- Wir: Dezimalzahlen und Alphabet
- Elektronik: fast immer binäre Codes
- Beispiel: Acht-Bit Zahl
- Sie besteht aus acht Stellen die entweder 0 oder 1 sein können. Jede Stelle nennen wir ein Bit
- 8-Bit: 2^8 = 256 Kombinationen, Zahlen 0-255 oder -128 bis 127
- 2^3 = 8
- 2^4 = 16
- …
- 2^10 =1024
- 2^(1N) ~ 2^N x 1000 = 2^N k

- Warum werden in digitaler Elektronik binäre Codes benutzt?
- Nehmen wir an wir möchten zwei Zahlen bis zur Größe N kodieren
- Wir brauchen lnN/lnX stellige Codes
- In Falle von Binären Codes X = 2
- Für eine Zahl bis 32 brauchen wir einen 5-stelligen Code für X = 2 und etwa dreistelligen für X = 3.
- Operation zwischen den zwei Zahlen
- Die Komplexität eines Bit-Operators steht im Zusammenhang mit der Größe der Ergebnistabelle
- In Fall von Binären Codes gibt es für zwei Eingangsvariablen vier Möglichkeiten, also vier Zeilen. Im allgeneinen Fall - X^2 Zeilen.
- Elektronik hätte die Größe:
- lnN/lnX * X^2
- Diese Funktion hat ein Minimum in der Nähe von 2

N = 32

A4	A3	A2	A1	A0
B4	B3	B2	B1	B0
C4	C3	C2	C1	C0

C=5x4=20

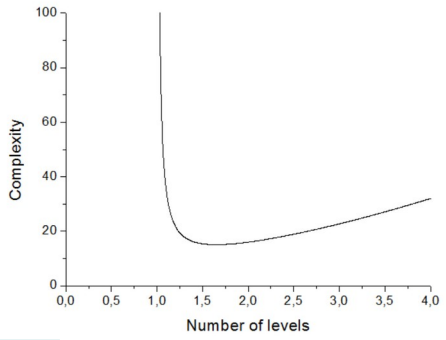
A	B	C
0	0	
0	1	
1	0	
1	1	

N = 27

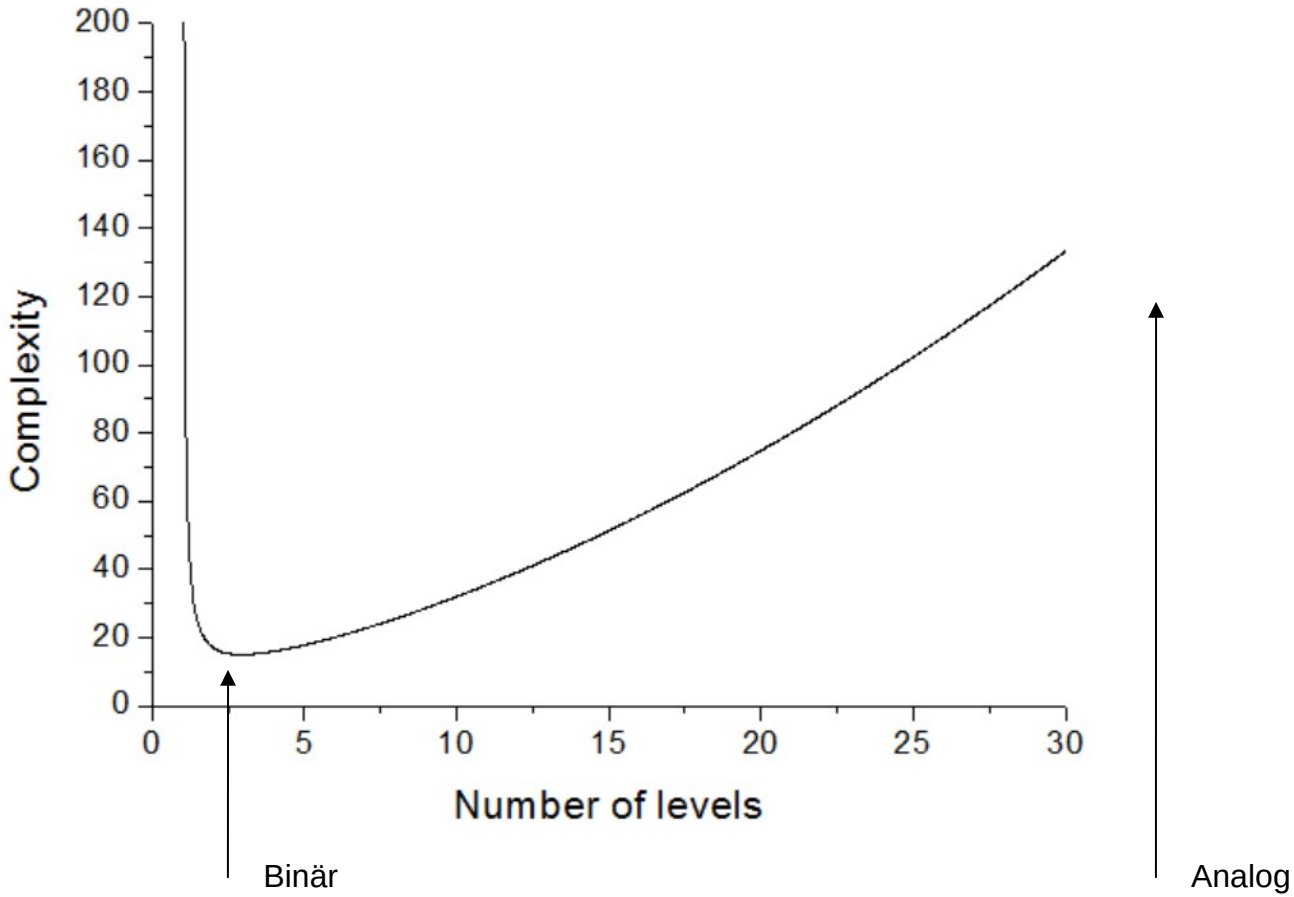
A2	A1	A0
B2	B1	B0
C2	C1	C0

A	B	C
-1	-1	
-1	0	
-1	1	
0	-1	
0	0	
0	1	
1	-1	
1	0	
1	1	

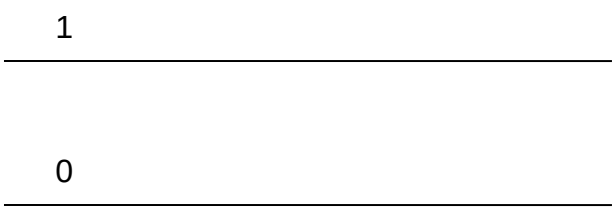
C=3x9=27



- …


Complexity
200
180
160
140
120
100
80
60
40
20
0
0
5
10
15
20
25
30
Number of levels
Binär
Analog

