

Structured computer organization

by Andrew Tanenbaum · Language: DE

Summary by booksummaryai.com —

<https://booksummaryai.com/country/de/structured-computer-organization-zusammenfassung>

Summary

In 60 Sekunden

Tanenbaum's argument is that a computer is not one machine but a stack of levels, each built on the one below: digital logic, microarchitecture, instruction set, operating system, assembly language. He starts with Pascal's brass gears and ENIAC's 18,000 vacuum tubes, then walks you up the stack. You build flip-flops and memory chips, microprogram a Java virtual machine, decode x86 instructions, and page memory like an operating system. By the top, the machine you feared is one you could have assembled yourself.

Warum es sich lohnt

This is the book that turns the black box into a building you can walk through. Programmers who only ever touch high-level languages finally see what a cache, a page fault, or a stack push physically costs. Students of computer science get the standard text for architecture courses, written by the man who built MINIX to teach exactly this. Read it if you want to stop memorizing jargon and start seeing the machinery underneath.

Die Kernideen

Der Computer ist ein Stapel virtueller Maschinen

Tanenbaum eröffnet mit einer Entscheidung, vor der IBM Anfang der 1960er Jahre stand: Jede neue Maschine bedeutete neuen Code, und die Kunden ertranken in inkompatiblen Befehlssätzen. Die Antwort war das System/360, angekündigt als eine Familie, bei der ein einziger Befehlssatz Maschinen völlig unterschiedlicher Preisklasse und Leistung umspannen sollte. Diese Entscheidung spaltete das Feld in zwei Hälften: Architektur, die Attribute, die ein Programmierer sehen kann, und Organisation, die operativen Einheiten und Verbindungen, die die Spezifikation tatsächlich realisieren. Von dort aus spannt das Buch sein Rückgrat auf, eine Hierarchie von Ebenen von der digitalen Logik über Mikroarchitektur, Befehlssatz, Betriebssystemmaschine und Assemblersprache bis hinauf zu den höheren Programmiersprachen.

Sobald du die geschichtete Sicht akzeptierst, findet jeder verwirrende Aspekt der Rechentechnik ein Zuhause. Ein Seitenfehler gehört auf die Ebene des Betriebssystems; ein Cache-Miss gehört zur Mikroarchitektur darunter. Die Einsicht des System/360 ist, dass die Schnittstelle jede einzelne Implementierung überleben kann – deshalb laufen Jahrzehnte später Chips immer noch

mit demselben Code. Überraschenderweise sind die Ebenen nicht bloße Didaktik; so hat die Branche den Wandel tatsächlich gemeistert, ohne alles darüber Liegende zu zerstören.

Probier es aus: Nimm ein beliebiges Programm, das du diese Woche geschrieben hast, und benenne, welche Ebene der Hierarchie für jeden seiner langsamen Momente zuständig ist: einen Festplattenzugriff, eine falsch vorhergesagte Verzweigung, eine Garbage-Collection-Pause.

Generation Null lief mit Zahnrädern, nicht mit Silizium

Vor den Transistoren gab es Messing. Blaise Pascals Rechenmaschine (die Pascaline, gebaut 1642 für seinen Vater, einen Steuereintreiber) kam zuerst, und ein Jahrhundert später folgte Charles Babbages Entwurf der Analytical Engine mit ihrer Mühle (mill) und ihrem Speicher (store). Ada Lovelace, die als erste Programmiererin gilt, sah, was so eine Maschine leisten könnte. Dann kam 1946 die elektronische Ära an der University of Pennsylvania an: ENIAC (1946), gebaut an der University of Pennsylvania von John Mauchly und J. Presper Eckert, war der erste universelle elektronische Computer. Er wog 30 Tonnen, enthielt 18.000 Vakuumröhren und verbrauchte 140.000 Watt Leistung. Tanenbaum verbucht all dies unter Generation Null in seinen Meilensteinen der Computerevolution.

