Vorwort

Die Digitaltechnik ist heute allgegenwärtig. Wir finden sie als eine Basistechnologie für ein großes Spektrum von informationstechnischen Anwendungen in nahezu allen Lebensbereichen. In der Digitaltechnik kommen viele verschiedene Disziplinen zusammen – naturwissenschaftliche wie die Physik, geisteswissenschaftliche wie die Mathematik, ingenieurwissenschaftliche wie die Elektrotechnik, und schließlich die Informatik, die Anteile aus all diesen Disziplinen vereinigt. An den Schnittstellen dieser Gebiete sind die Forschungsaktivitäten besonders lebendig – dies ist einer der Gründe für den rasanten Fortschritt in der Digitaltechnik.
Die thematische Breite ihrer Wurzeln macht die Digitaltechnik zu einem Grundlagenfach der verschiedensten Studiengänge an Universitäten und Fachhochschulen. Die Lebendigkeit des Gebietes bringt es mit sich, dass das bestehende Lehrmaterial immer wieder überarbeitet und ergänzt werden muss. Allerdings ist es nicht einfach, ein gutes Grundlagenlehrbuch zu schreiben. Zum einen liegt das an der schieren Größe des Gebietes. Wo soll man Schwerpunkte setzen, wo vertiefen, ohne die Darstellung zu überfrachten? Worauf kann man andererseits in einem Einführungsbuch verzichten? Und: wie berücksichtigt man den stetigen Wandel; welche Themen werden in fünf Jahren noch relevant sein? Zudem gilt es, ein ausgewogenes Verhältnis von theoretischen Grundlagen und Bezug zur Praxis zu finden.
All diese Abwägungen sind in dem vorliegenden Buch sorgfältig berücksichtigt worden. Das Buch vermittelt den Studierenden sowohl einen breiten Überblick über das Fachgebiet als auch eine fundierte Ausbildung in den wesentlichen theoretischen Grundlagen. Dabei wendet sich der Blick immer wieder auf moderne Entwicklungen im Bereich des Entwurfs informationstechnischer Systeme. Bekanntlich entwirft heutzutage kein Schaltungsentwickler seine Hardware noch von Hand. Stattdessen verwendet er Methoden des *Computer-Aided Design* (CAD), wofür leistungsfähige Softwarewerkzeuge zur Verfügung stehen. Der heutige industrielle Entwurfsprozess ähnelt in weiten Teilen eher dem Softwaredesign als der klassischen elektronischen Schaltungsentwicklung. Durch die damit einhergehende Automatisierung wurde die Produktivität des Entwurfsprozesses in den letzten Jahren drastisch erhöht und wird durch ständige Innovation immer weiter gesteigert. Um von diesen aktuellen Entwicklungen maximal profitieren zu können, ist es für einen Informa-

