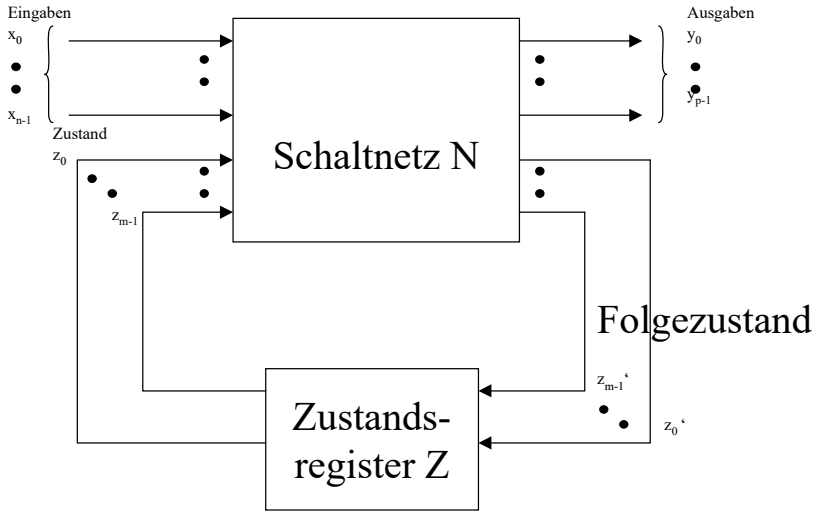
6 Schaltwerke

Bisher haben wir uns nur mit Schaltnetzen befasst, also Schaltungen aus Gattern, die die Ausgaben als eine Funktion der Eingaben unmittelbar (durch Schaltvorgänge) berechnen. Diese Schaltnetze (engl. combinatorial circuits) haben keine Zustände, also kein Gedächtnis, in dem frühere Schaltvorgänge eingespeichert sind. Schaltnetze können nicht abhängig von einem gespeicherten Zustand auf die eine oder andere Weise auf die Eingaben reagieren. Dies ist aber bei einer elektronischen Schaltung wie einem Prozessor eines Computers erwünscht. Wir wollen uns daher jetzt mit Schaltwerken (engl. sequential circuits) beschäftigen, die man als technische Realisierung von endlichen deterministischen Automaten (finite state machines) aus der theoretischen Informatik betrachten kann.

Definition: Eine Schaltung, deren Ausgänge von der Belegung der Eingänge und ihrem inneren Zustand abhängt, wird ein **Schaltwerk** genannt.



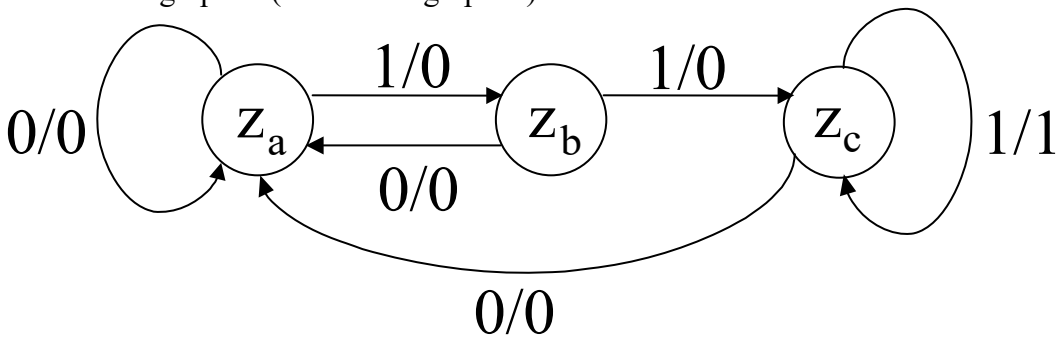
Jedes Schaltwerk enthält Speicherelemente, die den inneren Zustand speichern. Der aktuelle innere Zustand Z und die Belegung der Eingänge X bestimmen den Folgezustand Z' und die Ausgänge Y. Man kann sich die - beliebig in der Schaltung verteilten - Speicherelemente zu einem sogenannten **Zustandsregister** zusammengefasst denken.

Definition: Ein Speicherelement, das die beiden Werte 0 und 1 speichern (annehmen) kann, heißt **Flipflop**.

Der **Zustand** Z eines Schaltwerks ist ein m-Tupel (z_{m-1}, z_{m-2},...,z_1, z_0) aus Komponenten $z_i \in B = \{ 0, 1 \}$. Jede Komponente steht dabei für ein Flipflop.

Bevor wir auf die Realisierung dieser Flipflops eingehen, wollen wir uns ein Beispiel ansehen: Zu entwerfen ist eine Schaltung, die einen Strom aus eingehenden Bits empfängt und interpretiert. Pro Takt kommt über eine Leitung x ein Bit. Wenn hintereinander drei Einsen empfangen worden sind, soll am Ausgang eine Lampe y angehen. Also, y ist 1, wenn die letzten drei empfangenen Bits 1 waren, sonst ist y 0. Ein Schaltnetz wäre mit dieser Aufgabe überfordert, da es kein Gedächtnis hat, in dem es beispielsweise den vorletzten Wert speichern kann.

Um nun unser Schaltwerk zu bauen, veranschaulichen wir uns zunächst die Funktion anhand eines Schaltwerksgraphen (Automatengraphen).



Dieser Graph ist folgendermaßen zu interpretieren: Jeder Kreis ist ein möglicher Zustand, jeder Pfeil ein möglicher Zustandsübergang. An einem Pfeil steht eine Beschriftung vom Typ Eingabe/Ausgabe, d.h. bei Eingabe des Wertes vor dem / im Zustand, in dem der Pfeil beginnt, wird der Wert nach dem / ausgegeben und in den Zustand, auf den der Pfeil zeigt, gewechselt.

In unserem Falle beginnen wir im Zustand z_a. Bei Eingabe einer 0 bleiben wir in diesem Zustand und wir geben eine 0 aus (y=0, die Lampe leuchtet nicht). Bei Eingabe einer 1 wechseln wir in den Zustand z_b und geben eine 0 aus, denn bisher ist nur eine 1 eingegeben worden. Wenn nun eine 0 eingegeben wird, gehen wir wieder zurück in den Zustand z_a und geben eine 0 aus.

Wenn wir in z_b eine 1 eingeben, haben wir bereits zwei 1en hintereinander gesehen. Wir gehen in den Zustand z_c und geben eine 0 aus. Nun betrachten wir z_c. Wenn wir in diesem Zustand sind, haben wir als letzte beide Eingaben eine 1 gehabt. Wenn jetzt noch eine 1 kommt, dann muss die Lampe leuchten, d.h. unsere Ausgabe y muss aus 1 gehen. Aber welches ist in diesem Fall der Folgezustand? Nun, wiederum z_c, denn auch hier waren die beiden letzten Eingaben 1en. Wenn in z_c eine 0 eingegeben wird, gehen wir wieder mit Ausgabe 0 in z_a zurück, denn eine Folge von drei 1en müsste ganz neu beginnen.

Der nächste Schritt ist die Codierung der Eingaben, Ausgaben und Zustände als Binärzahlen. Nun, x und y können nur zwei Werte annehmen, daher sind sie bereits Binär codiert. Für die Zustände wählen wir folgende Codierung: z_a = 00, z_b = 01, z_c = 10. Diese beiden Bits bezeichnen wir mit z_1 und z_0.

Jetzt können wir die Wertetabelle für das erforderliche Schaltnetz aufstellen:

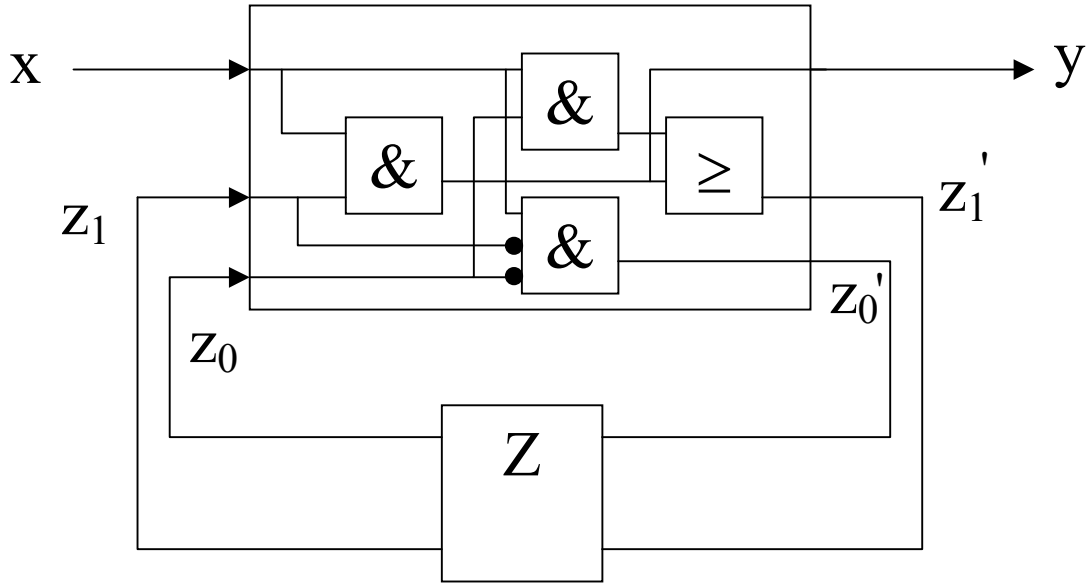
x	z_1	z_0	z_1‘	z_0‘	y
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	X	X	X
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	X	X	X

Die Anwendung von KV-Diagrammen führt uns zu folgenden disjunktiven Minimalformen für die z_i‘ und für y:

$z_0' = x \overline{z_0}\ \overline{z_1}$

$z_1' = x z_0 + x z_1$

$y = x z_1$

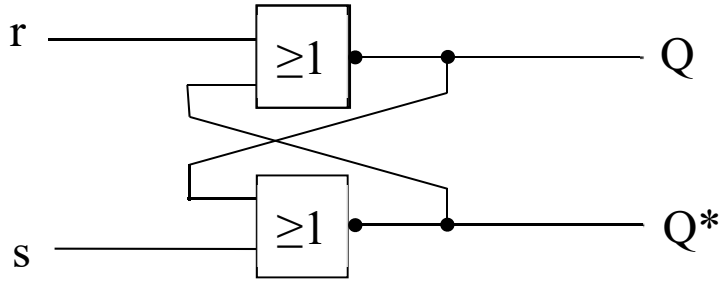


Das gesamte Schaltwerk kann also folgendermaßen realisiert werden:
Was uns nun noch fehlt, ist die Realisierung des Zustandsregisters Z. Wie sieht überhaupt ein Baustein aus, der ein Bit speichern kann, also das Flipflop?

Im folgenden werden wir einige verschiedene Flipfloptypen kennen lernen. Wir beginnen mit einfachen Flipflops, die wir dann mit zusätzlichen Funktionen ausstatten bis hin zu den sogenannten System-Flipflops, die wir schließlich für den Schaltwerksaufbau verwenden können.

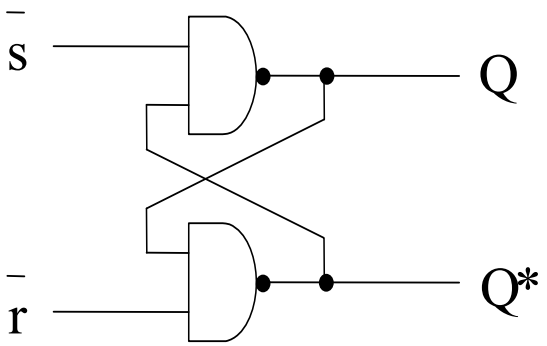
6.1 Das r-s-Flipflop

Das einfachste Speicherelement ist das r-s-Flipflop (Basis-Flipflop). Das r steht für rücksetzen (reset), das s für setzen (set). Es kann mit zwei rückgekoppelten Nor-Gattern realisiert werden:



r	s	Q	Q*
0	0	Q_{alt}	Q^*_{alt}
0	1	1	0
1	0	0	1
1	1	0	0

Wenn r = 0 und s = 0 ist, speichert das Flipflop seinen alten Wert am Ausgang. Wenn r = 0 und s = 1 ist, wird es gesetzt (Q = 1); bei r = 1 und s = 0 wird es rückgesetzt (Q = 0). Die Eingabe r = 1 und s = 1 ist verboten. Am Ausgang würde sich nämlich bei dieser Beschaltung der Zustand Q = 0 und Q* = 0 einstellen. Bei einer direkt darauf folgenden Eingabe r = 0 und s = 0 würde das Flipflop in einen instabilen Zustand geraten.



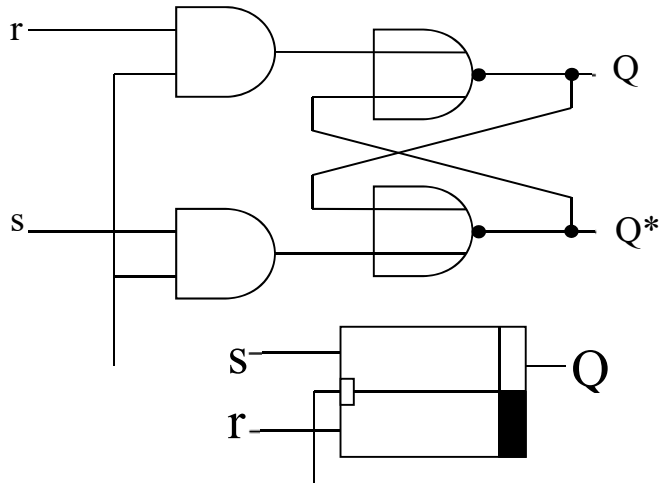
$\overline{r}$	$\overline{s}$	r	s	Q	Q*	
1	1	0	0	Q_{alt}	Q^*_{alt}	
1	0	0	1	1	0	
0	1	1	0	0	1	
0	0	1	1	1	1	← verboten

Das r-s-Flipflop kann auch mit NAND-Gattern aufgebaut werden. Bis auf die verbotene Eingabe ist die Wertetabelle identisch mit der des NOR-Flipflops (weil r und s invertiert eingehen).

