EF Informatik / *Elektronik und Digitaltechnik*

Vom Transistor zur CPU

Teil 1 – Eine einfache Central Processing Unit

Die **C**entral **P**rocessing **U**nit, auch „Prozessor“ genannt, ist das Herzstück jedes Computers. Moderne Computer verfügen bereits über mehrere CPUs oder enthalten Prozessoren mit mehreren Kernen (*Multicore-Prozessoren*). Ein Kern ist sozusagen eine CPU innerhalb einer CPU. Bild 1 zeigt das Blockschaltbild einer sehr einfachen 8-Bit-CPU. Es besteht aus mehreren Modulen, die über

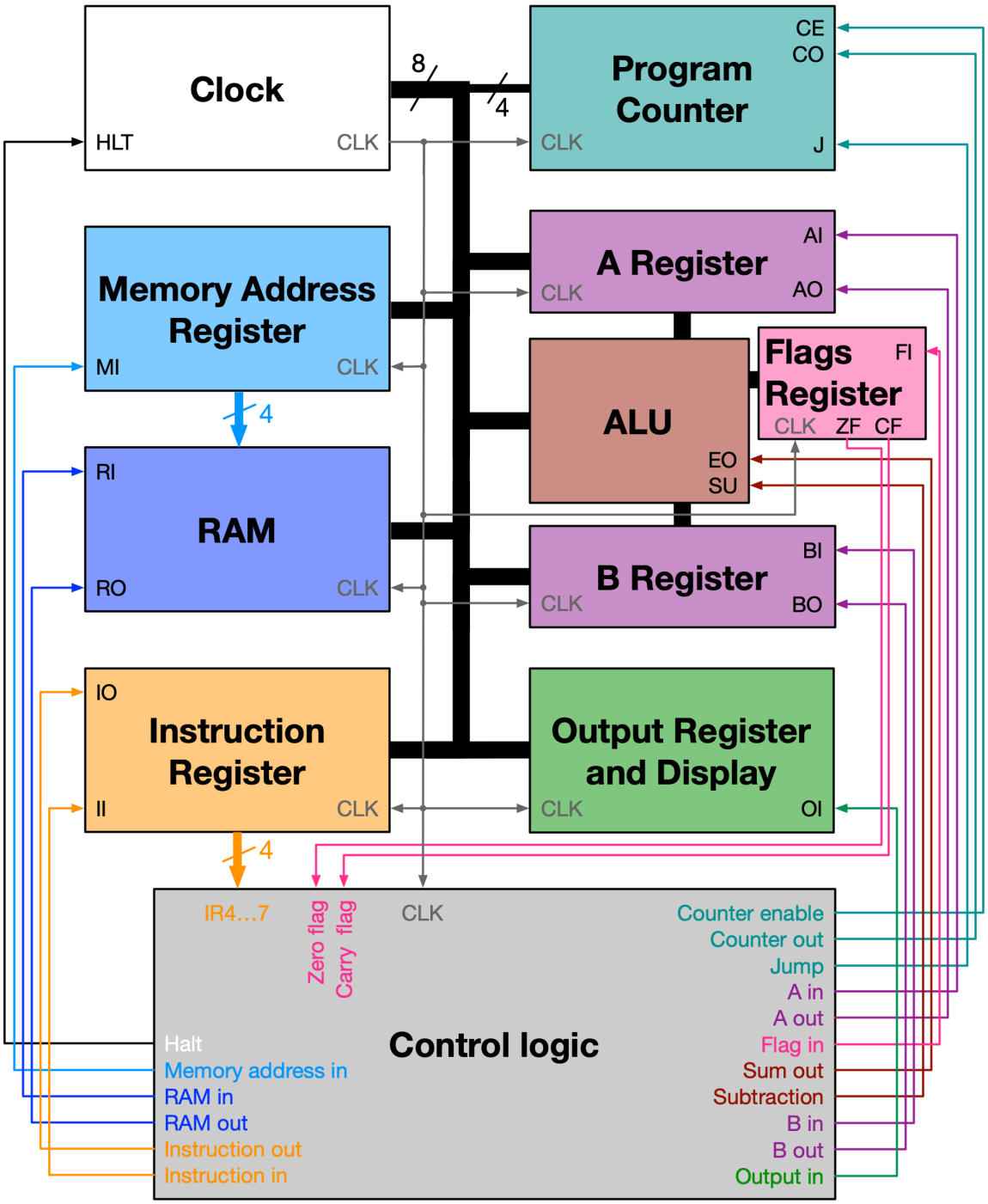


Bild 1 – Blockschaltbild der 8-Bit-CPU nach dem Design von Ben Eater.

Leitungen miteinander verbunden sind. Dieses CPU-Design stammt von Ben Eater. Auf seiner Webseite eater.net/8bit findest du Videos und Beschreibungen, die den Aufbau und die Funktionsweise jedes Moduls ausführlich erklären. Für eine Übersicht betrachten wir kurz die wichtigsten Module und ihre Aufgaben:

Der Bus besteht aus 8 parallelen Leitung, die in Bild 1 als eine dicke Linie gezeichnet sind. Der Bus ist quasi die Daten-Autobahn: über ihn tauschen die Module Daten aus. Es können gleichzeitig **8 Bits** übertragen werden. „Bit" steht für „**Bi**nary Digi**t**" (etwa: „binäre Ziffer"). Ein Bit kann zwei Werte haben; 0 oder 1.

Wert	Bezeichnungen	Spannung in 5-V-Logik
0	LOW, false	0 V
1	HIGH, true	5 V

Angenommen, die acht Bus-Leitungen haben die Werte 0, 1, 0, 1, 0, 1, 0, 1, so würden wir abwechslungsweise die Spannungen 0 V und 5 V messen. Zusammen bilden die Leitungen die binäre Zahl 0101010101. Das entspricht der Zahl 42, doch dazu später.

Das **Clock**-Modul generiert ein Rechteck-Signal, das dazu dient, die Module zeitlich zu steuern:

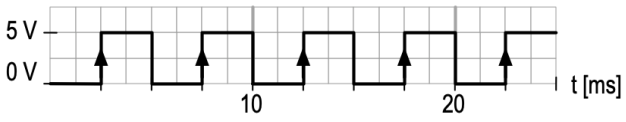


Bild 2 – Clock-Signal

Das Clock-Signal in Bild 2 wechselt alle 2.5 Millisekunden zwischen 0 V und 5 V. Alle 5 Millisekunden erfolgt eine positive Flanke, also ein Wechsel von 0 auf 5 V – hier durch die Pfeile bezeichnet. Wenn ein Modul eine Anweisung ausführt (wenn es zum Beispiel Daten auf den Bus schreibt), dann tut es das exakt zum Zeitpunkt der positiven Flanke des Clock-Signals. So wird erreicht, dass alle Anweisungen zeitlich gesteuert und damit in der richtigen Reihenfolge ausgeführt werden. Die Clock gibt den Takt an. Die Taktfrequenz in Bild 2 beträgt 1/0.005 s = 200 Hz. Eine CPU mit einer Taktfrequenz von 200 Hz kann 200 Anweisungen pro Sekunde ausführen – alle 5 ms eine Anweisung.

Der **Program Counter** ist ein einfacher Zähler. Er sendet über den Bus eine vierstellige Binärzahl, die er stets um 1 erhöht. Was passiert mit dieser Zahl?

Wenn du am Computer ein Programm schreibst – zum Beispiel in *python* oder in *C* – dann wird dieser Code irgendwann übersetzt in sogenannten *Maschinencode*. Dieser enthält eine Reihe von Befehlen und besteht nur aus *Einsen* und *Nullen*. Die Zahl vom Program Counter gibt die Position des nächsten Befehls an. Wenn der Counter hoch zählt, wird ein Befehl nach dem anderen ausgeführt. Es ist auch möglich, den Counter auf eine bestimmte Zahl zu setzen, um auf eine bestimmte Stelle im Programm zurück- oder vorzuspringen. Dadurch werden Schleifen (z. B. *for)* oder Verzweigungen (z. B. *if-then-else*) möglich.

Das **RAM** (Random Access Memory) speichert Daten – unter anderem den Maschinencode, der ausgeführt werden soll. Bei unserer CPU beträgt die Kapazität des Speichers 16 Byte. Ein Byte enthält 8 Bits. Wir können uns ein RAM als Tabelle vorstellen – hier mit 16 Zeilen und 8 Spalten:

	Byte							
Adresse	**Bit0**	**Bit1**	**Bit2**	**Bit3**	**Bit4**	**Bit5**	**Bit6**	**Bit7**
'0000'	0	0	0	0	0	0	0	0
'0001'	0	0	0	0	0	0	0	0
'0010'	0	0	0	0	0	0	0	0
…	…	…	…	…	…	…	…	…
'1111'	0	0	0	0	0	0	0	0

Oben sind die ersten drei und die letzte Zeile dargestellt. Jede Zeile hat eine eigene Adresse, von

0 (binär: 0000) bis 15 (binär 1111); unter jeder Adresse können 8 Bits oder eben 1 Byte gespeichert werden. Im Moment ist unter jeder Adresse der Wert 0000'0000 abgespeichert.

Das **Memory address register** speichert die vierstellige RAM-Adresse und gibt diese an das RAM weiter.

Die **Register A und B** dienen als kleine Zwischenspeicher. Sie können einen Wert vom Bus speichern und später an die ALU weitergeben oder wieder auf den Bus zurückschreiben. In grösseren Prozessoren werden solche Register auch „general purpose registers", also etwa „Mehrzweck-Register" genannt, da sie unterschiedlichen Zwecken dienen können.

Die **ALU** (Arithmetic logic unit) ist das Rechenwerk der CPU. Unsere CPU kann aber nur dies: Die Zahlen aus den Register A und B entweder addieren oder subtrahieren und das Resultat auf den Bus schreiben – das ist alles.

Das **Flags Register** speichert Flags. Ein Flag ist meistens ein 1-Bit-Speicher und kann entweder den Wert 1 oder 0 haben. Unsere CPU verfügt über zwei Flags, das *Zero-Flag* und das *Carry-Flag*. Das Zero-Flag zeigt an, ob das Ergebnis der Rechenoperation der ALU = 0 ist. Wenn das Ergebnis 0 ist, dann ist das Zero-Flag 1; in allen anderen Fällen ist es 0. Das Carry-Flag zeigt an, ob die Rechenoperation einen Übertrag ergab.

Das **Instruction register** speichert den Befehlscode für den Befehl, der als nächstes ausgeführt werden soll. In unserer CPU besteht ein Befehlscode aus einer vierstelligen Binärzahl. Wir haben bereits gesehen, dass mit vier Binärstellen maximal 16 Zustände möglich sind. Das heisst, unsere CPU kann maximal 16 verschiedene Befehle ausführen. Der vierstellige Befehlscode wird zur Control logic weitergeleitet.

Die **Control logic** führt die Befehle aus. Ein Befehl (engl.: *instruction*) steht für eine Reihe von Anweisungen (engl.: *micro instructions*). Das funktioniert etwa so:

Angenommen, der Code, der vom Instruction register kommt, lautet: 0011. Dieser Code steht zum Beispiel für den Befehl „Schreibe das Resultat aus der ALU ins RAM". Nun muss die Control Logic aufgrund dieses Codes einige Anweisungen ausführen: Sie muss zuerst die eigene Instruktion laden, dann muss sie dem Memory address register mitteilen, dass es die Adresse vom Bus lesen muss, welche bestimmt, wo im RAM das Resultat abgespeichert wird. Danach muss sie der ALU mitteilen, dass sie das Resultat auf den Bus schreiben soll und schliesslich muss sie dem RAM sagen, dass es dieses Resultat vom Bus lesen soll. Die Control Logic kann also allen Modulen sagen, was sie tun und lassen sollen – und das tut sie über die Steuerleitungen, die du im Bild 1 siehst.

Das **Output register** schliesslich speichert den Wert, der am Display angezeigt werden soll. Aus Sicht des Menschen ist das sehr wichtig, denn all das Herumschieben und Rechnen in der CPU wäre für uns sinnlos, wenn nicht irgendwann etwas angezeigt würde – wir könnten damit so viel tun wie mit einem Computer ohne Bildschirm.

Was unserer CPU aber fehlt, was sonst alle Computer haben, ist ein Input-Modul, das es erlaubt, auch Daten einzugeben – etwa mit einer Tastatur. Es wäre bei der bestehenden Architektur aber einfach, noch ein Input-Modul an den Bus anzuschliessen und die Control-Logic um Steuerleitungen für dieses Modul zu erweitern.

Im **Vergleich mit modernen Prozessoren** fehlt unserer CPU aber einiges mehr, zum Beispiel:

- Der Aufbau moderner Prozessoren ist sehr viel komplexer als die in Bild 1 gezeigte. Sie verfügen über ein Vielfaches an Modulen und Registern.
- Die Busbreite in heutigen Prozessoren beträgt bis zu 64 Bit. Viele Prozessoren verfügen über getrennte Bussysteme für Daten und Adressen. Also beispielsweise ein 64-Bit-Datenbus und ein 40-Bit-Adressbus.

Bild 3 – Fotografie eines Intel-Pentium-4-Chips aus dem Jahr 2000. Der Chip ist etwa so gross wie ein 10-Rappen-Stück. Hier die Beschriftung der Bereiche.

- Die Taktfrequenz moderner Prozessoren liegt bei 2 bis 5 GHz. Deren Clock tickt also etwa 10 Millionen mal schneller als unsere Clock.
- Der Arbeitsspeicher (RAM), der modernen Prozessoren zur Verfügung steht, liegt im zwei-bis dreistelligen Gigabyte-Bereich und ist damit milliardenmal höher als bei unserer CPU. Prozessoren in heutigen Computern können ausserdem auf Festspeicher (FLASH etc.) zugreifen, auf denen das Betriebssystem und andere Programme permanent gespeichert sind. Das RAM unserer CPU ist kein Festspeicher, das heisst, dass sowohl das Programm als auch alle Daten gelöscht sind, sobald die Stromversorgung wegfällt. Bei nur 16 Bytes ist das nicht schlimm, die sind schnell wieder reingehackt.
- Die ALUs in modernen Prozessoren beherrschen nebst der Addition und Subtraktion auch Multiplikation, Division und einige logische Operationen – und das mit sehr viel grösseren bzw. komplexeren Zahlen als bei der 8-Bit-CPU, die nur mit Zahlen von 0 bis 255 rechnen kann.

Die grundlegende Funktionsweise und Architektur moderner Prozessoren sind aber nicht anders als bei unserer 8-Bit-CPU. Das heisst: Wenn du verstehst, wie diese 8-Bit-CPU funktioniert, dann hast du gute Voraussetzungen, heutige Prozessoren zu verstehen.

Ein Vorteil der 8-Bit-CPU liegt darin, dass wir sie selbst zusammenbauen können: Dazu brauchen wir Breadboards (die weissen, löchrigen Platten), einige integrierten Schaltungen (so genannte „ICs“, die schwarzen Bausteine), Drähte etc.

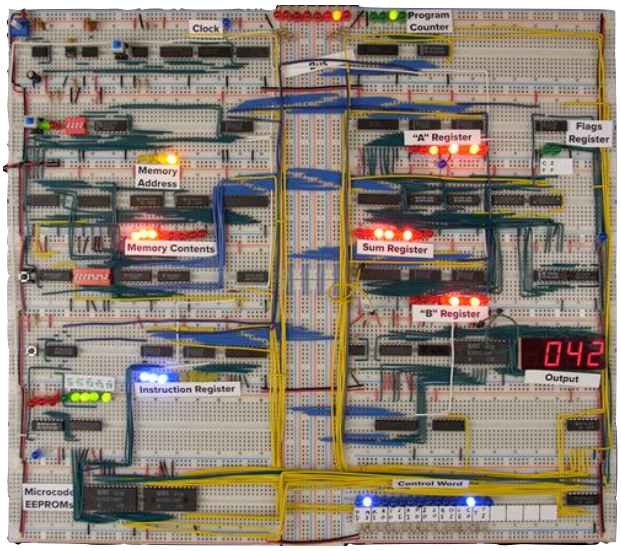


Bild 4 – Fotografie der aufgebauten 8-Bit-CPU. Der Aufbau 40 x 40 cm gross.

In diesem Kurs betrachten wir Aufbau und Funktionsweise der wichtigsten Module der 8-Bit-CPU. Zunächst aber schauen wir den einen, elementaren Baustein an, der in einer heutigen CPU milliardenfach verbaut ist und ohne den die heutige Digitaltechnik unmöglich wäre. Es handelt sich um den **Transistor**.

Teil 2 – Der Transistor

Ein Transistor ist ein Halbleiter-Bauteil mit drei Anschlüssen.Transistoren gibt es in vielen Grössen und Bauformen, hier eine kleine Auswahl:

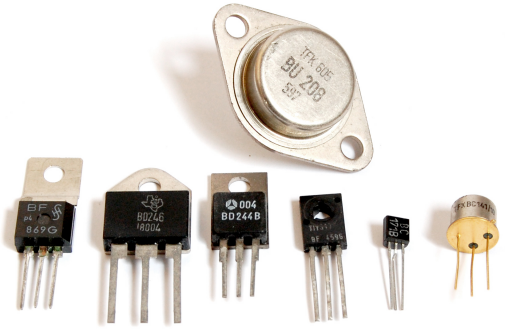


Bild 5 – Transistoren in unterschiedlichen Bauformen.