tiker oder Ingenieur wichtig zu verstehen, was „hinter den Kulissen“ solcher CAD-Systeme abläuft. Die Autoren vermitteln dieses Hintergrundwissen, indem sie die wichtigen Grundlagen der theoretischen Informatik in Beziehung setzen zu den Fragestellungen des rechnergestützten Entwurfs. Ein späterer „Designer“ digitaler Systeme wird zwar in der industriellen Praxis selbst keine Entwurfsalgorithmen entwickeln. Dennoch versetzt ihn erst eine fundierte Vorbildung in die Lage, existierende CAD-Systeme produktiv einzusetzen, ihre Ergebnisse richtig zu interpretieren und die richtigen Designentscheidungen zu treffen.
Der konsequente Bezug zur Praxis zieht sich wie ein roter Faden durch alle Kapitel des Buches. Erläuterungen theoretischer Konzepte wie etwa der Booleschen Funktionen werden sofort um konkrete Anwendungen ergänzt, indem beispielsweise aufgezeigt wird, wie in der industriellen Praxis Schaltungen mittels sogenannter Hardwarebeschreibungssprachen entworfen werden. Die Autoren haben dazu die weit verbreitete Sprache Verilog gewählt und benutzen sie im gesamten Buch, um Beispiele behandelter Schaltungskonzepte vorzustellen. Die Leser werden angehalten, mit Hilfe frei erhältlicher CAD-Tools selbst praktische Erfahrungen zu sammeln. Gerade durch die Wahl der Sprache Verilog, die viele Ähnlichkeiten zu höheren Programmiersprachen wie Java aufweist, werden die Gemeinsamkeiten – aber auch die Unterschiede – des modernen Hardwareentwurfs im Vergleich zum Softwareentwurf für die Studierenden besonders deutlich.
Das vorliegende Buch gehört zu den wenigen Einführungen, die dem Thema der *Verifikation* digitaler Systeme ein eigenes Kapitel widmen. Kein anderes Thema spielt in der Praxis derzeit eine größere Rolle. In der Industrie entfallen heute bis zu 80% der Entwurfskosten für ein *System-on-Chip* auf die Verifikation, also auf das Problem zu entscheiden, ob ein Systementwurf alle funktionalen Anforderungen erfüllt. Diese Situation spiegelt sich sogar in den Personalentscheidungen der Firmen wider: Für den Schaltungsentwurf werden heute genauso viele Verifikationsingenieure wie „klassische“ Designer gebraucht. Die Autoren tragen dieser Tatsache Rechnung. Es werden sowohl wichtige theoretische Grundlagen gelegt, als auch der Bezug zur Praxis anhand von CAD-Werkzeugen hergestellt, die für die Studierenden frei erhältlich sind.
Das Buch schließt mit dem beispielhaften Entwurf eines Prozessors. Hier kommt nochmals alles Gelernte zum Einsatz. Die Wahl der Befehlssatzarchitektur der Intel x86-Prozessorfamilie ist für die Studierenden von besonderem Reiz, denn sie können die hierfür erstellten Programme auch auf dem eigenen PC laufen lassen. Es lässt sich nämlich die gesamte für die x86-Architektur verfügbare „Toolchain“ (Assembler, Debugger, etc.) anwenden. Für den Lernenden – gerade in den neuen Bachelorstudiengängen der Infor-

matik, Elektrotechnik und Informationstechnik – entsteht so eine fundierte und umfassende Sicht auf Architekturen und Methodik der Digitaltechnik, die sowohl die Grundlagen für ein weiterführendes Studium als auch für die berufliche Praxis liefert.

TU Kaiserslautern, *Wolfgang Kunz*
Dezember 2007

4 Arithmetik

Die in den vorhergehenden Kapiteln vorgestellten Schaltungen haben ausschließlich einfache, Boole'sche Signale verarbeitet. In diesem Kapitel wird nun erklärt, wie Prozessoren mit Zahlen umgehen. Wir stellen verschiedene arithmetische Schaltungen vor, die Operationen wie Addition, Subtraktion und Multiplikation ermöglichen.

4.1 Zahlendarstellungen

Anders als beim Durchführen von Berechnungen auf Papier können wir in der Digitaltechnik nicht mit beliebig großen Zahlen rechnen: Der Spielraum ist durch die Hardware eingeschränkt. In diesem Abschnitt beschäftigen wir uns mit der Frage, wie man Zahlen möglichst effizient in Schaltungen darstellen und verarbeiten kann.

Dieses Wissen ist nicht nur für die Schaltungstechnik relevant. So ist z. B. die Explosion der Ariane 5 auf einen Softwarefehler zurückzuführen, der aus der inkorrekten Konvertierung einer 64-Bit-Fließkommazahl in eine vorzeichenbehaftete 16-Bit-Ganzzahl resultierte: Es kam zu einem *Overflow.* Eine allzu abstrakte Betrachtungsweise von Computersystemen kann also zu Fehlern in der Software führen.

4.1.1 Zahlensysteme

Computer speichern Informationen im Arbeitsspeicher oder in Prozessorregistern. Diese wiederum sind aus Flipflops oder konzeptuell ähnlichen Bauteilen (siehe Kapitel 6) zusammengesetzt, welche lediglich „Ja"/„Nein"-Informationen speichern können. Die kleinste Einheit von „Information" in einem Computer ist ein „Bit". Alle im Computer verarbeiteten Daten setzen sich aus diesen Bits zusammen, werden also intern *binär* dargestellt.

Wenn wir von einem binären Zahlensystem sprechen, meinen wir ein Zahlensystem mit der *Basis* 2, also ein System, das zwei verschiedene Symbole zur Darstellung von Zahlen verwendet. Dieses Konzept können wir verallgemeinern: Ein Zahlensystem mit der Basis r verwendet r verschiedene Symbole.