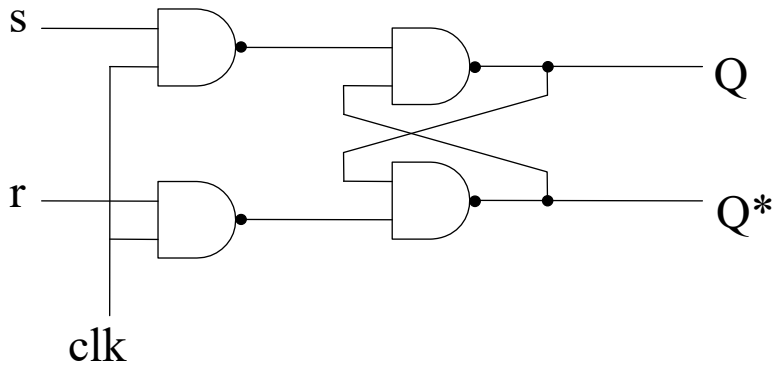
Ein solches Basisflipflop übernimmt beim Setzen oder Rücksetzen den Wert sofort an den Ausgang. Häufig ist diese Funktion aber nicht erwünscht, sondern man möchte den Zustand des Flipflops nur zu definierten Zeiten ändern. Zu diesem Zweck macht man das Übernehmen des Wertes abhängig von einem weiteren Eingang, z.B. einem Takteingang: Solange der Takt 0 ist, soll das Flipflop seinen alten Wert speichern. Wenn der Takt jedoch 1 ist, soll es gesetzt oder rückgesetzt werden können, es soll aber auch bei r=s=0 seinen alten Wert erhalten können. r=s=1 ist wiederum verboten.

Wertetabelle:

clk	r	s	Q	Q*
0	0	0	Q_{alt}	Q^*_{alt}
0	0	1	Q_{alt}	Q^*_{alt}
0	1	0	Q_{alt}	Q^*_{alt}
0	1	1	Q_{alt}	Q^*_{alt}
1	0	0	Q_{alt}	Q^*_{alt}
1	0	1	1	0
1	1	0	0	1
1	1	1	verboten	



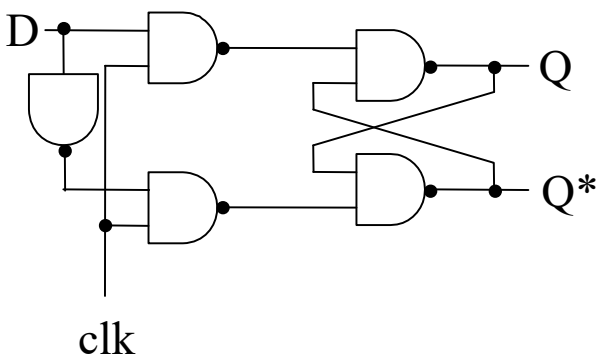
Dieses getaktete r-s-Flipflop nennt man auch ein r-s-Auffangflipflop (r-s-Latch). Es kann nämlich während der positiven Taktphase (also während clk=1 ist) einen Wert aufnehmen, der dann während der negativen Taktphase (clk=0) gespeichert bleibt. Man kann ein Latch natürlich wieder in NAND-Logik aufbauen.



Nun sind wir häufig lediglich daran interessiert, einen Eingabewert, also ein Bit, unverändert zu übernehmen und zu speichern. Dann brauchen wir nicht die Funktionalität eines r-s-Flipflops, das ja immer den Nachteil der verbotenen Eingabekombination hat, sondern ein D-Flipflop. D steht für delay. Wir sehen hier die Wertetabelle und eine mögliche Realisierung mit einem NAND-Basisflipflop.

Wertetabelle:

clk	D	Q	Q*
0	0	Q_{alt}	Q^*_{alt}
0	1	Q_{alt}	Q^*_{alt}
1	0	0	1
1	1	1	0

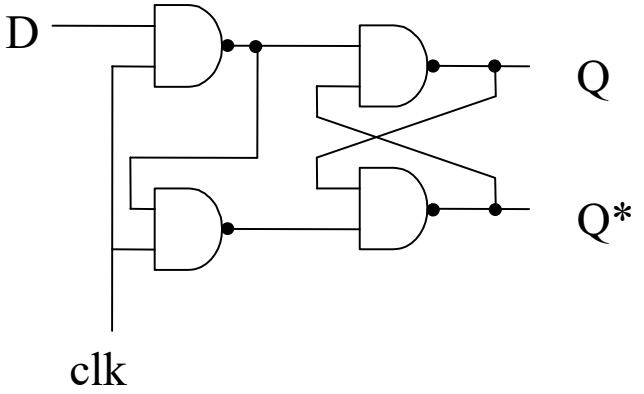


Man kann den Inverter einsparen, indem man das Layout geringfügig verändert. In dieser Version muss aber der Taktimpuls mindestens so lang sein wie eine Schaltzeit eines Nand-

Gatters, da sonst bei clk = 1 das invertierte D sich nicht mehr auf dem unteren NAND-Gatter auswirken kann.

Wertetabelle:

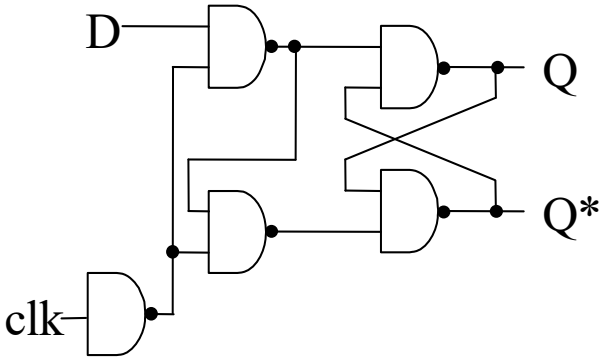
clk	D	Q	Q*
0	0	Q_{alt}	Q^*_{alt}
0	1	Q_{alt}	Q^*_{alt}
1	0	0	1
1	1	1	0



Alle bisher betrachteten Flipflops übernahmen die Eingangswerte, während der Takt auf 1 war und speicherten, wenn der Takt auf 0 war. Man nennt dies ein **positiv levelgesteuertes Auffangflipflop**. Man kann aber jetzt natürlich durch Inversion des Taktsignals ein negativ levelgesteuertes Auffangflipflop bauen, das bei clk = 0 übernimmt und bei clk = 1 speichert:

Wertetabelle:

clk	D	Q	Q*
0	0	0	1
0	1	1	0
1	0	Q_{alt}	Q^*_{alt}
1	1	Q_{alt}	Q^*_{alt}

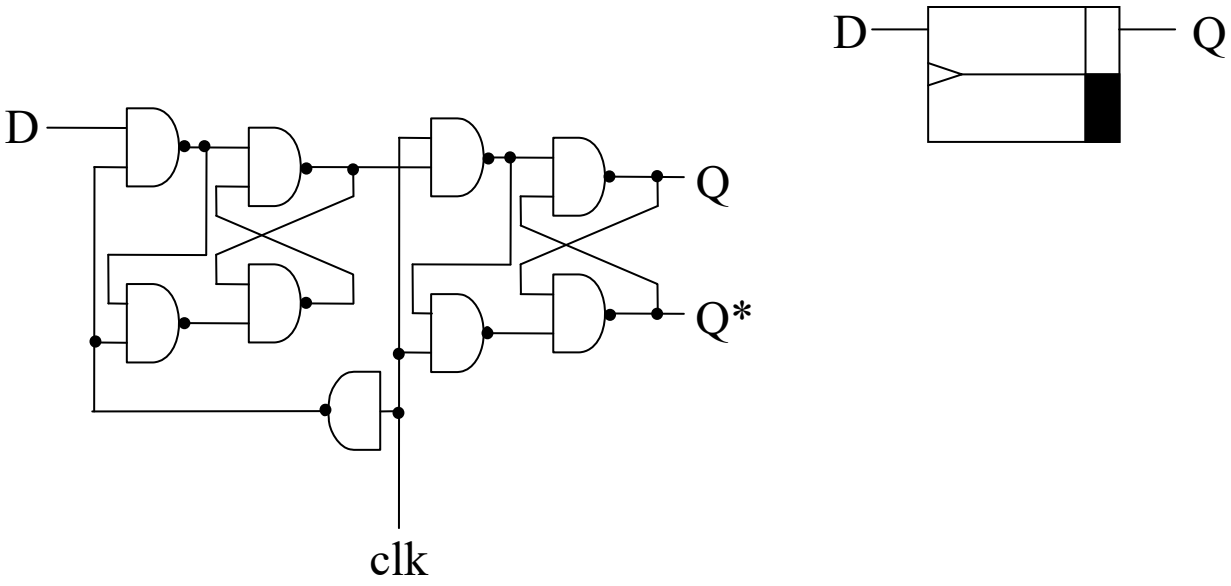


Leider genügen Auffangflipflops noch nicht den Anforderungen, die wir an die Speicherelemente unserer Schaltwerke stellen müssen. Was würde passieren? Sobald der Takt auf 1 ist, wird der Eingang der Flipflops als neuer Zustand an den Ausgang übernommen. Dieser neue Zustand liegt aber jetzt am Schaltnetz an, das nach einigen Gatterschaltzeiten neue Ausgänge produziert. Einige von diesen Ausgängen liegen als Folgezustand an den Eingängen der Flipflops. Da aber der Takt immer noch 1 sein kann, wirken diese sich nun wieder auf die Ausgänge der Flipflops aus und der Zustand wird wieder überschrieben. Mit anderen Worten: Die Signale würden einige Male in dieser Rückkopplungsschleife herumlaufen, bis irgendwann das Taktsignal auf 0 geht, und das Flipflop schließlich speichert.

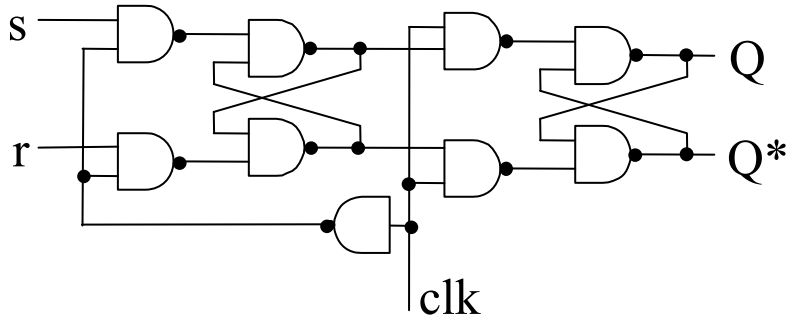
Was wir also stattdessen benötigen, ist ein Flipflop, das seinen Ausgang stabil lässt, solange die neuen Eingänge übernommen werden und dann, zu einem späteren Zeitpunkt, seine Eingänge nicht mehr übernimmt und stattdessen den übernommenen Wert an den Ausgang weiterschaltet. Ein Flipflop, das dieses leistet, nennt man ein **System-Flipflop**.

Ein System Flipflop kann man als sogenanntes **Master-Slave-Flipflop** aus zwei Auffangflipflops zusammensetzen, und zwar einem negativ levelgesteuerten und einem positiv levelgesteuerten Latch.

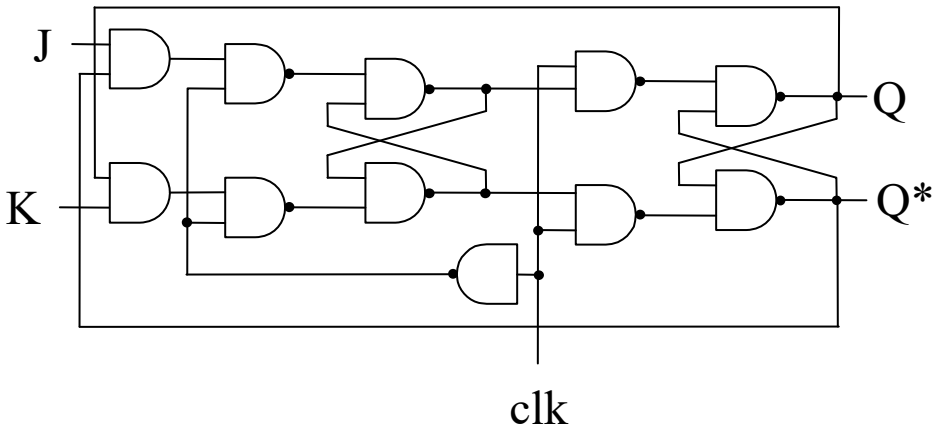
6.2 Master-Slave D-Flipflop



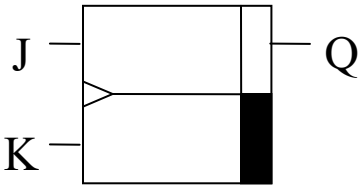
Entsprechend dem D-Flipflop kann man natürlich auch r-s-Flipflops als Master-Slave-Flipflops aufbauen. Allerdings bleibt das Problem mit der verbotenen Eingabe.



Um dieses Problem zu umgehen, kann man ein sogenanntes J-K-Flipflop bauen. Bei diesem sind die Ausgänge an die Eingänge rückgekoppelt.

