Prozessorarchitektur

Hans Jürgen Ohlbach

Keywords: CPU, von Neumann Architektur, Cache, Pipelining, Superskalarität, Hyperthreading, Multicore Architektur, Parallelrechner.

Inhaltsverzeichnis

1 Einführung

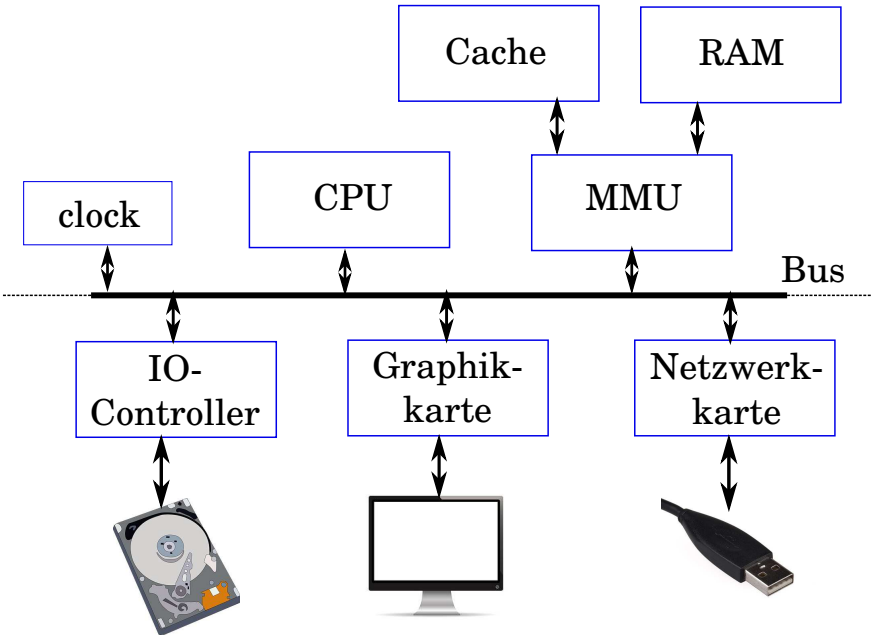
Wenn man sich einen neuen Computer kauft und die technische Beschreibung liest, dann tauchen da Begriffe auf wie „Multicore Prozessor“, „Level 1,2 Cache“, „Taktfrequenz“, „Pipelining“, „Hyperthreading“, usw. In diesem Miniskript sehen wir uns daher die Architektur von Computern etwas genauer an und erklären all diese Begriffe. Auch für die Programmierung ist es oft nützlich, zu wissen wie genau die Programme auf der Hardware abgearbeitet werden, zumindest wenn man effizienten Code schreiben will. Da kann es sogar einen signifikanten Performanzunterschied machen, ob man eine zweidimensionale Matrix zeilenweise oder spaltenweise abläuft.

Fast alle Rechner, die man heute kaufen kann haben einen Mehrkernprozessor auf dem Programme *parallel* abgearbeitet werden können. Dafür kann und sollte man die Programme so schreiben, dass man auch die Parallelität ausnutzt. Ohne genaue Kenntnis, wie solche Programme abgearbeitet werden, kann man allerdings schlimme und schwer zu entdeckende Fehler machen. Auch dafür ist es hilfreich, sich den Aufbau solcher Prozessoren anzusehen.

Aufbauend auf den technischen Details der Prozessoren kann man verstehen, wie deren Maschinenprogramme aussehen und verarbeitet werden. Das wird in einem weiteren Miniskript thematisiert.

2 Architektur eines Computers

Die heutige Architektur eines Computers geht auf den ersten frei programmierbaren Rechner zurück, den kurz nach dem 2. Weltkrieg von dem Mathematiker *John von Neumann* entwickelten MANIAC Rechner. Sein Entwurf war so überzeugend, dass man heute noch von der *von Neumann Architektur* spricht. Das folgende Bild gibt einen ersten groben einen Überblick über diese Architektur, allerdings in modernisierter Form.



Ein sog. *Bus* dient als zentraler Kommunikationskanal zwischen den einzelnen Komponenten des Rechners. Über ihn lauf die Bitsequenzen. Meist gebt es aber nicht nur einen, sondern mehrere Busse, die verschiedenen Zwecken dienen. Insbesondere unterteilt man ihn in den *Datenbus*, über den die Daten laufen, und den *Adressbus*, über den die Adressen der Daten verschickt werden.

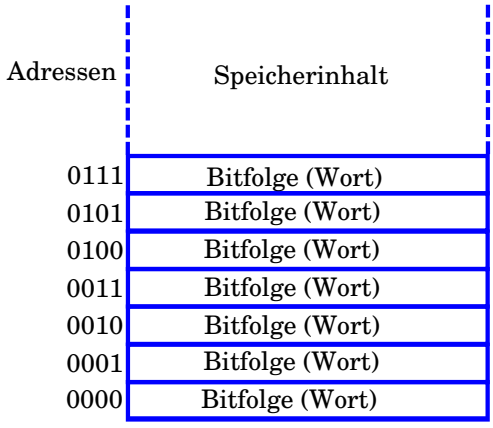
Die CPU (Central Processing Unit) ist die zentrale Recheneinheit, in der die eigentlichen Berechnungen ablaufen. Deren Struktur werden wir uns noch genauer ansehen. Die CPU hat selbst nur wenig Speichermöglichkeiten (sog. *Register*). I.A. holt sie ihre Daten aus dem *Hauptspeicher*, dem RAM (Random Access Memory). Um den Zugriff auf die Daten zu beschleunigen, gibt es oft zwischen CPU und RAM noch einen kleineren, aber schnelleren *Cache Speicher*. Allerdings greift die CPU nicht direkt auf den Speicher zu, sondern indirekt über die *Memory Management Unit* (MMU).

Am Bus hängen weitere Hardwarekomponenten, die die Verbindung zu den externen Geräten herstellen. Dazu gehören verschiedene IO-Controller für die interne und externe Festplattenanbindung, eine Graphikkarte für die Aufbereitung der Bildschirminhalte, und eine oder mehrere Netzwerkkarten für die Verbindungen nach außen, USB-Ports, Funk-Sender und -Empfänger für WLAN und Bluetooth, usw.

Die Synchronisation aller Abläufe im Computer geschieht über einen globalen Taktgeber, die *clock*. Sie schickt über einen separaten Bus regelmäßige *Taktsignale*, heute meist im Gigaherz Bereich, nach denen sich alle Komponenten richten müssen.

