

sam genutzt werden müssen. Der Zugriff auf diesen privaten Speicher erfolgt nicht über den Hauptbus, was den Busverkehr erheblich reduziert. Es sind auch andere Schemas denkbar, wie zum Beispiel die Verwendung von Caches (siehe weiter unten).

Mehrprozessorsysteme haben gegenüber anderen Typen von Parallelrechnern den Vorteil, dass sich mit dem Programmiermodell eines einzigen gemeinsam genutzten Speichers leichter arbeiten lässt. Stellen Sie sich beispielsweise ein Programm vor, das auf einer mit einem Mikroskop hergestellten Fotografie von Körpergewebe nach Krebszellen suchen soll. Man legt die digitalisierte Fotografie im gemeinsam genutzten Speicher ab und weist jedem Prozessor einen bestimmten Bereich der Fotografie für die Suche zu. Da jeder Prozessor auf den gesamten Speicher zugreifen kann, ist es auch kein Problem, wenn eine untersuchte Zelle im zugeordneten Bereich des einen Prozessors beginnt, sich aber bis in den nächsten Bereich erstreckt.

Multicomputersysteme

Multicomputer-systeme sind hoch skalierbar und bestehen aus gekoppelten Computern mit privatem Speicher. Ihre Programmierung mittels Nachrichtenaustausch ist sehr aufwendig.

Obwohl Mehrprozessorsysteme mit einer mittleren Anzahl von Prozessoren (≤ 256) relativ leicht zu bauen sind, ist es überraschend schwierig, große Systeme zu konstruieren. Die Schwierigkeit besteht darin, alle Prozessoren mit dem Speicher zu verbinden. Um diese Probleme zu umgehen, haben viele Entwickler einfach das Konzept eines gemeinsam genutzten Speichers verworfen und bauen stattdessen Systeme aus vielen miteinander verbundenen Computern, die jeweils ihren eigenen privaten Speicher besitzen, aber keinen Speicher gemeinsam nutzen. Derartige Systeme heißen **Multicomputersysteme**. Man spricht auch davon, dass die Prozessoren in einem Multicomputersystem **lose gekoppelt** sind, im Gegensatz zu den **eng gekoppelten** Prozessoren in einem Mehrprozessorsystem.

Die Prozessoren in einem Multicomputersystem kommunizieren, indem sie einander Nachrichten schicken – so wie beim E-Mail-Versand, nur viel schneller. Bei großen Systemen ist es nicht praktikabel, jeden Computer mit jedem anderen zu verbinden. Daher arbeitet man mit Topologien wie 2-D- und 3-D-Netzen, Bäumen und Ringen. Meldungen von einem Computer müssen daher oft durch einen oder mehrere Zwischencomputer oder Switches (Schalter) laufen, um von der Quelle zum Ziel zu gelangen. Dennoch lassen sich ohne größere Schwierigkeiten Nachrichtenlaufzeiten in der Größenordnung von wenigen Mikrosekunden erreichen. Es wurden bereits Multicomputersysteme mit über 250.000 Prozessoren gebaut, wie zum Beispiel der Blue Gene/P von IBM.

Da Multiprozessorsysteme einfacher zu programmieren und Multicomputersysteme einfacher zu bauen sind, beschäftigt sich die Forschung intensiv mit dem Entwurf von Hybrid-Systemen, die die positiven Eigenschaften von beiden Typen verbinden. Derartige Rechner versuchen, die Illusion eines gemeinsamen Speichers zu vermitteln, ohne den Aufwand zu treiben, ihn tatsächlich zu konstruieren. Kapitel 8 beschäftigt sich eingehend mit Multiprozessor- und Multicomputersystemen.

2.2 Hauptspeicher (Primärspeicher)

Der **Speicher** (Memory, Storage) ist der Teil des Computers, der die Programme und Daten aufnimmt. Ohne Speicher, den die Prozessoren lesen oder beschreiben können, gäbe es keine speicherprogrammierten Digitalrechner. Obwohl vielfach der Begriff „Storage“ für den Hauptspeicher verwendet wird, wird er zunehmend zur Bezeichnung von Plattspeicher verwendet.

2.2.1 Bits

Die Grundeinheit des Speichers ist die binäre Ziffer, genannt **Bit** (Binary Digit). Ein Bit enthält entweder eine 0 oder eine 1. Es ist die einfachste mögliche Einheit. (Eine Einrichtung, die nur Nullen speichern kann, kommt wohl kaum als Basis für ein Speichersystem infrage – es sind mindestens zwei Werte erforderlich.)

Oft hört man, dass Computer deshalb mit Binärarithmetik arbeiten, weil sie „effizient“ sei. Eigentlich ist gemeint, dass sich digitale Informationen speichern lassen, indem zwischen diskreten Werten einer stetigen physikalischen Größe wie Spannung oder Strom unterschieden wird. Je mehr Werte zu unterscheiden sind, desto geringer ist der Abstand zwischen benachbarten Werten und desto unzuverlässiger ist der Speicher. Beim binären Zahlensystem müssen sich nur zwei Werte voneinander unterscheiden lassen. Daher bietet es die zuverlässigste Methode für die Codierung digitaler Daten. Wenn Sie mit Binärzahlen nicht vertraut sind, sollten Sie sich *Anhang A* ansehen.

Für manche Computer, wie zum Beispiel die IBM-Großrechner, betont die Werbung, dass sie neben der binären auch die dezimale Arithmetik beherrschen. Der Trick dabei ist, dass man eine Dezimalziffer mithilfe von 4 Bits im sogenannten **BCD-Format (Binary Coded Decimal)** speichert. Mit vier Bits kann man 16 Kombinationen bilden, wovon man 10 für die Ziffern 0 bis 9 verwendet und 6 ungenutzt lässt. Das folgende Beispiel zeigt die Zahl 1944 in BCD- und rein binärer Darstellung mit jeweils 16 Bits:

BCD: 0001 1001 0100 0100Binär: 0000011110011000

Die **Binärdarstellung** von Zahlen ist effizienter als die Dezimaldarstellung, da mit der gleichen Zahl von Bits mehr Werte codiert werden können.