Der Sprung von Pascals Zahnrädern zu ENIACs Röhren ist nicht nur schnellere Arithmetik; es ist die Ankunft der Elektronik als Medium der Berechnung. Was Leser überrascht, ist, wie alt die Ideen sind: gespeicherte Programme, Programmierbarkeit, sogar die Rolle des Programmierers wurden skizziert, ein Jahrhundert bevor die Hardware mithalten konnte. Das 30-Tonnen-Gewicht und der 140.000-Watt-Hunger erklären, warum die nächste Generation, die Transistoren, so wichtig war. Miniaturisierung war keine Eitelkeit; sie war Überlebensnotwendigkeit.

Probier es aus: Skizziere die Mühle (mill) und den Speicher (store) der Analytical Engine neben einer modernen CPU und RAM und beschrifte, welcher Teil welche Rolle spielt.

Transistoren machten aus raumgroß schreibischgroß

Die erste Generation lief mit Vakuumröhren, im Stil des ENIAC, und sie brannten ständig durch. Die zweite Generation tauschte sie gegen Transistoren, und die Maschinen änderten ihren Charakter: Der IBM 1401 von 1959 war eine beliebte, aber keine wissenschaftliche Maschine, und der IBM 7094 von 1962 war ein typischer großer wissenschaftlicher Computer der zweiten Generation. Dann kamen für die dritte Generation integrierte Schaltkreise, mit dem IBM System/360 als wegweisender Familie der dritten Generation, und spätere Generationen trieben es mit LSI und VLSI weiter voran. Unter dem Ganzen läuft Moores Gesetz, zugeschrieben Gordon Moore (Intel-Mitgründer): Die Anzahl der Transistoren auf einem Chip verdoppelt sich alle 18–24 Monate.

Generationen sind keine Marketingetiketten; jede veränderte, was ein Computer sein konnte. Die Aufspaltung in 1401 und 7094 zeigt, dass selbst innerhalb einer Transistorgeneration kaufmännische und wissenschaftliche Maschinen unterschiedliche Produkte waren. Moores Gesetz ist der Motor hinter der ganzen Geschichte, eine Exponentialfunktion, die die raumgroße

ENIAC-Linie in Chips mit Milliarden von Schaltern verwandelte. Die Überraschung ist, wie vorhersehbar die Verdopplung war und wie viel der modernen Architektur einfach eine Antwort darauf ist.

Probier es aus: Schlage die Transistoranzahl des Chips in deinem Handy nach und zähle, wie viele Verdopplungen ihn von einem Prozessor der frühen 1970er Jahre trennen.

Ein gespeichertes Programm veränderte alles

Das Buch nennt John von Neumann und seinen Bericht von 1945 über die EDVAC, in dem das Konzept des gespeicherten Programms eingeführt wurde; die klassische Von-Neumann-Architektur folgt daraus. Der IAS-Entwurf hat fünf Kernteile: Speicher, arithmetisch-logische Einheit (ALU), Steuereinheit sowie Eingabegeräte, wobei der IAS-Speicher 4096 Wörter zu je 40 Bits fasst. Die CPU lebt von ihrem Zyklus aus Holen, Dekodieren und Ausführen, und Busse für Adressen, Daten und Steuerung tragen den Verkehr zwischen CPU, Speicher und Geräten, mit Gerätecontrollern wie einem Festplattencontroller dazwischen. Dasselbe Kapitel geht durch Spuren und Sektoren einer Festplattenplatte, definiert Seek-Zeit, Rotationslatenz und Übertragungszeit und stapelt dann Cache gegen RAM und Festplatte in der Speicherhierarchie.

Die Idee des gespeicherten Programms ist täuschend einfach: Befehle leben im selben Speicher wie Daten, sodass ein Programm geladen, verändert und wie Information behandelt werden kann. Alles Seitherige, von Compilern bis zu Betriebssystemen, hängt an diesem einen Schritt. Die Speicherhierarchie erklärt eine Tatsache, die jeder Programmierer irgendwann spürt: Schneller Speicher ist klein und teuer, also setzen Systeme auf Lokalität und halten ein wenig von dem, was du brauchst, nah – und den Rest fern. Seek-Zeit und Rotationslatenz sind der Grund, warum zufällige Festplattenlesevorgänge wehtun.