Dieses Video gibt dir eine Übersicht über den Transistor. Transistoren werden hauptsächlich als **Verstärker** und als **Schalter** verwendet. Sie haben drei Anschlüsse und erlauben es, mit einem kleinem Strom (oder mit einer kleinen Spannung) einen viel grösseren Strom (eine viel grössere Spannung) zu steuern. Das funktioniert wie folgt:

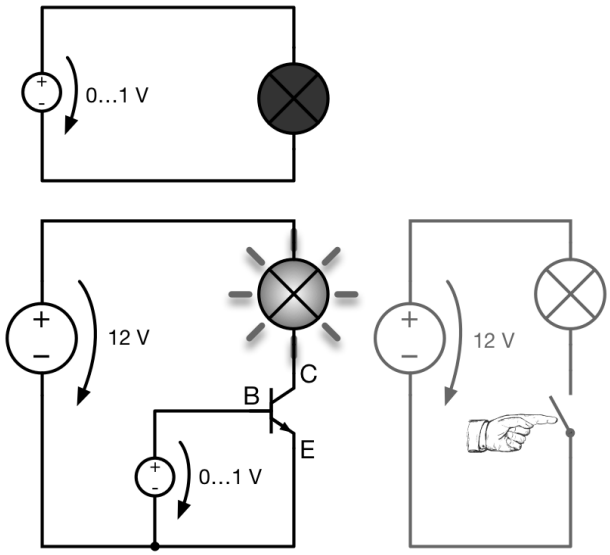


Bild 6 – Ein Transistor schaltet eine Glühlampe

In der Schaltung oben ist eine Glühlampe an eine sehr schwache Spannungsquelle angeschlossen: Die Spannung bzw. die Stromstärke reicht nicht aus, um die Glühlampe zum Leuchten zu bringen.

In der Schaltung unten links ist die Glühlampe an eine ausreichend starke 12V-Spannungsquelle angeschlossen – jedoch ist noch ein Transistor nachgeschaltet. Die schwache Spannungsquelle aus obiger Schaltung ist nun mit dem Transistor verbunden. Liegt zwischen dem mittleren (B) und dem unteren Anschluss (E) des Transistors eine Spannung an, so wird die Verbindung zwischen dem oberen (C) und dem unteren Anschluss (E) leitend.

Die Schaltung unten rechts zeigt dieselbe Schaltung, wenn wir uns den Transistor als normalen Schalter vorstellen.

Der Transistor ist also ein **elektronischer Schalter**, weil er mit einer elektrischen Spannung gesteuert werden kann. Er ist aber auch ein **dynamischer Schalter**: Ein normaler Schalter (z.B. ein Lichtschalter) kennt nur zwei Zustände: *offen* oder *geschlossen* – entweder fliesst Strom oder es fliesst kein Strom. Beim Transistor hingegen fliesst *mehr* oder *weniger* Strom: Je grösser die Spannung am Eingang, desto besser leitet er, desto mehr Strom fliesst durch. Im Vergleich mit einem Lichtschalter hiesse das: Je stärker du drückst, desto heller leuchtet das Licht.

Diese Eigenschaft des dynamischen Schaltens macht den Transistor als Verstärker verwendbar.

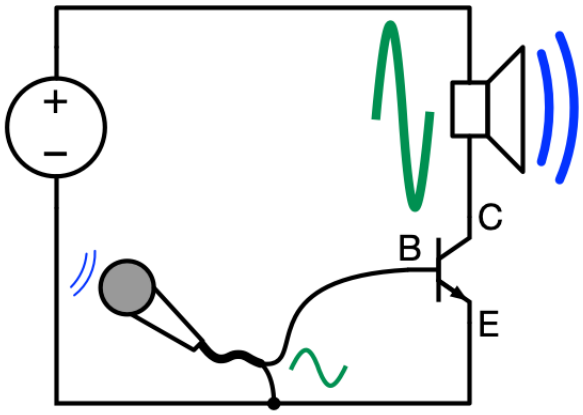


Bild 7 – Ein Transistor verstärkt ein Audiosignal.

Wenn du laute Musik hörst, sind heutzutage fast immer Transistoren im Einsatz. Bild 7 zeigt eine

vereinfachte Transistor-Verstärker-Schaltung: Das Mikrofon erzeugt eine kleine, wechselnde Spannung (das Audiosignal). Diese schaltet den Transistor dynamisch, so dass er analog zum Audiosignal mehr oder weniger leitet. Wenn der Transistor stark leitet, fliesst viel Strom durch den Lautsprecher, wenn er schwach leitet, fliesst wenig Strom. So erklingt dann der Lautsprecher, der an eine starke Spannungsquelle angeschlossen ist, analog zum Audiosignal am Mikrofon.

Im Folgenden interessiert uns diese dynamische Eigenschaft des Transistors jedoch nicht. Wie du wohl weisst, gibt es in der Digitaltechnik nur die Zustände 0 und 1. In digitalen Schaltungen sollen Transistoren also nicht mehr oder weniger schalten, wie sie das in Verstärkern tun, sondern sie sollen ganz oder gar nicht schalten.

Es gibt zahlreiche verschiedene Transistor-Arten mit unterschiedlichen Eigenschaften. Sie können in zwei Hauptgruppen unterteilt werden: In **bipolare** und **unipolare** Transistoren: In den Bildern 6 und 7 siehst du das Schemasymbol eines bipolaren Transistors. Seine Anschlüsse heissen *Base (B)*, *Emitter (E)* und *Collector (C)*. Die Schemasymbole von unipolaren Transistoren sehen etwas anders aus und die Anschlüsse heissen *Gate (G)*, *Source (S)* und *Drain (D)*. Hier eine kleine Übersicht:

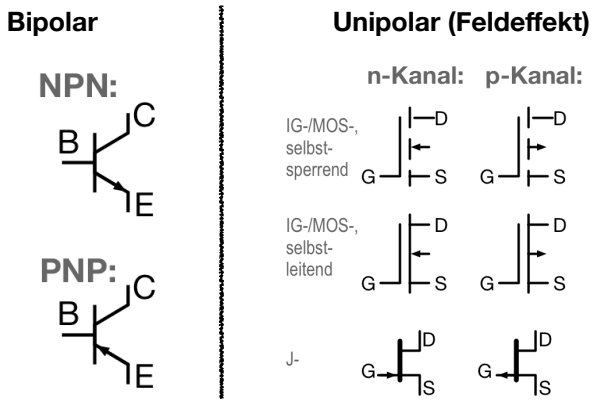


Bild 8 – Symbole der wichtigsten Transistor-Arten.

Unipolare Transistoren werden meist **Feldeffekt-Transistoren**, kurz **FET**, gennant. Warum sie so heissen, kannst du auf der nächsten Seite im Exkurs zur Funktionsweise von Transistoren nachlesen.

Was ist nun der grosse Unterschied zwischen bipolaren Transistoren und Feldeffekt-Transistoren? Oft wird gesagt: *„Bipolare Transistoren werden mit Strom gesteuert, Feldeffekt-Transistoren werden mit Spannung gesteuert.“* Weil Spannung und Strom zusammenhängen und sich gegenseitig beeinflussen, ist diese Aussage nicht ganz richtig. Bipolare Transistoren können mit einem kleinen Strom einen etwa hundertmal grösseren Strom schalten. Feldeffekt-Transistoren benötigen zum Schalten *nahezu keinen Strom*. Durch die geringe Verlustleistung entsteht auch weniger Wärme. In einer CPU, in der Milliarden von Transistoren auf einem Raum von der Grösse eines Daumennagels untergebracht sind, ist das wichtig. Wäre eine solche CPU mit bipolaren Transistoren aufgebaut, würde sie viel mehr Strom und Platz benötigen.

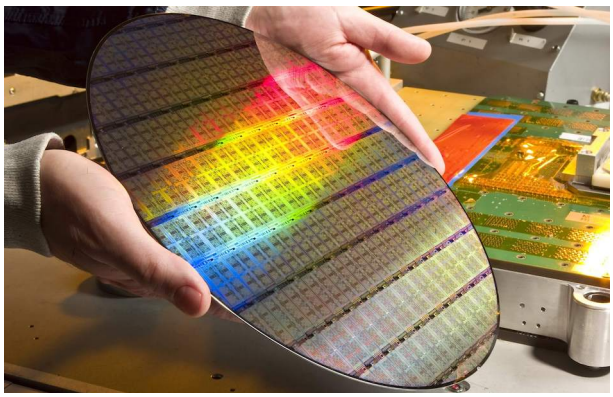


Bild 9 – Ein „Wafer“: Eine Silizium-Platte, die bereits mehrere hundert CPU-Chips enthält.

Fast alle Transistoren – und damit ein Grossteil aller Bausteine in digitalen Geräten – sind aus dem Halbleiter **Silizium** hergestellt. Die englische Bezeichnung **„Silicon“** gab dem Technologie-Tal in Kalifornien den Namen und wird heute auch synonym mit „Halbleiter“ und „Halbleiter-Technologie“ gebraucht. Hier ist knapp erklärt, wie eine CPU aus Silizium hergestellt wird (inkl. Video). Auf der folgenden Seite erfährst du, wie Transistoren aufgebaut sind und wie sie funktionieren.

Exkurs A – Die Vorgänge im Transistor

Folgende Erklärungen sind leichter zu verstehen, wenn du dich mit diesem Video über Halbleiter informierst.

Funktionsweise des bipolaren Transistors

Bild 10 zeigt den inneren Aufbau eines bipolaren Transistors in drei Zuständen. Als Ergänzung zu folgenden Erklärungen ist dieses Video hilfreich.

Löcher-Strom I_B. Der Widerstand zwischen C und E ist nun klein. Für ein Elektron, das zur Base geht, gehen etwa 99 zum Collector. Je grösser I_B, desto grösser I_C.

Funktionsweise des unipolaren Transistors:

Aufgrund der Wirkung durch ein elektrisches Feld wird der unipolare Transistor auch Feldeffekt-Transistor (FET) genannt.

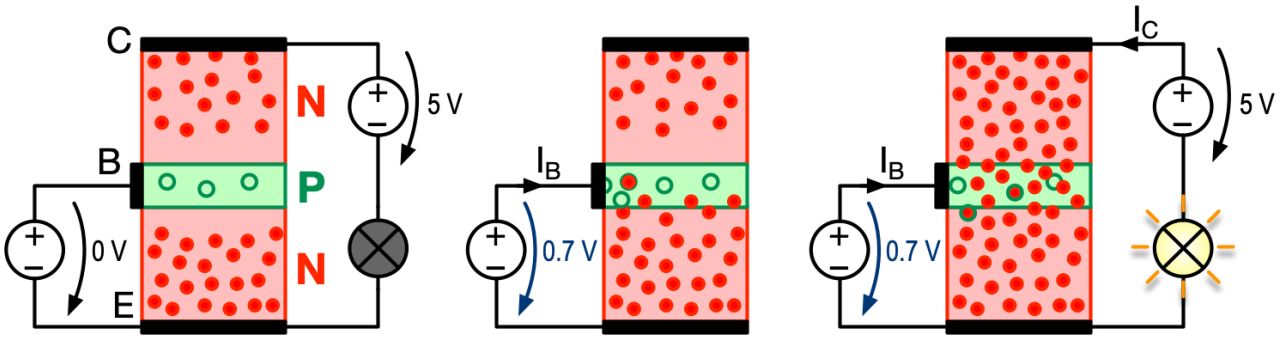


Bild 10 – Funktionsweise des bipolaren Transistors: Elektronen passieren zwei unterschiedlich gepolte Schichten.

Bild 10 links: Im nicht-beschalteten Zustand ist der Transistor nicht leitend. In den Grenzbereichen zwischen den N-Schichten und der P-Schicht bestehen sogenannte *Sperrschichten*, das sind Bereiche ohne freie Ladungsträger. Der Widerstand zwischen C und E ist hier sehr gross. Es fliesst kein Strom.

Bild 10 Mitte: Wird über B und E eine Spannung angelegt, so wird die Sperrschicht dieses PN-Übergangs abgebaut, Elektronen (rot) wandern vom *Emitter* zur *Base*, Löcher (grün) in die Gegenrichtung: Es fliesst ein Strom I_B.

Bild 10 rechts: Da die P-Schicht sehr dünn ist (viel dünner als hier dargestellt) und verhältnis-mässig wenige Löcher aufweist, springt ein Grossteil der Elektronen aus der unteren gleich in die obere N-Schicht. Nur ein geringer Teil rekombiniert mit den Löchern und führt so zum

Bild 11 links: Die beiden N-Schichten (rot) an den Anschlüssen D und S sind von einer P-Schicht (grün) und damit von Sperrschichten umgeben. Der FET leitet nicht. In der P-Schicht befinden sich freie Elektronen.

Bild 11 rechts: Wird eine positive Spannung zwischen G und S angelegt, so entsteht zwischen G und Bulk ein **elektrisches Feld** (grüne Pfeile). Dieses Feld lässt die Elektronen nach links wandern. So wird ein n-leitender Kanal erzeugt. Der FET leitet.

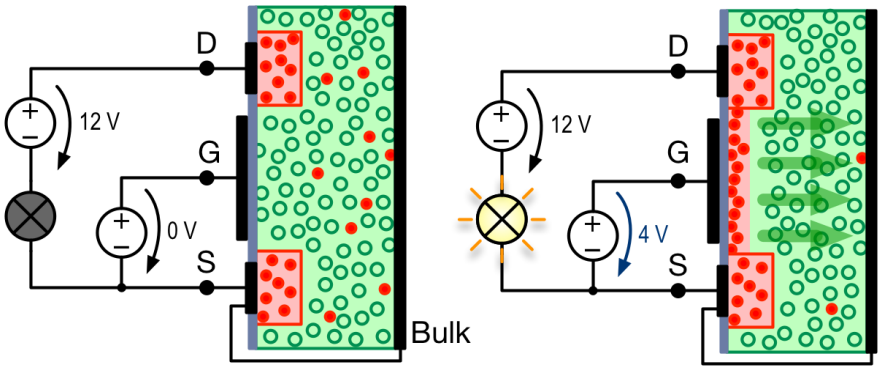


Bild 11 – Funktionsweise eine FET: Elektronen passieren einen unipolaren Kanal.

Teil 3 – Logische Verknüpfungen

Wie können aus Transistoren komplexe, programmierbare Strukturen entstehen? Der Prozessor in einem Smartphone schafft es, „gleichzeitig“ die Eingaben am Touchscreen auszuwerten, Bilder auf dem Display zu zeichnen, Musik abzuspielen und vieles mehr. Wie geht das mit einem Haufen elektronischer Schalter? Die Antwort erfolgt Schritt für Schritt und beginnt bei den logischen Verknüpfungen.

Die UND-Verknüpfung

Eine sehr einfache Aufgabe einer elektrischen Steuerung wäre die: Die Leuchtdiode (LED), die an einer Kaffeemaschine anzeigt, dass der Wassertank leer ist, soll dann leuchten, wenn A) der Wassertank eingesetzt ist ***und*** B) der Wasserstand gering ist:

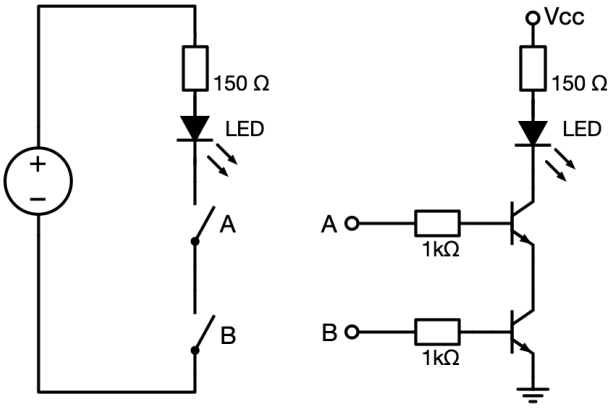


Bild 12 – Und-Verknüpfung mit Schaltern/Transistoren
Vcc-Symbol = Pluspol der Spannungsquelle. Symbol mit drei Strichen = Minuspol der Spannungsquelle.

Bild 12 zeigt links die beiden Sensoren A und B als Schalter. Die meisten Sensoren sind aber nicht so gebaut, dass sie wie Schalter einen Kontakt öffnen und schliessen. Viele Sensoren ändern passend zur Messgrösse ihren Widerstand oder geben eine bestimmte Spannung ab. Solche könnten in der Schaltung rechts verwendet werden: Wenn die Spannung vom Sensor A oder B gross genug ist, schaltet der entsprechende Transistor.

In beiden Schaltungen kann nur dann Strom durch die LED fliessen, wenn *beide* Schalter geschlossen bzw. *beide* Transistoren leitend sind.

Die ODER-Verknüpfung

Bei der ODER-Verknüpfung leuchtet die LED, wenn Sensor A *oder* B (oder beide) aktiv sind:

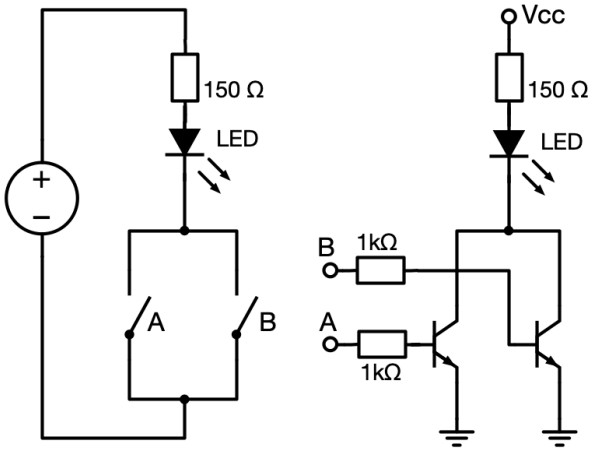


Bild 13 – Oder-Schaltung mit Schaltern/Transistoren

Damit Strom durch die LED fliessen kann, reicht es, wenn nur Schalter A oder nur Schalter B geschlossen ist. Für die Schaltung mit den Transistoren gilt das gleiche.