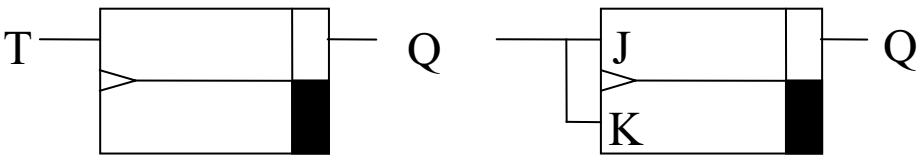


J	K	Q
0	0	Q_{alt}
0	1	0
1	0	1
1	1	$\overline{Q_{alt}}$

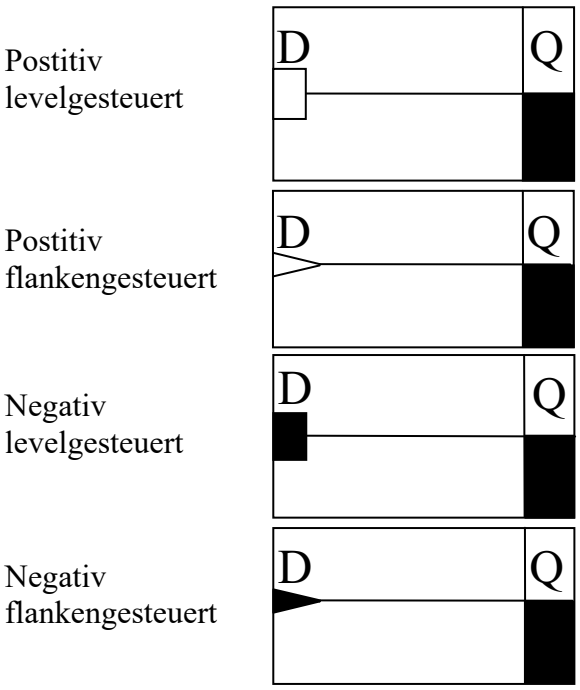


Gelegentlich benötigt man nur die Funktionalität der ersten und letzten Spalte der Wertetabelle eines J-K-Flipflops. In diesem Fall kann man ein Wechselflopflop (W-Flipflop oder auch T-Flipflop) verwenden, das nichts anderes ist als ein J-K-Flipflop mit verbundenen Eingängen:

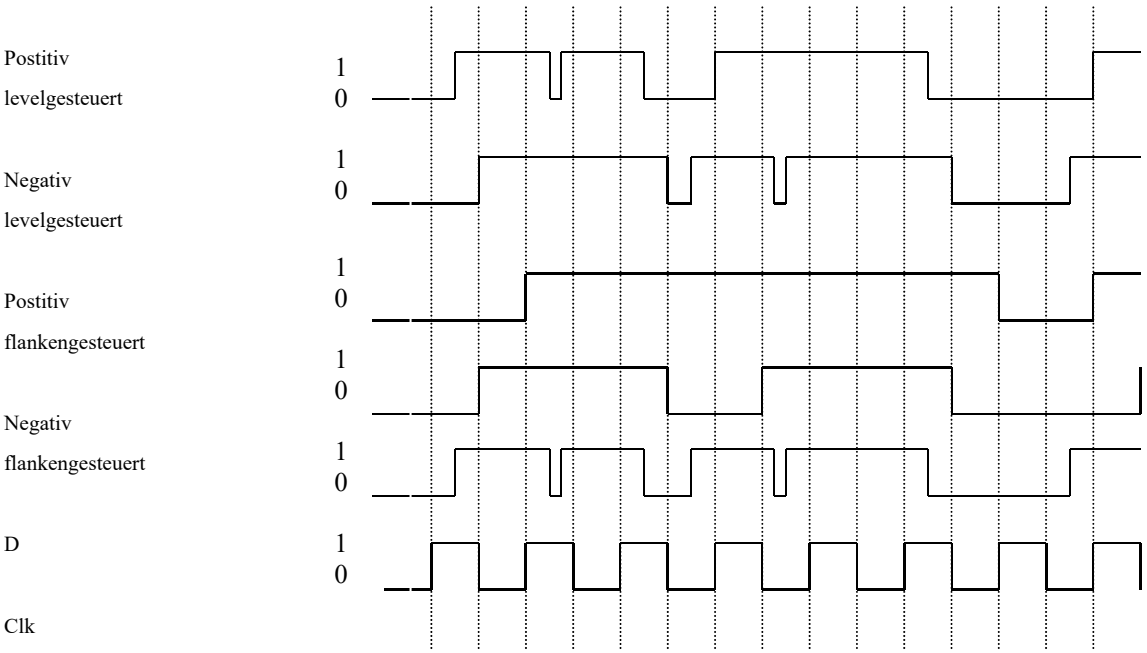
T	Q
0	Q_{alt}
1	$\overline{Q_{alt}}$



Alle diese Systemflipflops übernehmen die Werte des Eingangs während der negativen Taktphase und zeigen sie am Ausgang zu Beginn der positiven Taktphase. Sie wechseln also ihren Zustand mit der **positiven Flanke** des Taktes. Dies ist für Systemflipflops nicht zwingend erforderlich: Wenn man ein Master-Slave-Flipflop so aufbaut, dass der Master während der positiven Phase übernimmt und der Slave während der negativen, so hat man ein Systemflipflop, das bei der negativen Flanke den Zustand ändert. Man spricht auch von positiv oder negativ flankengesteuerten (oder auch flankengetriggerten) Flipflops. Entsprechend wurden Latches in positiv oder negativ pegelgesteuerte (oder auch levelgesteuerte) Flipflops eingeteilt:



Hier sehen wir beispielhaft die Verläufe der Zustandsänderungen dieser vier Flipfloptypen für ein gemeinsames Eingangssignal D:

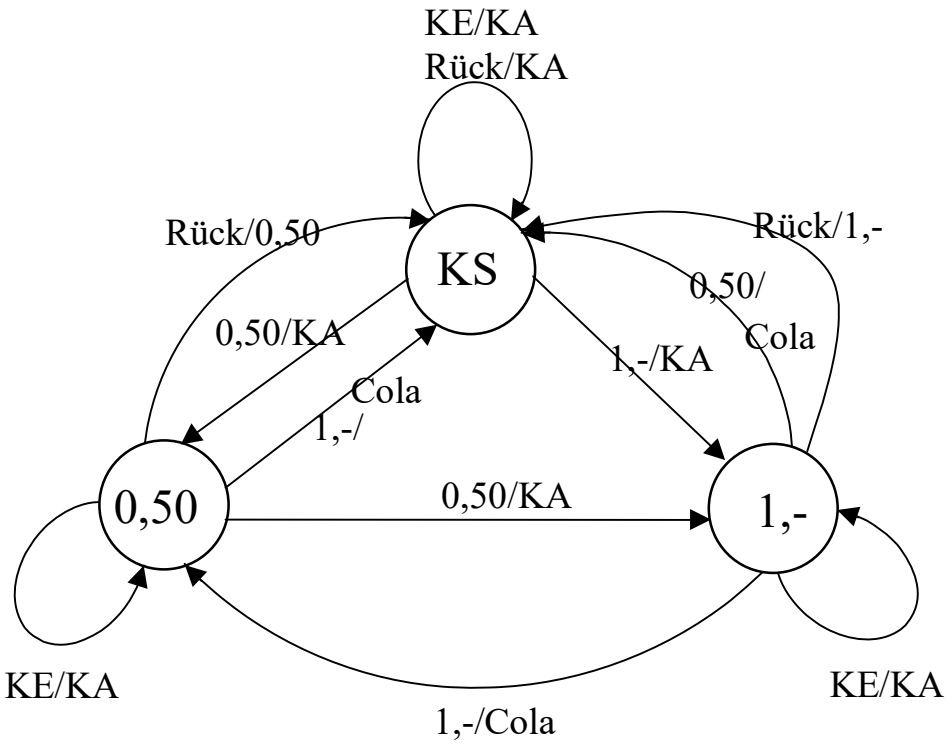


Beispiel: Cola-Automat
Ein Cola-Automat soll formal beschrieben und realisiert werden, an dem man für 1,50 eine Cola kaufen kann. Der Automat akzeptiert 50 Cent- und 1 Euro-Stücke. Sobald ein Betrag von 1,50 oder mehr eingegeben wurde, wird eine Cola ausgegeben. Durch Betätigung des Rückgabeknopfes kann der bereits eingeworfene Guthabenbetrag wieder ausgegeben werden. Da wir von einem getakteten System ausgehen, muss es eine "leere Eingabe" KE (keine Eingabe) und eine "leere Ausgabe" KA (keine Ausgabe) geben.

$A = (X, Y, Z, \delta, \lambda)$ $X = \{ 0.50, 1.-, KE, Rück \}$

$Y = \{ Cola, 0.50, 1.-, KA \}$

$Z = \{ KS, 0.50, 1.- \}$



δ	0.50	1.-	KE	Rück
KS	0.50	1.-	KS	KS
0.50	1.-	KS	0.50	KS
1.-	KS	0.50	1.-	KS

λ	0.50	1.-	KE	Rück
KS	KA	KA	KA	KA
0.50	KA	Cola	KA	0.50
1.-	Cola	Cola	KA	1.-

Codierung:

X	KE	0.50	1.-	Rück
x_1x_0	00	01	10	11

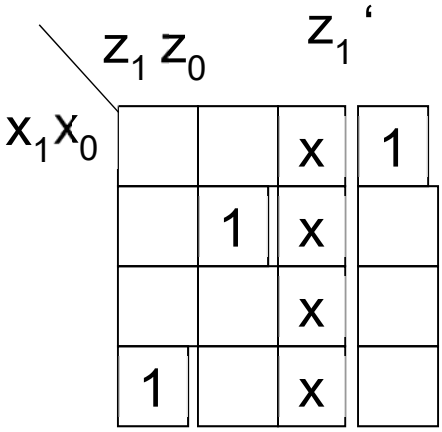
Y	KA	0.50	1.-	Cola
y_1y_0	00	01	10	11

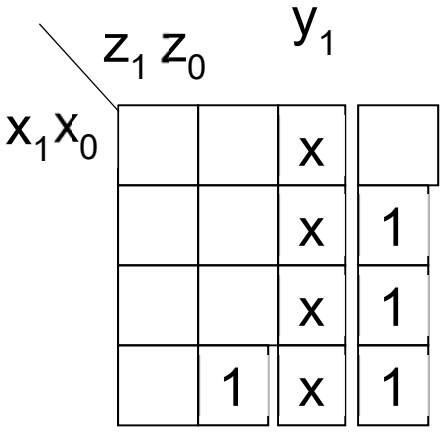
Z	KS	0.50	1.-
z_1z_0	00	01	10

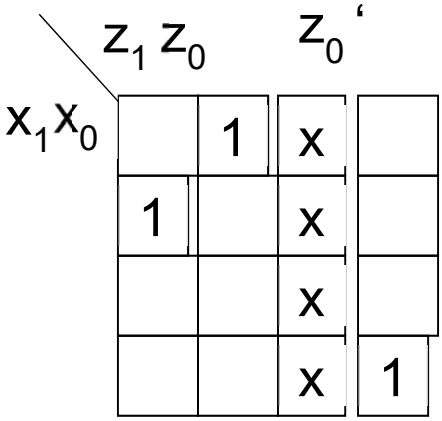
Wertetabelle:

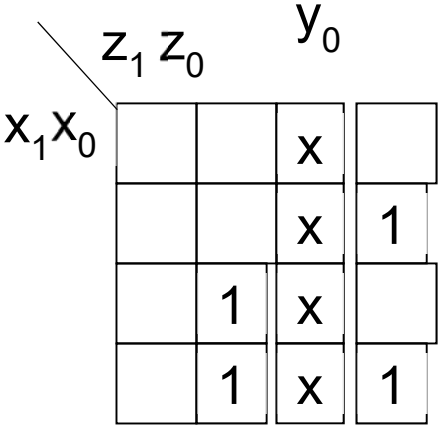
x_1	x_0	z_1	z_0	z_1'	z_0'	y_1	y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	X	X	X	X
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	1
0	1	1	1	X	X	X	X
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	1
1	0	1	0	0	1	1	1
1	0	1	1	X	X	X	X
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0
1	1	1	1	X	X	X	X

6.3 KV-Diagramme:









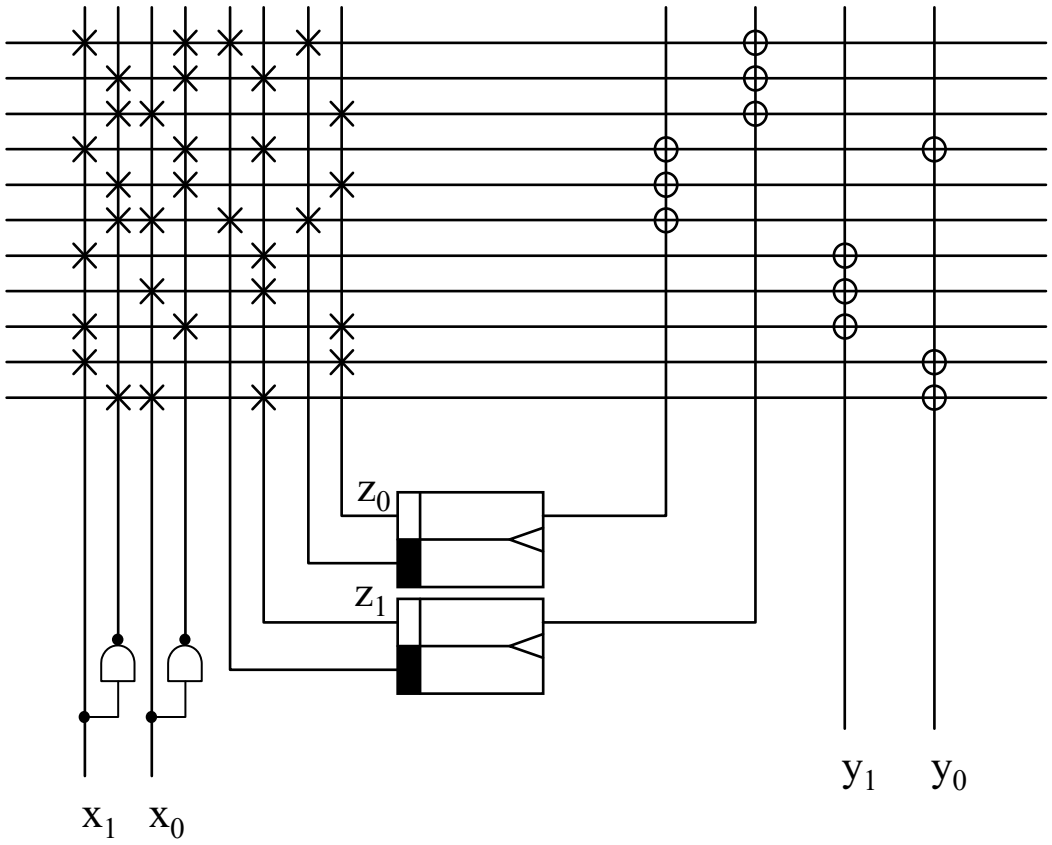
$$z_1' = \overline{x_0} x_1 \overline{z_0}\,\overline{z_1} + \overline{x_0}\, x_1 z_1 + x_0 \overline{x_1} z_0$$

$$z_0' = \overline{x_0} x_1 z_1 + \overline{x_0}\, x_1 z_0 + x_0 \overline{x_1}\,\overline{z_0}\,\overline{z_1}$$

$$y_1 = x_1 z_1 + x_0 z_1 + \overline{x_0} x_1 z_0$$

$$y_0 = x_1 z_0 + \overline{x_0} x_1 z_1 + x_0 \overline{x_1} z_1$$

Realisierung als FPLA:



