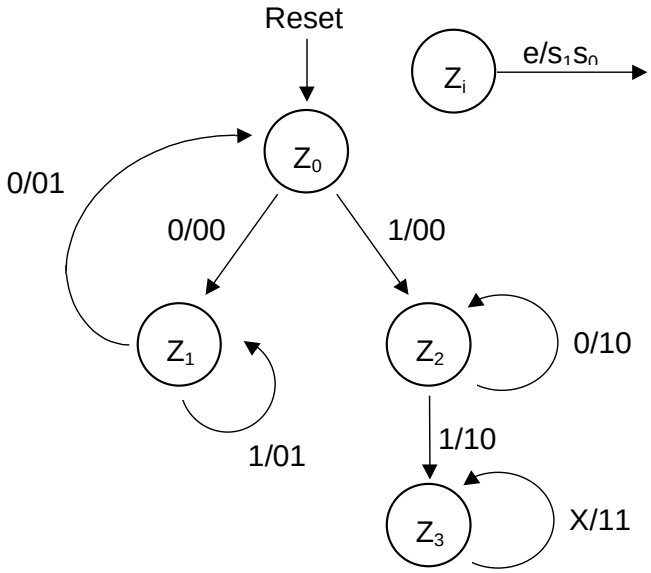
Digitale Integrierte Schaltungen
384.086



Beispielsammlung mit Lösungen

Peter Rössler
Herbert Nachtnebel
Rosemarie Velik
Christian Stoif
Martin Pongratz

ICT

E384 Institut für Computertechnik
Technische Universität Wien

WS 2012/2013

Inhaltsverzeichnis

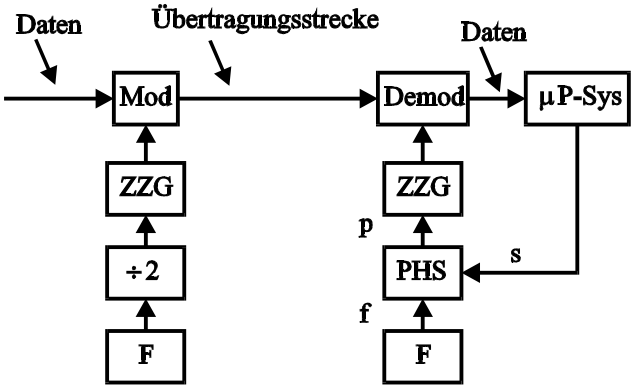
Angabenteil

Lösungsteil

1. Moore-Automaten

1.1. Phasenschieber

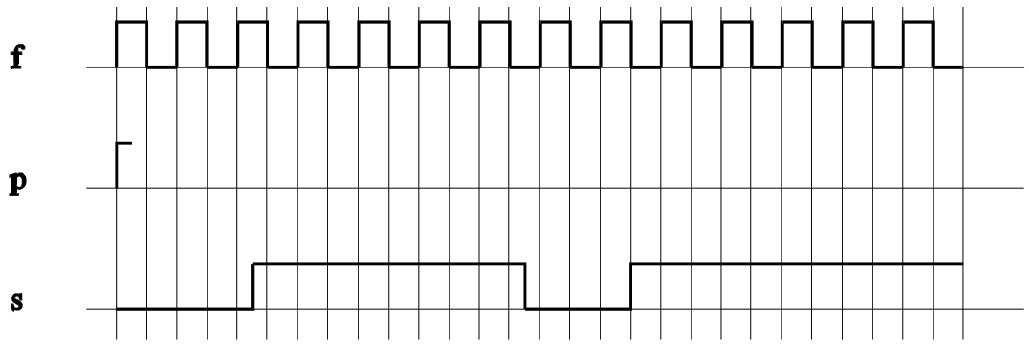
Für die Übertragung von Daten über stark gestörte Kanäle, wie zum Beispiel das 230V-Netz, ist die Spread-Spectrum-Technik mit Direct Sequence besonders gut geeignet. Der serielle Datenstrom wird dabei mit einer periodischen, binären Zufallszahlenfolge[1] moduliert, sodaß sich die Energie des Signals auf eine große Bandbreite verteilt. Der Empfang erfolgt durch eine zweite codesynchrone Modulation mit derselben Zufallszahlenfolge, die den Originalzustand der Daten wiederherstellt.



F	Quarz-Taktgenerator
÷2	Teiler durch zwei
ZZG	Zufallszahlengenerator
Mod	Modulator
Demod	Demodulator
µP-Sys	Mikroprozessorsystem
PHS	steuerbarer Phasenschieber
f	Taktsignal vom Quarzoszillator
p	phasenkorrigierter Takt
s	Steuersignal für Phasenschieber

Um die Synchronität der Zufallszahlenfolge im Empfänger einstellen zu können, benötigt man einen Phasenschieber PHS, der den quarzgenauen Takt f kontrolliert verzögert. Die Steuerung der Taktverzögerung erfolgt vom Mikroprozessorsystem über s. Ändert sich der Pegel an s nicht, so ist die Frequenz von p gleich f/2. Bei einer (asynchronen) Änderung des Pegels an s wird eine Halbwelle von p von 1/f auf 2/f verlängert. Ändert sich der Pegel an s während der nächsten Halbwelle von p erneut, so wird die Halbwelle von p um weitere 1/f verlängert. Es findet erst wieder ein Flankenwechsel an p statt, wenn sich der Zustand von s während einer Halbwelle 1/f von p nicht ändert. Die lokale periodische Zufallszahlenfolge läßt sich mit PHS über s solange verschieben, bis sie mit dem Eingangssignal codesynchron ist.

a) Ergänzen Sie im nachstehenden Zeitdiagramm das Signal p.



[1] Der gebräuchliche Ausdruck *periodische Zufallszahlenfolge* führt oft zu Mißverständnissen. Unter einer periodischen Zufallszahlenfolge versteht man eine Abfolge von Zahlen, welche sich periodisch wiederholt. Da die Abfolge sich periodisch wiederholt ist sie voraussagbar und damit nicht wirklich zufällig. Da die Folge jedoch nicht monoton ist, erscheint die Reihenfolge der Zahlen bei entsprechend großer Periode völlig zufällig. Daraus ergibt sich die Bezeichnung *periodische Zufallszahlenfolge*. Derartige Folgen werden in der Praxis mit rückgekoppelten Schieberegistern erzeugt.

b) Wie groß ist die Frequenz p in Abhängigkeit von f und s? Geben Sie den sinnvollen Wertebereich von s und den daraus folgenden Wertebereich für p an.

c) Nehmen Sie für das Propagation Delay aller verwendeten Bauteile den theoretischen Wert von 0 an. Muß der Automat mit positiv oder negativ flankengetriggerten Flip-Flops realisiert werden?

d) Warum wird für die Realisierung der Aufgabenstellung ein Moore-Automat und kein Mealy-Automat verwendet?

e) Wie viele Zustände besitzt der Automat?

f) Wie kann man erreichen, daß die Ausgangsgleichung für p möglichst einfach ist (siehe Fußnote [2])?

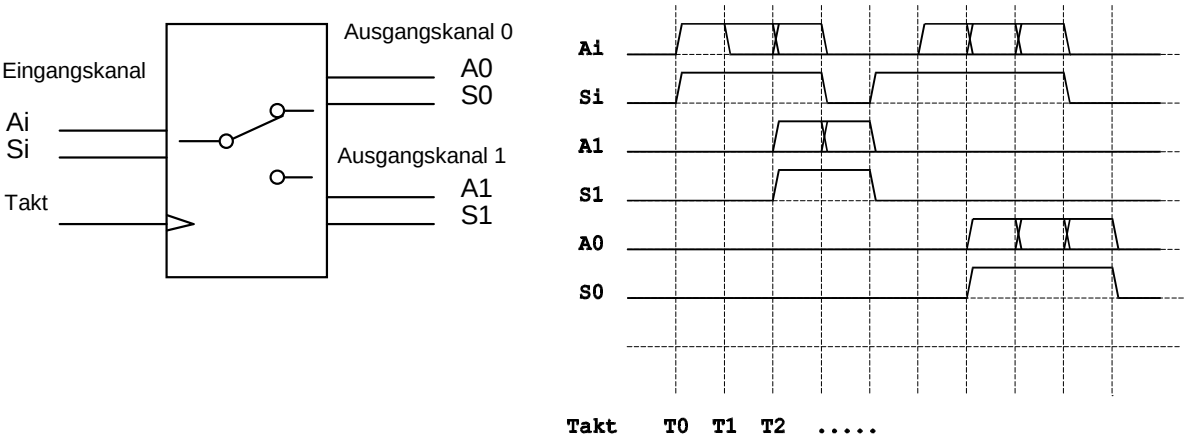
g) Entwerfen Sie den Zustandsgraphen und beschriften Sie die Übergangsvektoren mit den zugehörigen Eingangs-/Ausgangsgrößen. Geben Sie die Bedeutung der einzelnen Zustände an.

h) Ermitteln Sie die Zustandsübergangstabelle und die zugehörigen Ausgangswerte für p aus dem Zustandsgraphen unter Berücksichtigung von Punkt f).

i) Geben Sie die Gleichungen für die Realisierung mit D-Flip-Flops an.

1.2. Schaltknoten

Für eine Datenpaketvermittlungsanlage ist ein Schaltknoten zu entwerfen. Der Schaltknoten besitzt einen Eingangskanal und zwei Ausgangskanäle zur Vermittlung serieller synchroner Datenpakete. Zusätzlich steht der datensynchrone Takt T zur Verfügung. Der Eingangskanal wird entsprechend dem ersten Bit jedes Datenpakets (Adreßbit) an einen der beiden Ausgangskanäle durchgeschalten (Adreßbit=0 $\Rightarrow$ Ausgangskanal 0, Adreßbit=1 $\Rightarrow$ Ausgangskanal 1). Im Bild ist die Übertragung von zwei Paketen dargestellt, wobei das erste Paket an den Ausgangskanal 1 und das zweite Paket an den Ausgangskanal 0 weitergeschaltet wird. An den Ausgangskanälen werden die Daten ohne dem Adreßbit ausgegeben. Die ausgegebenen Pakete sind damit immer um ein Bit kürzer als die empfangenen Pakete.



Alle drei Kanäle bestehen jeweils aus einer A-Leitung und einer S-Leitung. Auf den A-Leitungen werden die Daten seriell übertragen. Die S-Leitungen sind während der Datenübertragung high und

[2] Anmerkung: Der Einfachheit halber wird der Index n bei Signalen und Variablen oft weggelassen. Es entspricht hier also beispielsweise der Ausgang p gleich p^n.

im Ruhezustand low. Beachten Sie, daß auch die Signalleitungen S*n* am Ausgang erst bei Ausgabe des um das Adreßbit verkürzten Datenpaketes den Pegel 1 annehmen. Die Größe der Pakete am Eingang (exklusive dem Adreßbit) liegt zwischen einem Bit und beliebig vielen Bits. Durch eine oder mehrere Nullen auf S werden die Datenpakete voneinander getrennt. Ist S auf 0, so sind die Daten auf A ungültig.
Um eine beliebige Kaskadierbarkeit der Knoten zu gewährleisten, sollen die Knoten als Moore-Automaten ausgeführt werden. Bei Mealy-Automaten würde jeder Knoten mindestens eine Gatterlaufzeit Verzögerung der Datenleitung gegenüber der Taktleitung verursachen, was bei einer Hintereinanderschaltung mehrerer Knoten zu ungewollten Laufzeiteffekten führt. Für die Realisierung sollen D-Flip-Flops verwendet werden.

a) Zeichnen Sie den Zustandsgraphen für den Schaltknoten. Beschriften Sie die Kanten des Graphen mit A_iS_i und die Zustände mit den zugehörigen Ausgangssignalen in der Form $A_0S_0A_1S_1$.

b) Durch Fehler bei der Übertragung werden Pakete generiert die nur mehr ein Bit lang sind. Das Paket besteht also nur mehr aus dem Adreßbit. Erweitern Sie den Zustandsgraphen derart, daß solche Pakete vom Schaltknoten einfach ignoriert, d.h. nicht weitergeleitet werden. Alle folgenden Punkte dieses Beispiels beziehen sich auf diesen erweiterten Zustandsgraphen.

c) Wie viele Flip-Flops sind für die Realisierung des Automaten notwendig?

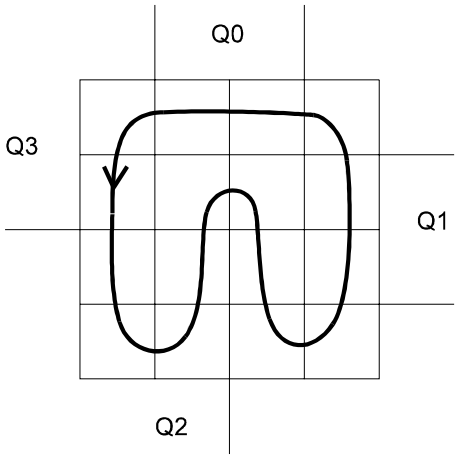
d) Wählen Sie eine Codierung für die Zustände und geben Sie die <u>vollständige</u> Zustandsübergangstabelle an. Wie wählen Sie die Folgezustände und Ausgangssignale in nicht benötigten Zuständen? Diskutieren Sie die verschiedenen Möglichkeiten.

e) Geben Sie die Gleichungen für A_0, S_0 und D_1 an.

f) Mithilfe der Schaltknoten soll eine Vermittlungsanlage für einen Sender und 6 Empfänger aufgebaut werden. Wie viele Schaltknoten werden zum Aufbau der Vermittlungsanlage benötigt? Welche Länge haben die einzelnen Pakete, wenn mit jedem Paket 8 Datenbits übertragen werden sollen?

1.3. Gray-Code Zähler

Es ist ein Gray-Code Zähler zu entwerfen, der mit Hilfe von 4 JK-Flip-Flops alle möglichen 16 Zustände entsprechend folgendem KV-Diagramm durchläuft:



a) Geben Sie die charakteristische Gleichung für das JK-FF an.

b) Ergänzen Sie die beiden folgenden Tabellen für das JK-FF.

J	K	Q^{n+1}

Q^n	Q^{n+1}	J	K

c) Zeichnen Sie ein KV-Diagramm mit den vier Variablen Q_3^n, Q_2^n, Q_1^n, Q_0^n. Tragen Sie in die untere Hälfte jedes der 16 Felder des KV-Diagramms ein, welchen Zuständen Q_3^n, Q_2^n, Q_1^n, Q_0^n das entsprechende Feld entspricht.

a) Tragen Sie in die obere Hälfte jedes der 16 Felder des KV-Diagramms aus c) die Folgezustände Q_3^{n+1}, Q_2^{n+1}, Q_1^{n+1}, Q_0^{n+1} ein.

b) Geben Sie die 8 KV-Diagramme für die Größen J_3^n, J_2^n, J_1^n, J_0^n und K_3^n, K_2^n, K_1^n, K_0^n an.

f) Bestimmen Sie mittels der KV-Diagramme aus e) die Gleichungen für alle J und K.

1.4. Arbiter

Es ist ein Arbiter (Arbitrierungsautomat) zu entwerfen. Er besitzt die Eingangssignale r_0 und r_1 (request), Reset und CLK sowie die Ausgangssignale s (status) und v (valid).

Über den Arbiter soll der Zugriff von zwei Einheiten E_0 und E_1 auf eine gemeinsame Resource geregelt werden. Will eine Einheit die Erlaubnis für den Zugriff auf die Resource erhalten, so setzt sie ihre Request-Leitung (r_0 bzw. r_1) auf 1. Der Arbiter setzt daraufhin zur nächsten aktiven Taktflanke den Valid-Ausgang v auf 0 und den Status-Ausgang auf 0 oder 1, entsprechend der Nummer der anfordernden Einheit. Mit der nächsten aktiven Taktflanke wird dann der Valid-Ausgang wieder auf 1 gesetzt (maximale Dauer von v=0 ist eine Taktperiode). Nach diesem Zeitpunkt werden auch wieder Änderungen an den Request-Eingängen (mit der nächsten Taktflanke) berücksichtigt. Sollten beide Request-Eingänge gleichzeitig gesetzt werden, so ist der Anforderung mit der niedrigsten Nummer der Vorzug geben. Nach dem Reset sollen v=1 und s=0 sein.

a) Zeichnen Sie ein Blockschaltbild des Arbiters und der beiden Anforderungseinheiten, wobei der Arbiter als Black Box mit allen Ein- und Ausgangssignalen zu zeichnen ist.

b) Geben Sie das Timing eines Anforderungszyklus mit allen Ein- und Ausgangssignalen des Arbiters an, wobei die Realisierung mit positiv flankengetriggerten Flip-Flops angenommen werden soll.

c) Geben Sie die Zustandsübergangstabelle und den Zustandsübergangsgraphen des Arbiters an.

d) Realisieren Sie den Automaten mit D-Flip-Flops.

2. Mealy-Automaten

2.1. Zweistelliger Binärzähler mit Übertragsausgang

Es ist ein zweistelliger Binärzähler zu entwerfen, der die Zustände 00, 01, 10, 11, 00, 01 ... zyklisch durchläuft. Der Zähler kann durch den Moduseingang M und die Setzeingänge E_0 und E_1 in einen definierten Anfangszustand gebracht werden. Ist M = 1, so werden bei der nächsten Taktflanke die Zustände von E_0 und E_1 in die beiden Flip-Flops (Ausgänge Q_0 und Q_1) übernommen. Für M = 0 wird der Zählerstand bei jeder Taktflanke um 1 erhöht.

Um den Zähler erweitern zu können, besitzt dieser einen Übertragsausgang C. Er soll im Zählmodus (M = 0) bei Zählerstand 11 gleich 1 sein und ist ansonsten 0. Die zweite und die weiteren Zählerstufen eines erweiterten Zählers müßten den C-Ausgang der ersten Zählerstufe auch als Eingang enthalten. In diesem Beispiel soll nur die erste Zählerstufe (also ohne C-Eingang) entworfen werden.

a) Zeichnen Sie ein Blockschaltbild der Zählers bestehend aus Schaltnetz (Black Box), Flip-Flops und allen Eingangs- und Ausgangssignalen.

b) Geben Sie die Übergangstabelle der Zählers an.

c) Zeichnen Sie den Zustandsgraphen des Zählers.

d) Bestimmen Sie die Lösungsfunktion D_1 für die Realisierung mit einem D-FF.

e) Bestimmen Sie die Lösungsfunktionen J_1 und K_1 für die Realisierung mit einem JK-FF durch Koeffizientenvergleich.

f) Bestimmen Sie die Lösungsfunktionen J_1 und K_1 für die Realisierung mit einem JK-FF direkt aus dem KV-Diagramm.

g) Geben Sie die charakteristische Gleichung für das RS-FF an und ergänzen Sie die folgende Tabelle:

Q^n	Q^{n+1}	S	R

h) Bestimmen Sie die Lösungsfunktionen R_1 und S_1 für die Realisierung mit einem RS-FF.

i) Warum ist zur Bestimmung der Lösungsfunktionen für ein RS-FF kein Koeffizientenvergleich, wie beim JK-FF möglich?

j) Woran sehen Sie, daß es sich bei diesem Automaten um einen Mealy-Automaten handelt?

2.2. Up/Down Counter

Es ist ein synchroner 8-Bit Up/Down Counter zu entwerfen. Der Zähler soll einen Carry/Borrow Ein- und Ausgang (CBI, CBO, aktiv high), einen Directioneingang (D, D = 0 entspricht Up, D = 1 entspricht Down), 8 Ausgänge für den Zählerstand (Q0..Q7) und einen Clockeingang (C) besitzen.

a) Zeichnen Sie ein Blockschaltbild des Zählers (Black Box) mit den Eingängen links und den Ausgängen rechts.

b) Skizzieren Sie, wie mehrere 8-Bit Zähler zu einem größeren Zähler kaskadiert werden können.

c) Entwickeln Sie die Schaltung des 8-Bit Zählers. Der Zähler ist modularisierbar! Überlegen Sie sich wie man die einzelnen Stellen des Zählers aus möglichst gleichartigen Modulen zusammensetzen kann. Die Schaltung jeder Stelle des Zählers kann durch einen Automat, welcher für alle Stellen des Zählers gleich ist, sowie durch zusätzliche kombinatorische Logik zusammengesetzt werden. Geben Sie das Schaltbild des gesamten Zählers an, wobei der Automat als Black Box gezeichnet wird.

d) Bestimmen Sie Zustandsgraph und Zustandsübergangstabelle des Automaten einer Zählstelle.

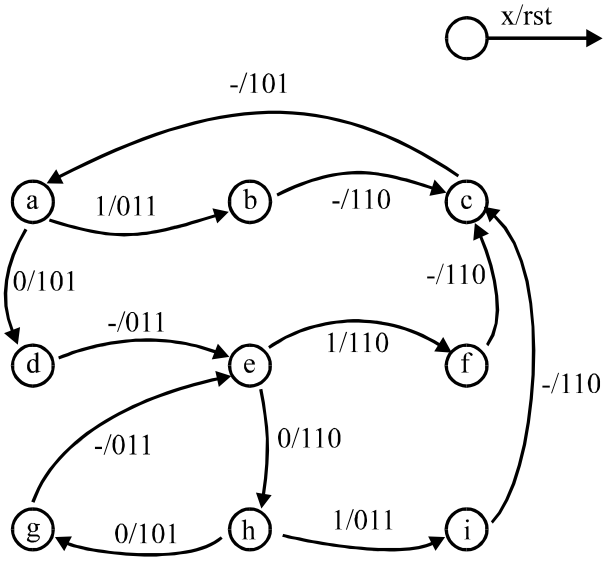
e) Geben Sie die Gleichungen für die Realisierung des Automaten einer Zählstelle an. Welchen Automatentyp (Moore oder Mealy) stellt der Automat einer Zählstelle dar?

f) Zeichnen Sie das Schaltbild des Automaten.

g) Versuchen Sie den 8-Bit Zähler als einen einzigen großen Automaten mit den Eingängen CBI und D sowie den Ausgängen Q0 bis Q7 und CBO zu entwickeln. Skizzieren Sie den Zustandsgraphen des Automaten und begründen Sie, warum solch eine Vorgangsweise nicht sinnvoll ist.

2.3. Vereinfachung von Zustandsgraphen

Aus dem untenstehenden Zustandsgraphen soll ein synchrones Schaltwerk in Form eines Mealy-Automaten entwickelt werden. Bei der Erstellung des Zustandsgraphen wurden redundante Zustände nicht beachtet.



a) Geben Sie den Zustandsgraphen in Listenform an. Ordnen Sie die Zeilen der Liste nach den Zuständen Z^n und geben Sie Eingangsgröße, Ausgangsgröße und nächsten Zustand Z^{n+1} an.

b) Formulieren Sie eine allgemeine Regel zur Vereinfachung dieser Zustandsliste durch Zusammenfassen von Zuständen. Die Funktion des Schaltwerks soll dabei vollständig unverändert bleiben.

c) Wenden Sie diese Regel auf die Zustandsliste an und zeichnen Sie den vereinfachten Zustandsgraphen. Zusammengefaßte Zustände sind durch einen neuen Buchstaben zu bezeichnen.

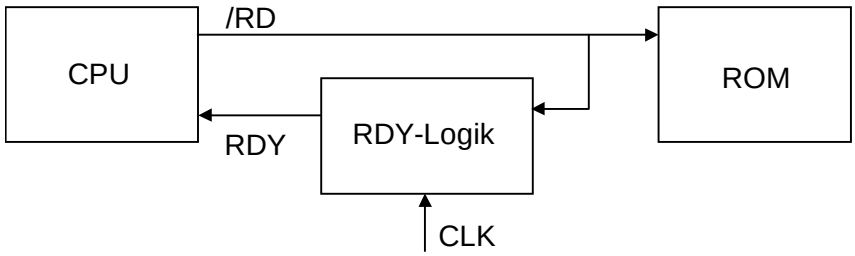
d) Wie viele Flip-Flops sind zur Realisierung des Automaten notwendig? Wählen Sie eine Zustandscodierung, sodaß die Gleichungen für r, s und t jeweils nur zwei disjunktive Terme enthalten.

e) Bestimmen Sie die Gleichungen für r, s und t mit der Hilfe von KV-Diagrammen, wobei Sie die gewählte Zustandscodierung aus Punkt d) zugrunde legen.

f) Bestimmen Sie die Lösungsgleichungen für eine Realisierung mit D-FFs.

2.4. Ready-Logik

Ein schneller Prozessor soll mit einem langsamen ROM verbunden werden. Der Lesezyklus, währenddessen der Prozessor das /RD-Signal auf 0 setzt, kann durch den RDY-Eingang des Prozessors verlängert werden. Das RDY-Signal muß dazu vom Ruhezustand 1 spätestens 30 ns nach der fallenden Flanke des /RD-Signals ebenfalls auf 0 gehen. 0 ns bis 50 ns nach der steigenden Flanke des RDY-Signals beendet der Prozessor den Lesezyklus und setzt das /RD-Signal wieder auf 1. Nach der steigenden Flanke des /RD-Signals erfolgt der nächste Lesezyklus vom Prozessor frühestens wieder nach 70 ns.



Die RDY-Logik ist als synchroner Automat mit D-Flip-Flops zu realisieren. Die Taktfrequenz des Taktes CLK der RDY-Logik beträgt 10 MHz und ist unabhängig vom Prozessortakt. Die Flip-Flops schalten bei der positiven Taktflanke. Das ROM verlangt eine minimale Dauer des /RD-Impulses von 200 ns. Es ist außerdem zu beachten, daß die Zeitdauer während der RDY = 0 ist, im Hinblick auf die Performance des Systems so kurz wie möglich sein soll. Die Bauelemente der RDY-Logik können als verzögerungsfrei angenommen werden (ideale Bauelemente).

a) Geben Sie das Timing eines Lesezyklus für die Signale /RD, RDY und CLK an. Zeitbereiche mit unbestimmtem Signalzustand sind dabei zu schraffieren.

b) Warum kann für die RDY-Logik kein Moore-Automat verwendet werden?

c) In welchem Zeitbereich t_{RDYmin} bis t_{RDYmax} ist RDY = 0?

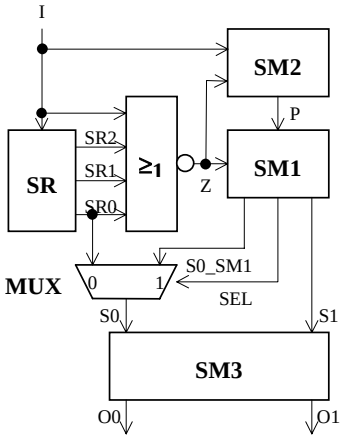
d) Geben Sie den Zustandsgraphen der RDY-Logik an. Kennzeichnen Sie im Timing-Diagramm, welcher Zeitbereich welchem Zustand des Automaten entspricht.

e) Leiten Sie aus dem Zustandsgraphen die Zustandsübergangstabelle ab. Codieren Sie die Zustände so, daß bei Verwendung von realen Bauelementen das Ausgangssignal RDY des Automaten frei von Hazards ist.

f) Geben Sie die zur Realisierung des Automaten erforderlichen Gleichungen an. Sie müssen dafür nicht extra ein KV-Diagramm zeichnen.

2.5. HDB3-Encoder

Es soll ein Encoder realisiert werden, welcher einen seriellen NRZ-Datenstrom in einen HDB3-codierten Datenstrom umwandelt. Der Encoder besitzt den Eingang I (Eingang für NRZ-Daten), den 2-Bit-Ausgangsvektor {O0, O1} (Ausgang der HDB3-codierten Daten), ein Takt- und ein Resetsignal (das Taktsignal und das Resetsignal wurden der Übersichtlichkeit halber nicht in das folgende Schaltbild eingezeichnet).



Funktionell läßt sich die Codierung in zwei aufeinanderfolgende Schritte zerlegen, wobei die Codierungsvorschrift für die beiden Schritte folgendermaßen lautet:

Im ersten Schritt wird aus dem seriellen NRZ-Datenstrom am Eingang I, ein Zwischensignal am Vektor {S0, S1} erzeugt. Dabei wird eine 1 am Eingang I ebenfalls auf eine 1 am Zwischensignal {S0, S1} abgebildet. Eine 0 am Eingang I wird, solange die 3 vorhergehenden Bits am Eingang I nicht ebenfalls 0 waren, auf eine 0 am Zwischensignal abgebildet. Ein Block von 4 aufeinanderfolgenden Nullen wird durch eines der beiden Ersatzmuster 100D oder 000D ersetzt. Das Ersatzmuster 100D wird substituiert, wenn seit dem letzten Ersatzmuster eine gerade Zahl von Einsen an I aufgetreten ist. Das Ersatzmuster 000D wird verwendet, wenn seit dem letzten Ersatzmuster eine ungerade Zahl von Einsen an I aufgetreten ist. Das Zwischensignal {S0, S1} wird von der Schaltung durch die Automaten SM1 und SM2, das Schieberegister SR, das NOR-Gatter und den Multiplexer MUX erzeugt.

Im zweiten Schritt wird aus dem Zwischensignal der HDB3-Code erzeugt und am Ausgangsvektor {O0, O1} ausgegeben. Eine 0 am Zwischensignal wird ebenfalls auf eine 0 am Ausgang abgebildet. Eine 1 am Zwischensignal wird abwechselnd in +1 und –1 am Ausgang abgebildet. Das mit D bezeichnete Bit am Zwischensignal wird in +1 und –1 abgebildet, aber so, das die Bipolar-Regel der

abwechselnden +1 und –1 verletzt wird. Das Ausgangssignal an {O0, O1} wird in der Schaltung vom Automaten SM3 aus dem Zwischensignal {S0, S1} erzeugt.

Beispiel:

```
Eingangssignal I:          ... 11011 0000 01 0000 0000 101 0000 ...
Zwischensignal {S0, S1}:   ... 11011 100D 01 000D 100D 101 100D ...
Ausgangssignal {O0, O1}:   ... +-0+- +00+ 0- 000- +00+ -0+ -00- ...
```

Die Werte +1 und –1 wurden in obigem Beispiel durch + und – dargestellt. Kurz nach jeder positiven Taktflanke wird ein neues Bit an I angelegt. Der Zeitversatz zwischen den Signalen I, {S0, S1} sowie {O0, O1}, welcher durch die Schaltung bedingt ist, wird in der Darstellung des obigen Beispiels nicht berücksichtigt. Die Codierung der Vektoren {S0, S1} und {O0, O1} erfolgt laut folgenden Tabellen:

S1	S0	entspricht
0	0	0
0	1	1
1	X	D

O1	O0	entspricht
0	X	0
1	0	-1
1	1	+1

Mit jeder positiven Taktflanke wird das Schieberegister SR um eine Stelle weiter geschoben, d.h.: I→SR2, SR2→SR1, SR1→SR0. Der Ausgang S0 des Multiplexers MUX wird bei SEL=0 mit dem Eingang 0 verbunden und bei SEL=1 mit dem Eingang 1 verbunden. Der Automat SM2 soll an seinem Ausgang P (Parity) eine 1 ausgeben, wenn seit dem letzten Ersatzmuster eine gerade Anzahl von Einsen aufgetreten ist. Wenn seit dem letzten Ersatzmuster eine ungerade Anzahl von Einsen aufgetreten ist, soll hingegen eine 0 an P ausgegeben werden. Alle 3 Automaten sollen mit positiv flankengetriggerten Flip-Flops realisiert werden. Die Resetzustände der Automaten und des Schieberegisters können in diesem Beispiel unberücksichtigt bleiben.

a) Geben Sie den Zustandsgraphen des Automaten SM1 an. Erklären Sie die Funktion der Schaltung um den Automaten SM1 (Schieberegister SR, NOR-Gatter und Multiplexer MUX). Muß der Automat als Moore-Automat oder als Mealy-Automat realisiert werden oder ist dies egal?

b) Geben Sie die Schaltung des Automaten SM2 an. Muß der Automat als Moore-Automat oder als Mealy-Automat realisiert werden oder ist dies egal?

c) Geben Sie den Zustandsgraphen des Automaten SM3 an wobei Sie den Automaten als Moore-Automat realisieren sollen.

d) Wie groß ist der zeitliche Versatz zwischen den Signalen I, {S0, S1} und {O0, O1} in Taktperioden?

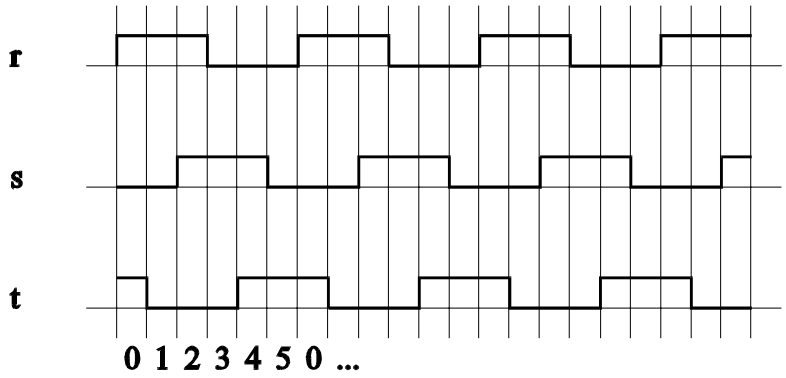
e) Stellt die Gesamtschaltung des HDB3-Encoders eines Mealy- oder einen Moore-Automaten dar?

f) Wie viele Flip-Flops werden für den gesamten HDB3-Encoder benötigt? Benötigt die Schaltung des HDB3-Encoders mehr oder weniger Bauteile, wenn der gesamte Encoder als ein einziger großer Automat realisiert wird?

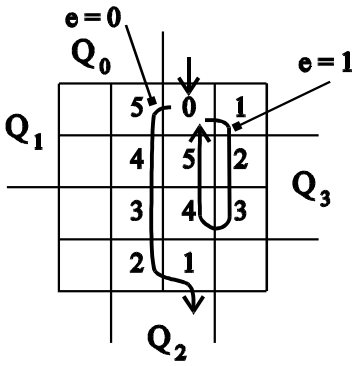
3. Programmschaltwerke

3.1. Programmschaltwerk für Drehstromsignal

Ein fiktiver Wechselrichter soll Gleichspannung aus einer Pufferbatterie in Drehstrom umwandeln. Als Steuersignale für die Wechselrichtung benötigt er die Signale r, s und t mit folgender Signalform:



Es lassen sich 6 unterschiedliche Zustände der Ausgänge r, s und t unterscheiden, die zyklisch durchlaufen werden. Der Umlaufsinn des Drehstroms soll umkehrbar sein. Abhängig vom Steuereingang e (Drehsinn) soll das Programmschaltwerk für e = 1 die Folge 0123450123.. und für e = 0 die Folge 0543210543.. an den Ausgängen r, s und t der Steuermatrix ausgeben. Der Drehsinn läßt sich nur im Zustand 0 umkehren. Um Hazards zu vermeiden, werden die Zustände des Schaltwerks im Gray-Code durchlaufen. Verwenden Sie folgende Zuordnung:



a) Geben Sie das Schaltbild des Programmschaltwerks an, das aus den Elementen Folgeadreßregister (D-FF), Dekoder, Folgeadreßmatrix und Steuermatrix besteht.

b) Ergänzen Sie die Diodenbrücken in der Folgeadreßmatrix für e = 1 (nur Drehrichtung 0123..). Zeichnen Sie für einen Kreuzungspunkt die exakte Beschaltung (Widerstände, Dioden) neben die Matrix. Für die restlichen Verbindungen in der Matrix reicht es einen Punkt an der entsprechenden Stelle zu machen.

c) Erweitern Sie die Schaltung durch Verwendung eines 5-fach-UND-Gatters und weiterer Gatter derart, daß im Zustand 0 für e = 0 auf den Zustand 5 und nicht auf den Zustand 1 verzweigt wird (Schaltung für zwei Drehrichtungen). Die Erweiterung erfolgt an zwei Leitungen die von der Folgeadreßmatrix zu den FF-Eingängen führen. Stammen vier der fünf Leitungen am UND-Gatter vom Ausgang oder vom Eingang der Folgeadreßmatrix? Ergänzen Sie die benötigten

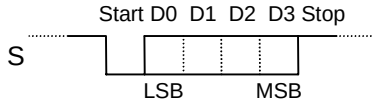
Verbindungen in der Folgeadreßmatrix, wobei Sie zur Unterscheidung der beiden Drehrichtungen diesmal Kreuze statt Punkte machen.

d) Ergänzen Sie die Verbindungen in der Steuermatrix für die Erzeugung der Signale r, s und t.

e) Welche Bauteile des Programmschaltwerks entsprechen dem Program Counter, dem ROM und dem Befehlsdekoder eines Programmschaltwerks bzw. Mikrocontrollers im allgemeinen Sinn? Welche Befehle können vom Programmschaltwerk abgearbeitet werden?

3.2. Anzeigeeinheit für seriell übertragene Daten

Über eine serielle Datenleitung S werden Nibbles (4-bit Worte) an eine Anzeigeeinheit übertragen. Nach dem Startbit (= 0) folgen die Datenbits in aufsteigender Reihenfolge (d.h. D_0 zuerst). Durch ein Stopbit (= 1) wird die Übertragung abgeschlossen. Der Ruhezustand der seriellen Leitung ist 1.

