Karlsruher Institut für Technologie

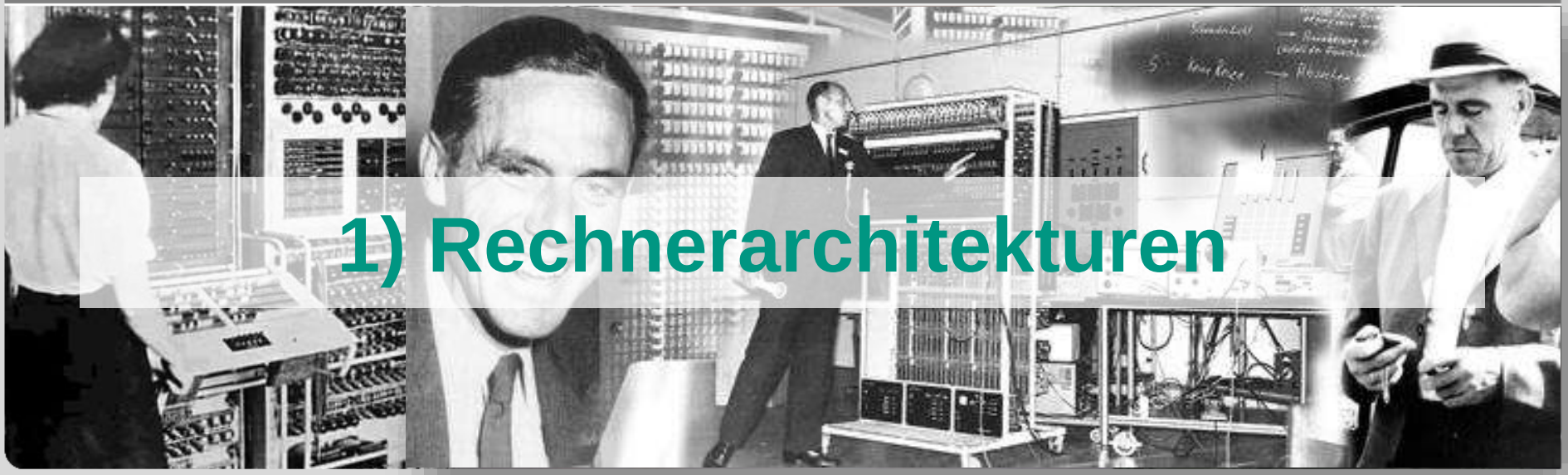
Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Vorlesung Informationstechnik (IT)

Sommersemester 2018

Institut für Technik der Informationsverarbeitung (ITIV)



1) Rechnerarchitekturen

IT-Veranstaltungsplan SS2018

Karlsruher Institut für Technologie

Veranstaltungsplan Informationstechnik SS2018

Vorlesung/Übung:	Mi, 09:45-11:15	10.21 Benz (HMU)
Vorlesung/Übung:	Do, 14:00-15:30	30.46 Neue Chemie
Praktikum:	28.Mai-13.Juli	SCC-Poolräume

Woche	Datum Mi.	Vorlesung	Datum Do.	Übung	Praktikum	Inhalt Mi.	Inhalt Do.
1.	18. Apr	VL	19. Apr			VL0: Einleitung und Motivation	
2.	25. Apr	VL	26. Apr			VL1: Rechnerarchitekturen	
3.	02. Mai	VL	03. Mai	ÜB		VL2: Rechnerarchitekturen	ÜB1: Organisatorisches; Besprechung Aufg. 1; C++ Grundlagen
4.	09. Mai	VL	10. Mai	frei	Einführungsveranstaltung I	VL3: Rechnerarchitekturen	Himmelfahrt
5.	16. Mai	VL	17. Mai		Einführungsveranstaltung II	VL4: Programmiersprachen, Weg zum ausführbaren Programm	
6.	23. Mai	frei	24. Mai	frei		Pfingsten	Pfingsten
7.	30. Mai		31. Mai	frei			Fronleichnam
8.	06. Jun	VL	07. Jun	ÜB	Projektmanagement	VL5: Programmstrukturen	ÜB2: Besprechung Aufg. 2; Funktionen, Zeiger, Strings
9.	13. Jun	VL	14. Jun	ÜB	Programmieren	VL6: SW-Engineering	ÜB3: Besprechung Aufg. 3; Header, Ablauf- und Nassi-Shneiderman Diagramm
10.	20. Jun	VL	21. Jun	VL	Programmieren	VL7: Objektorientierung	VL8: Datenstrukturen
11.	27. Jun	VL	28. Jun	ÜB	Programmieren	VL9: Algorithmen	ÜB4: Besprechung Aufg. 4; Objektorientierung, Vererbung, Klassendiagramm
12.	04. Jul	VL	05. Jul	ÜB	Test	VL10: Sortieralgorithmen	ÜB5: Besprechung Aufg. 5; Dynamische Speicherverwaltung, Polymorphie, STL
13.	11. Jul	VL	12. Jul	ÜB	Test	VL11: Suchalgorithmen	ÜB6: Besprechung Aufg. 6; Sortieralgorithmen, Tiefensuche
14.	18. Jul	VL	19. Jul	ÜB	Probefahrt	VL12: Optimierungsalgorithmen	ÜB7: Besprechung Aufg. 7; Optimierungsalgorithmen, Q&A

VL	13
ÜB	7
Praktikum	9

Einführungsveranstaltung I: Di. 08.05.2018, 17:30-19:00, HSaF
Einführungsveranstaltung II: Di. 15.05.2018, 17:30-19:00, HSaF

KW= Kalenderwoche; SW = Semesterwoche; VL = Vorlesung; ÜB = Übung; T= Tutorium; P= Praktikum; T4 u. T5 wertvoll für P

IT Inhalt

0. Einleitung und Motivation

- Organisatorisches
- Begriffe
- Typische Anwendungen

1. Rechnerarchitekturen

- Einführung
- Allgemeiner Aufbau der internen Hardware Architektur
- Instruction Set Architecture (ISA) am Beispiel ARM Cortex M4
- Klassifikation von Rechnerarchitekturen
- Performanzsteigerung
- ausgewählte Beispiele

Informationstechnische Systeme

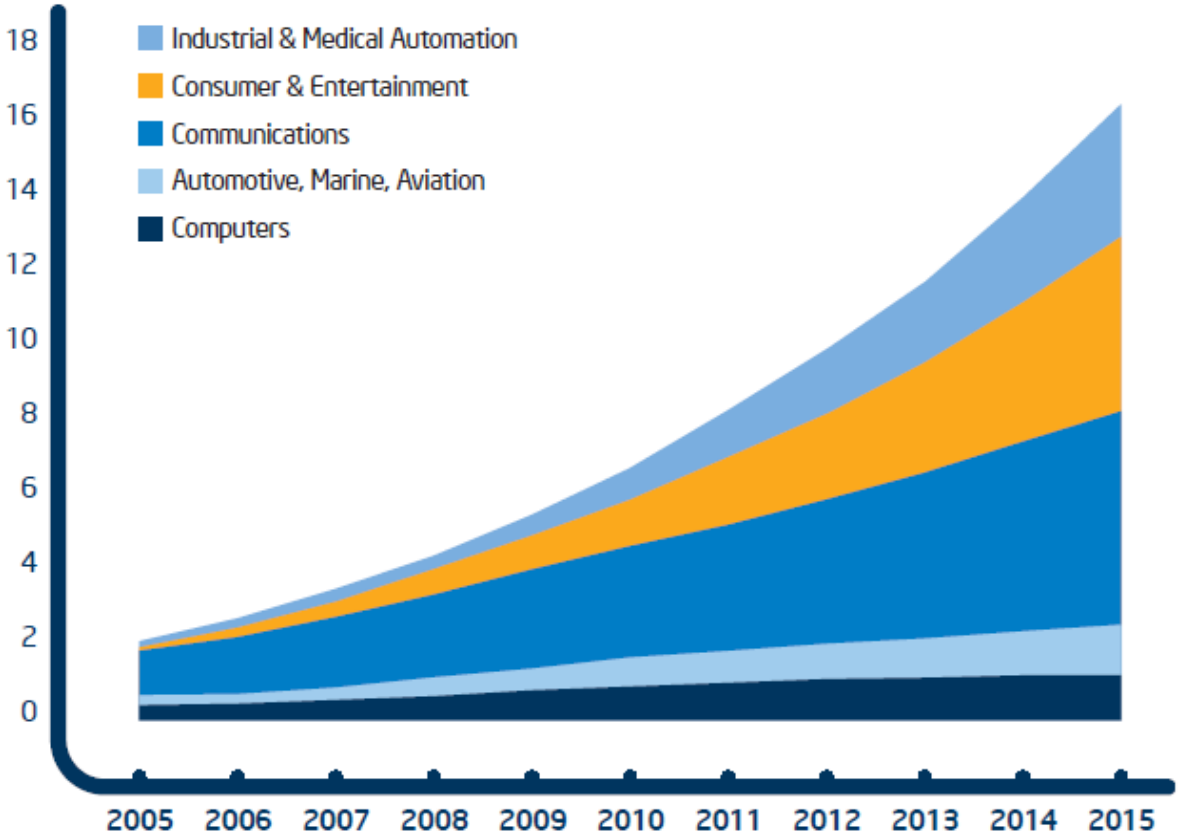







Institut für Technik der Informationsverarbeitung (ITIV)
Prof. Dr.-Ing. Eric Sax, © 2018