Speicheradressierung
Hauptspeicher und Cache sind aufgeteilt in Folgen von *Wörtern*. Jedes Wort besteht aus einer festen Anzahl von Bits bzw. Bytes. Typische Wortgrößen sind 32 Bits (4 Bytes) oder 64 Bits (8 Bytes). I.A. ist ein Wort so groß, dass alle Bits in dem Wort in einem Takt verarbeitet werden können.

Die Folge von Wörtern ist durchnummeriert. Jedes Wort kann über seine Nummer (Adresse) angeprochen werden. Man kann sich das vorstellen wie die einzelnen Stockwerke eines Hochhauses. Jedes Stockwerk hat eine Nummer.



Da die Adressen, sie sind ja nur Nummern, in Binärdarstellung ebenfalls nur Bitfolgen sind, kann man sie auch selbst in den Speicher schreiben. In Adresse 5 könnte also als Speicherinhalt die Adresse 7 stehen. Diese Möglichkeit wird sehr häufig in Programmen ausgenutzt.

Konkrete Prozessoren adressieren allerdings meist nicht wortweise, sondern feinergranular, byteweise. Bei einer Wortgröße von 32 Bits könnte man damit im ersten Wort die Bytes 0,1,2 und 3 separat ansprechen. Das zweite Wort würde dann bei Adresse 4 starten. Durch diese Möglichkeit kann man z.B. in ASCII-Texten, wo jeder Buchstabe durch ein Byte dargestellt ist, jeden einzelnen Buchstaben separat ansprechen.

Die Möglichkeit, Adressen auch als Daten im Speicher ablegen zu können, bedeutet, dass die Anzahl von Adressbits höchsten so groß sein kann, wie die Wortgröße. Bei einer Wortgröße von 32 Bits hätte man also auch 32 Adressbits zur Verfügung. Mit 32 Bits hat man insgesamt $2^{32} = 4.294.967.296$ Nummern zur Verfügung. Bei byteweiser Adressierung entspräche das einer Arbeitsspeichergröße von ca. 4 Gigabytes, was für heutige Computer noch ordentlich, aber nicht übermäßig groß ist.

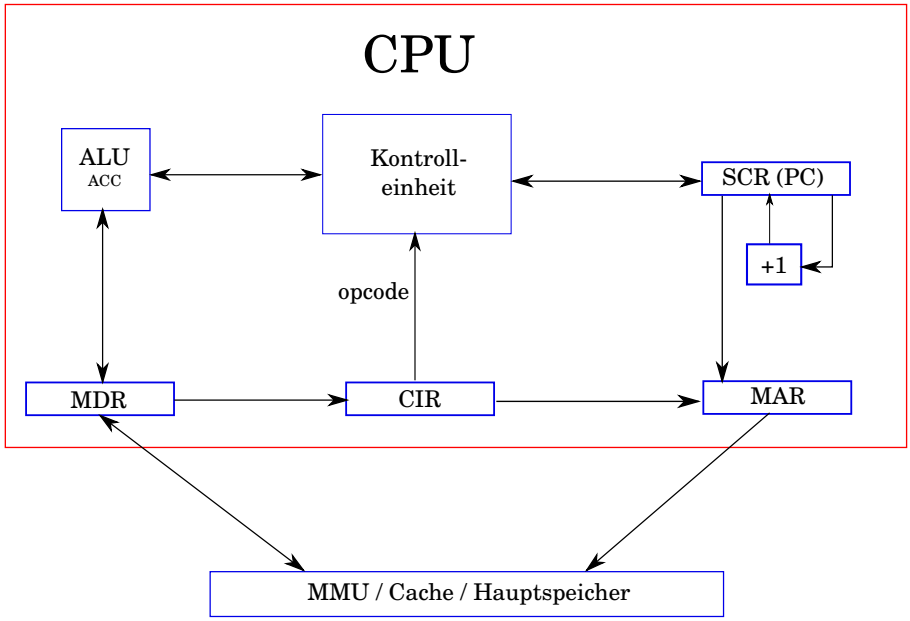
Mit 64 Bits hat man insgesamt $2^{64} = 18.446.744.073.709.551.616$ Nummern zur Verfügung. Bei byteweiser Adressierung entspräche das einer Speichergröße von ca. 18 Exabytes. Heute schon in Planung befindliche Supercomputer könnten in diese Bereiche kommen.

Maschinensprache

Eine weitere wichtige Erkenntnis, die John von Neumann in die Architektur hat einfließen lassen, bestand darin, dass man *Programm und Daten nicht trennen muss*. Auch Programme kann man als Bitfolgen repräsentieren, und diese genauso im Hauptspeicher halten wie die Daten. Dazu braucht man eine binäre Sprache, mit der man Anweisungen an die CPU ausdrücken kann, die sog. *Maschinensprache*. Ein sog. *Maschinenbefehl* besteht dann immer aus der binären Nummer des jeweiligen Befehls, dem *OP-Code*, gefolgt von binär dargestellten Argumenten. Z.B. könnte der Addierbefehl die Nummer 5 haben, und die Argumente enthalten dann die Adressen der Zahlen, welche zu addieren sind. Alles das lässt sich als Bitfolge repräsentieren, so dass es keinen strukturellen Unterschied mehr gibt zwischen Programm und Daten.

3 Die CPU

Die Bestandteile der CPU selbst in folgendem Bild, wiederum vereinfachend und schematisch, dargestellt:



ALU: (Arithmetic Logic Unit)
Dies ist die eigentliche Recheneinheit. Sie hat meist eigene Speicher, sog. *Register* und führt arithmetisch / logische Operationen mit den Werten in diesen Registern durch. Davon gibt es mindestens eines, den sog. *Akkumulator*.

Kontrolleinheit: Sie analysiert die Maschinenbefehle und steuert damit die Abläufe innerhalb der CPU. Wenn es sich z.B. um einen arithmetischen Befehl handelt, gibt sie der ALU die Anweisung, diesen auszuführen.

SCR (PC): (Sequence Control Register, Program Counter (PC) oder Programmzähler)
ist ein Register, welches die Adresse des nächsten auszuführenden Befehls enthält. Bei normaler Programmausführung, ohne Sprungbefehle, wird dieser Zähler nach jedem Programmschritt um 1 weiter gezählt[1].

CIR (Current Instruction Register)
enthält den Maschinencode des gerade auszuführenden Befehls.

