6. SCHALTWERKE

6.1 Schaltwerk-Grundlagen

In diesem Kapitel geht es darum, Informationen, die wir nun in Flipflops bzw. Registern speichern können, dafür zu nutzen, **endliche Automaten** zu realisieren. Im Gegensatz zur Automatentheorie geht es in der Technischen Informatik weniger um die Analyse, sondern mehr um den zielgerichteten Entwurf und die günstige Realisierung von Automaten in Form von **Schaltwerken**, die eine gegebene Aufgabenstellung mit Hilfe von digitalen Systemen realisieren.

Schaltwerke sind nach DIN Funktionseinheiten zum Verarbeiten von Schaltvariablen, wobei der Wert am Ausgang zu einem bestimmten Zeitpunkt abhängt von den Werten am Eingang zu diesem und endlich vielen vorangegangenen Zeitpunkten.
Der Zustand am Ausgang zu einem bestimmten Zeitpunkt hängt also ab von einem inneren Zustand und dem Wert am Eingang.

Kennzeichnend für ein Schaltwerk ist, dass für mindestens eine Kombination von Eingangssignalen mehrere Ausgaben möglich sind.
Das wird durch die Rückkopplung Zustandsspeicher $\rightarrow$ Schaltnetz $\rightarrow$ Zustandsspeicher erreicht.

Synchrone Schaltwerke:
Ein unabhängiger Takt sorgt für eine sequentielle Arbeitsweise, indem der neue Zustand taktsynchron übernommen und für die nächste Taktperiode gespeichert wird.

Asynchrone Schaltwerke:
Unmittelbare Zustandsänderung bei Änderung der Eingangssignale (abhängig von Laufzeiten, aber nicht von einem synchronisierenden Takt)

Bezug zu endlichen Automaten

Die Beschreibung von Schaltwerken erfolgt durch endliche Automaten.

In der Automatentheorie ist ein endlicher Automat durch ein 5-Tupel gekennzeichnet (vgl. Informatik A), ein Schaltwerk durch:

Eingangsvektor	$\underline{X} = (x_1, x_2, \ldots, x_m), \underline{X} \in \{0,1\}^m$
Ausgabevektor	$\underline{Y} = (y_1, y_2, \ldots, y_r), \ \underline{Y} \in \{0,1\}^r$
Zustandsvektor	$\underline{Z} = (z_1, z_2, \ldots, z_p), \ \underline{Z} \in \{0,1\}^p$
Übergangsfunktion	$\underline{Z}^{n+1} = g(\underline{X}, \underline{Z})^n$
Ausgabefunktion	$\underline{Y}^n = f(\underline{X}, \underline{Z})^n$

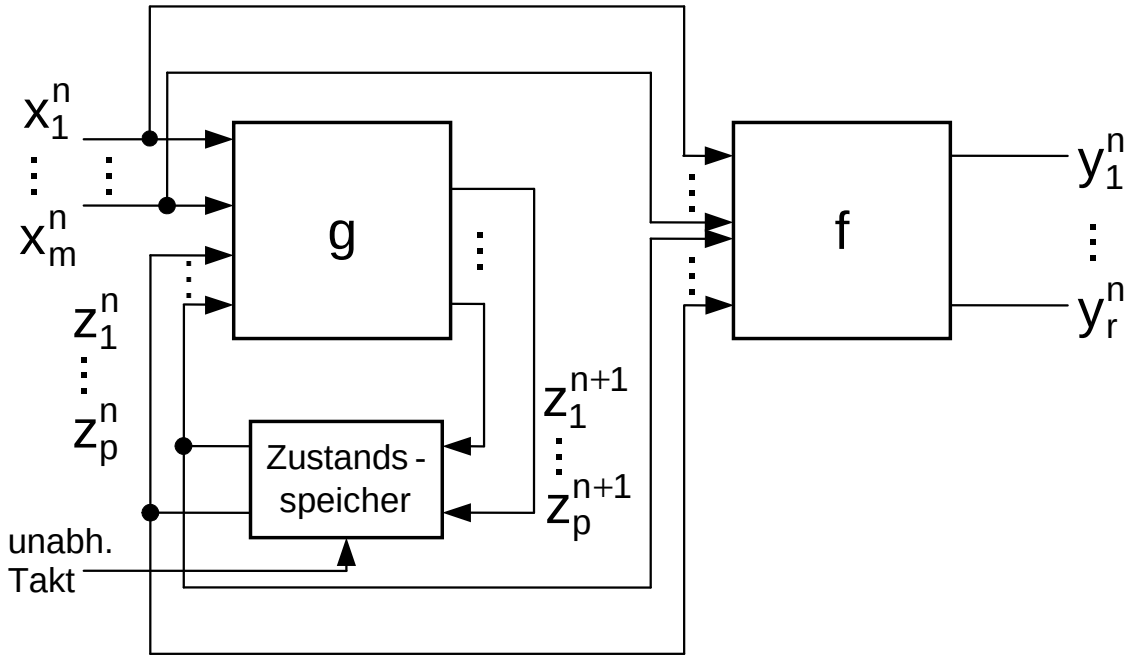
Endlicher Automat	Schaltwerk
Endliche Zustandsmenge *S*	Zustände *Z*
Endliches Eingabealphabet Σ	Eingangsvektor *X*
Überführungsfunktion δ	Übergangsfunktion g
Anfangs-/Startzustand $s_0 \in S$	Anfangs-/Startzustand
Endzustände $F \subseteq S$	
	Ausgabefunktion f

Jeder endliche Automat lässt sich in ein Schaltwerk umsetzen.

Die Ausgabe- und Übergangsfunktionen werden durch reine Schaltnetze realisiert, die mit den üblichen Methoden dargestellt und optimiert werden können.

Mealy-Automat (vgl. Schaltwerk-Definition nach Huffman)

reagiert als „Übergangsautomat“ quasi sofort auf Eingangssignaländerungen.

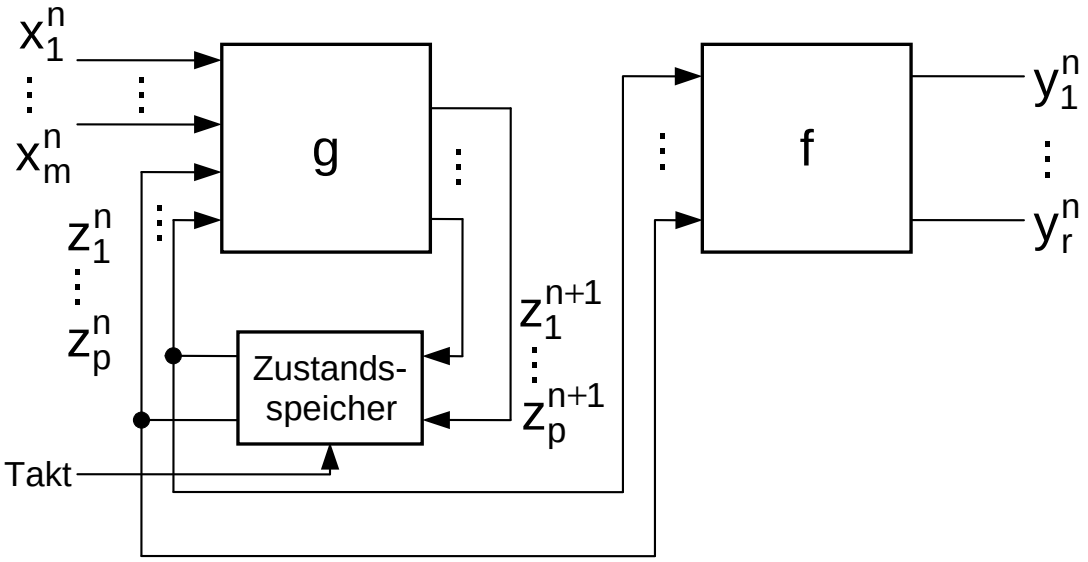


Ausgabefunktion: $\underline{Y}^n = f(\underline{X}, \underline{Z})^n$

Übergangsfunktion: $\underline{Z}^{n+1} = g(\underline{X}, \underline{Z})^n$

Moore-Automat

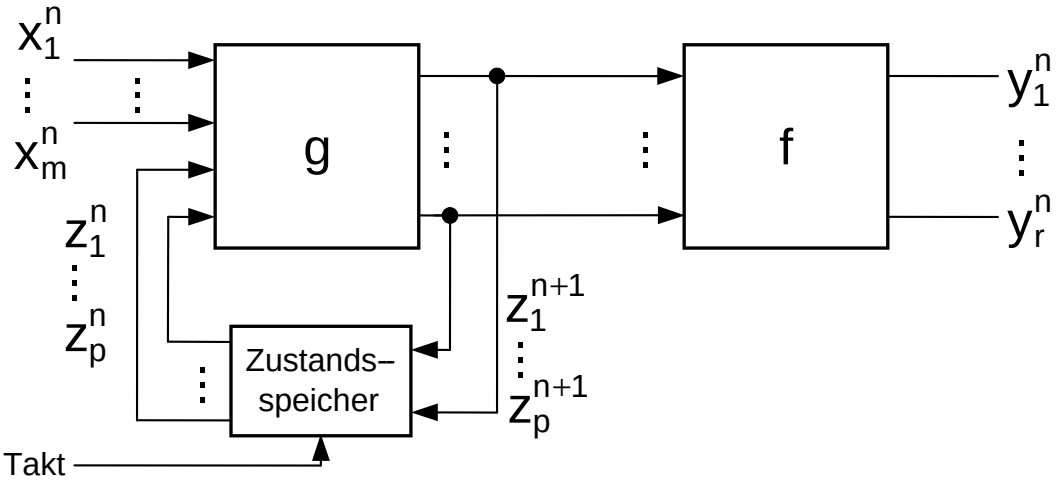
Quasi ein Sonderfall des Mealy-Automaten ist der Moore-Automat, bei dem der Ausgangsvektor nur vom Zustandsvektor abhängt. Er wird deshalb auch zustandsorientiertes Schaltwerk oder Zustandsautomat genannt.



Ausgabefunktion: $\underline{Y}^n = f(\underline{Z})^n$

Übergangsfunktion: $\underline{Z}^{n+1} = g(\underline{X}, \underline{Z})^n$

Sonderfall: Vorauslaufender Moore-Automat



Wichtig: Beim Mealy-Automat ist die Ausgabe abhängig von Zustand *und* Eingabe, beim Moore-Automat *nur* vom Zustand.

6.2 Funktionale Beschreibung

Übergangstabelle

Analog zu Schaltnetzen können die Übergangs- und Ausgabefunktion in Form von Wahrheitstafeln, hier (Zustands)Übergangstabellen genannt, dargestellt werden.

Es wird zu jedem Zeitpunkt **immer genau ein** Zustand angenommen (sequentielles Schaltwerk).

Für alle Eingangsvariablen und Zustände werden die Folgezustände und Ausgangsvariablen in einer Wahrheitstafel aufgeführt.

Oft gibt es einen ausgezeichneten Startzustand (oft 0).

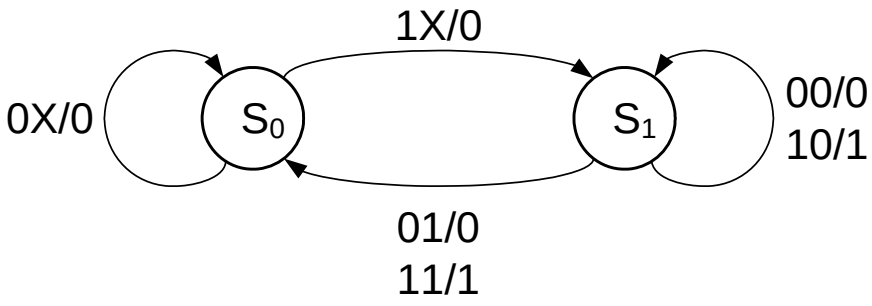
Die zugehörigen Schaltfunktionen (Ausgabefunktion, Übergangsfunktion) können dann mit den üblichen Methoden dargestellt und minimiert werden.

Zustands(übergangs)graph

Knoten:	Zustände des Schaltwerks
Kanten (gerichtet):	Zustandsübergänge
Kantenmarkierung: (Mealy-Automat)	Eingabe/Ausgabe (zum Zeitpunkt n)

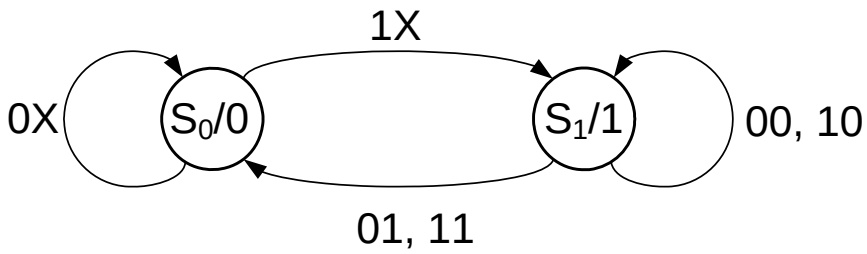
Von jedem Zustand gehen im Prinzip so viele Kanten ab, wie es mögliche Eingaben gibt.

Beispiel: für einen einfachen Mealy-Automat



(Kantenbeschriftung hier: x_1 x_2 / y)

Die Darstellung eines Moore-Automaten geschieht analog mit den Übergangsbedingungen an den Kanten, aber der Ausgabe als Beschriftung der Zustände.



(Kantenbeschriftung hier: x_1 x_2
Zustand: Name/y)

(Beachten: Die Darstellung hier ist analog zu oben, aber nicht die Funktion.)

Einfaches Beispielschaltwerk als Mealy-Automat

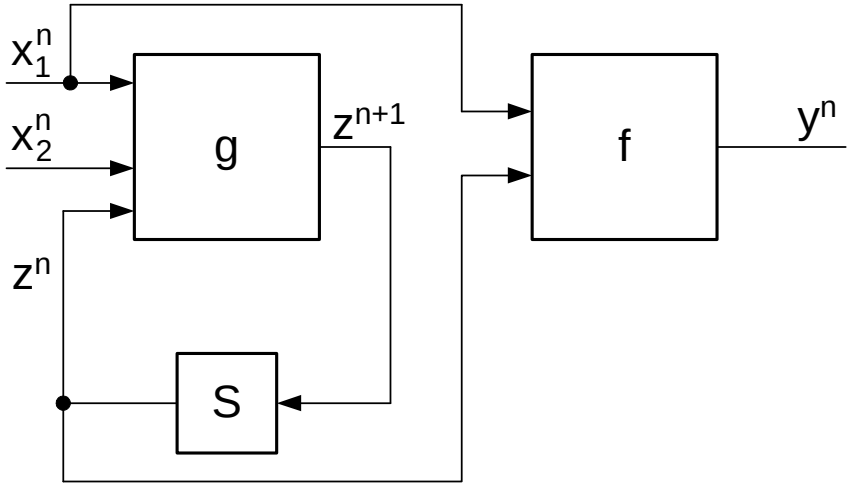
z^n	x_1^n	x_2^n	z^{n+1}	y^n
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	1

Übergangsfunktion:

$$z^{n+1} = \bar{z}^n x_1^n + z^n \bar{x}_2^n$$

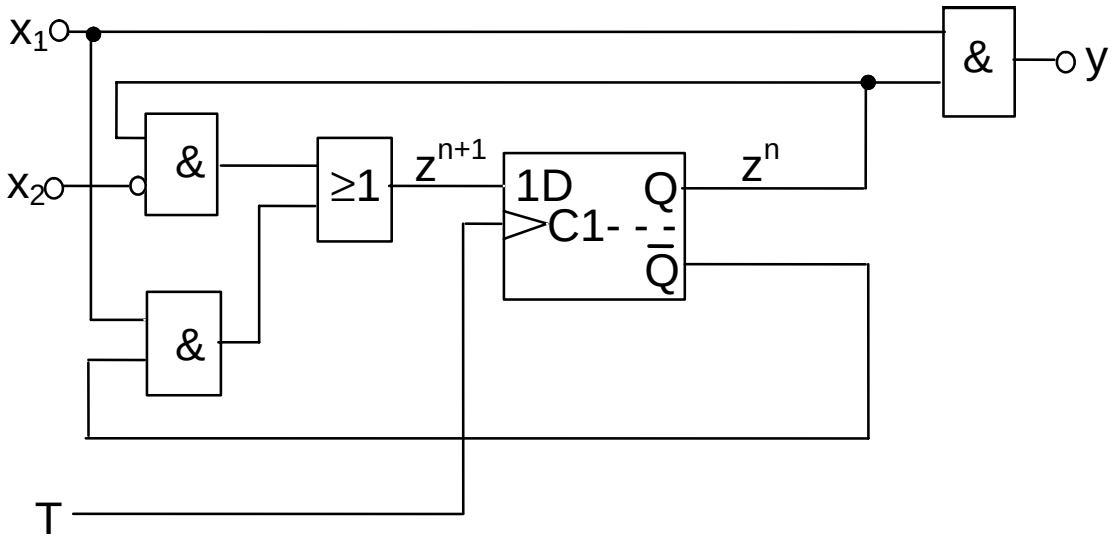
Ausgabefunktion:

$$y^n = z^n x_1^n$$



S: Speicher

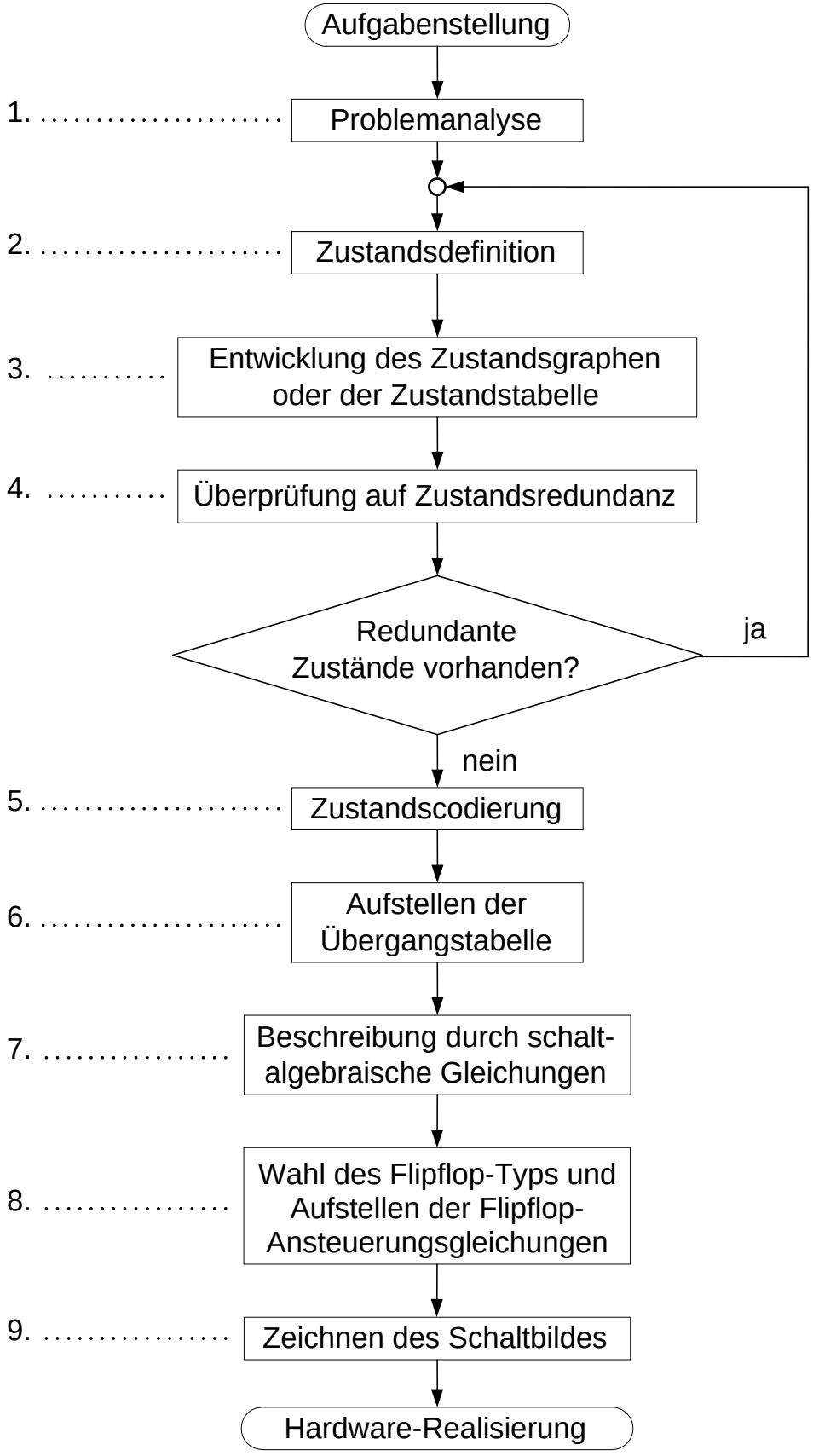
Beispiel: Realisierung mit D-Flipflop



6.3 Schaltwerksentwurf (Schaltwerksynthese)

6.3.1 Entwurfsablauf

a



Das Aufstellen des Zustandsgraphen oder der Zustandstabelle ist der kritischste Teil beim Entwurf von Schaltwerken. Bei komplexeren Schaltwerken ist daher ein formalisiertes Vorgehen angezeigt.

Insbesondere bei textuell gegebenen Aufgabenstellungen ist beim Entwurf als erstes ein Formalisierungsschritt sinnvoll. Weil dieser Schritt oft Schwierigkeiten bereitet, hier ein **Tipp zum Vorgehen**:

Analysiere den Text genau und leite daraus (iterativ) durch Herausschreiben die das Schaltwerk konstituierenden Größen ab:

a. was sind die ***Eingangsgröße(n)*** (vgl. Eingangsvektor X, Eingabealphabet Σ); welche (sinnvolle) Eingabemenge gibt es; gibt es don´t cares
b. was sind die ***Ausgabegröße(n)***
c. welche ***Zustände*** Z hat das System (vgl. Endliche Zustandsmenge S)
d. was ist der Anfangs-/*Startzustand* ($s_0 \in S$)
e. wie ist die ***Übergangsfunktion*** g (vgl. Überführungsfunktion δ); wie wird in welchem Zustand auf die Eingabemenge (alle möglichen Eingaben) reagiert
f. wie leitet sich die ***Ausgabefunktion*** f aus der Zustandsmenge (Moore-Automat) und/oder der Eingabemenge (Mealy-Automat) ab

Gerade bei der Bestimmung der Zustände gibt es vielfach große Freiheitsgrade. Hier sollte man auf eine möglichst kleine Zustandsmenge achten.

Die **Codierung der Zustände** sollte so erfolgen, dass die Übergangs- und/oder Ausgabefunktion möglichst einfach wird.

