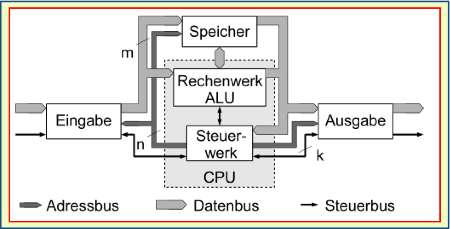
**1 Grundlegende Rechenarchitektur**

***Von-Neumann-Rechner = {Steuerwerk, Rechenwerk, Speicher(werk), Eingabe- & Ausgabewerk}***

* Program P & Data D im gleichen Speicher 🡪 keine Unterscheidung
* Steuerung über einzelne Befehle, gespeichert als Programm
* Universell: nicht fest verdrahtet, kann beliebige Programme/Berechnungen ausführen (Universalrechner; TM-mächtig)
* Programm:= sequentielle Abfolge v. Anweisungen; Befehlszähler gibt Adresse des nächsten Befehls an; Korrektheit wichtig
* Einfach in Technik und Funktion und realisierbar
* Steuerwerk/Control Unit:=Herzstück d. Rechners, interpretiert Programmcode, erstellt Verbindungen zwischen Daten & Rechenwerk, Steuert Reihenfolge d. Programmbefehle, beinhaltet *{Befehlsregister, Befehlsdecoder, Befehlszähler}*
* Rechenwerk/ALU:=arithmetische&logische Rechen-Operationen
* Speicherwerk:=Speichert alle Daten (P, D, Zwischen-&Endergebnisse im selben Speicher), Zugriff auf Speicherzelle via Adresse
* Eingabe/Ausgabewerk=I/O-Unit:=Steuert Ein-&Ausgaben aller Daten
* Bus-Systeme = {Daten-, Adress-&Steuerbus}:=Verknüpft Komponenten zum Austausch von Daten und Steuersignalen
* Register (Teil d. Steuerwerks):=Zwischenspeicherung v. Daten
* Übliche Register: *Befehlsregister*, *Befehlszähler* / *Programmzähler*, *Akkumulator, Zustandsregister, Statusregister*, *Interrupt-Register*
* Akku: Grösse d. Akkus & Arbeitsregister beschränkt pro Zyklus bearbeitbare Zahlen (e.g. 32Bit 🡪 232 differenzierte Darstellungen)
* Befehlszähler: Gr.d.Bz. bestimmt Grösse d. ansprechbaren=adressierbaren Speicherbereichs
* Befehlsregister: Gr.d.Br. bestimmt Anz. möglicher Befehle
* Register: Aktuelle Rechner haben i.d.R. 32 V 64-Bit-Register
* Programmablauf

1. Der aktuelle Befehl wird aus der Speicherzelle, auf die der Befehlszähler zeigt, ausgelesen und in das Steuerwerk übertragen
2. Das Steuerwerk dekodiert den Befehl und schaltet die ent- sprechenden Signale auf den Steuerleitungen
3. Die für den Befehl erforderlichen Operanden werden aus dem Speicherwerk gelesen und in das Rechenwerk bzw. die festgelegten Register übertragen
4. Die dekodierte Operation wird ausgeführt; das Ergebnis in ein Register (oder den Speicher) geschrieben
5. Der Befehlszähler wird um eins erhöht oder auf Grund eines Sprung-Befehls um einen anderen Wert verändert
6. Zyklus startet von vorne

* Programmcode modifizierbar 🡪 Programm und/oder Daten können beschädigt werden (be- oder unbeabsichtigt)
* Problem VN-Bottleneck: Trennung von Recheneinheit und Speicher 🡪 Daten müssen sehr häufig übertragen werden

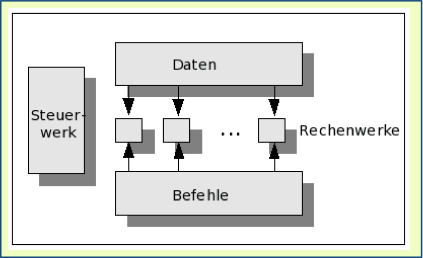
Damit wird der Datenbus, der für den Transport der Daten (P & D) zuständig ist, zum Engpass – dem sogenannten ***Von-Neumann-Flaschenhals***

🡪 wird verschärft, da expliziten Sequenzialismus besteht (keine Parallelität) und Datenbus für Übertragung von P & D genutzt wird (Aufteilung d. Kapazität)

* Lösung VN-Bottleneck: Einsatz v. schnellem Zwischenspeicher zwischen CPU und Speicher (Cache), Getrennte Zwischenspeicher für Datenbusse für P&D, Vorhersage von bedingten Programmsprüngen (branch prediction)

***Harvard-Rechner***

* Trennung von P&D-Speicher
* Nutzung getrennter Datenbusse für Zugriff auf P&D



|  |  |
| --- | --- |
| **Up’s** | **Down’s** |
| P&D können gleichzeitig geladen werden | Kann zu nichtdeterministischen Verhalten führen |
| P-Code kann nicht überschrieben werden |  |
| Befehls- und Datenwortbreite können unterschiedlich gross sein | Nicht benötigter Speicherplatz kann nicht für Daten genutzt werden (u. umgekehrt) |

* Super-Harvard-Architektur: gemeinsamer Speicher, verschiedene Daten-Busse 🡪 Speicher kann gemeinsam genutzt werden

**2 CPU**

***Leistungsmerkmale CPU := Befehlszahl, Zykluszeit, CPI***

* Befehlszahl
* Viele Befehle 🡪∑Befehle zur Berechnung eines Problems sinkt
* Anzahl Befehle hängt vom Compiler und Befehlssatz der CPU ab
* Taktzyklus(zeit) (Angabe als Frequenz): e.g. 4GHz = 0.25ns
* Kleine Taktzykluszeit => viele Befehle können pro Zeiteinheit aus- geführt werden
* Abh. von CPU-Implementierung
* soll minimal sein

***Taktzyklus(zeit) optimieren/ Reduktion von Leerlaufzeiten***

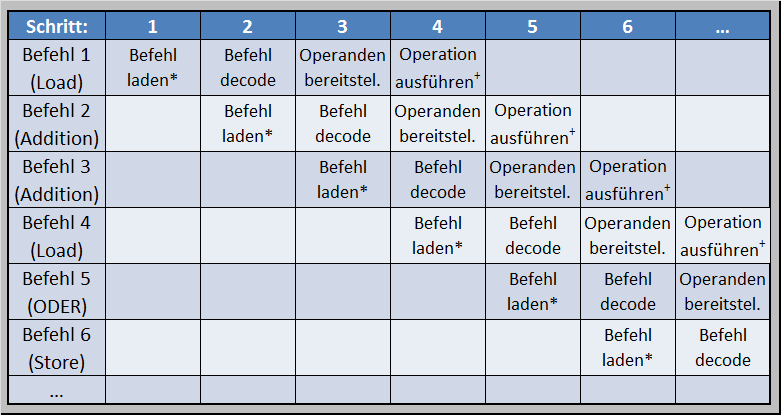
* Berechnung: Wenn alle Befehle in einem Zyklus abgearbeitet werden, bestimmt der Befehl mit dem längsten möglichen Datenpfad die Zykluszeit (worst case)
* Kann sehr komplex sein & ist i.d.R. nie für alle Befehle identisch 🡪 daher wird bei vielen Befehlen umsonst gewartet
* Optimierung nach seltenen Befehlen (nach häufigen  🡪 nicht möglich)

1. Architektur, Hardware, Befehlssatz so designen, dass alle Befehle gleich (d.h. Datenpfade sind gleich lang) lange brauchen (zur Ausführung) 🡪 in Realität nicht umsetzbar
2. Befehle in Gruppen unterteilen, die in etwa gleich lang brauchen (für Ausführung) 🡪 erfolgreiches Konzept, Optimierung für häufig genutzte Befehle möglich, Steuerung/Design aber komplexer
3. Pipelining: Mehrere Befehle überlappend/zeitgleich ausführen

* Nach Codierung und Verarbeitung (ALU) des 1. Befehls kann schon der nächste Befehl aus Speicher in Befehlszähler geladen werden
* Abarbeitung eines Befehls wird in kleinere Teilaufgaben gesplittet und
* Teilaufgaben parallel durchgeführt
* Details: s.u.