- In digitalen Schaltungen werden die Zahlen 0 oder 1 in Form von elektrischen Potentialen dargestellt



- Annahme: acht-Bit Zahlen.
- Die folgenden Operationen zwischen den Zahlen werden oft gebraucht:
- Addition, Subtraktion, Vergleich (Größer als, Gleichheit), Multiplikation
- Das Ergebnis der Addition und der Subtraktion sind 8-bit Zahlen (kein Übertrag), das Ergebnis der Multiplikation ist 16 Bit Zahl, und die Ergebnisse von Vergleichen sind binäre Zahlen, bzw. Boolesche Variablen

- Beispiel: Komparator
- Wahrheitstabelle
- Eine Zeile für jede Zahlenkombination
- 256 x 256 = 2^16 Kombinationen -> 2^16 Zeilen.

a7	a6	a5	a4	a3	a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	b0	E
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Wir definieren UND (Konjunktion) Funktion n Variablen als Funktion mit dem Wert 1 wenn alle Variablen 1 sind.
- Das Zeichen für Konjunktion ist ^ oder * oder &
- Konjunktion entspricht der Umgangssprache: Ergebnis ist wahr (=1) wenn X0 und X2 und … Xn-1 wahr sind.
- Wir definieren auch ODER Verknüpfung (Disjunktion) mit dem Ergebnis null nur wenn alle Variablen null sind.
- Das Zeichen für Disjunktion ist v oder + oder |
- Es entspricht dem Satz: Ergebnis ist wahr (=1) wenn X1 oder … Xn wahr sind.
- Boolesche und gewöhnliche Logik sind „dual“

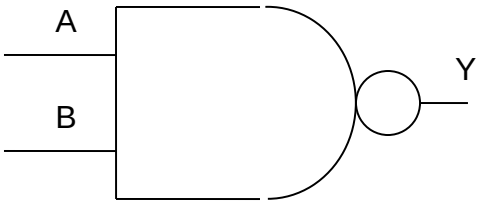
- Die Tabelle für Vergleich zwei 8-Bit zahlen können wir wie folgend als Disjunktive Normalform darstellen:
- Wir suchen alle Zeilen mit dem Ergebnis 1 – es gibt sie 256. Für jede solche Zeile bilden wir eine UND Verknüpfung, die nur für die Variablen-Werte aus dieser Zeile eins ergibt:
- ZB 0000_1111 0000_1111
- Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A2 & A0 & !B0 …
- Zeichen ! bedeutet Negation – wir verwenden es überall dort wo die Variable 0 ist. Die Gesamttabelle ist dann ODER Verknüpfung von allen Ki Funktionen.
- F = K0 | … | K255
- (Alternative Zeichen für die Negation sind ~, -| oder Oberstrich)

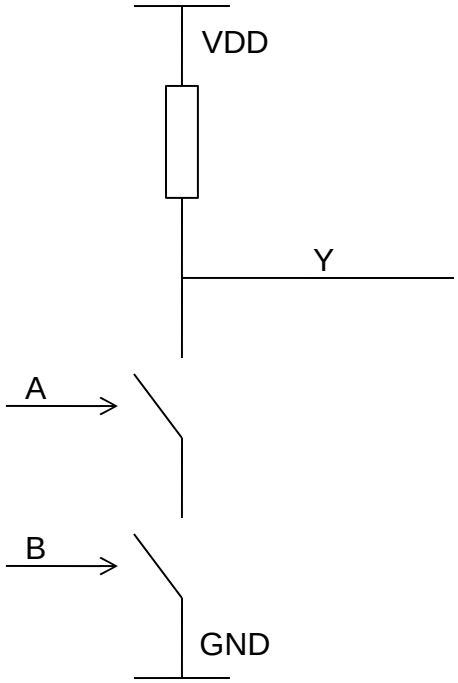
a7	a6	a5	a4	a3	a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	b0	E
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	0

- Die Normalform kann vereinfacht werden (Absorptionsregeln)
- (X & Ai) | (X & !Ai) können wie folgend vereinfacht werden:
- X & (Ai | !Ai) = X & 1 = X
- Distributivgesetz, Äquivalenz Ai |!Ai = 1 und X & 1 = X
- Bei den Paaren von Termen (X & Ai) und (X & !Ai), Ai kann weggelassen werden
- Eine weitere Variante solcher Regel ist
- (X & Ai) | X = X -> (X & Ai) | (X & (Ai | !Ai)) = (X & Ai) | (X & !Ai)
- (X UND etwas) ODER X ist wahr/falsch
- wenn X wahr/falsch ist
- Wenn die Minimierung nicht mehr möglich ist, haben wir die minimale Form.

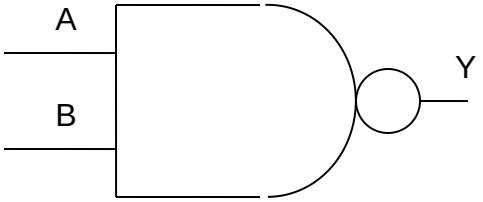
- Eine Disjunktive Normalform kann schaltungstechnisch realisiert werden
- Wie Brauchen Logische Elemente (Gates) - UND, ODER und Negation.
- Eine einfache Möglichkeit Logische Elemente zu realisieren sind die spannungsgesteuerten Schalter
- Ein Schalter ist geschlossen wenn sein Eingangspotential hoch ist – logische 1

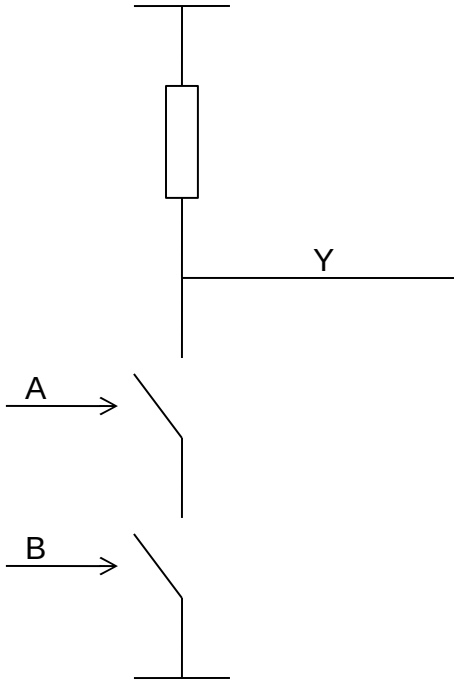
- UND Funktion von Variablen A, B
- Jede Variable ist ein Draht – Kabel („wire“), sein Potential Wert 0 oder 1.
- Zwei Schalter in Serie, an Masse angeschlossen.
- Widerstand zwischen dem Ausgang und der positiven Versorgungsspannung VDD
- Wir definieren das Potential um VDD als logische 1 und das Potential um GND als 0.
- Nur wenn alle Eingänge Eins sind ist auch der Ausgang null, sonst ist es 1.
- NAND



