	Answers

Von-Neumann-Architektur	- Prozessor
	  - Operation unit
	  - Controll Unit
	- Main Memory
	- I/O devices
	
	- Single-Bus-System
	
	- keine Aufteilung von Daten und Befehlen

Von-Neimann-Flaschenhals	Es kann zum Rückstau im Bussystem kommen, da alles über 1 Bussystem läuft.
Harvard Architektur	Trennung von Datenspeicher und Befehlsspeicher (2 Bussysteme dafür)

Aufbau eines Microcomputers	- Prozessor
	  - Operation unit
	  - Controll Unit
	- Main Memory
	- I/O devices

Aufgabe eines Prozessors	Kontrolliert den Ablauf von Instruktionen. Besteht aus der Controll-Unit und ALU
Aufgabe der Controll-Unit	Interpretiert die geladenen Befehle und erstellt alle Controllbefehle weiterer Einheiten (übersetzt Befehl in microinstructions die der Rechner ausführen kann)
Aufgabe der Arithmetic-Logic-Unit (ALU)	Erledigt alle Befehle, die ihr erteilt werden. Zentrum aller Berechnungen und Logischen Aufgaben
Aufgabe der Mainmemory	Speichert Befehle und Daten in Speicherzellen mit fester größe, die über Adressen angesprochen werden können
Aufgabe des IO	Akzeptiert Inputs/Gibt Outputs (Kommunication des Computers mit der Außenwelt)
Mooreschen Law	Alle 18 Monate verdoppelt sich die Rechenleistung der Computersysteme

Pro/Contra Von-Neumann-Architektur	Pro:
	  - minimale Hardware bedingungen
	  - minimaler Speicherbedarf
	
	Contra:
	  - Von-Neumann-Flaschenhals
	  - Keine trennung von Daten und Befehlen
	  - Kein Schutz vom Speicher (alle programme können überall zugreifen/schreiben und können sich so gegenseitig zerstören)

Definitionen von Computer-Architektur	- Amdahl, Blaauw and Brooks 1967
	  - Wie der Prozessor aufgebaut ist, was sein Instruction-Set ist und welche Register und Adressen angesprochen werden können
	
	- Processor architecture
	  - Instruction-Set und Prozessoraufbau gesehen als Blackbox => wie sich ein Prozessor verhält und nicht wie der Physikalische Aufbau ist

Klassification von Computersystemen	Klassifiziert werden Systeme über ihre Paralellität
	5 Level:
	  - Programm Ebene
	  - Prozess Ebene
	  - Tasks
	  - Block Ebene
	  - Threads
	  - Instruktionen Ebene
	  - Basis Instruktionen
	  - Sub-Operationen Ebene
	  - Aufgeteilt in Micro-operationen

Darstellungsformen von Negativen-Zahlen	- Betrag + Vorzeichen - Einerkomplement - Zweierkomplement - Offsetdual-Darstellung

Betrag + Vorzeichen	- Carrybit gibt an, ob die Zahl Positiv oder Negativ ist
	  - 0 => positiv
	  - 1 => negativ
	
	- 2^(n-1) positive als auch negative Zahlen
	- 2 Darstellungen für die 0!!! 1000 = 0000 (-0=+0)
	[-2^(n-1)-1 : 2^(n-2)-1]

Einerkomplement	- Kippen der Bits - Carrybit gibt wieder an, ob die Zahl positiv oder negativ ist - 2^(n-1) positive als auch negative Zahlen - 2 Darstellungen für die 0!!! [-2^(n-1)-1 : 2^(n-1)-1]
Zweierkomplement	- Kippen der Bits + 1 - Macht man, um bei der 0 einen überlauf zu erzugen (-0 im EK = 1111 => 0 im ZK => 0000) - Carrybit codiert die Zahl und das Vorzeichen - 2^(n-1) negative Zahlen und 2^(n-1)-1 positive Zahlen, da die 0 im negativen bereich nicht existiert [-2^(n-1) : 2^(n-1)-1]
Offsetdual-Darstellung	- Verschiebe den Zahlenraum um n positionen (schiebe die Darstellung der Negativen Zahlen in den Positiven Bereich) => genau gleichviele Positive wie negative Zahlen => Offset darf frei gewählt werden, doch nach konvention ist er für Zahlen 2^(n-1) Addiere auf alle Zahlen diesen Offset 2^(n-1) negative Zaheln und 2^(n-1)-1 positive Zahlen, da die 0 im positiven Bereich liegt [-2^(n-1) : 2^(n-1)-1]