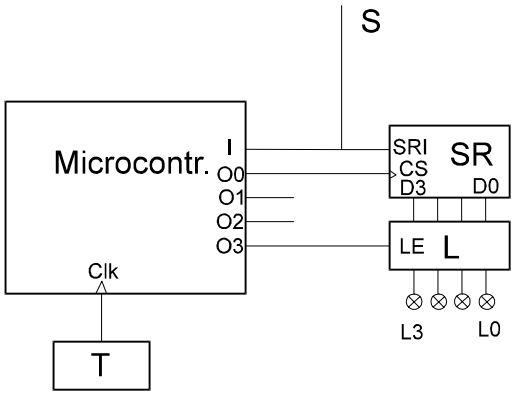


Das vollständig empfangene Nibble (Ausgabe erst nach Abschluß des Empfangs) wird von der Anzeigeeinheit über vier Lampen angezeigt, die mit den Steuerleitungen L_0 bis L_3 entsprechend den Datenbits D_0 bis D_3 angesteuert werden.

Die Anzeigeeinheit besteht aus zwei Empfangsregistern (Schieberegister SR und Latch L) und einem Microcontroller, der die Taktsignale für die beiden Empfangsregister aus dem seriellen Datenstrom am Eingang I berechnet.

Das Schieberegister SR besteht aus einer Kette von 4 D-Flip-Flops die gemeinsam durch eine positive Flanke am Takteingang CS getriggert werden. Der serielle Eingang des Schieberegisters heißt SRI (Eingang des höchstwertigsten D-FF) und die Ausgänge heißen D_0 bis D_3. Die Daten werden vom höchstwertigsten zum niederwertigsten Bit des Schieberegisters (also nach rechts) geschoben. Das D-Latch L übernimmt die Daten an seinen Eingängen vom Schieberegister, wenn an seinem Latch-Enable-Eingang LE eine 1 anliegt.

Der Microcontroller besitzt ein 1-Bit-Eingangsport I und ein 4-Bit-Ausgangsport O_0 bis O_3. Er generiert mit Hilfe des im internen ROM abgelegten Codes aus dem am Eingangsport I anliegenden seriellen Datenstrom die Taktsignale für die beiden Empfangsregister SR und L an seinem Ausgangsport O.



Der Microcontroller besteht im Wesentlichen aus vier Funktionsblöcken:

1.) Program Counter PC: Synchroner 4-Bit Zähler mit Preseteingang P. Mit P = 1 wird der Zählerstand vom Eingang übernommen, mit P = 0 wird der Counter inkrementiert. Die Übernahme des neuen Zählerstandes an die Ausgänge erfolgt bei der positiven Taktflanke.

2.) ROM: 4 Adreßeingänge, 7 Datenausgänge, die unteren 4 Bits enthalten Daten, die oberen 3 Bits enthalten den Befehlscode.

3.) 1 aus n Dekoder DEC: 3 Eingänge, 8 Ausgänge, die durch den Eingang ausgewählte Ausgangsleitung ist 1, alle anderen sind 0.

4.) Synchrones Ausgaberegister: 4-Bit negativ flankengetriggertes Register mit Takteingang C und Enableeingang EN. Ein 4-Bit Eingangswert wird vom Register mit der negativen Taktflanke C an die Ausgänge übernommen, wenn EN = 1 ist.

Außerdem stehen noch diverse Logikgatter zur Verfügung. Der Microcontroller soll die Befehle

JP	Adr	unbedingter Sprung zur angegebenen Adresse
JPZ	Adr	bedingter Sprung zur angegebenen Adresse wenn Eingangsport I = 0
JPNZ	Adr	bedingter Sprung zur angegebenen Adresse wenn Eingangsport I = 1
NOP		keine Aktion
OUT	Data	Ausgabe der angegebenen Daten am Ausgangsport

ausführen können. Das Hochzählen des Program Counters soll bei der positiven Flanke des Taktes Clk und das Ausgeben (Befehl OUT) bei der negativen Flanke von Clk erfolgen.

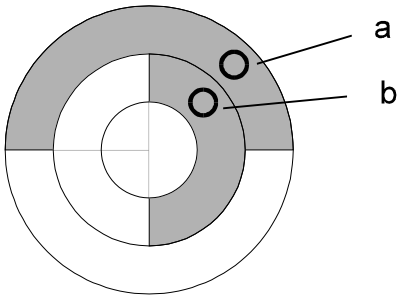
a) Zeichnen Sie die Schaltung des Microcontrollers. Beschriften Sie alle Ein- und Ausgänge der Bausteine denen in der Angabe ein Name gegeben wurde. Geben Sie die gewählte Zuordnung der Befehlscodes (obere 3 Bits im ROM) zu den Befehlen an. Erklären Sie die Funktion des Schaltungsteils für die Abarbeitung der bedingten und unbedingten Sprungbefehle. Kennzeichnen Sie Gattereingänge durch entsprechende Pfeile.

b) Wieviel mal (ganzzahlig) muß die Taktfrequenz des Microcontrollers größer sein als die Bitrate der seriellen Übertragung, wenn jedes Bit in der mittleren Hälfte (25% bis 75%) der Bitdauer vom Schieberegister SR eingelesen werden soll? Beachten Sie, daß der Program Counter mit der positiven Taktflanke inkrementiert wird bzw. einen neuen Zählerstand lädt, währenddessen die Datenausgabe bei der negativen Taktflanke erfolgt!

c) Wie lautet das Assemblerprogramm für den Microcontroller? Geben Sie für jede Adresse den Zustand aller Datenbits des ROMs an. Geben Sie an, welcher Adreßbereich im ROM nicht programmiert werden muß.

3.3. Programmschaltwerk zur Drehrichtungsbestimmung

Die Signale eines optischen Winkelgebers an einer Maschinenwelle sollen auf die Drehrichtung der Maschine durch ein Programmschaltwerk untersucht werden. Der Winkelgeber ist wie folgt konstruiert:



Ist das Feld schwarz, so gibt der Sensor 0 aus. Ist das Feld weiß, so gibt er 1 aus. Die Signale von b und a werden zum Eingangssignal e (b = MSB, a = LSB, Wertebereich = 0..3) zusammengefaßt.

Die Auswerteschaltung besitzt den Eingang e sowie den Ausgang a mit den Leitungen L und R (L = MSB, R = LSB, Wertebereich = 0..3), die zwei entsprechende Anzeigelampen einschalten (0 = dunkel, 1 = hell).

Bei Inbetriebnahme von Maschine und Auswerteelektronik steht die Maschine still und keine der beiden Lampen darf leuchten. Nach der ersten Vierteldrehung soll eine der beiden Lampen entsprechend der Drehrichtung eingeschaltet werden. Dreht sich die Welle bis zum nächsten Auswertezeitpunkt in die gleiche Richtung weiter oder bleibt der Winkel unverändert, so bleibt die entsprechende Anzeigelampe eingeschalten. Dreht sich die Welle um eine Vierteldrehung in die andere Richtung, so soll die Anzeige wechseln. Durch geeignete Anpassung der Abtasthäufigkeit an die maximale Maschinendrehzahl wird sichergestellt, daß sich die Welle um maximal 90° pro Abtastintervall dreht.

a) Geben Sie die zeitliche Abfolge des Eingangssignals e in Abhängigkeit von der Drehrichtung an.

b) Geben Sie die Schaltung eines Programmschaltwerkes mit dem Eingang e und dem Ausgang a an, das in der Lage ist, folgende Assemblerbefehle zu verarbeiten:

Befehl	Parameter	Binärcode	Erläuterung
SET	0..3	00xx xx--	Setzt den Ausgang a auf den angegebenen Wert
CP	0..3	01xx xx--	Vergleicht e mit dem angegebenen Parameter und setzt bei Gleichheit das Zerobit
JE	0..63	10-- ----	Springt, wenn das Zerobit gesetzt ist, auf die angegebene Adresse
JP	0..63	11-- ----	Springt auf die angegebene Adresse

0,1 ... vorgegeben, x ... beliebig, - ... mit Parameter belegt

Das Schaltwerk wird mit dem Taktsignal CLK versorgt. Mit einer positiven Flanke an CLK werden sowohl der Programmzähler inkrementiert bzw. mit einer Sprungadresse geladen als auch die Ergebnisse von Berechnungen (Zerobit, Ausgaberegister) in den zugehörigen Registern abgespeichert.

Als fertige Blöcke können verwendet werden: Ein Inkrementierer INC mit 6-Bit Ein- und Ausgang, ein 6 Bit breiter 2-zu-1 Multiplexer MUX, ein 64 Byte großes ROM, ein 2 x 2 Bit Komperator COMP, ein 2-zu-4 Dekoder DEK und diverse Gatter. Als speichernde Elemente sind positiv flankengetriggerte Register mit dem synchronen Enableeingang EN zu verwenden (der Wert am Eingang des Registers wird mit der positiven Flanke des Takteingangs CLK an den Ausgang des Registers übernommen, wenn EN zu diesem Zeitpunkt 1 ist). Über den Reset-Eingang kann der Ausgang des Registers beim Reset auf 0 gesetzt werden.

c) Wie lautet die absolute Reset-Adresse des Schaltwerks?

d) Geben Sie ein Programm für das entworfene Schaltwerk in der Form

[Adresse:] Opcode Parameter ; Erläuterung

an, das in der Lage ist die geforderten Funktionen zu erfüllen. Verwenden Sie der Einfachheit halber symbolische Adressen.

3.4. Programmschaltwerk mit Addierer

Es ist ein Programmschaltwerk zu entwerfen, welches nach dem Reset eine Binärzahl vom 4 Bit breiten Eingang IN einliest, die Zahl 5 dazu addiert und das Ergebnis am 8 Bit breiten Ausgangsport OUT ausgibt. Danach soll das Schaltwerk keine Veränderung an den Ausgängen mehr durchführen.

Die zur Verfügung stehenden Bausteine sind:

- Programm Counter PC: Ladbarer synchroner 4-Bit-Zähler (zählt bei positiver Taktflanke), low-aktiver Reset-Eingang, Steuereingang S (S = 0 count, S = 1 load).
- 16x7 Bit großes ROM
- 4-Bit-Volladdierer ADD: Zwei 4 Bit breite Eingänge, ein 4 Bit breiter Ausgang, Carryausgang CY.
- Multiplexer MUX: Zwei Eingänge, ein Ausgang, Steuereingang S zur Umschaltung zwischen den beiden Eingängen, die beiden Eingänge sind mit 0 und 1 (entsprechend Steuereingang S) zu beschriften.
- D-Flip-Flops in den Breiten 4 Bit und 1 Bit (OUT wird durch zwei getrennt gesteuerte 4-Bit-D-Flip-Flops realisiert) mit low-aktivem Reset-Eingang.
- Schaltnetz SN des Steuerwerks als Black Box (Block mit Ein- und Ausgangsleitungen ohne Innenschaltung). Die Übertragungsfunktion des Schaltnetzes wird in Punkt b) in Tabellenform angegeben.

Das Schaltwerk besitzt den Takteingang CLOCK sowie einen low-aktiven Reset-Eingang. Der Befehlssatz, den das Schaltwerk ausführen kann, lautet:

IN		ladet Akku mit Zahl am Eingangsport IN
CONST	Zahl	ladet Akku mit angegebener Zahl
ADDI		addiert Zahl am Eingangsport IN zum Akku
ADDC	Zahl	addiert angegebene Zahl zum Akku
OUTa		gibt Akku auf Ausgangsport OUTa aus

OUTb		gibt Akku auf Ausgangsport OUTb aus
JMP	Zahl	PC springt auf durch Zahl angegebene Adresse
JC	Zahl	Sprung auf durch Zahl angegebene Adresse, wenn Carryausgang von ADD logisch 1 ist

a) Zeichnen Sie das Schaltbild des beschriebenen Programmschaltwerks unter Verwendung der angegebenen Bauteile. Die Taktsignale müssen direkt an die Takteingänge der Bausteine gelegt werden (keine Verknüpfung über Inverter oder Gatter)! Die bedingten Aktionen werden durch Multiplexer vor den Eingängen der FFs realisiert. Alle getakteten Bauelemente sind mit der extern zur Verfügung gestellten Resetleitung zu verbinden. Durch den Reset werden alle angeschlossenen Bausteine auf 0 gesetzt. Für Busse sollen nur Striche mit Breitenangabe eingezeichnet werden.

b) Geben Sie die Wahrheitstabelle für das Schaltnetz SN des Steuerwerkes an.

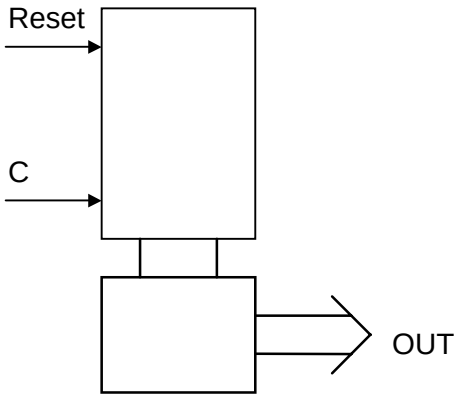
c) Geben Sie ein Programm für das entworfene Schaltwerk in der Form

[Adresse:] Opcode Parameter ; Erläuterung

an, das in der Lage ist die geforderten Funktionen zu erfüllen. Verwenden Sie der Einfachheit halber symbolische Adressen.

3.5. Multiplizierer

Es ist ein autonomes Programmschaltwerk zu entwerfen, welches in der Lage ist, einen Multiplikanden n mit einem Multiplikator m durch m-maliges addieren von n zu multiplizieren und das Ergebnis auf dem Bus OUT auszugegeben.



Das Programmschaltwerk ist eine 1-Operanden-Maschine mit dem Arbeitsregister W und den Hilfsregistern R_0, R_1 und R_2. Es kennt folgende Befehle:

SET	konst	Laden von W mit der Konstanten konst
LD	R_i	Laden von W mit dem Register R_i
STR	R_i	Speichern von W in das Register R_i
ADD	R_i	Addieren des Registers R_i zu W
DEC		Dekrementieren von W um 1
JNZ	Adr	Bedingter Sprung auf Adresse Adr wenn Zeroflag Z = 0 ist

JMP	Adr	Unbedingter Sprung auf Adresse Adr

Das Zeroflag Z soll nur durch die beiden Befehle *SET konst* und *DEC* beeinflußt werden.

a) Geben Sie das Programm für eine Multiplikation an. In R_0 ist der Multiplikand n, in R_1 der Multiplikator m und in R_2 das Ergebnis der Multiplikation abzulegen. Initialisieren Sie die Register mit den erforderlichen Werten, berechnen Sie in einer Schleife das geforderte Produkt und lassen Sie das Programm in einer Endlosschleife am Programmende enden. Die Konstanten n und m sind vom Typ unsigned integer und immer so klein, daß ihr Produkt ohne Überlauf in die Register paßt. Verwenden Sie der Einfachheit halber für die Konstanten und die Adressen symbolische Namen.

Das Programmschaltwerk besteht aus den drei Teilen Operatorblock, Operandenblock und Steuerwerk zerlegt.

Der Operatorblock enthält das Arbeitsregister W, die Logik für das Zeroflag, den Addierer ADD (Black Box), den Dekrementierer DEC (Black Box) sowie einen 4-zu-1-Busmultiplexer, der auswählt, mit welchem Wert das Arbeitsregister W geladen wird. Der Operatorblock besitzt als Eingang unter anderem den Bus R vom Operandenblock und den Datenbus D vom ROM.

Der Operandenblock enthält einen Dekoder DEK, der auswählt, welches Register geschrieben werden soll, die drei Register R_i und einen 3-zu-1-Busmultiplexer, der auswählt, welches Register dem Operatorblock über den Bus R zur Verfügung gestellt wird.

Das Steuerwerk enthält den ladbaren Program Counter PC, das ROM in erforderlicher Größe und das Schaltnetz des Steuerwerks SN (Black Box). Die Wahrheitstabelle des Schaltnetzes SN wird in Punkt c) in Tabellenform angegeben. Es sind unter anderem die Steuersignale c (Compute, 1 wenn W sich verändern soll), f (Flag, 1 wenn Zero-Flag sich verändern soll), s (Store, 1 wenn sich ein Register R_i verändern soll) und y_0 und y_1 (Auswahl des nächsten in W zu ladenden Werts) zu verwenden. Die Auswahl der Register R_i erfolgt über die beiden niederwertigsten Bits des Datenbusses D.

Das ganze Programmschaltwerk wird mit dem globalen Taktsignal C getaktet. Alle Register, Flip-Flops und der Program Counter werden über ein globales Resetsignal resetiert.

Alle Busse sind nur als einzelne Linien mit Breitenangabe auszuführen. Die Bauelemente können als einfache Kästchen gezeichnet werden. Als speichernde Elemente sind positiv flankengetriggerte Register mit dem synchronen Enableeingang E zu verwenden (der Wert am Eingang des Registers wird mit der positiven Flanke des Takteingangs an den Ausgang des Registers übernommen, wenn E zu diesem Zeitpunkt 1 ist). Bei Multiplexern und Dekodern sind die Anschlüsse zu numerieren. Die Register W und R_i besitzen eine Breite von 8 Bit. Für das ROM ist die Breite von Adreß- und Datenbus anzugeben. Die Signalrichtungen sind durch Pfeile zu markieren. Die angegebenen Bauteile und Interfaces sind **nicht** vollständig.

b) Geben Sie das Schaltbild des Schaltwerks an.

c) Geben Sie die Wahrheitstabelle für das Schaltnetz SN des Steuerwerks an.

4. Hardwarebeschreibungssprachen

4.1. Beschreibung von Multiplexern in VHDL

Im Rahmen dieses Beispiels sollen verschiedene Typen von Multiplexern in VHDL implementiert werden.

Aus welchem Zeichenvorrat setzen sich VHDL-Identifier zusammen? Geben Sie eine allgemeine Regel an.

a) Wie werden in VHDL Kommentare gekennzeichnet?

b) Was genau wird durch eine Entity bzw. eine Architecture beschrieben? Welche Abhängigkeit besteht zwischen Entity und Architecture?

c) Geben Sie die Entity für einen 2-zu-1-Multiplexer an, der jeweils eines der beiden Signale IN0 und IN1 in Abhängigkeit des Steuersignals SEL (IN0 für SEL = 0, IN1 für SEL = 1) an den Ausgang OUTMUX multiplext. Erklären Sie den Aufbau der Entity und der dabei verwendeten Statements.

d) Geben Sie eine Architecture an, die die Funktionalität des in Punkt d) beschriebenen 2-zu-1-Multiplexers implementiert. Der Multiplexer soll dabei durch eine bedingte Signalzuweisung (mittels *when*) realisiert werden. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

e) Wie lauten in VHDL die Operatoren für die boolschen Funktionen Negation, UND, ODER und Exklusiv-ODER?

f) Geben Sie eine Architecture an, die die Funktionalität des in Punkt d) beschriebenen 2-zu-1-Multiplexers implementiert. Der Multiplexer soll dabei über boolesche Operatoren realisiert werden. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements. Auf die Angabe des Headers kann hier ausnahmsweise verzichtet werden.

g) Geben Sie die Entity für einen 2x4-Bit-zu-1x4-Bit-Multiplexer an, der jeweils einen der beiden 4-Bit-Signalvektoren IN0 und IN1 in Abhängigkeit des Steuersignals SEL (IN0 für SEL = 0, IN1 für SEL = 1) an den 4-Bit-Ausgangsvektor OUTMUX multiplext. Auf die Angabe des Headers kann hier ausnahmsweise verzichtet werden.

4.2. Beschreibung von Flip-Flops und Registern in VHDL

Im Rahmen dieses Beispiels sollen verschiedene Typen von synchronen Flip-Flops und Registern in VHDL implementiert werden. Auf die Angabe des Headers kann in diesem Beispiel ausnahmsweise verzichtet werden.

a) Wie lautet die Richtlinie des Instituts zur Bezeichnung von low-aktiven Signalen? Welchen Sinn hat diese und ähnliche Richtlinien?

b) Was ist im Rahmen von Zuweisungen an Signalvektoren zu beachten? Wie werden Konstanten an Signale vom Typ *std_logic* und *std_logic_vector* zugewiesen? Was ist bei dabei zu beachten?

c) Geben Sie die Architecture für ein positiv flankengetriggertes D-Flip-Flop an. Das Flip-Flop besitzt den Dateneingang D, den Datenausgang Q und das Taktsignal CLK. Außerdem soll das Flip-Flop noch einen low-aktiven Reseteingang besitzen, mit dem der Ausgang Q des Flip-Flops asynchron zum Taktsignal auf logisch 0 gesetzt werden kann. Ein Reset soll dabei Priorität gegenüber dem Laden des Flip-Flops über den Eingang D haben. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements (insbesondere das *process*-Statement).

d) Was muß in der Architecture aus Punkt c) geändert werden, um das Flip-Flop in ein negativ flankengetriggertes Flip-Flop, bzw. ein Latch das für CLK = 1 transparent ist, umzuwandeln?

e) Geben Sie die Architecture für ein positiv flankengetriggertes 8-Bit breites Parallelregister an. Das Register besitzt den 8-Bit breiten Dateneingangsvektor D, den 8-Bit breiten Datenausgangsvektor Q, das synchrone Steuersignal LOAD, das Taktsignal CLK und das asynchrone low-aktive Resetsignal RESETB. Der 8-Bit-Wert am Eingang D soll an den 8-Bit-Ausgang Q mit der steigenden Flanke des Taktsignals CLK übernommen werden, wenn der Zustand von LOAD zu diesem Zeitpunkt logisch 1 ist. Andernfalls soll der alte Zustand Q im nächsten Takt beibehalten werden. Bei einem Reset soll der Ausgangsvektor Q des Flip-Flops asynchron zum Taktsignal auf logisch 0 gesetzt werden. Ein Reset soll dabei Priorität gegenüber dem Laden des Registers über LOAD haben. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

f) Geben Sie die Architecture für ein positiv flankengetriggertes 4-Bit-Schieberegister an. Das Register besitzt den seriellen Dateneingang SREGIN, den parallelen 4-Bit-Dateneingang LOADIN, den seriellen Datenausgang SREGOUT, die synchronen Steuersignale CLR, LOAD und SHIFT, das Taktsignal CLK und das asynchrone, high-aktive Resetsignal RESET. Alle 4 Stellen des Schieberegister sollen mit der steigenden Flanke des Taktsignals CLK auf 0 gesetzt werden, wenn der Eingang CLR zu diesem Zeitpunkt 1 ist. Der 4-Bit-Wert am Eingang LOADIN soll in die 4 Stellen des Schieberegisters mit der steigenden Flanke des Taktsignals CLK übernommen werden, wenn der Zustand von LOAD zu diesem Zeitpunkt logisch 1 ist. Alle 4 Stellen des Schieberegisters sollen mit der steigenden Flanke des Taktsignals CLK um eine Stelle, von der niederwertigeren zur höherwertigeren Stelle des Schieberegisters, geschoben werden, wenn der Eingang SHIFT zu diesem Zeitpunkt 1 ist. Der Wert des Eingangs SREGIN soll dabei in die niederwertigste Stelle des Schieberegisters übernommen werden. Ist mehr als eines der Signale CLR, LOAD und SHIFT logisch 1, soll CLR Priorität vor LOAD und LOAD Priorität vor SHIFT haben. Im Fall, daß alle 3 Steuersignale CLR, LOAD und SHIFT logisch 0 sind, soll der alte Zustand des Schieberegisters im nächsten Takt beibehalten werden. Bei einem Reset sollen alle 4 Stellen des Schieberegister asynchron zum Taktsignal auf logisch 0 gesetzt werden. Ein Reset soll dabei Priorität gegenüber allen synchronen Aktionen haben. Der Ausgang SREGOUT entspricht einfach der höchstwertigsten Stelle des Schieberegisters. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

g) Welche Teile im VHDL-Sourcecode sind für synchrone Schaltungen charakteristisch bzw. haben immer das gleiche Aussehen?

4.3. Beschreibung von Addierern, Subtrahierern und Zählern in VHDL

Im Rahmen dieses Beispiels sollen verschiedene Typen von Addierern, Subtrahierern und synchronen Zählern in VHDL implementiert werden. Auf die Angabe des Headers kann in diesem Beispiel ausnahmsweise verzichtet werden.

a) Geben Sie die Architecture für einen 4-Bit-Addierer an, der die beiden 4-Bit-Operanden OP1 und OP2 addiert und das Ergebnis am 4-Bit-Ausgang SUM ausgibt. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

b) Geben Sie die Architecture für einen 4-Bit-Subtrahierer an, der den 4-Bit-Operand OP2 vom 4-Bit-Operand OP1 subtrahiert und das Ergebnis am 4-Bit-Ausgang DIFF ausgibt. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

c) Geben Sie die Architecture für einen 4-Bit-Addierer mit Übertragsein- und Ausgang an, der die beiden 4-Bit-Operanden OP1 und OP2 und den Übertragseingang CIN addiert, am 4-Bit-Ausgang SUM ausgibt und bei einem Übertrag des Ergebnisses den Übertragsausgang COUT setzt. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

d) Geben Sie die Architecture für einen 6-Bit-Up-/Down-Counter an. Der Counter besitzt den 6-Bit breiten Eingang CNTIN, den 6-Bit breiten Zählerausgang CNTOUT, die synchronen Steuersignale LOAD, CNT und DIR, das Taktsignal CLK und das asynchrone, low-aktive Resetsignal RESETB. Der 6-Bit-Wert am Eingang CNTIN soll als neuer Zählerstand CNTOUT mit der steigenden Flanke des Taktsignals CLK in den Zähler geladen werden, wenn der Zustand von LOAD zu diesem Zeitpunkt logisch 1 ist. Der Zählerstand des Zählers CNTOUT soll mit der steigenden Flanke des Taktsignals CLK um 1 erhöht (DIR = 1) oder um 1 erniedrigt werden (DIR = 0), wenn der Zustand von COUNT zu diesem Zeitpunkt logisch 1 ist und LOAD dabei logisch 0 ist. Andernfalls soll der alte Zählerstand CNTOUT im nächsten Takt beibehalten werden. Bei einem Reset soll der Ausgang CNTOUT des Zählers asynchron zum Taktsignal auf logisch 0 gesetzt werden. Ein Reset soll dabei Priorität gegenüber allen synchronen Aktionen haben. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

e) Geben Sie die Architecture für einen 8-Bit-Aufwärtszähler an. Der Counter besitzt die synchronen Steuersignale COUNT, SET0, SET16 und SET64, den 8-Bit breiten Zählerausgang CNTOUT, den Ausgang CNTOUT100, das Taktsignal CLK und das asynchrone, low-aktive Resetsignal RESETB. Der Zählerstand CNTOUT soll mit der fallenden Flanke des Taktsignals CLK auf einen der dezimalen Werte 0, 16 oder 64 gesetzt werden, wenn der Zustand des entsprechenden Steuereingangs SET0, SET16 oder SET64 zu diesem Zeitpunkt logisch 1 ist. Der Zählerstand des Zählers CNTOUT soll hingegen mit der fallenden Flanke des Taktsignals CLK um 1 erhöht werden, wenn der Zustand von COUNT zu diesem Zeitpunkt logisch 1 ist und der Zählerstand kleiner als der dezimale Wert 199 ist. Sind mehrere der Signale SET0, SET16, SET64 und COUNT logisch 1, besitzen die Aktionen die folgende Priorität: SET0, SET16, SET64, COUNT. Der Ausgang CNTOUT100 soll schließlich zumindest für eine Taktperiode lang logisch 1 sein, wenn der Zähler den dezimalen Zählerstand 100 erreicht hat. Bei einem Reset soll der Ausgang CNTOUT des Zählers asynchron zum Taktsignal auf logisch 0 gesetzt werden. Ein Reset soll dabei Priorität gegenüber allen synchronen Aktionen haben. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

f) Erklären Sie den Unterschied zwischen VHDL und Low-Level-Gatterbeschreibungssprachen. Welche Aufgaben hat das Synthesetool in VHDL?

g) Vergleichen Sie die Erstellung von Hardware mittels VHDL mit dem Erstellen lauffähiger Programme über eine Hochsprache. Welche Unterschiede bzw. Gemeinsamkeiten lassen sich feststellen? Was kann als eine der wichtigsten Regeln bei der Erstellung von Hardware mittels VHDL angesehen werden?

4.4. Beschreibung von Decodern in VHDL

Im Rahmen dieses Beispiels sollen verschiedene Typen von Decodern in VHDL implementiert werden. Auf die Angabe des Headers kann in diesem Beispiel ausnahmsweise verzichtet werden.

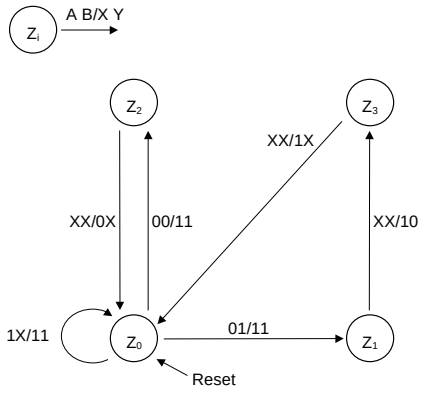
a) Geben Sie die Architecture für den Befehlsdecoder aus Beispiel 3.4 an. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

b) Geben Sie die Architecture für das ROM aus Beispiel 3.4 an. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements. Was sind Makrozellen?

4.5. Beschreibung von Moore- und Mealy-Automaten in VHDL

Im Rahmen dieses Beispiels sollen Moore- und Mealy-Automaten in VHDL implementiert werden. Auf die Angabe des Headers kann in diesem Beispiel ausnahmsweise verzichtet werden.

a) Geben Sie die Architecture für den in der folgenden Abbildung gegebenen Moore-Automaten an. Der Automat soll mit positiv flankengetriggerten Flip-Flops realisiert werden. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.



b) Ändern Sie die Architecture für den unter Punkt a) realisierten Automaten dahingehend, daß der Ausgang X des Automaten frei von Hazards ist.

c) Geben Sie die Architecture für den in Beispiel 2.4 gegebenen Mealy-Automaten an. Der Automat soll mit positiv flankengetriggerten Flip-Flops realisiert werden und bei einem Reset den Zustand Z0 annehmen. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements. Wie erkennt man an Automaten die nach der Struktur von Punkt a) und c) aufgebaut wurden im VHDL-Code auf den ersten Blick, ob es sich um einen Moore- oder einen Mealy-Automaten handelt?

d) Kann bei einer Beschreibung des in Beispiel 2.4 gegebenen Mealy-Automaten in VHDL, alleine durch die in Beispiel 2.4, Punkt e) gegebene Codierung eine Hazard-Freiheit des Ausgangssignals RDY garantiert werden?

4.6. Aufbau von hierarchischen Designs mittels VHDL

Implementieren Sie den Microcontroller aus Beispiel 3.2 in VHDL. Nehmen Sie dazu an, daß die in der Angabe des Beispiels 3.2 gegebenen Blöcke Counter, ROM, Ausgangsregister und Decoder bereits als fertige Entity-/Architecture-Paare existieren und binden Sie diese Komponenten in das Design des Microcontrollers ein (die Entities und Architectures dieser Module müssen nicht angegeben werden). Geben Sie die Architecture des Microcontrollers an. Erklären Sie den Aufbau der Architecture und der dabei verwendeten Statements.

5. Simulation, Testen und Design Rules

5.1. Design for Testability

Die Testbarkeit des Schaltwerks aus Beispiel 3.4 soll durch das Hineinführen bzw. Herausführen zusätzlicher Signale in bzw. aus dem Chip erhöht werden. Dazu soll einerseits die Beobachtbarkeit und andererseits die Steuerbarkeit des Systems erhöht werden.

a) Was versteht man beim ASIC-Design unter den Begriffen Simulation, Testen und Design Rules? Welche Zusammenhänge existieren zwischen den drei Begriffen?

b) Erfolgen Spezifikation, Beschreibung und Simulation eines ASICs Top Down oder Bottom Up?

c) Erläutern Sie die Begriffe Beobachtbarkeit und Steuerbarkeit im Bezug auf die Testbarkeit eines Systems an einem einfachen Beispiel.

d) Erweitern Sie das Schaltbild des Schaltwerks aus Beispiel 3.4 um die Testbarkeit des Designs zu erhöhen. Dazu sollen einerseits die Ausgangssignale des Schaltnetzes SN außerhalb des Chips sichtbar gemacht werden. Des weiteren soll der Program Counter PC im Testmodus mit einer beliebigen Adresse von außen zu laden sein. Am Gehäuse des Chips sind noch 11 unbenutzte Pins vorhanden. Geben Sie das Schaltbild der erweiterten Schaltung an. Wird durch die beiden zusätzlichen Funktionen die Beobachtbarkeit oder die Steuerbarkeit der Schaltung erhöht?

e) Erläutern Sie die Funktion eines bidirektionalen Pads.

f) Durch den Einsatz einer neuen Gehäuseform haben Sie nur mehr 2 unbenutzte Pins zur Verfügung um die geforderten Testfunktionen zu realisieren. Geben Sie das Schaltbild der erweiterten Schaltung an, welche die geforderten Testfunktionen realisiert. Sie können dabei annehmen, daß der Zustand der beiden 4-Bit-Ausgangssignale A und B nicht von Interesse ist, solange sich die Schaltung im Testmodus befindet.

5.2. Simulation in VHDL

Zur Simulation des D-Flip-Flops aus Beispiel 4.5, Punkt c) ist eine VHDL-Testbench zu schreiben.

a) Was versteht man unter Logiksimulation und Gate Level Simulation? Was sind die Vor- und Nachteile beider Simulationsmethoden?

b) Geben Sie eine Testbench zur Simulation des D-Flip-Flops aus Beispiel 4.5, Punkt c) an. Erklären Sie den Aufbau der Testbench und der dabei verwendeten Statements. Auf die Angabe des Headers kann in diesem Beispiel ausnahmsweise verzichtet werden.

c) Was hätte es für Auswirkungen, wenn das Signal D in die Sensitivity List des in Beispiel 4.5, Punkt c) abgebildeten Prozesses *D_Flip_Flop* hinzugefügt werden würde?

d) Ist die in Punkt b) entwickelte Testbench synthetisierbar? Begründen Sie ihre Antwort.

e) Mit welchen Problemen hat man als Entwickler zu kämpfen, wenn für Simulationen oder Tests innerer Signale an das Top Level Design des Chips herausgeführt werden müssen?

5.3. Scan-Path-Design

Der Mealy-Automat aus Beispiel 2.3 soll als Scan-Path Design realisiert werden. Dazu werden alle D-Flip-Flops des Designs durch positiv flankengetriggerte Scan-Flip-Flops ersetzt. Anschließend soll ein Testpattern entworfen werden, welches den Automaten in einen definierten Zustand bringt, von welchem aus das weitere Verhalten des Automaten im Normalbetrieb überprüft werden kann.

a) Geben Sie das Schaltbild eines Scan-Flip-Flops mit allen Ein- und Ausgängen an. Durch welche einfache Schaltung kann ein normales D-Flip-Flop in ein Scan-Flip-Flop umgewandelt werden?

b) Geben Sie das Schaltbild des Mealy-Automaten mit Scan-Flip-Flops an. Die Kombinatorik des Automaten kann dabei als Black Box (Modul mit Ein- und Ausgängen) gezeichnet werden.

c) Um wieviele Ein- und Ausgänge erhöht sich die Anzahl der Signale des Automaten bedingt durch das Scan-Path-Design? Wie heißen die neuen Ein- und Ausgänge? Erklären Sie die Funktion der Scan-Path-Kette.

d) Geben Sie das Timing aller benötigten Signale eines Testpatterns an, um den Automaten in den Zustand e zu versetzen. Anmerkung: Das Testpattern hängt von der gewählten Zustandscodierung des Automaten ab. Beschreiben Sie die Vorgangsweise.

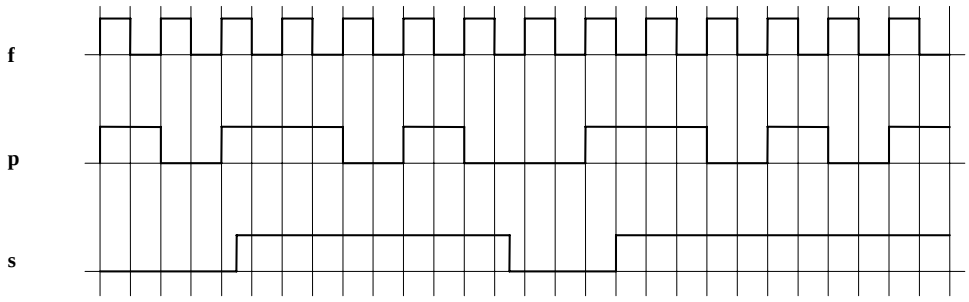
e) Nachdem der Automat über die Scan-Path-Kette in den Zustand e versetzt wurde, soll nach zwei weiteren aktiven Taktflanken (während denen der Automat sich im Normalbetrieb befindet) der aktuelle Zustand des Automaten über die Scan-Path-Kette ausgelesen werden. Der Eingang x des Automaten ist konstant auf 1 gelegt. Geben Sie das Timing aller benötigten Signale eines Testpatterns an, um den Zustand des Automaten auszulesen. Beschreiben Sie die Vorgangsweise. In welchem Zustand befindet sich der Automat?

f) Welche Auswirkungen hat ein Vergattern von Takteingängen von Flip-Flops im Bezug auf Scan-Path-Design?