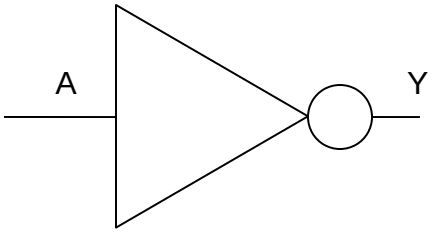


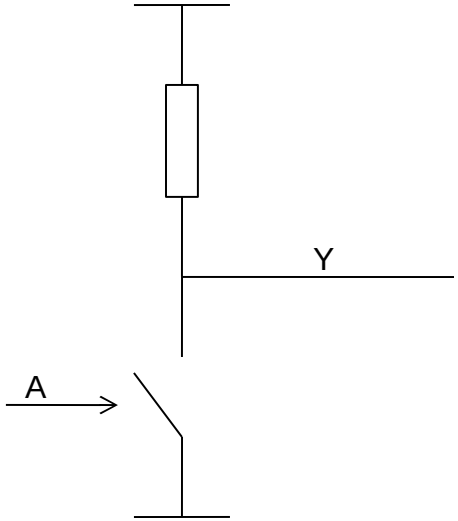
- Annahme: die geschlossenen Schalter sind deutlich niederohmiger als der Widerstand.
- PullUp Widerstand



