Heft 03
Kombinatorische Digitalschaltungen

Stand: 1.02

- Vorläufiges Exemplar. Nur zur Information -

1. Grundschaltungen

1.1. Funktionselemente

1.1.1. Grundgatter

Grundgatter sind logische Funktionselemente, die in Form von Gatterschaltkreisen oder elementaren Zellenstrukturen (in programmierbaren und anwendungsspezifischen Schaltkreisen) verfügbar sind. Kennzeichnende *funktionelle* Merkmale sind die jeweilige logische Verknüpfung und die Anzahl der Eingänge.Um die Anzahl der Eingänge zu bezeichnen, spricht man z. B. von einem Zweifach-NAND, einem Vierfach-NOR usw. Zu den *technischen* Merkmalen gehören u. a. die Verzögerungszeit, die Speisespannung, die Stromaufnahme sowie die Lastkennwerte der Ein- und Ausgänge.

1.1.2. Leistungsgatter und Treiber

Gelegentlich werden Schaltkreisausgänge in außergewöhnlichem Maße belastet. Dies ist zum einen der Fall, wenn viele Schaltkreiseingänge anzusteuern sind (typische Beispiele dafür sind Rücksetz- und Taktsignale, die in der gesamten Schaltung an vielen Stellen "verbraucht" werden). Zum anderen treten solche Betriebsverhältnisse oft an den Schnittstellen zur Außenwelt auf. Hierfür gibt es Schaltkreise mit entsprechend leistungsfähigen Ausgangsstufen: Leistungsgatter (Power Gates) und Treiber (Drivers), wobei zwischen invertierenden und nichtinvertierenden Treibern unterschieden wird.

1.1.3. Schmitt-Trigger-Eingänge

Signaländerungen an Digitalschaltkreisen müssen eine vorgeschriebene Mindest-Flankensteilheit haben. Nun haben nicht alle Signale, die eine Digitalschaltung aus ihrer Umgebung bezieht, Flankensteilheiten, die den jeweiligen Anforderungen entsprechen. Um aus beliebig langsamen ("lahmen") Signalverläufen am Eingang exakte, steile Impulsflanken am Ausgang zu machen, braucht man Schaltungen, die ein sog. Schwellwertverhalten zeigen. Unterschreitet das Eingangssignal einen vorgegebenen Schwellwert (Threshold), so führt der Ausgang beispielsweise Low-Signal, überschreitet es ihn, schaltet der Ausgang mit hoher Flankensteilheit auf High um. Die ausgangsseitige Flankensteilheit ist dabei unabhängig von der eingangsseitigen.

Zwei Schwellwerte: die Hysterese
In der Praxis sind alle Schaltvorgänge mit Schwankungen behaftet. Ein Schaltung, die einen einzigen Schwellwert für das Umschalten von Low nach High und von High nach Low aufweist, wird somit gelegentlich unsicher arbeiten (genauer: ausgangsseitig zu Schwingungen neigen). Nehmen wir an, das Eingangssignal durchläuft den Bereich des Schwellwertes sehr langsam. Nun kann es vorkommen, daß, wenn der Ausgang gerade umgeschaltet hat, der Schwellwert um einen geringen Betrag absinkt. Dann schaltet der Ausgang sofort wieder zurück usw. Abhilfe (Abbildung 1.1): man legt die Schaltung so aus, daß sie 2 Schwellwerte hat, einen für den (eingangsseitigen) Übergang von Low nach High (U_{ON}) und einen für den Übergang von High

nach Low (U_{OFF}). Dabei gilt stets $U_{ON} > U_{OFF}$ (Hysterese). Die Wirkung: nach dem ausgangsseitigen Umschalten, z. B. von Low nach High, werden geringe Schwankungen des Eingangssignals (oder des Schwellwertes) nicht dazu führen, daß der Ausgang sofort wieder zurückschaltet (vielmehr muß das Eingangssignal zunächst einmal merklich absinken, d. h. den Schwellwert U_{OFF} unterschreiten)

Abbildung 1.1 Schwellwert-Schaltverhalten mit Hysterese (Prinzip)

Erklärung:

a) zum Begriff des Hysteresebandes. Oben: Beispiel eines (analogen) Eingangssignals; darunter: das daraus gebildete Digitalsignal.

b) ein von Störungen überlagertes Eingangssignal wird bei zu schmalem Hystereseband (erst reccht dann, wenn gar keine Hysterese vorgesehen ist) in mehrere Impulse je Signalperiode umgesetzt,

c) das Hystereseband ist breit genug, so daß nur ein Impuls je Signalperiode gebildet wird.

Eine einfache, kostengünstige Grundschaltung, die ein solches Verhalten aufweist, ist der sog. *Schmitt-Trigger*. Derartige Schaltungen braucht man heutzutage nicht mehr mit Einzelbauelementen zusammenzubasteln - es gibt entsprechende Digitalschaltkreise in Form von Gattern und Negatoren.

Auch manche höher integrierte Schaltkreise haben an bestimmten Eingängen (wo dies anwendungspraktisch sinnvoll ist), Schmitt-Trigger-Eingangsstufen (Schmitt Trigger Inputs).

Schmitt-Trigger sind in der Lage, beliebig langsame Einganssignale in Logiksignale mit angemessen steilen Flanken umzusetzen, es sind aber keine Präzisionsbauelemente (die Schwellwerte haben beachtliche Toleranzen). Sind präzise Schwellwerte einzuhalten, so sind Comparatoren einzusetzen (Heft 11).

1.1.4. Wechselseitige Wandlungen

Grundsätzlich reicht ein einziger Gattertyp aus, um jede beliebige Schaltfunktion zu verwirklichen, vorausgesetzt es handelt sich um eine UND- bzw. ODER-Verknüpfung gefolgt von einer Negation (vollständige Realisierungsbasis; vgl. Heft 01). Diese Tatsache wird technisch ausgenutzt, um mit wenigen Gattertypen auszukommen, manchmal sogar (namentlich in ASICs) mit einem einzigen (z. B. mit einem Zweifach-NAND). Abbildung 1.2 zeigt, wie man mit NAND- und NOR-Gattern verschiedene Funktionen verwirklichen kann. Darüber hinaus ist die wechselseitige Wandlung von NAND und NOR dargestellt.

Abbildung 1.2 Funktionen aus Grundgattern (Beispiele) und wechselseitige Wandlungen von Gatterfunktionen. a) NAND, b) NOR als Realisierungsbasis. Darunter jeweils andere Funktionen, die sich damit realisieren lassen

Positive und negative Logik; NAND und NOR
Ein NAND in positiver Logik entpricht einem NOR in negativer Logik und umgekehrt (*Dualität* von NAND und NOR als Konsequenz der DeMorgan'schen Regeln (Heft 1)).

1.1.5. Negierte und nicht negierte Signale

Oftmals braucht man Signale in beiden Polaritäten, d. h. sowohl in negierter als auch in nicht negierter Form. Die Abbildungen 1.3 bis 1.5 zeigen, wie man solche Signale bereitstellen kann.

Abbildung 1.3 Negationen, die man praktisch umsonst bekommt

Erklärung:
Die Abbildung zeigt anhand von 3 Beispielen, daß gelegentlich negierte und nicht negierte Signale*) aus ohnehin vorhandenen Schaltmiteln gleichsam abgezapft werden können.

*): andere Bezeichnung für ein solches Signalpaar: *komplementäre* Signale.

a) Prinzipdarstellung einer bipolaren Logikschaltung. Das Kästchen enthält die Verknüpfung der Eingangssignale. Üblicherweise ist je Ausgangssignal eine Anordnung aus 2 Transistoren vorgesehen. Um ein Low-Ausgangssignal abzugeben, wird der untere Transistor angesteuert, um ein High-Ausgangssignal abzugeben, der obere. Für jeden der beiden Transistoren wird ein Ansteuersignal gebildet. Beide Ansteuersignale könnte man direkt weiterverwenden.

b) Prinzipdarstellung eines ODER-Gatters in CMOS-Technologie. Ausgangsseitig sind zwei Negatoren hintereinandergeschaltet, so daß sich beide Signale abnehmen lassen (das "wahre"Signal repräsentiert die ODER-Verknüpfung, das negierte die NOR-Verknüpfung).

c) Flipflops und Latches (Heft 04) haben von Hause aus komplementäre Ausgänge (linkes Bild). Dies kann man auch ausnutzen, um ein negiert ankommendes Signal zu invertieren (indem man die Ausgangssignale genau anders herum anschließt; rechtes Bild).

Hinweis:
In Schaltungen, die dem RTL-Schema (Register - Kombinatorik - Register) entsprechen, braucht man - zumindest in der Theorie - gar keine Negatoren (die Flipflops stellen sowohl die "wahren" als auch die negierten Signale bereit, und die Kombinatorik kann mit zweistufigen UND-ODER-Netzwerken aufgebaut werden (Vergegenständlichung der DNF)).

Die Lösungen gemäß Abbildung 1.3a und b lassen sich nur im Innern von Schaltkreisen anwenden (sie werden dort auch ausgiebig genutzt). Die Lösung gemäß Abbildung 1.3c ist hingegen auch zwischen verschiedenen Schaltkreisen nutzbar - vorausgesetzt, beide Ausgänge der Flipflops bzw. Latches sind auch auf Anschlüsse (Pins) geführt.

Abbildung 1.4 Negierte und nicht negierte Signale mit geringem Zeitversatz (Skew) bereitstellen

Erklärung zu Abbildung 1.4:
Is nur eine Polarität verfügbar, muß man die andere durch Negieren gewinnen. Oftmals besteht die Forderung, daß beide Signale eines komplementären Paares keinen Zeitversatz (Skew) gegeneinander haben.

a) das Nachschalten eines Negators ist einfach, hat aber unvermeidlich einen merklichen Zeitversatz (Skew) zur Folge (das negierte Signal ist gegenüber dem ursprünglichen um die Durchlaufzeit t_p des Negators verzögert (typischerweise um 2...7 ns).

b) XOR-Gatter gewährleisten eine näherungsweise gleich lange Verzögerung des "wahren" und des negierten Signals. Der Zeitversatz (Skew) zwischen beiden Signalen beträgt typischerweise 1...3 ns (Baureihen F oder AC, beide XOR-Gatter in einem Schaltkreis); dabei wird das negierte Signal etwas mehr verzögert.

c) spezielle Takttreiberschaltkreise sind ausdrücklich auf geringsten Zeitversatz (Skew) hin ausgelegt. Hier ist der Typ CDC328A von Texas Instruments dargestellt. Das eingangsseitige Taktsignal (hier: das zu prüfende Signal P) wird an 6 Ausgänge geliefert, die über XOR-Gatter paarweise als negierend oder als nicht negierend konfiguriert werden können. Der Zeitversatz (Skew) zwischen zwei beliebigen Ausgängen ist nicht größer als 1 ns.

d) handelt es sich um viele Signale, bieten negierende und nicht negierende Treiberschaltkreise eine kostengünstige Lösung bei noch erträglichem Zeitversatz (ABT-Baureihe: um 4 ns).

Negieren im Innern von CMOS-Schaltkreisen
In kombinatorischen CMOS-Schaltungen erfordert die Negation stets das Nachschalten eines Negators (d. h. eines komplementären Transistorpaares; vgl. Abbildung 1.3b). Der Skew bleibt aber gering (Länge der Signalwege höchstens im mm-Bereich, es sind keine Schaltkreisanschlüsse zu treiben usw.). Infolge der geringen kapazitiven Belastung liegt der Zeitversatz typischerweise weit unter 1 ns.

1.2. Kaskadierung

Kaskadierung bedeutet hier, aus Schaltungen mit vergleichsweise wenigen Eingängen funktionell gleichartige Schaltungen mit entsprechend mehreren Eingängen aufzubauen.

Kaskadierung von Grundgattern
Oftmals braucht man Verknüpfungen, wie UND, ODER, NAND, NOR usw. mit mehr als zwei Eingängen. So etwas läßt sich durch entsprechende elektrische Auslegung der Gatter erreichen. Es gelingt aber auch, Gatter mit mehr als zwei Eingängen aus Gattern mit zwei Eingängen und (wenn erforderlich) aus Negatoren aufzubauen (Abbildung 1.5).

Abbildung 1.5 Kaskadierung

Erklärung zu Abbildung 1.5:
Die Abbildung zeigt, wie aus Gattern mit 2 Eingängen Gatter-Funktionen mit 4 Eingängen aufgebau werden können. Nichtnegierende gleichartige Gatter kann man ohne weiteres hintereinanderschalten. Handelt es sich um negierende Gatter, sind jeweils Negatoren zwischenzuschalten (Abbildung 1.6). Grundsätzlich lassen sich Gatter mit beliebiger Eingangszahl durch fortgesetztes Kaskadieren aufbauen.

Abbildung 1.6 Kaskadierungsbeispiele: 5-fach-NAND und 5-fach-NOR

Wieviele Gatter braucht man wirklich?
Sehen wir uns Abbildung 1.6 genauer an, so ist erkennbar, daß eine vorgegebene Anzahl Grundgatter, namentlich bei geringer Eingangszahl, schnell aufgebraucht sein wird. Dies ist besonders dann von Bedeutung, wenn man FPGA- oder ASIC-Technologien nutzen will. Da mögen beispielsweise die 2000 Zweifach-NANDs eines Gate-Array-Schaltkreises zunächst viel erscheinen. Das ist aber die einzige Grundlage, aus der alle anderen Verknüpfungen aufgebaut werden müssen! Allerdings sieht es in der Praxis nicht ganz so pessimistisch aus wie die Abbildung vermuten läßt. So braucht man aus logischer Sicht für ein Fünffach-NAND 7 Zweifach-NANDs, im ASIC jedoch nur vier, da schaltkreisintern die Transistorstruktur, die das Zweifach-NAND bildet, trickreich ausgenutzt wird (für die Verdrahtung im Schaltkreis sind alle Anschlußpunke der Transistoren zugänglich, so daß man keine besonderen Negatoren benötigt; vgl. auch Abbildung 1.3).

Kaskadierung der Antivalenz
Antivalenzgatter kann man (wie UND bzw. ODER) auf beliebige Weise - ohne Zwischenschaltung von Negatoren - kaskadieren (Abbildung 1.8).

Abbildung 1.7 Kaskadierung der Antivalenz

Aus der Abbildung ist eine wichtige funktionelle Eigenschaft einer kaskadierten bzw. Mehrfach-Antivalenzverknüpfung erkennbar: der Ausgang führt stets dann den Signalwert 1, wenn eine ungerade Anzahl der Eingänge mit 1 belegt ist. (Technische Anwendung: Paritätskontrolle. Eine weitere Eigenart: eine Antivalenzfunktion über n Variable hat eine Lösungsmenge der Mächtigkeit 2^{n-1}; dies bereitet Schaltalgebra-Programmen (in Entwicklungsumgebungen) gelegentlich einige Probleme (vgl. Heft 01).)

Mehrfachkaskadierung
Sind Schaltfunktionen mit größerer Eingangszahl durch Kaskadieren zu verwirklichen, so hat man die Wahl zwischen 2 grundsätzlichen Strukturen (Abbildung 1.8).

Abbildung 1.8 Mehrfachkaskadierung

Erklärung:

a) Kettenschaltung (Daisy Chain bzw. Ripple Thru),
b) Binärbaum (Binary Tree). 2 Beispiele; oben für 4, unten für 8 Variable.

Wir nehmen hierbei an, daß die kaskadierte Struktur jeweils mit gleichartigen Gattern aufgebaut wird. Handelt es sich um negierende Gatter, so nehmen wir an, jedem Gatter sei ein - in der Abbildung nicht dargestellter - Negator nachgeschaltet (den wir einfach zum Gatter hinzurechnen).

Ersichtlicherweise erfordert es stets n - 1 Gatter mit 2 Eingängen, um ein Gatter mit n Eingängen aufzubauen. Worin sich beide Strukturen wesentlich unterscheiden, ist die Schaltungstiefe (und damit die Schaltverzögerungszeit):

- die Schaltungstiefe s der Kettenschaltung wächst linear mit der Eingangszahl (O(n)). Bei n Eingängen ist $s = n - 1$*).
- die Schaltungstiefe s der Baumstruktur wächst gemäß dem Zweierlogarithmus der Eingangszahl (O(ld n)). Bei n Eingängen ist $s = \text{ceil}(\text{ld } n)$*).

*): ohne Berücksichtigung technischer Nebenbedingungen (zwischenzuschaltende Negatoren, ggf. erforderliche Treiberstufen usw.).

Damit ist - was die Geschwindigkeit angeht, die Baumstruktur der Kettenschaltung grundsätzlich überlegen, und zwar umso mehr, je größer die Eingangszahl ist. Beispiel: Bei 32 Eingängen ist bei Wahl der Kettenschaltung s = 31, bei Wahl der Baumstruktur s = 5.

Die Probleme der Baumstruktur zeigen sich aber dann, wenn Folgen gleichartiger Verknüpfungen über gemeinsame Eingangssignale zu bilden sind. Genauer: es sind mehrere Ausgangssignale zu bilden, und das jeweils nachfolgende Ausgangssignal hängt vom jeweils vorhergehenden ab (typische Beispiele sind Prioritätsnetzwerke, die Übertragsweitergabe beim Addieren, Zählnetzwerke usw.; vgl. die Kapitel 2 und 3 sowie Heft 04). Derartige funktionelle Abhängigkeiten legen eine Kettenstruktur nahe (in Abbildung 1.8a durch die gestrichelten Linien angedeutet). Der Vorteil: die jeweils vorhergehenden Gatter werden für die nachfolgenden Verknüpfungen mitgenutzt.

