Design digitaler Schaltungen

Vorlesung 1

Themen dieser Vorlesung sind

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

Komparator: Bit-Weise Implementierung

Addierer

Kombinatorische und Sequenzschaltungen

Timer, Komponenten: Zähler, Speicherzelle

DRAM und Flipflop

Einführung

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

Logisch 0 ist (in CMOS-Implementierung) das Potential der Masse – GND.

Logisch 1 ist das Potential der Versorgungsspannung – VDD.

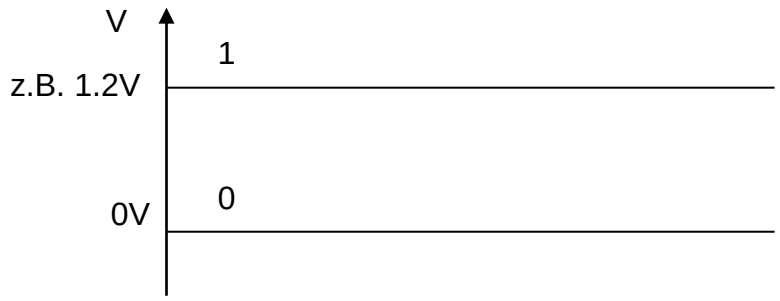

Abbildung 1: Logische Niveaus

Betrachten wir die acht-Bit Zahlen.

Die folgenden Operationen zwischen den Zahlen werden oft gebraucht:

Addition, Subtraktion, Vergleich („Größer als“, „=“), Multiplikation.

Das Ergebnis der Addition und Subtraktion sind 8-bit Zahlen (wir vernachlässigen den Übertrag), das Ergebnis der Multiplikation ist 16 Bit Zahl, und die Ergebnisse von Vergleichen sind binäre Zahlen, bzw. eine Boolesche Variablen („wahr“, „nicht wahr“).

Betrachten wir, als Beispiel, den Komparator (Abbildung 2).

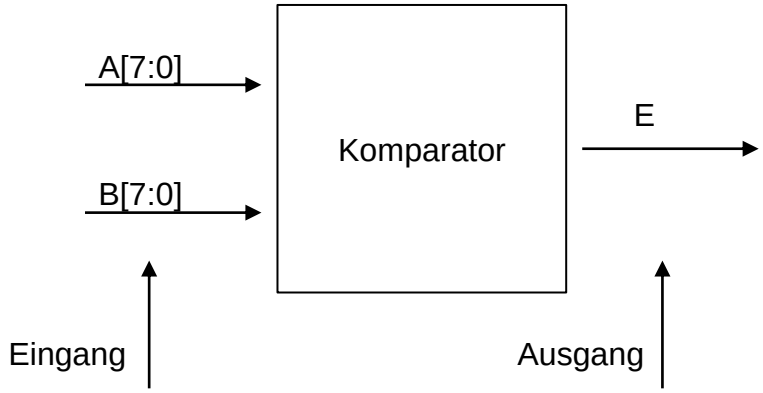

Abbildung 2: Komparator

Um den Komparator zu beschreiben, könnten wir eine große Ergebnisstabelle (Wahrheitstabelle) schreiben, wo wir eine Zeile für jede Zahlenkombination A und B haben. Es gibt 256 × 256 = 2^{16} solchen Kombinationen also 2^{16} Zeilen (~64k).

Variable 1								Variable 2								Ergebnis
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

Abbildung 3: Wahrheitstabelle

Normalformen

Wie definieren UND (Konjunktion) Funktion (Verknüpfung) von n Variablen als Funktion mit dem Wert 1, wenn alle Variablen 1 sind.

Konjunktion entspricht dem Satz: Ergebnis ist „wahr“ (=1) wenn X1 und X2 und … Xn „wahr“ sind.

Das Zeichen für Konjunktion ist ^ oder * oder &.

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: das Ergebnis ist „wahr“ (=1) wenn X1 oder … Xn „wahr“ sind.

Die Tabelle für Vergleich zwei 8-Bit zahlen können wir in Abbildung 4 als Disjunktive Normalform darstellen.

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

Abbildung 4: Wahrheitstabelle

Wir suchen alle Zeilen mit dem Ergebnis 1 – es gibt sie 256. Für diese Zeilen bilden wir eine UND Verknüpfung, die nur für die Variablen-Werte aus der Zeile 1 ergibt:

Z.B. der Zeile 0000_1111_0000_1111 entspricht folgende UND Funktion:

Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A1 & A0 & !B7 & !B6 & !B5 & !B4 & B3 & B2 & B1 & B0

Zeichen „!“ bedeutet Negation – wir verwenden es dort, wo die Variable 0 ist. Die Gesamttabelle ist dann ODER Verknüpfung von allen Ki Funktionen.

F = K1 | … | K256

Andere Zeichen für die Negation sind ~, ¬ oder Negationsstrich.

Vereinfachung

Die Normalform kann mithilfe von Absorptionsregeln vereinfacht werden.

Wenn wir z.B. zwei folgende Terme haben

(X & Ai) | (X & !Ai)

sie können wie folgend vereinfacht werden:

(X & Ai) | (X & !Ai) = X & (Ai | !Ai) = X & 1 = X

Im Beweis haben wir das Distributivgesetz benutzt sowie die folgende Äquivalenz:

Ai | !Ai = 1 und X & 1 = X

Also, um die normale Form zu vereinfachen, suchen wir die Paare von Termen in der Form (X & Ai) und (X & !Ai). Ai Variable kann in dem Fall weggelassen werden.

Ein Spezialfall dieser Regel ist

(X & Ai) | X = X

(X und „etwas Spezielles“) oder X ist wahr, wenn X wahr ist, oder unwahr, wenn X unwahr ist.

Wir nennen es Absorptionsregeln.

Wenn die Minimierung nicht mehr möglich ist, haben wir die Minimale Form.

Realisierung einer Normalform als Schaltung

Eine Disjunktive Normalform kann Schaltungstechnisch realisiert werden.

Wie Brauchen Logische Elemente (Gatter) die UND, ODER und Negation erzeugen.

Eine einfache Möglichkeit logische Elemente zu realisieren sind die spannungsgesteuerten Schalter.

Ein Schalter ist geschlossen, wenn sein Eingangspotential hoch ist, also, dem logischen Eins entspricht.