Gängige Zahlensysteme sind

- die bereits erwähnten binären Zahlen,
- oktale Zahlen, mit den Symbolen 0 bis 7,
- dezimale Zahlen, mit denen wir tagtäglich im Supermarkt konfrontiert werden,
- hexadezimale Zahlen, mit den Symbolen 0 bis 9 und A bis F.

In manchen Fällen kann es hier zu Verwechslungen kommen. So ist unklar, ob 107 eine oktale, dezimale oder hexadezimale Zahl ist (wir können lediglich ausschließen, dass es sich um eine Binärzahl handelt). In diesem Fall verwenden wir die Notation 107_8, um klarzustellen, dass es sich um eine oktale Zahl handelt.

Um eine Zahl mit der Basis r in das dezimale Zahlensystem umzuwandeln, gehen wir wie folgt vor: Jede Stelle i der Zahl wird mit r^i multipliziert, wobei

Warum Basis 2 und nicht Basis 10?

Praktisch alle modernen Computersysteme verwenden Binärzahlen zur Darstellung der Daten, also ein Zahlensystem mit der Basis 2. Warum ist das so? Wäre es nicht effizienter, zwischen mehr als zwei Spannungswerten zu unterscheiden (z. B. 10), um so mit einem „10er-Bit“ zehn verschiedene Werte darstellen zu können?

Die Entscheidung für die Basis 2 ist keineswegs willkürlich: Eine Erklärung dafür finden wir in der *Informationstheorie.* Betrachten wir eine Zahlendarstellung mit l Stellen und jeweils r möglichen Werten pro Stelle. Somit kann man insgesamt r^l verschiedene Zahlen darstellen. Den Hardwareaufwand, der notwendig ist, um diese Zahl im Speicher unseres Computers abzulegen, können wir grob mit $K(r) = l \cdot r$ abschätzen.

Wenn wir c Zahlen darstellen wollen, so folgt daraus $r^l = c$ und $l = \log_r c$. Entsprechend ersetzen wir l in unserer Kostenfunktion und erhalten

$$K(r) = r \cdot \log_r c \, .$$

Wir leiten nun die Kosten nach r ab, um den Wert von r zu berechnen, der die geringsten Kosten verursacht:

$$\frac{dK}{dr} = \frac{d}{dr}\left(r \cdot \log_r c\right) = \frac{d}{dr}\left(r \cdot \frac{\ln c}{\ln r}\right) = \ln c \cdot \frac{\ln(r) - (\ln(r))' \cdot r}{\ln^2 r}$$

Da $(\ln(r))' = \frac{1}{r}$, wird die Ableitung 0, wenn $\ln(r) - 1 = 0$, das heißt $r = e = 2,718$. Durch nochmaliges Ableiten können wir sicherstellen, dass es sich tatsächlich um ein Minimum handelt.

Da in der Praxis kein Bauteil mit einer irrationalen Basis produziert werden kann, muss man sich für eine Basis von zwei oder drei entscheiden. In der Schaltungstechnik sprechen viele Argumente (z. B. die in Kapitel 2 erklärten Konzepte) dafür, die Basis zwei anstatt drei zu verwenden.

i die *Wertigkeit* der Stelle ist. Eine dezimale Zahl erhalten wir schließlich, indem wir die Ergebnisse dieser Operationen aufsummieren:

$$\langle d_{n-1} d_{n-2} \dots d_1 d_0 \rangle := \sum_{i=0}^{n-1} d_i \cdot r^i$$

Beispiel 36 Die Binärzahl 101010 entspricht der dezimalen Zahl **36**

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 42_{10} .$$

Für die Oktalzahl $637,2_8$ errechnen wir den dezimalen Wert

$$6 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 + 2 \cdot 8^{-1} = 6 \cdot 64 + 3 \cdot 8 + 7 \cdot 1 + \frac{2}{8} = 415,25_{10} .$$