Die NICHT-Schaltung (Inverter-Schaltung)

Bei der NICHT-Schaltung leuchtet die LED, wenn der Sensor *nicht* aktiv ist:

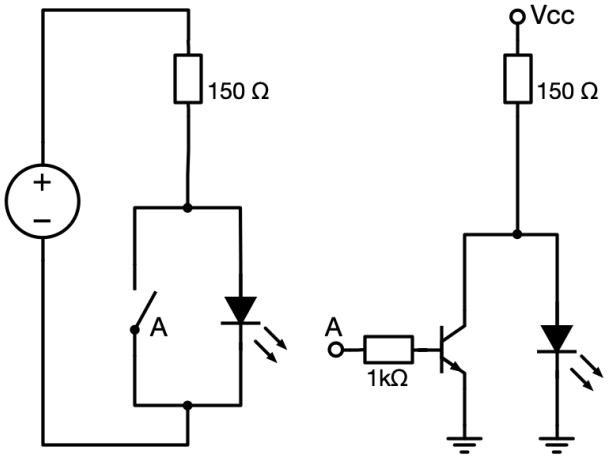


Bild 14 – NICHT-Schaltung mit Schaltern/Transistoren

Wenn Schalter A offen ist, fliesst Strom durch die LED. Wenn Schalter A geschlossen ist, fliest der Strom lieber durch Schalter A, da er auf diesem Weg viel weniger Widerstand hat.

HIGH oder LOW, 1 oder 0, true or false

In den Erklärungen oben hast du von Spannungen und Strömen gelesen. In der Digitaltechnik spricht man meistens bloss von 0 und 1 oder von LOW und HIGH:

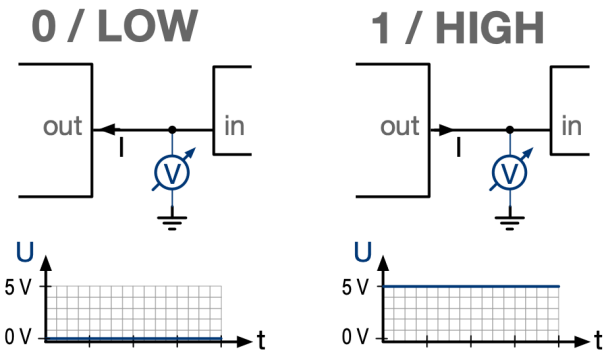


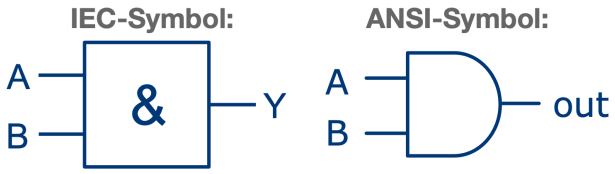
Bild 15 – Spannung und Strom am digitalen Ausgang.

Bild 15 zeigt zwei digitale Bausteine: der Ausgang des linken Bausteins ist mit dem Eingang des rechten Bausteins verbunden. Dazwischen ist ein Voltmeter angeschlossen. Im Diagramm darunter siehst du die Spannung, die das Voltmeter misst. Wenn ein Ausgang an einem digitalen Baustein 0 ist, dann beträgt die Spannung dort 0 Volt, die Spannung ist also tief, „LOW" und Strom fliesst in den Ausgang hinein. Wenn der Ausgang 1 ist, dann beträgt die Spannung je nach Logik-Standard 3.3 V oder 5 V (seltener auch andere Spannungen), die Spannung ist also hoch, „HIGH" und es fliesst Strom aus dem Ausgang heraus. In der Software-Entwicklung werden die beiden Werte oft *true* und *false* genannt.

Logik-Gatter / Logic gates

Mit Transistoren kannst du logische Verknüpfungen bauen. Diese gibt es auch als fertige Bausteine. Sie heissen Logik-Gatter (*logic gates*) und haben bestimmte Symbole und Funktionen, die du fortan kennen solltest. Folgend sind jeweils die Symbole der IEC (*International Electrotechnical Commission*) und des ANSI (*American National Standards Institute*) aufgeführt. Beide Standards sind verbreitet:

AND-Gate:



Wahrheitstabelle:

A	B	Y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Boolsche Algebra:

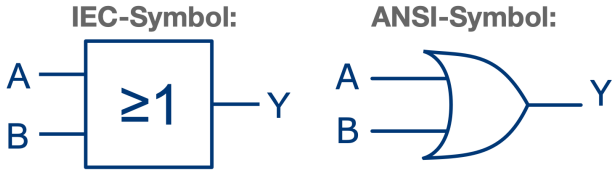
$Y = A \wedge B$

logisch UND

Bild 15 – UND-Gatter

Beim AND-Gate ist der Ausgang nur dann 1, wenn alle Eingänge 1 sind. Das Et-Zeichen '&' deutet die Funktion an. Das ANSI-Symbol kommt ohne Zeichen aus: Die Eingangs-Seite links ist eine gerade Linie, die Ausgangs-Seite ein runder Bogen.

OR-Gate:



Wahrheitstabelle:

A	B	Y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

Boolsche Algebra:

$Y = A \vee B$

logisch ODER

Bild 16 – ODER-Gatter

Das Grösser-Gleich-1-Zeichen '≥1' im IEC-Symbol bedeutet, dass einer oder mehrere Eingänge 1 sein können, damit der Ausgang 1 ist. Das ANSI-Symbol unterscheidet sich vom AND-Gate auf beiden Seiten: Die Eingangsseite ist hier ein Bogen, die Ausgangsseite ein Spitzbogen (wie in der gotischen Architektur –> Merkhilfe: G**o**tik–**O**der).

XOR-Gate:

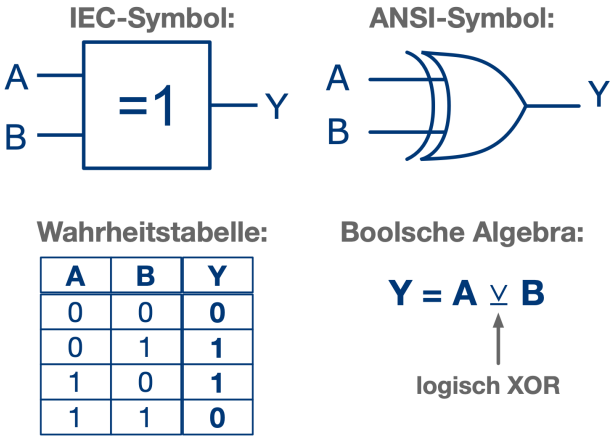


Bild 17 – Exklusiv-ODER-Gatter

Das Gleich-1-Zeichen im IEC-Symbol bedeutet, dass genau einer der Eingänge 1 sein muss, damit der Ausgang 1 ist. Das ANSI-Symbol entspricht dem OR-Gate – mit zusätzlichem Bogen an der Eingangsseite. XOR steht für „e**x**klusive-**OR**“, also „ausschliessendes ODER“. Wir können es auch „Entweder-ODER“ nennen: Der Ausgang ist 1, wenn *entweder* A *oder* B – aber nicht beide – 1 sind.

NOT-Gate/Inverter

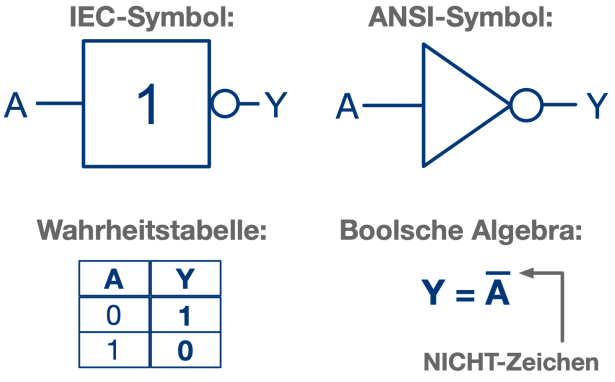


Bild 18 – NICHT-Gatter

Das NICHT-Gatter hat im Gegensatz zu allen anderen Gattern nur einen Eingang und eine sehr einfache Funktion: Der Ausgang ist 1, wenn der Eingang 0 ist – und umgekehrt.

Eine kleine Schaltung mit Gattern:

Die automatischen Sonnenstoren auf Karls Sitzplatz fahren aus (Motor M=1), wenn die Sonne scheint (Lichtsensor S=1) oder wenn es regnet (Regensensor R=1). Sie fahren auf jeden Fall ein (M=0), wenn es windet (Windsensor W=1). Diese Funktion wird durch folgende Schaltung erfüllt:

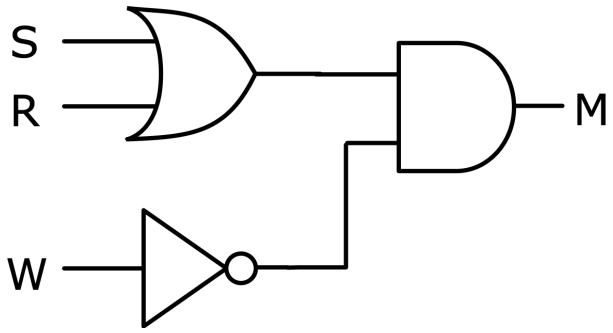


Bild 19 – Sonnenstorensteuerung

Der Motor M ist 1, wenn S oder R 1 ist (oder beide) *und* W 0 ist.

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**:

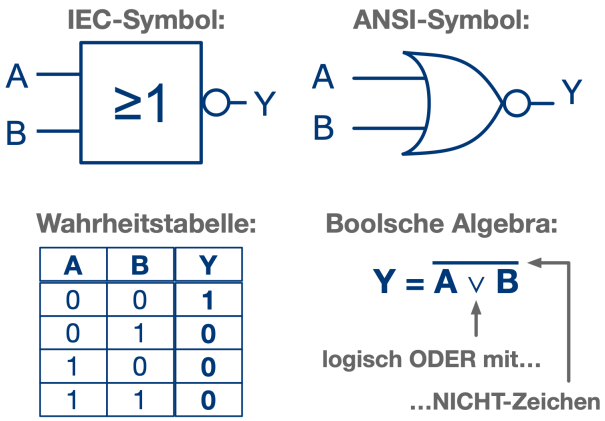


Bild 20 – Nicht-ODER-Gatter

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:

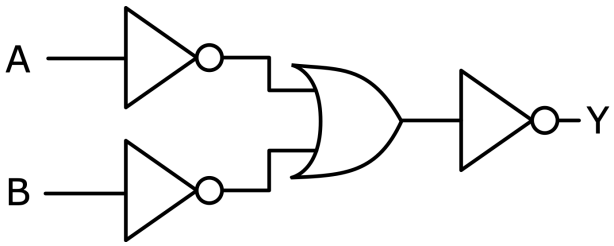


Bild 21 – UND-Verknüpfung aus NOT- und OR-Gates

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:

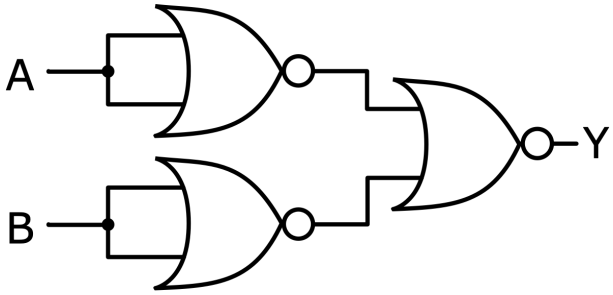


Bild 22 – UND-Verknüpfung aus NOR-Gates

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:

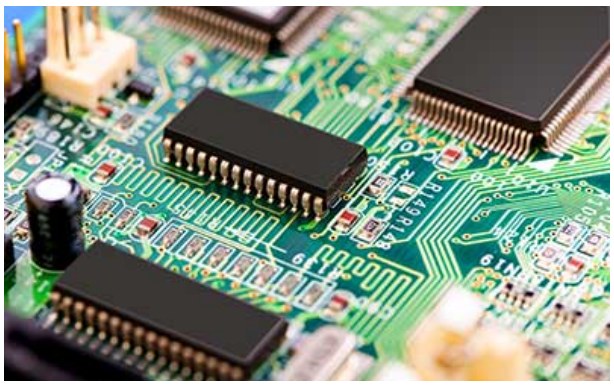


Bild 23 – ICs auf einer Leiterplatte

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

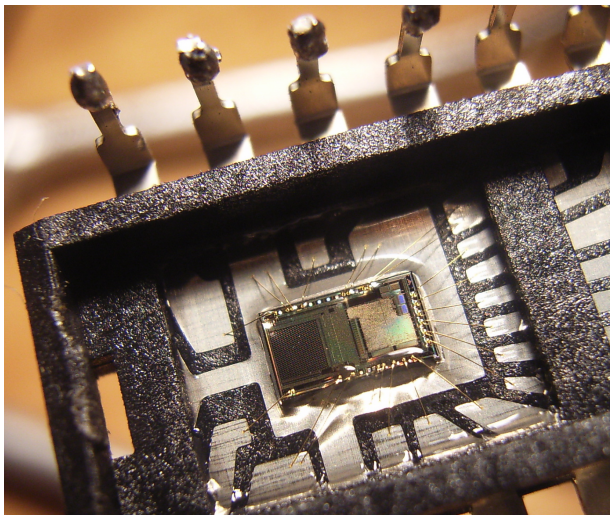


Bild 24 – Chip im Gehäuse eines IC

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:

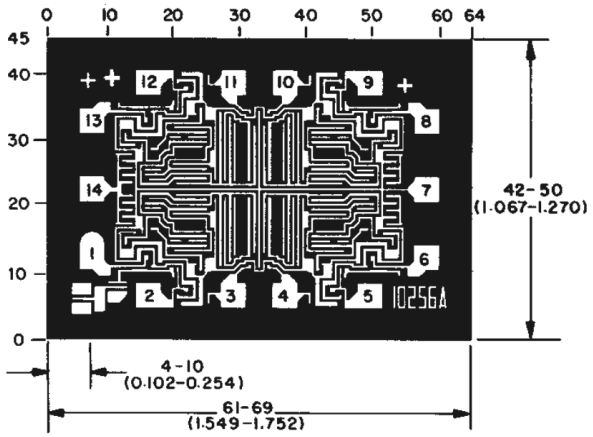


Bild 25 – Layout des IC *CD4011*, Maße in mil / (mm)

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:

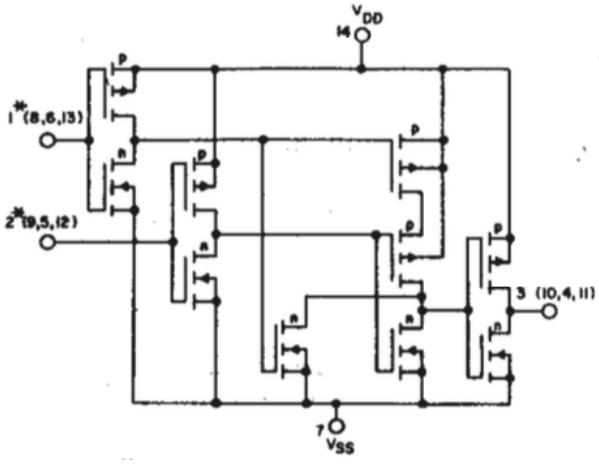


Bild 26 – Schaltplan eines NAND-Gate im IC *CD4011*

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:

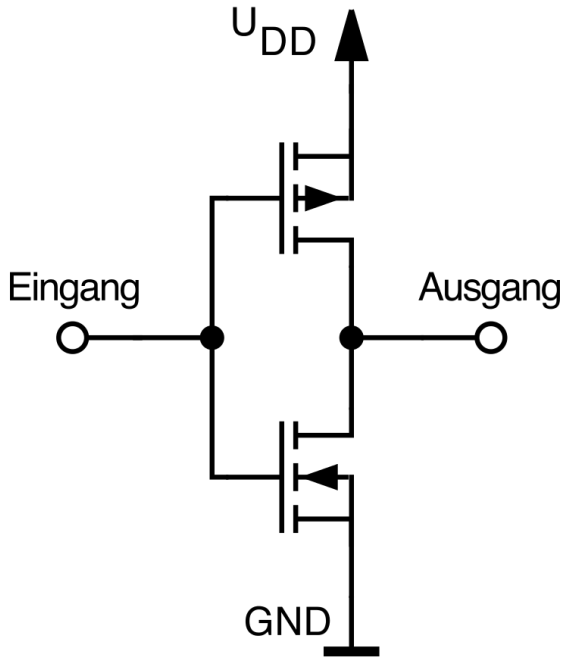


Bild 27 – CMOS-Inverter aus p- und n-Kanal-FET

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:

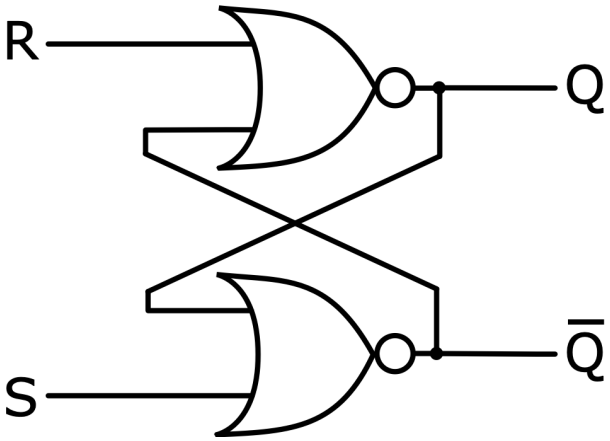


Bild 29 – ein RS-Flipflop aus zwei NOR-Gates

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:

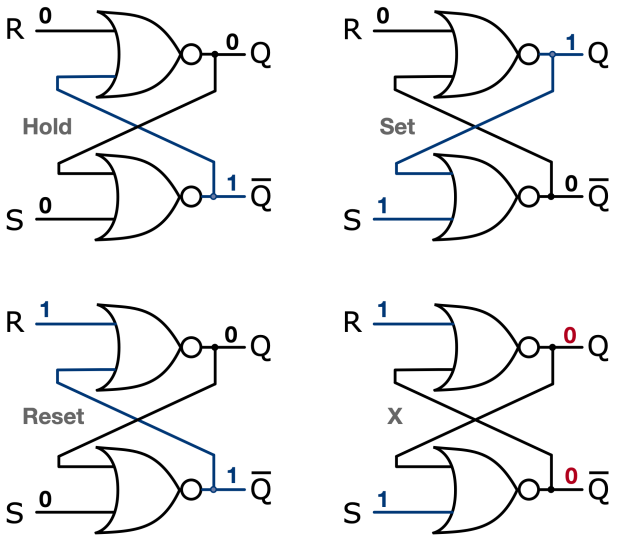


Bild 30 – vier Zustände des RS-Flipflop

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:

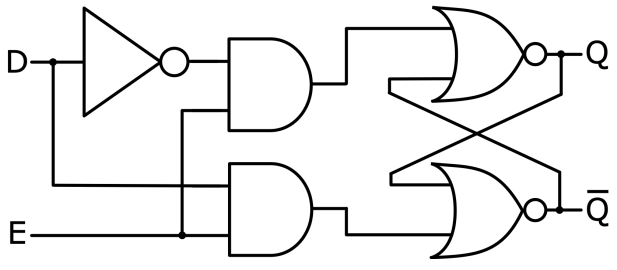


Bild 31 – D-Flipflop mit Enable-Eingang

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:

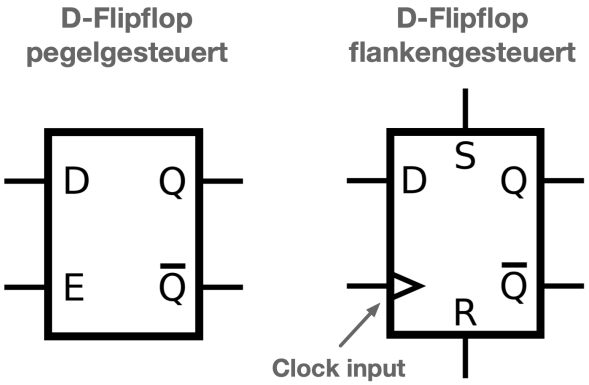


Bild 32 – zwei unterschiedliche D-Flipflops

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:

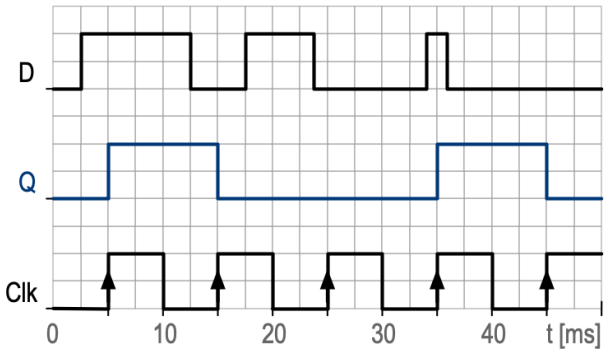


Bild 33 – Timing diagram eines D-Flipflops

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:

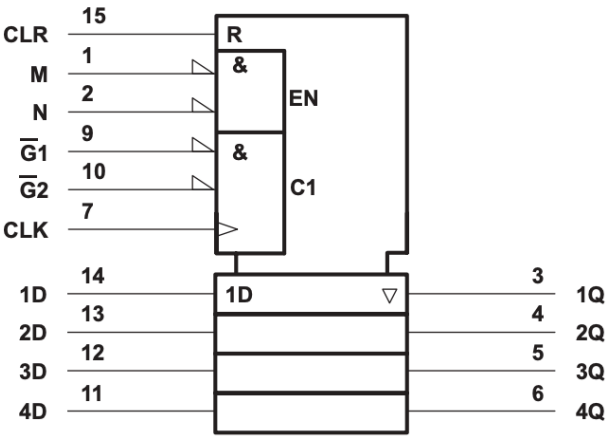


Bild 34 – Symbol des 4-Bit-Registers *74LS173*

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:

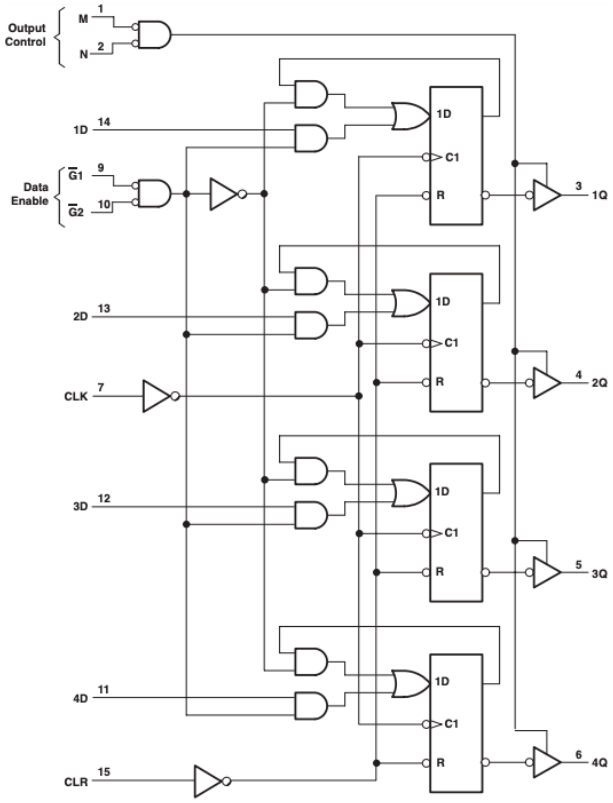


Bild 35 – logic diagram des 4-Bit-Registers *74LS173*

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:

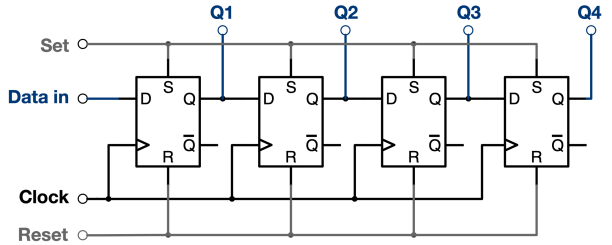


Bild 36 – Logikschaltung eines 4-Bit-Schieberegisters

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.

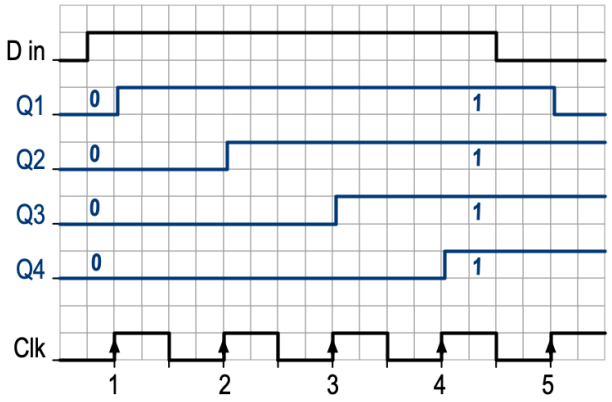


Bild 37 – Zeitdiagramm eines 4-Bit-Schieberegisters

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:

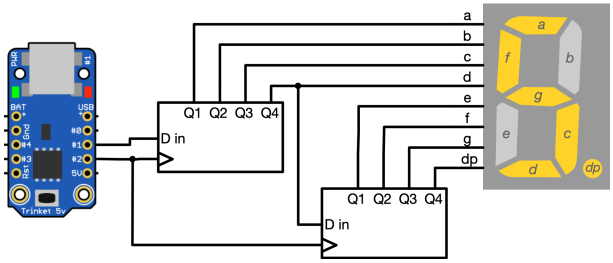


Bild 38 – Schieberegister vervielfachen Ausgänge

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:

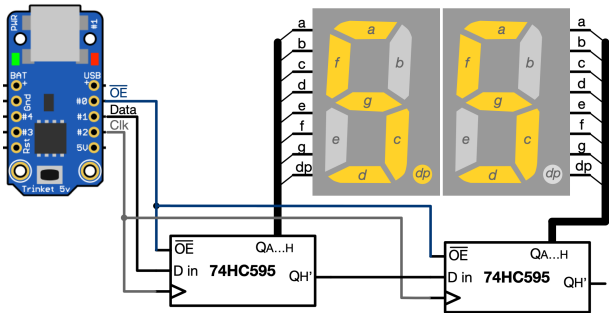


Bild 39 – Schieberegister mit *output-enable OE*

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)

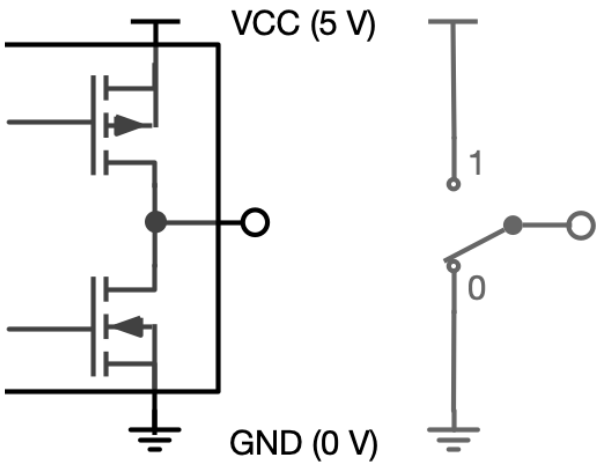


Bild 40 – Digitaler CMOS-Ausgang (vereinfacht)

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:

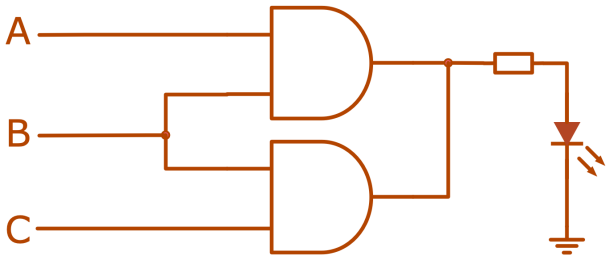


Bild 41 – Fehlerhafte Logikschaltung

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:

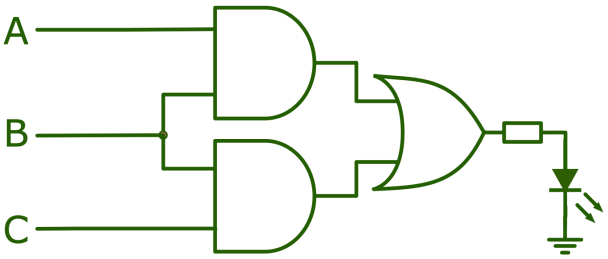


Bild 42 – Korrekte Logikschaltung

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*:

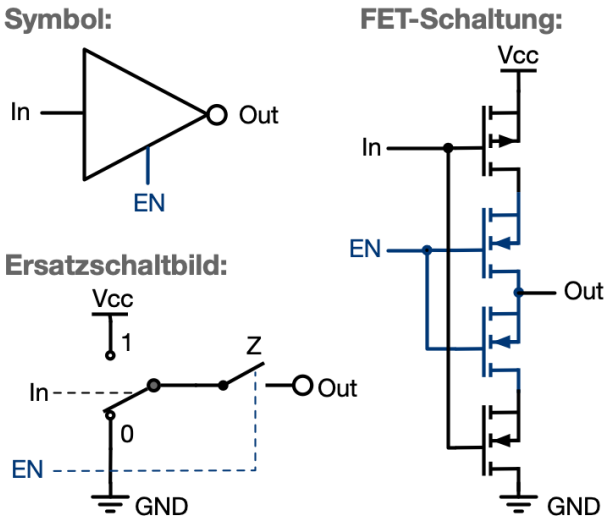


Bild 43 – Tri-State-Inverter

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:

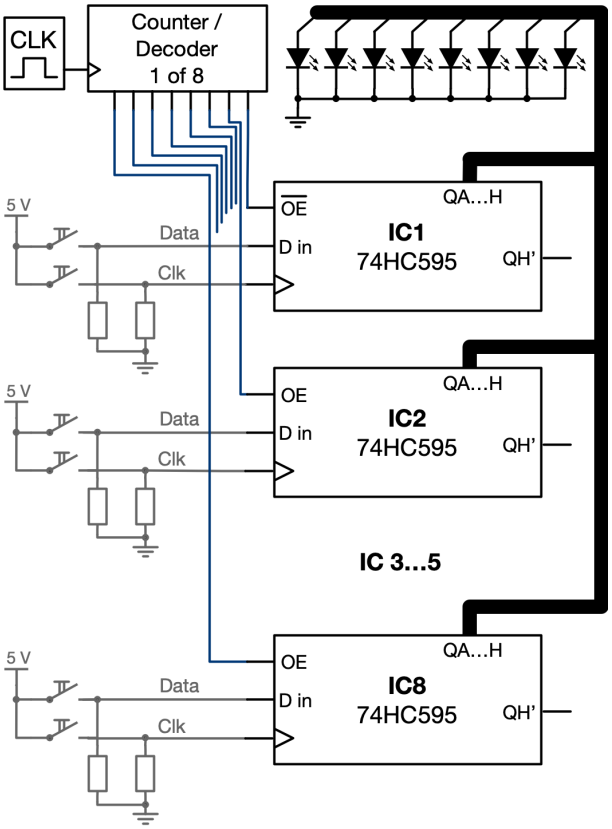


Bild 44 – Programmierbares Lichtmuster

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:

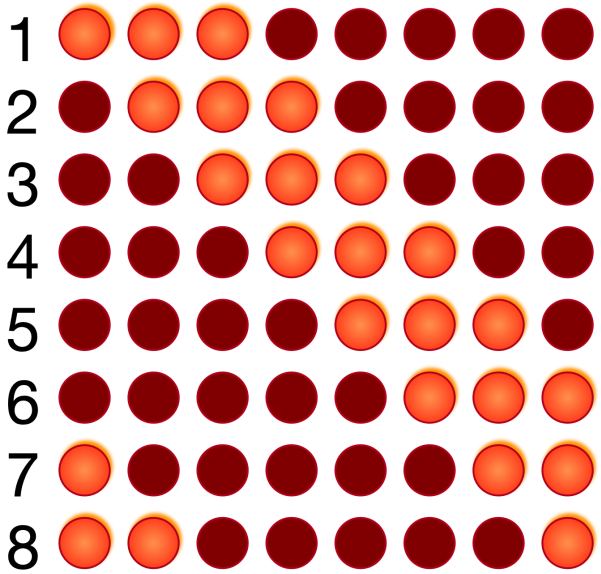


Bild 45 – Lauflicht-Muster

Drei LEDs würden nun ständig von links nach rechts laufen.

Teil 7 – Rechnen im Binärsystem

Jetzt weisst du schon, wie du mit digitalen Bausteinen Daten, also Nullen und Einsen, über einen Bus schicken und in Registern speichern kann. Das ist schon beinahe alles, was eine CPU kann. Doch es fehlen noch zwei wichtige Fähigkeiten: Zählen und Rechnen. Der Computer (lat. *computare* = berechnen) wird umgangssprachlich auch *Rechner* genannt und so könnte man sagen, dass eine CPU, die nicht rechnen kann, keine CPU ist. Wie rechnet also eine CPU?

Um das zu verstehen, musst du erst verstehen, wie man im *Binärsystem* – dieses System mit nur zwei Werten, 0 und 1 – Zahlen darstellt. Dazu hilft es, wenn du dir erst vor Augen führst, wie das dir vertraute Dezimalsystem funktioniert.

Die Zahl 234 würde ein Kind, das noch wenig von Zahlen weiss, vielleicht „Zwei-Drei-Vier“ nennen und vielleicht denken, dass sie kleiner ist als die Zahl „Sieben“. Du weisst aber: Wenn du 234 M&Ms abzählen musst, dann musst du *zweimal Hundert*, dann *dreimal zehn* und dann noch *vier* Einzelne abzählen. Denn dir ist klar: die Stelle ganz rechts ist 1-wertig, die links davon 10-wertig, dann 100-wertig etc. Das lässt sich auch folgendermassen sagen:

Die Stelle ganz rechts hat die Wertigkeit $\mathbf{10^0}$ (= 1), die links davon $\mathbf{10^1}$ (= 10), dann $\mathbf{10^2}$ (= 100) etc.

Das Binärsystem (lat. *bina* = doppelt) funktioniert gleich wie das Dezimalsystem (lat. *decem* = zehn). Nur ändert sich die Basis von 10 in 2: Die Stelle ganz rechts hat die Wertigkeit $\mathbf{2^0}$ (= 1), die links davon $\mathbf{2^1}$ (= 2), dann $\mathbf{2^2}$ (= 4) etc. So zählen wir im Binärsystem wie folgt:

Dezimalzahl	Binärzahl	Erklärung
0	0	…
1	1	$1 \times 2^0 = 1$
2	10	$1 \times 2^1 + 0 \times 2^0 = 2$
3	11	$1 \times 2^1 + 1 \times 2^0 = 3$
4	100	$1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4$
5	101	$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$
6	110	$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$
7	111	$1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$
8	1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8$
9	1001	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9$

Wenn du nun zum Beispiel die Binärzahl `10101` siehst, kannst du sie in eine Dezimalzahl umrechnen: Überall, wo eine 1 ist, zählst du die entsprechende Wertigkeit hinzu. Also, von links: $2^4 + 2^2 + 2^0 = 16 + 4 + 1 = 21$. Alles klar? Am besten schaust du noch dieses Video, das gleich auch noch das Hexadezimalsystem erklärt.