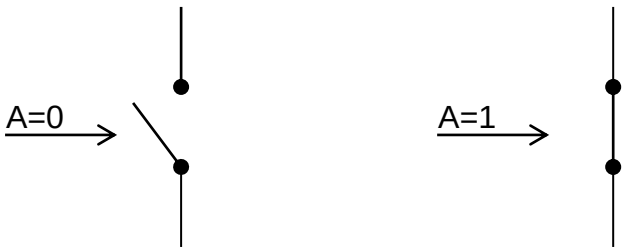

Abbildung 5: Schalter

Wie realisieren wir z.B. die UND Funktion von zwei Variablen A, B, C?

Wir schalten einfach zwei Schalter in Serie (Abbildung 6), sie werden an die Masse angeschlossen. An der anderen Seite haben wir den Ausgang. Zwischen dem Ausgang und einer positiven Versorgungsspan schließen wir einen Widerstand.

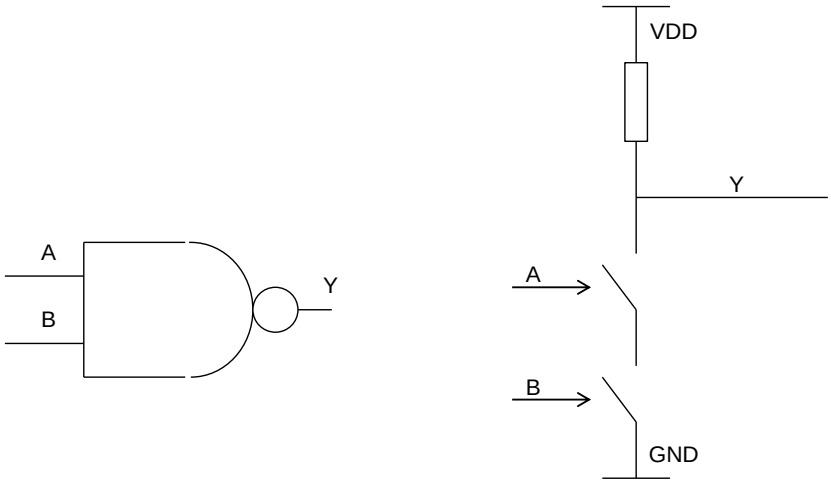

Abbildung 6: NAND Funktion

Nur wenn alle Eingänge Eins sind ist auch der Ausgang null, sonst ist es 1.

So bekommen wir eine Negation der UND Verknüpfung – englisch NAND.

Die Annahme ist, dass die geschlossenen Schalter deutlich niederohmiger sind als der Widerstand. Solche Widerstände zwischen dem Ausgang und VDD nennen wir PullUp Widerstände.

Wie realisieren wir die „echte“ (nicht negierte) UND Verknüpfung?

Dafür brauchen wir einen Inverter (Abbildung 7).

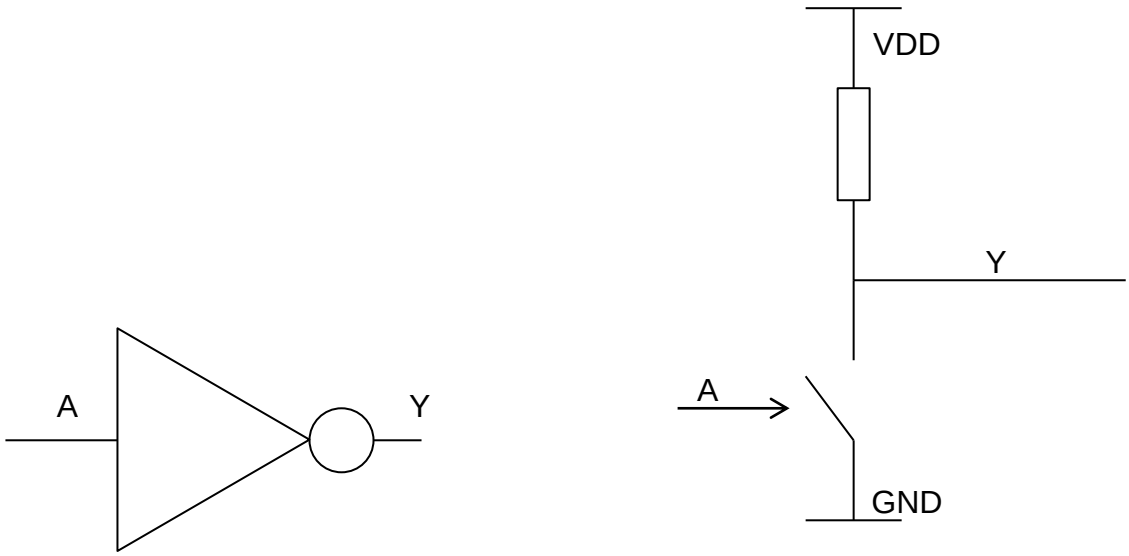

Abbildung 7: Inverter

Wir könnten es mit einem Schalter zwischen GND und dem Ausgang und einem Pull Up - Widerstand realisieren.

Wenn der Eingang A logisch 1 ist (=1), ist der Schalter geschlossen und der Ausgang ist logisch 0.

Wir schalten also einen Inverter an NAND und bilden auf diese Weise UND Gate (Abbildung 8).

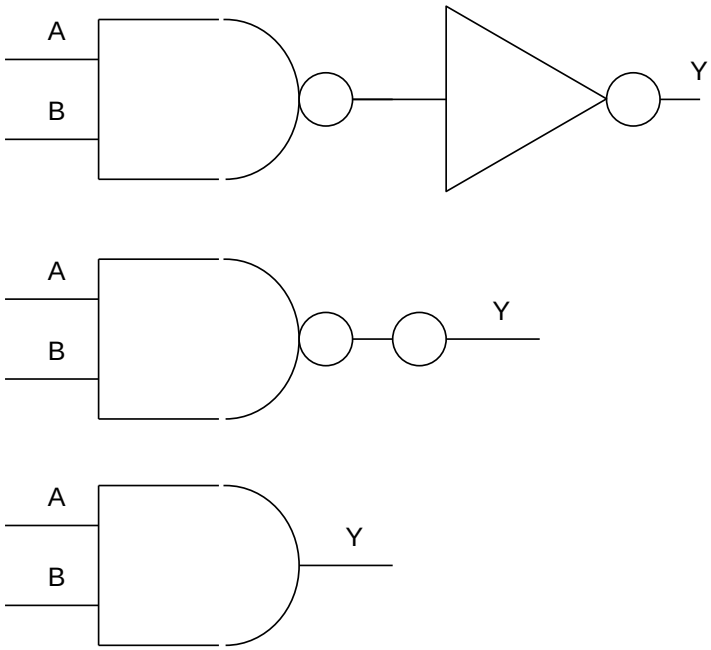

Abbildung 8: UND Funktion

Man kann das Inverter einfach durch einen Inverter-Kreis darstellen. Zwei Kreise „heben sich auf“ (Abbildung 8).

