

2.1 Prozessoren

► Abbildung 2.1 zeigt den Aufbau eines einfachen busorientierten Computers. Die **CPU** (Central Processing Unit – zentrale Verarbeitungseinheit) oder der **Prozessor**¹ ist das „Gehirn“ des Computers. Sie führt die im Hauptspeicher abgelegten Programme aus, indem sie deren Befehle nacheinander abruft, analysiert und dann ausführt. Die Komponenten sind über einen **Bus** miteinander verbunden, d.h. durch ein Leitungssystem, dessen (parallel geführte) Leitungen Adressen, Daten und Steuersignale übertragen. Externe Busse verbinden die CPU mit dem Speicher und den E/A-Geräten, aber auch innerhalb der CPU werden Busse verwendet – mehr dazu in Kürze. Moderne Computer verfügen über mehrere Busse.

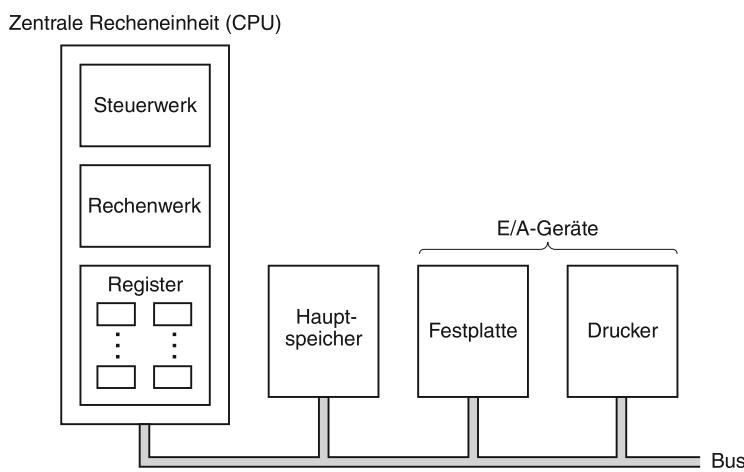


Abbildung 2.1: Der Aufbau eines einfachen Computers mit einer CPU und zwei E/A-Geräten

Die CPU gliedert sich in mehrere Funktionseinheiten. Das Steuerwerk (die Steuereinheit – Control Unit) ruft Befehle aus dem Hauptspeicher ab und bestimmt den Befehlstyp. Das Rechenwerk (Arithmetic Logic Unit, ALU) führt Operationen aus, die für den jeweiligen Befehl erforderlich sind, beispielsweise Addition oder boolesche AND-Verknüpfung.

Außerdem enthält die CPU einen kleinen Hochgeschwindigkeitsspeicher, der Zwischenergebnisse und bestimmte Steuerinformationen temporär aufnimmt. Dieser Speicher besteht aus mehreren Registern, die jeweils eine bestimmte Größe und Funktion haben. Gewöhnlich sind alle Register gleich groß. Jedes Register kann einen Wert aufnehmen, dessen Größe von der Registerbreite – d.h. der Anzahl der Bits in einem Register – abhängt. Da sich die Register innerhalb der CPU befinden, können sie sehr schnell gelesen und geschrieben werden.

Das wichtigste Register ist der **Befehlszähler** (Program Counter – PC). Er zeigt auf den nächsten Befehl, der zur Ausführung abzurufen ist. Die Bezeichnung „Befehlszähler“ ist etwas irreführend, da im eigentlichen Sinne nichts *gezählt* wird. Allerdings hat sich dieser Begriff so eingebürgert. Ebenfalls wichtig ist das **Befehlsregister** (Instruction Register – IR), das den momentan ausgeführten Befehl aufnimmt. Die meisten Computer besitzen außerdem eine Reihe weiterer Register, von denen einige universellen Charakter haben,

1 Sofern nicht anders angegeben, werden die Begriffe „CPU“ und „Prozessor“ in diesem Buch gleichberechtigt nebeneinander verwendet (Anm. d. Übersetzers).

andere nur für spezielle Aufgaben vorgesehen sind. Darüber hinaus gibt es noch Register, über die das Betriebssystem den Computer steuert.

2.1.1 Aufbau der CPU

► Abbildung 2.2 zeigt die innere Organisation für einen Teil einer typischen Von-Neumann-CPU. Dieser sogenannte **Datenpfad** (Data Path) besteht aus den Registern (normalerweise 1 bis 32), dem **Rechenwerk (ALU)** – Arithmetic Logic Unit und mehreren Busen, die die einzelnen Teile untereinander verbinden. Die Register münden in zwei Eingaberegister der ALU, die in der Abbildung mit *A* und *B* bezeichnet sind. Diese Register speichern die Eingabedaten der ALU, während diese beschäftigt ist. Der Datenpfad spielt in allen Prozessoren eine wichtige Rolle und wird deshalb im gesamten Buch ausführlich behandelt.

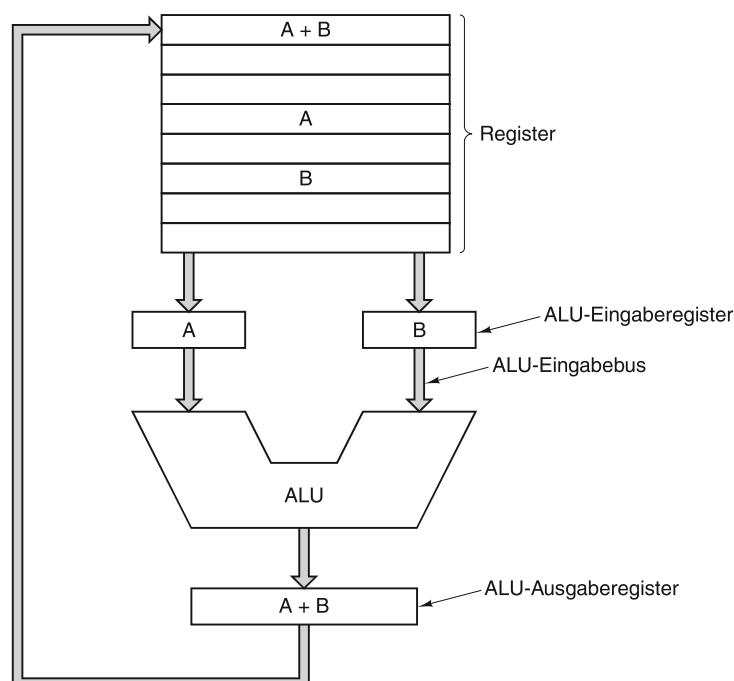


Abbildung 2.2: Der Datenpfad eines typischen Von-Neumann-Rechners

Die CPU besteht aus dem Steuerwerk, dem Rechenwerk (ALU) und den Registern, in denen Daten zwischen- gespeichert werden, die in der ALU verarbeitet werden.

Die ALU selbst führt Additionen, Subtraktionen und andere einfache Operationen mit ihren Eingabewerten aus und stellt das Ergebnis in das Ausgaberegister. Der Inhalt dieses Ausgaberegisters kann wieder zurück in ein Register gespeichert werden. Später lässt sich das Register bei Bedarf in den Hauptspeicher übernehmen. Nicht alle Entwürfe verfügen über die A-, B- und Ausgaberegister. Das Beispiel zeigt eine Addition, doch können ALUs auch andere Operationen ausführen.

Die meisten Befehle lassen sich in zwei Kategorien aufteilen: Register-Speicher- und Register-Register-Befehle. Mit Register-Speicher-Befehlen können Speicherwörter in Register übernommen werden, wo sie sich beispielsweise in darauffolgenden Befehlen als ALU-Eingaben verwenden lassen. („Wörter“ sind die Dateneinheiten, die zwischen Speicher und Registern bewegt werden. Ein Wort kann beispielsweise eine Ganzzahl sein. Auf den Speicheraufbau geht dieses Kapitel später ein.) Andere Register-Speicher-Befehle schreiben den Inhalt von Registern zurück in den Speicher.

Der zweite Typ sind die Register-Register-Befehle. Ein typischer Register-Register-Befehl ruft zwei Operanden aus den Registern ab, bringt sie in die ALU-Eingaberegister, führt mit ihnen eine Operation durch (z.B. eine Addition oder eine boolesche AND-Verknüpfung) und speichert das Ergebnis wieder in einem Register. Der Vorgang, bei dem zwei Operanden durch die ALU geschleust werden und das Ergebnis gespeichert wird, heißt **Datenpfadzyklus** (Data Path Cycle); er bildet die Basis der meisten Prozessoren. Er definiert maßgeblich, was der Computer tun kann. Moderne Computer besitzen mehrere ALUs, die parallel arbeiten und für unterschiedliche Funktionen spezialisiert sind. Je kürzer der Datenpfadzyklus ist, umso schneller ist der Prozessor.

