Inhaltsverzeichnis

Kapitel 1

Befehlssatzarchitekturen

Kapitelinhalt

„Ich denke, dass es weltweit einen Markt für vielleicht fünf Computer gibt.“
Thomas Watson (Chairman von IBM) zugeschrieben, 1943

Dieses zwar nicht nachgewiesene, aus damaliger Sicht aber durchaus denkbare Zitat hat sich glücklicherweise als kompletter Irrtum herausgestellt. Computer haben heute mehr denn je eine umfassende und vielfältige Verbreitung im Alltag und werden diese auch in den kommenden Jahren weiter ausbauen. Im vorherigen Kurs Computersysteme I (1608) haben Sie bereits die Grundlagen für das Verständnis von elektronischen Schaltungen kennen gelernt, aus denen jeder Computerchip aufgebaut ist. Auch haben Sie erste Erkenntnisse gesammelt, wie ein Computer prinzipiell funktioniert[1] und aus welchen Funktionseinheiten ein Prozessor besteht. Bevor wir daran anschließen und zeigen, wie ein Prozessor die Befehle im Detail abarbeitet, werden wir in diesem Kapitel zunächst die Sicht eines Assemblerprogrammierers bzw. Compilers übernehmen. Hierbei wird der Prozessor von außen betrachtet und untersucht, welche Funktionalität der Befehlssatz eines Prozessors dem Programmierer typischerweise bietet.

Abschnitt 1.1 führt die in diesem Kapitel angestrebten Lernziele auf. Abschnitt 1.2 dient als Gesamtüberblick über die im Kurs behandelten Inhalte. Abschnitt 1.3 führt in die Grundlagen des Befehlssatzes ein, d. h. die verschiedenen Befehlsformate und -typen, Adressierungsarten, Datenformate, Adressräume usw. In Abschnitt 1.4 und Abschnitt 1.5 werden Sie anschließend den Befehlssatz eines konkreten Prozessors kennenlernen und mit diesem praktische Übungen durchführen.[2]

1.1 Lernziele

Dieses Kapitel führt in die Grundlagen der Befehlssatzarchitektur ein. Sie werden dabei kennenlernen, welche Funktionen und Befehle ein Prozessor typischerweise nach außen bietet, ohne dabei auf die interne technische Realisierung einzugehen. Sie werden lernen, wie und wo Daten bzw. Programmcode im Computer gespeichert bzw. adressiert werden, wie typische Datenformate aussehen, welche Befehlsarten und Befehlsformate es gibt und wie sich verschiedene Architekturen voneinander unterscheiden können. Außerdem werden Sie lernen, wie man Assembler-Programme für einen Prozessor schreiben und analysieren kann. Hierfür werden Sie sich mit dem Befehlssatz eines Beispiel-Prozessors auseinandersetzen und so die zunächst theoretischen Inhalte auch praktisch wiederholen.

[1] Von-Neumann-Prinzip mit Steuerwerk, Rechenwerk, Speicher und Ein-/Ausgabe.
[2] Bzw. in einem Simulator für diesen Prozessor.

1.2 Einführung

Im letzten Kapitel des Kurses 1608 haben Sie gelernt, wie ein Computer nach dem von-Neumann-Prinzip aufgebaut ist. Er besteht aus vier Funktionseinheiten: dem Rechen- und Leitwerk, die zusammen den Prozessor bilden, dem Hauptspeicher und der Ein-/Ausgabe. Das Zusammenspiel dieser vier Komponenten bestimmt die Leistungsfähigkeit eines Computers. Speicher und Ein-/Ausgabe werden über die Systembusschnittstelle mit dem Prozessor verbunden und belegen jeweils verschiedene Bereiche des ansprechbaren Adressraumes. Der Aufbau des Speichersystems und dessen Einfluss auf die Leistungsfähigkeit eines Computers wird im Kapitel 3 ausführlich behandelt. Ein-/Ausgabeeinheiten bestehen im Allgemeinen aus spezialisierten Controller-Bausteinen, die Peripheriegeräte zur Interaktion mit dem Menschen (Tastatur, Monitor, Maus, Drucker), anderen Computern (lokale Netzwerke, Internet) oder nicht-flüchtigen Speichermedien (Festplatten, optischen Speichern) unterstützen. Die Controller-Bausteine werden ähnlich wie Speicher über einen speziellen Adressbereich angesprochen.

Entscheidenden Einfluss auf die Leistung und die Einsatzmöglichkeiten eines Computers hat das Programmiermodell des Prozessors. Es beschreibt die Sicht eines Systemprogrammierers auf den Prozessor und beinhaltet alle notwendigen Details, um ablauffähige Maschinenprogramme für diesen zu erstellen. Ebenso muss auch der Übersetzer das Programmiermodell kennen, um Hochsprachenprogramme in die entsprechende Maschinensprache eines Prozessor zu übersetzen. Da das Programmiermodell im Wesentlichen durch den Befehlssatz des Prozessors festgelegt ist, spricht man von der Befehlssatzarchitektur (Instruction Set Architecture, ISA) oder auch nur von der *Architektur* eines Prozessors. Die Architektur beschreibt dabei lediglich das Verhalten des Prozessors, nicht aber seine konkrete Implementierung. Diese hängt u. a. von der logischen Organisation und dem internen Zusammenschluss der verschiedenen Funktionseinheiten und Bausteine ab, siehe Abbildung 1.1.

Befehlssatz-architektur

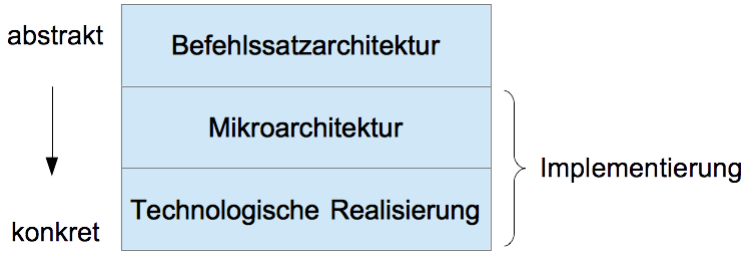

Abbildung 1.1: Ebenen der Rechnerarchitektur

Ähnlich wie ein Architekt zunächst mit dem Bauherrn die gewünschten Eigenschaften eines Bauwerks festlegt und diese bestmöglich auf die spätere Nutzung abstimmt (z. B. Anzahl, Größe, Ausstattung und Anordnung der benötigten Räume), geht auch ein Rechnerarchitekt systematisch an den Entwurf eines Prozessors heran. Zunächst wird die Befehlssatzarchitektur des Prozessors festgelegt. Sie beschreibt die für die spätere Anwendung benötigten prozessorinternen

Speichermöglichkeiten (Register), Operationen, Datenformate und Unterbrechungslogik. Aus dieser Architekturbeschreibung leitet der Rechnerarchitekt dann eine logische Struktur zur Implementierung ab, die *Mikroarchitektur* genannt wird. Dabei legt er fest, welche Funktionseinheiten (z. B. Register(sätze), ALUs[3], Multiplexer usw.) benutzt werden sollen, welche Datenpfade (*Data Path*) zwischen den Funktionseinheiten vorhanden sein sollen und wie alle diese Komponenten koordiniert werden (*Control Path*). Die Umsetzung dieser logischen Organisation in einer bestimmten Hardware-Technologie bezeichnet man als technologische Realisierung (in Analogie zu einem Bauwerk wären dies die verwendeten Baumaterialien).

Mikroarchitektur

Im Laufe der Entwicklung von Computersystemen kamen verschiedene Technologien zur Realisierung von Prozessoren zum Einsatz. Es begann mit elektromechanischen Bauelementen zu Beginn des 20. Jahrhunderts, welche später durch Elektronenröhren und im Anschluss daran durch einzelne Transistoren ersetzt wurden. Ab dem Ende der fünfziger Jahre wurden schließlich integrierte Schaltkreise (*Integrated Circuits, ICs*) entwickelt, welche komplette Schaltungen mit mehreren Transistoren auf einem einzigen Halbleiterchip realisieren. Die Integrationsdichte[4] hat sich dabei seit der Einführung von Integrierten Schaltkreisen stetig gesteigert. Gordon Moore, Mitbegründer des weltweit bekannten Prozessorherstellers Intel, stellte ab 1965 in einem Beitrag der Zeitschrift „Electronics“ fest, dass sich bis zu diesem Zeitpunkt die Anzahl der Transistoren pro Chip jedes Jahr verdoppelt hat. Dieser als *Mooresches Gesetz* bekannt gewordene Zusammenhang hat sich prinzipiell bis heute fortgesetzt. Lediglich die „Zeitkonstante“ muss etwas größer gewählt werden. Sie beträgt bei Speicherchips mit regulärer Struktur etwa 18 Monate, bei Prozessoren wegen der erhöhten Komplexität etwa 24 Monate. Es ist zu beachten, dass es sich hierbei nicht um ein Gesetz im Sinne eines Naturgesetzes handelt, sondern um einen aus einer empirischen Untersuchung abgeleiteten Zusammenhang. Auf zukünftige Entwicklungen in der Chipfertigung können daher keine Aussagen getroffen werden.

Integrierter Schaltkreis

Mooresches Gesetz

Um eine logische Organisation in Hardware umzusetzen, müssen mehrere Schichten geometrischer Strukturen (*Chip Layouts*) auf eine Halbleiterscheibe (*Wafer*) geätzt werden. Die elektronischen Funktionseinheiten setzen sich dann aus diesen geätzten Strukturen zusammen. Im Laufe der Jahre wurde die Verfahrenstechnik zur Herstellung solcher Mikrochips stetig verbessert, siehe Abbildung 1.2. Heute (Stand 2017) ist man in der Lage, Strukturen von weniger als 14 Nanometern herzustellen (Ein Nanometer entspricht 10^{-9} m, also einem Millionstel Millimeter). Durch die feiner werdenden Strukturen ist es möglich, immer mehr Transistoren bei gleichbleibender Fläche auf einem Chip unterzubringen und gleichzeitig die Taktfrequenz zu erhöhen. Wegen der kleineren Strukturen können die Transistoren schneller schalten und durch kürzere Verbindungsleitungen miteinander gekoppelt werden. Dadurch verkürzen sich auch die Laufzeiten zwischen den Transistoren, was eine Erhöhung der Taktfrequenz

Strukturgrößen

[3] Arithmetisch-logische Einheit, engl. *arithmetic logic unit*

[4] Die Integrationsdichte gibt die Anzahl an Transistoren pro Flächeneinheit an.

realisierbar macht.

Mikrochips lassen sich seit der Entstehung in verschiedene Generationen unterteilen, siehe Tabelle 1.1. Während man bei der SSI-Technologie mit Strukturgrößen von um die 100 Mikrometer gerade mal 100 Transistoren auf einem Mikrochip integrieren konnte, sind heute (Stand 2017) mit fortgeschrittenen GSI-Technologien bei Integrationsdichten von ca. 14 Nanometern 7,2 Milliarden Transistoren pro Chip realisierbar[5]. Hauptproblem zukünftiger Entwicklungen ist vor allem die Kühlung der Mikrochips, deren Wärmedichte die einer Herdplatte bei weitem übersteigt.

Abwärme-Problem

Tabelle 1.1: Technologien integrierter Schaltungen.

