Teil 4 – Alles aus Nicht und Oder oder Und

Du kennst nun vier verschiedene Logikgatter: AND, OR, XOR und NOT. Neben diesen gibt es noch einige weitere, die meisten sind aber nur Kombinationen dieser vier. Zum Beispiel ist da noch das **NOR-Gatter**:

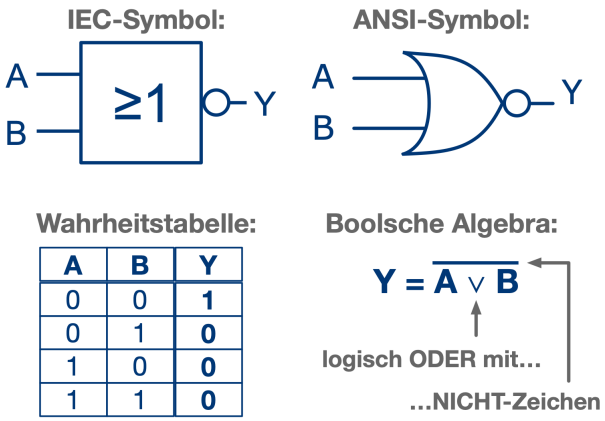

Bild 20 – Nicht-ODER-Gatter

Das Symbol des NOR zeigt schon, dass es sich hier um eine Kombination von OR und NOT handelt: Der Kreis am Ausgang bedeutet, das der Ausgang *invertiert* (man sagt auch *negiert*) ist. In der Wahrheitstabelle siehst du, dass das Resultat im Vergleich zum OR-Gate invertiert ist.

Boolesche Algebra

Logische Verknüpfungen können auch als Gleichungen dargestellt werden. Man spricht hier von *Funktionsgleichungen*. In obigen Bildern siehst du diese jeweils unten rechts. Die Bedeutungen der Zeichen und die Rechen-Regeln sind in der *booleschen Algebra* (benannt nach dem englischen Mathematiker und Philosophen *George Bool*, † 1864) zusammengefasst. Aus der „normalen" Algebra kennst du etwa die Regel, dass „Punkt vor Strich kommt": 2 * 3 + 4 ergibt was anderes als 2 * (3+4). Ähnliche Regeln gibt es auch in der booleschen Algebra (UND (∧) kommt vor ODER (∨)). So auch das *Distributivgesetz*:

$$(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)$$

Die Klammern auf der linken Seite dienen der Übersicht. Mathematisch gesehen sind sie nicht nötig, da ja „UND vor ODER kommt", der ∧-Operator also stärker bindet, als der ∨-Operator. Auf Wikipedia findest du weitere Gesetze der booleschen Algbra. Von diesen betrachten wir hier nur die vom englischen Mathematiker Augustus De Morgan († 1871) erkannten Gesetze.

De Morgansche Gesetze

Die De Morganschen Gesetze lassen sich wie folgt darstellen:

$$\overline{a \wedge b} = \overline{a} \vee \overline{b}$$

oder auch:

$$a \wedge b = \overline{\overline{a} \vee \overline{b}}$$

Wir können die Gesetze so formulieren: Wenn ich von einer bestimmten Funktion *alle* Elemente *und* die gesamte Funktion negiere, dann erhalte ich eine mathematisch identische Funktion: Betrachte die untere der beiden Gleichungen oben: Aus *a* wird *a-nicht*, aus *b* wird *b-nicht*, aus dem Und-Operator wird ein Oder-Operator und die ganze Funktion wird negiert. Die Wahrheitstabelle zeigt die Wahrheit; nämlich dass die beiden Funktionen das gleiche Resultat ergeben:

a	b	a ∧ b	$\overline{a}$	$\overline{b}$	$\overline{a} \vee \overline{b}$	$\overline{\overline{a} \vee \overline{b}}$
0	0	0	1	1	1	0
0	1	0	1	0	1	0
1	0	0	0	1	1	0
1	1	1	0	0	0	1

Und aus Oder und Nicht

Mit den De Morganschen Gesetzen können wir also eine UND-Verknüpfung (a ∧ b) durch eine Oder-Verknüpfung ersetzen, wenn wir noch ein

paar Nicht-Verknüpfungen hinzuziehen. Mit Logik-Gattern aufgebaut, sieht das wie folgt aus:

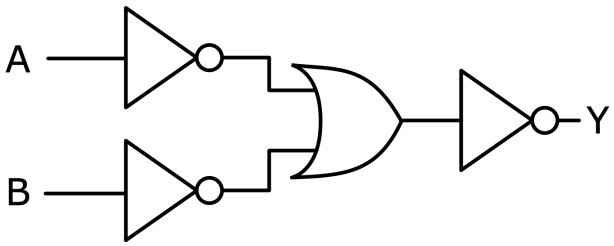

Bild 21 – UND-Verknüpfung aus NOT- und OR-Gates

Vergleiche die Schaltung in Bild 21 mit der Gleichung in der Tabelle oben (Spalte ganz rechts): Stimmen Gleichung und Schaltung überein?

Wie du bereits weisst, könnten die rechten beiden Gatter, das OR und das NOT, auch durch ein NOR ersetzt werden. Und auch die beiden NOT-Gates können ersetzt werden:

Zeichne ein NOR-Gatter und verbinde beide Eingänge miteinander. Jetzt hast du bloss noch einen Eingang. Überlege mit Blick auf die Wahrheitstabelle in Bild 20, was jeweils am Ausgang anliegt, wenn die verbundenen Eingänge 1 und wenn sie 0 sind.

Du siehst: Mit einem NOR lässt sich auch ein NOT-Gate bauen: wenn die beiden Eingänge des NOR verbunden werden, funktioniert es als NOT. Wir können obige Schaltung also auch wie folgt zeichnen:

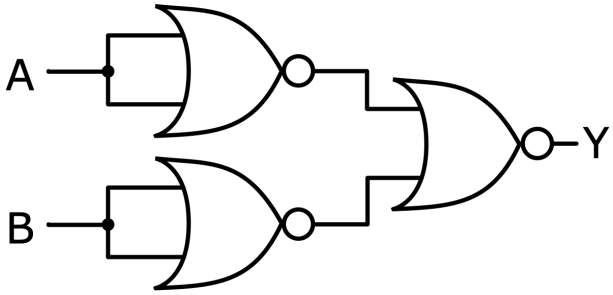

Bild 22 – UND-Verknüpfung aus NOR-Gates

Es ist also möglich, eine UND-Verknüpfung mit ausschliesslich NOR-Gates aufzubauen. Und nicht nur das: *Alle* logischen Verknüpfungen lassen sich mit NOR-Gates aufbauen. Damit lassen sich *alle* logischen Schaltungen – und seien sie noch so komplex – mit NOR-Gates aufbauen!

