T E C H N I S C H E U N I V E R S I T Ä T B E R L I N

Fakultät IV — Institut für Telekommunikationssysteme

Fachgebiet Nachrichtenübertragung — Prof. Dr.-Ing. Thomas Sikora

Einführung in Aufbau und Funktionsweise von Mikroprozessoren

Semesterarbeit

Christian Richter

<uni@ch-r.de>

(Matr. 192548)

Berlin, 14. Juli 2005

Zusammenfassung

Mikroprozessoren sind inzwischen geradezu allgegenwärtig und aus der heutigen Welt nicht mehr wegzudenken. Auf fast jedem Schreibtisch steht ein Computer, dessen Herz ein solcher Prozessor ist. Und auch in Autos, Telefonen, Kaffeemaschinen und anderen Alltagsgeräten tun sie – als sogenannte *Eingebettete Systeme* – ihren Dienst.

Die vorliegende Ausarbeitung soll einen kurzen und dennoch möglichst vollständigen Einblick in die Welt der Mikroprozessoren liefern. Besonderer Wert wurde darauf gelegt, daß auch Leser mit anderen Ausrichtungen als der Elektrotechnik Nutzen daraus ziehen können. Aus diesem Grund (und natürlich aus Platzgründen) wurde auch bewußt darauf verzichtet, die elektrotechnischen Zusammenhänge bis auf die Transistorebene hinab darzulegen. In der Literaturliste im Anhang finden sich dazu einige hervorragende Werke, welche die einzelnen Themen in beliebiger Tiefe behandeln.

Das erste Kapitel widmet sich der Frage, was ein Mikroprozessor eigentlich ist, was er tut und welche Möglichkeiten es gibt, die unüberschaubare Vielfalt von Prozessoren etwas zu ordnen. Das zweite und dritte Kapitel behandelt dann den eigentlichen Schwerpunkt dieser Ausarbeitung, den Aufbau und die Funktionsweise eines Prozessors. Zum Schluß wird in Kapitel vier noch ein kurzer Blick auf zukünftige Technologien riskiert.

Inhaltsverzeichnis

1 Was ist ein Mikroprozessor? 4

1.1 Aufgaben 5

1.2 Möglichkeiten der Einteilung 6

1.2.1 Harvard vs. von Neumann 7

1.2.2 RISC vs. CISC 8

1.2.3 Universal- vs. Spezialprozessoren 9

2 Aufbau und Funktionsweise 11

2.1 Komponenten eines Prozessors 11

2.1.1 Register 12

2.1.2 Rechenwerk 13

2.1.3 Steuerwerk (Control Unit) 16

2.1.4 Systembus-Interface 19

2.2 Ein beispielhafter Programmablauf 21

2.3 Interrupts 25

3 Optimierungen 28

3.1 Höhere Taktfrequenz – Der einfache Weg 28

3.2 Beschleunigung des Speicherzugriffs 31

3.3 Parallele Ausführung von Befehlen 33

3.3.1 Pipeline-Architektur 33

3.3.2 Superskalare Prozessoren 34

3.3.3 Simultaneous Multi Threading (SMT) 35

3.3.4 Multicore Prozessoren 36

3.4 Spezielle Befehlssatz-Erweiterungen 37

4 Zukünftige Möglichkeiten **38**

4.1 Polymer-Prozessoren 38

4.2 Prozessoren mit supraleitenden Elementen (RSFQ) 39

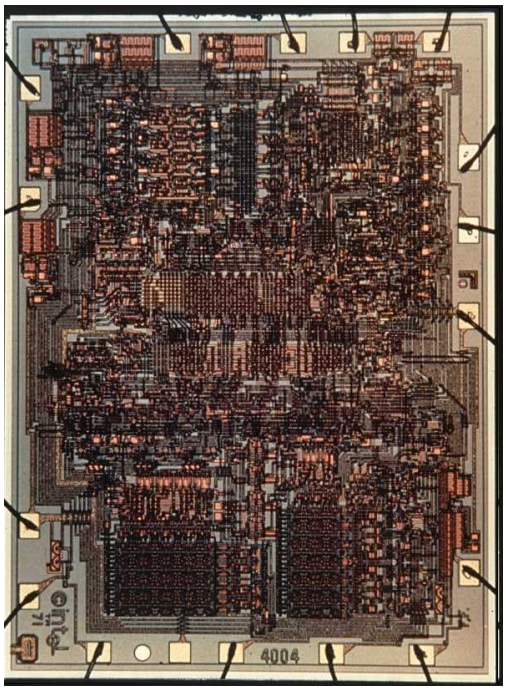
4.3 Weitere Möglichkeiten 40

Literaturverzeichnis **43**

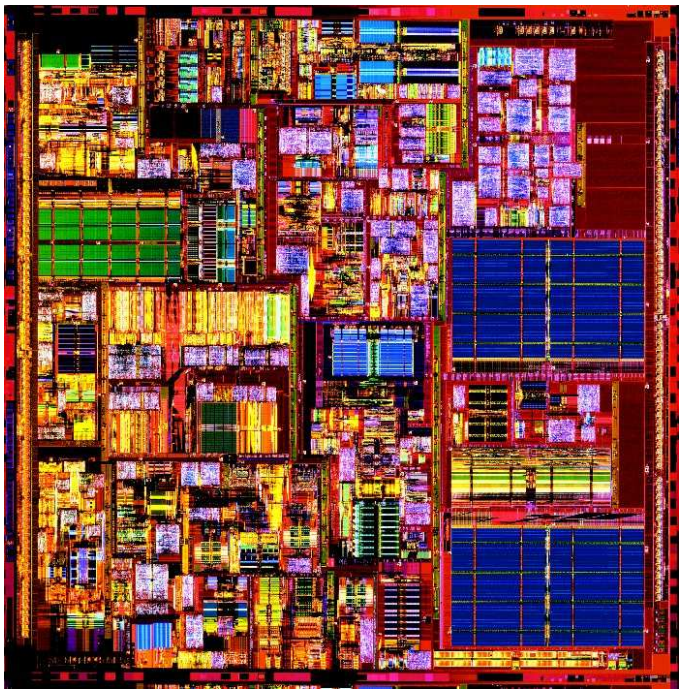
Kapitel 1

Was ist ein Mikroprozessor?

Als *Prozessor* wird eine Schaltung bezeichnet, die Daten entgegen nimmt, diese anhand von (ebenfalls entgegengenommenen) Befehlen verarbeitet und die erhaltenen Ergebnisse wieder ausgibt. Der Vorsatz *Mikro* bezieht sich dabei auf die Größe der verwendeten Strukturen.



(a) 1971: Intel 4004, 2250 Trans.



(b) 2002: Intel Pentium4 "Northwood", 55 Mio. Tran.

Abbildung 1.1: Entwicklungsstufen der Prozessoren **Quelle: [20]**

Während bis Anfang der 70er Jahre Prozessoren noch aus diskreten Bauelementen wie Transistoren, Widerständen und Kondensatoren aufgebaut wurden, gelang es schließlich, alle diese

Elemente als sogenannten Integrierten Schaltkreis (engl. Integrated Circuit, IC) auf einem Siliziumkristall, dem *Die*, unterzubringen. Die Strukturen der einzelnen Bauelemente waren dabei nur wenige Mikrometer groß.

Der in Abbildung 1.1a gezeigte 4004 von Intel, der als erster Mikroprozessor gilt[1], hatte beispielsweise Strukturbreiten von 10 μm. Auch heute noch hält sich der Begriff Mikroprozessor bzw. Mikroelektronik hartnäckig, obwohl wir es inzwischen mit Strukturen im Nanometerbereich zu tun haben. Der Ende 2002 von Intel herausgebrachte Northwood-Kern (Abbildung 1.1b) des Pentium 4 beispielsweise nutzt Strukturbreiten von 130 nm und bringt auf diese Weise 55 Mio. Transistoren auf nur 131mm^2 Chipfläche unter [20]. Aktuelle Prozessoren sind inzwischen dank verbesserter Lithographischer Verfahren bei 90 nm angekommen – mehr als 100 mal kleiner also als die 1971 erreichten 10 μm.

1.1 Aufgaben

Der Mikroprozessor ist das Herzstück eines jeden Rechners. Er wird dort auch als *Central Processing Unit* (CPU) bezeichnet und ist für die Steuerung aller Abläufe und die Verarbeitung der Daten zuständig. Was das im Einzelnen umfaßt, soll dieser Abschnitt verdeutlichen.
Dabei ist es nötig, ein wenig auf den grundlegenden Aufbau eines solchen Rechnersystems einzugehen. Auch wenn es in diesem Aufsatz vor allem um den Prozessor selbst gehen soll, kann dieser nicht völlig losgelöst von seiner Umgebung betrachtet werden.

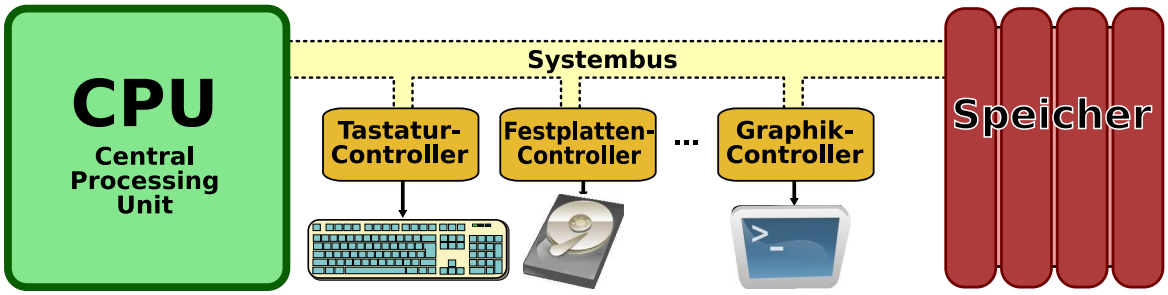


Abbildung 1.2: Prinzipieller Aufbau eines Rechners

In Abbildung 1.2 ist eine solche Umgebung angedeutet. Der Mikroprozessor (CPU) ist über einen Bus, den sogenannten Systembus mit dem Speicher des Systems (meist als RAM realisiert) verbunden. Über diesen Bus sind auch die Peripheriegeräte (Tastatur, Massenspeicher, Display ...) angekoppelt.[2] Natürlich sind diese nicht direkt mit den Steuer- und Datenleitungen des Prozessors verbunden. Zwischen Systembus und dem Peripheriegerät befindet sich ein spezialisierter

[1] Wer tatsächlich den ersten Mikroprozessor entwickelt hat, ist ein Streitpunkt zwischen Texas Instuments und Intel. Tatsache ist, daß TI bereits 1968 einen μP entwickelt hatte. Intels 4004 war aber der erste in Serie gefertigte Prozessor.

[2] In realen Systemen sind oft zusätzliche Umsetzer zwischengeschaltet, die den Systembus (bei PCs als Frontside-Bus (FSB) bezeichnet) auf andere Protokolle (z.B. PCI) umsetzen. Diese sind jedoch sehr architekturspezifisch.

Baustein (oft als Controller bezeichnet), der die Steuersignale des Prozessors gerätespezifisch umsetzt[3].

In dieser allgemeinen Form ist der dargestellte Aufbau prinzipiell für alle Systeme gültig, in denen Prozessoren eingesetzt werden, auch wenn hier als Beispiel typische PC-Komponenten verwendet wurden. In mobilen Geräten wie z.B. Smartphones oder PDAs würde sich nur die Art der Peripheriegeräte etwas unterscheiden. Statt Festplatten- und Tastaturcontrollern kämen dann beispielsweise Bausteine für den Anschluß von Speicherkarten und Touchscreen zum Einsatz. Auch eingebettete Systeme sind ähnlich aufgebaut. Hier sind lediglich viele der angesprochenen Komponenten zusammen mit dem Prozessor auf einem einzigen Chip integriert. Näheres zu den verschiedenen Prozessoren und zu möglichen Einteilungen ist in Abschnitt 1.2 zu finden.

Ganz allgemein ergeben sich also für den Prozessor folgende Aufgaben:

Ansteuerung der Peripheriegeräte-Controller Sowohl die zu verarbeitenden Daten als auch die Programme, die festlegen, was mit den Daten zu geschehen hat, werden über Peripheriegeräte (Tastatur, Scanner, Mikrofon, Kamera ...) entgegengenommen, oder dort abgelegt (Festplatte, optische Laufwerke ...). Die Ausgabe der berechneten Ergebnisse erfolgt ebenfalls über Peripheriegeräte (Display, Lautsprecher, Drucker ...).

Speicherzugriff Sind die Daten und die Programmanweisungen von der Peripherie entgegengenommen worden, werden sie im Hauptspeicher bis zu ihrer Abarbeitung zwischengelagert. Auch die Ergebnisse können dort abgelegt werden.

Verarbeitung der Daten Die Daten werden gemäß der Programminstruktionen durch arithmetische Operationen (Addition, Subtraktion, Multiplikation ...) und logische Operationen (AND, OR, XOR, NOT[4]) miteinander verknüpft.

1.2 Möglichkeiten der Einteilung

Prozessoren sind in der heutigen Zeit nicht nur in Computern und Notebooks zu finden. Sie verbergen sich in nahezu allen Geräten, vom Toaster über den Videorecorder bis hin zu Mobiltelefonen und anderen Kommunikationsgeräten. Dementsprechend vielgestaltig sind sie natürlich auch.

Um etwas Ordnung in die unüberschaubare Vielfalt zu bringen, existieren verschiedene Ansätze, die Prozessoren gemäß solcher Parameter wie Flexibilität, Art der Speichereinteilung oder Umfang des Befehlssatzes zu klassifizieren. Die so gezogenen Grenzen sind selbstverständlich recht unscharf. Es gibt immer wieder Zwischenstufen, die sich in keine oder mehrere Gruppen einordnen lassen. Sie bieten aber zumindest einen groben Anhaltspunkt.

[3] Diese Umsetzung kann durchaus auch sehr komplex sein. Ein Beispiel ist der Display-Controller (die „Grafikkarte"), welche selbst wieder einen Prozessor (GPU) und Speicher enthält.

[4] Eine Erklärung dieser Funktionen befindet sich in Abschnitt 2.1.2.

1.2.1 Harvard vs. von Neumann

Eine ganz klassische Möglichkeit ist die Einteilung in sogenannte *Harvard-* bzw. *von Neumann-*Architekturen. Hier geht es im Wesentlichen darum, ob der Prozessor für Daten und Anweisungen (Programme) getrennte Speicher benutzt (Harvard) oder ob beide gemischt in einem einzigen Speicher untergebracht werden (von Neumann).

von Neumann-Architektur

Die von Neumann-Architektur wurde 1946 von JOHN VON NEUMANN vorgestellt und erlaubte es erstmalig[5], überhaupt unterschiedliche Programme auf Daten anzuwenden. In Abbildung 1.3 ist das Prinzip dargestellt. Sowohl Programme (blau) als auch Daten (grün) liegen in einem gemeinsamen Speicher. An einen unidirektionalen Adreßbus legt der Prozessor die Adresse der gewünschten Speicherzelle an und liest bzw. schreibt dann das Datum über einen bidirektionalen Datenbus aus dem bzw. in den Speicher. Ob gelesen oder geschrieben wird, signalisieren Steuerleitungen.