Die hexadezimale Zahl $F1$ entspricht der dezimalen Zahl 241:

$$F1_{16} = F \cdot 16^1 + 1 \cdot 16^0 = 15 \cdot 16 + 1 \cdot 1 = 241_{10}$$

Wir sehen also, dass es auch durchaus möglich ist, Bruchzahlen in Oktal- oder Binärkodierung darzustellen. Für uns wird es relevant sein, Dezimalzahlen in andere Zahlensysteme umzuwandeln. Dazu teilen wir die Dezimalzahl in den Ganzzahlanteil und die Nachkommastellen auf. Die beiden Teile werden dann separat umgewandelt: Der Ganzzahlanteil wird sukzessiv durch die Basis r des Zielsystems dividiert; der *Rest* entspricht der jeweiligen Stelle der Zahl mit Basis r. Die Nachkommastellen erhält man analog durch Multiplizieren; die Ganzzahlwerte des Ergebnisses der Multiplikation ergeben dann die jeweilige Nachkommastelle.

Beispiel 37 Wir wollen die Dezimalzahl $47,625$ in eine Binärzahl umwandeln. Dazu teilen wir sie in den Ganzzahlanteil 47 und die Nachkommastellen 0,625 auf. Den Ganzzahlanteil der Binärzahl erhalten wir durch sukzessives Dividieren. Analog erhalten wir die Nachkommastellen durch Multiplizieren (siehe Abbildung 4.1): Die Zahl 0,625 wird mit 2 multipliziert. Der Ganzzahlanteil 1 des Ergebnisses dieser Multiplikation entspricht der ersten Nachkommastelle der Binärzahl. Die Nachkommastellen dieses Ergebnisses werden wieder mit 2 multipliziert. Dieser Prozess wird wiederholt, bis die gewünschte Genauigkeit erreicht ist. In unserem Beispiel ist das Ergebnis exakt, da alle Stellen nach der dritten Nachkommastelle 0 sind. **37**

Die Umwandlung zwischen binären, oktalen und hexadezimalen Zahlen ist bedeutend einfacher (jedoch kann man selbstverständlich auch die oben an-

47	
23	1
11	1
5	1
2	1
1	0
0	1

$$\begin{array}{r} 0{,}625 \\ \cdot 2 \\ \hline 1{,}250 \\ \cdot 2 \\ \hline 0{,}500 \\ \cdot 2 \\ \hline 1{,}000 \end{array}$$

$47_{10} = 101111_2$ $\qquad$ $0,625_{10} = 0.101_2$

Abbildung 4.1. Umwandeln der Dezimalzahl 47,625 in eine Binärzahl

$$\overbrace{1\,0\,1\,1}^{B}\,\overbrace{1\,1\,1\,0}^{E}\,\overbrace{1\,1\,1\,0}^{E}\,\overbrace{1\,1\,1\,1}^{F}$$

a. Binär zu hexadezimal

$$\underbrace{1}_{1}\;\underbrace{0\,1\,1}_{3}\,\underbrace{1\,1\,1}_{7}\,\underbrace{0\,1\,1}_{3}\,\underbrace{1\,0\,1}_{5}\,\underbrace{1\,1\,1}_{7}$$

b. Binär zu oktal

Abbildung 4.2. Umwandlung zwischen binären, oktalen und hexadezimalen Zahlen

geführte Methode verwenden). Um eine Binärzahl in die hexadezimale Darstellung umzuwandeln, fassen wir jeweils vier Stellen der Binärzahl zusammen und ersetzen sie durch die entsprechende hexadezimale Ziffer (siehe Abbildung 4.2a). So wird zum Beispiel 1010 zu A (entspricht der Dezimalzahl 10). Analog gehen wir für Oktalzahlen vor: Hier ersetzen wir jeweils drei Stellen der Binärzahl durch die entsprechende oktale Ziffer (siehe Abbildung 4.2b).
Aus dem Wunsch heraus, Dezimalzahlen ähnlich einfach in Binärzahlen umwandeln zu können, entstand das BCD-System (wobei BCD eine Abkürzung für binary-coded decimal ist). Dieses System verwendet für die Darstellung jeder Dezimalstelle vier Bits. Offensichtlich werden hier 6 binäre Kombinationen verschwendet: Für die Darstellung von 10 Ziffern würden 3,3219 (entspricht $\operatorname{ld} 10 = \frac{\log 10}{\log 2}$) Bits ausreichen. Dieser Umstand muss entsprechend bei den arithmetischen Operationen beachtet werden. Wir gehen hier nicht näher auf diese Darstellung ein und verweisen den interessierten Leser auf einschlägige Literatur (z. B. [Man93]).