1. Moore-Automaten

1.1 Phasenschieber

a) Vollständiges Zeitdiagramm:



b) Sinnvolle Wertebereiche von s und daraus folgende Wertebereiche von p:

Minimalwert der Frequenz von s:

$f_{s,min} = 0 \Rightarrow f_p = f/2$ (laut Angabe)

Maximalwert der Frequenz von s:

$f_{s,max} = f/2 \Rightarrow f_p = 0$ (die Halbwelle von p wird ins Unendliche verlängert)

$f_s > f/2$ wäre sinnlos, da Eingangssignale von synchronen Automaten nur ein Mal pro Takt abgefragt werden.

Dazwischen:

z.B.: $f_s = f/4 \Rightarrow f_p = f/4$

c) Haben alle verwendeten Bauteile ein Propagation Delay von 0, bedeutet dies, daß die Ausgänge des Automaten ihre aktuellen Zustände direkt nach der aktiven Taktflanke des Automaten annehmen. Im Timing-Diagramm der Angabe aus Punkt a) ist zu sehen, daß der Ausgang p seinen Zustand sofort nach der steigenden Taktflanke des Taktes f wechselt. Damit muß die steigende Taktflanke die aktive Taktflanke sein und der Automat muß mit positiv flankengetriggerten Flip-Flops realisiert werden.

d) Bei einem Mealy-Automaten wäre laut Definition das Ausgangssignal p des Automaten direkt vom Eingangssignal s (welches asynchron zu f ist) abhängig. Nach der Angabe erfolgen Zustandswechsel des Signals p aber nur zu Vielfachen der Periodendauer 1/f des Taktes $\Rightarrow$ Moore-Automat.

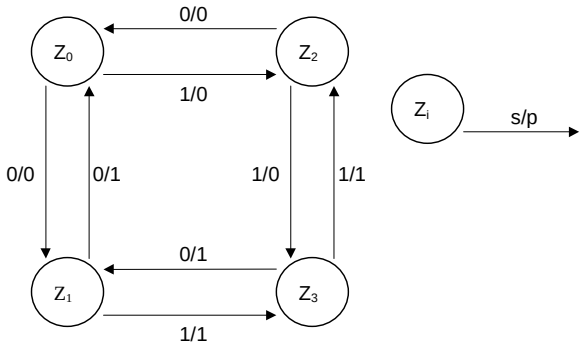
e) Im allgemeinen läßt sich die Anzahl der benötigten Zustände eines Automaten erst nach dem Zeichnen von Zustandsgraph bzw. Zustandsübergangstabelle bestimmen. Bei diesem einfachen Beispiel kann die Anzahl der benötigten Zustände jedoch bereits im vorhinein durch folgende

Überlegung bestimmt werden: Der Zustand des Signals p in der nächsten Halbwelle von p hängt von den Zuständen der beiden Signale p und s in der aktuellen Halbwelle von p ab. Es gibt 4 mögliche Signalkombinationen von p und s, was wiederum 4 Zuständen des Automaten entspricht.

f) Die Ausgangsgleichung für p wird dann möglichst einfach, wenn der Zustand eines Flip-Flops Q_i^n direkt als Ausgang p verwendet wird. Dies wird durch entsprechende Codierung der Zustände $Z_i^n \rightarrow Q_1^n Q_0^n$ erreicht. Damit degeneriert die Ausgangsgleichung für p zu:

$$p = Q_i^n$$

g) Zustandsgraph:



Ändert sich der Zustand des Eingangssignals s nicht, so pendelt der Automat immer zwischen den Zuständen $Z_0 \leftrightarrow Z_1$ bzw. $Z_2 \leftrightarrow Z_3$ hin und her und ändert dabei den Zustand des Ausgangs p. Ändert sich hingegen der Zustand von s, so wechselt der Automat zwischen den Zuständen $Z_0 \leftrightarrow Z_2$ bzw. $Z_1 \leftrightarrow Z_3$ und p ändert sich bei diesem Übergang nicht.

h) Übergangstabelle:

s^n	Q_1^n	Q_0^n	Z^n	p^n	Q_1^{n+1}	Q_0^{n+1}	Z^{n+1}
0	0	0	Z_0	0	0	1	Z_1
0	0	1	Z_1	1	0	0	Z_0
0	1	0	Z_2	0	0	0	Z_0
0	1	1	Z_3	1	0	1	Z_1
1	0	0	Z_0	0	1	0	Z_2
1	0	1	Z_1	1	1	1	Z_3
1	1	0	Z_2	0	1	1	Z_3
1	1	1	Z_3	1	1	0	Z_2

Unter Berücksichtigung von Punkt f) wurde $Q_0^n = p^n$ gewählt. Weiters wurde Q_1^n so gewählt, daß sich für $Z_i^n \rightarrow Q_1^n\, Q_0^n$ eine binäre Codierung ergibt (diese Codierung wurde allerdings vollkommen willkürlich so gewählt).

i) Realisierung mit D-Flip-Flops:

Charakteristische Gleichung für D-Flip-Flops:

$$Q^{n+1} = D^n$$

Q_0^{n+1}:

Q_1^n \ Q_0^n	0	1	0	1
0	1	0	1	0
1	0	1	0	1

s^n

Keine Vereinfachung möglich $\Rightarrow$

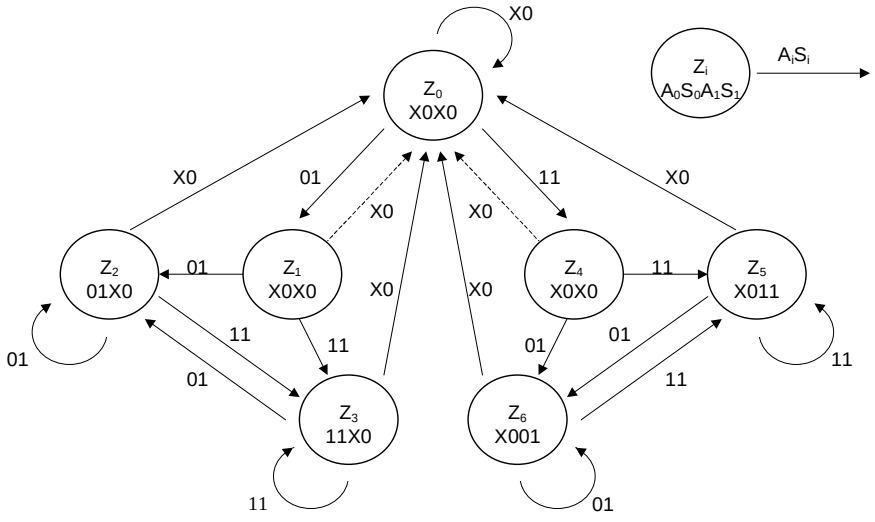
$$D_0{}^n = Q_0{}^{n+1} = \overline{S^n} Q_1{}^n Q_0{}^n \vee \overline{S^n}\,\overline{Q_1{}^n}\,\overline{Q_0{}^n} \vee S^n Q_1{}^n \overline{Q_0{}^n} \vee S^n \overline{Q_1{}^n} Q_0{}^n$$

Die Gleichungen für $D_1{}^n$ und p können einfach aus der Übergangstabelle abgelesen werden:

$$D_1{}^n = Q_1{}^{n+1} = s^n$$
$$p^n = Q_0{}^n$$

1.2. Schaltknoten

a), b) Zustandsgraph:



Die linke Seite des Zustandsgraphen ist für den Ausgangskanal 0, die rechte Seite des Zustandsgraphen ist für den Ausgangskanal 1 zuständig. Um jenen Fehlerfall zu berücksichtigen, daß überhaupt nur ein Adreßbit übertragen wird, ist der Zustandsgraph durch die Übergänge $Z_1 \rightarrow Z_0$ und $Z_4 \rightarrow Z_0$ zu ergänzen.

c) Der Automat besitzt 7 Zustände. Mit 3 Flip-Flops können 8 Zustände realisiert werden. Somit werden also 3 Flip-Flops benötigt.

d) Zustandsübergangstabelle:

Die Zustände werden hier ganz einfach binär codiert. Wie allerdings schon im Vorlesungsskriptum erläutert ist, kann sich durch die Wahl der Codierung eine Vereinfachung bzw. Verkomplizierung der kombinatorischen Logik des Automaten ergeben. Es kann also im konkreten Fall durchaus sein, daß Sie durch eine andere Codierung einfachere Gleichungen für A_0, S_0 und D_1 erhalten.

Übergangstabelle:

A_i	S_i	Z^n	Q_2^n	Q_1^n	Q_0^n	Z^{n+1}	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}	A_0	S_0	A_1	S_1
X	0	Z_0	0	0	0	Z_0	0	0	0	X	0	X	0
X	0	Z_1	0	0	1	Z_0	0	0	0	X	0	X	0
X	0	Z_2	0	1	0	Z_0	0	0	0	0	1	X	0
X	0	Z_3	0	1	1	Z_0	0	0	0	1	1	X	0
X	0	Z_4	1	0	0	Z_0	0	0	0	X	0	X	0
X	0	Z_5	1	0	1	Z_0	0	0	0	X	0	1	1
X	0	Z_6	1	1	0	Z_0	0	0	0	X	0	0	1
X	0	Z_7	1	1	1	Z_0	0	0	0	X	0	X	0
0	1	Z_0	0	0	0	Z_1	0	0	1	X	0	X	0
0	1	Z_1	0	0	1	Z_2	0	1	0	X	0	X	0
0	1	Z_2	0	1	0	Z_2	0	1	0	0	1	X	0
0	1	Z_3	0	1	1	Z_2	0	1	0	1	1	X	0
0	1	Z_4	1	0	0	Z_6	1	1	0	X	0	X	0
0	1	Z_5	1	0	1	Z_6	1	1	0	X	0	1	1
0	1	Z_6	1	1	0	Z_6	1	1	0	X	0	0	1
0	1	Z_7	1	1	1	Z_0	0	0	0	X	0	X	0
1	1	Z_0	0	0	0	Z_4	1	0	0	X	0	X	0
1	1	Z_1	0	0	1	Z_3	0	1	1	X	0	X	0
1	1	Z_2	0	1	0	Z_3	0	1	1	0	1	X	0
1	1	Z_3	0	1	1	Z_3	0	1	1	1	1	X	0
1	1	Z_4	1	0	0	Z_5	1	0	1	X	0	X	0
1	1	Z_5	1	0	1	Z_5	1	0	1	X	0	1	1
1	1	Z_6	1	1	0	Z_5	1	0	1	X	0	0	1
1	1	Z_7	1	1	1	Z_0	0	0	0	X	0	X	0

Da mit 3 Flip-Flops 8 Zustände kodiert werden können, der Automat aber nur 7 Zustände hat, wird ein Zustand nicht benötigt. Im Normalfall befindet sich der Automat niemals im Zustand Z7. Durch Verletzung von Setup- bzw. Hold-Zeiten der Flip-Flops des Automaten, kann es in der Realität jedoch prinzipiell durchaus vorkommen, daß der Automat in den Zustand Z7 verzweigt. Für die Wahl des Folgezustands sowie der Ausgangssignale in nicht verwendeten Zuständen gibt es nun mehrere Möglichkeiten:

1) Aufgrund des eher theoretischen Charakters der Aufgabenstellung wird davon ausgegangen, daß der Automat niemals in einen nicht verwendeten Zustand verzweigt. Damit ist der Folgezustand und der Zustand der Ausgangssignale beliebig und für die Folgezustände Q_i^{n+1} sowie für die Ausgangssignale a_i^n kann in der Übergangstabelle ein X geschrieben werden. Die zur Erzeugung der Ausgangssignale und der Folgezustände benötigte kombinatorische Logik kann sich dadurch dementsprechend vereinfachen. Dies bedingt wiederum, daß man zur Lösung der Übungsbeispiele weit weniger Zeit als vorhergesehen benötigt, man zum Telephonhörer greifen kann um seine(n) Freundin bzw. Freund anzutelephonieren und mit

den Worten „Ich bin fertig mit dem Mist. Kino geht sich also heute doch noch aus ...“ zum gemütlichen Teil des Abends übergehen kann.

2) Der Automat bleibt, so er sich in einem nicht verwendeten Zustand befindet, in demselbigen. In diesem Zustand wird eine spezielle Kombination der Ausgangssignale generiert, wodurch dem Benutzer das Auftreten eines nicht erwünschten Zustandes signalisiert wird. Durch einen vom Benutzer initiierten Reset des Automaten kann dieser (der Automat, nicht der Benutzer) anschließend wieder in einen „normalen“ Zustand gebracht werden.

3) Als Folgezustand wird ein „sinnvoller“ Zustand des Automaten gewählt. Ein „sinnvoller“ Zustand kann etwa der Zustand nach einem Reset des Automaten oder der Ruhezustand des Automaten sein. Was genau ein „sinnvoller“ Zustand ist, hängt jeweils von der Aufgabenstellung ab und kann nicht allgemein beantwortet werden. Auf jeden Fall muß darauf geachtet werden, daß in einem nicht verwendeten Zustand des Automaten die Ausgangssignale keine „gefährlichen“ Zustände annehmen. Was ein „gefährlicher“ Zustand ist, hängt wiederum von der Aufgabenstellung ab.

Ich empfehle sowohl in der Praxis, als auch für die Übungen, solange von der Angabe nichts anderes gefordert wird, Variante 3 (Sorry, nix is mit Kino ...). Als Folgezustand für Zustand Z_7 wird hier also Zustand Z_0 gewählt, was der Ruhezustand des Automaten ist. Wichtig ist außerdem, daß der Zustand der Ausgangssignale S_0 und S_1 im Zustand Z7 low ist.

e) Bestimmung der Gleichungen für A_0, S_0 und D_1:

Wie aus der obigen Übergangstabelle zu sehen ist, sind A_0 und S_0 von A_i und S_i unabhängig, wodurch sich die KV-Diagramme für A_0 und S_0 auf drei Variable beschränken.

A_0^n:

Q_2^n \ Q_1^n				
	X	X	X	X
	0	1	X	X
		Q_0^n		

$$A_0 = Q_0^{\,n}$$

Nachdem in der Angabe nichts über den Zustand von A_0 ausgesagt wird während S_0 low ist, kann durch entsprechende Wahl der „don`t care“-Felder im KV-Diagramm die Gleichung für A_0 erheblich vereinfacht werden.

S_0^n:

Q_2^n \ Q_1^n				
	0	0	0	0
	1	1	0	0
		Q_0^n		

$$S_0 = \overline{Q_2^{\,n}} Q_1^{\,n}$$

Charakteristische Gleichung für D-Flip-Flops:

$Q^{n+1} = D^n$

Q_1^{n+1} (für $S_i = 1$):

Q_0^n \ Q_1^n	0	1	2	3
0	1	1	1	1
1	0	0	0	1
2	1	0	0	1
3	1	1	0	0

Q_2^n, A_i

$$D_1^n = Q_1^{n+1} = S_i \wedge (Q_1^n \cdot \overline{Q_2^n} \vee Q_0^n \cdot \overline{Q_2^n} \vee \overline{Q_1^n} \cdot Q_2^n \cdot \overline{A_i} \vee \overline{Q_0^n} \cdot Q_1^n \cdot \overline{A_i})$$

f) Für eine Vermittlungsanlage mit einem Sender und 6 Empfängern werden 6 Schaltknoten benötigt. Durch die Hintereinanderschaltung von 3 Schaltknoten werden pro Paket 3 Adreßbits benötigt. Zusammen mit den 8 Datenbits ergibt sich eine Paketlänge von 11 Bits.

1.3. Gray-Code Zähler

a) Charakteristische Gleichung des JK-FF:

$$Q^{n+1} = J^n \cdot \overline{Q^n} \vee \overline{K^n} \cdot Q^n$$

b) Übergangstabellen des JK-FF:

J	K	Q^{n+1}
0	0	Q^n
0	1	0
1	0	1
1	1	$\overline{Q^n}$

Q^n	Q^{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

c), d) KV-Diagramm für Q_3^{n+1}, Q_2^{n+1}, Q_1^{n+1} und Q_0^{n+1}:

Q_3 \ Q_0																
	1	1	1	0	1	1	0	0	1	1	0	1	1	0	0	1
	1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0
	0	1	1	0	1	0	1	1	0	0	1	1	1	0	0	0
	1	1	1	0	1	1	1	1	1	0	1	1	1	0	1	0
	0	1	0	0	1	1	1	1	0	0	0	1	1	0	1	0
	0	1	1	0	0	1	1	1	0	0	1	1	0	0	1	0
	0	1	0	1	0	1	1	1	0	0	0	0	0	0	1	0
	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0

Q_1 (rechts) Q_2 (unten)

e) KV-Diagramme für J_3^n, J_2^n, J_1^n, J_0^n und K_3^n, K_2^n, K_1^n, K_0^n:

J_3^n:

Q_3 \ Q_0				
	X	X	X	X
	X	X	X	X
	0	1	0	1
	0	0	0	0

Q_1 (rechts) Q_2 (unten)

K_3^n:

Q_3 \ Q_0				
	0	0	0	0
	1	0	1	0
	X	X	X	X
	X	X	X	X

Q_1 (rechts) Q_2 (unten)

J_2^n:

Q_3 \ Q_0				
	X	X	1	0
	X	X	0	0
	X	X	0	0
	X	X	0	0

Q_1 (rechts) Q_2 (unten)

K_2^n:

Q_3 \ Q_0				
	0	0	X	X
	0	1	X	X
	0	0	X	X
	0	0	X	X

Q_1 (rechts) Q_2 (unten)

J_1^n:

Q_3 \ Q_0Q_1	00	01	11	10
00	1	0	0	0
01	X	X	X	X
11	X	X	X	X
10	0	1	0	1

K_1^n:

Q_3 \ Q_0Q_1	00	01	11	10
00	X	X	X	X
01	0	0	0	1
11	1	0	1	0
10	X	X	X	X

J_0^n:

Q_3 \ Q_0Q_1	00	01	11	10
00	0	X	X	1
01	0	X	X	0
11	0	X	X	0
10	1	X	X	0

K_0^n:

Q_3 \ Q_0Q_1	00	01	11	10
00	X	1	0	X
01	X	0	0	X
11	X	0	0	X
10	X	0	1	X

Die Bestimmung der KV-Diagramme für J_3^n, J_2^n, J_1^n, J_0^n und K_3^n, K_2^n, K_1^n, K_0^n geschieht über das KV-Diagramm aus Punkt c)-d) und der rechten Übergangstabelle aus Punkt b). Für das Feld in der linken unteren Ecke des KV-Diagramms aus Punkt c)-d) ist etwa ein Übergang von $Q_0^n=0 \rightarrow Q_0^{n+1}=1$ gefordert. Aus der rechten Übergangstabelle von Punkt b) sieht man, daß für solch einen Übergang $J_0^n=1$ und $K_0^n=X$ sein muß. Für den Übergang $Q_1^n=0 \rightarrow Q_1^{n+1}=0$ (ebenfalls in der linken unteren Ecke des KV-Diagramms) muß hingegen laut Übergangstabelle $J_1^n=0$ und $K_1^n=X$ sein. Auf diese Weise erhält man die Werte für alle Felder der KV-Diagramme für J_3^n, J_2^n, J_1^n, J_0^n und K_3^n, K_2^n, K_1^n, K_0^n.

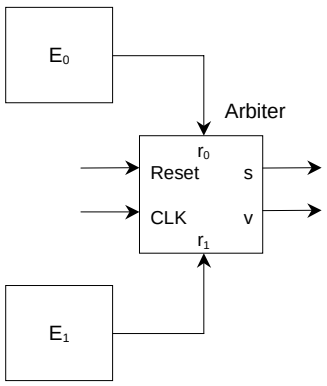
e) Bestimmung der Gleichungen für alle J und K aus dem KV-Diagramm:

$$J_3 = Q_2 \cdot Q_1 \cdot Q_0 \vee \overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} \qquad J_2 = Q_3 \cdot \overline{Q_1} \cdot Q_0$$
$$K_3 = Q_2 \cdot Q_1 \cdot \overline{Q_0} \vee \overline{Q_2} \cdot Q_1 \cdot Q_0 \qquad K_2 = Q_3 \cdot Q_1 \cdot Q_0$$

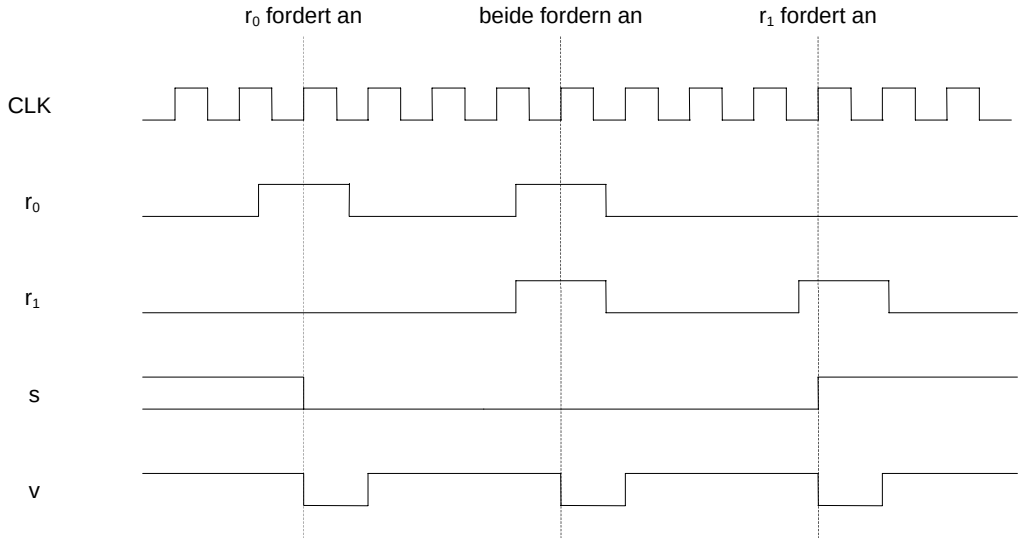
$$J_1 = Q_3 \cdot Q_2 \cdot \overline{Q_0} \vee \overline{Q_3} \cdot Q_2 \cdot Q_0 \vee \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_0} \qquad J_0 = Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1} \vee \overline{Q_3} \cdot Q_2 \cdot \overline{Q_1}$$
$$K_1 = Q_3 \cdot \overline{Q_2} \cdot \overline{Q_0} \vee \overline{Q_3} \cdot Q_2 \cdot \overline{Q_0} \vee \overline{Q_3} \cdot \overline{Q_2} \cdot Q_0 \qquad K_0 = Q_3 \cdot Q_2 \cdot \overline{Q_1} \vee \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1}$$

1.4. Arbiter

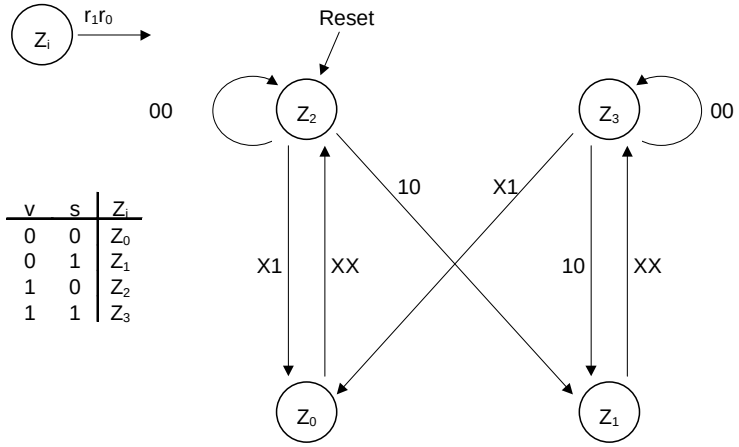
a) Blockschaltbild des Arbiters und der beiden Anforderungseinheiten:



b) Timing eines Anforderungszyklus:



c) Zustandsübergangsgraph und Zustandsübergangstabelle:



v	s	Z_i
0	0	Z_0
0	1	Z_1
1	0	Z_2
1	1	Z_3

	r_1^n	r_0^n	Q_1^n	Q_0^n	Q_1^{n+1}	Q_0^{n+1}	v^n	s^n
Z_0	X	X	0	0	1	0	0	0
Z_1	X	X	0	1	1	1	0	1
Z_2	0	0	1	0	1	0	1	0
Z_2	1	0	1	0	0	1	1	0
Z_2	X	1	1	0	0	0	1	0
Z_3	0	0	1	1	1	1	1	1
Z_3	1	0	1	1	0	1	1	1
Z_3	X	1	1	1	0	0	1	1

Die Codierung der Zustände wurde so gewählt, daß s und v direkt von den Ausgängen der Flip-Flops abgenommen werden können. Nachdem die Gleichungen für s und v aber in der Angabe nicht gefragt sind, ist die Wahl der Zustandscodierung bei diesem Beispiel im Prinzip egal.

d) Bestimmung der Gleichungen für alle J und K mit der Methode des Koeffizientenvergleichs:

Q_0^{n+1}:

Q_0^n \ Q_1^n	0	1	1	1
1	0	1	1	
1	0	0	0	r_1^n
0	0	0	0	
	r_0^n			

$$Q_0^{n+1} = \overline{Q_1^n} \cdot Q_0^n \vee \overline{r_0^n} \cdot Q_0^n \vee \underbrace{r_1^n \cdot \overline{r_0^n} \cdot Q_1^n}_{\cdot (Q_0^n \vee \overline{Q_0^n})}$$

Q_1^{n+1}:

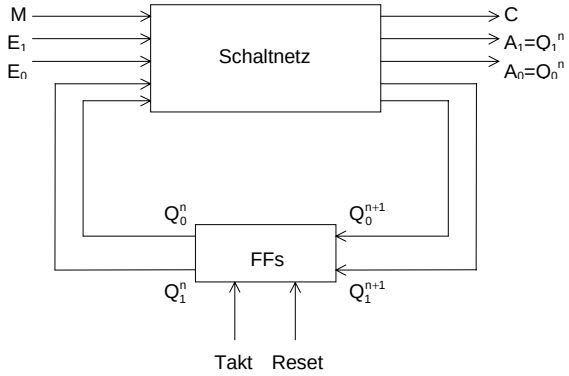
Q_0^n \ Q_1^n	0	1	1	1
1	0	1	1	
0	0	1	1	r_1^n
0	0	1	1	
1	0	1	1	
	r_0^n			

$$Q_1^{n+1} = \overline{Q_1^n} \vee \underbrace{\overline{r_1^n} \cdot \overline{r_0^n}}_{\cdot (Q_1^n \vee \overline{Q_1^n})}$$

2. Mealy-Automaten

2.1. Zweistelliger Binärzähler mit Übertragsausgang

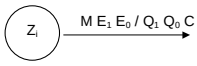
a) Blockschaltbild des Zählers:

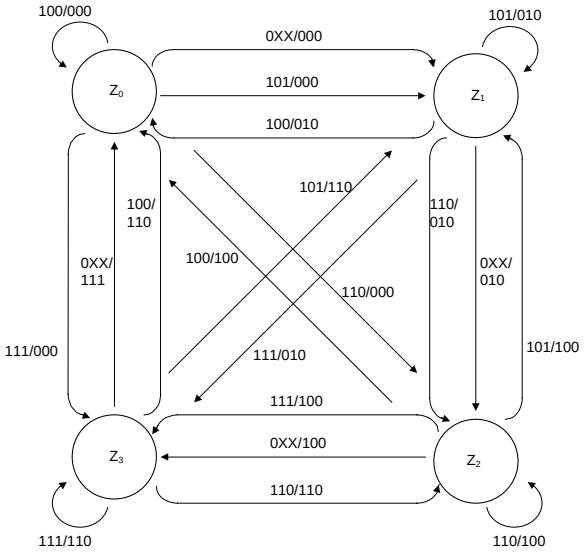


b) Zustandsübergangstabelle:

M	E_1	E_0	Q_1^n	Q_0^n	Q_1^{n+1}	Q_0^{n+1}	C^n	J_1^n	K_1^n	S_1^n	R_1^n
0	X	X	0	0	0	1	0	0	X	0	X
0	X	X	0	1	1	0	0	1	X	1	0
0	X	X	1	0	1	1	0	X	0	X	0
0	X	X	1	1	0	0	1	X	1	0	1
1	0	0	0	X	0	0	0	0	X	0	X
1	0	0	1	X	0	0	0	X	1	0	1
1	0	1	0	X	0	1	0	0	X	0	X
1	0	1	1	X	0	1	0	X	1	0	1
1	1	0	0	X	1	0	0	1	X	1	0
1	1	0	1	X	1	0	0	X	0	X	0
1	1	1	0	X	1	1	0	1	X	1	0
1	1	1	1	X	1	1	0	X	0	X	0

c) Zustandsgraph:





d) Die Lösungsfunktion D_1^n für die Realisierung mittels eines D-FFs läßt sich gleich aus der Übergangstabelle ablesen:

$$D_1^n = Q_1^{n+1} = M \cdot E_1 \vee \overline{M} \cdot Q_0^n \cdot \overline{Q_1^n} \vee \overline{M} \cdot \overline{Q_0^n} \cdot Q_1^n = M \cdot E_1 \vee \overline{M} \cdot (Q_0^n \circledR Q_1^n)$$ siehe Fußnote[3]

e) Lösungsfunktionen J_1^n und K_1^n für die Realisierung mittels JK-FF durch Koeffizientenvergleich:

Charakteristische Gleichung des JK-FF:

$$Q^{n+1} = J^n \cdot \overline{Q^n} \vee \overline{K^n} \cdot Q^n$$

Erweitern des ersten Minterms aus der Gleichung für Q_1^{n+1} aus Punkt d) mit $(Q_1^n \vee \overline{Q_1^n})$[4]:

$$Q_1^{n+1} = M \cdot E_1 \cdot Q_1^n \vee M \cdot E_1 \cdot \overline{Q_1^n} \vee \overline{M} \cdot Q_0^n \cdot \overline{Q_1^n} \vee \overline{M} \cdot \overline{Q_0^n} \cdot Q_1^n$$

$$Q_1^{n+1} = \underbrace{(M \cdot E_1 \vee \overline{M} \cdot Q_0^n)}_{J_1^n} \cdot \overline{Q_1^n} \vee \underbrace{(M \cdot E_1 \vee \overline{M} \cdot \overline{Q_0^n})}_{\overline{K_1^n}} \cdot Q_1^n$$

$$\Rightarrow J_1^n = M \cdot E_1 \vee \overline{M} \cdot Q_0^n$$

$$K_1^n = \overline{(M \cdot E_1 \vee \overline{M} \cdot \overline{Q_0^n})} = \overline{M \cdot E_1} \cdot \overline{\overline{M} \cdot \overline{Q_0^n}} = (\overline{M} \vee \overline{E_1}) \cdot (M \vee Q_0^n)$$

f) Lösungsfunktionen J_1^n und K_1^n für die Realisierung mittels JK-FF über KV-Diagramm:

Die Bestimmung der Spalten für J_1^n und K_1^n in der Übergangstabelle aus Punkt b) geschieht mittels der rechten Übergangstabelle aus Beispiel 1.3, Punkt b). Danach erfolgt, wie bereits in Beispiel 1.3 gezeigt, eine Minimierung von J_1^n und K_1^n über KV-Diagramme. Wie man aber aus der Übergangstabelle sieht, sind J_1^n und K_1^n für M = 0 von E_1^n und E_0^n unabhängig und man kann J_1^n und K_1^n hier direkt anhand der Übergangstabelle minimieren:

$J_1^n = Q_0^n$ für M = 0

$K_1^n = Q_0^n$ für M = 0

J_1^n und K_1^n sind für M = 1 von Q_0^n unabhängig $\Rightarrow$

J_1^n (für M = 1):

	Q_1^n			
E_1^n	X	X	1	1
	X	X	0	0
		E_0^n		

$J_1^n = E_1^n$ für M = 1

K_1^n (für M = 1):

[3] Das Symbol ® bezeichnet den boolschen Antivalenz Operator (EXOR-Verknüpfung)

[4] Ändert nichts am Ergebnis, da $(Q_1^n \vee \overline{Q_1^n})$ immer gleich 1 ist und ist daher erlaubt

E_1^n \ Q_1^n	0	0	X	X
	1	1	X	X
		E_0^n		

$K_1^n = \overline{E_1^n}$ für M = 1

Aus der Verknüpfung der Lösungen für M = 0 und M = 1 ergibt sich somit

$$J_1^n = M \cdot E_1^n \vee \overline{M} \cdot Q_0^n$$
$$K_1^n = M \cdot \overline{E_1^n} \vee \overline{M} \cdot Q_0^n$$

was auch mit den Lösungsgleichungen aus Punkt e) übereinstimmt (wer's nicht glaubt, daß die Lösungen für K_1^n aus Punkt e und f ident sind, mache eine Probe, indem er die Übergangstabelle für beide Fälle bestimmt).

Anmerkung: Prinzipiell ist es egal, ob man die Lösungsfunktionen eines JK-Flip-Flop mittels Koeffizientenvergleich oder mittels KV-Diagrammen bestimmt. Hat man jedoch bereits eine Gleichung für Q_i^{n+1} vorliegen (so wie es in diesem Beispiel der Fall ist, siehe Punkt d), ist eine Bestimmung mittels Koeffizientenvergleich zumeist weniger zeitaufwendig als die Bestimmung mittels KV-Diagramm.

f) Charakteristische Gleichung des RS-FF:

$Q^{n+1} = S^n \vee \overline{R^n} \cdot Q^n$ Nebenbedingung: $S^n \cdot R^n = 0$

Q^n	Q^{n+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

g) Lösungsfunktionen R_1^n und S_1^n für die Realisierung mittels RS-FF:

Die Bestimmung der Spalten für R_1^n und S_1^n in der Übergangstabelle aus Punkt b) geschieht mittels der Übergangstabelle aus Punkt g). Danach erfolgt eine Minimierung von R_1^n und S_1^n über KV-Diagramme. Die Vorgangsweise ist also im Prinzip dieselbe wie bei JK-Flip-Flops.

Wie man in diesem Beispiel aus der Übergangstabelle sieht, sind R_1^n und S_1^n für M = 0 von E_1^n und E_0^n unabhängig und man kann R_1^n und S_1^n hier direkt anhand der Übergangstabelle minimieren:

$S_1^n = \overline{Q_1^n} \cdot Q_0^n$ für M = 0
$R_1^n = Q_1^n \cdot Q_0^n$ für M = 0

R_1^n und S_1^n sind für M = 1 von Q_0^n unabhängig $\Rightarrow$

S_1^n (für M = 1):

E_1^n \ Q_1^n				
	X	X	1	1
	0	0	0	0
		E_0^n		

$S_1^n = E_1^n$ für M = 1

R_1^n (für M = 1):

E_1^n \ Q_1^n				
	0	0	0	0
	1	1	X	X
		E_0^n		

$R_1^n = \overline{E_1^n}$ für M = 1

Aus der Verknüpfung der Lösungen für M = 0 und M = 1 ergibt sich somit

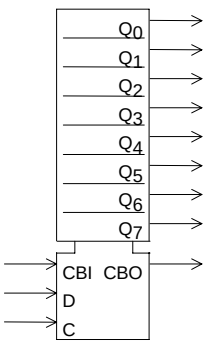
$$S_1^n = M \cdot E_1^n \vee \overline{M} \cdot \overline{Q_1^n} \cdot Q_0^n$$
$$R_1^n = M \cdot \overline{E_1^n} \vee \overline{M} \cdot Q_1^n \cdot Q_0^n$$

h) Wegen der Nebenbedingung zur charakteristischen Gleichung ist für das RS-Flip-Flop kein Koeffizientenvergleich möglich.

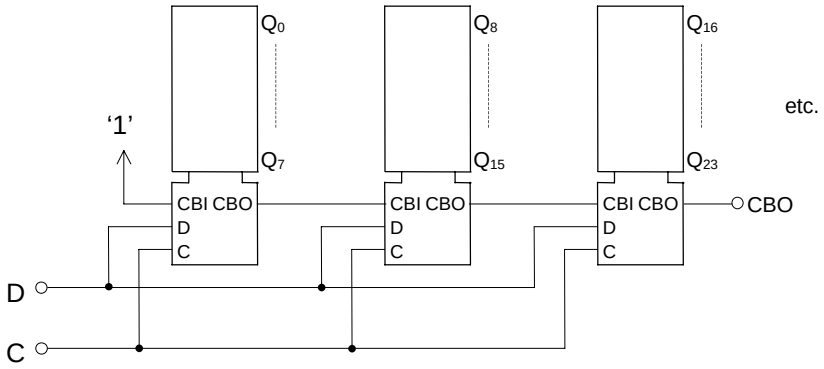
i) Wie aus der Zustandsübergangstabelle zu sehen ist, hängt das Ausgangssignal C des Automaten vom Eingangssignal M ab. Damit handelt es sich bei diesem Automaten um einen Mealy-Automaten.