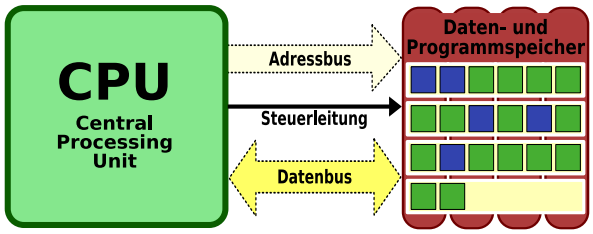


Abbildung 1.3: von Neumann-Architektur

Die Vorteile liegen auf der Hand. Der vorhandene Speicher kann optimal ausgenutzt werden und der Adreß- und Datenbus muß nur einfach ausgeführt werden, was den Aufbau des Rechners vereinfacht.
Diese Beschränkung auf einen einzigen Bus hat natürlich auch Nachteile. Besonders wenn große und kontinuierliche Datenströme zu verarbeiten sind – wie das beispielsweise bei der digitalen Signalverarbeitung der Fall ist – erweist sie sich oft als Flaschenhals. Schließlich müssen nicht nur die kontinuierlich eintreffenden Eingangsdaten gelesen und die Ergebnisse geschrieben, sondern nebenbei auch noch die Programmanweisungen übertragen werden.

Ein weiteres Problem, das sich durch die fehlende Trennung zwischen Programmen und Daten ergibt ist, daß durch fehlerhafte Programmierung eventuell Daten als Anweisungen interpretiert werden können. Häufig geschieht dies durch sogenannte *Buffer Overflows*[6]. Durch geschickte

[5] Teile der von Neumann-Architektur wurden bereits vorher von KONRAD ZUSE beschrieben.

[6] Für Daten – beispielsweise ein übers Netz eintreffendes IP-Paket – wird ein Speicherbereich definierter Größe reserviert. Ist das Paket jedoch größer (und benutzt man für die Erstellung seiner Programme unsichere Sprachen wie C oder C++, die keinerlei Prüfungen vorsehen), wird ein Teil der Daten über den reservierten Bereich hinaus geschrieben und kann unter ungünstigen Umständen als Folge von Anweisungen interpretiert werden.

Manipulation lassen sich so Flüchtigkeitsfehler beim Programmieren dazu ausnutzen, beliebigen Code auf fremden Rechnern auszuführen und so z.B. Daten auszuspähen oder Viren und Würmer zu verbreiten.

Harvard-Architektur

Der bis 1946 von HOWARD AIKEN an der Harvard-Universität entwickelte Computer Mark III verwirklichte ein anderes Konzept, das später als Harvard-Architektur bekannt wurde [19]. Hier stehen getrennte Speicher für Daten und Programmanweisungen zur Verfügung. Jeder Speicher ist einzeln über einen Adreß- und Datenbus angebunden (Abbildung 1.4). Auf diese Weise können auch große Datenströme verarbeitet werden, ohne daß es wegen dem gleichzeitig nötigen Nachladen der Programminstruktionen zu Unterbrechungen kommt. Die Harvard-Architektur wird daher in erster Linie von *Digitalen Signalprozessoren* (DSPs) verwendet.

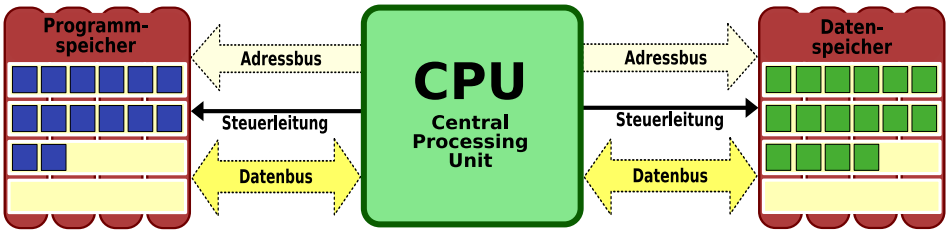


Abbildung 1.4: Harvard-Architektur

Der Nachteil dieser Architektur ist natürlich, daß die Aufteilung des Speichers nicht so flexibel gehandhabt werden kann. Ist das Programm zu groß für den Programmspeicher, kann es nicht verarbeitet werden, auch wenn im Datenspeicher möglicherweise noch Platz wäre. Dieser Nachteil wirkt sich allerdings bei DSPs nicht so stark aus, da hier meist ein relativ kleines, sich selten änderndes Programm verwendet wird, um große Datenmengen zu verarbeiten (siehe auch Abschnitt 1.2.3).

1.2.2 RISC vs. CISC

Der Umfang seines *Befehlssatzes* bietet einen weiteren Ansatzpunkt, um einen Mikroprozessor zu klassifizieren. Unter einem Befehlssatz versteht man die Gesamtheit aller Anweisungen (Programminstruktionen), die ein Prozessor zu verstehen in der Lage ist. Sie werden ihm in *Maschinensprache* (siehe Abschnitt 2.1.3 auf Seite 16) erteilt und umfassen Dinge wie:

- „**Addiere** zu der in Speicherstelle x gespeicherten Zahl den Wert y hinzu",
- „**Springe** zu dem in Speicherstelle x gespeicherten Befehl und fahre dort fort" oder
- „**Überspringe** den folgenden Befehl wenn die in x und y abgelegten Werte gleich sind".

Beim Erstellen eines solchen Befehlssatzes gibt es nun zwei unterschiedliche Philosophien: Der *RISC*-Ansatz (**R**educed **I**nstruction **S**et **C**omputing) verfolgt das Ziel, den Befehlssatz möglichst minimal zu halten. Es gibt wenige, einfache Befehle, die aufgrund ihrer Einfachheit allerdings sehr effizient in Hardware realisiert und so sehr schnell ausgeführt werden können. Um kompliziertere Aufgaben zu lösen, werden mehrere dieser einfachen Befehle nacheinander ausgeführt.

Der CISC-Ansatz (**C**omplex **I**nstruction **S**et **C**omputing) dagegen versucht, einen möglichst vollständigen Satz an Befehlen zur Verfügung zu stellen, so daß für fast jede Aufgabe ein spezieller Befehl zur Verfügung steht. Das ermöglicht kürzere Programme, die aus weniger Instruktionen bestehen. Die Ausführung der einzelnen Instruktionen dauert allerdings länger, da sie im Allgemeinen komplexer sind. Meist sind die einzelnen CISC-Instruktionen selbst kleine Programme (sog. *Mikroprogramme*), die im Prozessor gespeichert sind. Diese Mikroprogramme bestehen wiederum aus einzelnen Instruktionen (Mikro-Befehle oder *µOPs*), die dann so etwas wie RISC-Befehle darstellen und in Hardware ausgeführt werden können.

Reine RISC- bzw CISC-Prozessoren sind selten. In der Realität sind Mischformen dieser beiden Extreme verbreitet, die dann jeweils mehr in die eine oder in die andere Richtung tendieren. Die bekannten *x86*er Prozessoren von Intel bzw. AMD, die heute in den meisten PCs und Notebooks stecken, zählt man zur CISC-Architektur. IBMs PowerPC-Prozessoren, die Apple in seinen Macintosh-Rechnern verbaut[7], gelten als RISC-Prozessoren, obwohl sie durch diverse Befehlssatzerweiterungen[8] inzwischen auch komplexere Instruktionen kennen. Auch die SPARC-Prozessoren, die in Workstations von SUN arbeiten, gehören zur RISC-Familie [6].

Letztendlich geht es bei RISC vs. CISC um die Frage, wo die Komplexität angesiedelt ist. Bei CISC liegt sie im Prozessor, genauer gesagt im Steuerwerk (Abschnitt 2.1.3), welches sehr kompliziert aufgebaut ist und dadurch viel Chipfläche benötigt. Bei RISC dagegen liegt die Komplexität im Compiler, dem Programm das die vom Menschen geschriebenen Programme in Maschinensprache umsetzt.

1.2.3 Universal- vs. Spezialprozessoren

Eher an der Flexibilität ist die Einteilung in Universal- und Spezialprozessoren orientiert. Universalprozessoren, auch oft als *General Purpose Processors* (GPP) bezeichnet, sind frei programmierbar, besitzen einen umfangreichen Satz von Befehlen und können dementsprechend für viele verschiedene Aufgaben eingesetzt werden. Die in PCs, Notebooks und anderen Computern eingesetzten Prozessoren gehören alle zu dieser Klasse.

Auf der anderen Seite stehen die Spezialprozessoren, die weniger flexibel, dafür aber bei bestimmten Aufgaben sehr viel schneller als GPPs sind. Ein Beispiel hierfür sind die Digitalen Signalprozessoren (DSPs). Sie sind auf Multiplikationen und Additionen optimiert, so daß mit ihrer Hilfe z.B. digitale Filter besonders effektiv berechnet werden können. DSPs sind allerdings

[7] Wie Apples CEO Steve Jobs kürzlich bekannt gab, soll sich das ändern. Nach mehr als 10 Jahren will Apple nun x86er statt PowerPC-Prozessoren in Macintosh-Rechner einbauen. [7]

[8] z.B. *AltiVec*, eine Vektor-Recheneinheit, die ähnlich wie Intels *MMX* Multimedia-Anwend. beschleunigt. [2]

immer noch relativ frei programmierbar. Noch schneller, aber dafür nur für jeweils einen ganz bestimmten Zweck geeignet, sind *Spezialprozessoren*. Sie enthalten Hardwareimplementierungen häufig benötigter Algorithmen und können so beispielsweise eine FFT[9] oder gleich eine komplette JPEG-Kompression in wenigen Taktzyklen erledigen. Sie werden den GPPs oft als Co-Prozessoren zur Seite gestellt.

Es gilt also immer einen Kompromiss zu finden, zwischen der Optimierung auf einen bestimmten Zweck einerseits, und einem breiten Spektrum an möglichen Anwendungen andererseits.

[9] FFT: **F**ast **F**ourier **T**ransformation

Kapitel 2

Aufbau und Funktionsweise eines einfachen Prozessors

Im nun folgenden Kapitel soll der prinzipielle Aufbau eines Mikroprozessors dargestellt werden. Auf die zahlreichen Unterschiede zwischen den einzelnen Architekturen (vgl. Abschnitt 1.2) wird dabei nur am Rande eingegangen. Ziel ist es vielmehr, ein möglichst einfaches Modell, welches die meisten Architekturen gemeinsam haben, Schritt für Schritt zu erläutern und auf diese Weise die grundlegende Arbeitsweise eines Prozessors zu verdeutlichen. Erweiterungen und Optimierungen, die im Laufe der letzten Jahre hinzugekommen sind, werden dann detaillierter in Kapitel 3 erörtert.

Zunächst werden die einzelnen Komponenten vorgestellt und ihre Funktionsweise erläutert. Auf elektrotechnische Zusammenhänge wird dabei nur soweit eingegangen, wie dies zum Verständnis unbedingt notwendig ist. Für eine ausführlichere Darstellung empfiehlt sich [15], [10] und vor allem [23]. Anschließend wird dann in Abschnitt 2.2 ein beispielhafter Programmablauf durchexerziert, um das Zusammenspiel der einzelnen Teile deutlich zu machen.

2.1 Komponenten eines Prozessors

Ein normaler Mikroprozessor besteht den in Abbildung 2.1 dargestellten vier Funktionsgruppen. Diese bestehen jeweils wieder aus mehreren Untereinheiten, die im Bild allerdings nur angedeutet sind. Der genaue Aufbau wird in den entsprechenden Abschnitten beschrieben.

Das *Steuerwerk* (blau) kümmert sich um das Einlesen und Interpretieren der Programminstruktionen aus dem Speicher. Es steuert dann das *Rechenwerk* (grün) an, welches den Befehl ausführt. Steuerwerk und Rechenwerk benötigen sehr schnelle Zwischenspeicher, in denen sie Operanden, Zwischenergebnisse und Statusinformationen kurzzeitig ablegen können. Diese Zwischenspeicher sind die *Register* (rot) des Prozessors. Die Gesamtheit aller Register bezeichnet man als *Registersatz*. Verbunden sind die einzelnen Teile über ein internes Bussystem, das aus Daten-

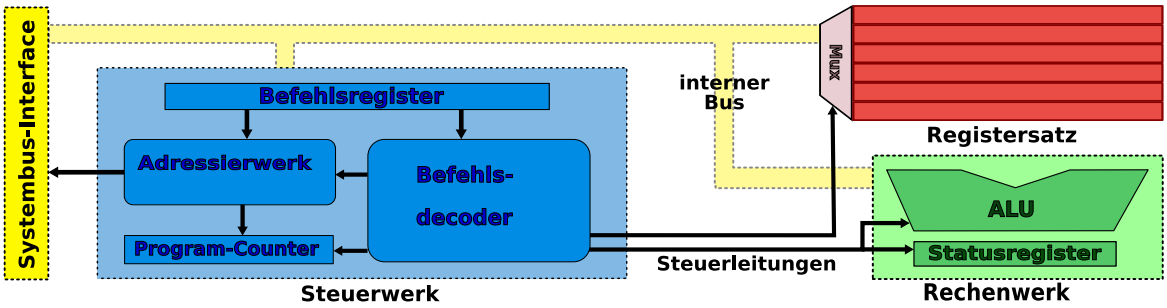


Abbildung 2.1: Funktionsgruppen eines Prozessors

und Adressbus (hellgelb) und einzelnen Steuerleitungen (schwarz) bestehen. Das interne Bussystem ist über ein *Interface* an den Systembus angeschlossen, über den der Prozessor auf Speicher und Peripheriegeräte zugreifen kann (vgl. Bild 1.2).

2.1.1 Register

Wie in Abschnitt 3.2 noch genauer ausgeführt wird, ist der Zugriff auf den Speicher eine sehr aufwändige – und vor allem langsame – Operation. Es muß zunächst die Adresse des gewünschten Datums berechnet, und über das Systembus-Interface (Abschnitt 2.1.4) an den Speicher übermittelt werden. Dieser beginnt, das Datum auszulesen und auf den Bus zu legen. Von dort liest es das Interface wieder ein und übermittelt das gewünschte Datum ans Steuerwerk. Da der Speicher sehr viel langsamer ist als der Prozessor, können so etliche Taktzyklen vergehen, in denen der Prozessor nicht arbeiten kann, da er auf seine Daten wartet.

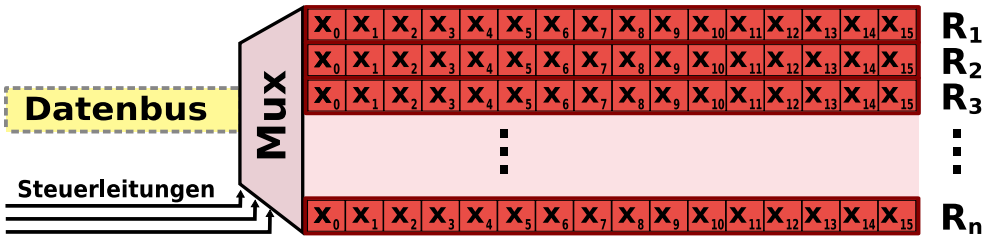


Abbildung 2.2: Register

Es werden also schnelle Speicher benötigt, in denen Operanden, Befehle, Ergebnisse, Speicheradressen und Statusinformationen abgelegt werden können. Diese *Register* sind aus schnellen Flipflops aufgebaut und sitzen direkt auf der CPU, arbeiten also mit dem vollen Prozessortakt.

Da es nur recht wenige von ihnen gibt (CISC-Prozessoren besitzen zwischen 8 und 16, RISC-Prozessoren um die 32 Register), können sie direkt durch das Schalten von Steuerleitungen ausgewählt und in einem Takt gelesen bzw. beschrieben werden (Abbildung 2.2). Man unterscheidet *General-Purpose*-Register und *Spezial*-Register.

General-Purpose-Register können vom Programmierer frei gelesen oder beschrieben werden. Sie werden benutzt, um Daten und Speicheradressen abzulegen.

