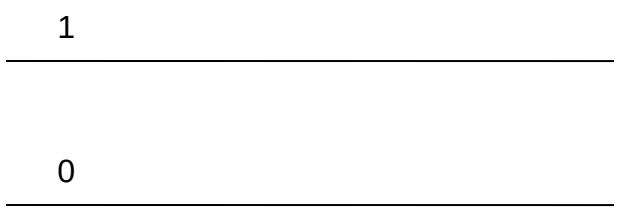
Vorlesung 1

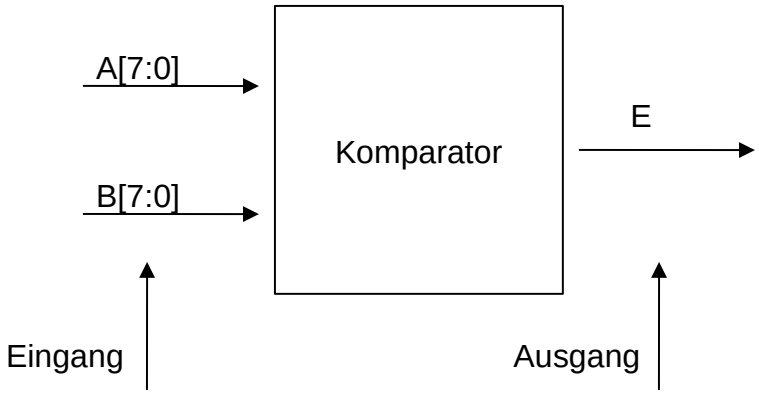
- Beispiel Komparator: Realisierung mit normaler Form
- UND, ODER, normale Form (Definitionen)
- Vereinfachung von normalen Formen
- Realisierung von Gattern mithilfe von Schaltern und Widerständen: NAND, AND, Inverter, NOR, OR
- Beispiel Komparator: Bit-Weise Implementierung
- Beispiel: Adierer
- Kombinatorische und Sequenzschaltungen
- Beispiel: Timer
- Komponenten: Zähler, Speicherzelle
- DRAM -> Flipflop

Binäre Zahlen

- In digitalen Schaltungen werden die Zahlen 0 oder 1 in Form von elektrischen Potentialen dargestellt
- 0 ist (in CMOS-Implementierung) das Potential der Masse - GND
- 1 ist das Potential vor Versorgungsspannung - VDD

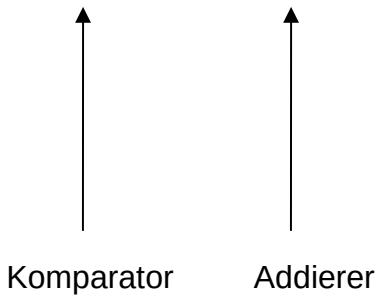


- Annahme: acht-Bit Zahlen
- Die folgenden Operationen zwischen den Zahlen werden oft benutzt:
- Addition, Subtraktion, Vergleich (>, =), Multiplikation
- Das Ergebnis der Addition und der Subtraktion sind 8-bit Zahlen
- Ergebnis der Multiplikation ist 16-bit Zahl
- Ergebnisse von Vergleichen sind 0 oder 1 (boolesche Variablen)



- Frage: Wie sieht die Schaltung aus, die dem Code unten entspricht?

if(A<100) A <= A + 1



Beispiel: Komparator

- Eine logische Funktion wird mit der Wahrheitstabelle beschrieben
- Die Tabelle enthält eine Zeile für jede Zahlenkombination am Eingang
- 256 x 256 = 2^16 Kombinationen -> 2^16 Zeilen (etwa 64 000)

Ergebnis

Variable 1								Variable 2								
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

UND, ODER

- Wir definieren UND-Funktion (Konjunktion) von n Variablen als Funktion mit dem Wert 1 wenn alle Variablen 1 sind (Ergebnis ist wahr (=1) wenn X0 *und* X2 *und* … Xn-1 wahr sind)
- Das Symbolzeichen für UND ist ^ , * , &
- Wir definieren auch ODER Verknüpfung (Disjunktion) – das Ergebnis ist 0nur wenn alle Variablen 0 sind (Ergebnis ist 1 wenn X1 *oder* … Xn 1 sind)
- Das Zeichen für Disjunktion ist v , + , |

Disjunktive Normalform (DNF)

- 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 Ergebnis = 1 nur für die Variablen-Werte aus dieser Zeile gibt:
- ZB für die Zeile 0000_1111 0000_1111
- Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A1 & 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 ~, -|)

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

Vereinfachungen der Normalform

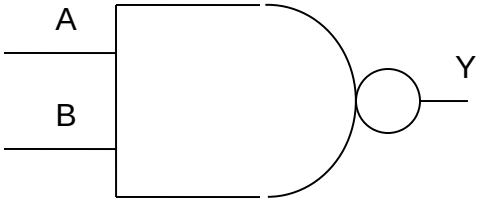
- Die Normalform kann mithilfe von Absorptionsregeln vereinfacht werden
- (X & Ai) | (X & !Ai) = X & (Ai | !Ai) = X & 1 = X
  - In diesem Beweis haben wir Distributivgesetz, Äquivalenz Ai |!Ai = 1 und X & 1 = X verwendet
- Eine weitere Variante: (X & Ai) | X = X
- (X UND etwas) ODER X ist wahr/falsch wenn X wahr/falsch ist
- Wenn die Minimierung nicht mehr möglich ist, ist eine DNF minimal