2.2. Up/Down Counter

a) Blockschaltbild des Zählers:



b) Durch CBI und CBO ist eine Kaskadierung mehrerer Zähler möglich:



c) Man entwickle die Schaltung für eine Stelle des Zählers durch Betrachtung der Ausgänge des Zählers beim Vorwärtszählen und Rückwärtszählen:

Vorwärtszählen:

...	Q_3^n	Q_2^n	Q_1^n	Q_0^n	...	Q_3^{n+1}	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}
.	0	0	0	0	.	0	0	0	1
.	0	0	0	1	.	0	0	1	0
.	0	0	1	0	.	0	0	1	1
.	0	0	1	1	.	0	1	0	0
.	0	1	0	0	.	0	1	0	1
.	0	1	0	1	.	0	1	1	0
.	0	1	1	0	.	0	1	1	1
.	0	1	1	1	.	1	0	0	0
.	.	.	.	.	.	.	.	.	.

Aus obiger Tabelle erkennt man:

Q_0^{n+1} ändert sich mit jedem Takt
Q_1^{n+1} ändert sich immer dann, wenn $Q_0^n = 1$
Q_2^{n+1} ändert sich immer dann, wenn $Q_0^n \wedge Q_1^n = 1$
Q_3^{n+1} ändert sich immer dann, wenn $Q_0^n \wedge Q_1^n \wedge Q_2^n = 1$
... etc.

Der Zustand einer Stelle ändert sich beim Vorwärtszählen genau dann, wenn alle vorhergehenden Stellen des Zählers 1 sind

Rückwärtszählen:

...	Q_3^n	Q_2^n	Q_1^n	Q_0^n	...	Q_3^{n+1}	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}
.	1	0	0	0	.	0	1	1	1
.	0	1	1	1	.	0	1	1	0
.	0	1	1	0	.	0	1	0	1
.	0	1	0	1	.	0	1	0	0
.	0	1	0	0	.	0	0	1	1
.	0	0	1	1	.	0	0	1	0
.	0	0	1	0	.	0	0	0	1
.	0	0	0	1	.	0	0	0	0
.	.	.	.	.	.	.	.	.	.

Aus obiger Tabelle erkennt man:

Q_0^{n+1} ändert sich mit jedem Takt

Q_1^{n+1} ändert sich immer dann, wenn $\overline{Q_0^n} = 1$

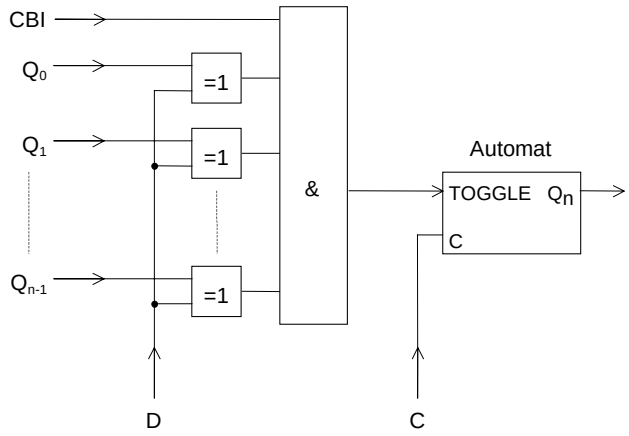
Q_2^{n+1} ändert sich immer dann, wenn $\overline{Q_0^n} \wedge \overline{Q_1^n} = 1$

Q_3^{n+1} ändert sich immer dann, wenn $\overline{Q_0^n} \wedge \overline{Q_1^n} \wedge \overline{Q_2^n} = 1$

... etc.

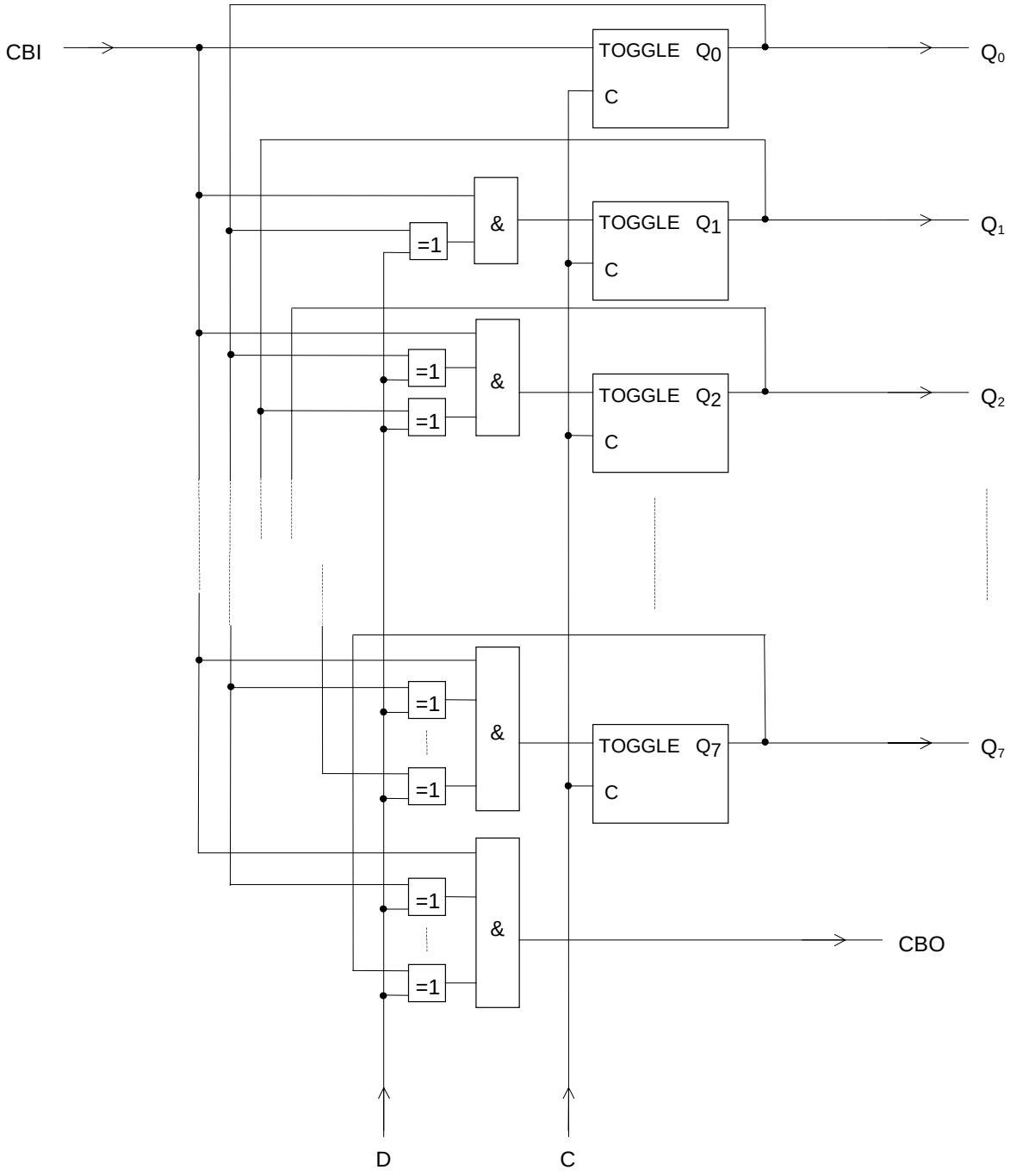
Der Zustand einer Stelle ändert sich beim Rückwärtszählen genau dann, wenn alle vorhergehenden Stellen des Zählers 0 sind.

Daraus ergibt sich die Schaltung für die n-te Stelle Q_n des Zählers:

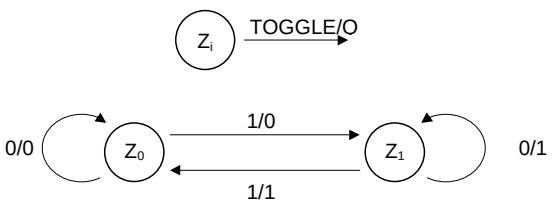


Der Automat ändert den Zustand an seinem Ausgang Q_n wenn der Eingang TOGGLE = 1 ist und behält bei TOGGLE = 0 den Zustand von Q_n bei. Über die EXOR-Gatter können beim Rückwärtszählen mit D = 1 die Eingänge Q_0, Q_1, Q_2, ... Q_{n-1} invertiert werden. Damit die Zählstufe nur bei CBI = 1 weiterzählt, wird CBI in die UND-Verknüpfung miteinbezogen.

Blockschaltbild des gesamten Zählers:



d) Zustandsgraph und Übergangstabelle des Automaten einer Zählstelle:



Ist TOGGLE = 1 ändert sich der Zustand der Stelle im nächsten Takt. Ist TOGGLE = 0 bleibt der Zustand der Stelle im nächsten Takt gleich.

TOGGLE	Q^n	Q^{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

Die Zustände wurden einfach binär durchcodiert.

e) Bestimmung der Lösungsgleichungen des Automaten einer Zählstelle für die Realisierung mittels D-FFs, JK-FFs und RS-FFs aus der Übergangstabelle:

TOGGLE	Q^n	Q^{n+1}	J^n	K^n	S^n	R^n
0	0	0	0	X	0	X
0	1	1	X	0	X	0
1	0	1	1	X	1	0
1	1	0	X	1	0	1

Realisierung mittels D-FFs:

$D^n = Q^{n+1} = \text{TOGGLE} \oplus Q^n$

Realisierung mittels JK-FFs:

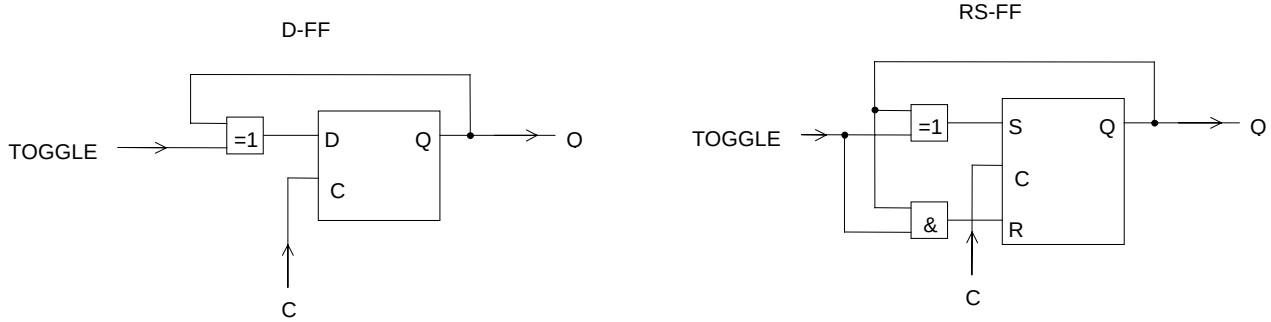
$J^n = \text{TOGGLE}$
$K^n = \text{TOGGLE}$

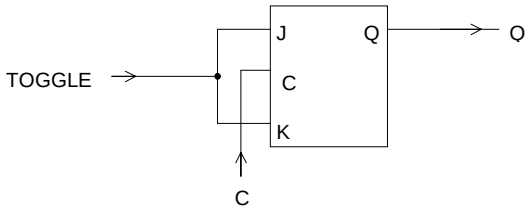
Realisierung mittels RS-FFs:

$S^n = \text{TOGGLE} \oplus Q^n$
$R^n = \text{TOGGLE} \wedge Q^n$

Der Automat einer Zählstelle alleine stellt einen Moore-Automaten dar. Nur der gesamte Zähler stellt infolge der direkten Abhängigkeit des Ausgangs CBO vom Eingang D einen Mealy-Automaten dar.

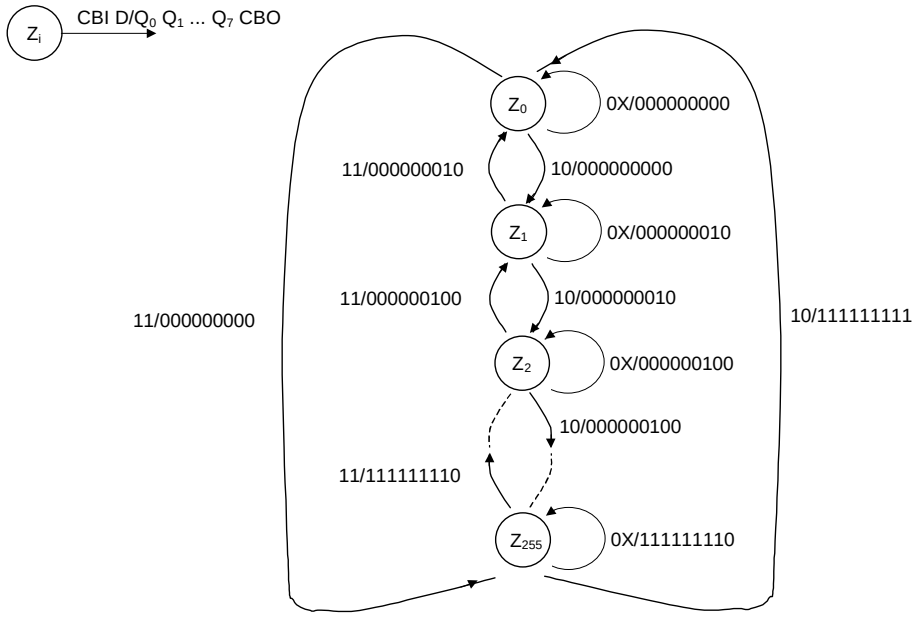
f) Schaltbilder des Automaten einer Zählstelle für die Realisierung mittels D-FFs, JK-FFs und RS-FFs:





Die Realisierung mittels JK-FFs ist für diesen Automaten am geeignetsten, da man außer dem Flip-Flop keinerlei zusätzliche Gatter benötigt.

g) Will man den gesamten 8-Bit Zähler als einen einzigen großen Automaten realisieren, sieht dessen Zustandsgraph folgendermaßen aus:



Der Automat besitzt 256 Zustände, wobei jeder Zustand einem 8-Bit-Zählerstand entspricht. Man kann sich denken, daß die Entwicklung der Schaltung des Automaten über Übergangstabellen und KV-Diagramme von Hand aus nicht mehr machbar ist, während eine Realisierung der Aufgabenstellung nach Punkt c)-f) bloß einer Entwicklung eines Automaten aus 2 Zuständen bedarf. Wie man also sieht, sollte man bei der Entwicklung eines Automaten immer vorher untersuchen, ob man diesen nicht in kleinere Automaten bzw. Schaltungseinheiten aufspalten kann, deren Lösung vielleicht weit einfacher ist.

2.3. Vereinfachung von Zustandsgraphen

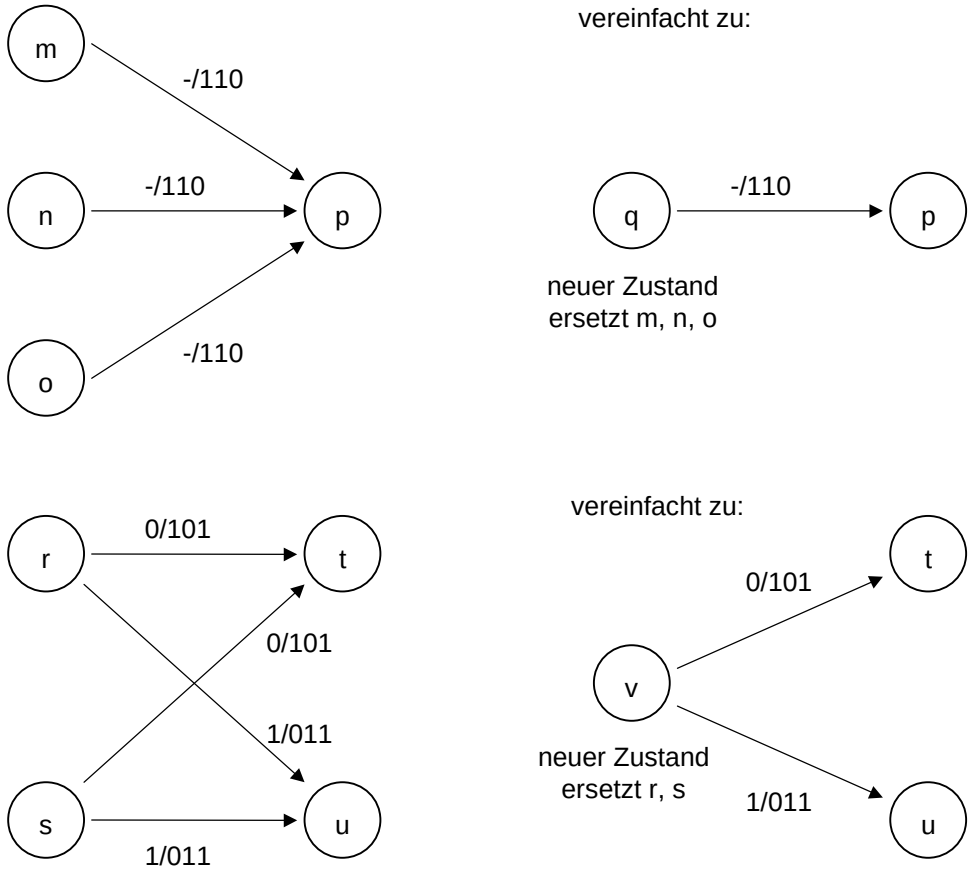
a) Zustandsliste des Automaten:

Z^n	x	r	s	t	Z^{n+1}
a	0	1	0	1	d
	1	0	1	1	b
b	-	1	1	0	c
c	-	1	0	1	a
d	-	0	1	1	e
e	0	1	1	0	h
	1	1	1	0	f
f	-	1	1	0	c
g	-	0	1	1	e
h	0	1	0	1	g
	1	0	1	1	i
i	-	1	1	0	c

b) Regel zur Elimination redundanter Zustände:

Alle Zustände die dieselbe Anzahl an abgehenden Kanten aufweisen, deren abgehende Kanten dieselbe Beschriftung tragen (sowohl gleiche Eingangssignale als auch gleiche Ausgangssignale) und die in den selben Zustand münden, können zu einem Zustand zusammengefaßt werden.

Allgemeine Beispiele:



c) Vereinfachung der Zustandsliste:

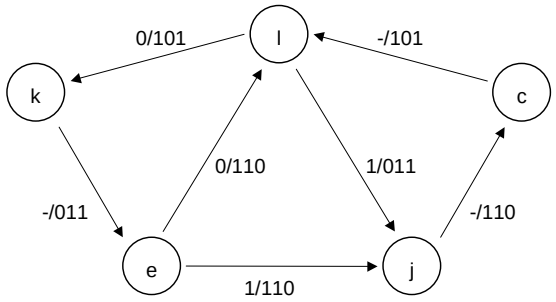
1. Schritt:
b, f, i ⇒ j
d, g ⇒ k

Z^n	x	r	s	t	Z^{n+1}
a	0	1	0	1	k
	1	0	1	1	j
c	-	1	0	1	a
e	0	1	1	0	h
	1	1	1	0	j
h	0	1	0	1	k
	1	0	1	1	j
j	-	1	1	0	c
k	-	0	1	1	e

2. Schritt:
a, h ⇒ l

Z^n	x	r	s	t	Z^{n+1}
c	-	1	0	1	l
e	0	1	1	0	l
	1	1	1	0	j
j	-	1	1	0	c
k	-	0	1	1	e
l	0	1	0	1	k
	1	0	1	1	j

Vereinfachter Zustandsgraph:



d) Bestimmung der Gleichungen für r, s und t:

Der Automat besitzt 5 Zustände. Mit 3 Flip-Flops lassen sich 8 Zustände codieren ⇒ Es werden 3 Flip-Flops zur Realisierung des Automaten benötigt.

Übergangstabelle mit gewählter Zustandscodierung:

x	Z^n	Q_2^n	Q_1^n	Q_0^n	r^n	s^n	t^n
X	c	1	0	0	1	0	1
0	e	1	1	0	1	1	0
1					1	1	0
X	j	1	1	1	1	1	0
X	k	0	1	0	0	1	1
0	l	0	0	0	1	0	1
1					0	1	1

$$r^n = Q_2^n \vee \bar{x} \cdot \overline{Q_2^n} \cdot \overline{Q_1^n} \cdot \overline{Q_0^n}$$
$$s^n = Q_1^n \vee x \cdot \overline{Q_2^n} \cdot \overline{Q_1^n} \cdot \overline{Q_0^n}$$
$$t^n = \overline{Q_1^n} \vee \overline{Q_2^n}$$

jeweils nur 2 disjunktive Terme

e) Die in Punkt d) aus der Übergangstabelle bestimmten Gleichungen sind bereits Lösungen für r, s und t. Mithilfe von KV-Diagrammen wird nun überprüft, ob sich die Gleichungen noch weiter vereinfachen lassen:

r^n:

x \ Q_2^n				
	1	X	X	0
	1	1	X	0
	1	1	X	0
	1	X	X	1
		Q_0^n		

(Q_1^n)

$$r^n = Q_2^n \vee \overline{x} \cdot \overline{Q_1^n}$$

s^n:

x \ Q_2^n				
	0	X	X	1
	1	1	X	1
	1	1	X	1
	0	X	X	0
		Q_0^n		

(Q_1^n)

$$s^n = Q_1^n \vee x \cdot \overline{Q_2^n}$$

Wie aus der Übergangstabelle ersichtlich ist, ist t^n von x unabhängig $\Rightarrow$

t^n:

Q_2^n				
0	0	X	1	Q_1^n
1	X	X	1	
	Q_0^n			

$$t^n = \overline{Q_1^n} \vee \overline{Q_2^n}$$

f) Bestimmung der Lösungsgleichungen bei Realisierung mittels D-FFs:

x	Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}
X	1	0	0	0	0	0
0	1	1	0	0	0	0
1	1	1	0	1	1	1
X	1	1	1	1	0	0
X	0	1	0	1	1	0
0	0	0	0	0	1	0
1	0	0	0	1	1	1

Q_2^{n+1}:

x \ Q_2^n	Q_0^n = 0	Q_0^n = X	Q_0^n = X	Q_0^n = 1
Q_1^n = 0	0	X	X	1
Q_1^n = 1	1	1	X	1
Q_1^n = 0	0	1	X	1
Q_1^n = 0	0	X	X	0

$$D_2^n = Q_2^{n+1} = Q_0^n \vee x \cdot Q_1^n \vee x \cdot \overline{Q_2^n} \vee \overline{Q_2^n} \cdot Q_1^n$$

Q_1^{n+1}:

x \ Q_2^n	Q_0^n = 0	Q_0^n = X	Q_0^n = X	Q_0^n = 1
Q_1^n = 0	0	X	X	1
Q_1^n = 1	1	0	X	1
Q_1^n = 0	0	0	X	1
Q_1^n = 0	0	X	X	1

$$D_1^n = Q_1^{n+1} = \overline{Q_2^n} \vee x \cdot Q_1^n \cdot \overline{Q_0^n}$$

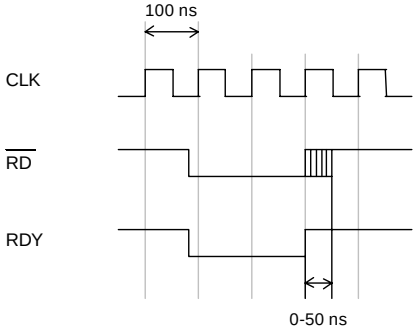
Q_0^{n+1}:

x \ Q_2^n	Q_0^n = 0	Q_0^n = X	Q_0^n = X	Q_0^n = 1
Q_1^n = 0	0	X	X	1
Q_1^n = 1	1	0	X	0
Q_1^n = 0	0	0	X	0
Q_1^n = 0	0	X	X	0

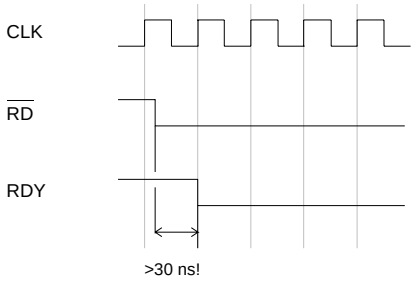
$$D_0^n = Q_0^{n+1} = x \cdot \overline{Q_2^n} \cdot \overline{Q_1^n} \vee x \cdot Q_2^n \cdot Q_1^n \cdot \overline{Q_0^n}$$

2.4. Ready-Logik

a) Timing eines Lesezyklus:

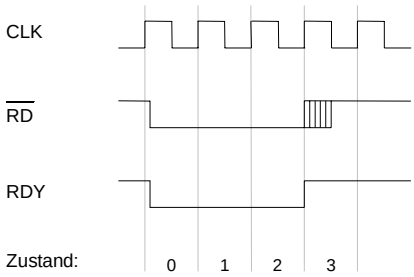


b) Warum für die RDY-Logik kein Moore-Automat verwendet werden kann, begründet sich durch folgendes Beispiel:

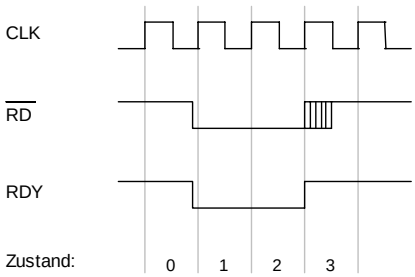


In vorliegendem Fall kommt die fallende Flanke des /RD-Signals kurz nach der steigenden Flanke des Taktsignals CLK. Nachdem sich Ausgangssignale von Moore-Automaten aber nur zu Beginn jedes Zustands ändern können, würde das RDY-Signal vom Automaten hier zu spät, nämlich im Extremfall erst 100 ns nach der fallenden Flanke von /RD auf 0 gesetzt werden.

c) Die beiden Extremfälle für den Zeitpunkt der fallenden Flanke von /RD sind kurz nach bzw. kurz vor der steigenden Taktflanke CLK. Daraus ergeben sich die beiden Zeiten t_{RDYmin} und t_{RDYmax}.



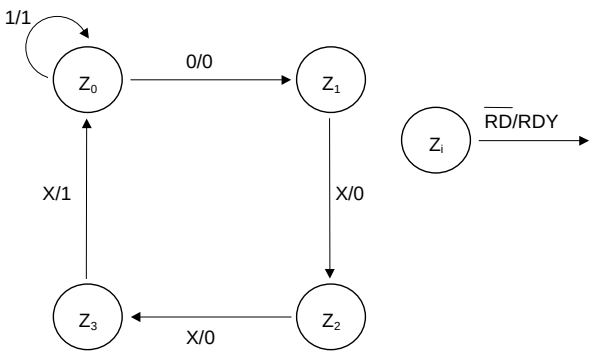
$t_{RDY_{max}} = 300ns$



$t_{RDY_{min}} = 200ns$ wie gefordert

$\Rightarrow$ RDY muß min. 2 Zustände lang (in Z_1, Z_2) auf 0 gehalten werden

d) Zustandsgraph der RDY-Logik:



e) Zustandsübergangstabelle der RDY-Logik:

$\overline{RD}$	Z_i^n	Q_1^n	Q_0^n	Z_i^{n+1}	Q_1^{n+1}	Q_0^{n+1}	RDY
0	0	0	0	1	0	1	0
1				0	0	0	1
X	1	0	1	2	1	1	0
X	2	1	1	3	1	0	0
X	3	1	0	0	0	0	1

Die Codierung der Zustände erfolgt so, daß sich beim Übergang von einem Zustand auf den nächsten jeweils nur eine der beiden Zustandsvariable Q_1^n und Q_0^n ändert. Würde man die Zustände einfach binär codieren, ändern sich beispielsweise beim Übergang von Zustand 1 auf Zustand 2 beide Zustandsvariable von $Q_1^n = 0$ und $Q_0^n = 1$ auf $Q_1^n = 1$ und $Q_0^n = 0$. Nachdem sich bei realen Flip-Flops die Ausgänge aber nicht exakt zum gleichen Zeitpunkt ändern, kann es dadurch beim Zustandswechsel kurzzeitig zu beliebigen Kombinationen von Q_1^n und Q_0^n kommen und am Ausgang RDY kann ein Hazard entstehen.

f) Lösungsgleichungen bei Realisierung mittels D-Flip-Flops:

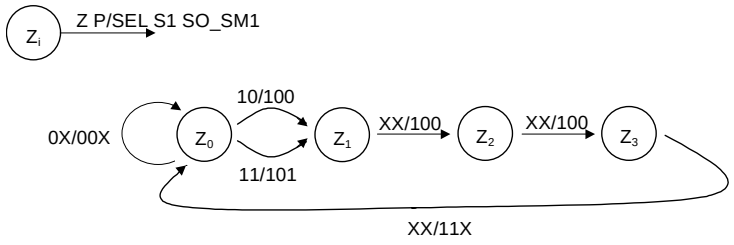
$$D_1^n = Q_1^{n+1} = Q_0^n$$

$$D_0^n = Q_0^{n+1} = \overline{\overline{RD}} \cdot \overline{Q_1^n} \vee \overline{Q_1^n} \cdot Q_0^n$$

$$RDY = \overline{RD} \cdot \overline{Q_0^n} \vee Q_1^n \cdot \overline{Q_0^n}$$

2.5. HDB3-Encoder

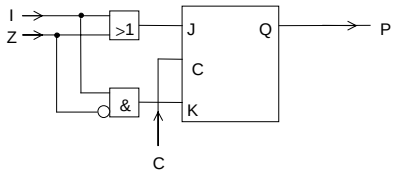
a) Zustandsgraph des Automaten SM1:



Schaltungserklärung: Das Schieberegister SR liefert an seinen Ausgängen SR2, SR1 und SR0 das um 1, 2 bzw. um 3 Takte verzögerte Eingangssignal I. Solange kein Block von 4 aufeinanderfolgenden Nullen auftritt, ist der Ausgang Z des NOR-Gatters logisch 0. Der Automat SM1 befindet sich im Zustand Z0 und gibt am Ausgang SEL logisch 0 aus. Damit wird S0 über den Multiplexer MUX mit SR0 verbunden, an S0 liegt also das um 3 Takte verzögerte Eingangssignal I an. S1 ist hingegen im Zustand Z0 des Automaten 0. Unter Zuhilfenahme der Codierungstabelle in der Angabe ist ersichtlich, daß eine 0 und eine 1 am Eingang I weiterhin auf 0 und 1 am Signalvektor {S0, S1} abgebildet wird, solange an I kein Block von 4 aufeinanderfolgenden Nullen auftritt.

Bei 4 aufeinanderfolgenden Nullen an I geht der Ausgang Z der NOR-Gatters auf 1. Der Automat SM1 gibt nun im Zustand Z0 an SEL eine 1 aus, wodurch S0 über den Multiplexer MUX mit dem Ausgang S0_SM1 des Automaten verbunden wird. In Abhängigkeit vom Zustand von P, also je nachdem ob seit dem letzten Viererblock von Nullen eine gerade oder ungerade Anzahl von Einsen aufgetreten ist, wird in Z0 an S0_SM1 eine 0 oder eine 1 ausgegeben. Danach durchläuft der Automat SM1 die Zustände Z1 bis Z3 wo an {S0, S1} die Folge 00D ausgegeben wird. Damit wird im Falle des Auftretens von 4 aufeinanderfolgenden Nullen an I, am Signal {S0, S1} eines der beiden Muster 000D oder 100D ausgegeben.
Der Automat SM1 muß als Mealy-Automat realisiert werden, da der Ausgang SEL des Automaten im Falle Z = 1 sofort umgeschaltet werden muß. Wird SM1 als Moore-Automat realisiert, kann SEL erst mit einem Takt Verzögerung nach Z = 1 auf 1 geschaltet werden, was aber bereits zu spät ist, da hier schon die erste 0 des Viererblocks über {S0, S1} an den Automaten SM3 weitergeleitet wurde.

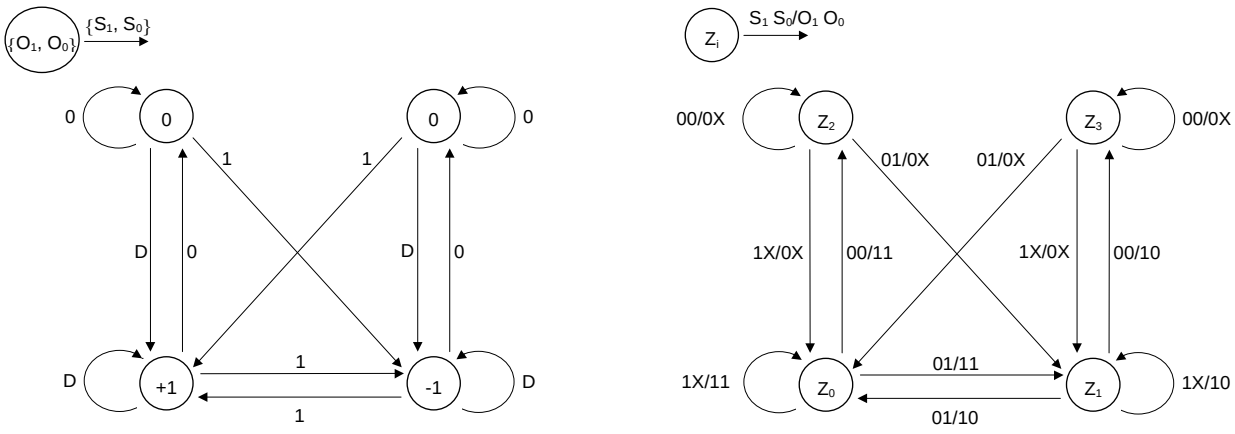
b) Schaltung des Automaten SM2 bei Realisierung über ein JK-Flip-Flop:



Ist Z = 1 wird J = 1 und K = 0 gesetzt. Damit wird also beim Auftreten von 4 aufeinanderfolgenden Nullen der Ausgang P auf 1 gesetzt. Wenn Z = 0 ist, wird hingegen der Eingang I direkt mit den Eingängen J und K des Flip-Flops verbunden. Ist I = 0 behält das JK-Flip-Flop seinen alten Zustand bei. Beim Auftreten einer 1 an I wird hingegen das Flip-Flop getoggelt und der neue Zustand von P entspricht dem invertierten vorherigen Zustand. Damit zeigt P an, ob seit dem letzten Viererblock von Nullen eine gerade (P = 1) oder eine ungerade (P = 0) Anzahl an Einsen an I aufgetreten ist.

Wichtig ist, daß P erst einen Takt nach Z = 1 auf 1 gesetzt wird, da der Automat SM1 in dem Takt, in dem Z = 1 gesetzt wird, noch die "alte" Information an P benötigt (studieren Sie nochmals die Funktionsbeschreibung von SM1 unter Punkt a). Ob P hingegen sofort oder einen Takt verzögert über I = 1 getoggelt wird, ist für SM1 unerheblich. Sinnvollerweise, also unter dem Aspekt der Gattereinsparung, wird der Automat SM2, so wie hier, als Moore-Automat realisiert.
Der Automat kann übrigens selbstverständlich auch über ein D-FF oder ein RS-Flip-Flop realisiert werden.

c) Zustandsgraph des Automaten SM3:



Zur besseren Verdeutlichung wurde der Zustandsgraph zweimal gezeichnet. Im linken Bild wurden die Werte D, 0 und 1 für {S0, S1} sowie 0, -1 und +1 für {O0, O1} direkt in den Graphen eingezeichnet, während im rechten Bild die binär codierten Werte für {S0, S1} und {O0, O1} angegeben sind.

d) Der zeitliche Versatz zwischen I und {S0, S1} beträgt 3 Takte (zur Erklärung siehe Punkt a). Der Versatz zwischen {S0, S1} und {O0, O1} beträgt, nachdem SM3 ja als Moore-Automat realisiert wurde, eine Taktperiode. Insgesamt ergibt sich damit ein zeitlicher Versatz des uncodierten NRZ-Datenstroms an I zum codierten HDB3-Datenstrom an {O0, O1} von 4 Takten.

e) Da keine direkte Abhängigkeit des Ausgangs {O0, O1} vom Eingang I besteht (d.h. keine reine Kombinatorik zwischen I und {O0, O1}) stellt die Gesamtschaltung des HDB3-Encoders einen Moore-Automaten dar.

f) Die Anzahl der Flip-Flops des gesamten HDB3-Encoders berechnet sich wie folgt: 3 FFs für Schieberegister SR, 2 FFs für SM1, 1 FF für SM2, 2 FFs für SM3. Der gesamte Encoder benötigt damit 8 Flip-Flops. Außerdem benötigt man zur Realisierung des Encoders ein 3-zu-1-NOR-Gatter, einen 2-zu-1-Multiplexer, sowie die Gatter, welche zum Aufbau der Automaten SM1-SM3 benötigt werden.