Spezialregister werden für die interne Funktion des Prozessors benötigt und sind daher entweder überhaupt nicht für den Programmierer zugänglich (z.B. diverse Puffer-Register) oder nur eingeschränkt les- bzw. beschreibbar (z.B. das in Abschnitt 2.1.2 vorgestellte *Status-Register* des Rechenwerks).
Weitere dieser Spezialregister sind den *Programmzähler* (PC-Register), das *Befehlsregister* und das *Interrupt-Vektor-Basisregister*, auf die in den Abschnitten 2.1.3 und 2.3 genauer eingegangen wird.

Registerbreite und Zahlendarstellung

Die Breite der Register bestimmt den Zahlenbereich, mit dem der Prozessor arbeitet. Je mehr Bit verwendet werden, desto größer ist der Zahlenbereich, der erfaßt werden kann. Für eine Zahl mit n Bits $x_0 \ldots x_{n-1}$ ergibt sich (bei Zweierkomplementdarstellung) ein Bereich von

$$-2^{n-1} - 1 \ \ldots \ + 2^{n-1} \tag{2.1}$$

Das erste Bit x_0 wird in diesem Format als Vorzeichenbit verwendet und der Wert $x_i = 0 \, \forall i \in n$ ist für die Darstellung der Null reserviert. Eine 16-Bit Zahl kann so also Werte von -32769 bis +32768 annehmen bzw. 65535 Speicherzellen (64KBit) adressieren. Bei Fließkommazahlen bestimmt die Bitbreite die Genauigkeit, mit der die Zahlen repräsentiert werden [1].

Bei heutigen Desktop-Rechnern sind die Register meist 32 oder 64 Bit breit. Eingebettete Systeme wie Microcontroller-CPUs benutzen teilweise noch 8 Bit für die Zahlendarstellung. Die Bitbreite ist ein wichtiges Leistungsmerkmal des Prozessors, da davon unmittelbar der adressierbare Speicherbereich abhängt. Der Übergang von 32- auf 64-Bit-Rechner wurde auch weniger wegen der erhöhten Rechengenauigkeit, sondern wegen der 16384 Petabyte ($1,72 \cdot 10^{10}$ MB), die man mit 64 Bit adressieren kann forciert. Mit den bis dahin üblichen 32 Bit liegt diese Grenze schon bei 4 Gigabyte (4096 MB), was gerade bei sehr speicherhungrigen Applikationen zu knapp ist.

2.1.2 Rechenwerk

Das Rechenwerk wird oft auch als *Ausführungseinheit* (Execution Unit) bezeichnet, da hier die eigentliche Verarbeitung der Befehle stattfindet. Es besteht aus einer oder mehreren ALUs (**A**rithmetic **L**ogical **U**nit), welche die vom Steuerwerk angeforderten Rechenoperationen ausführen. Es gibt ALUs für das Rechnen mit ganzen Zahlen (Integer Units) und für Gleitkomma-Operationen (Floatingpoint Units). Aus Gründen der Verständlichkeit wird der Aufbau einer ALU hier am Beispiel der Integer-Unit erklärt. Gleitkomma-Operationen funktionieren ähnlich, allerdings ist die Darstellung der Zahlen im Rechner (festgelegt in IEEE-754 [1]) etwas komplizierter.

Die ALU ist ein Schaltnetz[1], das ein bzw. zwei Operanden A und B am Eingang mit verschiedenen Funktionen verknüpfen kann. Sobald alle Gatter durchlaufen sind, liegt das Ergebnis X am Ausgang an. Möglich sind logische Verknüpfungen wie:

Konjunktion (AND): Am Ausgang liegt eine Eins an, wenn *beide* Eingänge Eins sind ($X = A \wedge B$).

Disjunktion (OR): Am Ausgang liegt eine Eins an, wenn der eine *oder* der andere Ausgang *oder* auch beide Eins sind ($X = A \vee B$).

Antivalenz (XOR): Am Ausgang liegt eine Eins an, wenn *entweder* der eine *oder* der andere Ausgang Eins ist, nicht jedoch falls beide Eins sind ($X = A \oplus B$).

Negation (NOT): Der Eingang wird invertiert. Ist er Eins, liegt am Ausgang eine Null an und umgekehrt. ($X = \bar{A}$) Diese Operation benötigt nur einen Operanden.

oder arithmetische wie:

Addition: Die beiden Eingänge werden binär addiert ($X = A + B$). Sind beide Eingänge 1, entsteht neben dem eigentlichen Ergebnisbit noch ein Übertrag (*Carry-Bit*). Die **Subtraktion** kann auf die Addition zurückgeführt werden ($X = A - B = A + (-B)$).

Multiplikation: ($X = A + B$)

Welche Operation angewendet wird, bestimmt das Steuerwerk über das Schalten eines Multiplexers. Auf diese Weise wird der Ausgang des gewünschten Funktionsblockes auf den Ausgang der ALU gelegt. In Tabelle 2.1 sind noch einmal die Wahrheitstabellen der einzelnen Funktionen zusammengefaßt.

Tabelle 2.1: Wahrheitstabellen einiger logischer Operationen

Eingänge		**AND**	**OR**	**XOR**	**NOT**	**Add**
A	**B**	$A \wedge B$	$A \vee B$	$A \oplus B$	$\bar{A}$	$A + B$
0	0	0	0	0	1	0
0	1	0	1	1	1	1
1	0	0	1	1	0	1
1	1	1	1	0	0	**1**0

Das Blockdiagramm solch einer ALU ist in Abbildung 2.3 zu sehen. Auf die Darstellung der internen Funktionsweise der einzelnen Schaltnetze bis hinunter zur Transistorebene wird an dieser

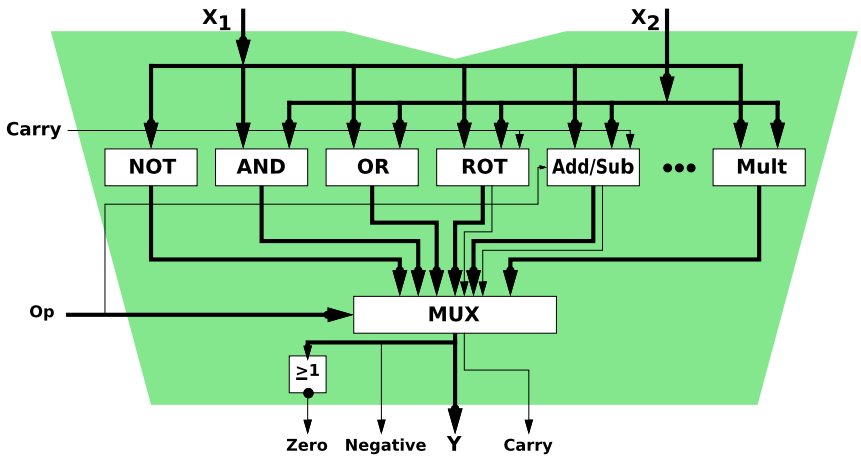


Abbildung 2.3: Schematische Darstellung einer 1-Bit ALU **Nach** [17, S.178]

Stelle aus Platzgründen verzichtet. Eine sehr gute Einführung in dieses Thema bieten Tietze und Schenk [23, S.643-682].

Die Abbildung 2.3 zeigt eine 1-Bit-ALU. Nun sind die Operatoren in Wirklichkeit natürlich nicht nur einfache boolsche Werte sondern binäre Zahlen, die aus mehreren Bits bestehen. Es müssen also entsprechend viele 1-Bit-ALUs parallel geschaltet werden.

Die Inhalte der Register werden an den Eingang des Schaltnetzes angelegt. Jedes Bit eines solchen Operanden ist ein Eingangswert für eine 1-Bit-ALU. Die Mux-Steuerleitungen sind mit jeder von ihnen verbunden, so daß alle die gleiche Operation ausführen. Das bei der Addition eventuell entstehende Carry-Bit ($1+1=\mathbf{1}0$) wird an den nächsten Block weitergereicht und dort in die Berechnung der nächsten Stelle einbezogen. Am Ausgang der einzelnen Blöcke liegt dann das Ergebnis der Operation an, welches wiederum in ein Register geschrieben wird.

Die Versorgung der ALU mit Operanden bzw. die Abspeicherung des Ergebnisses ist unterschiedlich implementiert. Früher gab es im Rechenwerk oft zwei dedizierte Operandenregister (meist mit A und B bezeichnet) und ein spezielles Ergebnisregister (den Akkumulator). In modernen Prozessoren gibt es oft nur noch eine Reihe von General-Purpose-Registern (siehe 2.1.1), die sowohl auf die Eingänge der ALU (Operanden) als auch an den Ausgang (Ergebnis) geschaltet werden können.

Neben dem eigentlichen Ergebnis werden noch einige Nebeninformationen im sogenannten *Statusregister* (engl. **C**ondition **C**ode **R**egister, CCR) gespeichert. Jedes Bit dieses Registers hat eine festgelegte Bedeutung und kann vom Steuerwerk einzeln ausgewertet werden. Auf diese Weise kann z.B. ein arithmetischer Überlauf signalisiert (Overflow-Bit) oder sehr effizient geprüft werden, ob das Ergebnis der Operation gleich Null (Zero-Bit) oder negativ (Negative-Bit) ist. Auch der Übertrag der höchstwertigen Stelle (Carry-Bit) wird dort gespeichert.

[1] Im Gegensatz zu einem Schalt*werk* hält ein Schalt*netz* keinerlei Zustandsinformationen. Der Ausgangswert ist nur von den Eingangsoperanden abhängig und nicht von eventuellen vorherigen Operationen (Gedächtnislosigkeit).

Gerade das Zero-Bit ist sehr wichtig für die Auswertung von Vergleichen, zum Beispiel bei bedingten Sprungbefehlen. Diese werden nämlich oft so durchgeführt, daß die beiden zu vergleichenden Werte subtrahiert werden. Ist das Ergebnis dann Null (Zero-Bit gesetzt), so ist der Vergleich wahr (*true*) und der Sprung wird durchgeführt.

2.1.3 Steuerwerk (Control Unit)

Das Steuerwerk ist dafür zuständig, das abzuarbeitende Programm Schritt für Schritt einzulesen, die erhaltenen Befehle zu dekodieren und je nach Befehl Steuersignale in der korrekten zeitlichen Abfolge an die anderen Rechnerkomponenten zu senden.

Maschinensprache

In Abschnitt 1.2 wurde bereits erwähnt, daß der Prozessor seine Anweisungen in Maschinensprache (auch Maschinencode oder Makroprogramm genannt) erhält. Diese werden vom Compiler aus dem Quelltext einer höheren Programmiersprache (Java[2], ADA, C++, Pascal, C, ...) erzeugt. Der Code ist hochgradig prozessorspezifisch und für Menschen nur sehr schwer lesbar. Tabelle 2.2 zeigt ein Beispiel. Es handelt sich hier um die ganzzahlige Addition zweier 16 Bit-Zahlen auf einem Motorola MC6800, einem sehr einfachen 8-Bit-Prozessor. [5]

Tabelle 2.2: Addition zweier Zahlen in Maschinencode Assembler und C, **Nach: [23, S.1078]**

	Maschinencode (Bin)	Maschinencode (Hex)	Assembler	C
1	10010110 00000010	96 02	LDA A $02	int x, y;
2	10011011 00000100	9B 04	ADD A $04	x = x+y;
3	00000001	01	NOP	
4	10010111 00000110	97 06	STA A $06	
5	10010110 00000001	96 01	LDA A $01	
6	10011001 00000011	99 03	ADC A $03	
7	00000001	01	NOP	
8	10010111 00000101	97 05	STA A $05	
9	00111001	39	RTS	

In der ersten Spalte ist das Programm so zu sehen, wie der Prozessor es einliest, als Folge von Bits. Der erste Teil eines Befehls (der sogenannte *Op-Code*) gibt immer die auszuführende Operation an. Beim MC6800 sind dies die ersten 8 Bits. Bei anderen Prozessoren kann das anders

[2] In Java und div. Derivaten wie z.B. .NET wird der Quelltext nicht direkt in Maschinensprache sondern zunächst in eine plattformunabhängige Zwischensprache, den sog. Bytecode (bzw. MSIL) übersetzt. Dieser wird dann zur Laufzeit in Maschinensprache umgesetzt. Aus Sicht des Prozessors spielt dieser Unterschied jedoch keine Rolle.

sein. Intels Prozessoren arbeiten z.B. mit unterschiedlich langen Op-Codes was das Einlesen verkompliziert. Die Gesamtheit aller Op-Codes bezeichnet man als den *Befehlssatz* des Prozessors. Die folgenden Bytes sind die zugehörigen Operanden. In diesem Fall sind die Operanden einfach Speicheradressen, an denen sich die zu verarbeitenden Daten befinden, bzw. an die das Ergebnis geschrieben werden soll. Es können jedoch auch absolute Werte (5, -7, 42 ...) sein.

In Spalte zwei ist das gleiche Programm noch einmal in hexadezimaler[3] Schreibweise dargestellt. Diese Darstellungsweise ist üblich, da die Hex-Zahlen vom Menschen (etwas) leichter zu lesen sind als endlose Folgen von Nullen und Einsen. Es handelt sich aber, unabhängig von der Darstellung, immer noch um reine Maschinensprache.

Die dritte Spalte geht noch eine Abstraktionsebene höher. Es handelt sich um sogenannten *Assembler*-Code. Befehle sind hier nicht mehr durch binäre Op-Codes sondern durch nmemonische Abkürzungen dargestellt, die einen Umgang mit dem Programm erleichtern. Die Operanden können in verschiedenen Schreibweisen angegeben werden. Da es sich oft um Speicheradressen handelt, ist die hexadezimale Form (meist gekennzeichnet durch ein vorangestelltes `$` oder `0x`) weit verbreitet.
Ein ebenfalls als Assembler bezeichnetes Programm wandelt solchen Code dann in Maschinensprache um. Es handelt sich aber immer noch um eine $1 \rightarrow 1$-Abbildung. Jeder Assembler-Befehl wird in sein Maschinensprache-Pendant umgesetzt.
Die direkte Programmierung in Assembler ist inzwischen unüblich, da die Programme sehr schwer zu warten und die Programmierung sehr mühsam und fehlerträchtig ist. Allenfalls in speziellen Routinen, wo es auf extreme Performance ankommt, werden noch sogenannte Inline-Assembler (kurze Stücke Assembler-Code, die in Quelltext von Hochsprachen eingebettet werden) verwendet.

Spalte vier schließlich zeigt den entsprechenden Ausschnitt eines C-Programmes. Hier wird zusätzlich noch ein Compiler benötigt, der die Anweisungen in entsprechende Assembler-Befehle übersetzt, so daß diese wiederum in Maschinensprache umgesetzt und von der CPU ausgeführt werden können.

Zurück zum Steuerwerk

Nachdem nun geklärt ist in welcher Form die auszuführenden Programmanweisungen vorliegen, stellt sich jetzt die Frage, wie ein Steuerwerk aussehen muß, das diese Befehle interpretieren soll. Abbildung 2.4 zeigt – stark vereinfacht – das Steuerwerk eines typischen CISC-Prozessors.[4]

Es besteht in prinzipiell aus einem Befehlsdecoder, und einem Adressierwerk. Der *Befehlsdecoder* nimmt einen Op-Codes entgegen, sucht den Maschinenbefehl dazu heraus und schaltet in einer durch diesen Befehl vorgegebenen Reihenfolge die entsprechenden Steuerleitungen, die dann die anderen Prozessorkomponenten (Register, ALU, ...) kontrollieren.