Um den Entwurf weiter zu verfeinern bzw. zu überprüfen, können noch folgende Schritte hilfreich sein:

I. Konstruiere einige typische Ein- und Ausgabesequenzen zur Veranschaulichung des Problems (bzw. der Aufgabenstellung).

II. Wähle den passenden Automatentyp (Mealy, Moore)

III. Bestimme, ob das Schaltwerk vollständig spezifiziert oder nur partiell definiert (Kap. 6.3.3) ist und unter welchen Bedingungen das Schaltwerk wieder in den Startzustand zurückkehrt.

IV. Wenn nur ein oder zwei Sequenzen zu einer Ausgabe ungleich 0 führen, sollten partielle Zustandsgraphen für die Sequenzen konstruiert werden.

V. Andernfalls können zunächst die Sequenzen(-gruppen) bestimmt werden, die sich das Schaltwerk merken muss. Die Zustände werden dann entsprechend aufgestellt.

VI. Jedes Mal, wenn eine neue Kante in den Zustandsgraph eingefügt wird, sollte überprüft werden, ob sie in einen bestehenden Zustand führen kann oder ob ein neuer Zustand eingeführt werden muss.

VII. Überprüfe den Zustandsgraphen, ob für alle 2^m möglichen Eingangskombinationen abgehende Kanten vorhanden sind und ob jeden Zustand nur genau eine Kante für jede Kombination von Eingangswerten verlässt.

VIII. Überprüfe den fertigen Zustandsgraphen auf korrekte Funktion (mindestens mit den Sequenzen aus Schritt I.).

Bei einem vollständig spezifizierten Zustandsgraphen gilt (**Konsistenzcheck**, vgl. VII.):

a) Die ODER-Verknüpfung aller Übergangsbedingungen der von einem Knoten abgehenden Kanten muss *wahr* sein. D.h., es gibt mindestens einen Nachfolgezustand (sonst Deadlock).

b) Die UND-Verknüpfung der Übergangsbedingungen zweier beliebiger von einem Knoten abgehenden Kanten darf *niemals wahr* sein. Dadurch wird sichergestellt, dass es nur einen Nachfolgezustand gibt (sonst Mehrdeutigkeit).

6.3.2 Beispiel für die Schaltwerkssynthese

Beispiel: Modulo-4-Zähler

(1) Aufgabenstellung (Spezifikation):

Entwurf eines Modulo-4 Vorwärts/Rückwärtszähler als Automat (mit Takt als Zählimpulse)

Problembeschreibung:

Der Zähler soll von 0 bis 3 zählen.

Dabei soll abhängig von dem Steuereingang x für x = 1 die Zahlenfolge 0-1-2-3 (Vorwärtszählen) und für x = 0 die Zahlenfolge 3-2-1-0 (Rückwärtszählen) durchlaufen und ausgegeben werden.

Bei Überschreiten der Zahl 3 ist auf 0 zurückzugehen bzw. bei Unterschreiten der 0 ist auf die Zahl 3 überzugehen (Ringzähler).

Am Ausgang ist der Zählerstand auszugeben (Ausgabevektor y_1y_0).

(2) Analyse

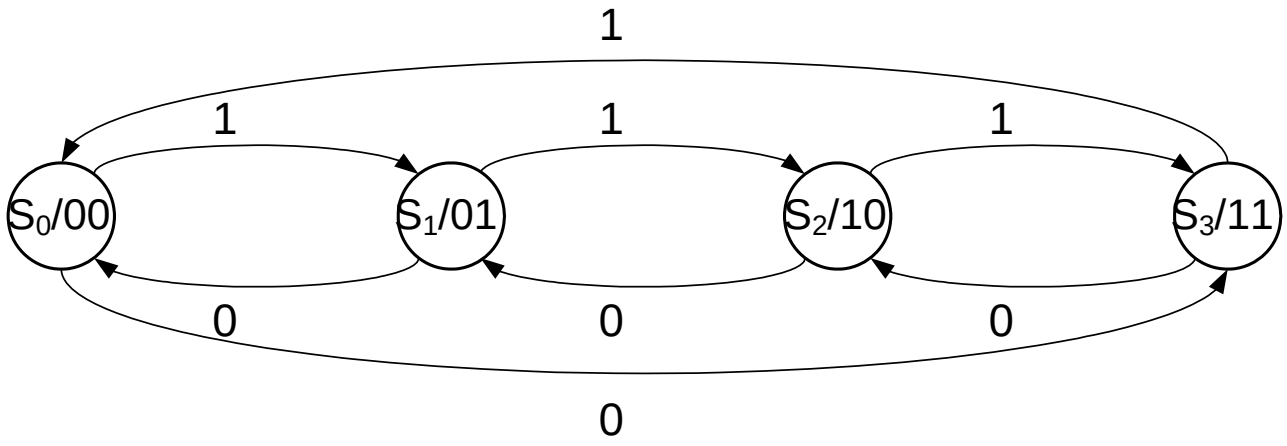
Eingangsgrößen: Steuervariable x

Ausgangsgröße: Zählerstand 0…3 (dual codiert: y_1y_0)

Zustandsdefinition: 4 Zustände: S_0, S_1, S_2, S_3

Automatentyp: Moore

(3) Entwicklung des Zustandsgraphen bzw. der Zustandstabelle:



mit: $\mathbf{x/Y_1Y_0}$

aktueller Zustand	aktuelle Ausgabe	Folgezustand	
	y_1y_0	x = 0	1
S_0	00	S_3	S_1
S_1	01	S_0	S_2
S_2	10	S_1	S_3
S_3	11	S_2	S_0

(4) Überprüfung auf Zustandsredundanz

- keine vorhanden -

(5) Zustandscodierung

Dualzahlen: 00, 01, 10, 11 für S_0, S_1, S_2, S_3

(6) Aufstellen der Übergangstabelle (codierte Zustände)

	x^n	Z_1^n	Z_0^n	Z_1^{n+1}	Z_0^{n+1}	Y_1^n	Y_0^n
Vorwärts-zählen	1	0	0	0	1	0	0
	1	0	1	1	0	0	1
	1	1	0	1	1	1	0
	1	1	1	0	0	1	1
Rückwärts-zählen	0	1	1	1	0	1	1
	0	1	0	0	1	1	0
	0	0	1	0	0	0	1
	0	0	0	1	1	0	0

(7) Schaltalgebraische Beschreibung der Ausgangs- und Übergangsfunktionen

Ausgangsfunktionen:

$$Y_0^n = Z_0^n$$
$$Y_1^n = Z_1^n$$

Hier ist also kein Ausgangsschaltnetz erforderlich.

Übergangsfunktionen:

$$Z_0^{n+1} = \left(x\overline{Z}_0\overline{Z}_1 + x\overline{Z}_0 Z_1 + \overline{x}\overline{Z}_0 Z_1 + \overline{x}\overline{Z}_0\overline{Z}_1\right)^n$$

$$Z_1^{n+1} = \left(\overline{x}\overline{Z}_0\overline{Z}_1 + \overline{x}Z_0 Z_1 + xZ_0\overline{Z}_1 + x\overline{Z}_0 Z_1\right)^n$$

Nach Minimierung:

$$Z_0^{n+1} = \overline{Z}_0^n$$

Z_1^{n+1} nicht weiter minimierbar

(8) Wahl des Flipflop-Typs und Ansteuergleichungen

Realisierung mit D-Flipflops

$$Q^{n+1} = D^n \text{ (charakteristische Gleichung)}$$

Für D-Flipflops ergeben sich durch Koeffizientenvergleich aus den Übergangsfunktionen unmittelbar die Ansteuergleichungen:

$$Q_0^{n+1} = D_0^n = Z_0^{n+1} = \overline{Z}_0^n$$

$$Q_1^{n+1} = D_1^n = Z_1^{n+1} = \left(\overline{x}\overline{Z}_0\overline{Z}_1 + \overline{x}Z_0Z_1 + xZ_0\overline{Z}_1 + x\overline{Z}_0Z_1\right)^n$$

Ansteuergleichungen:

$$D_0^n = \overline{Q}_0^n$$

$$D_1^n = \left(\overline{x}\overline{Q}_0\overline{Q}_1 + \overline{x}Q_0Q_1 + xQ_0\overline{Q}_1 + x\overline{Q}_0Q_1\right)^n$$

Ausgangsfunktionen:

$$Y_0^n = Q_0^n \qquad Y_1^n = Q_1^n$$

Realisierung mit JK-Flipflops

$$Q^{n+1} = J^n\overline{Q}^n + \overline{K}^nQ^n \text{ (charakteristische Gleichung)}$$

Umformung der Übergangsgleichungen in zwei Terme mit $\overline{Z}$ und Z, dann Koeffizientenvergleich

$$Q_0^{n+1} = J_0^n\overline{Q}_0^n + \overline{K}_0^nQ_0^n$$

$$Z_0^{n+1} = 1*\overline{Z}_0^n + 0*Z_0^n$$

$\Rightarrow J_0^n = 1,\ \overline{K}_0^n = 0 \Rightarrow$ $\mathbf{J_0^n = K_0^n = 1}$ Ansteuergleichung FF_0

$$Q_1^{n+1} = \quad J_1^n \ \overline{Q}_1^n + \quad \overline{K}_1^n \ Q_1^n$$

$$Z_1^{n+1} = (\overline{X}^n \overline{Z}_0^n + X^n Z_0^n)\overline{Z}_1^n + (\overline{X}^n Z_0^n + X^n \overline{Z}_0^n) Z_1^n$$

$$\Rightarrow J_1^n = (\overline{X}^n \overline{Q}_0^n + X^n Q_0^n),\ \overline{K}_1^n = (\overline{X}^n Q_0^n + X^n \overline{Q}_0^n),\ K_1^n = X^n Q_0^n + \overline{X}^n \overline{Q}_0^n$$

$$\Rightarrow \quad \mathbf{J_1^n = K_1^n = X^n Q_0^n + \overline{X}^n \overline{Q}_0^n}$$ Ansteuergleichung FF_1

Ein Koeffizientenvergleich ist für RS- oder T-Flipflops entsprechend möglich, aber schwieriger, da ggf. Zusatzbedingungen (z. B. RS = 0) eingehalten werden müssen bzw. explizit nach dem Wechsel von Zustandsbits gesucht werden muss.

Alternative Bestimmung der Ansteuergleichungen

Übergangstabelle mit FF-Eingängen

Alternativ zur algebraischen Methode mit Koeffizientenvergleich kann in der Übergangstabelle angegeben werden, welche Eingangsbelegungen den Zustandsübergang $Q^n \rightarrow Q^{n+1}$ bedingen.

Beispiel: Realisierung mit RS-Flipflops

Ansteuerungstabelle für ein RS-Flipflop (aus der Wahrheitstafel für das Setzen bzw. Rücksetzen abgeleitet)

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

- = don't care

Übergangstabelle mit Ansteuersignalen für RS-Eingänge

x	Z_1^n	Z_0^n	R_1^n	S_1^n	R_0^n	S_0^n	Z_1^{n+1}	Z_0^{n+1}	Y_1^n	Y_0^n
1	0	0	-	*0*	*0*	*1*	0	1	0	0
1	0	1	*0*	*1*	*1*	*0*	1	0	0	1
1	1	0	*0*	-	*0*	*1*	1	1	1	0
1	1	1	*1*	*0*	*1*	*0*	0	0	1	1
0	1	1	*0*	-	*1*	*0*	1	0	1	1
0	1	0	*1*	*0*	*0*	*1*	0	1	1	0
0	0	1	-	*0*	*1*	*0*	0	0	0	1
0	0	0	*0*	*1*	*0*	*1*	1	1	0	0

Spalten für R^n und S^n ergeben sich durch Anwendung der Ansteuerungstabelle für Wechsel von $Z^n \rightarrow Z^{n+1}$.

Ableitung der Ansteuerungsgleichungen für RS-FF durch Minimierung der Schaltfunktion für R und S (mit: $Q^n = Z^n$)

$$R_1^n = (xQ_1Q_0 + \bar{x}Q_1\bar{Q}_0)^n \qquad R_0^n = Q_0^n$$

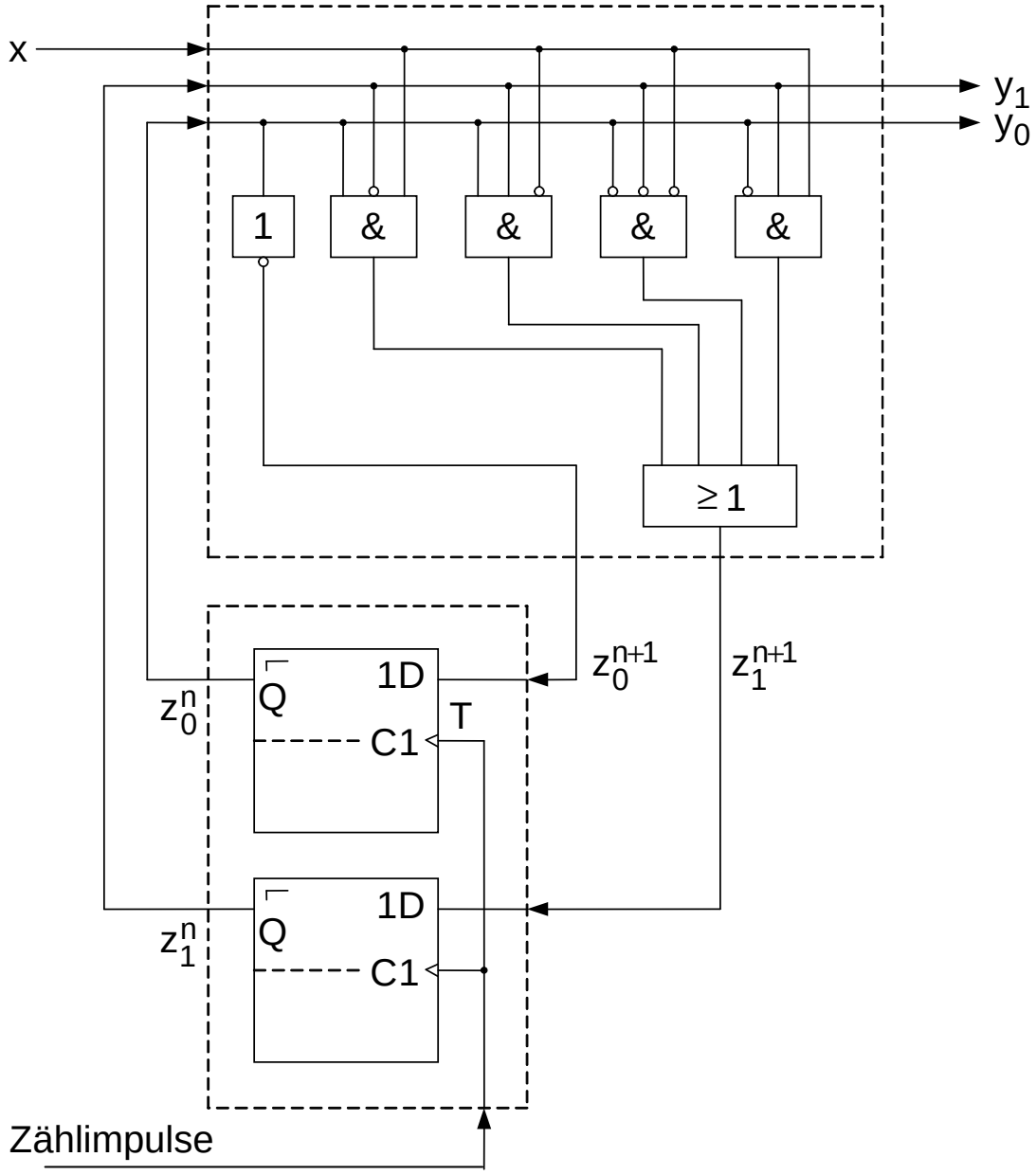
$$S_1^n = (\bar{x}\bar{Q}_1\bar{Q}_0 + x\bar{Q}_1Q_0)^n \qquad S_0^n = \bar{Q}_0^n$$

Für andere FF-Typen ganz analog, Ausgangsfunktionen siehe algebraische Methode.

Hinweis: Bei den Ansteuerungs- und Ausgangsgleichungen können bei der Minimierung natürlich auch *Koppelterme* genutzt werden (sofern vorhanden).

(9) Schaltbild des Modulo-4 Vorwärts-/Rückwärtszählers mit D-Flipflops

(x = 1: vorwärts zählen, x = 0: rückwärts zählen)



Obwohl evtl. Gatter eingespart werden könnten, dominieren heute in der Praxis D-Flipflops, da Schaltwerke mit ihnen besonders einfach zu realisieren sind.

Taktsteuerung: Master-Slave (Zweizustandssteuerung) oder Ein-Flankentriggerung

Hier: Takt = Zählimpuls!

6.3.3 Einsparung redundanter Zustände

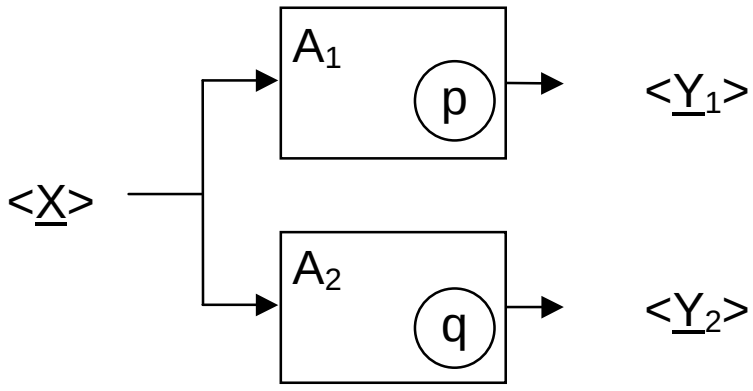
Der Aufwand zur Realisierung von Schaltwerken (Anzahl Flipflops, Anzahl Gatter) kann offenbar reduziert werden, wenn die Anzahl der Zustände minimiert wird.

Daher sollte das Schaltwerk auf *äquivalente Zustände* hin untersucht werden. Äquivalente Zustände können dann jeweils durch einen einzigen ersetzt werden.

Definition: Äquivalente Zustände

Seien A_1 und A_2 zwei (nicht unbedingt verschiedene) Schaltwerke und $< \underline{X} >$ eine Eingabesequenz beliebiger Länge.
Weiterhin sei $< Y_1 >= f_1(< \underline{X} >, p)$ die Ausgabesequenz von A_1 bei Eingabe der Sequenz $< \underline{X} >$ im Zustand p und $< Y_2 >= f_2(< \underline{X} >, q)$ die Ausgabesequenz von A_2 bei Eingabe von $< \underline{X} >$ im Zustand q.

Dann heißt Zustand $p = \underline{Z}_1$ in A_1 äquivalent zu Zustand $q = \underline{Z}_2$ in A_2 ($p \equiv q$) genau dann, wenn für die Ausgabesequenzen gilt $f_1(< \underline{X} >, p) = f_2(< \underline{X} >, q)$ für jede mögliche Eingabesequenz $< \underline{X} >$.



Für eine praktische Überprüfung ist diese Definition kaum geeignet, da potentiell unendlich lange Eingabefolgen untersucht werden müssen!

Satz:

Zwei Zustände p und q eines Schaltwerks sind genau dann äquivalent, d. h. ($p \equiv q$), wenn für jede *einzelne* Eingabe $\underline{X}$

(a) die Ausgaben *identisch* sind, d. h.

$$f(\underline{X},p) = f(\underline{X},q) \quad \textbf{und}$$

(b) die Folgezustände *äquivalent* sind, d. h.

$$g(\underline{X},p) \equiv g(\underline{X},q).$$

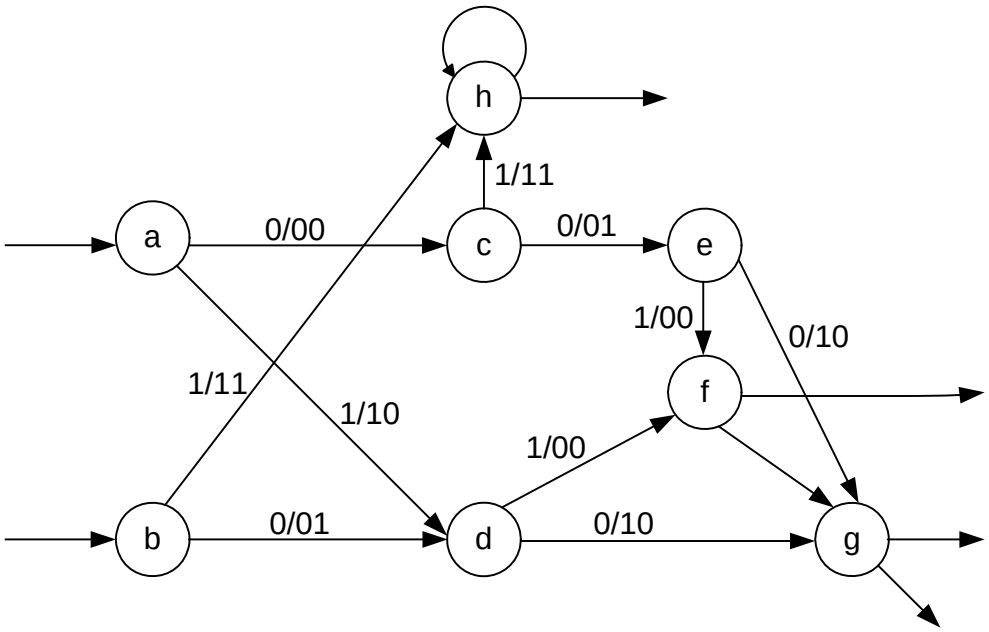
Die Überprüfung auf äquivalente Zustände reduziert sich damit auf einzelne Eingaben statt Eingabesequenzen!

Wichtiger Spezialfall:

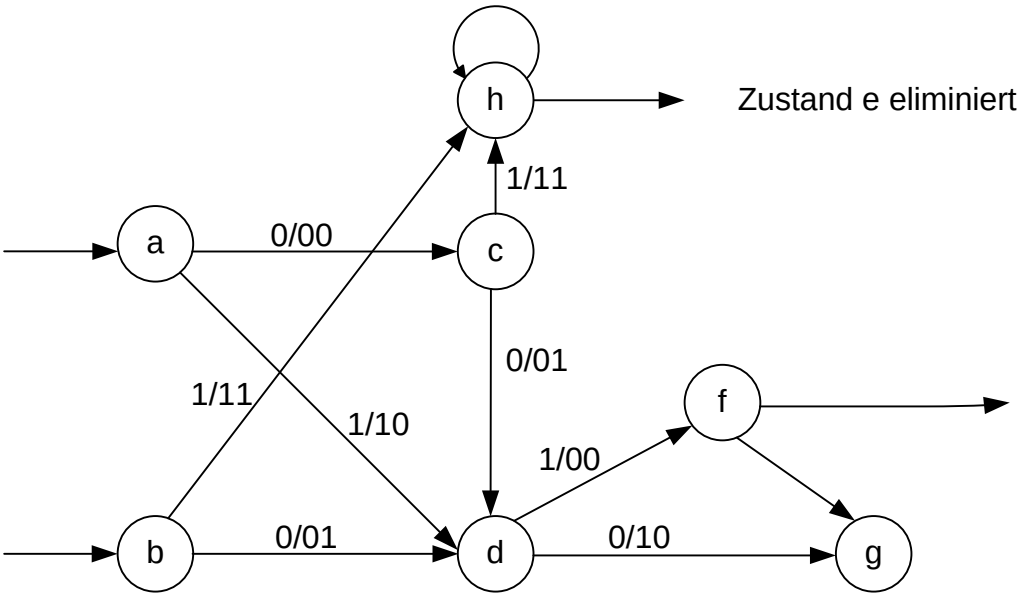
Die Folgezustände sind *identisch*, d. h. $g(\underline{X},p) = g(\underline{X},q)$.

