Entwurf digitaler Systeme

Vorlesung und Übungen

Ausgabe 1.0, 09.01.2017
Autor: Klaus Gosger
Basierend auf dem Skript zu EDS von Professor Stephan Rupp

Inhaltsverzeichnis

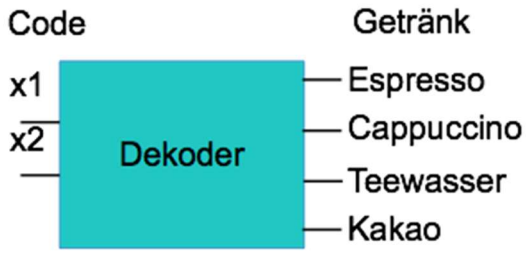
1. Kombinatorische Logik

1.1. Schaltfunktionen

Digitale Systeme lassen sich durch ihre Eingangssignale, ihre Ausgangssignale und eine Schaltfunktion beschreiben, die die Eingangssignale in die Ausgangssignale abbildet. Für eine Anzahl von n binären Eingangssignalen x_i und m binären Ausgangssignalen y_i versteht man unter der Schaltfunktion die Abbildung

$$f: \{0,1\}^n \rightarrow \{0,1\}^m \qquad (1.1)$$

Die Schaltfunktion lässt sich durch eine Wertetabelle (bzw. Wahrheitstabelle) beschreiben und als digitale Schaltung in Form eines Schaltnetzes realisieren. Als Beispiel sei ein einfacher Dekoder genannt, der zwei Eingangssignale x_1 und x_2 vier Ausgangssignale y_1 bis y_4 übersetzt, wie in der folgenden Abbildung gezeigt.



Wertetabelle

x1	x2	y1	y2	y3	y4	Kommentar
0	0	1	0	0	0	Espresso
0	1	0	1	0	0	Cappuccino
1	0	0	0	1	0	Teewasser
1	1	0	0	0	1	Kakao

Bild 1.1 Dekoder mit Wertetabelle

In dem dargestellten Beispiel ist die Anzahl der Eingangssignale n = 2 und die Anzahl der Ausgangsignale m = 4. Jedes Ausgangssignal nimmt gemäß der Zustände der Eingangssignale 4 mögliche Werte ein, wie die Wertetabelle zeigt. Die Anzahl möglicher Werte für n binäre Eingangssignale beträgt 2^n, d.h. die Wertetabelle steigt exponentiell mit der Anzahl der Eingangssignale: für n = 8 ergeben sich 256 Möglichkeiten, für n = 32 bereits über 4 Milliarden Möglichkeiten.

Bei der Realisierung als Schaltnetz strebt man natürlich eine Minimierung des Aufwandes an. Eine Möglichkeit der Realisierung geht von den Kombinationen an Eingangssignalen aus, an denen die Ausgangsfunktion „1“ beträgt. Diese Kombinationen bildet man durch das Schaltnetz ab. Alle anderen Kombinationen ergeben hierbei ein Ausgangssignal „0“. Die folgende Abbildung zeigt ein Beispiel einer Realisierung für die oben abgebildete Wertetabelle.

DNF

y1 = x1‘ x2‘
y2 = x1‘ x2
y3 = x1 x2‘
y4 = x1 x2

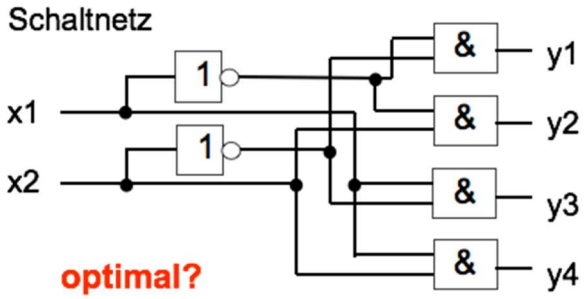


Bild 1.2 Realisierung als Schaltnetz

Die Ausgangsfunktion y_1 hat genau eine Kombination von Eingangswerten, die einen Ausgangswert „1“ erzeugen, nämlich x'_1 (wobei x‘ = x negiert bedeutet) „und“ x'_2 (wobei wiederum x‘ = x negiert bedeutet). Bemerkung: In der hier verwendeten Schreibweise wird die logische Verknüpfung „und“ als Multiplikation dargestellt (mit „·“, wobei der Punkt auch weggelassen werden kann), die Verknüpfung „oder“ als Addition (mit „+“). Das zugehörige Schaltnetz realisiert das Ausgangssignal also aus der „und“-Verknüpfung der invertierten Signale x_1 und x_2.

Alle anderen Ausgangssignale ergeben sich sinngemäß aus der Verknüpfung derjenigen Kombinationen von Eingangssignalen, an denen das Ausgangssignal den Wert „1“ annimmt. Die „oder“-Verknüpfung dieser Wertekombinationen wird auch als Disjunktive Normalform (DNF) bezeichnet. Die Frage, ob das Schaltnetz bereits optimal ist, ist natürlich keine rein mathematische Frage. Abhängig von den Optimierungskriterien bzw. Kosten können hier beispielsweise die Anzahl der benötigten Gatter oder Bauelemente, bzw. auch Signallaufzeiten eine Rolle spielen.

1.2. Realisierung als Schaltnetz

Im allgemeinen Fall entspricht der Lösungsweg zur Realisierung der Schaltfunktion dem im vorausgegangenen Abschnitt genannten Beispiel. Für einfache Schaltfunktionen f: $\{0,1\}^n$ -> { 0,1} (mit einem Ausgangssignal y) ergeben sich die folgenden Schritte:

- Wertetabelle erstellen
- Disjunktive Normalform (DNF) bestimmen
- DNF kann nach Regeln der Boolschen Algebra umgeformt werden

Der Lösungsweg für mehrdimensionale Schaltfunktionen f: {0,1}n -> { 0,1}^{m} lautet sinngemäß:

- Zerlegen: $(x_1, \ldots, x_n)$ -> $(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$
- DNF für jede Funktion $f_1, \ldots, f_m$ bestimmen
- DNF nach Regeln der Boolschen Algebra umformen.

Als Beispiel sei eine 8-fache „und“-Verknüpfung genannt, wie in folgender Abbildung wiedergegeben. Obwohl die Wertetabelle insgesamt 256 Zeilen hat, ergibt nur eine einzige Kombination an Eingangsvariablen den Ausgangswert „1“. Somit besteht auch die DNF nur aus dieser einzigen Kombination.

x1	x2	...	x8	y
0	0	...	0	0
0	0	...	1	0
...				...
1	1	...	1	1

Wertetabelle

DNF

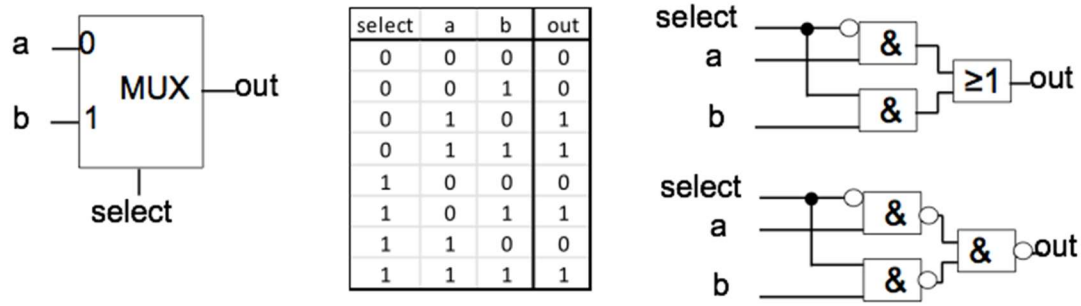
y = x1 x2 ... x8

Bild 1.3 8-fache UND-Verknüpfung

Als Realisierung als Schaltnetz könnte man die in der folgenden Abbildung gezeigten Varianten angeben. Die Kosten betragen in beiden Fällen 7 Gatter. Allerdings ist die rechts gezeigte Lösung bzgl. der Signallaufzeiten (Gatterlaufzeiten) günstiger. Diese Lösung ergibt sich mathematisch aus der DNF durch Anwendung des Assioziativgesetzes.

Bild 1.4 Schaltnetz für die 8-fache UND-Verknüpfung

Als weiteres Beispiel sei ein einfacher Multiplexer genannt. Der Multiplexer hat zwei Signaleingänge a und b. Durch ein weiteres Eingangssignal „select“ wird eins der beiden Eingangssignale ausgewählt und auf das Ausgangssignal „out“ gegeben. Die Wertetabelle lautet also wie in der folgenden Abbildung angegeben.



select	a	b	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Bild 1.5 Multiplexer

Übung 1.1: Erstellen Sie die DNF (Disjunktive Normalform).
Übung 1.2: Abbildung 1.5 zeigt auf der rechten Seite zwei mögliche Implementierungen, die durch Vereinfachungen der DNF hervorgehen. Vereinfachen Sie die DNF durch algebraische Umformungen. Hinweis. Nutzen Sie die hierbei die Beziehung x + x‘ = 1 (x „oder“ x-negiert = 1).

Musterlösung:

- DNF: out = select‘ a b‘ + select‘ a b + select a‘ b + select a b

Optimierung (Ziel: Anzahl Gatter minimieren):

- out = select‘ a (b + b‘) + select b (a‘ + a)
- out = select‘ a + select b

Einschränkung: nur NAND Logik

- out = select‘ a + select b = ((select‘ a)‘ (select b)‘)‘ (De Morgan: (x + y)‘ = x‘ y‘)

1.3. Terminologie

Wie in Abschnitt 1.2 an einigen Beispielen gezeigt, besteht die Optimierung eines Schaltnetzes als Implementierung der Schaltfunktion in der Bestimmung eines *Minimalpolynoms* der Schaltfunktion. Bei der Schaltungssynthese geht man hierbei von der Wahrheitstabelle aus. Im Falle des Multiplexers aus Abschnitt 1.3 ergab sich aus der DNF das Polynom:

out = select‘ a b‘ + select‘ a b + select a‘ b + select a b (1.2)

Hieraus ergibt sich durch algebraische Umformung das *Minimalpolynom* (1.4):

out = select‘ a (b + b‘) + select b (a‘ + a) (1.3)

out = select‘ a + select b (1.4)

Die Optimierung geschieht durch Beseitigung von Redundanz: zwei Monome, die sich genau in einer komplementären Variablen unterscheiden, können durch ihren gemeinsamen Teil ersetzt werden (Resolutionsregel). Mit Bezug auf Gleichung (1.2) gilt diese Regel für folgende Teile:

select‘ a b‘ + select‘ a b = select’ a (1.5)

select a‘ b + select a b = select b (1.6)

Diese Nachbarschaftsbeziehungen der Variablen und der negierten Variablen werden bei Umformungen mit Boolescher Algebra ausgenutzt, ebenso in dem graphischen Verfahren von Karnaugh-Veitch, sowie im algorithmischen Verfahren nach Quine-McCluskey.

Zur Terminologie an dieser Stelle noch folgende Erläuterungen: Unter einem *Minterm* versteht man die UND-Verknüpfung der Eingangsgrößen einer Zeile der Wahrheitstabelle mit Wert 1 der Ausgangsgröße. Die DNF (Disjunktive Normalform) besteht aus der disjunktive Verknüpfung (ODER-Verknüpfung) aller Minterme und bildet die Schaltfunktion eindeutig ab.

Sinngemäß versteht man unter einem *Maxterm* die ODER-Verknüpfung der Eingangsgrößen einer Zeile der Wahrheitstabelle mit Wert 0 der Ausgangsfunktion. Die Konjunktive Normalform (KNF) besteht aus der konjunktive Verknüpfung (UND-Verknüpfung) aller Maxterme und bildet die Schaltfunktion ebenfalls eindeutig ab.

1.4. Alternative Möglichkeiten zur Implementierung

Logische Gatter sind nur eine Möglichkeit zur Implementierung eines Schaltnetzes. Als weitere Logikbausteine kommen in Frage (1) Programmable Logic Arrays (PLA), (2) Speicherbausteine, z.B. Read Only Memories (ROM, auch in programmierbarer Form als EEPROM), (3) Field Programmable Gate Arrays (FPGAs).

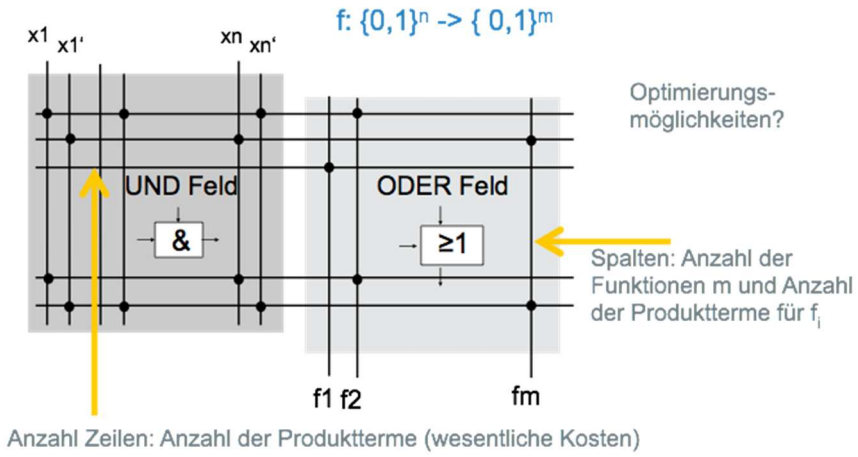


Bild 1.6 Programmable Logic Array

Abbildung 1.6 zeigt des grundsätzlichen Aufbaus eines Programmable Logic Arrays mit einem Feld zur UND-Verknüpfung der Eingangssignale x_i (bzw. der negierten Eingangssignale x'_i). Dieses Feld wird gefolgt von einem Feld zur ODER-Verknüpfung der Ergebnisse für die Ausgangsfunktionen $y_j = f_j$

(x_i). Pro Ausgangssignal f_j entspricht die UND-Verknüpfung der Eingangsgrößen entspricht unmittelbar den Monomen der DNF.

Man kann also ausgehend von der Wertetabelle unmittelbar die DNF in das PLA abbilden. Somit ist auch das Optimierungspotential durch die Möglichkeiten zur Bestimmung eines Minimalpolynoms der Schaltfunktion gegeben. Für das Beispiel Multiplexer zeigt die folgende Abbildung beide Varianten: Die Implementierung der DNF direkt aus der Wertetabelle, sowie die Implementierung des Minimalpolynoms.

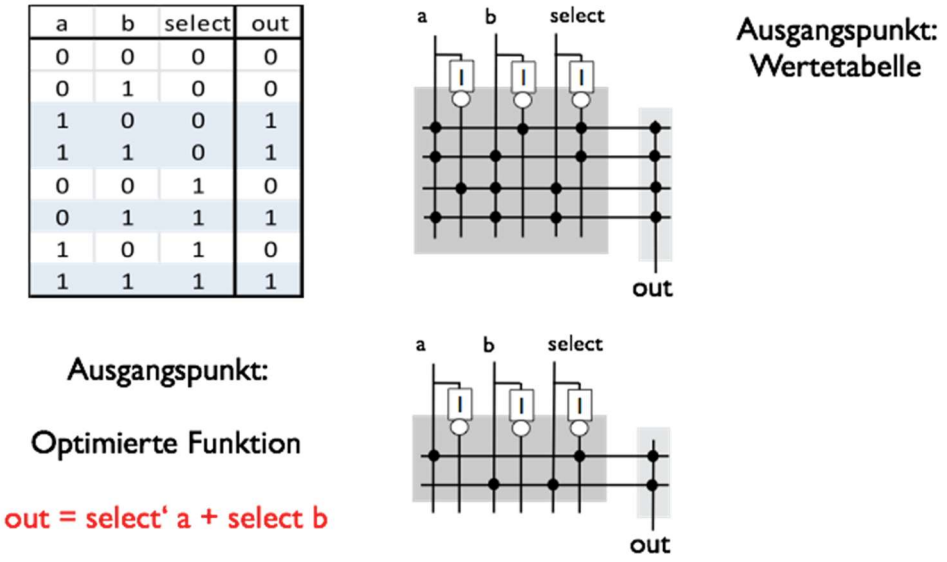


Bild 1.7 Implementierung der DNF und der optimierten Funktion im PLA

Als nächstes wird die Implementierung auf einem Speicherbaustein betrachtet. Ein ROM (Read Only Memory) (ROM) lässt sich darstellen als Dekoder für die n Eingangsvariablen x für die 2^n Zeilen der Wertetabelle mit nachfolgendem Speicher für jede Zeile. In der folgenden Darstellung ist dieser Speicher als programmierbares ODER-Feld dargestellt.

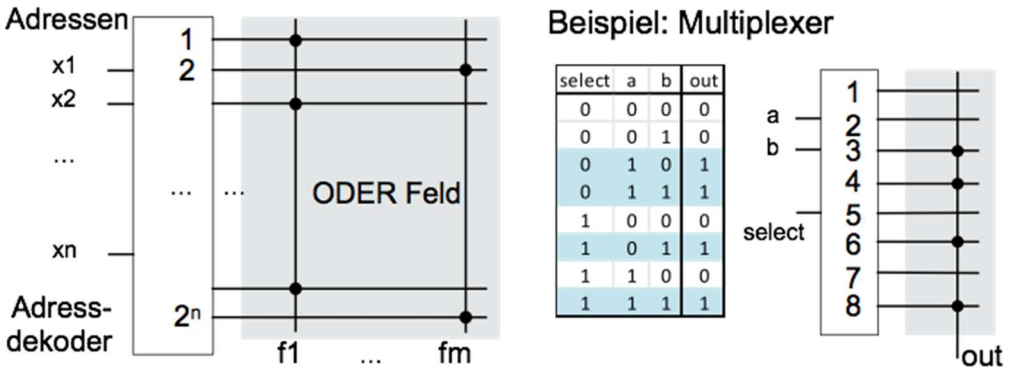


Bild 1.8 Implementierung auf ROM

Ein Speicherbaustein bietet wenig Spielraum für eine Optimierung: Alle Adressen werden dekodiert und die gesamte Wertetabelle wird im ROM gespeichert. Die Abbildung oben zeigt als Beispiel die Implementierung des Multiplexers mit drei Eingangssignalen (und 8 Speicherzellen). Dieses Verfahren ist vor allem dann wenig effizient, wenn die Wertetabelle viele Nullen enthält. Ein passendes

Beispiel wäre die 8-fache UND-Verknüpfung (siehe Abbildungen 1.3 und 1.4). Von den insgesamt 256 Speicherzellen enthält nur eine einzige den Wert „1“.

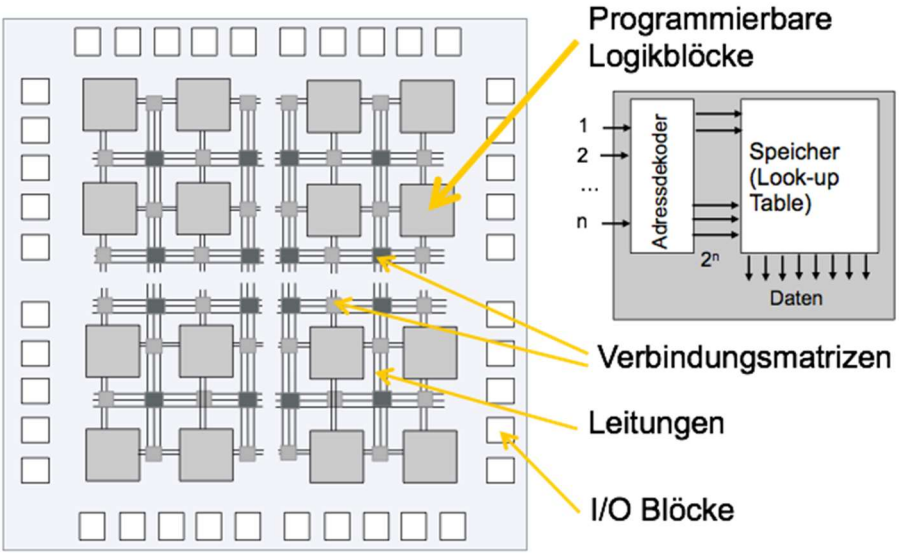


Bild 1.9 Field Programmable Gate Array

Eine große Flexibilität für den Entwurf digitaler Schaltungen bieten Field Programmable Gate Arrays (FPGAs). Wie in der Abbildung oben angedeutet, bestehen FPGAs aus herstellerspezifischen, programmierbaren Logikblöcken. Die Logikblöcke enthalten beispielsweise Adressdekoder mit folgenden Speicherzellen, die sich zum Nachschlagen in Wertetabellen (als Look-up Tabellen) ähnlich wie Speicherzellen nutzen lassen. Außerdem enthalten sind Leitungen und Verbindungsmatrizen, mit Hilfe derer sich die Logikblöcke zu komplexeren Schaltungen kombinieren lassen.

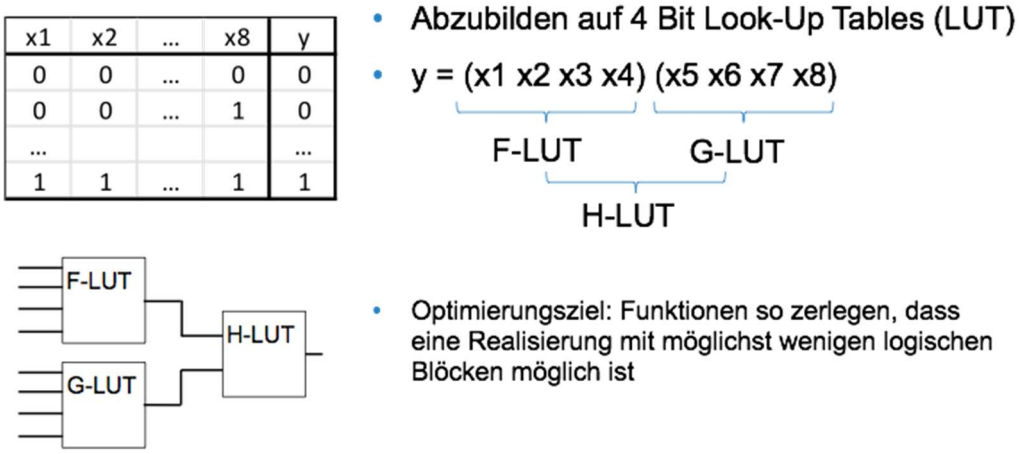


Bild 1.10 Implementierung der 8-fachen UND-Verknüpfung im FPGA

Am Rande der Bausteine finden sich die Eingangssignale und Ausgangssignale (als I/O-Blöcke in der Abbildung bezeichnet). FPGAs enthalten weiterhin Bereiche für die Taktgewinnung, sowie taktsynchrone Schaltelemente in den Logikblöcken.

Als Beispiel für eine kombinatorische Logik dient die 8-fache UND-Verknüpfung. Die Logikblöcke das verfügbaren FPGA enthalten Wertetabellen (Look-up Tables) mit 4-Bit Adressbreite (d.h. für jeweils 4 Eingangssignale). Optimierungsziel ist die Verwendung möglichst weniger Logikblöcke. Wie in der

Abbildung oben gezeigt, lässt sich die DNF in zwei Teilpolynome zu je 4 Eingangssignalen zerlegen, die sich in zwei 4-Bit Tabellen (LUT) abbilden lassen. Eine weitere Tabelle (LUT) wird für die Kombination beider Teilpolynome benötigt.

Übung 1.3: Eine Schaltfunktion mit 8 Eingangsgrößen besitzt eine Wertetabelle mit 256 Zeilen. Wie kommt es, dass in oben genanntem Beispiel eine Lösung mit nur 3 Tabellen möglich sind, die jeweils nur 4 Bit breit sind, also jeweils nur 16 Zeilen haben? Welchen Ausschnitt aus der Wertetabelle bildet diese Lösung ab?

1.5. Optimierungsverfahren

Für die Realisierung von Schaltfunktionen ist die Wertetabelle der Ausgangspunkt. Der Lösungsweg führt über die DNF (Disjunktive Normalform), für die es gilt, eine weitere Vereinfachung im Sinne eines Minimalpolynoms zu erzielen. Neben der eigenen Kreativität bei der algebraischen Umformung seinen hier noch zwei formale Optimierungsverfahren vorgestellt (1) das grafische Verfahren nach Karnaugh-Veitch (mit Hilfe der sogenannten KV-Diagramme), (2) eine algorithmische Methode nach Quine-McCluskey. Wie die algebraische Umformung zielen auch diese beiden Verfahren auf die Beseitigung von Redundanzen bei der Minimierung des Schaltpolynoms.

Das Verfahren von Karnaugh-Veitch stellt die Wertetabelle in einer Matrix dar, die es erleichtert, Redundanzen aufzuzeigen. Das Verfahren eignet sich daher nur für wenige Eingangsvariablen: für 3 Eingangsvariablen erhält man eine Matrix der Größe 2x4, für 4 Eingangsvariablen eine Matrix der Größe 4x4, wobei das grafische Abstraktionsvermögen bereits recht ordentlich beansprucht wird. Die folgende Abbildung zeigt einige Beispiele.

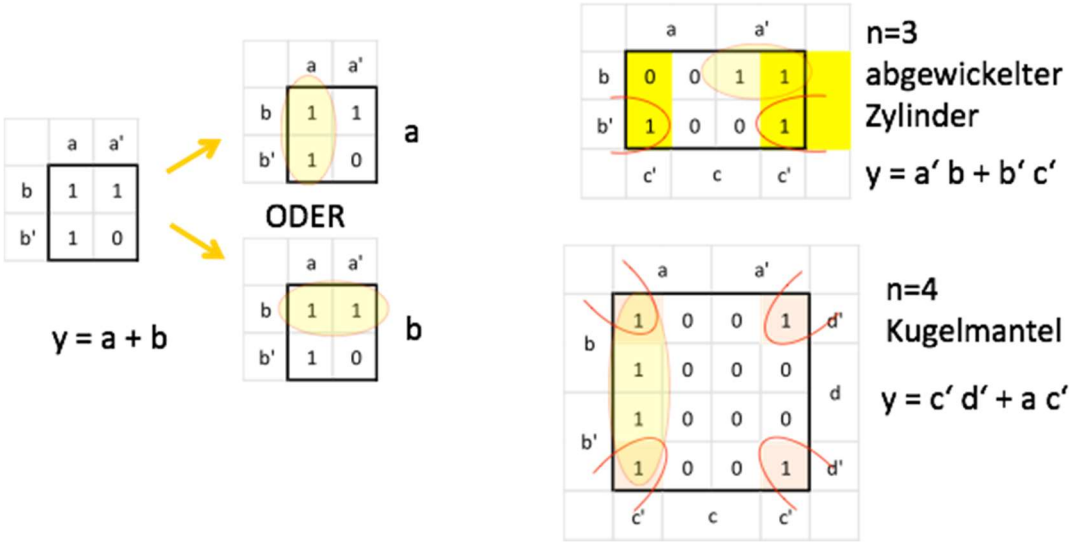


Bild 1.11 Karnaugh-Veitch Diagramme

Im linken Teil der Abbildung ist die Wertetabelle einer Schaltfunktion mit 2 Eingangsvariablen dargestellt. Die Matrix ist so angeordnet, dass Redundanzen a und a‘ (a negiert) sowie b und b‘ (b negiert) sich in benachbarten Feldern finden. An diesen Stellen genügt es, jeweils einen Zustand in das Schaltpolynom zu übernehmen. Im gezeigten Fall also b für a oder nicht-a, und a für b oder nicht-b. Man erhält so das Minimalpolynom y = a+b.

Die beiden anderen in der Abbildung gezeigten Fälle funktionieren genauso. Allerdings erfordert das Aufspüren der nachbarschaftlichen Beziehungen etwas Abstraktionsvermögen: für 3 Eingangsgrößen stellt die Matrix einen abgewickelten Zylinder dar, d.h. dir farblich hinterlegten Felder

für c‘ sind benachbart. In der oberen Zeile gibt es eine Redundanz bei c und c‘, diese Zeile ist also durch a‘ b zu beschreiben. In der unteren Zeile gibt es eine Redundanz bei a und a‘, diese Zeile kann man also durch b‘ c‘ beschreiben. Insgesamt erhält man für das Schaltpolynom y = a‘ b + b‘ c‘.

Für 4 Eingangsgrößen erhält man im KV-Diagramm eine weitere Nachbarschaft: Wenn man sich die Matrix als Abwicklung eines Kugelmantels vorstellt, sind auch die 4 Eckpunkte benachbart. Im gezeigten Beispiel sind in den Feldern in den Eckpunkten a und a‘ redundant, sowie b und b‘. Zur Beschreibung dieser Felder genügt also das Polynom c‘ d‘. Für die 4 Felder in der ersten Spalte spielen d und b keine Rolle. Insgesamt ergibt sich als Schaltpolynom also y = c‘d‘ + a c‘.

m	a	b	c	d	f1
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	0

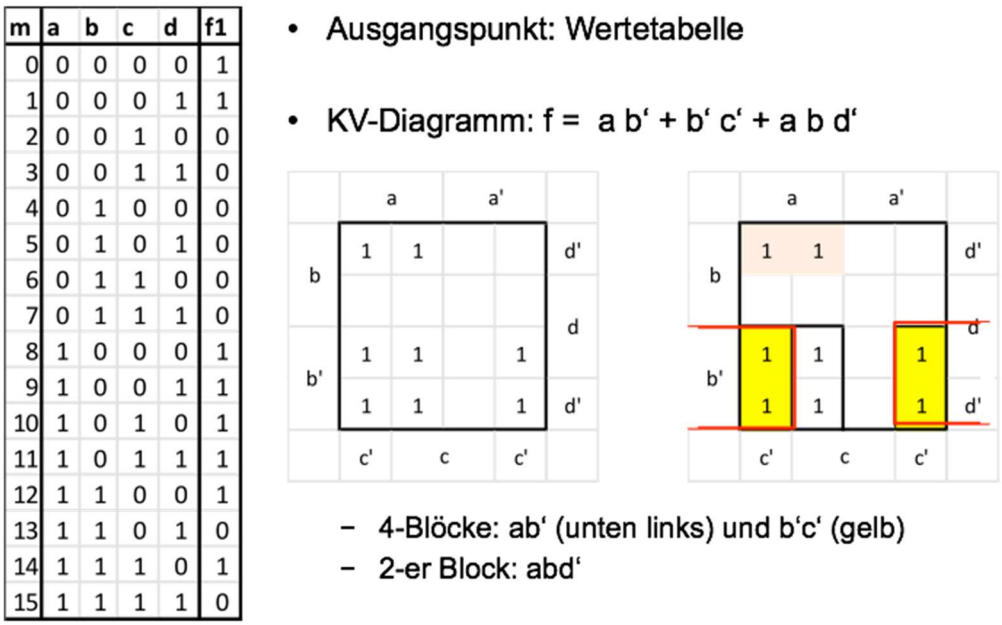


Bild 1.12 Minimierung des Schaltpolynoms mit KV-Diagramm

Übung 1.4: Abbildung 1.12 zeigt ein weiteres KV-Diagramm zusammen mit der Wertetabelle. Versuchen Sie, den Lösungsweg zu rekonstruieren.

Übung 1.5: Überprüfen Sie, ob sich die Funktion weiter minimieren lässt.

Das Verfahren von Quine-McCluskey geht aus von der Disjunktiven Normalform der Schaltfunktion. Das Verfahren wendet die Resolutionsregel wiederholt auf die Minterme der DNF aus. In diesem ersten Schritt bestimmt das Verfahren die Primimplikanten der Schaltfunktion. In einem zweiten Schritt trifft das Verfahren dann eine kostenminimale Auswahl der Primimplikanten. Was möglicherweise etwas kompliziert klingt, zeigt sich an einem Beispiel als recht einfach und zielstrebig.

Die folgende Abbildung zeigt die DNF einer Schaltfunktion mit 4 Eingangsvariablen. Von insgesamt möglichen 16 Einträgen (Mintermen) der Wertetabelle zeigt die DNF also insgesamt 8 Minterme. Im Verfahren von Quine-McCluskey werden die Minterme gemäß Ihrer Ordnungsnummer in der Wertetabelle durchnummeriert. Die Abbildung zeigt der Übersichtlichkeit halber die zugehörigen Spalten der Wertetabelle. Der Minterm m0 entspricht also der ersten Zeile, m8 der achten Zeile der Wertetabelle.

- f(a,b,c,d) = a‘b‘c‘d‘ + a‘b‘c‘d + ab‘c‘d‘ + ab‘c‘d + ab‘cd‘ + abc‘d‘ + ab‘cd + abcd‘

- Minterm-Tabelle:

Anzahl "1"	Minterm	Binär	Stufe 2
0	m0	0000	m(0,1): 000-; m(0,8):-000
1	m1	0001	m(1,9): -001
	m8	1000	m(8,9): 100-; m(8,10): 10-0
2	m9	1001	m(9,11): 10-1
	m10	1010	m(10,11): 101-; m(10,14): 1-10
	m12	1100	m(12,14): 11-0
3	m11	1011	
	m14	1110	

Bild 1.13 Beispiel zum Verfahren von Quine-McCluskey: Minterm Tabelle

Rechts in der Tabelle ist nun die 2. Stufe des Verfahrens dargestellt. Durch Anwendung der Resolutionsregel erkennt man, dass die Ausdrücke m0 und m1 sich der letzten Stelle unterscheiden, d.h. der Wert dieser Eingangsgröße hier nicht relevant ist. Man kann m0 und m1 also reduzieren auf m(0,1): 000-, wobei der Bindestrich ausdrückt, dass die letzte Stelle nicht relevant ist. Ebenso drückt das Kürzel m(0,8):-000 aus, dass hier die erste Eingangsgröße keine Rolle spielt. Alle anderen Aussagen unter Stufe 2 sind ebenso zu verstehen.

Anzahl “1“	Minterm	Binär	Stufe 2	Stufe 3
0	m0	0000	m(0,1): 000-; m(0,8=: -000	m(0,1,8,9): -00-
1	m1	0001	m(1,9): -001	
	m8	1000	m(8,9):100-; m(8,1): 10-0	m(8,9,10,11): 10- -
2	m9	1001	m(9,11): 10-1	
	m10	1010	m(10,11): 101-; m(10,14): 1-10	
	m12	1100	m(12,14): 11-0	
3	m11	1011		
	m14	1110		

Anzahl “1“	Minterm	Binär	Stufe 2	Stufe 3
0	m0	0000	m(0,1): 000-; m(0,8=: -000	**m(0,1,8,9): -00-**
1	m1	0001	m(1,9): -001	
	m8	1000	m(8,9):100-; m(8,1): 10-0	**m(8,9,10,11): 10- -**
2	m9	1001	m(9,11): 10-1	
	m10	1010	m(10,11): 101-; **m(10,14): 1-10**	
	m12	1100	**m(12,14): 11-0**	
3	m11	1011		
	m14	1110		

Bild 1.14 Beispiel zum Verfahren von Quine-McCluskey: Minterm Tabelle reduzieren

Stufe 3 wendet nun nochmals die Resolutionsregel auf die reduzierten Minterme aus Stufe 2 an. Hier bedeutet in Abbildung 1.14 das Kürzel m(0,1,8,9): -00-, dass sich die Minterme 0, 1,8 und 9 auf die Kombination b‘ und c‘ der Eingangsgrößen reduzieren lassen (die erste und letzte Eingangs-größe spielt keine Rolle, die zweite und dritte geht in negierter Form ein). Nach Abschluss der Stufe 3 nimmt man nur diejenigen reduzierten Minterme in die weitere Überlegung, die zur Abdeckung der ursprünglichen Minterme nötig sind. Das sind alle reduzierten Minterme aus Stufe 2, sowie alle nicht durch Stufe 3 abgedeckten Minterme aus Stufe 2. In der Abbildung sind diese Ausdrücke rot unterlegt.

- Primimplikanten-Tabelle:

	0	1	8	9	10	11	12	14		
m(0,1,8,9): -00-	x	x	x	x					m(0,1,8,9): -00-	b‘c‘
m(8,9,10,11): 10--			x	x	x	x			m(8,9,10,11): 10--	ab‘
m(10,14): 1-10					x			x	m(10,14): 1-10	
m(12,14): 11-0							x	x	m(12,14): 11-0	abd‘

f(a,b,c,d) = b‘c‘ + ab‘ + abd‘

Bild 1.15 zum Verfahren von Quine-McCluskey: Primimplikanten und Schaltpolynom

Die verbliebenen reduzierten Minterme sortiert man nun in einer Primimplikanten Tabelle. Wie in Abbildung 1.15 gezeigt, ist diese Tabelle so aufgebaut, dass in den Spalten gezeigt wird, welche Minterme jeweils überdeckt werden. Im Beispiel erkennt man, dass man auf den Ausdruck m(10,14) verzichten kann, da seine Minterme bereits durch m(12,14) und m(8,9,19,11) abgedeckt sind. Als Primimplikanten verbleiben also die in der Abbildung farblich markierten Ausdrücke. Das Minimalpolynom ergibt sich also als f(a,b,c,d) = b‘c‘ + a b‘ + a b d‘.

Übung 1.6: Lässt sich die Funktion weiter minimieren? Vollziehen sie die Tabelle 1.14 nach und suchen Sie nach weiter reduzierbaren Mintermen.

Übung 1.7: Geben Sie sich eine Wertetabelle für eine Schaltfunktion mit 4 Eingangsgrößen vor, die bei 8 Zeilen den Wert 1 annimmt. Optimieren Sie das Schaltpolynom nach den Verfahren von Karnaugh-Veitch und Quine-McCluskey.

1.6. Beschreibung mit VHDL

Außer in ganz einfachen Fällen wird man digitale Schaltungen mit Rechnerunterstützung entwerfen. Hierzu ist eine allerdings eine formale Beschreibung der Schaltung erforderlich. Zur Beschreibung digitaler Schaltungen haben sich Hardware-Beschreibungssprachen (HDL - Hardware Description Language) etabliert. Dieses Manuskript verwendet VHDL. Das Kürzel steht für „Very High Speed Integrated Circuit Hardware Description Language“. VHDL ist seit 1987 als IEEE Standard festgelegt (IEEE 1076).

Wie schaut die Beschreibung einer Schaltung nun in VHDL aus? Im einfachsten Fall seien einige Logikbausteine beschrieben:

```
--- Logic Gates (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity LogicGates is
 port (A, B : in std_logic;
        Qnot  : out std_logic;
        Qand  : out std_logic;
        Qor   : out std_logic;
        Qxor  : out std_logic;
        Qnand : out std_logic;
```

```
        Qnor  : out std_logic;
        Qxnor : out std_logic);
end LogicGates;

architecture RTL of LogicGates is
begin
        Qnot  <=   not  B;
        Qand  <= A and  B;
        Qor   <= A or   B;
        Qxor  <= A xor  B;
        Qnand <= A nand B;
        Qnor  <= A nor  B;
        Qxnor <= A xnor B;
end RTL;
```

Im Beispiel finden sich zwei Blöcke: (1) mit dem Schlüsselwort „entity“, (2) mit dem Schlüsselwort „architecture“. Der mit „entity“ eingeleitete Block folgt der Syntax:

```
entity <Name> is
 generic (<Liste>);
 port (<Liste>);
end <Name>;
```

Mit dem Block „entity“ werden die Schnittstellen der Hardware bzw. des Moduls beschreiben. Im Beispiel finden sich hier die beiden Eingangsports A und B, sowie die Ausgänge Qnot bis Qxnor. Der im Beispiel nicht vorhandene Eintrag „generic“ würde zusätzliche Parameter und Definitionen enthalten, die in der folgenden Beschreibung im Block „architecture“ verwendet werden.

Der mit dem Schlüsselwort „architecture“ eingeleitete Block folgt der Syntax:

```
architecture <Name> of <Entity Name> is
 <Deklarationen>;
begin
 <parallele Anweisungen>
end <Name>
```

Dieser Block enthält die Schaltungsbeschreibung. Im Beispiel ist der Typ der Architektur „RTL“ (für Register Transfer Logik) gefolgt vom gewählten Namen der Hardware, hier also die Referenz auf die zuvor beschriebene „entity“ LogicGates. Nach dem Schlüsselwort „begin“ folgen die Signalzuweisungen und die Beschreibung der Logikfunktionen. Alle Anweisungen im Block „begin“ bis „end“ werden hierbei parallel ausgeführt.

Das Schlüsselwort „library“ kennzeichnet Bibliotheken mit Funktionen, die in die VHDK-Beschreibung eingebunden werden sollen, hier z.B. die Standard Logikfunktionen. Die minimal erforderliche Beschreibung für einen Baustein besteht aus dem „entity“ Block, in dem alle Schnittstellen deklariert werden, und dem Block „architecture“ der die Funktion des Bausteins beschreibt.