[3] Zahlen zur Basis 16. Die verwendeten Ziffern sind `012345679ABCDEF`. Jede Ziffer entspricht 4 Bits.

[4] Aktuelle Prozessoren arbeiten meist *superskalar* bzw. mit *Pipelining* (siehe Abschnitt 3.3) was ein sehr viel komplizierteres Steuerwerk erfordert. Um die grundlegende Funktionsweise zu verstehen, ist dieses einfache Modell jedoch sehr gut geeignet.

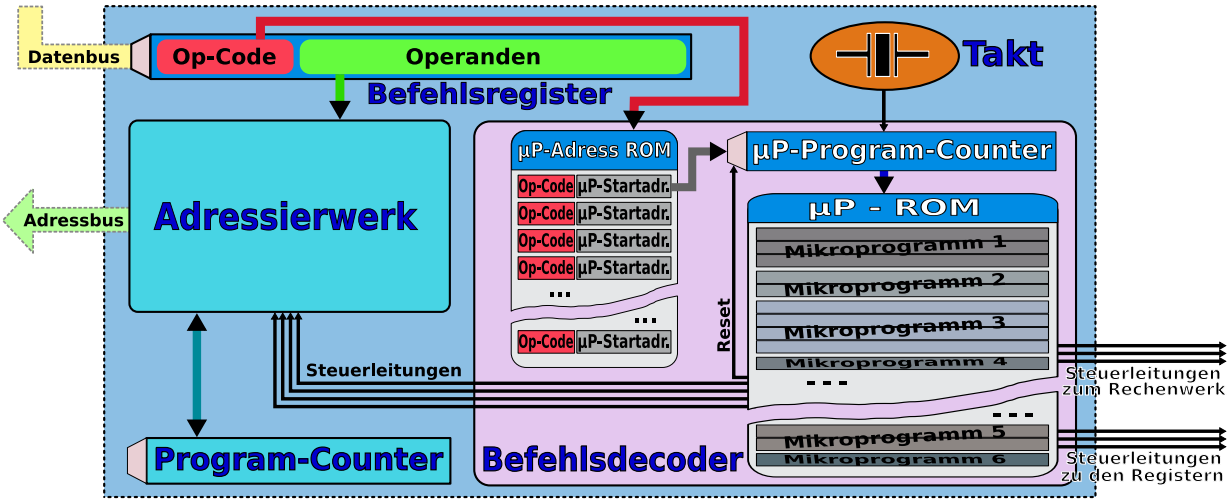


Abbildung 2.4: Aufbau eines CISC-Steuerwerkes

Das *Adressierwerk* wertet den zweiten Teil des Befehls aus – die Operanden. Diese sind nämlich oft nicht (wie im Beispiel in Tabelle 2.2 gezeigt)

- *absolute* Speicheradressen (z.B. `$01` oder `$06`), sondern sind
- *relativ* (Adresse ist aktuelle Adresse + Wert des Operanden) oder sogar
- *indirekt* (Adresse steht in der vom Operanden angegebenen Speicherzelle)

adressiert und werden erst vom Adressierwerk in reale Speicheradressen umgesetzt. Das hat den Vorteil, daß das Programm nicht immer an exakt der gleichen Stelle im Speicher liegen muß. Gerade bei modernen Multitasking-Betriebssystemen muß man davon ausgehen, daß es sich bei jedem Start in einem anderen Adreßbereich befinden wird. Früher wurde für solche Adreßarithmetik die ALU bemüht, inzwischen existiert mit dem Adressierwerk eine spezielle Funktionseinheit dafür.

Die Adresse des aktuellen Befehls befindet sich immer im Programmzähler (*Program Counter, PC)*, einem Spezialregister. Dieser wird aus dem Speicher geladen und zunächst im *Befehlsregister* abgelegt. Der Adreßteil (die Operanden des Befehls) wird an das Adressierwerk übergeben. Der Op-Code (der erste Teil des Befehls) landet im Befehlsdecoder.[5] Anschließend wird der Programmzähler erhöht, so daß er nun wieder auf die Adresse des als nächstes abzuarbeitenden Befehles zeigt.

[5]Die Darstellung des Befehlsregisters ist hier aus Gründen der Übersichtlichkeit etwas vereinfacht. Normalerweise handelt es sich dabei um einen ganzen Block von kaskadierten Registern, um mehrere Befehle auf einmal aufzunehmen. Zusätzlich wird ein Befehls-Vordecoder verwendet, der festlegt, welche Bits zum Op-Code und welche zum Adreßteil gehören.

Im Befehlsdecoder wird dann für jeden Op-Code ein bestimmtes *Mikroprogramm* (oft als μP abgekürzt) ausgeführt. Dieses Mikroprogramm ist nicht mit dem vom Compiler erzeugten Programm, welches die CPU gerade ausführen soll (zur besseren Unterscheidung oft als *Makroprogramm* bezeichnet) zu verwechseln! Es handelt sich beim Mikroprogramm einfach um mehrere Schritte (Mikrobefehle), die nötig sind um **einen** Befehl des Makroprogramms auszuführen. Jeder Schritt ist quasi eine Liste von Steuerleitungen die in diesem Moment geschaltet werden sollen..

In einem reinen RISC-Prozessor hat jedes Mikroprogramm nur einen Schritt. Es wird also nur eine einzige Aktion durchgeführt, so daß eine Ablaufsteuerung entfallen kann. Bei dem dargestellten CISC-Steuerwerk werden jedoch nacheinander mehrere Schritte durchgeführt. Diese sind in einem speziellen Speicher im Steuerwerk, dem Mikroprogramm-ROM (μP-ROM) abgelegt.[6] Dieses ist bei modernen Prozessoren oft als EEPROM ausgelegt, so daß der Mikrocode nachträglich geändert werden kann, um beispielsweise Fehler im Prozessordesign zu auszugleichen.

Der Ablauf bei der DekodierungDecodierung eines Befehls ist folgender: Zunächst wird im Mikroprogramm-Adreßspeicher (μP-Adreß-ROM) nachgesehen, zu welcher μP-Startadresse der übergebene Op-Code gehört. Diese Adresse wird dann in den μP-Programmzähler (μP-PC) geladen. Nun kann der Befehlsdecoder an die entsprechende Stelle im μP-ROM springen und den ersten Schritt ausführen. Der Taktgeber des Prozessors erhöht den μP-PC mit jedem Takt um eins, so daß in jedem Takt ein weiterer Schritt ausgeführt wird, d.h. andere Steuerleitungen gesetzt werden. Ist das Mikroprogramm komplett durchgelaufen und damit der Befehl abgearbeitet, wird der μP-PC wieder zurückgesetzt.

Nun kann der nächste Befehl ausgeführt werden. Dazu wird im Programmzähler (diesmal der für das Makroprogramm!) nachgesehen, wo dieser zu finden ist. Handelte es sich bei der zuletzt ausgeführten Anweisung um einen Sprungbefehl, so wurde während der Abarbeitung bereits der Operand mit dem Sprungziel vom Adressierwerk ausgewertet. Die errechnete Adresse muß dann vorher noch in den Programmzähler geschrieben werden. Anschließend wird die an dieser Stelle im Speicher stehende Bitfolge ins Befehlsregister geladen und der ganze Zyklus geht wieder von vorne los.

2.1.4 Systembus-Interface

Das Systembus-Interface ist für die Vermittlung zwischen dem internen Prozessorbus (über den Rechenwerk, Steuerwerk, Register etc. verbunden sind) und dem externen Bus, über den Peripheriecontroller und Speicher angekoppelt sind, zuständig.

Bei der Ankoppelung an den Bus ist zu beachten, daß immer nur ein einziger Teilnehmer gleichzeitig schreibend darauf zugreifen darf. Daher benutzt man sogenannte *Bus-Treiber*. Sie enthalten

[6] Um Platz und damit wertvolle Chipfläche zu sparen, werden die μP-Schritte oft noch weiter in sog. Nanoprogramme unterteilt. Das μP-ROM enthält dann wiederum nur Startadressen für die einzelnen Nanoprogramme. Diese zusätzliche Hierarchiestufe wurde hier jedoch der Übersichtlichkeit halber weggelassen.

Tri-State Gatter, die drei mögliche Zustände annehmen können, High (H), Low (L) und High-Impedance (Z). Im Z-Zustand hat das Gatter einen sehr hohen Ausgangswiderstand, so daß die Datenübertragung auf dem Bus nicht gestört wird. In Abbildung 2.5 ist so eine Ankoppelung über Tri-State Ausgänge an eine gemeinsame Datenleitung dargestellt.

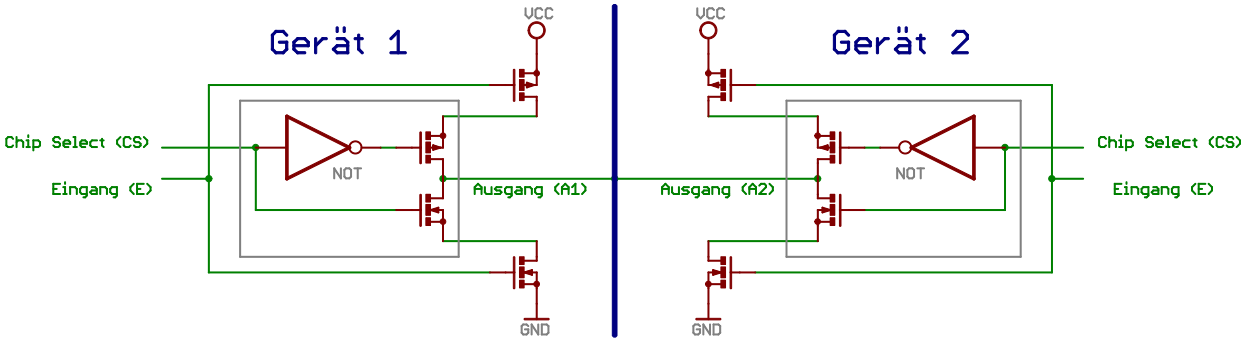


Abbildung 2.5: Tri-State-Ankoppelung von zwei Geräten an eine gemeinsame Datenleitung

Ohne den grau umrandeten Teil wären die Geräte nur über normale CMOS-Ausgangsstufen an die Leitung (blau) angeschlossen. Wollen aber nun zwei Teilnehmer unterschiedliche Pegel auf die Leitung legen, so würde nicht nur der Bus in einen undefinierten Zustand gehen, über die Ausgangstransistoren der zwei Treiber würde außerdem ein sehr hoher Strom fließen, der die Stufen zerstören kann. Die grau umrandete Schaltung sorgt jedoch dafür, daß das Gerät über ein Low setzen der Chip-Select Leitung (CS) hochohmig vom Ausgang abgekoppelt werden kann. Achtet man darauf, daß stets nur ein einziges Gerät den Pegel der Leitung bestimmt und alle anderen im hochohmigen Z-Zustand sind, ist eine störungsfreie Datenübertragung möglich.

Neben Bus-Treibern (die außer der Tri-State Ankoppelung auch gleich eine eventuell nötige Pegelwandlung erledigen), enthält das Interface auch noch *Pufferregister* für Adressen und Daten. Das ist nötig, weil sich die Busprotokolle (Zugriffssteuerung, Codierung etc.) des internen Prozessorbusses und des externen Systembusses in der Regel unterscheiden. Laufen beide Busse außerdem noch mit unterschiedlichen Frequenzen (asynchron), so sind diese Puffer meist als FIFO-Speicher[7] realisiert, um mehrere Adressen bzw. Daten zwischenspeichern zu können.

Aus Sicht des Steuerwerks stellt sich das Systembus-Interface letztendlich folgendermaßen dar: Es wird eine Adresse in den Adreßpuffer geschrieben und einige Zeit später (je nach Geschwindigkeit des Speichers bzw. des Peripheriegerätes) liegt das angeforderte Datum im Datenregister zum Auslesen bereit.

[7]FIFO: First In First Out; Daten die als erstes hineingeschrieben wurden, werden auch als erstes wieder ausgelesen (Warteschlange)

2.2 Und jetzt alle zusammen! – Ein beispielhafter Programmablauf

Nachdem nun alle Komponenten des Prozessors vorgestellt sind und ihre Funktionsweise angedeutet wurde, wird es nun Zeit, das Zusammenspiel all dieser Teile zu veranschaulichen. Am Besten gelingt dies durch ein kleines Beispielprogramm, wie das in Tabelle 2.3.

Gezeigt ist ein Teil des Speicherinhalts eines MC6800-Prozessors, der schon in Abschnitt 2.1.3 als Beispiel für die Darstellung der Maschinensprache benutzt wurde.

Tabelle 2.3: Programmstück für den MC6800

Adresse	Hex-Code	Assembler	Bedeutung
$0020	0E		8-Bit Wert $0E (Dezimal: 14)
$0021	05		8-Bit Wert $05 (Dezimal: 5)
...	...	...	...
$2000	96 20	LDA A $20	Lade den Wert aus Speicherzelle $20 in Register A
$2002	5F	CLR B	Setze Register B auf Null
$2003	5C	INC B	Erhöhe den Wert von Register B um Eins
$2004	9D 21	SUB A $21	Subtrahiere von Register A den Wert aus Speicherzelle $21
$2006	2E 20 03	BGT $2003	Fahre bei Adr. $2003 fort, wenn Ergebnis der letzten Operation > 0 war
$2009	5A	DEC B	Verringere den Wert von Register B um Eins

Die erste Spalte zeigt die Adressen der einzelnen (je 8 Bit großen) Speicherzellen und die zweite deren Inhalt in hexadezimaler Schreibweise. Auf die sich daraus ergebende Bitfolge wurde dieses Mal verzichtet. Dafür ist neben der Assembler-Darstellung noch die Bedeutung der einzelnen Befehle bzw. Daten notiert.

Im unteren Teil des Speichers, in den Zellen $0020 und $0021 sind die Daten gespeichert, mit denen das Programm arbeitet. Im oberen Teil (Zellen $2000 bis $2009) ist das Programm untergebracht.

Diese Einteilung ist absolut willkürlich und dient nur der besseren Übersicht. Da es sich hier um eine VON NEUMANN-Architektur (vgl. 1.2.1) handelt, könnten Daten und Programme auch wild gemischt sein. Probleme treten erst dann auf, wenn (beispielsweise durch eine falsche Sprunganweisung) Daten als Programmanweisungen interpretiert werden. Die Bitfolge 0E 05 kann vom Prozessor je nach Kontext als zwei 8-Bit Dezimalzahlen (14 und 5) oder als ein Befehl mit dem Op-Code 0E und dem Adreßteil 05 aufgefaßt werden.

Das Programm ist sehr einfach und besteht nur aus 6 Anweisungen (10 Byte). Zunächst wird ein Startwert (hier 14) geladen, und dann so lange um einen bestimmten Betrag (hier 5) vermindert, bis das Ergebnis erstmalig negativ oder Null ist. Die Anzahl der dafür notwendigen Durchläufe wird gezählt und ganz zum Schluß nochmal um Eins vermindert.

Die Entsprechung in C sähe ungefähr folgendermaßen aus:

```
short startwert = 14;
short schrittweite = 5;
short zaehler = 0;
do {
   startwert -= schrittweite;
   zaehler++;
} while (startwert > 0)
zaehler--;
```

oder – etwas eleganter – auch so:

```
short zaehler = 14 / 5;
```

da man obiges Konstrukt (zumindest für positive Zahlen) auch als eine – etwas unbeholfene – ganzzahlige Division ohne Rest auffassen kann. Das Ergebnis steht dann in der Variable `zaehler` bzw. im Register `B`.

