

Design digitaler Schaltkreise

- Themen
- Design digitaler Grundkomponenten
- Logik „Gates“ („Gatter“), Register, Speicherelemente, RAM, etc.
- Übung: Digitales Chipdesign, Schaltungsentwurf aus dem HDL Code
- ILIAS: 23683 Design digitaler Schaltkreise (Vorlesung (V)), SS 2017
- Übung: Anfang nächste Woche
- Email Adressen:
- ivan.peric@kit.edu
- richard.leys@kit.edu

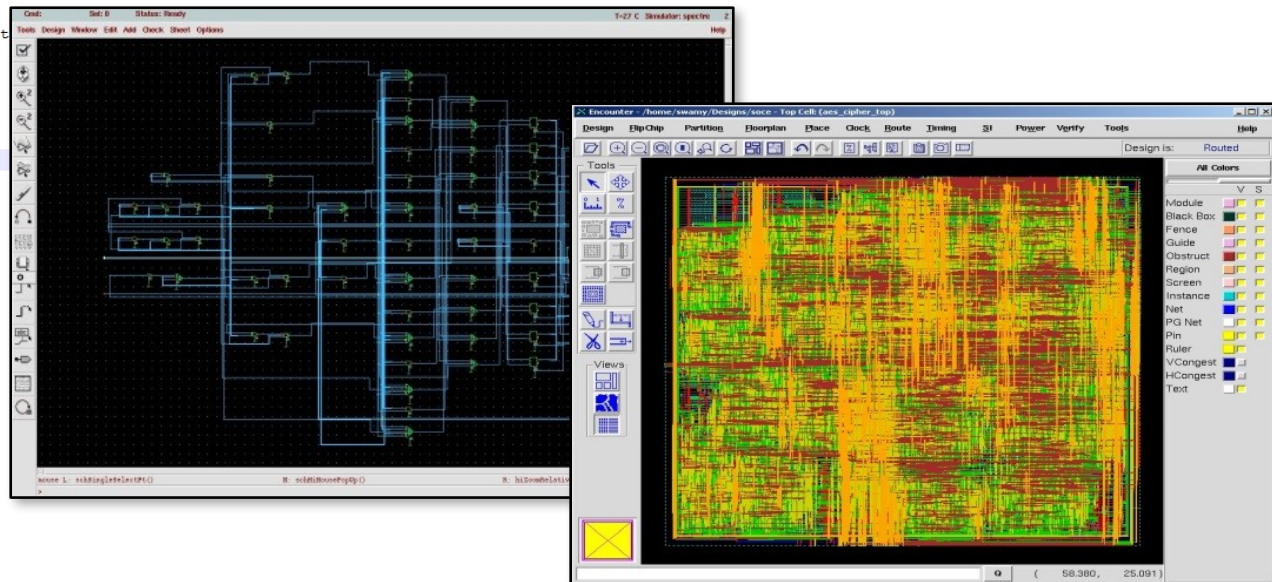
- Vorlesung:
- Einführung (heute)
- Transistoren, Entwicklungsgeschichte, digitale Revolution
- Einfache CMOS Gatter
- Komplexere CMOS Digitalzellen – von NAND bis Flipflop
- Setup und Hold Zeit, Optimierung von Digitalschaltungen, Zustandsautomaten
- Addierer, Zähler, noch komplexere Schaltungen
- Speicherzellen, SRAM, DRAM, EPROM, Flash,...
- Serielle Datenübertragung, Takterzeugung, PLL, Oszillator
- ...
- Übung:
- Digitales IC-Design
- Hardware-Programmiersprache Verilog
- Simulation von Verilog – Modulen (Digitalschaltungen)
- Synthese und Layout - Erzeugung

- Digitaldesign mit Softwarepaket „Cadence“
- HDL Code, Simulation -> Synthese -> RTL Code (Schaltplan mit Logikzellen und Registern), Simulation (automatisch) -> Place and Route -> Layout, Simulation (automatisch) -> DRC, LVS

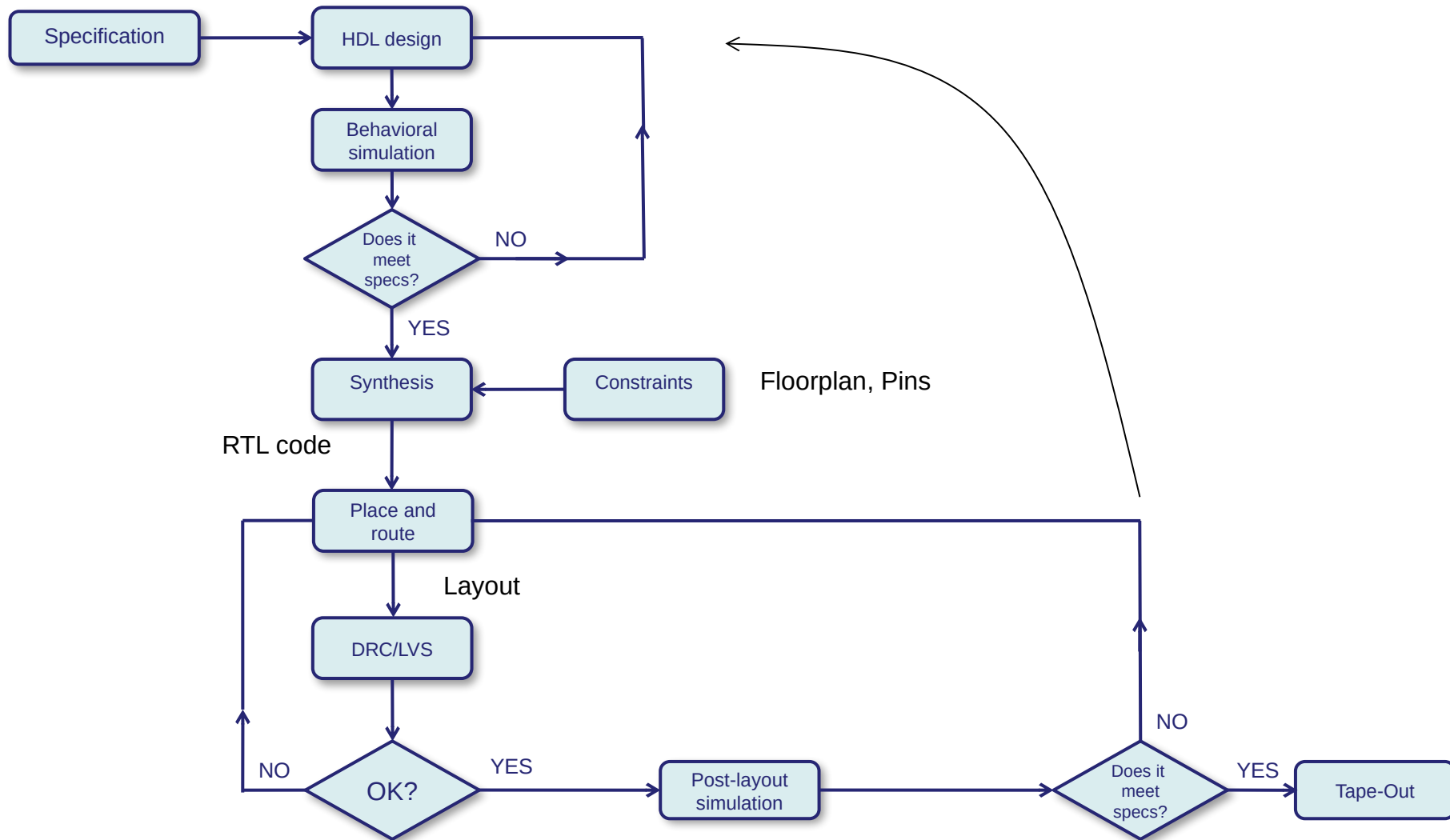
```

19 localparam IDCODE      = 4'b0010;
20 localparam INIT_IR_VALUE = 4'b0101;
21
22
23 reg [IR_LENGTH-1:0] jtag_ir;          // Instruction register.
24
25 always @ (posedge tck or posedge trst)
26 begin
27     if (trst) begin
28         jtag_ir[IR_LENGTH-1:0] <= {IR_LENGTH{1'b0}}; // Why not set to IDCODE ??? TODO
29         latched_jtag_ir <= IDCODE; // IDCODE selected after reset
30     end else if (state_capture_ir)
31         jtag_ir <= INIT_IR_VALUE; // This value is fixed for easier fault detection ???
32     moved
33     else if (state_shift_ir)
34         jtag_ir[IR_LENGTH-1:0] <= {
35     else if (state_update_ir)
36         latched_jtag_ir <= jtag_ir;
37     end
38
39 always @ (negedge tck)
40 begin
41     serout <= jtag_ir[0];
42 end