Integrationsdichte	Kurzbezeichnung	Anzahl Transistoren
Small Scale Integration	SSI	100
Medium Scale Integration	MSI	1.000
Large Scale Integration	LSI	10.000
Very Large Scale Integration	VLSI	100.000
Ultra Large Scale Integration	ULSI	1.000.000
Super Large Scale Integration	SLSI	10.000.000
Extra Large Scale Integration	ELSI	100.000.000
Giga Scale Integration	GSI	$>$ 1.000.000.000

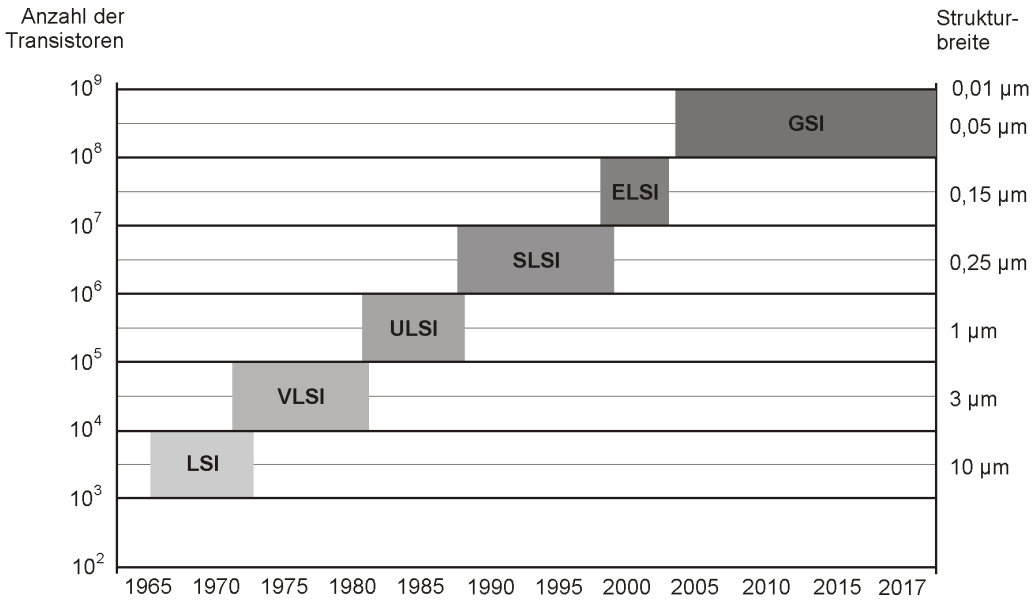

Abbildung 1.2: Entwicklung von Integrationsdichten und Strukturgrößen

[5]Intel's 22-Core Xeon Bradwell-E5

Es gibt verschiedene Arten von Mikrochips, die sich bezüglich der Regelmäßigkeit ihrer geätzten Strukturen unterscheiden. Speicherchips sind am regelmäßigsten aufgebaut, wodurch hiermit die höchsten Integrationsdichten erreicht werden können. Prozessoren erreichen aufgrund ihrer Komplexität und Unregelmäßigkeit weniger als die Hälfte der Integrationsdichte von Speicherbausteinen. Neben Prozessoren gibt es auch noch anwendungsspezifische Bausteine, die entweder maskenprogrammierbar (*Application Specific Integrated Circuit*, ASIC) oder elektrisch programmierbar sind (*Field Programmable Gate Array*, FPGA). Die Hersteller dieser Bausteine verwenden statt der Transistorenanzahl als Komplexitätsmaß häufig die Zahl der Gatteräquivalente. Als Faustregel gilt, dass zur Realisierung eines Gatters etwa vier Transistoren benötigt werden.

anwendungsspezifische Mikrochips

Betrachten wir als nächstes den Zusammenhang zwischen der Befehlssatzarchitektur und den beiden darunter liegenden Schichten. Die bisher im Kurs 1608 eingeführten Prozessoren basieren auf mikroprogrammierten Steuerwerken. Diese Art der logischen Organisation hatte sich in den sechziger und siebziger Jahren entwickelt. Damals hatten die Hauptspeicher nur geringe Speicherkapazitäten und man versuchte daher, möglichst mächtige Maschinenbefehle bereitzustellen, um die zur Lösung eines Problems benötigten Verarbeitungsschritte in einem kurzen Programm codieren zu können. Gleichzeitig wollte man die Maschinenprogrammierung ähnlich komfortabel gestalten wie die Programmierung in einer höheren Programmiersprache.

Die Befehlssätze derartiger *CISC-Prozessoren* (*Complex Instruction Set Computer*) bieten eine große Vielfalt an Adressierungsarten, die unterschiedlich viele Befehlsphasen (Taktzyklen) erfordern und nur mittels Mikroprogrammierung effizient implementiert werden können. Abbildung 1.3 a) zeigt beispielhaft die Abarbeitung von zwei unterschiedlich langen CISC-Befehlen. Die Auslastung der einzelnen Prozessor-Funktionseinheiten ist dabei schlecht, da die Maschinenbefehle nur nacheinander abgearbeitet werden können. So wird beispielsweise bei einem arithmetischen Befehl mit einem Speicherzugriff die ALU im Rechenwerk nur einen Taktzyklus lang genutzt, während der gesamte Befehl meist mehr als zehn Taktzyklen erfordert. Durch Einschränkungen der Befehlssatzarchitektur – vor allem bei den Adressierungsmöglichkeiten – erreicht man mit der Einführung so genannter *RISC-Prozessoren*[6] (*Reduced Instruction Set Computer*) eine deutliche Vereinfachung der Implementierung. Das Ziel der Architekturveränderungen bestand darin, die Implementierung *aller* Maschinenbefehle auf eine feste Anzahl von Mikroschritten zu beschränken, siehe skalarer RISC-Prozessor (ohne Pipeline) in Abbildung 1.3 b). Pro Taktzyklus ist immer genau ein Mikroschritt ausführbar. Durch die Vereinheitlichung der Befehle in der Anzahl der Mikroschritte war man dann in der Lage, das *Pipelining*-Prinzip mit einer überlappenden Verarbeitung anzuwenden, siehe skalarer RISC-Prozessor (mit Pipeline) in Abbildung 1.3 c). Im Idealfall kann mit diesem Ansatz die Auslastung der Prozessor-Funktionseinheiten auf 100% gesteigert und in jedem Taktzyklus ein Befehl beendet werden[7]. RISC-Prozessoren wurden zunächst *skalar* ausge-

CISC-Prinzip

RISC-Prinzip

skalare RISC

[6] Ab ca. 1980er Jahre

[7] Mehr zum Pipeline-Prinzip folgt in Kapitel 2.

legt, d. h. es gab nur eine ALU in der Ausführungsstufe. Durch Hinzunahme von mehreren Ausführungseinheiten entstehen so genannte *Superskalare*-RISC-Prozessoren, die gleichzeitig mehrere Befehle holen, verplanen (*Scheduling*) und parallel ausführen können.[8] Auf diese Weise können im Vergleich zum skalaren Prozessor mit jedem Taktzyklus sogar mehrere Befehle gleichzeitig beendet werden, siehe superskalarer RISC-Prozessor in Abbildung 1.3 d).

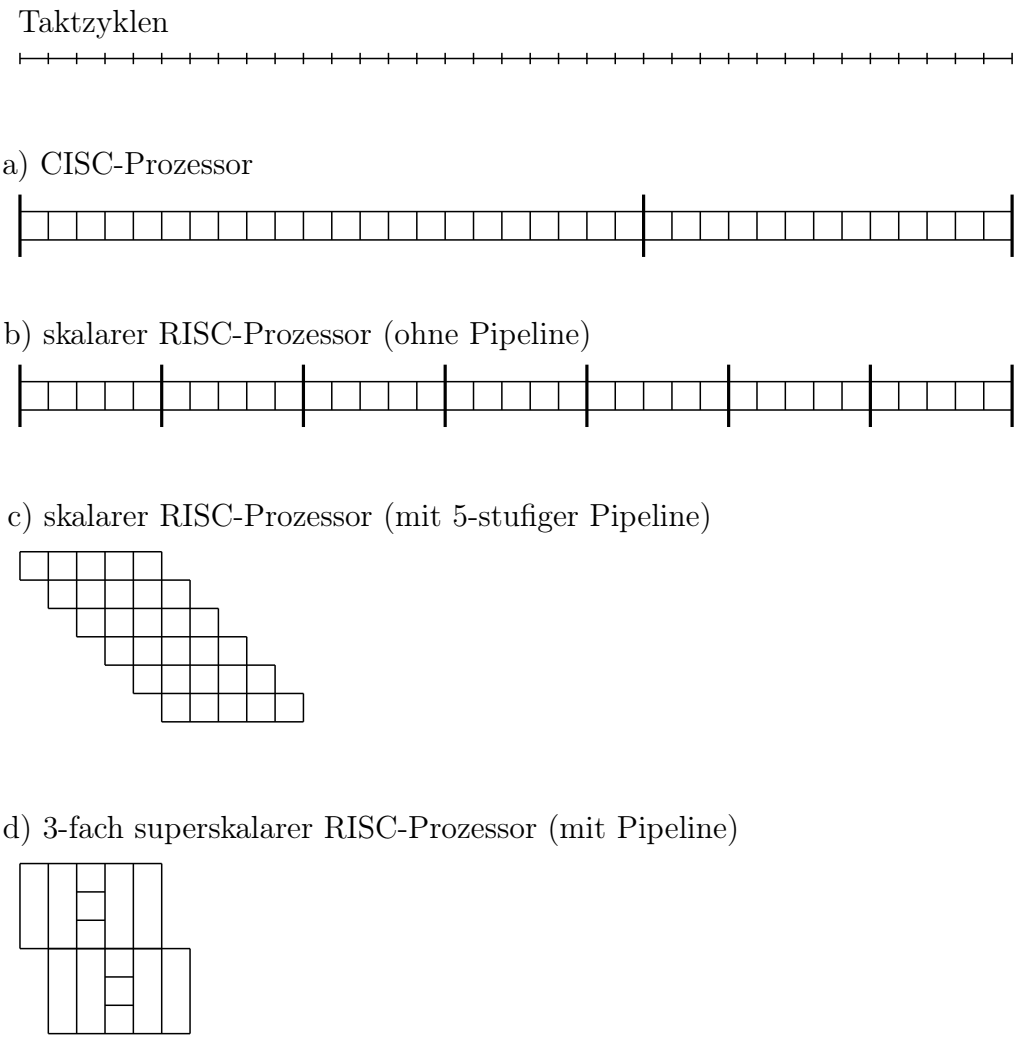

Abbildung 1.3: Verschiedene Abarbeitungsreihenfolgen.

superskalare RISC

Bei superskalaren RISC-Prozessoren erfolgt die Zuteilung der Befehle dynamisch zur Laufzeit mittels Hardware in einer entsprechenden Pipelinestufe. Darüber hinaus gibt es auch Prozessoren mit parallel arbeitenden Ausführungseinheiten, die auf einem *statischen* Scheduling basieren. Hierbei wird das Scheduling nicht erst zur Laufzeit ausgeführt sondern bereits vom Compiler

[8] Mehr zur Superskalarität folgt in Kapitel 4.

bei der Übersetzung in das Maschinenprogramm vorgegeben. Man spricht von VLIW-Prozessoren (*Very Long Instruction Word*), weil der Compiler mehrere parallelisierbare Befehle zu einem sehr langen Maschinenwort zusammenfügt, die dann im Prozessor auf die einzelnen Ausführungseinheiten verteilt werden. Eine Kombination von statischem und dynamischem Scheduling bildet das EPIC (*Explicitly Parallel Instruction Computing*), das für die IA64-Architektur des Intel-Itanium-Prozessors entwickelt wurde, sich auf dem Markt aber nicht durchsetzen konnte. Die zentrale Idee bestand darin, durch den Compiler das Hardware-Scheduling zu unterstützen und so den Hardware-Aufwand im Prozessor zu verringern.

statisches Scheduling bei VLIW

1.3 Grundlagen der Befehlssatzarchitektur

1.3.1 Befehlssatz- und Mikroarchitektur

Eine **Befehlssatzarchitektur** (*Instruction Set Architecture*, ISA) definiert die Grenze zwischen Hardware und Software. Sie umfasst den für den Systemprogrammierer und für den Compiler sichtbaren Teil des Prozessors und bildet die Schnittstelle nach außen. Als Synonym wird auch von der **Prozessorarchitektur** bzw. dem **Programmiermodell** eines Prozessors gesprochen. Hierzu gehören neben dem Befehlssatz (Menge der verfügbaren Befehle), das Befehlsformat, die Adressierungsarten, das System der Unterbrechungen und das Speichermodell (Register und Adressraumaufbau). Eine Prozessorarchitektur macht hingegen keine Aussagen über die Details der Hardware oder die technischen Realisierung eines Prozessors. Sie legt lediglich das äußere Erscheinungsbild eines Prozessors fest, ohne dabei auf interne Vorgänge einzugehen.

