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:

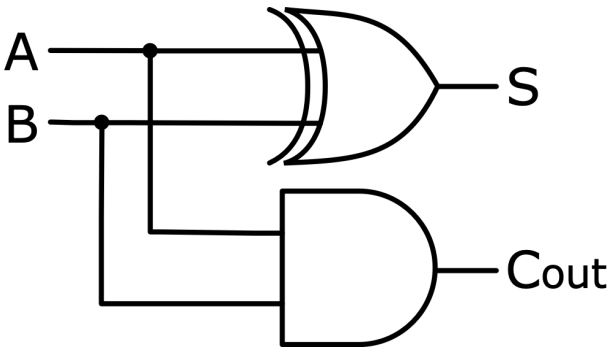

Bild 47 – Halbaddierer aus XOR und AND

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:

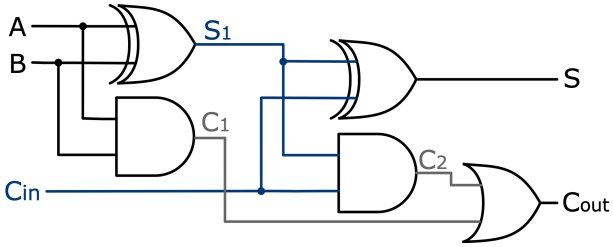

Bild 48 – Volladdierer aus zwei Halbaddierern und OR

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:

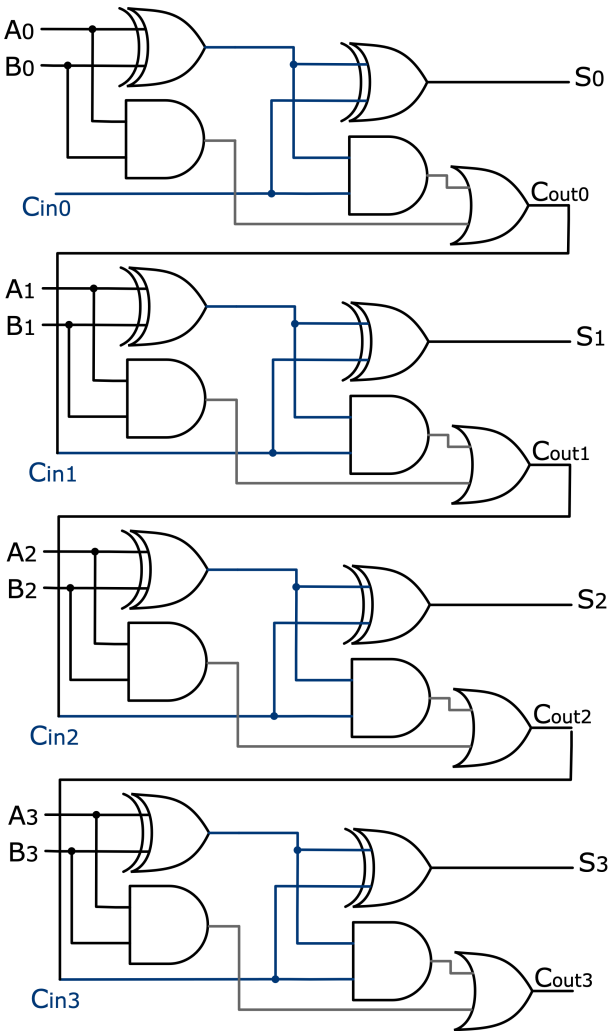

Bild 49 – 4-Bit-Addierschaltung aus 4 Volladdierern

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

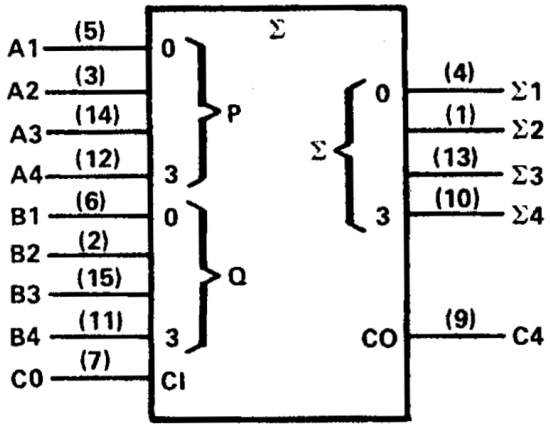

Bild 50 – 4-Bit-Volladdierer-IC 74LS283

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:

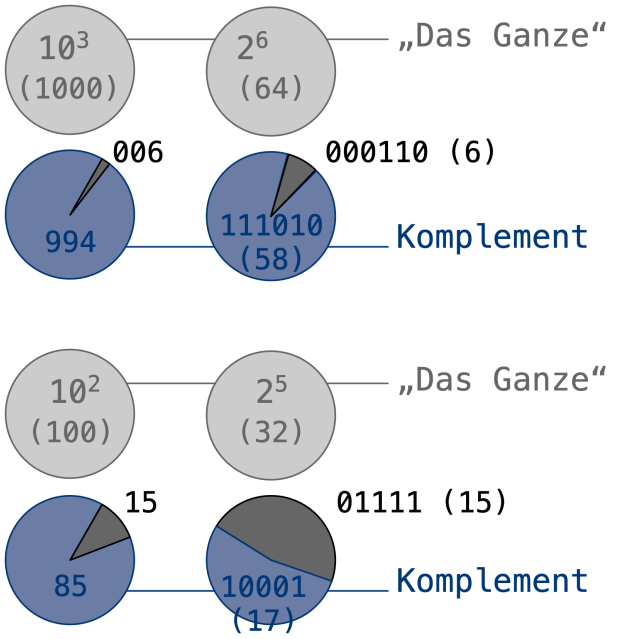

Bild 53 – Komplement ergänzt Zahl auf das Ganze

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:

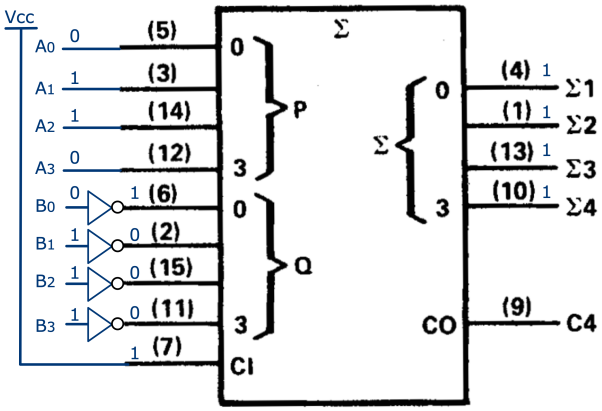

Bild 54 – Subtrahierer mit Volladaddier-IC 74LS283

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:

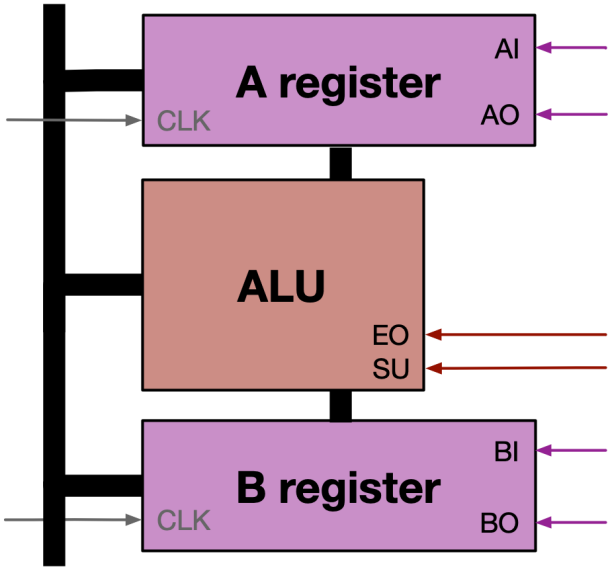

Bild 56 – 8-Bit-ALU, A-Register und B-Register

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:

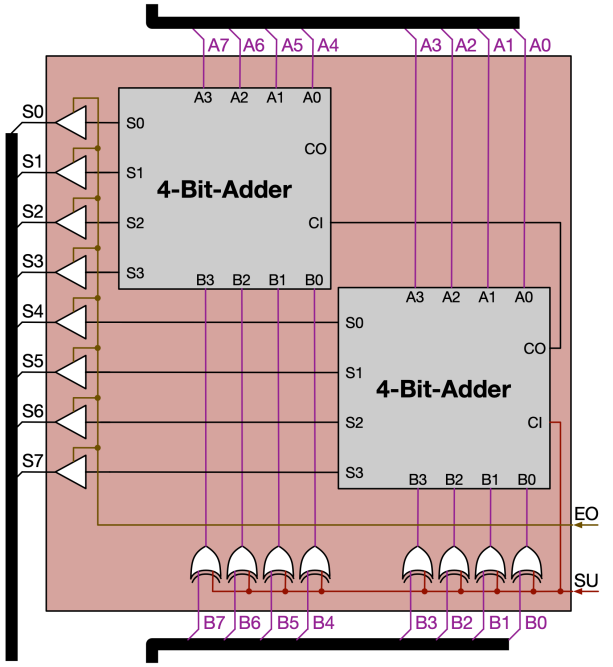

Bild 57 – Die Schaltung in der der 8-Bit-ALU

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

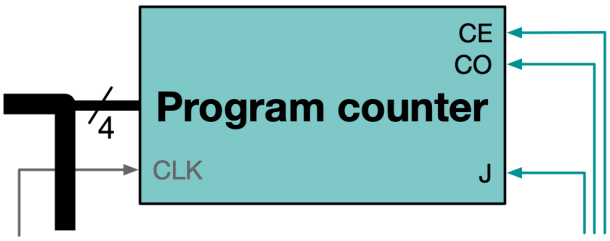

Bild 58 – Der Program Counter in der 8-Bit-CPU

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:

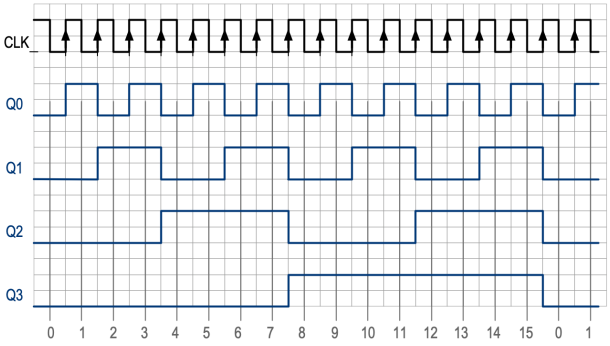

Bild 59 – Timing diagram eines 4-Bit-Zählers

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:

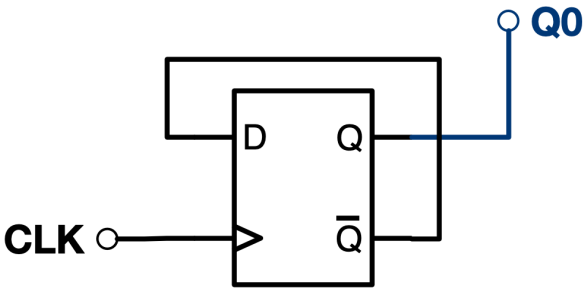

Bild 60 – Toggle-Schaltung mit D-Flipflop

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:

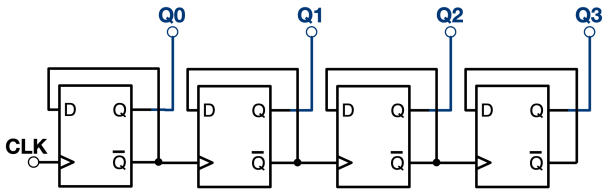

Bild 61 – 4-Bit-Zählschaltung

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:

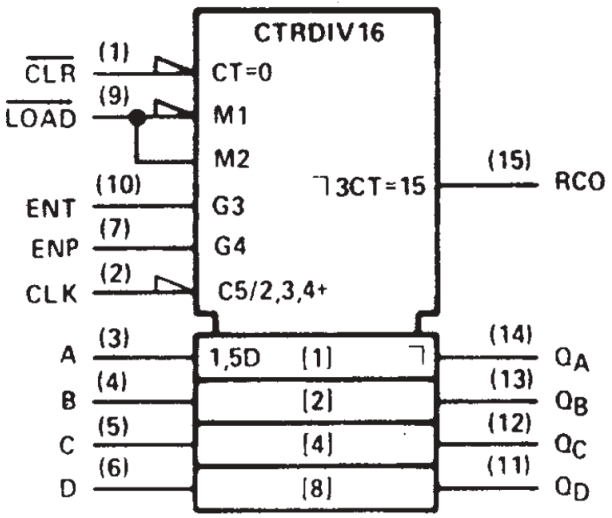

Bild 62 – Symbol des 4-Bit-Zähler-ICs 74LS161

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:

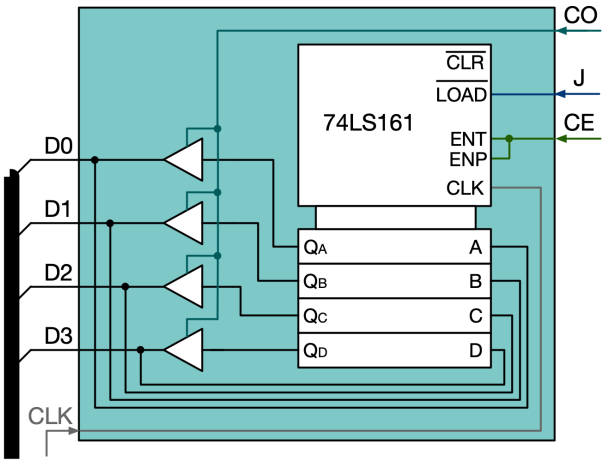

Bild 63 – Die Schaltung des Program Counters

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:

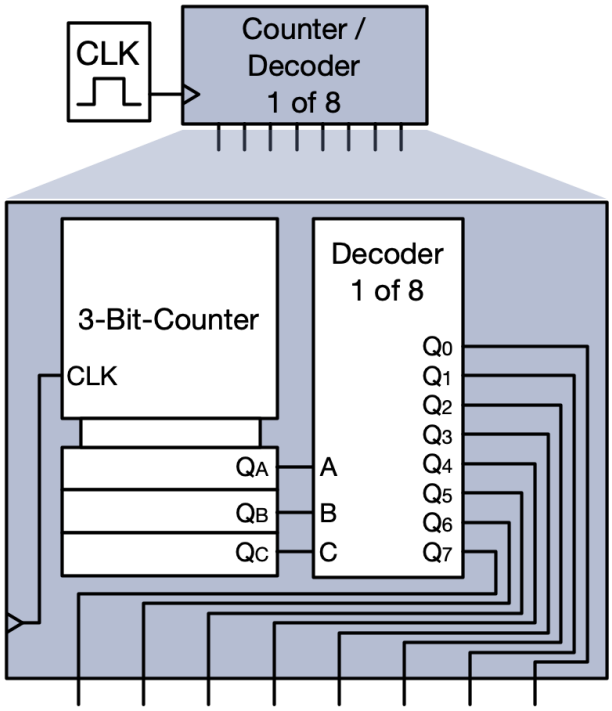

Bild 64 – Counter und Decoder in einem Baustein

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:

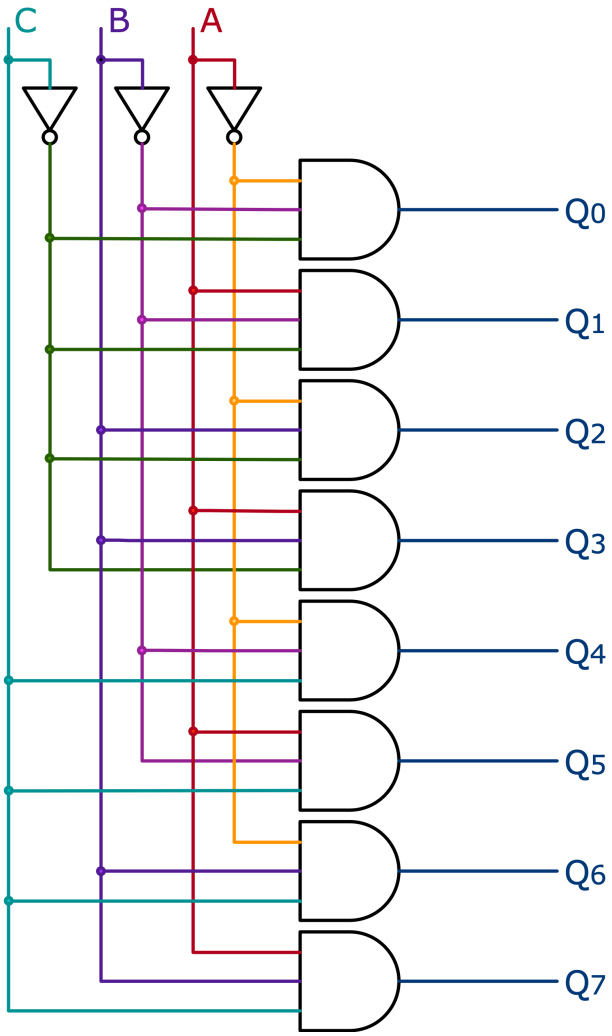

Bild 65 – Logik im 1-of-8-Decoder

Ein Binär-zu-7-Segment-Decoder

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

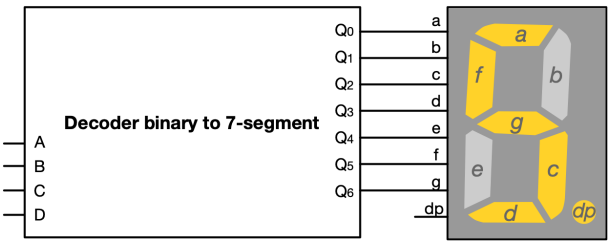

Bild 66 – Binär-zu-7-Segment-Decoder

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:

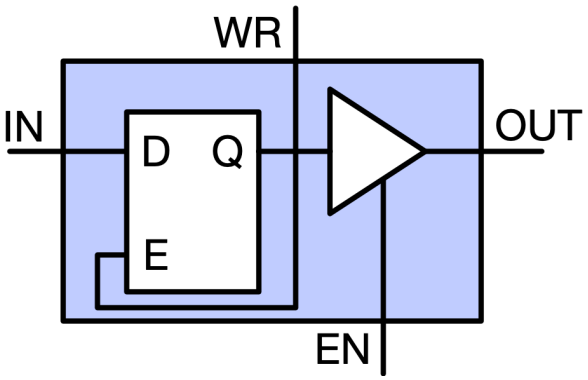

Bild 67 – Speicherzelle (binary cell) eines D-RAM

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):

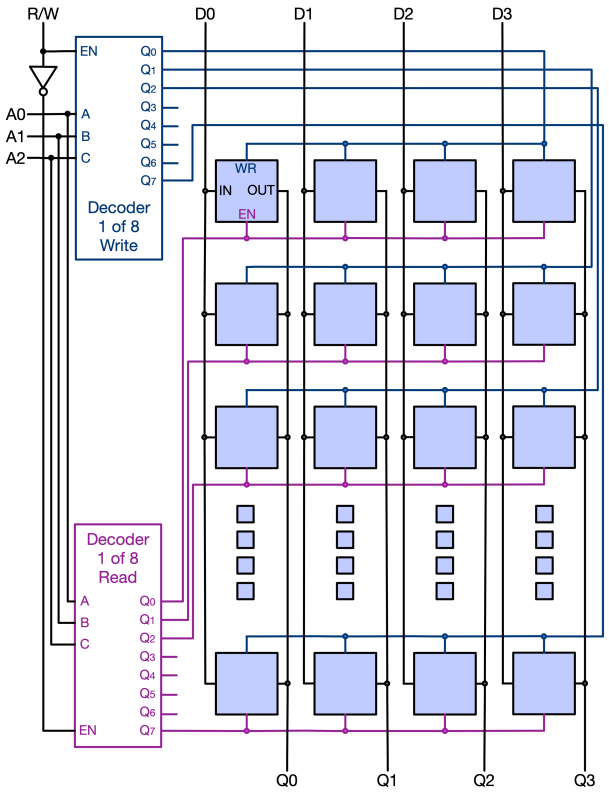

Bild 68 – Prinzipielle Schaltung eines 32-Bit-SRAM

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:

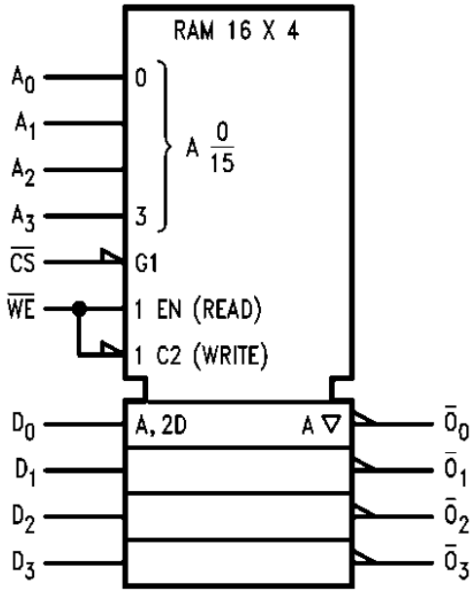

Bild 69 – Symbol des 64-Bit-RAM 74LS189

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:

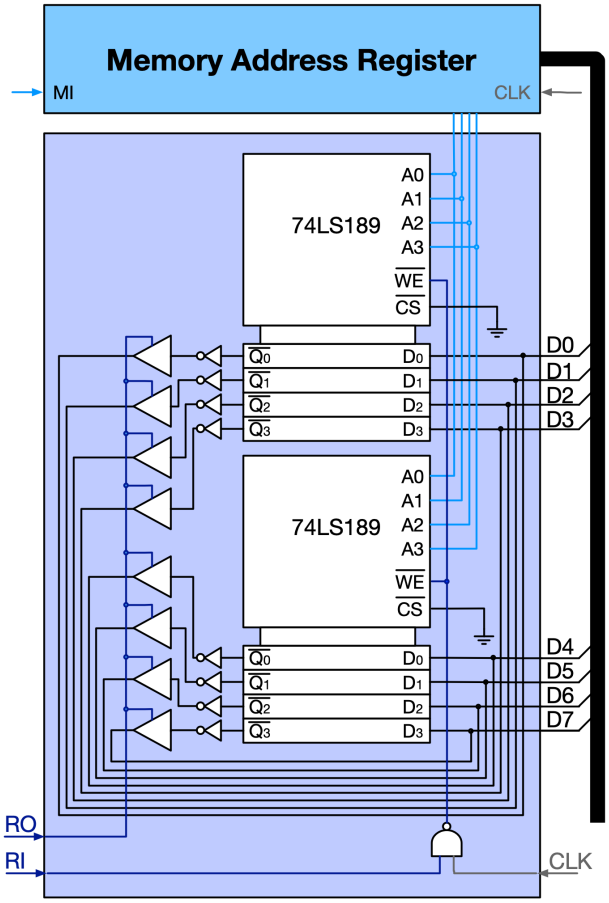

Bild 70 – Schaltung des RAM-Moduls (vereinfacht)

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.

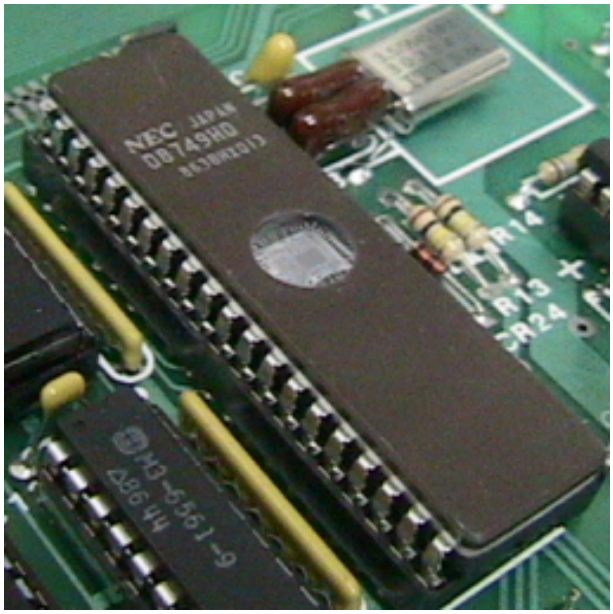

Bild 72 – EPROM mit Fenster fürs Löschen der Daten mit UV-Licht

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:

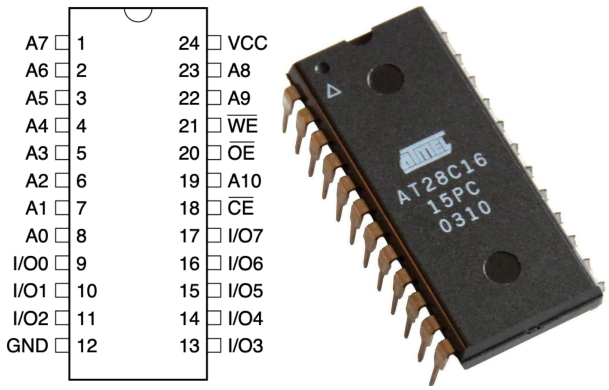

Bild 73 – Pin-Belegung/Bild des EEPROMs 28C16

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.