Kurs 01609 Computersysteme II

Autor: Prof. Dr. T. Ungerer

Überarbeitung:
Dr. H. Bähring
Prof. Dr. J. Keller
Prof. Dr. W. Schiffmann

Kurseinheiten 1 – 4

Inhaltsverzeichnis

Vorwort

Liebe Studierende,

wir begrüßen Sie herzlich zum Kurs Computersysteme II (1609). Dieser Kurs führt Sie in die Grundlagen der *Rechnerarchitektur* ein. Neben grundlegenden Prozessortechniken werden die Architekturkonzepte hochperformanter Mikroprozessoren behandelt und an Beispielen verdeutlicht. Der Kurs befasst sich aber auch mit der Speicherhierarchie und Zukunftstechniken für Mikroprozessoren. So werden z. B. mehrstufige Caches verwendet, um die hohen Taktraten moderner Mikroprozessoren voll auszunutzen. Die Leistungsgrenzen heutiger Prozessortechniken können nur durch parallel arbeitende Architekturen überwunden werden. Auf der Prozessorebene kann der Durchsatz durch eine mehrfädige Befehlsverarbeitung weiter gesteigert werden. Parallele Rechnersysteme verwenden entweder mehrere Prozessoren oder parallel arbeitende Rechenwerke. Bei Multiprozessorsystemen muss die Aufgabenstellung in Teilprobleme zerlegt werden, die durch miteinander kommunizierende Programme gleichzeitig bearbeitet werden. Im Kurs werden sowohl nachrichten- als auch speichergekoppelte Parallelrechnerysteme behandelt.

Der Kurs wurde zum Sommersemester 2009 überarbeitet. Dabei wurde ein in sich geschlossener Kurstext erstellt, der auch über ein gemeinsames Inhalts-, Literatur- und Stichwortverzeichnis (Index) verfügt. Die einzelnen Kapitel stimmen mit den jeweiligen Kurseinheiten 1-4 überein, d.h. die Bearbeitungszeiträume und zugehörigen Einsendetermine beziehen sich nun auf die Kapitelnummern. Der bisherige Kursinhalt wurde stark gekürzt und auf die wesentlichen Grundlagen reduziert. Wir hoffen, dass dadurch der überarbeitete Kurs leichter zu verstehen und der Bearbeitungsaufwand im Vergleich zur bisherigen Fassung geringer ist.

Wichtiger Hinweis

Diesem Kurs liegt in weiten Teilen das Buch „Mikrocontroller und Mikroprozessoren“ von U. Brinkschulte und T. Ungerer [3] zugrunde. Dort können Sie sich intensiv über Themen informieren, die aus Platzgründen in diesem Kurs nicht oder nicht sehr tiefgehend behandelt werden können. Insbesondere finden Sie dort zu allen behandelten Themen ein ausführliches Literaturverzeichnis.

Wir wünschen Ihnen viel Spaß beim Bearbeiten des Kurses!

Ihre Kursbetreuer

Kapitel 1

Grundlegende Prozessortechniken

Kapitelinhalt

Zusammenfassung

Wir werden in diesem Kapitel die wesentlichen Prozessortechniken, d.h. die für den System- und Assemblerprogrammierer zu beachtenden Details der Befehlssatzarchitektur und die Pipelining-Mechanismen, ausführlich behandeln. Die Grundlagen dieser Techniken wurden bereits in KE4 von Kurs 1608 kurz dargestellt.

Wir beginnen in Abschnitt 1 mit einer kurzen Einführung in die Rechnerarchitektur. Im 2. Abschnitt führen wir in die wichtigsten Eigenschaften der Befehlssatzarchitekturen ein. Am Ende des Abschnitts wird näher auf die Unterschiede zwischen CISC- und RISC-Befehlssätzen (Complex bzw. Reduced Instruction Set Computer) eingegangen.

Der 3. Abschnitt behandelt die besonderen Eigenschaften der RISC-Architekturen und zeigt als Anwendung der im 1. Abschnitt gelernten Techniken die Befehlssatzarchitektur des DLX-Prozessors – eines von den bekannten amerikanischen Wissenschaftlern Hennessy und Patterson entworfenen „Lehrprozessors“, der mit den in Steuerungssystemen, wie z.B. Navigationssystemen, Empfängern fürs Satellitenfernsehen und Spieleconsolen, häufig verwendeten MIPS-Prozessoren fast identisch ist. Die zu diesem Kapitel gehörenden Programmieraufgaben in Maschinensprache (bzw. Assembler) werden mit dem vorgestellten DLX-Befehlssatz programmiert.

Abschnitt 4 behandelt die Umsetzung des von-Neumann-Prinzips durch Befehls-Pipelining. Abschnitt 5 führt in den Entwurf von Pipelining-Prozessoren ein. Die Implementierung der Befehlssatzarchitektur des DLX-Prozessors durch eine fünfstufige Pipeline wird im Detail behandelt, ebenso die Lösungen zu auftretenden Pipeline-Konflikten.

Dieses Kapitel beschränkt sich auf Prozessoren, die pro Takt in jeder Stufe der Pipeline nur einen Befehl ausführen können. Die hier gelernten Programmier- und Prozessortechniken finden ihre direkte Anwendung bei den in heutigen Steuerungssystemen eingesetzten Mikrocontrollern. (Diese sog. „eingebetteten Systeme“ werden im Kurs 1706 ausführlich behandelt.) Abschnitt 6 verweist auf weitere Aspekte des Befehls-Pipelining. Dieser Abschnitt bildet den Übergang zum Kapitel 2, das der Superskalartechnik gewidmet ist, die insbesondere in den hochperformanten Mikroprozessoren moderner Personal Computer (PC) eingesetzt wird.

Lernziele

Die Lernziele dieses Kapitels sind:

- Zusammenhang zwischen Prozessor- und Mikroarchitektur,
- Komponenten einer Befehlssatzarchitektur,
- CISC- und RISC-Prinzipien,
- Mikroarchitektur einfacher Prozessoren und Prozessorkerne,
- Befehlspipelining.

1.1 Rechnerarchitektur

In der letzten Kurseinheit des Kurses 1608 haben Sie gelernt, wie ein Computer nach dem von Neumann Prinzip [4] aufgebaut ist. Er besteht aus vier Funktionseinheiten: dem Rechen- und Leitwerk, die zusammen den Prozessor bilden, dem Speicher und einer Ein-/Ausgabe. Das Zusammenspiel dieser vier Komponenten bestimmt die Leistungsfähigkeit eines Computers. Speicher und Ein-/Ausgabe werden über die Systembusschnittstelle angesprochen und belegen jeweils verschiedene Bereiche in dem vom Prozessor ansprechbaren Adressraum. 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 Controllerbausteinen, die Peripheriegeräte zur Interaktion mit dem Menschen (Tastatur, Monitor, Maus, Drucker), anderen Computern (lokale Netzwerke, Internet) oder nichtflüchtige Speichermedien (Festplatten, CDs/DVD) unterstützen. Diese Controllerbausteine 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 den betreffenden Prozessor zu erstellen. Natürlich muss auch der Compiler dieses Programmiermodell kennen, um Hochsprachenprogramme in Maschinensprache zu übersetzen. Da dieses Programmiermodell im Wesentlichen durch den Befehlssatz des Prozessors festgelegt ist, spricht man von der *Befehlssatzarchitektur* (Instruction Set Architectur – ISA) oder einfach auch nur von der *Architektur* eines Prozessors. Sie beschreibt zwar das Verhalten des Prozessors nicht aber seine Implementierung, die entweder durch die *logische Organisation* oder durch die tatsächliche *technologische Realisierung* beschrieben werden kann (s. Abbildung 1.1).

Befehlssatzarchitektur und Architektur werden synonym verwendet.

Abbildung 1.1: Ebenen der Rechnerarchitektur.

Ähnlich wie ein Architekt zunächst mit dem Bauherrn die gewünschten Eigenschaften eines Bauwerks festlegt und diese dann optimal 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. Zunächst muss die Befehlssatzarchitektur des Prozessors festgelegt werden. Sie beschreibt die für die spätere Anwendung benötigten prozessorinternen Speichermöglichkeiten, Operationen und Datenformate. Aus dieser Architekturbeschreibung leitet der Rechnerarchitekt dann eine logische Struktur zur Implementierung ab (diese wird auch *Mikroarchitektur* genannt). Dabei legt er fest, welche Funktionseinheiten (z.B. Register(sätze), ALUs, Mul-

Mikroarchitektur

tiplexer 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).

Im Laufe der Entwicklung von Computersystemen kamen verschiedene Technologien zur Realisierung von Prozessoren zum Einsatz. Es begann mit elektromechanischen Bauelementen, wenig später verwendete man Elektronenröhren bzw. einzelne Transistoren und schließlich ab Ende der fünfziger Jahre integrierte Schaltkreise (Integrated Circuits – ICs), die ganze Schaltungen mit mehreren Transistoren auf einem einzigen Halbleiterchip realisieren. Die Integrationsdichte hat sich seit der Einführung integrierter Schaltkreise stetig gesteigert. Gordon Moore, Mitbegründer des weltweit bekannten Prozessorherstellers Intel, stellte 1965 in einem Beitrag zur Zeitschrift „Electronics“ fest, dass sich bis zu diesem Zeitpunkt die Anzahl der Transistoren pro IC jedes Jahr in etwa verdoppelt hatte. Dieser als *Moore'sches Gesetz* bekannt gewordene Zusammenhang hat sich prinzipiell bis heute fortgesetzt. Lediglich die „Zeitkonstante“ muss etwas größer angesetzt werden. Sie beträgt bei Speicher-ICs mit regulärer Struktur etwa 18 Monate und bei Prozessoren wegen der erhöhten Komplexität etwa 24 Monate.

Integrierte Schaltkreise

Moore'sches Gesetz

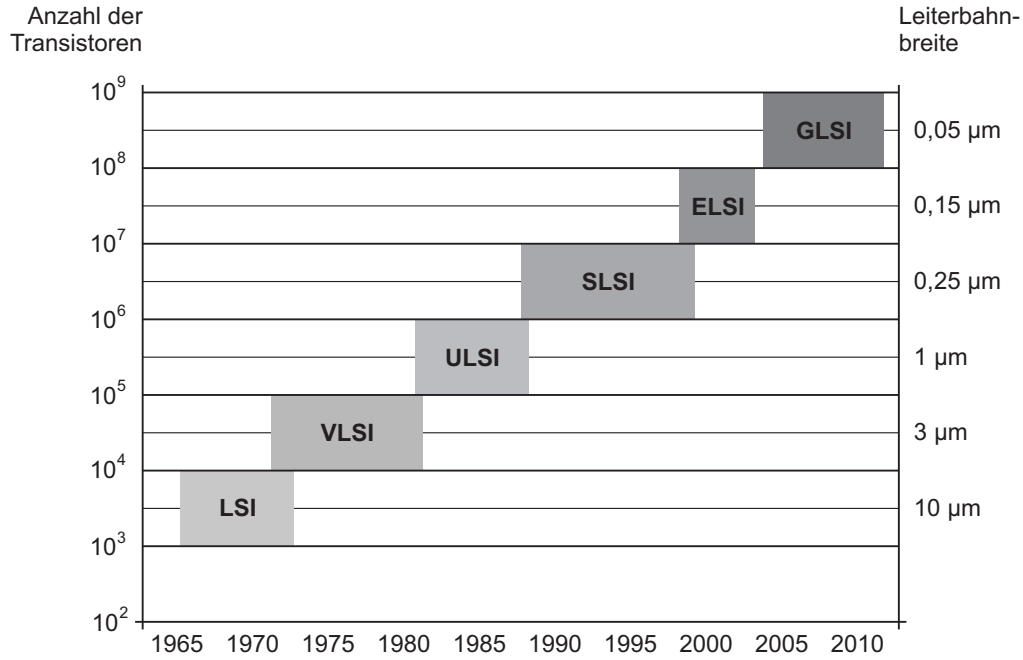

Abbildung 1.2: Entwicklung der Integrationsdichten und Strukturgrößen integrierter Schaltungen.

Um eine logische Organisation in Hardware umzusetzen, müssen mehrere Schichten geometrischer Strukturen (Chip layouts) auf eine Halbleiterscheibe (Wafer) geätzt werden. Diese implementieren dann elektronisch die einzelnen Funktionseinheiten. Im Laufe der Jahre wurde die Verfahrenstechnik zur Her-

stellung solcher Mikrochips stetig verbessert (s. Abbildung 1.2). Heute ist man in der Lage, kleinste Strukturen von weniger als 50 Nanometer (nm) herzustellen. 1 nm bedeutet 10^{-9} m, das ist ein Millionstel eines Millimeters. Durch die feineren Strukturen ist es möglich, immer mehr Transistoren auf einen Mikrochip zu integrieren und gleichzeitig auch die Taktfrequenzen zu erhöhen. Wegen der kleineren Strukturen können die Transistoren schneller schalten und können durch kürzere Verbindungsleitungen miteinander gekoppelt werden. Dadurch verkürzen sich auch die Laufzeiten zwischen den Transistoren.

Strukturgrößen kleiner als 50 nm

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

Seit 1960 entstanden verschiedene Generationen von Mikrochips (vgl. 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 (2007) bei der GSI-Technologie mit Strukturgrößen von ca. 50 Nanometer Integrationsdichten von fast einer Milliarde Transistoren möglich. Hauptproblem zukünftiger Entwicklungen ist vor allem die Kühlung der Mikrochips, deren Wärmedichten die von Kochplatten bei weitem übersteigen übersteigen.

Problem Kühlung

Es gibt verschiedene Arten von Mikrochips, die sich bezüglich der Regelmäßigkeit der geätzten Strukturen unterscheiden. Speicherchips sind am regelmäßigsten aufgebaut und können daher auch die höchsten Integrationsdichten erreichen. Die Prozessoren erreichen weniger als die Hälfte der Integrationsdichte von Speicherbausteinen. Neben Prozessoren gibt es auch noch anwendungsspezifische Bausteine, die entweder maskenprogrammiert werden (Application Specific Integrated Circuit – ASIC) oder elektrisch programmierbar sind (Field Programmable Gate Array – FPGA). Die Hersteller dieser Bausteine verwenden häufig statt der Transistorenanzahl als Komplexitätsmaß die Zahl der Gatteräquivalente. Als Faustregel gilt, dass zur Realisierung eines Gatters etwa vier Transistoren benötigt werden. Wenn Sie das Thema „technologische Realisierung“ näher interessiert, dann sollten Sie den Kurs 1721 belegen. Dort erfahren Sie mehr über den Aufbau und Entwurfsmethoden hochintegrierter Mikrochips.

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 sieb-

ziger 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 möglichst kurzen Programm zu codieren. Gleichzeitig wollte man die Maschinenprogrammierung ähnlich komfortabel gestalten wie die Programmierung in einer höheren Programmiersprache.

