KOMMENTIERTE AUFZEICHNUNG DER VORLESUNG

RECHNERSTRUKTUREN

Aufzeichnung der Vorlesung vom 20.10.2000

Aufbau des 1. Semesters:

a) Zahlendarstellung
b) Einfache logische Schaltungen
c) Bausteine eines Computers entwickeln (und per Computer simulieren)
d) Einen einfachen Computer entwickeln (etwa auf dem Stand von 1950)

Zahlen

Zahlen können in unterschiedlichen Systemen abgebildet werden. Das einfachste System wäre das System zur Basis 1.. Das bedeutet, man jede Zahl durch Striche kennzeichnet. Dieses System nennt man Unärsystem („Jeder Strich ist eine Zahl“)
Beispiel:

| = 1
|| = 2
||| = 3

|||| = 4
|||||||||| = 10
|||||||||||||||||||| = 20

Soweit so einfach. Man schreibt anbei nun (zumindest in den Vorlesungen von Rechnerstrukturen) zu jeder Zahl der Basis zu dem die Zahl gehört.
Beispiel:

$|||_1 = \mathbf{3}_{10}$

Basis 1 — Basis 10

Also, die Zahlen, wie wir sie immer geschrieben haben, stammen aus dem sogenannten Dezimalsystem. Das Dezimalsystem besteht aus Zahlen von 0,1,2,...,9.
Eine Zahl, wie z.B. 573 kommt in diesem System wie folgt zustande.

…	tausender	hunderter	zehner	Einer
	10^3	10^2	10^1	10^0
		5	7	3

Auch wenn das jetzt ein wenig kompliziert erscheinen mag, kann man jede Zahl, so auch die 573_{10} wie folgt zusammensetzen

$5 * 100 + 7 * 10 + 3 * 1$ das entspricht $5 * 10^2 + 7 * 10^1 + 3 * 10^0$
$= 573_{10}$

Das macht auch klar, warum das Ding Dezimalsystem genannt wird: man multipliziert immer mit $10^{irgendwas}$!

Aber leider wird das jetzt noch etwas komplizierter:
Wie man (vielleicht) weiß, mag der Computer lieber 0en und 1sen. Das System, das also auf 2 Zeichen basiert, nennt man Binärsystem, das System zur Basis 2. (Die Basis eines Systems ist abhängig von den Zeichen, die man zu Verfügung hat. Im Dezimalsystem hatte man 10 {0,1,2,…,9} Zeichen, also ist die Basis 10. Im Unärsystem hatte man 1 Zeichen, jetzt rat mal, welche Basis das Unärsystem hat)

Im Binärsystem sehen die uns bekannten Zahlen so aus

$$110010011010_2 \text{ oder } 101_2 \text{ oder } 111000_2$$

man kann die Zeichen wie folgt umrechen:

$0_2 = 0_{10}$ $\quad 11_2=3_{10}$ $\quad 1100_2=12_{10}$ $\quad 101100_2 = 44_{10}$

Die $_2$ steht hier für Basis 2, die $_{10}$ für Basis 10

Aber wie rechne ich das jetzt um?

Beispiel:

die Zahl 110011_2

Und so weiter	2^6	2^5	2^4	2^3	2^2	2^1	2^0
		1	1	0	0	1	1

Man rechnet das jetzt so aus

$$1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 32_{10} + 16_{10} + 2_{10} + 1_{10} = 51_{10}$$

alles klar?
Zur Sicherheit noch ein Paar Übungen:

Aufgabe No.	2^6	2^5	2^4	2^3	2^2	2^1	2^0
A)		1	0	1	0	0	1
B)			1	0	1	0	0
C)						1	1
D)				1	1	1	1
E)	1	0	0	0	0	0	0

a) $1 * 2^5 + 0* 2^4 + 1* 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 41_{10}$
b) $1 * 2^4 + 0 * 2^3 + 1* 2^2 + 0 * 2^1 + 0 * 2^0 = 20_{10}$
c) $1 * 2^1 + 1 * 2^0 = 3_{10}$
d) $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 15_{10}$
e) $1 * 2^6 + 0* 2^5 + 0* 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0* 2^0 = 64_{10}$

einfacher wirt es, wenn man anstatt $2^{irgendwas}$ durch die richtigen Zahlen ersetzt:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	4	2	1

Damit kann man dann einfacher rechen!

Diesen „Spaß“ kann man auch mit jeder anderen positiven ganzzahligen Basis machen z.B. mit 3 oder 8 oder 16. Man muss hat dann nicht $2^{irgendwas}$, sondern $8^{irgendwas}$ oder $16^{irgendwas}$. Ich mach zu der Basis 3 nur kurz ein Beispiel:

Und so weiter	3^6	3^5	3^4	3^3	3^2	3^1	3^0
			1	0	1	2	2

Fragt man sich spontan, was macht die blaue 2 da? Klar, man hat immer der Basis entsprechend viele Zeichen. Klartext Basis 3 hat 3 Zeichen, nämlich {0,1,2}.
Wie rechen wir das Beispiel
10122_3 in das Dezimalsystem um?
Wie gehabt: $1 * 3^4 + 0 * 3^3 + 1 * 3^2 + 2 * 3^1 + 2 * 3^0 = 98_{10}$

Wäre die Welt doch schön, gäbe es da nicht das Hexadezimalsystem. Das Hexadezimalsystem hat die 16 zur Basis und besteht daher aus 16 Zeichen und zwar {0,1,2,3,…,9,A,B,C,D,E,F} man hätte anstatt A auch 10 schreiben können, aber dann kann man eine Zeichenkette nicht mehr unterscheiden. Eine Zahl im Hexadezimalsystem sieht daher z.B. so aus: $1A5FF_H$, man rechnet das dann so aus:

Und so weiter	16^6	16^5	16^4	16^3	16^2	16^1	16^0
			1	A	5	F	F

$1 * 16^4 + 10 * 16^3 + 5 * 16^2 + 15 * 16^1 + 15 * 16^0 = 108031_{10}$

Das war das A (A=10)

Das waren mal F’s (F=15)

Also $1A5FF_H = 108031_{10}$

Aber warum $_H$? Das ist eine Vereinbarung, das H steht für Hexadezimalsystem, sonst schreibt man die Zahl der Basis.

Haben wir tatsächlich noch mehr gemacht? Ja, leider!
Und zwar, wie man z.B. eine Binärzahl in eine Hexadezimalzahl umwandelt, ohne das Dezimalsystem zu benutzen.
Konkret, was ist $1100100111101100 1_2$ als Hexadezimalzahl?
Nun, im Hexadezimalsystem haben wir 16 Zeichen zur Verfügung, um im Binärsystem eine 16 darzustellen, brauchen wir mindestens 4 Zeichen ($1111_2 = 15_{10}$, da die Null dabei ist haben wir 16 Zeichen, die wir mit 4 Zeichen aus dem Binärsystem unterscheiden können). Und was hilft uns das? Ganz einfach wir nehmen unsere Binärzahl und teilen sie in 4er Paare:

1	1001	0011	1101	1001
1	9	3	D	9

$1100100111101100 1_2 = 193D9_H$ ($=103385_{10}$)

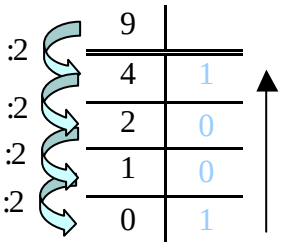
Jetzt müssen wir nur noch jedes 4er paar für sich nehmen und umrechnen. Für 1001 wäre das $1*2^3+0*2^2+0*2^1+1*2^0 = 9$ wir tragen unten also bei dem Hexadezimalfeld die 9 ein. Bei der 1101 kommt eine 13 raus, wie wir wissen entspricht im Hexadezimalsystem die 13 dem D, also kommt ein D in das Feld, und so weiter und so weiter.

Das macht zwar nicht glücklich und bei Bäcker kriegen wir auch keine Brötchen, wenn wir ihm das vorrechnen, aber wir können immerhin sagen, wir können Zahlen vom Binärsystem ins Hexadezimalsystem umrechnen (toll was?).

Hmm, ich nehme mir mal das Recht raus und umschreibe das folgende etwas knapper (genauer ist das dann in dem Auszug des Buches umschrieben, dass ich Dir kopiert habe). Solltest Du nicht ganz verstanden haben, würde es mich (wirklich) freuen, solltest Du mich fragen.

Denn Wie rechne ich eigentlich eine „normale“ Zahl (also im Dezimalsystem) in eine Binärzahl um?
Man nehme die Zahl und teile sie durch 2, sollte es einen Rest[1] geben, so notieren wir den Rest (wenn man durch 2 teilt kann das nur 0 oder 1 sein) in der rechten Spalte einer Tabelle und das ganzzahlige Ergebnis in der linken Spalte der Tabelle.

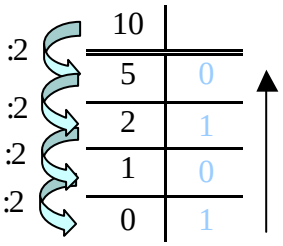
Beispiel: wir wollen die 9_{10} ins Binärsystem übersetzen:



Und Tada! Wir haben unsere Binärzahl. (man muss sie von unten nach oben lesen)

Also $9_{10} = 1001_2$

Noch ein Beispiel gefällig?



Also $10_{10} = 1010_2$

[1] Mit Rest ist hier nicht die zahl nach dem Komma gemeint, sondern z.B. bei 15 : 2 ist der Rest 1, d.h. 2 * 7 + 1 = 15. Einfacher zu verstehen ist das beim Teilen durch z. B. 7, denn z.B. bei 20 : 7 ist der Rest 6. Das kommt so zustande: Rest = 20 - das größte Vielfache von 7, das noch in die 20 passt. Ich bin zu dumm um das besser zu erklären!

Aufzeichnung der Vorlesung vom 27.10.2000

Nachdem wir jetzt Zahlen vom Dezimalsystem ins Binärsystem umrechnen können, sollten wir auch zeigen, dass das Binärsystem eindeutig ist. Eindeutigkeit heißt zweiunterschiedliche Zahlenketten im Binärsystem können nicht die gleiche Zahl im Dezimalsystem ergeben.

Beispiel

$17_{10} = 10001_2$, da

2^4	2^3	2^2	2^1	2^0
1	0	0	0	1

Wir versuchen die Eindeutigkeit anhand eines indirekten Beweises zu zeigen (indirekter Beweis: man zeigt, dass das Gegenteil nicht zutreffen kann)

Wir gehen also vom Gegenteil aus: zwei Zahlenketten sollen die gleiche zahl im Binärsystem ergeben.

$x = a_n a_{n-1} a_{n-2} a_{n-3} \ldots a_o$ (x ist hierbei die Zahl im Dezimalsystem)

Die Zahl im Dezimalsystem

Die eine Zahlenkette, die x ergeben soll

$x = b_n b_{n-1} b_{n-2} b_{n-3} \ldots b_o$

Die eine Zahlenkette, die x ergeben soll

da es nicht die gleichen Zahlenketten sein sollen:

$$a_n a_{n-1} a_{n-2} a_{n-3} \ldots a_o \neq b_n b_{n-1} b_{n-2} b_{n-3} \ldots b_o$$

uns ist jetzt schrecklich langweilig und weil wir gard' nix besseres zu tun haben (im Fernsehen kommt nur Müll) fangen wir an die Zahlenketten von vorn nach hinten Element für Element zu vergleichen.

$$a_n = b_n$$
$$a_{n-1} = b_{n-1}$$
$$a_{n-2} = b_{n-2}$$

das ist auch nicht grad interessant, aber dann kommt eine Stelle an der die Elemente nicht mehr gleich sind.

$a_k \neq b_k$ z.B. $a_k = 1$ und $b_k = 0$

das k ist hier einfach nur ein Index, der die erste unterschiedliche Stelle anzeigen soll.

$$a_n * 2^n + a_{n-1} * 2^{n-1} + a_{n-2} * 2^{n-2} + \ldots + a_k * 2^k + \ldots + a_o * 2^n$$

$$b_n * 2^n + b_{n-1} * 2^{n-1} + b_{n-2} * 2^{n-2} + \ldots + b_k * 2^k + \ldots + b_o * 2^n$$

Dieser Teil der Kelle ist gleich

Erstes unterschied-lichtes Element

Der Rest er Zahlenkette muss also den Unterschied es ersten unterschiedlichen Elementes ausgleichen

In unserem Beispiel ($a_k = 1$ und $b_k = 0$) muss der hier rot dargestellte Teil der Zahlenkette den Unterschied des grünen Elements ($a_k = 1$, $b_k = 0$) wieder ausgleichen.

Wir brauchen also folgendes
$b_{k-1} * 2^{k-1} + b_{k-2} * 2^{k-2} + \ldots + b_0 * 2^0 \geq 2^k$
warum? Klar das hier rote muss in der Summe das $2^k * a_k$ ausgleichen, da in unserem Beispiel $2^k * a_k = 2^k$ und $2^k * b_k = 0$.

Die größte Zahl, die wir mit $b_{k-1} * 2^{k-1} + b_{k-2} * 2^{k-2} + \ldots + b_0 * 2^0$ darstellen können ist der Fall, das alle b's 1 sind.
Klartext:

2^{k-1}	2^{k-2}	2^{k-3}	…	2^1	2^0
1	1	1	…	1	1

Momentchen mal, denkt sich da spontan der erfahrene Binärrechner. Wie war das noch? Richtig!

$$2^{k-1}*1 + 2^{k-2}*1 + 2^{k-3}*1 + \ldots + 2^0*1 = 2^k-1$$

(Beispiel $111_2 = 2^2 + 2^1 + 2^0 = 2^3 - 1 = 7$)

Tja, und da haben wir schon unseren Widerspruch, denn 2^k-1 ist nie und nimmer gleich oder gar 2^k, logisch oder?)

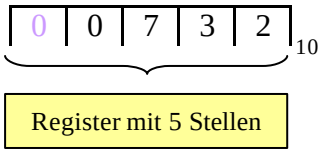
Also haben wir bewiesen, dass die Darstellung des Binärsystems eindeutig ist. ❑

RECHENOPERATIONEN

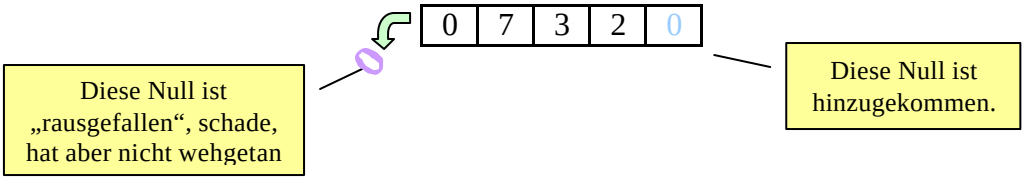
Toll jetzt können wir zahlen darstellen, aber mehr auch nicht! Also sollten wir uns mal ums Rechnen kümmern:

1. Multiplizieren

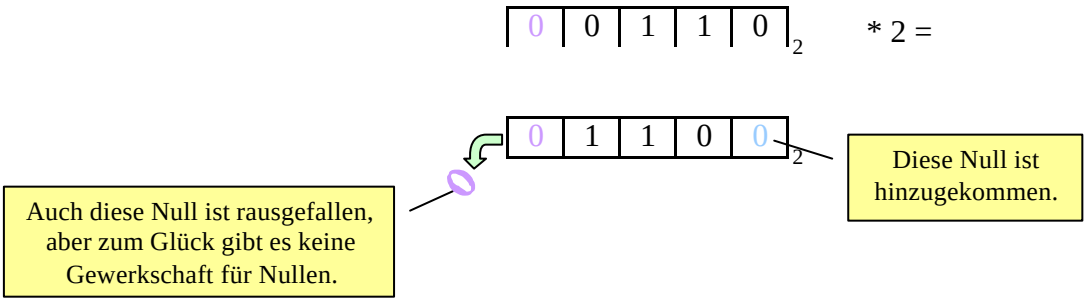
Man kann im Dezimalsystem ganz einfach mit 10 multiplizieren, in dem man eine 0 hinten an die Zahl anhängt. In der Informatik nennt man das einen „Shift left", soll heißen man rückt die Zahlen um eine Stelle nach links und fügt eine 0 ans Ende. Eine Besonderheit kommt jetzt noch hinzu: Der Computer kann nicht beliebig viele Stellen pro Zahl verwalten, sondern hat feste Stellenzahlen, sogenannte „Register". Eine zahl im Dezimalsystem z.B. 732 sieht in dem Computer mit Registern zu 5 stellen so aus:



Wenn wir unsere Zahl jetzt mit 10 multiplizieren sieht das so aus:



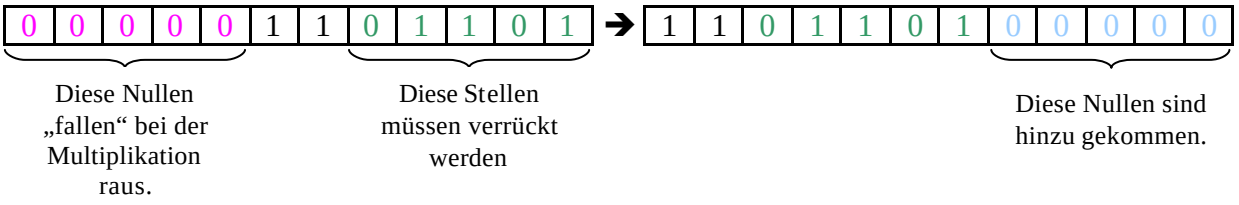
Das geht auch mit dem Binärsystem, nur ist es hier nicht die Multiplikation mit 10 sondern mit 2, die einen „Shift left“ ermöglichen:



Wenn man jetzt aber nicht nur mit 2 sondern z.B. mit 4 multiplizieren will, muss man einfach zweimal mit 2 multiplizieren:
0011_2 * 10_2

0011_2	mit Shift left =	0110_2	mit Shift left =	1100_2
3_{10}	* 2_{10} =	6_{10}	* 2_{10} =	12_{10}

Mann muss also in einem Register die Binärzahl um so viele Stellen verschieben, wie man den Multiplikator (hier 4) durch 3 Teilen kann.
Beispiel mit einem Register mit 12 Stellen:
Multiplikation mit $32_{10} = 2^5{}_{10}$

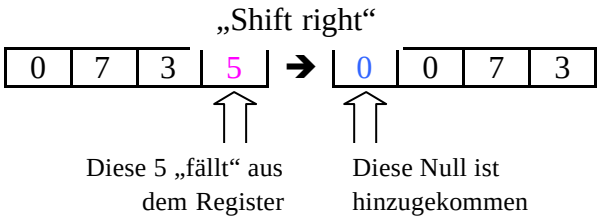


2. Division

Na, wenn wir bei einer Multiplikation einen „Shift left“ haben, was wird wohl bei einer Division sein? Klar ein „Shift right“. Wir betrachten die ganzzahlige Division[2]:

Beispiel: $\frac{735_{10}}{10_{10}} = 73_{10}$ Rest 5

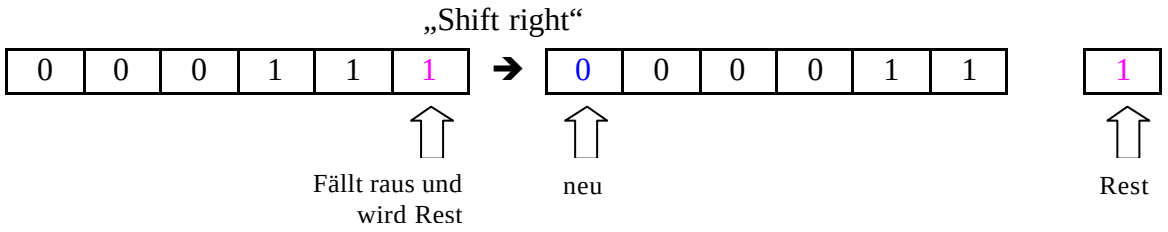
In einem Register mit z.B. 4 Bits sieht das so aus:



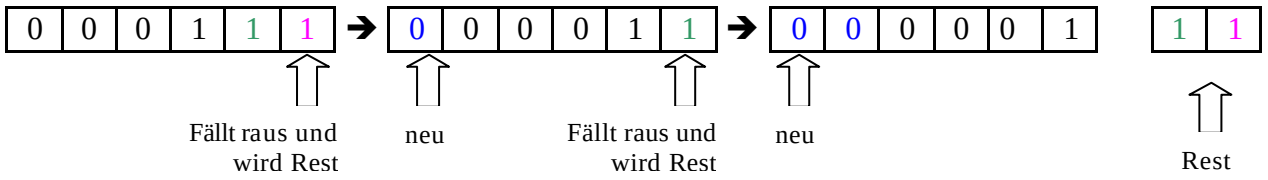
[2] Damit ist gemeint, dass wir eine Kommazahl beim Dividieren bekommen, sondern ganze Zahlen und einen ganzzahligen Rest.

Nützlich ist hierbei, dass das was auf dem Register fällt automatisch der Rest ist. Die 5, die in unserem Beispiel aus dem Register gefallen ist, stellt unseren Rest dar.

Das Ganze mal mit Binärzahlen (Register mit 6 Bits):



Wir können die 000111_2 ($=7_{10}$) auch durch 4 teilen, und wie? Klar durch 2 mal „Shift right“:



Wir haben also 000111_2 durch 100_2 geteilt und raus kommt 00001_2 Rest 11_2, stimmt auch, denn $7_{10} / 4_{10} = 1_{10}$ Rest 3_{10}. Mit Recht wir man sich jetzt aber fragen, was ist mit dem Rest? Was macht der denn so, wenn der aus den Register „rausgefallen“ ist? Urlaub auf Mallorca? Nun zumindest wollen wir irgendwas haben, um diesen Rest noch später noch zu behalten. Dafür gibt es die in der Informatik sehr wichtige Funktion MODULO, welche eigentlich nichts anderes macht, als den Rest einer Division zu bestimmen. 7 mod 4 ergibt also den Rest von 7 durch 4, wie im Beispiel ist das 3 (7 mod 2 ist 1). Wenn man eine Binärkette hat und z.B. $011011101011101101010011_2$ mod 2^n ausrechnen will, muss man nur die *letzten* n Stellen betrachten und das ist das Ergebnis von mod 2^n!

3. Addition

Das haben wir doch schon in der 1. klasse gehabt: ein Apfel plus ein Apfel = 2 Äpfel. Naja viel hat sich dazu nicht geändert, nur das wir jetzt im Binärsystem rechnen. Das geht aber auch wie gehabt.
Wir können zum Beispiel zwei 6stellige Register (man sagt auch Register mit der Wortlänge 6) addieren:

	0	0	0	1	1	1
+	0	0	1	0	1	1

Das machen wir wie in der ersten Klasse, fangen von hinten nach vorne an: 1 + 1 ergibt eigentlich 2, wir müssen im Binärsystem aber eine 0 eintragen und über der Rechnung (hier in blau) eine „1 im Sinn“ vermerken, da wir eine 2 im Binärsystem nicht darstellen können.

		1	1	1	1	⇐	„im Sinn“
	0	0	0	1	1	1	
+	0	0	1	0	1	1	
	0	1	0	0	1	0	

Eigentlich klar, aber warum steht in der 2ten Zeile (die mit den drei 1sen untereinander) eine 1? Nun, drei 1sen ergeben eine Drei, eine 3_{10} im Binärsystem ist 11_2, die hintere 1 schreiben wir hin und die andere kommt in der nächsten Zeile „in den Sinn“.

Jetzt stehen wir aber ganz entschieden vor einem Problem! Was passiert, wenn ich folgende zwei Zahlen in einem Register der Wortlänge 8 addiere?

	1	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	1

Eigentlich müsste da doch

1	1	1	1	1	1	1	1	
	1	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0

Kann man nicht in einem Register mit 8 Bits unterbringen!

rauskommen. Tut es irgendwie auch, aber das Ergebnis „passt“ nicht mehr in ein Register der Wortlänge 8. Was jetzt? So schwer es uns auch fällt, trennen wir uns von allem, was länger als unsere ursprünglichen 8 Bits lang ist, hier trennen wir also die eins ab und das Ergebnis ist 00000000_2 also Null.

Wir erinnern uns, mit der Funktion MODULO 2^n bekommen wir immer die letzten n Stellen einer Zahl. Wir können daher folgendes verallgemeinern:
Wenn wir zwei Zahlen a und b in einem Register mit der Wortlänge n addieren wollen, dann besteht das Ergebnis nur aus den letzten n Zeichen dieser Summe. Formel: $(a+b) \bmod 2^n$.
Um das in Zukunft einfacher zu haben schreiben wir:

$$(a+b) \bmod 2^n = a \oplus_n b$$

(n steht für die Wortlänge des Registers)

Also noch mal zu mitschreiben: wir addieren zwei Zahlen im Register mit der Wortlänge n und betrachten die letzten n Stellen mithilfe der Funktion MODULO 2^n.

4. Subtrahieren

ich hoffe bis hierher war alles soweit verständlich, denn jetzt kommt das schwierigste, das Subtrahieren. Subtrahieren ist deshalb so schwer, weil wir bis jetzt noch nicht einmal negative Zahlen darstellen können. Da ich mir aber zu Geburtstag gewünscht habe negative binäre Zahlen darstellen zu können (hätte mir doch die Eisenbahn wünschen sollen), gibt es das sogenannte ZWEIERKOMPLEMENT. Mit Komplement ist hier nicht Komplement im Sinne von höflicher Schmeichlung gemeint, sondern vielmehr Komplement im Sinne von komplementär also gegensätzlich.
Bei einem Zweierkomplement ändert sich eigentlich nicht viel, außer dass der erste Faktor negativ ist.

„normale“ Darstellung	2^n	2^{n-1}	…	2^3	2^2	2^1	2^0
Zweierkomplement	-2^n	2^{n-1}	…	2^3	2^2	2^1	2^0

Beispiel:

-2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	1	0	1

Wie rechen die zahl 10111101_2 genauso um wie gehabt, nur dass das erste Element eben negativ ist, also:

$1 * -2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = -64_{10} + 29_{10} = -35_{10}$

Also wir können in einem Register mit dem Zweierkomplement sowohl negative Zahlen darstellen (indem wir das erste Bit auf 1 setzen) als auch positive Zahlen (indem wir das erste Bit aus 0 setzen).
Ein paar Beispiele (in einem Register mit der Wortlänge 5):

$1_{10} = 00001_2$	$-1_{10} = 11111_2$	$2_{10} = 00010_2$	$-2_{10} = 11110_2$
$10_{10} = 01010_2$	$-10_{10} = 10110_2$	$-16_{10} = 10000_2$	$15_{10} = 01111_2$

Die höchste Zahl, die wir im Zweierkomplement darstellen können ist $2^{n-1}-1$ und die niedrigste ist $(-1) * 2^n-1$.

So können wir auch subtrahieren, wir addieren einfach die negative Zahl, denn $2_{10} - 2_{10}$ ist gleich $2_{10} + (-2_{10})$.

