Mikroprozessoren und Mikrorechner

Lehrheft 1

ITT Fachlehrgänge, Pforzheim

ITT

4. Auflage, September 1977

Druck: Druckerei Seiter, 7535 Königsbach-Stein

Mikroprozessoren und Mikrorechner

Einleitung

Für zahlreiche Bereiche der Technik wird die Einführung der Mikroprozessoren von enormer Bedeutung sein. Viele Experten prophezeien, daß der Einsatz von Mikroprozessoren technische Umwälzungen mit sich bringen wird, die in ihrem Ausmaß mit der Einführung des Transistors vergleichbar sind. Das mag hochgegriffen erscheinen, aber es ist ganz sicher, daß die Mikroprozessoren, die für wenig Geld und auf kleinem Raum die volle Leistungsfähigkeit und Flexibilität eines Computers zur Verfügung stellen, in zahllosen Geräten eingesetzt und darüber hinaus viele neue Produkte ermöglichen werden. Es dauert sicher nicht mehr lange, bis uns Mikroprozessoren in allen Bereichen unseres täglichen Lebens begegnen. Angefangen bei Haushaltsgeräten wie Herde, Waschmaschinen usw. über Heizungsregelungen, Meßgeräte im Labor, Werkzeugmaschinensteuerungen bis hin zur Automobiltechnik bieten sich Einsatzmöglichkeiten für den Mikroprozessor. Dies bedeutet aber, daß Techniker und Ingenieure aus praktisch allen Sparten der Technik mit diesem Bauelement konfrontiert werden. Die Betroffenen stehen plötzlich vor neuen Problemstellungen, für die sie zum großen Teil nicht ausgebildet worden sind und auch gar nicht ausgebildet werden konnten.

Kennzeichnend für diesen Trend ist weiterhin, daß in Zukunft ein immer größer werdender Anteil der Entwicklung eines Gerätes auf die sog. Softwareentwicklung fällt, d.h. die Entwicklung von Computerprogrammen zur Steuerung des Gerätes. So macht z. B. bei der Steuerung einer Waschmaschine mit Hilfe eines Mikroprozessors die Entwicklung des Programmes für diese spezielle Aufgabe einen wesentlichen Teil der Entwicklungsarbeit aus.

Dieser Lehrgang soll den Technikern und Ingenieuren eine fundierte Einführung in diese neue Technik geben. Um den Lernenden Schritt für Schritt an die neue Technik heranzuführen, wird zunächst die grundsätzliche Funktion sowie der prinzipielle Aufbau eines Computers oder Rechners besprochen. Hierzu sind einige Grundlagen der Digitaltechnik erforderlich, die in den Abschnitten 1. und 2. in knapper Form noch einmal erläutert werden.
Nach diesen grundlegenden Abschnitten wird dann in mehreren Stufen die Struktur eines einfachen funktionsfähigen Mikrocomputers zusammengestellt. Angefangen mit den einfachen Bausteinen der Digitaltechnik wird in mehreren Stufen ein komplettes Rechenwerk, also das Kernstück eines Mikrocomputers, entwickelt. Durch Hinzufügen von Speichern und eines Steuerwerkes entsteht schließlich ein einfacher aber funktionsfähiger Mikrocomputer. Zu jedem dieser Entwicklungsschritte können mit dem Experimentiersystem zahlreiche Experimente durchgeführt werden, so daß jeder Entwicklungsschritt nachvollzogen werden kann.
Nachdem der Teilnehmer mit dem Grundkonzept vertraut ist, wird der Mikrocomputer vom Standpunkt der Software, also der Programmierung, her behandelt. Die Eigenschaften eines Mikroprozessors sind festgelegt in Form eines sog. Instruktionssatzes, also der Summe der Befehle und Adressiermöglichkeiten, die ein Rechner bietet. Es wird nicht mehr erklärt, durch welche schaltungstechnische Maßnahmen jeder einzelne Befehl realisiert wird. Wichtig für den Anwender von Mikroprozessoren sind nicht die Kenntnisse der schaltungstechnischen Details, sondern vielmehr die Kenntnisse, die es ihm ermöglichen, anhand der vom Hersteller mitgelieferten Software Programme für seine Aufgaben zu entwickeln. Es werden also hier die Instruktionen oder Befehle sowie die Adressierarten, die in Mikrorechnern zur Verfügung stehen, behandelt. Über Programmbeispiele werden die Grundlagen der Programmierung und die Verwendung der Rechnerbefehle erläutert.
Für diese grundsätzlichen Betrachtungen und Programmübungen wird kein bestimmter Rechnertyp eines bestimmten Herstellers angesprochen. Vielmehr wollen wir Ihnen einen Überblick geben über die Befehle und Adressiermöglichkeiten, wie man sie heute in Mikroprozessoren findet.
Damit sind Sie befähigt, sich in jeden Rechnertyp einzuarbeiten und dann den für Ihre Problemstellung geeigneten Typ auszusuchen.
Aus diesem Grunde haben wir einen hypothetischen Mikrorechner entwickelt, der sich durch einen möglichst übersichtlichen Befehlssatz auszeichnet. Es werden hier alle die Tricks und Ausnahmen vermieden, die die Hersteller praktischer Mikroprozessoren anwenden, um an dieser und jener „Ecke" noch ein wenig mehr Leistung aus dem Produkt herauszuholen. Das Hauptaugenmerk liegt hier also auf einem klar strukturierten, leicht überschaubaren Instruktionssatz, in den Sie sich leicht einarbeiten können und an dem die wesentlichen Punkte klar herausgearbeitet werden können. Dieser zweite Hauptteil des Lehrganges wird ergänzt durch

Abschnitte über allgemeine Fragen der Programmierung und über die für das praktische Arbeiten unerläßlichen Hilfsprogramme.
Im dritten Hauptteil wird dann ein echter Mikrorechner, und zwar der weitverbreitete Typ 8080, besprochen. Dieser 8080 ist in dem Experimentiergerät enthalten, so daß Sie wieder selbst Programme schreiben und testen können, um sich somit in diesen populären Rechnertyp einzuarbeiten. Zu Ihrer Unterstützung ist in dem Experimentiergerät ein einfaches sog. Monitorprogramm enthalten, das Ihnen einen optischen Zugriff in die Register und Speicher des 8080 gestattet. Dieser Teil wird ergänzt durch einige ausgewählte Kapitel über Ein- und Ausgabemethoden, verfügbare Bauelemente usw.
Zum Schluß noch ein Wort über das Experimentiergerät selbst. Hierin ist ein vollständiges 8080-Mikrorechnersystem mit einem 8080-Mikroprozessor, Speicher, Ein- und Ausgabeschaltungen enthalten. Dieser Rechner wird dazu verwendet, den Addierer-Subtrahierer, den Akkumulator usw. und schließlich den vereinfachten Rechner und den hypothetischen Mikrorechner zu simulieren. Das Experimentiergerät ist also eine praktische Anwendung eines Mikrorechners. In diesem Zusammenhang können Sie sich vielleicht vorstellen, welcher Aufwand erforderlich wäre, wenn all die Funktionen dieses Experimentiersystems mit „normalen" Digitalschaltungen aufgebaut wären. Hier wird alles mit einem entsprechend programmierten Mikrorechner durchgeführt.

Verfasser:
Dr. Jürgen Gerlach
C. D. Nabavi, B. Sc.
Forschungszentrum der
Standard Elektrik Lorenz AG
Stuttgart

1. Logische Grundfunktionen und Grundbausteine eines Rechners

1.1 Schaltalgebra

Digitale Schaltungen arbeiten mit 2 diskreten Zuständen. Das einfachste Beispiel dafür ist ein Schalter, der die beiden Zustände offen – geschlossen hat. Die Schaltalgebra, auch Boolesche Algebra genannt, ist ein mathematischer Formalismus zur Beschreibung von logischen Zusammenhängen digitaler Schaltfunktionen. Die Schaltalgebra baut auf einer zweiwertigen Logik auf, es werden nur binäre Variablen, also Variablen mit 2 definierten Zuständen, betrachtet. Die Schaltalgebra erlaubt es ebenso wie die „normale" Algebra bei Einhaltung bestimmter Rechenregeln, mit diesen binären Variablen Rechenvorgänge durchzuführen und widerspruchsfreie Funktionen aufzustellen.

1.2 Grundverknüpfungen der Schaltalgebra: NICHT, UND, ODER

Eine logische Funktion oder logische Verknüpfung im Sinne der Schaltalgebra ist eine Verknüpfung binärer Variablen, z. B. der Eingangs- und Ausgangsgrößen eines digitalen Schaltnetzes. In der Schaltalgebra kann man zeigen, daß sich jeder logische Zusammenhang zwischen binären Variablen mit Hilfe von 3 logischen Grundverknüpfungen ausführen läßt:
- NICHT-Verknüpfung (Negation)
- UND-Verknüpfung (Konjunktion)
- ODER-Verknüpfung (Disjunktion)

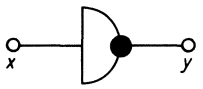
Verknüpfungen werden beschrieben durch:
1. Funktionstabellen (Wahrheitstabellen), die die Zuordnung der binären Variablen angeben
2. Schaltzeichen zur Beschreibung digitaler Schaltungen
3. Funktionsgleichungen

Logische NICHT-Verknüpfung

Diese Funktion negiert eine binäre Variable, d. h., ein L am Eingang wird zu H am Ausgang und umgekehrt. Im Bild 1.2.1 sind die 3 Beschreibungsmerkmale dargestellt.

x	y
L	H
H	L

a)



b)

$y = \bar{x}$

c)

Bild 1.2.1
NICHT-Verknüpfung
a) Funktionstabelle
b) Schaltzeichen
c) Funktionsgleichung

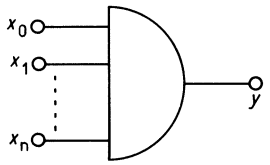
Bei der Funktionsgleichung $y = \bar{x}$ (lies: y ist gleich x NICHT) ist der Querstrich über dem x der Operator der Negation.

Logische UND-Verknüpfung

Die UND-Verknüpfung ist eine Beziehung zwischen **mindestens 2** unabhängigen Eingangsvariablen x_0, x_1 bis x_n und einer Ausgangsvariablen y. Die Ausgangsvariable hat **nur** dann den Zustand H, wenn **alle** Eingänge x_0, x_1 bis x_n **gleichzeitig** den Zustand H aufweisen. Die 3 Beschreibungsmerkmale sind in Bild 1.2.2 dargestellt.
In diesem Manuskript werden die Bezeichnungen aus der Mengenalgebra verwendet, da sie eindeutig sind. Das gelegentlich für die UND-Verknüpfung verwendete MAL-Zeichen · kann mit dem Multiplikationssymbol aus der Zahlenarithmetik verwechselt werden.

x_1	x_0	y
L	L	L
L	H	L
H	L	L
H	H	H

a)



b)

$$y = x_0 \wedge x_1 \wedge x_2 \wedge \ldots \wedge x_n$$

c)

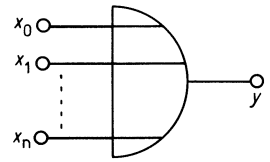
Bild 1.2.2
UND-Verknüpfung
a) Funktionstabelle (für 2 Eingangsvariablen)
b) Schaltzeichen
c) Funktionsgleichung

Logische ODER-Verknüpfung

Die ODER-Verknüpfung ist ebenfalls eine Beziehung zwischen mindestens 2 unabhängigen Eingangsvariablen x_0, x_1 bis x_n und einer Ausgangsvariablen y. Bei dieser Verknüpfung ist y immer dann H, wenn mindestens eine Eingangsvariable H ist (Bild 1.2.3).

x_1	x_0	y
L	L	L
L	H	H
H	L	H
H	H	H

a)



b)

$$y = x_0 \vee x_1 \vee x_2 \vee \ldots \vee x_n$$

c)

Bild 1.2.3
ODER-Verknüpfung
a) Funktionstabelle (für 2 Eingangsvariablen)
b) Schaltzeichen
c) Funktionsgleichung

Hier eine Hilfe für diejenigen Leser, die ebenso wie der Autor Schwierigkeiten haben mit dieser Symbolik aus der Mengenalgebra: Man weiß nicht, nach welcher Seite das Zeichen V umfällt, nach rechts **oder** nach links.

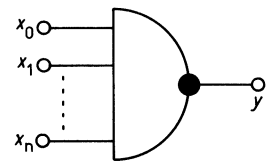
1.3 Einige von den Grundverknüpfungen abgeleitete logische Verknüpfungen

Logische NAND-Verknüpfungen

Die NAND-Verknüpfung entsteht durch Negation der UND-Verknüpfung (NAND = **N**OT **AND**). Für die NAND-Verknüpfung gelten die in Bild 1.3.1 dargestellten Beschreibungsmerkmale.

x_1	x_0	y
L	L	H
L	H	H
H	L	H
H	H	L

a)



b)

$$y = \overline{x_0 \wedge x_1 \wedge x_2 \wedge \ldots \wedge x_n}$$

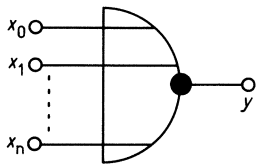
c)

Bild 1.3.1
NAND-Verknüpfung
a) Funktionstabelle (für 2 Eingangsvariablen)
b) Schaltzeichen
c) Funktionsgleichung

Logische NOR-Verknüpfung
Die Beschreibungsmerkmale zeigt Bild 1.3.2.

x_1	x_0	y
L	L	H
L	H	L
H	L	L
H	H	L

a)



b)

$$y = \overline{x_0 \vee x_1 \vee x_2 \vee \ldots \vee x_n}$$

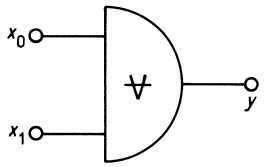
c)

Bild 1.3.2
a) Funktionstabelle (für 2 Eingangsvariablen)
b) Schaltzeichen
c) Funktionsgleichung

Logische EXCLUSIV-ODER-Verknüpfung
Die EXCLUSIV-ODER-Verknüpfung stellt eine Beziehung zwischen 2 Eingangsvariablen x_0, x_1 und der Ausgangsvariablen y in der Weise dar, daß y nur dann H ist, wenn x_0 und x_1 entgegengesetzte Zustände aufweisen (Bild 1.3.3).

x_1	x_0	y
L	L	L
L	H	H
H	L	H
H	H	L

a)



b)

$$y = x_0 \veebar x_1$$

c)

Bild 1.3.3
EXCLUSIV-ODER-Verknüpfung
a) Funktionstabelle
b) Schaltzeichen
c) Funktionsgleichung

Die Funktionstabelle zeigt, daß bei $x_0 = x_1 = L$ bzw. $x_0 = x_1 = H$ der Ausgang $y = L$ ist. Nur wenn die beiden Eingänge entgegengesetzte Zustände aufweisen, wird der Ausgang zu H. Aus diesem Grunde wird diese Verknüpfung auch als **ANTIVALENZ** bezeichnet.
Technisch läßt sich z. B. eine EXKLUSIV-ODER-Verknüpfung, wie in Bild 1.3.4 gezeigt, aus UND-, ODER- und NICHT-Verknüpfungen realisieren.

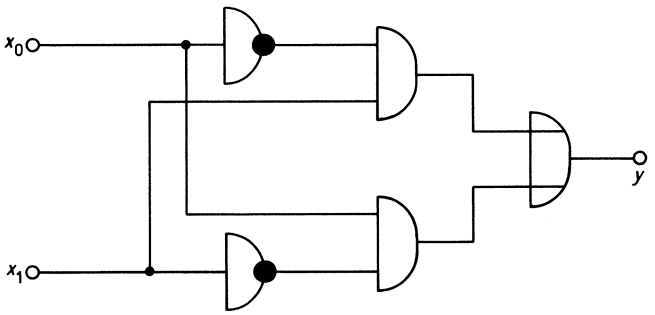


Bild 1.3.4
Aufbau einer EXKLUSIV-ODER-Verknüpfung aus den Grundfunktionen

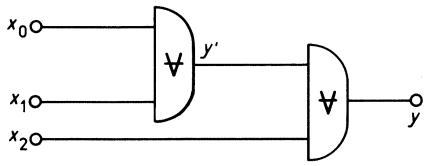


Bild 1.3.5
Kaskadierung von 2 EXKLUSIV-ODER-Verknüpfungen

In Bild 1.3.5 sind nun 2 EXKLUSIV-ODER-Verknüpfungen hintereinandergeschaltet (kaskadiert).
Insgesamt handelt es sich hier um eine logische Schaltung mit 3 Eingängen und einem Ausgang. Das logische Verhalten dieser Schaltung zeigt die Funktionstabelle Tab. 1.3.1.

x_2	x_1	x_0	y
L	L	L	L
L	L	H	H
L	H	L	H
L	H	H	L
H	L	L	H
H	L	H	L
H	H	L	L
H	H	H	H

Tab. 1.3.1
Funktionstabelle für Schaltung nach Bild 1.3.5

Hieraus ist zu erkennen, daß y immer dann H ist, wenn die an den Eingängen gleichzeitig vorhandenen H-Zustände ungeradzahlig sind.

1.4 Weitere logische Grundbausteine eines Rechners

Datenselektor (Multiplexer)

Ein Datenselektor ist ein steuerbarer Umschalter, mit dem in Abhängigkeit vom Zustand der Steuerleitungen S_i eine der Eingangsleitungen x_i zum Ausgang durchgeschaltet wird. Die Funktion ist in Bild 1.4.1 am Beispiel eines 4-zu-1-Datenselektors gezeigt.

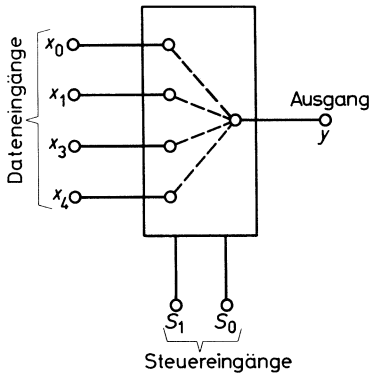


Bild 1.4.1
Funktion eines 4-zu-1-Datenselektors

Wenn man die Funktionstabelle für diesen Datenselektor wie bisher gewohnt angibt, so wird diese sehr groß, da insgesamt 6 Eingangsvariablen vorhanden sind (4 Daten- und 2 Steuereingänge). Außerdem ist die Funktion nicht sehr deutlich zu erkennen. Es soll deshalb eine abgekürzte Funktionstabelle verwendet werden (Tab. 1.4.1).

Steuereingänge S_1	S_0	Ausgang y
L	L	x_0
L	H	x_1
H	L	x_2
H	H	x_3

Tab. 1.4.1
Funktionstabelle eines 4-zu-1-Datenselektors

Die Ausgangsvariable y ist gleich einer Eingangsvariablen x_i, wobei i durch S_0 und S_1 bestimmt ist.

Decoder (Demultiplexer)
In Bild 1.4.2 ist das Prinzipschaltbild eines Decoders dargestellt.

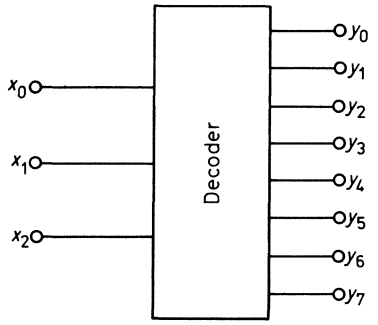


Bild 1.4.2
Prinzipschaltbild eines Decoders

Dieser Decoder hat 3 Eingangsleitungen x_0 bis x_2 und 8 Ausgangsleitungen y_0 bis y_7 (3-zu-8-Decoder). Je nach Kombination der Eingangszustände kann eine bestimmte Ausgangsleitung auf L oder H geschaltet werden. Welche Eingangskombination einen bestimmten Ausgang schaltet, hängt von der Art des jeweiligen Decoders ab. In Tab. 1.4.2 ist eine mögliche Funktionstabelle eines Decoders dargestellt.

x_2	x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
L	L	L	H	L	L	L	L	L	L	L
L	L	H	L	H	L	L	L	L	L	L
L	H	L	L	L	H	L	L	L	L	L
L	H	H	L	L	L	H	L	L	L	L
H	L	L	L	L	L	L	H	L	L	L
H	L	H	L	L	L	L	L	H	L	L
H	H	L	L	L	L	L	L	L	H	L
H	H	H	L	L	L	L	L	L	L	H

Tab. 1.4.2
Funktionstabelle eines Decoders

Aus der Tabelle ist z. B. zu erkennen, daß der Ausgang y_6 dann H wird, wenn an den Eingängen x_0 = L, x_1 = H und x_2 = H herrscht.
In der Praxis werden auch häufig Decoder mit negierten Ausgängen verwendet. In diesem Falle wären alle Ausgangszustände in Tab. 1.4.2 negiert.
Allgemein hat ein Decoder n Eingänge und 2^n Ausgänge.

1.5 Flipflop-Schaltungen

Ein Flipflop (kurz: FF) ist eine bistabile Kippstufe, die die beiden logischen Zustände L und H einnehmen kann. Über einen oder mehrere Eingänge kann der Zustand des Flipflops beeinflußt werden. Anders als bei den bisher behandelten Gatterfunktionen und kombinatorischen Netzwerken hängt der neue Zustand des Flipflops oft nur von dem Zustand ab, in dem sich das Flipflop unmittelbar vor einer Ansteuerung befand. Auch wenn die Eingangssignale verschwinden, die zu einem bestimmten Zustand des Flipflops geführt haben, kann ein Flipflop seinen Zustand beliebig lang beibehalten, es kann somit als Speicherelement verwendet werden.
Das einfachste FF ist das sog. Basis-FF oder auch RS-FF. Die Bezeichnung RS ist die Abkürzung für S = set = setzen und R = reset = zurücksetzen.
Das Schaltsymbol eines solchen FFs zeigt Bild 1.5.1

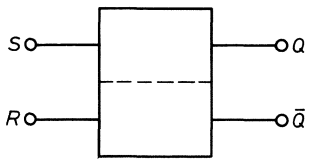


Bild 1.5.1
Schaltsymbol eines RS-FFs

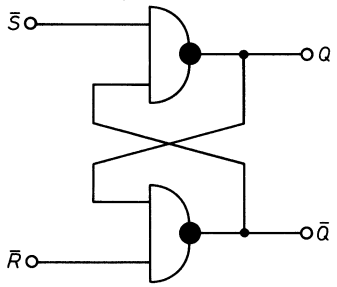
Nach der Definition wird das Flipflop durch S = H und R = L in den Zustand Q = H und $\overline{Q}$ = L gesetzt. Das Rücksetzen durch S = L und R = H. Dies sind die beiden stabilen Zustände des Flipflops.
Das Funktionsverhalten läßt sich am einfachsten an der Funktionstabelle erläutern (Tab. 1.5.1).

Tab. 1.5.1
Funktionstabelle eines RS-FFs

S	R	Q	$\overline{Q}$	
L	L			keine Änderung des vorherigen Zustandes
L	H	L	H	
H	L	H	L	
H	H	(H	H)	irregulärer Zustand

Werden beide Eingänge mit $S = R$ = L beschaltet, so bleibt das FF in seiner momentanen Lage Q = L, $\overline{Q}$ = H oder Q = H, $\overline{Q}$ = L. Durch die beiden mittleren Kombinationen stellen sich die beiden vorher genannten definierten Ausgangszustände ein. Der Zustand $S = R$ = H wird als irregulär bezeichnet, da hier beide Ausgänge H sind. Wird die Eingangsinformation von $S = R$ = H nach $S = R$ = L geändert, stellt sich ein nicht vorhersagbarer Zustand des Flipflops ein. Es hängt allein von Zufälligkeiten (Bauteiletoleranzen, zeitliche Verzögerungen usw.) ab, in welchen Zustand das Flipflop kippt. Aus diesem Grunde vermeidet man die Kombination $S = R$ = H. Die eigentliche Speicherherstellung des Flipflops ist die Kombination $S = R$ = L.

Praktisch läßt sich ein RS-Flipflop z. B. durch 2 kreuzgekoppelte NAND-Gatter realisieren (Bild 1.5.2).



a)

$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	
L	L	(H	H)	irregulär
L	H	H	L	
H	L	L	H	
H	H			keine Änderung des vorherigen Zustandes

b)

Bild 1.5.2
RS-Flipflop aus NAND-Gattern
a) Funktionsschaltbild
b) Funktionstabelle

Im Gegensatz zum Flipflop nach Bild 1.5.1 sind bei diesem Flipflop die Ausgangszustände vertauscht. Dies deuten die Querstriche über den Buchstaben der Eingangsvariablen an.

Ein weiterer sehr wichtiger Flipflop-Typ ist das sog. Zweispeicher-Flipflop, auch Master-Slave-Flipflop genannt. Es besteht im wesentlichen aus 2 Basis-Flipflops, die hintereinandergeschaltet sind. Die Steuerleitungen sind über zusätzliche NAND-Gatter durch einen Takt schaltbar. Der grundsätzliche Schaltungsaufbau ist in Bild 1.5.3 gezeigt.
Betrachten wir zunächst das Master-FF: Liegt am Takteingang L, sind die beiden Ausgänge der Taktsteuerung I H. Dies bedeutet, daß ein Zustandswechsel an den Eingängen S und R keinen Einfluß auf das Master-FF hat, das FF behält seinen momentanen Zustand. Liegt dagegen am Takteingang H, bestimmt der Zustand an S und R den Zustand des Master-FFs.

Das gleiche Verhalten weist auch das Slave-FF auf. Allerdings wird die Taktsteuerung durch den Taktinverter invertiert. Dies bedeutet, daß der Takt H am Master-FF beim Slave-FF als L anliegt und umgekehrt.
Die Funktion dieses FFs läßt sich am einfachsten am Zeitablauf eines Taktimpulses erläutern (Bild 1.5.4).

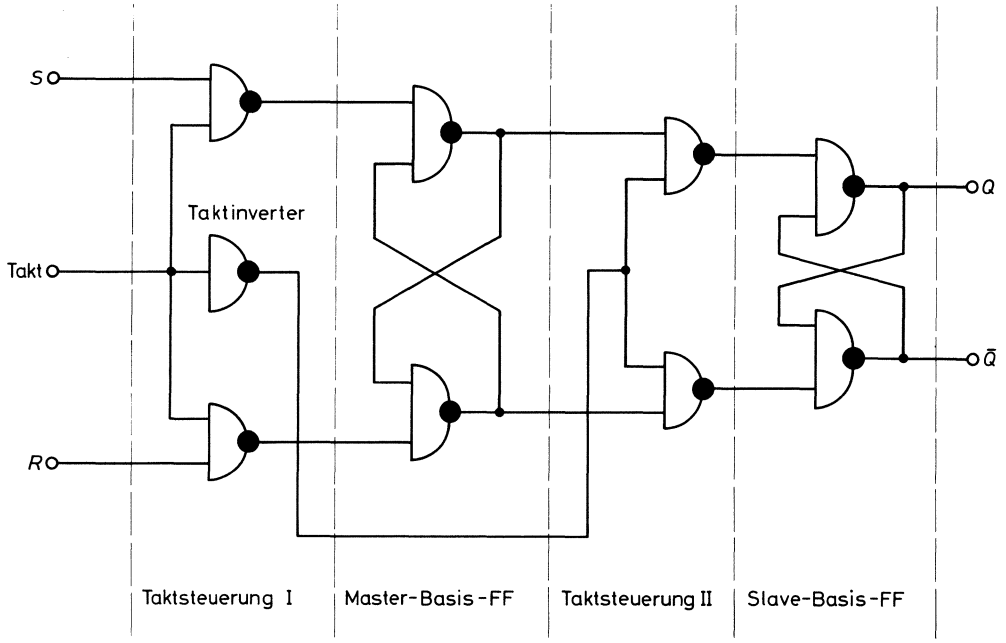


Bild 1.5.3
Zweispeicher-Flipflop aus NAND-Gattern

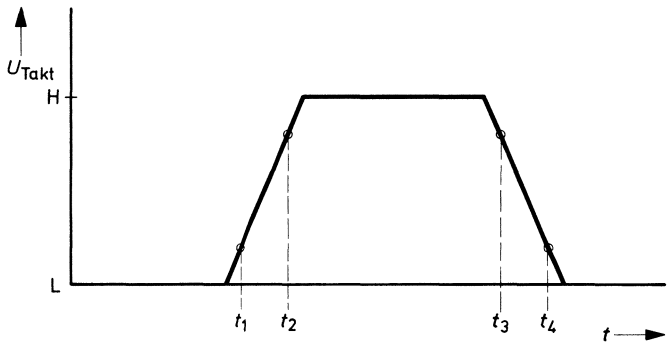
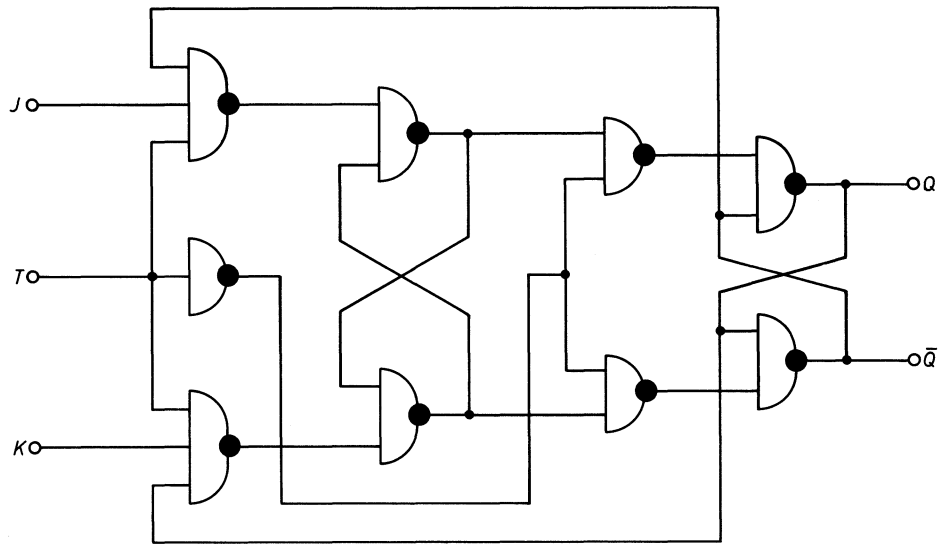


Bild 1.5.4
Zeitlicher Ablauf bei einem Master-Slave-FF

t_1: Verläßt der ansteigende Taktimpuls den L-Toleranzbereich in Richtung H, so geht der Ausgang des Taktinverters auf L, d. h., die Eingänge des Slave-Flipflops werden abgeschaltet, das Slave-Flipflop behält seinen Zustand bei.
t_2: Erreicht der ansteigende Taktimpuls den unteren Wert des H-Toleranzbereiches, werden die Eingänge des Master-Flipflops durchgeschaltet, d. h., das Master-Flipflop nimmt den durch die *R*- und *S*-Eingänge bestimmten Zustand an.
t_3: Verläßt die abfallende Flanke des Taktimpulses den H-Toleranzbereich in Richtung L, werden die Eingänge des Master-Flipflops wieder gesperrt, d. h., das Master-Flipflop behält seinen gewonnenen Zustand bei.
t_4: Erreicht die abfallende Flanke des Taktsignals den oberen Wert des L-Toleranzbereiches, werden die Eingänge des Slave-Flipflops durchgeschaltet, d. h., das Slave-Flipflop übernimmt den Zustand des Master-Flipflops.

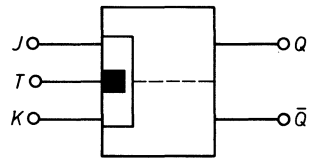
Damit ist also folgendes erreicht: Die in der Zeit t_1 bis t_2 an den *R*- und *S*-Eingängen anliegende Information wird in das Flipflop übernommen, dort zwischengespeichert und zum Zeitpunkt t_4 an den Ausgängen wirksam. Solange der Takteingang auf L ist, bleibt die Information im Flipflop gespeichert.
Eine besondere praktische Bedeutung hat eine Erweiterung des Master-Slave-FFs, das JK-Master-Slave-FF. Es wird dadurch realisiert, daß die Ausgänge über kreuz auf die Eingangsgatter geführt werden (Bild 1.5.5).
Die Funktion dieses FFs kann direkt aus der Funktionstabelle abgelesen werden:
Der Zustand $J = K =$ L vor einem Taktimpuls (Zeitpunkt t_n) bewirkt keine Änderung des vorhandenen Zustandes mit dem folgenden Taktimpuls (Zeitpunkt t_{n+1}).



a)

t_n		t_{n+1}		
K	J	Q	$\bar{Q}$	
L	L	Q	$\bar{Q}$	keine Änderung
L	H	H	L	
H	L	L	H	
H	H	$\bar{Q}$	Q	Komplement

b)



c)

Bild 1.5.5
JK-Master-Slave-FF
a) Schaltung mit NAND-Gattern
b) Funktionstabelle
c) Schaltsymbol

Der Eingangszustand J = H und K = L bewirkt mit einem Taktimpuls den Ausgangszustand Q = H und $\bar{Q}$ = L und umgekehrt. Bei $J = K$ = H ändert sich der Ausgangszustand mit jedem Taktimpuls (Bild 1.5.6).

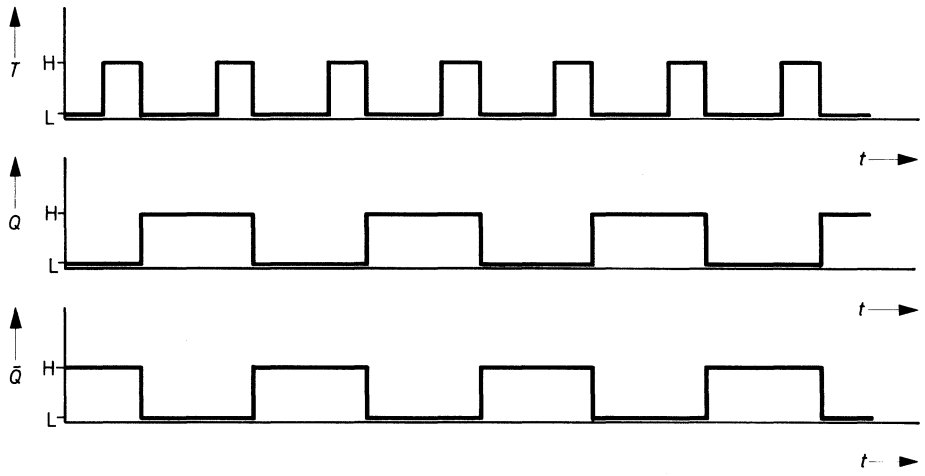
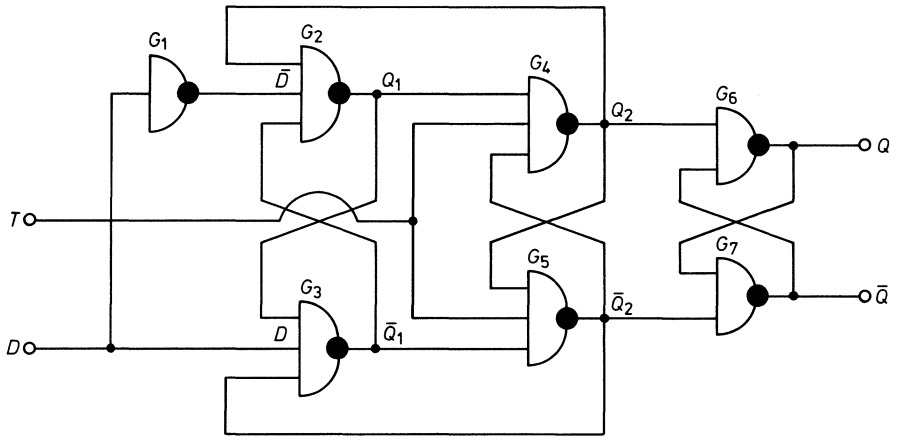


Bild 1.5.6
Impulsdiagramm eines JK-Master-Slave-FFs bei $J = K$ = H

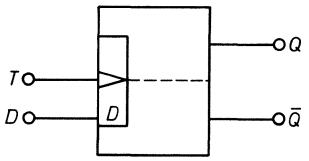
Ein FF, das sich ähnlich verhält wie das JK-Master-Slave-FF bei $J = K$ = H, ist das sog. D-FF. Eine in der Praxis sehr verbreitete Variante ist in Bild 1.5.7 dargestellt.



a)

t_n	t_{n+1}	
D	Q	$\bar{Q}$
L	L	H
H	H	L

b)



c)

Bild 1.5.7
Einflanken-getriggertes D-FF
a) Schaltung mit NAND-Gattern
b) Funktionstabelle
c) Schaltsymbol

Eine Besonderheit an diesem FF gegenüber dem vorher angesprochenen JK-Master-Slave-FF ist, daß die an D liegende Eingangsinformation während der L→H-Flanke des Taktimpulses an die Ausgänge übertragen wird. Ist der Takt H und ändert sich jetzt die Information an D, so hat diese Änderung keinen Einfluß mehr auf den Ausgangszustand. Eine Änderung an D während T = H wird also erst beim nächsten L→H-Sprung an den Ausgängen wirksam. Durch interne Verzögerungen ist es also bei diesem FF möglich, eine Rückkopplung von z. B. $\bar{Q}$ nach D einzuführen, ohne daß dieses FF eine Schwingneigung aufweist. Nach dieser Eigenschaft hat dieses FF auch die Bezeichnung D-FF von D = delay = Verzögerung.

Die Arbeitsweise dieses FFs ist folgende: Solange T = L ist, sind die Ausgänge der Gatter G_4 und G_5 H, so daß das FF G_6, G_7 den Ausgangszustand beibehält. Ist z. B. D = H, und der Takt steigt von L nach H (positive Flanke), so wird bei Erreichen eines bestimmten Stellenwertes das FF G_4, G_5 durchgeschaltet, d. h., eine Information von den Ausgängen des FFs G_2, G_3 bewirkt ein Setzen des FFs G_6, G_7. In unserem Beispiel wird Q = H und $\bar{Q}$ = L. Ist T = H, und erfolgt eine Änderung an D (z. B H→L), so kann durch die Rückführung von Q_2 und $\bar{Q}_2$ nach FF G_2, G_3 keine Änderung des Zustandes von FF G_6, G_7 erfolgen. Versuchen Sie einmal selbst, anhand von Bild 1.5.7a das Verhalten der Schaltung „durchzuspielen".

In der praktischen Schaltungsentwicklung werden eine Reihe verschiedener Flipflop-Typen eingesetzt, die sich in den Steuereingängen, im Triggerverhalten usw. unterscheiden. Eine genaue Kenntnis dieser Flipflop-Typen ist für die Schaltungsentwicklung unerläßlich, für das Verständnis der prinzipiellen Funktion von Mikroprozessoren reicht die Kenntnis dieser grundlegenden Tatsachen aus.