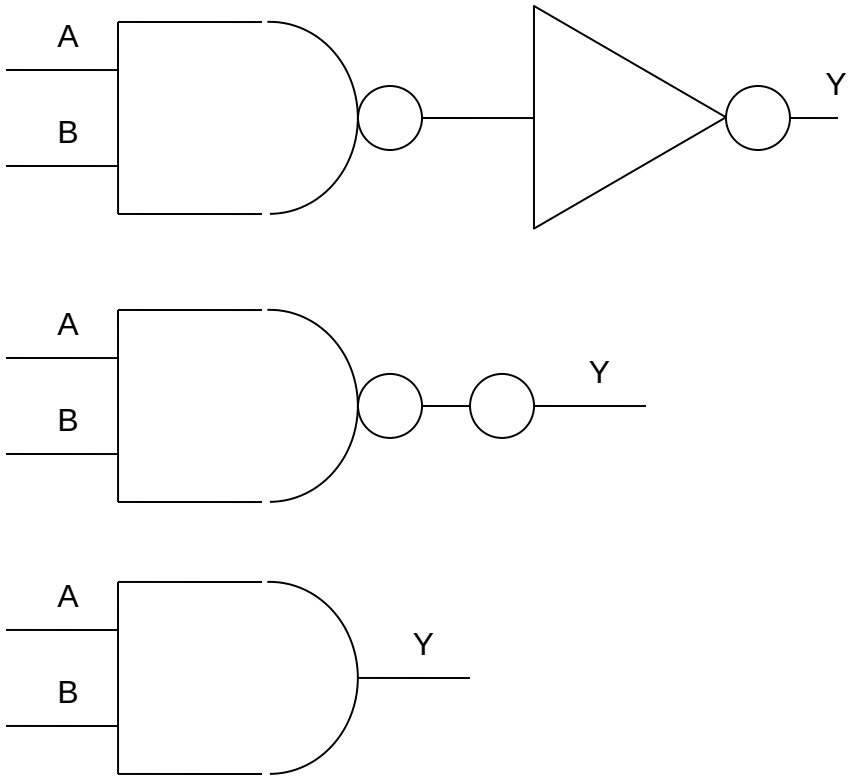


- Inverter

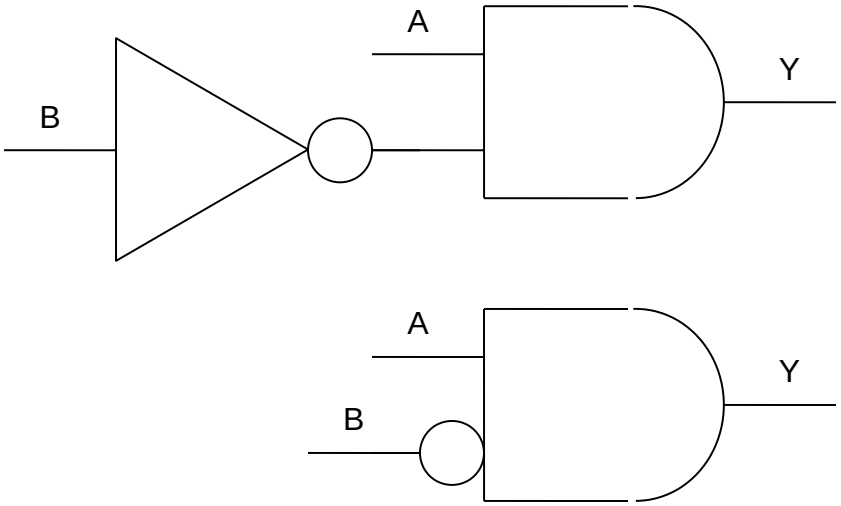




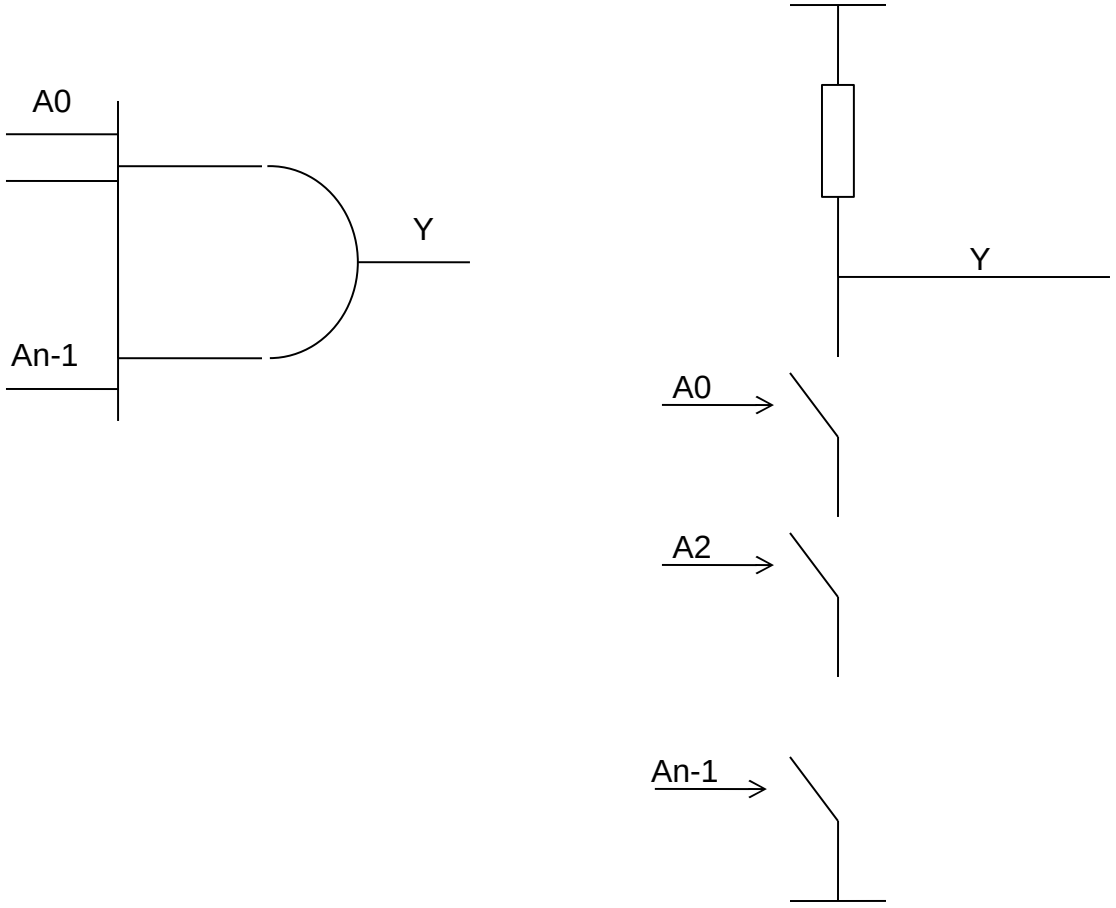
- Inverter und NAND -> UND/AND



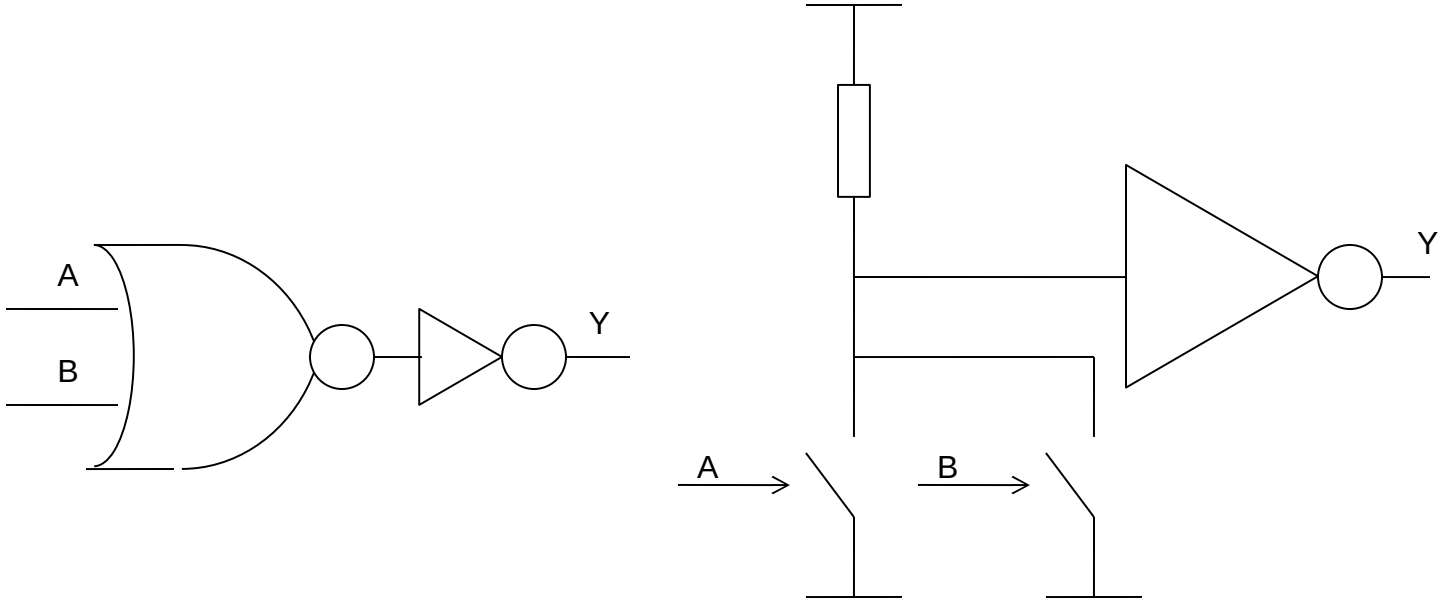
- Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A2 & A0 & !B0 …
- Auf ähnliche Weise können wir die Terme mit negierten Eingangsvariablen realisieren