Im Dezimalformat können 16 Bits die Zahlen von 0 bis 9999 speichern, d.h. nur 10.000 Kombinationen, während eine reine 16-Bit-Binärzahl 65.536 verschiedene Kombinationen erlaubt. Aus diesem Grund sagt man, dass die Binärdarstellung effizienter sei.

Wie sieht es nun aus, wenn ein begabter junger Elektroingenieur ein enorm zuverlässiges elektronisches Bauelement erfindet, das die Ziffern von 0 bis 9 unmittelbar speichern kann, indem es den Bereich von 0 bis 10 Volt in 10 Intervalle unterteilt? Vier dieser Bauelemente können jede Dezimalzahl von 0 bis 9999 speichern und 10.000 Kombinationen bereitstellen. Es lassen sich damit auch Binärzahlen speichern, wobei man lediglich 0 und 1 benutzt. Die vier Bauelemente können in diesem Fall nur 16 Kombinationen darstellen. Bei derartigen Bauelementen ist das Dezimalsystem offensichtlich viel effizienter als das Binärsystem.

2.2.2 Speicheradressen

Speicher bestehen aus einer Reihe von **Zellen** (oder **Speicherstellen**), von denen jede eine Informationseinheit aufnehmen kann. Jede Zelle hat eine Nummer, die sogenannte **Adresse**, über die Programme auf die Zellen verweisen können. Ein Speicher mit n Zellen hat die Adressen 0 bis $n - 1$. Alle Zellen in einem Speicher enthalten dieselbe Anzahl von Bits. Besteht eine Zelle aus k Bits, kann sie jede der 2^k verschiedenen Bitkombinationen aufnehmen. ►Abbildung 2.8 zeigt drei verschiedene Anordnungen für einen 96-Bit-Speicher. Beachten Sie, dass benachbarte Zellen (per Definition) aufeinanderfolgende Adressen haben.

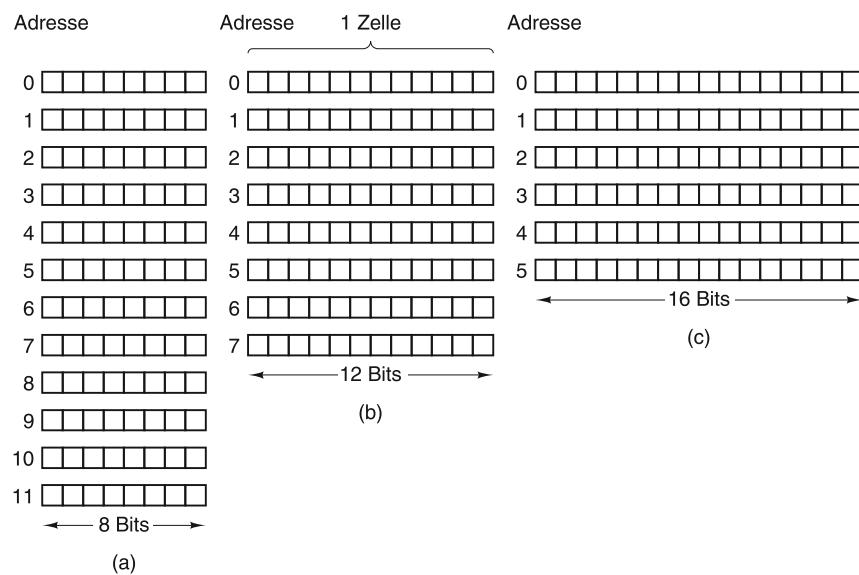


Abbildung 2.8: Drei Arten, einen 96-Bit-Speicher zu organisieren

Computer, die das binäre Zahlensystem (einschließlich oktaler und hexadezimaler Notation für Binärzahlen) verwenden, drücken Speicheradressen als Binärzahlen aus. Hat eine Adresse m Bits, so lassen sich maximal 2^m Zellen adressieren. Beispielsweise sind für eine Adresse, die auf den Speicher von ► Abbildung 2.8a verweisen soll, mindestens 4 Bits erforderlich, um alle Zahlen von 0 bis 11 auszudrücken. Für ► Abbildung 2.8b und c genügt dagegen eine 3-Bit-Adresse. Die Anzahl der Bits in der Adresse bestimmt die maximale Anzahl der direkt adressierbaren Zellen im Speicher und ist unabhängig von der Anzahl der Bits je Zelle. Sowohl ein Speicher mit 2^{12} Zellen zu je 8 Bit als auch ein Speicher mit 2^{12} Zellen zu je 64 Bit erfordert 12-Bit-Adressen.

► Tabelle 2.1 gibt die Anzahl der Bits je Zelle für einige kommerzielle Computer an.

Computer	Bits je Zelle
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

Tabelle 2.1: Anzahl der Bits je Zelle für einige historisch interessante, kommerzielle Computer

Die Bedeutung einer Zelle besteht darin, dass sie die kleinste adressierbare Einheit ist. In den letzten Jahren hat sich bei nahezu allen Computerherstellern eine 8-Bit-Zelle – ein sogenanntes **Byte** oder auch **Oktett** – als Standard etabliert. Bytes werden zu **Wörtern** gruppiert. Ein Computer mit einem 32-Bit-Wort verwendet 4 Byte/Wort und ein Computer mit einem 64-Bit-Wort 8 Byte/Wort. Die Bedeutung eines Wortes besteht darin, dass die meisten Befehle mit ganzen Wörtern arbeiten, beispielsweise bei der Addition zweier Wörter. Daher verfügt ein 32-Bit-Prozessor über 32-Bit-Register und Befehle, die 32-Bit-Wörter verarbeiten, während ein 64-Bit-Prozessor mit 64-Bit-Registern und Befehlen zum Verarbeiten von 64-Bit-Wörtern ausgestattet ist.

Eine **Speicherzelle** ist die kleinste adressierbare Einheit im Hauptspeicher. In heutigen Rechnern speichert eine Zelle typischerweise ein Byte.