Übung 1.8: Installieren Sie einen VHDL-Editor auf Ihrem Rechner, z.B. den Scriptum HDL-Editor von HDL-Works (siehe http://www.hdlworks.com). Geben Sie als Beispiel die Logik-Gatter ein. Der Editor identifiziert automatisch die Schlüsselwörter und korrekte Syntax und macht diese durch ein Farbschema kenntlich, siehe folgende Abbildung. Hinweis: Speichern Sie Ihren Textentwurf in einer Datei mit der Endung „.vhdl“. Hierdurch wird dem Editor das beabsichtigte Format für die Syntax mitgeteilt.

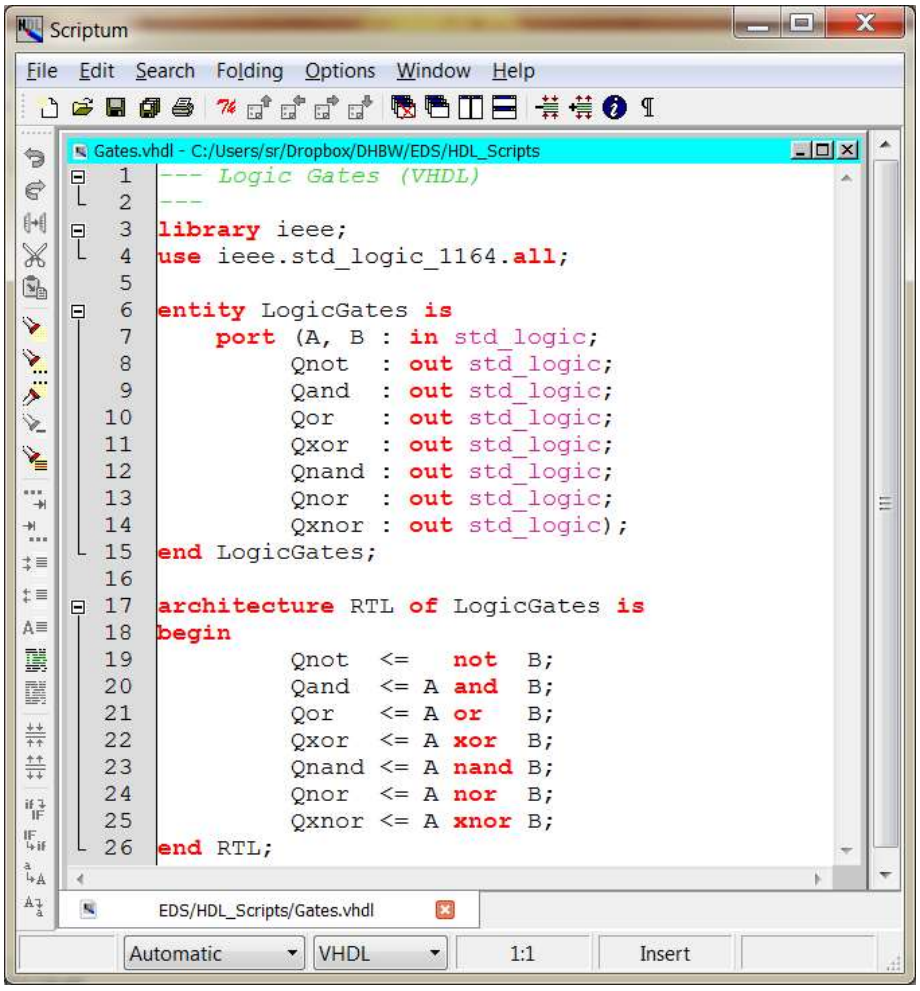


Bild 1.16 Beispiel im VHDL-Editor (HDL-Works Scriptum)

Als weiteres Beispiel sei ein Multiplexer beschrieben. Im Unterschied zu dem in Bild 1.5 wiedergegebenen Multiplexer soll dieser Multiplexer aus 8 möglichen Eingangssignalen D(0) bis D(7) ein Signal auswählen und dieses auf den Ausgang Q schalten. Zur Auswahl eines von 8 Signalen sind drei weitere Eingangssignale A(0), A(1) und A(2) erforderlich.

Übung 1.8: Beschreiben Sie die Funktion des 8-zu-1 Multiplexers durch ein Schaltsymbol und durch eine Wertetabelle. Muster: siehe Bild 1.5.

```
--- 8-to-1 multiplexer (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
entity MUX_8_to_1 is
 port (      A : in std_logic_vector (2 downto 0);
             D : in std_logic_vector (7 downto 0);
             Q : out std_logic);
end MUX_8_to_1;

architecture RTL of MUX_8_to_1 is
begin
 process (A, D)
 begin
       case A is
             when "000" => Q <= D(0);
             when "001" => Q <= D(1);
             when "010" => Q <= D(2);
             when "011" => Q <= D(3);
             when "100" => Q <= D(4);
             when "101" => Q <= D(5);
             when "110" => Q <= D(6);
             when "111" => Q <= D(7);
       end case;
 end process;
end RTL;
```

In der oben wiedergegebenen VHDL-Beschreibung findet sich wiederum der „entity“-Block mit der Beschreibung der Schnittstellen: die 8 Eingangssignale D(0) bis D(7) sind hier zu einem Vektor zusammengefasst. Die Auswahl eines der Eingänge geschieht durch das Eingangssignal A, das ebenfalls als Vektor mit 3 Signalen dargestellt ist.

Im Block „architecture“ findet sich wiederum die Beschreibung der Funktion der Schaltung. Das Schlüsselwort „process“ definiert, wie die Eingänge A und D verarbeitet werden sollen, d.h. welcher Wert den Ausgangssignalen in Abhängigkeit von den Eingängen zugewiesen werden soll. Die Anweisungen zur Verarbeitung sind mit „begin“ und „end process“ geklammert. In diesem Fall erfolgt für jeden Wert des Eingangsvektors A die Zuweisung eines Eingangssignals D(i) auf das Ausgangssignal.

Übung 1.10: Das folgende Beispiel beschreibt einen weiteren Logikbaustein in VHDL. Interpretieren Sie die Beschreibung und erstellen Sie ein Schaltdiagramm und eine Wertetabelle.
Hinweis: siehe Bild 1.9

```
--- LUT (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity LUT is
 port (      A : in std_logic_vector (3 downto 0);
             D : out std_logic_vector (1 downto 0));
end LUT;

architecture RTL of LUT is
```

```
        constant P: std_logic_vector (F downto 0) := "1001001100110100";
        constant Q: std_logic_vector (F downto 0) := "0010001001010001";

    begin
     process (A)
     begin
           case A is
                 when "0000" => D(0) <= P(0); D(1) <= Q(0);
                 when "0001" => D(0) <= P(1); D(1) <= Q(1);
                 when "0010" => D(0) <= P(2); D(1) <= Q(2);
                 when "0011" => D(0) <= P(3); D(1) <= Q(3);
                 when "0100" => D(0) <= P(4); D(1) <= Q(4);
                 when "0101" => D(0) <= P(5); D(1) <= Q(5);
                 when "0110" => D(0) <= P(6); D(1) <= Q(6);
                 when "0111" => D(0) <= P(7); D(1) <= Q(7);
                 when "1000" => D(0) <= P(8); D(1) <= Q(8);
                 when "1001" => D(0) <= P(9); D(1) <= Q(9);
                 when "1010" => D(0) <= P(A); D(1) <= Q(A);
                 when "1011" => D(0) <= P(B); D(1) <= Q(B);
                 when "1100" => D(0) <= P(C); D(1) <= Q(C);
                 when "1101" => D(0) <= P(D); D(1) <= Q(D);
                 when "1110" => D(0) <= P(E); D(1) <= Q(E);
                 when "1111" => D(0) <= P(F); D(1) <= Q(F);
           end case;
     end process;
    end RTL;
```

Übung 1.10: Beschreiben Sie eine Realisierung der Wertetabelle zu Bild 1.12 in VHDL. Verwenden Sie hierzu einen VHDL-Editor.

2. Taktsynchrone Logik

In diesem Abschnitt werden weitere Bausteine betrachtet, die für digitale Systeme von elementarer Bedeutung sind. Alle Bausteine werden exemplarisch in VHDL beschrieben. In der sogenannten Register Transfer Ebene werden Systeme als Mischung rein kombinatorischer Logik und taktsynchroner Logik beschrieben. Ziel der Beschreibung in VHDL ist schliesslich die Schaltungssynthese, wie sie beispielsweise für die Programmierung von FPGAs erforderlich ist.

2.1. Flip-Flops und Register

Wie der Name andeutet, sind Flip-Flops als bistabile Elemente in der Lage, einen Zustand zu speichern. Ja nach Art der Ansteuerung unterscheidet man folgende Typen: Ein RS-Flip-Flop besetzt einen Eingang zum Rücksetzen (R), sowie einen Eingang zum Setzen(S). Die Eingänge schliessen sich gegenseitig aus, d.h. gleichzeitiges Setzen und Rücksetzen sind nicht erlaubt.

Ein JK-Flip-Flop besitzt ausser den Eingangssignalen J (für Jump = Schalten) und K (für Kill = Ausschalten) einen Takteingang Clk (für Clock). Der Takteingang steuert die Signalübernahme in das Flop-Flop: bei ansteigender Taktflanke werden das J- bzw. K-Signal übernommen. Bei konstantem Pegel auf der Taktleitung bleiben die Eingänge inaktiv und der alte Zustand erhalten.

Eingänge		Ausgänge	
R	S	Q	Q'
0	0	Q_{alt}	Q'_{alt}
0	1	0	1
1	0	1	0
1	1	x	x

Eingänge			Ausgänge	
J	K	Clk	Q	Q'
0	0	^	Q_{alt}	Q'_{alt}
0	1	^	0	1
1	0	^	1	0
1	1	^	Q'_{alt}	Q_{alt}
x	x	1	Q_{alt}	Q'_{alt}
x	x	0	Q_{alt}	Q'_{alt}

Eingänge		Ausgänge	
D	Clk	Q	Q'
0	^	0	1
1	^	1	0
x	0	Q_{alt}	Q'_{alt}
x	1	Q_{alt}	Q'_{alt}

Bild 2.1 Flip-Flops

Das D-Flip-Flop lässt sich als vereinfachtes JK-Flip-Flop darstellen: Es erlaubt die Übernahme und das Halten eines Zustandes auf der Eingangsleitung D mit einer ansteigenden Flanke des Taktsignals. Das D-Flip-Flip ist somit auch elementarer Bestandteil eines Registers. Die Abbildung oben gibt die Typen von Flip-Flops zusammen mit ihren Wertetabellen wieder.

Folgende Abbildung zeigt den Aufbau eines 4-Bit Registers aus D-Flip-Flops. Hierzu werden die Taktleitungen zu allen Registerbausteinen synchronisiert. Bei steigender Taktflanke werden die Zustände der vier Eingänge D0 bis D3 übernommen.

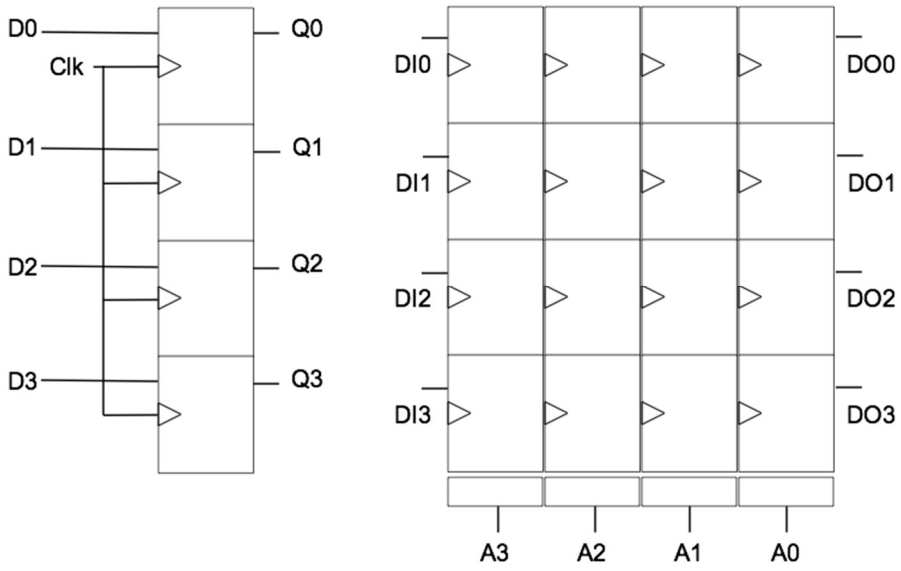


Bild 2.2 4-Bit-Register und Speicherzelle aus D-Flip-Flops aufgebaut

Mit mehreren solchen Registern lassen sich Speicherzellen aufbauen. Hierbei werden auch die Datenleitungen für die Eingänge bzw. Ausgänge zusammengefasst. Mit Hilfe zusätzlicher Adressleitungen wird jeweils eins der Register ausgewählt.

Mit taktsynchroner Logik sind Zeitbedingungen einzuhalten. Flip-Flops benötigen zum Setzen eine Zeit, in der das Eingangssignal D seinen Pegel bis zur ansteigenden Taktflanke halten muss (die sogenannte Setz-Zeit t_s, engl. set-up time). Auch nach der ansteigenden Taktflanke muss das Eingangssignal seinen Pegel noch eine Zeit halten, damit das Flip-Flop stabil schalten kann (die sogenannte Haltezeit t_h, engl. hold time). Schließlich benötigt das Flip-Flop von der ansteigenden Taktflanke an gemessen eine Zeit, bis der Ausgang Q seinen neuen Pegel erreicht, die sogenannte Schaltverzögerung (t_d für engl. delay time).

Übung 2.1: Stellen Sie die im letzten Abschnitt beschriebenen Zeiten in einem Zeitdiagramm für die Signale D, Clk und Q eines D-Flip-Flops dar.

```
--- D-Flip-Flop with rising edge clock (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity DFlipFlop is
 port (      Clk, D : in std_logic;
             Q : out std_logic);
end DFlipFlop;

architecture Behavioural of DFlipFlop is

begin
 process (Clk)
 begin
       if rising_edge (Clk) then
       Q <= D after 4 ns;
       end if;
 end process;
end Behavioural;
```

Übung 2.2: Interpretieren Sie die VHDL-Beschreibung eines D-Flip-Flops oben. Welche Begriffe sind neu? Welches Verhalten wird beschreiben? Hinweis: Sie können die Beschreibung auch von den Mustern auf der Web-Seite in den HDL-Editor übernehmen und erhalten so Informationen über Schlüsselworte und eigene Definitionen.

Der oben beschriebene VHDL-Code zielt auf die Beschreibung des Verhaltens. Für eine Synthese einer Schaltung beispielsweise auf einem FPGA kann man das beschriebene Zeitverhalten nicht aufprägen. Das tatsächliche Zeitverhalten ist dort abhängig von den Eigenschaften des Zielsystems. Für die Synthese würde man sich auf eine Beschreibung der Struktur beschränken, wie in folgendem Beispiel dargestellt.

```
--- D-Flip-Flop with raising edge clock (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity DFlipFlop is
 port (      Clk, D : in std_logic;
```

```
              Q : out std_logic);
end DFlipFlop;

architecture RTL of DFlipFlop is

begin
 process (Clk)
 begin
       if rising_edge (Clk) then
       Q <= D;
       end if;
 end process;
end RTL;
```

Übung 2.3: Interpretieren Sie die Beschreibung einen RS-Flip-Flops im folgenden VHDL-Beispiel. Was ist syntaktisch neu im Block „architecture“? Welche Anweisungen im „process“ Block werden sequentiell abgearbeitet, welche parallel?

```
--- RS-Flip-Flop RTL (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity RSFlipFlop is
 port (      R,    S : in std_logic;
                   Q : out std_logic);
end RSFlipFlop;

architecture RTL of RSFlipFlop is
 signal Qi: std_logic;   -- internal signal
begin
 process (R, S)
 begin
       if          R = '1' then Qi <= '0';
       elsif       S = '1' then Qi <= '1';
       else                     Qi <=  Qi;
       end if;
 end process;
 Q <= Qi;
end RTL;
```

Übung 2.4: Folgendes Beispiel beschreibt nochmals ein RS-Flip-Flop, wie man es durch Gatter aufbauen würde. Skizzieren Sie ein Schaltbild. Welche Anweisungen werden parallel ausgeführt? Welche Problematik birgt das RS-Flip-Flop? Würden Sie das RS-Flip-Flop zu den taktsynchronen Schaltungen rechnen?

```
--- RS-Flip-Flop with Logic Gates (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;
```

```
entity RSFlipFlop is
 port (      R,    S : in std_logic;
                   Q : out std_logic);
end RSFlipFlop;

architecture LogicGates of RSFlipFlop is
 signal Q1, Q2: std_logic;
begin
 Q1 <= S nor Q2;
 Q2 <= R nor Q1;
 Q  <= Q2;
end RTL;
```

2.2. Zähler

Mit Blick auf die Schaltungssynthese beispielsweise zur Programmierung eines FPGAs geht es in der VHDL Beschreibung gar nicht mehr darum, eine Ersatzschaltung nachzubilden, sondern das Verhalten so zu beschreiben, dass eine Synthese mit den vorhandenen Mitteln möglich ist. In diesem Sinne ist folgende Beschreibung eines 4-Bit Zählers zu verstehen: Es geht nicht um die Beschreibung einer Ersatzschaltung beispielsweise aus hinter einander geschalteten Flip-Flops, sondern nur um die abstrakte Beschreibung des gewünschten Verhaltens.

```
--- CTR16+ 4-Bit Counter (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CTR16 is
 port ( Clk, RS : in std_logic;                 -- clock, reset
        Q : out std_logic_vector (3 downto 0)); -- 4 bits out
end CTR16;

architecture RTL of CTR16 is
 signal Qin: std_logic_vector (3 downto 0);  -- internal counter

begin
 counter: process (Clk, RS)
 begin
       if (RS = '1') then
           Qin <= (others =>'0');          -- reset counter
             elsif (rising_edge(Clk))then
           Qin <= (Qin + 1);               -- increment counter
       end if;
 end process counter;

 Q <= Qin;                                 -- set output signal
end RTL;
```

Übung 2.5: Erläutern Sie die Eingangssignale, Ausgangssignale und die Funktion der Schaltung. Wie schaut die Wertetabelle in Abhängigkeit der Taktzyklen aus? Wozu dient das interne Signal Qin?

2.3. Schieberegister

Ein Schieberegister unterscheidet sich von einem Zähler dadurch, dass mit jedem Takt ein serielles Eingangssignal gespeichert wird. Das gespeicherte Signal steht hierbei innerhalb der Tiefe des Schieberegisters auch als paralleles Ausgangssignal zur Verfügung. Die folgende Beschreibung eines 8-Bit Schieberegisters in VHDL zeigt daher auch eine deutliche Ähnlichkeit mit dem Beispiel aus dem vorherigen Abschnitt.

```
--- ShiftRegister 8 bits (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity ShiftReg8 is
 port ( Clk, RS : in std_logic;               -- clock, reset
        D : in std_logic;                     -- serial input
        Q : out std_logic;                    -- 1 bit out
end ShiftReg8;

architecture RTL of ShiftReg8 is
 signal Qreg: std_logic_vector (7 downto 0);  -- internal signal
 begin
 process (Clk, RS)
        begin
        if (RS = '0') then
              Qreg <= (others =>'0');
        elsif rising_edge(Clk) then
              for i in 1 to 7 loop
                    Qreg(i-1) <= Qreg(i);
              end loop;
              Qreg(7) <= D;
        end if;
 end process;
 Q <= Qreg(0);
end RTL;
```

Übung 2.6: Klären Sie mit Hilfe des HDL-Editors neue syntaktische Elemente (z.B. die Zählschleife). Erläutern Sie die Eingangssignale, Ausgangssignale und die Funktion der Schaltung. Worin bestehen die Unterschiede zu Übung 2.5?

2.4. Taktsynchronisation

Bei der Beschreibung auf Register Transfer Ebene geht man davon aus, dass kombinatorische Logik und taktsynchrone Logik getrennt voneinander behandelt werden, d.h. in unterschiedlichen Prozessen. Wie in der folgenden Abbildung gezeigt, nimmt die getaktete Logik die Zustände der kombinatorischen Logik im Sinne einer Abtastung auf.

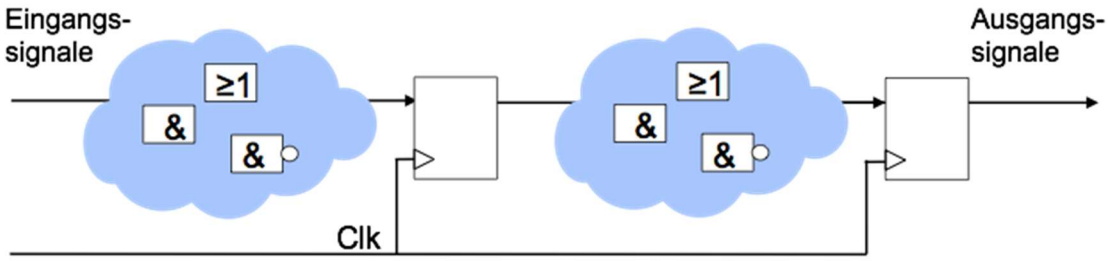


Bild 2.3 kombinatorische Logik und taktsynchrone Logik

Hierbei sollten Signaländerungen in einem vernünftigen Verhältnis zur Abtastrate stehen. Wenn das Signal zwischen 2 Pegeln pendelt, sollte die Grenzfrequenz des Signals im Sinne der Pegelwechsel höchstens die halbe Abtastrate betragen (Beispiel: wenn man das Signal zweimal pro Zeitintervall abtasten kann, sollte es in diesem Intervall den Pegel höchstens einmal wechseln).

Auch innerhalb taktsynchroner Logik kann mit unterschiedlichen Taktraten gearbeitet werden. An den Schnittstellen werden die Signalzustände dann zur Abtastung mit einer anderen Taktrate bereit gestellt, bzw. übergeben. Hierbei ist die nachfolgende Abtastung mit einer höheren Abtastrate immer möglich. Bei einer Übergabe an eine niedrigere Abtastrate muss die Grenzfrequenz des Signals wiederum in einem vernünftigen Verhältnis zur Abtastrate stehen.

Übung 2.7: Stellen Sie den Übergang zur Abtastung eines Signals in einem Zeitdiagramm dar. Erläutern Sie den Zusammenhang zwischen der maximalen Änderungsrate des Signals (bzw. der Grenzfrequenz) und hierzu minimal erforderlichen Abtastrate.

Arbeiten im Grenzbereich ist selten mit hoher Sicherheit verbunden. Bei der Kopplung zweier Taktbereiche kann man sicherheitshalber im Sinne der Redundanz mehrere Taktflanken verwenden. Hierdurch reduziert man die Wahrscheinlichkeit metastabiler Zustände der Flip-Flops an der Übergabestelle, die durch Verletzung der Zeitvorgaben der Flip-Flops zustande kommen können (Verletzung der Setz-Zeit bzw. Haltezeit, Einfluss der Schaltverzögerung). Die folgende Abbildung zeigt eine diesbezügliche Anordnung.

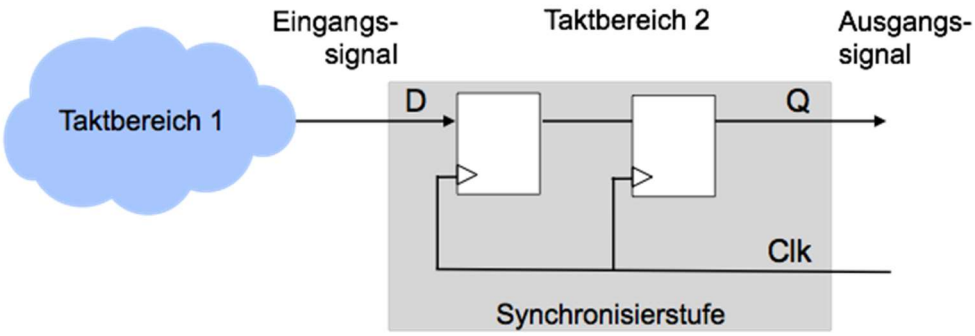


Bild 2.4 Synchronisierstufe

Übung 2.8: Erstellen Sie ein Zeitdiagramm für die Signale Clk, D und Q. Erläutern Sie die Zusammenhänge, das Funktionsprinzip und den Nutzen der Redundanz.

Die in der letzten Abbildung gezeigte Schaltung lässt sich in VHDL wie folgt beschreiben:

```
--- Clock Synchronisation (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity Synch is
 port ( Clk, RS : in std_logic;      -- clock, reset
             D : in std_logic;       -- input
             Q : out std_logic);     -- output
end Synch

architecture RTL of Synch is
 signal Qreg: std_logic_vector (1 downto 0);  -- internal signal
begin
 process (Clk, RS)
       begin
       if (RS = '0') then            -- active low
             Qreg <= (others =>'0');
       elsif rising_edge(Clk) then
                   Qreg(1) <= Qreg(0);
                   Qreg(0) <= D;
       end if;
 end process;
 Q <= Qreg(1);
end RTL;
```

Übung 2.9: Erläutern Sie die Eingangssignale, Ausgangssignale und die Funktion der Schaltung. Worin bestehen die Unterschiede zu Übung 2.6 (Schieberegister) bzgl. der technischen Lösung und bzgl. der jeweiligen Anforderungen?

Ein in der Nachrichtentechnik übliches Verfahren für die Übermittlung eines Signals ist die Verwendung von Quittungssignalen. Im einfachsten Fall wartet der Sender nach der Übermittlung einer Nachricht auf eine Quittung des Empfängers, bevor er die nächste Nachricht schickt. Auf diese Art lassen sich ebenfalls Taktbereiche miteinander koppeln.

Das folgende Beispiel beschreibt die Übergabe eines Signals S1 aus dem Taktbereich 1 an das Signal S2 im Taktbereich 2.

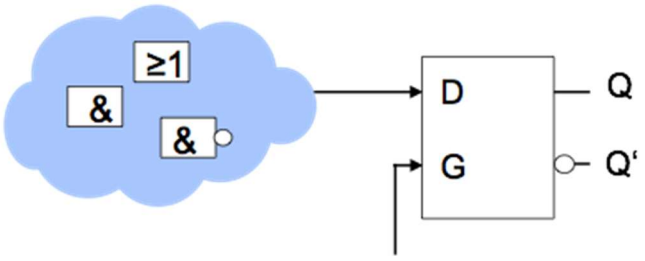
```
--- Synchronize signals by mutual acknowledge (VHDL)
library ieee;
use ieee.std_logic_1164.all;

entity AckSynch is
      port (      RS : in std_logic;   -- reset
                Clk1 : in std_logic;   -- clock domain 1
                Clk2 : in std_logic;   -- clock domain 2
                  S1 : in std_logic;   -- input signal domain 1
                  S2 : out std_logic;  -- signal to domain 2
                  Q2 : in std_logic;   -- acknowledge signal from domain 2
                  Q1 : out std_logic); -- acknowledge signal to domain 1
```

```
end AckSynch

architecture RTL of AckSynch is

      signal Si: std_logic;            -- internal signal
      signal Qi: std_logic;            -- internal signal

      component Synch
            port (Clk, Rst: in std_logic;
                  D: in std_logic;
                  Q: out std_logic);
      end Synch;

      begin
      DS1: Synch port map(Clk => Clk2, Rst => RS, D => S1, Q => Si);
      DS2: Synch port map(Clk => Clk1, Rst => RS, D => Q2, Q => Qi);
      S2 <= Si;
      Q1 <= Qi;
end RTL;
```

Übung 2.10: Klären Sie mit Hilfe des HDL-Editors neue syntaktische Elemente. Skizzieren sie die Zuordnung der äußeren und inneren Signale. Beachten Sie, dass S2 das aus S1 übertragene Signal in Bereich 2 ist, und Q1 die aus Q2 abgeleitete Quittung. Erläutern Sie die Funktion der Schaltung.

2.5. Übungen

Aus dem RS-Flip-Flop abgeleitet ist das Auffangregister (engl. latch wie Klinke bzw. einrasten). Da RS-Flip-Flops keine Takteingänge haben, eignen sie sich zum asynchronen Speichern von Signalen beispielsweise aus kombinatorischen Schaltungen. Neben dem Dateneingang D verfügt ein Auffangregister über einen Eingang zur Steuerung (E wie Enable, bzw. G wie Gate), wie in nachfolgender Abbildung gezeigt. In der transparenten Phase (G ='1') folgt das Latch dem Signaleingang. In der Haltephase (G='0') speichert es den letzten Zustand.



Eingänge		Ausgänge	
D	G	Q	Q'
0	1	0	1
1	1	1	0
x	0	Q_{alt}	Q'_{alt}

Bild 2.5 Auffangregister (Latch)

Übung 2.11: Erstellen Sie eine VHDL-Beschreibung für das Auffangregister.

Impulsgenerator: Mit Hilfe zweier D-Flip-Flops lässt sich eine Schaltung zur Erzeugung eines Signalimpulses erstellen. Hierzu werden Flip-Flops als Schieberegister geschaltet. Die Ausgänge werden logisch so miteinander verknüpft, dass ein Takt das Ausgangssignal schaltet, und der folgende Takt das Ausgangssignal wieder löscht. Es entsteht ein Impuls der Länge eines Taktzyklus. Voraussetzung hierfür ist, dass das Eingangssignal D gesetzt ist.

```
--- Pulse Generator made of two coupled D-Flip-Flops (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity PulseGen is
 port (  RS : in std_logic;    -- reset
        Clk : in std_logic;    -- clock
          D : in std_logic;    -- input signal "enable"
          Q : out std_logic;)  -- output signal „pulse“
end PulseGen;

architecture RTL of PulseGen is
signal Reg: std_logic_vector (1 downto 0);  -- internal register
begin
 process(RS, Clk)
 begin
 if (RS = '0') then Reg <= (others => '0');
 elsif rising_edge(Clk) then
       Reg(1) <= Reg(0);       -- shift register
       Reg(0) <= D;            -- load D to register
 end if;
 end process;
 Q <= not Reg(1) and Reg(0)          -- Out = Reg(1)' and Reg(0)
end RTL;
```

Übung 2.12: Erstellen Sie eine Skizze der Schaltung und das zugehörige Zeitdiagramm für den Impulsgenerator. Verwenden Sie die VHDL-Beschreibung als Hilfestellung.

Die folgende VHDL-Beschreibung ist ganz ähnlich aufgebaut wie das letzte Beispiel. Zweck der Schaltung ist das Erkennen von Änderungen im Signalpegel.

```
--- Detect Level Change in a signal (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity SignalDetection is
 port (      RS : in std_logic;      -- reset
            Clk : in std_logic;      -- clock
              D : in std_logic;
              Q : out std_logic;
            Clr : in std_logic;)
end SignalDetection;

architecture RTL of SignalDetection is
 signal Reg : std_logic_vector (1 downto 0);
 signal Edge: std_logic;
```

```
 begin
       process(RS, Clk)
       begin
             if (RS = '0') then
                   Reg  <= (others => '0');
                   Edge <= '0';
             elsif rising_edge(Clk) then
                   Reg(0) <= D;
                   Reg(1) <= Reg(0);
                   if (Clr = '1') then Edge <= '0';
                   elsif (Reg(0) <> Reg(1)) then Edge <= '1';
                   end if;
             end if;
       end process;
 Q <= Edge;
end RTL;
```

Übung 2.13: Analysieren Sie den VHDL-Code und beschreiben Sie die Funktion der Schaltung. Erstellen Sie ggf. ein Schaltbild bzw. eine Skizze des Aufbaus, wenn es der Übersichtlichkeit dient.

3. Testumgebung

Thema der beiden vorausgegangenen beiden Kapitel war der Schaltungsentwurf für kombinatorische Schaltungen bzw. taktsynchrone Schaltungen. Als Hilfsmittel wurde ein HDL-Editor eingesetzt. Der nächste Schritt im Entwurfsprozess ist der Test des Schaltungsentwurfs. Getestet wird hierbei, ob die Schaltung die gewünschte Funktion erfüllt. Es erfolgt eine funktionale Verifikation des Schaltungsentwurfs.

Wie lässt sich ein Schaltungsentwurf testen? Betrachtet man die Schaltung als System mit Eingängen und Ausgängen, so besteht die Funktion der Schaltung darin, zu jeder möglichen Kombination der Eingangswerte die gewünschten Ausgangswerte zu erzeugen. Bei einer kombinatorischen Schaltung kann die Verifikation also dadurch geschehen, dass man anhand der Wertetabelle prüft, ob für alle Kombinationen der Eingänge die gewünschten Ausgangswerte erzeugt werden. Bei taktsynchronen Schaltungen erfolgt der funktionale Test sinngemäß.

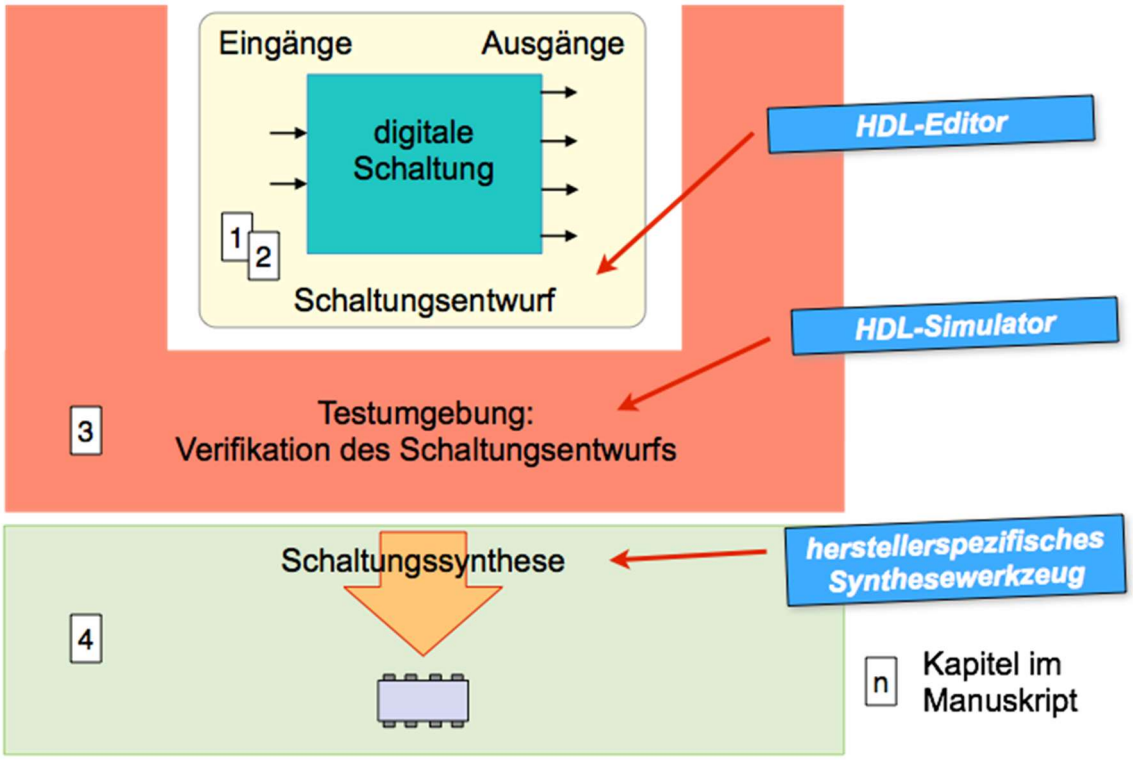


Bild 3.1 Übersicht - Entwurf digitaler Systeme

Für die Verifikation ist also eine Testumgebung zu erstellen, die den Schaltungsentwurf mit allen benötigten Kombinationen der Eingangswerte stimuliert und hierbei prüft, ob die gewünschten Ausgangswerte erzeugt werden. In diesem Fall betrachtet man den Schaltungsentwurf als geschlossenes System (engl. black box). Zusätzlich kann man auch den Entwurf des Innenlebens testen, indem man passende Stimuli zu ausgewählten internen Zuständen prüft. In diesem Fall öffnet man das geschlossene System, d.h. das Innenleben wird sichtbar (engl. white box tests).

Da die Schaltung mit Hilfe einer formalen Sprache beschrieben wurde (in HDL), kann man zum Entwurf der Testumgebung ebenfalls auf diese formale Beschreibung zurückgreifen, d.h. die Tests mit Hilfe eines HDL-Editors spezifizieren. Zur Ausführung der Tests wird dann ein Simulator benötigt, der das Verhalten der Schaltung abbildet. Abbildung 3.1 zeigt eine Übersicht über den Entwurfsprozess.

Der nächste Schritt im Anschluss an die Verifikation des Schaltungsentwurfs in der Testumgebung ist die Synthese der Schaltung beispielsweise auf einem FPGA. Hierzu wird dann ein herstellerspezifisches Synthesewerkzeug verwendet. Damit dieser nächste Schritt sich an die Tests nahtlos anschließt, ist es erforderlich, dass der Schaltungsentwurf mit diesem Werkzeug synthetisierbar ist. Es ist also zweckmäßig, das exemplarisch vorab zu prüfen. Die Testfälle müssen nicht synthetisierbar sein.

3.1. Zähler

Als erstes Testobjekt wird der 4-Bit-Zähler aus Abschnitt 2.2 verwendet. Der Übersichtlichkeit halber ist hier nochmals die HDL-Beschreibung der Schaltung aufgelistet.

```
--- CTR16+ 4-Bit Counter (VHDL)
---
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CTR16 is
 port ( Clk, RS : in std_logic;                 -- clock, reset
        Q : out std_logic_vector (3 downto 0)); -- 4 bits out
end CTR16;

architecture RTL of CTR16 is
 signal Qin: std_logic_vector (3 downto 0);  -- internal counter
begin
 counter: process (Clk, RS)
 begin
       if (RS = '1') then
           Qin <= (others =>'0');          -- reset counter
             elsif (rising_edge(Clk) and (Clk = '1'))then
           Qin <= (Qin + 1);               -- increment counter
       end if;
 end process counter;

 Q <= Qin;                                 -- set output signal
end RTL;
```

Übung 3.1: Welche Eingangssignale und Ausgangssignale hat die Schaltung? Welche Anweisungen werden parallel ausgeführt? Was sollte eine Testumgebung enthalten? Wie sollte ein Test ablaufen? Erstellen Sie ein Konzept für eine Testumgebung.

Folgender HDL-Text beschreibt eine mögliche Testumgebung.

```
--- CTR16+_T Testbench for 16-Bit Counter
---
library ieee;
use ieee.std_logic_1164.all;

entity test_CTR16 is    -- no external signals for testbench
end test_CTR16;

architecture Behavioural of test_CTR16 is

component CTR16 is
    port ( Clk, RS : in std_logic;                  -- clock, reset
     Q : out std_logic_vector (3 downto 0)); -- counter
end component;

-- testbench internal signals
signal T_Clk: std_logic := '0';
signal T_RS : std_logic := '0';
signal T_Q  : std_logic_vector (3 downto 0) := (others => '1');
```

```
begin

-- connect DUT to testbench
DUT: CTR16 port map (Clk => T_Clk, RS => T_RS, Q => T_Q);

-- run tests
reset : process                -- reset counter once
 begin
 wait for 5 ns; T_RS  <= '1';
 wait for 4 ns; T_RS  <= '0';
 wait;
end process reset;

count: process                 -- count forever
   begin
   T_Clk <= '0';
   wait for 10 ns;
   T_Clk  <= '1';
   wait for 10 ns;
end process count;

end Behavioural;
```

Übung 3.2: Beschreiben Sie den Testablauf. Wie ist der Prüfling gekennzeichnet? Wie wird der Prüfling in die Testumgebung eingebunden? Welche Anweisungen werden parallel ausgeführt?

Für die Durchführung der Tests auf dem HDL-Simulator werden beide Dateien auf den HDL-Simulator geladen: (1) der zu testende Schaltungsentwurf (als Prüfling, engl. auch kurz DUT für device under test), (2) die Testumgebung (engl. testbench). Beide Dateien werden in einer Projektdatei als zusammengehörig gekennzeichnet.

Für die Durchführung der Simulation sind weiterhin folgende Schritte nötig:

- Übersetzen (Compilierung) des Prüflings und seiner Testumgebung
- ggf. Behebung von Fehlern in den Dateien auf Basis der Fehlermeldungen des Compilers
- Ablauf (Run) der Simulation. Hierbei lassen sich die Eingangssignale und Ausgangssignale in einem Zeitdiagramm darstellen.

Übung 3.3: Installieren Sie den HDL-Simulator ModelSIM auf Ihrem Rechner. Folgen Sie hierzu den Instruktionen auf der Web-Seite des Projekts, siehe Literaturverzeichnis (4).

Übung 3.4: Starten Sie den HDL-Simulator. Unter dem Menüpunkt „Help“ finden sich unter „PDF-Documentation“ die Benutzerhandbücher. Machen Sie sich mit dem Dokument „Tutorial“ vertraut, bzw. kopieren Sie das Dokument an eine geeignete Stelle zum Nachschlagen außerhalb des Simulators.

Nach Start des Simulators ModelSIM erhalten Sie den unten abgebildeten Bildschirm. Legen Sie zunächst ein neues Projekt an, wie in der Abbildung gezeigt.