Reellezahlen	- Fix-Point-Numbers
	  - festes Komma und damit eine feste genauigkeit nach dem Komma
	
	- Floating-Point-Number
	  - Komma nicht fest und kann damit fließen
	  - Zahlen sind ungleich auf dem Zahlenraum verteilt:
	  - Höhere genauigkeit nahe 0 doch großer Zahlenraum (je größer die Zaheln, desto geringer ist die Genauigkeit nach dem Komma)

Floating-Point-Number	- Besteht aus Vorzeichen, Charakteristik (Exponent in Offsetdual darstellung) und Mantisse (im ZK)
	- Stelle des Kommas der Mantisse ist festgelegt (meistens vor dem most-significatnt-bit)
	- Normalisiert heißt ein Float, wenn die Mantisse: 1/b <= mantisse < 1 ist mit b = basis des floats (im Rechner basis = 2)
	  - Bei der Normalisierung trifft man häufig die annahme, dass die Zahle mit 1,... anfängt und spaart sich damit die 1 vor dem Komma
	  - Erhöht die größe der Mantisse um eine Stelle ohne einen größeren Speicher zu benötigen
	
	- Länge von Characteristik gibt den Darstellbaren Zahlenraum an
	- Länge der Mantisse gibt die genauigkeit des Floats an

Floating-Point-Number Overflow/Underflow	Vom Overflow spricht man, wenn man versucht eine zu große Zahl darzustellen. Vom Underflow spricht man, wenn man versucht eine Zahl nahe 0 darzustellen, die nicht darstellbar ist
Floating-Point-Number max real	Maximal Positive Zahl, die mit einem Float darstellbar ist (Bsp: Characteristik alles 1 und Mantisse alles 1)
Floating-Point-Number min real	Minimal Positive Zahl, die mit einem Float darstellbar ist (/= 0)
Floating-Point-Number small real	Minimaler Float, den man auf 1.0 Addieren kann und die Zahl verädert 1 + (small real/2) + (small real/2) = 1 und nicht 1 + ((small real/2) + (small real/2)) = 1 + small real Das ist der Fall, weil wir mit begrenzent resourcen arbeiten

IEEE Rundung	- Exates rechnen mit folgender Rundung muss das selbe Ergebnis liefern, als würden wir erst exact rechnen und dann runden
	
	- Einführung der g(uard) und r(ound) stelle + s(ticky bit)
	- sticky bit == 1, wenn durch das shiften der bits der Mantisse irgendwann eine 1 rausgeshiftet wurde
	- r = 0 -> g abrunden (nicht modifizieren)
	- r = 1 -> anschauen des sticky bits
	  - sticky bit = 1 -> aufrunden
	  - " = 0 -> abrunden