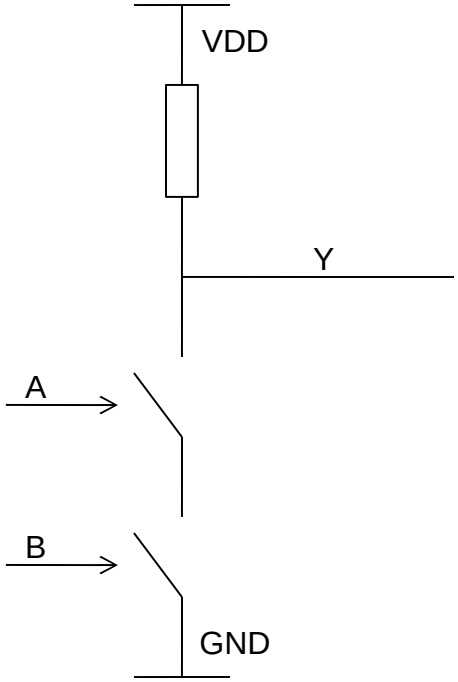
Realisierung als Normalform

- Eine Disjunktive Normalform kann schaltungstechnisch realisiert werden.
- Dafür brauchen wir logische Schaltungen (Gates/Gatter) für UND-, ODER-Funktionen und Negation.
- Eine einfache Möglichkeit Logische Schaltungen zu realisieren sind die spannungsgesteuerten Schalter
- Ein Schalter ist geschlossen wenn sein Eingangspotential hoch ist – logische 1

NAND - Implementierung

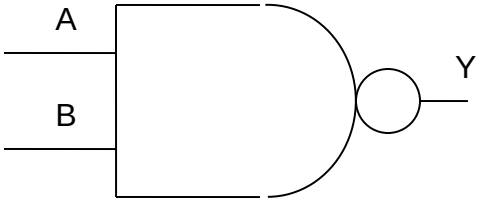
- UND Funktion von Variablen A, B
- Eine Variable (logisches Signal) wird durch Potential auf einer Leitung mit dem Wert 0 (GND) oder 1 (VDD) dargestellt
- Wir verwenden: zwei Schalter in Serie und einen Widerstand zwischen dem Ausgang und der positiven Versorgungsspannung VDD
- Nur wenn alle Eingänge 1 sind ist auch der Ausgang 0, sonst ist es 1.
- Es ist eine negierte UND Funktion: NAND

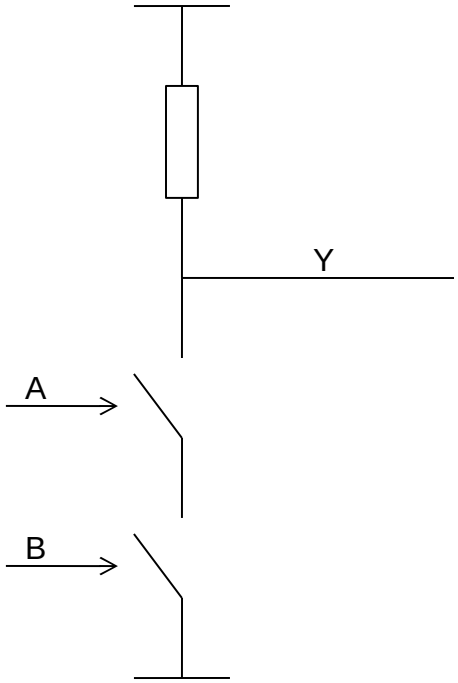




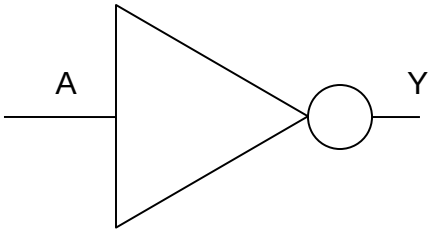
NAND - Implementierung

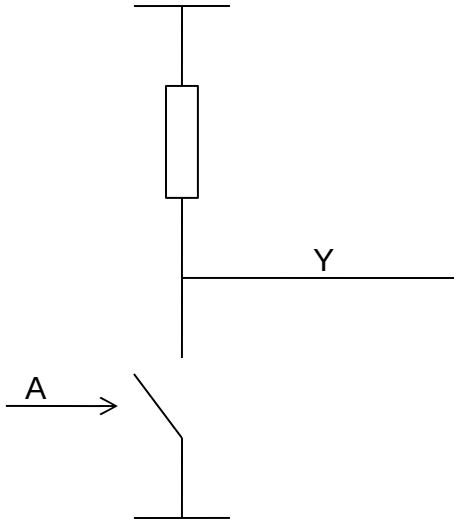
- Annahme: die geschlossenen Schalter sind deutlich niederohmiger als der Widerstand.
- PullUp Widerstand