Beispiel: es sind 31 Ergebnisbits zu bilden (Bit 1 durch Verknüpfung von a_0 und a_1, Bit 2 durch Verknüpfung von Bit 1 mit a_2 usw.). Dies führt zunächst auf eine Struktur gemäß Abbildung 1.8a. Diese kommt mit 31 Gattern aus, wenngleich sich (für Bit 31) eine Schaltungstiefe s = 31 ergibt. Eine Struktur gemäß Abbildung 1.8b braucht hingegen für Bit 1 ein Gatter (2 Variable), für Bit 2 zwei Gatter (3 Variable) usw. Bit 31 erfordert somit 31 Gatter (32 Variable $a_0 ... a_{31}$). Insgesamt brauchen wir also 1 + 2 + 3 +... + 31 Gatter, also allgemein für n Variable:

$$\frac{(n-1)^2 + (n-1)}{2} \text{ Gatter.}$$

Für 32 Variable ergeben sich somit 496 Gatter. Diese Aufwand (Wachstum gemäß $O(n^2)$) setzt der Anwendbarkeit von Baumstrukturen in solchen Schaltungen Grenzen, so daß typischerweise Kompromißlösungen zwischen Baum- und Kettenstrukturen gewählt werden. Beispielsweise liegt es nahe, für die Verknüpfung von 8 Variablen in Abbildung 1.8b (unten) die Verknüpfung der ersten 4 Variablen (oben) mitzunutzen (vgl. die gestrichelte Linie; die untere Verknüpfung

der Variablen $a^0 ... a^3$ könne somit entfallen). In der Schaltungspraxis sind die Grenzen dieser Mitnutzung durch Leitungslängen, kapazitive Belastung und verfügbare Verdrahtungsressourcen gegeben.

1.3. Elementare Schaltungsstrukturen

1.3.1. Verknüpfungen in disjunktiver Normalform

UND-ODER
Es liegt nahe, die konjunktiven Verknüpfungen mit UND-Gattern und die disjunktive Verknüpfung mit einem nachgeschalteten ODER-Gatter zu realisieren (Abbildung 1.9)

Abbildung 1.9 Verknüpfung in disjunktiver Normalform als UND-ODER-Struktur. Beispiel: $ab \vee cd$

NAND-NAND
Sowohl die Konjunktionen als auch die Disjunktion lassen sich mit NAND-Gattern verwirklichen (Abbildung 1.10).

Abbildung 1.10 Verknüpfung in disjunktiver Normalform als NAND-NAND-Struktur. Beispiel: $ab \vee cd$

Erklärung:
1 - NAND wirkt als Disjunktion für invertierte Signale (Ausgang = 1, falls wenigstens ein Eingang = 0); 2 - NAND wirkt als Konjunktion und liefert das zum "Odern" erforderliche negierte Signal (Ausgang = 0, falls alle Eingänge = 1, d. h. falls die UND-Bedingung erfüllt ist).

Hinweis:
Wir erkennen hieran, weshalb das NAND-Gatter als Realisierungsbasis so beliebt ist - NANDS reichen vollkommen aus, um Verknüpfungen in disjunktiver Normalform zu realisieren (NAND - NAND $\triangleq$ UND - ODER).

NOR-NOR
Ein NOR wirkt als konjunktive Verknüpfung negierter Signale (Ausgang = 1, falls alle Eingänge = 0). Demgemäß kann man disjunktive Nomalformen auch mit NOR-Gattern aufbauen. Allerdings kommen wir hierzu nicht ohne zusätzliche Negation aus (Abbildung 1.11).

Abbildung 1.11 Verknüpfung in disjunktiver Normalform als NOR-NOR-Struktur. Beispiel: $ab \vee cd$

Erklärung:
1 - konjunktive Verknüpfung der invertierten Eingangssignale (2. DeMorgansche Regel; $\overline{\overline{a} \vee \overline{b}} = a \cdot b$); 2 - disjunktive Verknüpfung, die wiederum ein invertiertes Ausgangssignal liefert.

Eine zweistufige NOR-NOR-Struktur entspricht einer AND-OR-INVERT-Struktur (s. weiter unten). Sie bildet die Negation der als DNF gegebenen Schaltfunktion. Die Eingangssignale sind folgendermaßen an das NOR-NOR-Netzwerk anzuschließen:

- **#** eine nicht negierte Variable in der Booleschen Gleichung entspricht einem negierten Signal,
- **#** eine negierte Variable in der Booleschen Gleichung entspricht einem nicht negierten Signal.

Wie werden die negierten Signale gebildet?

a) wir arbeiten grundsätzlich nur mit invertierten Signalen (Konventionssache). Da eine NOR-NOR-Anordnung ohehin ein negiertes Ausgangssignal liefert, ist ggf. nur an den Schnittstellen zur Außenwelt eine Negation erforderlich*).

b) es werden entsprechend Negatoren eingefügt,

c) die Negation ergibt sich gleichsam umsonst durch entsprechendes Anschließen an die invertierten Ausgänge der vorgeschalteten Flipflops. Ein nachgeschaltetes Flipflop liefert an seinem negierten Ausgang das nicht negierte ("wahre") Ausgangssignal. In Register-Transfer-Strukturen (Flipflops - NOR-NOR- Flipflops - NOR-NOR usw.) werden die negierten Signale abwechselnd von den negierten und nicht-negierten Ausgängen der Flipflops geliefert (Abbildung 1.12).

*): in anderer Sichtweise: wir wechseln die Zuordnung von Pegeln und logischen Werten (z. B. von positiver zu negativer Logik) und können so mit NANDs anstelle von NORs rechnen (Ausnutzung der DeMorganschen Regeln).

Abbildung 1.12 Register-Transfer-Struktur mit NOR-Gattern

Erklärung:
1 - die negierten Flipflop-Ausgänge liefern die negierten Eingangssignale; 2 - zweistufiges NOR-NOR-Netzwerk gemäß DNF; 3 - dieses Flipflop übernimmt vom Netzwerk 2 ein negiertes Ausgangssignal - deshalb wird das negierte Eingangssignal des nachfolgenden NOR-NOR-Netzwerks am nicht negierten Flipflop-Ausgang abgegriffen.

AND-OR-INVERT
Einer UND-ODER-Struktur ist ein Negator nachgeschaltet (AND-OR-Inverter; AOI). Somit ensteht ein invertiertes Ausgangssignal, das ggf. negiert werden muß (wird es in einem Flipflop gespeichert, so hat man an den Flipflop-Ausgängen die Negation gleichsam umsonst). Der AND-OR-Inverter war die übliche Grundschaltung zu jener Zeit, als Gatter mit Dioden und Negatoren mit Röhren oder Transistoren aufgebaut wurden.(Nach zwei hintereinandergeschalteten Dioden ist es erforderlich, die Signalpegel wieder zu regenerieren - hierzu wird ein Negator nachgesetzt.)

1.3.2. Tricks der Schaltungsoptimierung

Klassische Minimierungsverfahren
Verfahren gemäß Karnaugh/Veitch, Quine/McCluskey usw. führen gelegentlich zu weniger aufwendigen zweistufigen Netzwerken (Abbildung 1.13).

Abbildung 1.13 Optimierungsbeispiel

Erklärung:

a) das ursprüngliche Netzwerk (in ausführlicher Darstellung),
b) das nach Quine/McCluskey minimierte Netzwerk (vgl. auch Heft 1),
c) das minimierte Netzwerk in vereinfachter Darstellung.

Die KDNF der ursprünglichen Schaltfunktion:

$$a\,\overline{b}\,\overline{c}\,\overline{d} \vee a\,b\,\overline{c}\,\overline{d} \vee a\,\overline{b}\,\overline{c}\,d \vee a\,b\,\overline{c}\,d \vee \overline{a}\,b\,c\,\overline{d} \vee \overline{a}\,\overline{b}\,\overline{c}\,d \vee \overline{a}\,\overline{b}\,\overline{c}\,\overline{d}$$

Die DNF der minimierten Schaltfunktion:

$$\overline{a}\,b\,c\,\overline{d} \vee \overline{b}\,\overline{c} \vee a\,\overline{c}$$

Mehrstufige Netzwerke
Gelegentlich läßt sich der Schaltungsaufwand verringern, indem man von der disjunktiven Normalform bzw. vom zweistufigen Gatternetzwerk abgeht (Abbildungen 1.14, 1.15)

Abbildung 1.14 Optimierungstricks

Erklärung:

a) Vorziehen. Gleichartige Teil-Konjunktionen (Variablenbelegungen, die in mehreren Konjunktionstermen gemeinsam vorkommen), kann man in besonderen Gattern zusammenfassen, die dem zweistufigen Netzwerk vorgeschaltet werden.
b) Nachsetzen. Einzelne Variable, die in mehreren bzw. allen Konjunktionstermen gleichermaßen vorkommen, werden aus der zweistufigen Verknüpfung herausgenommen. Die erforderlichen konjunkiven Verknüpfungen werden ausgangsseitig nachgeschaltet. Das Nachsetzen ist im besonderen bei Signalen üblich, die als Takt- oder Strobe-Impuls wirken. Man hat dann die jeweils kürzeste Verzögerungszeit vom Impuls zum Ausgang.

Abbildung 1.15 Optimierungsbeispiel

Erklärung zu Abbildung 1.15:

a) die ursprüngliche Schaltfunktion als zweistufiges UND-ODER-Netzwerk,
b) Aufwandsverringerung durch Vorziehen gemeinsamer Teil-Konjunktionen,
c) weitere Vereinfachung.

Die ursprüngliche Schaltfunktion (vgl. auch Heft 1):

$$f = a\,\overline{b}\,c\,d\,e \vee \overline{a}\,\overline{b}\,c\,d\,\overline{e} \vee a\,b\,\overline{c}\,\overline{d}\,e \vee a\,b\,\overline{c}\,d\,e \vee a\,b\,c\,\overline{d}\,e$$

Die Verknüpfung *a b e* kommt in 3 Implikanden vor. Also läßt sich die Gleichung umformen:

$$f = a\,\overline{b}\,c\,d\,e \vee \overline{a}\,\overline{b}\,c\,d\,\overline{e} \vee a\,b\,e\,(\overline{c}\,\overline{d} \vee \overline{c}\,d \vee c\,\overline{d})$$

Wie man sofort sieht, läßt sich der eingeklammerte Ausdruck weiter vereinfachen, indem man dessen Negation (c d) verwirklicht und diese nochmals negiert:

$$f = a\,\overline{b}\,c\,d\,e \vee \overline{a}\,\overline{b}\,c\,d\,\overline{e} \vee a\,b\,e\,\overline{c\,d}$$ (Abbildung 1.15b)

Es kann sein, daß weitere Umformungen zu weiteren Vereinfachungen führen (ob dies der Fall ist oder nicht, hängt vom jeweils verfügbaren Bauelementesortiment ab):

$$f = \overline{b}\,c\,d\,(a\,e \vee \overline{a}\,\overline{e}) \vee a\,b\,e\,\overline{c\,d}$$

Der hier eingeklammerte Ausdruck entspricht offensichtlich der Äquivalenz bzw. (da es keine Äquivalenzgatter gibt) der invertierten Antivalenz (XOR):

$$f = \overline{b}\,c\,d\,(\overline{a \oplus e}) \vee a\,b\,e\,\overline{c\,d}$$ (Abbildung 1.15c).

Zur Sinnfälligkeit einschlägiger Optimierungsversuche

Herkömmliche Technologie (SSI/MSI)
Wichtigstes Kriterium (Faustregel): sowenig Schaltkreisgehäuse wie möglich - gleichgültig, was sie enthalten (Minumum Package Count).

Praxistips:

1. UND bzw. ODER mit mehr als 4 Eingängen auflösen (74xx-Baureihen: die “dicksten” NANDs haben 8 bzw. 13 Eingänge - es gibt sie aber nicht in in allen Technologien, zudem belegt so ein Gatter einen ganzen Schaltkreis),
2. in ohnehin zu bestückenden Schaltkreise enthaltene Gatter ausnutzen. Dabei darf auch getrickst werden (aber vorsichtig...). 1. Beispiel: Gatter als Negatoren (hierzu freie Eingänge entsprechend beschalten - vgl. Heft 1). 2. Beispiel: wir brauchen ein

Zweifach-ODER, es ist aber nur ein XOR (7486) frei. Dieses kann verwendet werden, sofern ausgeschlossen ist, daß die Eingangsbelegung 1, 1 jemals auftritt.

Grenzen solcher Optimierungsversuche (wenn sie womöglich mehr Schaden als Nutzen bringen):

- es müssen eigens Schaltkeisgehäuse gesetzt werden (Beispiel: das XOR in Abbildung 1.15c - wenn allein deswegen ein 7486 bestückt werden müßte, lohnt es sich womöglich nicht*),
- die ausnutzbaren Gatter sind derart über die Leiterplatte verstreut, daß die Verbindungen entweder zu lang werden oder sich gar nicht verlegen lassen.

*): *Übungsaufgabe:*
Was könnte man mit einem eigens gesetzten 7486 (= 4 Zweifach-XORs) in Hinsicht auf die Schaltung von Abbildung 1.15c noch anfangen?

Die verbleibenden 3 XORs könnten als Negatoren eingesetzt werden - wir hätten so genau die 3 Negatoren, die wir in der Schaltung brauchen.

CPLDs
Die UND-Gatter haben meist genügend viele Eingänge, so daß sich Vorziehen, Nachsetzen usw. nicht lohnt. Die Entwurfssoftware wird typischerweise Schaltungen ähnlich Abbildung 1.15c wieder in zweistufige UND-ODER-Netzwerke umformen.

FPGAs und ASICs
Die Entwurfssoftware sollte automatisch optimieren können.

Praxistips zu CPLDs, FPGAs und ASICs:

1. Wichig ist, wie die Zellen aufgebaut sind und was sich in der einzelnen Zelle unterzubringen ist. Daraus ergeben sich ggf. Optimierungsziele. Beispiel Xilinx 9500: jede Schaltfunktion mit bis zu 36 Variablen paßt in eine einzige Zelle, sofern deren DNF nicht mehr als 5 Konjunktionsterme (Implikanden) enthält.
2. Die Entwurfssoftware ist teuer genug - weshalb soll man da noch von Hand rechnen. Überprüfen Sie, was sie leistet (anhand überschaubarer Probeentwürfe).
3. Verlieren Sie keine Zeit mit Handoptimierungen im Sinne von Abbildung 1.15. Was aber sinnvoll ist: zu überlegen, ob sich auf der Ebene der funktionellen Zusammenhänge etwas vereinfachen läßt (Stichworte: Don't-Care-Bedingungen, Signalbelegungen die nie vorkommen oder sich gegenseitig ausschließen, allgemeine "Entfeinerung" = Erkennen/Abstellen unnötiger Kompliziertheit).

Elementare Verknüpfungen invertiert wirkender (Active-Low-) Signale
Manche Signale sind als aktiv-Low spezifiziert (das ist z. B. typisch für Schaltkreisauswahlsignale (Chip Enables, Chip Selects) von Treibern und Speichern, für Speicher-Schreibimpulse, für Rücksetzsignale usw.). Nicht selten ist es erforderlich, elementare

Verknüpfungen derartiger Signale zu realisieren. Hierzu lassen sich die DeMorgan'schen Regeln vorteilhaft ausnutzen (Abbildung 1.16).

Abbildung 1.16 Elementare Verknüpfungen von Signalen, die aktiv-Low wirken

Erklärung:

a) ein UND-Gatter wirkt als ODER: ist schon einer der Eingänge aktiv (Low), so wird auch der Ausgang aktiv (Low). Anwendungsbeispiel (1): das "Zusammen-Odern" eines generellen und eines funktionellen Rücksetzsignals. Anwendungsbeispiel (2): das Bilden eines Schaltkreisauswahlsignals (CE#) sowohl bei Befehls- als auch bei Datenzugriff eines Mikrocontrollers (z. B. 8051 - in diese Applikation fließen sicherlich jedes Jahr einige Millionen SMD-Einzelgatter).

b) ein ODER-Gatter wirkt als UND: nur dann, wenn beide Eingänge aktiv (Low) sind, wird der Ausgang aktiv (Low). Anwendungsbeispiel: konjunktive Verknüpfung eines generellen Schreibimpulses mit einem Schreibauswahlsignal zur Ansteuerung eines SRAMs. Des weiteren eignet sich das ODER-Gatter dazu, die Wirkung eines aktiv-Low-Signales zu blockieren (Sperrgatter): es genügt, daß *ein* Eingang mit High belegt ist, um zu verhindern, daß der Ausgang aktiv (Low) wird.

c) ein NAND-Gatter wirkt als ODER mit aktiv-High-Ausgang für aktiv-Low-Eingangssignale. Anwendungsbeispiel: ODER-Verknüpfung, wenn nur invertierte Signale vorliegen (billiger als 2 · Negator + ODER-Gatter).

d) ein NOR-Gatter wirkt als UND mit aktiv-High-Ausgang für aktiv-Low-Eingangssignale. Anwendungsbeispiel: UND-Verknüpfung, wenn nur invertierte Signale vorliegen (billiger als 2 · Negator + UND-Gatter).

Das XOR als steuerbarer Inverter/Noninverter

Ein XOR-Gatter kann als steuerbarer Inverter eingesetzt werden (Abbildung 1.17).

Abbildung 1.17 XOR als steuerbarer Inverter

Erklärung:

In den Signalweg wird ein XOR geschaltet. INVERT = 0: Signal wird nicht negiert: 0 am Eingang wird zu 0 am Ausgang ($0 \oplus 0 = 0$), 1 am Eingang wird zu 1 am Ausgang ($1 \oplus 0 = 1$). INVERT = 1: Signal wird negiert: 0 am Eingang wird zu 0 am Ausgang ($0 \oplus 1 = 1$), 1 am Eingang wird zu 1 am Ausgang ($1 \oplus 1 = 0$).

1.4. Elementare Funktionen

1.4.1. Auswählen

Der Datenselektor
Die Aufgabe besteht darin, einen von mehreren Eingängen zu einem Ausgang durchzuschalten. Das heißt: am Ausgang muß eine Eins erscheinen, wenn der betreffende Eingang eine Eins führt UND wenn er ausgewählt ist. Man muß also jeden Eingang mit einem Auswahl-Steuersignal konjunktiv verknüpfen und alle diese Verknüpfungen disjunktiv zusammenfassen ("odern"). Abbildung 1.18 zeigt die grundsätzliche Schaltung. Die Steuersignale liegen hier einzeln vor; für jeden auszuwählenden Eingang bzw. bei Parallelanordnung für jede Gruppe zusammengehörender Eingänge gibt es ein Steuersignal (SELECT..). Eine solche Anordnung heißt *Datenselektor*.