CISC-Prozessoren

Die Befehlssätze derartiger *CISC-Prozessoren* (Complex Instruction Set Computer) boten eine große Vielfalt an Adressierungsarten, die unterschiedlich viele Taktzyklen erforderten und nur mittels Mikroprogrammierung effizient implementiert werden konnten. Die Auslastung der einzelnen Prozessor-Funktionseinheiten war schlecht, da die Maschinenbefehle nur nacheinander abgearbeitet werden konnten. So wurde beispielsweise bei einem arithmetischen Befehl mit Speicherzugriff die ALU im Rechenwerk nur einen Taktzyklus lang genutzt, während der Befehl meist mehr als zehn Taktzyklen dauerte.

RISC-Prozessoren

Durch Einschränkungen der Befehlssatzarchitektur – vor allem bei den Adressierungsmöglichkeiten – erreichte man mit der Einführung so genannter *RISC-Prozessoren* (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 (Taktzyklen) zu beschränken. Damit war man dann in der Lage, das Pipelining-Prinzip mit einer überlappenden Verarbeitung dieser Mikroschritte anzuwenden. Im Idealfall kann mit diesem Ansatz die Auslastung der Prozessor-Funktionseinheiten auf 100% gesteigert und in jedem Taktzyklus ein Befehl beendet werden.

skalare RISC

RISC-Prozessoren wurden zunächst *skalar* ausgelegt, d.h. es gab nur eine ALU für die Ausführungsstufe. Wir werden uns in diesem Kapitel ausführlich mit einen Vertreter dieser skalaren RISC-Prozessoren befassen. Indem man mehrere ALUs (bzw. auch Gleitkommeinheiten) in der Ausführungsstufe hinzufügte, entstanden so genannte *Superskalar*-(RISC-)Prozessoren, die gleichzeitig mehrere Befehle holen, verplanen (Scheduling) und parallel ausführen. Auf diese Weise können mit jedem Taktzyklus mehrere Befehle beendet werden.

superskalare RISC

Während bei superskalaren RISC-Prozessoren die Verplanung der Befehle in einer entsprechenden Pipelinestufe mittels Hardware erfolgt, entstanden auch Prozessoren mit parallel arbeitenden Ausführungseinheiten, die auf einem *statischen* Scheduling basieren. Hier wird das Scheduling nicht zur Laufzeit sondern vom Compiler ausgeführt wird. Man spricht von VLIW-Prozessoren (Very Long Instruction Word), weil der Compiler sehr breite Maschinenbefehlcodes erzeugt, die dann im Prozessor zur Ansteuerung der einzelnen Ausführungseinheiten genutzt 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. Die zentrale Idee besteht darin, durch den Compiler das Hardware-Scheduling zu unterstützen und so den Hardwareaufwand im Prozessor zu verringern. Die Konzepte superskalarer RISC-, VLIW- und EPIC-Prozessoren werden im Kapitel 2 dieses Kurses ausführlicher behandelt. Im Folgenden beschränken wir uns auf die Konzepte skalarer RISC-Prozessoren.

statisches Scheduling bei VLIW

1.2 Befehlssatzarchitekturen

1.2.1 Prozessorarchitektur, Mikroarchitektur und Programmiermodell

Wiederholung: Prozessorarchitektur, Mikroarchitektur, Prozessortechniken

Eine **Prozessorarchitektur** definiert die Grenze zwischen Hardware und Software. Sie umfasst den für den Systemprogrammierer und für den Compiler sichtbaren Teil des Prozessors. Synonym wird deshalb oft auch von der **Befehlssatz-Architektur** (Instruction Set Architecture – ISA) oder dem **Programmiermodell** eines Prozessors gesprochen. Dazu gehören neben dem Befehlssatz (Menge der verfügbaren Befehle), das Befehlsformat, die Adressierungsarten, das System der Unterbrechungen und das Speichermodell, das sind die Register und der Adressraumaufbau. Eine Prozessorarchitektur betrifft jedoch keine Details der Hardware und der technischen Ausführung eines Prozessors, sondern nur sein äußeres Erscheinungsbild. Die internen Vorgänge werden ausgeklammert.

Eine **Mikroarchitektur** (entsprechend dem englischen Begriff *microarchitecture*) bezeichnet die Implementierung einer Prozessorarchitektur in einer speziellen Verkörperung der Architektur – also in einem Mikroprozessor. Dazu gehören die Hardware-Struktur und der Entwurf der Kontroll- und Datenpfade. Die Art und Stufenzahl des Befehls-Pipelining, der Grad der Verwendung der Superskalartechnik, Art und Anzahl der internen Ausführungseinheiten eines Mikroprozessors sowie Einsatz und Organisation von Primär-Cache-Speichern zählen zu den Mikroarchitekturtechniken. Die Mikroarchitektur definiert also die logische Organisation eines Prozessors. Diese Eigenschaften werden von der Befehlssatzarchitektur nicht erfasst. Systemprogrammierer und optimierende Compiler benötigen jedoch auch die Kenntnis von Mikroarchitektureigenschaften, um effizienten Code für einen speziellen Mikroprozessor zu erzeugen.

Architektur- und Implementierungstechniken werden beide im Folgenden als **Prozessortechniken** bezeichnet. Die Architektur macht die Benutzerprogramme von den Mikroprozessoren, auf denen sie ausgeführt werden, unabhängig. Alle Mikroprozessoren, die derselben Architekturspezifikation folgen, sind binärkompatibel zueinander. Die Implementierungstechniken zeigen sich bei den unterschiedlichen Verarbeitungsgeschwindigkeiten.

Definition: 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 ist nur eine Abwärtskompatibilität mit den älteren bzw. einfacheren Prozessoren der Familie gegeben, d.h. der Objektcode der älteren läuft auf den neueren Prozessoren, doch nicht umgekehrt.

Programmiermodell eines Prozessors

Die Programmierersicht eines Prozessors lässt sich durch Beantworten der folgenden fünf Fragen einführen:

- Wie werden Daten repräsentiert?
- Wo werden die Daten gespeichert?
- Welche Operationen können auf den Daten ausgeführt werden?

- Wie werden die Befehle codiert?
- Wie wird auf die Operanden zugegriffen?

Die Antworten auf diese Fragen definieren die Prozessorarchitektur bzw. das Programmiermodell. Im Folgenden werden diese Eigenschaften im Einzelnen kurz vorgestellt.

1.2.2 Datenformate

Wie in den höheren Programmiersprachen können auch in maschinennahen Sprachen die Daten verschiedenen Datenformaten zugeordnet werden. Diese bestimmen, wie die Daten repräsentiert werden.

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. MOVB, um ein

Mikrocontroller

Byte zu übertragen. 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.

n-Bit-Prozessor

Ein 32-Bit-Datenwort reflektiert dabei die Sicht eines 32-Bit-Prozessors bzw. ein 16-Bit-Datenwort diejenige eines 16-Bit-Prozessors oder -Mikrocontrollers. (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 meist auch die Breite der effektiven Adresse 32 Bit. Dies ist heute insbesondere bei den PC-Prozessoren der Intel Pentium-Familie der Fall. Workstation-Prozessoren, wie die Sun UltraSPARC-Prozessoren und die Itanium-Prozessoren der Firma Intel, sind 64-Bit-Prozessoren. Bei Mikrocontrollern sind oft auch 8-Bit- und 16-Bit-Prozessorkerne üblich.

Die Befehlssätze unterstützen jedoch auch Datenformate wie zum Beispiel Einzelbit-, Ganzzahl- (*integer*), Gleitkomma- und Multimediaformate, die mit den Datenformaten in Hochsprachen enger verwandt sind.

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

Ganzzahldatenformate

Ganzzahldatenformate (s. Abbildung 1.3) sind unterschiedlich lang und können mit Vorzeichen (*signed*) oder ohne Vorzeichen (*unsigned*) definiert sein. 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 Bitpositionen eines Bytes, also nur vier BCD-Zahlen pro 32-Bit-Wort. Der ASCII-Code belegt ein

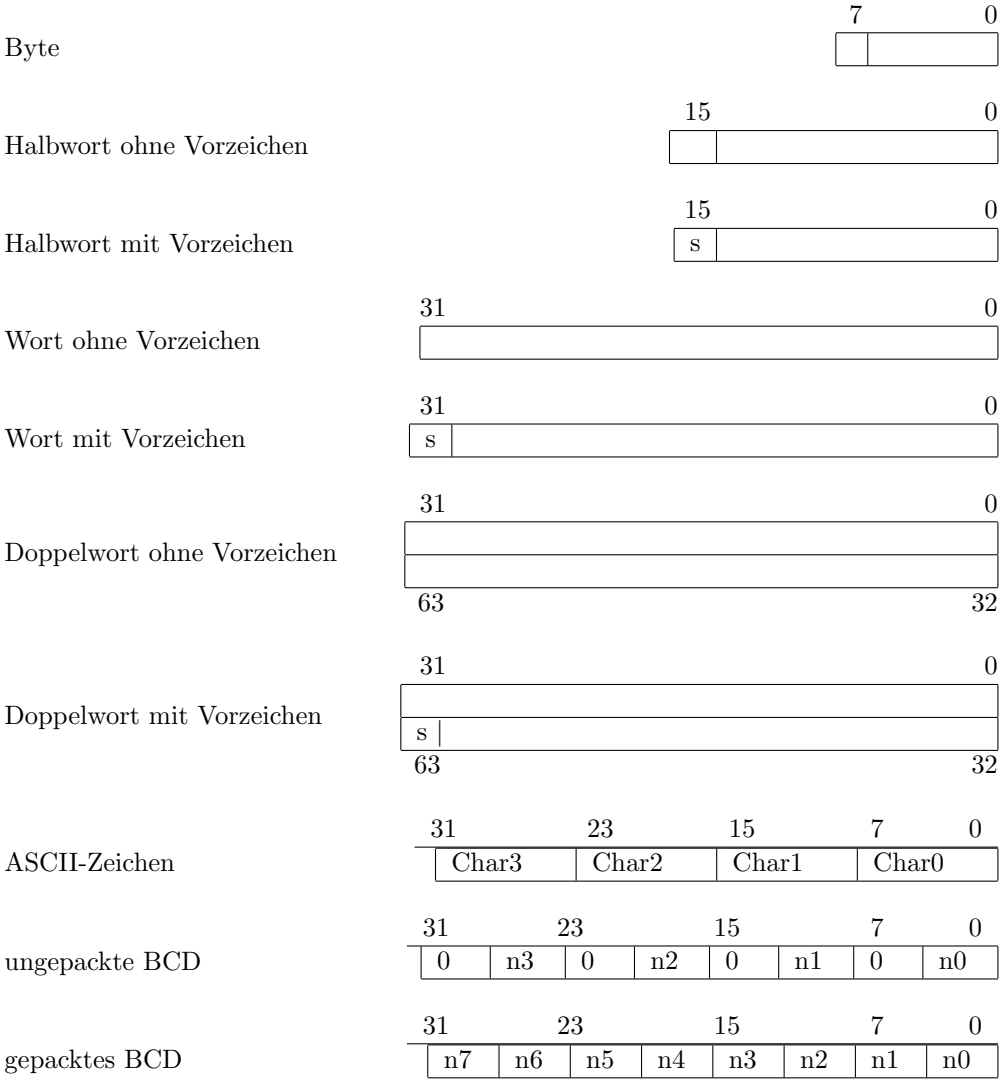

Abbildung 1.3: Ganzzahlendatenformate.

Byte pro Zeichen, so dass vier ASCII-codierte Zeichen in einem 32-Bit-Wort untergebracht werden.

Gleitkommadatenformate

Die Gleitkommadatenformate (s. Abbildung 1.4) wurden mit dem IEEE 754-1985-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 verwenden 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 die Mantisse oder den Signifikanten. Die führende Eins in der obigen Gleichung ist

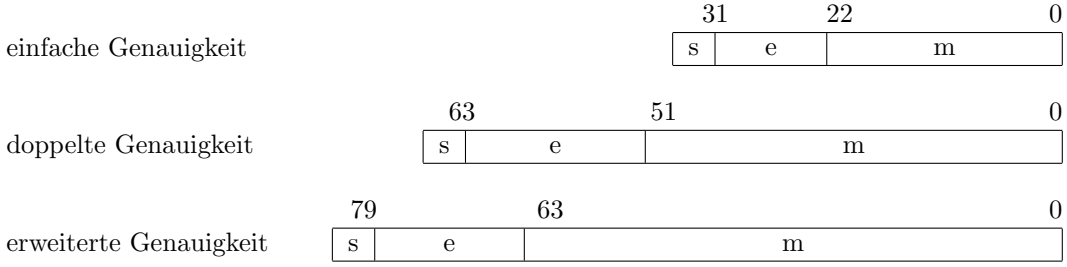

Abbildung 1.4: Gleitkomma-Datenformate.

implizit vorhanden und benötigt kein Bit im Mantissenfeld (s. Abbildung 1.4). Für die Mantisse m gilt:

$m = .m_1 \cdots m_p$ mit
$p = 23$ für die einfache,
$p = 52$ für die doppelte und
$p = 63$ für die erweiterte Genauigkeit (in diesem Fall wird die führende Eins der Mantisse mit dargestellt)

Die Verschiebung b ist definiert als:

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

wobei ne die Anzahl der Exponentenbits (8 bei einfacher, 11 bei doppelter und 15 bei erweiterter Genauigkeit) bedeutet.

Den richtigen Exponenten E erhält man aus der Gleichung:

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

Multimedia-datenformate

Multimediadatenformate definieren 64 oder 128 Bit breite Wörter. Man unterscheidet zwei Arten von Multimediadatenformaten (s. Abbildung 1.5): Die bitfeldnorientierten 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 bis vier einfach genaue Gleitkommazahlen.

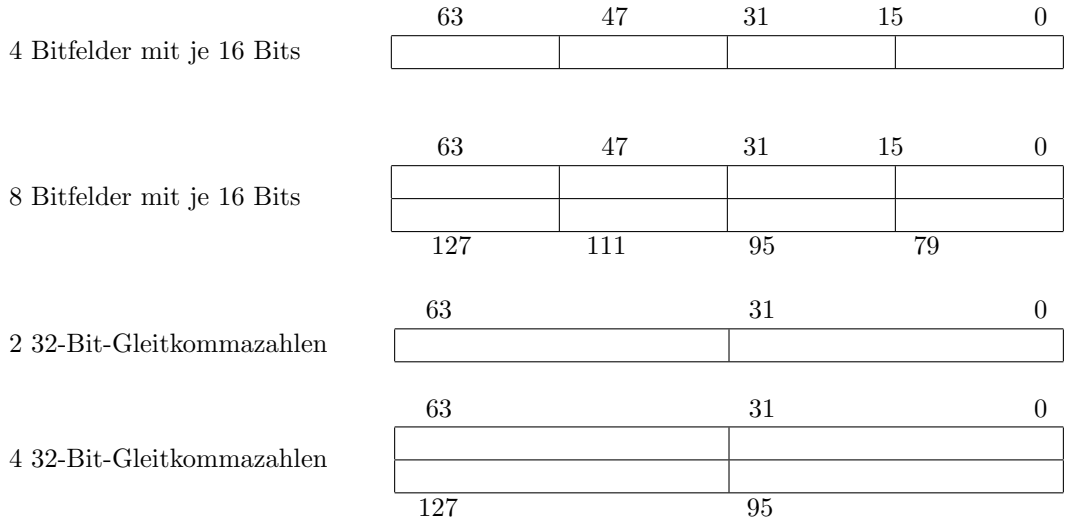

Abbildung 1.5: Beispiele bitfeld- und graphikorientierter Multimediadatenformate mit 64- und 128-Bitformaten.