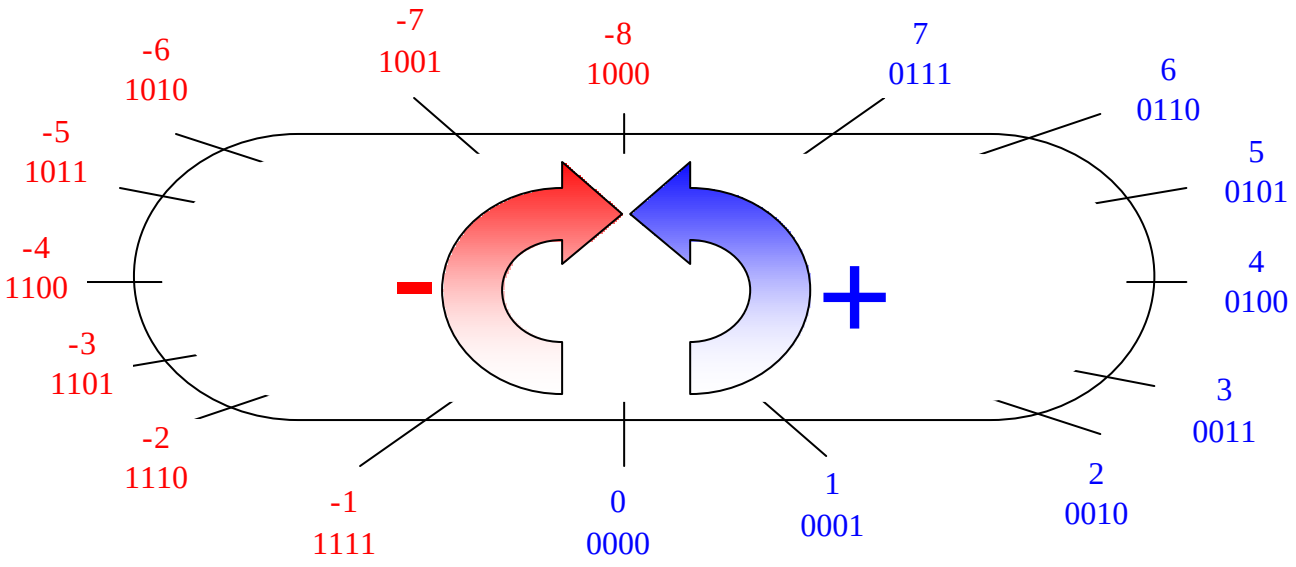
z.B. $2_{10} - 2_{10} = 00010_2 + 11110_2 =$

= 2_{10} + = -2_{10}

	1	1	1		
	0	0	0	1	0
+	1	1	1	1	0
	0	0	0	0	0

„im Sinn“

Die letzte blaue Eins im Sinn ist positiv und die rote Eins ist negativ, beide heben sich gegeneinander auf.
Man kann die Zahlen (z.B. im Register mit der Wortlänge 4) auch in einem „Zahlenkreis“ darstellen:



Aufzeichnung der Vorlesung vom 3.11.2000

So, da wir jetzt mit Binärzahlen rechnen können sollten, können wir uns jetzt mal um die Hardwareumsetzung des Ganzen im Computer kümmern. Der Computer kann natürlich nicht „verstehen“ wie wir rechnen wollen, er braucht eine geeignete Hardware um unseren Bedürfnissen entsprechen zu können. Der Computer besteht aus unterschiedlichen „logischen Gattern“ (das ist der Inhalt der Vorlesung gewesen).

Logische Gatter

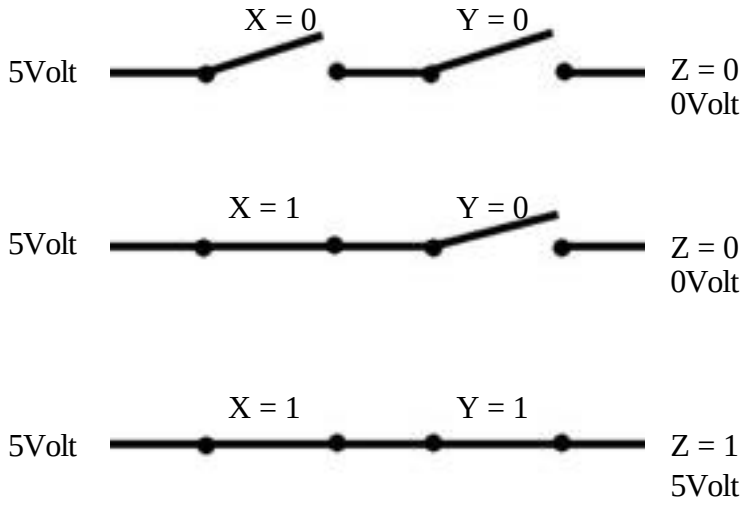
Unter logischen Gattern verstehen wir logische Operationen, die man durch Schaltungen simulieren kann. Im Folgenden werden die wichtigsten logischen Gatter vorgestellt:

A) AND

AND hat die Eigenschaft zwei eingehende Zustände (z.B. x und y) zu vergleichen, und gibt einen Zustand (z.B. z) wieder, wenn beide Zustande 1 sind. Dies wird in folgender „Wahrheitstabelle“ veranschaulicht:

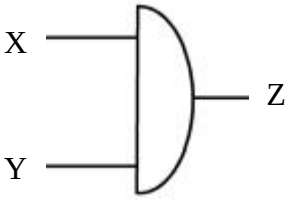
x	y	z
0	0	0
1	0	0
0	1	0
1	1	1

Umgangssprachlich könnte man sagen: die das logische Gatter wird nur dann 1 wenn auch x und y 1 sind. Mathematisch fällt auf, dass z = x * y. Doch wie kann man das durch eine Schaltung simulieren? Hierzu eine kleine Skizze:

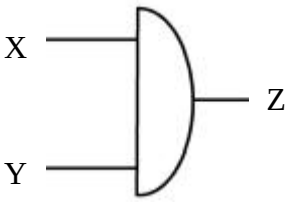


Wie legen also an einer Seite der Schaltung 5 Volt an und sehen nach, was an der anderen Seite herauskommt, ist einer er Schalter offen, so kann die Spannung nicht am Ende „herauskommen“ und der Zustand ist null, da null Volt ankommen. Erst wenn beide Schalter X und Y 1 sind (also den Strom hier weiterleiten) kann die Spannung am Ende ankommen und Z ist eins, da die 5Volt auch hier zu messen sind.

Man kann das auch Symbolisch darstellen (gut für größere Schaltpläne):



Man kann also
X AND Y
X * Y
Oder



schreiben und alles bedeutet das gleiche.

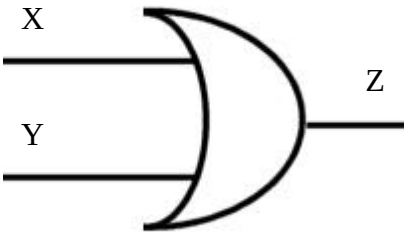
B) OR

Bei OR gilt folgende Wahrheitstabelle:

x	y	z
0	0	0
1	0	1
0	1	1
1	1	1

Das heißt entweder x oder y (oder beide) sind eins, dann ist auch Z eins. Man schreibt auch $A \oplus B$.

Man schreibt für OR symbolisch:



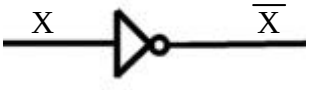
C) NOT

NOT oder NEGATION dreht die Werte um, die eingegeben werden, des gibt daher kein X und Y, sondern nur ein X:

x	z
0	1
1	0

Man schreibt auch:
NOT(X)
NEG(X)
oder auch $\overline{X}$

oder symbolisch



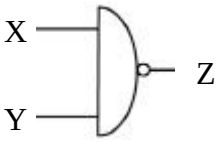
Mit diesen Drei Operationen kann man schon ne Menge anfangen. Es gibt jedoch noch ein paar mehr:
Mit zwei Zuständen (X und Y) kann man insgesamt (theoretisch) 16 unterschiedliche Kombinationen für einen Ausgang Z konstruieren. Ein Paar sind in folgender Tabelle verzeichnet:

X	Y	Z									
		f_0	NOR	f_2	NOT(X)	XOR	AND	OR	NAND		f_{15}
0	0	0	1	0	1	0	0	0	1		1
0	1	0	0	1	1	1	0	1	1	…	1
1	0	0	0	0	0	1	0	1	1		1
1	1	0	0	0	0	0	1	1	0		1

Ist immer 0 — Kennen wir schon — Ist immer 1

Neben den uns bekannten AND, OR und NOT gibt es da anscheinend noch andere, die einen Namen haben, z.B. NAND.

- NAND ist die Negation von AND, d.h. wann immer AND 1 wird ist NAND 0 und andersherum. Da diese Funktion recht interessant ist, hat sie sogar eine eigenes Symbol:



- NOTX betrachtet nur den X wert, ist der Null, dann ist auch Z null.
- XOR wird nur 1, wenn einer von beiden, also X oder Y 1 ist, nicht wenn beide 1 sind.
- NOR ist die Negation von OR und wird immer dann 1 wenn OR 0 wird und andersherum.
- f_0 und f_{15} haben keinen besonderen Namen, da sie, wie die anderen Operationen ohne eigenen Namen nicht so interessant sind.

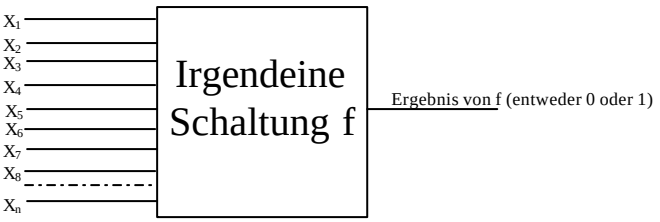
Was kann man damit anfangen? Man kann z.B. folgenden Satz aufstellen:

Satz: AND, OR und NOT bilden eine logische Basis, d.h. eine beliebige logische Funktion von n Argumenten kann damit gebaut werden.

Im Klartext soll das heißen, dass man was immer man mit n unterschiedlichen Zuständen erstellen mag kann man erstellen:

$$f(x_1, x_2, x_3, \ldots, x_n) = \begin{cases} 0 \\ \\ 1 \end{cases} \qquad x \in \{0,1\}$$

Immer noch nichtklar was man da tun will? Nun, man will n Zustände (die entweder 0 oder 1 sind) in eine Schaltung geben und je nach dem wie sie gebaut ist, erhält man 0 oder 1 aus der Schaltung zurück. Dazu folgendes Bildchen:



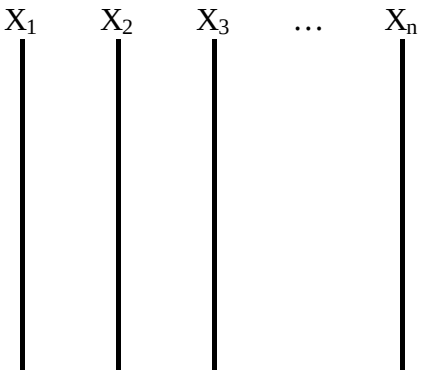
Eine solche Schaltung könnte so aussehen:
(in tabellarischer Form)

X_1	X_2	X_3	…	X_n	f
0	0	0	…	0	0
0	0	0	…	1	0
0	**1**	**0**	**…**	**0**	**1**
1	**1**	**0**	**…**	**1**	**1**
1	**1**	**1**	**…**	**1**	**1**

Hier sind viele Zustände dazwischen, die uns nicht interessieren insgesamt gibt es 2^n mögliche Zustände

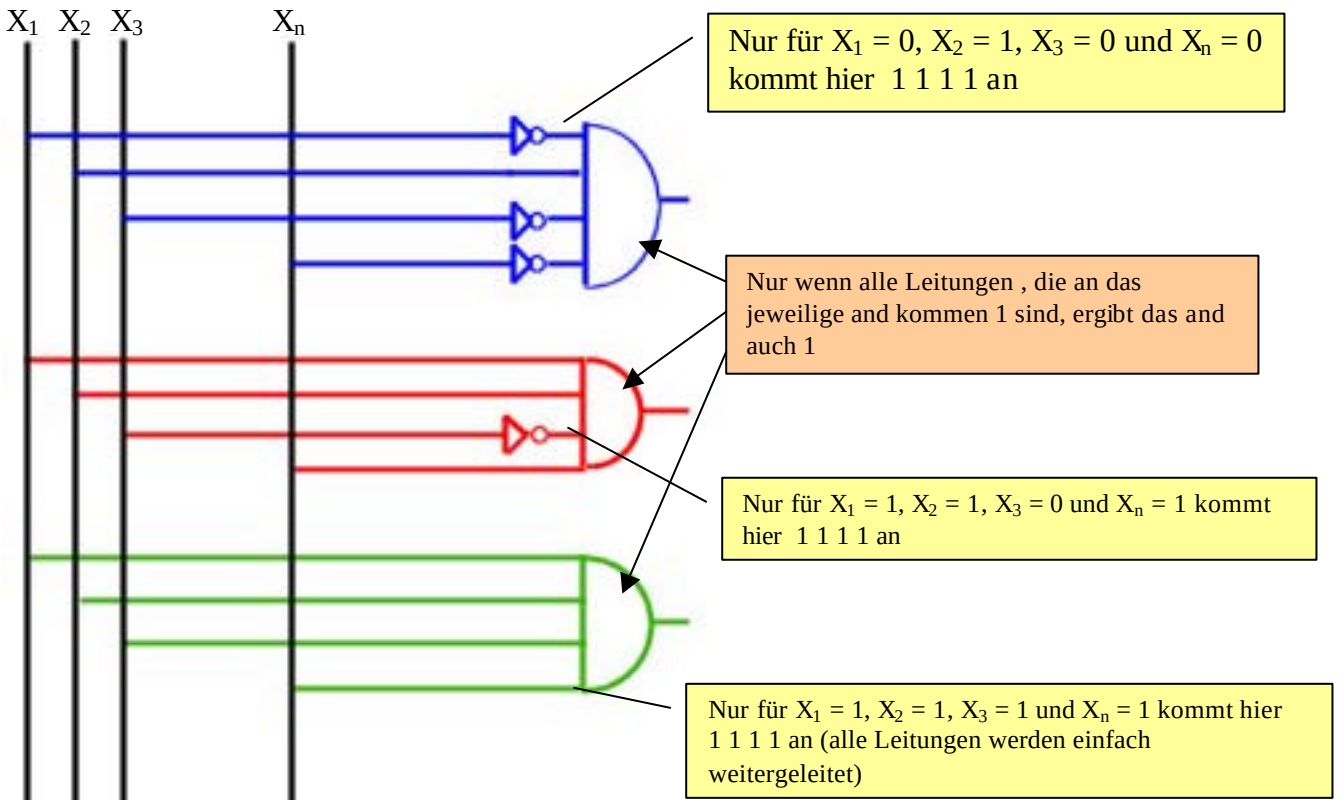
Nehmen wir an, diese Schaltung soll nur in den 3 Fällen, die hier blau, rot und grün dargestellt sind 1 ergeben in allen andern Fällen aber 0.

Tja wie machen wir das jetzt? Nun zunächst brauchen Wir die Zustände X_1 bis X_2 und geben jedem eine eigene Leitung:

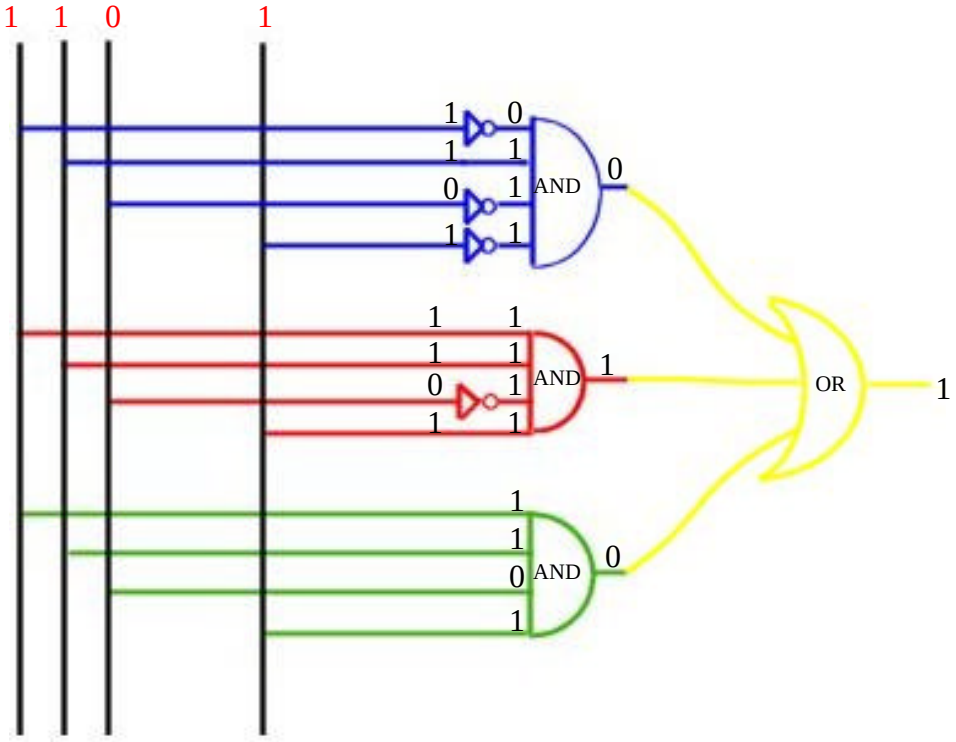


Ok, dafür gewinnen wir jetzt zwar keinen Nobelpreis, aber wir sind der Sache schon näher gekommen, und jetzt? Schauen wir uns die blaue Reihe an, wir Müssen also dafür sorgen, dass wenn $X_1 = 0$ und $X_2 = 1$ und $X_3 = 0$ und für $X_n = 0$ unsere Schaltung 1 ergibt (so eine Schaltung wird dann auch Decoder genannt, weil er unseren Code untersucht und uns 1 oder 0 herausgibt, wenn alles stimmt).

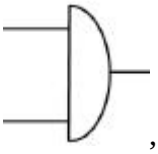
Wir nehmen also unsere Leitungen und (trickreich wie wir sind) „kippen“ die 0 zur 1 (mit unserem NOT), wenn die Leitung $X_{irgendwas}$ 0 ist und lassen sie 1 wenn die Leitung $X_{irgendwas}$ bereits 1 ist. Damit wir die Leitungen später noch benutzen können, zweigen wir diese Untersuchung nach rechts ab:



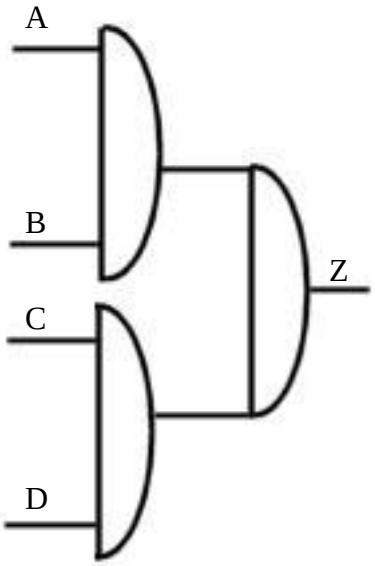
Es kann immer nur höchstens ein AND 1 ergeben, da alle Schaltungen unterschiedlich sind. Machen wir ein Beispiel und schauen uns an, was passiert, wenn wir die rote Zeile (oben in der Tabelle auf Seite 14) mit $X_1 = 1$, $X_2 = 1$, $X_3 = 0$ und $X_n = 1$ einsetzen:



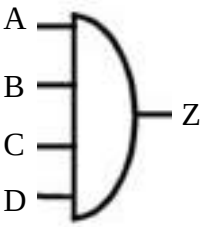
Also wir sehen, dass nur bei Rot (dem mittleren Decoder) das and eins ergibt und bei keinem anderen. Wir haben also einen Decoder gebaut, der immer dann 1 ergibt (hinter dem gelben or), wenn eine der drei bunten Zeilen in der Tabelle auf Seite 14 angelegt werden. Für alle anderen Möglichkeiten ergibt dieser Decoder 0. Hier ist (hoffentlich) auch erkennbar, warum immer nur ein and 1 sein kann.

Ach ja, Man kann (wie ich auch schon obern angewendet habe) das AND nicht nur für X und Y verwenden, also  , sondern auch für mehrere:

z.B. mit 4 eingehenden Zuständen:



Ist das gleiche wie



(Das kann man machen, weil AND kommutativ ist)

Anmerkung:

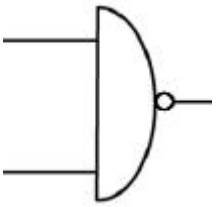
- Auch AND und NOT bilden zusammen eine logische Basis
- Or und NOT bildet zusammen eine logische Basis
- NAND bildet eine logische Basis
- NOR bildet eine logische Basis

Um das zu zeigen (z.B., dass NAND allein eine logische Basis darstellt), muss man nur zeigen, dass man aus den entsprechenden Komponenten die anderen Operationen (NOT, AND, OR) nachbauen kann.

Beispiel: Beweis, dass NAND allein eine logische Basis darstellt:

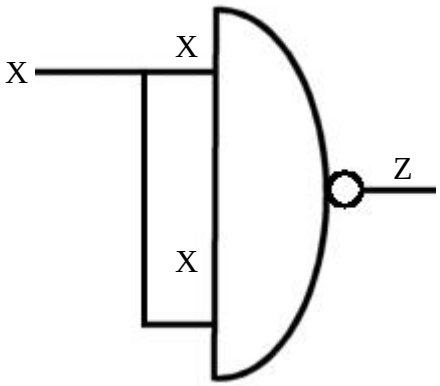
Wir erinnern uns, wie die Tabelle von NAND aussah:

X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

Das Zeichen von NAND war 

WIR BAUEN UNS EIN NOT AUS NAND:

Und das soll ein NOT sein?



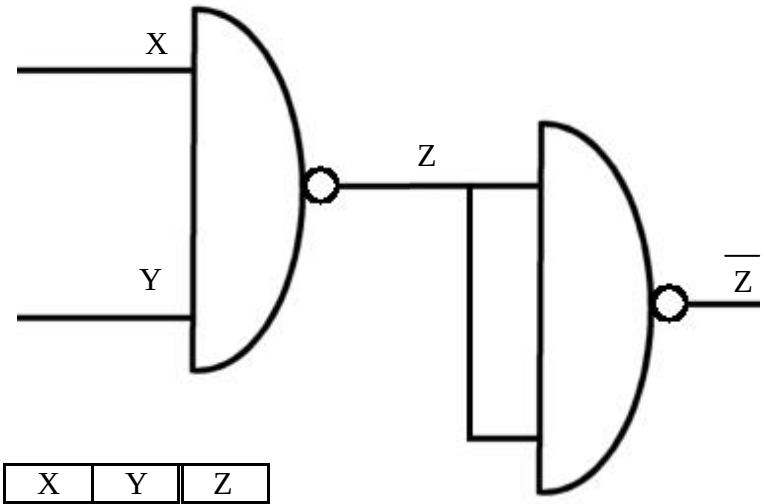
Hmm, wenn man genau hinschaut, wird ja folgendes gemacht: ein eingehender Zustand wird verdoppelt und es gehen immer die gleichen Zustände in die Schaltung. Also zweimal 1 oder zweimal 0. passen wir unsere Tabelle also an:

X	X	Z
0	0	1
1	1	0

Stimmt! Das ist die Tabelle für NOT.

WIR BAUEN UNS EIN AND AUS NAND:

Das ist einfach! NAND ist doch das Negativ von AND, wenn wir NAND wieder negativ nehmen bekommen wir ein AND:



Passt! Die Tabelle dieser Operation ergibt das gleiche wie and (auch wenn das Ding ein wenig zu kompliziert aussieht)

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

WIR BAUEN UNS OR AUS NAND:

Das wird schon schwerer. Zum glück gab's da mal eine Mathematiker, der uns da helfen könnte, der hat nämlich die (nach ihm benannten) De Morganschen Gesetze entwickelt:

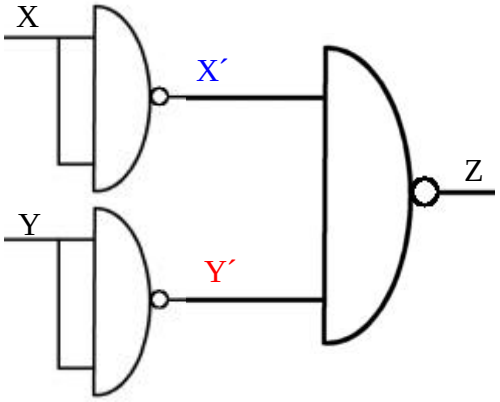
1. Gesetz: X AND Y = NOT(NOT(X) or NOT(Y)) in unserer Notation
$X \wedge Y = \neg(\neg X \vee \neg Y)$ für die Mathematiker
$X \cap Y = (X' \cup Y')'$ und so in der Mengenlehre

ist ja ganz nett, wir brauchen aber das andere Gesetz:

2. Gesetz: X OR Y = NOT(NOT(X) AND NOT(Y))

da haben wir also or ersetzt durch Ausdrücke, die wir schon kennen, nämlich NOT und AND.

Unsere or Schaltung muss also so aussehen:

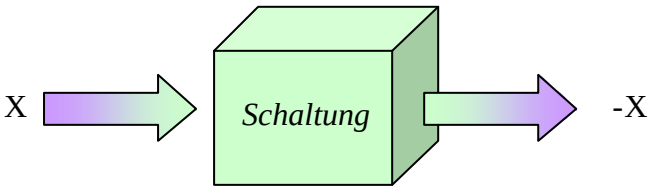


Sieht aber komisch aus, schauen wir mal, ob das noch stimmt. Aus x wird erst mal X´ und aus Y wird Y´ NOT(NOT(X) AND NOT(Y)), Ok soweit richtig. Und aus X´ und Y´ soll durch NOT(X´ AND Y´) or werten. Hey, auch richtig: NOT(AND) ist ja NAND!

Also wir haben jetzt NOT AND und OR aus NAND zusammengebastelt, also ist der Beweis, dass NAND eine logische Basis ist, abgeschlossen!

Machen wir mal ein praktischen Beispiel: wir wollen eine Schaltung zusammenbauen , die uns aus einer Bitkette (also 0en und 1sen in Binärdarstellung) ihren negativen Wert in der Zweierkomplementärdarstellung zurückgibt.

Also



Aber wie?

1. Schritt

Nehmen wir also eine Zahl z.B. 0010 und kippen sie bitweise. Also

0	0	1	0
1	1	0	1

Steht oben eine 0 nehmen wir unten eine 1 und andersherum

2. Schritt

nehmen wir diese Zahl und addieren eins hinzu

Im Sinn

		1		
	1	1	0	1
+	0	0	0	1
	1	1	1	0

Und raus kommt 1110, welches in der Zweierkomplementärdarstellung gleich -2_{10} ist. Unser Algorithmus zum negieren von Zahlen war also richtig.

Und warum?

Zu Schritt eins:

Das bitweise Komplement erreicht man, indem man von der Zahl 2^n-1 die Zahl X subtrahiert (da die Zahl 2^n-1 nur aus 1sen besteht kann man jedes Bit für sich alleine betrachten, also einfach „kippen“).

$X = X_{n-1}\ X_{n-2}\ X_{n-3}\ X_{n-4}\ldots\ X_0$

$$\begin{array}{rcccccc} & 1 & 1 & 1 & 1 & \ldots & 1 \\ - & X_{n-1} & X_{n-2} & X_{n-3} & X_{n-4} & \ldots & X_0 \\ \hline & \overline{X_{n-1}} & \overline{X_{n-2}} & \overline{X_{n-3}} & \overline{X_{n-4}} & & \overline{X_0} \end{array}$$

Zu Schritt 2:

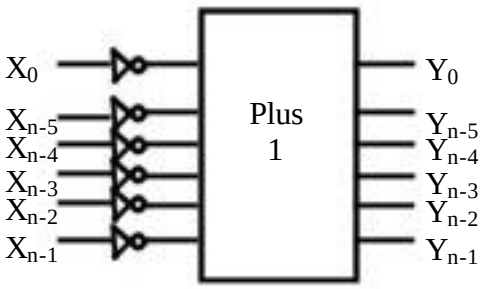
Haben wir das gemacht müssen wir nur noch eins addieren, also insgesamt:
$2^n - 1 - x + 1$, das ist das gleiche wie $2^n - X$.