2.2.3 Bytereihenfolge

Die Bytes in einem Wort kann man von links nach rechts oder von rechts nach links nummerieren. Auf den ersten Blick mag die jeweilige Wahl unwichtig erscheinen. Wie wir aber gleich sehen werden, hat dies beträchtliche Auswirkungen. ►Abbildung 2.9a zeigt einen Teil des Speichers eines 32-Bit-Computers, dessen Bytes von links nach rechts nummeriert sind, wie bei der SPARC oder bei IBM-Großrechnern. ►Abbildung 2.9b gibt dagegen die analoge Darstellung eines 32-Bit-Computers mit Nummerierung von rechts nach links an, wie es bei der Intel-Familie der Fall ist. Das erste System, bei dem die Zählung am „großen“ (höherwertigen) Ende beginnt, bezeichnet man als **Big Endian**, im Unterschied zu **Little Endian** von Abbildung 2.9b. Diese Begriffe gehen auf Jonathan Swift zurück, der in „Gullivers Reisen“ Politiker karikiert, die Kriege aus Anlass der Kontroverse führten, ob Eier am spitzen („little“) oder am breiten („big“) Ende aufzuschlagen seien. In der Rechnerarchitektur wurden diese Begriffe erstmals in einem amüsanten Artikel von Cohen (1981) verwendet.

Adresse	Big Endian				Adresse	Little Endian			
0	0	1	2	3	3	2	1	0	0
4	4	5	6	7	7	6	5	4	4
8	8	9	10	11	11	10	9	8	8
12	12	13	14	15	15	14	13	12	12
	Byte				Byte				
	32-Bit-Wort				32-Bit-Wort				
	(a)				(b)				

Abbildung 2.9: (a) Big-Endian-Speicher (b) Little-Endian-Speicher

Man muss wissen, dass sowohl in Big-Endian- als auch in Little-Endian-Systemen eine 32-Bit-Ganzzahl mit einem numerischen Wert von beispielsweise 6 durch die Bits 110 in den drei rechten (niederwertigen) Bits eines Wortes und Nullen in den linken 29 Bits dargestellt werden. Im Big-Endian-Schema befinden sich diese Bits in Byte 3 (oder 7, 11 usw.), während sie im Little-Endian-Schema in Byte 0 (oder 4, 8 usw.) stehen. In beiden Fällen hat das Wort, das diese Ganzzahl enthält, die Adresse 0.

Wenn Computer nur Ganzzahlen speichern müssten, gäbe es keine Probleme. Viele Anwendungen erfordern aber eine Kombination aus Ganzzahlen, Zeichenfolgen und anderen Datentypen. Nehmen wir beispielsweise einen einfachen Personaldatensatz mit einer Zei-

chenfolge (Name des Mitarbeiters) und zwei Ganzzahlen (Alter und Abteilungsnummer) an. Die Zeichenfolge wird mit einem oder mehreren 0-Bytes abgeschlossen, um ein Datenwort zu füllen. ► Abbildung 2.10a zeigt die Big-Endian-Darstellung für Jim Smith, Alter 21 Jahre, Abteilung 260 ($1 \times 256 + 4 = 260$), während die entsprechende Little-Endian-Darstellung in ► Abbildung 2.10b zu sehen ist.

Big Endian				Little Endian				Übertragung von Big Endian zu Little Endian				Übertragung und Vertauschung			
0	J	I	M	0	M	I	J	0	M	I	J	0	J	I	M
4	S	M	I	T	T	I	M	4	T	I	M	S	S	M	I
8	H	0	0	0	0	0	H	8	0	0	0	H	H	0	0
12	0	0	0	21	0	0	21	12	21	0	0	0	21	0	0
16	0	0	1	4	0	0	1	16	4	1	0	0	0	0	1

(a)

(b)

(c)

(d)

Abbildung 2.10: (a) Ein Personaldatensatz für einen Big-Endian-Computer (b) Der gleiche Datensatz in einem Little-Endian-Computer (c) Das Ergebnis nach Übertragung dieses Datensatzes von einem Big-Endian- auf einen Little-Endian-Computer (d) Das Ergebnis durch Vertauschen der Bytes von (c)

Little Endian und Big Endian bezeichnen die Bytereihenfolge von Daten. Die Verwendung zweier Formate erschwert den Datenaustausch zwischen Rechnern.

Beide Darstellungen sind korrekt und intern konsistent. Die Probleme setzen erst ein, wenn ein Computer den Datensatz über ein Netzwerk an einen anderen Computer sendet. Wir nehmen an, dass der Big-Endian-Computer den Datensatz byteweise an den Little-Endian-Computer sendet, wobei die Übertragung mit Byte 0 beginnt und mit Byte 19 endet. (Wir gehen als Optimisten davon aus, dass die Bits in den Bytes bei der Übertragung nicht umgedreht werden, da wir schon genug Probleme haben.) Somit wandert das Byte 0 des Big-Endian-Computers in den Little-Endian-Speicher bei Byte 0 usw., wie es ► Abbildung 2.10c zeigt.

Wenn der Little-Endian-Computer den Namen ausdrückt, klappt das vorzüglich; das Alter kommt aber als 21×2^{24} heraus und die Abteilungsnummer ist ebenfalls durcheinandergeraten. Diese Situation kommt dadurch zustande, dass die Übertragung die Reihenfolge der Zeichen in einem Wort umgedreht hat, wie es sein soll, aber auch die Bytes in einer Ganzzahl umgekehrt hat, was nicht sein darf.

Als naheliegende Lösung könnte man die Software die Bytes in einem Wort umdrehen lassen, nachdem die Kopie erfolgt ist. Das Ergebnis wäre dann die Darstellung in ► Abbildung 2.10d. Dieses Verfahren bringt die beiden Ganzzahlwerte in Ordnung, dreht aber die Zeichenfolge in „MIJTIMS“, und das „H“ steht auf verlorenem Posten. Die Umkehrung der Zeichenfolge findet statt, weil der Computer beim Lesen zuerst das Byte 0 (ein Leerzeichen), dann Byte 1 (M) usw. aus dem Speicher abruft.