Addition und Subtaktion	- Eine Addition kann über einen halb/volladdierer passieren - Eine Subtraktion ist das gleiche wie die Addition der Negativen Zahl
Halbaddierer	- Nimmt a und b - a xor b = s - a und b = ü a und b sind bits s ist das summen bit ü ist das Übertragsbit - Reicht nicht für mehrstellige Addition siehe 2.76
Volladdierer	Reihe von 2 halbaddierer nimmt a, b und ü a xor b xor ü = s (a xor b) und ü oder a und b siehe 2.78
Carry-Ripple-Addierer	- Aneinandereihung von Volladierer - Übertrag muss durchgereicht werden - sequienzielle Rechnung - dauer der übertragsleitung gibt max. taktung an
Carry-Lookahead-Addierer	- Löst Rechenzeit durch rekursive Berechnung des Übertrages - Übertrag wird mehrfach Berechnet => kein weiterleiten des Übertrages => konstante Laufzeit - Problem: braucht viel Platz
Überlaufserkennung bei der Addition	- Der Überlauf vom vorletzen bit und des letzten bits muss gleich sein - ist das nicht der Fall gab es einen Überlauf, der nicht gewollt ist
Komponenten einer einfachen ALU	- Registersatz - 2 Eingabe Ports - Multiplexer-Schaltnetz - entscheidet, welche und wie die Inputs gelesen werden - ALU-Schaltnetz - 2 Eingaben und 1 Ausgabe - Zuständig für alle Arithmetisch und Logische Funktionen - Schiebenetzt - Zuständig für das Verschieben der Bits - Alle Teile werden über eine Folge von Bits angesprochen siehe 2.101
Aufbau eines Micropozessors	siehe 3.4
Aufbau des Steuerwerks	- Taktgenerator - gibt die Geschwindigkeit der CPU vor - Befehlsregister - speichert gefeatchte Befehle - Decoder - wandelt die Befehle in microprogramme um (falls nötoig) - Steuerregister - Übermittelt Aufgaben an die ALU
Phasen der Befehlsausführung	- Instruction Featch - Instruction Decode - Execute
Register im Steuerwerk	- Befehlsregister - multible Register damit mehrere Befehle gleichzeitig vom Memory gefeatched werden können - Steuerregister - enthält Bitfolge zum ansprechen der ALU/des Rechenwerks
Register im Rechenwerk	- Statusregister - z.B. für Flags - Hilfsregister (für zwischenergebnisse?)
Verbindung des Steuerwerks und Rechenwerk	- Microbefehle aus dem Stuerwerk spricht ALU an
Prozessorstatusword (PSW)	- Besteht aus status- und stuerregister (im Bezug des Contextwechsel auch die regulären Register) - Bei Prozesswechsel wird PSW und PC gespeichert
Stackframe	- Begrenztert Bereich im Speicher, der Dynamisch vom Programm angelegt werden kann - Benötigt das Speichern der Basis des Stackframs und des Index, wo sich der Pointer befindet - Ermöglicht effizienteren Zugriff auf den Speicher und effizientere Speicherfreigabe
Automatisch modification von Registern	- Es existieren Register, die automatisch operationen auf sich selbst ausführen können - e.g - Increment - Decrement - Scaling
System-Bus-Schnitstelle	- Verbindet CPU mit Hauptspeicher - Rechnet Spannungen um - Datenbus größe mindestens == Adressbus größe
Leistungssteigernde Möglichkeiten	- Technologische Maßnahmen; - Transistoren anstatt vakkuum röhren (historisch) - Strukturelle Maßnahmen; - Anzahl der Transistoren erhöhen - Paralelisierung (ganz wichtig!) - Multicore
Rechnerarchitektur nach Flynn	- SISD (Single Instruction Single Data) - klassischer von-Neumann-Rechner - SIMD (Single Instruction Multiple Data) - bsp. GPU - MIMD (Multiple Instruction Multiple Data) - Klassische mehrprozessoren Systeme - MISD (Multiple Instruction Single Data) - Pipeline als weit entferntes Modell
Pipelining	- Paralelisierung von Aufgaben (wie am Fließband) - Währen 1. in Instruction Decode ist kann 2. schon in Instruction Featch - befehle müssen unabhängig von einander sein - Anzahl der Teilschritte == Pipeline Stages (k) - IF - ID - OF - EX - WB = 5 Pipeline Stages (k) - Taktung der Pipeline ist durch den Langsamsten Teilschritt vorgegeben - Throughput = number od instruction leaving pipeline per clock cycle
Effizienz von Pipelins	Ohne Pipeline: - k * n = cycles untill all instructions executed (n => instructions) Mit Pipeline: - k + (n-1) = cycles untill all instructions executed (n => instructions)
Pipeline Speedup s in perfect world	k/((k/n)+1+(1/n)) ; k = pipeline stages and n = number of instructions
Problems of Pipelines in real world	- Abhängigkeiten von Befehlen

Pipeline Hazards	- Data Hazard
	  - value needed, that is currently in calculation
	- Structural hazard
	  - needed resources, that are not available
	- Controll hazard
	  - Jumps and Conditional jumps = non-linear program

