Einführung in VHDL

Begleitmaterial zur Vorlesung
Rechnerarchitektur im Wintersemester 2012/13

Ursprüngliches Manuskript:
© 1994 Georg Baumann und Peter Marwedel
(FernUniversität Hagen)

Überarbeitet:
2012 Nicolas Roeser
mit Inhalten von Benjamin Menhorn,
Kilian Kempf und Tobias Bund
(Universität Ulm)

27. November 2012

Inhaltsverzeichnis

Abbildungsverzeichnis v

Tabellenverzeichnis vii

1 Einführung in VHDL 1
1.1 Motivation und Entstehungsgeschichte . . . 1
1.1.1 Welche Vorteile bieten HDLs? . . . 2
1.1.2 Die Entwicklung von VHDL . . . 2
1.2 Ein einführendes Beispiel . . . 2
1.2.1 Aufbau einer VHDL-Beschreibung . . . 3
1.2.2 Hierarchische Beschreibung in VHDL . . . 3
1.2.3 Erstes Beispiel . . . 4
1.3 Simulation von VHDL-Modellen . . . 6
1.3.1 Beispielhafter Ablauf einer Simulation . . . 6
1.3.2 Das Simulationskonzept von VHDL . . . 8
1.3.3 Das Delta-Delay . . . 9
1.3.3.1 Beispiel 1: „Normale Anwendung“ des Delta-Delay . . . 9
1.3.3.2 Beispiel 2: Unendlich viele Delta-Schritte . . . 10
1.4 Grundlegende Sprachelemente . . . 11
1.4.1 Lexikalische Elemente . . . 11
1.4.1.1 Bezeichner . . . 11
1.4.1.2 Literale . . . 12
1.4.1.3 Kommentare . . . 13
1.4.2 Daten und ihre Container . . . 13
1.4.2.1 Skalare Datentypen . . . 14
1.4.2.1.1 Aufzählungstypen . . . 14
1.4.2.1.2 Integer-Typen . . . 15
1.4.2.1.3 Physikalische Typen . . . 15
1.4.2.1.4 Fließkommatypen . . . 15
1.4.2.2 Zusammengesetzte Datentypen . . . 15
1.4.2.2.1 Feldtypen . . . 15
1.4.2.2.2 Strukturtypen . . . 16
1.4.3 Vordefinierte Operatoren . . . 16
1.4.4 Die Struktur von VHDL-Programmen . . . 16
1.4.4.1 Bausteindeklarationen . . . 19
1.4.4.2 Schnittstellenlisten . . . 19
1.4.4.3 Architekturrümpfe . . . 21
1.4.5 Beschreibung von Verhalten . . . 21
1.4.5.1 Die `process`-Anweisung . . . 21
1.4.5.2 Die `wait`-Anweisung . . . 22
1.4.5.3 Zuweisungen . . . 23
1.4.5.3.1 Variablenzuweisungen . . . 23

1.4.5.3.2 Zuweisungen an Signale 23
1.4.5.4 Kontrollstrukturen 24
1.4.5.4.1 Bedingte Ausführung 24
1.4.5.4.2 Iterative Ausführung 25
1.4.6 Beschreibung von Struktur 26
1.4.6.1 Assoziationslisten 28

2 Vertiefungen in VHDL 29
2.1 Modellierung einer Ampelschaltung 29
2.2 Paketdeklaration und Paketrumpf 31
2.3 VHDL-Bibliotheken 33
2.3.1 Entwurfseinheiten 33
2.3.2 Sichtbarkeit von Entwurfseinheiten 34
2.3.3 Sichtbarkeit von Bibliotheken 35
2.4 Implizite Prozesse 36
2.4.1 Nebenläufige Signalzuweisungen 37
2.4.2 Bedingte Signalzuweisungen 37
2.4.3 Selektive Signalzuweisungen 38
2.5 Inertial Delay und Transport Delay 38
2.5.1 Transport Delay 39
2.5.2 Inertial Delay 39
2.6 Funktionen und Prozeduren 40
2.7 Aufbau einer Testumgebung 42
2.8 Weitere VHDL-Konstrukte 45
2.8.1 Der Alias-Mechanismus 46
2.8.2 Attribute 46
2.8.3 Subtypen 48
2.8.4 Signale mit mehreren Treibern 48
2.8.5 Reguläre Strukturen 49
2.8.6 Konfigurationen 50

3 Übungen im Labor 53
3.1 Kurze VHDL-Zusammenfassung für den Laborbetrieb 53
3.2 Das erste Projekt in Quartus II 61

4 Literaturverzeichnis 63

Abbildungsverzeichnis

1.1 Aufbau eines Bausteinentwurfs 3
1.2 Hierarchische Beschreibung eines Bausteins 4
1.3 Schnittstelle eines Bausteins 5
1.4 Strukturdiagramm für den Baustein „`full_adder`“ 6
1.5 Simulationsergebnisse als Ablaufdiagramm 7
1.6 Die Simulationsphasen von VHDL 9
1.7 Anordnung von Delta-Einheiten auf der Zeitachse 9

2.1 Endlicher Automat zur Kontrolle der Ampelanlage 30
2.2 Aufbau eines VHDL-Pakets 32
2.3 Inhalt einer VHDL-Bibliothek 35
2.4 Verwendung von Elementen anderer Bibliotheken 36
2.5 Aufbau einer Testumgebung 43
2.6 Simulationsergebnisse 45
2.7 Befehlsformat 46

Tabellenverzeichnis

1.1 Systemzustände nach Delta-Schritten 10
1.2 Vordefinierte Datentypen von VHDL 14
1.3 In VHDL vordefinierte Operatoren 17
1.4 Erlaubte Datentypen zu den vordefinierten Operatoren 18
1.5 Modi zur Beschreibung der Datenflussrichtung 20

2.1 VHDL-Entwurfseinheiten 33

3.1 Signale der 9-wertigen Logik nach IEEE-Standard 1164-1993 55

1 Einführung in VHDL

Der Sinn dieses Textes ist es, eine kurze Einführung in die Sprache VHDL zu geben, damit der Leser sich einen Überblick über diese Sprache verschaffen kann. Dies wird anhand von Beispielen getan und die Eigenschaften der Sprache werden informell beschrieben. Es wird nicht jeder Aspekt von VHDL berücksichtigt, da dies den gegebenen Rahmen sprengen würde. Für Details sei auf das „Language Reference Manual“ (LRM) [IEE88] verwiesen. Gleichzeitig muss darauf hingewiesen werden, dass sich dieses Manual nicht als Lehrbuch eignet. Das LRM liest sich wie ein Gesetzestext und sollte nur als Referenz verwendet werden. Besser lesbare Lehrbücher, die genauer auf die Sprachelemente eingehen als dies hier möglich ist, sind der folgenden Literaturliste zu entnehmen:

Roger Lipset, Carl Schaefer, Cary Ussery: VHDL: Hardware Description and Design, Kluwer Academic Publishers, Boston, 1989.
Dieses Lehrbuch gibt eine recht umfassende und gute Einführung in VHDL. Dem Buch sind auch ein Teil der für diesen Text verwendeten Beispiele entnommen.

D. R. Coelho: The VHDL Handbook, Kluwer Academic Publishers, Boston, 1989.
In diesem Buch sind viele Beispiele für VHDL-Beschreibungen zusammengetragen worden, die man zu einem guten Teil für eigene Projekte übernehmen kann.

J. R. Armstrong: Chip-Level Modeling With VHDL, Prentice Hall, Englewood Cliffs, NJ, 1988.
Der Schwerpunkt dieses Werkes liegt in der Modellierung von Hardware auf höheren Abstraktionsebenen.

IEEE: Standard VHDL Language Reference Manual – Std 1076-1987, IEEE, New York, 1988.
Im Fall von detaillierten Fragen zur Syntax von VHDL ist dieses Manual die entscheidende Referenz. Als Lehrbuch nicht zu empfehlen.

1.1 Motivation und Entstehungsgeschichte

An dieser Stelle wird zunächst eine Definition des Begriffs Hardware-Beschreibungssprache (Hardware Description Language = HDL) gegeben, um solche Sprachen in der Menge aller maschinenlesbaren Sprachen einordnen zu können.

Definition 1.1.1 *HDLs sind zur Beschreibung (Modellierung) von Hardware-Systemen (Schaltungen) bestimmt. Die in einer HDL erstellten Beschreibungen können i. Allg. zur Eingabe an einen Simulator verwendet werden, mit dessen Hilfe man die Funktionalität des beschriebenen Systems validieren kann. HDLs unterstützen i. Allg. mehrere Abstraktionsebenen und verschiedene Sichten eines Hardware-Systems.*

1.1.1 Welche Vorteile bieten HDLs?

Die Verwendung von HDLs bringt für den Hardware-Entwickler eine ganze Reihe von Vorteilen mit sich. Im Wesentlichen sollen HDLs eine Steigerung der Arbeitseffizienz und eine Verbesserung der Wiederverwendbarkeit der Arbeitsergebnisse unterstützen. Im Einzelnen lassen sich die folgenden hauptsächlichen Argumente ausmachen:

- HDLs dienen als Kommunikationsmedium zwischen Hardwareentwicklern,
- HDLs erlauben eine formale Spezifikation,
- HDLs dienen der Dokumentation,
- HDL-Modelle sind simulierbar,
- HDLs stellen ein großes Hilfsmittel in einem integrierten und automatisierten Entwurfsprozess dar (auch als Ablage für Zwischenergebnisse zu benutzen).

1.1.2 Die Entwicklung von VHDL

Der Ursprung der Hardware-Beschreibungssprachen liegt in den 60-er Jahren und früher. HDLs wurden in Anlehnung an die Konzepte von Programmiersprachen entworfen. Vor VHDL war es üblich, dass jedem Software-System zur Unterstützung des Entwurfsprozesses eine eigene HDL zugrunde lag. So entwickelte sich eine Vielzahl von HDLs, was die Übertragung der Daten von einem Entwurfssystem zu einem anderen behinderte und die Kommunikation zwischen den Entwicklern erschwerte.

1980 wurde vom US-Verteidigungsministerium ein Programm gestartet, das die Entwicklung von VLSI (Very Large Scale Integration)-Produkten fördern sollte. Dies war das VHSIC (=Very High Speed Integrated Circuits)-Projekt. Im Rahmen dieses Projekts wurde sehr schnell deutlich, dass man ein einheitliches Kommunikationsmedium benötigte, um die Design-Daten auszutauschen. Damit war die Idee einer Standard-HDL geboren. 1981 wurden dann die Anforderungen an eine VHSIC-HDL (=VHDL) festgelegt und eine Aufforderung an die einschlägigen Firmen herausgegeben, Vorschläge für eine solche Sprache zu machen. Den Auftrag zur Implementierung von VHDL erhielt 1983 ein Konsortium aus den Firmen Intermetrics, IBM und Texas Instruments. 1984 wurde die Sprachspezifikation abgeschlossen (mit der Version 7.2) und 1987 der erste entsprechende Simulator freigegeben.

In diesem Jahr hat das IEEE (The Institute of Electrical and Electronics Engineers) ebenfalls den Auftrag zur Implementierung einer eigenen VHDL-Version vergeben. Diese Version wurde 1988 als Industriestandard 1076 bestätigt und anschließend auch vom US-Verteidigungsministerium übernommen. Dies ist auch die Version, mit der sich dieser Text beschäftigt.

1.2 Ein einführendes Beispiel

Um einen ersten Kontakt zu VHDL herzustellen, und um in den folgenden Abschnitten Beispiele zur Verfügung zu haben, wird zunächst ein einfaches Beispiel einer VHDL-Beschreibung gezeigt. Es ist nicht Sinn dieses Abschnitts, jede Einzelheit des VHDL-Programms zu verstehen. Vielmehr kommt es darauf an, ein Gefühl für den Aufbau solcher Programme zu bekommen.

1.2.1 Aufbau einer VHDL-Beschreibung

Jede zu modellierende Einheit (Hardware-Baustein) wird als Bausteinentwurf oder VHDL-Baustein („Design Entity“ oder „VHDL Entity“) bezeichnet. Ein Bausteinentwurf setzt sich aus zwei Bestandteilen zusammen. Dies sind die Bausteindeklaration („Entity Declaration“) und der Architekturrumpf („Architecture Body“). Erstere dient der Beschreibung der Schnittstelle. Hier wird also z. B. festgelegt, wie viele Ein- und Ausgänge der Baustein besitzt. Zu jeder Bausteindeklaration gibt es außerdem einen oder mehrere Architekturrümpfe, in denen die Realisierung des Bausteins beschrieben wird.

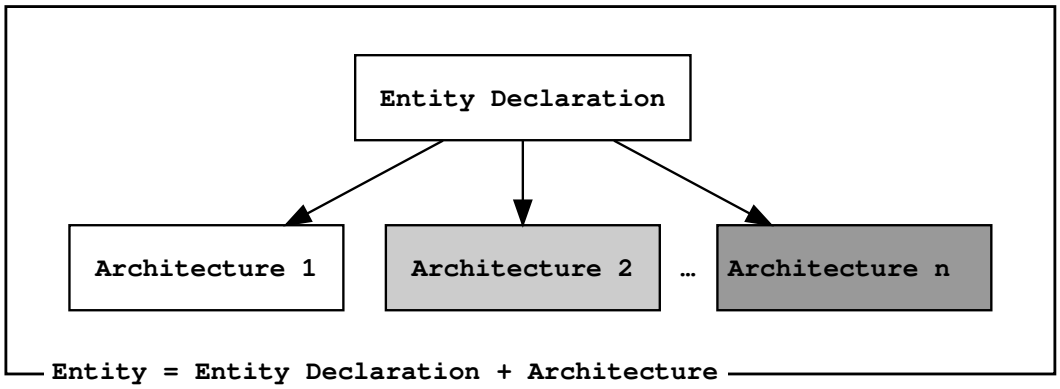

Abbildung 1.1: Aufbau eines Bausteinentwurfs

Mehrere Versionen einer Architektur werden unterstützt, um z. B. alternative Realisierungen eines Bausteins testen zu können. Innerhalb einer Architektur ist es nun möglich, entweder das Verhalten des Bausteins direkt zu beschreiben, oder eine Realisierung des Bausteins durch ein Zusammenschalten einfacherer Bausteine zu modellieren. Es kann also entweder eine Verhaltens- oder eine Strukturbeschreibung gegeben werden.

Welche der verschiedenen Versionen des Architekturrumpfs für die Simulation benutzt wird, ist vom aktuellen Zustand des Simulationssystems abhängig. Im Standardfall wird die Architektur verwendet, die als letzte vom System analysiert wurde. Es ist aber auch möglich, die zu verwendende Architektur explizit anzugeben.

1.2.2 Hierarchische Beschreibung in VHDL

Ein vollständiges Entwurfsmodell besteht (üblicherweise) aus einer Hierarchie von miteinander verbundenen Bausteinentwürfen. Wenn man diese Hierarchie als Baum darstellt, dann entsprechen die Blätter des Baumes VHDL-Bausteinen, die als Verhaltensbeschreibung gegeben sind, während alle inneren Knoten Strukturbeschreibungen darstellen. In VHDL ist es allerdings auch möglich, eine gemischte Beschreibung zu erstellen.

Die in Abbildung 1.2 vorgestellte Hierarchie zeigt einen Volladdierer, der aus zwei Halbaddierern und einem OR-Gatter aufgebaut ist. Beide Halbaddierer sind wiederum aus jeweils einem XOR-Gatter und einem AND-Gatter aufgebaut. Alle Bausteine mit Namen „…`gate`“ sind also durch Verhaltensbeschreibungen realisiert, während für die anderen Bausteine Strukturbeschreibungen gegeben werden.

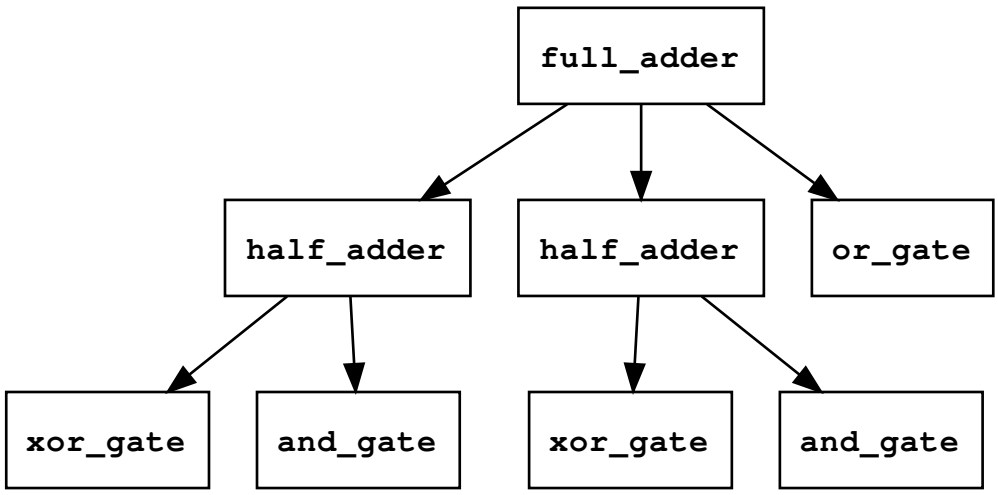

Abbildung 1.2: Hierarchische Beschreibung eines Bausteins

Es ist in VHDL natürlich nicht nötig, einen Baustein zweimal zu beschreiben. Stattdessen wird z. B. der Bausteinentwurf „`half_adder`“ einmal definiert und innerhalb des Bausteins „`full_adder`“ zweimal aufgerufen.

Das folgende Beispiel wird die gezeigte Hierarchie realisieren. Um das Beispiel nicht zu lang werden zu lassen, werden wir aber damit anfangen, dass wir direkt für den Volladdierer eine Verhaltensbeschreibung angeben.

1.2.3 Erstes Beispiel

Das erste hier vorgeführte Beispiel ist ein Standardbeispiel und wird z. B. auch in [LSU89] verwendet. Es soll ein Volladdierer in VHDL entworfen werden. Zunächst benötigen wir eine Bausteindeklaration (die Schlüsselworte von VHDL erscheinen **`fett`**):

Beispiel 1.2.1

```
entity full_adder is
  port(a, b, carry_in:  in Bit;   -- input ports
       sum, carry_out: out Bit);  -- output ports
end full_adder;
```

Mit dieser Bausteindeklaration ist bisher nur das Aussehen des Volladdierers nach außen beschrieben. (Die Zeichenfolge „`--`“ eröffnet in VHDL einen Kommentar, der bis zum Zeilenende reicht.) Wir wissen jetzt, dass der Baustein drei Eingänge mit den Namen „`a`“, „`b`“ und „`carry_in`“ vom Typ `Bit` hat und außerdem die beiden Ausgänge „`sum`“ und „`carry_out`“ ebenfalls Typ `Bit`. (Der Typ „`Bit`“ wird in Abschnitt 1.4.2 genauer erläutert.) Der beschriebene Baustein könnte also durch das Schaltungsbild in Abbildung 1.3 dargestellt werden.

Abbildung 1.3: Schnittstelle eines Bausteins

Die Realisierung des Bausteins wird getrennt in einem Architekturrumpf beschrieben:

Beispiel 1.2.2

```
architecture behavior of full_adder is
  signal s: Bit;
begin
  s <= a xor b after 5 Ns;
  sum <= s xor carry_in after 5 Ns;
  carry_out <= (a and b) or (s and carry_in) after 10 Ns;
end behavior;
```

In diesem Beispiel wird das Verhalten des Bausteins mit Hilfe von Booleschen Gleichungen beschrieben. Jede Anweisung innerhalb des `begin-end`-Blocks stellt einen eigenständigen Prozess dar. Alle Prozesse werden nebenläufig und unabhängig voneinander bearbeitet. (Eine genaue Definition des Begriffs „Prozess“ im Sinne von VHDL wird in Abschnitt 1.3.2 gegeben.)

Die einzige hier verwendete Anweisung ist die Zuweisung von Werten an Signale – eine der wichtigsten Anweisungen in VHDL. Der Ausdruck rechts von der Zeichenfolge „`<=`“ wird ausgewertet und anschließend wird eine entsprechende Zuweisung an das Signal links der Zeichenfolge „`<=`“ erzeugt. Der Zeitpunkt dieser Zuweisung ist abhängig von der `after`-Klausel hinter dem Ausdruck. Im gezeigten Beispiel führen die `after`-Klauseln dazu, dass die Zuweisungen mit einer Verzögerung von 5 bzw. 10 Nanosekunden ausgeführt werden. Mit Hilfe dieser Notation kann man die von den Gattern verursachten Verzögerungen von der Simulation berücksichtigen lassen.

„`xor`“, „`and`“ und „`or`“ sind für den Typ „`Bit`“ vordefinierte Operationen. Ausdrücke können – wie in Programmiersprachen üblich – geschachtelt aufgebaut werden.