7 Spezielle Schaltwerke

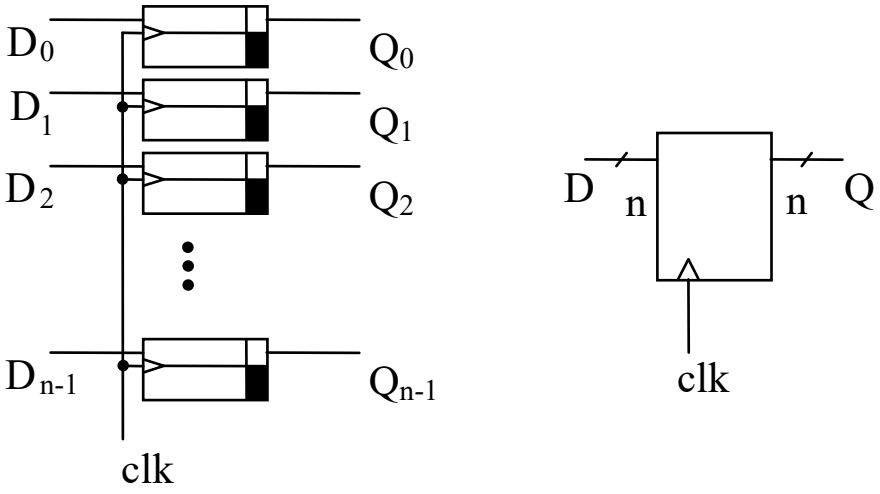
In diesem Abschnitt werden wir einige Schaltwerke kennen lernen, die als Basisbauteile überall im Aufbau digitaler Schaltungen verwendet werden.

7.1 Das Register

Das Register oder der Wortspeicher ist eine Parallelschaltung von n Flipflops. In einem Register kann ein binäres Wort der Länge n gespeichert werden. Die Flipflops können Latches (Auffangflipflops) oder System-Flipflops (z.B. D-Flipflops) sein.

Die folgende Folie zeigt den Aufbau eines Registers und sein Schaltbild.

Register:



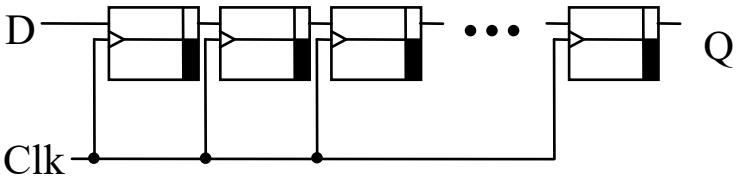
7.1.1 Das Schieberegister

Das Schieberegister ist ein Wortspeicher mit bit-seriellem Zugriff. Es besteht aus einer Serienschaltung von n D-Flipflops. In einem Schieberegister kann ein binäres Wort der Länge n dadurch gespeichert werden, dass pro Takt ein Bit eingegeben wird. Entsprechend wird das gespeicherte Wort in n aufeinanderfolgenden Takten am Ende des Schieberegisters ausgegeben. Die Flipflops müssen System-Flipflops sein.

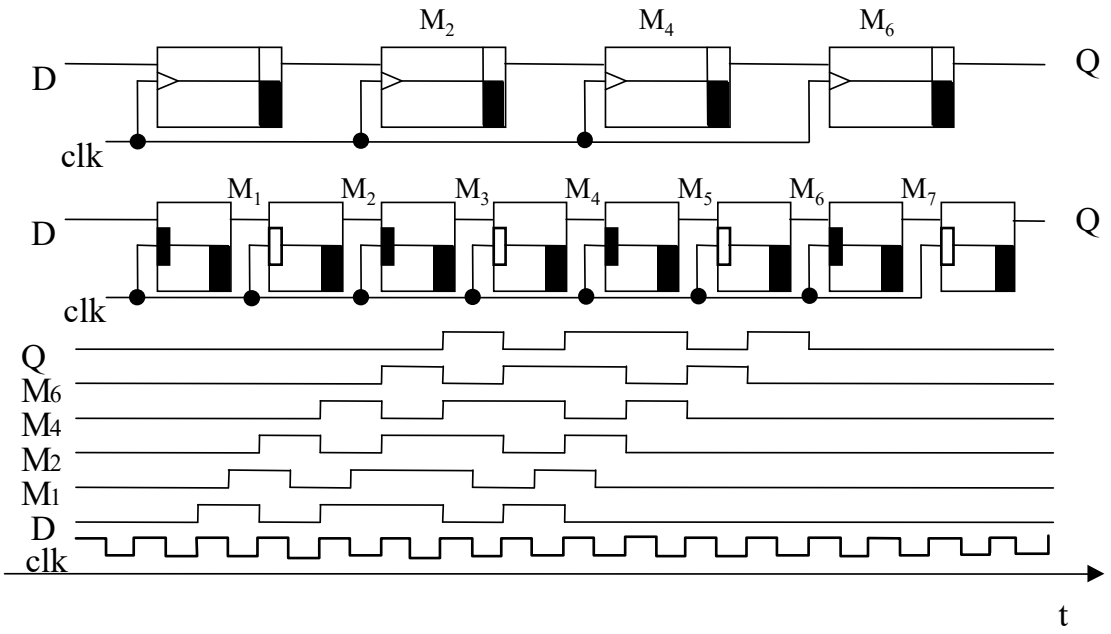
Die folgende Folie zeigt den Aufbau eines Schieberegisters.

Wir wissen bereits, dass man ein System-Flipflop als Master-Slave-Flipflop aus zwei Latches aufbauen kann. Diesen Aufbau und den Verlauf eines Signals 001011010000 entlang des Schieberegisters sieht man auf der übernächsten Folie.

Schieberegister:



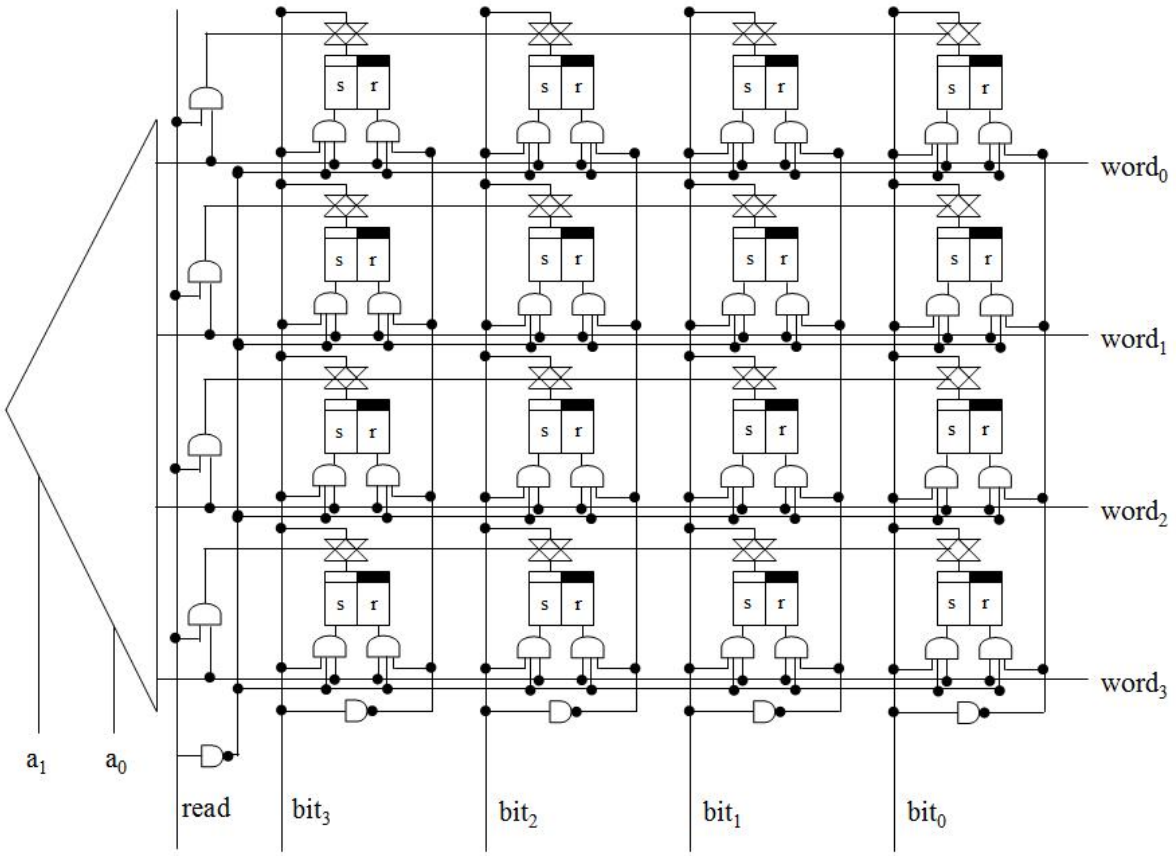
4-Bit-Schieberegister:



7.2 Das RAM

Das Random Access Memory ist ein Speicher für N Worte der Breite m Bit. Jedes dieser Worte ist durch eine Adresse identifizierbar. Die Adresse hat n = log N Bits. Wenn eine Adresse $a_{n-1}a_{n-1}...a_1a_0$ angelegt wird, kann auf das zugehörige Wort lesend oder schreibend zugegriffen werden. Zu diesem Zweck legt man die Adresse an einem Dekodierer an, der sie in einen 1-aus-N-Code dekodiert. Für jedes der N Worte gibt es nun eine sogenannte word-Leitung. Durch den Dekodiervorgang wird auf die gesuchte word-Leitung eine 1 gelegt, auf alle anderen eine 0. An jeder dieser word-Leitungen liegt nun ein Register der Breite m Bit. Durch das Aktivieren der word-Leitung kann dieses Register nun von außen gelesen oder beschrieben werden.

Ein einfacher Aufbau eines RAM mit N = 4 und m = 4 ist auf der nächsten Folie dargestellt. Die Speicherbausteine sind r-s-Flipflops. Jeweils vier davon sind zu einem parallelen Wortspeicher zusammengeschaltet.



7.2.1 Die Funktionsweise des RAM

Beim Schreiben wird an die Bit-Leitungen ein m-Bit-Wort angelegt. Dieses soll in die Zeile i geschrieben werden. Die Adresse i liegt am Decoder an. Dadurch wird die i-te word-Leitung auf 1 gelegt. Die And-Gatter vor den Flipflops der i-ten Zeile lassen somit am s-Eingang jedes r-s-Flipflops den Bit-Wert und am r-Eingang den invertierten Bit-Wert durch. Alle anderen word-Leitungen sind auf 0, d.h. die And-Gatter legen an alle anderen r-s-Flipflops die Speicherkombination r=0 und s=0 an. Somit wird in Zeile i das neue Wort „eingespeichert“. Alle anderen Zeilen speichern die alten Werte.

Beim Lesen wird an die Bit-Leitungen von außen nichts angelegt, sie sind im hochohmigen Zustand Z. Durch die word-Leitung werden nun die Transmission-Gates der Ausgang der Flipflops in der i-ten Zeile geöffnet, die gespeicherten Werte gelangen auf die Bit-Leitungen. Somit werden die Werte der i-ten Zeile an den Ausgang des RAM transportiert.

In der Realität werden RAMs nicht aus r-s-Flipflops und And-Gattern und Transmission-Gates aufgebaut, da diese Realisierung zu aufwendig wäre. Man unterscheidet zunächst zwischen SRAM (statischem RAM) und DRAM (dynamischem RAM). Das statische RAM speichert einen Wert und hält diesen, solange die Versorgungsspannung eingeschaltet bleibt. Unser RAM aus r-s-Flipflops ist ein Beispiel für eine Realisierung eines statischen RAM.

Das dynamische RAM speichert alle Werte nur für eine kurze Zeit. Und zwar wird als Speichermedium die Kapazität eines Transistor-Gates (polychristallines Silizium) gegenüber der Source/Drain (Diffusion) ausgenutzt. Natürlich ist ein solcher Speicher flüchtig. Daher muss jedes Bit in einem dynamischen RAM von Zeit zu Zeit aufgefrischt (refresh) werden. Das geschieht, indem automatisch in festen Zeitabständen zeilenweise alle Bits im RAM einmal gelesen und unverändert wieder geschrieben werden. Ein typisches Zeitintervall für den Refresh-Zyklus bei heutigen DRAMs ist eine Millisekunde.

SRAM ist schneller (kürzere Zugriffszeit) und teurer. Außerdem benötigt SRAM 6 Transistoren pro gespeichertem Bit, während DRAM mit einem Transistor pro Bit auskommt. Daher hat DRAM eine höhere Speicherkapazität pro Chipfläche.

Heutige Speicherchips sind quadratisch angeordnet. Es werden zwei Dekodierer verwendet, einen für die Zeile und einen für die Spalte. Zeilen- und Spaltenadresse sind dabei gleich lang. Auf diese Weise kann man mit der Hälfte der Adresspins auskommen, indem man die Adressleitungen im Zeitmultiplex verwendet. Immer wird zuerst die Zeilenadresse übertragen und dann über dieselben Pins die Spaltenadresse.
Um den Vorteil aus beiden Speicherarten (SRAM und DRAM) zu ziehen, wird heute zunehmend SDRAM verwendet. Dies ist eine Kombination aus beiden Techniken. Es handelt sich im Prinzip um DRAM, bei dem aber die Zeile, aus der zuletzt gelesen wurde in einem kleinen separaten SRAM gehalten wird. Da beim Zugriff auf den Speicher häufig mehrmals hintereinander auf dieselbe Zeile zugegriffen wird, kann jeder Zugriff mit der Geschwindigkeit des SRAM bedient werden. Trotzdem kann die hohe Speicherdichte des DRAM ausgenutzt werden.