4.1.2 Darstellung negativer Zahlen: Komplemente

Nachdem wir nun wissen, wie die natürlichen Zahlen (einschließlich 0) $\mathbb{N}_0$ und positive rationale[1] Zahlen ($\mathbb{Q}_+$) als Binärzahlen darstellen können, stellt sich die Frage, wie wir *negative Zahlen* behandeln sollen. Der einfachste Ansatz ist, das Vorzeichen einfach mithilfe des höchstwertigen (in unserer Darstel-

[1] Nicht zu verwechseln mit den reellen Zahlen $\mathbb{R}$ – diese können nämlich aus offensichtlichen Gründen *nicht* als endlich lange Binärzahl dargestellt werden.

lung ganz linken) Bits darzustellen und die restlichen Bits zu verwenden, um die Zahlen wie in Kapitel 4.1.1 beschrieben darzustellen. Diese Darstellung kommt der Darstellung von negativen Zahlen im Dezimalsystem am nächsten. Sie hat allerdings den Nachteil, dass Addition und Subtraktion gesondert behandelt werden müssen (erinnern Sie sich an die Algorithmen zur Addition und Subtraktion, die Ihnen in der Schule beigebracht wurden).
In der Digitaltechnik greift man daher auf eine Darstellung zurück, die es erlaubt, denselben Algorithmus sowohl für Addition als auch Subtraktion zu verwenden. Es handelt sich hierbei um die sogenannte *Komplementdarstellung.* Man unterscheidet zwischen dem r-Komplement und dem $(r-1)$-Komplement, wobei r für die Basis des Zahlensystems steht. Wir werden uns in diesem Buch ausschließlich mit dem 2er- und 1er-Komplement beschäftigen.

Definition 4.1 ($(r-1)$-Komplement, 1er-Komplement) Das $(r-1)$-Komplement einer Zahl N mit der Basis r und n Stellen ist **4.1**

$$(r^n - 1) - N \; .$$

Zur Berechnung geht man wie folgt vor: r^n bezeichnet eine $(n+1)$-stellige Zahl, die aus einer 1 gefolgt von n Nullen besteht (also z. B. $10^4 = 10000$ für die Dezimalzahlen). Zieht man nun 1 von dieser Zahl ab, so erhält man eine n-stellige Zahl, deren Stellen allesamt der höchsten Ziffer des Zahlensystems entsprechen (also 9999 in unserem Beispiel). Das 9er-Komplement von 3123 ist also 6876.
Im Falle der Binärzahlen ist die Basis $r = 2$, und $r^n - 1$ ist die n-stellige Zahl, bestehend aus lauter Einsen. Die Subtraktion ist in diesem Fall sehr einfach: Wenn im Subtrahend an der i-ten Stelle eine Eins steht, so steht im Ergebnis der Subtraktion an dieser Stelle eine Null (da im Minuend an jeder Stelle 1 steht). Steht im Subtrahend an der Stelle i eine 0, so ist die entsprechende Stelle im Ergebnis 1. Zur Berechnung des *1er-Komplements* invertieren wir also einfach jede Stelle der Binärzahl.
Die $(n+1)$-stellige Binärzahl $s_n d_{n-1} \ldots d_0$ im Einerkomplement entspricht also

- der positiven Dezimalzahl $\sum_{i=0}^{n-1} d_i \cdot 2^i$, falls $s_n = 0$.
- der negativen Dezimalzahl $-\sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i$, falls $s_n = 1$.

Um eine negative Zahl darzustellen, berechnen wir das Komplement dieser Zahl. Wir benötigen nach wir vor ein Bit, um zu vermerken, ob es sich um eine positive oder negative Zahl handelt. Um wieder den ursprünglichen Wert herzustellen, müssen wir die Zahl nochmals komplementieren. Hierbei offen-

7 Ein einfacher CISC-Prozessor

In diesem Kapitel wird ein einfacher Prozessor vorgestellt. Die Architektur, die wir implementieren, wurde von R. Bryant und D. O'Hallaron entworfen und verwendet eine Untermenge der Befehle der IA32-Architektur. Wir geben zunächst eine Architekturspezifikation in Form eines C++ Modells an. Anschließend wird die Architektur in Verilog implementiert.