Probier es aus: Miss die Zeit einer Schleife, die eine große Datei sequenziell liest, im Vergleich zum zufälligen Springen in ihr, und beobachte, wie die Kosten für Seek und Latenz auftauchen.

Booles Algebra traf 1937 auf Schaltkreise

George Boole hatte eine Algebra der Logik erfunden, doch sie lag ungenutzt für Schaltungen herum, bis Claude Shannon in seiner Masterarbeit von 1937 sie auf Schaltkreise anwandte und den digitalen Schaltungsentwurf begründete, wie wir ihn kennen. Aus dieser Verbindung stammen die Grundoperationen: AND, OR, NOT, NAND, NOR, XOR und XNOR. Das Kapitel stellt die grundlegenden Logikgatter vor (AND, OR, NOT, NAND, NOR, XOR) und zeigt, wie man aus Gattern einen Decoder (der einen von mehreren Ausgängen aktiviert) und einen Multiplexer (der einen von mehreren Eingängen auswählt) baut. Zwei Jahrhunderte Mathematik werden plötzlich zu Schaltplänen.

Die Einsicht ist, dass Berechnung und Logik dasselbe Ding in verschiedener Kleidung sind. Sobald Shannon sie verbunden hatte, ließ sich jeder logische Ausdruck physisch realisieren, und jede Schaltung ließ sich algebraisch analysieren. NAND und NOR verdienen ihren Ruhm, weil sich

alles andere aus ihnen bauen lässt. Decoder und Multiplexer sind die ersten Muster, die es zu merken lohnt; sie kehren auf jeder Ebene der Maschine wieder, von der Auswahl von Speicherchips bis zum Routing im Datenpfad.

Probier es aus: Baue auf dem Papier einen 2-zu-1-Multiplexer aus AND-, OR- und NOT-Gattern und schreibe dann seine Wahrheitstabelle anhand der Verdrahtung.

Der Takt zähmt das Chaos in Schaltungen

Drähte, die rechnen, sind leicht; Drähte, die sich erinnern, sind der Knackpunkt. Das Kapitel stellt Latches und Flipflops (z. B. SR-Latch, D-Flipflop) als fundamentale 1-Bit-Speicherzellen für sequenzielle Schaltungen vor und behandelt getaktete Schaltungen, flankengetriggerte D-Flipflops und daraus gebaute Register. Aus Flipflops aufgebaute Register werden zu Schieberegistern und Zählern, und der Takt mit Flankentriggerung ist es, was synchronen Schaltungen erlaubt, Wettlaufsituationen zu vermeiden. Dann werden die Zellen zu Arrays massiert: Speicherchips, adressiert über Chip-Select, Adressleitungen und Datenleitungen; beschrieben werden SRAM und DRAM, mit dem Hinweis, dass DRAM periodisch aufgefrischt werden muss, weil seine Kondensatoren Ladung verlieren.

Synchroner Entwurf ist ein Friedensvertrag: Statt Signale gegeneinander rasen zu lassen, einigt man sich darauf, sich nur zu bewegen, wenn der Takt tickt. Diese eine Disziplin macht Million-Gatter-Entwürfe analysierbar. DRAMs undichte Kondensatoren sind ein schöner Hack, der ständige Auffrischarbeit gegen enorme Dichte tauscht – deshalb ist der Hauptspeicher DRAM und der Cache SRAM. Speicher als riesiges adressiertes Array ist das mentale Modell, das sich bis hinauf zum virtuellen Speicher hält.

Probier es aus: Zeichne ein Timing-Diagramm eines D-Flipflops, das zeigt, warum der Ausgang nur an der Taktflanke wechseln kann, nicht wenn der Eingang wackelt.