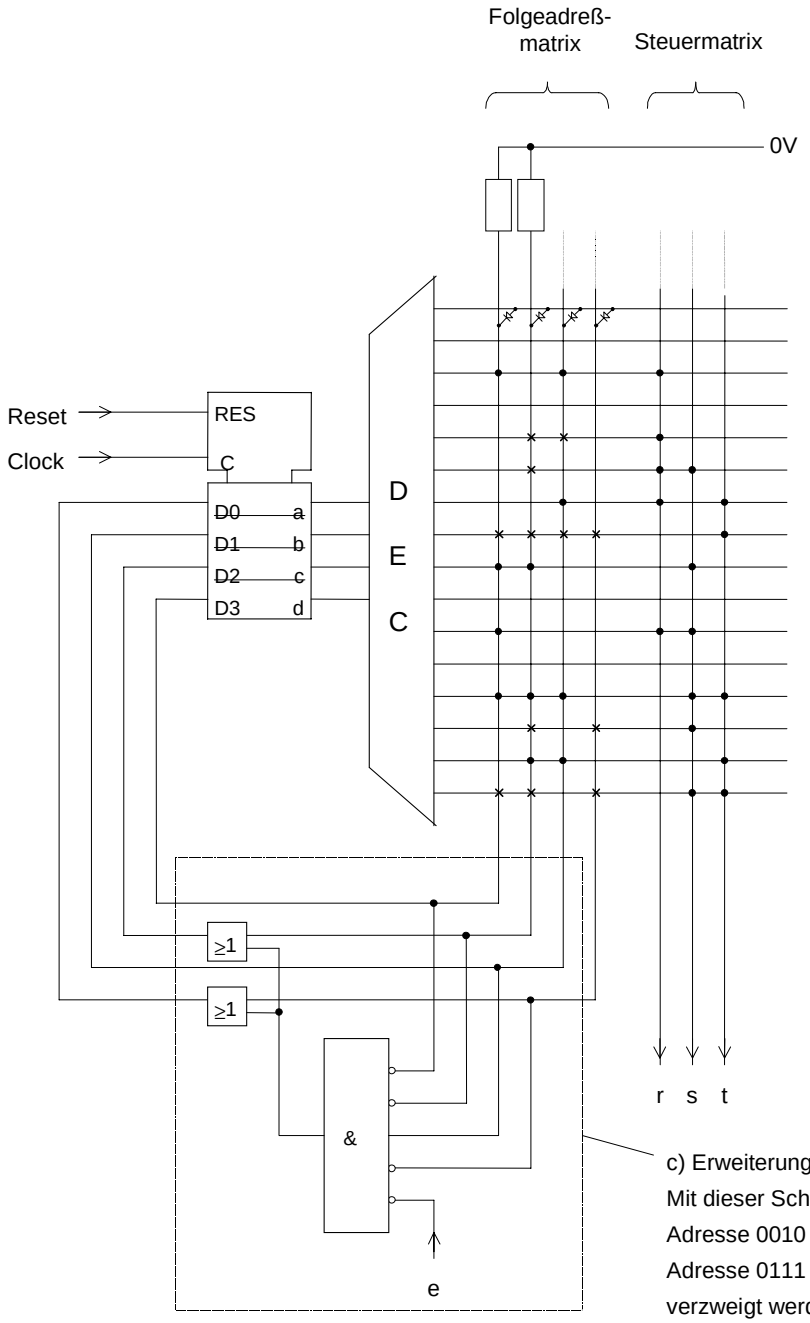
Wenn man nun den gesamten HDB3-Encoder als einen einzigen großen Automaten realisieren würde, wäre es durchaus möglich, daß man sich dadurch einige Gatter ersparen könnte. Allgemein ist jedoch zu sagen, daß eine Minimierung von Bauelementen sicher nicht der einzige Gesichtspunkt beim Entwurf von komplexen Schaltwerken ist, auch wenn die hier vorgestellten Beispiele vielleicht den Eindruck erwecken, daß beim Entwurf der Schaltungen immer um jedes einzelne Flip-Flop und jedes einzelne Gatter "gegeizt" wird.

Ein wesentlicher, wenn nicht sogar der kostenintensivste Faktor bei der Erzeugung und Entwicklung von Waren und Produkten ist die Arbeitszeit. Eine bis ins letzte Detail optimierte Schaltung erspart vielleicht Bauteilkosten, bedeutet aber möglicherweise wiederum größerer Entwicklungskosten, wenn für die Entwicklung der optimierten und deshalb komplizierteren Schaltung mehr Entwicklungszeit benötigt wird. Sollte ein Re-Design der Schaltung nötig sein, vervielfältigt sich dieses "Mehr" an Kosten. Das Modularisieren und Zerlegen in Subblöcke, wie es etwa in diesem Beispiel gezeigt wurde, macht eine Schaltung hingegen zumeist durchschaubarer. Auch wird das Simulieren und Testen zumeist vereinfacht, da nicht eine komplizierte Gesamtschaltung, sondern einzelne, voneinander unabhängige Blöcke getrennt simuliert und getestet werden können. Simulations- und Testzeiten können verkürzt werden, was wiederum geringere Entwicklungs- und Produktionskosten bedingt.

3. Programmschaltwerke

3.1 Programmschaltwerk für Drehstromsignal

a), b), c), d) Schaltbild des Programmschaltwerks:



Adresse					
d	c	b	a	Z^n	für e
0	0	0	0		
0	0	0	1		
0	0	1	0	1	1
0	0	1	1		
0	1	0	0	1	0
0	1	0	1	2	0
0	1	1	0	0	0 und 1
0	1	1	1	5	0
1	0	0	0	3	1
1	0	0	1		
1	0	1	0	2	1
1	0	1	1		
1	1	0	0	4	1
1	1	0	1	3	0
1	1	1	0	5	1
1	1	1	1	4	0

Leere Zeilen brauchen nicht programmiert zu werden!

c) Erweiterung
Mit dieser Schaltung kann statt auf Adresse 0010 (=Zustand 1 für e=1) auf Adresse 0111 (=Zustand 5 für e=0) verzweigt werden

Funktion der ROM-Matrix:

Der Dekoder dekodiert die an seinen Eingängen anliegende Adresse und setzt, entsprechend der jeweiligen Adresse, die entsprechende Zeile auf 1 (alle anderen Zeilen sind 0). Die an den 7

Kreuzungspunkten einer Zeile vorhandenen bzw. nicht vorhandenen Dioden bestimmen den Zustand der 7 Datenbits (Folgeadresse + Steuersignale) einer Adresse. Ist an einem Kreuzungspunkt die entsprechende Diode vorhanden, liegt diese Datenleitung auf 1. Fehlt die Diode, wird die Leitung durch einen Pull-Down-Widerstand auf 0 gezogen.

$\Rightarrow$ Diode vorhanden (bzw. Kreuz oder Punkt vorhanden): Datenbit = 1
Diode nicht vorhanden (bzw. Kreuz oder Punkt nicht vorhanden): Datenbit = 0

Zu Punkt c):Vier der fünf Eingänge des UND-Gatters stammen, wie im Schaltbild zu sehen ist, vom Ausgang der Folgeadreßmatrix. Theoretisch könnte man allerdings auch den Eingang bzw. die Zeile der Folgeadreßmatrix mit der Adresse 0111 (was dem Zustand 0 entspricht) mit dem Eingang e verknüpfen. Man würde statt eines 5-fach-UND-Gatters dann nur ein 2-fach-UND-Gatter benötigen. Da man bei ROMs (und die Folgeadreßmatrix bildet ja zusammen mit dem Dekoder ein ROM) jedoch keinen Zugriff auf die Eingänge der Folgeadreßmatrix hat, verbietet sich diese Lösung in der Praxis.

e) Die Bauteile des Programmschaltwerks nach a)-d) lassen sich durch folgende Grundbauelemente eine Mikrocontrollers darstellen:

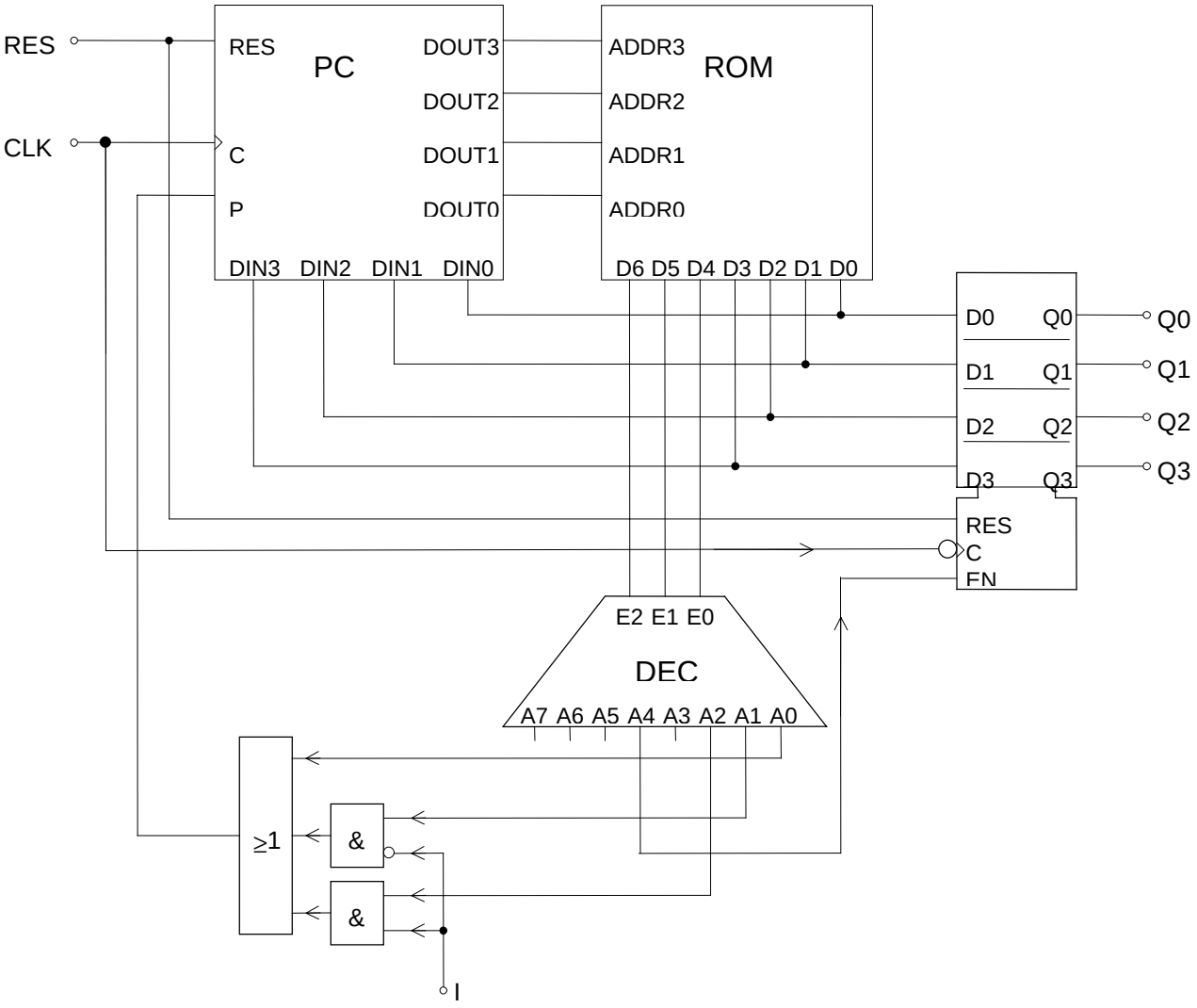
Folgeadreßmatrix, Steuermatrix und Dekoder entsprechen dem ROM eines Mikrocontrollers. Die Eingänge des Dekoders stellen dabei die Adreßeingänge des ROMs dar, während die Ausgänge von Folgeadreßmatrix- und Steuermatrix die Datenausgänge des ROMs darstellen.

Das Folgeadreßregister stellt den Program Counter eines Mikrocontrollers dar. Es handelt sich hier allerdings um einen Spezialfall eines Program Counters, welcher bei jedem Befehl mit einer Sprungadresse geladen wird, d.h. immer die Exekution eines Sprungbefehls durchführt.

Ein Befehlsdekoder existiert hier nicht. Das Programmschaltwerk kann nämlich nur genau einen Befehl abarbeiten. Der Mnemonic dieses Befehls könnte etwa *OUTNJP Data, Addr* also "Out and Jump" lauten. Durch diesen Befehl wird ein 3 Bit breiter Datenwert an r,s und t ausgegeben und danach an eine angegebene 4-Bit-Adresse gesprungen.

3.2. Anzeigeeinheit für seriell übertragene Daten

a) Schaltung des Microcontrollers:



Zuordnung der Befehlscodes:

D6	D5	D4	Befehl
0	0	0	`JP Adr`
0	0	1	`JPZ Adr`
0	1	0	`JPNZ Adr`
0	1	1	`NOP`
1	0	0	`OUT Data`

Realisierung der Sprungbefehle:

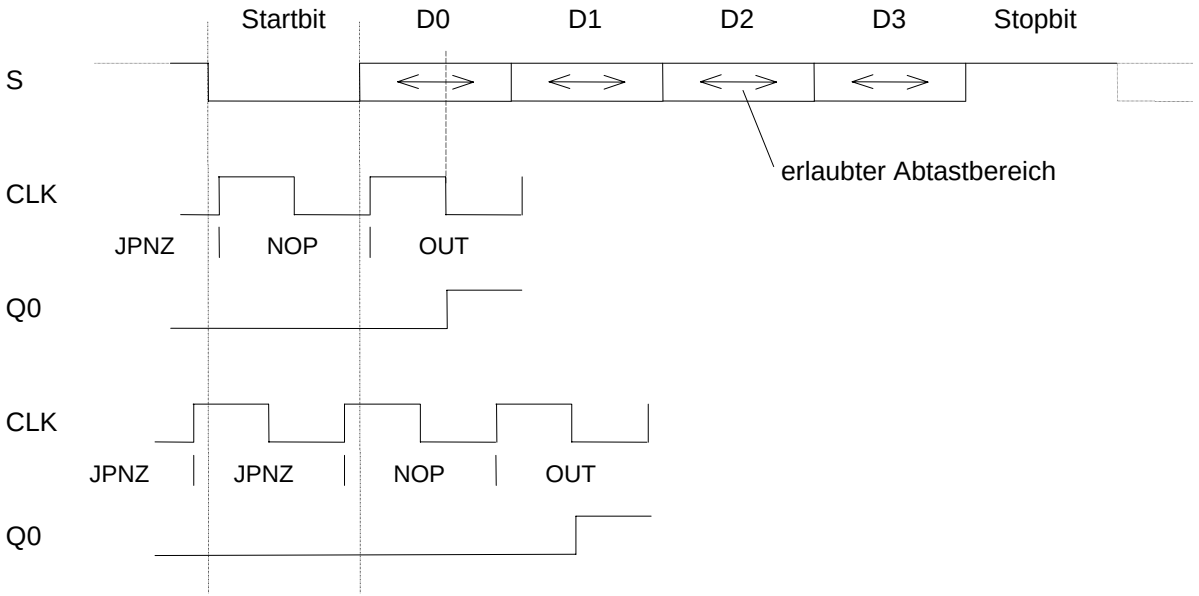
Um einen Sprung auszuführen, muß der Eingang P des Program Counters 1 sein. Durch das ODER-Gatter vor diesem Eingang werden drei Sprungmöglichkeiten zugelassen. Die Leitung A0 wird direkt an das ODER-Gatter geführt. Der Befehlscode, der diesen Ausgang aktiviert (000), bewirkt

also einen unbedingten Sprung. Die Leitungen A1 (aktiviert beim Befehlscode 001) und A2 (Befehlscode = 010) sind jeweils noch mit dem Eingangsport I UND-verknüpft. Diese beiden Sprünge werden also durch den Zustand an I entschieden.

b) Der Mikrocontroller liest mit der positiven Flanke von CLK den Zustand am Eingang I ein. Mit der negativen Flanke von CLK wird der OUT-Befehl durchgeführt. Da die Flip-Flops des Schieberegisters SR über den Takteingang CS durch das Ausgangsport des Controllers getaktet werden, erfolgt das Weiterschieben des Schieberegisters mit der negativen Taktflanke von CLK.

Im folgenden wird eine Taktfrequenz f_{CLK} gewählt und danach untersucht, ob das Einlesen des seriellen Bitstroms vom Schieberegister im erlaubten Bereich (25% bis 75% innerhalb der Bitdauer) liegt.

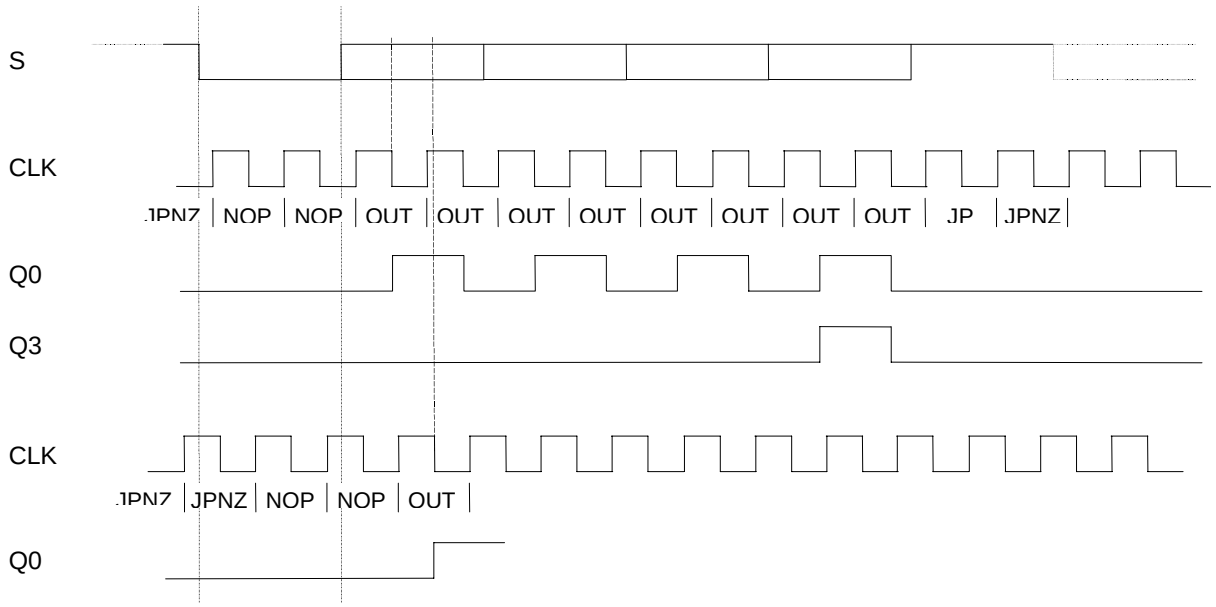
1. Versuch: $f_{CLK} = f_s$



In obigem Timing sind nun die beiden Extremfälle für die Phasenlage des Taktes CLK zum dazu asynchronen Signal S dargestellt. Im ersten Fall kommt die positive Taktflanke CLK kurz nach der fallenden Flanke an S, welche das Startbit einleitet. Der Microcontroller erkennt das Startbit somit gleich am Anfang der Bitdauer. Nach einem NOP-Befehl wird über einen folgenden OUT-Befehl das Schieberegister SR mit der fallenden Flanke von CLK getaktet. Das Bit D0 des Nibbles wird somit genau zu Bitmitte eingelesen.

Im zweiten Fall kommt die positive Taktflanke CLK kurz vor der fallenden Flanke an S, welche das Startbit einleitet. Damit könnte aber bei gleichem Programmablauf das Bit D0 nicht mehr eingelesen werden. Wie aus obigem Timing ersichtlich ist, wird durch den OUT-Befehl Bit D1 des Nibbles eingelesen. Eine Lösung der Aufgabe unter der Bedingung: $f_{CLK} = f_s$ ist also prinzipiell unmöglich.

2. Versuch: $f_{CLK} = 2 * f_s$



In obigem Timing sind nun wieder die beiden Extremfälle für die Phasenlage des Taktes CLK zum Signal S dargestellt. Wie zu sehen ist, erfolgt das Takten des Schieberegisters SR im ersten Fall bei 25% der Bitdauer, während im zweiten Fall das Schieberegister bei 75% der Bitdauer getaktet wird. Das Zeitpunkt des Einlesen des seriellen Bitstroms von S liegt damit gerade im erlaubten Toleranzbereich.

c) Assemblerprogramm für den Microcontroller:

Adresse				Opcode			Daten				Assemblerbefehl		
A3	A2	A1	A0	D6	D5	D4	D3	D2	D1	D0			
0	0	0	0	0	1	0	0	0	0	0	JPNZ	0000	Auf Startbit warten
0	0	0	1	0	1	1	X	X	X	X	NOP		1 Takt warten
0	0	1	0	0	1	1	X	X	X	X	NOP		1 Takt warten
0	0	1	1	1	0	0	0	X	X	1	OUT	0XX1	SR takten
0	1	0	0	1	0	0	0	X	X	0	OUT	0XX0	
0	1	0	1	1	0	0	0	X	X	1	OUT	0XX1	SR takten
0	1	1	0	1	0	0	0	X	X	0	OUT	0XX0	
0	1	1	1	1	0	0	0	X	X	1	OUT	0XX1	SR takten
1	0	0	0	1	0	0	0	X	X	0	OUT	0XX0	
1	0	0	1	1	0	0	1	X	X	1	OUT	1XX1	SR takten, Nibble über L ausgeben
1	0	1	0	1	0	0	0	X	X	0	OUT	0XX0	
1	0	1	1	0	0	0	0	0	0	0	JP	0000	Zurück in Ausgangszustand

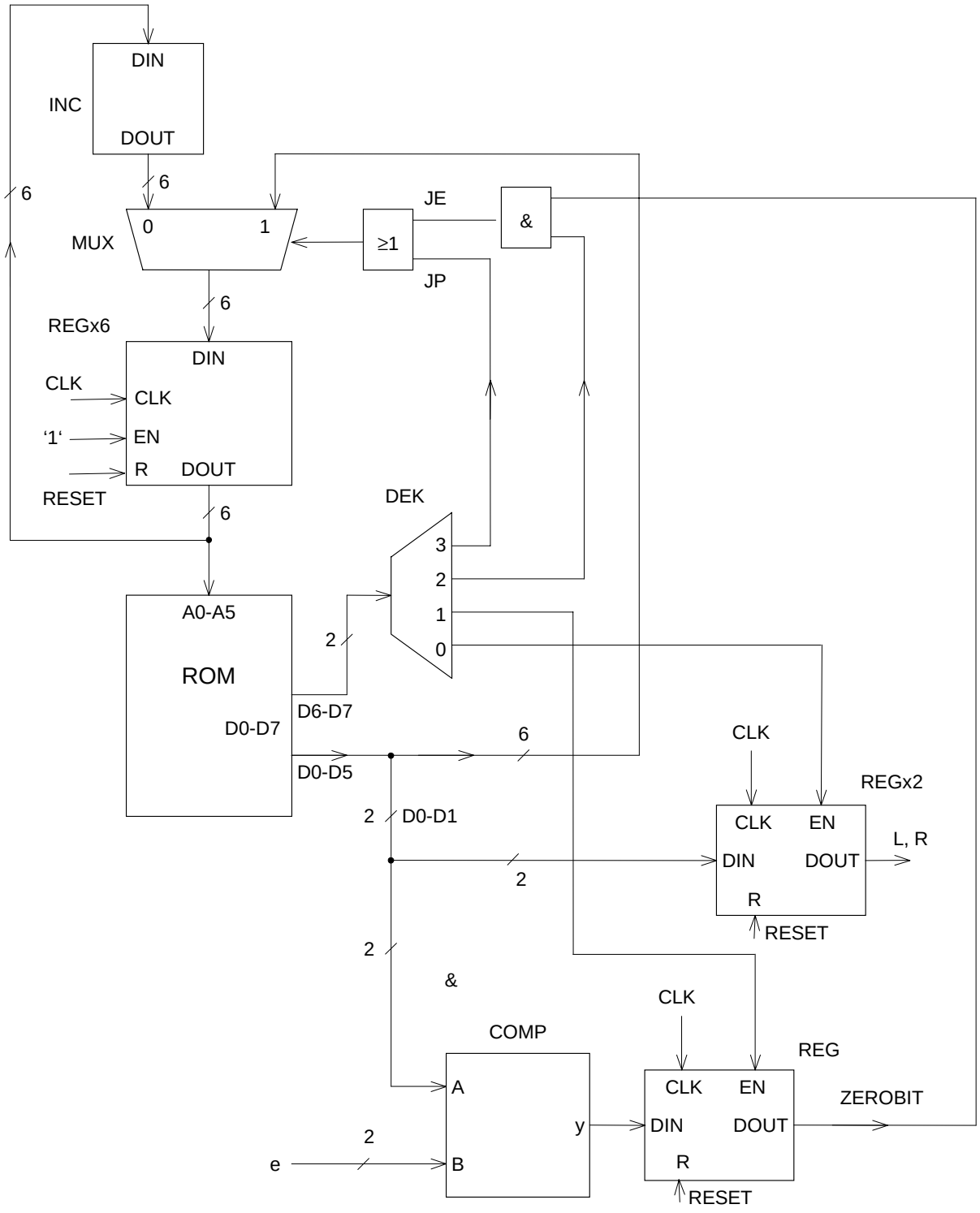
Die Adressen 12-15 im ROM müssen nicht programmiert werden.

3.3. Programmschaltwerk zur Drehrichtungsbestimmung

a) Zeitlich Abfolge des Eingangssignals e in Abhängigkeit von der Drehrichtung:

Linkslauf (Gegenuhrzeigersinn): 0, 1, 3, 2, 0, 1, 3 ...
Rechtslauf (Uhrzeigersinn): 0, 2, 3, 1, 0, 2, 3 ...

b) Schaltung des Programmschaltwerks:



Die Resetleitung setzt alle 3 Register bei einem Reset auf 0. Damit sind die Ausgänge L und R nach Inbetriebnahme 0 und der Programmzähler wird auf Adresse 0 gesetzt.

c) Die absolute Reset-Adresse des Schaltwerks ist 0, da die verwendeten Register und damit auch der Programmzähler bei einem Reset auf 0 gesetzt werden.

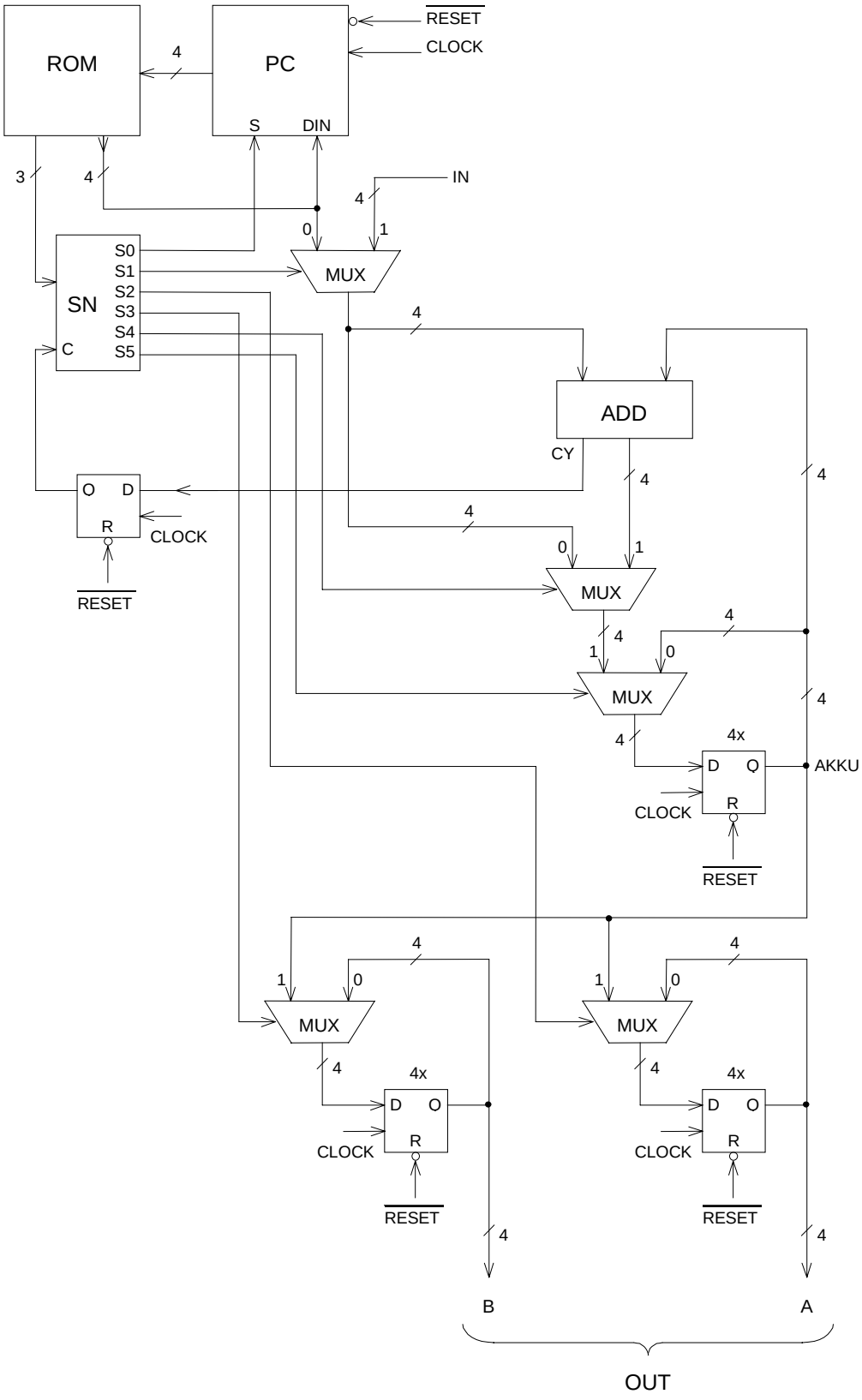
d) Assemblerprogramm für die Drehrichtungsbestimmung:

```
INIT:   CP 1          ; Initialisierung
        JE E_1
        CP 2
        JE E_2
        CP 3
        JE E_3
E_0:    CP 1          ; Eingang war 0
        JE L_01
        CP 2
        JE R_02
        JP E_0
L_01:   SET 2         ; Linkslauf 0 → 1
        JP E_1
R_02:   SET 1         ; Rechtslauf 0 → 2
        JP E_2
E_1:    CP 3          ; Eingang war 1
        JE L_13
        CP 0
        JE R_10
        JP E_1
L_13:   SET 2         ; Linkslauf 1 → 3
        JP E_3
R_10:   SET 1         ; Rechtslauf 1 → 0
        JP E_0
E_2:    .
        .
        .
        . etc.
```

L	R	Wert von D1-D0	
1	0	2	Linkslauf
0	1	1	Rechtslauf

3.4. Programmschaltwerk mit Addierer

a) Schaltbild des Programmschaltwerks:



b) Wahrheitstabelle für das Schaltnetz SN:

C	D7	D6	D5	S0	S1	S2	S3	S4	S5	Befehl
X	0	0	0	0	1	0	0	0	1	IN
X	0	0	1	0	0	0	0	0	1	CONST Zahl
X	0	1	0	0	1	0	0	1	1	ADDI
X	0	1	1	0	0	0	0	1	1	ADDC Zahl
X	1	0	0	0	X	1	0	X	0	OUTa
X	1	0	1	0	X	0	1	X	0	OUTb
X	1	1	0	1	X	0	0	X	0	JMP Zahl
1	1	1	1	1	X	0	0	X	0	JC Zahl
0	1	1	1	0	X	0	0	X	0	JC Zahl

c) Programm:

```
RESET:     IN             ; Zahl vom Eingangsport einlesen
           ADDC 5         ; 5 dazuaddieren
           JC CARRYSET    ; Carrybit gesetzt?
           OUTa           ; Summe auf Port OUTa ausgeben
           CONST 0
           OUTb           ; Kein Übertrag => OUTb=0 ausgeben
END1:      JMP END1       ; Endlosschleife
CARRYSET:  OUTa           ; Summe auf Port OUTa ausgeben
           CONST 1
           OUTb           ; Übertrag => OUTb=1 ausgeben
END2:      JMP END2       ; Endlosschleife
```

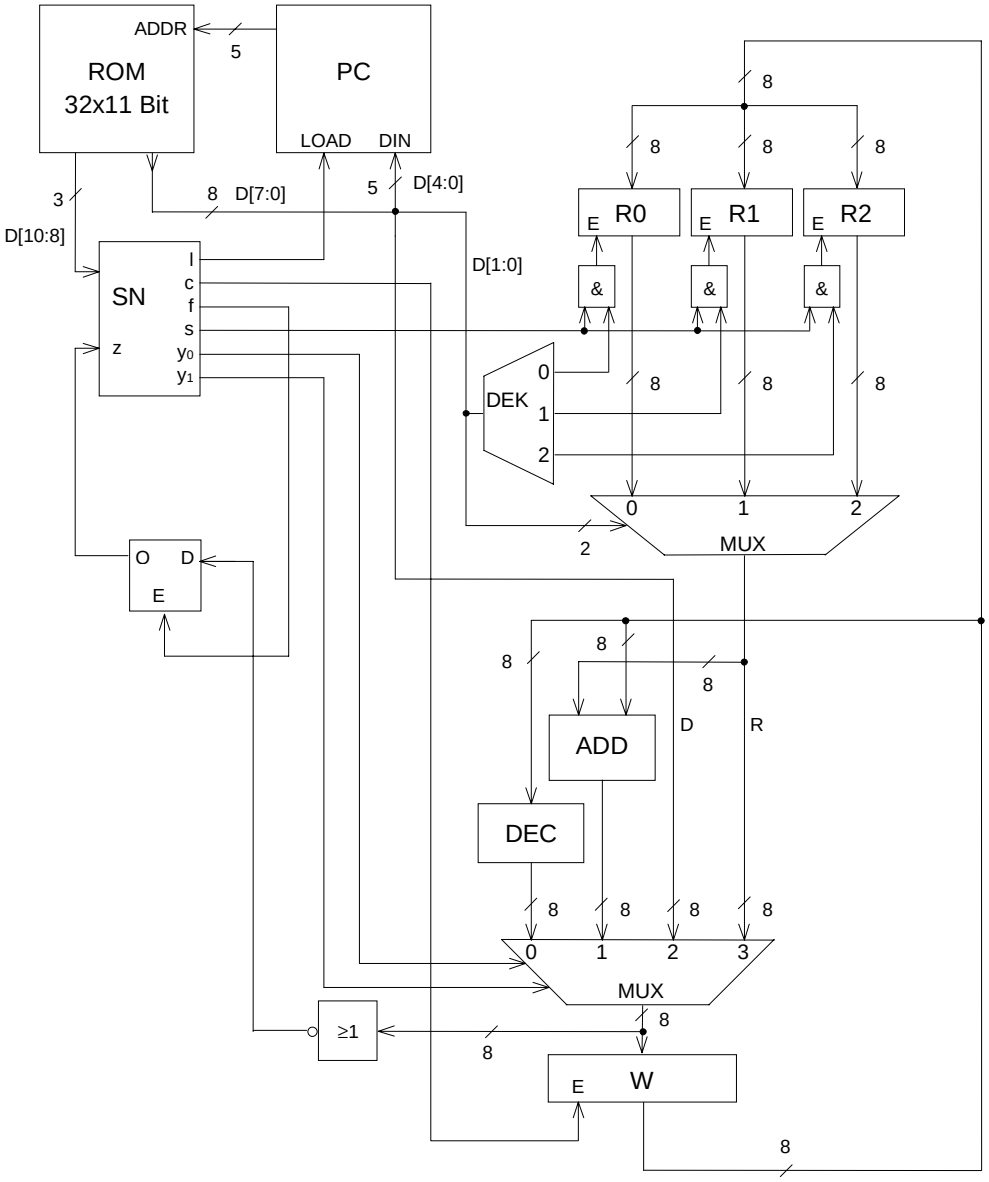
3.5. Multiplizierer

a) Assemblerprogramm für die Multiplikation zweier Integerzahlen m und n:

```
           SET 0
           STR R2         ; Register R2 (Produkt) mit 0 initialisieren
           SET n
           JNZ NOTZERO
           JMP END        ; Falls Multiplikand n=0 => Programmende
NOTZERO:   STR R1         ; Register R1 mit n initialisieren
           SET m
           JNZ LOOP
           JMP END        ; Falls Multiplikator m=0 => Programmende
LOOP:      STR R0         ; Register R0 aktualisieren
           LD R2
           ADD R1
           STR R2         ; R2=R2+R1
           LD R0
           DEC            ; R0=R0-1
```

```
        JNZ LOOP      ; Falls R0 ungleich 0 => Sprung zu LOOP
END:    JMP END       ; Endlosschleife
```

b) Schaltbild des Schaltwerks:



Das globale Taktsignal C und das globale Resetsignal wurden der Einfachheit halber ausnahmsweise nicht in die obige Schaltung eingezeichnet. Von C werden sowohl der Program Counter PC als auch das Register für das Zero-Flag und die Register W, R0, R1 und R2 getaktet. Alle Register und der Program Counter können über das globale Resetsignal resetiert werden.