Befehlssatzarchitektur

Eine **Mikroarchitektur** (entsprechend dem englischen Begriff *Microarchitecture*) bezeichnet die Implementierung einer Prozessorarchitektur in einer speziellen Verkörperung der Architektur – einem Mikroprozessor. Hierzu gehören die Struktur der Hardware und der Entwurf der Kontroll- und Datenpfade. Zu den Mikroarchitekturtechniken zählen die Art und Stufenzahl des Befehls-Pipelining, der Grad der Verwendung der Superskalartechnik, die Art und Anzahl der internen Ausführungseinheiten eines Mikroprozessors sowie der Einsatz und die Organisation von Cache-Speichern[9]. Die Mikroarchitektur definiert also die logische Organisation eines Prozessors, während die Befehlssatzarchitektur diese Eigenschaften nicht erfasst. Ein Systemprogrammierer oder ein optimierender Compiler benötigt daher auch Kenntnisse über Mikroarchitektureigenschaften, um effizienten Code für einen Mikroprozessor zu erzeugen und zu optimieren. Architektur- und Implementierungstechniken werden beide auch als **Prozessortechniken** bezeichnet.

Mikroarchitektur

Prozessortechniken

Die Trennung von Architektur und Mikroarchitektur macht Benutzerprogramme unabhängig von der konkreten Implementierung des Prozessors, auf dem

[9] Unter einem Cache-Speicher versteht man einen kleinen, schnellen und assoziativen Pufferspeicher, in dem Kopien derjenigen Teile des Hauptspeichers gepuffert werden, auf die aller Wahrscheinlichkeit nach vom Prozessor in naher Zukunft zugegriffen wird.

sie ausgeführt werden. Mikroprozessoren, die derselben Architekturspezifikation folgen, sind *binärkompatibel* zueinander. Die verschiedenen Implementierungstechniken zeigen sich dann durch die unterschiedlichen Verarbeitungsgeschwindigkeiten bei der Programmausführung.

Prozessorfamilie

Man spricht von einer **Prozessorfamilie** , wenn alle Prozessoren die gleiche Basisarchitektur haben, wobei häufig die neueren oder die komplexeren Prozessoren der Familie die Architekturspezifikation erweitern. In einem solchen Fall der Erweiterung ist nur eine Abwärtskompatibilität mit den älteren bzw. einfacheren Prozessoren der Familie gegeben, d. h. der Programmcode der älteren Prozessoren läuft auch auf den neueren Prozessoren, jedoch nicht unbedingt umgekehrt.

Programmiermodell

Das Programmiermodell eines Prozessors lässt sich durch das Beantworten der folgenden fünf Fragen konkretisieren:

- Wo werden die Daten gespeichert (Register- und Speichermodell)?
- Wie werden Daten repräsentiert (Datenformate)?
- Welche Operationen können auf den Daten ausgeführt werden (Befehlssatz)?
- Wie werden die Befehle codiert (Befehlsformate)?
- Wie wird auf die Operanden zugegriffen (Adressierungsarten)?

Diese Fragen definieren die Befehlssatz-Architektur und werden in den folgenden Unterkapiteln näher behandelt.

1.3.2 Adressraumorganisation

Die Adressraumorganisation bestimmt, wo die zu verarbeitenden Daten gespeichert werden und wie diese aus dem Programm heraus adressiert werden können. Dies umfasst zum einen die für den Programmierer sichtbaren Register und zum anderen den für den Prozess verfügbaren Hauptspeicherbereich. Wird eine virtuelle Speicherverwaltung verwendet, so kommt zum physikalischen Adressraum zusätzlich noch ein logischer Adressraum hinzu, siehe Kapitel 3.

Registerarten

Ein Prozessor enthält eine kleine Anzahl von **Registern** , d. h. schnellen Speicherplätzen, auf die in einem Taktzyklus zugegriffen werden kann. Aufgebaut werden diese durch Flipflops (siehe Kurs 1608) und befinden sich direkt im Prozessorkern. Bei den heute üblichen Pipeline-Prozessoren wird in der Operandenholphase der Befehls-Pipeline auf die Registeroperanden zugegriffen und in der Resultatspeicherphase in die Register zurückgeschrieben. Diese vom Programmierer ansprechbaren Register werden als **Architekturregister** bezeichnet, da sie in der „Architektur“ sichtbar sind. Hierbei kann weiter unterschieden werden in **allgemeine Register** (auch Universalregister oder Allzweckregister genannt), **Multimediaregister**, **Gleitkommaregister** und **Spezialregister** (Befehlszähler, Statusregister etc.). Zu beachten ist, dass die

im Register gespeicherte Bitfolge allein keine Aussage über deren Inhalt bzw. Wert macht. Erst mit einem zugehörigen Datenformat wird ersichtlich, ob es sich z. B. um eine Zahl im Zweierkomplement oder um einen in ASCII[10] kodierten String handelt. Die zu den Registern zugehörigen Funktionseinheiten (Ganzzahl-Einheiten, Gleitkomma-Einheiten etc.) müssen das Datenformat bei der Ausführung von Befehlen und Operationen auf den Registerwerten berücksichtigen. Register haben durch ihre Größe Einfluss auf die Wortbreite des Prozessors und den ansprechbaren Adressraum. Wir sprechen in diesem Kurs von einem n**-Bit-Prozessor**, wenn die allgemeinen Register n-Bit breit sind. Da diese Register häufig auch für die Speicheradressierung verwendet werden, ist dann auch die Breite der effektiven Adresse n-Bit und der adressierbare Speicherbereich 2^n Byte groß. Im Bereich von Desktop- und Workstation-Prozessoren liegt die Wortbreite heute üblicherweise bei 64-Bit (AMD64-Erweiterung von x86, Itanium, UltraSPARC und weitere). Bei Mikrocontrollern sind auch oft weiterhin 8-Bit- und 16-Bit-Prozessorkerne im Einsatz.

Datenlängen

Bei der Datenübertragung zwischen Speicher und Register kann die Größe der übertragenen Datenportion mit dem Maschinenbefehl festgelegt werden. Übliche Datenlängen sind Byte (8 Bits), Halbwort (16 Bits), Wort (32 Bits) und Doppelwort (64 Bits). In der Assemblerschreibweise werden diese Datengrößen oft durch die Buchstaben `b, h, w` und `d` abgekürzt, wie z. B. `lw`, um ein Wort zu laden. Beim Intel x86 oder bei Mikrocontrollern – das sind aus einem Mikroprozessor und verschiedenen Schnittstelleneinheiten bestehende vollständige Mikrorechner auf einem einzigen Chip – werden auch die Definitionen Wort (16 Bits), Doppelwort (32 Bits) und Quadwort (64 Bits) angewandt.

Neben den Architekturregistern gibt es üblicherweise noch zahlreiche weitere Register innerhalb des Prozessors, die für interne Vorgänge verwendet werden und für den Programmierer nach außen hin nicht zugreifbar sind (z. B. als Puffer bei Datenübertragungen oder Umbenennungsregister in der Piepline etc.).

logischer Adressraum

Außer den direkt adressierbaren Registern werden alle weiteren Adressbereiche eines Prozesses meist[11] auf einen einzigen, durchgehend nummerierten **logischen Adressraum** abgebildet. Dieser beinhaltet von der Adresse 0 an aufsteigend folgende Bereiche:

- Programmcode und statische Daten
- dynamische Daten (Heap)
- Laufzeitstapel (Stack)
- Bereich für Ein-/Ausgabe und Steuerdaten

Der Heap wächst im Programmverlauf von „unten nach oben“, d. h. von niedrigeren zu höheren Adressen. Er bietet Platz für die zur Laufzeit dynamisch

[10] *American Standard Code for Information Interchange*; Zeichenkodierung für häufig verwendete Buchstaben, Zahlen und Sonderzeichen.

[11] D.h. bei Verwendung einer virtuellen Speicherverwaltung, siehe Kapitel 3.

erzeugten Daten. Der Stack hingegen beginnt meist bei größeren Adressen und wächst „nach unten“. Er dient zur Verwaltung von Unterbrechungen und Unterprogrammaufrufen und nimmt z. B. die jeweiligen Rücksprungadressen und Registerinhalte auf.[12] Einem Prozess muss entsprechend genügend Speicherplatz zugeordnet werden, damit sich Stack und Heap niemals überschneiden, da die resultierenden Fehler ansonsten zu einem Programmabbruch oder einem unvorhergesehenem Programmverhalten führen. Mehr zur Speicherverwaltung wird in Kapitel 3 folgen.

Der Speicherbereich für Ein-/Ausgabe dient zur Kommunikation mit externen Geräten. Eine Möglichkeit dazu ist die Verwendung von Steuerregistern. Diese im externen Gerät befindlichen Register werden in den Speicherbereich eingeblendet und können so aus dem Programm heraus adressiert werden (*Memory-Mapped I/O*). Alternativen dazu sind die Verwendung eines separaten Adressraumes oder die Verwendung von I/O-Ports, auf welche wir hier nicht weiter eingehen werden.

Der Adressraum kann **byteadressierbar** oder **wortadressierbar** sein. Bei einem byteadressierbaren Adressraum kann jedes Byte einzeln adressiert werden. Bei wortadressierbaren Speichern kann jeweils nur ein Wort, d. h. das Vielfache eines Bytes adressiert, ausgelesen und beschrieben werden. Die Speicherzugriffe müssen dafür ausgerichtet (*aligned*) sein: Ein Zugriff auf ein Speicherwort mit einer Länge von n Bytes ab der Speicheradresse A heißt ausgerichtet, wenn A modulo $n = 0$ gilt, d. h. der Rest der ganzzahligen Division von A durch n den Wert 0 ergibt. Speicheradressen werden üblicherweise in hexadezimaler Schreibweise angegeben. Bei Wortadressierung mit 32 Bit unterscheiden sich die Adressen benachbarter Speicherwörter um den Wert 4.

Little- und Big-Endian-Format

Für die Speicherung der Bytes innerhalb eines Wortes unterscheidet man zwei Arten der Anordnung:

- Das ***Big-Endian*-Format**(„*most significant byte first*“) speichert von links nach rechts, d. h., die Adresse des Speicherworts ist die Adresse des höchstwertigen Bytes des Speicherworts.

- Das ***Little-Endian*-Format** („*least significant byte first*“) speichert von rechts nach links, d. h., die Adresse eines Speicherworts ist die Adresse des niedrigstwertigen Bytes des Speicherworts.

Die Anordnung betrifft nur die Speicherung der Bytes *innerhalb* eines Wortes. Für die Assemblerprogrammierung ist diese Unterscheidung meistens irrelevant, da beim Laden eines Operanden aus dem Speicher in ein Register die Maschine den zu ladenden Wert so anordnet, wie man es erwartet, nämlich die höchstwertigen Stellen links (Big-Endian-Format). Dies geschieht auch für das Little-Endian-Format, das bei einigen Prozessorarchitekturen, insbesondere den Intel-Prozessoren, aus Kompatibilitätsgründen mit älteren Prozessoren weiter

[12] Vgl. Kurseinheit 4 aus Computersysteme I. Alternativ werden Rücksprungadressen auch in einem speziellen Register gesichert.

verwendet wird. Beachten muss man das Byte-Anordnungsformat eines Prozessors hingegen beim direkten Zugriff auf einen Speicherplatz als Byte oder Wort oder bei der Arbeit mit einem *Debugger*, einem Werkzeug zum Finden von Fehlern in Programmen. Die Byte-Reihenfolge wird außerdem relevant, wenn Daten zwischen zwei Rechnern verschiedener Architekturen ausgetauscht werden. Heute setzt sich insbesondere durch die Bedeutung von Rechnernetzen das Big-Endian-Format durch, das häufig auch als Netzwerk-Format bezeichnet wird. In Abbildung 1.4 ist beispielhaft eine Zahl im Dezimal-, Binär- und Hexadezimalformat dargestellt und es wird gezeigt, wie diese im Speicher im Big- bzw. Little-Endian-Format gespeichert wird.