1.6 Speicher

Die Speicherbauelemente haben eine entscheidende Bedeutung in Mikrorechnersystemen. Sie werden benötigt zur Speicherung der Rechnerprogramme und der Daten. Auf dem Markt sind verschiedene Speichertypen erhältlich, die sich in Größe, Funktionsweise, Technologie usw. unterscheiden. Im Rahmen dieses Lehrganges wollen wir vor allem die Fragen behandeln, die für den Einsatz in Mikrorechnern wesentlich sind, und Fragen der verschiedenen Technologien und der internen Organisationen nur am Rande streifen.

Einen Speicher in einem Mikrorechnersystem kann man als eine zweidimensionale Matrix von Speicherelementen, z. B. von Flipflops, betrachten. Bild 1.6.1 zeigt den prinzipiellen Aufbau eines Speichers der Größe 8 x 4 bit. Jeder Punkt bedeutet eine Speicherstelle, ein **bit** (bit ist die Abkürzung für binary digit, das mit zweiwertiger Ziffer übersetzt werden kann). Sollen nun Daten geschrieben oder gelesen werden, so wird eine binäre Kombination zwischen L L L und H H H, eine sog. Adresse, auf die Adreßleitungen gelegt. Der Adreßdecoder selektiert dann eine der 8 Zeilen der Matrix und den Inhalt der Zeile, so daß ein Datenwort von 4 bit über die Datenein- oder ausgabe geschrieben oder gelesen werden kann.

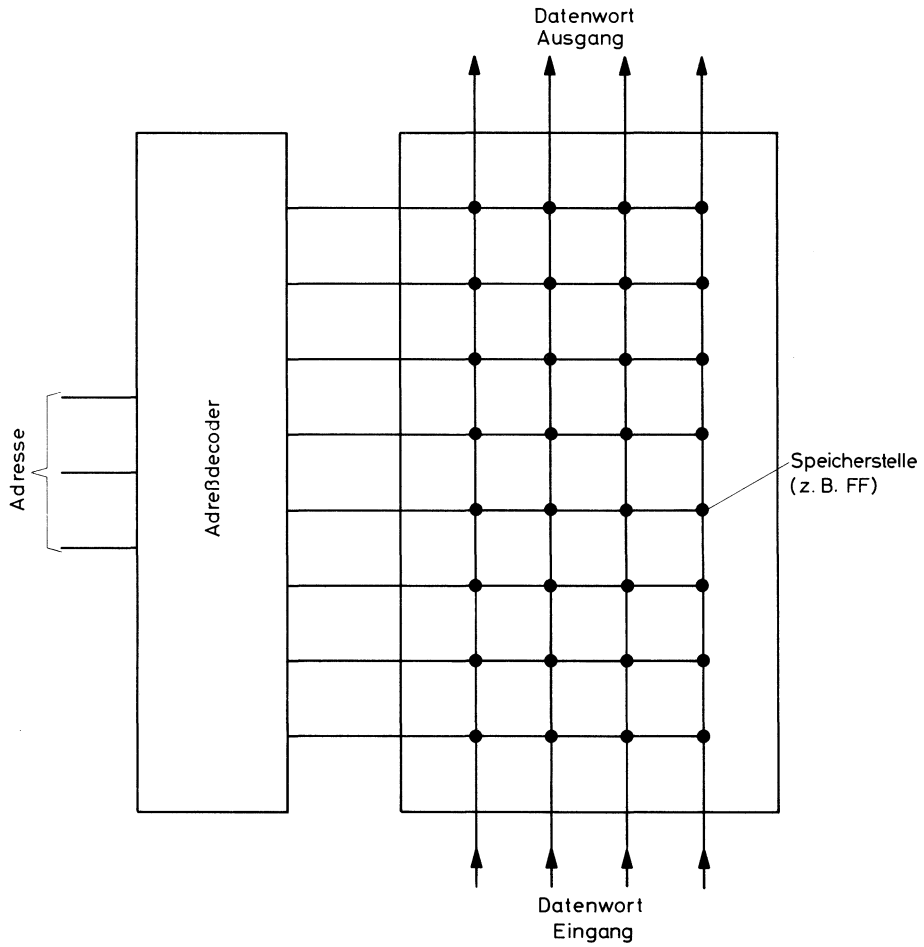


Bild 1.6.1
Prinzipieller Aufbau eines Speichers

Zur Steuerung von Lesen, Schreiben usw. sind zusätzliche Steuerleitungen vorhanden, die in Bild 1.6.1 nicht dargestellt sind.

Käufliche Speicherbauelemente haben üblicherweise Wortlängen von 1, 2, 4 oder 8 bit, es können also Datenwörter von 1, 2, 4 oder 8 bit gespeichert werden. Die Zahl der Zeilenmatrix ist üblicherweise eine Potenz von 2, da mit n Adreßleitungen eine aus 2^n Zeilen selektiert werden kann. Ein Speicherbauelement könnte im Datenblatt etwa folgendermaßen bezeichnet sein:

Kapazität 1 Kilobit = 1 k = 2^{10} bit = 1 024 bit = 1 024 Speicherelemente
Organisation 256 x 4, also 256 = 2^8 Zeilen mit je 4 bit

Wenn wir nun verschiedene Speichertypen betrachten, so müssen wir zunächst 2 Hauptgruppen unterscheiden:
1. **Schreib-Lese-Speicher**, also Speicher, in die man Daten einschreiben und später wieder auslesen kann. Dieser Speichertyp wird auch als RAM vom englischen **R**andom **A**ccess **M**emory bezeichnet.

Anmerkung:
Dieser Begriff hat sich eingebürgert, obwohl er nichts darüber aussagt, daß der Speicher gelesen und geschrieben werden kann. Er besagt lediglich, daß jedes Speicherwort wahlfrei, also beliebig, adressiert werden kann.

2. **Festwert-Speicher**, auch als ROM (engl.: **R**ead **O**nly **M**emory) bezeichnet. In einem solchen Speicher werden z. B. beim Herstellungsprozeß durch entsprechende Masken Daten eingeschrieben, die dann später nur noch ausgelesen werden können.
Die Speichergruppen lassen sich weiter nach anderen Gesichtspunkten untergliedern, nach der Technologie z. B. bipolar, MOS (engl.: **M**etal **O**xyde **S**emiconductor) oder bei RAMs danach, ob es notwendig ist, die einmal gespeicherte Information periodisch aufzufrischen oder nicht, d. h., ob der Speicher dynamisch oder statisch arbeitet.
In dynamischen Speichern sind die Speicherzellen im Prinzip Kondensatoren, deren Ladung periodisch aufgefrischt werden muß. Statische Speicher dagegen, bei denen die Speicherzellen Flipflops sind, benötigen eine solche Auffrischung (refresh) nicht.
Eine detaillierte Behandlung aller dieser Punkte würde den Rahmen dieses Lehrganges sprengen, im konkreten Anwendungsfall müssen die genauen Spezifikationen der Bauelemente den Datenblättern der Hersteller entnommen werden.
2 Gruppen von Festwertspeichern sollen aber noch erwähnt werden, da sie eine große praktische Bedeutung beim Einsatz von Mikroprozessoren haben. Neben den beim Herstellungsprozeß programmierten ROMs gibt es noch sogenannte PROMs (engl.: **P**rogrammable **R**ead **O**nly **M**emories), die auch vom Benutzer elektrisch **einmal** beliebig programmiert werden können. Danach können die Daten nur noch ausgelesen werden.
Außerdem gibt es die sog. RePROMs (engl.: **Re P**rogrammable **R**ead **O**nly **M**emories). Diese Festwertspeicher können vom Benutzer elektrisch programmiert und falls erforderlich wieder gelöscht und neu programmiert werden. Nach einer Programmierung werden die Daten dann wieder nur ausgelesen. Trotz dieser Programmierbarkeit sind diese RePROMs aber grundsätzlich von Schreib-Lese-Speichern zu unterscheiden. Es handelt sich um Festwertspeicher, die Änderung der Speicherinhalte erfordert einen erheblichen Aufwand im Vergleich zu Schreib-Lese-Speichern.

1.7 Schieberegister

Ein Schieberegister besteht grundsätzlich aus einer Reihenschaltung mehrerer FFs, bei denen mit einem Taktimpuls die Information des vorherigen FFs an das nächste FF übergeben wird (Bild 1.7.1).

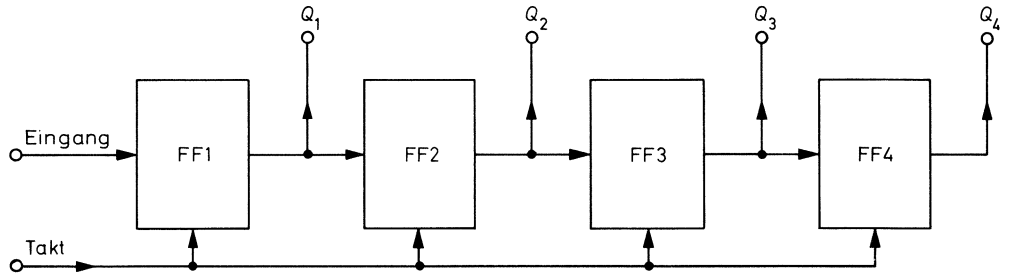


Bild 1.7.1
Prinzipschaltung eines Schieberegisters

Die Arbeitsweise ist folgende: Wenn z. B. vor dem ersten Taktimpuls (t_n) alle Ausgänge Q_1 bis Q_4 gleich L sind und am Eingang H liegt, erscheint nach dem ersten Taktimpuls (t_{n+1})

dieses H am Ausgang Q_1. Vor dem zweiten Taktimpuls wird jetzt der Eingang wieder auf L gebracht. Damit erscheint am Ausgang Q_1 bei t_{n+2} wieder L, während Q_2 = H wird. Nach t_{n+3} bleibt Q_1 = L, Q_2 wird L, während Q_3 = H wird. Nach t_{n+4} erreicht Q_4 den H-Zustand. Dieses Verhalten ist in Tab. 1.7.1 dargestellt.

Zeit	t_n	t_{n+1}	t_{n+2}	t_{n+3}	t_{n+4}	t_{n+5}
Eingang	H	L	L	L	L	L
Q_1	L	H	L	L	L	L
Q_2	L	L	H	L	L	L
Q_3	L	L	L	H	L	L
Q_4	L	L	L	L	H	L

Tab. 1.7.1
Funktion eines Schieberegisters (Informationsfluß)

Diese Tabelle zeigt deutlich, wie die Information H mit jedem Taktimpuls um eine Stelle nach rechts wandert. Nach dem 5. Taktimpuls sind alle Ausgänge wieder L, da der Eingang nach dem 1. Taktimpuls keine neue H-Information übernommen hat.
Ein reales Schieberegister aus JK-Master-Slave-FFs zeigt Bild 1.7.2.

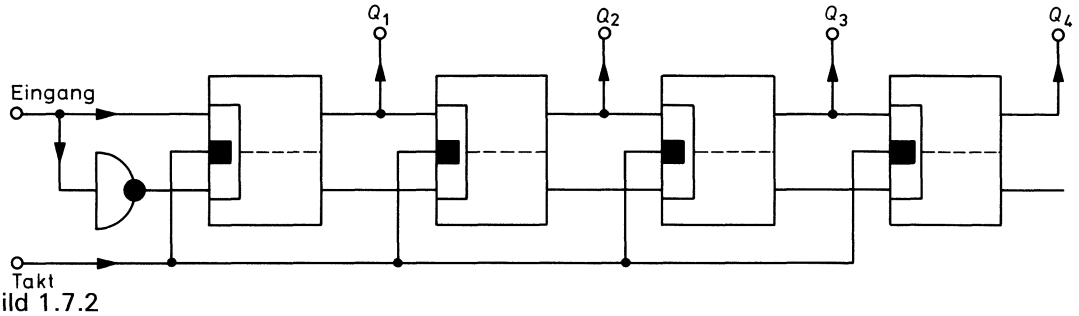


Bild 1.7.2
4-bit-Schieberegister aus JK-Master-Slave-FFs

Da dieses Schieberegister nur 4 FFs enthält, kann es auch nur 4 Informationen aufnehmen und wird deshalb als 4-bit-Schieberegister bezeichnet.
Es gibt bei diesem Schieberegister grundsätzlich 2 Möglichkeiten, die Informationen neu auszulesen:
– Nach 4 Taktimpulsen stehen an den Ausgängen Q_1 bis Q_4 gleichzeitig die eingegebenen Informationen zur Verfügung. Man kann also die seriell (nacheinander) eingelesenen Informationen parallel (gleichzeitig) auslesen.
– Benutzt man nur Q_4 als Ausgang, so können hier die seriell eingelesenen Daten auch seriell ausgelesen werden.
Schieberegister können also zur vorübergehenden Speicherung oder Verzögerung von Informationsfolgen benutzt werden. Ein weiteres bedeutendes Anwendungsgebiet ist die Parallel/Serien- oder Serien/Parallel-Umsetzung (Bild 1.7.3).
Beim Parallel/Serien-Betrieb werden die an den Eingängen *a* bis *d* liegenden Informationen durch einen Setzimpuls direkt in das Register übernommen. Am Serienausgang erscheinen diese Informationen im Zeitraster des nachfolgenden Schiebetaktes. Werden dagegen am Serieneingang Informationen synchron mit dem Schiebetakt eingegeben, so stehen sie nach abgeschlossener Eingabe parallel an den Ausgängen Q_1 bis Q_4 (Serien/Parallel-Betrieb). Über die Eingänge *C* können die einzelnen FFs über einen Impuls in eine Neutrallage zurückgestellt werden.
Schieberegister werden im Rechnersystem für verschiedene Aufgaben eingesetzt, wobei folgende Betriebsarten ausgenutzt werden:
– Verschieben von Informationen
– Serielle Dateneingabe mit serieller Datenausgabe
– Serielle Dateneingabe mit paralleler Datenausgabe
– Parallele Dateneingabe mit serieller Datenausgabe
– Parallele Dateneingabe mit paralleler Datenausgabe
Die letztgenannte Betriebsart kann mit dem Schieberegister nach Bild 1.7.3 dadurch realisiert werden, daß die Daten an den Paralleleingängen nach einem Setzimpuls an den Datenausgängen abgenommen werden. Sie stehen hier so lange zur Verfügung, bis neue Daten ein-

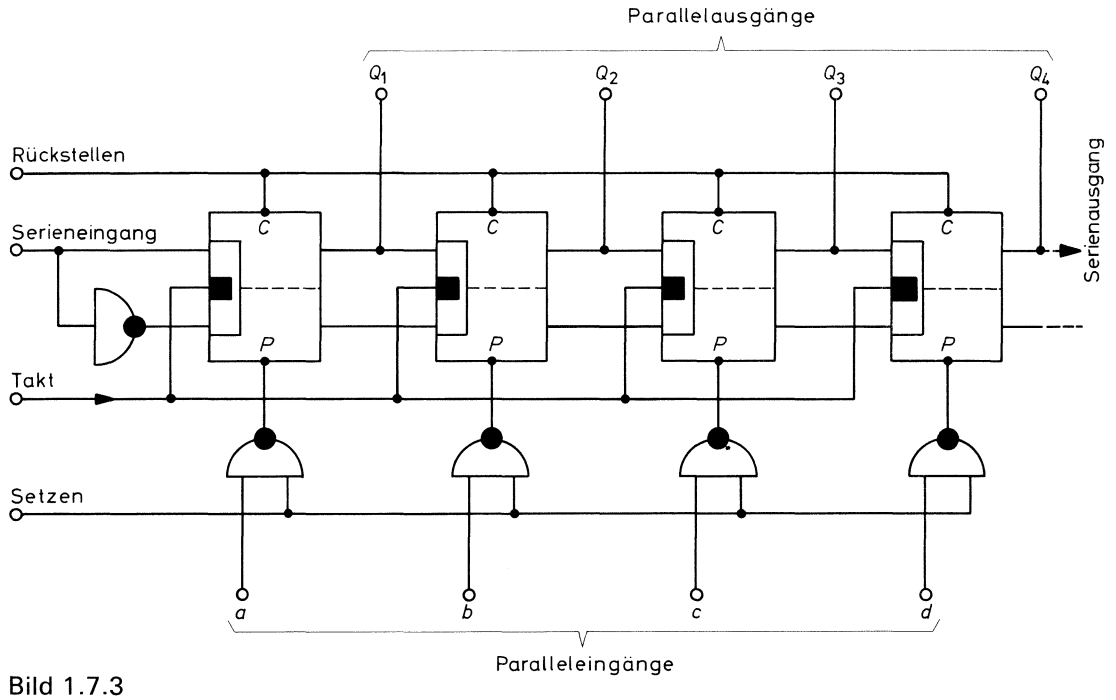


Bild 1.7.3
Schieberegister zur Parallel/Serien- oder Serien/Parallel-Umsetzung

geschrieben werden bzw. das Register über den Rückstelleingang gelöscht wird (Zwischenspeicherbetrieb).
Zum Schluß dieses Abschnittes soll noch eine Registerschaltung erklärt werden, die vom Prinzip her in Mikroprozessorsystemen Anwendung als Zwischenspeicher findet (Bild 1.7.4).

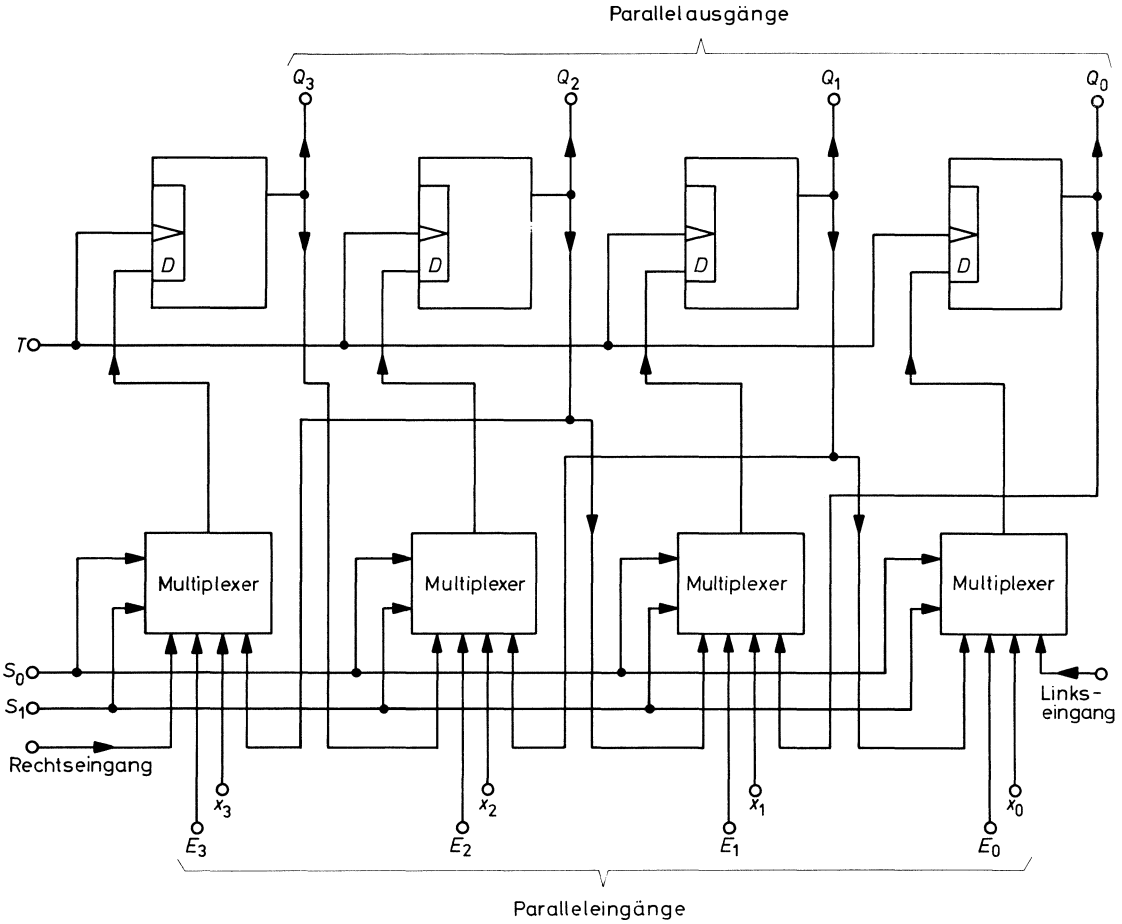


Bild 1.7.4
Register mit Multiplexern in den *D*-Eingängen der FFs

Die Arbeitsweise dieser Schaltung ist folgende: Über die Steuereingänge S_0, S_1 können die 4 Multiplexer auf einen ihrer 4 Eingänge geschaltet werden, d. h., die Information des selektierten Einganges erscheint jeweils am Ausgang. Werden z. B. die Paralleleingänge E_3 bis E_0 selektiert, so steht parallel zu den D-Eingängen der FFs die Eingangsinformation. Mit der nächsten positiven Taktflanke wird diese Information von den FFs übernommen und erscheint somit an den Ausgängen Q_3 bis Q_0. Diese Information wird so lange gespeichert, bis der nächste Taktimpuls eine neue Information von E_3 bis E_0 in das Register einliest.

Über eine andere Steuerkombination S_0, S_1 können z. B. die rechten Eingänge der Multiplexer mit deren Ausgängen verbunden werden. Jetzt kann eine Information, die am Linkseingang der Schaltung steht, seriell in das Register eingelesen werden. Der Ablauf ist folgender: Steht z. B. am Linkseingang die serielle Kombination H L H L, so erscheint mit dem 1. Taktimpuls das H an Ausgang Q_0 und am selektierten Eingang des nächsten Multiplexers. Mit dem 2. Taktimpuls wird Q_0 = L und Q_1 = H, mit dem 3. Taktimpuls Q_0 = H, Q_1 = L, Q_2 = H und mit dem 4. Taktimpuls Q_0 = L, Q_1 = H, Q_2 = L und Q_3 = H. Die serielle Eingangskombination ist also in Linksrichtung in das Register eingelesen worden. Analog dazu geschieht das serielle Einlesen über den Rechtseingang. Die Eingänge x_3 bis x_0 sind in diesem Beispiel nicht belegt. Sie könnten aber als gemeinsame Löscheingänge benutzt werden, indem alle Eingänge x_3 bis x_0 an L-Pegel gelegt werden. Werden jetzt über S_0, S_1 die x-Eingänge selektiert, werden mit dem folgenden Taktimpuls alle Ausgänge Q_3 bis Q_0 auf L gesetzt.

1.8 Zähler

Als Zähler werden Schaltungen bezeichnet, bei denen innerhalb gewisser Grenzen eine eindeutige Zuordnung zwischen der Anzahl der hineingegebenen Impulse und dem jeweiligen Ausgangszustand besteht. Die Kernbausteine eines Zählers sind ebenfalls FFs. In Bild 1.8.1 ist ein 4-bit-Binärzähler dargestellt.

Die Arbeitsweise dieses Zählers soll anhand des Impulsdiagramms erläutert werden:

Vor dem 1. Taktimpuls liegen die Ausgänge Q_1 bis Q_4 auf L. Dem Binärmuster L L L L wird die Zahl 0 zugeordnet. Nach dem 1. Taktimpuls erscheint das Muster L L L H, dem 1, nach dem 2. Taktimpuls L L H L, dem die Zahl 2 usw. zugeordnet ist. Insgesamt entstehen 16 unterschiedliche Binärkombinationen, denen die Zahlen 0 bis 15 zugeordnet werden können. Nach dem 16. Taktimpuls springen alle Ausgänge wieder in den Ausgangszustand L L L L, und bei weiteren Taktimpulsen wiederholt sich der vorher beschriebene Vorgang. Allgemein gilt:

Ein n-bit-Zähler kann 2^n unterschiedliche Ausgangskombinationen einnehmen. Da eine Kombination der Zahl 0 zugeordnet werden muß, kann ein solcher Zähler bis $2^n - 1$ zählen, bevor sich der Zählzyklus wiederholt.

Besteht ein Zähler z. B. aus 8 entsprechend Bild 1.8.1a hintereinandergeschalteten FFs, so können sich an seinen Ausgängen $2^8 = 256$ unterschiedliche Binärkombinationen ergeben, denen man die Zahlen von 0 bis 255 zuordnen kann.

Durch bestimmte Beschaltung ist es möglich, z. B. einen 4-bit-Zähler in seiner Zählkapazität von 16 verschiedenen Kombinationen auf 10 Kombinationen zu begrenzen. Ein Zählzyklus wiederholt sich also nach 10 Taktimpulsen. In diesem Falle können den Binärmustern die Zahlen 0 bis 9 zugeordnet werden, und es handelt sich um einen Dezimalzähler bzw. BCD-Zähler (siehe auch Abschnitt 2.7). Vielfach werden auch Zähler benötigt, die nicht in der Reihenfolge L L L L, L L L H, L L H L usw. sondern in der Richtung H H H H, H H H L, H H L H usw. zählen.

Bei derselben Zahlenzuordnung wie in Bild 1.8.1b zählt dieser Zähler also rückwärts von 15, 14, 13 usw. bis nach 0 und beginnt dann wieder bei 15. Solche Zähler werden als Rückwärtszähler bezeichnet. Durch eine Steuerlogik können auch Zähler über ein entsprechendes Signal in ihrer Zählrichtung umgeschaltet werden.

Zähler gibt es in den verschiedensten Ausführungsformen. Wichtige Beurteilungskriterien sind:

- Zählrichtung (vor- und rückwärts)
- Taktsteuerung (asynchron oder synchron)
- Zählkapazität
- Zählcode

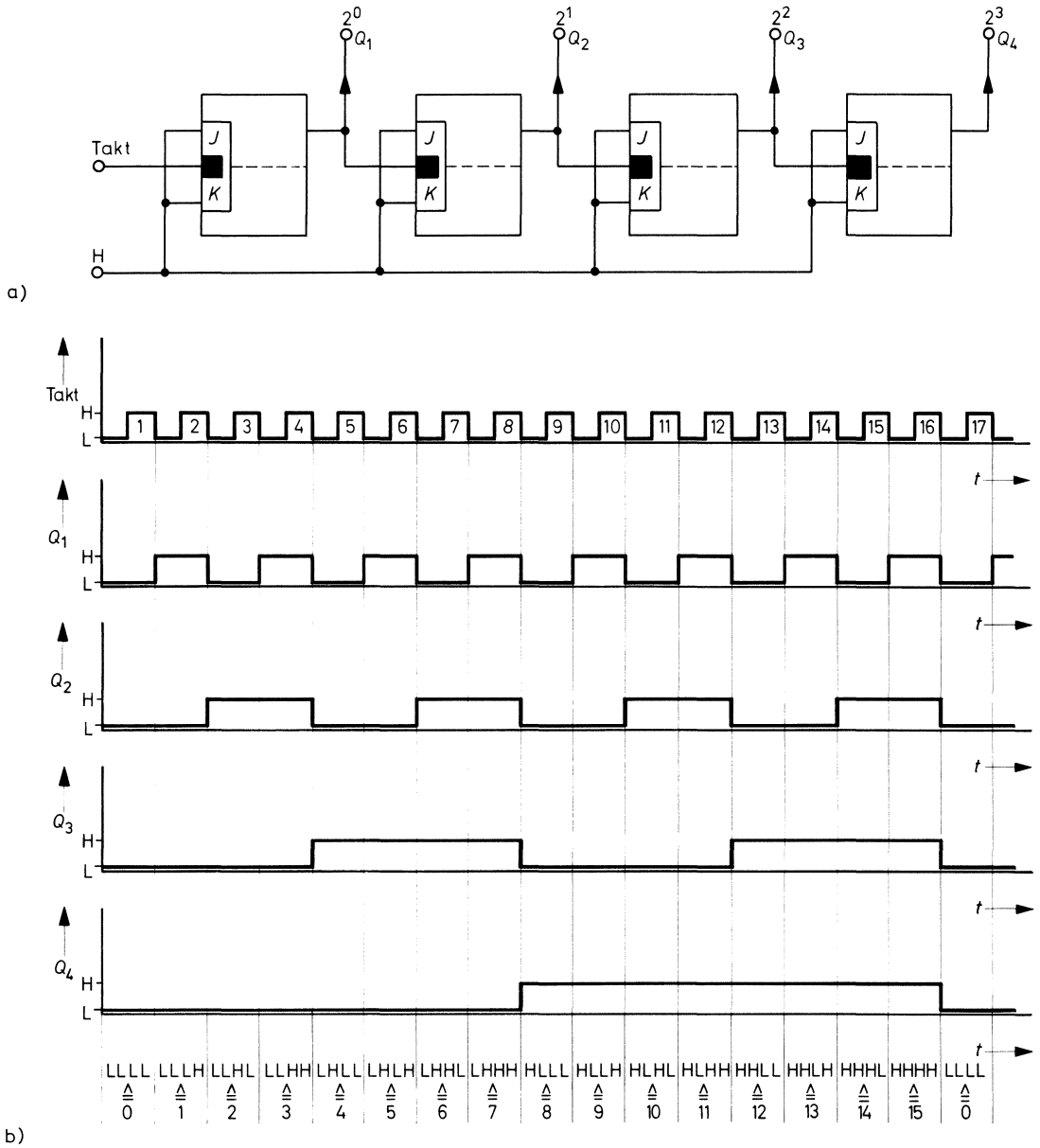


Bild 1.8.1
4-bit-Binärzähler
a) Schaltung mit JK-Master-Slave-FFs
b) Impulsdiagramm

– Zählgeschwindigkeit
– Vorprogrammierbarkeit

Unter Vorprogrammierbarkeit versteht man, daß ein Zähler über eine Logik auf eine bestimmte Zahl gesetzt werden kann, von der aus dann der Zählvorgang wie vorher beschrieben abläuft (Laden eines Zählers mit einer bestimmten Zahl). Es würde den Rahmen dieses Lehrganges sprengen, auf alle Kriterien näher einzugehen.

2. Rechnerarithmetik

2.1 Zahlensysteme

Das Dezimal- oder Zehnersystem ist sicher das am meisten benutzte und uns allen am besten bekannte Zahlensystem. Es baut auf der Zahl 10 auf. 10 ist die Basis des Dezimalsystems, die Ziffern 0 bis 9, also insgesamt 10 Zeichen, werden zur Darstellung benutzt.

Ein Beispiel einer Dezimalzahl ist:

$$2\,305{,}51 = 2 \cdot 10^3 + 3 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0 + 5 \cdot 10^{-1} + 1 \cdot 10^{-2}$$

In einem Digitalrechner müssen wir die Zahlen durch bestimmte Werte physikalischer Größen darstellen, z. B. durch Spannung, Strom, Magnetisierung usw. Wir könnten beispielsweise der Spannung 1 V die Zahl 1, der Spannung 2 V die Zahl 2 usw. zuordnen und hätten so im Bereich 0 bis 9 V die 10 Ziffern des Dezimalsystems dargestellt. Eine solche Zuordnung ist technisch durchaus möglich, aber kompliziert und teuer. Viel einfacher ist es, wenn man nur 2 Zustände einer bestimmten physikalischen Größe unterscheidet und diesen beiden Zuständen dann 2 Zahlen, nämlich 0 und 1, zuordnet. Z. B. auf die folgende Art:

0 ≙ kein Strom, keine Spannung, Magnetisierung im Uhrzeigersinn, Logikzustand L
1 ≙ Spannung, Strom vorhanden, Magnetisierung entgegen dem Uhrzeigersinn, Logikzustand H

Moderne Digitalrechner arbeiten ausnahmslos nach diesem Prinzip. Die interne Darstellung von Zahlen basiert somit auf einem Zahlensystem, das mit den beiden Zahlensymbolen 0 und 1 auskommt, dem sog. binären Zahlensystem. Für diese Binärzahlen gelten sinngemäß dieselben Regeln wie für die Dezimalzahlen.
Eine Binärzahl ist z. B. folgendermaßen zu verstehen:

$$1\,0\,1\,1{,}0\,1 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$$

Auch das binäre und das dezimale Zählen unterscheiden sich vom Prinzip her nicht. Wenn beim Zählen der Zahlenvorrat einer Stelle überschritten wird, ergibt sich ein Übertrag in die nächste Stelle; im Binärsystem, wenn beim Zählen die Zahl 1 in einer Stelle überschritten wird, im Dezimalsystem, wenn die Zahl 9 in einer Stelle überschritten wird:

Binäre Zahlenreihe	Dezimale Zahlenreihe
0	0
1	1
1 0	2
1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
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
.	.
.	.
.	.

Wenn verschiedene Zahlensysteme nebeneinander verwendet werden, so wird im allgemeinen zur Unterscheidung die Basis des Zahlensystems als Index angeschrieben:

$1\,001_{10}$ bedeutet die Dezimalzahl $1 \cdot 10^3 + 0 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0$
$1\,0\,0\,1_2$ bedeutet die Binärzahl $1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Eine Zahl im Binärsystem benötigt ungefähr 3mal so viele Stellen wie dieselbe Zahl im Dezimalsystem, wie nachfolgendes Beispiel zeigt:

$$2\,049_{10} \triangleq 1\,0\,0 \quad 0\,0\,0 \quad 0\,0\,0 \quad 0\,0\,1_2$$

Binäre Zahlen werden deshalb durch ihre Länge sehr schnell unhandlich. Beim praktischen Arbeiten mit Rechnern und binären Zahlen wurde deshalb eingeführt, die Stellen von binären Zahlen in Gruppen von 3 bzw. 4 zusammenzufassen. Man kommt dadurch zu dem Oktal- bzw. dem Hexadezimalsystem. Beim Oktalsystem verwendet man folgende Zuordnung:

Binär	Oktal
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

Im Oktalsystem gibt es somit $2^3 = 8$ verschiedene Zahlensymbole, nämlich die Ziffern 0 bis 7. Es hat den Vorteil, ebenso wie das Dezimalsystem, zu Zahlen handlicher Größe zu führen, andererseits aber ist die Übersetzung binär → oktal sehr einfach. Der Programmierer muß lediglich die Binärzahlen des Rechners in Dreiergruppen zusammenfassen, um dann die einzelnen Gruppen in Oktalstellen umzuwandeln:

Binär	1 0 1	1 1 0	0 1 1	1 1 1	,	1 0 1	0 1 0
Oktal	5	6	3	7	,	5	2

$$1\,0\,1\;\;1\,1\,0\;\;0\,1\,1\;\;1\,1\,1,1\,0\,1\;\;0\,1\,0_2 = 5\,6\,3\,7,5\,2_8$$

Im Hexadezimalsystem werden die Binärzahlen zu Vierergruppen zusammengefaßt. Man benötigt also $2^4 = 16$ verschiedene Zahlensymbole, die den Binärzahlen 0 0 0 0 bis 1 1 1 1 zugeordnet werden. Es ist üblich, dafür die Ziffern 0 bis 9 und die Buchstaben A bis F auf die folgende Art und Weise zu verwenden:

Binär	Hexadezimal
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	A
1 0 1 1	B
1 1 0 0	C
1 1 0 1	D
1 1 1 0	E
1 1 1 1	F

Ein Beispiel einer Hexadezimal → Binärumwandlung ist:

Binär:	1 1 0 1	1 1 1 1	0 0 1 1	0 1 0 1
Hexadezimal:	D	F	3	5

also: $1\,1\,0\,1\;\;1\,1\,1\,1\;\;0\,0\,1\,1\;\;0\,1\,0\,1_2 = D\,F\,3,5_{16}$

In der Programmierpraxis und der Literatur über Mikroprozessoren werden alle 4 bisher behandelten Zahlensysteme nebeneinander verwendet, manche Hersteller bzw. Autoren entscheiden sich für bestimmte Systeme, andere verwenden von Fall zu Fall das jeweils am besten geeignete Zahlensystem.

2.2 Umwandlungen zwischen Zahlensystemen

Wenn verschiedene Zahlensysteme nebeneinander verwendet werden, müssen zwangsläufig Zahlen von einem System in ein anderes umgerechnet werden. Dies wurde für Binär-, Oktal-, Hexadezimalzahlen bereits gezeigt. Es gibt allgemeine Methoden, die sich für Umwandlungen zwischen Zahlensystemen mit beliebigen Basen anwenden lassen, die wichtigste Anwendung ist in der Praxis aber stets die Umwandlung vom oder ins Dezimalsystem. Deshalb soll im folgenden diese Aufgabe genauer behandelt werden.

2.2.1 Divisionsmethode

Dieses Verfahren ist besonders geeignet um positive ganze Dezimalzahlen in beliebige andere Zahlensysteme umzurechnen. Das Verfahren soll am Beispiel einer Umwandlung einer Dezimalzahl in eine Binärzahl praktisch durchgeführt werden. Die Dezimalzahl sei 457_{10}. Diese Zahl wird durch die gewünschte Basis, im Falle des Binärsystems durch 2, dividiert:

$$457 : 2 = 228 \text{ Rest } 1$$

Dieser Rest 1 gibt die erste ganz rechts stehende Binärstelle an:

$$\underbrace{x\ x\ x\ x\ x}_{\text{noch nicht bestimmt}}\ 1$$

Die nächste Stelle wird durch erneute Division durch 2 bestimmt. Dividiert wird der Quotient der 1. Division, also:

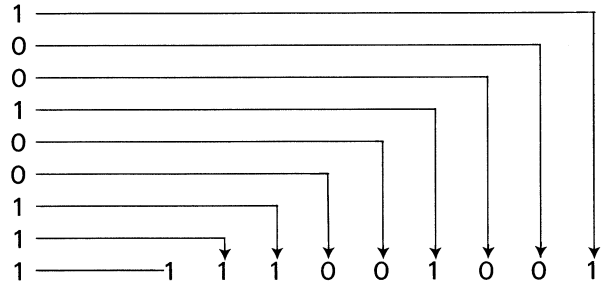
$$228 : 2 = 114 \text{ Rest } 0$$

Die zweite Binärstelle ist damit als 0 bestimmt. Die Binärzahl lautet also bis jetzt x x x x 0 1. Dieser Vorgang wird so lange fortgesetzt, bis sich ein Quotient von 0 ergibt, wie nachfolgendes Beispiel zeigt:

$$N_{10} = 457$$

Quotient	Rest
457 : 2 = 228	1
228 : 2 = 114	0
114 : 2 = 57	0
57 : 2 = 28	1
28 : 2 = 14	0
14 : 2 = 7	0
7 : 2 = 3	1
3 : 2 = 1	1
1 : 2 = 0	1

$457_{10} = 1\ 1\ 1\ \ 0\ 0\ 1\ \ 0\ 0\ 1_2$



Vertiefungsstoff:

Allgemeine Herleitung des Divisionsverfahrens.

