Holger Kreissl
TU-Chemnitz

Rechnerarchitektur und Betriebssysteme

Die wichtigsten Themen kurz zusammengefasst.

Holger Kreissl
Student der angewandten Informatik
TU-Chemnitz

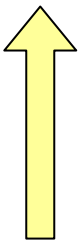
Bei Fehlerfindung bitte Mail an hokr@informatik.tu-chemnitz.de ☺

Holger Kreissl
TU-Chemnitz

Introduction to this paper

Dieses Blatt soll einen kleinen Überblick über die wichtigsten grundlegenden architektonischen Merkmale von modernen Rechnern und deren Schnittstellen zu Betriebssystemen aufzeigen. Es soll nur ein Überblick sein und ein Anhaltspunkt für eventuelle Vertiefungen. Ich werde versuchen den Zusammenhang zwischen Rechnerarchitektur und Betriebssystemmaschine klarer zu machen. Das wichtigste ist für mich aber eine Klassifikation der strukturellen Ebenen und Techniken.

Klassifikation der Ebenen



Anwendungsebene *(Anwendersoftware)*
Assemblerebene *(Beschreibung von Algorithmen, Link & Bind)*
Betriebssystem *(Speichermanagment, Prozesskommunikation)*
Instruction Set Architecture *(ISA,Adressierungsarten)*
Microarchitektur *(Risc,Cisc,Branch Prediction..)*
Logische Ebene *(Register,Schieber, Latches..)*
Transistorebene *(Transistoren, MOS)*

Mikroarchitekturebene

Möglichkeiten zur Geschwindigkeitserhöhung

1. Verringerung Taktzyklen pro Instruktion
2. Organisation vereinfachen um Taktzyklen zu verkürzen
3. Ausführungen von Instruktionen überlappen lassen (Pineline's)

Ausführungszeit = CPI * 1 / Taktfrequenz * Instructions per Task

Man kann die Optimierung in Bezug auf Implementierung (schnellere Speicher u.ä.) und in Bezug auf Architektur (Risc etc.) sehen. Es gibt die verschiedensten Techniken zur Verbesserung der CPU Leistung, wie Caches, **Sprungvorhersage**, Ausführung **Out Of Order** mit **Register-Renaming** und **spekulative Ausführung**. Der neuste Schrei sind die Superskalaren Prozessoren mit Superpipelines (z.B. Itanium mit 20 Stufen).

Caches

Caches sind ultrawichtig und versuchen den von-Neumannschen Flaschenhals zu minimieren. Caches sind abhängig von zwei Arten von Adresslokalitäten.

Räumliche Lokalität

Es ist wahrscheinlich, das auf eine kürzlich zugegriffene, numerisch vergleichbare Speicherzelle, erneut Zugegriffen wird. Das wird ausgenutzt, in dem mehr Daten in den Cache aufgenommen werden, als benötigt werden.

Zeitliche Lokalität

Diese Lokalität findet statt, wenn auf Zellen auf die zugegriffen wurde, erneut zugegriffen wird. (Meist Instruktionen von Schleifen) Diese Daten werden von Cache Ersetzungsalgorithmen genutzt, um ihre Entscheidungen zu treffen.

Cachearten

Direct Mapped Cache

Bestehen aus **Valid, Tag und Data** Teil. Problem ist hier, das ein Cache der 64 KByte aufnehmen kann, alle Vielfachen von 64 in die gleiche Cache Line schreibt. Ein RAM-Eintrag kann bei dieser Form nur auf einen bestimmten Cache-Eintrag abgebildet werden. Sozusagen ist diese Form das Gegenteil zum Vollassoziativen Cache. Zur Berechnung der Cachezeile werden meist simple **Modulo** Berechnungen genutzt.

Vollassoziativer Cache

Jeder Hauptspeichereintrag der Größe m, der an einer Adresse beginnt, welche ohne Rest durch seine Eintragslänge teilbar ist, kann auf jede Stelle im Cache abgebildet werden. Um einen Eintrag im Cache wieder zu finden, wird im TAG Bereich die komplette Adresse des Eintrages gespeichert. Um einen Eintrag im Cache zu suchen, werden durch eine komplexe Schaltung alle TAGs gleichzeitig mit dem Adress-Tag der Hauptspeicheradresse verglichen.

Teilassoziativer Cache (n-Wege Satz assoziativ)

Der Nachteil des **Direct Mapped** Cache's wird hier durch eine **n-Wege** Technik gelöst. Es werden jeder Cachezeile N Cache-Einträge zugewiesen. Dieses Verfahren ist aufwendiger, da die N Spalten der Line gleichzeitig mit der Adresse verglichen werden müssen. Es wird **LRU** genutzt, falls eine Zeile voll ist und eine neue **Cacheline** eingebracht werden soll.
Die wichtigsten Lese-/Schreinstrategien sind **Write Throung** (Schreiben auch in Speicher), **Write Back** (wenn Cache Line verworfen wird, wird zurückgeschrieben) und **Write Allocate** (bei Miss in Cache).

Sprungvorhersage

Extrem wichtig für Pipelining und Superskalarität, um **stalls** und Verzögerungen zu minimieren. Bei statischer Vorhersage werden Rückwärtssprünge meist erst durchgeführt und Vorwärtssprünge nicht. Wurde ein Sprung falsch vorhergesagt, muss die angefangene Instruktion rückgängig gemacht werden, was aufwendig ist. Deshalb gibt es ausgeklügelte Verfahren für die Branch Prediction…

Statische Sprungvorhersage

Es werden Compiler benutzt, welche **spezielle Sprungbefehle** mitführen, welche ein Bit für die Sprungvorhersage enthalten. Da der Compiler ja weiß, wie oft eine Schleife durchlaufen wird, ist das sehr effizient. Dies muss aber architektonisch von der Hardware unterstützt werden. Des Weiteren ist kein Speicher für die **History Table** notwendig, was es kostengünstiger macht. Statische Verfahren erreichen eine Trefferrate von 65 bis 85%, was für moderne CPU's mit Superpipelines zu wenig ist. Dynamische Verfahren erreichen Trefferraten bei der Vorhersage von 98% und mehr!

Dynamische Sprungvorhersage

Es gibt zwei grundlegende Methoden. BHT und BTB. Die **Branch History Table (Branch Predicion Buffer)** ist ein Cache, in der alle bedingten Sprünge protokolliert werden. (bis zu mehereren Tausend) Einfachste Version enthält ein **Valid-Bit (Branch taken oder nicht)**, welches durch den niederwertigen Teil der Sprungadresse adressiert wird. Kompliziere Implementationen arbeiten nach dem n-Wege Prinzip. Durch Second Chance kann dieses Verfahren noch verbessert werden.

Der **Branch Target Buffer** speichert nicht nur die taken-Bits, sondern auch die Sprungzieladresse, um null Verluste bei wiederholtem Aufruf zu haben. Das setzt voraus, dass nur taken branches aufgenommen werden. Bei einem Hit in der BTB kann somit während der Fetch Phase der Program Counter überschrieben werden. Werden keine History Bits mitgeführt spricht man vom BTAB.

Second Chance

Nach Beenden einer Schleife wird ein Sprung logischerweise falsch vorhergesagt. Um zu vermeiden, dass nun fälschlicherweise das **Sprungbit** falsch gesetzt wird (da ja die gleiche Schleife noch mal durchlaufen werden kann), ändert man dieses erst nach der zweiten falschen Vorhersage. Leicht zu implementieren als Finite State Machine mit vier Zuständen. Nachteil der dynamischen Vorhersage ist die notwendige teuere und komplexere Hardware.