Die Zustandsäquivalenz kann dann leicht (**rekursiv**) anhand der Zustandstabelle (Vergleich der Zeilen) oder des Zustandsgraphen ermittelt werden.

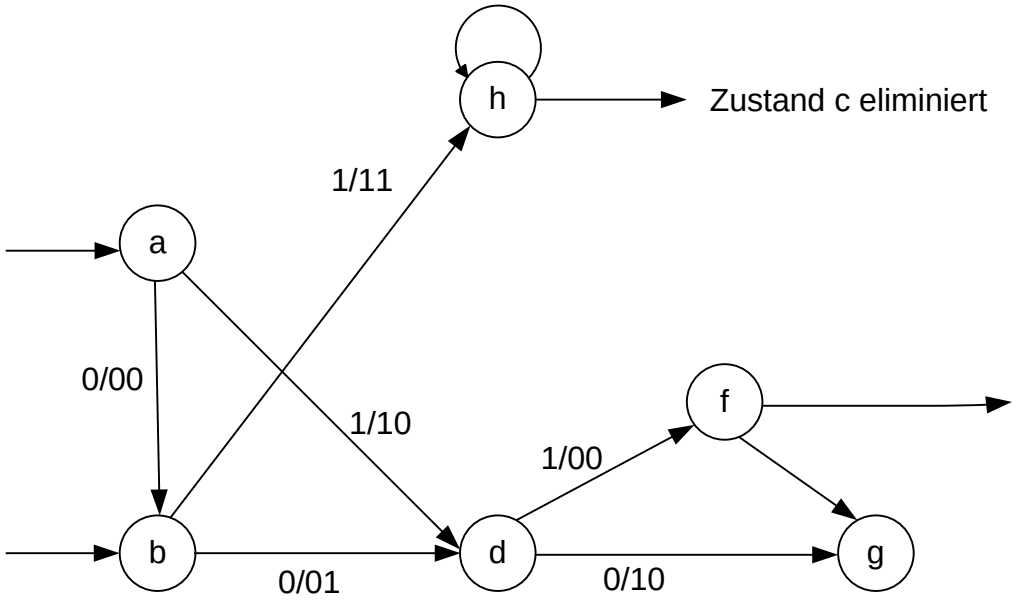
Beispiel:



´e´ und ´d´ sind äquivalent:



´b´ und ´c´ sind äquivalent:



Um *alle* äquivalenten Zustände zu finden, müssen alle möglichen Zustandspaarungen rekursiv bzgl. der Ausgaben und Übergangsbedingungen in die Folgezustände untersucht werden!

Tabellenmethode

Beispiel: (Vereinfachte Zustandsdarstellung)

Aktueller Zustand	Folgezustand		Ausgabe	
	X = 0	X = 1	X = 0	X = 1
a	d	e	1	0
b	d	f	0	0
c	b	e	1	0
d	b	f	0	0
e	f	c	1	0
f	c	b	0	0

Vorgehensweise:

Es werden die Zustände in Gruppen zusammengefasst, die im aktuellen Zustand die gleiche Ausgabe erzeugen.

hier: a-c-e, b-d-f

Für jeden Zustand einer Gruppe werden die Folgezustände für alle möglichen Eingaben untersucht.

hier: a-c-e → d-b-f für X = 0 und
e-e-c für X = 1

b-d-f → d-b-c für X = 0 und
f-f-b für X = 1

Falls alle Folgezustände jeweils nur zu einer der ursprünglichen Gruppen gehören, bleiben deren Zustände in den Gruppen (hier: d-b-f und e-e-c für a-c-e sowie f-f-b für b-d-f für X = 1), weil die Folgezustände wieder die gleiche Ausgabe erzeugen.

Ist ein Zustand nicht in den Ausgangsgruppen enthalten (hier: die c in d-b-c für b-d-f und X=0), weicht die Ausgabe des Folgezustands ab. Daher wird die Gruppe (hier: b-d-f) entsprechend der Ausgaben in zwei Untergruppen (hier in b-d, f) aufgeteilt.

Das ergibt hier die Gruppen: a-c-e, b-d, f

Das Verfahren wird wiederholt, bis sich keine Änderungen mehr ergeben.

Hier würde im nächsten Durchlauf a-c-e in die Gruppen a-c und e aufgeteilt, weil für X=0 d-b und f in verschiedenen Gruppen stehen.

Dann terminiert das Verfahren und es ergibt sich folgende reduzierte Zustandstabelle:

Aktueller	Folgezustand		Ausgabe	
Zustand	X = 0	X = 1	X = 0	X = 1
a-c	b-d	e	1	0
b-d	b-d	f	0	0
e	f	a-c	1	0
f	a-c	b-d	0	0

Die Zustände a und c sowie b und d können also zusammengefasst werden, ohne das Verhalten zu verändern.

Implikationstafelverfahren zur Ermittlung *aller* äquivalenten Zustände

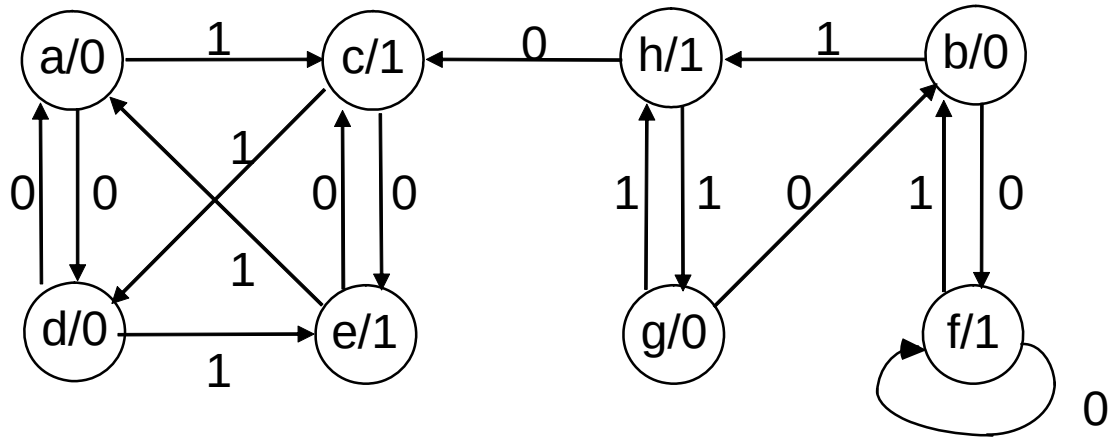
Das Implikationstafelverfahren dient der vollständigen Erfassung aller Zustandskombinationen und der Kombination aller Folgezustände in einer Zustandstabelle in einer einzigen anschaulichen Darstellung. Jeder Kombination entspricht genau ein Feld im Diagramm.

Beispiel (Moore-Automat):

Zustandstabelle:

Z^n	Z^{n+1}		Y^n
	$X^n = 0$	1	
a	d	c	0
b	f	h	0
c	e	d	1
d	a	e	0
e	c	a	1
f	f	b	1
g	b	h	0
h	c	g	1

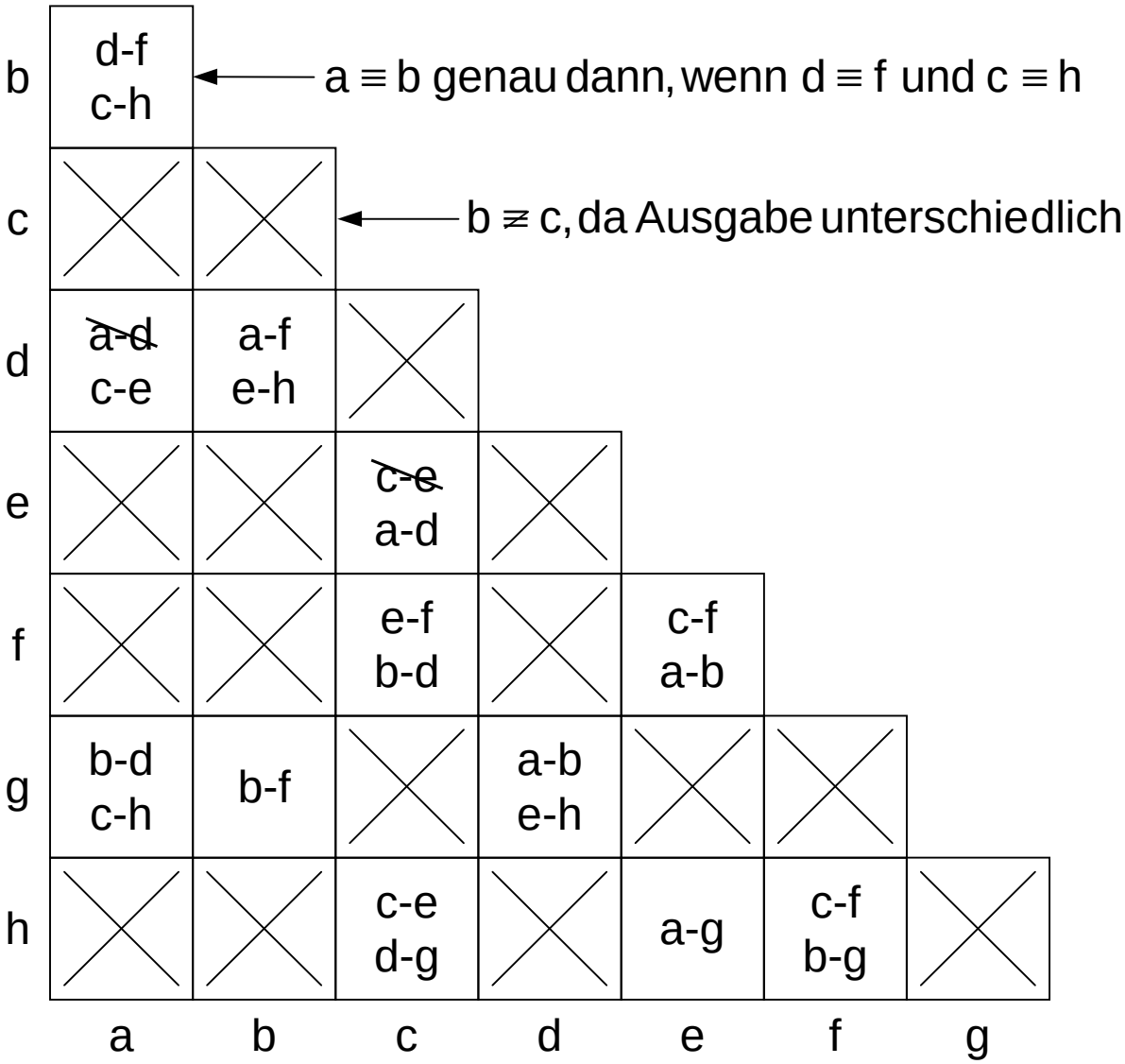
Zustandsgraph:



Beschriftung der Knoten: Zustand/Ausgabe
Beschriftung der Kanten: Eingabe

Bestimmung der äquivalenten Zustände

(1) Konstruieren der sog. Implikationstafel, die für jedes Paar von Zuständen ein Feld enthält.



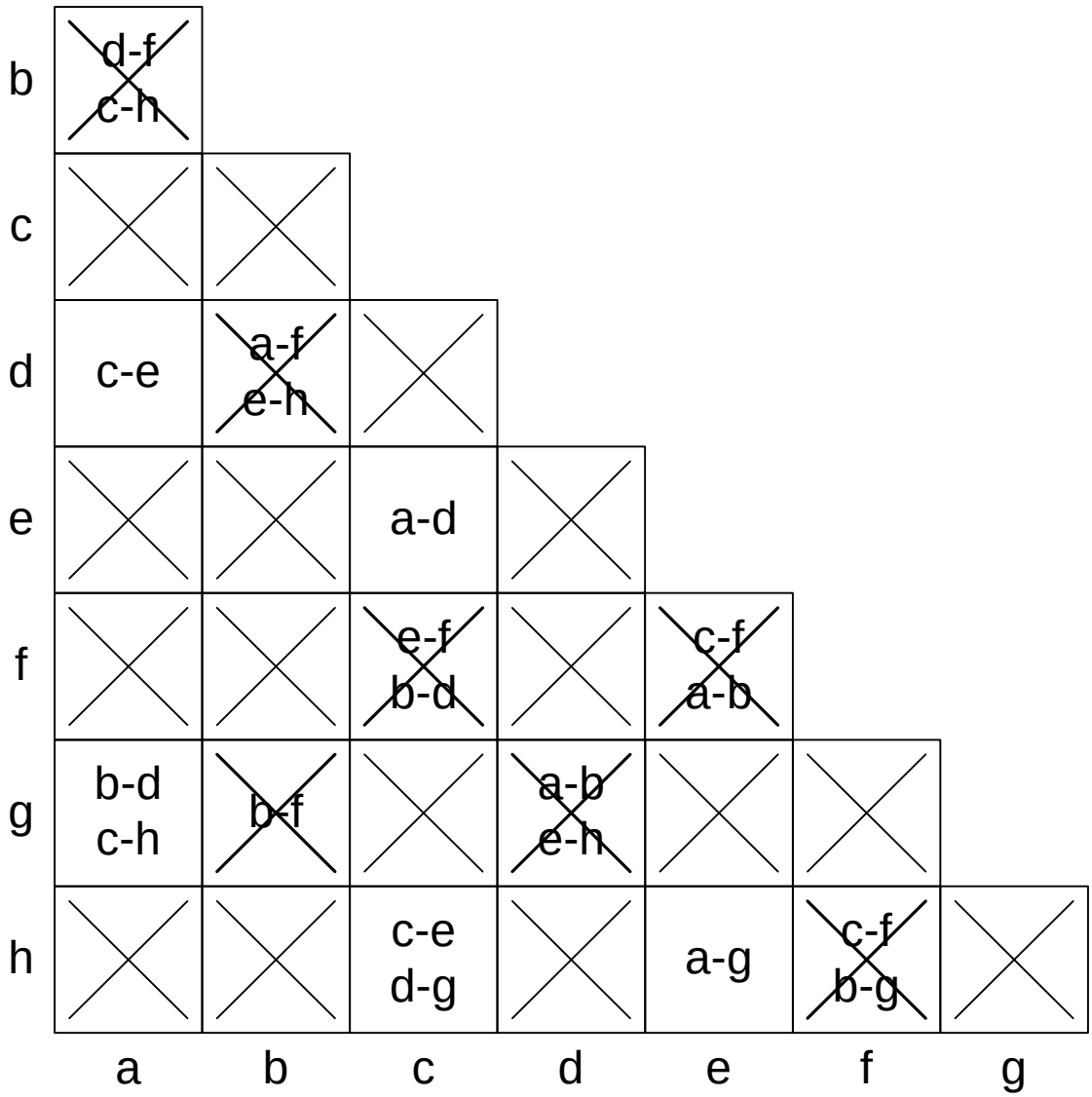
(2) Vergleiche jedes Paar von Zeilen der Zustandstabelle. Sind die Ausgaben zweier Zustände i und j *verschieden*, schreibe ein **X** in das Feld (i, j), da nicht i ≡ j.

Sind die Ausgaben *identisch*, schreibe alle sog. *implizierten Paare* in das Feld (i, j). Ein impliziertes Paar m-n liegt dann vor, wenn m-n die Folgezustände von i-j für eine Eingabe X sind.

(3) Überprüfung der Äquivalenz der Folgezustände:

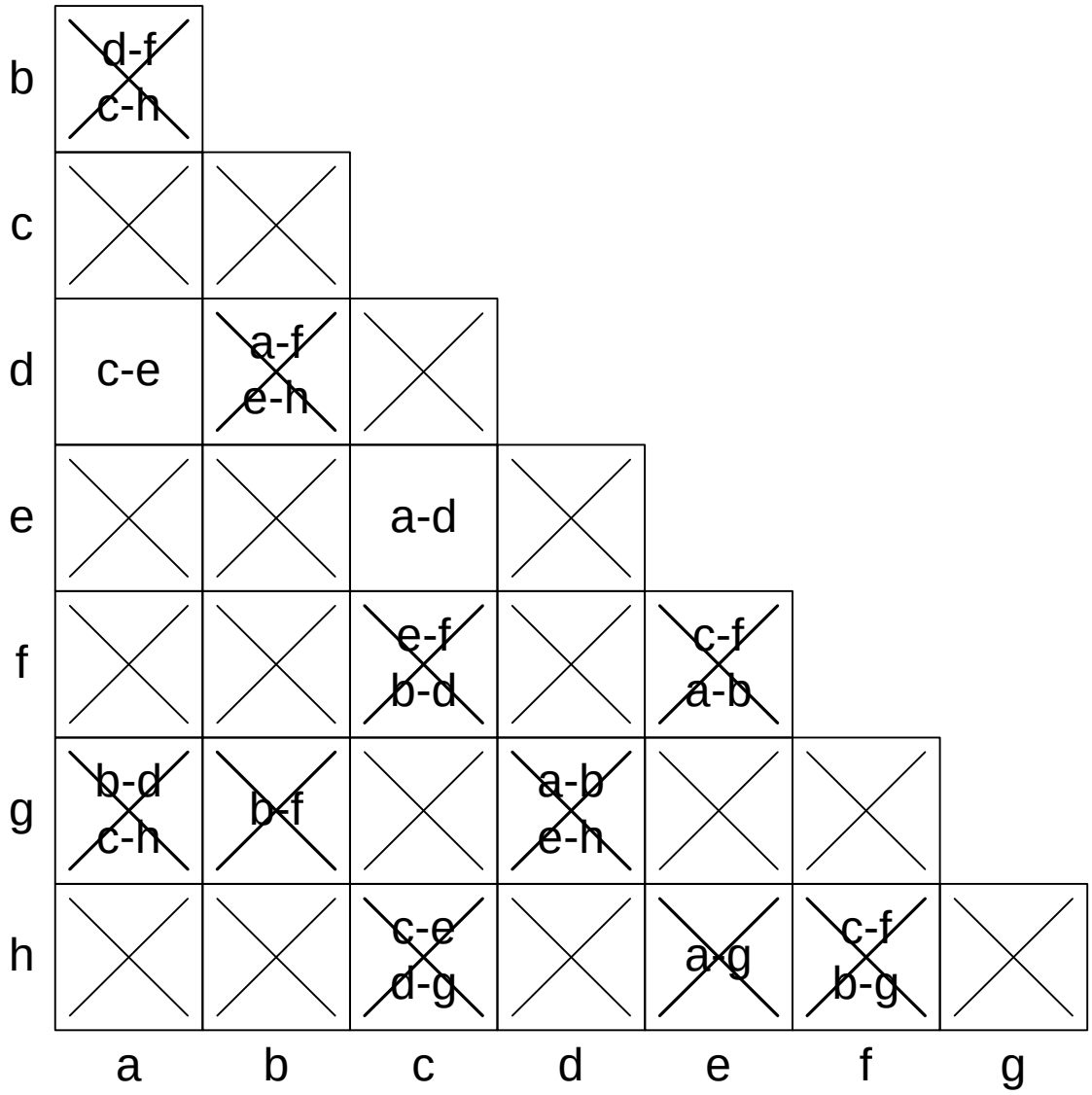
Gehe spaltenweise durch die Implikationstafel.

Falls das Feld (i, j) mindestens ein impliziertes Paar m-n enthält und in dem Feld (m, n) ein **X** steht, dann gilt **nicht** i ≡ j (ungleiche Folgezustände) und es wird ein **X** in Feld (i, j) geschrieben.



(4) Überprüfung der Äquivalenz weiterer Folgezustände:

Wiederhole Schritt (3) solange, wie ein neues **X** hinzukommt.



(5) Für jedes Feld (i, j) der Implikationstafel, das kein **X** enthält, gilt schließlich i ≡ j. D.h., die Zustände i und j sind äquivalent.

➔ Die redundanten Zustände können nun entfernt werden.

Zustandstabelle

nach Entfernen der redundanten Zustände d und e (a ≡ d und c ≡ e)

Z^n	Z^{n+1}		Y^n
	$X^n = 0$	1	
a	a	c	0
b	f	h	0
c	c	a	1
f	f	b	1
g	b	h	0
h	c	g	1

Das Implikationstafelverfahren ist für Mealy-Automaten ganz analog anwendbar.

6.3.4 Partiell definierte Zustandstabellen

Können einige Eingangsvektoren eines Schaltwerks nicht auftreten, bleiben die zugehörigen Folgezustände und Ausgaben undefiniert *(don't care)*, d. h. die Übergangsfunktion g bzw. die Ausgabefunktion f werden zu partiellen Funktionen.

Weiterhin ist es möglich, dass eine nachfolgende Schaltung nicht alle Ausgaben in allen Zuständen gleichzeitig auswertet, was ebenfalls zu 'don't cares' führt.

Sind don't care-Einträge (nicht vorkommende oder unrelevante Eingaben, Zustände, Ausgaben) in der Zustandstabelle enthalten, können sie so gesetzt werden, wie es für die Zustandsminimierung günstig ist (vgl. Minimierung von partiellen Schaltfunktionen).

Das Finden von optimalen Ersetzungen von don´t cares ist i.d.R. nicht trivial, so dass systematisch arbeitende Verfahren angewendet werden.

Anmerkung:
Eine Zustandsminimierung spart nicht unbedingt Flipflops (Überschreiten einer Zweierpotenzgrenze) oder Gatter ein.

Sie kann sich sogar negativ auswirken, weil z.B. die Übergangs- und/oder Ausgabefunktionen komplexer werden.