Test: $X + 2^n - X = 2^n \equiv 0$ (weil andere Lösungen nicht in unser Register passen)

und für n = 3 X =2:
$2^3 - 2 = 6$ → Test 2 + 6 = 8

$(x + 2^n - x) \bmod 2^n$
$= 2^n \bmod 2^n$
$= 0$ stimmt also

Wir bräuchten also eine Schaltung, die ungefähr so aussieht:



Aufzeichnung der Vorlesung vom 10.11.2000

Wir haben in den letzten Vorlesungen die Bedeutung des logischen Gatters kennen gelernt. Wir müssen aber nicht immer aufwendige Schaltpläne (aus einzelnen Decodern) konstruieren um eine Funktion korrekt umzusetzen, denn wir können erst einmal eine Formel aufstellen und diese dann vereinfachen.

Wie kann ich aus einer Funktion eine Formal erstellen?
Also angenommen, eine Funktion $f(x_1,x_2,x_3,x_4)$ mit 4 Eingabebits (x_1,x_2,x_3,x_4) sei wie folgt gegeben.

x_1	x_2	x_3	x_4	$f(x_1,x_2,x_3,x_4)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Hier ist also eine Tabelle angegeben, in der gewisse (die blauen) Zeilen und nur diese 1 ergeben.
Genauso wie wir auf Seite 15 einen Decoder gebaut haben, erstellen wir jetzt unsere Formel.
Wir nehmen uns jede Zeile und negieren den Eingabewert, wenn er 0 ist und lassen ihn so wie gehabt, wenn er bereits 1 ist:
$(\overline{x_1} * \overline{x_2} * x_3 * x_4)$ für die erste Zeile und so weiter und so weiter. Eine fertige Formel sieht dann so aus:
$(\overline{x_1} * \overline{x_2} * x_3 * \overline{x_4}) + (\overline{x_1} * \overline{x_2} * x_3 * x_4) + (\overline{x_1} * x_2 * \overline{x_3} * \overline{x_4}) + (\overline{x_1} * x_2 * x_3 * x_4) + (x_1 * \overline{x_2} * \overline{x_3} * x_4) + (x_1 * x_2 * \overline{x_3} * \overline{x_4}) + (x_1 * x_2 * \overline{x_3} * x_4) + (x_1 * x_2 * x_3 * x_4)$
Die Negation ist hier durch einen Strich über dem Wert ausgedrückt.
Ein AND bedeutet soviel wie * und OR soviel wie + (wichtig zu wissen!)

Jetzt haben wir zwar eine Formel und jetzt?
Nun vereinfachen ist leichter gesagt als getan, aber zum Glück gab es da einen Mathematiker, der erkannte, dass das rechnen mit einem Ausschnitt aus den Natürlichen Zahlen, nämlich 0 und 1 in Zukunft wichtig werden würde und daher die folgenden Booleschen Gesetze (nach ihm benannt) aufgestellt:
Die unterschiedlichen eingabewerte sind nicht x_1, x_2, x_3,…x_n sondern X, Y, Z,…
Bevor es mit den Gesetzten los geht noch mal zu AND und OR:

AND

X \ Y	0	1
0	0	0
1	0	1

Das entspricht rechnerisch X * Y

OR

X \ Y	0	1
0	0	1
1	1	1

und das ist X + Y (+ ist hier das sog. Boolsche plus, ist nicht das gleiche wie das normale plus, da 1 + 1 = 1)

DIE BOOLSCHEN GESETZTE

No.1a)

$$X * Y = Y * X$$

da müssen wir auch nichts weiter beweisen, weil das nach dem Kommutativgesetz des * gilt.

No.1b)

$$X + Y = Y + X$$

Da ist es schon schwerer, da es sich hier nicht um das „normale“ plus, sondern um das boolsche Plus handelt. Wir können nichts anderes machen, als eine Wertetabelle zu erstellen, und nachprüfen.

X \ Y	0	1
0	0	0
1	0	1

Y \ X	0	1
0	0	0
1	0	1

Ja das wirklich gleich (man kann anbei Kommutativität auch durch Spiegelung an der Diagonalen (von links oben nach rechts unten) nachweisen, ist die Funktion an der Diagonalen symmetrisch, so ist sie auch kommutativ).

No. 2a)

$$(X * Y) * Z = X * (Y * Z) = X * Y * Z$$

auch hier müssen wir nichts nachweisen, weil Assoziativität bei * gegeben ist.

No. 2b)

$$\underbrace{(X + Y) + Z}_{\text{FORMEL 1}} = \underbrace{X + (Y + Z)}_{\text{FORMEL 2}} = X + Y + Z$$

so unangenehm das auch ist müssen wir wieder eine Wertetabelle machen und nachweisen, dass FORMEL 1 das gleiche ist wie FORMEL 2.:

X	Y	Z	(X+Y)	Formel 1	(Y + Z)	Formel 2
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Man sieht Formel 1 und Formel 2 ist das Gleiche !

No. 3a)

$$X * Y + X * Z = X\,(Y*Z)$$

Und schon wieder eine Tabelle

X	Y	Z	X*Y	X*Z	Formel 1	(Y + Z)	Formel 2
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
…							
1	1	1	1	1	1	1	1

Ich habe hier einfach mal ein paar Zeilen ausgelassen, wer nicht glaubt, dass das stimmt darf gerne alle ausfüllen!

No. 3b)

$$(X + Y) * (X + Z) = X + (Y * Z)$$

X	Y	Z	X+Y	X+Z	Formel 1	(Y + Z)	Formel 2
…							
1	0	1	1	1	1	0	1
…							

Auch wenn nur ein Element hier respektive gezeigt wurde: es stimmt für alle Kombinationen!

No. 4a)

$$0 * X = 0$$

da es sich um das uns bekannte * handelt, kennen wir das ja schon

No. 4b)

$$1 + X = 1$$

eine Besonderheit des boolschen Plus (da 1 + 1 = 1)

No. 5a)

$$1 * X = X$$

da es sich hier um unser bekanntes * handelt, dürfte das klar sein

No. 5b)

$$0 + X = X$$

auch klar, denn in dieser Eigenschaft unterscheidet sich das boolsche + nicht von unserem „normalen“ plus.

No. 6a)

$$X * X = X$$

Für beide Fälle stimmt das: 1 * 1 = 1 und 0 * 0 = 0

No. 6b)

$$X + X = X$$

Wider aufgrund der Besonderheit des boolschen plus: 1 + 1 = 1 und 0 + 0 = 0

No. 7a)

$$X * \overline{X} = 0$$

für 1 * 0 = 0 und 0 * 1 = 0

No. 7b)

$$X + \overline{X} = 1$$

für 0 + 1 = 1 und 1 + 0 = 1

No. 8a)

$$X * Y + X * \overline{Y} = X$$

Hier können wir bereits obere Gesetze anwenden und so den Ausdruck vereinfachen:
nach 3a) $= X * (Y + \overline{Y})$ nach 7b) $= X * 1$ nach 1a) $= 1 * X$ nach 5a) $= X$

No. 8b)

$$(X + Y) * (X + \overline{Y}) = X$$

kann man auch durch die bereits erwähnten Rechengesetze beweisen, muss man aber nicht.

No. 9a)

$$X + X * Y = X$$

Das beweisen wir mal, weil's so schön ist:

nach 5a) $X + X * Y \rightarrow X * 1 + X * Y \rightarrow X * (1+Y) \rightarrow$ (nach 4b) $X * 1 \rightarrow X$

No. 9b)

$$X * (X+Y) = X$$

No. 10a)

$$X + \overline{X}\, Y = X + Y$$

Das zu beweisen ist schon etwas schwieriger, zunächst erweitern wir mal auf beiden Seiten mit + X Y:

$X + X\,\overline{Y} + X\,Y \quad = \quad X + Y + X\,Y$

$X + Y\,(\overline{X} + X) \quad = \quad X + X\,Y + Y$

$X + Y \quad = \quad X + Y$

Damit ist bewiesen, das $X + \overline{X}\, Y = X + Y$

No. 10b)

$$X\,(\overline{X} + Y) = X\,Y$$

No. 11a)

$$X\,Y + \overline{X}\,Z + Y\,Z = X\,Y + \overline{X}\,Z$$

No. 11b)

$$(X+Y)\,(\overline{X}+Z)\,(Y+Z) = (X+Y)\,(\overline{X}+Z)$$

No. 12a)

$$X\,Y + \overline{X}\,Z = (X+Z)\,(\overline{X}+Y)$$

No. 12b)

$$(X+Y)\,(\overline{X}+Z) = X\,Z + \overline{X}\,Y$$

No. 13a)

$$\overline{X\,Y} = \overline{X} + \overline{Y}$$

Beweis durch Inspektion (also durch erstellen einer Wertetabelle)

No. 13b)

$$\overline{(X + Y)} = \overline{X} * \overline{Y}$$

No. 14

$$\overline{f}(X,Y,\ldots,Z,+,*) = f(\overline{X},\overline{Y},\ldots,\overline{Z},*,+)$$

da wir jetzt schon so viele Rechenregeln aufgestellt haben, können wir eigentlich auch mal eine gescheite Übersicht erstellen:

ÜBERSICHT ÜBER DAS 1 X 1 DER

RECHNERSTRUKTUREN

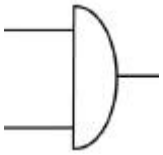
1. die boolschen Gesetze

1a)	$X * Y = Y * X$	1b)	$X + Y = Y + X$
2a)	$(X * Y) * Z = X * (Y * Z) = X * Y * Z$	2b)	$(X + Y) + Z = X + (Y + Z) = X + Y + Z$
3a)	$X * Y + X * Z = X * (Y + Z)$	3b)	$(X + Y) * (X + Z) = X + Y * Z$
4a)	$0 * X = 0$	4b)	$1 + X = 1$
5a)	$1 * X = X$	5b)	$0 + X = X$
6a)	$X * X = X$	6b)	$X + X = X$
7a)	$X * \overline{X} = 0$	7b)	$X + \overline{X} = 1$
8a)	$X * Y + X * \overline{Y} = X$	8b)	$(X + Y) * (X + \overline{Y}) = X$
9a)	$X + X * Y = X$	9b)	$X * (X + Y) = X$
10a)	$X + \overline{X} * Y = X + Y$	10b)	$X * (\overline{X} + Y) = X * Y$
11a)	$X * Y + \overline{X} * Z + Y * Z = X * Y + \overline{X} * Z$	11b)	$(X+Y) * (\overline{X}+Z) * (Y+Z) = (X+Y)\ (\overline{X}+Z)$
12a)	$X * Y + \overline{X} * Z = (X + Z) * (\overline{X} + Y)$	12b)	$(X + Y) * (\overline{X} + Z) = X * Z + \overline{X} * Y$
13a)	$\overline{X * Y} = \overline{X} + \overline{Y}$	13b)	$\overline{(X + Y)} = \overline{X} * \overline{Y}$
14	$\overline{f}(X,Y,\ldots Z,+,*)$	=	$f(\overline{X},\overline{Y},\ldots \overline{Z},*,+)$

2. logische Gatter

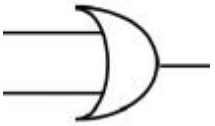
X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

X AND Y = X * Y



X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

X OR Y = X + Y



X	NOT
0	1
1	0

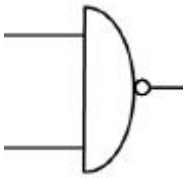
NOT X

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

X XOR Y = X $\oplus$ Y

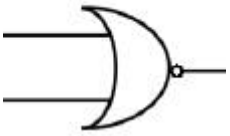
X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

X NAND Y



X	Y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

X NOR Y



Nach dieser wirklich heroischen Leistung uns annähernd unendliche Formeln herzuleiten und sie auswendig zu lernen (☺), könnten wir nun endlich angeln gehen, unseren Orden einfordern oder ähnlich spaßiges machen, aber leider ist da doch noch was, was ich loswerden muss: die KARNAUGH – MAPS. Das ist nicht direkt eine Schatzkarte, aber auf den ersten Blick ähnlich verwirrend. Also, eine sogenannte KARNAUGH – MAP ist eine andere Art der Darstellung von Formeln. Wir können entweder eine normale Tabelle machen (mit 2^{bits} Spalten), oder wir fertigen eine KARNAUGH – MAP an.
Kleines Beispiel zur Veranschaulichung: wir nehmen 4 zustände A,B,C,D und die Formel (A*B)+(C*D)

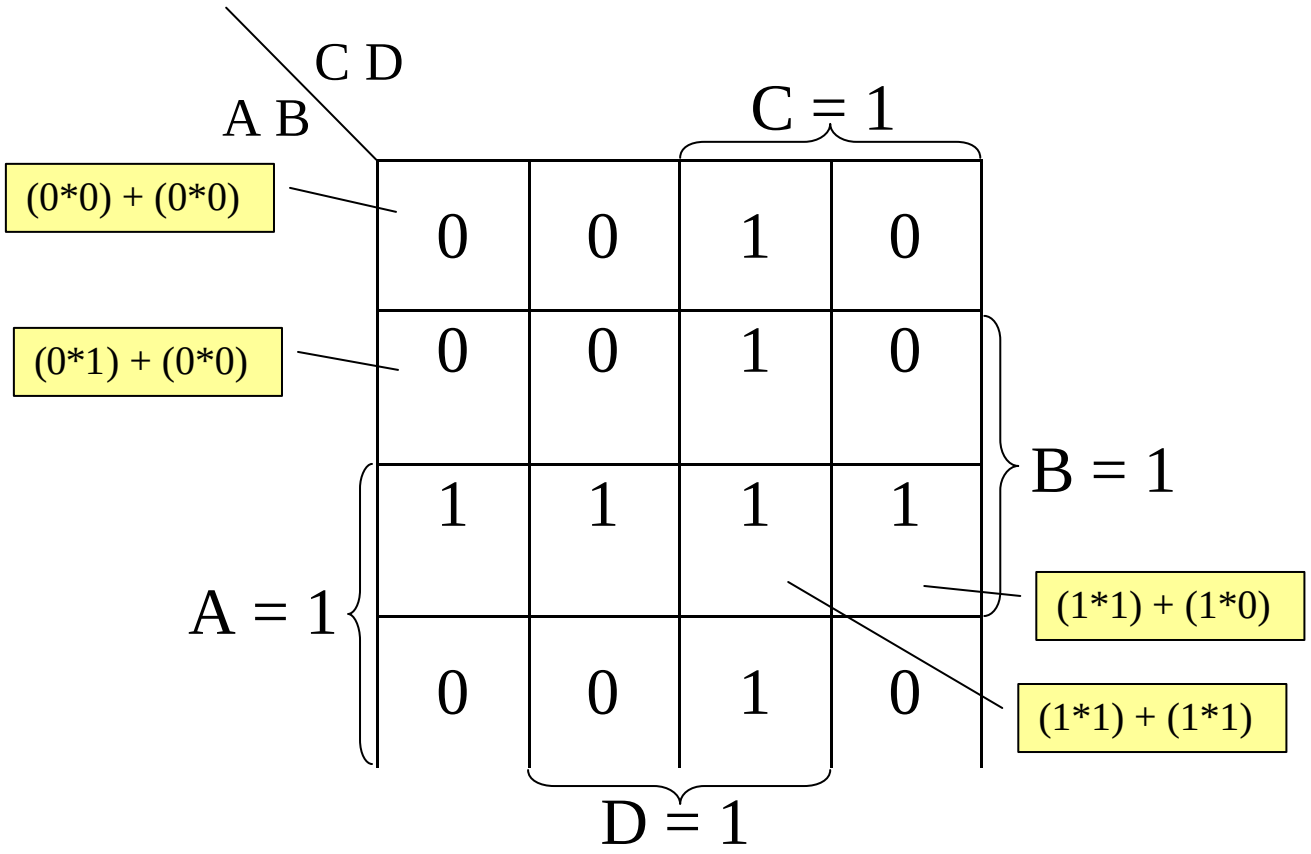
1. Möglichkeit eine „normale“ Tabelle

A	B	C	D	(A*B)	(C*D)	(A*B)+(C*D)
0	0	0	0	0	0	0
…				…		…
1	1	1	1	1	1	1

Hier fehlen $2^4 - 2$ Zustände, also noch 14

2. Möglichkeit eine KARNAUGH – MAP

Man rechnet in jedem Feld einfach damit den an der Seite stehenden Werten und trägt das Ergebnis in das betreffende Feld.



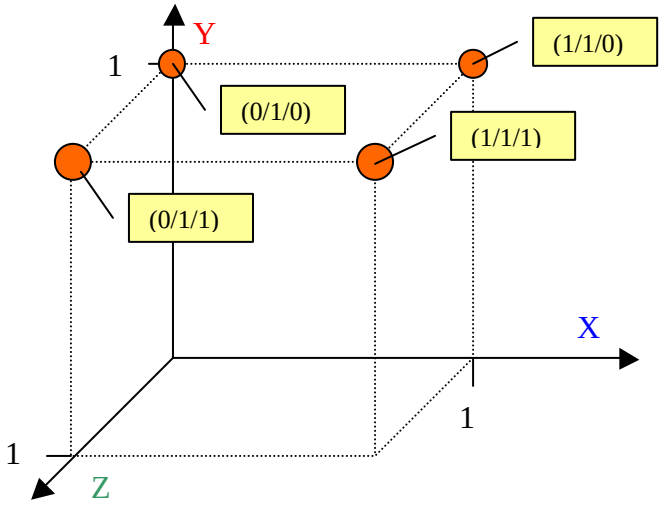
Aufzeichnung der Vorlesung vom 17.11.2000

Mit dem Urteil eine KARNAUGH – MAP sei nicht direkt eine Schatzkarte, war ich vielleicht ein wenig voreilig, diese Art der Darstellung birgt doch einigte Vorteile.
Nochmal zur herkömmlichen Darstellungsart einer Funktion z.B. f(x,y,z):

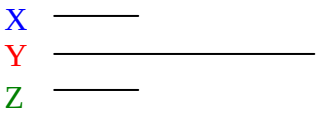
x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Man kann nun für diese Tabelle eine Schaltung aus Decodern basteln für jede Zeile, in der der Funktionswert 1 ist (in der Art wie wir es kennen).

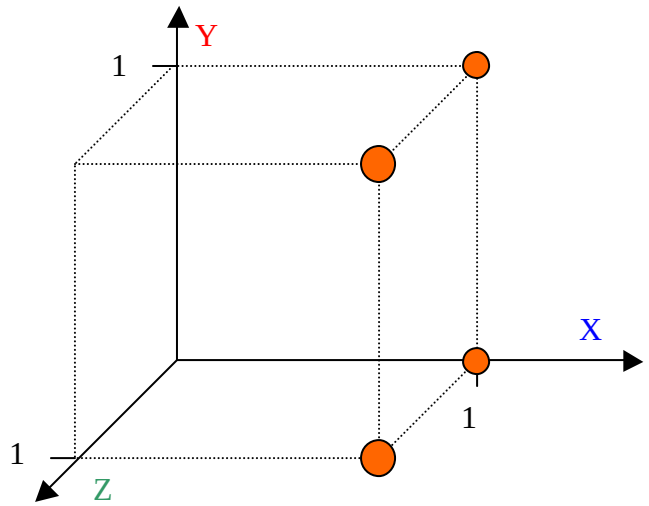
Stellt man sich man sich die Funktion aber im dreidimensionalen Raum vor, kann man einfachere Schaltungen finden:
Wir tragen an jeder Achse einen Wert an, 0 oder 1. Die Acht Ecken des Würfels sind unsere möglichen Zustände. Haben wir einen Eintrag an der betreffenden Stelle, machen wir dort einen Punkt (hier orange). Für unsere Funktion wäre das:
Die Koordinaten der Punkte entsprechen den Funktionswerten für X, Y und Z.



Wie man sieht, liegen unsere vier Punkte in einer Ebene. Sie liegen in einer Ebene, weil sie alle eines gemeinsam haben, nämlich
Y = 1. Für alle anderen Fälle, also dort wo Y nicht 1 ist haben wir keinen Punkt und unsere Funktion ist dort 0. Das zeigt, das die Funktion von X und Z unabhängig ist. Man kann also kurz schreiben:
f(x,y,z)= y. Als Schaltung sähe dass dann so aus:

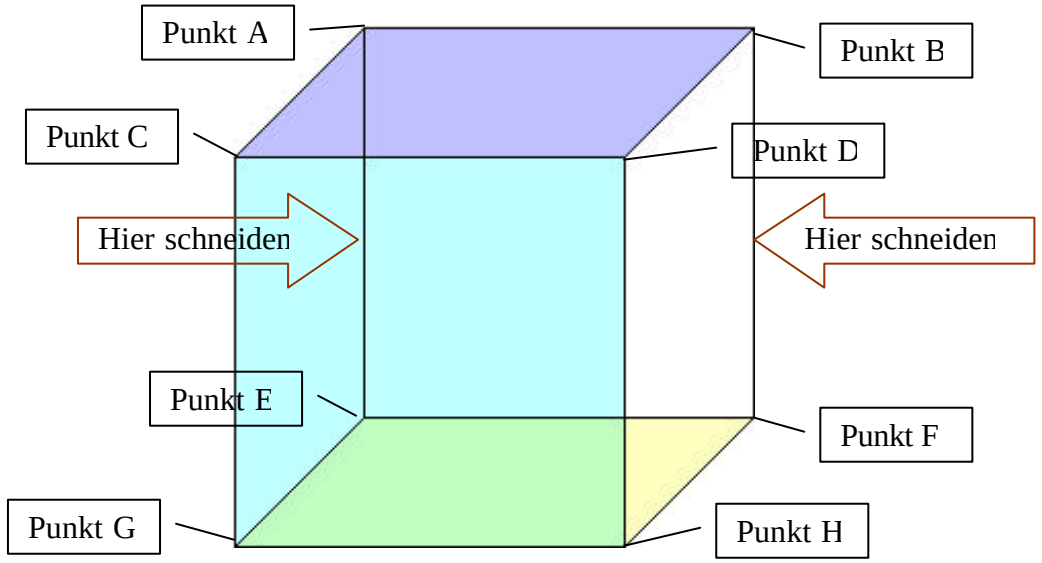


Ein anderes Beispiel:

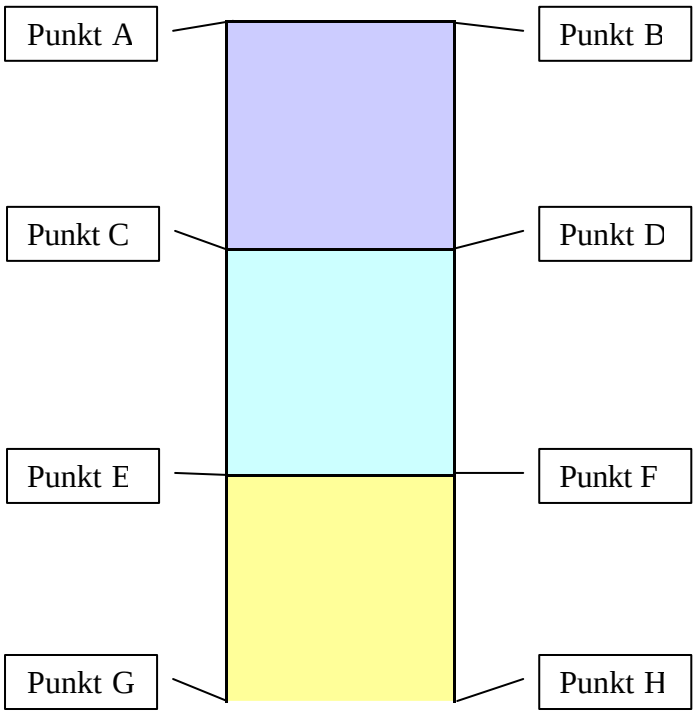


Hier wären jetzt die Werte der Funktion immer dann 1, wenn auch X gleich 1 ist, also f(x,y,z)= x.

Aber was hat das mit unseren KARNAUGH – MAPS zu tun? Nun wir können unseren Würfel nehmen und ihn „aufschneiden“:

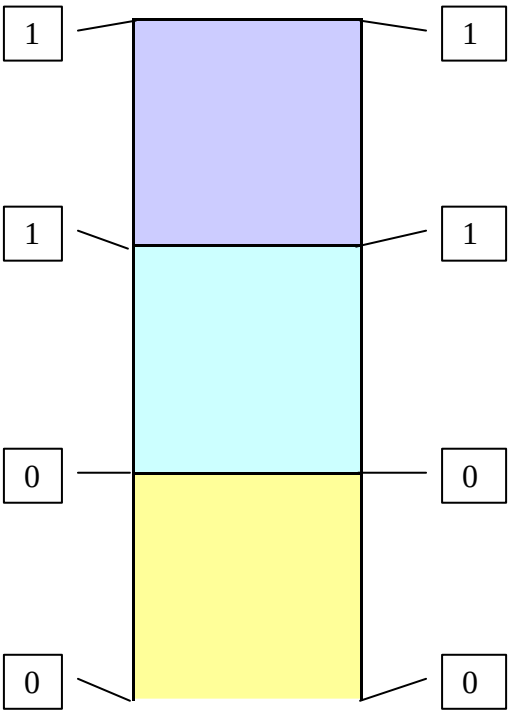


wenn wir das ausbreiten sieht das so aus (die durchgeschnittene Seite ist unwichtig, es kommt uns nur auf die Ecken an)

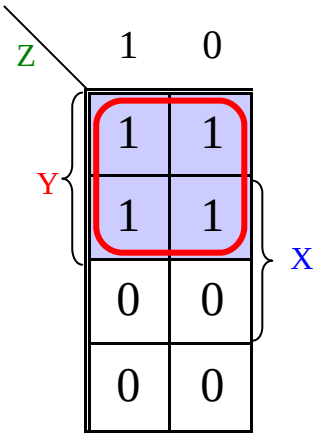


Nun, wenn wir nun statt eines Punktes eine 1 schreiben und für den fall, dass ein Punkt an einer Ecke ist, dann haben wir auch schon unsere KARNAUGH – MAP.

In unserem ersten Beispiel auf Seite 26 (Abhängigkeiten nur mit Y) sähe das dann so aus:



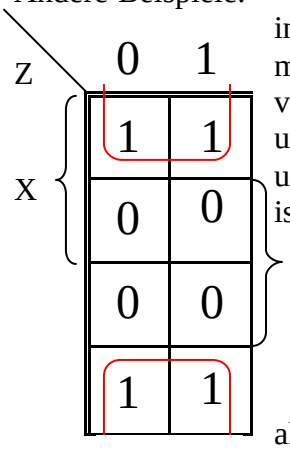
Das ist nicht anderes als



An den Seiten sind die Bereiche Angezeichnet, an denen der jeweilige Wert auf 1 gesetzt ist. Man muss in KARNAUGH – MAPS immer nach Rechtecken suchen. In diesem Beispiel ist das recht einfach. Nachdem wir das rechteck gefunden haben, ist es recht einfach die Formel herzuleiten.

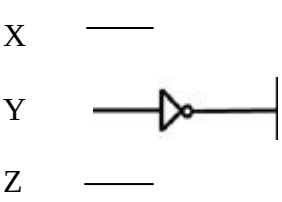
Es reicht nämlich sich nun zu überlegen, welche Eigenschaften die Einsen in diesem Rechteck erfüllen, in unserem Beispiel: Für alle Einsen gilt Y = 1, unsere Formel ist daher f(x,y,z)= x.

