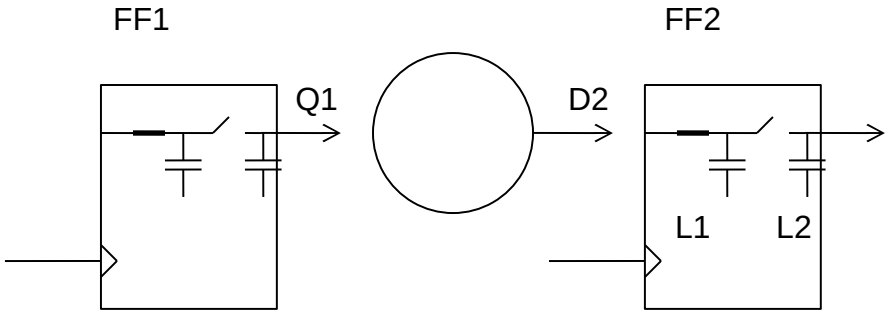
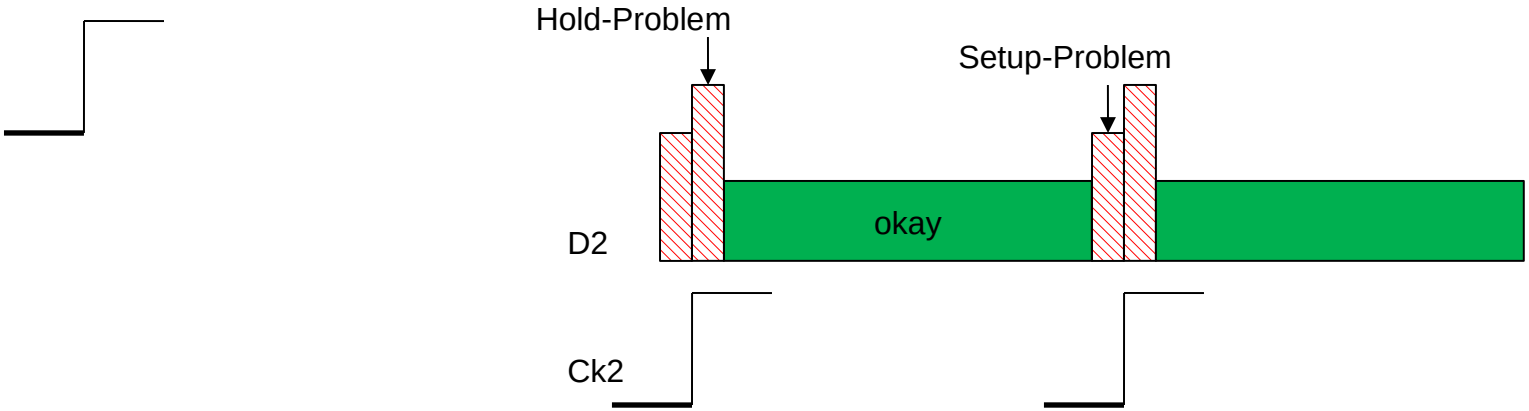
- Design digitaler Schaltkreise
- Setup and Hold
- Kodierer
- Karnough-Tafeln
- Glitch
- Grey Code
- *Statemaschine*

Setup und Hold Zeit

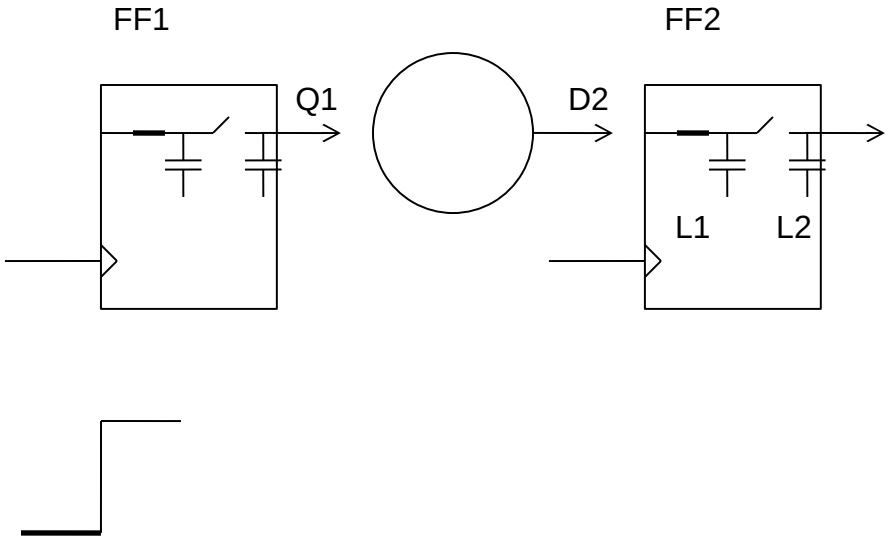
- Setup und Hold Zeit

- Die Änderung am D2 darf nicht zu früh passieren
- Die Änderung am D2 darf nicht passieren bevor das Latch 1/Flipflop 2 in Speichermodus kommt.

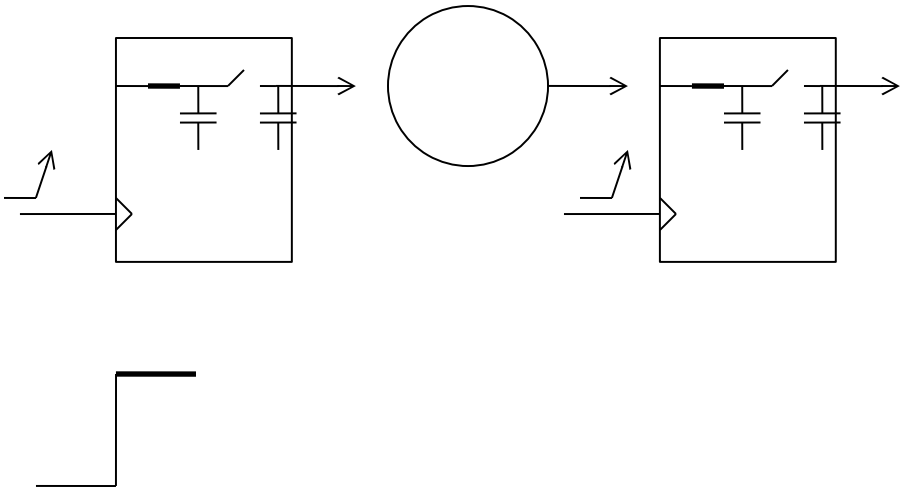




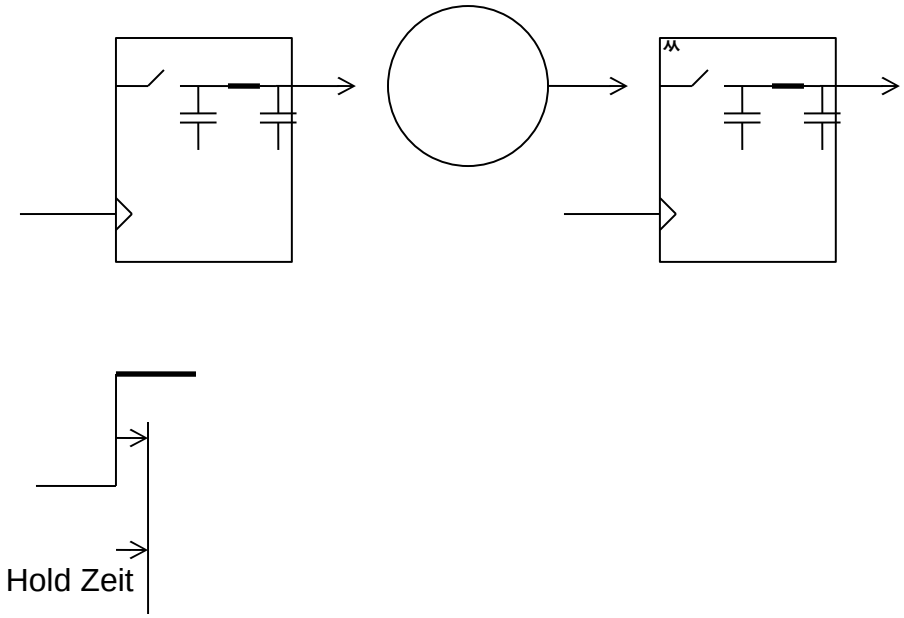
- Wir definieren Hold Time als
- Die Zeit für die der Eingang D2 nach der aktiven Taktflanke unverändert bleiben muss



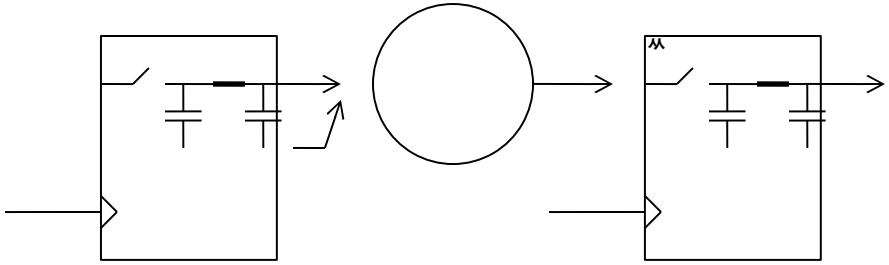
- Hold Time

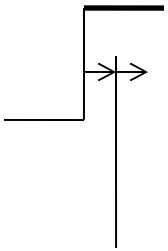


- Hold Time



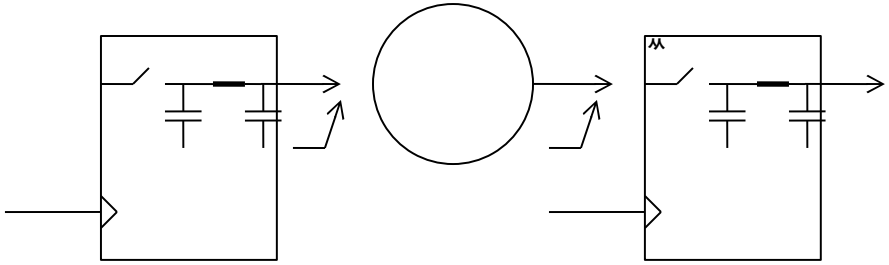
- Hold Time

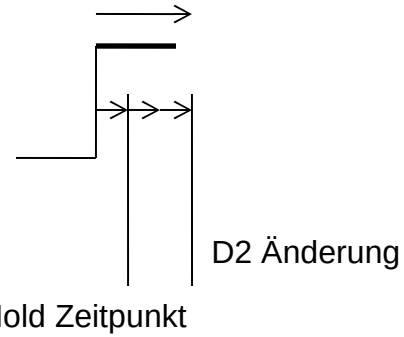




Hold Zeitpunkt

- Hold Time

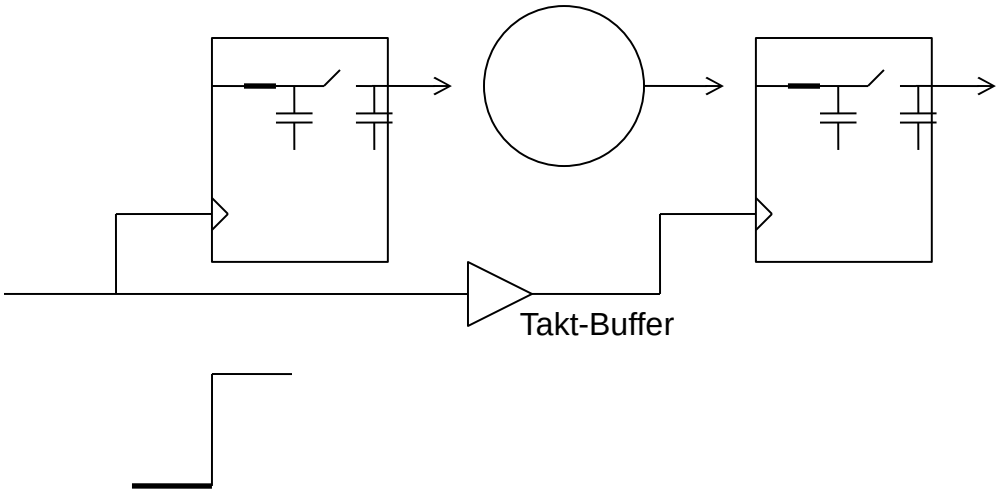




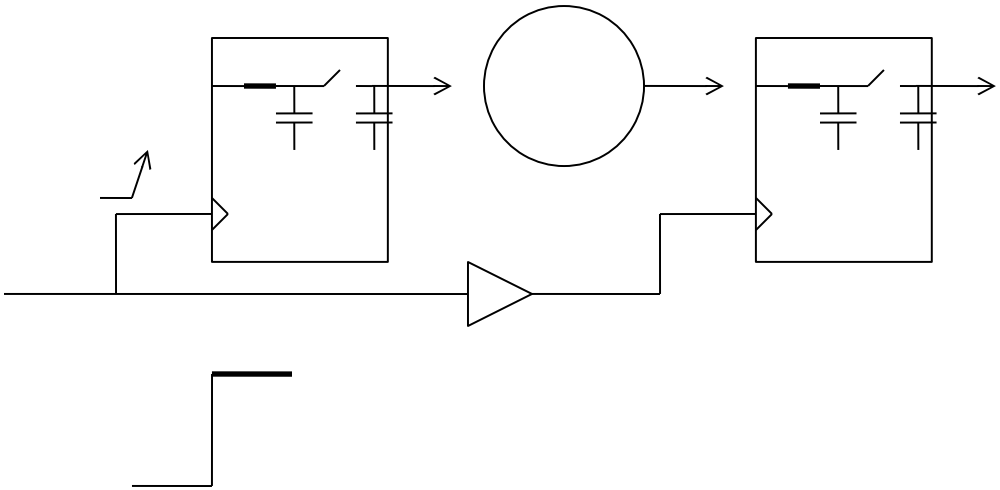
Keine Hold Zeit Violation

Slack = Ck1 + Delay – (Ck2 + Thold)

