Signale muss das Schlüsselwort `inout` verwendet werden. Dafür ist ein spezieller Datentyp erforderlich (vgl. Kap. 4.1). In der Tabelle 2-1 sind die Eigenschaften der `port`-Modi zusammengestellt.

Bild 2-2 zeigt die dazugehörigen Schaltplandarstellungen.

`port`-Modus	**Verwendung**
`in`	Eingangssignal; darf nur auf der rechten Seite einer Signalzuweisung oder in einer Signalabfrage stehen.
`out`	Ausgangssignal; darf nur auf der linken Seite einer Signalzuweisung stehen.
`buffer`	Ausgangssignal; darf auch auf der rechten Seite einer Signalzuweisung stehen oder in einer Signalabfrage verwendet werden.
`inout`	Bidirektionales Signal in Verbindung mit speziellem Datentyp `std_logic` (vgl. Kap. 4).

Tabelle 2-1: `port`-Modi für Signalschnittstellen

Von der Verwendung der `buffer`-Schnittstellen raten wir jedoch ab, da deren unterschiedliche Handhabung durch Simulations- und Synthesewerkzeuge in strukturellen Entwürfen zu inkonsistenten Aussagen führen kann. Vielmehr empfehlen wir, die `buffer`-Funktionalität dadurch zu realisieren, dass ein internes Signal definiert wird, welches gelesen und geschrieben werden darf. Das interne Signal wird mit einer unbedingten Signalzuweisung auf das Ausgangssignal kopiert.

Der Code 2-4 zeigt die entsprechende Realisierung bei einem RS-Latch: Die Signale Q_INT und NQ_INT, die der Signalrückkopplung dienen, werden auf der linken und rechten Seite von Signalzuweisungen verwendet. Als `port`-Signale müssten diese den `buffer` `port`-Modus haben. In der in Code 2-4 vorgestellten Architektur sind diese jedoch als lokale Signale definiert und werden den `out`-Signalen Q bzw. NQ zugewiesen. Das zu erwartende Syntheseergebnis zeigt Bild 2-3.

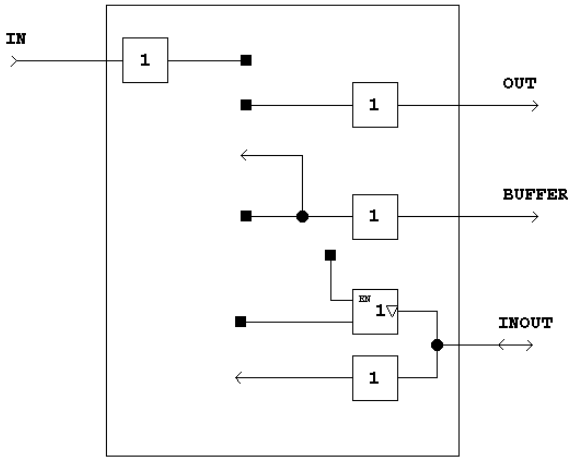


Bild 2-2: Ersatzschaltbilder der verschiedenen VHDL-Signalschnittstellentypen (`port`-Modi)

```
-- RS-Latch mit interner Signalrueckkopplung
-------------------------------------------------
entity RS_L is
       port(   R, S: in bit;
               Q, NQ: out bit);
end RS_L;

architecture VERHALTEN of RS_L is
signal Q_INT, NQ_INT: bit;    -- lokale Signale
begin
       NQ_INT <= S nor Q_INT;
       Q_INT  <= R nor NQ_INT;
       Q  <= Q_INT;           -- Uebergabe des lokalen Signals
       NQ <= NQ_INT;          -- Uebergabe des lokalen Signals
end VERHALTEN;
```

Code 2-4: Vermeidung von `buffer`-Schnittstellen durch Verwendung lokaler Signale

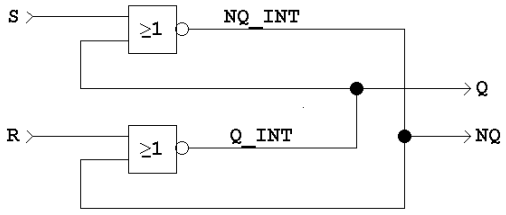


Bild 2-3: Syntheseergebnis des in Code 2-4 dargestellten RS-Latches

Wegen der Verwechslungsgefahr soll der Unterschied zwischen `buffer` und `inout` noch einmal herausgestellt werden: `inout`-Signale sollten überall dort verwendet werden, wo es einen bidirektionalen Datentransfer der `entity` gibt (vgl. Kap. 4.2). Hingegen können `buffer`-Signale überall dort deklariert werden, wo ein Signal nach außen geleitet werden muss, dieses Signal aber auch innerhalb der Architektur verwendet wird, d.h. auf der rechten Seite einer Signalzuweisung bzw. in einer Signalabfrage steht.

2.1.3 Deklaration von Bussignalen

VHDL-Busse sind auf zweierlei Weise, nämlich in absteigender und in aufsteigender Wertigkeit darstellbar. Die Zuordnung einzelner Bits zu `bit_vector`-Signalen erfolgt so, dass der zuerst genannte Index der Vereinbarung dem am weitesten links stehenden Vektorelement und der letztgenannte Index dem am weitesten rechts stehenden Element zugeordnet wird. Dies soll am Beispiel der Zuweisung

```
S <= "01";
```

erläutert werden. Bei absteigender Deklaration gilt durch die Vereinbarung:

`S: in bit_vector(1 downto 0)` `S(1)='0'` und `S(0)='1'`.

Hingegen gilt bei aufsteigender Deklaration:

`S: in bit_vector(0 to 1)` `S(1)='1'` und `S(0)='0'`.

Wenn die in der Digitaltechnik übliche duale Gewichtung der einzelnen Bitstellen gewünscht ist, muss also eine absteigende Vereinbarung eines `bit_vector`-Signals gewählt werden.

Ein abschließender Hinweis zur Bussignalzuweisung: Falls allen Bitstellen der gleiche Wert zugewiesen werden soll, so ist es sinnvoll, das `others`-Konstrukt zu verwenden, da dieses unabhängig von der Breite des Bit-Vektors ist und somit das Nachzählen der zugewiesenen Bitstellen entfällt. Im nachfolgenden Codeauszug wird das Signal A mit 0xFFFF und Q mit 0x0000 belegt:

```
...
signal A,Q: bit_vector(15 downto 0);
...
       A <= (others => '1');
       Q <= (others => '0');
...
```

2.2 Simulation von VHDL-Entwürfen

Im vorangegangenen Abschnitt wurden bereits erste vollständige VHDL-Entwürfe vorgestellt. Ziel dieses Abschnitts ist es, dem Leser eine einfache Methode an die Hand zu geben, mit der eigene Entwürfe syntaktisch und funktional überprüft werden können. Da das Erlernen der Sprache VHDL deutlich erleichtert wird, wenn eigene Entwürfe sofort auf Korrektheit überprüft werden, soll nun der Entwurf einfacher VHDL-Simulationsumgebungen erläutert werden. Diese können vom Leser u.a. mit Hilfe des VHDL-Simulators ModelSim

[59] bzw. mit dem Xilinx Paket ISE/Webpack [4] auf einem PC selbst entworfen bzw. synthetisiert werden.

Da die am Markt befindlichen VHDL-Werkzeuge unterschiedliche Syntax zur interaktiven Eingabe der Stimuli verwenden, soll hier der prinzipielle Aufbau einer einfachen Testumgebung (Testbench) nur in Form eines Schemas erläutert werden. Dafür soll auf einige Inhalte des Kap. 7 vorgegriffen werden:

Alle Entwurfseinheiten, also auch die Testbench, können zusammen in einer Datei mit der Dateierweiterung *.VHD abgespeichert werden. Einen übersichtlicheren Entwurf erhält man jedoch, wenn die einzelnen `entity/architecture`-Paare jeweils in einer eigenen Datei abgelegt werden. Nach erfolgreicher Compilation der Datei, die das zu analysierende Modell enthält, wird dessen Objektcode in einer VHDL-Bibliothek mit dem vordefinierten Namen WORK abgelegt. Anschließend muss eine Datei angelegt und compiliert werden, die die `entity` und `architecture` der Testbench enthält.