Alternativ kann man einen zweiten Architekturrumpf erstellen, in dem der Baustein „`full_adder`“ unter Verwendung zweier Halbaddierer realisiert wird:

Beispiel 1.2.3

```
architecture structure of full_adder is

  component half_adder
    port (i1, i2: in Bit;
          carry:  out Bit;
          sum:    out Bit);
  end component;
```

```
  component or_gate
    port (i1, i2: in Bit;
          o:      out Bit);
  end component;

  signal x, y, z: Bit;

begin
  s1: half_adder port map (a, b, x, y);
  s2: half_adder port map (y, carry_in, z, sum);
  s3: or_gate port map (x, z, carry_out);
end structure;
```

In einer Strukturbeschreibung müssen die verwendeten Bausteintypen zunächst mit einer `component`-Anweisung definiert werden. Dieses Definitionen haben eine Struktur, die einer Bausteindeklaration sehr ähnlich ist.

Die drei Anweisungen innerhalb der `begin-end`-Klammer sind sogenannte Komponenteninstanziierungen. In der ersten Anweisung wird z. B. ein Baustein mit dem Namen „`s1`“ vom Typ „`half_adder`“ erzeugt und auf geeignete Art und Weise mit den anderen Bausteinen und mit den Ein- und Ausgängen des zu definierenden Bausteins verbunden. Um die internen Verbindungen herstellen zu können, mussten mit der `signal`-Anweisung interne Leitungen definiert werden, wie sie auch in einem Strukturdiagramm auftauchen.

Die im Architekturrumpf „`structure`“ angegebene Struktur entspricht also dem Diagramm in Abbildung 1.4.

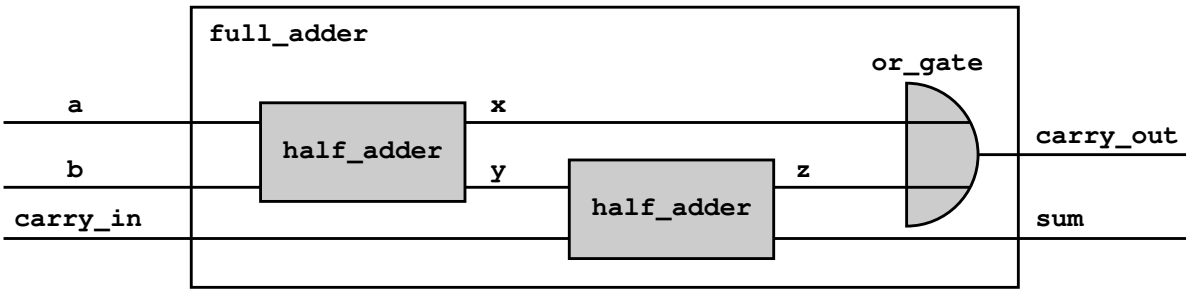

Abbildung 1.4: Strukturdiagramm für den Baustein „`full_adder`“

1.3 Simulation von VHDL-Modellen

1.3.1 Beispielhafter Ablauf einer Simulation

In diesem Abschnitt soll ein Eindruck von der Arbeit mit VHDL-Simulatoren vermittelt werden. Da die Software-Pakete von verschiedenen Herstellern sich aber in der Bedienung stark unterscheiden, kann hier nur versucht werden, einen ganz allgemeinen Überblick über die verschiedenen Arbeitsschritte zu geben.

Wir nehmen also an, dass die oben gegebenen Beispiele mit Hilfe eines Texteditors am Rechner eingegeben worden sind (alle drei Beispiele sollen in einer gemeinsamen

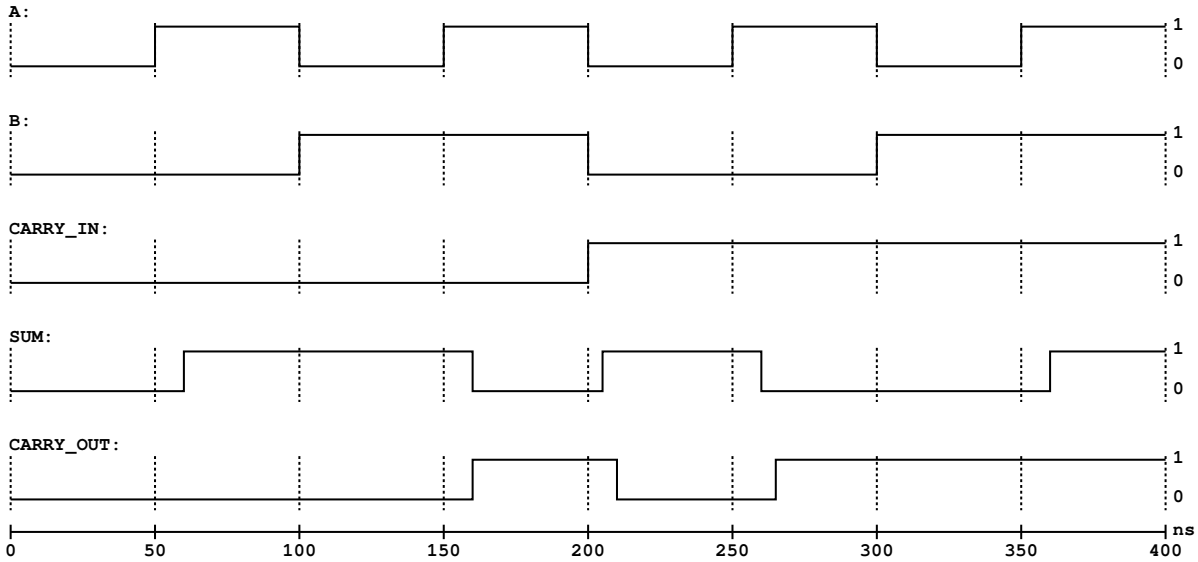

Abbildung 1.5: Simulationsergebnisse als Ablaufdiagramm

Datei abgelegt worden sein). Um nun eine Simulation der entsprechenden VHDL-Modelle starten zu können, muss die entsprechende Datei zunächst analysiert werden. Es wird also auf geeignete Art und Weise das Analyseprogramm gestartet. Dieses meldet – wenn keine Fehler aufgetreten sind –, dass die folgenden drei Entwurfseinheiten erfolgreich analysiert wurden:

- `full_adder`,
- `full_adder(behavior)`,
- `full_adder(structure)`.

Eine Entwurfseinheit („Design Unit“) ist ein zusammenhängender Block des VHDL-Programms. In unserem Fall haben wir zunächst die Entwurfseinheit „`full_adder`“, die unserer Bausteindeklaration entspricht. Weiterhin existieren zwei verschiedene Architekturrümpfe zu dieser Bausteindeklaration. Die hier gezeigte Namensgebung für die verschiedenen Entwurfseinheiten ist eine Möglichkeit unter vielen. Es wird zunächst der Name des Bausteins angegeben und anschließend im Fall eines Architekturrumpfs noch dessen Name.

Je nach verwendetem Simulationssystem müssen nun noch verschiedene Schritte, wie z. B. Erzeugung der benötigten Simulatoren, durchgeführt werden. Für das am Lehrstuhl Informatik vorhandene System der Fa. Vantage (die in die Fa. Viewlogic übernommen wurde) befinden sich entsprechende Informationen im zweiten Teil dieses Textes.

Je nach Simulationssystem kann man sich bereits während oder aber spätestens nach dieser Simulation die Simulationsergebnisse – also z. B. die Werte an den Ausgangsleitungen – anzeigen lassen. Für den gesamten oben angegebenen Ablauf ergibt sich ein Ablaufdiagramm ähnlich der Abbildung 1.5.

Man erkennt an diesem Diagramm sehr gut, wie sich die im Architekturrumpf „`behavior`“ spezifizierten Verzögerungen auswirken. Bei der Ausgabe von Diagrammen gibt es verschiedene Optionen. So könnte man sich z. B. auch das interne Signal „`s`“ anzeigen lassen oder die Ausgabe auf ein bestimmtes Zeitintervall beschränken.

1.3.2 Das Simulationskonzept von VHDL

Das Konzept, nach dem ein VHDL-Simulator zu implementieren ist, ist bereits im Language Reference Manual [IEE88] festgelegt. Dadurch soll sichergestellt werden, dass eine Simulation – auf verschiedenen Simulatoren ausgeführt – immer dieselben Ergebnisse liefert. Da es sich hier um ereignisorientierte Simulationen handelt, stützt sich das Konzept für die Simulation auf den Begriffen „Ereignis“ und „Prozess“ ab. Deshalb werden diese Begriffe zunächst definiert.

Definition 1.3.1 *Ein* **Ereignis** *beschreibt eine Zuweisung eines Wertes an ein Signal zu einem festgelegten Zeitpunkt (in der Zukunft). Ereignisse werden bei der Ausführung von Zuweisungsanweisungen generiert und dann zeitlich geordnet in eine Ereignistabelle eingetragen. Wenn im Laufe der Simulation die Zeit weitergeschaltet wird, dann wird zunächst nachgesehen, welche Ereignisse für den erreichten Zeitpunkt in der Ereignistabelle enthalten sind. Anschließend werden die entsprechenden Signale aufgrund der gefundenen Ereignisse aktualisiert. (Ein Ereignis kann auch der Ablauf der Wartezeit für einen Prozess sein, der nach Ablauf dieser Wartezeit wieder aktiviert werden soll.)*

Definition 1.3.2 *Ein* **Prozess** *ist eine Sammlung von Anweisungen, die in sich sequentiell, aber unabhängig von anderen Prozessen ausgeführt wird. (Die Ausführung einer Zuweisungsanweisung an ein Signal beinhaltet hierbei die Berechnung des zuzuweisenden Wertes, die Erzeugung eines entsprechenden Ereignisses und das Einordnen des Ereignisses in die Ereignistabelle, aber nicht die Aktualisierung des Signals.) Ein Prozess kennt zwei Zustände: Er ist entweder aktiv oder suspendiert. Zu Anfang der Simulation werden alle Prozesse einmal aktiviert. Jeder Prozess läuft dann so lange, bis er zur Ausführung einer* `wait`*-Anweisung gelangt (*`wait`*-Anweisungen können explizit notiert werden oder implizit gegeben sein; siehe Abschnitte 1.4.5.2 und 2.4). Dies führt zu einer Suspendierung des Prozesses. Durch die jeweilige* `wait`*-Anweisung wird festgelegt, durch welche Ereignisse ein Prozess wieder aktiviert wird. Meist sind dies Signalwechsel auf Eingangsleitungen, die dazu führen, dass ein neuer Wert für ein Ausgangssignal berechnet werden muss.*

Verschiedene Prozesse können nur über Signale miteinander kommunizieren. Dies führt dazu, dass eine Nachricht frühestens einen Simulationszyklus nach der Erzeugung dieser Nachricht empfangen wird.

Die Simulation läuft nun nach einem Zwei-Phasen-Modell ab. Die erste Phase ist die Aktualisierungsphase, in der die Werte der Signale aktualisiert werden. Diese Phase wird z. B. am Anfang der Simulation einmal ausgeführt, wenn allen Signalen der Initialwert zugewiesen wird. Die zweite Phase ist die Abarbeitungsphase, in der die jeweils aktiven Prozesse abgearbeitet werden.

Nach der Abarbeitungsphase – sobald alle Prozesse suspendiert sind – wird die Simulationszeit auf den Zeitpunkt gesetzt, für den das nächste Ereignis vorliegt. Dann startet die nächste Aktualisierungsphase. Wenn alle Aktualisierungen für den neuen Zeitpunkt ausgeführt wurden, wird wiederum eine Abarbeitungsphase gestartet. Zu Beginn einer Abarbeitungsphase muss bestimmt werden, welche Prozesse aufgrund der Aktualisierungen wieder aktiv werden. Diese Prozesse werden dann wieder ausgeführt usw.

Die Simulation wird beendet, wenn entweder ein vorher bestimmter Zeitpunkt erreicht ist, oder wenn die Ereignistabelle am Ende einer Abarbeitungsphase leer ist.

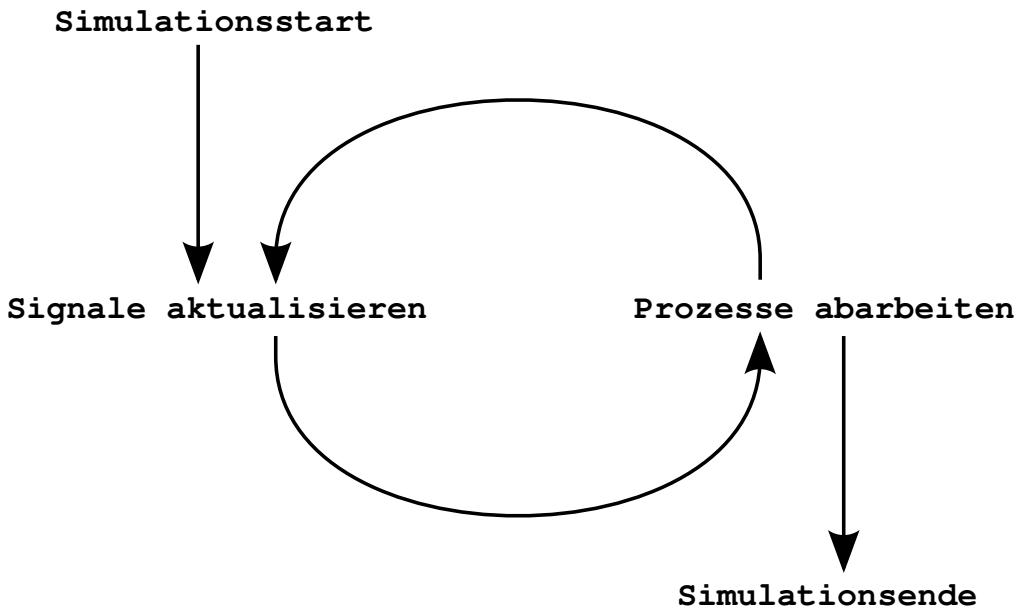

Abbildung 1.6: Die Simulationsphasen von VHDL

1.3.3 Das Delta-Delay

Die VHDL-Syntax schreibt nicht vor, dass bei jeder Zuweisung an ein Signal ein Verzögerungswert explizit angegeben werden muss. Um nun auch solche Zuweisungen behandeln zu können, denen kein Delay zugeordnet ist, wird in diesem Fall automatisch ein Delta-Delay zugeordnet. In diesem Fall ändert sich der Wert des Signals also ein Delta-Delay nach der Ausführung der Zuweisung, d. h. im nächsten Simulationszyklus. Die Einordnung eines Delta-Delay in den übrigen Zeitablauf hat man sich so vorzustellen, dass sich beliebig viele dieser Einheiten zwischen zwei aufeinanderfolgende Zeitpunkte der kleinsten verwendeten Zeiteinheit schieben können. Die Abbildung 1.7 versucht diesen Sachverhalt zu veranschaulichen.

Abbildung 1.7: Anordnung von Delta-Einheiten auf der Zeitachse

Unabhängig von der ansonsten verwendeten kleinsten Zeiteinheit werden also zunächst so viele Delta-Zeitschritte ausgeführt, bis kein Ereignis mit Delta-Verzögerung mehr erzeugt wird. Erst dann wird der nächste „echte“ Zeitschritt veranlasst. Dieses Konzept hat gewisse Auswirkungen, die der Hardware-Entwickler im Hinterkopf haben sollte. Die Auswirkungen werden in den folgenden Beispielen demonstriert.

1.3.3.1 Beispiel 1: „Normale Anwendung“ des Delta-Delay

Das Vorgehen von VHDL-Simulatoren wird mit Hilfe des Programmfragments in Beispiel 1.3.1 deutlich.

Beispiel 1.3.1

```
signal a: Bit := '0';
signal b: Bit := '0';
signal c: Bit := '0';

process begin
  b <= '1';
  c <= '1' after 1 Ns;
  wait for 1 Ns;

  a <= b;
  b <= a;
  wait for 1 Ns;

end process;
```

Der hier modellierte Prozess zeigt das folgende Verhalten: Alle drei Signale werden vor der Simulation mit dem Wert `'0'` initialisiert. Dies kann wie gezeigt direkt bei der Deklaration der Signale notiert werden. Im ersten Simulationszyklus werden dann die Anweisungen bis zur ersten `wait`-Anweisung ausgeführt. Dies führt zur Eintragung von zwei Zuweisungen in die Ereignistabelle. Die Zuweisung an das Signal „`b`“ wird zum Zeitpunkt 0 plus 1 Delta stattfinden. Zu diesem Zeitpunkt findet nur die Zuweisung an das Signal „`b`“ statt, da der Prozess erst wieder zum Zeitpunkt 1 Nanosekunde aktiviert wird. Dies ist auch schon der nächste zu bearbeitende Zeitpunkt. Zu diesem Zeitpunkt wird zunächst dem Signal „`c`“ der Wert `'1'` zugewiesen. Dann werden die Anweisungen bis zur zweiten `wait`-Anweisung ausgeführt. Diese führen wiederum zur Eintragung von zwei Zuweisungen in die Ereignistabelle. Da die Ausdrücke auf der rechten Seite der Zuweisung bereits jetzt ausgewertet werden, haben die gezeigten Anweisungen einen Austausch der Werte von „`a`“ und „`b`“ zum Zeitpunkt 1 Nanosekunde plus 1 Delta zur Folge. Der Ablauf der Simulation ist in Tabelle 1.1 zusammengefasst worden.

	0 ns	0 ns + 1 Δ	1 ns	1 ns + 1 Δ
a	0	0	0	1
b	0	1	1	0
c	0	0	1	1

Tabelle 1.1: Systemzustände nach Delta-Schritten

1.3.3.2 Beispiel 2: Unendlich viele Delta-Schritte

Dieses Beispiel zeigt eine Konstruktion, die zur Erzeugung unendlich vieler Ereignisse führt, die jeweils einen Delta-Schritt auseinander liegen.

Beispiel 1.3.2

```
  loop
    a <= not a;
    wait on a;
  end loop;
```

Sobald ein Prozess in die hier gezeigte Schleife läuft, wird der Simulator nicht mehr dazu kommen, einen weiteren „echten“ Zeitsprung vorzunehmen. Innerhalb dieser Programmschleife wird bei jedem Durchlauf ein Ereignis mit Delta-Verzögerung generiert. Dieses Ereignis negiert den Wert des Signals „`a`“. Anschließend wird auf das gerade generierte Ereignis gewartet.

Konstruktionen der gezeigten Art sollten also vermieden werden, da der Simulator an dieser Stelle in der Schleife hängen bleibt.

1.4 Grundlegende Sprachelemente

Nachdem in den vorangegangenen Abschnitten VHDL anhand von Beispielen eingeführt wurde und grundlegende Konzepte erläutert wurden, werden jetzt die wesentlichen Sprachelemente strukturiert und eingehend behandelt. Auch dieser Abschnitt beschränkt sich aber auf die Elemente, die für einfache Modellierungsaufgaben wichtig sind und lässt fortgeschrittenere Konstrukte außer Acht, da dies den gegebenen Rahmen sprengen würde.

1.4.1 Lexikalische Elemente

Die der Sprache VHDL zugrunde liegenden lexikalischen Regeln – also z. B. die Beschreibung der zulässigen Variablennamen – sollen in diesem Tutorial nur sehr kurz abgehandelt werden, da erstens nicht genug Zeit vorhanden ist, um auf jede Einzelheit einzugehen, und man diese Einzelheiten zweitens sowieso nicht im Kopf behält. Bei Unklarheiten über die Syntax von VHDL sei an das Language Reference Manual [IEE88] verwiesen.

1.4.1.1 Bezeichner

Bezeichner – das sind z. B. Namen von Variablen oder Konstanten – müssen in VHDL den folgenden Regeln genügen:

- Das erste Zeichen eines Bezeichners muss ein Buchstabe sein.
- Alle anderen Zeichen sind Buchstaben, Ziffern oder der Unterstrich (`_`).
- Zwei Unterstriche dürfen nicht aufeinander folgen.
- Das letzte Zeichen darf kein Unterstrich sein.

Dabei wird nicht zwischen Groß- und Kleinbuchstaben unterschieden. D. h., dass z. B. die folgenden drei Bezeichner für ein und dasselbe Objekt stehen:

```
ABC
abc
aBc
```

Weiterhin muss beachtet werden, dass einige Bezeichner (wie z. B. „`begin`“, „`end`“ und „`after`“) bereits als Schlüsselwort der Sprache VHDL reserviert sind und somit nicht als Variablen-, Signalnamen o. ä. benutzt werden dürfen. (Eine Liste aller Schlüsselworte von VHDL befindet sich im Anhang.)