2.1.2 Befehlsausführung

Die CPU führt jeden Befehl in einer Reihe kleiner Schritte aus. Diese stellen sich ungefähr wie folgt dar:

- 1 Den nächsten Befehl aus dem Speicher in das Befehlsregister abrufen.
- 2 Den Programmzähler ändern, damit er auf den nächsten Befehl zeigt.
- 3 Den Typ des gerade eingelesenen Befehls bestimmen.
- 4 Falls der Befehl ein Speicherwort benötigt, dessen Position bestimmen.
- 5 Das Wort bei Bedarf in ein CPU-Register laden.
- 6 Den Befehl ausführen.
- 7 Zu Schritt 1 gehen, um den nächsten Befehl auszuführen.

Diese Schrittfolge bezeichnet man auch als **Abrufen-Decodieren-Ausführen-Zyklus** (Fetch-Decode-Execute Cycle). Sie liegt der Arbeitsweise aller Computer zugrunde.

Die Beschreibung der Abläufe in einer CPU ähnelt einem in Umgangssprache formulierten Programm. ► Listing 2.1 zeigt, wie sich dieses formlose Programm als Java-Methode (d.h. als Prozedur) namens `interpret` neu schreiben lässt. Die damit interpretierte Maschine verfügt über zwei Register, die für die Benutzerprogramme sichtbar sind: den Befehlszähler (`PC`), der die Adresse des nächsten einzulesenden Befehls verfolgt, und den Akkumulator (`AC`), der das Ergebnis einer arithmetischen Operation aufnimmt. Weitere interne Register der Maschine speichern den aktuellen Befehl während seiner Ausführung (`instr`), den Typ des aktuellen Befehls (`instr_type`), die Operandenadresse des Befehls (`data_loc`) und den aktuellen Operanden selbst (`data`). Hier wird davon ausgegangen, dass Befehle jeweils eine einzige Speicheradresse enthalten. Die adressierte Speicherstelle enthält den Operanden, z.B. das zum Akkumulator zu addierende Datenelement.

```

public class Interp {
    static int PC;           // Programmzähler speichert Adresse des nächsten Befehls
    static int AC;           // Akkumulator, ein Register für Arithmetik
    static int instr;         // Register, das den aktuellen Befehl aufnimmt
    static int instr_type;   // Befehlstyp (Opcode)
    static int data_loc;     // Adresse der Daten oder -1, falls keine Daten
    static int data;          // Nimmt den aktuellen Operanden auf
    static boolean run_bit = true; // Ein Bit, das ausgeschaltet werden kann, um
                                  // die Maschine anzuhalten

    public static void interpret(int memory[ ], int starting_address) {
        // Diese Prozedur interpretiert Programme für eine einfache Maschine, wobei
        // jeder Befehl genau einen Speicheroperanden enthält.
        // Das Register AC (Akkumulator) ist für arithmetische Operationen vorgesehen.
        // Z.B. addiert der Befehl ADD eine im Speicher befindliche Ganzzahl zum AC.
        // Der Interpreter läuft so lange, bis das Ausführungsbit (run_bit) durch den
        // Befehl HALT ausgeschaltet wird.
        // Der Zustand eines auf dieser Maschine laufenden Prozesses ist durch den
        // Speicher, den Programmzähler, das Ausführungsbit und den Akkumulator
        // gekennzeichnet.
        // Die Eingabeparameter setzen sich aus dem Speicherbild und der
        // Anfangsadresse zusammen.

        PC = starting_address;
        while (run_bit) {
            instr = memory[PC];      // Den nächsten Befehl nach instr abrufen
            PC = PC + 1;             // Programmzähler inkrementieren
            instr_type = get_instr_type(instr); // Befehlstyp bestimmen
            data_loc = find_data(instr, instr_type); // Daten suchen (-1 falls keine)
            if (data_loc >= 0)        // Wenn data_loc gleich -1, kein Operand erforderlich
                data = memory[data_loc]; // Daten abrufen
            execute(instr_type, data); // Befehl ausführen
        }
    }

    private static int get_instr_type(int addr) { ... }
    private static int find_data(int instr, int type) { ... }
    private static void execute(int type, int data){ ... }
}

```

Listing 2.1: Ein in Java geschriebener Interpreter für einen einfachen Computer

Dass man mit einem Programm die Funktionen einer CPU nachbilden kann, zeigt bereits, dass Programme nicht unbedingt von einer „Hardware“-CPU – also einem Kästchen voller Elektronik – ausgeführt werden müssen. Stattdessen kann ein Programm auch dadurch ausgeführt werden, dass es seine Befehle durch ein anderes Programm abrufen, decodieren und ausführen lässt. Ein Programm (wie das in Listing 2.1), das die Befehle eines anderen Programms abruft, decodiert und ausführt, heißt (wie bereits in Kapitel 1 erwähnt) **Interpreter**.

Aus dieser Gleichwertigkeit von Hardware-Prozessoren und Interpretern ergeben sich wichtige Schlussfolgerungen für den Aufbau eines Computers und den Entwurf von Computersystemen. Nachdem die Maschinensprache L für einen neuen Computer spezifiziert ist, kann das Entwicklungsteam entscheiden, ob die Programme in L direkt durch einen Hardware-Prozessor oder stattdessen durch einen Interpreter ausgeführt bzw.

interpretiert werden sollen. Entscheidet sich das Team für einen Interpreter, muss es auch für eine Hardware-Maschine sorgen, die den Interpreter ausführen kann. Außerdem sind Hybridkonstruktionen möglich, bei denen ein Teil durch die Hardware ausgeführt und ein Teil durch die Software interpretiert wird.

Ein Interpreter zerlegt die Befehle seiner Zielmaschine in kleine Schritte. Daher kann die Maschine, auf der der Interpreter läuft, viel einfacher und billiger als ein Hardware-Prozessor für die Zielmaschine sein. Solche Einsparungen spielen besonders dann eine wichtige Rolle, wenn die Zielmaschine über eine große Zahl von Befehlen verfügt und es sich um relativ komplexe Befehle mit vielen Optionen handelt. Die Einsparungen ergeben sich im Wesentlichen daraus, dass Hardware durch Software (den Interpreter) ersetzt wird.

Die ersten Computer verfügten über kleine einfache Befehlssätze. Doch das Streben nach leistungsfähigeren Computern führte unter anderem auch zu leistungsstärkeren Einzelbefehlen. Schon recht früh erkannte man, dass komplexere Befehle oft eine schnellere Programmausführung zur Folge haben, auch wenn die Ausführung der einzelnen Befehle etwas länger dauert. Gleitkommabefehle und die direkte Unterstützung des Zugriffs auf Feldelemente sind Beispiele für komplexe Befehle. Manchmal ergab sich rein aus der Beobachtung, dass häufig dieselben zwei Befehle aufeinanderfolgten, sodass man diese beiden Befehle zu einem einzigen zusammenfassen konnte.

Die komplexeren Befehle waren besser, weil Operationen manchmal überlappend bzw. parallel auf verschiedener Hardware ausgeführt werden konnten. Bei teuren Hochleistungsrechnern ließen sich die Kosten für diese zusätzliche Hardware ohne Weiteres rechtfertigen. So kam es, dass teure Hochleistungsrechner über viel mehr Befehle als die billigeren verfügten. Doch steigende Kosten für die Softwareentwicklung und Forderungen nach Befehlskompatibilität führten dazu, dass man komplexe Befehle selbst in Computern des unteren Preissegments implementieren musste, wo es mehr auf niedrige Kosten als auf hohe Geschwindigkeit ankam.

Ende der 1950er Jahre hatte IBM (das zu dieser Zeit dominierende Computerunternehmen) erkannt, dass es sowohl für IBM als auch für ihre Kunden viele Vorteile bringt, wenn man nur eine einzige Familie von Rechnern unterstützt, die alle die gleichen Befehle ausführen. IBM führte den Begriff **Architektur** ein, um diese Kompatibilitäts Ebene zu beschreiben. Eine neue Computerfamilie bekam nun eine einheitliche Architektur, aber viele unterschiedliche Implementierungen, die alle dasselbe Programm ausführen konnten und sich nur hinsichtlich Preis und Geschwindigkeit unterschieden. Doch wie baut man einen billigen Computer, der all die komplexen Befehle der teuren Hochleistungsrechner ausführen kann?