Andere Beispiele:

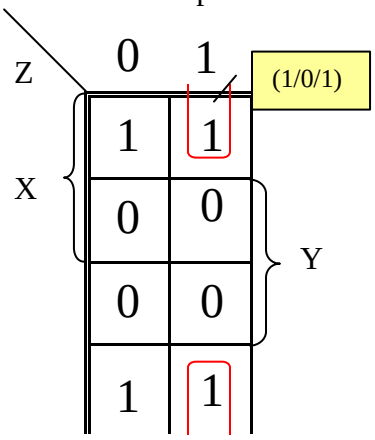


in diesem Beispiel haben wir auch ein Rechteck, da man die obere Seite mit der unteren Seite, sowie die rechte Seite mit der linken Seite verbinden kann (man danke an den Würfel, den kann man ja auch drehen und wenden wie man will, man muss nur darauf achten, dass die Nullen und Einsen mitkommen). Die gemeinsame Eigenschaft dieses Rechteckes ist Y ist 0. daraus ergibt sich f(x,y,z) = $\overline{y}$.

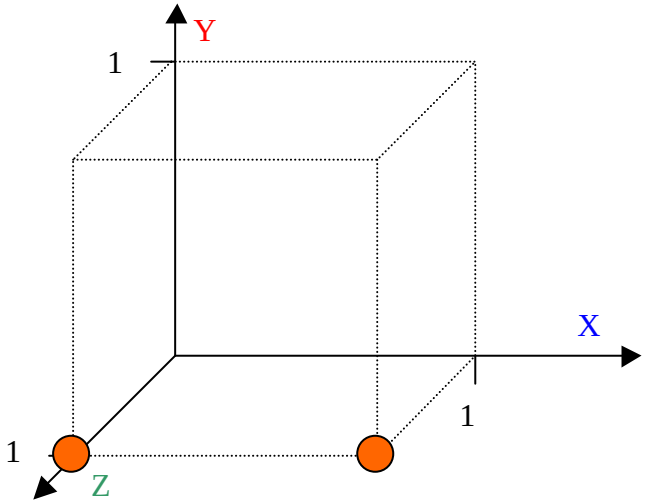
als Schaltung:



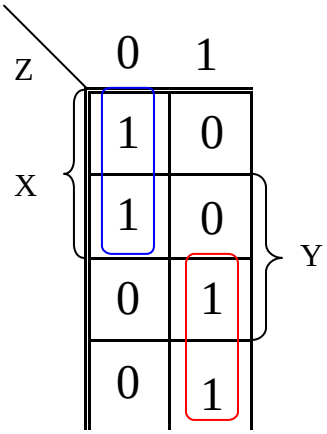
Noch ein Beispiel



Hier ist die gemeinsame Eigenschaft, dass Z = 1 und Y = 0 ist. Für die Formel ergibt sich daher f(x,y,z) = z * $\overline{y}$. Schreiben wir an die Ecken die Koordinaten können wir (zur Veranschaulichung) auch wieder unseren Würfel zur Hand nehmen.

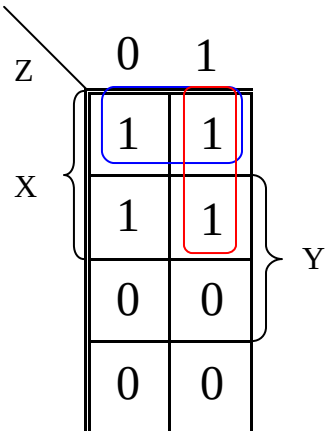


Man kann in KARNAUGH – MAPS auch zwei Rechtecke finden und dann nach gewohnter Weise verfahren:



Für das blaue Rechteck gilt:
X * $\overline{Z}$
Für das blaue Rechteck gilt:
Z * $\overline{X}$
Die Kombination aus beiden ergibt:
f(x,y,z) = X * $\overline{Z}$ + Z * $\overline{X}$

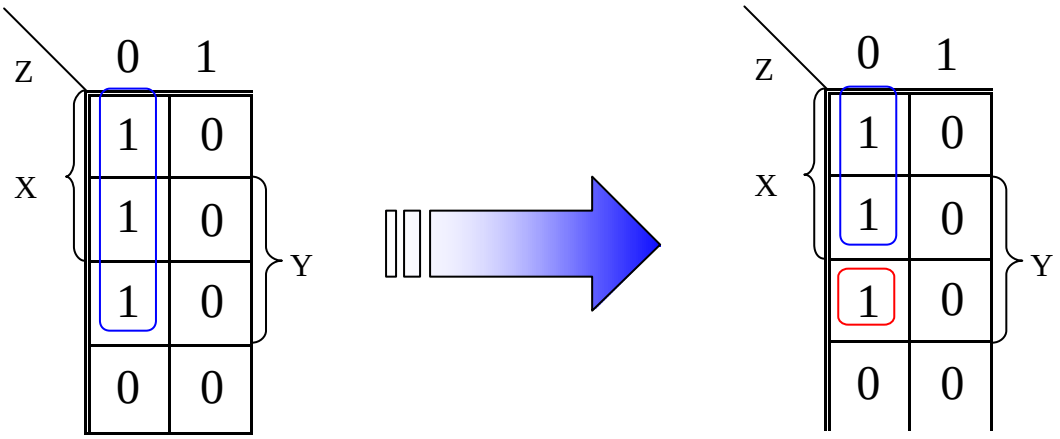
Die Rechtecke dürfen sich auch überschneiden, selbst wenn manche Einsen dann doppelt gezählt werden:



Für das blaue Rechteck gilt:
X * $\overline{Y}$
Für das blaue Rechteck gilt:
Z * $\overline{X}$
Die Kombination aus beiden ergibt:
f(x,y,z) = X * $\overline{Y}$ + Z * $\overline{X}$

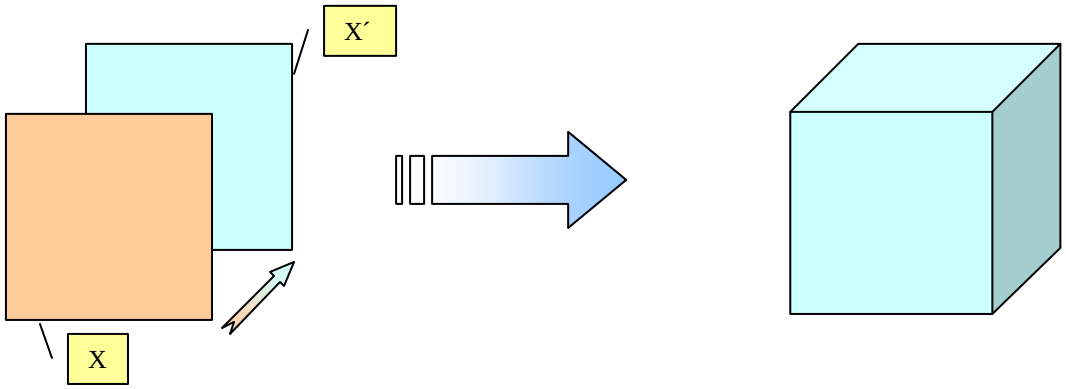
Für diese Art der Darstellung gilt : es gibt so viele Terme wie es Rechtecke in der Map gibt.

Es gibt aber auch Sonderfälle in dem man Rechtecke in kleinere Rechtecke unterteilen muss:

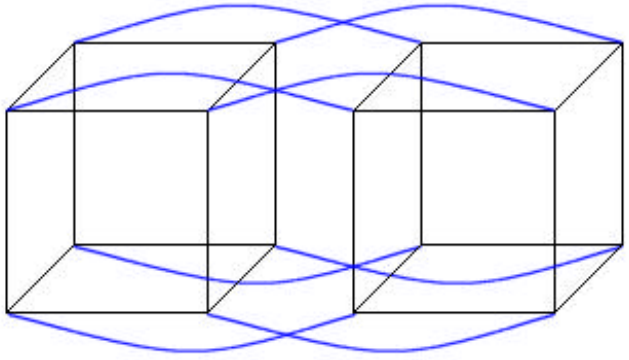


Das liegt daran, dass man diese drei Einsen nicht mit gemeinsamen Bedingungen umschreiben kann, dafür braucht man zwei Rechtecke (auch Rechtecke mit nur einem Element sind zulässig). Man kann prinzipiell sagen, dass sich bei diesen Darstellungen nur Rechtecke mit geraden Anzahl von Einsen darstellen lassen.

Ein Würfel schein ja ganz angebracht zu sein für drei Elemente, es gibt 3 Achsen, jede Achse jeder wird ein Element zugeordnet. Aber was machen wir, wenn wir eine Funktion mit 4 Elementen haben? Klar, wir brauchen einen 4D Würfel. Ich gestehe, das klingt ein wenig ungewöhnlich aber mit einer Menge gutem Willens kriegt man das auch hin.
Zunächst wie kann man aus einem Quadrat ein Würfel machen? Man nehme ein Quadrat und verschiebe es und verbinde die Ecken von dem ursprünglichem Quadrat und dem verschobenen Quadrat:

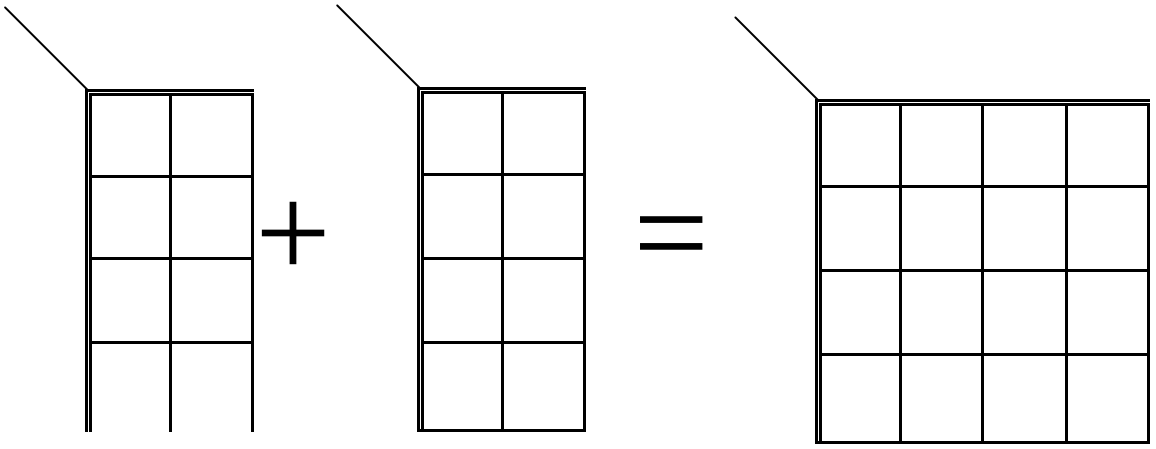


Nun, so ähnlich ist das auch mit dem 4D Würfel, man nehme einen Würfel, verschiebe ihn und verbinde die Ecken (klingt doch einfach, oder?):

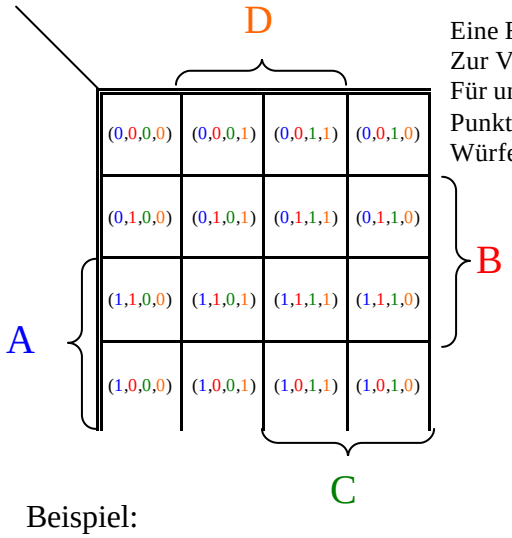


Tja, sieht zwar gewöhnungsbedürftig aus, ist aber richtig: von einem Würfel in der Dimension n gehen von jeder Ecke n Kanten aus. (Quadrat → 2, Würfel → 3, 4DWürfel → 4, …)

Die dazugehörige KARNAUGH – MAP sieht dementsprechend so aus:

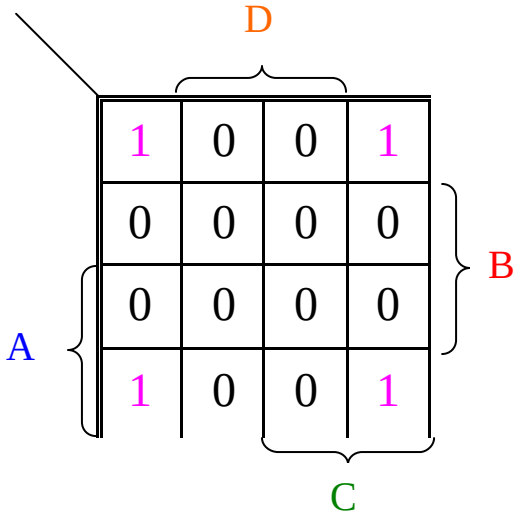


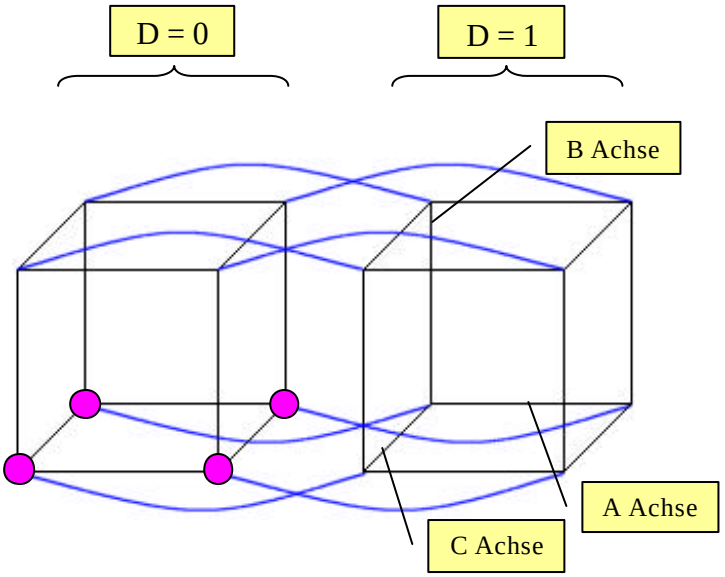
Für die horizontalen Werte bleibt alles gleich, aber wir tragen Z nun nicht mehr als 0 oder 1 über die Map, sondern geben Z einen eigenen Bereich, da es sich sie vertikalen Bereiche (also die Spalten) mit dem neuen vierten Wert D teilen muss.



Eine Funktion f(a,b,c,d) sähe dann so aus:
Zur Veranschaulichung tragen wir die Koordinaten in jedes Feld.
Für unseren 4D Würfel bedeutet dass, sollte D =1 sein tragen wir unseren Punkt im rechten Würfel ein, sollte D = 0 sein tragenn wir ihn am linken Würfel ein.

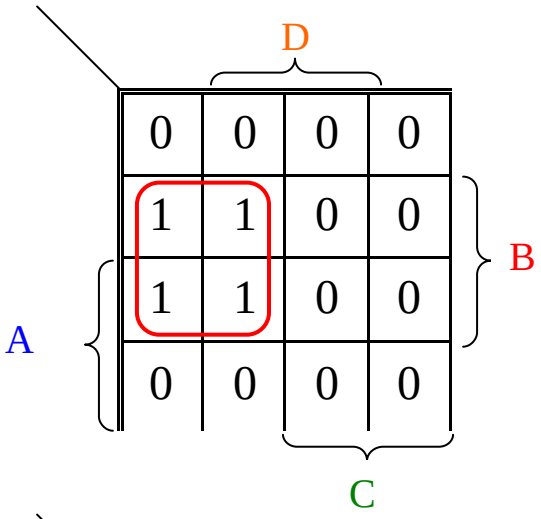
Beispiel:

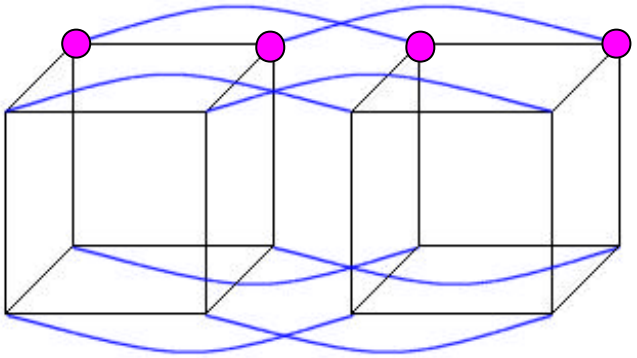




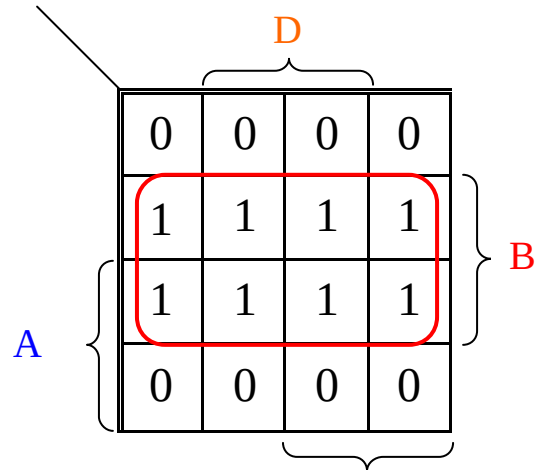
Auch mit vier Werten gilt: man kann die rechte Seite mit der linken Seite verbinden und die obere Seite mit der unteren Seite verbinden, daher liegen auch diese in einer Ebene und haben zwei Gemeinsamkeiten, die ihre Lage voll umschreibt: nicht D und nicht B, also
f(A,B,C,D) = $\overline{B}$ * $\overline{D}$

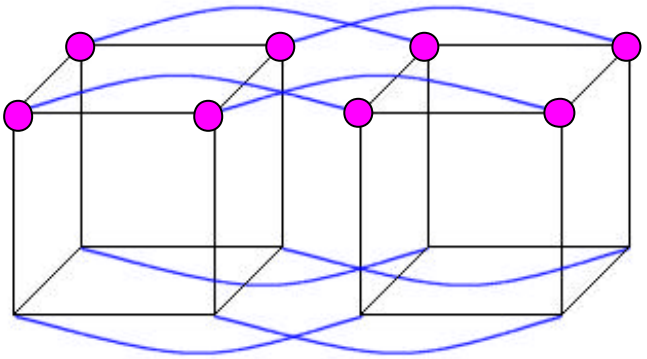
Weitere Bespiele um sich daran zu gewöhnen:



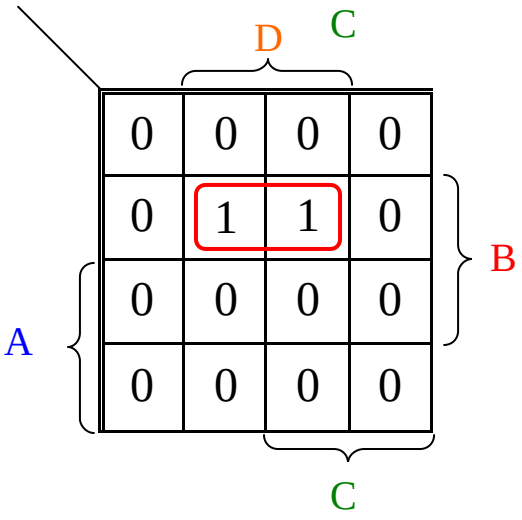


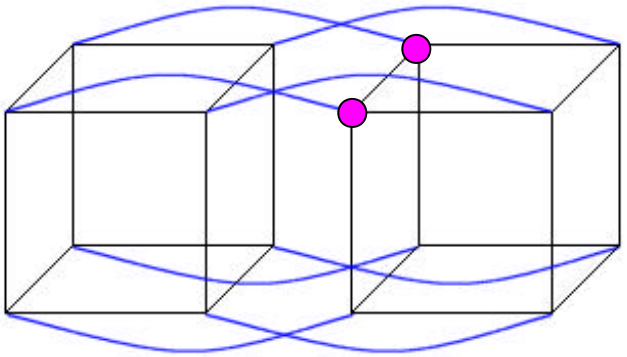
$f(A,B,C,D) = B * \overline{C}$



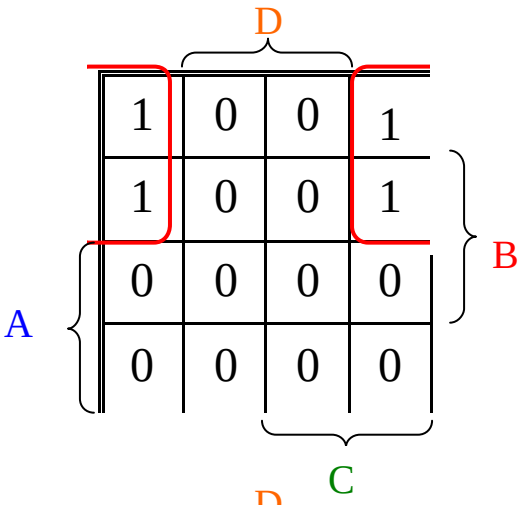


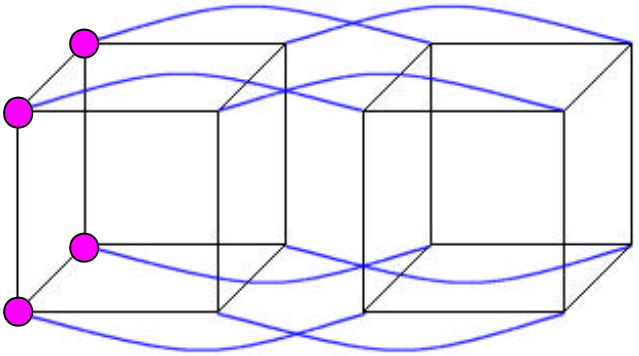
$f(A,B,C,D) = B$



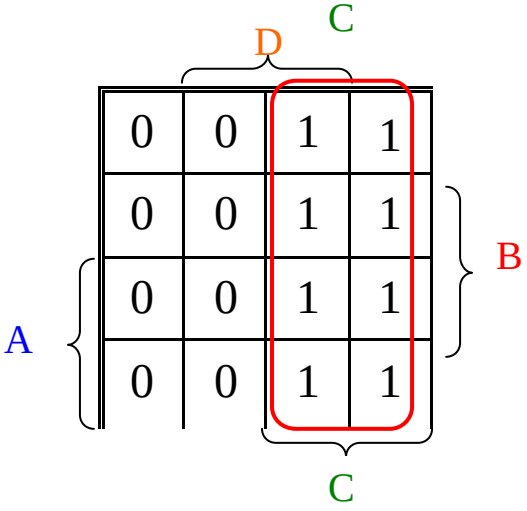


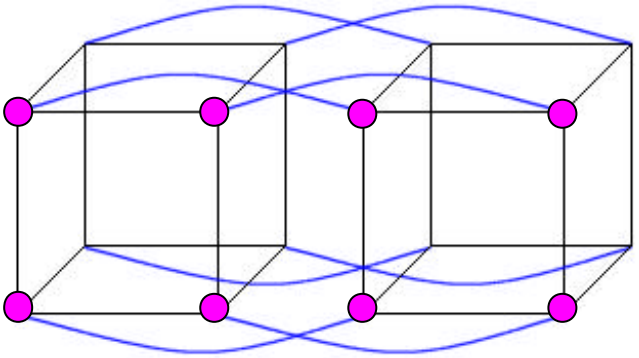
$f(A,B,C,D) = \overline{A} * B * D$



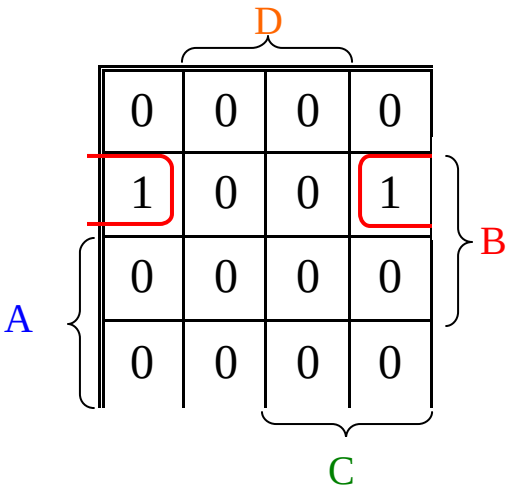


f(A,B,C,D) = $\overline{A} * \overline{D}$

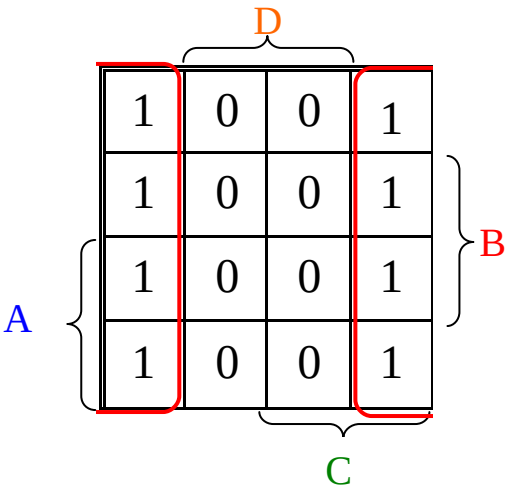




f(A,B,C,D) = C

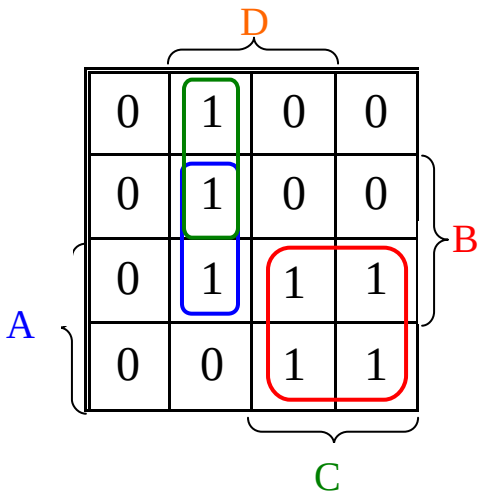


f(A,B,C,D) = $\overline{A} * B * \overline{D}$



f(A,B,C,D) = $\overline{D}$

Auch hier ist es erlaubt und oftmals auch nötig Rechtecke zu bilden, die sich teilweise überschneiden, hierzu ein Beispiel:



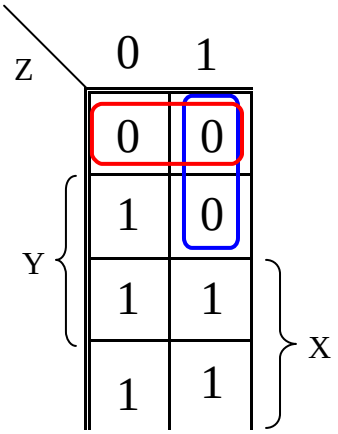
Für das grüne Rechteck:
$\overline{A} * D * \overline{C}$

Für das blaue Rechteck:
$B * D * \overline{C}$

Für das rote Rechteck:
$A * C$

zusammen macht das:
$f(A,B,C,D) = \overline{A} * D * \overline{C} + B * D * \overline{C} + A * C$

Die Art mit KARNAUGH – MAPS umzugehen ist dann recht effizient, wenn wir wenige Einsen haben und diese in wenige, eindeutig umschreibbare „Päcken“ zerlegen können. Eine Funktion, die aus fast nur Einsen besteht ist damit aber nicht effizient beschreibbar. Hier ist es sinnvoller sich die Nullen zu nehmen und diese in eindeutig umschreibbare „Päckchen“ zu unterteilen:



Für das blaue Rechteck:
$\overline{X} * Z$ ist das gleiche wie $X + \overline{Z}$ (nach Regel 14 Seite 24)

Für das rote Rechteck:
$\overline{X} * \overline{Y}$ ist das gleiche wie $X + Y$ (nach Regel 14 Seite 24)

zusammen macht das:
$f(X,Y,Z) = (X+Y)*(X+\overline{Z})$

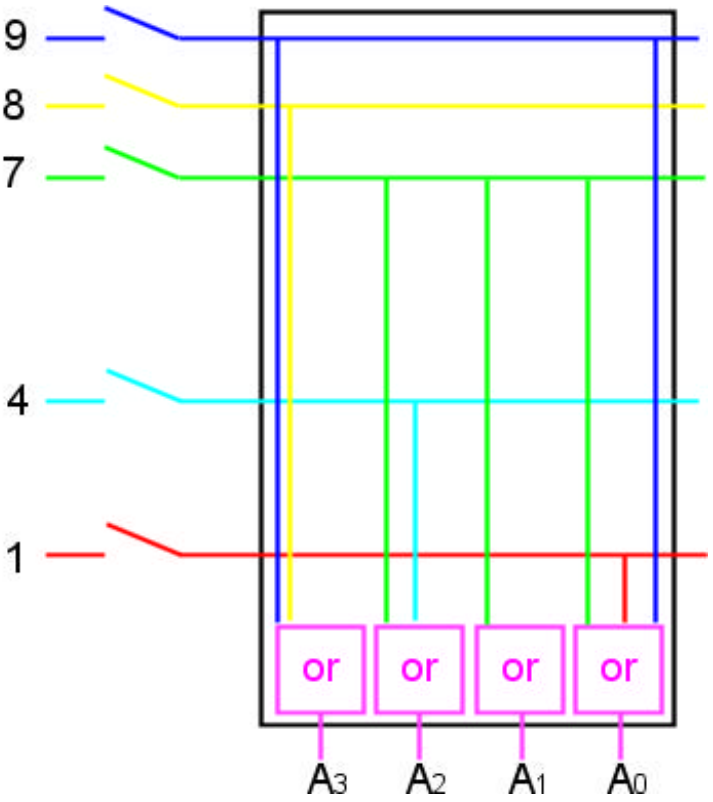
Aufzeichnung der Vorlesung vom 24.11.2000

Nach diesen eher abstrakten Exkursion in die Welten der logischen Gatter und der Funktionen beginnen wir nun die wesentlichsten Elemente eines Computers nachzubauen. Diese entscheidenden Elemente wären: 1.) Encoder 2.) Decoder 3.) Multiplexer und schließlich noch ein 4.) Addierer.