1.4.1.2 Literale

Ein Literal ist ein Symbol (nicht zu verwechseln mit einem Schlüsselwort), dessen Wert mit seiner Darstellung fest verknüpft ist. Z. B. ist „42“ ein Literal, welches die Integer-Zahl 42 darstellt. VHDL kennt 6 verschiedene Arten von Literalen:

Integer-Literale stellen ganze Zahlen dar. Sie bestehen aus einer Folge von Ziffern und können einen Exponenten zur Basis 10 beinhalten. Außerdem dürfen Unterstriche zur Verbesserung der Lesbarkeit eingefügt werden:

```
          2  -- Integer-Literal
        22E3  -- Integer-Literal
23_456_789  -- Integer-Literal
```

Im Standardfall werden Integer-Literale zur Basis 10 interpretiert. Man kann aber auch andere Basen zwischen 2 und 16 wählen:

```
  16#D2#  -- Integer-Literal zur Basis 16
   8#732#  -- Integer-Literal zur Basis 8
2#100110#  -- Integer-Literal zur Basis 2
```

Fließkommaliterale unterscheiden sich von Integer-Literalen dadurch, dass sie einen Punkt enthalten müssen und der Exponent auch negativ sein darf:

```
      1.0  -- Fließkommaliteral zur Basis 10
   33.6e-4  -- Fließkommaliteral zur Basis 10
8#43.6#e+4  -- Fließkommaliteral zur Basis 8
```

Zeichenliterale bestehen aus einzelnen Zeichen, umgeben von einfachen Anführungsstrichen:

```
'c'  -- Zeichenliteral
```

Zeichenfolgenliterale sind Folgen von ASCII-Zeichen, eingeschlossen von doppelten Anführungsstrichen. Wenn der doppelte Anführungsstrich in der Zeichenfolge vorkommen soll, dann ist er zu verdoppeln:

```
          "ABC"  -- Zeichenfolgenliteral
"ein ""-Zeichen"  -- Zeichenfolgenliteral
```

Bitfolgenliterale sind Zeichenfolgenliterale mit einem eingeschränkten Zeichensatz und vorangestelltem Indikator für die verwendete Basis. Auch hier dürfen Unterstriche zur Verbesserung der Lesbarkeit eingefügt werden:

```
B"1111_1111"  -- binäres Bitfolgenliteral
       O"73"  -- oktales Bitfolgenliteral
     X"F0A7"  -- hexadezimales Bitfolgenliteral
```

Physikalische Literale setzen sich aus einem Integer-Literal oder Fließkommaliteral und einem Bezeichner für eine Maßeinheit zusammen:

```
   35 Ns  -- physikalisches Literal
 10 kohm  -- physikalisches Literal
4.5 Sec  -- physikalisches Literal
```

1.4.1.3 Kommentare

Kommentare dürfen überall in den Programmtext eingefügt werden. Sie werden durch zwei aufeinanderfolgende Minuszeichen „`--`“ eingeleitet und reichen bis zum Ende der jeweiligen Zeile. Die oben in die Tabellen eingefügten Kommentare zu den verschiedenen Arten von Literalen sind also gültige VHDL-Kommentare (bis auf die Tatsache, dass in VHDL-Programmen keine Umlaute oder „ß“ erlaubt sind).

1.4.2 Daten und ihre Container

Sämtliche Daten werden in VHDL in Objekten gehalten. Es existieren drei verschiedene Arten von Objekten. Dies sind Konstanten, Signale und Variablen. Diese Objektarten unterscheiden sich wie folgt:

- **Konstanten** kann nur einmal ein Wert zugewiesen werden (nämlich bei der Deklaration der Konstanten). Dieser Wert bleibt während der gesamten Programmlaufzeit fest.
- **Signale** und Variable können im Verlauf einer Simulation eine Reihe von unterschiedlichen Werten annehmen. Signale werden benutzt, um Leitungen der zu modellierenden Hardware zu beschreiben. Wertzuweisungen an Signale werden in eine Liste zukünftiger Zustände des Signals eingetragen und wirken sich zu einem bestimmten Zeitpunkt in der Zukunft aus, aber niemals bereits zum Zeitpunkt der Zuweisung.
- **Variable** können mit den Variablen in üblichen Programmiersprachen verglichen werden. Eine Wertzuweisung an eine Variable hat zur Folge, dass die Variable im selben Moment diesen Wert annimmt. Variable können nur lokal in den VHDL-Prozessen existieren. Es kann also kein Datenaustausch zwischen den Prozessen über Variable stattfinden. Variable werden üblicherweise als lokaler Zwischenspeicher für temporäre Werte benutzt.

Neben der Zuordnung eines Objekts zu einer der drei Objektarten (Konstante, Signal oder Variable) ist jedem Objekt außerdem ein Datentyp zugeordnet. Hiermit wird festgelegt, welche Art von Daten das Objekt aufnehmen kann. In den bisher gesehenen Beispielen waren alle verwendeten Signale vom Datentyp „`Bit`“. VHDL bietet jedoch eine Vielfalt von verschiedenen und zum Teil auch recht komplexen Datentypen an. In diesem Kapitel sollen nur die einfachsten Datentypen behandelt werden, währen die anderen Möglichkeiten nur kurz aufgelistet werden.

Neben einer Reihe von vordefinierten Datentypen gibt es in VHDL auch die Möglichkeit, Datentypen selbst zu definieren. Um die große Zahl von verschiedenen Datentypen besser handhaben zu können, werden sie in vier große Gruppen, die teilweise wieder Untergruppen besitzen, aufgeteilt. Diese Gruppen – zusammen mit den jeweils vorhandenen vordefinierten Datentypen – werden in Tabelle 1.2 aufgelistet.

Eigene Datentypen kann der VHDL-Anwender mit der folgenden Anweisung definieren:

```
type neuer_Name is Datentypangabe;
```

- Skalare Datentypen
  - Aufzählungstypen
    vordefiniert: `Character`, `Bit`, `Boolean`, `Severity_Level`
  - Integer-Typen
    vordefiniert: `Integer`
  - Physikalische Typen
    vordefiniert: `Time`
  - Fließkommatypen
    vordefiniert: `Real`
- Zusammengesetzte Datentypen
  - Feldtypen
    vordefiniert: `String`, `Bit_Vector`
  - Strukturtypen
- Zeigertypen (werden hier nicht behandelt)
- Dateitypen (werden hier nicht behandelt)

Tabelle 1.2: Vordefinierte Datentypen von VHDL

Anschließend kann man die benutzerdefinierten Datentypen genau wie vordefinierte Datentypen verwenden, um Datenobjekte zu definieren. Im Folgenden sind Beispiele von Definitionen von Datenobjekten gegeben:

```
constant pi: Real := 3.14;
variable mem: neuer_Name;
signal s, t, u: Bit := '0';
```

1.4.2.1 Skalare Datentypen

Skalare Datentypen sind solche, deren Elemente nicht mehr weiter zerlegt werden können. VHDL kennt vier verschiedene Arten von skalaren Datentypen:

1.4.2.1.1 Aufzählungstypen

Ein Aufzählungstyp ist eine geordnete Menge von Bezeichnern oder Zeichen. Die Typdefinition erfolgt durch Aufzählung der Elemente des Typs. Diese Aufzählung wird in Klammern eingeschlossen. Die bereits vordefinierten Datentypen „`Bit`“, „`Boolean`“ und „`Severity_Level`“ sind wie folgt definiert:

```
type Bit is ('0', '1');
type Boolean is (False, True);
type Severity_Level is (Note, Warning, Error, Failure);
```

Mixturen aus Zeichen und Bezeichnern sind ebenfalls erlaubt. Außerdem ist zu beachten, dass derselbe Bezeichner oder dasselbe Zeichen in der Definition eines Typs zwar nur einmal vorkommen darf, dass es aber möglich ist, dasselbe Element in einer anderen Typdefinition noch einmal zu verwenden. In diesem Fall muss bei einer späteren Benutzung dieses Elements die Typzugehörigkeit aus dem Kontext ermittelt werden.

1.4.2.1.2 Integer-Typen

Der vordefinierte Typ „`Integer`“ beinhaltet die ganzen Zahlen in einem implementationsabhängigen Intervall. Dieses Intervall muss allerdings mindestens von −2.147.483.647 bis +2.147.483.647 reichen. Weitere Integer-Typen kann man durch eine Einschränkung dieses Intervalls definieren:

```
type byte_int is range 0 to 255;
type bit_index is range 31 downto 0;
```

1.4.2.1.3 Physikalische Typen

Physikalische Größen können in VHDL mit Hilfe von physikalischen Typen beschrieben werden. Ein gutes Beispiel für einen physikalischen Typ ist der vordefinierte Typ „`Time`“:

```
type Time is range -2147483647 to 2147483647
units
    Fs;                 -- femtosecond
    Ps   =  1000 Fs;    -- picosecond
    Ns   =  1000 Ps;    -- nanosecond
    Us   =  1000 Ns;    -- microsecond
    Ms   =  1000 Us;    -- millisecond
    Sec  =  1000 Ms;    -- second
    Min  =  60 Sec;     -- minute
    Hr   =  60 Min;     -- hour
end units;
```

1.4.2.1.4 Fließkommatypen

Fließkommatypen unterscheiden sich von Integer-Typen nur in der Verwendung von Fließkommaliteralen anstelle von Integer-Literalen. Die Auflösung von Fließkommazahlen ist im Language Reference Manual nicht festgelegt. Es muss aber mindestens ein Intervall von −1.0E38 bis +1.0E38 und eine Genauigkeit von 6 Dezimalstellen unterstützt werden.

```
type probability is range 0.0 to 1.0;
```

1.4.2.2 Zusammengesetzte Datentypen

Feldtypen und Strukturtypen werden zu den zusammengesetzten Datentypen zusammengefasst. Diese Typen erlauben es, mehrere Datenelemente zu einem neuen Datenelement zu kombinieren. Bei Feldtypen müssen alle Elemente vom gleichen Datentyp sein, während mit Hilfe von Strukturtypen auch Elemente unterschiedlicher Datentypen zusammengefasst werden können.

1.4.2.2.1 Feldtypen

Mit Hilfe von Feldtypen kann man sich ganze Felder von gleichartigen Werten definieren. Auf die einzelnen Elemente dieser Felder kann dann über Indizes zugegriffen werden, wie dies bei anderen Programmiersprachen auch üblich ist. Feldtypen werden wie folgt definiert:

```
type word is array (15 downto 0) of Bit;
```

Besonderheiten sind Feldtypen, bei denen die Größe des Feldes noch offen gelassen ist. Die beiden vordefinierten Feldtypen sind von dieser Art:

```
type String is array (Positive range <>) of Character;
type Bit_Vector is array (Natural range <>) of Bit;
```

(„`Positive`“ und „`Natural`“ sind vordefinierte Subtypen, siehe 2.8.3.)

Die Zeichenfolge „`<>`“ ist ein Platzhalter für ein Indexintervall, das erst dann angegeben wird, wenn der entsprechende Datentyp benutzt wird. Wenn man z. B. ein Feld von 16-Bit-Objekten benötigt, kann man dies wie folgt angeben:

```
signal vector: Bit_Vector (0 to 15);
```

1.4.2.2.2 Strukturtypen

Mit Hilfe von Strukturtypen kann man Datenelemente verschiedener Typen zu einem neuen Datenelement zusammenfassen. Die einzelnen Subelemente des neuen Datenelements werden dann mit Hilfe von Elementnamen referenziert.

```
type register_bank is record
  F0, F1: Real;
  R0, R1: Integer;
  A0, A1: Address;
  IR: Instruction;
end record;
```

Man erkennt, dass VHDL wahrlich eine reiche Auswahl an Datentypen bietet. Die Einzelheiten zur Definition und Verwendung aller möglichen Datentypen können dem Language Reference Manual [IEE88] entnommen werden.

1.4.3 Vordefinierte Operatoren

Nach der Deklaration von Datenobjekten möchte man mit diesen auch arbeiten können. Dazu benötigt man Operatoren, die verschiedene Datenobjekte miteinander verknüpfen. Für die vordefinierten Datentypen stellt VHDL die in Tabelle 1.3 zusammengefassten vordefinierten Operatoren zur Verfügung. In Tabelle 1.4 wird aufgelistet, welche Operationen für welche Datentypen definiert sind.

1.4.4 Die Struktur von VHDL-Programmen

Nachdem nun bekannt ist, mit welchen Elementen – Konstanten, Signalen und Variablen verschiedenster Datentypen – man in VHDL-Programmen arbeiten kann, wendet sich der folgende Abschnitt der Struktur dieser Programme zu.

Um die folgenden Darstellungen möglichst aussagekräftig zu machen, wird bei den weiteren Programmbeispielen einheitlich folgendes Schema verwendet: Schlüsselworte von VHDL erscheinen weiterhin in **`Fettschrift`**. Bezeichner u. ä., die beispielhaft in den Programmtext eingefügt wurden, erscheinen in normaler Schrift und Komponenten, die noch weiter aufgeschlüsselt werden müssen, werden in *`Schrägschrift`* gedruckt.

Gruppe	Symbol	Funktion
Arithmetik (binär)	`+`	Addition
	`-`	Subtraktion
	`*`	Multiplikation
	`/`	Division
	`mod`	Modulus
	`rem`	Rest
	`**`	Exponentiation
Arithmetik (unär)	`+`	unäres Plus
	`-`	unäres Minus
	`abs`	Absolutwert
Logik (binär)	`and`	logisches Und
	`or`	logisches Oder
	`nand`	Komplement zu `and`
	`nor`	Komplement zu `or`
	`xor`	logisches Exklusiv-Oder
Logik (unär)	`not`	Komplement
Vergleiche	`=`	Gleichheit
	`/=`	Ungleichheit
	`<`	kleiner als
	`>`	größer als
	`<=`	kleiner oder gleich
	`>=`	größer oder gleich
Konkatenation	`&`	Konkatenation

Tabelle 1.3: In VHDL vordefinierte Operatoren

Operatoren	Datentyp der Operanden
`+ -` (binär)	Integer, Fließkomma, physikalisch (beide Operanden müssen denselben Typ haben)
`+ - abs` (unär)	Integer, Fließkomma, physikalisch
`mod rem`	Integer (beide Operanden müssen denselben Typ haben)
`*`	Integer, Fließkomma (beide Operanden müssen denselben Typ haben) oder ein Operand physikalisch und ein Operand Integer oder Fließkomma
`/`	Integer, Fließkomma (beide Operanden müssen denselben Typ haben) oder linker Operand physikalisch und rechter Operand Integer oder Fließkomma oder beide Operatoren physikalisch
`**`	linker Operand Integer oder Fließkomma und rechter Operand Integer (negativ ist nur erlaubt, wenn der linke Operand ein Fließkommaoperand ist)
`= /=`	beliebiger Datentyp außer Dateityp (beide Operanden müssen denselben Typ haben)
`< > <= >=`	beliebiger skalarer Datentyp oder eindimensionale Felder mit Elementen eines Aufzählungstyps oder Integer-Typs (beide Operanden müssen denselben Typ haben)
`and or nand nor xor`	`Bit` oder `Boolean` oder eindimensionale Felder mit Elementen vom Typ `Bit` oder `Boolean` (beide Operanden müssen denselben Typ haben)
`not`	`Bit` oder `Boolean` oder eindimensionale Felder mit Elementen vom Typ `Bit` oder `Boolean`
`&`	eindimensionaler Feldtyp oder Element dieses Feldtyps

Tabelle 1.4: Erlaubte Datentypen zu den vordefinierten Operatoren

1.4.4.1 Bausteindeklarationen

Wie schon in Abschnitt 1.2.1 beschrieben, setzt sich ein Bausteinentwurf aus einer Bausteindeklaration und einem Architekturrumpf zusammen. Eine Bausteindeklaration baut sich wie folgt auf:

```
entity bausteinname is
  port ( Schnittstellenliste );
  generic ( Parameterliste );
  Deklarationen
  begin Anweisungen
end bausteinname;
```

Eine Bausteindeklaration wird durch die Konstruktionen „**`entity bausteinname is`**“ und „**`end`**“ geklammert. Alle anderen Teile sind optional. Eine minimale Bausteindeklaration ist also die folgende:

```
entity bausteinname is
end;
```

Diese Bausteindeklaration beschreibt einen Baustein, der keinen Kontakt zur Außenwelt hat. Derartige Bausteine werden häufig als Wurzelelement einer Hierarchie benutzt, die einen kompletten Aufbau inklusive Testumgebung beschreibt. Zwischen den beiden gezeigten Zeilen können die folgenden Teile eingefügt werden:

- **Schnittstellenlisten** dienen der Definition von Verbindungen eines Bausteins mit der Außenwelt. Diese Schnittstellenlisten werden durch das Schlüsselwort „`port`“ eingeleitet. Beispiele für derartige Listen sind in den Programmbeispielen in Abschnitt 1.2.3 enthalten. Die generelle Struktur von Schnittstellenlisten wird im Anschluss an diese Aufzählung erläutert.
- **Parameterlisten** erlauben es dem Entwickler, die modellierten Bausteine zu parametrisieren. Man kann z. B. die Verzögerungszeiten eines Bausteins über einen solchen Parameter flexibel halten oder sogar die Anzahl von Ein- und Ausgangsleitungen hiermit festlegen. Die dem Schlüsselwort „`generic`“ folgende Liste ist der Schnittstellenliste sehr ähnlich.
- Anschließend ist es möglich, eine Reihe von **Deklarationen** bereits in der Bausteindeklaration vorzunehmen. Hier kann man z. B. Typen, Konstanten oder auch interne Signale deklarieren. Dies hat den Vorteil, dass diese Deklarationen sich in den verschiedenen zugehörigen Architekturrümpfen nicht mehr unterscheiden können.
- Eine Bausteindeklaration kann nach dem Schlüsselwort „`begin`“ sogar gewisse **Anweisungen** enthalten. Diese Anweisungen sollen aber nicht das dynamische Verhalten des Bausteins beschreiben. Hier kann man Anweisungen unterbringen, die bestimmte Bedingungen überprüfen. Wenn z. B. zwei Eingangsleitungen eines Bausteins nicht gleichzeitig gesetzt sein dürfen, dann kann diese Überprüfung bereits hier notiert werden. Dies hat wiederum den Vorteil, dass nicht jeder einzelne Architekturrumpf eine solche Überprüfung enthalten muss.

1.4.4.2 Schnittstellenlisten

Schnittstellenlisten werden an verschiedenen Stellen innerhalb von VHDL-Beschreibungen benutzt, um die Schnittstelle eines Moduls zu seiner Umgebung zu beschreiben. In

den vorangehenden Abschnitten sind Schnittstellenlisten bereits in der Bausteindeklaration und als Bestandteil einer Komponentendeklaration (siehe Beispiel 1.2.3) vorgekommen. Schnittstellenlisten sind durch Semikolons unterteilte Listen einzelner Schnittstellenelemente. Jedes Schnittstellenelement kann wiederum mehrere Schnittstellenobjekte enthalten. Alle Objekte eines Elements haben die gleichen Eigenschaften. Die in einer Schnittstellenliste beschriebenen Eigenschaften sind:

- die Objektart des Schnittstellenobjekts (`signal`, `constant` oder `variable`),
- die Richtung des Datenflusses durch die Schnittstelle (`in`, `out`, `inout` oder `buffer`),
- der Datentyp der Schnittstelle.

Eine wichtige Eigenschaft eines Schnittstellenelements ist die Richtung des Datenflusses durch dieses Element. VHDL stellt dafür verschiedene Modi zur Verfügung. Für jedes Schnittstellenelement muss festgelegt werden, welcher Modus zugeordnet werden soll. Bei der Analyse des Programms kann dann überprüft werden, ob die Zusammenschaltung der Bausteine auch in dieser Hinsicht korrekt ist. Die in VHDL existierenden Modi zur Beschreibung der Datenflussrichtung sind in Tabelle 1.5 zusammengefasst.

Modus	Bedeutung
`in`	Eingangsleitung; nur lesender Zugriff erlaubt (ist Default)
`out`	Ausgangsleitung; nur schreibender Zugriff erlaubt
`inout`	bidirektionale Leitung; lesender und schreibender Zugriff erlaubt
`buffer`	Ausgangsleitung; schreibender und lesender Zugriff erlaubt

Tabelle 1.5: Modi zur Beschreibung der Datenflussrichtung

Der Modus „`buffer`“ kann verwendet werden, wenn eigentlich eine Ausgangsleitung gemeint ist, man deren Wert aber noch für weitere, interne Berechnungen verwenden will.

Jedes Schnittstellenelement hat somit das folgende Aussehen:

Objektart Bezeichnerliste : Richtung Datentyp := Default-Wert

Bis auf die Bezeichnerliste und die Angabe des Datentyps sind alle Elemente optional. Je nach Verwendung der Schnittstellenliste ergeben sich bestimmte Einschränkungen für die einzelnen Komponenten eines Schnittstellenelements:

- Alle Objekte einer Schnittstellenliste in einer `port`-Anweisung müssen die Objektart `signal` haben (deshalb gibt man dort die Objektart meist gar nicht an).
- Alle Elemente einer Schnittstellenliste in einer `generic`-Anweisung müssen die Objektart `constant` und die Datenflussrichtung `in` haben.

Hier einige Beispiele von gültigen Schnittstellenelementen:

```
carry_in: Bit
a, b: Bit
constant pi: Real := 3.14
signal a, b: in Bit := '0'
```

Komplette Schnittstellenlisten können z. B. wie folgt aussehen:

```
generic (wordlength: Integer; delay1: Time);
port (base: in Real; exp: in Positive; result: out Real);
```

1.4.4.3 Architekturrümpfe

Zu jeder Bausteindeklaration muss mindestens ein Architekturrumpf existieren. Der Aufbau des Rumpfs sieht wie folgt aus:

```
architecture rumpfname of bausteinname is
  Deklarationen
begin
  Anweisungen
end rumpfname;
```

Der Deklarationsteil kann z. B. Typdeklarationen oder Konstantendeklarationen enthalten. Eine weitere Möglichkeit ist die Deklaration von Komponenten, die im Anweisungsteil instanziiert werden (siehe Beispiel 1.2.3). Im Anweisungsteil folgt dann die eigentliche Realisierung. Dieser Teil besteht aus einer Folge von Prozessen (im Fall einer Verhaltensbeschreibung) oder von Komponenteninstanziierungen (im Fall einer Strukturbeschreibung). Diese beiden Möglichkeiten werden in den folgenden beiden Abschnitten behandelt.