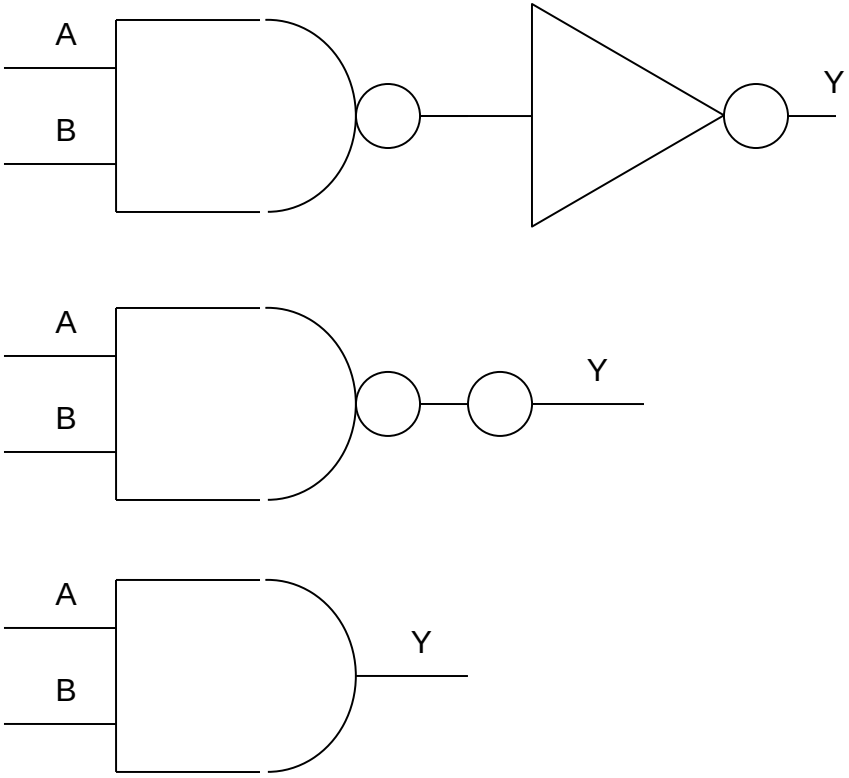
- Inverter



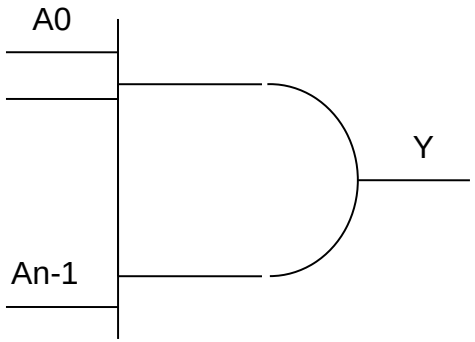


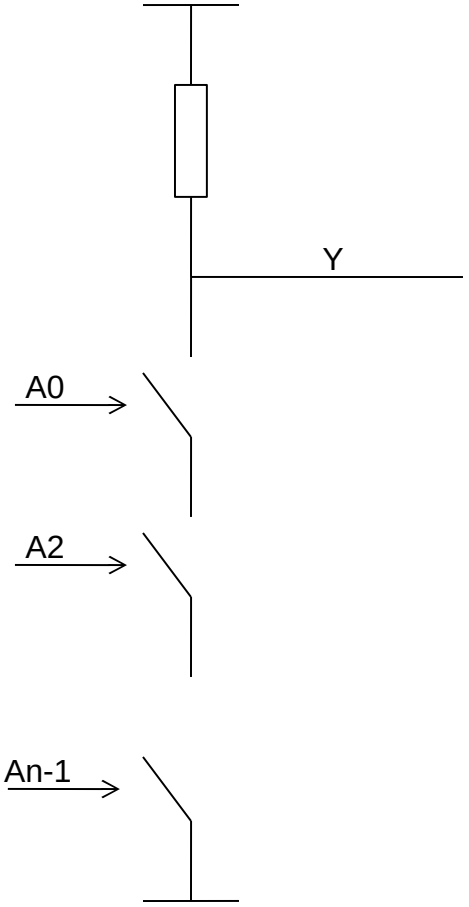
AND - Implementierung

- Inverter und NAND -> UND/AND



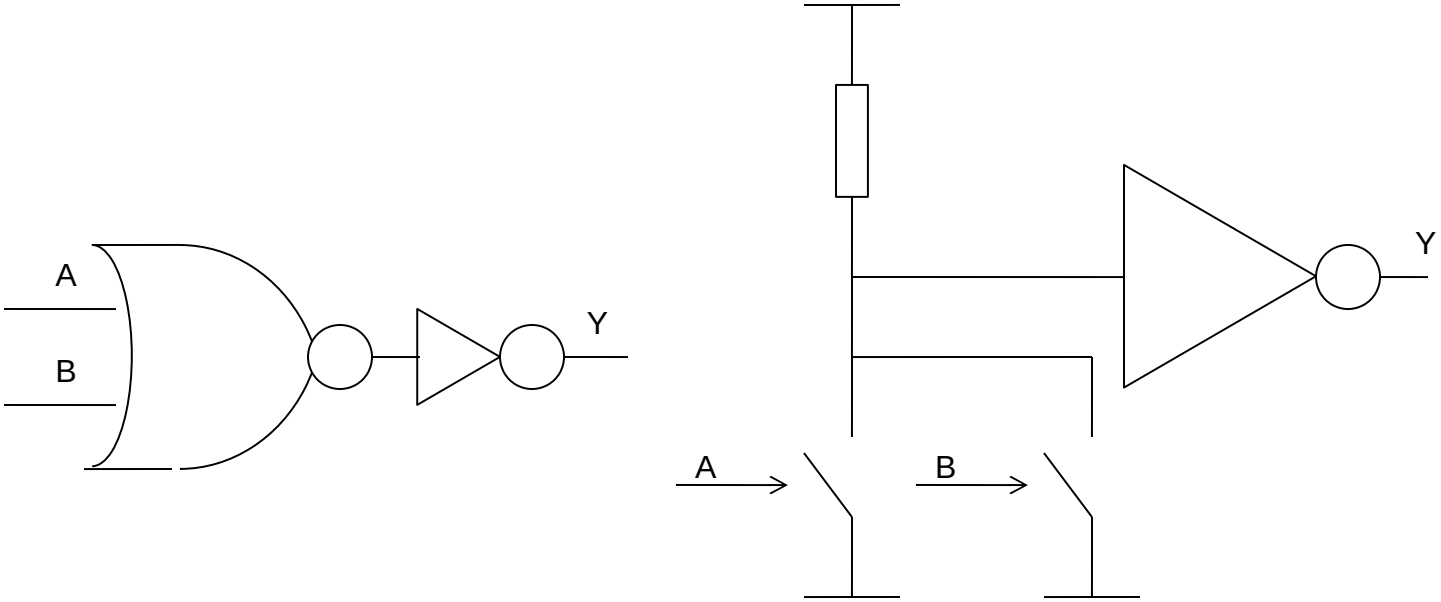
- UND mit mehreren Eingängen



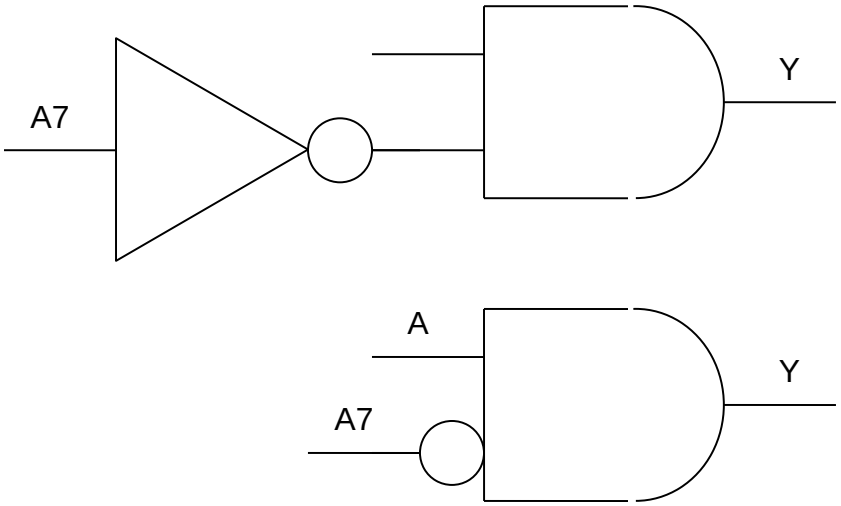


ODER

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

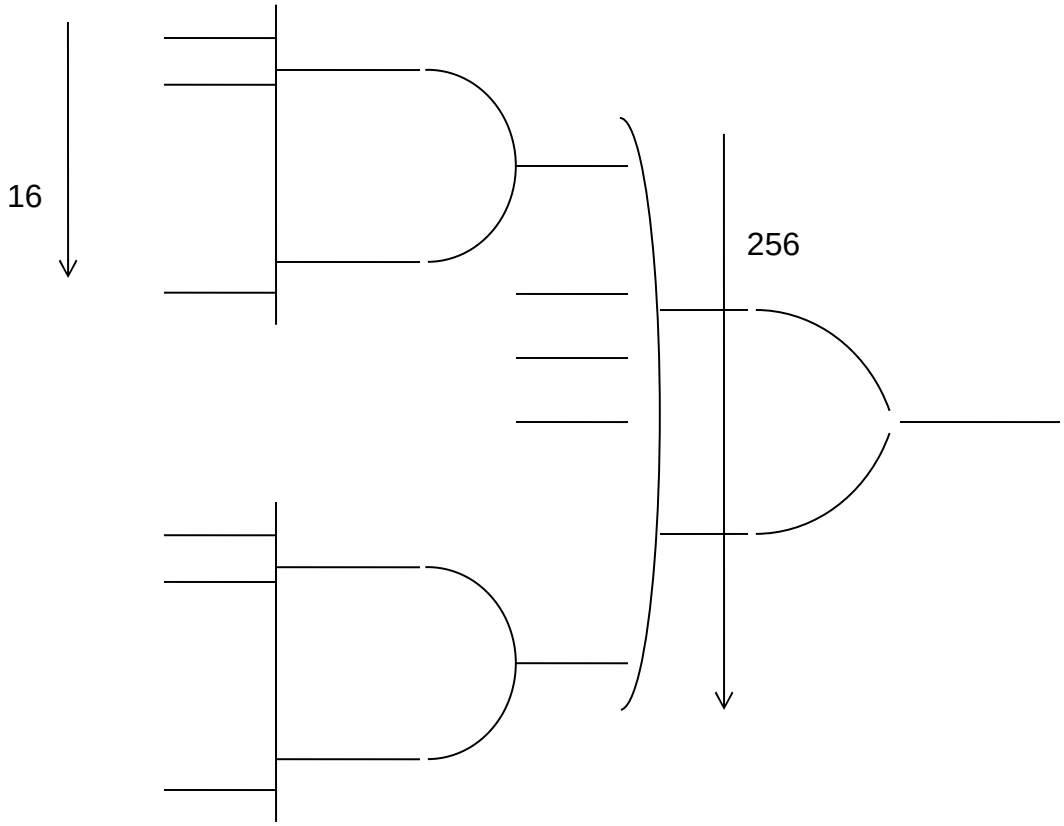


- Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A2 & A0 & !B0 …
- Mithilfe von Invertern realisieren wir die negierten Eingangsvariablen



Komparator: Realisierung als Normalform

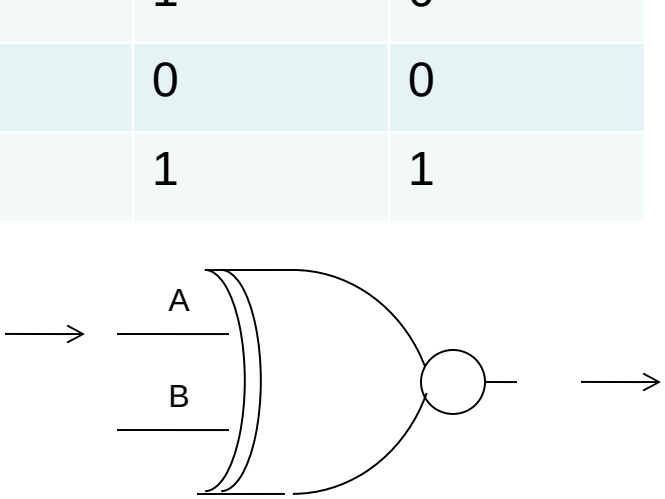
- Der Komparator braucht 256 UND Gatter mit jeweils 16 Eingängen und ein ODER mit 256 Eingängen – kompliziert!
- Eine weitere Vereinfachung der Normalform nach den Absorptionsregeln ist in diesem Fall nicht möglich