Ein CPU-Chip ist eine Stadt mit Busfahrplan

Wenn man von einzelnen Gattern zurückschwenkt, baut das Kapitel CPU-Chips als komplexe Integration einer ALU, von Registern und Steuerlogik auf einem einzigen Stück Silizium zusammen, mit Pins, die in Adress-, Daten- und Steuersignale gegliedert sind. Damit Chips miteinander reden können, erklärt das Buch Busprotokolle mit Lese- und Schreib-Timingdiagrammen und behandelt Busse auf der Logikebene, einschließlich Busmastern, Arbitrierung und dem Tri-State-Puffer als Schlüsselbauteil, das es vielen Treibern erlaubt, sich eine Leitung zu teilen. Hier hört die Maschine auf, ein Diagramm zu sein, und wird zu einem physischen Objekt, das man in der Hand halten könnte, mit Metallbeinchen und Regeln darüber, wer wann sprechen darf.

Der Tri-State-Puffer ist der stille Held: Ohne eine Möglichkeit, einen Chip von einer gemeinsam genutzten Leitung abzukoppeln, wären Busse unmöglich. Arbitrierung macht aus einer geteilten Ressource eine geordnete – dasselbe Problem, das Betriebssysteme später mit Scheduling lösen. Timingdiagramme vermitteln die tiefste Lektion der Hardware: Korrektheit ist nicht nur Logik,

sondern Logik plus Zeit. Jedes Lesen und Schreiben ist eine choreografierte Abfolge von Signalwechseln, und eine einzige verletzte Setup-Zeit ruiniert alles.

Probier es aus: Suche online das Pinout einer alten CPU heraus und sortiere seine Pins von Hand in Adress-, Daten- und Steuergruppen.

Mikroprogrammierung: die Maschine in der Maschine programmieren

Auf der Mikroarchitekturebene baut das Buch die MIC-1 (Mic-1), eine einfache mikroprogrammierte Lehrmaschine mit 32-Bit-Datenpfad (Register MAR, MDR, PC, SP, LV, CPP, TOS, OPC), einer ALU plus Shifter und einem Steuerspeicher. Ihre Aufgabe ist es, den IJVM-Befehlssatz (Integer Java Virtual Machine) zu implementieren, der ILOAD, ISTORE, IADD, ISUB, BIPUSH, DUP, SWAP und IF_ICMPEQ umfasst und auf einem stapelbasierten Modell ausgeführt wird. Das Mikroprogramm implementiert den IJVM-Interpreter über einen Fetch-Execute-Zyklus auf Mikroebene, geschrieben in MAL (Micro Assembly Language), mit Mikroinstruktionen, die über MBR und MPC sequenziert werden. Das Mikroinstruktionsformat enthält Felder: Addr (nächste Adresse), JAM (Sprungbedingung), ALU (ALU-Operation), C (Registerladung), Mem und B (Busquelle).

Das ist die konkreteste Demonstration der Schichten-These des Buches: Ein Befehlssatz kann eine Interpretation sein, die auf einem anderen läuft. Der IJVM sieht für den Programmierer wie eine echte Maschine aus, aber jeder seiner Befehle ist eine kurze Sequenz von Mikroinstruktionen, die aus dem Steuerspeicher geholt werden. Stapelbasierte Ausführung bedeutet, dass die meiste Arbeit im Pushen, Poppen und Blicken auf TOS besteht, was das dedizierte Register des Datenpfads antizipiert. Wenn man IADD einmal bis zur Mikroebene verfolgt hat, hört der Begriff „virtuelle Maschine“ auf, abstrakt zu sein.