Diese `entity` enthält keinerlei Schnittstellensignale. Die Schnittstellensignale des zu untersuchenden Modells werden vielmehr als lokale Signale der Testbench-`architecture` deklariert. Diese enthält eine Komponente, die der zu untersuchenden `entity` entspricht und an die die lokalen Testbenchsignale übergeben werden. Die Definition des Zeitverlaufs der Stimuli erfolgt durch nebenläufige Anweisungen, in der die Zeitpunkte der Signalübergänge festgelegt sind. Der syntaktische Aufbau einer Testbench sieht wie folgt aus:

```
entity TEST is
end TEST;
architecture TESTBENCH of TEST is
signal <Signalliste>;          -- Alle Schnittstellensignale
                               -- der zu untersuchenden entity
component <Komponentenname>    -- Identisch mit Entityname der
                               -- zu untersuchenden entity
          <Port-Liste>         -- Identisch mit der Port-Liste
                               -- der zu untersuchenden entity
end component;
for all: <Komponentenname> use entity work.<Entityname>(<Architekturname>);
begin
               -- Nebenläufige Signalzuweisungen an die Stimulisignale
               -- mit Angabe des Zeitpunkts der Signalübergänge
[<Bezeichner>]: <Komponentenname>
               port map( < Liste der angeschlossenen aktuellen Signale>);
end TESTBENCH;
```

Als Beispiel ist in Code 2-5 eine Testbench für den in Code 2-1 vorgestellten Multiplexer angegeben. Die darin deklarierten Signale S1, E1 und Y1 werden als aktuelle Signale an die als Komponente C1 platzierte `entity` MUX4X1 übergeben. Zum Zeitpunkt t=0 wird der Eingangsvektor E1 mit "1010" und zum Zeitpunkt t=400 ns mit "0101" vorbelegt. Das Auswahlsignal S1 durchläuft mit einer Impulsdauer von 100 ns zweimal die Sequenzen "00", "01", "10" und "11", womit sich eine Simulationsdauer von 800 ns ergibt. Bild 2-4 zeigt den Zeitverlauf dieser Eingangssignale sowie die Reaktion des Multiplexers, dessen Ausgang auf das Signal Y1 abgebildet wird.

```
-- Testbench fuer 4x1 Multiplexer
---------------------------------
entity TEST is
end TEST;

architecture VERHALTEN of TEST is
signal S1: bit_vector(1 downto 0);
signal E1: bit_vector(3 downto 0);
signal Y1: bit;

component MUX4X1
       port(   S: in bit_vector(1 downto 0);
               E: in bit_vector(3 downto 0);
               Y: out bit);
end component;
for all: MUX4X1 use entity work.MUX4X1(VERHALTEN);

begin
       E1 <= "1010", "0101" after 400 ns;
       S1 <= "00","01" after 100 ns,"10" after 200 ns,"11" after 300 ns,
                 "00" after 400 ns,"01" after 500 ns, "10" after 600 ns,
                 "11" after 700 ns;
C1:    MUX4X1 port map(S1, E1, Y1);
end VERHALTEN;
```

Code 2-5: Testbench für den in Code 2-1 vorgestellten 4×1-Multiplexer

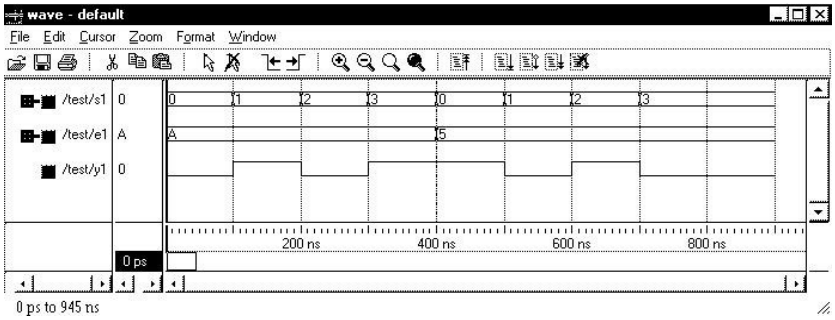


Bild 2-4: Simulation der Testbench zum 4×1-Multiplexer in Code 2-1

Mit den in diesem Abschnitt vermittelten Informationen können nun die meisten Beispiele mit den Programmen ModelSim [59] bzw. ISE / Webpack [4] praktisch erprobt werden. Sollten dafür andere Zeiteinheiten erforderlich sein, so sind die in VHDL definierten Einheiten zu verwenden:

fs	ps	ns	us	ms	sec	min	hr
10^{-15} s	10^{-12} s	10^{-9} s	10^{-6} s	10^{-3} s	1 s	60 s	3600 s

Tabelle 2-2: In VHDL vordefinierte Zeiteinheiten

Bei deren Verwendung ist unbedingt zu beachten, dass zwischen dem Zahlenwert und der Einheit ein Leerzeichen stehen muss!

Der wesentliche Vorteil einer rein VHDL-basierten Testumgebung liegt in der Unabhängigkeit von der Kommandosprache eines speziellen Simulators und in der einheitlichen Beschreibung von Testobjekt und Stimuliquellen. Eine entscheidende Qualitätssteigerung für den Test komplexer Entwürfe entsteht dadurch, dass die Testumgebung um Funktionen erweitert werden kann, die ergänzend zu den Stimuli einen Soll-Ist-Vergleich der Ausgangssignale durchführen. Da eine eingehende Darstellung dieser Möglichkeiten den Rahmen dieses Buches sprengen würde, soll auf weiterführende Literatur verwiesen werden (z.B. [6], [7], [15], [16], [37], [42], [43]).

2.3 Schaltnetze mit Boole'schen Gleichungen

Die VHDL-Syntax unterstützt verschiedene Boole'sche Operatoren. Diese sind für Signale vom Typ `bit` sowie für Signale vom Typ `boolean` definiert. Letzterer Datentyp kann die Werte `true` oder `false` annehmen. Er dient in der VHDL-Praxis vorwiegend dazu, architekturspezifische, logische Ausdrücke zu erzeugen, weniger dazu, die elektrischen Pegel von Signalen zu erzeugen. Ein Beispiel für die Verwendung dieses Datentyps ist dem nachfolgenden Codeauszug zu entnehmen:

```
architecture VERHALTEN of CHECK is
signal ERROR_FLG, GLEICH : boolean;
signal A, B : bit;

begin
.....
       ERROR_FLG <= false;
       GLEICH <= ( A = B );
.....
end VERHALTEN;
```

Das Synthesewerkzeug ersetzt Boole'sche Operatoren entsprechend den in der Synthesebibliothek vorhandenen Modulen. Üblicherweise sind dies die Basisfunktionen Inverter sowie AND-, NAND-, OR- und NOR-Gatter mit unterschiedlicher Anzahl von Eingängen. Manchmal sind auch Antivalenz-, seltener Äquivalenzgatter vorhanden. Nicht vorhandene Gatterfunktionen werden durch geeignete Kombinationen von digitalen Basisfunktionen ersetzt [90]. Anhand eines Volladdierers wird die Verwendung Boole'scher Funktionen verdeutlicht: In den logischen Gleichungen bezeichnen A_I und B_I die zu addierenden Eingangsbits, C_{IN} das Übertragsbit der vorhergehenden Stufe, S_I den Summenausgang und C_{OUT} das Übertragsbit [8]:

$$S_I = A_I \leftarrow\!+\!\rightarrow B_I \quad \leftarrow\!+\!\rightarrow \quad C_{IN}$$

$$C_{OUT} = (A_I \wedge B_I) \vee (C_{IN} \wedge (A_I \quad \leftarrow\!+\!\rightarrow B_I))$$

Die VHDL-Realisierung des Volladdierers ist in Code 2-6 dargestellt. Das Bild 2-5 enthält den zugehörigen, von einem Synthesewerkzeug automatisch erstellten Schaltplan.

```
-- Volladdierer
----------------------------
entity VOLLADD is
       port(  AI, BI, CIN: in bit;
              SI, COUT: out bit);
end VOLLADD;

architecture VERHALTEN of VOLLADD is
begin
       SI  <= AI xor BI xor CIN;
       COUT <= (AI and BI) or (CIN and (AI xor BI));
end VERHALTEN;
```

Code 2-6: VHDL-Beschreibung eines Volladdierers

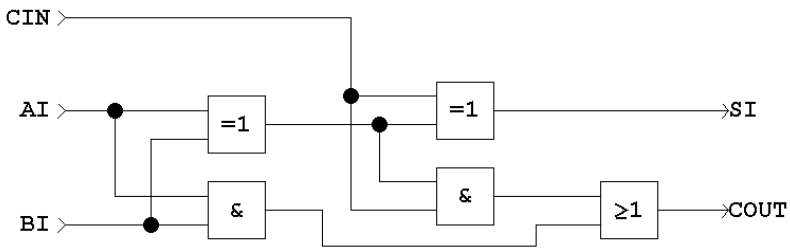


Bild 2-5: Syntheseergebnis des Volladdierers

Der Schaltplan zeigt, dass die beiden Ausgangssignale SI und COUT gleichzeitig generiert werden. Im VHDL-Code entspricht dies den beiden Signalzuweisungen der Architektur. Offensichtlich werden die VHDL-Signalzuweisungen also gleichzeitig bearbeitet. Man spricht von nebenläufigen Signalzuweisungen (concurrent signal assignment). Dieses Verhalten steht im Gegensatz zur Abarbeitung von Programmzeilen üblicher Programmiersprachen, wie z.B. C, bei denen die Quellcodezeilen nacheinander ausgewertet werden. Für VHDL-Einsteiger mit Programmiererfahrung führt das nebenläufige Verhalten von Signalzuweisungen häufig zu Verständnisproblemen. Für Hardware-Entwickler ist dieses Verhalten jedoch nicht überraschend, denn die beiden Signalzuweisungen des VHDL-Codes werden bei der Hardwaresynthese durch logische Gatter ersetzt, die zu jedem Zeitpunkt Signalübertragungen durchführen.