Eine einfache Lösung dafür gibt es nicht. Ein funktionierendes, jedoch ineffizientes Verfahren besteht darin, vor jedem Datenelement einen Header einzufügen, der die Art der nachfolgenden Daten (Zeichenfolge, Ganzzahl usw.) und deren Länge angibt. Dann kann sich der Empfänger auf die erforderlichen Konvertierungen beschränken. Jedenfalls dürfte nun klar sein, dass das Fehlen eines Standards für die Bytereihenfolge beim Datenaustausch zwischen unterschiedlichen Computern ein großes Ärgernis darstellt.

2.2.4 Fehlerkorrekturcodes

In Computerspeichern können aufgrund von Spannungsspitzen im Netzteil, durch kosmische Strahlung oder aus anderen Gründen gelegentlich Fehler auftreten. Als Schutz vor solchen Fehlern setzen manche Speicher Codes für die Fehlererkennung und -korrektur ein. Diese Codes ergänzen jedes Speicherwort in bestimmter Weise durch zusätzliche Bits. Beim Lesen eines Wortes aus dem Speicher wird anhand dieser Zusatzbits festgestellt, ob ein Fehler aufgetreten ist.

Um zu verstehen, wie sich Fehler behandeln lassen, muss zunächst einmal klar sein, was ein Fehler eigentlich ist. Angenommen, ein Speicherwort besteht aus m Datenbits, denen wir r redundante Prüfbits hinzufügen. Die Gesamtlänge ist dann n (d.h. $n = m + r$). Eine n -Bit-Einheit mit m Daten- und r Prüfbits wird auch als **n -Bit-Codewort** bezeichnet.

Für zwei beliebige Codewörter, z.B. 10001001 und 10110001, kann man dann bestimmen, wie viele korrespondierende Bits sich voneinander unterscheiden. In diesem Fall sind es drei. Um die Anzahl der abweichenden Bits zu ermitteln, verknüpft man die beiden Codewörter bitweise mit der booleschen XOR-Funktion (exklusives Oder, Antivalenz) und zählt die Anzahl der 1-Bits im Ergebnis. Die Anzahl der Bitstellen, in denen sich zwei Codewörter voneinander unterscheiden, wird **Hamming-Abstand** (Hamming Distance) genannt [Hamming, 1950]. Dieses Maß besagt insbesondere, dass d Einzelbitfehler erforderlich sind, um zwei Codewörter mit dem Hamming-Abstand d ineinander umzuwandeln. Beispielsweise haben die Codewörter 11110001 und 00110000 einen Hamming-Abstand von 3, weil drei Einzelbitfehler erforderlich sind, um das eine Wort in das andere zu überführen.

Bei einem m -Bit-Speicherwort sind alle 2^m Bitmuster zulässig. Aufgrund der Art, wie die Prüfbits berechnet werden, sind aber nur 2^m der 2^n Codewörter zulässig. Taucht bei einer Speicherleseoperation ein ungültiges Codewort auf, weiß der Computer, dass ein Speicherfehler aufgetreten ist. Mit dem Algorithmus zur Berechnung der Prüfbits ist es möglich, eine vollständige Liste der zulässigen Codewörter zusammenzustellen und aus dieser Liste die beiden Codewörter herauszusuchen, deren Hamming-Abstand minimal ist. Dieser Abstand ist der Hamming-Abstand des vollständigen Codes.

Die Eigenschaften eines Codes für Fehlererkennung und -korrektur hängen von dessen Hamming-Abstand ab. Um d Einzelbitfehler zu erkennen, benötigt man einen Code mit einem Abstand von $d + 1$, weil es bei einem solchen Code nicht möglich ist, dass d Einzelbitfehler ein gültiges Codewort in ein anderes gültiges Codewort umwandeln können. Entsprechend ist für die Korrektur von d Einzelbitfehlern ein Code mit einem Abstand von $2d + 1$ erforderlich, weil dann die zulässigen Codewörter so weit auseinanderliegen, dass das ursprüngliche Codewort selbst bei d Änderungen immer noch näher als irgendein anderes Codewort ist und daher eindeutig bestimmt werden kann.

Fehlerkorrekturcodes verwenden zusätzliche Bits in einem Wort, um fehlerhafte Daten bei z.B. Speicherzugriffen zu erkennen und eventuell zu korrigieren.

Als einfaches Beispiel eines Fehlererkennungscodes betrachten wir einen Code, bei dem den Daten ein einzelnes **Paritätsbit** (Parity Bit) angehängt wird. Das Paritätsbit wird so gewählt, dass die Anzahl der 1-Bits im Codewort gerade (oder ungerade) ist. Ein solcher Code hat den Abstand 2, da jeder Einzelbitfehler ein Codewort mit der falschen Parität erzeugt. Anders ausgedrückt: Es sind zwei Einzelbitfehler erforderlich, um von einem gültigen Codewort zu einem anderen gültigen Codewort zu gelangen. Mit diesem Code lassen sich Einzelfehler erkennen. Wenn der Prozessor ein Wort mit der falschen Parität aus dem Speicher liest, wird ein Fehlerzustand signalisiert. Das Programm kann dann zwar nicht fortgesetzt werden, es berechnet aber wenigstens keine falschen Ergebnisse mehr.

Sehen Sie sich als Beispiel für einen Fehlerkorrekturcode einen Code mit nur vier gültigen Codewörtern an:

0000000000, 0000011111, 1111100000 und 1111111111

Dieser Code hat den Abstand 5, kann also Doppelfehler korrigieren. Wenn das Codewort 0000000111 eintrifft, weiß der Empfänger, dass das Original 0000011111 gewesen sein muss (wenn nicht mehr als zwei Fehler aufgetreten sind). Ändert aber ein dreifacher Fehler das Codewort 0000000000 in 0000000111, so lässt sich der Fehler nicht korrigieren.