7.1 Die Y86 Instruction Set Architecture

Frei programmierbare Mikroprozessoren sind sicherlich die wichtigste Anwendung der Digitaltechnik. Sie stellen die Schnittstelle zwischen der Hardware- und der Softwarewelt dar. Es gibt eine Vielzahl an Architekturen und Modellen. Im Server- und Desktopbereich hat sich die 32-Bit Intel Architektur (IA32) durchgesetzt. Entsprechende Prozessoren werden aber auch von anderen Anbietern hergestellt, etwa von AMD.

Bei der IA32-Architektur handelt es sich um eine CISC-Architektur (Complex Instruction Set Computer), was darauf hinweist, dass die Architektur eine große Anzahl von Mikroprozessorbefehlen bietet. In der Tat findet sich in den Handbüchern von Intel eine Liste von etwa 300 Befehlen mit einer Vielzahl an Optionen. Die Architektur, die wir implementieren, wurde von R. Bryant und D. O'Hallaron zu pädagogischen Zwecken entworfen und ist auf eine sehr kleine Auswahl dieser Liste beschränkt. Sie wird als „Y86"-Architektur bezeichnet, in Anlehnung daran, dass die Produktnummern der ersten entsprechenden Intel-Prozessoren auf „86" endeten.

7.1.1 Befehle, Register und Speicher

Wir gehen davon aus, dass der Leser die Grundlagen der sequenziellen Programmierung beherrscht. In höheren Programmiersprachen, wie C oder Java, werden alle Daten in Variablen abgelegt, wobei jeder Variablen eine Speicherzelle zugeordnet ist. Die Speicherzellen sind nummeriert. Die „Nummer" einer Speicherzelle wird als *Adresse* bezeichnet. Wir gehen davon aus, dass jede Speicherzelle genau ein Byte (also 8 Bit) speichert.

Wir schreiben `Mem[ea]` für den Inhalt der Speicherzelle mit der Adresse `ea` (`ea` steht für *effective address*). Sowohl die Befehle als auch die Daten, mit denen das Programm arbeitet, werden im gleichen Speicher hinterlegt (Abbildung 7.1). Eine derartige Architektur wird als *von-Neumann-Architektur* bezeichnet.

Unser Prozessor verwendet einen gemeinsamen *Datenbus*, um auf den Speicher und die I/O-Geräte zuzugreifen (siehe Abschnitt 6.4). Wir nehmen an,

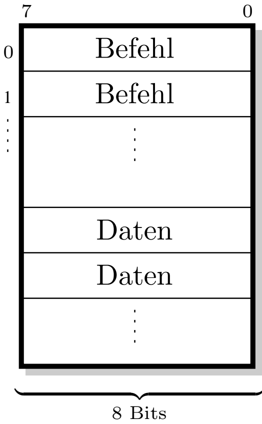

Abbildung 7.1. Schematische Darstellung des Speichers: Die Befehle und Daten werden in einem gemeinsamen Adressraum abgelegt

dass unser Prozessor die einzige Komponente ist, die diesen Bus steuert (Abbildung 7.2). Wir nennen diesen Bus *Systembus* (engl. system bus).

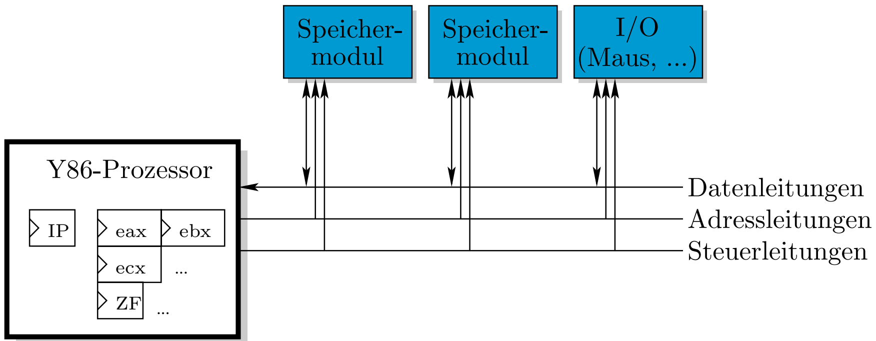

Abbildung 7.2. Der Y86-Prozessor und die anderen Systemkomponenten

Die Register

Da der Zugriff auf den Speicher über den Systembus mehrere Takte dauern kann und somit vergleichsweise langsam ist, können Datenworte im Prozessor in *Registern* zwischengespeichert werden. Diese Register sind in einer höheren Programmiersprache nicht direkt sichtbar. Es ist die Aufgabe des Compilers, die Register bei der Übersetzung des Programms effizient zu nutzen.
Die Y86-Architektur hat acht Datenregister mit jeweils 32 Bit, die wie folgt bezeichnet werden:

Index	0	1	2	3	4	5	6	7
Name	**eax**	**ecx**	**edx**	**ebx**	**esp**	**ebp**	**esi**	**edi**

Die ersten vier Register (**eax** bis **ebx**) dienen dazu, Daten zu halten.[1] Die Register **esp** und **ebp** werden im Normalfall verwendet, um einen *Stack* zu implementieren und werden daher *Stackregister* genannt. Da sie üblicherweise eine Adresse enthalten, werden sie auch *Zeigerregister* genannt. Die Register **esi** und **edi** sind für Array-Indizes gedacht.

Der Instruction Pointer (IP)

Damit der Prozessor auch weiß, wo im Speicher der nächste Befehl liegt, den er ausführen soll, gibt es ein weiteres Zeigerregister, das auf die Adresse des nächsten Befehls zeigt. Dieser Zeiger wird *Instruction Pointer* (IP) genannt. Sprunganweisungen verschieben diesen Zeiger, um das Programm an einer anderen Stelle fortzuführen. Befehle, die keine Sprunganweisung sind, ändern den IP ebenfalls: Der Zeiger wird einfach auf die Adresse der nächsten Instruktion des Programms gesetzt.
Die Befehle, die unser einfacher Prozessor ausführen kann, sind in drei Kategorien eingeteilt:

1. Die *arithmetischen und logischen Befehle* führen Berechnungen wie z. B. die Addition aus.
2. Die *Datentransfer-Befehle* ermöglichen es, Daten aus dem Prozessor über den Systembus in einem RAM-Modul abzulegen und wieder einzulesen. Diese Befehle werden auch zur Kommunikation mit den I/O-Geräten verwendet (sog. *Memory-mapped I/O*).
3. Die *Sprungbefehle* steuern den Kontrollfluss. Es handelt sich also um Befehle, die (möglicherweise bedingt) zu anderen Befehlen springen.

Zusätzlich zu den Datenregistern und dem Instruction Pointer verwendet die Y86-Architektur *Flag-Register*. Die Flag-Register speichern weitere Informationen über das Ergebnis des jeweils letzten arithmetischen oder logischen Befehls. Ein Flag-Register ist genau ein Bit breit. Wir betrachten zunächst nur ein einziges Flag-Register: Das Zero-Flag (**ZF**) wird gesetzt, wenn das Ergebnis der letzten arithmetischen Operation null (zero) war. Es wird als Bedingung für Sprungbefehle verwendet.

Die Befehle der Y86-Architektur

Bevor wir die einzelnen Befehle spezifizieren, gehen wir zuerst auf deren Kodierung ein. Jeder Befehl entspricht einer Folge von Bytes. Die Länge dieser Folge hängt von dem jeweiligen Befehl ab – einfachere Befehle benötigen weniger Platz. Dadurch soll erreicht werden, dass der Programmcode insgesamt

---

[1] Man beachte die etwas sonderbare, nicht-alphabetische Reihenfolge der Register.

kleiner wird.[2] Der Befehl, der ausgeführt werden soll, wird durch den Wert des ersten Bytes der Folge bestimmt. Dieses Byte wird daher auch als *Opcode* bezeichnet.

Da Zahlen schwer zu merken sind, wird der Opcode im Assembler-Programm durch ein *Mnemonic* (eine kurze Zeichenfolge) ersetzt. Ein Programm, das eine solche Textdatei wieder in Zahlen (Maschinencode) übersetzt, wird als *Assembler* bezeichnet.

Die Befehle, die wir implementieren, sind in Abbildung 7.3 zusammengefasst. Alle Befehle erhöhen den IP um die Länge des jeweiligen Befehls.