Das Gleiche gilt für NAND-Gates, also die Kombination von NOT- und AND-Gate. Mit zwei grundlegenden Elementen, nämlich mit:

NOT- und OR-Gates —> NOR-Gates

oder mit:

NOT- und AND-Gates —> NAND-Gates

… können wir *jede* logische Schaltung aufbauen.

Du fragst dich jetzt vielleicht, wozu das gut sein soll: Wieso sollte jemand drei NOR-Gates verwenden, um *ein* AND-Gate zu ersetzen? Die Antwort darauf erfährst du im Exkurs über integrierte Schaltkreise (ICs).

Exkurs B – Integrierte Schaltkreise (ICs)

Logikgatter müssen nicht jedes Mal mit Transistoren aufgebaut werden. Wer für eine Schaltung ein paar AND-Gates oder auch komplexere digitale (oder analoge) Schaltungen wie Zähler, Decoder, Audioverstärker etc. braucht, kauft diese meistens fertig ein – und zwar in Form von *Integrierten Schaltkreisen*, kurz *IC* (für ***I**ntegrated **C**ircuits*). ICs sind meist schwarze Bausteine mit vielen Anschlüssen und sehen zum Beispiel so aus:

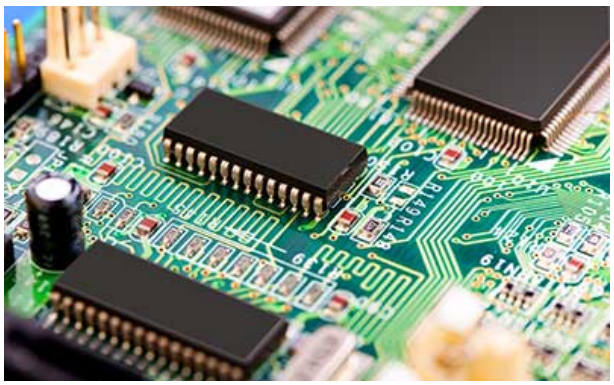

Bild 23 – ICs auf einer Leiterplatte

Die in diesen Bausteinen integrierte Schaltung ist auf einem viel kleineren Chip untergebracht:

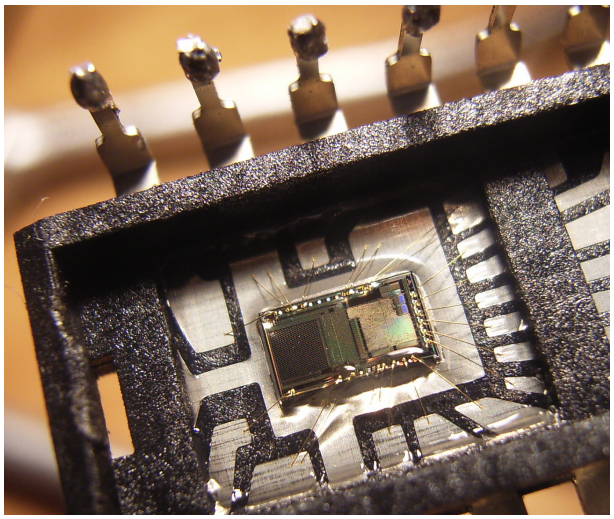

Bild 24 – Chip im Gehäuse eines IC

In Bild 24 siehst du, dass die einzelnen Anschüsse des Chips mit hauchdünnen Golddrähten auf die Anschlüsse des Gehäuses geführt sind. Das Gehäuse schützt den Chip und ermöglicht es, dass wir auf die einzelnen Anschlüsse zugreifen und den Chip in unsere Schaltung einbauen können.

Wie die Schaltung auf dem Chip im Detail aussieht, können wir fotografisch nicht erfassen, von einigen sehr einfachen ICs sind Teile des Chip-Designs bekannt, hier ein Beispiel:

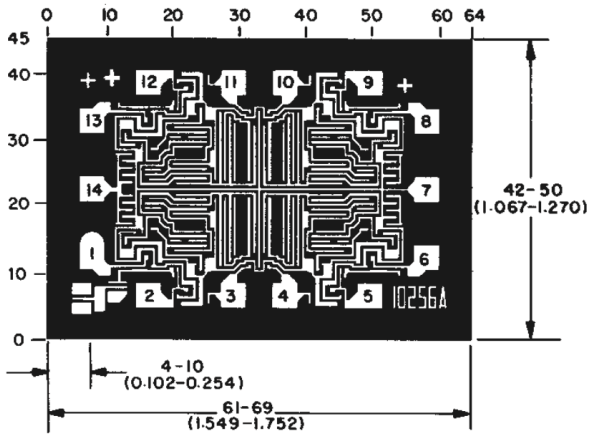

Bild 25 – Layout des IC *CD4011*, Maße in mil / (mm)

Bild 25 zeigt die Anschlüsse und Leitungen (weiss) auf dem Chip des ICs *CD4011*. Dieser IC enthält vier NAND-Gates. In den schwarzen Bereichen befinden sich Transistoren. Hier siehst du den Schaltplan *eines* dieser vier NAND-Gates:

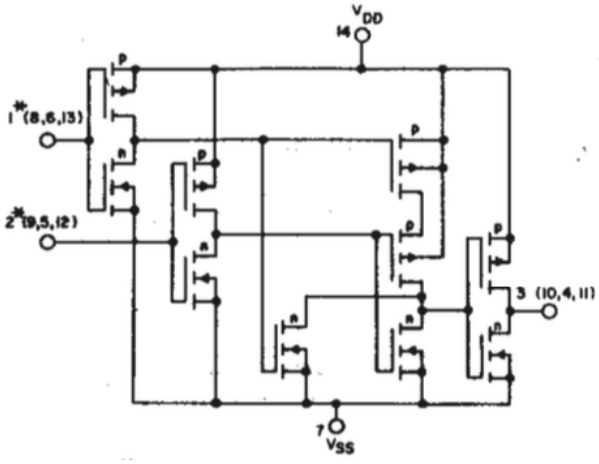

Bild 26 – Schaltplan eines NAND-Gate im IC *CD4011*

Im Bild 26 siehst du, dass ein NAND-Gate im *CD4011* aus 10 Feldeffekt-Transistoren (FETs) besteht. Für ein einfaches AND-Gate reichen eigentlich zwei Transistoren und für ein NOT-Gate reicht ein einziger Transistor. So sollte ein NAND-Gate doch mit drei Transistoren realisierbar sein – wieso sind hier so viele Transistoren verbaut?

