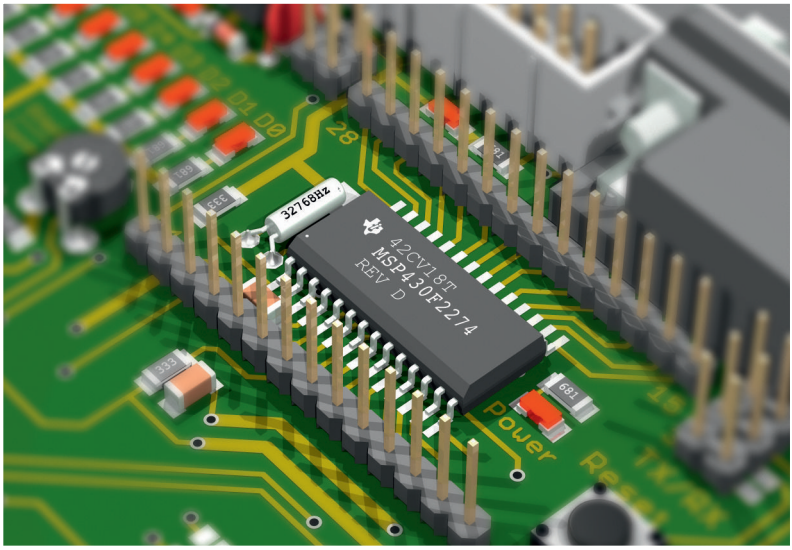
Matthias Sturm

Mikrocontroller-technik

Am Beispiel der MSP430-Familie



2., neu bearbeitete Auflage

HANSER

Matthias Sturm

Mikrocontrollertechnik

Am Beispiel der MSP430-Familie

2., neu bearbeitete Auflage

Mit 104 Bildern und 48 Tabellen

Fachbuchverlag Leipzig
im Carl Hanser Verlag

Prof. Dr.-Ing. Matthias Sturm
Hochschule für Technik, Wirtschaft und Kultur Leipzig

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN: 978-3-446-42231-5
E-Book-ISBN: 978-3-446-42964-2

Lektorat: Franziska Jacob, M.A.
Herstellung: Dipl.-Ing. (FH) Franziska Kaufmann
Satz: le-tex publishing services GmbH, Leipzig
Coverconcept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisierung: Stephan Rönigk
Druck und Bindung: Pustet, Regensburg
Printed in Germany

Vorwort

Für Christine, Martin, Bianca, Micha und Max

Dieses Buch ist ein Lehrbuch. Oder sollte ich besser sagen, ein Lernbuch. Es beschreibt sehr ausführlich die Funktion und die Handhabung von Mikrocontrollern am Beispiel des MSP430F2274 von Texas Instruments. Er ist ein Vertreter dieser kleinen, programmierbaren Bausteine, die uns fast unbemerkt täglich umgeben. Im Haushalt, im Auto, beim Telefonieren, Computern, einfach überall. Für alle, die gern verstehen möchten, wie diese Mikrocontroller funktionieren, wie man sie programmiert und einsetzt, ist dieses Buch geschrieben.

In unserer schnelllebigen Zeit muss man sich notwendiges Wissen sehr effizient aneignen. Dabei helfen Bücher, die Sachverhalte so beschreiben, dass keine Fragen offen bleiben.

Ein solches Buch halten Sie gerade in Ihren Händen.

Im Gegensatz zu manch anderem Fachbuch zu dieser Thematik wird in diesem sehr anschaulich und umfassend beschrieben, wie Mikrorechner aufgebaut sind und wie man sie handhabt. Ohne sich in Details zu verlieren, wird Schritt für Schritt erklärt, wie alle Rechnerkomponenten arbeiten und miteinander kommunizieren.

Ich sage öfter in Bezug zu den Controllern: „Kennt man einen, kennt man alle." Gemeint ist damit, dass in allen Mikrorechnern wiederkehrende Baugruppen und Methoden anzutreffen sind, so dass es keinen großen Unterschied macht, welcher konkrete Baustein beispielhaft beschrieben wird. Deshalb sind die Erläuterungen in diesem Buch allgemeingültig.

Mit Absicht habe ich jedoch den MSP430F2274 für dieses Buch ausgewählt. Er ist ein moderner, leistungsfähiger Mikrocontroller, der zahlreiche Eigenschaften moderner Rechnerbausteine in sich vereinigt. Außerdem ist er aufgrund seiner klaren Struktur hervorragend als Einstieg in die Welt programmierbarer Bausteine geeignet.

Der Buchinhalt ist didaktisch aufgebaut. Ausgehend von leicht verständlichen Grundschaltungen werden der Aufbau und die Funktion von Mikrocontrollern erklärt und beschrieben. Der Leser erlernt die Programmierung von Mikrocontrollern in Assemblersprache und anhand von Programmbeispielen in der Hochsprache C. Die kostenfreien Softwareentwicklungswerkzeuge auf der Website der Firmen IAR und Texas Instruments erlauben das Entwickeln und Testen eigener Programme, ohne dass man einen „echten" Mikrocontroller benötigt. Es gibt jedoch zahlreiche Entwicklungsboards, die mit der Entwicklungsumgebung zusammenarbeiten, so dass auch dem Experimentieren mit echten Mikrocontrollern nichts im Wege steht. Im Buch ist eine speziell für Ausbildungszwecke entwickelte Hardwareplattform näher beschrieben, mit deren Hilfe viele Aspekte der Mikrocontrolleranwendungen praktisch erlebbar werden.

Das erste Kapitel beschreibt ausgehend von logischen Schaltungen den Aufbau und wichtige Grundfunktionen eines Rechnerkerns. Dabei werden Grundbegriffe der Mikrorechentechnik eingeführt.

Im zweiten Kapitel werden Zahlensysteme und ihre Implementierung in Mikrorechnern und die Bedeutung von Statusbits beschrieben sowie unterschiedliche Rechnerarchitekturen aufgezeigt.

Das dritte Kapitel beschreibt den MSP430 als einen Vertreter der Mikrocontroller im Überblick. Vertieft dargestellt werden die Struktur der Zentralen Verarbeitungseinheit (CPU) sowie der Befehlssatz und verfügbare Adressierungsarten.

Das Kapitel 4 widmet sich der Softwareentwicklung, dem Aufbau von Programmen und der Bedienung von Entwicklungswerkzeugen. Hier hat der Leser erstmals Gelegenheit, erlerntes Wissen anhand eigener Projekte praktisch auszuprobieren. Durch praktische Anwendungen, die von nun an die weiteren Kapitel begleiten, wird das Lernen unterstützt. Der Leser wird in diesem Kapitel auch mit Softwaretechniken wie der Unterprogrammtechnik und der Interruptprogrammierung vertraut gemacht.

Die vorgestellten Beispielprogramme sowie weiter reichende Informationen findet man auf der Website zum Buch: *www.msp430-buch.de*

Im Kapitel 5 erfolgt die Beschreibung von Funktionseinheiten des Mikrocontrollers. Es ist das umfangreichste im Buch. Die ausführliche Darstellung und die vorgestellten Softwarebeispiele unterstützen nachhaltig das Verständnis der vermittelten Inhalte und regen zum eigenen Experimentieren an.

Das sechste Kapitel präsentiert die zuvor in Assemblersprache erklärten Beispiele vorangegangener Kapitel in der Hochsprache C. Auch hier werden die einzelnen Softwarelösungen genau erklärt, wobei die Programmfunktionen identisch denen in Assemblersprache sind. Durch die Beschreibung der gleichen Funktion in zwei unterschiedlichen Beschreibungssprachen ist der Weg zu Verständnis höheren Programmiersprachen geebnet.

In den Anlagen befinden sich Beschreibungen zur Softwareinstallation sowie Erläuterungen zum Education Board. Außerdem sind häufig benötigte Übersichten zum Befehlssatz, die Interruptvektortabelle sowie eine ausführliche Beschreibung interner Register angefügt. Dieses Material wird auch der erfahrene Entwickler schätzen.

Bis zum Kapitel 3 bietet das Buch anschauliche und ausführliche Beschreibungen, also Lesestoff. Ab dem Kapitel 4 sollte man parallel zum Lesen auch die praktischen Beispiele ausprobieren, modifizieren und mit eigenen Softwarelösungen experimentieren.