Pipelining Konzept

Pipelines werden in allen modernen CPUs benutzt. Die UltraSparc2 hat 9 und der P2 12. Der Intel Pentium Itanium weißt eine 20 stufige Superpinepline (pipeline in der sich einzelne stufen überlappen können) auf! Pipes werden heutzutage in Kombination mit der Superskalartechnik verwendet, um höchste Effizienz und Parallelverarbeitung gewährleisten zu können. Die fünf grundlegenden Stufen einer einfachen Pipeline sind **IF,ID,EX,MEM,WB**

Takte T = Befehle + (Pipestufen – 1)

Folgende Abhängigkeiten verhindern, dass die CPI auf eins gehen.

Strucual Hazards bzw. Ressourcenkonflikte

IF und MEM wollen gleichzeitig auf Speicher lesend oder schreibend zugreifen. Geht nicht, außer bei **Dual-Port-RAM**, welche aber schweineteuer sind. Dieses Problem tritt aber bei modernen CPU's kaum noch auf, da eh intern eine **Havard-**ähnliche Architektur mit getrenntem Befehls- und Datencache gearbeitet wird.

Data Hazards bzw. Datenabhängigkeiten

Ein Folgebefehl wartet auf das **Writeback** der darüber liegenden Pipe, da er von diesem Befehl abhängig ist.

Dies kann durch Nops bzw. Stalls ineffizient gelöst werden. Besser der Programmierer oder der Compiler löst diese Abhängigkeiten durch eine clevere Umordnung der Befehlsfolge auf. Es gibt aber noch eine andere Möglichkeit, welche aber hardwareseitig unterstützt werden muss.

Forwarding

Beim Forwarding werden Ergebnisse, sobald sie vorliegen an die nächste Stufe weitergereicht und nicht erst auf das Write Back gewartet. In anderen Worten: Das Ergebnis der **ALU** wird dieser sofort wieder eingespeist.

Control Hazards bzw. Sprungverzögerungen

Sprungergebnisse stehen erst in der Write Back Phase an. Moderne Prozessoren haben aber schon in der Fetch / Decode Einheit eine Logik, welche die Zieladresse des Sprunges berechnet.
Eine andere Möglichkeit ist die der spekulativen Ausführen. Hier tritt aber das Problem auf, dass viel Aufwand bei falscher Spekulation getrieben werden muss.

Speicherbuspinelining im Pentium 2

Das Pipeline Konzept wird nicht nur bei der Befehlsverarbeitung angewendet. Hier als Beispiel die Speicherbuspipe eines P2, welche sich um die Busvergabe kümmert.

1. Bus-Arbitation (Auswahl des Busmasters, welcher Bus zugesprochen bekommt)
2. Request (Adresse auf Bus legen und Anfrage stelle)
3. Error (erlaubt Slave gegebenenfalls Fehlermeldungen an Master zu senden)
4. **Snooping** (für Multiprozessorsysteme)
5. Response (alles kla?)
6. Data (Daten zurücksenden)

Diese sechs Phasen sind vollkommen unabhängig und können somit in einer 6-Stufigen Pipeline abgearbeitet werden.
Beim **Daisy Chaining** kann man Geräten Prioritäten entsprechend ihre Nähe zum **Arbiter** zuweisen. Das nächstliegende Gerät bekommt den Bus, falls es diesen will. Will es den Bus nicht, wird die Abfrage zum nächsten Treiber weitergegeben. Ist eine Methode der Bus-Arbitation, welche Bus-Konflikte bereinigen soll.

Superskalar-CPU's - dynamic scheduled pinelines

Da das Pipeline Konzept schnell an seine Grenzen gelangt ist, ist eine andere Methode notwendig. Es muss möglich sein, dass mehrere Befehle pro Takt beendet werden können und unabhängige Befehle, welche in Pipes wegen vorangehender Abhängigkeiten warten müssen, in der Ausführungsfolge **vorzuziehen**. Superskalare Prozessoren erkennen Parallel verarbeitbare Befehle von selbst und benötigen somit keine speziell optimierenden Compiler, wie sie z.B. VLIW verlangen. Dafür sind sie weitaus komplexer.

Um RAW, WAW und WAR Abhängigkeiten zu vermeiden, wird versucht Instruktionen anders anzuordnen. Dafür ist ein **Scoreboard** notwendig, das für jedes benutzte Register die Leseabhängigkeiten und Schreibabhängigkeiten zählt.

Der Tomasulo Algorithmus ist ein Verfahren, welches sich durchgesetzt hat, da es WAW und WAR Konflikte dynamisch auflösen kann. Dies ist so einfach möglich weil WAW und WAR keine echten Datenabhängigkeiten sind, sondern nur entstehen, wenn ein späterer Befehl ein Register wieder verwendet, ohne die darin enthaltenen Daten zu benötigen. Hier tritt das Register Renaming oder arbeiten auf Schattenregistern in Kraft. Würde es mehr Register geben, könnten diese Abhängigkeiten schon vom Compiler aufgelöst werden.

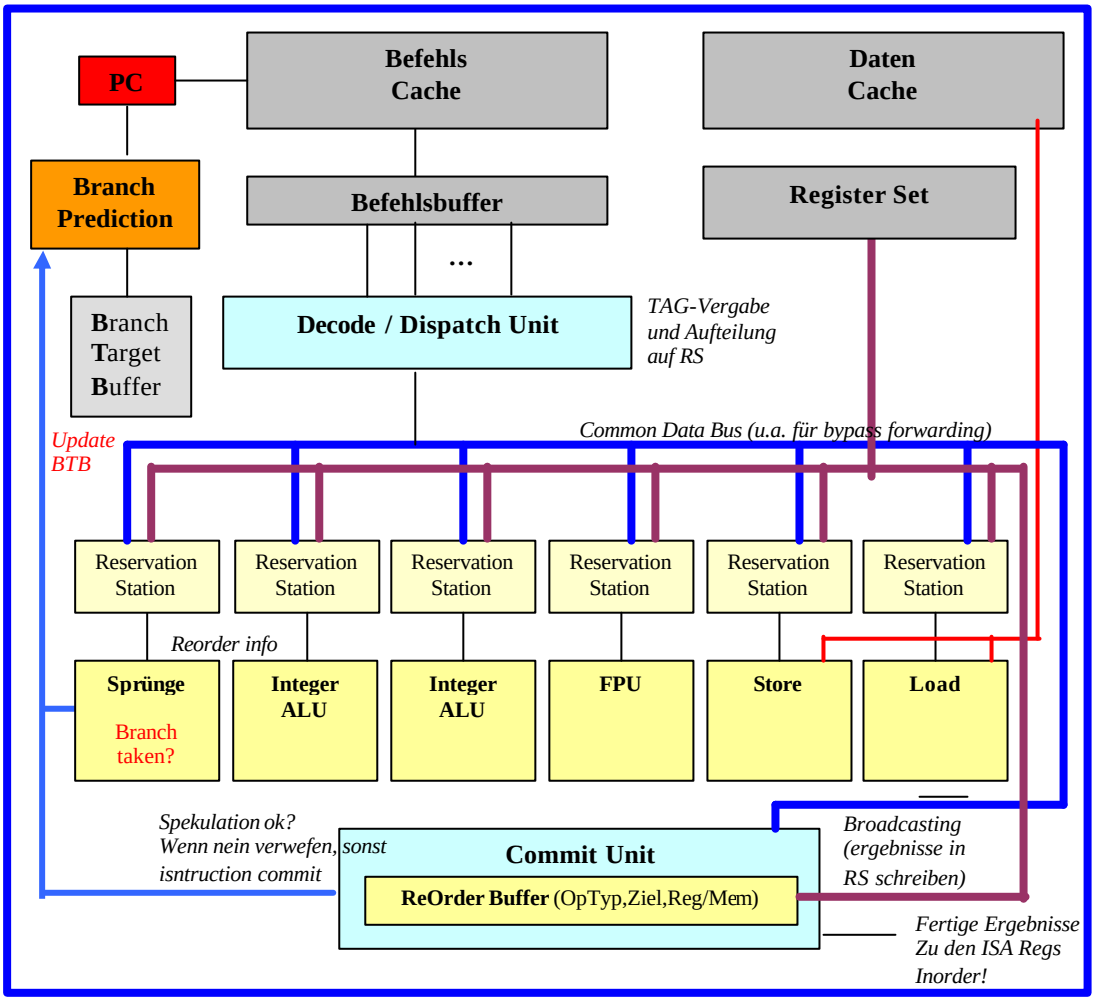
Mit **Register Renaming** werden intern versteckte Register benutzt, um Abhängigkeiten aufzulösen.