Angenommen, wir möchten einen Code mit m Datenbits und r Prüfbits entwickeln, bei dem sich alle Einzelbitfehler korrigieren lassen. Zu jedem der 2^m gültigen Speicherwörter gibt es n ungültige Codewörter mit einem Abstand von 1. Man generiert sie, indem man systematisch jedes der n Bits im daraus gebildeten n -Bit-Codewort invertiert. So erfordert jedes der 2^m gültigen Speicherwörter $n + 1$ Bitmuster (für die n möglichen Fehler und das korrekte Muster). Da die Gesamtzahl der Bitmuster 2^n ist, muss $(n + 1) 2^m \leq 2^n$ sein. Mit $n = m + r$ wird diese Bedingung zu $(m + r + 1) \leq 2^r$. Bei gegebenem m erhält man damit eine untere Grenze für die Anzahl der Prüfbits, die zur Korrektur von Einzelfehlern erforderlich sind. ►Tabelle 2.2 gibt die Anzahl der Prüfbits an, die für verschiedene Speicherwortgrößen notwendig sind.

Wortgröße	Prüfbits	Gesamtgröße	Overhead in %
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Tabelle 2.2: Anzahl der erforderlichen Prüfbits für einen Code, der einen Einzelfehler korrigieren kann

Diese theoretische untere Grenze lässt sich mit einer Methode erreichen, die von Richard Hamming (1950) stammt. Bevor wir Hamming's Algorithmus zeigen, betrachten wir eine einfache Grafik, die das Konzept eines Fehlerkorrekturcodes für 4-Bit-Speicherwörter veranschaulicht. Das Venn-Diagramm in ► Abbildung 2.11 enthält drei Kreise A , B und C , die zusammen sieben Mengen bilden. Als Beispiel codieren wir das 4-Bit-Speicherwort 1100 in den Mengen AB , ABC , AC und BC mit einem Bit je Menge (in alphabetischer Reihenfolge). ► Abbildung 2.11a zeigt diese Codierung.

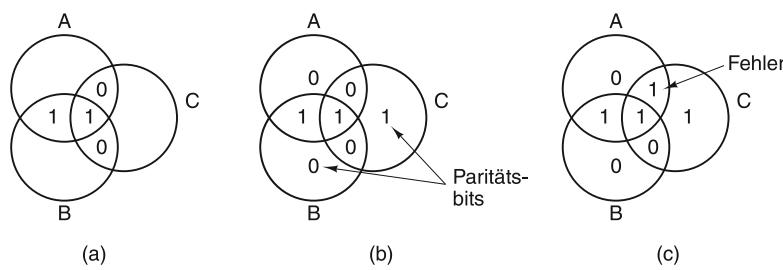


Abbildung 2.11: (a) Codierung von 1100 (b) Gerade Parität ergänzt (c) Fehler in AC

Als Nächstes fügen wir jeder der drei leeren Mengen ein Paritätsbit hinzu, um gerade Parität zu erzeugen, wie es ►Abbildung 2.11b zeigt. Per Definition ist jetzt die Summe der Bits in jedem der drei Kreise A , B und C eine gerade Zahl. In Kreis A haben wir die vier Zahlen 0, 0, 1 und 1, die zusammen 2, also eine gerade Zahl, ergeben. In Kreis B haben wir die Zahlen 1, 1, 0 und 0, die zusammen 2, also ebenfalls eine gerade Zahl, ergeben. Das Gleiche gilt für Kreis C . Bei diesem Beispiel sind die Ergebnisse in allen Kreisen gleich, in anderen Beispielen wären aber auch Summen von 0 und 4 möglich. Diese Abbildung entspricht einem Codewort mit 4 Datenbits und 3 Paritätsbits.

Nehmen wir nun an, dass das Bit in der Menge AC verfälscht wird, d.h. von 0 auf 1 wechselt, wie es in Abbildung 2.11c zu sehen ist. Der Computer erkennt jetzt, dass die Kreise A und C eine falsche (ungerade Parität) aufweisen. Sie lassen sich nur mit der Einzelbitänderung korrigieren, die AC wieder auf 0 bringt, womit der Fehler behoben ist. Auf diese Weise kann der Computer Einzelbit-Speicherfehler automatisch reparieren.

Sehen wir uns nun an, wie sich ein Fehlerkorrekturcode für Speicherwörter beliebiger Größe nach dem Hamming-Algorithmus konstruieren lässt. Ein Hamming-Code fügt einem m -Bit-Wort r Paritätsbits hinzu und bildet ein neues Wort mit der Länge $m + r$ Bits. Die Bits werden mit 1 (nicht 0) beginnend nummeriert, wobei Bit 1 das höchstwertige Bit (ganz links) ist. Alle Bits, deren Bitnummer eine Potenz von 2 ist, sind Paritätsbits; die übrigen werden für die eigentlichen Daten verwendet. Bei einem 16-Bit-Wort kommen also 5 Paritätsbits hinzu. Die Bits 1, 2, 4, 8 und 16 sind Paritätsbits, der Rest Datenbits. Insgesamt hat das Speicherwort 21 Bits (16 Datenbits, 5 Paritätsbits). Wir arbeiten in diesem Beispiel (willkürlich) mit gerader Parität.

Jedes Paritätsbit berechnet sich aus bestimmten Bitpositionen, und zwar so, dass sich eine gerade Anzahl von Einsen in den Prüfpositionen ergibt. Es gilt:

- Bit 1 berechnet sich aus den Bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21.
- Bit 2 berechnet sich aus den Bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19.
- Bit 4 berechnet sich aus den Bits 4, 5, 6, 7, 12, 13, 14, 15, 20, 21.
- Bit 8 berechnet sich aus den Bits 8, 9, 10, 11, 12, 13, 14, 15.
- Bit 16 berechnet sich aus den Bits 16, 17, 18, 19, 20, 21.

Im Allgemeinen ergibt sich Bit b aus den Bits b_1, b_2, \dots, b_j , sodass $b_1 + b_2 + \dots + b_j = b$ gilt. So wird beispielsweise Bit 5 von den Bits 1 und 4 gesetzt, weil $1 + 4 = 5$. Bit 6 wird von den Bits 2 und 4 gesetzt, weil $2 + 4 = 6$, usw.