1.) ein Encoder:

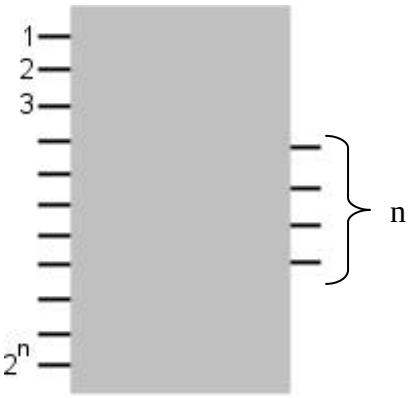
Ein Encoder ist nichts anderes als ein Übersetzer, eine Tastatur zum Beispiel. Der Encoder „übersetzt“ der Taste (z.B. „5“ oder „8“) in Binärcode. Nehmen wir z.B. die Zahlen auf einem Taschenrechner.

Der Encoder besteht aus einzelnen Leitungen für jede einzige zu interpretierende Taste eine separate, die alle am unteren Ende mit einem „or“ Gatter verbunden sind. Dieses „or“ Gatter verhindert einen Kurzschluss in dem fall, dass mehrere Tasten zur gleichen Zeit gedrückt werden.



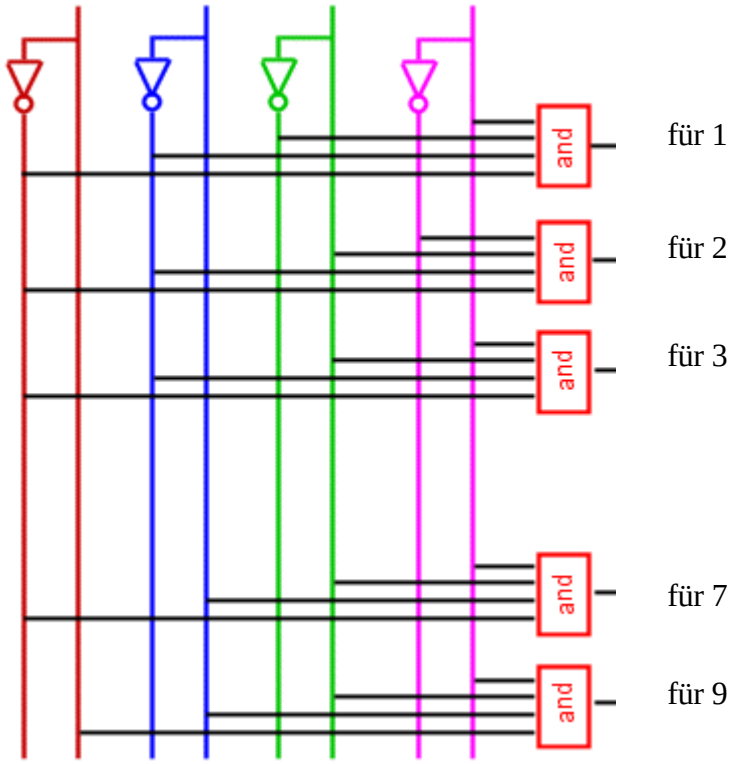
für 9	1	0	0	1
für 8	1	0	0	0
für 7	0	1	1	1
für 4	0	1	0	0
für 1	0	0	0	1

Ein Encoder sieht vereinfacht so aus:

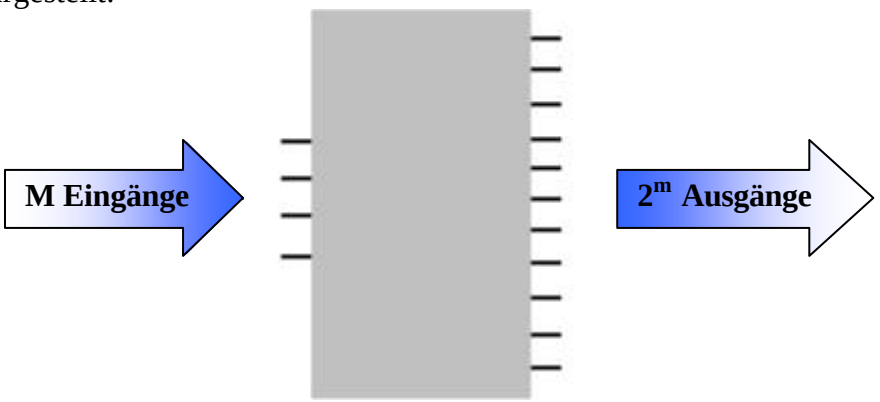


2.) ein Decoder

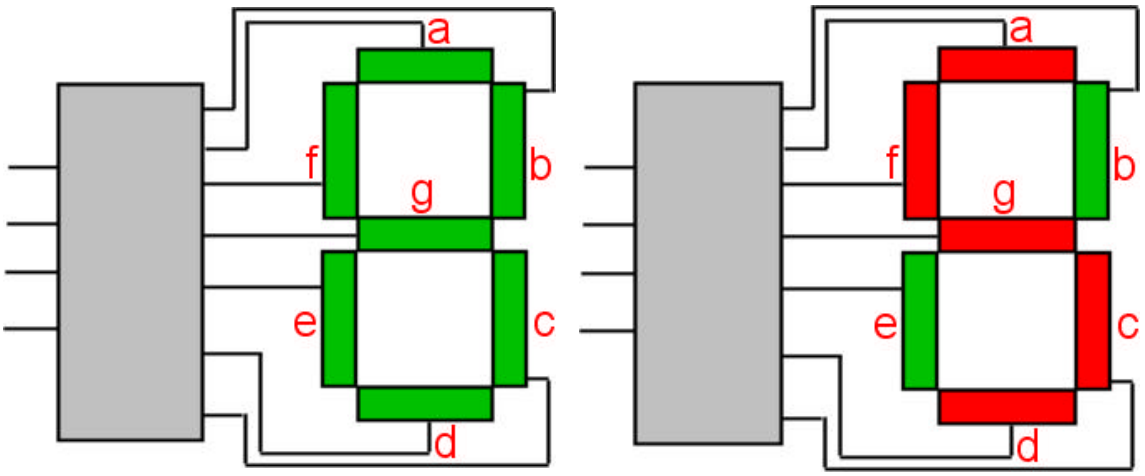
Ein Decoder interpretiert eine Schaltung und gibt je nach Eingabe eine entsprechende Ausgabe. In unserem Taschenrechnerbeispiel wäre das eine Stelle des Displays, das je nach Situation ein entsprechendes Zeichen auf dem Bildschirm ausgibt.



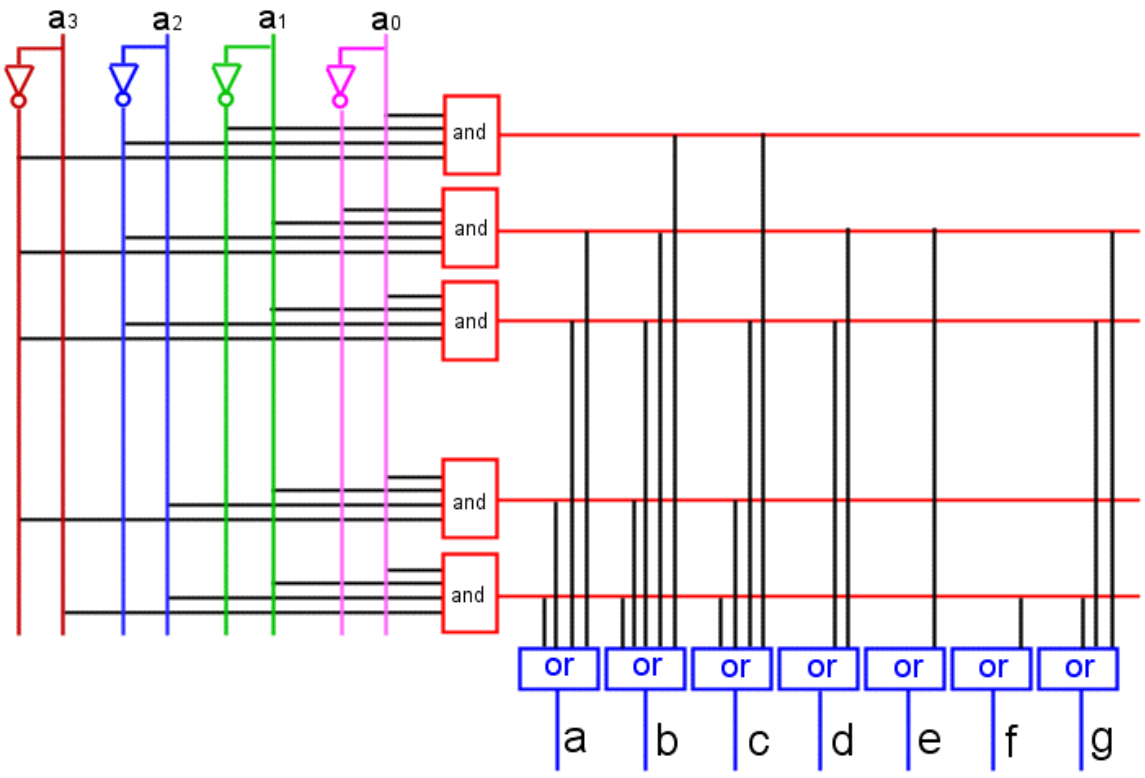
vereinfach dargestellt:



So oder so ähnlich könnte also ein Decoder für eine Stelle unseres Taschenrechners aussehen. Die grünen Leuchtstäbe sind (für uns) mit Namen von „a“ bis „g“ versehen und je nach dem welches Zeichen dargestellt werden soll, werden bestimmte Leuchtstäbe aktiviert.

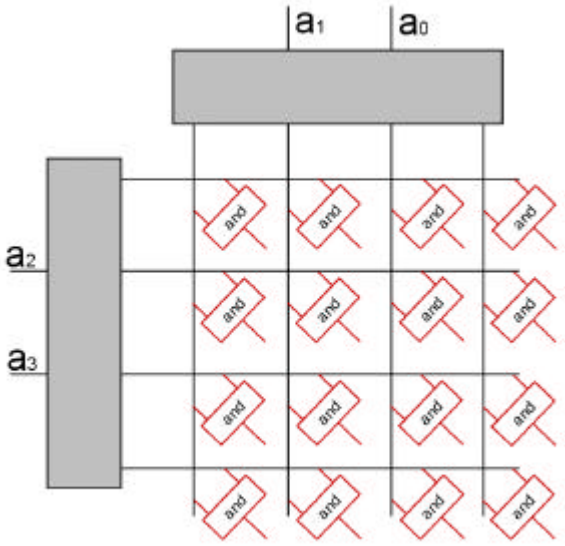


Für den Fall, dass wir z.B. eine „5“ darstellen wollten müssten wir a, f, g, c, d aufleuchten lassen. Und wie machen wir das? Nun, wir nehmen unseren Decoder auf Seite 36 und führen ihn weiter:

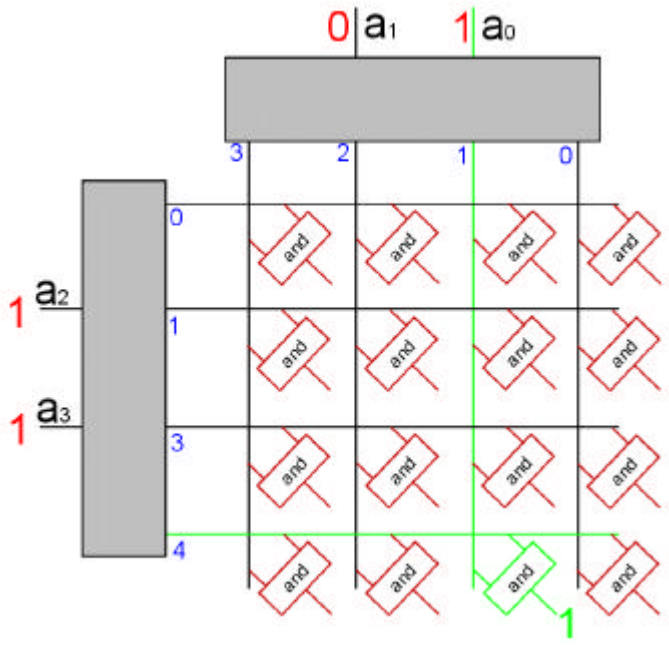


Eigentlich recht simpel so ein Display (etwas länger scharf hinsehen und man komt dahinter, wie er funktioniert).

Größere Decoder aus mehreren kleinen basteln:
Angenommen wir haben mehrere „2 zu 4“ Decoder (2 Eingänge und 4 Ausgänge) vorrätig, wollen aber einen 4 zu 16 Decoder haben. Da unser Chiplieferant gerade keine mehr auf Lager hat, basteln wir uns aus zweien unserer „2 zu 4“ Decodern einen „4 zu 16“ Decoder und das geht wie folgt:



Das sieht ja ganz nett aus, aber was soll das? Nun, schauen wir es uns mal näher an: Der hergestellte Decoder hat 4 Eingänge (a_0, a_1, a_2, a_3) und 16 Ausgänge (die Ausgänge an den AND- Gattern). Jedes Gatter an einer „Kreuzung“ kann nur dann 1 ausgeben, wenn beide Leitungen der „Kreuzung“ 1 sind. Da die beiden „2 zu 4“ Decoder jeweils nur eine Leitung auf 1 setzten können, kann immer nur (max.) ein AND- Gatter 1 augeben. Machen wir mal ein Beispiel mit a_0=1 a_1=0 a_2 = 1 a_3 = 1



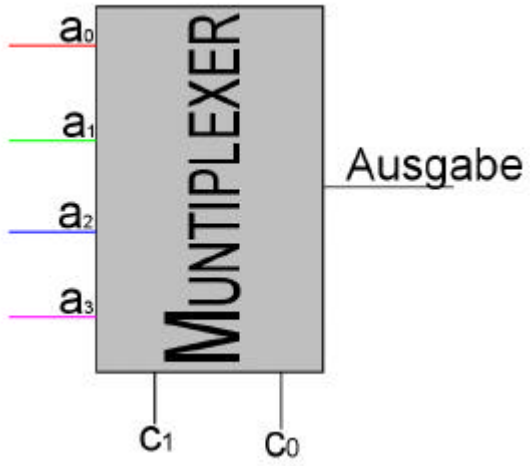
Die blauen Zahlen verdeutlichen hier die Resultate, die ein Gatter für sich alleine produzieren würde. Nur das **grüne AND Gatter** ergibt 1 bei den Werten 1101. Die andern sind demnach 0. Das grüne Gatter ist demnach die „13“ (wegen der Binärumrechnung $1101_2 = 13_{10}$ siehe Seiten 2 bis 10). Man kann sich jetzt für alle anderen Werte überlegen, für welche Werte sie stehen. Das ganz rechts oben steht z.B. für die „0“ das links unten für „15“.

Wir haben also einen Decoder mit einem „fan–in“ (Anz. der Eingänge) von 4 und 16 ausgehenden Werten (das die mir aber nicht zu lange weg bleiben!).

3.) ein Multiplexer

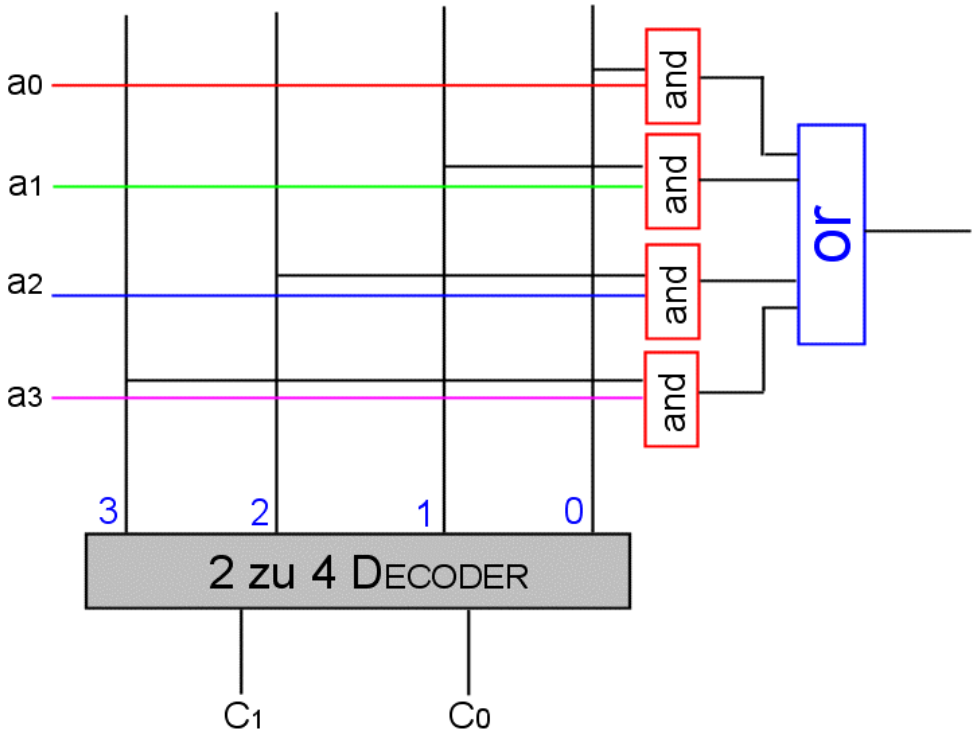
ein Multiplexer ist so was wie ein Türsteher- Gatter, dass von den ankommenden Werten nur einen weiter lässt (je nach dem welche Instruktionen er hat).

vereinfacht:



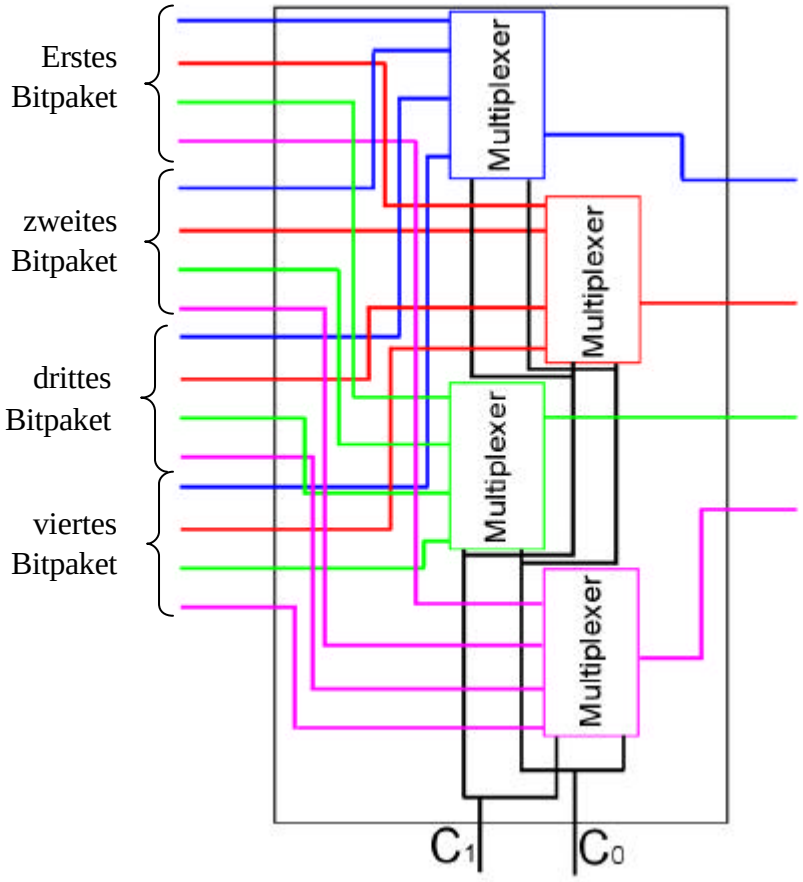
Je nach dem welche Instruktionen (gegeben durch c_1 und c_2 das sogenannte select) der Muntiplexer bekommt, leitet er entweder den Wert von a_0 a_1 a_2 a_3 weiter.

Betrachten wir das auf der Gatter-Ebene:



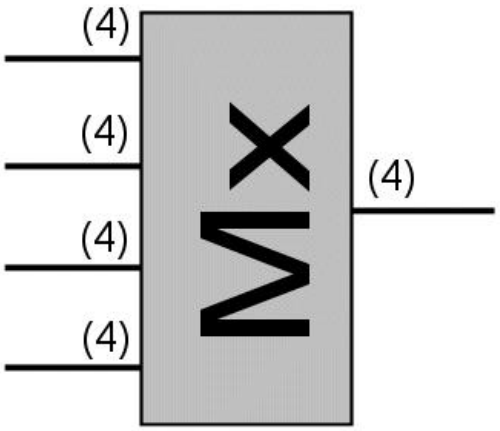
Eines der 4 and Gatter am rechten Ende der Schaltung geben immer dann den Wert weiter, wenn der Decoder eine der Leitungen auf 1 gesetzt hat (siehe blaue Zahlen), die Werte fasst das or zusammen. Würde z.B. c_1 =1 und c_2=0 angelegt würde der Wert von a_2 (blau) weitergereicht.

Man kann auch einen Multiplexer für 4 x 4 Eingangswerte und 4 Ausgangswerte entwickeln:



je nach dem welches Datenpaket wir durchlassen wollen, müssen mir an c_1 und c_0 nur entsprechende Werte anlegen. Wollen wir z.B. das zweite Datenpaket durchlassen, dann legen wir an $C_1 = 0$ und $C_0 = 1$ an, die Multiplexer im inneren merken dann, dass sie immer den zweiten Wert durchlassen müssen (das n jedem Multiplexer eingehende unseres Datenpaketes). Hier sieht man wieder, dass man größere Multiplexer durch mehrere kleinere gleicher Gattung nachbauen kann.

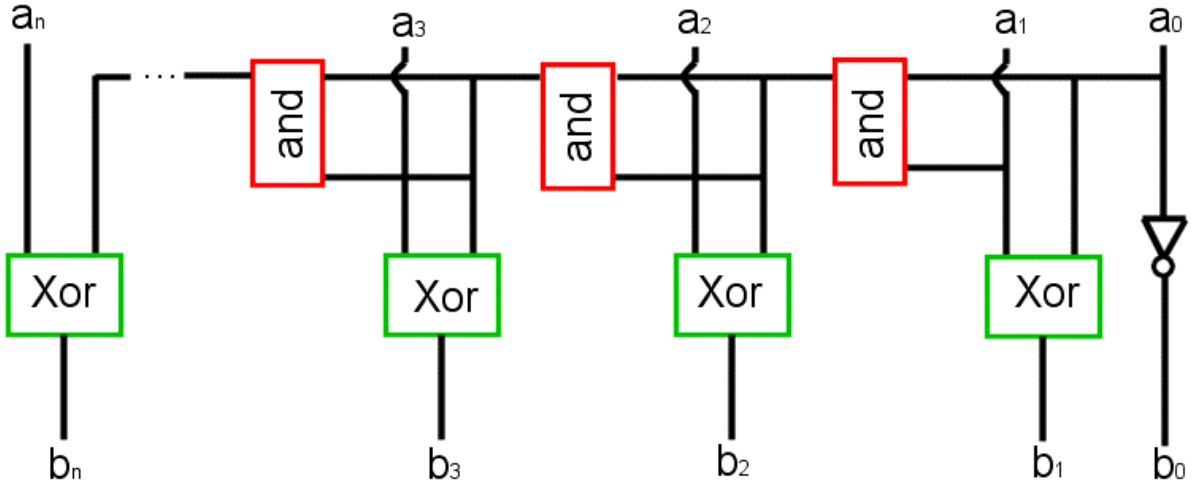
Um nicht immer 4 mal 4 Eingänge zeichnen zu müssen, kann man auch vereinbaren, dass folgende Schaltung der oberen entspricht:



4) zu guter letzt ein Addierer

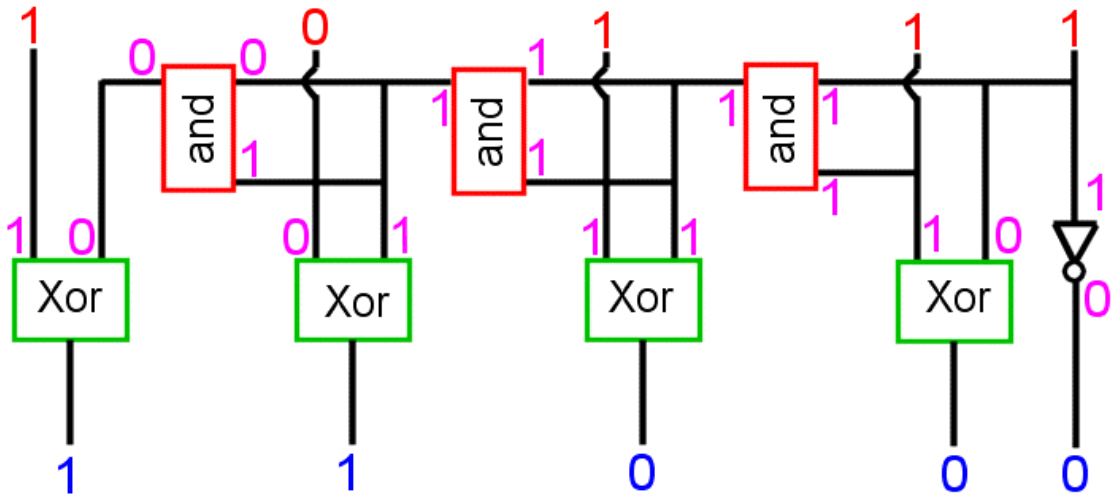
(auch Incrementer genannt)
dieser Addierer ist kein Addierer im eigentlichen Sinn, denn er kann nicht mehr als eine eingegebene Bitkette um eins zu erhöhen.