MAR: (Memory Address Register)
In dieses Register wird die Adresse einer Speicherzelle geschrieben, von der die Daten geholt werden sollen. Das kann entweder die Adresse des nächsten Maschinenbefehls sein, oder auch die Adresse des nächsten Datenwortes, welches verarbeitet werden muss. Der Inhalt des MAR wird über den Bus an die Memory Management Unit geschickt, welche dann die Daten aus dem Cache oder Hauptspeicher holt.

[1] Falls die Adressierung byteweise ist, muss um 4 bzw. 8 weitergezählt werden.

MDR (Memory Data Register oder auch Memory Buffer Register)
Sobald ein Datenwort über den Bus aus dem Speicher kommt, wird es im MDR zwischengespeichert. Von dort kommt es entweder in das CIR, falls es sich um einen Maschinenbefehl handelt, oder in die ALU, falls es sich um ein Argument für eine Berechnungen handelt.

3.1 Der Fetch-Execute Zyklus

Die Verarbeitung eines Maschinenbefehls geschieht in folgenden Schritten:

1. Zunächst ist die Adresse des Befehls im Programmzähler (SCR/PC).
2. Von dort wird sie in das Memory Address Register (MAR) kopiert. Das hat zur Folge, dass die Adresse über den Bus an die MMU geschickt wird.
3. Der Inhalt der Adresse, d.h. der Maschinenbefehl selbst, kommt über den Bus in das Memory Data Register (MDR)
4. Da der CPU in diesem Moment bekannt ist, dass es sich um einen Maschinenbefehl handeln muss, wird er in das Current Instruction Register (CIR) kopiert.
5. Jetzt kann die Kontrolleinheit den Befehl analysieren, und abhängig davon die passenden Aktionen einleiten:
   - Falls es sich um einen arithmetisch/logischen Befehl handelt, dessen Argumente, bzw. deren Adressen ebenfalls im Maschinenbefehl enthalten sind, wird die ALU aktiviert, und über die Sequenz MAR $\rightarrow$ MMU $\rightarrow$ Speicher $\rightarrow$ MDR $\rightarrow$ Akkumulator werden die Daten bereit gestellt.

     Sobald die ALU mit der Berechnung fertig ist, wird das Ergebnis in das MDR geschrieben, und die Adresse des Zielwortes, in das die Daten kopiert werden sollen, in das MAR. Beide gelangen über den Bus und die MMU in den Speicher.
   - Falls es sich um einen Sprungbefehl handelt, wird die Adresse des Sprungziels unmittelbar in das Sequence Control Register (SCR) geschrieben.
6. Falls der letzte Befehl kein Sprungbefehl war, wird das Sequence Control Register automatisch weiter gezählt, um den nächsten Maschinenbefehl laden zu können.

In konkreten Prozessoren sind noch andere Abfolgen als den oben geschilderten möglich. Insbesondere haben die meisten ALUs nicht nur ein Register, den Akkumulator, sondern bis zu 32 interne Register. Daher ist es oft nicht nötig, die Berechnungsergebnisse zurück in den Speicher zu schreiben. Sie können in einem der Register zwischengespeichert werden, um sofort für anschließende Berechnungen bereit zu stehen. Das beschleunigt die Abarbeitung ungemein.

4 Optimierungen der Hardware

Die Prozessorhardware ist schon lange nicht mehr so, wie sie in den Anfängen entwickelt wurde. Technische Neuerungen haben nicht nur einfach Geschwindigkeitsverbesserungen gebracht, sondern auch massive Umstrukturierungen nötig gemacht. Eine davon sind die Cache Speicher.

4.1 Cache Speicher

In den ersten Computern war die CPU langsam und der Speicherzugriff schnell. Das hat sich irgendwann umgekehrt. Seit ca. Mitte der 90er Jahre sind die CPUs schnell, aber die Busse und Speicher langsam. D.h. die CPU könnte mehr verarbeiten als sie Daten geliefert bekommt.

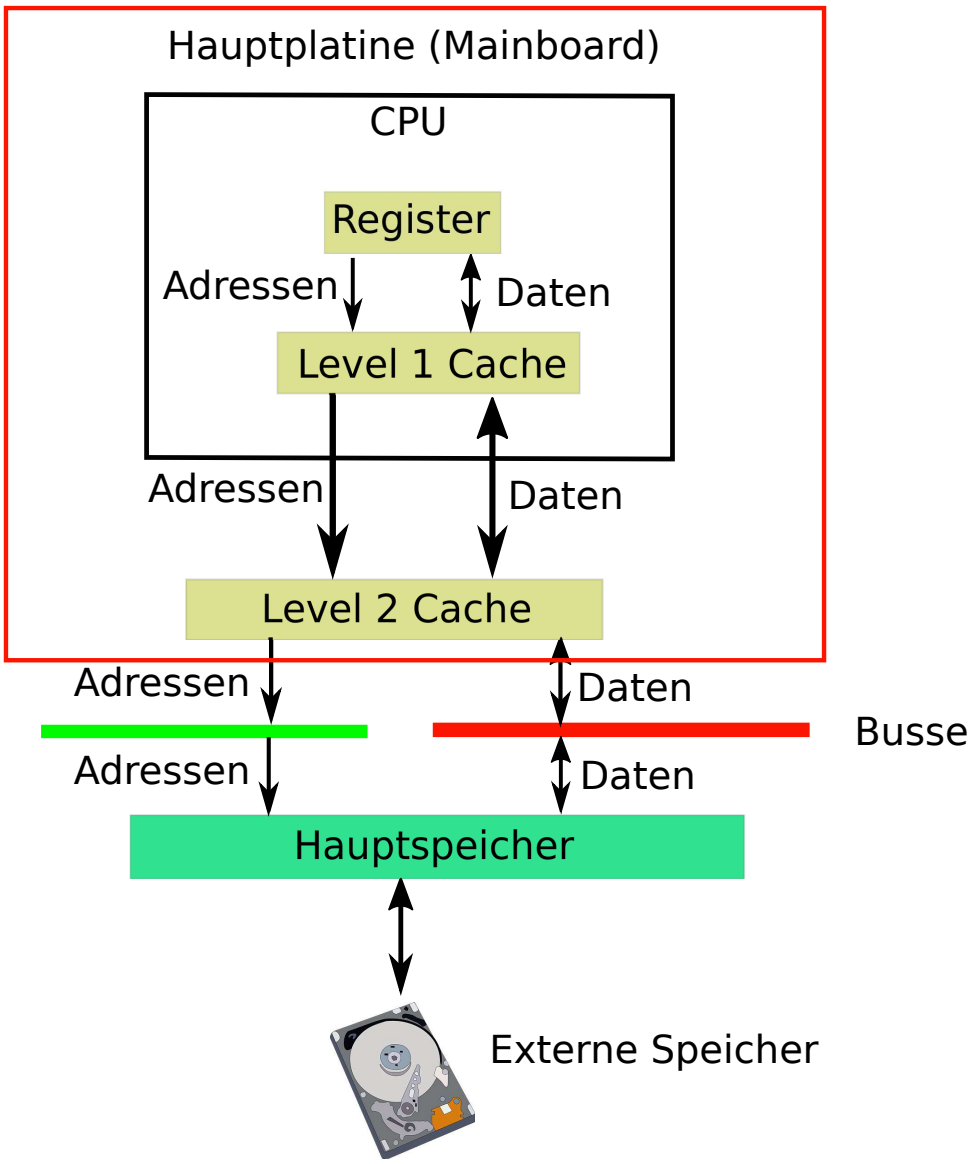
Man hat auf verschiedene Arten versucht, dem Problem entgegenzuwirken:

- Schnellere und mehr Busse, z.B. ein direkter Bus zwischen CPU und Hauptspeicher.
- Breitere Busse, 64 Bits parallel statt 32 Bits parallel. Das begrenzt aber die Geschwindigkeit, da man alle 64 Leitungen synchron halten muss.
- Schnelle Zwischenspeicher (Caches) in der Nähe der CPU.

Der letzte Ansatz war der erfolgreichste. Zwischen Hauptspeicher und CPU werden kleinere Zwischenspeicher installiert, die häufig benötigte Daten bereit halten und einen schnellen Zugriff erlauben.

Der schnellste Zwischenspeicher sind die Prozessorregister selbst mit einer Zugriffszeit von ca. 1 Nanosekunde. Entweder noch im Prozessor oder direkt daneben auf der Hauptplatine befindet sich der Prozessor Cache als statischer RAM (SRAM) Speicher mit Zugriffszeiten von einigen Nanosekunden. Der Arbeitsspeicher (Hauptspeicher) selbst mit dynamischen RAM (DRAM) Technik hat dagegen Zugriffszeiten von 60-70 Nanosekunden. D.h. während die CPU auf Daten aus dem Arbeitsspeicher wartet, könnte sie mehrere Duzend Anweisungen ausführen.

In vielen Computern gibt es sogar mehrere Ebenen (Levels) von Cache Speicher. Man hat dann eine ganze Speicherhierarchie.



Bei jedem Speicherzugriff muss also zuerst im Cache nachgeschaut werden, ob die Daten schon dort sind. Wenn nicht, werden sie sowohl in die Register als auch in den Cache geladen. Falls dabei festgestellt wird, dass der Cache schon voll ist, muss man Platz machen, d.h. andere Daten rauswerfen. Die Entscheidung, für welche Daten es sich lohnt, sie längerfristig im Cache zu lassen, wird nach *Lokalitätsprinzipien* getroffen:

Zeitliche Lokalität: Ein einmal gebrauchtes Wort aus dem Arbeitsspeicher wird wahrscheinlich öfter wieder gebraucht (in Schleifen, z.B.)

Räumliche Lokalität: Zu einem gebrauchten Wort aus dem Arbeitsspeicher werden auch häufiger die Nachbarworte gebraucht. Insbesondere bei sequentielle Programmabarbeitung oder bei Arrayzugriffen ist das oft so. Daher werden u.U. mit einem angeforderten Wort auch vorsorglich mehrere Wörter aus seiner Nachbarschaft in den Cache Speicher geladen.

Das Prinzip der räumlichen Lokalität kann konkrete Auswirkungen auf die Programmierung haben. In vielen Anwendungen braucht man zweidimensionale Matrizen. Diese können nur linear abgespei-

chert werden, entweder zeilenweise oder spaltenweise. Wenn sie zeilenweise abgespeichert werden, dann bewirkt ein Zugriff auf ein Element in einer Zeile, dass u.U. gleich die ganze Zeile in den Cache Speicher geladen wird. Arbeitet ein Algorithmus also die Matrix zeilenweise durch, dann profitiert er direkt von dem Cache Mechanismus. Arbeitet er aber die Matrix spaltenweise durch, dann wird mit jedem weiteren Zugriff auf das nächste Element in der Spalte dessen ganze Zeile in den Cache Speicher geladen, obwohl sie überhaupt nicht gebraucht wird. Bei großen Matrizen kann der Zeitverlust dadurch enorm sein.

4.2 Pipelining, Superskalarität

Die bisher vorgestellte Arbeitsweise im Fetch-Execure Zyklus hat den großen Nachteil, dass fast alle Komponenten der CPU die meiste Zeit arbeitslos sind. Als Beispiel betrachten wir den Maschinenbefehl

add t3, t1, t2

mit der Bedeutung: Addiere die Daten an Adresse t1 zu den Daten an Adresse t2 und speichere das Ergebnis in Adresse t3.

Um so einen Befehl auszuführen, müssen folgende Schritte gemacht werden:

1.	Lade Daten aus Adresse t1	Die ALU tut nichts
2.	Lade Daten aus Adresse t2	Die ALU tut nichts
3.	addiere	Die Kontrolleinheit tut nichts
4.	Speichere das Ergebnis in t3	Die ALU tut wieder nichts.

Man kennt das Phänomen auch aus dem Alltag, z.B. beim Wäsche waschen:

1.	Waschen in der Waschmaschine	der Trockener tut nichts, der Bügler tut nichts
2.	Trocknen im Trockner	die Waschmaschine tut nichts, der Bügler tut nichts
3.	bügeln	die Waschmaschine tut nichts, der Trockner tut nichts
4.	Wäsche einräumen	Waschmaschine, Trockner und Bügler tun nichts.

Bei nur einer Ladung Wäsche macht das nichts aus. Niemand würde aber mehrere Ladungen Wäsche nacheinander auf diese Weise bearbeiten. Stattdessen arbeitet man *versetzt*: Sobald die Waschmaschine mit der ersten Ladung Wäsche fertig ist, wird sie in den Trockner getan, und sofort kommt die zweite Ladung Wäsche in die Waschmaschine.

Dieses Prinzip des *Pipelinings* hat man auch in den Prozessoren verwirklicht. Dazu werden die Arbeitsschritte in der CPU in möglichst kleine Teilschritte aufgeteilt, so dass eine ganze Befehlsfolge versetzt durch diese Teilschritte laufen kann, und damit alle Teile der Hardware permanent beschäftigt werden können.