Sicher gibt es elegantere Arten, zwei Zahlen zu dividieren aber zum Verdeutlichen der Arbeitsweise eines Prozessors erfüllt das Programm seinen Zweck.

Ablauf im Prozessor

Was passiert nun im Prozessor, wenn das oben vorgestellte Programm abläuft? Zum Start muß zunächst die Adresse der ersten Instruktion `$2000` im Programmzähler stehen.

`LDA A $20` → Schreibe den Wert in `$20` in Register A Das Adressierwerk schreibt die Adresse des Befehls (ggf. nach einer Umrechnung in reale Speicheradressen) in den Adreßpuffer vom Systembus-Interface und erhöht den Zähler auf `$2002`, so daß er nun auf die Startadresse des nächsten Befehls zeigt. Die beiden Bytes[8] `96` und `20` werden aus Adresse `$2000` und `$2001` des Speichers ausgelesen und stehen kurze Zeit später im Datenpuffer zur Verfügung. Von dort gelangen sie ins Befehlsregister des Steuerwerkes und die Auswertung des soeben erhaltenen Befehls beginnt.

Dazu vergleicht der Befehlsdecoder den Op-Code – in diesem Fall die `96` – mit seinen Einträgen im μP-Adreß-ROM, sucht die Startadresse des Mikroprogrammes für den `LDA A`-Befehl heraus und lädt diese in den μP-Programmzähler. Nun springt er an die entsprechende Stelle im μP-ROM und setzt die Steuerleitungen entsprechend der dort vorgefundenen Informationen. Mit dem nächsten Taktzyklus wird der μP-Programmzähler um eins erhöht und der nächste Mikroprogrammschritt wird ausgeführt. Durch den Ablauf der Mikroprogrammschritte werden über die Steuerleitungen die anderen Komponenten der CPU kontrolliert.

[8] wenn nicht anders angegeben handelt es sich bei allen folgenden Zahlenangaben immer um hexadezimale Werte

So wird das Adressierwerk veranlaßt, über das Systembus-Interface den Inhalt von $20 aus dem Speicher zu holen und auf den internen Prozessorbus zu legen. Danach werden die Register angesteuert, so daß der am Bus anliegende Wert in Register A gespeichert wird. Als letztes wird der µP-Programmzähler wieder auf Null gesetzt. Damit ist das Mikroprogramm für LDA A beendet. Der Wert 0E (dezimal: 14) wurde von $20 in das Register A übertragen. Der nächste Befehl kann ausgeführt werden.

CLR B → Setze Register B auf Null Im Programmzähler steht nun die $2002, was auf 5F verweist. Das Byte wird wie gehabt vom Adressierwerk über das Systembus-Interface aus dem Speicher geholt (Programmzähler erhöhen nicht vergessen!) und auf den internen Bus gelegt, von wo es dann ins Befehlsregister geschrieben wird. Diesmal sind keine Operanden vorhanden, so daß nur der Op-Code vom Befehlsdecoder ausgewertet werden muß.

Wieder wird die µP-Startadresse im Adreß-ROM nachgeschlagen, der µP-Programmzähler damit geladen und die Ausführung des im µP-ROM gespeicherten Mikroprogramms begonnen. Dieses ist erheblich kürzer als das Vorhergehende. Es werden lediglich die zu den Registern gehenden Steuerleitungen so geschaltet, daß der Inhalt von Register B gelöscht wird. Nach dem Reset des µP-Zählers ist das Mikroprogramm abgeschlossen und der nächste Befehl kann aus $2003 geholt werden.

INC B → Erhöhe den Wert von Register B um Eins Der als nächstes ins Befehlsregister geladene Op-Code 5C führt zu einem ähnlich einfachen Mikroprogramm. Allerdings ist hier erstmalig das Rechenwerk beteiligt. Der Inhalt des Registers B (im Moment 00) wird auf den Eingang der ALU gelegt. Am anderen Eingang liegt der feste Wert 01 an. Über die Multiplex-Steuerleitungen wird als Operation die Addition ausgewählt und das Ergebnis (01) wird anschließend zurück nach B geschrieben.

SUB A $21 → Subtrahiere von Register A den Wert aus Speicherzelle $21 Der in $2004 stehende Subtraktionsbefehl (9D) (mit seinem Adreßteil 21 in $2005) läuft im Prinzip so ähnlich ab wie der vorherige INC B-Befehl. Allerdings ist hier der zweite Wert der Rechenoperation nicht fest Eins, sondern er ist durch den Operanden des Befehls gegeben.

Bevor also das Rechenwerk in Aktion treten kann, veranlaßt das Mikroprogramm im Befehlsdecoder das Adressierwerk, den Inhalt der Zelle $21 (also 05) aus dem Speicher zu holen und auf den zweiten Eingang der ALU zu schalten. Am ersten Eingang liegt der Inhalt des Registers A (im Moment noch 0E, also dezimal 14) an. Am Ausgang liegt – nach Auswahl der Subtraktion durch den Multiplexer – dann das Ergebnis 09, welches zum Schluß zurück ins Register A geschrieben wird.

Gleichzeitig wird – wie bei jeder Operation des Rechenwerks – das Statusregister (CCR) aktualisiert. Für den weiteren Programmablauf sind in diesem Fall besonders die Z- und N-Bits interessant. Das Z-Bit gibt an, ob das Ergebnis der Operation Null (zero) und das N-Bit ob es

negativ war (siehe Abschnitt 2.1.2). Da weder das Eine noch das Andere zutrifft (das Ergebnis ist ja 14), sind beide Bits Null.

`BGT $2003` → Fahre bei Adresse `$2003` fort, wenn das Ergebnis der letzten Rechenoperation größer als Null war Der `BGT`-Befehl (**B**ranch if **G**reater **T**hen) ist ein Sprungbefehl, der das Statusregister auswertet. Ist dort weder das Z-, noch das N-Bit gesetzt (das Ergebnis also größer als Null), wird als nächstes an der im Operanden angegeben Adresse mit der Programmausführung fortgefahren. Der Programmablauf verzweigt (branch). Andernfalls geht es einfach mit dem nächsten Befehl weiter.

Der Ablauf ist folgender: Nach dem Laden des `BGT`-Befehls aus Speicherstelle `$2006` bis `$2008` (der Operand ist diesmal 16 Bit breit, weswegen der gesamte Befehl 3 Bytes einnimmt) wird wie immer der Programmzähler erhöht. Er zeigt nun auf `$2009`. Der Befehlsdecoder prüft nun das Z- und N-Bit im Statusregister und weist, falls eines der beiden Eins ist, das Adressierwerk an, im Programmzähler die Adresse des nächsten Befehls durch die im Adreßteil gegebene Speicheradresse `$2003` zu ersetzen. Andernfalls geschieht einfach gar nichts.

Nach Beendigung des `BGT`-Mikroprogrammes steht also im Programmzähler entweder `$2003` oder `$2009`, je nach Wert des Statusregister. In unserem Fall war das Ergebnis der vorherigen Subtraktion (noch) größer Null, als nächstes wird also der an Adresse `$2003` stehende `INC B`-Befehl geladen.

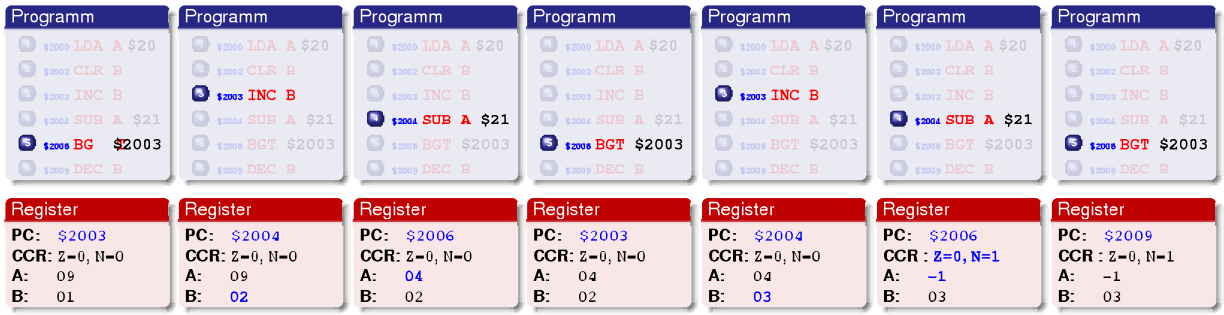


Abbildung 2.6: Schleifenablauf beim Beispielprogramm

Da das Programm anschließend von dort aus weitergeführt wird, ist das Resultat eine Art do-while Schleife. Die letzten drei Befehle (Erhöhe B um 1, vermindere A um 5, Springe zurück wenn A>0) werden so lange ausgeführt, bis der Wert von A schließlich nicht mehr größer Null ist und daher entweder das Z- oder das N-Bit im Statusregister gesetzt wurde). Die einzelnen Schritte dieses Ablaufes sind in Abbildung 2.6 gezeigt.

In den angegebenen Beispieldaten ist A zu Beginn des Programmes 14. Beim ersten Durchlauf wird A auf 9 reduziert, beim zweiten auf 4 und beim dritten schließlich auf -1. Nach der dritten Subtraktion wird also im CCR das N-Bit gesetzt und es findet kein Sprung mehr statt. Der Programmzähler wird nicht überschrieben und behält seinen nach dem Laden des Befehls ursprünglich zugewiesenen Wert `$2009`.

`DEC B` → Verringere den Wert von Register B um Eins An Adresse `$2009` steht im Speicher der Wert `5A`. Dieser Op-Code gehört zum `DEC B` Befehl. Er funktioniert exakt wie der bereits besprochene `INC B`-Befehl, nur daß das Steuerwerk diesmal an den zweiten Eingang der ALU eine -1 anlegt, und so den Wert des Registers B um eins verringert, anstatt ihn zu erhöhen.

Da die oben besprochene Schleife im Beispiel dreimal durchlaufen wurde und jedes Mal ein `INC B`-Befehl ausgeführt wurde, hat B den Wert 3 und wird durch diesen letzten Befehl auf 2 gesetzt. Das Programm ist damit beendet und Register B enthält das „Ergebnis" des Durchlaufs.

Dank moderner Prozessortechnologie wissen wir nun also, daß $\frac{14}{5}$ ganzzahlig 2 ist.

2.3 Interrupts

Im letzten Abschnitt wurde die Ausführung eines Programmes „am Stück" beschrieben. Was passiert jedoch, wenn der Prozessor zwischendurch auf plötzlich auftretende Ereignisse, wie beispielsweise einen Tastendruck reagieren muß? Das laufende Programm muß unterbrochen und das Ereignis behandelt werden.

Eine Möglichkeit wäre, alle potentiellen Quellen solcher Ereignisse (Peripheriecontroller, CPU-Komponenten etc.) in regelmäßigen Abständen abzufragen, ob Arbeit anliegt. Diese Vorgehensweise wird als *Polling* (von engl. to poll = abfragen) bezeichnet und ist sehr ineffizient, da die CPU einen Großteil ihrer Zeit mit solchen Abfragen verbringt. Man stelle sich nur vor, es gäbe keine Türklingel und man müßte in regelmäßigen Abständen nachsehen gehen, ob nicht eventuell jemand vor der Tür steht.

Eleganter ist es, wenn die Ereignisquelle selbst signalisiert, daß etwas passiert ist, indem sie einen *Interrupt Request* (IRQ) auslöst. Der Prozessor kann daraufhin das laufende Programm unterbrechen (einen sogenannten *Interrupt* auslösen) und sich um das Problem kümmern. Zu diesem Zweck wird eine *Interrupt-Serviceroutine* (ISR) ausgeführt. Das ist ein bestimmtes Programmstück, daß den Interrupt behandelt, also im Falle einer gedrückten Taste beispielsweise das Zeichen vom Tastaturcontroller einliest und zur späteren Verarbeitung in einen Puffer schreibt.

Man unterscheidet zwischen

- *Software-Interrupts* und
- *Hardware-Interrupts*, wobei sich diese wiederum in
  - *externe* Hardware-Interrupts und
  - *interne* Hardware-Interrupts

einteilen lassen.

Software-Interrupts werden vom Steuerwerk selbst aufgerufen und sind immer synchron zum Programmablauf. Soll beispielsweise ein unzulässiger Op-Code (ein Op-Code der im Befehlssatz nicht auftaucht und dem kein Mikroprogramm zugeordnet ist) ausgeführt werden, wird üblicherweise ein solcher Software-Interrupt ausgelöst um den Fehler zu behandeln. Am häufigsten werden Software-Interrupts jedoch explizit vom Programmierer aufgerufen, um bestimmte privilegierte Operationen nutzen zu können.

Hardware-Interrupts sind nicht synchron zum Programmablauf, da sie nicht vom Steuerwerk selbst sondern von anderer Hardware ausgelöst werden – und zwar zu beliebigen Zeitpunkten. Befindet sich die auslösende Hardware auf dem Chip, so handelt es sich um einen *internen Interrupt*. Ein solcher wäre beispielsweise ein Fehler in der ALU, wie z.B. eine Division durch Null. *Externe Interrupts* werden von Hardwarekomponenten ausgelöst, die sich nicht auf der CPU selbst befinden. Zu nennen wäre hier z.B der Tastaturcontroller, der das Drücken einer Taste signalisiert oder die Echtzeituhr (realtime clock, RTC) des Rechners, die meldet, daß schon wieder eine Millisekunde vorüber ist.

Wie funktioniert die Signalisierung?

Software-Interrupts sind synchron zum Programmablauf und werden vom Steuerwerk selbst ausgelöst. Da die Behandlung der Interrupts ebenfalls durch das Steuerwerk geschieht, ist eine separate Benachrichtigung desselben nicht notwendig.

Hardwarekomponenten signalisieren dem Steuerwerk einen Unterbrechungswunsch über speziell dafür vorgesehene Eingänge – die *Interrupt-Leitungen*. Die meisten Prozessoren haben allerdings nur ein oder zwei solcher Eingänge. Um beim Auftreten eines Interrupt-Requests feststellen zu können, von welchem Gerät er stammt, ist daher ein zusätzlicher Hardwarebaustein – der *Interrupt-Controller* – nötig.[9] Er nimmt die Unterbrechungswünsche der verschiedenen Geräte entgegen, sortiert sie nach Priorität und löst dann schließlich über die Interrupt-Leitung des Prozessors eine Unterbrechung aus. Anschließend übermittelt er dem Prozessor über den Datenbus eine ID, welche die Herkunft des Interrupts identifiziert.

Diese *IRQ-Vektornummer* muß nicht unbedingt eindeutig sein. Es können sich auch mehrere Geräte eine solche Kennung teilen (*IRQ sharing*). Der Prozessor muß dann allerdings beim Erhalt eines IRQs alle unter dieser Nummer registrierten Geräte kurz ansprechen, um festzustellen, von wem der Unterbrechungswunsch denn nun stammt.

Interrupts können im allgemeinen auch *maskiert* werden. Das bedeutet, daß die Interruptleitung hardwaremäßig (beispielsweise durch ein Flipflop mit nachgeschaltetem UND-Gatter) unterbrochen wird. Auf diese Weise kann sichergestellt werden, daß bestimmte kritische Programmabschnitte tatsächlich ohne Unterbrechung durchlaufen.

[9] Eine gewisse Berühmtheit hat hier der jahrelang als Interrupt-Controller für x86er-PCs eingesetzte **PIC 8259** erlangt. Inzwischen wird unter der Bezeichnung **APIC** ein moderneres System zur Interrupt-Verarbeitung genutzt.