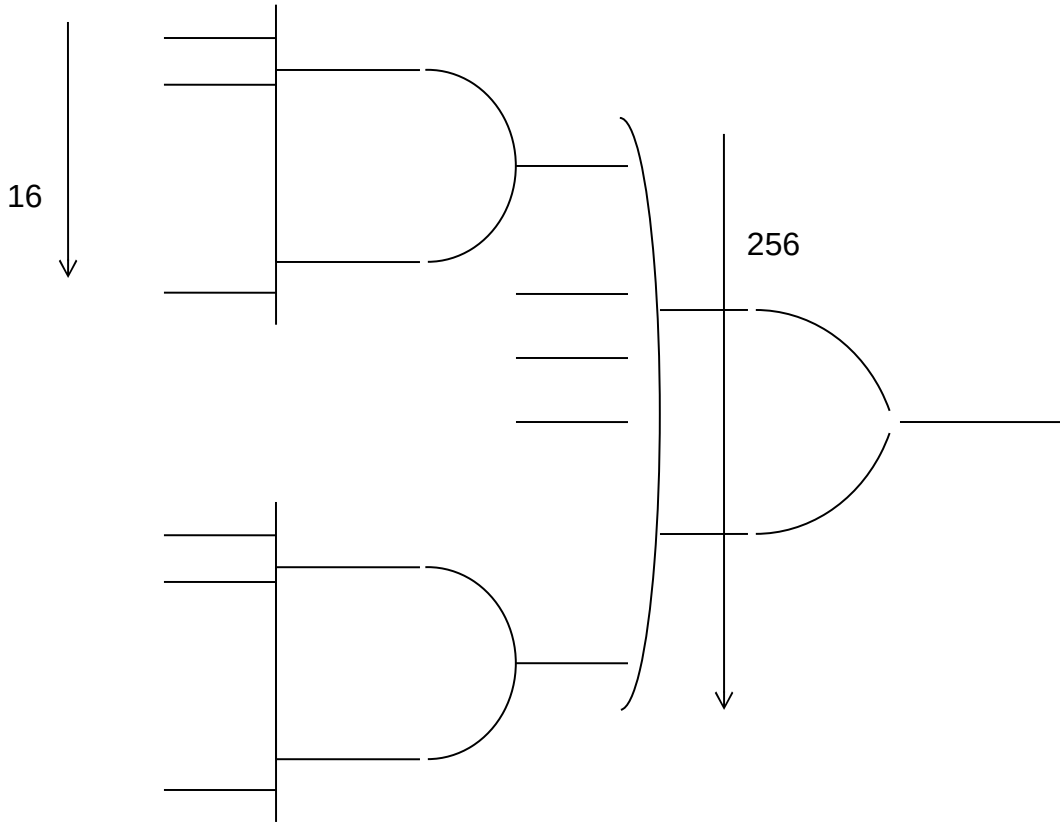
- UND mit mehreren Eingängen



- ODER Verknüpfung kann man auch mit den Schaltern implementieren.
- Die Schalter sind zwischen GND und Ausgang angeschlossen, PullUp Widerstand. Wir bilden zuerst NOR, hängen den Inverter an.

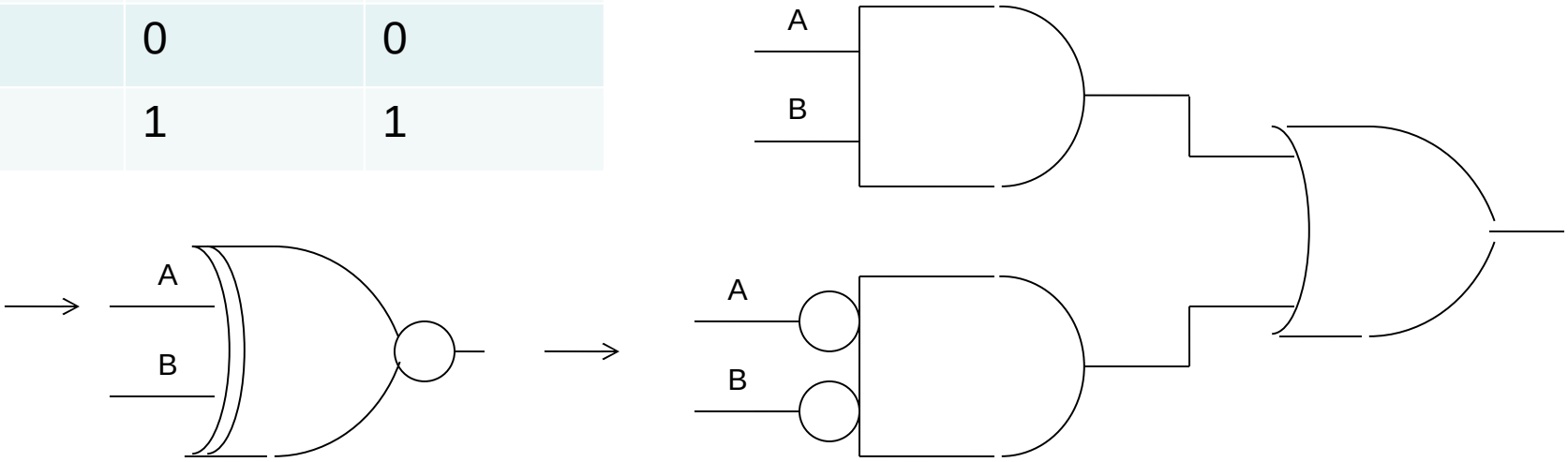


- Unser Komparator braucht also 256 UND Gatter mit jeweils 16 Eingänge und ein ODER mit 256 Eingängen – kompliziert!
- Eine weitere Vereinfachung der Normalform nach den Absorptionsregeln ist in diesem Fall nicht möglich.
- Man kann aber die Terme umzugruppieren – Distributivregeln. Schwierig ohne Rechner/Programm



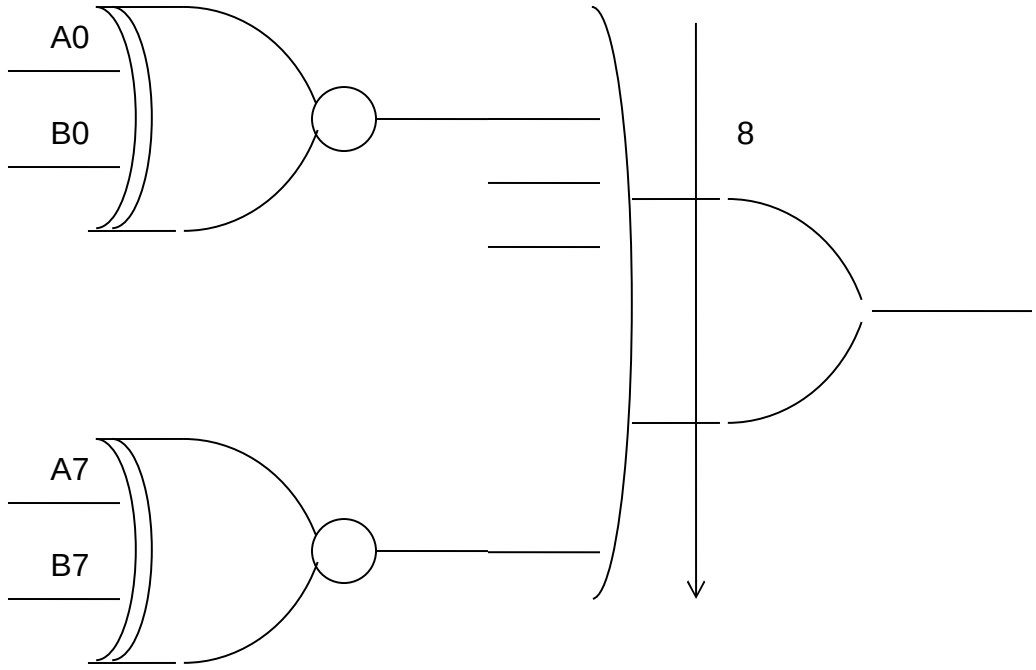
- Einfachere Logik kann entsprechend der gewöhnlichen iterativen Vergleichsmethode gebildet werden – keine Normale Form, mehr Stufen
- Bitweise Vergleich -> wir vergleichen alle Bits einzeln, wenn alle gleich sind -> sind auch die Zahlen gleich
- Äquivalenz: Normalform Y = (a & b) | (!a & !b)