Binäre Addition

Um zwei binäre Zahlen zu addieren, gehts du vor wie bei der schriftlichen Addition. Hier ein Beispiel:

```
 18   10010  Operand A
 27   11011  Operand B
  1   10010  Übertrag (carry)
 45  101101  Summe (sum)
```

Bild 46 – Schriftliche Addition

Von rechts nach links wird jeweils nur eine Ziffer der beiden Operanden zusammengezählt. Übersteigt die Summe die Zahl 9 (im Dezimalsystem) bzw. 1 (im Binärsystem), so ergibt sich ein Übertrag.

Der Halbaddierer

Wie werden zwei Binärzahlen in der CPU addiert? Wir suchen erstmal eine Logikschaltung, die bloss zwei *einstellige* Binärzahlen zusammenrechnen kann. 2 Bits also. Da gibt es, wie du weisst, nur vier mögliche Kombinationen. Hier die Wahrheitstabelle:

A	B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Die Logikschaltung hat also zwei Eingänge für die beiden Operanden A und B und zwei Ausgänge: Summe und Übertrag. Die Summe ist in der ersten Zeile 0, weil beide Operanden 0 sind. In der vierten Zeile ist sie 0, weil beide Operanden 1 sind, denn das ergäbe die Zahl 2, also binär `10` – es entsteht ein Übertrag (carry = 1).

Die beiden Muster für *sum* und *carry* sollten dir bekannt sein: *sum* ist genau dann 1, wenn entweder A oder B, aber nicht beide 1 sind. Das ist die XOR-Funktion. Und *carry* ist genau dann 1 wenn, sowohl A als auch B 1 sind. Das ist die AND-Funktion. Die gesamte Schaltung nennt man *Halbaddierer* und sie sieht also so aus:

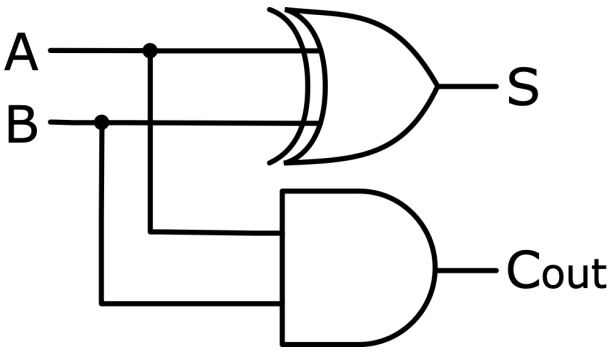


Bild 47 – Halbaddierer aus XOR und AND

Wieso heisst die Schaltung *Halbaddierer* – kann sie etwa nur halb addieren? Irgendwie schon: Sie kann zwar *zwei Bits* tadellos miteinander addieren, aber sie eignet sich nicht dazu, mehrfach hintereinandergeschaltet zu werden, damit mehrstellige Zahlen addiert werden können: Sie hat zwar einen Ausgang *Cout* für den Übertrag, der dann bei der nächsten Schaltung hinzu addiert würde. Aber die nächste Schaltung bräuchte dann eben einen dritten Eingang *Cin*, den sie zu A und B hinzu addieren könnte. Ein *Volladdierer*, der voll geeignet ist, Binärzahlen zu addieren, kann also *drei Bits* miteinander addieren.

Der Volladdierer

Sobald du bei der schriftlichen Addition einen Übertrag aufschreibst, rechnest du bei der nächsten Stelle diesen Übertrag mit; du zählst dann *drei* Ziffern zusammen. Der Volladdierer muss also zu *A* und *B* auch den Übertrag von der letzten Stelle hinzurechnen. Dazu verbinden wir zwei Halbaddierer miteinander:

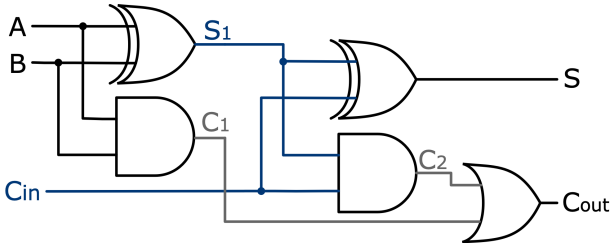


Bild 48 – Volladdierer aus zwei Halbaddierern und OR

In Bild 48 siehst du die Halbaddierer-Schaltung zweimal hintereinander. Die Eingänge *A* und *B* werden erst zu S1 addiert. Diese Zwischensumme wird dann noch mit *Cin* addiert. *Cout* ist 1, wenn mindestens eine der beiden Additionen einen Übertrag ergab, also wenn *C1* oder *C2* 1 sind. Die Schaltung erfüllt folgende Wahrheitstabelle:

Cin	A	B	S1	C1	C2	S	Cout
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	1	0	0	1	0
0	1	1	0	1	1	0	1
1	0	0	0	0	0	1	0
1	0	1	1	0	1	0	1
1	1	0	1	0	1	0	1
1	1	1	0	1	0	1	1

Du kannst nun kontrollieren, ob die Schaltung stimmt: Zum Beispiel die zweitletzte Zeile: Wenn *Cin* = *A* = 1 und *B* = 0 ist, dann muss die Summe 0 und der Übertrag 1 sein – richtig?

Mehrstellige Binärzahlen addieren

Du kannst mehrere Volladdierer zusammenschalten und so Addierer-Schaltungen für beliebig grosse Binärzahlen bauen. Zum Beispiel für 4-Bit-Zahlen:

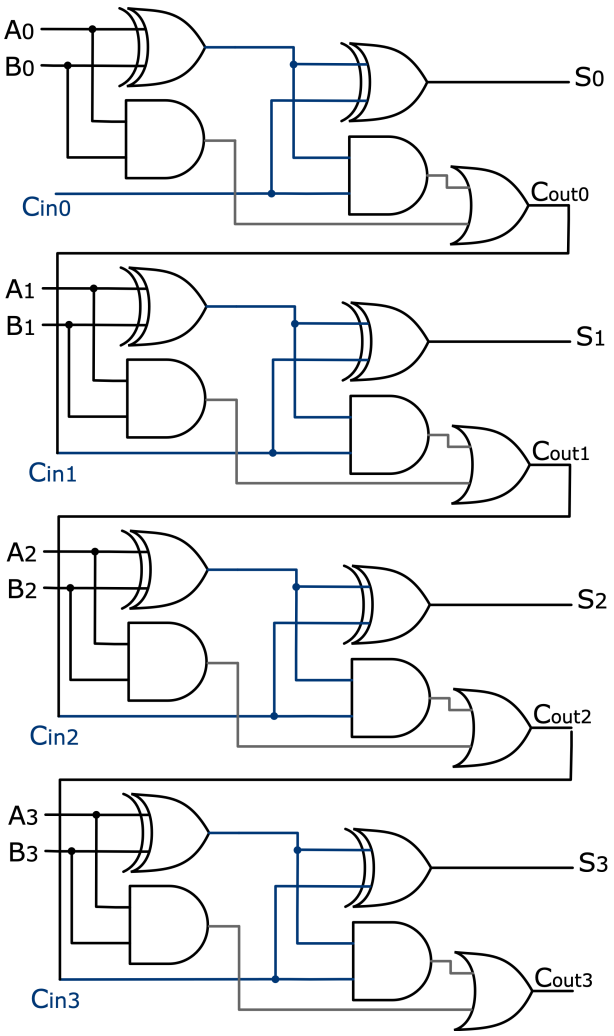


Bild 49 – 4-Bit-Addierschaltung aus 4 Volladdierern

Diese Schaltung gibt es auch schon fix fertig als Integrierte Schaltung, zum Beispiel im IC 74LS283, dessen Symbol so aussieht:

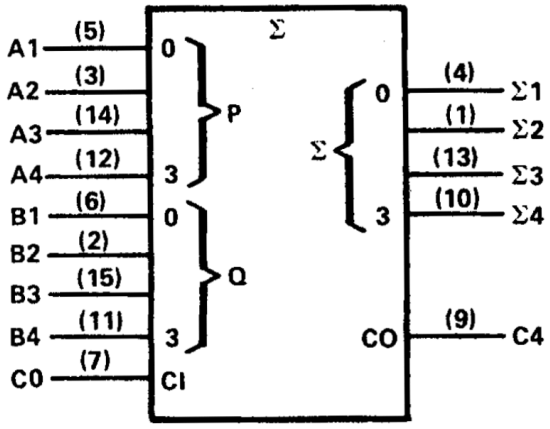


Bild 50 – 4-Bit-Volladdierer-IC 74LS283

Der IC 74LS283 in Bild 50 hat – wie auch die Schaltung in Bild 49 – zwei mal vier Eingänge für die beiden 4-Bit-Operanden und vier Ausgänge für den 4-Bit-Summanden. Ausserdem hat er noch den Eingang CI und den Ausgang CO – damit können mehrere dieser 4-Bit-Addierer zu 8-, 12-, 16- und mehr-Bit-Addierern zusammengeschaltet werden.

Liegt an den Eingängen A0…A3 der Wert `0110` (6) und an den Eingängen B0…B3 der Wert `0111` (7) an, so liegt an den Ausgängen S0…S1 der Wert `1101` (13) an. Die Addition erfolgt sozusagen in Echtzeit: Es dauert nur wenige Nanosekunden, bis ein Wechsel auf 1 oder auf 0 an einem Eingang einen Wechsel am Ausgang bewirkt. Dieses Video erklärt die 4-Bit-Addierer-schaltung von A bis Z.

Binäre Subtraktion

Um zwei Binärzahlen voneinander abzuzählen, gehst du vor wie bei der Subtraktion:

```
25   11001  Operand A
19   10011  Operand B
 1    0110  Übertrag (carry)
06   00110  Differenz
```

Bild 51 – Schriftliche Subtraktion

Ziffer um Ziffer wird von rechts nach links Operand B von Operand A abgezählt: Ist das Ergebnis kleiner als 0, so ergibt sich ein Übertrag. Im Beispiel in Bild 51 klappt das ohne Problem. Aber was passiert, wenn sich ein negatives Resultat ergeben soll?

```
 19   10011  Operand A
 25   11001  Operand B
 10   11000  Übertrag (carry)
994  111010  Differenz
```

Bild 52 – Subtraktion mit negativem (?) Resultat

Du weisst ja: 19 minus 25 ergibt -6. Das übliche Vorgehen der schriftlichen Subtraktion scheitert hier offenbar. Die Resultate in Bild 52 sehen ziemlich falsch aus – sowohl im Dezimal- als auch im Binärsystem. Die Resultate sind aber nicht

„kreuzfalsch“, es sind bloss die *Komplemente* des richtigen Resultats.

Das Komplement: Ergänzung aufs Ganze

„Komplement“ bedeutet hier Ergänzung: Wenn *das Ganze* die Zahl 10^3 (1000) ist, dann ist 994 *die Ergänzung* zur Zahl 6. Und im Binärsystem: Wenn *das Ganze* die Zahl 2^6 (64) ist, dann ist `111010` (58) *die Ergänzung* zur Zahl 6:

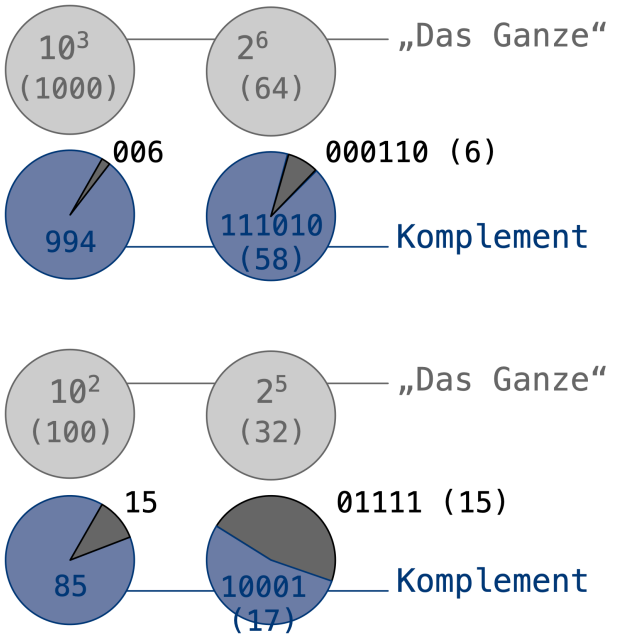


Bild 53 – Komplement ergänzt Zahl auf das Ganze

In Bild 53 siehst du zwei Beispiele, in denen eine Zahl (schwarz) um ihr Komplement (blau) auf das Ganze (grau) ergänzt wird. Wie gross dieses Ganze ist, ist abhängig von der Anzahl Stellen. Im oberen Beispiel sind es drei bzw. sechs Stellen, im unteren Beispiel sind es zwei bzw. fünf Stellen.

Ab jetzt lassen wir das Dezimalsystem beiseite und kümmern uns nur noch um das Binärsystem. Es gilt:

Das Komplement einer N-stelligen Zahl ist ihre Differenz zur Zahl 2^N.

Also: Das Komplement von `110` (6) ist `010` (2), weil es *dreistellige* Zahlen sind: 2^3 gibt 8, die Differenz von 6 zu 8 ist 2.

Aber: Das Komplement von `0110` (6) ist `1010` (10), weil es *vierstellige* Zahlen sind: 2^4 gibt 16, die Differenz von 6 zu 16 ist 10.

Das 2er-Komplement (two's complement)

Damit Computer mit negativen Zahlen rechnen können, müssen sie negative von positiven Zahlen unterscheiden. Wir Menschen machen das mit einem Minus-Zeichen. Der Computer hat aber nur Einsen und Nullen und sonst nichts. Die Lösung für dieses Problem könnte so aussehen: Wenn die Binärzahl mit einer 1 beginnt, dann ist sie negativ, wenn sie mit einer 0 beginnt, dann ist sie positiv. Diese Idee ist in der *2er-Komplement-Darstellung* umgesetzt. In dieser Darstellung gilt:

1. Steht links eine 1, so ist die Zahl negativ.

2. Das 2er-Komplement von X ist -X.

Nehmen wir die Binärzahl `101`. In der normalen Darstellung entspricht das der Dezimalzahl **5**. Aber in der 2er-Komplement-Darstellung ist das eine negative Zahl, da ganz links eine 1 steht. Doch welche negative Zahl? Wenn wir das Komplement von `101` bilden, erhalten wir die entsprechende positive Zahl, denn das 2er-Komplement von X ist -X. Es gibt zwei Varianten, das 2er-Komplement zu bilden:

Variante A: Bilde die Differenz zum Ganzen.

Das Ganze: $2^N = 2^3 = 8$.

Die Differenz zum Ganzen: 8 - 5 = 3: `011`.

Variante B: Invertiere alles und addiere um 1:

Invertiere alles: `101 -> 010`

Addiere um 1: `010 + 001 = 011`.

Das 2er-Komplement von `101` ist `011` (3). Also entspricht `101` in der 2er-Komplement-Darstellung der Zahl **-3**.

Betrachte nun erneut das binäre Resultat in Bild 52 in der 2er-Komplement-Darstellung: Das Resultat beginnt mit einer 1, ist also negativ. Um zu wissen, wie gross die Zahl ist, bilden wir das Komplement, zum Beispiel nach Variante B:

Invertieren: `111010 -> 000101` und addieren um 1: `000101 + 1 = 000110` (6). `111010` entspricht der Zahl -6. In der 2er-Komplement-Darstellung ist die Rechnung in Bild 52 also richtig!

Ein Nachteil der 2er-Komplement-Darstellung ist natürlich, dass sie 1 Bit für das Vorzeichen braucht: In der normalen Binärdarstellung kannst du mit vier Bits bis 15 zählen. In der 2-er-Komplement-Darstellung kannst du mit vier Bits von -8 bis 7 zählen:

Binärzahl	Dezimalwert normal	Dezimalwert bei 2er-Komplement	2er-K. der Binärzahl
0000	**0**	**0**	**0000**
0001	1	1	1111
0010	2	2	1110
0011	3	3	1101
0100	4	4	1100
0101	5	5	1011
0110	6	6	1010
0111	7	7	1001
1000	8	-8	**1000**
1001	9	-7	0111
1010	10	-6	0110
1011	11	-5	0101
1100	12	-4	0100
1101	13	-3	0011
1110	14	-2	0010
1111	15	-1	0001

In der Tabelle kannst du einige Besonderheiten und Muster erkennen: Das 2er-Komplement der Zahl 0 ist immer 0. Das macht Sinn, weil 0 ja weder positiv noch negativ ist. Das 2er-Komplement der Zahl `1000` ist ebenfalls `1000`, denn: Das Ganze ist hier (bei vier Bits) 16. `1000` (8) ist genau die Hälfte des Ganzen. Also ist das Komplement von 8 ebenfalls 8. Du siehst: Wenn du die Werte in der Spalte ganz links mit ihren Komplementen in der Spalte ganz rechts addierst, erhältst du immer das Ganze – ausser eben bei der Zahl 0.

Am besten schaust du dir dieses Video an, das das 2er-Komplement nochmal etwas einfacher erklärt.

Eine Subtrahier-Schaltung

Zurück zur Hardware: Wie werden zwei Binärzahlen in der CPU subtrahiert? Ganz einfach: von der zweiten Zahl wird das 2er-Komplement gebildet, womit sie negativ wird – und dann wird sie zur ersten addiert. Statt 7 - 5 rechnet die CPU einfach 7 + (-5).