Abbildung 1.18 Auswahlschaltung (Datenselektor)

Der Multiplexer (MUX)
Oft liegen die Auswahlangaben in binär codierter Form vor. Um einen von n Eingängen auszuwählen, braucht man ld n Auswahlsignale. Deren Belegung muß decodiert werden, um die einzelnen UND-Gatter gemäß Abbildung 1.18 anzusteuern. Die Kombination aus Decoder und Datenselektor heißt *Multiplexer*. Aus Geschwindigkeitsgründen ist es sinnvoll, die Decodier- und Auswahlgatter miteinander zu kombinieren (Verringerung der Schaltungstiefe). Solche Schaltungen (Abbildung 1.19) sind als Schaltkreise erhältlich.Typische Multiplexer ermöglichen es, einen von 2, 4, 8 oder 16 Eingängen auszuwählen (man spricht dann vom 2-fach-, 4-fach- usw. bzw. vom 1-aus-2-, 1-aus-4- usw. Multiplexer). Manche Schaltkreise enthalten mehrere gleichartige Multiplexer mit gemeinsamen Adreßeingängen (Abbildungen 1.20, 1.21).

Abbildung 1.19 Multiplexer

Abbildung 1.20 Verschiedene Multiplexerschaltkreise

Erklärung:

a) Multiplexer 1-aus-8 mit Strobeeingang sowie direktem und invertiertem Ausgang,
b) Multiplexer 1-aus-8 mit Steuereingang (Enable) sowie direktem und invertiertem Ausgang. Der direkte Ausgang ist ein Tri-State-Ausgang.
c) zwei Multiplexer 1-aus-4 mit gemeinsamer Auswahladresse und unabhängigen Strobeeingängen. Je ein Ausgang (zweiwertig, direkt).
d) zwei Multiplexer 1-aus-4 mit gemeinsamer Auswahladresse und unabhängigen Steuereingänge. Je ein nicht-invertierter Tri-State-Ausgang.
a) Schaltkreis mit zwei 1-aus-4-Multiplexern (vgl. c) als Schaltsymbol nach DIN 40900.

Auswahlgesichtspunkte:

- die Anzahl der Dateneingänge des einzelnen Multiplexers (typisch sind 16, 8, 4, 2),
- die Anzahl der gemeinsam adressierten Multiplexer im Schaltkreis (typisch sind 1, 2 oder 4),
- die Wirkung der Ausgänge (direkt oder invertierend oder beide),
- die Art der Ausgangsstufen (zweiwertig oder Tri-State).

Das Steuersignal
Typische Bezeichnungen: Strobe, Gate, Enable, Output Control. Dessen Wirkung: nur bei aktivem Steuersignal wird das ausgewählte Eingangssignal zum Ausgang durchgereicht. Ansonsten ist der Ausgang inaktiv. Und hier heißt es aufpassen:

- ein zweiwertiger direkter (nicht-invertierter) Ausgang verharrt auf 0 (Low),
- ein zweiwertiger invertierter Ausgang verharrt auf 1 (High),
- ein Tri-State-Ausgang wird hochohmig (liefert also “nichts”).

Abbildung 1.21 Multiplexer als Schalterbauelement

Erklärung:
Als Beispiel ist hier ein 1-aus-8-Multiplexer mit FET-Schaltern (vgl. Heft 1) dagestellt. Die Besonderheiten:

- die Signallaufzeit zwischen ausgewähltem Eingang und Ausgang ist praktisch vernachlässigbar (z. B. maximal 250 ps),
- der Signalfluß ist in beiden Richtungen möglich. Damit ist das Bauelement auch als Verteiler (Demultiplexer) einsetzbar.
- wird bei aktivem Steuereingang E die Adresse umgeschaltet, so können kurzzeitig mehrere Signale miteinader verbunden werden (wenn infolge der Umschalt-Spikes mehrere FETS gleichzeitig aktiv werden),
- in gewissen Spannungsbereichen eignen sich solche Bauelemente auch als Analogschalter.

Hinweis:
Verschiedene Multiplexerkonfigurationen sind praktisch Industriestandards. Sie werden in verschiedenen Baureihen und mit unterschiedlichen Ausgängen (zweiwertig oder Tri-State, direkt oder invertierend) angeboten.

Multiplexer kaskadieren
Ist eine größere Anzahl von Signalen auszuwählen, so kann man mehrere Multiplexer zu einer Art Pyramiden - oder Baumstruktur zusammenschalten (Abbildung 1.22). Die Alternative: Multiplexer mit Tri-State-Ausgängen werden auf eine gemeinsame Busleitung geschaltet (Abbildung 1.23).

Abbildung 1.22 Multiplexer-Pyramide

Erklärung:
1 - die auszuwählenden Signale; 2 - Multiplexer der ersten Ebene; 3 - Multiplexer der zweiten Ebebe; 4 - Auswahlsignale. Beispiel: es ist eines von 64 Signalen (1) auszuwählen. Eine der möglichen Lösungen verwendet 9 1-aus-8-Multiplexer; 8 in der ersten Ebene und einen in der zweiten. Jeder der Multiplexer der ersten Ebene (2) wählt eines von 8 Signalen (1) aus, der Multiplexer der zweiten Ebene wählt das Ausgangssignals eines der Multiplexer der ersten Ebene aus.

Abbildung 1.23 über Busleitung zusammengeschaltete Multiplexer

Erklärung:
1 - die auszuwählenden Signale; 2 - Multiplexer; 3- Auswahlsignal(e); Multiplexerauswahl (Adreßdecoder). Ein typischer Einsatzfall: die Anschaltung von abzufragenden Signalen an einen Mikrocontroller. Beispiel: ein Mikrocontroller soll 16 Signale (1) abfragen können. Es steht aber nur ein Abfragebus für 4 Signale zur Verfügung. Lösung: zwei Multiplexerschaltkreise mit je vier 1-aus-2-Multiplexern (vgl. Abbildung 1.20d). Die 16 abzufragenden Signale werden in 4 Gruppen zu je 4 Signale aufgeteilt.Jeder Multiplexer wählt eine von zwei Gruppen aus (gesteuert mittels Auswahlsignal 3). Jeweils ein Multiplexer wird auf den Abfragebus aufgeschaltet. Hierzu wird dessen Erlaubniseingang (EN) vom Adreßdecoder 4 angesteuert.

Hinweis:
In solchen Tri-State-Schaltungen muß sichergestellt werden, daß nie zwei Multiplexer gleichzeitig aktiviert werden (auch nicht kurzzeitig). Eine typische Lösung: der Mikrocontroller steuert neben den Auswahl- bzw. Adreßsignalen ein Aufschalterlaubnissignal an (vgl. auch die Erklärung zu Abbildung 1.26). Die Abfrage läuft dann nach folgendem Schema ab: Adresse anlegen - Aufschalterlaubnis ein - Abfragen - Aufschalterlaubnis aus - nächste Adresse anlegen usw. (Prinzip: erst die eine Stufe deaktivieren, dann die andere aktivieren (Break before Make)).

1.4.2. Verteilen/Sperren

Beim Verteilen (Demultiplexing) sind ankommende Signale wahlweise an verschiedene Einrichtungen zu liefern; beim Sperren ist die Weitergabe von Signalen zu verhindern. Beide Funktionen lassen sich gleichermaßen durch konjunktive Verknüpfung lösen (Abbildung 1.24): an der ausgewählten Einrichtung muß eine Eins ankommen, wenn das betreffende Signal den Wert Eins hat UND wenn die betreffende Einrichtung ausgewählt ist. Ansonsten ist eine feste Nullbelegung zu liefern (das ist gleichbedeutend mit dem Sperren des Signalflusses).

Abbildung 1.24 Verteiler- bzw. Sperrschaltung

Der Demultiplexer
Der Demultiplexer ist eine Verteilerschaltung, die über binär codierte Auswahlsignale gesteuert wird. Hierzu werden die einzelnen Auswahlsignale (in Abbildung 1.24: *zu A*, *zu B* usw.) einem Decoder nachgeschaltet. Analog zum Multiplexer können die UND-Gatter gemäß Abbildung

1.24 mit denen des Decoders zusammengefaßt werden. Siehe weiterhin die Abbildungen 1.26 und 1.27 nebst Erklärung.

1.4.3. Decodieren

Decodieren heißt: es gibt mehrere Eingangsleitungen und es sind bestimmte Belegungen dieser Leitungen zu erkennen. Ist eine solche Belegung gegeben, so soll die betreffende Ausgangsleitung aktiv sein. Jede einzelne Bedingung läßt sich durch eine UND-Verknüpfung der betreffenden Eingangssignale erfassen, die gemäß der zu decodierenden Belegung entweder direkt oder negiert der UND-Verknüpfung zugeführt werden. Handelt es sich um n Eingangsleitungen, so können maximal 2^n verschiedene Belegungen decodiert werden. Man hat dann 2^n Ausgangsleitungen, von denen nur jeweils eine aktiv ist (1-aus-n-Decoder). Abbildung 1.25 zeigt einen Decoder für zweistellige Binärzahlen (1-aus-4), Abbildung 1.26 einen für dreistellige (1-aus-8). Decoder für mehrstellige Binärzahlen erfordern entsprechend viele UND-Verknüpfungen. Durch eine mehrstufige, matrixförmige Struktur läßt sich der Aufwand deutlich verringern, wie Abbildung 1.28 am Beispiel eines Decoders für 6-Bit-Binärzahlen zeigt (1-aus-64).

Abbildung 1.25 Decoder für zweistellige Binärzahlen

Abbildung 1.26 Ein Decoder- und Demultiplexerschaltkreis (74x138)

Erklärung:
Der 74x138 ist ein 1-aus-8-Decoder/Demultiplexer mit invertierten Ausgängen und 3 zusätzlichen, konjunktiv verknüpften Steuereingängen (Enable Inputs). Ist die Steuerbedingung (G1 = 1, G2A = 0, G2B = 0) nicht erfüllt, so verharren alle 8 Ausgänge auf 1 (High).

Die Steuereingänge sind vorgesehen, um verschiedene Einsatzfälle zu unterstützen:

- **#** Adreßdecoder in E-A- und Speichersubsystemen (die - der Stückzahl nach - wohl wichtigste Anwendung)
- **#** Demultiplexer (Abbildung 1.27),
- **#** Kaskadierung mehrerer Decoder/Demultiplexer (Abbildung 1.29).

Der 74x138 als Adreßdecoder
Die Steuereingänge der E-A- und Speicherschaltkreise sind typischerweise Low-aktiv, die Steuereingänge des Decoders können meist direkt mit Steuersignalen beispielsweise eines Mikroprozessor-Bussystems zusammengeschaltet werden. Bei nicht erfüllter Steuerbedingung werden alle Ausgänge inaktiv, so daß alle hierüber angesteuerten Tri-State-Stufen ebenfalls inaktiv werden. Somit wird es einfach, das Prinzip Break before Make zu implementieren: nämlich indem die Steuerbedingung erst dann aktiviert wird, nachdem die jeweilige Adresse stabil anliegt und indem sie vor jeder Änderung der Adresse deaktivert wird. (Die Signalspiele der typischen Mikroprozessor-Bussysteme sind so organisiert, daß diese Forderung eingehalten wird. Der 74x138 kann somit zumeist “nach Kochbuch” eingesetzt werden).

Abbildung 1.27 Der 74138 als Demultiplexer

Erklärung:
Da der Schaltkreis invertierende Ausgänge hat, wird das zu verteilende Signal an einen negierten Steuereingang angeschlossen. Der über die Adreßeingänge ausgewählte Ausgang schaltet so wie das zu verteilende Signal; die anderen Ausgänge verharren auf High.

Abbildung 1.28 6-Bit-Decoder (1-aus-64) in Matrixstruktur

Erklärung:
Ein 1-aus-64-Decoder würde 64 6-fach-UNDs erfordern. Hier werden die zu decodierenden 6 Bits in zwei Abschnitte (Spalte und Zeile) zu je 3 Bits zerlegt. Beide Abschnitte werden unabhängig voneinander decodiert (1-aus-8; das erfordert jeweils 8 3-fach-UNDs). Die Zeilen- und Spaltensignale werden mit insgesamt 64 2-fach-UNDs zu den eigentlichen Ausgangssignalen verknüpft.

Abbildung 1.29 6-Bit-Decoder/Demultiplexer (1-aus-64), aufgebaut mit kaskadierten 74x138s

Erklärung:
Es handelt sich um eine baum- bzw. pyramidenförige Struktur aus 9 Schaltkreisen, die in zwei Ebenen angeordnet sind. 1 - erste Ebene (ein Schaltkreis); 2 - zweite Ebene (acht Schaltkreise); 3 - Adreßeingänge; 4 - Steuereingänge; 5 - zu verteilendes Signal. Jeder Schaltkreis der zweiten Ebene (2) erregt 8 der insgesamt 64 Ausgangssignale. Der Schaltkreis der ersten Ebene bestimmt, welcher der Schaltkreise der zweiten Ebene aktiviert wird. Die Steuersignale (4) werden am Schaltkreis der ersten Ebene wirksam (ein inaktiver Schaltkreis (1) kann auch keinen der Schaltkreise (2) aktivieren). Wird die Anordnung als Decoder betrieben, so sind die beiden verbleibenden Steuereingänge (G1, G2A) der Schaltkreise der zweiten Ebene fest beschaltet. Bei Nutzung als Demultiplexer wird das zu verteilende Signal (5) an die negierten Steuereingänge (G2A) der zweiten Ebene geführt (die Anordnung wirkt dann so, wie in Abbildung 1.27 erklärt).

1.4.4. Codieren

Codieren heißt, einen bestimmten Code auf Ausgangsleitungen bilden, wenn jeweils eine von n Eingangsleitungen aktiv ist (also, einen 1-aus-n-Code in einen beliebigen anderen Code zu wandeln). Dazu wird für jede Ausgangsleitung eine ODER-Verknüpfung all jener Eingangsleitungen vorgesehen, bei deren Aktivierung die betreffende Ausgangsleitung auch aktiv werden muß. Abbildung 1.30 zeigt das Prinzip am Beispiel eines Codieres (Encoders) für binär codierte Dezimalzahlen.

Abbildung 1.30 Codierer (Encoder) für binär codierte Dezimalzahlen

Erklärung:
Aus der Codetabelle ergibt sich die Schaltung auf nahliegende Weise. Beispiel: das Signal 2^1 ist zu aktivieren, wenn eingangsseitg eine der Dezimalzahlen ZWEI oder DREI oder SECHS oder SIEBEN anliegt.

1.4.5. Umcodieren (Codes wandeln)

Beliebige Codewandlungen (von einem Code A in einen Code B) lassen sich durch Hintereinanderschalten eines Decoders und eines Encoders verwirklichen (Abbildung 1.31). Gelegentlich sind in einer solchen Anordnung schaltalgebraische Vereinfachungen möglich. Heutzutage werden für Codewandlungen oft Speicherschaltkreise (z. B. PROMs) eingesetzt.

Abbildung 1.31 Codewandlung (Prinzip)

1.4.6. Kombinatorische Operationswerke

Kombinatorische Operationswerke führen in Prozessoren und Spezialschaltungen Transport- und Verknüpfungsoperationen aus (Abbildung 1.32). Die Eingangsbelegungen sind Operanden und Operationscodes, wobei der Operationscode die jeweils auszuführende Operandenverknüpfung auswählt. Ausgangsseitig werden die Ergebnisse der Operandenverknüpfungen geliefert. Diese werden zumeist durch zusätzliche Bedingungssignale (Flagbits) ergänzt.

Abbildung 1.32 Das kombinatorische Operationswerk: die allgemeine Struktur

Die ALU
ALU = Arithmetic/Logic Unit. Dies ist die gängige Bezeichnung des Operationswerks typischer Unvversalrechner (Prozessoren). Ein solches Operationswerk kann elementare logische Verknüpfungen sowie Rechenoperationen mit Binärzahlen ausführen (hinzu kommen Einzelbitoperationen, Verschiebeoperationen usw.). In den folgenden beiden Kapiteln wollen wir die Auslegung solcher Operationswerke näher betrachten.

2. Nichtarithmetische Operatoren

Im folgenden wollen wir uns Schaltungsanordnungen für elementare nichtarithmetische Datenverknüpfungen ansehen. Solche Schaltungen sind beispielsweise in Prozessoren erforderlich, um die “logischen” Operationsbefehle auszuführen.

2.1. Elementare bitweise Verknüpfungen

Die Logikeinheit eines Prozessors führt üblicherweise wenigstens folgende Operationen aus: unmodifizierte Weiterleitung eines Operanden, Negation eines Operanden, sowie Konjunktion, Disjunktion und Antivalenz als Verknüpfung zweier Operanden (die entsprechenden Befehle werden beispielsweise mit MOV, NOT, AND, OR, XOR bezeichnet). Dazu muß man lediglich für die einzelnen Funktionen bitweise entsprechende Gatter anordnen, diese ausgangsseitig disjunktiv zusammenfassen und jeweils eine konjunktive Verknüpfung mit Steuersignalen vorsehen, die aus dem Operationscode des Befehls decodiert werden, also die jeweilige Befehlswirkung im 1-aus-n-Code kennzeichnen (Abbildung 2.1).

Abbildung 2.1 Universelle Logikeinheit

Funktionserklärung an Beispielen:

1. Operand A zum Ausgang transportieren (GATE A): Steuersignal GATE A aktiviert das UND-Gatter 1, so daß das Operandenbit a zum Ausgang (Resultat) durchgesteuert wird,
2. Operanden A und B disjunktiv verknüpfen (OR A, B): die Verknüpfung wird mittels ODER-Gatter 2 gebildet. Steuersignal OR aktiviert das UND-Gatter 3, so daß das Verknüpfungsergebnis zum Ausgang (Resultat) durchgesteuert wird.