Ablauf eines Interrupts

Tritt ein IRQ auf, wird zunächst einmal die Abarbeitung des gerade laufenden Befehls beendet. Eine Unterbrechung ist ohne Gefahr für die Konsistenz der Registerinhalte nämlich nur zwischen zwei Befehlen möglich. Bei Software-Interrupts erübrigt sich das Warten, da diese ja sowieso immer befehlssynchron ausgelöst werden.

Ist der aktuelle Befehl abgeschlossen, werden zunächst die Inhalte einiger Spezialregister (Programmzähler, Statusregister ...) in den Speicher kopiert, um später wieder an exakt der gleichen Stelle im Programm fortfahren zu können. Außerdem werden alle anderen Interrupts maskiert, um weitere Unterbrechungen zu verhindern.

Anschließend wird die Vektornummer des Interrupts bestimmt (siehe oben) und der Programmzähler mit der Startadresse der entsprechende Interrupt-Serviceroutine (ISR) geladen. Diese Einsprungadresse steht in der Interrupt-Vektortabelle (IVT), die sich an einer festgelegten Stelle im Speicher befindet. Da der Programmzähler immer die Adresse des nächsten Befehls enthält, wird nun nicht das normale Programm, sondern die ISR ausgeführt.

Hier werden nun alle zur Behandlung des Interrupts notwendigen Maßnahmen durchgeführt. Wird dabei auf Register zugegriffen, muß deren Inhalt vorher in den Speicher kopiert und anschließend wiederhergestellt werden. Die Register enthalten ja noch die Werte des gerade unterbrochenen Programms, die natürlich nicht verändert werden dürfen! Manche Prozessoren sichern daher vor dem Aufruf der ISR pauschal *alle* Register und stellen sie nach dem Ende der ISR wieder her. Das bedeutet jedoch einen sehr großen Overhead, was die Ausführungszeit eines Interrupts signifikant verlängert. Die meisten Architekturen verlassen sich daher auf die Umsicht des Programmierers und sichern nur die Spezialregister automatisch. Der Rest muß in der ISR „von Hand” erledigt werden.

Nach Beendigung der ISR wird der Programmzähler wieder mit seinem ursprünglichen Wert geladen und die Ausführung des unterbrochenen Programms wieder aufgenommen.

Kapitel 3

Optimierungen

Im letzten Kapitel wurde der Prozessor in seiner einfachsten Form beschrieben, so wie er bereits seit Jahrzehnten existiert. Die dargestellten Prinzipien und Abläufe sind dabei in weiten Teilen bis heute gültig. Natürlich ist die Entwicklung in den letzten Jahren nicht stehen geblieben und so wurde die grundlegende Architektur an zahlreichen Stellen erweitert und optimiert. Dieses Kapitel soll nun die wesentlichen Ansätze verdeutlichen, die dafür verwendet werden. Erst durch diese Optimierungen und die daraus resultierende kontinuierliche Beschleunigung der Prozessoren sind aktuelle Anwendungen mit ihrem enormen Bedarf an Rechenleistung überhaupt in den Bereich des Möglichen gerückt.

3.1 Höhere Taktfrequenz – Der einfache Weg

Der einfachste Ansatz um die Ausführung von Befehlen zu beschleunigen ist es, die Taktfrequenz des Prozessors zu erhöhen. Da mit jedem Takt ein Mikroschritt ausgeführt wird, bedeutet eine höhere Taktfrequenz natürlich auch mehr ausgeführte Befehle pro Sekunde. Daß sich auf diese Weise die Rechenleistung steigern läßt, zeigt die Gegenüberstellung in Abbildung 3.1. Dort ist links die Entwicklung der Taktfrequenz im Laufe der Jahre (1985-2005) und rechts der entsprechende SpecInt2000-Wert (ein Benchmark für die Rechenleistung) aufgetragen.

Zu beachten ist, daß die im SpecInt gemessene Rechenleistung natürlich noch von vielen anderen Faktoren abhängt und keinesfalls allein auf das Konto der höheren Taktfrequenz geht. Das ist schon daran zu erkennen, daß der Takt in den betrachteten 20 Jahren um etwa zwei Größenordnungen steigt, die Leistung jedoch um mehr als drei Größenordnungen. Die Taktfrequenz läßt sich jedoch nicht beliebig steigern. Das hat vor allem drei Gründe:

- die endliche *Ausbreitungsgeschwindigkeit* der Signale,
- die begrenzte *Schaltgeschwindigkeit* der Transistoren und die
- *Verlustleistung* der gesamten Schaltung.

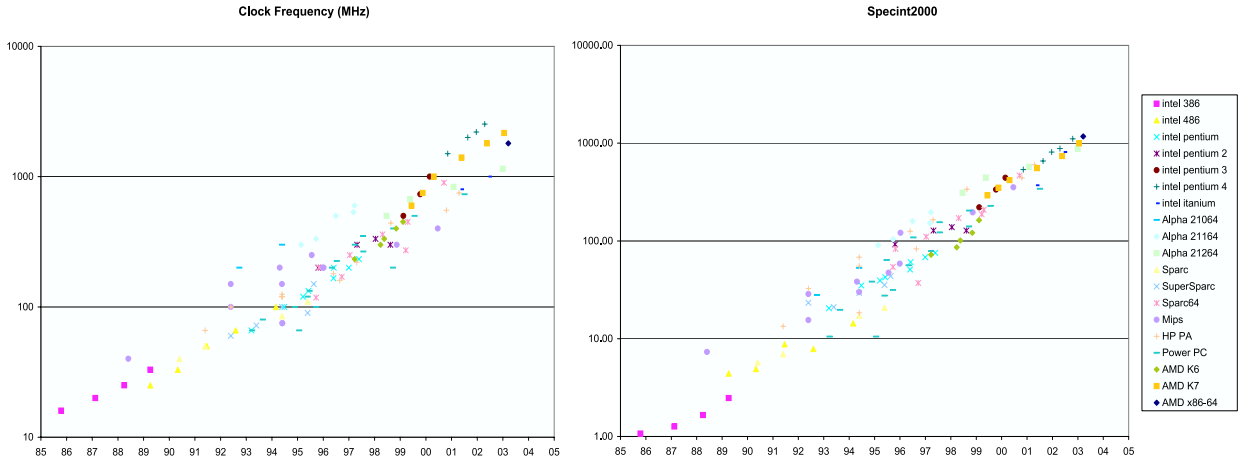


Abbildung 3.1: Zunahme von Taktfrequenz und Rechenleistung **Quelle:** [20]

Signale breiten sich im Chip typischerweise mit 50%-70% der Lichtgeschwindigkeit aus. Bei einer momentan durchaus möglichen Frequenz von 3 GHz dauert ein Takt nur noch 0,33 ns. In dieser Zeit legt das Signal gerade einmal 5 bis 7 cm zurück. Bei momentanen Chipgrößen mit Kantenlängen von maximal 1 bis 2 cm ist das noch nicht kritisch. Interessant wird es jedoch bei der äußeren Beschaltung. Komponenten die weiter entfernt vom Prozessor liegen (wie z.B. der Speicher), müssen zwangsläufig mit einer geringeren Frequenz arbeiten.

Die *Schaltgeschwindigkeit* der CMOS-Transistoren ist eine weitere Grenze für die Taktfrequenz. Sie hängt von der Integrationsdichte der Schaltung ab (je kleiner die Transistoren, desto schneller schalten sie) und vom Material des Halbleiters (Silizium, Galliumarsenid ...) ab.

Viel aktueller und dringender stellt sich jedoch das dritte Problem dar – die *Verlustleistung*. Ohne an dieser Stelle auf den genauen Aufbau eines CMOS-Gatters einzugehen, läßt sich sagen, daß die Verlustleistung P gemäß

$$P = C \cdot U_{Core}^2 \cdot f_{Takt} \tag{3.1}$$

linear mit der Taktfrequenz f_{Takt} ansteigt. Ein CMOS-Transistor der mit 2 GHz schaltet setzt also doppelt soviel Leistung um, wie einer der nur mit 1 GHz schaltet. Hinzu kommt die Tendenz, die Anzahl der Transistoren in modernen Prozessoren immer weiter zu erhöhen, so daß selbst kleine Erhöhungen des Pro-Transistor-Verbrauchs zu immer höheren Verlustleistungen führen.

Abbildung 3.2 zeigt die Entwicklung der Leistungsdichte bei Prozessoren. Zu beachten ist hierbei, daß nicht nur die Leistung pro Quadratmillimeter gestiegen ist, sondern die Chips wegen der Integration von immer mehr Transistoren trotz fortschreitender Miniaturisierung auch immer größer geworden sind.

Während also ein Intel 80386-DX-20 von 1987 mit seinen 20 MHz noch ca. 1,3 W Verlustleistung (2,8 W/mm^2) hatte, sind es bei dem auf Seite 4 abgebildeten Pentium 4 mit 2,5 GHz bereits 55 Watt (42 W/mm^2). [20]

Zum Vergleich: eine normale Herdplatte hat eine Leistungsdichte von 7,8 W/cm^2 und liegt damit deutlich darunter.

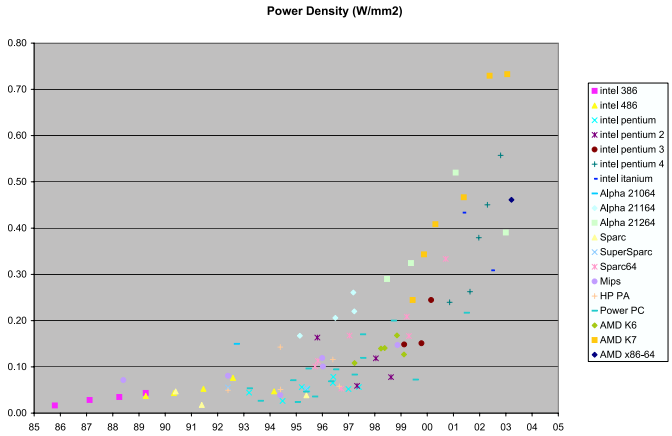


Abbildung 3.2: Zunahme der Verlustleistungsdichte **Quelle:** [20]

Dies hat zwei unerwünschte Nebenwirkungen

1. **der Prozessor wird heiß**

   Da die gesamte Verlustleistung in Wärme umgewandelt wird, der Prozessor aber gleichzeitig durch Temperaturen jenseits der 85°C zerstört wird, muß ein sehr großer Aufwand für die Kühlung betrieben werden. Bei Luftkühlung führt das (klobige Kühlkörper, schnell drehende Lüfter) zu einem hohen Lärmpegel.

2. **der Stromverbrauch steigt**

   Hohe Verlustleistung bedeutet selbstverständlich auch hohen Stromverbrauch. Dies ist besonders fatal bei mobilen Geräten, deren Akku-Kapazität ja beschränkt ist. Aber auch bei stationären Geräten ist ein hoher Stromverbrauch aus ökologischen und wirtschaftlichen Erwägungen unerwünscht.

Es existieren verschiedene Lösungen für die genannten Probleme. Gerade bei Notebooks und anderen mobilen Geräten ist es inzwischen üblich, die Taktfrequenz abzusenken, wenn der Prozessor nicht ausgelastet ist. Die entsprechenden Techniken heißen *Speed-Step* (Intel) oder *PowerNow!* bzw. *Cool'n'Quiet* (AMD).

Zudem ist man bestrebt, die Kernspannung U_{core} des Prozessors möglichst niedrig zu halten (und teilweise ebenfalls dynamisch abzusenken). Da diese quadratisch in die Verlustleistung eingeht (siehe Gleichung 3.1), ist der Spareffekt hier besonders hoch. Leider läßt sich U_{core} nicht beliebig absenken, da ansonsten – besonders bei hohen Frequenzen – keine eindeutig Unterscheidung von High- und Low-Pegeln mehr gewährleistet ist. Aktuelle Prozessoren arbeiten mit Kernspannungen zwischen 1,2 und 1,8 V.

Es ist auch nicht unbedingt nötig, Beschleunigungen allein durch Anheben der Taktfrequenz zu erzielen. In den folgenden Abschnitten werden einige andere Ansätze zur Steigerung der Rechenleistung vorgestellt.

3.2 Beschleunigung des Speicherzugriffs

Sowohl Befehle, als auch Operanden müssen aus dem Speicher des Systems gelesen, und die Ergebnisse wieder dorthin geschrieben werden. Dieser (meist als DRAM realisierte) Speicher ist aber sehr viel langsamer als der Prozessor. Bei 2 GHz Taktfrequenz dauert ein Prozessortakt nur noch 0,5 ns. Heutige SDRAM-Speicher haben aber Latenzzeiten von etwa 20 ns, extrem schnelle Module von 12 ns [3]. Der Prozessor muß also bei der Abarbeitung eines Befehls häufig Wartetakte einlegen, weil die Daten, die er für die Ausführung benötigt, noch nicht aus dem Speicher eingetroffen sind. Ein vielversprechender Ansatz zur Erhöhung der Rechenleistung ist es also, den Zugriff auf die benötigten Daten zu beschleunigen.

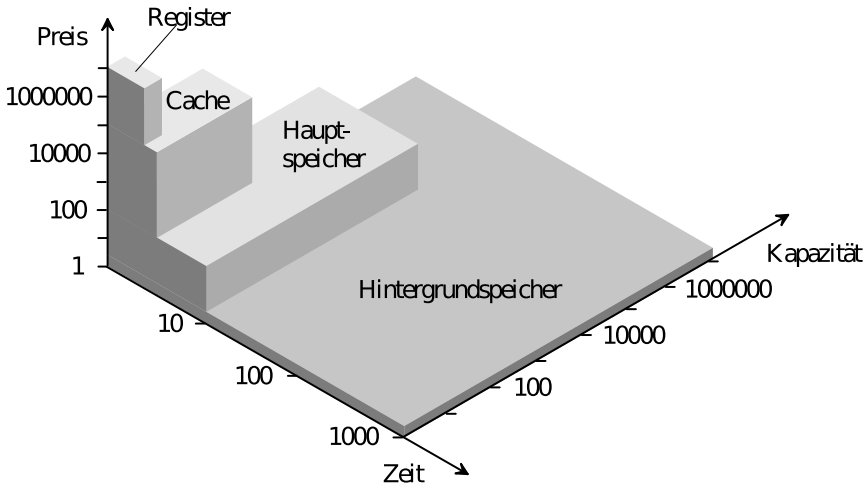


Abbildung 3.3: Vergleich verschiedener Speicherhierarchiestufen **Quelle: [11]**

Dies geschieht durch die Einführung einer Hierarchie aus unterschiedlich schnellen Speichern (siehe Abbildung 3.3). Die schnellsten Speicher sind die direkt auf dem Prozessor untergebrachten *Register*. Da sie nur wenige Bytes speichern können, werden die Daten im größeren, dafür aber auch langsameren *Hauptspeicher* (RAM) abgelegt. Dazwischen wird ein schneller Speicher – der *Cache* – eingefügt. In [18] findet sich eine sehr kompakte und trotzdem treffende Definition für einen Cache:

> „Ein Cache ist ein schneller Zwischenspeicher innerhalb der CPU oder in CPU-Nähe. Er soll Befehle und/oder Operanden von Programmen bereithalten, auf die in einem bestimmten Zeitraum häufig zugegriffen werden muß. Jeder Cache besteht aus einem Daten- und einem Tag-Bereich (Identifikationsbereich). Die Kapazität des Caches ist in der Regel deutlich kleiner als die Hauptspeicherkapazität. Der Tag-Bereich enthält den Teil der Adresse, der die Herkunft eines Cache-Eintrags aus dem Hauptspeicher eindeutig identifiziert. [...]"