In diesem Zusammenhang soll darauf hingewiesen werden, dass die tatsächliche Implementierung der Schaltung technologiespezifisch meist von der Darstellung im Schaltplan abweicht: Bei (C)PLDs findet eine Umwandlung in eine zweistufige UND-ODER-Logik

statt, bei vielen FPGAs werden die logischen Gleichungen hingegen in Wahrheitstabellen (Look-Up-Tabellen) umgesetzt.

Eine Übersicht zu den in VHDL verfügbaren logischen Operatoren gibt

Tabelle 2-3. Mit Ausnahme der Negation verknüpfen diese Operatoren jeweils zwei Operanden. In dieser Liste hat die Negation die höchste Priorität. Die Priorität der übrigen Operatoren in Verknüpfungen mit mehreren Signalen sollte durch Klammerung sichergestellt werden. Die Klammern können bei den assoziativen Operatoren `and`,`or` und `xor` weggelassen werden, wenn die Operation auf mehr als zwei Operanden angewendet werden soll. Insbesondere muss bei den nichtassoziativen Operatoren `nand`, `nor` und `xnor` die Ausführungsreihenfolge durch Klammerung festgelegt werden. Die folgenden Beispiele machen dies deutlich:

```
Y <= not ( A and B and C); -- entspricht einem NAND3
Y <= (A nand B) nand C;    -- legal, ist aber kein NAND3
Y <= A nand B nand C;      -- VERBOTEN !!!
```

Operator	Bedeutung	Beispiel
not	Negation	`Y <= not A;`
and	UND-Verknüpfung	`Y <= A and B;`
nand	NAND-Verknüpfung	`Y <= A nand B;`
or	ODER-Verknüpfung	`Y <= A or B;`
nor	NOR-Verknüpfung	`Y <= A nor B;`
xor	Antivalenz	`Y <= A xor B;`
xnor	Äquivalenz	`Y <= A xnor B;`

Tabelle 2-3: Boole'sche Operatoren in VHDL

Die in Tabelle 2-3 angegebenen Operatoren lassen sich nicht nur auf `bit`-Signale anwenden, sondern auch auf Signale vom Typ `bit_vector`. Die Verknüpfung wird in diesem Fall für alle Bitstellen ausgeführt. Dazu muss die Bitbreite der zu verknüpfenden Operanden sowie die des Ergebnisvektors gleich sein. Sofern bei Signalverknüpfungen die Breite der Bit-Vektoren unterschiedlich ist, kann diese durch Angabe eines Indexbereichs in Zusammenhang mit den Schlüsselworten `to` bzw. `downto` angepasst werden (vgl. Code 2-7).

```
entity TEST is
       port( R : out bit_vector( 3 downto 0));
end TEST;

architecture VERHALTEN of TEST is
signal A, B, C: bit_vector(7 downto 0);
signal Q: bit_vector(3 downto 0);
```

```
begin
        A <= "10101010";
        B <= "01010101";
        C <= A or B;
        Q <= A(3 downto 0);          -- unteres Nibble
        R <= Q and B(7 downto 4);    -- Verknuepfung zweier Nibbles
end VERHALTEN;
```

Code 2-7: Logische Verknüpfungen von Bit-Vektoren in VHDL

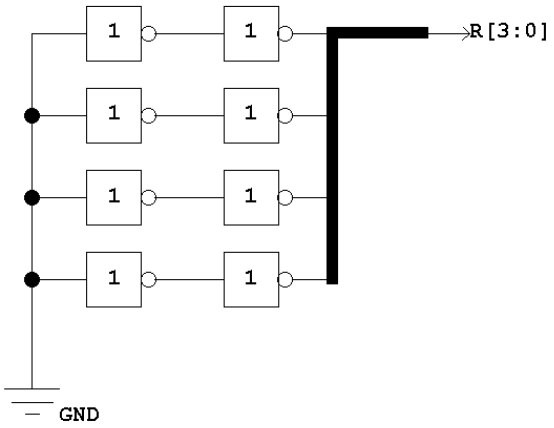


Bild 2-6: Syntheseergebnis der `architecture` VERHALTEN der `entity` TEST

Das Ergebnis der Synthese zeigt weiterhin, dass die Signale A, B, C und Q durch Substitution komplett entfernt wurden. Außerdem wurde erkannt, dass alle Bits des Busses R statisch '0' sind. Die Tatsache, dass trotzdem jeweils zwei Inverter in Reihe verwendet wurden, zeigt die Grenzen der Minimierungsfähigkeit des Synthesewerkzeugs. Dieses ist offensichtlich davon ausgegangen, der Entwickler hätte sich bei dem VHDL-Code etwas sinnvolleres gedacht.

2.4 Synthese selektiver und bedingter Signalzuweisungen

Die selektive Signalzuweisung (selective signal assignment) wurde im einführenden Beispiel eines Multiplexers in Kap. 2.1 bereits verwendet. Funktional gesehen entspricht die selektive Signalzuweisung einer Auswahl aus einer Reihe von gleichberechtigten Möglichkeiten, also einer Multiplexer-Struktur. Sofern geeignete Multiplexer in der Entwurfsbibliothek zur Verfügung stehen, erscheint das Selektionsergebnis am Ausgang der Schaltung somit unabhängig von der Auswahl immer nach genau einer Signallaufzeit.

Die Syntax der selektiven Signalzuweisung lautet wie folgt:

```
[ Bezeichner : ]
with <Signal-Kombination> select
<Signalname> <=       <logischer Ausdruck_1> when <Sig_Wert_1> ,
                      [<logischer Ausdruck_2> when <Sig_Wert_2>],
                                    ..............
                      [<logischer Ausdruck_n> when others ];
```

Darin ist der Bezeichner optional. Er dient in den Simulations- bzw. Synthesewerkzeugen zur Kennzeichnung von hierarchischen Strukturen bzw. Signalen. Insbesondere in komplexen Entwürfen sollten Bezeichner verwendet werden, um eine gute Übersichtlichkeit sicherzustellen. Die „`when others`“ Verzweigung ist immer dann zwingend erforderlich, wenn die Anzahl der zuvor angegebenen Kombinationen der Signalwerte kleiner ist, als die Anzahl der prinzipiell möglichen Kombinationen.

Die bedingte Signalzuweisung (conditional signal assignment*)* kann in einigen Anwendungsfällen eine Alternative zur selektiven Signalzuweisung darstellen. Die Funktion dieser Anweisung ist eine Verschachtelung von `if then else` -Strukturen, wie sie aus den meisten Programmiersprachen bekannt sind. Im Unterschied zur selektiven Signalzuweisung werden die einzelnen Bedingungen jedoch in der spezifizierten Reihenfolge abgefragt, was in der Hardware zu einer Schachtelung von Gatterstufen führt (Prioritätsencoder). Die Signallaufzeit durch diese Struktur hängt somit davon ab, an welcher Stelle eine bestimmte Abfrage steht: Die Priorität der „Bedingung_1“ ist höher, als die der nachfolgenden Bedingungen. Entsprechend wird das Ergebnis des ersten logischen Ausdrucks dem Ausgangssignal früher zugewiesen, als das der nachfolgenden logischen Ausdrücke, denn deren Auswertung erfordert Signallaufzeiten durch weitere logische Stufen. Die Syntax der bedingten Signalzuweisung lautet wie folgt:

```
[ Bezeichner : ]
       < Signalname > <= < logischer Ausdruck_1 >
                                          when < Bedingung_1> else
                              [< logischer Ausdruck_2 >
                                          when < Bedingung_2> else]
                                    ..............
                              < logischer Ausdruck_n >;
```

Im Unterschied zur selektiven Signalzuweisung, bei der letztlich eine einzige Bedingung auf ihre verschiedenen Möglichkeiten abgefragt wird, können bei einer bedingten Signalzuweisung prinzipiell auch völlig unterschiedliche Signale bzw. deren Kombinationen in den einzelnen Bedingungen verwendet werden.

Einige Anwendungen der selektiven und bedingten Signalzuweisung werden in den nachfolgenden Beispielen vorgestellt. In allen Fällen werden in einer VHDL-Quelldatei beide Anweisungstypen in jeweils einer eigenen Architektur vorgestellt. Ebenso wie es möglich ist, eine vorgegebene Schaltungsfunktionalität mit verschiedenen Hardware-Architekturen zu realisieren, so lassen sich auch verschiedene VHDL-Architekturen in einer Datei angeben. Sofern keine besondere Konfigurationsdatei existiert, wird bei der Simulation bzw.

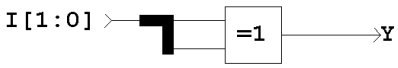
Synthese üblicherweise die an letzter Stelle stehende Architektur verwendet. In den Beispielen wird durch beide Architekturvarianten jeweils die gleiche Schaltung synthetisiert.