Die Antwort liegt in der Interpretation. Diese erstmals von Maurice Wilkes (1951) vorgeschlagene Technik ermöglichte die Konstruktion einfacher und billiger Computer, die dennoch eine große Anzahl von Befehlen ausführen konnten. Das Ergebnis war die IBM-System/360-Architektur – eine Familie kompatibler Computer, die in Preis und Fähigkeiten nahezu zwei Größenordnungen umspannte. Nur in den teuersten Modellen kam eine direkte Hardware-Implementierung (d.h. ohne Interpretation) zur Anwendung.

Einfache Computer mit interpretierten Befehlen hatten noch weitere Vorteile. Zu den wichtigsten gehörten:

Durch **Interpretation** kann eine komplexe Architektur auf einer Familie von Rechnern unterschiedlicher Leistungsfähigkeit kostengünstig realisiert werden.

- 1 Die Möglichkeit, falsch implementierte Befehle vor Ort zu korrigieren und sogar Entwurfsmängel in der zugrunde liegenden Hardware auszugleichen.
- 2 Die Gelegenheit, neue Befehle zu minimalen Kosten selbst nach Auslieferung des Rechners hinzuzufügen.
- 3 Ein strukturiertes Design, das effizientes Entwickeln, Testen und Dokumentieren komplexer Befehle erlaubt.

Als der Computermarkt in den 1970er Jahren einen gewaltigen Aufschwung erfuhr und das Rechenpotenzial schnell wuchs, kam die Nachfrage nach billigen Rechnern insbesondere den Computerkonzepten mit Interpretern zugute. Die Möglichkeit, Hardware und Interpreter auf einen bestimmten Befehlssatz zuzuschneiden, erwies sich als kostengünstiges Entwurfsprinzip für Prozessoren. Da sich die zugrunde liegende Halbleitertechnik schnell weiterentwickelte, kam den Kostenvorteilen mehr Gewicht zu als den Möglichkeiten für höhere Leistung, und Interpreter-basierte Architekturen etablierten sich als Methode für den Computerentwurf. Fast alle in den 1970er Jahren neu entwickelten Computer – von Minicomputern bis Mainframes – basierten auf der Interpretation.

Ende der 1970er Jahre waren einfache Prozessoren, auf denen Interpreter liefen, mit Ausnahme der teuersten und leistungsstärksten Modelle wie der Cray-1 und der Cyber-Serie von Control Data, weit verbreitet. Durch Interpreter konnte man die höheren Kosten abfangen, die mit komplexen Befehlen verbunden sind. Die Architekten untersuchten immer komplexere Befehle und insbesondere Wege, um die zu verwendenden Operanden zu spezifizieren.

Dieser Trend erreichte seinen Höhepunkt mit dem VAX-Computer der Digital Equipment Corporation (DEC), der mehrere Hundert Befehle beherrschte und mehr als 200 unterschiedliche Varianten kannte, um die in jedem Befehl zu verwendenden Operanden zu spezifizieren. Leider war die VAX-Architektur von Anfang an als Interpreter-Implementierung konzipiert und man hatte sich wenig um die Implementierung eines Hochleistungsmodells gekümmert. Durch diese Einstellung führte man sehr viele Befehle mit geringem Nutzen ein, die sich zudem nur schwer direkt ausführen ließen. Dieser Mangel erwies sich letztlich als tödlich für die VAX und letztlich auch für DEC (Compaq kaufte DEC im Jahr 1998 und Hewlett-Packard übernahm Compaq im Jahr 2001).

Auch wenn die ersten 8-Bit-Mikroprozessoren einfache Maschinen mit sehr einfachen Befehlssätzen waren, hatte man bis Ende der 1970er Jahre sogar bei den Mikroprozessoren auf Interpreter-basierte Konzepte gewechselt. Zu dieser Zeit mussten sich die Entwickler von Mikroprozessoren vor allem damit auseinandersetzen, wie sie die zunehmende, durch integrierte Schaltungen ermöglichte Komplexität in den Griff bekommen. Mit dem Interpreter-basierten Ansatz konnte man einen einfachen Prozessor entwickeln, wobei sich die Komplexität hauptsächlich auf den Speicher beschränkte, der den Interpreter aufnahm. So ließ sich ein komplexes Hardware-Design in ein komplexes Software-Design überführen.

Der Erfolg des Motorola 68000 mit seinem großen interpretierten Befehlssatz und der gleichzeitige Misserfolg des Zilog Z8000 (mit einem ebenso großen Befehlssatz, aber ohne Interpreter) stellten die Vorteile des Interpreter-Ansatzes unter Beweis, wenn es darum ging, einen neuen Mikroprozessor so schnell wie möglich auf den Markt zu bringen. Besonders überraschend war dieser Erfolg angesichts des Vorsprungs, den Zilog hatte (der Z80 als Vorgänger des Z8000 war weitaus populärer als der Vorgänger des

68000, der 6800). Es kamen hier natürlich auch noch andere Faktoren ins Spiel, nicht zuletzt der Umstand, dass Motorola bereits eine lange Geschichte als Chiphersteller hinter sich hatte und Exxon (der Eigentümer von Zilog) eine Ölgesellschaft und kein Chiphersteller war.

Dem Interpreter-Konzept kam während dieser Zeit noch ein weiterer Faktor zugute: die Existenz schneller Nur-Lese-Speicher, auch als **Steuerspeicher** (Control Store) bezeichnet, die den Interpreter beherbergten.

Nimmt man an, dass der Interpreter einen typischen interpretierten Befehl in 10 sogenannten **Mikrobefehlen** zu jeweils 100 ns und zwei Referenzen auf den Hauptspeicher zu je 500 ns abgearbeitet hat, dann beträgt die Gesamtausführungszeit mit 2000 ns nur das Doppelte gegenüber der (bestmöglichen) direkten Ausführung. Ohne Steuerspeicher hätte der Befehl 6000 ns beansprucht. Und der 6-fache Zeitbedarf ist erheblich schwerer zu tolerieren als der doppelte.

2.1.3 RISC kontra CISC

Ende der 1970er Jahre experimentierte man mit sehr komplexen Befehlen – der Interpreter machte es möglich. Die Entwickler versuchten, die „semantische Lücke“ zwischen den Fähigkeiten der Maschinen und den Anforderungen von Hochsprachen zu schließen. Fast niemand dachte daran, einfachere Maschinen zu entwickeln – so wie man heutzutage kaum etwas in die Entwicklung vereinfachter Betriebssysteme, Netzwerke, Webserver usw. investiert (was vielleicht bedauerlich ist).

Eine Gruppe, die dem vorherrschenden Trend nicht folgte und stattdessen versuchte, einige Ideen von Seymour Cray in einen leistungsfähigen Minicomputer einzubringen, wurde von John Cocke bei IBM geleitet. Die Arbeit dieser Gruppe führte zu einem experimentellen Minicomputer mit der Bezeichnung **801**. Obwohl IBM diesen Rechner nie vermarktet hat und die Ergebnisse erst Jahre später veröffentlichte [Radin, 1982], wurde die Entwicklung bekannt und auch andere machten sich daran, ähnliche Architekturen zu untersuchen.

Im Jahre 1980 begann eine Gruppe an der Berkeley-Universität unter der Leitung von David Patterson und Carlo Séquin mit der Entwicklung von VLSI-CPU-Chips, die ohne Interpretation auskamen [Patterson, 1985; Patterson und Séquin, 1982]. Sie prägten den Ausdruck **RISC** für dieses Konzept und nannten ihren CPU-Chip RISC I, dem kurz darauf RISC II folgte. Ein Jahr später entwickelte und fertigte John Hennessy an der Stanford-Universität einen etwas anderen Chip mit der Bezeichnung **MIPS** [Hennessy, 1984]. Diese Chips mauserten sich zu kommerziell wichtigen Produkten – den SPARC- bzw. MIPS-Prozessoren.

Diese neuen Prozessoren unterschieden sich deutlich von den damaligen kommerziellen Prozessoren. Da die neuen Prozessoren nicht abwärtskompatibel zu bisherigen Produkten sein mussten, konnten die Entwickler neue Befehlssätze wählen, mit denen sich die Gesamtsystemleistung maximieren ließ. Lag der Schwerpunkt anfangs auf einfachen Befehlen, die sich schnell ausführen ließen, erkannte man bald, dass der Schlüssel zu hoher Rechnerleistung in der Entwicklung von Befehlen lag, die schnell **initialisiert** (gestartet) werden konnten. Es kam weniger darauf an, wie viel Zeit ein Befehl tatsächlich in Anspruch nahm, als vielmehr, wie viele davon pro Sekunde gestartet werden konnten.