Zur Aufwandsverringerung verwendet man oft eine "aufgelöste" Bauweise. Das heißt, es ist in jeder Bitposition eine Gatter-Anordnung vorgesehen, die - als Verbund - durch entsprechende Beschaltung mit Steuersignalen alle gewünschten Wirkungen hervorbringt (die Abbildungen 2.2, 2.3 zeigen ein Beispiel). Das Prinzip wird in Arithmetik-Logik-Einheiten (Arithmetic/Logic Units, ALUs) verwendet, die als Schaltkreise verfügbar sind (Beispiel-Typ: ‘181). Dort sind die Verknüpfungsschaltungen so erweitert, daß sie auch arithmetische Funktionen (Addition/Subtraktion) ausführen können.

Abbildung 2.2 Universelle Logikeinheit in "aufgelöster" Bauweise

Abbildung 2.3 Zur Funktionsweise

Funktionserklärung an Beispielen:

a) Operand A zum Ausgang transportieren (GATE A): A kann beispielsweise über UND-Gatter 1 zum Ausgang durchgeschaltet werden. Hierzu wird sowohl C_0 als auch C_1 aktiviert. Damit ist - unabhängig von der Belegung des Operanden B - Signal 2 stets = 1.

b) Operanden A und B disjunktiv verknüpfen (OR A, B): die Verknüpfung wird durch das ausgangsseitige ODER-Gatter 3 gebildet. Damit die Operandenbits a, b zum ODER-Gatter 3 gelangen, müssen die UND-Gatter 1, 4 aktiviert werden. Hierzu werden alle 4 Steuersignale C_0... C_3 gesetzt.

Übungsaufgabe:
Weshalb wird die OR-Verknüpfung ($a \vee b$) auch dann ausgeführt, wenn entweder C_0 oder C_2 = 0 ist?

Flagbits
Bei logischen Operationen gibt es eine Bedingung, die anwendungspraktisch besonders wichtig ist und deshalb als Flagbit oder Bedingungscode bereitgestellt werden muß: sie besagt, ob das Ergebnis nur Nullen enthält oder nicht (Zero Flag). Das ist durch eine NOR-Verknüpfung aller Ergebnisleitungen erkennbar (wie aus Abbildung 2.1 ersichtlich).

In manchen Prozessoren werden darüber hinaus noch andere Bedingungen erkannt. So werden in der x86-Architektur bei logischen Verknüpfungen noch die Flagbits SF und PF gestellt. SF entspricht dem höchstwertigen Ergebnisbit. PF wird gesetzt, wenn die Anzahl der Ergebnisbits ungerade ist (Paritätsgeneratorschaltung; Grundlage: Anitvalenzgatter).

2.2. Modifizieren, Testen, Vergleichen

In diesem Abschnitt wollen wir keine weiteren Schaltungen erklären, sondern zeigen, wie man mit den elementaren logischen Verknüpfungen wichtige Anwendungsaufgaben lösen kann (Abbildung 2.4).

Abbildung 2.4 Elementare Bitoperationen anhand von Beispielen

Setzen von Bits
Um bestimmte Bits (in einem Register, Maschinenwort usw.) zu setzen, ist ein OR-Befehl notwendig, wobei der zweite Operand an allen zu setzenden Stellen Einsen enthält und sonst Nullen.

Löschen von Bits
Um bestimmte Bits (in einem Register, Maschinenwort usw.) zu löschen, ist ein AND-Befehl notwendig, wobei der zweite Operand an allen zu löschenden Stellen Nullen enthält und sonst Einsen.

Wechseln von Bits
Um bestimmte Bits (in einem Register, Maschinenwort usw.) in ihrem Wert zu ändern (von 0 nach 1 und umgekehrt), ist ein XOR-Befehl notwendig, wobei der zweite Operand an allen zu ändernden Stellen Einsen enthält und sonst Nullen.

Entnehmen von Bits
Um zusammenhängende Bitfelder oder bestimmte einzelne Bits aus einem Register, Maschinenwort usw. zu entnehmen (andere Redeweisen: ausblenden, maskieren), ist ein AND-Befehl notwendig, wobei der zweite Operand an allen ausgewählten Stellen Einsen enthält und sonst Nullen. (Dieser Operand heißt üblicherweise Maskenoperand.)

Einfügen von Bits
Um zusammenhängende Bitfelder oder bestimmte einzelne Bits in ein Register, Maschinenwort usw. einzufügen, ist auf das Register, Maschinenwort usw. zunächst ein AND-Befehl anzuwenden, dessen zweiter Operand an allen betreffenden Stellen Nullen enthält und sonst Einsen

(Maskenoperand). Nachfolgend ist ein OR-Befehl anzuwenden, dessen zweiter Operand an den betreffenden Stellen die einzufügenden Werte enthält und sonst Nullen.

Testen auf gelöschte bzw. gesetzte Bits
Um zu prüfen, ob bestimmte Bits (in einem Register, Maschinenwort usw.) alle gelöscht sind oder ob wenigstens eines dieser Bits gesetzt ist, braucht man einen AND-Befehl, dessen zweiter Operand an allen betreffenden Stellen Einsen enthält und sonst Nullen. Sind alle so geprüften Bits gelöscht, ist das Zero-Flagbit gesetzt. Ist wenigstens eines der geprüften Bits gesetzt, ist das Flagbit gelöscht. Diese Funktion steht beispielsweise in der x86-Architektur als TEST-Befehl zur Verfügung (das ist ein AND-Befehl, der kein Ergebnis zurückschreibt).

Vergleichen
Um zu prüfen, ob zwei Bitmuster einander gleich sind oder nicht, kann man einen XOR-Befehl verwenden. Bei Gleichheit ist das Zero-Flag gesetzt, bei Ungleichheit gelöscht. XOR-Befehle ohne Zurückschreiben des Ergebnisses gibt es in manchen Architekturen als "logische" Vergleichsbefehle (Compare Logical). Will man nur ausgewählte Bits miteinander vergleichen, so müssen die anderen Bits ausgeblendet werden, und zwar entweder durch OR mit Einsen oder durch AND mit Nullen (man muß nur beide Operanden auf gleiche Weise maskieren).

Die "logische" Vergleichsfunktion ist nichts anderes als eine disjunktive Zusammenfassung von Antivalenzverknüpfungen der zu vergleichenden Bits. Dafür gibt es auch besondere Schaltkreise (Identity Comparators, z. B. die Typen '518 bis '521 und '688). Sie sind beispielsweise als Adreßvergleicher einsetzbar.

Maskierte Logikoperationen
Wir haben gesehen, daß vielfältige nützliche Operationen eine "Maskierung" erfordern. Damit sollen nur bestimmte Bits bei der Verarbeitung berücksichtigt, die anderen hingegen davon ausgeschlossen werden. Diese Maskierung erfordert oft zusätzliche Befehle, z. B. ein AND mit dem Maskenoperanden, um in den auszuschließenden Bitpositionen Nullen zu erzwingen. Es gibt Anwendungsfälle, wo solche Operationen leistungsbestimmend sind. Deshalb hat man maskierte Verknüpfungen gelegentlich als Maschinenbefehle vorgesehen. Diese Befehle haben einen zusätzlichen (dritten) Masken-Operanden, und in der Verknüpfungshardware sind entsprechend zusätzliche UND-Verknüpfungen in den Datenweg eingefügt (Drei-Operanden-Logikeinheit).

2.3. Nullerweiterung

Hat ein Operand weniger Bits als die Verarbeitungsbreite der Hardware, so müssen die ungenutzten Bitpositionen mit Nullen belegt werden (Nullerweiterung; Zero Extend). Dies läßt sich durch konjunktive Verknüpfungen mit entsprechenden Steuersignalen erreichen (Abbildung 2.5).

Abbildung 2.5 Nullerweiterung (Zero Extend)

2.4. Verschieben und Rotieren

In den meisten Prozessoren sind "logische" Verschiebe- und Rotationsbefehle vorgesehen. Auch diese Operationen werden häufig mit kombinatorischen Schaltungen ausgeführt. Was bedeutet eigentlich "Verschieben"? - Beim *Rechtsverschieben* um ein Bit gelangt Bit 1 nach Bit 0, Bit 2 nach Bit 1 usw. Entsprechend gelangt beim *Linksverschieben* um ein Bit Bit 6 nach Bit 7, Bit 5 nach Bit 6 usw. Sinngemäß werden die Daten bewegt, wenn eine Verschiebung um n Bits auszuführen ist (so kommt beim Linksverschieben um zwei Bits Bit 5 nach Bit 7, Bit 4 nach Bit 6 usw.).

Verschieben und Rotieren
Betrachten wir das "Innere" einer Datenstruktur, so bereitet es uns keine Schwierigkeiten, die Bewegung der Bits beim Verschieben zu verstehen. Was geschieht aber links und rechts, also gewissermaßen an den "Rändern"? - Dies ist der Punkt, in dem sich Verschiebe- und Rotationsbefehle unterscheiden (Abbildung 2.6).

Abbildung 2.6 Verschieben und Rotieren

Beim *Rotieren* werden die "hinausgeschobenen" Bits am jeweils anderen Ende wieder zurückgeführt (Wrap Around). Rotieren ist also ein zyklisches Verschieben in den Grenzen der jeweiligen Operandenlänge. Wird ein Byte um ein Bit nach links rotiert, gelangt Bit 7 nach Bit 0, bei Rechtsrotation entsprechend Bit 0 nach Bit 7.

Hingegen gehen beim eigentlichen *Verschieben* die "hinausgeschobenen" Bits verloren. Die Bitpositionen, die "am anderen Ende" frei werden, werden meist mit Nullen aufgefüllt. In manchen Architekturen (beispielsweise ab 386) gibt es zusätzlich Verschiebebefehle mit Erweiterung. Diese haben einen zweiten Operanden, aus dem die besagten freien Bitpositionen aufgefüllt werden.

In vielen Architekturen werden bestimmte Flagbits in Verschiebe- bzw. Rotationsabläufe einbezogen. Zumeist ist das jeweils zuletzt hinausgeschobene oder umlaufende Bit in einem Flagbit direkt abfragbar (in der x86-Architektur: CF). Ist das Flagbit beim Rotieren beteiligt, so laufen die herausgeschobenen Bits nicht unmittelbar am anderen Ende des Operanden wieder ein, sondern gelangen zunächst in das Flagbit und von dort aus in das eigentliche Ergebnis zurück (in der x86-Architektur gibt es solche Befehle zum Rotieren über CF).

Wie lassen sich solche Abläufe schaltungstechnisch verwirklichen? - Es ist dazu nur notwendig, für jede Ergebnis-Bitposition eine Auswahlschaltung vorzusehen, an die alle Operanden-Bits angeschlossen sind, die irgendwie als Ergebnis wirksam werden können. Abbildung 2.7 zeigt das allgemeine Prinzip: Multiplexer sind mit kombinatorischen Steuerschaltungen verbunden, die aus Befehlscode und der Zahl der Bitpositionen, über die verschoben/rotiert werden soll (COUNT-Angabe) die jeweilige Auswahl-Adresse ermitteln.

Abbildung 2.7 Universelle Schaltung zum Verschieben/Rotieren (Prinzip)

Wenn man die Auswahlschaltungen jeweils in passender Reihenfolge mit den einzelnen Operandenbits beschaltet, sind besondere kombinatorische Schaltungen zur Bildung der Auswahlsignale gar nicht notwendig.

Der Barrel Shifter

Das ist die übliche Bezeichnung für eine universelle Verschiebeeinheit, mit der man Daten um mehr als ein Bit verschieben bzw. rotieren kann. Wir wollen uns die Funktionsweise zunächst an einer 8-Bit-Ausführung klarmachen. Abbildung 2.8 zeigt die Schaltung. Im Sinne der Universalität (beliebige Verschiebungen um 0..7 Bits) sind 8 Multiplexer mit 8 Eingängen vorgesehen. Wir wollen sie so beschalten, daß besondere Kombinatorik zur Bildung der Auswahladressen unnötig ist. Alle Adreßeingänge sind gleichermaßen an einen 3-Bit-Datenweg angeschlossen, der die Anzahl der Bits liefert, um die verschoben werden soll (SHIFT COUNT). Ein SHIFT COUNT von Null heißt: Operand nicht verschieben, sondern einfach durchreichen.

Abbildung 2.9 verdeutlicht den Anschluß der Operandenbits an die Multiplexer. Die Schaltung löst zunächst die Aufgabe des Rotierens. Ein SHIFT COUNT a bedeutet Rechtsrotieren um a Bits bzw. ein Linksrotieren um 8-a Bits (mit anderen Worten: zum Linksrotieren ist der SHIFT COUNT im Zweierkomplement anzugeben).

Anhand der Abbildung 2.9 können wir uns leicht überzeugen, daß wegen des "Wrap Around" ein Rechtsrotieren um a und ein Linksrotieren um 8-a Bits tatsächlich auf das gleiche Ergebnis führt.

Abbildung 2.8 8-Bit-Verschiebeanordnung (Barrel Shifter)

Abbildung 2.9 Anschluß der Operandenbits an die Multiplexer-Eingänge beim 8-Bit-Barrel Shifter

Das eigentliche Verschieben (SHIFT) bereitet nun keine Probleme mehr - es genügt, die freiwerdenden Positionen mit Nullen zu belegen. Verschieben ist also ein Rotieren mit Ausblenden der freiwerdenden Bits. Zum Ausblenden können wir den Multiplexern UND-Gatter nachschalten. Diese sind folgendermaßen mit einem Maskenvektor anzusteuern: bei Linksverschiebung um ein Bit mit 1111 1110B, um zwei Bits mit 1111 1100B usw.; entsprechend bei Rechtsverschiebung mit 0111 1111B, 0011 1111B usw. Beim Rotieren muß der gesamte Maskenvektor Einsen enthalten. Gehen wir von der Linksverschiebung aus. SHIFT COUNT 0 entspricht 1111 1111B, SHIFT COUNT 1 entspricht 1111 1110B usw. Wir müssen somit nichts weiter tun, als den SHIFT COUNT zu decodieren und die im 1-aus-8-Code gelieferte Eins zu allen jeweils höheren Bitpositionen durchzureichen. Für die Rechtsverschiebung müssen wir den so gebildeten Maskenvektor invertieren. Abbildung 2.10 zeigt die entsprechende Schaltung.

Abbildung 2.10 Universelle 8-Bit-Verschiebe- und Rotationseinrichtung

Für Verschiebeoperationen mit Erweiterung sind dem Barrel Shifter weitere Auswahlschaltungen nachzusetzen, um die jeweiligen Bits des zweiten Operanden zu den Resultatbitstellen durchzuschalten. Da Sie das Prinzip zweifellos verstanden haben, verzichten wir auf eine genauere Darstellung.

Solche Barrel Shifter sind Bestandteil praktisch aller Hochleistungsprozessoren. Damit können Verschiebe- und Rotationsabläufe über die volle Verarbeitungsbreite der Hardware in einem einzigen Maschinenzyklus ausgeführt werden. Das ist nicht nur für die eigentlichen Verschiebe- und Rotations*befehle* von Bedeutung, sondern auch für alle anderen internen Abläufe, bei denen solche Vorgänge benötigt werden (dies gilt beispielsweise für die Multiplikation und Division).

Das Crossbar-Netzwerk als Barrel Shifter
In der Praxis wird der Barrel-Shifter meist nicht mit Multiplexern aufgebaut. Vielmehr wird die steuerbare Verbindung "Jeder mit Jedem" durch eine Matrixanordnung von Koppelstufen (Transfer-Gates) verwirklicht (Abbildung 2.11). Der Name "Crossbar" stammt aus der Fernsprechtechnik, wo solche Strukturen erstmals entwickelt wurden, und zwar seinerzeit noch auf elektromechanischer Grundlage (Kreuzschienenverteiler).

Abbildung 2.11 Crossbar-Netzwerk als technische Verwirklichung eines Barrel Shifters

Anwendungsbeispiel: Byte- und Halbwortauswahl
Das Szenarium: es sind Daten zu speichern oder zur Verarbeitung aufzubereiten. Die Daten können an nicht-integralen Adressen gespeichert sein. Der Speicherdatenweg ist breiter als 1 Byte.

Beispielsweise ergeben sich bei einer Zugriffsbreite von 32 Bits und Byteadressierung die in Abbildung 2.12 gezeigten Möglichkeiten, Bytes, 16-Bit-Worte (Halbworte) und 32-Bit-Worte zu speichern. All diese Datenstrukturen müssen aber rechtsbündig zu den Verarbeitungsschaltungen geliefert werden. Andererseits stehen am Ausgang des Operationswerkes die zu speichernden Daten in rechtsbündiger Anordnung bereit und sind zwecks Speicherung entsprechend aufzubereiten.

Abbildung 2.12 Anordnungs-Beispiele für Datenstrukturen

Erklärung:
In Abbildung 2.12 hat ein Wort 4 Bytes und ein Halbwort 2. Die Werte 0...3 sind Byteadressen. Wir beschränken uns hier auf die Anordnung von Worten, Halbworten und Bytes unter jeweils einer Speicheradresse in einem Speicher mit 32 Bits Zugriffsbreite. Das Überschreiten der Zugriffsbreite erfordert jeweils zwei Zugriffe (zu aufeinanderfolgenden Speicheradressen), um die gewünschte Datenstruktur zusammenzusetzen oder abzuspeichern. Das Prinzip (Verschiebung) ist aber das gleiche.

2.5. Markierungsvektoren, Prioritätscodierung, Indexwerte

Diese Begriffe wollen wir verwenden, um Schaltmittel zur Lösung folgender Aufgaben zu betrachten:

- aus einem Operanden-Binärvektor ist ein Binärvektor gleicher Länge zu erzeugen, der nur eine einzige Eins enthält, und zwar - wählbar - jene, die der niedrigstwertigen oder der höchstwertigen Eins im Operanden entspricht (Markierungsvektor),
- die Position dieser Eins ist als Binärzahl zu bestimmen (Indexwert oder Bitadresse),
- aus einem Indexwert (einer Bitadresse) ist ein Binärvektor mit einer Eins an der entsprechenden Position zu bilden.