Mit diesem Buch hoffe ich, zukünftigen Studenten und Technikinteressierten eine wertvolle Hilfe bei der Einarbeitung in das Thema Mikrocontrollertechnik in die Hand gegeben zu haben.

Danksagung

Die meisten Bücher haben viele Väter. Dies gilt besonders bei Fach- und Lehrbüchern. Auch wenn nur der Name eines Autors auf dem Umschlag steht, haben doch viele fleißige Helfer mitgearbeitet, um es in die endgültige Fassung zu bringen.

Das ist bei diesem Buch nicht anders gewesen. Ich möchte Ihnen deshalb hier einige Personen vorstellen und nennen, die mich maßgeblich bei der Erarbeitung unterstützt haben. Da sind Lutz Birl und Horst Diewald, die Väter der MSP430 Mikrocontrollerfamilie. Sie haben wesentliche Teile des MSP430 Mikrocontrollers entwickelt und ihm zahlreichen herausragenden Eigenschaften verliehen. Dadurch ist er vielseitig einsetzbar und besonders geeignet für den

mobilen Betrieb. Vielen Dank für die Erlaubnis, zahlreiche Bilder aus den Originaldokumenten verwenden zu dürfen.

Meine ehemaligen Studenten, die inzwischen erfolgreich in international agierenden Unternehmen tätig sind, sowie meine derzeitigen Studenten haben durch Beiträge in Graduierungsarbeiten Teile dieses Buches erst möglich gemacht. Ich danke Michael Junghans und Andreas Dannenberg, Mirko Fuchs und Dr. Maik Müller sowie Wolfgang Lutsch und Tobias Wengemuth, die besonders engagiert an der Hard- und Softwareentwicklung des Education Boards für die erste Auflage des Buches gewirkt haben. Für die nun vorliegende zweite Ausgabe gab es zahlreiche Neuerungen, denn die Entwicklung bleibt nicht stehen. Der nun auf dem Education Board integrierte MSP430F2274 besitzt im Vergleich zum Vorgängermodel deutlich leistungsfähigere Funktionseinheiten, verfügt über mehr Speicher und eine neue Programmierschnittstelle. Das neue Board wurde von Herrn Michael Eiserbeck entwickelt, dem ich herzlich für sein engagiertes Arbeiten und zahlreiche Ideen danken möchte. Für die ausgezeichnete Zusammenarbeit mit dem Forschungs- und Transferzentrum (FTZ) der HTWK Leipzig danke ich Dirk Lippik und Daniel Käßler. Ohne die kompetente Unterstützung der Mitarbeiter des FTZ wäre ein ganzheitliches Konzept, bestehend aus Lehrbuch, Experimentalplatine, Aufgaben und Lösungsbeispielen nicht realisierbar gewesen.

Viel Dank gebührt Mirja Werner, Franziska Kaufmann und Franziska Jacob vom Hanser Verlag, die Verständnis für meine zahlreichen anderen Aufgaben aufbrachten und mich dennoch sehr zielstrebig zum Abschluss des Buches führten.

Matthias Sturm, Markkleeberg, Februar 2014

Inhalt

1 Ein-Bit-Rechner

In diesem Kapitel wollen wir den Aufbau und die Funktion eines einfachen Ein-Bit-Rechners kennen lernen. Zugegeben, dieses System ist ziemlich klein und nicht besonders leistungsfähig. Aber es beinhaltet alle notwendigen Funktionsgruppen und ist bei voller Funktionalität sehr überschaubar. Wichtig ist es, die beschriebenen Funktionsbausteine und deren Bezeichnung zu verinnerlichen. Sie kehren bei allen Prozessoren und Controllern wieder.

Unser Ein-Bit-Rechner besteht aus Logikbausteinen. Im Rechnerkern, oft auch als **Core** bezeichnet, werden UND- sowie Exclusive-OR-Gatter, Multiplexer und Demultiplexer sowie D-Flipflops eingesetzt. Später kommen noch ein Speicherbaustein sowie ein Binärzähler hinzu. Diese sieben Bausteintypen, deren Zusammenschaltung und Zusammenwirken nachfolgend erläutert wird, bilden den Kern des Ein-Bit-Rechners. Die Funktionen der einzelnen Bausteine erschließen sich in den nachfolgenden Erläuterungen.

1.1 Rechenwerk

Wie eingangs erwähnt, wollen wir eine Rechenmaschine entwerfen, die in der Lage ist, unterschiedliche Rechenoperationen nacheinander auszuführen. Wir brauchen also ein Rechenwerk. Es soll später Bestandteil einer zentralen Verarbeitungseinheit sein. Wie aber kann man Hardware so verschalten, dass sie zu unterschiedlichen Zeiten unterschiedliche Aufgaben erfüllt?

Ein Vorschlag ist im Bild 1.1 dargestellt.

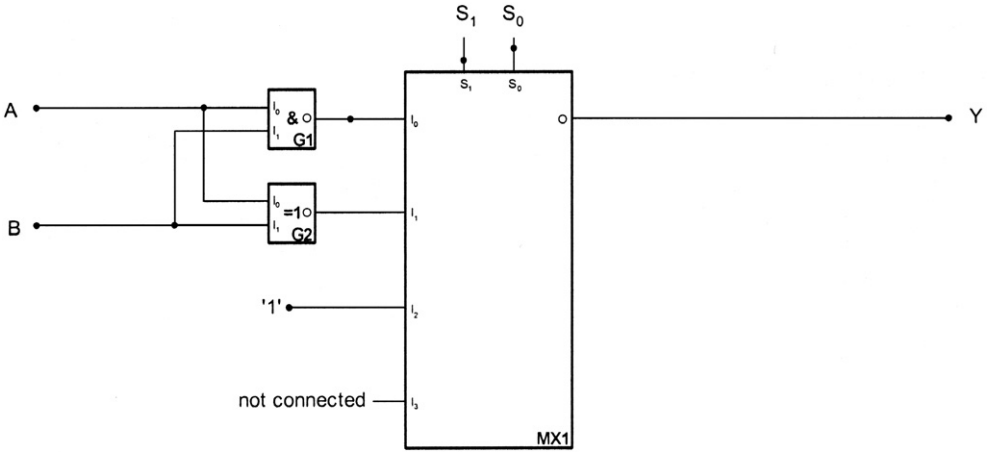


Bild 1.1 Das Rechenwerk des Ein-Bit-Rechners

Zwei Eingänge A und B sind jeweils auf zwei Logikgatter geführt. Am Ausgang des UND-Gatters (G1) kann das Ergebnis der logischen UND-Verknüpfung abgegriffen werden. Entsprechend steht am Ausgang des Exclusive-OR-Gatters (G2) das Ergebnis der XOR-Verknüpfung zur Verfügung. Es schließt sich ein Multiplexer (MX1) an. Er verfügt über vier Eingänge (I_0 bis I_3) und einen Ausgang (O). Welcher der Eingänge mit dem Ausgang verbunden wird, legen die Logikpegel der beiden Steuerleitungen S_0 und S_1 fest. Die Wahrheitstabelle 1.1 zeigt das Schaltverhalten des Multiplexers.

Tabelle 1.1 Wahrheitstabelle des Multiplexers

Steuereingang S_1	Steuereingang S_0	Ausgangszustand identisch mit
0	0	Eingang I_0
0	1	Eingang I_1
1	0	Eingang I_2
1	1	Eingang I_3

Es besteht im Bild 1.1 somit die Möglichkeit, am Ausgang Y des Multiplexers (MX1) entweder das Ergebnis der UND-Verknüpfung, der XOR-Verknüpfung oder den Logikpegel ‚1' abzugreifen. Dazu ist es nur notwendig, zum richtigen Zeitpunkt die Steuerleitungen S_0 und S_1 in den erforderlichen Zustand zu versetzen. Die Kombination der Schaltzustände der Steuersignale könnte man auch als **Befehle** für die Schaltung bezeichnen.

Und damit sind wir bei dem Dilemma der Schaltung in Bild 1.1. Sobald sich die Pegel der Steuerleitungen S_0 und/oder S_1 ändern, kann sich auch der Ausgangszustand des Multiplexers (MX1) verändern. Auch wirken Veränderungen der Eingangspegel *A* und *B* unmittelbar auf den Ausgang Y des Rechenwerkes. Das ist schlecht.