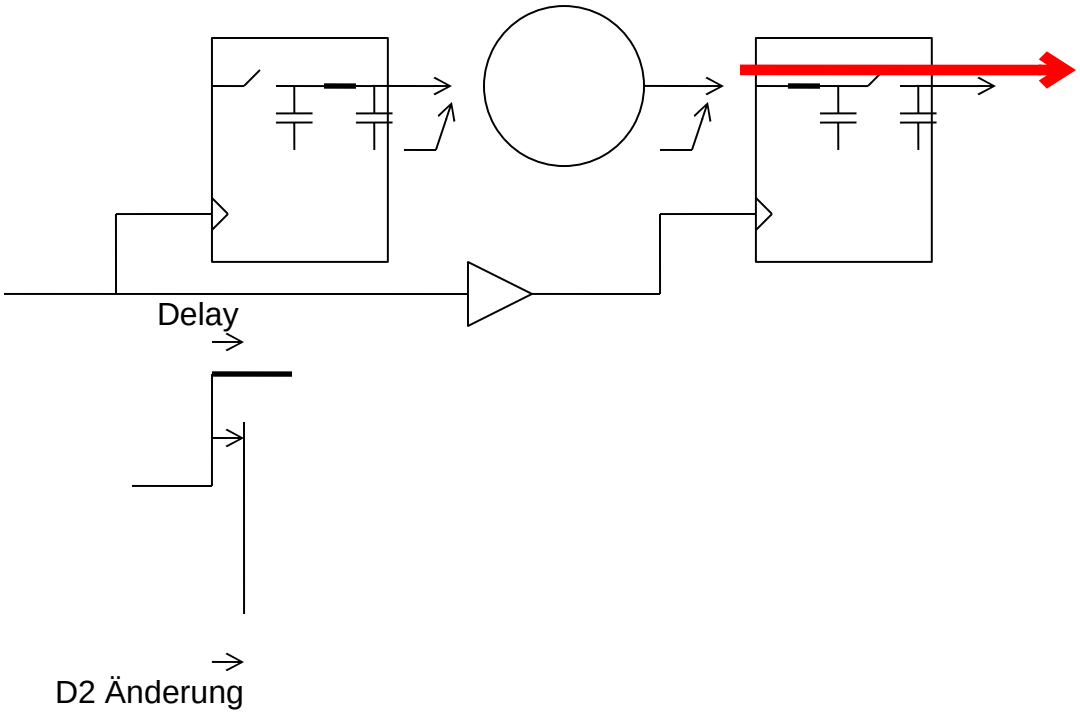
- Hold Time Verletzung



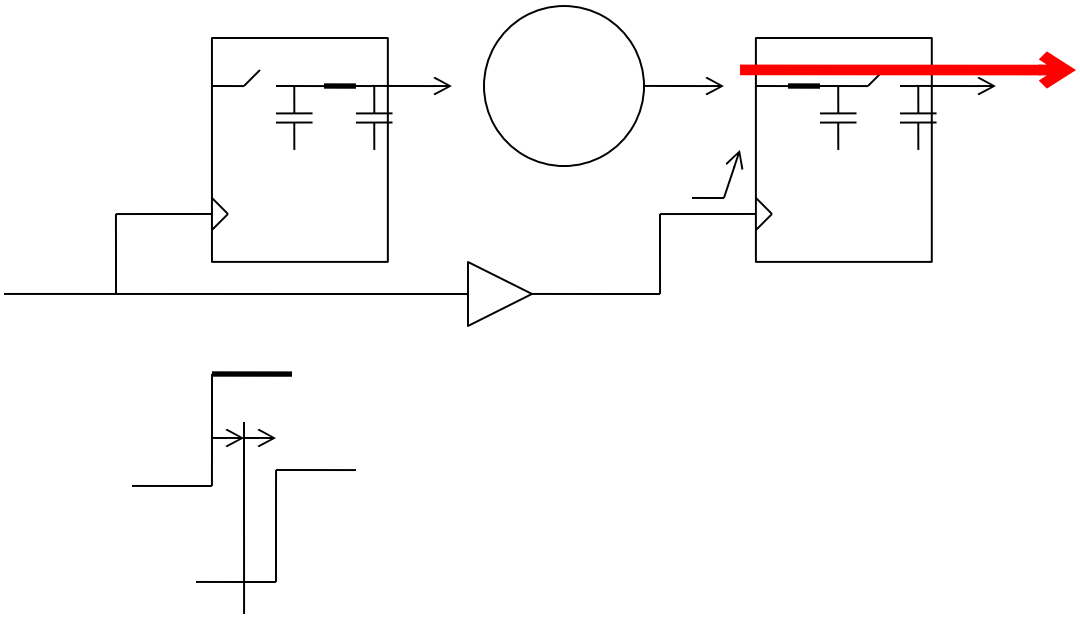
- Hold Time Verletzung



- Hold Time Verletzung

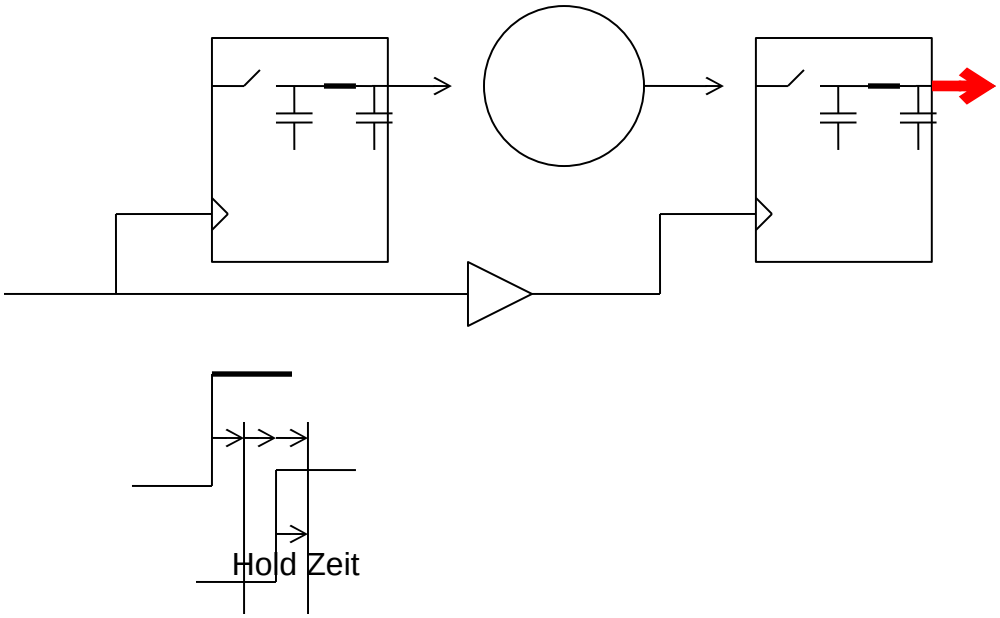


- Hold Time Verletzung



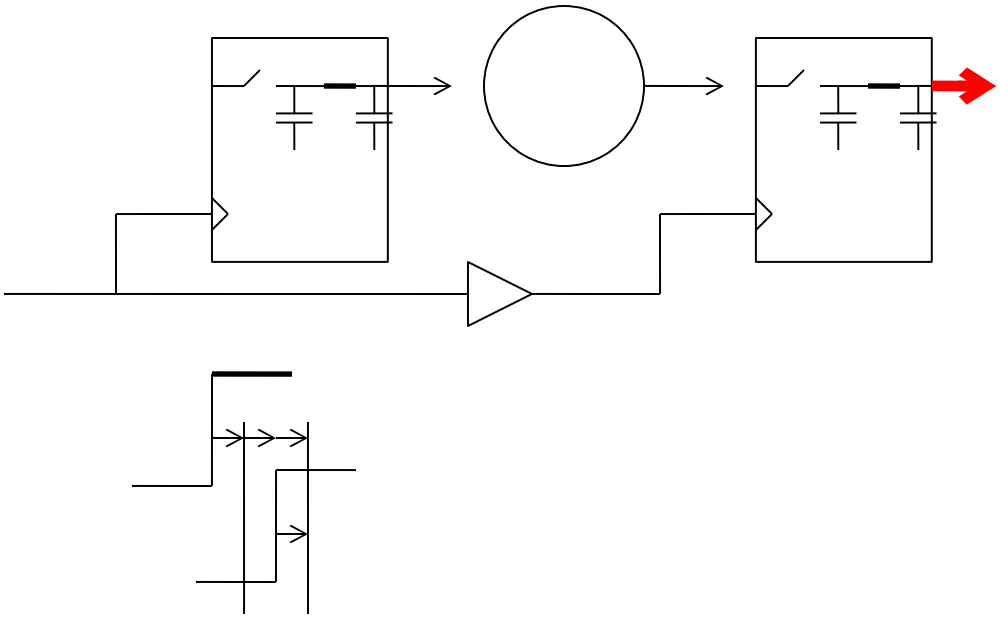
D2 Änderung

- Hold Time Verletzung



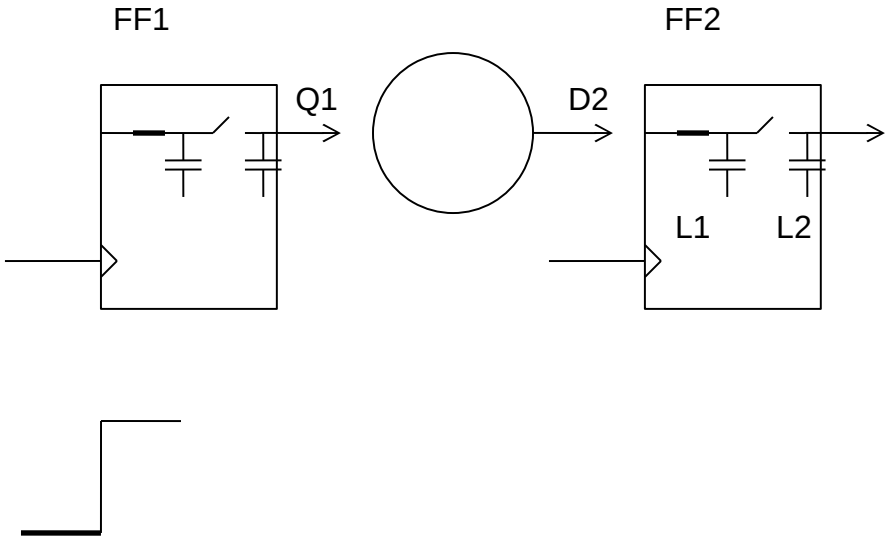
D2 Änderung

- Hold Time Verletzung passiert wenn sich Niveau am Eingang D2 zu schnell ändert. Die Ursache könnte ein schlechtes Design des Flipflops sein oder, dass der Takt Ck2 später ankommt als Ck1. Das letzte könnte bei einem nichtoptimalen Taktbaum passieren. Verzögerung in der kombinatorischen Logik zwischen den Flipflops hilft.

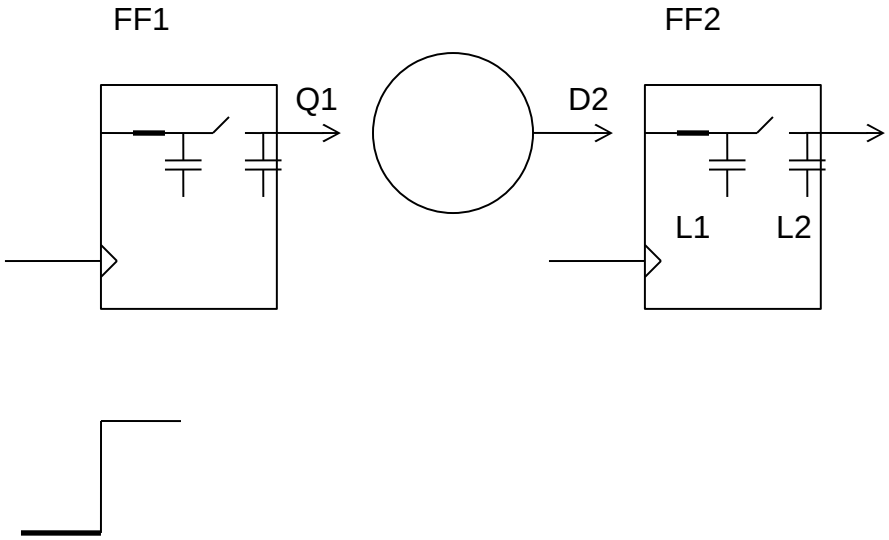


Slack = Ck1 + Delay – (Ck2 + Thold)

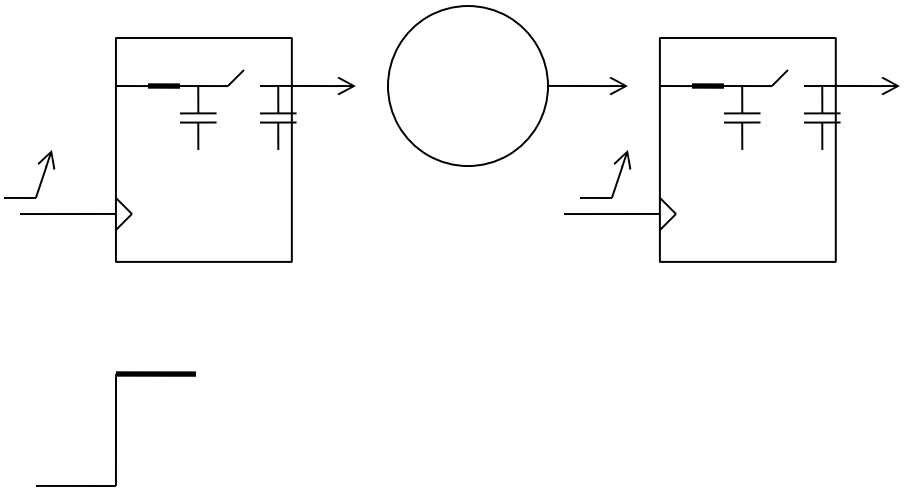
- Die Änderung am D2 soll geschehen eine Weile bevor die nächste Taktflanke Ck(i+1) das Flipflop 2 erreicht (und das Latch 1/FF2 den transparenten Modus verlässt)



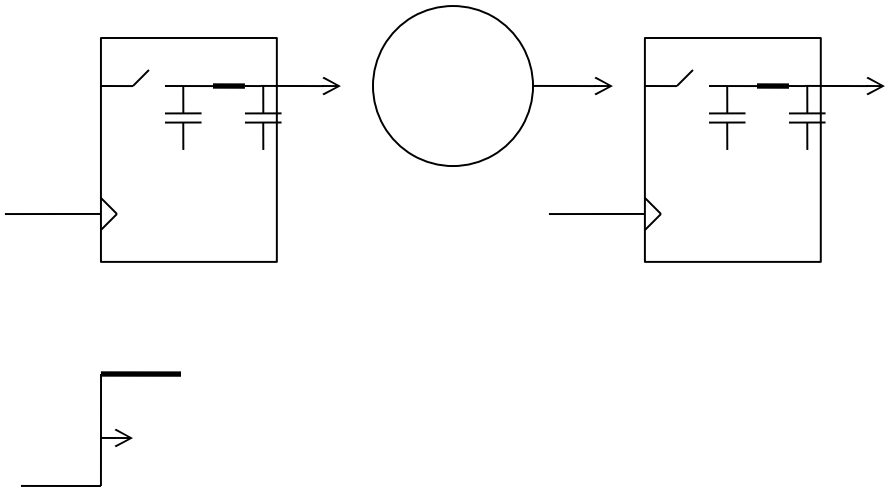
- Wir definieren die Setup-Time als den letzten Zeitpunkt vor der aktiven Taktflanke, wo sich D noch ändern muss so dass die Änderung sicher gespeichert wird.