Im ersten Beispiel wird ein Antivalenzgatter mit Hilfe der beiden Anweisungsarten entworfen (vgl. Code 2-8). Zum Syntheseergebnis ist zu bemerken, dass in Entwurfsbibliotheken, in denen kein Antivalenzgatter existiert, dieses durch eine UND-ODER-Kombination ersetzt wird.

```
-- Antivalenz mit selektiver und bedingter Signalzuweisung
-------------------------------------------------------------
entity ANTIVALENZ is
       port( I : in bit_vector( 1 downto 0);        -- 2 Dateneingaenge
             Y : out bit);                          -- Ausgang
       end ANTIVALENZ;

architecture ANTI_1 of ANTIVALENZ is
begin
       with I select
       Y <=    '0' when "00",
               '0' when "11",
               '1' when others;
end ANTI_1;

architecture ANTI_2 of ANTIVALENZ is
begin
       Y <=    '0' when I = "00" else
               '0' when I = "11" else
               '1';
end ANTI_2;
```



Code 2-8: Synthese eines Antivalenzgatters mit selektiver und bedingter Signalzuweisung

Das zweite Beispiel erläutert den Entwurf eines 1 aus 2 Multiplexers mit Freigabeeingang (vgl. Bild 2-7).

E	S	I1	I0	Y
1	X	X	X	0
0	0	X	0	0
0	0	X	1	1
0	1	0	X	0
0	1	1	X	1

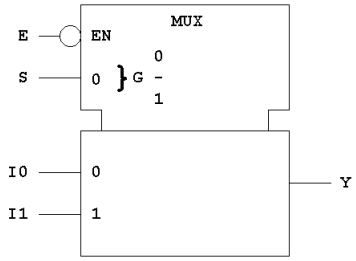


Bild 2-7: Wahrheitstabelle und Schaltsymbol eines 2 zu 1 Multiplexers

Der dazugehörige VHDL-Code 2-9 enthält drei Architekturvarianten: Die Architektur MUX1 bildet die Wahrheitstabelle mit Boole'schen Funktionen ab. In der Architektur MUX2 wird der Multiplexer mit einer selektiven Signalzuweisung und in der Architektur MUX3 mit der bedingten Signalzuweisung realisiert.

```
-- Multiplexer 2 zu 1 mit Freigabeeingang
----------------------------------------
entity MUX2ZU1 is
       port( I: in bit_vector (1 downto 0);
             S, E : in bit; --Select und Enable Eingaenge
             Y: out bit);
       end MUX2ZU1;

architecture MUX1 of MUX2ZU1 is
begin
       Y <= (I(0) and not E and not S)
             or (I(1) and not E and S);
end MUX1;

architecture MUX2 of MUX2ZU1 is
begin
       with S select
       Y <=   (I(0) and not E) when '0',
              (I(1) and not E) when '1';
end MUX2;

architecture MUX3 of MUX2ZU1 is
begin
       Y <=   (I(0) and not E) when S='0' else
              (I(1) and not E);
end MUX3;
```

Code 2-9: Beschreibungsvarianten eines 2 zu 1 Multiplexers

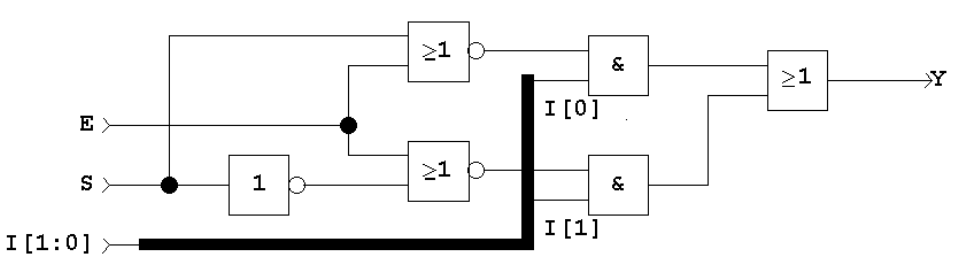


Bild 2-8: Syntheseergebnis des 2 zu 1 Multiplexers

Der synthetisierte Schaltplan in Bild 2-8 zeigt die durch die bedingte Signalzuweisung resultierende Priorität durch unterschiedliche Signalverzögerungen: Der Signalpfad, der den oberen Eingang des letzten ODER-Gatters definiert, entspricht dem Ausgangssignal falls die erste Bedingung (also S='0') gültig ist. Aus $\neg S \wedge \neg E \wedge I(0)$ wird das DeMorgan-Äquivalent synthetisiert: $\neg(S \vee E) \wedge I(0)$. Zusammen mit dem letzten ODER-Gatter, welches die beiden Fälle S='0' und S='1' zusammenfasst, also eine dreistufige Logik. Der geringer prioritisierte `else`-Zweig erfordert hingegen vier Gatterstufen.

Bei diesem Beispiel erfolgt die Prioritisierung über das Steuersignal S. In weiter gefächerten Verzweigungen, bei denen insbesondere auch die Datensignale I abgefragt werden,

sollten diese Abfragen an oberster Stelle stehen, um einen möglichst kurzen Datensignalpfad zu garantieren.

In diesem Zusammenhang soll noch einmal betont werden, dass das Syntheseergebnis in starkem Maße von der verwendeten Design-Bibliothek abhängt. Falls andere Entwurfsbibliotheken verwendet werden, die Logikstrukturen mit invertierten Eingängen und mehr als zwei Gattereingänge besitzen, kann aus dem gleichen VHDL-Quellcode ein anderer Schaltplan realisiert werden.

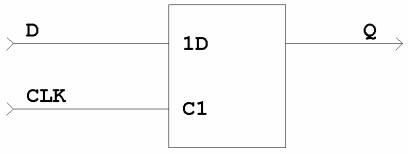
Im konkreten Fall wurde eine (C)PLD-Bibliothek verwendet. Da derartige programmierbare Bausteine eine zweistufige UND-ODER-Logikstruktur besitzen, ist für die Hardware-Implementierung eine der eigentlichen VHDL-Synthese nachfolgende Logikminimierung erforderlich. Diese wiederum kann das aufgrund des Schaltplans zu erwartende Zeitverhalten verändern.

Wie mit Hilfe der selektiven bzw. bedingten Signalzuweisung ein Speicherverhalten der resultierenden Schaltung erzeugt werden kann, soll in einem weiteren Beispiel erläutert werden. Dazu wird in Code 2-10 das Ausgangssignal ausnahmsweise als „`buffer`“ definiert, damit es in einer Rückführung verwendet werden kann. Durch die Verwendung des Ausgangssignals Q als Eingangssignal wird das Speicherverhalten erzeugt: In beiden Architekturen wird das Eingangssignal D an den Ausgang weitergereicht, solange der Takt CLK den Wert '1' hat. Falls das Taktsignal '0' ist, wird der alte Wert Q zugewiesen, also gespeichert. Bei der so beschriebenen Funktionalität handelt es sich also um ein D-Latch (zustandsgesteuertes D-Flipflop). Das neben dem Quellcode angegebene Schaltsymbol wird durch beide Architekturen synthetisiert.

```
-- Latch mit selektiver bzw. bedingter Signalzuweisung
-------------------------------------------------------
entity LATCH is
       port( D, CLK: in bit;
             Q: buffer bit);
end LATCH;

architecture LATCH1 of LATCH is
begin
       Q <= D when CLK = '1' else Q;
end LATCH1;

architecture LATCH2 of LATCH is
begin
       with CLK select
       Q <=   D when '1',
              Q when others;
end LATCH2;
```



Code 2-10: Synthese eines D-Latches unter Verwendung der bedingten und selektiven Signalzuweisung

Zusammenfassend sollen die Unterschiede zwischen den beiden Anweisungsarten noch einmal herausgestellt werden:

- In der selektiven Signalzuweisung werden Antwortkombinationen einer einzelnen Abfrage, d.h. einer einzelnen logischen Bedingung, getestet. Sie bietet sich also überall dort an, wo ein und dieselbe Kombination von Eingangssignalen zu testen ist (z.B. Multiplexer und Decoder).
- In der bedingten Signalzuweisung werden in den einzelnen Abfragen jeweils unterschiedliche Bedingungen getestet. Sie ist daher besonders in den Fällen geeignet, in denen unterschiedliche Kombinationen von Eingangssignalen relevant sind. Durch die Reihenfolge der Abfragen ist automatisch eine Priorität vorgegeben. Eine geschickt gewählte Reihenfolge kann ggf. zu erheblich reduziertem Hardwareaufwand und einer Laufzeitoptimierung spezieller Signalpfade führen. Dazu ist die am häufigsten auftretende bzw. die zeitkritischste Bedingung in den ersten `if-then`-Pfad zu stellen.