►Abbildung 2.12 zeigt den Aufbau eines Hamming-Codes für das 16-Bit-Speicherwort 1111000010101110. Das 21-Bit-Codewort lautet 001011100000101101110. Um zu sehen, wie die Fehlerkorrektur abläuft, sehen wir uns einmal an, was geschieht, wenn Bit 5 durch eine Spannungsspitze in der Stromversorgung invertiert wird. Das neue Codewort ist dann 001001100000101101110 statt 001011100000101101110. Die 5 Paritätsbits werden überprüft und liefern die folgenden Ergebnisse:

- Paritätsbit 1 falsch (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 enthalten fünf Einsen)
- Paritätsbit 2 richtig (2, 3, 6, 7, 10, 11, 14, 15, 18, 19 enthalten sechs Einsen)
- Paritätsbit 4 falsch (4, 5, 6, 7, 12, 13, 14, 15, 20, 21 enthalten fünf Einsen)
- Paritätsbit 8 richtig (8, 9, 10, 11, 12, 13, 14, 15 enthalten zwei Einsen)
- Paritätsbit 16 richtig (16, 17, 18, 19, 20, 21 enthalten vier Einsen)

Die Gesamtzahl der Einsen in den Bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 und 21 sollte eine gerade Zahl sein, da wir hier mit gerader Parität arbeiten. Das falsche Bit muss eines der Bits sein, aus denen Paritätsbit 1 berechnet wird, nämlich Bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 oder 21. Paritätsbit 4 ist falsch, was bedeutet, dass eines der Bits 4, 5, 6, 7, 12, 13, 14, 15, 20 oder 21 falsch sein muss. Der Fehler muss bei einem der Bits liegen, die in beiden Listen enthalten sind, also 5, 7, 13, 15 oder 21. Da Bit 2 richtig ist, scheiden die Bits 7 und 15 aus. Bit 8 ist ebenfalls richtig, sodass Bit 13 ausscheidet. Schließlich ist Bit 16 korrekt, was Bit 21 eliminiert. Es kommt also nur noch Bit 5 als Fehler infrage. Da es als 1 gelesen wurde, muss es 0 gewesen sein. Auf diese Weise lassen sich Fehler korrigieren.

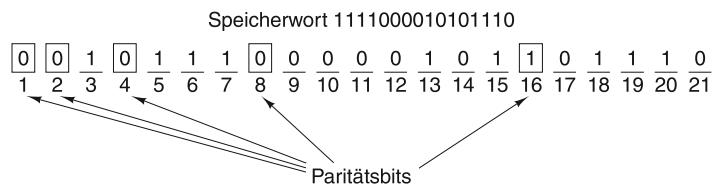


Abbildung 2.12: Aufbau des Hamming-Codes für das Speicherwort 1111000010101110 durch Hinzufügen von 5 Prüfbits zu den 16 Datenbits

Das falsche Bit lässt sich nach einer einfachen Methode finden, indem man zunächst alle Paritätsbits berechnet. Wenn alle richtig sind, gab es keinen (oder mehr als einen) Fehler. Dann addiert man alle falschen Paritätsbits, indem man 1 für Bit 1, 2 für Bit 2, 4 für Bit 4 usw. zählt. Die sich ergebende Summe ist die Position des falschen Bits. Sind beispielsweise die Paritätsbits 1 und 4 falsch, während 2, 8 und 16 richtig sind, muss Bit 5 ($1 + 4$) invertiert worden sein.

2.2.5 Cache-Speicher

Bisher waren Prozessoren immer schneller als der Speicher. Mit jeder Speicherverbesserung ging auch eine Prozessorverbesserung einher, sodass das Ungleichgewicht erhalten blieb. Tatsächlich wurden die Prozessoren aber sogar noch schneller, da sich immer mehr Schaltungen auf einem Chip unterbringen ließen und die CPU-Entwickler diese neuen Möglichkeiten für Fließbandverarbeitung und superskalaren Betrieb nutzten. Die Speicherentwickler haben neue Technologien gewöhnlich genutzt, um die Kapazität ihrer Chips und nicht die Geschwindigkeit zu erhöhen. Das Ungleichgewicht wird sich in Zukunft also eher noch verstärken. Es bedeutet in der Praxis, dass die CPU nach Ausgabe einer Speicherabfrage erst einmal viele CPU-Zyklen abwarten muss, bevor sie das benötigte Wort erhält. Je langsamer der Speicher ist, umso mehr Zyklen muss die CPU warten.

Wie bereits angedeutet, kann man dieses Problem auf zweierlei Arten lösen. Am einfachsten ist es, Speicherzugriffe (LOAD-Befehle) zu starten, wenn sie im Programm auftauchen, das Programm aber weiter auszuführen, ohne auf die Daten zu warten. Erst wenn ein Befehl das Speicherwort benötigt, dieses aber noch nicht angekommen ist, wird der Prozessor angehalten und auf die Daten gewartet. Je langsamer der Speicher ist, umso häufiger tritt dieses Problem auf und umso größer ist in diesem Fall der Zeitverlust. Wenn zum Beispiel ein Befehl von fünf auf den Speicher zugreift und die Speicherzugriffszeit 5 Zyklen beträgt, verdoppelt sich die Ausführungszeit gegenüber einem Ablauf, der den Speicher ohne Verzögerung lesen kann. Dauert aber der Speicherzugriff 50 Zyklen, so erhöht sich die Ausführungszeit um den Faktor 11 (5 Zyklen für die Befehlsausführung plus 50 Zyklen für das Warten auf den Speicher).

Bei der zweiten Lösung wartet der Prozessor nicht, bis die Daten verfügbar sind. Stattdessen darf der Compiler keinen Code erzeugen, der auf noch nicht eingetroffene Wörter zugreift. Leider ist das leichter gesagt als getan. Oft gibt es nach einem `LOAD` nichts zu tun; daher muss der Compiler `NOP`-Befehle (`NOP` = No Operation) einfügen, die nichts weiter bewirken, als einen Slot zu belegen und Zeit zu verschwenden. Praktisch ersetzt diese Vorgehensweise den Hardware- durch einen Software-Stillstand, der Leistungsrückgang ist aber der gleiche.