1.2.3 Adressraumorganisation

Registerarten

Die Adressraumorganisation bestimmt, wo die Daten gespeichert werden. Jeder Prozessor enthält eine kleine Anzahl von **Registern**, d.h. schnellen Speicherplätzen, auf die in einem Taktzyklus zugegriffen werden kann. Bei den heute üblichen Pipeline-Prozessoren wird in der Operandenholphase der Befehls-Pipeline auf die Registeroperanden zugegriffen und in der Resultatspeicherphase in Register zurückgespeichert. Diese Register können **allgemeine Register** (auch Universalregister oder Allzweckregister genannt), **Multimediaregister**, **Gleitkommaregister** oder **Spezialregister** (Befehlszähler, Statusregister etc.) sein. Heutige Mikroprozessoren verwenden meist 32 allgemeine Register R0 – R31 von je 32 oder 64 Bit und zusätzlich 32 Gleitkommaregister F0,...,F31 von je 64 oder 80 Bit. Oft ist außerdem das Universalregister R0 fest mit dem Wert 0 verdrahtet. Die Multimediaregister stehen für sich oder sind mit den Gleitkommaregistern identisch. All diese für den Programmierer sichtbaren Register werden als **Architekturregister** bezeichnet, da sie in der „Architektur“ sichtbar sind. Im Gegensatz dazu sind die auf heutigen Mikroprozessoren oft vorhandenen physikalischen Register oder Umbenennungspufferregister (vgl. Abschnitt 2.2.3 in Kapitel 2) nicht in der Architektur sichtbar.

Um Daten zu speichern, werden mehrere Adressräume unterschieden. Diese sind neben den Registern und Spezialregistern insbesondere Speicheradressräume für

- den Laufzeitstapel (*run-time stack)*, insbesondere zur Ablage von Rücksprungadressen aus Unterprogrammen und Unterbrechungsroutinen,
- den *Heap*, einen Speicherbereich für dynamisch zur Programmlaufzeit erzeugte Datenobjekte,
- die Ein-/Ausgabe- und Steuerdaten.

Wortadresse:	x0				x4			
Bytestelle im Wort:	7	6	5	4	3	2	1	0

Wortadresse:	x0				x4			
Bytestelle im Wort:	0	1	2	3	4	5	6	7

Abbildung 1.6: Big-endian- (oben) und Little-endian-Formate (unten).

Adressierbarkeit

Abgesehen von den Registern werden alle anderen Adressräume meist auf einen einzigen, durchgehend adressierten Adressraum abgebildet. Dieser kann **byteadressierbar** sein, d.h. jedes Byte in einem Speicherwort kann einzeln adressiert werden. Oft sind heutige Prozessoren jedoch **wortadressierbar**, so dass nur 16-, 32- oder 64-Bit-Wörter direkt adressiert werden können. In der Regel muss deshalb der Zugriff auf Speicherwörter **ausgerichtet** (*aligned*) sein. Ein Zugriff zu einem 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.

Ein 64-Bit-Wort umfasst 8 Bytes und benötigt auf einem byteadressierbaren Prozessor 8 Speicheradressen. Für die Speicherung im Hauptspeicher unterscheidet man zwei Arten der Byteanordnung innerhalb eines Wortes (s. Abbildung 1.6):

Little- und Big-endian-Format

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

Für die Assemblerprogrammierung ist diese Unterscheidung häufig irrelevant, da beim Laden eines Operanden 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 angewandt wird. Beachten muss man das Byteanordnungsformat eines Prozessors insbesondere beim direkten Zugriff auf einen Speicherplatz als Byte oder Wort oder bei der Arbeit mit einem *Debugger*, also einem Software-Werkzeug[1] zum Finden von Fehlern in Programmen. Die Bytereihenfolge wird ein Problem, wenn Daten zwischen zwei Rechnern verschiedener Architektur ausgetauscht werden. Heute setzt sich insbesondere durch die Bedeutung von Rechnernetzen das Big-endian-Format durch, das häufig auch als Netzwerk-Format bezeichnet wird.

1.2.4 Befehlssatz

Befehlssatz

Der **Befehlssatz** (*instruction set*) definiert, welche Operationen auf den Daten

[1] Beim Entwurf von Steuerungssystemen enthält der Debugger meist auch noch eine

ausgeführt werden können. Er legt daher die Grundoperationen eines Prozessors fest. Man kann 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 Wortes um eine Anzahl von Stellen entweder nach links oder nach rechts bzw. rotieren die Bits nach links oder rechts (s. Abbildung 1.7). 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 Bitkette betrachtet. Ein sogenanntes Übertragsbit (*carry flag*) im Prozessorstatusregister kann wahlweise mitbenutzt oder als zusätzliches Bit einbezogen werden.

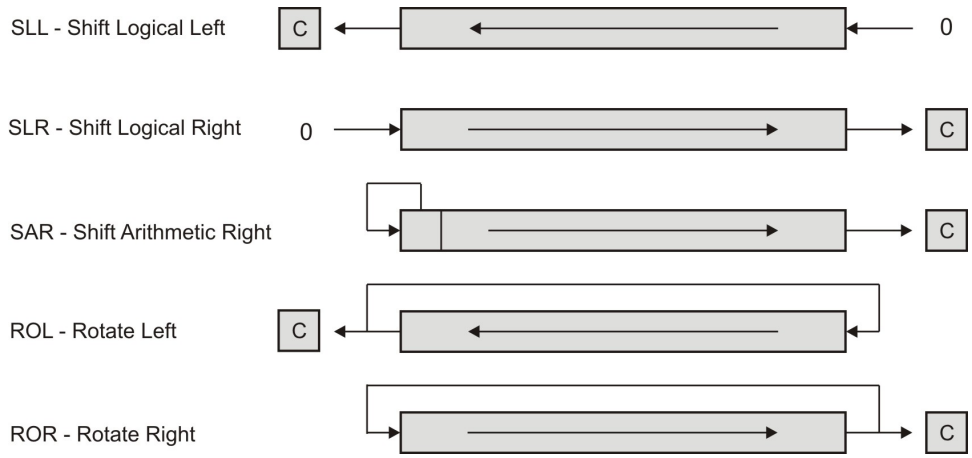

Abbildung 1.7: Einige Schiebe- und Rotationsbefehle.

Hardware-Komponente.

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

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.

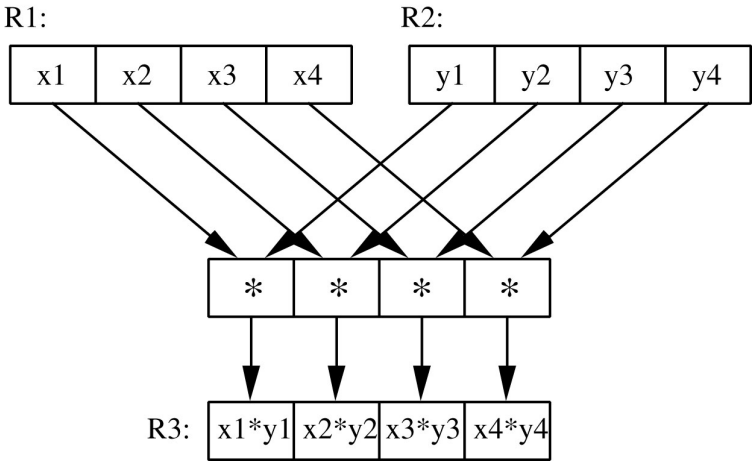

Abbildung 1.8: Grundprinzip einer Multimediaoperation.

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, also 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 Daten-Cache-Speicher oder die Speicherverwaltungseinheit zu nehmen. 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. (Unter einem Prozess versteht man dabei ein in Ausführung befindliches oder ausführbereites Programm mit seinen Daten, also den Konstanten und Variablen mit ihren aktuellen Werten.) 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.

1.2.5 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. Diese sind für die arithmetisch-logischen Befehle die Adressfelder, um Quell- und Zieloperanden zu spezifizieren, und für die Lade-/Speicherbefehle die Quell- und Zieladressangaben. Bei Programmsteuerbefehlen wird der als nächstes auszuführende Befehl adressiert.

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

- Das **Dreiadressformat** *(3-address instruction format)* besteht aus dem Opcode, zwei Quell- (im Folgenden Src1, Scr2) und einem Zieloperandenbezeichner (Dest):

Opcode	Dest	Src1	Scr2

- Das **Zweiadressformat** *(2-address instruction format)* besteht aus dem Opcode, einem Quell- und einem Quell-/Zieloperandenbezeichner, d.h. ein Operandenbezeichner bezeichnet einen Quell- und gleichzeitig den Zieloperanden:

Opcode	Dest/Src1	Src2

- Das **Einadressformat** *(1-address instruction format)* besteht aus dem Opcode und einem Quelloperandenbezeichner:

Opcode	Src

  Dabei wird ein im Prozessor ausgezeichnetes Register, das sogenannte Akkumulatorregister, implizit adressiert. Dieses Register enthält immer einen Quelloperanden und nimmt das Ergebnis auf.

- Das **Nulladressformat** *(0-address instruction format)* besteht nur aus dem Opcode:

Opcode

Voraussetzung für die Verwendung eines Nulladressformats ist eine Stackarchitektur (s.u.).

Befehlssatz-architekturen

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 Register 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 , so muss jeder Operandenbezeichner eine Speicheradresse sein und man kann von einer **Speicher-Speicher-Architektur** sprechen. (Der einzige uns bekannte Prozessor diesen Typs war der TI 9900 der Firma Texas Instruments.) Die ersten Mikroprozessoren besaßen ein sogenanntes **Akkumulatorregister**, das bei arithmetisch-logischen Befehlen immer implizit eine Quelle und das Ziel darstellte, so dass **Einadressbefehle** genügten. Solche Akkumulatorarchitekturen sind gelegentlich noch bei einfachen Mikrocontrollern und Digitalen Signalprozessoren (DSPs) zu finden.

- Man kann sogar mit **Nulladressbefehlen** auskommen: Dies geschieht bei den sogenannten **Stackarchitekturen** oder **Kellerarchitekturen**, welche ihre Operandenregister als Stapel (*stack*) verwalten. 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.

Tabelle 1.2 zeigt, wie ein Zweizeilenprogramm in Pseudo-Assemblerbefehle für die vier Befehlssatz-Architekturen übersetzt werden kann. Die Syntax der Assemblerbefehle schreibt vor, dass nach dem Opcode erst 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.

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 i8086 gab es beliebig 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.

Tabelle 1.2: Programm C=A+B; D=C-B; in den vier Befehlsformatsarten codiert.

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

Die Befehlscodierung kann eine feste oder eine variable Befehlslänge festlegen. Um die Decodierung zu vereinfachen, nutzen RISC-Befehlssätze meist ein Dreiadressformat mit einer festen Befehlslänge von 32 Bit. CISC-Befehlssätze dagegen nutzen Register-Speicher-Befehle und benötigen dafür meist variable Befehlslängen. Um den Speicherbedarf zu minimieren hat man früher bei CISC-Befehlssätzen mit variablen Opcodelängen gearbeitet. Hierzu wurden häufig benutzten Befehlen kurze Opcodes zugeordnet. Variable Befehlslängen finden sich auch in Stackmaschinen wie beispielsweise bei den Java-Prozessoren.

1.2.6 Adressierungsarten

Die **Adressierungsarten** definieren, wie auf die Daten zugegriffen wird. Sie bestimmen die verschiedenen Möglichkeiten, wie eine Operanden- oder eine Sprungzieladresse in dem Prozessor berechnet werden kann. Eine Adressierungsart kann eine im Befehlswort stehende Konstante, ein Register oder einen Speicherplatz im Hauptspeicher spezifizieren.

Zusammenhang: effektive, logische und physikalische Adresse

Wenn ein Speicherplatz im Hauptspeicher bezeichnet wird, so heißt die durch die Adressierungsart spezifizierte Speicheradresse die **effektive Adresse**. Eine effektive Adresse entsteht im Prozessor nach Ausführung der Adressrechnung. In modernen Prozessoren, die eine virtuelle Speicherverwaltung anwenden, wird die effektive Adresse als sogenannte logische Adresse weiteren Speicherverwaltungsoperationen in einer Speicherverwaltungseinheit (*Memory Management Unit – MMU*) unterworfen, um letztendlich eine physikalische Adresse zu erzeugen, mit der dann auf den Hauptspeicher zugegriffen wird. Im Folgenden betrachten wir zunächst nur die Erzeugung einer effektiven Adresse aus den Angaben in einem Maschinenbefehl.

Neben den „expliziten“ Adressierungsarten kann die Operandenadressierung auch „implizit“ bereits in der 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 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 Register- 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 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.

Wenig 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 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.

Bei der **Registeradressierung** (*register*) steht der Operand direkt in einem Register (s. Abbildung 1.9).

Bei der **unmittelbaren Adressierung** (*immediate* oder *literal*) steht der Operand als Konstante direkt im Befehlswort (s. Abbildung 1.10).

Bei der **direkten** oder **absoluten Adressierung** (*direct, absolute*) steht die Adresse eines Speicheroperanden im Befehlswort (s. Abbildung 1.11).

Bei der **registerindirekten Adressierung** (*register indirect* oder *register deferred*) steht die Operandenadresse in einem Register. Das Register dient als Zeiger auf eine Speicheradresse (s. Abbildung 1.12).

Als Spezialfälle der registerindirekten Adressierung können die **registerindirekten Adressierungen mit Autoinkrement/Autodekrement** (*auto-*

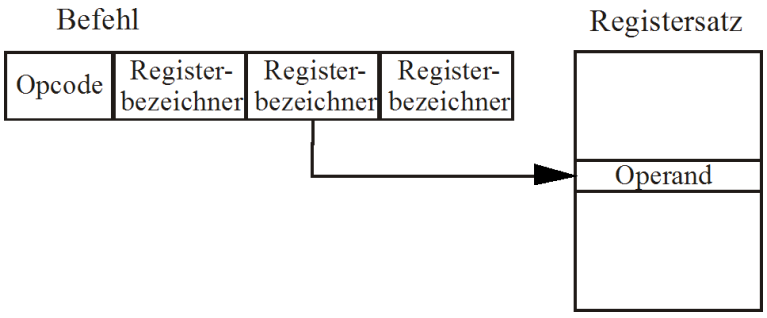

Abbildung 1.9: Registeradressierung.