```

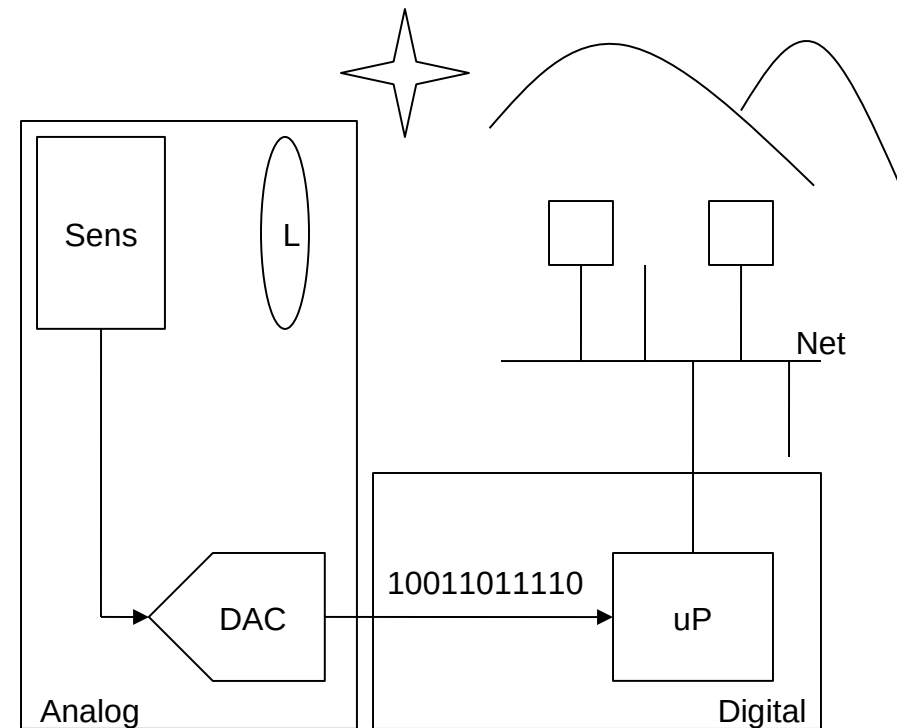
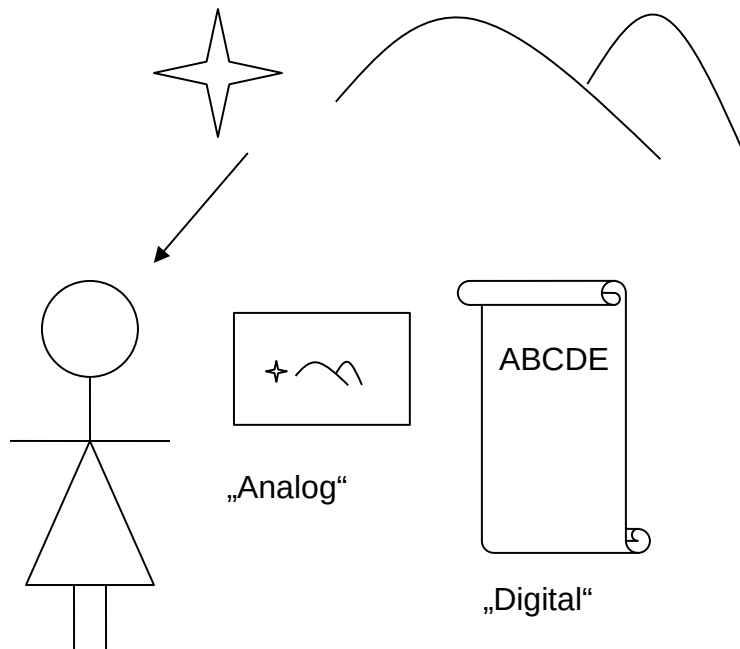


- Digitaldesign mit Softwarepaket „Cadence“



- Digital vs. Analog
- Warum Binäre Zahlen?
- 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: Addierer
- Kombinatorische und Sequenzschaltungen
- Beispiel: Timer
- Komponenten: Zähler, Speicherzelle
- DRAM -> FF

- Wir leben in einer analogen Welt - alle 'Messgrößen' unserer Umwelt (Licht, Töne, Temperatur, elektrische Spannungen, Druck etc..) sind analog
- Aber: Information war und ist oft „digital“ gespeichert und bearbeitet



- Vorteile von digitaler Datenverarbeitung
- Schaltungen und Geräte sind kleiner und flexibler
- Mathematische Operationen mit Bits sind einfacher zu realisieren als mit analogen Größen
- Rauschen beeinträchtigt ein digitales Signal weniger – falls Rauschen unter einer Schwelle liegt -> keine Störung
- Aber: digitales Signal ist an sich verlustbehaftet (Quantisierungs- und Abtastfehler)
- Auch: digitale Signalbearbeitung ist nie in Echtzeit
- CMOS Transistoren (geeignet für DS) sind deutlich kleiner als bipolare Transistoren (geeignet für AS)
- Aber: Alle digitalen Komponenten sind im Grunde analog
- Schnittstellen zwischen uns und digitalen Geräten sind analog
- -> Ingenieure im Bereich AE sind auch gefragt



- 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
- $2^3 = 8$
- $2^4 = 16$
- ...
- $2^{10} = 1024$
- $2^{(1N)} \sim 2^N \times 1000 = 2^N \text{ k}$

- Warum werden in digitaler Elektronik binäre Codes benutzt?
- Nehmen wir an wir möchten zwei Zahlen bis zur Größe N kodieren
- Wir brauchen $\ln N / \ln X$ 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 allgemeinen Fall - X^2 Zeilen.
- Elektronik hätte die Größe:
- $\ln N / \ln X * X^2$
- Diese Funktion hat ein Minimum in der Nähe von 2