Es soll die Dezimalzahl N_{10} in das Zahlensystem der Basis b umgewandelt werden, also in eine Zahl der Form:

$$a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \ldots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0$$

Dabei sind die Koeffizienten a_i zu bestimmen. Wir haben also folgende Gleichung:

$$N_{10} = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \ldots + a_1 \cdot b + a_0$$

Wenn wir diese Gleichung auf beiden Seiten durch b dividieren, bekommen wir folgenden Ausdruck:

$$\frac{N_{10}}{b} = \underbrace{a_k \cdot b^{k-1} + a_{k-1} \cdot b^{k-2} + \ldots + a_1}_{\text{positive ganze Zahl}} + \underbrace{\frac{a_0}{b}}_{\frac{\text{Divisionsrest}}{b}} = Q + \frac{R}{b}$$

Bei der Division der positiven ganzen Zahl N_{10} durch b erhalten wir im allgemeinen einen ganzzahligen Quotienten Q und einen Rest R. Den Ausdruck auf der rechten Seite können wir aufspalten in die Summe

$$a_k \cdot b^{k-1} + a_{k-1} \cdot b^{k-2} + \ldots + a_1$$

die eine positive ganze Zahl darstellt und den echten Bruch:

$$\frac{a_0}{b}$$

Wir haben also folgende Zuordnung:

ganzzahliger Anteil $\quad Q = a_k \cdot b^{k-1} + a_{k-1} \cdot b^{k-2} + \ldots + a_1$

gebrochener Anteil $\quad \frac{R}{b} = \frac{a_0}{b}$ oder $R = a_0$

Der Rest der 1. Division gibt also das gesuchte a_0. Den Koeffizienten a_1 bestimmen wir entsprechend durch Division von Q durch b:

$$\underbrace{\frac{Q}{b} = a_k \cdot b^{k-2} + a_{k-1} \cdot b^{k-3} + \ldots + a_2}_{\text{positive ganze Zahl}} + \underbrace{\frac{a_1}{b}}_{\frac{\text{Rest}}{b}}$$

a_1 ist gleich dem Rest bei der 2. Division usw., bis alle Koeffizienten bestimmt sind.

Ende Vertiefungsstoff.

2.2.2 Multiplikationsmethode

Dieses Verfahren wird verwendet, um echte Dezimalbrüche, also Zahlen der Form 0,xxx . . ., in andere Zahlensysteme umzuwandeln. Das Verfahren soll am Beispiel der Umwandlung des echten Dezimalbruches $0{,}5625_{10}$ in einen binären Bruch praktisch durchgeführt werden.

Der Dezimalbruch wird mit der gewünschten Basis, im Falle des Binärsystems also mit 2, multipliziert:

$$\frac{0{,}5625 \cdot 2}{1{,}1250}$$

Die bei dem Produkt vor dem Komma stehende Zahl, in unserem Beispiel also eine 1, ergibt die erste Stelle des binären Bruches nach dem Komma, also:

$$0,\ 1\ \underbrace{x\ x\ x\ \ldots}_{\text{noch nicht bestimmt}}$$

Im nächsten Schritt wird der gebrochene Anteil des vorigen Produktes, in unserem Beispiel also 0,1250, mit 2 multipliziert:

$$\underline{0{,}1250 \cdot 2}$$
$$0{,}2500$$

Die Zahl vor dem Komma ergibt die nächste Binärstelle also 0. Die Binärzahl lautet bis jetzt:

$$0{,}1\ 0\ x\ x\ x$$

Diese Multiplikationen werden so lange fortgesetzt, und bei jedem Schritt wird eine binäre Stelle bestimmt, bis sich bei den letzten Multiplikationen kein gebrochener Anteil mehr ergibt. Damit ist die Umwandlung abgeschlossen, wie nachfolgendes Beispiel noch näher erläutert:

$$N_{10} = 0{,}5625$$

$$\underline{0{,}5625 \cdot 2}$$
$$1{,}1250$$

$$\underline{0{,}1250 \cdot 2}$$
$$0{,}2500$$

$$\underline{0{,}2500 \cdot 2}$$
$$0{,}5000$$

$$\underline{0{,}5000 \cdot 2}$$
$$1{,}0000$$

gebrochener Anteil = 0

$0{,}1\ 0\ \ 0\ \ 1$

$$0{,}5625_{10} = 0{,}1\ 0\ 0\ 1_2$$

Vertiefungsstoff:
Allgemeine Herleitung der Multiplikationsmethode.
Der dezimale Bruch F_{10} soll in einem Bruch im Zahlensystem der Basis b, also eine Zahl der Form

$$a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + \ldots + a_{-n} \cdot b^{-n}$$

umgewandelt werden. Es gilt die Gleichung:

$$F_{10} = a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + \ldots + a_{-n} \cdot b^{-n}$$

Zu bestimmen sind die Koeffizienten a_i. Multipliziert man diese Gleichung links und rechts mit b, so erhält man folgenden Ausdruck:

$$b \cdot F_{10} = \underbrace{a_{-1}}_{\text{ganze Zahl}} + \underbrace{a_{-2} \cdot b^{-1} + a_{-3} \cdot b^{-2} + \ldots + a_{-n} \cdot b^{-n+1}}_{\text{echter Bruch}}$$

Bei der Multiplikation des echten Dezimalbruches F_{10} mit b erhält man im allgemeinen ein Produkt, das sich aus einem ganzzahligen und einem gebrochenen Anteil zusammensetzt.

Auf der rechten Seite der Gleichung erhält man ebenfalls einen ganzzahligen Anteil a_{-1} und einen gebrochenen Anteil:

$$a_{-2} \cdot b^{-1} + a_{-3} \cdot b^{-2} + \ldots + a_{-n} \cdot b^{-n+1}$$

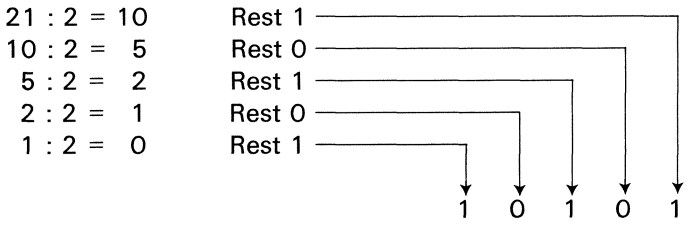
Der Ausdruck a_{-1} ist also gleich dem ganzzahligen Anteil des Produktes $b \cdot F_{10}$. Entsprechend werden die anderen Koeffizienten durch sukzessive Multiplikation der gebrochenen Anteile mit b bestimmt.

Ende Vertiefungsstoff.

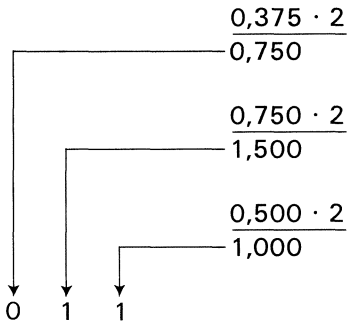
2.2.3 Umwandlung rationaler Zahlen

Rationale Zahlen werden in den ganzzahligen und den gebrochenen Anteil zerlegt. Diese können nun mit den beschriebenen Verfahren getrennt umgewandelt und dann die Umwandlungsergebnisse wieder zusammengefügt werden. Z. B. die Umwandlung der Zahl $21{,}375_{10}$ in eine Binärzahl wird folgendermaßen durchgeführt:

Ganzzahliger Anteil: 21
Umwandlung mit der Divisionsmethode:



Gebrochener Anteil: 0,375
Umwandlung mit der Produktmethode:



Wir erhalten also:

$$21{,}375_{10} = 1\,0\,1\,0\,1,0\,1\,1_2$$

2.2.4 Umwandlung ins Dezimalsystem

Beispiel: Umwandlung der Binärzahl $1\,1\,0\,0\,1_2$ in eine Dezimalzahl. Es gilt:

$$1\,1\,0\,0\,1_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Durch Summation ergibt sich:

$$1\,1\,0\,0\,1_2 = 16_{10} + 8_{10} + 1_{10} = 25_{10}$$

Allgemein gilt bei einer beliebigen Basis b:

$$N_b = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \ldots + a_2 \cdot b^2 + a_1 \cdot b + a_0$$

Die Auswertung der Summe ergibt die gesuchte Dezimalzahl.

Fragen zu den Abschnitten 2.1 und 2.2

1. Wandeln Sie nachfolgende Binärzahlen in Oktalzahlen um!
   a) $1\,0\,1\;\;0\,1\,1\;\;1\,1\,1\;\;0\,0\,1_2$ =
   b) $1\,1\,0\;\;0\,1\,0\;\;1\,1\,0\;\;1\,1\,1_2$ =

2. Wandeln Sie die nachfolgenden Oktalzahlen in Binärzahlen um!
   a) $2\,1\,7\,0_8$ =
   b) $3\,5\,7\,1_8$ =

3. Führen Sie für folgende Zahlen die Umwandlung binär → hexadezimal und umgekehrt durch!
   a) $1\,1\,0\,1\;\;1\,1\,1\,1\;\;0\,0\,1\,0\;\;1\,1\,1\,0_2$ =
   b) $0\,1\,1\,0\;\;1\,0\,0\,1\;\;1\,0\,0\,0\;\;0\,0\,1\,0_2$ =
   c) $0\,0\,1\,1\;\;1\,1\,0\,0\;\;0\,1\,1\,1\;\;1\,1\,0\,1_2$ =
   d) $A\,B\,C\,D_{16}$ =
   e) $2\,1\,7\,0_{16}$ =
   f) $B\,7\,5\,F_{16}$ =

4. Wandeln Sie folgende Dezimalzahlen in Binärzahlen um!
   a) 1234_{10} =
   b) 5670_{10} =
   c) 2321_{10} =

5. Wandeln Sie um von dezimal in oktal!
   a) 2115_{10} =
   b) 4321_{10} =
   c) 7688_{10} =
   d) 3821_{10} =

6. Wandeln Sie um von dezimal in hexadezimal!
   a) 1780_{10} =
   b) 3666_{10} =
   c) 5230_{10} =
   d) 6744_{10} =

7. Wandeln Sie folgende Dezimalbrüche in Binärzahlen um!
   a) $0{,}3125_{10}$ =
   b) $0{,}656\;25_{10}$ =
   c) $0{,}343\;75_{10}$ =
   d) $0{,}140\;625_{10}$ =

8. Wandeln Sie folgende Brüche von dezimal in oktal um!
   a) $0{,}494\;14_{10}$ =
   b) $0{,}406\;25_{10}$ =
   c) $0{,}451_{10}$ =
   d) $0{,}121_{10}$ =

9. Wandeln Sie folgende Brüche von dezimal in hexadezimal um!
   a) $0{,}301_{10}$ =
   b) $0{,}8213_{10}$ =
   c) $0{,}022_{10}$ =

10. Wandeln Sie folgende Zahlen in Dezimalzahlen um!
    a) $1\,0\,1,0\,1_2$ =
    b) $7\,2\,3,1\,4_8$ =
    c) $A\,1,5\,E_{16}$ =

2.3 Rechnen mit binären Zahlen

In allen Zahlensystemen gelten sinngemäß dieselben Rechenregeln, es ist also möglich, die vom Rechnen mit Dezimalzahlen gewohnten Verfahren auf das Rechnen mit Zahlen in anderen Zahlensystemen zu übertragen. In diesem Abschnitt sollen die 4 Grundrechnungsarten im Binärsystem sowie die logischen Verknüpfungen binärer Zahlen betrachtet werden.

Addition

Bei der Addition binärer Zahlen erhält man immer dann einen Übertrag (carry) von einer Stelle in die nächste, wenn die Summe in einer Spalte gleich oder größer als 2 wird.

```
  1 0 0 1 0 1
+   1 0 1 1 1
      1 1 1        Übertrag
-----------
  1 1 1 1 0 0
```

Subtraktion

Bekanntlich wird bei dieser Rechenart vom Minuenden *M* der Subtrahend *S* abgezogen und das Ergebnis ist die Differenz *D*. Dies macht auch im Binärsystem keine Schwierigkeiten, wenn pro Stelle der Minuend gleich oder größer als der Subtrahend ist. Ist dies nicht der Fall, muß von der nächstwerthöheren Stelle eine **Entlehnung** erfolgen. Hierzu ein einfaches Beispiel:

2^4	2^3	2^2	2^1	2^0	Stellenwertigkeit
1	0	0	1	0	
0	1	1	0	1	
1	1	0	1		Entlehnung
0	0	1	0	1	

In der 2^0-Stelle muß von 0 eine 1 abgezogen werden. Aufgeschlüsselt bedeutet dies $0 \cdot 2^0 - 1 \cdot 2^0$. Dies ist nur möglich, wenn von der 2^1-Stelle eine Entlehnung erfolgt. Gedanklich wird also die 1 der 2^1-Stelle eine Stelle nach rechts gerückt, und wir erhalten aufgeschlüsselt die Rechenoperation $1 \cdot 2^1 - 1 \cdot 2^0 = 2 - 1 = 1$. Damit erscheint als Differenz in dieser Spalte eine 1. Die erfolgte Entlehnung wird in der 2^1-Stelle durch eine 1 in der Entlehnungszeile ausgedrückt. In dieser Stelle lautet jetzt der Rechengang $1 - 0 - 1 = 0$. In der 2^2-Stelle wiederholt sich der beschriebene Vorgang. In der 2^3-Stelle sind von 0 einmal 1 und von diesem Ergebnis noch einmal 1 durch die Entlehnung zu subtrahieren. Wir lassen zunächst die Entlehnung außer acht und rechnen $0 - 1 = 1$ mit 1 als Entlehnung aus der 2^4-Stelle. Von diesem Zwischenergebnis wird die Entlehnung subtrahiert, und wir erhalten als Ergebnis 0. In der 2^4-Stelle ergeben Minuend und Entlehnung die Differenz 0.

Multiplikation

Die binäre Multiplikation wird nach denselben Regeln durchgeführt wie die dezimale Multiplikation. Man bildet die Produkte mit den einzelnen Stellen des Multiplikators und summiert stellenrichtig auf:

```
1 0 0 1 · 1 1 0 1
  1 0 0 1
    1 0 0 1
      0 0 0 0
        1 0 0 1
-----------------
1 1 1 0 1 0 1
```

Hierbei ergibt sich eine Vereinfachung gegenüber einer dezimalen Multiplikation. Da die Stellen des Multiplikators nur die Zahlenwerte 0 oder 1 annehmen können, muß der Multiplikand nur mit 0 oder 1 multipliziert werden, d. h., bei der binären Multiplikation kommen als Teiloperationen nur die Addition und die Verschiebung vor.

Division

Die einzelnen Operationen bei einer binären Division sind Subtraktion und Verschiebung. Der Algorithmus (Rechenvorschrift) ist derselbe wie bei der dezimalen Division:

```
 1 1 1 0 1 0 1 : 1 0 0 1 = 1 1 0 1
-1 0 0 1
 0 1 0 1 1
-1 0 0 1
  0 0 1 0 0
 -0 0 0 0
    1 0 0 1
   -1 0 0 1
    0 0 0 0
```

2.3.1 Logische Verknüpfung zweier binärer Zahlen *A* und *B*

Hierunter versteht man in der Rechnertechnik eine bit-weise Verknüpfung von zwei Binärzahlen. Bit-weise Verknüpfung bedeutet, daß nur jeweils die gleichen Binärstellen durch die entsprechende Logikfunktion miteinander verknüpft werden. Hierzu nachfolgende Beispiele:

UND

A	1 1 0 1 1 0 1 0
B	1 0 1 0 0 1 1 0
$A \wedge B$	1 0 0 0 0 0 1 0

ODER

A	1 1 0 1 1 0 1 0
B	1 0 1 0 0 1 1 0
$A \vee B$	1 1 1 1 1 1 1 0

EXKLUSIV-ODER

A	1 1 0 1 1 0 1 0
B	1 0 1 0 0 1 1 0
$A \veebar B$	0 1 1 1 1 1 0 0

Unter dem logischen Komplement $\bar{A}$ oder dem Einerkomplement der Zahl A versteht man die Komplementierung der einzelnen binären Stellen, d. h., aus jeder 0 wird eine 1, aus jeder 1 wird eine 0.

A	1 1 0 1 1 0 1 0
$\bar{A}$	0 0 1 0 0 1 0 1

Fragen zu Abschnitt 2.3

1. Addieren Sie folgende Binärzahlen!

   a)
   ```
     0 1 0 1 1 0 1 1
   + 0 1 1 0 1 0 1 1
   ```

   b)
   ```
       1 0 1 1
   +   0 0 1 1
   ```

c) 1 1 1 1 1 1 1 1
+ 0 0 0 0 0 0 0 1

d) 1 1 0 1 1 1 0 0
+ 1 0 1 1 1 0 0 1

2. Subtrahieren Sie folgende Binärzahlen!

a) 1 0 1 1
− 0 0 1 1

b) 1 1 0 1 1 0 1 1
− 0 1 1 0 1 0 1 1

c) 1 1 0 0 0 0 0 0
− 1 0 1 1 0 1 0 1

d) 1 1 0 1 1 1 0 0
− 1 0 1 1 1 0 0 1

3. Multiplizieren Sie nachfolgende Binärzahlen!

a) 1 1 0 0 1 0 0 · 1 0 1 =
b) 1 1 0 0 1 · 1 0 0 0 1 =
c) 1 0 1 0 0 · 1 0 1 0 0 =
d) 1 1 1 0 1 0 1 · 1 1 0 0 0 1 1 =

4. Dividieren Sie nachfolgende Binärzahlen!

a) 1 1 1 0 1 0 0 : 1 0 0 =
b) 1 1 1 1 1 0 1 1 1 : 1 0 1 =
c) 1 1 0 1 0 1 0 1 1 : 1 0 0 1 =

5. Führen Sie die UND-Verknüpfung für nachfolgende Ausdrücke durch!

a) 0 1 0 1 1 0 1 1
0 1 1 0 1 0 1 1

b) 1 0 1 1
0 0 1 1

c) 1 0 1 0 1 1 0 1
0 0 0 0 1 1 1 1

6. Für nachfolgende Ausdrücke ist die ODER-Verknüpfung durchzuführen!

a) 0 1 0 1 1 0 1 1
0 1 1 0 1 0 1 1

b) 1 0 1 1
0 0 1 1

c) 1 0 1 0 1 1 0 1
0 0 0 0 1 1 1 1

7. Führen Sie für nachfolgende Ausdrücke die EXKLUSIV-ODER-Verknüpfung durch!

a) 0 1 0 1 1 0 1 1
0 1 1 0 1 0 1 1

b) 1 0 1 1
0 0 1 1

c) 1 0 1 0 1 1 0 1
1 1 1 1 1 1 1 1

2.4 Arithmetik mit positiven ganzen Zahlen (Integer Arithmetic)

Wir haben bis jetzt nur die Zahlen mit positiven Vorzeichen betrachtet. Außerdem haben wir bis jetzt nicht berücksichtigt, daß jeder Rechner eine bestimmte Wortlänge hat, d. h., daß er nur binäre Zahlen mit einer bestimmten Anzahl von Stellen darstellen kann. Ein 8-bit-Mikroprozessor z. B. kann die Zahlen von 0 0 0 0 0 0 0 0 bis 1 1 1 1 1 1 1 1, also von 0 bis $2^8 - 1 = 255$, verarbeiten. Wenn dieser Zahlenvorrat nicht ausreicht, muß das bei der Programmierung berücksichtigt werden. Für folgende Betrachtungen wollen wir annehmen, wir hätten einen Rechner mit einer Wortlänge von 4 bit, d. h., der Zahlenvorrat geht von 0 0 0 0 bis 1 1 1 1, also von 0 bis $2^4 - 1 = 15$.
Man kann die arithmetischen Operationen anhand des Zahlenstrahles grafisch darstellen (Bild 2.4.1).

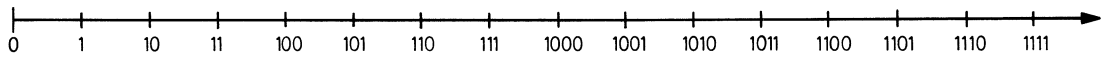


Bild 2.4.1
Zahlenstrahl

Jeder positiven Binärzahl entspricht ein Punkt auf diesem Zahlenstrahl, man kann z. B. die Addition zweier Binärzahlen zurückführen auf die Addition von 2 Strecken auf dem Zahlenstrahl.
Die Operationen in dem 4-bit-Rechner kann man grafisch verdeutlichen durch einen Zahlenkreis (Bild 2.4.2).

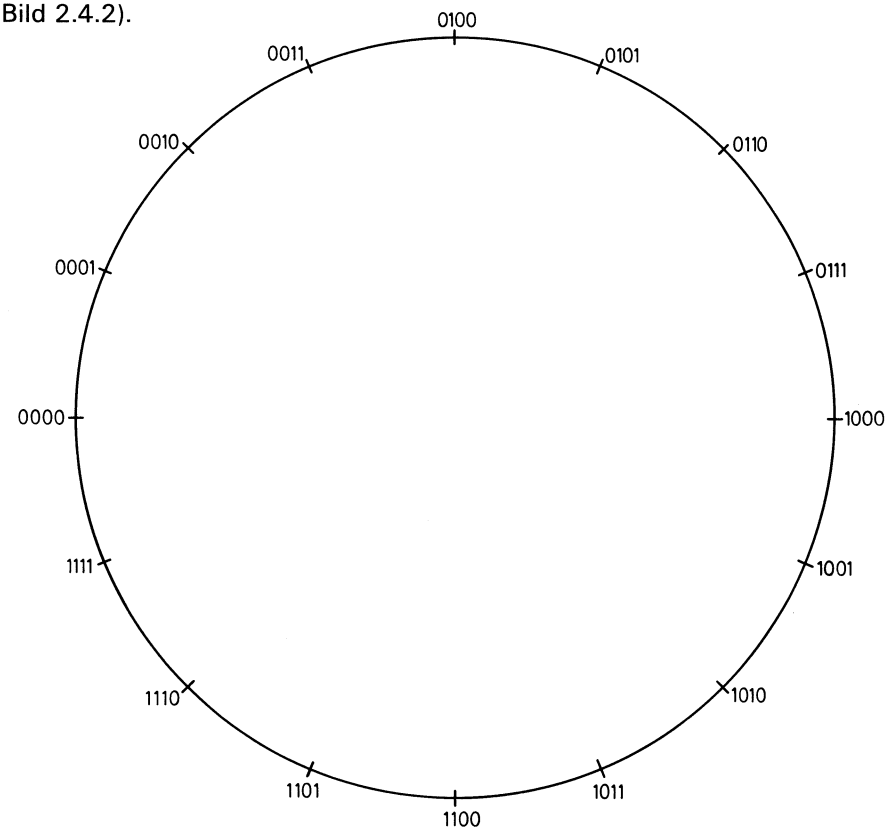


Bild 2.4.2
Darstellung positiver ganzer Zahlen in einem 4-bit-Register anhand eines Zahlenkreises

Die Addition zweier Binärzahlen entspricht hier der Addition zweier Kreisbögen. Solange man den zugelassenen Zahlenvorrat von 0 0 0 0 bis 1 1 1 1 nicht überschreitet, sind die Verhältnisse gleich wie beim Zahlenstrahl. Addiert man jedoch z. B.

$$\begin{array}{r} 1\,1\,1\,1 \\ +\,0\,0\,1\,0 \\ \hline 1\,0\,0\,0\,1 \end{array}$$

so sieht man, daß die Summe 5 bit benötigt, da sich in der vierten Stelle ein **Übertrag** (carry) in die fünfte Stelle ergibt. Diese Zahl paßt jedoch nicht in einen 4-bit-Rechner, man erhält vielmehr ein falsches Ergebnis, nämlich 0 0 0 1. Dieses Ergebnis ist auch auf dem Zahlenkreis abzulesen. Das ist der Fall des sog. **Überlaufes**. Um beim Arbeiten mit dem Rechner diesen Fall erkennen und gegebenenfalls korrigieren zu können, geht der Übertrag in die fünfte Stelle nicht verloren, er wird vielmehr in einem Flipflop gespeichert, dem sog. Carry-FF, kurz Carry-Flag genannt (engl.: flag = Flagge, Signal, Zeichen). Das Carry-Flag kann über ein Programm nach einer arithmetischen Operation geprüft werden, so daß im Falle eines Überlaufes die entsprechenden Maßnahmen ergriffen werden können.
Diese Betrachtungen an dem 4-bit-Beispiel gelten sinngemäß für Rechner mit beliebiger Wortlänge. Beim Rechnen mit positiven Zahlen wird ein Überlauf angezeigt durch einen Übertrag von der werthöchsten binären Stelle. Dieser Übertrag wird in einem Flag gespeichert.

2.5 Zweierkomplementarithmetik (Two's Complement Arithmetic)

Positive und negative Zahlen werden auf der Zahlengeraden folgendermaßen dargestellt (Bild 2.5.1).

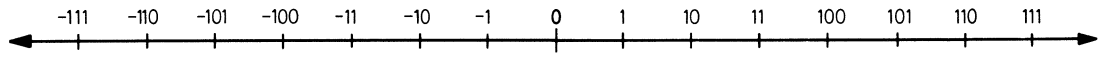


Bild 2.5.1
Darstellung von positiven und negativen Zahlen auf der Zahlengeraden

Bevor wir diese Zahlendarstellung auf den Zahlenkreis übertragen, sollen zuerst einige grundlegende Betrachtungen über negative Zahlen angestellt werden. Es gibt mehrere Möglichkeiten, eine negative Binärzahl darzustellen. Im Hinblick auf den Mikroprozessor soll hier nur die **Zweierkomplementdarstellung** näher erläutert werden. Allgemein versteht man unter Komplement die Ergänzung zur größten gleichstelligen Zahl. Auf das Dezimalsystem übertragen bedeutet dies, daß z. B. das Komplement $\overline{A}$ der Zahl A = 1977 sich ergibt zu:

$$\overline{A} = 9999 - 1977 = 8022$$

Auf Binärzahlen angewandt bedeutet dies, daß das Komplement $\overline{A}$ der Invertierung der Binärzahl entspricht. Beispiel:

$$A = 0\,1\,1\,0\,0\,1$$
$$\overline{A} = 1\,0\,0\,1\,1\,0$$

Hierbei erfolgt also immer eine Ergänzung zu 1. Aus diesem Grunde wird $\overline{A}$ bei Binärzahlen auch als **Einerkomplement** bezeichnet. Addiert man eine Zahl mit ihrem Einerkomplement, so hat das Ergebnis immer die Form 1 1 1 . . . 1 1. Addiert man zu diesem Ergebnis eine 1, so entsteht als neues Ergebnis eine Binärzahl mit der Form **1** 0 0 0 . . . 0 0. Abgesehen von der 1 in der werthöchsten Stelle, die bei einer n-bit-Zahl den Übertrag in die n+1-Stelle darstellt, ist für die n bit das Ergebnis 0. Abgesehen von diesem Übertrag gilt also:

$$A + (\overline{A} + 1) = 0$$

bzw.

$$\overline{A} + 1 = -A$$

Damit ist $\overline{A}$ + 1 eine Darstellung für $-A$. Der Ausdruck $-A$ wird als **Zweierkomplement** bezeichnet. Grundsätzlich gilt: Die Addition einer Binärzahl A mit ihrem Zweierkomplement $-A$ ergibt immer das Ergebnis Null.
Folgendes Beispiel verdeutlicht die Bildung des Zweierkomplementes:

A:	0 1 0 1 1 0 1 0	
$\overline{A}$:	1 0 1 0 0 1 0 1	(Einerkomplement)
+1:	1	
$-A = \overline{A} + 1$:	1 0 1 0 0 1 1 0	(Zweierkomplement)

Wenn wir nun $-A$ mit A addieren, erhalten wir folgendes Ergebnis:

$$\begin{array}{rr} A: & 0\,1\,0\,1\,1\,0\,1\,0 \\ -A: & +1\,0\,1\,0\,0\,1\,1\,0 \\ \hline A+(-A): & \underline{1}\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

Wenn Sie weitere Beispiele nach diesem Schema durchrechnen, werden Sie feststellen, daß als Ergebnis immer eine Zahl der Form $\underline{1}\,0\,0\,0\,\ldots\,0\,0$ entsteht.
Wenn nun eine negative Binärzahl über das Zweierkomplement dargestellt werden kann, ist es möglich eine Subtraktion von Binärzahlen auf eine Addition zurückzuführen. Es ist bekannt, daß z. B. die Subtraktionsaufgabe $9 - 3 = 6$ auch als $9 + (-3) = 6$ geschrieben werden kann. Beziehen wir dieses Beispiel auf Binärzahlen, so erhalten wir:

$$\begin{array}{rl} 1\,0\,0\,1 & \triangleq 9 \\ +\,1\,1\,0\,1 & \text{(Zweierkomplement von } 0\,0\,1\,1 \triangleq 3) \\ \hline \underline{1}\,0\,1\,1\,0 & \triangleq 6 \end{array}$$

Vom Übertrag 1 abgesehen, erhalten wir als Ergebnis $0\,1\,1\,0 = 6$, also das richtige Ergebnis.
Als nächstes Beispiel wird eine Aufgabe nach dieser Methode gerechnet, die ein negatives Ergebnis hat:

$0\,1\,1\,0 - 1\,0\,0\,1 = \quad \triangleq 6 - 9 = -3$

$$\begin{array}{rl} 0\,1\,1\,0 & \\ +\,0\,1\,1\,1 & \text{(Zweierkomplement von } 1\,0\,0\,1) \\ \hline 1\,1\,0\,1 & \triangleq 13 \end{array}$$

Dieses Ergebnis ist offensichtlich falsch. Dem **Betrag** nach muß bei dieser Aufgabe als Ergebnis $0\,0\,1\,1$ herauskommen. Wenn wir von $0\,0\,1\,1$ das Zweierkomplement bilden, so erhalten wir:

$$\begin{array}{rrl} A: & 0\,0\,1\,1 & \\ \overline{A}: & 1\,1\,0\,0 & \\ & +\,1 & \\ \hline -A: & 1\,1\,0\,1 & \text{(Zweierkomplement von } 0\,0\,1\,1) \end{array}$$

Aus diesem Beispiel geht hervor, daß das erste Ergebnis dem Zweierkomplement der tatsächlichen Lösung entspricht. Da aber nach der Beziehung $\overline{A} + 1 = -A$ durch das Zweierkomplement ein negatives Ergebnis ausgedrückt wird, ist die Lösung $1\,1\,0\,1$ richtig. Wesentlich ist hierbei, daß dieses Ergebnis als Zweierkomplement interpretiert wird. Bilden wir einmal die Zweierkomplemente der Zahlen $0\,0\,0\,1$ bis $0\,1\,1\,1$ (1 bis 7):

A	$-A$
0 0 0 1	1 1 1 1
0 0 1 0	1 1 1 0
0 0 1 1	1 1 0 1
0 1 0 0	1 1 0 0
0 1 0 1	1 0 1 1
0 1 1 0	1 0 1 0
0 1 1 1	1 0 0 1

Die linke Reihe bildet die positiven Zahlen von $0\,0\,0\,1$ bis $0\,1\,1\,1$, die rechte die negativen Zahlen von $-0\,0\,0\,1$ bis $-0\,1\,1\,1$. An diesem Beispiel ist zu erkennen, daß die positiven Zahlen in ihrer werthöchsten Stelle eine 0, die negativen Zahlen eine 1 haben. Durch diese Überlegung ist man nun in der Lage, negative Binärzahlen durch eine 1 in der werthöchsten Stelle zu kennzeichnen. Diese Art der negativen Zahlendarstellung heißt **Zweierkomplementdarstellung**.

Merke:
In der Zweierkomplementdarstellung wird durch eine 0 in der werthöchsten Stelle eine positive Zahl, durch eine 1 eine negative Zahl gekennzeichnet.

Dieser Sachverhalt ist in Bild 2.5.2 dargestellt.

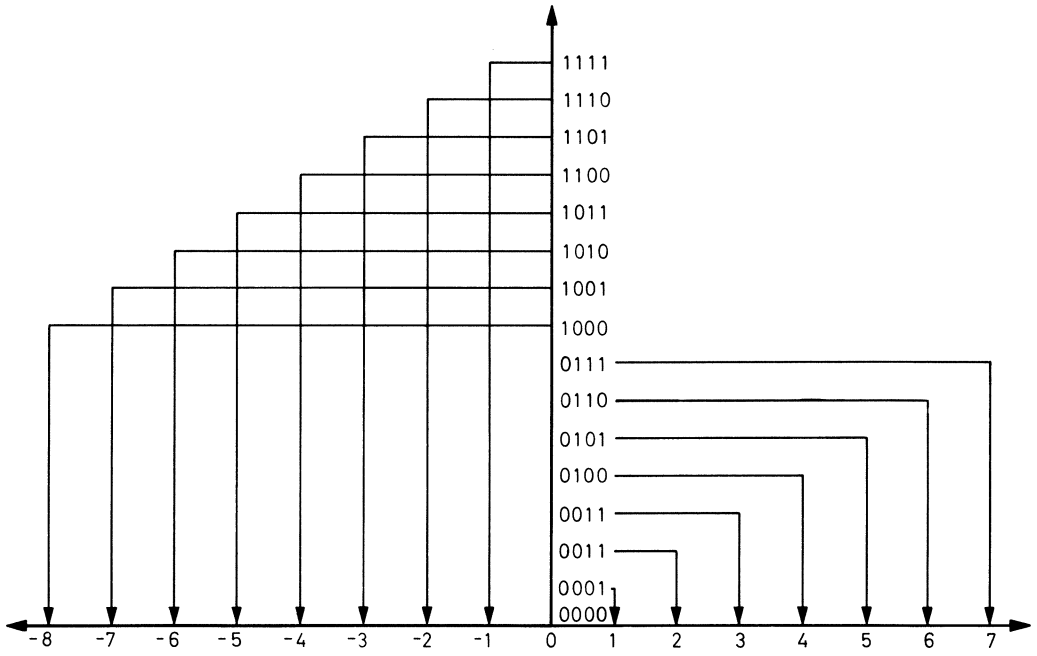


Bild 2.5.2
Zweierkomplementdarstellung

Wenn wir dieses Schema auf den Zahlenkreis übertragen, ergibt sich eine Darstellung nach Bild 2.5.3.

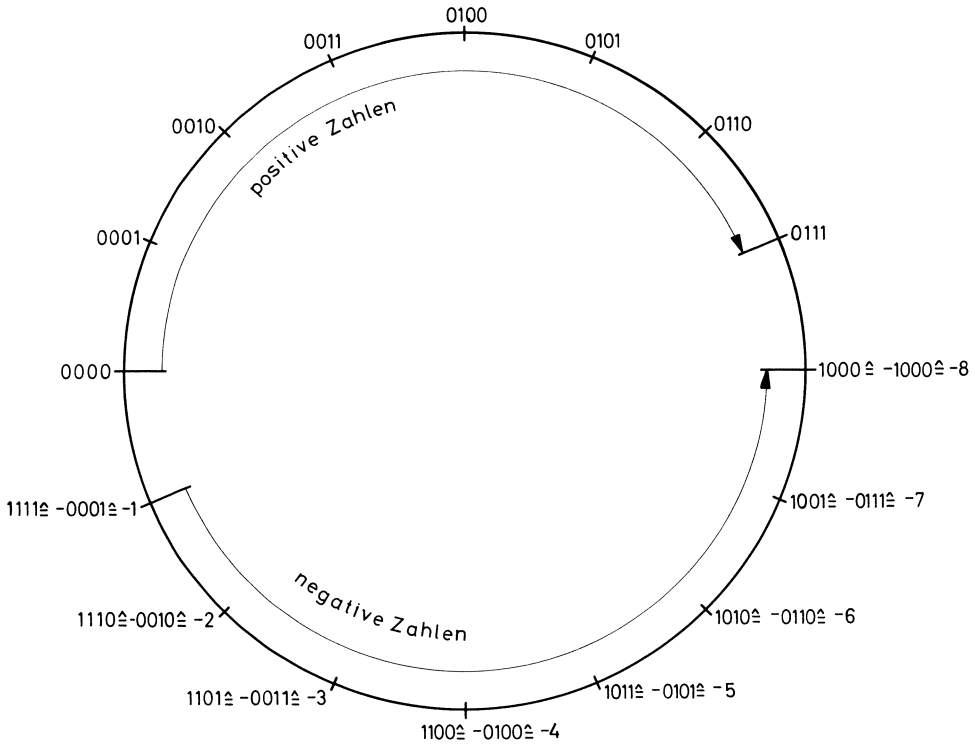


Bild 2.5.3
Darstellung der Zweierkomplementzahlen am Beispiel von 4-bit-Zahlen

Bei dieser Darstellungsart liegen die positiven Zahlen auf dem oberen Halbkreis, die negativen auf dem unteren Halbkreis. Aus diesem Beispiel geht auch deutlich hervor, daß das werthöchste bit das Vorzeichen bestimmt.
Die dem **Betrag** nach größte negative Zahl ist beim Beispiel der 4-bit-Zahl die Zahl 1 0 0 0, bei einer 8-bit-Zahl ist es 1 0 0 0 0 0 0 0, also immer eine Zahl der Form 1 0 0 0 . . . 0 0. Diese Zahl spielt eine Sonderrolle in dem Zahlenvorrat. Dies soll am Beispiel einer 8-bit-Zahl demonstriert werden, es gilt aber allgemein für beliebige Wortlängen. Bildet man das Zweierkomplement nach der Beziehung $-A = \overline{A} + 1$, so erhält man:

$$\begin{array}{rr} A: & 1\,0\,0\,0\,0\,0\,0\,0 \\ \overline{A}: & 0\,1\,1\,1\,1\,1\,1\,1 \\ 1: & + \qquad\qquad 1 \\ \hline -A: & 1\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

Man erhält also das merkwürdige Ergebnis, daß diese Zahl gleich ihrem negativen Wert ist, eine Eigenschaft die sonst nur die Zahl 0 hat. Dieser Punkt muß vom Programmierer beim Rechnen mit Zweierkomplementzahlen bedacht werden, da er zu falschen Rechenergebnissen führen kann.
Beim Rechnen mit positiven ganzen Zahlen hatten wir die Möglichkeit eines Überlaufes betrachtet. Auch beim Rechnen mit Zweierkomplementzahlen kann ein arithmetischer Überlauf, also ein Überschreiten des zulässigen Zahlenvorrates, vorkommen. Betrachten wir als Beispiel wieder einen 4-bit-Rechner und führen folgende Operation aus:

$$\begin{array}{r} 0\,0\,1\,0 \\ +\,0\,0\,1\,1 \\ \hline 0\,1\,0\,1 \end{array} \qquad \text{oder dezimal} \qquad \begin{array}{r} 2 \\ +\,3 \\ \hline 5 \end{array}$$

Wir sind bei dieser Operation im oberen Halbkreis des Zahlenkreises geblieben, das Ergebnis ist richtig. Betrachten wir dagegen folgende Aufgabe:

$$\begin{array}{r} 0\,0\,1\,1 \\ +\,0\,1\,1\,0 \\ \hline 1\,0\,0\,1 \end{array} \qquad \begin{array}{r} 3 \\ +\,6 \\ \hline -\,7 \end{array}$$

Wir erhalten in diesem Falle eine Summe, die im unteren Halbkreis liegt, d. h., die Summe ist nach Bild 2.5.3 eine negative Zahl, was ein offensichtlich falsches Ergebnis ist. Diese Operation führte zu einem **arithmetischen** Überlauf.
Hätten wir diese Aufgabe am Zahlenkreis nach Abb. 2.4.2, d. h., also nur mit positiven Zahlen, gerechnet, so wäre das richtige Ergebnis entstanden. Legt man hingegen wie in Bild 2.5.3 die Zweierkomplementdarstellung, d. h. also positive und negative Binärzahlen, zugrunde, so wird der dem **Betrag** nach vorhandene Zahlenbereich halbiert. Damit hat ein 4-bit-Rechner vom **Betrag** her nur eine 3-bit-Kapazität, so daß bei Überschreiten der positiven Zahl 0 1 1 1 $\triangleq$ 7 bereits ein arithmetischer Überlauf entsteht.