Abbildung 9 zeigt UND mit mehreren Eingängen.

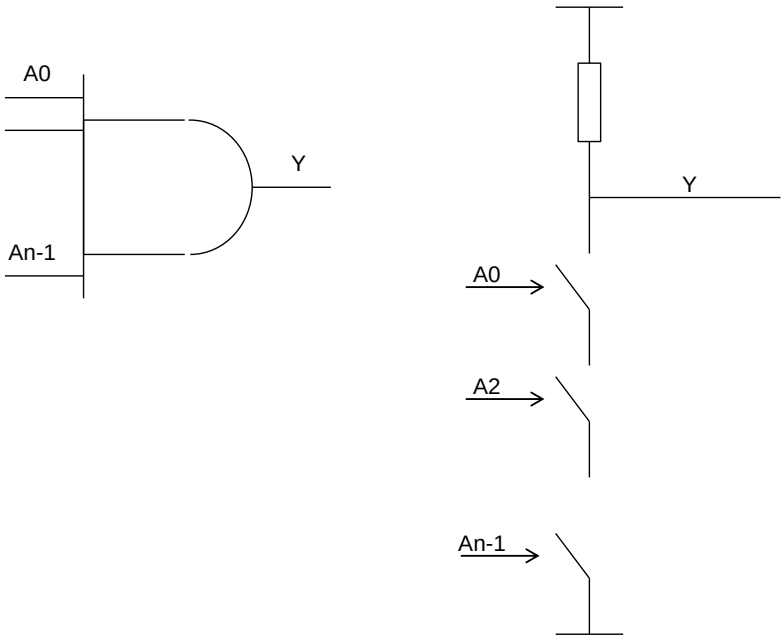

Abbildung 9: UND mit mehreren Eingängen

Auch „ODER“ Funktion kann man mit Schaltern implementieren (Abbildung 10). Hier verwenden wir zwei Schalter in Parallel. Die Schalter sind zwischen GND und dem Ausgang

angeschlossen, wir benutzen einen PullUp Widerstand. Wir bilden zuerst NOR, dann hängen wir einen Inverter an.

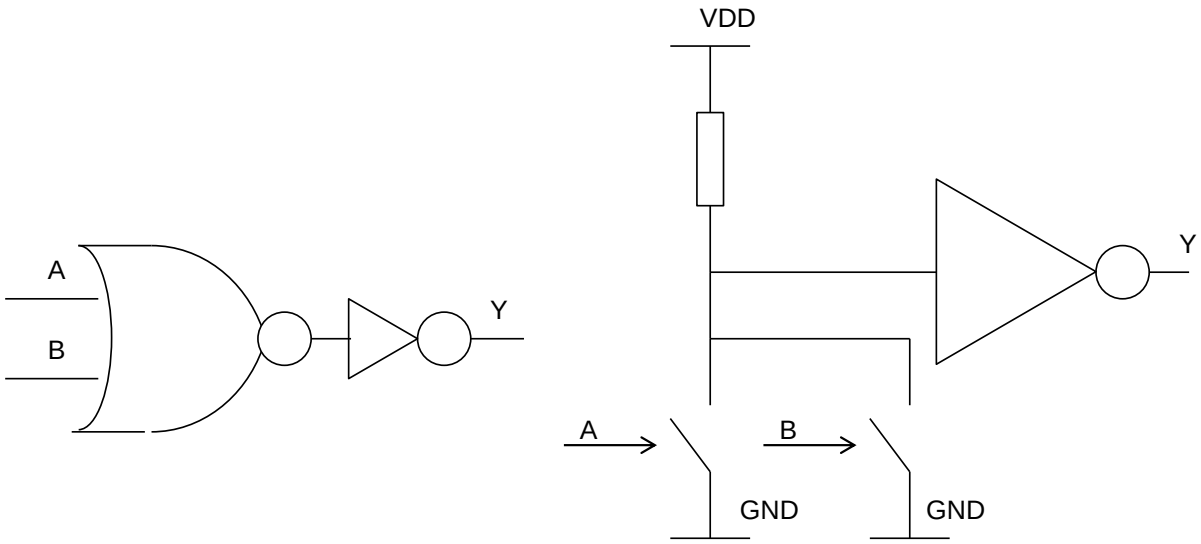

Abbildung 10: ODER Funktion

Komparator: Realisierung als Normalform

Betrachten wir die Funktion:

Ki = !A7 & !A6 & !A5 & !A4 & A3 & A2 & A1 & A0 & !B7 & !B6 & !B5 & !B4 & B3 & B2 & B1 & B0

Mithilfe von Invertern realisieren wir die negierten Eingangsvariablen (Abbildung 11).

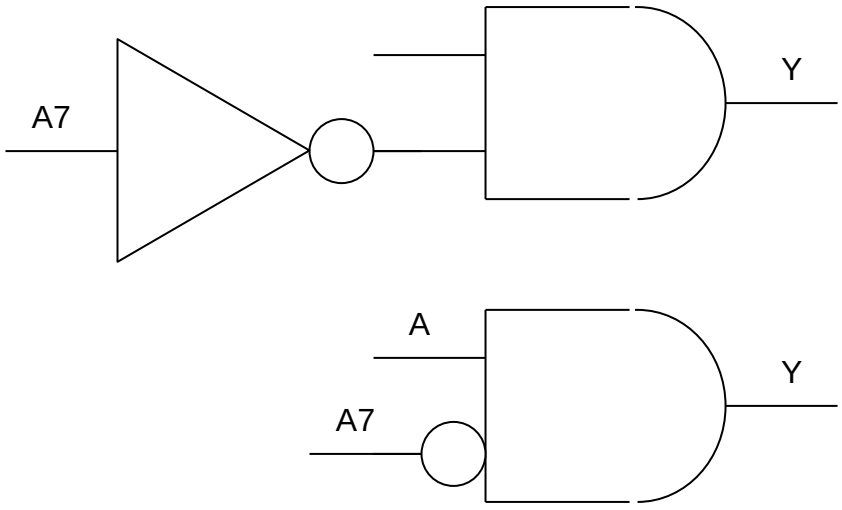

Abbildung 11: Negation von Variablen

Funktion Ki kann wie in Abbildung 12 implementiert werden.