Fetch und Decode	Dispatch / Issue	Execution	Commit
Holt Instruktionen in einen Befehlscache. Die **Decode-Unit** holt sich einen Teil der Befehle und versucht mehrere gleichzeitig zu **decodieren** (In-Order) . Dabei wird versucht **Sprünge vorherzusagen**. (BTB-Branch Target Buffer etc) Übergibt dekodierten Befehle in der richtigen Reihenfolge an die Dispatch (**Issue**) Unit	Holt dekodierte Befehle aus Befehlspuffer und übergibt sie InOrder an die **Reservation Stations der Execute Units, sobald alle Operanden verfügbar sind (Issue)**. Solange im **Reorder** Buffer Platz ist, reserviert sie ein Feld für diesen Befehl mit Hilfe des Tags und gibt dieses an die RS weiter. Wenn nicht wartet sie, bis ein Platz frei wird. (**Dispatch** Phase)	Führt Befehle auf **Schattenregistern** aus, um Data Hazards zu meiden. Nun werden Befehle in den Reorder-Buffer geschrieben und **Out-Of-Order** ausgeführt, solange es keine RAW-Konflikte gibt. Nach Beenden eines Befehls wird Ergebnis an alle RS gebroadcastet, so dass wartende Befehle fortfahren können. (**Write result**)	Commit/Completion oder Retire Einheit schreibt die Ergebnisse aus den Renaming Registern in die echten **ISA Register** zurück, nachdem sie geprüft hat, ob abhängige vorangehende Befehle ihre Ergebnisse geliefert haben und keine falsche Sprungvorhersage eingetreten war.

Spekulative Ausführung

Wird benutzt um weitere Lücken im Ablauf zu füllen. Hierbei werden Sprungziel-Befehle gegebenenfalls schon ausgeführt, wenn noch gar nicht klar ist, ob dieser überhaupt wahr werden. Dies wird meist mit dynamischen Scheduling nach Tomasulo kombiniert.
Falsch vorhergesagte Befehle erreichen nie die Commit-Phase und werden verworfen. Dies ist möglich, da sich die Ereignisschreibphase in **Bypassing** (Zwischenspeicherung im Reorder-Buffer) und dem **Rückschreiben** auf die echten ISA Register aufteilen lässt.

Pentium Pro und Power PC – abstrahierter Risc Kern



Beim Pentium heißen die Phasen Fetch, Decode, Dispatch, Execute und Retire

Unterschied Scoreboard – Tomasulo

Scoreboards wurden in den 70ern verwendet. Sie benötigen weniger Hardwareaufwand. Haben aber auch Nachteile gegenüber dem Tomasulo.
Scoreboards haben einen zentralen Controller, welcher die advanced Pipe überwacht. Konflikte werden überwacht und falls welche auftreten werden Stalls eingefügt. Der Tomasulo dagegen basiert auf einer verteilten Steuerung und vergibt ein 4-Bit TAG (enthält z.B. die Nummer der RS und der Execute-Unit) für jeden Befehl und löst dann die Konflikte dynamisch auf. Dabei verwendet er den Common Data Bus für Broadcasting, Forwarding bzw. Bypassing zu den Reservation Stations.

Pentium 2

Instruktionen werden in eine oder mehrere Mikrooperationen dekodiert und in eine Warteschlange gestellt. Sprungerkennung erfolgt erst statisch und dann dynamisch mit 4 History Bits. Dann werden Register im **Reorder-Buffer** (der P4 hat Platz für 40 µOps) zugewiesen. Dabei werden um Konflikte zu vermeiden **Renaming-Register** (Schattenregister) eingesetzt.

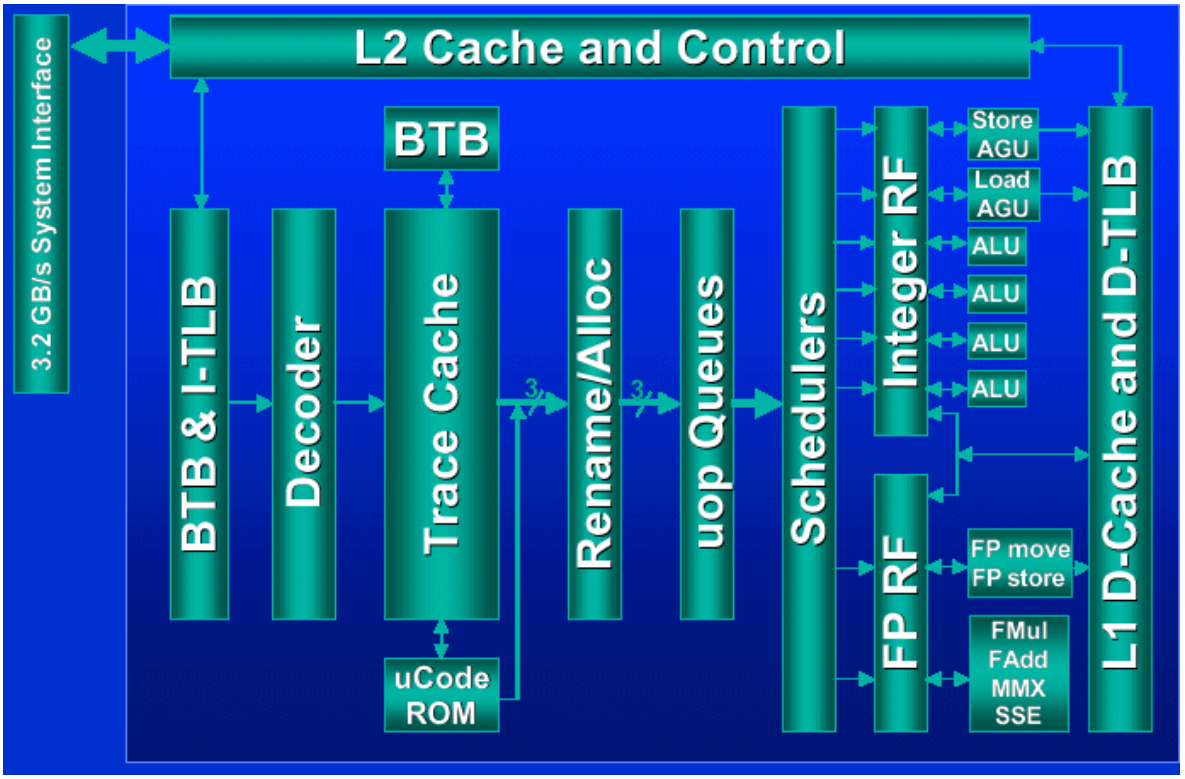
In der Execute Phase werden Mikrooperationen aus dem Reorderbuffer entnommen und spekulativ ausgeführt. Es wird ein komplexes **Scoreboard** verwendet, um aktive Operationen und Register zu verwalten. Falls mehrere Mikrooperationen bereit sind, wählt ein Algorithmus den nächst Wichtigsten aus (z.B. Sprünge).

Mikrooperationen deren Operanden alle verfügbar sind, werden in die **Reservation Station** geschrieben, welche 20 solcher Einträge aufnehmen kann. Dort warten die Ops, bis eine entsprechende Ausführungseinheit frei wird. Die **Execute-Units** haben fünf Ports, um Operationen auszuführen. Manche Ausführungseinheiten teilen sich einen Port (FPU,MMX).