2.5 Übungsaufgaben

2.1

Welche der folgenden VHDL-Bezeichner sind korrekt?

a) Hilfe

b) help

c) 2ter_Versuch

d) Case

e) Zweiter_Versuch

f) Dieser_Bezeichner_ist_lang_aber_ist_er_auch_gueltig

2.2

Die Sprache VHDL verlangt, dass alle Bezeichner innerhalb einer Entwurfseinheit unterschiedlich sind. Für welche der folgenden Bezeichnerpaare trifft dies nicht zu?

a) Mein_Name , MeinName

b) nummer , NUMMER

c) Nummer , Nummern

d) two , too

2.3

Entwerfen Sie den VHDL-Code eines Schaltnetzes, welches die beiden logischen Funktionen Y1=(E1 $\wedge$ E2) $\vee$ E3 und Y2=(E1 $\vee$ E2) $\wedge$ E3 darstellt. Überprüfen Sie dessen syntaktische Korrektheit mit einem VHDL-Simulator.

2.4

Entwerfen Sie den VHDL-Code eines Drei-Bit-Decoders (Demultiplexers). Verwenden Sie in jeweils einer `architecture` eine bedingte und eine selektive Signalzuweisung. Von den acht Bit des Decoder-Ausgangs soll jeweils nur das adressierte Bit '1' sein. Alle anderen Bits sind '0'. Überprüfen Sie dessen syntaktische Korrektheit mit einem VHDL-Simulator.

2.5

Entwerfen Sie ein VHDL-Schaltnetz mit dem Eingang E, welches durch den Steuereingang S programmierbar ist: Für S=0 soll der Ausgang Y=E sein, für S=1 soll Y=¬E sein. Überprüfen Sie die syntaktische Korrektheit und simulieren Sie das Verhalten nachdem Sie eine geeignete Testbench entworfen haben. Wie lautet die logische Gleichung, die ISE / Webpack bei der PLD-Synthese generiert?

Hinweis: Stellen Sie in ISE / Webpack ein: Device-Family = XC9500 CPLDs, Device = Auto xc9500, implementieren Sie den Code und entnehmen Sie die logische Gleichung dem „Fitter Report“ im Abschnitt „Equations“.

2.6

Entwerfen Sie eine Look-Up-Tabelle zur Steuerung logischer Funktionen: Abhängig von einem zwei Bit breiten Steuersignal soll das Ausgangssignal Y vier unterschiedliche logische Kombinationen zweier Eingangssignale A und B darstellen können:

Überprüfen Sie die syntaktische Korrektheit und simulieren Sie das Verhalten nachdem Sie eine geeignete Testbench entworfen haben. Wie lautet die logische Gleichung, die ISE / Webpack bei der PLD-Synthese generiert?

S(1)	S(0)	Y
0	0	$A \wedge B$
0	1	$A \vee B$
1	0	$\neg (A \wedge B)$
1	1	$\neg (A \vee B)$

2.7

Entwerfen Sie einen 1 zu 4 Demultiplexer [36]: Das Eingangsbit E soll abhängig vom Selektionssignal SEL auf einen von vier Ausgängen geschaltet werden. Die anderen Ausgänge sollen '0' sein. Verwenden Sie in jeweils einer `architecture` eine bedingte und eine selektive Signalzuweisung. Simulieren Sie jeweils das Verhalten der beiden Architekturen und vergleichen Sie die Syntheseergebnisse, die sich bei einer PLD-Implementierung ergeben.

3 Entwurf digitaler Funktionselemente mit Prozessen

Die im letzten Kapitel vorgestellten nebenläufigen Anweisungen erlauben nur eine stark eingeschränkte Möglichkeit der Modellierung. Aus diesem Grund soll in diesem Kapitel als neues Strukturierungselement zur Beschreibung von Schaltnetzen und Schaltwerkelementen das Syntaxkonstrukt `process` eingeführt werden. Mit den zugehörigen sequentiellen Anweisungen, die Verzweigungen und Schleifen erlauben, und den deklarierbaren Variablen wird eine neue Klasse von synthesefähigen Modellierungsmöglichkeiten eröffnet. Als Beispiele werden wieder einfache Schaltnetze wie Multiplexer, Volladdierer Paritätsgeneratoren sowie D-Flipflops und Zähler auf Basis von Schaltwerktabellen vorgestellt.

Mit den in diesem Kapitel vorgestellten VHDL-Syntaxelementen soll der Entwickler in die Lage versetzt werden, mit Prozessen komplexere Digitalentwürfe zu realisieren. Dazu gehört neben der korrekten Aktivierung der Prozesse auch die richtige Verwendung von Signalen und Variablen. Der Unterschied zwischen nebenläufigen und sequentiellen Anweisungen soll ebenso verstanden werden, wie die aus den sequentiellen Anweisungen resultierenden Hardwarestrukturen.

Da mit Prozessen neben kombinatorischer und sequentieller (taktgesteuerter) Logik eine Vielzahl nur simulationsfähiger Zusammenhänge realisierbar ist, besteht das Ziel, dem Leser die Einschränkungen der Prozess-Syntax zu erläutern, die bei der VHDL-Synthese zu den gewünschten Digitaltechnikelementen führen.

3.1 Prozesse

Prozesse stellen ergänzend zu den nebenläufigen Anweisungen ein mächtiges Gestaltungsmittel dar, digitale Hardwarefunktionalität zu realisieren. Dabei werden Eingangssignalvektoren auf Ausgangssignalvektoren abgebildet. Alle Prozesse einer `architecture` werden nebenläufig ausgeführt.

Da synthetisierbare Prozesse auf Hardwarekomponenten zurückgeführt werden, ist zur Unterscheidung kombinatorischer und getakteter (sequentieller) Hardware die Einhaltung einiger Regeln erforderlich. In der VHDL-Praxis bedeutet dies, dass im Quellcode nur jeweils eine Teilmenge der VHDL-Syntax verwendet werden kann.

Die Kommunikation zwischen verschiedenen Prozessen und eventuell vorhandenen nebenläufigen Anweisungen, also den digitalen Funktionsblöcken, erfolgt mit lokalen Signalen einer `architecture`.

Für die Verwendung innerhalb von Prozessen wurde eine eigene Klasse von Anweisungen definiert (sequential statements). Eine besondere Eigenschaft der sequentiellen Anweisungen ist die Tatsache, dass diese im Simulator, wie in einer Programmiersprache, nacheinander abgearbeitet werden. Im Synthesewerkzeug erfolgt hingegen eine Abbildung der einzelnen sequentiellen Anweisungen auf die bekannten Hardwarefunktionselemente. Aus diesem Grund ist in der synthetisierten Hardware die Ausführungszeit natürlich endlich langsam, obwohl der Simulator den Prozess in infinitesimal kurzer Zeit abarbeitet. Insbesondere lassen sich mit speziellen sequentiellen Konstrukten auch getaktete Funktionselemente beschreiben, die zeitliche Signalabfolgen definieren.

Neben den sequentiellen Anweisungen sind innerhalb von Prozessen unbedingte Signalzuweisungen erlaubt, hingegen sind die selektiven und bedingten Signalzuweisungen (vgl. Kap. 2.4) verboten. Bei der Verwendung der unbedingten Signalzuweisung muss jedoch berücksichtigt werden, dass die tatsächliche Aktualisierung aller Signale immer am Prozessende erfolgt. Während der Prozessausführung ist es also **nicht** möglich auf die aktuellen Werte bereits ausgeführter Signalzuweisungen zuzugreifen. Einem Signal können in einem Prozess mehrere unterschiedliche Werte zugewiesen werden. Der tatsächlich übernommene Signalwert entspricht dem der zuletzt ausgeführten Signalzuweisung im Prozess. Insbesondere der Zeitpunkt der Signalaktualisierung am Prozessende bereitet VHDL-Anfängern, die ausschließlich Erfahrungen mit imperativen Programmiersprachen wie C oder PASCAL haben, häufig Schwierigkeiten, da in diesen Sprachen die Wertzuweisungen in genau der Reihenfolge ausgeführt werden, wie sie im Programmtext erscheinen [28].

Wenn innerhalb eines Prozesses auf einen gerade aktualisierten Wert zugegriffen werden soll, so sind Variablen zu verwenden. Eine `variable` wird innerhalb des Prozesses mit Angabe des Datentyps deklariert und besitzt nur für diesen Prozess Gültigkeit. Mit Variablen werden Zwischenergebnisse innerhalb eines Prozesses bis zum nächsten Prozessdurchlauf gespeichert, sie können abgefragt und ggf. modifiziert werden. Wenn der Wert von Variablen in anderen Prozessen verwendet werden soll, so ist die Variable in ein Signal zu kopieren.

Am Ende dieser Einführung in Prozesse soll auf ein mögliches Verständnisproblem hingewiesen werden, welches bei der Doppelverwendung des Begriffs „sequentiell“ auftauchen könnte: Mit Hilfe sequentieller Anweisungen in einem `process` kann einerseits rein kombinatorische Logik, wie auch andererseits sequentielle Logik aufgebaut werden. Letztere enthält im Unterschied zur kombinatorischen Logik speichernde Bauelemente, wie z.B. Latches oder Flipflops.