a	b	y
0	0	1
0	1	0
1	0	0
1	1	1

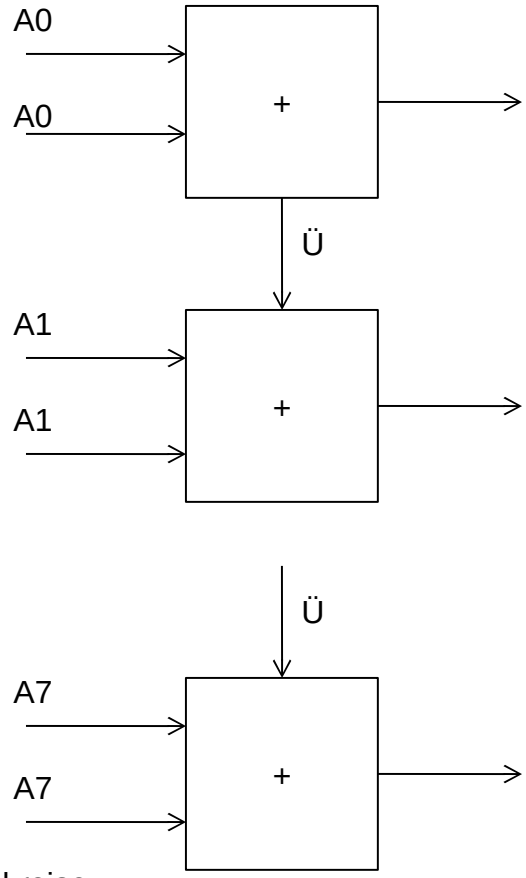


Äquivalenz - EXNOR

- Einfachere Logik kann entsprechend der gewöhnlichen iterativen Vergleichsmethode gebildet werden – keine Normale Form, mehr Stufen
- Bitweise Vergleich



- Weiteres Beispiel ist ein 8-Bit Addierer.
- Auch hier bietet sich an, gewöhnlichen Algorithmus für die Addition mehrstelligen Zahlen, den wir aus der Schule kennen, schaltungstechnisch zu implementieren.
- Binäre Zahlen



- Tabelle für die Addition von zwei Bits a und b
- c ist der Übertrag aus der vorherigen Addition
- Summe = !c & (a exor b) | c & (a == b)

C=0

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

C=1

a	b	y
0	0	1
0	1	0
1	0	0
1	1	1

- Übertrag
- Cout = !c & (a & b) | c & (a | b)
- Hier & vor | - für Programmiersprachen soll immer geprüft werden

C=0

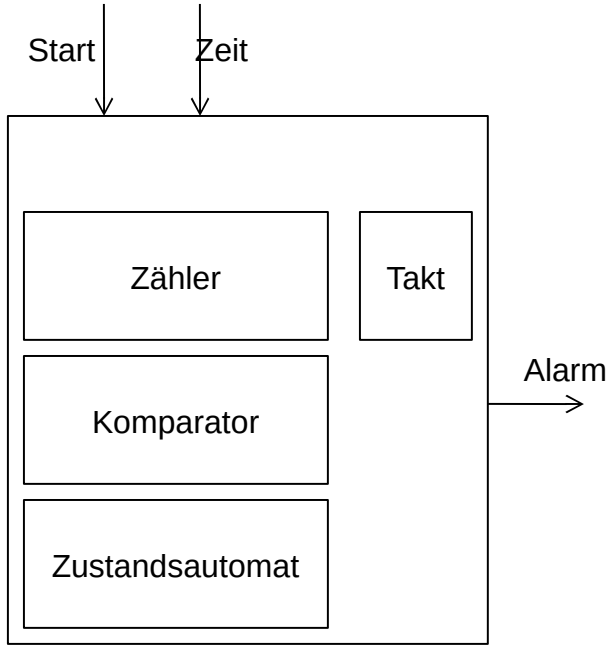
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

C=1

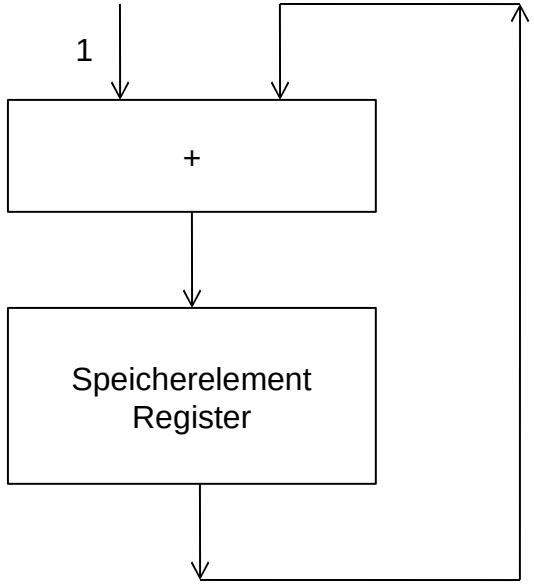
a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

- Kombinatorische Logik.
- Der Ausgang der Schaltung ist definiert wenn man die Eingänge kennt.
- Andere Art: Schaltungen mit Speicherelementen, mit denen man, zum Beispiel, zyklische Operationen durchführen kann, Zustandsautomaten oder Programme realisiert
- Sequenzielle Schaltungen - Ausgang hängt nicht nur von den momentanen Eingangswerten sondern auch von der Vorgeschichte des Systems.

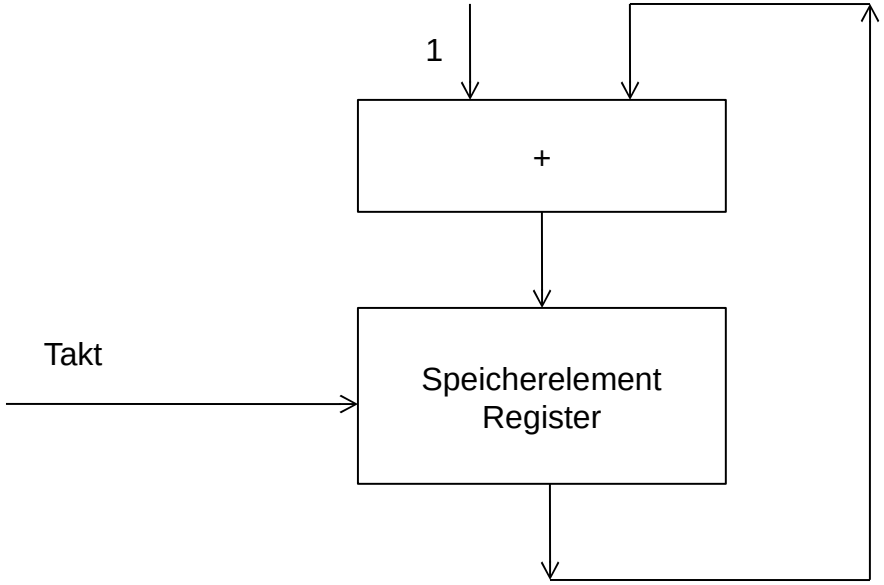
- Beispiel - Timer
- Der Eingang: die eingestellte Zeit – z.B. achtstellige binäre Zahl, und ein Start-Knopf. Der Ausgang ist ein Alarm-Signal
- Solche Systeme brauchen 1) ein internes Taktsignal, also einen Oszillator.
- 2) einen Zähler
- 3) Komparator und Zustandsmaschine