RISC-Computer verwenden einen einfachen Befehlssatz, dessen Befehle sehr effizient abgearbeitet werden können. Die Verwendung eines Interpreters konnte zu Gunsten der direkten Hardwaresteuerung aufgegeben werden.

Als diese einfachen Prozessoren entwickelt wurden, ließ vor allem eine Eigenschaft aufhorchen – die relativ geringe Zahl von verfügbaren Befehlen, in der Regel um die 50. Das waren weit weniger als die 200 bis 300 Befehle der etablierten Rechner, wie zum Beispiel der DEC VAX oder der mächtigen Großrechner von IBM. Schließlich steht das Akronym RISC auch für **Reduced Instruction Set Computer** (Computer mit reduziertem Befehlssatz) – im Gegensatz zu CISC, was für **Complex Instruction Set Computer** (Computer mit komplexem Befehlssatz) steht. Letzteres war ein kaum verhüllter Hinweis auf die VAX, die damals die Informatikfachbereiche der Universitäten beherrschte. Heute glaubt fast niemand mehr, dass die Größe des Befehlssatzes entscheidend ist, doch der Name hält sich eben.

Um es kurz zu machen: Es entflammt ein „Religionskrieg“, in dem die RISC-Verfechter die etablierte Ordnung (VAX, Intel, IBM-Großrechner) angriffen. Ihrer Meinung nach ließe sich ein Computer am besten mit wenigen einfachen Befehlen konstruieren, die in einem einzigen Zyklus des in Abbildung 2.2 dargestellten Datenpfads ausgeführt werden können – nämlich zwei Register abrufen, sie irgendwie kombinieren (z.B. addieren oder per AND verknüpfen) und das Ergebnis wieder in ein Register schreiben. Denn selbst wenn ein RISC-Computer vier oder fünf Befehle benötigt, um das zu tun, wofür bei einem CISC-Computer nur ein einziger Befehl notwendig ist, gewinnt RISC, da die RISC-Befehle zehnmal schneller sind (weil sie nicht interpretiert werden). Außerdem muss man wissen, dass seinerzeit die Geschwindigkeit von Hauptspeichern mit der Geschwindigkeit der Nur-Lese-Steuerspeicher gleichgezogen hatte. Die Nachteile der Interpretation traten damit stärker zutage und es sprach immer mehr für RISC-Prozessoren.

Man sollte glauben, dass RISC-Maschinen (wie die Sun UltraSPARC) angesichts der Leistungsvorteile der RISC-Technologie die CISC-Prozessoren (wie den Intel Pentium) vom Markt verdrängt haben müssten. Doch nichts dergleichen ist geschehen. Warum nicht?

Intels CISC-Prozessoren verwenden eine Kombination von Interpretation und direkter Ausführung in einem RISC-Kern, um Abwärtskompatibilität und Geschwindigkeit zu garantieren.

Der erste Punkt betrifft die Abwärtskompatibilität und die Milliarden Dollar, die die Unternehmen bereits in die Software für Intel-Prozessoren investiert hatten. Zweitens war Intel überraschenderweise in der Lage, die einschlägigen Ideen auch in eine CISC-Architektur einzubringen. Angefangen beim 486, enthalten Intel-Prozessoren einen RISC-Kern, der die einfachsten (und typischerweise auch häufigsten) Befehle in einem einzigen Datenpfadzyklus ausführt, während die komplexeren Befehle in der üblichen CISC-Weise interpretiert werden. Unterm Strich sind häufig vorkommende Befehle also schnell und seltener vorkommende Befehle langsam. Dieser Hybridsatz kommt zwar noch nicht an einen reinen RISC-Entwurf heran, liefert aber immer noch eine konkurrenzfähige Gesamtperformance, wobei alte Software ohne Modifikationen weiterhin lauffähig ist.

2.1.4 Designprinzipien moderner Computer

Seit Einführung der ersten RISC-Prozessoren sind nun etwa drei Jahrzehnte vergangen und es haben sich bestimmte Entwurfsprinzipien durchgesetzt, die einen praktikablen Weg darstellen, um Computer nach dem aktuellen Stand der Hardware-Technologie zu entwerfen. Falls eine größere Veränderung in der Technologie eintreten sollte (beispielsweise ein neuer Fertigungsprozess, der die Speicherzykluszeit plötzlich um den Faktor 10 gegenüber der CPU-Zykluszeit verbessert), kann alles wieder ganz anders aussehen. Die

Computerentwickler sollten daher immer ein Auge auf technologische Veränderungen haben, die sich auf das Gleichgewicht zwischen den Komponenten auswirken können.

Trotzdem gibt es einen Satz von Designprinzipien, auch als **RISC-Designprinzipien** bezeichnet, die Architekten von Universalprozessoren nach Möglichkeit einhalten sollten. Externe Randbedingungen, wie die Forderung nach Abwärtskompatibilität zu einer bereits bestehenden Architektur, verlangen nicht selten Kompromisse. Diese Prinzipien sind aber Ziele, die von den meisten Designern angestrebt werden. Nachstehend werden die wichtigsten Prinzipien behandelt.

Die Hardware führt alle Befehle direkt aus

Alle gängigen Befehle werden direkt von der Hardware ausgeführt und nicht durch Mikrobefehle interpretiert. Indem man die Ebene der Interpretation ausschaltet, erreichen die meisten Befehle eine hohe Ausführungsgeschwindigkeit. Bei Computern mit CISC-Befehlsatz lassen sich komplexere Befehle in separate Bestandteile aufgliedern und als Sequenz von Mikrobefehlen ausführen. Dieser zusätzliche Schritt verlangsamt den Prozessor, was man bei weniger häufig vorkommenden Befehlen aber in Kauf nehmen kann.

Die Befehle werden mit maximaler Rate initiiert

Moderne Computer bedienen sich vieler Tricks, um ihre Leistung zu maximieren. Einer der häufigsten ist es, möglichst viele Befehle pro Sekunde zu starten. Wenn man 500 Millionen Befehle pro Sekunde initiieren kann, hat man immerhin einen 500-MIPS-Prozessor gebaut, ungeachtet dessen, wie lange die Ausführung dieser Befehle dauert. (**MIPS** steht für Millions of Instructions Per Second – Millionen Befehle je Sekunde. Der Name des MIPS-Prozessors ist als Wortspiel von diesem Akronym abgeleitet. Offiziell steht er für Microprocessor without Interlocked Pipeline Stages – etwa: Mikroprozessor ohne Pipeline-Sperren.) Dieses Prinzip unterstellt, dass Parallelität bei der Verbesserung des Leistungsverhaltens eine wichtige Rolle spielt, da man nur dann eine große Zahl von langsamem Befehlen in kurzer Zeit initiieren kann, wenn sich mehrere Befehle gleichzeitig ausführen lassen.

Die Verarbeitungsleistung von RISC-Prozessoren ergibt sich daraus, dass in jedem Takt ein neuer Befehl gestartet werden kann und die gestarteten Befehle parallel bearbeitet werden.

Obwohl der Prozessor die Befehle immer in der vom Programm vorgegebenen Reihenfolge übernimmt, werden die Befehle nicht immer in der Programmreihenfolge gestartet (weil möglicherweise eine benötigte Ressource gerade belegt ist) und sie müssen auch nicht in der Programmreihenfolge fertiggestellt werden. Wenn Befehl 1 ein Register setzt und Befehl 2 dieses Register verwendet, ist selbstverständlich zu gewährleisten, dass Befehl 2 das Register erst liest, wenn es den korrekten Wert enthält. Diese Arbeitsweise erfordert zwar einen erheblichen Verwaltungsaufwand, bietet aber auch das Potenzial für Leistungsgewinne, indem mehrere Befehle gleichzeitig ausgeführt werden.

Die Befehle müssen leicht zu decodieren sein

Die Decodierung der einzelnen Befehle, um die von ihnen benötigten Ressourcen zu bestimmen, bildet eine kritische Grenze für die Rate, mit der sich Befehle initiieren lassen. Alles, was diesen Vorgang unterstützt, ist dienlich. Dazu gehört, die Befehle nach einem regelmäßigen Muster mit einer festen Länge und einer geringen Anzahl von Feldern aufzubauen. Je geringer die Anzahl unterschiedlicher Formate ist, desto besser ist es.