Allgemein gilt:
Der Zahlenvorrat eines n-bit-Rechners ist bei Darstellung von
- nur **positiven** Zahlen begrenzt auf 0 bis $2^n - 1$
- positiven **und** negativen Zahlen auf $-(2^{n-1})$ bis $+(2^{n-1}-1)$

Das Ergebnis 1 0 0 1 des letzten Beispieles würde ein Rechner, der nach der Zweierkomplementdarstellung arbeitet, als $-0\,1\,1\,1 = -7$ interpretieren. Manche Mikroprozessoren enthalten ein weiteres Flag-FF, das sog. V-Flag, das beim Auftreten eines arithmetischen Überlaufes, also eines Überlaufes bei Zweierkomplementarithmetik, gesetzt wird. Die Stellung dieses V-Flags kann wie das Carry-Flag über ein Programm geprüft werden, so daß im Falle des arithmetischen Überlaufes geeignete Maßnahmen ergriffen werden können. In der Integer Arithmetic reicht z.B. bei einem 8-bit-Rechner der Zahlenbereich von 0 bis 256. Per Definition entsteht dann ein Überlauf, wenn z. B. eine Additionsaufgabe das Ergebnis 258 bringt.
Ein 8-bit-Rechner in Zweierkomplementarithmetik hat einen Zahlenbereich von -128 bis $+127$. Wenn z. B. die Aufgabe $73 + 58 = 131$ gerechnet werden soll, entsteht ein Überlauf

in den negativen Zahlenbereich. Ein Überlauf in den positiven Zahlenbereich würde entstehen, wenn wir (−73) + (−58) = −131 rechnen würden. In beiden genannten Fällen wird der vorhandene Zahlenbereich überschritten. Kein Überlauf entsteht, wenn z. B. die Aufgabe −20 + 30 = +10 rechnen. Nachfolgend ist dieses Beispiel mit Binärzahlen durchgeführt:

```
  1 1 1 0 1 1 0 0 ≙ −20 in Zweierkomplementarithmetik
+ 0 0 0 1 1 1 1 0 ≙ +30 in Zweierkomplementarithmetik
  1 0 0 0 0 1 0 1 0 ≙ +10 in Zweierkomplementarithmetik
```

Auf den ersten Blick scheint es hier so, als ob ein Überlauf entstehen würde. Da der Rechner aber nur 8 bit Wortlänge hat, nimmt die 1 im 9. bit keinen Einfluß auf das Ergebnis. Die 0 im 8. bit sagt aus, daß es sich um ein positives Ergebnis mit dem Betrag $1\ 0\ 1\ 0_2 = 10_{10}$ handelt.
Ein weiteres Beispiel:

```
  1 1 0 1 1 1 1 1 ≙ −33 in Zweierkomplementarithmetik
+ 0 0 1 1 1 0 0 0 ≙ +56 in Zweierkomplementarithmetik
  1 0 0 0 1 0 1 1 1 ≙ +23 in Zweierkomplementarithmetik
```

Auch hier hat die 1 im 9. bit keinen Einfluß auf das Ergebnis, ein Überlauf entsteht ebenfalls nicht. Im nächsten Beispiel werden 2 negative Binärzahlen addiert:

```
  1 0 0 1 1 1 1 1 ≙ −97 in Zweierkomplementarithmetik
+ 1 0 1 0 0 1 1 1 ≙ −89 in Zweierkomplementarithmetik
  1 0 1 0 0 0 1 1 0 ≙ +70 in Zweierkomplementarithmetik
```

In diesem Falle wird der negative Zahlenbereich überschritten, es entsteht also ein Überlauf. Zum Abschluß noch eine Subtraktionsaufgabe:

```
  0 1 0 1 1 0 1 1 ≙ +91 in Zweierkomplementarithmetik
− 1 0 1 1 1 0 1 1 ≙ −69 in Zweierkomplementarithmetik
```

Durch den Rechenbefehl − bildet der Rechner vom Subtrahenden das Zweierkomplement und addiert dieses zum Minuenden. Die Aufgabe lautet dann:

```
  0 1 0 1 1 0 1 1
+ 0 1 0 0 0 1 0 1 ≙ +69 in Zweierkomplementarithmetik
  1 0 1 0 0 0 0 0 ≙ −96 in Zweierkomplementarithmetik
```

Auch hier erfolgt ein Überlauf, da das richtige Ergebnis 91 − (−69) = 91 + 69 = 160 den positiven Zahlenbereich überschreitet.

Vertiefungsstoff:
Wenn der Mikroprozessor kein V-Flag enthält und der Fall des arithmetischen Überlaufes bei Zweierkomplementarithmetik trotzdem überwacht werden soll, so ist es erforderlich, das V-Flag durch Programmieren der entsprechenden booleschen Gleichungen zu simulieren. Es sei:

D das Vorzeichen-bit des 1. Operanden
S das Vorzeichen-bit des 2. Operanden und
R das Vorzeichen-bit des Ergebnisses der Operation

Bei 8-bit-Zahlen:

```
  0 1 1 0 1 0 0 0     D = 0
+ 1 1 0 1 1 1 1 1     S = 1
  0 1 0 0 0 1 1 1     R = 0
```

Damit müßte im Falle der Addition ein *V*-Signal der Beziehung

$$V = (D \wedge S \wedge \overline{R}) \vee (\overline{D} \wedge \overline{S} \wedge R)$$

gebildet werden.
Diese Gleichung läßt sich folgendermaßen deuten:
Die 1. Klammer ist nur dann 1, wenn $D = 1$, $S = 1$ und $\overline{R} = 1$, also $R = 0$ ist, d. h., wenn 2 negative Operanden ($D = 1$, $S = 1$) addiert ein positives Ergebnis ($R = 0$) liefern, was in der Tat einem arithmetischen Überlauf entspricht. Die 2. Klammer ist nur dann 1, wenn $D = 0$, $S = 0$ und $R = 1$ sind, d. h., wenn die Addition zweier positiver Zahlen ($D = 0$, $S = 0$) zu einem negativen Ergebnis ($R = 1$) führt, was ebenfalls einem arithmetischen Überlauf entspricht. V ist dann 1, wenn entweder die 1. Bedingung oder die 2. Bedingung erfüllt ist.

Im Falle der Subtraktion

0 1 1 0 1 0 0 0	$D = 0$
− **1** 1 0 1 1 1 1 1	$S = 1$
1 0 0 0 1 0 0 1	$R = 1$

lautet die entsprechende boolesche Gleichung:

$$V = (D \wedge \overline{S} \wedge \overline{R}) \vee (\overline{D} \wedge S \wedge R)$$

Diese Gleichung kann sinngemäß ebenso gedeutet werden wie die vorhergehende Gleichung.

Ende Vertiefungsstoff.

Fragen zu den Abschnitten 2.4 und 2.5

1. Bilden Sie zu folgenden Binärzahlen das Einer- und das Zweierkomplement!
   a) 1 0 0 1
   b) 0 1 1 1 0 0 1
   c) 0 0 0 0 0 0 0 0
   d) 1 1 1 1 1

2. Geben Sie an, welche der folgenden 8-bit-Zweierkomplementzahlen positiv und welche negativ sind!
   a) 1 0 1 1 0 1 1 1
   b) 1 1 1 1 0 0 0 0
   c) 0 1 0 1 1 1 1 1
   d) 0 0 0 0 0 0 0 0

3. Berechnen Sie nachfolgende Ausdrücke durch Addition des Zweierkomplementes!
   a)
   ```
     1 1 0 1 1 0 1 1
   - 0 1 1 0 1 0 1 1
   ```
   b)
   ```
         1 0 1 1
       - 0 0 1 1
   ```
   c)
   ```
     0 1 1 0 1 0 1 1
   - 1 1 0 1 1 0 1 1
   ```

4. Was ist bei arithmetischen Operationen ein Überlauf, und was versteht man unter einem Übertrag?

5. Bei welchen der nachfolgenden Operationen in Integer Arithmetic entsteht bei einem 8-bit-Rechner ein Überlauf bzw. Übertrag?
   a)
   ```
     1 1 0 1 1 1 1 1
   + 0 0 1 1 1 0 0 0
   ```

b)
$$\begin{array}{r} 0\,1\,0\,1\,1\,0\,1\,1 \\ +\,1\,0\,1\,1\,1\,0\,1\,1 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 1\,0\,0\,0\,1\,0\,1\,1 \\ +\,0\,0\,1\,1\,1\,0\,1\,0 \\ \hline \end{array}$$

6. Bei welcher der nachfolgenden Aufgaben in Zweierkomplementarithmetik entsteht bei einem 8-bit-Rechner ein Überlauf?

a)
$$\begin{array}{r} 0\,1\,0\,1\,1\,1\,1\,1 \\ +\,0\,1\,1\,0\,0\,0\,0\,1 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 1\,1\,1\,0\,1\,0\,0\,0 \\ +\,1\,1\,1\,1\,0\,1\,1\,1 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 1\,0\,0\,1\,1\,0\,1\,0 \\ -\,0\,0\,0\,1\,0\,1\,1\,1 \\ \hline \end{array}$$

2.6 Rechnen mit mehrfacher Genauigkeit

Der Zahlenvorrat eines Rechners mit einer Wortlänge von n bit ist begrenzt auf 0 bis $2^n - 1$, wenn nur positive Zahlen dargestellt werden, bzw. auf -2^{n-1} bis $2^{n-1} - 1$ bei Zweierkomplementzahlen, bei einem 8-bit-Mikroprozessor also auf 0 bis 255 bzw. −128 bis 127. In der Praxis braucht man in der Regel einen größeren Zahlenbereich zum Rechnen. Man rechnet dazu mit mehrfacher Genauigkeit (multi precision arithmetic). Zur Darstellung einer Zahl werden dabei 2 oder mehr Rechnerwörter verwendet. Diese Rechnerwörter werden nacheinander, also seriell abgearbeitet. Das Rechnen mit mehrfacher Genauigkeit ist also langsamer als das Rechnen mit einfacher Genauigkeit. Das folgende Beispiel soll die Addition mit mehrfacher Genauigkeit verdeutlichen.
Gegeben ist ein Rechner mit 8 bit Wortlänge. Es sind 2 Binärzahlen mit je 24 bit, also dreifacher Genauigkeit, zu addieren.

0 1 0 1 1 1 0 1	0 0 1 1 1 0 0 1	1 0 0 1 0 0 1 1	
+ 0 0 1 0 0 1 0 0	1 1 0 0 0 1 0 1	1 0 1 0 0 0 0 1	
	1		Übertrag
1 0 0 0 0 0 0 1	1 1 1 1 1 1 1 1	0 0 1 1 0 1 0 0	

Diese Aufgabe wird in 3 Schritten durchgeführt:

1. Schritt: Die beiden rechts stehenden Wörter werden in dem 8-bit-breiten Rechenwerk des Rechners addiert

$$\begin{array}{rr} & 1\,0\,0\,1\,0\,0\,1\,1 \\ & +\,1\,0\,1\,0\,0\,0\,0\,1 \\ & \quad\quad\quad 1\,1 \\ \hline \text{Übertrag } \boxed{1} & 0\,0\,1\,1\,0\,1\,0\,0 \end{array}$$

Dabei erhält man das rechte Wort, also die rechten 8 bit der gesuchten Summe. Zusätzlich ergibt sich evtl. ein Übertrag in der werthöchsten Stelle, also ganz links. Dieser Übertrag muß im 2. Schritt berücksichtigt werden.

2. Schritt: Die nächsten beiden Wörter werden addiert, zusätzlich wird ein evtl. Übertrag von der vorhergehenden Stelle ganz rechts addiert.

$$\begin{array}{rr} & 0\,0\,1\,1\,1\,0\,0\,1 \\ & 1\,1\,0\,0\,0\,1\,0\,1 \\ & \quad\quad\quad\quad\quad 1 \quad \text{Übertrag vom vorherigen Wort} \\ \text{kein Übertrag in} & \quad\quad\quad\quad 1 \\ \text{die nächste Stelle } \boxed{0} & 1\,1\,1\,1\,1\,1\,1\,1 \end{array}$$

Man erhält das nächste Wort der Summe und evtl. einen Übertrag in das nächste Wort.

3. Schritt: Wie Schritt 2 mit den nächsten Wörtern

```
  0 1 0 1 1 1 0 1
  0 0 1 0 0 1 0 0
              0      Übertrag vom vorherigen Wort
  1 1 1 1 1      1 1
  ---------------
  1 0 0 0 0 0 0 1
```

Dieses Verfahren ist leicht für beliebige Genauigkeiten zu verallgemeinern, indem man die Addition mit Berücksichtigung des Übertrages entsprechend oft durchführt.

Die Subtraktion wird genauso durchgeführt wie die Addition.
Beispiel einer 24-bit-Subtraktion mit 8-bit-Rechnern:

0 1 0 1 1 1 0 1	0 0 1 1 1 0 0 1	1 0 0 1 0 0 1 1	
− 0 0 1 0 0 1 0 0	1 1 0 0 0 1 0 1	1 0 1 0 0 0 0 1	
1	1		Entlehnung
1	1 1 1	1 1	
0 0 1 1 1 0 0 0	0 1 1 1 0 0 1 1	1 1 1 1 0 0 1 0	
3. Schritt	2. Schritt	1. Schritt	

2.7 Binärcodierung von Dezimalzahlen

Bei einer Reihe von Meßgeräten und bei Anwendungsfällen von Rechnern, bei denen es erforderlich ist, dezimal zu rechnen, hat sich eine Gruppe von Codes für Dezimalzahlen eingeführt, die sogenannten BCD(binary-coded-decimal)-Codes. Hierbei wird jede Dezimalstelle, die ja die Zahlenwerte 0 bis 9 annehmen kann, für sich in einer 4-bit-Binärzahl codiert. Der vor allem in Mikroprozessoren am häufigsten verwendete Code ist der sog. **8421-Code**, der nach den Gewichtsfaktoren, die den 4 bit zugeordnet sind, benannt ist (Tab. 2.7.1).

Dezimal	8421-Code
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Tab. 2.7.1
8421-Code

Die verbleibenden 6 bit-Kombinationen von 1 0 1 0 bis 1 1 1 1 sind illegal (sog. Pseudotetraden) und werden bei arithmetischen Operationen nicht verwendet.
Beispiel einer BCD-Zahl im 8421-Code:

19 378 = 0 0 0 1 1 0 0 1 0 0 1 1 0 1 1 1 1 0 0 0

Daneben ist noch der sog. **3-Exzeß-Code** üblich, der von der in Tab. 2.7.2 gezeigten Zuordnung der Dezimalziffern und der Binärzahlen ausgeht.
Dieser Code hat den Vorteil, daß die illegalen bit-Kombinationen 0 0 0 0 bis 0 0 1 0 und 1 1 0 1 bis 1 1 1 1 symmetrisch liegen zu dem zulässigen Zahlenvorrat, so daß gewisse arithmetische Operationen vereinfacht werden.
Beispiel einer BCD-Zahl im 3-Exzeß-Code:

19 378 = 0 1 0 0 1 1 0 0 0 1 1 0 1 0 1 0 1 0 1 1

Dezimal	3-Exzeß-Code
0	0 0 1 1
1	0 1 0 0
2	0 1 0 1
3	0 1 1 0
4	0 1 1 1
5	1 0 0 0
6	1 0 0 1
7	1 0 1 0
8	1 0 1 1
9	1 1 0 0

Tab. 2.7.2
3-Exzeß-Code

Die Verarbeitung von BCD-Zahlen bedeutet, daß zwar Dezimalzahlen eingegeben werden, intern im Rechner werden jedoch Binärzahlen verarbeitet.
Als Beispiel für die Arithmetik mit BCD-Zahlen soll hier die Addition im 8421-Code behandelt werden, da eine Reihe von Mikroprozessoren spezielle Instruktionen dafür hat.
Die Arbeitsweise soll anhand einiger Beispiele demonstriert werden:

1.

```
  5        0 1 0 1
+ 4      + 0 1 0 0
  9        1 0 0 1
```

In diesem Beispiel wurden 2 BCD-Zahlen nach den Regeln der binären Addition addiert, und es ergab sich ein richtiges Ergebnis.

2.

```
  5        0 1 0 1
+ 7      + 0 1 1 1
 12        1 1 0 0
```

Bei diesem Beispiel wurden ebenfalls die Regeln der binären Addition angewandt, dabei ergab sich aber im Ergebnis ein illegaler Code, nämlich 1100.
Dieses Ergebnis kann dadurch korrigiert werden, daß man zu dem Ergebnis der binären Addition die Zahl 6, also 0 1 1 0, addiert, d.h., man „überspringt" bei der Addition die 6 illegalen Codes von 1 0 1 0 bis 1 1 1 1. Die Korrektur

```
          1 1 0 0
        + 0 1 1 0
  0 0 0 1   0 0 1 0
```

ergibt das richtige Ergebnis 12 im 8421-Code.

3. Ein weiterer Fall muß noch betrachtet werden:

```
  8          1 0 0 0
+ 9        + 1 0 0 1
 17    0 0 0 1   0 0 0 1
```

Die binäre Addition ergibt in diesem Fall das Ergebnis 11 im 8421-Code.
Das ist ein legaler BCD-Code, aber ein falsches Ergebnis. Dieser Fehler entsteht dadurch, daß bei der Addition die 6 illegalen Codes mitgezählt wurden, die Korrektur führt man also ebenso wie im vorigen Fall durch Addition von 0 1 1 0 aus.

```
  0 0 0 1      0 0 0 1
+ 0 0 0 0    + 0 1 1 0
  0 0 0 1      0 1 1 1
```

Man erhält also 17 in BCD-Darstellung. Dieser Fall ist daran erkennbar, daß das Ergebnis eine legale BCD-Zahl war, daß aber ein Übertrag in die nächsthöhere BCD-Stelle erfolgte.

Eine Addition im 8421-Code erfolgt somit in mehreren Schritten. Dies ist in einem einfachen Flußdiagramm (Bild 2.7.1) grafisch dargestellt.

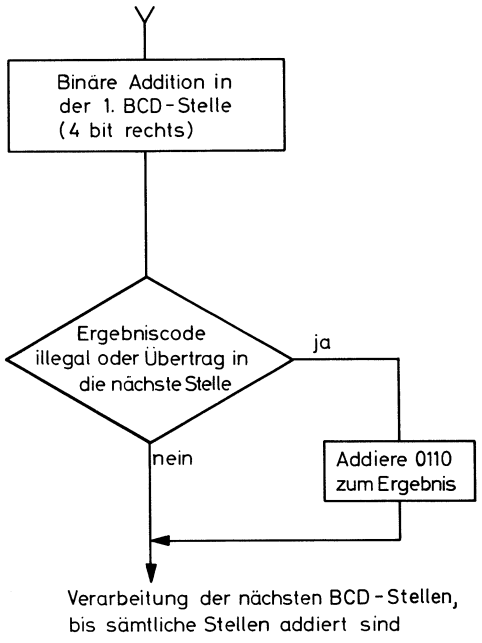


Bild 2.7.1
Flußdiagramm zur Erläuterung einer Addition im 8421-Code

Bei der Subtraktion müssen die illegalen Codes auf ähnliche Weise berücksichtigt werden, ebenso bei Multiplikation und Division, die ja eine wiederholte Anwendung der Addition bzw. Subtraktion sind.
Zusammenfassend kann man also über BCD-Codes folgendes sagen:
Die Tatsache, daß illegale Codewörter vorkommen, bedeutet, daß für einen gegebenen Zahlenvorrat in BCD-Codierung eine größere Wortlänge erforderlich ist, als in der binären Darstellung. Eine größere Wortlänge bedeutet größeren Speicherbedarf und mehr Schaltungsaufwand.
Die arithmetischen Operationen in BCD-Codes sind generell komplizierter und somit in der Praxis auch langsamer und aufwendiger als im Binärcode. Dagegen stehen die Vorteile der dezimalen Arbeitsweise, das Wegfallen der Binär-Dezimalumwandlungen bei Ein- und Ausgabe, so daß in jedem Einzelfall geprüft werden muß, ob BCD-Codes zweckmäßig sind.

Fragen zu Abschnitt 2.7

1. Addieren Sie folgende BCD-Zahlen

   a) 0 0 0 1 0 1 0 1 1 0 0 1
   + 0 0 1 1 0 1 1 1 0 0 1 0

   b) 0 0 0 1 1 0 0 0 0 1 1 1
   + 1 0 0 1 1 0 0 1 0 1 1 0

3. Arbeitsweise eines Rechners bzw. Computers

Grundsätzlich läßt sich die Arbeitsweise eines Rechners dadurch beschreiben, daß er in der Lage ist, Daten so zu verarbeiten, wie es durch ein Programm vorgeschrieben ist. Durch verschiedene Programme kann ein Rechner zur Lösung verschiedener Aufgaben eingesetzt werden. Bis vor einigen Jahren waren Rechner nur als sehr voluminöse und teure Einrichtungen zu haben. Dadurch war ihr Einsatz auf komplexe Aufgaben begrenzt, wie z.B. die Verarbeitung von Massendaten im kommerziellen Bereich. Erst die rapide Weiterentwicklung der Halbleitertechnologie ermöglichte es, auch kleinere und nicht so teure Rechner herzustellen,

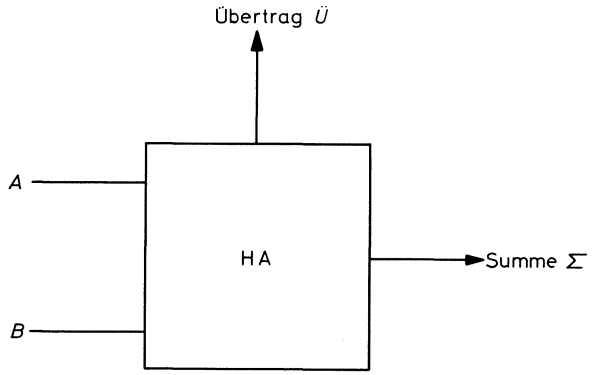
die allerdings auch weniger leistungsfähig und weniger komfortabel vom ganzen Bedienungsablauf her gesehen waren. Diese kleineren Computer wurden und werden allgemein als **Minicomputer** bezeichnet.
Der nächste Schritt in dieser Entwicklungsrichtung war der, mittels der sog. **L**arge **S**cale **I**ntegration (LSI), was mit hohem Integrationsgrad übersetzt werden kann, durch mehrere tausend Bauelemente die **Zentraleinheit CPU** (CPU = **C**entral **P**rocessing **U**nit) eines Mikrocomputers auf einem Chip herzustellen. Eine solche Zentraleinheit eines Mikrocomputers wird als **Mikroprozessor** bezeichnet. Interessant in diesem Zusammenhang dürfte für Sie sein, daß der in unserem Experimentiersystem verwendete Mikroprozessor vom Typ 8080 auf einem Chip mit 23 mm^2 (rund 4,8 mm x 4,8 mm) mehr als 4 500 MOS-Transistoren enthält. Mit diesen über 4 500 Transistoren ist es nun möglich, ein komplettes Steuer- und Rechenwerk für einen Mikrocomputer zu realisieren. Im Rechenwerk werden dabei die arithmetischen und logischen Operationen ausgeführt, während das Steuerwerk den internen Ablauf im Rechner steuert. Damit aus einem Mikroprozessor ein Mikrocomputer wird, sind weitere zusätzliche Einrichtungen wie Programmspeicher, Datenspeicher, Ein- und Ausgabebausteine, zusätzliche Logikschaltungen usw. erforderlich. Wie umfangreich diese zusätzlichen Einrichtungen werden, hängt dabei vom jeweiligen Anwendungsfall ab. Wichtig ist jedoch die Erkenntnis, daß man in der Praxis mit einem Mikroprozessorbaustein alleine noch keine Aufgabenstellungen lösen kann, dazu ist immer ein Mikrorechner erforderlich. In den nachfolgenden Abschnitten soll die prinzipielle Arbeitsweise eines Mikrorechners erläutert werden. Da sich prinzipiell die Arbeitsweise eines Mikrorechners von einem anderen Rechner nicht unterscheidet, werden Schritt für Schritt die einzelnen Baustufen behandelt, die für einen Rechner allgemein notwendig sind. Jeder Schritt wird dabei durch entsprechende Experimente verdeutlicht.

3.1 Addierwerk

Aus Abschnitt 2. ist bekannt, wie man Binärzahlen addiert. Wenn im einfachsten Falle 2 1-bit-Binärzahlen addiert werden sollen, so gibt es grundsätzlich folgende Möglichkeiten:

$A + B$	Summe	Übertrag
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

Eine Schaltung, die in der Lage ist, diese Rechenoperationen durchzuführen, wird als Halbaddierer (HA) bezeichnet. In Bild 3.1.1 ist ein HA in Blockschaltbildform dargestellt.



a)

A	B	Σ	Ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

b)

Bild 3.1.1
Halbaddierer
a) Blockschaltbild
b) Funktionstabelle

Aus der Funktionstabelle können die Funktionsgleichungen abgeleitet werden, die für die Summen- und Übertragsbildung erfüllt sein müssen:

$$\text{Summe } \Sigma = (A \wedge \overline{B}) \vee (\overline{A} \wedge B) = A \veebar B \quad \text{(EXCLUSIV-ODER)}$$
$$\text{Übertrag } \ddot{U} = A \wedge B \quad \text{(UND)}$$

Aus diesen Gleichungen kann jetzt die logische Schaltung eines Halbaddierers abgeleitet werden (Bild 3.1.2).

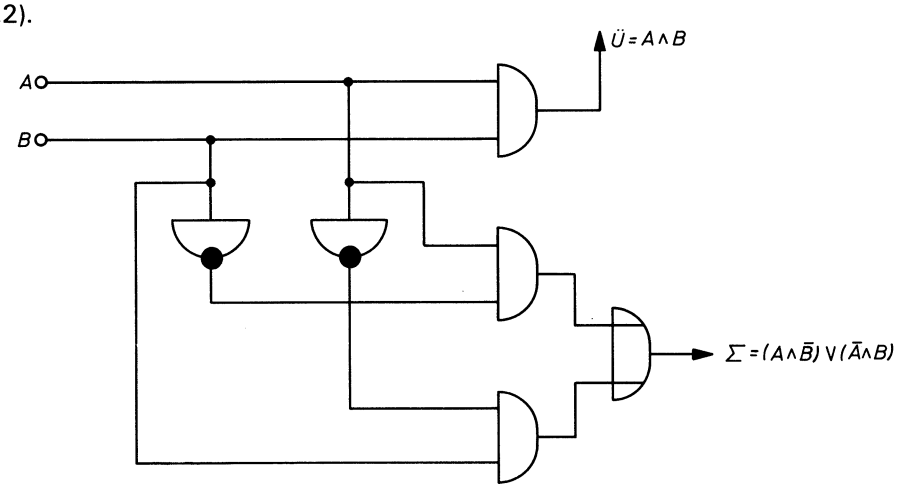
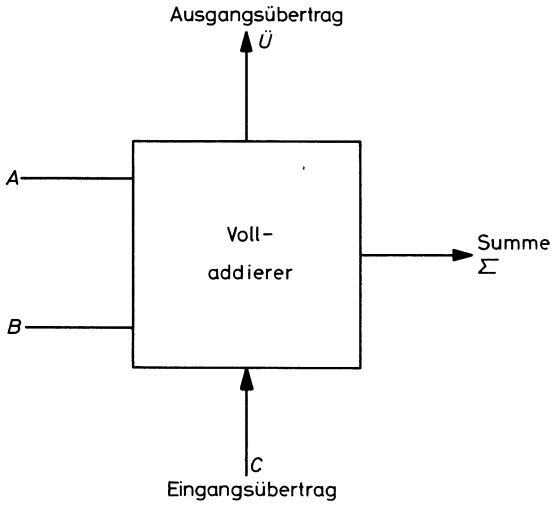


Bild 3.1.2
Schaltung eines Halbaddierers

Wenn nun mehrstellige bit-Kombinationen (sog. Wörter) addiert werden sollen, so reicht ein Halbaddierer hierfür nicht aus. In diesem Fall muß nämlich bei einem Übertrag dieser in der nächsthöherwertigen Stelle berücksichtigt werden. Dies kann nur mit einem Volladdierer gelöst werden, der außer den Eingängen A und B noch einen Übertragseingang C hat (Bild 3.1.3).



a)

A	B	C	Σ	$\ddot{U}$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

b)

Bild 3.1.3
Volladdierer
a) Blockschaltbild
b) Funktionstabelle

Für die Summe gilt jetzt die Funktionsgleichung:

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

Diese Gleichung läßt sich nach den Regeln der Schaltalgebra umformen in:

$$\Sigma = (A \veebar B) \veebar C = A \veebar B \veebar C$$

Diese Gleichung entspricht der Kaskadierung von 2 EXCLUSIV-ODER-Verknüpfungen, wie schon in Abschnitt 1.3, Bild 1.3.5 gezeigt.
Für den Ausgangsübertrag gilt die Beziehung:

$$\ddot{U} = (A \wedge B \wedge \overline{C}) \vee (\overline{A} \wedge B \wedge C) \vee (A \wedge \overline{B} \wedge C) \vee (A \wedge B \wedge C)$$

Diese Gleichung läßt sich vereinfachen zu:

$$\ddot{U} = (A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$$

Damit ergibt sich für den Volladierer eine Schaltung entsprechend Bild 3.1.4.

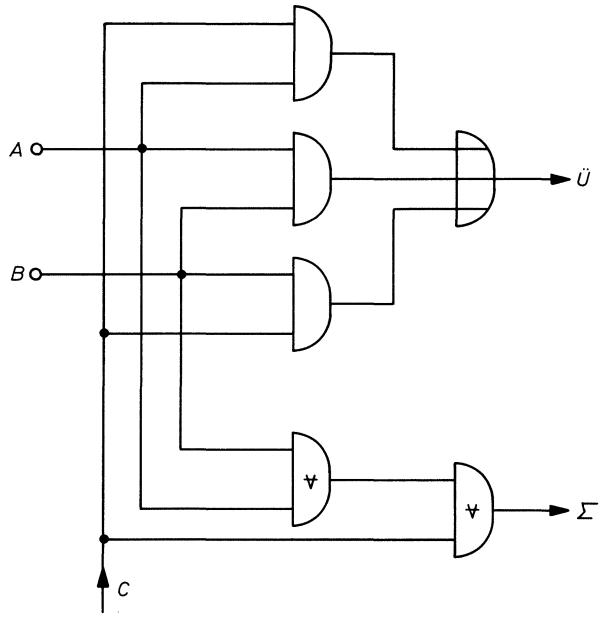


Bild 3.1.4
Schaltung eines Volladdierers

Sind nun 2 *n*-bit-Wörter zu addieren, müssen mehrere Volladdierer zusammengeschaltet werden. Hierzu ein einfaches Beispiel mit 2 3-bit-Wörtern:

	2^2	2^1	2^0
A:	1	0	1
B:	1	1	1
Ü:	1	1	
	1	1	0	0

Genau genommen werden für dieses Beispiel ein HA und 2 VA benötigt, da in der Stelle 2^0 noch kein Übertrag vorhanden sein kann. In der Praxis wird man jedoch auch für diese Stelle einen VA benutzen, und den Eingang C_0 für ein solches Beispiel mit 0 belegen. Damit ergibt sich für die Lösung dieser Aufgabe eine Schaltung entsprechend Bild 3.1.5.

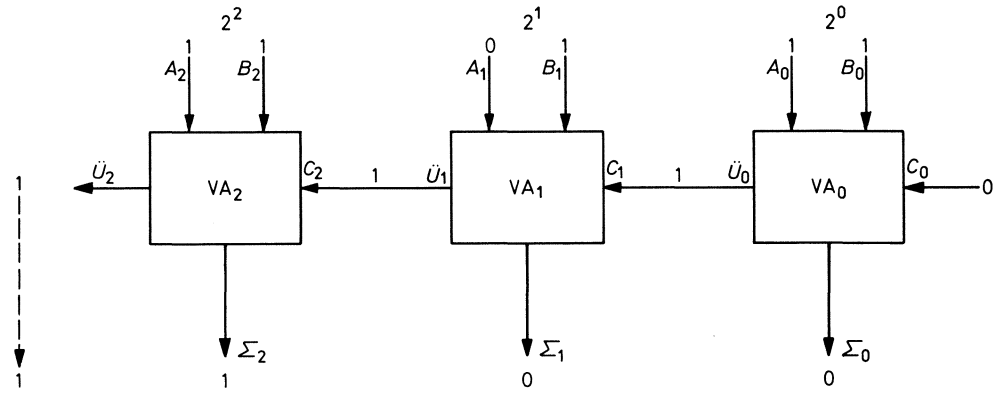


Bild 3.1.5
Addierwerk für 2 3-bit-Binärzahlen

Würde man bei diesem Addierwerk den Eingang C_0 statt mit 0 mit 1 beschalten, so würde das Ergebnis um 1 erhöht. Dies ist in der Praxis für bestimmte Anwendungsfälle erforderlich. Aus diesem Grunde erhält dieser Übertragseingang die Bezeichnung **Inkrementiereingang** INC (von Increment = Zuwachs). Ein Addierer mit einem Inkrementiereingang wird dann als **Ripple-Carry-Addierer** bezeichnet. In Bild 3.1.6 ist ein *n*-bit-Ripple-Carry-Addierer dargestellt.

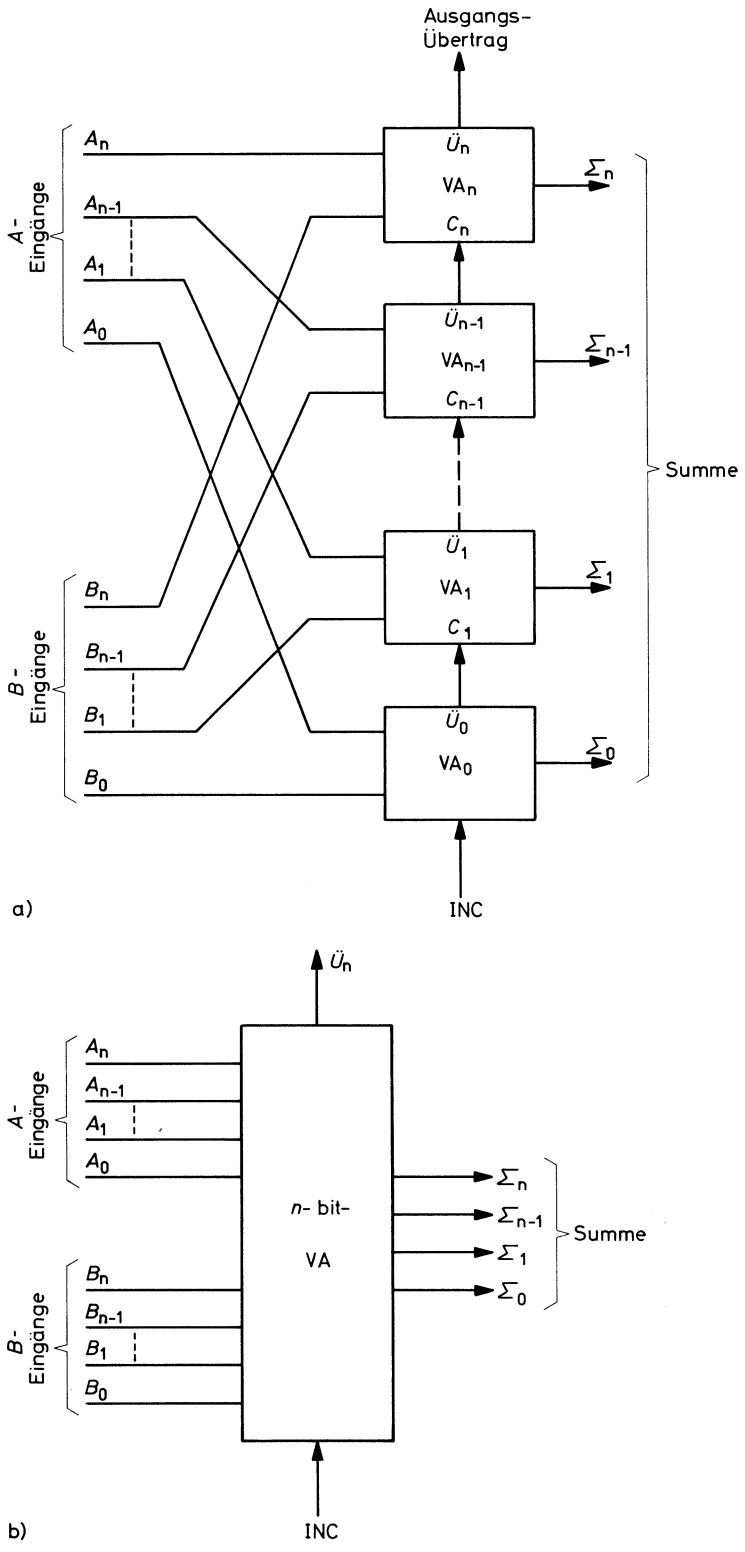


Bild 3.1.6
Ripple-Carry-Addierer
a) Funktionsschaltbild
b) Blockschaltbild

3.2 Addier-Subtrahierwerk

Ein Addierwerk nach Bild 3.1.6 läßt sich durch Vorschalten von Gattern so erweitern, daß viele weitere Funktionen damit verwirklicht werden können. Die in Bild 3.2.1 dargestellte Schaltung ermöglicht u. a. auch die Subtraktion von Binärzahlen und wird deshalb auch als Addier-Subtrahierwerk bezeichnet.

