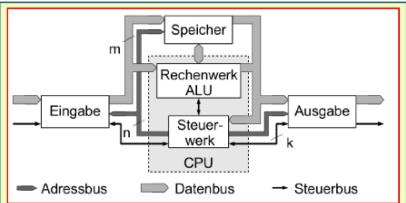


## 1 Grundlegende Rechenarchitektur

**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



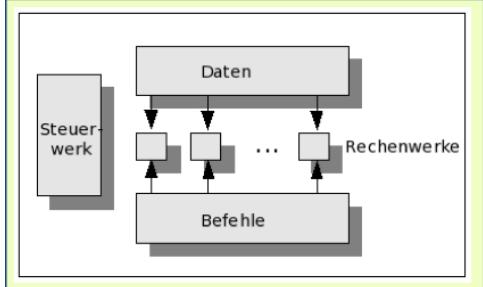
- Steuerwerk/Control Unit:=Herzstück d. Rechners, interpretiert Programmcode, erstellt Verbindungen zwischen Daten & Rechenwerk, Steuert Reihenfolge d. Programmbefehle, beinhaltet {Befehlsregister, Befehlsdecoder, Befehlszähler}
- Rechenwerk/ALU:=arithmetische&logische Rechen-Operationen
- Speicherwerk:=Speichert 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 → 2<sup>32</sup> 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
  - Der aktuelle Befehl wird aus der Speicherzelle, auf die der Befehlszähler zeigt, ausgelesen und in das Steuerwerk übertragen
  - Das Steuerwerk dekodiert den Befehl und schaltet die entsprechenden Signale auf den Steuerleitungen
  - Die für den Befehl erforderlichen Operanden werden aus dem Speicherwerk gelesen und in das Rechenwerk bzw. die festgelegten Register übertragen
  - Die dekodierte Operation wird ausgeführt; das Ergebnis in ein Register (oder den Speicher) geschrieben
  - Der Befehlszähler wird um eins erhöht oder auf Grund eines Sprung-Befehls um einen anderen Wert verändert
  - 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 Programmprüfungen (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 → Σ 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)
- a) Architektur, Hardware, Befehlssatz so designen, dass alle Befehle gleich (d.h. Datenpfade sind gleich lang) lange brauchen (zur Ausführung) → in Realität nicht umsetzbar
- b) Befehle in Gruppen unterteilen, die in etwa gleich lang brauchen (für Ausführung) → erfolgreiches Konzept, Optimierung für häufig genutzte Befehle möglich, Steuerung/Design aber komplexer
- c) 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

Schritt:	1	2	3	4	5	6	...
Befehl 1 (Load)	Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>			
Befehl 2 (Addition)		Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>		
Befehl 3 (Addition)			Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>	
Befehl 4 (Load)				Befehl laden*	Befehl decode	Operanden bereitstel.	Operation ausführen <sup>+</sup>
Befehl 5 (ODER)					Befehl laden*	Befehl decode	Operanden bereitstel.
Befehl 6 (Store)						Befehl laden*	Befehl decode
...							