Du könntest also aus einem 4-Bit-Volladdierer (wie in Bild 52) einen 4-Bit-Subtrahierer machen:

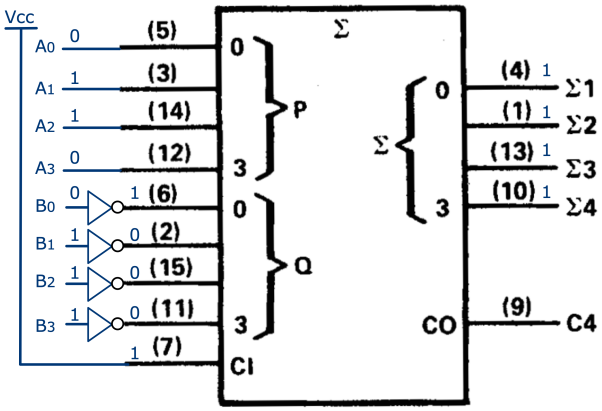


Bild 54 – Subtrahierer mit Volladaddier-IC 74LS283

Angenommen, bei A0…A3 liegt der Wert `0110` (6) und bei B0…B3 liegt der Wert `0111` (7) an: B0…B3 werden mit den vier NOT-Gates invertiert. Also liegt an Q0…Q3 der Wert `1000` an. Weil *CI* auf Vcc, also auf HIGH liegt, wird zur Zahl `1000` an Q0…Q3 noch `1` addiert. Das ergibt `1001` – und das ist die Zahl -7; das 2er-Komplement von `0111`. Die „Summe“ am Ausgang ergibt also:

```
A0…A3:   0110    6
Q0…Q3:   1001   +(-7)
          000
S0…S3:   1111   -1
```

Bild 55 – Subtraktion durch Addition der Negation

Du siehst: Die Schaltung in Bild 54 ist ein 4-Bit-Subtrahierer. Damit die Rechnung aufgeht, ist die 2er-Komplement-Darstellung nötig.

Eine kleine ALU (Arithmetic logic unit)

Es wäre praktisch, wenn sowohl Addieren als auch Subtrahieren in *einer* Schaltung geschehen könnten. Eine ALU, das „Rechenzentrum“ jeder CPU, ist so aufgebaut: Sie erhält zwei Werte, A und B, die je nach Datenbus 8 bis 64 Bit breit sind. Ausserdem erhält sie Steuersignale, die ihr sagen, was sie mit diesen Werten tun soll – ob sie sie addieren, subtrahieren, multiplizieren, dividieren, logisch „verunden“, „verodern“ oder sonst was tun soll.

Wir schauen folgend eine Schaltung an, die bloss addieren und subtrahieren kann – es ist die ALU des 8-Bit-Computers aus Bild 1 dieses Skripts:

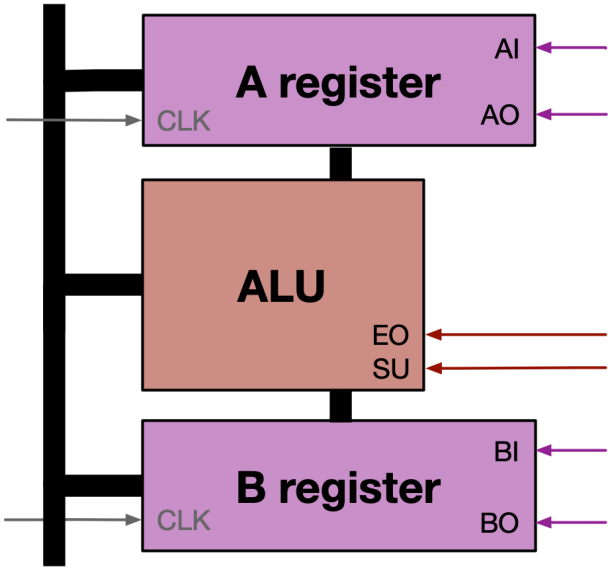


Bild 56 – 8-Bit-ALU, A-Register und B-Register

Die ALU in Bild 56 erhält vom A-Register und vom B-Register je einen 8-Bit-Wert. Wenn das Signal *EO* = 1 ist, dann schreibt die ALU das Resultat ihrer Rechnung bei der nächsten Clock-Flanke auf den Bus (links). Ob diese Rechnung eine Addition oder eine Subtraktion ist, wird über das Signal *SU* bestimmt: Wenn *SU* = 1 ist, dann soll die ALU subtrahieren, wenn *SU* = 0 ist, soll sie addieren.

Du hast gesehen, dass eine Addierschaltung zu einer Subtrahierschaltung wird, wenn von einem Operanden das 2er-Komplement gebildet wird. Die ALU muss also folgendes tun: Wenn *SU* = 1, muss sie vom Wert aus Register B das 2er-Komplement bilden. Ansonsten soll sie den Wert aus Register B unverändert lassen.

Der erste Schritt zur 2er-Komplement-Bildung ist das Invertieren. Wie kann ein Signal abhängig von einem anderen Signal invertiert werden? Die Lösung: mit einem XOR-Gatter. Folgend die Wahrheitstabelle des XORs:

A	B	Y (A ⊻ B)
0	0	0
0	1	1
1	0	1
1	1	0

Wenn A = 0 ist, dann entspricht der Ausgang des XOR-Gatters dem Wert B. Wenn aber A = 1 ist, dann entspricht der Ausgang des XOR-Gatters der Inversion von B. B wird also dann invertiert, wenn A = 1 ist. Die ALU lässt sich nun so aufbauen:

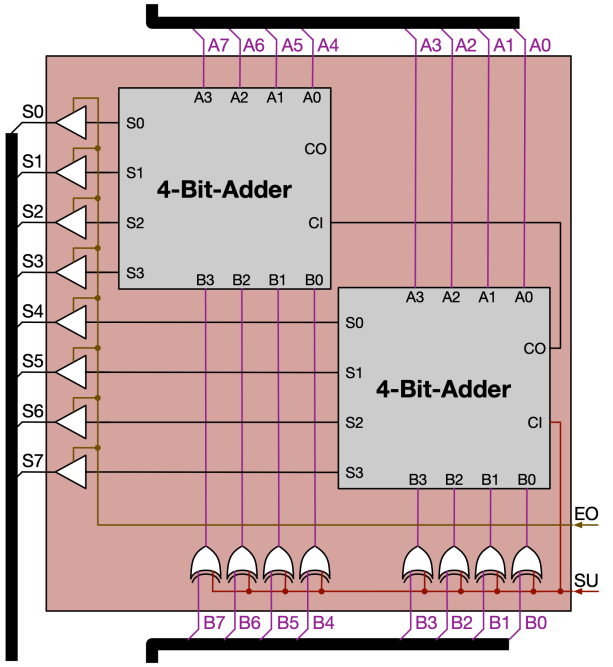


Bild 57 – Die Schaltung in der der 8-Bit-ALU

In Bild 57 siehst du, dass das Signal *SU* auf den Carry-In-Eingang *CI* des ersten 4-Bit-Addierer und auf acht XOR-Gatter geführt ist. Das heisst: Wenn *SU* = 0 ist, bleiben *B0…B7* unverändert und auch an CI liegt eine 0 an. Wenn aber *SU* = 1 ist, dann werden *B0…B7* invertiert und es wird 1 hinzu addiert. Somit wird das 2-er-Kompliment genau dann gebildet, wenn SU = 1 ist. Das Signal SU steuert damit, ob die Werte aus A- und B-Register addiert oder subtrahiert werden.

Das Signal *EO* schaltet die Tri-State-Buffer, die du im letzten Kapitel kennengelernt hast. Wenn *EO* = 0 ist, dann sind die Ausgänge der Tri-State-Buffer alle *Z*, also „nicht verbunden“. Wenn *EO* = 1, dann liegen an den Ausgänge der Tri-State-Buffer die gleichen Werte wie an den Eingängen. Dieses Video erklärt den Aufbau dieser ALU nochmals Schritt für Schritt.

Teil 8 – Zähler und Decoder

Addieren und subtrahieren ist schon mal nicht schlecht. Ein Computer sollte aber auch weitere Rechenoperationen wie die Multiplikation beherrschen. Die ALUs der meisten CPUs schaffen das hardware-technisch und damit in kürzester Zeit – ähnlich schnell, wie unsere einfache ALU in der 8-Bit-CPU zwei Zahlen addiert oder subtrahiert.

Wenn du mit der einfachen ALU zwei Zahlen multiplizieren willst, geht das auch *software-technisch* (sprich durch Herumschieben und Zwischenspeichern von Daten), jedoch etwas langsamer: 4 mal 7 ist ja nichts anderes als 7 + 7 + 7 + 7. Das Multiplikations-Programm auf der 8-Bit-CPU müsste einfach die erste Zahl speichern und sie so oft hinzu addieren, wie es die zweite Zahl vorgibt. Damit die CPU weiss, wie oft sie eine Addition durchgeführt hat, muss sie zählen können – eins, zwei, drei, vier etc. Zählen ist ja bloss eine fortlaufende Addition plus eins. Unsere 8-Bit-CPU mit ALU und Registern könnte also auch software-technisch zählen.

Zähler (counter)

Damit eine CPU aber überhaupt irgendein Programm ausführen kann, braucht sie bereits einen internen, *hardware-technischen* Zähler. Jedes Programm – egal ob du es in *python*, *C#* oder sonst einer Sprache codierest – besteht, wenn es bei der CPU ankommt, aus einer Reihe von Befehlen. Die Befehle werden ins RAM geladen und müssen in der richtigen Reihenfolge abgearbeitet werden: Erst Befehl 1, dann Befehl 2 und so weiter. Es braucht also ein Modul, das zählt, damit stets der richtige Befehl aus dem RAM geholt wird. In unserer 8-Bit-CPU (siehe auch Bild 1 in diesem Skript) ist das der *Program Counter*.

In Bild 58 siehst du, dass nur vier der acht Datenbus-Leitungen mit dem Program Counter verbunden sind. Er kann damit also Zahlen von null (`0000`) bis fünfzehn (`1111`) auf den Bus

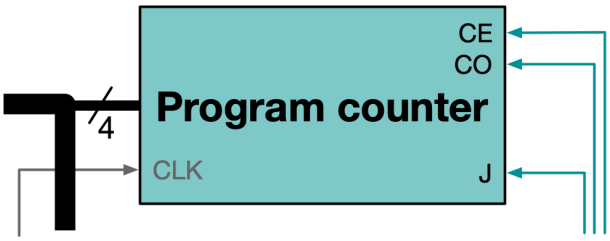


Bild 58 – Der Program Counter in der 8-Bit-CPU

schreiben. Sofern er freigegeben ist (Eingang *Counter Enable CE* = 1), zählt er bei jeder positiven Flanke des Clock-Signals (Eingang *CLK*) um eins hoch. Um zu verstehen, wie ein Zähler funktioniert, betrachte zuerst die Zahlenfolge:

Binärzahl				Dezimalwert
2^3	2^2	2^1	2^0	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Wenn du dir das niederwertigste Bit (2^0, engl. *least significant bit*, kurz LSB) anschaust, siehst du, dass es bei jedem Zählschritt zwischen 0 und 1 wechselt. Das nebenstehende Bit (2^1) wechselt halb so schnell, nach jedem zweiten Zählschritt: 00-11-00-11 usw. Das 2^2-Bit wechselt nach jedem vierten Zählschritt, also wieder halb so schnell wie dasjenige nebenan. So geht es fortlaufend weiter: das höchstwertige Bit (engl. *most significant bit*, kurz MSB) einer 8-Bit-Zahl würde erst bei der Zahl 128 zum ersten Mal von 0 auf 1 wechseln und so bleiben bis zur Zahl 255.

Für einen 4-Bit-Zähler brauchen wir also eine Schaltung mit einem Clock-Eingang und vier Ausgängen, *Q0* bis *Q3*. Der niederwertigste Ausgang soll mit jeder positiven-Clock-Flanke zwischen 0 und 1 wechseln, der nebenstehende Ausgang soll nach jeder zweiten positiven Clock-Flanke zwischen 0 und 1 wechseln etc. Auf dem Zeitdiagramm sieht das so aus:

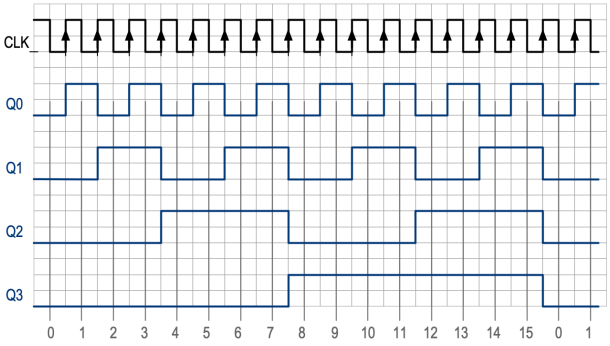


Bild 59 – Timing diagram eines 4-Bit-Zählers

In Bild 59 liesst du die vierstellige Binärzahl jeweils von unten nach oben: Ganz links ergibt sich die Zahl `0000`, dann `0001`, dann `0010` etc. Nach der Zahl `1111` beginnt der 4-Bit-Zähler wieder bei 0.

Toggeln (to toggle)

Wir suchen zunächst eine Schaltung für den Ausgang *Q0*: Mit jeder positiven Clock-Flanke wechselt der Ausgang zwischen 0 und 1. Das geht mit einem D-Flipflop, bei dem der Ausgang *Q-nicht* auf den Daten-Eingang zurückgeführt wird:

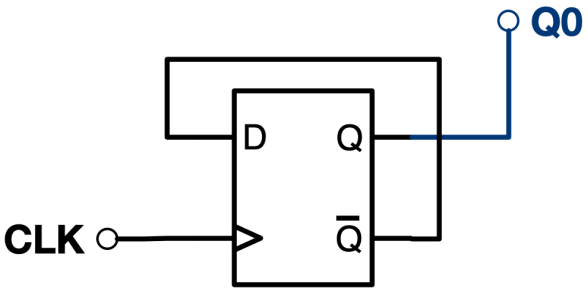


Bild 60 – Toggle-Schaltung mit D-Flipflop

Du erinnerst dich an die Funktionsweise eines D-Flipflops: Genau dann, wenn der Clock-Eingang von 0 auf 1 wechselt, wird der Wert am Dateneingang an den Ausgang übernommen und bleibt dort bis zur nächsten positiven Clock-Flanke. Am Ausgang *Q-nicht* liegt immer der zu *Q* inverse Wert. Angenommen, *Q-nicht* ist zu Beginn 0: Weil *Q-nicht* mit *D* verbunden ist, wird bei der ersten positiven Flanke eine 0 an den Ausgang *Q* übernommen – in diesem Moment wechselt *Q-nicht* auf 1. Bei der nächsten Flanke wird also eine 1 an *Q* übernommen, womit *Q-nicht* wieder 0 ist. So geht das immer weiter: Der Ausgang Q-wechselt stets zwischen 0 und 1. Dieses Wechseln zwischen Zuständen wird auch „Toggeln“ genannt.

Eine 4-Bit-Zählschaltung

Du hast vielleicht schon gemerkt, dass die eben betrachtete Toggle-Schaltung im Grunde bloss ein Clock-Signal ausgibt, das halb so schnell zwischen 0 und 1 wechselt, wie das Clock-Signal am Eingang. Wenn du auf Bild 59 schaust, siehst du: Genau diese Schaltung brauchen wir auch zwischen *Q0* und *Q1*. Wir können also einfach *Q0* auf den Clock-Eingang einer zweiten Toggle-Schaltung führen und erhalten so den Ausgang Q1.

Allerdings soll *Q1* bei jeder *negativen* Flanke von *Q0* wechseln (siehe Bild 59). Das ist kein Problem: Jede negative Flanke von *Q0* ist eine positive Flanke von *Q0-nicht*. Wir müssen also *Q0-nicht* auf den Clock-Eingang der nächsten Toggle-Schaltung führen. Für einen 4-Bit-Zähler brauchen wir vier hintereinander geschaltete Toggle-Schaltungen:

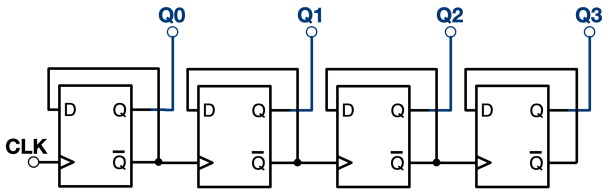


Bild 61 – 4-Bit-Zählschaltung

So einfach ist es, eine Zählschaltung zu bauen! Schaue nun dieses Video, indem der 4-Bit-Zähler nochmals kurz erklärt und vorgeführt wird.

Der 4-Bit-Zähler 74LS161

Natürlich sind Zählschaltungen auch „fixfertig“ als ICs erhältlich, sodass man nicht jedes Mal mehrere Flipflops zusammenschalten muss, wenn man eine

Zählschaltung braucht. Es gibt ganz viele unterschiedliche Zähler-ICs, einer davon ist der 74LS161, sein Symbol sieht so aus:

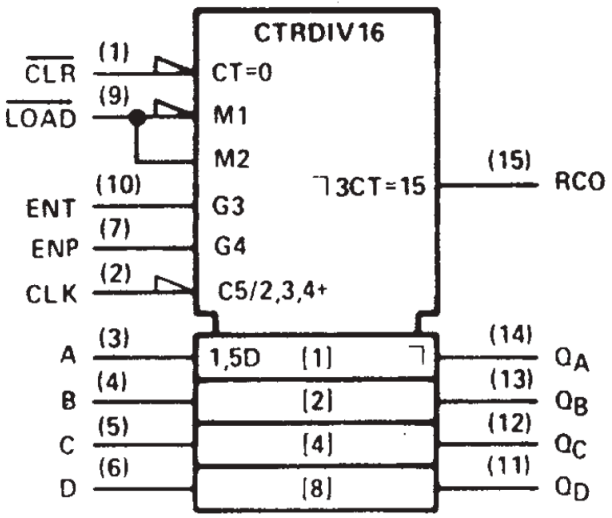


Bild 62 – Symbol des 4-Bit-Zähler-ICs 74LS161

Dieses Symbol sieht ähnlich aus wie das Symbol des 4-Bit-Registers 74LS173 in Bild 34 (Teil 5). Das macht Sinn, denn dieser Zähler ist gleichzeitig ein Register! Zähler und Register sind ja ähnlich aufgebaut: wie ein 4-Bit-(Schiebe-)Register besteht auch ein 4-Bit-Zähler aus vier Flipflops – sie unterscheiden sich nur in der Art und Weise, wie diese Flipflops miteinander verbunden sind. Wenn du dir die Logikschaltung des 74LS161 auf Seite 2 des Datenblatts anschaust, siehst du, dass die vier Flipflops in eher komplexer Weise miteinander verbunden sind. Diese Schaltung ermöglicht folgende Funktionen:

- Mit dem CLR-Eingang werden die Ausgänge auf 0 gesetzt. Der Zählstand ist dann `0000`.
- Mit dem LOAD-Eingang werden die Ausgänge auf die Werte gesetzt, die an den Eingängen A… D anliegen. Damit lässt sich der Zähler auf einen bestimmten Zählstand setzen, von dem aus er dann weiter hoch zählt.
- Mit den Eingängen ENT und ENP kann der Zähler freigeben oder gesperrt werden. Ist der Zähler gesperrt, so zählt er bei der nächsten Clock-Flanke nicht weiter.