Dez	1.513.551.467
Bin	0101 1010 0011 0110 1111 0110 0110 1011
Hex	5A 36 F6 6B

Little-Endian

Speicheradresse	Inhalt
1024	6B
1025	F6
1026	36
1027	5A

Big-Endian

Speicheradresse	Inhalt
1024	5A
1025	36
1026	F6
1027	6B

Abbildung 1.4: Anordnung der Bytes im Speicher bei dem Big- und Little-Endian-Format.

1.3.3 Datenformate

Wie in höheren Programmiersprachen gibt es auch in maschinennahen Sprachen verschiedene Datenformate. Diese geben vor, wie eine Bitfolge zu interpretieren ist bzw. wie ein bestimmter Wert als Bitfolge repräsentiert werden kann. Im Vergleich zu höheren Sprachen sind die Datenformate bei Maschinensprachen oft einfach gehalten und es wird sich auf wenige, häufig benötigte Datenformate beschränkt.

Gebräuchliche Datenformate sind Einzelbit-, Ganzzahl- (Integer), Gleitkomma-, und Multimediaformate, welche im Folgenden näher erläutert werden.

Einzelbitdatenformate können alle Datenlängen von 8 Bit bis hin zu 256 Bit aufweisen. Das Besondere dabei ist, dass einzelne Bits eines solchen Worts manipuliert werden können.

Ganzzahldatenformate

Ganzzahldatenformate (siehe Abbildung 1.5) können unterschiedlich lang und jeweils mit Vorzeichen (*signed*) oder ohne Vorzeichen (*unsigned*) definiert sein. Die Darstellung als Zweierkomplement mit implizitem Vorzeichen wurde

bereits im Kurs 1608 eingeführt. Gelegentlich findet man auch gepackte (*packed*) und ungepackte (*unpacked*) BCD-Zahlen (*Binary Coded Decimal*) und ASCII-Zeichen (*American Standard Code for Information Interchange*). Eine BCD-Zahl codiert die Ziffern 0 bis 9 als Dualzahlen in vier Bits. Das gepackte BCD-Format codiert zwei BCD-Zahlen pro Byte, also acht BCD-Zahlen pro 32-Bit-Wort. Das ungepackte BCD-Format codiert eine BCD-Zahl an den vier niederwertigen Bit-Positionen eines Bytes, also nur vier BCD-Zahlen pro 32-Bit-Wort. Der ASCII-Code belegt ein Byte pro Zeichen, sodass vier ASCII-codierte Zeichen in einem 32-Bit-Wort untergebracht werden.

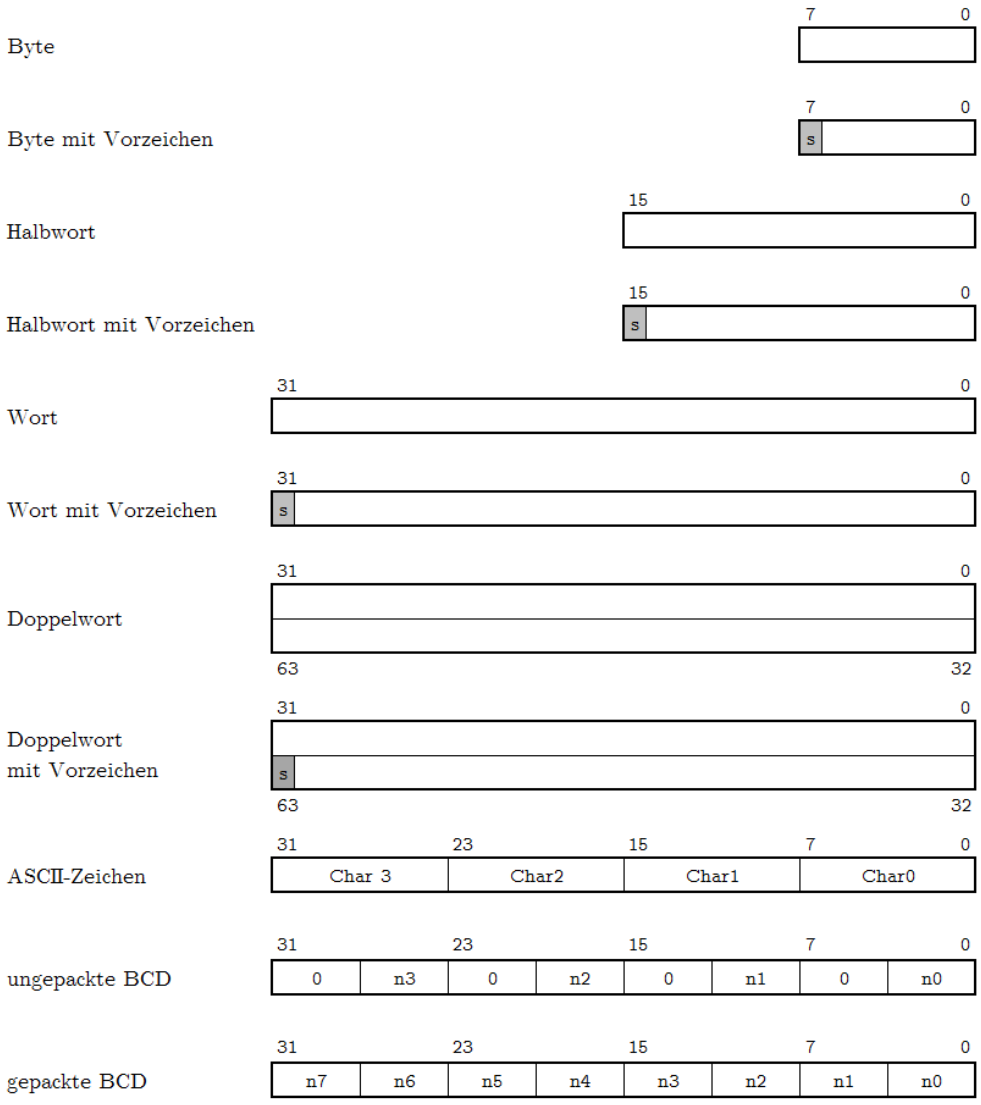

Abbildung 1.5: Ganzzahldatenformate.

Gleitkomma–datenformate

Die **Gleitkommadatenformate** (siehe Abbildung 1.6) wurden mit dem

IEEE-754-Standard definiert und unterscheiden Gleitkommazahlen mit einfacher (32 Bit) oder doppelter (64 Bit) Genauigkeit. Das Format mit erweiterter Genauigkeit umfasst 80 Bit und kann herstellergebunden variieren. Beispielsweise verwendeten die Intel Pentium-Prozessoren intern ein solches erweitertes Format mit 80 Bit breiten Gleitkommazahlen.

Eine Gleitkommazahl f wird nach dem IEEE-Standard wie folgt dargestellt:

$$f = (-1)^s \cdot 1.m \cdot 2^{e-b}$$

Dabei steht s für das Vorzeichenbit (0 für positiv, 1 für negativ), e für den verschobenen (*biased*) Exponenten, b für die Verschiebung (*bias*) und m für den Mantissenteil. Die führende Eins in der obigen Gleichung ist implizit vorhanden und benötigt kein Bit im Mantissenfeld. Für den Mantissenteil m gilt:

$m = .m_1 \cdots m_p$ mit [13]
$p = 23$ für die einfache,
$p = 52$ für die doppelte und
$p = 63$ für die erweiterte Genauigkeit (inkl. führender Eins der Mantisse)

Die Verschiebung b ist definiert als:

$$b = 2^{ne-1} - 1$$

wobei ne die Anzahl der Exponentenbits (je nach Genauigkeit 8, 11 oder 15) angibt. Den Exponenten E erhält man aus der Gleichung:

$$E = e - b = e - (2^{ne-1} - 1)$$

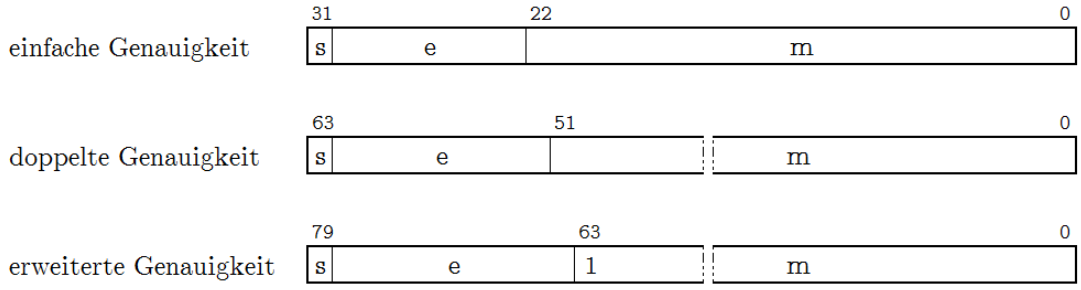

Abbildung 1.6: Gleitkomma-Datenformate.

[13] Beachten Sie den führenden Dezimalpunkt.

Multimedia–datenformate

Multimediadatenformate definieren 64 oder mehr Bit breite Wörter.[14] Durch Multimedia-Erweiterungen des Befehlssatzes ist es möglich, häufig verwendete und wiederkehrende Operationsfolgen, wie sie bei Multimedia-Anwendungen typischerweise auftreten, effizienter und mit einer einzigen Operation abzuarbeiten. Dafür werden spezielle Multimediaeinheiten verwendet, welche Multimedia-Operationen ausführen und die Programmlaufzeit so erheblich verkürzen können. Man unterscheidet allgemein zwei Arten von Multimediadatenformaten: die *bitfeldorientierten* Formate unterstützen Operationen auf Pixeldarstellungen, wie sie für die Videocodierung oder -decodierung benötigt werden. Die *graphikorientierten* Formate unterstützen komplexe graphische Datenverarbeitungsoperationen. Die Multimediadatenformate für die bitfeldorientierten Formate sind in 8 oder 16 Bit breite Teilfelder zur Repräsentation jeweils eines Pixels aufgeteilt. Die graphikorientierten Formate beinhalten zwei oder mehr einfach genaue Gleitkommazahlen.

Darüber hinaus gibt es noch weitere Befehlssatzerweiterungen (z. B. für Verschlüsselung, Virtualisierung), welche wiederum ihre eigenen Datenformate verwenden. Auf diese soll hier aber nicht weiter eingegangen werden.

Selbsttestaufgabe 1.1

1. Wandeln Sie folgende Dezimalzahlen in das 32-Bit IEEE-754-Format um:
   - 12.78125
   - -784
   - -0.15625
2. Wandeln Sie folgende 32-Bit-Zahlen im IEEE-754-Format in das Dezimalsystem um:
   - `11000010010110000000000000000000`
   - `01000100010111000100011000000000`
   - `00111111110111000000000000000000`
3. Wandeln Sie folgende Ganzzahlen in das 32-Bit-Zweierkomplement um:
   - 125
   - -1348
   - -80292
4. Geben Sie folgende Strings als eine Bytefolge im ASCII-Format an:
   - „FernUni“
   - „Cache“
   - „Sandbox“

[14] Die Befehlssatzerweiterung AVX512 verwendet z. B. Wortbreiten bis 512 Bit.