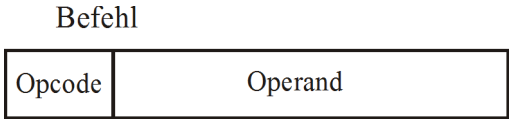

Abbildung 1.10: Unmittelbare Adressierung

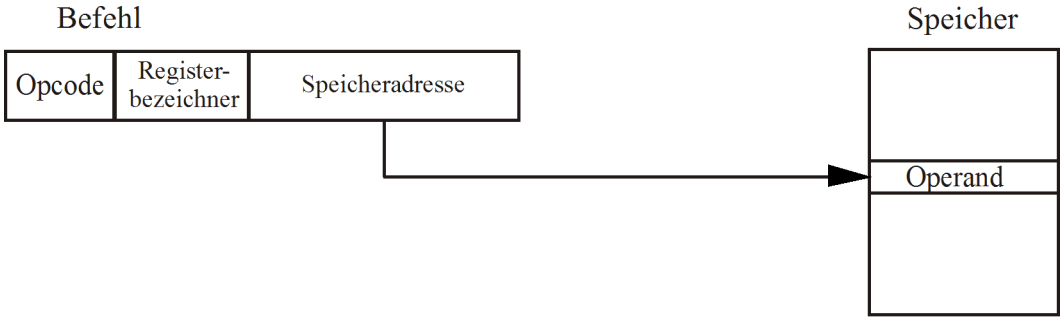

Abbildung 1.11: Direkte Adressierung

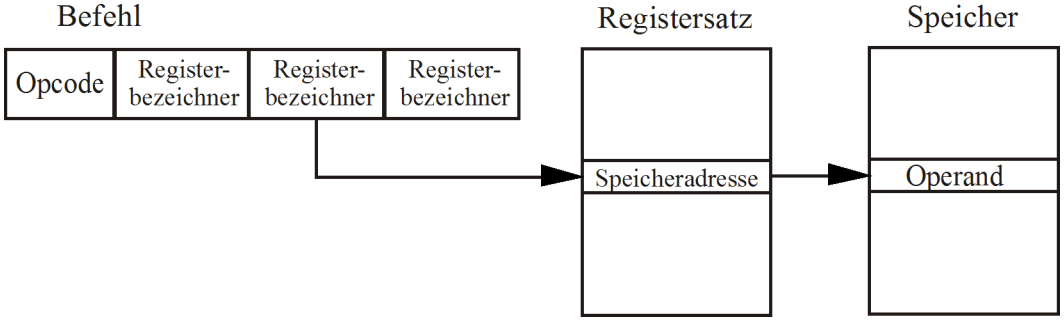

Abbildung 1.12: Registerindirekte Adressierung

increment/autodecrement) betrachtet werden. Diese arbeiten wie die registerindirekte Adressierung, aber inkrementieren bzw. dekrementieren den Registerinhalt. In Abbildung 1.12 ist dies die „Speicheradresse“, die vor oder nach dem Benutzen dieser Adresse um die Länge des adressierten Operanden inkrementiert oder dekrementiert wird. Dementsprechend unterscheidet man die registerindirekte Adressierung mit **Präinkrement**, mit **Postinkrement**, mit **Prädekrement** und mit **Postdekrement.** (Üblich sind Postinkrement und Prädekrement.) Diese Adressierungsarten sind für den Zugriff zu Feldern (*arrays*) in Schleifen nützlich. Der Registerinhalt zeigt auf den Anfang oder das letzte Element eines Feldes und jeder Zugriff erhöht oder erniedrigt den Registerinhalt um die Länge eines Feldelements.

Die **registerindirekte Adressierung mit Verschiebung** (*displacement*, *register indirect with displacement* oder *based*) errechnet die effektive Adresse eines Operanden als Summe eines Registerwerts und des Verschiebewerts (*displacement*), d.h. eines konstanten Werts, der im Befehl steht (s. Abbildung 1.13).

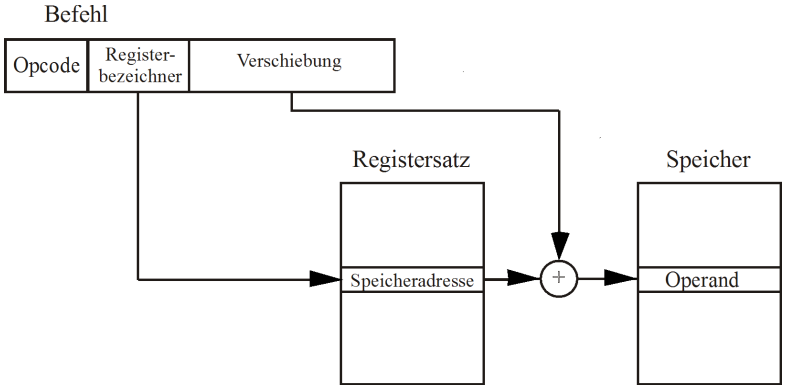

Abbildung 1.13: Registerindirekte Adressierung mit Verschiebung.

Die **indizierte Adressierung** (*indirect indexed*) errechnet die effektive Adresse als Summe eines Registerinhalts und eines weiteren Registers, das 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 (s. Abbildung 1.14).

Die **indizierte Adressierung mit Verschiebung** (*indirect indexed with displacement*) ähnelt der indizierten Adressierung, allerdings wird zur Summe der beiden Registerwerte noch ein im Befehl stehender Verschiebewert (*displacement*) hinzuaddiert (s. Abbildung 1.15).

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 Verschiebewert (*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

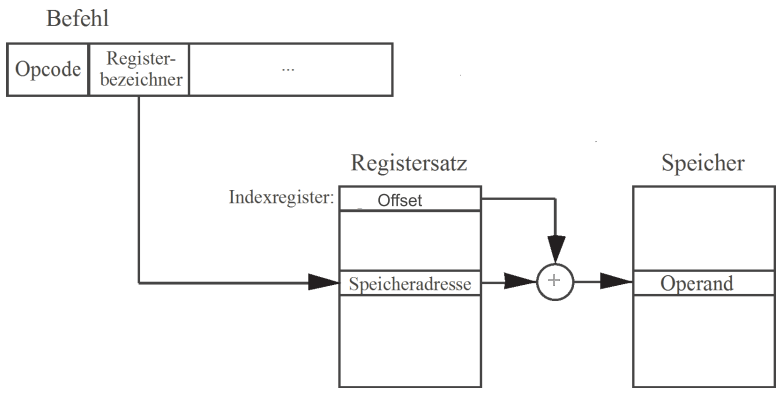

Abbildung 1.14: Indizierte Adressierung.

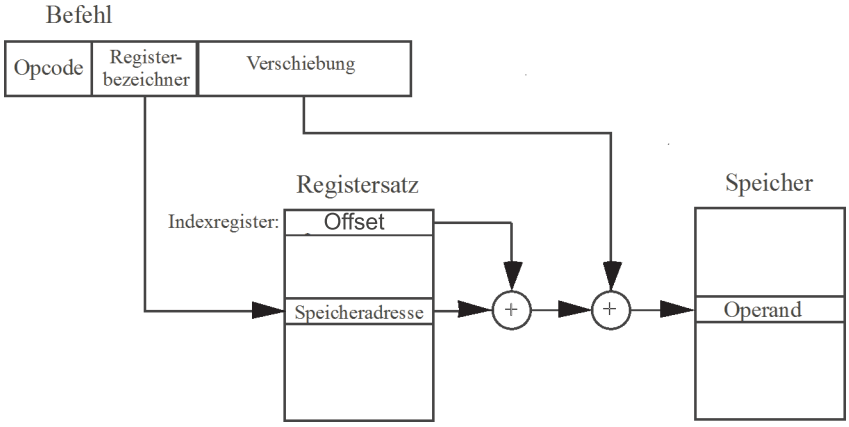

Abbildung 1.15: Indizierte Adressierung mit Verschiebung.

Befehlszählerwertes, so dass nur wenige Bits für den Verschiebewert im Befehl benötigt werden (s. Abbildung 1.16).

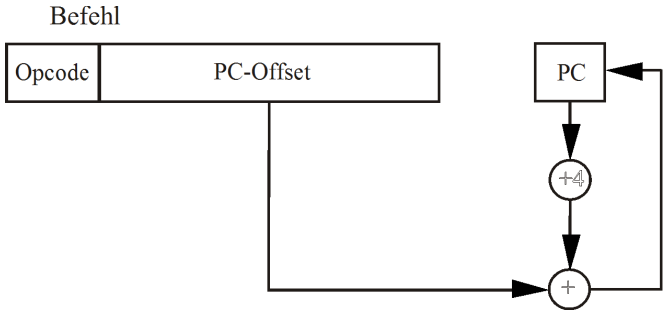

Abbildung 1.16: Befehlszählerrelative Adressierung.

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 (s. Abbildung 1.17).

Tabelle 1.3 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` kön-

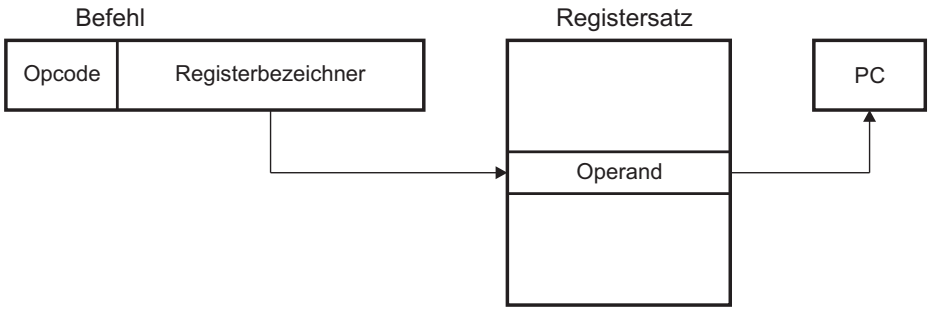

Abbildung 1.17: Befehlszählerindirekte Adressierung.

nen Dezimal-, Hexadezimal-, Oktal- oder Binärzahlen sein; `step` bezeichnet die Feldelementbreite und `inst_step` bezeichnet die Befehlsschrittweite in Abhängigkeit von der Befehlswortbreite, z.B. vier bei Vierbyte-Befehlswörtern.

Tabelle 1.3: 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)

Es gibt darüber hinaus eine ganze Anzahl weiterer Adressierungsarten, die beispielsweise in [1], [2] und [6] beschrieben sind.

Selbsttestaufgabe 1.1 (Registerindirekte Adressierung) *Welche einfacheren Adressierungsarten lassen sich aus der registerindirekten Adressierung mit Verschiebung zusammensetzen?*

Lösung auf Seite 61

1.2.7 CISC- und RISC-Prinzipien

Bei der Entwicklung der Großrechner in den 60er und 70er Jahren hatten technologische Bedingungen wie der teure und langsame Hauptspeicher (es gab noch keine Cache-Speicher) zu einer immer größeren Komplexität der Rechnerarchitekturen geführt. Um den teuren Hauptspeicher optimal zu nutzen, wurden komplexe Maschinenbefehle entworfen, die mehrere Operationen mit einem Opcode codieren. 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. KE4 von 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 werden. Einfache Maschinenbefehle ermöglichten es, bei der Befehlsausführung das Pipelining-Prinzip anzuwenden. Möglichst alle Befehle sollten dabei so implementierbar sein, dass pro Prozessortakt die Ausführung eines Ma-

schinenbefehls 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 eine Mikroprogramm-Steuerwerk implementiert.)

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 begleitende Maßnahmen 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 natürlich 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* Befehls-

ausführung 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 der nächsten Kurseinheit ausführlich vorgestellt.

Im Folgenden werden wir zunächst die skalaren RISC-Prozessoren behandeln.

1.3 Beispiele für RISC-Architekturen

1.3.1 Das Berkeley RISC-Projekt

Das 1980 an der Universität in Berkeley begonnene RISC-Projekt ging von der Beobachtung aus, dass Compiler den umfangreichen Befehlssatz eines CISC-Rechners nicht optimal nutzen können. Daher wurden anhand von compilierten Pascal- und C-Programmen für VAX-, PDP-11- und Motorola-68000-Prozessoren zunächst die Häufigkeiten einzelner Befehle, Adressierungsmodi, lokaler Variablen, Verschachtelungstiefen usw. untersucht. Es zeigten sich zwei wesentliche Ergebnisse:

Ein hoher Aufwand entsteht bei häufigen Prozeduraufrufen oder Kontextwechseln durch Sicherung von Registern und Prozessorstatus und durch Parameterübergabe. Diese geschieht über den Laufzeitstapel im Haupt- bzw. Cache-Speicher und ist dementsprechend langsam. Wesentlich sinnvoller wäre jedoch eine Parameterübergabe in den Registern, was jedoch durch die Gesamtzahl der Register begrenzt ist. Für die Berkeley RISC-Architektur wurde daher eine neue Registerorganisation – die Technik der überlappenden Registerfenster – entwickelt, die sich heute noch in den SPARC-, SuperSPARC- und UltraSPARC-Prozessoren von Sun findet und den Datentransfer zwischen Prozessor und Speicher bei einem Prozeduraufruf minimiert. Der Registersatz mit 78 Registern ist durch seine Fensterstruktur (*windowed register organization*) gekennzeichnet, wobei zu jedem Zeitpunkt jeweils nur eine Teilmenge des Registerblocks sichtbar ist. Als Konsequenz ergibt sich eine effiziente Parameterübergabe für eine kleine Anzahl von Unterprogrammaufrufen. Das Prinzip der überlappenden Registerfenster wird in Abschnitt 3.1.2 ausführlich beschrieben.

RISC-I-Prozessor

Der RISC-I-Prozessor wurde als 32-Bit-Architektur konzipiert; eine Unterstützung von Gleitkommaarithmetik und Betriebssystemfunktionen war nicht vorgesehen. Der Befehlssatz bestand aus nur 31 Befehlen mit einer einheitlichen Länge von 32 Bit und nur zwei Befehlsformaten. Die Befehle lassen sich folgendermaßen einteilen:

- 12 arithmetisch-logische Operationen,
- 9 Lade/Speicheroperationen und

- 10 Programmsteuerbefehle.

Der RISC I verfügt nur über drei Adressierungsarten:

- Registeradressierung `Rx`: Der Operand befindet sich im Register `x`.
- Unmittelbare Adressierung `#num`: Der Operand `#num` ist als 13-Bit-Zahl unmittelbar im Befehlswort angegeben.
- Registerindirekte Adressierung mit Verschiebung `Rx+#displ`: Der Inhalt eines Registers `Rx` wird als Adresse interpretiert, zu der eine 13-Bit-Zahl `#displ` addiert wird. Das Ergebnis der Addition ist die Speicheradresse, in der der Operand zu finden ist.

Durch die Verwendung von `R0+#displ` (Register `R0` enthält immer den Wert Null und kann nicht überschrieben werden) wird die direkte oder absolute Adressierung einer Speicherstelle gebildet. Setzt man für ein Register Ri, i≠0, die Verschiebung `#displ` zu Null, erhält man eine registerindirekte Adressierung.