Probier es aus: Verfolge IADD von Hand: Liste auf, welche Register sich ändern und welchen Speicher – falls überhaupt – jede Mikroinstruktion berührt.

x86: vierzig Jahre Ballast, läuft trotzdem noch

Die Ebene der Befehlssatzarchitektur nimmt die Intel-x86-Familie (8086 bis Pentium, IA-32) als laufendes Beispiel eines CISC-Entwurfs. Du lernst ihren Registersatz, das segmentierte Speichermodell (Segmentregister wie CS, DS, SS) und die Befehlskodierung variabler Länge kennen, dazu Adressierungsarten (unmittelbar, direkt, Register, Register indirekt, indiziert, Stapel). Befehlstypen (Datenbewegung, dyadische Arithmetik, monadische Operationen, Vergleiche und bedingte Sprünge, Prozeduraufruf und -rückkehr, E/A, Zeichenkettenverarbeitung), mit der Verwendung des Stapels zum Sichern von Rückkehradressen und lokalen Variablen, sowie rekursive Prozeduren. Das Kapitel führt die RISC-Ideen außerdem auf Arbeiten von David Patterson in Berkeley und John Hennessy in Stanford zurück und merkt an, dass moderne x86-Prozessoren CISC-Befehle intern in RISC-ähnliche Mikrooperationen übersetzen.

Die ISA ist der Vertrag zwischen Hardware und Software, und x86 zeigt, was passiert, wenn ein Vertrag nie gebrochen werden kann: Er wächst an. Kodierung variabler Länge und Segmentierung sind Geschichte, die du in jeder modernen Binärdatei mitschleppst. Die RISC-Geschichte erklärt den Dreh in jedem modernen Chip: Die CISC-Befehle, die du siehst, sind eine Fassade, darunter in einfache RISC-ähnliche Mikrooperationen dekodiert. Kompatibilität gewann den Markt; Einfachheit gewann den Silizium.

Probier es aus: Disassembliere ein winziges C-Programm und identifiziere je einen Befehl aus jeder Adressierungsart-Familie, die das Kapitel auflistet.

Virtueller Speicher: die Illusion, in der jedes Programm lebt

Auf der Maschinenebene des Betriebssystems hört der Speicher auf, physikalisch zu sein. Das Kapitel erklärt Paging (die Aufteilung des Speichers in Seiten fester Größe), Seitentabellen (die virtuelle Adressen in physikalische übersetzen) und Seitenersetzungsalgorithmen wie LRU, FIFO und den Second-Chance-Algorithmus. Ein Seitenfehler hält das Programm an, während das System die Seite von der Platte holt; dazu das Working-Set-Modell und Thrashing, wenn zu wenig Speicher verfügbar ist. Beschrieben wird auch Segmentierung als Alternative oder Ergänzung zum Paging, einschließlich segmentiertem Paging in der Intel-Pentium-Architektur. Unter all dem übersetzt die MMU (Memory Management Unit) virtuelle Adressen in physikalische. Ein Prozess wird als Programm in Ausführung mit eigenem virtuellem Adressraum und eigenen Ressourcen beschrieben, mit Zuständen (Running, Ready, Blocked) und der Rolle von Scheduler und Dispatcher beim Kontextwechsel.

Virtueller Speicher ist der größte Vertrauensbetrug der Informatik: jedes Programm glaubt, die ganze Maschine zu besitzen. Der Mechanismus ist eine Übersetzungstabelle pro Prozess plus Hardware – deshalb ist die MMU so wichtig wie die CPU. Seitenersetzung ist ein Vorhersageproblem, und das Working-Set-Modell ist seine Theorie; Thrashing ist das, was passiert, wenn die Vorhersage systemweit scheitert. Der Running-Ready-Blocked-Zyklus ist der Herzschlag jeder Multitasking-Maschine, die du je benutzt hast.

Probier es aus: Öffne den Speichermonitor deines Systems, suche die Seitenfehlerzähler eines laufenden Prozesses und beobachte, wie sie ausschlagen, wenn du schwere Anwendungen wechselst.