N = 32

A	A	A	A	A
4	3	2	1	0

B	B	B	B	B
4	3	2	1	0

C	C	C	C	C
4	3	2	1	0

$C = 5 \times 4 = 20$

A	B	C
0	0	
0	1	
1	0	
1	1	

N = 27

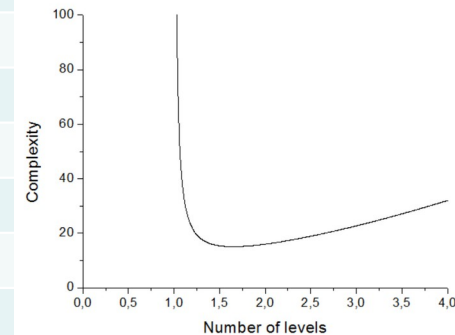
A	A	A
2	1	0

B	B	B
2	1	0

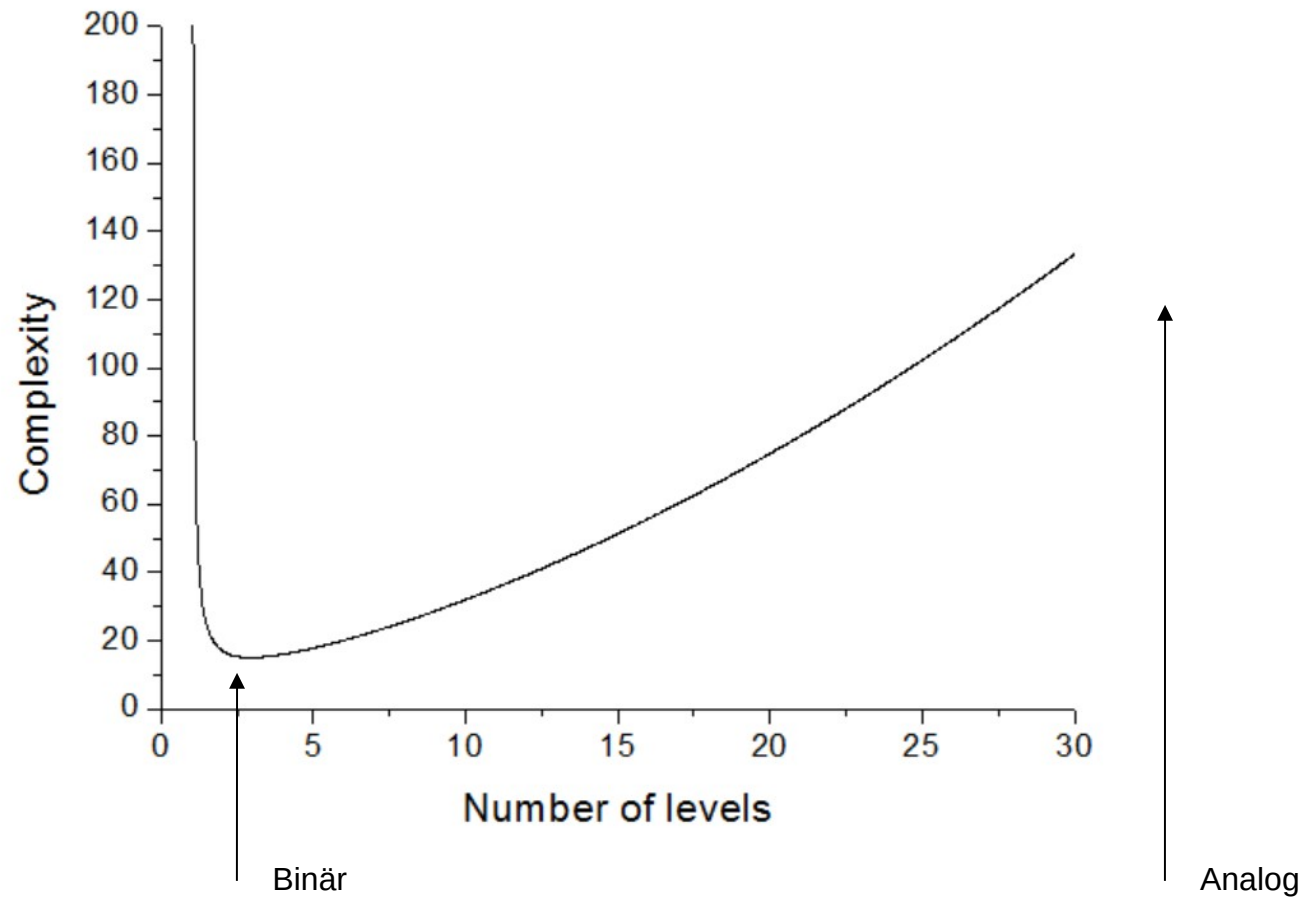
C	C	C
2	1	0

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

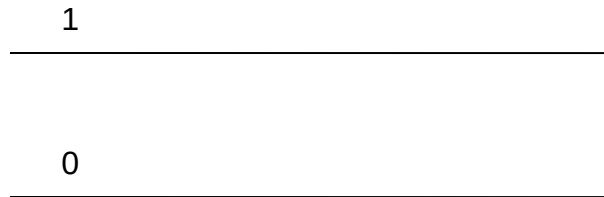
$C = 3 \times 9 = 27$



• ...



- In digitalen Schaltungen werden die Ziffern 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
- $256 \times 256 = 2^{16}$ Kombinationen $\rightarrow 2^{16}$ 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 $\wedge$ oder $*$ oder $\&$
- Konjunktion entspricht der Umgangssprache: Ergebnis ist wahr ($=1$) wenn X_0 und X_2 und ... X_{n-1} wahr sind.
- Wir definieren auch ODER Verknüpfung (Disjunktion) mit dem Ergebnis null nur wenn alle Variablen null sind.
- Das Zeichen für Disjunktion ist $\vee$ oder $+$ oder $|$
- Es entspricht dem Satz: Ergebnis ist wahr ($=1$) wenn X_1 oder ... X_n wahr sind.
- Boolesche und gewöhnliche Logik sind „dual“

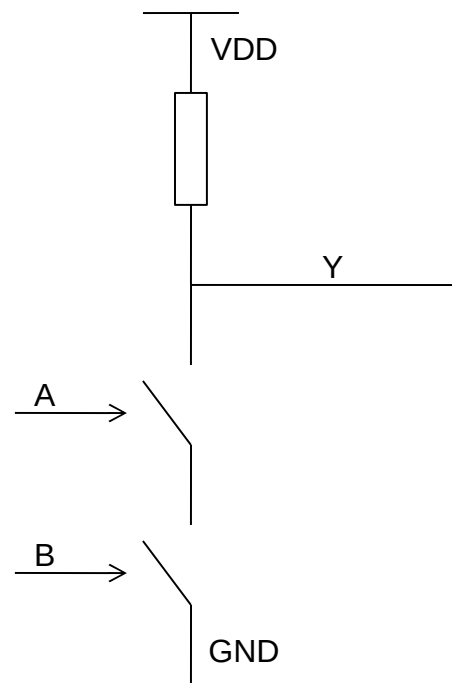
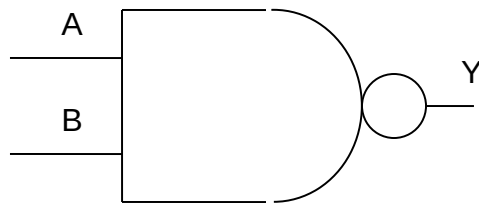
- Die Tabelle für Vergleich zwei 8-Bit zahlen können wir wie folgend als Disjunktive Normalform darstellen:
- Wir suchen alle Zeilen mit dem Ergebnis 1 – es gibt sie 256. Für jede solche Zeile bilden wir eine UND Verknüpfung, die nur für die Variablen-Werte aus dieser Zeile eins ergibt:
- ZB 0000_1111 0000_1111
- $K_i = !A_7 \ \& \ !A_6 \ \& \ !A_5 \ \& \ !A_4 \ \& \ A_3 \ \& \ A_2 \ \& \ A_1 \ \& \ A_0 \ \& \ !B_7 \ \& \ !B_6 \ \& \ !B_5 \ \& \ !B_4 \ \& \ B_3 \ \& \ B_2 \ \& \ B_1 \ \& \ B_0$...
- Zeichen ! bedeutet Negation – wir verwenden es überall dort wo die Variable 0 ist. Die Gesamttabelle ist dann ODER Verknüpfung von allen K_i Funktionen.
- $F = K_0 \ | \ ... \ | \ K_{255}$
- (Alternative Zeichen für die Negation sind $\sim$, $-$ 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

