Die Beschreibung der Gesamtfunktion besteht aus einem Abtastprozess zur Modellierung des Zustandsregisters und der Übergangsfunktion sowie je einer nebenläufigen Signalzuweisung mit einem Funktionsaufruf für die Bildung der beiden Ausgabesignale.

```
-- Vereinbarungen in der Entwurfseinheit
signal z: tZustand;
signal x: tByte;
signal I_del, T, Start_del, Busy, TxD: STD_LOGIC;
Sender:process(T, I_del)
begin
 if I_del='1' then
  z <= (z=>zStopp, idx=>"000", P=>'0');
 elsif RISING_EDGE(T) then
  z <= fz(Start_del, x, z);
 end if;
end process;
-- nebenläufige Signalzuweisung für die Ausgabe
TxD  <= f_TxD(x, z);
Busy <= f_Busy(z);
```

⇒WEB-Projekt: P3.4/Test_UART3.vhdl

An dieser Stelle soll die Demonstration des Entwurfsablaufs enden. Nach dem Entwurf des Operationsablaufgraphen, der Übergangsfunktion und der Ausgabefunktionen für den Sender würden als Nächstes dieselben Entwurfsschritte für den Empfänger folgen. Dann sind der Sender und der Empfänger als Entwurfseinheiten zu beschreiben, zu simulieren, zu synthetisieren, nochmal zu simulieren und als fertige Schaltungen zu testen.

Nach dem berühmten Motto der Software-Technik

Plan beats no Plan!

ist auch für den Hardware-Entwurf eine Politik der kleinen Schritte ratsam. Zu Beginn des Entwurfsprozesses empfiehlt es sich, die Zielfunktion Software-orientiert zu beschreiben, zu simulieren und solange nachzubessern, bis sie alle Anforderungen erfüllt. Dann ist schrittweise auf eine Hardware-orientierte Beschreibung bis hin zu einer synthesefähigen Beschreibung zuzuarbeiten. Nach jedem Teilschritt sollten Testschritte folgen und erkennbare Fehler beseitigt werden. Kleine überschaubare Schritte sind die Voraussetzung dafür, dass auch für größere Entwurfsprojekte der Entwurfsaufwand kalkulierbar bleibt und die entworfenen Systeme später zuverlässig arbeiten [29].

3.4.6 Entwicklung eines CORDIC-Rechenwerks

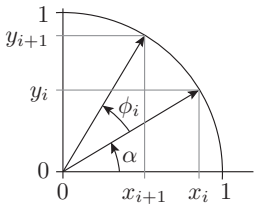
Der Sender und der Empfänger einer UART sind noch relativ kleine Schaltungen, die nur aus einigen Hundert Gattern bestehen. Das zweite Beispiel – die Entwicklung eines Rechenwerks, das für einen vorgegebenen Winkel ϕ

den Sinus und den Kosinus berechnet – ist eine Stufe komplexer und selbst für einen erfahrenen Entwickler kaum noch ohne Zwischenschritte mit simulierbaren Teilergebnissen zu lösen.

Schritt 1: Ableitung des Algorithmus aus der Aufgabenstellung

Der gebräuchliche Algorithmus für die Berechnung von Winkelfunktionen ist ein CORDIC [39]. Das Akronym CORDIC steht für **Co**ordinate **R**otation **Di**gital **C**omputer und beschreibt einen Algorithmus, der genau wie der Divisionsalgorithmus in Abschnitt 2.6.5 im Wesentlichen aus Additionen und Verschiebeoperationen besteht. Die Grundidee ist ein rotierender Zeiger in einem kartesischen Koordinatensystem. Wie aus Abb. 3.35 ablesbar ist, ergeben sich die Koordinaten eines rotierenden Vektors aus dem Produkt einer Rotationsmatrix mit dem Startvektor:

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} \cos(\phi_i) & -\sin(\phi_i) \\ \sin(\phi_i) & \cos(\phi_i) \end{pmatrix} \cdot \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$



$$\begin{aligned} x_i &= \cos(\alpha) \\ y_i &= \sin(\alpha) \\ x_{i+1} &= \cos(\alpha + \phi_i) = \cos(\alpha) \cdot \cos(\phi_i) - \sin(\alpha) \cdot \sin(\phi_i) \\ &= x_i \cdot \cos(\phi_i) - y_i \cdot \sin(\phi_i) \\ y_{i+1} &= \sin(\alpha + \phi_i) = \cos(\alpha) \cdot \sin(\phi_i) + \sin(\alpha) \cdot \cos(\phi_i) \\ &= x_i \cdot \sin(\phi_i) + y_i \cdot \cos(\phi_i) \end{aligned}$$