3.1.1 Deklaration und Ausführung von Prozessen

VHDL-Prozesse stellen als Ergänzung zu den nebenläufigen Anweisungen ein zusätzliches Strukturierungselement dar. Die Deklaration von Prozessen erfolgt innerhalb einer `architecture` entsprechend der nachfolgenden Syntax:

```
[<Prozessname>:] process [( <Empfindlichkeitsliste> )]
       <Deklarationsteil>
begin
       {<sequentielle Anweisungen>}
end process [<Prozessname>]
```

Darin bedeuten:

- `<Prozessname>` ein Bezeichner, der den VHDL-Prozess eindeutig kennzeichnet. Es empfiehlt sich, alle Prozesse mit Namen zu versehen, damit bei der Simulation das Debuggen des VHDL-Quellcodes erleichtert wird.
- `<Empfindlichkeitsliste>` (sensitivity list) eine geklammerte Liste von Signalen, die durch Kommata getrennt werden. Die Signale in der Empfindlichkeitsliste starten die Bearbeitung des Prozesses durch den Simulator bzw. sie aktivieren die durch den Prozess abgebildete Hardware. Die Empfindlichkeitsliste kann in einigen Fällen weggelassen werden (s. unten).
- `<Deklarationsteil>` eine Liste von Deklarationen. In Prozessen können insbesondere auch Variable deklariert werden (neben Konstanten und selbstdefinierten Datentypen). Variable sind nur innerhalb des Prozesses gültig, in dem sie deklariert wurden. Als Datentypen für Variable kommen alle Typen in Frage, die auch für Signale erlaubt sind. Der Wert einer Variablen wird sofort nach einer Zuweisung innerhalb des Prozesses verändert. Ähnlich wie die Variablen von Programmiersprachen. Wenn der Prozess verlassen wird, bleibt der Variablenwert bis zum nächsten Durchlauf erhalten. Signale können in Prozessen nicht deklariert werden, allerdings sind alle lokalen Signale der zugehörigen `architecture` sowie die der `port`-Schnittstelle verwendbar.
- Innerhalb eines Prozesses sind nur sequentielle Anweisungen sowie unbedingte Signalzuweisungen erlaubt. Die Liste der Anweisungen, die die eigentliche Funktionalität des Prozesses beschreiben, wird durch die Schlüsselworte `begin` und `end process` geklammert. Falls der Prozess mit Namen bezeichnet wurde, muss dieser Name hinter `end process` ab VHDL-2002 nicht mehr wiederholt werden.

Zwei Prozessvarianten werden nach der Art ihrer Aktivierung und Ausführung durch den Simulator unterschieden:

- Prozesse **mit Empfindlichkeitsliste** werden immer dann aktiviert, wenn eins der in der Empfindlichkeitsliste aufgeführten Signale ein Ereignis, also die Änderung des Signalwertes aufweist. Derartige Prozesse werden immer bis zur `end process` -Anweisung ausgeführt: Alle Signaländerungen, die bei der Ausführung der sequentiellen Anweisungen durchzuführen sind, werden erst am Prozessende aktualisiert. Anschließend sind diese Prozesse bis zur nächsten Aktivierung inaktiv.
- Prozesse **ohne Empfindlichkeitsliste** müssen, wenn sie synthesefähig sein sollen, (mindestens) eine `wait until`-Anweisung enthalten, in der eine Taktflankenabfrage erfolgt. Diese dient zur Synchronisation. Derartige Prozesse werden automatisch bis zur (nächsten) `wait`-Anweisung ausgeführt. Die dabei auszuführenden Signaländerungen erfolgen erst zu diesem Zeitpunkt. Sollte sich die `wait`-Anweisung in der Mitte eines Prozesses befinden, so fängt diese Art von Prozessen automatisch erneut mit der Aus-

führung an, bis die `wait`-Anweisung erreicht ist. Prozesse, die nicht synthetisiert werden sollen, können auch durch eine `wait for`-Anweisung synchronisiert werden.

Eine Mischung beider Varianten ist nicht erlaubt.

Prozesse mit Empfindlichkeitsliste sind in ihrer Funktionalität meist überschaubarer und können von Synthesewerkzeugen leichter in Hardware umgesetzt werden. Prozesse, die in mehreren `wait`-Anweisungen unterschiedliche Signale bzw. Flanken überprüfen, sind von den zur Zeit auf dem Markt befindlichen Synthesewerkzeugen nicht synthetisierbar [6]. Die kürzlich vom IEEE beschlossenen Syntheserichtlinien [20] gehen sogar noch weiter: Demnach darf ein Prozess höchstens eine `wait`-Anweisung erhalten, die außerdem zu Beginn des Prozesses stehen muss. Aus diesem Grunde wird dem VHDL-Anfänger empfohlen, Prozesse so zu formulieren, dass eine Empfindlichkeitsliste verwendet werden kann.

Mit den weiter unten vorgestellten sequentiellen Anweisungen kann jede nebenläufige Signalzuweisung in einen äquivalenten Prozess umgewandelt werden, wobei die auf der rechten Seite und in den Signalabfragen stehenden Signale in die Empfindlichkeitsliste aufzunehmen sind.

Als einführendes Beispiel soll die Schaltung der in Bild 3-1 dargestellten Ausgangsmakrozelle (Output Logic Macro Cell OLMC) entworfen werden, die in ähnlicher Form in vielen (C)PLDs verwendet wird [30], [52].

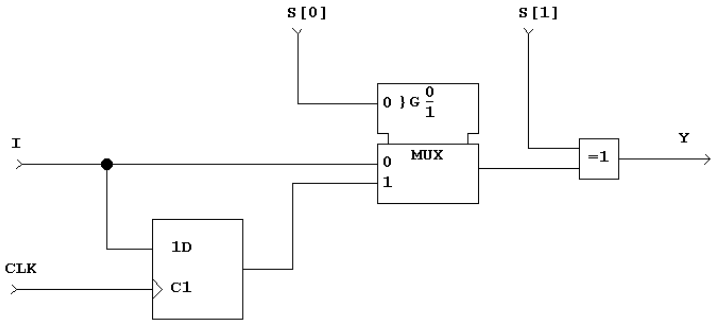


Bild 3-1: Schaltplan einer einfachen (C)PLD-Ausgangsmakrozelle

Abhängig vom Steuersignal S(0) wird das Signal I entweder direkt oder aber taktflankengesteuert auf das Antivalenzgatter gegeben. Falls das Steuersignal S(1)='0' ist, wird der Multiplexerausgang an den Ausgang Y weitergeleitet. Für S(1)='1' erscheint das Signal invertiert am Ausgang.

In der Architektur ARCH1 in Code 3-1 werden die in Bild 3-1 dargestellten drei Funktionselemente Multiplexer, D-Flipflop und Antivalenzgatter durch einen taktunabhängigen Prozess MUX, einen taktflankenabhängigen Prozess FF sowie eine nebenläufige Anweisung dargestellt.

```
-- Einfache OLMC mit 2 Prozessen und nebenl. Anweisung
-----------------------------------------------------
entity OLMC is
       port(  CLK, I: in bit;               -- Dateneingang
              S: in bit_vector(1 downto 0); -- Selektionseingang
              Y: out bit);                  -- Ausgang
end OLMC;

architecture ARCH1 of OLMC is
signal TEMP1, TEMP2: bit;
begin
MUX: process(I, TEMP1, S(0))
       begin
              TEMP2 <= (I and not S(0)) or (TEMP1 and S(0));
       end process MUX;
FF:  process
       begin
              wait until CLK='1' and CLK'event;
              TEMP1 <= I;
       end process FF;
     Y <= TEMP2 xor S(1);
end ARCH1;
```

Code 3-1: VHDL-Beschreibung einer einfachen Ausgangsmakrozelle eines (C)PLDs

An diesem Quellcode sollen die folgenden Dinge herausgestellt werden:

- Die Kommunikation zwischen den Prozessen bzw. der nebenläufigen Anweisung erfolgt mit den Signalen TEMP1 und TEMP2
- Im kombinatorischen Prozess MUX stehen alle Signale, die in der Signalzuweisung verwendet werden, in der Empfindlichkeitsliste.
- Die ansteigende Taktflanke des in Bild 3-1 dargestellten D-Flipflops wird im Prozess FF durch die Formulierung `CLK='1' and CLK'event` beschrieben. Entsprechend ließe sich eine abfallende Flanke durch `CLK='0' and CLK'event` beschreiben. Das in diesen Formulierungen verwendete Signalattribut `'event` ist Bestandteil des VHDL-Sprachschatzes und bezeichnet einen beliebigen Signalwechsel.

Dieses Beispiel zeigt, wie einfach sich Schaltplanvorgaben auf VHDL-Prozesse bzw. nebenläufige Anweisungen abbilden lassen. Natürlich hätte man auch auf den Prozess MUX verzichten können und diesen gegen eine nebenläufige Anweisung austauschen können. Allgemein gilt, dass jede nebenläufige Anweisung durch einen äquivalenten Prozess ersetzt werden kann. In umgekehrter Richtung gilt dies jedoch nicht.