Im Oktober 1982 war der erste Prototyp des RISC I fertig. Der Prozessor bestand aus etwa 44 500 Transistoren. Auffällig war der stark verminderte Aufwand von nur 6% Chip-Flächenanteil für die Steuerung gegenüber mehr als 50% bei CISC-Architekturen. Der mit einer Taktfrequenz von 1,5 MHz schnellste RISC-I-Chip, der gefertigt wurde, erreichte die Leistung kommerzieller Mikroprozessoren.

Die Entwicklung des Nachfolgemodells RISC II wurde 1983 abgeschlossen. Die Zahl der Transistoren wurde auf 41 000 und die Chip-Fläche sogar um 25% verringert. Die beiden Prozessoren RISC I und II waren natürlich keineswegs ausgereift.

Die wesentliche Weiterentwicklung der RISC-I- und RISC-II-Rechner geschah jedoch durch die Firma Sun, die Mitte der 80er Jahre die SPARC-Architektur definierte, welche mit den SuperSPARC- und heute den UltraSPARC-Prozessoren fortgeführt wird.

1.3.2 Die DLX-Architektur

In diesem Abschnitt wird die Architektur des DLX-Prozessors (DLX steht für „Deluxe“) eingeführt. Dieser ist ein hypothetischer Prozessor, der von Hennessy und Patterson für Lehrzwecke entwickelt wurde und auch im vorliegenden Kapitel als Referenzprozessor dient. Der DLX kann als idealer, einfacher RISC-Prozessor charakterisiert werden, der sehr eng mit dem MIPS-Prozessor verwandt ist.

Registersatz DLX

Die Architektur enthält 32 jeweils 32 Bit breite Universalregister (*General-Purpose Registers*, *GPRs*) `R0, ..., R31`, wobei der Wert von Register `R0` immer Null ist. Dazu gibt es einen Satz von Gleitkommaregistern (*Floating-Point Registers FPRs*), die als 32 Register einfacher (32 Bit breiter) Genauigkeit (`F0, F1, ..., F31`) oder als 16 Register doppelter (64 Bit breiter) Genauigkeit (`F0, F2, ..., F30`) nach IEEE 754-Format genutzt werden können.

(Dabei werden jeweils zwei 32-bit-Register R(2i), R(2i+1) zu einem 64-bit-Register verbunden.) Es gibt einen Satz von Spezialregistern für den Zugriff auf Statusinformationen. Das Gleitkomma-Statusregister wird für Vergleiche und zum Anzeigen von Ausnahmesituationen verwendet. Alle Datentransporte zum bzw. vom Gleitkomma-Statusregister erfolgen über die Universalregister.

Speicherorganisation des DLX

Der Speicher ist byteadressierbar im Big-endian-Format mit 32-Bit-Adressen. Man beachte, dass daher der Befehlszähler immer automatisch nach jedem Befehl um 4 erhöht wird, auch im Falle der Verzweigungsbefehle. Alle Speicherzugriffe müssen ausgerichtet sein und erfolgen mittels der Lade-/Speicherbefehle zwischen Speicher und Universalregistern oder Speicher und Gleitkommaregistern. Weitere Transportbefehle erlauben es, Daten zwischen Universalregistern und Gleitkommaregistern zu verschieben. Der Zugriff auf die Universalregister kann byte-, halbwort- (16 Bit) oder wortweise (32 Bit) erfolgen. Auf die Gleitkommaregister kann mit einfacher oder doppelter Genauigkeit zugegriffen werden.

Befehlsformate des DLX

Alle Befehle sind 32 Bit lang und müssen im Speicher ausgerichtet sein. Dabei ist das Opcode-Feld 6 Bit breit; die Quell- `(rs1, rs2)` und Zielregisterangaben (`rd`) benötigen 5 Bit, um die 32 Register eines Registersatzes zu adressieren. Für Verschiebewerte (*displacement*) und unmittelbare Konstanten (*immediate*) sind 16-Bit-Felder vorgesehen. Befehlszählerrelative Verzweigungsadressen (PC-offset) können 26 Bits lang sein. Ein Befehl ist in einem der folgenden drei Befehlsformate codiert:

- I-Typ: zum Laden und Speichern von Bytes, Halbwörtern und Wörtern, für alle Operationen mit unmittelbaren Operanden, bedingte Verzweigungsbefehle sowie unbedingte Sprungbefehle mit Zieladresse in einem Universalregister (JR, JALR).

6 Bits	5 Bits	5 Bits	16 Bits
Opcode	rs1	rd	immediate

- R-Typ: für Register-Register-ALU-Operationen, wobei das 11-Bit-func-Feld die Operation codiert, und für die Transportbefehle.

6 Bits	5 Bits	5 Bits	5 Bits	11 Bits
Opcode	rs1	rs2	rd	func

- J-Typ: für Jump- (J), Jump-and-Link- (JAL), Trap- und RFE-Befehle.

6 Bits	26 Bits
Opcode	PC-offset

Es gibt vier Klassen von Befehlen: Transportbefehle, arithmetisch-logische Befehle, Verzweigungen und Gleitkommabefehle.

Transportbefehle des DLX

Für die Transportbefehle, also die Lade-/Speicherbefehle (s. Tabelle 1.4) gibt es nur die Adressierungsart „registerindirekt mit Verschiebung“ in der Form „Universalregister + 16-Bit-Verschiebewert mit Vorzeichen“, doch können daraus die registerindirekten Adressierungsarten (Verschiebewert = 0) und die absoluten Adressierungsarten (Basisregister = R0) erzeugt werden.

Tabelle 1.4: Transportbefehle.

Opcode	Bedeutung
LB, LBU	Laden eines Bytes, Laden eines Bytes vorzeichenlos (U – unsigned),
SB	Speichern eines Bytes (B)
LH, LHU	Laden eines Halbwortes (H), Laden eines Halbwortes vorzeichenlos,
SH	Speichern eines Halbwortes
LW, SW	Laden eines Wortes (W), Speichern eines Wortes (von/zu Universalregistern)
LF, LD	Laden eines einfach (F) bzw. doppelt (D) genauen Gleitkommaregisters,
SF, SD	Speichern eines einfach bzw. doppelt genauen Gleitkommaregisters
MOVI2S, MOVS2I	Transport von einem Universalregister (I) zu einem Spezialregister (S) bzw. umgekehrt
MOVF, MOVD	Transport von einem Gleitkommaregister (F) bzw. Gleitkommaregisterpaar (D) zu einem anderen Register bzw. Registerpaar
MOVFP2I, MOVI2fp	Transport von einem Gleitkommaregister (FP) zu einem Universalregister (I) bzw. umgekehrt

Arithmetisch-logische Befehle des DLX

Alle arithmetisch-logischen Befehle (s. Tabelle 1.5) sind Dreiadressbefehle mit zwei Quellregistern (oder einem Quellregister und einem vorzeichenerweiterten unmittelbaren Operanden) und einem Zielregister. Nur die einfachen arithmetischen und logischen Operationen Addition, Subtraktion, AND, OR, XOR und Schiebeoperationen sind vorhanden. Vergleichsbefehle vergleichen zwei Register auf Gleichheit, Ungleichheit, größer, größer-gleich, kleiner oder kleiner-gleich. Wenn die Bedingung erfüllt ist, wird das Zielregister auf Eins gesetzt, andernfalls auf Null.

Tabelle 1.5: Arithmetisch-logische Befehle.

Opcode	Bedeutung
ADD,ADDI, ADDU,ADDUI	Addiere, Addiere mit unmittelbarem (16-Bit-)Operanden (I) mit oder ohne Vorzeichen (U – *unsigned*)
SUB,SUBI, SUBU,SUBUI	Subtrahiere, Subtrahiere mit unmittelbarem (16-Bit-)Operanden mit oder ohne Vorzeichen
MULT,MULTU, DIV,DIVU	Multiplikation und Division mit und ohne Vorzeichen; alle Operanden sind Gleitkommaregister-Operanden und nehmen bzw. ergeben 32-Bit-Werte
AND,ANDI	Logisches UND, logisches UND mit unmittelbaren Operanden
OR,ORI	Logisches ODER, logisches ODER mit unmittelbaren Operanden;
XOR,XORI	Exklusiv-ODER, Exklusiv-ODER mit unmittelbaren Operanden

LHI	*Load High Immediate*-Operation: Laden der oberen Registerhälfte mit einem unmittelbaren Operanden, niedrige Hälfte wird auf Null gesetzt
SLL,SRL SRA,SRAI SLLI,SRLI	Schiebeoperationen: links-logisch (LL), rechts-logisch (RL), rechts-arithmetisch (RA) ohne und mit (I) unmittelbaren Operanden, links-logisch und rechts-logisch mit unmittelbaren Operanden
S_, S_I	*Set conditional*-Vergleichsoperation; _ kann sein: LT (less than), GT (greater than), LE (less equal), GE (greater equal), EQ (equal), NE (not equal)

Verzweigungsbefehle des DLX

Die ersten vier Verzweigungsbefehle (s. Tabelle 1.6) sind bedingt. Die Verzweigungsbedingung ist durch den Opcode des Befehls spezifiziert, wobei das Quellregister auf Null oder auf von Null verschieden getestet wird. Die Zieladresse der bedingten Sprungbefehle ist durch eine vorzeichenerweiterte 16-Bit-Verschiebung gegeben, die zum Inhalt des inkrementierten Befehlszählerregisters (PC + 4) addiert wird. Die Jump-and-Link-Befehle (JAL, JALR) speichern die Adresse des Befehls, der dem Sprungbefehl folgt, im Register R31. Dadurch wird ein Rücksprung an die „Unterbrechungsstelle“ im Programm erleichtert – ähnlich wie beim Aufruf eines Unterprogramms und Rücksprung in das Hauptprogramm.

Tabelle 1.6: Verzweigungsbefehle.

Opcode	Bedeutung
BEQZ, BNEZ	Verzweigen, falls das angegebene Universalregister gleich (EQZ) bzw. ungleich (NEZ) Null ist; befehlszählerrelative Adressierung (16-Bit-Verschiebung + Befehlszähler + 4)
BFPT, BFPF	Testen des Vergleichsbits im Gleitkomma-Statusregister und Verzweigen, falls Wert gleich „*true*“ (T) oder „*false*“ (F); befehlszählerrelative Adressierung (16-Bit-Verschiebung + Befehlszähler + 4)
J, JR	Unbedingter Sprung mit befehlszählerrelativer Adressierung (26-Bit-Verschiebung + Befehlszähler + 4) oder mit befehlszählerindirekter Adressierung (Zieladresse in einem Universalregister)
JAL, JALR	Speichert PC+4 in R31, Ziel ist befehlszählerrelativ adressiert (JAL) oder ein Register (JALR)
TRAP	Ausnahmebefehl: Übergang zum Betriebssystem bei einer vektorisierten Adresse
RFE	Rückkehr zum Benutzerprogramm von einer Unterbrechung; Wiederherstellen des Benutzermodus

Gleitkommabefehle des DLX

Alle Gleitkommabefehle (s. Tabelle 1.7) sind Dreiadressbefehle mit zwei Quell- und einem Zielregister. Bei jedem Befehl wird angegeben, ob es sich um einfach oder doppelt genaue Gleitkommaoperationen handelt. Die letzte-

ren können nur auf ein Registerpaar `(F0,F1), ..., (F30,F31)` angewendet werden.

Tabelle 1.7: Gleitkommabefehle.

Opcode	Bedeutung
ADDF, ADDD	Addition von einfach (F) bzw. doppelt (D) genauen Gleitkommazahlen
SUBF, SUBD	Subtraktion von einfach bzw. doppelt genauen Gleitkommazahlen
MULTF, MULTD	Multiplikation von einfach bzw. doppelt genauen Gleitkommazahlen
DIVF, DIVD	Division von einfach bzw. doppelt genauen Gleitkommazahlen
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Konvertierungsbefehle: CVTx2y konvertiert von Typ x nach Typ y, wobei x und y einer der Grundtypen I (Integer), F (einfach genaue Gleitkommazahl) oder D (doppelt genaue Gleitkommazahl) ist.
_F, _D	Vergleichsoperation auf einfach genauen (F) bzw. doppelt genauen (D) Gleitkommazahlen: _ kann sein: LT (less than), GT (greater than), LE (less equal), GE (greater equal), EQ (equal), NE (not equal); Es wird jeweils ein Vergleichsbit im Gleitkomma-Statusregister gesetzt.

Selbsttestaufgabe 1.2 (DLX-Simulator)
Arbeiten Sie sich in den DLX-Simulator ein. Schreiben Sie ein einfaches Programm, welches das Quadrat einer Zahl berechnet, die sich im Register R5 befindet. Das Ergebnis soll im Register R6 abgespeichert werden. Schauen Sie sich dabei die Registerbelegungen im DLX-Simulator genau an.

Lösung auf Seite 61

Selbsttestaufgabe 1.3 (Eingabeaufforderung)
Über die Eingabeaufforderung sollen zunächst zwei Zahlen eingelesen werden. Schreiben Sie ein Programm:

a.) welches die Summe dieser zwei Integerzahlen berechnet und das Ergebnis ausgibt;

b.) welches die zweite Zahl von der ersten abzieht und das Ergebnis ausgibt;

c.) welches das Produkt der beiden Zahlen berechnet, wobei das Ergebnis in das Gleitkommazahlen-Format umgewandelt werden soll;

d.) welches die erste Zahl durch die zweite teilt, wobei die Zahlen vorher in Gleitkommazahlen umformatiert werden sollen.

e.) Schreiben Sie ein Programm, welches zu jeder Berechnung aus a) bis d) ein Unterprogramm besitzt, das über das Hauptprogramm aufgerufen wird.

Lösung auf Seite 61

Selbsttestaufgabe 1.4 (Programm-Eingabeaufforderung)
Die Fakultät einer Zahl ist definiert als:

$$a! = a \cdot (a-1) \cdot (a-2) \cdots 2 \cdot 1$$

Schreiben Sie ein Programm, welches eine Zahl über die Eingabeaufforderung einliest und die Fakultät dieser Zahl ausgibt.

Lösung auf Seite 65

1.4 Einfache Prozessoren und Prozessorkerne

1.4.1 Grundlegender Aufbau eines Mikroprozessors

Ein **Mikroprozessor** ist ein Prozessor, der auf einem, manchmal auch auf mehreren VLSI-Chips implementiert ist. Heutige Mikroprozessoren haben meist auch Cache-Speicher und verschiedene Steuerfunktionen auf dem Prozessor-Chip untergebracht.

Mikroprozessor

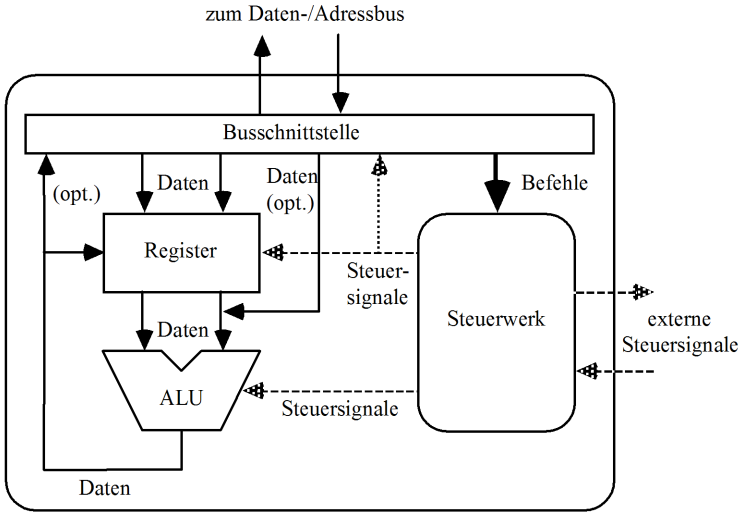

Abbildung 1.18: Struktur eines einfachen Mikroprozessors.