Abb. 3.35. Rotation eines Zeigers der Länge eins in einem Koordinatensystem

Bei dem CORDIC-Algorithmus wird der Drehwinkel aus einer festen Anzahl konstanter Winkelschritte abnehmender Größe zusammengesetzt und jeweils der zugehörige Sinus- und Kosinus-Wert berechnet (Abb. 3.36). In jedem Schritt besteht die Möglichkeit, den nächstkleineren Winkelschritt zu addieren

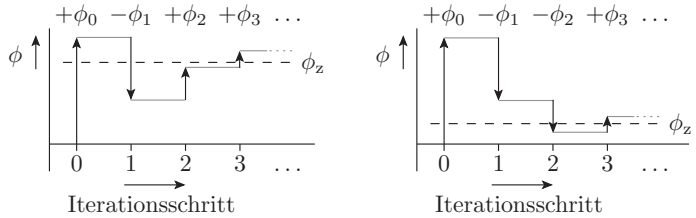


Abb. 3.36. Schrittweise Annäherung an den Zielwinkel durch Addition oder Subtraktion konstanter Winkelschritte abnehmender Größe

oder zu subtrahieren. Zur Berechnung von Sinus und Kosinus ist der Winkel ϕ_z vorgegeben und die Vorzeichen der Winkelkonstanten ϕ_i werden bei der Summation so gewählt, dass die Summe den Zielwinkel möglichst genau annähert. Mit einer ähnlichen Iteration werden auch die Exponentialfunktion, sinh(x) und praktisch alle anderen Winkelfunktionen berechnet. Eine ausführlichere Darstellung zum CORDIC-Algorithmus ist in [39] zu finden.

Wenn ein Algorithmus in Hardware ausgelagert wird, werden vorher alle anderen Optimierungsmöglichkeiten ausgereizt. Denn anderenfalls genügt auch eine Software-Lösung. Der CORDIC führt parallel zu den n Additionen oder Subtraktionen der Winkelkonstanten ϕ_i die folgenden n Matrixmultiplikationen aus:

$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} \cos(\pm\phi_{n-1}) & -\sin(\pm\phi_{n-1}) \\ \sin(\pm\phi_{n-1}) & \cos(\pm\phi_{n-1}) \end{pmatrix} \cdot \ldots \cdot \begin{pmatrix} \cos(\pm\phi_0) & -\sin(\pm\phi_0) \\ \sin(\pm\phi_0) & \cos(\pm\phi_0) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Den Hauptaufwand verursachen die vier Multiplikationen mit den Konstanten $\cos(\phi_i)$ und $\sin(\phi_i)$ in jedem Iterationsschritt. Durch Ausklammern der Kosinus-Konstanten werden die Hälfte der Koeffizienten zu eins, so dass die Hälfte der Multiplikationen entfallen. Aus den Sinus-Konstanten werden Tangens-Konstanten. Der Kosinus ist eine gerade Funktion

$$\cos(\phi_i) = \cos(-\phi_i)$$

so dass das Produkt der Kosinus-Konstanten, unabhängig davon, ob die Winkel positiv oder negativ gezählt werden, auch eine Konstante ist:

$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = \left(\underbrace{\prod_{i=0}^{n-1} \cos(\pm\phi_i)}_{\text{Konstante: } SCS} \right) \cdot \begin{pmatrix} 1 & -\tan(\pm\phi_{n-1}) \\ \tan(\pm\phi_{n-1}) & 1 \end{pmatrix} \cdot \ldots \cdot \begin{pmatrix} 1 & -\tan(\pm\phi_0) \\ \tan(\pm\phi_0) & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Die Multiplikationen mit den Tangens-Konstanten werden durch Verschiebe-Operatoren ersetzt, indem für die Tangens-Konstanten absteigende Zweierpotenzen gewählt werden:

$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = SCS \cdot \begin{pmatrix} 1 & \mp 2^{-n+1} \\ \pm 2^{-n+1} & 1 \end{pmatrix} \cdot \ldots \cdot \begin{pmatrix} 1 & \mp 2^{-1} \\ \pm 2^{-1} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & \mp 2^{-0} \\ \pm 2^{-0} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Daraus folgt für die Winkelkonstanten der Iterationsschritte:

i	0	1	2	3	4	5	6	$i > 6$
$\tan(\phi_i)$	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-i}
ϕ_i	0,785	0,464	0,245	0,124	0,062	0,031	0,015	$\approx 2^{-i}$

Das Produkt der Kosinus-Konstanten für diese Winkelkonstanten beträgt:

$$SCS = \prod_{i=0}^{n-1} \cos\left(\arctan\left(2^{-i}\right)\right) \approx 0{,}617$$

Der resultierende Algorithmus ist genial einfach. In einer Wiederholschleife für jeden Tabellenindex i wird, wenn der Zielwinkel größer als der aktuelle Winkel ist,

- zum x-Wert der um i Stellen vorzeichenerweitert nach rechts verschobene y-Wert addiert,
- vom y-Wert der um i Stellen vorzeichenerweitert nach rechts verschobene x-Wert subtrahiert und
- zum Winkel der Tabellenwert addiert.

Sonst, wenn der Zielwinkel nicht größer als der aktuelle Winkel ist, werden aus den beiden Additionen Subtraktionen und aus der Subtraktion eine Addition. Abschließend werden die so berechneten Zwischenergebnisse für den Sinus- und den Kosinuswert mit der Konstanten SCS multipliziert (Abb. 3.37).

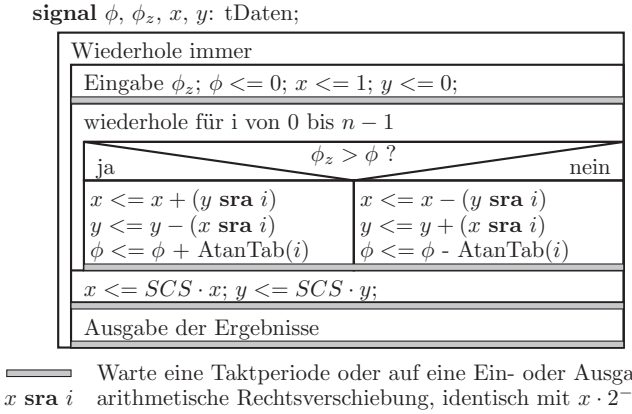


Abb. 3.37. Optimierter Algorithmus für die Berechnung von Sinus und Kosinus

Das Struktogramm in der Abbildung berechnet die Zwischenergebnisse in Signalen. Zwischen der Zuweisung an ein Signal und der Auswertung des zugewiesenen Wertes muss eine Warteanweisung liegen. Denn der Simulator aktualisiert Signalwerte nur, wenn der Prozess schläft. Der skizzierte Algorithmus benötigt deshalb nach der Eingabe, nach jedem CORDIC-Schritt und nach den Multiplikationen je einen Warteschritt. Zusätzlich wartet ein Rechenwerk im Allgemeinen auch auf die Bereitstellung neuer Eingabewerte und auf das Abholen der Ergebnisse. Die Warteanweisungen legen den Zeitablauf fest.

Schritt 2: Ein erstes Simulationsmodell

Das erste Simulationsmodell sei ein Software-orientiertes Ablaufmodell mit Texteingaben und -ausgaben und mit Warteanweisungen vom Typ »warte für eine Taktperiode«. Zu praktisch jedem Simulationsmodell gehört ein Package mit den Vereinbarungen der modellspezifischen Datentypen, Konstanten und Unterprogrammen. Der Beispielalgorithmus benötigt einen Datentyp für vorzeichenbehaftete Zahlenwerte im Bereich zwischen -2 und 2 und den zugehörigen Vektortyp für die Konstantentabelle. Für die ersten Simulationsexperimente genügen reellwertige Zahlentypen für die Wertedarstellung:

```
subtype tDaten is REAL range -2.0 to 2.0;
type tDatenArray is array (NATURAL range <>) of tDaten;
```

Später ist der Zahlentyp durch einen geeigneten Bitvektortyp zu ersetzen. Der Verschiebeoperator »sra«[16] für die n-fache Halbierung von Bitvektor-Werten ist für Zahlentypen durch die Division durch eine Zweierpotenz zu ersetzen:

```
function "sra"(a: tDaten; b: NATURAL) return tDaten is
begin
 return a/(2.0**b);
end function;
```