***Pipelining***

1. Befehl laden (Speicher 🡪 Befehlsregister)
2. Befehl decodieren (Steuerwerk)
3. Operanden bereitstellen
4. Rechenoperation durchführen (ALU) + Ergebnis schreiben



🡪 ∑Stufen d. Pipeline abh. von Implementierung

|  |  |
| --- | --- |
| **Ideale Bedingungen** | **Reale Bedingungen** |
| Pipeline-Stufen idealerweise gleich lang (längste Stufe bestimmt Zkl.zt.) | Nur ungefähr gleich lang, Wartezeiten |
| ∑Befehle möglichst gross 🡪 Stufen ausgelastet | Es gibt Anlauf- und Auslaufphase 🡪 Stufen nicht immer voll ausgelastet |
| - | Komplexität (Hardware) grösser 🡪 Stufen komplexer & i.d.R. langsamer |
| - | Nicht alle Befehle nutzen alle Stufen (Wartezeiten) |
| Befehl wird gleich schnell abgearbeitet | Wird etwas langsamer abgearbeitet |
| Gesamtausführungszeit: n-mal so schnell (Bei n Stufen) | Weniger als n-mals so schnell |

* Gesamtausführungszeit T (n Befehle, k-Stufen): T = (k + n – 1) \* Zykluszeit
  + e.g. T = (7+20-1)\*0.3125ns=0.00825µs 0.04/0.00825=4.92 knapp 5-mal schneller
* Wenige Stufen, um Komplexität zu beherrschen (i.d.R. 4-12)
* Optimierung Befehlssätze: Formate identisch/sehr ähnlich, Wenige Befehlsformate, Speicherzugriff nur via Load oder (xor) Store, Organisation d. Daten im Speicher (wortweise)
* Konflikte
* Strukturkonflikte (structural hazard) := Gleichzeitiger Zugriff auf Ressourcen durch aufeinanderfolgende Befehle 🡪 L: abgestimmter Befehlssatz (e.g. nur in Stufe 1 Daten für Befehle im Speicher zugreifbar)
* Datenkonflikte (data hazard) := Befehl greift auf Daten eines vorherigen Befehls zu, der noch nicht abgeschlossen ist (e.g. 2 Additionen: A:=A+R1; A=A+R2 *[A~Akku]*) 🡪 L: Umordnen d. Code V Forwarding/Bypassing (zusätzl. HW stellt Ergebnis früh genug bereit, reduziert Wartezeit)
* Steuerkonflikte (control hazard) := Immer, wenn Programmausführung von Sprungbefehlen beeinflusst wird 🡪 L: Branch Prediction (zusätzl. HW; sehr komplex, aber sehr erfolgreich (>90%, da meist eingesetzt bei Schleifen: e.g. bei n Durchgängen wird n-1 mal geraten, beim Verlassen des Loops falsch geraten), kurze Pipeline vorteilhaft)
* CPI := clock cycles per instruction (Taktzyklen pro Befehl, d.h. øAnz. Taktzyklen, welche ein Befehl zur Ausführung benötigt)  🡪Abh. von CPU-Implementierung
* Ausführungsdauer Program P: tP= (Anz. Befehle) \* Zykluszeit \* CPI = e.g. 100 000 \* 2ns \* 1 (CPI=1) = 0.2ms
* Komponenten = {Befehlzähler/Befehlsregister, Steuerwerk, Register, ALU=Rechenwerk} // inkl. dazugehörige Steuer-, Daten-, Adressleitungen, Datenlogik, Speicher (entspricht d. Von-Neumann-Rechner- Architektur)
* Für die Bearbeitung von Befehlen wird auf den Speicher zurückgegriffen
* Befehle führen arithmetisch-logische Funktionen aus (mittels ALU ~ Arithmetisch Logical Unit)
* Schaltnetz  – besteht aus logischen Bauelementen (keine Speicherbausteine)
* Eingangswerte bestimmen Ausgangswerte – Kombination von Schaltnetzen sind Schaltnetze
* e.g. Schaltnetz: *3-Bit-Multiplexer = {n Steuereingänge, 2ˆn Eingänge, 1 Ausgang}* 🡪 (genau 1 Eingang wird via Steuerausgänge selektiert und mit Ausgang verbunden/durchgeschaltet)
* e.g. Schaltnetz: *3-Bit-Demultiplexer={n Steuereingänge, 1 Eingang, 2ˆn Ausgänge}* 🡪 (Eingang wird via Steuereingänge mit genau einen Ausgang verbunden/durchgeschaltet)

• Optimierte Schaltungen

* e.g. Addierer (theor. realisierbar über bools. Schaltungen, in Praxis aber Addierwerke := spezielle bzl. Geschwindigkeit optimierte Schaltungen)
* e.g. Optimierte Schaltungen: Halb- & Volladierer, Carry-Ripple, Carry-Skip, Carry-Look-Ahead, Conditional Sum Addition, Carry- Select

• Schaltwerk

* können intern Datenwerte speichern
* min. 1 Dateneingang, 1 Datenausgang, 1 Takteingang
* Takteingang bestimmt, wann Werte geschrieben werden
* Unterscheidung: pegel- und flankengesteuerte (edge triggered) Schaltwerke
* heute: meist frankengesteuerte genutzt (Hazard, Glitch, race- condition)

• Schaltwerk vs. Schaltnetz

* Schaltwerke = Speicher, Schaltnetz = Intelligenz
* Eingabe von: Schaltwerk, Berechnung: Schaltnetz, Ausgabe nach:  Schaltwerk (SW1 -> SN -> SW2)
* Bei flankengest. Schaltw.: Lesen & Schreiben von/ins gleiche Schaltw.  (SW<->SN)

***CPU-Komponenten***

* Akkumulator ~ spezielles Register, in welchem Berechnungen der ALU gespeichert werden (oft direkt mit ALU verbunden)
* Befehlszähler ~ spezielles Register, in welchem die Speicheradressen der auszuführenden Befehle stehen. Wird nach Ausführung von Befehlen inkrementiert, so dass der Pointer auf der Speicheradresse liegt, das den nächsten auszuführendem Befehl enthält
* Befehlsregister
* ALU
* Steuerwerk