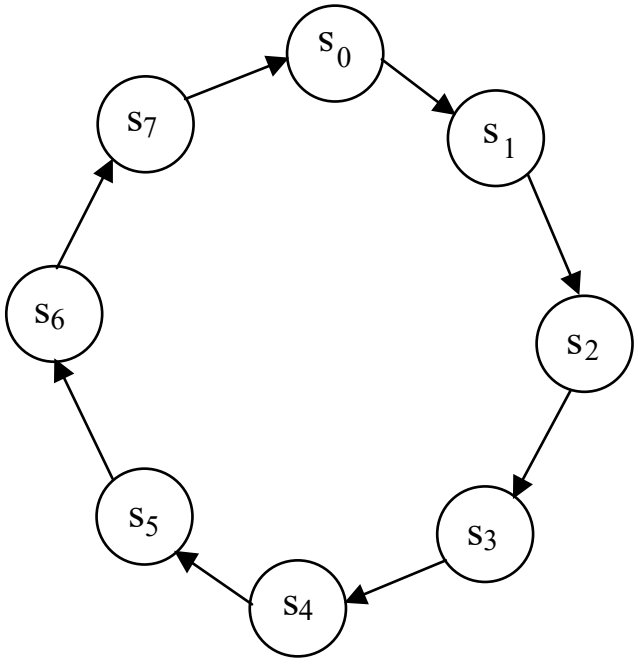
7.3 Zähler

Zähler sind spezielle Schaltwerke, die in der Regel wenige (u.U. gar keine) Eingänge haben. Oft sind auch Ihre Ausgaben identisch mit den Zuständen. Der Zustand ist der Stand des Zählers. Häufig ist der Zählerstand eine binär codierte Zahl. Der Takt bewirkt, dass der Zähler zählt, also seinen Zustand wechselt. Wir zählen also quasi die Anzahl der ansteigenden Taktflanken.

Aus der Sicht der Automatentheorie haben wir bisher alle Schaltwerke als Mealy-Automaten gebaut. Wenn die Ausgabe aber nun nicht von der aktuellen Eingabe sondern nur vom aktuellen Zustand abhängt, so handelt es sich um einen Moore-Automaten. Wenn die Ausgabe identisch mit dem aktuellen Zustand ist, so sprechen wir von einem Medwedew-Automaten.

Wir werden in diesem Kapitel eine Reihe unterschiedlicher Zähler kennen lernen. Als einleitendes Beispiel wählen wir einen synchronen Modulo-8-Zähler. Auf der folgenden Folie ist der Automatengraph für diesen dargestellt: Der Modulo-8-Zähler wandert zyklisch durch 8 Zustände, wobei er bei jedem Takt einen Schritt macht.

Modulo-8-Zähler:

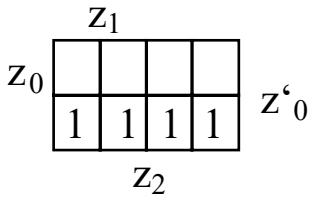


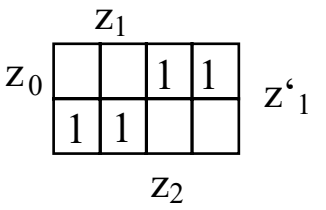
Codierung der Zustände:

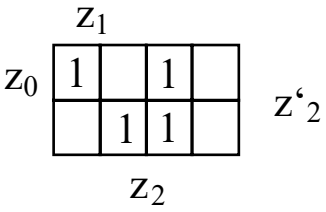
	z_2	z_1	z_0
s_0:	0	0	0
s_1:	0	0	1
s_2:	0	1	0
s_3:	0	1	1
s_4:	1	0	0
s_5:	1	0	1
s_6:	1	1	0
s_7:	1	1	1

Wertetabelle:

z_2	z_1	z_0	z'_2	z'_1	z'_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0





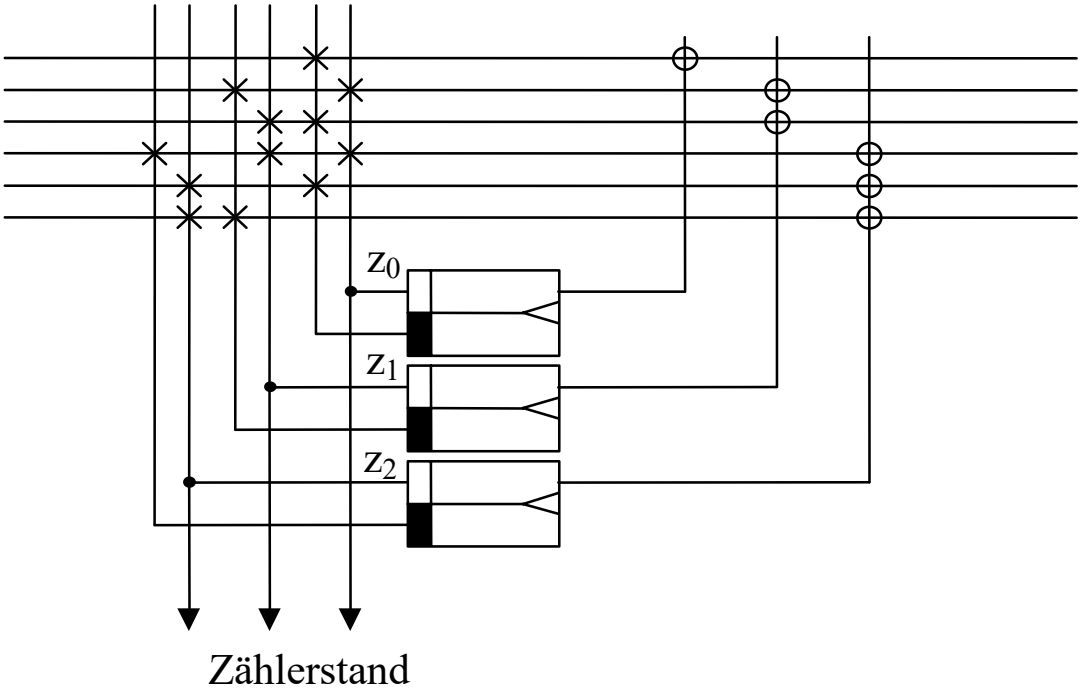


Ergebnis: $z'_0 = \overline{z_0}$

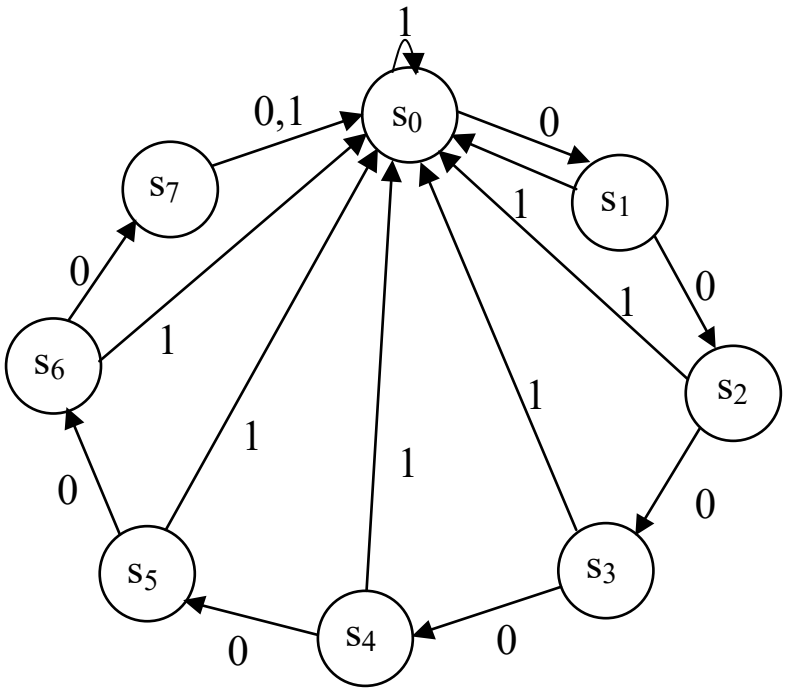
$z'_1 = z_0 z_1 + \overline{z_0}\,\overline{z_1}$

$z'_2 = z_0 z_1 \overline{z_2} + z_0 \overline{z_2} + \overline{z_1} z_2$

Realisierung als FPLA:



Unser Modulo-8-Zähler hat noch einen kleinen Schönheitsfehler: Wir können ihn noch nicht auf einen definierten Anfangszustand setzen. Wenn wir das zusätzlich wollen, braucht das Schaltwerk doch einen Eingang, z.B. ein Signal „reset“. Wenn dieses 1 ist, soll der Zähler in den Zustand s_0 gehen. Der entsprechend veränderte Zustandsgraph sieht dann so aus:



Wertetabelle:

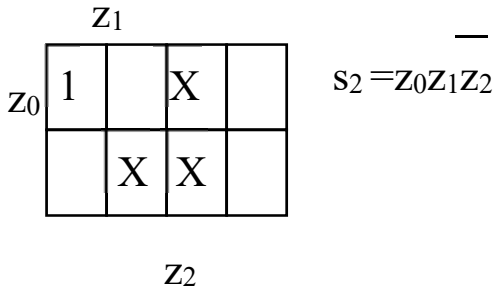
reset	z_2	z_1	z_0	z'_2	z'_1	z'_0
0	0	0	0	0	0	1
0	0	0	1	0	1	0
0	0	1	0	0	1	1
0	0	1	1	1	0	0
0	1	0	0	1	0	1
0	1	0	1	1	1	0
0	1	1	0	1	1	1
0	1	1	1	0	0	0
1	X	X	X	0	0	0

Der Leser möge sich die Realisierung selber überlegen.

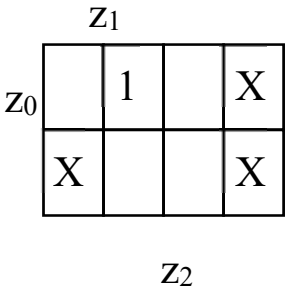
7.3.1 Andere Zähler

Während D-Flipflops in der Regel die geeigneten Bausteine zum Speichern von Zuständen in Schaltwerken sind, werden insbesondere bei Zählern häufig auch andere Flipfloptypen eingesetzt, weil sich dadurch Schaltungsaufwand in den Schaltnetzen einsparen lässt. Das folgende Beispiel ist ein Modulo-8-Zähler, der mit r-s-Flipflops aufgebaut werden soll. In der Wertetabelle können an vielen Stellen „dont cares“ eingesetzt werden, nämlich immer dann, wenn der erwünschte Wert durch „setzen“ oder „speichern“ erreicht werden kann:

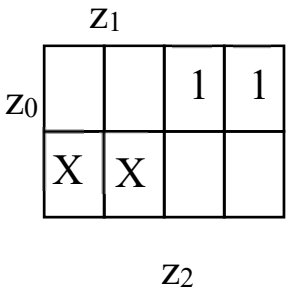
z_2	z_1	z_0	s_2	r_2	s_1	r_1	s_0	r_0
0	0	0	0	X	0	X	1	0
0	0	1	0	X	1	0	0	1
0	1	0	0	X	X	0	1	0
0	1	1	1	0	0	1	0	1
1	0	0	X	0	0	X	1	0
1	0	1	X	0	1	0	0	1
1	1	0	X	0	X	0	1	0
1	1	1	0	1	0	1	0	1