Data Hazards	- RAW (Read after write)
	  - Instruction 2 wants to read operant before instruction 1 wrote it
	- WAR (Write after read)
	  - Instruction 2 writes operant before instruction 1 read it
	  - only in superscalar pipelines
	- WAW (Write after Write)
	  - Instruction 2 wantes t owrite before instruction 1 wrote it
	  - only in superscalar pipelines

Solutions to Data hazards	Software solutions:
	  - No-Operations (NOP's)
	  - Reordering of commands
	
	Hardware solutions:
	  - Pipeline Stall / Bubbles
	  - Forwarding of informations
	  - Result forwarding: Exe -> Exe
	  - Forwarding vom MEM -> Exe
	
	(In MIPS-Architecture)

Structural Hazards	- Resource Problems - Stalling or resource upgrade is the only solution
Control Hazard	- Stalling of Pipeline till conditional jump is evaluated - Branch Prediction to have a "linear"-programm - Worst-Case: Pipelineflush!

Branch Prediction	- Deciding, if a conditional jump should be taken or not (as a prediction)
	- One Solution is the Branch-Delay-Slot: Loading a command into the Pipeline that can be ececuted in both scenarios
	
	- Static Predictions:
	  - always take/not take jump
	  - compile decides if jump is always taken/not taken
	- Dynamic Predictions:
	  - evaluate the past predictions and therefore adapt behaviour
	  - one-bit predictor and two-bit predictor

Branch-Target-Buffer	- Table of known jumps + it's locations to jump to - In complex versions it also contains the state of the Branch-Prediction-Automat
One-Bit-Predictor	2 states: - Predicate Take - Predicate Not Take Miss judgement will change the state for next predictions
Two-Bit-Predictor	4 states: - Strongly Take - Weakly Take - Strongly Not Take - Weakly Not Take
Hystersis Two-Bit-Predicator	See: 3.130
Saturation-Counter Two-Bit-Predicator	See: 3.131
Hystersis vs Saturation-Counter	Hystersis has the advantage not to flutter between Weakly Not Taken and Weakly Taken because there are no direct connections between the two
Predicated Instructions	Predicated Instructions are always executed, but the result has only an impact, if the condtion is true

Pro/Contra Predicated Instructions	Pro:
	  - eleminated the need for a branch prediction
	  - keeps the pipeline busy
	
	Contra:
	  - Unneccessary code might be executed
	  - Might be executing to much code that is unneccessary, if the execution of the predicated instructions takes longer than a flush would take

Vektor-Pipelining	Ausführung des selben commands auf einen Vektor von Daten. Bsp. Array von n Zahlen in einem großen register und auf jede Zahl 3 Addieren schneller als n Register mit je 1 Addition SIMD Prinzip
Chaining	Eine Instruction führt mehrere Vektor-Instructionen hintereinander aus

Superscalare Prozesse	- Eine Pipeline hat mehrere Einheiten von verschiedenen Operationen (Multiple Addition, Multiplikations, etc. Einheitem)
	- Langsamster Takt gibt nicht die gesamte Taktung vor
	- echte Gleichzeitigkeit!
	  - mehrere Befehle können pro Takt ausgeführt werden
	  - CPU (Cycles per Instruction) << 1.0 möglich!
	- Befehle können einander überholen (da verschiedene Operationen verschiedene Zeiten haben)
	- in-order -> out-of-order -> in-order
	  - IF ID -> OF EXE -> Retire/WB
	- Siehe 3.150 für Darstellung

Issue-Unit	Verteilt in Superscalaren Prozessen die Befehle auf die verschiedenen Recheneinheiten Entweder in-/out-of-order (je nach implementation)
Reservation- Station	Vor der Executstation; hier warten die instructionen auf ihrere Daten
Completion of Superscalar Processes	Happens in-order Has to wait for operations to finish in order before it can write them back

Instruction-Set-Architectur (ISA)	RISC (Reduced-Instruction-Set)
	  - directly executable without microprograms
	CISC (Complex-Instruction-Set)
	  - execution of microprograms

CISC (Complex-Instruction-Set)	- ISA soll mächtige Befehle enthalten, sodass selten auf den Speicher zugegriffen werden muss
	- 1 CISC-Befehle => mehrere Befehle im Microprogramm
	
	Contra:
	  - Benötigt sehr großes Schaltnetz
	  - nur wenig Befehle werden oft genutzt (lohnt sich also der Aufwand?)