***Befehle***

* Store / Load Befehle = { Arithmetisch-Logische Fkt. (ALU), Speichern und Laden v. Daten in/aus Register (oder Speicher)}

**3 Befehle**

***Einführung***

* Wörter ~ Befehle, Sprache ~ Befehlssatz (instruction set)
* Intelligenz im Rechner steckt im Programm (Rechner wird von Befehlen  gesteuert)
* Begriff := Rechner berechnet arithmetisch-logische Funktionen (e.g. arithmetisch: Addition, logisch: OR)
  + 1. – e.g.1 OP-Code, Operand-1, Operand-2, Operand-3, (Option)
    2. – e.g.2 ADD, a, b, s, - // s=a+b, keine Option (-)
    3. – e.g.3 OR,u,v,w,- //w=uORv, keineOption(-)
    4. – e.g.4 ADD, R1, 100, R2 // Addition mit Register (R1, R2) und  Wert 100 (Schreibe (Inhalt R1)+100 nach R2)
* (Operanden-)Register (e.g. R1,R2) := Werden benötigt, um Speicherzugriffe zu reduzieren/eliminieren, da jene ≥ Faktor 100 x langsamer ggü. Operationen in CPU sind (e.g.1-4: keine Speicherzugriffe notwendig, da mit R1,R2 gearbeitet wird). Daher werden arithm.-log. Fkt. oft mit Operanden-Register berechnet.

🡪 Register: sehr schnell, teuer, hardwaretechnisch gross, verschlechtert Zykluszeit (daher nicht beliebig viele einsetzbar, e.g. 32).

* Befehle und Register
  1. Ausgezeichnetes Register ~ Akkumulator (dafür nur 1-2 Operanden) 🡪 Akku wird für viele Operationen implizit als ein Operand genutzt, ggf. als dritter für Ergebnis

e.g. ADD R1 // A := A + R1

* 1. Befehl > 1 Wort 🡪 Befehl setzt sich aus n>1 Wörtern zusammen, dafür müssen mehrere Wörter ins Befehlregister geladen werden, welches ebenfalls entsprechend breiter ist
* Indirekte Adressierung (statt direkter Angabe einer Speicheradresse): Inhalt eines Registers spezifiziert die Speicheradresse und/oder ein (festgelegtes\*) Basisregister definiert einen Basiswert

e.g. Load, R1 // A := Inhalt des durch Wert von R1 adressierten Speichers

\* soll beim Programmstart festgelegt werden

* Offset: Bei Lese- und Schreibbefehlen zum Wert eines Registers addieren

e.g. Ldoff, R1, 5 // A := Inhalt d. Speicher mit Adresse Wert(R1)+5 🡪 kompl. Sprünge

* Branch Instructions (Bedingte Sprungbefehle): Häufig wird hier ein ausgezeichnetes Register auf Null\* geprüft (\*da sehr einfach und schnell realisierbar)

e.g. Bnull, R1 // if A\*\* = 0: Branch (go s’where else) to Adresse #Inhalt(R1)

*🡪 further conditions: 0, >0, <0, R1=R1, R1≠R2, R1=4, R1≠4*

*\*\* Neben Akku werden häufig spezielle Register wie Status-, Interruptregister, oder Stack-Pointer angesprochen.*

*Theoretisch 1 Sprungbefehl ausreichend, um alle abzudecken. Denoch werden verschienene implementiert, da somit der Umfang des Programmcodes erheblich reduziert bzw. vereinfacht werden kann (“Bequemlichkeit”)*

* e.g. IF: “branch if not null” // if(!0) {C1} else {C2}; C3

1. Bedingung berechnen und Resultat in R1 schreiben
2. R1 != 0?
   1. 1 🡪 kein Sprung (Programmfortsetzung & Ausführung der Befehle des Schleifenkörpers C1 & Sprung nach C3)
   2. 0 🡪 Sprung nach Schleifenkörper (C2)

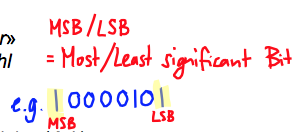
* e.g. WHILE “branch if not null” // while(!0) {C1}; C2

1. Bedingung berechnen und Resultat in R1 schreiben
2. R1 != 0?
   1. 1 🡪 kein Sprung (Programmfortsetzung & Ausführung der Befehle des Schleifenkörpers C1 & Sprung zu while-Condition)
   2. 0 🡪 Sprung nach C2

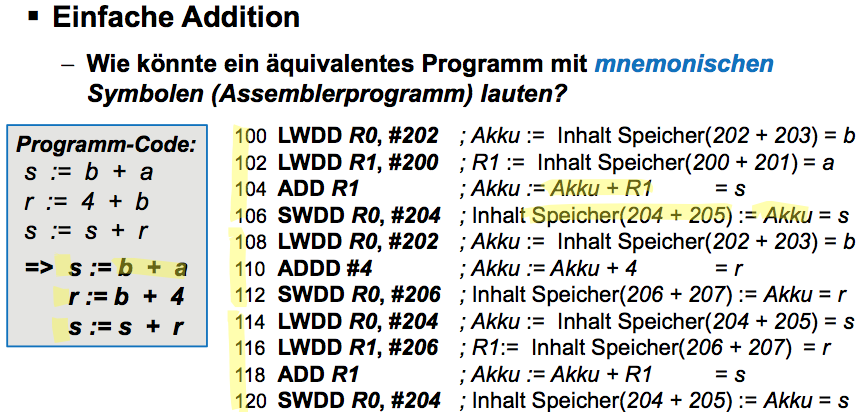
* e.g. FOR // do {C1} for(i=0; R1>0; R1--)

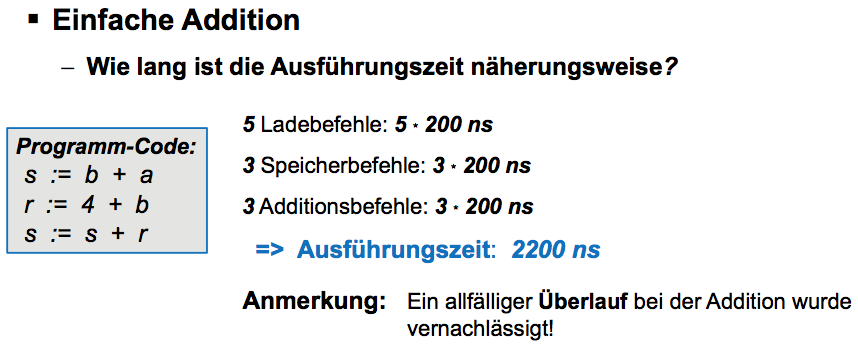
1. Schleifenzähler initialisieren // R1 = n>0
2. Schleifenkörper (C1) durchlaufen // C1
3. Schleifenzähler (R1) um 1 reduzieren // R1 -= 1
4. R1 = 0?
   1. 1 🡪 kein Sprung // exit
   2. 0 🡪 Sprung zu C1 // GoTo C1