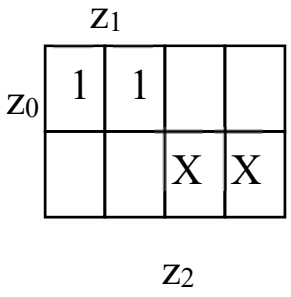
$s_2 = z_0 z_1 \overline{z_2}$



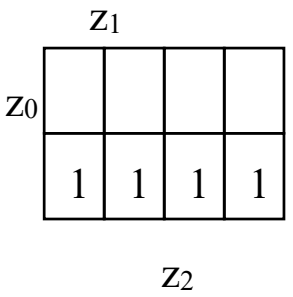
$r_2 = z_0 z_1 z_2$



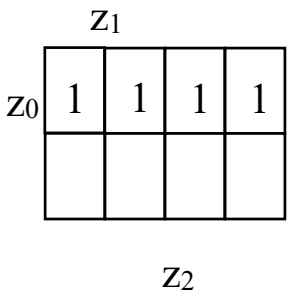
$s_1 = z_0 \overline{z_1}$



$r_1 = z_0 z_1$

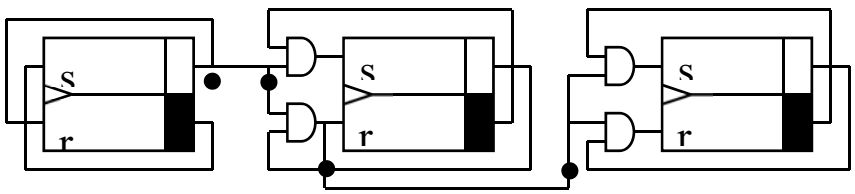


$s_0 = \overline{z_0}$



$r_0 = z_0$

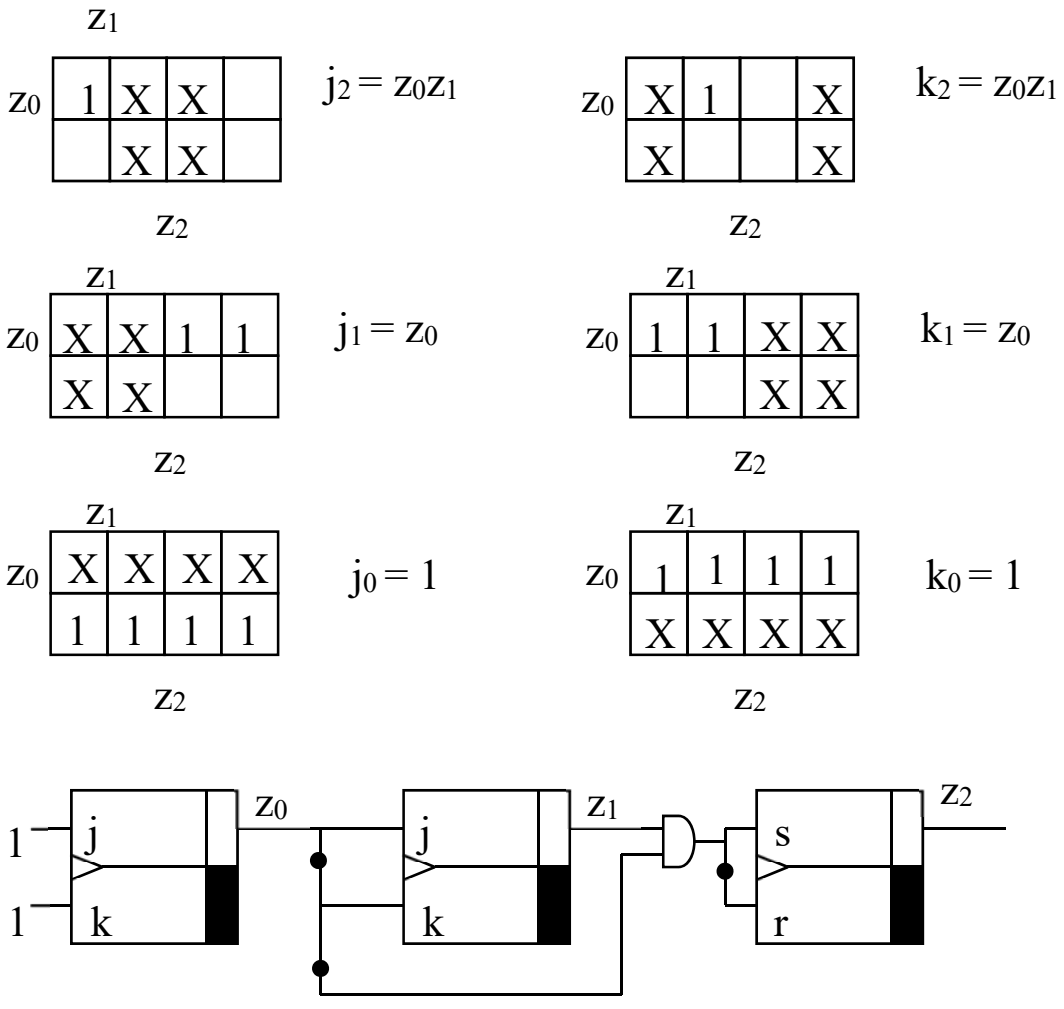
Realisierung:



7.3.2 Einsatz von J-K-Flipflops zum Bau von Zählern

Wir können weiteren Schaltungsaufwand einsparen, wenn wir statt r-s-Flipflops J-K-Flipflops verwenden. Hier kommt uns zu Hilfe, dass die Eingabe j=k=1 zum Toggeln des Zustands genutzt werden kann.

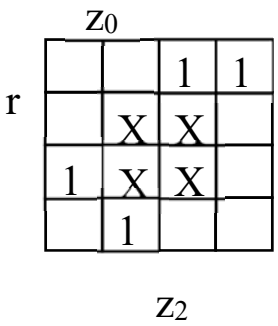
z_2	z_1	z_0	j_2	k_2	j_1	k_1	j_0	k_0
0	0	0	0	X	0	X	1	X
0	0	1	0	X	1	X	X	1
0	1	0	0	X	X	0	1	X
0	1	1	1	X	X	1	X	1
1	0	0	X	0	0	X	1	X
1	0	1	X	0	1	X	X	1
1	1	0	X	0	X	0	1	X
1	1	1	X	1	X	1	X	1



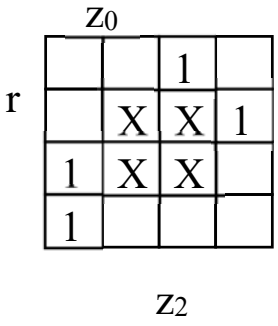
7.3.3 Aufbau eines Modulo-6-vorwärts/rückwärts Zahlers mit T-Flipflops

Dieser Zähler soll über ein Eingabesignal r (für rückwärts) so gesteuert werden, dass er aufwärts (r=0) oder abwärts (r=1) zählen kann. Zu seiner Realisierung sollen T-Flipflops verwendet werden, also solche, die bei Eingabe einer 0 den alten Zustand speichern und bei Eingabe einer 1 den Zustand wechseln (toggeln).

r	z_2	z_1	z_0	T_2	T_1	T_0
0	0	0	0	0	0	1
0	0	0	1	0	1	1
0	0	1	0	0	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	1
0	1	0	1	1	0	1
0	1	1	0	X	X	X
0	1	1	1	X	X	X
1	0	0	0	1	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	1
1	0	1	1	0	0	1
1	1	0	0	1	1	1
1	1	0	1	0	0	1
1	1	1	0	X	X	X
1	1	1	1	X	X	X



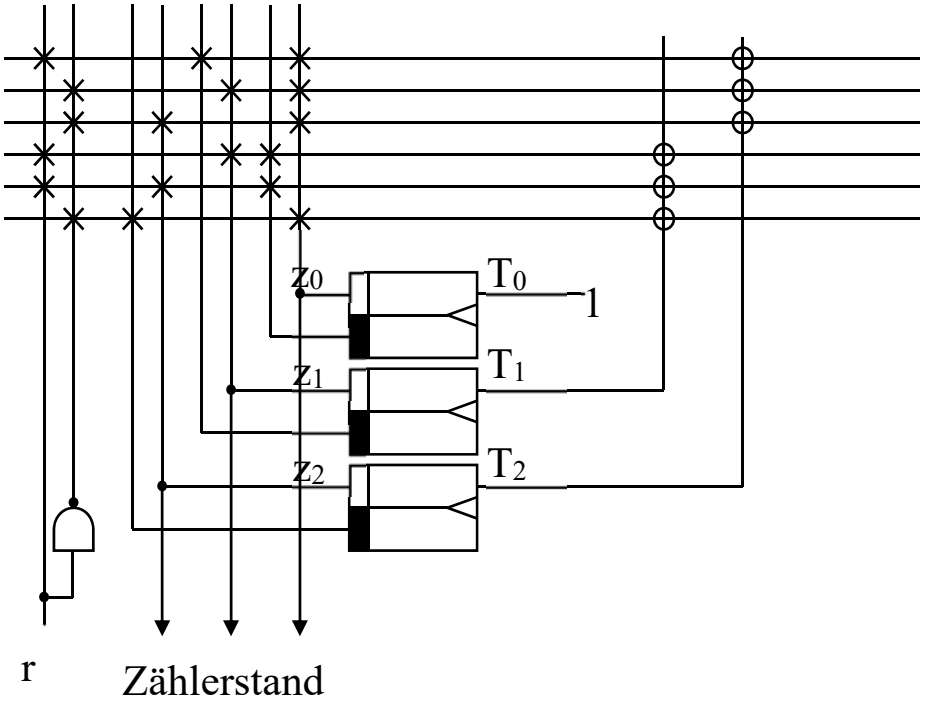
$$T_2 = \bar{z}_0\,\bar{z}_1 r + z_0\,z_1\bar{r} + z_0\,z_2\bar{r}$$



$$T_1 = \bar{z}_0\,z_1 r + \bar{z}_0\,z_2 r + z_0\,\bar{z}_2\bar{r}$$

$$T_0 = 1$$

Realisierung als FPLA:

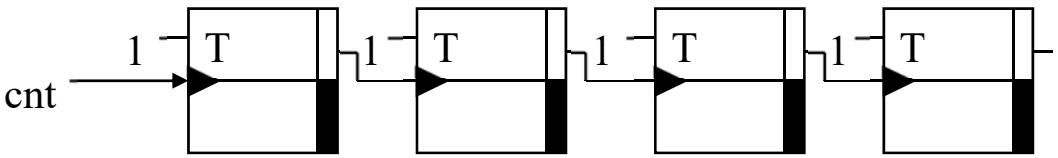


7.3.4 Asynchrone Zähler

Alle Schaltwerke, die wir bisher betrachtet haben, waren synchrone Schaltwerke. Das gilt insbesondere für alle Zähler.

Definition: Ein Schaltwerk heißt synchrones Schaltwerk, wenn alle Takteingänge mit einem einzigen, zentral verteilten Signal, genannt Taktsignal, beschaltet werden.

Es gibt viele gute Gründe, Schaltwerke synchron zu betreiben. Speziell bei den Zählern gibt es aber auch gebräuchliche asynchrone Varianten, die sich durch besonders einfachen Aufbau auszeichnen. Man betrachte den folgenden Modulo-16-Zähler aus T-Flipflops:



Dieses Schaltwerk zählt die Anzahl der Impulse am Eingang cnt. Sei der Zähler am Anfang im Zustand 0000 (gelesen von links nach rechts, so wie die Anordnung der Flipflops in der Zeichnung). Die vier Flipflops sind negativ flankengetriggert. Nach dem ersten Impuls von cnt wird das erste Flipflop getoggelt, der Zählerstand ist 1000. Alle anderen Flipflops haben noch keine negative Flanke am Takteingang gesehen, halten also ihren Zustand. Beim nächsten cnt-Impuls wechselt das erste Flipflop wieder seinen Wert (diesmal von 1 auf 0). Dadurch bekommt das zweite Flipflop seine erste negative Flanke, es toggelt, der Zählerwert ist 0100. Beim nächsten cnt-Impuls geht der Zähler auf 1100 usw.

Wir sehen, dass die Binärzahlen von 0 bis 15 durchgezählt werden (gespiegelt dargestellt). Beim nächsten Impuls würden nacheinander alle Flipflops den Zustand wechseln, der Zähler fängt wieder bei 0000 an.

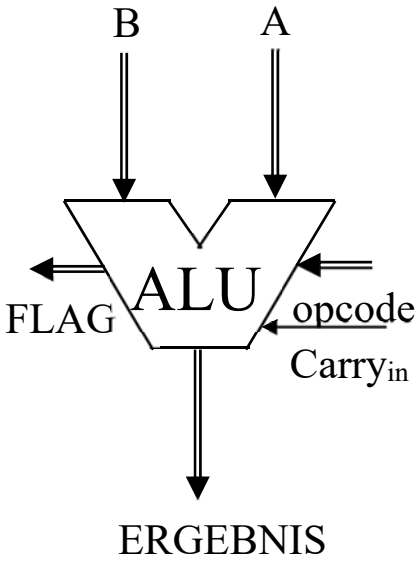
Natürlich funktioniert ein solcher asynchroner Modulo-N-Zähler genauso für alle Zweierpotenzen N mit n = log N Flipflops.

7.4 ALU-Aufbau

Eine ALU (arithmetisch-logische Einheit) besteht in der Regel aus

- Addierer
- Logischer Einheit
- Shifter

Eingänge in eine ALU: zwei Operanden, Instruktionscode



Ausgänge einer ALU: Ergebnis, Flags

7.5 Addierer

Kernstück jeder ALU ist ein Addierer. Wir sehen einen Ripple-Carry-Addierer auf der nächsten Folie.

Wie können wir diesen aber auch zum Subtrahieren benutzen?

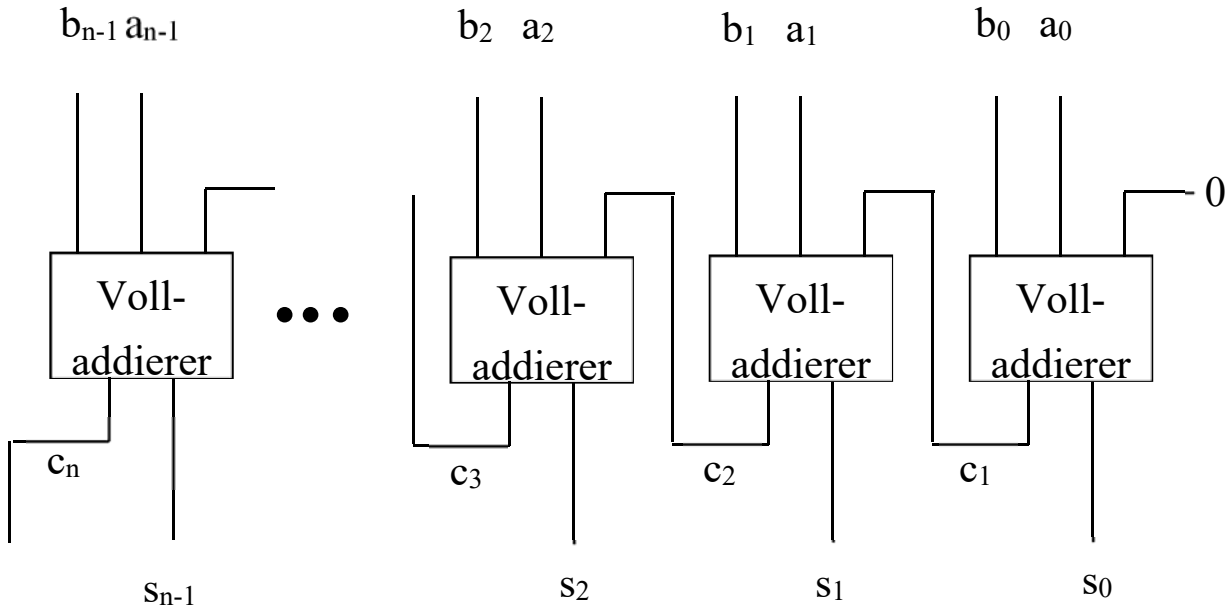
Indem wir das Zweierkomplement des einen Operanden bilden und an den einen Eingang des Addierers führen.

Wie bilden wir das Zweierkomplement?