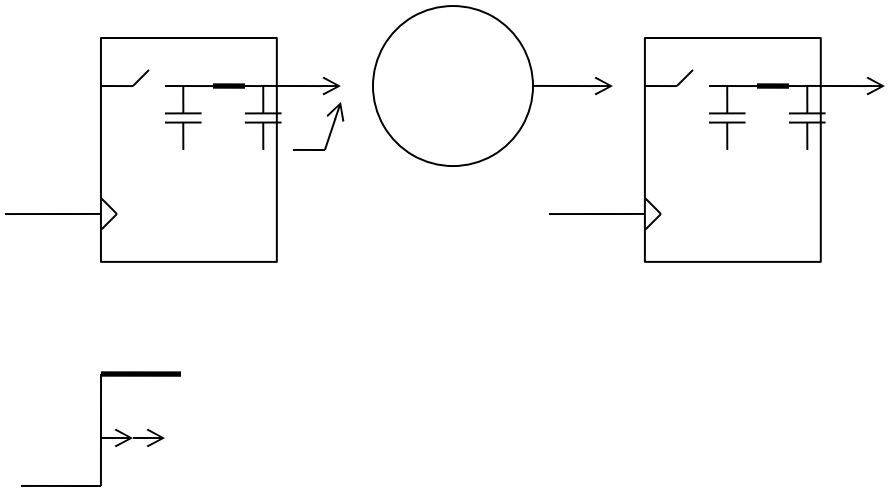
- Setup Zeit



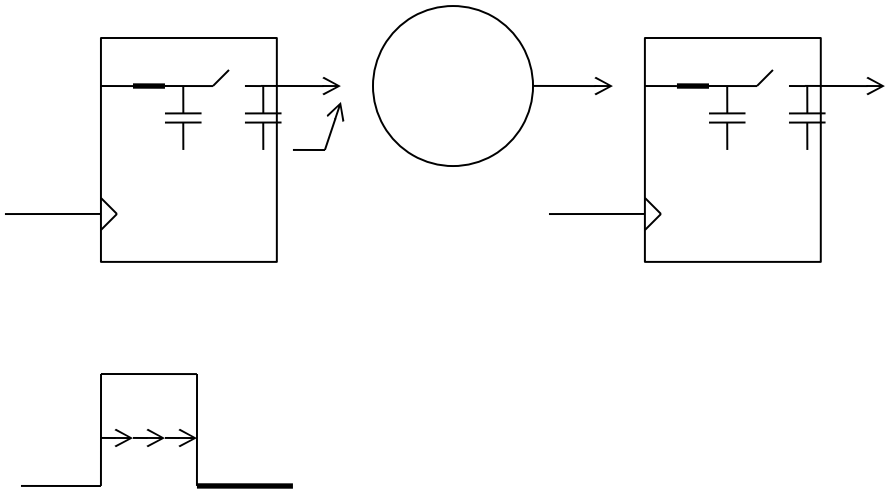
- Setup Zeit



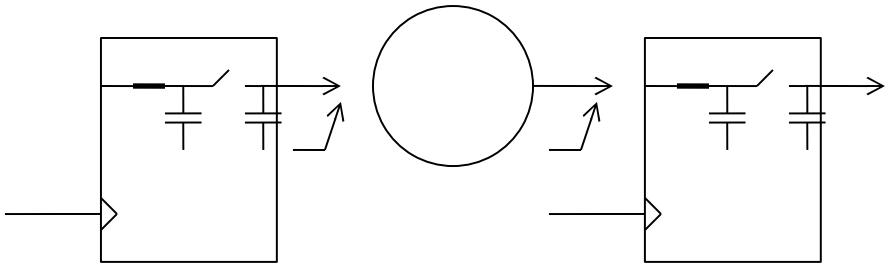
- Setup Zeit



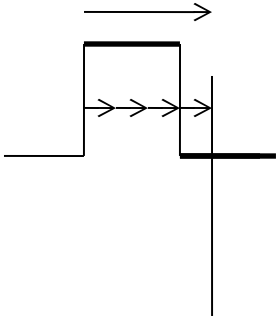
- Setup Zeit



- Setup Zeit

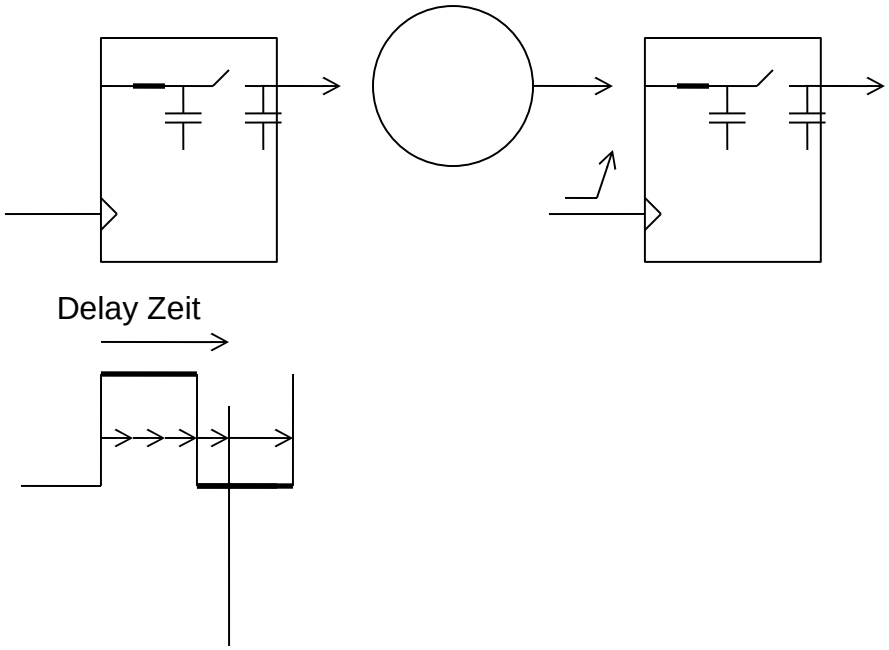


Delay Zeit

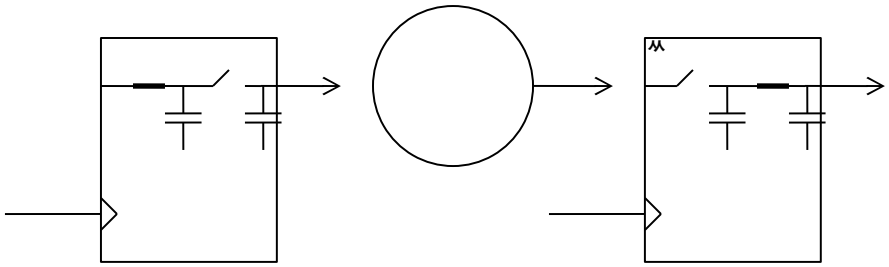


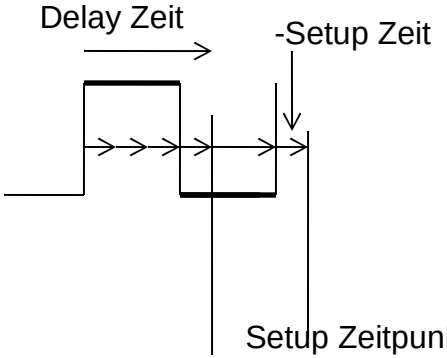
D2 Änderung

- Setup Zeit



- Setup Zeit

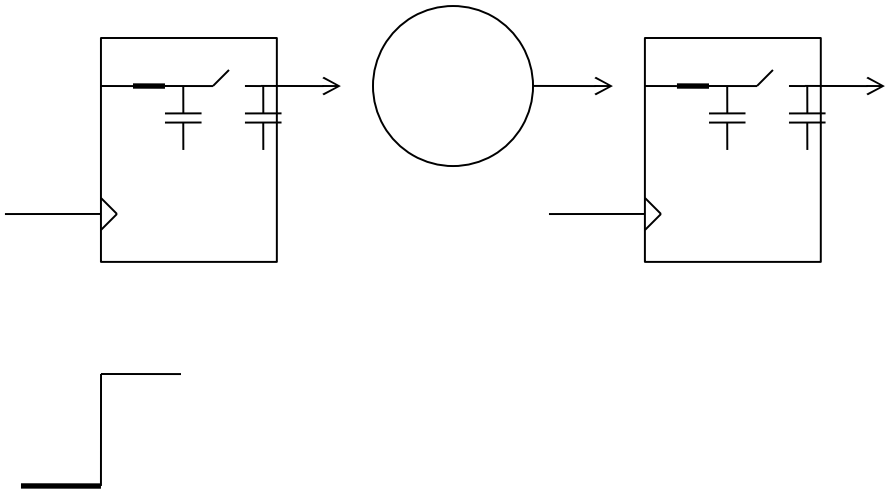




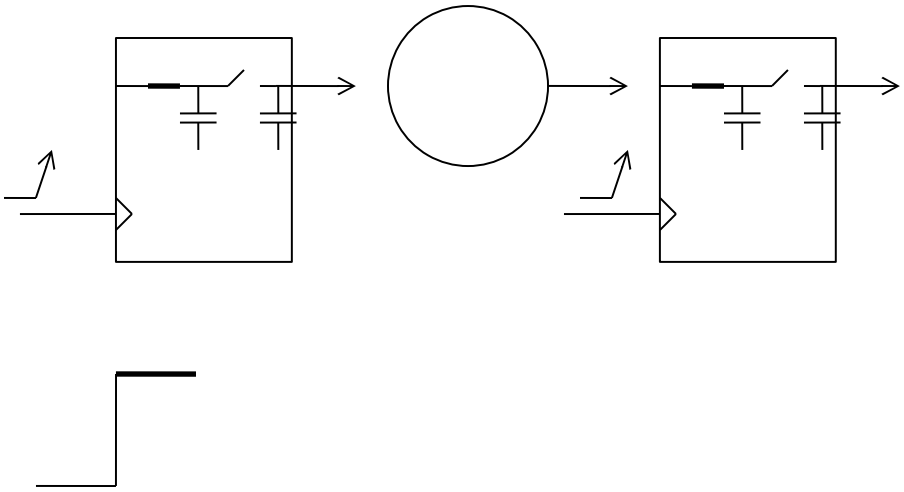
D2 Änderung Keine Setup Zeit Violation

Slack = Ck2 - Tsetup - (Ck1-Tck) - Delay

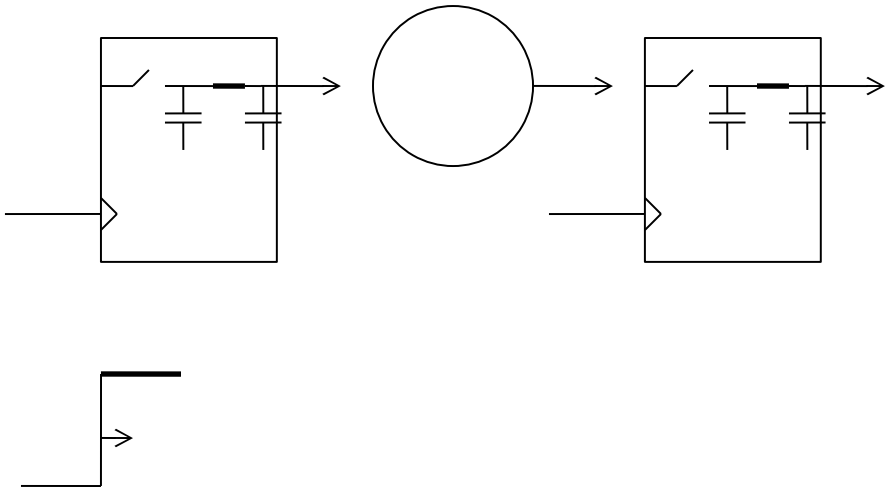
- Setup Zeit Verletzung



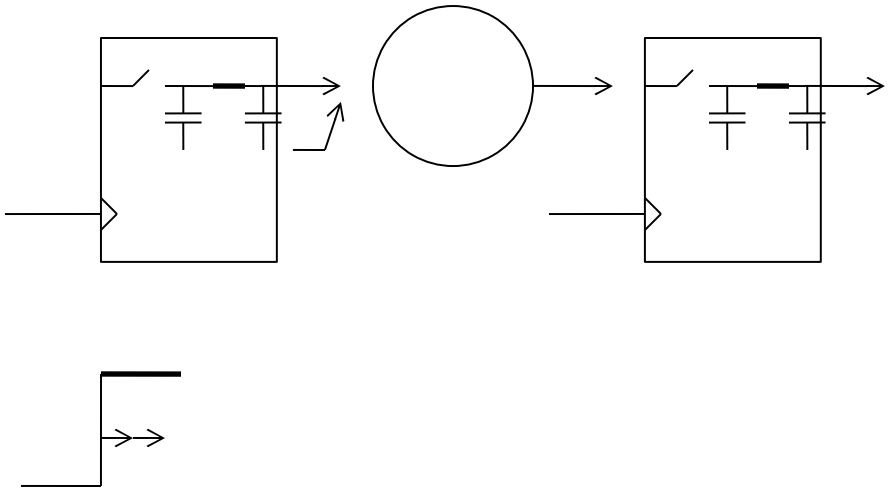
- Setup Zeit Verletzung



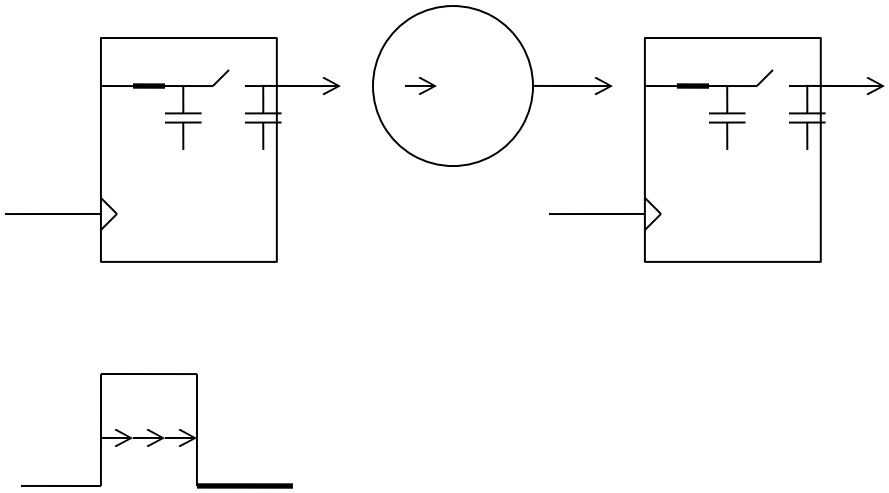
- Setup Zeit Verletzung



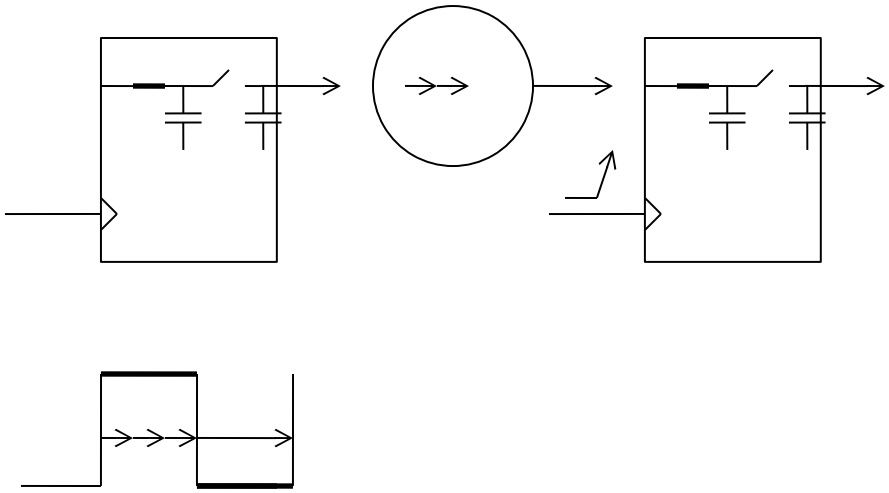
- Setup Zeit Verletzung



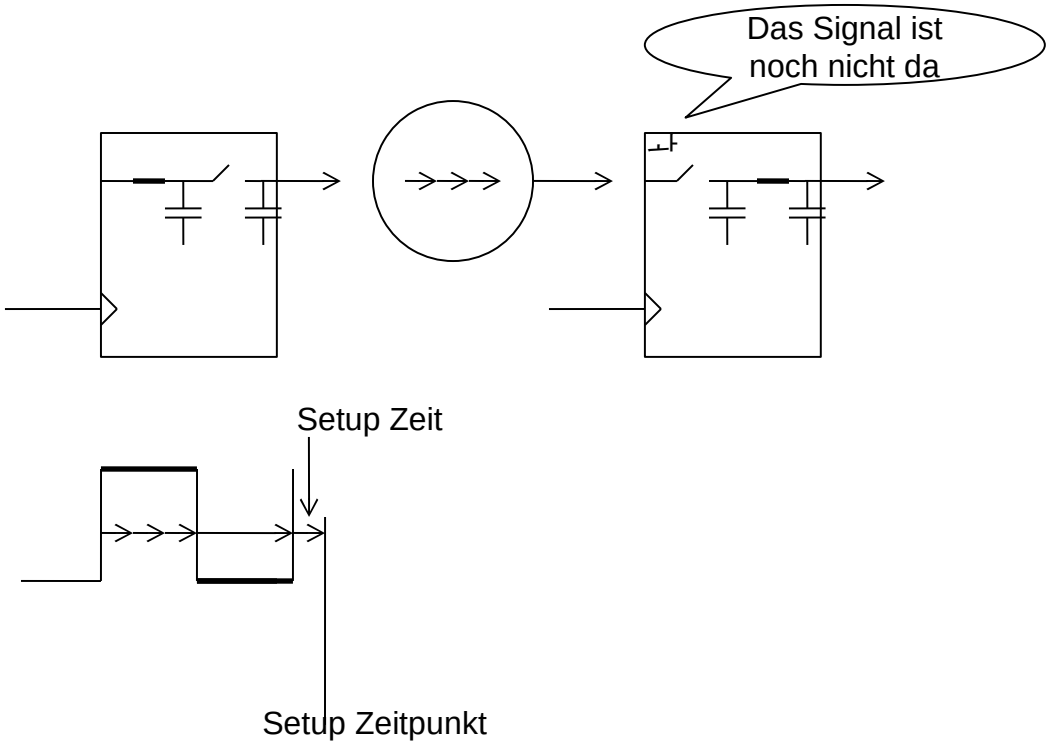
- Setup Zeit Verletzung



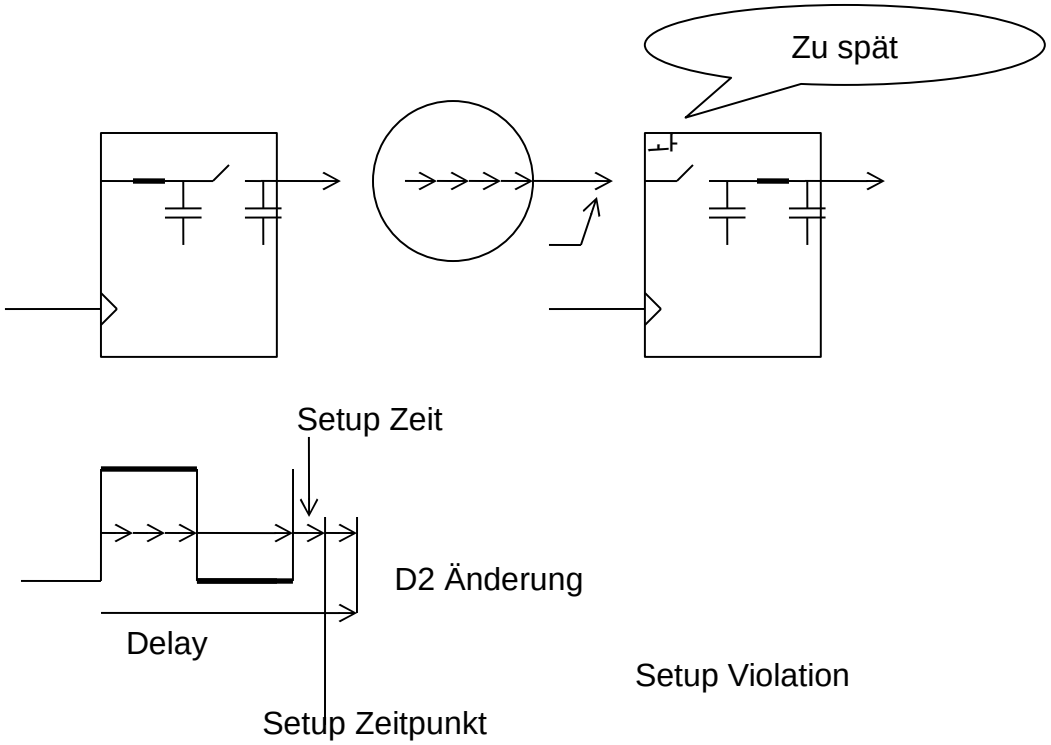
- Setup Zeit Verletzung



- Setup Zeit Verletzung



- Setup Zeit Verletzung
- Setupzeit Verletzung passiert wenn sich Niveau am Eingang D2 zu langsam ändert. Das passiert am meistens wenn die Taktfrequenz zu hoch ist oder die kombinatorische Logik zu langsam.



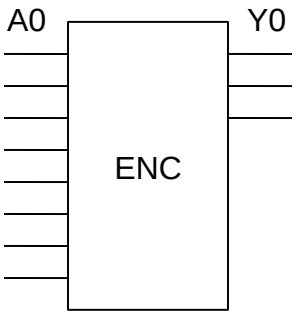
Slack = Ck2 - Tsetup - (Ck1-Tck) - Delay

- Setup-Zeit Verletzungen kann man durch langsamere Taktfrequenz verhindern.
- Hold-Zeit Verletzungen kann man, wenn sie vorhanden sind, nicht mehr entfernen.
- Wenn eine Schaltung Hold-Zeit Probleme hat, kann man sie in der Regel nicht verwenden.
- *Hold-Zeit Probleme verhindert man im Design durch eine scheinbare Taktverlangsamung am Empfänger Flipflop. Diese nennt man Clock Uncertainty. Es ist ein overconstraint*
- *Auf diese Weise wird Synthese Tool gezwungen D2 in Bezug auf Ck-Eingang am FF2 zu verlangsamen. Das erreicht das Tool z.B. durch Einfügen von Invertern im Datenpfad.*

Kodierer


Snapshots at jasonlove.com
WARNING
Are you sure you want to reformat the hard drive?