Nur Lade- und Speichervorgänge sollen auf den Speicher verweisen

Operationen lassen sich am einfachsten in Einzelschritte auflösen, wenn die Operanden für die meisten Befehle aus CPU-Registern kommen und wieder dort abgelegt werden. Separate Befehle können die Operanden aus dem Speicher in Register übertragen. Da Speicherzugriffe viel Zeit beanspruchen können und die Verzögerungen nicht vorhersehbar sind, lassen sich diese Befehle am besten mit anderen Befehlen überlappen, wenn sie ausschließlich Operanden zwischen Registern und dem Speicher übertragen. Dies bedeutet, dass nur LOAD- und STORE-Befehle auf den Speicher verweisen sollten. Alle anderen Befehle sollten nur mit Registern arbeiten.

Ausreichend Register bereitstellen

Da Speicherzugriffe relativ zeitaufwendig sind, sollten viele Register (mindestens 32) bereitgestellt werden, damit ein einmal abgerufenes Wort in einem Register gehalten werden kann, bis es nicht mehr benötigt wird. Wenn nicht mehr genügend Register vorhanden sind und ihre Inhalte in den Speicher ausgelagert werden müssen, nur um sie später wieder zurückzuladen, ist das ein unbefriedigender Zustand, der möglichst zu vermeiden ist. Das erreicht man am besten mit einer ausreichend großen Anzahl von Registern.

2.1.5 Parallelität auf Befehlsebene

Die Computerarchitekten streben ständig danach, die Leistung der von ihnen entwickelten Rechner zu steigern. Eine Möglichkeit besteht darin, die Taktfrequenz der Chips zu erhöhen. Für jedes neue Design gibt es aber eine Grenze, was zu einem bestimmten Zeitpunkt in der Entwicklungsgeschichte gerade noch machbar ist. Daher weichen die meisten Computerarchitekten auf die Parallelität aus (wobei zwei oder mehrere Dinge gleichzeitig ablaufen), um bei gegebener Taktfrequenz noch mehr Leistung aus ihrem Design herauszuholen.

Eine Beschleunigung der Verarbeitung lässt sich durch die **parallele Ausführung** von Befehlen in einem Prozessor oder durch die Verwendung mehrerer Prozessoren erreichen.

Parallelität ist in zwei allgemeinen Formen möglich, nämlich auf Befehlsebene und auf Prozessorebene. Im ersten Fall wird die Parallelität innerhalb der einzelnen Befehle genutzt, um mehr Befehle je Sekunde von einem und demselben Prozessor verarbeiten zu lassen. Im zweiten Fall arbeiten mehrere Prozessoren am selben Problem zusammen. Beide Ansätze haben ihre Vorteile. Dieser Abschnitt beschäftigt sich mit Parallelität auf Befehlsebene; im nächsten Abschnitt geht es um Parallelität auf Prozessorebene.

Pipelining (Fließbandverarbeitung)

Seit Jahren ist bekannt, dass das Abrufen von Befehlen aus dem Speicher einen entscheidenden Engpass für die Geschwindigkeit der Befehlsausführung darstellt. Um dieses Problem zu lindern, verfügten schon Computer wie der IBM Stretch (1959) über die Fähigkeit, Befehle vorab aus dem Speicher abzurufen, damit sie sofort verfügbar sind, wenn der Prozessor sie benötigt. Diese Befehle wurden in einen speziellen Registersatz mit der Bezeichnung **Prefetch-Puffer** übernommen. Sobald der Prozessor einen Befehl benötigt hat, konnte er ihn gewöhnlich aus dem Prefetch-Puffer abrufen und musste nicht warten, bis der Speicherlesevorgang abgeschlossen ist.

Praktisch gliedert das Prefetching die Befehlsausführung in zwei Teile auf: Speicherzugriff und eigentliche Ausführung. Das Konzept der **Pipeline** (Fließband) führt diese Strategie noch wesentlich weiter. Die Befehlsausführung wird dabei nicht nur in zwei, sondern in

viele Teile (oftmals ein Dutzend oder mehr) aufgegliedert, die jeweils von einem dafür bestimmten Hardware-Block abgewickelt werden und alle parallel ablaufen können.

► Abbildung 2.3a zeigt eine Pipeline mit fünf Einheiten oder **Stufen** (Stages). Stufe 1 ruft den Befehl aus dem Speicher ab und stellt ihn in einen Puffer, bis er benötigt wird. Stufe 2 decodiert den Befehl und bestimmt dabei seinen Typ und die benötigten Operanden. Stufe 3 sucht die Operanden und ruft sie entweder aus Registern oder aus dem Speicher ab. Stufe 4 führt die eigentliche Abarbeitung des Befehls aus, die normalerweise darin besteht, die Operanden durch den Datenpfad (siehe Abbildung 2.2) zu schleusen. Schließlich schreibt Stufe 5 das Ergebnis in das dafür vorgesehene Register zurück.

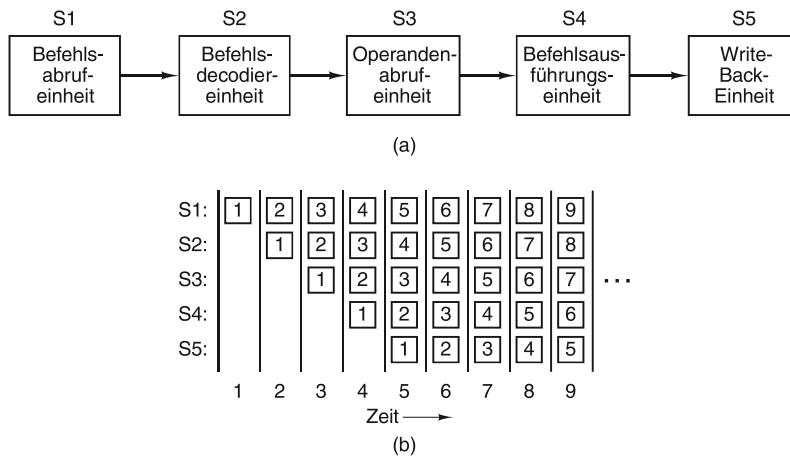


Abbildung 2.3: (a) Eine fünfstufige Pipeline (b) Der Status jeder Stufe als Funktion der Zeit. Es sind neun Taktzyklen dargestellt.

In ► Abbildung 2.3b ist zu sehen, wie die Pipeline als Funktion der Zeit arbeitet. In Taktzyklus 1 arbeitet die Stufe S1 an Befehl 1 – sie ruft ihn aus dem Speicher ab. In Zyklus 2 decodiert die Stufe S2 den Befehl 1, während Stufe S1 bereits Befehl 2 abruft. In Zyklus 3 ruft die Stufe S3 die Operanden für Befehl 1 ab, die Stufe S2 decodiert den Befehl 2 und die Stufe S1 ruft den dritten Befehl ab. In Zyklus 4 führt die Stufe S4 den Befehl 1 aus, S3 ruft die Operanden für Befehl 2 ab, S3 decodiert Befehl 3 und S1 ruft Befehl 4 ab. Schließlich schreibt in Zyklus 5 die Stufe S5 die Ergebnisse von Befehl 1 zurück, während die anderen Stufen an den darauffolgenden Befehlen arbeiten.

Eine Analogie soll das Konzept der Fließbandverarbeitung verdeutlichen. Stellen Sie sich eine Großbäckerei vor, in der das Kuchenbacken und das versandfertige Verpacken der Kuchen voneinander getrennt ablaufen. Nehmen wir an, die Versandabteilung hat ein langes Fließband, an dem in bestimmten Abständen fünf Arbeiter (Verarbeitungseinheiten) stehen. Alle 10 Sekunden (Taktfrequenz) legt Arbeiter 1 eine leere Kuchenschachtel auf das Fließband. Die Schachtel wandert zu Arbeiter 2, der einen Kuchen hineinlegt. Etwas später kommt die Schachtel an der Station von Arbeiter 3 an, wo sie geschlossen und versiegelt wird. Sie wandert weiter zu Arbeiter 4, der ein Etikett auf die Schachtel klebt. Schließlich nimmt Arbeiter 5 die Schachtel vom Fließband und legt sie in einen großen Behälter, der später an einen Supermarkt ausgeliefert wird. Nach dem gleichen Prinzip läuft die Fließbandverarbeitung im Computer ab: Jeder Befehl (Kuchen) durchläuft mehrere Verarbeitungsschritte, bis er am anderen Ende vollständig ausgeführt ist.