1.4.5 Beschreibung von Verhalten

Das wichtigste Konstrukt zur Beschreibung von Verhalten in VHDL ist die `process`-Anweisung. Innerhalb dieser Anweisung (oder innerhalb eines `process`-Blocks) wiederum ist die `wait`-Anweisung von besonderer Bedeutung, wenn es darum geht, das Verhalten von Bausteinen zu synchronisieren. Aus diesem Grund werden in den folgenden beiden Abschnitten diese Anweisungen behandelt, bevor im dritten Abschnitt die Zuweisungen und im vierten die anderen Kontrollstrukturen von VHDL diskutiert werden.

1.4.5.1 Die `process`-Anweisung

Eine `process`-Anweisung in VHDL definiert einen Prozess wie er in Definition 1.3.2 beschrieben ist. Die Struktur einer `process`-Anweisung ist der eines Architekturrumpfes sehr ähnlich:

```
prozessmarke:
  process
    Deklarationen
  begin
    Anweisungen
  end process;
```

Mit Hilfe von „`prozessmarke`“ kann man jedem Prozess einen Namen geben. Die Marke ist wie alle anderen Elemente außer den Schlüsselwörtern optional. Im Deklarationsteil eines Prozesses kann man z. B. lokale Datenobjekte definieren. So können Variable nur hier deklariert werden, da sie immer lokal zu einem Prozess gehören müssen.

Der Anweisungsteil beschreibt das Verhalten des Prozesses. Solange der Prozess aktiv ist, werden die Anweisungen dieses Teils sequentiell abgearbeitet. Dies geschieht so lange, bis der Prozess auf eine `wait`-Anweisung trifft. Wenn die letzte Anweisung in einem Prozess abgearbeitet wurde, dann wird danach wieder zur ersten Anweisung gesprungen. Zum genaueren Verständnis von Prozessen muss zunächst die `wait`-Anweisung erläutert werden.

1.4.5.2 Die `wait`-Anweisung

Die `wait`-Anweisung kontrolliert den Zustand von Prozessen. Wenn eine solche Anweisung ausgeführt wird, dann wird der entsprechende Prozess suspendiert, und gleichzeitig wird festgelegt, wann er wieder aktiviert werden kann. Für die Aktivierung gibt es drei verschiedene Arten von Bedingungen. Dazu kann die `wait`-Anweisung neben der Standardform drei weitere Formen annehmen:

```
wait ;
wait on Signalliste ;
wait until Bedingung ;
wait for Dauer ;
```

Die erste gezeigte Form einer `wait`-Anweisung führt zu einer unbegrenzten Suspendierung des Prozesses. In der laufenden Simulation wird dieser Prozess nicht mehr aktiviert. In der zweiten Form wird eine Menge von Signalen angegeben, die den Prozess wieder aktivieren können. Wenn sich der Wert eines der angegebenen Signale ändert, dann wird der Prozess wieder aktiviert. Mit Hilfe der dritten Form kann man Bedingungen formulieren, die erfüllt sein müssen, damit der Prozess wieder aktiviert werden kann, und in der vierten Form wird eine maximale Dauer für die Suspendierung angegeben. Man kann die verschiedenen Formen beliebig mischen, wie das folgende Beispiel zeigt:

Beispiel 1.4.1

```
process begin
  wait on x, y until enable = '1';
  sum <= x or y;
end process;
```

Dieser Prozess wird immer dann aktiviert, wenn ein Ereignis auf einem der Signale „`x`“, „`y`“ oder „`enable`“ auftritt und gleichzeitig der Wert von „`enable`“ gleich `'1'` ist.

Anstelle von expliziten `wait`-Anweisungen innerhalb eines Prozesses kann man einem Prozess auch eine sogenannte Sensibilitätsliste zuordnen. Wenn eine solche Liste gegeben ist, dann sind weitere `wait`-Anweisungen im Prozessrumpf allerdings nicht mehr erlaubt.

Beispiel 1.4.2

```
process(x, y) begin
  prod <= x and y;
end process;
```

Im gezeigten Beispiel enthält die Sensibilitätsliste die Signale „`x`“ und „`y`“. Das Beispiel ist damit äquivalent zur folgenden Notation:

Beispiel 1.4.3

```
process begin
  prod <= x and y;
  wait on x, y;
end process;
```

1.4.5.3 Zuweisungen

Eine weitere sehr wichtige und in VHDL häufig verwendete Anweisung ist die Zuweisungsanweisung. Genau genommen gibt es in VHDL zwei verschiedene Zuweisungsanweisungen, da es auch zwei verschiedene Objektarten gibt, denen im Verlauf der Simulation Werte zugewiesen werden können. Dies sind Variable und Signale. Es ist sehr wichtig, diese Zuweisungen zu unterscheiden, da die Semantik recht unterschiedlich ist.

1.4.5.3.1 Variablenzuweisungen

Wie bereits in Abschnitt 1.4.2 beschrieben, kann man die Zuweisung an Variablen mit der entsprechenden Anweisung in anderen Programmiersprachen vergleichen. Der Ausdruck auf der rechten Seite des Zuweisungssymbols wird ausgewertet und der berechnete Wert wird gleich anschließend der Variablen zugewiesen. Eine Variablenzuweisung in VHDL hat die folgende Form:

`variablen_name :=` *Ausdruck*`;`

Zu beachten ist lediglich, dass im Gegensatz zur Signalzuweisung hier das Zuweisungssymbol „`:=`“ benutzt wird.

1.4.5.3.2 Zuweisungen an Signale

Zuweisungen an Signale wurden bereits im Abschnitt 1.3.2 diskutiert. Die Zuweisung des ermittelten Wertes findet auf keinen Fall im aktuellen Simulationszyklus statt, sondern entweder zu einem Zeitpunkt, der mit Hilfe der `after`-Klausel spezifiziert wurde, oder im folgenden Simulationszyklus, wenn keine `after`-Klausel angegeben wurde. Die Signalzuweisung kann also eine der beiden folgenden Formen annehmen:

`signal_name <=` *Ausdruck*`;`
`signal_name <=` *Ausdruck* **`after`** *Dauer*`;`

Hierbei muss die Auswertung von *Ausdruck* einen Wert von dem Datentyp ergeben, der dem Signal „`signal_name`“ zugeordnet ist. *Dauer* darf ein beliebiger Ausdruck vom Datentyp „`Time`“ sein.

Zu beachten ist außerdem, dass normalerweise jedes Signal seine Werte nur von einem einzigen Prozess erhalten darf. Da Signale im Deklarationsteil eines Architekturrumpfs deklariert werden und dann von allen Prozessen aus zugreifbar sind, die sich im Anweisungsteil dieses Rumpfs befinden, ist dies nicht von vornherein sichergestellt. Es ist zulässig, dass mehrere Zuweisungen innerhalb eines Prozesses Werte an dasselbe Signal zuweisen:

Beispiel 1.4.4

```
architecture bsp of ent is
  signal s: Integer;
begin
  process begin
    s <= 1 after 1 Ns;
    s <= 2 after 2 Ns;
    wait;
  end process;
end bsp;
```

Nicht zulässig ist dagegen die folgende Architektur, da hier von zwei verschiedenen Prozessen an ein und dasselbe Signal zugewiesen wird:

Beispiel 1.4.5

```
architecture bsp of ent is
  signal s: Integer;
begin
  process begin
    s <= 1 after 1 Ns;
    wait;
  end process;
  process begin
    s <= 2 after 2 Ns; -- UNZULÄSSIG
    wait;
  end process;
end bsp;
```

Diese Konstruktion ist nur dann zulässig, wenn das Signal „s“ ein „berechnetes Signal“ ist. Darauf wird in Abschnitt 2.8.4 eingegangen.

1.4.5.4 Kontrollstrukturen

Was jetzt noch fehlt, um das Verhalten von Bausteinen beschreiben zu können, sind Strukturen, die die Abarbeitungsreihenfolge der verschiedenen Anweisungen innerhalb eines Prozesses kontrollieren. VHDL stellt hier verschiedene Anweisungen zur Verfügung, die den entsprechenden Anweisungen in üblichen Programmiersprachen recht ähnlich sind. Aus diesem Grund werden die Kontrollstrukturen hier nur kurz aufgelistet.

1.4.5.4.1 Bedingte Ausführung

Zur bedingten Ausführung bestimmter Anweisungen gibt es, wie in vielen andren Programmiersprachen auch, die `if`-Anweisung. Sie hat in VHDL folgende Syntax:

```
if Bedingung then
  Anweisungen
end if;
```

Damit kann man das Beispiel 1.4.1 auch anders formulieren:

Beispiel 1.4.6

```
  process begin
    wait on x, y;
    if enable = '1' then
      sum <= x or y;
  end process;
```

Bedingung muss ein Ausdruck sein, bei dessen Auswertung sich ein Wert vom Typ „`Boolean`“ ergibt, und *Anweisungen* ist eine beliebig lange Liste von Anweisungen. Für den Fall, dass bei nicht erfüllter Bedingung andere Aktionen erfolgen sollen, gibt es noch zwei weitere Formen der `if`-Anweisung:

```
if Bedingung then
  Anweisungen
else
  Anweisungen
end if;

if Bedingung then
  Anweisungen
elsif Bedingung then
  Anweisungen
else
  Anweisungen
end if;
```

Bei Verwendung der letzten Form dürfen beliebig viele `elsif`-Zweige eingefügt werden. Außerdem ist der abschließende `else`-Zweig optional.

Wenn je nach Ergebnis der Auswertung eines Ausdrucks eine ganze Liste von verschiedenen Anweisungen ausgeführt werden soll, dann empfiehlt sich die Verwendung der `case`-Anweisung. Ein Beispiel ist das folgende:

Beispiel 1.4.7

```
case opcode is
  when 1      => result <= a + b;
  when 2      => result <= a - b;
  when 3 | 4  => result <= a;
  when 5 to 7 => result <= a;
  when others => result <= b;
end case;
```

Mit dieser Anweisung ließe sich das Verhalten einer – etwas eigenartigen – ALU beschreiben. Das Beispiel ist so gewählt, um alle möglichen Arten von `when`-Zweigen zu demonstrieren. Die ersten beiden Zweige der Anweisung werden ausgeführt, wenn die Auswertung des Ausdrucks – in diesem Fall der aktuelle Wert von „`opcode`“ – genau den im jeweiligen Zweig angegebenen Wert trifft. Im dritten Zweig ist gezeigt, wie man eine Liste von möglichen Werten angeben kann, während der vierte Zweig die Angabe eines Bereichs von Werten demonstriert. Wenn keine der Angaben in den anderen Zweigen die Auswertung trifft, dann wird (falls vorhanden) der `others`-Zweig ausgeführt.

Die Anzahl der Zweige in der `case`-Anweisung ist natürlich beliebig. Wenn ein `others`-Zweig benutzt wird, dann muss dies der letzte Zweig in der Liste sein. Außerdem ist zu beachten, dass die den einzelnen Zweigen zugeordneten Wertebereiche für den steuernden Ausdruck sich gegenseitig ausschließen müssen.

1.4.5.4.2 Iterative Ausführung

Zur wiederholten Ausführung einer Liste von Anweisungen stellt VHDL eine `loop`-Anweisung zur Verfügung. Die drei Varianten dieser Anweisung sind die einfache `loop`-Anweisung, die `for`-Schleife und die `while`-Schleife. Die einfache `loop`-Anweisung hat die folgende Struktur:

```
schleifenmarke: loop
  Anweisungen
end loop schleifenmarke;
```

Die „schleifenmarke“ ist optional. Die Verwendung einer loop-Anweisung nur mit den bisher beschriebenen Anweisungen führt zu einer Endlosschleife, da es bisher keine Möglichkeit gibt, eine Abbruchbedingung zu formulieren. Dies leistet die exit-Anweisung. Diese Anweisung gibt es in zwei Formen:

```
exit schleifenmarke;
exit schleifenmarke when Bedingung;
```

Im ersten Fall wird die bezeichnete Schleife verlassen, während im zweiten Fall die Bedingung erfüllt sein muss, damit dies geschieht. Auch hier ist „schleifenmarke“ optional. Wird keine Marke angegeben, dann wird die direkt umgebende Schleife verlassen.

Wenn man die Bedingung zum Verlassen der Schleife von vornherein angeben möchte, dann kann man eines der beiden folgenden Konstrukte verwenden:

```
schleifenmarke:
while Bedingung loop
  Anweisungen
end loop schleifenmarke;

schleifenmarke:
for schleifenvariable in Intervall loop
  Anweisungen
end loop schleifenmarke;
```

Im zweiten Fall wird eine Schleifenvariable vom Typ „Integer“ implizit deklariert. Diese Variable muss also vorher nicht existent sein.

Eine weitere nützliche Anweisung innerhalb von Schleifen ist die next-Anweisung. Auch hier gibt es wieder zwei Formen:

```
next schleifenmarke;
next schleifenmarke when Bedingung;
```

Wieder ist „schleifenmarke“ optional und bei Auslassung wird die innerste Schleife angesprochen. Diese Anweisung bewirkt – im zweiten Fall nur bei erfüllter Bedingung – einen Sprung hinter die letzte Anweisung innerhalb der betreffenden Schleife. Anschließend wird also z. B. bei einer while-Schleife die Bedingung überprüft und je nach Resultat dieser Prüfung entweder mit der Ausführung des Schleifenrumpfs von vorne begonnen oder nach der Schleife fortgefahren.

1.4.6 Beschreibung von Struktur

Zur Beschreibung von Strukturen – d. h. von der Verschaltung mehrerer Bausteine zu einer größeren Einheit – benötigt man im Wesentlichen eine Darstellung der einzelnen Komponenten und der Verbindungen dazwischen. Ein Beispiel für eine solche Strukturbeschreibung wurde bereits in Beispiel 1.2.3 gegeben.

Zunächst muss vereinbart werden, welche Typen von Bausteinen in der Beschreibung verwendet werden sollen. Dies geschieht im Deklarationsteil des Architekturrumpfes mit Hilfe der component-Anweisung. Diese Anweisung ähnelt einer Bausteindeklaration, da

hier ebenfalls eine Schnittstellenliste und eine Parameterliste verwendet werden kann. Die Bedeutung dieser Anweisung unterscheidet sich aber ganz wesentlich von einer Bausteindeklaration. Während letztere einen Bausteintyp definiert, der später (im Architekturrumpf) genau spezifiziert wird, ist eine `component`-Definition nur eine Schablone für einen Bausteintyp. Die Realisierung dieses Bausteintyps wird nicht spezifiziert. Stattdessen wird später ein VHDL-Baustein mit diesem Bausteintyp assoziiert – im einfachsten Fall über die Namensgleichheit.

Die mit Hilfe der `component`-Anweisung spezifizierte Schablone wird benötigt, damit ein Analyseprogramm die folgende Strukturbeschreibung auf Korrektheit überprüfen kann, ohne die tatsächliche Spezifikation des entsprechenden Bausteintyps zu kennen. Es wäre für den Hardware-Entwickler sehr hinderlich, wenn er immer zuerst alle Bausteintypen definieren und analysieren lassen müsste, bevor er eine Strukturbeschreibung erstellen könnte, die diese Bausteintypen verwendete.

Eine `component`-Anweisung hat die folgende generelle Form:

```
component comp_typname
  generic Parameterliste;
  port Schnittstellenliste;
end component;
```

Neben den Bausteinen, aus denen die Struktur aufgebaut werden soll, benötigt man außerdem die entsprechenden Verbindungsleitungen. Diese werden durch `signal`-Definitionen mit dem benötigten Datentyp definiert, z. B.:

```
signal l1, l2: Bit;
```

Nachdem alle Deklarationen gemacht sind, kann man im Anweisungsteil mit dem Aufbau der Struktur beginnen. Dazu ist nur eine einzige Anweisung notwendig: die Komponenteninstanziierung. Sie hat die folgende Form:

```
comp_name : comp_typname
  generic map Assoziationsliste;
  port map Assoziationsliste;
```

Beispiel 1.4.8

```
s2 : half_adder
  port map (y, carry_in, z, sum);
```

„`comp_name`“ ist ein Bezeichner, der dem mit dieser Anweisung erzeugten Element eigen ist. Damit kann jedes Element der erzeugten Struktur einzeln referenziert werden. „`comp_typname`“ ist einer der Bezeichner, die im Deklarationsteil mit einer `component`-Anweisung vereinbart wurden. Hiermit wird also festgelegt, wie das erzeugte Element aussieht, also z. B. welche Ein- und Ausgänge es hat. Anschließend muss angegeben werden, wie dieses Element mit den anderen Elementen und mit der Schnittstelle der zu erzeugenden Struktur verbunden ist. Dies geschieht hinter den Schlüsselworten `port map` mit Hilfe einer Assoziationsliste. Eine solche Liste wird auch in der `generic-map`-Anweisung und an anderen Stellen von VHDL-Programmen verwendet. Deshalb ist ihr ein eigener Abschnitt gewidmet.

1.4.6.1 Assoziationslisten

An verschiedenen Stellen eines VHDL-Programms ist es notwendig, einem formalen oder lokalen Element eines Bausteins einen aktuellen Wert oder ein aktuelles Element zuzuweisen. Dies ist z. B. der Fall, wenn in einer Strukturbeschreibung ein lokales Element einer Schnittstellenbeschreibung mit einem aktuellen Signal verbunden werden soll. Für derartige Zuordnungen wird in VHDL die Assoziationsliste verwendet, deren zwei mögliche Formen anhand des folgenden Beispiels beschrieben werden sollen.

Es sei die Deklaration eines Bausteintyps „`and_gate`“ und der drei Signale „`a`“, „`b`“ und „`c`“ – der Einfachheit halber alle vom Datentyp „`Bit`“ – gegeben:

Beispiel 1.4.9

```
component and_gate
  port (in1, in2 : in Bit;
        result   : out Bit);
end component;
signal a, b, c: Bit;
```

Eine Instanz vom Typ „`and_gate`“ kann z. B. wie folgt erzeugt werden:

Beispiel 1.4.10

```
and1 : and_gate
  port map (a, b, c);
```

Die hier gezeigte Assoziationsliste „`(a, b, c)`“ ist nach der Methode „Assoziation durch Position“ erstellt worden. Hierbei ist die Reihenfolge der Schnittstellenelemente in der `component`-Anweisung maßgebend. Die aktuellen Elemente „`a`“, „`b`“ und „`c`“ werden aufgrund ihrer Position in der Liste mit den lokalen Elementen „`in1`“, „`in2`“ und „`result`“ respektive verbunden. Neben dieser Methode existiert die Methode „Assoziation durch Namen“. Hierbei werden sowohl die Namen der aktuellen als auch der lokalen Elemente angegeben, so dass die Position nicht mehr relevant ist. Beispiel 1.4.11 hat also die gleiche Bedeutung wie Beispiel 1.4.10.

Beispiel 1.4.11

```
and1 : and_gate
  port map (result => c, in1 => a, in2 => b);
```

Es ist auch möglich, beide Formen zu mischen. In diesem Fall darf allerdings nach einem Element der Methode „Assoziation durch Namen“ kein Element der Methode „Assoziation durch Position“ mehr auftreten.

2 Vertiefungen in VHDL

Nachdem die wichtigsten Elemente von VHDL im vorhergehenden Kapitel eingeführt wurden, soll nun zur Vertiefung ein etwas größeres Beispiel behandelt werden. In diesem Beispiel werden viele bekannte Elemente auftauchen, die an dieser Stelle keiner Erläuterung mehr bedürfen. Es wird allerdings auch eine Anzahl von Elementen geben, die bisher noch nicht besprochen wurden. Diese Elemente und die zugrunde liegenden Konzepte sollen in diesem Kapitel eingeführt und diskutiert werden.

2.1 Modellierung einer Ampelschaltung

Sehr groß kann auch das hier verwendete Beispiel nicht sein, da die zur Verfügung stehende Zeit einen engen Rahmen setzt. Es wurde ein Standardbeispiel gewählt, das bereits in [MC80] verwendet wurde. In diesem Beispiel wird eine Ampelanlage modelliert, die die Kreuzung zwischen einer Haupt- und einer Nebenstraße kontrolliert. Es soll dabei von einer Vorrangschaltung ausgegangen werden, die die Ampel der Nebenstraße nur bei Bedarf auf Grün schaltet. Der Bedarf wird z. B. über eine Induktionsschleife festgestellt. Die Nebenstraßenampel soll so lange grün bleiben wie weiterer Bedarf besteht, maximal jedoch für eine gewisse Höchstdauer. Dann wechselt die Freigabe wieder zur Hauptstraße. Weiterhin soll es eine Mindestdauer für die Grünphase der Hauptstraße geben.

Eine derartige Schaltung wird üblicherweise mit Hilfe eines endlichen Automaten beschrieben. Dazu wird zunächst eine Menge von Zuständen definiert, die das System annehmen kann. Anschließend wird für jeden Zustand und jede mögliche Kombination von Eingangssignalen festgelegt, welchen Folgezustand das System einnehmen soll und wie in diesem Zustand die Ausgangssignale beschaltet werden wollen. Eingangssignal ist in diesem Beispiel eine Leitung, die anzeigt, ob sich Autos auf der Nebenstraße befinden. Ausgangssignale müssen erzeugt werden, um die beiden Ampelpaare zu steuern. Die Zustände und Zustandsübergänge der in diesem Abschnitt modellierten Ampelschaltung sind der Abbildung 2.1 zu entnehmen.