Für den Program Counter in unserer 8-Bit-CPU ist der 74LS161 gut geeignet. Um zu verstehen warum, nimm Bild 1 dieses Skripts zur Hand und lies folgende Beschreibung:

Die CPU führt ein Programm aus. Dieses besteht aus einer Reihe von Befehlen. Die Befehle werden im RAM abgespeichert – jeder Befehl an einer anderen Adresse. Der Program Counter gibt die RAM-Adresse an, an welcher der nächste Befehl (*instruction*) liegt. Normalerweise zählt er um eins hoch, sobald alle Anweisungen (*micro instructions*) für den aktuellen Befehl ausgeführt sind. So wird zuerst der Befehl aus Adresse `0000` geladen, dann aus `0001`, dann `0010` und so weiter. Nun gibt es auch Jump-Befehle, die es erlauben, an eine bestimmte Adresse im Programm zu springen. Angenommen, das Programm im RAM besteht aus vier Befehlen:

RAM Adresse	Befehl in assembly language	Befehl in Maschinensprache
`0000`	`LDA 7`	`0001'0111`
`0001`	`ADD 6`	`0010'0110`
`0010`	`OUT`	`0100'0000`
`0011`	`JMP 1`	`0111'0001`

Dieses Programm macht folgendes:

- Der Wert aus Adresse 7 wird ins A-Register geladen (LDA 7).
- Dann wird der Wert aus Adresse 6 zum Wert im A-Register hinzuaddiert und das Resultat der Addition wird ins A-Register geladen (ADD 6).
- Dann wird das Resultat ins Output-Register geladen, wo es angezeigt wird (OUT).
- Dann wird der Program Counter auf den Wert 1 gesetzt (JMP 1), wodurch als nächstes wieder der Befehl in Adresse 1 geladen und ausgeführt wird. So entsteht eine Schleife: Die Befehle in Adressen 1 bis 3 werden endlos wiederholt.

Angenommen, im RAM steht unter Adresse `0111` die Zahl 12 und unter Adresse `0110` die Zahl 22: Dann gibt die CPU, die obiges Programm ausführt, folgende Zahlen aus: 34, 56, 78, 100, 122, 144, 166, 188, 210, 232, 254 – und dann?

Der JMP-Befehl lässt sich mit dem 74LS161 ist gut umsetzen, weil dieser Zähler auf einen bestimmten Wert gesetzt werden kann: Hier der Aufbau des Program Counters:

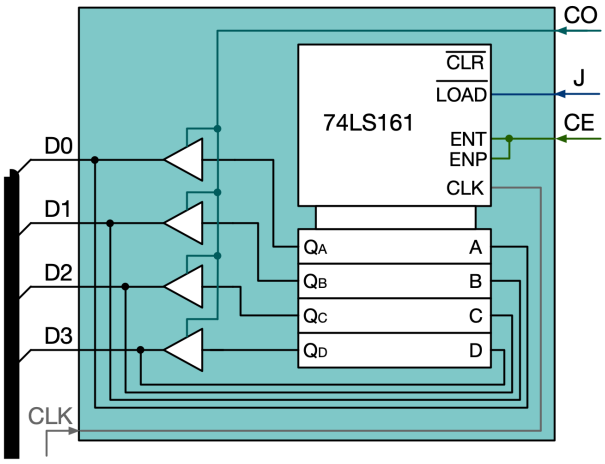


Bild 63 – Die Schaltung des Program Counters

Die vier Datenbusleitungen D0…D3 sind über die Tri-State-Buffer mit den Ausgängen des 74LS161 verbunden. Sie sind aber auch mit den Eingängen des 74LS161 verbunden: Ist das Signal *CO* aktiv, so schreibt der Program Counter den Zählstand auf den Bus. Ist das Signal J aktiv, so wird der Wert auf dem Bus in den 74LS161 übernommen. Ist das Signal *CE* aktiv, so wird der Zählstand mit jeder positiven Clock-Flanke um eins erhöht. Schaue nun dieses Video zur Funktion und dieses Video zum Aufbau des Program Counters.

Decoder

Wenn du dir die Schaltung für das programmierbare Lichtmuster (Bild 44, Teil 6) anschaust, siehst du, dass dort ein Baustein namens „Counter/Decoder“ dafür sorgt, dass von den acht Schieberegistern eines nach dem andern aktiviert wird. Du weisst nun, wie ein Zähler funktioniert: Um acht Ausgänge anzusteuern, muss ein Zähler von 0 bis 7 zählen können. Dazu braucht es bloss einen 3-Bit-Zähler, denn dieser zählt von `000` bis `111`. Aber der „Counter/Decoder“ hat acht Ausgänge – nicht drei. Es ist der Decoder, der aus drei Ausgängen acht macht. Im Counter/Decoder sind zwei Bausteine untergebracht:

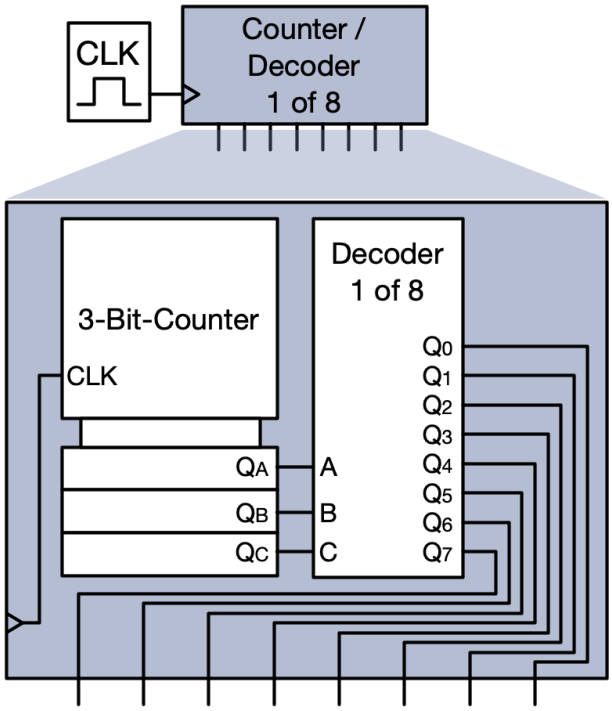


Bild 64 – Counter und Decoder in einem Baustein

Der 1-of-8-Decoder in Bild 64 hat also die Aufgabe, aufgrund der Binärzahl an den drei Eingängen einen der acht Ausgänge zu setzen. Wir können es etwas allgemeiner formulieren:

Für eine bestimmte Kombination aus Einsen und Nullen an den Eingängen setzt der Decoder eine gewünschte Kombination aus Einsen und Nullen an den Ausgängen.

Diese Beschreibung trifft auf alle Decodertypen zu: Decoder enthalten also eine *combinational logic*. Der 1-of-8-Decoder muss folgende Wahrheitstabelle erfüllen:

C	B	A	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**

Aufgrund der Wahrheitstabelle kann für jeden der acht Ausgänge eine Funktionsgleichung und eine

Logikschaltung erstellt werden. Die Funktionsgleichung für Q_0 lautet:

$$Q_0 = \overline{A} \wedge \overline{B} \wedge \overline{C}$$

Es werden also AND- und NOT-Gatter benötigt. Hier die komplette Schaltung:

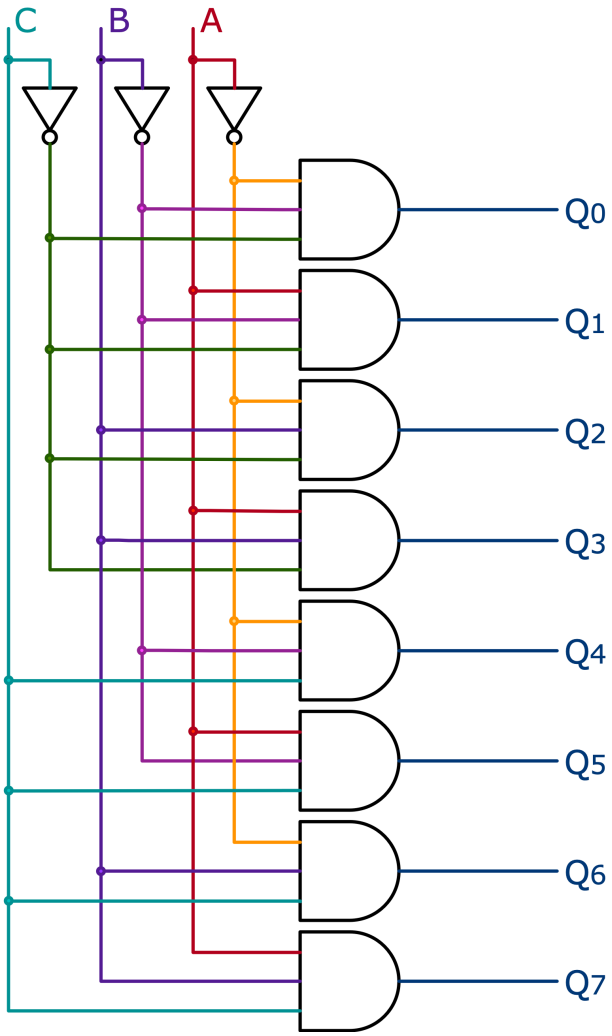


Bild 65 – Logik im 1-of-8-Decoder

Ein Binär-zu-7-Segment-Decoder

Auf ähnliche Weise lassen sich andere Decoder bauen – etwa ein Binär-zu-7-Segment-Decoder:

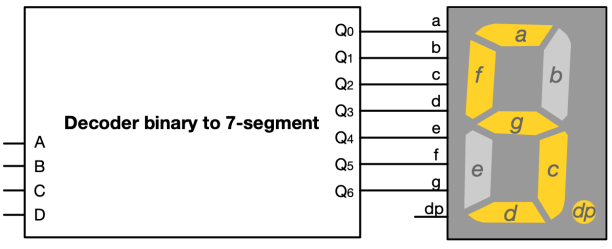


Bild 66 – Binär-zu-7-Segment-Decoder

Wenn du zum Beispiel den Wert eines 4-Bit-Zählers nicht in der Binärdarstellung mit vier LEDs anzeigen möchtest, kannst du einen Binär-zu-7-Segment-Decoder bauen. Damit kann der binäre Zählstand `(0000…1111)` als Hexadezimalzahl `(0…F)` angezeigt werden. Die Wahrheitstabelle für einen solchen Decoder sieht so aus:

D	C	B	A	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	0	1	1	1	0	1	1	1	A
1	0	1	1	0	0	1	1	1	1	1	b
1	1	0	0	1	0	0	1	1	1	0	C
1	1	0	1	0	1	1	1	1	0	1	d
1	1	1	0	1	0	0	1	1	1	1	E
1	1	1	1	1	0	0	0	1	1	1	F

Wenn du dir zum Beispiel die Funktionsgleichung für den Ausgang **a** anschaust:

$$a = \overline{\overline{a}} = \overline{(A \wedge \overline{B} \wedge \overline{C} \wedge \overline{D}) \vee (\overline{A} \wedge \overline{B} \wedge C \wedge \overline{D}) \vee (A \wedge B \wedge \overline{C} \wedge D) \vee (A \wedge \overline{B} \wedge C \wedge D)}$$

…dann merkst du wohl, dass die Logikschaltung im 7-Segment-Decoder viel komplexer wird als diejenige im 1-of-8-Decoder. Es gibt aber eine Möglichkeit, *combinational logic* ohne unzählige Logikgatter zu realisieren. Mehr dazu im nächsten Teil.

Für eine ausführliche Erklärung zum 7-Segment-Decoder schaust du dieses Video.

Teil 9 – Speicherbausteine RAM und ROM

Schau du dir Bild 1 dieses Skripts an und überlege, welche Module dieser CPU du schon bauen oder zumindest verstehen kannst:

Das **Clock-Modul** gibt das Clock-Signal (steter Wechsel von 0 auf 1 auf 0…) aus. Wie dieses zustande kommt, werden wir nicht genauer betrachten. Den **Program Counter** hast du eben kennen gelernt. Du weisst nun, was er tut, wozu er da ist und wie er aufgebaut ist. Auch die **ALU** hast du in Teil 7 angeschaut. Du weisst nun, was sie tut und wie sie aufgebaut ist. Die **Register A und B** dienen dazu, 8-Bit-Werte zu speichern, die dann von der ALU verrechnet, via Output Register ausgegeben oder im RAM gespeichert werden. Du weisst nun, wie Register aufgebaut sind und wie sie funktionieren.

Das **Memory Address Register** ist ebenfalls bloss ein Register. Anders als die Register A und B speichert es nur einen 4-Bit-Wert. Wie der Name schon sagt, ist dieses Register dazu da, die Memory-Adresse (die RAM-Adresse) zu speichern. Diese kommt entweder vom Program Counter oder vom Instruction Register. Hier im Memory Address Register wird sie gespeichert und ans RAM weitergeleitet: Die 4-Bit-Adresse, die im Memory Address Register gespeichert ist, bestimmt, welcher RAM-Speicherplatz ausgewählt und später mit dem Datenbus verbunden wird.

Das **Instruction Register** ist wie die Register A und B ein 8-Bit-Register. Es speichert den Befehl (die *instruction*). Hier nochmal das Beispiel-Programm aus der Beschreibung des Program Counters:

RAM Adresse	Befehl in assembly language	Befehl in Maschinensprache
0000	LDA 7	0001'0111
0001	ADD 6	0010'0110
0010	OUT	0100'0000
0011	JMP 1	0111'0001

Du siehst, dass ein Befehl aus zwei Teilen besteht: Die linken vier Bits enthalten den Befehls-Code. In diesem Beispiel steht der Code `0001` für den Befehl LDA und der Code `0010` für den Befehl ADD. Die rechten vier Bits enthalten den Wert zu diesem Befehl – meistens eine Adresse: Der Befehl LDA bedeutet: „Lade den Wert *aus der angegebenen Adresse* ins Register A“. Also muss zu diesem Befehl noch eine Adresse angegeben werden. Im Beispiel-Programm wird die Adresse 7 angegeben, das heisst: Es wird der Wert aus dem RAM-Speicherplatz 7 ins A-Register geladen. Einige Befehle wie z. B. OUT brauchen keinen Wert – bei diesen Befehlen spielt es keine Rolle, welche Werte die rechten vier Bits haben.

Das **Output Register** enthält ebenfalls ein 8-Bit-Register, das den Wert, der angezeigt werden soll, speichert. Ausserdem enthält dieses Modul einen Decoder, der diesen 8-Bit-Wert so umwandelt, dass er auf vier 7-Segment-Anzeigen (für Zahlen von 0 bis 255 und ein Minus-Zeichen) als Dezimalzahl angezeigt wird. Der Decoder besteht aus einem vorprogrammierten EEPROM.

Es bleiben noch die Module **Control Logic** und **RAM**. Auch die Control Logic besteht im Wesentlichen aus einem vorprogrammierten EEPROM und das RAM-Modul besteht hauptsächlich aus einem SRAM-Baustein. Um diese Module zu verstehen, müssen wir zuerst die Speichertypen RAM und ROM kennen.

RAM (random access memory)

Auf einem RAM können Daten sowohl gespeichert (write) als auch ausgelesen (read) werden. RAM erlauben den Zugriff (access) auf ein beliebiges (random) Speicherelement in gleicher Zeit. Bei älteren Speicherformen (Festplatte, Floppy-Disk) hängt die Zugriffszeit davon ab, wo sich das Element auf dem Speicher befindet. RAM sind flüchtige Speicher, das heisst, die Daten gehen verloren, sobald die Spannungsversorgung abbricht.

Du kannst ein RAM als einen Stapel von Registern betrachten: Sind in einem RAM sechzehn 8-Bit-Register enthalten, so hat es eine Kapazität von 16 Byte. Da ein Register aus mehreren 1-Bit-Speicherzellen besteht, können wir auch sagen: Ein RAM besteht aus einer Matrix an Speicherzellen.