Abbildung 1.18 zeigt einen Mikroprozessor einfachster Bauart, wie er heute noch als Kern von einfachen Mikrocontrollern (Prozessoren zur Steuerung technischer Geräte) vorkommt. Er enthält ein Rechenwerk (oder arithmetisch-logische Einheit, *Arithmetic Logic Unit* – ALU), das seine Daten aus internen Speicherplätzen, den Registern, oder über die Busschnittstelle direkt aus dem Hauptspeicher empfängt. Das Rechenwerk führt auf den Eingabedaten eine arithmetisch-logische Operation aus. Der Resultatwert wird wieder in einem Register abgelegt oder über die Busschnittstelle in den Hauptspeicher transportiert. Die Rechenwerksoperationen werden durch Steuersignale vom Steuerwerk

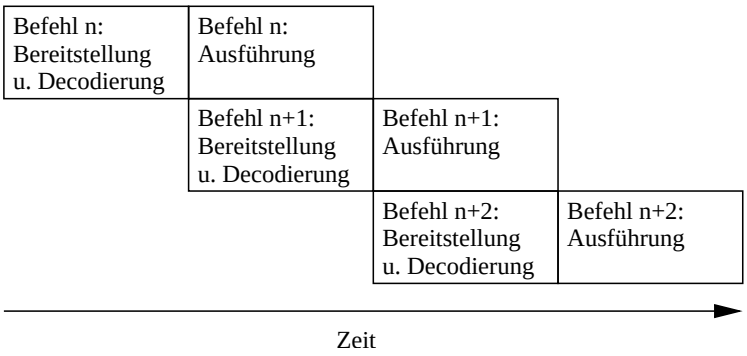

Abbildung 1.19: Überlappende Befehlsausführung.

bestimmt. Das Steuerwerk erhält seine Befehle, adressiert durch das interne Befehlszählerregister, ebenfalls über die Busschnittstelle aus dem Hauptspeicher. Es setzt die Befehle in Abhängigkeit vom Prozessorzustand, der in einem internen Prozessor-Statusregister gespeichert ist, und eventuell auch abhängig von externen Steuersignalen in Steuersignale für das Rechenwerk, die Registerauswahl und die Busschnittstelle um.

1.4.2 Einfache Implementierungen

Ein Rechner mit von-Neumann-Prinzip benötigt zur Bearbeitung eines Befehls zwei grundlegende Phasen: eine Befehlsbereitstellungs- und eine Ausführungsphase. In der Ausführungsphase werden die Operanden bereitgestellt, in der arithmetisch-logischen Einheit (ALU) verarbeitet und schließlich wird das Resultat gespeichert.

Vom Prinzip her muss jede Befehlsbearbeitung beendet sein, bevor die nächste Befehlsbearbeitung beginnen kann. In der einfachsten Implementierung ist jede der beide Phasen abwechselnd aktiv – die beiden Phasen werden sequentiell zueinander ausgeführt. Jede dieser Phasen kann je nach Implementierung und Komplexität des Befehls wiederum eine oder mehrere Takte in Anspruch nehmen. Für die Durchführung eines Programms mit n Befehlen von je k Takten werden $n{\cdot}k$ Takte benötigt.

Überlappende Befehlsverarbeitung

Jedoch können die beiden Phasen auch überlappend zueinander ausgeführt werden (s. Abbildung 1.19). Während der eine Befehl sich in seiner Ausführungsphase befindet, wird bereits der nach Ausführungsreihenfolge nächste Befehl aus dem Speicher geholt und decodiert. Diese Art der Befehlsverarbeitung ist erheblich komplizierter als diejenige ohne Überlappung, denn nun müssen Ressourcenkonflikte bedacht und gelöst werden.

Ressourcenkonflikt

Ein Ressourcenkonflikt tritt ein, wenn beide Phasen gleichzeitig dieselbe Ressource benötigen. Zum Beispiel muss in der Befehlsbereitstellungsphase über die Verbindungseinrichtung auf den Speicher zugegriffen werden, um den Befehl zu holen. Gleichzeitig kann jedoch auch in der Ausführungsphase ein Speicherzugriff zum Holen eines Operanden notwendig sein. Falls Daten und Befehle im gleichen Speicher stehen (von-Neumann-Architektur im Gegensatz zur Harvard-Architektur) und zu einem Zeitpunkt nur ein Speicherzugriff erfolgen kann, so muss dieser Zugriffskonflikt hardwaremäßig erkannt und gelöst

werden.

Ein weiteres Problem entsteht bei Programmsteuerbefehlen, die den Programmfluss ändern. In der Ausführungsphase eines Sprungbefehls, in der die Sprungzieladresse berechnet werden muss, wird bereits der im Speicher an der nachfolgenden Adresse stehende Befehl geholt. Dies ist im Falle eines unbedingten Sprungs jedoch nicht der richtige Befehl. Dieser Fall muss von der Hardware erkannt werden und entweder das Holen des nachfolgenden Befehls zurückgestellt oder der bereits geholte Befehl wieder gelöscht werden.

Idealerweise sollten beide Phasen etwa gleich viele Takte benötigen und keinerlei Konflikte hervorrufen. Dann kann unter Vernachlässigung der Start- und der Endphase der Verarbeitung die Verarbeitungsgeschwindigkeit gegenüber der Implementierung ohne Überlappung verdoppelt werden.

1.4.3 Pipeline-Prinzip

Die konsequente Fortführung der im letzten Abschnitt beschriebenen überlappenden Verarbeitung ist bei heutigen Mikroprozessoren das **Pipelining** – die „Fließband-Bearbeitung‘, die die Ausführungsgeschwindigkeit von Befehlen in ähnlicher Weise beschleunigt, wie z.B. die Herstellung von Produkten des täglichen Lebens.

Pipelining

Unter dem Begriff **Pipelining** versteht man die Zerlegung eines Verarbeitungsauftrages (im Folgenden eine Maschinenoperation) in mehrere Teilverarbeitungsschritte, die dann von hintereinander geschalteten Verarbeitungseinheiten taktsynchron bearbeitet werden, wobei jede Verarbeitungseinheit genau einen speziellen Teilverarbeitungsschritt ausführt. Die Gesamtheit dieser Verarbeitungseinheiten nennt man eine **Pipeline**. Pipelining ist eine Implementierungstechnik, bei der mehrere Befehle überlappt ausgeführt werden. Jede Stufe der Pipeline heißt **Pipeline-Stufe** oder **Pipeline-Segment**. Die einzelnen Pipeline-Stufen sind aus Schaltnetzen (kombinatorischen Schaltkreisen) aufgebaut, die die Funktion der Stufe realisieren.

Wichtige Merkregel: Ein *Pipeline-Register latch*) sollte nicht mit den *Registern des Registersatzes* des Prozessors verwechselt werden.

Die Pipeline-Stufen werden durch getaktete **Pipeline-Register** (auch *latches* genannt) getrennt, welche die jeweiligen Zwischenergebnisse aufnehmen (s. Abbildung 1.20). Dabei werden alle Register vom selben Taktsignal gesteuert, sie arbeiten also synchron. Ein Pipeline-Register sollte nicht mit den Registern des Registersatzes des Prozessors verwechselt werden. Pipeline-Register sind Pipeline-interne Pufferspeicher, die das Schaltnetz, durch das eine Pipeline-Stufe realisiert wird, von der nächsten Pipeline-Stufe trennen. Erst wenn alle Signale die Gatter einer Pipeline-Stufe durchlaufen haben und sich in den Pipeline-Registern ein stabiler Zustand eingestellt hat, kann der nächste Takt der Pipeline erfolgen.

Die Schaltnetze in den Pipeline-Stufen S_j besitzen u.U. unterschiedliche Verzögerungszeiten und die Pipeline-Register eine feste Verzögerungszeit. Die benötigte **Länge eines Taktzyklus** (**Taktperiode**) eines Pipeline-Prozessors wird bestimmt durch die Summe aus der Verzögerungszeit der Pipeline-Register und der Verzögerungszeit der langsamsten Pipeline-Stufe.

Ein **Pipeline-Maschinentakt** ist die Zeit, die benötigt wird, um einen Befehl eine Stufe weiter durch die Pipeline zu schieben.

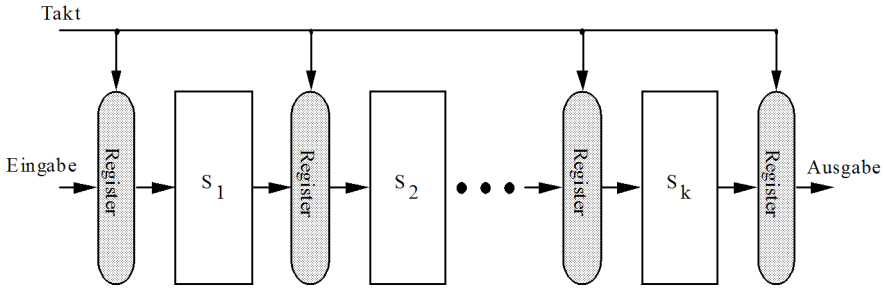

Abbildung 1.20: Pipeline-Stufen und Pipeline-Register.

k-stufigen Pipeline

Sobald die Pipeline „aufgefüllt“ ist, kann unabhängig von der Stufenzahl des Pipeline-Prozessors ein Ergebnis pro Taktzyklus berechnet werden. Idealerweise wird ein Befehl in einer ***k*-stufigen Pipeline** in k Takten verarbeitet. Wird in jedem Takt ein neuer Befehl geladen, dann werden zu jedem Zeitpunkt unter idealen Bedingungen k Befehle gleichzeitig behandelt und jeder Befehl benötigt k Takte bis zum Verlassen der Pipeline.

Definition: Latenz und Durchsatz

Man definiert die **Latenz** als die Zeit, die ein Befehl benötigt, um alle k Pipeline-Stufen zu durchlaufen.

Der **Durchsatz** einer Pipeline wird definiert als die Anzahl der Befehle, die eine Pipeline pro Takt verlassen können. Dieser Wert spiegelt die Rechenleistung einer Pipeline wider. Im Gegensatz zu $n \cdot k$ Takten eines hypothetischen Prozessors ohne Pipeline dauert die Ausführung von n Befehlen in einer k-stufigen Pipeline $k+n-1$ Takte (unter der Annahme idealer Bedingungen mit einer Latenz von k Takten und einem Durchsatz von 1). Dabei werden k Taktzyklen benötigt, um die Pipeline aufzufüllen bzw. den ersten Verarbeitungsauftrag auszuführen, und n-1 Taktzyklen, um die restlichen n - 1 Verarbeitungsaufträge (Befehlsbearbeitungen) durchzuführen.

Definition: Beschleunigung

Daraus ergibt sich eine **Beschleunigung** (*Speedup* S) von

$$S = \frac{n \cdot k}{k+n-1} = \frac{k}{k/n+1-1/n}$$

Ist die Anzahl der in die Pipeline gegebenen Befehle sehr groß, so ist die Beschleunigung näherungsweise gleich der Anzahl k der Pipeline-Stufen.

Durch die mittels der überlappten Verarbeitung gewonnene Parallelität kann die Verarbeitungsgeschwindigkeit eines Prozessors beschleunigt werden, wenn die Anzahl der Pipeline-Stufen erhöht wird. Außerdem wird durch eine höhere Stufenzahl jede einzelne Pipeline-Stufe weniger komplex, so dass die Gattertiefe des Schaltnetzes, welches die Pipeline-Stufe implementiert, geringer wird und die Signale früher an den Pipeline-Registern ankommen. Vom Prinzip her kann deshalb eine lange Pipeline schneller getaktet werden als eine kurze. Dem steht jedoch die erheblich komplexere Verwaltung gegenüber, die durch die zahlreicher auftretenden Pipeline-Konflikte benötigt wird.

Definition: Befehls-Pipeline

Bei einer **Befehls-Pipeline** (*instruction pipeline*) wird die Bearbeitung eines Maschinenbefehls in verschiedene Phasen unterteilt. Aufeinanderfolgende Maschinenbefehle werden jeweils um einen Takt versetzt im Pipelining-Verfahren verarbeitet. Befehls-Pipelining ist eines der wichtigsten Merkmale

moderner Prozessoren. Bei einfachen RISC-Prozessoren bestand das Entwurfsziel darin, einen durchschnittlichen CPI-Wert (cycles per instruction, Taktzyklen pro Befehl) zu erhalten, der möglichst nahe bei 1 liegt. Heutige Hochleistungsprozessoren kombinieren das Befehls-Pipelining mit weiteren Mikroarchitekturtechniken wie der Superskalartechnik, der VLIW- und der EPIC-Technik, um bis zu sechs Befehle pro Takt ausführen zu können.

1.5 Befehls-Pipelining

1.5.1 Grundlegende Stufen einer Befehls-Pipeline

Wie teilt sich die Verarbeitung eines Befehls in Phasen auf? Als Erweiterung des zweistufigen Konzepts aus Abschnitt 1.3.3 kann man eine Befehlsverarbeitung folgendermaßen feiner unterteilen:

- Befehl bereitstellen,
- Befehl decodieren,
- Operanden (in den Pipeline-Registern) vor der ALU bereitstellen,
- Operation auf der ALU ausführen und
- das Resultat zurückschreiben.

Randbedingungen für den Entwurf einer Befehls-Pipeline

Da alle diese Phasen als Pipeline-Stufen etwa gleich lang dauern sollten, gelten die folgenden Randbedingungen:

- Die Befehlsbereitstellung sollte möglichst immer in einem Takt erfolgen. Das kann nur unter der Annahme eines Code-Cache-Speichers auf dem Prozessor-Chip geschehen. Falls der Befehl aus dem Speicher geholt werden muss, wird die Pipeline-Verarbeitung für einige Takte unterbrochen.
- Vorteilhaft für die Befehlsdecodierung ist ein einheitliches Befehlsformat und eine geringe Komplexität der Befehle. Das sind Eigenschaften, die für die RISC-Architekturen als Ziele formuliert wurden. Falls die Befehle unterschiedlich lang sind, muss erst die Länge des Befehls durch einen aufwändigen Decodierschritt erkannt werden.
- Vorteilhaft für die Operandenbereitstellung ist eine Register-Register-Architektur (ebenfalls eine RISC-Eigenschaft), d.h. die Operanden der arithmetisch-logischen Befehle müssen nur aus den Registern des Registersatzes in die Pipeline-Register übertragen werden. Falls Speicheroperanden ebenfalls zugelassen sind, wie es für CISC-Architekturen der Fall ist, so dauert das Laden der Operanden eventuell mehrere Takte und der Pipeline-Fluss muss unterbrochen werden.