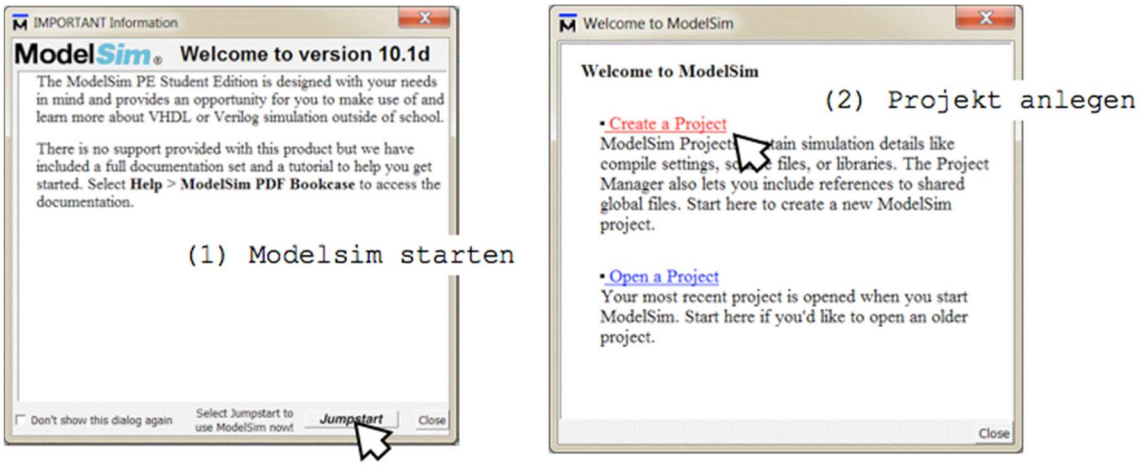


Bild 3.2 Simulator Starten und ein neues Projekt anlegen

Bei Anlegen des neuen Projektes geben Sie bitte einen willkürlichen Namen an. Die anderen Einstellungen lassen Sie bitte unverändert. Der Projektname dient dazu, die für die Schaltung und für die Testumgebung benötigten Dateien als zusammengehörig zu kennzeichnen. Wählen Sie als nächstes den Punkt „Add Existing File“ aus dem folgenden Menü.

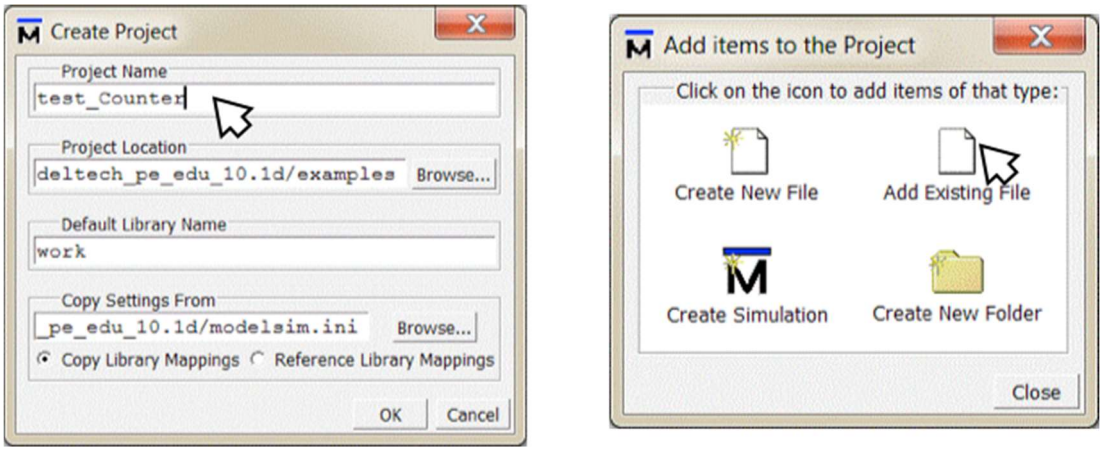


Bild 3.3 Projektname vergeben und Dateien hinzufügen (für Schaltung und Testumgebung)

Im Fenster „Add file to Project“ suchen Sie bitte per „Browse“ nach dem Verzeichnis, in dem sich Ihre Schaltungsentwürfe befinden. Das Verzeichnis bleibt als Pfadangabe erhalten, es wird keine Kopie der VHDL-Datei erstellt. Auf diese Weise finden erforderliche Änderungen bei der Simulation gleich im Quelltext statt. Nach Hinzufügen des Schaltungsentwurfs wählen Sie nochmals „Add existing File“ und ergänzen die Testumgebung zum Projekt.

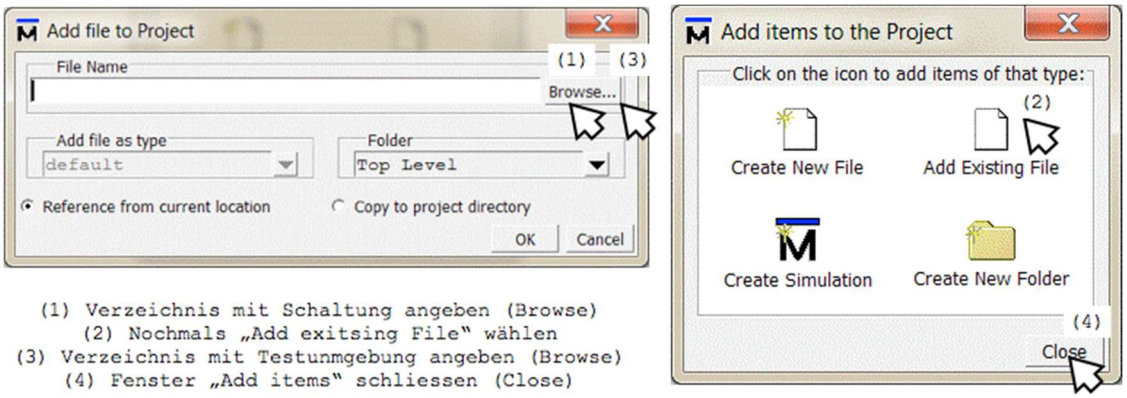


Bild 3.4 Hinzufügen der Dateien mit Pfadangaben für Schaltungsentwurf und Testumgebung

Der Simulator startet nun mit einem Projektfenster (Reiter „Project" unten), das die beiden ausgewählten Dateien zeigt. Das Projekt ist nun angelegt. Die Dateien lassen sich nun für die Simulation übersetzen. Wählen Sie dazu das Schaltfeld „Compile" und weiter die Option „Compile all".

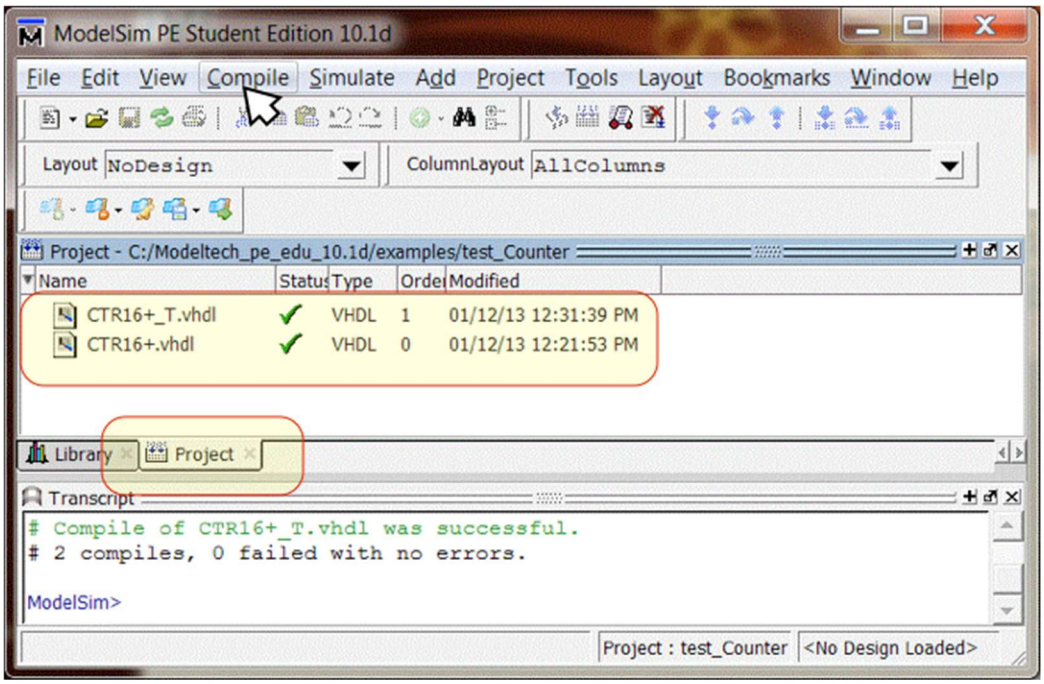


Bild 3.5 Projekt angelegt und bereit zum Übersetzen

Nach erfolgreicher Übersetzung (siehe Meldung unten in der letzten Abbildung „Compile of ... successful") sind Sie nun bereit die Simulation zu starten. Wählen Sie hierzu das Schaltfeld „Simulate" mit der Option „Start Simulation". Sie werden dann aufgefordert, die Zieldatei auszuwählen, wie in folgender Abbildung gezeigt.

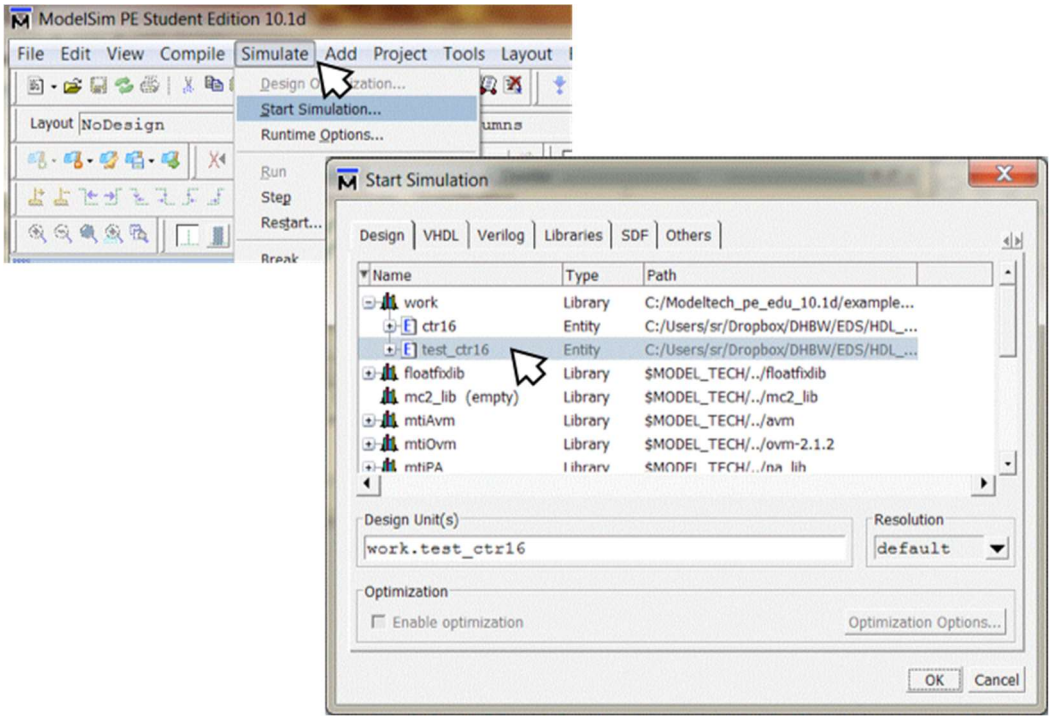


Bild 3.6 Start der Simulation

Der HDL-Simulator wechselt nun in der Simulationsperspektive, wie in der folgenden Abbildung gezeigt. In dieser Perspektive erhalten Sie nun [1] das Simulatorfenster mit der Struktur der Testumgebung, [2] die aktiven Prozesse aus dem Prüfling und aus der Prozessumgebung (Bemerkung: in HDL werden diese Prozesse parallel ausgeführt), [3] unter „Objects“ die zur Verfügung stehenden Signale und [4] das Fenster mit dem Zeitdiagramm.

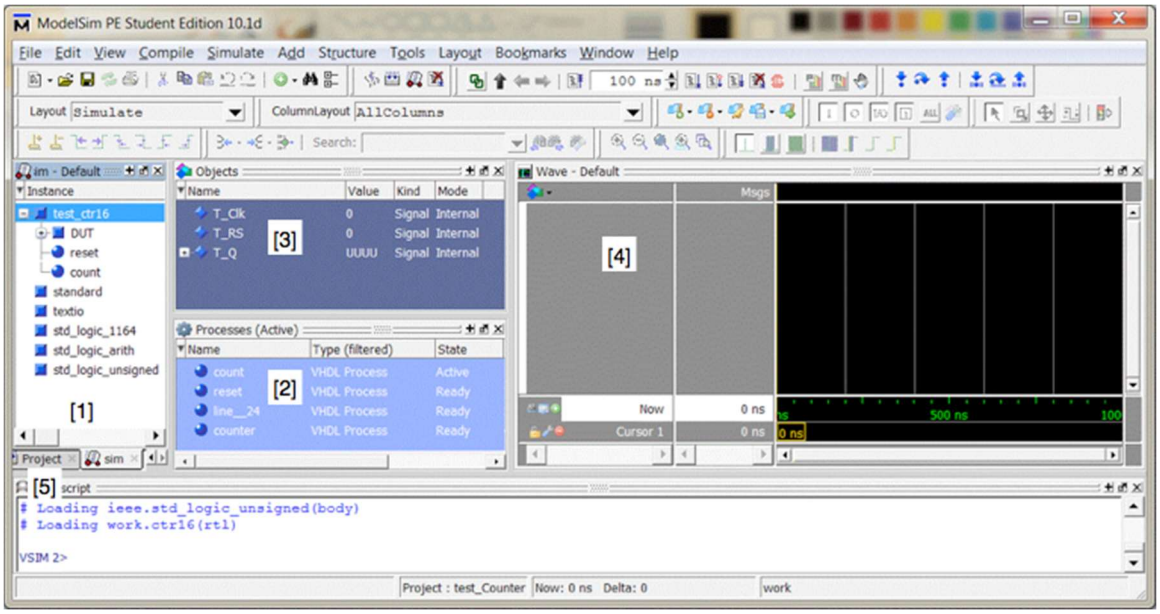


Bild 3.7 Simulationsperspekive des HDL-Simulators

Außerdem findet sich unter dem Reiter „Project“ in [5] noch das Projektverzeichnis mit der Referenz zu den beiden Quelldateien für die Schaltung und die Testumgebung. Über diesen Reiter lässt

sich jederzeit auf die Dateien zugreifen. Als nächste bringen Sie bitte die Signale aus [3] in Beziehung zum Zeitdiagramm [4] im Fenster „Wave“.

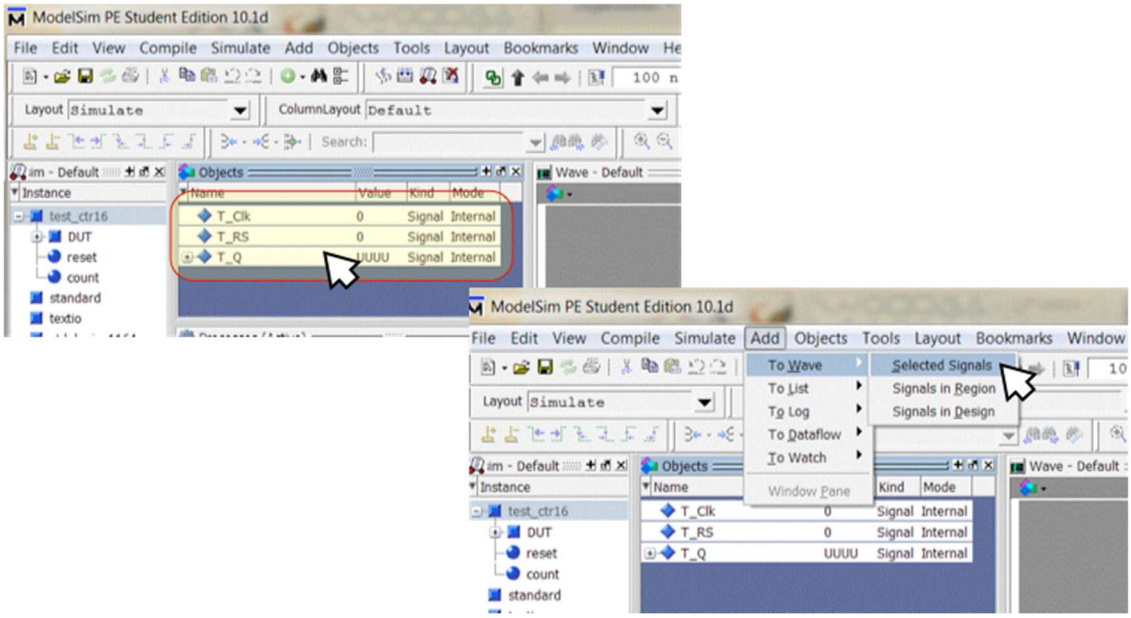


Bild 3.8 Auswahl der Signale für das Zeitdiagramm im Fenster „Wave“

Selektieren Sie hierzu die Signale im Fenster [3] und wählen Sie mit der Maus die Schaltfläche „Add“ mit den Optionen „To Wave“ und weiter „Selected Signals“, wie in der letzen Abbildung gezeigt. Die Signale werden nun mit dem Zeitdiagramm in Verbindung gebracht. Die Simulatorperspektive sollte nun wie folgt aussehen.

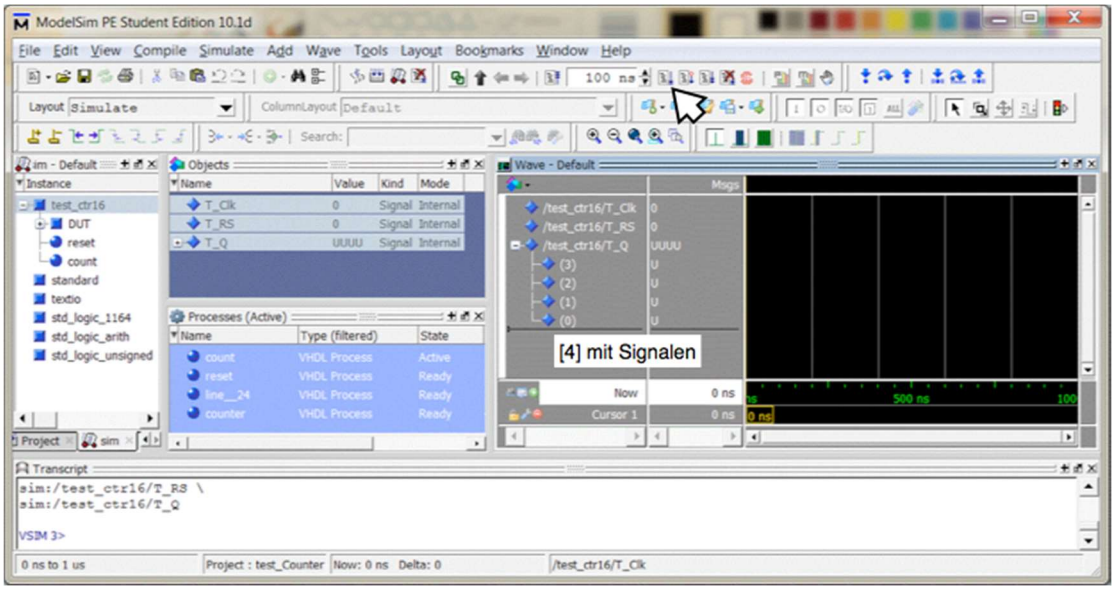


Bild 3.9 Bereit zum Ablauf der Simulation

Der Ablauf der Simulation lässt sich nun mit dem in der Abbildung gezeigten Schaltfeld „Run (F9)“ schrittweise starten. Parameter wie die Schrittweite lassen sich natürlich verändern. Das Ergebnis des Simulationslaufs zeigt die folgende Abbildung.

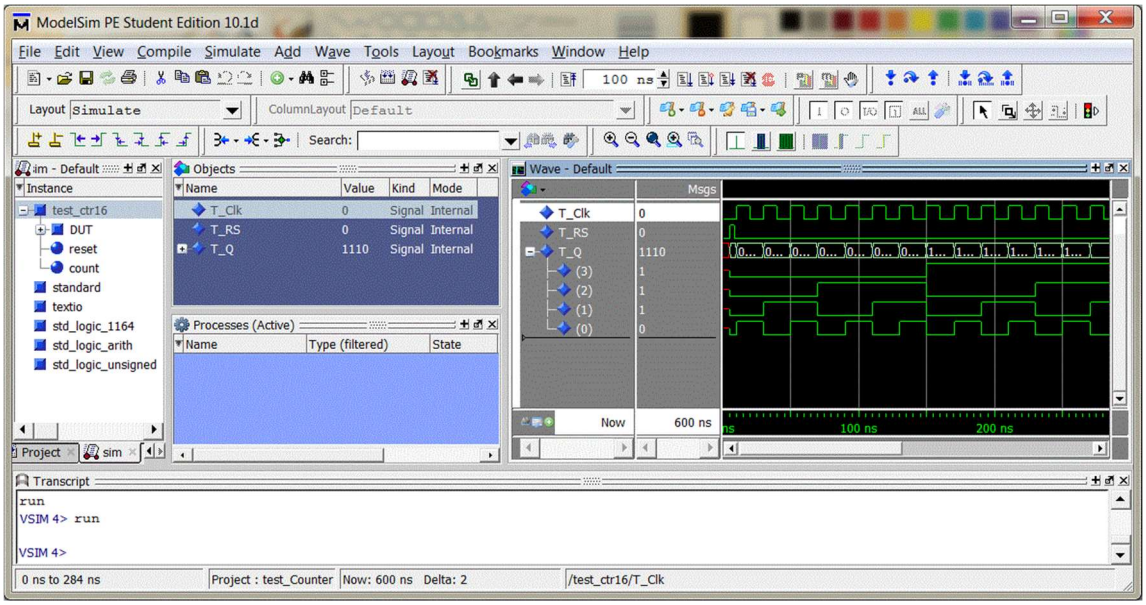


Bild 3.10 Ergebnisse der Simulation

Im Zeitdiagramm erkennt man oben das Taktsignal. Die anfangs undefinierten Signalpegel (rot markiert, bzw. vor Start der Simulation mit dem Zustand „U“ gekennzeichnet) des Ausgangs sind nach den Reset-Impuls in der zweiten Zeile definiert. Darunter erkennt man die Stufen des Zählers, der mit jeder ansteigenden Taktflanke weiter zählt. Zum Abschluss der Simulation beenden Sie bitte den Simulator durch Klicken der Schaltfläche „Simulate“ mit der Option „Break“ bzw. „End Simulation“.

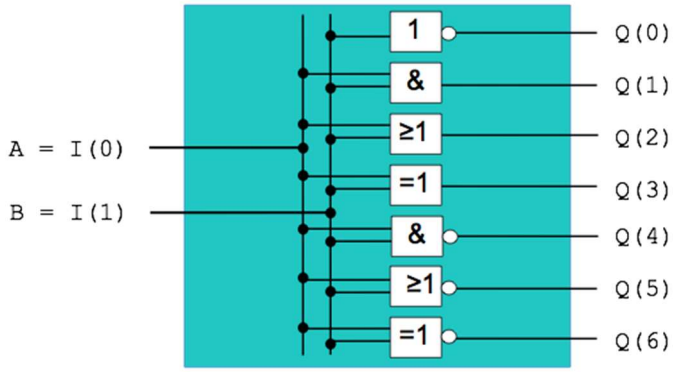
3.2. Logik Gatter

Um eine die Testumgebung näher zu spezifizieren, werden die Logik Gatter aus Abschnitt 1.5 nochmals bemüht. Das aus den Gattern bestehende System hat zwei Eingänge und insgesamt 7 Ausgänge und wird durch folgende Wertetabelle beschrieben.

```
--- Logic Gates (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity LogicGates is
 port (A, B : in std_logic;
        Qnot  : out std_logic;
        Qand  : out std_logic;
        Qor   : out std_logic;
        Qxor  : out std_logic;
        Qnand : out std_logic;
        Qnor  : out std_logic;
        Qxnor : out std_logic);
end LogicGates;

architecture RTL of LogicGates is
```

```
    begin
            Qnot  <=   not   B;
            Qand  <= A and   B;
            Qor   <= A or    B;
            Qxor  <= A xor   B;
            Qnand <= A nand  B;
            Qnor  <= A nor   B;
            Qxnor <= A xnor  B;
    end RTL;
```

Wenn man die Wertetabelle so sortiert, dass man alle Eingangssignale und Ausgangssignale als Testvektoren zusammenfasst, so besteht der Test darin, die Testvektoren sukzessive durch zu schalten und das Ergebnis mit den erwarteten Ausgängen der Testvektoren zu vergleichen. Der letzte Testvektor ist hierbei identisch mit dem ersten und dient dazu, alle Zustandsübergänge zu schalten, wobei mit dem Übergang in den letzten Testvektor wieder der Anfangszustand erreicht wird.



Testvektor	Signale								
	A	B	NOTB	AND	OR	XOR	NAND	NOR	XNOR
Nr.	I(0)	I(1)	Q(0)	Q(1)	Q(2)	Q(3)	Q(4)	Q(5)	Q(6)
1	0	0	1	0	0	0	1	1	1
2	1	0	1	0	1	1	1	0	0
3	0	1	0	0	1	1	1	0	0
4	1	1	0	1	1	0	0	0	1
5	0	0	1	0	0	0	1	1	1

Bild 3.11 Wertetabelle als Testvektoren

Die Testvektoren lassen sich in der Testumgebung als Datenstrukturen anlegen. Diese Vorgehensweise lässt sich generell für Tests so übernehmen. Der Ablauf der Tests verläuft wie folgt.

```
    --- Testbench for Logic Gates (VHDL)
    ---
    library ieee;
    use ieee.std_logic_1164.all;

    entity test_LogicGates is   -- no external signals
    end test_LogicGates;

    architecture Behavioural of test_LogicGates is

    component LogicGates is
```

```
 port (A, B : in std_logic;
       Qnot  : out std_logic;
       Qand  : out std_logic;
       Qor   : out std_logic;
       Qxor  : out std_logic;
       Qnand : out std_logic;
       Qnor  : out std_logic;
       Qxnor : out std_logic);
end component;

-- testbench internal signals
signal clk : std_logic := '0';
signal I   : std_logic_vector (0 to 1);
signal Q   : std_logic_vector (0 to 6);

-- testbench test vectors
type test_rec is record
  I : std_logic_vector (0 to 1);
  Q : std_logic_vector (0 to 6);
end record;

type test_arr is array(positive range <>) of test_rec;

constant test_vector : test_arr := (
  ("00", "1000111"), ("10", "1011100"), ("01", "0011100"),
  ("11", "0110001"), ("00", "1000111"));

begin

-- connect DUT to testbench
DUT: LogicGates port map (A=>I(0), B=>I(1), Qnot=>Q(0), Qand=>Q(1),
Qor=>Q(2), Qxor=> Q(3), Qnand=>Q(4), Qnor=>Q(5), Qxnor=>Q(6) );

-- run tests
test: process
begin
 L1: for j in 1 to 5 loop     -- step to next transition

   I <= test_vector(j).I; -- set input signal to test vector
   clk <= '0';
   wait for 50 ns;
   clk <= '1';
   wait for 50 ns;

   if (Q /= test_vector(j).Q ) then     -- compare output signals
       report "test step failed" severity NOTE;
   else
       report "test step passed" severity NOTE;
   end if;
 end loop L1;
```

```
    wait;
    end process test;

    end Behavioural;
```

Übung 3.5: Analysieren Sie die Testumgebung. Wie ist der Ablauf? Welche neuen Konstrukte gibt es? Wie werden Eingangssignale und Ausgangssignale für den Test verarbeitet? Wie funktioniert die Schleife für das Schalten der Transitionen zwischen den Vektoren? Wie werden Eingangssignale und Ausgangssignale in den Testvektoren indiziert?

Übung 3.6: Geben Sie beide Dateien in ein Projekt auf dem HDL-Simulator und führen Sie eine Simulation durch. Wo finden sich die Test Reports? In der Schaltung greifen alle Logik-Gatter parallel auf die Eingangssignale zu. Wie äußert sich das im Simulator (siehe aktive Prozesse)?

Im HDL-Simulator sollte sich folgendes Ergebnis zeigen. Hinweis: Im Verzeichnis „Project“ links neben dem Simulatorfenster Mitte links finden Sich die Pfade mit den beiden VHDL-Dateien. Hier kann man direkt Änderungen im Quelltext vornehmen.

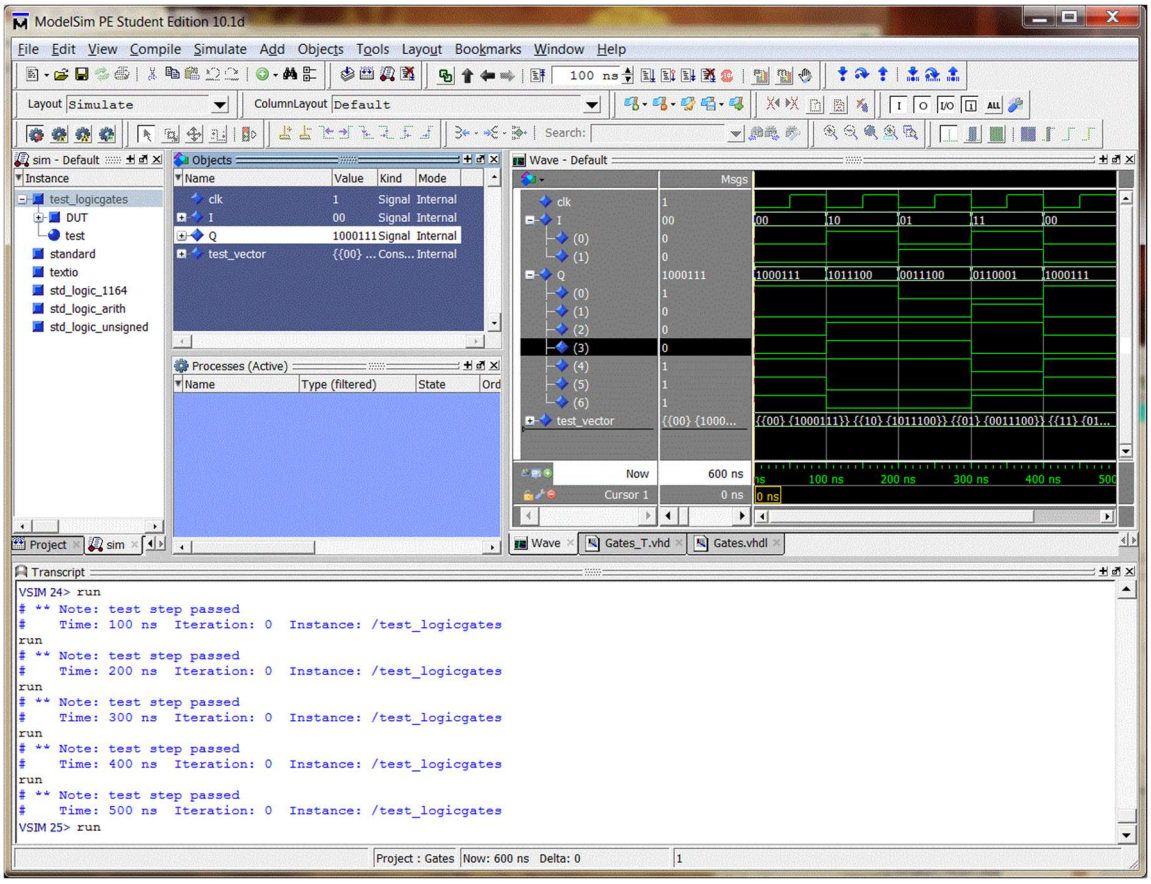


Bild 3.12 Test der Logik Gatter

3.3 RS-Flip-Flop

Als weiteres Beispiel dient das RS-Flip-Flop aus Abschnitt 2.1, Übung 2.3. Die Beschreibung der Schaltung ist hier nochmals aufgeführt.

```
--- RS-Flip-Flop RTL (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity RSFlipFlop is
 port (      R,    S : in std_logic;
                   Q : out std_logic);
end RSFlipFlop;

architecture RTL of RSFlipFlop is
 signal Qi: std_logic;
begin
 process (R, S)
 begin
       if          R = '1' then Qi <= '0';
       elsif       S = '1' then Qi <= '1';
       else        Qi <=  Qi;
       end if;
 end process;
 Q <= Qi;
end RTL;
```

Für die Testumgebung wird auf das Verfahren aus dem letzten Beispiel zurückgegriffen. Die folgende Auflistung zeigt, dass sich das Verfahren ohne Probleme übertragen lässt. Der Testablauf ist nahezu unverändert, nur die Länge der Vektoren und die Wertetabelle wurden angepasst.

```
--- RS-Flip-Flop Testbench (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity test_RSFlipFlop is
end test_RSFlipFlop;

architecture Behavioural of test_RSFlipFlop is

component RSFlipFlop is
 port (      R,    S : in std_logic;
                   Q : out std_logic);
end component;

-- testbench internal signals
signal clk : std_logic := '0';
```

```
    signal I   : std_logic_vector (0 to 1);
    signal Q   : std_logic_vector (0 to 0);

    -- testbench test vectors
    type test_rec is record
      I : std_logic_vector (0 to 1);
      Q : std_logic_vector (0 to 0);
    end record;

    type test_arr is array(positive range <>) of test_rec;

    constant test_vector : test_arr := (
    ("00", "0"), ("01", "0"), ("10", "1"),("11", "0"), ("00", "0"));

    begin
    -- connect DUT to testbench
    DUT: RSFlipFlop port map (S=>I(0), R=>I(1), Q=>Q(0) );

    -- run tests
    test: process
     begin
     L1: for j in 1 to 5 loop -- step to next transition

     I <= test_vector(j).I; -- set input signal to test vector
           clk <= '0';
           wait for 50 ns;
           clk <= '1';
           wait for 50 ns;

     if (Q /= test_vector(j).Q ) then     -- compare output signals
                 report "test step failed" severity NOTE;
     else
                 report "test step passed" severity NOTE;
           end if;

     end loop L1;
     wait;
    end process test;

    end Behavioural;
```

Die Durchführung der Simulation zeigt, dass nach Fehlermeldungen durch die eingangs undefinierten Signale die Schaltung nach dem ersten Schaltzyklus wie erwartet funktioniert. Der Simulator verarbeitet also nicht nur die Schaltpegel, sondern auch undefinierte Zustände.

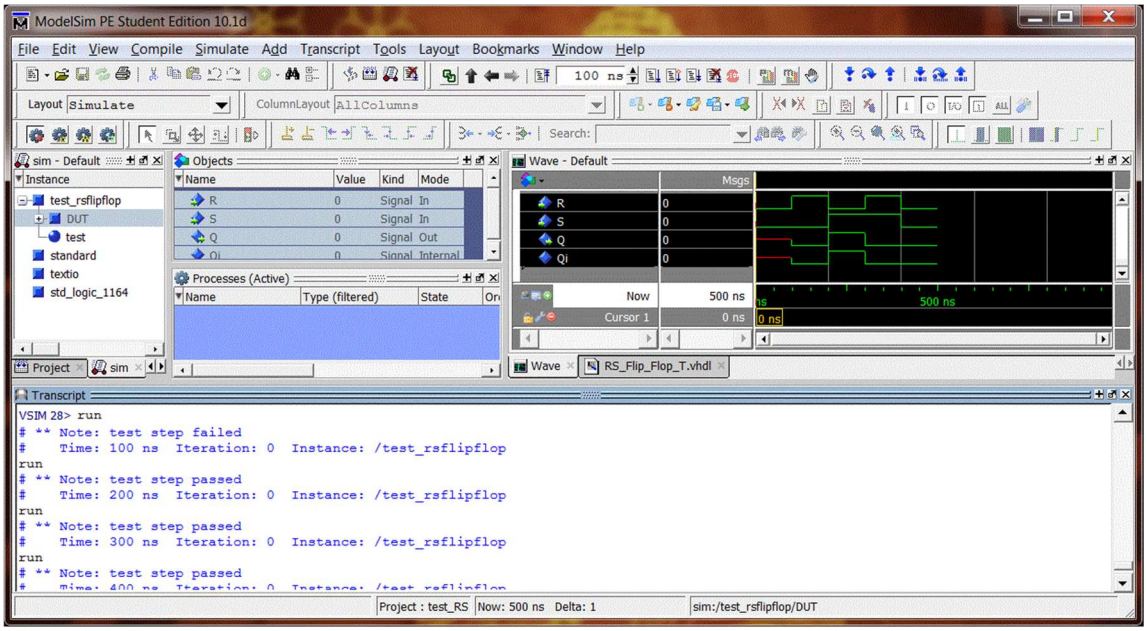


Bild 3.13 Testumgebung für RS-Flip-Flop

3.4. Multiplexer

In Übung 1.6 wurde ein 8-zu-1 Multiplexer als Schaltung entworfen.

Übung 3.7: Erstellen Sie eine Testumgebung für den Multiplexer. Übung 3.9: Testen Sie den Schaltungsentwurf mit Hilfe der Testumgebung im Simulator. Fassen Sie die Ergebnisse zusammen.

3.5. Schieberegister

In Abschnitt 2.3 wurde ein Schieberegister entworfen.

Übung 3.9: Erstellen Sie eine Testumgebung für das Schieberegister. Übung 3.10: Testen Sie den Schaltungsentwurf mit Hilfe der Testumgebung im Simulator. Fassen Sie die Ergebnisse zusammen.

4. Schaltungssynthese

Im Anschluss an den Schaltungsentwurf und die funktionale Verifikation des Entwurfs in einer Testumgebung im Simulator kann die Synthese der Schaltung zur Implementierung auf einem programmierbaren Baustein erfolgen. Beim Schaltungsentwurf und in der Verifikation betrachtet man die Schaltung auf einem höheren Abstraktionsniveau, beispielsweise als Struktur auf der sogenannten Register Transfer Ebene (engl. Register Transfer Level, oder kurz RTL).

Bei der Synthese erfolgt denn maschinell die Übersetzung auf eine niedrigere strukturelle Ebene: statt Register und Logikbausteine wird übersetzt in die Ebene der Gatter, die zu sogenannten Netzlisten verschaltet sind. Beide strukturellen Ebene sind sinngemäss vergleichbar mit der Erstellung und Tests eines Computerprogramms mit einer höheren Programmiersprache wie C, C++ oder Java. Durch den

Compiler wird dieser Kode übersetzt in ein maschinennäheres Format. Grundsätzlich kann man Programme natürlich auch maschinennah erstellen, beispielsweise in einer Assemblersprache.