Solche Funktionen sind in einigen modernen Architekturen vorgesehen. Die Bezeichnungsweise ist allerdings nicht einheitlich.

Markierungsvektor
Wir wollen zunächst einen Markierungsvektor für die niedrigstwertige Eins bilden. Dazu müssen wir nichts weiter tun, als in jeder Bitposition ein UND-Gatter vorzusehen, das das "eigene" Operandenbit nur dann durchläßt, wenn alle vorhergehenden (jene mit niedrigerem Bit-Index) mit Null belegt sind. Sinngemäß erfordert ein Markierungsvektor für die höchstwertige Eins UND-Gatter, die das jeweilige "eigene" Operandenbit nur dann weitergeben, wenn alle nachfolgenden Operandenbits (mit höherem Bit-Index) Nullen sind. Dabei können alle jeweils in Frage kommenden Operandenbits parallel verknüpft werden, es ist aber auch "Weiterschleifen" von Bitposition zu Bitposition (Kaskadierung) möglich (Abbildung 2.13). Zudem leigt es nahe, an Kompromißlösungen zu denken (abschnittsweise Parallelverknüpfung (z. B. über jeweils 8 Bits) und Kaskadierung von Abschnitt zu Abschnitt).

Solche Netzwerke heißen auch *Prioritätscodierer (Priority Encoder)*, da sie einen Ergebnisvektor bilden, in dem nur die Bitposition mit der niedrigsten bzw. höchsten "Priorität" aktiv ist (eine offensichtliche Anwendung besteht darin, aus verschiedenen gleichzeitig anhängigen Anforderungen diejenige auszuwählen, welche zuerst - mit höchster Priorität - bearbeitet werden soll). Sie sind als Schaltkreise verfügbar. Als Beispiel mag der Typ '149 dienen, der aus 8 Eingangs-Bits einen 8-Bit-Markierungsvektor (im 1-aus-n-Code) gemäß der höchsten Priorität bildet (Abbildung 2.14).

Hinweis:
Für die niedrigstwertige Eins wird gelegentlich die Bezeichnung FIRST OCCURRENCE verwendet, entsprechend für die höchstwertige Eins die Bezeichnung LAST OCCURRENCE.

Abbildung 2.13 Bildung von Markierungsvektoren (Prioritätscodierung)

Abbildung 2.14 Der Prioritätscodierer (Priority Encoder) ‘149

Erklärung zu Abbildung 2.14:
Sowohl die Eingangs- als auch die Ausgangssignale sind aktiv Low. Bitposition 7 hat höchste, Bitposition 0 niedrigste Priorität. RI7...0 - Prioritätseingänge; RO7...0 - Prioritätsausgänge; RQE - Steuereingang (Request Enable); RQP - Anforderungsausgang (Request Output). Ist RQE inaktiv (High), sind alle Ausgänge inaktiv. Bei aktivem RQE (Low) führt ein aktiver Eingang RI7 zur Aktivierung des Ausgangs RO7 und verhindert, daß alle nachfolgenden Ausgänge RO6...0 aktiv werden. Ist RI7 inaktiv und RI6 aktiv, so wird RO6 aktiv usw. (die Schaltung entspricht Abbildung 2.13b). RQP ist inaktiv (High), solange alle Prioritätseingänge inaktiv (High) sind. Sobald wenigstens ein Prioritätseingang aktiv (Low) ist, wird auch RQP aktiv (Low).

Kaskadierung
Mehrere ‘149s können gleichsam in Reihe geschaltet werden. Da das höchstwertige aktive Signal durchgeschaltet werden soll, sind die für niederwertige Bitpositionen zuständigen Schaltkreise zu deaktivieren, sobald ein für höherwertige Bitpositionen zuständiger Schaltkreis ein aktives Eingangssignal erkannt hat (Abbildung 2.15).

Abbildung 2.15 Kaskadierung von Prioritätscodierern ‘149

Erklärung:
1 - Schaltkreis für die höherwertigen Bitpositionen; 2 - Schaltkreis für die niederwertigen Bitpositionen. Schaltkreis 2 darf nur dann aktiv werden, wenn an Schaltkreis 1 kein einziger Prioritätseingang ist. Dieser Zustand (alle Prioritätseingänge auf High) wird durch RQP = High signalisiert. Folglich ist RQP des Schaltkreises 1 unter Zwischenschaltung eines Negators mit RQE des Schaltkreises 2 zu verbinden. Um ein Gesamt-Anforderungssignal (REQUEST) zu bilden (welches anzeigt, daß wenigstens eines der 16 Eingangssignale aktiv ist), sind alle RQP-Ausgänge disjunktiv zu verknüpfen.

Indexwert aus Markierungsvektor
Hat man einmal einen Markierungsvektor, so ist nur eine gewöhnliche Codierschaltung (aus ODER-Gattern) notwendig, um aus dem 1-aus-n-Code die Bitadresse bzw. den Index des jeweils gesetzten Bits zu bestimmen. Es gibt Priority-Encoder-Schaltkreise, die auch die Codierfunktion enthalten, z. B. die Typen '147 und '148 (Abbildung 2.16).

Abbildung 2.16 Prioritätscodierer, die codierte Indexwerte liefern

Erklärung:

a) Prioritätscodierer ‘147 mit 9 Prioritätseingängen und 4 Prioritätscode-Ausgängen. Alle Signale sind aktiv Low. Eingang 9 hat höchste, Eingang 1 niedrigste Priorität. Ist Eingang 9 aktiv, so wird an den Ausgängen eine binär codierte 9 abgegeben (und zwar invertiert, also als 0110), ist Eingang 9 inaktiv und Eingang 8 aktiv, so liegt an den Ausgängen eine binär codierte 8 (invertiert = 0111) usw. Ist keiner der Prioritätseingänge aktiv, so erscheint an den Ausgängen eine binär codierte 0 (invertiert = 1111). Somit werden insgesamt 10 verschiedene Prioritätscodes ausgegeben (10-to-4 Priority Encoder).

b) Prioritätscodierer ‘148 mit 8 Prioritätseingängen, 3 Prioritätscode-Ausgängen (8-to-3) sowie weitere Anschlüsse zu Kaskadierungszwecken (Abbildung 2.17). Alle Signale sind aktiv Low. GS - kennzeichnet, daß wenigstens einer der Prioritätseingänge aktiv (Low) ist (Group Signal). EI - Kaskadierungs-Eingang: der Schaltkreis wird nur dann aktiv, wenn EI aktiv (Low) ist. EO - Kaskadierungs-Ausgang: ist nur dann aktiv (Low), wenn alle Prioritätseingänge inaktiv (High) sind. Eingang 7 hat höchste, Eingang 0 niedrigste Priorität. Ist Eingang 7 aktiv, so wird an den Ausgängen eine binär codierte 7 abgegeben (und zwar invertiert, also als 111), ist Eingang 7 inaktiv und Eingang 6 aktiv, so liegt an den Ausgängen eine binär codierte 6 (invertiert = 001) usw. Die Belegungen “Eingang 0 aktiv” und “gar kein Prioritätseingang aktiv” werden gleichermaßen als invertierte 0 (111) wiedergegeben. Zur Unterscheidung zwischen beiden Belegungen kann man die Signale GS bzw. EO heranzuziehen.

Hinweise:

1. Die Gatternetzwerke lassen sich unmittelbar aus den Wahrheitstabellen (Function Tables) ableiten.
2. Beim Entwerfen für programmierbare Logik oder für ASICs: die Schaltungen nicht sklavisch übernehmen, sondern gemäß den jeweiligen Erfordernissen abwandeln bzw. von Grund auf systematisch entwickeln. Varianten ausprobieren (womöglich ist ein Prioritätscodierer für den 1-aus-n-Code mit nachgeschaltetem Codiernetzwerk besser als eine Kaskadierung mehrerer ‘148s...).

Abbildung 2.17 Kaskadierung von Prioritätscodieren ‘148

Erklärung:
1 - Schaltkreis für die höherwertigen Bitpositionen; 2 - Schaltkreis für die niederwertigen Bitpositionen; 3 - disjunktiver Verknüpfung der drei niederwertigen binären Ausgänge (invertiert); 4 - Gesamt-Anforderungssignal (REQUEST): zeigt an, daß wenigstens eines der 16 Eingangssignale aktiv ist. Das Gruppen-Aktivitätssignal GS des Schaltkreises 1 liefert die höchstwertige (4.) Bitposition des Prioritätscodes. Der gesamte Prioritätscode wird invertiert geliefert.

Abwandlung für nicht invertierten Prioritätscode: (1) Gatter 3: NANDs statt ANDs; (2) höchstwertige Stelle A3 an E0 des Schaltkreises 1 statt an GS anschließen.

Markierungsvektor aus Indexwert
Die Wandlung einer als Binärzahl codierten Bitadresse in einen Markierungsvektor erfordert lediglich einen 1-aus-n-Decoder (für eine 32-Bit- Maschine würde beispielsweise "die Hälfte" des Decoders von Abbildung 1.28 genügen).

Solche Markierungsvektoren erlauben es, mit einer üblichen universellen Logikeinheit, wie wir sie kennengelernt haben, beliebige Einzelbitoperationen auszuführen (Bits setzen, löschen, wechseln, testen usw.).

2.6. Operationen mit variablen Bitfeldern

Der höchste Grad an Flexibilität ist erreicht, wenn man nicht nur Bytes, Worte usw. verarbeiten kann, sondern wenn es möglich ist, beliebige Bitfelder zu transportieren und miteinander zu verknüpfen. Abbildung 2.18 veranschaulicht solche Bitfelder und die beiden Elementaroperationen, die wir hier als Entnehmen (EXTRACT) und Einfügen (DEPOSIT) bezeichnen wollen.

Abbildung 2.18 Elementare Operationen mit variablen Bitfeldern

Welche Angaben bestimmen ein Bitfeld?
Ein Bitfeld ist eindeutig bestimmt (1) durch die Bitadresse (den Index) des ersten Bits und (2) durch die Länge (Anzahl der Bits) oder - alternativ - durch den Index des letzten Bits.

Entnehmen (EXTRACT)
Das Bitfeld wird gemäß seinen bestimmenden Angaben aus dem Operanden entnommen und rechtsbündig im Rahmen der Verarbeitungsbreite bereitgestellt. Die verbleibenden Bits werden mit Nullen belegt.

Einfügen (DEPOSIT)
Ein rechtsbündig bereitstehendes Bitfeld wird gemäß seinen bestimmenden Angaben in den Operanden eingefügt. Die verbleibenden Bits des Operanden werden nicht verändert.

Auf den ersten Blick erscheint es spitzfindig, solche Operationen in der Hardware vorzusehen. Die gleiche Wirkung kann man schließlich durch logische Verknüpfungs- und durch Verschiebebefehle erreichen. Es hat sich aber gezeigt, daß derartige Abläufe vergleichsweise häufig benötigt werden. Deshalb wurden sie auch in RISC-Architekturen vorgesehen. Beispiele: (1) Precision Architecture (PA) von Hewlett-Packard, (2) IA-64 (Intel).

Hinweis:
Wir verwenden hier die Begriffe aus den einschlägigen Architekturbeschreibungen (andere Anbieter verwenden gelegentlich andere Bezeichnungen).

Ein Grund für die besondere Bedeutung liegt in dem Bestreben nach guter Speicherausnutzung. Moderne Programmiersprachen erlauben es, Verbundstrukturen (records) zu definieren, die Einzelbits, Binärzahlen, Dezimalzahlen usw. "bunt durcheinander" enthalten. Werden diese Angaben dicht gepackt, so muß man sie vor der eigentlichen Verarbeitung vereinzeln und die Ergebnisse dann wieder in die jeweiligen Speicherpositionen hineinbringen, ohne den Rest der Datenstruktur dabei zu beeinflussen. Als Anwendungsprogrammierer merken Sie davon nichts. Die erforderlichen Informationswandlungen werden vielmehr im Verbund von Compiler und Laufzeitsystem gewährleistet. Fehlen besondere Bitfeldbefehle, so werden die weitaus meisten der logischen Verknüpfungs- und Verschiebebefehle, die während der Abarbeitung von Anwendungsprogrammen ausgeführt werden, nur für solche Zwecke verwendet (also für reine Hilfsdienste, die mit der eigentlichen Verarbeitung gar nichts zu tun haben).

Hardware
Man braucht einen Barrel Shifter und ein nachgeordnetes Maskierungsnetzwerk (Abbildung 2.19). Der Maskenvektor wird je nach auszuführender Operation gebildet (Abbildung 2.20). Diese Hardware liefert ein Ergebnis, das die Bits des Bitfeldes in der jeweils richtigen Position und sonst Nullen enthält. Es kann somit auf einfache Weise mit üblichen logischen Operationen (AND, OR usw.) weiterverarbeitet werden.

Abbildung 2.19 Hardware für Bitfeldoperationen (EXTRACT, DEPOSIT)

Abbildung 2.20 Bildung der Maskenvektoren für EXTRACT und DEPOSIT

Voraussetzungen für die weitere Erklärung:

1. das Bitfeld ist in einem Wort der Länge n angeordnet,
2. das niedrigstwertige Bit des Bitfeldes hat den Index a, das höchstwertige den Index b,
3. die Länge des Bitfeldes bezeichnen wir mit l; l = b - a + 1. In der Hardware verwenden wir aber die um 1 verminderte Längenangabe L (L = b - a).

EXTRACT bedeutet, das ausgewählte Bitfeld soweit nach rechts zu verschieben, daß dessen erstes Bit auf Bitposition 0 kommt. Es ist eine Rechtsverschiebung um a Bits erforderlich. Gemäß der Längenangabe des Bitfeldes müssen die niedrigstwertigen l Bits des verschobenen Bitfeldes erhalten bleiben und die verbleibenden (n-l) Bits gelöscht werden. Den Maskenvektor kann man auf einfache Weise aus einem Markierungsvektor ableiten, der durch 1-aus-n-Decodierung der Bitfeldlänge L erzeugt wird.

DEPOSIT erfordert, das Bitfeld auf seine angegebene Position zu verschieben (Linksverschiebung um a Bits). Um das Bitfeld in die jeweilige Datenstruktur "hineinmodifizieren" zu können, brauchen wir einen Maskenvektor, der von Position a an L aufeinanderfolgende Bits und sonst Nullen enthält. Durch konjunktive Verknüpfung mit diesem Vektor am Ausgang des Barrel Shifter erhält man das Bitfeld in seiner gewünschten Position. Der Maskenvektor läßt sich aus zwei Markierungsvektoren A, B bilden, die aus dem Index der ersten und der letzten Bitfeldposition abgeleitet werden (ist nur die Bitfeldlänge l gegeben, ergibt sich letztere gemäß a+L-1). Die Eins des Markierungsvektors B wird zu allen niederwertigen Stellen geleitet (über die ODER-Gatter in Abbildung 2.20). Der Markierungsvektor A bestimmt die Stelle, von der an die Weiterleitung gesperrt wird (über die zwischengeschalteten UND-Gatter).

Weiterverarbeitung des Ergebnisses: Die Datenstruktur, die es aufnehmen soll, muß dafür durch UND mit dem *invertierten* Maskenvektor zunächst vorbereitet werden. Dann wird das verschobene Bitfeld in die Datenstruktur "hineingeodert" (vgl. dazu auch Abschnitt 2.2.).

3. Arithmetische Operatoren

Im folgenden wollen wir die Datenstrukturen und Schaltungen behandeln, die für das eigentliche "Rechnen" erforderlich sind. Von seiten der Hardware beschränken wir uns auf kombinatorische Schaltungen. Das ist keine bedeutende Einschränkung, da in Hochleistungs-Prozessoren ohnehin Verarbeitungseinrichtungen vorgesehen werden, die möglichst viele wichtige Operationen in einem Zyklus, also durch kombinatorische Zuordnung, erledigen. Operationen, die auf diese Weise nicht ausführbar sind, werden entweder mit Maschinenbefehlen ausprogrammiert (RISC-Architekturen) oder durch Mikroprogramme gesteuert (in CISC-Architekturen).

3.1. Addition und Subtraktion von Binärzahlen

Addieren und Subtrahieren in einer Binärstelle
Im Binären wird genau so schulmäßig gerechnet wie im Dezimalen. In einer beliebigen Stelle haben wir zwei Operandenbits und einen einlaufenden Übertrag zu verarbeiten. Wir erhalten ein Summen- und ein Übertragsbit für die nächste Stelle. Beim Subtrahieren kennzeichnen die Überträge ein "Borgen" von der jeweils höherwertigen Stelle, genau wie im Dezimalen. Allerdings sind im Binären die Rechenregeln recht einfach (vgl. Heft 2):

#	0 + 0 = 0,	
#	0 + 1 = 1	(gleichgültig, woher die Eins kommt, ob Operandenbit oder einlaufender Übertrag),
#	1 + 1 = 2	bzw. binär 10, d. h. Ergebnisbit 0, Übertrag 1,
#	1 + 1 + 1 = 3	bzw. binär 11,
#	0 - 0 = 0,	
#	1 - 1 = 0,	
#	1 - 0 = 1,	
#	0 - 1 = 1	und "geborgter" Übertrag, also 11 (es wird von 1 auf 2 ergänzt!),
#	0 - 1 - 1 = 10	(es wird von 1 + 1 = 2 auf 2 ergänzt, folglich werden die Differenz 0 und der "geborgte" Übertrag 1),
#	1 - 1 - 1 = 11	(es wird von 2 auf 3 ergänzt, folglich werden Differenz und "geborgter" Übertrag beide 1).

Abbildung 3.1 zeigt die vollständigen Wahrheitstabellen und die Schaltsymbole für Addition und Subtraktion in einer Binärstelle sowie die Zusammenschaltung für mehrere Binärstellen, wobei die Überträge jeweils in die nächsthöhere Stelle weitergegeben werden (Kaskadierung).