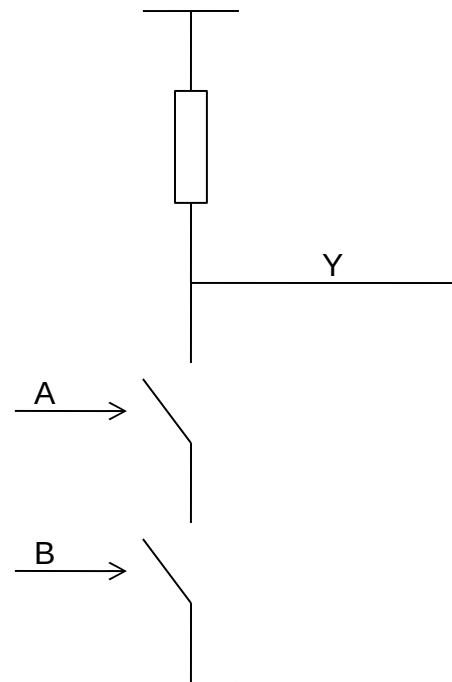
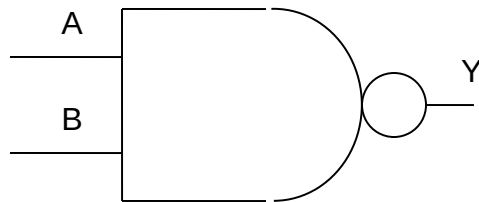
- Die Normalform kann vereinfacht werden (Absorptionsregeln)
- $(X \& A_i) \mid (X \& !A_i)$ können wie folgend vereinfacht werden:
- $X \& (A_i \mid !A_i) = X \& 1 = X$
- Distributivgesetz, Äquivalenz $A_i \mid !A_i = 1$ und $X \& 1 = X$
- Bei den Paaren von Termen $(X \& A_i)$ und $(X \& !A_i)$, A_i kann weggelassen werden
- Eine weitere Variante solcher Regel ist
- $(X \& A_i) \mid X = X \rightarrow (X \& A_i) \mid (X \& (A_i \mid !A_i)) = (X \& A_i) \mid (X \& !A_i)$
- $(X \text{ UND etwas}) \text{ ODER } X \text{ ist wahr/falsch}$
- 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

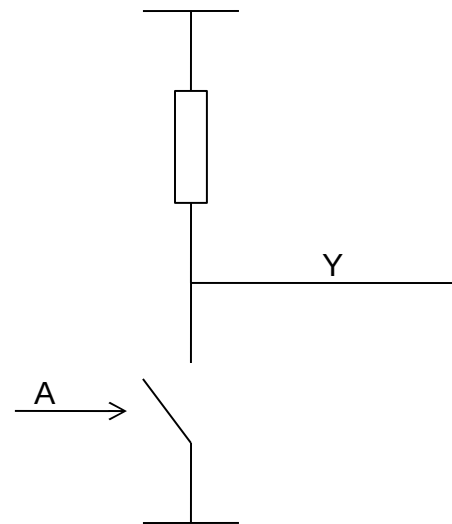
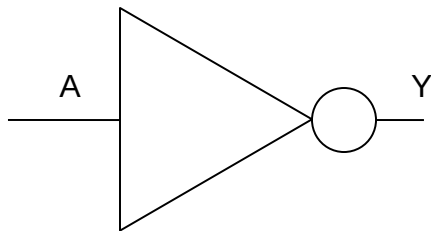
- UND Funktion von Variablen A, B
- Jede Variable ist ein Draht – Kabel („wire“), sein Potential Wert 0 oder 1.
- Zwei Schalter in Serie, an Masse angeschlossen.
- Widerstand zwischen dem Ausgang und der positiven Versorgungsspannung VDD
- Wir definieren das Potential um VDD als logische 1 und das Potential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



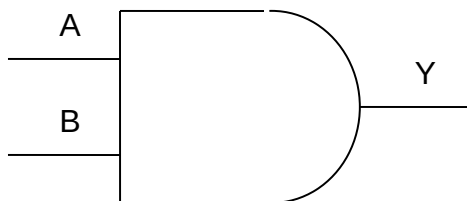
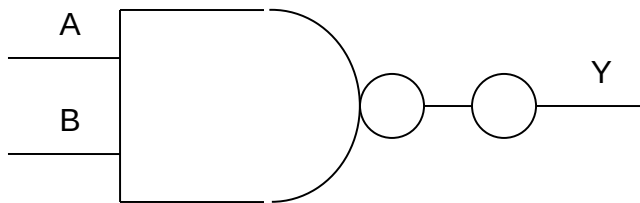
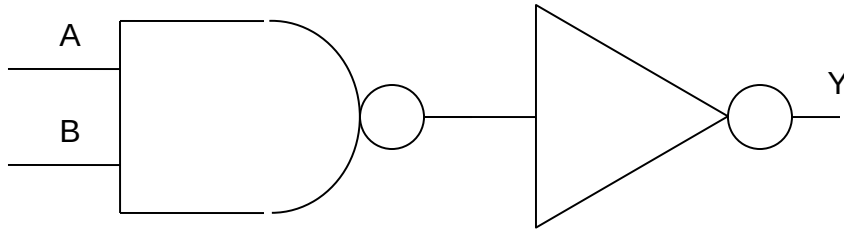
- Annahme: die geschlossenen Schalter sind deutlich niederohmiger als der Widerstand.
- PullUp Widerstand



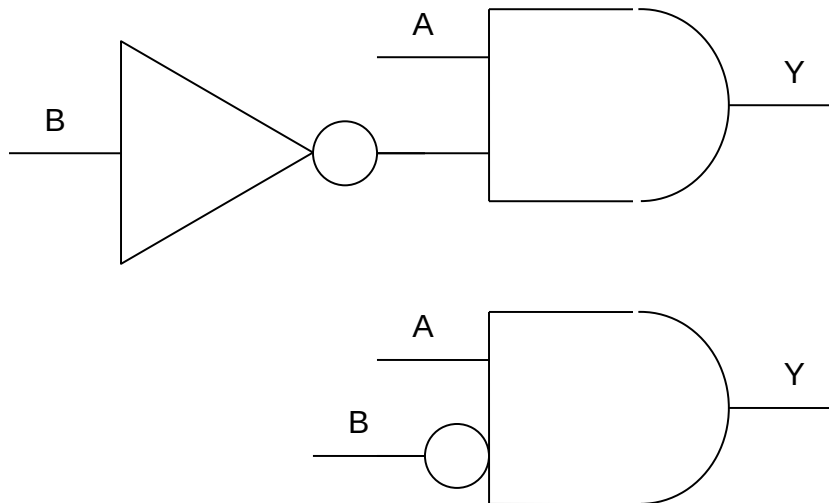
- Inverter



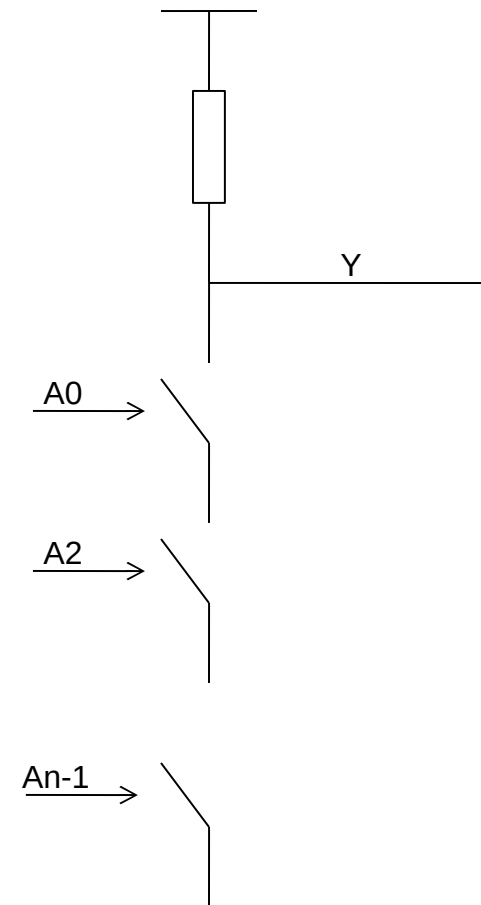
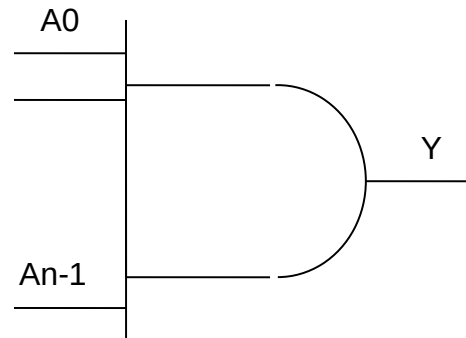
- Inverter und NAND -> UND/AND