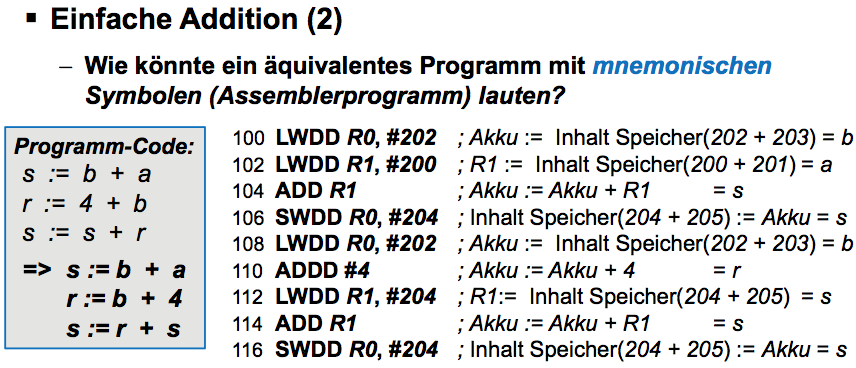
* e.g. CASE SWITCH 🡪 Folge von IF-Anweisungen V Tabelle mit Sprungadressen, die zuvor inidiert wurde (Manche Befehlssätze sehen dafür ein bereits ausgezeichnetes Register vor: Jump-Register)
* Prozedur-Aufruf/Unterprogramm: Fortsetzung an anderer Stelle (unbedingter Sprungbefehl) 🡪 Parameter-Übergabe und Rückgabewerte (beides durch Abspeichern der Werte an vereinbarten Stellen=Speicher V spezielle Register) 🡪Zurückspringen zum ursprünglichen Programm (unbedingter Sprungbefehl an zuvor abgespeicherte Adresse des Befehlszählers des aufrufenden Programms) 🡪Fortsetzung des Programms
  + Verschachtelte Aufrufe: erforderliche Registerinhalte müssen jwl. Abgespeichert & wiederhergestellt werden (Realisierung: Stack/LIFO-Queue, oft via Stack Pointer (festgelegtes Register) zur Vereinfachung der Verwaltung)
* Befehlsgruppen = { Einfache arithmetische & logische Befehle, Lade- & Speicherbefehle, bedingte & unbedingte Sprünge (Branch), Sonderfunktionen (Stack-Verwaltung, Interrupts (e.g. ESC-Interrupt)) }
* Adressierung/Operandenangabe: Absolut oder Indirekt (Registerinhalt enthalten Werte/Adressen), Offset (Wert/Adresse setzen sich zusammen aus Registerinhalt und im Befehl absolut angegebenem Wert)
* e.g. Prozessormodell: Wortbreite 2Byte(16Bit), Zahlendarstellung 2er-Kompl. (16Bit MSb/MSB ‘most significant bit/Byte’ je ganz links), Arbeitsspeicher 1KiB(210Bytes), Register (Befehlsregister, Befehlszähler, Akku, Arbeitsregister R1-R3, Carry-Flag), kein Cache, Zykluszeit 200ns, CPI 1

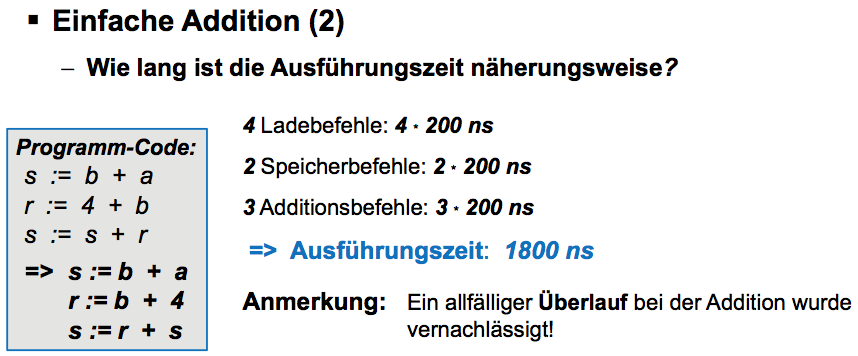


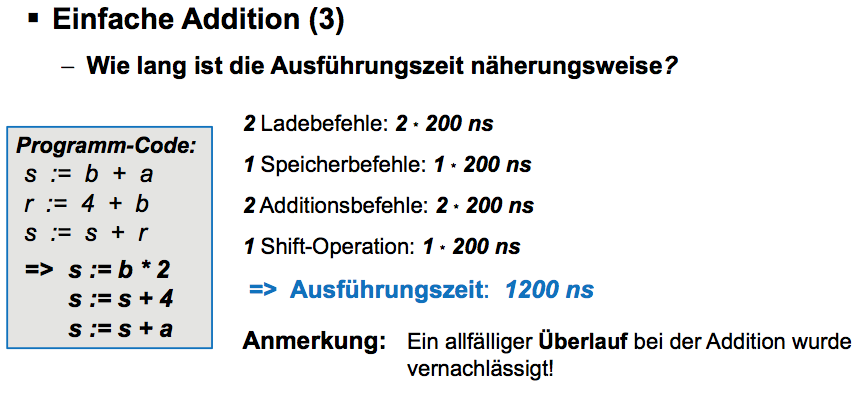
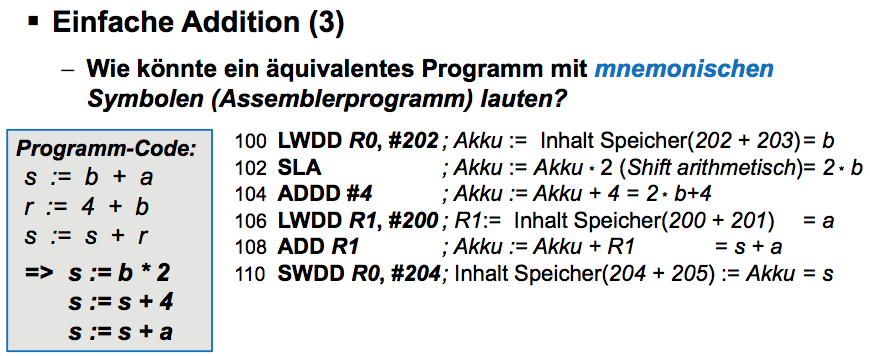
* Mnemonics: erheblich lesbarer als Maschinencode, Assembler-Compiler:=Compiler, der mnemonische Symbole in Maschinencode übersetzt, Assembler-Code:=Code auf Basis von Mnemonics, Assembler-Sprache:=Umfang der Symbole (des Codes)
* Codierung: Operanden (Mnemonics) und Optionen werden in Maschinensprache (festgelegte Bitfolge, i.d.R. Wortlänge V Vielfaches davon) codiert (bin. Code)
  + e.g. **00100**x<Adresse> // BD #Adr
  + e.g. **00100**000 0**1100100** // BD #100 (\*0🡪nicht relevant)
* Befehle alle gleich lang (e.g. 16 Bit) 🡪 vereinfach Realisierung, aber nicht zwingend nötig
* Beispiele-Programme