1. Die Befehle **add** und **sub** führen eine Addition bzw. Subtraktion durch. Im Befehl sind zwei Register kodiert, genannt RD (Destination) und RS (Source). Die Summe RD+RS (bzw. die Differenz RD-RS) wird in RD abgelegt. Falls der neue Wert von RD null ist, wird das Flag **ZF** gesetzt, und ansonsten gelöscht.
2. Der Befehl **RRmov** (kurz für Register-Register **mov**)[3] kopiert den Wert des Registers RS in das Register RD.
3. Der Befehl **RMmov** (kurz für Register-Memory **mov**) kopiert den Wert des Registers RS in den Speicher. Die Adresse (`ea`) wird dabei als Summe des Registers **esi** und des 8-Bit *Displacements* (Byte 3) berechnet:

   $$\texttt{ea} = \texttt{esi} + \text{Displacement}$$

   Beachten Sie, dass sich die Befehle **RRmov** und **RMmov** nur durch die oberen Bits des zweiten Bytes unterscheiden lassen. Dieses Feld wird als `mod` (wie Modifier) bezeichnet.
4. Der Befehl **MRmov** (kurz für Memory-Register **mov**) kopiert einen Wert aus dem Speicher in das Register RS. Die Adresse (`ea`) wird dabei genau wie beim Befehl **RMmov** berechnet.
5. Der Befehl **jnz** addiert den im Befehl angegebenen Offset (genannt *Distance*) zum Instruction Pointer, falls das Flag **ZF** nicht gesetzt ist (das Ergebnis der arithmetischen Operation also nicht null war).
6. Der Befehl **hlt** stoppt die Ausführung des Programms.

Die Kodierung und die Bedeutung dieser Befehle ist in Abbildung 7.3 zusammengefasst. Die Notation R1 $\leftarrow$ R2 ist wie folgt zu lesen: Der Wert in Register R2 wird in das Register R1 kopiert. Die im rechten Teil der Abbil-

---

[2] Andere Architekturen, wie z. B. die MIPS-Architektur, verwenden Befehlsworte, die immer gleich lang sind, was die Decodierung vereinfacht.

[3] Wir verwenden eine Langform der Befehlsnamen, um die Art der Operation klarzustellen. Ein Assembler kann die Unterscheidung anhand der Operanden durchführen und generiert den passenden Opcode.

Mnemonic	Bedeutung	Opcode		
			7 6 3 0	
add	RD←RD+RS	01	11 ⁞ RS ⁞ RD	
sub	RD←RD-RS	29	11 ⁞ RS ⁞ RD	
jnz	if(¬ZF) IP←IP+Distance	75	Distance	
RRmov	RD←RS	89	11 ⁞ RS ⁞ RD	
RMmov	MEM[ea]←RS	89	01 ⁞ RS ⁞ 110	Displacement
MRmov	RS←MEM[ea]	8b	01 ⁞ RS ⁞ 110	Displacement
hlt		f4		

Abbildung 7.3. Die Befehle unseres Y86-Prozessors

dung dargestellten Opcodes sind in hexadezimaler Schreibweise notiert. Bei den in den Befehlen enthaltenen Offsets ist zu beachten, dass es sich um vorzeichenbehaftete Zahlen handelt (im Zweierkomplement). Im Falle des **jnz** Befehls bewirkt eine negative Zahl einen Rückwärtssprung.

> Der Offset bei Sprungbefehlen bezieht sich auf den Befehl *nach* dem Sprungbefehl, da der IP ja schon um zwei erhöht worden ist.

7.1.2 Ein kleines Assembler-Beispiel

Mit den oben angeführten Befehlen ist es schon möglich, ein erstes Assembler-Programm zu schreiben, das von einem Y86-Prozessor ausgeführt werden kann. Da die Kodierung der Befehle unseres Y86-Prozessors mit denen der Intel-Prozessoren übereinstimmt, können die Programme auch auf jedem IA32-Prozessor ausgeführt werden. Umgekehrt klappt dies in der Regel nicht: Von Compilern generierte IA32-Programme verwenden viele zusätzliche Befehle, die unsere Y86-Architektur nicht kennt.
Ein Assembler-Befehl wie

sub R1, R2

ist wie folgt zu lesen: Der Inhalt des Registers R2 wird von dem Inhalt des Registers R1 abgezogen. Das Resultat wird schließlich im Zielregister R1 gespeichert.