RISC (Reduced-Instruction-Set)	- Reduzierte Befehlsanzahl
	  - <= 128
	- Reduzierte Formate
	  - <= 4
	- Reduzierte adressierungs modelle
	  - <= 4
	
	Pro:
	  - Creates space on chip
	  - memory access only allowed for load and store (so schedueling of instructions is easier because these two are the "heavierst" for the cpu)
	  - Pipelining developed favoured because of this
	
	Contra:
	  - von-Neumann-Bottleneck

Procedures, Traps and Interrupts	- Programme werden unterbrochen durch:
	  - Procedures (functions Aufrufe von z.B. OS funktionen),
	  - Multithreading (OS wechselt auf einen anderen prozess) und
	  - Hardware bzw. Software Interrups

Interrupts	- Programme können unterbrochen werden, wenn etwas unmittelbar unterbrochen werden muss
	- Der Prozess des Interrupts wird Interrupt Service Routine (ISR) genannt
	  - sollte rellativ schnell sein
	- Interrupt-Vektor-Table (IVT)
	  - Mapping von der Service routine zu dem jeweiligen Interrupt
	  - Die Adresse des Vektor-Tables is fixed, dass heißt das betriebssystem muss diesen Table füllen mit ISA
	- Bevor die Interrupt-Service-Routine ausgeführt werden kann muss der PC und Context des Programms gesichert werden
	  - Während der Contextsicherung darf der Interrupt nicht Unterbrochen werden! Sonst kommt man in einen nicht definierten Programmstatus
	
	- Interrupts können einander unterbrechen, wenn dies gewünscht ist. Regel ist: Interrupt eines Interrupts nur wenn die Priorität des Interrupts höher ist
	
	- Software Interrupts => Process internal (synchron ausführung)
	- Hardware Interrupts => Process external (asynchron ausführung)

Ablauf einer Interrupt Service Routine (ISR)	- Aktivierung des Interrupts
	- Beenden des gerade in Ausführung befindlichen Befehls
	- Festellen ob es sich um ein SW oder HW interrupt handelt
	- Festellen, ob Interrupts erlaubt sind (Interrupt Enable bit == true)
	- Falls HW-Interrupt:Quelle festellen und Interrupt Acknowledge schicken
	- Interrupt Enable Bit = 0
	- Save PSW und PC
	- Startadresse des ISR ermitteln und starten
	- ISR ausführen
	  - Context sicherung
	  - Interrupt Enable Bit = 1
	  - Routine ausführen
	  - IRET-Befehl ausführen (Ende der ISR)
	- Conext, PSW und PC wieder hestellen

Interrupt-Controller	- Managed einkommende Interrupt anfragen

Methoden von Interrupt Handelling	- Polling:
	  - CPU checkt periodisch Interrupt-Controller, ob ein Gerät einen Interrupt angefordert hat
	  - Polling 1:
	  - cyclische abfrage der Geräte, ob sie ein Interrupt haben. Wenn ja und der bearbeitet wurde, dann chekt die CPU das nächste
	  - Polling 2:
	  - cyclische abfrage der Geräte, startet nach bearbeitung eines Interrupts wieder bei 1
	- Interrupt:
	  - Interrupt-Controller meldet sich bei der CPU sobald ein Gerät ein Interrupt meldet

Speicherhierachrie	- Durch den effekt: klein und schnell oder groß und langsam wird der Speicher in Systemen hierarchisch angeordnet
	  - Kommunikation zwischen den verschiedenen Speicherschichten ist nur unmittelbar erlaubt (see 4.8)
	  - um Zugriffszeiten zu minimieren (es wird immer im nächst langsameren Medium nach dem Datum gesucht)
	- Speicher soll so wirken als sei er beides: groß und schnell
	- Speicherverschiebung wird genutzt um zu gewährleisten, dass benötigte Daten so schnell wie möglich gelesen/bearbeitet werden können