Tatsächlich hat dieses Problem nichts mit Technik, sondern mit Wirtschaftlichkeit zu tun. Die Ingenieure könnten durchaus Speicher bauen, die so schnell wie Prozessoren sind. Um bei voller Geschwindigkeit zu laufen, müssten sich die Speicher aber auf dem CPU-Chip befinden (weil der Weg über den Bus zum Speicher sehr langsam ist). Bestückt man einen CPU-Chip mit einem großen Speicher, wird er größer und teurer. Aber auch wenn es auf die Kosten nicht ankäme, gibt es Grenzen für die Größe eines CPU-Chips. So muss man letztlich zwischen einem kleinen schnellen und einem großen langsamen Speicher wählen. Wir möchten aber einen großen schnellen Speicher zu geringen Kosten haben.

Erfreulicherweise sind Techniken bekannt, wie man einen kleinen schnellen Speicher so mit einem großen langsamen Speicher kombinieren kann, dass man (fast) die Geschwindigkeit des schnellen und die Kapazität des großen Speichers zu angemessenem Preis erzielt. Der kleine schnelle Speicher heißt **Cache** (von französisch *cacher* – verstecken). Wir beschreiben nun kurz, wie man Caches einsetzt und wie sie funktionieren. *Kapitel 4* geht noch ausführlich auf dieses Thema ein.

Der Cache beruht auf dem einfachen Grundkonzept, dass die am häufigsten gebrauchten Speicherwörter im Cache aufbewahrt werden. Wenn die CPU ein Wort benötigt, sieht sie zuerst im Cache nach. Nur wenn sie das Wort hier nicht findet, greift sie auf den Hauptspeicher zu. Befindet sich ein wesentlicher Anteil der benötigten Wörter im Cache, dann lässt sich die durchschnittliche Zugriffszeit erheblich verringern.

Ein **Cache** ist ein schneller Zwischenspeicher, der den Zugriff der CPU auf häufig verwendete Daten beschleunigt.

Über Erfolg oder Misserfolg entscheidet demnach, welcher Anteil der Wörter im Cache vorhanden ist. Es ist längst bekannt, dass Programme nicht vollkommen zufällig auf ihre Speicher zugreifen. Zeigt eine bestimmte Speicherreferenz auf Adresse A , so liegt die nächste Speicherreferenz sehr wahrscheinlich in der Nähe von A . Das Programm selbst ist ein einfaches Beispiel dafür. Außer bei Sprüngen und Prozeduraufrufen kommen die Befehle aus aufeinanderfolgenden Speicherstellen. Zudem geht der größte Teil der Abarbeitungszeit von Programmen auf das Konto von Schleifen, in denen eine begrenzte Zahl von Befehlen wiederholt ausgeführt wird. Analog greift ein Programm, das Matrizen verarbeitet, wahrscheinlich immer wieder auf dieselbe Matrix zu, bevor es mit einer anderen Aufgabe fortfährt.

Die zu beobachtende Eigenschaft, dass Computerprogramme innerhalb eines gewissen Zeitabschnitts nur auf einen kleinen Teil des gesamten Speicherraums zugreifen, bezeichnet man als **Lokalitätseigenschaft** (Locality Principle), auf der alle Cache-Systeme beruhen. Dieser Eigenschaft entsprechend werden beim Verweis auf ein Wort nicht nur dieses Wort, sondern auch einige seiner Nachbarn aus dem großen langsamen Speicher in den Cache geholt. Wenn der Prozessor diese Wörter beim nächsten Mal benötigt, kann er schnell darauf zugreifen. ►Abbildung 2.13 zeigt eine übliche Anordnung von CPU, Cache und Hauptspeicher. Wird ein Wort in einem kurzen Zeitintervall k -mal gelesen oder geschrieben, dann benötigt der Computer nur 1 Referenz auf den langsamen Speicher und $k - 1$ Referenzen auf den schnellen Speicher. Je größer k , desto besser die Gesamtleistung.

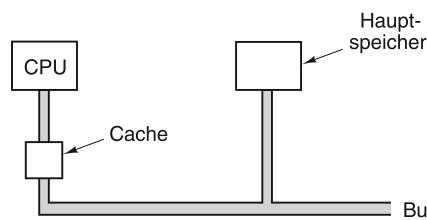


Abbildung 2.13: Der Cache befindet sich logisch zwischen CPU und Hauptspeicher. Es gibt aber mehrere physische Positionen, an denen er sich unterbringen lässt.

Wir können diese Berechnung formalisieren, indem wir die Cache-Zugriffszeit c , die Hauptspeicherzugriffszeit m und die **Trefferquote** (Hit Ratio) h einführen. Die Trefferquote verkörpert den Anteil aller Referenzen, die der Cache bedienen kann. Für unser kleines Beispiel im vorhergehenden Absatz gilt: $h = (k - 1)/k$. Einige Autoren definieren auch die **Fehlerrate** (Miss Ratio), die $1 - h$ beträgt.

Mit diesen Definitionen können wir die mittlere Zugriffszeit wie folgt berechnen:

$$\text{Mittlere Zugriffszeit} = c + (1 - h)m$$

Für $h \rightarrow 1$ lassen sich alle Referenzen aus dem Cache bedienen und die Zugriffszeit nähert sich c an. Andererseits ist bei $h \rightarrow 0$ jedes Mal eine Referenz auf den Speicher erforderlich und die Zugriffszeit nähert sich $c + m$ an – zuerst eine Zeit c zur (erfolglosen) Prüfung des Cache und dann eine Zeit m für den Speicherzugriff. Bei manchen Systemen lässt sich parallel zur Suche im Cache ein Speicherzugriff einleiten, sodass bei erfolgloser Cache-Suche der Speicherzyklus bereits im Gang ist. Diese Strategie setzt aber voraus, dass der Speicherzyklus bei einem Cache-Treffer sofort gestoppt werden kann, was die Implementierung verkompliziert.