Zur Illustration betrachten wir folgende Aufspaltung der Instruktionsausführung in mehrere Stufen

S1: Instruktion laden
S2: Instruktion dekodieren
S3: Argumente bereitstellen
S4: Instruktion ausführen
S5: Ergebnis abspeichern

Eine *sequentielle Ausführung* von zwei Instruktionen wäre folgendermaßen:

$$\underbrace{\texttt{S1 S2 S3 S4 S5}}_{\text{1. Instuktion}}\ \underbrace{\texttt{S1 S2 S3 S4 S5}}_{\text{2. Instuktion}}$$

Dagegen ist die versetzte Ausführung mittels *Pipelining* viel schneller:

```
S1  S2  S3  S4  S5  S1  S2  S3  S4  S5 ..
    S1  S2  S3  S4  S5  S1  S2  S3  S4  S5 ..
        S1  S2  S3  S4  S5  S1  S2  S3  S4  S5 ..
            S1  S2  S3  S4  S5  S1  S2  S3  S4  S5 ..
                S1  S2  S3  S4  S5  S1  S2  S3  S4  S5 ..
```

Bei 5 Stufen ist die maximale Beschleunigung ein Faktor 5!

Der Gewinn ist meist nicht ganz so groß, da beim Pipelining sich die Zeit für die Ausführung einer einzelnen Stufe nach der *langsamsten* Stufe richtet.

Beispiel: Sequentielle Ausführung einer add Anweisung (die Zeiten sind erfunden)

Instruktion holen	5 ms
Dekodieren	1 ms
Argumente bereitstellen	1 ms
addieren	2 ms
Ergebnis in Register schreiben	1 ms

Die Gesamtzeit für eine Ausführung ist 10 ms.
Die Gesamtzeit für 1000 Ausführungen ist 10000 ms.

Mit Pipelining richten sich die einzelnen Zeiten nach dem langsamsten Schritt:

Instruktion holen	5 ms
Dekodieren	5 ms
Argumente bereitstellen	5 ms
addieren	5 ms
Ergebnis in Register schreiben	5 ms

Die Gesamtzeit für eine Ausführung ist dann 25 ms.
Die Gesamtzeit für 1000 versetzte Ausführungen ist aber 5020 ms.
Dazu kommt noch der zusätzliche Verwaltungsaufwand für die Pipelining Verwaltung (ca. 10%).

Es ist natürlich ärgerlich, wenn sich die Ausführungszeit der einzelnen Schritte nach der langsamsten Hardware richten muss. Beim Wäsche waschen könnte es z.B. sein, dass der Trockner doppelt so

lange braucht wie die Waschmaschine. Was macht man dagegen? Man kauft einen zweiten Trockner, und lässt beide parallel arbeiten.

Genau das hat man auch in der Prozessorarchitektur gemacht. Man hat analysiert, welche Komponente am langsamsten ist, und hat diese einfach verdoppelt oder vervielfacht. Wenn z.B. die ALU am langsamsten ist, baut man zwei ALUs ein. Der Fachausdruck dafür ist *Superskalare Architektur*.

4.2.1 Probleme beim Pipelining

Pipelining funktioniert gut wenn die Instruktionen genau in der Reihenfolge im Arbeitsspeicher stehen, in der sie auch abgearbeitet werden müssen, und wenn die Instruktionen unabhängig voneinander sind.

Es gibt aber zwei Situationen, wo dies nicht so schön funktioniert:

Abhängigkeit der Instruktionen voneinander:

Als Beispiel betrachten wir zwei aufeinanderfolgende Additionen:
add t3 t1 t2
add t4 t3 t2
Es soll also erst t1 + t2 gerechnet werden, und das Ergebnis dann in t3 geschrieben werden. Danach soll t3 + t2 ausgerechnet werden. Um die zweite Anweisung auszuführen braucht man das Ergebnis der ersten Anweisung.

Gegenmaßnahmen: (beschränkt anwendbar)

Umordnung der Programmschritte:

Manchmal kann der Compiler erkennen, dass er ohne Gefahr andere Instruktionen dazwischenschalten kann, um Zeit zu gewinnen.

Beispiel:

Original:	umgeordnet:
x = 3;	y = a + b;
y = a + b;	x = 3;
z = 2*y;	z = 2*y;

Abkürzungen in der ALU:

Manchmal geht es, ein Ergebnis nicht erst in ein Register zu schreiben, und dann wieder aus dem Register zu laden, sondern das Ergebnis direkt in der ALU weiterzuverarbeiten.

Bei einer Befehlskette

Laden	Dekodieren	Daten holen	ALU	Daten schreiben	
	Laden	Dekodieren	Daten holen	ALU	Daten schreiben

wo die Daten von der ALU in ein Register geschrieben werden, und dann vom Register wieder in die ALU zurückgeholt werden, kann man als Abkürzung die Daten einfach in der ALU lassen.

Sprungbefehle:

Pipelining baut darauf, dass während der Abarbeitung von Befehl 1 schon Befehl 2 geholt und in die Pipeline gegeben werden. Leider gibt es aber viele Programme mit Sprüngen. Die sequentielle Abarbeitung der Programmschritte wird dabei verletzt. Der nachfolgende Befehl

nach dem Sprungbefehl ist nicht der unmittelbar folgende Befehl, der schon in die Pipeline geladen ist, sondern ein ganz anderer. Daher müsste man in diesem Moment die Pipeline stoppen, den falschen Befehl löschen und den richtigen laden. Das unterbricht die Pipeline und verzögert die Abarbeitung.

Gegenmaßnahmen: (ebenfalls beschränkt anwendbar):

Sprungvorhersage:
Statische Sprungvorhersage: Sprünge rückwarts sind häufiger als Sprünge vorwärts (vom Ende einer Schleife an den Anfang).

Dynamische Sprungvorhersage: Statistiken über Sprungverhalten während der Laufzeit werden erstellt (Sehr hoher zusätzlicher Aufwand).

Spekulative Ausführung: Man führt eine ganze Befehlskette auf Verdacht aus. Falls es die falsche Entscheidung war, muss der alte Zustand wiederhergestellt werden. Dazu braucht man weitere interne Register, um sich den alten Zustand zu merken.