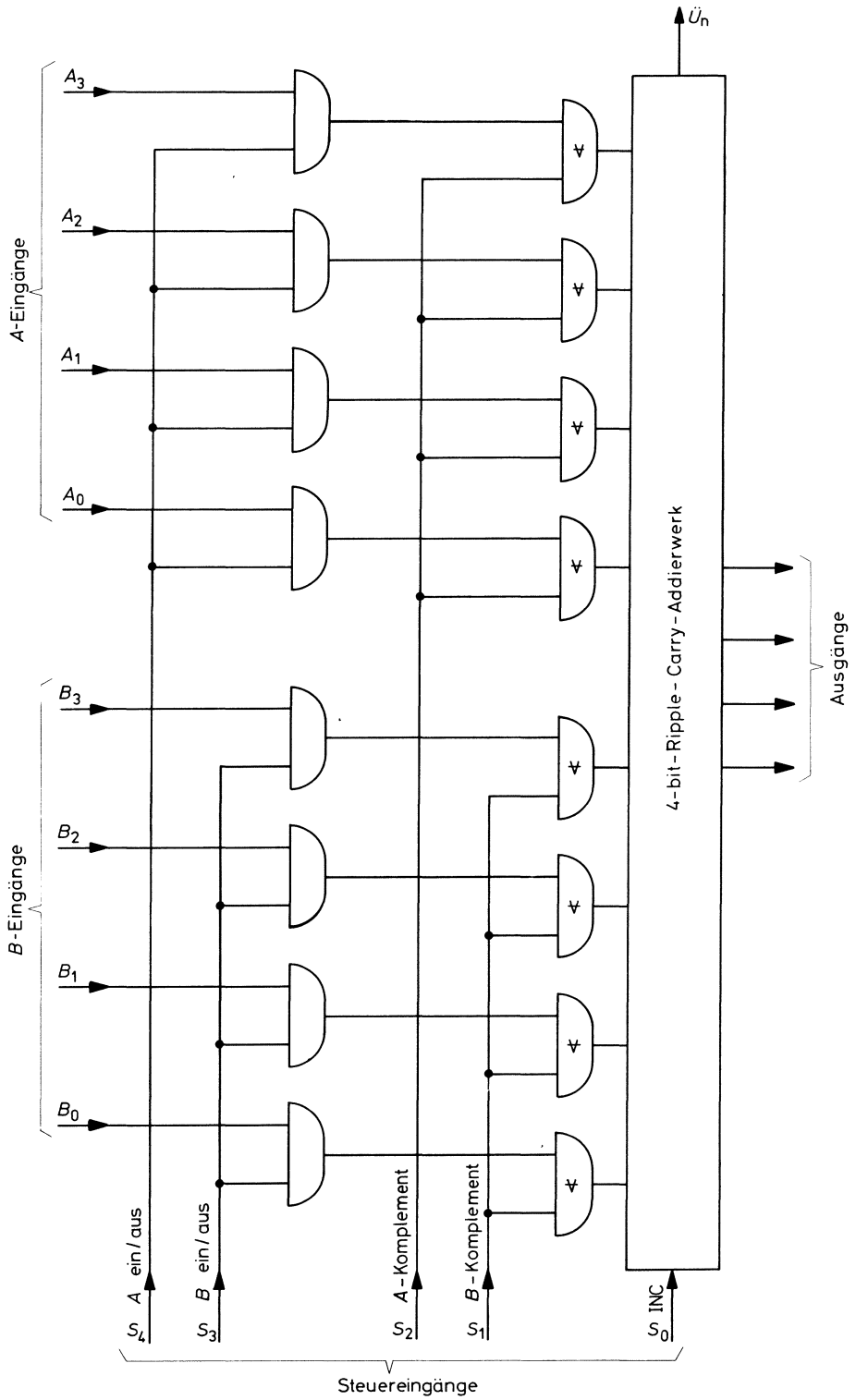


Bild 3.2.1
4-bit-Addier-Subtrahierwerk

Die so entstandene Schaltung enthält 5 Steuereingänge S_4 bis S_0, die es ermöglichen, die A- und B-Eingänge auf die unterschiedlichste Art miteinander zu verknüpfen. In Tab. 3.2.1 sind alle möglichen Eingangskombinationen mit den dazugehörigen Ausgangsfunktionen dargestellt.

S_4	S_3	S_2	S_1	S_0	Ausgangs-funktion
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	−1
0	0	0	1	1	0
0	0	1	0	0	−1
0	0	1	0	1	0
0	0	1	1	0	−2
0	0	1	1	1	−1
0	1	0	0	0	B
0	1	0	0	1	$B + 1$
0	1	0	1	0	$-B - 1 = \overline{B}$
0	1	0	1	1	$-B$
0	1	1	0	0	$B - 1$
0	1	1	0	1	B
0	1	1	1	0	$-B - 2$
0	1	1	1	1	$-B - 1 = \overline{B}$
1	0	0	0	0	A
1	0	0	0	1	$A + 1$
1	0	0	1	0	$A - 1$
1	0	0	1	1	A
1	0	1	0	0	$-A - 1 = \overline{A}$
1	0	1	0	1	$-A$
1	0	1	1	0	$-A - 2$
1	0	1	1	1	$-A - 1 = \overline{A}$
1	1	0	0	0	$A + B$
1	1	0	0	1	$A + B + 1$
1	1	0	1	0	$A - B - 1$
1	1	0	1	1	$A - B$
1	1	1	0	0	$B - A - 1$
1	1	1	0	1	$B - A$
1	1	1	1	0	$-A - B - 2$
1	1	1	1	1	$-A - B - 1$

Tab. 3.2.1
Steuerfunktionen für das Addier-Subtrahier-Werk nach Bild 3.2.1

Es würde zu weit führen, wollten wir alle 32 möglichen Eingangskombinationen ausführlich diskutieren, wir beschränken uns deshalb auf einige Beispiele. So liefert z. B. die Steuerfunktion S_4 bis S_0 = 1 1 0 0 0 die Ausgangsfunktion $A + B$, d. h., die Eingangssignale werden addiert. Wird S_3 oder S_4 auf 0 gebracht, werden die A- bzw. B-Eingänge abgeschaltet. Am Ausgang erscheint dann nur B oder A. Die oberen EXCLUSIV-ODER-Gatter verknüpfen die A-Eingänge mit S_2. Bei S_2 = 0, werden die A-Eingänge unverändert durchgeschaltet ($A \veebar 0 = A$). Bei S_2 = 1, werden die A-Eingänge komplementiert ($A \veebar 1 = \overline{A}$). Die Steuereingänge S_1 und S_2 dienen also dazu, die A- bzw. B-Eingänge zu komplementieren. Bei der Steuerfunktion S_4 bis S_0 = 0 0 0 1 0 z. B. ist S_1 = 1 während alle anderen Steuereingänge 0 sind. Dies bedeutet, daß alle Ausgänge der 4 oberen EXCLUSIV-ODER-Gatter 0 sind während die 4 unteren eine 1 liefern. Im Addierwerk wird also folgende Rechenoperation ausgeführt:

$$\begin{array}{rr} A: & 0\,0\,0\,0 \\ B: & +\,1\,1\,1\,1 \\ \hline \text{Ausgang:} & 1\,1\,1\,1 \end{array}$$

Dieses Ergebnis entspricht in der hier gewählten Zweierkomplementarithmetik −1.

Merke:
Bei allen Ausgangsfunktionen in Tab. 3.2.1 liegt die Zweierkomplementarithmetik zugrunde.

Die Steuerfunktion S_4 bis S_0 = 1 1 0 1 1 zeigt die Bedingungen für die Subtraktion $A - B$. Bei der Zweierkomplementarithmetik muß hierzu die Zahl B komplementiert werden (Einer-

komplement $\overline{B}$), hierzu wird dann eine 1 addiert (Zweierkomplement $\overline{B}$ + 1), und das so gewonnene Zwischenergebnis muß zur Zahl A addiert werden. Die Einerkomplementbildung von B wird durch $S_1 = 1$ bewirkt. Durch INC = 1 wird zu $\overline{B}$ eine 1 addiert ($\overline{B}$ + 1). Durch $S_2 = 0$ und $S_3 = S_4 = 1$ gelangt A in der anliegenden Form an das Addierwerk, so daß als Ergebnis die Funktion $A + \overline{B} + 1 = A - B$ erscheint. Durch Umpolen von S_1 und S_2 wird am Ausgang die Differenz $B - A$ gebildet. Wir empfehlen Ihnen, selbst nachzuvollziehen, wie die weiteren Ausgangsfunktionen erklärt werden können.

Exp. 1

3.3 Arithmetische logische Einheit (Arithmetic-Logic-Unit ALU)

Da alle Mikroprozessoren nicht nur arithmetische Verknüpfungen sondern auch logische Verknüpfungen ausführen können, muß die Schaltung nach Bild 3.2.1 erweitert werden. Es gibt hierzu mehrere Möglichkeiten. Im Rahmen dieses Lehrganges werden wir eine Variante behandeln, die einfach zu verstehen ist, obwohl die Lösung zu einer unökonomischen Schaltung führt. Dabei gehen wir davon aus, daß als zusätzliche Funktionen die 3 logischen Verknüpfungen

$A \wedge B$
$A \vee B$ und
$A \barwedge B$

zu bilden sind (jedes bit von A wird mit dem entsprechenden bit von B verknüpft). In Bild 3.3.1 ist eine Lösung für diese Aufgabenstellung gezeigt.

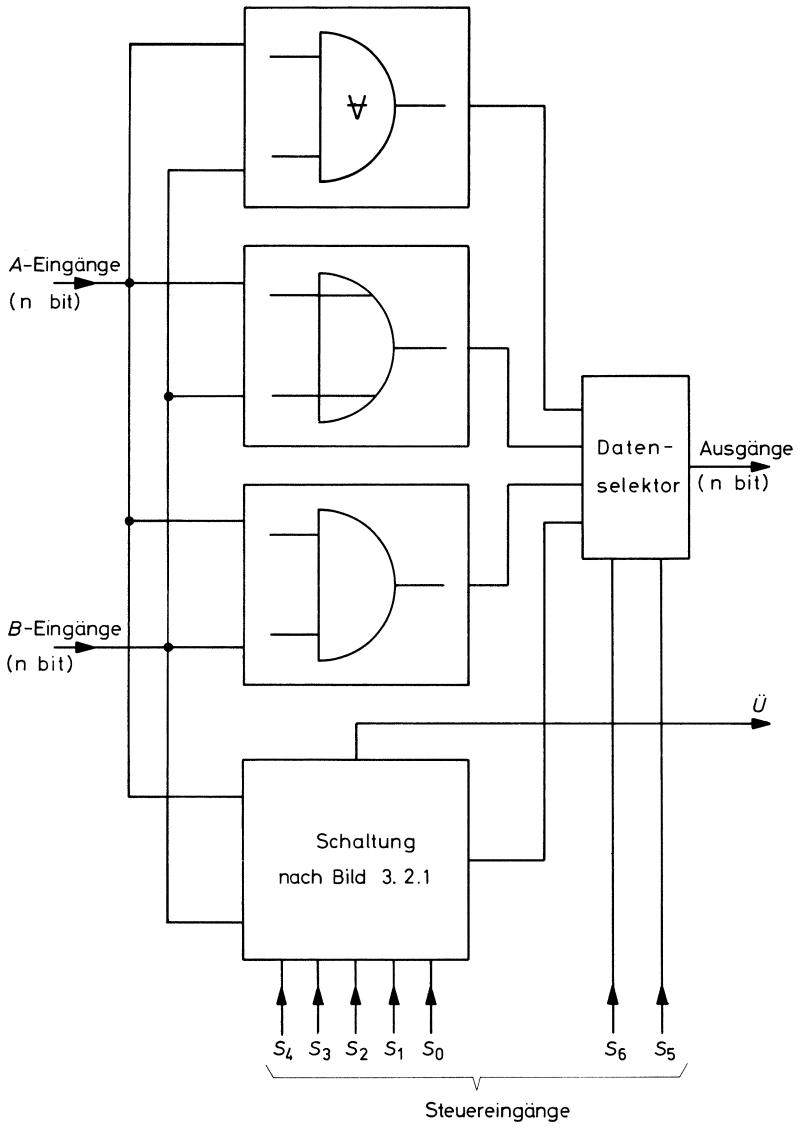


Bild 3.3.1
Arithmetic-Logic-Unit

Wenn die Steuereingänge $S_5 = S_6 = 0$ sind, wird das Addier-Subtrahierwerk durchgeschaltet, und die Funktion entspricht der nach Tab. 3.2.1. Bei einer anderen Beschaltung von S_5 und S_6 wird eine der angegebenen Verknüpfungen durchgeschaltet. In diesem Falle beeinflussen dann die Steuereingänge S_0 bis S_4 das Ergebnis nicht.
Mit 7 Steuereingängen könnte man vom Prinzip $2^7 = 128$ verschiedene Funktionen bilden, die allerdings von der Schaltung gar nicht alle geliefert werden können. Schon die Schaltung nach Bild 3.2.1 enthält Redundanzen, d. h., bestimmte Funktionen wiederholen sich. Tab. 3.2.1 zeigt, daß z. B. die Funktion 0 allein 3 mal vorkommt. Insgesamt enthält die Tabelle nur 24 verschiedene Funktionen. Mit den 3 neuen Funktionen gibt es insgesamt 27 Funktionen, von denen allerdings mehrere keine praktische Bedeutung haben (z. B. $-A - B - 2$). Wir werden uns deshalb auf 13 Funktionen beschränken und kommen dadurch nach einer Umcodierung mit 4 Steuereingängen aus. Die Umcodierung geschieht entsprechend Bild 3.3.2 mit einem ROM.

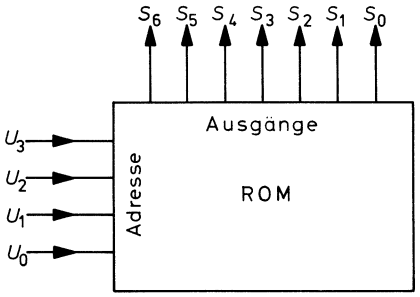


Bild 3.3.2
ROM zur Umcodierung der Steuereingänge

In Tab. 3.3.1 sind die 13 gewünschten Funktionen aufgeschlüsselt.

U_3	U_2	U_1	U_0	Funktion
0	0	0	0	A
0	0	0	1	1
0	0	1	0	$\overline{A}$
0	0	1	1	B
0	1	0	0	0
0	1	0	1	$A + 1$
0	1	1	0	$A - 1$
0	1	1	1	$A + B$
1	0	0	0	$A - B$
1	0	0	1	$A \wedge B$
1	0	1	0	$A \vee B$
1	0	1	1	$A \forall B$
1	1	0	0	-1
1	1	0	1	} für späteren Ausbau
1	1	1	0	
1	1	1	1	

Tab. 3.3.1
Umcodierte Steuerfunktionen

Da mit 4 bit 16 Funktionen möglich sind, bleiben bei der getroffenen Auswahl 3 Funktionen für den späteren Ausbau des Systems übrig. Das für die Umcodierung verwendete ROM benötigt 4 Adreßeingänge, d. h. $2^4 = 16$ Wörter zu je 7 bit. Der Inhalt jedes Wortes ist leicht zu bestimmen. Als Beispiel betrachten wir die Steuerfunktion U_3 bis $U_0 = 0\ 1\ 0\ 1$ mit $A + 1$. Die entsprechende Adresse ist 0 1 0 1. Aus Tab. 3.2.1 ist zu entnehmen, daß auf die Steuereingänge S_4 bis S_0 das bit-Muster 1 0 0 0 1 gelegt werden muß, um die Funktion $A + 1$ zu erhalten. Die Werte für S_5 und S_6 hängen direkt von der Zuordnung des Datenselektors ab. Da, wie bereits erwähnt, für die arithmetischen Funktionen $S_5 = S_6 = 0$ sein müssen, ergibt sich ein Gesamt-bit-Muster S_6 bis S_0 von 0 0 1 0 0 0 1. Dieses Muster muß in dem ROM unter der Adresse 0 1 0 1 gespeichert sein. Nach ähnlichen Überlegungen können auch die restlichen 15 Adresseninhalte bestimmt werden.
Steht ein ROM mit mehr als 7 bit Wortlänge zur Verfügung, können auch noch bestimmte Nebenfunktionen ausgeführt werden. So könnte man z. B. ein zusätzliches bit S_7 dazu

benutzen, den Übertrag der letzten Stelle dann abzuschalten, wenn es das Systemkonzept erfordert. Dies wäre z. B. angebracht im Falle der logischen Verknüpfungen, da hier ein arithmetischer Übertrag keine vernünftige Bedeutung hat. In Bild 3.3.3 ist die gesamte Schaltung der ALU dargestellt.

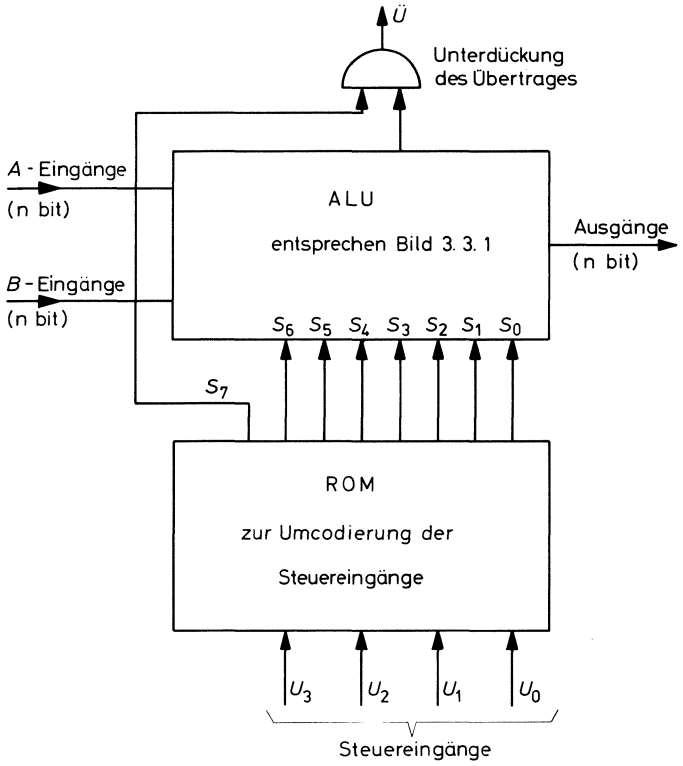


Bild 3.3.3
Komplette ALU mit Umcodierung der Steuereingänge

Exp. 2

3.4 Akkumulator

Der Akkumulator (kurz Akku) ist die nächste Erweiterungsstufe der ALU (Bild 3.4.1).

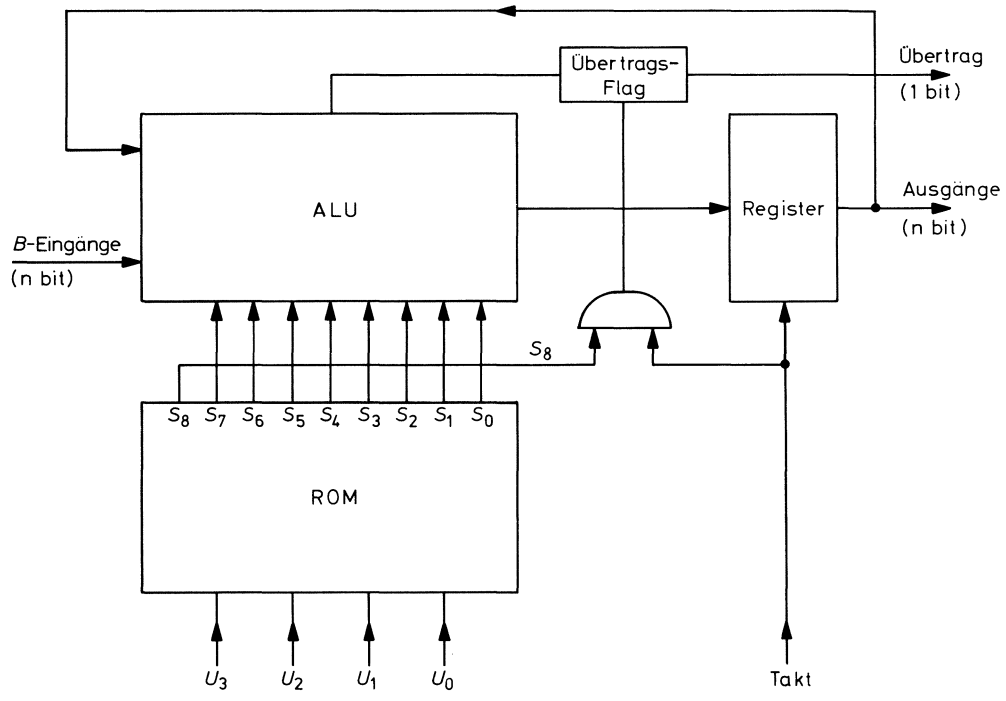


Bild 3.4.1
Akkumulator mit Übertrags-Flag

Außer den Funktionen nach Bild 3.3.3, enthält diese Schaltung als wesentliche Bestandteile noch ein Register sowie ein Übertrags-Flag. Das Register dient zum Zwischenspeichern der Ergebnisse. Hierzu wird eine der Eingangsgruppen (im Beispiel die *A*-Eingänge) mit den Ausgängen des Registers verbunden, so daß eine Art Rückkopplung entsteht. Jetzt werden die Informationen an den *B*-Eingängen mit dem Inhalt des Registers verknüpft. Durch Taktimpulse wird das Verknüpfungsergebnis in das Register geladen, wobei der alte Registerinhalt verloren geht. Damit ein möglicher Übertrag nicht nur kurzzeitig erscheint, wird er in einem Flag ebenfalls zwischengespeichert. Ob dieses Flag getaktet wird oder nicht, wird von einem zusätzlichen bit S_8 im ROM bestimmt (UND-Funktion in Bild 3.4.1). Dies ist erforderlich, da ein Übertrag nur bei sinnvollen ALU-Funktionen gespeichert wird.
Die Funktionen, die mit diesem Akkumulator ausgeführt werden können, lassen sich aus Tab. 3.3.1 ableiten. So führt diese Anordnung z. B. bei einer Steuerkombination U_3 bis U_0 von 0 1 0 1 die Funktion $A + 1$ aus. Da A durch die Rückkopplung dem jeweils vorhandenen Registerinhalt entspricht, wird in diesem Falle mit jedem Taktimpuls der Akkumulatorinhalt um 1 erhöht, d. h., der Akkumulator arbeitet bei dieser Steuerkombination als Zähler. Soll der Zähler rückwärts zählen, so muß über U_3 bis U_0 gleich 0 1 1 0 die Funktion $A - 1$ ausgelöst werden. In Tab. 3.4.1 sind die Funktionen des Akkumulators unter Berücksichtigung der Rückkopplung dargestellt. Die dabei in der Spalte Abkürzung verwendeten Ausdrücke sind allgemein gebräuchlich und von englischen Bezeichnungen abgeleitet.

U_3	U_2	U_1	U_0	Abkürzung	Funktion	Übertrags-Flag
0	0	0	0	NOP	Keine Operation	ja
0	0	0	1	SP1	Setze Akku = 1	ja
0	0	1	0	CMA	Komplementiere Akku	nein
0	0	1	1	LDA	Lade *B* in den Akku	nein
0	1	0	0	CLA	Lösche Akku	nein
0	1	0	1	INC	Incrementiere Akku	ja
0	1	1	0	DEC	Decrementiere Akku	ja
0	1	1	1	ADD	Addiere *B* in den Akku	ja
1	0	0	0	SUB	Subtrahiere *B* von Akku	ja
1	0	0	1	AND	Akku UND *B* in den Akku	ja
1	0	1	0	IOR	Akku ODER *B* in den Akku	ja
1	0	1	1	XOR	Akku EXCLUSIV-ODER in den Akku	ja
1	1	0	0	SM1	Setze Akku = −1	ja
1	1	0	1	–	–	nein
1	1	1	0	–	–	nein
1	1	1	1	–	–	nein

Tab. 3.4.1
Akkumulatorfunktionen der Schaltung nach Bild 3.4.1

Aus der Spalte Übertrags-Flag kann entnommen werden, ob das Flag getaktet wird oder nicht. In vielen Mikroprozessoren wird dieses Flag auch bei logischen Operationen getaktet. Da hierbei aber normalerweise kein Übertrag entsteht, wird das Flag gelöscht.
Sollen mit dem Akkumulator kompliziertere Funktionen ausgeführt werden, müssen diese zunächst in einfachere zerlegt werden. Für die Durchführung einer solchen Operation sind dann mehrere Taktzyklen erforderlich. Als Beispiel soll die Aufgabe $3 \cdot B$ (B = Zahl an den *B*-Eingängen) gerechnet werden. Da es keine Multiplizierfunktion gibt, muß diese durch mehrere einfachere erzeugt werden.
Hierzu sind folgende 3 Steuerkombinationen an U_3 bis U_0 erforderlich:

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

Die erste Kombination bewirkt, daß beim Takten die Zahl *B* in den Akkumulator geladen wird. Dann wird die Kombination 0 1 1 1 eingestellt und ein zweiter Taktzyklus erzeugt. Jetzt

wird B zum Akku-Inhalt addiert, d. h. $B + B$ gebildet. Mit unveränderter Steuerfunktion wird ein weiterer Taktzyklus benötigt, um zum Zwischenergebnis $B + B$ noch einmal B zu addieren. Als Ergebnis enthält der Akku $3 \cdot B$.
Durch die Auswahl geeigneter Steuerkombinationsfolgen können sehr komplizierte Ausdrücke errechnet werden. Eine solche Steuerkombination wird als **Rechnerbefehl** und eine sinnvolle Folge davon als **Rechnerprogramm** bezeichnet.

3.5 Akkumulator mit Datenspeicher

Um den Akkumulator im Rechner einsetzen zu können, muß die Möglichkeit vorhanden sein, Zwischenergebnisse abzuspeichern und sie später zurückzuholen. Dieses wird mit Hilfe eines Schreib-Lese-Speichers (RAM) erreicht (Bild 3.5.1).

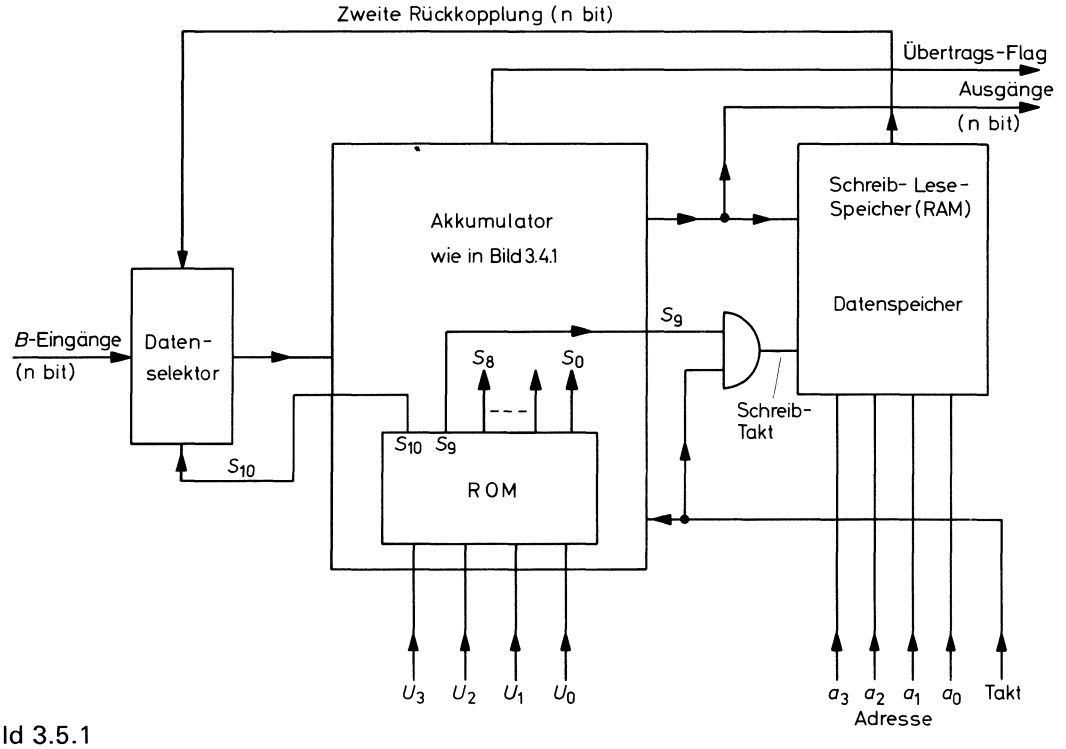


Bild 3.5.1
Akkumulator mit Datenspeicher

Bei dieser Anordnung werden die B-Eingänge über einen Datenselektor gesteuert. Mit dem Datenselektor werden entweder die B-Eingänge oder die Ausgänge des Datenspeichers durchgeschaltet. Die Akkumulatorausgänge gehen nicht nur nach außen, sondern auch zu den Eingängen des Datenspeichers. Dadurch ist es möglich, den Akku-Inhalt in den Datenspeicher oder auch Daten vom Speicher über den Datenselektor in den Akkumulator zu laden. Das Umcodierungs-ROM wird um 2 bit erweitert (S_9 und S_{10}). Die Funktionen dieser Anordnung zeigt Tab. 3.5.1.
Der zusätzliche ROM-Ausgang S_{10} dient zum Umschalten des Datenselektors. Normalerweise ist der Selektor so geschaltet, daß der Datenspeicher mit dem Akkumulator verbunden ist. Bei dem bit-Muster U_3 bis U_0 gleich 1 1 0 1 schaltet der Datenselektor auf die B-Eingänge um, die dann direkt mit dem Akkumulator verbunden sind und deren Daten in dessen Register zwischengespeichert werden. Dabei dürfen die an den Eingängen stehenden Daten in der ALU nicht verändert werden, d. h., die Steuerfunktion der ALU muß so gewählt werden, daß die B-Eingänge unverändert durchgeschaltet werden (S_6 bis S_0 = 0 0 0 1 0 0 0 nach Tab. 3.2.1). Das gleiche gilt für die Steuerfunktion U_3 bis U_0 gleich 0 0 1 1. In diesem Falle werden allerdings die Akku-Eingänge über den Datenselektor mit dem Datenspeicher verbunden.
Das zweite zusätzliche bit im ROM (S_9) dient dazu, dann einen Takt für den Datenspeicher zu erzeugen, wenn das bit-Muster U_3 bis U_0 gleich 1 1 1 0 ist. Bei dieser Steuerfunktion

U_3	U_2	U_1	U_0	a_3	a_2	a_1	a_0	Abkürzung	Funktion	Übertrags-Flag
0	0	0	0	*x*	*x*	*x*	*x*	NOP	Keine Operation	ja
0	0	0	1	*x*	*x*	*x*	*x*	SP1	Setze Akku = 1	ja
0	0	1	0	*x*	*x*	*x*	*x*	CMA	Komplementiere Akku	nein
0	0	1	1	*a*	*a*	*a*	*a*	LDA	Lade Inhalt Adresse *a a a a*	nein
0	1	0	0	*x*	*x*	*x*	*x*	CLA	Lösche Akku	nein
0	1	0	1	*x*	*x*	*x*	*x*	INC	Incrementiere Akku	ja
0	1	1	0	*x*	*x*	*x*	*x*	DEC	Decrementiere Akku	ja
0	1	1	1	*a*	*a*	*a*	*a*	ADD	Addiere Inhalt Adresse *a a a a*	ja
1	0	0	0	*a*	*a*	*a*	*a*	SUB	Subtrahiere Inhalt Adresse *a a a a*	ja
1	0	0	1	*a*	*a*	*a*	*a*	AND	Akku UND Inhalt Adresse *a a a a*	ja
1	0	1	0	*a*	*a*	*a*	*a*	IOR	Akku ODER Inhalt Adresse *a a a a*	ja
1	0	1	1	*a*	*a*	*a*	*a*	XOR	Akku EXCLUSIV-ODER Adresse *a a a a*	ja
1	1	0	0	*x*	*x*	*x*	*x*	SM1	Setze Akku = −1	ja
1	1	0	1	*x*	*x*	*x*	*x*	INP	Lade *B*-Eingänge in den Akku	nein
1	1	1	0	*a*	*a*	*a*	*a*	STA	Speichere Akku in Adresse *a a a a*	nein
1	1	1	1	*x*	*x*	*x*	*x*	–	–	–

a a a a = eine Datenspeicheradresse
x x x x = „don't care"-Zustand, d.h. beliebig

Tab. 3.5.1
Funktionen des Akkumulators mit Datenspeicher

wird nämlich der Akkumulatorinhalt in den Datenspeicher geschrieben. Damit der Akkumulatorinhalt durch diese Operation nicht verändert wird, muß an den Steuerausgängen S_6 bis S_0 des ROMs das bit-Muster 0 0 1 0 0 1 1 erzeugt werden (siehe auch Tab. 3.3.1), da dann der Akkumulatorinhalt wieder in sich zurückgeschrieben wird.

Anmerkung:
Bei Experiment 4 können Sie feststellen, daß die Steuerfunktion U_3 bis U_0 = 1 1 0 1 die *B*-Eingänge durchschaltet, während die Funktionen 1 1 1 0 und 1 1 1 1 den Akku-Inhalt in sich selbst zurückschreiben. Auf das bit-Muster 1 1 1 1 kommen wir noch zu sprechen.

Aus Tab. 3.5.1 ist zu erkennen, daß nicht alle Rechnerbefehle U_3 bis U_0 den Datenspeicher verwenden. In solchen Fällen wie z.B. 0 1 0 0 = CLA = Lösche Akku, haben die Adressen-bit a_3 bis a_0 keine Bedeutung und können deshalb beliebige Werte annehmen. Dies wird durch ein *x* gekennzeichnet.

Exp. 4

3.6 Vereinfachter Rechner

Der nächste Schritt zur Entwicklung eines vollständigen Rechners ist, die Steuermusterfolge in einem **Programmspeicher** zwischenzuspeichern. Damit ist dann letztlich ein automatischer Betrieb möglich (Bild 3.6.1).

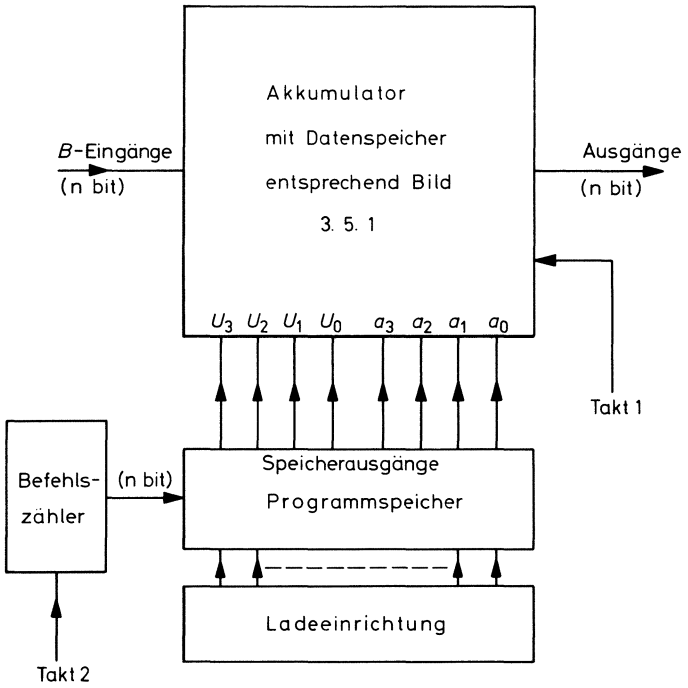


Bild 3.6.1
Anwendung von Befehlszähler und Programmspeicher

Die abzuarbeitende Folge von Steuerwörtern oder Befehlen (das Programm) wird zunächst in den Programmspeicher geladen. Dabei ist natürlich die Reihenfolge der einzelnen Befehle wichtig. Die Befehle werden deshalb im Programmspeicher mit **steigenden aufeinanderfolgenden** Adressen gespeichert. Wenn dann über einen Zähler die Programmspeicheradressen automatisch erzeugt werden, erscheinen die Befehle in der richtigen Reihenfolge und können nacheinander ausgeführt werden. Bevor man allerdings ein solches System benutzen kann, muß das Programm zunächst in den Programmspeicher geladen werden. Dabei sind 2 Möglichkeiten zu unterscheiden:
Wenn das System eine feste Aufgabe hat, z. B. Steuerung eines Aufzuges, wird im allgemeinen während des ganzen Betriebes ein festes Programm benötigt. In solchen Fällen wird ein Festwertspeicher (ROM) benutzt und das Programm beim Herstellungsprozeß des ROMs eingespeichert.
Wenn dagegen das Programm häufig geändert werden muß, z. B. bei der Entwicklung und beim Testen des Programms oder in den Experimenten dieses Lehrganges, wird als Programmspeicher ein Schreib-Lese-Speicher (RAM) benutzt. In diesem Falle muß über eine zusätzliche Logik das Programm in den Programmspeicher geladen werden. Außerdem muß eine Kontrolle des Programms möglich sein.
Obwohl einige Mikroprozessoren getrennte Daten- und Programmspeicher enthalten, wird normalerweise nur ein gemeinsamer Speicher für beide Zwecke benutzt, d. h., der Mikroprozessor hat einen gemeinsamen Adreß- und Datenraum. Der Speicher selbst kann intern als gemischtes ROM und RAM verwirklicht werden. Ein gemeinsamer Speicher hat mehrere Vorteile:
– Größere Flexibilität. Je nach Aufgabenstellung kann die Grenze zwischen Programm- und Datenkapazität vom Anwender festgelegt werden, da manche Aufgaben viel Programm und wenig Daten oder umgekehrt benötigen.
– Es werden weniger Anschlüsse benötigt (bei Mikroprozessoren besonders wichtig).
– Es können auch die Programmschritte als Daten verarbeitet werden (dieser Punkt wird in einem späteren Abschnitt näher behandelt).
Als Nachteil der Einspeicherversion kann der größere Schaltungsaufwand im Mikroprozessor genannt werden. Die Adresse muß hierbei ja entweder vom Befehlszähler oder aber vom Adressenteil des Befehles kommen können. Dafür wird ein Datenselektor benötigt. Auch die Speichereingänge benötigen einen Datenselektor. Zusätzlich wird noch ein Zwischenspeicher (Befehlsregister) für das Befehlswort benötigt, damit der Befehl so lange festgehalten wird, wie der Speicher die Daten aus- oder einliest.
Eine mögliche Realisierung zeigt Bild 3.6.2.
Damit der Rechner anhält, wenn das Programm abgearbeitet worden ist, muß am Ende eines Programms ein HALT-Befehl den Ablauf stoppen. Ohne diesen Befehl hätte das Programm

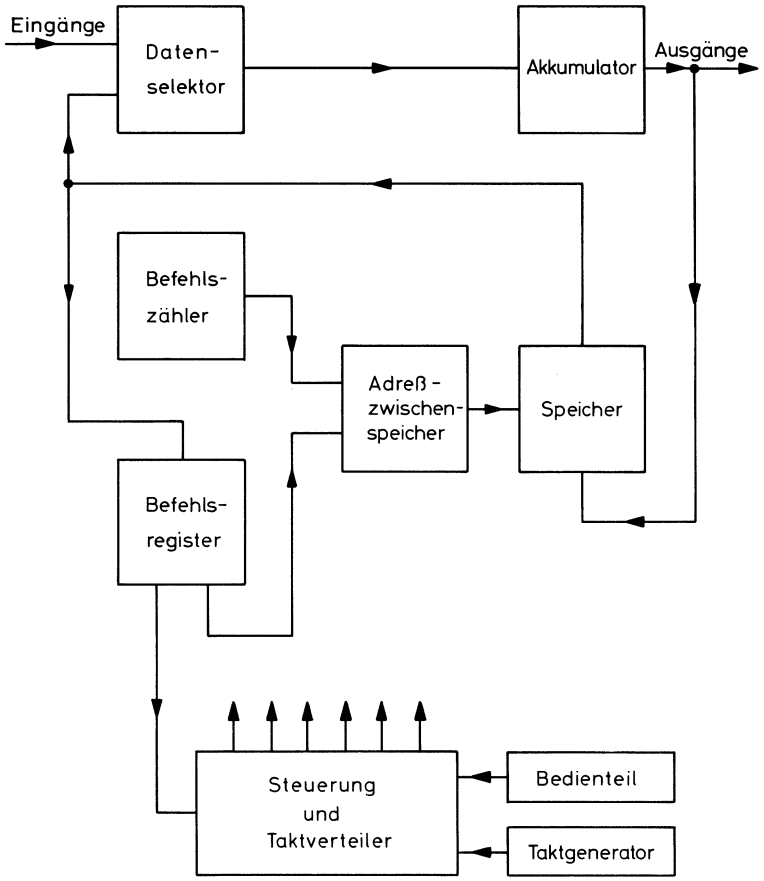


Bild 3.6.2
Einfacher Rechner mit gemeinsamem Daten- und Programmspeicher

kein Ende. Der Rechner würde auch die Daten ausführen und am Ende des Speichers wieder von vorne beginnen. Dem HALT-Befehl ist das Steuermuster U_3 bis U_0 gleich 1 1 1 1 zugeordnet.