Die **Retire (Completion)** Einheit sendet fertige Ergebnisse an die richtige Stelle. Der P2 unterstützt speculative execution. Falls begonnene Instruktionen nicht mehr benötigt werden, wird die Rollback Fähigkeit genutzt.

Der Pentium 4 nutzt einen Trace Cache für die letzten drei dekodierten Befehle. Im Hit Falle, bedient sich die Execution Unit direkt dieser Befehle, was vor allem bei Schleifen einen beträchtlichen Speedup bringt.

Die **UlraSparc** 2 ist dagegen eine reine Risc Maschine mit vierfachen statt 2facher Superskalarität. Durch das Risc Prinzip müssen hier Instruktionen nicht in mehrere Mikrooperationen umgesetzt werden. Des weiteren unterstützt die meist gleich große Befehlslänge das Pipelining besser, als die z.T. komplexeren Cisc Befehle des Pentium. Neu ist das so genannte **Folding** der **PicoJava** CPU, welche Instruktionen faltet, d.h. zusammenfasst und gleiche Instruktionen in einem Zyklus (auf verschiedenen Registern) abarbeiten kann. (5 x schneller als P2, obwohl auch CISC)



Quelle: tecchannel.de

Pentium 4 Core Pipeline

Risc / Cisc Unterschied

Vorteile von Risc

- Einfachere Befehlssätze mit ca. 40-80 Befehlstypen
- Einfachere Steuerung durch die Hardware ohne Mikroprogramme
- Effizientere Pipelines durch gleichlange, eintaktige Stufen
- Befehle können meist in einem Takt ausgeführt werden
- Datenzugriffe nur durch Load und Store um Speicherzugriffe zu vermeiden
- Mehr Register und Optimierung des Befehlssatzes durch Compiler

Typische Riscsysteme haben eine hartverdrahtete Steuereinheit und somit keinen Mikrocodespeicher.

Der Pentium ist ein „**hybrid**"-System mit RISC Kern. Dabei werden komplexe CISC Befehle durch ein Mikroprogramm in RISC zerlegt und im Kern ausgeführt. Die einfachen Befehle werden direkt im RISC Kern in einem einzigen Datenzyklus ausgeführt.

Alle wichtigen elementaren (Risc) Befehle werden direkt von Level 0 (der Hardware) ausgeführt und somit nicht via **Mikroinstruktionen** interpretiert. Dies ist ein Vorteil von reinen RISC Systemen, welche diese Interpretationsebene zwischen Hardware und ISA (Instruction Set Architecture) Ebene nicht durchlaufen müssen.

Mikroinstruktionen steuern den Datenweg für einen Zyklus. Sie enthält alle notwendigen Bit-Belegungen für ALU, MEM, Register… etc. um einen Zyklus zu abarbeiten lassen zu können. Die Adresse der nächsten Mikroinstruktion wird ebenso mit codiert, wie die Art und Weise des Aufrufes. Die Mikroinstruktionen werden in einem **Steuerspeicher** gehalten, welcher das jeweilige **Mikroprogramm** enthält.

Der Steuerspeicher muss die Mikroinstruktionen nicht in geordneter oder sequentieller Folge enthalten, wie es beim Hauptspeicher der Fall ist. Es kann jede Instruktion einen Verweis auf die Nächste enthalten. Sprünge sind einfach möglich. Angewandt wird dies in Form von Opcodes, welche nichts anderes als Adressen auf Mikroinstruktionen im Steuerspeicher sind.

Parallelität

- auf der ISA-Ebene
  - Pipelining (Befehlszyklen aber auch Speicherbus-Pinelining)
  - Superskalarität, compined superscalar superpipeline

- Auf Prozessor-Ebene
  - Matrizenrechner (jeder Prozessor besitzt eigenen Speicher)
  - Vektorrechner
  - Mehrprozessorsysteme mit gemeinsamen und getrennten Speicher
  - Mehrrechnersysteme

Für Parallelität ist von Neumann Architektur im reinen Maße nicht anwendbar. Die Harvard Architektur kommt z.B. im Kern des Pentium vor. Nämlich gibt es dort einen getrennten Cache für Daten und Instruktionen, um Parallelität beim Lesen gewährleisten zu können.

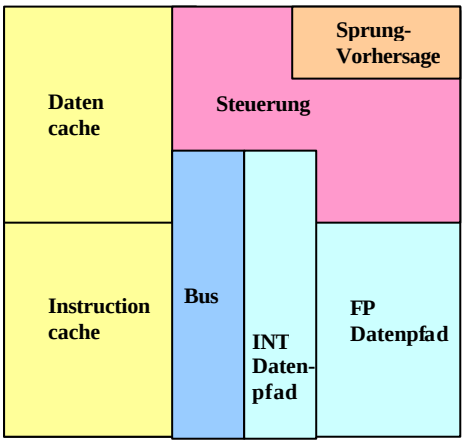
VLIW

Der IA-64-Befehlssatz von Intel wird auf „**Very Long Instruction Words**“ basieren. Es werden drei Instruktionen ein einen fetten 128 Bit Befehl gepackt. Hier besteht nun die Möglichkeit, explizit festzulegen, welche Befehle parallel abgearbeitet werden sollen bzw. können. So eröffnen sich völlig neue Optimierungsmöglichkeiten im Compilerbau. Man nennt dieses Prinzip **EPIC**. Der Transmeta verwendet auch VLIW. Hier muss aber der Compiler nicht die Optimierungen vornehmen, da der Transmeta um der Core eine Morphing Softwareebene hat, welche die Aufteilung in parallel abarbeitbare Befehle ausführt.
Dadurch, daß nun explizit gesagt wird, welche Instruktionen parallel ausführbar sind, ist nun nicht mehr so viel Chipfläche zum Auflösen von Hazards notwendig und kann z.B. für mehr Register verwendet werden.

Prozessoraufteilungen

Die fünf klassischen Bestandteile der CPU sind Eingabe, Ausgabe, Speicher, Datenpfad und einer Steuerung.

Intel Pentium Chipflächenaufteilung



Der Pentium hat keinen externen Cache. Deshalb nimmt auch ein drittel der Fläche der Cache ein. Der Pentium Pro dagegen hat einen externen Cache und somit mehr Platz auf der Chipfläche.

Pentium Pro Chipfläche:

I/O Einheit, Daten und Befehlscache, Memorybuffer, Integer und FPU Datenpfad, Steuerung bestehend aus Reservation Station und Reorder Buffer, Befehlsdekodiereinheit und Microcode (z.B. unter anderem das Mikroprogramm für die Control Unit (Fetch, Decode, Execute)

ISA Ebene

Diese Ebene verbindet Mikroarchitekturebene und Betriebssystem. Sie stellt den Befehlssatz zur Verfügung und definiert somit implizit die Befehlsformate. Unterschiedliche Prozessoren mit gleicher ISA lassen Programme identisch ablaufen.
Außerdem werden Adressierungsarten (unmittelbar, direkt, Reg, Reg indirekt, Indiziert, Basisindiziert, Stack (picoJava)) Interrupt Handling u.s.w. auf dieser Ebene definiert.

Es werden wichtige grundlegende Einteilungen getroffen. Z.B. die Zugriffsberechtigungen von Registern und Instruktionen für Kernelmode und Usermode.
Es müssen Traps ausgelöst werden, wenn Fehler auftreten und es wird z.B. bei Intel zwischen Real, Virual und **Protected Mode** unterschieden.