5. Wandeln Sie folgende Dezimalzahlen sowohl in die gepackte als auch die ungepackte BCD-Darstellung um:
   - 1548
   - 4828

1.3.4 Befehlssatz

Befehlssatz

Der **Befehlssatz** (*Instruction Set*) definiert, welche Operationen auf den Daten ausgeführt werden können. Er legt daher die Grundoperationen eines Prozessors fest, die ein Assembler-Programmierer (bzw. der Compiler) in seinem Programm verwenden kann. Man kann dabei die folgenden **Befehlsarten** unterscheiden:

Datenbewegungsbefehle (*Data Movement*) übertragen Daten von einer Speicherstelle zu einer anderen. Falls es einen separaten Ein-/Ausgabeadressraum gibt, so gehören hierzu auch die Ein-/Ausgabebefehle. Auch die Stapelspeicherbefehle *Push* und *Pop* fallen – sofern vorhanden – in diese Kategorie.

Arithmetisch-logische Befehle (*Integer Arithmetic and Logical*) können Ein-, Zwei- oder Dreioperandenbefehle sein. Prozessoren nutzen meist verschiedene Befehle für verschiedene Datenformate ihrer Operanden. Meist werden durch den Befehlsopcode arithmetische Befehle mit oder ohne Vorzeichen unterschieden. Beispiele arithmetischer Operationen sind Addieren ohne/mit Übertrag, Subtrahieren ohne/mit Übertrag, Inkrementieren und Dekrementieren, Multiplizieren ohne/mit Vorzeichen, Dividieren ohne/mit Vorzeichen und Komplementieren im Zweierkomplement. Beispiele logischer Operationen sind die bitweise Negation-, Und-, Oder- und Antivalenz-Operationen.

Schiebe- und Rotationsbefehle (*Shift, Rotate*) schieben die Bits eines Worts um eine Anzahl von Stellen entweder nach links oder nach rechts bzw. rotieren die Bits nach links oder rechts, siehe Abbildung 1.7).

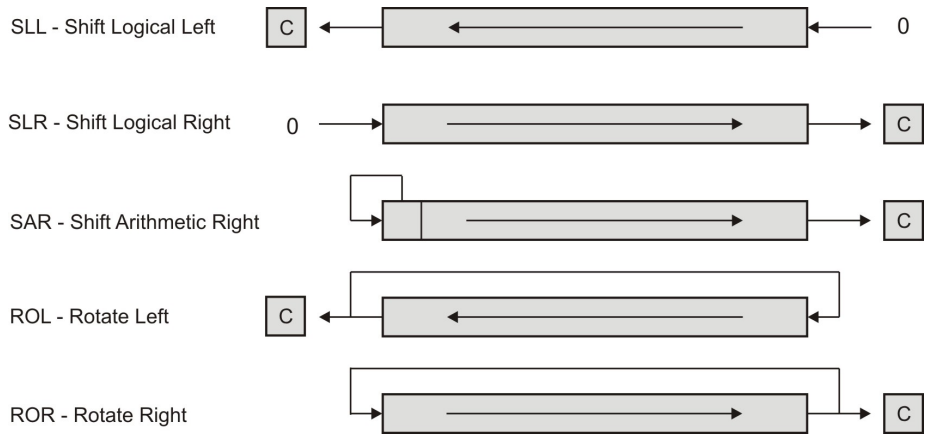

Abbildung 1.7: Einige Schiebe- und Rotationsbefehle.

Beim Schieben gehen die herausfallenden Bits verloren, beim Rotieren werden diese auf der anderen Seite wieder eingefügt. Beispiele von Schiebe- und

Rotationsoperationen sind das Linksschieben, Rechtsschieben, Linksrotieren ohne Übertragsbit, Linksrotieren durchs Übertragsbit, Rechtsrotieren ohne Übertragsbit und das Rechtsrotieren durchs Übertragsbit. Dem arithmetischen Linksschieben entspricht die Multiplikation mit 2. Dem arithmetischen Rechtsschieben entspricht die ganzzahlige Division durch 2. Dies gilt jedoch nur für positive Zahlen. Bei negativen Zahlen im Zweierkomplement muss zur Vorzeichenerhaltung das höchstwertige Bit in sich selbst zurückgeführt werden. Daraus ergibt sich der Unterschied des logischen und arithmetischen Rechtsschiebens. Beim Rotieren wird ein Register als geschlossene Bit-Kette betrachtet. Ein so genanntes Übertragsbit (*Carry Flag*) im Prozessorstatusregister kann wahlweise mitbenutzt oder als zusätzliches Bit einbezogen werden.

Multimediabefehle (*Multimedia Instructions*) führen taktsynchron dieselbe Operation auf mehreren Teiloperanden innerhalb eines Operanden aus, siehe Abbildung 1.8. Man unterscheidet zwei Arten von Multimediabefehlen: bitfeldorientierte und graphikorientierte Multimediabefehle.

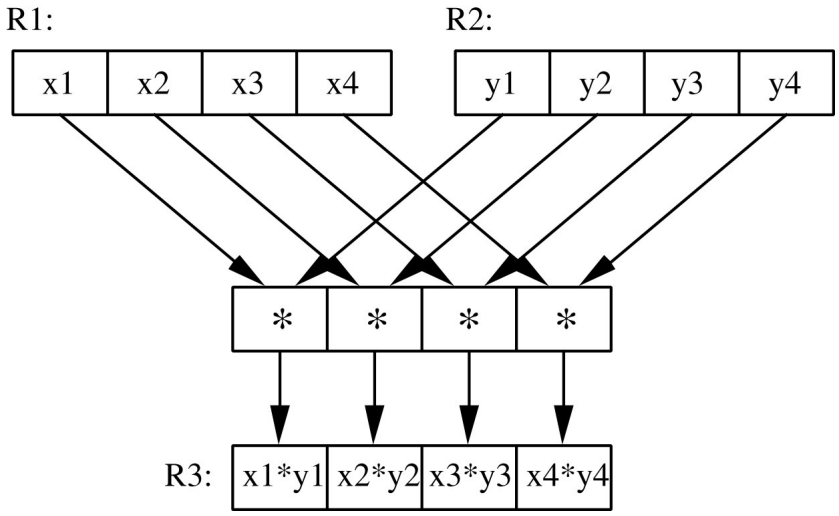

Abbildung 1.8: Grundprinzip einer Multimediaoperation.

Bei den *bitfeldorientierten Multimediabefehlen* repräsentieren die Teiloperanden Pixel. Typische Operationen sind Vergleiche, logische Operationen, Schiebe- und Rotationsoperationen, Packen und Entpacken von Teiloperanden in oder aus Gesamtoperanden sowie arithmetische Operationen auf den Teiloperanden entsprechend einer Saturationsarithmetik: Bei einer solchen Arithmetik werden Zahlbereichsüberschreitungen auf die höchstwertige bzw. niederstwertige Zahl abgebildet. Dadurch wird z. B. verhindert, dass die Summe zweier positiver Zahlen negativ wird.

Bei den *graphikorientierten Multimediabefehlen* repräsentieren die Teiloperanden einfach genaue Gleitkommazahlen, also zwei 32-Bit-Gleitkommazahlen in einem 64-Bit-Wort bzw. vier 32-Bit-Gleitkommazahlen in einem 128-Bit-Wort. Die Multimediaoperationen führen dieselbe Gleitkommaoperation auf allen Teiloperanden aus.

Gleitkommabefehle (*Floating-point Instructions*) repräsentieren arithmetische Operationen und Vergleichsoperationen, aber auch zum Teil komplexe Operationen wie Quadratwurzelbildung oder transzendente Funktionen auf Gleitkommazahlen.

Programmsteuerbefehle (*Control Transfer Instructions*) sind alle Befehle, die den Programmablauf direkt ändern, die bedingten und unbedingten Sprungbefehle, Unterprogrammaufruf und -rückkehr sowie Unterbrechungsaufruf und -rückkehr.

Systemsteuerbefehle (*System Control Instructions*) erlauben es in manchen Befehlssätzen, direkten Einfluss auf Prozessor- oder Systemkomponenten wie z. B. den Cachespeicher oder die Speicherverwaltungseinheit zu nehmen[15]. Weiterhin gehören der HALT-Befehl zum Anhalten des Prozessors und Befehle zur Verwaltung der elektrischen Leistungsaufnahme zu dieser Befehlsgruppe, die üblicherweise nur vom Betriebssystem genutzt werden dürfen.

Synchronisationsbefehle ermöglichen es, Synchronisationsoperationen zur Prozess- und Unterbrechungsbehandlung durch das Betriebssystem zu implementieren.[16] Wesentlich ist dabei, dass bestimmte, eigentlich sonst nur durch mehrere Befehle implementierbare Synchronisationsoperationen ohne Unterbrechung (auch als „atomar“ bezeichnet) ablaufen müssen. Ein Beispiel ist der SWAP-Befehl, der als atomare Operation einen Speicherwert mit einem Registerwert vertauscht. Noch komplexer ist der TAS-Befehl (*Test and Set*), der als atomare Operation einen Speicherwert liest, diesen auf Null testet, ggf. ein Bedingungsbit im Prozessorstatuswort setzt und einen bestimmten Wert zurückspeichert. Die Ausführung als atomare Operation verhindert, dass z. B. ein weiterer Prozessor zwischenzeitlich dieselbe Speicherzelle liest und dort noch den alten, unveränderten Wert vorfindet.

1.3.5 Befehlsformate

Das **Befehlsformat** (*instruction format*) definiert, wie die Befehle codiert sind. Eine Befehlscodierung beginnt mit dem Opcode (*Operation Code*), der den Befehl selbst festlegt. In Abhängigkeit vom Opcode werden weitere Felder im Befehlsformat benötigt. Für arithmetisch-logische Befehle sind dies die Adressfelder, um Quell- und Zieloperanden zu spezifizieren, und für die Lade-/Speicherbefehle die Quell- und Zieladressangaben. Je nach Architektur kann es sich hierbei um Register oder Speicheradressen handeln. Bei Programmsteuerbefehlen wird der als nächstes auszuführende Befehl adressiert.

Adressformate

Je nach der Art, wie die arithmetisch-logischen Befehle ihre Operanden adressieren, unterscheidet man vier Klassen von Befehlssätzen:

- Das **Dreiadressformat** *(3-Address Instruction Format)*, bestehend aus dem Opcode, zwei Quell- (im Folgenden mit src1, scr2 bezeichnet) und einem Zieloperandenbezeichner (dest):

[15] Mehr zu Cache-Speichern und Speicherverwaltung folgt in Kapitel 3.

[16] Unter einem Prozess versteht man ein in der Ausführung befindliches oder ausführbereites Programm mit seinen Daten, den Konstanten und Variablen mit ihren aktuellen Werten.

opcode	dest	src1	scr2

- Das **Zweiadressformat** *(2-Address Instruction Format)*, bestehend aus dem Opcode, einem Quell- und einem kombinierten Quell-/Zieloperandenbezeichner, d. h. einem Operandenbezeichner, der sowohl Quell- als auch Zieloperanden festlegt:

opcode	dest/src1	src2

- Das **Einadressformat** *(1-Address Instruction Format)*, bestehend aus dem Opcode und einem Quelloperandenbezeichner:

opcode	src

  Hierbei wird ein im Prozessor ausgezeichnetes Register, das so genannte Akkumulatorregister, implizit adressiert. Dieses Register enthält immer einen Quelloperanden und nimmt das Ergebnis der Operation auf.

- Das **Nulladressformat** *(0-Address Instruction Format)*, bestehend nur aus dem Opcode:

opcode

  Voraussetzung für die Verwendung eines Nulladressformats ist eine Stackarchitektur, die weiter unten beschrieben wird.