⇒WEB-Projekt: P3.4/cordic1_pack.vhdl

Für die Verständlichkeit, Änderungsfreundlichkeit und Fehlervermeidung ist es zu empfehlen, Konstanten durch Funktionen berechnen zu lassen, statt sie als Ziffernfolgen manuell einzutippen. Die Winkelfunktionen für reelle Zahlen, die für die Konstantenberechnungen benötigt werden, stellt das standardisierte Package IEEE.MATH_REAL bereit:

```
use IEEE.MATH_REAL.COS;
use IEEE.MATH_REAL.ATAN; -- laut [11] arctan
```

Die Funktion zur Berechnung der Tabelle der Winkelkonstanten benötigt die Anzahl der Iterationsschritte N als Eingabeparameter und liefert ein Ergebnis vom Typ »tDatenArray«:

```
function fAtanTab(N: POSITIVE) return tDatenArray is
 variable Tab: tDatenArray(0 to N-1);
begin
 for idx in Tab'RANGE loop
  Tab(idx) := atan(1.0 sra idx);
 end loop;
 return Tab;
end function;
```

⇒WEB-Projekt: P3.4/cordic1_pack.vhdl

Die Konstante SCS wird aus der Konstantentabelle berechnet und hat den Typ »tDaten«:

[16] arithmetische Rechtsverschiebung

```
function fSCS(Tab: tDatenArray) return tDaten is
 variable w: tDaten := 1.0;
begin
 for idx in Tab'RANGE loop
  w := w * cos(Tab(idx));
 end loop;
 return w;
end function;
```

⇒WEB-Projekt: P3.4/cordic1_pack.vhdl

Für die Untersuchung, ob der Algorithmus richtig funktioniert, genügt ein einzelner Prozess in einem Testrahmen, der zur Eingabe eines Winkels auffordert, den Sinus- und den Kosinuswert berechnet und der die Ergebnisse auf dem Bildschirm ausgibt:

```
-- Vereinbarungen im Testrahmen
signal x, y, phi: tDaten;
constant tP: DELAY_LENGTH := 10 ns;
constant cAtanTab: tDatenArray := fAtanTab(16);
constant SCS: tDaten := fSCS(cAtanTab);
...
process
 variable phi_z: tDaten;
begin
 read("Winkel im Bereich -pi/4 bis pi/4 eingeben", phi_z);
 x <= 1.0; y <= 0.0; phi <= 0.0;
 wait for tP;   -- Warte einen Takt nach der Eingabe
 for idx in cAtanTab'RANGE loop
  if phi_z>phi then
   x <= x - (y sra idx); y <= y + (x sra idx); phi <= phi + cAtanTab(idx);
  else
   x <= x + (y sra idx); y <= y - (x sra idx); phi <= phi - cAtanTab(idx);
  end if;
  wait for tP;   -- Warte einen Takt nach jedem CORDIC-Schritt
 end loop;
 x <= SCS * x; y <= SCS * y;
 wait for tP;    -- Warte einen Takt nach den Multiplikationen
 write("Ergebnis   phi=" & str(phi) & " cos=" & str(x )
   & " sin=" & str(y));
 write("Soll-Werte: phi=" & str(phi_z) & " cos=" & str(cos(phi_z)
   & " sin=" & str(sin(phi_z));
end process;
```

⇒WEB-Projekt: P3.4/cordic1.vhdl

Der Testrahmen gibt außer den berechneten Ergebnissen auch den Eingabewert des Winkels und die mit den Package-Funktionen aus IEEE.MATH_REAL berechneten Kosinus- und Sinus-Werte aus. Zur Kontrolle der Zeitabläufe können die drei Signale des Simulationsmodells »x«, »y« und »phi« auch graphisch darstellt werden.