1.1.1 Register und Takt

Abhilfe schafft hier ein D-Flipflop (FF1), das in den Ausgang des Rechenwerkes geschaltet wird. Da die durch das Flipflop verfügbare Speicherzelle dicht am Rechenwerk platziert ist, ja sogar Bestandteil von ihm ist, wollen wir sie zukünftig als ein **Register** bezeichnen.

Immer wenn am Takteingang C (clock) des Flipflops (FF1) eine steigende Flanke, also ein Wechsel von logischem 0- auf logischen 1-Pegel, registriert wird, übernimmt es den logischen Zustand vom Eingang D, stellt ihn an den Ausgang O, und damit am Ausgang des Rechenwerkes Y bereit, und hält ihn konstant am Ausgang O, bis mit der nächsten steigenden Flanke am Eingang C der Zustand des FF1-Ausgangs aktualisiert wird. Bild 1.2 zeigt den ergänzten Schaltungsaufbau.

Um also ständig eine Aktualisierung am Ausgang des Rechenwerkes zu erhalten, bedarf es eines zyklischen Vorganges – das Flipflop braucht einen **Takt**. Takte werden zur Synchronisation der Abläufe in einem Rechner eingesetzt. Unser System ist derzeit noch zu klein, um dies anhand der Schaltung auch zu erkennen.

Dennoch wollen wir für unseren Aufbau eine Konvention vereinbaren:

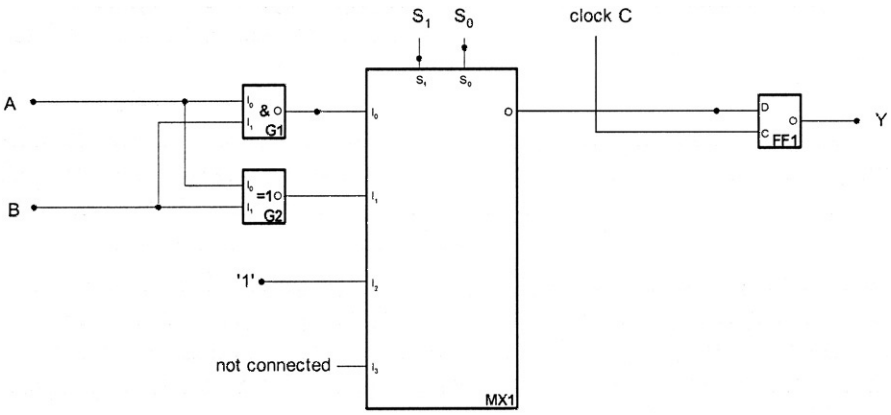


Bild 1.2 Ein Flipflop wird Bestandteil des Rechenwerkes

Nur während der Takt logischen 1-Pegel besitzt, dürfen sich die Zustände der Eingänge A und B ändern. Sobald der Takt logisch 0 ist, müssen diese stabil sein, da jetzt die Auswahl der geforderten Funktion durch die entsprechende Beschaltung der Steuerleitungen S_0 und S_1 erfolgt. Mit der steigenden Taktflanke wird das Ergebnis der ausgewählten Funktion in das Flipflop (FF1) übertragen und steht gleichsam am Ausgang Y des Rechenwerkes bereit.

Die Taktfolgefrequenz kann dabei sehr hoch sein und mehrere Megahertz betragen, schließlich handelt es sich bei unseren Komponenten um elektronische Bauelemente. Wir haben somit ein recht schnelles Rechenwerk, das allerdings nicht allzu viel leisten kann: UND A mit B, EXOR A mit B sowie Setzen des Ausgangs Y auf logisch 1.

1.1.2 Zwischenspeicher

Mit einer weiteren Hardwareergänzung gelingt es, die Schaltung so zu ergänzen, dass das Berechnen von Klammerausdrücken möglich wird. ((A UND B) EXOR A) könnte ein solcher Klammerausdruck sein. Dazu werden drei weitere Bauelemente in die Schaltung eingefügt.

Ein weiteres Flipflop (FF2) dient als Zwischenspeicher, das wollen wir ebenfalls als Register (nennen wir es Register 2) bezeichnen. Über einen Demultiplexer (DC1), der den Takt entweder auf das Ausgangsflipflop (FF1) oder das Zwischenspeicherflipflop (FF2) leitet, erfolgt die Zielzuweisung des Rechenergebnisses aus dem Multiplexer (MX1). Die Steuerung dieses Vorganges erfolgt über die Demultiplexer-Steuerleitung S_2. Wir halten einmal fest, dass Rechenergebnisse an unterschiedlichen Orten abgelegt werden können. Man nennt diese Orte im Zusammenhang mit der Befehlsabarbeitung auch **Operanden**. In unserem Fall könnte man folglich von einem Zieloperanden sprechen.

Doch auch im Eingangsteil unserer Schaltung ist ein weiteres Bauelement hinzugekommen – ein Multiplexer (MX2). Mit dessen Hilfe kann man zwischen den Datenquellen B und dem Inhalt des Zwischenspeichers (FF2), oder anders gesagt: des Registers 2, auswählen. Dazu dient die Steuerleitung S_3.

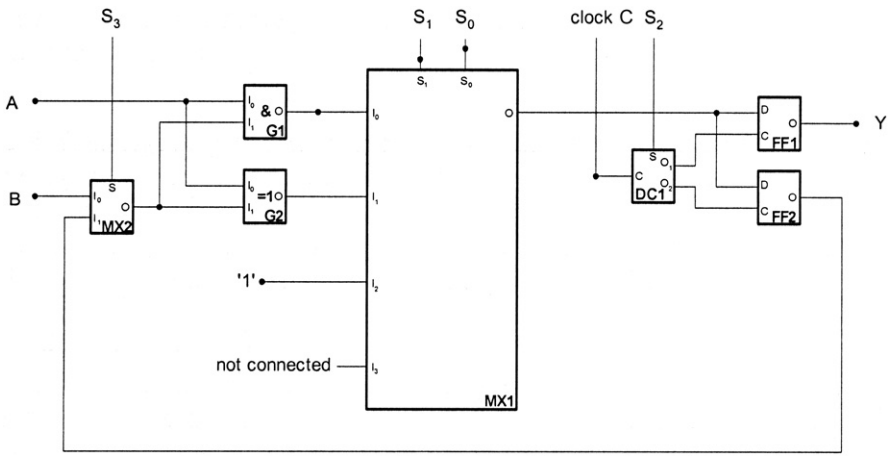


Bild 1.3 Rechenwerk mit Zwischenspeichern

Somit stehen nun neben dem Takt vier Steuerleitungen zur Verfügung. Zwei davon (S_0 und S_1) selektieren die Funktion des Rechenwerkes, und zwei weitere (S_2 und S_3) selektieren Quell- und Zieloperanden für die gewählte Funktion. Wir wollen in Zukunft das Auswählen von Datenquellen und Datenzielen **adressieren** nennen. Aus den verschiedenen Kombinationen der Steuerleitungszustände lassen sich nun unterschiedliche Befehle konstruieren. Da nach wie vor ein Eingang des Funktionsmultiplexers nicht beschaltet ist, lassen sich derzeit zehn sinnvolle Befehle generieren.

1.1.3 Native und emulierte Datenmanipulationsbefehle

Nativ nennt man jene Befehle, die direkt aus den Systemfunktionen heraus generiert werden können. Wenn man sich die Schaltung des Rechenwerkes genau ansieht, so können durch die Kombination zweier, aufeinander folgender, nativer Befehle weitere Befehle gebildet werden. Dabei leistet die Konstantenfunktion hilfreiche Dienste. Dazu zwei Beispiele:

1. Setzt man den Zwischenspeicher auf logisch 1 und vollzieht anschließend eine UND-Verknüpfung von A mit dem Zwischenspeicherinhalt, so ergibt sich der Zustand A am Ausgang.
2. Setzt man den Zwischenspeicher auf logisch 1 und vollzieht anschließend eine EXOR-Verknüpfung von A mit dem Zwischenspeicher, so ergibt sich der invertierte Zustand von A am Ausgang. Es steht somit eine ‚Komplement-A'-Funktion zur Verfügung.

Derartig generierte Befehle nennt man auch **emulierte Befehle**, da sie nicht unmittelbarer Bestandteil des Funktionsumfangs des Rechenwerkes sind, sich aber aus diesem ableiten lassen.