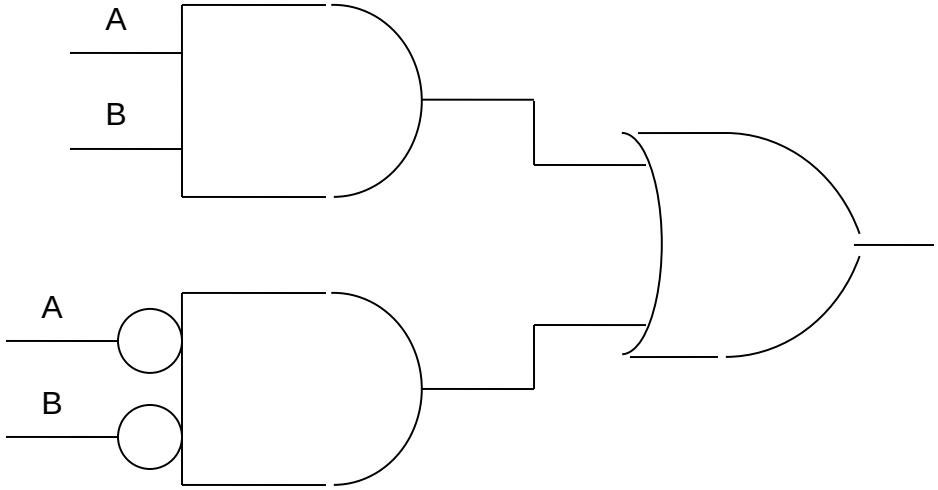


Komparator: Bit-Weise Implementierung

- Einfachere logische Schaltung 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

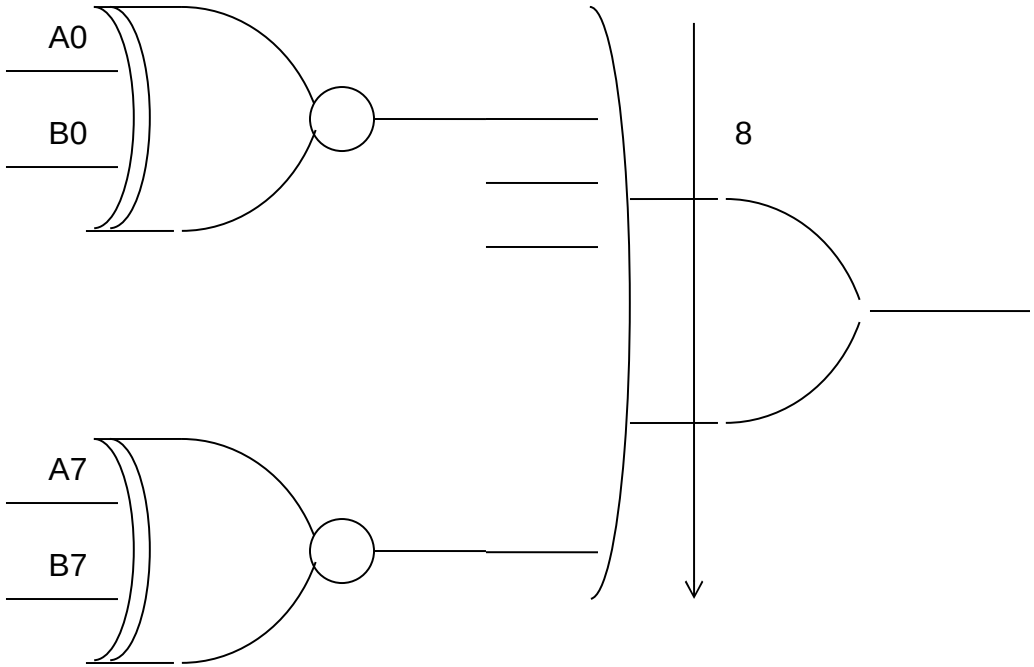




Äquivalenz - EXNOR

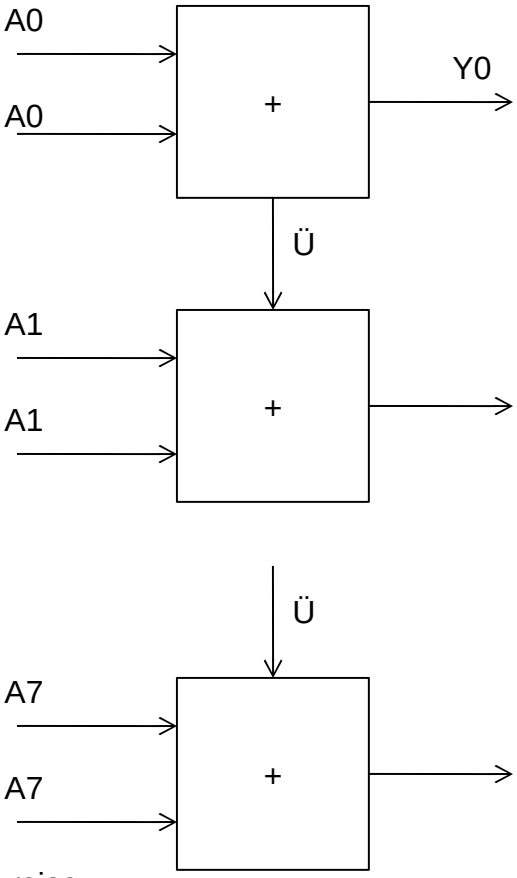
Komparator: Bit-Weise Implementierung

- Bitweise Vergleich



8-Bit Addierer

- 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)

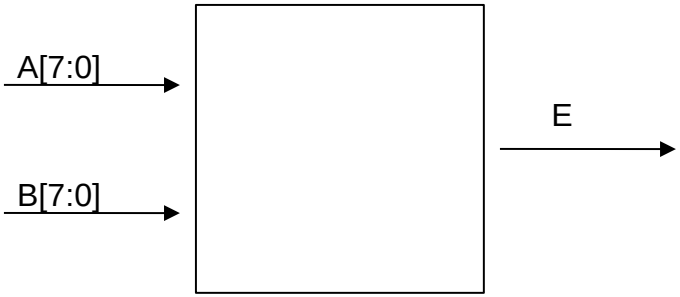
C=0

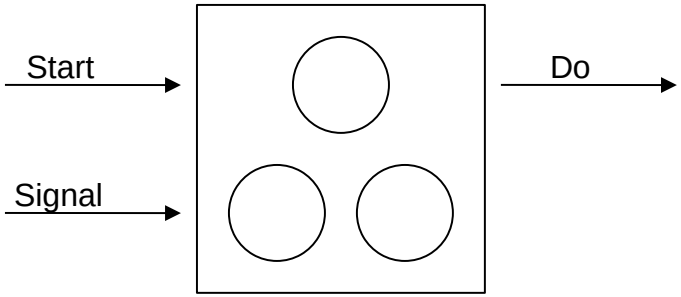
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