Im Anfangszustand „`hauptstr_gruen`“ sind die Ampeln der Hauptstraße auf Grün und die der Nebenstraße auf Rot geschaltet. Wenn eine Anforderung vorliegt und die Grünphase bereits lange genug gedauert hat (`timeout_vorbei`), dann erfolgt ein Übergang in den Zustand „`hauptstr_gelb`“. Nachdem in diesem Zustand eine gewisse Zeit verstrichen ist (`gelb_vorbei`), wird in den Zustand „`nebenstr_gruen`“ weitergeschaltet. Die Hauptstraßenampel wird rot und die Nebenstraßenampel grün. (Es wird eine amerikanische Ampelanlage modelliert: der Lichtwechsel von Rot nach Grün erfolgt ohne die Zwischenphase Rot/Gelb; eine Erweiterung auf deutsche Verhältnisse bleibt dem Leser überlassen.) Der Zustand „`nebenstr_gruen`“ bleibt so lange bestehen, bis entweder keine Anforderung mehr vorliegt oder ein Zeitlimit überschritten ist (`timeout_vorbei`: die beiden Limits „minimal Hauptstraße grün“ und „maximal Nebenstraße grün“ haben denselben Wert). Wenn dies der Fall ist, erfolgt ein weiterer Zustandsübergang in den Zustand „`nebenstr_gelb`“, und nach der entsprechenden Wartezeit wird wieder Zustand „`hauptstr_gruen`“ aktiv.

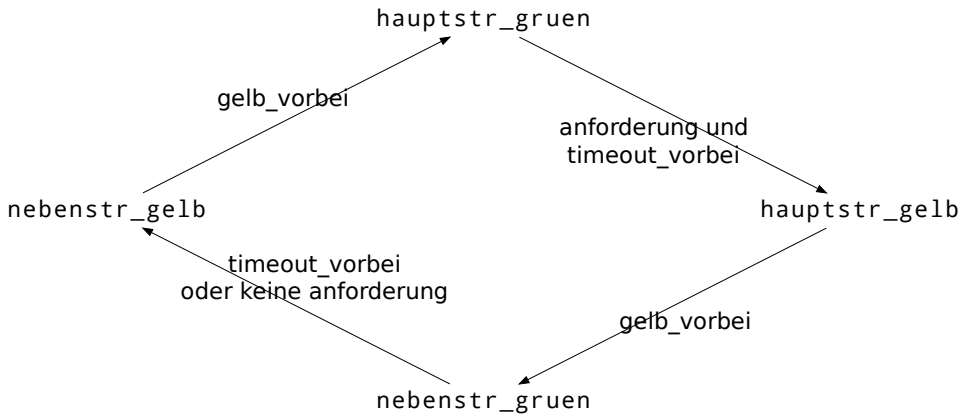

Abbildung 2.1: Endlicher Automat zur Kontrolle der Ampelanlage

Im Folgenden wird eine mögliche VHDL-Beschreibung der Ampelschaltung gezeigt. In den nächsten Abschnitten wird auf die bisher noch nicht eingeführten Elemente eingegangen, die in dieser Beschreibung benutzt werden.

Beispiel 2.1.1

```
package ampelpaket is
  type farben is (gruen, gelb, rot, unbekannt);
  type zustaende is (
      hauptstr_gruen,
      hauptstr_gelb,
      nebenstr_gruen,
      nebenstr_gelb);
end ampelpaket;


use work.ampelpaket.all;
entity ampelschaltung is
  generic (timeout: Time; gelbdauer: Time);
  port (anforderung: in Boolean;
        hauptstr_ampel: out farben;
        nebenstr_ampel: out farben);
end ampelschaltung;

architecture eins of ampelschaltung is
  signal zustand: zustaende := hauptstr_gruen;
  signal timeout_vorbei: Boolean := False;
  signal gelb_vorbei: Boolean := False;
  signal taktstart: Boolean := False;
begin  -- Rumpf der Architektur „eins"
  Kontrolle:  -- endlicher Automat zur Kontrolle des Systems
  process begin
    case zustand is
      when hauptstr_gruen =>
        if anforderung and timeout_vorbei then
          taktstart <= not taktstart;
```

```
          zustand <= hauptstr_gelb;
        end if;
      when hauptstr_gelb =>
        if gelb_vorbei then
          taktstart <= not taktstart;
          zustand <= nebenstr_gruen;
        end if;
      when nebenstr_gruen =>
        if not anforderung or timeout_vorbei then
          taktstart <= not taktstart;
          zustand <= nebenstr_gelb;
        end if;
      when nebenstr_gelb =>
        if gelb_vorbei then
          taktstart <= not taktstart;
          zustand <= hauptstr_gruen;
        end if;
    end case;
    wait on anforderung, timeout_vorbei, gelb_vorbei;
  end process;
  Hauptstrassenampel:
  with zustand select hauptstr_ampel <=
    gruen when hauptstr_gruen,
    gelb when hauptstr_gelb,
    rot when nebenstr_gruen | nebenstr_gelb;
  Nebenstrassenampel:
  with zustand select nebenstr_ampel <=
    gruen when nebenstr_gruen,
    gelb when nebenstr_gelb,
    rot when hauptstr_gruen | hauptstr_gelb;
  Taktgeber:
  process begin
    timeout_vorbei <= false, true after timeout;
    gelb_vorbei <= false, true after gelbdauer;
    wait on taktstart;
  end process;
end eins;
```

2.2 Paketdeklaration und Paketrumpf

Gleich der erste Block des oben gezeigten Beispiels ist eine bisher noch nicht vorgestellte Entwurfseinheit. In Abschnitt 1.2.1 wurden die Entwurfseinheiten „Bausteindeklaration“ und „Architekturrumpf“ eingeführt. Des Weiteren kennt VHDL die Paketdeklaration („Package Declaration“), den Paketrumpf („Package Body“) und die Konfigurationsdeklaration („Configuration Declaration“). Letztere Einheit wird in Abschnitt 2.8.6 behandelt.

VHDL-Pakete können benutzt werden, um häufig verwendete Definitionen in einem übergeordneten Block zusammenzufassen. Wenn z. B. eine Bausteinbibliothek in einer mehrwertigen Logik beschrieben werden soll und diese Logik in VHDL nicht vordefiniert ist,

dann wäre es sehr mühsam, den zugrunde liegenden Datentyp in jeder Entwurfseinheit von Neuem zu definieren. Stattdessen würde man die entsprechenden Definitionen in einem Paket zusammenfassen und mit Hilfe einer `use`-Anweisung (wird in Abschnitt 2.3 erläutert) auf dieses Paket Bezug nehmen – wie es auch in Beispiel 2.1.1 gezeigt wird.

Ein VHDL-Paket besteht aus einer Paketdeklaration und einem Paketrumpf. Es ist also vom Konzept her einem VHDL-Baustein sehr ähnlich, mit der Ausnahme, dass es zu jeder Paketdeklaration nur einen Paketrumpf gibt.

Paketdeklaration und Paketrumpf haben die folgende Struktur:

```
package paketname is
  Deklarationen
end paketname;


package body paketname is
  Deklarationen
end paketname;
```

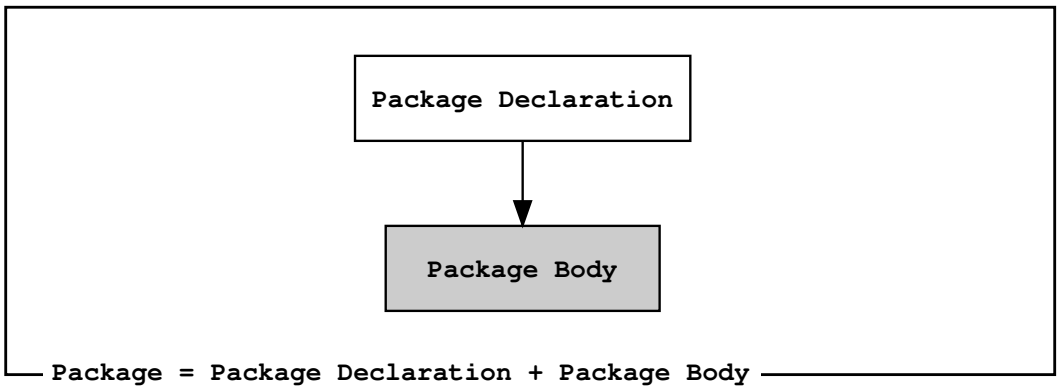

Abbildung 2.2: Aufbau eines VHDL-Pakets

Auffällig ist, dass im Gegensatz zum Bausteinentwurf die Deklaration und der Rumpf den gleichen Namen haben. Hierdurch ist festgelegt, dass es zu jeder Paketdeklaration nur einen Rumpf geben kann. Der einzige Unterschied zwischen Deklaration und Rumpf – bis auf das Schlüsselwort `body` – ist der Inhalt des Blocks *Deklarationen*. In einer Paketdeklaration dürfen unter anderem Typen, Konstanten, Signale, Unterprogramme und Komponenten deklariert werden. Der Paketrumpf hingegen enthält die Realisierungen der in der Paketdeklaration vereinbarten Unterprogramme. Um ein Beispiel für einen Paketrumpf geben zu können, müssen also zunächst Unterprogramme eingeführt werden, was in Abschnitt 2.6 geschieht.

Die in Beispiel 2.1.1 gezeigte Paketdeklaration enthält die Vereinbarung von zwei Datentypen, die dem Beispiel zugrunde liegen. Der Datentyp „`farben`“ wird für die Farben der Ampeln benutzt, und „`zustaende`“ enthält die Menge der Zustände, die der verwendete endliche Automat annehmen kann. Man erkennt, dass hier auf einer recht abstrakten Ebene modelliert wird. Es ist noch nicht zu sehen, wie der Zustand des Automaten oder die Farbe einer Ampel später einmal kodiert wird.

2.3 VHDL-Bibliotheken

Das nächste noch unbekannte Element in Beispiel 2.1.1 ist die Anweisung

```
use work.ampelpaket.all;
```

Bevor die Einzelheiten dieser Anweisung erläutert werden können, muss zunächst das Konzept der VHDL-Bibliotheken eingeführt werden. VHDL sieht vor, dass mehrere Entwurfseinheiten zu einer Bibliothek zusammengefasst werden. Außerdem kann es natürlich auch mehrere Bibliotheken geben. Innerhalb einer Bibliothek kann auf Elemente einer anderen Bibliothek zugegriffen werden.

2.3.1 Entwurfseinheiten

Unter einer Entwurfseinheit („Design Unit“) versteht man – wie bereits in Abschnitt 1.3 erläutert – einen zusammenhängenden Block von Anweisungen, der vom Analyseprogramm einzeln bearbeitet werden kann. Das Analyseprogramm überprüft die Syntax einer Entwurfseinheit und fügt die Einheit in eine Bibliothek ein, falls die Syntax korrekt ist. In VHDL gibt es fünf verschiedene Arten von Entwurfseinheiten, die in zwei Gruppen eingeteilt und in Tabelle 2.1 aufgelistet werden.

- Primäre Entwurfseinheiten
  - Bausteindeklaration (siehe Abschnitt 1.4.4.1)
  - Paketdeklaration (siehe Abschnitt 2.2)
  - Konfigurationsdeklaration (siehe Abschnitt 2.8.6)
- Sekundäre Entwurfseinheiten
  - Architekturrumpf (siehe Abschnitt 1.4.4.3)
  - Paketrumpf (siehe Abschnitt 2.2)

Tabelle 2.1: VHDL-Entwurfseinheiten

Folgende Regeln und Einschränkungen zur Existenz und Analyse von Entwurfseinheiten innerhalb einer VHDL-Bibliothek müssen beachtet werden:

- In einer VHDL-Bibliothek muss jede primäre Entwurfseinheit einen eindeutigen Namen haben. D. h., es darf keine zwei primären Entwurfseinheiten mit demselben Namen geben.
- Es darf mehrere sekundäre Entwurfseinheiten mit demselben Namen geben. Insbesondere darf eine sekundäre Entwurfseinheit auch einen bereits für eine primäre Einheit verwendeten Namen annehmen.
- Zusammengehörige primäre und sekundäre Entwurfseinheiten müssen in derselben Bibliothek stehen – dürfen also nicht getrennt werden.
- Wenn eine Entwurfseinheit auch einzeln analysierbar ist, so müssen doch gewisse Reihenfolgeregeln eingehalten werden:
  - Eine sekundäre Entwurfseinheit kann erst dann analysiert werden, wenn die zugehörige primäre Entwurfseinheit erfolgreich analysiert wurde. (Dies wird

meist dadurch eingehalten, dass z. B. eine Bausteindeklaration im Programmtext vor den entsprechenden Architekturrümpfen steht.

– Eine beliebige Entwurfseinheit, die auf eine andere primäre Entwurfseinheit zugreift (z. B. bei Verwendung von Datentypen, die in einer Paketdeklaration vereinbart wurden), kann erst dann analysiert werden, wenn die entsprechende primäre Entwurfseinheit erfolgreich analysiert wurde.

2.3.2 Sichtbarkeit von Entwurfseinheiten

Eine Entwurfseinheit und die darin enthaltenen Vereinbarungen sind für andere Entwurfseinheiten nicht automatisch sichtbar. In Beispiel 2.1.1 gibt es das Paket „`ampelpaket`“. Der Bausteindeklaration „`ampelschaltung`“ muss aber zunächst mitgeteilt werden, dass die Vereinbarungen aus diesem Paket Gültigkeit haben sollen. Dies wird mit Hilfe der `use`-Anweisung gemacht. Eine `use`-Anweisung gilt immer nur für die folgende Entwurfseinheit – also nicht für den gesamten folgenden Programmtext – und kann eine der folgenden Formen annehmen[1]:

```
use bibname.unitname;
use unitname.objekt;
use bibname.unitname.objekt;
```

„`bibname`“ ist der Name einer beliebigen VHDL-Bibliothek. Ein besonderer Bibliotheksname ist „`work`“: Er bezeichnet immer die Bibliothek, in die das Analyseprogramm die erfolgreich analysierten Einheiten ablegt. „`work`“ bezeichnet also die aktuell benutzte VHDL-Bibliothek und kann eingesetzt werden, um VHDL-Programme portabel zu halten. Auch wenn die Bibliothek, in der diese Anweisung vorkommt, umbenannt wird, stellt der Bezeichner „`work`“ immer noch einen Selbstbezug her.

Ebenso ist „`all`“ ein spezieller Objektbezeichner, der alle Vereinbarungen innerhalb der entsprechenden Funktionseinheit anspricht. Anstelle der in Beispiel 2.1.1 verwendeten `use`-Anweisung hätte man also auch die folgenden Anweisungen verwenden können:

Beispiel 2.3.1

```
use work.ampelpaket;          -- an dieser Stelle „weiß" das
                              -- Analyseprogramm, dass „ampelpaket"
                              -- eine primäre Entwurfseinheit ist
use ampelpaket.farben;
use ampelpaket.zustaende;     -- anstelle der Verwendung von „all"
                              -- kann man auch jede benötigte
                              -- Vereinbarung einzeln ansprechen
```

Folgende Ausnahmen zu den Sichtbarkeitsregeln erleichtern den Umgang mit der `use`-Anweisung:

– Für eine sekundäre Entwurfseinheit ist die zugehörige primäre Entwurfseinheit immer automatisch sichtbar. (Deshalb enthält Beispiel 2.1.1 keine Anweisung „`use work.ampelschaltung`“ vor dem Architekturrumpf „`eins`“.)

[1] Es gibt weitere Formen der `use`-Anweisung, die im Zusammenhang mit Konfigurationsdeklarationen verwendet werden können und deshalb hier nicht berücksichtigt werden.

– Da in einer sekundären Entwurfseinheit alle Vereinbarungen der zugehörigen primären Entwurfseinheit automatisch gelten, brauchen die `use`-Anweisungen hier nicht wiederholt zu werden. (Deshalb enthält Beispiel 2.1.1 keine Anweisung „`use work.ampelpaket.all`“ vor dem Architekturrumpf „`eins`“.)

Abbildung 2.3 veranschaulicht den Inhalt der aktuellen VHDL-Bibliothek nach der vollständigen Analyse des Beispiels 2.1.1 und die darin vereinbarten Sichtbarkeiten (gestrichelt gezeichnete Entwurfseinheiten werden später in diesem Kapitel hinzugefügt).

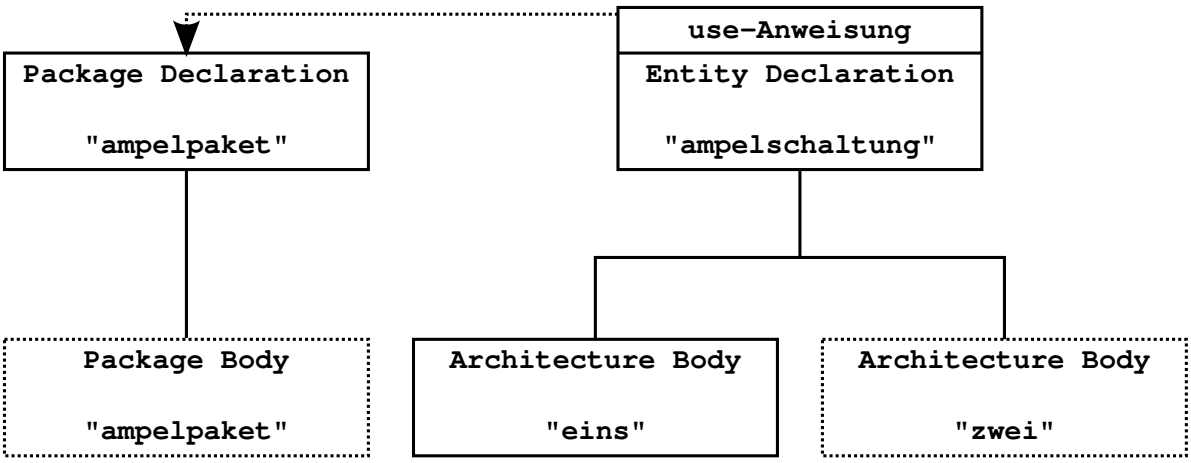

Abbildung 2.3: Inhalt einer VHDL-Bibliothek

2.3.3 Sichtbarkeit von Bibliotheken

Genau wie primäre Entwurfseinheiten sind auch Bibliotheken nicht automatisch sichtbar. Angenommen, die Paketdeklaration „`ampelpaket`“ befindet sich in einer getrennten Bibliothek namens „`paketbib`“, so hätte dem Analyseprogramm zunächst mitgeteilt werden müssen, dass „`paketbib`“ der Name einer VHDL-Bibliothek ist. Dies ist mit der `library`-Anweisung möglich. Im angenommenen Fall würden die Anweisungen

```
library paketbib;
use paketbib.ampelpaket.all;
```

zum gewünschten Ziel führen. Diese Konfiguration lässt sich durch Abbildung 2.4 veranschaulichen.