Architekturklassen

Die Adressformate hängen eng mit folgender Klassifizierung von Befehlssatzarchitekturen zusammen:

- Arithmetisch-logische Befehle sind meist Dreiadressbefehle, die zwei Operandenregister und ein Zielregister angeben. Falls nur die Lade- und Speicherbefehle Daten zwischen dem Hauptspeicher (bzw. Cache-Speicher) und den Registern transportieren, spricht man von einer **Lade-/Speicherarchitektur** (*Load/Store Architecture*). Da arithmetische und logische Operationen nur mit Operanden aus Registern ausgeführt und die Ergebnisse auch nur in Registern gespeichert werden können, spricht man auch von einer **Register-Register-Architektur**.

- Analog kann man von einer **Register-Speicher-Architektur** sprechen, wenn in arithmetisch-logischen Befehlen mindestens einer der Operandenbezeichner ein Register bzw. einen Speicherplatz im Hauptspeicher adressiert. Falls gar keine Register existieren, muss jeder Operandenbezeichner eine Speicheradresse sein und man kann von einer **Speicher-Speicher-Architektur** sprechen.[17] Die ersten Mikroprozessoren besaßen ein Akkumulatorregister, das bei arithmetisch-logischen Befehlen immer implizit sowohl Quelle als auch Ziel darstellte, so dass Einadressbefehle genügten. Solche **Akkumulatorarchitekturen** sind gelegentlich noch bei einfachen Mikrocontrollern und Digitalen Signalprozessoren (DSPs) zu finden.

[17] Der einzige uns bekannte Prozessor dieses Typs war der TI 9900 der Firma Texas Instruments.

- So genannten **Stack-** oder **Kellerarchitekturen** verwalten ihre Operandenregister als Stapel (*Stack*). Eine zweistellige Operation verknüpft die beiden obersten Stackeinträge miteinander, löscht beide Inhalte vom Registerstapel und speichert das Resultat auf dem obersten Register des Stapels wieder ab.

In der Regel kann bei heutigen Mikroprozessoren jeder Registerbefehl auf jedes beliebige Register gleichermaßen zugreifen. Bei älteren Prozessoren war dies jedoch keineswegs der Fall. Noch beim Intel 8086 gab es viele Anomalien beim Registerzugriff. Beispiele für Stackarchitekturen sind die von der *Java Virtual Machine* hergeleiteten Java-Prozessoren, die Verwaltung der Gleitkommaregister der Intel 8087- bis 80387-Gleitkomma-Coprozessoren sowie der 4-Bit-Mikroprozessor ATAM862 der Firma Atmel.

Befehlslänge

Die Befehlscodierung kann eine feste oder eine variable Befehlslänge benutzen. Um die Decodierung zu vereinfachen, nutzen RISC-Befehlssätze meist ein Dreiadressformat mit einer festen Befehlslänge. CISC-Befehlssätze dagegen nutzen Register-Speicher-Befehle und benötigen dafür meist variable Befehlslängen. Um den Speicherbedarf zu minimieren, hat man früher mit variablen Opcodelängen bei CISC-Befehlssätzen gearbeitet. Hierzu wurden häufig benutzten Befehlen möglichst kurze Opcodes zugeordnet, was den Gesamtspeicherplatz des Programms verringert. Variable Befehlslängen finden sich auch in Stackmaschinen wie z. B. den Java-Prozessoren.

Abbildung 1.9 zeigt, wie sich die verschiedenen Befehlssatz-Architekturen auf die Übersetzung eines Programms in (Pseudo-)Assemblersprache auswirken. Das Beispielprogramm hat folgenden Aufbau:

$$C = A + B$$
$$D = C - B$$

Die Syntax der Assemblerbefehle schreibt vor, dass nach dem Opcode zuerst der Zieloperandenbezeichner und dann der oder die Quelloperandenbezeichner stehen. Manche andere Assemblernotationen verfahren genau umgekehrt und verlangen, dass der oder die Quelloperandenbezeichner vor dem Zieloperandenbezeichner stehen müssen. Gut zu erkennen ist, dass Register-Register-Architekturen und Register-Speicher-Architekturen Zwei- oder Dreiadressbefehle verwenden. Akkumulator- bzw. Stackarchitekturen kommen hingegen mit Ein- oder sogar Nulladressbefehlen aus.

Hinweis

Betrachten Sie erneut Abbildung 1.9 und überlegen Sie, wie sich die Befehlssatz-Architektur auf die Größe und die Laufzeit eines Programms auswirkt! Beachten Sie außerdem, wie sich dies bei festen oder variablen Befehlslängen ändern würde! Die Architektur hat, wie gut zu erkennen ist, auch einen Einfluss auf die Anzahl der benötigten Speicherzugriffe.

Register-Register	Register-Speicher	Akkumulator	Stack
load Reg1, A	load Reg1, A	load A	push B
load Reg2, B	add Reg1, B	add B	push A
add Reg3, Reg1, Reg2	store C, Reg1	store C	add
store C, Reg3	sub Reg1, B	sub B	pop C
sub Reg3, Reg3, Reg2	store D, Reg1	store D	push B
store D, Reg3			push C
			sub
			pop D

Abbildung 1.9: Beispielprogramm in unterschiedlichen Befehlssatz-Architektur.

1.3.6 Adressierungsarten

Die **Adressierungsart** definiert, wie aus dem Programm auf Daten zugegriffen wird. Sie legt die verschiedenen Möglichkeiten fest, wie eine Operanden- oder eine Sprungzieladresse in dem Prozessor berechnet wird. Die Adresse kann eine im Befehlswort stehende Konstante, ein Register oder einen Speicherplatz im Hauptspeicher spezifizieren.

effektive, virtuelle und physikalische Adresse

Wenn ein Speicherplatz im Hauptspeicher adressiert wird, so heißt die durch die Adressierungsart spezifizierte Speicheradresse die **effektive Adresse**. Eine effektive Adresse entsteht im Prozessor nach Ausführung der jeweiligen Adressierungsart. Ohne virtuelle Speicherverwaltung entspricht die effektive Adresse der physikalischen Adresse für den Zugriff auf den Hauptspeicher. Üblicherweise verwenden heutige Prozessoren allerdings eine virtuelle Speicherverwaltung, wodurch eine weitere Adressumrechnung notwendig wird. Die effektive Adresse entspricht dann der virtuellen Adresse, welche mit Hilfe einer in Hardware realisierten Speicherverwaltungseinheit (*Memory Management Unit – MMU*) mit Segmentierung und/oder Seitenverwaltung in die physikalische Adresse umgerechnet wird. Das Ergebnis der Adressumrechnung vom virtuellen in den physikalischen Adressraum ist ebenfalls wieder die physikalische Adresse für den eigentlichen Speicherzugriff. Der Zusammenhang der verschiedenen Bezeichnungen wird noch einmal in Abbildung 1.10 verdeutlicht. Im Folgenden betrachten wir zunächst nur die Erzeugung einer effektiven Adresse aus den Angaben in einem Maschinenbefehl[18].

Neben den „expliziten“ Adressierungsarten kann die Operandenadressierung auch „implizit“ über die Befehlssatz-Architektur oder durch den Opcode des Befehls festgelegt sein. In der oben erwähnten Stackarchitektur sind z. B. die beiden Quelloperanden und der Zieloperand einer arithmetisch-logischen Operation implizit als die beiden bzw. der oberste Stackeintrag festgelegt. Ähnlich ist bei einer Akkumulatorarchitektur das Akkumulatorregister bereits implizit

[18] Näheres zur virtuellen Speicherverwaltung folgt in Kapitel 3

Abbildung 1.10: Adressumrechnung.

als ein Quell- und als Zielregister der arithmetisch-logischen Operationen fest vorgegeben. Bei einer Register-Speicher-Architektur ist meist das eine Operandenregister fest vorgegeben, während der zweite Operand und das Ziel explizit adressiert werden müssen.

Durch den Opcode eines Befehls wird bei Spezialbefehlen (Programm- oder Systemsteuerbefehle) häufig ein besonderes Register als Quelle und/oder Ziel adressiert. Weiterhin wird in vielen Befehlssätzen im Opcode festgelegt, ob ein Bit des Prozessorstatusregisters mit verwendet wird.

Bei den im Folgenden aufgeführten expliziten Adressierungsarten können drei Klassen unterschieden werden:

- die Klasse der Registeradressierung und unmittelbaren Adressierung,
- die Klasse der einstufigen und
- der Klasse der zweistufigen Speicheradressierungen.

Bei der Registeradressierung steht der Operand in einem Register und bei der unmittelbaren Adressierung steht der Operand direkt im Befehlswort. In beiden Fällen sind weder Adressrechnung noch ein zusätzlicher Speicherzugriff nötig.

Bei der einstufigen Speicheradressierung steht der Operand im Speicher und für die effektive Adresse ist nur *eine* Adressrechnung notwendig. Diese Adressrechnung kann einen oder mehrere Registerinhalte sowie einen im Befehl stehenden Verschiebewert oder einen Skalierungsfaktor, jedoch keinen weiteren Speicherinhalt betreffen.

Weniger gebräuchlich sind die zweistufigen Speicheradressierungen, bei denen mit einem Teilergebnis der Adressrechnung wiederum auf den Speicher zugegriffen wird, um einen weiteren Datenwert für die Adressrechnung zu holen. Es ist somit ein doppelter Speicherzugriff notwendig, bevor der Operand für die eigentliche Berechnung zur Verfügung steht.

Im Folgenden zeigen wir eine Auswahl von **Datenadressierungsarten**, die in heutigen Mikroprozessoren und Mikrocontrollern Verwendung finden. Abgesehen von der Register- und der unmittelbaren Adressierung sind dies allesamt einstufige Speicheradressierungsarten.

Datenadressierungsarten

Bei der **Registeradressierung** (*Register*) steht der Operand direkt in einem Register, siehe Abbildung 1.11. Der Bezeichner eines Registers im Assemblercode wird bei der Übersetzung in Maschinensprache in eine interne Nummer des Registers überführt. Wie bereits weiter oben erwähnt, handelt es sich hierbei je nach Befehl um ein Ganzzahl-, Gleitkomma- Multimedia- oder sonst ein Spezialregister.

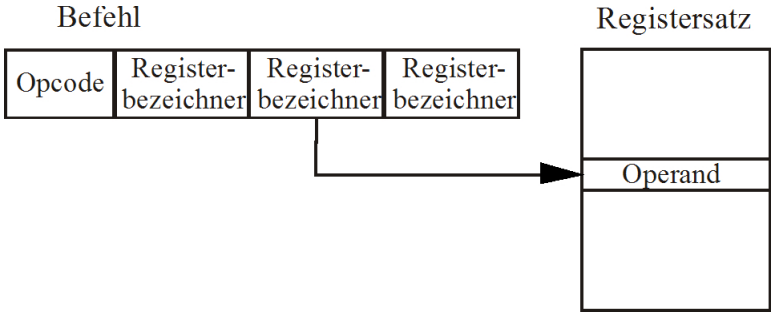

Abbildung 1.11: Registeradressierung.

Bei der **unmittelbaren Adressierung** (*immediate* oder *literal*) steht der Operand als Konstante direkt im Befehlswort, siehe Abbildung 1.12. Je nach Befehlslänge können hierfür unterschiedlich viele Bits zur Verfügung stehen.

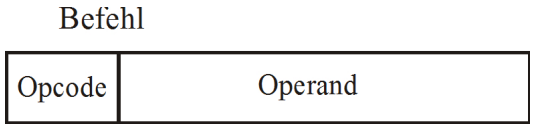

Abbildung 1.12: Unmittelbare Adressierung.

Bei der **direkten** oder **absoluten Adressierung** (*direct, absolute*) steht die Adresse eines Speicheroperanden im Befehlswort, siehe Abbildung 1.13. Auch hierfür können je nach Befehlslänge unterschiedlich viele Bits zur Verfügung stehen. Die Länge der Speicheradresse gibt die Größe des logischen bzw. physikalischen[19] Adressraums an.