→ Σ 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
Σ 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):  $T = (k + n - 1) * \text{Zykluszeit}$ 
  - o e.g.  $T = (7+20-1)*0.3125\text{ns}=0.00825\mu\text{s}$  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 Programmabföhrung 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. #Anz. Taktzyklen, welche ein Befehl zur Ausführung benötigt) → Abh. von CPU-Implementierung
  - Ausführungszeit P:  $P = (\text{Anz. Befehle}) * \text{Zykluszeit} * \text{CPI}$  = 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, 2^n 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, 2^n Ausgänge} → (Eingang wird via Steuereingänge mit genau einem Ausgang verbunden/durchgeschaltet)
  - Optimierte Schaltungen
    - e.g. Addierer (theoret. realisierbar über bools. Schaltungen, in Praxis aber Addierwerke := spezielle bzl. Geschwindigkeit optimierte Schaltungen)
    - e.g. Optimierte Schaltungen: Halb- & Volladdierer, 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 flankengesteuerte 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 auszuführenden 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)
    - e.g. 1 OP-Code, Operand-1, Operand-2, Operand-3, (Option)
    - e.g. 2 ADD, a, b, s, - // s=a+b, keine Option (-)
    - e.g. 3 OR,u,v,w,- // w=uOrv, keineOption(-)
    - 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
    - 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
    - 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 Addressierung (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 to 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. Dennoch werden verschiedene 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
      - Bedingung berechnen und Resultat in R1 schreiben
      - R1 != 0?
        - 1 → kein Sprung (Programmfortsetzung & Ausführung der Befehle des Schleifenkörpers C1 & Sprung nach C3)
        - 0 → Sprung nach Schleifenkörper (C2)
    - e.g. WHILE "branch if not null" // while(!0) {C1}; C2
      - Bedingung berechnen und Resultat in R1 schreiben
      - R1 != 0?
        - 1 → kein Sprung (Programmfortsetzung & Ausführung der Befehle des Schleifenkörpers C1 & Sprung zu while-Condition)
        - 0 → Sprung nach C2
    - e.g. FOR // do {C1} for(i=0; R1>0; R1--)
      - Schleifenzähler initialisieren
      - Schleifenkörper (C1) durchlaufen
      - Schleifenzähler (R1) um 1 reduzieren
      - R1 = 0?
        - 1 → kein Sprung // exit
        - 0 → Sprung zu C1 // GoTo C1
    - e.g. CASE SWITCH → Folge von IF-Anweisungen V Tabelle mit Sprungadressen, die zuvor iniidiert 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)) }
    - Addressierung/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/y' je ganz links), Arbeitsspeicher 1KiB(2^10Bytes), Register (Befehlsregister, Befehlszähler, Akku, Arbeitsregister R1-R3, Carry-Flag), kein Cache, Zykluszeit 200ns, CPI 1
  - MSB / LSB**  
↳ **MSB** = Most/Least significant Bit  
*e.g. 1 00001011*  
↳ **LSB**
  - 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. 00100x <Adresse> // BD #Addr
    - e.g. 00100000 01100100 // BD #100 (\*0→nicht relevant)
  - Befehle alle gleich lang (e.g. 16 Bit) → vereinfach Realisierung, aber nicht zwingend nötig
  - Beispiele-Programme
  - Einfache Addition**
    - Wie könnte ein äquivalentes Programm mit mnemonischen Symbolen (Assemblerprogramm) lauten?
  - Programm-Code:**

```

100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
102 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a
104 ADD R1 ; Akku := Akku + R1 = s
106 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
108 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
110 ADD #4 ; Akku := Akku + 4 = r
112 SWDD R0, #206 ; Inhalt Speicher(206 + 207) := Akku = r
114 LWDD R0, #204 ; Akku := Inhalt Speicher(204 + 205) = s
116 LWDD R1, #206 ; R1 := Inhalt Speicher(206 + 207) = r
118 ADD R1 ; Akku := Akku + R1 = s
120 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s

```

## ▪ Einfache Addition

- Wie lang ist die Ausführungszeit näherungsweise?

**Programm-Code:**

```
s := b + a
r := 4 + b
s := s + r
```

5 Ladebefehle: 5 · 200 ns  
 3 Speicherbefehle: 3 · 200 ns  
 3 Additionsbefehle: 3 · 200 ns  
**=> Ausführungszeit: 2200 ns**

Anmerkung: Ein allfälliger Überlauf bei der Addition wurde vernachlässigt!

## ▪ Einfache Addition (2)

- Wie könnte ein äquivalentes Programm mit **mnemonischen Symbolen** (Assemblerprogramm) lauten?

**Programm-Code:**

```
s := b + a
r := 4 + b
s := s + r
=> s := b + a
r := b + 4
s := r + s
```

100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
 102 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a
 104 ADD R1 ; Akku := Akku + R1 = s
 106 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
 108 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
 110 ADDD #4 ; Akku := Akku + 4 = r
 112 LWDD R1, #204 ; R1 := Inhalt Speicher(204 + 205) = s
 114 ADD R1 ; Akku := Akku + R1 = s
 116 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s

## ▪ Einfache Addition (2)

- Wie lang ist die Ausführungszeit näherungsweise?

**Programm-Code:**

```
s := b + a
r := 4 + b
s := s + r
=> s := b + a
r := b + 4
s := r + s
```

4 Ladebefehle: 4 · 200 ns  
 2 Speicherbefehle: 2 · 200 ns  
 3 Additionsbefehle: 3 · 200 ns  
**=> Ausführungszeit: 1800 ns**

Anmerkung: Ein allfälliger Überlauf bei der Addition wurde vernachlässigt!

## ▪ Einfache Addition (3)

- Wie könnte ein äquivalentes Programm mit **mnemonischen Symbolen** (Assemblerprogramm) lauten?