Doch was wird nun aus dem bisher unbeschalteten Multiplexereingang I_3 des MX1? Die Lösung findet sich im Bild 1.4.

Ein weiteres Flipflop (FF3) sowie ein 3-Eingangs-UND-Gatter (G3) sind hinzugekommen.

Die zusätzliche Schaltung bewirkt, dass immer dann, wenn eine EXOR-Funktion ausgeführt wird, das Ergebnis der UND-Verknüpfung im neuen Flipflop (FF3) gespeichert wird. Bisher

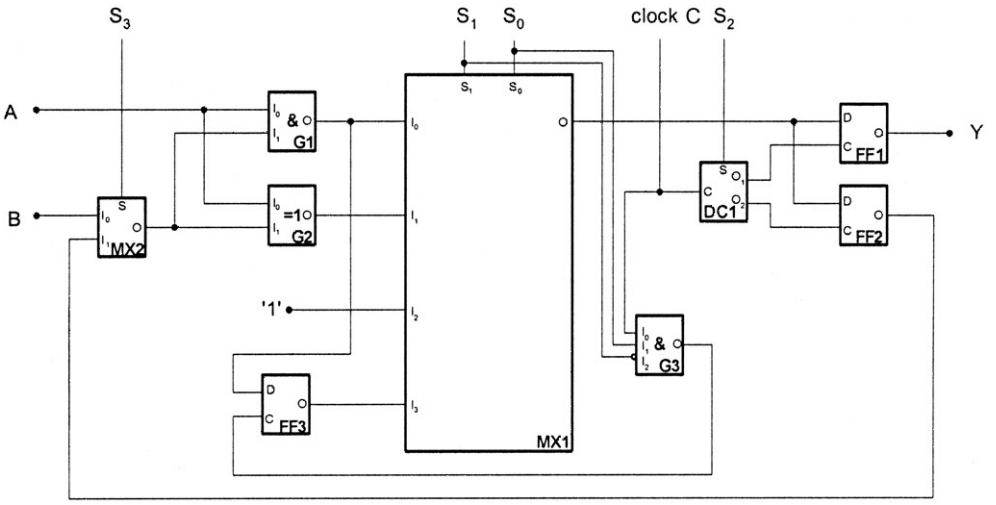


Bild 1.4 Schaltungsergänzung für ein Statusbit

Tabelle 1.2 Wahrheitstabellen der UND- sowie der XOR-Verknüpfung

Eingang	Eingang	Ergebnis
I_1	I_0	AND
0	0	0
0	1	0
1	0	0
1	1	1

Eingang	Eingang	Ergebnis
I_1	I_0	XOR
0	0	0
0	1	1
1	0	1
1	1	0

wurde das Speichern in Flipflops stets durch die Auswahl der Steuerleitung S_2 des Demultiplexers (DC1) geregelt. Dieses Speichern im Flipflop (FF3) erfolgt selbstständig durch die Schaltung und unabhängig vom Zustand der Steuerleitungen S_2. Um den Sinn dieser Schaltungserweiterung zu verstehen, schauen wir uns die Wahrheitstabellen der XOR- und der UND-Funktion nochmals an.

Die EXOR-Funktion stellt eigentlich eine Ein-Bit-Addition dar. Bei der Addition von 1 + 1 ergibt sich 0 und ein Übertrag 1. Bei der UND-Verknüpfung ist gerade das Ergebnis von 1 UND 1 gleich 1. Folglich wird im Flipflop (FF3) der Übertrag bei der Addition gespeichert. Das Flipflop (FF3) zeigt somit einen inneren Zustand des Rechenwerkes, nämlich den Übertrag bei einer Addition an. Ein Flipflop, das interne Zustände des Rechenwerkes speichert und trägt, wird als **Statusregister**, sein Inhalt als **Zustandsbit** oder auch als **Flag** (engl., Flagge) bezeichnet. Es gibt in komplexeren Rechenwerken noch eine ganze Reihe weiterer zustandsspeichernder Flipflops, die zu einem Statusregister zusammengefasst sind und einzelne Statusbits bzw. Flags beinhalten. Doch dazu später. Festzuhalten bleibt:

> Flags werden im Ergebnis von logischen oder arithmetischen Operationen des Rechenwerkes beeinflusst. Flags können (aber müssen nicht) zur bedingten Programmverzweigung eingesetzt werden.

Auch dies werden wir erst später besprechen, da unsere kleine Hardware diese Arbeit noch nicht leisten kann.

Wir wollen an dieser Stelle zusammenfassen, was wir im Abschnitt ‚Rechenwerk' kennen gelernt haben. Der prinzipielle Aufbau eines Rechenwerkes sollte uns vertraut sein. Alle Prozesse sind taktgetrieben und werden über die Steuerleitungen realisiert. Auch mit den Bezeichnungen Register als Zwischen- oder Ergebnisspeicher sowie Flag als Zustandsanzeiger dürfte es keine Schwierigkeiten geben. Auch sollte klar sein, was ein Operand ist und welchen Vorgang man als Adressierung bezeichnet.

Wenn wir uns von der Schaltung im Bild 1.4 wegbewegen, so bleiben vom Datenpfad – das ist der Weg, den die Daten bei ihrer Bearbeitung durchlaufen, zwei Eingänge A und B sowie ein Ausgang Y übrig. Die Steuerung des Rechenwerkes erfolgt taktsynchron über die Steuerleitungen S_0 bis S_3.

1.2 Steuerwerk

Nun wollen wir sehen, wie das Rechenwerk in eine komplexere Schaltung eingebettet ist. Dazu dient die Darstellung in Bild 1.5.

Stark vereinfacht ist das Rechenwerk dargestellt. Darüber befindet sich der Befehlsdecoder (= Steuerwerk), dessen Aufgabe nachfolgend erläutert wird.

Es gibt in Rechnersystemen Befehle, die der Datenmanipulation dienen. Einen Teil der Befehle haben wir bereits kennen gelernt.

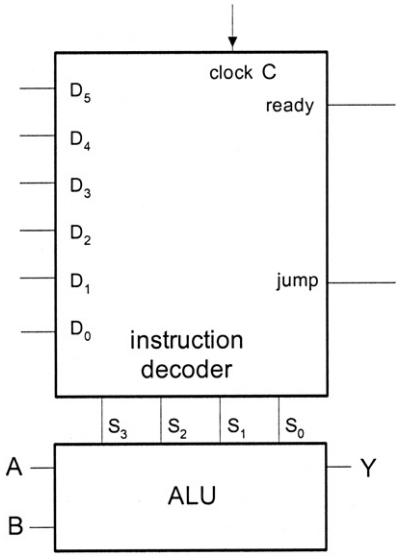


Bild 1.5 Rechenwerk und Befehlsdecoder

1.2.1 Programmsteuerbefehle

Mitunter ist es notwendig, die Reihenfolge bei der Abarbeitung der einzelnen Befehle während des Programmlaufes zu verändern. Man nennt dies: ‚im Programm verzweigen'. Solche **Programmverzweigungen** können ohne eine Bedingung, also *unbedingt* erfolgen. Oder aber sie werden von Zustandsbits des Rechenwerkes abhängig gemacht, dann nennt man die Verzweigungen *bedingt*. Unser Rechenwerk besitzt nur ein Zustandsbit, das Übertragsbit: der Zustand des FF3.

Die Schaltung in Bild 1.6, die einen wesentlichen Teil des Befehlsdecoders darstellt, besitzt Eingangs- und Ausgangssignalleitungen. Mit den drei Eingangssignalleitungen D_4, D_5 und flag werden die Funktion des Befehlsdecoders und die Einbeziehung des Rechenwerkes gesteuert. Signalisiert die Steuerleitung D_4, dass das Rechenwerk für die Befehlsausführung notwendig ist (1-Pegel), so legen die Zustände der Eingänge D_0 bis D_3 (die direkt mit den Steuerleitungen S_0 bis S_3 des Rechenwerkes verbunden sind) fest, welche Funktion das Rechenwerk ausführen soll (vgl. Bild 1.5).

Besitzt die Steuerleitung D_4 jedoch einen 0-Pegel, so wird dem Rechenwerk der Takt entzogen und in Abhängigkeit der beiden anderen Steuerleitungen D_5 und flag ein Sprungbefehl ausgeführt. In diesem Fall bekommen die Steuerleitungen D_0 bis D_3 eine andere Bedeutung zugewiesen, da ihr Schaltzustand ohne Takt im Rechenwerk keinerlei Aktion bewirkt. Sie werden verwendet, um das Sprungziel anzuzeigen. Doch zurück zur Befehlsdecoderschaltung im Bild 1.6.