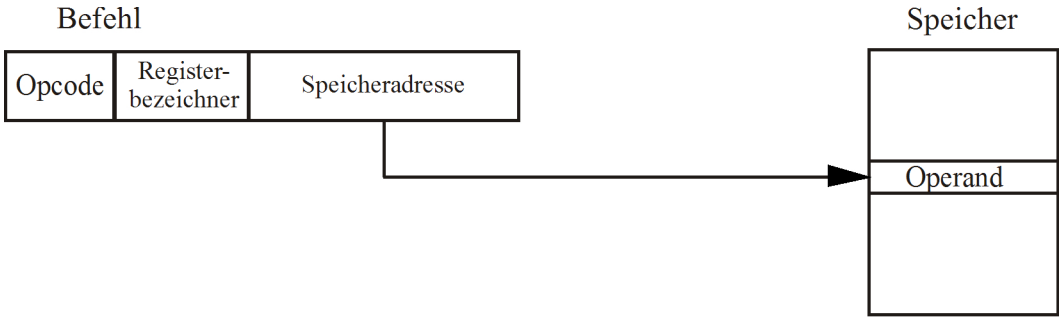

Abbildung 1.13: Direkte Adressierung.

[19] Falls keine virtuelle Speicherverwaltung verwendet wird.

Bei der **registerindirekten Adressierung** (*Register indirect* oder *Register deferred*) steht die Operandenadresse in einem Register. Der Inhalt des Registers dient als Zeiger auf eine Speicheradresse, siehe Abbildung 1.14.

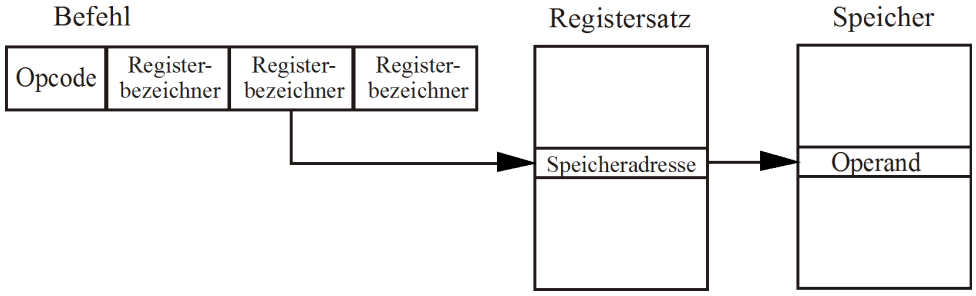

Abbildung 1.14: Registerindirekte Adressierung.

Als Spezialfälle der registerindirekten Adressierung können die **registerindirekten Adressierungen mit Autoinkrement/Autodekrement** (*Autoincrement/Autodecrement*) betrachtet werden. Diese arbeiten wie die registerindirekte Adressierung, inkrementieren bzw. dekrementieren aber den Registerinhalt. In Abbildung 1.14 ist dies die „Speicheradresse“, die vor oder nach dem Benutzen um die Länge des adressierten Operanden inkrementiert bzw. dekrementiert wird. Dementsprechend unterscheidet man die registerindirekte Adressierung mit **Präinkrement**, mit **Postinkrement**, mit **Prädekrement** und mit **Postdekrement** (üblich sind Postinkrement und Prädekrement)[20]. Diese Adressierungsarten sind für den Zugriff auf Feldern (*Arrays*) in Schleifen nützlich. Der Registerinhalt zeigt auf den Anfang oder das letzte Element eines Feldes und jeder Zugriff erhöht bzw. erniedrigt den Registerinhalt um die Länge eines Feldelements.

Bei der **registerindirekten Adressierung mit Verschiebung** (*Displacement, Register indirect with Displacement* oder *based*) wird die effektive Adresse eines Operanden als Summe eines Registerwerts und des Verschiebewerts (*Displacement*) berechnet, d. h. eines konstanten, vorzeichenbehafteten Werts, welcher im Befehl steht, siehe Abbildung 1.15.

[20] Im Befehlssatz oft geklammert angegeben als +(Reg), (Reg)+, −(Reg) und (Reg)−

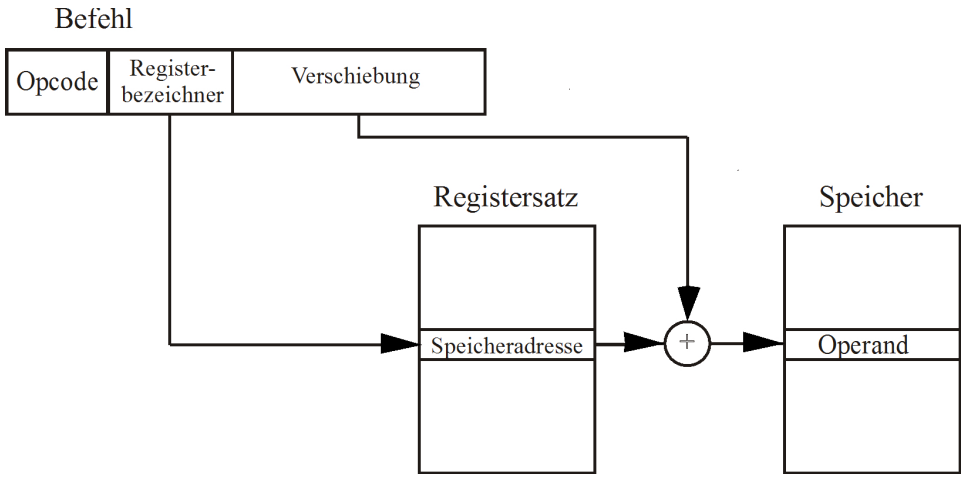

Abbildung 1.15: Registerindirekte Adressierung mit Verschiebung.

Die **indizierte Adressierung** (*indirect indexed*) errechnet die effektive Adresse als Summe eines Registerinhalts und dem Wert eines weiteren Registers, welches bei manchen Prozessoren als spezielles Indexregister vorliegt. Damit können Datenstrukturen beliebiger Größe und mit beliebigem Abstand durchlaufen werden. Angewendet wird die indizierte Adressierung auch beim Zugriff auf Tabellen, wobei der Index erst zur Laufzeit ermittelt wird, siehe Abbildung 1.16.

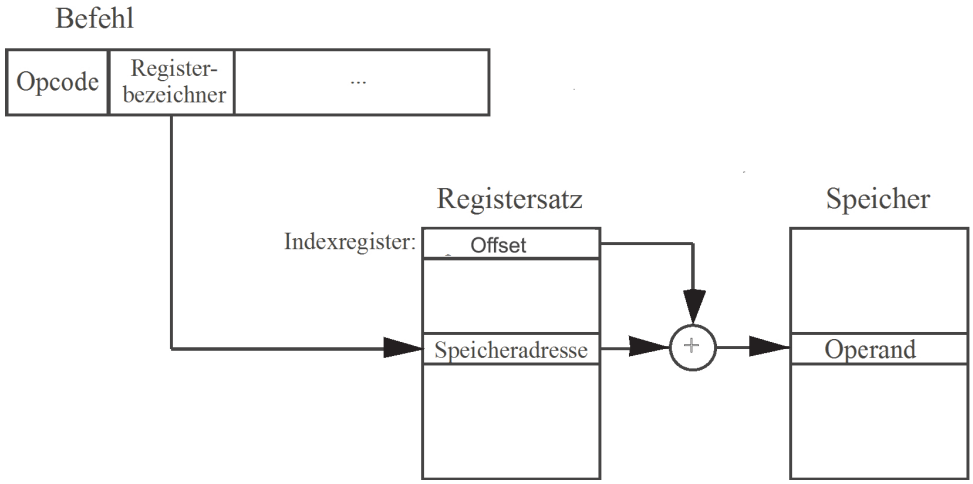

Abbildung 1.16: Indizierte Adressierung.

Die **indizierte Adressierung mit Verschiebung** (*indirect indexed with Displacement*) ist eine Erweiterung der indizierten Adressierung, bei der zur Summe der beiden Registerwerte noch ein im Befehl stehender Verschiebewert (*Displacement*) hinzuaddiert wird, siehe Abbildung 1.17.

Befehls-
adressierungsarten

Zur Änderung des Befehlszählregister (*Program Counter* – PC) durch Programmsteuerbefehle (bedingte oder unbedingte Sprünge sowie Unterprogrammaufruf und -rücksprung) sind nur zwei **Befehlsadressierungsarten** üblich:

Der **befehlszählerrelative Modus** (*PC-relative*) addiert einen Verschie-

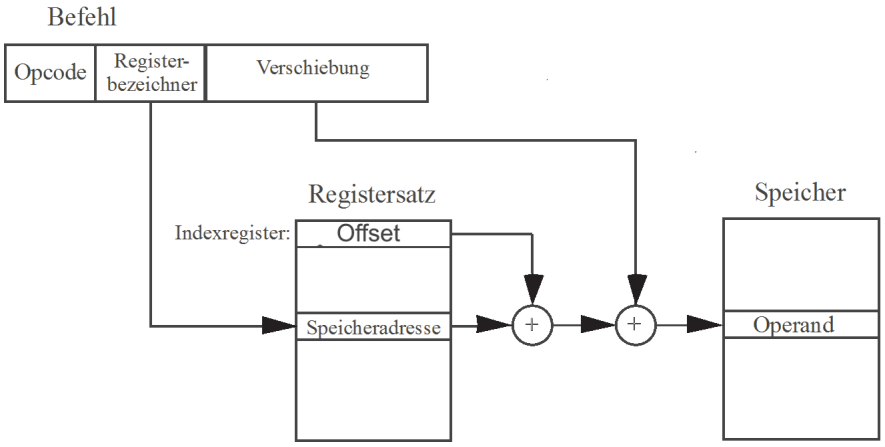

Abbildung 1.17: Indizierte Adressierung mit Verschiebung.

bewert (*Displacement, PC-Offset*) zum Inhalt des Befehlszählerregisters bzw. häufig auch zum Inhalt des inkrementierten Befehlszählers `PC + 4`, denn dieser wird bei Architekturen mit 32-Bit-Befehlsformat meist automatisch um vier erhöht. Die Sprungzieladressen sind häufig in der Nähe des augenblicklichen Befehlszählerwertes, so dass nur wenige Bits für den Verschiebewert im Befehl benötigt werden, siehe Abbildung 1.18.

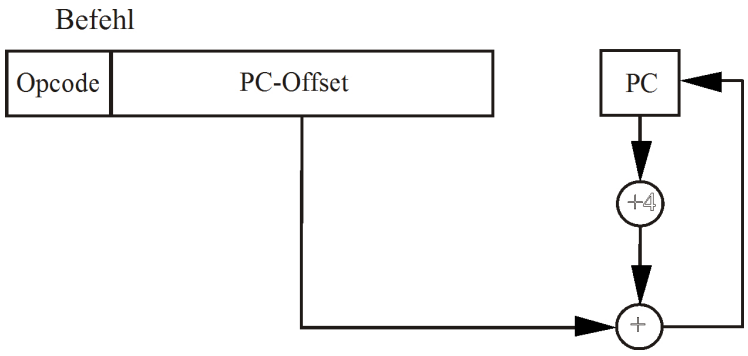

Abbildung 1.18: Befehlszählerrelative Adressierung.

Der **befehlszählerindirekte Modus** (*PC-indirect*) lädt den neuen Befehlszähler aus einem allgemeinen Register. Das Register dient als Zeiger auf eine Speicheradresse, bei der im Programmablauf fortgefahren wird, siehe Abbildung 1.19.

Tabelle 1.2 fasst die verschiedenen Adressierungsarten nochmals zusammen. In der Tabelle bezeichnet `Mem[R2]` den Inhalt des Speicherplatzes, dessen Adresse durch den Inhalt des Registers `R2` gegeben ist; `const, displ` können Dezimal-, Hexadezimal-, Oktal- oder Binärzahlen sein; `step` bezeichnet die Feldelementbreite und `inst_step` die Befehlsschrittweite in Abhängigkeit von der Befehlswortbreite, z. B. vier bei Vierbyte-Befehlswörtern.