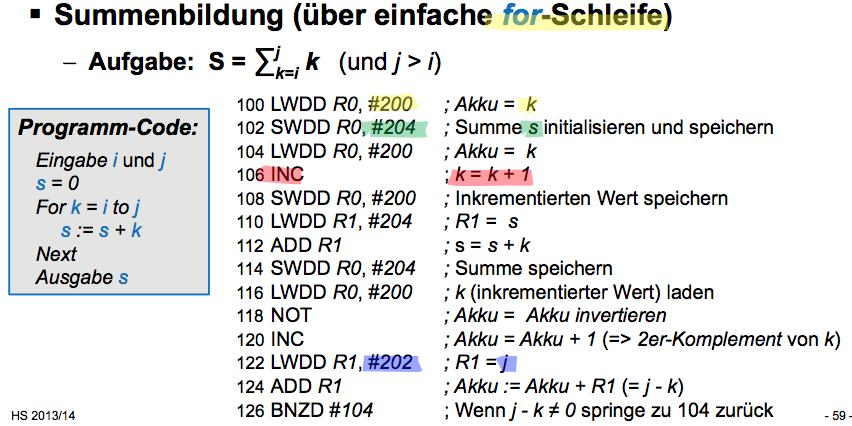
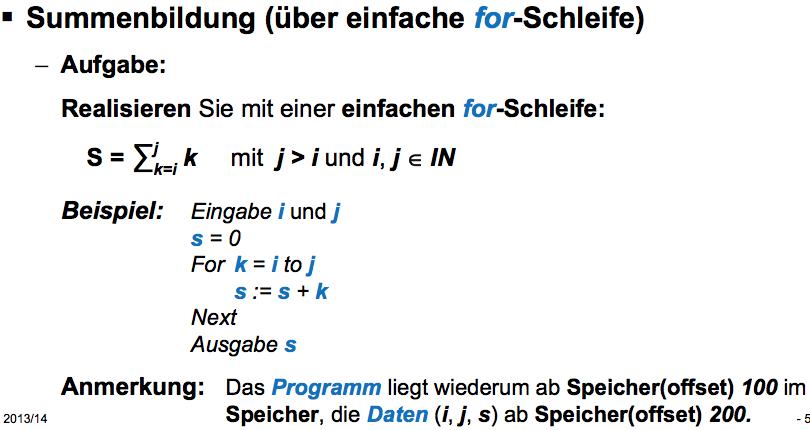


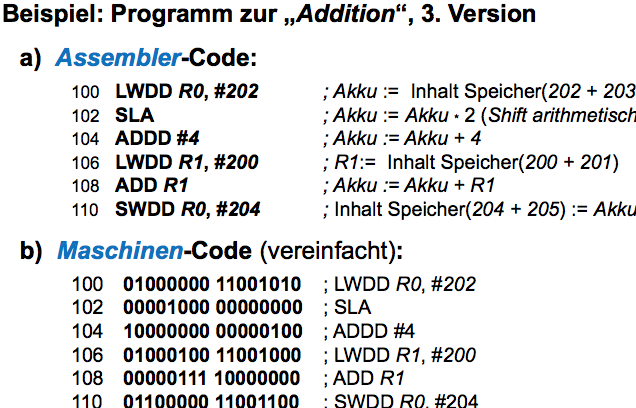












**04 Speicher**

• Hit Time ~ Zugriffszeit bei Treffer • Miss Penalty ~ Fehlzugriffaufwand

* RAM := Random Access Memory  – Random := Jede Information/Byte kann innert einer konstanten Zeit abgefragt werden, unabhängig von der Position im Speicher und Ab- hängigkeiten zur zuvor abgefragten Information/Byte
* SRAM := Static RAM
* benötigt 2-4 Transistoren/Bit
* SRAM-Transistoren können Informationen beliebig lange speichern,  solange sie unter Spannung stehen (kein Refresh wird benötigt).
* DRAM := Dynamic RAM –> heute verwendete Arbeitsspeicher- Technologie
* benötigt 1 Tranistor/Bit
* kleinere Speicherzellen als bei SRAM –> spart Platz
* aber: deutlich langsamer als SRAM
* und: benötigt Refreshs (i.d.R. alle 32 v 64ms) , wegen Leckströmen,  welche in Kondensatoren gespeicherte Ladungsmenge verändern kann

*Refresh-Analogie: Kübel mit Löcher, gefüllt mit Wasser, welcher in einem bestimmten Intervall wieder gefüllt werden muss (repräsentiert  ein Bit, welches true=1 ist), bevor das ganze Wasser ausgelaufen ist. DRAM-Transistoren verhalten sich genau gleich, da die Spannung nur kurz zwischengespeichert werden kann. Bevor die Spannung ver- lohren geht, muss der Transistor daher erneut aufgeladen werden.*

* SDRAM := Synchronized DRAM
* Synchronizes := welcher mit dem Systembus synchronisiert ist: Das SDRAM-Interface wartet auf das Clock-Signal des Buses, bevor es die Inputs verarbeitet.
* Die Befehle werden zudem via Pipeline übermittelt.
* Daher kann der Chip einen komplexeren Befehlssatz verarbeiten, was  ihn schneller als DRAM macht
* **MaskRom**:1xBeimFertigungProgrambar
* **PROM**:1xDurchÜberspannung
  + MetallDampf, spez.SW,HW nötig
* **EPROM:** mehrfach Löschbar:UV,teuer mehrer min.
* **EEPROM** : löschen mit software:Schnell, flexible, beständigkeit10Jahre, begrzte SchreibZyklen 1mio, Schreibvorgang langs. einige ms,
* **Flash-EEPROM**: (Flash Electrically Erasable Programmable ROM) gleich wie EEPROM nur schneller Schreiben(us) und günstiger, LeseSchreibenErfolgt Gruppenweise,nicht wahlfrei pro Byte, Geringerer Temperaturbereich (noch) gegenüber EEPROM

**05 Cache**

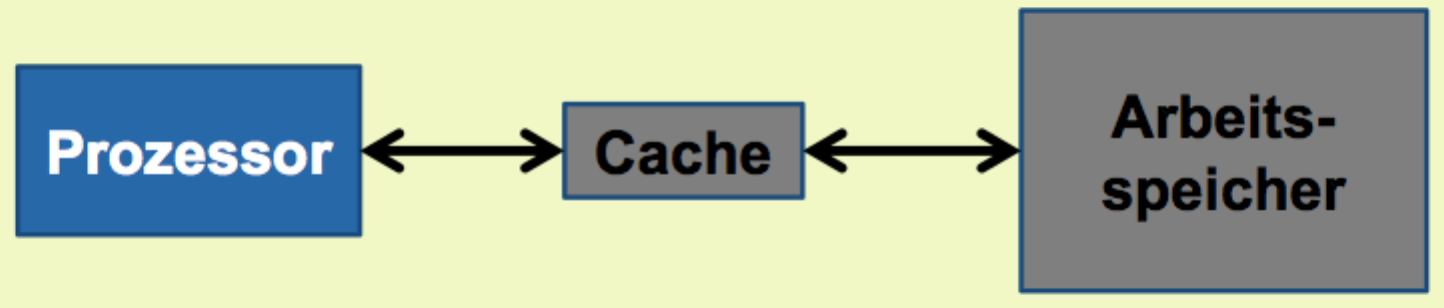
***Gründe für Cache***

* Optimierung des Speichers bringt am meisten Performance
* Mehr Taktrate ~ mehr Hitze, mehr Stromverbrauch, heute daher in der  Entwicklung nicht mehr so interessant
* Cache kann die Rechenleistung erheblich steigern -> Verbesserung der  Leistungsfähigkeit des Caches bringt erheblich mehr als Steigerung der Rechengeschwindigkeit (e.g. Taktrate)