Lokalitätsprinzipien des Speichers	Zeitliche lokalität:
	  - was benötigt wurde wird schnell wieder benötigt
	Örtliche lokalität:
	  - was nebeneinander steht wird oft gemeinsam/nacheinander benötigt
	  - bsp. Arrays

Zugriffszeit und Zykluszeit	Zugriffszeit:
	  - Definiert als Zeit die benötigt wird um ein Datum aus dem Speichermedium zu lesen
	
	Zykluszeit:
	  - Definiert als Zeit die benötit wird um das Datum auszulesen und sich der Speicher regeneriert hat um den nächsten Zugriff zu erlauben

Cache-Speicher	- Wird genutzt um die verhältnissmäßig langsame Geschwindigkeit des RAMs zu kompensieren - Wichtige Informationen aus dem DRAM werden als Blöcke in den Cache gelagter, in der Hoffnungs, dass sie bald wieder benötigt werden - Darstellung des Cache-Speichers 4.26

Was Passiert bei einer Abfrage eines Datums	- Adresse wird über den Adressbus an den Cache und Speicher weitergeleitet
	  - Steuersignal wird zunächst nur an den Cache geschickt
	- ist das Datum im Cache (Cache-Hit) wird das Datum auf den Datenbus gelegt
	- ist das Datum im Cache (Cache-Miss) so wird das Steuersignal an den Hauptspeicher weitergeleitet, der dann nach dem Datum sucht
	  - Datum wird auf den Datenbus gelegt
	  - Cache speichert Datenblock ab mit dem benötigtem Datum

Cache Performance	- Cachegeschwindigkeit ist am höchsten, wenn die beiden Lokalitätsprinzipien eingehalten werden
	- Hitrate = Anzahl der Treffer Pro Zugriff
	- t_access = (Hitrate)*t_hit+(1-Hitrate)*t_miss
	  - t_hit = time needed for hit
	  - t_miss = time needed for miss

Cache-Schreibzugriffe	- Bei Cache-Miss:
	  - Änderung des Datums im Speicher und Schreiben des neuen Datums in den Cache
	- Bei Cache-Hit
	  - Veränderung des Datums im Cache und je nach implementation directe Weiterleitung des Datums an den Hauptspeicher

Cache write through	Änderung des Datums direct im Cache und weiterleitung an den Arbeitsspeicher
	
	Pro:
	  - garantierte konsistenz zwischen Cache und Speicher
	Contra:
	  - Schreibzugriffe dauern lange

Cache buffered write through	Änderung des Datums im Cache und weiterleitung über einen Buffer an den Arbeitsspeicher um die Schreibgeschwindigkeit des Speichers zu kompensieren
	
	Pro:
	  - Prozessor muss nicht auf beendigung des Schreibbefehls warten
	Contra:
	  - Temporäre Inkonsistenz
	  - machtigerer Controller wird benötigt um doppelte Schreibzugriffe abzufangen (neuer Schreibbefehl obwohl vorheriger noch läuft)

Cache write back	Änderung des Datums im Cache, doch keine weiterleitung des Datums an den Speicher.
	Erst bei Auslagerung des Datums wird geschaut, ob das Dirty-bit gesetzt ist (ob das Datum verändert wurde) und wenn ja dann wird es im Speicher geändert
	
	Pro:
	  - Schreibbefehle können in schneller Cache geschwindigkeit geschehen
	Contra:
	  - Inkonsistenz zwischen Cache und Speicher (großes problem für multi-core prozessoren)

Aufbau eines Cache-Speichers	See 4.48
	
	- Aufgeteilt in Adress und Datenspeicher
	- Komparator checkt, ob Adresse im Cache ist
	- Cache-Lines werden Datenspeicher abgespeichert
	  - mehrere bytes auf einmal einladen
	  - Örtliche lokalität