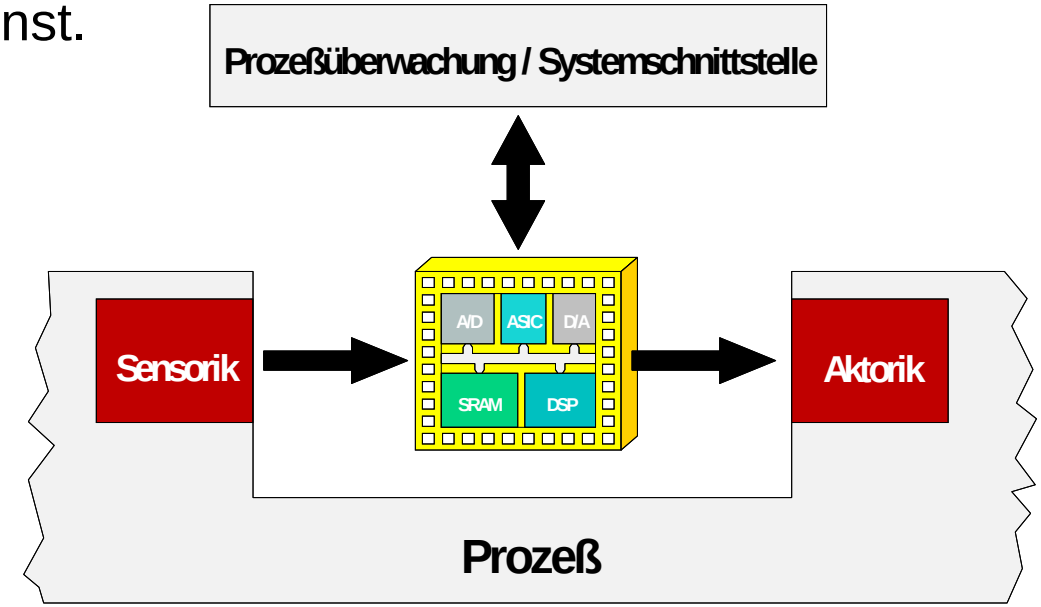
Prozessorstückzahlen



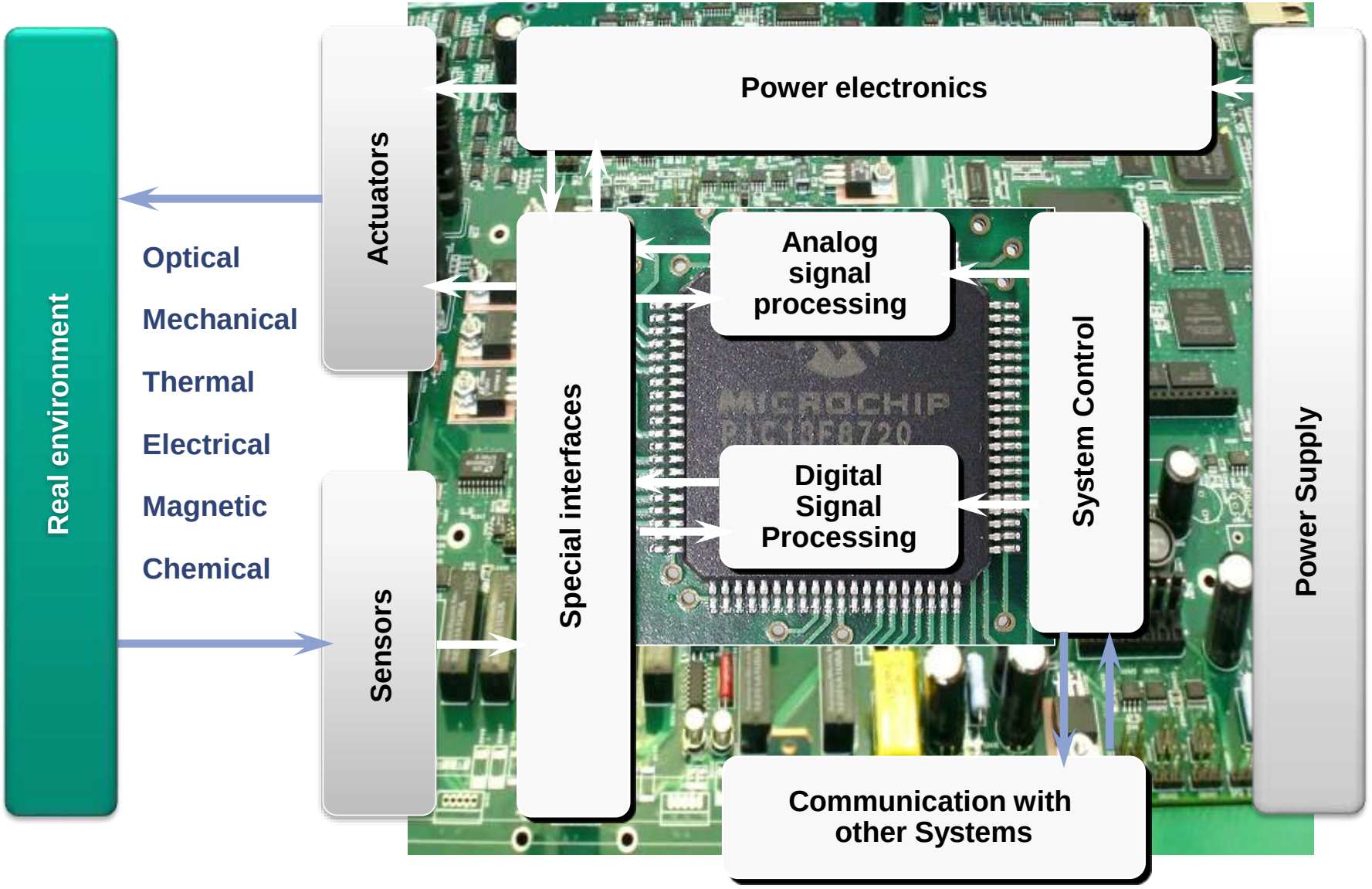
- Entwicklung der weltweiten Anzahl an Prozessoren in den verschiedenen Domänen (in Milliarden Stück) [Quelle: Intel Developer Conference Jan. 2009]

Eingebettetes System (Embedded System)

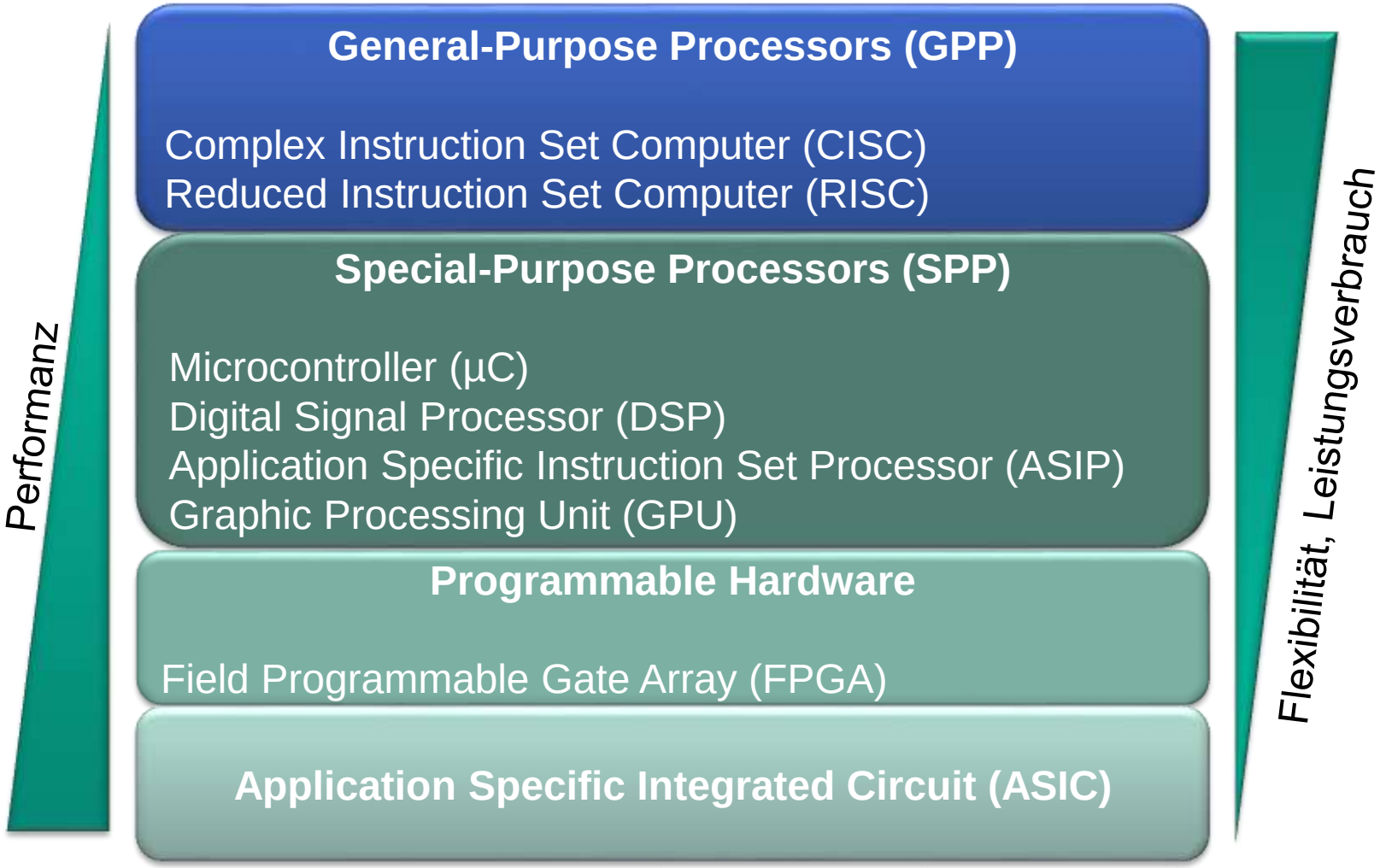
- Der Ausdruck „eingebettetes System“ bezeichnet einen elektronischen Rechner oder auch Computer, der in einen technischen Kontext eingebunden (eingebettet) ist.
- Dabei übernimmt der Rechner entweder Überwachungs-, Steuerungs- oder Regelfunktionen oder ist für eine Form der Daten- bzw. Signalverarbeitung zuständig.
- Eingebettete Systeme verrichten – weitestgehend unsichtbar für den Benutzer – den Dienst.