CMOS

Die NAND-Schaltung im *CD4011* hat gegenüber einer einfacheren Schaltung einige Vorteile. Sie ist in der CMOS-Technologie ausgeführt. CMOS steht für ***C**omplementary **M**etal-**O**xide-**S**emiconductor*. Wichtig ist hier vor allem das Wort *komplementär* (*gegensätzlich*, *ergänzend*). Bei jedem Eingang sind zwei FETs komplementär geschaltet:

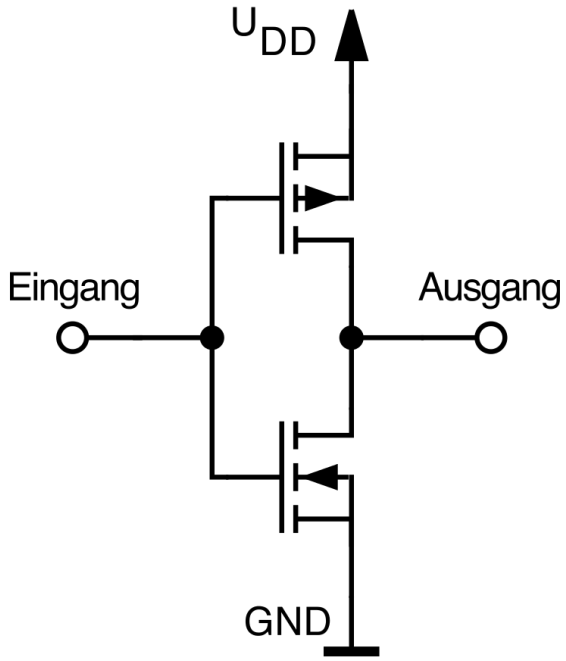

Bild 27 – CMOS-Inverter aus p- und n-Kanal-FET

Der obere Transistor in Bild 27 ist ein p-Kanal-, der untere ein n-Kanal-FET:

- Wenn am Eingang eine 1 (also eine positive Spannung) anliegt, dann sperrt der obere FET und der untere FET leitet. Der Ausgang ist dann 0, weil mit GND, also mit 0 V verbunden.
- Wenn am Eingang eine 0 (also keine Spannung) anliegt, dann leitet der obere FET und der untere FET sperrt. Der Ausgang ist dann 1, weil mit U_{DD}, also mit der Speisespannung verbunden.

In Bild 26 siehst du an beiden Eingänge des NAND-Gates eine solche Schaltung aus zwei komplementären FETs. Durch diese Schaltung besteht zwischen U_{DD} und GND stets ein sehr grosser Widerstand. Dadurch fliesst in der gesamten Schaltung fast kein Strom. Wenig Stromfluss bedeutet wenig Leistung und damit wenig Wärmeabgabe. Dies erlaubt eine hohe Schaltungsdichte: Es können auf geringen Raum extrem viele FETs untergebracht werden! Die komplementäre Anordnung der FETs macht die Schaltungen auch weitgehend resistent gegen Störsignale (Rauschen). Ausserdem erlaubt die CMOS-Technologie, dass die Schaltungen mit unterschiedlichen Speisespannungen zuverlässig und sehr schnell schalten. Darum wird CMOS in fast allen ICs verwendet – vom NAND-Gate bis hin zum Intel-Prozessor oder Kamera-Bild-Sensor.

Millionen NAND-Gates

Betrachte das Layout des CD4011 in Bild 25. Die Schaltung enthält vier NAND-Gates. Du kannst sehen, wo sich welches Gate befindet: Stell dir eine vertikale und eine horizontale Spiegelachse durch die Mitte vor und du erkennst vier gleiche Muster. Du erinnerst dich an die Gesetze von De Morgan und die daraus folgende Erkenntnis: Jede logische Schaltung lässt sich aus NAND-Gates (oder aus NOR-Gates) aufbauen. Hat eine Chip-Firma mal ein Layout (ein dreidimensionaler Bauplan aus unterschiedlichen Halbleiter- und Metalloxid-Schichten) für ein NAND- oder NOR-Gate entwickelt, so kann es daraus alle möglichen digitalen Schaltungen in einem Chip unterbringen: Der Bauplan wird einfach „kopiert“ und tausend- bis millionenfach „eingefügt“. De Morgans Gesetze sind so gesehen sehr nützlich. Jedoch müssen die Gates dann noch richtig miteinander verbunden werden!

Je dichter ein solches Gate gebaut ist, desto schneller die Schaltvorgänge und desto mehr Gates lassen sich im Chip unterbringen. Führende Halbleiterfirmen wie *Samsung*, *Intel oder TSMC* arbeiten daran, immer kleinere FETs zu realisieren und erreichen, dass die Abstände zwischen zwei Anschlüssen eines FETs nur noch 7 bis 10 nm betragen!

Teil 5 – Flipflops und Register

Mit Logikgattern können wir zwar viele Schaltungen bauen, etwa für die Steuerung von Verkehrsampeln oder für die Steuerung eines Fahrstuhls. Aber Schaltungen, die nur aus Logikgattern bestehen, können im Grunde nur dies: Für eine bestimmte Kombination aus *Einsen* und *Nullen* an den Eingängen setzen sie eine gewünschte Kombination aus *Einsen* und *Nullen* an den Ausgängen. Diese Art von Logik wird deshalb auch ***combinational logic*** genannt. In dem Moment, indem sich die Kombination der Eingänge ändert, ändert sich (eventuell) die Kombination der Ausgänge. Die *combinational logic* ist also zeitunabhängig, sie kennt kein Vorher und kein Nachher, denn sie kann sich nichts merken.

Schon beim Fahrstuhl ergibt sich hier ein Problem: Damit der Fahrstuhl zum richtigen Stockwerk gesteuert wird, müsste die Taste, die den Fahrstuhl ruft, solange gedrückt sein, bis der Fahrstuhl da und die Türe geöffnet ist. Obschon viele Leute gerne Fahrstuhltasten drücken (oft mehrmals, in der Hoffnung, dass der Fahrstuhl dann schneller kommt): Kaum jemand möchte eine Fahrstuhltaste so lange gedrückt halten. Und das ist auch nicht nötig, denn einmal gedrückt, leuchtet da ein Licht, das uns sagt, dass der Fahrstuhl gerufen ist. Die Steuerung merkt sich also, dass die Taste gedrückt wurde. Aber wie macht sie das?