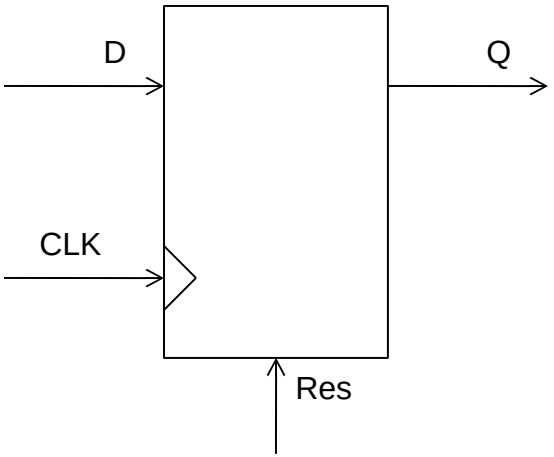
- Zähler: Addierer + Speicherelement wo das Ergebnis gespeichert wird.

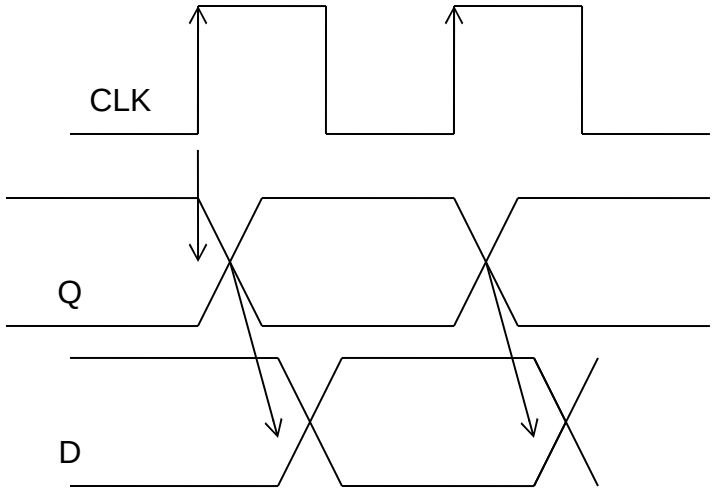


- Zähler: Addierer + Speicherelement wo das Ergebnis gespeichert wird.
- Wie wird der Zähler angesteuert?
- Register aus FlipFlops (Speicherzellen)

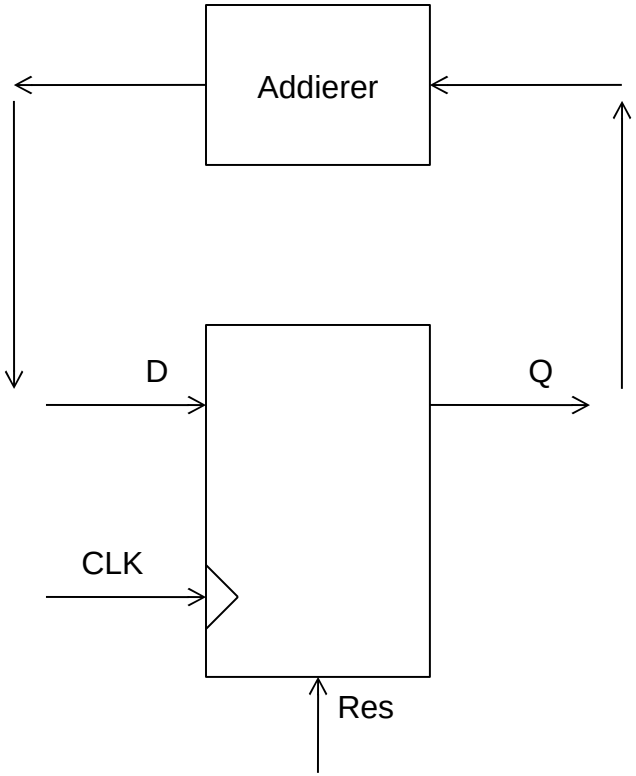


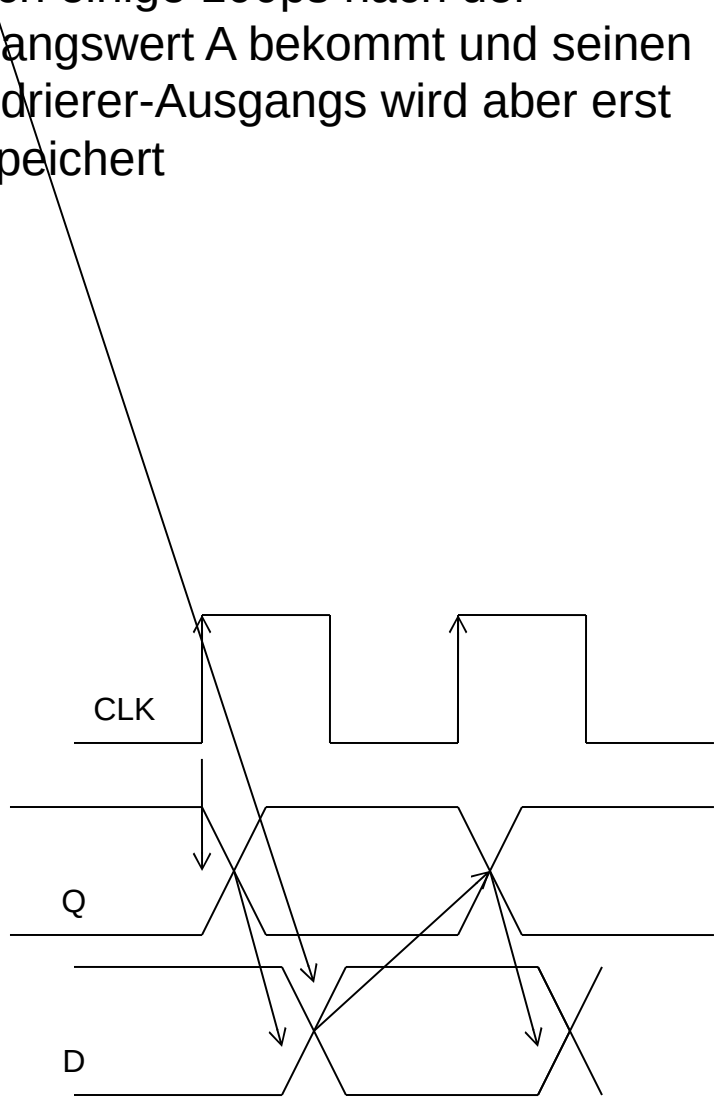
- Die Flip Flops haben einen Eingang, Ausgang, einen Takteingang und oft ein Reset Signal.
- Flipflops haben die folgende Eigenschaft. Der Wert am Eingang wird im Moment der steigenden Taktflanke gespeichert. Der gespeicherte Wert taucht auf dem Ausgang eine gewisse kurze Zeit danach auf, etwa ~ n x 100ps. => Auf jede steigente Taktflanke erhöht sich der Zustand des Zählers.