Yes
No
© Love 2004

- Kodierer (Encoder)
- Um eine Information bearbeiten zu können, muss sie in einem geeignetem Zahlsystem z.B. im binären System dargestellt werden
- Bsp. Tastatur
- Jeder Taste gehört ein Digitaleingang
- Kodierer erzeugt den binären Code, der der aktivierten Taste entspricht
- Der einfachste Kodierer setzt voraus dass nur ein Eingang in einem Moment aktiv ist



A0	A1	A2	A3	A4	A5	A6	A7	Y0	Y1	Y2
1	0	0	0	0	0	0	0	0	0	0
	1							0	0	1
		1						0	1	0
			1					0	1	1
				1				1	0	0
					1			1	0	1
						1		1	1	0
							1	1	1	1

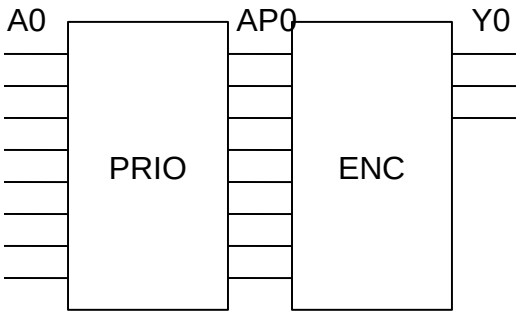
- Kodierer
- Y0 = A1 || A3 || A5 || A7
- Y1 = A2 || A3 || A6 || A7
- Y2 = A4 || A5 || A6 || A7
- Umgekehrte Funktionalität wie der Dekoder

A0	A1	A2	A3	A4	A5	A6	A7	Y0	Y1	Y2
1	0	0	0	0	0	0	0	0	0	0
	1							0	0	1
		1						0	1	0
			1					0	1	1
				1				1	0	0
					1			1	0	1
						1		1	1	0
							1	1	1	1

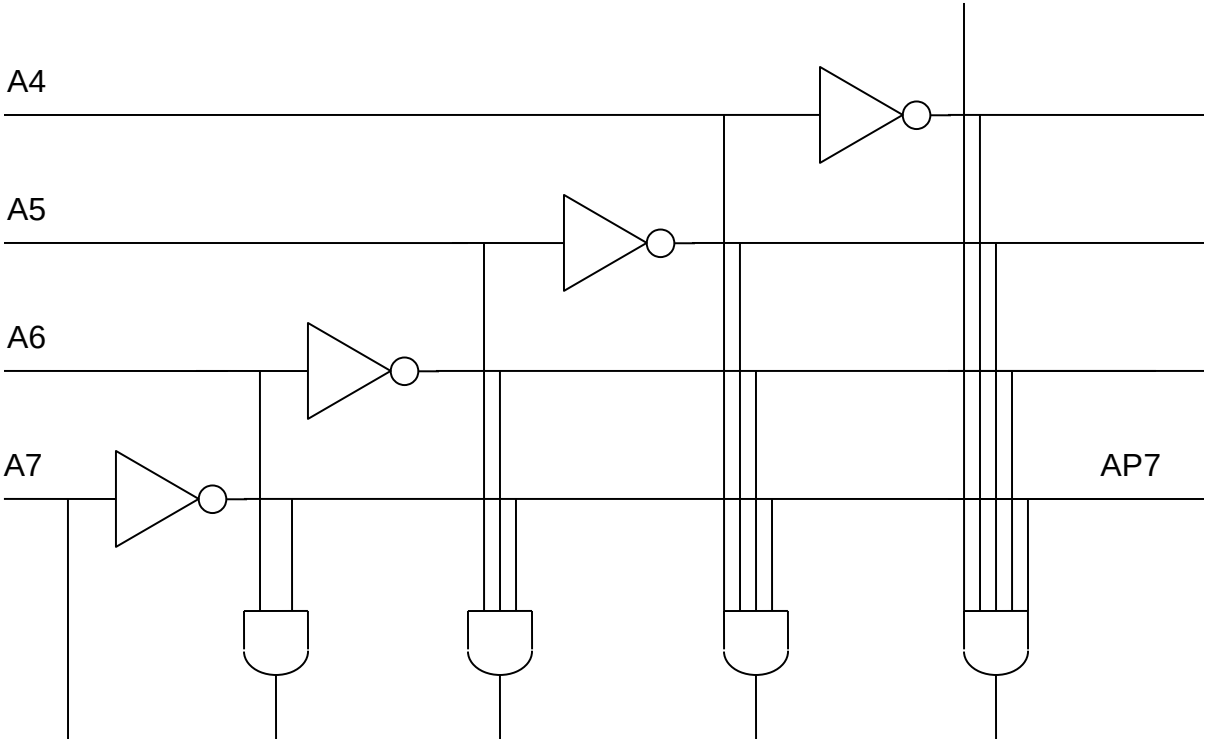
- Beachten wir dass der Kodierer nur dann richtig funktioniert, wenn nur ein Eingangssignal aktiv ist. Wenn z.B. A3 und A4 gleichzeitig aktiv werden, bekommen wir am Ausgang den Code Y0 = Y1 = Y2 = 1 -> 7 statt 3 oder 4.
- In den Systemen wo mehrere Eingänge gleichzeitig aktiv werden können werden die Prioritätskodierer benutzt. Diese erzeugen den Kode des Eingangs mit höchster Priorität – z.B. den größeren Kode.

A0	A1	A2	A3	A4	A5	A6	A7	Y0	Y1	Y2
1	0	0	0	0	0	0	0	0	0	0
	1							0	0	1
		1						0	1	0
			1					0	1	1
				1				1	0	0
					1			1	0	1
						1		1	1	0
							1	1	1	1

- Man kann den Prioritätskodierer mithilfe eines einfachen Kodierers und eines Prioritäts-Netzwerks aufbauen. Das Prioritätsnetzwerk soll gewährleisten, dass nur ein Ausgang aktiv ist, ungeachtet von der Zahl der aktiven Eingängen. Z.B., wenn A3 und A4 Eingänge aktiv sind, soll nur AP4 aktiv werden.
- Das Prioritätsnetzwerk kann mit folgenden Funktionen beschrieben werden:
- AP7 = A7
- AP6 = A6 & !A7
- AP5 = A5 & !A6 & !A7
- …
- AP0 = A0 & !A1 & !A2 & !A3 & !A4 & !A5 & !A6 & !A7