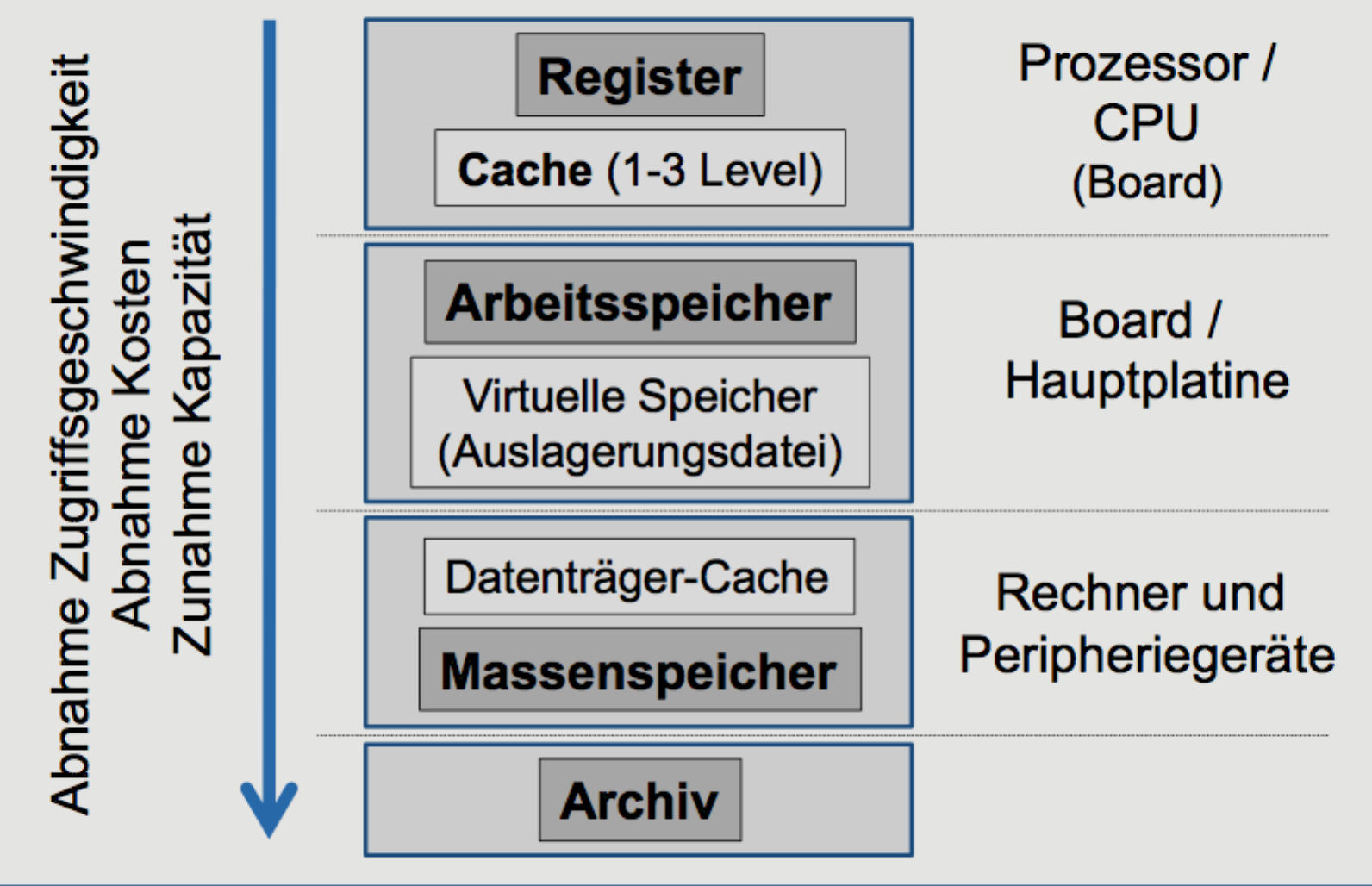
***Aufbau***

* CPU greift auf Daten im Cash statt RAM zu

*Daten werden zuvor aus RAM in Cache kopiert:*



* Komponenten
  + Register 🡪 SRAM (sehr schnelle HW)
  + Cache 🡪 SRAM, Assoziativ-Speicher
  + RAM 🡪 DRAM
  + Auslagerungsdatei 🡪 DRAM, Flash, schnelle HD, SSD
  + Datenträger-Cache 🡪 DRAM, SRAM (Controller @HD)
  + Massenspeicher 🡪 HD (magn./opt.), SSD, Flash=USB-Stick
  + Archiv 🡪 HD, CD, DVD, Tapes (magn.)

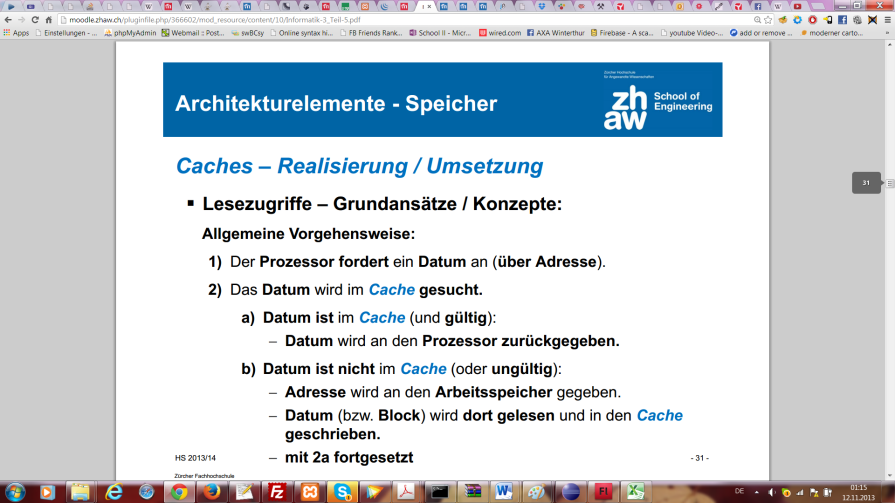


***CPU VS Cache VS RAM***

* CPU sehr schnell, RAM-Zugriff (heute DRAMs) langsam 🡪 nur wenige Daten können in Registern von Rechen-&Steuerwerk schnell bearbeitet werden 🡪 CPU muss sehr häufig auf Daten warten (e.g. Textbearbeitung, Metrizen, Audio, Video, Näherungsverfahren)
* Lösung: Zwischenspeicher=Cache
  + Dem Nutzer (Rechner) „viel“ kostengünstigen Speicher mit schnellen Zugriffszeiten (vergleichbar der schnellsten Speichertechnologie) zur Verfügung zu stellen.
* Zugriffszeiten: (schnell) Register < Cache < RAM (langsam) 🡪 Cache etwas langsamer als Register aber erheblich schneller als RAM
* Grösse: (klein) Register < Cache < RAM (gross) 🡪 Cache deutlich grösser als Register, deutl. kl. als RAM

*Dieser Ansatz lässt sich auf alle Speicherebenen übertragen/nutzen!*

* Cache: Strukturen erheblich komplexer (mehr Bauteile, Daten müssen vom Register und RAM zu Cache übertragen werden)

Y Cache efficiant? 🡪 ***zeitl. Loklalität*** (Daten, die zuletzt genutzt werden, e.g. Befehle in Programmschleifen, Variablen), ***räuml. Lokalität*** (Daten in räuml. Nähe (auf Datenträger) des zugegriffenen Datums werden in Cache kopiert, e.g. Program Loops, Feld-Elemente, Text Search)

**a) Assoziativer Zugriff** – Schema

 Bei einem Assoziativspeicher wird die Adresse des gesuchten

Datums mit allen Adressen der sich im Speicher befindenden

Daten parallel verglichen

und

Falls eine Adresse übereinstimmt das Datum unmittelbar

gelesen / geschrieben.