Auch wenn anstelle des speziellen Bibliotheksbezeichners „`work`“ in Beispiel 2.1.1 der tatsächliche Bezeichner der aktuellen Bibliothek (z. B. „`ampelbib`“) verwendet worden wäre, hätte dieser zunächst mit einer `library`-Anweisung vereinbart werden müssen:

```
library ampelbib;
use ampelbib.ampelpaket.all;
```

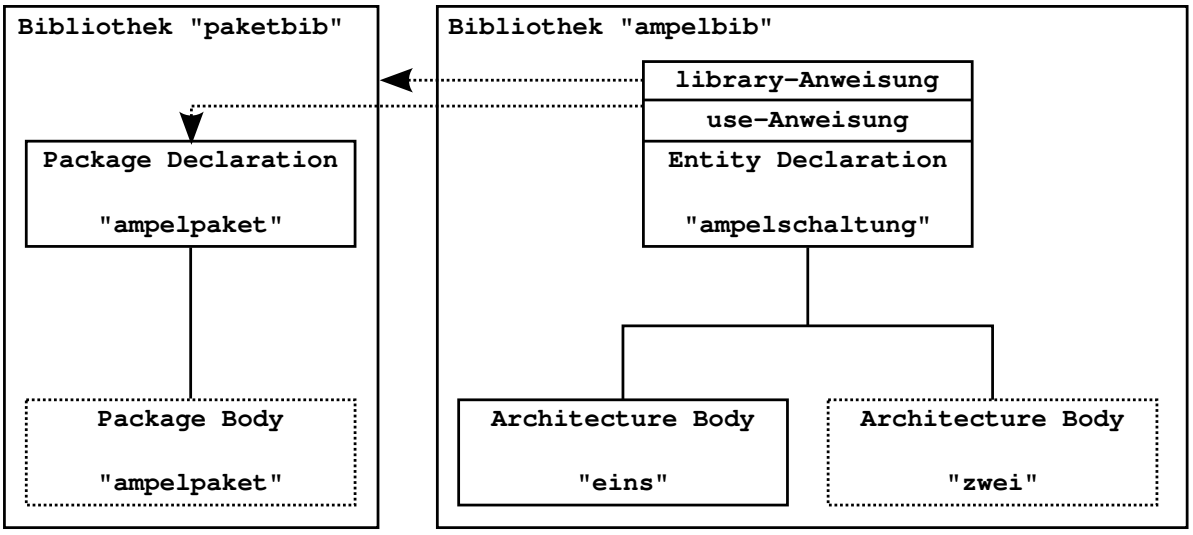

Abbildung 2.4: Verwendung von Elementen anderer Bibliotheken

Der Bezeichner „`work`“ ist nur deshalb immer verwendbar, weil implizit vor jeder Entwurfseinheit die Anweisung

```
library work;
```

angenommen wird. Das Gleiche gilt für die vordefinierten Datentypen und die vordefinierte Funktion „`now`“ („`now`“ gibt den aktuellen Wert der Simulationszeit zurück). Diese werden in der standardmäßig vorhandenen Bibliothek „`std`“ im Paket „`standard`“ definiert. Implizit stehen vor jeder Entwurfseinheit zusätzlich die Anweisungen

```
library std;
use std.standard.all;
```

2.4 Implizite Prozesse

Wenn man das Beispiel 2.1.1 weiter betrachtet, dann folgen nach der `use`-Anweisung zunächst eine Reihe von Elementen, die bereits in Kapitel 1 erläutert wurden. Die Bausteindeklaration „`ampelschaltung`“ enthält nur bekannte Elemente und auch der Architekturrumpf „`eins`“ sollte bis einschließlich zum Prozessblock mit der Marke „Kontrolle“ nichts Neues enthalten. Anschließend folgt eine `with`-Anweisung mit der Marke „`Hauptstrassenampel`“. Diese Anweisung wurde bisher nicht behandelt und scheint auch zunächst nicht in einen Architekturrumpf zu passen, da in Abschnitt 1.4.4.3 erläutert wurde, dass ein Architekturrumpf aus einer Liste von Prozessen und/oder Komponenteninstanziierungen besteht. Im Prinzip ist eine `with`-Anweisung jedoch auch ein Prozess. Sie ist ein Vertreter der sogenannten nebenläufigen Anweisungen. Diese sind nichts anderes als Kurzschreibweisen für bestimmte, einfache Prozessblöcke. D. h., zu jeder nebenläufigen Anweisung kann man eine `process`-Anweisung derart konstruieren, dass beide Anweisungen äquivalent sind. Nebenläufige Anweisungen dienen dazu, VHDL-Programme kürzer und besser lesbar zu machen.

2.4.1 Nebenläufige Signalzuweisungen

Die einfachste nebenläufige Anweisung ist die nebenläufige Signalzuweisung. Sie kommt in Beispiel 2.1.1 nicht vor, wurde aber bereits in Beispiel 1.2.2 benutzt. Signalzuweisungen, die nicht innerhalb von Prozessen stehen, sind nebenläufige Signalzuweisungen. Die Anweisung

```
s <= a xor b after 5 Ns;
```

ist äquivalent zu der folgenden `process`-Anweisung:

```
process begin
  s <= a xor b after 5 Ns;
  wait on a, b;
end process;
```

Der äquivalente Prozess ist immer sensibel für alle Signale, die auf der rechten Seite der Zuweisung vorkommen.

2.4.2 Bedingte Signalzuweisungen

Eine zweite Kurzform ist die bedingte Signalzuweisung. Sie entspricht einer `if-then-else`Anweisung innerhalb eines Prozesses und hat die folgende Form:

```
signalname <= Option
  Ereignis1 when Bedingung1 else
    …
  EreignisN-1 when BedingungN-1 else
  EreignisN;
```

Als *Option* darf „`transport`“ angegeben werden.[2] Die `transport`-Option beeinflusst die Wirkung einer Zuweisungsanweisung und wird in Abschnitt 2.5 erläutert. Der zur oben gezeigten bedingten Signalzuweisung äquivalente Prozessblock sieht wie folgt aus:

```
process (Sensibilitätsliste) begin
  if Bedingung1 then
    signalname <= Option Ereignis1;
      …
  elsif BedingungN-1 then
    signalname <= Option EreignisN-1;
  else
    signalname <= Option EreignisN;
  end if;
end process;
```

[2] Außerdem darf als *Option* auch „`guarded`“ angegeben werden. Diese Option wird in diesem Tutorial jedoch nicht behandelt.

2.4.3 Selektive Signalzuweisungen

Die letzte nebenläufige Signalzuweisung ist die in Beispiel 2.1.1 verwendete `with`-Anweisung. Auch hierzu kann man eine äquivalente `process`-Anweisung angeben, die im Fall der mit der Marke „`Hauptstrassenampel`“ versehenen `with`-Anweisung wie folgt aussieht:

```
process (zustand) begin
  case zustand is
    when hauptstr_gruen => hauptstr_ampel <= gruen;
    when hauptstr_gelb => hauptstr_ampel <= gelb;
    when nebenstr_gruen | nebenstr_gelb => hauptstr_ampel <= rot;
  end case;
end process;
```

2.5 Inertial Delay und Transport Delay

Das letzte bisher nicht erläuterte Konzept, das in Beispiel 2.1.1 benutzt wird, tritt in den Signalzuweisungen im Taktgeberprozess zu Tage. Einem Signal kann nicht nur ein Wert, sondern gleich eine ganze Folge von Werten mit verschiedenen Verzögerungen zugewiesen werden:

```
signalname <= Ausdruck after Dauer,
              Ausdruck after Dauer,
                …
              Ausdruck after Dauer;
```

Eine derartige Anweisung hat zur Folge, dass für das Zielsignal gleich eine ganze Liste von Ereignissen erzeugt wird. Solche zu einem einzelnen Signal gehörenden Ereignislisten nennt man Signaltreiber oder einfach Treiber. Ein Element eines Treibers besteht aus einem Paar (Signalwert, Zeitpunkt), wobei der Zeitpunkt immer relativ zum aktuellen Simulationszeitpunkt angegeben wird. Zu beachten ist, dass die anstelle von *Dauer* angegebenen Verzögerungen aufsteigend sortiert sein müssen.

Unter der Voraussetzung, dass der Treiber für das Signal „`s`“ vorher leer war, erzeugt die Anweisung

Beispiel 2.5.1

```
s <= 1 after 10 Ns, 2 after 20 Ns;
```

den folgenden Treiber:

s ⟵ (1,10Ns)(2,20Ns)

Die Anweisungsfolge

Beispiel 2.5.2

```
s <= 1 after 10 Ns;
s <= 2 after 20 Ns;
```

erzeugt dagegen nur ein Element im Treiber, da bei der zweiten Zuweisung das erste Element wieder gelöscht wird.

```
s <= 1 after 10 Ns;   s ⟵ (1,10Ns)
s <= 2 after 20 Ns;   s ⟵ (2,20Ns)
```

Dies hängt mit den unterschiedlichen Delay-Typen zusammen, die VHDL zur Verfügung stellt. Die Standardform, die auch bei den beiden zuletzt gezeigten Signalzuweisungen zur Anwendung kommt, heißt „inertial delay“. Außerdem gibt es die Variante „transport delay“, die dadurch gekennzeichnet wird, dass dem Zuweisungssymbol „`<=`“ das Schlüsselwort „`transport`“ folgt. Da das transport delay vom Konzept her etwas einfacher ist, soll es im Folgenden zuerst behandelt werden.

2.5.1 Transport Delay

Die Verwendung des transport delay führt zu einem Verhalten des modellierten Signals, das einer typischen Leitungsverzögerung entspricht. Das Konzept sieht vor, dass ein neues Ereignis, das später als alle anderen bisher im Treiber enthaltenen Ereignisse auftritt, einfach an den vorhandenen Treiber angehängt wird. Wird hingegen ein neues Ereignis erzeugt, das vor anderen, bereits vorhandenen Ereignissen stattfinden wird, dann werden die nachfolgenden Ereignisse aus dem Treiber gelöscht. Das zuletzt gezeigte Beispiel führt also bei Verwendung des transport delay zu einem anderen Ergebnis:

Beispiel 2.5.3

```
s <= transport 1 after 10 Ns;   s ⟵ (1,10Ns)
s <= transport 2 after 20 Ns;   s ⟵ (1,10Ns)(2,20Ns)
```

Werden die beiden Signalzuweisungen vertauscht, so hat der Treiber nach Ausführung der Anweisungen nur ein Element:

Beispiel 2.5.4

```
s <= transport 2 after 20 Ns;   s ⟵ (2,20Ns)
s <= transport 1 after 10 Ns;   s ⟵ (1,10Ns)
```

Ein letztes Beispiel zum transport delay zeigt die Anwendung beider Regeln gleichzeitig:

Beispiel 2.5.5

```
s <= transport 1 after 10 Ns, 3 after 30 Ns, 5 after 50 Ns;
                            s ⟵ (1,10Ns)(3,30Ns)(5,50Ns)
s <= transport 4 after 40 Ns;
                            s ⟵ (1,10Ns)(3,30Ns)(4,40Ns)
```

2.5.2 Inertial Delay

Mit Hilfe des inertial delay kann man z. B. das Verhalten von Flipflops modellieren, die das Auftreten von kurzzeitigen Störungen auf einer Eingangsleitung ignorieren. Man kann also Bauteile modellieren, deren Eingangsleitungen eine gewisse Zeit lang stabil sein müssen, bevor der Wert auf der Leitung übernommen wird.

In VHDL gibt es kein Schlüsselwort „`inertial`“. Das inertial delay wird immer dann angewendet, wenn das Schlüsselwort `transport` nicht verwendet wurde.

Bei Verwendung des inertial delay wird mit im Treiber bereits vorhandenen Ereignissen, die auf ein neues Ereignis folgen würden, ebenso verfahren wie beim transport delay: sie werden gelöscht. Der Unterschied liegt bei den Ereignissen, die bereits im Treiber enthalten sind und zu einem früheren Zeitpunkt stattfinden würden als das neu einzufügende Ereignis. Sie werden mit einer Ausnahme ebenfalls gelöscht. Ausgenommen sind die Ereignisse, die den gleichen Wert an das Signal zuweisen wie das neue Ereignis.

Ein erstes Beispiel wurde bereits in Beispiel 2.5.2 gegeben. Es verdeutlicht den grundlegenden Unterschied zwischen inertial und transport delay. Das Beispiel 2.5.4 würde mit inertial delay – also bei Streichung der Schlüsselworte „`transport`“ – das gleiche Ergebnis liefern.

Beispiel 2.5.6 zeigt die Anwendung der Ausnahmeregel bei der Streichung von früheren Ereignissen.

Beispiel 2.5.6

```
s <= transport 1 after 10 Ns, 3 after 30 Ns, 5 after 50 Ns;
```
s ⟵ (1,10Ns)(3,30Ns)(5,50Ns)

```
s <= transport 3 after 40 Ns;
```
s ⟵ (3,30Ns)(3,40Ns)

In diesem Fall wird das 10-Ns-Element gestrichen, und das 30-Ns-Element wird nicht gestrichen, weil es den gleichen Wert zuweist wie das neu einzufügende Ereignis. Das 50-Ns-Element wird gestrichen, weil es später stattfinden würde als das 40-Ns-Element.

Eine weitere Ausnahmeregel besagt, dass das zweite und alle folgenden Elemente einer Signalzuweisung mit mehreren Ereignissen immer automatisch transport-delay-Ereignisse sind. Dies ist sinnvoll, da ansonsten Zuweisungen mit mehreren Ereignissen wenig Sinn hätten, weil die meisten Ereignisse gleich wieder gelöscht würden. Mit dieser Regel wird auch klar, warum in Beispiel 2.5.1 beide Ereignisse in den Treiber gelangen. Ohne die Regel hätte es keinen Unterschied zwischen den Beispielen 2.5.1 und 2.5.2 gegeben.

Die Signalzuweisung

```
s <= 1 after 10 Ns, 3 after 30 Ns, 5 after 50 Ns;
```

ist also äquivalent zu der Anweisungsfolge

```
s <= 1 after 10 Ns;
s <= transport 3 after 30 Ns;
s <= transport 5 after 50 Ns;
```

2.6 Funktionen und Prozeduren

Ein wichtiges und mächtiges Konzept von VHDL ist die Möglichkeit der Definition und Verwendung von Funktionen und Prozeduren. Derartige Unterprogramme können ähnlich wie in C oder Pascal vereinbart und an anderer Stelle im Programmtext aufgerufen werden. In VHDL müssen allerdings einige Besonderheiten beachtet werden, die mit dem Konzept der nebenläufig auszuführenden Prozesse zusammenhängen.

Beispiel 2.1.1 enthält bisher keine Funktionen oder Prozeduren. Eine Möglichkeit, die Ampelschaltung mit Hilfe von Funktionen zu beschreiben, wird im Folgenden gezeigt. Die verwendeten Funktionen sollen im Paket „`ampelpaket`“ vereinbart werden. Dann

wird die Paketdeklaration um die Vereinbarung der Funktionen „hauptstr_farbe“ und „nebenstr_farbe“ erweitert, und es wird ein Paketrumpf mit den Realisierungen der beiden Funktionen erzeugt.

Beispiel 2.6.1

```
package ampelpaket is
  type farben is (gruen, gelb, rot, unbekannt);
  type zustaende is (
      hauptstr_gruen,
      hauptstr_gelb,
      nebenstr_gruen,
      nebenstr_gelb);
  function hauptstr_farbe(z: in zustaende) return farben;
  function nebenstr_farbe(z: in zustaende) return farben;
end ampelpaket;


package body ampelpaket is
  function hauptstr_farbe(z: in zustaende) return farben is
  begin
    case z is
      when hauptstr_gruen => return gruen;
      when hauptstr_gelb => return gelb;
      when nebenstr_gruen | nebenstr_gelb => return rot;
    end case;
  end hauptstr_farbe;
  function nebenstr_farbe(z: in zustaende) return farben is
  begin
    case z is
      when nebenstr_gruen => return gruen;
      when nebenstr_gelb => return gelb;
      when hauptstr_gruen | hauptstr_gelb => return rot;
    end case;
  end nebenstr_farbe;
end ampelpaket;
```

Anschließend können die Prozesse „Hauptstrassenampel“ und „Nebenstrassenampel“ wie im Folgenden gezeigt verändert werden:

Beispiel 2.6.2

```
  Hauptstrassenampel:
  hauptstr_ampel <= hauptstr_farbe(zustand);


  Nebenstrassenampel:
  nebenstr_ampel <= nebenstr_farbe(zustand);
```

Jeder der beiden Prozesse besteht jetzt nur noch aus einer einfachen nebenläufigen Signalzuweisung, die auf der rechten Seite den Aufruf der entsprechenden Funktion enthält.

Dies veranschaulicht auch schon den wesentlichen Unterschied zwischen Funktionen und Prozeduren. Der Aufruf einer Funktion liefert wie gezeigt einen Ausdruck, während der Aufruf einer Prozedur bereits eine vollständige Anweisung darstellt. Die geänderte Version des Architekturrumpfes soll als Architektur „`zwei`“ abgelegt werden.

Analog zu Bausteindeklaration und Architekturrumpf wird auch bei Unterprogrammen zwischen einer Unterprogrammdeklaration und einem Unterprogrammrumpf unterschieden. Eine Unterprogrammdeklaration hat eine der beiden folgenden Formen:

```
procedure procname (Schnittstellenliste);
function funcname (Schnittstellenliste) return Datentyp;
```

Die Schnittstellenliste hat dabei die gleiche Syntax wie in der `port`- und `generic`-Anweisung innerhalb einer Bausteindeklaration (siehe Abschnitt 1.4.4.2). Es sind allerdings folgende Einschränkungen hinsichtlich der Schnittstellenelemente zu beachten:

- Alle Parameter einer Funktion müssen die Objektart „`signal`“ oder „`constant`“ und den Schnittstellenmodus „`in`“ haben.
- Die Parameter einer Prozedur dürfen die Objektarten „`signal`“, „`variable`“ oder „`constant`“ und die Schnittstellenmodi „`in`“, „`out`“ oder „`inout`“ annehmen (Schnittstellenelemente der Objektart „`constant`“ können natürlich nur den Modus „`in`“ haben).

Ein Unterprogrammrumpf hat die folgende Form:

```
Unterprogrammspezifikation is
  Deklarationen
begin
  Anweisungen
end unterprogrammname;
```

Unterprogrammspezifikation ist hierbei eine Unterprogrammdeklaration ohne das abschließende Semikolon. Im Fall von Funktionen muss im Anweisungsteil mindestens eine `return`-Anweisung vorhanden sein. In einem Unterprogramm dürfen keine Hardwarestrukturen instanziiert werden. D. h. es dürfen keine Komponenten vereinbart oder instanziiert werden, es dürfen keine neuen Signale vereinbart werden und es dürfen keine Prozesse erzeugt werden. Es ist erlaubt, aus einem Unterprogramm andere Unterprogramme oder rekursiv dasselbe Unterprogramm aufzurufen.

2.7 Aufbau einer Testumgebung

In Abschnitt 1.3.1 wurde gezeigt, wie man das Verhalten eines mit VHDL beschriebenen Bausteins mit Hilfe eines Simulators simulieren lassen kann. Die Werte für die Eingangssignale wurden dabei von Hand eingegeben. Dies ist bei kleineren Beispielen sicherlich sinnvoll, wird aber mit größer werdenden Beschreibungen immer aufwändiger und fehleranfälliger. Bei komplexen Systemen ist diese Vorgehensweise nicht mehr praktizierbar.

Es existieren verschiedene Auswege aus dieser Misere. Die erste Möglichkeit ist eine Option, die die Mehrzahl der existierenden Simulatoren bietet. Man kann die Befehle, die benötigt werden, um die Simulation zu steuern, in einer Datei ablegen. Anschließend gibt man dem Simulator die Anweisung, diese Datei zu lesen und die Befehle auszuführen.

Diese Möglichkeit bietet eine erste Vereinfachung, da man die Befehle zwar immer noch von Hand eingeben muss, die Befehlsfolge dann aber mehrfach verwenden kann.

Eine zweite Lösung des Problems bieten die sogenannten Stimulisprachen. Dies sind Sprachen, mit denen man Eingabefolgen an ein System mit einfachen Mitteln beschreiben kann. Die Stimulisprachen sind den Hardware-Beschreibungssprachen meist sehr ähnlich. Damit wird der Nachteil, dass eine zusätzliche Sprache erlernt werden muss, gelindert.

Wenn man das Konzept der Stimulisprachen konsequent weiterverfolgt, dann bietet es sich an, die Hardware-Beschreibungssprache selbst zur Erzeugung der Eingaben an das zu simulierende System zu verwenden. An die Stelle der Konstruktion von sinnvollen Eingabefolgen an das System – einer Aufgabe, die oft sehr mühsam ist – tritt die Modellierung einer möglichen Umgebung für das System. Diese Umgebung wird natürlich auf einer sehr abstrakten Ebene modelliert, da in diesem Fall nur die Kommunikation mit dem Zielsystem interessiert. Das Zielsystem wird dann in die Testumgebung eingebettet und das Zusammenwirken des Gesamtsystems kann beobachtet werden.

Eine derartige Testumgebung soll für das Beispiel 2.1.1 aufgebaut werden. In diesem Fall wäre natürlich auch die Zuweisung der Eingabewerte von Hand problemlos. Da das Beispiel klein ist und nur ein Eingabesignal existiert, kann auch die Testumgebung sehr einfach gehalten werden. Sie besteht lediglich aus einem „Anforderungsgenerator“ und einer übergeordneten Einheit, die den Generator und die Ampelschaltung miteinander verbindet.

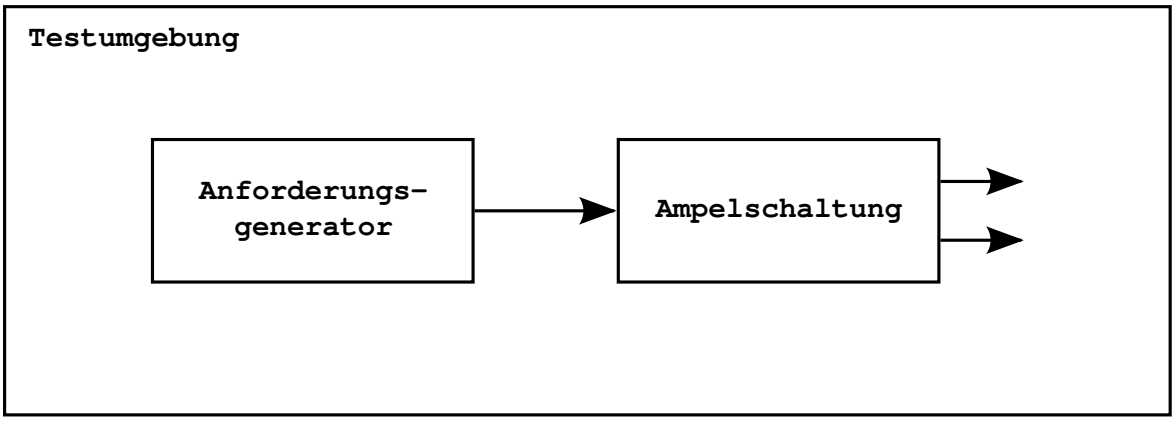

Abbildung 2.5: Aufbau einer Testumgebung

Die VHDL-Beschreibungen sehen wie folgt aus:

Beispiel 2.7.1

```
entity anforderungsgenerator is
  port (anforderung: out Boolean);
end anforderungsgenerator;


architecture realisierung of anforderungsgenerator is
begin
  process begin
    anforderung <= True,
                   False after 5 Sec,
                   True after 60 Sec,
                   False after 160 Sec,
                   True after 170 Sec,
                   False after 200 Sec;
  end process;
end realisierung;


entity testumgebung is
end testumgebung;


use work.ampelpaket.all;
use work.all;
architecture struktur of testumgebung is
  component anforderungsgenerator
    port (anforderung: out Boolean);
  end component;
  component ampelschaltung
    generic (timeout: Time;
             gelbdauer: Time);
    port (   anforderung: in Boolean;
             hauptstr_ampel: out farben;
             nebenstr_ampel: out farben);
  end component;
  signal eingabe: Boolean;
begin
  baustein1: anforderungsgenerator
    port map (eingabe);
  baustein2: ampelschaltung
    generic map (50 Sec, 5 Sec)
    port map (eingabe, open, open);
end struktur;
```

Wenn man eine Simulation dieser Testumgebung startet, dann muss man keine Signale per Hand zuweisen. Alle Zuweisungen sind bereits in der Beschreibung enthalten. Die

Simulation wird nach 255 Sekunden anhalten, da zu diesem Zeitpunkt keine weiteren Ereignisse mehr vorhanden sind.