- Die Befehlsdecodierung ist für Befehlsformate einheitlicher Länge und Befehle mit geringer Komplexität so einfach, dass sie mit der Operandenbereitstellung aus den Registern eines Registersatzes zu einer Stufe zusammengefasst werden kann. In der ersten Takthälfte wird decodiert und die (Universal-)Register der Operanden werden angesprochen. In der zweiten Takthälfte werden die Operanden aus den (Universal-)Registern in die Pipeline-Register übertragen.

- Die Ausführungsphase kann für einfache arithmetisch-logische Operationen in einem Takt durchlaufen werden. Komplexere Operationen wie die Division oder die Gleitkommaoperationen benötigen mehrere Takte, was die Organisation der Pipeline erschwert.

- Bei Lade- und Speicheroperationen muss erst die effektive Adresse berechnet werden, bevor auf den Speicher zugegriffen werden kann. Der Speicherzugriff ist wiederum besonders schnell, wenn das Speicherwort aus dem Daten-Cache-Speicher gelesen oder dorthin geschrieben werden kann. Im Falle eines Daten-Cache-Fehlzugriffs oder bei Prozessoren ohne Daten-Cache dauert der Speicherzugriff mehrere Takte, in denen die anderen Pipeline-Stufen leer laufen. In der nachfolgend betrachteten DLX-Pipeline wird deshalb nach der Ausführungsphase, in der die effektive Adresse berechnet wird, eine zusätzliche Speicherzugriffsphase eingeführt, in der der Zugriff auf den Daten-Cache-Speicher durchgeführt wird.

- Das Rückschreiben des Resultatwerts in ein Register des Registersatzes kann in einem Takt oder sogar einem Halbtakt der Pipeline geschehen. Das Rückschreiben in den Speicher nach einer arithmetisch-logischen Operation, wie es durch Speicheradressierungen von CISC-Prozessoren möglich ist, dauert länger und erschwert die Pipeline-Organisation.

1.5.2 Die DLX-Pipeline

Überblick über die Stufen der DLX-Pipeline

Als Beispiel einer Befehlsausführung mit Pipelining betrachten wir eine einfache Befehls-Pipeline mit den in Abbildung 1.21 dargestellten Stufen. Eine überlappte Ausführung dieser fünf Stufen führt zu einer fünfstufigen Pipeline mit den folgenden Phasen der Befehlsausführung:

- **Befehlsbereitstellungs**- oder **IF-Phase** (*Instruction Fetch*): Der Befehl, der durch den Befehlszähler adressiert ist, wird aus dem I-Cache-Speicher (*Instruction Cache*) in einen Befehlspuffer geladen. Der Befehlszähler wird weitergeschaltet.

- **Decodier**- und Operandenbereitstellungsphase oder **ID-Phase** (*Instruction Decode/Register Fetch*): Aus dem Operationscode des Maschinenbefehls werden durch ein Decodierschaltnetz Pipeline-interne Steuersignale erzeugt. Die Operanden werden aus Registern (*Register File*) bereit gestellt.

- **Ausführungs**- oder **EX-Phase** (*Execute/Address Calculation*): Die Operation wird von der ALU auf den Operanden ausgeführt. Bei Lade-/Speicherbefehlen berechnet die ALU die effektive Adresse.

- **Speicherzugriffs**- oder **MEM-Phase** (*Memory Access*): Der Speicherzugriff auf den D-Cache (*Data Cache*) wird durchgeführt.

- **Resultatspeicher**- oder **WB-Phase** (*Write Back*): Das Ergebnis wird in ein Register geschrieben.

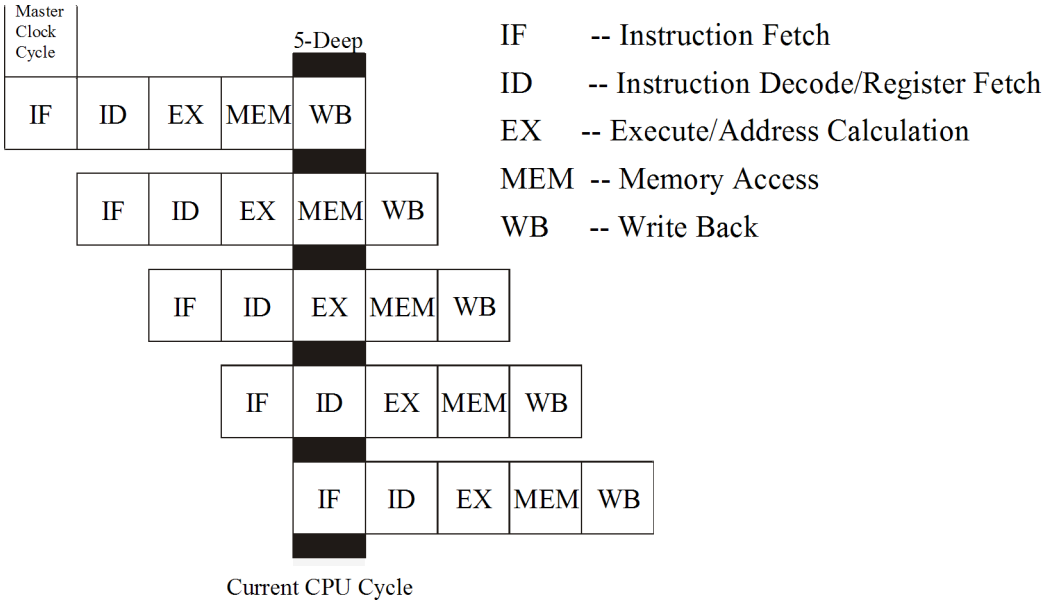

Abbildung 1.21: Grundlegendes Pipelining.

In der ersten Phase wird ein Befehl von einer Befehlsbereitstellungseinheit geladen. Wenn dieser Vorgang beendet ist, wird der Befehl zur Decodiereinheit weitergereicht. Während die zweite Einheit mit ihrer Aufgabe beschäftigt ist, wird von der ersten Einheit bereits der nächste Befehl geladen.

Im Idealfall bearbeitet diese fünfstufige Pipeline fünf aufeinander folgende Befehle gleichzeitig, jedoch befinden sich die Befehle jeweils in einer unterschiedlichen Phase der Befehlsausführung. Die RISC-Architektur des DLX-Prozessors führt die meisten Befehle in einem Takt aus. Ausnahmen sind die Gleitkommabefehle. Somit beendet im Idealfall jede Pipeline-Stufe ihre Ausführung innerhalb eines Takts und es wird auch nach jedem Taktzyklus ein Resultat erzeugt.

Diese für den DLX-Prozessor vorgesehene Pipeline wurde z.B. auch im MIPS R3000-Prozessor implementiert. Heutzutage gibt es ähnlich einfache Pipelines noch in den Kernen von Digitalen Signalprozessoren (DSPs) und einigen Multimediaprozessoren.

Abbildung 1.22 zeigt detailliert die grundsätzlichen Stufen der Befehls-Pipeline, die auf eine gegenüber Abschnitt 1.3.2 vereinfachte DLX-Architektur (im Wesentlichen ohne die Gleitkommabefehle) angepasst ist.

Die Pipeline-Stufen werden jeweils von mehreren **Pipeline-Registern** ge- Die Funktionen der Pipeline-Register

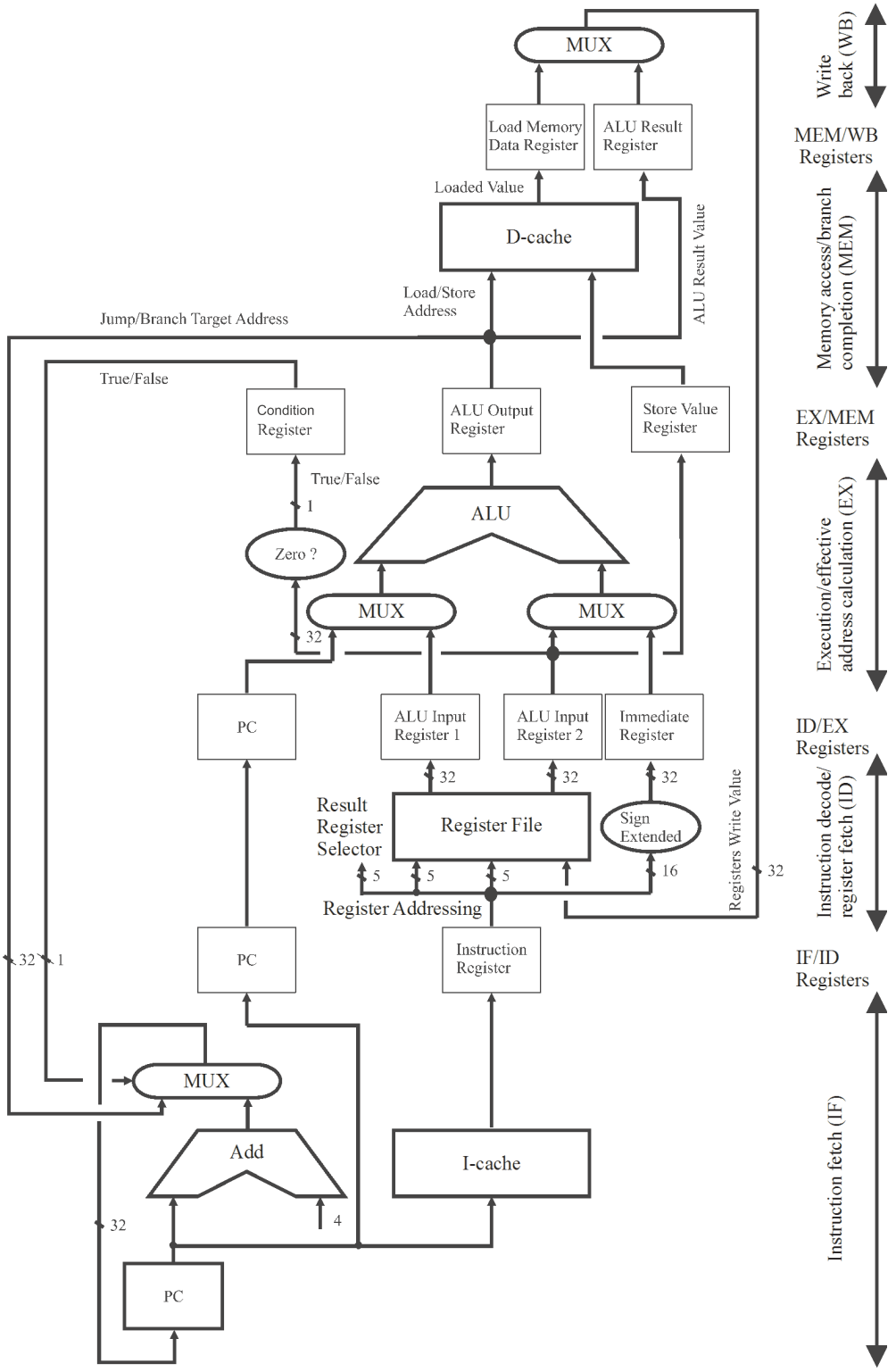

Abbildung 1.22: Implementierung einer DLX-Pipeline (ohne Gleitkommabefehle).

trennt, die funktional verschiedene Zwischenwerte puffern. Nicht alle notwendigen Pipeline-Register sind in Abbildung 1.22 aufgeführt, um die Abbildung nicht noch komplexer werden zu lassen. Die in der Abbildung gezeigten, meist 32 Bit breiten Pipeline-Register besitzen die folgenden Funktionen:

- In der IF-Stufe befindet sich das Befehlszähler-Register (*Program Counter Register* – PC), das den zu holenden Befehl adressiert.
- Zwischen den IF- und ID-Stufen befindet sich ein weiteres PC-Register, das die um vier erhöhte Befehlsadresse des in der Stufe befindlichen Befehls puffert. Ein weiteres Pipeline-Register, das Befehlsregister (*Instruction Register*), enthält den Befehl selbst.
- Zwischen den ID- und EX-Stufen befindet sich erneut ein PC-Register mit der um vier erhöhten Adresse des in der Stufe befindlichen Befehls, da diese Befehlsadresse im Falle eines Sprungbefehls für die Berechnung der Sprungzieladresse benötigt wird. Weiterhin gibt es das erste und zweite ALU-Eingaberegister (*ALU Input Register 1* und *ALU Input Register 2*) zur Pufferung von Operanden aus dem Registersatz und ein Register (*Immediate Register*) zur Aufnahme von im Befehl angegebenen Konstanten, insbesondere auch der Anzahl der Bitstellen in Verschiebeboperationen. Durch die mit *Sign extended* bezeichnete Einheit werden Konstanten, die kürzer sind als 32 Bit, vorzeichenrichtig auf 32 Bit erweitert.
- Zwischen den EX- und MEM-Stufen befinden sich das ein Bit breite Bedingungsregister (*Condition Register*), das zur Aufnahme eines Vergleichsergebnisses dient, das ALU-Ausgaberegister (*ALU Output Register*) zur Aufnahme des Resultats einer ALU-Operation und das Speicherwertregister (*Store Value Register*) zum Puffern des zu speichernden Registerwerts im Falle einer Speicheroperation.
- Zwischen den MEM- und WB-Stufen befindet sich das Ladewertregister (*Load Memory Data Register*) zur Aufnahme des Datenwortes im Falle einer Ladeoperation und das ALU-Ergebnisregister (*ALU Result Register*) zur Zwischenspeicherung des ALU-Ausgaberegisterwertes aus der vorherigen Pipeline-Stufe.

Die Stufen der DLX-Pipeline im Detail

Während der Befehlsausführung werden folgende Schritte ausgeführt:

- In der *IF-Stufe* wird der Befehl, auf den der PC zeigt, aus dem Code-Cache (*I-cache*, *Instruction Cache*) in das Befehlsregister geholt und der PC um vier erhöht, um auf den nächsten Befehl im Speicher zu zeigen. (Die Erhöhung um vier wird wegen des einheitlich 32 Bit breiten Befehlsformats und der Byteadressierbarkeit des DLX-Prozessors durchgeführt, um die Befehlsadresse um vier Bytes weiterzuschalten.) Im Fall eines vorangegangenen Sprungbefehls kann die Zieladresse aus der MEM-Stufe benutzt werden, um den PC auf die im nächsten Takt zu holende Anweisung zu setzen.