Auswahl des in W zu ladenden Bytes:

y_1	y_0	MUX
0	0	DEC
0	1	ADD
1	0	D
1	1	R_i

c) Wahrheitstabelle für das Schaltnetz SN des Steuerwerks:

z	D_{10}	D_9	D_8	l	c	f	s	y_1	y_0	Befehl
X	0	0	0	0	1	1	0	1	0	SET const
X	0	0	1	0	1	0	0	1	1	LD R_i
X	0	1	0	0	0	0	1	X	X	STR R_i
X	0	1	1	0	1	0	0	0	1	ADD R_i
X	1	0	0	0	1	1	0	0	0	DEC
0	1	0	1	1	0	0	0	X	X	JNZ Adr
1	1	0	1	0	0	0	0	X	X	JNZ Adr
X	1	1	0	1	0	0	0	X	X	JMP Adr

4. Hardwarebeschreibungssprachen

4.1. Beschreibung von Multiplexern in VHDL

a) VHDL-Identifier haben eine Länge von 1 bis 32 Zeichen können sich aus folgendem Zeichensatz zusammensetzen[5]:

{a, b, c, ... z, A, B, C, ... Z, 0, 1, ... 9, _}

Das erste Zeichen eines Identifiers muß ein Buchstabe sein. Es dürfen keine zwei Underscores direkt aufeinander folgen und keine VHDL-Schlüsselworte verwendet werden. Es gibt keine Unterscheidung zwischen Klein- und Großbuchstaben. Aber Vorsicht! Manche Synthesewerkzeuge führen leider wieder eine Unterscheidung der Groß- und Kleinschreibung der Signalnamen ein und können dann sehr unangenehme Effekte verursachen, wenn man im VHDL Sourcecode unterschiedliche Schreibweisen verwendet hat. Es ist daher am sichersten, auch VHDL so zu behandeln, als wäre die Bezeichner Case-sensitve! Am Institut werden Signalnamen explizit mit einem s_ Präfix gekennzeichnet, wobei Ein- und Ausgangsports einen _i, _o, oder _b Suffix erhalten, je nachdem in Welche Richtung sie gerichtet sind. Im Rahmen dieses Skriptums werden alle Signalnamen GROSS geschrieben, damit sie im Code leichter sichtbar sind.

b) Kommentare werden durch die beiden Zeichen -- eingeleitet und gehen bis zum Zeilenende.

a) Eine Schaltung wird in VHDL durch ein Entity-/Architecture-Paar beschrieben. Eine Entity beschreibt dabei im Wesentlichen die Ein- und Ausgänge zur „Außenwelt“, die sogenannten *Ports*, während die Architecture die Funktion der Schaltung beschreibt. Sinn dieser Zweiteilung ist es, mehrere Implementierungsmöglichkeiten der Schaltung zur Auswahl stellen zu können. Während die Ein- und Ausgänge der Schaltung, also die Entity, immer gleich bleibt, können für eine Entity mehrere Architectures, also mehrere Implementierungen, existieren. Bei der Auswahl einer Implementierung wird dann einer Entity genau eine Architecture zugeordnet.

Dazu muß angemerkt werden, daß die im Labor verwendete Entwicklungsumgebung MAX+plus II nicht die Auswahl mehrerer Architectures unterstützt. Außerdem müssen sich bei der Verwendung von MAX+plus II ein zusammengehöriges Entity-/Architecture-Paar immer in ein- und demselben File befinden.

b) Entity für einen 2-zu-1-Multiplexer:

```
--------------------------------------------------------------------
-- ICT                          Institute of Computer Technology
--                              University of Technology, Vienna
--                              All Rights Reserved
--
-- project          : KOSCH
-- designer(s)      : Peter Roessler
-- version          : 1.0
-- file name        : mux_entity.vhd
-- title            : 2 to 1 multiplexer, entity
-- tools            : MAX+plus II
-- ref. library     : IEEE
```

[5] Nach dem Standard VHDL 87

```
-- ref. package    : std_logic_1164
-- date            : 1999-04-01
-------------------------------------------------------------
-- Purpose:
--
-- Multiplexes either IN0 or IN1 to OUTMUX depending on SEL
-------------------------------------------------------------

-- Include library IEEE and use package std_logic_1164:
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX is
   -- Declaration of all ports of the multiplexer:
   port(
            IN0    : in std_logic;  -- Input 0 of multiplexer
            IN1    : in std_logic;  -- Input 1 of multiplexer
            SEL    : in std_logic;  -- Select signal of multiplexer
            OUTMUX : out std_logic  -- Output of multiplexer
       );
end MUX;
```

Eine Entity sollte immer mit einem Header, also mit einem Kommentarfeld beginnen, in dem diverse Daten wie Projektname, Autor und Datum sowie eine kurze funktionelle Beschreibung angegeben werden.

Über die Schlüsselwort *library* wird im obigen Beispiel die Bibliothek *IEEE* eingebunden und über das nachfolgende Schlüsselwort *use* wird das Package *std_logic_1164* zugänglich gemacht. In diesem Package ist die Definition des Datentyps *std_logic* enthalten. Dieser Datentyp wird zur Beschreibung von elektrischen, digitalen Signalen verwendet. Nachdem keine Schaltung ohne Signale auskommt, enthalten praktisch alle Entities, aber auch die Architectures, in denen jegliche Art von Hardware beschrieben wird die obigen beiden Statements.

Die Beschreibung der eigentlichen Entity beginnt mit dem *entity*-Statement, wobei für die Entity ein frei wählbaren Namen angegeben wird.

Über das *port*-Statement werden alle Ein- und Ausgänge der Schaltung deklariert. Nachdem es im Rahmen dieser Lehrveranstaltung um die Beschreibung von digitaler Hardware geht, sind hier alle Ports immer vom Typ *std_logic* oder *std_logic_vector*. Außer dem Signalnamen muß noch für jedes Port die Signalrichtung, also *in* oder *out* angegeben werden[6].

Das Schlüsselwort *end*, gefolgt vom Namen der Entity, beendet die Entity-Deklaration.

c) Architecture für einen 2-zu-1-Multiplexer unter Verwendung einer Signalzuweisung in Verbindung mit einem *when*-Statements:

```
-------------------------------------------------------------
-- ICT                        Institute of Computer Technology
--                            University of Technology, Vienna
--                            All Rights Reserved
--
-- project         : KOSCH
-- designer(s)     : Peter Roessler
-- version         : 1.0
```

[6] Anmerkung: Es ist selbstverständlich auch möglich bidirektionale Signale zu verwenden, doch wird hier davon, insbesondere im Rahmen der in Kapitel 5 noch zu behandelnden Design-Rules, kein Gebrauch gemacht.

```
-- file name       : mux_architecture.vhd
-- title           : 2 to 1 multiplexer, architecture
-- tools           : MAX+plus II
-- ref. library    : IEEE
-- ref. package    : std_logic_1164
-- date            : 1999-04-01
-------------------------------------------------------------------
-- Purpose:
--
-- Multiplexes either IN0 or IN1 to OUTMUX depending on SEL
-------------------------------------------------------------------

-- Include library IEEE and use package std_logic_1164:
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_MUX of MUX is
   -- Functional behaviour of the multiplexer:
   begin
      OUTMUX <= IN0 when SEL = '0' else IN1;
   end beh_MUX;
```

Eine Architecture sollte, wie auch die Entity, mit einem Header beginnen. Üblicherweise werden bei Projekten die Entities und die Architectures in unterschiedlichen Dateien abgespeichert. VHDL lässt es aber auch zu, mehr als eine Übersetzungs-Unit in einer Datei unterzubringen. Dazu aber eine Warnung! Die library und use Statements beziehen sich bei vielen VHDL Compilern auf die folgende Entity / Architecture / Component etc. Man muss daher oft die Library und Use Statements in einer Datei mehrfach unterbringen! Das mag nicht sehr logisch sein, ist aber leider so.

Zunächst wird wieder, wie bei der Entity, das Package *std_logic_1164* eingebunden. Danach wird die Beschreibung der eigentlichen Architecture über das *architecture*-Statement eingeleitet. Hier geben Sie den Namen der Architecture und den Namen der dazugehörigen Entity an. Als Name der Architecture wurde *beh_MUX* gewählt, um zu verdeutlichen, daß es sich hier um eine Beschreibung in Form des *Verhaltens* eines Multiplexers handelt (*beh* steht dabei als Abkürzung *behavioural*). Eine alternative Beschreibungsform wäre eine *strukturelle* Beschreibung der Schaltung, wobei man als Name der Architecture in diesem Fall zum Beispiel *struct_MUX* wählen könnte.

Die eigentliche Funktion der Schaltung befindet sich zwischen den Schlüsselworten *begin* und *end* (wobei dem Schlüsselwort *end* noch der Name der Architecture folgt). Im Falle des Multiplexers besteht die Funktionsbeschreibung nur aus einer einzigen Zeile. Der Ausdruck <= stellt in VHDL den Zuweisungsoperator für Signale dar. Dem Signal OUTMUX wird also der Zustand von IN0 zugewiesen, wenn SEL = 0 ist. Andernfalls wird OUTMUX der Zustand von IN1 zugewiesen.

d) Beschreibung der boolschen Operationen Negation, UND, ODER und Exklusiv-ODER in VHDL:

VHDL-Schlüsselwort	Operation
not	Negation
and	UND
or	ODER
xor	Exklusiv-ODER

e) Architecture für einen 2-zu-1-Multiplexer unter Verwendung von boolschen Operatoren:

```
-- Include library IEEE and use package std_logic_1164:
library IEEE;
```

```
use IEEE.std_logic_1164.all;

architecture beh_boolean_MUX of MUX is
   -- Functional behaviour of the multiplexer:
   begin
       OUTMUX <= (not(SEL) and IN0) or (SEL and IN1);
   end beh_boolean_MUX;
```

Die Funktionsbeschreibung besteht hier, wie die Funktionsbeschreibung mithilfe des *when*-Statements, ebenfalls nur aus einer Zeile. Wie zu sehen ist, wurde der Multiplexer diesmal über eine boolsche Funktion beschrieben. An der Funktion der Schaltung ändert sich jedoch nichts. Im vorliegenden Fall wird man eher die Bescheibung nach Punkt e) bevorzugen, da diese auf den ersten Blick zu erkennen gibt, daß es sich hier um einen Multiplexer handelt. Sie können im übrigen davon ausgehen, daß die in diesem Kapitel vorgestellen Beschreibungen gebräuchliche bzw. sinnvolle Beschreibungsformen sind, wenngleich es für jede Schaltung zumeist noch eine Reihe weiterer möglicher Beschreibungsformen in VHDL gibt.

f) Entity für einen 2x4-Bit-zu-1x4-Bit Multiplexer:

```
-- Include library IEEE and use package std_logic_1164:
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX is
   -- Declaration of all ports of the multiplexer:
   port(
             IN0    : in std_logic_vector(3 downto 0); -- 4 bit input 0
             IN1    : in std_logic_vector(3 downto 0); -- 4 bit input 1
             SEL    : in std_logic;                    -- Select signal of MUX
             OUTMUX : out std_logic_vector(3 downto 0) -- 4 bit output of MUX
        );
end MUX;
```

Zur Deklaration von Signalvektoren bzw. Bussen verwendet man den Datentyp *std_logic_vector*, welcher ebenso wie *std_logic* im Package *std_logic_1164* der Library *IEEE* deklariert ist. Die Breite des Vektors wird in der nachfolgenden Klammer angegeben.

Die Architecture für den 2x4-Bit-zu-1x4-Bit Multiplexer sieht genauso wie in Punkt e) bzw. Punkt g) aus.

4.2. Beschreibung von Flip-Flops und Registern in VHDL

a) Für low-aktive Signale ist die Endung B (als Abkürzung für *BAR*, also der Strich über dem Signal, welcher die Invertierung anzeigt) an den Signalnamen anzuhängen. Eine sinnvolle Bezeichnung für ein low-aktives Resetsignal lautet beispielsweise etwa RESETB.

Sinn dieser Richtlinie ist es (ebenso wie die Richtlinie, das in Signalnamen keine Kleinbuchstaben verwendet werden sollen) eine Einheitlichkeit des VHDL-Sourcecodes zu erzielen, was insbesondere von großer Wichtigkeit ist, wenn mehrere Personen an einem Design arbeiten (man vergleiche ähnliche Richtlinien bei der Erstellung von Software mittels Hochsprachen).

b) Bei der Zuweisungen von Signalvektoren muß beachtet werden, daß die Signale bzw. Ausdrücke auf der rechten und der linken Seite des Zuweisungsoperators die gleiche Breite besitzen. Folgenden Beispiele sind korrekte Signalzuweisungen. Dabei wird angenommen, daß die

Signalvektoren A und B eine Breite von 8 Bit, C eine Breite von 6 Bit und D eine Breite von 10 Bit besitzen.

```
A <= B;
A(7 downto 0) <= B(7 downto 0);
A(4 downto 2) <= B(6 downto 4);
A(6 downto 1) <= C;
A <= D(8 downto 1);
```

Bei der Zuweisung von Konstanten an Signale vom Typ *std_logic* werden hauptsächlich folgende Werte verwendet:

Wert	Bedeutung
'0'	Logisch 0
'1'	Logisch 1
'-'	Don't Care

Die Verwendung des Wertes für *Don't Care* wird in nachfolgenden Beispielen noch gezeigt. Für Signale vom Typ *std_logic* existiert übrigens noch eine Anzahl weiterer möglicher Werte, doch wird im Rahmen dieser Übung nicht darauf eingegangen.

Für Signale vom Typ *std_logic_vector* existieren die gleichen Werte wie die vom Typ *std_logic*, wobei hier die Werte von Double-Quotes begrenzt werden. In den folgenden Beispielen ist E vom Typ *std_logic*, während A vom Typ *std_logic_vector* mit einer Breite von 8-Bit ist.

```
E <= '1';
A <= "01010101";
A(6 downto 1) <= "111111";
A(4) <= '1';
```

c) Architecture eines D-Flip-Flops mit Namen *beh_DFF* für die zugehörige Entity *DFF*. Die Entity besitzt die Eingänge CLK, RESETB und D sowie den Ausgang Q.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_DFF of DFF is
  begin
      D_Flip_Flop:process (CLK, RESETB)
      begin
        if RESETB = '0' then Q <= '0'; -- clear FF on reset
        elsif rising_edge (CLK) then Q <= D; -- Q <= D on rising clock edge
        end if;
      end process D_Flip_Flop;
  end beh_DFF;
```

Die Architecture enthält einen Prozeß mit dem Namen *D_Flip_Flop* in dem die Funktionalität des D-Flip-Flops beschrieben wird. Der Prozeß ist notwendig, da das innerhalb des Prozesses stehende *if-*/*elsif*-Statement ein sogenanntes *Sequential Statement* ist, das nicht für sich alleine[7] stehen kann. Im Gegensatz dazu, ist die Signalzuweisungen (auch die bedingte Signalzuweisungen mittels *when*, welche bereits in Beispiel 4.1 verwendet wurde) ein sogenanntes *Concurrent Statement*, das für sich alleine stehen kann. Alle Concurrent Statements werden parallel abgearbeitet, was ja die spezifische Eigenschaft jeglicher Hardware ist. Sequential Statements werden jedoch wie die Anweisungen in

[7] Das Wort *alleine* wird hier im Sinne von „nicht innerhalb eines Prozesses" verwendet.

einer Programmiersprache sequentiell abgearbeitet. Durch die Kapselung von Sequential Statements mit Hilfe eines Prozesses, wird die Funktionalität die über die Sequential Statements beschrieben wird, zu einer Einheit, welche wiederum parallel zu anderen Concurrent Statements abgearbeitet werden kann. Der Prozeß als ganze Einheit, in diesem Fall also das D-Flip-Flop, ist also selbst ein Concurrent Statement.

Dem Schlüsselwort *process* folgt die sogenannte *Sensitivity List* eines Prozesses. Der Prozeß ist im Normalfall inaktiv und wird nur durch eine Änderung zumindest eines der in der Sensitivity List angegebenen Signale aktiviert[8]. In der Sensitivity List müssen sich demnach alle Signale befinden, aufgrund deren Zustandsänderung eine direkte Zustandsänderung von anderen Signalen des Prozesses erfolgt. Bei allen synchron aufgebauten, speichernden Elementen wie Flip-Flops, Registern etc. befinden sich immer nur das Taktsignal und das Resetsignal in der Sensitivity List, da Änderungen von Ausgangssignalen oder inneren Zuständen immer nur mit der aktiven Taktflanke oder bei einem Reset erfolgen können. Im obigen Fall befinden sich also nur die Signale RESETB und CLK in der Sensitivity List des Prozesses.

Die eigentliche Funktionalität des Prozesses befindet sich zwischen den Schlüsselworten *begin* und *end process* (wobei *end process* noch der Name der Prozesses folgt). Zunächst wird das low-aktive Resetsignal RESETB abgefragt und im Falle eines Resets durch RESETB = 0 der Ausgang Q des D-Flip-Flops sofort, also asynchron zum Taktsignal CLK, auf 0 gesetzt. Wurde kein Reset ausgelöst wird mit der steigenden Taktflanke (Funktion *rising_edge*) der Zustand des Eingangs D vom Ausgang Q übernommen. Wichtig ist, daß der Zustand von RESETB in der *if*-Bedingung vor dem Zustand von CLK abgefragt wird, da RESETB laut Angabe ja Priorität vor CLK haben soll.

d) Um das positiv flankengetriggerten D-Flip-Flop aus Punkt c) in ein negativ flankengetriggertes D-Flip-Flop umzuwandeln, ist einfach die Zeile

```
elsif rising_edge (CLK) then Q <= D; -- Q <= D on rising clock edge
```

durch

```
elsif falling_edge (CLK) then Q <= D; -- Q <= D on falling clock edge
```

zu ersetzen. Will man ein D-Flip-Flop realisieren, daß für CLK = 1 transparent ist, ersetzt man obige Zeile hingegen durch

```
elsif CLK = '1' then Q <= D; -- Q <= D if CLK = high state (latch)
```

Nachdem der Ausgang Q aber im Falle CLK = 1 jetzt direkt von D abhängig ist, muß D bei einer Realisierung eines transparenten D-Flip-Flops zur Sensitivity List hinzugefügt werden.

e) Architecture eines 8-Bit breiten synchronen Registers mit Namen *beh_REG8* für die zugehörige Entity *REG8*. Die Entity besitzt die Eingänge CLK, RESETB, LOAD, den 8-Bit breiten Eingang D sowie den 8-Bit breiten Ausgang Q.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_REG8 of REG8 is
  begin
      REG8:process (CLK, RESETB)
```

[8] Abgesehen davon wird ein Prozeß aber immer zumindest ein einziges Mal, nämlich bei der Initialisierung einmal durchlaufen.

```
    begin
      if RESETB = '0' then Q <= "00000000"; -- clear REG on reset
      elsif rising_edge (CLK) then
          if LOAD = '1' then Q <= D; -- load Q with D on LOAD synchronously
          end if;
      end if;
    end process REG8;
  end beh_REG8;
```

Die Architecture aus Punkt c) wurde lediglich um ein *if*-Statement erweitert, wodurch der Zustand des Eingangsvektors D nur in dem Fall vom Ausgangsvektor Q mit der steigenden Taktflanke übernommen wird, wenn dabei gleichzeitig LOAD logisch 1 ist. Ist LOAD logisch 0, behält das Register bei einer positiven Taktflanke den alten Zustand bei.

Im Gegensatz zu CLK und RESETB befindet sich LOAD nicht in der Sensitivity List des Prozesses, da Zustandsänderungen von LOAD erst mit der aktiven Taktflanke Zustandsänderungen an Q bewirken können. Es existiert also keine direkte Abhängigkeit von Signalen des Prozesses mit dem Signal LOAD, wie das bei synchron aufgebauten, speichernden Elementen allgemein so ist (siehe auch Punkt c). Wird das Signal LOAD ebenfalls in die Sensitivity List aufgenommen, wirkt sich dies in keinstem Maße auf die Funktionalität der Schaltung aus, doch wird die Simulationsgeschwindigkeit damit verringert. Auf diesen Aspekt wird in Kapitel 5 nochmals eingegangen.

f) Architecture eines 4-Bit-Schieberegisters mit Namen *beh_SREG4* für die zugehörige Entity *SREG4*. Die Entity besitzt die Eingänge CLK, RESET, SHIFT, LOAD, CLR, SREGIN, den 4-Bit breiten Eingang LOADIN sowie den Ausgang SREGOUT.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_SREG4 of SREG4 is

signal Q : std_logic_vector(3 downto 0); -- 4 FFs of the SREG

  begin
    SREG:process (CLK, RESET)
    begin
      if RESET = '1' then Q <= "0000"; -- clear SREG on reset
      elsif rising_edge (CLK) then
          if CLR = '1' then Q <= "0000"; -- clear SREG on CLR synchronously
          elsif LOAD = '1' then Q <= LOADIN; -- load SREG with LOADIN on LOAD
          elsif SHIFT = '1' then -- shift SREG
                Q(3) <= Q(2);
                Q(2) <= Q(1);
                Q(1) <= Q(0);
                Q(0) <= SREGIN;
          end if;
      end if;
    end process SREG;

    SREGOUT <= Q(3);
  end beh_SREG4;
```

Neu in dieser Architecture (im Vergleich zu den vorherigen Architectures dieses Skriptums) ist die Deklaration des internen Signals *Q* über das Schlüsselwort *signal*. Dieser 4-Bit-Signalvektor stellt die Ausgänge der Flip-Flops des Schieberegisters dar und kann im Gegensatz zu den Ports der Entity nicht außerhalb der Architecture verwendet werden, stellt also ein internes Signal der

Architecture dar. Im Unterschied zu Ports wird bei der Deklaration von internen Signalen keine Signalrichtung angegeben. Dies erscheint klar, denn nachdem es sich ja um interne Signale handelt, müssen diese sowohl von einer Quelle innerhalb der Architecture getrieben werden als auch in eine Senke innerhalb der Architecture münden.

Im Prozeß *SREG* wird das funktionelle Verhalten des Schieberegisters beschrieben. Höchste Priorität vor allen Aktionen hat wieder ein Reset über das Signal RESET, wodurch alle Bits des Schieberegisters asynchron auf 0 gesetzt werden. Nächste Priorität hat das Löschen des Schieberegisters über CLR = 1. Im Unterschied zu einem Reset wird hier das Schieberegister synchron, also mit der steigenden Taktflanke gelöscht. Falsch wäre es, das CLR-Signal mit dem RESET-Signal zu verknüpfen, etwa durch

```
if (RESET = '1' or CLR = '1')  then Q <= "0000"; -- clr SREG on reset or CLR
```

Dadurch würde das Schieberegister asynchron zum Takt gelöscht. Schaltungsmäßig würde obige Zeile eine ODER-Verknüpfung des Signals RESET mit CLR an den Reset-Eingängen der Flip-Flops des Schieberegisters bedeuten. Man hüte sich auch davor, als Signalname für das Löschsignal etwa RESETSREG statt CLR zu verwenden. Das führt nur zu Mißverständnissen („ ... ist RESETSREG jetzt ein asynchrones oder ein synchrones Löschsignal? “). Bezeichnungen wie RESET oder RST sollten alleine dem asynchronen Resetsignal des Designs vorbehalten sein. Für synchrone Löscheingänge von Flip-Flops, Registern, Zählern, etc. sollte man generell lieber Bezeichnungen wie CLR, SET0, etc. verwenden.

Ähnliches gilt übrigens auch für Taktsignale. Oftmals werden in Signalnamen für „sogenannte Taktsignale“ einer Schaltung, die aber in Wirklichkeit zumeist synchrone Enable-Signale sind, Bezeichnungen wie CLOCK oder CLK verwendet. Bei synchronen Designs gibt es aber genau ein Taktsignal, mit dem alle Flip-Flops des Designs getaktet werden. Bezeichnungen wie CLOCK oder CLK sollten alleine dem globalen Taktsignal des Designs vorbehalten sein.

Ist RESET = 0 und während der steigenden Taktflanke CLR = 0 und LOAD = 1, wird das Schieberegister mit dem an LOADIN anliegenden 4-Bit-Wert geladen.

Ist RESET = 0 und während der steigenden Taktflanke CLR = 0, LOAD = 0 und SHIFT = 1, wird das Schieberegister um eine Stelle von der niederwertigeren zur höherwertigeren Stelle geschoben. Die niederwertigste Stelle wird mit dem Wert des Eingangs SREGIN geladen.

Sind alle 3 Steuereingänge CLR, LOAD und SHIFT des Schieberegisters logisch 0, behält das Register seinen Zustand bei einer positiven Taktflanke bei. Man beachte nochmals die Reihenfolge der Auswertungen im *if-/elsif*-Statement des obigen Prozesses, welche die Priorität der Steuereingänge CLR, LOAD und SHIFT bestimmt.

Am Ende der Architecture, bereits wieder außerhalb des Prozesses SREG, wird noch der Ausgang SREGOUT des Schieberegisters mit dem Ausgang des höchstwertigsten Flip-Flops Q(3) verbunden.

g) Synchrone Schaltungen haben in VHDL immer dasselbe Aussehen:

```
...
XXX:process (CLK, RESET)
begin
    if RESET = '1' then Q <= "0000"; -- define reset state
    elsif rising_edge (CLK) then -- on active clock edge do:
```

```
      -- synchronous actions like load, clear, shift, set ...
   ...
```

Über das erste if-Statement am Beginn des Prozesses wird der Reset-Zustand der Schaltung festgelegt. Als Bedingung darf hier einzig und alleine das Reset-Signal vorkommen. Direkt nach diesem *if*-Statement wird in einem *elsif*-Statement auf die aktive Taktflanke überprüft. Erst nach diesem Statement folgen alle Aktionen wie löschen, setzen, laden, speichern u.s.w. Damit wird sichergestellt, daß alle Aktionen synchron zum Takt erfolgen, da diese nur zum Zeitpunkt der aktiven Taktflanke bearbeitet werden.

In der Sensitivity List des Prozesses befinden sich nur das Takt- und das Resetsignal, da Änderungen von Ausgangssignalen oder inneren Zuständen immer nur mit der aktiven Taktflanke oder bei einem Reset erfolgen können.

4.3. Beschreibung von Addierern, Subtrahierern und Zählern in VHDL

a) Architecture eines 4-Bit-Addierers mit Namen *beh_ADD4* für die zugehörige Entity *ADD4*. Die Entity besitzt die zwei 4-Bit-Eingänge OP1, OP2 sowie den 4-Bit-Ausgang SUM.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- Use arithmetic operators

architecture beh_ADD4 of ADD4 is
   begin
      SUM <= OP1 + OP2; -- Adder
   end beh_ADD4;
```

Zur Realisierung von Addierern kann der + Operator, der in der Package *numeric_std* der Library IEEE definiert ist, verwendet werden. Im vorliegenden Beispiel wird durch die Signalzuweisung ein 4-Bit-Addierer definiert, da die Signale SUM, OP1 und OP2 der zugehörigen Entity als 4-Bit-Ports deklariert wurden.

Es steht einem natürlich selbstverständlich frei, den Addierer auch ohne die Verwendung des + Operators, also über Gatterfunktionen zu realisieren. Vorteile bringt das jedoch nicht. Die Funktionalität bleibt jedenfalls dieselbe.

b) Architecture eines 4-Bit Subtrahierers mit Namen *beh_SUB4* für die zugehörige Entity *SUB4*. Die Entity besitzt die 4-Bit-Eingänge OP1, OP2 sowie den 4-Bit-Ausgang DIFF.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- Use arithmetic operators

architecture beh_SUB4 of SUB4 is
   begin
      DIFF <= OP1 - OP2; -- Subtracter
   end beh_SUB4;
```

Zur Realisierung von Subtrahierern kann der – Operator verwendet werden, der ebenso wie der + Operator im package *std_logic_unsigned* der Library IEEE definiert ist. Im vorliegenden Beispiel wird durch die Signalzuweisung ein 4-Bit-Subtrahierer definiert, da die Signale DIFF, OP1 und OP2 der zugehörigen Entity als 4-Bit-Ports deklariert wurden.

c) Architecture eines 4-Bit-Addierers mit Übertragsein- und Ausgang mit Namen *beh_ADD4C* für die zugehörige Entity *ADD4C*. Die Entity besitzt den Eingang CIN, die zwei 4-Bit-Eingänge OP1 und OP2, den Ausgang COUT sowie den 4-Bit-Ausgang SUM.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- Use arithmetic operators

architecture beh_ADD4C of ADD4C is

signal SUM5 : std_logic_vector(4 downto 0);     -- 5 bit sum

  begin
      SUM5 <= ('0' & OP1) + ('0' & OP2) + ("0000" & CIN); -- 5 bit Adder
      SUM  <= SUM5(3 downto 0); -- Assign 4 bit sum
      COUT <= SUM5(4); -- Assign carry out
  end beh_ADD4C;
```

Zur Realisierung eines Addierers mit Übertragsausgang wird einfach ein interner Signalvektor SUM5 deklariert, der um genau 1 Bit breiter ist, als die gewünschte Breite des Addierers. Zur Berücksichtigung des Übertragseingangs muß in der nachfolgenden Addition zusätzlich CIN zu OP1 und OP2 hinzuaddiert werden. Nachdem alle Operanden der + Operation die gleiche Breite wie das Ergebnis haben müssen, werden die Signale OP1, OP2 und CIN mit Hilfe des Verkettungsoperators & auf 5 Bits ergänzt. Das höchstwertigste Bit des Ergebnisses SUM5 stellt dann das Übertragsbit COUT dar, während die niederwertigeren Bits die gewünschte 4-Bit-Summe darstellen.

Man könnte nun vielleicht meinen, daß durch die obige Beschreibung tatsächlich ein 5-Bit-Addierer mit drei Eingängen beschrieben wird, bei denen die höherwertigen Bits der Eingänge auf logisch 0 gelegt werden. Schaltungsmäßig wird aber nur ein 4-Bit-Addierer mit Übertragsein- und Ausgang erzeugt. Auf diesen Aspekt wird in Punkt f) noch näher eingegangen.

d) Architecture eines 6-Bit-Up-/Down-Counters mit Namen *beh_CNTUPDWN* für die zugehörige Entity *CNTUPDWN*. Die Entity besitzt die Eingänge CLK, RESETB, LOAD, COUNT und DIR, den 6-Bit-Eingang CNTIN sowie den 6-Bit-Ausgang CNTOUT.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- Use arithmetic operators

architecture beh_CNTUPDWN of CNTUPDWN is

signal Q : std_logic_vector(5 downto 0); -- 6 bit counter value

  begin
      CNT:process (CLK, RESETB)
      begin
        if RESETB = '0' then Q <= "000000"; -- clear counter on reset
        elsif rising_edge (CLK) then
            if LOAD = '1' then Q <= CNTIN; -- load counter with CNTIN on LOAD
            elsif COUNT = '1' then
                  if DIR= '0' then Q <= Q - '1'; -- count down
                  else Q <= Q + '1'; -- count up
                  end if;
            end if;
        end if;
```

```
    end process CNT;

    CNTOUT <= Q;
end beh_CNTUPDWN;
```

Zähler können sehr einfach mit Hilfe des + bzw. – Operators realisiert werden. Das interne Signal Q stellt den aktuellen 6-Bit Zählerstand des Zählers dar. Ist LOAD = 1 wird der Zähler mit der positiven Taktflanke mit dem Wert von CNTIN geladen. Ist LOAD = 0 und COUNT = 1 zählt der Zähler mit der positiven Taktflanke. Im Falle DIR = 0 wird dabei vom aktuellen Zählerstand Q der Wert 1 subtrahiert, der Zähler zählt somit abwärts. Im Falle DIR = 1 wird hingegen zum aktuellen Zählerstand Q der Wert 1 addiert und der Zähler zählt damit aufwärts. Sind beide Steuereingänge LOAD und COUNT logisch 0, behält der Zähler den aktuellen Zählerstand bei.

Man stellt sich möglicherweise die Frage, warum für die Operationen im Prozeß CNT nicht direkt das Signal CNTOUT verwendet wurde, sondern das interne Signal Q eingeführt wurde. Die Erklärung ist, daß CNTOUT in der Entity als Ausgangsport definiert wurde. An Ausgangsports dürfen aber generell immer nur Werte zugewiesen werden. Es darf niemals von einem Ausgangsport gelesen werden, das ist nur von Eingangsports oder Signalen möglich. Anders gesagt, dürfen Ausgangsports niemals auf der rechten Seite einer Signalzuweisung oder als Bedingung in *if*- oder *case*-Statements verwendet werden. Das wäre hier aber in den obigen Zuweisungen `Q <= Q - '1'` und `Q <= Q + '1'` genau der Fall, wenn statt des internen Signals Q der Ausgang CNTOUT verwendet werden würde. In solch einem Fall wird also einfach ein internes Signal eingeführt, das dem entsprechenden Ausgangsport im Rahmen einer Signalzuweisung (im obigen Fall als letzte Zeile der Architecture) zugewiesen wird.

e) Architecture eines 8-Bit-Aufwärtszählers mit Namen *beh_CNTUP* für die zugehörige Entity *CNTUP*. Die Entity besitzt die Eingänge CLK, RESETB, COUNT, SET0, SET16 und SET64, den Ausgang CNTOUT100 sowie den 8-Bit-Ausgang CNTOUT.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all; -- Use arithmetic operators

architecture beh_CNTUP of CNTUP is

signal Q : std_logic_vector(7 downto 0); -- 8 bit counter value

  begin
    CNT:process (CLK, RESETB)
    begin
      if RESETB = '0' then Q <= "00000000"; -- clear counter on reset
      elsif falling_edge (CLK) then
          if SET0 = '1' then Q <= "00000000"; -- clear counter synchronously
          elsif SET16 = '1' then Q <= "00010000"; -- set counter to 16 decimal
          elsif SET64 = '1' then Q <= "01000000"; -- set counter to 64 decimal
          elsif COUNT = '1' and Q /= "11000111" then
                Q <= Q + '1'; -- count up if counter has not reached 199
          end if;
      end if;
    end process CNT;

    CNTOUT <= Q;
    -- Generate pulse on CNTOUT100 if counter reaches 100 decimal:
    CNTOUT100 <= '1' when Q(7 downto 0) = "01100100" else '0';
  end beh_CNTUP;
```

Über die Signale SET0, SET16 und SET64 wird dem Signal Q mit der negativen Taktflanke eine bestimmte Konstante zugewiesen und der Zähler damit auf einen definierten Wert gesetzt. Ist hingegen COUNT gleich 1 und der aktuelle Zählerstand kleiner als der dezimale Wert 199, wird der Zähler mit der negativen Taktflanke um 1 inkrementiert. Im Falle, das der Zählerstand 199 erreicht wurde oder alle 4 Steuereingänge SET0, SET16, SET64 und COUNT logisch 0 sind, bleibt der alte Zählerstand erhalten.

Man beachte, daß der Zählerstand hier auf Ungleichheit mit dem dezimalen Wert 199 überprüft wurde (der Operator /= stellt in VHDL den *Ungleichheitsoperator* dar). Ein Vergleich auf Gleichheit oder Ungleichheit ist in Hardware relativ einfach zu realisieren. Im wesentlichen beschränkt sich dies schaltungsmäßig auf ein Exklusiv-ODER-Verknüpfung der zu vergleichenden Operanden. Würde man im obigen *if*-Statement als Bedingung nun `Q < "11000111"` statt `Q /= "11000111"` schreiben, würde das zwar funktionell dasselbe Ergebnis bedeuten, schaltungsmäßig würde hier aber ein 8-Bit-Subtrahierer statt eines 8-fach-EXOR-Gatters realisiert werden. Ein schaltungstechnischer Mehraufwand also, der für die gewünschte Funktion nicht notwendig ist.

Ähnliche Überlegungen gelten für die Zuweisung des Wertes von CNTOUT100. Wie von der Angabe gefordert, wird der Ausgang CNTOUT100 beim Erreichen des Zählerstandes 100 mindestens für eine Taktperiode (und hier genau für eine Taktperiode, nämlich wenn der Zählerstand genau 100 beträgt) auf 1 gesetzt. Würde man bei der Zuweisung von CNTOUT100 im obigen *when*-Statement als Bedingung `Q(7 downto 0) >= "01100100"` statt `Q(7 downto 0) = "01100100"` schreiben, würde dies zwar ebenfalls die Angabe erfüllen, würde aber nur einen unverhältnismäßig höheren Schaltungsaufwand bedeuten, da in diesem Fall ein Subtrahierer realisiert wird.