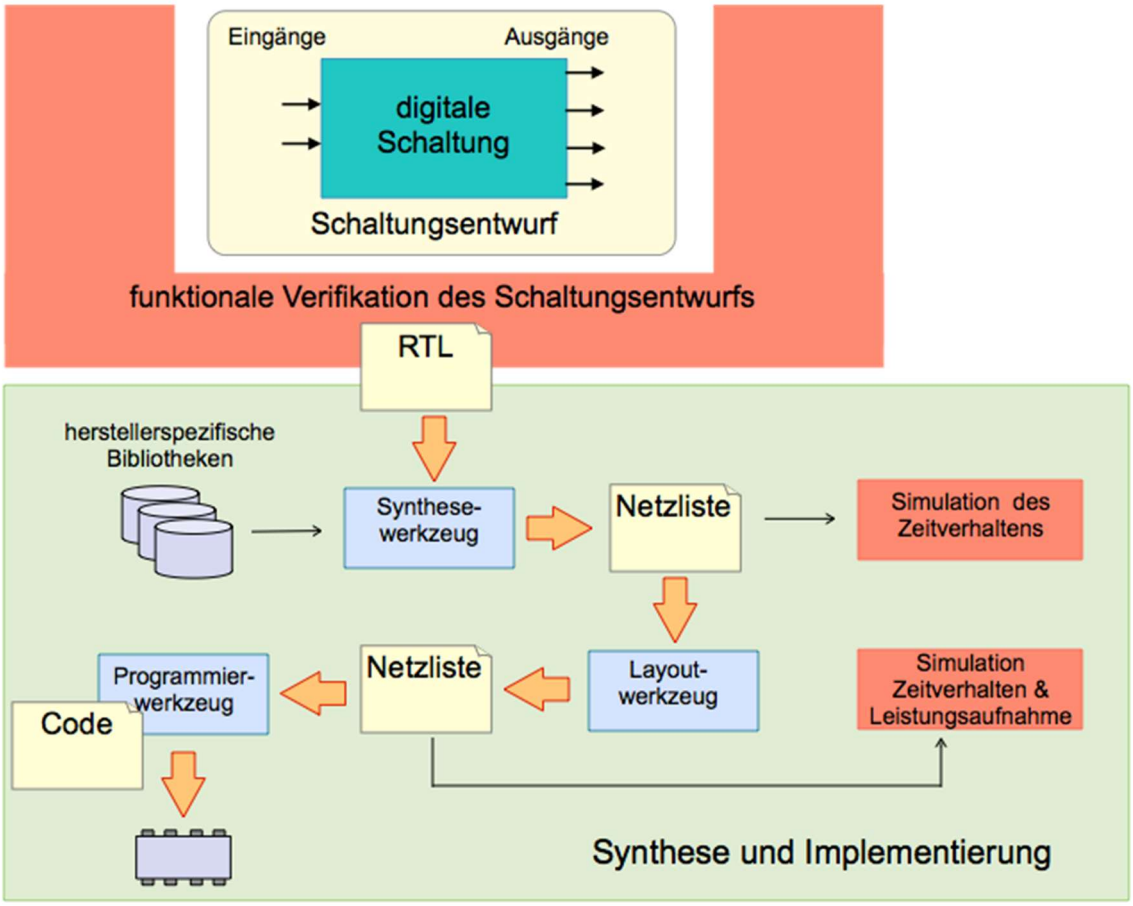


Bild 5.1 Synthese und Implementierung des Schaltungsentwurfs

Ebenso sind Netzlisten ebenfalls mit Programmierwerkzeugen zugänglich, speziell wenn sie ebenfalls in einem HDL-Format notiert sind, wie beispielsweise VHDL. Allerdings wird diese Dar-stellung schnell sehr unübersichtlich. Insgesamt gehören zur Synthese und zur Implementierung einer auf Register Transfer Ebene beschriebenen Schaltung folgende Schritte:

- Erzeugung der Netzlisten aus der HDL-Beschreibung auf RTL Ebene mit Hilfe eines Syntheseprogramms.
- Gegebenenfalls eine Simulation des Zeitverhaltens der nun auf Gatter-Ebene (als Netz-liste) verfügbaren Schaltung im Simulator. Das Zeitverhalten konnte in der funktionalen Verifikation noch nicht berücksichtigt werden, da dieses vom Zielbaustein abhängt.
- Layout (Platzierung und Routing) für den beabsichtigten programmierbaren Baustein (beispielsweise ein FPGA). Hier werden beispielsweise die Eingabe- und Ausgabepins des Bausteins der Schaltung zugeordnet. Ergebnis ist (1) einerseits eine Netzliste mit Annotationen über die beabsichtigte Implementierung. Diese Netzliste kann für weitere Simulationen verwendet werden, beispielsweise zum Zeitverhalten bzw. über die voraus-sichtliche Leistungsaufnahme. Ergebnis ist (2) weiterhin ein Datenformat, das als Programmdatei in den Baustein geladen werden kann.
- Programmierung des Bausteins durch Laden der Programmdatei auf den Baustein.
- Tests der fertigen Implementierung auf dem Baustein.

Alle Schritte bereits ab der Synthese der Schaltung ausgehend von deren Beschreibung auf RTL Ebene sind herstellerspezifisch. Der Grund hierfür ist, dass bei der Erstellung der Netzlisten die tatsächlich auf dem beabsichtigten Zielbaustein verfügbaren Logikelemente und Gegebenheiten berücksichtigt werden müssen. Hierzu werden bereits bei der Synthese herstellerspezifische Bibliotheken eingebunden, wie in der Abbildung gezeigt.

Diese Vorlesung beschränkt sich auf alle herstellerunabhängigen Methoden und Schritte bei der Konzeption und beim Test eines digitalen Systems. Aus diesem Grund wird hier nicht näher auf die Schaltungssynthese und Implementierung eingegangen. In der Vorlesung enthalten sind allerdings die Nutzung eines HDL-Editors und die Nutzung eines HDL-Simulators zum Schaltungsentwurf und zur funktionalen Verifikation des Entwurfs. Die Anwendung dieser Werkzeuge zusammen mit einem Synthesewerkzeug und herstellerabhängigen Programmierwerkzeugen ist Bestandteil des Labors zur Vorlesung.

5. Zustandsautomaten

Die Aufgaben einer digitalen Schaltung lassen sich bei komplexeren Systemen aufteilen in die Datenverarbeitung, die beispielsweise parallel ausgeführt werden kann, und in die Steuerlogik, die die Verarbeitung in mehreren Schritten abhängig von Zustand des Systems schalten kann. Die Steuerung ist hierbei in aller Regel ein sequenziell arbeitendes Schaltwerk.

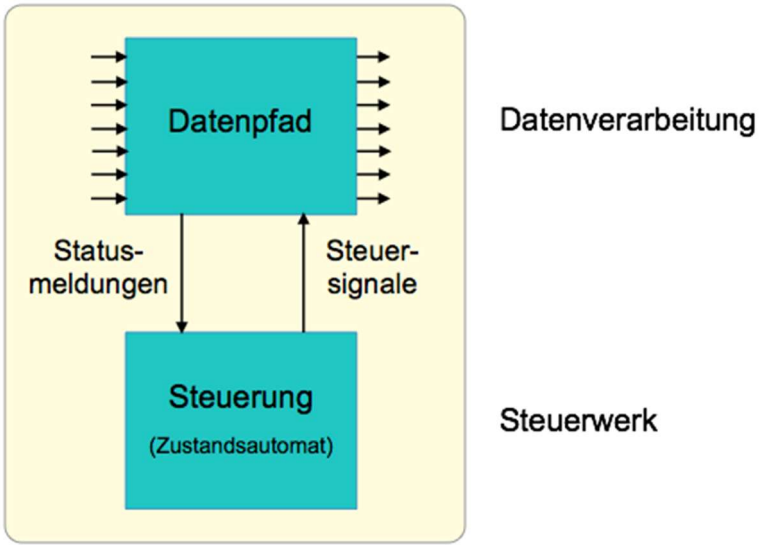


Bild 5.1 Aufteilung von Datenverarbeitung und Steuerung

Wie in der Abbildung gezeigt, erhält die Steuerung Statusinformationen aus dem Datenpfad. In Abhängigkeit vom Zustand des Systems greift die Steuerung in die Datenverarbeitung mittels Steuersignalen ein. Eine solche Steuerung funktioniert als Zustandsautomat (engl. Finite State Machine, da die Anzahl der Zustände in aller Regel begrenzt ist).

Im einfachsten Fall werden die Zustände nach Start des Automaten zyklisch durchlaufen (z.B. bei einem Waschprogramm). Im allgemeinen Fall hängen die Zustände von externen Einflüssen ab (z.B. bei einem Getränkeautomaten in Abhängigkeit von der Wahl des Benutzers, vom eingeworfenen Betrag in Münzen, bzw. bei Abbruch durch den Benutzer).

Durch Definition von Zuständen lässt sich der Ablauf einer Steuerung recht zielstrebig beschreiben. Ein Getränkeautomat durchläuft beispielsweise die in der folgenden Abbildung gezeigten Zustände. Transitionen zwischen den Zuständen werden bei diesem Beispiel durch den Benutzer ausgelöst. Alternativ kann der Fortschritt im Bearbeitungsprozess bzw. der Ablauf definierter Zeiten Zustandswechsel auslösen.

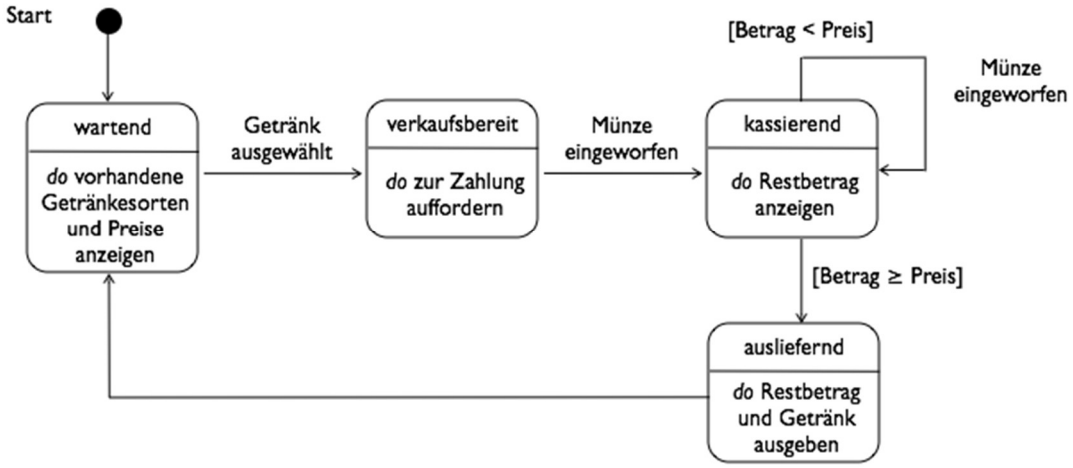


Bild 5.2 Beispiel für den Ablauf eines Zustandsautomaten

Übung 5.1: Interpretieren Sie den in der Abbildung gezeigten Ablauf. Identifizieren Sie Zustände und Zustandsübergänge. Welche Aktivitäten werden in Abhängigkeit des jeweiligen Zustands ausgeführt? Welchen Vorteil bietet diese Betrachtungsweise? Wo gibt es Bedingungen für Zustandsübergänge? Wie könnte man den Ablauf um fehlende Funktionen ergänzen (beispielsweise Abbruch oder Time-Out)? Wie könnte man den Ablauf weiter detaillieren?

Bei der Realisierung des Zustandsautomaten durch ein Schaltwerk sind also folgende Aufgaben zu lösen: (1) die Speicherung des aktuellen Zustands in einem Zustandsregister, (2) die Logik für Zustandsübergänge in Abhängigkeit des Prozesses und der definierten Bedingungen. Somit ergibt sich für einen Zustandsautomaten die in folgender Abbildung gezeigte Struktur.

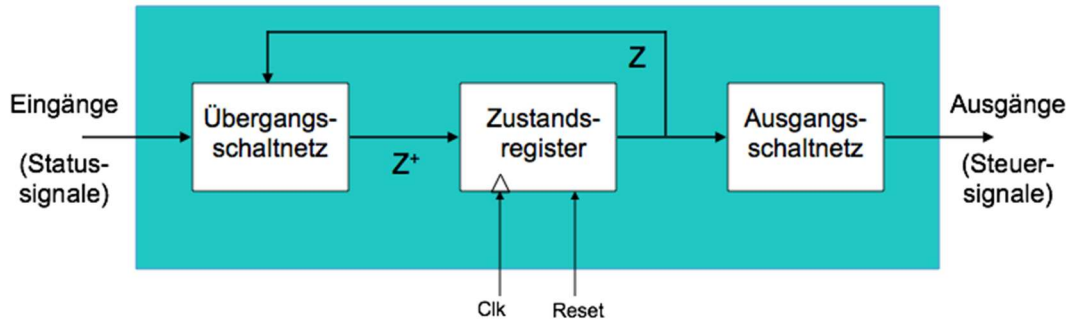


Bild 5.3 Struktur des Zustandsautomaten

Die Eingangssignale aus dem Datenpfad werden in einem Übergangsschaltnetz verarbeitet und erzeugen den Folgezustand Z^+ des Automaten. Dieser Folgezustand wird bei Anliegen eines Taktes ins Zustandsregister übernommen (Signal Clk am Takteingang). Der im Zustandsregister aktuell gespeicherte Zustand Z wirkt über ein Ausgangsschaltnetz auf die Steuersignale im Prozess. Zugleich hat der aktuelle Zustand Z Einfluss auf die Verarbeitung der Eingangssignale aus dem Prozess (über das Übergangsschaltnetz).

Diese letztgenannte Rückkopplung erklärt auch die Bezeichnung „Übergangsschaltnetz“: Da der Automat in Abhängigkeit seines aktuellen Zustandes reagiert, erfolgt auch die Bearbeitung der Eingangssignale zustandsabhängig. Der als Beispiel genannte Getränkeautomat liefert das gewünschte Getränk erst aus, nachdem der hierfür erforderliche Betrag eingezahlt wurde. Sofern sich der Zustand

gegenüber dem letzten Taktsignal nicht verändert hat, wird der unveränderte Zustand ins Zustandsregister übernommen.

Damit der Automat sinnvoll funktionieren kann, muss er nach dem Einschalten von einem initialen Zustand aus starten. Hierfür und ggf. zur Fehlerbehebung ist der Reset-Eingang am Zustandsregister vorgesehen. Bei der in der Abbildung gezeigten Realisierung gibt es keinen direkten Einfluss der Eingangssignale auf die Ausgangssignale. Eine solche direkte Verknüpfung unabhängig vom Zustand des Automaten wäre in dieser Realisierung innerhalb des Datenpfades abzubilden.

5.1. Zustandsdiagramm

Zentraler Bestandteil eines Zustandsautomaten ist die Definition seiner Zustände und Zustandsübergänge. Diese Zusammenhänge lassen sich in einem Zustandsdiagramm beschreiben. Für die Implementierung und für Tests des Automaten lässt sich das Zustandsdiagramm in Tabellenform übersetzen, zu einer sogenannten Zustandsübergangstabelle (engl. state event table). Als Beispiel sei der Getränkeautomat in vereinfachter Form beschrieben.

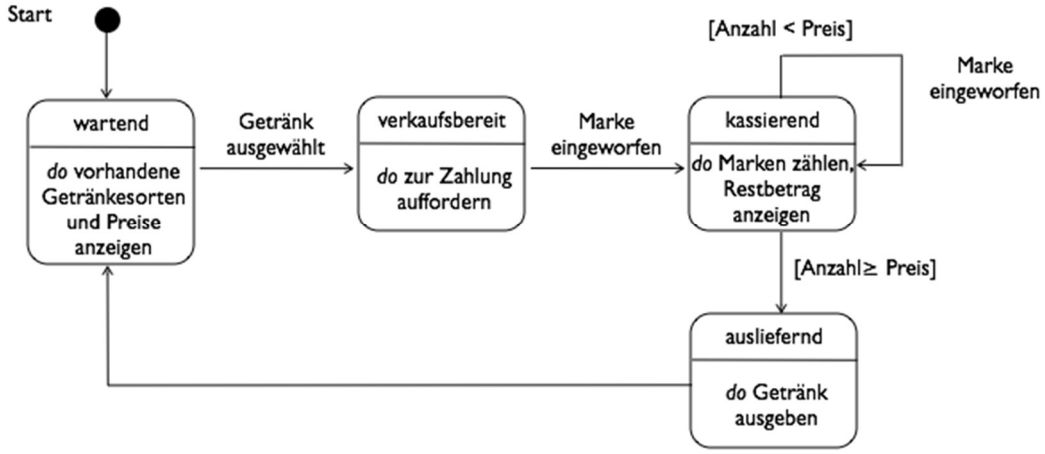


Bild 5.4 Steuerung des Getränkeautomaten

In der vereinfachten Form wird die Bezahlung per Getränkemarken angenommen. Der Automat startet im initialen Zustand „wartend“. Für die Eingangssignale und Ausgangssignale der Steuerung werden folgende Annahmen getroffen. Der Datenpfad wäre auf diese Signale auszulegen.

Eingänge (Statussignale):
- Getränk gewählt: e0
- Marke eingeworfen: e1
- Zahlung erfolgt: e2
- Auslieferung erfolgt: e3

Ausgangssignale:
- Getränkewahl anzeigen: a0,
- Betrag anzeigen: a1
- Restbetrag anzeigen: a2
- Getränk ausgeben: a3

Die Signale werden nun den Zustandsübergängen zugeordnet. An den Zustandsübergängen werden hierbei die jeweils empfangenen Signale vermerkt, die den Zustandsübergang auslösen. Ebenso werden die Ausgangssignale vermerkt, die beim Übergang gesendet werden. Die Sende- bzw. Empfangsrichtung kann man hierbei entweder durch ein Vorzeichen andeuten (wobei „-“ den Empfang

eines Signals und „+“ das Senden eines Signals bedeuten), oder durch Kennzeichnung der Eingangssignale z.B. durch „e“ und Kennzeichnung der Ausgangssignale durch „a“.

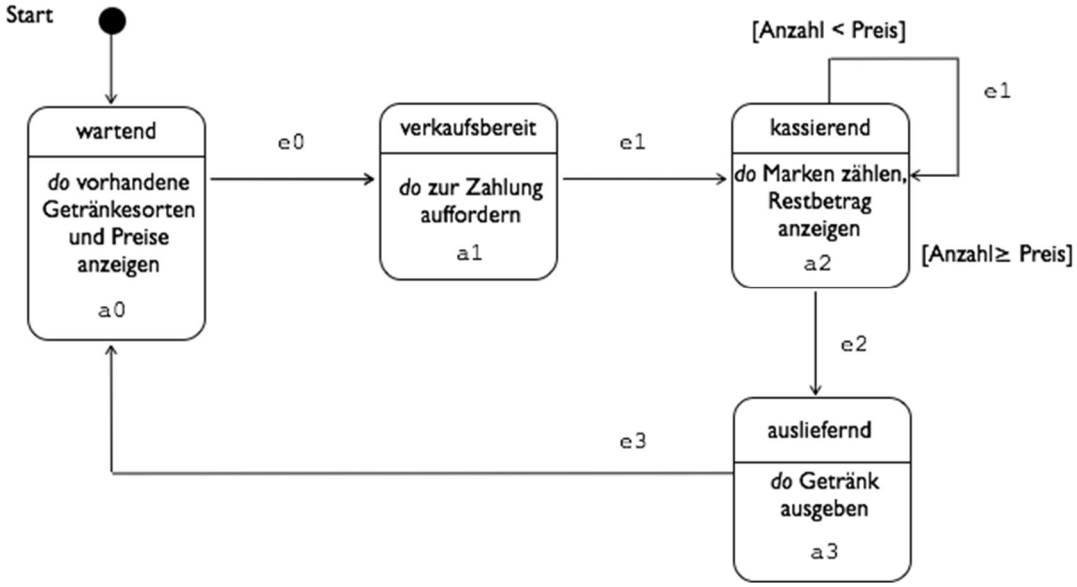


Bild 5.5 Zustandsdiagramm mit Eingangssignalen und Ausgangssignalen

Hinweis: Ein Zustandsübergang muss durch den Empfang eines Eingangssignals ausgelöst werden. Hierzu gehören auch zeitbasierte Signale, wie z.B. ein Time-Out. Fehlende Eingangssignale an den Zustandsübergängen im Zustandsdiagramm deuten auf Fehler im Konzept hin. Wenn man im Beispiel die vier Zustände durchnummeriert, ergibt sich folgende Struktur des Zustandsdiagramms.

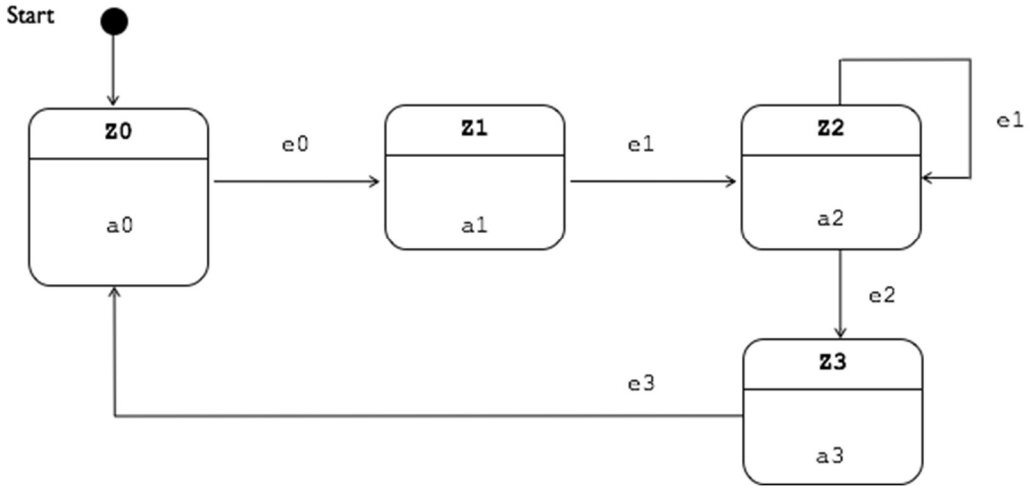


Bild 5.6 Zustandsdiagramm mit formal spezifizierten Signalen und Zuständen

Hinweis: Beim gezeigten Entwurf wird davon ausgegangen, dass alle Aktivitäten, die innerhalb der Zustände hinter dem Schlüsselwort „do“ angedeutet sind, im Datenpfad realisiert werden. Hierzu gehören also die Anzeigen der Auswahl, der Zahlungsaufforderung und des Restbetrags. Auch der Markenzähler wäre im Datenpfad angesiedelt und liefert das Signal „Zahlung erfolgt“. Der Zustandsautomat beschränkt sich rein auf die Speicherung der Zustände und Transitionen zwischen den Zuständen.

5.2. Schaltwerkstabelle für Zustandsübergänge

Das Zustandsdiagramm lässt sich auch komplett in Tabellenform darstellen (Zustandsfolgetabelle, Schaltwerkstabelle bzw. engl. State Event Table). Mit den Ereignissen (Events) sind hierbei die Eingangssignale gemeint, die einen Zustandsübergang auslösen. Aus der formalisierten Darstellung in der letzten Abbildung ergibt sich folgende Tabelle:

aktueller Zustand	Ereignis (Eingangssignal)	Folgezustand	Ausgangssignal
Z0	e0	Z1	a0
Z1	e1	Z2	a1
Z2	e1	Z2	a2
Z2	e2	Z3	a2
Z3	e3	Z0	a3

Nicht in der Tabelle gesondert vermerkt ist der initiale Zustand Z0, der zu jedem Zeitpunkt durch das Signal „Reset“ am Zustandsautomaten erreicht wird. Ebenfalls nicht gesondert vermerkt ist der Takteingang „Clk“ des Zustandsautomaten. Mit dem Takt werden die Eingangssignale abgefragt und im Falle neuer Ereignisse die Zustandsübergänge geschaltet. Aus der Tabelle ergibt sich auch unmittelbar die Beschreibung des Automaten in HDL.

```
--- Finite State Machine (VHDL)
---
library ieee ;
use ieee.std_logic_1164.all;

entity sequence_logic is
port(  E:          in std_logic_vector (0 to 3);
 Clk:        in std_logic;
 Reset:      in std_logic;
 A:          out std_logic_vector (0 to 3));
end sequence_logic;

architecture FSM of sequence_logic is

    -- define the states of the FSM
    type state_type is (Z0, Z1, Z2, Z3);
    signal next_state, current_state: state_type;

begin
```

```
    -- concurrent process#1: update state registers
state_reg: process(Clk, Reset)
begin
  if (Reset='1') then
    current_state <= Z0;

  elsif (rising_edge(Clk) and Clk='1') then
   current_state <= next_state;
  end if;
end process state_reg;

 -- concurrent process#2: combinational logic for next state
comb_logic_next: process(current_state, E)
begin
 -- case statement for state transitions (may contain flaws)
 case current_state is

   when Z0 =>
        if E(0) ='1' then     next_state <= Z1;
        end if;

   when Z1 =>
        if E(1)='1' then      next_state <= Z2;
       end if;

   when Z2 =>
        if E(1)='1' then      next_state <= Z2;
        end if;
        if E(2)='1' then      next_state <= Z3;
        end if;

   when Z3 =>
        if E(3)='1' then      next_state <= Z0;
        end if;

     when others => next_state <= Z0;

  end case;

  end process comb_logic_next;

  -- concurrent process#3: combinational logic for output signals
comb_logic_out: process(current_state)
begin
 -- case statement for output signals
 case current_state is
   when Z0 =>          A <= (others => '0'); A(0) <= '1';
   when Z1 =>          A(1) <= '1';
   when Z2 =>          A(2) <= '1';
   when Z3 =>          A(3) <= '1';
```

```
        when others => A <= (others => '0');
      end case;

       end process comb_logic_out;

    end FSM;
```

Übung 5.2: Erläutern Sie die HDL-Beschreibung des Zustandsautomaten. Aus welchen Blöcken bzw. Bausteinen besteht der Automat? Was ist die Funktion der drei parallelen Prozesse? Identifizieren Sie Zustandsregister, Übergangsschaltnetz und Ausgangsschaltnetz. Wie geschehen die Transitionen zwischen den Zuständen? Wann werden die Ausgänge geschaltet?

Übung 5.3: Wie könnte man den Zustandsautomaten testen? Erstellen Sie ein Konzept. Worin besteht der Unterschied zu einem rein kombinatorischen Schaltnetz?

5.3. Testumgebung

Für einen Test des Zustandsautomaten sind unterschiedliche Fragestellungen relevant:

- Funktioniert das Rücksetzen (Reset)?
- Schaltet der Automat die in der Zustandsfolgetabelle geforderten geforderten Übergänge zuverlässig durch?
- Lässt sich der Automat im jeweiligen Zustand durch andere Ereignisse stören?
- Funktioniert das Rücksetzen aus jedem beliebigen Zustand?

Da ein Zustandsautomat definitionsgemäß abhängig von seinem Zustand jeweils anders auf seine Eingangssignale reagieren soll, gehen die Testfälle also über die Transitionen einer Wertetabelle hinaus. Für den als Beispiel gewählten Automaten könnte man die Testfälle folgendermassen zusammenfassen: (1) Rücksetzen, (2) zyklisches Schalten der Transitionen aus der Zustands-folgetabelle, (3) zyklisches Schalten der Transitionen mit spontanem Rücksetzen (spontan im Sinne von asynchron). Folgender HDL-Text beschreibt eine Testumgebung.

```
--- Test bench for Finite State Machine (VHDL)
---
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity test_FSM is             -- no external signals
end test_FSM;

architecture Behavioural of test_FSM is

component sequence_logic
port(  E:          in std_logic_vector (0 to 3);
 Clk:        in std_logic;
 Reset:      in std_logic;
 A:          out std_logic_vector (0 to 3));
end component;
```

```
-- testbench internal signals
signal T_E         : std_logic_vector (0 to 3);
signal T_Clk       : std_logic;
signal T_Reset     : std_logic;
signal T_A         : std_logic_vector (0 to 3);

begin

-- connect DUT to testbench
DUT: sequence_logic port map(E=>T_E,Clk=>T_Clk,Reset=>T_Reset,A=>T_A);

-- run tests
count : process          -- clock cycle 40 ns
  begin
 T_Clk <= '0';
 wait for 20 ns;
 T_Clk <= '1';
 wait for 20 ns;
end process count;

reset : process          -- periodic resets (565 ns)
  begin
       T_Reset <= '0';
 wait for 555 ns; T_Reset <= '1';
 wait for 10 ns; T_Reset <= '0';
end process reset;

transitions : process    -- transition cycle (250 ns)
  begin
  wait for 10 ns; T_E(0)<='1';T_E(1)<='0';T_E(2)<='0';T_E(3)<='0';
  wait for 50 ns; T_E(0)<='0';T_E(1)<='0';T_E(2)<='0';T_E(3)<='0';
  wait for 10 ns; T_E(0)<='0';T_E(1)<='1';T_E(2)<='0';T_E(3)<='0';
  wait for 50 ns; T_E(0)<='0';T_E(1)<='0';T_E(2)<='0';T_E(3)<='0';
  wait for 10 ns; T_E(0)<='0';T_E(1)<='0';T_E(2)<='1';T_E(3)<='0';
  wait for 50 ns; T_E(0)<='0';T_E(1)<='0';T_E(2)<='0';T_E(3)<='0';
  wait for 10 ns; T_E(0)<='0';T_E(1)<='0';T_E(2)<='0';T_E(3)<='1';
  wait for 50 ns; T_E(0)<='0';T_E(1)<='0';T_E(2)<='0';T_E(3)<='0';
end process transitions;

end Behavioural;
```

Übung 5.4: Erläutern Sie die Funktion der Testumgebung. Aus welchen Blöcken besteht die Testumgebung? Welche parallelen Prozesse gibt es? Was ist die Funktion der Prozesse? Welches Verhalten erwarten Sie vom Prüfling? Welche parallelen Prozesse laufen zusammen mit dem Prüfling insgesamt beim Start der Testumgebung?

Der Simulator gestattet auch die Beobachtung der internen Zustände des Prüflings. Wählen Sie hierzu im Simulationsfenster statt der Testumgebung den Prüfling aus („DUT“, siehe folgende Abbildung). Unter den Objekten finden sich nun die Eingangssignale inklusive Clock und Reset, die

Ausgangssignale sowie die internen Variablen „current state“ und „next_state“. Im Prozessfenster darunter sind übrigens alle parallelen Prozesse aus der Testumgebung und aus dem Prüfling dargestellt.

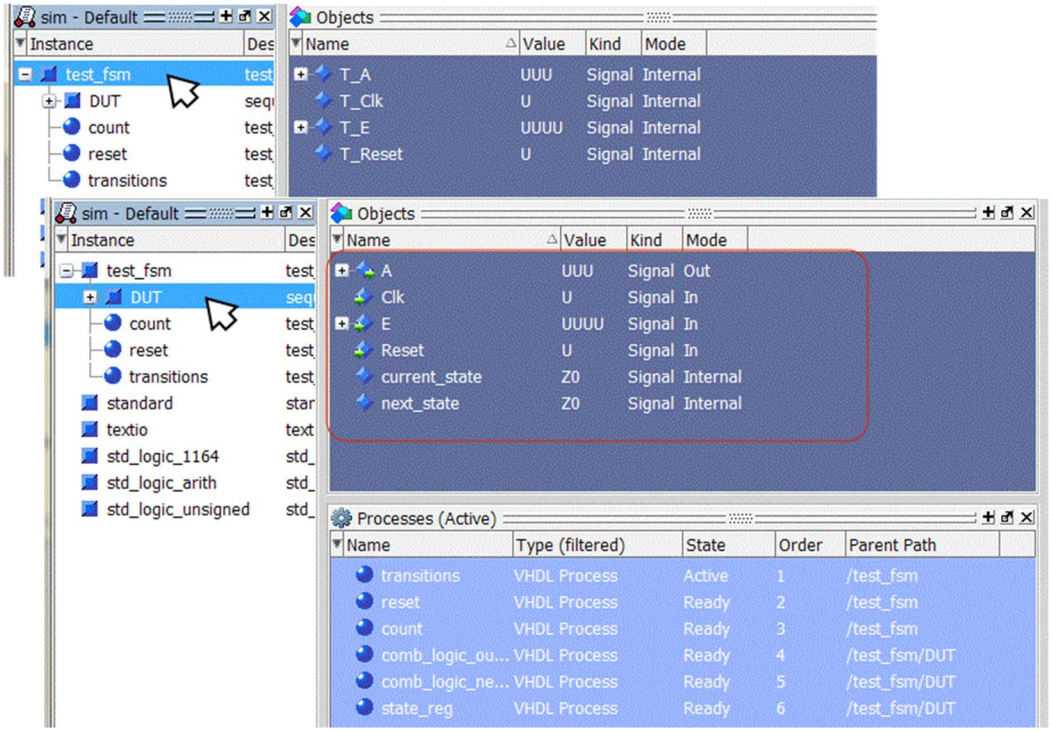


Bild 5.7 Beobachtung der internen Zustände des Automaten

Nach Start der Testumgebung sollte sich das in folgender Abbildung gezeigte Verhalten beobachten lassen. Man erkennt, dass der Automat nach dem initialen Rücksetzen vom Zustand Z0 aus startet (nach Eintreffen von e1 wird taktsynchron in den Zustand Z1 geschaltet, taktsynchron wird a1 ausgegeben).

Nach Auftreten des ersten spontanen Resets bei 565 ns schaltet der Automat in den Zustand Z0. Da das Eingangssignal e1 zu diesem Zeitpunkt bereits verstrichen ist, bleibt der Automat im Zustand Z0, bis das nächste Eingangssignal e1 auftritt.

Ab diesem Zeitpunkt werden die Zustände nach Eintreffen der passenden Eingangssignale wieder zyklisch weiter geschaltet. Die Anzeige der internen Zustände „current_state“ und „next_state“ erleichtert die Beobachtung des Verhaltens erheblich. Andernfalls müsste man von den Ausgangssignalen des Zustandsautomaten aus darauf schliessen, was in seinem Inneren vorgeht.

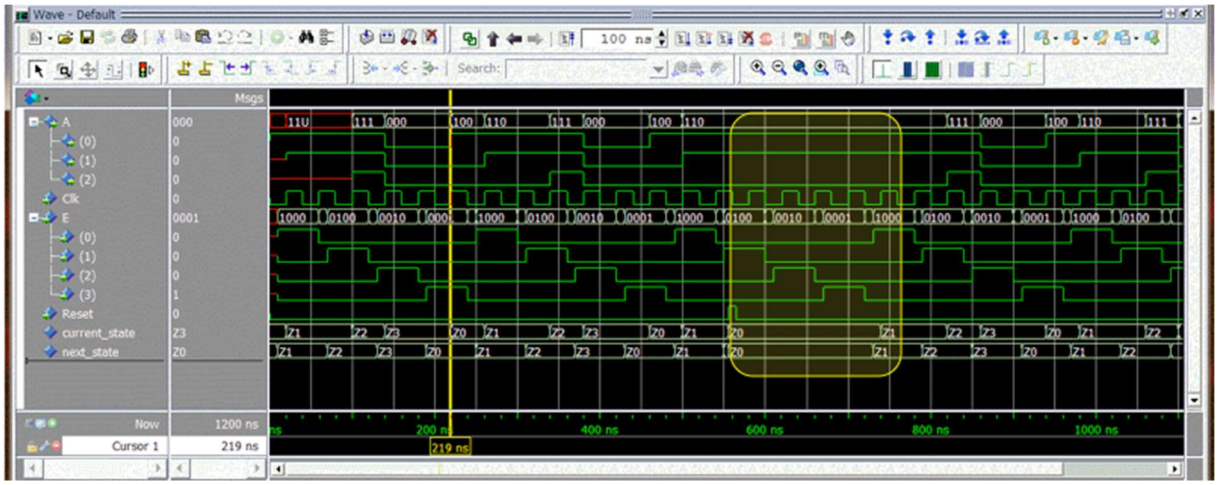


Bild 5.8: Test des Zustandsautomaten

Übung 5.5: Betrachten Sie die HDL-Beschreibung des Automaten. Nur der Prozess für die Aktualisierung des Zustandsregisters ist taktsynchron. Die Prozesse für die kombinatorische Logik zur Berechnung des folgenden Zustands und der Prozess zur Berechnung der Ausgangssignale sind nicht taktsynchron. Diese beiden Prozesse reagieren jederzeit auf Änderungen der Signale in ihren Übergabeparametern. Warum erscheinen die Ausgangssignale in der Simulation dennoch getaktet?

Übung 5.6: Starten Sie die Testumgebung und Prüfling auf dem Simulator und untersuchen Sie das Verhalten des Automaten. Nehmen Sie gegebenfalls Veränderungen am VHDL Code des Prüflings vor, um ein korrektes Verhalten zu erhalten. Machen Sie gegebenfalls Notizen in ihrem Skript zu den erforderlichen Veränderungen.

Übung 5.7: In der gezeigten Testumgebung werden nicht wirklich alle möglichen Kombinationen der vier Eingangssignale zyklisch getestet. Erweitern Sie die Testumgebung um einen solchen Test und untersuchen Sie das Verhalten des Automaten.

5.4. Realisierungsvarianten

Die in Abbildung 5.3 gezeigte Struktur des Zustandsautomaten lässt sich im Sinne der Abbil-dung der Eingänge, Zustände und Ausgänge folgendermassen interpretieren:

$$Z^+ = F_1 (E, Z) \quad (5.1)$$

$$A = F_2 (Z) \quad (5.2)$$

Die beiden Gleichungen sind so zu interpretieren, dass der Folgezustand Z^+ aus den Eingangs-signalen E und dem aktuellen Zustand Z hervorgeht. Die Abbildung F_1 entspricht hierbei dem in Bild 5.3 dargestellten Übergangsschaltnetz. Die Ausgangssignale A gehen aus dem aktuellen Zustand Z hervor, sind also nicht unmittelbar abhängig von den Eingangssignalen E. Diese Abbildung F_2 wird durch das Ausgangsschaltnetz dargestellt. Eine Variante der Gleichung (5.2) wäre:

$$A = I (Z) \quad (5.3)$$

Hierbei ist mit I ein Schaltnetz bezeichnet, dass die Zustände im Sinne einer identischen Transformation in die Ausgangssignale übersetzt, also A(i) = ‚1' im Falle von Z(i). Ein Vorteil besteht darin, dass sich die Zustände des Automaten so direkt an den Ausgangssignalen erkennen lassen. Ein Nachteil ist die Spezialisierung der Zustände auf Ausgangsignale. Bei dem im Beispiel genannten Automaten ist diese Variante jedoch leicht realisierbar.

Übung 5.8: Modifizieren Sie den Ausgangsprozess des Automaten in Abschnitt 5.2 in der HDL-Beschreibung so, dass die Ausgangssignale A(i) den Zuständen Z(i) entsprechen, d.h. A(i) = ‚1' nur, wenn der Automat sich im Zustand Z(i) befindet.

Eine weitere Realisierungsvariante besteht darin, die kombinatorische Logik der beiden Schaltnetze für die Übergänge und Ausgänge zu einer kombinatorischen Logik zusammen zu fassen. Es ergibt sich dann ein Blockschaltbild des Automaten, wie in folgender Abbildung gezeigt. Die Abbildungen (5.1) und (5.2) kann man mit dieser Anordnung natürlich ebenfalls realisieren.

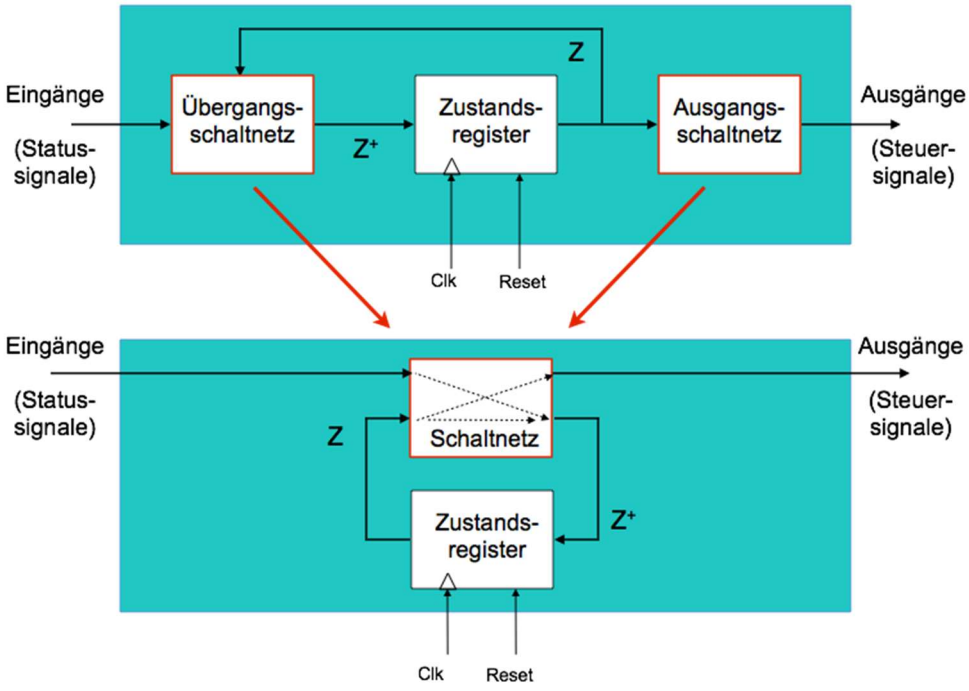


Bild 5.9 Automat mit 2 Prozessen

In der HDL-Beschreibung realisiert man diese Variante durch Zusammenfassung der beiden kombinatorischen Prozesse #2 und #3, wie in folgendem Text gezeigt. Hierbei leidet allerdings die Übersichtlichkeit, wodurch diese Art der Beschreibung auch fehleranfällig ist.

```
-- concurrent process#2_new:
-- combinational logic for next state and output signals
comb_logic_next_and_out: process(current_state, E)
begin
 -- case statement for state transitions
 case current_state is

   when Z0 => A(0) <= '1';
         if E(0)='0' then      next_state <= Z0;
         elsif E(0) ='1' then  next_state <= Z1;
         end if;

   when Z1 => A(1) <= '1';
         if E(1)='0' then      next_state <= Z1;
         elsif E(1)='1' then   next_state <= Z2;
```

```
            end if;

        when Z2 => A(2) <= '1';
              if E(2)='0' then      next_state <= Z2;
              elsif E(2)='1' then   next_state <= Z3;
              end if;

        when Z3 => A <= (others => '0');
              if E(3)='0' then      next_state <= Z3;
              elsif E(3)='1' then   next_state <= Z0;
              end if;

          when others => A <= (others => '0'); next_state <= Z0;

       end case;

      end process comb_logic_next_and_out;
```

Übung 5.9: Vergleichen Sie die HDL-Beschreibung des Automaten mit 2 Prozessen (ein Schaltnetz für Zustandsübergänge und Ausgänge) mit der Beschreibung aus Abschnitt 5.2. Sind die Zustandsübergänge nur Abhängig vom vorausgehenden Zustand und den Eingängen (Gleichung 5.1)? Sind die Ausgänge nur abhängig vom vorausgegangenen Zustand (Gleichung 5.2)?

```
      when Z0 =>
            if E(0)='0' then  next_state <= Z0;
            elsif E(0) ='1' then
                        next_state <= Z1;
                        A(1) <= '1';
            end if;

      when Z1 => ... sinngemäß fortgesetzt
```

Übung 5.10: Würde die oben gezeigte Beschreibung den Aufwand nicht reduzieren, da Ausgangssignale nur dann zugewiesen werden, wenn sich ein Zustand wirklich ändert? Hätte diese Variante irgendwelche Auswirkungen auf das zeitliche Verhalten bzw. auf Gleichungen (5.1) und (5.2)? Wie unterscheiden sich Aufwand im Simulator und Aufwand in der finalen Implementierung z.B. auf FPGA?