**Programm-Code:**

```
s := b + a
r := 4 + b
s := s + r
=> s := b * 2
s := s + 4
s := s + a
```

100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
 102 SLA ; Akku := Akku - 2 (Shift arithmetisch) = 2 · b
 104 ADDD #4 ; Akku := Akku + 4 = 2 · b + 4
 106 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a
 108 ADD R1 ; Akku := Akku + R1 = s + a
 110 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s

## ▪ Einfache Addition (3)

- Wie lang ist die Ausführungszeit näherungsweise?

**Programm-Code:**

```
s := b + a
r := 4 + b
s := s + r
=> s := b * 2
s := s + 4
s := s + a
```

2 Ladebefehle: 2 · 200 ns  
 1 Speicherbefehle: 1 · 200 ns  
 2 Additionsbefehle: 2 · 200 ns  
 1 Shift-Operation: 1 · 200 ns  
**=> Ausführungszeit: 1200 ns**

Anmerkung: Ein allfälliger Überlauf bei der Addition wurde vernachlässigt!

## ▪ Summenbildung (über einfache for-Schleife)

### - Aufgabe:

- Realisieren Sie mit einer **einfachen for-Schleife**:

$$S = \sum_{k=j}^i k \quad \text{mit } j > i \text{ und } i, j \in IN$$

**Beispiel:** Eingabe i und j  
 s = 0  
 For k = i to j  
 s := s + k  
 Next  
 Ausgabe s

Anmerkung: Das Programm liegt wiederum ab Speicher(offset) 100 im Speicher, die Daten (i, j, s) ab Speicher(offset) 200.

## ▪ Summenbildung (über einfache for-Schleife)

- Aufgabe:  $S = \sum_{k=j}^i k$  (und  $j > i$ )

**Programm-Code:**

```
Eingabe i und j
s = 0
For k = i to j
s := s + k
Next
Ausgabe s
```

100 LWDD R0, #200 ; Akku = k
 102 SWDD R0, #204 ; Summe s initialisieren und speichern
 104 LWDD R0, #200 ; Akku = k
 106 INC ; k = k + 1
 108 SWDD R0, #200 ; Inkrementierten Wert speichern
 110 LWDD R1, #204 ; R1 = s
 112 ADD R1 ; s = s + k
 114 SWDD R0, #204 ; Summe speichern
 116 LWDD R0, #200 ; (inkrementierter Wert) laden
 118 NOT ; Akku = Akku invertieren
 120 INC ; Akku = Akku + 1 (= 2er-Komplement von k)
 122 LWDD R1, #202 ; R1 = j
 124 ADD R1 ; Akku = Akku + R1 (= j - k)
 126 BNZD #104 ; Wenn j - k ≠ 0 springe zu 104 zurück

HIS 2013/14 Beispiel: Programm zur „Addition“, 3. Version

### a) Assembler-Code:

100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203)
 102 SLA ; Akku := Akku - 2 (Shift arithmetisch)
 104 ADDD #4 ; Akku := Akku + 4
 106 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201)
 108 ADD R1 ; Akku := Akku + R1
 110 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku

### b) Maschinen-Code (vereinfacht):

100 01000000 11001010 ; LWDD R0, #202
 102 00001000 00000000 ; SLA
 104 10000000 00000100 ; ADDD #4
 106 00100010 11001000 ; LWDD R1, #200
 108 00000111 00000000 ; ADD R1
 110 01100000 11001100 ; SWDD R0, #204

## 04 Speicher

- Hit Time ~ Zugriffszeit bei Treffer • Miss Penalty ~ Fehlzugriffaufwand
- RAM := Random Access Memory – Random := Jede Information/Byte kann innerhalb einer konstanten Zeit abgefragt werden, unabhängig von der Position im Speicher und Abhängigkeiten zur zuvor abgefragten Information/Byte

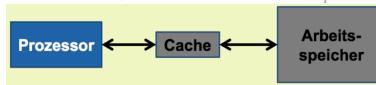
- SRAM := Static RAM
  - benötigt 2-4 Transistoren/Bit
  - SRAM-Transistoren können Informationen beliebig lange speichern, solange sie unter Spannung stehen (kein Refresh wird benötigt).
- DRAM := Dynamic RAM → heute verwendete Arbeitsspeicher-Technologie
  - benötigt 1 Transistor/Bit
  - kleinere Speicherzellen 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 Löchern, 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 verloren 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 Befehlsatz verarbeiten, was ihn schneller als DRAM macht
  - MaskRom: 1xBeimFertigungProgrammbar
  - PROM: 1xDurchÜberspannung MetallDampf, spez.SW,HW nötig
  - EPROM: mehrfach Löschbar:UV, teuer mehrer min.
  - EEPROM: löschen mit software:Schnell, flexible, beständigkeit10Jahre, begrenzte SchreibZyklen 1mio, Schreibvorgang lang. 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