Man überlege sich also bei der Verwendung der Vergleichsoperatoren <, <=, > und >= stets, ob diese wirklich notwendig sind oder ob nicht = und /= die gewünschte Funktionalität ebenso erfüllen.

f) Das VHDL-Synthesetool hat die Aufgabe eine in VHDL beschriebene Funktionalität, mit Hilfe einer Herstellerbibliothek, auf eine digitale Hardware abzubilden. Die Herstellerbibliothek[9] enthält alle Grundelemente, wie etwa diverse logische Gatter und Flip-Flops, aus denen eine Schaltung zusammengesetzt ist. Ergebnis der Synthese ist eine Netzliste, in der die Grundelemente der Herstellerbibliothek instanziert und miteinander verbunden werden.

Je nachdem, ob man einen ASIC bei Toshiba, Motorola, Siemens oder einer anderen Halbleiterfirma fertigen läßt, hat man es dabei mit unterschiedlichsten Bibliotheken zu tun. Die Herstellerbibliothek ist darüber hinaus von der verwendeten Technologie abhängig. In Abhängigkeit davon, ob man eine Schaltung als ASIC in einem CMOS-Prozeß oder in Gallium-Arsenid-Technologie fertigen läßt oder in einem FPGA oder CPLD implementiert, befinden sich die unterschiedlichsten Bauelemente mit unterschiedlichsten Timing-Parametern in der Bibliothek.

Als es noch keine abstrakten Hardwarebeschreibungssprachen wie VHDL gab, wurden digitale Designs mit Hilfe von Gatterbeschreibungssprachen beschrieben. Die in einer Bibliothek vorhandenen Bauelemente wurden dabei direkt bei der Beschreibung einer Schaltung angegeben. Das brachte enorme Probleme bei einem Wechsel des Herstellers oder bei einer Umstellung der Technologie mit sich. Wurden beispielsweise in einem Design 4-fach-UND-Gatter verwendet und das Design auf eine andere Technologie umgestellt, konnte das nur problemlos geschehen, wenn in auch in der neuen Bibliothek äquivalente 4-fach-UND-Gatter (mit gleicher Treiberstärke, gleichem Timing, etc.) vorhanden waren. War das nicht der Fall, mußte die Beschreibung auf die in der

[9] Die Herstellerbibliothek hat nichts mit der IEEE-Library oder ähnlichen Bibliotheken zu tun, die über das *library*-Statement eingebunden werden.

Bibliothek vorhanden Gatter umgeändert werden. Schlimmer noch war es bei einem Wechsel des Herstellers, da das Datenformat bzw. die vorhandenen Gatterbeschreibungssprachen oftmals vom Hersteller abhängig waren.

Mit der Einführung von VHDL sollte deshalb ein vom Hersteller und der Technologie unabhängiger, weltweiter Standard zur Beschreibung von digitaler Hardware geschaffen werden. Bei der Beschreibung mittels VHDL werden, anders als bei Gatterbeschreibungssprachen, keine Gatter oder Flip-Flops direkt angegeben, sondern nur deren Funktionalität. Wird zum Beispiel eine 4-fache UND-Verknüpfung in VHDL angegeben, wird vom Synthesetool diese Funktionalität beispielsweise in Form eines 4-fach-UND-Gatters realisiert, so es dieses Gatter in der verwendeten Bibliothek gibt. Wird nun der Hersteller oder die Technologie gewechselt und in der nun verwendeten Bibliothek existieren keine 4-fach-UND-Gatter, sondern nur 2-fach-UND-Gatter, so wird die Funktionalität des 4-fach-UND-Gatters vom Synthesetool einfach durch drei 2-fach-UND-Gatter realisiert. Dies geschieht vollautomatisch und ohne Zutun des Anwenders! Hardwarebeschreibungen in VHDL sind damit unabhängig vom Hersteller und der verwendeten Technologie.

Es soll nochmals betont werden, daß in VHDL nur die Funktionalität einer Schaltung beschrieben wird. Die etwa in Beispiel 4.2, Punkt c) beschriebene Funktionalität eines D-Flip-Flops muß vom Synthesetool nicht wirklich mit Hilfe eines D-Flip-Flops realisiert werden. Wie aus der Vorlesung bekannt ist und sich auch jederzeit leicht beweisen läßt, kann die Funktionalität eines D-Flip-Flops auch z.B. mit Hilfe eines JK-Flip-Flops und ein paar Gattern realisiert werden. Es kommt also immer auf die in der Herstellerbibliothek vorhandenen Bauelemente darauf an, wie eine Funktionalität vom Synthesetool letztlich schaltungsmäßig realisiert wird.

Es soll nicht verschwiegen werden, daß sich nicht unbedingt jede in VHDL beschriebene Funktionalität vom Synthesetool synthetisieren, also auf eine digitale Hardware abbilden läßt (das ist auch vom verwendeten Synthesetool abhängig). In VHDL gibt es Befehle die sich prinzipiell nicht synthetisieren lassen. Diese Befehle werden auch nicht zum Beschreiben von Hardware, sondern bei der Simulation von Schaltungen verwendet. Auf diesen Aspekt wird in Kapitel 5 noch eingegangen. Die in diesem Kapitel angegebenen Beispiele sollten sich jedenfalls mit allen gebräuchlichen Synthesetools synthetisieren lassen.

Die Aufgabe eines Synthesetools beschränkt sich im übrigen nicht nur auf die prinzipielle Erstellung einer Schaltung, sondern auch auf deren Minimierung. Es wurde schon bei dem in Punkt c) behandelten Addierer erwähnt, daß durch die angegebene Beschreibung nicht wirklich ein 5-Bit Addierer mit drei Eingängen erzeugt wird, bei denen die höherwertigen Bits der Eingänge auf logisch 0 gelegt werden. Das Synthesetool erkennt vielmehr, daß die höchstwertigsten Bits der beiden Operanden OP1 und OP2 sowie die 4 höherwertigen Bits von CIN durch die Konstantenzuweisung immer logisch 0 sind und erzeugt somit nur einen Addierer mit zwei 4-Bit Eingängen und einem Übertragsein- und Ausgang.

g) Vergleicht man das Erstellen von Hardware mittels VHDL mit dem Erstellen von lauffähigen Programmen über eine höhere Programmiersprache, läßt sich der Vorgang des Synthetisierens mit dem Vorgang des Compilierens vergleichen. Die Grundelemente, die sich in einer vom Hersteller und der Technologie abhängigen Bibliothek befinden, lassen sich mit dem vorhandenen Befehlssatz einer bestimmten CPU vergleichen. Das Synthesetool erzeugt mit Hilfe einer spezifischen Herstellerbibliothek aus dem VHDL-Quellcode eine funktionierende Hardware, während der Compiler aus dem Quellcode der Hochsprache ein auf dem Befehlssatz eines Prozessors aufbauendes, lauffähiges Programm erzeugt.

Im allgemeinen kümmert man sich bei der Erstellung von Programmen mittels einer Hochsprache jedoch nicht darum, wie der exekutierbare Code, der vom Compiler erzeugt wird, aussieht (abgesehen von Fällen, bei denen es um die Geschwindigkeit oder den notwendigen Speicherplatz des erzeugten Codes geht). Bei der Erstellung von Schaltungen mittels VHDL sollte man sich jedoch immer bewußt sein, welche Schaltung vom Synthesetool im konkreten Fall wirklich erzeugt wird. Wie in Punkt e) gezeigt wurde, kann alleine durch eine falsche Verwendung von Vergleichsoperatoren der schaltungsmäßige Umfang dramatisch ansteigen.

Die vielleicht wichtigste Regel bei der Erstellung von Hardware mittels VHDL lautet daher: Man sollte in VHDL-Programmen immer möglichst einfache Konstrukte verwenden, bei denen man weiß, welche Schaltung vom Synthesetool erzeugt wird. Ziel bei der Erstellung von Hardware mittels VHDL ist nicht die Minimierung von Quellcode, sondern die Minimierung der vom Synthesetool erzeugten Schaltung!

4.4. Beschreibung von Decodern in VHDL

a) Architecture des Befehlsdecoders aus Beispiel 3.4 mit Namen *beh_DEC* für die zugehörige Entity *DEC*. Die Entity besitzt den Eingang C, den 3-Bit Eingang OPCODE und den 6-Bit Ausgang S.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_DEC of DEC is
  begin
      DECODE:process (OPCODE, C)
      begin
            case OPCODE is
                  -- Decoder input => decoder output
                  when "000" => S <= "100010";
                  when "001" => S <= "100000";
                  when "010" => S <= "110010";
                  when "011" => S <= "110000";
                  when "100" => S <= "0-01-0";
                  when "101" => S <= "0-10-0";
                  when "110" => S <= "0-00-1";
                  when "111" =>
                        if C = '1' then S <= "0-00-1";
                        else S <= "0-00-0";
                        end if;
                  when others => S <= "------";
            end case;
      end process DECODE;
  end beh_DEC;
```

Die Architecture besteht im wesentlichen aus einem *case*-Statement, in dem allem möglichen Kombinationen des Eingangsvektors OPCODE abgefragt werden und der Ausgangsvektor S dementsprechend gesetzt wird. Im Falle OPCODE = "111" (bedingter Sprungbefehl) ist der zugewiesene Wert für S zusätzlich noch vom Zustand des Eingangs C abhängig, was durch das *if*-Statement berücksichtigt wird. Bei der Wertzuweisung von S wurde übrigens die in Beispiel 3.4, Punkt b) gewählte Befehlscodierung angenommen. Bei einer anderen Codierung ergeben sich zwar andere Werte für S, der prinzipielle Aufbau der Architecture bleibt allerdings derselbe.

Wie zu sehen ist, wird bei der Zuweisung an S ausgiebig Gebrauch vom *Don't Care* Operator gemacht. Dies erlaubt dem Synthesetool eine größere Freiheit beim Optimieren der zu erzeugenden Schaltung. Allerdings machen nicht alle Synthesetools von dieser Möglichkeit Gebrauch. Das im Rahmen der Laborübung verwendete Synthesetool der Entwicklungsumgebung MAX+plus II ersetzt Werte von – defaultmäßig immer durch logisch 0 (und gibt dabei während dem Synthetisieren eine entsprechende Meldung aus).

Auffällig ist in der obigen Architecture der Zweig *when others* am Ende des *case*-Statements. Man könnte meinen, daß bereits alle möglichen Eingangskombinationen von OPCODE berücksichtigt wurden und der *when others* Zweig damit nie durchlaufen wird. Tatsächlich ist es aber so, daß Signale nicht nur die Werte logisch 0 oder 1 annehmen können, sondern auch andere Werte, etwa wenn sie sich im hochohmigen Zustand befinden oder ihr Wert undefiniert ist (darauf wurde schon kurz bei der Vorstellung des Wertevorrats der Signale vom Typ *std_logic* und *std_logic_vector* in Beispiel 4.2, Punkt b hingewiesen). Damit wären aber durch die im obigen Beispiel vorhandenen *when*-Zweige nicht alle möglichen Eingangskombinationen abgedeckt und beim Kompilieren des VHDL-Sourcecodes würde ein Fehler angezeigt werden. Es gilt also die wichtige Regel, daß es für jedes *case*-Statement immer einen *when others* Zweig geben muß.

Wichtig ist auch, daß für das *if*-Statement im obigen Beispiel ein dazugehöriges *else*-Statement existiert. Wird dieses nicht eingefügt, nimmt das Synthesetool an, daß im Fall von C = 0 der Zustand von S beibehalten werden soll und erzeugt damit in der resultierenden Schaltung ein speicherndes Element d.h. ein Flip-Flop. Bei der Beschreibung von kombinatorischer Logik existiert also folgende wichtige Regel: Im Rahmen von *if*-Statements müssen immer alle möglichen Kombinationen einer Bedingung berücksichtigt werden, wobei in jedem Zweig immer allen Ausgangssignalen ein Wert zugewiesen werden muß. Dementsprechend muß es bei der Beschreibung von kombinatorischer Logik zu jedem *if*-Statement immer auch ein dazugehöriges *else*-Statement geben.

Nachdem der Ausgang S direkt vom Zustand der beiden Signale OPCODE und C abhängig ist, müssen sich sowohl OPCODE als auch C in der Sensitivity List des Prozesses befinden. Würde z.B. das Signal C in der Sensitivity List fehlen, wird bei einer Zustandsänderung von C der Prozeß nicht durchlaufen und S würde seinen alten Wert beibehalten. Das bedeutet aber, daß vom Synthesetool damit ein speicherndes Element, also ein Flip-Flop erzeugt werden würde.

b) Architecture des ROMs aus Beispiel 3.4 mit Namen *beh_ROM* für die zugehörige Entity *ROM*. Die Entity besitzt den 4-Bit-Eingang A und den 7-Bit-Ausgang D.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_ROM of ROM is
  begin
      ROM:process (A)
      begin
            case A is
                  -- ROM address => ROM data
                  when "0000" => D <= "000----"; -- RESET:    IN
                  when "0001" => D <= "0110101"; --           ADDC 5
                  when "0010" => D <= "1110111"; --           JC CARRYSET
                  when "0011" => D <= "100----"; --           OUTa
                  when "0100" => D <= "0010000"; --           CONST 0
                  when "0101" => D <= "101----"; --           OUTb
                  when "0110" => D <= "1100110"; -- END1:     JMP END1
                  when "0111" => D <= "100----"; -- CARRYSET: OUTa
```

```
                when "1000" => D <= "0010001"; --           CONST 1
                when "1001" => D <= "101----"; --           OUTb
                when "1010" => D <= "1101011"; -- END2:     JMP END2
                when others => D <= "-------";
            end case;
    end process ROM;
end beh_ROM;
```

Bei der Wertzuweisung von D wurde die in Beispiel 3.4, Punkt b) gewählte Befehlscodierung und das in Beispiel 3.4, Punkt c) beschriebene Programm zugrunde gelegt. Bei einer anderen Codierung oder bei einem anderen Programm ergeben sich zwar andere Werte für D bzw. A, der prinzipielle Aufbau der Architecture bleibt allerdings gleich.

Wie zu sehen ist, entspricht der Aufbau der Architecture des ROMs genau dem des Decoders aus Punkt a). Dies erscheint logisch, da funktionell zwischen einem Dekoder und einem ROM kein Unterschied besteht. Hier kündigt sich aber bereits an, daß eine funktionelle Beschreibungsweise nicht immer nur Vorteile bietet. Das Synthesetool wird nämlich das ROM aus Gattern implementieren, was zwar für obiges Beispiel noch tragbar ist, für große ROMs aber wegen der enormen Größe der Schaltung und damit des Platzbedarfs auf dem Chip in keinster Weise zu vertreten ist. Wie aus Grundlagenvorlesungen bekannt ist, werden ROMs in der Realität deshalb nicht aus Gattern, sondern aus wesentlich platzsparenderen Strukturen aufgebaut.

Nachdem es keinerlei Möglichkeit gibt dem Synthesetool mitzuteilen wie es obige Schaltung realisieren soll, ist VHDL damit zur Beschreibung von ROMs oder RAMs praktisch ungeeignet. ROMs oder RAMs werden deshalb als sogenannte *Makrozellen* implementiert. Es sind dies fertige Blöcke, die die Schaltung des ROMs oder RAMs enthalten und nicht mehr synthetisiert werden. Bei der Verwendung von Makrozellen werden diese als quasi fertiges Entity-/Architecture-Paar in das Design eingebunden. Die interne Schaltung der Makrozelle kann dabei vom Benutzer nicht mehr verändert werden. Der Benutzer hat nur Zugriff auf die Ports, also im Falle von ROMs oder RAMs auf die Adreß-, Daten- und Steuerleitungen der Makrozelle.

Makrozellen sind immer vom Hersteller und von der verwendeten Technologie abhängig. Bei einem Wechsel des Herstellers oder der Technologie muß demnach das Design gegebenenfalls auf die nunmehr vorhandenen Makrozellen angepaßt werden.

Im Rahmen der Laborübung sollen im übrigen keine Makrozellen verwendet werden. Aufgrund der geringen Größe der hier verwendeten ROMs werden diese immer über Gatter, also so wie im obigen Beispiel, realisiert.

4.5. Beschreibung von Moore- und Mealy-Automaten in VHDL

a) Architecture des Moore-Automaten mit Namen *beh_SM1* für die zugehörige Entity *SM1*. Die Entity besitzt die Eingänge CLK, RESETB, A und B sowie die beiden Ausgänge X und Y.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_SM1 of SM1 is

-- State encoding of the state machine: States are binary encoded here
constant S0 : std_logic_vector(1 downto 0) := "00";
constant S1 : std_logic_vector(1 downto 0) := "01";
constant S2 : std_logic_vector(1 downto 0) := "10";
```

```
constant S3 : std_logic_vector(1 downto 0) := "11";

-- Current state Qn of the state machine:
signal CURRENT_STATE : std_logic_vector(1 downto 0);

-- State after next active clock edge Qn+1:
signal NEXT_STATE : std_logic_vector(1 downto 0);

  begin
      -- State definition of the state machine:
      STATES:process (CURRENT_STATE, A, B)
      begin
            case CURRENT_STATE is

                  ------------------------------------------------------
                  -- STATE 0: In this state something happens ...
                  ------------------------------------------------------
                  when S0 =>
                        X <= '1'; -- Set X to 1
                        Y <= '1'; -- Set Y to 1

                        if A = '1' then -- State of A=1 => move to S0
                              NEXT_STATE <= S0;
                        elsif B = '0' then -- A=0, B=0 => move to S2
                              NEXT_STATE <= S2;
                        else NEXT_STATE <= S1; -- A=0, B=1 => move to S1
                        end if;

                  ------------------------------------------------------
                  -- STATE 1: In this state something happens ...
                  ------------------------------------------------------
                  when S1 =>
                        X <= '1'; -- Set X to 1
                        Y <= '0'; -- Set Y to 0

                        NEXT_STATE <= S3;

                  ------------------------------------------------------
                  -- STATE 2: In this state something happens ...
                  ------------------------------------------------------
                  when S2 =>
                        X <= '0'; -- Set X to 0
                        Y <= '-'; -- State of Y doesn't care

                        NEXT_STATE <= S0;

                  ------------------------------------------------------
                  -- STATE 3: In this state something happens ...
                  ------------------------------------------------------
                  when S3 =>
                        X <= '1'; -- Set X to 1
                        Y <= '-'; -- State of Y doesn't care

                        NEXT_STATE <= S0;

                  ------------------------------------------------------
                  -- OTHER STATE: Will never happen
                  ------------------------------------------------------
                  when others =>
                        X <= '-'; -- State of X doesn't care
                        Y <= '-'; -- State of Y doesn't care
```

```
                    NEXT_STATE <= "--";

        end case;
    end process STATES;

    -- Flip-Flops of the state machine:
    FF:process (CLK, RESETB)
    begin
      if RESETB = '0' then CURRENT_STATE <= S0; -- Reset state
      elsif rising_edge (CLK) then
          CURRENT_STATE <= NEXT_STATE; -- Change state on rising clock edge
      end if;
    end process FF;
end beh_SM1;
```

Die Architecture beginnt mit der Zustandscodierung des Automaten. Dabei werden über das Schlüsselwort *constant* den codierten Werten, welche den Zuständen des Automaten entsprechen, die logischen Namen *S0*, *S1*, *S2*, ... zugeordnet. Die Codierung wird vom Benutzer vorgegeben und wurde in diesem Fall einfach binär gewählt. Das die für den Automaten erforderliche kombinatorische Logik dann nicht immer minimal sein muß, versteht sich von selbst, ist doch der Umfang der benötigten kombinatorischen Logik von der Wahl der Zustandscodierung abhängig. Bei einer Beschreibung in dieser Form ist dies jedoch nicht zu verhindern und wird hier ganz einfach hingenommen. Im Falle, daß nicht auf Aspekte wie Hazards geachtet werden muß (siehe Punkt b), empfehle ich der Einfachheit halber eine binäre Codierung zu verwenden.

Die beiden nach der Zustandscodierung deklarierten internen Signale CURRENT_STATE und NEXT_STATE geben den aktuellen Zustand des Automaten und den Zustand nach der nächsten aktiven Taktflanke (abhängig vom aktuellen Zustand des Automaten und vom Wert der Eingangssignale) an.

Im nachfolgenden Prozeß *STATES* wird die Zustandsabfolge des Automaten beschreiben. Dabei entspricht jeder Zustand einem *when*-Zweig des *case*-Statements. Für jeden Zustand sollte in einem kurzen Kommentar beschrieben werden, was genau sich in diesem Zustand funktionell tut. Danach werden die Werte für die Ausgänge und der Folgezustand des Automaten (in Abhängigkeit vom Zustand der Eingänge) über Signalzuweisungen zugewiesen.

Dabei sind folgende Dinge zu beachten: Da im Prozeß *STATES* reine kombinatorische Logik beschrieben wird, muß in jedem Zustand immer ausnahmslos allen Ausgängen des Automaten ein Wert zugewiesen werden (dabei kann natürlich ausgiebiger Gebrauch vom *don't care* Operators gemacht werden). Andernfalls wird das Synthesetool bei der Synthese ein ungewolltes speicherndes Element erzeugen. Aus dem selben Grund muß es bei der Abfrage der Eingangssignale für jedes *if*-Statement auch immer ein *else*-Statement geben. Ebenso muß für das *case*-Statement immer ein *when others* Statement vorhanden sein, auch wenn dies, so wie in diesem Fall, für die Realität keinerlei Relevanz hat, da der in diesem Zweig behandelte Fall nie auftreten kann. Weiters müssen sich in der Sensitivity List des Prozesses STATES das Signal CURRENT_STATE und alle Eingangssignale des Automaten (in diesem Fall A und B) befinden. Zur Erläuterung all dieser Punkte siehe nochmals Beispiel 4.4, Punkt a).

Der zweite Prozeß der Architecture, genannt *FF*, beschreibt die Flip-Flops des Automaten, deren Ausgänge durch den Signalvektor CURRENT_STATE gegeben sind. Über die Zuweisung im Fall RESETB = 0 wird CURRENT_STATE der Wert für den Reset-Zustand des Automaten zugewiesen, d.h. hier wird der Reset-Zustand des Automaten definiert. Tritt kein Reset auf, wird

hingegen der Wert von NEXT_STATE mit der positiven Flanke über die Zuweisung `CURRENT_STATE <= NEXT_STATE` von CURRENT_STATE übernommen.

An Kommentaren sollte bei der Zustandsbeschreibung des Automaten nie gespart werden, da die Verständlichkeit ohnehin nicht so klar ist, wie eine graphischen Beschreibung (etwa durch einen Zustandsgraphen). Geben Sie möglichst immer folgende Dinge an: Was geschieht in jedem Zustand, warum wird der Wert eines Ausgangs auf diesen oder jenen Wert gesetzt und warum erfolgt ein Zustandswechsel in diesen oder jeden Zustand. Dabei sollten die Kommentare immer das funktionelle Verhalten wiedergeben und nicht so aussehen wie im obigen Beispiel (aus Ermangelung an der Kenntnis der Funktion des Automaten). Beispiele für schlechte Kommentare sind etwa:

```
RD <= '1'; -- Set RD to 1

if INT = '1' then -- if INT = 1 => move to state 21
    NEXT_STATE <= S21;
```

Solche Kommentare kann man sich sparen. Das beispielsweise das Signal RD auf logisch 1 gesetzt wird, sieht man ohnehin im Code. Schreiben Sie in diesem Fall lieber:

```
RD <= '1'; -- Begin of memory read access

if INT = '1' then -- Interrupt indication => process interrupt in state 21
    NEXT_STATE <= S21;
```

Abschließend soll angemerkt werden, daß sich Automaten in VHDL selbstverständlich auch strukturell, also durch Zusammenschaltung von Flip-Flops und Gattern (etwa durch Verwendung von D-Flip-Flops aus Beispiel 4.2, Punkt c und boolschen Funktionen nach Beispiel 4.1, Punkt f) beschreiben lassen. Nachdem in diesem Fall das Aufstellen der Gleichungen für das Schaltnetz des Automaten aber vom Benutzer selbst durchgeführt werden muß, sei der Sinn der strukturellen Beschreibung von Automaten in VHDL dahingestellt.

Am Beispiel der Automaten zeigt sich übrigens wieder der Vorteil von VHDL gegenüber Low-Level-Beschreibungssprachen. Als es noch keine abstrakten Beschreibungssprachen wie VHDL gab, existierten selbstverständlich auch Tools zur funktionellen Spezifikation von Automaten. Das Verhalten des Automaten wurde dabei zumeist über eine Übergangstabelle in tabellarischer Form angegeben. Das Tool erzeugte dann aus der Übergangstabelle die Schaltung mittels Flip-Flops und Gattern im Format einer bestimmten Gatterbeschreibungssprache. Die Beschreibung von digitalen Designs mußte also bei Automaten im (herstellerabhängigen) Format für die Beschreibung von Automaten und sonst im Format der (herstellerabhängigen) Gatterbeschreibungssprache erfolgen. Bei einer Beschreibung mittels VHDL erfolgt hingegen sowohl die Beschreibung von Automaten, als auch von Gatterfunktionen, Flip-Flops etc. im Format einer einzigen, noch dazu standardisierten Sprache.

b) Die Realisierung hazardfreier Ausgangssignale eines Automaten ist bei einer Beschreibung über VHDL prinzipiell nur möglich, wenn das Ausgangssignal direkt vom Ausgang eines Flip-Flops des Automaten entnommen wird. Das Ausgangssignal darf also nicht über eine kombinatorische Verknüpfung zweier oder mehrerer Signale erzeugt werden. Eine Begründung findet sich unter anderem in Punkt d).

Das Erzeugen von Ausgangssignalen, die direkt einem Ausgang eines Flip-Flops des Automaten entnommen sind, wird durch entsprechende Zustandscodierung des Automaten erzwungen. Um

etwa den Ausgang X des Automaten hazardfrei zu machen, kann folgende Codierung des Automaten verwendet werden:

```
-- State encoding of the state machine:
-- Used encoding achieves hazard free output X
constant S0 : std_logic_vector(2 downto 0) := "001";
constant S1 : std_logic_vector(2 downto 0) := "011";
constant S2 : std_logic_vector(2 downto 0) := "100";
constant S3 : std_logic_vector(2 downto 0) := "111";
```

Durch obige Codierung ist das niederwertigste Bit des Zustandsvektors in den Zuständen S0, S1 und S3 logisch 1. In genau diesen Zuständen soll aber auch das Signal X logisch 1 sein. Das Synthesetool wird somit den Ausgang X mit dem Ausgang des niederwertigsten Flip-Flops verbinden, da sich hierdurch die einfachste Gleichung zur Erzeugung des Signals X ergibt.

Wie zu sehen ist, muß der Zustandsvektor für diese Codierung allerdings um ein Bit erweitert werden, da ein einzelnes Bit bei einem 2 Bit breiten Zustandsvektor nicht in 3 Zuständen an der selben Stelle 1 sein kann. Man erkauft sich die Hazardfreiheit des Ausgangs X in diesem Fall also mit einem zusätzlichen Flip-Flop.

Mit den 3 benötigten Flip-Flops der Schaltung kann sich der Automat nunmehr in 8 internen Zuständen befinden, wobei 4 Zustände (S0 bis S3) von der gegebenen Aufgabenstellung benötigt werden. In der Realität kann es nun passieren, daß der Automat in einen der nicht benötigten Zustände S4 bis S7 kippt. Man sollte in diesem Fall bzw. generell immer dann wenn es nicht benötigte Zustände in einem Automaten gibt, den Ausgängen und dem Folgezustand im *when others* Zweig des *case*-Statements innerhalb des *STATES*-Prozesses "ungefährliche" Werte zuweisen (im Sinne der Überlegungen aus Beispiel 1.2, Punkt d, Variante 3).

c) Architecture des Mealy-Automaten aus Beispiel 2.4 mit Namen *beh_RDYLOGIC* für die zugehörige Entity *RDYLOGIC*. Die Entity besitzt die Eingänge CLK, RESETB und RDB sowie den Ausgang RDY.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_RDYLOGIC of RDYLOGIC is

-- State encoding of the state machine: States are encoded in gray code
constant S0 : std_logic_vector(1 downto 0) := "00";
constant S1 : std_logic_vector(1 downto 0) := "01";
constant S2 : std_logic_vector(1 downto 0) := "11";
constant S3 : std_logic_vector(1 downto 0) := "10";

-- Current state Qn of the state machine:
signal CURRENT_STATE : std_logic_vector(1 downto 0);

-- State after next active clock edge Qn+1:
signal NEXT_STATE : std_logic_vector(1 downto 0);

  begin
      -- State definition of the state machine:
      STATES:process (CURRENT_STATE, RDB)
      begin
            case CURRENT_STATE is

                  ------------------------------------------------------
```

```
                -- STATE 0: IDLE state, wait for read access from CPU
                -----------------------------------------------------
                when S0 =>
                      if RDB = '1' then -- IDLE state, no read access
                            RDY <= '1'; -- No wait cycle required
                            NEXT_STATE <= S0; -- Resume with IDLE state

                      else -- Read access from CPU
                            RDY <= '0'; -- Start CPU wait state
                            NEXT_STATE <= S1; -- Resume with wait state
                                              -- in state S1
                      end if;

                -----------------------------------------------------
                -- STATE 1: Resume with wait state 1
                -----------------------------------------------------
                when S1 =>
                      RDY <= '0'; -- delay read access for another 100 nS

                      NEXT_STATE <= S2; -- Resume with wait state in S2

                -----------------------------------------------------
                -- STATE 2: Resume with wait state 2
                -----------------------------------------------------
                when S2 =>
                      RDY <= '0'; -- delay read access for another 100 nS

                      NEXT_STATE <= S3; -- End wait state in S3

                -----------------------------------------------------
                -- STATE 3: End of wait state
                -----------------------------------------------------
                when S3 =>
                      RDY <= '1'; -- set RDY to 1 for 100 nS

                      NEXT_STATE <= S0; – go back to IDLE state S0

                -----------------------------------------------------
                -- OTHER STATE: Will never happen
                -----------------------------------------------------
                when others =>
                      RDY <= '-'; -- State of RDY doesn't care

                      NEXT_STATE <= "--";

          end case;
    end process STATES;

    -- Flip-Flops of the state machine:
    FF:process (CLK, RESETB)
    begin
      if RESETB = '0' then CURRENT_STATE <= S0; -- Reset state
      elsif rising_edge (CLK) then
          CURRENT_STATE <= NEXT_STATE; -- Change state on rising clock edge
      end if;
    end process FF;
  end beh_RDYLOGIC;
```

Die Architecture des Mealy-Automaten besitzt den gleichen Aufbau wie die Architecture des Moore-Automaten. Der einzige Unterschied besteht darin, daß hier eine Zuweisung an das Ausgangssignal RDY innerhalb des *if*-Statements stattfindet. Der Wert des Ausgangs RDY ist

damit direkt von Zustand des Eingangs RDB abhängig, was ja für Mealy-Automaten charakteristisch ist.

Ein Moore-Automat kann von einem Mealy-Automat im VHDL-Code einer Architecture obigen Aussehens generell daran unterschieden werden, daß sich Signalzuweisungen an Ausgänge bei einem Moore-Automaten niemals innerhalb der *if*-Statements befinden können. Innerhalb der *if*-Statements dürfen sich bei einem Moore-Automat nur Zuweisungen an den Signalvektor NEXT_STATE befinden. Andernfalls hat man es mit einem Mealy-Automaten zu tun.

d) Eine Zustandscodierung im Gray-Code, für die in Punkt c) angegeben Architecture des Mealy-Automaten garantiert nicht, daß der Ausgang RDY frei von Hazards ist. Eine Hazardfreiheit wäre nur gegeben, wenn die kombinatorische Logik zum Erzeugen des Signals RDY genau so realisiert wird, wie dies in der Gleichung von Beispiel 2.4, Punkt f) gegeben ist. Nachdem aber die letztendliche schaltungstechnische Realisierung der kombinatorischen Logik vom Synthesetool bestimmt wird, kann die Hazard-Freiheit hier nicht garantiert werden.

Will man daher, daß der Ausgang RDY frei von Hazards ist, muß das Synthesetool durch eine Codierung nach Punkt b) so beeinflußt werden, daß das Signal RDY direkt einem Ausgang der Flip-Flops des Automaten entnommen wird.

4.6. Aufbau von hierarchischen Designs mittels VHDL

Architecture des Microcontroller aus Beispiel 3.2 mit Namen *struct_MC* für die zugehörige Entity *MC*. Die Entity besitzt die Eingangssignale CLK, RES und I sowie den 4-Bit-Ausgangsvektor Q.

```
library IEEE;
use IEEE.std_logic_1164.all;

architecture struct_MC of MC is

-- Component declarations:

-- Component declaration of counter CNT
component CNT
  port(
            C     : in  std_logic;
            RES   : in  std_logic;
            P     : in  std_logic;
            DIN   : in  std_logic_vector(3 downto 0);
            DOUT  : out std_logic_vector(3 downto 0)
       );
end component;

-- Component declaration of ROM
component ROM
  port(
            ADDR  : in  std_logic_vector(3 downto 0);
            D     : out std_logic_vector(6 downto 0)
       );
end component;

-- Component declaration of decoder DEC
component DEC
  port(
            E     : in  std_logic_vector(2 downto 0);
```

```
            A     : out std_logic_vector(7 downto 0)
       );
end component;

-- Component declaration of register REG
component REG
  port(
            C     : in  std_logic;
            RES   : in  std_logic;
            EN    : in  std_logic;
            D     : in  std_logic_vector(3 downto 0);
            Q     : out std_logic_vector(3 downto 0)
       );
end component;

-- Signal declarations:

signal P : std_logic; -- inc (P=0) or load (P=1) PC
signal CMD : std_logic_vector(7 downto 0);-- Command vector:
                                          -- CMD[0] = JP (Jump)
                                          -- CMD[1] = JPZ (Jump Zero)
                                          -- CMD[2] = JPNZ (Jump Not Zero)
                                          -- CMD[4] = Out Data (Output Data)
signal ADDR : std_logic_vector(3 downto 0); -- 4 bit ROM address
signal DATA : std_logic_vector(6 downto 0); -- 7 bit databus

  begin
      -- Component instanciations:

      -- Component instanciation of counter CNT = program counter PC
      PC : CNT
            port map(
                        C => CLK,
                        RES => RES,
                        P => P,
                        DIN => DATA(3 downto 0),
                        DOUT => ADDR
                    );

      -- Component instanciation of ROM = program memory PROGMEM
      PROGMEM : ROM
            port map(
                        ADDR => ADDR,
                        D => DATA
                    );

      -- Component instanciation of decoder DEC = command decoder CMDDEC
      CMDDEC : DEC
            port map(
                        E => DATA(6 downto 4),
                        A => CMD
                    );

      -- Component instanciation of register REG = output register OUTREG
      OUTREG : REG
            port map(
                        C => CLK,
                        RES => RES,
                        EN => CMD(4),
                        D => DATA(3 downto 0),
                        Q => Q
                    );
```

```
    -- Generate load signal for program counter:
    P <= CMD(0) or (CMD(1) and not (I)) or (CMD(2) and I));

end struct_MC;
```

Die Architecture beginnt mit den *Component Declarations* aller verwendeten Komponenten des Microcontrollers (Schlüsselwort *component*). Hierbei werden alle Entity-/Architecture-Paare deklariert, die im Rahmen des Designs *MC* von der Architecture *struct_MC* als niederhierarchische Submodule verwendet werden. Nach der Deklaration aller verwendeten Komponenten, werden die internen Signale des Microcontrollers deklariert.

Am Beginn der Beschreibung der Funktionalität der Architecture, welche durch das Schlüsselwort *begin* eingeleitet wird, steht dann die *Instanzierung* der verwendeten Komponenten. Der Counter CNT wird also hier zur Realisierung des Program Counters PC verwendet, das ROM wird als Programmspeicher PROGMEM verwendet etc. Bei jeder Instanzierung wird im Rahmen eines *Port Mappings* angegeben, wie die Ports der verwendeten Komponenten mit den Signalen des Designs *MC* verbunden werden.

Am Schluß der Architecture befindet sich schließlich noch eine einzelne Zuweisung zur Erzeugung der Schaltung für das Signal P aus den Ausgängen des Befehlsdecoders und dem Eingang I.

Das Design des Microcontrollers *MC* ist ein Beispiel für eine *strukturelle Beschreibung* einer Schaltung mittels VHDL. Im Gegensatz zur *Verhaltensbeschreibung* einer Schaltung (*behaviour*), welche in den vorhergehenden VHDL-Beispielen dieses Kapitels zur Anwendung kam, ergibt sich die Funktionalität der Schaltung hier durch Verbindung von Unterkomponenten bzw. Submodulen. Durch die strukturelle Beschreibung ist die Möglichkeit zum Aufbau von komplexen hierarchischen Designs gegeben.

Zur Beschreibung von komplexen Designs werden zumeist beide Beschreibungsarten verwendet, wobei die Einordnung in eine der beiden Modellierungsarten nicht immer möglich ist. Streng genommen ist auch das Design *MC* eine Mischform beider Beschreibungsarten, da die Signalzuweisung des Signals P am Ende der Architecture wiederum eine Verhaltensbeschreibung darstellt.