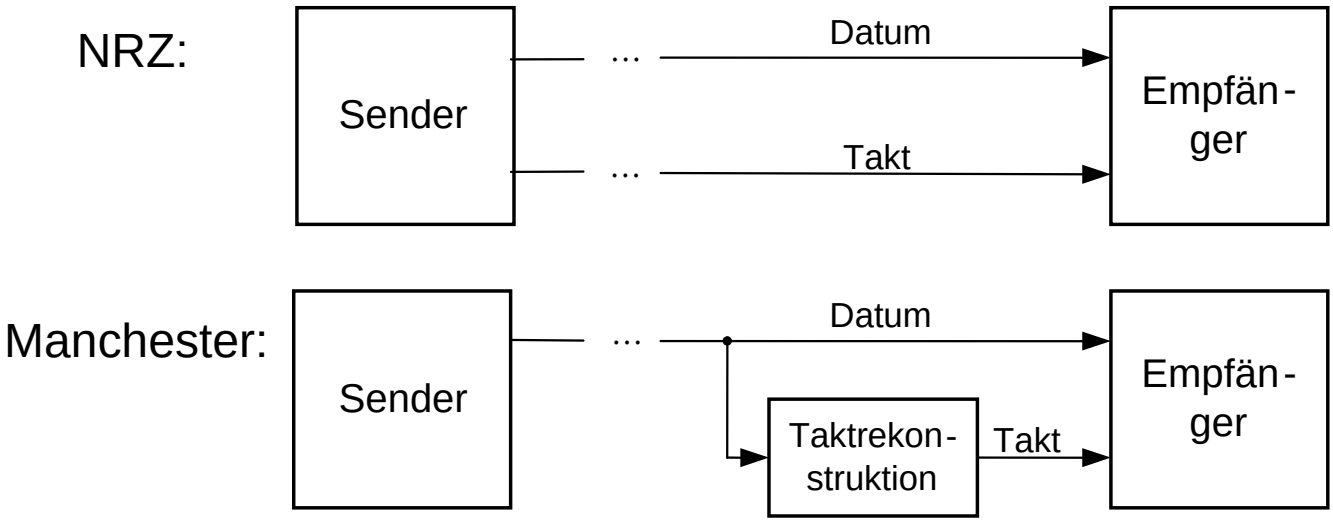
Eine geschickte Wahl der Zustandscodierung hat in der Praxis oft mehr Einfluss auf eine Reduktion des Hardwareaufwands.

6.3.5 Weitere Schaltwerkssynthese-Beispiele

Beispiel: Serielle Datenwandlung

(1) Aufgabenstellung: NRZ-Manchester-Konverter

Der Konverter soll einen NRZ-codierten, bitseriellen Datenstrom X mittels eines synchronen Schaltwerks in einen Manchester-codierten, seriellen Datenstrom Y umwandeln. Weil aus einem Mancherster-codierten Datenstrom der Takt rekonstruiert werden kann, braucht der Takt (zur Abtastung der Bitwerte) dann nicht mehr explizit übertragen zu werden.

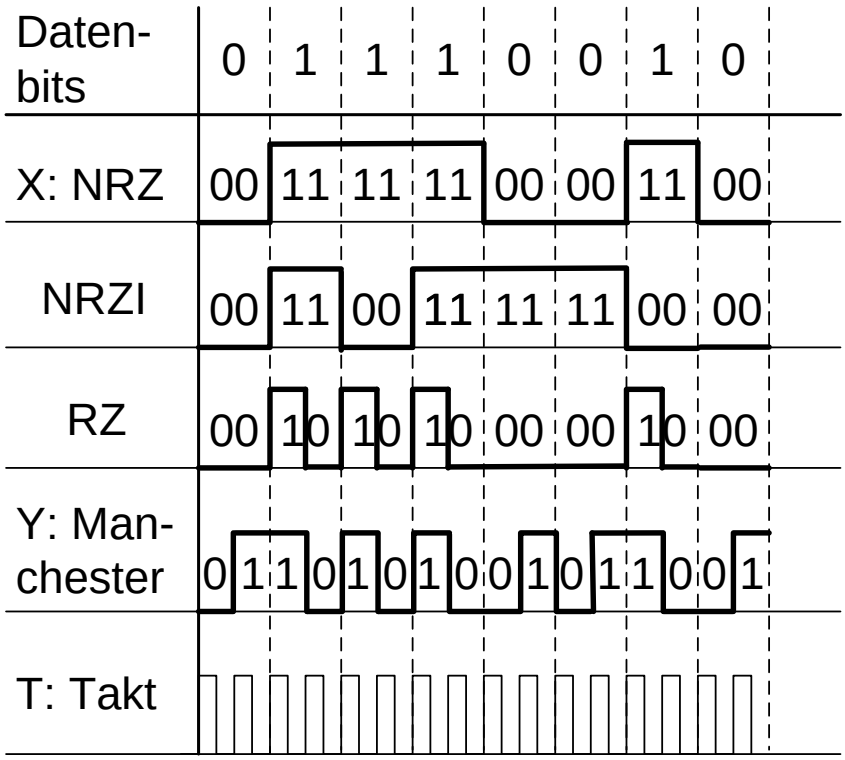


Für die Umwandlung und die Rekonstruktion steht ein Takt T mit der doppelten Frequenz der Datenbits zur Verfügung.

Anwendungen:

- serielle Datenübertragung (z. B. Ethernet, Funk)
- serielle Speicherung von Daten (z. B. Platten, Bänder, CD, DVD)

Codierung serieller Datenströme



NRZ-Code: (non-return-to-zero) Übertragung ohne Änderung, d.h. 0-Pegel für Datenbit “0”, 1-Pegel für “1”

NRZI-Code: (NRZ-inverted) Signalwechsel bei Datenbit “1”, kein Signalwechsel bei Datenbit “0”

RZ-Code: (return-to-zero) Signalflanke bei Datenbit “1”, keine Signalflanke bei Datenbit “0”

Vorteil: Signalwechsel einfacher zu erkennen

Nachteil: separater Takt bei der Datenübertragung für Abtastung erforderlich

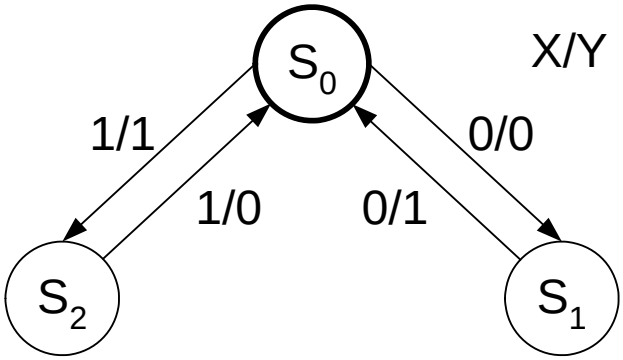
Manchester-Code: 0 → 1-Übergang **in der Mitte** für Datenbit “0”
1 → 0-Übergang **in der Mitte** für Datenbit “1”

Vorteil: Bitwechsel für jedes Datenbit, dadurch kann der Takt aus dem übertragenen Signal im Empfänger rekonstruiert und muss folglich nicht separat übertragen werden; mittelwertfrei

Nachteil: doppelte Übertragungsbandbreite erforderlich

Realisierung als Mealy-Automat

(2) (3) Zustandsdefinition und Zustandsgraph



3 Zustände: S_0, S_1, S_2, Startzustand: S_0

Bem.: In der Eingabesequenz ⟨X⟩ treten wegen des doppelten Taktes immer Paare 00 oder 11 auf (siehe Impulsdiagramm). Daher sind nicht alle Eingaben und alle Folgezustände definiert, d. h. es handelt sich um einen *partiell definierten* Automaten.

(4) Überprüfung auf Zustandsredundanz

- keine vorhanden -

(5) Zustandscodierung

Dualzahlen: S_0: 00, S_1: 01, S_2: 10

(6) Aufstellen der Zustandstabelle

X^n	Z_1^n	Z_0^n	Z_1^{n+1}	Z_0^{n+1}	Y^n
0	0	0	0	1	0
0	0	1	0	0	1
0	1	0	-	-	-
0	1	1	-	-	-
1	0	0	1	0	1
1	0	1	-	-	-
1	1	0	0	0	0
1	1	1	-	-	-

- = Zustand existiert nicht (don't care)

(7) Schaltalgebraische Beschreibung der Ausgangs- und Übergangsfunktion

Ausgangsfunktion:
(minimiert unter Nutzung von don't care-Termen)

$$Y^n = Z_0^n + \overline{Z}_1^n X^n$$

Übergangsfunktion:
(minimiert unter Nutzung von don't care-Termen)

$$Z_1^{n+1} = \overline{Z}_1^n X^n$$

$$Z_0^{n+1} = \overline{Z}_0^n \overline{X}^n$$

(8) Wahl des Flipflop-Typs:

D-Flipflops

Charakteristische Gleichung

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

und damit die Ansteuergleichungen

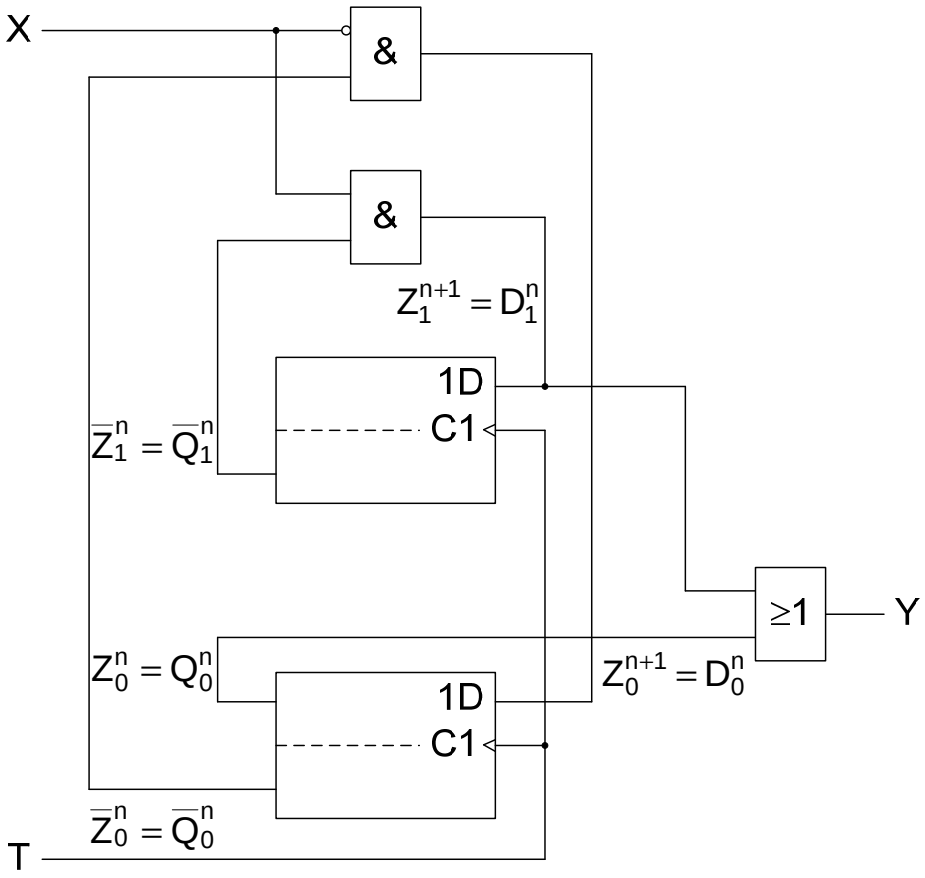
$$D_1^n = \overline{Q}_1^n X^n$$

$$D_0^n = \overline{Q}_0^n \overline{X}^n$$

Ausgangsfunktion:

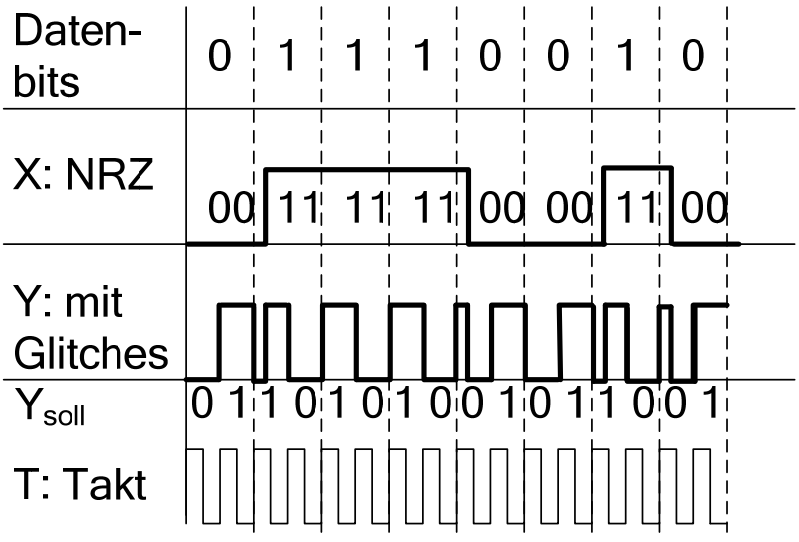
$$Y^n = Q_0^n + \overline{Q}_1^n X^n$$

(9) Schaltbild des NRZ-Manchester-Konverters



Koppelterm $\overline{Z}_1^n X^n$ wird genutzt.

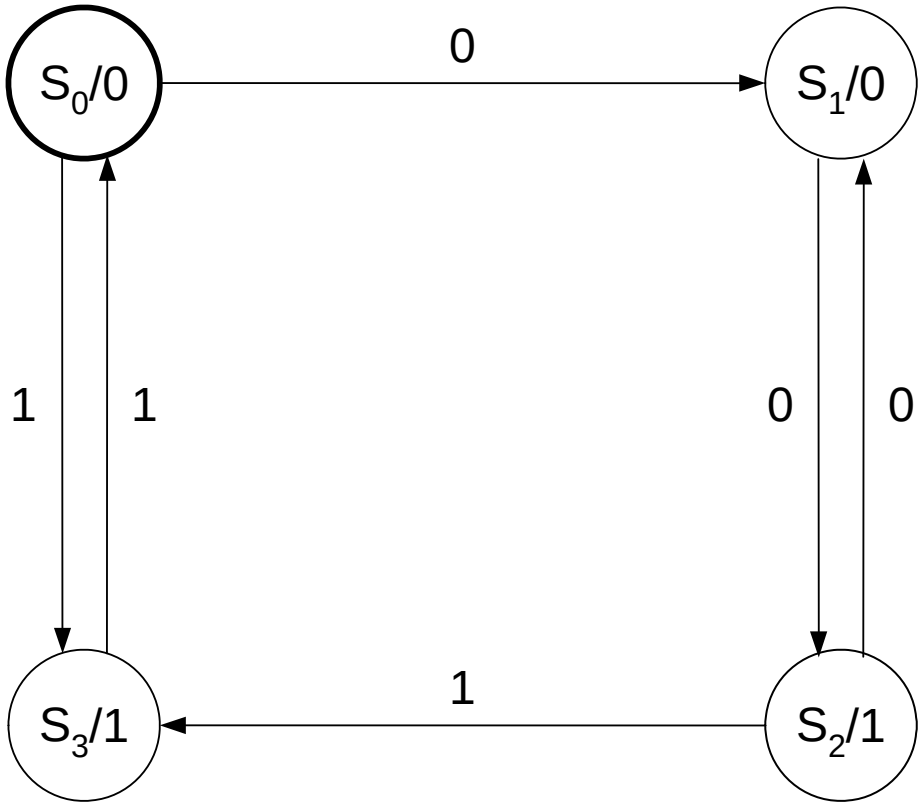
<u>Problem:</u> Relativ zum Takt verschobene Flanken am Eingang X können beim **Mealy**-Automat zu kurzen fehlerhaften Signalen (‘Glitches‘) am Ausgang Y führen, weil Eingangssignale quasi „durchgereicht“ werden!



Realisierung als Moore-Automat

Ein Moore-Automat ist hier vorteilhaft, weil die Ausgangsgrößen nur von den Zustandsgrößen abhängen und diese sich synchron mit dem Takt ändern.

(2) (3) Zustandsdefinition und Zustandsgraph



(4) Zustandsredundanzen: keine

(5) Zustandscodierung

S_0: 00, S_1: 01, S_2: 10, S_3: 11

(sinnvollerweise so, dass die Ausgabe einem Zustandsbit entspricht)

(6) Übergangstabelle

X^n	Z_1^n	Z_0^n	Z_1^{n+1}	Z_0^{n+1}	Y^n
0	0	0	0	1	0
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	-	-	1
1	0	0	1	1	0
1	0	1	-	-	0
1	1	0	1	1	1
1	1	1	0	0	1

(7) Ausgangs- und Übergangsfunktionen

$$Y^n = Z_1^n$$

$$Z_1^{n+1} = \overline{Z}_0^n X^n + \overline{Z}_1^n Z_0^n$$

$$Z_0^{n+1} = \overline{Z}_0^n$$

(8) Flipflop-Typ: D-Flipflops

Ansteuergleichungen

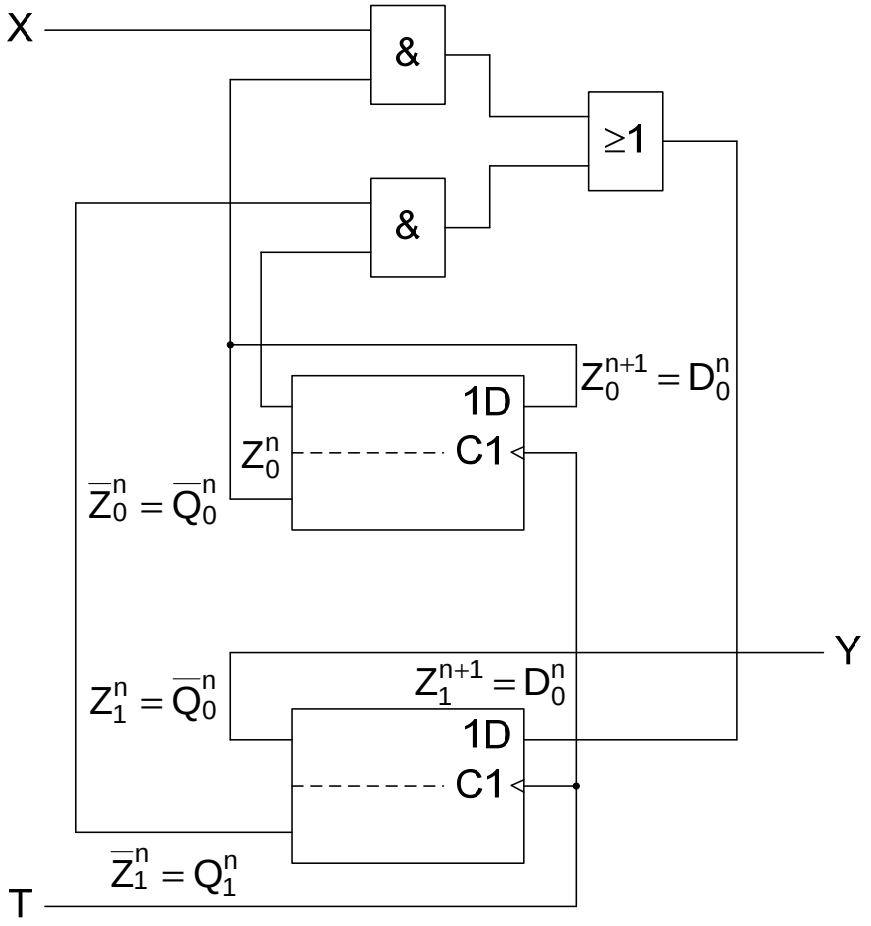
$$D_1^n = \overline{Q}_0^n X^n + \overline{Q}_1^n Q_0^n$$

$$D_0^n = \overline{Q}_0^n$$

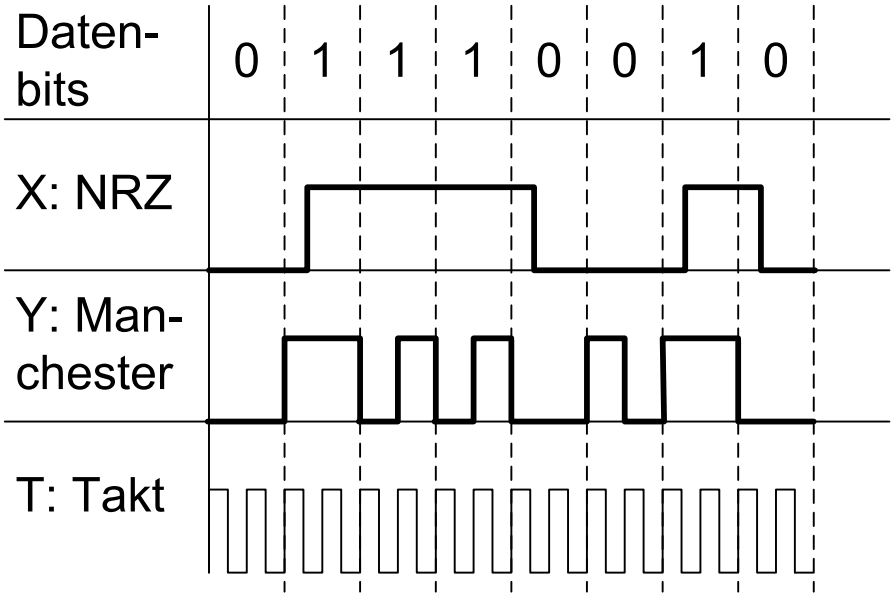
Ausgangsgleichung

$$Y^n = Q_1^n$$

(9) Schaltbild



Impulsdiagramm bei verschobenem X:



Die Ausgabe hat keine ‘Glitches‘, ist aber um eine Taktphase verzögert!

Beispiel: Detektor für positive Flanken

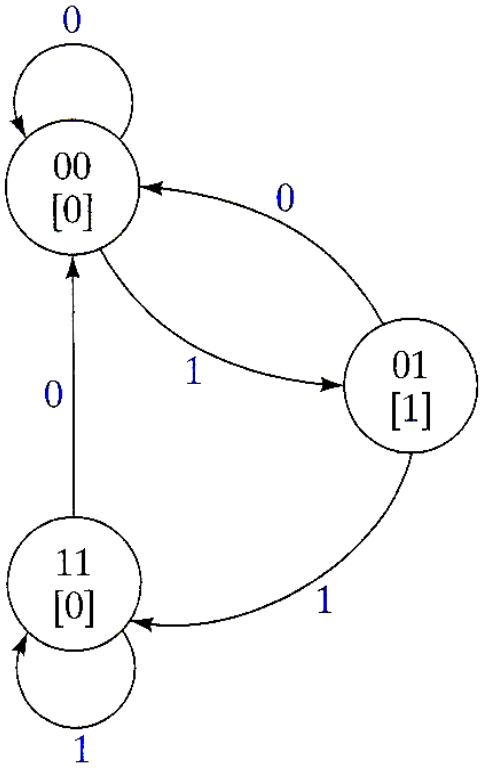
(1) Aufgabenstellung

In einem seriellen Bitstrom sollen positive Flanken erkannt werden, d. h. eine 0 gefolgt von einer 1.