Der Pegel der Signalleitung D_4 entscheidet darüber, ob eine Datenmanipulation im Rechenwerk stattfindet (D_4 = ‚1') oder ob eine Programmverzweigung durchgeführt werden soll (D_4 = ‚0').

Ist eine Verzweigung im Programmlauf vorgesehen, so entscheidet der Zustand der Signalleitung D_5 darüber, ob eine unbedingte Verzweigung stattfinden soll (D_5 = ‚0') oder ob der Wert des Zustandsbits im Rechenwerk über die Ausführung der Verzweigung entscheidet (Verzweigung bei Zustandsbit = ‚1').

Die Schaltung des Befehlsdecoders generiert auch ein Signal ‚ready', das die vollständige Befehlsabarbeitung signalisieren soll. Dies gelingt jedoch nur auf Grund der vorhin getroffenen

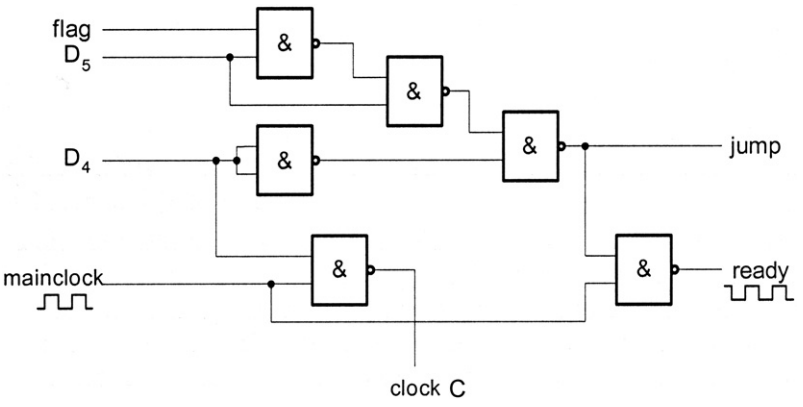


Bild 1.6 Schaltungsbeispiel für das Steuerwerk des Ein-Bit-Rechners

Tabelle 1.3 Beschreibung der Signalleitungen aus Bild 1.6

$D_4 = 0$ $D_4 = 1$	Eingangssignal	Sprungbefehl (bedingt oder unbedingt) Arithmetik- oder Logikbefehl
$D_5 = 0$ $D_5 = 1$	Eingangssignal	unbedingter Sprung bedingter Sprung
flag = 0 flag = 1	Eingangssignal	bedingten Sprung nicht ausführen bedingten Sprung ausführen
jump	Ausgangssignal	Signal zum Springen an eine andere Programmstelle
ready	Ausgangssignal	Takt zum Ausführen des nächsten Befehls
clock	Ausgangssignal	Takt zur Befehlsausführung im Rechenwerk

Konvention. Zur Erinnerung: Sie besagt, dass mit der steigenden Flanke des Taktsignals ‚clock' die Funktion im Rechenwerk ausgeführt wird, folglich der Befehl abgearbeitet wurde.

Doch wenden wir uns zunächst den zur Steuerung unseres System zur Verfügung stehenden Signalleitungen zu. Man kann sie nebeneinander aufschreiben:

$$D_5 \; ; \; D_4 \; ; \; D_3 = S_3 \; ; \; D_2 = S_2 \; ; \; D_1 = S_1 \; ; \; D_0 = S_0.$$

Theoretisch ergeben sich aus der Kombination der Signalpegel der sechs Steuerleitungen $2^6 = 64$ unterschiedliche Bitmuster. Jedoch sind nicht alle generierbaren Bitmuster bei der Anwendung auf unsere Schaltung sinnvoll. So ist zum Beispiel der Zustand der D_5-Signalleitung bei Befehlen im Rechenwerk ohne Auswirkungen, wodurch sich die Anzahl verfügbarer Bitmuster auf 48 reduziert. Auch bewirken einige Bitmuster die gleichen Funktionen im Rechenwerk (Redundanz), so dass sich die Anzahl sinnvoll nutzbarer Bitmuster weiter reduziert.

1.2.2 Befehlsliste

Die Bitmuster der nativen Befehle sind in der Tabelle 1.4 aufgelistet. Die emulierten Befehle findet man in der Tabelle 1.5.

Wie man bereits aus der Schaltung des Rechenwerkes entnehmen konnte, sind die Steuerleitungen S_0 und S_1 für die Auswahl der Funktion des Rechenwerkes verantwortlich. Mit der Steuerleitung S_2 wird der Zieloperand ausgewählt. S_3 entscheidet darüber, ob Eingang B oder der Inhalt des Zwischenspeichers FF2 als Quelloperand genutzt wird.

D_4 entscheidet über die Bedeutung der Bits 0 bis 3 als Rechenwerkssteuerleitungen oder Adressoperand bei Verzweigungsbefehlen. Die Signalleitung D_5 schließlich dient der Selektion von bedingten oder unbedingten Verzweigungen.

Damit ist der linke Teil der Tabellen hinreichend beschrieben. Mit dem Begriff Operationscode (Opcode) ist nichts anderes gemeint als die hexadezimale Darstellung der Bitmuster, die dem Befehlsdecoder zugeführt werden. Fügt man gedanklich zwei Nullen für die Bitpositionen 6 und 7 links in die Tabellen ein, so kann man den Operationscode leicht selbst aus den Bitzuständen bilden.

Tabelle 1.4 Liste der nativen Befehle des Ein-Bit-Rechners

D_5	D_4	D_3	D_2	D_1	D_0	Op-code	Funktion	Flag-ein-fluss
Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0			
0	0	0	0	0	0	00h	JMP 00h unbedingte Verzweigung auf Adresse 0	nein
0	0	0	0	0	1	01h	JMP 01h unbedingte Verzweigung auf Adresse 1	nein
	…						…	
0	0	1	1	1	1	0Fh	JMP 0Fh unbedingte Verzweigung auf Adresse 15	nein
1	0	0	0	0	0	10h	JC 00h bedingte Verzweigung auf Adresse 0	nein
1	0	0	0	0	1	11h	JC 01h bedingte Verzweigung auf Adresse 1	nein
	…						…	
1	0	1	1	1	1	1Fh	JC 0Fh bedingte Verzweigung auf Adresse 15	nein
0	1	0	0	0	0	20h	AND AB,Y	nein
0	1	0	0	0	1	21h	EXOR AB,Y	ja
0	1	0	0	1	0	22h	SET Y	nein
0	1	0	0	1	1	23h	MOVE C,Y	nein
0	1	0	1	0	0	24h	AND AB,Z	nein
0	1	0	1	0	1	25h	EXOR AB,Z	ja
0	1	0	1	1	0	26h	SET Z	nein
0	1	0	1	1	1	27h	MOVE C,Z	nein
0	1	1	0	0	0	28h	AND AZ,Y	nein
0	1	1	0	0	1	29h	EXOR AZ,Y	ja
0	1	1	0	1	0	2Ah	SET Y	nein
0	1	1	0	1	1	2Bh	MOVE C,Y	nein
0	1	1	1	0	0	2Ch	AND AZ,Z	nein
0	1	1	1	0	1	2Dh	EXOR AZ,Z	ja
0	1	1	1	1	0	2Eh	SET Z	nein
0	1	1	1	1	1	2Fh	MOVE C,Z	nein

Die Befehle in der Spalte ‚Funktion' sind wie folgt aufgeschrieben:

Befehl Quelloperand(en), Zieloperand.

Mitunter gibt es nur einen Zieloperanden und keinen Quelloperanden, so z. B. bei den Sprungbefehlen oder beim SET-Befehl.

Eine Randbemerkung zur Abarbeitungsgeschwindigkeit und zum erforderlichen Speicherplatz der Befehle unserer kleinen Schaltung sei hier eingefügt.