C=1

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

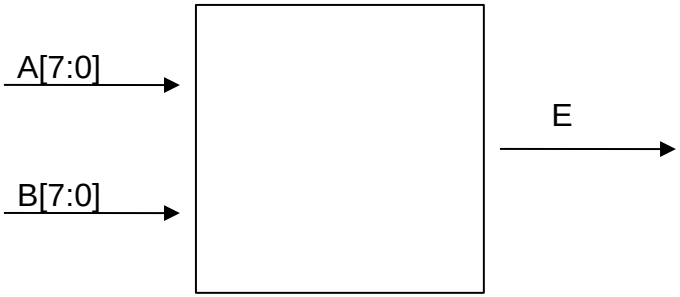
- Kombinatorische und Sequenzschaltungen

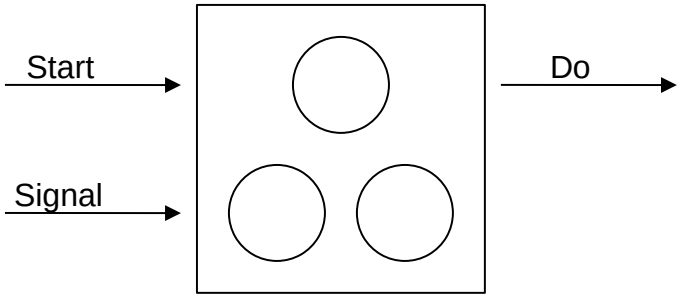




Sequenzielle Schaltungen

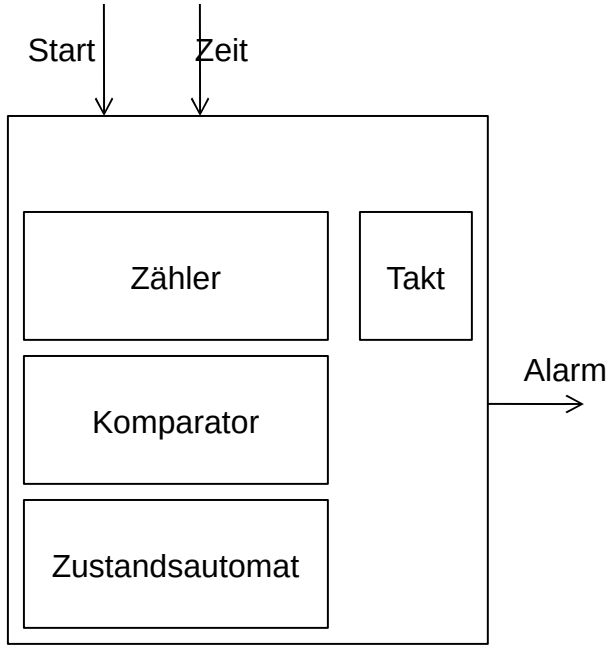
- Kombinatorische Logik: Der Ausgang der Schaltung ist definiert wenn man die Eingänge kennt
- Andere Art: Schaltungen mit Speicherelementen, mit denen man, zum Beispiel, zyklische Operationen durchführen kann, Zustandsautomaten oder Programme realisiert
- Sequenzielle Schaltungen - Ausgang hängt nicht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



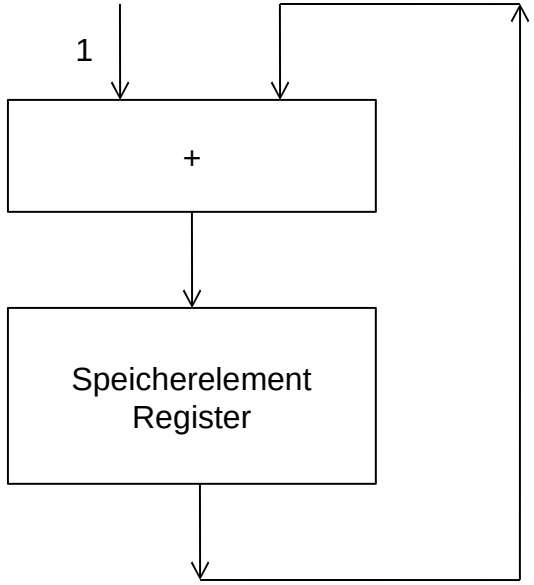


Beispiel - Timer

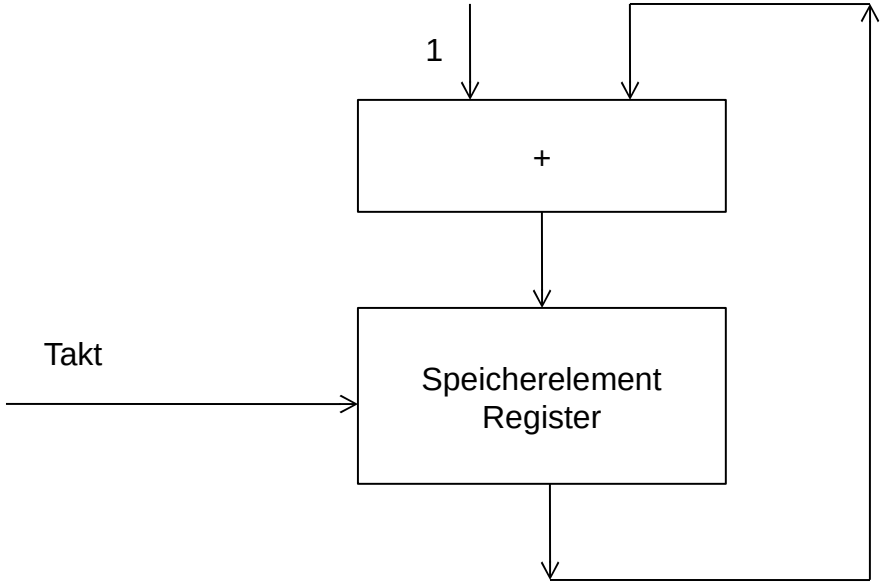
- Beispiel - Timer
- Der Eingang: die eingestellte Zeit – z.B. achtstellige binäre Zahl, und ein Start-Knopf. Der Ausgang ist ein Alarm-Signal
- Solche Systeme brauchen 1) ein internes Taktsignal, also einen Oszillator.
- 2) einen Zähler
- 3) Komparator und Zustandsmaschine



- Zähler: Addierer + Speicherelement in dem das Ergebnis gespeichert wird.