Das RS-Flipflop (SR latch)

Wir suchen eine Schaltung, die am Ausgang einen Zustand (1 oder 0) auch dann noch hält, wenn dieser am Eingang schon nicht mehr anliegt. Die Schaltung soll den Zustand solange halten, bis ein weiterer Eingang ihr sagt, dass sie den Zustand löschen soll. Folgende Schaltung kann das:

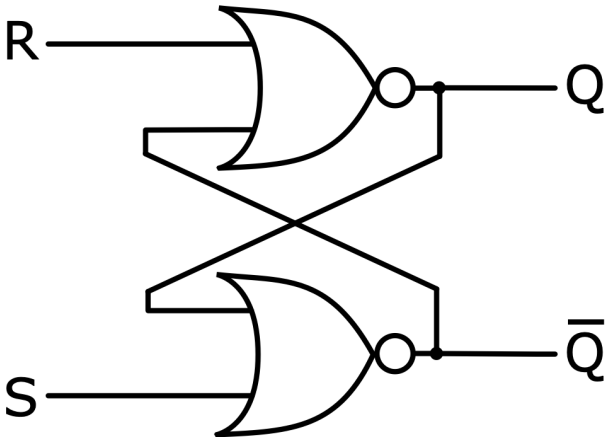

Bild 29 – ein RS-Flipflop aus zwei NOR-Gates

In Bild 29 sieht du die Schaltung eines RS-Flipflops. Sie hat zwei Eingänge, **R**eset und **S**et und zwei Ausgänge, Q und Q-nicht (¬Q). ¬Q soll natürlich stets den Wert haben, den Q nicht hat. Der besondere Trick dieser Schaltung ist das *Rückführen des Ausgangs auf den Eingang*. Zum Verständnis machst du dir folgende Überlegung: Der Ausgang eines der beiden **NOR-Gates** ist nur dann 1, wenn *beide* Eingänge 0 sind. In den anderen Fällen ist sein Ausgang 0:

		OR	NOR
A	B	A v B	$\overline{A \text{ v } B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Reset: Wenn **R = 1 und S = 0**, dann ist **Q = 0** (weil ja mindestens ein Eingang des oberen NOR-Gates 1 ist). Also sind beide Eingänge des unteren NOR-Gates 0. Also ist **¬Q = 1**.

Set: Wenn **R = 0 und S = 1**, dann ist **Q = 1** und **¬Q = 0** (alles komplementär zu Reset).

Hold: Wenn **R = S = 0**, dann kommt es darauf an, welcher Zustand an den Ausgängen liegt: Wenn Q = 0 war, bleibt es 0, wenn Q = 1 war, bleibt es 1.

Hier siehst du die vier möglichen Zustände der Schaltung:

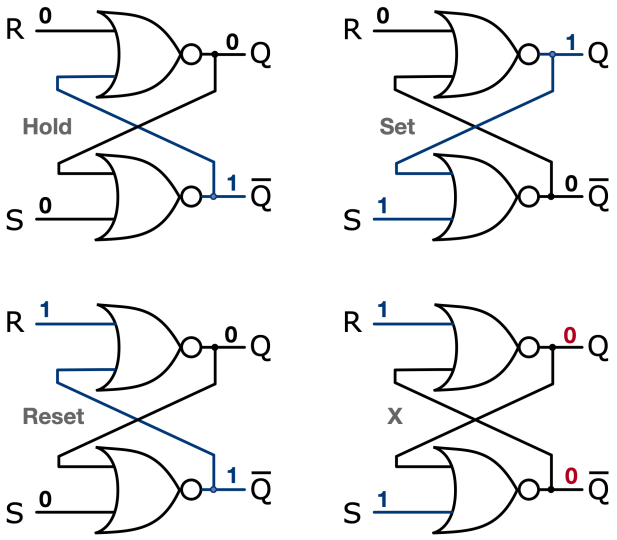

Bild 30 – vier Zustände des RS-Flipflop

Der letzte Zustand X ist nicht erlaubt, denn in diesem Fall wären Q und Q-nicht gleichwertig. Die Wahrheitstabelle des RS-Flipflops sieht so aus:

Zustand	R	S	Q
Hold	0	0	Q
Set	0	1	0
Rest	1	0	1
X	1	1	–

So einfach also: Die Taste, die den Fahrstuhl ruft, könnte auf den S-Eingang geführt werden. Q bliebe auch dann noch 1, wenn die Taste losgelassen wird und S auf 0 fällt. Erst wenn der Reset-Eingang 1 wird, wird Q wieder auf 0 gesetzt. Alles klar? Sonst erklärt dieses Video das RS-Flipflop von A bis Z.

Das D-Flipflop (D latch)

Das mit dem verbotenen Zustand X beim RS-Flipflop ist etwas unschön. Das könnte man noch verbessern, zum Beispiel mit dieser Schaltung:

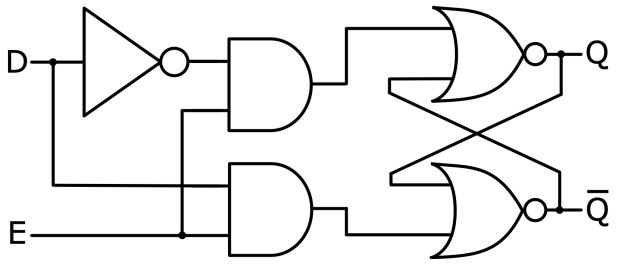

Bild 31 – D-Flipflop mit Enable-Eingang

Der rechte Teil der Schaltung in Bild 31 entspricht der Schaltung des RS-Flipflop. Auf der linken Seite ist ein *Daten-Eingang D* einmal direkt und einmal invertiert (über ein NOT-Gate) auf zwei AND-Gates geführt. Diese AND-Gates dienen als „Schleusen“. Wenn der *Enable-Eingang E* = 0 ist, sind die Ausgänge der AND-Gates natürlich 0 – egal was D ist. Das heisst: Wenn E = 0 ist, ist das Flipflop im Hold-Zustand. Wenn E = 1 ist, dann ist das Flip-Flop abhängig von D entweder im Set- oder im Reset-Zustand:

Zustand	E	D	Q
Hold	0	0	Q
	0	1	Q
Rest	1	0	0
Set	1	1	1

Dieses Video erklärt das D-Flipflop (engl. *D latch*) nochmals Schritt für Schritt.

Das flankengesteuerte D-Flipflop

Flipflops können natürlich mehr als nur für die Ruftasten in Fahrstühlen verwendet werden. Du ahnst wohl bereits: Mit einem Bauteil, das einen Zustand speichern kann, lassen sich Bauteile bauen, die viele Zustände speichern können. Klar. Und deshalb gibt es Flipflops auch als fertige ICs. Folgend siehst du die Symbole des ‘normalen’ und des flankengesteuerten D-Flipflops:

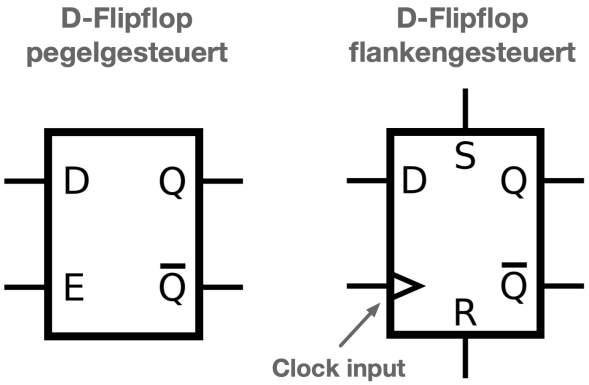

Bild 32 – zwei unterschiedliche D-Flipflops

Beim pegelgesteuerten Flipflop wird der Wert am Eingang *D* jeweils an den Ausgang *Q* übernommen, solange der Enable-Eingang 1 ist. Beim flankengesteuerten Flipflop ist das anders: Der