Als Schaltung sieht er so aus:



Zunächst wird das letzt Bit a_0 gekippt (wenn man eine 1 addiert wird eine gerade Zahl immer ungerade und eine ungerade Zahl immer gerade). Wenn die Zahl eine 0 als a_0 hatte sind wir eigentlich schon fertig, für den Fall, dass $a_0 = 1$ haben wir aber einen Übertrag, den wir zusammen mit dem a_1 durch das XOR zu b_1 machen. Gab das wieder einen Übertrag, (wenn $a_0 = 1$ und $a_1 = 1$), so wird er durch das AND- Gatter weitergegeben. Das setzt sich weiter so fort, bis der letzte Übertrag mit a_n verrechnet wird zu b_n.

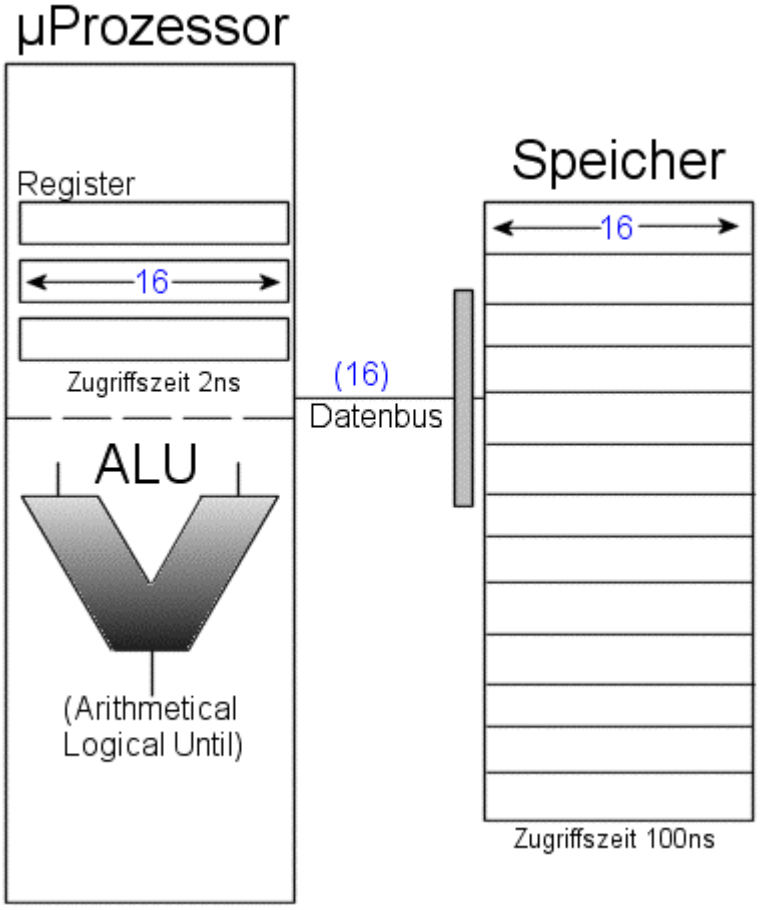
Beispiel:
Wir erhöhen 1 0 1 1 1 um eins



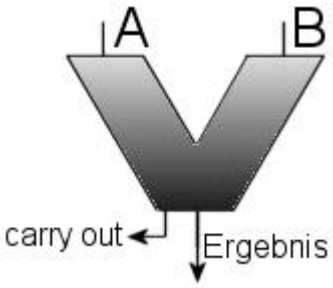
und raus kommt: 1 1 0 0 0, faszinierend, es stimmt! (kann man auch mit allen anderen Werten mal ausprobieren wenn man Zeit & Spaß hat gibt nur bei 1 1 1 1 1 Probleme)

Aufzeichnung der Vorlesung vom 1.12.2000

Nach aller Theorie versuchen wir jetzt mal „wirklich“ Teile eines real existierenden Computers nachzubauen. Aber wie sieht so’n Ding aus? Vereinfacht könnte man einen Computer, besser den Speicher und die CPU (Central Processing Unit) so darstellen:

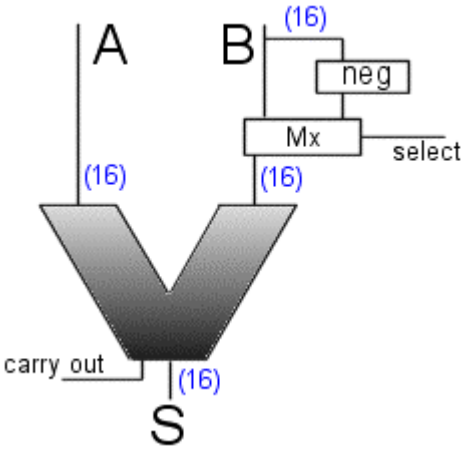


man Kann drei Komponenten erkennen: zum einen den µProzessor, den Speicher und den Datenbus, der die beiden verbindet. Der Speicher besteht in diesem Beispiel aus Registern mit der Wortlänge 16 Bit, der Prozessor hat ebenfalls einige Register der Wortlänge 16 Bit und damit das Ganze auch Sinn macht: einen Datenbus der 16 Bits von Speicher nach Prozessor und andersherum transportieren kann.
Doch was ist die ALU? Die ALU (Arithmetical Logical Unit) ist ein Bestandteil unseres Prozessors und ist verantwortlich für die (schwer zu raten) arithmetischen Operationen auf unsere Binärzahlen (in der Zweierkomplementdarstellung siehe → Seite 9).

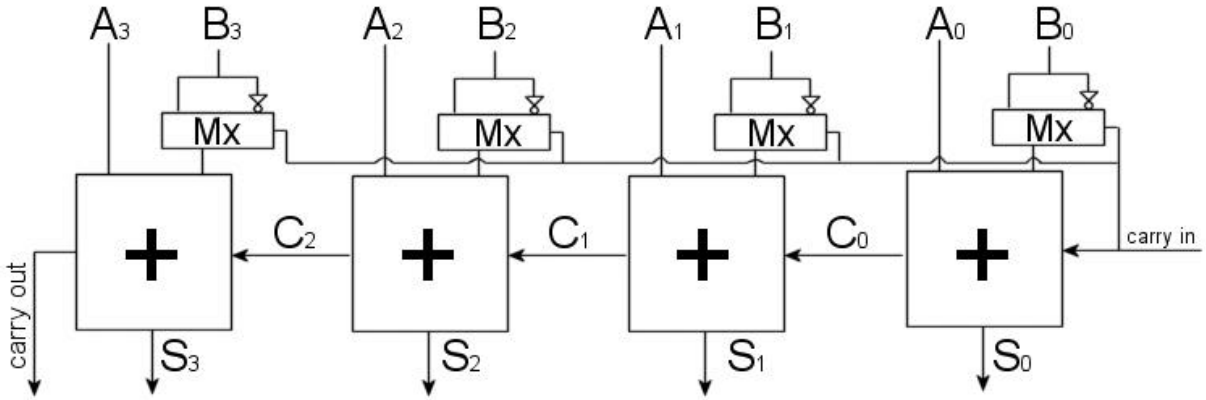


Unsere ALU bekommt zwei 16 Bit Argumente und errechnet uns das Ergebnis (je nach dem was wir ihm „sagen“ was er rechen soll). Nachdem wir auf Seite 41 gesehen haben, wie man eine Zahl um eins erhöht, wäre es sicherlich auch noch ganz interessant **zwei** Zahlen addieren

zu können. Der dazu benötigte „Volladdierer“ kann aber noch mehr: subtrahieren. Praktisch 2 in einem! Aber warum geht das? Nun, wie wir zum Thema Zweierkomplementdarstellung (Subtraktion zweier Binärzahlen) gesehen haben, kann man eine Binärzahl (z.B. B) von einer anderen Binärzahl (z.B. A) subtrahieren, indem man die entsprechende Zweierkomplementzahl von B zu A addiert. Und wie machen wir das? Dazu müssen wir von der Zahl B das bitweise Komplement bilden und schließlich 1 hinzuaddieren.
Wir müssen unserem Volladdierer also „sagen“ (per Multiplexer) ob wir addieren oder subtrahieren wollen und er nimmt je nach dem was Ihm gesagt wurde entweder A und B und addiert sie schlicht oder er nimmt A und die Zweierkomplementdarstellung von B und addiert diese.

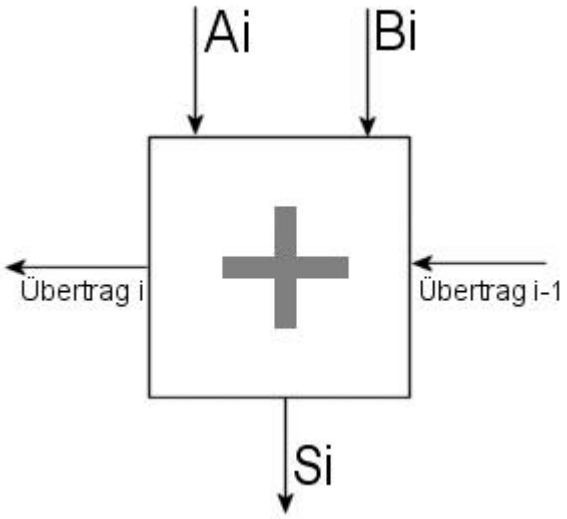


Hmm, dieser Versuch ist leider etwas unglücklich, weil der Baustein „neg“ (oben rechts) bereits addieren muss (nämlich unsere 1 am ende der Negation). Also versuchen wir es anders:



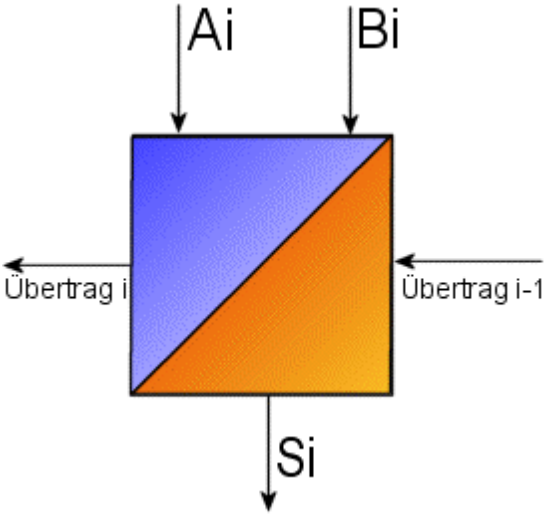
Das sieht schlimmer aus als es ist! Von rechts nach links: das „carry in“ ist die Stelle, an der wir dem Computer „sagen“ ob wir addieren (0) oder subtrahieren (1) wollen.
Fürs addieren: wir geben dem ersten Baustein eine 0 als übertrag und der Multiplexer unter dem B_0 verändert den Wert von B_0 nicht, sondern gibt ihn an das Addiermodul weiter. Dieses errechnet aus A_0 und B_0 das erste Bit S_0 des Ergebnisses und gibt ein Übertrag C_0 weiter, sollte ein Übertrag bei der Addition von A_0 und B_0 entstanden sein. Das geht so weiter bis S_3 (in diesem Beispiel haben wir nämlich einen 4 Bit- Addierer). Das letzte Addiermodul liefert die Information, ob wir über die darstellbaren Zahlen hinausgekommen sind und gibt uns für diesen Fall eine 1 als „carry out“. Wie solch ein Addiermodul aufgebaut ist, wird gleich erläutert, zunächst noch das Subtrahieren.
Fürs Subtrahieren: wenn wir subtrahieren wollen, geben wir eine 1 als „carry in“ in den Volladdierer hinein. Diese 1 ist sozusagen ein Übertrag den wir gleich zu Beginn der Addition mit in den Volladdierer hineingeben, es ist die 1, die wir nach dem bitweisen Komplement

von B hinzuaddieren mussten. Die Multiplexer über dem Addiermodul bekommen jetzt die „Anweisung“ nicht mehr die nicht das Bit B_X durchzulassen, sondern das Inverse Bit $\overline{B_X}$. Wenn wir es schaffen dieses Addiermodul richtig zu bauen, sind wir auch schon fertig, denn abgesehen von dem Komplement von B und der Addition von 1 sind Addition und Subtraktion identisch.
Und das Addiermodul?

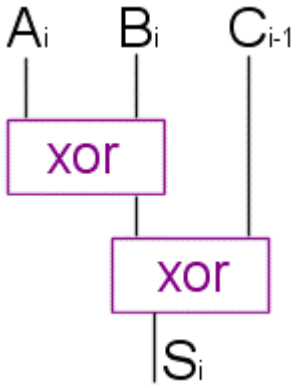


Unser Addiermodul soll folgendes können: zum einen die Werte A_i, B_i und Übertrag $_{i-1}$ addieren zu S_i und zum anderen einen Übertrag $_i$ ausgeben, sollte es einen bei der Addition geben.

Ein Addiermodul besteht also aus zwei Komponenten



Zur Addition:

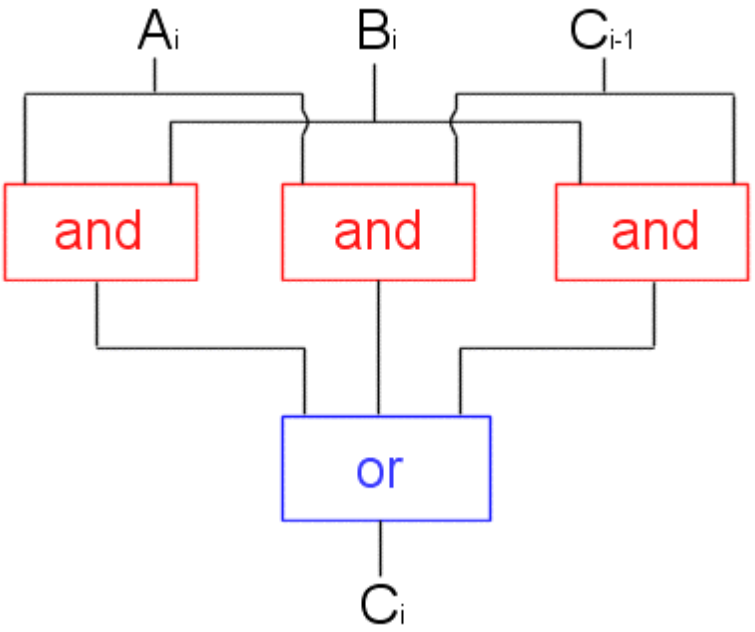


Folgende Wertetabelle ergibt sich:

A_i	B_i	C_{i-1}	S_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

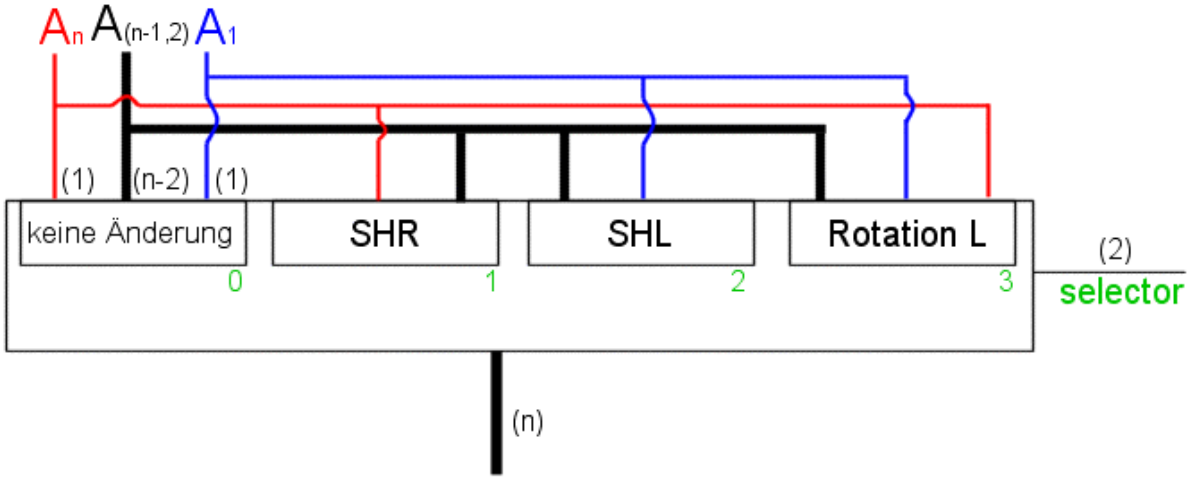
Wenn man ganz scharf hinschaut, erkennt man, dass es immer bei einer ungeraden Parität (die Anzahl der Eingabeeinsen ist ungerade) eine 1 im Ergebnis gibt. Nun das können wir mit Hilfe der Xor- Gatter prüfen und haben so immer ein richtiges Ergebnis für die Addition von A_i, B_i und C_{i-1} (das ist der Übertrag von Voraddiermodul).

Zum Errechnen des Übertrages:



Ein Übertrag ergibt sich immer dann, wenn mindestens 2 der Eingabeweite A_i B_i oder C_{i-1} eins sind. Die drei And- Gatter prüfen genau das. Der Fall, dass alle drei Eingabewerte 1 sind ist auch mit dieser Schaltung abgedeckt und ergibt sich ebenfalls aus den And-Gattern. Wenn zumindest ein And-Gatter 1 ergibt ist auch $C_i = 1$. Tada: unser Volladdierer (der auch subtrahieren kann) ist fertig!

Nachdem wir nun unserer ALU „beigebracht“ haben, wie man addiert und subtrahiert, wäre es nicht schlecht, könnte er auch multiplizieren uns dividieren! Wie wir (hoffentlich) bereits wissen ist eine Multiplikation mit 2 nichts anderes als ein „Shift Left“ und eine Division durch 2 ein „Shift Right“ (siehe → Seite 6). Eine Schaltung, die sowohl Multiplizieren, Dividieren als auch eine „Rotaion Left“ durchführen kann sähe so aus:

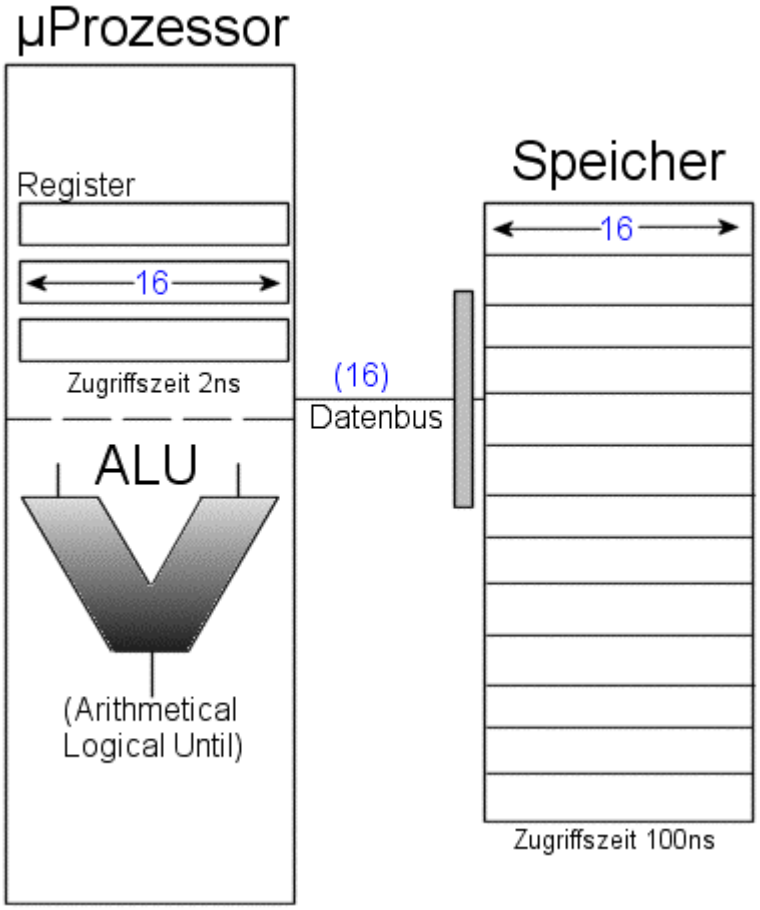


Es handelt sich hier um einen „4 mal n Multiplexer“, der 4 mal n Eingänge und n Ausgänge hat. Ja nach dem was an den „Selector“ gelegt wird, bleiben die Eingabe unverändert (für 0), wird mit 2 multipliziert (für 2), wird durch 2 dividiert (für 1), oder (für 3) ein „Rotation Left“ (dessen Bedeutung uns für diese Vorlesung noch verborgen bleibt) durchgeführt. Zum Verständnis der Schaltung: Nehmen wir an, dass wir 16 Bits haben und dieses multiplizieren wollen: der Multiplexer nimmt diese 16 Bits auseinander in 1 + 14 + 1 Bits, der nimmt also das erste (blau) und das letzte Bit (rot) aus dem Bündel heraus. Wenn wir multiplizieren

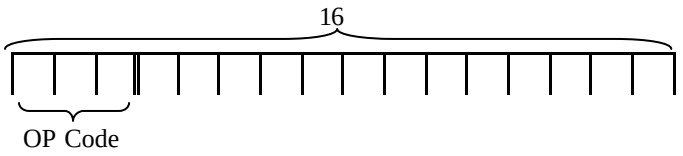
wollen, setzten wir den „Selector“ auf 2. Unser (rotes) letztes Bit wird rausgeworfen und alle anderen „rücken“ eins auf und anstatt des ersten (blauen, das jetzt das zweite ist) kommt eine 0 an diese Stelle. Entsprechend rücken alle Bits um eins nach rechts, wenn man dividieren möchte: das (blaue) erste Bit wird rausgeworfen (passiert, wenn man sich nicht benehmen kann) und an die Stelle des letzten (roten) Bits, das jetzt das Vorletzte ist, kommt eine 0.

Aufzeichnung der Vorlesung vom 15.12.2000 & vom 22.12.2000

Kommen wir zu etwas ganz anderem: dem „Programmieren“ unseres zukünftigen Computers. Unser Computer (wir gehen mal von einer von Neumann Konstruktion aus) ist wie folgt organisiert:



Unser Speicher hier mit der Wortlänge 16 besteht aus mehreren Registern (die wir später och bauen werden). Ein solches Register ist wie folgt aufgebaut:



Die ersten 3 Bits unseres Registers ist der sog. Op-Code (Operation Code). Da der Speicher (meist) von oben nach unten abgearbeitet wird, um dem Computer zu sagen, was er mit dem jeweiligen register tun soll, muss man zu jedem Register einen Befehl fügen, dieser Befehl steckt in unserem „OP-Code“.

Einige Möglichkeiten:

OP-Code	*Name*	*Zusatz*	*Inhalt*
000	LOAD	Adresse	Läd den Inhalt der Adresse in den Accumulator
001	STORE	Adresse	Speichert den Wert des Accumulator in eine Adresse
010	CLR		Setzt den Accumulator auf 0
011	INC		Erhöht den Accumulator um 1
100	BRZ	Adresse	Springt zur Adresse, wenn der Acccumulator leer ist
101	HALT		Stoppt das System
	DEC		Vermindert den Accumulator um 1

Es mag ungewöhnlich klingen, aber mehr brauchen wir nicht um Programme (sogenante Macros) für unseren Computer zu schreiben. Man kann Macros die schon vorhanden sind in ein neues Macro einbinden und dort verwenden.
Keine Macros wären z.B.:

```
CLR A =     Clear
            Store A
```
Löscht Inhalt von Adresse A

```
MOV A, B =  Load A
            Store B
```
kopiert Inhalt von Adresse A nach Adresse B

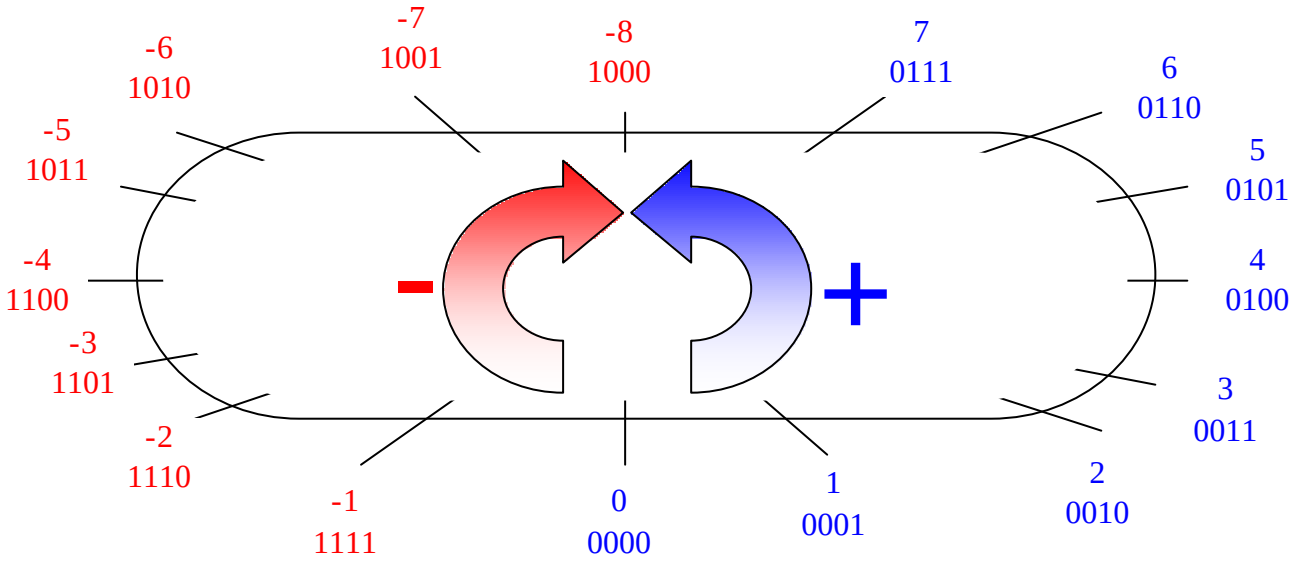
```
BRZ A, X =  Load A
            BRZ X
```
Springt zu Adresse X, wenn A = 0 ist.

```
GOTO X =    CLR
            BRZ X
```
Springt zu Adresse X

Naja, diese Macros sind nicht Weltbewegend, aber sehr nützlich im Umgang mit Assembler Programmen.
Mal ein schwierigeres, wir wollen ein Macro, das uns das Komlement von Adresse A liefert.

```
CMPL A =        CLR T1        // eine Hilfsvariable T1 wird gelöscht
          Loop: INC A         // der Wert A wird erhöht
                BRZ A, end    // wenn A = 0 soll zu END gesprungen werden
                INC T1        // T1 wird um eins erhöht
                GOTO loop     // es wird zu loop gesprungen
          End:  MOV T1, A     // der Wert von T1 wird nach A kopiert
```

Und warum funktioniert das? Beztrachten wir doch mal den binären Zahlenkreis:



Der Wert von A wird solange erhöt, bis er 0 ist und für jeden Schritt wird T1 um eins erhöht. Nehmen wir z.B. A= 0111 wir brauchen 9 Schritte um von 0111 auf 0000 zu kommen, erhöhen wir T1= 0000 9 mal ist sein Wert = 1001also das Komplement (als 2K Zahl) von 0111, ja it's magic!