- In der *ID-Stufe* wird der im Befehlsregister stehende Befehl in der ersten Takthälfte decodiert. In der zweiten Hälfte der Stufe wird abhängig vom Opcode eine der folgenden Aktionen ausgeführt:

  - *Register-Register* (bei arithmetisch-logischen Befehlen): Die Operandenwerte werden von den (Universal-)Registern (*Register File*) in das erste und das zweite ALU-Eingaberegister verschoben.
  - *Speicherreferenz* (bei Lade- und Speicherbefehlen): Ein Registerwert wird von einem (Universal-)Register in das erste ALU-Eingaberegister übertragen. Konstanten (*immediate*) oder Verschiebewerte (*displacement*) aus dem Befehl werden vorzeichenerweitert und in das *Immediate*-Register transferiert. Da die „registerindirekte Adressierung mit Verschiebung“ die komplexeste Adressierungsart des DLX-Prozessors ist, kann im nachfolgenden Pipeline-Schritt der Registeranteil mit dem Verschiebewert addiert werden, um die effektive Adresse zu berechnen. Im Falle der direkten oder absoluten Adressierung wird der Registerwert Null aus Register R0 in das erste ALU-Eingaberegister geladen. Im Falle der registerindirekten Adressierung (ohne Verschiebung) wird statt des vorzeichenerweiterten Verschiebewertes der Wert Null erzeugt und in das Verschieberegister geladen. (Eigentlich wird dafür ein weiterer MUX benötigt, der in der Abbildung nicht gezeigt ist.) Im Fall eines Speicherbefehls wird der zu speichernde Registerwert in das zweite ALU-Eingaberegister transferiert.
  - *Steuerungstransfer* (bei Sprungbefehlen): Der Verschiebewert innerhalb des Befehls wird vorzeichenerweitert und zur Berechnung der Sprungzieladresse in das Verschieberegister transferiert (wir erlauben nur den befehlszählerrelativen Adressierungsmodus mit 16 Bit breiten Verschiebewerten - abweichend von den 26 Bit PC-Offset-Werten aus Abschnitt 1.3.2). Im Falle eines bedingten Sprungs wird der Wert der Verzweigungsbedingung (*true* oder *false*) zum zweiten ALU-Eingaberegister transferiert. Eine vorangegangene Vergleichsoperation muss diesen Wert produziert und im Universalregister gespeichert haben. Der Ausgang des zweiten ALU-Eingaberegisters ist mit einem Vergleicherschaltnetz (zero?) verbunden, das gepuffert über ein Bedingungsregister (Conditional Register) den Multiplexer vor dem PC-Register der IF-Stufe ansteuert. Wenn die Verzweigungsbedingung erfüllt wurde, wird der neue PC-Wert aus dem ALU-Ausgaberegister übernommen.

- In der *EX-Stufe* erhält die arithmetisch-logische Einheit ihre Operanden je nach Befehl aus den ALU-Eingaberegistern, aus dem (ID/EX-) PC-Register oder aus dem *Immediate-Register* und legt das Ergebnis der arithmetisch-logischen Operation in dem ALU-Ausgaberegister ab. Die Inhalte dieses Registers hängen vom Befehlstyp ab, welcher die MUX-Eingänge auswählt und die Operation der ALU bestimmt. Je nach Befehl können folgende Operationen durchgeführt werden:

  - *Register-Register* (bei arithmetisch-logischen Befehlen): Die ALU führt die arithmetische oder logische Operation auf den Operanden aus dem ersten und dem zweiten ALU-Eingaberegister durch und legt das Ergebnis im ALU-Ausgaberegister ab.
  - *Speicherreferenz* (bei Lade- und Speicherbefehlen): Von der ALU wird die Berechnung der effektiven Adresse durchgeführt und das Ergebnis im ALU-Ausgaberegister abgelegt. Die Eingabeoperanden erhält die ALU aus dem ersten ALU-Eingaberegister und dem *Immediate-Register*. Im Fall eines Speicherbefehls wird der Inhalt des zweiten ALU-Eingaberegisters (das den zu speichernden Wert beinhaltet) unverändert in das Speicherwertregister verschoben.
  - *Steuerungstransfer* (bei Sprungbefehlen): Die ALU berechnet die Zieladresse des Sprungs aus dem (ID/EX-)PC-Register und dem *Immediate-Register* und legt die Sprungzieladresse im ALU-Ausgaberegister ab. Gleichzeitig wird die Sprungrichtung (die feststellt, ob der Sprung ausgeführt wird oder nicht) aus dem zweiten ALU-Eingaberegister auf Null getestet und das Boole'sche Ergebnis im Bedingungsregister gespeichert.

- Die *MEM-Stufe* wird nur für Lade-, Speicher- und bedingte Sprungbefehle benötigt. Folgende Operationen werden abhängig von den Befehlen unterschieden:

  - *Register-Register*: Das ALU-Ausgaberegister wird in das ALU-Ergebnisregister übertragen.
  - *Laden*: Das Speicherwort wird so, wie es vom ALU-Ausgaberegister adressiert wird, aus dem Daten-Cache (D-Cache) gelesen und im Ladewertregister platziert.
  - *Speichern*: Der Inhalt des Speicherwertregisters wird in den Daten-Cache geschrieben, wobei der Inhalt des ALU-Ausgaberegisters als Adresse benutzt wird.
  - *Steuerungstransfer*: Für genommene Sprünge wird das Befehlszähler-Register (PC) der IF-Stufe durch den Inhalt des ALU-Ausgaberegisters ersetzt. Für nicht genommene Sprünge bleibt der PC unverändert. (Das Bedingungsregister trifft die MUX-Auswahl in der IF-Stufe.)

- In der *WB-Stufe* wird während der ersten Takthälfte der Inhalt des Ladewertregisters (im Falle eines Ladebefehls) oder des ALU-Ergebnisregisters (in allen anderen Fällen) in das (Universal-)Register gespeichert. Der Resultatregisterselektor (*Result Register Selector*) aus dem Befehl, der das (Universal-) Register als Resultatregister bezeichnet, wird ebenfalls durch die Pipeline weitergegeben. Diese Weitergabe der Registerselektoren ist in Abbildung 1.22 jedoch nur angedeutet.

In Abbildung 1.22 wird nur der Datenfluss durch die Pipeline-Stufen gezeigt. Die Steuerungsinformation, die während der ID-Stufe aus dem Opcode

generiert wurde, fließt durch die nachfolgenden Pipeline-Stufen und steuert die Multiplexer und die Operation der ALU.

Dabei benutzen alle Pipeline-Stufen unterschiedliche Ressourcen. Deshalb werden zum Beispiel, nachdem ein Befehl zur ID geliefert wurde, die von der IF genutzten Ressourcen frei und zum Holen des nächsten Befehls benutzt. Idealerweise wird in jedem Takt ein neuer Befehl geholt und an die ID-Stufe weiter geleitet. Die Taktzeit wird durch den „kritischen Pfad“ vorgegeben, das bedeutet durch die langsamste Pipeline-Stufe. Ideale Bedingungen bedeuten, dass die Pipeline immer mit aufeinander folgenden Befehlen bzw. deren Befehlsausführungen gefüllt sein muss.

Es gibt leider mehrere potentielle Probleme, die eine reibungslose Befehlsausführung in der Pipeline stören können. Wenn zum Beispiel sowohl der Befehls- als auch der Datencache nachgeladen werden müssen und nur ein Speicherkanal (*memory port*) existiert, so erzeugt ein Ladebefehl in der MEM-Stufe einen Speicher-Lesekonflikt zwischen der IF- und der MEM-Stufe. In diesem Fall muss die Pipeline einen der Befehle anhalten, bis der benötigte Speicherkanal zum Nachladen des zweiten Caches wieder verfügbar ist. Auch gehen wir davon aus, dass der Registersatz mit zwei Lesekanälen und einem Schreibkanal ausgestattet ist, so dass gleichzeitig sowohl in der ID-Stufe zwei Operanden aus den Registern gelesen werden können als auch in der WB-Stufe ein Resultat in ein Register geschrieben werden kann. Trotzdem können bestimmte Hemmnisse die reibungslose Ausführung in einer Pipeline stören und zu sogenannten Pipeline-Konflikten führen.

1.5.3 Pipeline-Konflikte

Als **Pipeline-Konflikt** bezeichnet man die Unterbrechung des taktsynchronen Durchlaufs der Befehle durch die einzelnen Stufen der Befehls-Pipeline. Pipeline-Konflikte werden durch Daten- und Steuerflussabhängigkeiten im Programm oder durch die Nichtverfügbarkeit von Ressourcen (Ausführungseinheiten, Registern etc.) hervorgerufen. Diese Abhängigkeiten können, falls sie nicht erkannt und behandelt werden, zu fehlerhaften Datenzuweisungen führen. Die Situationen, die zu Pipeline-Konflikten führen können, werden auch als **Pipeline-Hemmnisse** (*pipeline hazards*) bezeichnet.

Es werden drei Arten von Pipeline-Konflikten unterschieden:

- **Datenkonflikte** treten auf, wenn ein Operand in der Pipeline (noch) nicht verfügbar ist oder das Register bzw. der Speicherplatz, in den ein Resultat geschrieben werden soll, noch nicht zur Verfügung steht. Datenkonflikte werden durch Datenabhängigkeiten im Befehlsstrom erzeugt.

- **Struktur**- oder **Ressourcenkonflikte** treten auf, wenn zwei Pipeline-Stufen dieselbe Ressource benötigen, auf diese aber nur einmal zugegriffen werden kann (vgl. den oben beschriebenen Speicher-Lesekonflikt).

- **Steuerflusskonflikte** treten bei Programmsteuerbefehlen auf, wenn in der Befehlsbereitstellungsphase die Zieladresse des als nächstes auszuführenden Befehls noch nicht berechnet ist bzw. im Falle eines bedingten Sprunges noch nicht klar ist, ob überhaupt gesprungen wird.

In den nächsten Abschnitten werden die Pipeline-Konflikte und Möglichkeiten, wie man sie eliminiert oder zumindest ihre Auswirkung mindert, diskutiert.

1.5.4 Datenkonflikte und deren Lösungsmöglichkeiten

Man betrachte zwei aufeinander folgende Befehle (*Instructions*) I_1 und I_2, wobei I_1 vor I_2 ausgeführt werden muss. Zwischen diesen Befehlen können verschiedene Arten von **Datenabhängigkeiten** bestehen, die **Datenkonflikte** zwischen zwei Befehlen verursachen können, wenn die beiden Befehle so nahe beieinander sind, dass ihre Überlappung innerhalb der Pipeline ihre Zugriffsreihenfolge auf ein Register oder einen Speicherplatz im Hauptspeicher verändern würde.

Datenabhängigkeitsarten

- Es besteht eine **echte Datenabhängigkeit** (*true dependence*) von Befehl I_1 zu Befehl I_2, wenn I_1 seine Ausgabe in ein Register *Reg* (oder in den Speicher) schreibt, das von I_2 als Eingabe gelesen wird. Eine echte Datenabhängigkeit kann einen **Lese-nach-Schreibe-Konflikt** *(Read After Write* – RAW) verursachen.
- Es besteht eine **Gegenabhängigkeit** (*anti dependence)* von I_1 zu I_2, falls I_1 Daten von einem Register R (oder einer Speicherstelle) liest, das anschließend von I_2 überschrieben wird. Durch eine Gegenabhängigkeit kann ein **Schreibe-nach-Lese-Konflikt** *(Write After Read* – WAR) verursacht werden.
- Es besteht eine **Ausgabeabhängigkeit** (*output dependence*) von I_2 zu I_1, wenn beide in das gleiche Register R (oder eine Speicherstelle) schreiben und I_2 sein Ergebnis nach I_1 schreibt. Eine Ausgabeabhängigkeit kann zu einem **Schreibe-nach-Schreibe-Konflikt** *(Write After Write – WAW)* führen.

Als Beispiel betrachte man die folgende Befehlsfolge, deren Abhängigkeitsgraph in Abbildung 1.23 dargestellt ist:

```
S1: ADD R1,R2,2 ; R1 = R2+2 S2: ADD R4,R1,R3; R4 = R1+R3 S3: MULT
R3,R5,3 ; R3 = R5*3 S4: MULT R3,R6,3 ; R3 = R6*3
```

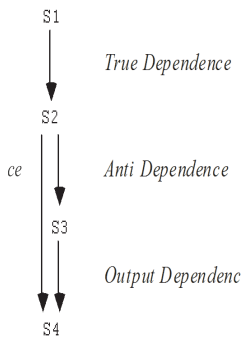

Abbildung 1.23: Abhängigkeitsgraph.

In diesem Fall besteht:

- eine echte Datenabhängigkeit von `S1` nach `S2`, da `S2` den Wert von Register `R1` benutzt, der erst in `S1` berechnet wird;
- eine Gegenabhängigkeit von `S2` nach `S3`, da `S2` den Wert von Register `R3` benutzt, bevor `R3` in `S3` einen neuen Wert zugewiesen bekommt; eine weitere Gegenabhängigkeit besteht von `S2` nach `S4`;
- eine Ausgabeabhängigkeit von `S3` nach `S4`, da `S3` und `S4` beide dem Register `R3` neue Werte zuweisen.

Gegenabhängigkeiten und Ausgabeabhängigkeiten werden häufig auch **falsche Datenabhängigkeiten** oder entsprechend dem englischen Begriff *Name Dependency* **Namensabhängigkeiten** genannt. Diese Arten von Datenabhängigkeiten sind nicht problemimmanent, sondern werden durch die Mehrfachnutzung von Speicherplätzen (in Registern oder im Arbeitsspeicher) hervorgerufen. Sie können durch Variablenumbenennungen entfernt werden.

Echte oder wahre Datenabhängigkeiten werden häufig auch einfach als Datenabhängigkeiten bezeichnet. Echte Datenabhängigkeiten repräsentieren den Datenfluss durch ein Programm.

Selbsttestaufgabe 1.5 (Programm-Daten- und Steuerflussabhängigkeiten)

Es sei folgende Programmsequenz gegeben:

```
S1: ADDI    R1, R2, #2 ; R1 = R2 + 2
S2: SUB     R4, R1, R3 ; R4 =R1 -- R3
S3: SGE     R7, R4, R0 ; R4 >= 0 ? Status in R7
S4: BNEZ    R7, S7 ; wenn ja, gehe zu S7
S5: MULT    R3, R5, R6 ; R3 = R5 * R6
S6: J S8    ; Goto S8
S7: ADDI    R3, R3, #2 ; R3 = R3 + 2
S8: ADDI    R4, R4, #1 ; R4 = R4 +1
```

Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in dieser Programmsequenz. Stellen Sie diese in einem Abhängigkeitsbaum dar.

Lösung auf Seite 66

Die ersten drei Datenabhängigkeiten in Abbildung 1.23 erzeugen Datenkonflikten in der Reihenfolge RAW, WAR, WAW. Datenkonflikte werden zwar durch Datenabhängigkeiten hervorgerufen, sind jedoch auch wesentlich von der Pipeline-Struktur bestimmt. Wenn die datenabhängigen Befehle weit genug voneinander entfernt sind, so wird kein Datenkonflikt ausgelöst. Wie weit die Befehle voneinander entfernt sein müssen, hängt jedoch von der Pipeline-Struktur ab. Für einfache skalare Pipelines wie unsere DLX-Pipeline sind nur Lese-nach-Schreibe-Konflikte von Bedeutung. Für Superskalarprozessoren müssen jedoch auch Schreibe-nach-Lese- und Schreibe-nach-Schreibe-Konflikte beachtet werden.