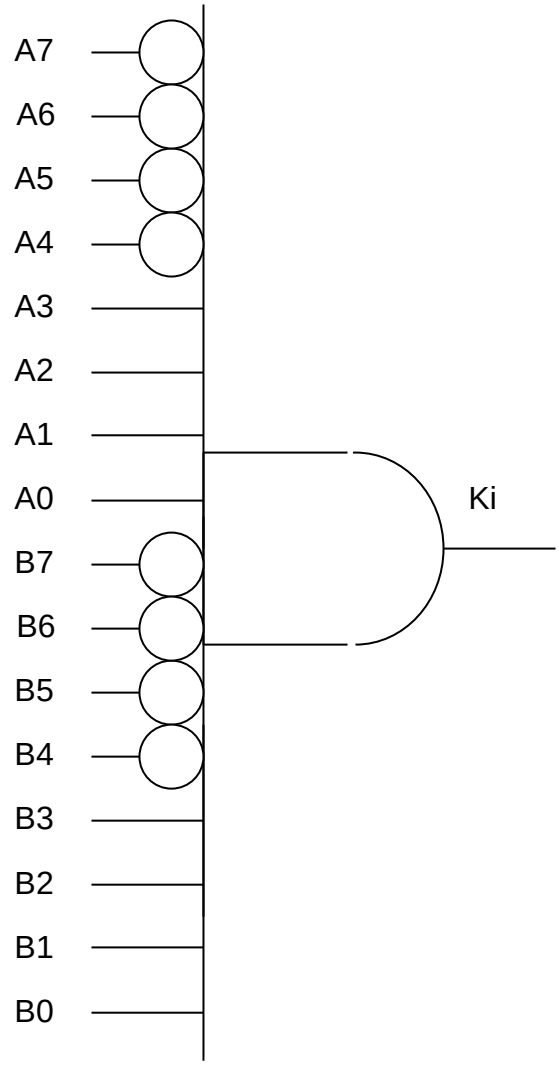

Abbildung 12: Funktion Ki.

Der Komparator benötigt also 256 UND Gatter mit jeweils 16 Eingängen und ein ODER mit 256 Eingängen.

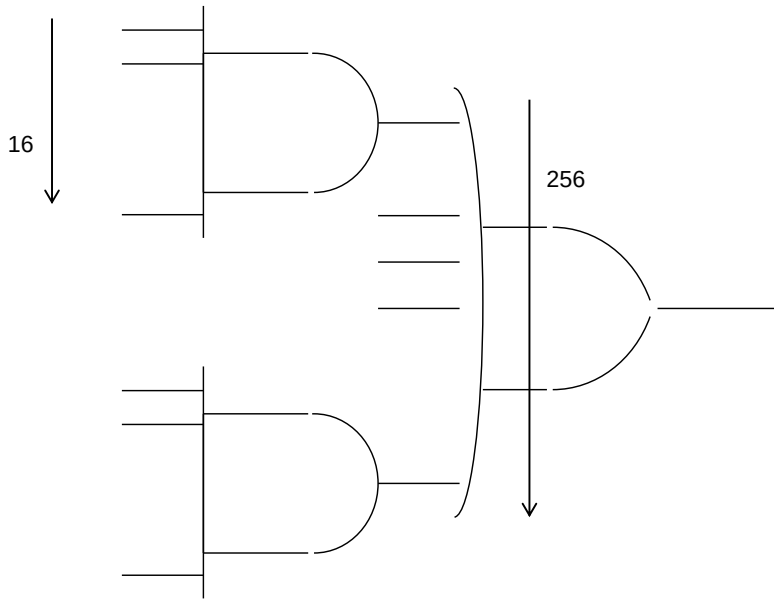

Abbildung 13: Implementierung einer DNF

Bitweise Implementierung von Operationen zwischen binären Zahlen

Eine einfachere logische Schaltung kann entsprechend der gewöhnlichen iterativen Vergleichsmethode gebildet werden – keine Normale Form, mehr Stufen.

Nehmen wir an, wir möchten zwei 8-Bit Zahlen vergleichen.

Wir würden die Zahlen bitweise vergleichen und wenn alle Bits gleich sind ist das Ergebnis auch gleich.

Die Schaltung für Bitweise Vergleich bekommen wir aus der einfachen Tabelle mit vier Zeilen.

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

Abbildung 14: Bitweise Komparator - Äquivalenz

Die entsprechende Normalform ist:

Y = (a & b) | (!a & !b)

Diese Funktion nennt man Äquivalenz. Die Inverse Funktion von Äquivalenz ist EXOR – Exklusiv ODER (Abbildung 15).

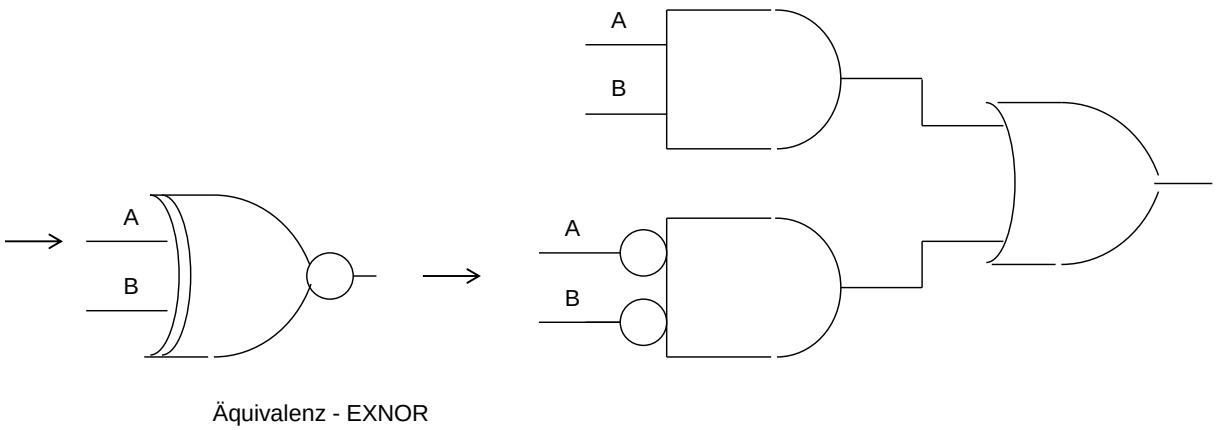

Abbildung 15: Äquivalenz - Implementierung

Ein 8-Bit Komparator basiert auf Bitvergleich ist in Abbildung 14 dargestellt.

Wir brauchen hier nur 8 × 2 UND Gatter und 8 ODER Gatter mit zwei Eingängen und ein UND mit 8 Eingängen.