Dijkstras Semaphore lehren Prozesse, zu warten, bis sie dran sind

Programme auf dieser Ebene erbitten Dienste vom Betriebssystem über virtuelle E/A-Befehle auf der Maschinenebene des Betriebssystems, darunter Dateioperationen, Prozesserzeugung und Interprozesskommunikation – und wie das Betriebssystem sie über Traps implementiert. Behandelt werden Gerätetreiber, Interrupt-Handler und DMA (Direct Memory Access) für effiziente Datenübertragung. Wenn Prozesse Ressourcen teilen müssen, führt das Kapitel Semaphore ein, von Edsger Dijkstra eingeführt, zur Lösung von kritischen Abschnitten und Producer-Consumer-Problemen. Die Maschinenebene, so zeigt sich, ist der Ort, an dem das

Betriebssystem den Frieden wahrt.

Traps sind das Scharnier der ganzen Ebene: derselbe Befehl, der wie ein Prozeduraufruf aussieht, überquert tatsächlich eine Schutzgrenze in den Kernel. DMA zählt, weil Daten mit der CPU zu kopieren Verschwendung ist; lässt man das Gerät direkt mit dem Speicher sprechen, wird der Prozessor für echte Arbeit frei. Semaphore lösen ein Problem, das trivial klingt – zwei Prozesse, eine Ressource – und dessen naive Lösungen sämtlich in den Lücken zwischen Befehlen scheitern. Dijkstras Einsicht war, das Warten selbst zu einem erstklassigen, zählbaren Objekt zu machen.

Probier es aus: Schreibe zwei Threads, die einen gemeinsamen Zähler ohne Sperre erhöhen, beobachte die verlorenen Updates und repariere es dann mit einer Semaphore oder einem Mutex.

Der Assembler ist ein Zweipass-Übersetzer mit Gedächtnis

Die Assemblersprache wird als symbolische Darstellung der Maschinensprache auf der ISA-Ebene definiert, eins zu eins zu den Maschinenbefehlen. Ein Assembler arbeitet in zwei Durchgängen: Im ersten baut er eine Symboltabelle auf, im zweiten erzeugt er Code, übersetzt Mnemonics in Opcodes und löst Labels in Adressen auf. Behandelt werden Makros und Makroexpansion zur Assemblierungszeit, mit der Unterscheidung von Makros und Prozeduren – ein Unterschied, der Leute überrascht, die sie für austauschbar halten. Schließlich Linking und Loading, einschließlich Relocation, absoluter und linkender Loader sowie Objektdateiformaten: die Rohrleitungen, die aus getrennten Kompilaten ein laufendes Programm machen.

Der Zweipass-Entwurf existiert wegen der Vorwärtsreferenzen: einen Sprung zu einem Label, das du noch nicht gesehen hast, kannst du nicht ausfüllen – also sammelt Durchgang eins, und Durchgang zwei löst auf. Makros gegen Prozeduren ist in Wahrheit eine Wahl zwischen Geschwindigkeit und Platz, entschieden zu unterschiedlichen Zeiten: Assemblierungszeit gegen Laufzeit. Relocation ist der Grund, warum es Objektdateien überhaupt gibt: Code, der ohne Kenntnis seiner endgültigen Adresse geschrieben wurde, lässt sich trotzdem kombinieren. Jedes Build-System, das du heute benutzt, erbt diese Pipeline.

Probier es aus: Schreibe eine zweizeilige Assemblerdatei mit einem Vorwärtssprung und simuliere Durchgang eins und zwei von Hand, um zu sehen, wie sich die Symboltabelle füllt.

Flynns vier Kästchen ordnen jede Parallelmaschine ein

Wenn ein Prozessor nicht reicht, greift das Kapitel zu Flynns Taxonomie: SISD (single instruction, single data), SIMD (single instruction, multiple data), MISD und MIMD, mit Diskussion von SIMD, MIMD und Vektorrechnern. Es erklärt Multiprozessoren mit gemeinsamem Speicher (SMPs), darunter busbasierte SMPs, Crossbar-Schalter und mehrstufige Vermittlungsnetze wie das Omega-Netzwerk. Dann kommt das Problem, das das Fachgebiet definiert: das Cache-Kohärenzproblem. Lösungen wie Snooping-Busse und das MESI-Protokoll (Modified, Exclusive, Shared, Invalid), das die Caches mehrerer Prozessoren konsistent hält.