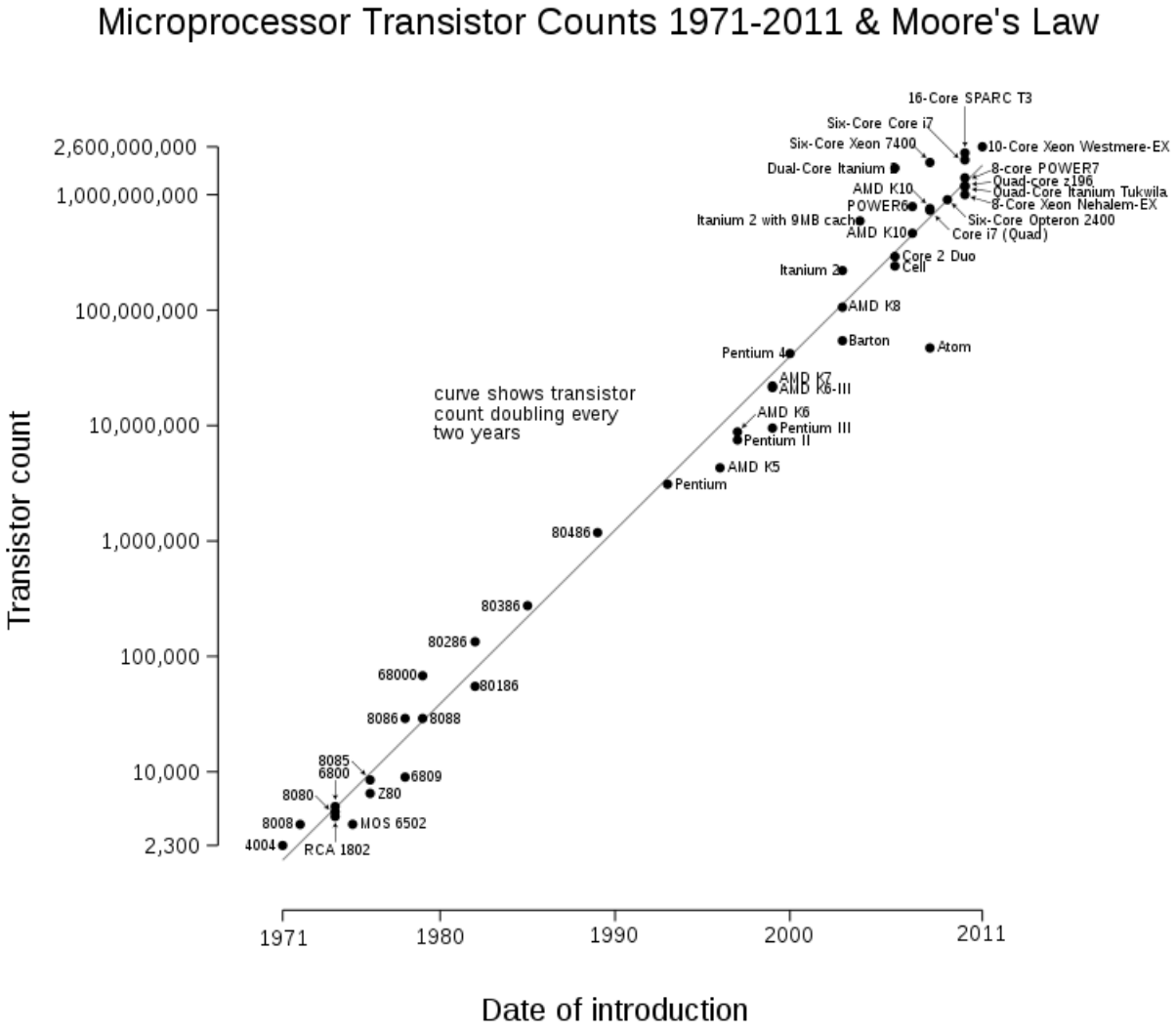
All diese Maßnahmen wirken aber nicht immer. Daher kann es vorkommen, dass die Pipeline angehalten werden muss, bis eine Berechnung fertig ist.

Für den Programmierer bedeutet das, Sprünge im Programm möglichst zu vermeiden, sowie Folgen von Anweisungen, die Daten erzeugen und gleich wiederverwenden, auseinanderzuziehen.

4.3 Hyper-Threading und Mehrkernrechner

Die Taktfrequenzen von Computern haben lange Zeit rasant zugenommen, von ca. 1 Herz beim Maniac Anfang der 50er Jahre, bis zu 3 Gigaherz beim Intel Core i3 in 2007. Seither stagniert die Entwicklung der Taktfrequenzen. Das liegt im wesentlichen daran, dass für höhere Taktfrequenzen auch mehr Energie benötigt wird. Der Energieverbrauch spielt aber eine immer größere Rolle.

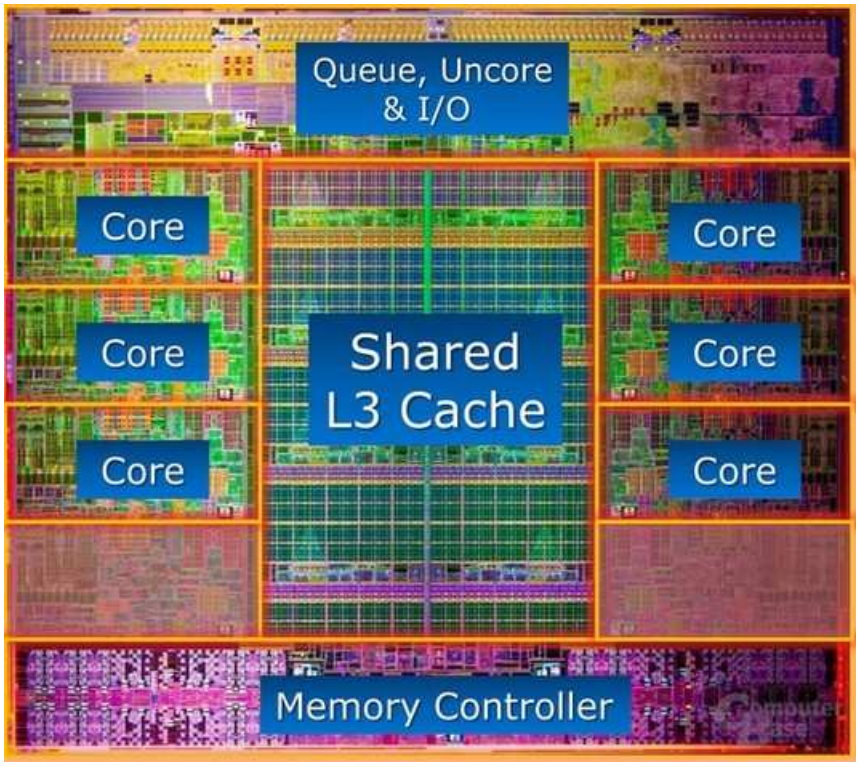
Eine andere Entwicklung hat aber bisher angehalten. Schon 1965 hat Gordon Moore, Mitbegründer von Intel, prognostiziert, dass sich die Anzahl von Schaltelementen auf einem Prozessorchip alle 2 Jahre verdoppelt. Das ist als *Mooresches Gesetz* in die Geschichte eingegangen.