Wert, der am Daten-Eingang anliegt, wird *genau dann* an den Ausgang übernommen, wenn der Clock-Eingang von 0 auf 1 wechselt. Also zu einem *bestimmten Zeitpunkt*. Hier siehst du die Funktion des flankengesteuerten D-Flipflops im Zeit-Diagramm:

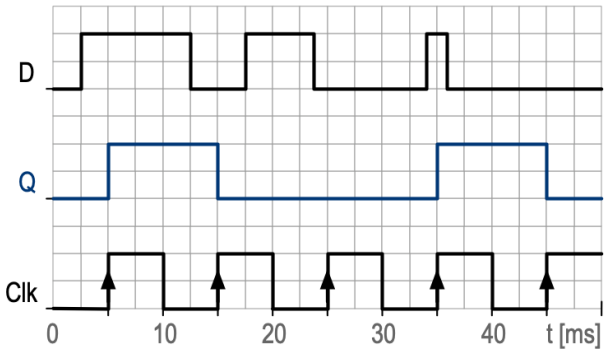

Bild 33 – Timing diagram eines D-Flipflops

Bei jeder *positiven Flanke* des Signals *Clk* (durch Pfeile gekennzeichnet), wird der Wert an *D* (0 oder 1) an den Ausgang *Q* übernommen. Diese Fähigkeit, eine Aktion zu einem genauen Zeitpunkt auszuführen, ist für die Datentechnik essentiell. Wie bereits in Teil 1 erwähnt, verfügt jede CPU über ein Clock-Modul, das den Takt angibt: Mit jeder positiven Flanke wird ein weiterer Befehl ausgeführt. In diesem Video ist das D-Flipflop ausführlich erklärt.

Register

Wenn du 4 D-Flipflops parallel schaltest, erhältst du ein 4-Bit-Register. Damit kannst du eine 4-stellige Binärzahl speichern. Diese Schaltung ist im IC *74LS173* enthalten. Der IC hat folgendes Symbol:

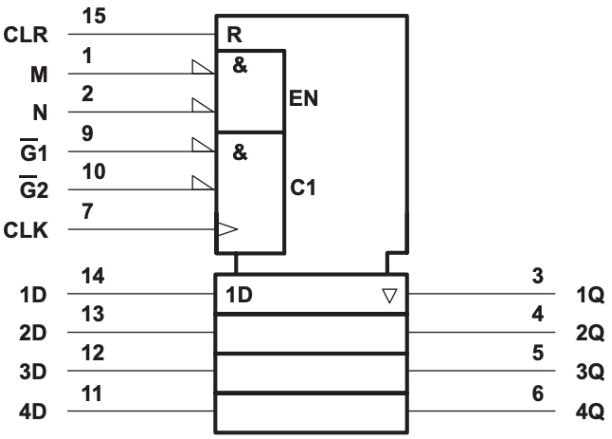

Bild 34 – Symbol des 4-Bit-Registers *74LS173*

Das Symbol in Bild 34 besteht aus einem oberen und einem unteren Teil: Der untere Teil, die vier Zellen mit den Anschlüssen *1D…4D* und *1Q…4Q*, umfasst die vier Flipflops mit den Daten-Ein- und Ausgängen. Die Anschlüsse am oberen Teil (*CLR* etc.) sind Steuerleitungen. Hier die Logikschaltung:

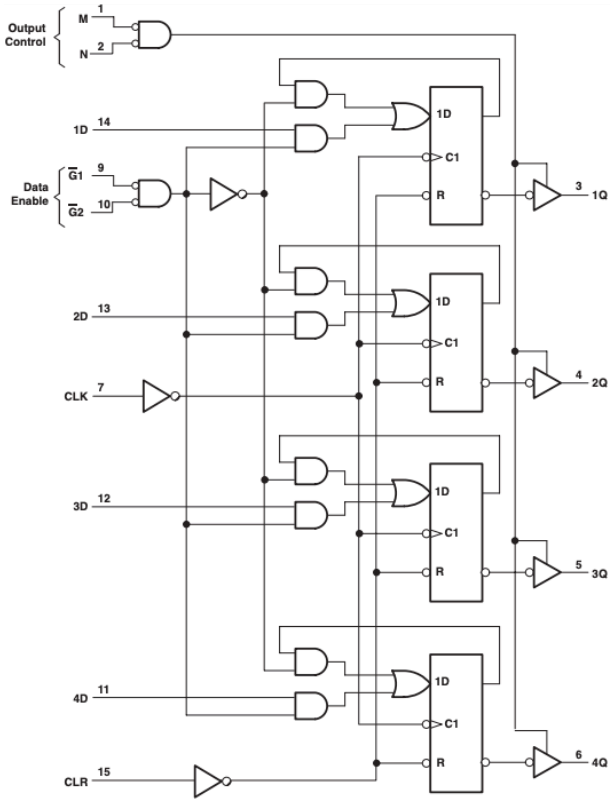

Bild 35 – logic diagram des 4-Bit-Registers *74LS173*

Du musst an dieser Stelle nicht die ganze Schaltung verstehen. Aber du kannst zum Beispiel sehen, dass das Clock-Signal CLK auf jedes Flipflop geführt ist. Mit der positive Flanke dieses Signals laden alle vier Flipflops den Wert, der am Daten-Eingang anliegt, an den Ausgang. Register wie dieses gibt es viele, auch mit 8 und mehr Bits.

Serielle Datenübertragung

Mit dem eben gesehenen Register *74LS173* kannst du „auf einen Schlag“ 4 Ausgänge setzen. Natürlich musst du dazu auch die vier Eingänge auf einmal setzen können. Was aber, wenn du nur einen Eingang auf einmal setzen kannst? In digitalen Geräten wird immer mehr Elektronik auf immer kleinerem Raum untergebracht. Es gibt heute viele Mikrocontroller, die mit wenigen

Anschlüssen auskommen (so etwa der ATtiny85) und dennoch Displays und andere Hardware ansteuern können. Das ist nur möglich, wenn über *wenige* Leitungen viele Daten (Einsen und Nullen) übertragen werden können. Wenn du Daten über ein USB-Kabel auf den Computer lädst, werden die Daten nicht gleichzeitig über viele parallele Leitungen übertragen, sondern *nacheinander* – wir sagen auch *seriell* (USB steht für *Universal Serial Bus*). Daten werden also oft seriell übertragen, an bestimmten Stellen ist es aber nötig, dass sie gleichzeitig, also *parallel* bereitstehen. Hier sind Module nötig, die serielle Daten in parallele Daten wandeln können, zum Beispiel:

Schieberegister (shift registers)

Die Idee des Schieberegisters ist im Namen enthalten: Man *schiebt* Einsen und Nullen durch das Register. Hier siehst du eine allgemeine Logik-Schaltung eines 4-Bit-Schieberegisters:

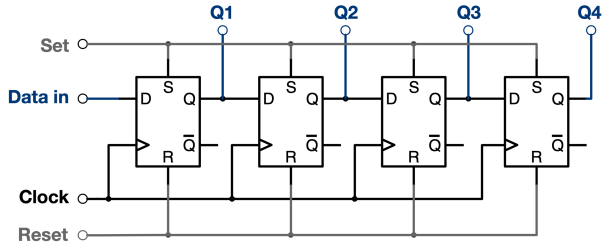

Bild 36 – Logikschaltung eines 4-Bit-Schieberegisters

Angenommen, am Anschluss *Data in*, liegt eine 1 an. Sobald das Clock-Signal von 0 auf 1 wechselt, liegt die 1 am Ausgang Q1 an – und damit am Eingang des zweiten Flipflops von links.