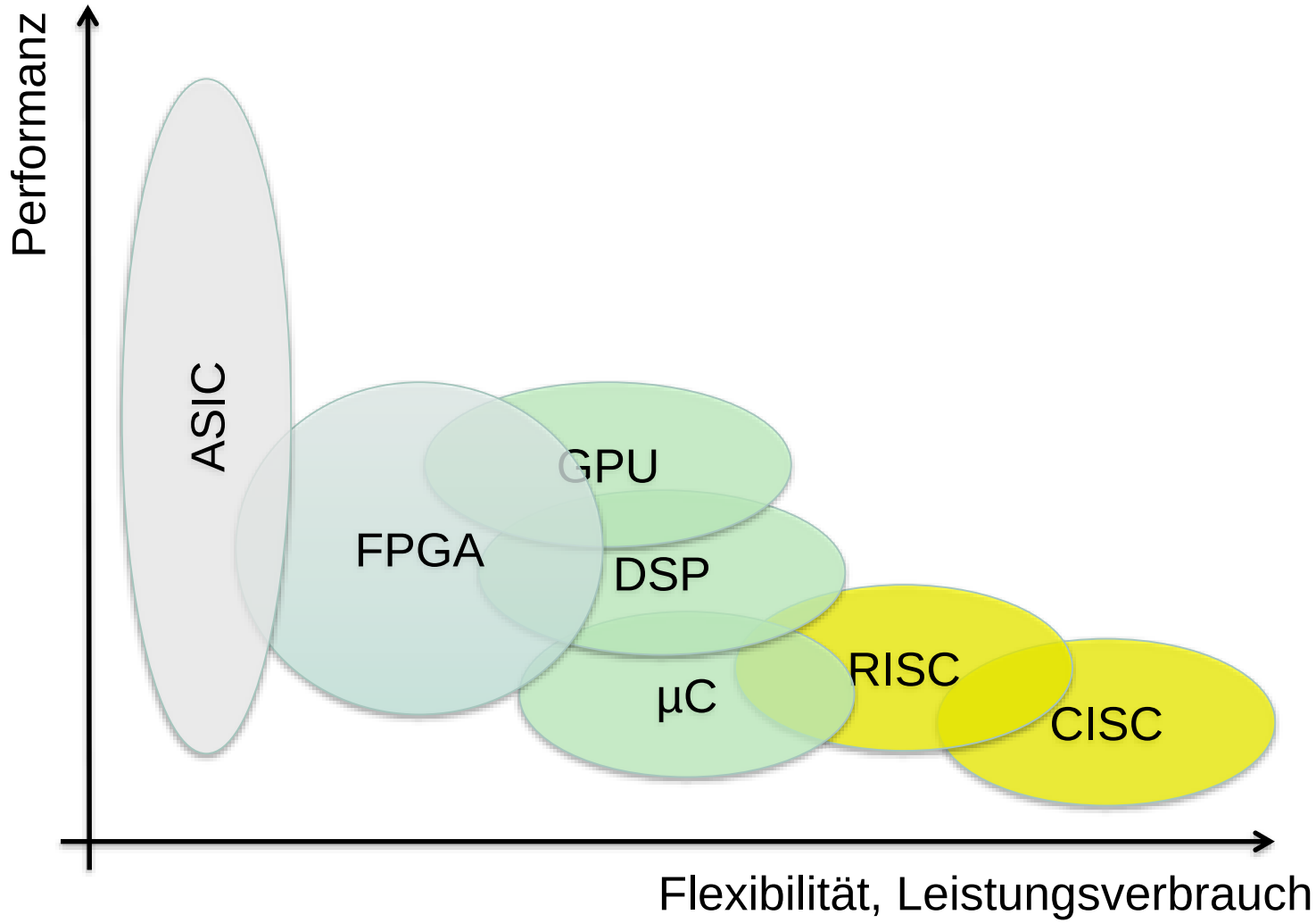
Eingebettete Systeme – prinzipielle Struktur