Man kann daher zwar die Prozessoren nicht schneller machen, aber man kann immer mehr Schaltkreise auf einen Chip packen, um z.B. gleich mehrere CPUs auf einem Chip unterzubringen. In einem ersten Schritt hat man nicht alle Komponenten der CPU verdoppelt, sondern nur solche, die häufig gebraucht werden. In der sog. Hyper-Threading Architektur sieht das aber von außen so aus, als ob auf einem Chip zwei getrennte CPUs wären.

Im nächsten Schritt hat man dann in der Tat mehrere Rechenkerne auf einem Chip untergebracht. Der neuste Prozessor von Intel hat inzwischen 18 Rechenkerne, von denen jeder nochmal 2 CPUs simuliert, so dass man 36 Prozesse/Threads parallel darauf ausführen kann.

Auf dem Bild des etwas älteren Intel i7-Prozessors sieht man deutlich die 6 verschiedenen Rechenkerne:



Programmierung
Die Programmierung solcher Mehrkern-CPUs wird insbesondere in Java durch das dort verfügbare Konzept der *Threads* sehr gut unterstützt. Threads sind parallele Durchläufe durch ein Programm, wobei jeder Thread einen eigenen Programmzähler hat, und damit einen Teil des Programms unabhängig von den anderen Threads durchlaufen kann. Das funktioniert problemlos, solange diese Threads nicht miteinander kommunizieren müssen. Sobald eine Kommunikation, z.B. über gemeinsam genutzte globale Variablen, nötig ist, muss der Programmierer Vorkehrungen treffen, damit die Threads sich nicht gegenseitig die Variablen unkontrolliert überschreiben.

Noch heimtückischer kann es werden, wenn die Cache-Speicher genutzt werden. Es kann passieren dass sich verschiedene Threads unterschiedliche Kopien *der gleichen Zelle* des Arbeitsspeichers in den Cache laden. Sie können dann ihre eigene Kopie im Cache verändern, ohne dass es die anderen Threads mitbekommen. Mit entsprechenden Markierungen im Programm kann man solche Situationen verhindern. (In Java ist das z.B. die Markierung `volatile`).

4.4 Vektorprozessoren

Bei vielen naturwissenschaftlichen Anwendungen kommt es häufig vor, dass man eine Operation nacheinander auf vielen Daten durchführen muss. Z.B. zur Berechnung des Skalarprodukts zweier Vektoren muss man deren Elemente nacheinander laden, multiplizieren und zu dem Akkumulator dazu addieren. In der bisher geschilderten Architektur müssen dazu nicht nur die Daten geladen werden, sondern für jedes Paar von Vektorelementen auch die immer gleichen Maschineninstruktionen zur Multiplikation und Addition. Eigentlich würde es ja genügen, die beiden Anfangsadressen der Vektoren, sowie deren Länge zu landen, und dann einen Maschinenbefehl zur Vektoraddition ausführen zu

lassen.

Genau dies hat man im ersten Supercomputer von Cray, der Cray I realisiert. Seither unterscheidet man zwischen *Skalarprozessoren* und *Vektorprozessoren*. Für die Programmierung von Vektorprozessoren braucht man aber spezielle Kommandos in der Programmiersprache. Die gibt es z.B. für die Programmiersprache Fortran.

Vektorprozessoren findet man vorwiegend in Supercomputern für wissenschaftliche Anwendungen. Aber auch die Erzeugung von aufwendigen Graphiken, gerade für Computerspiele, kann von Vektoroperationen profitieren. Daher sind Graphikprozessoren sehr ähnlich wie Vektorprozessoren aufgebaut.

Die Verallgemeinerung von Vektorprozessoren sind *Array-Prozessoren*. Diese haben i.A. sehr viele ALUs, die parallel auf verschiedenen Daten arbeiten. Es gibt aber nur eine Kontrolleinheit, die alle ALUs gleich steuert. Wenn man sehr viele Daten exakt auf gleiche Weise bearbeiten muss, sind Array-Prozessoren eine geeignete Wahl.

4.5 Klassifikation (nach Flynn)

Die unterschiedlichen Architekturen hat man folgendermaßen klassifiziert:

SISD (Single Instruction, Single Data) sind all gängigen PCs und Server

SIMD (Single Instruction, Multiple Data) sind Vektorrechner und Array-Prozessoren

MISD (Multiple Instruction, Single Data) ist ziemlich unbrauchbar

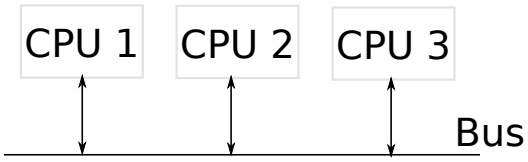
MIMD (Multiple Instruction, Multiple Data) sind Mehrprozessorsysteme oder Mehrrechnersysteme, die über ein Kommunikationsnetz zusammengeschaltet sind.

4.6 Mehrrechnersysteme

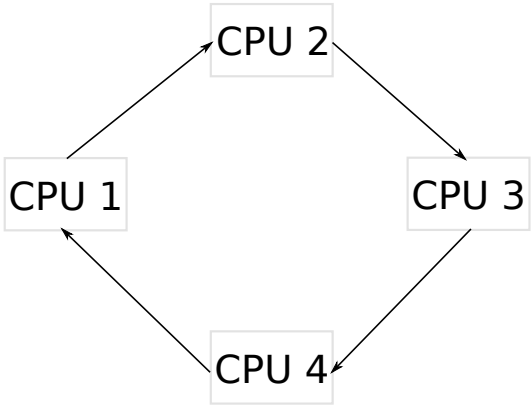
Sobald wirklich viel Rechenleistung gebraucht wird, kommt man mit einem Prozessorchip nicht mehr aus. Dann muss man viele davon zu einem Prozessornetzwerk zusammenschalten. Seit 2016 ist der Rekordhalter, d.h. der schnellste Supercomputer der Welt der chinesische *Sunway TaihuLight*. Er besteht aus 40960 64-Bit Prozessoren, von denen jeder 256 Rechenkerne hat. Damit hat man insgesamt 10,649,600 Rechenkerne. Damit schafft er 93 Petaflops. Das sind $93 \cdot 10^{15}$ Floating Point Operationen pro Sekunde. Allerdings verbraucht er dafür 15 Megawatt Leistung.