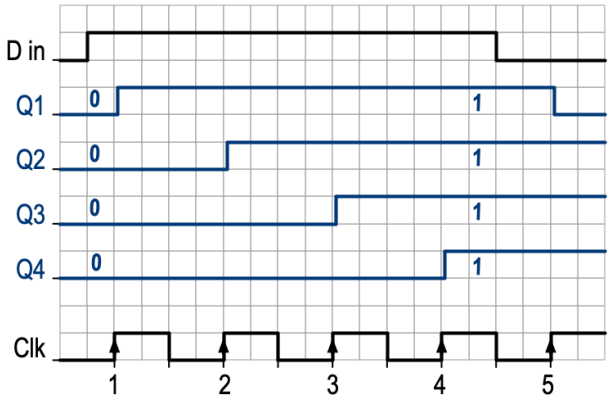

Bild 37 – Zeitdiagramm eines 4-Bit-Schieberegisters

Nach vier positiven Flanken des Clock-Signals ist die 1 bei Q4 angelangt. Und sofern *Data in* die ganze Zeit 1 war, sind nun alle vier Ausgänge 1. Mit den Signalen *Set* und *Reset* in Bild 36 könntest du auf einmal alle Ausgänge auf 1 oder 0 setzen.

Eine kleine Schieberegister-Anwendung

Angenommen, du schreibst ein Programm für ein kleines Mikrocontroller-Modul (das hier) und willst eine Zahl auf einer 7-Segment-Anzeige anzeigen. Dafür bräuchtest du 8 digitale Ausgänge. Du hast aber nur noch 2 digitale Ausgänge (die anderen sind schon in Verwendung). Die Lösung:

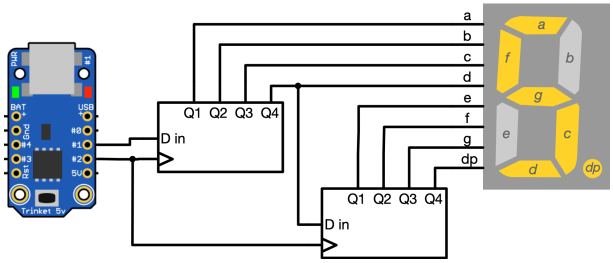

Bild 38 – Schieberegister vervielfachen Ausgänge

Mit der Schaltung in Bild 38 kannst du alle 8 Segmente (inkl. Punkt *dp*) der 7-Segment-Anzeige mit bloss 2 Ausgängen des Mikrocontrollers ansteuern: Um die Zahl *'5.'* anzuzeigen, sollen auf der Anzeige alle Segmente ausser *b,* und *e* leuchten. Also soll an allen Anschlüssen ausser *b* und *e* eine 1 anliegen – also die Zahlenfolge *10110111*. Dein Programm muss dazu nur den Ausgang, der zu *D in* führt, gemäss dieser Zahlenfolge auf 1 oder 0 setzen und dazwischen jeweils den Ausgang, der zum *Clock-Eingang* führt, von 0 auf 1 und wieder auf 0 setzen. Nach 8 Clock-Impulsen steht da die Zahl 5. Jedoch: Während der Ausführung deines Programms entstehen ein paar merkwürdige Zeichen auf der Anzeige, da jederzeit angezeigt wird, was an den Ausgängen der beiden Flipflops anliegt. Besser wäre es, wenn du zuerst alle Ausgänge innerhalb des Schieberegisters setzen und dann alle gleichzeitig nach aussen *freigeben* könntest. Mehr dazu im nächsten Teil.

Teil 6 – Null, Eins und Z auf dem Datenbus

Schieberegister gibt es in unterschiedlichen Varianten. Schau dir das Datenblatt des 8-Bit-Schieberegisters 74HC595 an. Hier siehst du auf Seite 3, dass die Ausgängen Q_A bis Q_H über den Eingang *OE* (output-enable) *freigegeben* werden können. Wir erweitern die Schaltung in Bild 38 und nutzen diesen Eingang:

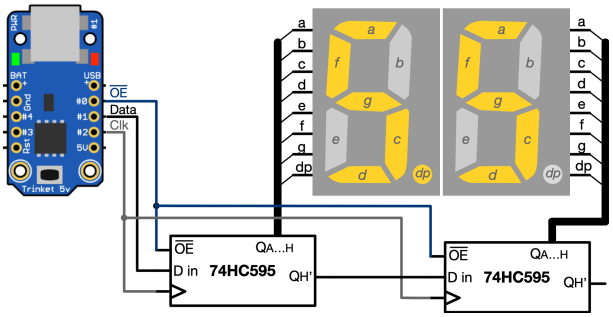

Bild 39 – Schieberegister mit *output-enable OE*

In der Schaltung in Bild 39 führen drei Signale vom Mikrocontroller-Modul weg: *Clock*, *Data* und *OE*. Der Strich über OE sagt uns, dass das Signal *active low* ist, das heisst: Wenn das Signal 0 ist, dann ist das Signal aktiv; sprich dann sind die Ausgänge freigegeben (*enable* bedeutet hier *freigeben*). Wenn es 1 ist, sind die Ausgänge nicht freigegeben.

Damit die beiden Anzeigen die Zahl '*5.5*' anzeigen, muss an den Ausgängen $Q_A \ldots Q_H$ der beiden Schieberegister die Zahlenfolge *10110111* bzw. *10110110* anliegen. Das Problem mit den merkwürdigen Zeichen während des Bit-Schiebens kannst du jetzt umgehen: Dein Programm muss erst das Signal *OE* auf 1 setzen – nun sind die Ausgänge der Schieberegister nicht freigegeben und das Programm kann die Zahlenfolge rausschieben (*Clk* wechselt 16 mal von 0 auf 1), ohne dass irgendwas angezeigt wird. Schliesslich wird OE auf 0 gesetzt, und die Anzeigen leuchten auf.