- AP7 = A7
- AP6 = A6 & !A7
- AP5 = A5 & !A6 & !A7
- …
- AP0 = A0 & !A1 & !A2 & !A3 & !A4 & !A5 & !A6 & !A7



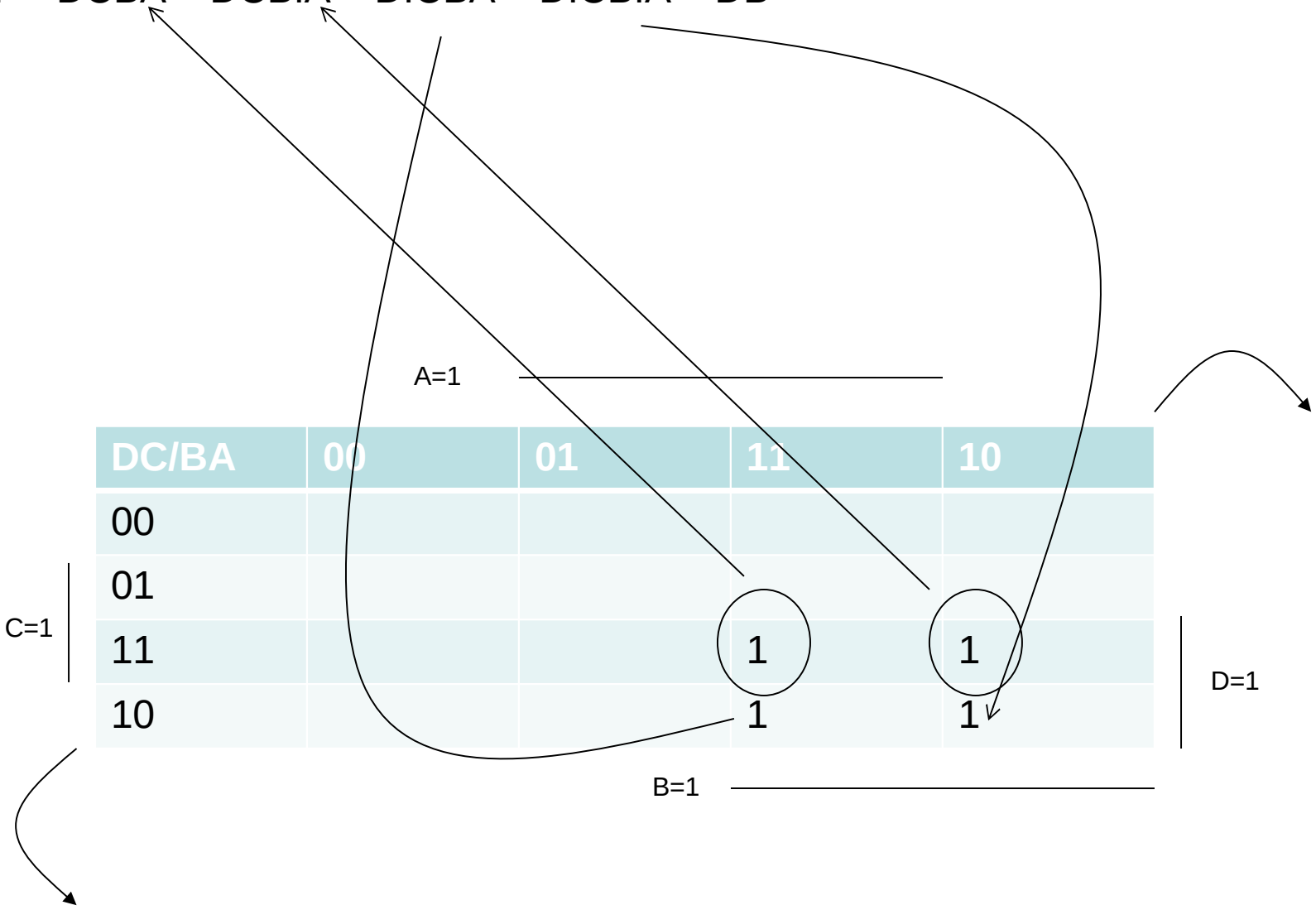
- Ein Prioritätskodierer hat oft die Signale Prio_Input und Prio_Output, die benutzt werden können um größeren Kodierer als Kaskade von mehreren kleineren zu realisieren.
- Prio Output ist die ODER Funktion vom Prio_Input und allen Eingängen.
- Die Ausgänge sind nur dann Aktiv wenn Prio Input = 0 ist.

Minimierung von Schaltfunktionen
Karnaugh Tabellen

- Karnaugh Tabellen
- Kombinatorische Tabelle kann man als z.B. disjunktive Normalform darstellen
- Jeder Zeile mit 1 entspricht eine UND Funktion -> die Gesamttabelle ist ODER Funktion von einzelnen Zeilen
- Normalform kann oft vereinfacht werden
- Bsp. AB || !AB = B
- & = *

- Karnaugh-Tabelle ist eine Graphische Darstellung der Wahrheitstabelle. Es ist aus einer Kanaugh-Tabelle leicht zu erkennen ob eine Normalform vereinfacht werden kann und wie.
- Eine Karnaugh-Tabelle für n Eingangsvariablen hat 2^n Felder. (4 -> 16)
- Am Rand der Tabelle werden die Variablen beschriftet – jede Zeile gehört einer Variable (negiert oder nicht-negiert) oder einem Produkt von zwei/drei (negierten oder nicht-negierten) Variablen – die negierte Variable wird mit Null oder !Xi beschriftet.
- Wichtig ist, dass sich horizontal und vertikal benachbarte Felder nur in genau einer Variablen unterscheiden dürfen. *Gray Code* wird verwendet.
- Mithilfe von Wahrheitstabelle wird in einzelnen Feldern Eins eigetragen wenn für die gegebene Variablen-Kombination die entsprechende Zeile eins ist.
- Karnaugh-Diagramme eignen sich für die Vereinfachung von Funktionen mit maximal ca. 4–6 Eingangsvariablen; bis 4 Variablen sind sie übersichtlich. Ab dann Spiegelung

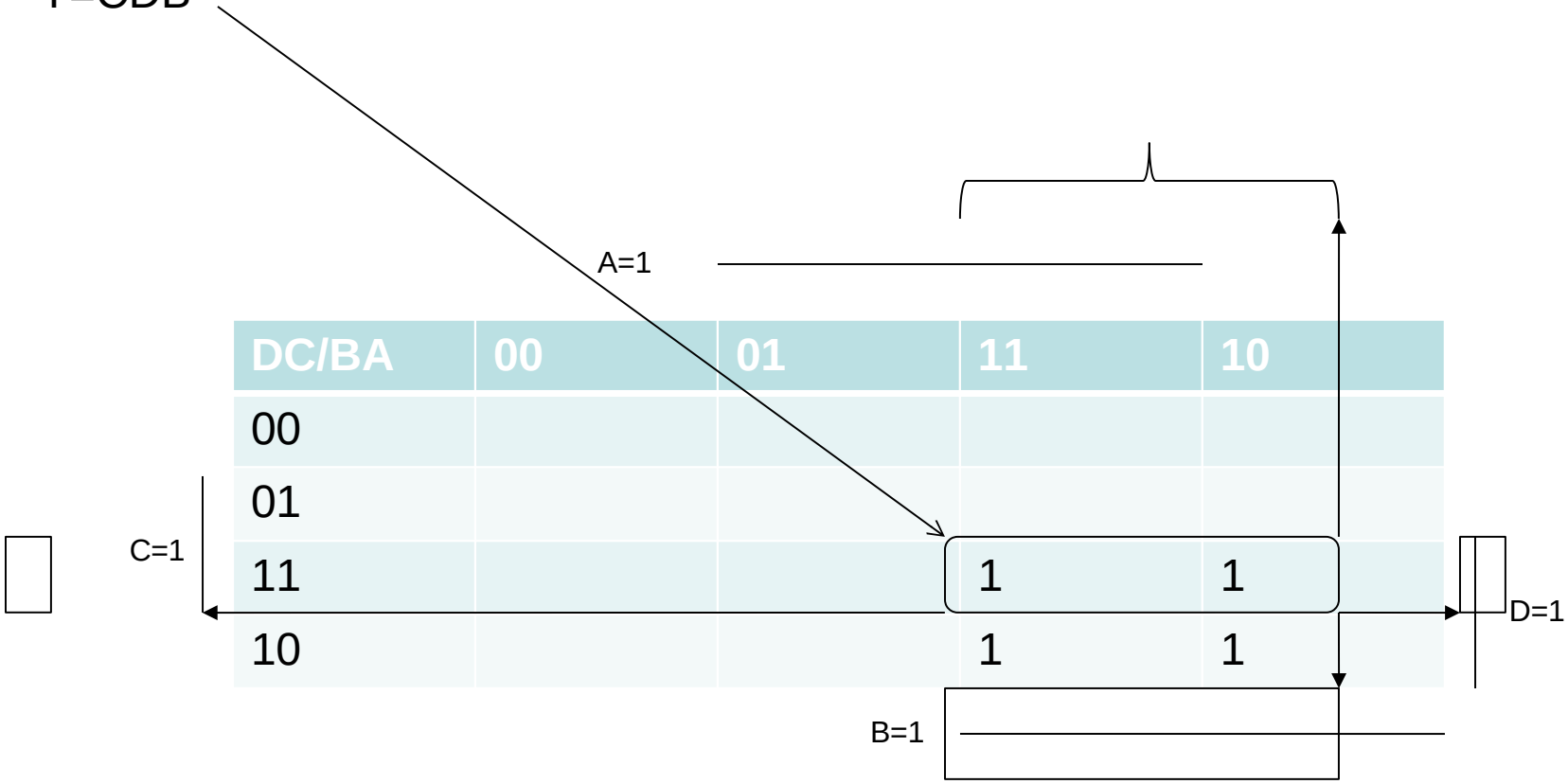
- Y = DCBA + DCB!A + D!CBA + D!CB!A = DB



DC/BA	00	01	11	10
00				
01				
11			1	1
10			1	1

Wenn wir ein Block mit Einsen haben und wenn in diesem Block einige Variablen alle Kombinationen durchlaufen, können wir diese aus dem UND Produkt eliminieren. Der Block wird nur durch die feste Variablen dargestellt.

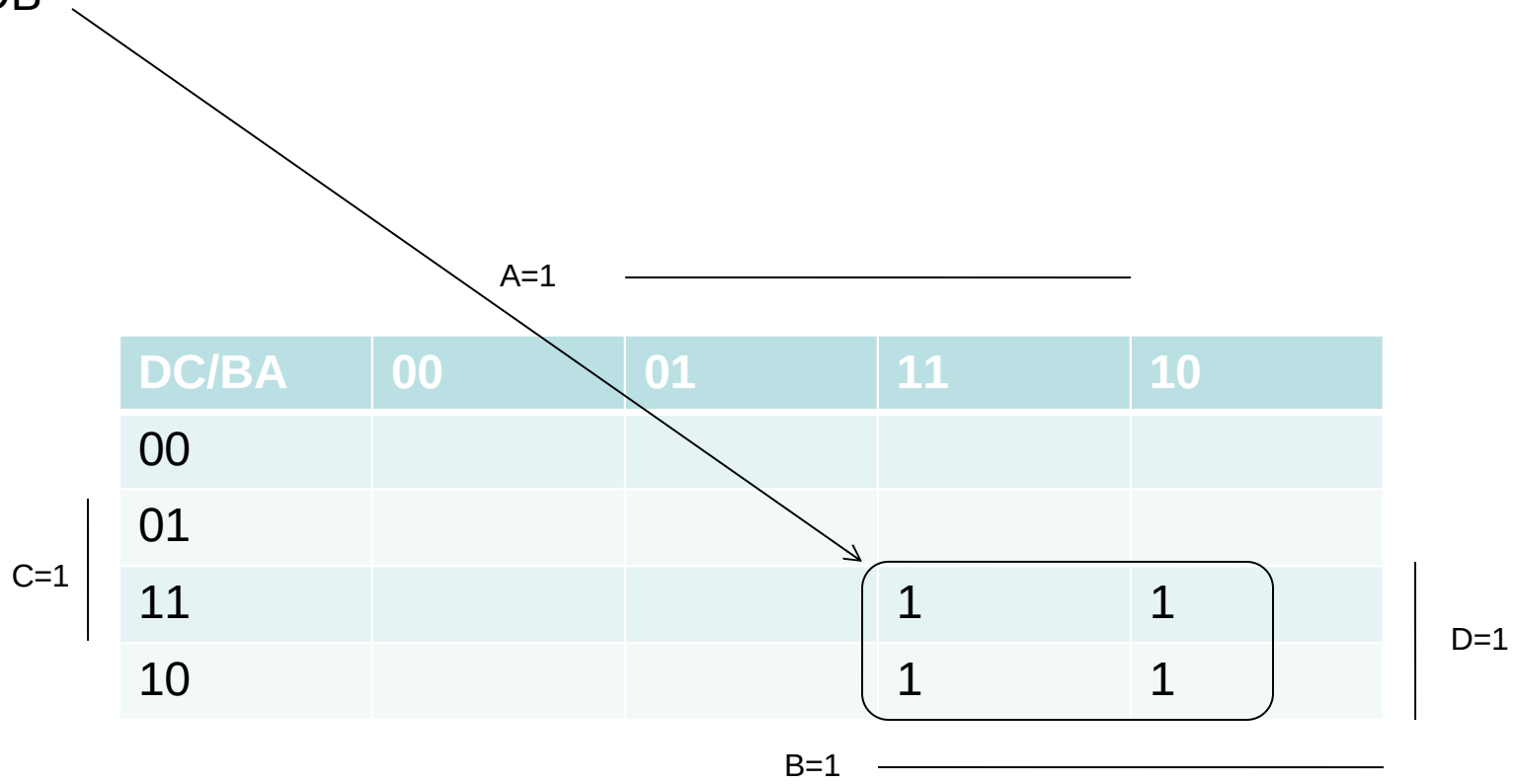
Y=CDB



DC/BA	00	01	11	10
00				
01				
11			1	1
10			1	1

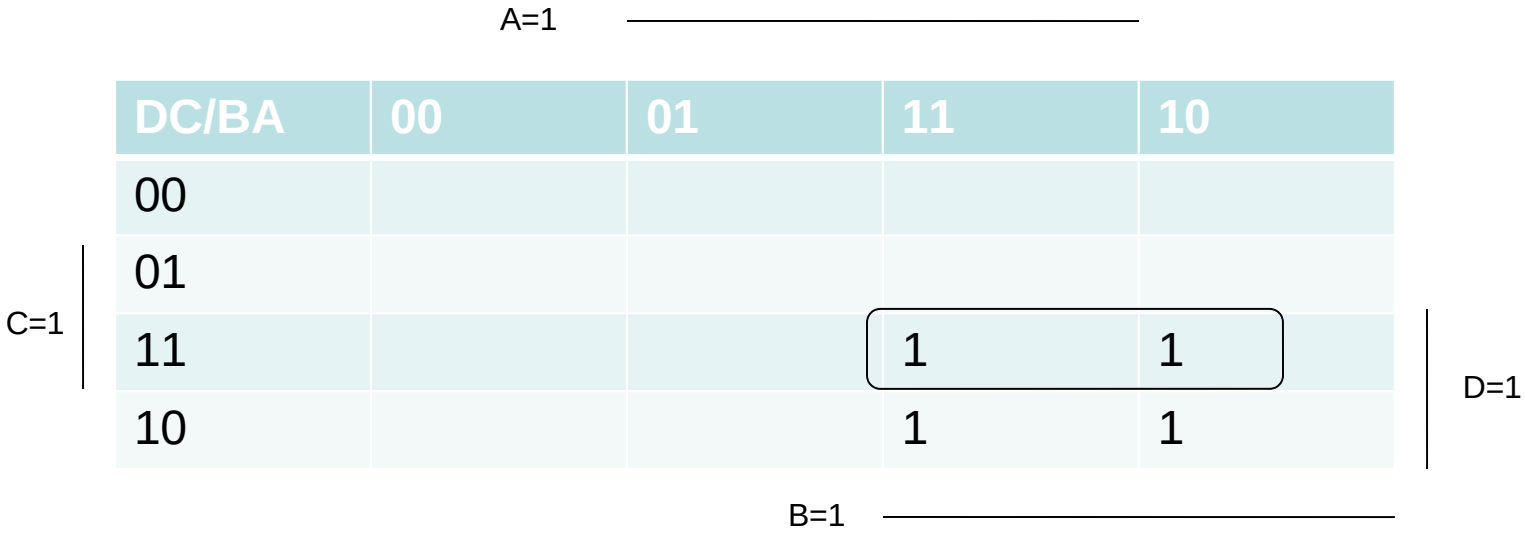
Wenn wir ein Block mit Einsen haben und wenn in diesem Block einige Variablen alle Kombinationen durchlaufen, können wir diese aus dem UND Produkt eliminieren. Der Block wird nur durch die feste Variablen dargestellt.