3.2 Schaltnetze mit sequentiellen Anweisungen

Zur Modellierung des Zeitverhaltens sowie für Verzweigungen und Schleifen existiert eine Gruppe von sequentiellen Anweisungen, die nur innerhalb von Prozessen verwendet werden können. Dazu gehören:

- die `case`-Anweisung
- die `if`-Anweisung
- Schleifenkonstrukte (loops):
  - die `for loop`
  - die `while loop`
- die `wait`-Anweisung
- die `null`-Anweisung

Diese Anweisungen bieten eine enorme Vielfalt von Modellierungsmöglichkeiten, die während der Synthese auf eine relativ geringe Zahl digitaler Grundstrukturen abgebildet werden muss. Aus diesem Grund sollen hier zunächst nur die sequentiellen Syntaxkonstrukte erläutert werden, die zu synthesefähigen Schaltnetzen führen.

3.2.1 Die case-Anweisung

Ähnlich wie die switch/case-Anweisung in C existiert in VHDL eine Anweisung, die eine gleichberechtigte Mehrfachverzweigung zulässt. Die `case`-Anweisung ist wie folgt zu verwenden:

```
case <Kontrollausdruck> is
        when <Testausdruck_1> => {<Sequentielle Anweisungen>;}
        [when <Testausdruck_2> => {<Sequentielle Anweisungen>;}]
        ...
        [when others => {<Sequentielle Anweisungen>;}]
end case;
```

Darin bedeutet `<Kontrollausdruck>` ein Signal oder eine Variable. Die Testausdrücke müssen einen erlaubten Wert des Kontrollausdrucks darstellen. Alle Testausdrücke müssen sich gegenseitig ausschließen, d.h. kein Testausdruck darf mehr als einmal in einer `when`-Verzweigung berücksichtigt werden. Weiterhin müssen alle möglichen Werte des Kontrollausdrucks in den `when`-Zweigen berücksichtigt sein. Falls dies in den expliziten Testausdrücken sichergestellt ist, so kann auf die `when others`-Verzweigung am Ende der `case`-Anweisung verzichtet werden. Am Beispiel des in Kap 2.1 vorgestellten 4 zu 1 Multiplexers soll die Verwendung demonstriert werden:

```
-- MUX4x1 mit case-Anweisung
--------------------------
entity MUX4X1_2 is
       port(   S: in bit_vector(1 downto 0);
               E: in bit_vector(3 downto 0);
               Y: out bit);
end MUX4X1_2;

architecture VERHALTEN of MUX4X1_2 is
begin
MUXPROC: process(S, E)
       begin
              case S is
                     when "00" => Y <= E(0);
                     when "01" => Y <= E(1);
                     when "10" => Y <= E(2);
                     when "11" => Y <= E(3);
              end case;
       end process MUXPROC;
end VERHALTEN;
```

Code 3-2: Realisierung eines Multiplexers mit Hilfe einer `case`-Anweisung

Im nachfolgenden Beispiel soll die Verwendung der `case`-Anweisung beim Aufbau einer Wahrheitstabelle vorgestellt werden. Als praktisches Beispiel wird die in

Tabelle 3-1 angegebene Wahrheitstabelle eines 1-Bit-Volladdierers mit Carry-Eingang CIN und den Summanden AI und BI gewählt.

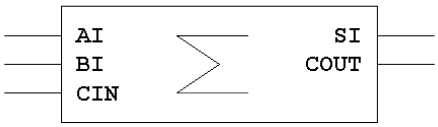


Bild 3-2: Schaltsymbol eines Volladdierers

CIN	BI	AI	COUT	SI
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabelle 3-1: Wahrheitstabelle eines Volladdierers

```
-- Volladdierer mit Wahrheitstabelle als case-Anweisung
-------------------------------------------------------
entity VOLLADD_TAB is
       port(   CIN, AI, BI: in bit;
               SI, COUT: out bit);
end VOLLADD_TAB;

architecture ARCH1 of VOLLADD_TAB is
begin

P1: process(CIN, BI, AI)
```

```
        variable TEMP_IN: bit_vector(2 downto 0);    -- temporaerer Vektor
        variable TEMP_OUT: bit_vector(1 downto 0);   -- temporaerer Vektor
        begin
                TEMP_IN := CIN & BI & AI;
                case TEMP_IN is
                        when "000" =>  TEMP_OUT := "00";
                        when "011" =>  TEMP_OUT := "10";
                        when "101" =>  TEMP_OUT := "10";
                        when "110" =>  TEMP_OUT := "10";
                        when "111" =>  TEMP_OUT := "11";
                        when others => TEMP_OUT := "01";
                end case;
                SI <= TEMP_OUT(0);                    -- Kopie ins Summenbit
                COUT   <= TEMP_OUT(1);                -- Kopie ins Carry-Bit
        end process P1;
end ARCH1;
```

Code 3-3: Beschreibung eines Volladdierers durch eine VHDL-Wahrheitstabelle

Besonders praktisch ist die Darstellung der Wahrheitstabelle, wenn, wie im Quellcode dargestellt, die drei Eingangssignalbits zu einem `bit_vector` zusammengefasst werden. Dafür wird die im Prozess deklarierte Variable TEMP_IN verwendet. Der Verkettungs-(concatenation-) operator „&“ dient dazu, einzelne Bits bzw. Bitvektoren zu einem Bit-String bzw. `bit_vector` zu verknüpfen. Dabei wird das am weitesten links stehende Signal CIN als höchstwertiges und das am weitesten rechts stehende Signal AI als niederwertigstes Bit in der Variablen TEMP_IN abgelegt. Zu beachten ist außerdem, dass die Zuweisung von Werten bzw. Ausdrücken an Variable mit Hilfe des Variablenzuweisungsoperators „:=“ erfolgt. Die Wahl einer Variablen ist in dieser Anwendung zwingend, da sofort nach der Wertzuweisung in der `case`-Anweisung auf den Variablenwert zugegriffen werden soll. Bei Verwendung eines Signals würde in der `case`-Anweisung fehlerhafterweise immer auf den im letzten Prozessdurchlauf zugewiesenen Signalwert zugegriffen.

Prinzipiell wäre es denkbar, die Verkettung `CIN&B&A` direkt als Kontrollausdruck der `case`-Anweisung zu verwenden, mit dem Ziel die Variable TEMP_IN einzusparen. Dies ist jedoch nicht zulässig, da als Kontrollausdruck sowie als Testausdruck lokal statische Werte verlangt werden, diese also nicht erst während der Ausführung der `case`-Anweisung berechnet werden dürfen [39], [42].

Die beiden Ausgangssignale der Wahrheitstabelle werden in der `case`-Anweisung ebenfalls zunächst als temporäre `bit_vector`-Variable generiert, bevor die eigentlichen Ausgangsbits SI und COUT aus den beiden Komponenten der Variablen TEMP_OUT gebildet werden.

Beim Vergleich der acht Einträge umfassenden Wahrheitstabelle mit der VHDL-Beschreibung fällt auf, dass die drei Varianten, in denen jeweils gerade ein Eingangsbit gesetzt ist, zur Vereinfachung der Schreibweise in einem `when others`-Zweig zusammengefasst sind.

Durch Verwendung des Alternativoperators „|“ können in einer `case`-Anweisung sowie in einer selektiven nebenläufigen Signalzuweisung mehrere Testausdrücke zusammengefasst werden. Die nachfolgende Variante der case-Anweisung in Code 3-3 zeigt, wie dadurch Codezeilen eingespart werden können.

```
...
        case TEMP_IN is
                when "000" =>  TEMP_OUT := "00";
                when "011" | "101" | "110" =>  TEMP_OUT := "10";
                when "111" =>  TEMP_OUT := "11";
                when others => TEMP_OUT := "01";
        end case;
        ...
```

In einem zweiten Beispiel sollen einige Varianten der Zuweisung bzw. Abfrage von `bit_vector`-Konstanten in einer Wahrheitstabelle vorgestellt werden. Als Anwendungsbeispiel wird dazu ein Impulsmustergenerator mit vier Ausgangskanälen A gewählt, der acht mögliche Signalmuster erzeugen kann, die durch einen drei Bit breiten, hexadezimal dargestellten Eingangssignalvektor E ausgewählt werden. Diese Muster sind in der Tabelle 3-2 dargestellt. Die Beschreibung der Bit-Strings dieser Tabelle im VHDL-Code kann dadurch vereinfacht werden, dass diese nicht dual sondern oktal bzw. hexadezimal dargestellt werden. Code 3-4 zeigt, dass in diesem Fall dem Bit-String die Basis des Zahlensystems voranzustellen ist.

```
-- Vierkanal Impulsgenerator mit Wahrheitstabelle
-------------------------------------------------
entity IMPULSGEN is
        port(   E: in bit_vector(2 downto 0);
                A: out bit_vector(3 downto 0));
end IMPULSGEN;
-------------------------------------------------
architecture TABELLE of IMPULSGEN is
begin
P1: process(E)
        begin
                case E is
                        when o"0" => A <=x"7";
                        when o"1" => A <=x"A";
                        when o"2" => A <=x"3";
                        when o"3" => A <=x"F";
                        when o"4" => A <=x"6";
                        when o"5" => A <=x"C";
                        when o"6" => A <=x"0";
                        when o"7" => A <=x"E";
                end case;
        end process P1;
end TABELLE;
```

Code 3-4: Wahrheitstabelle mit oktaler und hexadezimaler Darstellung von Bit-Strings

E	A
0x0	0x7
0x1	0xA
0x2	0x3
0x3	0xF
0x4	0x6
0x5	0xC
0x6	0x0
0x7	0xE

Tabelle 3-2: Zu synthetisierende Wahrheitstabelle eines Vierkanal Impulsmustergenerators

Die erlaubten Kennbuchstaben zur Kennzeichnung der Zahlenbasis sind in der Tabelle 3-3 angegeben. Wie für alle VHDL-Syntaxelemente üblich, können diese groß oder klein geschrieben werden. Der Defaultkennbuchstabe b bzw. B kann, wie bereits zuvor verwendet, auch weggelassen werden.