Betriebssystem Ebene

Ist wie die ISA Ebene abstrakt zu sehen. Hier werden Virueller Speicher, Pages und mehr verwaltet. Das OS ist somit eine Art Interpreter für diese virtuellen Architekturmerkmale. Das BS abstrahiert Hardware, verwaltet BM, Steuert den Betriebsablauf, Protokolliert und bietet Schutz. Es wird Grundsätzlich in Einprogramm / Mehrprogramm und in Single und Multiuser BS unterschieden. Modellarten für BS sind das Monolithische (jeder darf alles), das Geschichtete (Schichten-Schalenmodell), Virtuelle Maschinen und Client- Server Modelle. Es logische und physikaltische Betriebsmittel, welche vom BS verwaltet und verteilt werden. **BM=(id, adr, wb, functions)**

Virtueller Speicher

Um mehr Speicher bereitzustellen werden statt echter physikalischer Adressen virtuelle benutzt, welche durch die **MMU** in reale Adressen bei Benutzung umgewandelt werden. Durch das **Paging** können Seiten aus- oder eingelagert (nach einem **Page Fault**) werden. Eine **Page Table** referenziert virtuelle auf physikalische Adressen. **Swapping** ist in Reinform sehr langsam. Sinnvollerweise werden **Segmente** in **Seiten** geteilt, welche durch **Paging** ein oder ausgelagert werden. (**Protected Mode**)
Die **MMU** kann, muss aber nicht auf der CPU liegen. Jede Seitentabelleneintrag hat ein **Present/Absent Bit**, welches Auskunft darüber gibt, ob eine Seite sich im Speicher befindet oder nicht. Des Weiteren vermerkt eine Art **Dirty Bit**, ob eine Seite im Speicher geändert wurde, um entscheiden zu können, ob ein zurück schreiben notwendig wird.
Die momentan verwendeten Seiten eines Programmablaufes wird **Working Set** genannt. **Demand Paging** bedeutet, dass Seiten erst dann abgefordert werden, wenn sie benötigt werden.
Eine virtuelle Adresse besteht aus **Seitenummer** und **Offset**. Bei einem **Contextswitch** wird nach der entsprechenden Seitennummer in der Page Table gesucht und die zugehörige physikalische Adresse errechnet. Es wird meist ein n-stufiges Paging angewandt, um die Suche mach den Seiten zu beschleunigen (unix). Bei i386 gibt es ein so genanntes Seitenverzeichnis (DIR) mit 1024 Zeilen, welche wiederum je auf eine Seitentabelle verweisen. Somit enthält eine lineare Adresse beim Intel DIR,PAGE und OFFSET Teil.

Seitenersetzung

NRU benötigt in der Seitentabelle je 2 Bits pro Seite zur Kennzeichnung des Status. Eins für **M**odifiziert und eins für **R**eferenziert. Bei einem Seitenfehler durchsucht das BS die vier implizierten Klassen. (von Klasse 0 wie **nicht refeferenziert und nicht modifiziert bis referenziert und modifiziert** (Klasse 3). NRU wählt nun zufällig eine Seite aus der niedrigsten Klasse zur Ersetzung aus.
LRU ist ein beliebtes Prinzip. Hier gibt es aber wie bei allen anderen Verfahren Probleme, wenn Arbeitsspeicher kleiner als Working Set ist. Dann tritt im Worst Case immer ein Page Fault auf. LRU setzt Hardware voraus. Mit LFU (Last Frequently Used) simuliert man LRU auf Softwarebasis.
Das **FiFo** Prinzip für Seitenersetzung entfernt die jeweils zu letzt geladene Seite. Verbessert wird FiFo durch **Second Chance**, wo jedem Eintrag ein Referenziert-Bit mitgegeben wird. FiFo sucht nun nach der ältesten, nichtreferenzierten Seite und lagert diese aus. Falls alle referenziert sind, lagert Fifo logischerweise die älteste im Speicher befindliche Seite aus. Der Uhr-Algorithmus macht das gleiche, unterscheidet sich aber von der Implementierung. Er hat den Vorteil, dass er nicht konstant Seiten in seiner Liste verschiebt, wie das **Second Chance** macht.
Das Problem der **internen Fragmentierung** tritt überall auf wo ein Speicher in logische Teile aufgeteilt wird. Es steht fest, dass bei einer **Seitengröße** von n Bytes immer n/2 Bytes in der letzten Seite eines Programms verschwendet werden. Kleine Seiten haben den Nachteil, das diese eine große Seitentabelle benötigen. Größere Seiten sind sinnvoller in Betracht auf Festplattentransfers. (da die mittlere Zugriffszeit bei ca. 10 ms liegt!)

Segmentierung

Wird genutzt um statt einen linearen Adressraum (wie beim virtuellen Speicher), **mehrere virtuelle Adressräume** nutzen zu können. Segmentierung wurde entworfen, um dynamisch wachsende Tabellen besser Handhaben zu können. Somit ist schafft Segmentierung einen **mehrdimensionalen Adressraum**.

Unter **Swapping** versteht man das komplette Ein- und Auslagern von Prozessen. Das **Swapping** des segmentierten Speichers ist vergleichbar mit dem Demand-Paging des virtuellen Speichers. Nur das Segmente unterschiedlich groß sein können. Aus diesem Grund tritt hier das Problem der **externen Fragmentierung** auf.
Um die externe Fragmentierung zu minimieren, werden die **Löcher** als verkettete Liste im Speicher gehalten. Falls ein Segment geladen werden soll, sucht z.B. **Best Fit**, dass nächst größere Loch unter allen, wo das Segment passen würde. **First Fit** nimmt das Nächste Loch, welches für das Segment groß genug wäre.
Neben dem Swapping kann auch **Paging** zum Auslagern von Segmenten benutzt werden. Hierbei sind die auszulagernden Blöcke gleich groß, da die Segmente in gleich große Seiten eingeteilt werden. Zur Auslagerung wird das bekannte Demand Paging benutzt.
Meist wird eine Kombination aus Segmentierung und Paging angewandt, bei der die Adresse aus zwei Teilen besteht. (**Segmentnummer und Offset** innerhalb des Segments). Segmente werden also in Seiten unterteilt. Zur Leistungsverbesserung werden die zuletzt verwendeten Segment-Seiten-Kombinationen in einem **Assoziativspeicher (TLB)** gehalten. Im Gegensatz zum Paging ist es beim reinen Swapping nicht möglich, Prozesse auszuführen, die alleine schon nicht in den Hauptspeicher passen.

Freispeicherverwaltung des Swappings

Bitmaps unterteilen den Speicher in Allokationseinheiten. Für jede Einheit gibt es ein Bit im Bitmap. **Verknüpfte Listen** bieten Suchmöglichkeiten mit First Fit, Best Fit oder Quick Fit. Quick Fit setzt mehrere Listen bestimmter Lochgößen voraus. Das **Buddy System** verwaltet n Listen von 1,2, 4, 8 bis zur Größe des Speichers. D.h. ein 1 MByte großer Speicher benötigt 21 Listen und hat Initial nur einen einzigen Eintrag in der letzten Liste, der das 1 MByte große Loch beschreibt. Alle anderen Listen sind leer. Speicher wird nun immer in Abhängigkeit von einer Potenz von 2 vergeben, der gerade noch groß genug ist, um die Daten aufzunehmen. Dies wird einfach implementiert, in dem der Große block einfach solange geteilt wird, bis der Datenblock in den Freispeicherblock passt. Das Buddysystem ist zwar schnell, aber impliziert eine starke interne Fragmentation, da ja immer auf Zweierpotenzen gerundet werden muss.