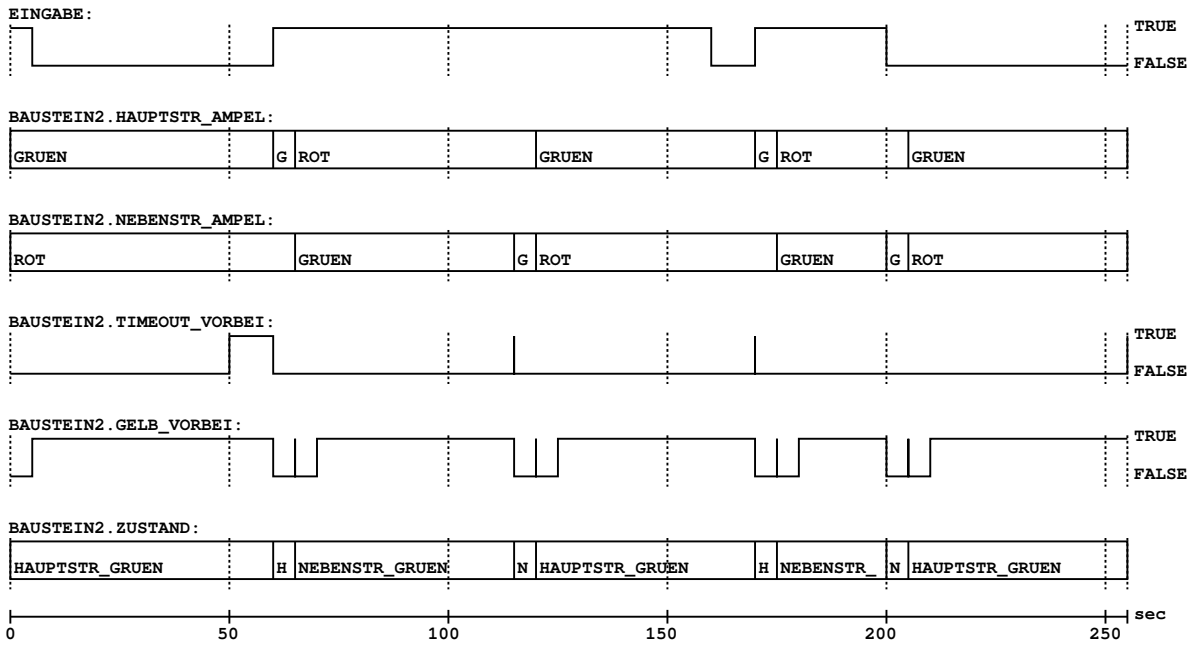

Abbildung 2.6: Simulationsergebnisse

Am Ablaufdiagramm in Abbildung 2.6 erkennt man, dass der Zustandszyklus während der Simulation zweimal durchlaufen wurde. Die erste Grünanforderung auf der Nebenstraße (Signal „`eingabe`“ auf „`True`“) wird verschluckt, da zu dieser Zeit das Timeout für die Hauptstraße keine Umschaltung zulässt. Auf die nächsten beiden Anforderungen kann sofort reagiert werden. Das erste Mal wird wieder zur Hauptstraße zurück geschaltet, nachdem das entsprechende Timeout abgelaufen ist, und beim zweiten Mal wird zurück geschaltet, weil keine Anforderung mehr vorliegt.

Man erkennt an den Simulationsergebnissen, dass das Modell noch etwas realitätsfern ist. Die Eingabeleitung „`anforderung`“ muss angeben, ob jemand auf der Nebenstraße wartet. Eine Induktionsschleife kann jedoch nur in dem Moment einen Impuls geben, in dem ein Fahrzeug die Schleife überquert. Es bleibt dem Leser überlassen, die Schaltung um einen zusätzlichen Baustein zu erweitern, der eine Impulsleitung liest und das Signal „`anforderung`“ generiert.

2.8 Weitere VHDL-Konstrukte

Die Hardware-Beschreibungssprache VHDL ist so umfangreich, dass im Rahmen dieses Tutorials nicht alle Elemente der Sprache behandelt werden können. Im Rest dieses Kapitels werden einige weitere Konstrukte kurz angesprochen, die bisher noch nicht erläutert wurden. Die folgenden Ausführungen sind aber knapp gehalten und erheben keinen Anspruch auf Vollständigkeit.

2.8.1 Der Alias-Mechanismus

VHDL stellt einen Mechanismus zur Verfügung, der es ermöglicht, Teile eines Datenobjekts mit verschiedenen Namen anzusprechen. Dies ist z. B. dann sinnvoll, wenn man häufig die verschiedenen Teile eines Befehlswortes einzeln referenzieren muss.

Sei z. B. ein Instruktionsregister „`ir`“ als `signal` wie folgt deklariert

```
signal ir: Bit_Vector 15 downto 0;
```

und ein Befehlswort gemäß Abbildung 2.7 unterteilt.

Bits:	15 .. 13	12 11	10....0
Bedeutung:	Opcode	Registernummer	Distanz

Abbildung 2.7: Befehlsformat

Dann kann man die einzelnen Teile des Instruktionsregisters mit den Bezeichnern „`ir_opcode`“, „`ir_reg`“ und „`ir_dist`“ ansprechen, wenn im entsprechenden Deklarationsteil die folgenden Anweisungen stehen:

```
alias ir_opcode: Bit_Vector (3 downto 0) is ir (15 downto 13);
alias ir_reg:    Bit_Vector (1 downto 0) is ir (12 downto 11);
alias ir_dist:   Bit_Vector (10 downto 0) is ir (10 downto 0);
```

2.8.2 Attribute

Manchen Elementen einer VHDL-Beschreibung können sogenannte Attribute zugeordnet werden. Attributierbare Elemente sind:

Typen und Subtypen,

Prozeduren und Funktionen,

Signale, Variable und Konstanten,

Entities, Architekturen, Konfigurationen und Pakete,

Komponenten,

Anweisungsmarke.

Attribute beschreiben Eigenschaften des zugehörigen Elements und sind entweder vordefiniert oder vom Benutzer definiert. Ein Element kann beliebig viele Attribute haben. Die Schreibweise für Attribute ist:

```
elementname'attributname
```

Benutzerdefinierte Attribute werden mit der `attribute`-Anweisung definiert und mit Werten versehen:

```
attribute Cost: Integer;
attribute Cost of alu:entity is 22;
```

Einige wichtige vordefinierte Attribute sind (den folgenden Beispielen liegen die Datentypvereinbarungen

```
type bsp_up is range 0 to 10;
type bsp_down is range 100 downto 1;
type nummern is (eins, zwei, drei, vier);
```

zugrunde):

- Attribute von Typen und Subtypen:

  `T'Left`: Dieses Attribut enthält die linke Grenze eines skalaren Datentyps (`bsp_up'Left` = 0; `bsp_down'Left` = 100).

  `T'Right`: Dieses Attribut enthält die rechte Grenze eines skalaren Datentyps (`bsp_up'Right` = 10; `bsp_down'Right` = 1).

  `T'High`: Dieses Attribut enthält die obere Grenze eines skalaren Datentyps (`bsp_up'High` = 10; `bsp_down'High` = 100).

  `T'Low`: Dieses Attribut enthält die untere Grenze eines skalaren Datentyps (`bsp_up'Low` = 0; `bsp_down'Low` = 1).

  `T'Pos(e)`: Dieses Attribut enthält die Position eines anzugebenden Elementes innerhalb eines diskreten[3] oder physikalischen Datentyps (`nummern'Pos(eins)` = 0; `bsp_down'Pos(100)` = 100).

  `T'Val(i)`: Dieses Attribut ist die Umkehrfunktion zum Attribut „`Pos`“ (`nummern'Val(3)` = `vier`; `bsp_up'Val(0)` = 0).

- Attribute von Feldern:

  `F'Left(i)`: Dieses Attribut enthält die linke Schranke der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

  `F'Right(i)`: Dieses Attribut enthält die rechte Schranke der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

  `F'High(i)`: Dieses Attribut enthält die obere Schranke der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

  `F'Low(i)`: Dieses Attribut enthält die untere Schranke der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

  `F'Range(i)`: Dieses Attribut enthält das Intervall der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1). Mit Hilfe dieses Attributs kann man sehr einfach Schleifen über das Feld programmieren:

  ```
  for i in nummern'Range loop …
  ```

  `F'Length(i)`: Dieses Attribut enthält die Anzahl der Elemente in der i-ten Dimension des Feldes (der Parameter ist optional und default ist 1).

Außerdem gibt es vordefinierte Attribute von Signalen, mit Hilfe derer man z. B. feststellen kann, wann das letzte Ereignis stattgefunden hat und ähnliches.

[3] Ein diskreter Datentyp ist ein Integer-Typ oder ein Aufzählungstyp.

2.8.3 Subtypen

Zusätzlich zur `type`-Anweisung gibt es noch die `subtype`-Anweisung. Mit Hilfe dieser Anweisung kann man Datentypen definieren, die nur einen Teilbereich des zugrunde liegenden Typs abdecken. Ein gutes Beispiel sind die vordefinierten Datentypen „`Positive`“ und „`Natural`“, die wie folgt definiert sind:

```
subtype Natural is Integer range 0 to Integer'High;
subtype Positive is Integer range 1 to Integer'High;
```

Der wesentliche Unterschied zwischen der Vereinbarung eines Datentyps mit der `subtype`- oder der `type`-Anweisung liegt darin, dass Datentypen, die mittels `subtype` vereinbart wurden, mit dem zugrunde liegenden Datentyp kompatibel sind. D. h., folgender VHDL-Programmausschnitt – die oben gegebenen Vereinbarungen vorausgesetzt – ist zulässig:

Beispiel 2.8.1

```
signal i: Integer := 42;
signal n: Natural := 0;
begin
  :
n <= i;
```

Wenn der Datentyp „`Natural`“ mit einer `type`-Anweisung deklariert worden wäre, dann hätte das Analyseprogramm bei der Signalzuweisung eine Fehlermeldung ausgegeben. Datentypen, die mit verschiedenen `type`-Anweisungen definiert wurden, sind grundsätzlich nicht miteinander kombinierbar. Anders ist dies mit Subtypen. Objekte, die sich im Subtyp unterscheiden, deren Basisdatentyp aber gleich ist, können miteinander kombiniert werden. Die Überprüfung, ob der zugewiesene Wert im zulässigen Bereich liegt, erfolgt in diesem Fall erst zur Laufzeit. Mit Hilfe von Subtypen kann der Entwerfer Design-Annahmen in das Modell einbringen, die zur Laufzeit automatisch überprüft werden sollen.

2.8.4 Signale mit mehreren Treibern

In Abschnitt 1.4.5.3.2 wurde darauf hingewiesen, dass einem Signal normalerweise nur von einem Prozess Werte zugewiesen werden dürfen. Eine besondere Situation liegt vor, wenn mehrere Prozesse Werte für ein Signal generieren. Dies kann man mit einem Hardware-Aufbau vergleichen, in dem die Ausgangsleitungen mehrerer Bausteine miteinander verbunden sind. Da die Ergebnisse derartiger Konstruktionen stark von der Technologie abhängig sind, ist in VHDL nicht von vornherein festgelegt, was in einem solchen Fall zu geschehen hat. Stattdessen wird gefordert, dass Signalen, die mehrere Treiber haben, eine Berechnungsfunktion zugeordnet wird. Dies geschieht bereits bei der Typvereinbarung des Datentyps, der derartigen Signalen zugeordnet wird.

```
subtype spezialsignal is wired_or Bit;
```

„`wired_or`“ ist die Berechnungsfunktion, die allen Objekten vom Typ „`spezialsignal`“ zugeordnet wird. Derartige Funktionen müssen zunächst geeignet definiert werden. Der Eingabeparameter an die Funktion muss ein eindimensionales, in der Länge flexibles Feld sein, damit beliebig viele Treiber an ein Signal angeschlossen werden können. Der

Typ der Feldelemente des Parameters sowie der Ergebnistyp der Berechnungsfunktion müssen mit dem Basisdatentyp des Signals übereinstimmen (im gezeigten Fall ist dies der Datentyp „`Bit`“). Die Berechnungsfunktion „`wired_or`“ müsste also wie folgt vereinbart werden:

```
function wired_or(eingang: Bit_Vector) return Bit;
```

2.8.5 Reguläre Strukturen

Hardware-Systeme weisen häufig reguläre Strukturen auf. Solche Strukturen können in VHDL mit der `generate`-Anweisung erzeugt werden. Dadurch werden VHDL-Beschreibungen kürzer und besser lesbar.

Eines der einfachsten Beispiele für eine reguläre Struktur ist ein Mehrfach-Inverter. Ein 16-fach-Inverter kann mit Hilfe der `generate`-Anweisung wie in Beispiel 2.8.2 gezeigt beschrieben werden:

Beispiel 2.8.2

```
entity invert16 is
  port (eingang: Bit_Vector(1 to 16);
        ausgang: out Bit_Vector(1 to 16));
end invert16;


architecture generiert of invert16 is
  component inverter
    port (eingang: Bit; ausgang: out Bit);
  end component;
begin
  G: for i in 1 to 16 generate
    inv: inverter port map (eingang(i), ausgang(i));
  end generate;
end generiert;
```

Sehr sinnvoll ist es, die `generate`-Anweisung mit `generic`-Parametern zu kombinieren. Damit kann man einen n-fach-Inverter konstruieren:

Beispiel 2.8.3

```
entity invertN is
  generic (N: Positive);
  port (eingang: Bit_Vector(1 to N);
        ausgang: out Bit_Vector(1 to N));
end invertN;


architecture generiert of invertN is
  component inverter
    port (eingang: Bit; ausgang: out Bit);
  end component;
begin
```

```
  G: for i in 1 to N generate
    inv: inverter port map (eingang(i), ausgang(i));
  end generate;
end generiert;
```

Neben der `for…generate`-Anweisung gibt es eine weitere Anweisung `if…generate`. Mit dieser Anweisung kann man Ausnahmefälle erfassen, die häufig für das erste und das letzte Element einer regulären Struktur gelten. Ein serieller n-Bit-Addierer ließe sich z. B. wie folgt beschreiben:

Beispiel 2.8.4

```
entity nbit_addierer is
  generic (N: Positive);
  port (a, b: Bit_Vector(1 to N);
        carry_in: Bit;
        summe: out Bit_Vector(1 to N);
        carry_out: out Bit);
end nbit_addierer;


architecture generiert of nbit_addierer is
  component volladdierer
    port (a, b, cin: Bit; summe, cout: out Bit);
  end component;
  signal c: Bit_Vector(1 to N-1);
begin
  G: for i in 1 to N generate
    G1: if i = 1 generate
      va1: volladdierer port map (a(i), b(i), carry_in, summe(i), c(i));
    end generate;
    Gi: if i > 1 and i < N generate
      vai: volladdierer port map (a(i), b(i), c(i-1), summe(i), c(i));
    end generate;
    GN: if i = N generate
      vaN: volladdierer port map (a(i), b(i), c(i-1), summe(i), carry_out);
    end generate;
  end generate;
end generiert;
```

2.8.6 Konfigurationen

Da VHDL mehrere Architekturrümpfe zu einer Bausteindeklaration zulässt, muss dem Anwender die Möglichkeit gegeben werden, das zu simulierende System zu konfigurieren. D. h., er muss festlegen können, welche Architektur benutzt werden soll und welcher Bausteinentwurf zur Realisierung einer Komponente herangezogen werden soll.

Standardmäßig wird immer der zuletzt analysierte Architekturrumpf verwendet. Zur Realisierung einer Instanz einer Komponente wird nachgesehen, ob eine Entwurfseinheit mit dem Komponentennamen sichtbar ist. Deshalb wurde in Beispiel 2.7.1 die Anweisung

```
use work.all;
```

benutzt, um für den Architekturrumpf „struktur“ die Entwurfseinheit „anforderungsgenerator“ und „ampelschaltung“ sichtbar zu machen. Damit konnten diese Einheiten an die Instanzen „baustein1“ und „baustein2“ gebunden werden.

Um vom Standard abweichende Konfigurationen vornehmen zu können, existieren weitere Formen der use-Anweisung. In Beispiel 2.7.1 hätte man mit den Anweisungen

```
for all:anforderungsgenerator use work.anforderungsgenerator;
for baustein2:ampelschaltung use work.ampelschaltung(zwei);
```

genau festlegen können, welche Entwurfseinheiten benutzt werden sollen. Auch hier ist dem Schlüsselwort „all“ eine spezielle Bedeutung derart zugeordnet, dass man damit alle Instanzen einer Komponente ansprechen kann. Außerdem kann man an der gleichen Stelle das Schlüsselwort „others“ verwenden, um die bisher noch nicht angesprochenen Instanzen zu referenzieren. Weiterhin kann man an die zuletzt gezeigten Anweisungen noch port-map- und generic-map-Anweisungen anhängen, um damit die Schnittstellenlisten der gegebenen Komponente mit den Schnittstellenlisten der angesprochenen Entwurfseinheiten abzustimmen.

Die Konfiguration muss jedoch nicht direkt innerhalb eines Architekturrumpfes festgeschrieben werden. Um die Konfiguration eines Systems flexibel zu halten, kann man diese auch zunächst offen lassen und dann in einer anderen Entwurfseinheit – einer Konfigurationsdeklaration – beschreiben. So kann man völlig verschiedene Konfigurationen zusammenschalten und simulieren lassen, ohne die zugrunde liegenden Entwurfseinheiten verändern zu müssen.

Eine Konfigurationsdeklaration hat die folgende Form:

```
configuration konfigname of bausteinname is
  Deklarationen
  Konfiguration
end konfigname;
```

Für Beispiel 2.7.1 ließe sich also die folgende Konfigurationsdeklaration erstellen:

Beispiel 2.8.5

```
configuration beispiel of testumgebung is
  for struktur
    for baustein1:anforderungsgenerator
      use entity ampelbib.anforderungsgenerator;
    end for;
    for baustein2:ampelschaltung
      use entity ampelbib.ampelschaltung(zwei);
    end for;
  end for;
end beispiel;
```

Damit sind die meisten in VHDL vorhandenen Sprachelemente zumindest kurz angesprochen worden. Nicht behandelt wurden u. a. solche Sprachelemente, die nicht in erster Linie der Modellierung von Hardware dienen. Dies sind z. B. Zeiger- und Dateitypen und die Zugriffsmöglichkeiten auf die entsprechenden Objekte.

VHDL ist eine sehr umfangreiche Sprache, mit der versucht wurde, alle Aspekte der Modellierung und Simulation von Hardware abzudecken. Deshalb ist es im gegebenen Rahmen nicht möglich, alle Elemente im Detail kennenzulernen. Stattdessen wurde versucht, einen Überblick zu geben und einen Eindruck von den Konzepten zu vermitteln, die bei der Definition der Sprache berücksichtigt wurden.

3 Übungen im Labor

Begleitend zur Vorlesung „Rechnerarchitektur“ werden im Labor die Hardwaredesign-Kenntnisse praktisch angewendet und vertieft. Dabei sollen auch die Unterschiede zwischen einer Hardwarebeschreibungssprache und der bekannten Softwareprogrammierung klar werden. Beide Implementierungsmöglichkeiten bieten bestimmte Vor- und Nachteile. Für viele Probleme, zum Beispiel in eingebetteten Systemen, bietet sich eine Hardware-Software-Co-Design-Lösung an. Dabei entsteht die Möglichkeit bestimmte (oft rechenintensive) Komponenten in Hardware auszulagern ohne dabei auf die Vorteile einer Softwareimplementierung verzichten zu müssen.