Tabelle 1.5 Liste der emulierten Befehle des Ein-Bit-Rechners

EQUAL A,Y								
1	0	0	1	1	0	26h	SET Z	nein
1	0	1	0	0	0	28h	AND AZ,Y	nein
EQUAL A,Z								
1	0	0	1	1	0	26h	SET Z	nein
1	0	1	1	0	0	2Ch	AND AZ,Z	nein
NOT A,Y								
1	0	0	1	1	0	26h	SET Z	nein
1	0	1	0	0	1	29h	EXOR AZ,Y	ja
NOT A,Z								
1	0	0	1	1	0	26h	SET Z	nein
1	0	1	1	0	1	2Dh	EXOR AZ,Z	ja

Alle nativen Befehle werden in einem Takt ausgeführt. Die emulierten Befehle, da sie aus zwei nativen Befehlen bestehen, benötigen in unserer Schaltung folglich zwei Takte. Diese Angabe ist jedoch für die eigentliche Funktion unserer Schaltung unerheblich. Wichtig wären diese Informationen, wenn man die Laufzeit eines Programms berechnen möchte.

In den Befehlslisten sind meist auch Hinweise auf den erforderlichen Speicherbedarf bei Verwendung eines bestimmten Befehls zu finden. Alle unsere nativen Befehle benötigen 6 Bit, die emulierten Befehle 12 Bit.

Bis jetzt sollte uns klar geworden sein, welche Aufgabe ein Steuerwerk zu erfüllen hat, dass es Datenmanipulations- und Programmsteuerbefehle gibt, letztere in bedingter und unbedingter Form. Wir haben kennen gelernt, wie Steuer- und Rechenwerk miteinander verbunden sind und wie das Rechenwerk durch den Flagzustand Einfluss auf die Programmsteuerung nehmen kann. Weiterhin müssten die Begriffe Operationscode, Quell- und Zieloperanden vertraut sein.

1.3 Programmspeicher

Nachdem wir nun genaue Kenntnis über das Rechen- und das Steuerwerk unseres Ein-Bit-Rechners gewonnen haben, wollen wir uns ansehen, woher und über welche Wege die Befehle zum Steuerwerk gelangen.

Soll ein Programm abgearbeitet werden, so ist klar, dass die Befehle einzeln nacheinander dem Steuerwerk zugeführt werden müssen.

> Die **Befehle** sind, genau genommen, Bitmuster, die in einem Halbleiterspeicher abgelegt sind.

Dieser Speicherbaustein besteht aus einzelnen Speicherzellen. Die Zellen sind in der Lage, parallel mehrere Bit aufzunehmen. Meist sind es 8 Bit, für unseren Ein-Bit-Rechner genügt aber

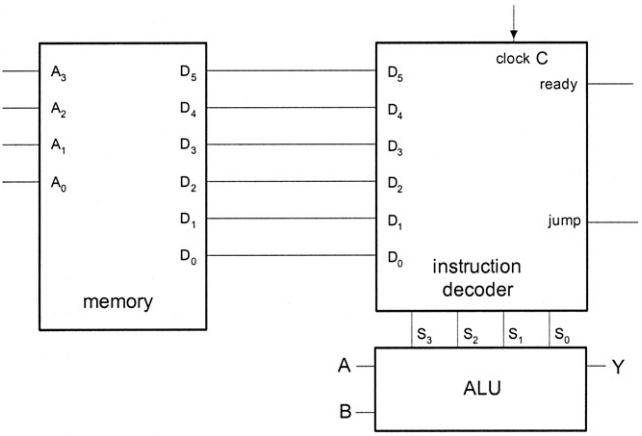


Bild 1.7 Speicheranschluss an das Steuerwerk

ein Speicherbaustein mit 6 Bit breiten Speicherzellen (den es nicht wirklich zu kaufen gibt). Unser Speicherchip soll 16 Zellen umfassen, also maximal 16 Befehle aufnehmen können. Es ist immer nur eine der 16 Zellen ansprechbar. Man erreicht jede einzelne Zelle über ihre Adresse. Genau genommen ist es die Zellennummer, die wir als Adresse bezeichnen wollen. Nun wäre es mühsam, für jede Speicherzelle eine separate Aktivierungsleitung vorzusehen, zumal wir ohnehin stets nur eine einzelne Zelle aktivieren wollen. Aus diesem Grund sind in jedem Speicherbaustein so genannte **Adressdecoder** implementiert. Diese Decoder selektieren aus einem binär angelegten Adressmuster (Adresswert oder einfach die **Adresse**) die entsprechende Speicherzelle und aktivieren diese. Bei 16 Speicherzellen ist alles noch überschaubar.

Wie die Befehlsbitmuster (also auch unser Programm), in den Speicher kommen, soll an dieser Stelle nicht ausgeführt werden. So viel sei aber gesagt: Es hängt von der Speichertechnologie ab, wie der Programmiervorgang erfolgen muss. ROM-, EPROM-, EEPROM- oder Flashspeicher verlangen jeweils unterschiedliche Verfahren.

Nehmen wir an, die einzelnen Befehle unseres Programms sind beginnend ab der nullten Speicherzelle (Adresse 0h) fortlaufend im Speicher abgelegt. Um diese Zelle zu aktivieren, muss an die Adressleitungen A_0 bis A_3 des Speichers das binäre Adressmuster 0000b angelegt werden. Daraufhin stellt die Speicherzelle 0h ihren Inhalt am Datenausgang (D_0 bis D_5) bereit. Diese Daten werden über Drähte zur Steuereinheit, dem Befehlsdecoder, geleitet.

Solche logisch zusammengehörigen Leitungsbündel, in unserem Fall sechs Befehlsinformationsleitungen, nennt man in der Rechentechnik allgemein einen **Bus**.

Da es in Rechnern viele solcher zusammengehöriger Leitungen gibt, werden die Busse noch genauer spezifiziert. So findet man Daten-, Steuer- oder auch Adressbusse. Sie können beliebig viele Einzelleitungen umfassen.

Ist unser erster Befehl vollständig abgearbeitet, was uns durch den ‚ready'-Ausgang des Befehlsdecoders mitgeteilt wird, so können wir den Inhalt der nächsten Speicherzelle durch Anlegen einer neuen Adresse aufrufen.

1.4 Befehlszähler

Zweckmäßig erscheint an dieser Stelle die Einbindung einer weiteren Einheit, die die Aufgabe des Weiterschaltens der Speicheradressen übernimmt. Und damit kommen wir zu Bild 1.8.

Als weitere und damit letzte Einheit wurde ein 4-Bit-Binärzähler in die Schaltung aufgenommen. Außer einer Zählfunktion hat der Baustein auch eine Ladefunktion. Über sie kann er auf jeden beliebigen Zählerstand zwischen 0h und 0Fh voreingestellt werden. Doch der Reihe nach.

Die Ausgänge des Zählers (O_0 bis O_3) sind direkt mit den Adresseingängen des Speichers (A_0 bis A_3) verbunden. Er generiert also die Adresse der Speicherzelle, die den jeweils abzuarbeitenden Befehl trägt. Aus diesem Grund wird dieser Zähler auch **Befehlszähler**, oder englisch **program counter**, genannt.

Die nachfolgend beschriebenen Abläufe sind ein Kreisprozess, d.h., es laufen immer wieder die gleichen Vorgänge ab. Beginnen wir beim Befehlszähler. Sein innerer Zählerstand sei 0h.

Dieser Wert wird als binäres Muster an seinen Ausgängen (O_0 bis O_3) bereitgestellt. Über den **Adressbus** (das sind die vier Signalleitungen zu den Adresseingängen, A_0 bis A_3, des Speichers) wird dem Speicher die zu aktivierende Speicherzelle angezeigt. Der Speicher wird das Adressmuster intern dekodieren und den Inhalt der ausgewählten Speicherzelle an den Speicherausgängen (D_0 bis D_5) verfügbar machen. Wir wissen aus den vorangegangenen Beschreibungen, dass das Ausgangsmuster des Speichers der Operationscode des abzuarbeitenden Befehls ist. Über den Befehlsdatenbus gelangt der Operationscode zum Steuerwerk. Für den angenommenen Fall, dass es sich um einen *Datenmanipulationsbefehl* handelt, wird das Rechenwerk entsprechend eingestellt und der Befehl ausgeführt. Dazu benötigt unsere kleine Schaltung einen Takt. Das Steuerwerk wird danach die Information der vollständigen Abarbeitung des Befehls über die entsprechende Signalausgangsleitung ‚ready' an den Eingang des Befehlszählers weiterleiten. Dieser wiederum erhöht seinen internen Zählerstand um einen Zählwert.