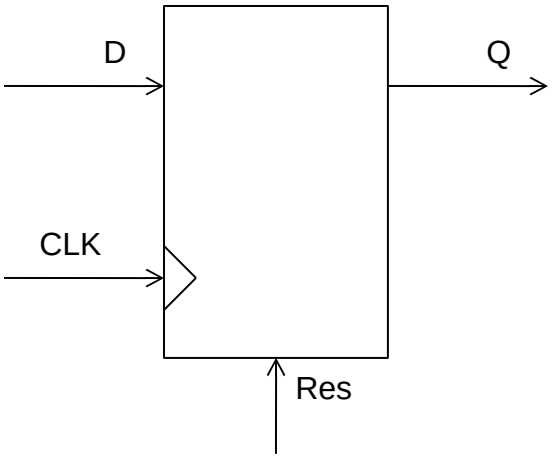


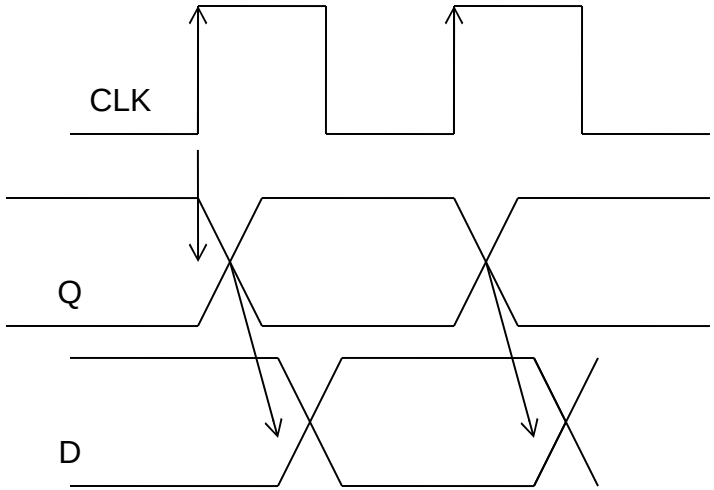
- Wie wird der Zähler angesteuert?
- Register aus Flipflops (Speicherzellen)



Speicherzellen

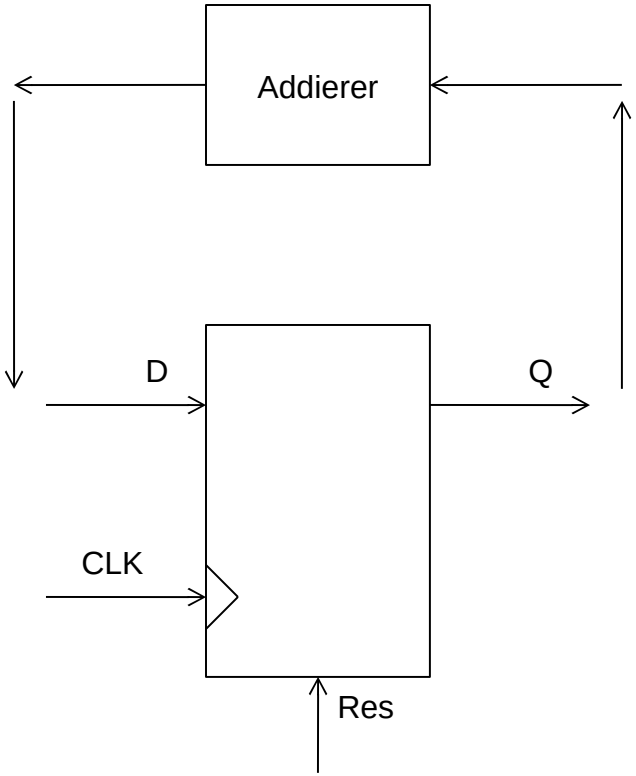
- Die Flipflops haben einen Eingang, Ausgang, einen Takteingang
- Flipflops haben die folgende Eigenschaft. Der Wert am Eingang wird im Moment der steigenden Taktflanke gespeichert. Der gespeicherte Wert taucht auf dem Ausgang eine kurze Zeit danach auf, etwa ~ n x 100ps