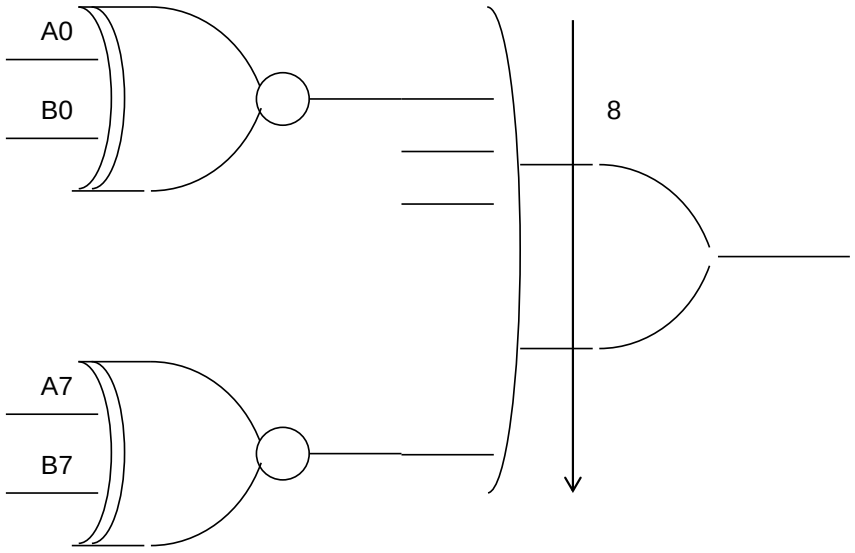

Abbildung 16: Bitweise Komparator

Ein weiteres wichtiges Beispiel ist ein 8-Bit Addierer.

Auch hier bietet sich an, gewöhnlichen Algorithmus für die Addition von mehrstelligen Zahlen Schaltungstechnisch zu implementieren (Abbildung 17).

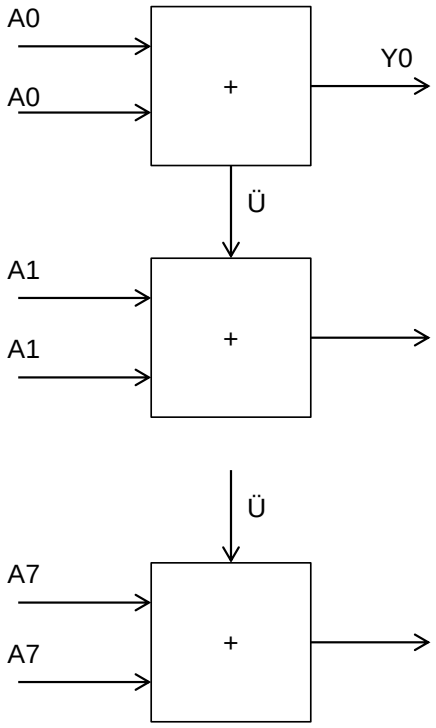

Abbildung 17: Addierer

Es ergibt sich folgende Tabelle für die Addition von zwei Bits a und b, wobei c der Übertrag aus der Addition des vorherigen Bits ist.

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

Abbildung 18: Eine Einheit des Addierers

Die Gleichung für diese Addition ist:

Y = !c & (a exor b) | c & (a == b)

Wichtig ist auch der Übertrag

Cout = !c (a & b) | c & (a | b)

Kombinatorische und Sequenzschaltungen

Die logischen Schaltungen, die die oben genannten Funktionen implementieren nennt man kombinatorische Logik. Der Ausgang der Schaltung ist definiert, wenn man die Eingänge kennt.

In den digitalen Schaltungen verwendet man auch die Schaltungen mit Speicherelementen, mit denen man, zum Beispiel, zyklische Operationen durchführen kann, Zustandsautomaten baut oder Programme realisiert und durchführt.

Solche Schaltungen nennt man sequenziell, deren Ausgang hängt nicht nur von momentanen Eingangswerten, sondern auch von der Vorgeschichte des Systems.

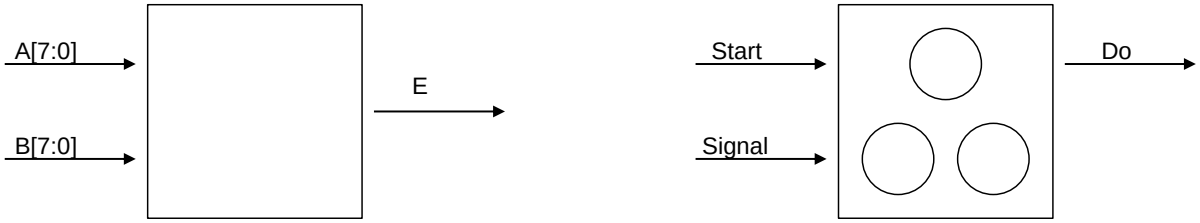

Abbildung 19: Links kombinatorische Schaltung, rechts sequenzielle Schaltung

Nehmen wir als Beispiel eine Uhr – Timer.

Der Eingang ist die eingestellte Zeit – eine achtstellige binäre Zahl, und ein Start Knopf. Der Ausgang ist ein Signal, das auf die abgelaufene Zeit aufmerksam macht (Abbildung 20).

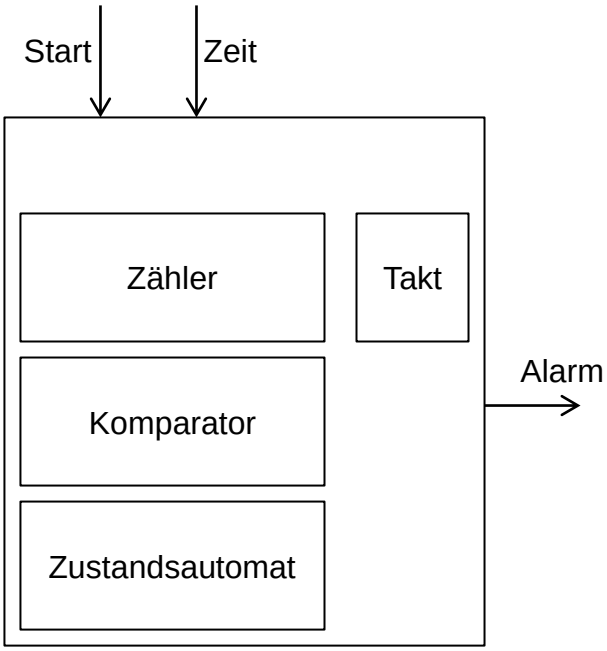

Abbildung 20: Timer

Solche Systeme brauchen zunächst ein internes Taktsignal, also einen Oszillator.

Wir brauchen, weiterhin, einen Zähler. Den Zähler können wir mithilfe vom Addierer aufbauen. Wir brauchen zusätzlich Komparator und Zustandsmaschine.