Schritt 3: Ein Festkommaformat für die Wertedarstellung

Für die Synthese muss der reellwertige Zahlentyp, mit dem im bisherigen Simulationsmodell die Daten dargestellt werden, durch einen geeigneten Bitvektortyp ersetzt werden. Alle Operanden und Zwischenergebnisse liegen bei dem gewählten Algorithmus im Bereich zwischen ±1,7. Es bietet sich ein Format für vorzeichenbehaftete ganze Zahlen mit einem gedachten Komma hinter den führenden zwei Bits an. Der damit darstellbare Wertebereich von $10{,}0...0_2 = -2$ bis $01{,}1...1_2 = 2 - 2^{-14}$ ist ausreichend. Der Datentyp wird umdefiniert in

```
subtype tDaten is tSigned(15 downto 0);
type tDatenArray is array (NATURAL range <>) of tDaten;
```

Für den Typ »tDaten« sind als Nächstes die erforderlichen Hilfsfunktionen und Prozeduren zu programmieren. Die Zahlenwerte sollen wie in dem ersten Modell als dezimale Festkommawerte auf dem Bildschirm angezeigt werden. Nach der Konvertierung in eine ganze und weiter in eine Gleitkommazahl ist der Wert vor der Str-Konvertierung mit 2^{-14} zu multiplizieren. Für vierzehn binäre Nachkommastellen genügen vier dezimale Nachkommastellen:

```
function str_real(x: tDaten) return STRING is
begin
 return str(REAL(int(x))/(2.0**14), 4);
end function;
```

⇒WEB-Projekt: P3.4/cordic2_pack.vhdl

Für die Konvertierung der reellwertigen Konstanten in die Festkommadarstellung ist der Wert zuerst mit 2^{14} zu multiplizieren, in eine ganze Zahl und dann weiter in einen 16-Bit-Vektor vom Typ »tSigned« zu konvertieren. Die Assert-Anweisung kontrolliert, dass die Eingabewerte im darstellbaren Bereich liegen:

```
function to_tDat(w: REAL) return tDaten is
begin
 assert w>=-2.0 and w<=1.9999 report "Wertebereichsverletzung"
  severity FAILURE;
 return to_tSigned(INTEGER(w*2.0**14), 16);
end function;
```

⇒WEB-Projekt: P3.4/cordic2_pack.vhdl

Mit Hilfe dieser Konvertierungsfunktion werden unter anderem auch die Konstanten in den Funktionen »fATanTab(...)« und »fSCS(...)« in ihre Festkommadarstellung konvertiert. Bei einer Multiplikation der Festkommazahlen vom Typ »tDaten« verdoppelt sich die Anzahl der Vorkommastellen auf vier und die der Nachkommastellen auf 28. Zur Konvertierung des Ergebnisses in das Darstellungsformat von »tDaten« werden die zusätzlichen Vorkomma- und Nachkommastellen wieder abgeschnitten:

```
function mult(a, b: tDaten) return tDaten is
 constant y: tSigned(2*tDaten'LENGTH-1 downto 0) := a*b;
begin
 return y(y'HIGH-2 downto y'HIGH-tDaten'LENGTH-1);
end function;
```

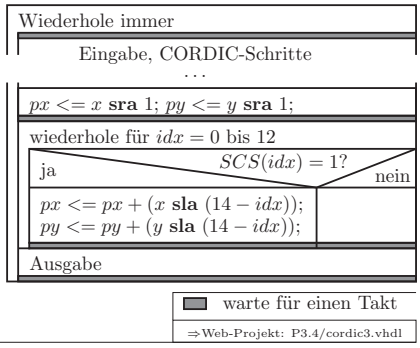
⇒WEB-Projekt: P3.4/cordic2_pack.vhdl

Im Testrahmen selbst ist das Package auszutauschen (siehe Web-Projekt CORDIC/cordic2.vhdl). Die Konstanten und Eingabewerte sind vor ihrer Verarbeitung in den neu definierten Bitvektortyp zu konvertieren. Die Multiplikationsoperatoren sind durch die Mult-Funktion und die Str-Funktionen für die Ausgabedaten durch die neu definierte Funktion »str_real(....)« zu ersetzen. Das überarbeitete Simulationsmodell verhält sich fast wie das erste. Nur die Ergebnisse sind bei einer Berechnung mit nur 16 Bit etwas ungenauer.