Vergleich der Zielarchitekturen

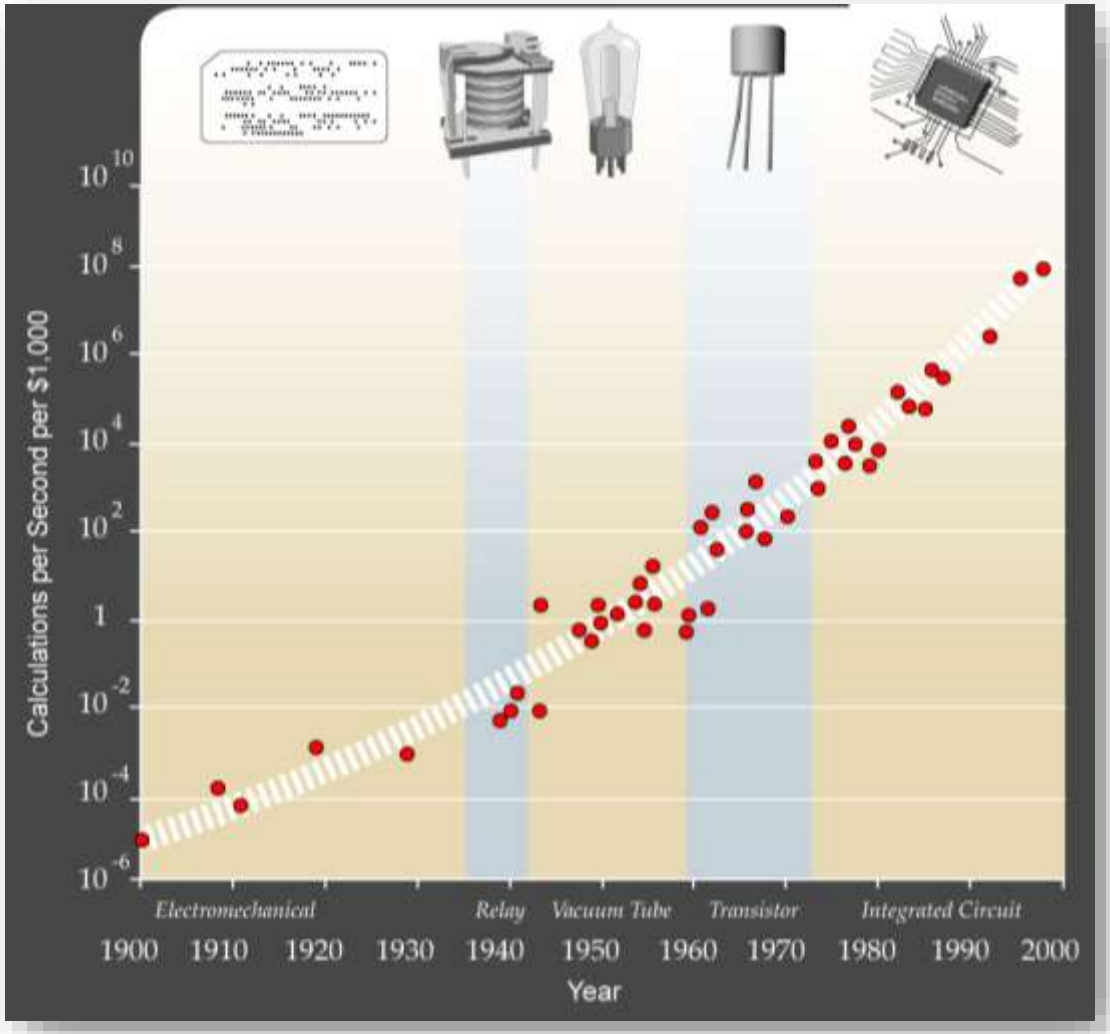


Vergleich der Zielarchitekturen



Moore‘s Law

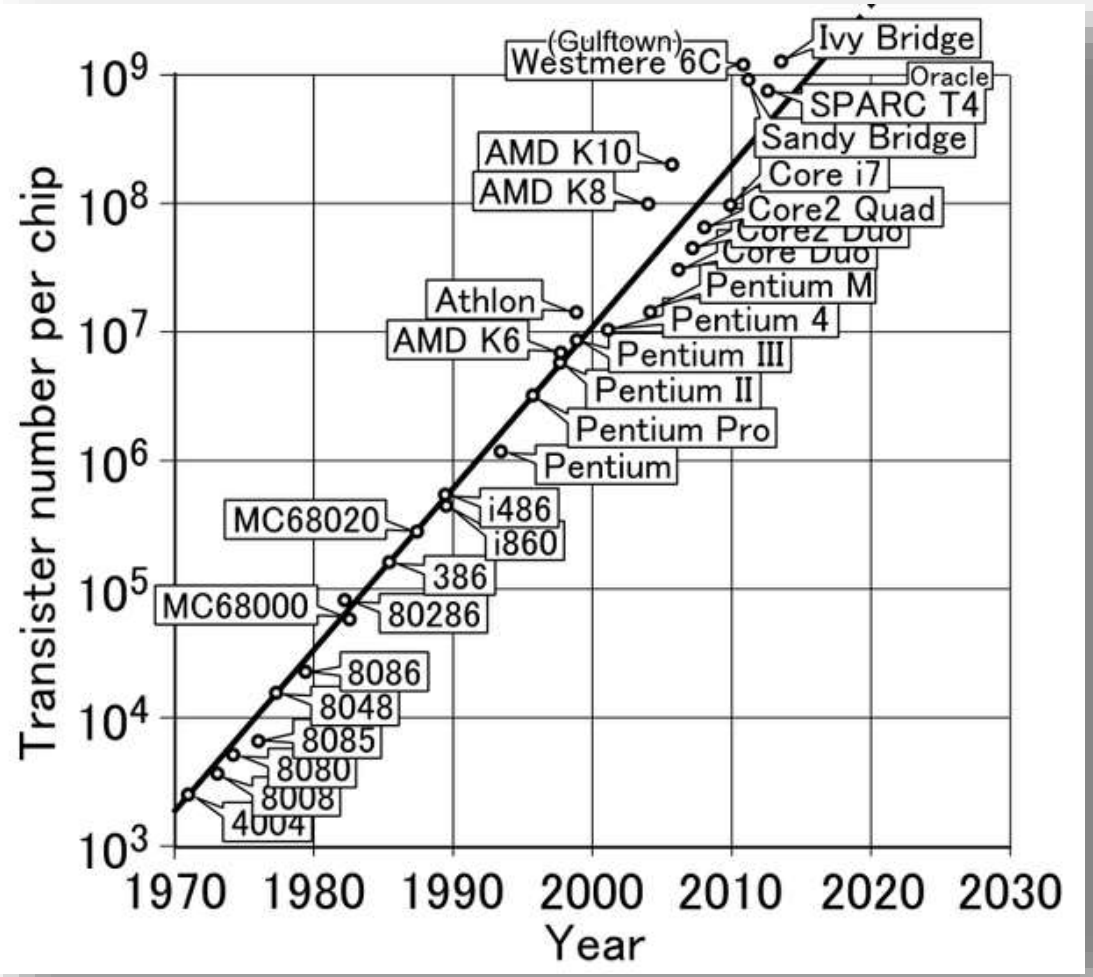
- Das **mooresche Gesetz** besagt, dass sich die Komplexität integrierter Schaltkreise regelmäßig verdoppelt
  - je nach Quelle werden 12 bis 24 Monate als Zeitraum genannt.



Taken from Betanews.com

Moore‘s Law

- Das **moorescher Gesetz** besagt, dass sich die Komplexität integrierter Schaltkreise regelmäßig verdoppelt
  - je nach Quelle werden 12 bis 24 Monate als Zeitraum genannt.



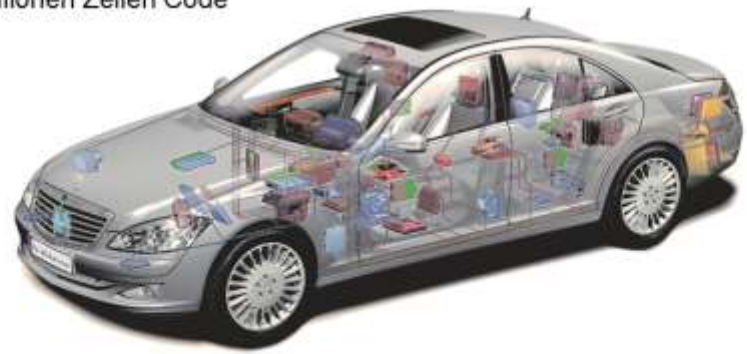
Taken from Betanews.com

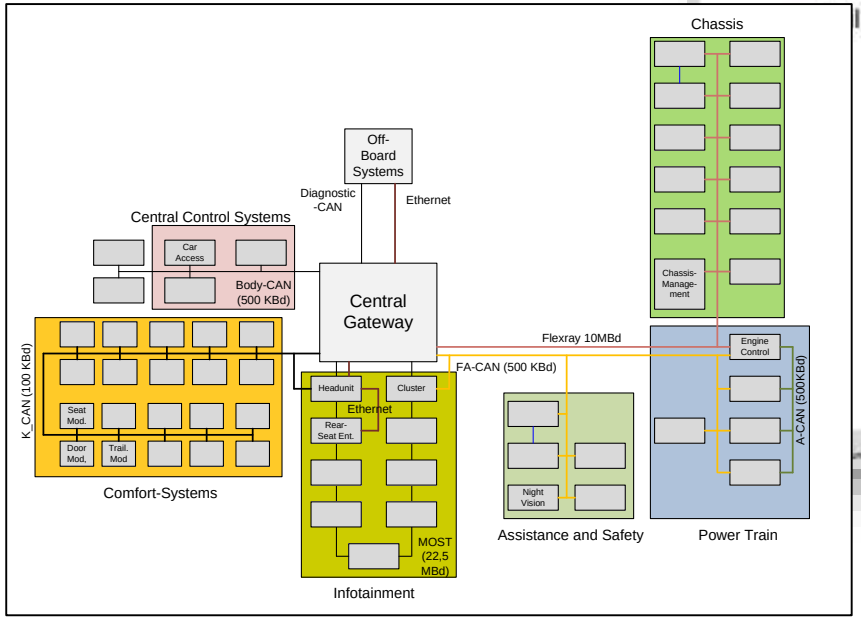
“Komplexe” eingebettete Systeme

- Im Fall von komplexen Gesamtsystemen handelt es sich dabei meist um eine Vernetzung einer Vielzahl von ansonsten autonomen, eingebetteten Systemen (z. B. im Fahrzeug oder Flugzeug).