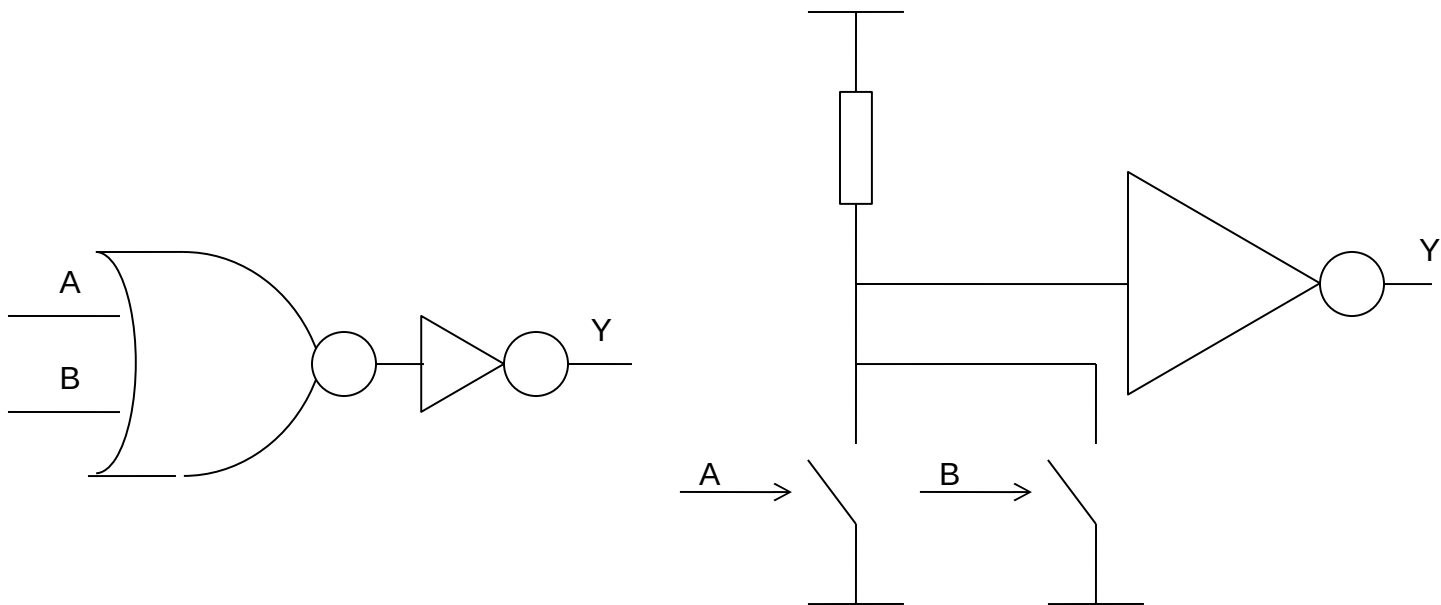
- $K_i = !A7 \ \& \ !A6 \ \& \ !A5 \ \& \ !A4 \ \& \ A3 \ \& \ A2 \ \& \ A2 \ \& \ A0 \ \& \ !B0 \ \dots$
- Auf ähnliche Weise können wir die Terme mit negierten Eingangsvariablen realisieren



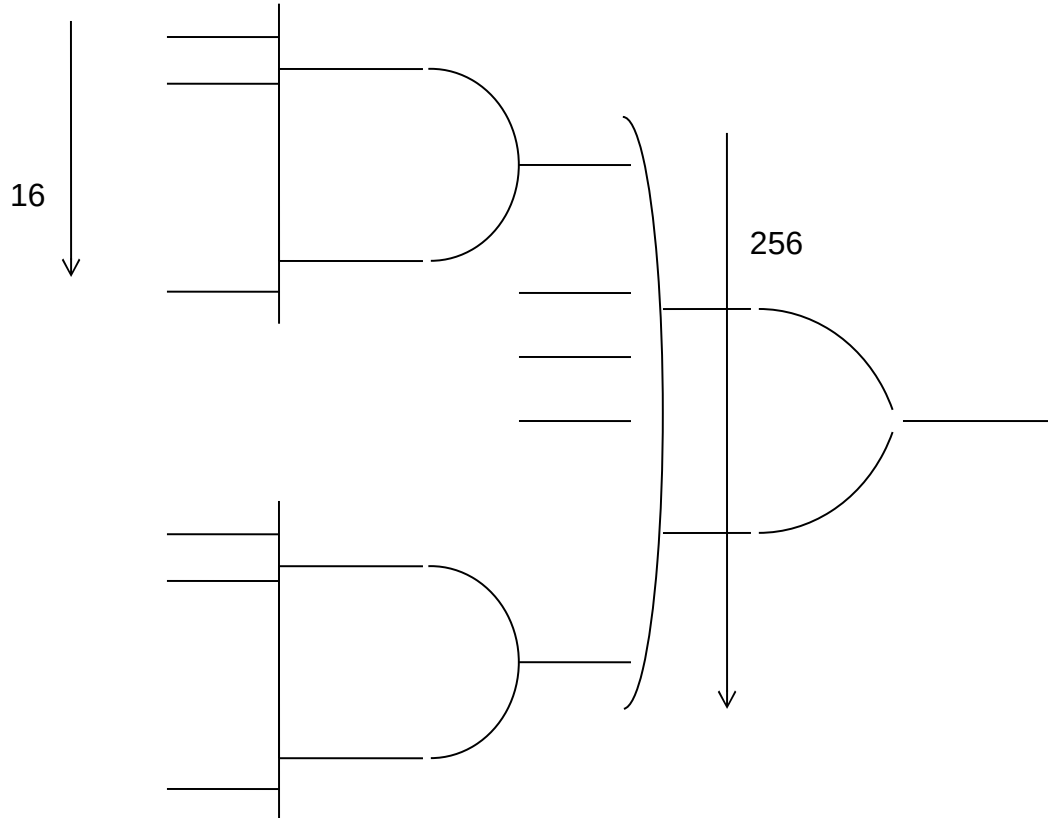
- UND mit mehreren Eingängen



- 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.

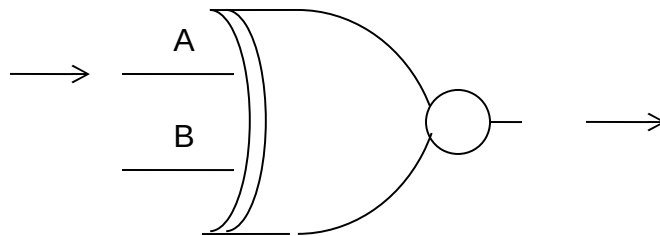


- Unser Komparator braucht also 256 UND Gatter mit jeweils 16 Eingänge und ein ODER mit 256 Eingängen – kompliziert!
- Eine weitere Vereinfachung der Normalform nach den Absorptionsregeln ist in diesem Fall nicht möglich.
- Man kann aber die Terme umzugruppieren – Distributivregeln. Schwierig ohne Rechner/Programm