Damit kann man wieder an der obigen Eintrittsstelle fortfahren und alle 16 gespeicherten Befehle in einer Endlosschleife abarbeiten. Ist der 16. Zählerwert (Fh) erreicht, so kommt es beim

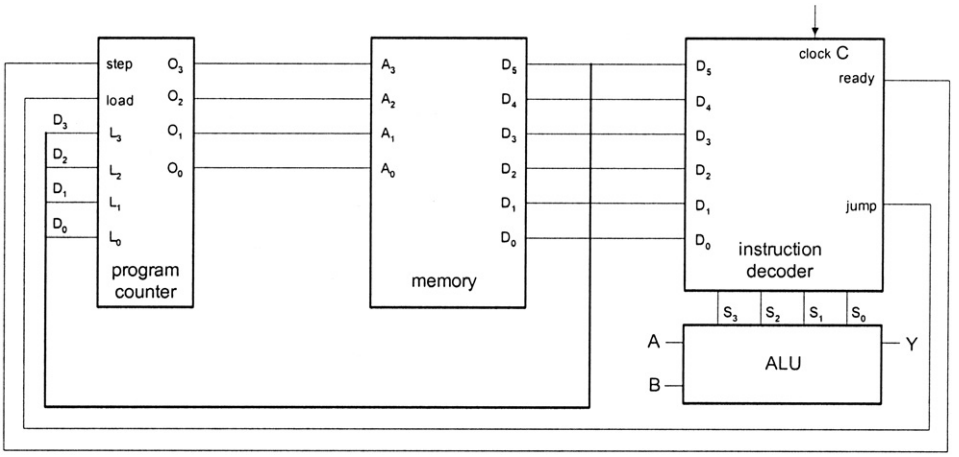


Bild 1.8 Der Ein-Bit-Rechner ist eine CPU (s. Abschnitt 1.5)

nächsten Takt zu einem *Befehlszählerüberlauf*, d. h., der Zähler, der ja nur Werte innerhalb eines geschlossenen Zählkreislaufes von 0h bis 0Fh annehmen kann, wird auf den internen Zählerwert null eingestellt, um dann wieder Takt für Takt alle 16 Zählerstände einzunehmen.

Anders verhält es sich bei einem *Sprungbefehl*. Im Bild 1.8 ist ersichtlich, dass die niederwertigen oder auch die unteren vier Signalleitungen (D_0 bis D_3) des Befehlsdatenbusses auch an die Ladeeingänge des Befehlszählers geführt sind. In Datenmanipulationsbefehlen tragen diese vier Signalleitungen die Information, welche Funktion mit welchen Datenquellen zu welchem Datenziel durchzuführen ist. Dazu dienten die Erläuterungen der vorangegangenen Abschnitte. Bei Sprungbefehlen tragen diese Signalleitungen aber die Adresse des nächsten abzuarbeitenden Befehls.

Die **Abläufe in unserem Ein-Bit-Rechner** sind folgende:
Wird der Operationscode eines *unbedingten Sprungbefehls* durch den Speicher ausgegeben, so erkennt der Befehlsdecoder diesen und aktiviert die Signalausgangsleitung für den Sprungbefehl (eine fallende Flanke an der Signalleitung ‚jump'). Die unteren vier Signalleitungen des Speichers signalisieren an den Ladeeingängen des Zählers (L_0 bis L_3) bereits, welcher Zählerwert oder auch welcher Adresswert als nächster in den Zähler geladen werden soll. Trifft also vom Steuerwerk das Signal zum Laden eines neuen Adresswertes beim Befehlszähler ein, so wird unmittelbar dieser Wert geladen und eine neue Adresse zum Speicher geschickt. Daraufhin wird die adressierte Zelle aktiviert und ihr Speicherinhalt ausgegeben.

Für den Fall, dass ein *bedingter Sprungbefehl* durch das Steuerwerk decodiert wird, entscheidet der Inhalt des Statusregisters (C-Flag), ob die Signalleitung (jump) zum Laden einer neuen Adresse aktiviert wird. Ist das C-Flag gelöscht (Inhalt 0), so erfolgt kein Sprung und die Befehlsabarbeitung setzt sich an der um 1 erhöhten Adresse fort.

Eine Randbemerkung zum Zeitverhalten sei hier eingeflochten. Analysiert man unsere kleine Schaltung, so wird man feststellen, dass die Datenmanipulationsbefehle genau einen Takt für ihre Abarbeitung benötigen. Die zeitliche Abfolge bei der Ausführung der Sprungbefehle hingegen basiert auf den Gatterlaufzeiten, also auf der Geschwindigkeit, mit der sich die Impulse durch die Schaltelemente bewegen. Dies ist nur in unserer Beispielschaltung so und der simplen, doch damit übersichtlichen Struktur geschuldet.

1.5 Zusammenfassung

Unsere kleine Schaltung ist eine programmierbare Funktionseinheit. Sie arbeitet durch einen Takt getrieben und ist in der Lage, eine begrenzte Anzahl von Befehlen abzuarbeiten. Es können dies Datenmanipulations- oder auch Sprungbefehle sein. Die eigentliche Datenverarbeitung findet im Rechenwerk statt, zwischen den Eingängen A und B sowie dem Ausgang Y. Das Rechenwerk der Schaltung kann Daten zwischenspeichern und Zustandsinformationen zur späteren Programmsteuerung speichern. All diese Eigenschaften sind Kennzeichen einer **zentralen Verarbeitungseinheit**, die meistens **CPU** (engl., **Central Processing Unit**) genannt wird. Die CPU ist der Kern eines jeden Rechnerbausteins, weshalb in der Literatur auch die Bezeichnung **Core** zu finden ist. In der Regel wird der Programmspeicher nicht als Bestandteil der Zentraleinheit betrachtet.

Die CPU ist nicht für den Datenaustausch mit der Umwelt zuständig. Periphere Subsysteme, die die Schaltung ergänzen, übernehmen diese Aufgabe.

Gemeinsam mit der Befehlsliste bildet die Darstellung verfügbarer CPU-Register das **Programmiermodell** des Rechnerkerns. Mit Hilfe dieses Programmiermodells können Anwender die CPU nutzen, ohne genaue Kenntnisse über die darunter liegende Schaltung zu besitzen.

In diesem Kapitel haben wir eine Fülle wichtiger Begriffe der Mikrorechentechnik sowie den Aufbau und die Funktion eines kleinen Ein-Bit-Rechners kennen gelernt. Zugegeben, mit einem Ein-Bit-Rechner kommt man nicht sehr weit. Stellt man sich aber vor, dass man den Funktionsumfang des Rechenwerkes durch einen größeren Multiplexer deutlich erweitert, dass man acht oder sechzehn Rechenwerke eines solchen Ein-Bit-Rechners parallel schaltet und dass man den Befehlszähler statt mit bisher 4 Bit auf 16 Bit vergrößert, so entsteht ein Rechnersystem, mit dem sich ganz gut arbeiten lässt. Natürlich fehlen noch Speicherbausteine und eine ganze Menge peripherer Systeme, um einen wirklich leistungsfähigen Rechner entstehen zu lassen. Die erläuterten Abläufe in ‚echten' Rechnersystemen sind aber annähernd die gleichen wie die unserer kleinen Beispielschaltung. Anspruchsvollere Schaltungskonzepte und viel ingenieurtechnisches Know-how lassen schließlich die modernen Mikrorechnerbausteine entstehen, mit denen wir uns in den nächsten Kapiteln befassen werden.

Festzuhalten bleibt, dass die eingeführten Begriffe für bestimmte Baugruppen und deren Funktion allgemein gültig sind und bei der Beschreibung großer und kleiner Rechner immer wieder auftauchen.

2 Mikrorechentechnik-Grundlagen

Gleich zu Beginn wollen wir einige grundlegende Begriffe nennen. 1 **Bit** ist die kleinste Darstellungseinheit binärer Daten und kann die Werte 0 oder 1 annehmen. 4 Bit werden als **Nibble**, 8 Bit als **Byte** und 16 Bit als **Word** bezeichnet.

Auch ist es wichtig zu wissen, dass in der Mikrorechentechnik bei Aufzählungen oder Nummerierungen meistens mit der Zahl 0, und nicht mit der Zahl 1 begonnen wird.