Schritt 4: Ablaufoptimierung

In der nächsten Nachbesserungsiteration sollen Hardware- und Zeitaufwand genauer betrachtet werden. Den größten Schaltungsaufwand erfordern, falls in Hardware realisiert, die beiden abschließenden 16-Bit-Multiplikationen. Da eine Multiplikation auch durch bitverschobene bedingte Additionen nachgebildet werden kann, bietet es sich im Beispiel an, die vorhandenen Addierer und Shifter auch für die Multiplikationen zu nutzen. Abbildung 3.38 zeigt die Struktogrammerweiterung um eine Schleife mit bedingten Additionen und den zugehörigen VHDL-Code. Der Wert der Konstanten *SCS* liegt zwischen 0,5 und eins. Das größte Bit, das eins ist, ist $SCS(13)$ mit dem Stellenwert 0,5. Die Register für die Akkumulation der Produkte werden entsprechend mit dem halbierten ersten Faktor »x sra 1« bzw. »y sra 1« initialisiert. Dann wird in einer Schleife für alle niederwertigeren Bits von *SCS*, falls diese eins sind, der

```
signal px, py: tDaten;
constant SCS: tDaten := ...
...
px <= x sra 1;  py <= y sra 1;
wait for tP;
 for idx in 12 downto 0 loop
  if SCS(idx)='1' then
    px <= px + (x sra (14-idx));
    py <= py + (y sra (14-idx));
  end if;
  wait for tP;
 end loop;
```



⇒Web-Projekt: P3.4/cordic3.vhdl

Abb. 3.38. Nachbildung der abschließenden Multiplikationen mit der Konstanten *SCS* durch Additionen und Verschiebeoperationen

um $14 - idx$ Stellen arithmetisch (vorzeichenerweitert) nach rechts verschobene erste Faktor addiert. Der Preis für die Einsparung der beiden Multiplizierer sind vierzehn zusätzliche Berechnungsschritte. Das heißt aber nicht unbedingt, dass die Berechnung so viel länger dauert. Denn ohne die Multiplizierer wird die Schaltung kleiner und einfacher und lässt sich möglicherweise mit einer höheren Taktfrequenz betreiben.

Unser Entwurfsbeispiel endet hier, obwohl es noch nicht abgeschlossen ist. Es gibt weitere Optimierungsmöglichkeiten. Die Signale und Zeitabläufe für die Übernahme der Eingabedaten und die Übergabe der Ergebnisse sind noch zu definieren und in die Modellabläufe einzupassen. Die Warteanweisungen sind an Takten auszurichten etc. und zum Abschluss ist die ablauforientierte Funktionsbeschreibung in eine synthesefähige Beschreibung zu transformieren.

> *Selbst für einen erfahrenen Entwickler ist ein Hardware-Entwurf mit der Komplexität eines CORDIC-Rechenwerks (mehrere Tausend Gatter) kaum noch ohne Zwischenschritte mit simulierbaren Teilergebnissen zu lösen.*

3.4.7 Zusammenfassung und Übungsaufgaben

Ein fehlerarmer Entwurf mit einem kalkulierbaren Aufwand verlangt ein planvolles Vorgehen. Komplexe Entwürfe sind sowohl strukturell als auch im zeitlichen Ablauf in separat zu testende Beschreibungen aufzuspalten. Für den zeitlichen Entwurfsablauf empfiehlt es sich, die Zielfunktionen zuerst wie eine normale Software zu beschreiben und zu simulieren, dann mit Warteanweisungen die zeitlichen Abläufe einzubauen und dann iterativ die Abläufe, die Operationen und die Datenstrukturen zu optimieren. Die so entwickelten Algorithmen werden anschließend über ihre Operationsablaufgraphen in synthesefähige Beschreibungen überführt, simuliert, synthetisiert, wieder simuliert und nach der Umsetzung in die Hardware getestet. Für die strukturelle Aufteilung der Beschreibungen in den Anfangsphasen des Entwurfsprozesses empfiehlt sich eine Auslagerung der Typvereinbarungen, der Beschreibungen von kombinatorischen Funktionen und der Beschreibungen von Testhilfen in Packages. Auch eine objektorientierte Modellentwicklung ist für Hardware in einem begrenzten Maße möglich und sinnvoll. Die Berechnungsabläufe für die kombinatorischen Funktionen sollten sich an der angestrebten Zielstruktur der Schaltungsnachbildung orientieren. So sollten Schleifen zur Beschreibung kombinatorischer Funktionen möglichst einen baumartigen statt einen kettenartigen Berechnungsfluss haben. Weiterführende und ergänzende Literatur siehe [11, 16, 39, 45].

Aufgabe 3.14

Entwerfen Sie eine VHDL-Funktion