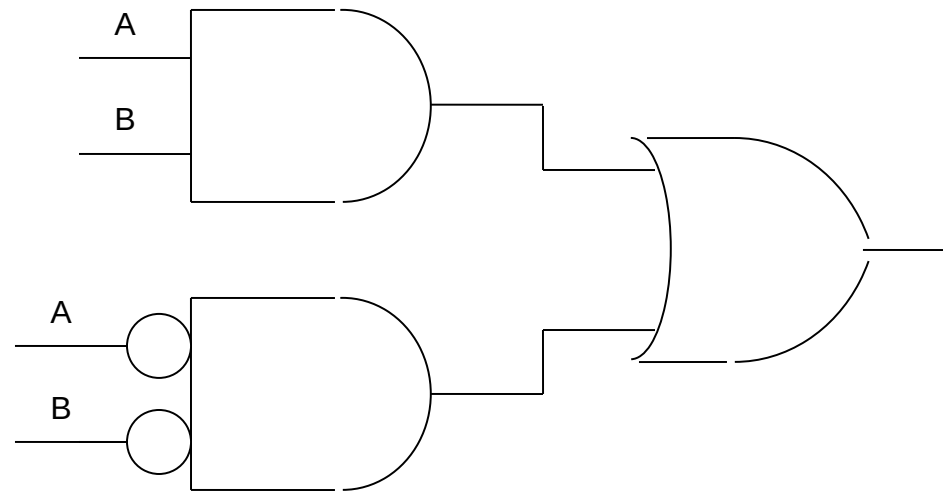


- Einfachere Logik kann entsprechend der gewöhnlichen iterativen Vergleichsmethode gebildet werden – keine Normale Form, mehr Stufen
- Bitweise Vergleich -> wir vergleichen alle Bits einzeln, wenn alle gleich sind -> sind auch die Zahlen gleich
- Äquivalenz: Normalform $Y = (a \& b) \mid (!a \& !b)$

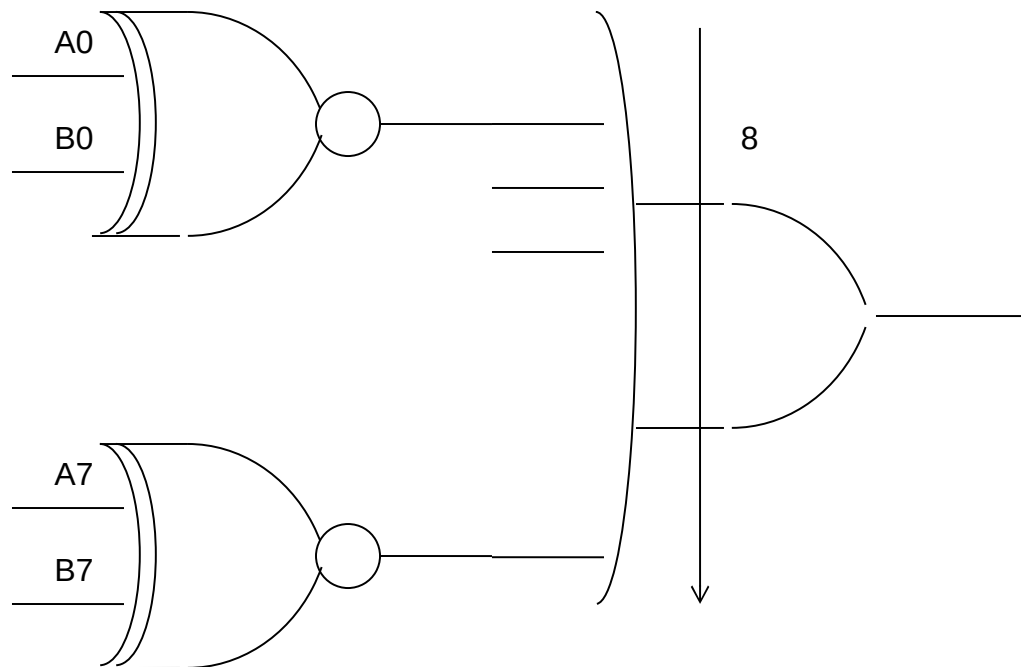
a	b	y
0	0	1
0	1	0
1	0	0
1	1	1



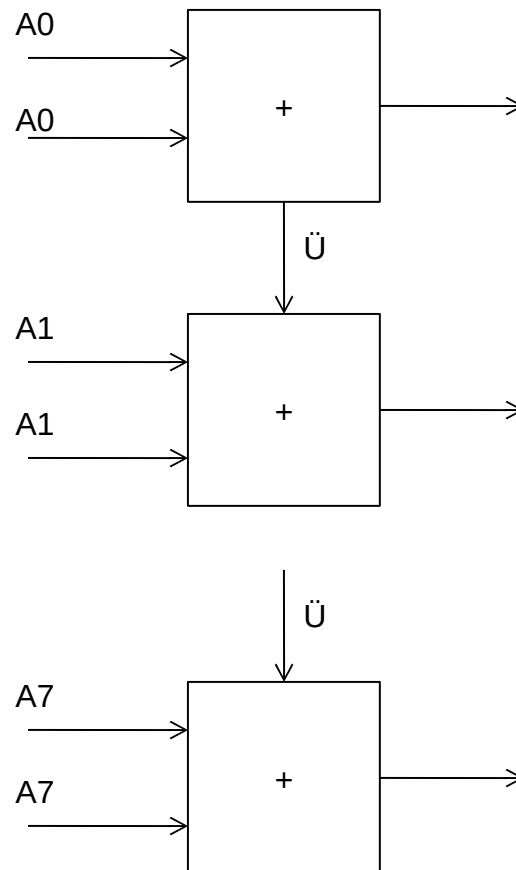
Äquivalenz - EXNOR



- 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
- $\text{Summe} = !c \ \& \ (a \text{ exor } b) \mid c \ \& \ (a == b)$

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
- $C_{out} = !c \& (a \& b) \mid c \& (a \mid b)$
- Hier $\&$ vor $\mid$ - für Programmiersprachen soll immer geprüft werden

C=0

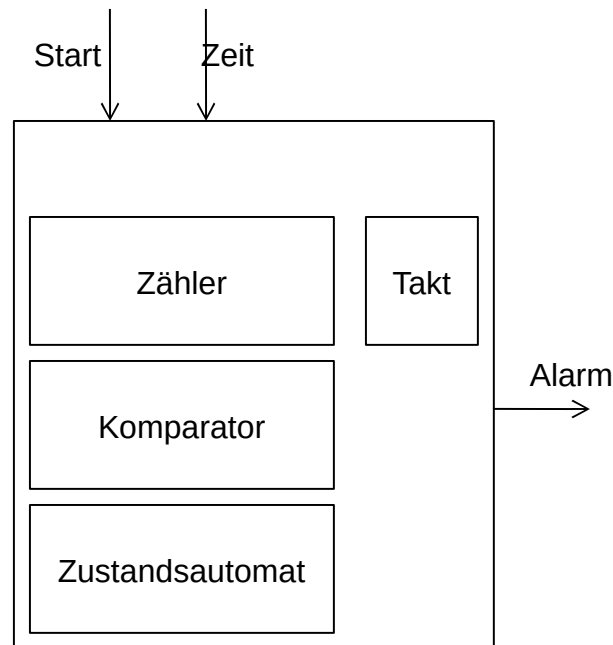
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

C=1

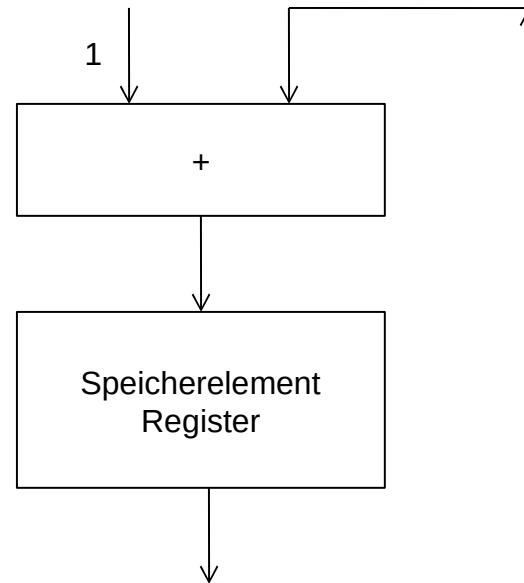
a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

- 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.