Y=DB



DC/BA	00	01	11	10
00				
01				
11			1	1
10			1	1

- Wenn wir bis zwei Variablen an einem Rand haben, und Gray Code verwenden, kann für jeden 2x1 Block eine Variable eliminiert werden, für jeden 2x2 Block zwei Variablen, für jeden 2x4 Block drei Variablen.



- Die Minimierung wird wie folgend gemacht:
- Man versucht, möglichst viele horizontal und vertikal benachbarte Felder, die eine 1 enthalten, zu rechteckigen zusammenhängenden Blöcken zusammenzufassen. Als Blockgröße sind alle Potenzen von 2 erlaubt
- Dabei sind alle 1-Felder mit Blöcken zu erfassen
- Ein Block kann über den rechten bzw. unteren Rand des Diagramms fortgesetzt werden

A=1

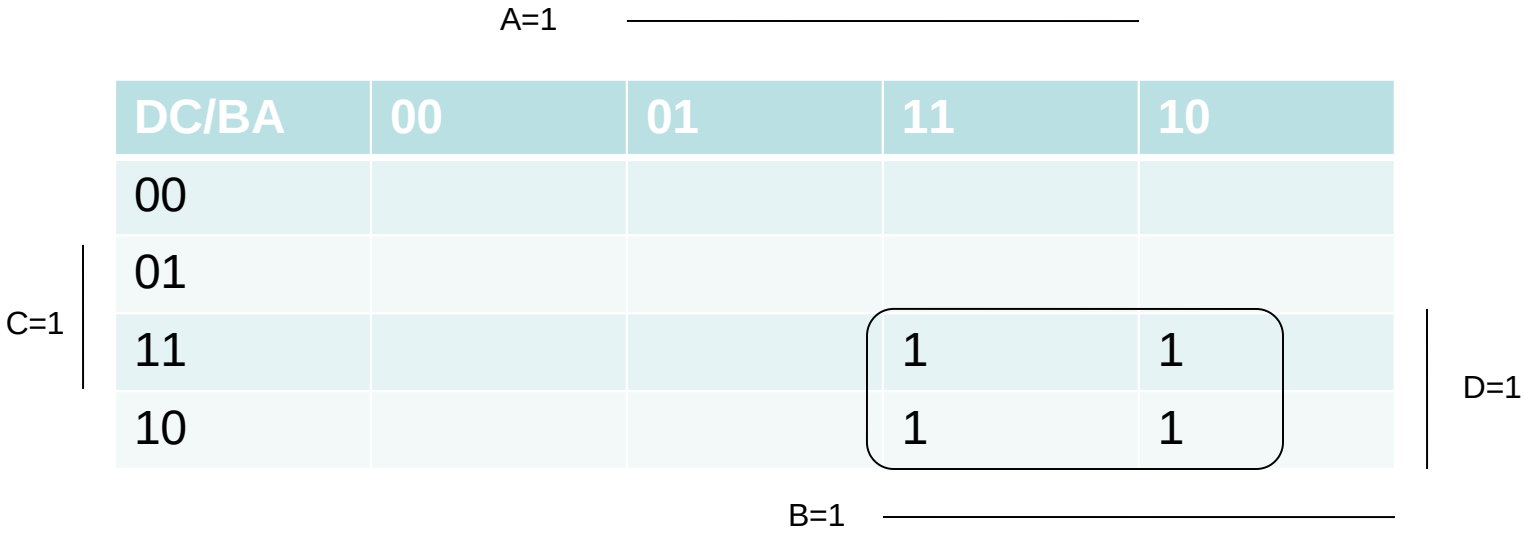
DC/BA	00	01	11	10
00				
01				
11			1	1
10			1	1

C=1

D=1

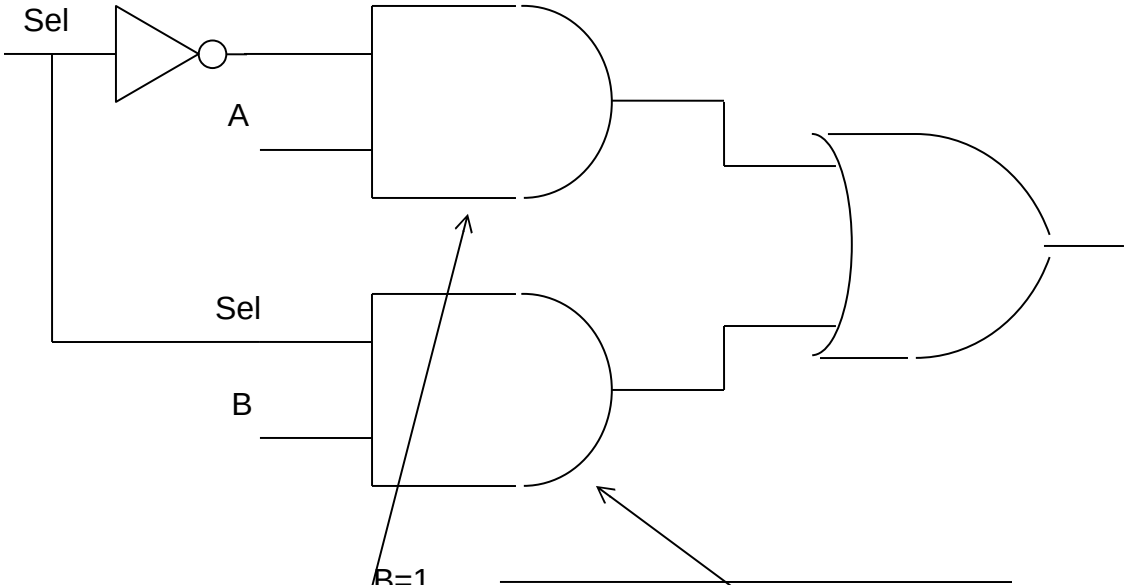
B=1

- Die gebildeten und ausgewählten Blöcke/Päckchen wandelt man nun in Konjunktionsterme um. Dabei werden Variablen innerhalb eines Blockes, die in allen Formenkombinationen auftreten, weggelassen.
- Diese UND-Verknüpfungen werden „ver-ODERt“ und ergeben eine disjunktive Minimalform.



Glitch

- Die Verwendung minimaler Logik führt oft zu einem Problem genannt Glitch.



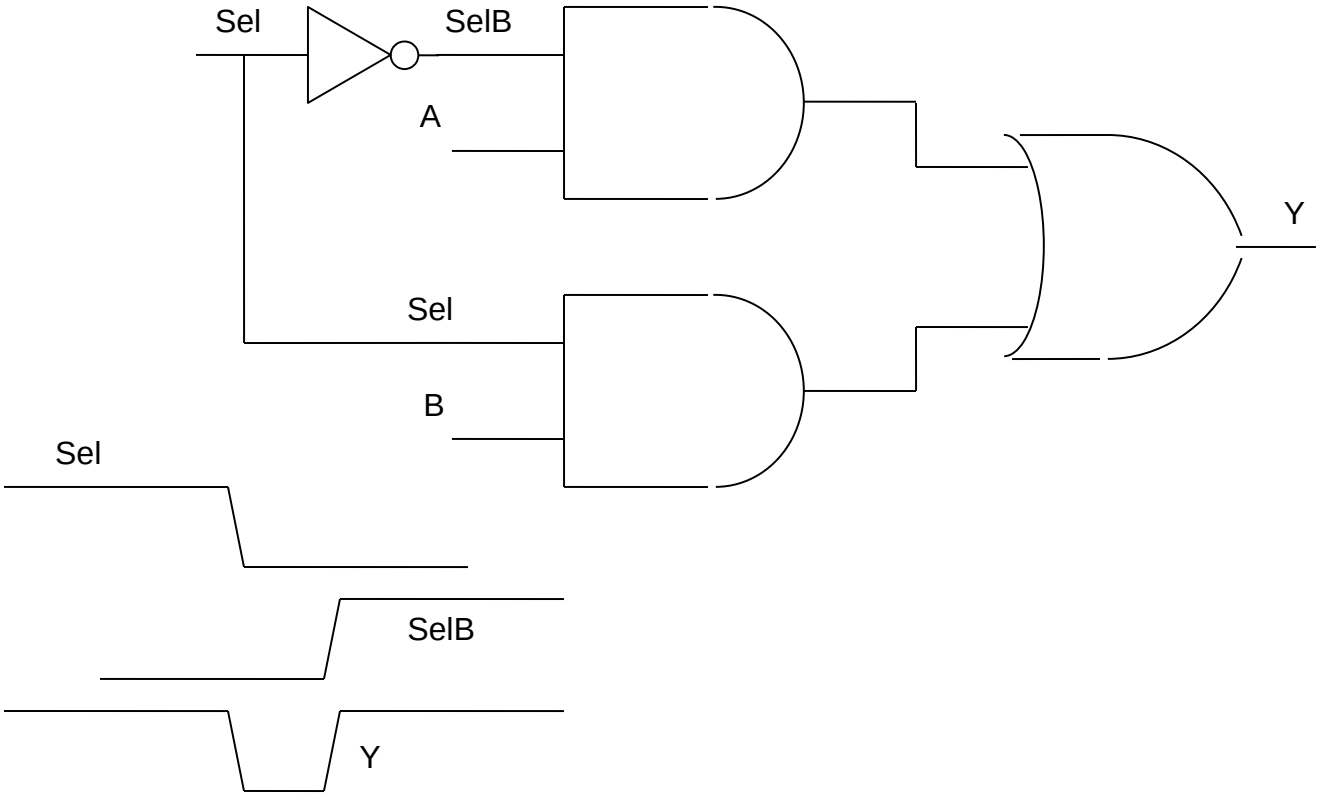
Sel/B	00	01	11	10
0			1	
1	1	1	1	

B=1

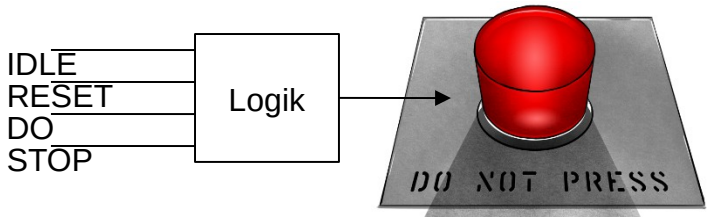
A=1

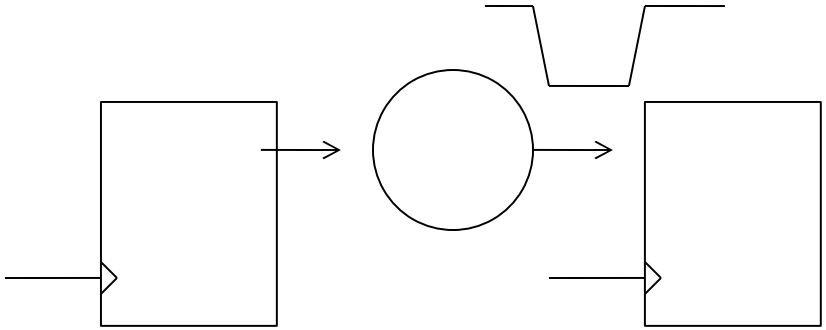
Sel=1

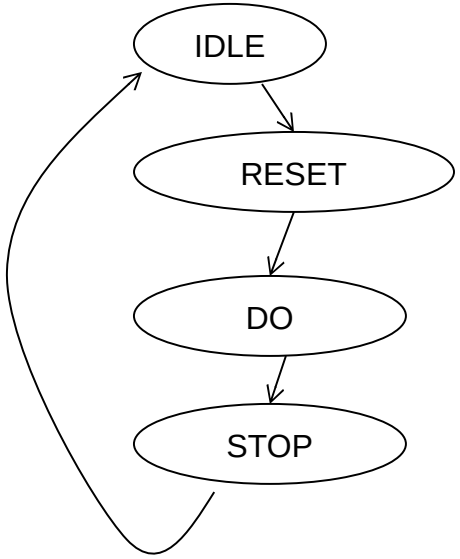
- Y = !Sel A + Sel B
- Nehmen wir an, dass beide Eingänge „1“ sind: A = B = 1
- Sel ist anfangs 1 und ändert sich auf 0 -> wir erwarten Y = 1
- Ein kurze Zeit sehen beide AND Gatter den Select Eingang 0, wir bekommen für eine kurze Zeit 0 am Ausgang



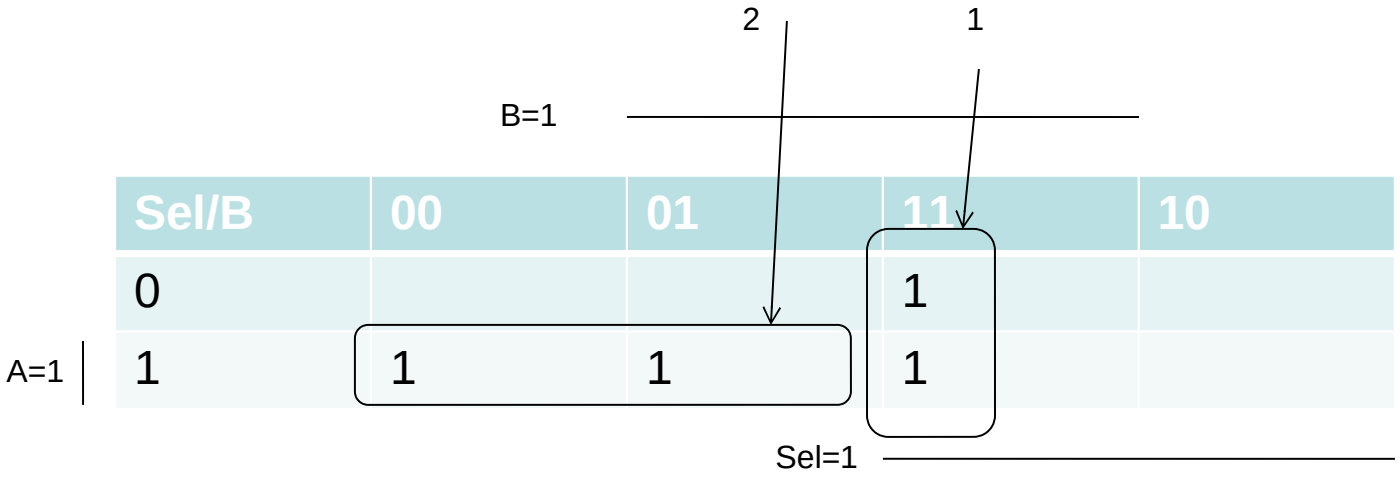
- Ist Glich ein Problem?
- Synchrone Schaltungen: Unproblematisch falls es kürzere Zeit als eine Taktperiode dauert
- Problem: Ausgänge der Statemaschine ohne Register