Die vielen Prozessoren müssen alle miteinander verbunden werden, damit sie auch miteinander kommunizieren können. Dafür hat man verschiedene *Topologien* entwickelt:

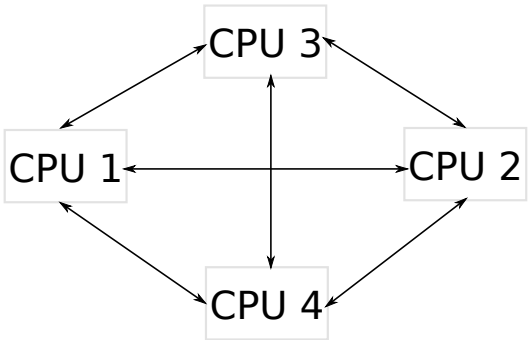
Bus: Wenige Computer, die auch wenig miteinander kommunizieren, verbindet man über das Ethernet als Bus miteinander. Das Problem dabei ist, dass immer nur zwei Computer gleichzeitig Daten austauschen können. Während sie das tun, müssen alle anderen warten.



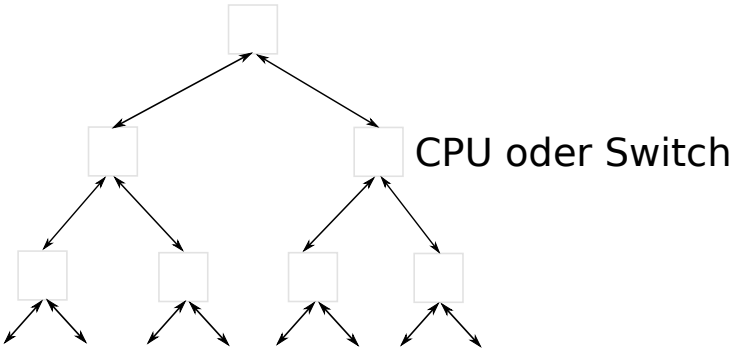
Token Ring: Wenn man den Bus schließt, bekommt man einen Ring. In diesem laufen sog. Token herum, die die Daten transportieren. Das kann man sich vorstellen wie ein Zug mit mehreren Waggons, in die man Daten hinein laden kann. Der Zug läuft immer im Ring herum. Wenn eine CPU etwas schicken will, wartet sie bis ein leerer Waggon vorbei kommt, und lädt ihre Daten hinein. Die Ziel-CPU liest alle vorbeikommenden Waggons mit für sie bestimmten Daten aus. Sobald eine CPU ausfällt, ist allerdings der ganze Token Ring blockiert.



Kompletter Zusammenschluss: Hier verbindet man jeden Prozessor mit jedem anderen. Damit erreicht man natürlich die bestmögliche Kommunikation dazwischen. Da man dafür quadratisch viele Leitungen braucht ist das leider nur praktikabel wenn man wenige CPUs hat.

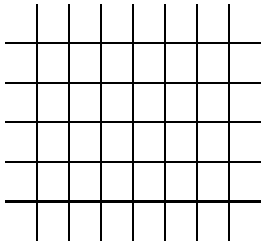


Baum: Für Rechenaufgaben, die man hierarchisch in unabhängige Rechnungen aufgliedern kann, eignet sich eine baumartige Topologie:



Ein Switch ist eine Art Weiche, die die Verbindung in unterschiedliche Wege dirigieren kann.

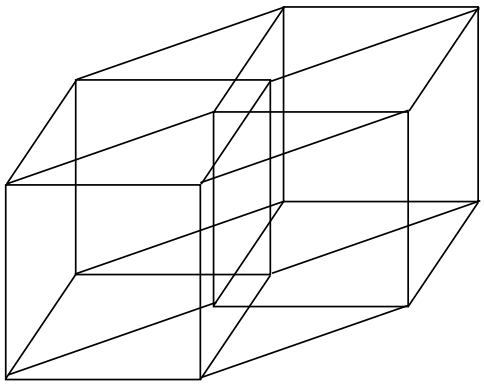
Gitter: Für sehr viele CPUs bei sparsamer Verkabelung eignet sich eine gitterförmig Topologie. Hierbei sitzen die CPUs in den Knotenpunkten der Gitter. Neben ihren Rechenaufgaben müssen sie aber auch dafür sorgen, dass ankommende Daten, die nicht für sie bestimmt sind, richtig weitergeleitet werden.



Crossbar-Switches: Dies ist auch eine gitterförmige Topologie. Aber die CPUs sitzen nicht an den Knoten des Gitters, sondern an den Enden der Gitterlinien. An den Knoten sitzen nur Switches, die die ankommenden Daten richtig weiterleiten. Diese Topologie eignet sich für sehr große Mengen an CPUs.

Würfel: Die CPUs sitzen an den Eckpunkten eines Würfels. Das reicht für 8 CPUs.

Hypercube: Die CPUs sitzen an den Eckpunkten eines 4-dimensionalen Hyperwürfels.
Eine zweidimensionale Projektion davon würde etwa so aussehen:



Das reicht dann für 16 CPUs.

5 Ausblick

Die Entwicklung der Prozessoren, zumindest für den normalen Anwender, nimmt inzwischen eine andere Richtung als „immer schneller“, „immer mehr Speicher“. Gerade durch die weite Verbreitung der mobilen Geräte steht der *Energieverbrauch* und die Hitzeentwicklung im Vordergrund. Dies begrenzt insbesondere die Taktfrequenz, aber auch die Menge an Schaltkreisen auf einem Chip. Der neuste Intel Prozessor mit 16 Kernen, z.B., verbraucht bis zu 165 Watt. Er eignet sich daher nicht für Notebooks. Auch bei Supercomputern spielt der Energieverbrauch inzwischen eine sehr große Rolle. Man misst dabei wieviel GigaFlops/Watt erreicht werden. Im Juni 2017 war ein japanischen Supercomputer, der TSUBAME3.0 mit 14.110 GigaFlops/Watt der führende Supercomputer in der Green500 Liste, der Liste der 500 effizientesten Supercomputern.

Stichwortverzeichnis