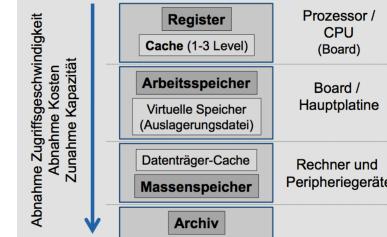
## 05 Cache

### 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 Cache 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, Metriken, Audio, Video, Navigationsverfahren)
- 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 effizient? → **zeitl. Lokalität** (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,

#### Lesezugriffe – Grundansätze / Konzepte:

##### Allgemeine Vorgehensweise:

- 1) Der Prozessor fordert ein Datum an (über Adresse).
- 2) Das Datum wird im **Cache** gesucht.
  - a) Datum ist im **Cache** (und gültig):
    - Datum wird an den Prozessor zurückgegeben.
  - b) Datum ist nicht im **Cache** (oder ungültig):
    - Adresse wird an den Arbeitsspeicher gegeben.
    - Datum (bzw. Block) wird dort gelesen und in den **Cache** geschrieben.
    - mit 2a fortgesetzt

13/14

Feld-Elemente, Text Search)

#### a) Assoziativer Zugriff – Schema

Ø 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.

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

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

#### b) Direktabbildend – Schema:

Ø 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:

$$\text{Blockgröße} = \text{Wortgröße} * 2^m, m \in \mathbb{N}$$

#### b) Direktabbildend – Beispiel Größenberechnung

Gegeben sei für einen Arbeitsspeicher mit  $2^{32}$  Byte ein direktabbildender Cache mit 4 KiB Daten(+), einer Blockgröße von 4 Wörtern und einer Wortlänge von 32 Bit (4 Byte).

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

(+) Zur Erinnerung: 4 KiB bedeuten exakt  $2^{12} = 4096$  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“ t<sub>hit</sub> (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“ t<sub>miss</sub> (Fehlzugriffsauwand):

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.

#### b) Direktabbildend – Grundprinzip (Bsp. Abbildung)

**Cache** mit **Tags** (des Beispiels):

– Gesucht der Eintrag für die Block-Adresse: „0 10 1“ Index

Index	Tag	Datenblöcke
0 0 0	0 0	„Daten: 32 Bit“
0 0 1	0 1	„Daten: 32 Bit“
0 1 0	1 1	„Daten: 32 Bit“
0 1 1	0 1	„Daten: 32 Bit“
1 0 0	0 1	„Daten: 32 Bit“
1 0 1	1 1	„Daten: 32 Bit“
1 1 0	1 1	„Daten: 32 Bit“
1 1 1	0 0	„Daten: 32 Bit“

#### b) Direktabbildend – Beispiel Adressberechnung

Gegeben sei ein **Cache** mit 32 Blöcken von je 4 Bytes

➤ Auf welchen Block (**Tag**) im **Cache** wird das Byte im Arbeitsspeicher mit der Adresse 2410 abgebildet? (Beginn bei Adr. 0)

$$\text{Blockadresse} = [\text{Byteadresse} / (\text{Bytes pro Block})]$$

$$\text{Blockadresse} = [(2410 + 1) / 4] = [602.75] = 602$$

$$\text{Cache-Blocknummer (Tag)} = (\text{Blockadresse}) \bmod (\text{Anzahl Blöcke im Cache})$$

$$\text{Cache-Blocknummer (Tag)} = 602 \bmod 32 = 26$$

(es ist dort das 4. Byte, 2407 ist das 1. Byte)

#### ▪ Grundbegriffe: Bedeutung – Beispiel (vereinfacht)

##### Beispieldaten für Prozessor ohne Cache:

- Die **Zykluszeit** t<sub>c</sub> für einen Befehl beträgt 1 Zeiteinheit (mit **CPI = 1**).
- Der **Zugriffszeit** t<sub>s</sub> für den Zugriff auf ein **Datum** direkt im Arbeitsspeicher beträgt 50 Zeiteinheiten.
- Durchschnittlich 40% (Anteil A<sub>s</sub>) aller **Befehle** eines Programms erfordern den Zugriff auf den **Arbeitsspeicher** (Lesen oder Schreiben).