Die **Fließbandverarbeitung** von Befehlen beruht auf der parallelen Bearbeitung von Teilaufgaben unterschiedlicher Befehle. Die Initiierungsrate des Fließbandes bestimmt die Bandbreite (den Durchsatz) des Prozessors.

Gehen wir zurück zu unserer Pipeline in Abbildung 2.3 und nehmen wir an, dass die Zykluszeit dieses Prozessors 2 ns beträgt. Dann dauert es 10 ns, bis ein Befehl die gesamte fünfstufige Pipeline durchlaufen hat. Auf den ersten Blick möchte man meinen, dass der Prozessor mit 100 MIPS laufen kann, wenn ein Befehl 10 ns beansprucht, doch in Wirklichkeit leistet er viel mehr. In jedem Taktzyklus (2 ns) wird ein Befehl fertiggestellt, sodass die tatsächliche Verarbeitungsrate 500 MIPS und nicht 100 MIPS beträgt.

Die Fließbandverarbeitung ermöglicht einen Kompromiss zwischen der **Latenzzeit** (der Zeit, die für die Abarbeitung eines Befehls erforderlich ist) und der **Prozessorbandbreite** (wie viele MIPS der Prozessor leistet). Bei einer Zykluszeit von T ns und n Stufen in der Pipeline beträgt die Latenzzeit nT ns, weil jeder Befehl n Stufen durchläuft, die jeweils T ns benötigen.

Da jeweils ein Befehl in jedem Taktzyklus fertiggestellt wird und es $10^9/T$ Taktzyklen/Sekunde gibt, berechnet sich die Anzahl der je Sekunde ausgeführten Befehle zu $10^9/T$. Ist zum Beispiel $T = 2$ ns, dann werden jede Sekunde 500 Millionen Befehle ausgeführt. Um die Anzahl der MIPS zu erhalten, müssen wir die Befehlausführungsrate durch 1 Million dividieren, was $(10^9/T)/10^6 = 1000/T$ MIPS ergibt. Theoretisch könnten wir die Befehlausführungsrate in BIPS² statt MIPS messen, aber das macht niemand – wir auch nicht.

Superskalare Architekturen

Wenn eine Pipeline gut ist, sind zwei Pipelines sicherlich noch besser. ►Abbildung 2.4 zeigt ein mögliches Design für eine CPU mit zwei Pipelines auf der Basis von Abbildung 2.3. Hier ruft eine Befehlsabrufeinheit ein Befehlspaar gemeinsam ab und stellt jeden Befehl in eine eigene Pipeline, die wegen der Parallelverarbeitung auch beide über eine eigene ALU verfügen. Damit der Parallelbetrieb möglich ist, dürfen die beiden Befehle bei der Nutzung der Ressourcen (d.h. der Register) nicht kollidieren und auch nicht vom Ergebnis des jeweils anderen Befehls abhängen. Wie bei einer einzigen Pipeline muss entweder der Compiler dafür sorgen, dass dies gewährleistet ist (d.h., die Hardware führt keine Prüfung durch und liefert falsche Ergebnisse, wenn die Befehle nicht kompatibel sind), oder der Prozessor erkennt mit zusätzlicher Hardware Konflikte und löst sie während der Ausführung auf.

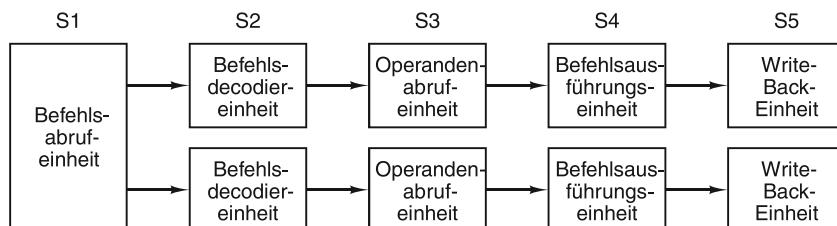


Abbildung 2.4: Zwei fünfstufige Pipelines mit einer gemeinsamen Befehlsabrufeinheit

Obwohl Pipelines – einzelne oder doppelte – ursprünglich bei RISC-Prozessoren zu finden waren, hat Intel mit dem 486 begonnen, Datenpipelines in seinen Prozessoren einzuführen (der 386 und seine Vorgänger besaßen keine Pipelines). Im 486 ist eine Pipeline realisiert und der ursprüngliche Pentium verfügt über zwei fünfstufige Pipelines, die ungefähr so wie in Abbildung 2.4 aufgebaut sind. Allerdings sieht die konkrete Auf-

² Billions of Instructions Per Second (engl. Billion = deutsch Milliarde) (Anm. d. Übersetzers).

teilung der Arbeit zwischen den Stufen 2 und 3 (namens Decode-1 und Decode-2) geringfügig anders aus als in unserem Beispiel. Die als **u-Pipeline** bezeichnete Haupt-Pipeline kann jeden beliebigen Pentium-Befehl ausführen. Die zweite Pipeline, die **v-Pipeline**, führt nur einfache Integer-Befehle (und einen einfachen Gleitkommabefehl – FXCH) aus.

Feststehende Regeln bestimmen, ob ein Befehlspaar kompatibel ist, sodass es sich parallel abarbeiten lässt. Wenn die Befehle in einem Paar zu komplex oder inkompatibel sind, wird nur der erste (in der u-Pipeline) ausgeführt. Der zweite Befehl wird zurückgehalten und mit dem nächsten gepaart. Die Reihenfolge der Befehle bleibt unverändert. Folglich können Pentium-spezifische Compiler, die kompatible Paare bilden, schnellere Programme als ältere Compiler erzeugen. Wie Messungen zeigen, ist bei gleicher Taktfrequenz ein Pentium, der für ihn optimierten Code ausführt, bei Integer-Programmen genau doppelt so schnell wie ein 486 [Pountain, 1993]. Dieser Gewinn ist allein der zweiten Pipeline zuzuschreiben.

Denkbar wäre zwar, auf vier Pipelines überzugehen, dies würde aber zu unnötiger Vervielfachung von Hardware führen (anders als Volkskundler glauben Informatiker nicht an die Magie der Zahl Drei). Stattdessen geht man bei Prozessoren des oberen Leistungsbereichs anders vor. Die Grundidee dabei ist, nur mit einer einzigen Pipeline zu arbeiten, dieser jedoch mehrere Funktionseinheiten zur Verfügung zu stellen, wie in ▶ Abbildung 2.5 gezeigt. Beispielsweise hat die Intel-Core-Architektur eine Struktur, die der in diesem Bild ähnelt. Kapitel 4 geht näher darauf ein. Für diesen Ansatz wurde 1987 der Begriff **superskalare Architektur** (Superscalar Architecture) geprägt [Agerwala und Cocke, 1987]. Die Wurzeln reichen jedoch mehr als 40 Jahre bis zum Computer CDC 6600 zurück. Der 6600 rief alle 100 ns einen Befehl ab und gab ihn an eine von zehn Funktionseinheiten zur parallelen Abarbeitung weiter, während die CPU den nächsten Befehl holte.

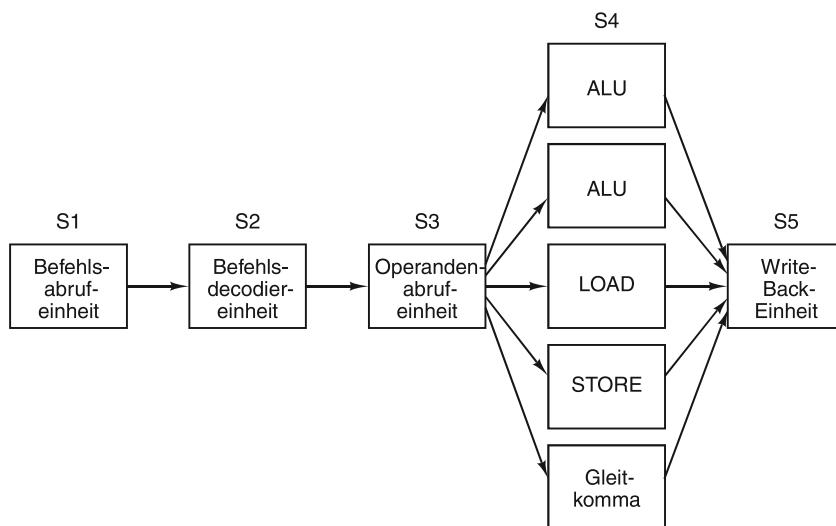


Abbildung 2.5: Ein superskalarer Prozessor mit fünf Funktionseinheiten

Superskalare Prozessoren sind in der Lage, mehrere Befehle pro Takt zu starten und auszuführen. Hierzu werden mehrere parallele Funktionseinheiten eingesetzt.