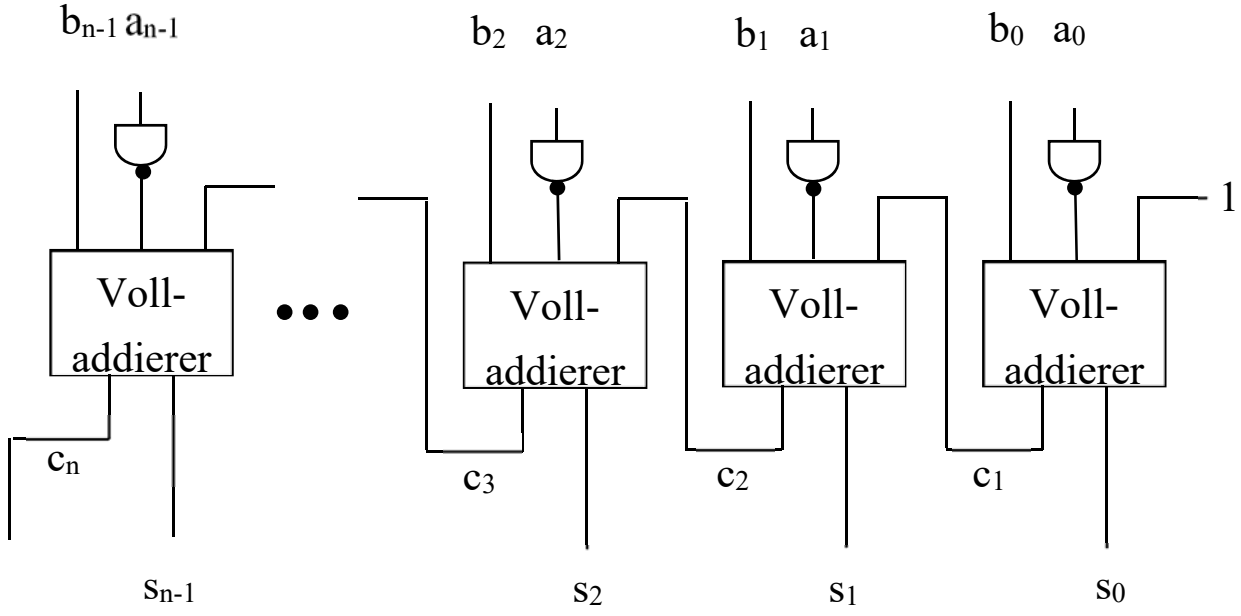
Indem wir jedes Bit invertieren (Einer-Komplement) und noch 1 addieren. Um für diese Addition nicht einen weiteren Addierer zu benötigen, missbrauchen wir einfach den Carry-Eingang des Addierers, in den wir bei der Subtraktion eine 1 anstelle der 0 (bei der Addition) eingeben. Ein entsprechendes Schaltnetz sehen wir auf der übernächsten Folie.

Später werden wir noch weitere Operationen durch den Addierer ausführen lassen. Dadurch wird die Beschaltung seiner Ein- und Ausgänge noch etwas aufwendiger.

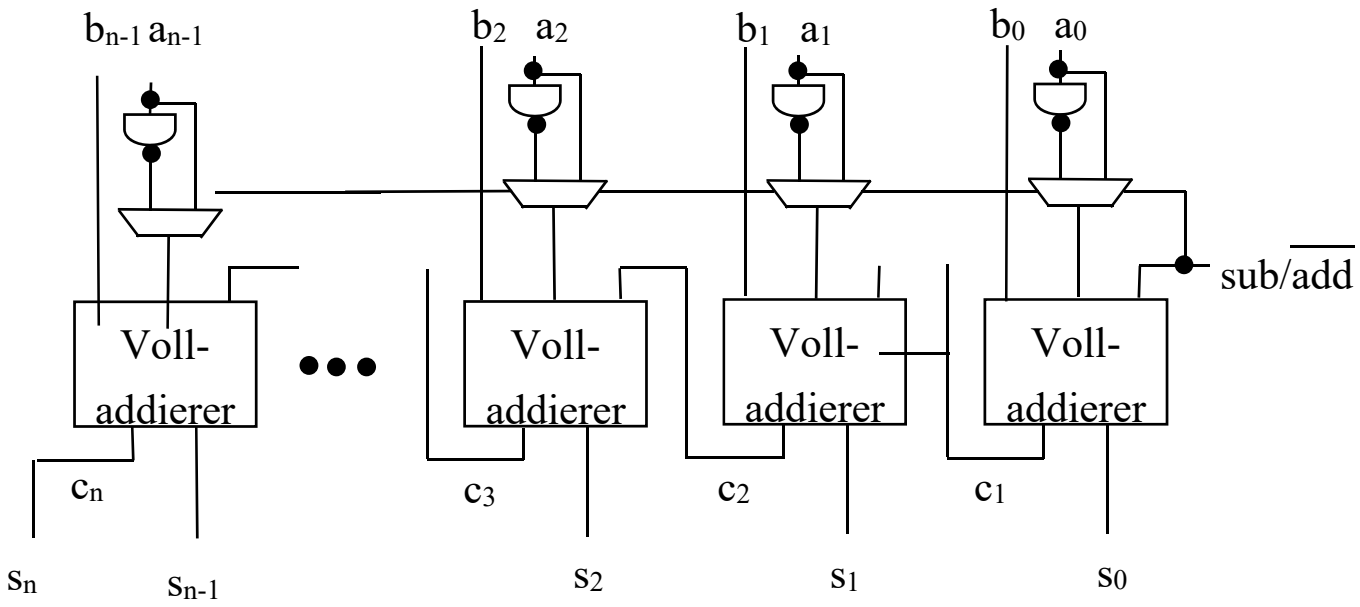
Addierer:



Subtrahierer:



Addierer/Subtrahierer:



7.6 Befehlssatz:

Nun müssen wir uns im Klaren darüber sein, was für einen Befehlssatz wir mit unserer ALU ausführen können wollen. Die folgende Folie zeigt eine typische Auswahl der Operationen, die auf der ALU eines modernen RISC-Prozessors ausgeführt werden können. Man beachte, dass diese Befehlsauswahl einige Redundanz beinhaltet (der SET-Befehl ist zweimal vorhanden, die logischen Befehle könnten anders codiert werden usw.). Wir entscheiden uns für diese einfache Version, um die Implementierung möglichst übersichtlich zu halten.

Die 16 Befehle werden in vier Bits $cntrl_3$,...,$cntrl_0$ codiert. Dabei entscheidet $cntrl_3$, ob es eine arithmetische Operation ist oder nicht und $cntrl_2$ ob eine shift- oder logische Operation bzw. eine Addition oder Subtraktion.

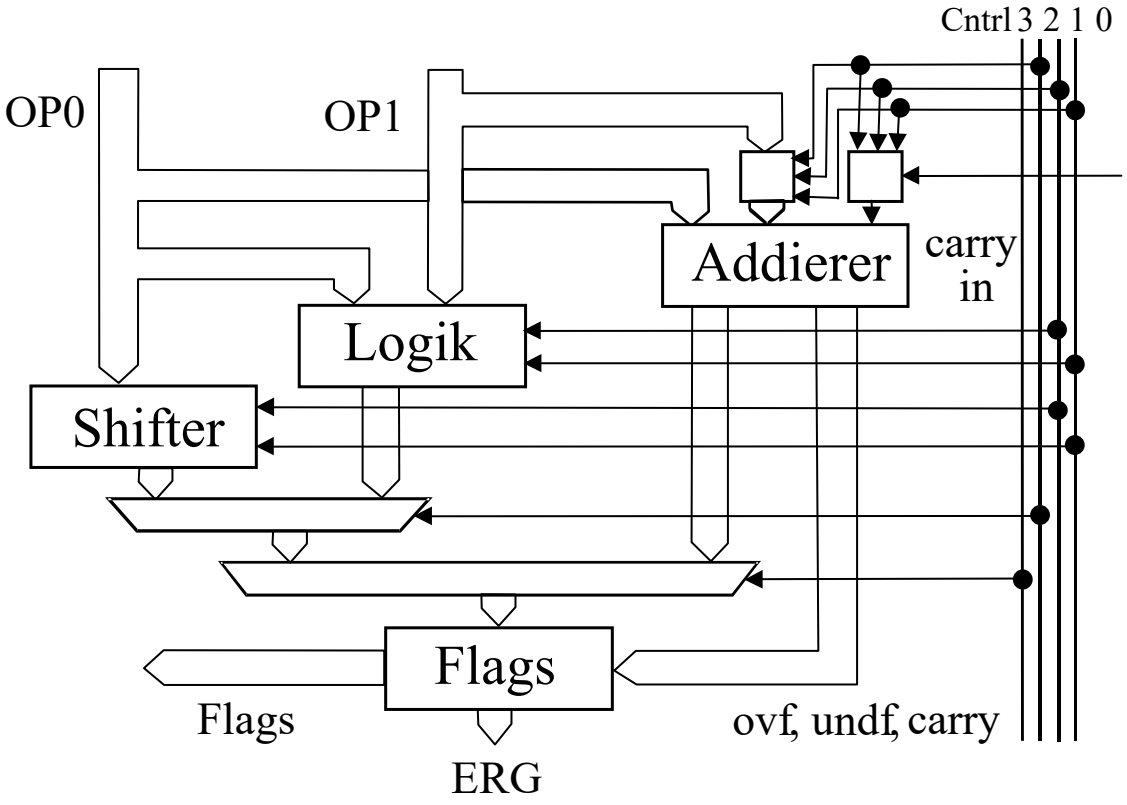
Befehlssatz:

Befehl	**Bedeutung**	**7.6.1.1 Codierung**			
		cntrl3	**cntrl2**	**cntrl1**	**cntrl0**
SET	ERG:=OP0	0	0	0	0
DEC	ERG:=OP0-1	0	0	0	1
ADD	ERG:=OP0+OP1	0	0	1	0
ADC	ERG:=OP0+OP1+$Carry_{in}$	0	0	1	1
SET	ERG:=OP0	0	1	0	0
INC	ERG:=OP0+1	0	1	0	1
SUB	ERG:=OP0-OP1	0	1	1	0
SBC	ERG:=OP0-OP1+$Carry_{in}$	0	1	1	1
SETF	ERG:=0	1	0	0	0
SLL	ERG:=2*OP0	1	0	0	1

SRL	ERG:=OP0 div 2	1	0	1	0
SETT	ERG:=-1	1	0	1	1
NAND	ERG:=OP0 NAND OP1	1	1	0	0
AND	ERG:=OP0 AND OP1	1	1	0	1
NOT	ERG:=NOT OP0	1	1	1	0
OR	ERG:=OP0 OR OP1	1	1	1	1

7.7 Struktur der ALU:

Die nächste Folie zeigt den prinzipiellen Aufbau der ALU. Die Steuereingänge wirken einerseits auf die Einheiten (Shifter, Logik, Addierer) selbst, andererseits wählen sie durch Steuerung zweier Datenweg-Multiplexer das Ergebnis aus der jeweils für die aktuell zuständige Einheit, um es an den ERG-Ausgang der ALU weiterzuleiten.



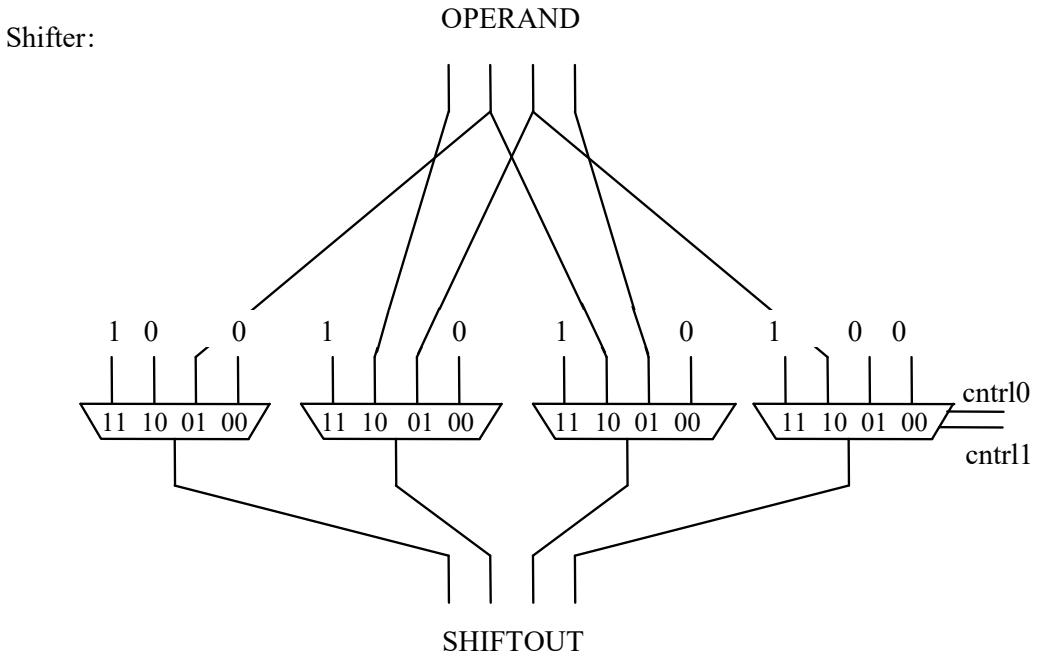
7.7.1 Der Shifter:

Wir wissen bereits, was eine Leitung ist und was ein Datenweg-Multiplexer ist. Es bleibt die Aufgabe, die Einheiten mit Leben zu füllen, von denen wir durch den Befehlssatz zunächst nur eine funktionale Beschreibung haben.

Wir fangen mit der einfachsten Einheit an, dem Shifter. Seine Aufgabe ist, die Befehle SETF (setze auf FALSE, setze auf 0), SLL (shift logical left), SLR (shift logical right) und SETT (setze auf TRUE, setze auf -1). Diese Funktionen können mit einfachen 4-auf 1-Multiplexern wahrgenommen werden, einen für jedes Bit des Ergebnisses. Bei den beiden Shift-Befehlen wird das jeweils neu eingeschobene Bit auf 0 gesetzt und das herausgeschobene Bit wird verworfen.

Wenn ein Ringschieben realisiert werden soll, kann man in der Einheit eine entsprechende Modifikation realisieren.