Zur Steuerung des Rechenablaufes wird ein **Steuerwerk** benötigt. Ein solches Steuerwerk ist recht kompliziert, und wir werden uns deshalb im Rahmen dieses Lehrganges auf eine kurze Beschreibung beschränken. Die Aufgabe des Steuerwerkes ist es, die verschiedenen Taktimpulse und Steuerwörter für die einzelnen Stufen des Rechners zu erzeugen. Die zu erzeugenden Steuerimpulse hängen jeweils vom gerade auszuführenden Befehl ab. Anhand eines vereinfachten Ablaufdiagramms eines Rechnerbefehles soll das Ganze näher erläutert werden (Bild 3.6.3).

Der Befehlszähler zeigt an, welcher Befehl des Programms (z. B. Nr. 17 des Programms) ausgeführt werden soll. Der Befehlszählerinhalt wird also zuerst auf die Adreßeingänge des Speichers übertragen. Der auszuführende Befehl wird jetzt aus dem Speicher geholt und im Befehlsregister zwischengespeichert. Da der Befehl im allgemeinen aus dem Operationsteil (U_3 bis U_0) und dem Adreßteil (a_3 bis a_0) besteht, muß der Inhalt des Befehlsregisters in Operations- und Adreßteil aufgespalten werden. Aus dem Operationsteil (Op-Code) des Befehles erkennt das Steuerwerk durch eine entsprechende Logik, ob dieser Befehl eine Adresse benötigt oder nicht. Wenn nicht, veranlaßt das Steuerwerk direkt die entsprechende Operation (z. B. U_3 bis U_0 = 0 0 0 1 in Tab. 3.5.1). Wenn ja, wird der Adreßteil über den Adreßzwischenspeicher auf die Adreßeingänge des Speichers gegeben). Das unter der angesprochenen Adresse liegende Datenwort gelangt aus dem Speicher zur Ausführung der Operation in den Akkumulator. Damit ist der Befehl ausgeführt, und der Rechner kann nach Erhöhen des Befehlszählers den nächsten Befehl der Programmliste durchführen. War dieser Befehl ein HALT-Befehl (letzter Befehl jedes Programms), stoppt das Steuerwerk den Rechenablauf.

Die Durchführung eines Befehles erfordert eine bestimmte Anzahl von Taktimpulsen bzw. Taktzyklen. Die Anzahl der Taktzyklen für die verschiedenen Befehle kann verschieden groß sein.

Um bei Mikroprozessoren die vielen Befehle zu ermöglichen, ist eine große Anzahl von Verbindungen zwischen den einzelnen Baustufen erforderlich. Dieses erfordert eine große

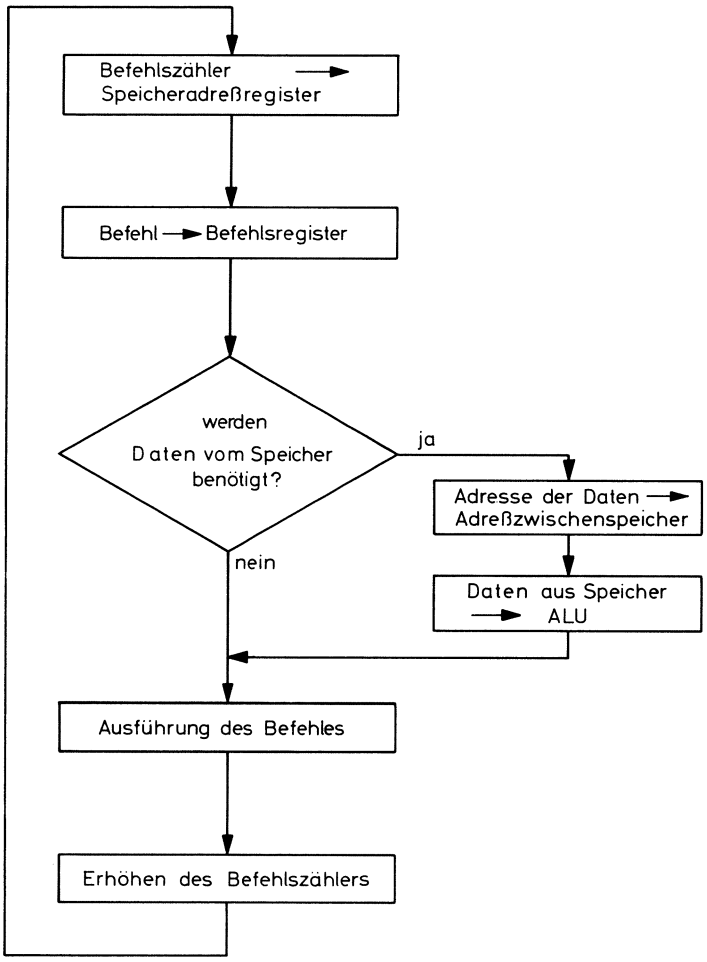


Bild 3.6.3
Ablaufdiagramm eines
Rechnerbefehles

Anzahl von Datenselektoren. Um dies zu vermeiden, wird häufig eine andere Struktur benutzt (Bild 3.6.4).
Diese Struktur basiert auf dem Konzept einer bi-direktionalen Datenbus. Unter Bus versteht man eine Datenleitung, an der mehrere Einheiten gleichzeitig angeschlossen sind. Bi-direktional

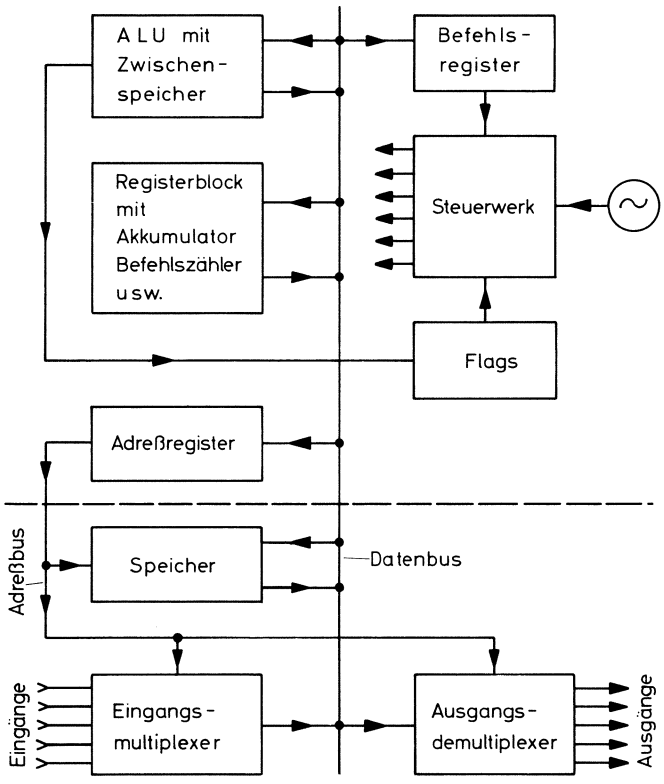


Bild 3.6.4
Busstrukturierter Rechner

bedeutet in diesem Zusammenhang, daß die Daten in beiden Richtungen übertragen werden können. Je nach System gibt es zwischen 1 und 3 Daten- und Adreßbusse, die nach einem Zeitmultiplexbetrieb gesteuert werden. Die verschiedenen Baustufen des Rechners oder Mikroprozessors können Daten auf eine Bus geben und Daten davon abrufen. Das Steuerwerk sorgt dafür, daß zum selben Zeitpunkt nur von einer Baustufe Daten auf die Bus gelangen. Nach diesem System können Daten beliebig innerhalb des Mikroprozessors übertragen werden. Da dieses Verfahren etwas langsamer ist als das Verfahren mit Einzelleitungen, wird in der Praxis häufig eine Mischung zwischen den beiden Systemen benutzt.

Exp. 5

3.7 Vollständiger Rechner

Bei jedem Mikroprozessor gibt es eine ganze Reihe zusätzlicher Befehle, um den Programmablauf zu steuern. Der wichtigste Befehl ist der **Sprungbefehl**. Mit einem Sprungbefehl wird der Programmzähler mit einem im Programm gegebenen Wert geladen. Normalerweise zählt der Programmzähler bei jedem Befehl um einen Schritt weiter, um die Befehle der Reihe nach abzuarbeiten. Mit Hilfe eines Sprungbefehles dagegen kann ein Teil des Programms (oder auch ein Datenteil) übersprungen werden. Auch ist es möglich rückwärts zu springen und einen Teil des Programms zu wiederholen. Dadurch werden Programmschleifen ermöglicht, die sehr häufig gebraucht werden. Beispiele davon werden in den nächsten Abschnitten gebracht. Damit eine Schleife nicht fortlaufend ausgeführt wird, muß es auch noch sog. **bedingte Sprünge** geben. Hier wird der Sprung zu einer gegebenen Adresse nur dann ausgeführt, wenn eine im Befehl spezifizierte Bedingung erfüllt wird, z. B. wenn das Übertrags-Flag gleich 1 ist. Andere Bedingungen werden durch zusätzliche Flags ermöglicht. Typische Flags in einem Mikroprozessorsystem sind:

- **Übertrags-Flag oder C-Flag (C = Carry)**

Dieses Flag zeigt das Verlassen des Zahlenbereiches bei der Integer Arithmetic an.

- **Arithmetischer-Übertrag-Flag oder V-Flag (V = Overflow)**

Dient zur Anzeige beim Verlassen des Zahlenbereiches bei Zweierkomplementarithmetik.

- **Null-Flag oder Z-Flag (Z = Zero)**

Zeigt den Nullzustand eines Ergebnisses an.

- **Negativ-Flag oder N-Flag**

Zeigt ein negatives Ergebnis an.

- **Paritäts-Flag oder P-Flag**

Zeigt die Parität des Ergebnisses an. Darunter versteht man hier, ob das Ergebnis eine gerade oder ungerade Anzahl von Einsen enthält.

Die aufgeführten Flags sind nicht immer alle vorhanden.
Die meisten Mikroprozessoren haben auch Befehle, um die Flags künstlich zu beeinflussen, z. B. um diese zu setzen oder zu löschen.
Ein weiterer wichtiger Befehl ist auch der Sprung-zum-Unterprogramm-Befehl. Hierbei handelt es sich um einen Befehl, nicht nur zu einer bestimmten Adresse zu springen, sondern zusätzlich die momentane Adresse abzuspeichern. Damit besteht die Möglichkeit, später auf die sog. **Rücksprungadresse** zurückzuspringen. Der Begriff Unterprogramm wird in den folgenden Abschnitten ebenfalls noch näher erläutert.

Fragen zu den Abschnitten 3.2 bis 3.6

1. Welche Funktion realisiert das Addier-Subtrahierwerk nach Bild 3.2.1, wenn die Steuereingänge S_4 bis S_0 gleich 1 0 0 1 1 sind? Wie unterscheidet sich diese Funktion von der Funktion, die durch S_4 bis S_0 gleich 1 0 0 0 0 gebildet wird? (Überlegen Sie sich, was die einzelnen UND- und EXCLUSIV-ODER-Gatter im Bild 3.2.1 tatsächlich erzeugen).

2. Welches Steuermuster S_7 bis S_0 wird bei der in Bild 3.3.3 dargestellten ALU für die Funktion $A - B$ benötigt, bzw. welchen Inhalt muß das ROM in der Adresse 1 0 0 0 aufweisen?

3. Mit dem Akkumulator nach Bild 3.4.1 soll die Funktion $1 - 2 \cdot B$ erzeugt werden. Welche Befehle sind hierfür nach Tab. 3.4.1 erforderlich, damit für diese Aufgabe ein Programm geschrieben werden kann?

4. Mit dem Akkumulator nach Bild 3.5.1 soll die EXCLUSIV-ODER-Verknüpfung der unter den Adressen 4_{16} und 5_{16} im Datenspeicher gespeicherten Informationen gebildet werden. Geben Sie hierfür das Programm an, wobei der Maschinencode hexadezimal zu schreiben ist.

5. Die Aufgabe nach 4. soll gelöst werden, ohne dafür den XOR-Befehl zu benützen. Schreiben Sie auch hierfür ein Programm.

Hinweise:

$$A \forall B = (A \wedge \overline{B}) \vee (B \wedge \overline{A})$$

Die 2 UND-Verknüpfungen sind zuerst zu bilden. Hierbei muß eine UND-Verknüpfung zwischengespeichert werden, während die andere gebildet wird. Benutzen Sie hierfür irgendeine Adresse, z. B. F_{16}.

6. Schreiben Sie ein Programm für den vereinfachten Rechner nach Bild 3.6.2, um den Ausdruck $(1 + P - Q) \forall R$ zu berechnen. Hierbei sind P, Q und R im Speicher unter den Adressen D_{16}, E_{16} und F_{16} zu finden.
Überprüfen Sie das Programm mit dem Experimentiersystem und benutzen Sie für P, Q und R irgendwelche Zahlen.

Anhang

Antworten auf die Fragen zu den Abschnitten 2.1 und 2.2

1. a) $1\,0\,1\ \ 0\,1\,1\ \ 1\,1\,1\ \ 0\,0\,1_2 = 5\,3\,7\,1_8$
   b) $1\,1\,0\ \ 0\,1\,0\ \ 1\,1\,0\ \ 1\,1\,1_2 = 6\,2\,6\,7_8$

2. a) $2\,1\,7\,0_8 = 0\,1\,0\ \ 0\,0\,1\ \ 1\,1\,1\ \ 0\,0\,0_2$
   b) $3\,5\,7\,1_8 = 0\,1\,1\ \ 1\,0\,1\ \ 1\,1\,1\ \ 0\,0\,1_2$

3. a) $1\,1\,0\,1\ \ 1\,1\,1\,1\ \ 0\,0\,1\,0\ \ 1\,1\,1\,0_2 = D\,F\,2\,E_{16}$
   b) $0\,1\,1\,0\ \ 1\,0\,0\,1\ \ 1\,0\,0\,0\ \ 0\,0\,1\,0_2 = 6\,9\,8\,2_{16}$
   c) $0\,0\,1\,1\ \ 1\,1\,0\,0\ \ 0\,1\,1\,1\ \ 1\,1\,0\,1_2 = 3\,C\,7\,D_{16}$
   d) $A\,B\,C\,D_{16} = 1\,0\,1\,0\ \ 1\,0\,1\,1\ \ 1\,1\,0\,0\ \ 1\,1\,0\,1_2$
   e) $2\,1\,7\,0_{16} = 0\,0\,1\,0\ \ 0\,0\,0\,1\ \ 0\,1\,1\,1\ \ 0\,0\,0\,0_2$
   f) $B\,7\,5\,F_{16} = 1\,0\,1\,1\ \ 0\,1\,1\,1\ \ 0\,1\,0\,1\ \ 1\,1\,1\,1_2$

4. a) $1234_{10} = 1\,0\,0\,1\,1\,0\,1\,0\,0\,1\,0_2$
   b) $5670_{10} = 1\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1\,1\,0_2$
   c) $2321_{10} = 1\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1_2$

5. a) $2115_{10} = 4\,1\,0\,3_8$
   b) $4321_{10} = 1\,0\,3\,4\,1_8$
   c) $7688_{10} = 1\,7\,0\,1\,0_8$
   d) $3821_{10} = 7\,3\,5\,5_8$

6. a) $1780_{10} = 6\,F\,4_{16}$
   b) $3666_{10} = E\,5\,2_{16}$
   c) $5230_{10} = 1\,4\,6\,E_{16}$
   d) $6744_{10} = 1\,A\,5\,8_{16}$

7. a) $0{,}3125_{10} = 0{,}0\,1\,0\,1_2$
   b) $0{,}656\,25_{10} = 0{,}1\,0\,1\,0\,1_2$
   c) $0{,}343\,75_{10} = 0{,}0\,1\,0\,1\,1_2$
   d) $0{,}140\,625_{10} = 0{,}0\,0\,1\,0\,0\,1_2$

8. a) $0{,}494\,14_{10} = 0{,}3\,7\,5_8$ (gerundet auf 3 Stellen)
   b) $0{,}406\,25_{10} = 0{,}3\,2\,0\,0\,0_8$
   c) $0{,}451_{10} = 0{,}3\,4\,7_8$ (gerundet auf 3 Stellen)
   d) $0{,}121_{10} = 0{,}076_8$ (gerundet auf 3 Stellen)

9. a) $0{,}301_{10} = 0{,}4\,D\,1_{16}$ (gerundet auf 3 Stellen)
   b) $0{,}8213_{10} = 0{,}D\,2\,4\,1_{16}$ (gerundet auf 4 Stellen)
   c) $0{,}022_{10} = 0{,}0\,5\,A_{16}$ (gerundet auf 3 Stellen)

10. a) $1\,0\,1{,}0\,1_2 = 5{,}25_{10}$
    b) $7\,2\,3{,}1\,4_8 = 467{,}1875_{10}$
    c) $A\,1{,}5\,E_{16} = 161{,}3672_{10}$

Antworten auf die Fragen zu Abschnitt 2.3

1. a)
$$\begin{array}{r} 0\,1\,0\,1\,1\,0\,1\,1 \\ +\,0\,1\,1\,0\,1\,0\,1\,1 \\ \hline 1\,1\,0\,0\,0\,1\,1\,0 \end{array}$$

   b)
$$\begin{array}{r} 1\,0\,1\,1 \\ +\,0\,0\,1\,1 \\ \hline 1\,1\,1\,0 \end{array}$$

c)
$$\begin{array}{r} 1\,1\,1\,1\,1\,1\,1\,1 \\ +\,0\,0\,0\,0\,0\,0\,0\,1 \\ \hline 1\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

d)
$$\begin{array}{r} 1\,1\,0\,1\,1\,1\,0\,0 \\ +\,1\,0\,1\,1\,1\,0\,0\,1 \\ \hline 1\,1\,0\,0\,1\,0\,1\,0\,1 \end{array}$$

2. a)
$$\begin{array}{r} 1\,0\,1\,1 \\ -\,0\,0\,1\,1 \\ \hline 1\,0\,0\,0 \end{array}$$

b)
$$\begin{array}{r} 1\,1\,0\,1\,1\,0\,1\,1 \\ -\,0\,1\,1\,0\,1\,0\,1\,1 \\ \hline 0\,1\,1\,1\,0\,0\,0\,0 \end{array}$$

c)
$$\begin{array}{r} 1\,1\,0\,0\,0\,0\,0\,0 \\ -\,1\,0\,1\,1\,0\,1\,0\,1 \\ \hline 0\,0\,0\,0\,1\,0\,1\,1 \end{array}$$

d)
$$\begin{array}{r} 1\,1\,0\,1\,1\,1\,0\,0 \\ -\,1\,0\,1\,1\,1\,0\,0\,1 \\ \hline 0\,0\,1\,0\,0\,0\,1\,1 \end{array}$$

3. a)
$$\begin{array}{l} 1\,1\,0\,0\,1\,0\,0 \cdot 1\,0\,1 \\ \hline 1\,1\,0\,0\,1\,0\,0 \\ \quad\;\; 1\,1\,0\,0\,1\,0\,0 \\ \hline 1\,1\,1\,1\,1\,0\,1\,0\,0 \end{array}$$

b)
$$\begin{array}{l} 1\,1\,0\,0\,1 \cdot 1\,0\,0\,0\,1 \\ \hline 1\,1\,0\,0\,1 \\ \quad\;\; 1\,1\,0\,0\,1 \\ \hline 1\,1\,0\,1\,0\,1\,0\,0\,1 \end{array}$$

c)
$$\begin{array}{l} 1\,0\,1\,0\,0 \cdot 1\,0\,1\,0\,0 \\ \hline 1\,0\,1\,0\,0 \\ \quad\;\; 1\,0\,1\,0\,0 \\ \quad\;\;\; 0\,0\,0\,0\,0 \\ \quad\;\;\;\; 0\,0\,0\,0\,0 \\ \hline 1\,1\,0\,0\,1\,0\,0\,0\,0 \end{array}$$

d)
$$\begin{array}{l} 1\,1\,1\,0\,1\,0\,1 \cdot 1\,1\,0\,0\,0\,1\,1 \\ \hline 1\,1\,1\,0\,1\,0\,1 \\ \quad\;\; 1\,1\,1\,0\,1\,0\,1 \\ \quad\;\;\;\; 1\,1\,1\,0\,1\,0\,1 \\ \quad\;\;\;\;\;\; 1\,1\,1\,0\,1\,0\,1 \\ \hline 1\,0\,1\,1\,0\,1\,0\,0\,1\,1\,1\,1\,1\,1 \end{array}$$

4. a)
$$\begin{array}{l} 1\,1\,1\,0\,1\,0\,0 : 1\,0\,0 = 1\,1\,1\,0\,1 \\ -\,1\,0\,0 \\ \hline \quad 0\,1\,1\,0 \\ \quad -\,1\,0\,0 \\ \quad\;\; \hline \quad\;\; 0\,1\,0\,1 \\ \quad\;\; -\,1\,0\,0 \\ \quad\;\;\; \hline \quad\;\;\; 0\,0\,1\,0\,0 \\ \quad\;\;\;\; -\,1\,0\,0 \\ \quad\;\;\;\; \hline \quad\;\;\;\;\; 0\,0\,0 \end{array}$$

b) 1 1 1 1 1 0 1 1 1 : 1 0 1 = 1 1 0 0 1 0 0, 1 0 0 1

```
 1 1 1 1 1 0 1 1 1 : 1 0 1 = 1 1 0 0 1 0 0, 1 0 0 1
-1 0 1
 0 1 0 1
-1 0 1
 0 0 0 1 0 1
   -1 0 1
     0 0 0 1 1 0
       - 1 0 1
         0 0 1 0 0 0
           -1 0 1
             0 0 1 1   (abgebrochen)
```

c) 1 1 0 1 0 1 0 1 1 : 1 0 0 1 = 1 0 1 1 1 1, 0 1 1 1

```
 1 1 0 1 0 1 0 1 1 : 1 0 0 1 = 1 0 1 1 1 1, 0 1 1 1
-1 0 0 1
 0 1 0 0 0 1
  -1 0 0 1
    1 0 0 0 0
   -1 0 0 1
     0 1 1 1 1
    -1 0 0 1
      0 1 1 0 1
     -1 0 0 1
       0 1 0 0 0 0
      -1 0 0 1
        0 1 1 1 0
       -1 0 0 1
         0 1 0 1 0
        -1 0 0 1
          0 0 0 1   (abgebrochen)
```

5. a) 0 1 0 1 1 0 1 1
   0 1 1 0 1 0 1 1
   0 1 0 0 1 0 1 1

   b) 1 0 1 1
      0 0 1 1
      0 0 1 1

   c) 1 0 1 0 1 1 0 1
      0 0 0 0 1 1 1 1
      0 0 0 0 1 1 0 1

6. a) 0 1 0 1 1 0 1 1
   0 1 1 0 1 0 1 1
   0 1 1 1 1 0 1 1

   b) 1 0 1 1
      0 0 1 1
      1 0 1 1

   c) 1 0 1 0 1 1 0 1
      0 0 0 0 1 1 1 1
      1 0 1 0 1 1 1 1

7. a) 0 1 0 1 1 0 1 1
   0 1 1 0 1 0 1 1
   0 0 1 1 0 0 0 0

b)
```
  1 0 1 1
  0 0 1 1
  -------
  1 0 0 0
```

c)
```
  1 0 1 0 1 1 0 1
  1 1 1 1 1 1 1 1
  ---------------
  0 1 0 1 0 0 1 0
```

Antworten auf die Fragen zu den Abschnitten 2.4. und 2.5

1.

	Einerkomplement	Zweierkomplement
a)	0 1 1 0	0 1 1 1
b)	1 0 0 0 1 1 0	1 0 0 0 1 1 1
c)	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
d)	0 0 0 0 0	0 0 0 0 1

2. a) negativ
   b) negativ
   c) positiv
   d) positiv

3. a)
```
    1 1 0 1 1 0 1 1
  + 1 0 0 1 0 1 0 1
  -----------------
  1 0 1 1 1 0 0 0 0
```

b)
```
      1 0 1 1
    + 1 1 0 1
    ---------
    1 1 0 0 0
```

c)
```
    0 1 1 0 1 0 1 1
  + 0 0 1 0 0 1 0 1
  -----------------
    1 0 0 1 0 0 0 0
```

4. Von einem Überlauf spricht man, wenn bei der Addition oder Subtraktion zweier Zahlen der durch die Wortlänge des Rechners gegebene Zahlenbereich überschritten wird.
Ein Übertrag entsteht dann, wenn bei der Addition binärer Zahlen in einer Stelle eine Summe größer als 1 entsteht, z. B.:

```
    1
  + 1
  ---
   10
```
Übertrag

5. a)
```
    1 1 0 1 1 1 1 1
  + 0 0 1 1 1 0 0 0
  -----------------
  1 0 0 0 1 0 1 1 1
```
Überlauf

b)
```
    0 1 0 1 1 0 1 1
  + 1 0 1 1 1 0 1 1
  -----------------
  1 0 0 0 1 0 1 1 0
```
Überlauf

c)
```
    1 0 0 0 1 0 1 1
  + 0 0 1 1 1 0 1 0
  -----------------
    1 1 0 0 0 1 0 1
```
kein Übertrag

6. a) Es entsteht ein Überlauf, da dezimal betrachtet 95 + 97 = 192 größer als 127 ist.

b) Es entsteht kein Überlauf, da dezimal betrachtet −24 + (−9) = −33 die Zahl −128 im Betrag nicht überschreitet.

c) Es entsteht kein Überlauf, da dezimal betrachtet −102 − (+23) = −102 − 23 = −125 im Betrag noch unter 128 liegt.

Antworten auf die Fragen zu Abschnitt 2.7

1. a)
```
  0 0 0 1   0 1 0 1   1 0 0 1  ≙ 159₁₀
+ 0 0 1 1   0 1 1 1   0 0 1 0  ≙ 372₁₀
                      1 0 1 1  → Pseudotetrade, also + 0 1 1 0
                    + 0 1 1 0
                      1 0 0 0 1 → Ergebnis 1. Stelle

    0 0 0 1   0 1 0 1
  + 0 0 1 1   0 1 1 1
              1 1 1 1
              1 1 0 1  → Pseudotetrade, also + 0 1 1 0
            + 0 1 1 0
              1 0 0 1 1 → Ergebnis 2. Stelle

    0 0 0 1
  + 0 0 1 1
        1 1 1
    0 1 0 1  → Ergebnis 3. Stelle
```

Gesamtergebnis: 0 1 0 1 0 0 1 1 0 0 0 1 ≙ 531_{10}

Kontrolle über Dezimalzahlen: 159 + 372 = 531

b)
```
  0 0 0 1   1 0 0 0   0 1 1 1  ≙ 187₁₀
+ 1 0 0 1   1 0 0 1   0 1 1 0  ≙ 996₁₀
                      1 1 0 1  → Pseudotetrade, also + 0 1 1 0
                    + 0 1 1 0
                      1 0 0 1 1 → Ergebnis 1. Stelle

    0 0 0 1   1 0 0 0
    1 0 0 1   1 0 0 1
            +       1 1
          1 0 0 1 0  → Übertrag in nächste BCD-Stelle, also + 0 1 1 0
        +   0 1 1 0
            1 0 0 0  → Ergebnis 2. Stelle

    0 0 0 1
    1 0 0 1
  +       1 1
    1 0 1 1  → Pseudotetrade, also + 0 1 1 0
  + 0 1 1 0
  1 0 0 0 1  → Übertrag in 4. Stelle
```

Gesamtergebnis: 0 0 0 1 0 0 0 1 1 0 0 0 0 0 1 1 ≙ 1183_{10}

Kontrolle über Dezimalzahlen: 187 + 996 = 1183

Antworten auf die Fragen zu den Abschnitten 3.2 bis 3.6

1. Bedingt durch $S_4 = 1$ und $S_3 = 0$ werden die A-Eingänge durchgeschaltet, während die B-Eingänge durch die UND-Gatter gesperrt werden. Da $S_2 = 0$ ist, werden die A-Eingänge nicht komplementiert. Durch $S_1 = 1$ werden die 4 Nullen an den unteren UND-Gattern auf 1 1 1 1 addiert. Da in der Zweierkomplementarithmetik 1 1 1 1 gleich −1 ist, ist diese Summe $A - 1$. Da $S_0 = 1$ ist, wird hierzu 1 addiert, so daß die Ausgangsfunktion A ist (siehe auch Tab. 3.2.1).
Im Fall S_4 bis S_0 gleich 1 0 0 0 0 entsteht laut Tab. 3.2.1 auch A als Ausgangsfunktion. Entsprechend der allgemeinen Beziehung für die Zweierkomplementarithmetik $-A = \overline{A} + 1$ wird B = 0 0 0 0 als 1 1 1 1 + 1 = 1 0 0 0 0 interpretiert. Es wird also die Funktion A + 1 0 0 0 0 gebildet. Dies bedeutet, daß ein Übertrag in die nächste Stelle entsteht.

2. Da es sich um eine arithmetische Operation handelt, muß ein eventueller Übertrag erscheinen, d.h., S_7 muß 1 sein. S_6 und S_5 müssen beide 0 sein, damit der Datenselektor (siehe Bild 3.3.1) die arithmetischen Funktionen selektiert. S_4 bis S_0 müssen nach Tab. 3.2.1 1 1 0 1 1 sein, damit die Funktion $A - B$ gebildet wird. Damit hat das Gesamtmuster S_7 bis S_0 die Form 1 0 0 1 1 0 1 1.

3. Da diese Funktion nicht direkt vorkommt, muß sie über mehrere einfachere Schritte verwirklicht werden. Eine Möglichkeit ist die, über den Befehl SP1 den Akkumulator auf 1 zu setzen. Mit dem SUB-Befehl kann dann hiervon B 2 mal abgezogen werden.
Das Programm hat dann folgende Form:

Steuerfunktion	mnemonische Abkürzung
0 0 0 1	SP1
1 0 0 0	SUB
1 0 0 0	SUB

Probieren Sie dieses Programm auf dem Experimentiersystem aus.

Anmerkung:
Wenn der Akkumulator am Anfang 0 enthalten hätte, könnte die 1 auch mit dem INC-Befehl erzeugt werden. Allgemein darf man aber keine Annahme über den Anfangszustand des Rechners treffen.

4.

Maschinencode	Befehl
3 4	LDA 4
B 5	XOR 5

5. Wenn ein Mikroprozessor keinen XOR-Befehl besitzt, muß diese Funktion in einfachere Funktionen zerlegt werden. Zuerst wird $A \wedge \overline{B}$ gebildet und zwischengespeichert, dann $B \wedge \overline{A}$. Diese beiden Teilfunktionen werden dann über ODER miteinander verknüpft.

Maschinencode	Befehl	Kommentar
3 5	LDA 5	lade B
2 0	CMA	bilde $\overline{B}$
9 4	AND 4	bilde $A \wedge \overline{B}$
E F	STA F	zwischenspeichern
3 4	LDA 4	lade A
2 0	CMA	bilde $\overline{A}$
9 5	AND 5	bilde $B \wedge \overline{A}$
A F	IOR F	bilde $A \vee B$

6.

Adresse	Maschinencode	Befehl	Kommentar
00	1 0	SP1	setze Akku = 1
01	7 D	ADD D	bilde 1 + P
02	8 E	SUB E	bilde 1 + P − Q
03	B F	XOR F	bilde (1 + P − Q) ∀ R
04	F 0	HLT	halte an

Allgemeine Hinweise zu den Experimenten

Das Experimentiersystem enthält einen vollständigen Rechner. Das Kernstück ist der INTEL 8080 Mikroprozessor. Als Speicher sind ein 1-k-ROM (1024 Wörter à 8 bit) und ein RAM mit 256 Wörtern à 8 bit vorhanden. In dem ROM sind 7 Programme zur Simulierung von verschiedenen Systemen fest abgespeichert. Welches Programm ablaufen soll, kann mit dem SYSTEM-Schalter (BCD-Schalter auf der linken Seite) festgelegt werden. Den 7 Programmen sind die Nummern 0 bis 6 zugeordnet. Die Stellung 7 des SYSTEM-Schalters ist für eine eventuelle Erweiterung des Systems vorgesehen, die Stellungen 8 und 9 werden nicht verwendet. Bei jeder Schalterstellung können mehrere Experimente durchgeführt werden. In den Stellungen 4, 5 und 6 stehen sogar 3 unterschiedliche Rechner zur Verfügung, die für beliebig viele Experimente benutzt werden können.

Ein Programm wird mit der RESET-Taste gestartet. Diese Taste entspricht in etwa der Löschtaste eines Taschenrechners und **muß am Anfang jedes Experimentes gedrückt werden.** Mit den restlichen Schiebeschaltern können Daten und Steuerinformationen eingegeben werden.

Die Schaltergruppe C_4 bis C_0 wird zur Steuerung des Experimentierablaufes benötigt.

Die Schaltergruppen A_7 bis A_0 und B_7 bis B_0 werden bis auf einige Spezialfälle für die Daten oder Programmeingabe benutzt.

Bei allen Experimenten gelten folgende Festlegungen:

Schalter oben ≙ logisch 1
Schalter unten ≙ logisch 0

Die 2 von 8 Leuchtdioden dienen zur Anzeige der Rechenergebnisse sowie zur Anzeige interner Schaltzustände. Hier gelten folgende Festlegungen:

Leuchtdiode leuchtet ≙ logisch 1
Leuchtdiode dunkel ≙ logisch 0

Alle Schalter, die bei einem bestimmten Experimentiervorgang nicht benötigt werden, müssen auf log. 0 geschaltet werden.

Es ist zu empfehlen, daß zu Beginn eines Experimentes alle Schalter auf log. 0 geschaltet werden, bevor die RESET-Taste gedrückt wird. Ausnahmen hiervon werden bei den einzelnen Experimenten angegeben.

Bei falscher Schalterbetätigung können grundsätzlich keine Schäden am Experimentiersystem entstehen. Allerdings können dadurch die selbst eingegebenen Programme und Daten verändert werden, so daß ein falsches Ergebnis entsteht. Bei umfangreichen und komplizierten Experimenten kann eine falsche Betätigung viel Zeit kosten.

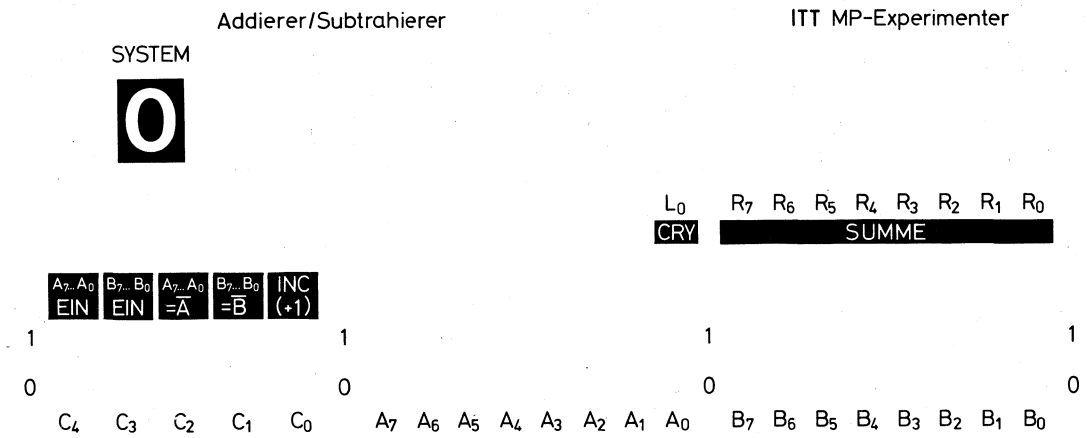
Die grundsätzliche Experimentiervorbereitung ist folgende:

1. Die in der Experimentieranweisung angegebene Schablone auflegen
2. SYSTEM-Schalter auf das verlangte Programm einstellen
3. Alle Schiebeschalter auf Null (unten) stellen
4. RESET-Taste drücken

In den Experimentieranleitungen ist zu Beginn jedes Experimentes auch noch einmal die entsprechende Schablone angegeben. Aus der Schablone kann die Bedeutung bzw. Funktion der einzelnen Schalter entnommen werden.

Zu jedem Experimentierprogramm werden ein oder auch mehrere Musterexperimente durchgeführt. Danach sind Aufgabenstellungen gegeben, die Sie selbst lösen sollen. Die Musterlösungen finden Sie im Experimentieranhang.

Experiment 1: Arbeitsweise eines 8-bit-Ripple-Carry-Addierers



Nach der Experimentiervorbereitung Schalter C_4 und C_3 auf 1. Damit können die an den Schaltern A_7 bis A_0 und B_7 bis B_0 eingestellten Informationen in das System gelangen. Mit den Schaltern $C_2 = \overline{A}$ und $C_1 = \overline{B}$ können die eingegebenen Informationen komplementiert werden (Einerkomplement). Der Schalter C_0 = INC legt bei 1 eine 1 auf den INC-Eingang. Die Schalter C_2 bis C_0 sind zunächst in der Stellung 0 zu belassen. Das Ergebnis der Addition erscheint in den rechten 8 LEDs (R_7 bis R_0). Die LED L_0 in der linken Lampengruppe zeigt einen Übertrag (Carry) an. Bei diesen Programmen haben die LEDs L_7 bis L_1 keine Bedeutung.