Implementierung des Zählers

Das Ergebnis wird an A - Eingang des Addieres rückgekoppelt, am B-Eingang haben wir die feste Zahl „1“ (). In einer Programmiersprache würden wir folgendes schreiben:

A = A + 1

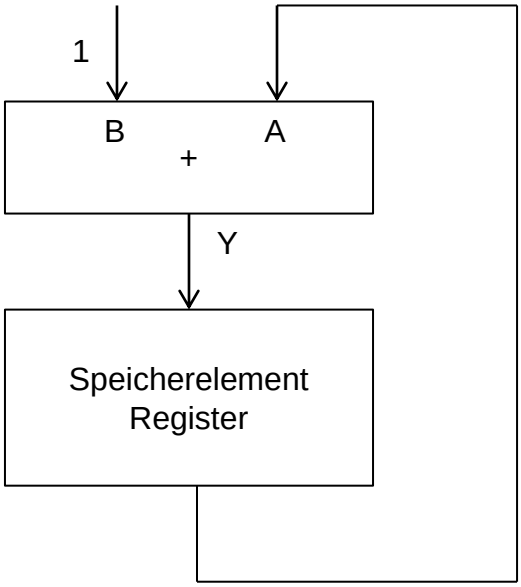

Abbildung 21: Zähler

Die Frage ist nun wann und wie oft wird die Operation durchgeführt. Als Speicherelement wird üblicherweise ein Register benutzt. Dieses Register wird aus Speicherzellen aufgebaut – so genannten Flipflops.

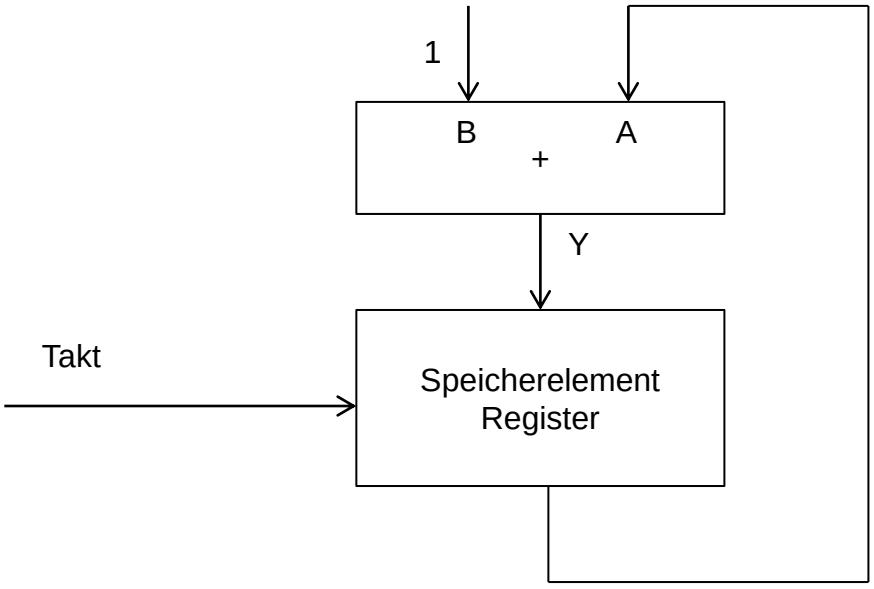

Abbildung 22: Takt

Die Flipflops haben einen Eingang, Ausgang, einen Takteingang und oft ein Reset Signal (Abbildung 23).

Das Taktsignal hat eine periodische Form wie in der Abbildung 23 dargestellt. 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, etwa ~ n × 100ps. So aufgebaut funktioniert die Schaltung wie ein Zähler. Auf jede steigente Taktflanke erhöht sich der Zustand des Zählers.

Wichtig ist hier das folgende:

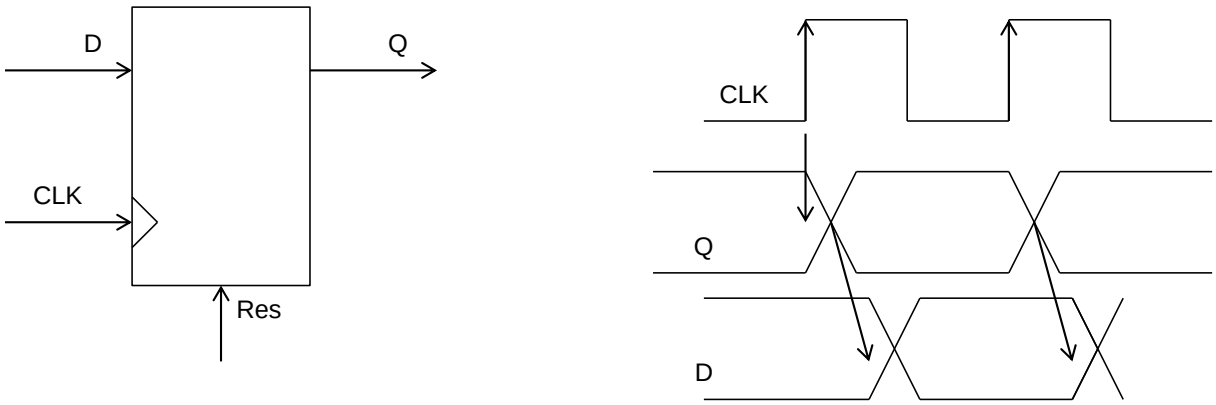

Abbildung 23: Flipflop

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 des Addierer-Ausgangs wird aber erst auf die nächste Taktflanke in Register gespeichert (Abbildung 24). Folgendes ist wichtig: die Änderung des Registerausgangs, verursacht durch Taktflanke i (Qi) führt zu einer Signalwelle durch die kombinatorische Logik. Diese soll vor der nächsten Taktflanke i+1 die Eingänge der nächsten Registerstufe erreichen als Zustand Qi+1 erreichen. Auf Taktflanke i+1 wird der Zustand Qi+1 gespeichert. Bei uns ist die nächste Registerstufe dasselbe Register.