Die Definition von „superskalar“ hat sich im Laufe der Zeit gewandelt. Heute beschreibt man damit Prozessoren, die mehrere – oftmals vier oder sechs – Befehle in einem einzigen Taktzyklus initiieren. Natürlich muss ein superskalarer Prozessor mehrere Funktionseinheiten besitzen, an die er alle diese Befehle übergibt. Da superskalare Prozessoren im Allgemeinen eine Pipeline haben, ähneln sie dem Aufbau in Abbildung 2.5.

Der 6600 wäre nach dieser Definition aus technischer Sicht nicht superskalar, weil er nur einen Befehl je Zyklus gestartet hat. Allerdings war die Wirkung fast gleich: Befehle wurden mit einer wesentlich höheren

Rate initiiert als sie sich ausführen ließen. Der konzeptionelle Unterschied zwischen einer CPU mit einem 100-ns-Takt, die je Zyklus einen Befehl an eine Gruppe von Funktionseinheiten ausgibt, und einer CPU mit einem 400-ns-Takt, die je Zyklus vier Befehle an dieselbe Gruppe von Funktionseinheiten ausgibt, ist äußerst gering. Beiden Fällen liegt das Konzept zugrunde, dass die Initierungsrate wesentlich höher als die Ausführungsrate ist, wobei die Verarbeitungslast auf einen Verbund von Funktionseinheiten verteilt wird.

Das Konzept eines superskalaren Prozessors impliziert, dass die Stufe S3 Befehle wesentlich schneller verteilen kann als Stufe S4 sie ausführt. Wenn S3 alle 10 ns einen Befehl startet und alle Funktionseinheiten ihre Arbeit in 10 ns erledigen, ist immer nur eine Einheit beschäftigt – die ganze Idee wäre hinfällig. In Wirklichkeit brauchen die meisten Funktionseinheiten der Stufe 4 wesentlich mehr als einen Taktzyklus für die Befehlausführung. Dies gilt insbesondere für diejenigen, die auf den Speicher zugreifen oder Gleitkommaberechnungen durchführen. Wie Abbildung 2.5 zeigt, können in Stufe S4 mehrere ALUs vorhanden sein.

2.1.6 Parallelität auf Prozessorebene

Der Bedarf an immer schnelleren Computern scheint unstillbar zu sein. Astronomen möchten simulieren, was in der ersten Mikrosekunde nach dem Urknall geschah, Wirtschaftswissenschaftler streben danach, die Weltwirtschaft in einem Modell abzubilden, und Jugendliche wollen mit ihren virtuellen Freunden über das Internet interaktive, multimediale 3-D-Spiele spielen. Bei den immer schneller werdenden Prozessoren bekommt man schließlich Probleme mit den Signallaufzeiten aufgrund der Lichtgeschwindigkeit, die in Kupferdraht oder Glasfaserkabeln wohl bei 20 cm/ns bleiben wird, egal wie findig die Ingenieure bei Intel sind. Schnellere Chips produzieren zudem mehr Wärme, deren Ableitung ein Problem ist. Tatsächlich stagniert die Erhöhung der CPU-Taktfrequenzen in den letzten zehn Jahren vor allem aufgrund der Schwierigkeit, die in der CPU produzierte Wärme loszuwerden.

Parallelität auf Befehlsebene hilft schon etwas, doch Fließbandverarbeitung und superskalarer Betrieb bringen kaum mehr als das Fünf- oder Zehnfache an Gewinn. Ein Zuwachs um den Faktor 50, 100 und mehr lässt sich nur durch Computer mit mehreren Prozessoren erreichen. Deshalb sehen wir uns nun an, wie einige dieser Rechner organisiert sind.

Datenparallele Rechner

Viele Probleme in Physik, Technik und Computergrafik haben mit Schleifen und Datenfeldern (Arrays) zu tun oder lassen sich anderweitig durch eine sehr regelmäßige Struktur darstellen. Oftmals werden gleichartige Berechnungen wiederholt auf viele verschiedene Datenmengen angewendet. Aufgrund ihrer Regelmäßigkeit und Struktur lassen sich derartige Programme besonders einfach durch Parallelverarbeitung beschleunigen. Um solche Programme schnell und effizient auszuführen, sind vor allem zwei Methoden üblich: SIMD-Prozessoren und Vektorprozessoren. Obwohl sich diese beiden Schemas in vielerlei Hinsicht bemerkenswert ähnlich sind, betrachtet man ironischerweise das erste im Allgemeinen als Parallelcomputer, das zweite aber als Erweiterung eines Einzelprozessors.

Wegen ihrer bemerkenswerten Effizienz sind Parallelrechner in zahlreiche Anwendungsbereiche eingedrungen. Sie bewältigen eine beträchtliche Rechenleistung mit weniger Transistoren als alternative Konzepte. Gordon Moore (von dem das Moore'sche Gesetz stammt) hat einmal gesagt, dass Silizium ungefähr 1 Milliarde Dollar pro Acre (4047 m^2) kostet. Folglich kann eine Computerfirma beim Verkauf von Silizium mehr Gewinn erzielen, je mehr Rechenleistung sie aus dieser Siliziumfläche herausquetschen kann. Datenparallele Prozessoren gehören zu den effizientesten Mitteln, um Leistung aus Silizium zu pressen. Da alle Prozessoren denselben Befehl ausführen, braucht das System nur ein „Gehirn“, das den Computer steuert. Folglich kommt ein Prozessor mit nur einer Stufe zum Abrufen des Befehls, einer Stufe zum Decodieren und einem Satz an Steuerlogik aus. Dies ergibt eine erhebliche Einsparung an Silizium und verschafft Parallelrechnern einen haushohen Vorteil gegenüber anderen Prozessoren, sofern die auszuführende Software von Regelmäßigkeit und möglicher Parallelität geprägt ist.

Datenparallele Rechner führen die gleiche Operation auf einer großen Datenmenge aus. Dies kann durch eine große Zahl von ALUs erfolgen (SIMD-Rechner) oder durch die Zerlegung der Operation in Stufen einer Pipeline (Vektorrechner).

Ein **SIMD-Prozessor** (**Single Instruction-stream Multiple Data-stream**) besteht aus einer großen Anzahl identischer Prozessoren, die die gleiche Befehlssequenz auf unterschiedlichen Datenmengen ausführen. Der erste SIMD-Prozessor der Welt war der ILLIAC IV an der Universität von Illinois [Bouknight et al., 1972]. Das ursprüngliche Design des ILLIAC IV besteht aus vier Quadranten, wobei jeder Quadrant ein quadratisches 8×8 -Gitter aus Prozessor- und Speicherelementen enthält. Je eine Steuereinheit pro Quadrant verteilt die Befehle, die dann alle Prozessoren im Gleichtakt ausführen, wobei jeder Prozessor seine eigenen Daten aus seinem eigenen Speicher bearbeitet. Wegen Budgetbeschränkungen wurde allerdings nur ein Quadrant aufgebaut, der eine Leistung von 50 Megaflops (Millionen Gleitkommaoperationen in der Sekunde) erreichte. Man sagt, dass sich die Rechenleistung der ganzen Welt mit einem Schlag verdoppelt hätte, wenn man den Rechner jemals fertiggestellt und er sein ursprüngliches Leistungsziel (1 Gigaflop) erreicht hätte.

Moderne Grafikprozessoren (Graphics Processing Units, GPUs) machen ausgiebig von der SIMD-Verarbeitung Gebrauch, um massive Rechenleistung mit weniger Transistoren zu bieten. SIMD-Prozessoren sind für die Grafikverarbeitung prädestiniert, da die meisten Algorithmen sehr regelmäßig und durch wiederholte Operationen auf Pixeln, Eckpunkten, Texturen und Kanten gekennzeichnet sind. ► Abbildung 2.6 zeigt den SIMD-Prozessor im Kern der Fermi-GPU von Nvidia. Eine Fermi-GPU enthält bis zu 16 SIMD-Stream-Multiprozessoren (SM), wobei jeder SM über 32 SIMD-Prozessoren verfügt. In jedem Zyklus wählt der Scheduler zwei Threads aus, um sie auf dem SIMD-Prozessor auszuführen. Der nächste Befehl von jedem Thread läuft dann auf bis zu 16 SIMD-Prozessoren. Allerdings können auch weniger Prozessoren beteiligt sein, wenn nicht genügend Datenparallelität

vorhanden ist. Sofern jeder Thread in der Lage ist, 16 Operationen pro Zyklus auszuführen, schafft ein voll ausgestatteter Fermi-GPU-Kern mit 16 SMs stolze 512 Operationen je Zyklus – eine beeindruckende Leistung angesichts der Tatsache, dass ein Quad-Core-Universalprozessor etwa gleicher Größe gerade einmal 1/32 dieser Verarbeitungsleistung erreicht.