5.5. Übungen

Der in folgender Abbildung gezeigte Automat wurde dahingehend erweitert, dass die Eingangssignale nun abgetastet werden. Der Signalzustand von E wird also nur noch mit dem Systemtakt übernommen und in ein synchronisiertes Eingangssignal E_S übersetzt. Auch das Ausgangssignal aus dem Ausgangsschaltnetz wird in einem weiteren Speicherelement im Systemtakt aufgezeichnet und bis zum nächsten Takt konserviert.

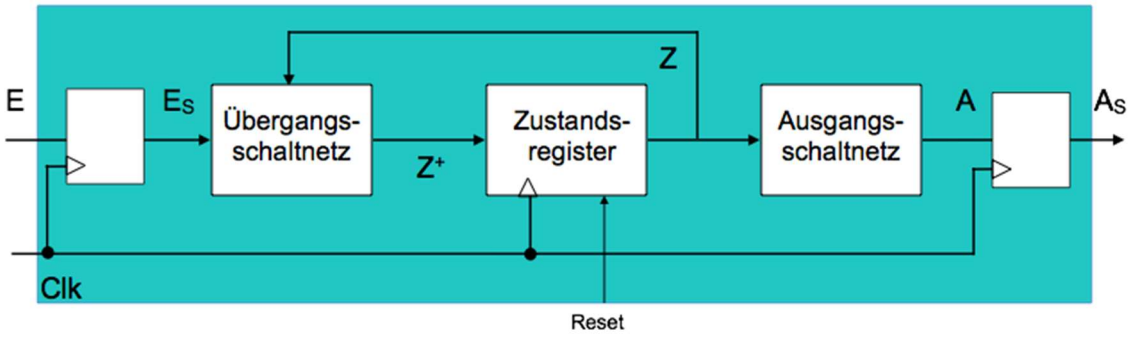


Bild 5.10 Automat mit synchronisierten Eingängen und Ausgängen

Es entsteht ein mehrstufiges Schaltwerk, bei der die Signale im Systemtakt die Schaltnetze durchlaufen und zu den Abtastzeitpunkten zur Verfügung stehen. Am Eingang des Automaten befindet sich weiterhin das nicht synchronisierte Eingangssignal E. Am Ausgang des Automaten steht nun allerdings ein synchronisiertes Ausgangssignal A_S bereit. Die Signale E_S und A sind interne Signale des Automaten. Die Abtastung des Eingangssignals und die Speicherung des Ausgangssignals sollen in der HDL-Beschreibung durch einen zusätzlichen Prozess zur Synchronisation realisiert werden.

```
    -- concurrent process#0: synchronize input and output signals
synch: process(Clk, Reset)
begin
if (Reset='1') then
       ES <= (others => '0'); AS <= (others => '0');
  elsif (rising_edge(Clk) and Clk='1') then
 ES <= E; AS <= A;
  end if;
end process synch;
```

Übung 5.11: Was bewirkt der zusätzliche Prozess? Welcher Einfluss ergibt sich durch das neue interne Signal E_S und das neue Ausgangssignal A_S auf die übrigen Prozesse?

Übung 5.12: Komplettieren Sie die HDL-Beschreibung und testen Sie den Automaten im Simulator.

Das Zustandsregister eines Automaten soll ein zusätzliches asynchrones Eingangssignal mit „Enable“ erhalten, wie in folgender Abbildung gezeigt. Mit Hilfe dieses Signals soll die Funktion des Automaten zugeschaltet bzw. ausgesetzt werden. Die Speicherung neuer Zustände ist nur möglich, wenn das Signal „Enable“ gesetzt ist.

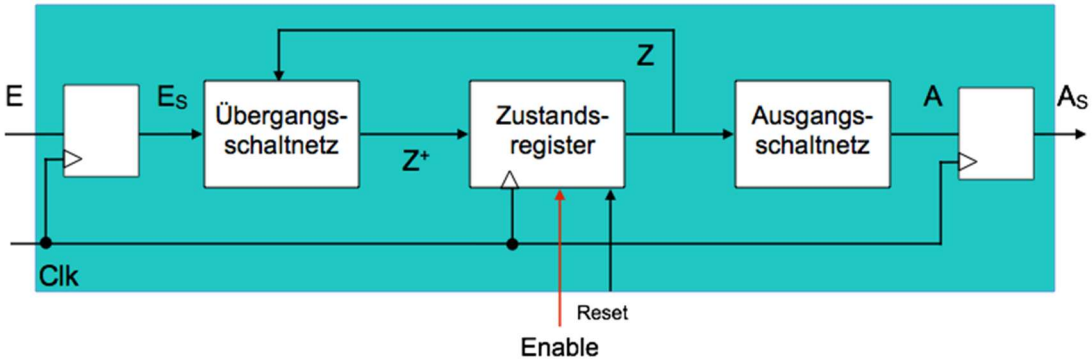


Bild 5.11 Zustandsregister mit zusätzlichem Enable-Eingang

Übung 5.13: Modifizieren Sie die HDL.-Beschreibung des Zustandsautomaten so, dass er einen zusätzlichen Eingang für das Signal „Enable“ erhält. An welcher Stelle setzt die Modifikation am geschicktesten an? Wie testet man die zusätzliche Funktion?

Übung 5.14: Testen Sie den erweiterten Automaten im Simulator.

Die HDL-Prozesse werden im Simulator nur quasi-parallel ausgeführt: Das Laufzeitsystem des Simulators bietet eine Multitasking-Umgebung, in der jeweils einer der parallelen Prozesse in einer Zeitscheibe bearbeitet wird. Für die parallelen Prozesse unterscheidet das Laufzeitsystem Zustände, wie in folgendem Zustandsdiagramm gezeigt.

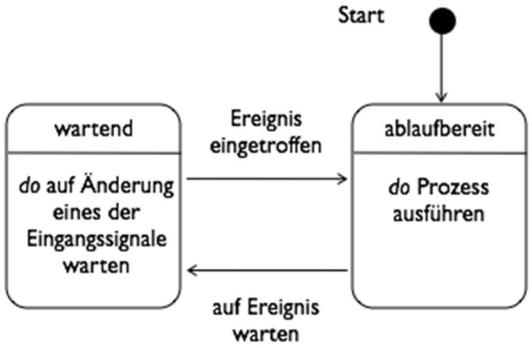


Bild 5.12 Prozesszustände im Simulator

Ein Prozess befindet sich im Zustand wartend, wenn er auf eine Zustandsänderung eines der in seinen Übergabeparametern angegebenen Signale wartet. Anders als die Übergabeparameter einer Methode oder einer Funktion in einer Programmiersprache stellen die einem HDL-Prozess übergebenen Signale für die Laufzeitumgebung Ereignisse (engl. Events) dar. Der Übergabebereich des Prozesses wird auch als sensibler Bereich bezeichnet (engl. sensitivity list). Trifft das erwartete Ereignis ein, wird der Prozess ablaufbereit. Die Laufzeitumgebung arbeitet alle ablaufbereiten Prozesse quasi-parallel ab.

Übung 5.15: Was geschieht im Simulator mit allen ablaufbereiten Prozessen? Worin besteht die Bearbeitung eines Prozesses in Bezug auf seine HDL-Beschreibung? Was geschieht mit dem Prozess nach Abschluss der Bearbeitung?

6. Übungsaufgaben

6.1. Code-Umsetzer

Eine Schaltung soll einen 4-Bit Gray Code in einen 4-Bit BCD-Code umsetzen gemäß folgender Wertetabelle.

Binärcode	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Gray-Code	0	1	3	2	6	7	5	4	C	D	F	E	A	B	9	8

Bild 1.1 Wertetabelle des Code-Umsetzers

Frage 6.1.1 (4 Punkte): Skizzieren Sie ein Blockschaltbild des Code-Umsetzers mit den benötigten Eingangssignalen und Ausgangssignalen

Frage 6.1.2 (6 Punkte): Entwerfen Sie den Code-Umsetzer in VHDL. Hinweis: Verwenden Sie eine `case`-Anweisung.

Frage 6.1.3 (4 Punkte): Erweitern Sie die Schaltung so, dass die binär kodierten Daten taktsynchron abgetastet werden. Verwenden Sie hierzu ein Clock-Signal in Kombination mit einem Register. Skizzieren Sie ein Blockschaltbild.

Frage 6.1.4 (6 Punkte): Erweitern Sie die VHDL-Beschreibung um die Abtastung (Frage 6.1.3). Erläutern Sie Ihren Entwurf in einigen Stichworten.

6.2. Zustandsautomat Variante 1

Folgender HDL-Text beschreibt einen Zustandsautomaten.

```
--- Finite State Machine (VHDL)
library ieee ;
use ieee.std_logic_1164.all;

entity sequence_logic is
 port(       E:          in std_logic_vector(1 downto 0);
                  Clk:        in std_logic;
                  Reset:      in std_logic;
                  A:          out std_logic);
end sequence_logic;

architecture FSM of sequence_logic is

    type state_type is (Z0, Z1, Z2, Z3);
    signal next_state, current_state: state_type;
begin

update_state_reg: process(Clk, Reset)
  begin
       if (Reset='1') then
         current_state <= Z0;
       elsif rising_edge(Clk) then
         current_state <= next_state;
```

```
        end if;
  end process update_state_reg;

  comb_logic_next: process(current_state, E)
    begin
        case current_state is
          when Z0 =>
                if    E = "01" then   next_state <= Z1;
                else                  next_state <= Z0;
                end if;
     when Z1 =>
                if    E = "11" then   next_state <= Z2;
                elsif E = "01" then   next_state <= Z1;
           else                  next_state <= Z0;
                end if;
     when Z2 =>
                if    E = "10" then   next_state <= Z3;
                elsif E = "01" then   next_state <= Z1;
           else                  next_state <= Z0;
                end if;
     when Z3 =>
                if    E = "01" then   next_state <= Z1;
                else                  next_state <= Z0;
                end if;
     when others => next_state <= Z0;
        end case;
  end process comb_logic_next;

  comb_logic_out: process(current_state)
    begin
   case current_state is
        when Z3 => A <= '1';
        when others => A <= '0';
   end case;
  end process comb_logic_out;

  end FSM;
```

Frage 6.2.1 (6 Punkte): Erstellen Sie ein Blockdiagramm des Automaten.

Frage 6.2.2 (8 Punkte): Erläutern Sie die Funktion des Automaten gemäß Blockdiagramm unter Verwendung der Zustandsgleichungen (siehe Abschnitt 5.5 im Manuskript). Erläutern Sie die Zusammenhänge der Komponenten mit den Prozessen in der HDL-Beschreibung.

Frage 6.2.3 (8 Punkte): Skizzieren Sie ein Zustandsdiagramm (inklusive des initialen Zustands). Erläutern Sie die Funktion des Zustandsautomaten mit Hilfe des Diagramms.

Frage 6.2.4 (6 Punkte): Erstellen Sie eine Zustands-Übergangstabelle (State-Event Table).

Frage 6.2.5 (8 Punkte): Testumgebung. Erstellen Sie ein Konzept für eine Testumgebung des Automaten (in Worten, kein HDL-Text). In welchen Schritten gehen Sie vor (welche

Funktionsblöcke bzw. welchen Ablauf hätte ein Testprogramm)? Nach welchen Kriterien können Sie den Automaten testen? Welche Tests sind im speziellen Fall sinnvoll?

Frage 6.2.6 (8 Punkte): Zustandskodierung. Die Zustände des Automaten wurden im HDL-Text wir folgt definiert.

```
type state_type is (Z0, Z1, Z2, Z3);
signal next_state, current_state: state_type;
```

Diese Beschreibung enthält keine Vorgaben für die Synthese der Zustände. Die Implementierung ist somit den Synthesewerkzeugen überlassen. Folgende Varianten sollen nähere Vorgaben über die Zustandskodierung geben.

```
--- Variante 1
type state_type is (Z0, Z1, Z2, Z3);
attribute ENUM_ENCODING: String;
attribute ENUM_ENCODING of state_type: type is "00 01 10 11";
signal next_state, current_state: state_type;
```

```
--- Variante 2
subtype state_type is std_logic_vector (0 to 3);
constant Z0: state_type := "1000"
constant Z1: state_type := "0100"
constant Z2: state_type := "0010"
constant Z3: state_type := "0001"
signal next_state, current_state: state_type;
```

Welche Möglichkeiten zur Zustandskodierung gibt es in der ursprünglichen HDL-Beschreibung, die die Implementierung dem Synthesewerkzeug überlässt? Interpretieren Sie die beiden alternativen Varianten. Welche Vorgaben werden für die Zustandskodierung gegeben? Beschreiben Sie die Unterschiede in der Realisierung.

6.3. FIR Filter

Die Faltung einer zeitdiskreten Funktion x(i) mit der Impulsantwort h_k wird durch die Faltungssumme

$$y[n] = \sum h_k \cdot x[n-k] \qquad (3.1)$$

beschrieben, wobei der Index k über alle vorhandenen Stützstellen h_k verläuft. Für eine Impulsantwort mit insgesamt 5 Stützstellen ergibt sich folgende Gleichung.

$$y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3] + h_4 \cdot x[n-4] \qquad (3.2)$$

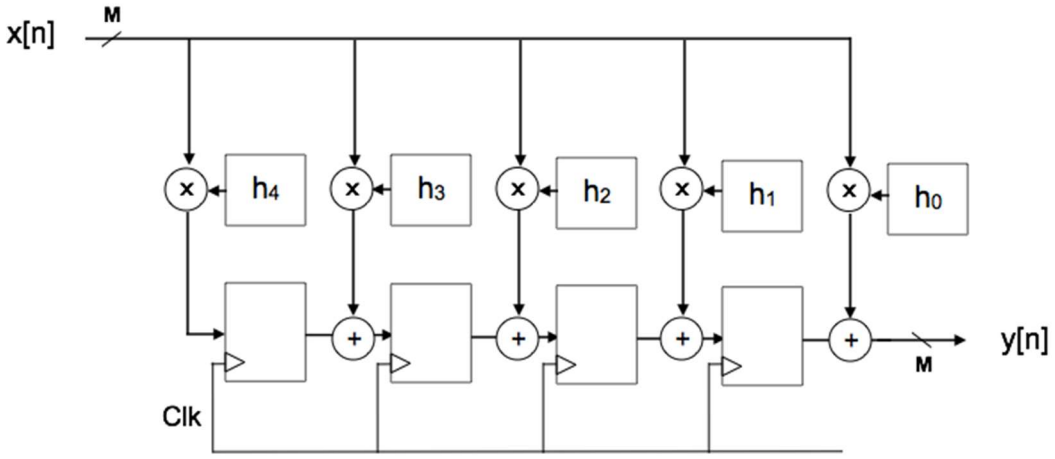
Frage 6.3.1 (4 Punkte): Erläutern Sie Gleichung (3.2). Wie wirken die Stützstellen der Impuls-antwort (Filterkoeffizienten) auf die Werte der Eingangswerte x[n-k]?

Frage 6.3.2 (6 Punkte): Geben Sie ein Blockschaltbild zur Realisierung des Filters aus Gleichung (3.2) an. Verwenden Sie Register, Addierer und Multiplizierer.

Frage 6.3.3 (6 Punkte): Die Werte der Eingangsfunktion x[n-k], sowie die Filterkoeffizienten h_k seien als 12 Bit Festkommazahlen gegeben. Das Ausgangssignal y[n-k] sei ebenfalls als 12 Bit Festkommazahl realisiert. Geben Sie die Wortbreiten im Blockschaltbild so vor, dass innerhalb der Schaltung keine Rundungsfehler auftreten. Begründen Sie Ihre Entscheidung.

Frage 6.3.4 (8 Punkte): Folgendes Blockschaltbild beschreibt eine alternative Realisierung des Filters. Weisen Sie die Äquivalenz zu Ihrem Blockschaltbild bzw. zu Gleichung (3.2) nach. Vergleichen

Sie diese Implementierung mit Ihrem Blockschaltbild (Vorteile, Nachteile). Welche Wortbreiten sind erforderlich?



Frage 6.3.5 (6 Punkte): Folgender HDL-Text beschreibt ein digitales Filter. Analysieren Sie die Beschreibung und erläutern Sie den Entwurf (Struktur, Funktionsblöcke, Funktionen, ...).

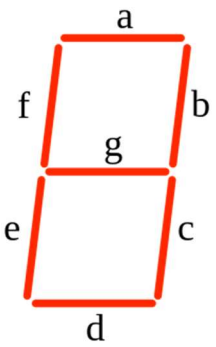
```
--- FIR Filter (VHDL), simplified form without reset

   library IEEE;                                --- part #1
     use IEEE.std_logic_1164.all;
     use IEEE.numeric_std.all;

   entity FIR_4 is                            --- part #2
    port (Clk    : in  std_logic;             -- clock
          XN     : in  std_logic_vector (11 downto 0);
          YN     : out std_logic_vector (11 downto 0));
   end FIR_4;

   architecture RTL of FIR_4 is               --- part #3
                                              --- part #3.1
     type COEFF_TYPE is array (0 to 4) of signed (11 downto 0);
     type PRODUCT_TYPE is array (0 to 4) of signed (23 downto 0);
     type SUM_TYPE is array (0 to 4) of signed (23 downto 0);

     signal COEFF    : COEFF_TYPE;     -- coefficients array
     signal PRODUCT  : PRODUCT_TYPE;   -- product array
     signal SUM            : SUM_TYPE;       -- sum array

     signal XD       : signed (11 downto 0); -- register for X[0]
     signal lastsum : std_logic_vector (23 downto 0);
     begin                                    --- part #3.2
         COEFF(0) <= "010011011101";          --  0,60769
         COEFF(1) <= "001110101110";          --  0,45992
         COEFF(2) <= "000100100011";          --  0,14212
         COEFF(3) <= "100011000010";          -- -0,09475
         COEFF(4) <= "100011101011";          -- -0,11498
```

```
    new_sample : process (Clk)          --- part #3.3
      begin
        if rising_edge(Clk) then
              XD <= signed(XN(11 downto 0));
        end if;
    end process new_sample;

    multiply : process (COEFF, XD)      --- part #3.4
      variable MUL : signed (23 downto 0);
      begin
        for i in 0 to 4 loop
              MUL := XD * COEFF(i);
              PRODUCT(i) <= MUL;
        end loop;
    end process multiply;

    add_stages : process (CLK)          --- part #3.5
      begin
        if rising_edge(Clk) then
         SUM (4) <= PRODUCT (4);
              for i in 0 to 3 loop
                    SUM(i) <= SUM(i+1) + PRODUCT(i);
              end loop;
        end if;
    end process add_stages;

   lastsum <= std_logic_vector (SUM(0));          --- part #3.6
   YN <= lastsum (23 downto 12);

  end RTL;
```

Frage 6.3.6 (6 Punkte): Erstellen Sie ein Konzept für ein Testprogramm des Filters (in Worten). In welchen Schritten gehen Sie vor (welche Funktionsblöcke bzw. welchen Ablauf hätte das Testprogramm)? Nach welchen Kriterien können Sie das Filter testen?

6.4. Dekoder für Segmentanzeige

Eine Schaltung soll einen 4-Bit BCD-Code umsetzen zur Ansteuerung einer Anzeige mit 7 Segmenten, wie in der folgenden Abbildung gezeigt.



7 Segmentanzeige

Frage 6.4.1 (4 Punkte): Skizzieren Sie ein Blockschaltbild des Dekoders. Hinweis: Der Dekoder soll als getaktete Logik ausgeführt werden.

Frage 6.4.2 (4 Punkte): Erstellen Sie eine Wertetabelle zur Anzeige der Ziffern 0 bis 9.

Frage 6.4.3 (8 Punkte): Entwerfen Sie den Code-Umsetzer in VHDL. Hinweis: Verwenden Sie eine `case`-Anweisung.

Frage 6.4.4 (8 Punkte): Entwerfen Sie einen Testprozess, der das BCD-Signal durch einen Zähler erzeugt. Skizzieren Sie Testumgebung und Prüfling als Blockschaltbild, aus dem die benötigten Testsignale hervorgehen. Skizzieren Sie den Testprozess in VHDL (Hinweis: bitte nur den Testprozess in einigen Zeilen, nicht die komplette Testumgebung).

6.5. Zustandsautomat Variante 2

Folgender HDL-Text beschreibt einen Zustandsautomaten.

```
--- Finite State Machine (VHDL)
library ieee ;
use ieee.std_logic_1164.all;

entity sequence_logic is
 port(       E:          in std_logic_vector(1 downto 0);
                  Clk:        in std_logic;
                  Reset:      in std_logic;
                  A:          out std_logic);
end sequence_logic;

architecture FSM of sequence_logic is

    type state_type is (Z0, Z1, Z2);
    signal next_state, current_state: state_type;

begin

update_state_register: process(Clk, Reset)       -- process #1
  begin
       if (Reset='1') then
         current_state <= Z0;
       elsif rising_edge(Clk) then
         current_state <= next_state;
       end if;
end process update_state_register;

logic_next_state: process(E, current_state)      -- process #2
  begin
       case current_state is
         when Z0 =>
              if    E = "01" then   next_state <= Z1;
              else                  next_state <= Z0;
              end if;
   when Z1 =>
               if    E = "11" then   next_state <= Z2;
               elsif E = "01" then   next_state <= Z1;
```

```
            else                    next_state <= Z0;
                 end if;
       when Z2 =>
                 if    E = "01" then   next_state <= Z1;
            else                    next_state <= Z0;
                 end if;
       when others => next_state <= Z0;
           end case;
    end process logic_next_state;

    logic_out: process(E, current_state)             -- process #3
      begin
     if ((current_state = Z2) and (E = "10")) then   A <= '1';
     else                                            A <= '0';
     end if;
    end process logic_out;

    end FSM;
```

Frage 6.5.1 (8 Punkte): Erläutern Sie die Funktion des Automaten. Welche Rolle spielen die Prozesse innerhalb des Automaten? Vergleichen Sie die Rolle der Prozesse mit den Zustandsgleichungen aus dem Manuskript (siehe Abschnitt 5.5, Gleichungen 5.1 und 5.2). Wie lauten die Zustandsgleichungen dieses Automaten?.

Frage 6.5.2 (6 Punkte): Erstellen Sie ein Blockdiagramm des Automaten.

Frage 6.5.3 (8 Punkte): Skizzieren Sie ein Zustandsdiagramm (inklusive des initialen Zustands). Erläutern Sie die Funktion des Zustandsautomaten mit Hilfe des Diagramms.

Frage 6.5.4 (6 Punkte): Erstellen Sie eine Zustands-Übergangstabelle (State-Event Table).

Frage 6.5.5 (8 Punkte): Testumgebung. Erstellen Sie ein Konzept für eine Testumgebung des Automaten (in Worten, kein HDL-Text). In welchen Schritten gehen Sie vor (welche Funktionsblöcke bzw. welchen Ablauf hätte ein Testprogramm)? Nach welchen Kriterien können Sie den Automaten testen? Welche Tests sind im speziellen Fall sinnvoll?

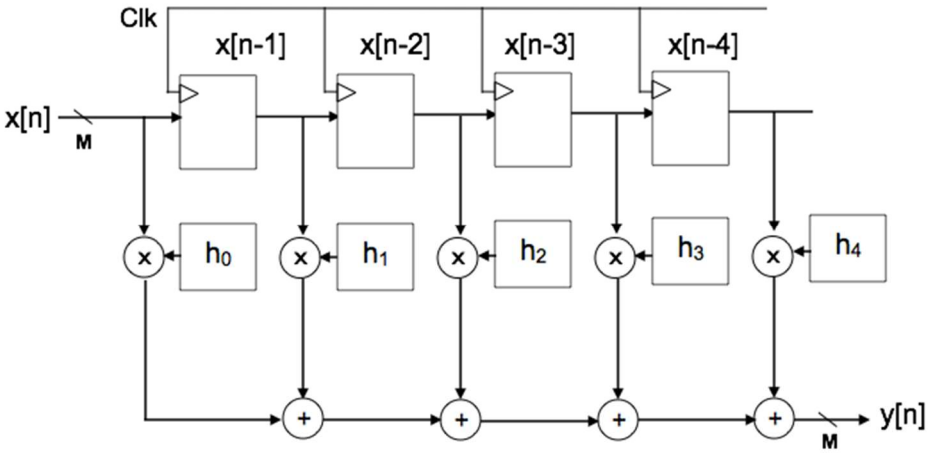
Frage 6.5.6 (8 Punkte): Zusammenfassung beider Schaltnetze. Führen Sie die Prozesse 2 und 3 zu einem gemeinsamen Prozess zusammen und skizzieren Sie die diesbezügliche HDL-Beschreibung (nur den Prozess). Bewerten Sie die Vorteile und Nachteile dieser Realisierung mit der vorgegebenen Implementierung.

6.6. Digitale Filter

Die Faltung einer zeitdiskreten Funktion x(i) mit der Impulsantwort h_k wird durch die Faltungssumme

$$y[n] = \sum h_k \cdot x[n-k] \qquad (3.1)$$

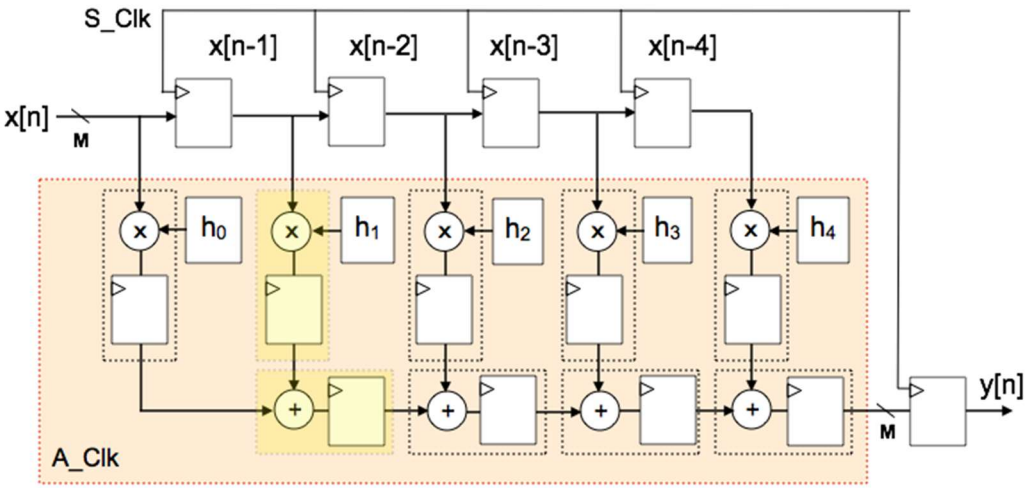
beschrieben, wobei der Index k über alle vorhandenen Stützstellen h_k verläuft. Mit Hilfe von Registern zur Speicherung vergangener Werte der Eingangsfunktion x[n−k], sowie Bausteinen zur Addition und Multiplikation lässt sich diese Gleichung als digitales Filter abbilden. Folgende Abbildung zeigt ein digitales Filter für eine Impulsantwort mit insgesamt 5 Stützstellen (bzw. für 5 Filterkoeffizienten).



Frage 6.6.1 (4 Punkte): Welche Gleichung beschreibt die Schaltung für das Ausgangssignal y[n]?

Frage 6.6.2 (4 Punkte): Welche Schwierigkeiten ergeben sich bei Implementierung dieser Struktur? Wie geht man mit diesen Schwierigkeiten um?

Frage 6.6.3 (6 Punkte): Folgende Abbild zeigt eine Realisierung des Filters, das erst nach einer vorgegebenen Anzahl von Systemtakten den nächsten Wert des Eingangssignals übernimmt. Beschreiben Sie die Funktion dieser Struktur. Wie vermeidet diese Struktur die Schwierigkeiten aus Frage 6.6.2?



Frage 6.6.4 (4 Punkte): In der in Frage 6.6.3. gezeigten Struktur liegen die Stützstellen des Eingangssignals, des Ausgangssignals, sowie der Filterkoeffizienten mit einer Wortbreite von M Bit vor. Welche Wortbreiten werden innerhalb des Rechenwerkes benötigt? Begründen Sie Ihren Entwurf.

Frage 6.6.5 (8 Punkte): Folgender HDL Text beschreibt eine Implementierung eines der Glieder der im Bild zu Frage 6.6.3 gezeigten Kettenstruktur. Skizzieren Sie ein Blockdiagramm der Zelle mit den benötigten Signalen (extern und intern). Beschreiben Sie den Aufbau und die Funktion des Bausteins.

```
--- FIR Filter Cell (VHDL), simplified form without reset

library IEEE;                          --- part #1
  use IEEE.std_logic_1164.all;
```

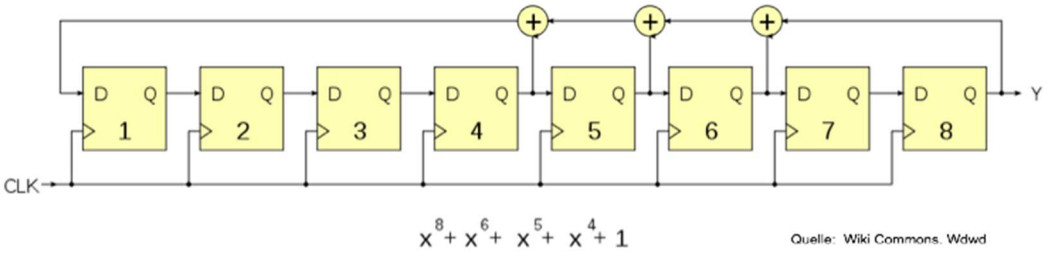
```
    use IEEE.std_logic_signed.all;
    use IEEE.numeric_std.all;

entity FIR_CELL is                    --- part #2
    port (S_Clk   : in  std_logic;     -- sample clock
          A_Clk   : in  std_logic;     -- algorithm clock
          X_IN    : in  std_logic_vector (11 downto 0);
          COEFF   : in  std_logic_vector (11 downto 0);
          SUM_IN  : in  std_logic_vector (23 downto 0);
          SUM_OUT : out std_logic_vector (23 downto 0);
          X_OUT   : out std_logic_vector (11 downto 0));
end FIR_CELL;

architecture RTL of FIR_CELL is       --- part #3
                                  --- part #3.1
    signal PRODUCT : std_logic_vector (23 downto 0);
    signal SAMPLE  : std_logic_vector (11 downto 0);

    begin

    new_sample : process (S_Clk)        --- part #3.2
      begin
   if rising_edge(S_Clk) then
         SAMPLE <= X_IN;
   end if;
    end process new_sample;

    filter : process (A_Clk)            --- part #3.3
      begin
   if rising_edge(A_Clk) then
         PRODUCT <= SAMPLE * COEFF;
         SUM_OUT <= PRODUCT + SUM_IN;
   end if;
    end process filter;

    X_OUT <= SAMPLE;                    --- part #3.4

end RTL;
```

Frage 6.6.6 (6 Punkte): Die in Frage 3.5 gezeigten Filterbausteine lassen sich als Glieder einer Kette zu der in der Abbildung zu Frage 3.3 gezeigten Struktur aufbauen. Skizzieren Sie den Aufbau durch ein Blockschaltbild, das die äußere und innere Struktur beschreibt. Erläutern Sie, wie das Filter den eingangs beschriebenen Algorithmus abbildet.

6.7. Zufallszahlen

Pseudo-Zufallszahlen lassen sich mit Hilfe eines rückgekoppelten Schieberegisters erzeugen, wie in folgender Abbildung gezeigt. Die zurück gelesenen Werte einzelner Stufen sind hierbei mit Hilfe

des Operators „+“ für XOR verknüpft. Als Initialwert des Registers sollten nicht alle Werte zu Null gesetzt werden.



Frage 6.7.1 (6 Punkte): Verwenden Sie die Anordnung in einer Schaltung, mit Hilfe derer sich Zufallszahlen erzeugen lassen. Geben Sie ein Blockdiagramm der Schaltung mit allen Eingangssignalen und Ausgangssignalen an. Beschreiben Sie stichwortartig die Funktionsweise der Schaltung.

Frage 6.7.2 (8 Punkte): Beschreiben Sie die Schaltung in HDL.

Frage 6.7.3 (6 Punkte): Definieren Sie eine Testumgebung für die Schaltung. Welches ist der grundsätzliche Aufbau der Testumgebung? Welche Prozesse verwenden Sie für die Durchführung der Tests?

Frage 6.7.4 (6 Punkte): Beschreiben Sie die Testschaltung in HDL.

6.8. Arithmetisch-Logische Einheit

Es soll eine Arithmetisch-Logische-Einheit (ALU) für zwei 4 Bit breite, vorzeichenlose Operanden unter Verwendung geeigneter VHDL-Operatoren entworfen werden. Das Ergebnis soll ebenfalls 4 Bit breit sein. Außerdem sollen zwei Flags (Zero- und Carry-Flag) erzeugt werden. Die Funktion der ALU wird von dem 2 Bit breiten Signal Opcode wie folgt gesteuert:

Opcode	**Funktion**	**C_Flag**
00	Addition A + B	1 falls Ergebnis > 0xF, sonst 0
01	Subtraktion A – B	1 falls Ergebnis < 0, sonst 0
10	Bitweise ODER-Verknüpfung	0
11	Bitweise UND-Verknüpfung	0

Tabelle 2: Funktionen der Arithmetisch-Logischen-Einheit

Das Z_Flag wird gesetzt, falls das ALU-Ergebnis = 0 ist.

Frage 6.8.1 (8 Punkte): Zeichnen Sie ein Blockdiagramm der Schaltung mit externen und internen Signalen für die beschriebene Arithmetisch-Logische Einheit (ALU). Bitte erläutern sie die internen Funktionsblöcke in Stichworten.

Frage 6.8.2 (4 Punkte): Welche Breite wird für die internen Operanden benötigt? Begründen Sie Ihre Entscheidung und erläutern Sie die Auswirkung auf die Operationen

Frage 6.8.3 (8 Punkte): Im Folgenden ist das Gerüst eines VHDL Codes für die oben beschriebene Arithmetisch-Logische Einheit (ALU) angegeben. Bitte ergänzen Sie die fehlenden Stellen so, dass die oben beschriebene Funktionalität realisiert wird.

```
-- 4-Bit ALU
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ALU4 is
     port( A, B: in std_logic_vector(3 downto 0); --4-Bit operand A and B
       Opcode: in std_logic_vector(1 downto 0);    --2-Bit Opcode
       Result: out std_logic_vector(3 downto 0);   --4-Bit result
       C_Flag, Z_Flag: out std_logic);             --Carry Flag / Zero Flag
end ALU4;

architecture Behavioural of ALU4 is

-- internal signals
signal Temp_Result, Temp_A, Temp_B: _ _ _ _ _ __ ; -- hier bitte Code ergänzen

begin

-- assign A & B to internal signals
_ _ _ _ _ _ _ _ _ _ _ _ _ _                    -- hier bitte Code ergänzen
_ _ _ _ _ _ _ _ _ _ _ _ _ _                    -- hier bitte Code ergänzen

calc: process (Temp_A, Temp_B, Opcode)  -- calculate temporary result
begin
                                               -- hier bitte Code ergänzen
                                               -- (oder auf separatem Blatt)
           _
           _
           _
           _
           _
           _
end process calc;

setzflag: process (Temp_Result) -- Set Z_Flag
begin
                                                -- hier bitte Code ergänzen
                                                -- (oder auf separatem
           _
     Blatt)
           _
           _
           _
           _
           _
end process setzflag;
```

```
setcflag: process (Temp_Result) -- Set C_Flag
begin
                                                -- hier bitte Code ergänzen
                                                -- (oder auf separatem
    Blatt)
end process setcflag;

Result <= _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ ;       -- hier bitte Code ergänzen

end Behavioural;
```

Frage 6.8.4 (8 Punkte): Für die Arithmetisch-Logische-Einheit (ALU) soll die unten skizzierte Testumgebung zum Einsatz kommen. Beschreiben sie das grundsätzliche Vorgehen beim Test in Worten. Welche Testfälle machen für die Arithmetisch-Logische-Einheit (ALU) Sinn? Wählen Sie sechs sinnvolle Testfälle aus und geben Sie den Testvektor für diese 6 Testfälle an.

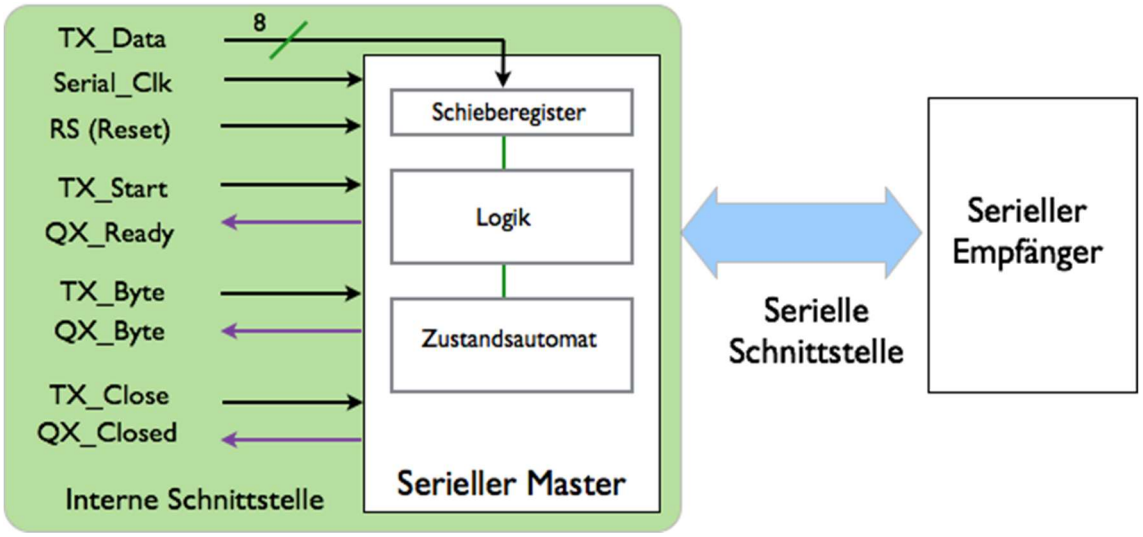
```
-- Testbench for ALU4
library ieee;
use ieee.std_logic_1164.all;

entity test_ALU4 is
end test_ALU4;

architecture behavior of test_ALU4 is

    -- component Declaration for the Device Under Test (DUT)
    component ALU4
    port(
         A : in  std_logic_vector(3 downto 0);
         B : in  std_logic_vector(3 downto 0);
         Opcode : in  std_logic_vector(1 downto 0);
         Result : out  std_logic_vector(3 downto 0);
         C_Flag : out  std_logic;
         Z_Flag : out  std_logic
        );
    end component;

   --inputs
   signal A : std_logic_vector(3 downto 0) := (others => '0');
   signal B : std_logic_vector(3 downto 0) := (others => '0');
   signal Opcode : std_logic_vector(1 downto 0) := (others => '0');

     --outputs
   signal Result : std_logic_vector(3 downto 0);
   signal C_Flag : std_logic;
   signal Z_Flag : std_logic;

    -- testbench clock signals
```

```
    signal clk : std_logic := '0';

    -- testbench test vectors
    type test_rec is record
           A : std_logic_vector(3 downto 0);
           B : std_logic_vector(3 downto 0);
           Opcode : std_logic_vector(1 downto 0);
           Result : std_logic_vector(3 downto 0);
           C_Flag : std_logic;
           Z_Flag : std_logic;
    end record;

    type test_arr is array(positive range <>) of test_rec;

    constant test_vector : test_arr := (
       -- hier wird der Testvektor gesetzt,
       -- geben sie diesen für Ihre Testfälle an
      );
begin
    -- Instantiate the Unit Under Test (UUT)
   DUT: ALU4 port map (
          A => A,
          B => B,
          Opcode => Opcode,
          Result => Result,
          C_Flag => C_Flag,
          Z_Flag => Z_Flag
       );

    -- run tests
    test: process
    begin
     L1: for j in 1 to 6 loop           -- step to next transition

       A <= test_vector(j).A;           -- set input signal to test vector
       B <= test_vector(j).B;           -- set input signal to test vector
       Opcode <= test_vector(j).Opcode;  -- set input signal to test vector

       clk <= '0';
       wait for 50 ns;
       clk <= '1';
       wait for 50 ns;

        -- compare output signals
       if((Result/=test_vector(j).Result)                                or
     (C_Flag/=test_vector(j).C_Flag)
       or (Z_Flag/=test_vector(j).Z_Flag)) then
           report "test step failed" severity NOTE;
       else
           report "test step passed" severity NOTE;
```

```
        end if;
      end loop L1;
     wait;
     end process test;

end behavior;
```

6.9. Zustandsautomat für serielles Protokoll

Zur Übertragung einzelner Bytes auf ein externes Bauteil (serieller Empfänger) soll eine serielle Schnittstelle verwendet werden. Zur Steuerung der Schnittstelle wird ein Zustandsautomat eingesetzt. Folgendes Diagramm beschreibt die Anordnung.