5. Simulation, Testen, Design Rules

5.1. Design for Testability

a) Unter Simulation versteht man die Überprüfung des funktionellen Verhaltens eines Designs. Dabei werden während der Simulation an die Eingänge des Designs vom Benutzer definierte Werte, sogenannte *Input Stimuli* angelegt. Vom Simulator werden die Werte der Ausgänge des Designs berechnet und können dann mit den erwarteten Ergebnissen verglichen werden. Die Bewertung der Ergebnisse kann entweder manuell (durch den Benutzer bei Auswertung am Bildschirm) oder automatisch (vom Computer, über den Vergleich mit Referenzdaten) erfolgen.

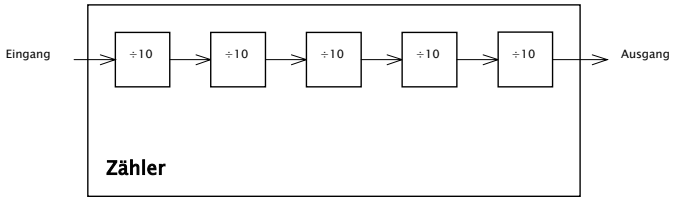
Beim Testen erfolgt, ebenso wie bei der Simulation, eine Überprüfung des funktionellen Verhaltens eines Designs. Im Unterschied zur Simulation erfolgt hier jedoch ein Test des physikalischen ASICs, wobei die Stimulation der Eingänge des ASICs über einen Pattern-Generator erfolgt. Die Ausgänge sind an einen Logik-Analysator angeschlossen und deren Zustände können entweder manuell (praktisch nur bei Prototypen) oder automatisch mit den erwarteten Ergebnissen verglichen werden. Bei der Massenfertigung werden hierbei eigene Tester verwendet, welche Pattern-Generator, Logik-Analysator und Auswertesysteme in einem Gerät beinhalten.

Schon bei der Erstellung der Input Stimuli für die Simulation sollte darauf geachtet werden, diese so auszulegen, daß sie möglichst ohne weitreichende Änderungen als Test-Pattern für einen anschließenden physikalischen Test herangezogen werden können. Dies ist nicht selbstverständlich, da bei der Simulation oft Gebrauch von internen Signalen gemacht wird, die man bei einem physikalischen Test aber nicht mehr zur Verfügung hat.

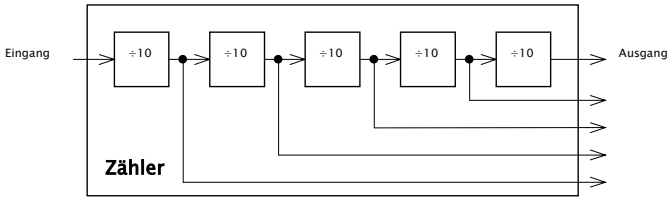
Die Design Rules entsprechen einem Satz allgemeiner Richtlinien für die Beschreibung von Designs, durch deren Einhaltung mögliche Fehlerquellen im vorhinein ausgeschlossen werden sollen. Insbesondere soll dadurch eine Unabhängigkeit von Technologie und Hersteller gewährleistet werden (Stichwort: *Synchrones Design*). Andererseits macht erst die Einhaltung gewisser Design Rules ein sinnvolles Simulieren und Testen möglich (Stichwort: *Design for Testability*).

b) Die Spezifikation eines ASICs erfolgt *Top Down*. Man geht also zunächst vom gewünschten funktionellen Verhalten aus und entwickelt ausgehend von der obersten Hierarchieebene in Richtung niedrigerer hierarchischer Schichten. Die Beschreibung und Simulation erfolgt dann *Bottom Up*. Zunächst wird ein Submodul der niedrigsten Hierarchieebene des Designs beschrieben. Hat man sich durch eine nachfolgenden Simulation von der korrekten Funktion des Moduls überzeugt, wird das nächste Modul dieser Hierarchieebene nach eben beschriebener Vorgangsweise entwickelt. Nachdem alle Module einer Ebene beschrieben und simuliert wurden, kann mit der Beschreibung des Moduls auf der nächsten Hierarchieebene, unter Einbindung der vorher entwickelten Submodule, begonnen werden. Auf diese Weise kommt man schließlich zum Top Level Design, welches nur mehr die physikalischen Ein- und Ausgänge des ASICs enthält.

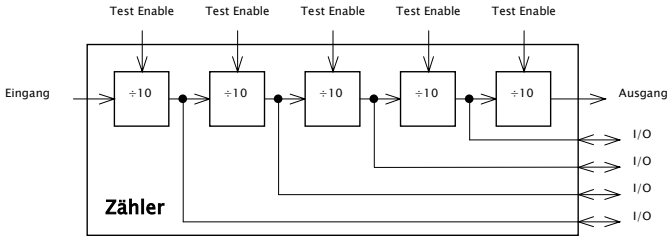
c) Folgendes Bild zeigt einen Dezimalzähler mit einem Zählbereich bis 100000:



Um den Zähler zu testen, müssen vom Testprogramm 100000 Impulse am Eingang des Zählers angelegt werden. Auch bei schnellen Testern ergeben sich hier lange Testzeiten. Da die Testzeit auf Chiptestern sehr teuer ist, würde sich der Bauteilpreis erheblich verteuern. Wird beim Test ein Fehlverhalten erkannt, kann außerdem nicht erkannt werden, in welcher Zählstufe der Fehler vorliegt. Durch das Herausführen der Ausgangssignale aller Zählstufen kann die Beobachtbarkeit des Systems erhöht werden:

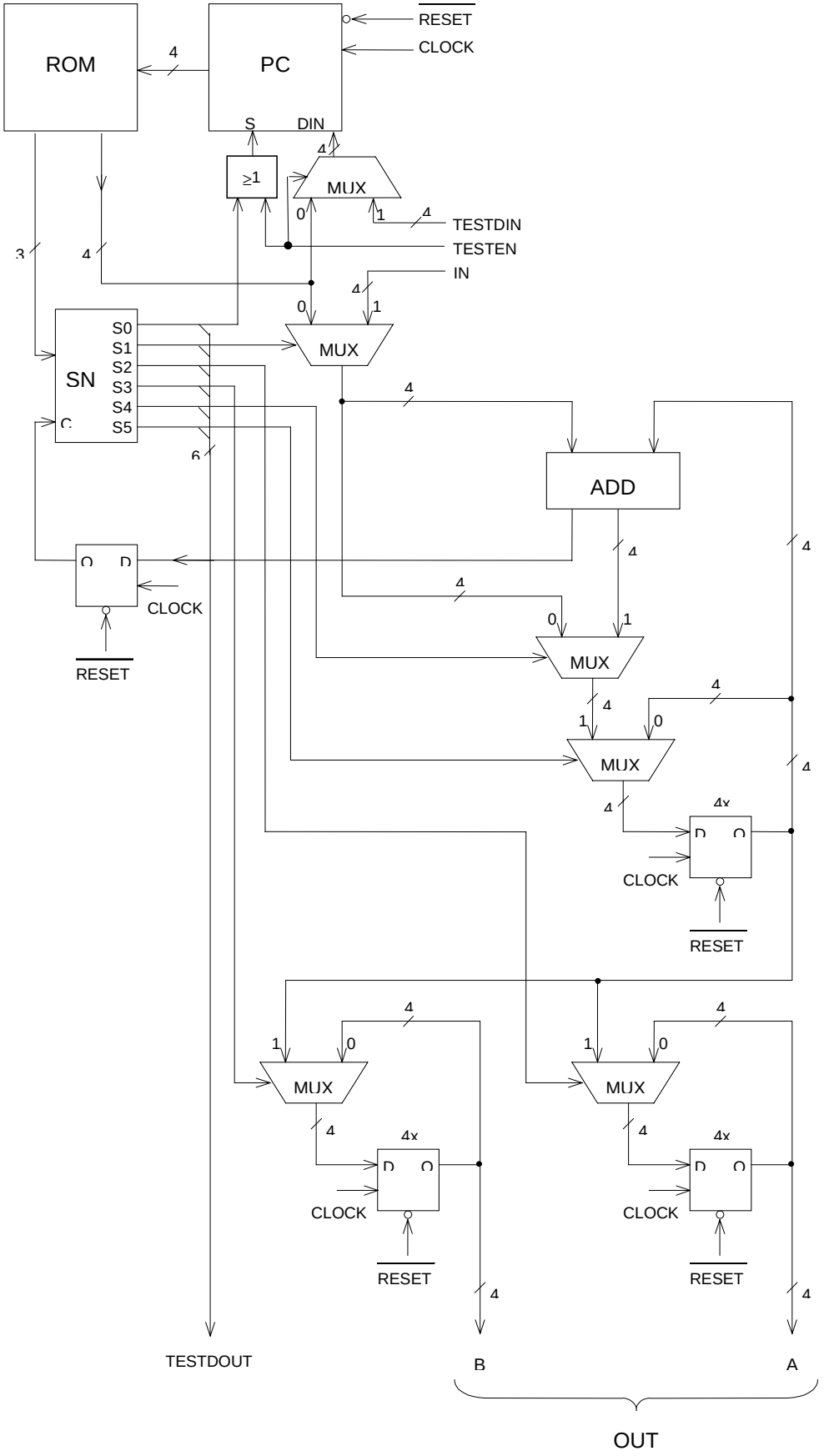


In obiger Schaltung kann nun bei einem Fehlverhalten die defekte Zählstufe lokalisiert werden. Will man etwa nur die niederwertigsten Zählstufen austesten, ergibt sich durch die gesteigerte Beobachtbarkeit außerdem eine Reduktion der benötigten Testschritte. Will man jedoch den ganzen Zähler austesten, benötigt man wiederum 100000 Testschritte. Obige Schaltung weist also eine gute Beobachtbarkeit, aber eine schlechte Steuerbarkeit auf. Durch folgende Erweiterung kann nun zusätzlich die Steuerbarkeit der Schaltung erhöht werden:



Die Ausgänge der einzelnen Zählstufen sind nun bidirektional ausgeführt. Jede Zählstufe erhält einen Test-Enable-Pin wodurch die bidirektionalen Pins als Eingang geschaltet werden können und jede einzelne Zählstufe mit einem Wert von außen geladen werden kann. Pro Zählstufe sind somit nur mehr 10 Testschritte notwendig. Obige Schaltung weist sowohl eine gute Beobachtbarkeit als auch eine gute Steuerbarkeit auf.

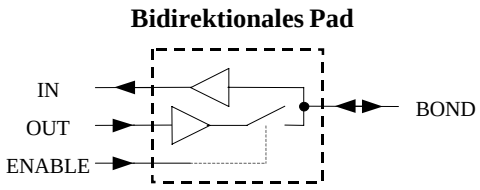
d) Erweiterung des Testbarkeit des Schaltwerks aus Beispiel 3.4 mittels 11 zusätzlicher Gehäusepins:


ROM
PC
RESET
CLOCK
S
DIN
≥1
MUX
TESTDIN
TESTEN
IN
SN
S0
S1
S2
S3
S4
S5
ADD
CLOCK
RESET
TESTDOUT
B
A
OUT

Damit die Ausgangssignale des Schaltnetzes SN außerhalb des Chips sichtbar sind, werden diese einfach an 6 der bisher nicht belegten Gehäusepins des Chips geführt (Signalvektor TESTDOUT[5:0]). Damit wird die Beobachtbarkeit der Schaltung erhöht.

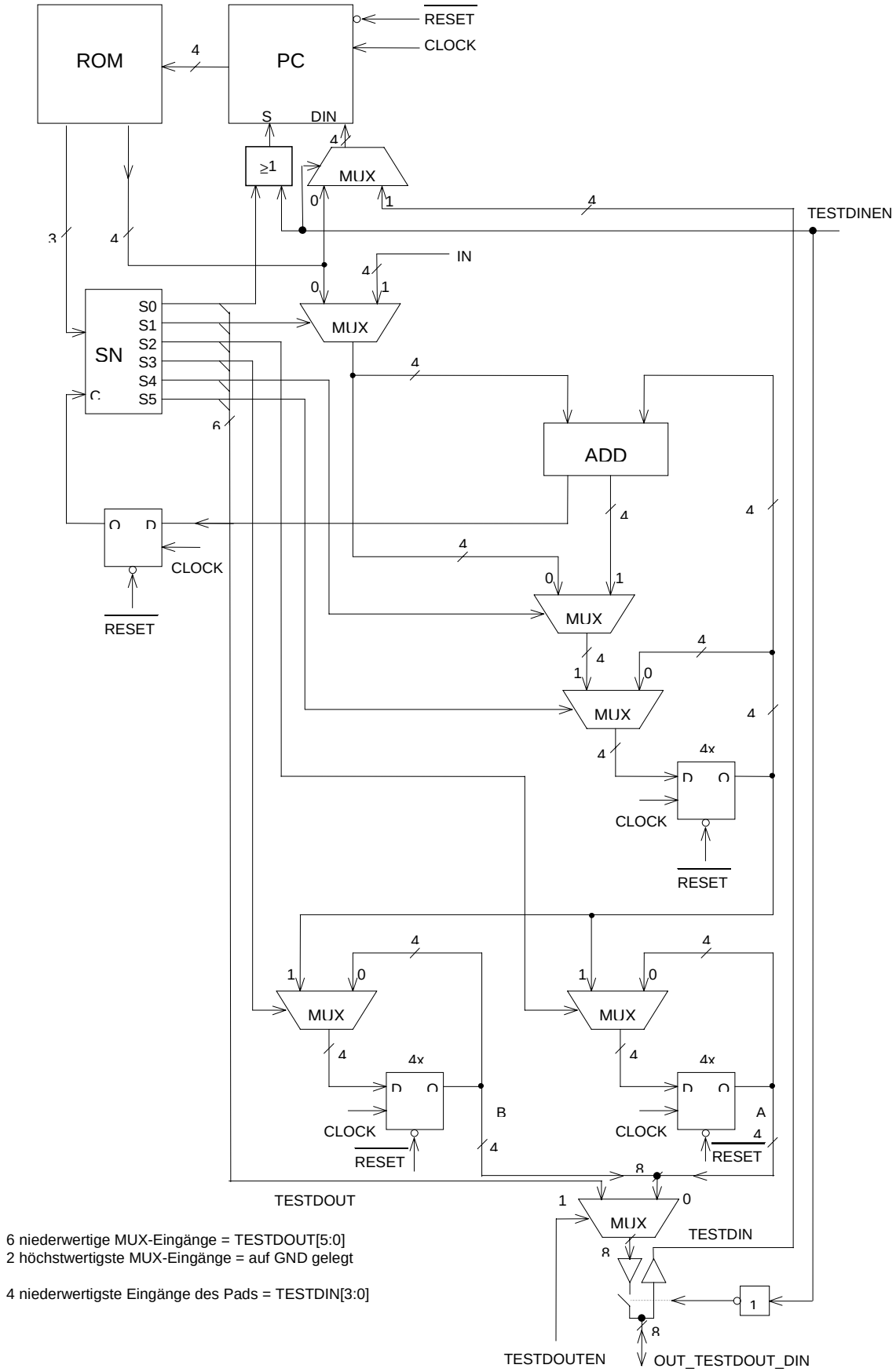
Setzt man den Test-Enable-Eingang TESTEN = 1, wird der Program Counter PC bei der nächsten aktiven Taktflanke über den Multiplexer mit der am Testdateneingang TESTDIN[3:0] anliegenden Adresse geladen. Damit wird die Steuerbarkeit der Schaltung erhöht.

e) Bidirektionale Pads dienen zum Umwandeln von unidirektionalen Signalen in bidirektionale Signale. Folgenden Abbildung zeigt das Schaltbild eines bidirektionalen Pads:



Die Signale IN, OUT und ENABLE kommen vom Inneren des Chips, während das Signal BOND über einen Bonddraht mit dem Gehäusepin verbunden ist. Ist das Signal ENABLE = 0 ist der Ausgangstreiber des Pads deaktiviert und das Pad ist als Eingang geschaltet. In diesem Fall wird das Signal IN über BOND von einem externen Signal getrieben. Ist ENABLE = 1 ist der Ausgangstreiber des Pads aktiviert. Das Pad ist als Ausgang geschälten und BOND sowie IN werden vom Signal OUT getrieben.

f) Erweiterung des Testbarkeit des Schaltwerks aus Beispiel 3.4 mittels 2 zusätzlicher Gehäusepins:



Zur Realisierung der geforderten Testfunktionen mit nur 2 zusätzlichen Gehäusepins TESTDINEN (Test Data In) und TESTDOUTEN (Test Data Out Enable) werden die 8 Pins des Ports OUT (4

Pins von Port OUTa und 4 Pins von Port OUTb) mehrfach verwendet. Man betrachte zunächst den Fall TESTDINEN = 0. In diesem Fall sind die Ausgangstreiber der 8 bidirektionalen Pads aktiviert, d.h. die Pins OUT_TESTDOUT_DIN sind als Ausgang geschaltet. Über das Signal TESTDOUTEN können wahlweise die Ports OUTa und OUTb (wenn TESTDOUTEN = 0) oder die internen Signale des Schaltnetzes TESTDOUT (wenn TESTDOUTEN = 1) an OUT_TESTDOUT_DIN ausgegeben werden. Da das Signal TESTDOUT nur 6 Bit breit ist, werden die zwei höchstwertigsten Bits des Multiplexer-Eingangs 1 auf GND gelegt (es könnten hier aber auch weitere interne Signale des Schaltwerks angeschlossen werden).

Im Fall TESTDINEN = 1 sind die Ausgangstreiber der 8 bidirektionalen Pads deaktiviert. Die Pins OUT_TESTDOUT_DIN sind als Eingang geschaltet. Die 4 niederwertigsten Bits von OUT_TESTDOUT_DIN sind über einen Multiplexer mit dem Dateneingang des Program Counters PC verbunden. Damit kann der Program Counter von außen mit einer Adresse geladen werden.

Die Verwendung von Multiplexern und bidirektionalen Pads ist ein oft verwendetes, weil relativ einfaches Mittel, um interne Signale einer Schaltung aus dem Chip heraus- bzw. in den Chip hineinzuführen.

5.2. Simulation in VHDL

a) Unter der Logiksimulation versteht man die Simulation vor der Synthese, also nur anhand der im VHDL-Sourcecode beschriebenen Funktionalität. Im Gegensatz dazu spricht man bei einer Simulation nach der Synthese auch von einer sogenannten *Gate Level Simulation*, da die Simulation hier auf einer wesentlich tieferen Ebene, eben unter Zugrundelegung der vom Synthesetool aus den Library-Elementen zusammengesetzten Schaltung, stattfindet.

Die Logiksimulation hat den Vorteil, daß man bei einer Änderung des Designs nicht auf die Synthese warten muß. Die Synthese komplexer Digital-Designs kann nämlich auch mit modernen Workstations Stunden und Tage dauern. Nachteilig bei der Logiksimulation ist die fehlende Timing-Information. Nachdem zu diesem Zeitpunkt noch nicht bekannt ist, wie das Synthesetool die Schaltung letztendlich aus den Elementen der Herstellerbibliothek zusammensetzt, ist somit auch keinerlei Information über Laufzeiten bekannt.

b) Zusammengehöriges Entity-/Architecture-Paar einer Testbench zur Simulation des positiv flankengetriggerten D-Flip-Flops aus Beispiel 4.2, Punkt c):

```
entity tb_DFF is
end tb_DFF;

library IEEE;
use IEEE.std_logic_1164.all;

architecture beh_tb_DFF of tb_DFF is

-- Component declaration of D-Flip-Flop:
component DFF
  port(
            CLK         : in  std_logic;
            RESETB      : in  std_logic;
            D           : in  std_logic;
            Q           : out std_logic
      );
end component;
```

```
-- All ports of the component above must be declared as testbench signals:
signal CLK : std_logic; -- Clock signal
signal RESETB : std_logic; -- Active low reset signal
signal D : std_logic; -- Data input of DFF
signal Q : std_logic; -- Data output of DFF

  begin
      -- Component instanciation of D-Flip-Flop
      DFF : DFF
            port map(
                        CLK => CLK,
                        RESETB => RESETB,
                        D => D,
                        Q => Q
                    );

      -- Simulation starts in process tb:
      tb:process
      begin
        -- Initial state of all input signals:
        CLK <= '0';
        RESETB <= '0';
        D <= '0';

        wait for 100 ns; -- Suspend execution of process tb for 100 ns

        assert Q = '0' report "Simulation error 1: Incorrect reset state";
        RESETB <= '1'; -- End reset state

        wait for 100 ns; -- Suspend execution of process tb for 100 ns

        CLK <= '1'; -- Rising clock edge

        wait for 50 ns; -- Suspend execution of process tb for 50 ns

        assert Q = '0' report "Simulation error 2: State of Q unequal 0";
        CLK <= '0'; -- Falling clock edge
        D <= '1'; -- Set data input of flip flop to 1

        wait for 50 ns; -- Suspend execution of process tb for 50 ns

        CLK <= '1'; -- Rising clock edge

        wait for 50 ns; -- Suspend execution of process tb for 50 ns

        assert Q = '1' report "Simulation error 3: State of Q unequal 1";
        CLK <= '0'; -- Falling clock edge
        D <= '0'; -- Set data input of flip flop to 0

        wait for 50 ns; -- Suspend execution of process tb for 50 ns

        CLK <= '1'; -- Rising clock edge

        wait for 50 ns; -- Suspend execution of process tb for 50 ns

        assert Q = '0' report "Simulation error 4: State of Q unequal 0";
        assert false report "End of simulation reached";

        wait; -- wait forever

      end process tb;
```

```
end beh_tb_DFF;
```

Wie in obigem Beispiel zu sehen ist, befinden sich hier die zusammengehörige Entity und die Architecture in ein- und demselben File, was bei Testbenches oft so der Fall ist. Die Entity selbst besteht nämlich nur aus zwei Zeilen, sie besitzt keine Ports.

Die Architecture beginnt mit einer Component Declaration der zu simulierenden Komponente. In der nachfolgenden Signaldeklaration wird für alle Ports der zu simulierenden Komponente ein Signal deklariert, welches in der Instanzierung der Komponente mit dem entsprechenden Port verbunden wird.

Die eigentliche Simulation beginnt im Prozeß *tb*, wobei *tb* hier als Abkürzung für Testbench steht. Auffällig ist, daß für den Prozeß *tb* keine Sensitivity List existiert. Dies rührt einerseits daher, daß in diesem Prozeß keine Hardware wie in Kapitel 4, sondern ein Simulationsfall beschrieben wird. Der Prozeß wird damit genau ein Mal (nämlich bei der Initialisierung) zum Zwecke der Simulation durchlaufen. Ein anderer Grund für die fehlende Sensitivity List ist, daß der im Prozeß verwendete *wait*-Befehl nur in Prozessen ohne Sensitivity List verwendet werden darf.

Im Prozeß wird nun die sequentielle Abfolge der Simulation beschrieben. Zunächst wird allen Eingangssignalen der zu simulierende Komponente ein Wert zugewiesen, welcher den Default-Wert zum Zeitpunkt 0 der Simulation angibt. Über den nachfolgenden *wait*-Befehl wird die Exekution des Prozesses für eine Zeit von 100 ns angehalten. Die folgende Zeile mit `RESETB <= '1'` (Ende des Resets) wird damit zur absoluten Simulationszeit 100 ns ausgeführt.

Vorher wird noch über das *assert*-Statement überprüft, ob der Wert des Ausgangs Q des D-Flip-Flops nach dem Reset logisch 0 ist. Die Meldung nach dem Schlüsselwort *report* wird genau dann am Device Standard Output (etwa ein Terminal-Fenster) ausgegeben, wenn die Bedingung im *assert*-Statements *false* ist. Um die Fehlermeldungen voneinander unterscheiden zu können, wird hier bei der Ausgabe jeder Meldung eine fortlaufende Nummer ausgegeben. Manche Simulatoren geben aus diesem Grunde bei der Ausgabe auch die Zeilennummer der Zeile des entsprechenden *assert*-Statements an.

Dann wird wieder 100 ns gewartet und danach (absolute Simulationszeit = 200 ns) die erste positive Taktflanke in der Simulation generiert. Nach weiteren 50 ns (absolute Simulationszeit = 250 ns) wird eine fallende Taktflanke erzeugt und der Eingang D des D-Flip-Flops auf 1 gesetzt (vorher wird wieder der korrekte Zustand von Q überprüft und bei einem Fehler eine entsprechende Meldung ausgegeben).

Wichtig ist, daß Wertzuweisungen an Eingangssignale während einer Simulation niemals zum Zeitpunkt der aktiven Taktflanke erfolgen dürfen. Was in solch einem Fall während der Simulation passiert (Fehlermeldung, undefinierte Ausgangssignale, ...) hängt einerseits von der Art der Simulation (Logiksimulation oder Simulation nach der Synthese) und andererseits vom verwendeten Simulator ab. In der obigen Testbench wird das Eingangssignal D beispielsweise immer zum Zeitpunkt der inaktiven (fallenden) Taktflanke gesetzt. Aus dem selben Grund erfolgt die Überprüfung von Q über das *assert*-Statement immer zum Zeitpunkt der fallenden Taktflanke.

Auf die eben beschriebene Weise kann der gesamte sequentielle Ablauf der Simulation beschrieben werden. Am Ende des Prozesses wird für den Benutzer noch eine entsprechende Meldung ausgegeben, daß das Ende der Simulation erreicht wurde. Auf die Ausgabe dieser Meldung sollte nie verzichtet werden. Im Gegensatz dazu kann auf die restlichen *assert*-Statements verzichtet werden, wenn der Benutzer die Verifikation der Ausgaben ohnehin manuell am Bildschirm

kontrollieren will. Der Prozeß endet mit *wait;* womit die Fortführung des Prozesses ins Unendliche verzögert wird.

Hier sieht man übrigens wieder einen großen Vorteil von VHDL, da es sich sowohl zur Beschreibung von Hardware als auch zur Beschreibung von Simulationen eignet. Bei der Erstellung einer Testbench muß man sich dann natürlich, im Gegensatz zur Beschreibung von Hardware (siehe Beispiel 4.3, Punkt g), nicht mehr auf wenige Befehle und Konstrukte beschränken, sondern kann auf den vollen Befehls- und Funktionsumfang dieser Sprache zurückgreifen.

Die im Labor verwendete Entwicklungsumgebung MAX+plus II unterstützt übrigens nicht die Simulation mittels Testbench. Hier werden die Input Stimuli über eine graphische Benutzeroberfläche eingegeben.

c) Würde man den Eingang D in die Sensitivity List des in Beispiel 4.2, Punkt c) abgebildeten Prozesses *D_Flip_Flop* hinzufügen, hätte das für die vom Synthesetool erzeugte Schaltung keinerlei Bedeutung. Für die Simulation hätte dies aber sehr wohl eine Bedeutung, da der Prozeß damit bei jeder Änderung des Zustandes von D, also auch ohne einer Änderung von CLK, durchlaufen würde. Dies macht die Simulation unnötigerweise langsamer. Im Hinblick auf eine schnelle Simulation sollten sich also bei allen Prozessen nur die wirklich notwendigen Signale in der Sensitivity List des Prozesses befinden.

d) Die in Punkt b) angegebene Testbench ist aufgrund des verwendeten *wait*-Befehls nicht synthetisierbar. Der *wait*-Befehl ist damit ein Beispiel eines nicht synthetisierbaren Befehls. Dies erklärt sich dadurch, daß es mit reinen Digitalschaltungen nicht möglich ist, beliebige zeitliche Verzögerungen zu realisieren. Mit reiner Digitallogik sind immer nur zeitliche Verzögerungen mit einem Vielfachen der verwendeten Taktfrequenz realisierbar[10].

e) Manche Simulationen und Tests lassen sich erst durch die Kenntnis des Zustandes innerer Signale rasch und effektiv erstellen und durchführen. Dazu muß man im konkreten Fall das innere Signal als Ausgangsport der zu simulierenden Entity herausführen. Benötigt man beispielsweise für die Simulation der CPU den Zustand eines inneren Signals des Addierers, muß das Signal über den Addierer und die ALU zur CPU „durchgeschleift“ werden. Dazu sind die Entities von Addierer, ALU und CPU um ein zusätzliches Port zu erweitern, in den Architectures ist eine zusätzliche Signaldeklaration nötig, die Komponentendeklarationen und -Instanzierungen sind zu modifizieren, ... u.s.w.

Eine Methode, den Aufwand in solch einem Fall zu verringern, besteht in der Einführung eines globalen Testbusses. Jedes Modul des Designs *XY* wird dabei von vornherein mit einem Ausgangsport *TESTBUS_XY* versehen. Die Breite dieses Testbusses ist von der Komplexität jedes Moduls abhängig. Die Testbusse aller Module werden durch das ganze Design geführt und bilden im Top Level Design einen einzigen breiten *TESTBUS*.

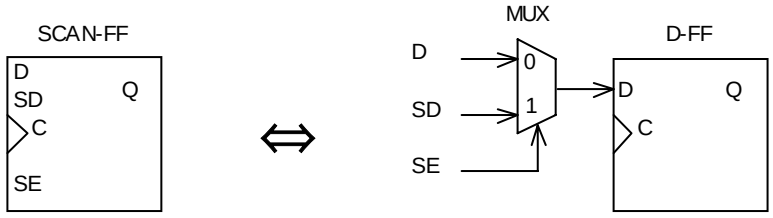
Den einzelnen Bits der Testbusse werden in den Architectures jedes Moduls zunächst konstante Werte (logisch 0 oder 1) zugewiesen. Soll nun ein inneres Signal eines beliebigen Submoduls zum Top Level Design herausgeführt werden, muß nur mehr das gewünschte Signal in der Architecture des betreffenden Moduls mit einem Bit des Testbusses verbunden werden.

[10] „Bastelschaltungen“, die die Laufzeiten von Gattern zur Erzeugung von zeitlichen Verzögerungen verwenden, stellen eine mißbräuchliche Verwendung von Bauelementen dar. Die Funktion eines logischen Gatters ist einzig und alleine die Realisierung einer bestimmten boolschen Funktion. Die Laufzeit eines Gatters (die noch dazu temperaturabhängig ist) ist jedoch keine erwünschte Funktion sondern ein Störeffekt.

Auf die Größe der Schaltung wirkt sich die Einführung des Testbusses nicht aus, denn werden Signale des Testbusses nicht verwendet, also beispielsweise an einen physikalischen Pin des Designs herausgeführt, werden sie ohne ordentliches Gerichtsverfahren vom Synthesetool wegoptimiert.

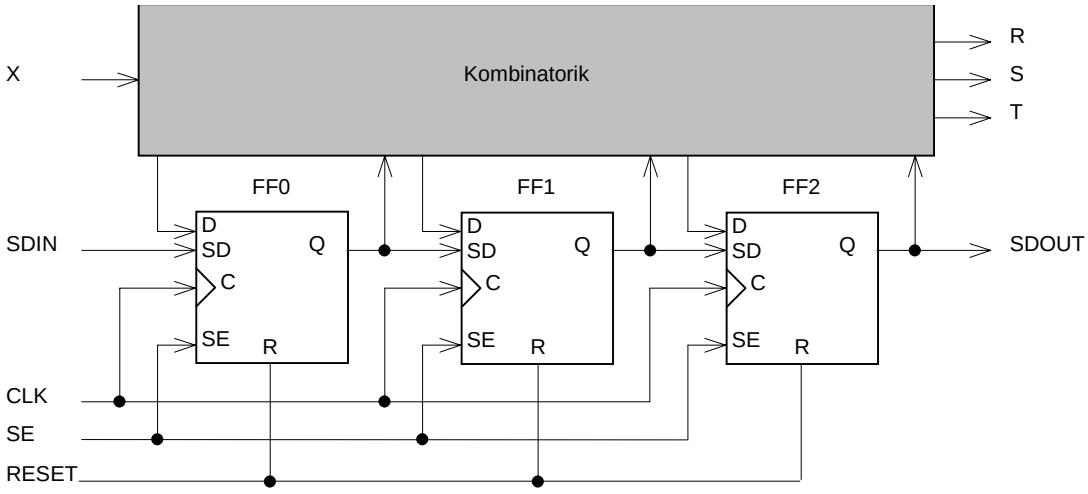
5.3. Scan-Path-Design

a) Schaltbild und Innenschaltung eines Scan-Flip-Flops:



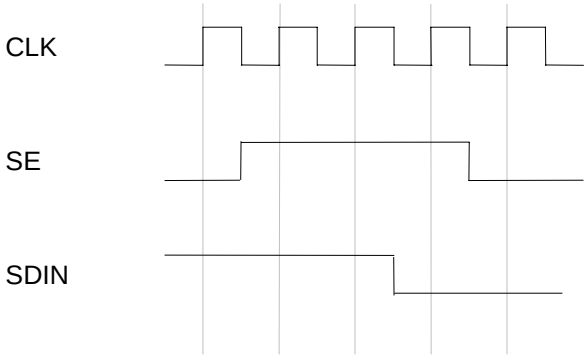
Über den Eingang SE (Scan Enable) wird zwischen Normalbetrieb und Scan-Betrieb umgeschaltet. Ist SE = 0, funktioniert das Scan-Flip-Flop wie ein normales D-Flip-Flop mit dem Dateneingang D. Ist hingegen SE = 1 (Scan-Betrieb), wird mit der nächsten Taktflanke vom Eingang SD (Scan Data) statt vom Eingang D eingelesen. Ein Scan-Flip-Flop stellt vom Prinzip her also nur ein D-Flip-Flop mit zwei Eingängen (D und SD) dar, welche über die Select-Leitung SE ausgewählt werden.

b) Schaltbild des Mealy-Automaten aus Beispiel 2.3 mit Scan-Flip-Flops:



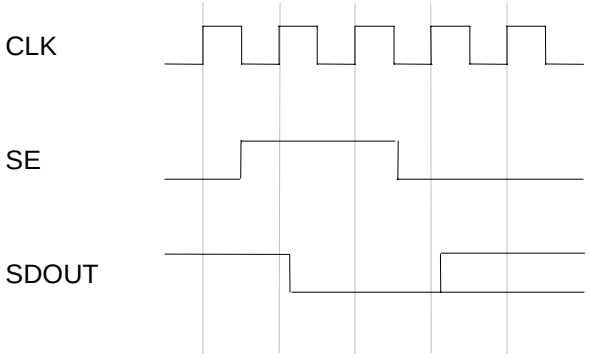
c) Der Automat besitzt zwei neue Eingänge SDIN und SE und den neuen Ausgang SDOUT. Aus den einzelnen Scan-Flip-Flops wird eine Kette gebildet, sodaß die Zusammenschaltung aller Flip-Flops wie ein großes Schieberegister wirkt. Der Eingang SD des ersten Flip-Flops der Kette wird als Eingang SDIN (Scan Data In), der Ausgang des letzten Flip-Flops der Kette wird als Ausgang SDOUT (Scan Data Out) aus dem Chip herausgeführt. SDIN und SDOUT wirken wie der Eingang und der Ausgang des Scan-Schieberegisters.

d) Timing des Testpatterns, um Automat in Zustand e zu setzen (mit Codierung aus Beispiel 2.3):



Nach dem Aktivieren des Scan-Modus durch SE = 1 werden mit den nächsten 3 positiven Taktflanken die an SDIN anliegenden Werte $1 \rightarrow 1 \rightarrow 0$ in die Flip-Flops geladen. Damit befindet sich der Automat in jenem Zustand, welcher durch die Zustandsvariablen $Q_2^n = 1$, $Q_1^n = 1$ und $Q_0^n = 0$ bestimmt ist. Nach der in Beispiel 2.3 gewählten Codierung entspricht dies dem Zustand e.

e) Timing aller Signale um Zustand des Automaten auszulesen:



Nach dem Aktivieren des Scan-Modus durch SE = 1 werden mit den nächsten 2 positiven Taktflanken die Ausgänge der niederwertigen Flip-Flops in SDOUT geschoben. Aus den anliegenden Werten $1 \rightarrow 0 \rightarrow 0$, welche den Zustandsvariablen $Q_2^n = 1$, $Q_1^n = 0$ und $Q_0^n = 0$ entsprechen, läßt sich somit der Zustand bestimmen, in dem sich der Automat vor dem Aktivieren von SE = 1 befunden hat. Nach der in Beispiel 2.3 gewählten Codierung entspricht dieser Zustandsvektor dem Zustand c.

f) Werden entgegen den Richtlinien für synchrones Design Takteingänge von Flip-Flops vergattert (d.h. das Takten wird von einer Bedingung abhängig gemacht), können Teststrategien wie Scan-Path nicht angewandt werden. Wie etwa im Schaltbild des Automaten in Punkt b) zu sehen ist, müssen zum Shiften des Scan-Schieberegisters alle Flip-Flops des Automaten mit jedem Takt getaktet werden. Die Nichtanwendbarkeit von Testmethoden wie Scan-Path ist somit ein weiterer Grund (abgesehen von der Clock Skew Problematik), warum das Vergattern von Taktleitungen in synchronen Designs nicht erlaubt ist.