Abbildung 3.1 Rechnen in einer Binärstelle

Halbaddierer
Ein Halbaddierer (Half Adder) ist eine Schaltung, die zwei Binärstellen zueinander addiert, ohne einen einlaufenden Übertrag zu berücksichtigen. Abbildung 3.2 zeigt Wahrheitstabelle und Schaltung.

Abbildung 3.2 Halbaddierer

Volladdierer
Ein Volladdierer (Full Adder) addiert zwei Binärstellen unter Berücksichtigung des einlaufenden Übertrages zueinander. Die vollständige Wahrheitstabelle haben wir bereits anhand von Abbildung 3.1 kennengelernt. Ein Volladdierer kann aus zwei Halbaddierern aufgebaut werden, wobei der zweite zur Summe, die der erste gebildet hat, den einlaufenden Übertrag addiert. Der Ausgangsübertrag entsteht durch ODER-Verknüpfung der Ausgangsüberträge beider Halbaddierer. Alternativ dazu kann man die Wahrheitstabelle von Abbildung 3.1 direkt in eine DNF-Schaltung umsetzen (Abbildung 3.3).

Abbildung 3.3 Volladdierer

Mehrstelliger Addierer
Die Kaskadierung der einzelnen Volladdierer zum mehrstelligen Addierer geht bereits aus Abbildung 3.1 hervor. Abbildung 3.4 enthält einen genaueren Schaltplan eines vierstelligen Addierers. Solche Addierer sind als Schaltkreise erhältlich (z. B. der Typ '83).

Abbildung 3.4 Addierer für vier Binärstellen

Durchlaufender Übertrag (Ripple Carry)
Bei der gezeigten Schaltung läuft der Übertrag nacheinander über alle Stellen. Das kostet nur wenig Aufwand. Die Addition ist aber erst dann abgeschlossen, wenn der Ausgangsübertrag gebildet wurde. Durch die hohe Schaltungstiefe (namentlich bei Verarbeitungsbreiten von 16 Bits, 32 Bits und mehr) ergeben sich sehr lange Additionszeiten (bei einem zweistufigem Netzwerk je Binärstelle kommt man für 32 Bits auf eine Schaltungstiefe von 64).

Übertragsvorausschau (Carry Look Ahead)
Mit zusätzlichen Schaltmitteln werden Überträge parallel zur Summe gebildet. Das betrifft neben dem Ausgangsübertrag auch die Teil-Überträge, die zur Summenbildung in bestimmten Binärstellen benötigt werden. Dieses Prinzip wird in modernen Verarbeitungsschaltungen praktisch ausschließlich angewendet, weshalb wir es uns im folgenden klarmachen wollen.

Das Prinzip: wir setzen die Booleschen Gleichungen, die die Übertragsbildung in den einzelnen Stellen beschreiben, ineinander ein.

Betrachten wir zunächst die Übertragsbildung von Stelle zu Stelle (Ripple Carry).

Der Übertrag aus der ersten in die zweite Stelle:

$$C_0 = A_0B_0 \vee C_I (A_0 \vee B_0)$$

Der Übertrag aus der zweiten in die dritte Stelle:

$$C_1 = A_1B_1 \vee C_0 (A_1 \vee B_1)$$

Der Übertrag aus der dritten in die vierte Stelle:

$$C_2 = A_2B_2 \vee C_1(A_2 \vee B_2)$$

usw.

Wir können nun in der zweiten Gleichung C_0 durch die rechte Seite der ersten Gleichung ersetzen, in der dritten Gleichung C_1 durch die rechte Seite der (soeben umgeformten) zweiten Gleichung usw.

Hierzu wollen wir zwei Hilfsvariable K_i, D_i einführen, nämlich:

- K_i für die Konjunktion ($K_i = A_iB_i$),
- D_i für die Disjunktion ($D_i = A_i \vee B_i$).

Damit wird:

$C_0 = K_0 \vee D_0C_I$,

$C_1 = K_1 \vee D_1 (K_0 \vee D_0C_I) = K_1 \vee D_1K_0 \vee D_1D_0C_I$ (Klammern aufgelöst)

$C_2 = K_2 \vee D_2 (K_1 \vee D_1K_0 \vee D_1D_0C_I)$ usw.

Allgemein:

$C_i = K_i \vee D_iK_{i-1} \vee D_iD_{i-1}K_{i-2} \vee D_iD_{i-1}D_{i-2}K_{i-3} \vee ...$

Eine andere Interpretation
Wir betrachten zunächst eine einzige Binärstelle i mit den Operandenbits A_i, B_i. Auf welche Weise kann ein Übertrag zustande kommen?

1. er kann unabhängig von einem einlaufenden Übertrag *entstehen*. Diese Bedingung bezeichnet man als CARRY GENERATE G_i. Der Übertrag entsteht, wenn beide Operandenbits mit Eins belegt sind, also gilt: $G_i = A_iB_i$. (also = K_i).

2. es kann ein *einlaufender Übertrag* zum Ausgangsübertrag *durchgereicht werden*. Diese Bedingung bezeichnet man als CARRY PROPAGATE P_i. Sie wird wirksam, wenn wenigstens eines der Operandenbits mit Eins belegt ist (ansonsten würde der einlaufende Übertrag in der Stelle i "verschluckt" (absorbiert) werden). Es gilt: $P_i = A_i \vee B_i$ (also = D_i).

Wie wird nun der Übertrag in der ersten Stelle gebildet? Er kann dort entstehen, oder ein einlaufender Übertrag (CARRY INJECT C_i) wird weitergereicht.

Es gilt also: $C_0 = G_0 \text{ v } P_0C_i$.

Das gilt auch für die zweite Stelle sinngemäß; also: $C_1 = G_1 \text{ v } P_1C_0$. Für C_0 können wir aber die oben genannte Formel einsetzen: $C_1 = G_1 \text{ v } P_1G_0 \text{ v } P_1P_0C_i$.

Entsprechend gilt für die dritte Stelle $C_2 = G_2 \text{ v } P_2C_1$. Hier können wir die beiden oben genannten Formeln einsetzen: $C_2 = G_2 \text{ v } P_2G_1 \text{ v } P_2P_1G_0$.

Sinngemäß gilt in einer beliebigen Stelle i:

$C_i = P_iG_{i-1} \text{ v } P_iP_{i-1}G_{i-2} \text{ v } P_iP_{i-1}G_{i-3} \text{ v } ...$

Abbildung 3.5 zeigt das Prinzip der Übertragsweitergabe in einer Stelle sowie eine Schaltung für vier Stellen.

Abbildung 3.5 Übertragsvorausschau (Carry Look Ahead)

Erklärung:

a) zum Prinzip. Es ist der Ausgangsübertrag zu bestimmen, der über einen vorgegebenen Bereich von Binärstellen gebildet wird. Dieser Ausgangsübertrag kann zum einen innerhalb dieser Binärstellen entstehen (Carry Generate G). Zum anderen können diese Binärstellen so belegt sein, daß ein einlaufender Übertrag (Carry In CI) durchgereicht wird (Carry Porpagate P). Demgemäß ergibt sich der Ausgangsübertrag (Carry Out CO) wie folgt: $CO = G \vee P \cdot CI$.

b) Bildung der Übertragssignale P und G über 4 Binärstellen. Dieser Schaltung ist die Anordung a) nachzuschalten.

c) Bildung des Ausgangübertrags CO über 4 Binärstellen (Vereinigung von a) und b); Gatter einzeln dargestellt, einige Verknüpfungen des Verbundes a), b) zusammengefaßt.

1 - Übertragsweitergabe. Ein einlaufender Übertrag CI wird nur dann weitergegeben, wenn in allen 4 Stellen jeweils die Weitergabebedingung erfüllt ist ($P_3 \cdot P_2 \cdot P_1 \cdot P_0$ - ansonsten wird ein einlaufender Übertrag absorbiert - er verschwindet gleichsam innerhalb der 4 Stellen). 2 - ein Ausgangsübertrag wird gebildet, wenn in der 4. höchstwertigen Stelle ein Übertrag entsteht (G_3). 3 - ein Ausgangsübertrag wird gebildet, wenn in der 3. Stelle ein Übertrag entsteht und dieser über die 4. Stelle weitergegeben wird ($G_2 \cdot P_3$). 4 - ein Ausgangsübertrag wird gebildet, wenn in

der 2. Stelle ein Übertrag entsteht und dieser über die 3. und die 4. Stelle weitergegeben wird ($G_1 \cdot P_2 \cdot P_3$). 5 - ein Ausgangsübertrag wird gebildet, wenn in der 1. Stelle ein Übertrag entsteht und dieser über die 2., 3. und die 4. Stelle weitergegeben wird ($G_0 \cdot P_1 \cdot P_2 \cdot P_3$).

Mit solchen Schaltungen (die auch als Schaltkreise verfügbar sind, z. B. als Typ '182) kann man einen Addierer gemäß Abbildung 3.4 ergänzen, um den Ausgangsübertrag über vier Stellen schneller zu bilden. Man kann aber auch im Addierer selbst den Übertrag jeder Binärstelle mit einer solchen Schaltung bestimmen. Das ist beispielsweise in Arithmetik-Logik-Einheiten (ALUs) verwirklicht, die als Schaltkreis erhältlich sind (z. B. als Typ '181).

Bei großen Verarbeitungsbreiten ist es nicht mehr möglich, alle Übertragsbits parallel vorausschauend zu bestimmen: die Netzwerke werden - zu den höherwertigen Stellen hin - immer breiter, so daß man die einzelnen Verknüpfungen ihrerseits aus elementaren Gattern kaskadieren muß (was wiederum längere Durchlaufzeiten zur Folge hat). Deshalb werden solche Addierer gleichsam in Scheiben (Slices) geschnitten, und innerhalb der einzelnen Scheiben bzw. Stufen wird die Übertragsvorausschau verwirklicht. Richtwert zur Breite der einzelnen Stufe: 4...8 Bits sind typisch. Es gibt verschiedene Auslegungen (Abbildungen 3.6 und 3.7). Man kann beispielsweise den Übertrag über mehrere Bits (z. B. 4 oder 8) vorausschauend erkennen und die einzelnen Stufen kaskadieren. Es ist aber auch möglich, in der einzelnen Addierer-Stufe ausgangsseitige G- und P-Signale zu bilden und dann mit nachgeordneten gleichartigen Schaltmitteln die Überträge in die nächsten Stufen und den Gesamt-Ausgangsübertrag vorausschauend zu bilden (zweistufige Übertragsvorausschau; dafür können beispielsweise Schaltkreise der Typen '181 und '182 zusammengeschaltet werden).

Hinweis:
Die Durchlaufzeit durch ein Addierwerk wird im wesentlichen von der Übertragsweiterschaltung bestimmt. Der kritische Weg: vom eingespeisten Eingangsübertrag CI zum Ausgangsübertrag (Carry Flag CF) bzw. bis zum höchstwertigen Ergebnisbit. Im ungünstigsten Fall wirken hierbei Signale "quer" über alle Binärstellen. Betrieben wir einen Prozessor oberhalb der höchsten zulässigen Taktfrequenz, so kann es vorkommen, daß manche Rechenoperationen fehlerhaft ausgeführt werden und manche nicht. Nachweis: mit entsprechend kritischen Testbeispielen prüfen. Naheliegend: 00...0H + FF...FH + Eingangsübertrag.

Abbildung 3.6 Kaskadierte und zweistufige Übertragsvorausschau

Erklärung:

a) kaskadierte Übertragsvorausschau (Carry Skip Adder). Jede Stufe des Addierwerks besteht aus einem Ripple-Carry-Addierer ADD (z. B. Abbildung 3.4) und aus einem Übertragsvorausschaunetzwerk CLA gemäß Abbildung 3.5c. Die jeweils nachfolgende (höherwertige) Stufe erhält einen vorausschauend gebildeten Eingangsübertrag; der von den Addierern selbst gebildete Ausgangsübertrag wird gleichsam übersprungen (deshalb heißen solche Addierer auch Carry Skip Adder).

b) zweistufige Übertragsvorausschau. Jede Stufe des Addierwerks besteht aus einem Ripple-Carry-Addierer und einem Übertragsvorausschaunetzwerk gemäß Abbildung 3.5b, das die Signale P und G der gesamten Stufe liefert. Ein nachgeschaltetes Vorausschaunetzwerk (2nd Level CLA) bildet daraus die eigentlichen Übertragssignale. Der Übertrag in die höherwertige Stufe des Addierwerks (CI_1) ergibt sich so, wie anhand von Abbildung 3.5a erklärt. Ein Ausgangübertrag (CO) wird gebildet, wenn (1) in der höchstwertigen Addierer-Stufe ein Übertrag entsteht (G_1) oder wenn (2) in der vorgeordneten Stufe ein Übertrag entsteht und durch die nachfolgende Stufe weitergeleitet wird ($G_0 \cdot P_1$) oder wenn (3) ein Eingangsübertrag über beide Stufen weitergereicht wird ($CI \cdot P_0 \cdot P_1$).

Abbildung 3.7 Carry-Select-Addierer

Erklärung:
1 - erste Stufe des Addierwerks (ein Ripple-Carry-Addierer RCA); 2, 3, 4- die weiteren (höherwertigen) Stufen des Addierwerks. 5, 6 - Ripple-Carry-Addierer; 7 - Übertragsvorausschaunetzwerk; 8 - Ergebnisauswahl (1-aus-2-Multiplexer). Jede der Stufen 2, 3, 4 hat zwei Ripple-Carry-Addierer 5, 6, deren Summenausgängen ein Multiplexer 8 nachgeschaltet ist. Der Grundgedanke: wir bilden in jeder der Stufen 2, 3, 4 mit den Addierern 5, 6 zwei Ergebnisse gleichzeitig - einmal ohne und einmal mit Eingangsübertrag. Je nachdem, iob tatsächlich ein Eingangsübertrag eintrifft oder nicht, wählen wir das entsprechende Ergebnis aus (über den Multiplexer 8). Zur Übertragsvorausschau 7: auch hier entsteht der Eingangsübertrag in die jeweils nächste Stufe durch disjunktive Verknüpfung eines in der Stufe entstandenen Übertrags G und eines ggf. weitergereichten Eingangsübertrags (Weiterreichbedingung P UND Eingangsübertrag). Die Signale G und P können beispielsweise gemäß Abbildung 3.5b gebildet werden.

Hinweis:
Auch die Addierer 1 und 4 können mit Vorausschaunetzwerken (z. B. gemäß Abbildung 3.5c) versehen sein.

Zum Geschwindigkeitsgewinn
Wir beziehen uns auf die bisher erläuterten Schaltungsbeispiele. Ein aus zweistufigen Netzwerken aufgebauter vierstelliger Ripple-Carry-Addierer (Abbildungen 3.3b, 3.4) hat eine Schaltungstiefe von 8, ein Vorausschaunetzwerk gemäß Abbildung 3.5c hat eine Schaltungstiefe von 3. Die Multiplexer 8 des Carry-Select-Addierers haben eine Schaltungstiefe von 2. Tabelle 3.1 veranschaulicht das Leistungsvermögen verschiedener Auslegungen anhand der jeweils der im ungünstigsten Fall zu durchlaufenden Schaltungstiefe (vom Eingangsübertrag bis zum Ausgangsübertrag oder bis zum höchstwertigen Ergebnisbit).

Ausführung	Schaltungstiefe
Ripple-Carry, 32 Bits	64
Carry Skip, 32 Bits in 8 Stufen zu 4 Bits	um das höchstwertige Ergebnisbit zu bilden, muß der Eingangsübertrag über 7 CLA-Stufen und über einen Ripple-Carry-Addierer laufen. Schaltungstiefe also $(7 \cdot 3) + 8 = 29$
zweistufige Übertragsvorausschau, 32 Bits in 8 Stufen zu 4 Bits	erste Vorausschaustufen gem. Abb. 3.5b. In der zweiten Vorausschaustufe werden mit Netzwerken ähnlich Abb. 3.6b die Eingangsüberträge der einzelnen Addierer sowie der Ausgangsübertrag gebildet. Die zweite Vorausschaustufe ist aus 2 Netzwerken ähnlich Abb. 3.6b kaskadiert. Schaltungstiefe: Vorausschau 1. Stufe: 3, Vorausschau 2. Stufe bis Eingangsübertrag in den höchstwertigen Addierer: 4, Bildung des höchstwertigen Ergebnisbits: 8. Schaltungstiefe also $8 + 7 = 15$
Carry Select, 32 Bits in 8 Stufen zu 4 Bits	wenn jeder Addierer mit einem Vorausschaunetzwerk ausgerüstet ist, sind zur Bildung des Ausgangsübertrags insgesamt 8 Vorausschaunetzwerke zu durchlaufen. Schaltungstiefe also $8 \cdot 3 = 24$ (bei praktisch doppeltem Aufwand gegenüber Carry Skip)
Kompromißlösung: zweistufige Übertragsvorausschau und Carry-Select-Addierer in der höchstwertigen Stufe	die zweistufige Übertragsvorausschau hat eine Schaltungstiefe von 7 vom Eingangsübertrag zum Ausgangsübertrag und zum Übertrag in die höchstwertige Stufe. Die Addition hat eine Schaltungstiefe von 8, die Ergebnisauswahl eine von 2. Da der Übertrag eher feststeht als das Ergebnis, hat die Ergebnisbildung die größere Schaltungstiefe: $8 + 2 = 10$

Tabelle 3.1 Addierwerke im Leistungsvergleich

Hinweis:
Das Entwerfen von Hochleistungs-Addierwerken ist ein Geduldsspiel. Die jeweils günstigtse Auslegung wird maßgeblich von der FPGA- oder ASIC-Architektur bestimmt. Aus Handbüchern übernommene Schaltungen müssen nicht immer so implementiert werden, wie man naiverweise annimmt... Ggf. sind mehrere Probeentwürfe verschiedener Auslegung und verschiedener Stufenzahl durchzuarbeiten.