Der Cache hält also häufig benutzte Daten vor, so daß diese schneller zur Verfügung stehen. Er ist meist aus schnellen SRAM-Zellen aufgebaut.

Man unterscheidet drei Arten. Der

First-Level-Cache (L1) ist direkt auf dem Prozessor untergebracht. Er hat eine Größe von etwa 8 bis 128 KB und ist meist in einen *Data-Cache* für die Operanden und einen *Instruction-Cache* für die Befehle eingeteilt. Der

Second-Level-Cache (L2) liegt entweder auch auf dem Prozessor-Die oder ist in unmittelbarer Nähe auf einem externen Modul untergebracht. Er ist langsamer als der L1, dafür allerdings auch wesentlich größer (256 KB bis 4 MB). Er kann größere Programmabschnitte puffern und damit die Ausführung der Befehle erheblich beschleunigen. Der

Third-Level-Cache (L3) ist hauptsächlich bei großen Server-Systemen zu finden. Er ist ein noch größerer, allerdings wiederum langsamerer Speicher, der als zusätzliche Hierarchiestufe eingefügt werden kann, um den Datenzugriff noch weiter zu beschleunigen.

Erst wenn die Daten in keinem der zwei bzw. drei Caches zu finden sind (*cache miss*), wird der langsame Hauptspeicher bemüht.

Das Hierarchie-Konzept ermöglicht es außerdem, den vorhandenen Hauptspeicher nahezu beliebig zu vergrößern, indem sogenannter *Hintergrundspeicher* d.h. Speicher von Festplattenlaufwerken oder aus dem Netzwerk hinzugenommen wird. Dieser ist natürlich wiederum mehrere Größenordnungen langsamer als der Hauptspeicher[1].

Bekannt ist diese Technik unter dem Namen *swaping*. Nicht benutzte Daten werden aus dem Hauptspeicher in eine Swap-Datei bzw. -Partition ausgelagert. Die Anwendung „sieht" trotzdem einen einzigen virtuellen Speicher von bis zu 4 GB Größe bei 32-Bit Prozessoren bzw. bis zu 16384 Petabyte bei 64-Bit Prozessoren (siehe Seite 13).

Bei der Zuordnung der virtuellen Speicheradressen zu realen Adressen (im Speicher oder auf der Festplatte) hilft in modernen Systemen eine extra Hardwarekomponente, die *Memory Management Unit* (MMU). Das Konzept der virtuellen Speicherverwaltung ist jedoch sehr komplex und liegt nicht ganz im Fokus dieser Ausarbeitung. Für einen tieferen Einblick sei daher lediglich auf [15, Abschnitt 4.3] und [22] verwiesen.

[1] DRAM-Zugriffszeiten werden in Nanosekunden (10^{-9}), Festplatten-Zugriffszeiten in Millisekunden (10^{-3}) gemessen!

3.3 Parallele Ausführung von Befehlen

Wenn die sequentielle Abarbeitung der Befehle nicht – oder nur unter Schwierigkeiten – weiter beschleunigt werden kann (siehe Abschnitt 3.1), ist der nächste logische Schritt, ein gewisses Maß an Parallelität einzuführen.

3.3.1 Pipeline-Architektur

Die Pipeline-Architektur wird auch als *Synchronparallele Organisation* bezeichnet. Man macht sich den Umstand zunutze, daß die auf dem Prozessor vorhandenen Funktionseinheiten bei der Abarbeitung eines Befehls kaum alle gleichzeitig benutzt werden.

Während beispielsweise das Systembus-Interface ein Befehlswort aus dem Speicher anfordert und das Befehlsregister dieses abspeichert, hat der Befehlsdecoder nichts zu tun, da noch kein zu dekodierender Befehl zur Verfügung steht. Auch die ALU ist noch nicht ausgelastet, da noch kein Befehl dekodiert wurde usw.

Bei der Pipeline-Architektur wird daher die Abarbeitung eines Befehls in mehrere Phasen eingeteilt, die dann – um jeweils eine Phase versetzt – quasi nebenläufig ausgeführt werden (Abbildung 3.4). Während also die ALU den ersten Befehl ausführt, wird der zweite bereits vom Befehlsdecoder dekodiert und der dritte schon vom Systembus-Interface ins Befehlsregister geladen.

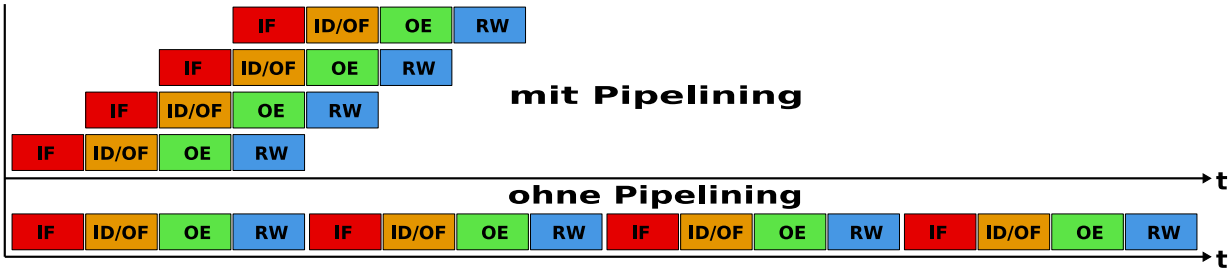


Abbildung 3.4: Vorteile der Pipeline-Architektur

Natürlich ist hierzu ein recht komplexes Steuerwerk nötig, daß all diese Vorgänge gleichzeitig koordinieren kann. Außerdem werden zusätzliche Busse als „Direktverbindung" zwischen den Komponenten benötigt. Die zusätzliche Komplexität lohnt sich jedoch. Wie in Abbildung 3.4 deutlich zu erkennen ist, benötigt die Ausführung von vier Befehlen bei synchronparalleler Organisation deutlich weniger Zeit als bei sequentieller Ausführung. Zumindest theoretisch wird mit jedem Takt ein Befehl fertig, auch wenn die einzelnen Befehle mehrere Takte für ihre Ausführung benötigen.

Wie die Einteilung in Ausführungsphasen genau vorgenommen wird, ist sehr unterschiedlich und hängt von der konkreten Prozessorarchitektur ab. Es gibt aber eine Art „klassische" Einteilung,

die in Lehrbüchern (z.B. [15] und [18]) immer wieder zitiert wird. Sie soll natürlich auch hier nicht fehlen:

Instruction Fetch (IF) Das Befehlswort wird aus dem Speicher (bzw. Cache) geholt und ins Befehlsregister geschrieben. Der Programmzähler wird erhöht.

Instruction Decode (ID) Der Befehl wird dekodiert.

Operand Fetch (OF) Das Adressierwerk wertet den Adreßteil des Befehlswortes aus und lädt die benötigten Operanden aus dem Speicher/Cache.

Operation Execute (OE) Das Rechenwerk führt den Befehl aus d.h. die Operanden werden miteinander verknüpft.

Result Write (RW) Das Ergebnis der Berechnung wird entweder in ein Register oder in den Speicher/Cache zurückgeschrieben.

Diese Einteilung ist nur ein grober Anhaltspunkt. Manche Architekturen besitzen vierstufige Pipelines und fassen das Dekodieren des Befehls und die Auswertung des Adreßteils (ID+OF) in einer Phase zusammen. Andere zerlegen das Rechenwerk in mehrere kleine Einheiten, die unabhängig voneinander arbeiten können und gliedern so die OE-Phase noch weiter auf.

Die dabei entstehenden, extrem langen Pipelines[2] bringen allerdings auch Probleme mit sich. Immer dann, wenn Wartetakte anfallen oder das berechnete Ergebnis eines Befehls Voraussetzung für die Ausführung eines folgenden Befehls ist (sog. *Datenabhängigkeiten*), kann die Pipeline nicht optimal ausgelastet werden.

Schlimmer noch, wenn das Programm plötzlich verzweigt. Dann muß der Inhalt der Pipeline mit all den bereits geladenen, möglicherweise schon dekodierten oder sogar teilweise ausgeführten Befehlen verworfen, und die Pipeline neu gefüllt werden. Aus diesem Grund ist man bemüht, Sprünge durch immer aufwändigere Verfahren, die unter dem Schlagwort *branch prediction* zusammengefaßt werden, möglichst präzise vorauszusagen. Näheres dazu ist in [18, Abschnitt 4.4.3] und [11, S.78 ff] zu finden.

3.3.2 Superskalare Prozessoren

Die nächst höhere Stufe der Parallelität stellen superskalare Prozessoren dar. Auch hier werden die Befehle in einer Pipeline abgearbeitet. Es stehen jedoch mehrere Funktionseinheiten (ALUs etc.) für die Ausführung zur Verfügung.

Es werden mehrere Instruktionen in einem Takt geladen, dekodiert und anschließend auf die Ausführungseinheiten verteilt. Das ermöglicht es, mehrere Befehle tatsächlich nebenläufig (und

[2] der Intel Pentium 4 hat 20 (Northwood-Kern) bzw. 31 (Prescot-Kern) Pipelinestufen

nicht nur, wie in der Pipeline-Architektur, um je eine Phase versetzt) auszuführen. Diese Parallelität auf Instruktionsebene (auch als *Instruction Level Parallelism* (ILP) bezeichnet) führt dann dazu, daß der Prozessor tatsächlich mehr als 1 Befehl/Takt ausführen kann. Der Befehlsfluß einer superskalaren Architektur ist in Abbildung

Programmverzweigungen sind weiterhin ein Problem. Deshalb wird schon beim Laden der Befehle versucht, Sprungziele vorherzusagen und dadurch „die richtigen" Befehle zu laden. Falsch prädizierte Sprünge machen sich bei superskalaren Architekturen sogar noch viel störender bemerkbar als beim einfachen Pipelining, da hier meist eine ganze Reihe von bereits fertig ausgeführten Ergebnissen verworfen werden muß, was einen starken Performance-Einbruch bedeutet.

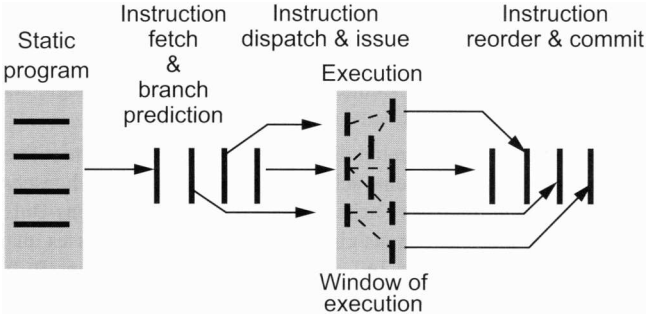


Abbildung 3.5: Befehlsfluß einer superskalaren Architektur **Quelle:** [18]

Das Dekodieren der einzelnen Befehle und die Verteilung der Arbeit auf die verschiedenen Funktionseinheiten erledigt der sogenannte *Dispatcher*. Er versucht dabei auch, Datenabhängigkeiten so gut wie möglich zu berücksichtigen, was natürlich nicht immer vollständig gelingt. Je breiter das *Ausführungsfenster* (window of execution) ist, desto mehr Befehle können parallel verarbeitet werden, desto komplexer wird aber auch der Dispatcher.

Am Schluß werden die Befehle (bzw. deren Ergebnisse) wieder in eine gültige Reihenfolge sortiert. Befehle, die aufgrund von falsch prädizierten Sprüngen zwar ausgeführt wurden, so aber nicht im tatsächlichen Programmablauf vorgesehen waren, werden dabei verworfen. Diese Aufgabe übernimmt die *Commit-Unit*.

3.3.3 Simultaneous Multi Threading (SMT)

Treibt man die Parallelisierung der Befehlsabarbeitung noch weiter, landet man beim *Simultaneous Multi Threading*[3]. Hier werden nicht nur mehrere Verarbeitungseinheiten auf einem Chip untergebracht wie bei superskalaren Architekturen, auch Registersatz, Befehlsdecoder, Programmzähler usw. werden dupliziert.

Auf diese Weise erscheint ein SMT-fähiger Prozessor dem Betriebssystem wie zwei unabhängig voneinander arbeitende Prozessoren. Es ist aber trotzdem nur ein einziger Prozessor, der sozusagen zwei Pipelines besitzt.

[3]Etwas bekannter ist SMT unter dem Marketing-Namen *Hyper-Threading*, den Intel Anfang 2002 geprägt hat.

Das ermöglicht es, die ohnehin mehrfach vorhandenen Verarbeitungseinheiten moderner Prozessoren besser auszunutzen, indem mehrere voneinander unabhängige Ausführungsstränge (*threads*) eines Programmes gleichzeitig abgearbeitet werden können. Diese Art der Parallelverarbeitung wird daher auch als *Thread Level Parallelism* (TLP) bezeichnet.

3.3.4 Multicore Prozessoren

Die höchste denkbare Stufe der Parallelität stellen Multicore-Prozessoren dar. Anders als bei SMT-Prozessoren sind hier zwei (oder mehr) v*ollkommen voneinander unabhängige* Prozessorkerne auf einem Die untergebracht (siehe Abbildung 3.6).

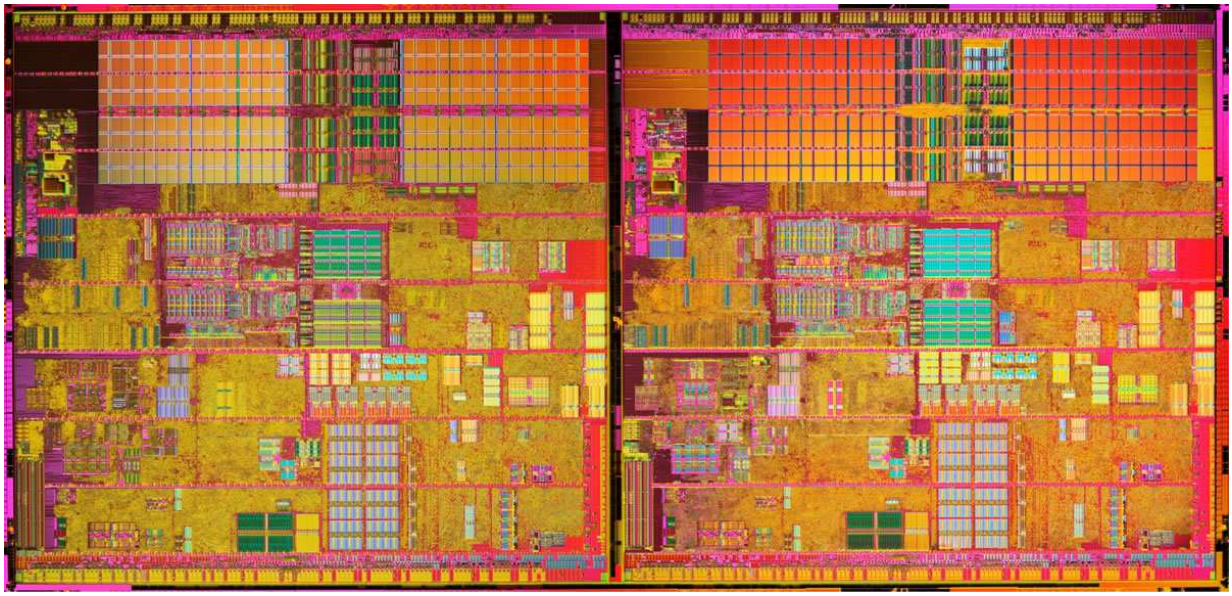


Abbildung 3.6: Dualcore-Die (Intel Pentium D „Smithfield") **Quelle:** [8]

Im Gegensatz zu herkömmlichen Mehrprozessorsystemen (SMP), bei denen mehrere separat gesockelte CPUs auf einem Mainboard untergebracht sind, reicht bei Multicore-Prozessoren ein einziger Sockel und ein einziges (ggf. etwas größer dimensioniertes) Kühlsystem aus. Auf diese Weise kann Platz und vor allem Geld gespart werden.