Es werden folgende Signale zur Steuerung der Schnittstelle verwendet:

- TX_Start: Eröffnet die serielle Übertragung; Automat quittiert mit QX_Ready
- TX_Byte: Übertragung eines Bytes über die serielle Schnittstelle; Automat quittiert nach Übertragung des Bytes mit QX_Byte
- TX_Close: Es sind keine weiteren Bytes zu übertragen; Automat gibt serielle Schnittstelle frei, geht in den Wartezustand und quittiert mit QX_Closed

Die Signale bzw. Quittungen werden als Eingänge E bzw. Ausgänge A des Zustandsautomaten kodiert. Für den Test der internen Schnittstelle wird folgender HDL-Programmtext verwendet:

```
--- Uebliche Struktur eines Testprogramms (Component, Signale, DUT über Port-
Map einspannen), dann folgende Testschritte:

-- generate clock
  clock : process begin
    T_Serial_CLK <= '1';
    wait for 10 ns;
    T_Serial_CLK <= '0';
    wait for 10 ns;
```

```
  end process clock;

  test : process begin

    T_RS <= '0';            -- Reset (active low)
    wait for 5 ns;
    T_RS <= '1';
    wait for 5 ns;

    T_TX_Data <= x"00";     -- transmit first byte
    T_E <= "01";            -- TX_Start
    wait until T_A = "01";  -- QX_Ready

    T_E <= "10";            -- TX_Byte
    wait until T_A = "10";  -- QX_Byte
    wait for 5 ns;

    T_TX_Data <= x"bb";     -- transmit next byte
    T_E <= "01";            -- TX_Start
    wait until T_A = "01";  -- QX_Ready

    T_E <= "10";            -- TX_Byte
    wait until T_A = "10";  -- QX_Byte
    wait for 5 ns;

    --  close transmission
    T_E <= "11";            -- TX_Close
    wait until T_A = "11";  -- QX_Closed

    wait;

  END PROCESS test;
```

Frage 6.9.1 (8 Punkte): Erstellen Sie das Zustandsdiagramm des Automaten.

Frage 6.9.2 (6 Punkte): Erläutern Sie die Funktion des Automaten mit Hilfe des Zustandsdiagramms. Ordnen Sie die Ereignisse den Eingängen des Automaten zu, sowie die Quittungen den Ausgängen.

Frage 6.9.3 (4 Punkte): Erstellen Sie eine Zustandsübergangstabelle.

Frage 6.9.4 (8 Punkte): Skizzieren Sie eine HDL-Implementierung des Automaten, in dem Sie den folgenden HDL-Text geeignet ergänzen. Erläutern Sie die Funktion der Zustände. Erläutern Sie die Funktion der Prozesse.

```
entity Serial_Master is
  Port ( RS   : in STD_Logic;                    -- Reset (active low)
    TX_Data: in STD_LOGIC_VECTOR (7 downto 0);  -- Byte Register
    Serial_Clk    : in  STD_LOGIC;              -- Serial clock

    E       : in  STD_LOGIC_vector (1 downto 0); -- Control Events
    -- "01" TX_Start, "10" TX_Byte,"11" TX_Close
```

```
    A  : out  STD_LOGIC_vector (1 downto 0);     -- State Actions
    -- "01" QX_Ready, "10" QX_Byte,"11" QX_Closed

      -- serial bus, e.g. SPI
           MOSI       : out STD_LOGIC;    -- SPI Master out signal
           SCLK       : out STD_LOGIC;    -- SPI clock signal out
           SS         : out STD_LOGIC     -- SPI chip select signal
         );
end Serial_Master;

architecture RTL of Serial_Master is

-- internal signals and variables
signal txreg        : std_logic_vector(7 downto 0) := (others=>'0');
signal counter      : integer := 0;

type state_type is (Z0, Z1, Z2);
 -- Z0: . . . . . .
 -- Z1: . . . . . .
 -- Z2: . . . . . .

signal next_state, current_state: state_type;

begin

  -- update state register in engine
  update_state_register: process(Serial_Clk, RS) -- process #1
    begin
      if (RS = '0') then
          current_state <= Z0;
      elsif rising_edge(Serial_Clk) then
          current_state <= next_state;
      end if;
  end process update_state_register;

  -- implement state logic
  logic_next_state: process(E, current_state) -- process #2
    begin
      case current_state is
        when Z0 =>
            if . . . . . then next_state <= . . . ;


            end if;
       when Z1 =>
            if . . . . . then next_state <= . . . ;


            end if;
        when Z2 =>
```

```
            if . . . . . then next_state <= . . . ;


            end if;
        when others => next_state <= Z0;
      end case;
    end process logic_next_state;
```

Frage 6.9.5 (6 Punkte): Ergänzen Sie folgenden HDL-Text mit den der Aktionen des Zustandsautomaten (den Prozess zur Schaltung der Zustandsaktionen). Kodieren Sie hierzu die Ausgangssignale in geeigneter Weise.

```
-- state actions
 comb_logic_out: process(current_state, Serial_Clk) -- process 3
  begin
    case current_state is
      when Z0 =>
          SS <='1';       -- deselect chip
          SCLK <= '0';    -- SCLK operated in CPOL = 0
          A <= . . . . . .

      when Z1 =>            -- initialize bus/ serial interface
          counter <= 8;  -- initialize counter for byte transmission
          SS <='0';         -- chip select is active low
          SCLK <= '0';      -- SCLK operated in CPOL = 0 (active low)
          txreg <= TX_Data;
          A <= . . . . . .

      when Z2 =>            -- transfer one byte
        if(falling_edge(Serial_Clk) and (counter > 0))then
          SCLK <= '0'; -- SCLK follows serial clock for 8 counts
          MOSI <= txreg(7);
          for i in 0 to 6 loop
            txreg(7-i) <= txreg (6-i);
          end loop;
        elsif (rising_edge(Serial_Clk) and (counter > 0)) then
          SCLK <= '1'; -- SCLK follows serial clock for 8 counts
          counter <= counter - 1;
        end if;

        if (counter = 0) then
           A <= . . . . . .
        end if;

      when others => A <= . . . . . .

    end case;
  end process comb_logic_out;
end RTL;
```

Frage 6.9.6 (8 Punkte): Erläutern Sie die Funktionen der einzelnen Prozesse der vorliegenden Implementierung im Zusammenhang mit den Zustandsgleichungen. Skizzieren Sie ein Blockdiagramm des Automaten (mit Bezug zu den Prozessen). Welche Vorteile bzw. Nachteile hat die Verwendung eines Zustandsautomaten zur Realisierung der seriellen Schnittstelle?

7. Projektübung - Signalgenerator

Ziel der Projektübung ist der Entwurf und die Implementierung eines Funktionsgenerators zur Erzeugung digitaler Signale variabler Signalform und variabler Frequenz. Die Signale werden auf einem FPGA erzeugt und können per DA-Wandler auf einem Messgerät angezeigt werden (Oszilloskop, Spektrumanalysator). In diesem Projekt beschränken wir uns auf die Simulation.

7.1. Funktionsprinzip

Die Funktion der Schaltung beruht auf der direkten Synthese der Signale (Direct Digital Synthesis, DDS). Bei dieser Methode wird die gewünschte Signalform in einem Speicher abgelegt und von dort aus ausgelesen und auf einen Digital-Analogwandler gegeben. Das Auslesen kann hierbei mit unterschiedlichen Schrittweiten geschehen, wodurch sich eine Variation der Frequenz des Signals ergibt. Die folgende Abbildung zeigt das Prinzip für eine harmonische Signalform.

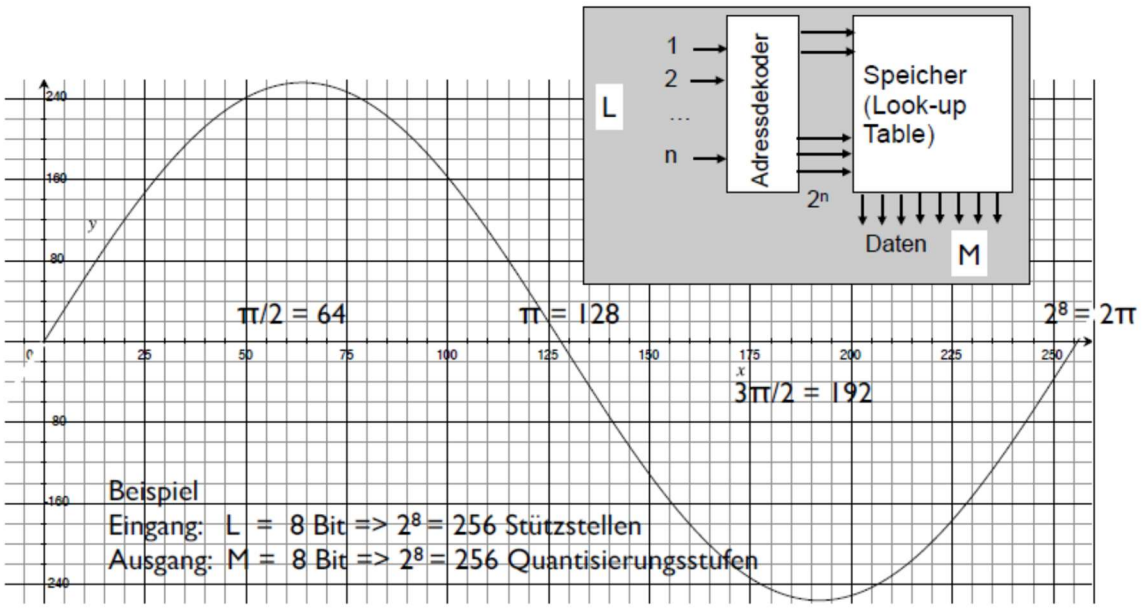


Bild 7.1 Look-up-Table mit Funktionswerten

Im Beispiel wurde das gewünschte Signal (Sinus) über eine Periode mit 256 Stützstellen gespeichert. Wenn man mit Hilfe eines Zählers alle Stützstellen der Reihe nach abfragt, ergibt sich die langsamste mit dieser Anordnung realisierbare Frequenz. Bei einer Taktrate von beispielsweise 20MHz dauert das Abspielen einer kompletten Periode 256 / 20 MHz = 12.8 µs. Die Frequenz des erzeugten Signals beträgt also 78,125 kHz.

Die mit dieser Methode maximal realisierbare Frequenz ergibt sich, wenn man pro Periode nur die minimal erforderliche Anzahl von 2 Stützstellen ausliest (also z. B. nur die Werte bei π/2 und 3π/2). Diese Grenzfrequenz ergibt sich bei einer Taktrate von 20 MHz also zu 10 MHz. Somit beträgt in diesem Beispiel der Umfang des realisierbaren Frequenzbereichs 78,125 kHz bis 10 MHz. Wegen der auf 256 begrenzten Anzahl der Stützstellen der Funktion sind Frequenzen in Vielfachen von f_d = 78,125 kHz realisierbar. Ausgehend von der Taktrate f_c bestimmt sich der Frequenzbereich zu $f_d = f_c / 2L$ und der Grenzfrequenz $f_g = f_c / 2$.

Aufgabe 7.1: Nehmen Sie an, die Eingangswortbreite der zum Speichern der Stützstellen verwendeten LUT (Look-up Table) beträgt 12 bit. Die Taktrate zum Auslesen der Stützstellen wird wiederum mit 20 MHz angenommen. Welchen Frequenzbereich kann die Schaltung abdecken?

Aufgabe 7.2: Nehmen Sie an, die Taktrate des FPGA-Bausteins beträgt 80 MHz, Sie möchten allerdings bei einer Grenzfrequenz von 10 MHz bleiben, und die gleiche Schaltung wie in Aufgabe 1.1 verwenden. Wie gehen Sie vor?

Die Berechnungen zeigen, dass sich Eingangswortbreite L des Speicherelements (LUT) nach der Anzahl der gewünschten Stützstellen richtet. Diese wiederum richten sich nach dem gewünschten Frequenzumfang der Schaltung. Die Ausgangswortbreite M des Speicherelements (LUT) richtet sich nach der gewünschten Güte des Ausgangssignals. Stehen nur 8-Bit D/A Wandler zur Verfügung, genügen auch 8-Bit Ausgangswortbreite. Für anspruchsvollere Aufgaben würde man mit 12-Bit oder 16-Bit AD-Wandlern arbeiten.

Zur Erzeugung der Signale in der beschriebenen Weise würde man den Funktionsspeicher (LUT) in geeigneter Weise ansteuern. Die Ansteuerung soll ein zyklisches Abfragen der Eingänge mit variabler Schrittweite ermöglichen. Außerdem ermöglicht die Ansteuerung auch das Abfragen von Phasenbezügen.

Aufgabe 7.3: Beschreiben Sie, wie Sie ein Signal mit halber Anzahl der Stützstellen (also doppelter minimaler Frequenz f_d) erzeugen? Welche Schrittweite ist hierfür erforderlich? Wie können Sie die Schrittweite variable gestalten?

Aufgabe 7.4: Beschreiben Sie, wie Sie mit Hilfe des Funktionsspeichers ein Kosinus-Signal erzeugen können. Wie lässt sich eine variable Phasenbeziehung bei vorgegebener Frequenz einstellen?

Folgendes Blockschaltbild beschreibt die komplette Signalkette, bestehend aus:

- einem Register für das Phaseninkrement (PIR, Phase Increment Register). Das Phaseninkrement beschreibt die Schrittweite, in der die Stützstellen der Funktion ausgelesen werden sollen.
- einer Schaltung zur Adressierung des Funktionsspeichers (LUT) mit der vorgegebenen Schrittweite, bestehend aus einem Addierer und einem Phasenregister• dem Funktionsspeicher (LUT)
- dem Digital-Analogwandler.

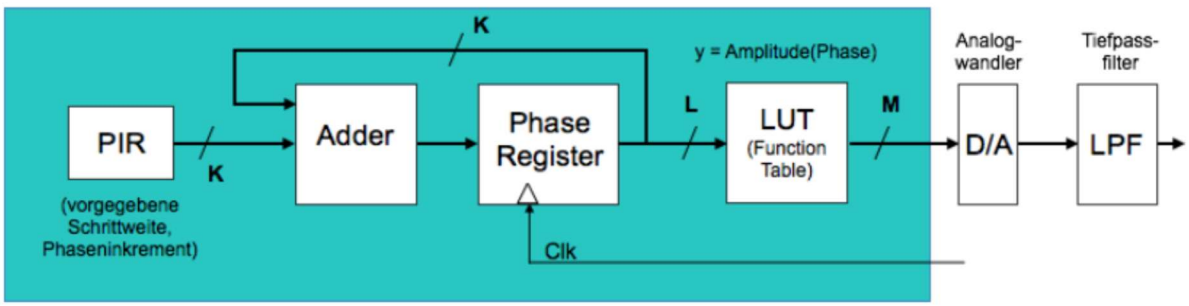


Bild 7.2 Blockschaltbild des Signalgenerators mit variabler Frequenz

Aufgabe 7.5: Erläutern Sie die Funktion der Schaltung. Hinweis: Nehmen Sie folgende Wortbreiten an: K = L = 12 Bit, M = 8 Bit. Geben Sie eine Schrittweite vor. Wie wird die zyklische Ansteuerung des Funktionsspeichers erreicht (Überlauf am Ende der Tabelle)?

Aufgabe 7.6: Die Wortbreite des Phasenregisters wird auf K = 24 Bit erhöht, wobei zur Adressierung des Funktionsspeichers weiterhin L = 12 Bits verwendet werden. Welche Änderung ergibt sich hierdurch?

Aufgabe 7.7: In Ergänzung zu Aufgabe 1.6: Würde es in der gegebenen Anordnung Sinn machen, die Wortbreite des Funktionsspeichers ebenfalls auf L = 24 Bit zu erhöhen? Begründen Sie Ihre Aussage.

Bemerkung: Mathematisch betrachtet nutzt man folgende Beziehung zwischen Frequenz und Phase: f = dφ /dt. Bei konstanter Frequenz schreitet die Phase linear fort. Je größer die Phasenänderung pro Zeiteinheit, desto höher ist also die Frequenz. Somit lässt sich durch das Phaseninkrement also die Frequenz definieren. Dieser Zusammenhang ist natürlich auch an der Funktionstabelle unmittelbar erkennbar.

Die in Abbildung 7.2 wiedergegebene Schaltung ermöglicht zwar die Einstellung der Frequenz (= Phaseninkrement) mit Hilfe des Phaseninkrement-Register (PIR). Die Schaltung ermöglicht jedoch nicht die Einstellung einer Phasenverschiebung, wie z.B. φ0 = 90o. Um bei vorgegebener Frequenz eine Phasenverschiebung (engl. phase offset) einzustellen, wird daher ein weiteres Register ergänzt. Dieses Register (Phase Offset Register, POS) erlaubt die Einstellung einer konstanten Phasenverschiebung unabhängig von der Frequenz.

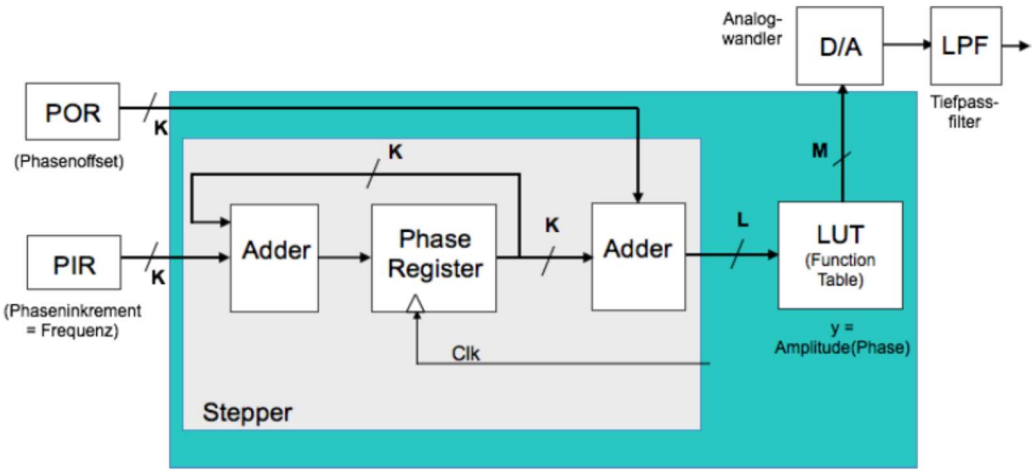


Bild 7.3 Blockschaltbild des Signalgenerators

Die Schaltung enthält somit folgende Funktionsblöcke: (1) ein Schaltwerk für Phaseninkrement und Phasenoffset (als „Stepper“ bezeichnet), (2) den gespeicherten Signalverlauf (Funktionstabelle, LUT). Bemerkung: Sofern nicht wirklich eine flexible Phase erwünscht ist, sondern beispielsweise nur zueinander orthogonale Signale (mit +/- 90o Phasenverschiebung), kann man auch mit zwei Speichern (LUT) für ein sinusförmigen und kosinusförmigen Signalverlauf arbeiten. Diese Signale addiert man dann in Abhängigkeit des zu modulierenden Eingangssignals.

Die Vorgabe des Phaseninkrements und des Phasenoffsets erlauben viele Möglichkeiten zur Erzeugung digitaler Signale, beispielsweise phasenmodulierte Signale durch definierte Phasensprünge, bzw. frequenzmodulierte Signale, Frequenzsprungverfahren, bzw. Signaldurchläufe durch den gesamten Frequenzbereich (signal sweeps). Solche Spezialfälle lassen sich durch eine

komplexere Beschaltung des Phaseninkrements und Phasenoffsets realisieren, wobei diese dann zeitlich veränderliche Größen werden.

Natürlich lassen sich mit sehr geringem Aufwand mit der in Abbildung 1.3 gezeigten Anordnung auch andere Signalformen als harmonische Signale erzeugen. Durch alternative Belegung des Funktionsspeichers sind beispielsweise Dreieck-Signale, Sägezahn, Rechteck und sonstige Signalverläufe realisierbar.

7.2. Schaltungsentwurf

Für den Signalgenerator soll folgendes Blockschaltbild verwendet werden. Für den Signalausgang stehen 8-Bit Analogwandler zur Verfügung. Die Schaltung wird mit 20 MHz Taktfrequenz betrieben. Eine Periode wird bzgl. der Vorgaben für die Frequenz (= Phaseninkrement) und den Phasenoffset in 220 Stützstellen unterteilt, d.h. die Wortbreite der Register und des Schaltwerks wird mit 20 Bit gewählt. Die Schaltung soll außerdem über einen Eingang für das Taktsignal (Clk), sowie über einen Eingang zum Rücksetzen (Reset) verfügen.

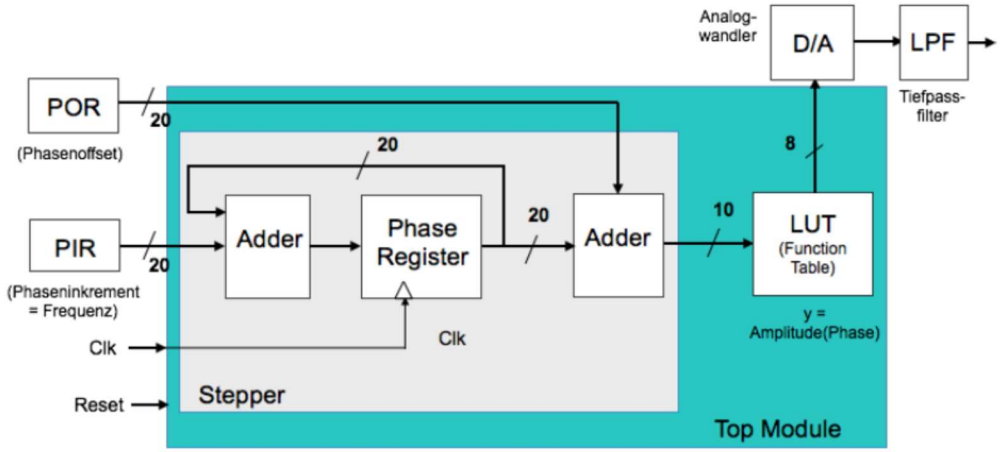


Bild 7.4 Blockschaltbild des Signalgenerators

Aufgabe 7.8: Wir groß ist der Frequenzumfang der Schaltung? Reicht die Schaltung in den Audiobereich? Welche Frequenzschritte sind möglich, ausgehend von der tiefsten Frequenz?

Aufgabe 7.9: Für den Eingang der Funktionstabelle (LUT) wird nur eine Wortbreite von 10 Bit verwendet (die oberen 10 Bits des Schaltwerks). Welche Einschränkungen ergeben sich hierdurch?

Für den Schaltungsentwurf ergeben sich aus dem Blockschaltbild folgende Module:

• das Schaltwerk für den Stepper (grau unterlegter Block)

• die Funktionstabelle (LUT)

• ein übergeordnetes Modul (Top Modul), das beide Module verbindet (grün unterlegter Block).

Diese drei Module sind nun als Schaltung zu entwerfen, d.h. in VHDL zu beschreiben. Für den Entwurf werden Muster vorgegeben, die Sie als Basis verwenden können

Aufgabe 7.10: Entwerfen Sie das Top Modul.

Als Vorlage können Sie die folgende HDL-Beschreibung verwenden. Analysieren Sie die Vorlage und ändern Sie gegebenenfalls nach Bedarf. Speichern Sie die Datei auf einem Verzeichnis zusammen mit den folgenden HDL-Beschreibungen im VHDL-Format (.vhdl) so, dass der HDLSimulator später direkt darauf zugreifen kann.

```
--- Top Module for DDS Generator (Direct Digital Synthesis)
--- VHDL
library ieee;
use ieee.std_logic_1164.all;
entity TOP is

      -- define K, L and M as constants (natural = positive integer)
      generic ( STEPPER_BITS : natural := 20; -- stepper is K=20 bits
                  LUT_BITS : natural := 10; -- LUT is L=10 bits
                  DAC_BITS : natural := 8); -- DAC is M=8 bits

      -- declare external ports of top module
      port (      Clk, RS : in std_logic; -- clock, reset
                  PIR : in std_logic_vector (STEPPER_BITS-1 downto 0);
                  POR : in std_logic_vector (STEPPER_BITS-1 downto 0);
                  DAC : out std_logic_vector(DAC_BITS-1 downto 0));
end TOP;

architecture RTL of TOP is

component Stepper is

      generic (STEPPER_BITS : natural;
                  LUT_BITS : natural);

      port (      Clk, RS : in std_logic;
                  PIR : in std_logic_vector (STEPPER_BITS-1 downto 0);
                  POR : in std_logic_vector (STEPPER_BITS-1 downto 0);
                  LUT : out std_logic_vector(LUT_BITS-1 downto 0));

end component Stepper;

component LookUpTable is

      generic (LUT_BITS : natural;
                  DAC_BITS : natural);

      port (      LUT : in std_logic_vector(LUT_BITS-1 downto 0);
                  DAC : out std_logic_vector(DAC_BITS-1 downto 0));

end component LookUpTable;

-- declare internal signal of Top module (from Stepper to LookUpTable)
signal stepperOut : std_logic_vector(LUT_BITS-1 downto 0);

-- connect Stepper to LookUpTable
begin

      STEP: Stepper generic map(STEPPER_BITS=>STEPPER_BITS,
      LUT_BITS=>LUT_BITS)

      port map(Clk=>Clk, RS=>RS, PIR=>PIR, POR=>POR, LUT=>stepperOut);
```

```
      LUTB: LookUpTable generic map(LUT_BITS=>LUT_BITS, DAC_BITS=>DAC_BITS)

      port map(LUT=>stepperOut, DAC=>DAC);

end RTL;
```

Aufgabe 7.11: Entwerfen Sie das Schaltwerk (Stepper).

Folgende HDL-Beschreibung können Sie als Vorlage verwenden. Analysieren Sie die Vorlage und ändern Sie gegebenenfalls. Speichern Sie die Datei zusammen mit den anderen zur Aufgabe gehörigen HDL-Beschreibungen auf einem Verzeichnis ab.

```
    --- Stepper Module for DDS Generator (Direct Digital Synthesis)
    --- VHDL
    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all; -- IEEE 1076.3 standard

    entity Stepper is
    generic (STEPPER_BITS : natural; -- K bits wide
           LUT_BITS : natural); -- L bits wide

    port ( Clk, RS : in std_logic;
           PIR : in std_logic_vector (STEPPER_BITS-1 downto 0);
           POR : in std_logic_vector (STEPPER_BITS-1 downto 0);
           LUT : out std_logic_vector(LUT_BITS-1 downto 0));

    -- internal port to Top
    end Stepper;

    architecture RTL of Stepper is
    -- store accumulated phase in Phase Register
    signal phaseReg : unsigned ((STEPPER_BITS -1) downto 0);
    -- phase output to LUT
    signal phaseOut : unsigned ((STEPPER_BITS -1) downto 0);

    begin
     process (Clk, RS)
     begin
           if RS ='1' then -- reset to initial state
                 phaseReg <= (others => '0');
                 phaseOut <= (others => '0');
           elsif rising_edge(Clk) then
                 -- increment phase
                 phaseReg <= phaseReg + unsigned(PIR);
                 -- add offset
                 phaseOut <= phaseReg + unsigned(POR);
           end if;
    end process;

    -- use top K-M bits of phaseOut to address loop up table
    LUT <= std_logic_vector(phaseOut((STEPPER_BITS-1) downto
    (STEPPER_BITS - LUT_BITS)));

    end RTL;
```

Aufgabe 7.12: Entwerfen Sie die Funktionstabelle (LUT).

Als Vorlage können Sie wiederum folgende HDL-Beschreibung verwenden. Analysieren Sie die Vorlage und ändern Sie sie gegebenenfalls. In der Vorlage fehlen noch die Funktionswerte für den Funktionsspeicher (LUT). Erzeugen Sie diese Werte mit der gewünschten Auflösung (8-Bit Werte) und mit der gewünschten Anzahl Stützstellen. Stellen Sie den Schaltungsentwurf fertig.

```
--- LookUpTable Module for DDS Generator (Direct Digital Synthesis)
--- VHDL
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity LookUpTable is

      generic (LUT_BITS : natural; -- L bits wide
                  DAC_BITS : natural); -- M bits wide
      port (      LUT : in std_logic_vector(LUT_BITS-1 downto 0);
                  DAC : out std_logic_vector(DAC_BITS-1 downto 0));

end LookUpTable;

architecture RTL of LookUpTable is
-- function table of 2**L samples, each M bits wide

type lut_array is array(0 to 2**LUT_BITS - 1) of std_logic_vector(DAC_BITS
- 1 downto 0);
-- fill function table with 2**M sample values
signal ltb : lut_array := (x"80",
                                        -- fill in sample values here
);

-- look-up function in table: y = amplitude(phase)
begin

DAC <= ltb(conv_integer(LUT));

end RTL;
```

7.3. Funktionale Verifikation

Übernehmen Sie die Schaltungsentwürfe für den Generator bestehen aus den drei Modulen aus Abschnitt 1.2 in den HDL-Simulator und übersetzen Sie den Schaltungsentwurf. Übersetzen Sie die Dateien zunächst einzeln. Arbeiten Sie bitte direkt auf dem Verzeichnis, auf dem Sie die HDL-Dateien abgelegt haben, so dass bei der Fehlerkorrektur dort der fehlerfreie Quellcode abgelegt bleibt.

Aufgabe 7.13: Übersetzen Sie den Schaltungsentwurf und beheben Sie alle ggf. auftretenden Fehler, die der Compiler meldet.

Überlegen Sie sich eine Methode, um den Schaltungsentwurf zu verifizieren. Hierzu benötigen Sie eine Testumgebung, die alle Eingangssignale des Top Moduls stimuliert und die Ausgangssignale aufnimmt.

Aufgabe 1.14: Entwerfen Sie eine Testumgebung für das Top-Modul.

Folgende HDL-Beschreibung können Sie als Vorlage verwenden. Analysieren Sie die Vorlage und ändern Sie gegebenenfalls. Speichern Sie die Datei zusammen mit den anderen zur Aufgabe gehörigen HDL-Beschreibungen auf einem Verzeichnis ab.

```
--- Testbench for Top Module of DDS Generator (VHDL)
library ieee;
use ieee.std_logic_1164.all;
entity test_TOP is
-- define K, L and M as constants
generic ( STEPPER_BITS : natural := 20; -- stepper is K=20
LUT_BITS : natural := 10; -- LUT is L=10 bits
DAC_BITS : natural := 8); -- DAC is M=8 bits
-- no external signals in test bench
end test_TOP;

architecture Behavioural of test_TOP is

component TOP is
      generic ( STEPPER_BITS : natural;
            LUT_BITS : natural;
            DAC_BITS : natural);
      port ( Clk, RS : in std_logic; -- clock, reset
            PIR : in std_logic_vector (STEPPER_BITS-1 downto 0);
            POR : in std_logic_vector (STEPPER_BITS-1 downto 0);
            DAC : out std_logic_vector(DAC_BITS-1 downto 0));
end component TOP;

-- define test signals
signal T_Clk : std_logic := '0';
signal T_RS : std_logic := '1';
signal T_PIR : std_logic_vector (STEPPER_BITS-1 downto 0):= (others =>
'0');
signal T_POR : std_logic_vector (STEPPER_BITS-1 downto 0):= (others =>
'0');
signal T_DAC : std_logic_vector(DAC_BITS-1 downto 0);

-- connect DUT to testbench
begin
DUT: TOP generic map(STEPPER_BITS => STEPPER_BITS,
                           LUT_BITS => LUT_BITS, DAC_BITS => DAC_BITS)

            port map(Clk => T_Clk, RS => T_RS, PIR => T_PIR,
                        POR => T_POR, DAC => T_DAC);

-- run tests
test: process
begin
      wait for 50 ns;
      T_RS <= '0';
      T_PIR <= b"_____________________";   -- insert test values for PIR
here
      T_POR <= b"_____________________";   -- insert test values for POR
here
      for j in 1 to 10000 loop
            T_Clk <= '0';
            wait for 25 ns;
            T_Clk <= '1';
            wait for 25 ns;
      end loop;
```

```
      wait;
end process;
end Behavioural;
```

Aufgabe 7.15: Testen Sie das Modul im Simulator. Geben Sie hierzu sinnvolle Werte für das Phaseninkrement und den Phasenoffset vor. Analysieren Sie die Ausgangssignale des Prüflings (DUT). Prüfen Sie die Ergebnisse auf Plausibilität.

Aufgabe 7.16: Dokumentieren Sie die Ergebnisse Ihrer Tests.

Anhang 1: Lösungen zu den Übungsaufgaben aus Kapitel 6

A.6.1 Code-Umsetzer

Eine Schaltung soll einen 4-Bit Gray Code in einen 4-Bit BCD-Code umsetzen gemäß folgender Wertetabelle.

Binärcode	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Gray-Code	0	1	3	2	6	7	5	4	C	D	F	E	A	B	9	8

Bild 1.1 Wertetabelle des Code-Umsetzers

Frage 6.1.1 (4 Punkte): Skizzieren Sie ein Blockschaltbild des Code-Umsetzers mit den benötigten Eingangssignalen und Ausgangssignalen.

Lösung:

Frage 6.1.2 (6 Punkte): Entwerfen Sie den Code-Umsetzer in VHDL. Hinweis: Verwenden Sie eine `case`-Anweisung.

Lösung:

```
--- Gray-to-Binary Decoder (VHDL)
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity GB_Decoder is
 port (      GC : in std_logic_vector (3 downto 0);
             BC : out std_logic_vector (3 downto 0));
end GB_Decoder;

architecture RTL of GB_Decoder is
begin
 process (GC)
 begin
       case GC is
             when x"0" => BC <= x"0";
             when x"1" => BC <= x"1";
             when x"3" => BC <= x"2";
             when x"2" => BC <= x"3";
             when x"6" => BC <= x"4";
             when x"7" => BC <= x"5";
             when x"5" => BC <= x"6";
             when x"4" => BC <= x"7";
             when x"C" => BC <= x"8";
```

```
                when x"D" => BC <= x"9";
                when x"F" => BC <= x"A";
                when x"E" => BC <= x"B";
                when x"A" => BC <= x"C";
                when x"B" => BC <= x"D";
                when x"9" => BC <= x"E";
                when x"8" => BC <= x"F";
        end case;
  end process;
end RTL;
```

Frage 6.1.3 (4 Punkte): Erweitern Sie die Schaltung so, dass die binär kodierten Daten taktsynchron abgetastet werden. Verwenden Sie hierzu ein Clock-Signal in Kombination mit einem Register. Skizzieren Sie ein Blockschaltbild.

Lösung:

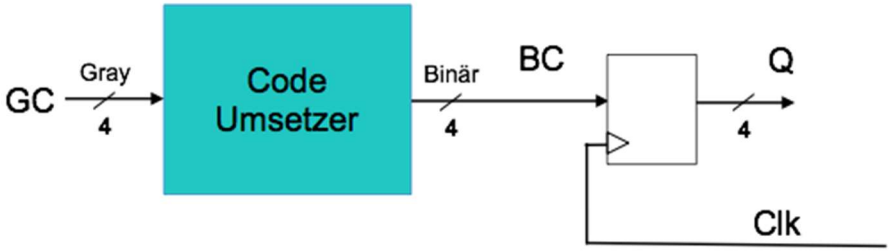


Bild 1.2 Code-Umsetzer mit Abtastung

Frage 6.1.4 (6 Punkte): Erweitern Sie die VHDL-Beschreibung um die Abtastung (Frage 6.1.3). Erläutern Sie Ihren Entwurf in einigen Stichworten.

Lösung:

```
--- Gray-to-Binary Decoder (VHDL)
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity GB_Decoder is
  port (      Clk: in std_logic;
              GC : in std_logic_vector (3 downto 0);
               Q : out std_logic_vector (3 downto 0));
end GB_Decoder;

architecture RTL of GB_Decoder is
  signal BC: std_logic_vector (3 downto 0);
begin
  decode: process (GC)
    begin
        case GC is
              when x"0" => BC <= x"0";
              when x"1" => BC <= x"1";
              when x"3" => BC <= x"2";
```

```
                when x"2" => BC <= x"3";
                when x"6" => BC <= x"4";
                when x"7" => BC <= x"5";
                when x"5" => BC <= x"6";
                when x"4" => BC <= x"7";
                when x"C" => BC <= x"8";
                when x"D" => BC <= x"9";
                when x"F" => BC <= x"A";
                when x"E" => BC <= x"B";
                when x"A" => BC <= x"C";
                when x"B" => BC <= x"D";
                when x"9" => BC <= x"E";
                when x"8" => BC <= x"F";
        end case;
  end process decode;

  sample: process (Clk)
    begin
        if rising_edge(Clk) then
              Q <= BC;
        end if;
  end process sample;
end RTL;
```

- Die Schaltung wird erweitert um ein 4-Bit breites Register, das zu den Abtastzeitpunkten die umkodierten Werte aufnimmt.
- In VHDL wird die Ergänzung durch einen zusätzlichen, taktabhängigen Prozess dargestellt (Prozess mit Label „sample“ im HDL-Text).
- Dem taktabhängigen Prozess wird der binär kodierte Wert als internes Signal übergeben.

A.6.2. Zustandsautomat Variante 1

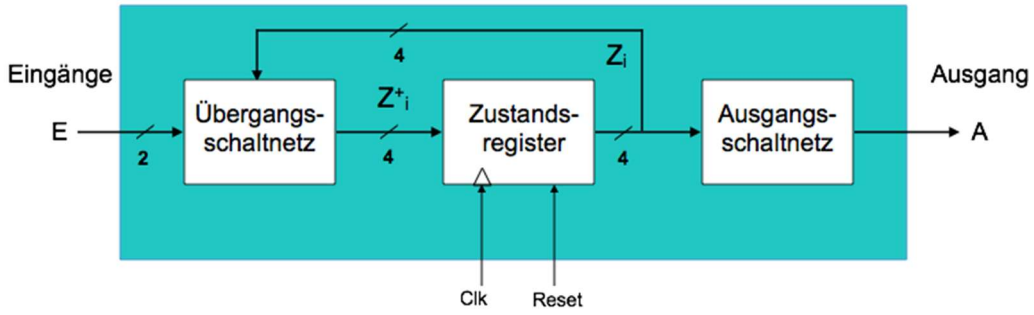
Folgender HDL-Text beschreibt einen Zustandsautomaten.

```
--- Finite State Machine (VHDL)
library ieee ;
use ieee.std_logic_1164.all;

entity sequence_logic is
 port(       E:          in std_logic_vector(1 downto 0);
                  Clk:        in std_logic;
                  Reset:      in std_logic;
                  A:          out std_logic);
end sequence_logic;

architecture FSM of sequence_logic is

    type state_type is (Z0, Z1, Z2, Z3);
    signal next_state, current_state: state_type;
begin
```

```
update_state_reg: process(Clk, Reset)
  begin
       if (Reset='1') then
         current_state <= Z0;
       elsif rising_edge(Clk) then
         current_state <= next_state;
       end if;
end process update_state_reg;

comb_logic_next: process(current_state, E)
  begin
       case current_state is
         when Z0 =>
               if    E = "01" then   next_state <= Z1;
               else                  next_state <= Z0;
               end if;
        when Z1 =>
               if    E = "11" then   next_state <= Z2;
               elsif E = "01" then   next_state <= Z1;
               else                  next_state <= Z0;
               end if;
       when Z2 =>
               if    E = "10" then   next_state <= Z3;
               elsif E = "01" then   next_state <= Z1;
               else                  next_state <= Z0;
               end if;
       when Z3 =>
               if    E = "01" then   next_state <= Z1;
               else                  next_state <= Z0;
               end if;
       when others => next_state <= Z0;
      end case;
end process comb_logic_next;

comb_logic_out: process(current_state)
  begin
 case current_state is
       when Z3 => A <= '1';
       when others => A <= '0';
 end case;
end process comb_logic_out;

end FSM;
```

Frage 6.2.1 (6 Punkte): Erstellen Sie ein Blockdiagramm des Automaten.