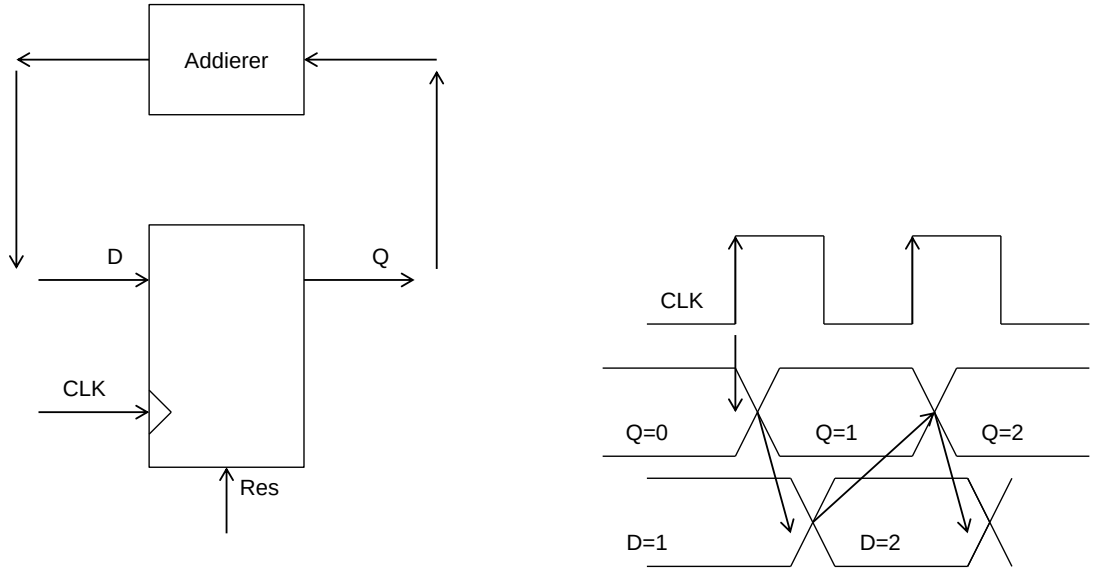

Abbildung 24: Funktionsweise des Addierers

In einer Hardware-Programmiersprache schreibt man:

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 (Abbildung 25). Wenn der Schalter geschlossen ist, verbinden wir den Eingang mit einem Kondensator. Der Kondensator wird auf das Eingangspotential aufgeladen. Erinnern wir uns, dass die Kondensatoren Spannungen behalten, wenn kein Strom aus ihnen fließt. Wenn der Schalter geöffnet wird, behält der Kondensator das Potential. Das logische Niveau wird auf diese Weise gespeichert.

Auf solche einem Prinzip funktionieren die DRAM Zellen.

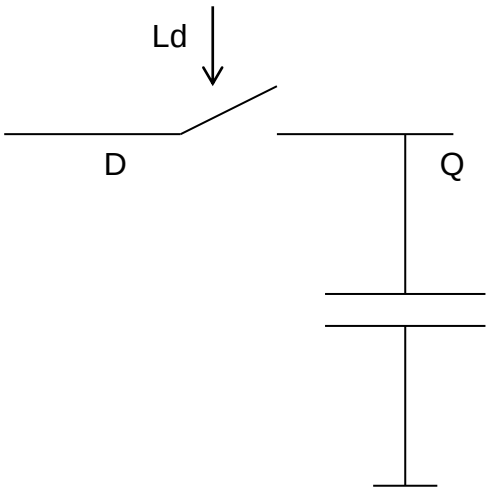

Abbildung 25: Einfache Speicherzelle - DRAM

Wenn wir solche Speicherzellen für den Zähler verwenden würden, gebe es ein Problem (Abbildung 26):

Nach der steigenden Taktflanke wird der Eingang gespeichert - das ist in Ordnung. Das Flipflop aus einem Kondensator würde aber jede weitere Änderung am Eingang ebenfalls speichern, bzw. das anfangs gespeicherte Wert überschreiben, solange Taktsignal eins ist. Das wollen wir nicht. Der Eingangswert soll nur auf die Flanke gespeichert werden, und der gespeicherte Zustand soll sich bis zur nächsten Talkflanke nicht ändern, auch wenn sich der Eingang ändert.

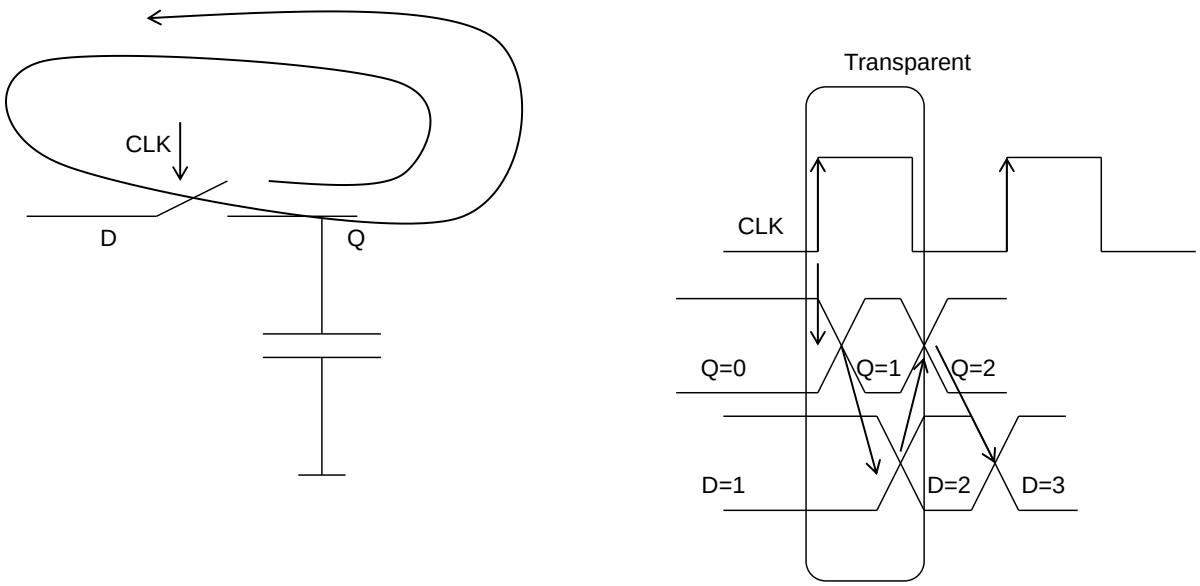

Abbildung 26: Signalverlauf im Fall einer einfachen Speicherzelle

Wie realisieren wir dann eine Schaltung deren Zustand sich nur auf die Flanke ändert?

Eine Möglichkeit ist zwei DRAM Zellen hintereinander zu schalten (Abbildung 27). Die erste Zelle wird an negiertes Taktsignal und die zweite an Originaltakt angeschlossen.