Die Idee der Multicore-Systeme ist nicht neu. Entsprechende Prozessoren sind jedoch erst seit kurzem auf dem Markt bzw. angekündigt. [8]

3.4 Spezielle Befehlssatz-Erweiterungen

Eine weitere Beschleunigung von Prozessoren kann die Erweiterung des Prozessor-Befehlssatzes bringen. Für einige besonders häufig benutzte Operationen werden spezielle Befehle hinzugefügt, die speziell für diese Aufgabe optimierte Ausführungseinheiten nutzen.

Insbesondere bei der Verarbeitung von Multimediadaten ist das sinnvoll. Hier tritt beispielsweise oft der Fall ein, daß eine einzige Operation (z.B. eine Multiplikation) auf viele gleichartige Daten angewendet werden muß. Dieses Verfahren bezeichnet man als *SIMD* (**S**ingle **I**nstruction **M**ultiple **D**ata). Es ist in gängigen Befehlssatzerweiterungen heutiger CPUs implementiert. [9] Das sind z.B:

- *MMX*[4] und deren Nachfolger **SSE 1**, **2** und **3**[5] von Intel,
- *3Dnow!* von AMD und
- *AltiVec* von IBM (PowerPC-Prozessoren)

Neben der – unterschiedlich gut implementierten [14] – SIMD-Funktionalität berücksichtigt auch die sogenannte *Sättigungsarithmetik* die Belange von Multimediadaten.

Hierzu ein Beispiel:

Eine Grafik soll als Matrix aus 8 Bit breiten Grauwerten (0: schwarz; 255: weiß) dargestellt werden. Um jetzt die Helligkeit des Bildes zu erhöhen, wird auf jeden Pixel ein konstanter Wert (z.B. 10) aufaddiert.
Ist ein Pixel aber schon recht hell, beispielsweise 250, so führt eine Addition mit 10 zu einem Überlauf. Der Pixel hat nun den Wert 4, was einem sehr dunklen Schwarz entspricht. In diesem Fall wäre es besser gewesen, wenn der Wert beim Erreichen der oberen (bzw. unteren) Grenze nicht weiter verändert wird, dort also in die Sättigung geht. Genau das berücksichtigt die Sättigungsarithmetik.

Sättigungsarithmetik und SIMD sind nur einige der möglichen Spezialfälle, die durch Befehlssatzerweiterungen abgedeckt werden können. Zwar muß man dabei immer im Auge behalten, daß ein ausufernd komplexer Befehlssatz den Entwurf und auch die Programmierung des Chips erschwert. Trotzdem liegt hier viel Potential für Optimierungen.

Ein Problem bei zusätzlichen Befehlen ist jedoch, daß die Software diese auch verwenden muß. Programme und Compiler müssen also speziell dafür angepaßt werden. Besonders bei weniger weit verbreiteten Erweiterungen ist das oft nicht der Fall und all die schönen Features bleiben ungenutzt.

[4] MMX**: M**ulti**m**edia E**x**tension
[5] SSE**: S**IMD **S**treaming **E**xtension

Kapitel 4

Zukünftige Möglichkeiten

Trotzdem diese Ausarbeitung sich – wie es der Titel schon andeutet – eigentlich nur mit dem Aufbau und der Funktionsweise von Mikroprozessoren befaßt, kann ein Blick in die Zukunft sicherlich spannend sein. Im letzten Kapitel wird daher kurz auf mögliche Zukunftsszenarien der Prozessortechnologie eingegangen.

4.1 Polymer-Prozessoren

Mit der Entdeckung daß Plastik unter bestimmten Umständen Halbleitereigenschaften besitzt, begann eine faszinierende Entwicklung. Der bisher überwiegend für Leuchtdioden und Displays (OLEDs) genutzte Effekt könnte sich auch für die Herstellung von Prozessorstrukturen eignen.

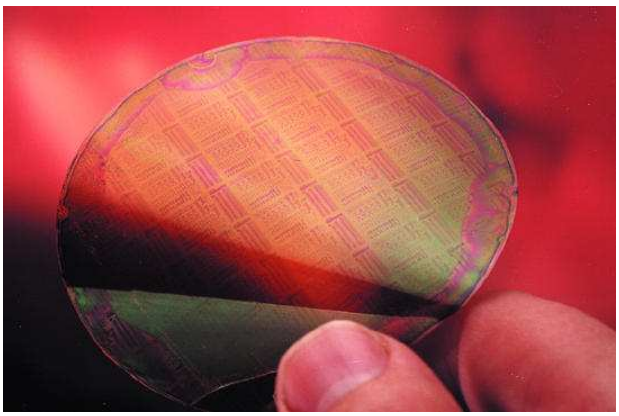


Abbildung 4.1: 15-Bit Codegenerator auf einem 3”-Wafer aus Polyamid **Quelle: [21]**

Diese Prozessoren wären sehr leicht, biegsam und außerdem preiswert in der Massenproduktion. Es wäre sogar möglich, die Schaltstrukturen in einem dem Tintenstrahldruck ähnlichen Verfahren

aufzubringen und so die Fertigung wesentlich zu vereinfachen University of Cambridge, OE-FET-Research Group [25].

Diesen Vorteilen stehen jedoch noch einige ungelöste Probleme gegenüber. Die Ladungsträgerbeweglichkeit in den momentan verwendeten Materialien ist noch sehr gering. Dadurch bedingt sind lange Schaltzeiten. Die momentan schnellsten Plastik-Transistoren schalten mit maximal 75μs, was einer maximalen Betriebsfrequenz von 13kHz entspricht. Der in Abbildung 4.1 gezeigte Chip [13] ist sogar noch langsamer. Momentan wird allerdings noch sehr intensiv an unterschiedlichen Materialien geforscht, die schnellere Schaltvorgänge ermöglichen.

Auch wenn Polymer-Prozessoren möglicherweise nie so schnell werden wie ihre Pendants auf Siliziumbasis, so eröffnen sich dennoch interessante Anwendungsgebiete. Biegsame Prozessoren könnten in Kleidung eingenäht werden oder – in Verbindung mit ebenfalls biegsamen OLED-Displays – den Traum vom elektronischen Papier wahr machen.

4.2 Prozessoren mit supraleitenden Elementen (RSFQ)

Wie in Abschnitt 3.1 dargelegt gehen bei herkömmlichen Prozessoren hohe Taktfrequenzen und die daraus resultierenden Ströme stets mit einer hohen Verlustleistung einher, die in thermische Energie – also Wärme – umgewandelt wird.

Durch supraleitende Materialien und eine völlig andere Art der Darstellung der logischen Zustände in einem Prozessor, läßt sich diese Verlustleistung drastisch vermindern.

Die sogenannten Einzelflußquanten-Bauelemente (Single Quantum Flux, SQF) bestehen aus supraleitenden Materialien und nutzen zur Informationsverarbeitung keine konstanten Spannungspegel mehr, sondern sehr kurze Spannungspulse. Die Bezeichnung Einzelflußquanten-Bauelemente tragen diese Schaltungen, da der bei der Integration dieser Spannungsverläufe entstehende Wert nur noch vom sogenannten magnetischen Flußquantum,

$$\Phi_0 = \frac{h}{2e} = 2,07 \cdot 10^{-15} Vs$$

einer Naturkonstante, abhängt. Die Schaltungstechnik, die auf dieser Art der Signale basiert, wird inzwischen als RapidSFQ- bzw. RSFQ-Technik bezeichnet. Eine sehr schöne Einführung findet sich in [12].

Die RSFQ Technik macht sich den quantenphysikalischen Josephson-Effekt, weshalb die als Bauelemente verwendeten Strukturen auch Josephson-Kontakte heißen. Eine Erläuterung des technischen Hintergrunds würde hier zu weit führen. Leicht verständliche Darstellungen dazu sind jedoch in [4] und [24] zu finden.

Der Entwurf von RSFQ-Schaltungen ist sehr komplex, da die Eigenschaften der verwendeten supraleitfähigen Materialien stark von der Geometrie der Bauelemente und von der Betriebstemperatur abhängen.

Trotzdem existiert bereits ein erster 8-Bit Mikroprozessor, der aus etwa 63.000 Josephson-Kontakten besteht. Der FLUX-1 genannte Prozessor [16] soll bei einer Taktfrequenz von 24 GHz eine Verlustleistung von nur 9,2 mW (!) haben.

Ein – zugegebenermaßen komplexerer – Pentium 4 produziert bei einem Zehntel dieser Frequenz mit seinen 55 W fast 6000 mal soviel Verlustleistung.

4.3 Weitere Möglichkeiten

Sicherlich gibt noch viele weitere sehr interessante Entwicklungen im Bereich der der Rechnertechnik.

Der *Quantencomputer* zum Beispiel ist eine reale, wenn auch noch äußerst ferne Vision. Er nutzt die superponierten Quantenzustände einzelner Teilchen zur Informationsverarbeitung und kann so viele Operationen gleichzeitig ausführen.

Auch die *Optische Datenverarbeitung* bietet viel Potential für die Zukunft, besonders wenn man bedenkt, daß die Übertragung der Daten ja schon heute weitestgehend optisch geschieht. Eine entsprechende rein optische Weiterverarbeitung ist im Moment jedoch noch Zukunftsmusik.

Leider können die vielen verschiedenen Ansätze hier nicht alle erläutert werden. Es bleibt jedoch spannend. Welche Technologie sich am Ende durchsetzen wird, kann nur die Zukunft zeigen ...

Abbildungsverzeichnis

1.1 Entwicklungsstufen der Prozessoren **Quelle: [20]** 4
1.2 Prinzipieller Aufbau eines Rechners 5
1.3 von Neumann-Architektur 7
1.4 Harvard-Architektur 8

2.1 Funktionsgruppen eines Prozessors 12
2.2 Register 12
2.3 Schematische Darstellung einer 1-Bit ALU **Nach** [17, S.178] 15
2.4 Aufbau eines CISC-Steuerwerkes 18
2.5 Tri-State-Ankoppelung von zwei Geräten an eine gemeinsame Datenleitung . . . 20
2.6 Schleifenablauf beim Beispielprogramm 24

3.1 Zunahme von Taktfrequenz und Rechenleistung **Quelle:** [20] 29
3.2 Zunahme der Verlustleistungsdichte **Quelle:** [20] 30
3.3 Vergleich verschiedener Speicherhierarchiestufen **Quelle: [11]** 31
3.4 Vorteile der Pipeline-Architektur 33
3.5 Befehlsfluß einer superskalaren Architektur **Quelle:** [18] 35
3.6 Dualcore-Die (Intel Pentium D „Smithfield”) **Quelle:** [8] 36

4.1 15-Bit Codegenerator auf einem 3”-Wafer aus Polyamid **Quelle: [21]** 38

Literaturverzeichnis

[1] *IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, 1985 (IEEE 754)

[2] AltiVec. In: *Wikipedia-Enzyklopädie* 22.04.2005 22:05 (2005). – URL `http://de.wikipedia.org/wiki/AltiVec`

[3] Double Data Rate Synchronous Dynamic Random Access Memory. In: *Wikipedia-Enzyklopädie* 09.07.2005 16:27 (2005). – URL `http://de.wikipedia.org/wiki/DDR-SDRAM`

[4] Josephson-Effekt. In: *Wikipedia-Enzyklopädie* 04.06.2005 14:29 (2005). – URL `http://de.wikipedia.org/wiki/Josephson-Effekt`

[5] Motorola 6800. In: *Wikipedia-Enzyklopädie* 06.07.2005 17:27 (2005). – URL `http://de.wikipedia.org/wiki/Motorola_6800`

[6] Reduced Instruction Set Computing. In: *Wikipedia-Enzyklopädie* 23.05.2005 12:53 (2005). – URL `http://de.wikipedia.org/wiki/Reduced_Instruction_Set_Computing`

[7] BEIER, Andreas ; STILLER, Andreas: Apple fällt vom Stamm. In: *c't-Magazin für Computer und Technik* 13 (2005), Jun, S. 18 ff.

[8] BENZ, Benjamin: Doppelkopf. In: *c't-Magazin für Computer und Technik* 15 (2005), Feb, S. 88 ff.

[9] BLEUL, Andreas: Vektorarchitekturen aktueller Prozessoren im Vergleich. In: *c't-Magazin für Computer und Technik* 4 (2000), Feb, S. 341 ff.

[10] BÄHRING, Helmut: *Mikrorechnersysteme*. 2. Auflage. Springer Verlag, 1994

[11] BÖTTCHER, Axel: *Foliensatz zur Vorlesung Rechnertechnik Wintersemester 2004/2005*. Fachhochschule München, 2004

[12] FELDMAN, Marc J.: Digital Applications of Josephson Junctions. In: *Physics and Applications of Mesoscopic Josephson Junctions* (1999), S. 289–304. – URL `http://www.ece.rochester.edu/users/sde/research/rsfq/index.html`

[13] HOFSTRAAT, Hans: *Organic Electronics: From Science To Application*. Philips Research Laboratories. – URL http://www.iee.org/oncomms/sector/electronics/Articles/Download/2E2715F%3-275F-4F85-BB2E6582F7C901A5

[14] MEYER-SPRADOW, Jennis: Großspurig - Ein kritischer Blick auf MMX. In: *c't-Magazin für Computer und Technik* 1 (1997), Jan, S. 228 ff.

[15] ORGLMEISTER, Reinhold: *Analog- und Digitalelektronik*. Technische Universität Berlin, 2002

[16] P. BUNKY ET AL.: FLUX-1 RSFQ-Microprocessor: Physical Design and Test Results. In: *Applied Superconductivity Conference*. Houston, 2002

[17] SIEH, Volkmar: *Skript Technische Informatik II*. Universität Erlangen-Nürnberg, Jun 2004. – URL http://www3.informatik.uni-erlangen.de/Lehre/OTRS_IV/SS2004/skript.html

[18] SIEMERS, Christian: *Prozessor-Technologie*. tecChannel (IDG Interactive GmbH), 2004

[19] SPEISER, Ambros: Wer hat den Computer erfunden? In: *Neue Züricher Zeitung* 20. Nov. (1996). – URL http://www.math.ethz.ch/undergraduate/lectures/ws0405/other/linalg_INFK%/ge.pdf

[20] STANFORD UNIVERSITY, VLSI-RESEARCH GROUP: *Microprocessors through the ages*. Jun 2005. – URL http://velox.stanford.edu/group/chips_micropro.html

[21] STIELER, Wolfgang: Aus dem Reagenzglas - Plastik wird die Computertechnik verändern. In: *c't-Magazin für Computer und Technik* 2 (1999), Jan, S. 76–81

[22] STILLER, Andreas: Speicherschieber. In: *c't-Magazin für Computer und Technik* 3 (2000), Feb, S. 260 ff.

[23] TIETZE, Ulrich ; SCHENK, Christoph: *Halbleiter-Schaltungstechnik*. 11. Auflage. Springer Verlag, 1999

[24] TOEPFER, Hannes ; STIELER, Wolfgang: Coole Visionen. In: *c't-Magazin für Computer und Technik* 25 (2002), Jul, S. 170 ff.

[25] UNIVERSITY OF CAMBRIDGE, OE-FET-RESEARCH GROUP: *Research Projects*. Jun 2005. – URL http://www-oe.phy.cam.ac.uk/fet/research.htm