(2) Zustandsdefinition

3 Zustände:
- mehrere Nullen folgen aufeinander
- mehrere Einsen folgen aufeinander
- eine Null wird von einer 1 gefolgt

(3) Zustandsgraph und Übergangstabelle (Moore)



X^n	Z_1^n	Z_0^n	Z_1^{n+1}	Z_0^{n+1}	Y^n
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	-	-	-
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	-	-	-
1	1	1	1	1	0

(4) Überprüfung auf Zustandsredundanz

- keine vorhanden –

(5) Übergangs- und Ausgangsfunktionen

Übergangfunktionen:

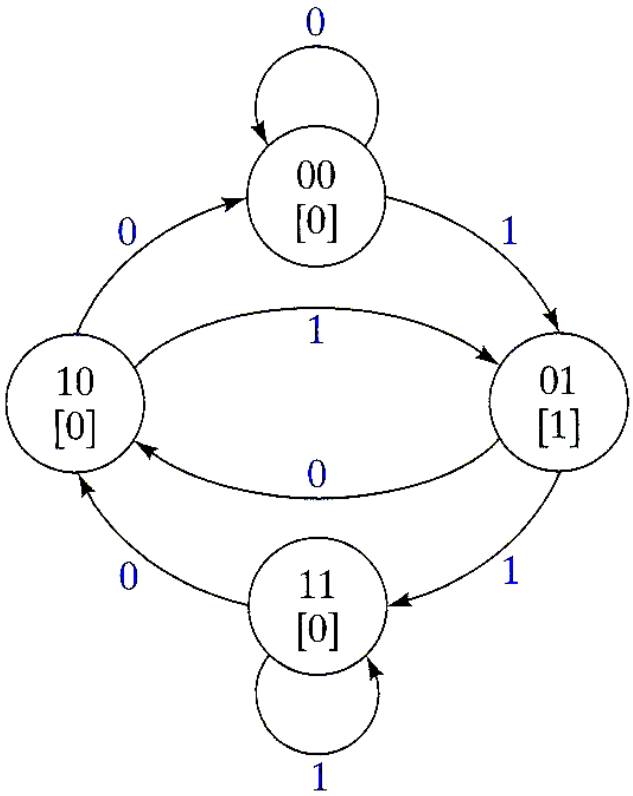
$$Z_1^{n+1} = X^n \left(Z_1^n \oplus Z_0^n \right)$$

$$Z_0^{n+1} = X^n \overline{Z}_1^n Z_0^n$$

Ausgangsfunktion:

$$Y^n = \overline{Z}_1^n Z_0^n$$

Alternative Realisierung mit nicht-minimalen Zuständen



Übergangfunktionen:

$$Z_1^{n+1} = Z_0^n, \qquad Z_0^{n+1} = X^n$$

Ausgangsfunktion:

$$Y^n = \overline{Z}_1^n Z_0^n$$

Hier geringerer Realisierungsaufwand als bei min. Zuständen

Konsequenz:
Die Zustandsminimierung ist sehr hilfreich, um kleinere Schaltwerke zu bilden, insbesondere wenn die Anzahl Flipflops reduziert wird. Denn dann sind auch weniger (Übergangs-)Schaltfunktionen zu realisieren. Aber die Schaltfunktionen können komplexer werden, so dass der Gesamtrealisierungsaufwand effektiv doch steigen kann.

Die Codierung der Zustände ist dabei entscheidend, denn sie bestimmt die Komplexität des Übergangs- **und** des Ausgangsschaltnetzes.

Das Finden optimaler Zustandscodierungen wird schnell sehr komplex. So gibt es bei *m* Zuständen und *n* Bits für die Zustandscodierung $\frac{2^n!}{(2^n - m)!}$ Codierungsmöglichkeiten.

Um dennoch eine günstige (nicht notwendigerweise optimale) Zustandscodierung zu finden, kann folgendes Vorgehen dienen:

1) Der Startzustand erhält die Zustandsnummer „0“.
2) Zustände, die den gleichen Folgezustand haben, sollten bei der Darstellung der Zustandsbits in einem KV-Diagramm (für die Folgezustände) benachbart sein.
3) Folgezustände sollten bei ihrer Darstellung der Zustandsbits in einem KV-Diagramm (·) benachbart sein.
4) Zustände, die dieselbe Ausgabe für die gleiche Eingabe erzeugen, sollten bei ihrer Darstellung der Zustandsbits in einem KV-Diagramm (·) benachbart sein.

2) und 3) reduzieren das Schaltnetz für die Übergangsfunktion, 4) für die Ausgabefunktion.

6.4 Systematische Analyse von Schaltwerken

Die Analyse gegebener Schaltwerke muss für jeden möglichen Zustand alle möglichen Folgezustände mit deren Übergangsbedingungen ermitteln. Das kann z. B. durch folgendes Schema geschehen:

1) Klassifikation des Schaltwerks
   - synchron / asynchron
   - autonom / abhängig von externen Eingängen
   - ggf. Mealy / Moore

2) Darstellung als Automat

3) Ermittlung der Vektorkomponenten ($\underline{X}$, $\underline{Y}$, $\underline{Z}$)

4) Beschreibung der Ausgangs- und Übergangsfunktionen durch Boolesche Ausdrücke

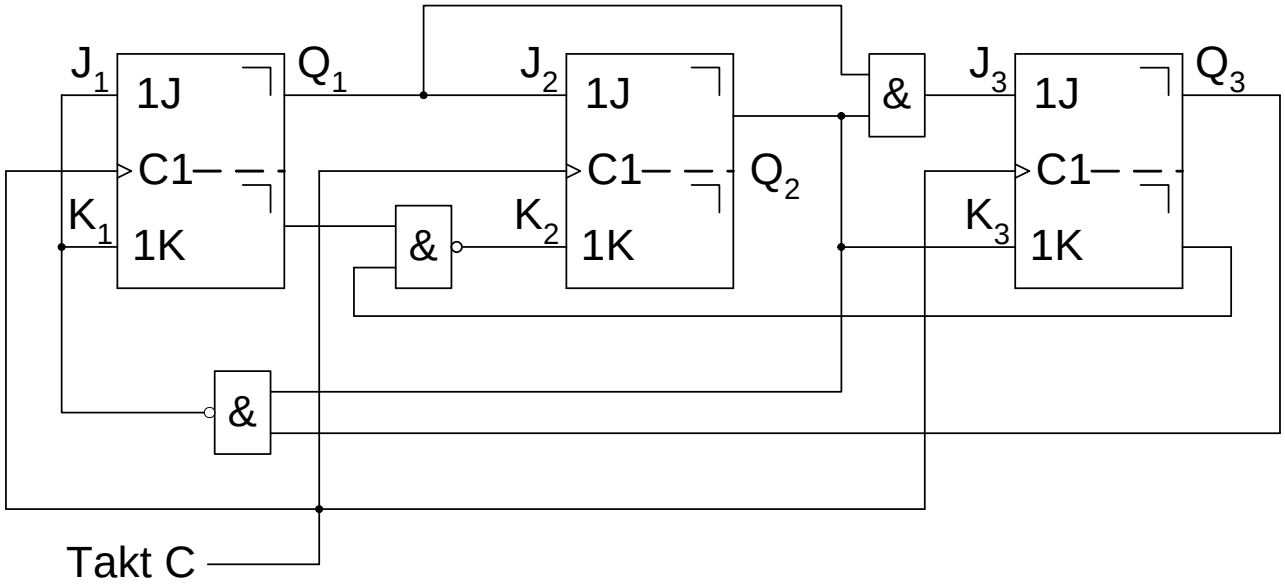
5) Aufstellen der Zustandstabelle

6) Zeichnen des Zustandsgraphen

7) Beschreibung der Funktionsweise

Wenn die Zustandsbits nicht zugänglich sind und ihre Anzahl nicht bekannt ist, kann die vollständige Analyse von Schaltwerken sehr aufwändig werden.

Beispiel:



1) Klassifizierung: autonomes, synchrones Schaltwerk
(keine Eingangsvariablen, aber Takt)

Keine explizite Ausgabe
(Annahme: Ausgabe gleich Zustand)

2), 3) entfällt hier ausnahmsweise

4) Ansteuergleichungen:

$$J_1^n = K_1^n = \overline{Q_2^n Q_3^n} = \overline{Q}_2^n + \overline{Q}_3^n$$

$$J_2^n = Q_1^n, \qquad K_2^n = \overline{\overline{Q}_1^n \overline{Q}_3^n} = Q_1^n + Q_3^n$$

$$J_3^n = Q_1^n Q_2^n, \quad K_3^n = Q_2^n$$

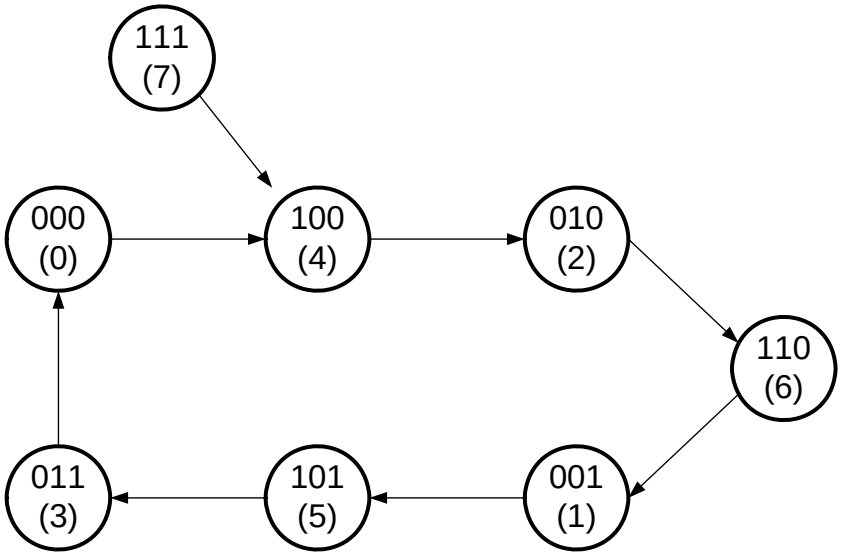
5) Übergangstabelle mit Angabe der FF-Eingänge

(aus Übergangsgleichung für JK-FF bestimmbar)

Q_1^n	Q_2^n	Q_3^n	dez.	J_1^n	K_1^n	J_2^n	K_2^n	J_3^n	K_3^n	Q_1^{n+1}	Q_2^{n+1}	Q_3^{n+1}	dez.
0	0	0	0	1	1	0	0	0	0	1	0	0	4
0	0	1	1	1	1	0	1	0	0	1	0	1	5
0	1	0	2	1	1	0	0	0	1	1	1	0	6
0	1	1	3	0	0	0	1	0	1	0	0	0	0
1	0	0	4	1	1	1	1	0	0	0	1	0	2
1	0	1	5	1	1	1	1	0	0	0	1	1	3
1	1	0	6	1	1	1	1	1	1	0	0	1	1
1	1	1	7	0	0	1	1	1	1	1	0	0	4

Anmerkung: Die Aufnahme der FF-Eingänge in die Zustandstabelle erleichtert die Analyse (besonders bei JK- und RS-Flipflops).

6) Zustandsgraph



7) Funktionsweise:

Schaltwerk mit 7 Zuständen, die zyklisch durchlaufen werden (als mod 7-Zähler verwendbar).

Zustand 111 wird nie erreicht. Wird er z.B. beim Einschalten angenommen, Übergang in Zustand 100.

Timing-Analyse von Schaltwerken

Wichtige Punkte:

- Zustandsänderungen können je nach Flipflop-Typ nur bei der aktiven Taktflanke (-phase) auftreten.
- Die Eingänge der Flipflops sollten (wegen der *Setup*- und *Hold-Zeiten*) kurz vor und meist auch kurz nach der aktiven Taktflanke stabil sein.
- Bei einem Moore-Automaten ändern sich die Ausgänge nur nach der aktiven Taktflanke, beim Mealy-Automaten auch wenn sich Eingänge ändern. D.h., bei einem Mealy-Automaten können auch asynchrone Änderungen des Ausgangs (gewollt oder ungewollt !!!) auftreten.
- Insbesondere bei Mealy-Automaten ist es die sicherste Variante, Ausgänge kurz vor (oder synchron mit) der **nächsten** aktiven Taktflanke zu lesen, weil die Ausgänge dann stabil (und korrekt) sein sollten.

Vorgehensweise:

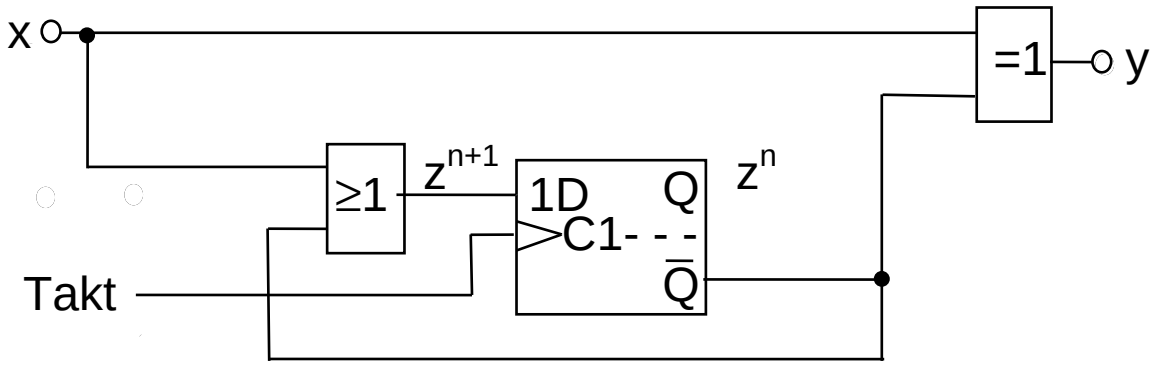
a) Lese den Ausgang für den ersten Eingangsvektor ab und stelle ihn dar.

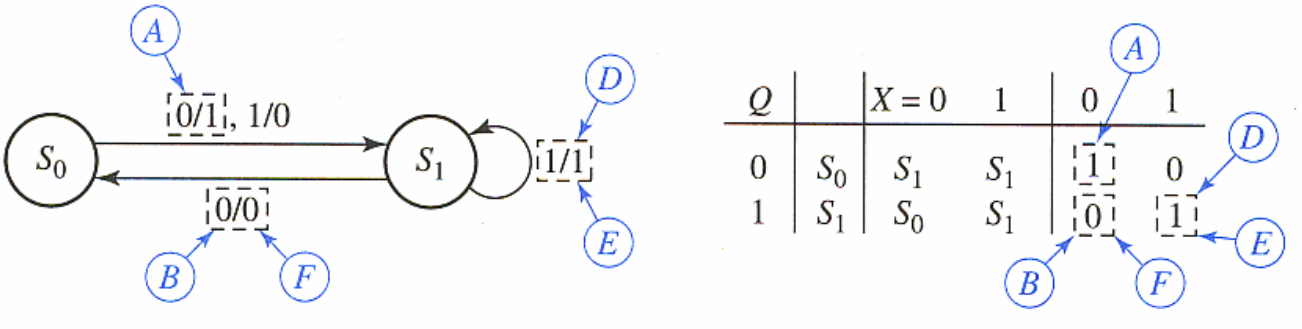
b) Bestimme den nächsten Zustand und stelle ihn dar (nach der nächsten aktiven Taktflanke)

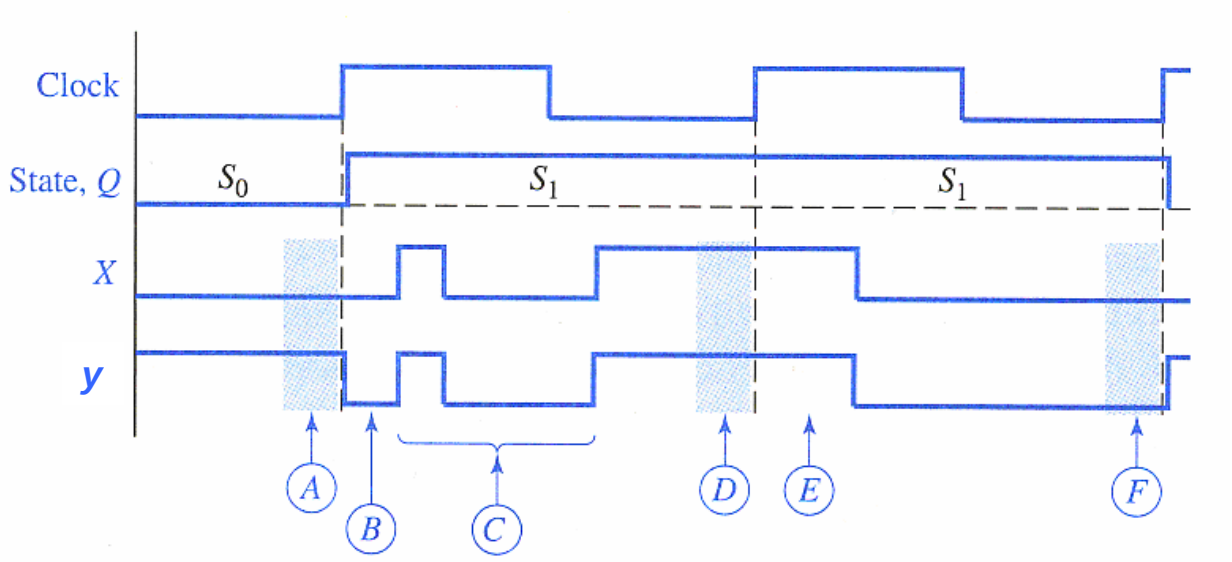
c) Bestimme die Ausgabe für den neuen Zustand und stelle ihn dar.

d) Ändere die Eingabe und wiederhole ab a).

Beispiel für Timing-Analyse (Mealy-Automat)







- Der Ausgangspunkt ist hier: S_0 = 0 und x = 0 → y = 1.
- Dieser Zustand ist stabil vor der aktiven Taktflanke (A).
- Zustands- und Ausgangswechsel durch Taktflanke (B)
- Änderungen des Eingangs x wirken bis auf den Ausgang y, weil es sich um einen Mealy-Automaten handelt (C).

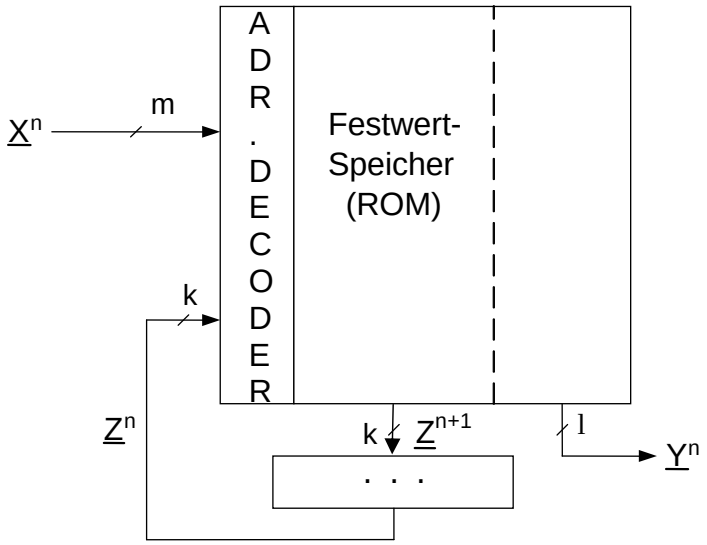
- Der Flipflop-Eingang x ist vor der nächsten aktiven Taktflanke wiederum stabil (D).
- Der Zustand S_1 bleibt stabil (E).
- Änderungen am Eingang x wirken auch in diesem Zustand auf den Ausgang y (E).
- Mit der nächsten Taktflanke wechselt der Zustand wieder (F).

Die Abfolge der Zustandswechsel sollte sich auch im Zustandsgraph verifizieren lassen.

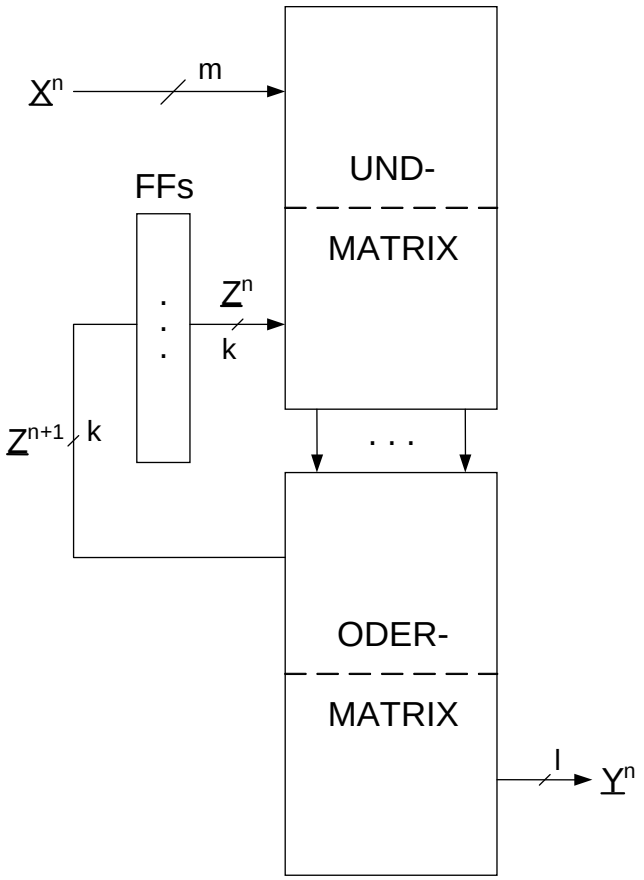
6.5 Andere Schaltwerksrealisierungen

Neben der Variante, Flipflops mit Gattern fest zu verdrahten, gibt es auch programmierbare Varianten (s. auch Kapitel 7):

Schaltwerk mit Festwertspeicher (ROM)



Schaltwerk mit PAL/PLA



Ausgangsbasis ist in beiden Fällen die disjunktive Normalform.

Die Größe des ROMs ($(k+l)\cdot 2^{k+m}$ Speicherstellen) hängt nur von der Anzahl der Eingangs- und Zustandsvariablen sowie der Breite des Ausgangsvektors, aber nicht von den zu realisierenden Schaltwerksfunktionen ab.

Deshalb spielt u. a. die Zustandscodierung keine Rolle (Aufwand), solange die Anzahl k der Zustandsbits minimal ist.

Im Gegensatz dazu sollte bei einer PAL- bzw. PLA-Realisierung die Zustandscodierung so gewählt werden, dass eine (hinreichend) kleine Anzahl von Produkttermen für die Übergangs- bzw. Ausgangsfunktion entsteht.

Bei beiden Realisierungsvarianten sollten D-Flipflops als Zustandsspeicher verwendet werden, weil sie im Gegensatz zu JK- und RS-Flipflops nur einen Ausgang pro Zustandsbit erfordern.