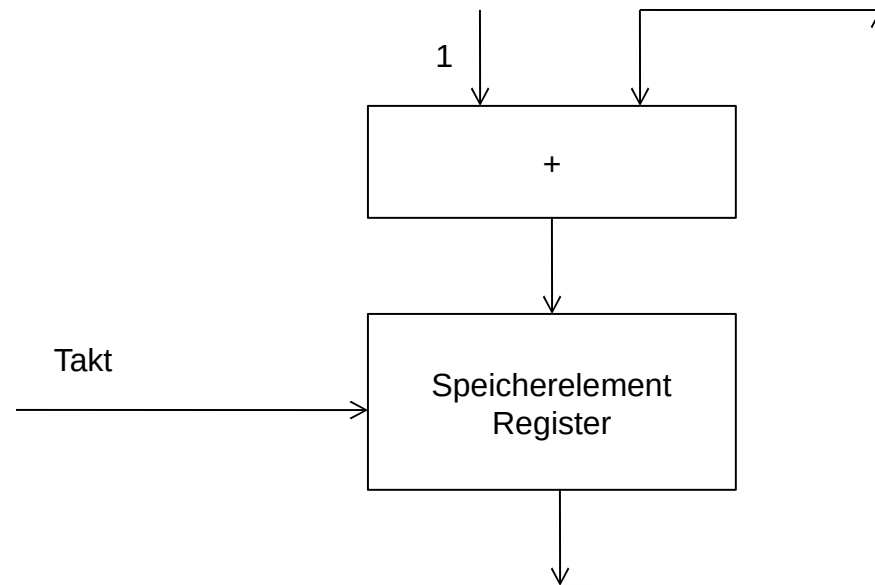
- Beispiel - Timer
- Der Eingang: die eingestellte Zeit – z.B. achtstellige binäre Zahl, und ein Start-Knopf. Der Ausgang ist ein Alarm-Signal
- Solche Systeme brauchen 1) ein internes Taktsignal, also einen Oszillator.
- 2) einen Zähler
- 3) Komparator und Zustandsmaschine



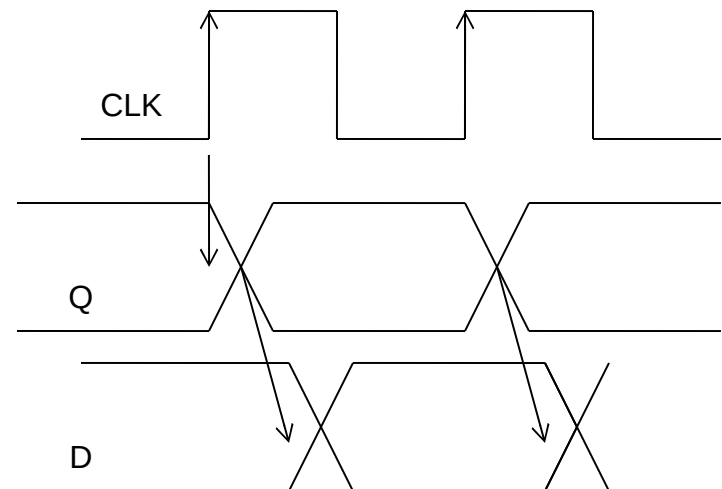
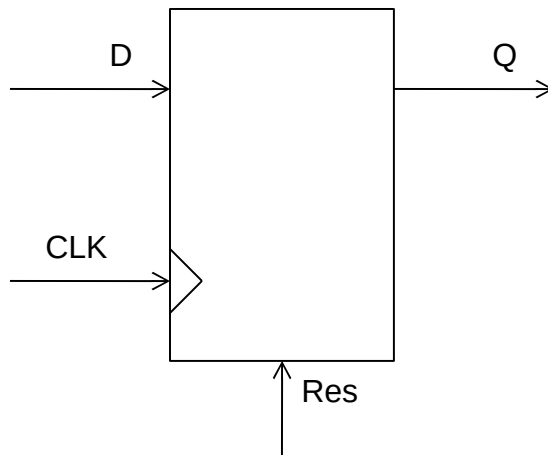
- Zähler: Addierer + Speicherelement wo das Ergebnis gespeichert wird.



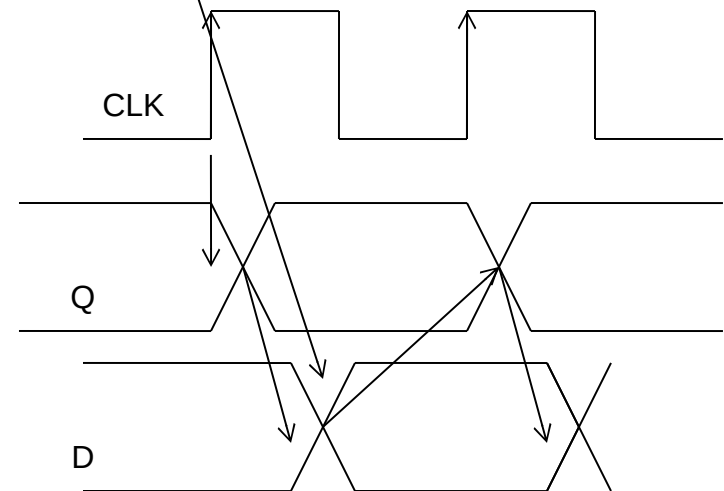
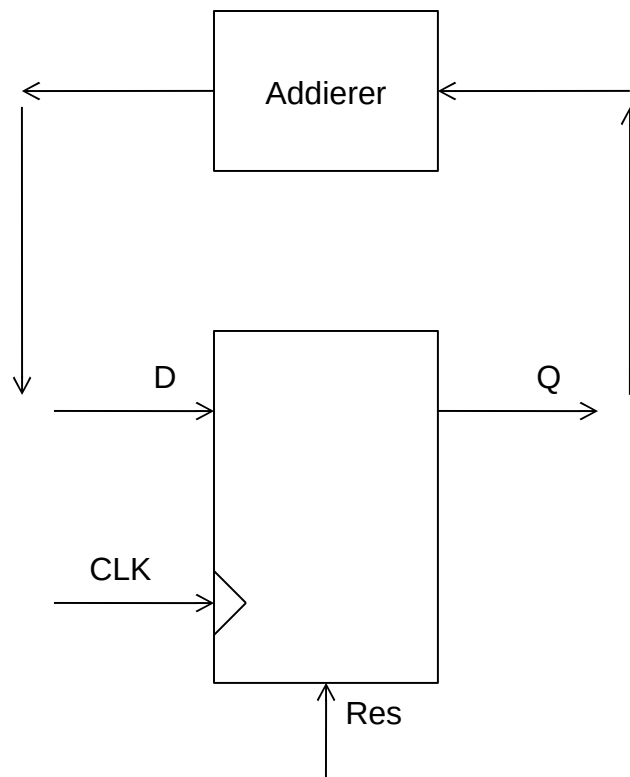
- Zähler: Addierer + Speicherelement wo das Ergebnis gespeichert wird.
- Wie wird der Zähler angesteuert?
- Register aus FlipFlops (Speicherzellen)



- 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 $\sim n \times 100\text{ps}$. => Auf jede steigende Taktflanke erhöht sich der Zustand des Zählers.