 Dazu muss zu jedem Datum zusätzlich die Adresse abgespeichert werden (wird allgemein als Tag bezeichnet).

>Jeder Block beinhaltet ein zusätzliche Bit (**valid bit**), das angibt, ob die Daten gültig sind oder nicht.

**b)Direktabbildend** – Schema:

 Bei einem direktabbildenden Speicher wird jeder Adresse im

Speicher genau eine Adresse (Position) im Cache zugeordnet.

Die Abbildung ist sehr einfach – in der Regel über die „modulo“-Funktion realisiert: **Cache-Adresse = (Block-Adresse) modulo (Anzahl Blöcke im Cache)**

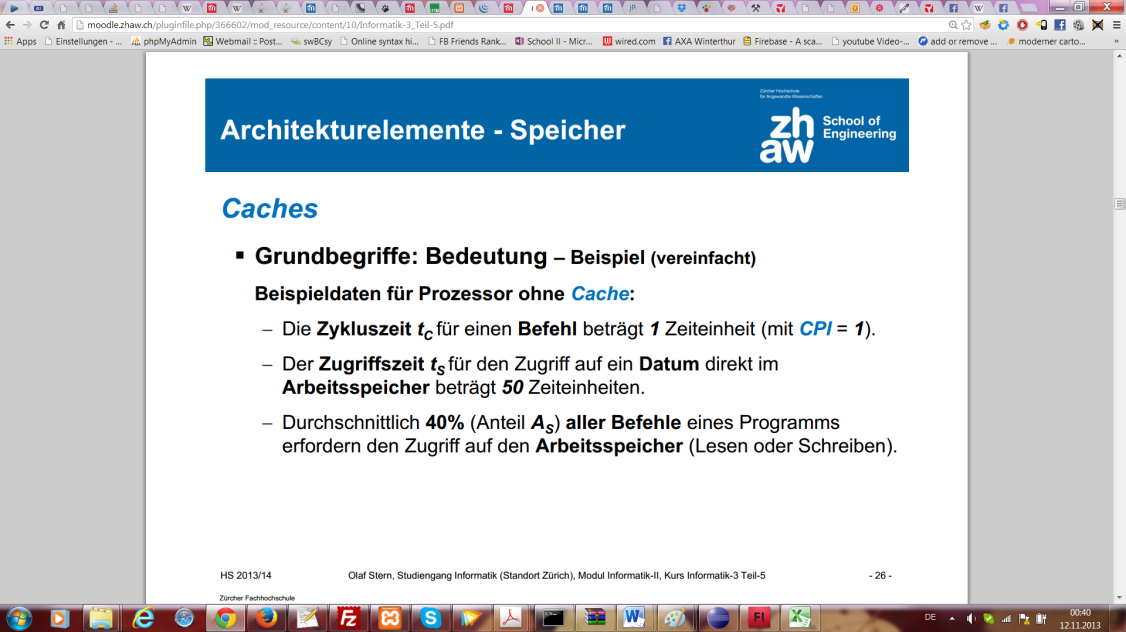
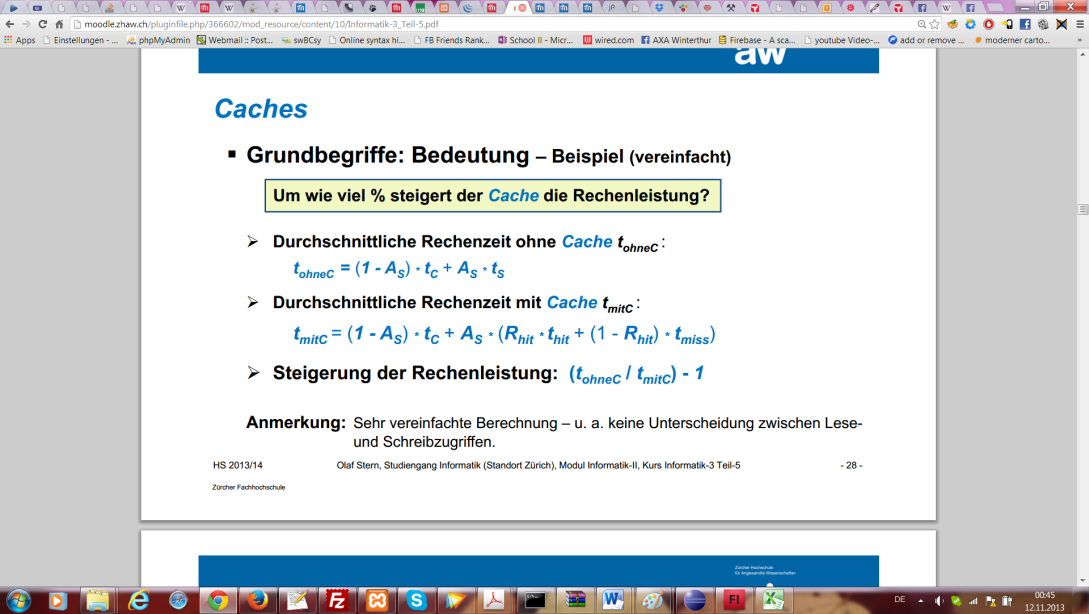
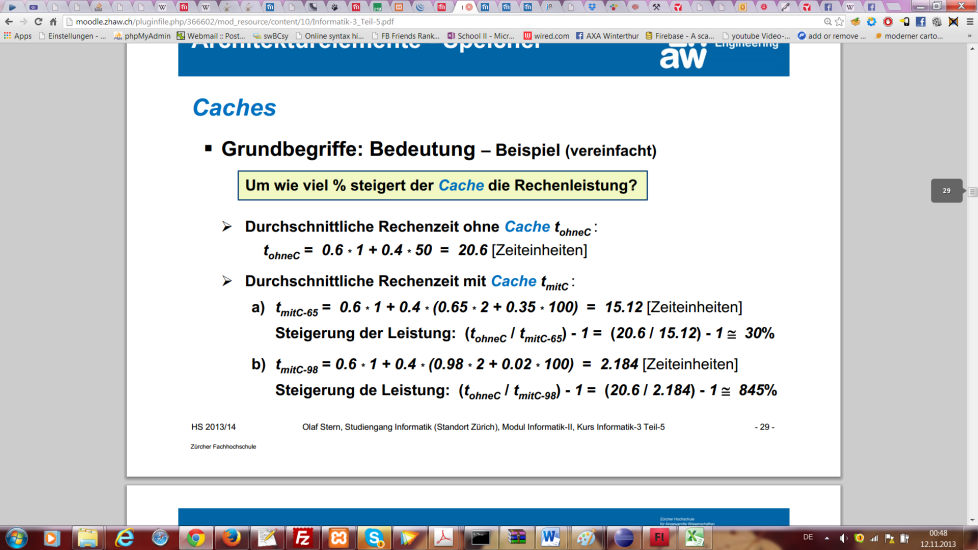
 Arbeitsspeicher mit 32 Blöcken