Addierer ohne Übertragsverrechnung (Carry-Save-Adder)
Ein Carry-Save-Addierer für n Stellen ist nichts anderes als die Nebeneinander-Anordnung von n Volladdierern, ohne daß die Überträge "durchgeschleift" werden. Eine solche Schaltung (Abbildung 3.8) hat 3 Eingangs-Operanden (1. Summand, 2. Summand, Eingangsübertrag) und liefert 2 Ergebnisse (Summe und Ausgangsübertrag). Der Vorteil: geringste Schaltungstiefe (im Extremfall: 2; vgl. Abbildung 3.3b), und zwar unabhängig von der Verarbeitungsbreite. Das gewährleistet geringste Verzögerungszeiten. Wozu ist eine solche Anordnung brauchbar? - Für die einzelne Addition hat das Nicht-Verrechnen der Überträge keinen Sinn. Der Vorteil kommt vielmehr dann zur Wirkung, wenn eine Vielzahl von Additionen nacheinander auszuführen ist und dazu mehrere Addierer zusammengeschaltet werden müssen (das typische Beispiel ist die Multiplikation; vgl. Abschnitt 3.7.). Man bildet dann die Zwischensummen mit Carry-Save-Addierern und sieht nur für die Schlußrechnung einen "richtigen" Addierer (mit Übertragsverrechnung) vor, der Summen und Einzel-Überträge zum Endergebnis aufaddiert.

Anwendungsbeispiel: die 3-Operanden Addition (A + B + C)
Die Addition A + B + C erfordert entweder (1) zwei hintereinandergeschaltete Ripple-Carry- oder Carry-Lookahead-Addierer (Rechnung (A + B) + C) oder (2) einen Carry-Save-Addierer und einen nachgeschalteten "gewöhnlichen" Addierer zur Übertragsverrechnung. Die Lösung (2) ist deutlich schneller - und zudem weniger aufwendig.

Abbildung 3.8 Carry-Save-Addierer

3.2.2. Die universelle Arithmetikeinheit

Eine solche Einrichtung soll - typischerweise als Teil einer ALU - Additions- und Subtraktionsoperationen mit natürlichen und ganzen Binärzahlen ausführen können.

Subtraktion im Zweierkomplement
Die Bildung des Zweierkomplements erfordert (1) die bitweise Negation des Operanden und (2) das Addieren einer Eins. Letzteres läßt sich durch Einspeisen eines Eingangsübertrags erreichen. Eine Schaltung, die addieren und subtrahieren kann, läßt sich auf Grundlage eines Addierers mit vorgeschalteten Antivalenzgattern zum bedarfsweisen Negieren aufbauen. Will man auch eine reine Zweierkomplementbildung vorsehen (ohne zweiten Operanden), so muß man diesen entweder als festen Nullvektor zuführen oder entsprechende Sperrgatter vorsehen.

Eingangsübertrag
Man braucht insgesamt drei Möglichkeiten, um den Eingangsübertrag zu bilden:

1. fest Null zum Addieren,
2. fest Eins zum Subtrahieren (Zweierkomplementbildung),
3. Einspeisung eines "geretteten" Übertrages (Carry Inject) zum Rechnen mit Binärzahlen, die länger sind als die Verarbeitungsbreite.

Erklärung:
Sind die Zahlen länger als die Verarbeitungsbreite, so kann man sie trotzdem addieren oder subtrahieren, und zwar gewissermaßen stückweise, mit den niederwertigen Stellen beginnend. Der Ausgangsübertrag muß dabei gespeichert und bei der nachfolgenden Addition oder Subtraktion wieder als Eingangsübertrag verwendet werden. Um solche Abläufe programmieren zu können, gibt es Additions- und Subtraktionsbefehle sowohl ohne als auch mit Einspeisen des geretteten Übertrags (in der x86-Architektur: ADD, SUB und ADC, SBC).

Abbildung 3.9 zeigt eine universelle 4-Bit-Arithmetikeinheit.

Abbildung 3.9 Universelle 4-Bit-Arithmetikeinheit

Erklärung:
1 - Addierwerk; 2 - XOR-Gatter zum Bilden des Zweierkomplements; 3 - UND-Gatter zum Ausblenden des 2. Operanden (nur erforderlich für die die Operation “Komplement von B”); 4 -

Steuerung Eingangsübertrag; 5 - Nullerkennung; 6 - Erkennung der Überlaufbedingung (beim Rechnen mit vorzeichenbehafteten Binärzahlen; Überlauf = Übertrag in die höchstwertige Binärstelle $\oplus$ Ausgangsübertrag); 7 - Ausgangsübertrag (Carry Flag).

Rekonstruktion des Übertrags in die höchstwertige Stelle
Verwendet man fertige Addierer- oder ALU-Schaltkreise, so ist der in die höchstwertige Stelle einlaufende Übertrag unzugänglich. Er wird aber benötigt, um die Überlaufbedingung zu bestimmen. Abhilfe: der Übertrag wird außerhalb des Schaltkreises aus den Operandebits rekonstruiert (Abbildung 3.10; vgl. auch Heft 2): $C_n = A \oplus B \oplus S$. Die Overflow-Bedingung ergibt sich dann zu $A \oplus B \oplus S \oplus C_o$.

Abbildung 3.10 Rekonstruktion des Übertrags in die höchstwertige Stelle

3.2.3. Bildung des Zweierkomplements - eine Alternative

Die Abbildungen 3.11 und 3.12 zeigen, wie man das Zweierkomplement auf (womöglich) einfachere Weise (als durch Invertieren und Addieren einer 1) bilden kann. Das Prinzip: ein beliebiges Bit n wird nur dann invertiert, wenn sein rechter (= niederwertiger) Nachbar = 1 ist oder seinerseits invertiert wurde.

Abbildung 3.11 Alternative Zweierkomplementbildung (1)

Erklärung:
1 - das niedrigstwertige Bit wird direkt durchgereicht; 2 - das folgende Bit wird dann invertiert, wenn das niedrigstwertige Bit = 1 ist; 3 - durch Vergleichen von Ein- und Ausgangsbelegung wird festgestellt, ob das rechts benachbarte (= niederwertige) Bit invertiert wurde oder nicht; 4 - ist das rechts benachbarte Bit = 1, so ist auch zu invertieren.

Zum Verfahren: es ist das bitweise Invertieren und das Addieren einer 1 nachzubilden. Wird zu einem invertierten Bit eine 1 addiert, so stellt das die ursprüngliche Belegung wieder her (Bit = 0: 1 + 1 = 0; Bit = 1: 0 + 1 = 1). In den folgenden Stellen ist dann nur noch ein einlaufender Übertrag zu berücksichtigen. Kommt ein Übertrag, so ist die ursprüngliche Belegung weiterzugeben, kommt keiner, so ist das Bit zu invertieren. In die zweite Stelle läuft nur dann ein Übertrag ein, wenn Stelle $2^0 = 0$ ist. Folglich ist zu invertieren, wenn Stelle $2^0 = 1$ ist. In den nachfolgenden Stellen läuft in folgenden Fälle *kein* Übertrag ein (es ist also zu invertieren):

- wenn die rechts benachbarte Stelle = 1 ist (beim Invertieren würde sie zu 0 werden und so einen ggf. dort einlaufenden Übertrag absorbieren),
- wenn die rechts benachbarte Stelle iherseits invertiert wurde (das heißt, wenn bereits in diese Stelle kein Übertrag eingelaufen ist - dann kann auch keiner weitergegeben werden).

Abbildung 3.12 Alternative Zweierkomplementbildung (2)

Erklärung zur Funktionsweise (Abbildung 3.12):

- Bit 2^0 wird stets durchgereicht,
- die folgenden Bits werden durchgereicht, sofern sie = 0 sind,
- die erste (niedrigstwertige) 1 wird ebenfalls durchgereicht,
- die auf die erste 1 folgenden Bits werden invertiert.

Der Priority Encoder stellt die Position der 1. Eins fest. Über die nachgeschalteten kaskadierten ODER-Gatter werden die XOR-Gatter aller folgenden höherwertigen Bitpositionen mit Einsen versorgt, um diese Bits zu invertieren.

Hinweis:
Als weitere Alternative verbleibt die Invertierung und das Nachschalten eine Increment-Netzwerks (s. den folgenden Abschnitt). Welche Lösung besser ist, hängt von der jeweiligen FPGA- oder ASIC-Architektur ab (ist also ggf. anhand von Probeentwürfen herauszufinden...).

3.2.4. Addieren und Subtrahieren von Festwerten (Increment - Decrement)

Ist einer der Operanden beim Addieren oder Subtrahieren ein Festwert, so können die Rechenschaltungen entsprechend vereinfacht werden (Abbildungen 3.13, 3.14):

- alle Bitpositionen, in die nur Überträge einlaufen, können mit Halbaddierern bestückt werden,
- mit Festwerten beschaltete Gatter können gelegentlich weggelassen werden,
- Festwerte bieten gelegentlich die Möglichkeit, Kürzungsregeln der Schaltalgebra anzuwenden.

Abbildung 3.13 Festwertaddition am Beispiel R := A + 1 (Increment)

Erklärung:
In der Stelle 2^0 genügt eine einfache Negation (0 + 1 = 1; 1 + 1 = 0). Ist $a_0 = 1$, so wird ein Übertrag weitergereicht (1 + 1 = 10). In den folgenden Stellen ist nur noch der einlaufende Übertrag zu verrechnen. Hierzu genügen Halbaddierer (vgl. Abbildung 3.2).

Abbildung 3.14 Festwertsubtraktion am Beispiel R := A - 1 (Decrement)

Erklärung:
Wir rechnen mit dem Zweierkomplement von - 1. A - 1 = A + FF...FH (gebildet aus FF...EH + 1). a) - damit werden die Volladdierer in den einzelnen Bitpositionen vereinfacht; b) - Ergebnis der Vereinfachung (die Bildung der negierten Operandenbits ist nicht dargestellt). In Stelle 2^0 ist eine 1 zu addieren, r_0 entspricht deshalb dem negierten Operandenbit a_0. In den folgenden Stellen ist jeweils eine 1 zu addieren und der einlaufende Übertrag zu verrechnen.

Hinweis:
Mit “richtigen” Addierwerken entwerfen und die Entwicklungssoftware die Optimierung tun lassen (zu empfehlen: anhand überschaubarer Schaltungen - z. B. gemäß den Abbildungen 3.13 und 3.14 - überprüfen, ob die Sofware das überhaupt kann...).

3.3. Laden mit Vorzeichenerweiterung

Ist eine ganze Binärzahl kürzer als die Verarbeitungsbreite, so muß beim Auffüllen das Vorzeichen berücksichtigt werden: es ist in alle höherwertigen Stellen zu übernehmen (Vorzeichenerweiterung; Sign Extend). Abbildung 3.15 zeigt eine entsprechende Schaltung (die u. a. der Arithmetikeinheit gemäß Abbildung 3.9 vorgeschaltet werden kann).

Abbildung 3.15 Vorzeichenerweiterung (Sign Extend)

3.4. Arithmetische Verschiebungen

Arithmetische Verschiebungen unterscheiden sich nur in einem Punkt von den logischen Verschiebungen, die wir in Abschnitt 2.4. behandelt haben: beim Rechtsverschieben wird nicht die Null oder ein Füllwert, sondern das Vorzeichen (also die Belegung der höchstwertigen Stelle) in alle freiwerdenden Stellen eingetragen (Vorzeichenerweiterung). Beim Linksverschieben gibt es keine Unterschiede.

3.5. Vergleichen

Vergleichen durch Subtrahieren
Zwei Binärzahlen können durch Subtraktion miteinander verglichen werden. Zu den auswertbaren Bedingungen siehe Tabelle 3.2 sowie Heft 2.

Hinweis:
Sind beide Operanden gleich, so entsteht beim Subtrahieren stets ein Ergebnis 0, unabhängig davon, ob die Operanden als natürliche oder als ganze Binärzahlen interpretiert werden. Somit ist, um zwei Operanden Bit für Bit auf Gleichheit hin zu überprüfen, die Subtraktion ("arithmetischer Vergleich") ebenso nutzbar wie die bitweise Antivalenz ("logischer Vergleich"; vgl. Abschnitt 2.2.). Manche Architekturen haben deshalb keine "logischen" Vergleichsbefehle (Beispiel: x86).

Vorsicht, Falle:
Typischerweise wird beim "logischen" Vergleichen ein kürzerer Operand *vorzeichenlos* auf die Verarbeitungsbreite erweitert (Nullerweiterung), beim "arithmetischen" Vergleichen (z. B. bei den x86-CMP-Befehlen) hingegen *vorzeichengerecht* (vgl. auch Heft 2).

Vergleich natürlicher (vorzeichenloser) Binärzahlen durch Subtraktion A- B			
Vergleichsergebnis	Rechenergebnis	Vergleichsbedingung	Signalverknüpfung
A = B	0	Summe = 0 (dabei entsteht stets ein Ausgangsübertrag)	Z
A ≠ B	nicht 0	Summe ≠ 0	$\overline{Z}$
A < B	negativ (= Bereichs-unterschreitung)	kein Ausgangsübertrag	$\overline{C}$
A > B	positiv	Summe ≠ 0, Ausgangsübertrag	$\overline{Z}\ C$ bzw. $\overline{Z \vee \overline{C}}$
A ≤ B	negativ oder 0	Summe = 0 oder kein Ausgangsübertrag	$Z \vee \overline{C}$
A ≥ B	positiv oder 0	Ausgangsübertrag	C
Vergleich ganzer (vorzeichenbehafteter) Binärzahlen durch Subtraktion A- B			
Vergleichsergebnis	Rechenergebnis	Vergleichsbedingung	Signalverknüpfung
A = B	0	Summe = 0 (dabei entsteht stets ein Ausgangsübertrag)	Z
A ≠ B	nicht 0	Summe ≠ 0	$\overline{Z}$
A < B	entweder negativ oder positiv und Bereichs-unterschreitung (Overflow)	Overflow ≠ Vorzeichen	$O \oplus S$
A > B	nicht 0 und entweder positiv oder negativ und Bereichsüberschreitung (Overflow)	Summe ≠ 0 und Overflow = Vorzeichen	$\overline{Z}\ (\overline{O \oplus S})$ bzw. $\overline{Z \vee (O \oplus S)}$
A ≤ B	0 bzw. entweder negativ oder positiv und Bereichs-unterschreitung (Overflow)	Summe = 0 oder Overflow ≠ Vorzeichen	$Z \vee (O \oplus S)$
A ≥ B	entweder positiv oder negativ und Bereichs-überschreitung (Overflow)	Overflow = Vorzeichen	$\overline{O \oplus S}$

Tabelle 3.2 Vergleichsbedingungen beim Subtrahieren

Erklärung:
Z - Ergebnis = 0; C - Ausgangübertrag; O - Overflow (Augangsübertrag ⊕ Übertrag in höchstwertige Stelle); S - Vorzeichen (höchstwertige Stelle des Ergebnisses).

Vergleichen mit Vergleichsnetzwerken - der Magnitude Comparator

Solche Vergleichsnetzwerke dienen dazu, vorzeichenlose Binärzahlen miteinander zu vergleichen. Die Vergleichsbedingungen werden direkt in einer Wahrheitstabelle erfaßt und in eine Schaltung umgesetzt. Wir beginnen zunächst mit einer einzigen Binärstelle (Tabelle 3.3, Abbildung 3.16).

A	B	A = B	A < B	A > B
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

Tabelle 3.3 Vergleichsbedingungen in einer Binärstelle (die Bits A, B werden miteinander verglichen)

Aus Tabelle 3.3 können folgende Schaltgleichungen abgelesen werden:

- A = B: $\overline{A}\ \overline{B} \vee AB$
- A < B: $\overline{A}\ B = \overline{(A \vee \overline{B})}$
- A > B: $A\ \overline{B} = \overline{(\overline{A} \vee B)}$

Abbildung 3.16 Vergleichsnetzwerke für eine Binärstelle

Solche Netzwerke lassen sich kaskadieren (Abbildung 3.17).

Abbildung 3.17 Kaskadierung einstelliger Vergleichsnetzwerke

Erklärung:
Wir betrachten eine höherwertige Binärstelle i. Die Vergleichssignale aus den niederwertigen Stellen sind gegeben. Die kaskadierten Vergleichsbedingungen werden dann folgendermaßen gebildet:

- A < B gilt, wenn in der aktuellen Stelle A < B erkannt wird oder wenn in der aktuellen Stelle A = B erkannt wird und wenn in den niederwertigen Stellen A < B gilt,
- A = B gilt, wenn in der aktuellen Stelle A = B erkannt wird und wenn auch in den niederwertigen Stellen A = B gilt,
- A > B gilt, wenn in der aktuellen Stelle A > B erkannt wird oder wenn in der aktuellen Stelle A = B erkannt wird und wenn in den niederwertigen Stellen A > B gilt.

Prinzip: ist in der aktuellen Stelle A = B, so bleibt die Vergleichsaussage aus den niederwertigen Stellen erhalten, ansonsten wird sie durch die Vergleichsaussage der aktuellen Stelle ersetzt.

Fertige Vergleicherschaltkreise betreffen typischerweise 4 Binärstellen (Abbildungen 3.18, 3.19).