Two-State vs. Tri-State Outputs

Nun fragst du dich vielleicht, was *freigeben* genau bedeutet: Wenn ein Ausgang nicht freigegeben ist, ist er dann 1 oder 0? Die Antwort: Er ist weder 1, noch 0, er ist *Z*. Dieser dritte Zustand ist datentechnisch zwar unbedeutend, aber hardwaretechnisch sehr wichtig. Um dies zu verstehen, rufen wir uns in Erinnerung, wie ein normaler digitaler Ausgang aussieht:

Digitale Ausgänge (two-state)

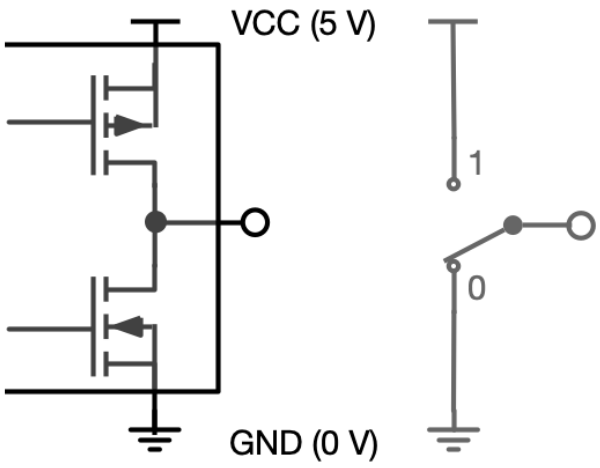

Bild 40 – Digitaler CMOS-Ausgang (vereinfacht)

In Bild 40 siehst du links den prinzipiellen Aufbau eines CMOS-Ausgangs, rechts eine Repräsentation dieser Schaltung mit normalem Schalter: Entweder ist der Ausgang mit GND (Minuspol der Spannungsquelle) oder mit VCC (Pluspol der Spannungsquelle) verbunden. Deshalb kannst du zwei digitale Ausgänge nicht einfach verbinden. Dazu ein Beispiel einer Logikschaltung:

Eine LED soll leuchten, wenn die Eingänge A und B = 1 sind. Sie soll auch dann leuchten, wenn die Eingänge B und C = 1 sind. Dafür scheint folgende Logikschaltung dem ungeschulten Auge auf den ersten Blick sinnvoll:

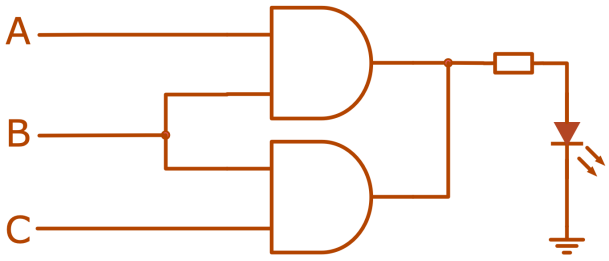

Bild 41 – Fehlerhafte Logikschaltung

Das Problem bei der Schaltung in Bild 41 sollte dir nun klar sein: Wenn das obere AND-Gate eine 1 ausgibt (weil A und B = 1 sind) und das unter AND-Gate eine 0 (weil C = 0 ist): Dann ist der Ausgang des oberen Gates mit dem Pluspol und der des unteren Gates mit dem Minuspol der Spannungsquelle verbunden. Da hier beide Ausgänge miteinander verbunden sind, heisst das: Pluspol und Minuspol der Spannungsquelle sind miteinander verbunden. Dieser Situation sagen wir **Kurzschluss**. Wenn du irgendwo in deiner Schaltung einen Kurzschluss hast, funktioniert nichts mehr, weil jeglicher Strom durch diesen Kurzschluss fliesst. Entweder bricht dann die Spannung zusammen oder es fliesst so lange sehr viel Strom, bis die Gates zerstört sind. Die richtige Logikschaltung sieht natürlich so aus:

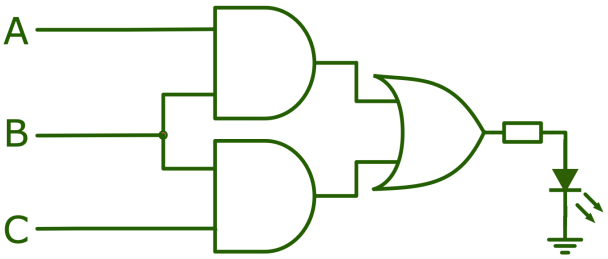

Bild 42 – Korrekte Logikschaltung

Tri-State-Ausgänge

Schau dir Bild 1 auf der ersten Seite des Skripts an. Das Blockschaltbild der 8-Bit-CPU zeigt, dass alle Module über eine dicke Linie miteinander verbunden sind. Diese Linie steht für 8 parallele Leitungen; für den 8-Bit-Datenbus. Einige Module können sowohl vom Bus lesen als auch auf den Bus schreiben. Diese Module haben also sowohl 8 Eingänge, die mit dem Datenbus verbunden sind, als auch 8 Ausgänge, die mit dem Datenbus verbunden sind. Das bedeutet *erstens*, das Eingänge mit Ausgängen verbunden sind und *zweitens*, dass die Ausgänge mehrere Module miteinander verbunden sind. Wie kann das funktionieren?

Du ahnst es schon: Das funktioniert mit Ausgängen die nicht nur 1 oder 0, sondern auch Z sein können.

Betrachte erneut Bild 35, das die Logikschaltung des 4-Bit-Registers zeigt. Dort siehst du, dass zwischen den Ausgängen der (internen) Flipflops und den wirklichen Ausgängen des Registers (*1Q*…*4Q*) noch Elemente geschaltet sind, die aussehen wie NOT-Gates. Das sind *Tri-State-Inverter*:

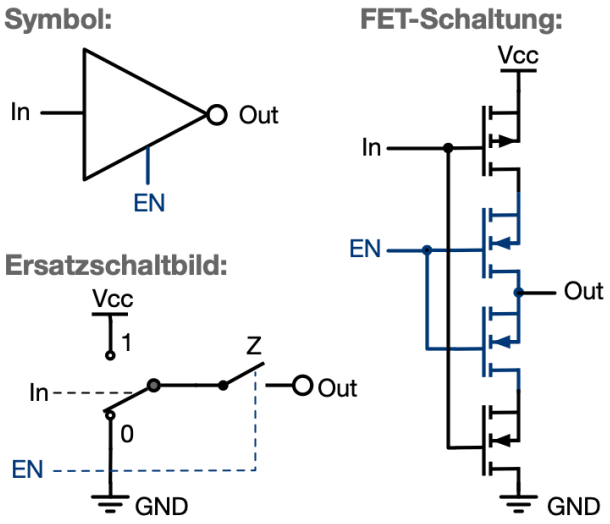

Bild 43 – Tri-State-Inverter

Ein Tri-State-Inverter hat zwei Eingänge (*In* und *EN*) und einen Ausgang. Der Ausgang kann 0, 1 oder Z sein. Z heisst, dass der Ausgang weder mit dem Pluspol (Vcc), noch mit dem Minuspol (GND) der Spannungsquelle verbunden ist.