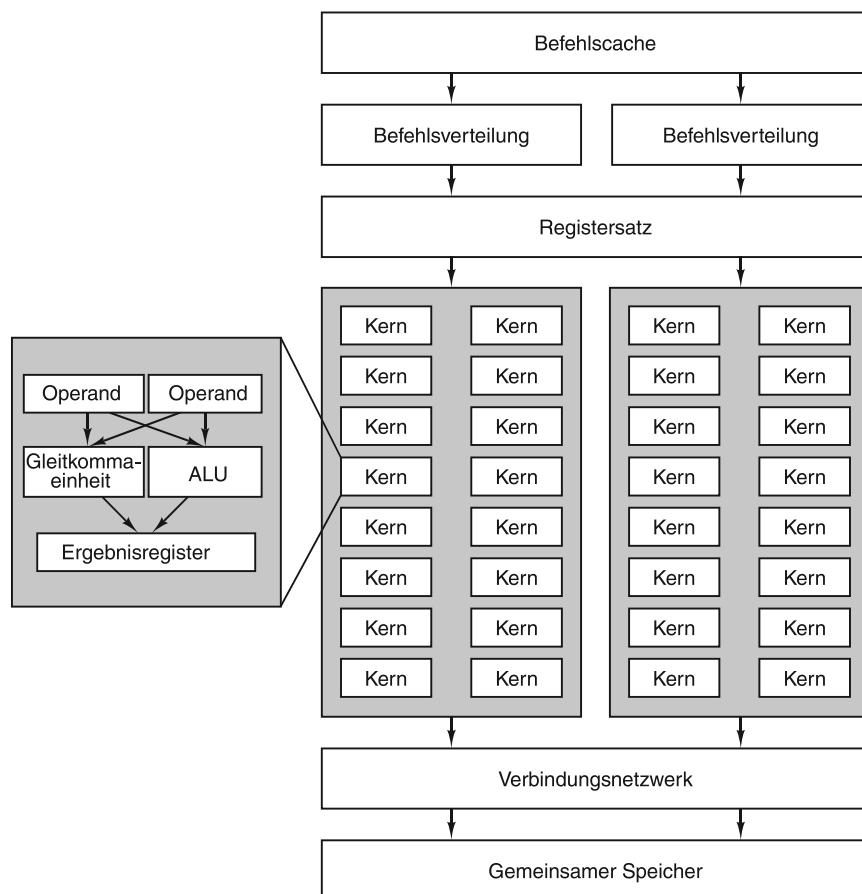


Abbildung 2.6: Der SIMD-Kern der Fermi-GPU

Für den Programmierer ist ein **Vektorprozessor** einem SIMD-Prozessor sehr ähnlich. Wie der SIMD-Prozessor führt er eine Sequenz von Operationen auf Paaren von Datenelementen sehr effizient aus. Doch anders als beim SIMD-Prozessor werden alle Operationen in einer einzigen Funktionseinheit mit massiver Fließbandverarbeitung ausgeführt. Die von Seymour Cray gegründete Firma Cray Research produzierte viele Vektorrechner, was 1974 mit der Cray-1 begann und sich bis zu aktuellen Modellen fortsetzt.

Sowohl SIMD- als auch Vektorprozessoren arbeiten auf Datenfeldern. Beide führen einzelne Befehle aus, die beispielsweise die Elemente von zwei Vektoren paarweise addieren. Während der SIMD-Prozessor dazu aber genauso viele Addierer wie Elemente im Vektor verwendet, arbeitet der Vektorprozessor nach dem Prinzip eines **Vektorregisters** (Vector Register), das aus einem Satz konventioneller Register besteht. Um diese Register aus dem Speicher zu laden, ist nur ein einziger Befehl erforderlich, der die Register

aber tatsächlich seriell aus dem Speicher lädt. Ein Befehl zur Vektoraddition addiert dann paarweise die Elemente zweier solcher Vektoren, indem er sie aus den beiden Vektorregistern abruft und einem Addierer mit Fließbandverarbeitung zuführt. Das Ergebnis des Addierers ist ein weiterer Vektor, der entweder in einem Vektorregister gespeichert oder unmittelbar als Operand für eine weitere Vektoroperation verwendet werden kann.

Die in der Core-Architektur von Intel verfügbaren SSE-Befehle (Streaming SIMD Extensions) nutzen dieses Ausführungsmodell, um regelmäßig wiederkehrende Berechnungen zu beschleunigen, beispielsweise in Multimediacomponenten und wissenschaftlicher Software. In dieser Beziehung kann man den ILLIAC IV als einen Vorgänger der Intel-Core-Architektur ansehen.

Mehrprozessorsysteme

Die verarbeitenden Elemente in einem datenparallelen Rechner sind keine unabhängigen CPUs, da es nur eine Steuereinheit gibt, die alle Einheiten gemeinsam nutzen. Das erste hier beschriebene Parallelsystem mit mehreren vollständigen Prozessoren ist das **Mehrprozessorsystem** (Multiprozessor), d.h. ein System mit mehreren Prozessoren, die auf einen gemeinsamen Speicher zugreifen – etwa wie eine Gruppe von Menschen in einem Raum, die alle dieselbe Wandtafel benutzen. Da jeder Prozessor jeden beliebigen Teil des Speichers lesen oder schreiben kann, ist eine Koordination (per Software) erforderlich, damit sich die Prozessoren nicht gegenseitig stören. Wenn zwei oder mehr Prozessoren eng miteinander zusammenarbeiten können, wie es bei Multiprozessorsystemen der Fall ist, sagt man, dass sie eng gekoppelt sind.

Mehrprozessorsysteme koppeln vollständige Prozessoren über einen gemeinsamen Speicher, der zum Austausch von Daten genutzt wird. Die Skalierbarkeit dieser Systeme ist durch den gemeinsamen Speicher begrenzt.

Es sind verschiedene Implementierungsschemata möglich. Das einfachste besteht aus einem einzelnen Bus, an den mehrere Prozessoren und ein Speicher angeschlossen sind.
► Abbildung 2.7a zeigt ein derartiges busbasiertes Mehrprozessorsystem.

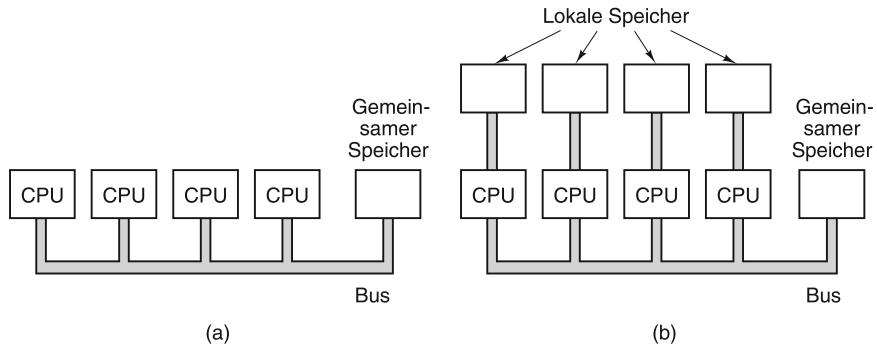


Abbildung 2.7: (a) Ein Mehrprozessorsystem mit einzelnen Bus (b) ein Mehrprozessorsystem mit lokalen Speichern

Man kann sich leicht vorstellen, dass es zwangsläufig zu Konflikten kommt, wenn eine große Anzahl schneller Prozessoren ständig versucht, über denselben Bus auf den Speicher zuzugreifen. Die Entwickler von Mehrprozessorsystemen haben sich verschiedene Schemas ausgedacht, um diese Konkurrenzsituation zu entschärfen und die Leistung zu verbessern. Der in ► Abbildung 2.7b gezeigte Entwurf ordnet jedem Prozessor einen eigenen lokalen Speicher zu, der den anderen Prozessoren nicht zugänglich ist. Dieser Speicher lässt sich für Programmcode und diejenigen Daten verwenden, die nicht gemein-