Beispiel: Code-Konverter mit D-Flipflops
(serielle BCD-nach-Excess-3-Umsetzung mit LSB first)

Zustandstabelle

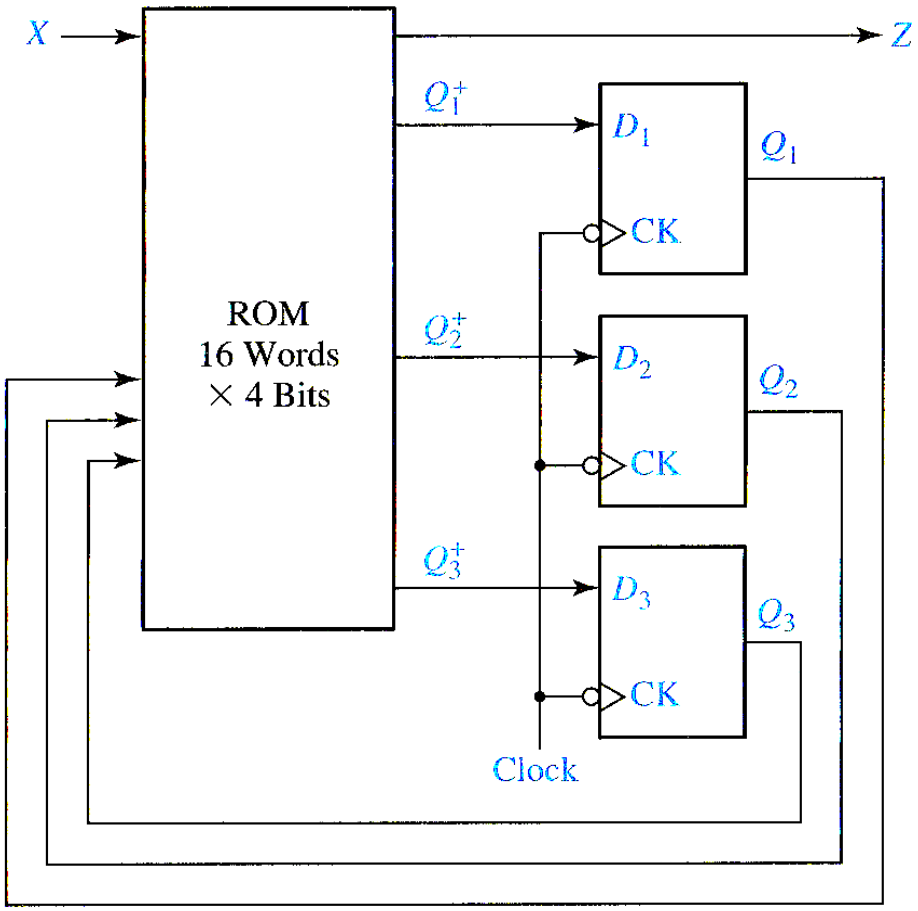
aktueller Zustand	Folgezustand x = 0	Folgezustand x = 1	akt. Ausgabe Z x = 0	akt. Ausgabe Z x = 1
A	B	C	1	0
B	D	E	1	0
C	E	E	0	1
D	F	F	0	1
E	F	G	1	0
F	A	A	0	1
G	A	-	1	-

Wahrheitstafel

X^n	Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}	Z^n
0	0	0	0	0	0	1	1
0	0	0	1	0	1	1	1
0	0	1	0	1	0	0	0
0	0	1	1	1	0	1	0
0	1	0	0	1	0	1	1
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	1
0	1	1	1	-	-	-	-
1	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	1
1	1	0	0	0	1	0	0
1	1	0	1	1	0	0	1
1	1	1	0	-	-	-	-
1	1	1	1	-	-	-	-

Die direkte Umsetzung der Wahrheitstafel in ein ROM benötigt einen Baustein mit (mindestens) 4 Eingängen und 4 Ausgängen (Größe 16×4 Bit) und 3 D-Flipflops.

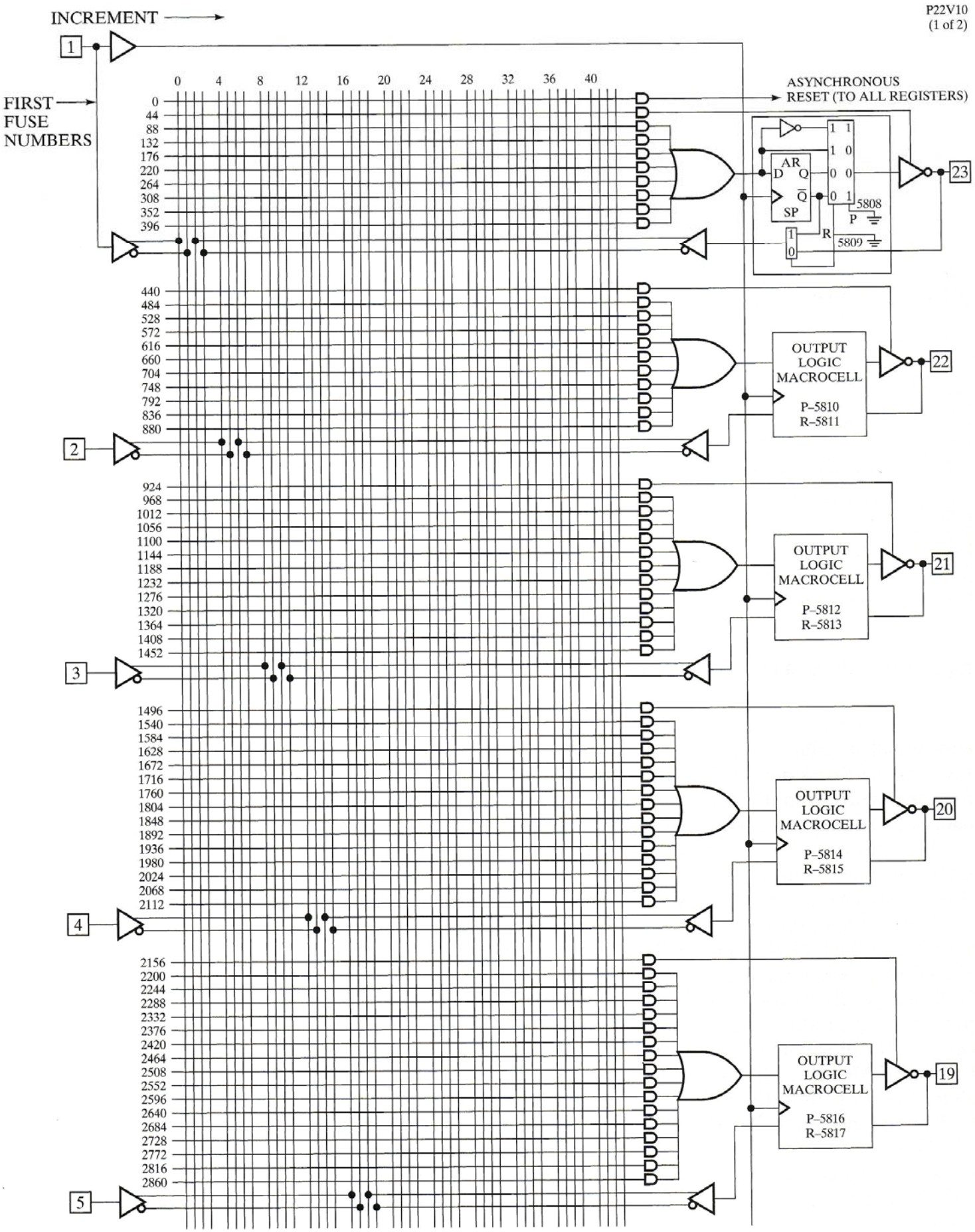
Don´t cares bzw. „-" müssen fest mit ´0´ oder mit ´1´ belegt werden.



Bei einer PLA-/PAL-Realisierung reichen ebenfalls 3 D-Flipflops als Zustandsspeicher sowie Bausteine mit 4 Ein- und 4 Ausgängen. Allerdings sollte die Anzahl der Produktterme (hier: 13) ggf. reduziert werden um ins PLA/PAL zu passen.

So entsteht z. B. folgende Zuordnung für die Ansteuerung der D-Flipflops, die nur noch 7 Produktterme benötigt:

X^n	Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}	Z^n
-	-	0	-	1	0	0	0
-	1	-	-	0	1	0	0
-	1	1	1	0	0	1	0
0	1	-	0	0	0	1	0
1	0	0	-	0	0	1	0
0	-	-	0	0	0	0	1
1	-	-	1	0	0	0	1



Eine Hälfte eines PLD-Bausteins vom Typ GAL 22V10

Schaltwerkrealisierung mittels VHDL

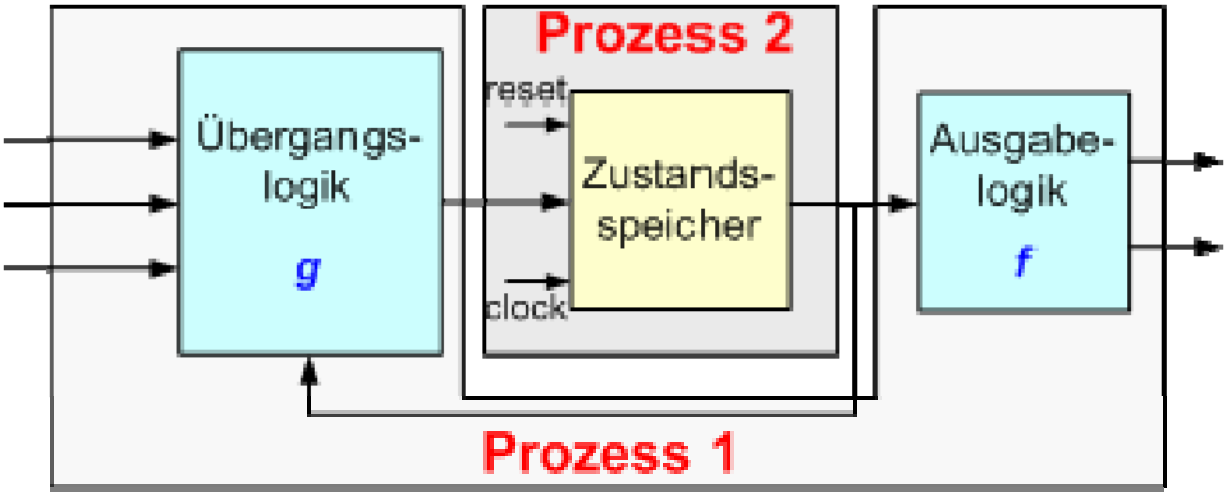
Für komplexere Schaltungen z. B. mit Schaltwerken mit vielen Zuständen oder mit mehreren gekoppelten Schaltwerken ist der direkte Entwurf einer digitalen Schaltung mit Gattern und Flipflops sehr mühsam, fehleranfällig und schlecht wartbar. Deshalb werden bei modernen Designs abstraktere Beschreibungsmethoden angewendet, die auch eine einfachere Modularisierung und Wiederverwendung unterstützen.

Bei einer VHDL-Realisierung von Schaltwerken lassen sich sowohl Mealy- als auch Moore-Automaten einfach implementieren. Der Unterschied liegt darin, ob die Zuweisung der Ausgabe taktsynchron oder nebenläufig erfolgt.

Es gibt mehrere Varianten, den Zustandsspeicher und die Schaltnetze für die Übergangs- und Ausgabefunktion je nach Wartbarkeit und Kompaktheit des VHDL-Codes (in 1, 2 oder 3 Prozesse) aufzuteilen.

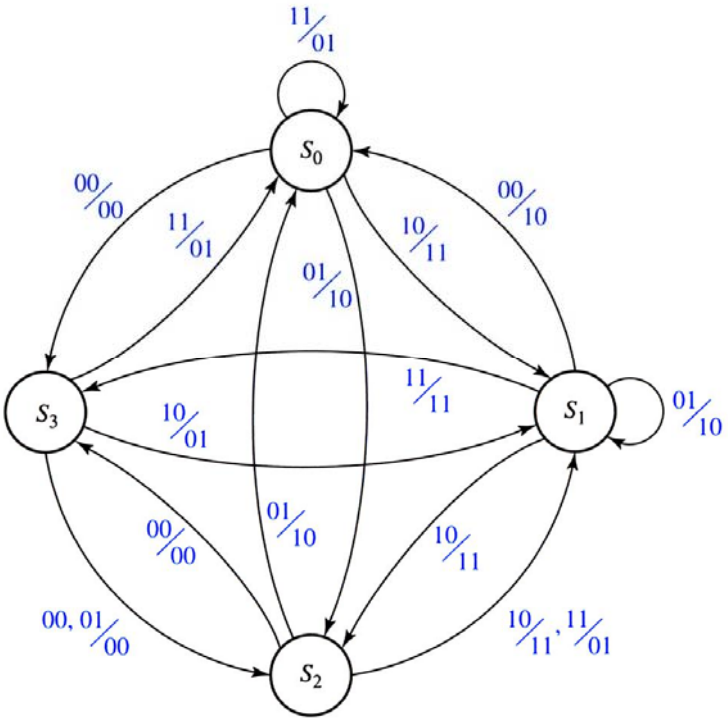
Beispiel: Schaltwerk in der 2-Prozess-Variante

Hier Umsetzung der Zustandstabelle in getrennte Realisierung der Schaltfunktionen und des Zustandsspeichers in zwei Prozessen:



Beispiel:

Zustandsgraph:



Zustandstabelle:

aktueller Zustand	Folgezustand	Ausgabe Z_1Z_2
	X_1X_2 = 00 01 10 11	X_1X_2 = 00 01 10 11
S_0	S_3 S_2 S_1 S_0	00 10 11 01
S_1	S_0 S_1 S_2 S_3	10 10 11 11
S_2	S_3 S_0 S_1 S_1	00 10 11 01
S_3	S_2 S_2 S_1 S_0	00 00 01 01

VHDL-Code für eine 2-Prozess-Variante

```
ENTITY state_machine_1 IS
    PORT(x1, x2, clk : IN bit;
         z1, z2      : OUT bit);
END state_machine_1;
```

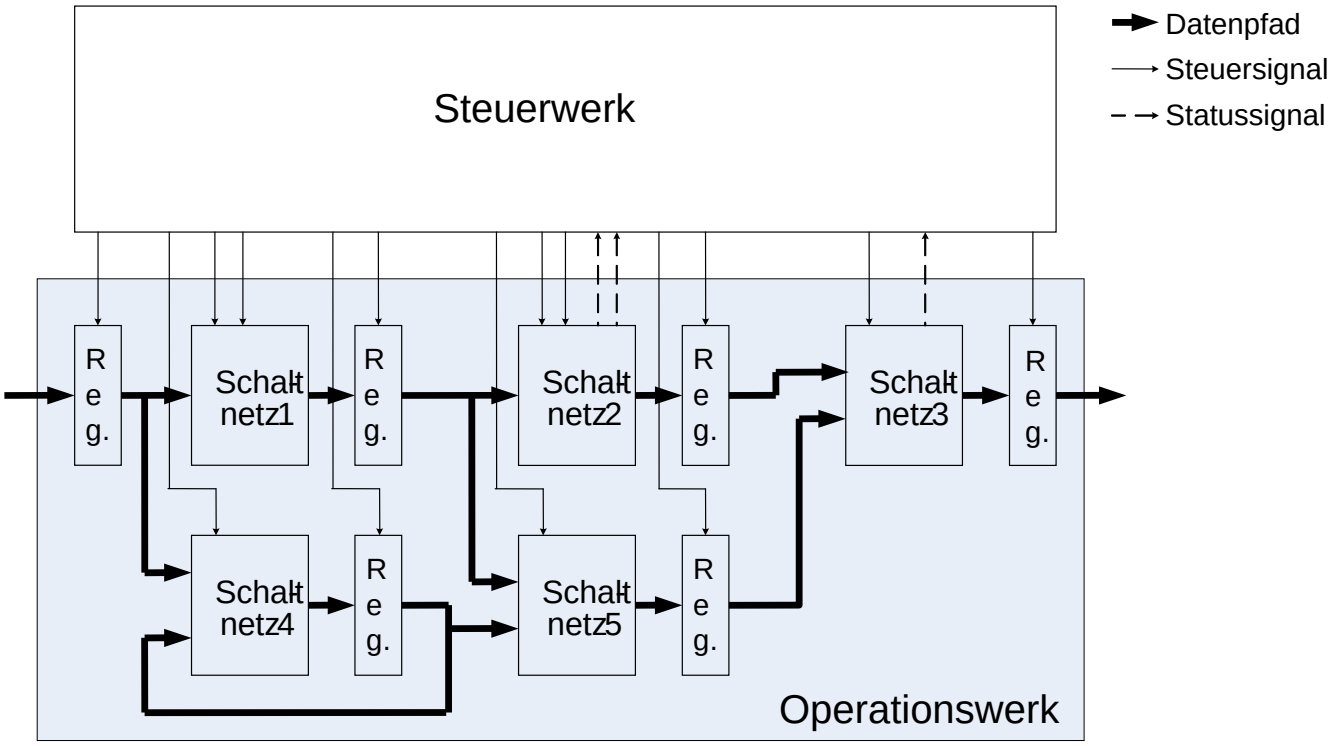
```
ARCHITECTURE sm_1 OF state_machine_1 IS
SIGNAL state, nextstate: INTEGER RANGE 0 TO 3 := 0;

BEGIN
  x12 <= x1 & x2;          -- Concatenation for CASE
  PROCESS(state, x12)      -- Combinational Circuit
  BEGIN                    -- for next state and
    CASE state is          -- output logic
      WHEN 0 =>      -- State S0
        CASE x12 IS
          WHEN "00" => nextstate <= 3;
                       z1 <= '0', z2 <= '0';
          WHEN "01" => nextstate <= 2;
                       z1 <= '1', z2 <= '0';
          WHEN "10" => nextstate <= 1;
                       z1 <= '1', z2 <= '1';
          WHEN "11" => nextstate <= 0;
                       z1 <= '0', z2 <= '1';
        END CASE     -- State S0

      WHEN 1 =>      -- State S1
        CASE x12 IS
          WHEN "00" => nextstate <= 0;
                       z1 <= '1', z2 <= '0';

          -- similar to before until

          WHEN "11" => nextstate <= 0;
                       z1 <= '0', z2 <= '1';
        END CASE;    -- State S3
    END CASE;
  END PROCESS;

  PROCESS(clk)               -- State Register
    BEGIN
      IF clk'event AND clk='1' THEN
      -- raising edge of clock
      state <= nextstate;
    END IF;
  END PROCESS;
END sm_1;
```

6.6 Verhaltensbeschreibung auf Registertransfer-Ebene

6.6.1 Vorbetrachtung

Bei der Registertransfer-(RT-)Ebene spezifiziert man eine digitale Schaltung durch die Beschreibung des Ablaufs über den Datentransport mit den auf die Daten anzuwendenden RT-Operationen. Die Spezifikation auf RT-Ebene ähnelt dadurch stark dem konventionellen Programmentwurf, wobei RT-Operationen vergleichbar zu (Variablen)Zuweisungen sind.

Dem liegt folgende Architektur zugrunde:



Beides, der Datentransport und die auszuführenden Operationen, wird auf Steuersignale für das Aktivieren der Quellregister (*enable*) und das Schreiben der Zielregister (*write*) sowie ggf. für die Auswahl der auszuführenden Operation abgebildet.

Außerdem können aus dem Ergebnis von Operationen abgeleitete Statussignale („Kriterien“ wie Überlauf, Resultat=0 usw.) den Ablauf (des Registertransfers/Algorithmus) verändern.

Verarbeitungseinheit (Operationswerk):

- führt Datentransporte und Verknüpfungen darauf durch
- besitzt Kontrollpunkte, an denen die einzelnen Operationen durch Kontrollsignale c_i ausgelöst werden können
- wegen der Nebenläufigkeit der Hardware ist die **parallele Aktivierung** mehrerer Aktionen möglich

Kontrolleinheit (Steuerwerk, Ablaufsteuerung):

- führt den Algorithm,us aus, indem es den (sequentiellen) Ablauf der Kontrollsignale steuert
- legt fest, welche Signale gleichzeitig anliegen
- sorgt für das richtige Zeitverhalten (Timing)
- festverdrahtete Ausführung oder (mikro)programmierbar (s. Kap. 7)

Kennzeichnend ist also die Aufteilung in eine Verarbeitungseinheit (**Operationswerk**, i.d.R. Realisierung aus Schaltnetzen und Registern), die den Datentransport und die Datenverarbeitung vornimmt, und eine Kontrolleinheit (**Steuerwerk**, Realisierung als Schaltwerk), die über die Steuersignale den Ablauf ggf. abhängig von Statussignalen steuert.

Sowohl die Spezifikation für das Operationswerk als auch für das Steuerwerk lassen sich dann relativ leicht aus einer RT-Beschreibung des Algorithmus ableiten (siehe unten).

Eine RT-Beschreibung muss dazu enthalten:

- Deklarationsteil, d.h. Beschreibung der
  - Busse
  - Register
- Anweisungsteil

Das erlaubt eine abstrahierte, funktionale Sicht auf das Steuer- und das Operationswerk. Dadurch muss lediglich der Transport und die Verarbeitung von Daten (auf Registerebene) beschrieben werden.

Die Umsetzung in Hardware, also der Ablauf und die Steuersignale, werden dadurch implizit festgelegt.

Umsetzung in Hardware

Am einfachsten ist die Umsetzung einer RT-Beschreibung in Hardware nachzuvollziehen, wenn man davon ausgeht, dass

- die transferierten Daten Bits oder Bit-Vektoren sind,
- jede verschiedene Anweisung zu einem eigenen (Teil-) Schaltnetz führt, dessen Funktion (falls nötig) durch Steuersignale kontrolliert wird
- der Automat für das Steuerwerk wie folgt bestimmt wird:
    - RT-Anweisungen korrespondieren zu Zuständen des Steuerwerks.
    - Für jede Fallunterscheidungen in den RT-Anweisungen wird durch ein (spezifisches) Schaltnetz mit einem Ausgang ein eigenes Statussignal für jedes Kriterium generiert, das anzeigt, ob die Bedingung erfüllt ist oder nicht.
    - Die in einer RT-Anweisung spezifizierten Aktionen entsprechen den (parallelen) Steuersignalen, die das Steuerwerk in dem jeweiligen Zustand ausgibt und die über Kontrollpunkte die Verarbeitung im Operationswerk steuern.

Damit stellen die Menge der Kriterien (Statussignale) für das Steuerwerk die Eingangssignale eines Schaltwerks dar und die Menge der Steuersignale (Kontrollsignale) für das Operationswerk die Ausgangssignale des Schaltwerks.

Ein rechnergestützter Entwurf und eine Umsetzung in Hardware sind ausgehend von der RT-Beschreibung automatisiert möglich.