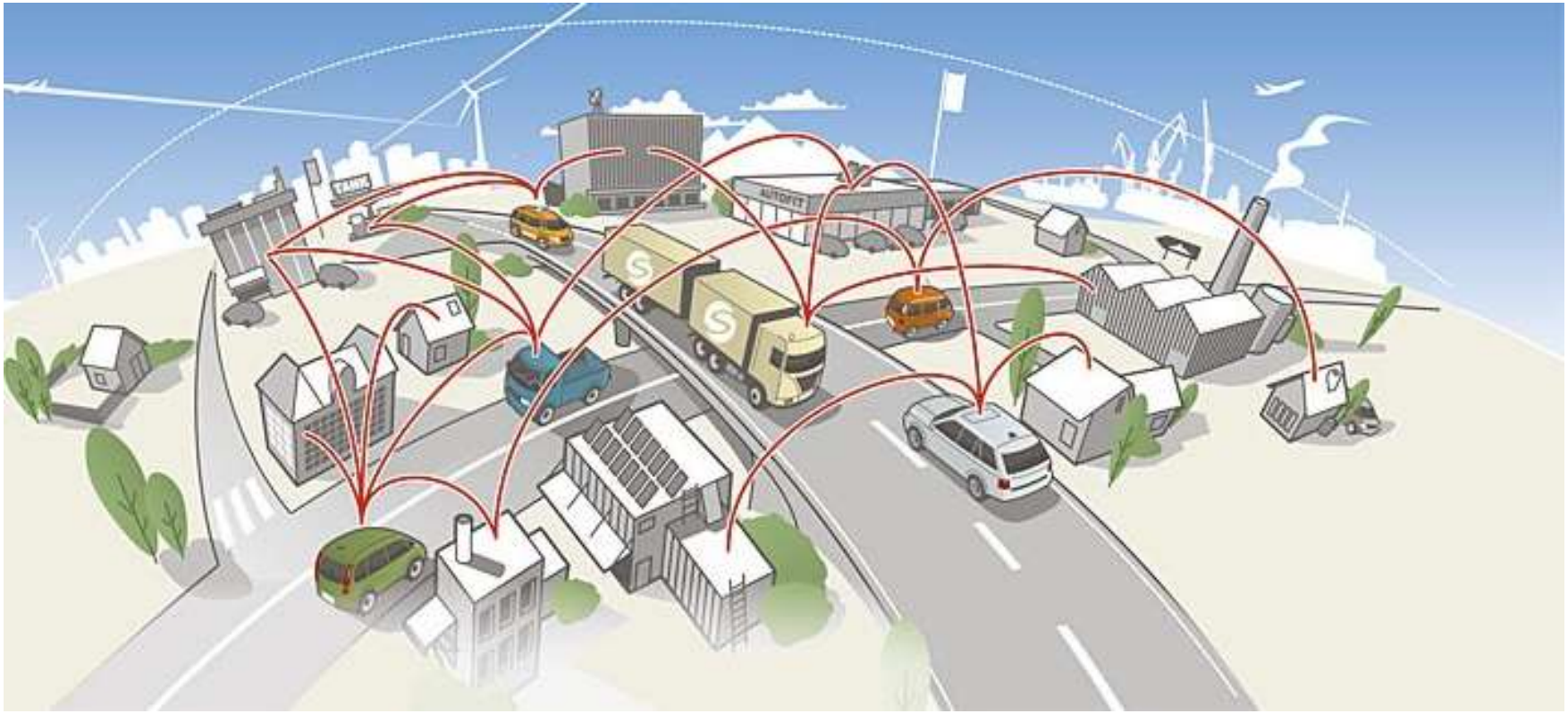
Automotive Electronic Control Units (ECUs)

- Charakteristika: Verteiltheit
- Für ein Premium Fahrzeug gilt:
  - 80 Steuergeräte (Electronic Control Units ECU)
  - > 100 Elektrische Motoren
  - > 2 km Kabelsatz
  - Millionen Zeilen Code





Scenario Smart Mobility



Project-References for Auto-Driving

FZI Referenz

Assistiertes Fahren im städtischen Umfeld

- Integrated Cruise Assist (2(
  - Erkennung von Hinderniss
  - Analyse der relevantesten
  - Generierung von Manöver

FZI Referenz

Hoch- und Vollautomatisiertes Fahren im Stadtverkehr und auf Landstraßen

- Bertha-Benz Fahrt 2013 -

FZI-Referenz

Vollautomatisiertes und autonomes Fahren im niedrigen Geschwindigkeitsbereich

- BMBF-Projekt AutoPLES – Automatisiertes Parken und Laden
- Ergebnis-Demonstration in Ho

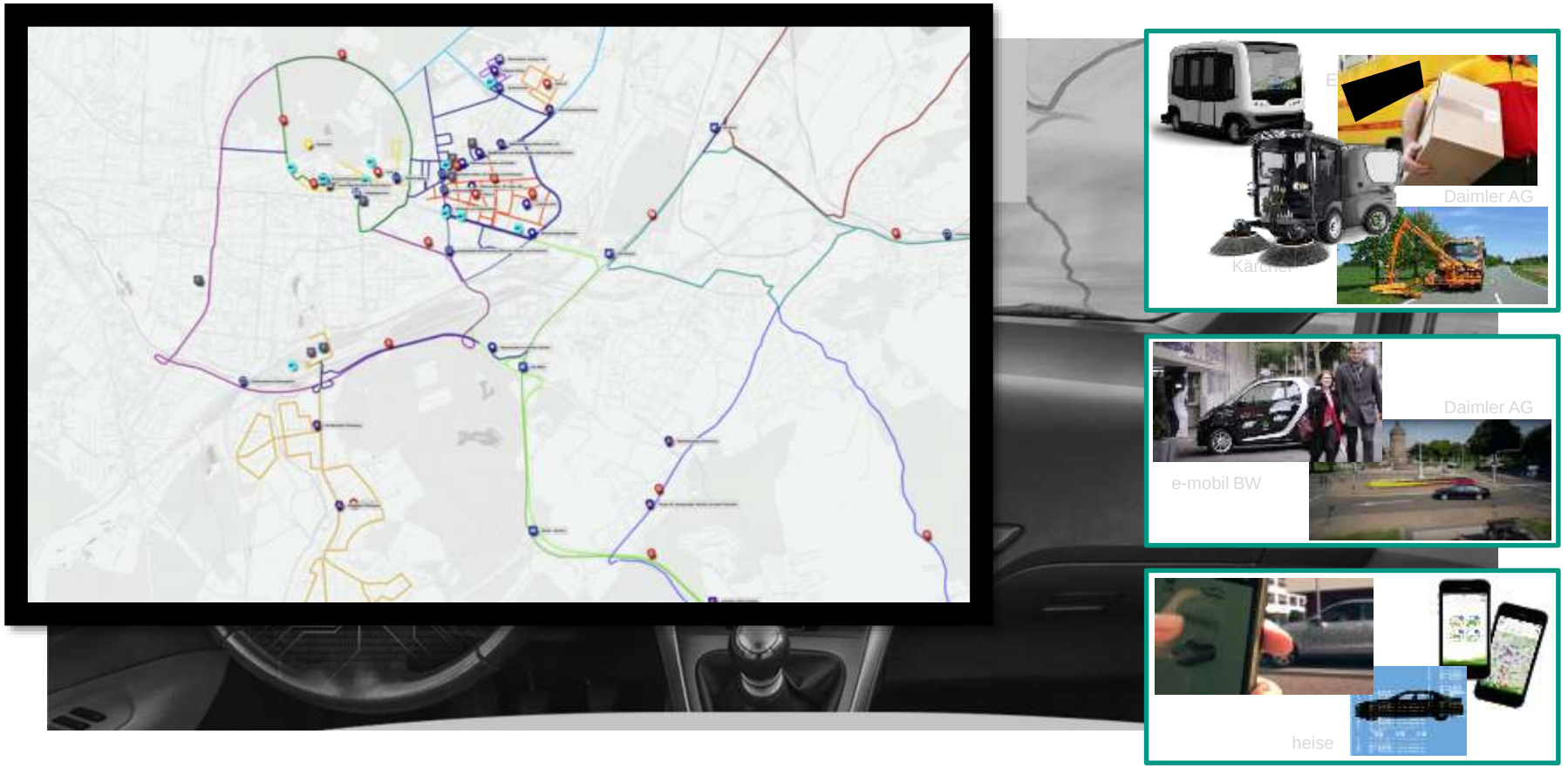
FZI Referenz

Automatisiertes Fahren im ÖPNV

- Vorstellung und life Demo Juli 2016, Amsterdam
- Hochgenaue Lokalisierung, Genaues Anfahren, Sicherheit ....

Testfeld Autonomes Fahren Baden Württemberg

Testfeld Autonomes Fahren Baden Württemberg




Fragen?

IT Inhalt

0. Einleitung und Motivation

- Organisatorisches
- Begriffe
- Typische Anwendungen

1. Rechnerarchitekturen

- Einführung
- Allgemeiner Aufbau der internen Hardware Architektur
- Instruction Set Architecture (ISA) am Beispiel ARM Cortex M4
- Klassifikation von Rechnerarchitekturen
- Performanzsteigerung
- ausgewählte Beispiele

John von Neumann

- * 28. Dezember 1903 in Budapest (Österreich-Ungarn) als *János Lajos Neumann*
- † 8. Februar 1957 in Washington, D.C.
- Mathematiker österreichisch-ungarischer Herkunft.
- leistete bedeutende Beiträge zur mathematischen Logik, Funktionalanalysis, Quanten-mechanik und Spieltheorie und gilt als einer der Väter der Informatik.
- Später nannte er sich **Johann von Neumann (von Margitta)**
- heutzutage ist vor allem unter seinem in den USA gewählten Namen John von Neumann bekannt.