Innerhalb des Labors wird ein MIPS entwickelt. Es wird dabei Technik eingesetzt, die von der Altera Corporation hergestellt wird. Als Hardwareplattform wird das DE2-Board (Development and Education) von Altera verwendet. Herzstück dieses Boards ist ein FPGA-Chip (FPGA=Field Programmable Gate Array) namens „Cyclone II“. Dieser bietet ausreichend viele Logikeinheiten für die Versuche im Labor. Als Hardwarebeschreibungssprache wird VHDL (Very High Speed Integrated Circuit Hardware Description Language) verwendet; die Implementierungsplattform für VHDL ist das Altera-Tool „Quartus II“. Dieses Tool ermöglicht auch die direkte Programmierung des DE2-Boards über den sog. USB-Blaster.

Als grundlegende Einführungen sind die Tutorials von Altera empfehlenswert. Diese können von der Altera-Website heruntergeladen werden: `http://www.altera.com/`. Dort findet man auch weiterführende Literatur und Dokumentationen zu Quartus II.

Quasi als „Kick-Start“ fürs Labor gibt es im kommenden Abschnitt nochmals eine Zusammenfassung zu VHDL und insbesondere zu Fallstricken und häufig auftretenden Fehlerquellen speziell im Labor.

3.1 Kurze VHDL-Zusammenfassung für den Laborbetrieb

Wie bereits in der Einleitung erwähnt ist VHDL eine Hardwarebeschreibungssprache, die sich im Gegensatz zu Software-Programmiersprachen dadurch auszeichnet, dass Abarbeitungen parallel ablaufen können. Dies beutetet, dass eine sequentielle Beschreibung, wie sie in den meisten Software-Programmiersprachen stattfindet, nicht direkt in VHDL umgesetzt werden kann. Hierzu muss selbst eine Taktung eingeführt werden. Wie dieses im einzelnen aussieht, hängt von der Situation ab. Typische Konzepte sind die State-Machine oder die Benutzung der Clock in Zusammenhang mit einem Counter. Als Grundregel kann man zuerst einmal davon ausgehen, dass alle innerhalb eines Prozesses beschriebenen Aktionen parallel ausgeführt werden und die Belegung der Signale sowie Ein- und Ausgänge erst am Ende des Prozesses stattfindet. Somit ist folgende Beschreibung in VHDL möglich:

```
ARCHITECTURE Tauschen OF Beispiel IS
SIGNAL a,b: STD_LOGIC;

BEGIN
    PROCESS(a,b)
        BEGIN
            a <= b;
            b <= a;
    END PROCESS;
END Tauschen;
```

VHDL bietet die Möglichkeit, sowohl eine strukturelle Beschreibung als auch eine Verhaltensbeschreibung eines Bausteins zu implementieren, um dessen Ein- und Ausgabeverhalten zu spezifizieren. Die strukturelle Beschreibung eines Bausteines gibt an, wie viele Instanzen der einzelnen Komponenten zur Realisierung dieses Bausteines benutzt werden, wie diese Komponenten untereinander sind und wie diese mit den Ein- und Ausgängen des Bausteins verdrahtet sind. Die Verhaltensbeschreibung dagegen beschreibt einen Baustein nicht durch seine Komponenten, sondern durch gleichzeitig parallel laufende Prozesse, welche die Belegung der Signale des Bausteines funktional bestimmen. Prozesse beschreiben damit, wie sich Signale in Abhängigkeit anderer Signale verhalten.

Bevor im nächsten Kapitel konkret die Beschäftigung mit dem DE2-Board und Quartus II behandelt wird, sollen im Folgenden einige grundlegende Konzepte von VHDL erklärt werden. Durch die weite Verbreitung von VHDL ist auch ausreichend Literatur in der Bibliothek oder online zu finden.

Zu jeder (simulierbaren und synthetisierbaren) VHDL-Beschreibung existiert eine sogenannte Top-Level-Entity. Das Hauptmerkmal dieser Komponente besteht darin, dass diese den Projektnamen trägt und dass alle nach extern geführten Signale aufgelistet sind. In unserem Fall sind dies die PINs des FPGA.

Bibliotheken

Jeder Baustein beginnt mit dem Einbinden der benutzten Bibliotheken (Libraries). Folgende zwei Zeilen werden in der Regel am Anfang von von im Labor benutzten Bausteinen stehen:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

Der Datentyp **std_logic** ist eine nach dem IEEE-Standard 1164-1993 definiert 9-wertige Logik. Dabei können Signale, die in Tabelle 3.1 angegebenen Werte annehmen:

Im Gegensatz zu dem Datentyp **bit**, der nur die Werte `'0'` und `'1'` annehmen kann, hat **std_logic** den Vorteil, alle Zustände, die auch auf der Zielplattform auftreten können, zu beschreiben. Innerhalb des Labors wird fast ausschließlich mit starken logischen Nullen und Einsen (`'0'` bzw. `'1'`) gearbeitet. Jedoch sollten trotzdem alle Signale immer als **std_logic**-Datentypen deklariert werden, da innerhalb des Syntheseprozesses von Quartus II auch andere Zustände der Signale betrachtet werden können, beispielsweise wenn ein Signal noch nicht initialisiert wurde (`'U'`) oder wenn der Zustand undefiniert ist (`'X'`).

'U'	*nicht initialisiert*
'X'	*undefiniert*
'0'	*starke logische Null*
'1'	*starke logische Eins*
'Z'	*hochohmig*
'W'	*schwach unbekannt*
'L'	*schwach logische Null*
'H'	*schwach logische Eins*
'-'	*nicht beachten (don't care)*

Tabelle 3.1: Signale der 9-wertigen Logik nach IEEE-Standard 1164-1993

Wollte man elementare Rechenoperationen benutzen, beispielsweise Plus (+) oder Minus (-), müsste man mehr aus der **ieee**-Bibliothek einbinden:

```
USE ieee.std_logic_unsigned.all;
```

Schnittstellenspezifikation

Die Schnittstellen eines Bausteines werden über die Schnittstellenspezifikation (**entity**-Konstrukt) beschrieben. Diese stellt die externe Sicht der Komponente dar. Den Namen des Bausteines kann man selbst festlegen (natürlich sollten keine reservierten Wörter, Zahlen, Umlaute, Leerzeichen usf. verwendet werden). Der oberste Baustein muss den Namen des Projektes tragen. Beispielhaft sieht dies folgendermaßen aus:

```
ENTITY baustein IS
PORT (
  portname : INOUT   datentyp
  );
END baustein;

ENTITY most_used IS
PORT (
    CLOCK_50    :   IN  STD_LOGIC;
    KEY         :   IN  STD_LOGIC_VECTOR(3 downto 0);
    SW          :   IN  STD_LOGIC_VECTOR(17 downto 0);
    LEDG        :   OUT STD_LOGIC_VECTOR(7 downto 0);
    LEDR        :   OUT STD_LOGIC_VECTOR(17 downto 0)
);
END most_used;
```

Für die Bezeichnungen der externen Schnittstellen sollten Sie auf die im Handbuch angegebenen Bezeichnungen zurückgreifen. Dies macht Ihnen die Arbeit leichter, da die jeweiligen Hardwarekomponenten auf dem Board auch mit den gleichen Bezeichnungen beschriftet sind. Die in diesen Übungen am meisten benutzten Ein- und Ausgaben sind im Beispiel *most_used* aufgelistet.

Technisch gesehen sind die PINs des FPGA direkt über Leiterbahnen mit den jeweiligen elektronischen Bauteilen verbunden. Jedoch erscheint es sehr umständlich, die PINs als Schnittstellen anzugeben. Daher ist es wesentlich einfacher, den PINs Namen und

Vektoren zuzuordnen, so dass beispielsweise alle grünen LEDs einen gemeinsamen Vektor LEDG haben. Dieser ist 8 Bit breit, also von 7 bis 0.

Um Ihnen die Arbeit abzunehmen, gibt es von Altera schon die vorbereitete Datei namens „`DE2pinassignments.csv`“, welche die Zuordnung zwischen den FPGA-PINs und den auf dem Board angegebenen Namen enthält. In Kürze wird erklärt werden, wie diese Datei eingebunden wird.

Zusätzlich soll noch darauf hingewiesen werden, dass vor der Programmierung des FPGA alle nicht verwendeten PINs auf „tri-state“ gesetzt werden sollen. Das verhindert, dass ungewollt Spannungspegel an Bauteile und PINs gelegt werden, die nicht verwendet werden. In Quartus II finden Sie unter `Assignments` →`Settings` in der Kategorie `Device` einen Button `Device and Pin Options…`. Unter der Registerkarte `Unused Pins` können Sie nun alle unbenutzten Pins auf `As input tri-stated` setzen.

Beschreibung

Nach dem Einbinden der Bibliotheken und Definieren der Schnittstellen können Sie eine Komponente durch ihr Verhalten und/oder ihre Struktur beschreiben. Der grundsätzliche Aufbau sieht folgendermaßen aus:

```
ARCHITECTURE behavior OF baustein IS
-- Hier koennen Signale definiert werden
BEGIN
--Code
END behavior;
```

Die Unterschiede zwischen einer Verhaltens- und einer Strukturbeschreibung können Sie den vorherigen Kapiteln dieses Skripts entnehmen. Im Folgenden finden Sie einige oft verwendete Routinen. Es soll noch darauf hingewiesen werden, dass Sie sich innerhalb der Komponente Hilfssignale definieren können. Diese sollten auch für die Schnittstellensignale verwendet werden. Sollten sich die Signalnamen einer Komponente ändern (beispielsweise, weil sie andere LEDs benutzen oder reservierte Namen verwendet haben), müssen Sie nur die Verdrahtung am Anfang Ihrer Komponente ändern und nicht alle Namen innerhalb der Komponente ersetzen.

Für die Clock kann das beispielsweise so aussehen:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY signaluse IS
PORT (
  CLOCK_50     : IN     STD_LOGIC;
);
END singaluse;

ARCHITECTURE behavior OF signaluse IS

SIGNAL clk        : STD_LOGIC_VECTOR;

BEGIN
  clk <= CLOCK_50;

PROCESS (clk)
  -- ...
END PROCESS;

END behavior;
```

Das Clock-Signal von einem Taster statt von `CLOCK_50` (Clock des DE2-Boards) zu nehmen, kann zum Debuggen teilweise nützlich sein.

Hilfreiche Routinen

- Bedingungen

```
ARCHITECTURE behavior OF Bedingung IS
SIGNAL a: STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR (2 DOWNTO 0);
BEGIN
PROCESS(a,b)
BEGIN
  IF (a="01") THEN
      b<="001";
  ELSIF (a="11") THEN
      b<="000";
  ELSE
      b<="111";
  END IF;
END PROCESS;
END behavior;
```

- Fallunterscheidung

```
ARCHITECTURE behavior OF Fallunterscheidung IS
SIGNAL a,b: STD_LOGIC_VECTOR (2 DOWNTO 0);
BEGIN
PROCESS(a,b)
BEGIN
  CASE a IS
      WHEN "001" => b <= "000";
      WHEN "011" | "110" => b <= "100";
      WHEN OTHERS => b <= "111";
  END CASE;
END PROCESS;
END behavior;
```

- Bedingung mit „WITH ...SELECT“

```
ARCHITECTURE structure OF Verzweigung IS
SIGNAL a            :STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL out,c,d,e        : STD_LOGIC;
BEGIN
WITH a SELECT b <=  c WHEN "10",
                    d WHEN '01',
                    e WHEN OTHERS;

END structure;
```

- Bedingung für Signale außerhalb eines Prozesses

```
ARCHITECTURE structure OF Signalbedingung IS
SIGNAL a,b,c,d: STD_LOGIC;
BEGIN
a<= NOT a WHEN b = '1' ELSE (c XOR d);
END structure;
```

- Clock-Signale

```
ARCHITECTURE behavior OF Clocksignal IS
SIGNAL clk: STD_LOGIC;
SIGNAL count: STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
clk <= clock_50;
PROCESS(clk)
  IF (clk='1' AND clk'event) THEN -- wenn clk steigende Flanke
      count <= count + 1;
  ELSE
      count <= count;
  END IF;
END PROCESS;
END behavior;
```

- Variablen und Schleifen

```
ARCHITECTURE behavior OF Schleife IS
SIGNAL clk, reset: STD_LOGIC;
SIGNAL register: STD_LOGIC_VECTOR(6 DOWNTO 0);
BEGIN
clk <= clock_50;
reset <= NOT KEY(0);

PROCESS(clk)
  VARIABLE i : INTEGER := 0;
  IF reset ='1' THEN
      FOR i IN 0 TO 5 LOOP
          IF i = 0 OR i = 1 THEN
              register(i) <= NOT register(i+1);
          ELSE
              register(i) <= register(i+1);
          END IF;
      END LOOP;
  END IF;
END PROCESS;
END behavior;
```

Simulation und Synthese

Ein wichtiger Unterschied besteht zwischen simulierbarem (oder funktionellem) und synthetisierbarem VHDL-Code. Wie der Name schon sagt, lässt sich simulierbarer Code nicht unbedingt auf der Hardware ausführen. Im Labor wird in der Regel synthetisierbarer Code geschrieben. Dieser zeichnet sich dadurch aus, dass er vom Übersetzungsprogramm/Synthesetool (in unserem Fall Quartus II) in eine Netzliste übersetzt werden und somit auf den FPGA programmiert werden kann. Der simulierbare Code dagegen kann auch Elemente enthalten, welche nur für die Simulation übersetzt werden können, woraus aber keine Synthese und damit keine Netzliste erstellt werden kann. Dies soll an folgendem, abstrakten Beispiel anschaulicher gemacht werden:

```
ARCHITECTURE Oszillator OF Simulationsbeispiel IS
CONSTANT clk: time := 1000 ns;
BEGIN
PROCESS(clk, input)
BEGIN
  out <= REJECT 10 ns INERTIAL NOT input AFTER 10 ns;
  WAIT FOR clk/2;
  out <= REJECT 10 ns INERTIAL input AFTER 10 ns;
  WAIT FOR clk/2;
END PROCESS;
END Oszillator;
```

Mit den Befehlen **REJECT**, **INERTIAL** und **AFTER** kann das (zeitliche) Verhalten von realen Bausteinen beschrieben werden. Soll der Code synthetisiert werden, ist dies nicht mehr nötig bzw. möglich, da die Beschreibung auf den realen Bauelementen läuft und daher die Zeitangaben nicht mehr zum Tragen kommen. Bei der **WAIT**-Anweisung

handelt es sich ebenfalls um eine nicht synthetisierbare Anweisung. Die reale Hardware kann nicht wie eine Simulation oder ein Softwareprozess einfach „anhalten“. Soll ein Warten implementiert werden, muss dieses indirekt über einen Counter realisiert werden. Auch das Angeben von Konstanten (sowie von Variablen) ist nur sehr eingeschränkt möglich. Das Zuweisen von direkten Zeiten – wie im Beispiel – ist nicht möglich. Daher kann auf diese Weise kein Taktgeber realisiert werden. Allgemein können Variablen wie Integer-Variablen nur dazu verwendet werden, den „Schreibaufwand“ zu minimieren, jedoch nicht als Variablen, die innerhalb der Laufzeit benutzt werden können, wie sie in Software-Programmiersprachen auftreten. Ein Beispiel für einen der wenigen Fälle, wo `Integer` zur Anwendung kommt:

```
PROCESS(clk)
VARIABLE i : INTEGER := 0;
BEGIN
IF (clk = '1' AND clk'event) THEN
  FOR i IN 0 TO 10 LOOP
      IF i = 0 OR i = 2 THEN
          shift_reg(i + 1) <= shift_reg(i);
      ELSE
          shift_reg(i + 1) <= NOT shift_reg(i);
      END IF;
  END LOOP;
ELSE
  shift_reg <= shift_reg
END IF;
END PROCESS;
```

Dies ist äquivalent zu folgendem Code. Das Synthesetool wird auch, bevor die Netzliste erstellt wird, das obige Beispiel zu folgendem Code umbauen, da die **LOOP**-Schleife in der oben gegebenen Form nicht synthetisierbar ist.

```
PROCESS(clk)
BEGIN
IF (clk = '1' AND clk'event) THEN
  shift_reg(1) <= shift_reg(0);
  shift_reg(2) <= NOT shift_reg(1);
  shift_reg(3) <= shift_reg(2);
  shift_reg(4) <= NOT shift_reg(3);
  shift_reg(5) <= NOT shift_reg(4);
  shift_reg(6) <= NOT shift_reg(5);
  shift_reg(7) <= NOT shift_reg(6);
  shift_reg(8) <= NOT shift_reg(7);
  shift_reg(9) <= NOT shift_reg(8);
  shift_reg(10) <= NOT shift_reg(9);
  shift_reg(11) <= NOT shift_reg(10);
ELSE
  shift_reg <= shift_reg
END IF;
END PROCESS;
```

3.2 Das erste Projekt in Quartus II

Beim ersten Termin im Labor wollen wir uns mit dem Entwurfswerkzeug Quartus II beschäftigen. Da der Hersteller Altera hierfür bereits Tutorials bereit stellt, kann auf diese zurückgegriffen werden. Die Tutorials finden Sie auf der Website von Altera, `http://www.altera.com/`, unter `Training` →`University Program` →`Software Tools` → `Quartus II`. Achten Sie darauf, dass Sie als Filter als Hardwarebeschreibungssprache „VHDL“, als Board „DE2“ und als Quartus-II-Version die korrekte, im Pool installierte Version angeben.

Achtung: Bitte beachten Sie zur Inbetriebnahme der Boards unbedingt die korrekte Einschaltreihenfolge: Der kleine Schalter links vom LCD-Display muss auf **RUN** stehen, das Board mit Strom versorgt und **eingeschaltet** sein. Stecken Sie erst dann das USB-Kabel in die USB-BLASTER-Buchse. Ist der Schalter nicht auf **RUN** gestellt und/oder das Board beim Programmieren ausgeschaltet, wird es beschädigt! (Und Ihre Betreuer werden nicht erfreut sein, es reparieren zu müssen.)

Beim Ausschalten der Boards ist die umgekehrte Reihenfolge einzuhalten.

Im Folgenden finden Sie Vorschläge für Übungen, die Sie machen können, um sich mit Quartus II vertraut zu machen.

Aufgabe 1

Bearbeiten Sie das Tutorial *Quartus II Introduction*. Lassen Sie bitte den Teil des Tutorials über das „active serial programming“ aus. Damit Sie die PINs des Boards nicht einem Signal zuweisen müssen, stellt Altera ein sogenanntes PIN-PIN-Assignment-File für Sie bereit, wie schon weiter oben erklärt. Diese Datei finden Sie auf der Altera-Website unter `Training` →`University Program` →`Dev. & Education Boards` →`DE2` oder direkt unter `ftp://ftp.altera.com/up/pub/Altera_Material/9.1/Boards/DE2/DE2.qsf`. Es wird Ihnen empfohlen, die Datei an einer zentralen Stelle zu speichern, da Sie diese fast bei jedem Projekt benötigen werden.

Vergessen Sie dabei nicht, wie in Abschnitt 3.1 beschrieben, alle nicht benutzten PINs auf „tri-state input“ zu setzen. Dies müssen Sie bei allen folgenden Projekten tun.

Aufgabe 2

Ziel dieser Aufgabe ist es, zu lernen, wie einfache Eingabe- und Ausgabe-Bausteine an einen FPGA angebunden werden. Dazu sollen die 18 Eingabeschalter SW_{17-0} mit den 18 roten LEDs $LEDR_{17-0}$ direkt verbunden werden. Das Handbuch *DE2 User Manual* gibt dazu vor, dass beispielsweise der Schalter SW_0 mit dem PIN $N25$ und die $LEDR_0$ mit dem PIN $AE23$ verbunden ist. Um nicht alle PINs suchen zu müssen, bietet es sich an, die Namen aus dem DE2-Pin-Assignment-File zu verwenden.

Nun können Sie eine Zuordnung in folgender Form vornehmen:

```
LEDR(17) <= SW(17);
LEDR(16) <= SW(16);
...
LEDR(0) <= SW(0);
```

Dabei scheint es sinnvoll, beide Schnittstellen als Standard-Logik-Vektoren zu implementieren. Damit kann die Zuordnung wie folgt gemacht werden:

```
LEDR(17 downto 0) <= SW(17 downto 0);
```

Oder, da beide Vektoren die gleiche Größe haben:

```
LEDR <= SW;
```

Erstellen Sie nun ein Projekt und testen Sie dieses auf dem Board, das die 18 Schalter mit den 18 LEDs so verbindet, dass jeweils ein Schalter eine LED ein- bzw. ausschaltet.

Erweitern Sie Ihren Code so, dass der folgende Code entsprechend für jeden der Schalter funktioniert. Dabei soll auch der Vorteil der Vektoren genutzt werden, so dass am Ende nicht für jede LED eine eigene Zeile geschrieben werden muss.

```
LEDR(0) <= SW(0) AND SW(9);
LEDR(1) <= SW(1) AND SW(10);
...
```

4 Literaturverzeichnis

[Arm88] James R. Armstrong. *Chip-Level Modeling with VHDL*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Coe89] David Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, Boston, 1989.

[IEE88] IEEE. *IEEE Standard VHDL Language Reference Manual – Std 1076-1987*. New York, 1988.

[LSU89] Roger Lipset, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, Boston, 1989.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison Wesley, Reading, Massachusetts, 1980.