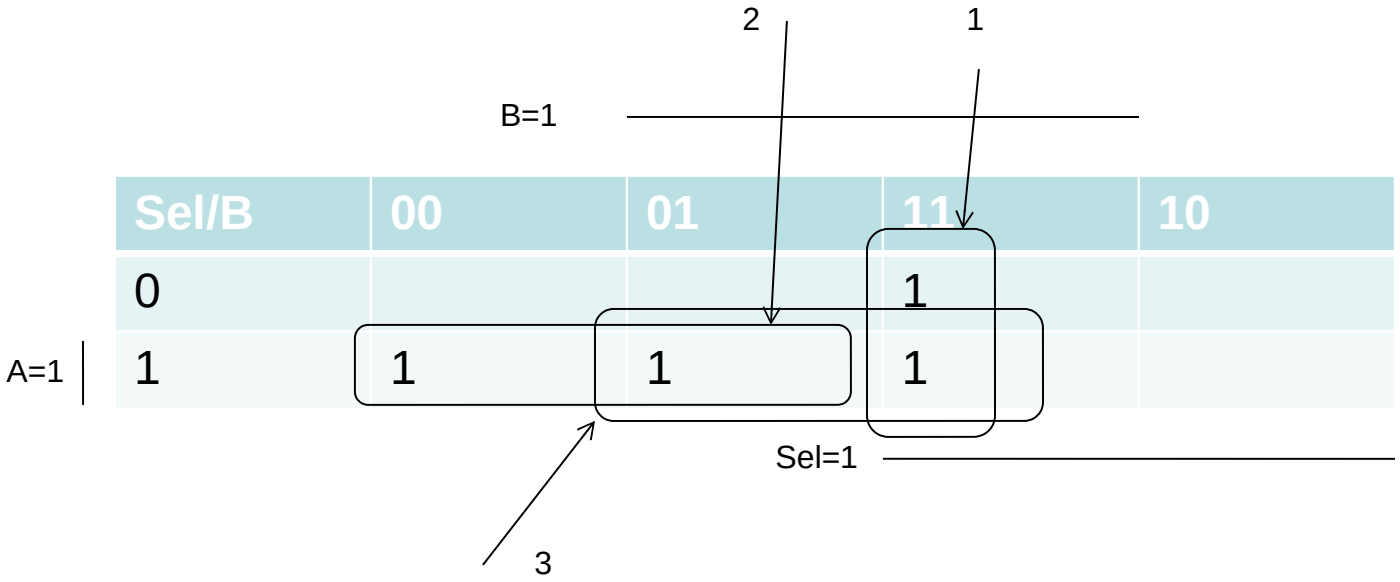


- Man kann die Möglichkeit eines Glitch-es aus Karnaugh Tabelle erkennen
- Zwei Gruppen sind getrennt und liegen naheinander.
- Wenn sich die Variable Sel von 1 auf 0 oder 1 auf 0 ändert, für A = B = 1, wird die Gruppe 1 „ausgeschaltet“ (bzw. ihre UND Funktion wird 0) und 2 eingeschaltet (bzw. ihre UND Funktion wird 1)
- Wenn das nicht synchron passiert, können wir 0 als Glitch bekommen.

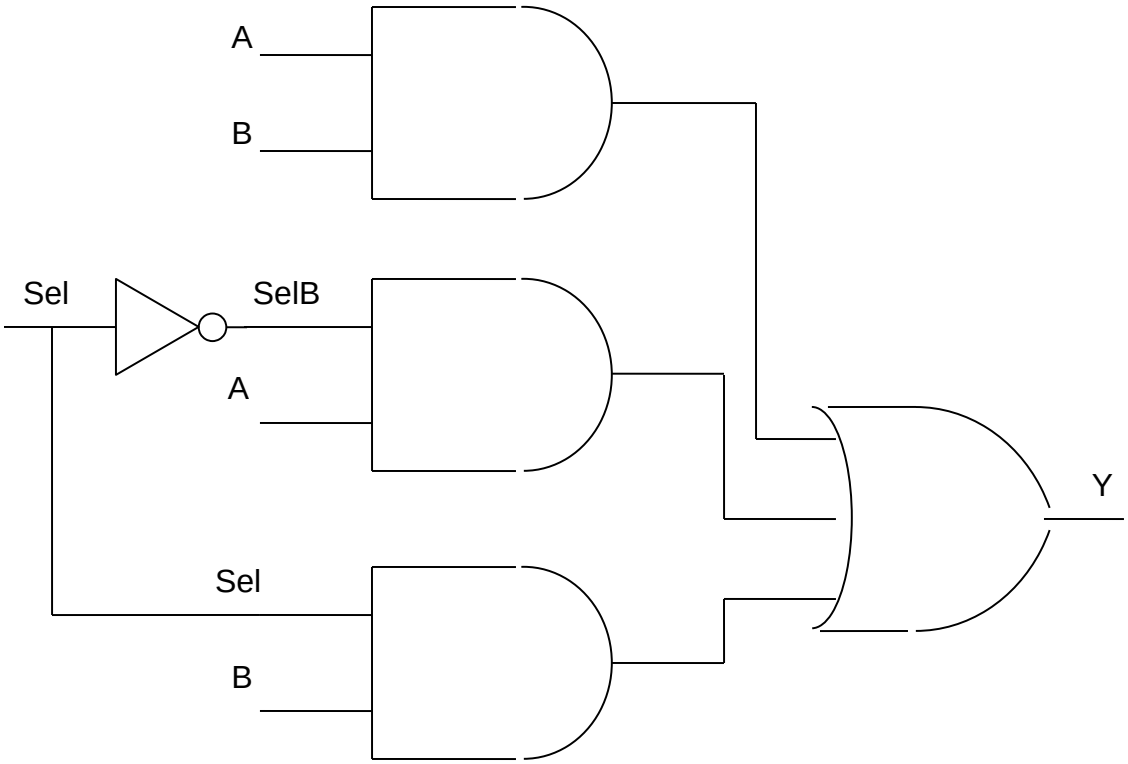


Sel/B	00	01	11	10
0			1	
1	1	1	1	

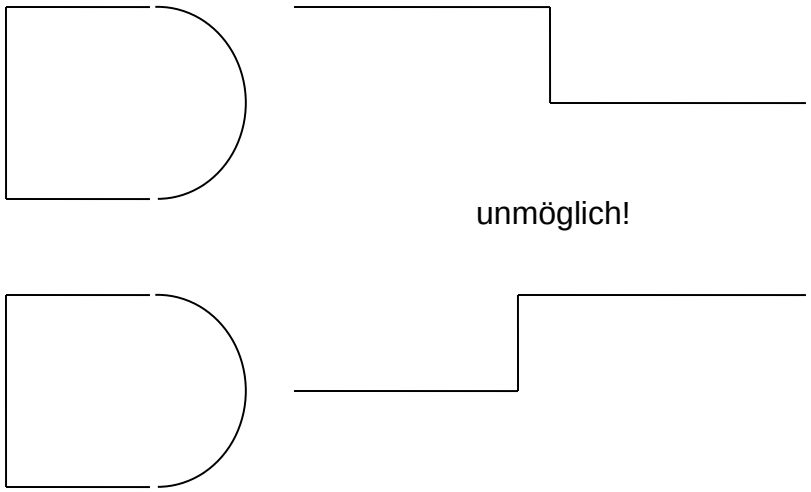
- Man kann ein Glitch verhindern indem man eine zusätzliche Gruppe 3 hinzufügt die als „Brücke“ zwischen den Gruppen 1 und 2 dient.
- A & B
- Beim Sel Änderung (für A = B = 1) wird die Gruppe 3 nicht ausgeschaltet – Select ist nicht als Variable vorhanden. Das verhindert ein 0-Glitch.



- Glitch-freie Schaltungen sind normalerweise komplizierter als die minimalen Schaltungen



- Beachten wir, dass die kombinatorischen Schaltungen, implementiert als disjunktive Normalform, kein 1-Glitch erzeugen können (unter Annahme dass sich nur eine Eingangsvariable ändert)
- Einzige Möglichkeit für Glitch 1 wäre wenn ein UND zu früh und das andere zu spät 1 wird. Das kann nicht passieren wenn sich nur eine Variable ändert



- Kombinatorischen Schaltungen können auch als konjunktive Normalform implementiert werden.
- UND Verknüpfung von vielen ODER Funktionen.
- Mit ODER „Summen“ stellt man die Zeilen in der Wahrheitstabelle dar, die null sind (Variablen die 1 sind werden negiert)
- Eine Konjunktive Normalform kann keine 0-Glitches haben wenn sich nur eine Variable ändert.
- Konjunktive Normalform ist für die Funktionen geeignet die „viele Einsen“ als Ergebnis haben.
- Bsp. Y = !A || B || !C || D

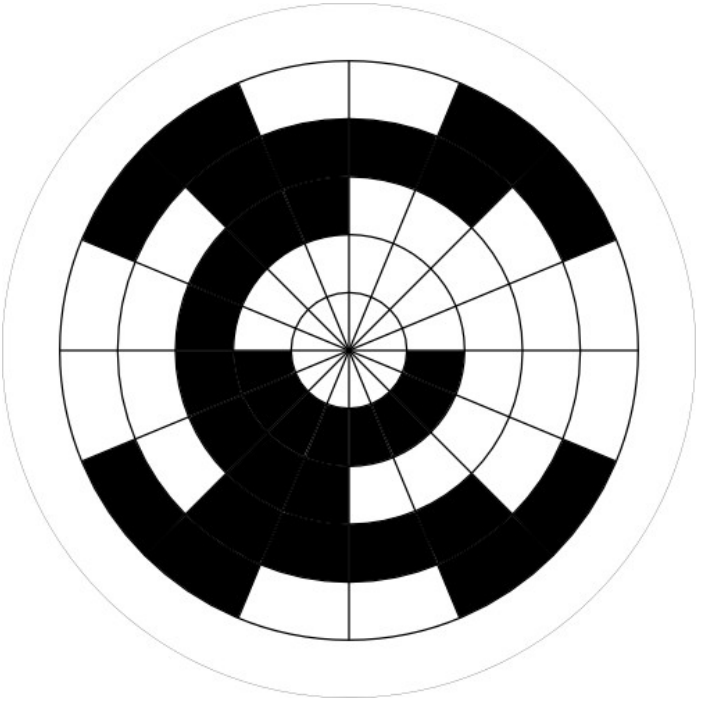
DC/BA	00	01	11	10
00	1	1	1	1
01	1	0	1	1
11	1	1	1	1
10	1	1	1	1

C=1

D=1

B=1

Gray Code



- Grey Code
- Grey Code hat die Eigenschaft, dass sich immer nur ein Bit ändert wenn man hochzählt
- Weniger Glitch-es
- Zeitmessung von asynchronen Signalen

	B2	B1	B0	G2	G1	G0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

- Es gilt:
- G0 = B1 exor B0
- G1 = B2 exor B1
- …
- Gn-1 = Bn-1

	B2	B1	B0	G2	G1	G0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

- Es gilt auch:
- Bn-1 = Gn-1
- Bn-2 = Bn-1 exor Gn-2
- …
- B0 = B1 exor G0

	B2	B1	B0	G2	G1	G0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

Statemaschine (Zustandsautomat)

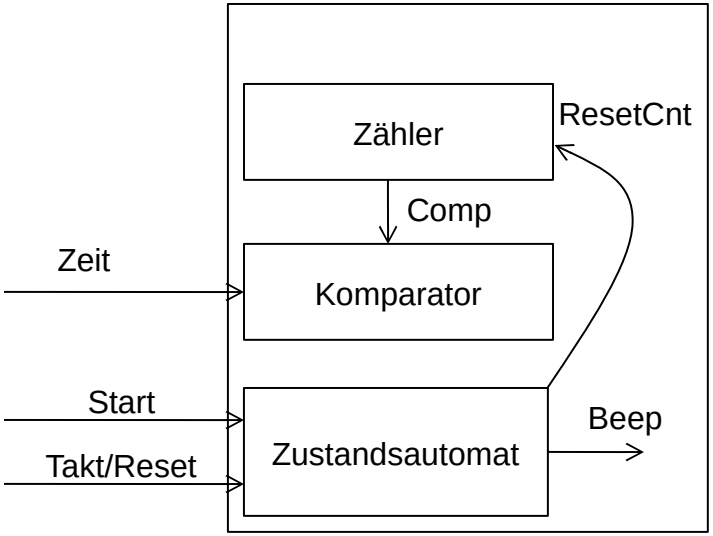
- Zustandsmaschinen werden für die Ansteuerung von digitalen Systemen verwendet. Man kann sie mit Programmen vergleichen - der Programmcode ist fest
- Ausgangssignale von Zustandsmaschinen (*und allen sequentiellen Schaltungen*) hängen nicht nur von momentanen Werten der Eingangsvariablen sondern auch von deren Reihenfolge. Dieses Verhalten ist möglich wenn die Schaltung Speicherelemente hat.
- N Speicherelemente -> maximal 2^n Zustände
- Da es eine endliche Zahl von möglichen Zuständen gibt, nennt man solche Zustandsautomate finite-state Maschinen
- Den Zustand des Speicherelements nennt man Zustandsvariable.

- Die Zustandsmaschinen kann man in zwei Klassen unterteilen.
- Moore Typ: Der Ausgang hängt nur von der Zustandsvariable – d.h. nur vom Zustand der Maschine
- Mealy Typ: Der Ausgang hängt auch von Eingängen
- Mealy Maschinen können oft mit weniger Zuständen realisiert werden, brauchen aber kombinatorische Schaltung für die Erzeugung von Ausgangssignalen. Moore Typ Automaten sind einfacher zu beschreiben, brauchen oft mehr Zuständen.

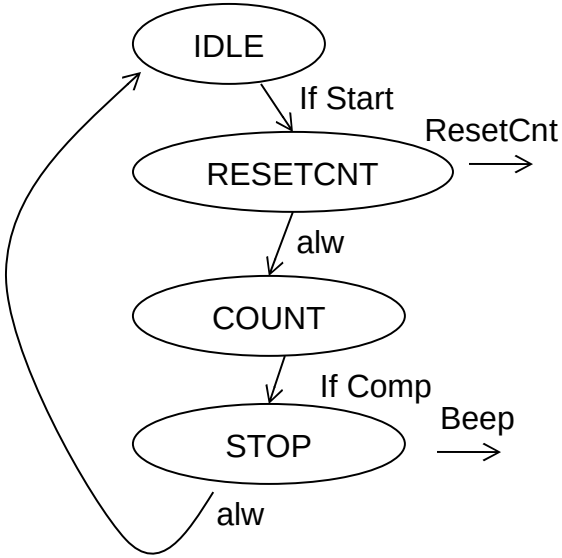
- Man kann in einem Zustandsautomat jede Art von Speicherzellen verwenden um den Zustand zu speichern
- Wenn alle Speicherzellen in einer Statemaschine den Zustand gleichzeitig ändern, z.B. auf steigende Taktflanke, nennen wir dieses Netzwerk synchron. Synchrone Zustandsmaschine verwenden Filp-Flops als Speicherelemente
- Man kann auch Zustandsmaschinen bauen, deren Zustand sich durch Änderung von verschiedenen Eingangssignalen ändert. Solche Netzwerke nennen wir asynchron. Asynchrone Zustandsmaschinen verwenden Latches als Speicherelemente.