Funktionentheorie II, Dr. **Neumann von Margitta**, Mi So 9-11, p. [634
Analytische Zahlentheorie II, Prof. **Schur**, Mo Di Do Fr 11-12, p. [635
Axiomatik der Mengenlehre und mathematische Logik, Dr. **Neumann von Margitta**, Mo 16-17, Do 18-20, p. [636

Mathematisches Kolloquium, Prof. **Bieberbach**, Dr. **Feigl**, Prof. **Hammerstein**, Dr. **Hopf**, Dr. **Neumann von Margitta**, Prof. Erhard **Schmidt** und Prof. **Schur**, Di 19-21, publ. [645

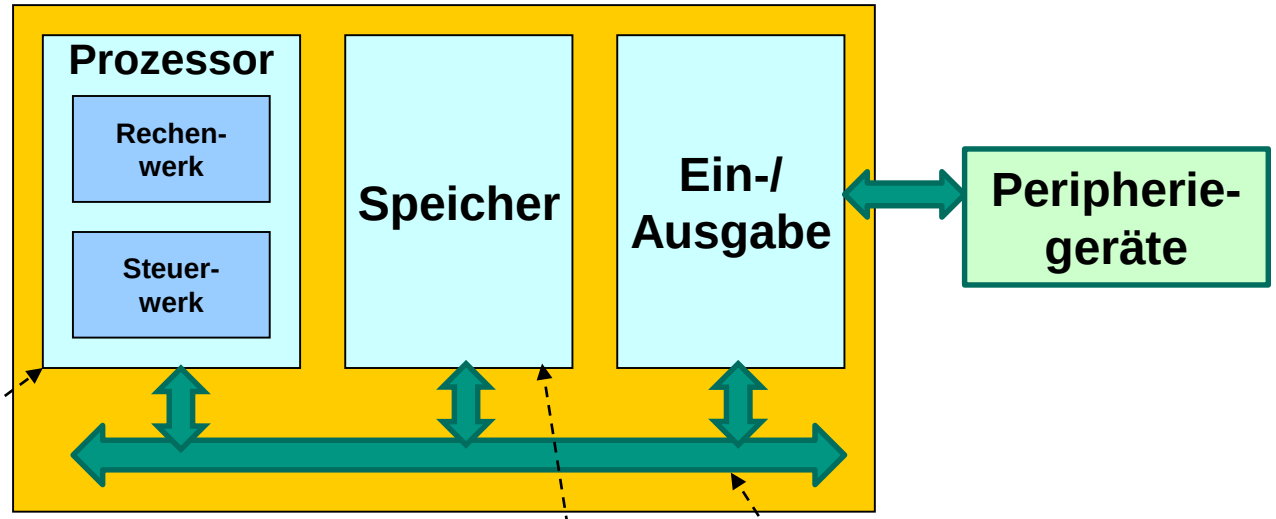
Besprechung neuerer Arbeiten zur Quantentheorie, Dr. **Szilard** mit Dr. **Neumann von Margitta**, Fr 15½-17, pg. [689

Spezielle Funktionen der mathematischen Physik, Dr. **Neumann von Margitta**, Mi So 9-11, p. [659
Galoissche Theorie, Prof. **Schur**, Mo Di Do Fr 11-12, p. [660
Partielle Differentialgleichungen, Prof. **Hammerstein**, Mo Di Do Fr 12-13, p. [661
Kombinatorische Topologie, Dr. **Hopf**, Di Do Fr 10-11, p. [662
Hilbertsche Beweistheorie, Dr. **Neumann von Margitta**, Do 16-18, p. [663

- Von Neumann in den Vorlesungsverzeichnissen der Friedrich-Wilhelms-Universität zu Berlin.
- Die ersten drei Ausschnitte stammen aus dem Sommersemester 1928, der vierte Ausschnitt aus dem Wintersemester 1928/29.

Rechnerarchitekturen (Wiederholung aus DT)

- Interne Architektur (nach John von Neumann Architektur)



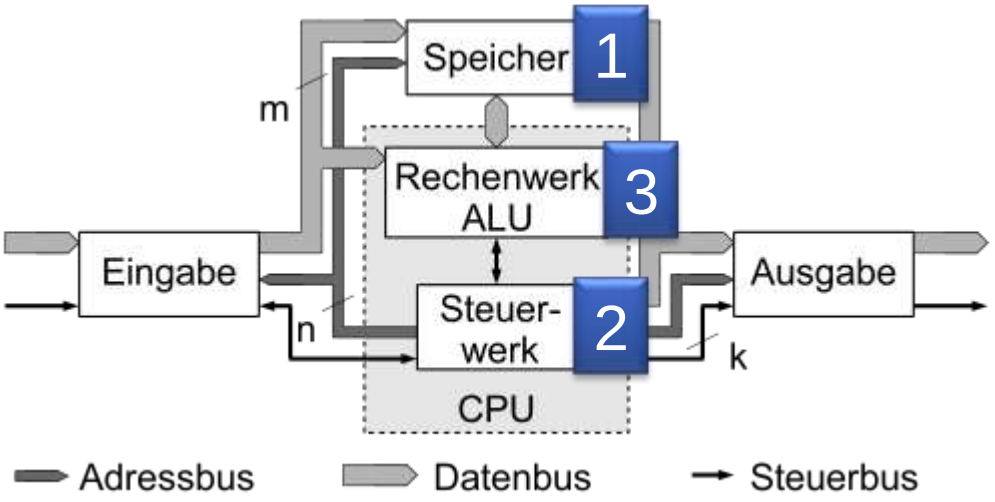
„Rechnerkern“
CPU (central processing unit)
führt Maschinenprogramm aus
Programm ist Folge von Maschinenbefehlen

Speicher
Arbeitsspeicher speichert Maschinenbefehle
und zugehörige Daten

Systembus
(bidirektional)
verbindet vorhandene
Komponenten,
überträgt Daten, Befehle,
Steuersignale,
Busbelegung nach Vorrang
der Geräte

Von-Neumann-Architektur

- Arbeitsweise:
  1. Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
  2. Befehl dekodieren, gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
  3. Befehl ausführen, gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)
  - Weiter bei Schritt 1

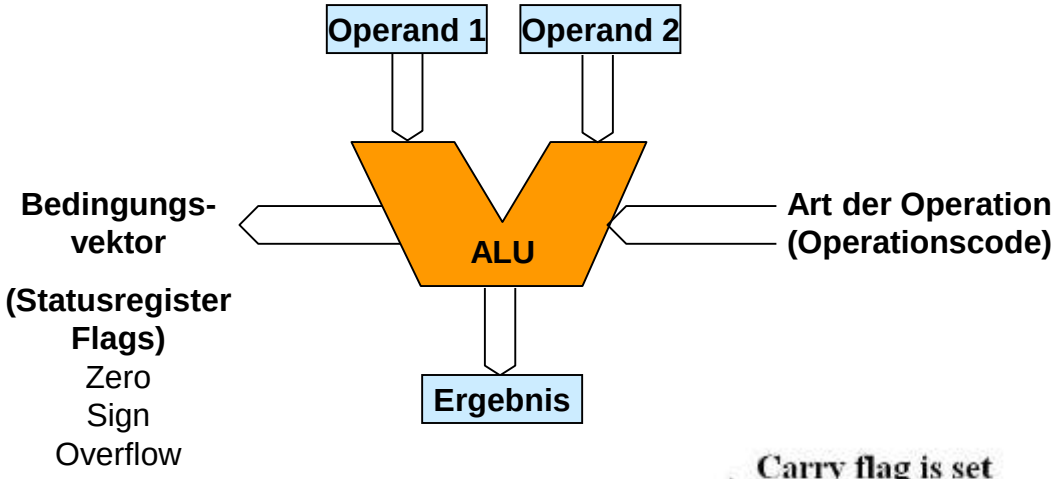


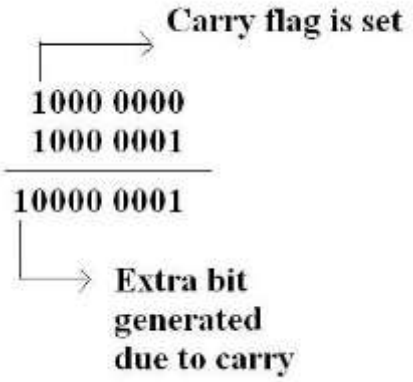
- Daten und Befehle sind binär codiert
- Auf Maschinenbefehlsebene nur binäre Werte 0 und 1

Rechenwerk 3

Arithmetisch/Logische Einheit ALU

- Verarbeitung der Daten
- arithmetische und logische Basisfunktionen für ein einfaches Rechenwerk:
  - Arithmetische Befehle:
    - Addition
    - Komplementbildung
  - Logische Befehle:
    - Negation
    - Konjunktion
    - Disjunktion
  - Liefert „Flags“ zurück an Steuerwerk
    - zum Beispiel bei arithmetischen Berechnungen
      - den Übertrag (*Carry-Flag*)
      - den Überlauf (*Overflow-Flag*)