Abbildung 3.18 Der Vergleichsschaltkreis ‘85 (Innenschaltung und Funktionstabelle)

Abbildung 3.19 Kaskadierung von Vergleichsschaltkreisen (2 Schaltungsbeispiele)

3.6. BCD-Arithmetik

Binär codierte Dezimalzahlen sind 4-Bit-Angaben im Bereich zwischen 0000B und 1001B. Sie werden zunächst wie "gewöhnliche" Binärzahlen zueinander addiert oder voneinander subtrahiert. Das geht aber nur solange gut, wie das Ergebnis den Wert 9 nicht überschreitet. Beispiel: 5 + 6 = 11. Genauer: ein Übertrag in die folgende Stelle und ein Ergebnis 0001B. Aus 0101B + 0110B ergibt sich aber 1011B. Der Ausweg (Tabelle 3.4): in einem solchen Fall ist beim Addieren zusätzlich der Wert "6" zu addieren (Dezimalkorrektur). Beim Subtrahieren ist entsprechend eine "6" zusätzlich abzuziehen.

binäre Rechenergebnisse in einer Dezimalstelle				gewünschtes Ergebnis in der betreffenden Dezimalstelle	
Addition		Subtraktion[1)]			
dezimal	binär	dezimal	binär	dezimal	binär[2)]
10	0 1010	-10	1 0110	0 und DC	1 0000
11	0 1011	-9	1 0111	1 und DC	1 0001
12	0 1100	-8	1 1000	2 und DC	1 0010
13	0 1101	-7	1 1001	3 und DC	1 0011
14	0 1110	-6	1 1010	4 und DC	1 0100
15	0 1111	-5	1 1011	5 und DC	1 0101
16	1 0000	-4	1 1100	6 und DC	1 0110
17	1 0001	-3	1 1101	7 und DC	1 0111
18	1 0010	-2	1 1110	8 und DC	1 1000
19	1 0011	-1	1 1111	9 und DC	1 1001

Tabelle 3.4 Zum Prinzip der Dezimalkorrektur

Erklärung:
1) - dezimal: Zehnerkomplement, binär: Zweierkomplement; 2) - BCD-Ergebnis mit weiterzugebendem Dezimalübertrag (die 5. (höchstwertige) Binärstelle); DC - Dezimalübertrag in die nächst-höhere Stelle.

Die Dezimalkorrektur setzt ein, wenn das Additionsergebnis den Wert 9 über- oder wenn das Subtraktionsergebnis der Wert 0 unterschreitet. Die Tabelle enthält alle beim Addieren oder beim

Subtrahieren möglichen Ergebnisse, die in diesem Bereich liegen. Gewünscht wird aber ein Ergebnis im Bereich von 0...9 sowie die Weitergabe eines dezimalen Übertrags an die nächsthöhere Dezimalstelle.

Es istz ersichtlich, daß man jeweils auf das gewünschte Ergebnis kommt, wenn man beim Addieren zum binären Ergebnis eine 6 (0110B) addiert bzw. beim Subtrahieren eine 6 vom binären Ergebnis abzieht (bzw. deren Zweierkomplement 1 1010B addiert)n

Abbildung 3.20 zeigt 2 Rechenbeispiele; Abbildung 3.21 veranschaulicht eine entsprechende Schaltung, die addieren und subtrahieren kann.

Abbildung 3.20 Rechenbeispiele der Dezimalkorrektur

Abbildung 3.21 BCD-Addition und Subtraktion

3.7. Flagbits

Wir wollen im folgenden zusammengefaßt darstellen, welche Bedingungen, die bei arithmetischen Operationen entstehen, zur programmseitigen Auswertung (z. B. Abfrage in Verzweigungsbefehlen) vorgesehen sind. Als Beispiel wählen wir die x86-Architektur. Dabei erläutern wir nur jene Flagbits, die als Resultate arithmetischer Operationen von anwendungspraktischer Bedeutung sind. Weitere Flagbits bleiben unberücksichtigt, auch wenn sie bei der Ausführung arithmetischer Operationen gestellt oder ausgewertet werden.

CF: Übertrag (Carry Flag)
Das Bit wird gesetzt, wenn beim Addieren ein Ausgangsübertrag bzw. beim Subtrahieren ein Borgen aus der jeweils höchsten Binärstelle heraus auftritt (zu den Spitzfindigkeiten beim Subtrahieren vgl. Heft 2). Tritt in den entsprechenden Befehlen keine derartige Bedingung auf, so wird CF gelöscht. Je nach der aktuellen Operandenlänge (8, 16 bzw.. 32 Bits) gilt die Bitposition 7, 15 bzw. 31 als höchste Binärstelle. Das Bit CF kann als Eingangsübertrag in eine nachfolgende Addition oder Subtraktion einfließen (Befehle ADC, SBC).

ZF: Null (Zero Flag)
Das Bit wird gesetzt, wenn bei einer arithmetischen oder logischen Operation das Resultat in allen Bits den Wert Null hat. Ist es verschieden von Null, wird ZF gelöscht. Das Resultat wird entsprechend der jeweiligen Operandenlänge (8, 16 bzw. 32 Bits) ausgewertet.

SF: Negativ (Sign Flag)
Das Bit bei einer arithmetischen oder logischen Operation auf den Wert des jeweils höchstwertigen Resultatbits gesetzt. Das heißt, SF entspricht der Belegung des Resultatbits 7, 15 bzw. 31, je nach aktueller Operandenlänge (8, 16 bzw. 32 Bits). SF repräsentiert die Vorzeichenstelle in entsprechend langen ganzen Binärzahlen (0: positiv; 1: negativ).

OF: Überlauf (Overflow Flag)
Das Bit zeigt an, daß das als ganze Binärzahl interpretierte Resultat seinen Wertebereich über- bzw. unterschritten hat. Das Bit wird gesetzt, (1) wenn ein Übertrag in die höchste Binärstelle (d. h. in das Vorzeichen) des Resultats, aber kein Ausgangsübertrag aufgetreten ist, oder (2) wenn kein Übertrag in die höchste Binärstelle, aber ein Ausgangsübertrag aufgetreten ist. Andernfalls wird es gelöscht. Die jeweilige Operandenlänge (8, 16 bzw. 32 Bits) bestimmt die Auswahl der Binärstellen (Vorzeichenbits 7, 15 bzw. 31).

Hinweis:
OF kann mit Befehlen INTO geprüft werden. INTO wirkt so, daß bei gesetztem OF ein Interrupt 4 wirksam wird.

AF: BCD-Übertrag (Auxiliary Carry Flag)
Das Bit wird gesetzt, wenn beim Addieren ein Ausgangsübertrag bzw. beim Subtrahieren ein Borgen aus der Bitposition 3 heraus auftritt. Tritt in den entsprechenden Befehlen keine derartige Bedingung auf, wird AF gelöscht. AF betrifft nur Bitposition 3, unabhängig von der aktuellen Operandenlänge.

3.8. Multiplikation und Division

Im Binären wird genau so multipliziert und dividiert, wie wir es aus dem Schulunterricht vom Dezimalen her kennen (vgl. Heft 2).

In der Hardware laufen solche Operationen zumeist in mehreren Maschinenzyklen ab. Das gilt für die Multiplikation in vielen Fällen, für die Division praktisch immer (auf Grund des Verhältnisses von Nutzungshäufigkeit und Aufwand lohnt sich für die Division keine besondere Hochleistungs-Hardware). Die Abläufe werden in vielen Prozessoren durch ein internes Mikroprogramm gesteuert. In RISC-Prozessoren (d. h. in solchen, die gemäß der "reinen akademischen Lehre" ausgeführt sind), müssen Multiplikation und Division ausprogrammiert werden (durch Unterprogramme oder durch eingefügte Befehlsfolgen). Dazu sind in manchen Architekturen besondere Befehle vorgesehen, in denen die elementaren Verschiebe- und Additions- bzw. Subtraktionsoperationen kombiniert sind (Multiplikationsschritt, Divisionsschritt).

Sonderfälle
Die Multiplikation mit bzw. die Division durch Zweierpotenzen läßt sich durch entsprechende Links- bzw. Rechtsverschiebung des Multiplikanden bzw. Dividenden ersetzen. Manche Compiler machen davon Gebrauch, um das vergleichsweise oft vorkommende Verdoppeln, Vervierfachen, Halbieren usw. mit Verschiebebefehlen auszuführen, die deutlich schneller ablaufen als Multiplikation oder Division (solche Operationen werden besonders in Adreßrechnungen häufig benötigt).

Einfache Beschleunigungsmaßnahmen
Ein leistungsfähiger Barrel Shifter ist die wichtigste Voraussetzung für schnelle Multiplikations- bzw. Divisionsschritte. Hinzu kommt eine Schaltung, die die höchstwertige Eins in einem

Binärvektor erkennt (LAST OCCURRENCE; vgl. Abschnitt 2.5.). Bei der Multiplikation kann man dann (1) die kleinere Zahl als Multiplikator verwenden und (2) die Multiplikation nach Verarbeitung der höchstwertigen Eins des Multiplikators abbrechen (das ist z. B. im i486 verwirklicht). Bei der Division kann man, ganz analog zum schulmäßigen Rechnen, anfänglich die höchstwertigen Einsen von Dividend und Divisor gewissermaßen untereinanderschieben. Die einzelnen Schaltungen haben wir bereits kennengelernt. Um die gewünschten Beschleunigungs-Wirkungen hervorzubringen, ist es nur noch erforderlich, innerhalb des Prozessors entsprechende Verbindungswege und Steuermöglichkeiten (z. B. über Mikrobefehle) vorzusehen.

Hardware-Multiplizierer

Die Multiplikation läuft praktisch darauf hinaus, gegeneinander verschobene Multiplikandenwerte aufzuaddieren. Jedem Multiplikatorbit entspricht dabei ein Multiplikandenwert. Dessen Rechtsverschiebung entspricht dem Index (der Bitadresse) des jeweiligen Multiplikandenbits (bei Rechtsindizierung: Bit 0: keine Verschiebung, Bit 1: Verschiebung um 1 Bit usw.). Ist das Multiplikatorbit gleich Eins, wird der betreffende Multiplikandenwert aufaddiert, ansonsten nicht.

Dieses Prinzip läßt sich direkt in eine Schaltung umsetzen (Abbildung 3.22). Es ist ein "Baum" von Addierern vorgesehen, wobei den "obersten" Addierern der Multiplikand über UND-Verknüpfungen vorgeschaltet ist, die jeweils alle mit dem entsprechenden Bit des Multiplikators verbunden sind (je nach dessen Belegung wird also entweder der Multiplikand oder eine Null auf den Addierer-Eingang geschaltet). Die nachfolgenden Addierer dienen dazu, die Teilprodukte zu summieren. Die notwendige Verschiebung wird durch entsprechend "versetzte" Verbindungen gewährleistet (siehe dazu das Anschlußschema in Abbildung 3.22).

Abbildung 3.22 4-Bit-Binärmultiplizierer

Multiplizieren mit Carry-Save-Addierern

Der als Addiererbaum aufgebaute Binärmultiplizierer nach Abbildung 3.22 ist ein gutes Modell, das wirklich funktioniert (ausprobieren!) und das Wesentliche veranschaulicht. Der Nachteil: er ist - angesichts des ohnehin getriebenen Aufwands - einfach noch zu langsam. Der Grund: die Verzögerungszeit jeder Addierer-"Schicht" wird durch die Verzögerungszeit der Übertragsrechnung bestimmt. Man verwendet deshalb in der Praxis in den oberen Schichten Carry-Save-Addierer und setzt nur zur "Schlußrechnung" einen Carry-Lookahead-Addierer ein, um aus Summe und Übertrag das Endergebnis zu bilden. Abbildung 3.23 veranschaulicht eine solche Struktur (die üblicherweise - nach dem Erfinder - als "Wallace Tree" bezeichnet wird). Beachten Sie, daß jeweils ein Addierer der ersten Schicht gleich drei verschobene Multiplikanden aufaddieren kann. (Einzelheiten, wie das Anschließen und Durchschalten des (verschobenen) Multiplikanden und die Stellenzahl der einzelnen Additionen ergeben sich durch Vergleich mit Abbildung 3.22.)

Abbildung 3.23 Multiplizierer mit Carry-Save-Addierern (Wallace Tree)

Booth's Algorithmus

Bei der "gewöhnlichen" Binärmultiplikation (der Hardware-Vergegenständlichung des schulmäßigen Multiplizierens), die wir soeben besprochen haben, werden für zwei n-stellige Argumente n Teilprodukte gebildet, und zwar durch konjunktive Verknüpfung des Multiplikanden mit jeweils einem Bit des Multiplikators. Die Teilprodukte werden, unter entsprechender Stellenverschiebung, durch einen Addiererbaum aufaddiert, der eine Tiefe von ld n hat. Dafür braucht man n-1 Addierer.

Das Ziel von Booth's Algorithmus besteht darin, Teilprodukte unter Auswertung von jeweils *zwei Bits des Multiplikators* zu bilden. Diese werden, um jeweils zwei Bits gegeneinander verschoben, zum Endergebnis aufaddiert. So hat man nur n/2 Teilprodukte, und die Tiefe des Addiererbaums verringert sich auf ld (n/2). Man kommt so mit (n/2)-1 Addierern aus.

Damit das Vorhaben Sinn hat, müssen die Teilprodukte durch ein Verfahren gebildet werden, das (1) eine geringere Schaltungstiefe und (2) einen geringeren Schaltungsaufwand hat als die Addition.

Booth's Algorithmus gilt für Binärzahlen in Zweierkomplementdarstellung. Es werden, von der Vorzeichenstelle an, Gruppen aus zwei Multiplikatorbits gebildet, die durch das jeweils benachbarte niederwertige Bit ergänzt werden (die niedrigstwertige Gruppe wird fest mit Null ergänzt). Aus diesen Drei-Bit-Gruppen (Recode Groups) wird das Teilprodukt gemäß Tabelle 3.5 gebildet.

Dreiergruppe			Teilproduktbildung
Bit i+1	Bit i	Bit i-1	
0	0	0	Null addieren
0	0	1	Multiplikand addieren
0	1	0	Multiplikand addieren
0	1	1	doppelten Multiplikand addieren
1	0	0	doppelten Multiplikand subtrahieren
1	0	1	Multiplikand subtrahieren
1	1	0	Multiplikand subtrahieren
1	1	1	Null subtrahieren

Tabelle 3.5 Booth's Algorithmus: Teilproduktbildung

Negative Teilprodukte entstehen durch Invertieren des Teilproduktes und Einspeisen einer Eins in den Eingangsübertrag des jeweiligen nachgeschalteten Addierers. Die Teilproduktbildung

erfordert also nur Verschiebung (zum Verdoppeln des Multiplikanden) und Komplementbildung (zum Subtrahieren).

Booth's Algorithmus ist die Grundlage vieler Hardware-Multiplizierer, die als fertige Schaltkreise erhältlich sind.

3.9. Gleitkommaoperationen

Eine Gleitkommazahl besteht aus Vorzeichen, Exponent und Signifikand (bzw. Mantisse). Vorzeichen und Wert sind voneinander unabhängig (Sign-Magnitude- Darstellung). Der Exponent wird vorzeichenlos dargestellt. Die codierte Angabe ist dabei um einen Versatz (Bias) gegenüber dem "wirklichen" Wert des Exponenten erhöht.

Multiplikation und Division
Bei Multiplikation und Division können die Exponenten und Signifikanden unabhängig voneinander verarbeitet werden. Die Exponenten werden zueinander addiert bzw. voneinander subtrahiert; die Signifikanden werden miteinander multipliziert bzw. durcheinander dividiert. (Potenzen zur gleichen Basis werden multipliziert, indem man ihre Exponenten addiert; sie werden dividiert, indem man ihre Exponenten subtrahiert.)

Addition und Subtraktion
Potenzen mit verschiedenen Exponenten kann man nicht zueinander addieren oder voneinander subtrahieren. Dazu müssen vielmehr die Exponenten beider Operanden gleich sein. Bei Gleitkommazahlen müssen wir also die Mantissen so verändern, daß beide Exponenten gleich sind (*Normalisierung*). Betrachten wir den einfachsten Fall, daß einer der Operanden nicht verändert wird. Um die Aufgabe $s_1 \cdot 2^b + s_2 \cdot 2^c$ zu lösen, können wir die Zweierpotenz im zweiten Operanden folgendermaßen umschreiben: $2^c = 2^{(c-b)} \cdot 2^b$, denn es gilt $2^{(c-b)+b} = 2^c$ (Potenzmultiplikation bedeutet Exponentenaddition).

Vor der eigentlichen Addition müssen wir den Ausdruck $2^{(c-b)}$ noch mit dem Signifikanden s_2 multiplizieren. Also in Vorbereitung einer Addition erst eine Multiplikation? - So aufwendig ist dies aber gar nicht: Eine Multiplikation mit einer Zweierpotenz ist schließlich nichts anderes als eine Rechtsverschiebung. Wir müssen somit den Signifikanden s_2 nur um (c-b) Bits nach rechts verschieben. Den Verschiebe-Wert erhalten wir aus der Differenz beider Exponenten.

Wir sehen, daß auch die Gleitkommaverarbeitung auf den Grundschaltungen beruht, die wir bereits kennengelernt haben. Abbildung 3.24 gibt einen Überblick über Hardware-Strukturen für die vier Grundrechenarten.

Hinweis:
Die vielfältigen Sonder- und Nebenaufgaben einer Gleitkomma-Hardware, wie Runden, Reagieren auf besondere Zahlendarstellungen, auf Genauigkeitsverlust, Bereichsüberschreitung usw. wollen wir nicht näher betrachten.

Abbildung 3.24 Hardware zum Rechnen mit Gleitkommazahlen

In praktisch ausgeführten Gleitkomma-Verarbeitungswerken (Floating-Point Processing Units; FPUs) werden üblicherweise viele Schaltmittel kombiniert genutzt, und die einzelnen Rechenabläufe erfordern mehrere Maschinenzyklen. Sie werden durch ein internes Mikroprogramm gesteuert. Hochleistungs-Schaltungen haben gesonderte Verarbeitungseinrichtungen für die einzelnen Operationen, die auch in sich weitgehend mit kombinatorischen Zuordnungen verwirklicht sind, z. B. die Multiplikation durch einen Hardware-Multiplizierer. Die Division (a : b) wird in manchen Maschinen durch eine Multiplikation mit dem Kehrwert (Reciprocal) des Divisors ersetzt (a · 1/b), wobei 1/b mit einem schnell ablaufenden Näherungsverfahren berechnet wird. Im obersten Leistungsbereich sind mehrere Verarbeitungswerke gleichzeitig nutzbar. Verschiedene Architekturen sehen auch mehrere Verarbeitungswerke für gleichartige Operationen vor, z. B. mehrere Addier- und Multiplizierwerke (Superskalar-Architekturen).