Das Ersatzschaltbild in Bild 43 zeigt: Wenn der Schalter rechts auf Stellung Z ist, dann ist der Ausgang weder mit GND, noch mit Vcc verbunden. In der FET-Schaltung siehst du, wie das funktioniert: Wenn EN = 0 ist, dann sperren die beiden FETs in der Mitte, der Ausgang ist dann von Vcc und GND getrennt. Neben Tri-State-Invertern gibt es auch Tri-State-Buffer. Diese invertieren das Eingangssignal nicht. Hier die Wahrheitstabellen:

EN	In	Out		EN	In	Out
1	0	1		1	0	0
1	1	0		1	1	1
0	0	Z		0	0	Z
0	1	Z		0	1	Z

Eine Schritt-für-Schritt-Erklärung zum Datenbus und zu Tri-State-Ausgängen findest du in diesem Video.

Eine kleine Datenbus-Anwendung

Wenn ein digitaler Baustein Tri-State-Ausgänge hat, dann ist er "busfähig", das heisst: dann kannst du seine Ausgänge mit den Ausgängen anderer 3-State-Bausteine verbinden. Über entsprechende Signale – meistens heissen sie *Output Enable* – kannst du dann steuern, dass immer nur einer der Bausteine freigegeben ist. Also: immer nur ein Baustein setzt seine Ausgänge auf 1 oder 0 und definiert damit die Werte auf dem Bus – die anderen setzen ihre Ausgänge auf Z – sie nehmen, bildlich gesprochen, ihre Finger weg vom Bus.

Das bereits bekannte Schieberegister *74HC595* hat Tri-State-Ausgänge. Du könntest also die Ausgänge mehrerer solcher Register zusammenschliessen. Aber wozu? Vielleicht könntest du mit jeder Bus-Leitung eine LED verbinden. Wenn nun jedes Schieberegister einen anderen Wert gespeichert hätte, dann würden die LEDs immer den Wert des gerade freigegebenen Schieberegisters anzeigen. Damit liesse sich ein bewegliches Lichtmuster programmieren!

Hier die (vereinfachte) Schaltung:

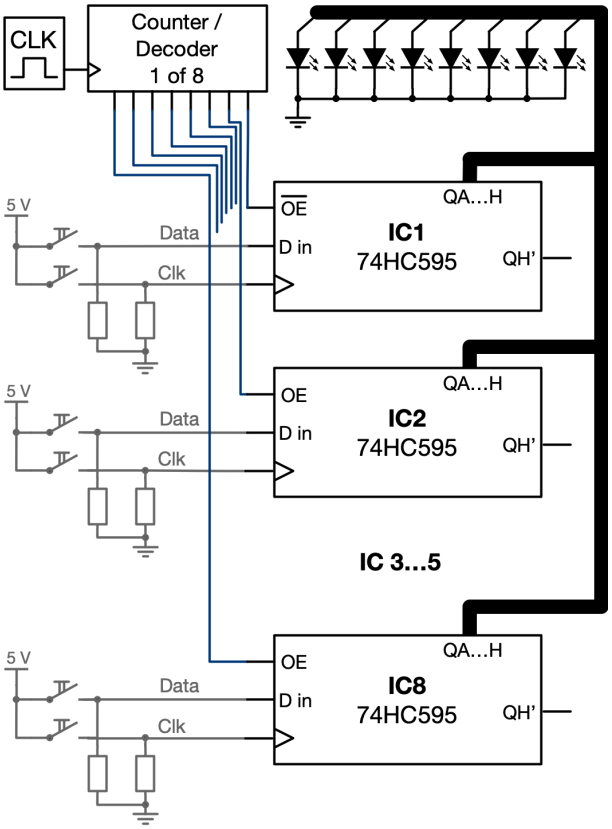

Bild 44 – Programmierbares Lichtmuster

Oben links in Bild 44 siehst du einen Clock-Generator und einen Zähler/Decoder. Der Clock-Generator gibt eine stets zwischen 0 und 1 wechselnde Spannung aus – ein Clock-Signal. Dieses ist auf einen Zähler geführt, der bei jeder positiven Flanke um 1 hoch zählt. (Auch einen Zähler kann man aus Flipflops bauen, doch dazu später). Sobald der Zähler bei 7 angelangt ist, wechselt er wieder zu 0. Der im Zähler integrierte Decoder setzt bei jeder Zahl von 0…7 einen anderen Ausgang auf 0 – die übrigen Ausgänge sind auf 1. Diese Ausgänge sind mit den *Output-Enable*-Eingängen der Schieberegister IC1…IC8 verbunden (blaue Leitungen, IC3 bis 5 sind aus Platzgründen nicht gezeichnet). Also wird mit jedem Clock ein anderes Schieberegister freigegeben. Jedes Schieberegister ist von Hand programmierbar: Mit zwei Tasten kannst du die Eingänge *Din* und *Clk* setzen und so das Schieberegister mit Daten füttern. Die Ausgänge der Schieberegister sind alle zu einem 8-Bit-Datenbus verbunden (dicke Leitung). Mit jeder Bus-Leitung ist eine LED verbunden, die leuchtet, wenn der Wert auf der Leitung 1 ist. Du könntest die Schieberegister so mit Einsen und Nullen füllen, dass die LEDs wie folgt leuchten:

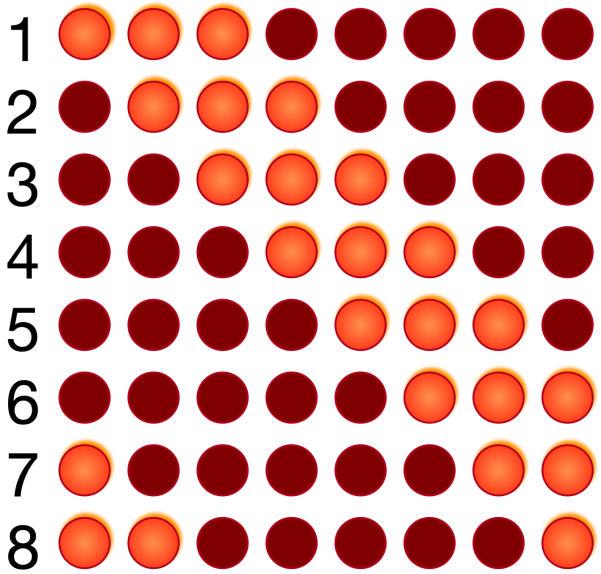

Bild 45 – Lauflicht-Muster

Drei LEDs würden nun ständig von links nach rechts laufen.