Der virtuelle Speicher des P2

Es gibt eine **LDT** und eine **GDT**. Die Local Descriptor Table enthält die programmeigenen Segmente, wie **Stack-, Code- und Datensegment**. Die Global Descriptor Table enthält dagegen die Systemsegmente samt deren des Betriebssystems, welche erst geladen werden muss.
Wird ein **Selektor** in ein Segmentregister geladen, wird der entsprechende Bezeichner aus der LDT oder GDT geholt und in MMU Registern gespeichert. Ob L oder GDT kann dem Selektor entnommen werden.
Der **Bezeichner oder Deskriptor** besteht aus der Basisadresse, **Größe** des Segmentes, Privilegbits…
Es wird nun eine Adresse über **Selektor + Offset** gebildet. Bei deaktiviertem Paging ist nun diese Adresse die lineare phyiskalische. Ist aber Paging aktiv wird die Adresse als virtuell interpretiert und über die **Seitentabelle** auf den realen Speicher abgebildet.

Sicherheit durch Deskriptoren zur Segmentverwaltung

Jeder Deskriptor in einer Deskriptortabelle ist mehrere Byte breit und enthält Beschreibungsinformationen für ein Segment aus dem linearen Adreßraum. Neben der Segment-Basisadresse *(BASE)* enthält er das LIMIT, das die Segmentgröße angibt. Dabei wird durch ein Granularitätsbit festgelegt, ob das LIMIT direkt als Länge interpretiert wird (Segmentgrößen bis 1 MB) oder mit dem Wert 4096 multipliziert wird und damit Segmentgrößen bis 4 GB unterstützt. Der **Descriptor-Privilege-Level** gibt an, mit welcher Berechtigungsstufe der Zugriff auf das Segment erfolgen muss:

· Level 0: Betriebssystem
· Level 1, 2: Betriebssystemdienste, Treiber, Systemsoftware
· Level 3: Anwendungssoftware

Weitere Informationen im Deskriptor zeigen an, ob auf das Segment lesend, schreibend oder ausführend zugegriffen werden darf und ob es sich um ein System- oder Anwendungssegment handelt. Ein **Present-Bit** gibt an, ob das Segment sich derzeit überhaupt im Hauptspeicher befindet.

Translation Lookaside Buffer

Translation Lookaside Buffer sind Teil der **MMU** bilden virtuelle Seitennummern auf Nummern physikalischer Seitenrahmen ab. Die TLB enthält je die 64 zuletzt genutzten Instruktions- und Datenseitennummern getrennt. (Die komplette Pagetable wird meistens im Hauptspeicher gehalten) Der TLB ist somit ein meist **vollassoziativer Cache im Prozessor**, welcher einen Teil der Seiten der kompletten PageTable enthält.
Im Unterschied zu Cache Misses werden Page Faults nicht von der Hardware, sondern vom Betriebssystem geregelt. Im Falle eines Page Faults tritt ein TRAP ins OS in Kraft. Der laufende Befehl wird unterbrochen, bis die Seite von der Platte gelesen wurde und die Page Table aktualisiert wurde.

Prozesskommunikation und Synchronisation

Semaphore (Init(id), P(id), V(id)) haben zwei unteilbare Operationen **up** und **down**. Ein Semaphor ist ein Counter. Führt ein Semaphor ein Up (P) aus, kann ein anderer wartender Prozess sein down (V) beenden. (falls der Semaphore Null war) Dabei muss die CPU so genannte TSL (**Test and Set Lock**) Instruktionen unterstützen. Solange der Semaphor Null ist, müssen Prozesse welche ein Down ausführen warten, bis ein Prozess seinen Semaphor durch ein Up wieder freigibt. Weitere Kommunikationsmethoden sind **Ereigniszähler** und **Monitore**. **Nachrichten** sind im Gegensatz zu **Pipes** (Bytestrom) typendeklariert und priorisierbar. In Mehrprozessorsystemen kommen noch so genannte **Barriers** zum Einsatz, wo ein Thread gegebenenfalls auf einen anderen Thread wartet.

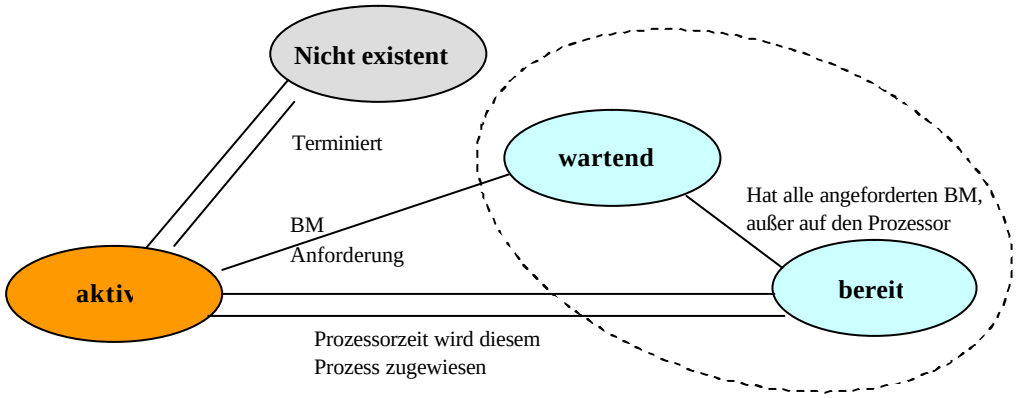
Ein **Mutex** ist eine Ressource die entweder gesperrt oder freigegeben ist. Sie dienen eher zum kurzzeitigen Sperren z.B. einer Variablen, aber nicht zum Synchronisieren von Prozessen.
Im NT gibt es so genannte **Critical Sections**, welche Ähnlichkeiten zum Mutex aufweisen. Jedoch gehören diese CS immer zum Adressraum des betreffenden Threads und können nicht zwischen Prozessen interagieren. **Events** sind **Ereignisse** und sind ein weiteres Synchronisationsmittel.

Zur Kommunikation gibt es auch neben den schon genannten weitere Möglichkeiten. Im **Shared Memory** teilen sich mehrere Prozesse einen jeweiligen Adressraum. Dies wird durch eine Umabbildung der jeweiligen Seite auf die beteiligten Prozesse gemacht. **Sockets** sind Pipes ähnlich, können aber Prozesse auf entfernten Rechnern verbinden.

Typische Kommunikationsprobleme sind das Philosophenproblem (exklusiver Zugriff auf begrenzte BM und folgendes **Verhungern**) und das Leser-Schreiberproblem (falls einer schreibt, dürfen andere nicht lesen).

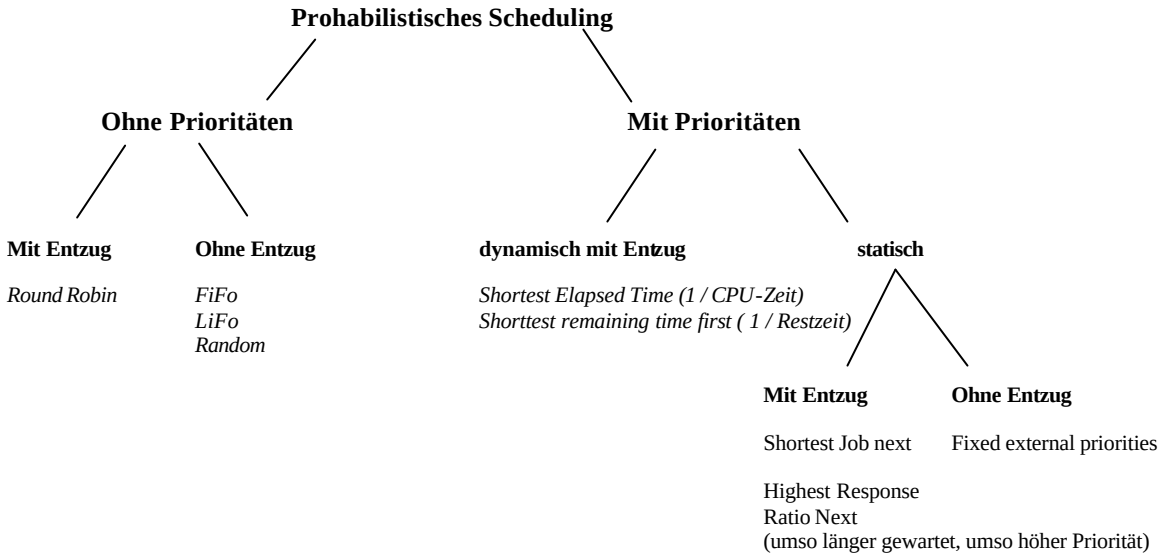
Kommunikation ist Synchronisation und Datenübertragung!