 Cache mit 8 Blöcken

 Block z. B. 4 Byte

Die Blockgrösse ist i. d. R. ein Vielfaches der Wortgrösse:

**Blockgrösse = Wortgrösse \*2m, m £ IN**

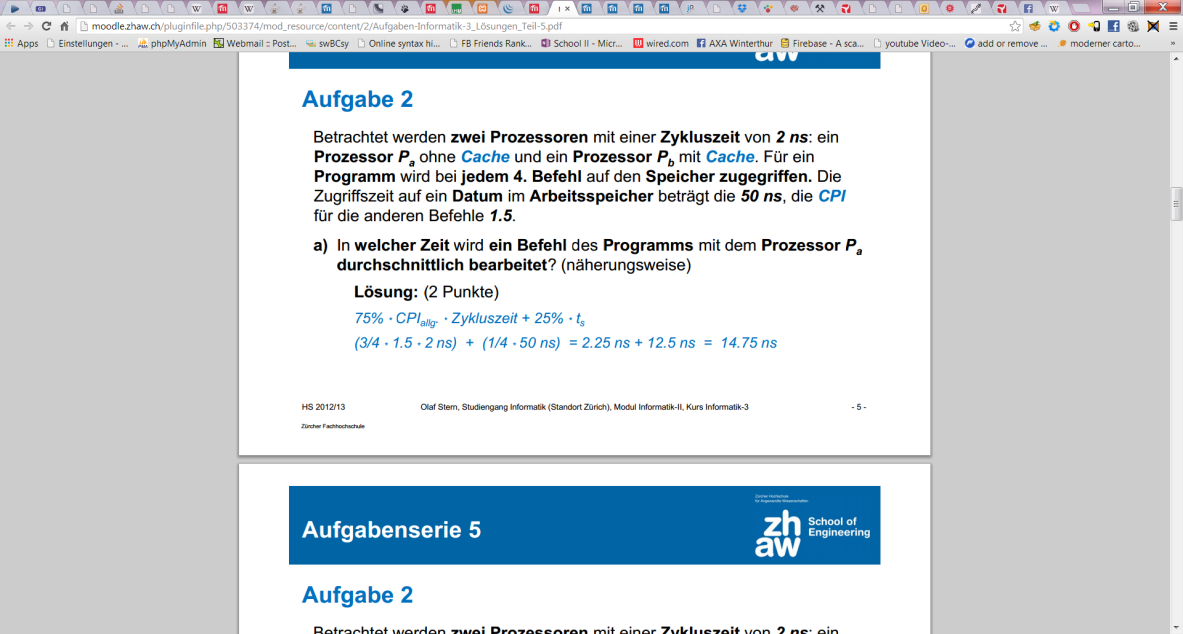
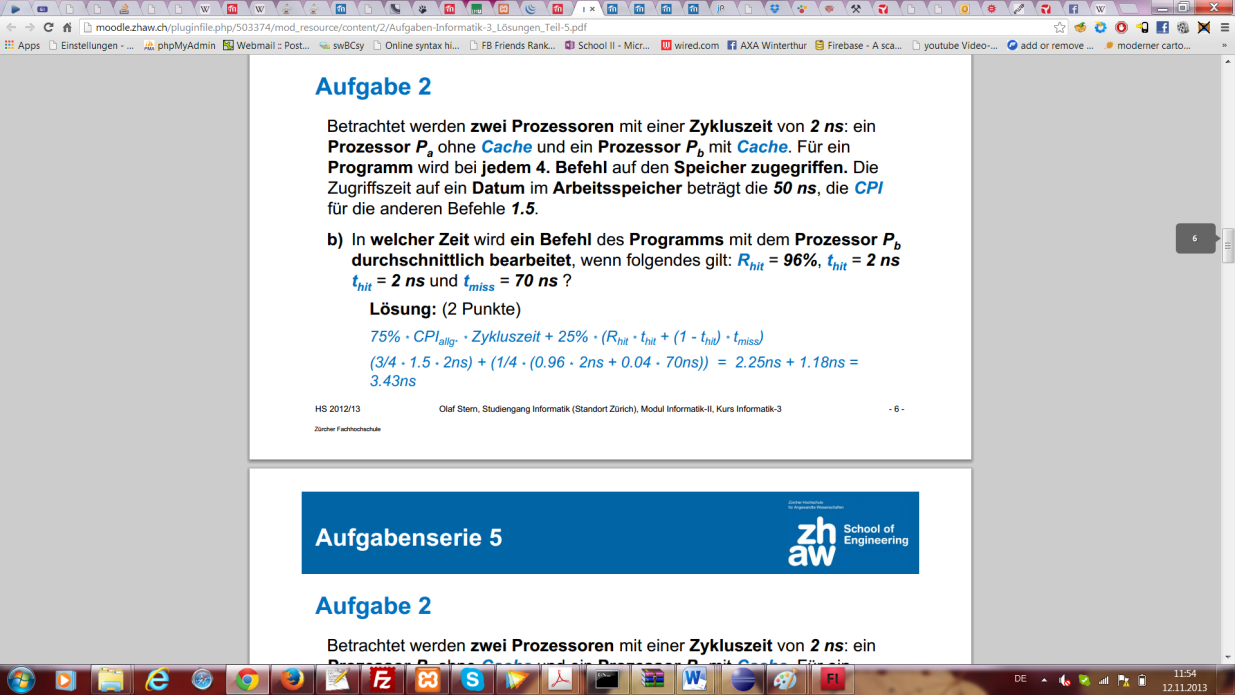


**b) Direktabbildend – Beispiel Grössenberechnung**

Gegeben sei für einen Arbeitsspeicher mit 232Byte ein direktabbildender Cache mit 4 KiB Daten(+), einer Blockgrösse von 4 Wörtern und einer Wortlänge von 32 Bit (4 Byte).

 Wie gross ist die tatsächliche Anzahl von Bits für den Cache?

(+)Zur Erinnerung: 4 KiB bedeuten exakt 212 = 4096 Byte, leider wird umgangssprachlich auch häufig 4 KB gesagt.



**„Hit Rate“ Rhit** (Trefferrate):

Anteil der Speicherzugriffe auf einen Cache, die zu einem Treffer

führen (Daten befinden sich im Cache).

**„Miss Rate“ Rmiss**(Fehlzugriffsrate):

Anteil der Speicherzugriffe auf einen Cache, die nicht zu einem

Treffer führen (Daten befinden sich nicht im **Cache**).

***Es gilt:* Rmiss= 1 – Rhit**

**„Hit Time“ thit** (Zugriffszeit bei Treffer):

Erforderliche Zeit für den erfolgreichen Zugriff auf ein Datum(+) im Cache, inkl. der Zeit, die für den Test erforderlich ist, ob das Datum im Cache vorhanden ist.

**„Miss Penalty“ tmiss**(Fehlzugriffsaufwand):

Erforderliche Zeit für den Austausch eines Blocks(+) im Cache aus der nächsten Ebene (z. B. dem Arbeitsspeicher), inkl. der Zeit, diesen Block Nutzer / Prozessor zur Verfügung zu stellen.