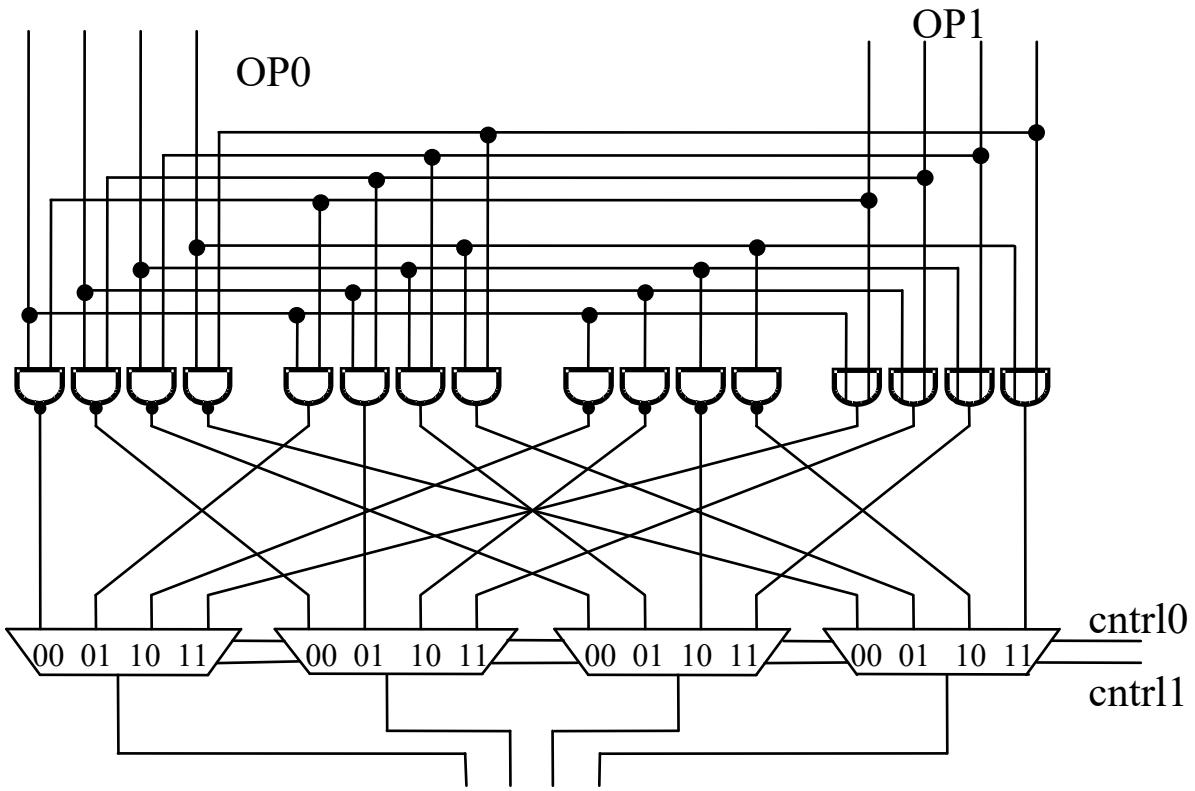
Die folgende Folie zeigt den Shifter für eine Datenbreite von 4 Bit.



7.7.2 Die Logik-Einheit:

Die folgende Folie stellt eine (von vielen möglichen gleichwertige) Realisierung der logischen Befehle dar. NAND, AND, NOT, oder OR werden gesondert bit-weise berechnet und über Multiplexer wird das durch den aktuellen Befehl geforderte Ergebnis ausgewählt.

Die Logik-Einheit:



Eingänge des Addierers

Der Addierer hat als einen Operanden immer OP0. Der zweite zu addierende Operand kann OP1, 0, -1, 1, oder -OP1 sein, je nachdem, ob addiert, subtrahiert, inkrementiert, dekrementiert oder unverändert durchgereicht werden soll. Diese Fälle werden folgendermaßen behandelt:

Bei der einfachen Addition ist der zweite Eingang gleich OP1, der Carry-Eingang gleich 0. Bei der Addition mit Berücksichtigung des alten $Carry_{in}$ ist der Eingang gleich OP1, der Carry-Eingang gleich $Carry_{in}$. Bei der Subtraktion werden alle Bits von OP1 invertiert und der Carry-Eingang ist 1. Auf diese Weise wird das Zweierkomplement von OP1 zu OP0 addiert. Bei der Subtraktion mit Carry werden die Bits von OP1 invertiert und $Carry_{in}$ wird an den Carry-Eingang des Addierers gelegt.

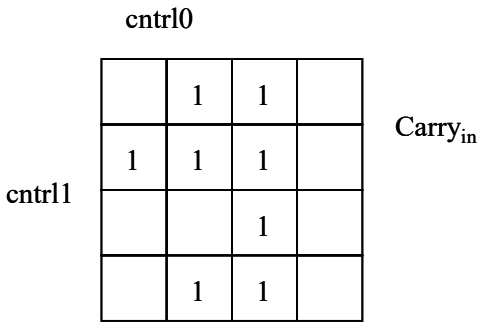
Bei SET-Befehlen wird OP0 + 0 berechnet. Es gibt zwei SET-Befehle. Beim ersten wird +0 addiert, beim zweiten -0 subtrahiert. Also ist beim ersten der zweite Addierer-Eingang gleich 0 und das Carry ist 0 und beim zweiten der zweite Addierereingang auf -1 (alle Bits sind 1) und das Carry auf 1.

Beim Inkrementieren ist der zweite Addierer-Eingang auf 0 und der Carry-Eingang auf 1, beim Dekrementieren ist der zweite Addierer-Eingang auf -1 (alle Bits sind 1) und der Carry-Eingang ist auf 0.

Die Schaltnetze für den Carry-Eingang und den zweiten Eingang des Addierers werden auf den folgenden Folien abgeleitet. Dabei ist das Schaltnetz für den zweiten Eingang des Addierers für jedes Bit einzeln vorzusehen.

C-Eingang des Addierers

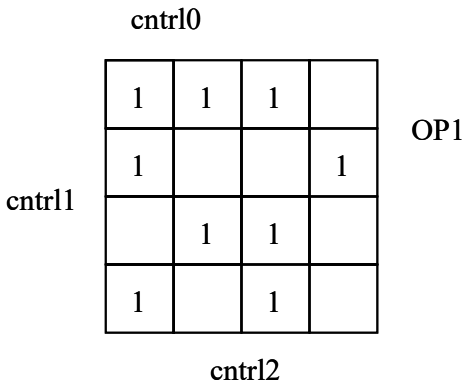
Befehl	$Carry_{in}$	cntrl2	cntrl1	cntrl0	C-Eingang
set	0	0	0	0	0
dec	0	0	0	1	0
add	0	0	1	0	0
adc	0	0	1	1	0
set	0	1	0	0	1
inc	0	1	0	1	1
sub	0	1	1	0	1
sbc	0	1	1	1	0
set	1	0	0	0	0
dec	1	0	0	1	0
add	1	0	1	0	0
adc	1	0	1	1	1
set	1	1	0	0	1
inc	1	1	0	1	1
sub	1	1	1	0	1
sbc	1	1	1	1	1



$$\text{C-Eingang} = \text{cntrl2}\,\overline{\text{cntrl1}} + \overline{\text{cntrl0}}\,\text{cntrl2} + \text{cntrl0}\,\text{cntrl1}\,\text{carry}_{in}$$

OP-Eingang des Addierers

Befehl	OP1	cntrl2	cntrl1	cntrl0	OP-Eingang
set	0	0	0	0	0
dec	0	0	0	1	1
add	0	0	1	0	0
adc	0	0	1	1	0
set	0	1	0	0	1
inc	0	1	0	1	0
sub	0	1	1	0	1
sbc	0	1	1	1	1
set	1	0	0	0	0
dec	1	0	0	1	1
add	1	0	1	0	1
adc	1	0	1	1	1
set	1	1	0	0	1
inc	1	1	0	1	0
sub	1	1	1	0	0
sbc	1	1	1	1	0



$$\text{OP-Eingang} = \text{cntrl0}\,\overline{\text{cntrl1}}\,\text{OP1} + \overline{\text{cntrl0}}\,\text{cntrl2}\,\text{OP1} + \overline{\text{cntrl1}}\,\text{cntrl2}\,\text{OP1} + \text{cntrl1}\,\text{cntrl2}\,\overline{\text{OP1}} + \overline{\text{cntrl0}}\,\overline{\text{cntrl1}}\,\text{cntrl2}$$

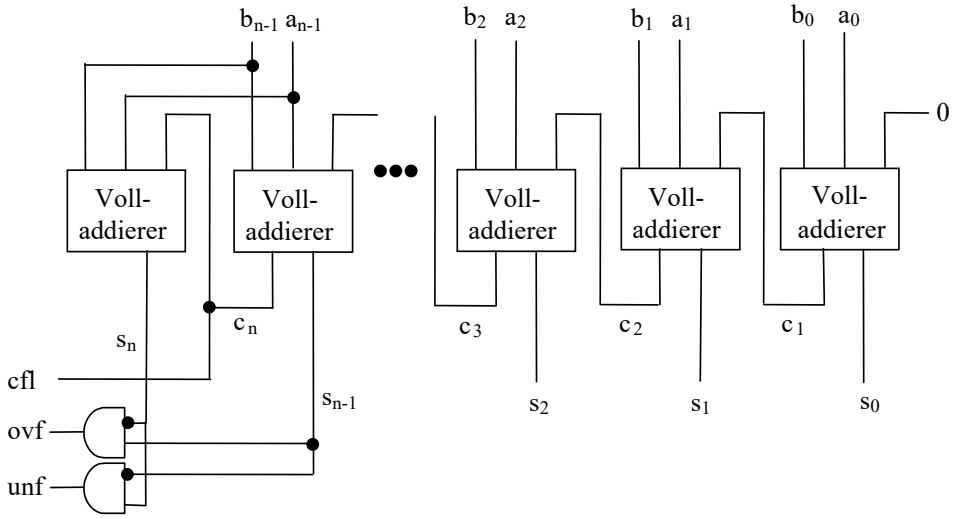
Überlauf-Erkennung beim Addierer

Wie bereits aus der ersten Vorlesung bekannt, können Über- und Unterläufe bei der Addition anhand eines zusätzlichen Sicherungsbits erkannt werden, das eine Kopie des höchstsignifikanten Bits darstellt. Man benutzt nun für eine m-Bit-Addition einen m+1-Bit Addierer,

wobei die Sicherungsstelle ganz normal mitaddiert wird. Das Ergebnis der Addition ist also die Summe s_{m-1}, s_{m-2}, ..., s_1, s_0, das ausgehende Carry-Bit (= Carry-Flag) c_m, sowie ein künstliches Summenbit s_m. Das künstliche Carry-Bit c_{m+1} hat keine Bedeutung und wird daher nicht verwendet. Ein Überlauf (Ergebnis ist größer als die größte darstellbare Zahl) ist nun aufgetreten, wenn s_{m-1} gleich 0 und s_m gleich 1 ist. Wenn s_m gleich 0 und s_{m-1} gleich 1 ist, hat ein Unterlauf (Ergebnis ist kleiner als die kleinste darstellbare Zahl) stattgefunden. Wenn s_{m-1} gleich s_m ist, ist weder Überlauf noch Unterlauf aufgetreten, also ist die Addition fehlerfrei verlaufen.

Das Schaltnetz auf der folgenden Folie zeigt die Ergänzung unseres Addierers, mit der wir Über- und Unterläufe erkennen und als Flags (ovf (overflow) und unf (underflow)) an die Flag-Einheit übermitteln können.

Signale für Flags

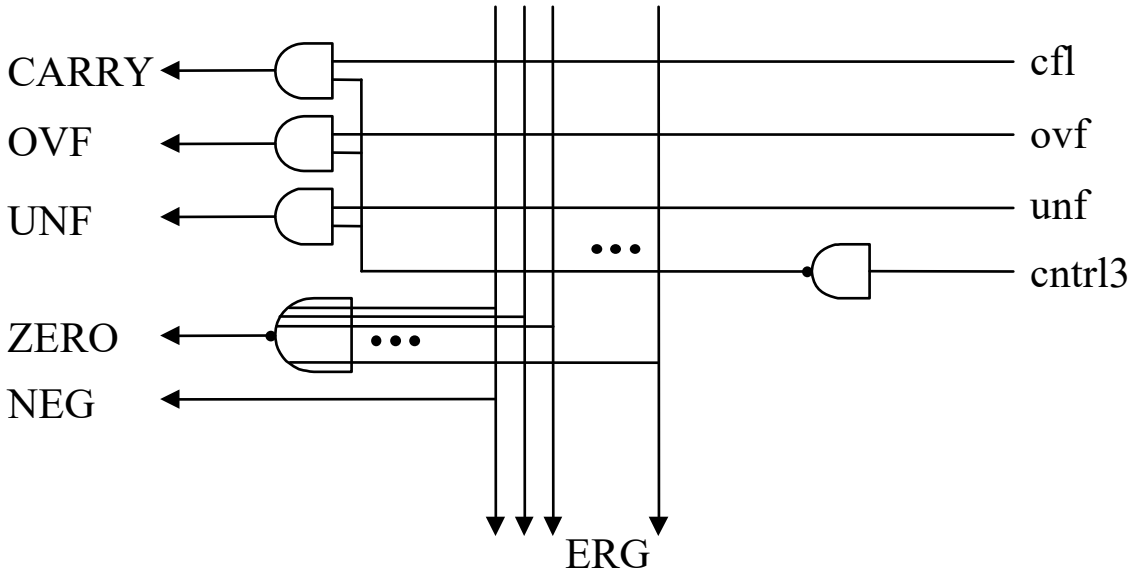


Flag-Test

Fünf Flags sollen generiert werden:

- Carry-Flag (=1, falls eine arithmetische Operation ein Carry erzeugt hat)
- Neg-Flag (=1, wenn das Ergebnis eine negative Zahl darstellt)
- Zero-Flag (=1, wenn das Ergebnis gleich 0 ist)
- ovf-Flag (=1, wenn eine arithmetische Operation einen Überlauf erzeugt hat)
- unf-Flag (=1, wenn eine arithmetische Operation einen Unterlauf erzeugt hat)

Im Einzelfall kann die Anforderung an Flag-Einheiten sehr viel komplizierter sein, insbesondere, wenn es sich um einen Prozessor mit impliziter Condition-Behandlung handelt. Für unsere Zwecke genügt aber eine so einfache Einheit, wie sie auf der folgenden Folie dargestellt ist.


CARRY
OVF
UNF
ZERO
NEG
cfl
ovf
unf
cntrl3
ERG