Begriffe zum Cache	Block:
	  - Datenportion die zu einem Block zusammen gefasst ist
	Blockrahmen:
	  - Eine Reihe im Cache mit der Basis Adresse des jeweiligen Blocks, Statusbit (ob Cacheline valide ist) und Daten des Blocks (und vielleicht Dirty-bit)
	Blocklänge:
	  - Anzahl der Speicherplätze im Blockrahmen
	
	Assoziativität:
	  - Anzahl der Blockrahmen, die zu einem Satz zusammengefasst wurden

Cachespeicher Assoziativitätsvarianten	Voll-Assoziativ:
	  Cache besteht nur aus einem Satz (alle Blöcke konkurieren um alle Plätze, dafür kann jeder Block überall im Cache liegen)
	Direct-Abgebildet:
	  Jeder Satzt enthält nur einen Blockrahmen
	  (Ein Block kann immer nur an einer Stelle im Cache stehen und Konkurriert mit allen anderen die diesen Platz brauchen)
	n-fach-Assoziativ:
	  Sätze bestehn aus n Blöcke die nur miteinander konkurieren, wenn sie auf den selben Satz mappen

Verdrängungsstrategien	- Verdrängungsstrategien werden genutzt um Platz für Blöcke zu schaffen der bereit belegt ist - Nur verwendet bei Voll-Assoziativ und n-fach-Assiziativ - Bei Direct-Abgebildet ist das nicht notwendig, dass der Block nur an einer stelle stehen kann

Voll-assoziativer Cache	- Alle Blöcke können überall im Cache stehen
	- Echtparallel aller Adressen im Cache
	
	Pro:
	  - Ein Block kann überall im Cache stehen
	  - Optimale Vorraussetzungen um am effektivsten zu verdrängen
	  - Optimale Ausnutzung des Caches
	Contra:
	  - Sehr hoher Hardwareaufwand
	  - nur für sehr kleine Caches verwirklichbar
	  - Verdrängungstrategie braucht komplexe Logik in der Hardware

Direct-mapped-Cache	- Ein Datum Mapped auf nur einen Platz im Cache
	
	Pro:
	  - Gerringer Hardwareaufwand
	  - Keine Logik benötigt für die Verdrängung
	Contra:
	  - Ständiges Konkurrieren der Blöcke (Flattern kann passieren, da viele Blöcke um genau den selben Platz konkurrieren)

n-Way Set Associative Cache	- Sätze enthalten mehre Blöcke, dass heißt weniger Konkurrenz zwischen verschiedenen Blöcken, da sie auf mehr Plätze pro Satz mappen - Bei Suche nach Datum müssen alle n Tags des Satzes überprüft werden - Ist ein kompromiss zwischen Voll-assoziativer- und Direct-mapped-Cache Je größer der Satz, desto näher n-Way an den Pro und Contras des Voll-assoziativen und umso kleiner desto näher an Direct-mapped Pro und Contra

Virtueller-Speicher	- Jeder Prozess bekommt seinen eigenen Logischen-Adressraum
	  - Die Adressen des logischen-Adressraumes müssen auf die realen-Adressen gemappt werden
	  - Bei Programmstart wird entschieden, wo das Programm im realen-Adressraum abegelegt wird
	
	- Betriebssystem kümmert sich um die Verwaltung des Virtuellen-/Logischen-Adressraums
	  - Hardwaretechnisch von der MMU (Memory-Management-Unit) unterstützt
	  - Enthält eine Übersetzungstabellen, die vom Betriebssystem verwaltet wird
	- Wird genutzt um nicht benötigte Daten aus dem Hauptspeicher in den Harddrive auszulagern
	
	-Sinnvoll für Multitasking/Multithreading/Multiuser

Aufgaben der MMU (Memory-Management-Unit)	- Umrechung der Logischen-Adressen in Reale-Adressen - Schutz vor Falschzugriff auf nicht berechtigte Real-Adressen - Datenschutz

Virtuelle-Speicher-Verwaltung	- Paging
	  - Daten werden in Seiten mit fest bytelänge unterteilt
	  - "Feste Segmentlänge"
	- Segmentierung
	  - Programme können in verschiedene Segmente unterteilt werden
	  - jedes Segement verhält sich wie ein eigener Speicherraum