1. Beispiel:

A-Schalter:	0 0 0 0 1 0 1 0	$\triangleq 10_{10}$
+ B-Schalter:	0 0 0 0 0 0 1 1	$\triangleq 3_{10}$
Ergebnis:	0 0 0 0 1 1 0 1	$\triangleq 13_{10}$
	R_7 bis R_0	

2. Beispiel:

A-Schalter:	1 1 1 1 1 1 1 1	$\triangleq 255_{10}$
+ B-Schalter:	0 0 0 0 0 0 1 0	$\triangleq 2_{10}$
Ergebnis:	1 | 0 0 0 0 0 0 0 1	$\triangleq 257_{10}$
	L_0 R_7 bis R_0	

3. Beispiel:
Darstellung von negativen Zahlen über das Zweierkomplement

a)

A-Schalter:	0 0 0 0 1 0 1 0	$\triangleq 10_{10} = A$
$\overline{A}$-Schalter 1:	1 1 1 1 0 1 0 1	$\triangleq \overline{A}$
INC-Schalter 1:	1 1 1 1 0 1 1 0	$\triangleq -A = \overline{A} + 1$
	R_7 bis R_0	

b)

A-Schalter:	1 1 1 1 1 1 1 1	$\triangleq 255_{10} = A$
$\overline{A}$-Schalter 1:	0 0 0 0 0 0 0 0	$\triangleq \overline{A}$
INC-Schalter 1:	0 0 0 0 0 0 0 1	$\triangleq -A = \overline{A} + 1$
	R_7 bis R_0	

c)

A-Schalter:	0 0 0 0 0 0 0 1	$\triangleq 1_{10} = A$
$\overline{A}$-Schalter 1:	1 1 1 1 1 1 1 0	$\triangleq \overline{A}$
INC-Schalter 1:	1 1 1 1 1 1 1 1	$\triangleq -A = \overline{A} + 1$
	R_7 bis R_0	

4. Beispiel:
Subtraktion über das Zweierkomplement nach der Beziehung $A + (-B)$

a)

A-Schalter:	0 0 0 0 1 0 1 0	$\triangleq 10_{10}$
B-Schalter:	0 0 0 0 0 1 0 1	$\triangleq 5_{10}$
$\overline{B}$-Schalter 1:	1 | 0 0 0 0 0 1 0 0	$\triangleq A + \overline{B}$
	L_0 R_7 bis R_0	
INC-Schalter 1:	1 | 0 0 0 0 0 1 0 1	$\triangleq A + \overline{B} + 1 = A - B$
	L_0 R_7 bis R_0	
Ergebnis:	0 0 0 0 0 1 0 1	$\triangleq 5_{10}$
	R_7 bis R_0	

b)

A-Schalter:	0 0 0 0 0 1 0 1	$\triangleq 5_{10}$
B-Schalter:	0 0 0 0 1 0 1 0	$\triangleq 10_{10}$
$\overline{B}$-Schalter 1:	1 1 1 1 1 0 1 0	$\triangleq A + \overline{B}$
	R_7 bis R_0	
INC-Schalter 1:	1 1 1 1 1 0 1 1	$= A + \overline{B} + 1 = A - B$
	R_7 bis R_0	
Ergebnis:	1 1 1 1 1 0 1 1	$= -5_{10}$
	R_7 bis R_0	

Kontrollieren Sie dieses Ergebnis entsprechend Beispiel 3.

Lösen Sie experimentell folgende Aufgaben (Angaben in Dezimalzahlen).

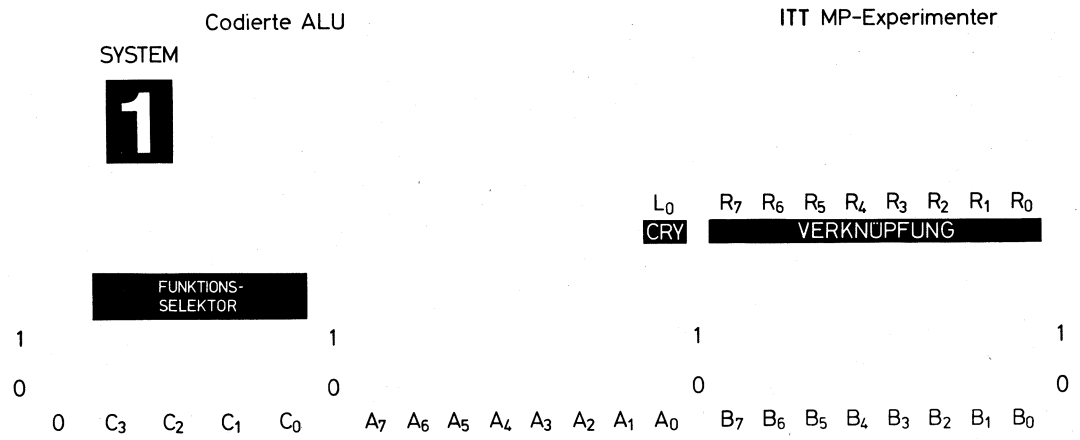
1. a) $125_{10} + 40_{10} =$
   b) $184_{10} + 100_{10} =$

2. Stellen Sie über das Zweierkomplement folgende Zahlen dar:
   a) -120_{10}
   b) -12_{10}
   c) -2_{10}

3. Lösen Sie folgende Subtraktionsaufgaben über die Beziehung A + (−B):
   a) $120_{10} - 100_{10} =$
   b) $130_{10} - 140_{10} =$

Die Lösungen finden Sie auf Seite E21.

Experiment 2: Arbeitsweise einer codierten ALU



Für dieses Experiment ist eine 8-bit-Version der in Bild 3.3.3 dargestellten codierten ALU simuliert worden. Die Funktionen U_3 bis U_0 in Tab. 3.3.1 werden mit den Schaltern C_3 bis C_0 festgelegt. Diese Tabelle ist auch auf der Karte „Codierte ALU" zu finden.
Nach der Experimentiervorbereitung (alle Schalter 0, RESET-Taste drücken) überprüfen wir zunächst alle Funktionen nach Tab. 3.3.1.

	C_3	C_2	C_1	C_0		
1.	0	0	0	0	→	Ein an den A-Schaltern eingestelltes bit-Muster erscheint in der Anzeige R_7 bis R_0. Die B-Schalter haben keine Funktion.
2.	0	0	0	1	→	Unabhängig von der Stellung der A- und B-Schalter erscheint in R_7 bis R_0 eine 1 in der Stelle R_0.
3.	0	0	1	0	→	In R_7 bis R_0 erscheint das Einerkomplement des an A_7 bis A_0 eingestellten bit-Musters. B_7 bis B_0 haben keine Funktion.
4.	0	0	1	1	→	Ein an B_7 bis B_0 eingestelltes bit-Muster erscheint in R_7 bis R_0. A_7 bis A_0 haben keine Auswirkung.
5.	0	1	0	0	→	Beide Schalterreihen haben keine Funktion.
6.	0	1	0	1	→	In R_7 bis R_0 erscheint die an A_7 bis A_0 eingestellte Zahl plus 1.
7.	0	1	1	0	→	In R_7 bis R_0 erscheint die an A_7 bis A_0 eingestellte Zahl minus 1.
8.	0	1	1	1	→	In R_7 bis R_0 erscheint die Summe der in A_7 bis A_0 und B_7 bis B_0 eingestellten Zahlen. Ein Übertrag erscheint in L_0.
9.	1	0	0	0	→	In R_7 bis R_0 erscheint die Differenz $A - B$ der an den Schaltern A_7 bis A_0 und B_7 bis B_0 eingestellten Zahlen (Zweierkomplement beachten!).
10.	1	0	0	1	→	Die in A_7 bis A_0 und B_7 bis B_0 stehenden Informationen werden bit-weise miteinander UND-verknüpft.
11.	1	0	1	0	→	Die in A_7 bis A_0 und B_7 bis B_0 stehenden Informationen werden bit-weise miteinander ODER-verknüpft.
12.	1	0	1	1	→	Die in A_7 bis A_0 und B_7 bis B_0 stehenden Informationen werden bit-weise miteinander EXCLUSIV-ODER-verknüpft.
13.	1	1	0	0	→	Unabhängig von A_7 bis A_0 und B_7 bis B_0 erscheint in R_7 bis R_0 −1 (Zweierkomplement beachten!).

Die angesprochenen Funktionen sind alle ausreichend bekannt und bedürfen daher keiner weiteren Erläuterung. Nachfolgend einige Übungsbeispiele:

1. Beispiel: $-30_{10} - 64_{10} =$

Diese Aufgabe wird durch eine Addition der Zweierkomplemente gelöst

$$-A + (-B) = -A - B$$

Funktion C_3 bis C_0:	0 1 1 1	
A-Schalter:	1 1 1 0 0 0 1 0	$\triangleq -30_{10}$
B-Schalter:	1 1 0 0 0 0 0 0	$\triangleq -64_{10}$
	1 | 1 0 1 0 0 0 1 0	
	L_0 R_7 bis R_0	
Ergebnis:	1 0 1 0 0 0 1 0	$= -94_{10}$

2. Beispiel: $34_{10} - 128_{10} =$

Funktion C_3 bis C_0:	1 0 0 0	
A-Schalter:	0 0 1 0 0 0 1 0	$\triangleq 34_{10}$
$-B$-Schalter:	1 0 0 0 0 0 0 0	$\triangleq 128_{10}$
	1 | 1 0 1 0 0 0 1 0	
	L_0 R_7 bis R_0	
Ergebnis:	1 0 1 0 0 0 1 0	$= -94_{10}$

3. Beispiel:
In A_7 bis A_0 ist folgendes bit-Muster eingestellt:

0 1 1 0 1 0 0 1

Dieses bit-Muster ist in ein Muster der Form

0 0 0 0 1 0 0 1

abzuändern, ohne dabei die Schalterstellung A_7 bis A_0 zu ändern. Diese Aufgabenstellung, das Ausblenden von bestimmten bit oder bit-Gruppen, läßt sich mit der UND-Funktion leicht lösen, indem an den B-Schaltern eine sog. Maske eingestellt wird. In unserem Beispiel wird:

Funktion C_3 bis C_0:	1 0 0 1	
A-Schalter:	0 1 1 0 1 0 0 1	
B-Schalter:	0 0 0 0 1 1 1 1	→ Maske
	0 0 0 0 1 0 0 1	
	R_7 bis R_0	

Eine Änderung der Schalterstellung A_7 bis A_4 hat keinen Einfluß auf das Ergebnis in R_7 bis R_0.

4. Beispiel:
An den A-Schaltern ist folgendes bit-Muster eingestellt:

0 1 1 0 1 0 0 1

Dieses bit-Muster ist in ein Muster der Form

1 0 0 1 1 0 0 1

abzuändern, ohne dabei die Schalterstellung A_7 bis A_0 zu ändern. Diese Aufgabenstellung läßt sich mit der EXCLUSIV-ODER-Funktion lösen.

Funktion C_3 bis C_0:	1 0 1 1
A-Schalter:	0 1 1 0 1 0 0 1
B-Schalter:	1 1 1 1 0 0 0 0
	1 0 0 1 1 0 0 1
	R_7 bis R_0

5. Beispiel:
Ein an den A-Schalter eingestelltes bit-Muster der Form

1 0 0 0 0 1 0 0

soll in ein Muster der Form

1 0 1 1 0 1 0 1

geändert werden, ohne dabei die Schalterstellung A_7 bis A_0 abzuändern. Diese Aufgabe läßt sich über die ODER-Verknüpfung lösen.

Funktion C_3 bis C_0:	1 0 1 0
A-Schalter:	1 0 0 0 0 1 0 0
B-Schalter:	0 0 1 1 0 0 0 1
	1 0 1 1 0 1 0 1
	R_7 bis R_0

Lösen Sie folgende Aufgaben experimentell. Die Daten sind im Hexadezimalsystem angegeben.

1. a) $3\,4_{16} + 2\,1_{16} =$
   b) $1\,4_{16} + (-4\,8_{16}) =$

2. a) $8\,1_{16} - 7\,5_{16} =$
   b) $7\,6_{16} - 8\,4_{16} =$

3. a) Ein bit-Muster in A_7 bis A_0 der Form

   1 1 0 0 1 1 1 1

   ist in die Form

   1 1 0 0 1 1 0 0

   umzuformen, ohne A_7 bis A_0 zu ändern.

   b) Ein bit-Muster in A_7 bis A_0 der Form

   1 1 0 1 0 1 1 0

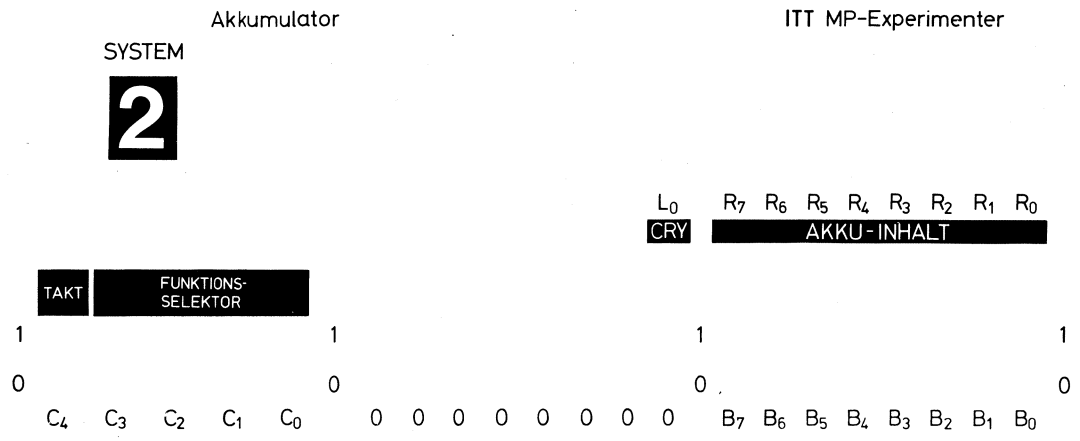
   ist in die Form

   0 0 0 0 0 1 1 0

   umzuformen, ohne dabei A_7 bis A_0 zu ändern.

Die Lösungen dieser Aufgaben finden Sie auf Seite E21.

Experiment 3: Arbeitsweise eines Akkumulators



Mit diesem Programm wird der in Bild 3.4.1 dargestellte Akkumulator simuliert. Die in Tab. 3.4.1 gezeigten Funktionen werden mit den Schaltern C_3 bis C_0 ausgewählt. Der Schalter C_4 dient jetzt als Taktschalter. Durch einmaliges hin- und herschieben wird ein Ergebnis in das Register übernommen und zur Anzeige gebracht. Die A-Schalter werden in diesem Beispiel nicht gebraucht, weil die A-Eingänge der im Akkumulator enthaltenen ALU mit den Ausgängen des Registers verbunden sind. Das Ergebnis bzw. der momentane Inhalt des Akkus wird wieder in R_7 bis R_0 angezeigt, ein Übertrag in L_0.
Zu Beginn eines Experimentes ist der Akku-Inhalt beliebig. Er muß daher zunächst auf Null gebracht werden (Vergleich: Löschtaste eines Taschenrechners). Das Löschen erfolgt laut Tab. 3.4.1 über die CLA-Funktion U_3 bis U_0 = C_3 bis C_0 = 0 1 0 0. Der Vorgang ist folgender:

1. C_3 bis C_0 auf 0 1 0 0 stellen
2. Schalter C_4 einmal takten (einmal hin- und herschieben)
3. Kontrollieren, ob alle LEDs R_7 bis R_0 einschließlich L_0 ausgehen.

Da die A-Schalter keine Funktion haben, werden die entsprechenden Operationen immer zwischen dem Akku-Inhalt und den an den B-Schaltern eingestellten Informationen durchgeführt. Hierzu einige Beispiele:

1. Beispiel:
Die Hexadezimalzahlen 1 5_{16} und 3 3_{16} werden addiert.

Ablauf:
1. 1 5_{16} an den B-Schaltern einstellen
2. LDA (Lade den Zustand der B-Schalter in den Akku) mit C_3 bis C_0 gleich 0 0 1 1 einstellen
3. Mit einem Takt (Schalter C_4) Inhalt der B-Schalter in den Akku laden
4. 3 3_{16} an den B-Schaltern einstellen
5. Mit C_3 bis C_0 = 0 1 1 1 den Befehl ADD wählen
6. Mit C_4 Takten
7. In R_7 bis R_0 steht jetzt das Ergebnis mit 0 1 0 0 1 0 0 0 $\triangleq$ 4 8_{16}

2. Beispiel:
Das Zweierkomplement der Binärzahl 0 1 0 1 1 1 0 1 wird gebildet.

Ablauf:
1. Die Zahl 0 1 0 1 1 1 0 1 an den B-Schaltern einstellen
2. Mit C_3 bis C_0 den Befehl LDA = 0 0 1 1 einstellen
3. Mit C_4 einmal takten
4. Mit C_3 bis C_0 den Befehl CMA = 0 0 1 0 einstellen
5. Mit C_4 takten. In R_7 bis R_0 erscheint jetzt 1 0 1 0 0 0 1 0, also das Einerkomplement der vorher eingestellten Zahl
6. Mit C_3 bis C_0 den Befehl INC = 0 1 0 1 einstellen

7. Mit C_4 takten. In R_7 bis R_0 erscheint die Zahl 1 0 1 0 0 0 1 1, das Zweierkomplement der eingegebenen Zahl

3. Beispiel:
Lösen der Aufgabe $5\ 1_{16} - 4\ B_{16}$

Ablauf:
1. Mit dem Befehl CLA den Akkumulator löschen (siehe vorher)
2. $5\ 1_{16}$ an B_7 bis B_0 einstellen und über Befehl LDA in den Akku laden
3. $4\ B_{16}$ an B_7 bis B_0 einstellen
4. Mit Befehl SUB = 1 0 0 0 die an B_7 bis B_0 eingestellte Zahl von der im Akku befindlichen Zahl subtrahieren. Als Ergebnis erscheint in R_7 bis R_0 die Zahl $0\ 6_{16}$ (kontrollieren Sie selbst das Ergebnis über Dezimalzahlen nach).

4. Beispiel:
Der Akkumulator wird als Aufwärtszähler betrieben, der bei 0 beginnt und mit jedem Takt um 1 weiterzählt.

Ablauf:
1. Mit Befehl CLA den Akkumulator löschen
2. Den Befehl INC einstellen
3. Mit C_4 das System takten. In R_7 bis R_0 erscheint das jeweilige Zählergebnis

5. Beispiel:
Zwischen den beiden bit-Kombinationen 1 0 0 0 0 0 0 1 und 1 1 1 1 0 0 0 1 wird die EXCLUSIV-ODER-Verknüpfung gebildet.

Ablauf:
1. Mit Befehl CLA Akkumulator löschen
2. Kombination 1 0 0 0 0 0 0 1 an B_7 bis B_0 einstellen
3. Mit Befehl LDA Information B_7 bis B_0 in den Akkumulator laden
4. Kombination 1 1 1 1 0 0 0 1 an B_7 bis B_0 einstellen
5. Befehl XOR = 1 0 1 1 einstellen und takten. In R_7 bis R_0 erscheint das Ergebnis 0 1 1 1 0 0 0 0

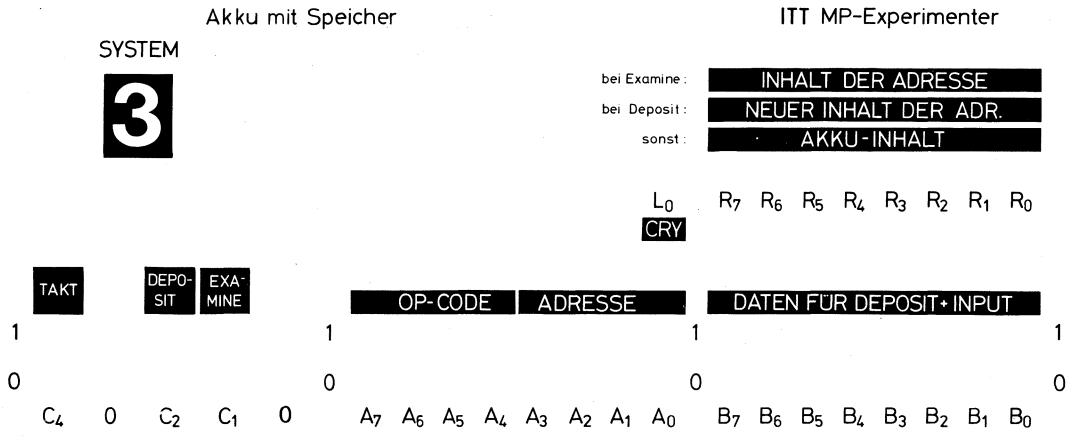
Aufgaben:

1. Subtrahieren Sie folgende Aufgaben:
   a) $F\ 8_{16} - C\ C_{16} =$
   b) $1\ 5_{16} - 2\ 2_{16} =$
   c) $-4\ 8_{16} - 3\ 2_{16} =$

2. Bilden Sie die ODER-Verknüpfung zwischen folgenden bit-Kombinationen:
   a) 1 1 0 0 0 1 1 0
   b) 0 1 0 1 0 0 0 0

Die Lösungen finden Sie auf Seite E22.

Experiment 4: Arbeitsweise eines Akkumulators mit Datenspeicher



Das System 3 enthält grundsätzlich die gleichen Funktionen wie das System 2. Der Unterschied besteht darin, daß entsprechend Bild 3.5.1 die Daten nicht mehr von den *B*-Schaltern kommen sondern von einem RAM. Mit dem neuen Befehl STA (Speichere Akku-Inhalt in Adresse *a a a a* ab), können die Daten im RAM zurückgeschrieben werden. Mit dem Befehl INP (Lade *B*-Eingänge in den Akku) werden jetzt die Daten an B_7 bis B_0 in den Akkumulator eingelesen (entspricht Befehl LDA in System 2). Bei allen Befehlen, die den Speicher nutzen, muß jetzt eine bestimmte Adresse spezifiziert werden. Der hier verwendete Speicher hat eine Kapazität von 16 Wörtern à 8 bit. Damit jedes dieser 16 Wörter spezifiziert bzw. adressiert werden kann, werden 4 bit benötigt. Damit besteht ein Befehl jetzt aus insgesamt 8 bit. Hiervon legen 4 bit die Funktion fest, die ausgeführt werden soll. Sie bilden den sog. OP-Code (Operation-Code). Die anderen 4 bit bestimmen die Speicheradresse. Aus diesem Grunde werden jetzt die *A*-Schalter für die Befehlseingabe benutzt.
Aus Tab. 3.5.1 geht hervor, daß es Befehle gibt, die unbedingt die Angabe einer Adresse benötigen (*a a a a*), und andere, die ohne spezielle Adresse auskommen (*x x x x*).
Bevor Befehle, die Daten aus dem Speicher unter einer bestimmten Adresse benötigen, benutzt werden können, müssen die entsprechenden Daten in den Speicher geladen werden. Das Laden einer bestimmten Speicheradresse erfolgt mit dem Schalter C_2 DEPOSIT (Laden). Wird dieser Schalter betätigt, d.h. auf 1 und dann wieder auf 0 geschaltet, werden die Daten, die an B_7 bis B_0 liegen, im Speicher bei der Adresse abgespeichert, die von den Schaltern A_3 bis A_0 spezifiziert ist.

1. Beispiel:
Die Zahl $1\ 5_{16}$ wird in Adresse 0 1 1 1 abgespeichert.

Ablauf:
1. B_7 bis B_0 auf 0 0 0 1 0 1 0 1 einstellen
2. A_7 bis A_0 auf 0 0 0 0 0 1 1 1 einstellen
3. Mit DEPOSIT-Schalter das System takten

Während DEPOSIT = 1 ist, erscheinen in R_7 bis R_0 die abzuspeichernden Daten. Dies ist als Kontrolle gedacht. Ist DEPOSIT wieder gleich 0, erscheinen in R_7 bis R_0 wieder die zufällig im Akku vorhandenen Daten.
Nachdem C_2 bzw. DEPOSIT wieder 0 ist, sind die Daten unter der Adresse 0 1 1 1 im Speicher abgespeichert. Die *B*-Schalter können jetzt beliebig verstellt werden.
Möchte man nachträglich die Daten in Adresse 0 1 1 1 kontrollieren, kann dies über Schalter C_1 EXAMINE (Lese) geschehen.

2. Beispiel:
Der Inhalt der Adresse 0 1 1 1 wird kontrolliert.

Ablauf:
1. A_7 bis A_0 auf 0 0 0 0 0 1 1 1 einstellen

2. Schalter EXAMINE auf 1 stellen
3. In R_7 bis R_0 erscheinen die Daten der Adresse 0 1 1 1

Solange EXAMINE = 1 ist, können mit Hilfe der Schalter A_3 bis A_0 alle Adresseninhalte kontrolliert werden.

3. Beispiel:
Folgene Daten werden unter der angegebenen Adresse abgespeichert:

Adresse	Daten
0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 1	0 0 0 0 0 0 0 1
0 0 1 0	0 0 0 0 0 0 1 0
0 0 1 1	0 0 0 0 0 0 1 1
0 1 0 0	0 0 0 0 0 1 0 0
0 1 0 1	0 0 0 0 0 1 0 1
0 1 1 0	0 0 0 0 0 1 1 0
0 1 1 1	0 0 0 0 0 1 1 1
1 0 0 0	0 0 0 0 1 0 0 0
1 0 0 1	0 0 0 0 1 0 0 1
1 0 1 0	0 0 0 0 1 0 1 0
1 0 1 1	0 0 0 0 1 0 1 1
1 1 0 0	0 0 0 0 1 1 0 0
1 1 0 1	0 0 0 0 1 1 0 1
1 1 1 0	0 0 0 0 1 1 1 0
1 1 1 1	0 0 0 0 1 1 1 1

Ablauf:
1. B_7 bis B_0 auf 0 0 0 0 0 0 0 0 einstellen
2. A_7 bis A_0 auf 0 0 0 0 0 0 0 0 einstellen
3. Mit DEPOSIT-Schalter System takten
4. B_7 bis B_0 auf 0 0 0 0 0 0 0 1 einstellen
5. A_7 bis A_0 auf 0 0 0 0 0 0 0 1 einstellen
6. Mit DEPOSIT-Schalter System takten usw.

Die so abgespeicherten Daten bleiben beliebig lang enthalten. Sie werden nur zerstört bei:
– Stromausfall oder Abschalten des Gerätes
– Abspeichern neuer Daten mit DEPOSIT unter derselben Adresse (alte Daten werden überschrieben)
– Abspeichern neuer Daten mit dem STA-Befehl in derselben Adresse
– Umschalten des SYSTEM-Schalters auf ein neues Experimentierprogramm

4. Beispiel:
Die in Beispiel 3 abgespeicherten Daten werden über den EXAMINE-Schalter nachkontrolliert.

Ablauf:
1. EXAMINE-Schalter auf 1
2. Mit A_3 bis A_0 die verschiedenen Adressen einstellen
3. In R_7 bis R_0 erscheinen die abgespeicherten Daten

5. Beispiel:
Überschreiben des Inhaltes der Adresse 0 1 1 1 mit den neuen Daten 1 1 1 1 0 0 0 0.

Ablauf:
1. B_7 bis B_0 auf 1 1 1 1 0 0 0 0 einstellen
2. A_7 bis A_0 au 0 0 0 0 0 1 1 1 einstellen
3. Mit DEPOSIT-Schalter System takten

Kontrollieren Sie über EXAMINE nach, ob der neue Inhalt in Adresse 0 1 1 1 tatsächlich vorhanden ist.

6. Beispiel:
Überschreiben des Inhaltes der Adresse 1 0 1 0 mit den Daten 1 1 0 0 1 1 0 0 mit Hilfe des STA-Befehles.

Ablauf:
1. B_7 bis B_0 auf 1 1 0 0 1 1 0 0 einstellen
2. A_7 bis A_0 auf 1 1 0 1 0 0 0 0 einstellen (dies entspricht dem Befehl INP = Lade *B*-Eingänge in den Akkumulator)
3. Mit Schalter C_4 System takten (in R_7 bis R_0 muß jetzt die Information B_7 bis B_0 erscheinen)
4. A_7 bis A_0 auf 1 1 1 0 1 0 1 0 einstellen (dies entspricht laut Tab. 3.5.1 dem Befehl STA = Speichere Akku in Adresse *a a a a* ab)
5. Mit Schalter C_4 System takten

Kontrollieren Sie über EXAMINE, den neuen Inhalt der Speicheradresse 1 0 1 0.

7. Beispiel:
Den Inhalt der Adresse 1 1 1 0 in den Akkumulator laden.

Ablauf:
1. A_7 bis A_0 auf 0 0 1 1 1 1 1 0 einstellen (entspricht dem Befehl LDA = Lade Inhalt Adresse *a a a a*)
2. Mit Schalter C_4 System takten. In R_7 bis R_0 erscheint 0 0 0 0 1 1 1 0

8. Beispiel:
Addition der Hexadezimalzahlen $1\ 7_{16}$ und $3\ 2_{16}$

Ablauf:
1. B_7 bis B_0 auf 0 0 0 1 0 1 1 1 = $1\ 7_{16}$ einstellen
2. Daten B_7 bis B_0 über DEPOSIT in der Adresse 0 0 0 0 abspeichern
(A_3 bis A_0 auf 0 0 0 0 einstellen!)
3. B_7 bis B_0 auf 0 0 1 1 0 0 1 0 = $3\ 2_{16}$ einstellen
4. Über INP-Befehl die Daten B_7 bis B_0 in den Akku laden
5. A_7 bis A_0 auf 0 1 1 1 0 0 0 0 einstellen (entspricht dem Befehl ADD = Addiere Inhalt der Adresse 0 0 0 0)
6. Mit Schalter C_4 System takten. In R_7 bis R_0 erscheint das Additonsergebnis
0 1 0 0 1 0 0 1 $\triangleq$ $4\ 9_{16}$

Bei den arithmetischen und logischen Funktionen ADD, SUB, AND, IOR und XOR wird also immer der Inhalt einer Adresse mit dem jeweiligen Inhalt des Akkus verknüpft. Es ist gleichgültig, wie dabei die Schalter B_7 bis B_0 stehen. Nur über den Befehl INP an A_7 bis A_0 können die Daten B_7 bis B_0 in den Akku gelangen.
Im nächsten Beispiel wollen wir den Inhalt von 2 unterschiedlichen Adressen EXCLUSIV-ODER-verknüpfen.

9. Beispiel:
Die Inhalte der Adressen 1 0 1 0 und 0 1 1 1 werden EXCLUSIV-ODER-verknüpft.

Anmerkung: Wenn Sie in der Zwischenzeit genau das vorgeschriebene Experimentierprogramm durchgeführt haben bzw. das Experimentiersystem nicht zwischendurch abgeschaltet haben, steht in den Adressen folgender Inhalt:

Adresse 1 0 1 0 → 1 1 0 0 1 1 0 0
Adresse 0 1 1 1 → 1 1 1 1 0 0 0 0

Ist das nicht der Fall, über DEPOSIT die beiden Adressen entsprechend laden.

Ablauf:
1. A_7 bis A_0 auf 0 0 1 1 1 0 1 0 einstellen (entspricht Befehl LDA)
2. Mit Schalter C_4 System takten. In R_7 bis R_0 erscheint der Inhalt der Adresse 1 0 1 0, der jetzt auch Inhalt des Akkus ist

3. A_7 bis A_0 auf 1 0 1 1 0 1 1 1 einstellen (entspricht Befehl XOR)
4. Mit Schalter C_4 System takten. In R_7 bis R_0 erscheint das Ergebnis 0 0 1 1 1 1 0 0

Der neue Akku-Inhalt ist also 0 0 1 1 1 1 0 0. Wenn Sie jetzt z.B. noch einmal mit C_4 takten (A_7 bis A_0 bleiben unverändert), erscheint in R_7 bis R_0 1 1 0 0 1 1 0 0. Bei dem erneuten Takt wird nämlich die XOR-Verknüpfung des neuen Akku-Inhaltes mit dem nach wie vor unveränderten Inhalt der Adresse 0 1 1 1 gebildet. Es ergibt sich somit:

0 0 1 1 1 1 0 0	Inhalt Akku
1 1 1 1 0 0 0 0	Inhalt Adresse 0 1 1 1
1 1 0 0 1 1 0 0	

Alle anderen Befehle sind Ihnen vom Prinzip her bekannt.

Aufgaben:

1. Subtrahieren Sie folgende Aufgaben:
   a) $3\,4_{16} - 2\,1_{16} =$
   b) $-2\,4_{16} - 5\,4_{16} =$

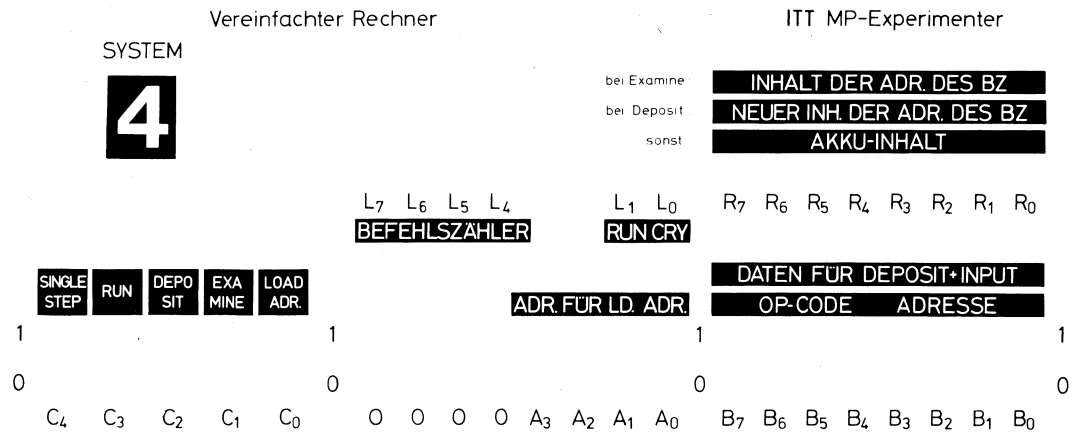
2. Erhöhen Sie den Inhalt der Adresse 0 1 1 1 über Programm um 1.

3. Führen Sie folgende Rechenoperationen durch:

$$3\,5_{16} + 1\,7_{16} - 2\,4_{16} =$$

Die Lösungen finden Sie auf Seite E23.

Experiment 5: Vereinfachter Rechner



Bei diesem Experiment wird ein vereinfachter, aber kompletter Rechner simuliert. Er hat denselben Befehlsvorrat wie der Akkumulator mit Datenspeicher in Experiment 4. Zusätzlich hat er einen HALT-Befehl (HLT), damit der Rechner am Ende eines Programms angehalten werden kann. Im Gegensatz zum Experiment 4 werden im 16-Wort-Speicher nicht nur Daten sondern auch das Programm abgespeichert. Das Programm und die Daten werden mit dem DEPOSIT-Schalter C_2 in den Speicher geladen. Damit ein Programm automatisch ablaufen kann, enthält der simulierte Rechner einen Befehlszähler (BZ). Welche der 16 Adressen gerade selektiert ist, wird durch die LEDs L_7 bis L_4 angezeigt. Die Funktionsweise des Befehlszählers können Sie wie folgt kontrollieren:

– Alle Schalter auf 0 stellen

– Schalter C_0 (LOAD ADR.) takten. Der Befehlszähler wird jetzt mit der in den Schaltern A_3 bis A_0 stehenden Adresse – in diesem Falle 0 0 0 0 – geladen. Die LEDs L_7 bis L_4 müssen jetzt auch 0 0 0 0 anzeigen.

– Takten Sie jetzt mit dem Schalter C_1 (EXAMINE) das System. Anhand der LEDs L_7 bis L_4 ist zu erkennen, daß nach jedem Takt der Befehlszähler um eine Adresse weiterspringt. Solange C_4 auf 1 steht, wird in den LEDs R_7 bis R_0 der **zufällige** Inhalt der Speicheradresse angezeigt, die vom Befehlszähler selektiert ist.

Soll der Befehlszähler auf eine bestimmte Adresse geladen werden, so kann dies über die Schalter A_3 bis A_0 erfolgen.

Wenn Sie z.B. A_3 bis A_0 auf 0 1 1 0 einstellen und den Schalter LOAD-ADRESS (C_0) betätigen, wird der Befehlszähler auf diese Adresse gesetzt (Anzeige durch L_7 bis L_4). Wenn Sie jetzt mit dem RUN-Schalter weitertakten, zählt der Zähler von dieser Stellung weiter.

Mit den Schaltern B_7 bis B_0 können OP-Code und Adresse eingegeben werden. Hierbei ist unbedingt zu berücksichtigen, daß es sich um einen **Befehl** handelt, der in einer bestimmten Adresse abgespeichert wird. Wenn Sie z.B. B_7 bis B_0 auf 0 1 1 1 0 0 1 0 einstellen und den Schalter DEPOSIT C_2 takten, wird dieser Befehl in der Adresse abgespeichert, die gerade vom Befehlszähler selektiert ist. Der Befehl 0 1 1 1 0 0 1 0 besagt laut Tab. 3.5.1: Addiere den Inhalt der Adresse 0 0 1 0 zum Inhalt des Akkus. Dies bedeutet – und das ist unbedingt zu beachten – daß bei der Befehlszählerstellung, bei der dieser Befehl eingegeben wurde, diese Rechenoperation durchgeführt wird. Da hier eine neue Denkweise einsetzt, wollen wir das Prinzip an einem Beispiel ausführlich erläutern:

1. Beispiel:

Folgende Aufgabenstellung ist zu programmieren: Die an B_7 bis B_0 eingestellten Daten sollen mit dem Inhalt der Adresse 0 1 0 0 addiert werden.