- Der Eingang des Registers ändert sich auch einige 100ps nach der Taktflanke, da der Adierer den neuen Eingangswert A bekommt und seinen Ausgang anpasst. Diese Änderung der Addrierer-Ausgangs wird aber erst auf die nächste Taktflanke in Register gespeichert

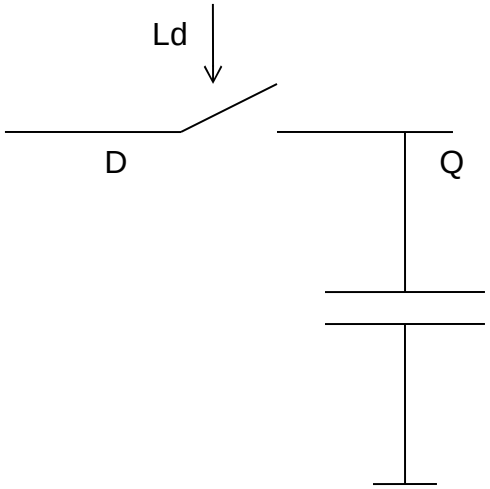




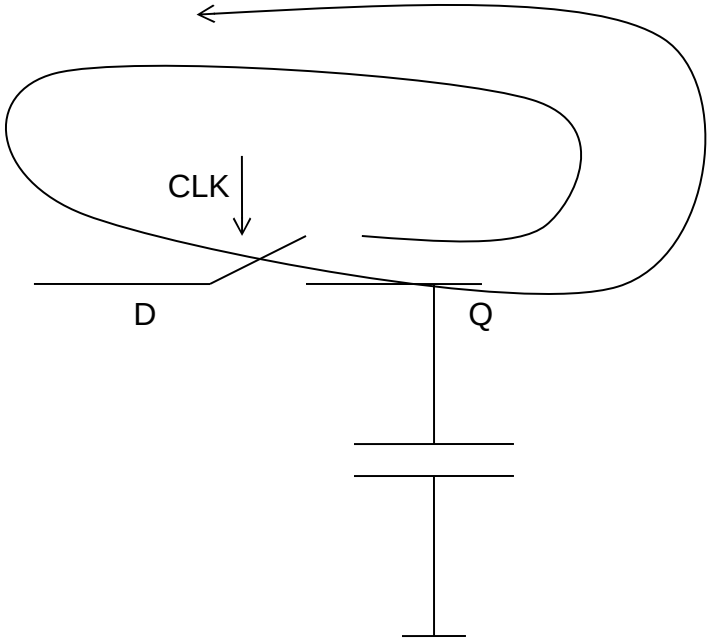
- Hardware-Programmiersprache:

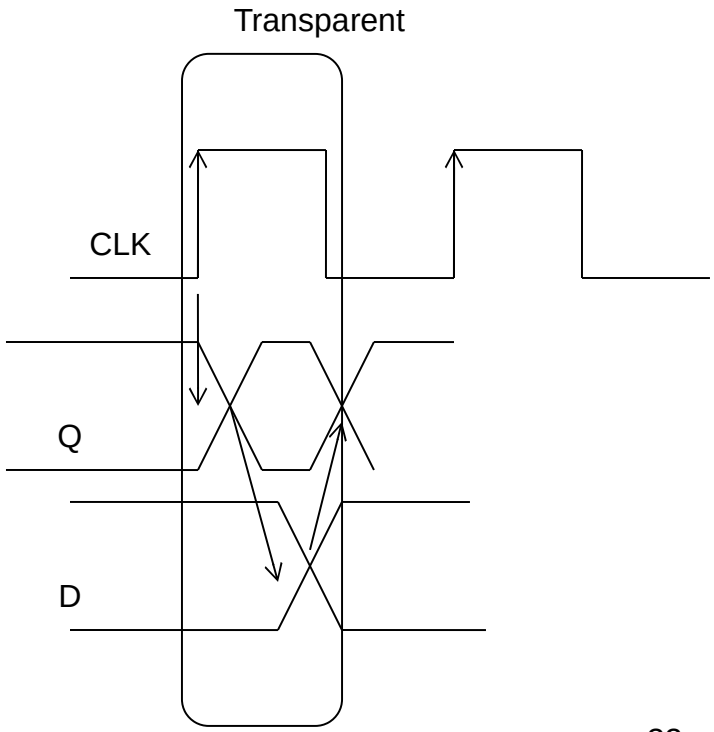
```
always @ (posedge CLK) begin
        A <= A + 1
end
```

- Wie realisieren wir ein Flipflop?
- Am einfachsten stellen wir uns eine Speicherzelle wie einen getakteten Kondensator vor. Wenn der Schalter geschlossen ist, verbinden wir den Eingang mit einem Kondensator. Der Kondensator wird auf das Eingangspotential aufgeladen.
- Wenn der Schalter geöffnet wird, behält der Kondensator das Potential. Das logische Niveau wird auf diese Weise gespeichert.
- DRAM Zellen.



- Problem:
- Nach der steigenden Taktflanke wird der Eingang gespeichert - OK. Das Flip-Flop aus einem Kondensator würde jede weitere Änderung am Eingang ebenfalls speichern, bzw. das anfangs gespeicherte Wert überschreiben, solange Taktsignal eins ist. (Race Condition)
- Der gespeicherte Zustand soll sich bis zur nächsten Talkflanke nicht ändern, auch wenn sich der Eingang ändert





- Lösung
- Zwei DRAM Zellen hintereinander zu schalten