Speicherverwaltung Paging	- Daten werden in Seiten mit fester länge unterteilt
	- Seiten sind realtiv klein (max. 4kbyte)
	- Ein Prozess wird auf viele Seiten verteilt
	
	- Im Virtuellen Speicherraum sind die Prozesse zusammenhängend; im Realen-Speicherraum können die Pages verteilt sein
	- Page-Table speichert die Abbildung des Virtuellen-Speicherraums auf den Reallen-Speicherraums
	  - Übersetzung der Virtuellen-Adressen in Real-Adressen
	
	Problem:
	  - Dynamische Speicherstrukturen können nur schlecht dargestellt werden, da Pages eine feste Länge besitzen

Speicherverwaltung Segmentierung	- Virtueller-Adressraum wird in Segmente verschiedener Länge unterteilt
	- Ein Programm besteht aus verschiedent vielen Segmenten
	- Jedes Segment ist in sich logisch zusammenhängend
	
	Pro:
	  - Segmentierung spiegeld logische Programmstruktur wieder
	  - Große Segmente => seltener Datentransfer
	  - Segmentspeziefische Zugangsrechte
	  - Read-Only oder Read-Write Zugriffe
	Contra:
	  - Wenn Segmente transferiert werden müssen, dann sind es große Transfere
	  - Keine einlagerung von Teilsegmenten

Virtueller Cache	- Adressen werden im Cache Virtuell abgespeichert
	- Keine Umrechnung der MMU benötigt um auf die Adressen zu zugreifen
	- Pro:
	  - bei Cache-Hit keine Umrechnung der MMU
	- Contra:
	  - Dadurch, dass Virtuelle Adressen öfters vergeben werden können kann der Cache nicht zwischen den Virtuellen Adressen verschiedener Prozesse Unterscheiden

Physikalischer Cache	- Wahre Adressen werden im Cache abgespeichert
	- MMU muss Adressen zunächst umrechenen bevor der Cache abgefragt werden kann
	- Pro:
	  - Keine Duplizierten Adressen
	- Contra:
	  - Umrechnung benötigt von der MMU
	  - Potentiell kleinerer Adressraum

Schutzmechanismenen des Hauptspeichers	- Schutz der Realenspeicherräume ist Hardwaretechnisch unterstützt - MMU

Cache-Verwaltung bei Multicoresystemen	- Speichergekoppelt (geteilter Speicher)
	  - Besitzen einen geteilten Adressraum pro Prozess
	  - Shared Memory (beide Prozessoren greifen auf den Physikalisch seleben Speicher zu)
	  - Destributed Memory (beide Prozessoren besitzen eigenen Physikalischen Speicher (Daten sind auf die verschiedenen Speicher aufgeteilt))
	- Nachichten basiert (Send/Recieve von Infos)
	  - Physikalisch zwei Speicher die über Send und Recieve miteinander Kommunizieren

Probleme von Multicoreprozessoren	- Höherer Verwaltungsaufwand - Systemverklemmung möglich (A wartet auf B wartet auf C wartet auf A) - Bottleneck wenn Prozessoren miteinander Kommunizieren müssen - Abstimmung des Cache (MESI)
Cache-Kohärenzproblem	- Kohärenz -> Caches können unterschiedliche Daten beinhalten (== nicht konstistent) doch die korrekte Funktionsweise ist gewährleistet - Konsistenz -> Cache ist bei allen Prozessoren auf dem selben Stand - Cache-Kohärenzprotokolle sicheren den korrekten ablaufen auch bei nicht konsistenten Datenverteilungen

Cache-Kohärenzprotokolle	Write-Update-Protokoll:
	  - Kopien aller Caches sind konsistent (koheränz gesichert, da konsistenzt existiert)
	Write-Invalidate-Protokoll:
	  - falls veränderung in einem anderen Cache geschieht, wird die alte kopie Invalid gemacht
	MESI-Protokoll:
	  - beruht auf Mitlauschen des Bussystems
	  - Besteht aus 4 Zuständen
	  - Modified
	  - Exclusive unmodified
	  - Shared unmodified
	  - Invalid