Typische Komponenten auf RT-Ebene

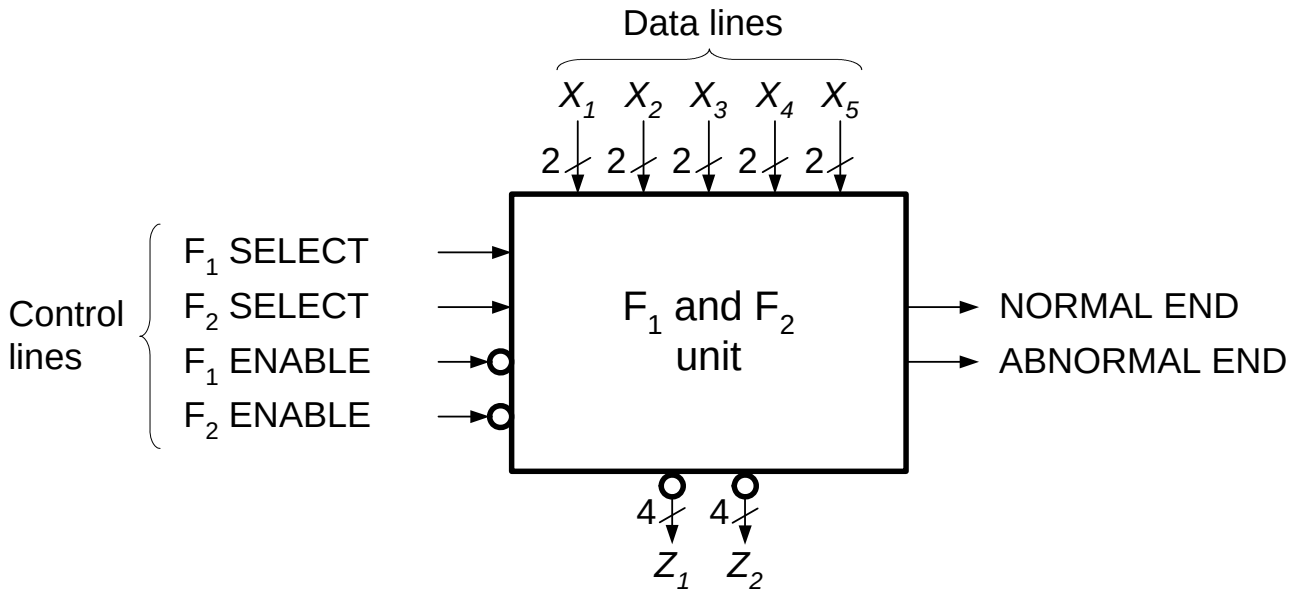
Schaltnetze:

- Wortgatter (UND, ODER, NICHT, ...)
- Multiplexer/Demultiplexer
- Encoder/Decoder
- Arithmetische/logische Einheiten (Addierer, ALUs etc.)
- allgemeinere Schaltnetze (z. B. aus PLAs)

Schaltwerke:

- Automaten
- Register
- Schieberegister, Zähler
- Speicher (RAM, ROM)

Darstellung als Blockdiagramm



Solche schematischen Darstellungen geben nur die Struktur der Schaltung wieder und abstrahieren von der Implementierung und vom (exakten) zeitlichen Verhalten!

6.6.2 Registertransfer-Sprachen (Hardware-Beschreibungssprachen)

Der Entwurf auf RT-Ebene ähnelt stark dem Programmentwurf. RT-Sprachen beschreiben die Verarbeitung von Daten durch Ausdrücke über Datenquellen auf einer abstrakteren, algorithmischen Ebene und sind daher adäquate Beschreibungs- und Entwurfshilfsmittel für komplexe Systeme mit impliziter Taktung der Transitionen.

Grundoperation: RT-Operation (Zuweisung)

$$Z \leftarrow f(X_1, X_2, ..., X_m) ,$$

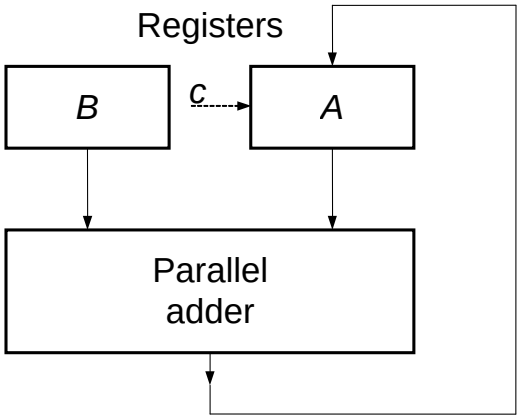
wobei $Z, X_1, X_2, ..., X_m$ Register darstellen und f eine (Schalt-)Funktion der (Quell-)Registerinhalte von $X_1, X_2, ..., X_m$ angibt, deren Ergebnis im (Ziel-)Register Z abgelegt wird.

Bedingte RT-Operationen

if c = 1 **then** $Z \leftarrow f(X_1, X_2, ..., X_m)$ **fi**

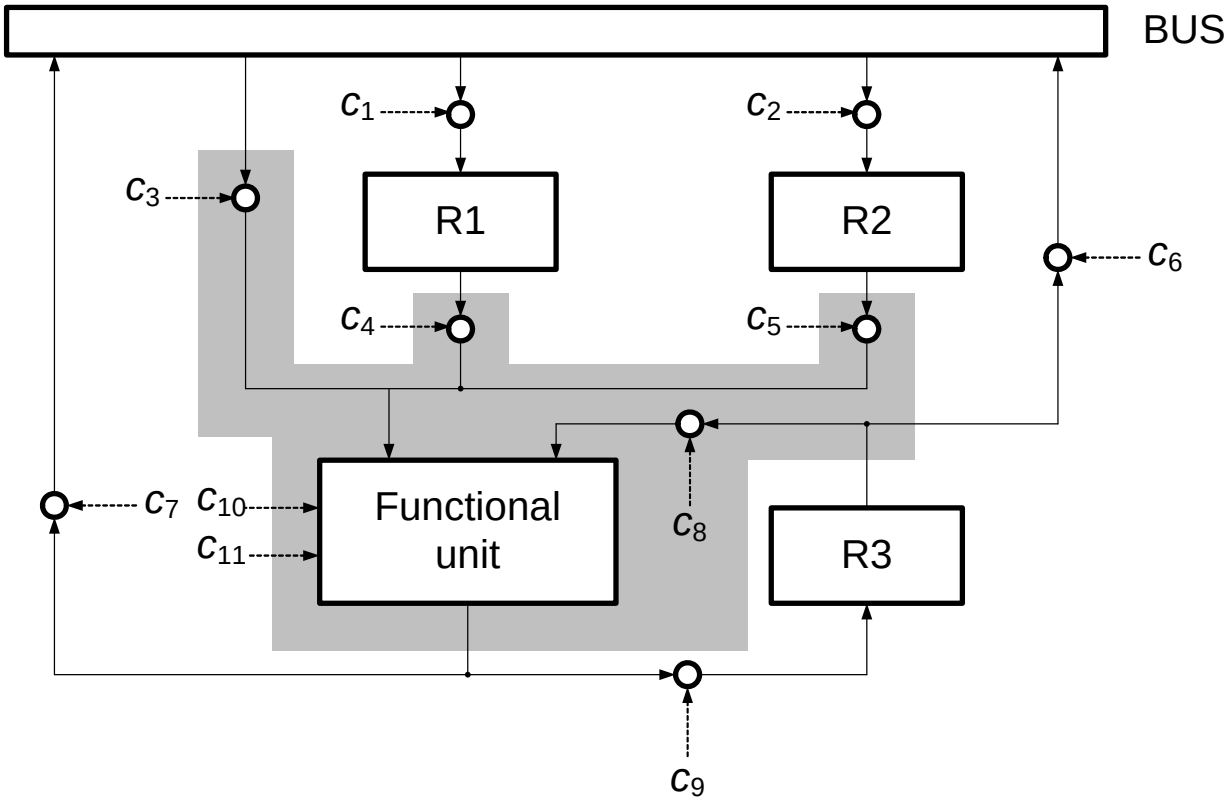
werden nur ausgeführt, wenn die Bedingung erfüllt ist. D.h., nur dann werden die Daten ins Zielregister (hier Z) übernommen.

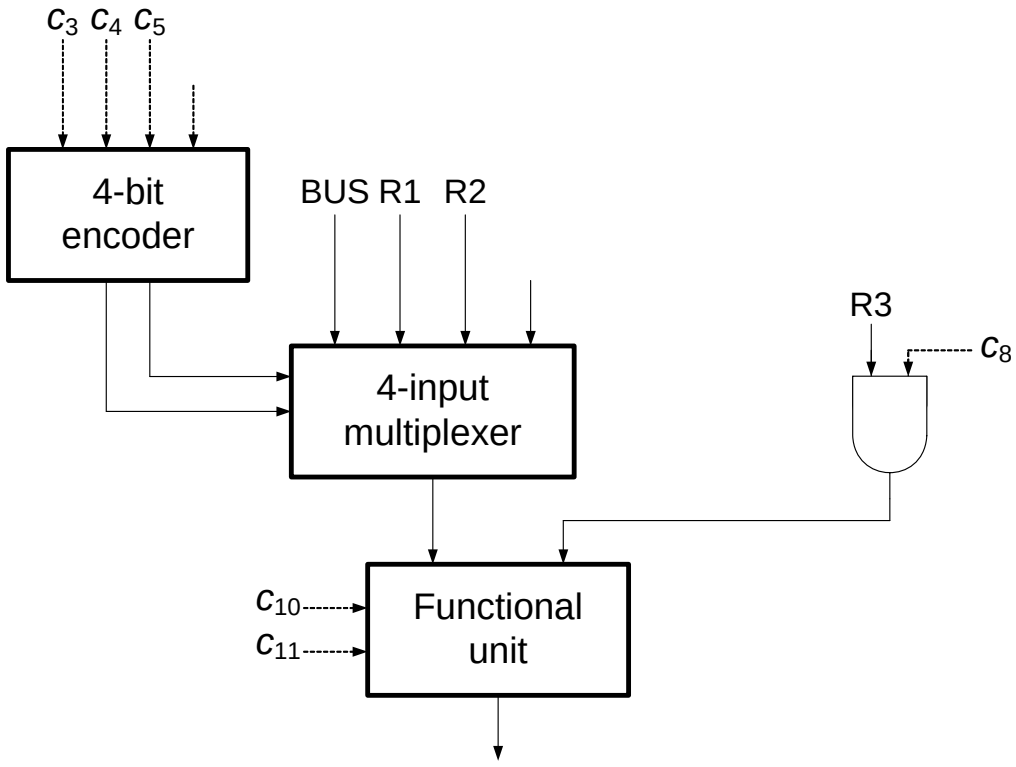
Beispiel: **if** c = 1 **then** A ← A + B **fi**



Die Steuerung des Datentransfers erfolgt über Kontrollpunkte, die durch UND-Gatter oder Multiplexer implementiert werden, so dass die Daten stabil beim Zielregister anliegen, bevor das Schreibsignal kommt.

Beispiel: Steuerung des Datentransfers





6.6.3 Einfache Beispielsprache RTeasy

Die RT-Sprache *RTeasy* und der zugehörige Simulator stammen von der Universität zu Lübeck[*)]. Sie gehen (der Einfachheit halber o.B.d.A.) von folgenden Denkmodell aus:

- Die Ausführung einer RT-Operation geschieht hier in ***einem Taktzyklus*** (je nach RT-Sprache ist auch ein komplexeres Timing möglich).
- Als Register werden positiv flankengesteuerte D-Flipflops verwendet (wenn nicht anders angegeben).
- Ausgangssignale sind normalerweise „0“ (Default). Sie werden beim Beschreiben mit einer „1“ nur für einen Takt auf „1“ gesetzt.
- Das MSB steht links, das LSB rechts, unabhängig von der Nummerierung.

Wichtig:

- Bei Master/Slave-Flipflops kann ein Register im gleichen Taktzyklus gelesen und beschrieben werden.
- Busse können nicht speichern, d. h. auf den Bus gelegte Daten müssen im gleichen Takt abgeholt werden.

INBUS ← A, M ← INBUS

Busse finden auch Verwendung für Ein- und Ausgaben an die umgebende Schaltung/Anwendung.

[*)] Hinweis: http://www.iti.uni-luebeck.de/index.php?id=rteasy&L=1

RTeasy und ein Hilfsblatt mit Details zur RTeasy-Syntax werden im Stud.IP bereitgestellt.

Deklarationen

Alle verwendeten Register müssen deklariert werden.

declare register M(0:7), A(0:7);

deklariert zwei 8-Bit Register M und A.

Alternativ konkatenierte Schreibweise für Register:

M = M(0).M(1).M(2).M(3).M(4).M(5).M(6).M(7);

wobei M(i) das i-te Bit von M darstellt.

Busse werden analog zu Registern deklariert.

declare bus INBUS (0:7), OUTBUS (0:7);

Hier: Busse der Breite 8 Bit.

Externe Signale werden wie Busse deklariert, aber ohne Angabe der Breite.

Beachten: Ausgänge werden bei RTeasy nur für einen Takt auf ’1’ gesetzt.

Speicher werden zusammen mit ihrem Adress- und Datenregister deklariert.

declare register AR(0:15), DR (0:7);
declare memory MEM(AR, DR);

deklariert einen Speicher der Größe 64 KByte (16 Bit Adressraum) und 1 Byte Breite mit Adressregister AR, Datenregister DR.

Sonstiges:

Kommentare werden durch „#“ eingeleitet.

Operationen

Unbedingte Operationen

Pro Taktzyklus können wegen der *Nebenläufigkeit der Hardware* mehrere Operationen gleichzeitig (**parallel**) ausgeführt werden. In RTeasy werden dazu die *nebenläufigen Anweisungen* durch Kommata getrennt.

Die Trennung zum nächsten Taktzyklus erfolgt durch Semikolon.

A ← A + M, M ← INBUS;
OUTBUS ← A, A ← M, M ← 0;

Im ersten Takt wird A um den Inhalt von M erhöht und gleichzeitig M vom INBUS neu geladen (MS-Prinzip).

Im nächsten Takt werden gleichzeitig A auf OUTBUS ausgegeben, M nach A transferiert und M gelöscht.

Es kann auch bitweise zugegriffen werden.

A(0) ← M(0) XOR A(0), A(7) ← 0;

Zähloperationen:

COUNT ← COUNT + 1;

Shiftoperationen:

A(7) ← A(0), A(0:6) ← A(1:7);

bewirkt Ringshift nach links in A.

Aus Timinggründen kann es sinnvoll sein, in einem Takt keine Verarbeitung zu machen. Dafür dient die **NOP**-Anweisung („No Operation“).

Unbedingte Sprünge:

MARKE: M ← 0;
...

A ← A + M, **goto** MARKE;

Nach A ← A + M wird im *nächsten* Takt M ← 0 ausgeführt und das Programm im übernächsten Takt mit der Anweisung nach MARKE: M ← 0; fortgesetzt.

Bedingte Operationen

wie **if** COUNT = 0 **then** A ← 0, M ← 0 **fi**;

werden nur ausgeführt, wenn die Bedingung erfüllt ist. Test und Ausführung finden im *gleichen* Takt statt (wg. *Mealy*-Automat).

ADD: A ← A + M;
...
if COUNT <> 7 **then goto** ADD **else** A ← 0 **fi**;

Wenn COUNT <> 7 ist, dann wird im *nächsten Takt*
A ← A + M ausgeführt und im übernächsten Takt mit der Anweisung nach ADD: A ← A + M; fortgefahren **(then**-Zweig).

Sonst erfolgt im *gleichen Takt* A ← 0 und Fortsetzung im nächsten Takt mit Anweisung nach der **if**-Anweisung (**else**-Zweig).

Mehrere if-Anweisungen können ineinander geschachtelt werden.

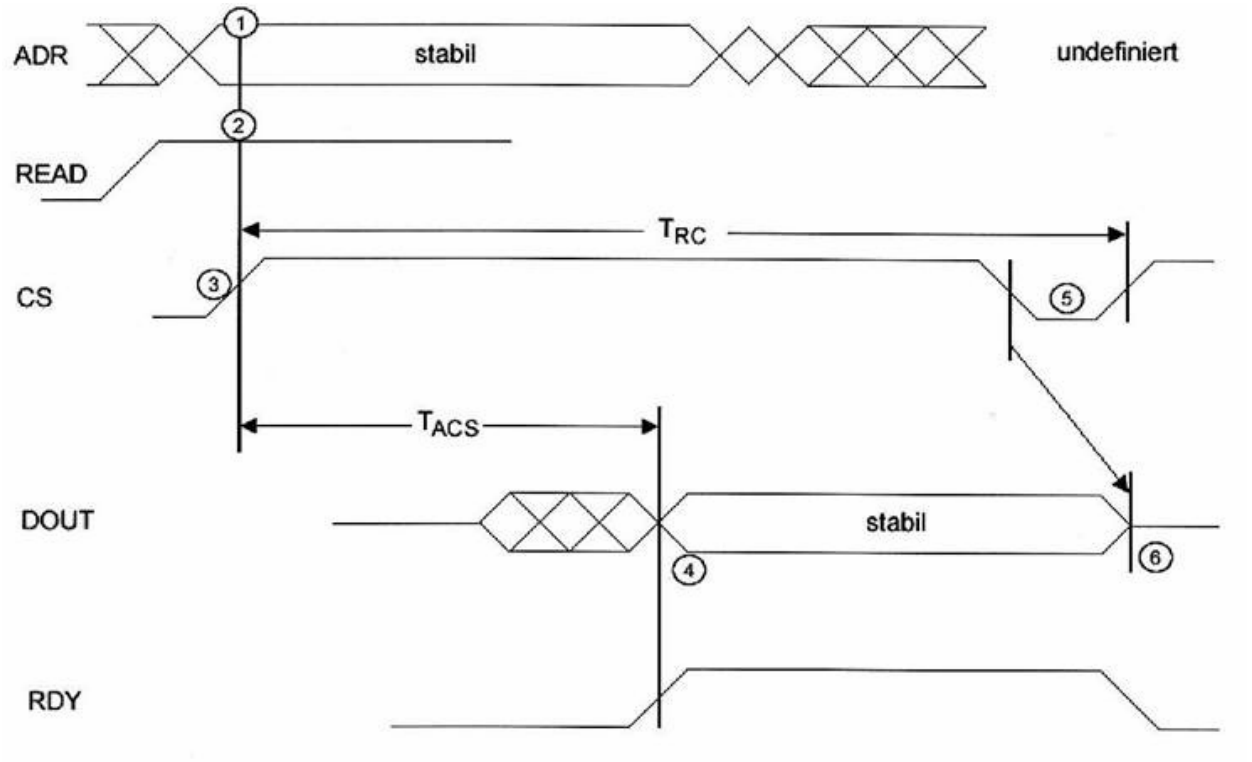
Es kann nicht in if-Anweisungen hinein gesprungen werden.

Speicheroperationen:

READ MEM; WRITE MEM;

Beim Lesen bzw. Schreiben von Speicher MEM müssen das Adressregister AR (und beim Schreiben auch das Datenregister DR) *vorher* gesetzt sein.
Beim Lesen stehen die Daten *nachher* im Datenregister DR.

Typisches Zeitdiagramm eines Speicherzugriffs (Lesen)



1. Die CPU legt die Adresse auf den Adressbus.
2. Die CPU legt durch Aktivierung der Leitung (READ /WRITE) fest, ob sie lesen oder schreiben möchte.
3. Die CPU stößt den Lesevorgang durch Aktivierung von CS (chip-select) an.
4. Der Speicher legt das Datum –mit der Verzögerungszeit T_{ACS}– auf den Datenbus und meldet der CPU über die Leitung RDY, dass die Daten stabil anliegen.
5. Die CPU übernimmt die Daten und quittiert die Übernahme durch Rücksetzen von CS.
6. Der Speicher deaktiviert die Datenleitungen und RDY.

6.6.4 Korrespondenz zur Hardware

Komponententyp	RTeasy	Abb. in Hardware
Register	Explizit deklarieren	Direkte Abbildung
Busse	Explizit als BUS deklarieren	Direkte Abbildung + Kontrolllogik
Speicher	Explizit als MEMORY deklarieren	Direkte Abbildung + Steuersignale f. Lesen u. Schreiben
Ausdrücke	Verknüpfen von Bits / Bitvektoren	Schaltnetz
RT-Operationen	Transfer von Reg.-inhalten (i.d.R. mit Ausführung von Operationen)	Steuersignale für Schaltnetzfunktionen, Busse, Register-Schreiben
IF-Statement	Bedingte Ausführung einer RT-Operation	Schaltnetz zur Bedingungsprüfung → 1 Bit zur Beeinflussung der Zust.-übergangsfkt.
GOTO-Statement	Festlegung des Folgezustands	Direkter Sprung in Zustandsübergangsfkt.
NOP-Statement	Keine Aktion, aber Zeit (Takt) verbrauchen	Zustand ohne Auswirkung in Zustandsübergangsfkt
READ / WRITE	Datentransfer von / zu Speicher	Speicheradressierung, Kontrollsignale, Bus-Transfers

6.7 Entwurf auf Registertransfer-Ebene

Die RT-Programmausführung findet normalerweise sequentiell in der Befehlsreihenfolge im Programm statt (ein Befehl pro Takt). Abweichungen durch (bedingte) Sprünge.

Verschiedene Timing-Modelle sind je nach Ausführung der Kontrolleinheit realisierbar (siehe Kap. 7)

Welche Operationen im Einzelnen für eine gegebene Hardware verwendet werden können, hängt von den vorgesehenen RT-Komponenten ab (Addierer, Zähler, Shiftregister, ...).

Entwurfsschritte für eine Realisierung in Hardware

(1) Beschreibung des Algorithmus als eine Sequenz S von Registertransfer-Operationen.

(2) Analyse von S zur Bestimmung von Anzahl und Typ der Komponenten für die Verarbeitungseinheit (Register, Multiplexer, Funktionseinheiten etc.).

(3) Konstruktion eines Blockdiagramms D für die Verarbeitungseinheit.

(4) Analyse von S und D zur Festlegung der Kontrollpunkte, Kontroll- und Statussignale. Festlegung der Logik zur Implementierung der Kontroll- und Statussignale.

(5) Entwurf der Kontrolleinheit, so dass sie die Kontrollsignale in der durch S festgelegten Reihenfolge generiert.

(6) Leistungs- und Kostenanalyse, evtl. Vereinfachung bzw. Verbesserung des Entwurfs.

Oft gibt es hier je nach Art der Kontrolleinheit mehrere Entwurfsalternativen, die sich bzgl. Leistung und Kosten unterscheiden, oder verschiedene Timing-Modelle.

Z. B. kann

$$A \leftarrow A + B,\ C \leftarrow C + D;$$

durch zwei Addierer parallel oder durch die sequentielle Nutzung eines Addierers realisiert werden.