##### Beispieldaten für Prozessor mit Cache:

- Die **Hit Time** t<sub>hit</sub> beträgt 2 Zeiteinheiten.
- Der **Fehlzugriffsauwand** t<sub>miss</sub> beträgt 100 Zeiteinheiten:
  - a) für Test, ob Datum im **Cache** steht,
  - b) um Datum im **Arbeitsspeicher** zu adressieren / anzusprechen und
  - c) um **Datum/Bock** in den **Cache** zu übertragen.
- Die **Hit Rate R<sub>hit</sub>** beträgt a) 65% bzw. b) 98%.

#### ▪ Grundbegriffe: Bedeutung – Beispiel (vereinfacht)

##### Um wie viel % steigt der **Cache** die Rechenleistung?

➤ Durchschnittliche Rechenzeit ohne Cache t<sub>ohneC</sub>:

$$t_{\text{ohneC}} = (1 - A_s) * t_c + A_s * t_s$$

➤ Durchschnittliche Rechenzeit mit Cache t<sub>mitC</sub>:

$$t_{\text{mitC}} = (1 - A_s) * t_c + A_s * (R_{\text{hit}} * t_{\text{hit}} + (1 - R_{\text{hit}}) * t_{\text{miss}})$$

➤ Steigerung der Rechenleistung:  $(t_{\text{ohneC}} / t_{\text{mitC}}) - 1$

##### Um wie viel % steigt der **Cache** die Rechenleistung?

➤ Durchschnittliche Rechenzeit ohne Cache t<sub>ohneC</sub>:

$$t_{\text{ohneC}} = 0.6 * 1 + 0.4 * 50 = 20.6 \text{ [Zeiteinheiten]}$$

➤ Durchschnittliche Rechenzeit mit Cache t<sub>mitC</sub>:

$$a) t_{\text{mitC-65}} = 0.6 * 1 + 0.4 * (0.65 * 2 + 0.35 * 100) = 15.12 \text{ [Zeiteinheiten]}$$

Steigerung der Leistung:  $(t_{\text{ohneC}} / t_{\text{mitC-65}}) - 1 = (20.6 / 15.12) - 1 \approx 30\%$

$$b) t_{\text{mitC-98}} = 0.6 * 1 + 0.4 * (0.98 * 2 + 0.02 * 100) = 2.184 \text{ [Zeiteinheiten]}$$

Steigerung der Leistung:  $(t_{\text{ohneC}} / t_{\text{mitC-98}}) - 1 = (20.6 / 2.184) - 1 \approx 845\%$

Betrachtet werden zwei Prozessoren mit einer Zykluszeit von 2 ns: ein **Prozessor P<sub>a</sub>** ohne **Cache** und ein **Prozessor P<sub>b</sub>** mit **Cache**. Für ein Programm wird bei jedem 4. Befehl auf den Speicher zugegriffen. Die Zugriffszeit auf ein Datum im **Arbeitsspeicher** beträgt die 50 ns, die **CPI** für die anderen Befehle 1.5.

a) In welcher Zeit wird ein Befehl des Programms mit dem **Prozessor P<sub>a</sub>** durchschnittlich bearbeitet? (näherungsweise)

Lösung: (2 Punkte)

$$75\% * \text{CPI}_{\text{avg}} * \text{Zykluszeit} + 25\% * t_s$$

$$(3/4 * 1.5 * 2ns) + (1/4 * 50ns) = 2.25ns + 12.5ns = 14.75ns$$

b) In welcher Zeit wird ein Befehl des Programms mit dem **Prozessor P<sub>b</sub>** durchschnittlich bearbeitet, wenn folgendes gilt: R<sub>hit</sub> = 96%, t<sub>hit</sub> = 2 ns, t<sub>miss</sub> = 2 ns und t<sub>miss</sub> = 70 ns?

Lösung: (2 Punkte)

$$75\% * \text{CPI}_{\text{avg}} * \text{Zykluszeit} + 25\% * (R_{\text{hit}} * t_{\text{hit}} + (1 - R_{\text{hit}}) * t_{\text{miss}})$$

$$(3/4 * 1.5 * 2ns) + (1/4 * (0.96 * 2ns + 0.04 * 70ns)) = 2.25ns + 1.18ns = 3.43ns$$

c) Um wie viel % steigt der **Cache** des **Prozessor P<sub>b</sub>** die Rechenleistung?

Lösung: (2 Punkte)

$$(14.75ns / 3.43ns) - 1 \approx 3.3 \text{ (d. h. um 330%; fast 4.5x so schnell)}$$