Ein **Cache** ist in Zeilen mit mehreren Wörtern organisiert. Wird eine ganze Zeile geladen, ist dies effizienter als das Laden der einzelnen Wörter. Aufgrund der Lokalität im Programm werden die anderen Wörter mit hoher Wahrscheinlichkeit auch benötigt.

Gemäß der Lokalitätseigenschaft werden Hauptspeicher und Caches in Blöcke fester Größe unterteilt. Im Zusammenhang mit dem Cache bezeichnet man diese Blöcke allgemein als **Cachezeilen** (Cache Lines). Bei einem Cache-Zugriff fehler wird die gesamte Cachezeile – und nicht nur das benötigte Wort – aus dem Hauptspeicher geladen. Bei einer Zeilengröße von 64 Byte wird bei einem Zugriff auf die Speicheradresse 260 die gesamte Zeile, bestehend aus Byte 256 bis 319, in eine Cachezeile geladen. Mit etwas Glück werden kurz danach auch einige der anderen Wörter in der Cachezeile benötigt. Dieser Vorgang ist effizienter als der Abruf einzelner Wörter, denn es geht schneller, k Wörter in einem Durchgang abzurufen, als k -mal ein Wort. Außerdem ist weniger Verwaltungsaufwand (Overhead) erforderlich, wenn Cache-Einträge mehr als ein Wort umfassen, weil es dann weniger Einträge gibt. Schließlich können viele Computer 64 oder 128 Bit parallel in einem einzigen Buszyklus übertragen, selbst auf 32-Bit-Maschinen.

Das Cache-Design ist bei Hochleistungsprozessoren ein Thema von zunehmender Bedeutung. Ein Aspekt ist die Cache-Größe. Je größer der Cache, umso mehr leistet er, aber umso teurer ist er auch. Ein zweiter Aspekt betrifft die Größe einer Cachezeile. Ein 16-KB-Cache lässt sich in 1024 Zeilen zu je 16 Byte, 2048 Zeilen zu je 8 Byte und in andere Kombinationen aufteilen. Drittens ist in Bezug auf die Organisation des Cache zu klären, wie die Cache-Steuerung verfolgt, auf welche Speicheradressen sich der Cache gerade bezieht. Mit dem Thema Cache beschäftigt sich *Kapitel 4* ausführlich.

Ein vierter Designaspekt hat damit zu tun, ob Befehle und Daten in demselben Cache oder in verschiedenen Caches gehalten werden sollen. Bei einem **gemeinsamen Cache** (Unified Cache; Befehle und Daten liegen im selben Cache) ist das Design einfacher und das Gleichgewicht zwischen Befehls- und Datenaufrufen bleibt automatisch gewahrt. Trotzdem geht heute die Tendenz in Richtung eines **geteilten Cache** (Split-Cache), bei dem die Befehle in den einen Cache und die Daten in den anderen geladen werden. Dieses Design wird auch als **Harvard-Architektur** bezeichnet und geht zurück auf Howard Aikens Mark-III-Computer, der über unterschiedliche Speicher für Befehle und Daten verfügte. Was die Entwickler in diese Richtung drängt, ist die weite Verbreitung von Pipeline-Prozessoren. Die Befehlsleseeinheit muss zur selben Zeit auf Befehle zugreifen, in der die Operandenleseeinheit die Daten benötigt. Der geteilte Cache erlaubt parallele Zugriffe, was bei einem gemeinsamen Cache nicht möglich wäre. Und da Befehle normalerweise während der Ausführung nicht modifiziert werden, muss der Inhalt des Befehlscache niemals in den Speicher zurückgeschrieben werden.

Ein fünfter Punkt ist die Anzahl der Cache-Speicher. Nicht ungewöhnlich sind heute Chips mit einem internen Primär-Cache und einem externen, aber im gleichen Gehäuse befindlichen Sekundär-Cache, sowie mit einem dritten, noch etwas weiter entfernt liegenden Cache.

2.2.6 Speichermodule und -typen

Von den ersten Tagen der Halbleiterspeicher bis Anfang der 1990er Jahre wurden Speicher als einzelne Chips produziert, gekauft und installiert. Die Speicherdichte stieg von 1 Kbit auf 1 Mbit und mehr pro Chip. Jeder Chip war aber eine separate Komponente. Die ersten Personalcomputer hatten oftmals leere Sockel, in die der Käufer zusätzliche Speicherchips stecken konnte, falls er sie überhaupt brauchte.

Seit Anfang der 1990er Jahre bedient man sich einer anderen Anordnung. Mehrere Chips, üblicherweise 8 oder 16, werden auf eine kleine Leiterplatte montiert und als Baugruppe – Speichermodul – verkauft. Abhängig davon, ob sie nur auf einer oder aber auf beiden Seiten der Leiterplatte einen Kontaktstreifen aufweisen, werden sie als **SIMM (Single Inline Memory Module)** oder **DIMM (Dual Inline Memory Module)** bezeichnet. SIMM-Module besitzen eine Kontaktreihe mit 72 Kontakten (Pins) und übertragen 32 Bit je Taktzyklus. Heutzutage sind sie kaum noch im Einsatz. DIMM-Module weisen gewöhnlich auf beiden Seiten der Leiterplatte Kontaktreihen mit je 120 Pins (insgesamt 240 Pins) auf und übertragen 64 Bit je Taktzyklus. Derzeit üblich sind DIMMs vom Typ DDR3, die die dritte Version der Speicher mit doppelter Datenrate verkörpern.
► Abbildung 2.14 zeigt ein typisches DIMM-Modul.

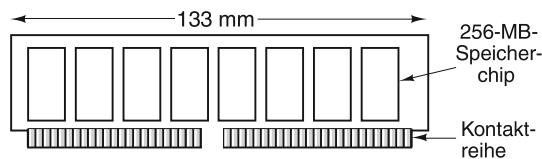


Abbildung 2.14: Draufsicht auf ein DIMM-Speichermodul mit 8 Chips zu je 256 MB auf jeder Seite, d.h. einer Gesamtkapazität von 4 GB. Die andere Seite sieht genauso aus.