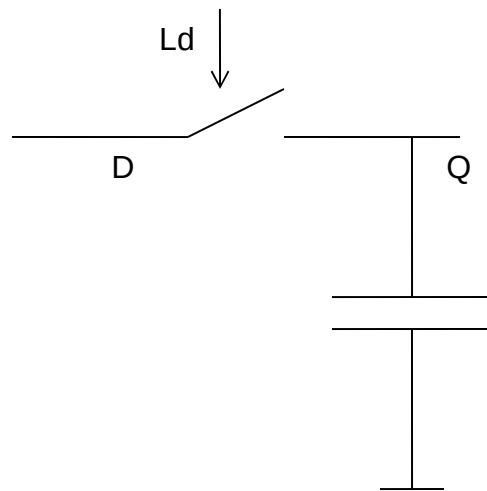
- Der Eingang des Registers ändert sich auch einige 100ps nach der Taktflanke, da der Addierer den neuen Eingangswert A bekommt und seinen Ausgang anpasst. Diese Änderung der Addierer-Ausgangs wird aber erst auf die nächste Taktflanke in Register gespeichert



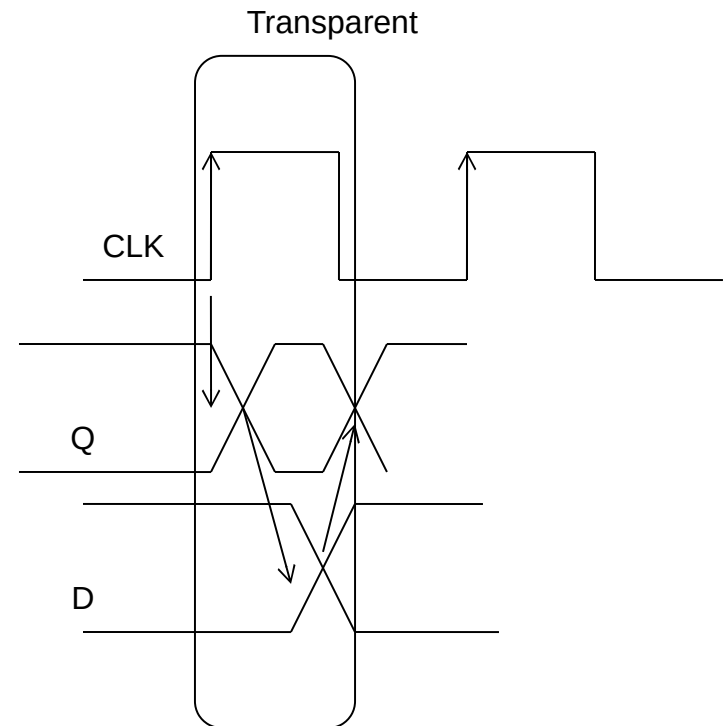
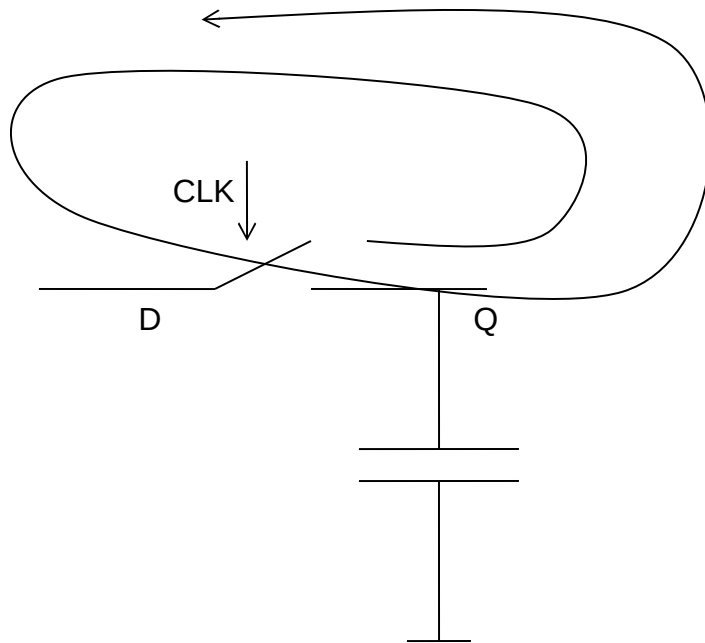
- Hardware-Programmiersprache:

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

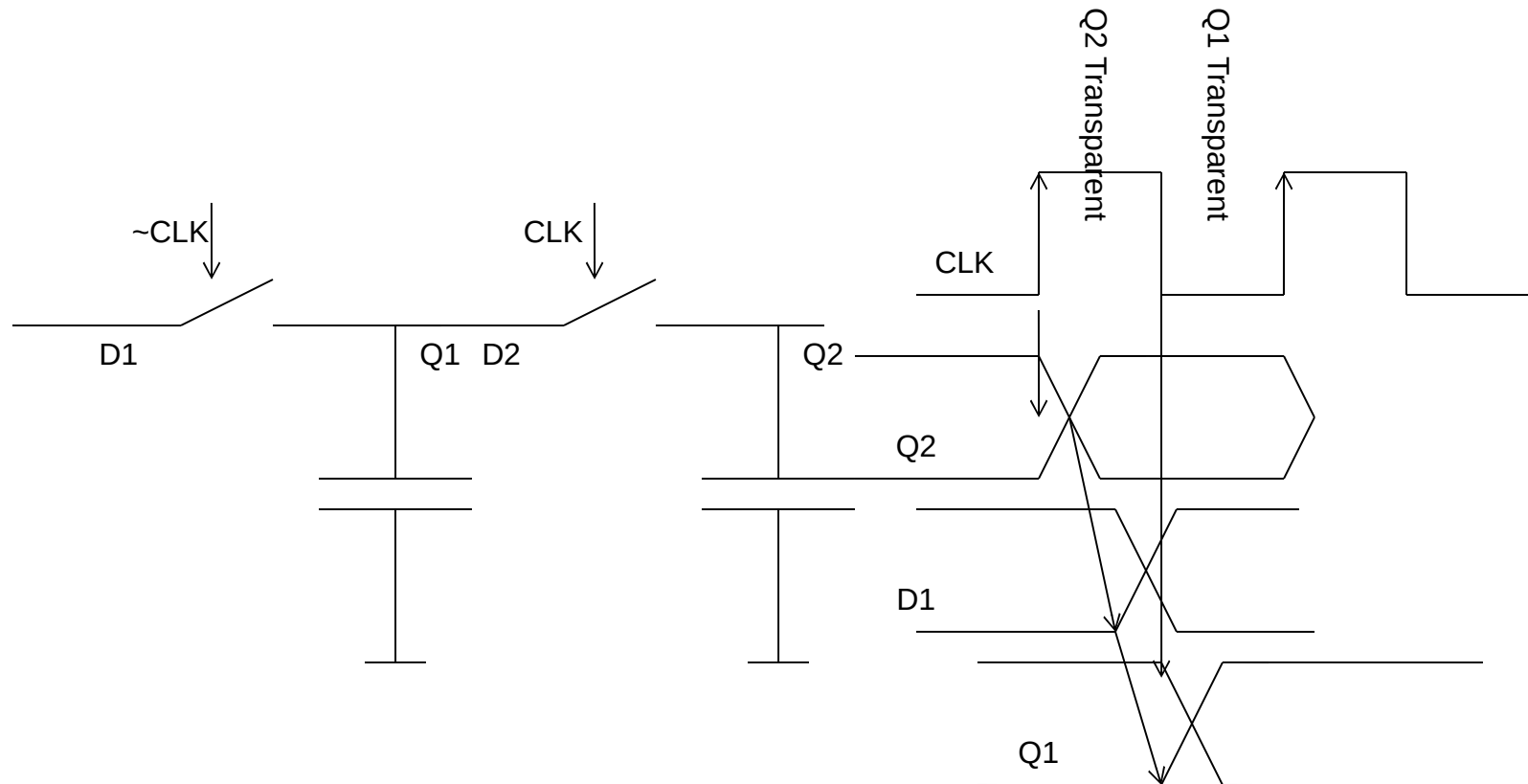
- Wie realisieren wir ein Flipflop?
- Am einfachsten stellen wir uns eine Speicherzelle wie einen getakteten Kondensator vor. Wenn der Schalter geschlossen ist, verbinden wir den Eingang mit einem Kondensator. Der Kondensator wird auf das Eingangspotential aufgeladen.
- Wenn der Schalter geöffnet wird, 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 Taktflanke nicht ändern, auch wenn sich der Eingang ändert



- Lösung
- Zwei DRAM Zellen hintereinander zu schalten



- Themen
- Design von digitalen Grundkomponenten wie Flip Flops oder Gates/RAM - analoge Elektronik.
- Entwurf von digitalen Schaltungen auf einem FPGA/ASIC.
- HDL Code -> Schaltung
- 1) Netzliste
- 2) Belegungsplan (Layout) für die Komponenten und die physikalischen Verbindungen