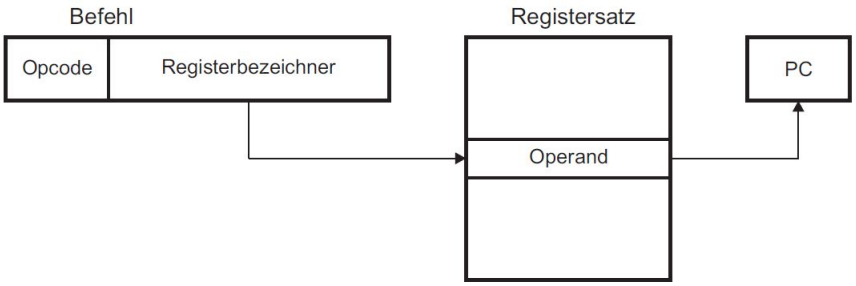

Abbildung 1.19: Befehlszählerindirekte Adressierung.

Tabelle 1.2: Beispielbefehle mit verschiedenen Adressierungsarten.

Adressierungsart	**Beispielbefehl**	**Bedeutung**
Register	load R1,R2	R1 ← R2
unmittelbar	load R1,const	R1 ← const
direkt, absolut	load R1,(const)	R1 ← Mem[const]
registerindirekt	load R1,(R2)	R1 ← Mem[R2]
Postinkrement	load R1,(R2)+	R1 ← Mem[R2] R2 ← R2 + step
Prädekrement	load R1,–(R2)	R2 ← R2 – step R1 ← Mem[R2]
registerindirekt mit Verschiebung	load R1,displ(R2)	R1 ← Mem[displ+R2]
indiziert	load R1,(R2,R3)	R1 ← Mem[R2 + R3]
indiziert mit Verschiebung	load R1,displ(R2,R3)	R1 ←Mem[displ+R2+R3]
befehlszählerrelativ	branch displ	PC←PC+inst_step+displ, falls Sprung genommen PC ← PC+inst_step , sonst
befehlszählerindirekt	branch R2	PC ← R2, falls Sprung genommen PC ← PC+inst_step, sonst

Selbsttestaufgabe 1.2
Welche einfacheren Adressierungsarten lassen sich durch die registerindirekte Adressierung mit Verschiebung ersetzen?

1.3.7 CISC- und RISC-Prinzipien

Bei der Entwicklung der Großrechner in den 60er und 70er Jahren hatten technologische Bedingungen wie der teure und langsame Hauptspeicher[21] zu einer immer größeren Komplexität der Rechnerarchitekturen geführt. Um den teuren Hauptspeicher optimal zu nutzen, wurden komplexe Maschinenbefehle entworfen, die mehrere Operationen mit nur einem Opcode codieren können. Damit konnte auch der im Verhältnis zum Prozessor langsame Speicherzugriff überbrückt werden, denn ein Maschinenbefehl umfasste genügend Operationen, um die Zentraleinheit für mehrere, eventuell sogar mehrere Dutzend Prozessortakte zu beschäftigen.

Mikroprogrammierung

Eine auch heute noch übliche Implementierungstechnik, um den Ablauf komplexer Maschinenbefehle zu steuern, ist die **Mikroprogrammierung**, bei der ein Maschinenbefehl durch eine Folge von Mikrobefehlen implementiert wird. Diese Mikrobefehle stehen in einem Mikroprogrammspeicher innerhalb des Prozessors und werden von einer Mikroprogrammsteuereinheit interpretiert, vgl. Kurs 1608.

CISC und RISC

Die Komplexität der Großrechner zeigte sich in mächtigen Maschinenbefehlen, umfangreichen Befehlssätzen, vielen Befehlsformaten, Adressierungsarten und spezialisierten Registern. Rechner mit diesen Architekturcharakteristika wurden später mit dem Akronym **CISC** (*Complex Instruction Set Computers*) bezeichnet. Ähnliche Architekturcharakteristika zeigten sich auch bei den Intel-80x86- und den Motorola-680x0-Mikroprozessoren, die ab Ende der 70er Jahre entstanden. Etwa 1980 entwickelte sich ein gegenläufiger Trend, der die Prozessorarchitekturen bis heute maßgeblich beeinflusst hat: das **RISC**(*Reduced Instruction Set Computer*) genannte Architekturkonzept.

Bei der Untersuchung von Maschinenprogrammen war beobachtet worden, dass manche Maschinenbefehle und komplexe Adressierungsarten fast nie verwendet wurden. Komplexe Befehle lassen sich durch eine Folge einfacher Befehle ersetzen, komplexe Adressierungsarten entsprechend durch eine Folge einfacherer Adressierungsarten. Die vielen unterschiedlichen Adressierungsarten, Befehlsformate und -längen von CISC-Architekturen erschwerten die Codegenerierung durch den Compiler. Das komplexe Steuerwerk, das notwendig war, um einen großen Befehlssatz in Hardware zu implementieren, benötigte viel Chip-Fläche und führte zu langen Entwicklungszeiten.

Das RISC-Architekturkonzept wurde Ende der 70er Jahre mit dem Ziel entwickelt, durch vereinfachte Architekturen Rechner schneller und preisgünstiger zu machen. Die Speichertechnologie war billiger geworden, erste Mikroprozessoren konnten bereits seit Beginn der 1970er Jahre auf Silizium-Chips implementiert

[21] Es gab noch keine Cache-Speicher!

werden. Einfache Maschinenbefehle ermöglichten es, bei der Befehlsausführung das Pipelining-Prinzip anzuwenden: Möglichst alle Befehle sollen dabei so implementierbar sein, dass pro Prozessortakt die Ausführung eines Maschinenbefehls in der Pipeline beendet wird. Durch die Implementierung mittels einer Befehlspipeline kann (für die meisten Befehle) auf die Mikroprogrammierung verzichtet werden. Stattdessen werden die Befehle (Opcodes) durch ein schnelles Schaltnetz in einem einzigen Taktzyklus decodiert. Nur sehr komplexe Befehle, wie z. B. bei den PC-Prozessoren von Intel oder AMD, werden weiterhin durch ein Mikroprogramm-Steuerwerk dekodiert und intern in eine Reihe von RISC-Befehlen überführt.

Dies war natürlich nur durch eine konsequente Verschlankung der Prozessorarchitektur möglich. Folgende Eigenschaften charakterisieren frühe RISC-Architekturen:

- Der Befehlssatz besteht aus wenigen, unbedingt notwendigen Befehlen (Anzahl ≤ 128) und Befehlsformaten (Anzahl ≤ 4) mit einer einheitlichen Befehlslänge von 32 Bit und mit nur wenigen Adressierungsarten (Anzahl ≤ 4). Damit wird die Implementierung des Steuerwerks erheblich vereinfacht und auf dem Prozessor-Chip Platz für weitere Funktionseinheiten geschaffen.

- Eine große Registerzahl von mindestens 32 allgemein verwendbaren Registern ist vorhanden.

- Der Zugriff auf den Speicher erfolgt nur über Lade-/Speicherbefehle. Alle anderen Befehle, d. h. insbesondere auch die arithmetischen Befehle, beziehen ihre Operanden aus den Registern und speichern ihre Resultate in Registern. Dieses Prinzip der Register-Register-Architektur ist für RISC-Rechner kennzeichnend und hat sich heute bei allen neu entwickelten Prozessorarchitekturen durchgesetzt.

- Weiterhin wurde bei den frühen RISC-Rechnern die Überwachung der Befehls-Pipeline von der Hardware in die Software verlegt, d. h., Abhängigkeiten zwischen den Befehlen und bei der Benutzung der Ressourcen des Prozessors mussten bei der Codeerzeugung bedacht werden. Die Implementierungstechnik des Befehls-Pipelining wurde damals zur Architektur hin offen gelegt. Eine klare Trennung von (Befehlssatz-)Architektur und Mikroarchitektur war für diese Rechner nicht möglich. Das gilt für heutige Mikroprozessoren – abgesehen von wenigen Ausnahmen – nicht mehr, jedoch ist die Beachtung der Mikroarchitektur für Compiler-Optimierungen auch weiterhin notwendig.

Die RISC-Charakteristika galten zunächst nur für den Entwurf von Prozessorarchitekturen, die keine Gleitkomma- und Multimediabefehle umfassten, da die Chip-Fläche einfach noch zu klein war, um solch komplexe Einheiten aufzunehmen. Inzwischen können auch Gleitkomma- und Multimediaeinheiten auf dem Prozessor-Chip untergebracht werden. Gleitkommabefehle benötigen

nach heutiger Implementierungstechnik üblicherweise drei Takte in der Ausführungsstufe einer Befehls-Pipeline. Da die Gleitkommaeinheiten intern als Pipelines aufgebaut sind, können sie jedoch ebenfalls pro Takt ein Resultat liefern.

Skalar und superskalar

RISC-Prozessoren, die das Entwurfsziel von durchschnittlich *einer* Befehlsausführung pro Takt erreichen, werden als **skalare RISC-Prozessoren** bezeichnet. Doch gibt es heute keinen Grund, bei der Forderung nach *einer* Befehlsausführung pro Takt stehen zu bleiben. Die Superskalartechnik ermöglicht es heute, pro Takt mehreren Ausführungseinheiten gleichzeitig bis zu sechs Befehle zuzuordnen und eine gleiche Anzahl von Befehlsausführungen pro Takt zu beenden. Solche Prozessoren werden als **superskalare (RISC)-Prozessoren** bezeichnet, da die oben definierten RISC-Charakteristika auch heute noch weitgehend beibehalten werden. Solche hochperformante Prozessoren werden in Kapitel 4 ausführlich vorgestellt.

Im Folgenden werden wir exemplarisch die RISC-Prozessor-Architektur des recht verbreiteten MIPS-Prozessors (*Microprocessor without Interlocked Pipeline Stages*) behandeln. Wir werden den MIPS-Befehlssatz vorstellen und Assembler-Programme für diesen analysieren. Außerdem werden wir einen MIPS-Simulator (*MARS, MIPS Assembler and Runtime Simulator*) vorstellen, mit dem Sie sich intensiv mit dem MIPS auseinandersetzen, eigene Programme schreiben und testen können.

1.4 Die MIPS-Architektur

In diesem und dem folgenden eher praktisch ausgerichteten Abschnitt wird die Architektur des MIPS-Prozessors vorgestellt und Sie lernen, wie man Assemblerprogramme für den MIPS schreibt und diese in einem Simulator ausführen kann. Der MIPS wurde ab 1981 an der Stanford-Universität entwickelt und wird heute von der MIPS Technologies Inc. an verschiedene Hardwarehersteller lizenziert. Ziel dabei war es, durch eine geeignete Definition der Befehlssatzarchitektur eine effiziente Pipeline-Architektur zu ermöglichen. Verbreitung findet der MIPS-Prozessor heute besonders in Eingebetteten Systemen wie z. B. in Druckern, Netzwerkgeräten oder Mobiltelefonen. Auch bei modernen Tablet-Computern werden mehrkernige MIPS-Prozessoren verwendet. Aufgrund der Pipeline-Mikroarchitektur des MIPS erreicht man einen hohen Befehlsdurchsatz bei geringer Leistungsaufnahme. Der MIPS gilt durch Eigenschaften wie eine einheitliche Befehlslänge, die Realisierung einer Load-Store-Architektur und die Beschränkung auf nur wenige Adressierungsarten als ein klassischer Vertreter der RISC-Architektur. Ursprünglich handelte es sich um eine reine 32-Bit-Architektur, welche im Laufe der Entwicklung aber auf 64-Bit erweitert wurde[22]. Es können zudem mehrere Koprozessoren angesprochen werden, welche unter anderem für Gleitkommaoperationen zuständig sind. Die Ausführungen in diesem Abschnitt beziehen sich auf die Version R2000 des MIPS-Prozessors. Ein

[22] Im Kurs beschränken wir uns auf eine reine 32-Bit Version.