Mit diesem Macro können wir weitere nützliche Macros definieren, wie z.B.

```
NEG A =     CMPL A      // nimmt das Komplement von A
            INC A       // erhöht A um eins
```

Dieses Macro invertiert also den Inhalt der Adresse A

Auch nützlich:

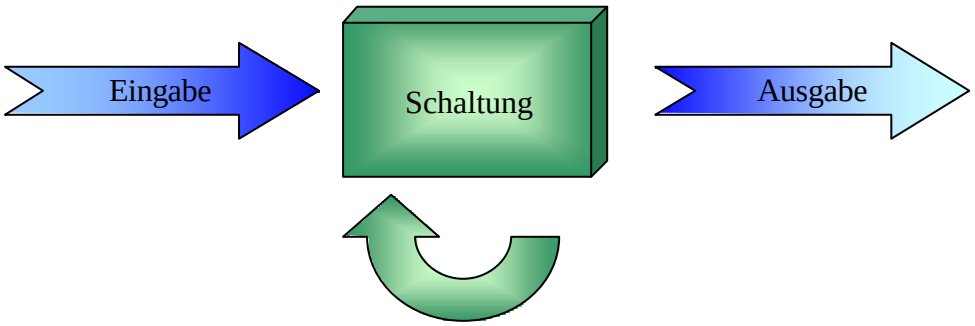
```
SHL A=      CLR T2      // löscht eine Hilfsvariable T2
      Loop: BRZ A, end  // wenn A = 0 dann springe zu end
            DEC A       // vermindere A um 1
            BRZ A, end  // wenn A = 0 dann springe zu end
            DEC A       // vermindere A um 1
            INC T2      // erhöhe A um 1
            GOTO loop   // springe nach loop
      End:  MOV T2, A   // kopiere den Wert von T2 nach A
```

Dieses Makro führt auf Adresse A ein SHL durch. Magic? Nicht ganz, wenn man genau hinschaut, merkt man, dass dieses Macro in T2 die Anzahl der Möglichkeiten von A 2 abzuziehen zählt.
Bspl.: man kann von 14 genau 7 mal eine 2 anzeihen, also SHL 15 = 7 (nätrülich binär in unserem Programm und Computer), bei ungeraden Zahlen wird nach unten gerundet (deshalb wird nach dem ersten DEC schon geprüft, ob A = 0 ist). Der Rest wird bei dieser Rechnung weggeworfen.

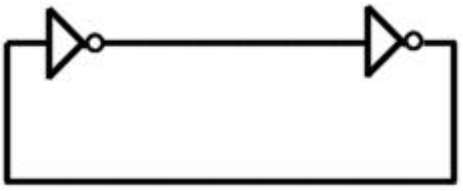
Aufzeichnung der Vorlesung vom 12.1.2001

So, genug elementar programmiert, zurück zu den wichtigen Dingen des Lebens: der Hardware. Bis jetzt haben wir nur Schaltungen und Systeme zusammengebaut, die eine (oder mehrere) Eingaben und eine oder mehrere Ausgaben hatten und diese Ausgabe war ausschließlich abhängig von der Eingabe. Also so was:

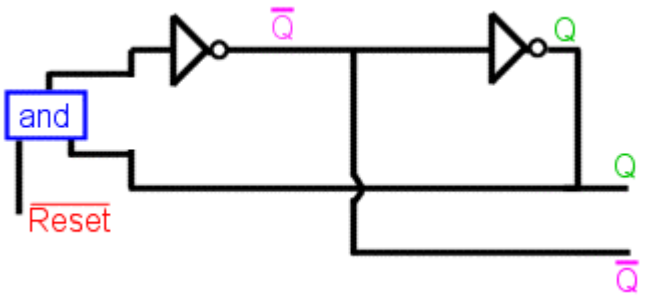
Das ist aber auf die Dauer ziemlich langweilig, deshalb wollen wir da etwas ändern, also eine Schaltung bauen, die von der Eingabe und von dem Zustand der in der Schaltung bereits vorherrscht eine Ausgabe liefert. Als Bildchen vielleicht so



Konkret könnte so eine Schaltung so aussehen:

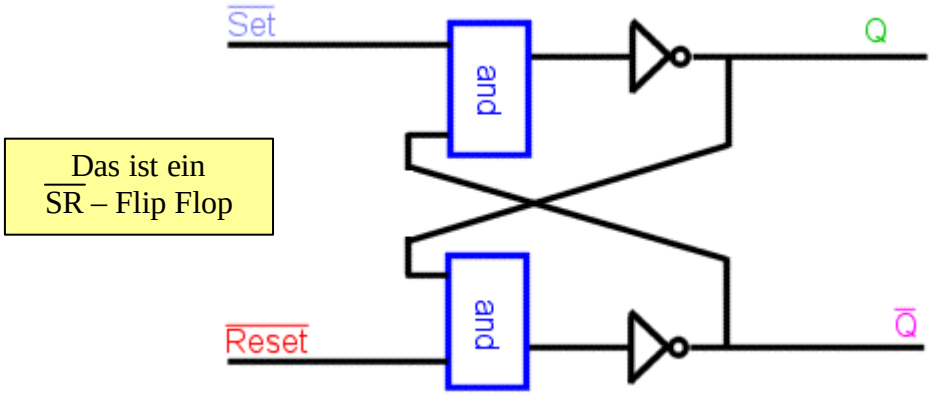


Man kann sich vorstellen, dass der Strom in dieser Schaltung immer im kreis läuft und pro runde zweimal negiert wird. Da es zweimal negiert wird kommt der Strom in der nächsten Runde also genau wieder in dem gleichen Zustand an, den es auch die vorigen Runden an diesem Punkt (z.B. vor dem ersten Negator) hatte. Basteln wir an unsere Schaltung noch ein bisschen Schaltlogik, so kommen wir auf so was:

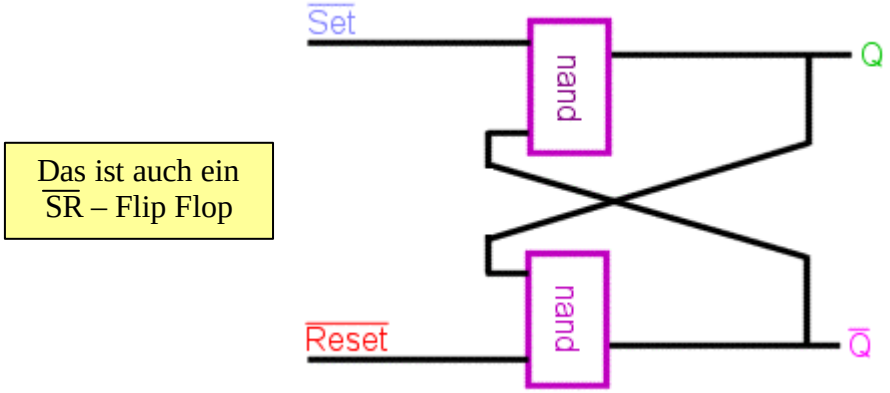


Q und nicht Q, das ist offensichtlich, ist unsere Ausgabe und nicht Q ist (wie der Name schon sagt) immer das Gegenteil von Q, da zwischen den Leitungen eine Negation liegt. Wenn $\overline{\text{Reset}}$ = 1 ist ändert sich an der Schaltung nicht (wenn ein and eine 1 als Eingabe bekommt, so leitet sie die andere Eingabe einfach weiter, als wäre es eine einfache Leitung). Ist $\overline{\text{Reset}}$ hingegen = 0 so wird die Schaltung und damit Q auf 0 gesetzt, egal was der vorige Zustand

der Schaltung war. Jetzt kommt zwar nur eine geringfügige Designänderung, die aber zuweilen sehr verwirrend wirken kann. Alles was sich ändert, ist das wir an entsprechender Stelle noch ein and hinzufügen, und die Schaltung selbst etwas anders verlegen. Wenn man genau hinschaut erkennt man, das (abgesehen von dem and) sich eigentlich kaum etwas verändert hat).



das können wir aber auch einfacher haben:



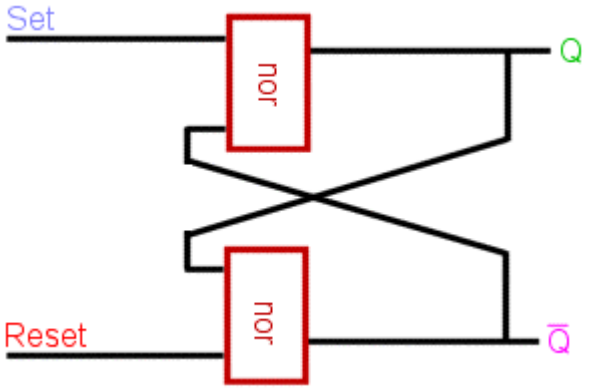
Wenn Set = Reset = 1 ändert sich nichts in der Schaltung und Q bleibt für immer und ewig = Q. Wenn man das macht dreht sich der Strom in der Schaltung immer im Kreis (irgendwann wird ihm schwindelig und es muss sich übergeben aber das dauert eine Weile!). Setzt man hingegen Set auf 0, so wird die Schaltung und damit Q auf 1 gesetzt. Setzt man Reste auf 0 so wird die Schaltung und damit Q auf 0 gesetzt. Beide gleichzeitig auf 0 zu setzten ist nicht empfehlenswert, weil dann die Schaltung völlig verwirrt ist, mit anderen Worten: man kann dann nicht voraussehen, was dann passiert. Es ergibt sich also folgende Wertetabelle:

$\overline{Set}$	$\overline{Reset}$	Q_{next}
1	1	Q
0	1	1
1	0	0
0	0	???

Nicht vorhersagbar

Da wir aber nicht immer bei 0 sondern bei 1 „setten“ oder „resetten“ wollen, ersetzten wir die nand’s durch nor’s:

Das ist ein
SR – Flip Flop

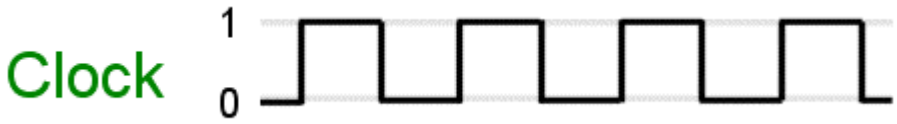


und tada es ergibt sich folgende Wertetabelle:

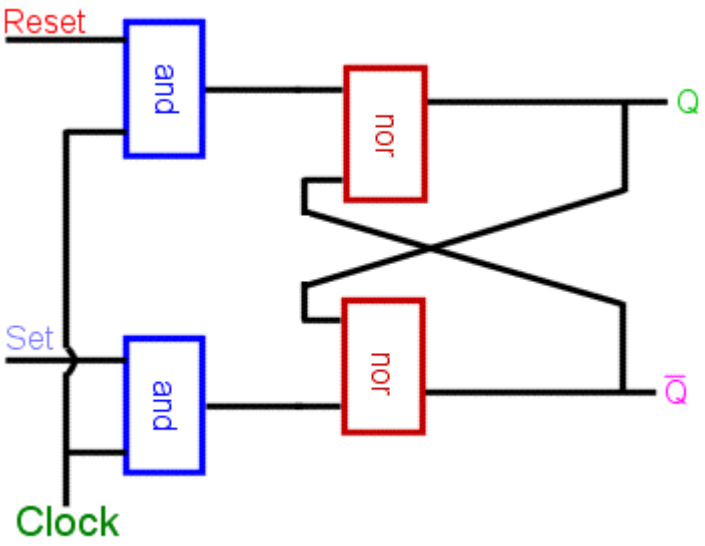
Set	Reset	Q_{next}
0	0	Q
1	0	1
0	1	0
1	1	???

Nicht vorhersagbar

Soweit alles klar (?)! Ein wenig müssen wir aber noch verändern, nämlich ein sog. Clocksignal hinzufügen. Das Clocksignal „sagt“ unserer Schaltung, ob Sie überhaupt auf Set oder Reset reagieren darf. Dieses Clocksignal kann man sich als Taktgeber vorstellen, der in regelmäßigen (manchmal auch in unregelmäßigen) einen Impuls gibt. Diesen Impuls kann man in einem Impulsdiagramm darstellen und das sieht dann so aus:

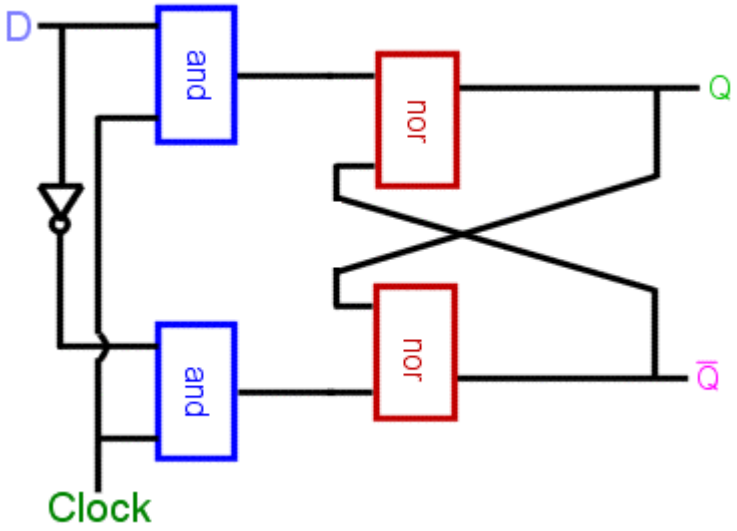


und die veränderte Schaltung dazu sieht so aus:



Nur wenn die Clock = 1 können die and’s etwas durchlassen, also nur dann kann sich was ändern. Sowas nennt man dann auch Level-triggeret, weil nur wenn die Clock auf „Level“ 1 ist kann sich etwas ändern. Es gibt auch andere Flip Flops, z.B. sogenannte D - Flip Flops

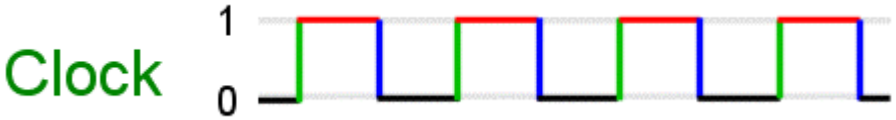
alles was daran verändert ist, ist dass es Statt Set und Reset nur noch ein D gibt, das sich wie folgt ergibt:



Hier kommt man nicht mehr in die Gefahr 1 und 1 anzusetzen, es gibt ja nur eine Eingabe ☺. Die Wertetabelle hierfür ist:

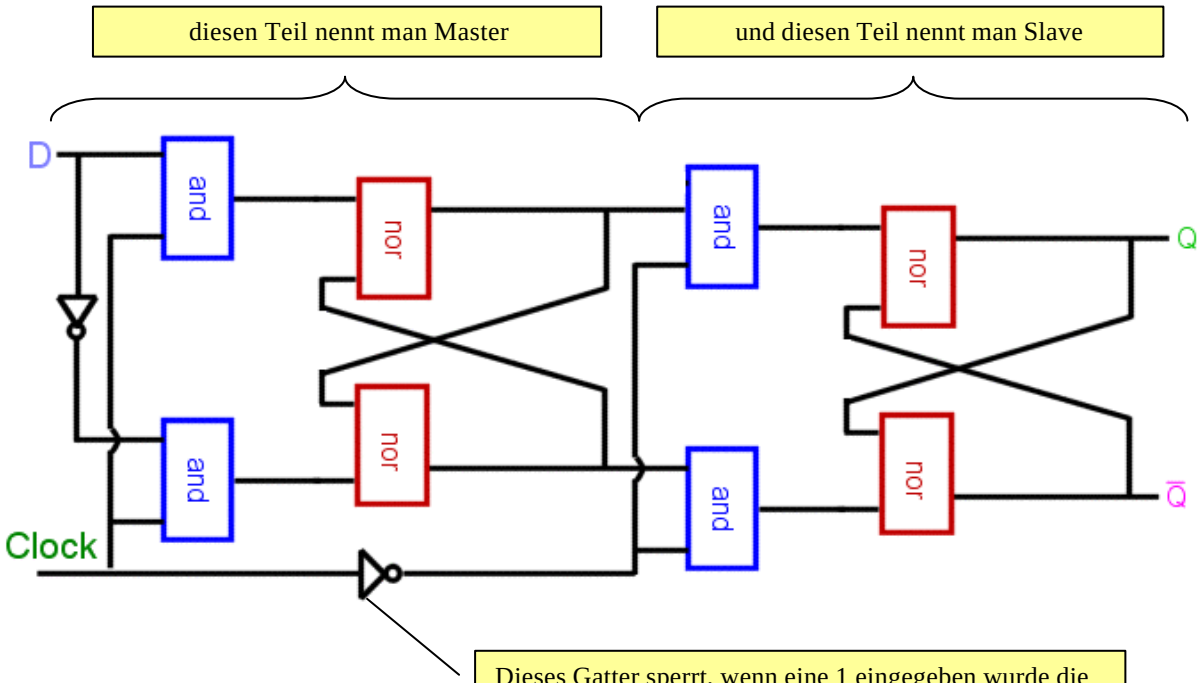
D-Flip-Flop	
D	Q_{next}
0	0
1	1

Schauen wir uns doch noch mal dieses Pulsdiagramm an:

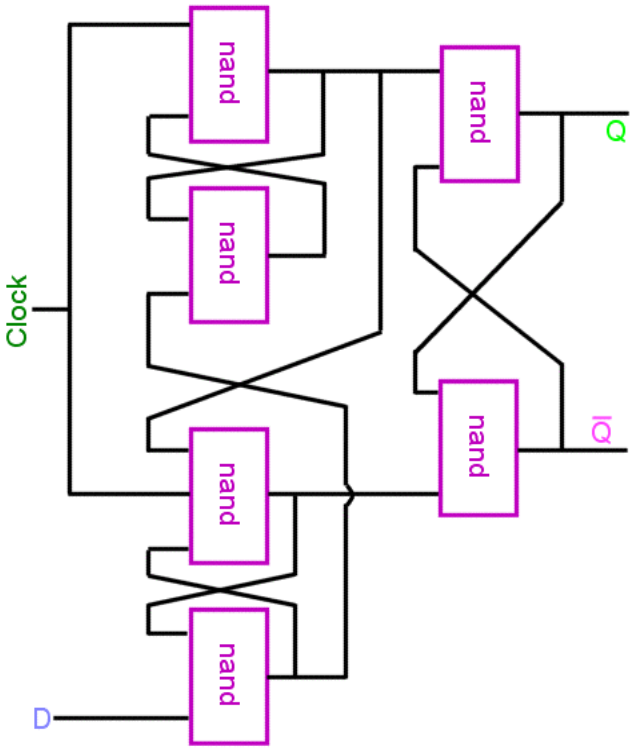


Hmm, eigentlich ist uns so eine Schaltung ja zu langweilig. Reagiert nur wenn sie auf Level 1 (rot dargestellt) ist, diese Zeitdauer ist uns viel zu lang. Interessanter ware doch eine Schaltung, die auf die steigenden (rot) oder fallenden (blau) Flanken reagieren würde.

Gesagt getan, wir haben uns ein edge-triggered FlipFlop (hier ein D-Flip Flop) gebaut:



Dieses Gatter sperrt, wenn eine 1 eingegeben wurde die and's des Slave Teils. Da jedes Gatter eine Zeitverzögerung hat, kommt das Signal schnelle durch das eine and als es sich durch den master Flip-Flop gewurschtelt hat. Dieses Flip Flop reagiert also nur bei dem Wechsel von 0 zu eins1 (grün im Impulsdiagramm auf Seite 52)

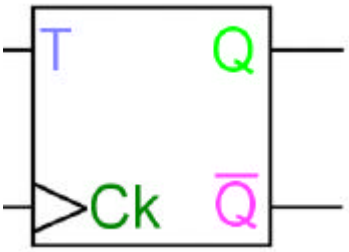


Solch ein D Flip-Flop kann man aber auch so konstruiert werden, dass es auf die roten Flanken im Impulsdiagramm Seite 52 reagiert, also bei dem Wechsel von 1 zu 0 (siehe links).

Auf der folgenden Seite noch einmal eine Übersicht über die bekanntesten Flip-Flops, deren Bedeutung und deren Symbol.

ÜBERSICHT ÜBER DAS 1 X 1 DER

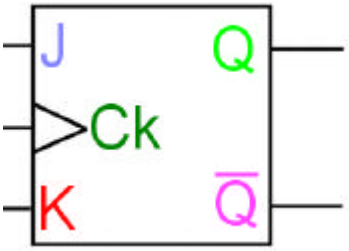
FLIP-FLOPS



T Flip Flop

Bei T = 0 ändert sich der Wert nicht, bei T = 1 wird er gekippt, was immer auch drin war ist jetzt das Gegenteil

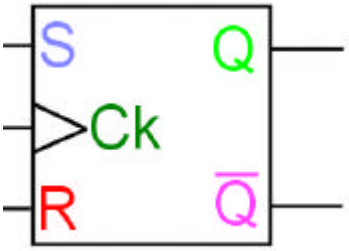
T-Flip-Flop	
T	Q_{next}
0	Q
1	Q̄



J K Flop Flop

Das mit wichtigste Flip-Flop überhaupt. Aus diesem Flip-Flop kann man alle anderen nachbauen

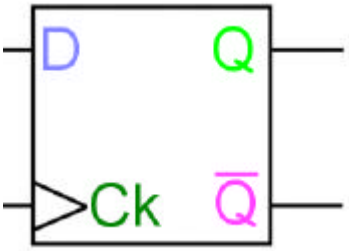
J-K-Flip-Flop		
J	K	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	Q̄



S R Flip Flop

Set setzt die Schaltung auf 1, Reset das ganze auf 0. Wenn beides 0 ändert sich nichts

S-R-Flip-Flop		
S	R	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	X

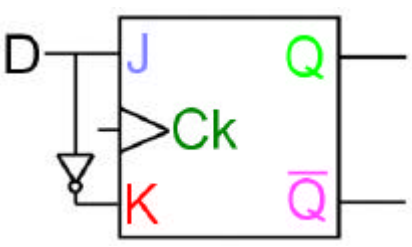


D Flip Flop

Wenn D = 0 wird die Schaltung auf 0 gesetzt, bei 1 auf 1.

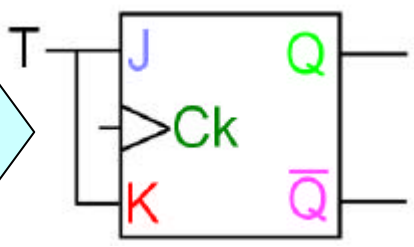
D-Flip-Flop	
D	Q_{next}
0	0
1	1

Das Ck steht für Clock und das Dreieck an diesem Eingang dient nur dem Zweck der einfacheren Erkennbarkeit dieses Eingangs, hat sonst nichts zu bedeuten.



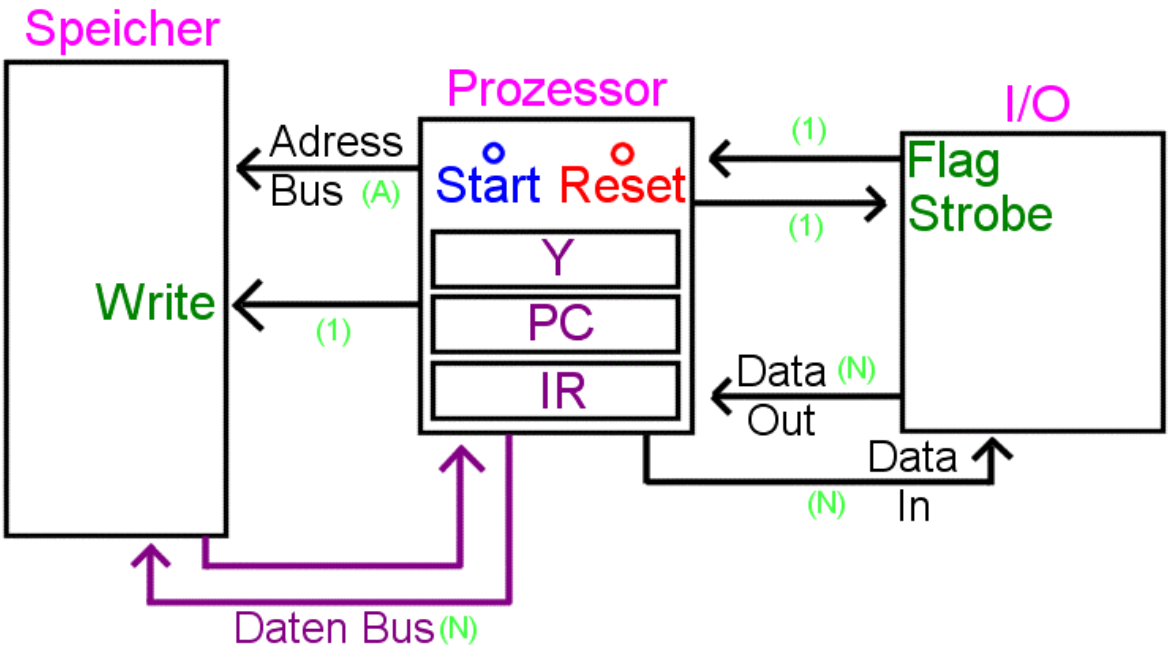
Das ist ein D-Flip-Flop gebastelt aus einem J-K-Flip-Flop

Und das ist ein T-Flip-Flop gebastelt aus einem J-K-Flip-Flop



Aufzeichnung der Vorlesung vom 19.1.2001

So, jetzt geht's los! Nachdem wir nun so viele und entscheidende Teile eines Computers nachgebaut haben, wollen wir uns nun endlich einem „richtigen" Computer zuwenden. Wie soll des grundsätzliche Design aussehen? Nun, nehmen wir folegndes Grundkonzept, bestehend aus Speicher, Prozessor und Ein-, Ausgabe (auch IO von Input Output genannt):



soweit so verwirrend, also was ist was?
Speicher

Eingänge:

Adressbus =	A Bits breit, der Prozessor „sagt" welche Adresse angesprochen werden soll
Write=	wenn „Write" auf 1 gesetzt wurde, dann will der Prozessor in den Speicher schreiben, andererseits steht der Datenbus auf Lesebereitschaft
Datenbus =	In Abhängigkeit von Write bekommt der Speicher durch diesen Bus seine Informationen

Ausgänge:

Datenbus =	In Abhängigkeit von Write liefert der diesen Bus seine Informationen an den Prozessor

IO

Eingänge:

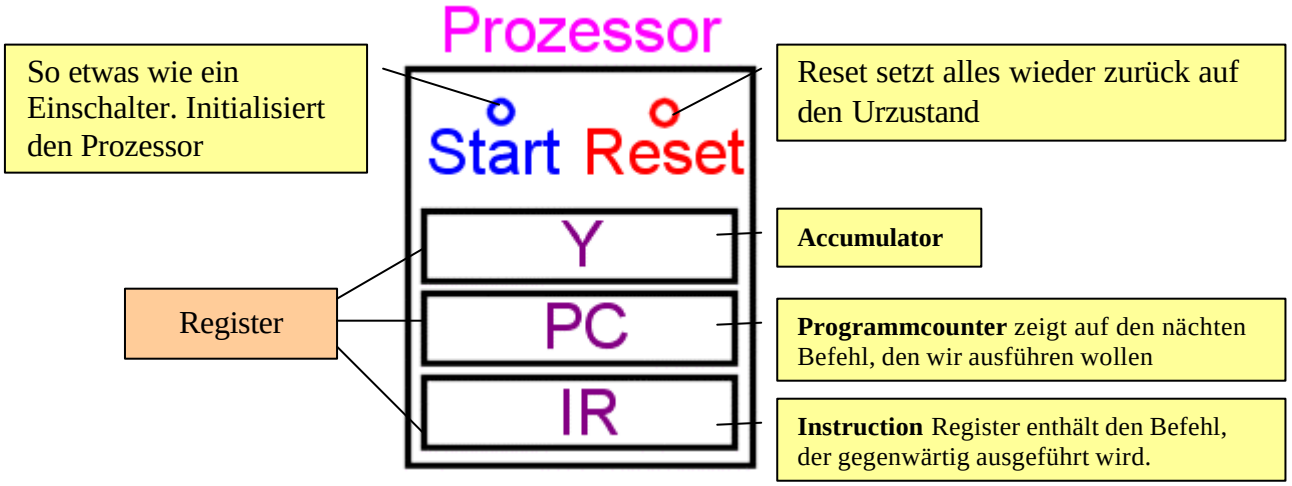
Flag =	Sollten Eingaben an den Prozessor geleitet werden ist diese Leitung = 1
Data Out =	Liefert die Daten an den Prozessor

Ausgänge:

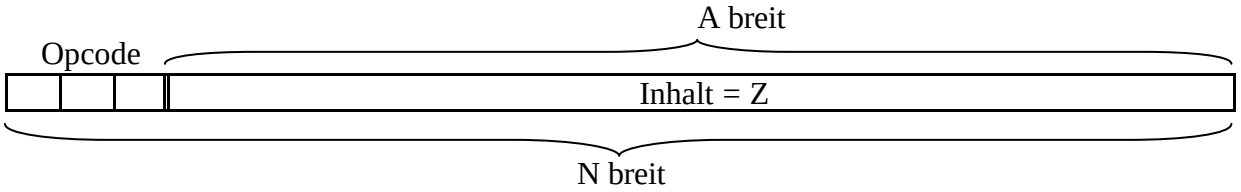
Strobe =	sollten Daten empfangen werden, so setzt der Prozessor diese Leitung auf 1

Data In = auf diesem Weg kommen die Daten an

Und was ist mit unserem Prozessor?