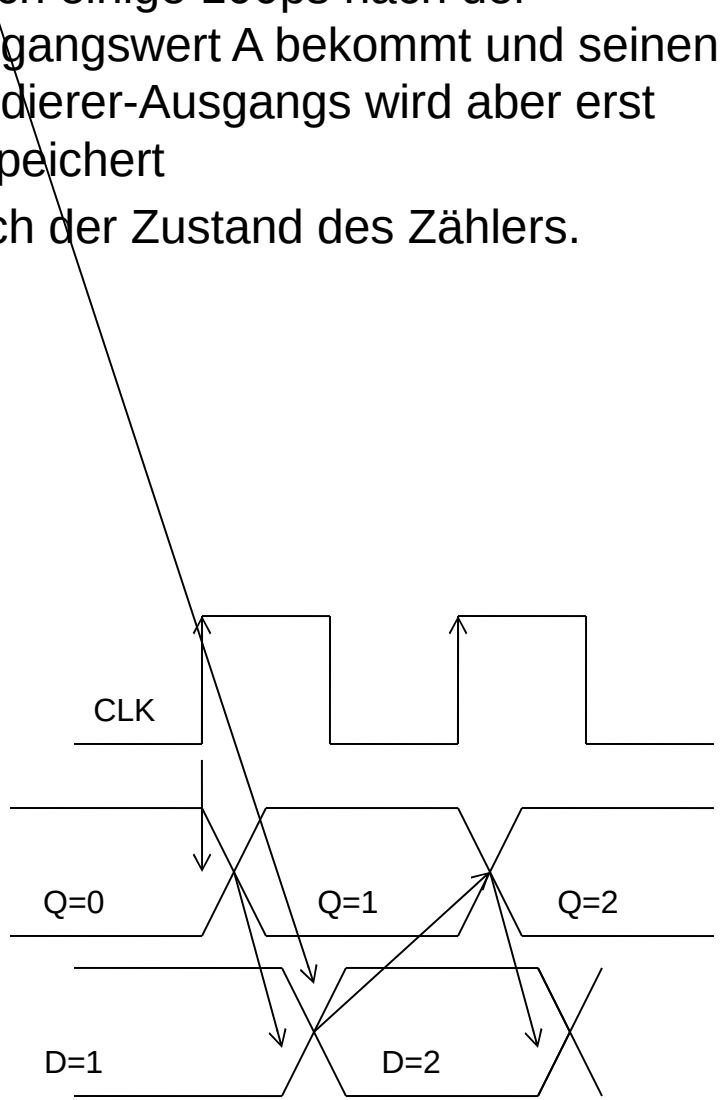




Speicherzellen

- Der Eingang des Registers ändert sich auch einige 100ps nach der Taktflanke, da der Addierer den neuen Eingangswert A bekommt und seinen Ausgang anpasst. Diese Änderung der Addierer-Ausgangs wird aber erst auf die nächste Taktflanke in Register gespeichert
- => Auf jede steigente Taktflanke erhöht sich der Zustand des Zählers.





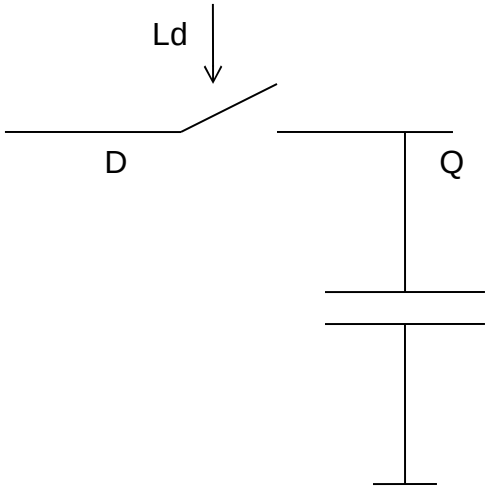
Speicherzellen

- Hardware-Programmiersprache:

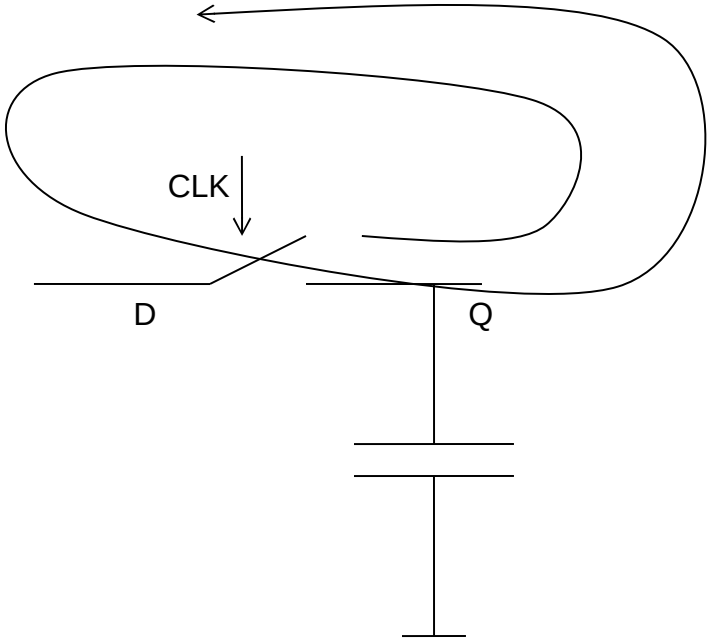
```
always @ (posedge CLK) begin
        A <= A + 1
end
```

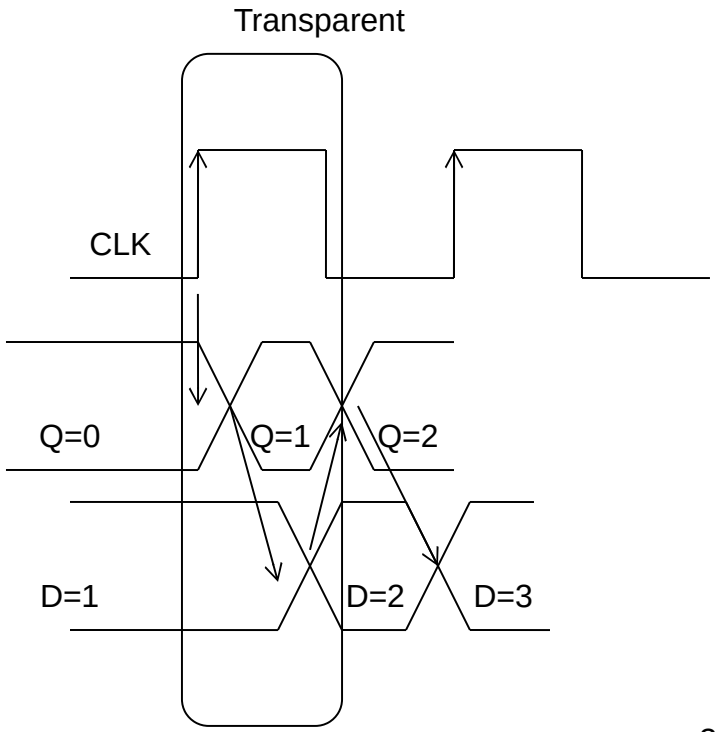
Speicherzellen

- 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 Flipflop 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