Beispiel: RT-Entwurf eines binären Multiplizierwerks für Festpunktzahlen

Zahlendarstellung:

- 8 Bit
- negative Zahlen mit Betrag und Vorzeichen (Sign Magnitude)
- Betrag wird als echter Dualbruch dargestellt (d. h. Komma vor der ersten Stelle):

$$X = (x_0.x_1\, x_2 \ldots x_7)_2$$

ergibt

$$N = (-1)^{x_0} \cdot (\sum_{i=1}^{7} x_i \cdot 2^{-i})$$

Multiplikation $P \leftarrow X \times Y$

- Vorzeichen:

$$p_0 \leftarrow x_0 \text{ XOR } y_0$$

- Produkt der Beträge (Addition und Schieben):

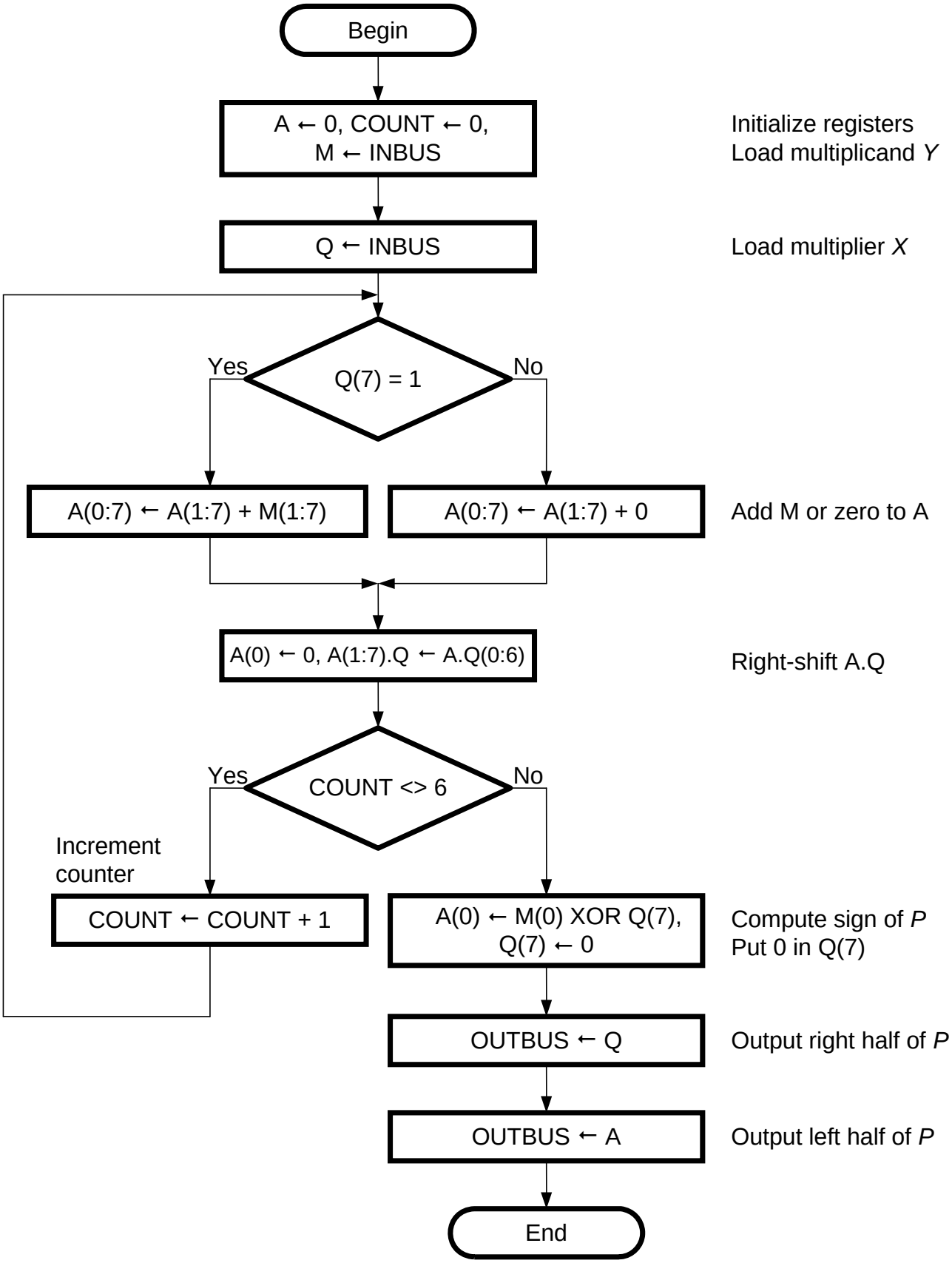
Addition (abhängig von Bit x_{7-i}):

$$P_i \leftarrow P_i + x_{7-i} \cdot Y$$

Rechtsshift: $P_{i+1} \leftarrow 2^{-1} \cdot P_i$

mit $P_0 = 0$, $P_7 = P$ und $i = 0, \ldots, 6$

Flussdiagramm zur Beschreibung des Multiplikations-algorithmus als Sequenz von Registertransfer-Operationen



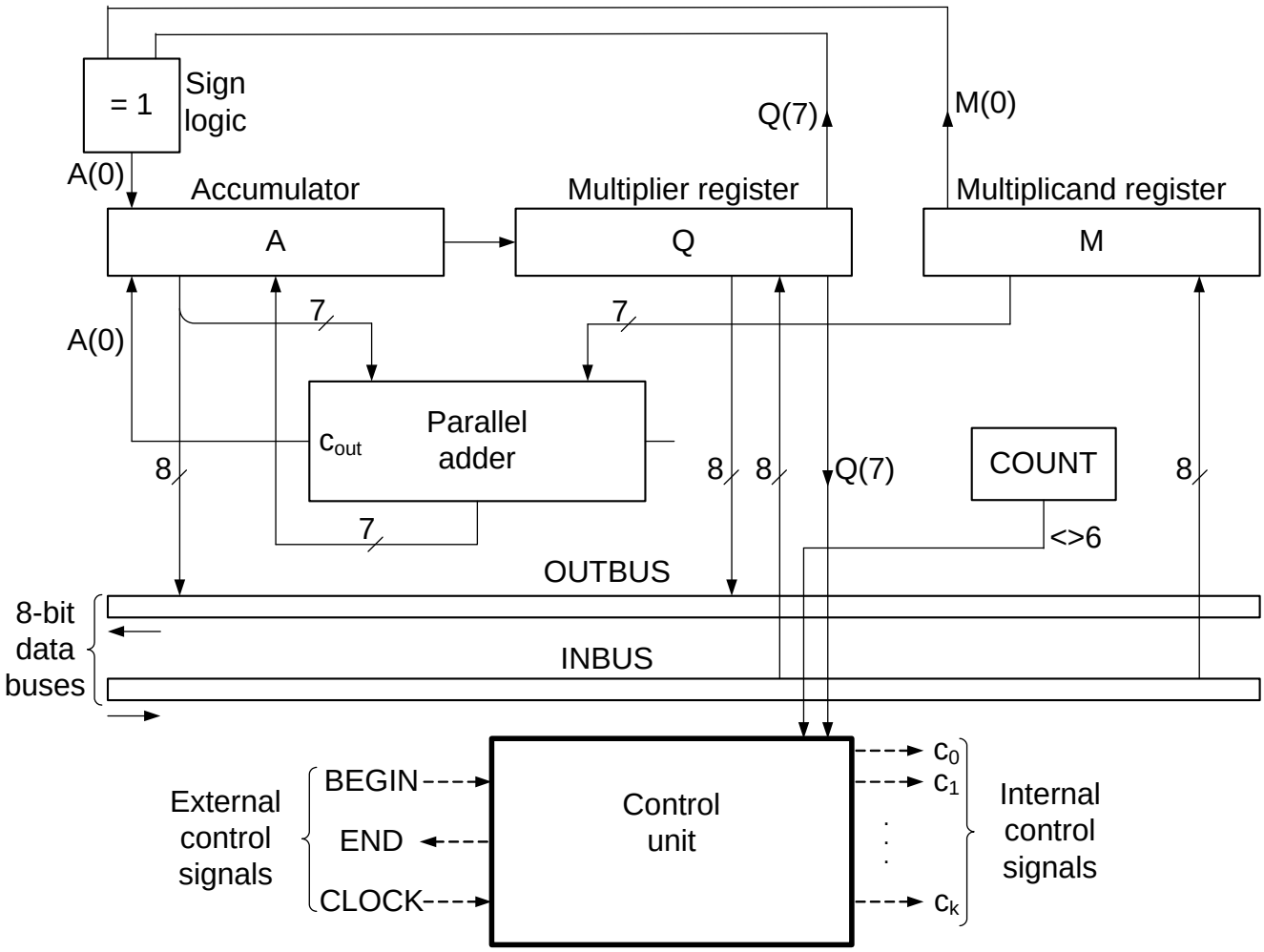
Beschreibung des Multiplikationsalgorithmus mittels RT-Sprache RTeasy

```
declare register  A(0:7), M(0:7), Q(0:7), COUNT(0:2);

declare bus       INBUS(0:7), OUTBUS(0:7);

BEGIN:            A <- 0, COUNT <- 0,
INPUT:            M <- INBUS;

                  Q <- INBUS;

ADD:              if Q(7) = 1 then A(0:7) <- A(1:7) + M(1:7)
                              else A(0:7) <- A(1:7) + 0 fi;

RSHIFT:           A(0) <- 0, A(1:7).Q <- A.Q(0:6),
TEST:             if COUNT <> 6 then COUNT <- COUNT+1,
                                     goto ADD fi;

SIGN:             A(0) <- M(0) XOR Q(7), Q(7) <- 0;

OUTPUT:           OUTBUS <- Q;

                  OUTBUS <- A;
```

Beispiel für den Ablauf einer Multiplikation

Step	Action	Accumulator A	Register Q	
0	Initialize registers	00000000	10110011	= multiplier *X*
1		01010101		= multiplicand *Y* = M
	Add M to A	01010101	10110011	
	Shift A.Q	00101010	11011001	
2		01010101		
	Add M to A	01111111	11011001	
	Shift A.Q	00111111	11101100	
3		00000000		
	Add zero to A	00111111	11101100	
	Shift A.Q	00011111	11110110	
4		00000000		
	Add zero to A	00011111	11110110	
	Shift A.Q	00001111	11111011	
5		01010101		
	Add M to A	01100100	11111011	
	Shift A.Q	00110010	01111101	
6		01010101		
	Add M to A	10000111	01111101	
	Shift A.Q	01000011	10111110	
7		00000000		
	Add zero to A	01000011	10111110	
	Shift A.Q	00100001	11011111	
8	Put sign of *P* in A(0) and set Q(7) to 0	10100001	11011110	= product *P*

Blockdiagramm des Multiplizierwerks



INBUS und OUTBUS dienen zur Ein- bzw. Ausgabe.

Akkumulator A und Multiplikator-Register Q sind zu einem parallel lad- und lesbaren Schieberegister verbunden.

Der Addierer addiert zwei 7-Bit Operanden (Bit 1..7). Ein evtl. entstehender Übertrag (C_{out}) wird in Bit A(0) des Akkumulators transferiert.

Bit Q(7) und das Kriterium „<>6“ können von der Kontroll-einheit abgefragt werden.

Die Einbindung in einen übergeordneten Ablauf kann dadurch erfolgen, dass ein externes BEGIN-Signal die Multiplikation startet und ein END-Signal das Ende der Verarbeitung nach außen meldet.

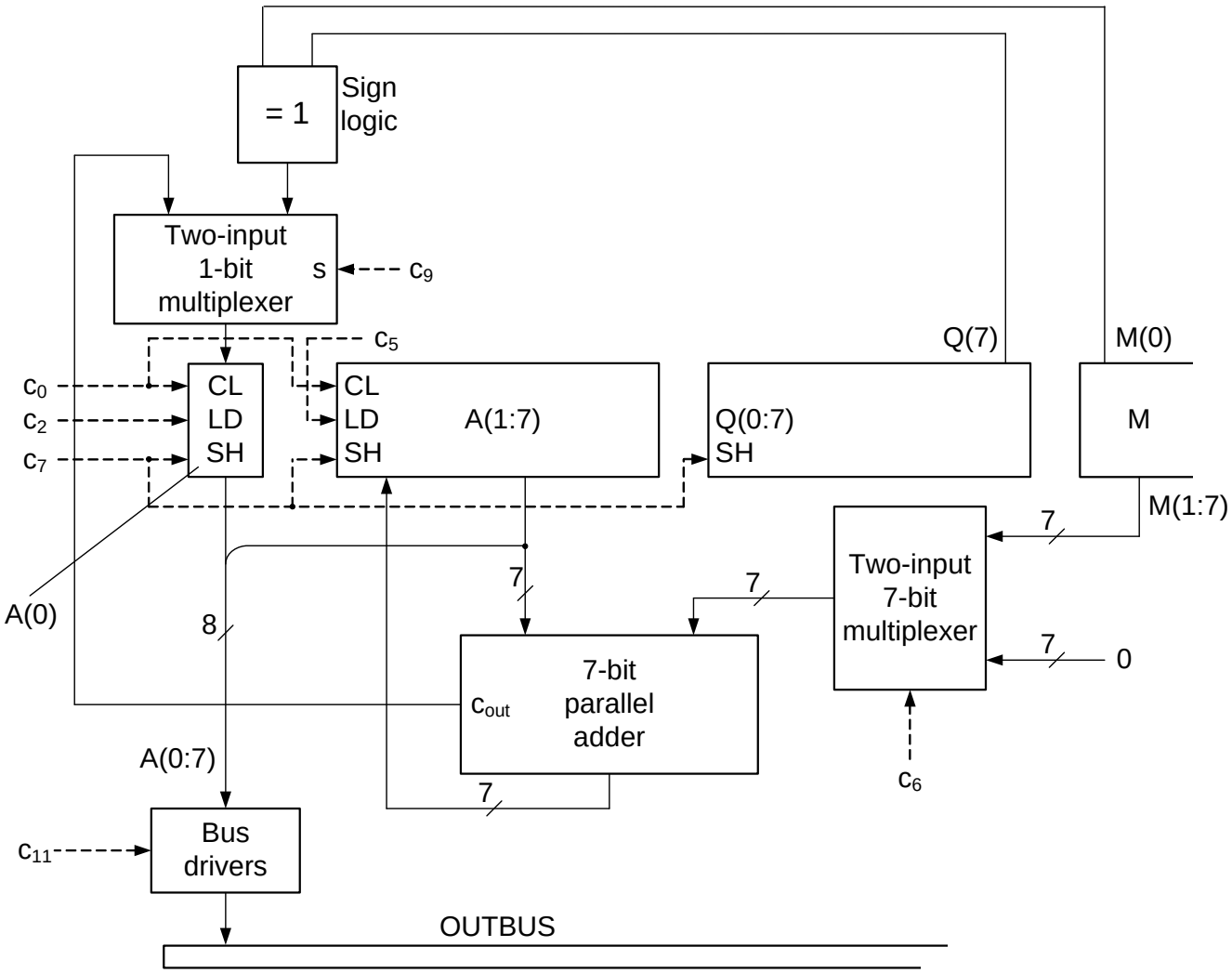
Festlegung der Kontrollsignale und Kriterien

c_0 Clear accumulator A (Reset to zero)
c_1 Clear counter COUNT (Reset to zero)
c_2 Load A(0)
c_3 Load multiplicand register M from INBUS
c_4 Load multiplier register Q from INBUS
c_5 Load main adder outputs into A(1:7)
c_6 Select M or zero to apply to right input of adder
c_7 Right-shift A.Q
c_8 Increment counter COUNT
c_9 Select c_{out} or M(0) XOR Q(7) to load into A(0)
c_{10} Clear Q(7)
c_{11} Transfer contents of A to OUTBUS
c_{12} Transfer contents of Q to OUTBUS
END END signal (external)

k_1 BEGIN (external)
k_2 Q(7) = 1
k_3 COUNT <> 6

Implementierung der Kontrollsignale (Ausschnitt)

(siehe auch Kap. 7)



Rekapitulation:

- Deklaration von Registern, deren Elemente einzeln angesprochen werden können, z. B.

  declare register A(0:7);

  deklariert Register A mit Elementen A(0), ..., A(7)

- Transfer- bzw. Verknüpfungsanweisungen, z. B.

  A ← 0, COUNT ← 0, M ← INBUS;

- Anweisungen, die mit Komma getrennt sind, werden parallel ausgeführt. (Datenabhängigkeiten prüfen!!)

- Busse werden wie Register behandelt.

- Bei Master-Slave-Flipflops sind Schreiben und Lesen im gleichen Taktzyklus möglich.

- Bei bedingten Anweisungen, wie z. B.:

  if Q(7) = 1 **then** A(0:7) <- A(1:7) + M(1:7)
  else A(0:7) <- A(1:7) + 0 **fi**;

  wird der Test im gleichen Taktzyklus wie die Anweisung(en) im **then**- bzw. **else**-Zweig ausgeführt.

- Bei der Anweisung

  A ← 0, A(1:7).Q ← A.Q(0:6),
  if COUNT <> 6 **then** COUNT ← COUNT + 1,
  goto ADD **fi;**

  kann der Test von COUNT wegen der Parallelität der Hardware in gleichem Takt wie die Shift-Operation und das Inkrementieren ausgeführt werden (kein Konflikt).

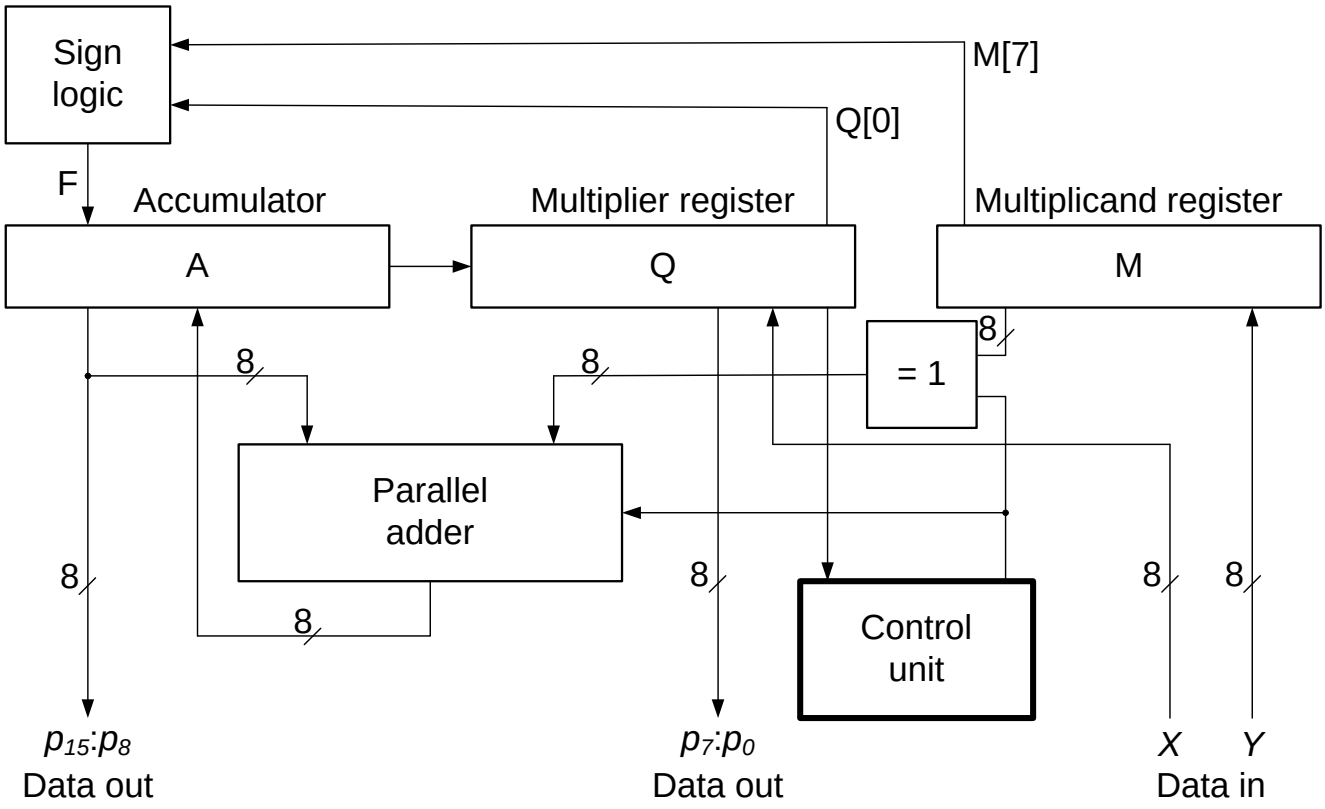
Beispiel: Zweierkomplement-Multiplikation
(nach Robertson (s. Kap. 2) in Registertransfer-Sprache RTeasy)

```
declare register  F, A(0:7), Q(0:7), M(0:7), COUNT(0:2);

declare bus       INBUS(0:7), OUTBUS(0:7);

BEGIN:            A <- 0, COUNT <- 0, F <- 0,
INPUT:            M <- INBUS;

                  Q <- INBUS;

ADD:              if Q(7) then A <- A + M,
                  F <- M(0) and Q(7) or F fi;

RSHIFT:           A(0) <- F, A(1:7).Q <- A.Q(0:6),
                  COUNT <- COUNT + 1;

TEST:             if COUNT <> 7 then goto ADD else
                     if Q(7) then A <- A - M, Q(7) <- 0 fi fi;

OUTPUT:           OUTBUS <- Q;

                  OUTBUS <- A;
```

Anmerkungen:

- Addition bei ADD wird nur bei Q(7) = 1 ausgeführt, sonst Leertakt, danach stets RSHIFT (SHIFT/ADD-Schleife).
- Counter COUNT wird im gleichen Takt erhöht wie RSHIFT.
- Abschließende Subtraktion bei TEST wird nur ausgeführt bei Q(7) = 1, sonst Leertakt.
- Verschiedene Timings möglich, müssen i. Allg. je nach Implementierung der Kontrolleinheit angepasst werden! (s. Kapitel 7).

Blockschaltbild des Zweierkomplement-Multiplizierwerks (Operationswerk)



Prinzipielle Arbeitsweise („Schieben und Addieren“) wie bei Multiplizierer mit Betrag und Vorzeichen.

8-Bit Addition statt 7-Bit Addition (d. h. Vorzeichen wird mit addiert).

Zusätzliche Logik für ***’Korrekturen’***, die bei der Addition im Zweierkomplement bei negativen Multiplikator oder/und Multiplikand nötig werden.

Beispiel für den Ablauf einer Multiplikation im Zweierkomplement

Step	Action	F	Accumulator A	Register Q	
0	Initialize registers	0	00000000	10110011	= multiplier *X*
1			11010101		= multiplicand *Y* = M
	Add M to A	1	11010101	10110011	
	Shift A.Q	1	11101010	11011001	
2			11010101		
	Add M to A	1	10111111	11011001	
	Shift A.Q	1	11011111	11101100	
3			00000000		
	Add zero to A	1	11011111	11101100	
	Shift A.Q	1	11101111	11110110	
4			00000000		
	Add zero to A	1	11101111	11110110	
	Shift A.Q	1	11110111	11111011	
5			11010101		
	Add M to A	1	11001100	11111011	
	Shift A.Q	1	11100110	01111101	
6			11010101		
	Add M to A	1	10111011	01111101	
	Shift A.Q	1	11011101	10111110	
7			00000000		
	Add zero to A	1	11011101	10111110	
	Shift A.Q	1	11101110	11011111	
8			11010101		
	Substract M from A	1	00011001	11011111	
	Set Q(7) to 0		00011001	11011110	= product *P*

Hinweis: Andere Rechenwerke und Schaltwerksoperationen sind ebenfalls leicht mittels Registertransfer-Sprachen zu beschreiben.