Natürlich braucht unser Computer noch unsere elementaren Rechenoperationen und irgendwie müssen wir ja auch noch sagen, was er womit tun soll, dafür haben wir in dem Register IR (unten abgebildet) einen Kopf namens Opcode bestehend aus 3 Bits und einen Inhalt (Z) bestehend aus A Bits das Ganze ist also A + 3 = N Bits breit. Der Inhalt Z ist jedoch „nur“ der Verweis auf die Adresse im Speicher, an der die gewünschte Information steht, nicht die Zahl die, z.B. addiert werden soll selbst (in den Beispielen wird das vielleicht klarer)



In den 3 Bits des Opcodes wird die Instruktion gespeichert und in dem Rest der Inhalt unsere Befehle werden also in Zahlen gespeichert:

Opcode			Dezimal	Name	Bedeutung
0	0	0	0	LOADI	Läd den Inhalt von Z in den Akkumulator
0	0	1	1	STORE	Speicher an der Stelle Z wird mit Y geladen
0	1	0	2	JUMP	Der PC “springt” zur Stelle Z. In sog. Von Neumann Computerarchitekturen sind Datenspeicher und Befehlsspeicher nicht voneinander getrennt, also befinden sich auch die Befehle (als Binärzahlen codiert) im Speicher. Der PC springt also zu einem Befehl
0	1	1	3	LOAD	Läd den Y mit Inhalt der Adresse Z
1	0	0	4	ADD	Y = Y + Inhalt von Adresse Z
1	0	1	5	SUB	Y = Y - Inhalt von Adresse Z
…					
1	1	1	7	HALT	System wird angehalten und wartet auf Restart

Mit den drei Bits dees Opcodes können wir ja bekanntlich nur bis 7 zählen, leider reicht das nicht aus, deshalb gibt es mehrere Befehle unter 7. Damit der Computer diese Befehle noch auseinanderhalten kann, benutzt er noch die rechten 4 Bit der Adresse, also von Z. In der Tabelle sind nur die Befehle aufgelistet, die wir unmittelbar brauen und weitestgehend ja

schon kennen. Der Befehl HALT „greift“ (wie man in der Tabelle sehen kann) bei (111) = 7 also werden die rechten 4 Bits von Z auch noch abgefragt diese müssen im Falle von Halt (1000)=8 sein, bei anderen 4 letzten Bits in Z wird ein anderer Befehl ausgeführt.

Bspl.:

LOADI 3 =

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Läd Inhalt von Adresse 3 in den Akkumulator

STORE 3 =

0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Inhalt von Akkumulator kommt nach Adresse 2

LOAD 20 =

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0

Läd Inhalt von Adresse 20 in den Akkumulator.

HALT =

1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

System wird angehalten

Jetzt aber los! Zunächst basteln wir uns ein Register.

Wir erinnern uns (hoffentlich) noch, dass Flip-Flops ein Bit speichern und das waren die Wertetabellen von Flip-Flops:

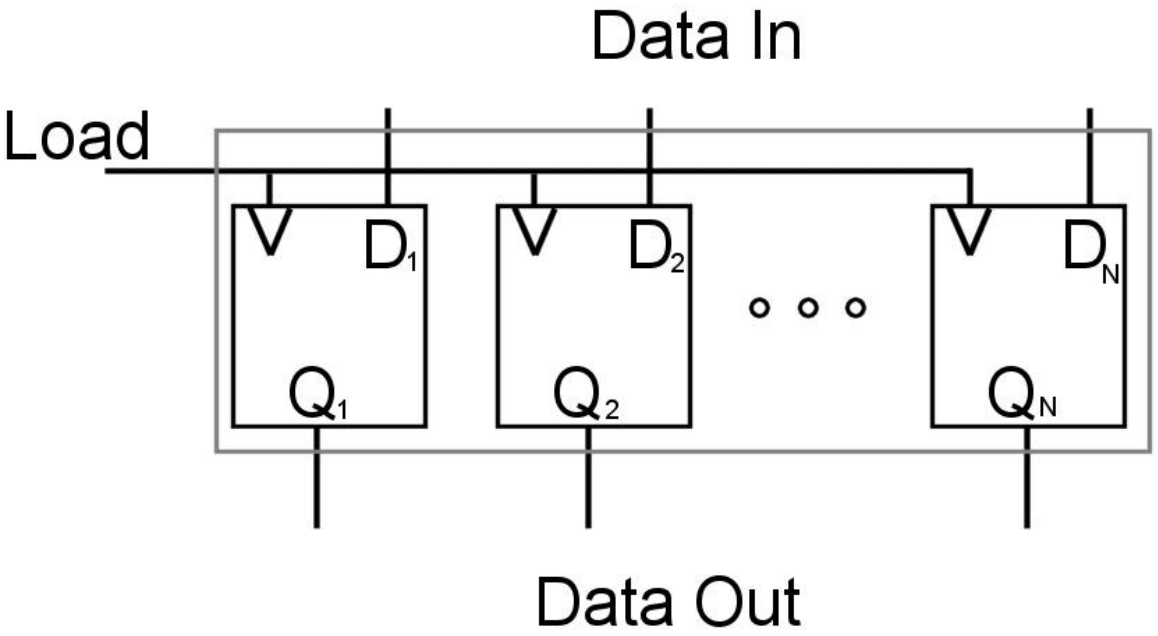
D-Flip-Flop	
D	Q_{next}
0	0
1	1

T-Flip-Flop	
T	Q_{next}
0	Q
1	$\overline{Q}$

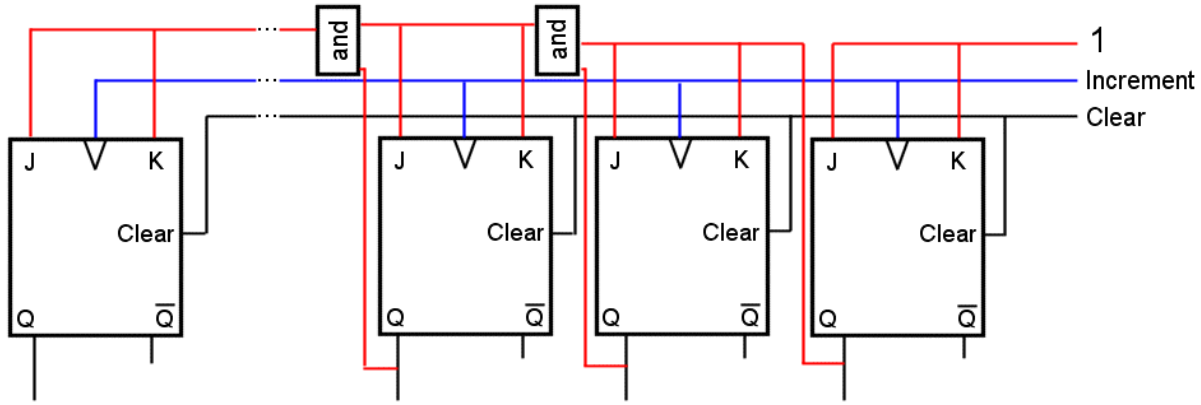
S-R-Flip-Flop		
S	R	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	X

J-K-Flip-Flop		
J	K	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

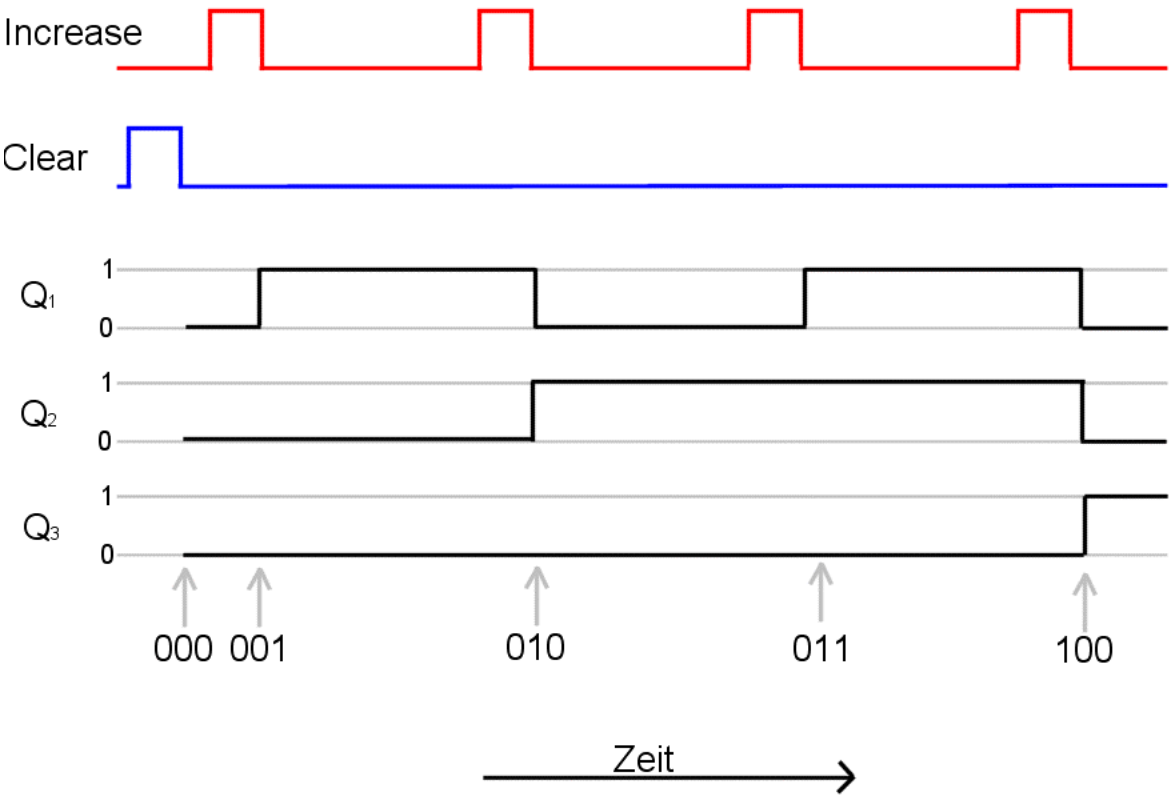
Machen wir erste Gehversuchen mit einem D-Flip-Flop



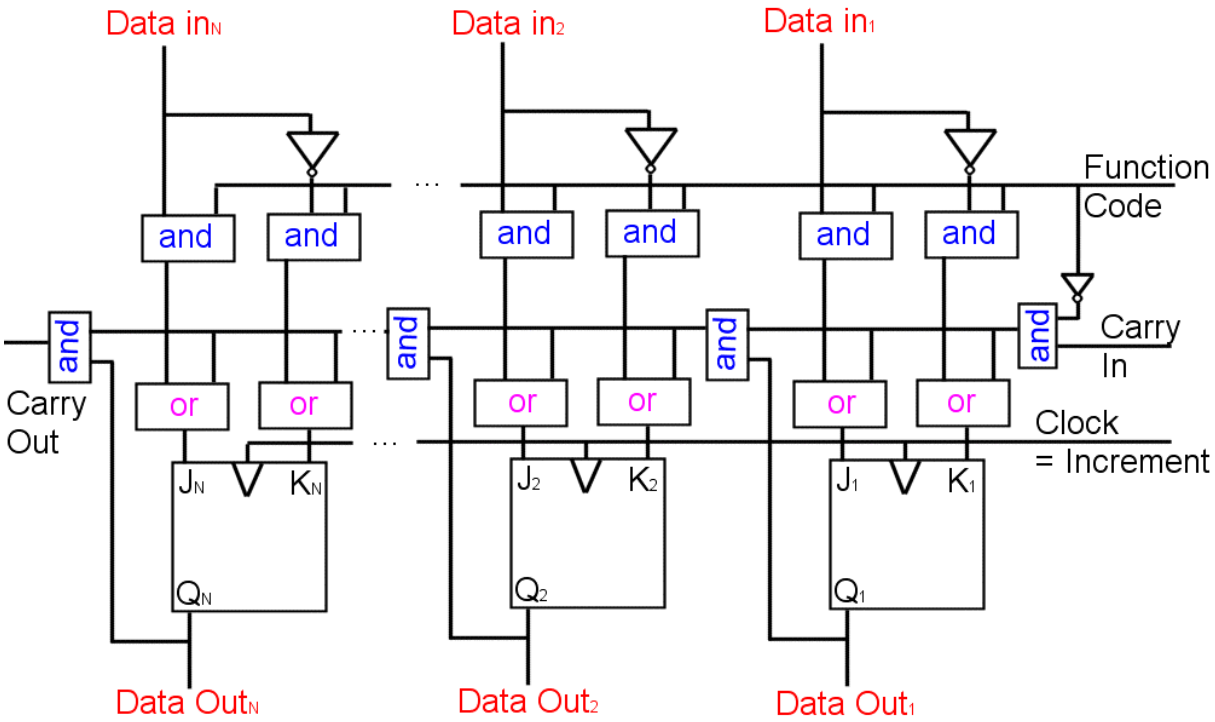
war ja eigentlich nicht schwierig, aber wir wollen ein etwas komplizierteres aber dafür leistungsfähigeres Register bauen. Dafür basteln wir zunächst einen „Hochzähler“ aus Flip-Flops dieser soll von 0 ab an hochzählen, wann immer ein „increase“ Impuls gegeben wird. Nun unser Hardwarehändler hat schon wieder ein Sonderangebot für uns (und zwar J-K-Flip-Flops), wir sind ja auch mittlerweile gute Kunden und da wir ja wissen, dass wir mit J-K-Flip-Flops alles machen können, was wir auch mit den andern anstellen können, kaufen wir einen Haufen davon (es war ja gerade Weihnachten) und los geht’s.



Analysieren wir das mal: Also in bei jedem incrementen wird kehrt sich der letzte Wert um (aus einer 0 wird eine 1 und andersherum) soweit richtig. Und wann immer alle rechtsstehenden Ergebnisse 1 sind wird der Wert selbst zur 1 und alle rechtsstehenden 1sen zur 0 (aus 00111 wird 01000), ich empfehle 5 mal diese Seite ausdrucken und mit Bleistift von 0 nach 4 aufaddieren!
Je nach dem wie das J-K-Flip-Flop gebaut wurde reagiert es auf positive oder negative Spannungsflanken. Baut man also einen Zähler (der nei negativen Spannungsflanken reagiert) nach, so ergibt sich folgendes Zustandsdiagramm:



So, warum wir das ganze gemacht haben? Nun, unser Zähler kann nur von 0 ab an hochzähen und wir kombinieren das mit unserem Register, das wir bereits gebaut haben und tada... wir haben ein (bedrohlich kompliziert wirkendes) „Speicherundhochzähl-“ Register geboren:

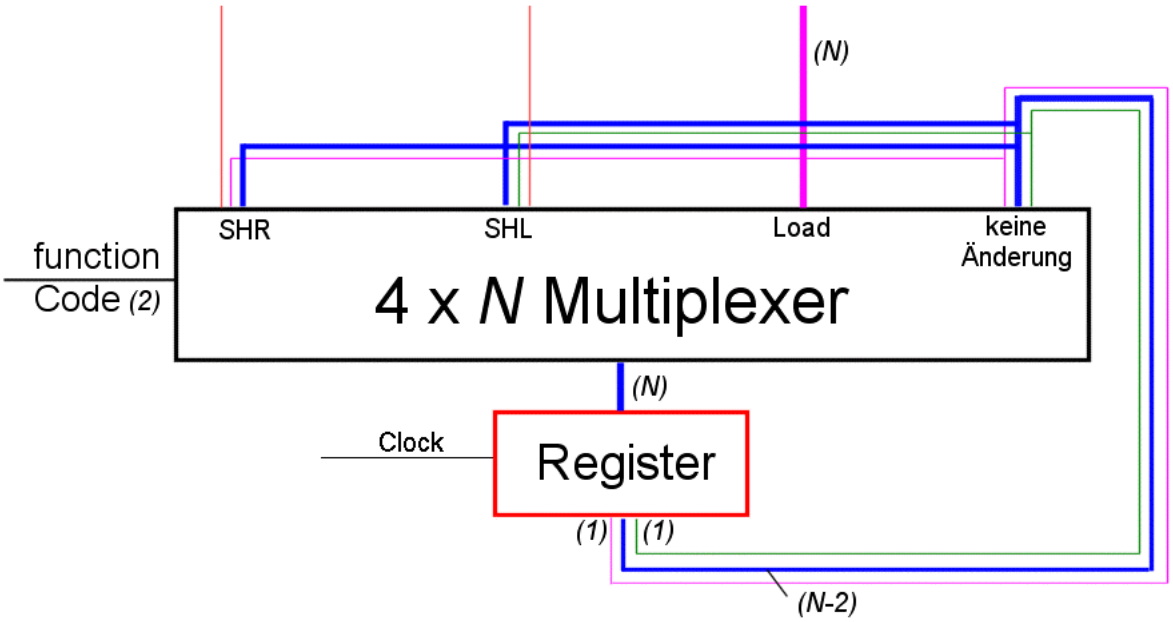


Auch wenn das jetzt nicht so aussieht, das ist exakt die Kombination aus unseren beiden vorausgegangenen Überlegungen. Was passiert hier? Nun, wenn bei „Function Code“ eine 1 eingegeben wird, leiten die oberen and’s die „Data in“ Leitungen einfach durch und bei den unteren or’s (weil das rechte and eine 0 durch Negation vom „Function Code“ produziert) passiert das gleiche. In diesem Fall wird also das Register mit den Werten aus „Data in“

geladen. Wenn der „Function Code“ auf 0 gesetzt wird kann durch die oberen and’s nichts mehr hindurchkommen, die oberen Leitungen sind also blockiert. Wenn die Clock einen Impuls sendet bewirkt das die Erhöhung unseres Registers um 1. Das „Carry In” entspricht der konstanten 1 unseres Zählers.

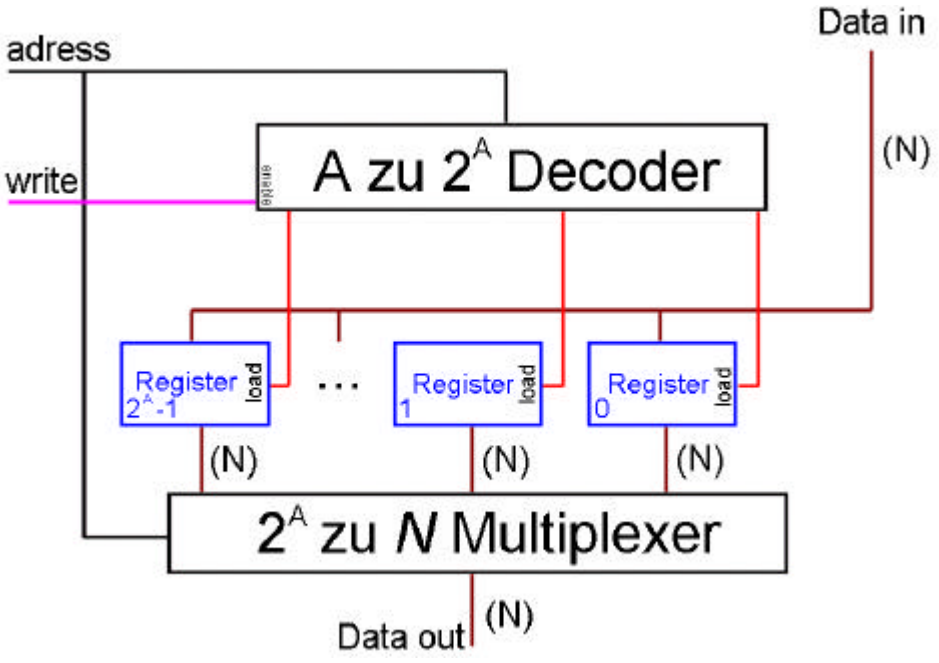
Aufzeichnung der Vorlesung vom 26.1.2001

Eigentlich könnte uns dieses Register ja schon ausrechen, aber wir könnten da noch etwas verändern, genauer wir wollen unser Register um einen Multiplexer erweitern, der je nach Befehl, läd, ein SHL, ein SHR oder gar nix macht:
Die Schaltung hierfür sieht so aus:

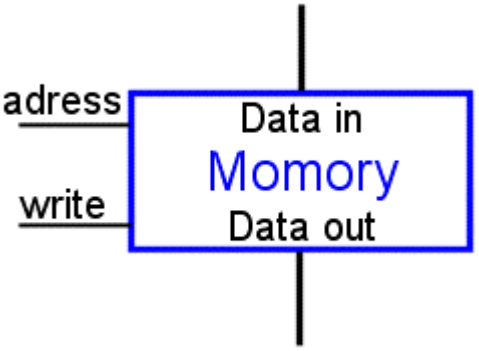


was man für Function Code 2 oder 3 also bei SHR oder SHL als externes Bit in den Multiplexer gibt, kann entweder das Vorzeichen, oder eine konstante 0 oder das herausgeworfene Bit sein, je nach dem, wie man diese Schaltung konstruiert. Anbei: es gibt auch Schifter, die in einem Durchgang beliebig viele Stellen shiften können nennt man barrel-shifter.

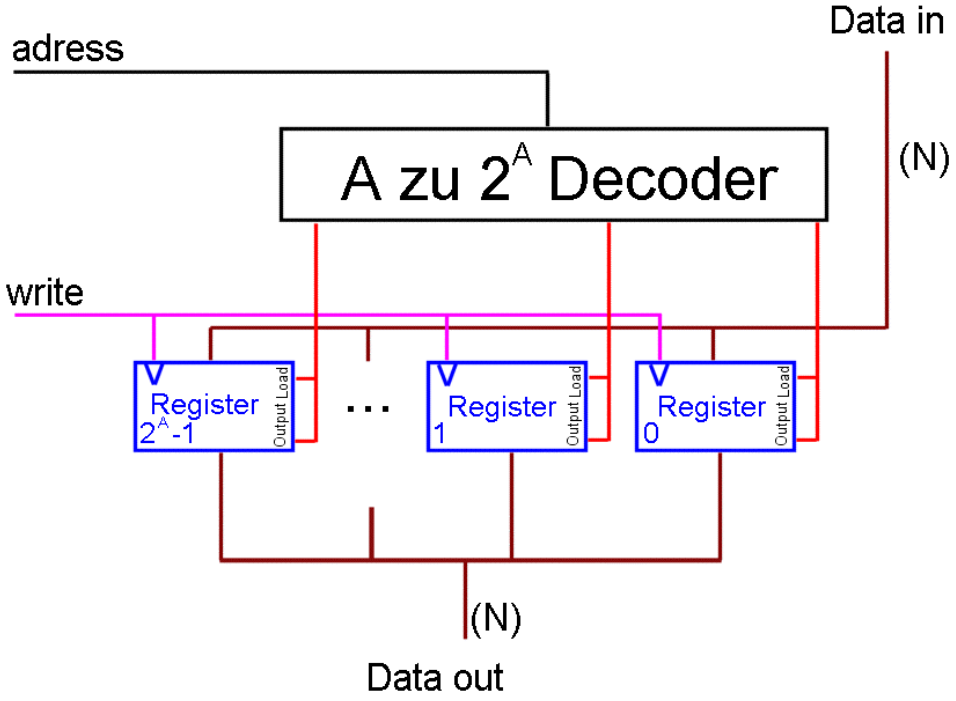
So, jetzt wollen wir einen adressierbaren Speicher bauen, also ein Modul, dem man „sagen“ kann, welchen Zustand man auslesen will. Konkret sieht das so aus:



Wann immer wir dieser Schaltung eine Adresse geben, wird die in dem Register gespeicherte Adresse beim Data-out ausgegeben. Wird zusätzlich noch ein Write = 1 gesetzt, so setzt der Decoder das entsprechende Register auf „Load“ und so wird das „Data In“ (von oben kommend) in das Register geladen. Der Multiplexer ist für das Auslesen verantwortlich. Vereinfacht sieht das so aus:

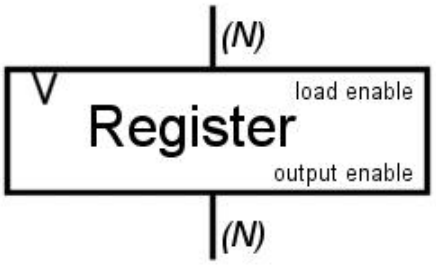


Wenn man aber viele Register in ein Speichermodul einbauen will, dann steht man vor einem Problem: Die „Data In“- Leitung teilt sich in so viele Leitungen, wie Register im Speicherbaustein stecken und in der Praxis verliert dann leider (ab einer gewissen Anzahl von Leitungen) die Leitung das Signal. Was man da macht ist einfach, man baut ein neues Register, welche beim „Data In“ und beim „Data Out“ (dann brauchen wir kein Multiplexer mehr) einfach das Register von der Leitung trennt. Dieser Zustand, der sich die Leitung dann befindet nennt sich „third state“. Man spart sich den Multiplexer beim Ausgang, weil man die „Data Out“ Leitungen einfach zusammenführen kann, ein Kurzschluss kann nicht erzeugt werden, weil sich alle bis auf ein Register im „third state“ befinden. Wie so ein „third state“ hergestellt wird, wird an anderer Stelle noch gezeigt, die dazugehörige Schaltung sieht jedoch so aus:



Die Adresse des Decoders „sagt“ dem Register, ob es eine Verbindung zu den „Data In“ Leitungen und zu den „Data Out“ Leitungen erstellen darf oder nicht.

Ein solches Register sieht so aus:



Leitung	**Wert**	**Beschreibung**
Output enable	0	Die Kabel werden von der Leitung getrennt, das es kann nichts ausgelesen werden.
Output enable	1	Die Kabel werden verbunden, der Wert kann ausgelesen werden.
Load enable	0	Three-state-Zustand der “Laden Leitung”.
Load enable	1	Das Register ist in der Lage einen Zustand zu lesen und zu speichern.

Fragen, Feedback, Lob und konstruktive Kritik an
stehn@inf.fu-berlin.de
oder direkt an mich: Fabian A. Stehn

(Zitieren und verwenden des Bildmateriales ist bei Angabe der Quelle erlaubt und erwünscht!)

ich habe es leider nicht geschafft die Fehler der Version 1 auszubügeln, ist aber in Arbeit! Danke für Eure eifrige Fehlersuche!

Ich übernehme keine Verantwortung für die korrektheit dieser erweiterten Aufzeichnungen, benutzen auf eigene Gefahr, Eltern haften für ihre Kinder!