Ablauf:

1. Alle Schalter zunächst in Stellung 0 bringen
2. Schalter LOAD-ADRESS (C_0) takten. Damit wird der Befehlszähler auf Adresse 0 0 0 0 gesetzt
3. Schalter B_7 bis B_0 auf 0 1 0 0 0 0 0 0 stellen. Dies entspricht dem Befehl CLA = Lösche Akku
4. Mit DEPOSIT-Schalter (C_2) takten. Damit ist der CLA-Befehl in der Adresse 0 0 0 0 gespeichert. Gleichzeitig springt der Befehlszähler auf Adresse 0 0 0 1, d.h., jetzt kann eine Information in dieser Adresse gespeichert werden
5. Schalter B_7 bis B_0 auf 1 1 0 1 0 0 0 0 einstellen. Dies entspricht dem INP-Befehl

6. Mit DEPOSIT-Schalter takten; der INP-Befehl ist in Adresse 0 0 0 1 gespeichert, Befehlszähler springt auf Adresse 0 0 1 0
7. Schalter B_7 bis B_0 auf 0 1 1 1 0 1 0 0 einstellen. Dies entspricht dem Additionsbefehl mit der Adresse 0 1 0 0
8. Mit DEPOSIT-Schalter takten, ADD-Befehl ist in Adresse 0 0 1 0 gespeichert
9. Schalter B_7 bis B_0 auf 1 1 1 1 0 0 0 0 stellen. Dies entspricht dem HALT-Befehl. Wenn am Ende eines Programms dieser Befehl nicht erscheint, rechnet das System unkontrolliert weiter
10. Mit DEPOSIT-Schalter takten
11. Damit sich ein kontrollierbares Ergebnis ergibt, speichern wir in Adresse 0 1 0 0 die Daten 1 1 1 1 0 0 0 0 ab. Hierzu B_7 bis B_0 auf 1 1 1 1 0 0 0 0 einstellen und mit DEPOSIT takten

Jetzt ist der Programmiervorgang abgeschlossen, und der Rechner kann die jeweils an den B-Schaltern eingestellten Daten mit dem Inhalt 1 1 1 1 0 0 0 0 der Adresse 0 1 0 0 addieren. Damit wir die einzelnen Schritte genau verfolgen können, schalten wir C_4 auf 1, d.h. SINGLE-STEP-Betrieb. Da der Befehlszähler jetzt bei Adresse 0 1 0 0 steht, setzen wir ihn durch Takten von C_0 = LOAD-ADRESS auf Adresse 0 0 0 0 zurück (A_3 bis A_0 auf 0 0 0 0). Als erste Aufgabe rechnen wir 0 0 0 0 1 1 1 1 plus Inhalt Adresse 0 1 0 0. Hierzu stellen wir B_7 bis B_0 auf 0 0 0 0 1 1 1 1 ein. Jetzt takten wir einmal mit Schalter RUN. Die Anzeige R_7 bis R_0 muß Null sein, da mit dem 1. Takt der Akkumulator gelöscht wird. Der Befehlszähler springt auf Adresse 0 0 0 1. Jetzt mit Schalter RUN wieder takten. Daten B_7 bis B_0 werden in den Akku geladen und erscheinen in R_7 bis R_0. Mit RUN nochmals takten. Jetzt erfolgt die Addition. Im Akkumulator und in der Anzeige steht das Ergebnis 1 1 1 1 1 1 1 1.
Mit RUN takten. Der Rechner arbeitet nicht weiter, da im Programm ein HALT-Befehl gespeichert ist. Sie können jetzt selbst mit verschiedenen Daten an B_7 bis B_0 das Programm wiederholen. Wesentlich ist, daß zu Beginn einer Aufgabe immer erst der Befehlszähler auf 0 0 0 0 zurückzustellen ist. Wenn Sie Beispiele wählen, die einen Übertrag ergeben, leuchtet die Carry-Anzeige L_0 auf.
Wenn der Schalter SINGLE-STEP auf 0 steht und dann der RUN-Schalter betätigt wird, läuft das Programm automatisch mit einer sehr hohen Systemfrequenz ab. Die einzelnen Zwischenschritte können dann nicht mehr mit dem Auge verfolgt werden. Lediglich ein kurzes Aufleuchten der Anzeige RUN (L_1) signalisiert, daß der Rechner arbeitet.

2. Beispiel:
Die Aufgabenstellung lautet, einen Vorwärtszähler zu simulieren. Hierzu verwenden wir den INC-Befehl.

Ablauf:
1. Befehlszähler auf 0 0 0 0 stellen
2. B_7 bi B_0 auf 0 1 0 1 0 0 0 0 einstellen
3. Mit DEPOSIT das System 16mal takten. Jetzt ist in jeder Adresse der INC-Befehl gespeichert. Da kein HALT-Befehl eingegeben wurde, läuft das System so lange, wie mit RUN getaktet wird. Wenn Sie C_4 = 0 einstellen und den RUN-Schalter auf 1 stellen, läuft der Vorgang automatisch ab. Ab R_3 können Sie ein deutliches Blinken der Lampen erkennen. Die Zählfrequenz läßt sich verringern, wenn Sie nur einen INC-Befehl und z.B. 15 NOP-Befehle eingeben. Wenn jetzt der Befehlszähler läuft, wird nur immer bei einer Adresse der Akku-Inhalt um 1 erhöht, während der übrigen 15 Adressen führt das System keine Operation aus.

3. Beispiel:
Vorwärtszähler mit niedriger Zählfrequenz.

Ablauf:
1. Befehlszähler auf 0 0 0 0 stellen
2. INC-Befehl laden
3. NOP-Befehl 15mal laden

Sie können die wesentlich niedrigere Zählfrequenz an R_7 bis R_0 deutlich erkennen.
Über den DEC-Befehl läßt sich ein Rückwärtszähler simulieren. Erstellen Sie selbst einmal ein Programm für einen langsamen Rückwärtszähler. Dies dürfte an dieser Stelle bestimmt keine Schwierigkeiten mehr bereiten.

Ein wesentliches Merkmal eines Rechners ist, daß er Entscheidungen treffen kann. Wie Sie später noch sehen werden, gibt es hierfür bestimmte Befehle. Der hier simulierte einfache Rechner verfügt nicht über diese Befehle. Über eine geschickte Programmierung ist es jedoch möglich, bestimmte Entscheidungen auch von diesem System treffen zu lassen.

4. Beispiel:
Größer/kleiner-Vergleich von 2 Zahlen.

In diesem Beispiel werden 2 Zahlen *A* und *B* miteinander verglichen. Die jeweils größere Zahl wird dabei zur Anzeige gebracht. Als Zahl *A* wählen wir $20_{16} \triangleq 100000_2$. Die Zahl *A* wird im Speicher abgespeichert. Die Zahl *B* kann an den Schaltern B_7 bis B_0 eingestellt werden. Sie wird dann mit *A* verglichen. Ist *A* größer als *B*, wird *A* angezeigt und umgekehrt. Der Rechner bildet die Differenz $B - A$ und trifft aufgrund des Ergebnisses die Entscheidung, welche Zahl in R_7 bis R_0 angezeigt wird. Der Ablauf ist in einem Flußdiagramm dargestellt (Bild 1).

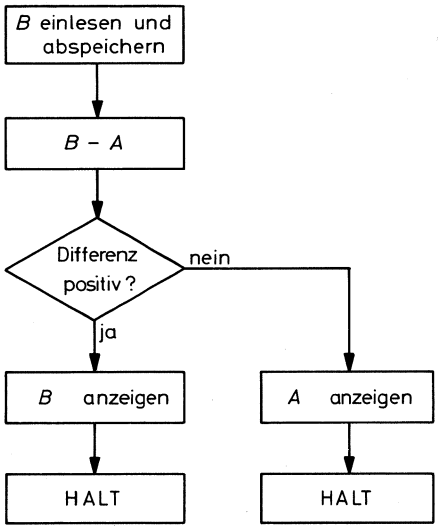


Bild 1
Flußdiagramm zum 4. Beispiel

Wir geben zunächst das Programm für diese Aufgabe an und besprechen anschließend die einzelnen Schritte (Tab. 1).

Adresse		Inhalt		Befehl	Kommentar
hexadez.	binär	hexadez.	binär		
0	0000	D *x*	1101 *xxxx*	INP	*B*-Eingänge in den Akku laden
1	0001	E E	11101110	STA	Inhalt Akku in Adresse E abspeichern
2	0010	8 F	10001111	SUB	Subtraktion *B* minus Inhalt Adresse F
3	0011	9 D	10011101	AND	höchste Stelle ausblenden
4	0100	7 C	01111100	ADD	Entscheidungsaddition
5	0101	E 7	11100111	STA	Ergebnis in Adresse 7 abspeichern
6	0110	3 F	00111111	LDA	Zahl *A* in Akku laden
7	0111	– – –	– – – – – – – –	ADD/HLT	Entscheidungsadresse
8	1000	3 E	00111110	LDA	Zahl *B* in Akku laden
9	1001	F *x*	1111 *xxxx*	HLT	System HALT
A	1010		*xxxxxxxx*	DATEN	nicht belegt
B	1011		*xxxxxxxx*		nicht belegt
C	1100		01110000		Daten für Entscheidungsaddition
D	1101		10000000		Maske für Ausblendung
E	1110		– – – – – – – –		Adresse für Zahl *B*
F	1111		00100000		Zahl *A*

Tab. 1
Programm zum 4. Beispiel

Entsprechend der Aufgabenstellung soll die im Speicher abgespeicherte Zahl $2\,0_{16}$ mit einer an B_7 bis B_0 eingestellten Zahl verglichen werden. Die Zahl A ist in Adresse F abgespeichert. Wird jetzt an B_7 bis B_0 eine Zahl B eingestellt, so muß diese zunächst in den Akku geladen werden (INP-Befehl).
Mit dem STA-Befehl wird dann die Zahl B in der Adresse E abgespeichert. Dabei wird der Inhalt des Akkus nicht verändert, d.h., B steht weiterhin auch im Akku.
Durch den SUB-Befehl wird die Differenz $B - A$ gebildet. Dabei entsteht bei $B > A$ ein positives und bei $B < A$ ein negatives Ergebnis. Nach der Zweierkomplementarithmetik wird eine positive Zahl durch eine 0, eine negative Zahl durch eine 1 in der werthöchsten Stelle gekennzeichnet. Entscheidungskriterium ist also das werthöchste bit des Ergebnisses der Subtraktion.
Aus diesem Grunde wird jetzt mit einem AND-Befehl das werthöchste bit ausgeblendet.
Beispiel:

```
   0 x x x x x x x      beliebige positive Zahl
∧  1 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0

   1 x x x x x x x      beliebige negative Zahl
∧  1 0 0 0 0 0 0 0
   1 0 0 0 0 0 0 0
```

In Adresse D ist 1 0 0 0 0 0 0 0 gespeichert. Der in Adresse 3 gespeicherte AND-Befehl bildet die UND-Verknüpfung zwischen Inhalt Akku und Inhalt Adresse D. Als Ergebnisse können nur 0 0 0 0 0 0 0 0 (positives Ergebnis der Subtraktion) bzw. 1 0 0 0 0 0 0 0 (negatives Ergebnis der Subtraktion im Akku) erscheinen.
Im nächsten Programmschritt wird nun eine Entscheidungsaddition durchgeführt. Entsprechend dem ADD-Befehl in Adresse 4 wird zu dem ausgeblendeten Ergebnis der Inhalt von Adresse C = 0 1 1 1 0 0 0 0 addiert.
Ist $B > A$ erhalten wir:

```
   0 0 0 0 0 0 0 0
+  0 1 1 1 0 0 0 0
   0 1 1 1 0 0 0 0
```

Ist $B < A$ erhalten wir:

```
   1 0 0 0 0 0 0 0
+  0 1 1 1 0 0 0 0
   1 1 1 1 0 0 0 0
```

Entsprechend der Befehlsstruktur des einfachen Rechners stellen die 4 werthöheren bit den OP-Code, die 4 wertniedrigeren bit die Adresse dar. Entscheidend ist, daß der OP-Code 1 1 1 1 den HLT-Befehl darstellt. Wenn wir nun das Ergebnis der Addition über den STA-Befehl in Adresse 7 abspeichern, wird der Rechner beim Abarbeiten des Programmes bei der Adresse 7 entweder anhalten (bei $B < A$) oder, wenn $B > A$ ist, eine Addition zwischen Akku-Inhalt und Inhalt Adresse 0 durchführen. Wenn der Befehlszähler bei $B < A$ gestoppt wird, muß die größere Zahl A angezeigt werden. Dies geschieht dadurch, daß in Adresse 6 der Akku über den LDA-Befehl mit dem Inhalt der Adresse F = $2\,0_{16}$ geladen wird.
Ist dagegen $B > A$, erfolgt bei Adresse 7 eine Addition von Akku-Inhalt und Inhalt Adresse 0, deren Ergebnis aber keine Bedeutung hat. In diesem Falle muß die in Adresse E gespeicherte Zahl B über einen LDA-Befehl in den Akku geladen werden, damit sie in R_7 bis R_0 angezeigt werden kann. In Adresse 9 ist für diesen Fall dann der HLT-Befehl programmiert.
Sicherlich werden Ihnen die hier dargelegten Gedankengänge kompliziert erscheinen. Das liegt ganz einfach daran, daß der vereinfachte Rechner noch keine direkten Entscheidungsbefehle enthält. Wir müssen ihn vielmehr so programmieren, daß er bei einem bestimmten Kriterium selbst einen HALT-Befehl erzeugt. Bevor wir das Programm mit konkreten B-Zahlen noch einmal durchsprechen, muß das Programm zunächst geladen werden:

1. Alle Schalter auf Null stellen
2. Schalter LOAD-ADR. takten

3. Schalter B_7 bis B_4 auf 1 1 0 1 einstellen und DEPOSIT takten
4. B_7 bis B_0 auf 1 1 1 0 1 1 1 0 einstellen und DEPOSIT takten
5. B_7 bis B_0 auf 1 0 0 0 1 1 1 1 einstellen und DEPOSIT takten
6. B_7 bis B_0 auf 1 0 0 1 1 1 0 1 einstellen und DEPOSIT takten
7. B_7 bis B_0 auf 0 1 1 1 1 1 0 0 einstellen und DEPOSIT takten
8. B_7 bis B_0 auf 1 1 1 0 0 1 1 1 einstellen und DEPOSIT takten
9. B_7 bis B_0 auf 0 0 1 1 1 1 1 1 einstellen und DEPOSIT takten
10. DEPOSIT takten. Damit springt der Befehlszähler auf Adresse 8
11. B_7 bis B_0 auf 0 0 1 1 1 1 1 0 einstellen und DEPOSIT takten
12. B_7 bis B_0 auf 1 1 1 1 *x x x x* einstellen und DEPOSIT takten
13. DEPOSIT 2mal takten. Damit springt der Befehlszähler auf Adresse C
14. B_7 bis B_0 auf 0 1 1 1 0 0 0 0 einstellen und DEPOSIT takten
15. B_7 bis B_0 auf 1 0 0 0 0 0 0 0 einstellen und DEPOSIT takten
16. DEPOSIT takten, der Befehlszähler springt auf Adresse F
17. B_7 bis B_0 auf 0 0 1 0 0 0 0 0 einstellen und DEPOSIT takten

Damit steht das Programm im Speicher. Der Befehlszähler steht wieder bei Adresse 0 (L_7 bis L_4). Als Beispiel 1 wollen wir einen Vergleich zwischen $B = 7_{16}$ und $A = 2\ 0_{16}$ durchführen. Damit die einzelnen Schritte nachvollzogen werden können, System auf SINGLE-STEP-Betrieb schalten ($C_4 = 1$). An B_7 bis B_0 wird 0 0 0 0 0 1 1 1 $\triangleq$ 7_{16} eingestellt. Jetzt wird das System mit dem RUN-Schalter schrittweise getaktet.

1. Takt: In R_7 bis R_0 erscheinen die Daten B_7 bis B_0
2. Takt: Daten B_7 bis B_0 werden in Adresse E gespeichert
(Kontrollieren Sie über EXAMINE-Funktion. Vergessen sie nicht Befehlszähler über LOAD-ADR wieder auf Adresse 2 zurückzustellen)
3. Takt: R_7 bis R_0 gleich 1 1 1 0 0 1 1 1 $= -1\ 9_{16}$
4. Takt: R_7 bis R_0 gleich 1 0 0 0 0 0 0 0. Das werthöchste bit wird ausgeblendet
5. Takt: R_7 bis R_0 gleich 1 1 1 1 0 0 0 0. Durch die Addition wird der HLT-Befehl für Adresse 7 gebildet
6. Takt: R_7 bis R_0 bleibt, Akku-Inhalt wird in Adresse 7 abgespeichert
7. Takt: Inhalt Adresse F wird in den Akku geladen und erscheint in R_7 bis R_0
8. Takt: Keine Änderung, da HLT-Befehl. Auch ein weiteres Takten hat keinen Einfluß

Jetzt führen wir den Vergleich mit der Zahl $B = 3\ F_{16}$ durch. Über LOAD-ADR Befehlszähler auf Adresse 0 einstellen, und B_7 bis B_0 auf 0 0 1 1 1 1 1 1 $\triangleq$ 3 F_{16} einstellen.

1. Takt: In R_7 bis R_0 erscheinen die Daten B_7 bis B_0
2. Takt: Daten werden in Adresse E gespeichert
3. Takt: R_7 bis R_0 gleich 0 0 0 1 1 1 1 1 $\triangleq$ 1 F_{16}
4. Takt: R_7 bis R_0 gleich 0 0 0 0 0 0 0 0 (Ausblenden)
5. Takt: R_7 bis R_0 gleich 0 1 1 1 0 0 0 0. Entspricht hier einem Additionsbefehl für Adresse 7
6. Takt: R_7 bis R_0 bleibt, Akku-Inhalt wird in Adresse 7 gespeichert
7. Takt: Inhalt Adresse F wird in Akku geladen und erscheint in R_7 bis R_0
8. Takt: R_7 bis R_0 gleich 1 1 1 1 0 0 0 0. Dieses Resultat ergibt sich aus der Addition von Akku-Inhalt und Inhalt Adresse 0

Inhalt Adresse 0:	1 1 0 1 0 0 0 0
Inhalt Akku:	+ 0 0 1 0 0 0 0 0
	1 1 1 1 0 0 0 0

Es kann auch ein anderes Ergebnis erscheinen, wenn bei der Programmierung der Adresse 0 die bit b_3 bis b_0 einen anderen Wert gehabt haben. Vom Programm ändert sich nichts, da der INP-Befehl keine bestimmte Adresse spezifiziert (*x x x x*)
9. Takt: Die Zahl B (Inhalt Adresse E) wird in den Akku geladen. R_7 bis R_0 gleich 0 0 1 1 1 1 1 1
10. Takt: Keine Änderung, da HLT-Befehl. Auch ein weiteres Takten hat keinen Einfluß

Entscheidend bei diesem Programm ist der Gedankengang, über eine bestimmte Operation in einer bestimmten Adresse unter einer bestimmten Voraussetzung einen HALT-Befehl zu erzeugen.

5. Beispiel:
Das System soll so programmiert werden, daß beim Takten die LEDs R_3 bis R_0 nach folgendem Schema aufleuchten:

	R_3	R_2	R_1	R_0
1. Takt	0	0	0	0
2. Takt	0	0	0	1
3. Takt	0	0	1	0
4. Takt	0	1	0	0
5. Takt	1	0	0	0
6. Takt	0	1	0	0
7. Takt	0	0	1	0
8. Takt	0	0	0	1
9. Takt	0	0	0	0
10. Takt	0	0	0	1
11. Takt	0	0	1	0
12. Takt	0	1	0	0
13. Takt	1	0	0	0
14. Takt	0	1	0	0

Für diese Aufgabenstellung ergibt sich z.B. folgende Programmierungsmöglichkeit (Tab. 2).

Adresse		Inhalt		Befehl	Kommentar
hexad.	binär	hexad.	binär		
0	0 0 0 0	4 *x*	0 1 0 0 *x x x x*	CLA	Lösche Akku
1	0 0 0 1	5 *x*	0 1 0 1 *x x x x*	INC	Incrementiere Akku
2	0 0 1 0	5 *x*	0 1 0 1 *x x x x*	INC	Incrementiere Akku
3	0 0 1 1	7 F	0 1 1 1 1 1 1 1	ADD	Addiere Akku-Inhalt mit 2
4	0 1 0 0	7 E	0 1 1 1 1 1 1 0	ADD	Addiere Akku-Inhalt mit 4
5	0 1 0 1	8 E	1 0 0 0 1 1 1 0	SUB	Subtrahiere von Akku-Inhalt 4
6	0 1 1 0	8 F	1 0 0 0 1 1 1 1	SUB	Subtrahiere von Akku-Inhalt 2
7	0 1 1 1	6 *x*	0 1 1 0 *x x x x*	DEC	Decrementiere Akku
8	1 0 0 0	6 *x*	0 1 1 0 *x x x x*	DEC	Decrementiere Akku
9	1 0 0 1	5 *x*	0 1 0 1 *x x x x*	INC	Incrementiere Akku
A	1 0 1 0	5 *x*	0 1 0 1 *x x x x*	INC	Incrementiere Akku
B	1 0 1 1	7 F	0 1 1 1 1 1 1 1	ADD	Addiere Akku-Inhalt mit 2
C	1 1 0 0	7 E	0 1 1 1 1 1 1 0	ADD	Addiere Akku-Inhalt mit 4
D	1 1 0 1	8 E	1 0 0 0 1 1 1 0	SUB	Subtrahiere von Akku-Inhalt 4
E	1 1 1 0	0 4	0 0 0 0 0 1 0 0	DATEN	
F	1 1 1 1	0 2	0 0 0 0 0 0 1 0		

Tab. 2
Programm zum 5. Beispiel

Dieses Programm ist relativ einfach und bedarf daher keiner umfangreichen Erklärung. Für die Speicherung der Daten werden nur 2 Adressen benötigt, da die Additions- und Subtraktionsbefehle dieselben Daten benötigen. Zu bemerken ist außerdem noch, daß die Aufgabenstellung auch noch über andere Programme möglich ist. So kann z.B. der Befehl INC, der beim 2. Takt das Muster 0 0 0 1 in R_3 bis R_0 erzeugen soll, durch den Befehl SP1 = Setze Akku gleich 1, ersetzt werden.
Störend im Programmablauf ist, daß die Adressen E und F NOP-Befehle darstellen. Dadurch ändert sich bei 2 Takten der Akku-Inhalt nicht. Umfangreichere Mikrorechner lassen sich dagegen so programmieren, daß ein kontinuierlicher Ablauf entsteht.
Vielleicht taucht bei Ihnen jetzt die Frage auf, welche praktische Nutzanwendung ein solches Programm haben könnte. Beispielsweise könnte man anstatt der LEDs R_3 bis R_0 einen

Digital-Analog-Wandler (D/A-Wandler) an das System anschließen. Ein D/A-Wandler wandelt ein digitales Signal in ein analoges Signal um. Er könnte z.B. folgendes Verhalten aufweisen (Bild 2).

R_3	R_2	R_1	R_0	A
0	0	0	0	0 V
0	0	0	1	1 V
0	0	1	0	2 V
0	1	0	0	4 V
1	0	0	0	8 V

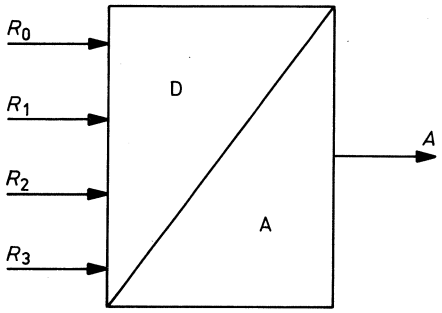


Bild 2
D/A-Wandler

Liegt an seinen Eingängen das Wort 0 0 0 0, beträgt die Ausgangsspannung 0 V, bei 0 0 0 1 ist sie 1 V usw.
Läßt man nun den Befehlszähler mit einer bestimmten Frequenz laufen, so kann man mit einem Oszilloskop am Ausgang des D/A-Wandlers folgendes messen (Bild 3).

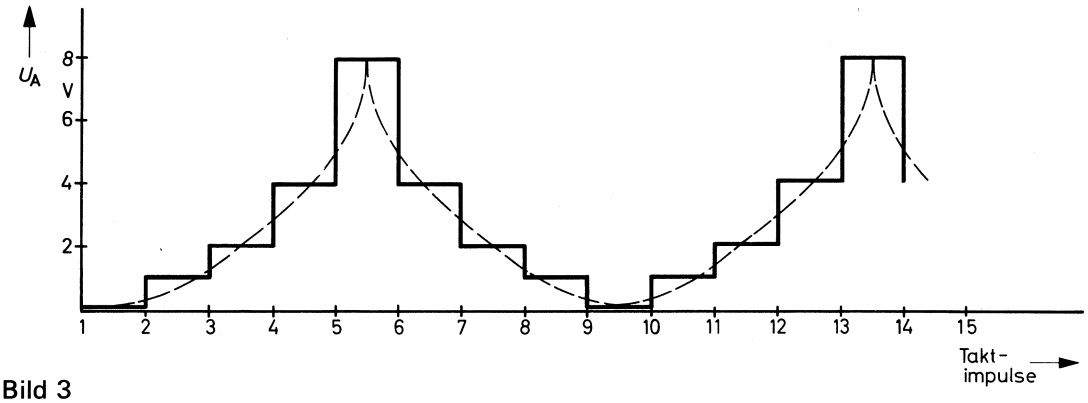


Bild 3
Ausgangsspannung eines D/A-Wandlers

Die so gewonnene Treppenspannung kann durch entsprechende Formung als eine quadratische Funktion benutzt werden, die irgendeinen Vorgang steuert.

Aufgaben:

1. Erstellen Sie die Programme für folgende Aufgaben:
   a) $5_{10} \cdot B =$
   b) $8_{10} \cdot B =$

2. Der 8421-BCD-Code entspricht den Binärzahlen 0 0 0 0 bis 1 0 0 1. Mit 4-bit-Wörtern lassen sich 16 verschiedene Zahlen darstellen. Damit werden für den 8421-BCD-Code die Zahlen von 1 0 1 0 bis 1 1 1 1 nicht benötigt und als Pseudotetraden bezeichnet. Entwerfen Sie ein Programm, das bei Eingabe einer Pseudotetrade mit B_3 bis B_0 in der Anzeige R_7 bis R_0 −1, d.h. 1 1 1 1 1 1 1 1, erscheinen läßt. Handelt es sich bei der Eingabe um keine Pseudotetrade, soll in R_3 bis R_0 die Eingabe B_3 bis B_0 erscheinen.

Anmerkung:
Orientieren Sie sich bei dieser Aufgabe an dem 4. Beispiel.

Die Lösungen finden Sie auf Seite E23.

Lösungen zu Experiment 1

1. a) A-Schalter: $01111101 \triangleq 125_{10}$
+ B-Schalter: $00101000 \triangleq 40_{10}$
Ergebnis: $\underbrace{10100101}_{R_7 \text{ bis } R_0} \triangleq 165_{10}$

b) A-Schalter: $10111000 \triangleq 184_{10}$
+ B-Schalter: $01100100 \triangleq 100_{10}$
$1\,\underbrace{00011100}_{R_7 \text{ bis } R_0} \triangleq 284_{10}$
L_0

2. a) A-Schalter: $01111000 \triangleq 120_{10}$
$\overline{A}$-Schalter 1: $10000111 \triangleq \overline{A}$
INC-Schalter 1: $\underbrace{10001000}_{R_7 \text{ bis } R_0} \triangleq -A = \overline{A} + 1$

b) A-Schalter: $00001100 \triangleq 12_{10}$
$\overline{A}$-Schalter 1: $11110011 \triangleq \overline{A}$
INC-Schalter 1: $\underbrace{11110100}_{R_7 \text{ bis } R_0} \triangleq -A = \overline{A} + 1$

c) A-Schalter: $00000010 \triangleq 2_{10}$
$\overline{A}$-Schalter 1: $11111101 \triangleq \overline{A}$
INC-Schalter 1: $\underbrace{11111110}_{R_7 \text{ bis } R_0} \triangleq -A = \overline{A} + 1$

3. a) A-Schalter: $01111000 \triangleq 120_{10}$
B-Schalter: $01100100 \triangleq 100_{10}$
$\overline{B}$-Schalter 1: $1\,\underbrace{00010011}_{R_7 \text{ bis } R_0} \triangleq A + \overline{B}$
L_0
INC-Schalter 1: $1\,\underbrace{00010100}_{R_7 \text{ bis } R_0} \triangleq A + \overline{B} + 1 = A - B$
L_0
Ergebnis: $00010100 = 20_{10}$

b) A-Schalter: $10000010 \triangleq 130_{10}$
B-Schalter: $10001100 \triangleq 140_{10}$
$\overline{B}$-Schalter 1: $\underbrace{11110101}_{R_7 \text{ bis } R_0} \triangleq A + \overline{B}$
INC-Schalter 1: $\underbrace{11110110}_{R_7 \text{ bis } R_0} = A + \overline{B} + 1 = A - B$
Ergebnis: $11110110 = -10_{10}$

Lösungen zu Experiment 2

1. a) Funktion C_3 bis C_0: 0111
A-Schalter: $00110100 \triangleq 34_{16}$
+ B-Schalter: $00100001 \triangleq 21_{16}$
$\underbrace{01010101}_{R_7 \text{ bis } R_0} \triangleq 55_{16}$

b) Funktion C_3 bis C_0: 1 0 0 0
A-Schalter: 0 0 0 1 0 1 0 0 ≙ $1\,4_{16}$
$-B$-Schalter: 0 1 0 0 1 0 0 0 ≙ $4\,8_{16}$
1 | 1 1 0 0 1 1 0 0 ≙ $-3\,4_{16}$
L_0 R_7 bis R_0

2. a) Funktion C_3 bis C_0: 1 0 0 0
A-Schalter: 1 0 0 0 0 0 0 1 ≙ $8\,1_{16}$
$-B$-Schalter: 0 1 1 1 0 1 0 1 ≙ $7\,5_{16}$
0 0 0 0 1 1 0 0 ≙ $0\,C_{16}$
R_7 bis R_0

b) Funktion C_3 bis C_0: 1 0 0 0
A-Schalter: 0 1 1 1 0 1 1 0 ≙ $7\,6_{16}$
$-B$-Schalter: 1 0 0 0 0 1 0 0 ≙ $8\,4_{16}$
1 | 1 1 1 1 0 0 1 0 ≙ $-0\,E_{16}$
L_0 R_7 bis R_0

3. a) Funktion C_3 bis C_0: 1 0 1 1
A-Schalter: 1 1 0 0 1 1 1 1
B-Schalter: 0 0 0 0 0 0 1 1
1 1 0 0 1 1 0 0
R_7 bis R_0

b) Funktion C_3 bis C_0: 1 0 0 1
A-Schalter: 1 1 0 1 0 1 1 0
B-Schalter: 0 0 0 0 1 1 1 1
0 0 0 0 0 1 1 0
R_7 bis R_0

Lösungen zu Experiment 3

1. a) 0 0 1 0 1 1 0 0 ≙ 44_{10}
R_7 bis R_0

b) 1 | 1 1 1 1 0 0 1 1 ≙ -13_{10}
L_0 R_7 bis R_0

c) 1 0 0 0 0 1 1 0 ≙ -122_{10}
R_7 bis R_0

Anmerkung: Bei der Aufgabe c) ist darauf zu achten, daß $-4\,8_{16}$ als Zweierkomplement zu bearbeiten ist. Der Rechenablauf ist also folgender:

1. Über CLA löschen
2. $4\,8_{16}$ eingeben
3. Vom Akku-Inhalt das Zweierkomplement bilden
4. $3\,2_{16}$ hiervon über SUB-Befehl subtrahieren

2. 1 1 0 1 0 1 1 0
R_7 bis R_0

Lösungen zu Experiment 4

1. a)
   1. 2 1_{16} an B_7 bis B_0 einstellen
   2. Über DEPOSIT B-Daten in eine Adresse laden, z.B. Adresse 1 1 1 1
   3. 3 4_{16} an B_7 bis B_0 einstellen
   4. Über INP-Befehl Akku mit B-Daten laden
   5. SUB-Befehl mit Adresse 1 1 1 1 einstellen

   In R_7 bis R_0 erscheint das Ergebnis 0 0 0 1 0 0 1 1 ≙ $1\,3_{16}$

   b)
   1. 5 4_{16} an B_7 bis B_0 einstellen
   2. Über DEPOSIT B-Daten in eine Adresse laden, z.B. Adresse 1 1 1 1
   3. 2 4_{16} an B_7 bis B_0 einstellen
   4. Über INP-Befehl Akku mit B-Daten laden
   5. Über CMA-Befehl Akku-Inhalt komplementieren
   6. Über INC-Befehl 1 zum Akku-Inhalt addieren
      In R_7 bis R_0 steht jetzt das Zweierkomplement von 2 4_{16} (1 1 0 1 1 1 0 0)
   7. SUB-Befehl mit Adresse 1 1 1 1 einstellen und System takten
      In R_7 bis R_0 erscheint das Ergebnis 1 0 0 0 1 0 0 0 ≙ -78_{16}

2.
   1. Inhalt der Adresse 0 1 1 1 über EXAMINE feststellen z.B. 0 1 0 1 0 0 0 0
   2. Über LDA-Befehl Inhalt Adresse 0 1 1 1 in Akku laden
   3. Über INP-Befehl Akku-Inhalt plus 1 bilden
   4. Neuen Akku-Inhalt über STA-Befehl in Adresse 0 1 1 1 laden

3.
   1. 2 4_{16} in Speicher laden, z.B. Adresse 0 0 0 0
   2. 1 7_{16} in Speicher laden, z.B. Adresse 0 0 0 1
   3. 3 5_{16} in Akku laden
   4. Über ADD-Befehl mit Adresse 0 0 0 1 die Rechenoperation
      3 5_{16} + 1 7_{16} durchführen
   5. Über SUB-Befehl mit Adresse 0 0 0 0 2 4_{16} vom Zwischenergebnis subtrahieren
      In R_7 bis R_0 erscheint das Ergebnis 0 0 1 0 1 0 0 0 ≙ 28_{16}

Lösungen zu Experiment 5

1. a)

Adresse	Inhalt	Befehl	Kommentar
0	1 1 0 1 *x x x x*	INP	Lade B-Eingänge in Akku
1	1 1 1 0 1 1 1 1	STA	Speichere Akku-Inhalt in Adresse F
2	0 1 1 1 1 1 1 1	ADD	Addiere Akku mit Adresse F
3	0 1 1 1 1 1 1 1	ADD	Addiere Akku mit Adresse F
4	0 1 1 1 1 1 1 1	ADD	Addiere Akku mit Adresse F
5	0 1 1 1 1 1 1 1	ADD	Addiere Akku mit Adresse F
6	1 1 1 1 *x x x x*	HLT	HALT

b)

Adresse	Inhalt	Befehl
0	1 1 0 1 *x x x x*	INP
1	1 1 1 0 1 1 1 1	STA
2	0 1 1 1 1 1 1 1	ADD
3	0 1 1 1 1 1 1 1	ADD
4	0 1 1 1 1 1 1 1	ADD
5	0 1 1 1 1 1 1 1	ADD
6	0 1 1 1 1 1 1 1	ADD
7	0 1 1 1 1 1 1 1	ADD
8	0 1 1 1 1 1 1 1	ADD
9	1 1 1 1 *x x x x*	HLT

2. Kriterium für die Aufgabenstellung ist ein größer/kleiner-Vergleich der Eingabe B_3 bis $B_0 = n$ mit der Zahl $9_{10} \triangleq 1\ 0\ 0\ 1$. Ist $n > 9_{10}$, handelt es sich um eine Pseudotetrade, bei $n < 9_{10}$ handelt es sich um ein Codewort des 8421-Codes. Anhand der Subtraktion $n - 9_{10}$ kann die entsprechende Entscheidung getroffen werden. Damit ergibt sich folgendes Flußdiagramm (Bild 4).

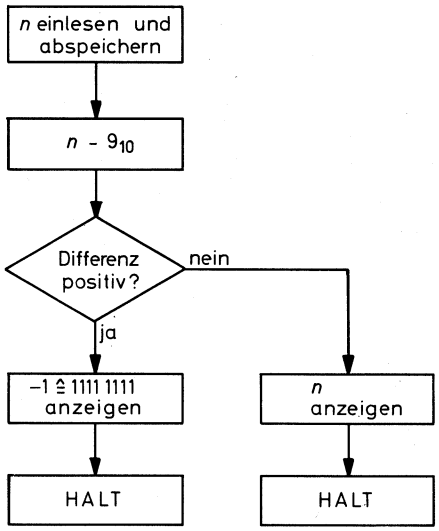


Bild 4
Flußdiagramm zu Aufgabe 2

Adresse	Inhalt	Befehl	Kommentar
0	1 1 0 1 *x x x x*	INP	B_3 bis B_0 in Akku laden
1	1 1 1 0 1 1 1 0	STA	Akku in Adresse E speichern
2	1 0 0 0 1 1 1 1	SUB	Inhalt Adresse F von Akku subtrahieren
3	1 0 0 1 1 1 0 1	AND	höchste Stelle ausblenden
4	0 1 1 1 1 1 0 0	ADD	Entscheidungsaddition
5	1 1 1 0 0 1 1 1	STA	Ergebnis in Adresse 7 abspeichern
6	0 0 1 1 1 1 1 0	LDA	B_3 bis B_0 in Akku laden
7	– – – – – – – –	ADD/HLT	Entscheidungsadresse
8	1 1 0 0 *x x x x*	SM1	Akku −1 setzen
9	1 1 1 1 *x x x x*	HLT	HALT
A	*x x x x x x x x*		nicht belegt
B	*x x x x x x x x*		nicht belegt
C	0 1 1 1 0 0 0 0		DATEN
D	1 0 0 0 0 0 0 0		
E	B_3 bis B_0		
F	0 0 0 0 1 0 0 1		

Da bei diesem Programm als Entscheidungskriterium $n - 9_{10}$ gewählt wurde, entsteht der Nachteil, daß bei B_3 bis B_0 gleich $1\ 0\ 0\ 1 \triangleq 9_{10}$ eine Pseudotetrade angezeigt wird. Das ist aber falsch. Wenn wir als Kriterium $9_{10} - n$ wählen, wird dieser Nachteil aufgehoben. Damit ergibt sich folgendes Programm:

Adresse	Inhalt	Befehl	Kommentar
0	1 1 0 1 *x x x x*	INP	B_3 bis B_0 in Akku laden
1	1 1 1 0 1 1 1 0	STA	Akku in Adresse E speichern
2	0 0 1 1 1 1 1 1	LDA	Inhalt Adresse F in Akku laden
3	1 0 0 0 1 1 1 0	SUB	Inhalt Adresse E von Akku subtrahieren
4	1 0 0 1 1 1 0 1	AND	höchste Stelle ausblenden
5	0 1 1 1 1 1 0 0	ADD	Entscheidungsaddition
6	1 1 1 0 1 0 0 0	STA	Ergebnis in Adresse 8 abspeichern
7	1 1 0 0 *x x x x*	SM1	Akku −1 setzen
8	– – – – – – – –	ADD/HLT	Entscheidungsadresse
9	0 0 1 1 1 1 1 0	LDA	Inhalt Adresse E in Akku laden
A	1 1 1 1 *x x x x*	HLT	HALT
B	*x x x x x x x x*		nicht benutzt
C	0 1 1 1 0 0 0 0		DATEN
D	1 0 0 0 0 0 0 0		
E	B_3 bis B_0		
F	0 0 0 0 1 0 0 1		