Prozesszustände



Prozessscheduling

Die wichtigsten Kriterien für Scheduling sind **Fairness**, **Effizienz** /CPU Auslastung, **Antwortzeit**, **Verweilzeit** und **Durchsatz**. Beim deterministischen Scheduling müssen alle Zustände schon vorm Ablauf des Prozesses bekannt sein. Drum wird dieses Scheduling nur bei geschlossenen, interaktiven Systemen angewandt.
Für offene, interaktive Systeme ist das **prohabilistische Modell** von Bedeutung, welches nur für Eingeschwungene Zustände gilt. Das geschlossene prohabilistische Modell basiert auf der Warteschlangentheorie (Erwartungswertberechnungen) und hat als typischen Vertreter das Single Server Modell.



Round-Robin

Teilung der Zeit in Quanten. **Quanten** von 100 ms für einen Prozess haben sich durchgesetzt. Viel höhere bringen nicht viel, da Kontextwechsel auch eine gewisse Zeit in Anspruch nehmen. RR ist das fairste Verfahren, da alle Prozesse gleichberechtigt sind und eignet sich somit für Dialogbasierende Systeme.

Prioritätsscheduling

Hat den Nachteil, dass Prozesse mit geringer Priorität nie ausgeführt werden, falls immer Prozesse mit hoher Priorität sich im System befinden. Um Interaktivität zu gewährleisten wurden dann **Prioritätsklassen** eingeführt. Je nach Klasse wurden mehr oder weniger Quanten zugewiesen. Wenn ein Prozess seine Quanten aufgebraucht hat, wird er eine Klasse tiefer gestuft. Z.B. bekommt ein Prozess der 100 Quanten zur Abarbeitung benötigt erst 2, dann 4, 8, 16 … Quanten…

Shortest Job First

Besonders gut für **Stapelaufträge** geeignet und ist nur optimal, wenn alle Aufträge gleichzeitig vorliegen. Dabei müssen die Ausführungszeiten aller Prozesse gemerkt werden. Auf Basis vorhergehender Ausführungszeiten wird meist der Durchschnitt gebildet. Mit Hilfe eines **Alterungsfaktors** wird bestimmt, wie lange diese Zeiten gemerkt werden.

Deadlocks

Eine Menge von Prozessen befindet sich im **Deadlock**, falls jeder Prozess der Menge auf ein Ereignis wartet, das nur ein anderer Prozess der Menge auslösen kann. Da alle Prozesse einer solchen Menge warten, kann keiner jemals ein solches Ereignis auslösen. Somit würden die Prozesse für immer warten. Voraussetzungen sind **exklusiv nutzbare Betriebsmittel** auf einem **Mehrprozesssystem**. Deadlocks können auch auftreten, wenn es gar nicht um BM geht. Werden z.B. Semaphoren falsch programmiert (vertauschte Up und Down Operationen), entstehen auch Deadlocks!

Bedingungen für Deadlocks

Hinreichende Bedingung für Deadlocks ist, das **keine externe BM Instanz** bestehen darf.
Die folgenden **vier notwendigen Bedingungen** müssen alle eintreten. Sonst entsteht kein Deadlock!

1. **Mutual Exclusion** (jedes BM wird von genau einem Prozess belegt oder ist verfügbar)
2. **Belegungs- und Wartebedingung** (ein Prozess kann immer weiter BM anfordern)
3. **Ununterbrechbarkeitsbedingung** (BM könne nicht entzogen werden)
4. **zyklische Wartebedingung** (Zyklus im BM-Graphen)

Bedingung eins wird z.B. beim Druckerspooler aufgehoben, indem ein Ersatzbetriebsmittel bereitgestellt wird. **Livelocks (Verhungern)** entstehen, wenn ein Prozess theoretisch schon das BM bekommen kann, aber es trotzdem nicht zugewiesen bekommt. (siehe Prioritätsscheduling)

Auflösen und Umgehen von Deadlocks

Es gibt vier Grundlegende Verfahren zum Umgang mit Deadlocks.

1. Vogel-Strauß Algorithmus (Ignorieren)
2. Versuchen Deadlocks zu erkennen und zu beheben
3. dynamisches Verhindern (vor BM-Zuteilung prüfen ob DL entstehen würde)
4. konzeptionelles Vermeiden durch Auflösen einer der vier notwendigen Bedingungen

Erkennen kann man DL durch **Tiefensuche** im Betriebsmittelgraphen oder mit der **Vektoren-Matrix Variante**. (BM-Vektor, BM-Restvektor, Belegungs- und Anforderungsmatrix) Dabei muss ein Restmittelvektor einen Forderungsvektor vollständig befriedigen können. Beheben kann man Deadlocks mittels **Unterbrechung** (temporär Entziehen), teilweiser **Wiederholung** (Checkpunkthistory) oder **Prozessabbruch**.

Dynamische Verhinderung ist extrem aufwendig, da vor jeder BM-Zuteilung geprüft werden muss, ob das System noch sicher wäre, falls das BM zugeteilt wird. Unter einem **sicheren System** versteht man, das es eine Folge von BM-Zuteilungen gibt, dass alle aktiven Prozesse terminieren können. Der **Bankier-Algorithmus** ist ein Scheduling Algorithmus, welcher Deadlocks verhindern kann.

Zwei-Phasen-Sperren

Für **Datenbanken** und Nichtechtzeitsysteme gibt es noch einen weiteren sehr einfachen Ansatz. Er beruht auf der Idee, alle Betriebsmittel auf einmal Anzufordern. Sind in **Phase 1** alle BM vorhanden und keine gesperrt, werden alle Betriebsmittel auf einmal gesperrt. Phase zwei beginnt, wenn Phase eins abgeschlossen ist. Ist alle arbeit getan, werden alle BM wieder freigegeben. Falls in der ersten Phase ein Problem auftritt, d.h. ein benötigtes BM schon von einem anderen Prozess gesperrt ist, wird alles Gesperrte wieder freigegeben und das Ganze noch mal von vorn probiert. Aus diesem Grund eignet sich das 2-Phasen-Sperren wohl kaum für Prozesskontroll- oder Echtzeitsysteme. Das Verfahren kann nur angewendet werden, wenn das Programm an jeder Stelle in Phase eins abgebrochen und neu gestartet werden kann.

Assembler Ebene

Diese Ebene ist erstmals eine Ebene, welche nicht von darunter liegenden Ebenen interpretiert wird, sondern übersetzt wird. D.h. Assembler wird in Maschinensprache umgewandelt und kann von unteren Ebenen direkt ausgeführt werden.

In dieser Ebene geht es also um das Kompilieren und Binden von Programmen. Auf darüber liegenden Ebenen sind Anwenderprogramme und Entwicklungsumgebungen, welche diese Ebene nutzen, in dem Sie in einer Hochsprache verfasste Quellcodes, letztendlich auch in Maschinencode wandeln.

Wichtig ist, dass bei Systemen mit Paging und virtuellem Speicher darauf geachtet wird, dass keine absoluten Adressen verwendet werden können. Sonst würden ja nach sämtlichen Aus- und Einlagerungen die Adressen nicht mehr stimmen. Durch dynamisches Binden kann Code mehrfach genutzt werden, in dem mehrere Prozesse die gleichen Bibliotheken im Speicher nutzend. Natürlich nur lesend.