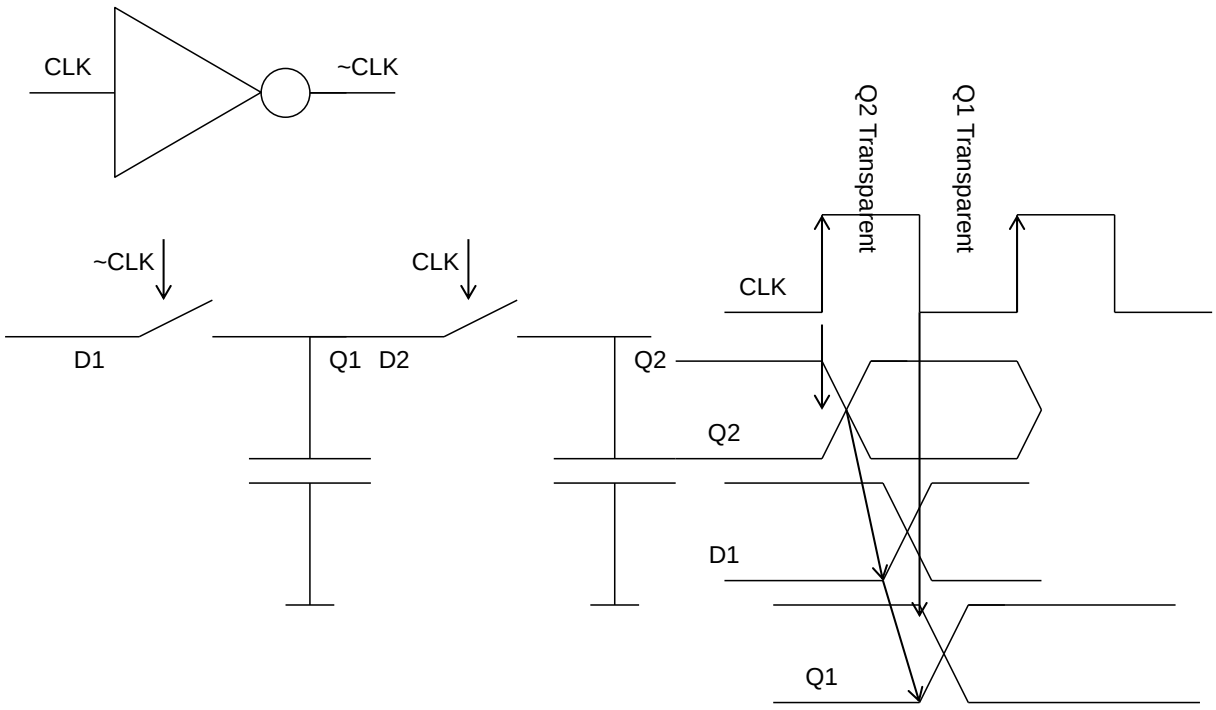

Abbildung 27: Realisierung eines Flipflops

Im Moment unmittelbar vor der steigenden Taktflanke ist die erste DRAM Zelle transparent, was heißt ihr Eingang (D1) ist mit dem Ausgang (Q1) kurzgeschossen. Die zweite Zelle ist vom Eingang D2=Q1=D1 getrennt und hält den alten Zustand Qi.

Nach der Taktflanke wird D1 in der ersten Zelle auf ihrem Kondensator gespeichert. Gleichzeitig geht die zweite Zelle in den transparenten Zustand so dass D1 auch an Q2 Hauptausgang sichtbar wird. Da der Eingang der ersten Zelle D1 jetzt vom Ausgang Q1 getrennt ist, haben weitere Änderungen am Haupteigang keinen Einfluss auf Q2.

Das gilt auch wenn das Taktsignal wieder auf null kommt. Dann wird zwar D1 mit Q1 kurzgeschlossen, gleichzeitig wird aber auch Q1=D2 von Q2 getrennt. Der zweite Kondensator hält den Zustand.

Beachten wir, dass die Schaltung nur dann richtig funktioniert, wenn die zwei Schalter genau in Gegenphase sind. Jede Verzögerung könnte zu Fehlern führen. Z.B. es könnte passieren, dass die zweite Zelle auf die fallende Taktflanke überschrieben wird. Das Design von Flipflops soll vorsichtig gemacht werden.

Digitales Design

Wir haben gesehen wie man den Zähler, seine kombinatorische Logik und Register mithilfe von Schaltern, Kondensatoren und Widerständen realisieren könnte.

In Wirklichkeit ist die Implementierung ein Bisschen anders, aber nach dem gleichen Prinzip wie die eben beschrieben.

Unser Timer braucht noch einige Elemente vor allen eine Zustandsmaschine, die, nachdem das Startsignal erzeugt wird, den Zähler resetet und startet. Wenn der Zählstand den vorgegebenen Wert erreicht, wird der Zähler von der Zustandsmaschine stoppt und ein Alarmsignal erzeugt. Diese Zustandsmaschine ist eine Art Hardwareprogramm und die eigentliche Intelligenz des Systems. Zustandsmaschinen sind, ähnlich wie Zähler, sequenzielle Schaltungen und werden auf die gleiche Weise aufgebaut. Wir werden uns damit in späteren Vorlesungen befassen.

Das Design von digitalen Grundkomponenten wie Flipflops oder Gattern wird ein Thema dieses Kurses sein, obwohl es mehr im Gebiet analoge Elektronik liegt. Solche digitalen Komponenten werden heute ausschließlich auf den Chips als ICs implementiert.

Wenn man heute über Digitaldesign spricht, meint man hauptsächlich den Entwurf von digitalen Schaltungen auf einem FPGA oder einem ASIC. Die Schaltung wird dabei automatisch aus dem HDL Code in mehreren Schritten generiert. Im ersten Schritt erzeugt die Software eine Netzliste mit vorgegebenen Komponenten, welche dem Code entspricht. Im zweiten Schritt wird ein Belegungsplan für die Komponenten erzeigt und die physikalischen Verbindungen auf dem ASIC, FPGA werden vorgesehen.

Im letzten Schritt werden die Verbindungen und die Schaltungen realisiert, indem z.B. das RAM des FPGA programmiert wird oder ein ASIC hergestellt wird.

Obwohl die digitalen Schaltungen aus dem Code automatisch generiert werden, ist es nützlich, die Realisierung zu kennen. Nur wenn wir ungefähr wissen welche Netzliste aus einem Code generiert wird, können wir die Performanz der Schaltung, wie die maximale Taktfrequenz, abschätzen und den Code entsprechend anpassen. Ein Beispiel. Wir hätten einen 10GHz 8 Bit

Zähler. In dem Fall wäre es sinnvoll z.B. den Zähler auf zwei 4-Bit Zähler zu zerlegen, und den langsamen „MSB-Zähler“ an das 16× langsamere Taktsignal anzuschließen.