Basis	Kennbuchstabe	Beispiel
Dual	b oder B	`b"1010_1010"`
Oktal	o oder O	`o"1_4_7"`
Hexadezimal	x oder X	`x"AF_fe"`

Tabelle 3-3: Bit-String-Größen

Zur besseren Lesbarkeit innerhalb eines Bit-Strings sind Unterstriche (underscore) erlaubt, sodass dadurch Zifferngruppen gebildet werden können. Bei der Verwendung von oktalen bzw. hexadezimalen Bit-Strings ist zu beachten, dass bei der Deklaration der Bit-Vektoren geeignete Bit-Breiten gewählt werden müssen. So müssen für oktale Strings ganzzahlige Vielfache von drei und für hexadezimale Strings ganzzahlige Vielfache von vier vorgesehen werden.

3.2.2 Die if-Anweisung

Alternativ zur `case`-Anweisung kann auch die `if-then-elsif/else`-Anweisung zur Strukturierung von Prozessen verwendet werden. Diese Anweisung erlaubt, ähnlich wie die bedingte nebenläufige Signalzuweisung, eine Abfrage mehrerer Bedingungen in den `elsif`-Zweigen und erzeugt damit eine Rangfolge der Bedingungen (Prioritätsencoder).

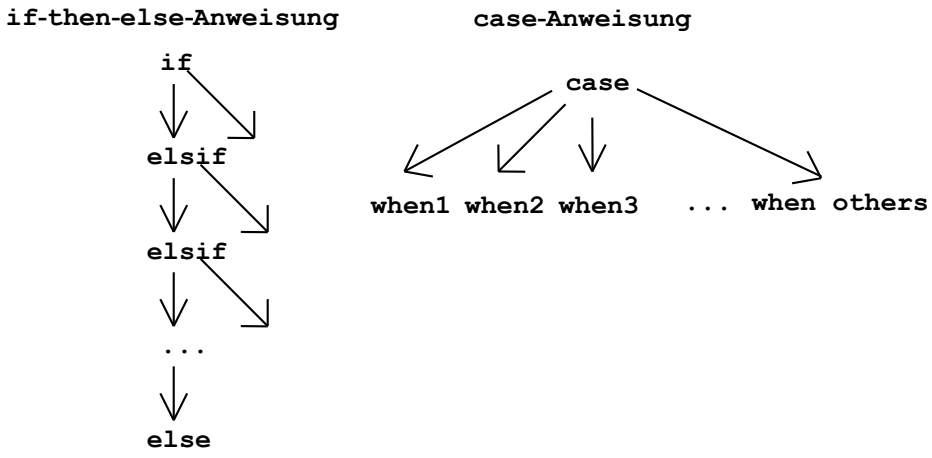


Bild 3-3: Vergleich der Prioritätsstrukturen bei `if-then-elsif/else`- und `case`-Anweisungen

Im Gegensatz zur `case`-Anweisung kann man bei der `if`-Anweisung also nicht davon ausgehen, dass die Signalverzögerungen in der Hardware für alle Verzweigungen gleich lang sind. Vielmehr ist dem Bild 3-3 zu entnehmen, dass die Verzögerungszeit des Ausgangssignals umso kürzer ist, desto weiter oben eine Bedingung wahr wird.

Die Syntax der `if-then-else`-Anweisung ist:

```
if  <Bedingung_1>  then
       {<Sequentielle Anweisungen>;}
[elsif  <Bedingung_2>  then
       {<Sequentielle Anweisungen>;}
elsif  ....
...]
[else
       {<Sequentielle Anweisungen>;}]
end if;
```

Die Anzahl der `elsif`-Zweige ist beliebig. `elsif`-Zweige können ebenso wie der `else`-Zweig auch entfallen. VHDL-Anfänger sollten sich die Schreibweise von „`elsif`“ (ohne zweites e) sowie die Trennung innerhalb von „`end if`“ einprägen. Ein häufiger Anfängerfehler, dessen Suche langwierig sein kann.

Als Abfragebedingung innerhalb der `if-then-else`-Anweisung sind Boole'sche Ausdrücke erlaubt. Dies bedeutet, dass dort entweder Signale vom Typ `boolean` stehen dürfen, oder aber auch logische Verknüpfungen von Vergleichsaussagen. Das nachfolgende Codefragment zeigt eine fehlerhafte sowie eine korrekte Verwendung der Syntax:

```
architecture TEST of TEST_ENT is
signal S1, S2 : bit;
signal B1, B2 : boolean;
begin
PROC: process(S1, S2)
       begin
              if S1 then             -- Fehler da kein Boole'sches Signal
                  ....
              elsif  S1 = '0' then   -- Korrekt da Boole'scher Ausdruck
                  ....
              elsif  B1 then         -- Korrekt da Boole'sches Signal
                  ....
              elsif  B2 and (S1 = '0') then-- Korrekt, Boole'scher Ausdruck
                  ....
              end if;
       end process PROC;
end TEST;
```

Innerhalb jeder Verzweigung können weitere unbedingte Signalzuweisungen und sequentielle Anweisungen, also insbesondere auch weitere `if-then-else`-Anweisungen in beliebiger Verschachtelungstiefe verwendet werden.

Eine zum Code 3-2 äquivalente Funktionalität wird mit der `if`-Anweisung realisiert, die im Code 3-5 dargestellt ist.

```
P1: process(S, E)
       begin
              if S="00" then
                     Y <= E(0);
```

```
            elsif S="01" then
                    Y <= E(1);
            elsif S="10" then
                    Y <= E(2);
            else
                    Y <= E(3);
            end if;
    end process P1;
```

Code 3-5: Realisierung einer 4 zu 1 Multiplexerfunktionalität mit einer `if`-Anweisung

Am Beispiel des Code 3-6 soll erläutert werden, welche Konsequenzen die bereits erläuterte Aktualisierung der Signale am Prozessende haben kann: Insbesondere VHDL-Anfänger versuchen häufig, zuerst einem Signal einen Wert zuzuweisen und während des gleichen Prozessdurchlaufs auf diesen Signalwert zuzugreifen. Dies ist jedoch wegen der geschilderten verzögerten Signalaktualisierung erst bei der nächsten Prozessaktivierung möglich. Angenommen zu Beginn einer Simulation ist das Signal SIG='0', dann wird bei der ersten Prozessausführung zwar die Zuweisung SIG<='1' ausgeführt. Die `if`-Bedingung ist jedoch (noch) nicht wahr. Für manchen unerfahrenen VHDL-Anwender völlig unerwartet verzweigt der Simulator erst bei der zweiten Ausführung durch Wechsel des Signals A in den `then`-Zweig der `if`-Abfrage, da am Ende des ersten Prozessdurchlaufs SIG='1' gesetzt worden war. In der Konsequenz toggelt SIG mit jedem Prozessaufruf zwischen '0' und '1'. In der Simulation fallen diese verzögerten Aktualisierungen als Speichereffekte auf, die von Schaltnetzen nicht zu erwarten sind.

```
architecture ARCH1 of TEST is
signal SIG : bit;
       process ( A )
       begin
       SIG <= '1';
       ...;
              if SIG = '1' then -- Wird erst im 2. Durchlauf ausgefuehrt
              SIG <= '0';
              ..;
       else
              ..;
       end if;
       end process;
end ARCH1;
```

Code 3-6: Signalwertzuweisung und -abfrage in einem Prozess

Dieses Beispiel macht deutlich, dass die unbedingte Wertzuweisung an ein Signal – wenn sie nur einmal erfolgt und sie nicht selbst Teil einer sequentiellen Anweisung ist – letztlich an beliebiger Stelle des Prozesses stehen kann, ohne dass sich die Funktionalität des Prozesses ändert.

Wenn innerhalb eines Prozessdurchlaufs Größen verändert werden sollen und der Prozess sofort auf diese Veränderung reagieren soll, so zeigt Code 3-7, dass dafür eine Variable (SIG_VAR) verwendet werden muss. Variablen dienen somit als Rechenzwischengrößen innerhalb von Prozessen. Wenn der Wert der Variablen auch außerhalb des Prozesses benötigt wird, so ist die Variable am Prozessende an das Signal (SIG) zu übergeben. Im Unter-