2.1 Codes

Codes sind Vorschriften zur eindeutigen Zuordnung von Zeichenmengen. Ziffern und Zahlen, aber auch Buchstaben und Sonderzeichen werden in Rechnern codiert verarbeitet. Die gebräuchlichsten Codes wollen wir jetzt kennen lernen.

2.1.1 ASCII-Code

Zur Darstellung von Text hat sich sehr früh der ASCII-Code (American Standard Code for Information Interchange) etabliert. Die Codetabelle enthält neben Buchstaben und Ziffern auch Sonder- und Steuerzeichen. Wann immer Buchstaben, Zahlen oder Sonderzeichen auf einem Display ausgegeben werden sollen, ist der ASCII-Zeichensatz zu verwenden. Denn die meisten Hersteller alphanumerischer Displays haben sich auf die Verwendung des ASCII-Zeichensatzes verständigt. In ihm sind Textzeichen einem 7 Bit umfassenden Muster zugeordnet. Der Bitmusterumfang reicht vom Wert 0 bis 127. Da Rechner meist mit einer internen Bitbreite von 2^n aufgebaut sind, wurde der ASCII-Code auf 8 Bit erweitert, so dass weitere 128 Sonderzeichen (zum Beispiel Umlaute) aufgenommen worden sind. Die genaue Zuordnung der Bitmuster von 128 bis 255 muss man jedoch dem Datenblatt des Displays entnehmen.

Die genaue Zuordnung der ersten 128 Zeichen in der ASCII-Tabelle ist im Anhang F zu finden.

2.1.2 BCD-Code

Möchte man Ziffern in Mikrorechnern verarbeiten, so müssen diese in einen Binärcode umgewandelt werden. Eine mögliche Form, Ziffern binär darzustellen ist der BCD-Code (Binary Coded Decimal). Um die Ziffern 0 bis 9 darstellen zu können, werden zehn verschiedene Bitmuster benötigt. Sie sind in Tabelle 2.1 dargestellt.

Tabelle 2.1 BCD-Codierung

Dezimal-zahl	Binär-code	Dezimal-zahl	Binär-code	Dezimal-zahl	Binär-code	Dezimal-zahl	Binär-code
0	0000	4	0100	8	1000	ungenutzt	1100
1	0001	5	0101	9	1001	ungenutzt	1101
2	0010	6	0110	ungenutzt	1010	ungenutzt	1110
3	0011	7	0111	ungenutzt	1011	ungenutzt	1111

Da für eine Ziffer 4 Bit zur Codierung notwendig sind, kann ein Byte zwei BDC-codierte Ziffern tragen. Man spricht dann von *gepacktem BCD-Format*. Auffällig beim BCD-Code ist, dass von den 16 verschiedenen Bitmustern, die aus 4 Bit gebildet werden können, nur 10 genutzt werden, also ganze 62,5 %. Mit den meisten Zentraleinheiten ist es möglich, auf das BCD-Format einfache Addition und Subtraktion anzuwenden. Andere Zahlendarstellungen haben sich gegenüber dem BCD-Code in Mikrorechnern durchgesetzt. Vorteilhaft anzuwenden ist er jedoch bei der Zifferndarstellung, zumal es elektronische Bausteine gibt, die es erlauben 7-Segment-Anzeigen direkt im BCD-Format anzusteuern.

2.2 Darstellung von Zahlen in Mikrorechnern

Alle Zahlensysteme, mit denen wir umgehen, sind sogenannte Stellenwertsysteme. Dabei bestimmt nicht allein das Ziffernelement den Wert der Zahl, sondern auch die Position, an der die Ziffer steht.

$$Z = A \cdot x^n + B \cdot x^m + \ldots$$

Das dezimale Zahlensystem ist uns aus dem täglichen Umgang bekannt. Die Ziffern 0 bis 9 sowie die Potenzen zur Basis 10 bilden dieses Zahlensystem. Wir wenden es an, ohne wirklich über den Aufbau der Zahlen nachzudenken. Vielmehr haben wir ein Gefühl für Zahlen entwickelt. Es fällt nicht schwer, Zahlen aufsteigend zu sortieren oder ganzzahlige Vorgänger bzw. Nachfolger einer Zahl zu nennen. Diese Fähigkeit sollen wir auch für andere Zahlensysteme entwickeln.

2.2.1 Binäres Zahlensystem

Binäre Zahlen werden durch ein nachgestelltes kleines b gekennzeichnet. Innerhalb der binären Zahlendarstellung gibt es verschiedene Möglichkeiten, die verfügbare Bitbreite des Rechners zu nutzen. Die folgenden Erläuterungen beziehen sich auf einen 8 Bit breiten Rechenraum. Dadurch fällt es leichter, die Darlegungen zu verstehen.

2.2.1.1 Vorzeichenlose ganze Zahlen

Das Binärsystem kennt nur die Ziffern 0 und 1. Die Basis des Zahlensystems bildet die 2. Folgt man diesen Gegebenheiten, so erkennt man den darstellbaren Werteumfang eines Bytes. Es

sind 256 verschiedene Bitmuster darstellbar. Bei der angegebenen Wertigkeit von 2^0 bis 2^7 ergeben sich die Zahlen 0 bis 255 als der darstellbare Werteumfang. Die folgende Darstellung verdeutlicht die Wertzuordnung im Binärsystem.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	1	1	1

$$1001\,0111\,\text{binär} = 1\cdot 2^7 + 1\cdot 2^4 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 151\,\text{dezimal}.$$

Das Bitmuster 0111 1011 entspricht folglich dem dezimalen Zahlenwert 123. Bei Binärzahlen, die 2 Byte umfassen, ergibt sich eine Darstellung von 0 (2^0) bis 65535 (2^{15}).

Wir wollen uns bereits an dieser Stelle angewöhnen, zwischen den einzelnen Nibbles (jeweils 4 Bit) eine kleine Lücke zu lassen, das erhöht die Übersichtlichkeit.

2.2.1.2 Vorzeichenlose gebrochene Zahlen, Festkommazahlen

Prinzipiell werden vorzeichenlose gebrochene Zahlen wie vorzeichenlose ganze Zahlen dargestellt. Lediglich die Exponenten sind andere. Im folgenden Beispiel sind die Wertigkeiten 2^{-7} bis 2^0 festgelegt. Der darstellbare Werteumfang reicht hier von 0 bis 1,9921875 bei einer kleinsten Schrittweite von 0,0078125.

2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
1	0	0	1	0	1	1	1

$$\begin{aligned} 1001\,0111\,\text{binär} &= 1\cdot 2^0 + 1\cdot 2^{-3} + 1\cdot 2^{-5} + 1\cdot 2^{-6} + 1\cdot 2^{-7} \\ &= 1{,}1796875\,\text{dezimal}. \end{aligned}$$

2.2.1.3 Vorzeichenbehaftete ganze Zahlen

Vorzeichen-Betrags-Darstellung

Es gibt verschiedene Ansätze, vorzeichenbehaftete Zahlen in Mikrorechnern darzustellen. Am einfachsten erscheint die Möglichkeit, das höchstwertige Bit, das MSB (Most Significant Bit), als Vorzeichenbit einzusetzen. Gesetztes Bit 7 bedeutet negative Zahl, gelöschtes Bit 7 bedeutet positive Zahl. Dann würde das Bitmuster aus den vorangegangenen Beispielen 1001 0111 der Zahl −23 entsprechen. Ein denkbar einfaches Prinzip – jedoch mit einem gravierenden Nachteil: Es gibt zwei Darstellungen der Zahl 0, nämlich +0 und −0. Oder anders dargestellt: Die Bitmuster 0000 0000 und 1000 0000 besitzen den gleichen Zahlenwert. Dass zwei unterschiedliche Bitmuster das Gleiche bedeuten, ist dem Rechner nur schwer klar zu machen. Auch gibt es gravierende Probleme bei arithmetischen Operationen, wie der Addition und Subtraktion.

Einerkomplementdarstellung

Man könnte jetzt auf die Idee kommen, durch die Komplementbildung einer Zahl (gemeint ist die bitweise Umkehr jedes einzelnen Bitzustandes der Zahl) das Vorzeichen zu ändern. Aus +1 wird −1 durch das Komplement des Bitmusters 0000 0001 zu 1111 1110. So weit, so gut. Aber auch hier gibt es zwei unterschiedliche Darstellungen für die Ziffer 0 (00000000 =