Lösung:



Frage 6.2.2 (8 Punkte): Erläutern Sie die Funktion des Automaten gemäß Blockdiagramm unter Verwendung der Zustandsgleichungen (siehe Abschnitt 5.5 im Manuskript). Erläutern Sie die Zusammenhänge der Komponenten mit den Prozessen in der HDL-Beschreibung.

Lösung:

(1) Zustandsgleichungen: $Z+ = F1\,(E,Z)$ Folgezustand (Gleichung 1)

$A = F_2\,(Z)$ Ausgangssignal (Gleichung 2)

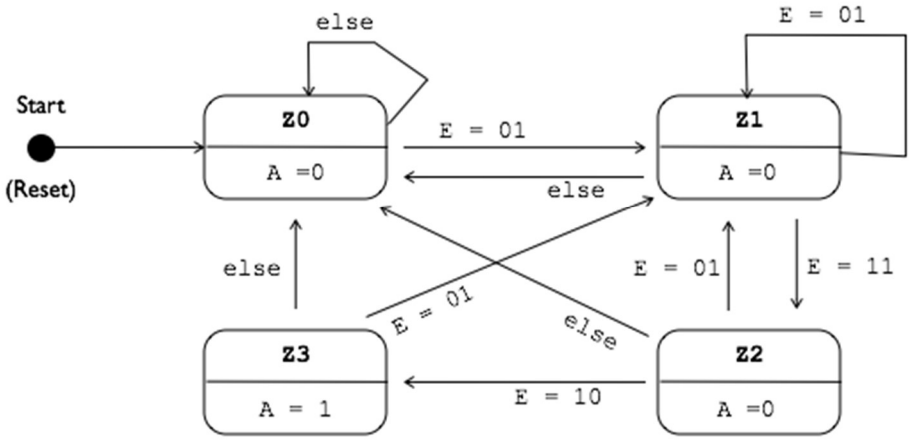
(2) Gleichung 1 (Folgezustand) beschreibt das Übergangsschaltnetz. In der HDL-Beschreibung ist das Übergangsschaltnetz durch den Prozess „comb_logic_next“ abgebildet, dessen Sensitivitätsbereich der aktuelle Zustand und das Eingangssignal bildet.

(3) Gleichung 2 (Ausgangssignal) beschreibt das Ausgangsschaltnetz. In der HDL-Beschreibung ist das Ausgangsschaltnetz durch den Prozess „com_logic_out“ abgebildet, dessen Sensitivitätsbereich den aktuellen Zustand beinhaltet.

(4) Das Zustandsregister ist im HDL-Code im Prozess „update_state_reg“ enthalten. Dieser Prozess reagiert auf den Takt (Clk), sowie auf das Reset-Signal (Reset). Mit jeder steigenden Taktflanke wird der Zustand aktualisiert.

Frage 6.2.3 (8 Punkte): Skizzieren Sie ein Zustandsdiagramm (inklusive des initialen Zustands). Erläutern Sie die Funktion des Zustandsautomaten mit Hilfe des Diagramms.

Lösung:



Funktion des Automaten: Der Automat schaltet genau dann sukzessive bis zum Zustand Z3, wenn es eine Folge der Eingangssignale E = 01, 11, 10 gibt. In diesem Fall wird das Ausgangssignal gesetzt (A = 1). In allen anderen Fällen fällt der Automat zurück in den Zustand Z0 mit folgenden Ausnahmen: E = 01 in Z1, Z2 oder Z3 führt zum Verbleib in Z1 bzw. zum Übergang nach Z1.

Frage 6.2.4 (6 Punkte): Erstellen Sie eine Zustands-Übergangstabelle (State-Event Table).

Lösung:

aktueller Zustand	Ereignis (Eingangssignal)	Folgezustand	Ausgangssignal
Z0	E = 01	Z1	A = 0
Z0	else	Z0	A = 0
Z1	E = 11	Z2	A = 0
Z1	E = 01	Z1	A = 0
Z1	else	Z0	A = 0
Z2	E = 10	Z3	A = 0
Z2	E = 01	Z1	A = 0
Z2	else	Z0	A = 0
Z3	E = 01	Z1	A = 1
Z3	else	Z0	A = 1

Frage 6.2.5 (8 Punkte): Testumgebung. Erstellen Sie ein Konzept für eine Testumgebung des Automaten (in Worten, kein HDL-Text). In welchen Schritten gehen Sie vor (welche Funktionsblöcke bzw. welchen Ablauf hätte ein Testprogramm)? Nach welchen Kriterien können Sie den Automaten testen? Welche Tests sind im speziellen Fall sinnvoll?

(1) Lösung Testschritte:

- Komponente des Prüflings deklarieren
- Testsignale definieren (T_E, T_Clk, T_Reset, T_A)
- Prüfling einspannen (Port map des Device Under Test / DUT)
- Testprozesse starten (Clk, Reset, Transitionen)
- Ausgänge und Transitionen überprüfen

(2) Lösung Kriterien: Für einen Test des Zustandsautomaten sind unterschiedliche Fragestellungen relevant:

- Funktioniert das Rücksetzen (Reset)?
- Schaltet der Automat die in der Zustandsfolgetabelle geforderten geforderten Übergänge zuverlässig durch?
- Lässt sich der Automat im jeweiligen Zustand durch andere Ereignisse stören?

- Funktioniert das Rücksetzen aus jedem beliebigen Zustand?

(3) Lösung Kriterien im speziellen Fall: Eingangsignale aus einer zufälligen Folge binärer Werte wählen. Hierbei sollte eine Sequenz 01, 11, 10 erkannt werden.

Frage 6.2.6 (8 Punkte): Zustandskodierung. Die Zustände des Automaten wurden im HDL-Text wir folgt definiert.

```
type state_type is (Z0, Z1, Z2, Z3);
signal next_state, current_state: state_type;
```

Diese Beschreibung enthält keine Vorgaben für die Synthese der Zustände. Die Implementierung ist somit den Synthesewerkzeugen überlassen. Folgende Varianten sollen nähere Vorgaben über die Zustandskodierung geben.

```
--- Variante 1
type state_type is (Z0, Z1, Z2, Z3);
attribute ENUM_ENCODING: String;
attribute ENUM_ENCODING of state_type: type is "00 01 10 11";
signal next_state, current_state: state_type;
```

```
--- Variante 2
subtype state_type is std_logic_vector (0 to 3);
constant Z0: state_type := "1000"
constant Z1: state_type := "0100"
constant Z2: state_type := "0010"
constant Z3: state_type := "0001"
signal next_state, current_state: state_type;
```

Welche Möglichkeiten zur Zustandskodierung gibt es in der ursprünglichen HDL-Beschreibung, die die Implementierung dem Synthesewerkzeug überlässt? Interpretieren Sie die beiden alternativen Varianten. Welche Vorgaben werden für die Zustandskodierung gegeben? Beschreiben Sie die Unterschiede in der Realisierung.

Lösung:

(1) In der ursprünglichen Vorgabe ist es den Synthesewerkzeugen überlassen, wie die Zustände kodiert werden. Mögliche Interpretationen wären eine mit 2 Bits kodierte Implemen-tierung der Zustände (zwei Zustandsregister), bzw. eine mit 4 Bits kodierte Implementierung (ein eigenes Zustandsregister pro Zustand).

(2) Variante 1 gibt eine Enumeration mit 2-Bit Codes pro Zustand vor. Variante 2 definiert Konstanten für die Zustände mit einer Breite von 4 Bit, wobei jeweils ein Bit einem Zustand zugeordnet ist (d.h. 4 Bit breite Zustandsvektoren).

(3) Variante 1: Kodierung mit 2 Bits (d.h. zwei Zustandsregister), der Enumeration "11 10 01 00" folgend. Variante 2: Kodierung mit einem Bit pro Zustand (ein Zustandsregister pro Zustand) gemäß den vorgegebenen Konstanten. Variante 1 ist effizienter (weniger Register), Variante 2 ist leichter nachvollziehbar (eigenes Register pro Zustand).

A.6.3 FIR Filter

Die Faltung einer zeitdiskreten Funktion x(i) mit der Impulsantwort h_k wird durch die Faltungssumme

$$y[n] = \sum h_k \cdot x[n-k] \qquad (3.1)$$

beschrieben, wobei der Index k über alle vorhandenen Stützstellen h_k verläuft. Für eine Impulsantwort mit insgesamt 5 Stützstellen ergibt sich folgende Gleichung.

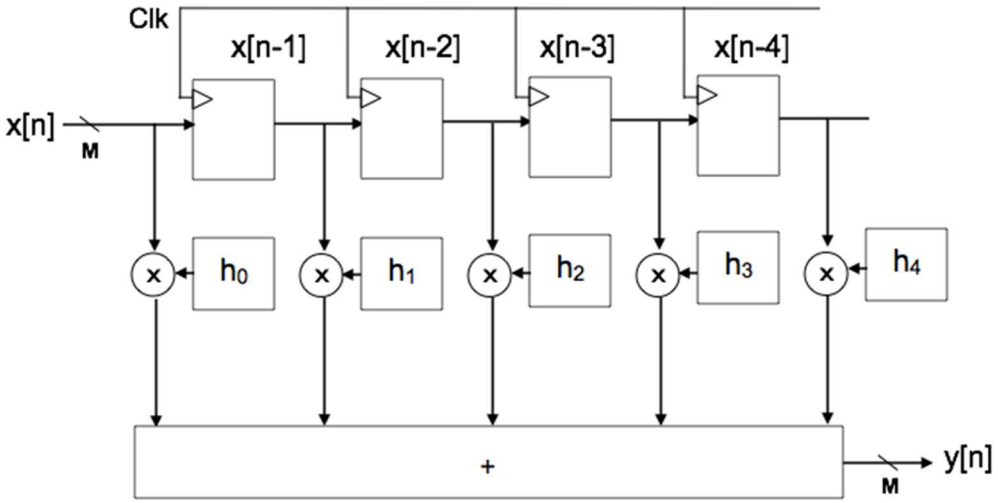
$$y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3] + h_4 \cdot x[n-4] \qquad (3.2)$$

Frage 6.3.1 (4 Punkte): Erläutern Sie Gleichung (3.2). Wie wirken die Stützstellen der Impuls-antwort (Filterkoeffizienten) auf die Werte der Eingangswerte x[n-k]?

Lösung: Die Filterkoeffizienten h_k wirken auf die vergangenen Werte des Eingangssignals x[n-k]. Das Ausgangssignal y[n] ergibt sich somit aus der Summe des mit dem Koeffizienten h_0 gewichteten aktuellen Wertes des Eingangssignals x[n], sowie der mit den Koeffizienten h_k gewichteten vergangenen Werte des Eingangssignals x[n-k].

Frage 6.3.2 (6 Punkte): Geben Sie ein Blockschaltbild zur Realisierung des Filters aus Gleichung (3.2) an. Verwenden Sie Register, Addierer und Multiplizierer.

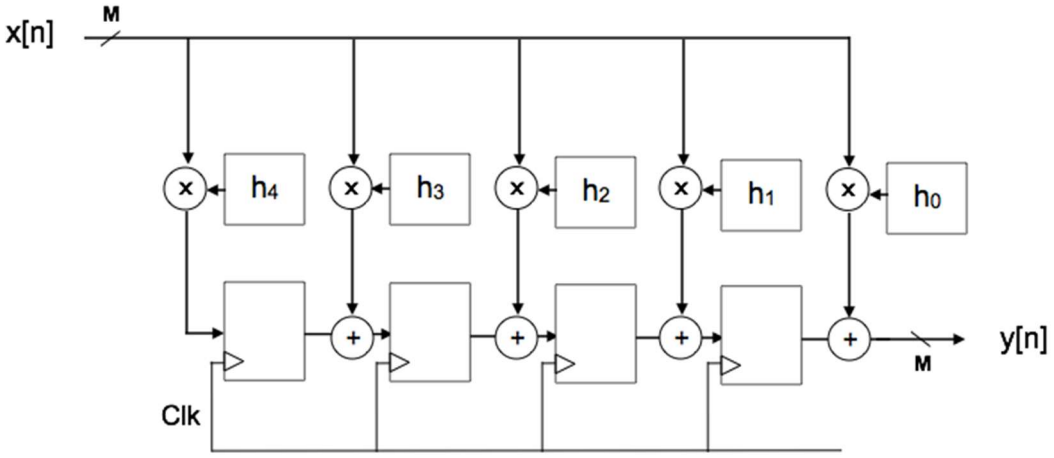
Lösung: Die vergangenen Werte x[n-k] lassen sich mit Hilfe eines Schieberegisters realisieren, wobei mit jedem Schritt (Takt) der jeweils aktuelle Wert x[n] ergänzt wird, und der vergangene Wert x[n-k-1] herausfällt. Blockschaltbild:



Frage 6.3.3 (6 Punkte): Die Werte der Eingangsfunktion x[n-k], sowie die Filterkoeffizienten h_k seien als 12 Bit Festkommazahlen gegeben. Das Ausgangssignal y[n-k] sei ebenfalls als 12 Bit Festkommazahl realisiert. Geben Sie die Wortbreiten im Blockschaltbild so vor, dass innerhalb der Schaltung keine Rundungsfehler auftreten. Begründen Sie Ihre Entscheidung.

Lösung: Bei beliebig vielen Filterkoeffizienten sollte die Wortbreite nach der Multiplikation so groß wie die Summe der Wortbreiten des Eingangssignals und der Filterkoeffizienten sein, in diesem Fall also 24 Bit. Im speziellen Fall mit nur 5 Filterkoeffizienten und einer Ausgangswortbreite von 12 Bit können bei der Addition höchstens 4 Bit hinter der 12-ten Stelle zum Ergebnis beitragen, d.h. es genügt nach der Multiplikation eine Wortbreite von 16 Bit.

Frage 6.3.4 (8 Punkte): Folgendes Blockschaltbild beschreibt eine alternative Realisierung des Filters. Weisen Sie die Äquivalenz zu Ihrem Blockschaltbild bzw. zu Gleichung (3.2) nach. Vergleichen Sie diese Implementierung mit Ihrem Blockschaltbild (Vorteile, Nachteile). Welche Wortbreiten sind erforderlich?



(1) Nachweis der Äquivalenz zum Blockschaltbild aus Aufgabe 3.3: Durch Rekonstruktion des Ausgangssignals y[n] aus dem Schaltbild erhält man die Gleichung

$$y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3] + h_4 \cdot x[n-4]$$

d.h. Gleichung (3.2). Somit realisiert die Schaltung das Filter.

(2) Im Blockschaltbild nach Aufgabe 3.3 ist die Realisierung der Blöcke zur Addition problematisch. Um alle Produkte zu addieren, wären laufzeitkompensierte Ketten von Addierwerken nötig, d.h. wiederum der Einsatz von Registern. Im oben abgebildeten Blockschaltbild (transponierte Form zu Aufgabe 6.3.3) besteht dieses Problem nicht, da statt der Eingangssignale die Produktterme in Registern gespeichert und sukzessive kumuliert werden.

(3) Wortbreiten: Für Eingangssignale mit Wortbreite von M Bits und Filterkoeffizienten mit Wortbreite von M Bits ist nach der Multiplikation ist nach der Multiplikation eine Wortbreite von 2 M erforderlich. Diese Wortbreite muss auch in den Registern vorgehalten werden. Nach Bildung der Summen kann das Ausgangssignal auf die obersten 12 Bits reduziert werden.

Frage 6.3.5 (6 Punkte): Folgender HDL-Text beschreibt ein digitales Filter. Analysieren Sie die Beschreibung und erläutern Sie den Entwurf (Struktur, Funktionsblöcke, Funktionen, ...).

```
--- FIR Filter (VHDL), simplified form without reset

   library IEEE;                                --- part #1
     use IEEE.std_logic_1164.all;
     use IEEE.numeric_std.all;

   entity FIR_4 is                        --- part #2
    port (Clk    : in  std_logic;         -- clock
          XN     : in  std_logic_vector (11 downto 0);
          YN     : out std_logic_vector (11 downto 0));
   end FIR_4;

   architecture RTL of FIR_4 is           --- part #3
                                          --- part #3.1
     type COEFF_TYPE is array (0 to 4) of signed (11 downto 0);
     type PRODUCT_TYPE is array (0 to 4) of signed (23 downto 0);
     type SUM_TYPE is array (0 to 4) of signed (23 downto 0);
```

```
    signal COEFF    : COEFF_TYPE;     -- coefficients array
    signal PRODUCT  : PRODUCT_TYPE;   -- product array
    signal SUM            : SUM_TYPE;       -- sum array

    signal XD       : signed (11 downto 0); -- register for X[0]
    signal lastsum : std_logic_vector (23 downto 0);
    begin                                   --- part #3.2
        COEFF(0) <= "010011011101";         --  0,60769
        COEFF(1) <= "001110101110";         --  0,45992
        COEFF(2) <= "000100100011";         --  0,14212
        COEFF(3) <= "100011000010";         -- -0,09475
        COEFF(4) <= "100011101011";         -- -0,11498

    new_sample : process (Clk)        --- part #3.3
      begin
        if rising_edge(Clk) then
              XD <= signed(XN(11 downto 0));
        end if;
    end process new_sample;

    multiply : process (COEFF, XD)    --- part #3.4
      variable MUL : signed (23 downto 0);
      begin
        for i in 0 to 4 loop
              MUL := XD * COEFF(i);
              PRODUCT(i) <= MUL;
        end loop;
    end process multiply;

    add_stages : process (CLK)        --- part #3.5
      begin
        if rising_edge(Clk) then
         SUM (4) <= PRODUCT (4);
              for i in 0 to 3 loop
                    SUM(i) <= SUM(i+1) + PRODUCT(i);
              end loop;
        end if;
    end process add_stages;

   lastsum <= std_logic_vector (SUM(0));          --- part #3.6
   YN <= lastsum (23 downto 12);

  end RTL;
```

(1) Lösung:

- part #1 - Bibliotheken deklarieren
- part #2 - Eingänge des Filters (Clk, x[n]), Ausgang y[n]

- part #3.1 - Register für Zwischenergebnisse: COEFF, PRODUCT, SUM enthalten jeweils 5 Speicherplätze für die Filterkoeffizienten h_0 bis h_4 (COEFF), Zwischenprodukte x[i] * h_i, und Zwischensummen (h_i * x[i] + h_{i-1} * x[i-1])
- part #3.2 - Festlegung der Filterkoeffizienten
- part #3.3 bis 3.5: parallele Prozesse; #3.3 Übernahme des aktuellen Wertes x[n] in ein internes Signal XD; #3.4 Berechnung der Zwischenprodukte; #3.5 Berechnung der Zwischensummen durch Addition der benachbarten Stufen und weiterschieben der Ergebnisse nach rechts (siehe Blockschaltbild in Frage 6.3.4);
- part #3.6: Zuweisung der letzten Stufe der Zwischensumme zum aktuellen Ausgangssignal y[n].

Frage 6.3.6 (6 Punkte): Erstellen Sie ein Konzept für ein Testprogramm des Filters (in Worten). In welchen Schritten gehen Sie vor (welche Funktionsblöcke bzw. welchen Ablauf hätte das Testprogramm)? Nach welchen Kriterien können Sie das Filter testen?

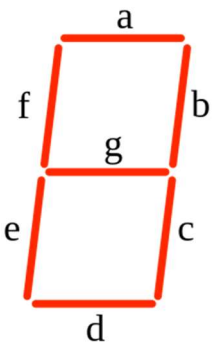
(1) Lösung Testschritte:

- Komponente des Prüflings deklarieren
- Testsignale definieren (Eingangssignal x[n])
- Prüfling einspannen (Port map des Device Under Test / DUT)
- Testprozesse starten
- Ausgangssignal y[n] überprüfen

(2) Lösung Kriterien: Für einen Test des Filters sind aus der Systemtheorie bekannte Testsignale sinnvoll, z.B. ein Delta-Impuls zum Zeitpunkt Null als Eingangssignal (gibt am Ausgang die Impulsantwort wieder, d.h. also die Filterkoeffizienten), bzw. Rauschen (Pseudonoise) als Eingangssignal und Analyse des Betragsspektrums des Ausgangssignals (Betrag der Übertragungsfunktion).

A.6.4 Dekoder für Segmentanzeige

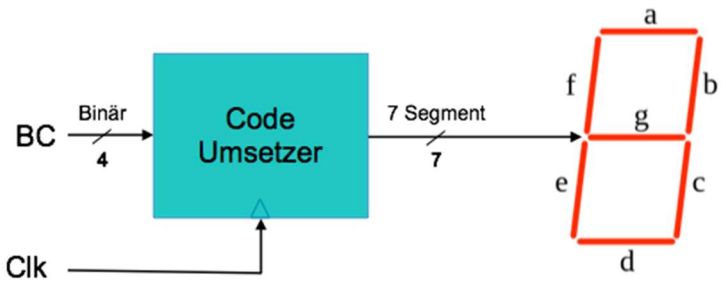
Eine Schaltung soll einen 4-Bit BCD-Code umsetzen zur Ansteuerung einer Anzeige mit 7 Segmenten, wie in der folgenden Abbildung gezeigt.



7 Segmentanzeige

Frage 6.4.1 (4 Punkte): Skizzieren Sie ein Blockschaltbild des Dekoders. Hinweis: Der Dekoder soll als getaktete Logik ausgeführt werden.

Lösung:



Frage 6.4.2 (4 Punkte): Erstellen Sie eine Wertetabelle zur Anzeige der Ziffern 0 bis 9.

Lösung:

Binärcode	7 Segment Code						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Frage 6.4.3 (8 Punkte): Entwerfen Sie den Code-Umsetzer in VHDL. Hinweis: Verwenden Sie eine `case`-Anweisung.

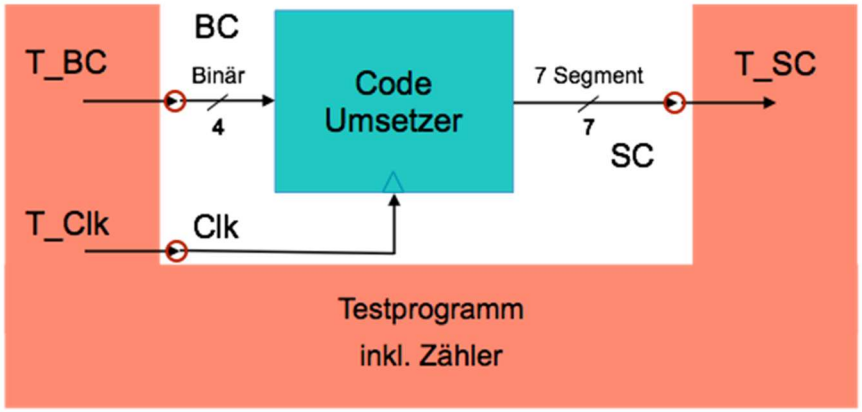
Lösung:

```
--- Binary-to-7-Segment Decoder (VHDL)
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity SEG7_Decoder is
 port (      Clk: in std_logic;
             BC : in  std_logic_vector (3 downto 0);
             SC : out std_logic_vector (6 downto 0));
end SEG7_Decoder;

architecture RTL of SEG7_Decoder is

signal XD          : std_logic_vector (6 downto 0);

begin
 process (Clk, BC)
 begin
       if rising_edge(Clk) then
       case BC is
             when x"0" => XD <= "1111110";
             when x"1" => XD <= "0110000";
             when x"2" => XD <= "1101101";
```

```
                when x"3" => XD <= "1111001";
                when x"4" => XD <= "0110011";
                when x"5" => XD <= "1011011";
                when x"6" => XD <= "1011111";
                when x"7" => XD <= "1110000";
                when x"8" => XD <= "1111111";
                when x"9" => XD <= "1111011";
                when others => XD <= "1001111"; -- E: Error
        end case;
        end if;
  end process;

  SC <= XD;

end RTL;
```

Frage 6.4.4 (8 Punkte): Entwerfen Sie einen Testprozess, der das BCD-Signal durch einen Zähler erzeugt. Skizzieren Sie Testumgebung und Prüfling als Blockschaltbild, aus dem die benötigten Testsignale hervorgehen. Skizzieren Sie den Testprozess in VHDL (Hinweis: bitte nur den Testprozess in einigen Zeilen, nicht die komplette Testumgebung).

Lösung: (1) Blockschaltbild



(2) VHDL Testprozess:

```
--- at declaration of test signals
signal T_Clk: std_logic := '0';
signal T_BC : std_logic_vector (3 downto 0) := (others => '1');

signal Qin: std_logic_vector (3 downto 0);  -- internal counter

--- at test process
 counter: process (T_Clk)
 begin
        if rising_edge(T_Clk) then
                    Qin <= (Qin + 1);  -- increment counter
        end if;
        end process counter;
        T_BC <= Qin;
```

A.6.5 Zustandsautomat Variante 2

Folgender HDL-Text beschreibt einen Zustandsautomaten.

```
--- Finite State Machine (VHDL)
library ieee ;
use ieee.std_logic_1164.all;

entity sequence_logic is
 port(       E:          in std_logic_vector(1 downto 0);
                  Clk:        in std_logic;
                  Reset:      in std_logic;
                  A:          out std_logic);
end sequence_logic;

architecture FSM of sequence_logic is

    type state_type is (Z0, Z1, Z2);
    signal next_state, current_state: state_type;

begin

update_state_register: process(Clk, Reset)       -- process #1
  begin
       if (Reset='1') then
         current_state <= Z0;
       elsif rising_edge(Clk) then
         current_state <= next_state;
       end if;
end process update_state_register;

logic_next_state: process(E, current_state)      -- process #2
  begin
       case current_state is
         when Z0 =>
              if    E = "01" then   next_state <= Z1;
              else                  next_state <= Z0;
              end if;
   when Z1 =>
              if    E = "11" then   next_state <= Z2;
              elsif E = "01" then   next_state <= Z1;
        else                  next_state <= Z0;
              end if;
   when Z2 =>
              if    E = "01" then   next_state <= Z1;
        else                  next_state <= Z0;
              end if;
   when others => next_state <= Z0;
```

```
        end case;
end process logic_next_state;

logic_out: process(E, current_state)              -- process #3
  begin
 if ((current_state = Z2) and (E = "10")) then    A <= '1';
 else                                             A <= '0';
 end if;
end process logic_out;

end FSM;
```

Frage 6.5.1 (8 Punkte): Erläutern Sie die Funktion des Automaten. Welche Rolle spielen die Prozesse innerhalb des Automaten? Vergleichen Sie die Rolle der Prozesse mit den Zustandsgleichungen aus dem Manuskript (siehe Abschnitt 5.5, Gleichungen 5.1 und 5.2). Wie lauten die Zustandsgleichungen dieses Automaten?

Lösung:

(1) Prozess 1 (update_state_register) aktualisiert das Zustandsregister. Dieser Prozess reagiert auf den Takt (Clk), sowie auf das Reset-Signal (Reset). Mit jeder steigenden Taktflanke wird der aktuelle Zustand übernommen.

(2) Prozess 2 (logic_next_state) definiert den Folgezustand in Abhängigkeit des aktuellen Zustands, sowie des Eingangssignals. Aktueller Zustand und Eingangssignal stehen daher im Sensitivitätsbereich des Prozesses.

(3) Prozess 3 (logic_out) reagiert auf das Eingangssignal und den aktuellen Zustand. In Abhängigkeit dieser Größen schaltet der Prozess das Ausgangssignal.

(4) Die Zustandsgleichungen aus dem Manuskript lauten:

$Z+ = F1\,(E,Z)$ Folgezustand (Gleichung 1)

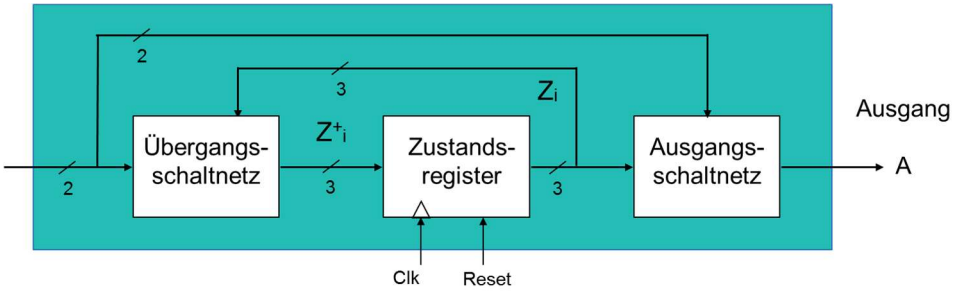
$A = F_2\,(Z)$ Ausgangssignal (Gleichung 2)

Gleichung 1 entspricht hierbei dem Prozess 2 aus der HDL-Beschreibung des Zustandsautomaten. Das Ausgangssignal erzeugt der Automat anders als in Gleichung 2 nicht nur aus dem aktuellen Zustand, sondern aus dem aktuellen Zustand und dem Eingangssignal. Für diesen Automaten lautet die zweite Zustandsgleichung also:

$A = F_2\,(E, Z)$ Ausgangssignal (Gleichung 2).

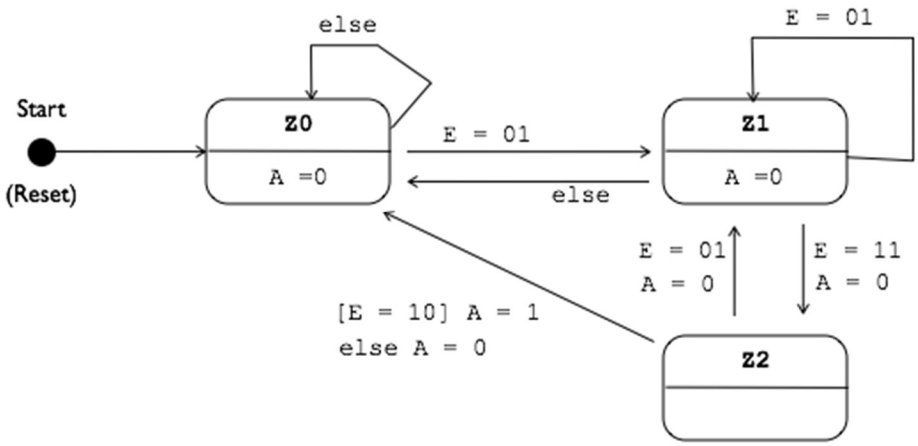
Frage 6.5.2 (6 Punkte): Erstellen Sie ein Blockdiagramm des Automaten.

Lösung:



Frage 6.5.3 (8 Punkte): Skizzieren Sie ein Zustandsdiagramm (inklusive des initialen Zustands). Erläutern Sie die Funktion des Zustandsautomaten mit Hilfe des Diagramms.

Lösung:



Funktion des Automaten: Der Automat schaltet genau dann sukzessive durch die Zustände Z0, Z1, Z2, Z0, wenn es eine Folge der Eingangssignale E = 01, 11, 10 gibt. In diesem Fall wird das Ausgangssignal gesetzt (A = 1), sobald der Zustand Z2 verlassen wird. In allen anderen Fällen fällt der Automat zurück in den Zustand Z0 mit folgenden Ausnahmen: E = 01 in Z1 (bzw. in Z2) führt zum Verbleib in Z1 (bzw. zum Rückfallen auf Z1).

Frage 6.5.4 (6 Punkte): Erstellen Sie eine Zustands-Übergangstabelle (State-Event Table).

Lösung:

aktueller Zustand	Ereignis (Eingangssignal)	Folgezustand	Ausgangssignal
Z0	E = 01	Z1	A = 0
Z0	else	Z0	A = 0
Z1	E = 11	Z2	A = 0
Z1	E = 01	Z1	A = 0
Z1	else	Z0	A = 0
Z2	E = 10	Z0	A = 1
Z2	else	Z0	A = 0

Frage 6.5.5 (8 Punkte): Testumgebung. Erstellen Sie ein Konzept für eine Testumgebung des Automaten (in Worten, kein HDL-Text). In welchen Schritten gehen Sie vor (welche

Funktionsblöcke bzw. welchen Ablauf hätte ein Testprogramm)? Nach welchen Kriterien können Sie den Automaten testen? Welche Tests sind im speziellen Fall sinnvoll?

(1) Lösung Testschritte:

- Komponente des Prüflings deklarieren
- Testsignale definieren (T_E, T_Clk, T_Reset, T_A)
- Prüfling einspannen (Port map des Device Under Test / DUT)
- Testprozesse starten (Clk, Reset, Transitionen)
- Ausgänge und Transitionen überprüfen

(2) Lösung Kriterien: Für einen Test des Zustandsautomaten sind unterschiedliche Fragestellungen relevant:

- Funktioniert das Rücksetzen (Reset)?
- Schaltet der Automat die in der Zustandsfolgetabelle geforderten geforderten Übergänge zuverlässig durch?
- Lässt sich der Automat im jeweiligen Zustand durch andere Ereignisse stören?
- Funktioniert das Rücksetzen aus jedem beliebigen Zustand?

(3) Lösung Kriterien im speziellen Fall: Eingangsignale aus einer zufälligen Folge binärer Werte wählen. Hierbei sollte eine Sequenz 01, 11, 10 erkannt werden.

Frage 6.5.6 (8 Punkte): Zusammenfassung beider Schaltnetze. Führen Sie die Prozesse 2 und 3 zu einem gemeinsamen Prozess zusammen und skizzieren Sie die diesbezügliche HDL-Beschreibung (nur den Prozess). Bewerten Sie die Vorteile und Nachteile dieser Realisierung mit der vorgegebenen Implementierung.

Lösung:

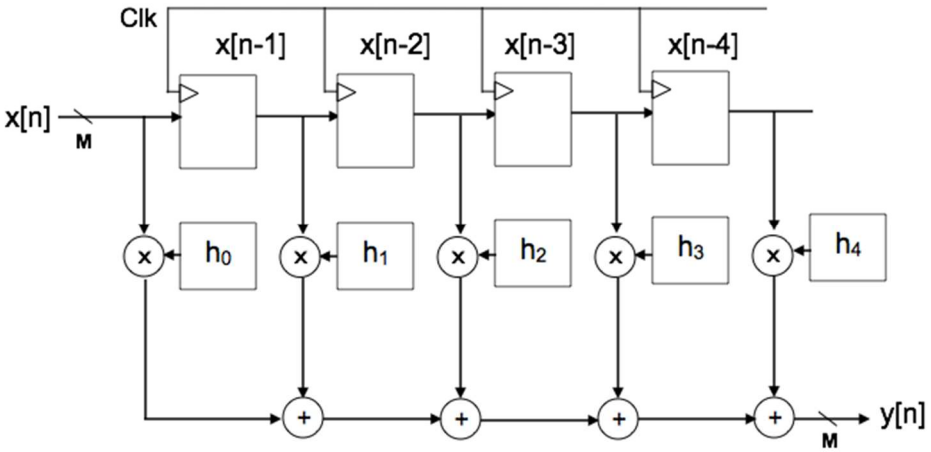
```
logic_next_state_and_out: process(E, current_state)
  begin
 case current_state is
   when Z0 => A <= '0';
              if     E = "01" then   next_state <= Z1;
              else                   next_state <= Z0;
              end if;
   when Z1 => A <= '0';
               if    E = "11" then   next_state <= Z2;
               elsif E = "01" then   next_state <= Z1;
         else                  next_state <= Z0;
               end if;
   when Z2 =>
               if E = "01" then      next_state <= Z1;
         elsif E = "10" then   next_state <= Z0;
                               A <= '1';
         else                  next_state <= Z0;
               end if;
   when others => next_state <= Z0;
       end case;

  end process logic_next_state_and_out;
```

A.6.6. Digitale Filter

Die Faltung einer zeitdiskreten Funktion x(i) mit der Impulsantwort h_k wird durch die Faltungssumme

$$y[n] = \sum h_k \cdot x[n-k] \qquad (3.1)$$

beschrieben, wobei der Index k über alle vorhandenen Stützstellen h_k verläuft. Mit Hilfe von Registern zur Speicherung vergangener Werte der Eingangsfunktion x[n−k], sowie Bausteinen zur Addition und Multiplikation lässt sich diese Gleichung als digitales Filter abbilden. Folgende Abbildung zeigt ein digitales Filter für eine Impulsantwort mit insgesamt 5 Stützstellen (bzw. für 5 Filterkoeffizienten).



Frage 6.6.1 (4 Punkte): Welche Gleichung beschreibt die Schaltung für das Ausgangssignal y[n]?

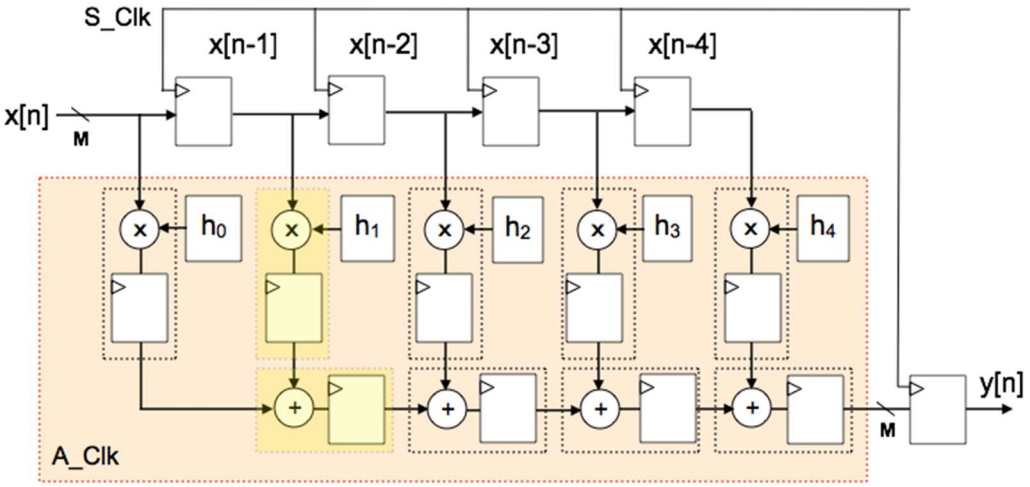
Lösung: $y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3] + h_4 \cdot x[n-4]$

Das Ausgangssignal ergibt sich aus einer Summe von Produkten der Filterkoeffizienten h_k mit dem Eingangssignal x[n-k] .

Frage 6.6.2 (4 Punkte): Welche Schwierigkeiten ergeben sich bei Implementierung dieser Struktur? Wie geht man mit diesen Schwierigkeiten um?

Lösung: Die Laufzeit für die Kette der Additionen muss beachtet werden. Damit sich die Laufzeiten nicht über alle Produkte kumulieren, lassen sich die Additionen durch eine geeignete Baumstruktur ordnen, bzw. man kompensiert die Laufzeiten durch zusätzliche Register. Eine weitere Möglichkeit besteht in der Verwendung einer transponierten Form des Algorithmus, bei der statt des Eingangssignals die Produktterme in Registern gespeichert werden.

Frage 6.6.3 (6 Punkte): Folgende Abbild zeigt eine Realisierung des Filters, das erst nach einer vorgegebenen Anzahl von Systemtakten den nächsten Wert des Eingangssignals übernimmt. Beschreiben Sie die Funktion dieser Struktur. Wie vermeidet diese Struktur die Schwierigkeiten aus Frage 6.6.2?



Lösung: (1) Die Struktur besteht aus einer Kette Multiplizierer mit eigenem Register und Addierer mit eigenem Register. Das Rechenwerk arbeitet mit einem eigenen Takt A_Clk, der schneller ist als die Abtastrate S_Clk (Sample Clock) und alle benötigten Operationen innerhalb eines Abtastintervalls durchführt. (2) Abtastrate und Rechenwerk sind entkoppelt. Die Produkte lassen sich parallel berechnen, die Additionen dann sukzessive kumulieren.

Frage 6.6.4 (4 Punkte): In der in Frage 6.6.3. gezeigten Struktur liegen die Stützstellen des Eingangssignals, des Ausgangssignals, sowie der Filterkoeffizienten mit einer Wortbreite von M Bit vor. Welche Wortbreiten werden innerhalb des Rechenwerkes benötigt? Begründen Sie Ihren Entwurf.

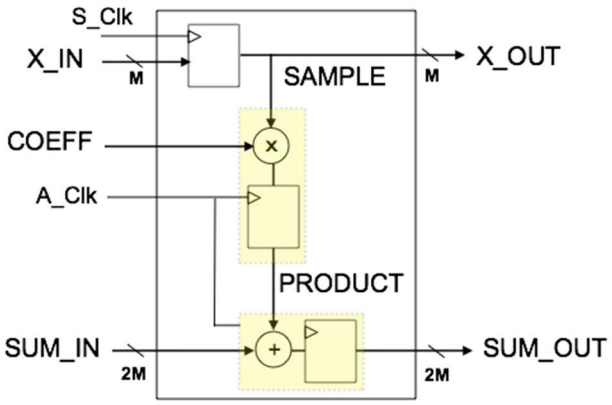
Lösung: (1) Die Multiplikation erfordert die doppelte Wortbreite (2M). (2) Bei jeder Addition könnte sich theoretisch die Summe verdoppeln, d.h. jeweils ein weiteres Bit erfordern. Im speziellen Fall ist die Summe aller Filterkoeffizienten jedoch kleiner gleich 1. In diesem Fall genügt die Wortbreite 2M. (3) Für das Ausgangssignal werden die obersten M-Bits verwendet (durch Abschneiden bzw. mit vorheriger Rundung).