Static RAM (SRAM)

In einem SRAM werden Flipflops für die Speicherzellen verwendet. Hier siehst du einen möglichen Aufbau einer dieser Speicherzellen:

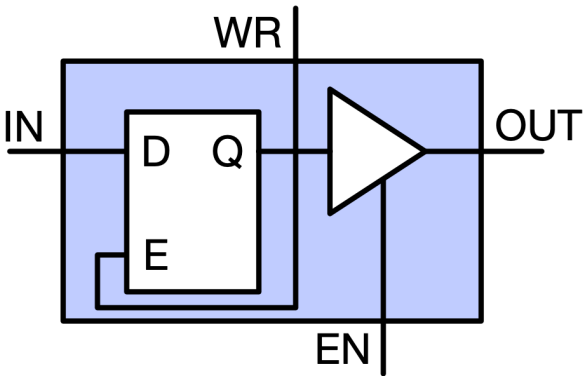


Bild 67 – Speicherzelle (binary cell) eines D-RAM

Die Speicherzelle hat einen Dateneingang IN, einen Datenausgang OUT sowie die Eingänge WR und EN. Sie enthält ein pegelgesteuertes D-Flipflop und einen Tri-State-Buffer. Ist der Write-Eingang WR = 1, so ist das Flipflop freigegeben: Der Zustand am Dateneingang wird an den Ausgang Q des Flipflops übernommen. Geht WR wieder auf 0, so bleibt der Zustand am Ausgang Q des Flipflops gespeichert, eine Änderung am Dateneingang hat nun keinen Einfluss. Solange der Enable-Eingang EN = 0 ist, ist der Ausgang des Tri-State-Buffers $\mathcal{Z}$ *(„nicht verbunden“)*. Geht EN auf 1, so liegt am Ausgang OUT der Zustand, der am Ausgang Q des Flipflops anliegt. Kurz: Mit dem Eingang WR wird der Zustand am Eingang IN in die Speicherzelle geschrieben; mit dem Eingang EN wird der Zustand in der Speicherzelle an den Ausgang OUT ausgegeben. Hier kann er ausgelesen werden.

Mit solchen Speicherzellen lassen sich nun beliebig grosse RAM bauen. Folgend der Aufbau eines kleinen 32-Bit-RAM (8 mal 4-Bit):

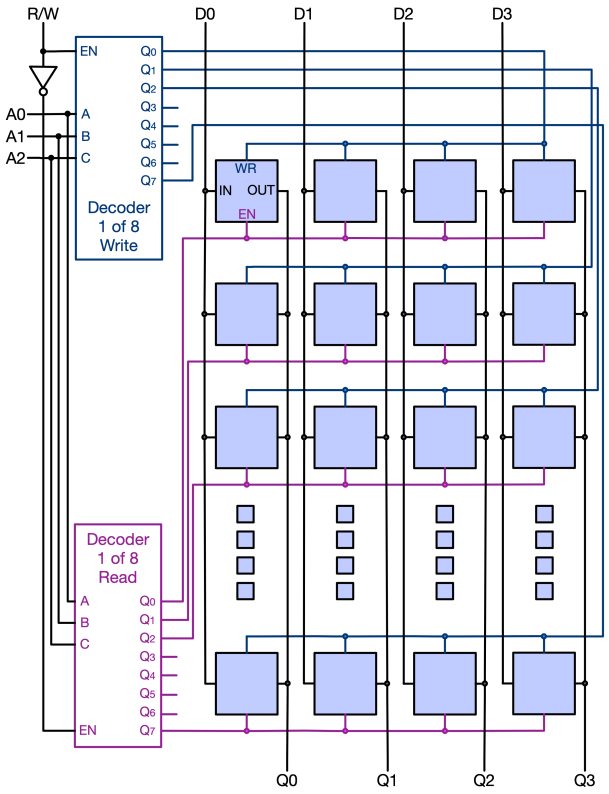


Bild 68 – Prinzipielle Schaltung eines 32-Bit-SRAM

Das SRAM in Bild 68 hat vier Dateneingänge D0…D3 und vier Datenausgänge Q0…Q3. Es können also vier Bits gleichzeitig ins RAM geschrieben oder ausgelesen werden. Die 32 Speicherzellen sind in 8 Zeilen und 4 Spalten angeordnet. Jede Zeile entsprich einer Speicher-Adresse. Über die Address-Eingänge A0…A3 wird eine der Zeilen angesteuert. Das geschieht über die Decoder: Wenn der Eingang R/W (Read/Write) = 1 ist, ist der Write-Decoder aktiv, wenn er 0 ist, ist der Read-Decoder aktiv.

Angenommen, an den Adress-Eingängen liegt der Wert `010` an und R/W ist 1: Jetzt ist der Write-Decoder und dessen Ausgang Q2 aktiv. Das heisst, dass der Write-Eingang jeder Speicherzelle in der dritten Zeile aktiv ist. Daten können so in Adresse 3 (`010`) des RAMs geschrieben werden. Wechselt R/W nun auf 0, so können die Daten aus der Adresse 3 des RAMs ausgelesen werden.

Dynamic RAM (DRAM)

In einem DRAM bestehen die Speicherzellen aus einfachen Kondensator-Transistor-Schaltungen.

Weil eine Ladung in einem Kondensator nur einige Sekunden erhalten bleibt, müssen die Daten in einem DRAM ständig aktualisiert werden. Das geschieht durch eine interne Schaltung. Die Daten bleiben nicht wie beim SRAM „statisch“ erhalten. Deshalb nennt man es „dynamisches“ RAM. DRAM sind langsamer als SRAM, benötigen dafür weniger Platz: Die Speicherzelle eines DRAMs enthält nur einen Transistor – die eines SRAMs viele, denn Flipflops bestehen ja aus vielen Transistoren. Static RAM sind damit grösser und teurer als Dynamic RAM.

Schaue dieses Video zum Aufbau eines SRAMs und zum Unterschied zwischen SRAM und DRAM.

Das RAM-Modul in der 8-Bit-CPU

Für das RAM-Modul in der 8-Bit-CPU wird der IC 74LS189 verwendet. Sein Symbol sieht so aus:

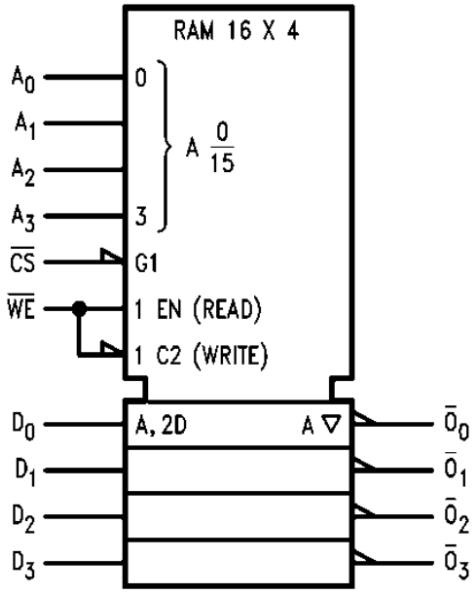


Bild 69 – Symbol des 64-Bit-RAM 74LS189

Im Symbol in Bild 69 siehst du, dass der 74LS189 vier Daten-Eingänge, vier Daten-Ausgänge und vier Adress-Eingänge hat. Es können damit $2^4 = 16$ Adressen angesteuert werden. Das RAM verfügt damit über 16 Speicherplätze mit je 4 Bits. Die Speicher-Kapazität beträgt damit 64 Bit oder 8 Byte. Allerdings können nur 4 Bits gleichzeitig geschrieben oder gelesen werden. Für die 8-Bit-CPU werden deshalb zwei dieser ICs verwendet, die Gesamtkapazität beträgt damit 16 Byte:

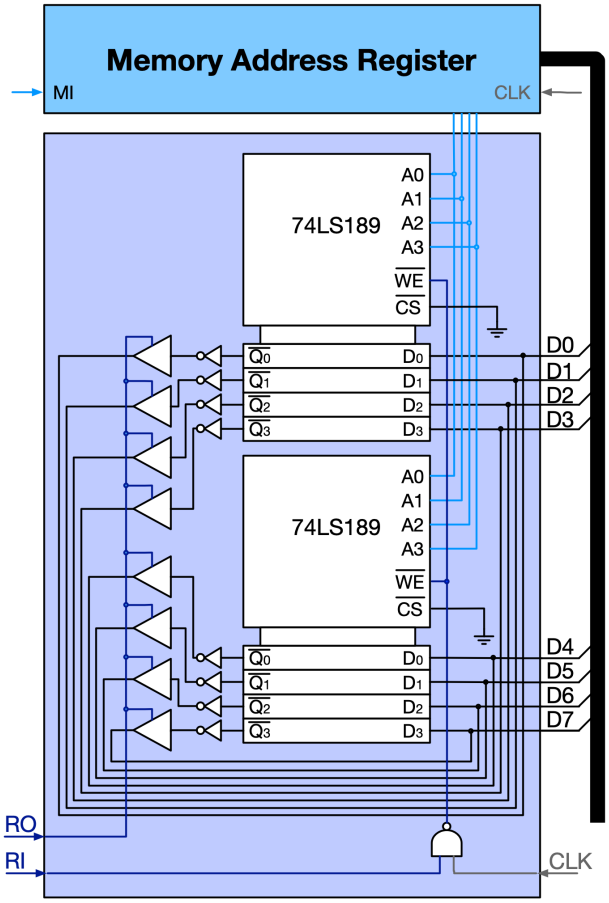


Bild 70 – Schaltung des RAM-Moduls (vereinfacht)

In Bild 70 siehst du, wie die vier Adress-Leitungen vom Memory Address Register auf die beiden RAM-ICs geführt sind. Der obere IC speichert die vier LSB (*least significant Bits*), der untere die vier MSB (*most significant Bits*) vom Datenbus. Da die Ausgänge der beiden ICs jeweils invertiert sind, müssen sie über die NOT-Gates wieder zurück-invertiert werden. Danach sind die Ausgänge auf Tri-State-Buffer geführt: Nur wenn der Eingang RO (RAM Out) aktiviert ist, werden die Werte aus jenem RAM-Speicherplatz, der durch die Address-Eingänge ausgewählt ist, auf den Bus geschrieben. Damit Daten ins RAM geschrieben werden können, muss erstens der Eingang RI (RAM In) aktiv sein und zweitens eine positive Clock-Flanke erfolgen: Dann ist der Write-Enable-Eingang (WE) für kurze Zeit aktiv und die Werte vom Datenbus werden in die ausgewählte Speicher-Adresse geschrieben.

Dieses Video zeigt den ersten Teil zum Aufbau des RAM-Moduls wie du es in Bild 70 (vereinfachte Schaltung) siehst.

Die komplette Schaltung des RAM-Moduls ist ein bisschen komplexer: Ein Programm, das auf der 8-Bit-CPU läuft, muss ja irgendwie erst ins RAM geschrieben werden. Dies geschieht manuell über kleine DIP-Schalter und Tasten. Im RAM-Modul und im Memory Address Register sind also zusätzliche Schaltungsteile vorhanden, die es erlauben, zwischen Programmier- und Lauf-Modus umzuschalten: Im Programmiermodus kann das Programm und andere Daten von Hand ins RAM geschrieben werden; im Laufmodus wird dieses Programm dann ausgeführt. Wenn du wissen willst, wie diese Schaltungsteile funktionieren, schaue die Folge-Videos (part 2, part 3 und testing and trouble shooting).

ROM (Read-only memory)

Wie der Name sagt, ist ein ROM ein Speichertyp, der nur gelesen (read-only) werden kann. Vielleicht kennst du noch die (nicht wiederbeschreibbare) CD-ROM. Ist die CD einmal gebrannt, bleiben die Daten drauf und man kann sie nur noch lesen. ROM sind nicht-flüchtige Speicher (auch *Festspeicher* genannt), das heisst, die Daten bleiben auch ohne Spannungsversorgung erhalten.

Vom ROM zum EEPROM

ROM-Bausteine gab es als ICs. Sie hatten fixe, vorgegebene Daten drauf. Diese Bausteine wurden weiterentwickelt zu PROM (*programmable* ROM) die man mit einem Programmier-Gerät einmal programmieren und in eine Schaltung einbauen konnte. Eine Weiterentwicklung davon war das EPROM (*erasable programmable* ROM), das man per UV-Licht *löschen* und dann mit dem Programmiergerät neu programmieren konnte.

Für ein Software-Upgrade musste man dann "bloss" noch das EPROM aus der Schaltung

Bild 71 – Programmiergerät für PROM und EPROM

nehmen, es einige Zeit unter die UV-Lampe legen, dann im Programmiergerät neu „brennen“ und wieder in die Schaltung einsetzen.

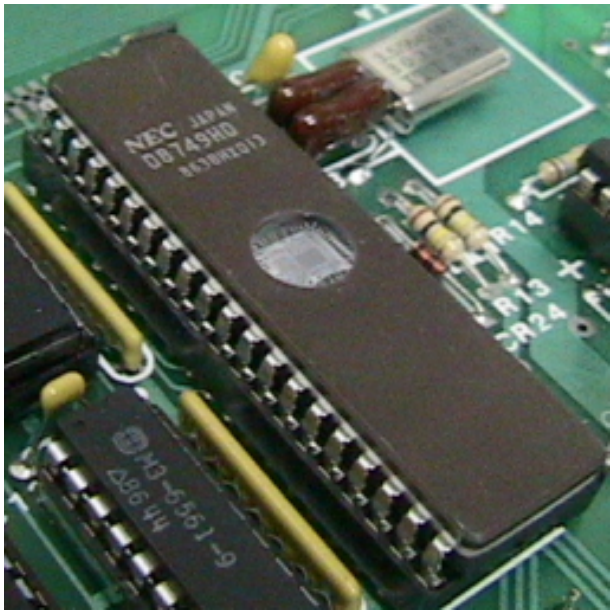


Bild 72 – EPROM mit Fenster fürs Löschen der Daten mit UV-Licht

Schliesslich wurde das EEPROM (*electrically erasable programmable* ROM) entwickelt. Das nun *elektrisch* gelöscht und programmiert werden konnte. Ein EEPROM kann damit in der laufenden Schaltung neu beschrieben werden.

Je fortgeschrittener die Entwicklung, desto weniger trifft die Bezeichnung „Read-only memory" zu: Ein EEPROM kann wie ein RAM jederzeit gelesen *und* beschrieben werden.

EEPROMs in der 8-Bit-CPU

Die 8-Bit-CPU hat keinen Festspeicher. Der einzige Ort, an dem Daten und Programme gespeichert werden, ist das RAM. Jedes Mal, wenn die Spannungsversorgung wegfällt, gehen alle Daten im RAM verloren. Somit muss die 8-Bit-CPU bei jedem Einschalten neu programmiert werden.

Zwar sind in der 8-Bit-CPU auch EEPROMs – und damit Festspeicher – eingebaut, doch diese dienen nicht dem Speichern von Programmen und Daten, sondern als Decoder. In der 8-Bit-CPU wird das EEPROM 28C16 verwendet:

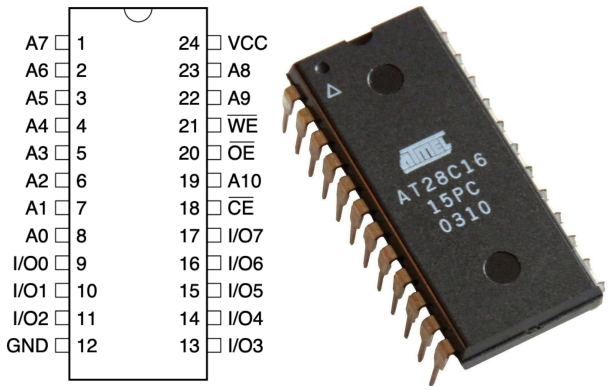


Bild 73 – Pin-Belegung/Bild des EEPROMs 28C16

In Bild 73 siehst du, dass das 28C16 elf Adress-Eingänge (A0…A10) und acht Daten-Ein- und Ausgänge (I/O0…I/O7) hat. Mit 11 Adress-Eingängen können $2^{11} = 2048$ verschiedene Adressen ausgewählt werden. Das EEPROM hat also 2048 Speicherplätze. An jedem Speicherplatz kann ein Byte (8 Bits) gespeichert werden. Die Kapazität des 28C16 beträgt also 2 Kilobytes.

Wie wird ein EEPROM als Decoder verwendet? Du erinnerst dich: Ein Decoder enthält eine *combinational logic*, das heisst: Für eine bestimmte Kombination aus Einsen und Nullen an den Eingängen wird eine bestimmte Kombination aus Einsen und Nullen an den Ausgängen gesetzt. Ein einfacher Binär-zu-7-Segment-Decoder (vgl. Bild 66, Teil 8) hat 4 Eingänge und 7 Ausgänge (falls das achte Segment, der Punkt, ebenfalls angesteuert werden soll, hat er 8 Ausgänge). Hier die ersten 3 von 16 Zeilen der Wahrheitstabelle aus Teil 8:

D	C	B	A	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2

Für jede der 16 Kombinationen an den Eingängen (A…D) werden bestimmte Ausgänge (a…g) auf 0 oder 1 gesetzt. Statt nun dutzende AND- und NOT-Gates zu verwenden, nimmst du einfach ein EEPROM: Die Address-Eingänge sind die Eingänge, die Datenleitungen die Ausgänge des Decoders. In Adresse `0000` speicherst du den Wert `1111110`, in Adresse `0001` den Wert `0110000` usw. Die Wahrheitstabelle zeigt dir für jede Speicher-Adresse die Werte für die Datenleitungen. Wenn du das programmierte EEPROM dann in die Schaltung einbaust und in den Lesemodus setzt, funktioniert es als Decoder: Liegt an den Adress-Eingängen der Wert 2 (`0010`), so liegt an den Datenleitungen der Wert `1101101` an, wodurch die Anzeige eine 2 anzeigt.

Im **Output-Register** der 8-Bit-CPU werden vier 7-Segment-Anzeigen angesteuert. Der Decoder, der diese Anzeigen ansteuert, hat nicht vier, sondern acht Eingänge. Das 28C16 mit seinen elf Adress-Eingängen reicht dazu aus. Schaue dieses Video zur Entwicklung vom ROM bis zum EEPROM und dazu, wie du mit einem EEPROM einen Binär-zu-7-Segment-Decoder baust.

Die **Control Logic** könnte man auch *Instruction Decoder* nennen: In der Control Logic wird der Befehl, der vom Instruction Register kommt, so decodiert, dass die Steuerleitungen, die zu den verschiedenen Modulen führen, richtig gesetzt werden. Genaueres zu Aufbau und Funktion der Control Logic erfährst du in dieser Video-Reihe.