Parallelrechnerarchitekturen

Defininitonsgemäß wird folgendermaßen klassifiziert. Die Korngröße gibt Auskunft über die Art der Parallelität. Bei hohem Kommunikationsaufwand zwischen den Parallel laufenden Prozessen wird von feinkörniger Parallelität gesprochen. Ist wenig Kommunikation notwendig von grobkörniger. Grobkörnige Parallelarchitekturen sind meist lose gekoppelt. D.h. über geringere Bandbreite verbunden, als eng gekoppelte Systeme für feinkörnige Parallelapplikationen.

Modelle

Mehrprozessorsystem oder Shared Memory Sytem

Alle CPU's teilen sich gemeinsamen Speicher. Deshalb ist Kommunikation hier besonders einfach. Ist jede CPU gleichberechtigt und austauschbar, wird das System SMP genannt.

Mehrrechnersystem

Diese Systeme definieren sich dadurch, dass jede CPU ihren eigenen Speicher besitzt. Kommunikation erfolgt hier über ein Verbindungsnetz, was Mehrrechnersysteme weitaus komplizierte macht, als Multiprozessorsysteme.
Eine Möglichkeit des gemeinsamen Speichers ist **Distributed Shared Memory**. Hier stellt das Betriebssystem den systemweiten Adressraum als gemeinsamen Speicher zur Verfügung.
Der Vorteil ist, dass LOAD und STORE Befehle auf fremde Speicher möglich sind. Es wird sozusagen die Seitenersetzung nicht von der Festplatte, sondern vom entfernten Rechner vollzogen.
Kommunikation erfolgt mit Hilfe von Eingangs- und Ausgangspuffern an den jeweiligen Maschinen.

Routing von Paketen

Wie die einzelnen Pakete vom Sender zum Empfänger gelangen, bestimmt das Routing. Es ist ebenso dafür verantwortlich **Deadlocks** zu vermeiden. Dabei sind die Prozesse die Rechner und die Betriebsmittel die Ein- und Ausgabeports.

Leistungsmerkmale

Wichtig ist, dass die n-fache Beschleunigung nur theoretisch erfolgt. Jedes Programm hat sequentielle Bestandteile, welche nicht parallelisierbar sind. Der wichtigste Faktor ist hier die Latenzzeit. Diese muss minimiert werden, um maximale Leistung zu erhalten.

Paradigmen

SPMD, Pineline, Phasenaufteilung, Divide and Conquer, Replicated Worker Principle oder Task Farm (zentrale Arbeitswarteschlange mit Aufgaben)
Es gilt beim zugriff auf gemeinsame Variablen bzw. Speicher das Prinzip des gegenseitigen Anschlusses. Dafür sind die bekannten Methoden anwendbar, wie Semaphoren, Kritische Abschnitte und Mutexe. Eine weitere Synchronisationsmethode stellen Barrieren dar. Hierbei warten alle beteiligten Prozesse warten, bis jeder beteiligte Prozess eine bestimmte Phase abgearbeitet hat.

Klassifizierung nach Flynn

Instruktionsströme	Datenströme	Bezeichner	Typische Anwendungsfälle
1	1	SISD	Von-Neumann Rechner
1	N	SIMD	Vektorrechner
N	1	MISD	(noch) nicht existent
N	N	MIMD	Mehrprozessor und -rechner

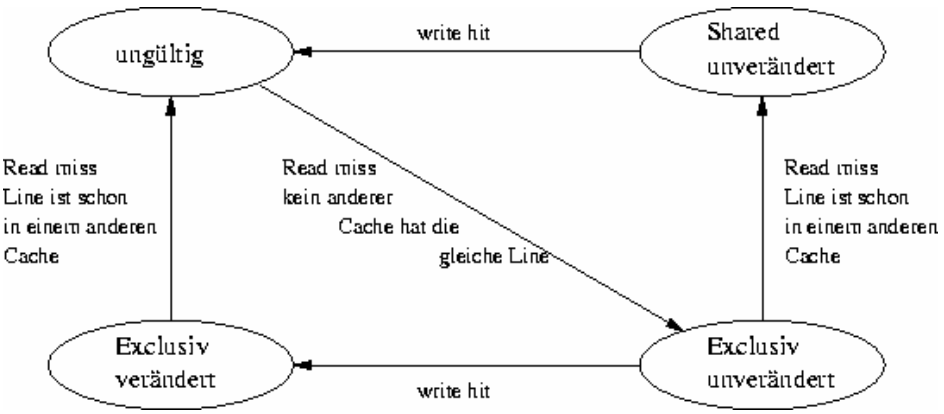
Cache-Kohärenz-Protokolle

Write-Through und Write-Back sind die wichtigsten Methoden auf Singleprozessorsystemem. Write Throung ist das einfachste und Write-Back das effizienteste, welches in Form des MESI-Protokoll z.B. im Pentium angewendet wird. Bei SMP's dominieren **Write Invalidate** (alle anderen Kopien werden ungültig gemacht) und **Write Broadcast** (alle anderen Kopien werden aktualisiert).

MESI-Protokoll für Multiprozessorsysteme

Ein Cache Eintrag kann einen von vier verschiedenen Zuständen besitzen. Treten Zustandswechsel auf, werden alle Prozessoren benachrichtigt.

1. **Invalid** (die Cacheline ist veraltet)
2. **Shared** (mehrere Prozessoren haben die aktuelle Cacheline)
3. **Exklusive** (Prozessor hat die korrekte Cacheline alleine)
4. **Modified** (Die Cacheline ist aktuell, aber nicht mehr mit Speicher konsistent)



Holger Kreissl
TU-Chemnitz

Vektorrechner

Viele numerische Probleme lassen sich auf Skalarprozessoren nur in vielen Schleifendurchlaufen berechnen. Vektorcomputer bieten **high level instructions**, um diese Schleife mit einem Befehl zu berechnen. Pipelines bieten sich mehr als an. Da die Datenelemente unabhängig sind, treten keine Data Hazards auf und das Fehlen von Schleifensprungbefehlen lässt keine Control Hazards in der Pipeline zu. Um die Speicherbandbreite zu erhöhen, verwendet man das Prinzip der **Speicherverschränkung**. Caches haben wenig Effizienz, da Vektorprobleme geringe Lokalität aufweisen. Die Bandbreite wird durch Blockzugriff und zeitlich verschränkten Speicherbankzugriff maximiert. (**memory interleaving und memory banking**)

Interleaved Memory

Da Speicher zu langsam, werden nun mehrere **unabhängige Speicherbänke** realisiert, von denen jeder einen bestimmten Teil des Adressraums zugesprochen bekommt. Beziehen sich aufeinander folgende Speicherzugriffe jeweils auf Adressen verschiedener Speicherbänke, können die Folgezugriffe überlappt werden. Bis 512 Speicherbänke sind durchaus normal, um diese Latenzzeiten der DRAMs zu minimieren. Um Speicherbankkonflikte zu vermeiden, müssen Programmablauf und Speicherimage aufeinander abgestimmt sein. Dabei wird der Vorteil genutzt, dass die Position von aufeinander folgenden Vektorelementen im Speicher nicht sequentiell sein muss.
Wie bei Pipelines wird hier die Forwarding-Ähnliche Technik des **Chainings** angewandt, um Resultatelemente einer Vektoroperation sofort an den Folgebefehl weiterzuleiten, ohne auf das Fertigstellen des Befehls zu warten. Unter Nutzung mehrerer Funktionseinheiten können Vektoroperationen damit quasi parallel ausgeführt werden.

Quellenangabe:

A.S. Tanenbaum	*„Moderne Betriebssysteme“*
Patterson, Hennessy	*„Computer Organisation & Design“*
Christian Märtin	*„Rechnerarchitekturen“*
A.S. Tanenbaum	*„Computerarchitektur“*
Prof. W. Rehm	*„Ergänzendes Skript Rechnerachitektur“*
Www.techchannel.de	*„techchannel“*