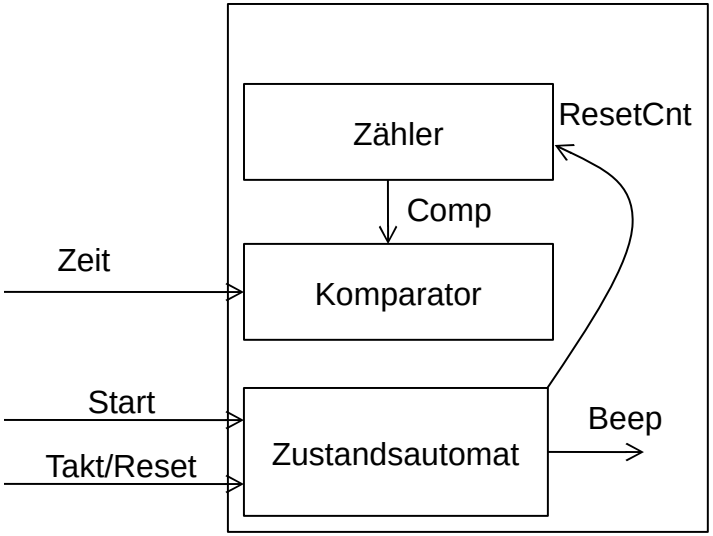
- Die Funktionalität einer Zustandsmaschine kann man z.B. mit einem Zustandsdiagramm beschrieben. Solch ein Zustandsdiagram hat für eine Statemaschine die gleiche Bedeutung wie eine Wahrheitstabelle für eine kombinatorische Schaltung.
- Ein Zustandsdiagramm kann entweder graphisch oder als Verilog/VHDL Code dargestellt werden.

- Beispiel Timer
- Der Timer besteht aus folgenden Komponenten – einem Zähler, einem Komparator, einem Startkopf, einem Drehregler für die Zeiteinstellung und einem Lautsprecher. Der Komparator vergleicht den Zähler-Zustand mit der eingestellten Zeit.
- Die Eingänge für die State-Maschine sind das Startsignal und der Komparator-Ausgang. Die Ausgangssignale sind ein Reset Signal für den Zähler und ein Signal für den Lautsprecher.
- Statmeschine braucht noch ein Taktsignal und asynchrons Reset

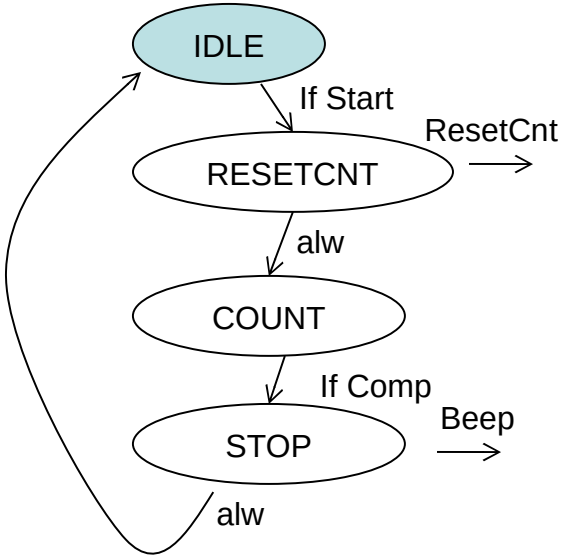


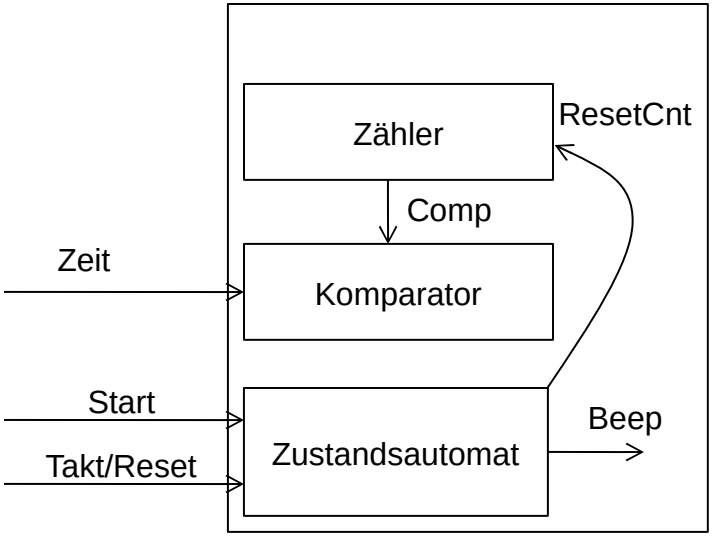
```
Input clk, reset, start, comp;
Output resetcounter, beep;

Reg [1:0] State;

Parameter IDLE = 2‘b00, RESETCNT = 2‘b01, COUNT = 2’b11, STOP = 2’b10;

Assign resetcounter = (State == RESETCNT);
Assign beep = (State == STOP);

Always @ (posedge clk or posedge reset) Begin
        If (reset) State <= IDLE;
        Else begin
                Case (State)
                        IDLE: begin
                        If (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                        End
                        RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                        End
                        COUNT: begin
                        //Counter <= Counter + 1;
                        If (comp) State <= STOP;
                        End
                        STOP: begin
                        State <= IDLE;
                        End
                Endcase
        End//not reset
        End//always
```



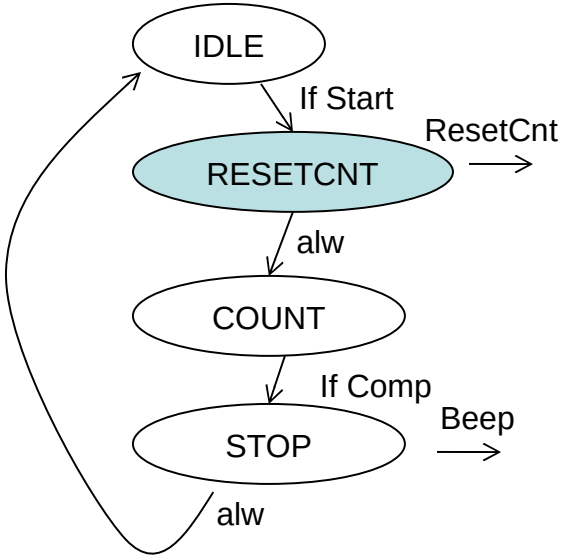


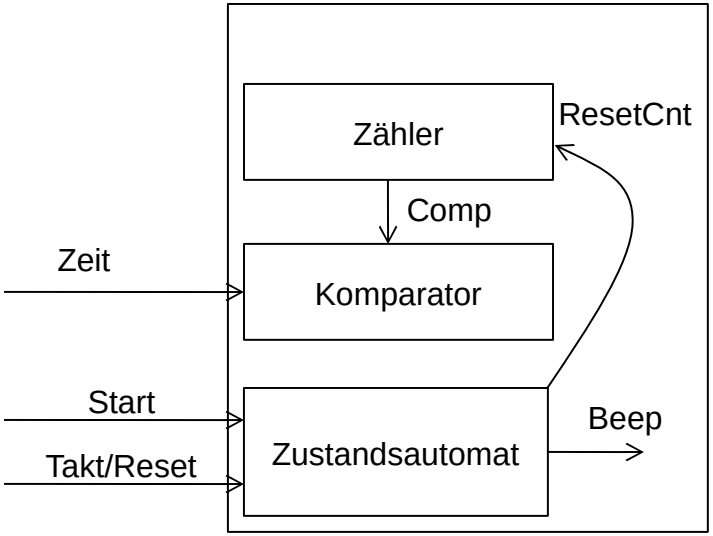
```
Input clk, reset, start, comp;
Output resetcounter, beep;

Reg [1:0] State;

Parameter IDLE = 2‘b00, RESETCNT = 2‘b01, COUNT = 2’b11, STOP = 2’b10;

Assign resetcounter = (State == RESETCNT);
Assign beep = (State == STOP);

Always @ (posedge clk or posedge reset) Begin
        If (reset) State <= IDLE;
        Else begin
                Case (State)
                        IDLE: begin
                        If (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                        End
                        RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                        End
                        COUNT: begin
                        //Counter <= Counter + 1;
                        If (comp) State <= STOP;
                        End
                        STOP: begin
                        State <= IDLE;
                        End
                Endcase
        End//not reset
        End//always
```



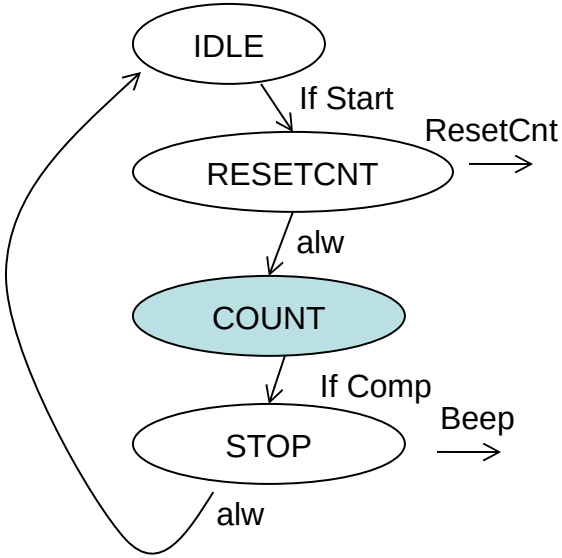


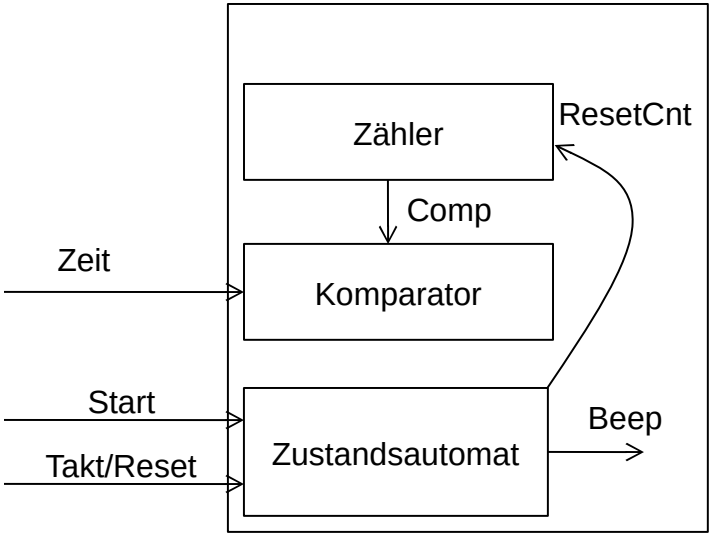
```
Input clk, reset, start, comp;
Output resetcounter, beep;

Reg [1:0] State;

Parameter IDLE = 2‘b00, RESETCNT = 2‘b01, COUNT = 2’b11, STOP = 2’b10;

Assign resetcounter = (State == RESETCNT);
Assign beep = (State == STOP);

Always @ (posedge clk or posedge reset) Begin
        If (reset) State <= IDLE;
        Else begin
                Case (State)
                        IDLE: begin
                        If (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                        End
                        RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                        End
                        COUNT: begin
                        //Counter <= Counter + 1;
                        If (comp) State <= STOP;
                        End
                        STOP: begin
                        State <= IDLE;
                        End
                Endcase
        End//not reset
        End//always
```



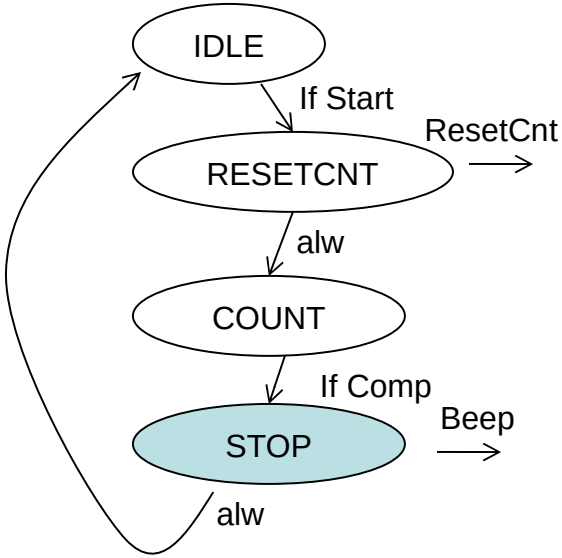


```
Input clk, reset, start, comp;
Output resetcounter, beep;

Reg [1:0] State;

Parameter IDLE = 2‘b00, RESETCNT = 2‘b01, COUNT = 2’b11, STOP = 2’b10;

Assign resetcounter = (State == RESETCNT);
Assign beep = (State == STOP);

Always @ (posedge clk or posedge reset) Begin
        If (reset) State <= IDLE;
        Else begin
                Case (State)
                        IDLE: begin
                        If (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                        End
                        RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                        End
                        COUNT: begin
                        //Counter <= Counter + 1;
                        If (comp) State <= STOP;
                        End
                        STOP: begin
                        State <= IDLE;
                        End
                Endcase
        End//not reset
        End//always
```

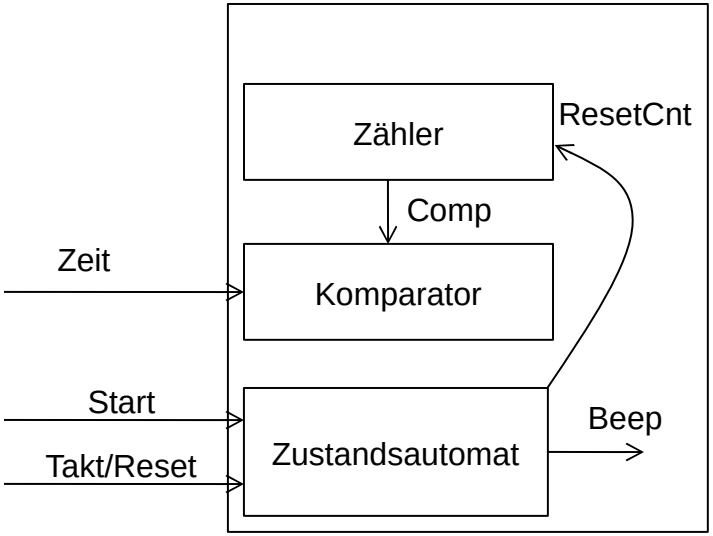




```
Input clk, reset, start, comp;
Output resetcounter, beep;

Reg [1:0] State;

Parameter IDLE = 2‘b00, RESETCNT = 2‘b01, COUNT = 2’b11, STOP = 2’b10;

Assign resetcounter = (State == RESETCNT);
Assign beep = (State == STOP);

Always @ (posedge clk or posedge reset) Begin
        If (reset) State <= IDLE;
        Else begin
                Case (State)
                        IDLE: begin
                        If (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                        End
                        RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                        End
                        COUNT: begin
                        //Counter <= Counter + 1;
                        If (comp) State <= STOP;
                        End
                        STOP: begin
                        State <= IDLE;
                        End
                Endcase
        End//not reset
        End//always
```



ResetCnt



- Oft enthält der Code der Statemaschine auch die Digitalschaltungen, die die Statemaschine ansteuert.
- Gray Code verwendet