Flynns vier Kästchen sind das Periodensystem des Fachgebiets: Fast jede Maschine, der du begegnest, passt in eines davon. Das eigentliche Drama ist die Kohärenz, denn Caches existieren, um langsamen Speicher zu verbergen – und ihn vor mehreren Prozessoren gleichzeitig zu verbergen erzeugt Lügen. MESI funktioniert, indem es jeden Cache zum Tratscher am gemeinsamen Bus macht: Zustandswechsel werden bekannt gegeben, damit kein Prozessor mit veralteten Daten arbeitet. Die Hierarchie der Vermittlungsnetze – Bus, Crossbar, Omega – ist ein Kompromiss zwischen Kosten und Konflikten, der sich auf jeder Skala wiederholt.

Probier es aus: Finde die Cacheline-Größe und die Kernanzahl deiner CPU und überlege, in welchem MESI-Zustand eine gemeinsam genutzte, nur-lesbare Nachschlagetabelle leben würde.

Der Cosmic Cube: Multicomputer lernen, Zettel durchzureichen

Der andere Weg zur Parallelität verabschiedet sich völlig vom gemeinsamen Speicher. In Multicomputern mit Nachrichtenübermittlung haben die Prozessoren privaten Speicher und kommunizieren über explizite Nachrichten in einem Verbindungsnetzwerk. Früher Meilenstein war der Cosmic Cube, gebaut am Caltech von Charles Seitz als früher Hypercube-Multicomputer. Besprochen werden Topologien von Verbindungsnetzwerken wie Gitter, Hypercube und schalterbasierte Netze etwa mit Crossbar-Schaltern. Wo der SMP um einen Speicher streitet, ist der Multicomputer ein Bund kleiner Maschinen, die per Post verhandeln.

Nachrichtenübermittlung tauscht Bequemlichkeit gegen Skalierung: kein Kohärenzproblem, weil nichts geteilt wird, aber jeder Austausch muss explizit sein. Der Hypercube ist eine elegante Topologie: Jeder Knoten ist mit seinen binären Nachbarn verbunden, wodurch der Durchmesser logarithmisch bleibt, wenn die Maschine wächst. Seitz' Cosmic Cube bewies, dass ein Cluster bescheidener Prozessoren sich wie eine große Maschine verhalten kann. Das ist die Herkunft der heutigen Rechenzentren, in denen deine Cloud-Instanz ein Knoten ist, der Nachrichten an seine Nachbarn schickt.

Probier es aus: Zeichne einen vierdimensionalen Hypercube, beschrifte seine 16 Knoten mit binären Adressen und prüfe, dass jeder Knoten mit genau 4 Nachbarn verbunden ist.

Die besten Zitate

- "Generation Zero" - Andrew S. Tanenbaum
- "mill and store" - Andrew S. Tanenbaum
- "Modified, Exclusive, Shared, Invalid" - Andrew S. Tanenbaum
- "single instruction, multiple data" - Andrew S. Tanenbaum

Kritik

The book's age shows at the edges: hardware examples and capacity numbers reflect the editions' era, so readers wanting current GPUs, NVMe storage, or modern multicore cache hierarchies will need supplements. The layered model, while pedagogically clean, smooths over

how blurry the boundaries are in real designs, where compilers, microcode, and operating systems leak into each other. Some chapters are dense with detail, timing diagrams and protocol tables, that reward patience more than excitement. Absolute beginners may want a gentler on-ramp before tackling the microprogram chapters.

In einem Satz

A computer is a tower of levels, each one a machine pretending to be the one above it, and you can climb every rung.

Für wen sich das Buch eignet

Computer science students facing architecture courses, and self-taught programmers who suspect their high-level language is hiding a machine they should meet. Also ideal for curious engineers who want the whole stack, gates to grids, in one coherent frame.