Frage 6.6.5 (8 Punkte): Folgender HDL Text beschreibt eine Implementierung eines der Glieder der im Bild zu Frage 6.6.3 gezeigten Kettenstruktur. Skizzieren Sie ein Blockdiagramm der Zelle mit den benötigten Signalen (extern und intern). Beschreiben Sie den Aufbau und die Funktion des Bausteins.

```
--- FIR Filter Cell (VHDL), simplified form without reset

library IEEE;                           --- part #1
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity FIR_CELL is                      --- part #2
  port (S_Clk   : in  std_logic;        -- sample clock
        A_Clk   : in  std_logic;        -- algorithm clock
        X_IN    : in  std_logic_vector (11 downto 0);
        CEOFF   : in  std_logic_vector (11 downto 0);
        SUM_IN  : in  std_logic_vector (23 downto 0);
        SUM_OUT : out std_logic_vector (23 downto 0);
        X_OUT   : out std_logic_vector (11 downto 0));
end FIR_CELL;
```

```
architecture RTL of FIR_CELL is       --- part #3
                                  --- part #3.1
  signal PRODUCT : std_logic_vector (23 downto 0);
  signal SAMPLE  : std_logic_vector (11 downto 0);

  begin

  new_sample : process (S_Clk)        --- part #3.2
    begin
 if rising_edge(S_Clk) then
       SAMPLE <= X_IN;
 end if;
  end process new_sample;

  filter : process (A_Clk)            --- part #3.3
    begin
 if rising_edge(A_Clk) then
       PRODUCT <= SAMPLE * COEFF;
       SUM_OUT <= PRODUCT + SUM_IN;
 end if;
  end process filter;

  X_OUT <= SAMPLE;                    --- part #3.4

end RTL;
```

Lösung: (1) Blockdiagramm des Filterbausteins:



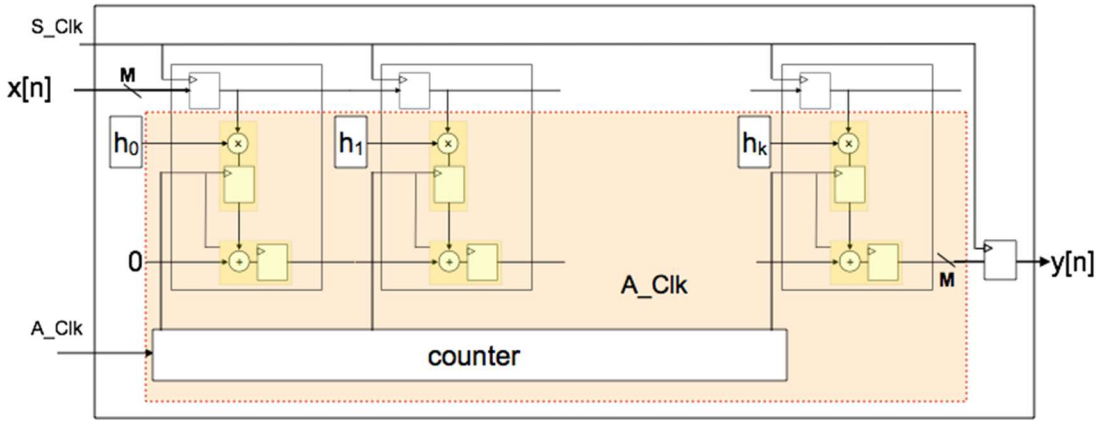
(2) Aufbau des Filterbausteins:

- Deklaration der Bibliotheken (part 1)
- Definition der Eingänge und Ausgänge (entity, part 2, siehe Blockschaltbild)
- Definition der internen Signale (part 3.1, siehe Blockschaltbild)
- parallele Prozesse: (a) nächsten Abtastwert übernehmen (Prozess new_sample, part 3.2), (b) Filterkette berechnen (Prozess filter, part 3.3), (c) gepufferten Abtastwert ausgeben (part 3.4)

(3) Funktion des Bausteins: Der Baustein arbeitet mit zwei unabhängigen Takten für die Übergabe des nächsten Abtastwertes, sowie für das interne Rechenwerk. Der Baustein enthält keine eigenen Daten, sondern bildet nur eine Stufe des Algorithmus ab. Der Baustein ist für eine Kettenstruktur aus identischen Bausteinen geeignet.

Frage 6.6.6 (6 Punkte): Die in Frage 6.6.5 gezeigten Filterbausteine lassen sich als Glieder einer Kette zu der in der Abbildung zu Frage 6.6.3 gezeigten Struktur aufbauen. Skizzieren Sie den Aufbau durch ein Blockschaltbild, das die äußere und innere Struktur beschreibt. Erläutern Sie, wie das Filter den eingangs beschriebenen Algorithmus abbildet.

Lösung:



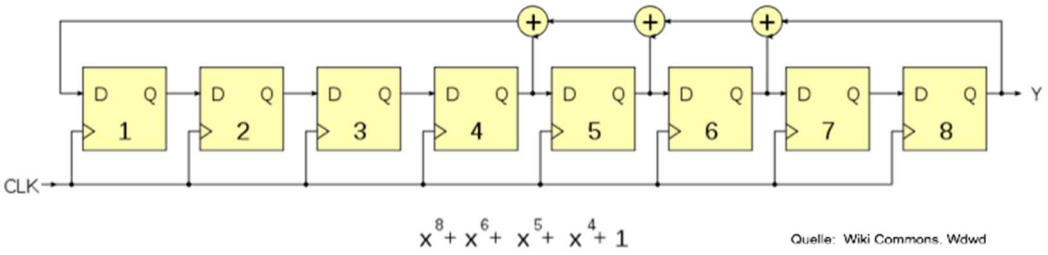
(1) Die Bausteine lassen sich zu einer Kette miteinander verbinden: hierbei wird jeweils ein Summenausgang mit dem nächsten Summeneingang verbunden, sowie jeder der Ausgänge X_OUT mit dem nächsten Ausgang X_IN.

(2) Das gesamte Filter besitzt als Eingänge (a) den aktuellen Wert des Eingangssignals x[n], (b) den aktuellen Wert des Ausgangssignals y[n], (c) den Takt zur Abtastung S_CLK, (d) den Takt für den Filteralgorithmus A_Clk. Die Filterkoeffizienten werden innerhalb des übergeordneten Bausteins abgebildet und mit den Eingangssignalen der Koeffizienten der Filterbausteine verbunden.

(3) Für jeden Takt der Abtastrate wird der Filteralgorithmus mit dem Systemtakt A_Clk durchlaufen, wobei ein Zähler verwendet wird. Der Filteralgorithmus ist in den einzelnen Elementen enthalten, siehe HDL-Code aus Frage 6.6.5. Die äußere Struktur instanziiert nur die Glieder der Kette, verknüpft die einzelnen Elemente, und stellt die Filterkoeffizienten bereit. Systemtakt und Abtastrate stehen als Eingangssignale der äußeren Struktur für die Durchführung des Algorithmus zur Verfügung.

A.6.7. Zufallszahlen

Pseudo-Zufallszahlen lassen sich mit Hilfe eines rückgekoppelten Schieberegisters erzeugen, wie in folgender Abbildung gezeigt. Die zurück gelesenen Werte einzelner Stufen sind hierbei mit Hilfe des Operators „+“ für XOR verknüpft. Als Initialwert des Registers sollten nicht alle Werte zu Null gesetzt werden.



Frage 6.7.1 (6 Punkte): Verwenden Sie die Anordnung in einer Schaltung, mit Hilfe derer sich Zufallszahlen erzeugen lassen. Geben Sie ein Blockdiagramm der Schaltung mit allen Eingangssignalen und Ausgangssignalen an. Beschreiben Sie stichwortartig die Funktionsweise der Schaltung.

Lösung:

Eingangssignale: Clock, Reset.

Ausgangssignale: Y bzw. Q als 8-Bit Vektor.

Funktionsweise: Taktung des Schieberegisters mit der Taktrate Clock. An den Eingang gibt man das Ergebnis der XOR-Verknüpfung der Stellen Q(1) = Q(8) XOR Q(6) XOR Q(5) XOR Q(6). Für die Berechnung ist ein internes Signal (Register) zu verwenden.

Frage 6.7.2 (8 Punkte): Beschreiben Sie die Schaltung in HDL.

```
--- Pseudo Noise 8 bits (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity Pseudo_Noise_8 is
      port ( Clk, RS : in std_logic;  -- clock, reset (active low)
             Q : out std_logic_vector (8 downto 1)); -- 8 bits out
end Pseudo_Noise_8;

architecture RTL of Pseudo_Noise_8 is
      signal Qreg: std_logic_vector (8 downto 1);   -- internal signal
      begin
            process (Clk, RS) begin
               if (RS = '0') then           -- reset is active low!
                  Qreg <= "01001001";
               elsif rising_edge(Clk) then
               for i in 8 downto 2 loop
                  Qreg(i) <= Qreg(i-1);
                end loop;
                Qreg(1)<=Qreg(8) XOR Qreg(6) XOR Qreg(5) XOR Qreg(4);
              end if;
      end process;
      Q <= Qreg;
end RTL;
```

Frage 6.7.3 (6 Punkte): Definieren Sie eine Testumgebung für die Schaltung. Welches ist der grundsätzliche Aufbau der Testumgebung? Welche Prozesse verwenden Sie für die Durchführung der Tests?

Lösung: grundsätzlicher Aufbau siehe vorausgegangene Musterlösungen. Testprozesse: (1) Reset, (2) Takt. Ausgang lässt sich im Zeitdiagramm beobachten.

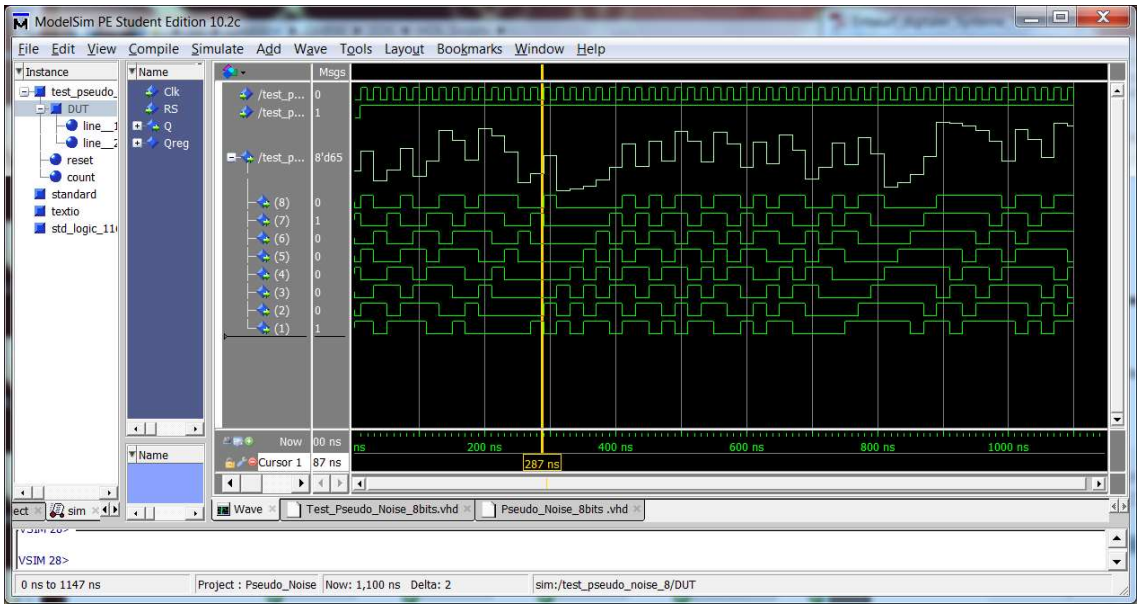
Frage 6.7.4 (6 Punkte): Beschreiben Sie die Testschaltung in HDL.

Lösung: siehe folgender HDL-Text

```
--- Testbench for Pseudo Noise 8 bits (VHDL)
---
library ieee;
use ieee.std_logic_1164.all;

entity Test_Pseudo_Noise_8 is
end Test_Pseudo_Noise_8;

architecture Behavioural of Test_Pseudo_Noise_8 is

  -- state DUT as component
  component Pseudo_Noise_8 is
        port ( Clk, RS : in std_logic;  -- clock, reset
               Q : out std_logic_vector (8 downto 1)); -- 8 bits out
  end component;

  -- testbench internal signals
  signal T_Clk: std_logic := '0';
  signal T_RS : std_logic := '0';
  signal T_Q : std_logic_vector (8 downto 1);

  begin

  -- connect DUT to testbench
  DUT: Pseudo_Noise_8 port map (Clk => T_Clk, RS => T_RS, Q => T_Q);

  -- run tests
  reset : process -- reset counter once
  begin
     wait for 5 ns; T_RS <= '0';
     wait for 4 ns; T_RS <= '1';
     wait;
  end process reset;

  count: process -- count forever
    begin
     T_Clk <= '0';
     wait for 10 ns;
     T_Clk <= '1';
     wait for 10 ns;
```

```
  end process count;

end Behavioural;
```

Zur Ergänzung noch ein Simulatorlauf (wird in der Prüfung nicht verlangt):



A.6.8 Arithmetisch-Logische Einheit

Es soll eine Arithmetisch-Logische-Einheit (ALU) für zwei 4 Bit breite, vorzeichenlose Operanden unter Verwendung geeigneter VHDL-Operatoren entworfen werden. Das Ergebnis soll ebenfalls 4 Bit breit sein. Außerdem sollen zwei Flags (Zero- und Carry-Flag) erzeugt werden. Die Funktion der ALU wird von dem 2 Bit breiten Signal Opcode wie folgt gesteuert:

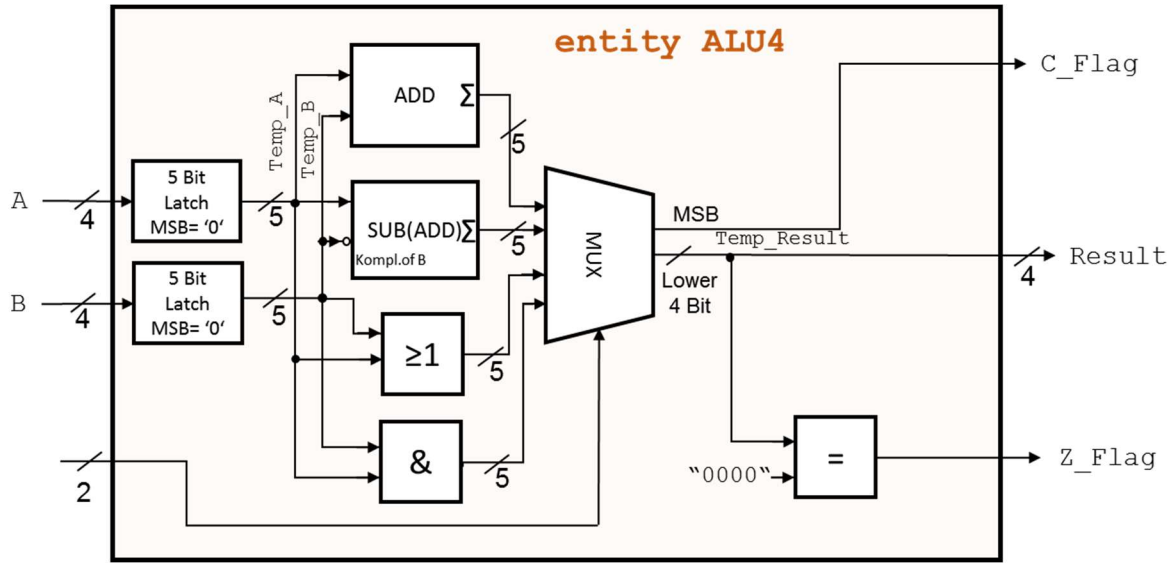
Opcode	Funktion	C_Flag
00	Addition A + B	1 falls Ergebnis > 0xF, sonst 0
01	Subtraktion A – B	1 falls Ergebnis < 0, sonst 0
10	Bitweise ODER-Verknüpfung	0
11	Bitweise UND-Verknüpfung	0

Tabelle 2: Funktionen der Arithmetisch-Logischen-Einheit

Das Z_Flag wird gesetzt, falls das ALU-Ergebnis = 0 ist.

Frage 6.8.1 (8 Punkte): Zeichnen Sie ein Blockdiagramm der Schaltung mit externen und internen Signalen für die beschriebene Arithmetisch-Logische Einheit (ALU). Bitte erläutern sie die internen Funktionsblöcke in Stichworten.

Lösung:



Es werden ein Addierer, ein Subtrahierer (also ein weiterer Addierer mit A und dem Komplement von B als Operanden), ein ODER und ein UND (jeweils bitweise). Je nach gewähltem Opcode wird eines der 4 Ergebnisse dieser Funktionsblöcke als Ergebnis ausgegeben, wobei der Übertrag der Addierer, der im MSB erscheint, als C-Flag verwendet wird. Mit einem Komparator wird das Z-Flag ermittelt. Gegebenfalls würde ein Synthesetool die beiden Addierer zu einem zusammenfassen um Ressourcen zu sparen, in diesem Fall würde das Komplement von B oder B abhängig vom Opcode als zweiter Operand dem Addierer zugeführt.

Frage 6.8.2 (4 Punkte): Welche Breite wird für die internen Operanden benötigt? Begründen Sie Ihre Entscheidung und erläutern Sie die Auswirkung auf die Operationen

Lösung: Die Auswertung des Carry-Flags ist nur möglich, wenn die arithmetischen Operationen statt auf einer Operandenbreite von 4 Bit auf einer Operandenbreite von 5 Bit durchgeführt werden da sonst der Überlauf (Carry) verloren geht. Dies betrifft die Addition und die Subtraktion, die logischen Verknüpfungen sind nicht davon betroffen und könnten mit 4 Bit durchgeführt werden.

Frage 6.8.3 (8 Punkte): Im Folgenden ist das Gerüst eines VHDL Codes für die oben beschriebene Arithmetisch-Logische Einheit (ALU) angegeben. Bitte ergänzen Sie die fehlenden Stellen so, dass die oben beschriebene Funktionalität realisiert wird.

Lösung:

```
-- 4-Bit ALU
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ALU4 is
     port( A, B: in std_logic_vector(3 downto 0); --4-Bit operand A and B
       Opcode: in std_logic_vector(1 downto 0);    --2-Bit Opcode
       Result: out std_logic_vector(3 downto 0);   --4-Bit result
       C_Flag, Z_Flag: out std_logic);             --Carry Flag / Zero Flag
end ALU4;

architecture Behavioural of ALU4 is
```

```
-- internal signals
signal Temp_Result, Temp_A, Temp_B: std_logic_vector(4 downto 0) :="00000";

begin
-- assign A & B to internal signals
Temp_A (3 downto 0) <= A;  -- set lower 4 bit of Temp_A to A
Temp_B (3 downto 0) <= B;  -- set lower 4 bit of Temp_B to B

calc: process (Temp_A, Temp_B, Opcode)  -- calculate temporary result
begin
     case Opcode is
      when "00" => Temp_Result <= Temp_A + Temp_B;
      when "01" => Temp_Result <= Temp_A - Temp_B;
      when "10" => Temp_Result <= Temp_A or Temp_B;
      when "11" => Temp_Result <= Temp_A and Temp_B;
      when others => Null;
     end case;
end process calc;

setzflag: process (Temp_Result) -- Set Z_Flag
begin
     if Temp_Result(3 downto 0) = "0000" then
      Z_Flag <= '1';
     else
      Z_Flag <= '0';
     end if;
end process setzflag;

setcflag: process (Temp_Result) -- Set C_Flag
begin
     if Temp_Result(4)='1' then
      C_Flag <= '1';
     else
      C_Flag <= '0';
     end if;
end process setcflag;

Result <= Temp_Result(3 downto 0);

end Behavioural;
```

Frage 6.8.4 (8 Punkte): Für die Arithmetisch-Logische-Einheit (ALU) soll die unten skizzierte Testumgebung zum Einsatz kommen. Beschreiben sie das grundsätzliche Vorgehen beim Test in Worten. Welche Testfälle machen für die Arithmetisch-Logische-Einheit (ALU) Sinn? Wählen Sie sechs sinnvolle Testfälle aus und geben Sie den Testvektor für diese 6 Testfälle an.

```
-- Testbench for ALU4
```

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

entity test_ALU4 is
end test_ALU4;

architecture behavior of test_ALU4 is

    -- component Declaration for the Device Under Test (DUT)
    component ALU4
    port(
         A : in  std_logic_vector(3 downto 0);
         B : in  std_logic_vector(3 downto 0);
         Opcode : in  std_logic_vector(1 downto 0);
         Result : out  std_logic_vector(3 downto 0);
         C_Flag : out  std_logic;
         Z_Flag : out  std_logic
        );
    end component;

   --inputs
   signal A : std_logic_vector(3 downto 0) := (others => '0');
   signal B : std_logic_vector(3 downto 0) := (others => '0');
   signal Opcode : std_logic_vector(1 downto 0) := (others => '0');

     --outputs
   signal Result : std_logic_vector(3 downto 0);
   signal C_Flag : std_logic;
   signal Z_Flag : std_logic;

    -- testbench clock signals
     signal clk : std_logic := '0';

     -- testbench test vectors
     type test_rec is record
            A : std_logic_vector(3 downto 0);
            B : std_logic_vector(3 downto 0);
            Opcode : std_logic_vector(1 downto 0);
            Result : std_logic_vector(3 downto 0);
            C_Flag : std_logic;
            Z_Flag : std_logic;
     end record;

     type test_arr is array(positive range <>) of test_rec;

     constant test_vector : test_arr := (
         -- hier wird der Testvektor gesetzt,
         -- geben sie diesen für Ihre Testfälle an
       );
begin
```

```
    -- Instantiate the Unit Under Test (UUT)
   DUT: ALU4 port map (
          A => A,
          B => B,
          Opcode => Opcode,
          Result => Result,
          C_Flag => C_Flag,
          Z_Flag => Z_Flag
        );

    -- run tests
    test: process
    begin
     L1: for j in 1 to 6 loop            -- step to next transition

       A <= test_vector(j).A;            -- set input signal to test vector
       B <= test_vector(j).B;            -- set input signal to test vector
       Opcode <= test_vector(j).Opcode;  -- set input signal to test vector

       clk <= '0';
       wait for 50 ns;
       clk <= '1';
       wait for 50 ns;

        -- compare output signals
       if((Result/=test_vector(j).Result)                                or
      (C_Flag/=test_vector(j).C_Flag)
       or (Z_Flag/=test_vector(j).Z_Flag)) then
           report "test step failed" severity NOTE;
       else
           report "test step passed" severity NOTE;
       end if;
     end loop L1;
    wait;
    end process test;

end behavior;
```

Lösung:

Der Prüfling wird in die Testbench eingespannt und mit vorgegeben Testsignalen aus dem Testvektor stimuliert. Die Ergebnisse aus dem Prüfling werden mit den erwarteten Ergebnissen aus dem Testvektor verglichen. Im Fall eines korrekten Ergebnisses wird eine positive Meldung ausgegeben, im anderen Fall eine Fehlermeldung.

Getestet werden sollten alle Operationen, im Idealfall mit allen Kombinationen von Eingangssignalen. Insbesondere sollten aber Fälle mit Übertrag bei Addition und Subtraktion getestet werden, sowie Fälle mit dem Ergebnis “0000“, da dann die entsprechenden Flags gesetzt sein müssen.

Als Testfälle wurde gewählt:

1. Addition von 1 + 1 = 2, keine Flags sollten gesetzt sein

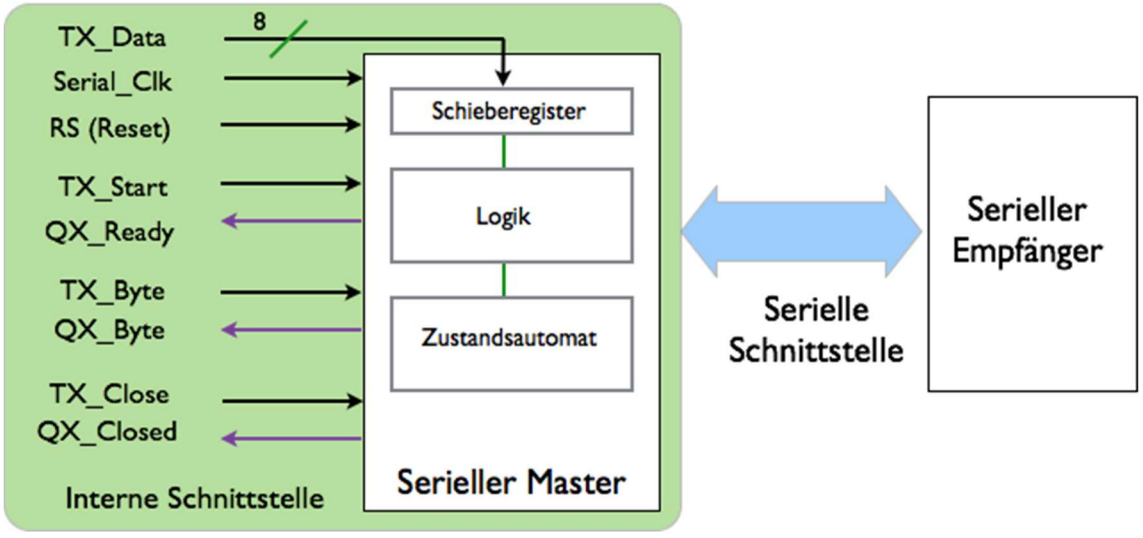
2. Subtraktion von 1 – 1 = 0, dabei muss als Ergebnis das Z_Flag gesetzt werden,
3. "0101" ODER "1001" = "1101"
4. "0101" UND "1001" = 0001"
5. Subtraktion von 1 – 3 = -2 (≙ "1110"), dabei muss als Ergebnis das C_Flag gesetzt werden,
6. Addition 15 + 1 = 16, dabei muss sich als Ergebnis in der 4 – Bit Zahl „0000" ergeben und beide Flags sollten gesetzt sein.

Der Testvektor in der Testbench dafür ist:

```
constant test_vector : test_arr := (
  ("0001", "0001", "00", "0010", '0', '0'),
  ("0001", "0001", "01", "0000", '0', '1'),
  ("0101", "1001", "10", "1101", '0', '0'),
  ("0101", "1001", "11", "0001", '0', '0'),
  ("0001", "0011", "01", "1110", '1', '0'),
  ("1111", "0001", "00", "0000", '1', '1')
  );
```

A.6.9 Zustandsautomat für serielles Protokoll

Zur Übertragung einzelner Bytes auf ein externes Bauteil (serieller Empfänger) soll eine serielle Schnittstelle verwendet werden. Zur Steuerung der Schnittstelle wird ein Zustandsautomat eingesetzt. Folgendes Diagramm beschreibt die Anordnung.



Es werden folgende Signale zur Steuerung der Schnittstelle verwendet:

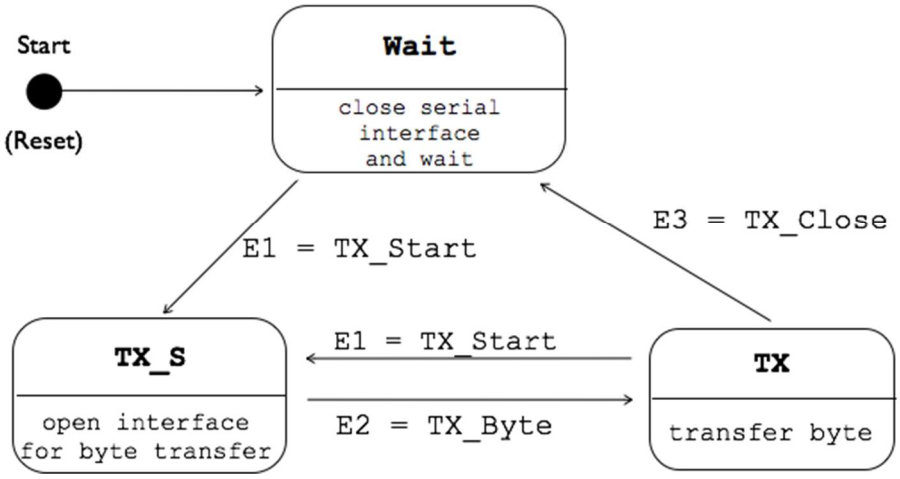
- TX_Start: Eröffnet die serielle Übertragung; Automat quittiert mit QX_Ready
- TX_Byte: Übertragung eines Bytes über die serielle Schnittstelle; Automat quittiert nach Übertragung des Bytes mit QX_Byte
- TX_Close: Es sind keine weiteren Bytes zu übertragen; Automat gibt serielle Schnittstelle frei, geht in den Wartezustand und quittiert mit QX_Closed

Die Signale bzw. Quittungen werden als Eingänge E bzw. Ausgänge A des Zustandsautomaten kodiert. Für den Test der internen Schnittstelle wird folgender HDL-Programmtext verwendet:

```
--- Übliche Struktur eines Testprogramms (Component, Signale, DUT über Port-
Map einspannen), dann folgende Testschritte:

-- generate clock
  clock : process begin
    T_Serial_CLK <= '1';
    wait for 10 ns;
    T_Serial_CLK <= '0';
    wait for 10 ns;
  end process clock;

  test : process begin

    T_RS <= '0';            -- Reset (active low)
    wait for 5 ns;
    T_RS <= '1';
    wait for 5 ns;

    T_TX_Data <= x"00";     -- transmit first byte
    T_E <= "01";            -- TX_Start
    wait until T_A = "01";  -- QX_Ready

    T_E <= "10";            -- TX_Byte
    wait until T_A = "10";  -- QX_Byte
    wait for 5 ns;

    T_TX_Data <= x"bb";     -- transmit next byte
    T_E <= "01";            -- TX_Start
    wait until T_A = "01";  -- QX_Ready

    T_E <= "10";            -- TX_Byte
    wait until T_A = "10";  -- QX_Byte
    wait for 5 ns;

    --  close transmission
    T_E <= "11";            -- TX_Close
    wait until T_A = "11";  -- QX_Closed

    wait;

  END PROCESS test;
```

Frage 6.9.1 (8 Punkte): Erstellen Sie das Zustandsdiagramm des Automaten.

Lösung:



Frage 6.9.2 (6 Punkte): Erläutern Sie die Funktion des Automaten mit Hilfe des Zustandsdiagramms. Ordnen Sie die Ereignisse den Eingängen des Automaten zu, sowie die Quittungen den Ausgängen.

Siehe vorausgegangene Musterlösungen. Zum Transfer mehrerer Bytes wechselt man zwischen den Zuständen TX_S (Initialisierung eines Byte-Transfers) und TX (Übertragung eines Bytes). Durch das Ereignis E3 beendet man die Übertragung. In diesem Zustand wird die serielle Schnittstelle (serieller Bus) wieder freigegeben.

Die Ereignisse stellen die Anweisungen an den Zustandsautomaten dar (E1 bis E3 mit den Bedeutungen wie im Z-Diagramm oben). Gesteuert durch diese Ereignisse wechselt der Automat zwischen den Zuständen. Als Rückmeldung schickt der Automat Quittungen. Hierzu kann man beispielsweise jedes Ereignis einzeln quittieren, z.B. A1 = QX_Start (Quittung, dass die Übertragung startklar ist), A2 = QX_Byte (Quittung, dass ein Byte übertragen wurde), A3 = QX_Close (Quittung, das die Schnittstelle freigegeben wurde und der Automat in den Wartezustand übergegangen ist.

Frage 6.9.3 (4 Punkte): Erstellen Sie eine Zustandsübergangstabelle.

Lösung: siehe vorausgegangene Musterlösungen.

Frage 2.4 (8 Punkte): Skizzieren Sie eine HDL-Implementierung des Automaten, in dem Sie den folgenden HDL-Text geeignet ergänzen. Erläutern Sie die Funktion der Zustände. Erläutern Sie die Funktion der Prozesse.

```
entity Serial_Master is
  Port ( RS   : in STD_Logic;                    -- Reset (active low)
    TX_Data: in STD_LOGIC_VECTOR (7 downto 0);  -- Byte Register
    Serial_Clk    : in  STD_LOGIC;              -- Serial clock

    E       : in  STD_LOGIC_vector (1 downto 0); -- Control Events
    -- "01" TX_Start, "10" TX_Byte,"11" TX_Close

    A  : out  STD_LOGIC_vector (1 downto 0);     -- State Actions
    -- "01" QX_Ready, "10" QX_Byte,"11" QX_Closed

      -- serial bus, e.g. SPI
```

```
           MOSI       : out STD_LOGIC;    -- SPI Master out signal
           SCLK       : out STD_LOGIC;    -- SPI clock signal out
           SS         : out STD_LOGIC     -- SPI chip select signal
         );
end Serial_Master;

architecture RTL of Serial_Master is

-- internal signals and variables
signal txreg        : std_logic_vector(7 downto 0) := (others=>'0');
signal counter      : integer := 0;

type state_type is (Z0, Z1, Z2);
 -- Z0: . . . . . .
 -- Z1: . . . . . .
 -- Z2: . . . . . .

signal next_state, current_state: state_type;

begin

  -- update state register in engine
  update_state_register: process(Serial_Clk, RS) -- process #1
    begin
      if (RS = '0') then
          current_state <= Z0;
      elsif rising_edge(Serial_Clk) then
          current_state <= next_state;
      end if;
  end process update_state_register;

  -- implement state logic
  logic_next_state: process(E, current_state) -- process #2
    begin
      case current_state is
        when Z0 =>
            if . . . . . then next_state <= . . . ;


            end if;
       when Z1 =>
            if . . . . . then next_state <= . . . ;


            end if;
        when Z2 =>
            if . . . . . then next_state <= . . . ;


            end if;
```

```
        when others => next_state <= Z0;
      end case;
    end process logic_next_state;
```

Lösung: Mit Bezug auf das Zustandsdiagramm: Z0 = Wartezustand (Wait), Z1 = Schnittstelle zur Übertragung öffnen (TX_S), Z2 = Byte übertragen (TX). Funktion der Prozesse: (1) Update_State_Register: Das Zustandsregister wird mit jedem Takt aktualisiert, (2) Logic_Next_State: Abhängig von den Ereignissen, sowie vom aktuellen Zustand erfolgen Zustandsübergänge. Muster für den HDL-Text:

```
 logic_next_state: process(E, current_state) -- process #2
    begin
      case current_state is
        when Z0 =>
            if E =  "01" then next_state <= Z1;
            end if;
       when Z1 =>
            if E =  "10" then next_state <= Z2;
            end if;
       when Z2 =>
            if E =  "01" then next_state <= Z1;
            elsif E = "11" then next_state <= Z0;
            end if;
        when others => next_state <= Z0;
      end case;
    end process logic_next_state;
```

Frage 6.9.5 (6 Punkte): Ergänzen Sie folgenden HDL-Text mit den der Aktionen des Zustandsautomaten (den Prozess zur Schaltung der Zustandsaktionen). Kodieren Sie hierzu die Ausgangssignale in geeigneter Weise.

```
-- state actions
 comb_logic_out: process(current_state, Serial_Clk) -- process 3
  begin
    case current_state is
      when Z0 =>
          SS <='1';        -- deselect chip
          SCLK <= '0';     -- SCLK operated in CPOL = 0
          A <= . . . . . . . . . . . . . . . . . . . .

      when Z1 =>             -- initialize bus/ serial interface
          counter <= 8;  -- initialize counter for byte transmission
          SS <='0';          -- chip select is active low
          SCLK <= '0';       -- SCLK operated in CPOL = 0 (active low)
          txreg <= TX_Data;
          A <= . . . . . . . . . . . . . . . . . . . .

      when Z2 =>             -- transfer one byte
        if(falling_edge(Serial_Clk) and (counter > 0))then
```

```
          SCLK <= '0'; -- SCLK follows serial clock for 8 counts
          MOSI <= txreg(7);
          for i in 0 to 6 loop
            txreg(7-i) <= txreg (6-i);
          end loop;
        elsif (rising_edge(Serial_Clk) and (counter > 0)) then
          SCLK <= '1'; -- SCLK follows serial clock for 8 counts
          counter <= counter - 1;
        end if;

        if (counter = 0) then
           A <= . . . . . .
        end if;

      when others => A <= . . . . . .

    end case;
  end process comb_logic_out;

end RTL;
```

Lösung: siehe Muster für einen HDL-Entwurf

```
 -- state actions
 comb_logic_out: process(current_state, Serial_Clk) -- process 3
  begin
    case current_state is
      when Z0 =>
          SS <='1';       -- deselect chip
          SCLK <= '0';    -- SCLK operated in CPOL = 0
          A <= "11";      -- handshake signal QX_Closed

      when Z1 =>            -- initialize bus/ serial interface
          counter <= 8;  -- initialize counter for byte transmission
          SS <='0';         -- chip select is active low
          SCLK <= '0';      -- SCLK operated in CPOL = 0 (active low)
          txreg <= TX_Data;
          A <= "01";        -- handshake signal init. ready (QX_Ready)

      when Z2 =>            -- transfer one byte
        if(falling_edge(Serial_Clk) and (counter > 0))then
          SCLK <= '0'; -- SCLK follows serial clock for 8 counts
          MOSI <= txreg(7);
          for i in 0 to 6 loop
            txreg(7-i) <= txreg (6-i);
          end loop;
        elsif (rising_edge(Serial_Clk) and (counter > 0)) then
          SCLK <= '1'; -- SCLK follows serial clock for 8 counts
          counter <= counter - 1;
        end if;

        if (counter = 0) then
```

```
            A <= "10"; -- handshake signal for byte transfrd. (QX_Byte)
          end if;

        when others => A <= "00";

      end case;
    end process comb_logic_out;

end RTL;
```

Frage 6.9.6 (8 Punkte): Erläutern Sie die Funktionen der einzelnen Prozesse der vorliegenden Implementierung im Zusammenhang mit den Zustandsgleichungen. Skizzieren Sie ein Blockdiagramm des Automaten (mit Bezug zu den Prozessen). Welche Vorteile bzw. Nachteile hat die Verwendung eines Zustandsautomaten zur Realisierung der seriellen Schnittstelle?

Lösung: Prozesse und Zustandsgleichungen: siehe Manuskript bzw. vorausgegangene Musterlösungen. Vorteile für eine serielle Schnittstelle: Übersichtliche Bedienung, Trennung der Aufgaben, z.B. der Bit-Synchronisation von Zustands-übergängen (siehe Sensitivitätsbereich der Prozesse), sowie klare Strukturierung in Zustände, Ereignisse und Quittungen. Nachteil: Einstiegsniveau und Aufwand etwas höher, da Kenntnisse über Z-Automaten erforderlich.

Anhang 2: Englisch - Deutsch

Black box	geschlossenes System
Combinatorial circuit	Schaltnetz
Combinational logic	kombinatorische Logik
Control	Steuerung
Data path	Datenpfad
Device under test (DUT)	Prüfling
Finite State Machine	Zustandsautomat
Latch	Auffangregister
Run Time Environment	Laufzeitumgebung
Sequential circuit	Schaltwerk, getaktete Logik
State diagram	Zustandsdiagramm
State event table	Zustandsübergangstabelle
Testbench	Testumgebung
White box	offenes System

Anhang 3: Abkürzungen

CPLD	Complex Programmable Logic Device
DNF	Disjunktive Normalform
DUT	Device under Test
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
KNF	Konjunktive Normalform
LUT	Look-up Table
PLA	Programmable Logic Array
ROM	Read Only Memory
VHDL	Very High Speed Integrated Circuit HDL

Anhang 4: Literatur

(1) Peter Sauer, Hardware-Design mit FPGA: Eine Einführung in den Schaltungsentwurf mit FPGA-Bausteinen, Elektor-Verlag, 2010, ISBN-13: 978-3895762093

(2) Jürgen Reichardt, Bernd Schwarz, VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme, Oldenbourg Wissenschaftsverlag; 2012 (6. aktualisierte Auflage), ISBN-13: 978-3486716771

(3) HDL-Editor http://www.hdlworks.com/products/scriptum/index.html und VHDL Online Reference Guide: http://www.hdlworks.com/hdl_corner/vhdl_ref/index.html

(4) HDL-Simulator:https://www.mentor.com/company/higher_ed/modelsim-student-edition (Lizenzschlüssel nach Registrierung)

(5) VHDL-Beispiele für Schaltungen und Tests im Web, z.B. unter http://esd.cs.ucr.edu/labs/tutorial/ (University of California, Computer Science)

(6) Jürgen Reichardt, Lehrbuch Digitaltechnik: Eine Einführung mit VHDL, Oldenbourg Wissenschaftsverlag, 2011 (2. Auflage), ISBN-13: 978-3486706802