Steuerwerk (Leitwerk) 2

- Steuerung und Koordination des zeitlichen Ablaufs der Komponenten des Prozessors
  - Versorgung mit Daten und Adressen
  - Selektion von Geräten
  - Auswahl von Operanden und Operationen
- Ursprünglich Steuerung durch den Bediener, später Automatisierung durch zeitliche Abfolge von Binärinformationen im Maschinenprogramm (Steuersignale)
- Formalisierung der Anweisung nötig:
  - Befehl:
    - (maschinenlesbare) Anweisung, aus der [illegible] geht, welche Operation mit welchen Operanden durchgeführt w[illegible]oll
  - Befehlsformat:
    - Festlegung, wie die für ei[illegible]ehl notwendigen Angaben dargestellt werden

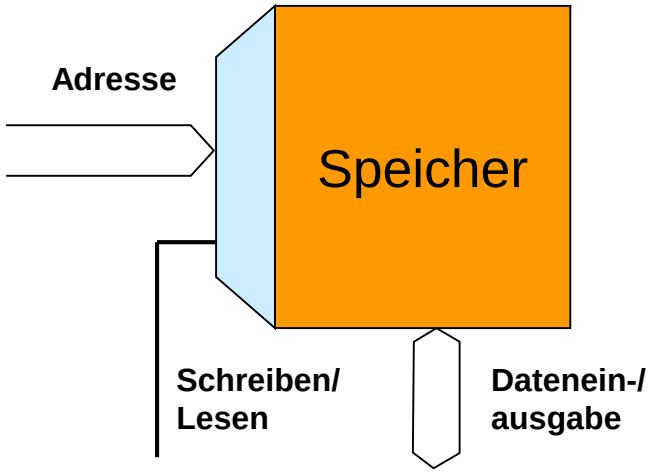
s. weitere Folien später

Speicherwerk 1

- Speicherwerk ist frei adressierbarer Schreib/Lese-Speicher
  - nicht für jede Operation muss das Ein-/Ausgabewerk benutzt werden
- Komponenten
  - Hauptspeicher (oder Arbeitsspeicher)
    - Enthält die aktuell auszuführenden Daten oder Programmteile (flüchtig)
    - Enthält (je nach Architektur) die Firmware (BIOS) (persistent)
  - Cache
    - Schneller Pufferspeicher für bereits einmal geladenene Daten (flüchtig)
  - Register im Rechnerkern
    - Speicherbereich, der direkt mit der eigentlichen Recheneinheit verbunden ist und die unmittelbaren Operanden und Ergebnisse aller Berechnungen aufnimmt (flüchtig)
    - Register sind in der Regel höchstens so groß wie die Wortgröße des Prozessorkerns (8, 16, 32, 64 Bit).
    - Die Gesamtheit aller Register bezeichnet man als dessen Registersatz.
    - Verschiedene Register dienen zum Zwischenspeichern von Befehlen, Speicheradressen, Rechenoperanden und auch zum Zwischenspeichern von Ein- und Ausgabewerten.

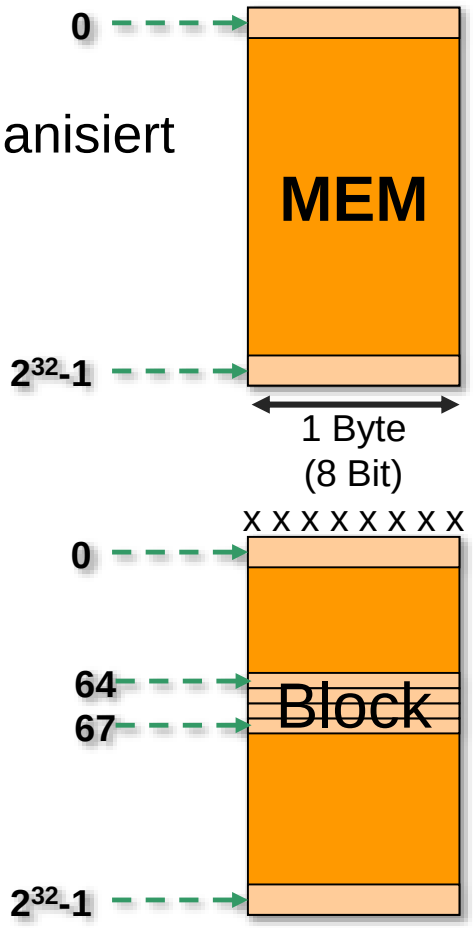
Speicherwerk 1

- Festplattenspeicher (persistent) gehören nicht zum Speicherwerk, sondern zum E/A-Werk
  - Daten werden von der Festplatte zunächst in den Arbeitsspeicher geladen, bevor sie verarbeitet werden können.
  - *Spezialfall: Virtueller Arbeitsspeicher auf externer Festplatte (extrem langsam aber kostengünstig).*

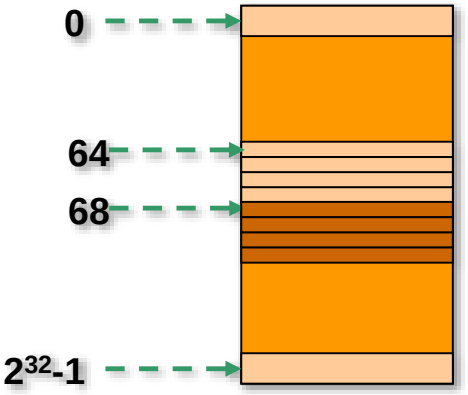


Speicherorganisation

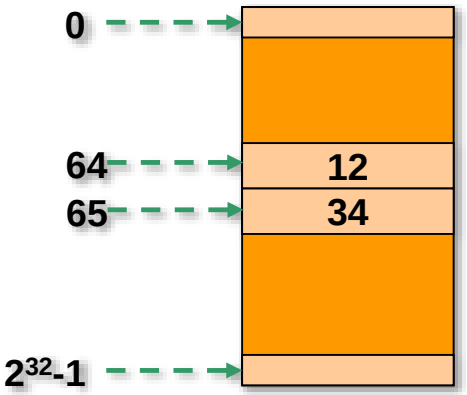
- Breite des Adressvektors ergibt die maximale Zahl der verfügbaren Speicherworte: $2^{<\text{Breite}>}$
  - z.B. 32 Bit Adresse $\Rightarrow 2^{32}$ Bytes adressierbar
- Ein Speicherwort wird typischerweise in Bytes organisiert
  - Wort (1 Byte)
  - Doppelwort (2 Bytes)
  - Quadwörter (4 Bytes)
  - Halbwort (4 bit)
- Ausrichtung (alignment)
  - Daten mit einer festen Länge können nur beginnend mit einer festen Adresse im Speicher (1 Byte Breite) untergebracht werden, z.B. Quadwörter nur an Adressen, die ein Vielfaches von 4 sind.
  - Speicherbereich für zusammenhängende Daten wird Block genannt.



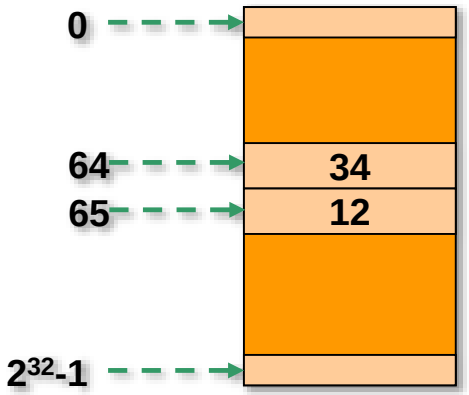
Speicherorganisation



- Besteht ein Datum (oder auch Befehl) aus 4 Bytes, so muss die Byte-Adresse um 4 erhöht werden, um das nächste Datum (oder nächsten Befehl) zu holen
- Endianess
  - Beschreibt, wie Daten >1 Datenbyte im Speicher abgelegt werden.
  - Big Endian (Mac)/Little Endian (Intel x86)
- Beispiel Speicherung von 1234_h in einem Wort:



„big endian“



„little endian“

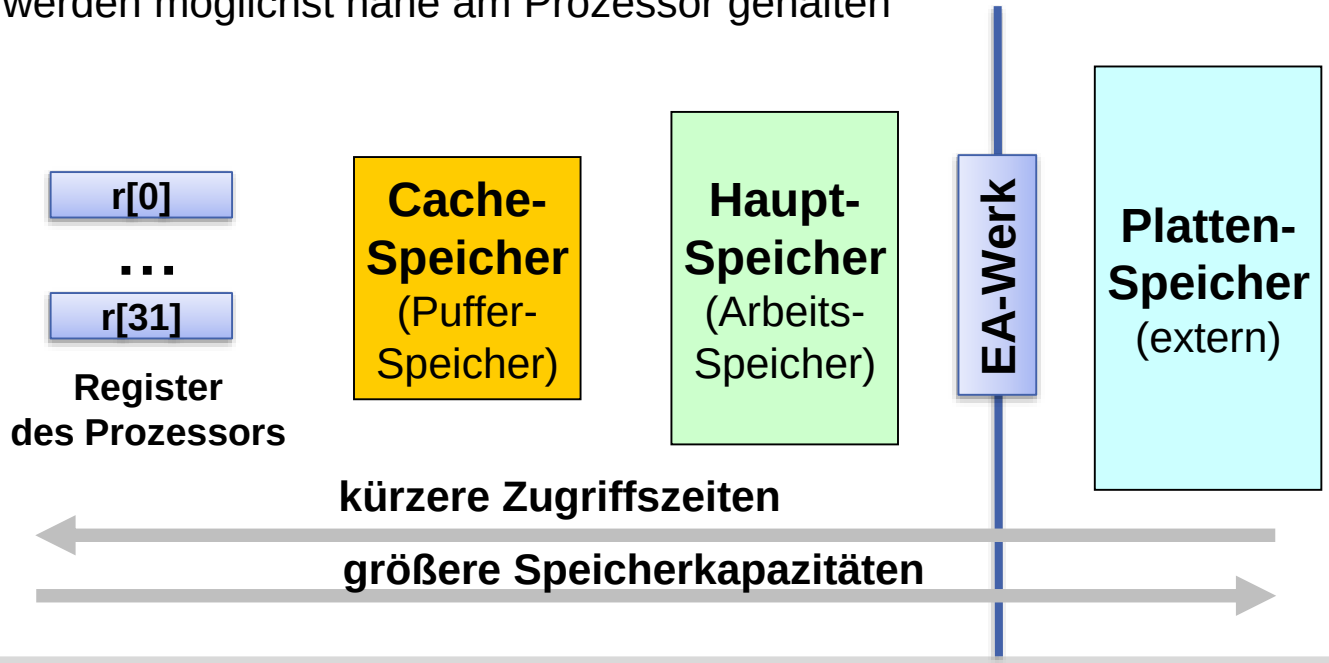
Endianess
Weiteres Beispiel

- Beispiel:
  - $439.041.101_D = 1A_h\,2B_h\,3C_h\,4D_h$
  - Speicherung ab Adresse 10000 (1 Byte Datenbreite / 8 Stellen)
- Big Endian speichert in Reihenfolge
  - $1A_h2B_h3C_h4D_h$ (Wert bei Adresse 10000: 0001 1010)
- Little Endian speichert Bytes in umgekehrter Reihenfolge
  - $4D_h3C_h2B_h1A_h$ (Wert bei Adresse 10000: 0100 1101)

	Big Endian		Little Endian	
Adresse	Hex	Dez	Hex	Dez
10000	1A	26	4D	77
10001	2B	43	3C	60
10002	3C	60	2B	43
10003	4D	77	1A	26

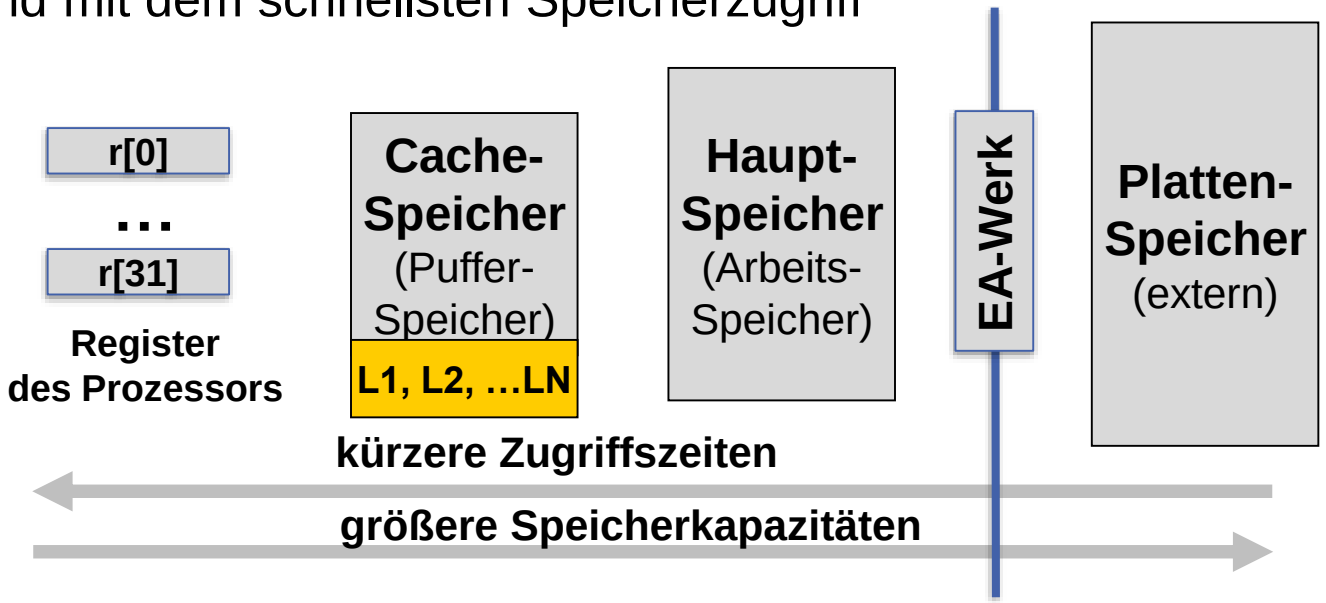
Speicherorganisation - Speicherhierarchie

- Speicherhierarchie
  - Prinzip der Lokalität:
    - Zeitlich: benutzte Daten/Befehle werden bald wieder benutzt
    - Räumlich: auf Daten/Befehle, die im Adressraum nahe zu gerade benutzten liegen, wird wahrscheinlicher verwiesen als auf weiter weg liegende
  - Konsequenz:
    - Räumlich nahe beieinander liegende und oft referenzierte Teile (Blöcke) werden möglichst nahe am Prozessor gehalten



Caches - Hierarchie

- Caches reduzieren unperformante Hauptspeicherzugriffe, durch Einsatz schnellerer Zwischenspeicher
  - Hauptspeicher ist ca. 15 x langsamer
- Das Cache ist software-transparent, d.h. der Benutzer braucht nichts von seiner Existenz zu wissen.
- In der Regel besitzt ein Rechner einen getrennten Cache für Instruktionen (Instruktionscache) und für Daten (Datencache)
- Cache Level: Level 1 bis Level N bezeichnet die Cache Speicher beginnend mit dem schnellsten Speicherzugriff



Cache-Einträge

- Begriffsdefinitionen:
  - Cache Hit := Benötigtes Datenwort ist im Cache und kann direkt von dort geholt werden
  - Cache Miss := Benötigtes Datenwort ist nicht im Cache und muss aus dem Speicher geholt werden
  - Hit-Rate := Verhältnis Cache Hits zu Zugriffszahl

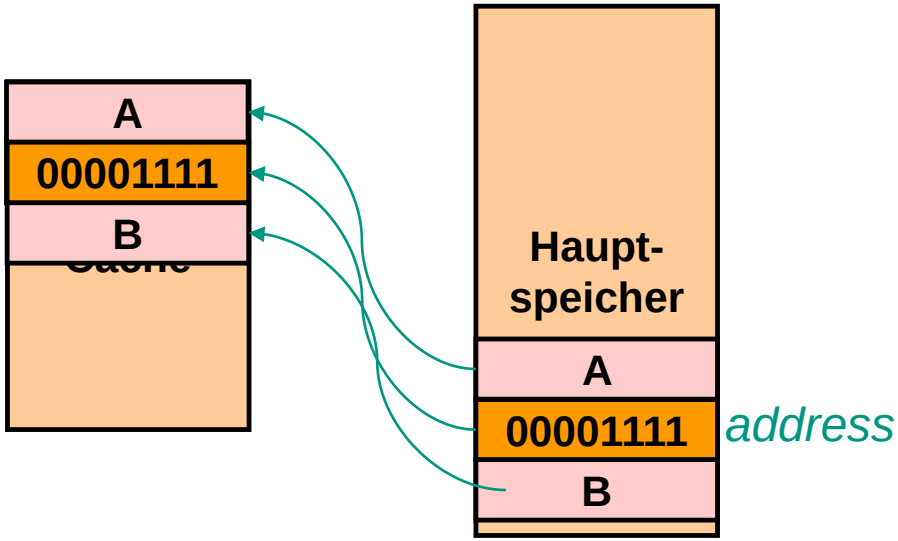
Cache-Einträge

- Eine Zeile des Caches (Cache Line) enthält verschiedene Informationen:
  - Die originale Hauptspeicheradresse
  - Einen Datenblock, der den entsprechenden Speicherinhalt des Hauptspeichers enthält

- Flag bits
  - Das “valid bit” zeigt an, ob der Block valide Daten enthält
    - Nach “power-up” werden alle valid bits auf "invalid“ gesetzt.
  - Daneben gibt es das “dirty bit”, das anzeigt, dass im Cache Block die Daten seit dem Lesen aus dem Hauptspeicher geändert wurde.
    - Das bedeutet, dass der Prozessor Daten in den Cache zurückgeschrieben hat und diese noch nicht an den Hauptspeicher weitergereicht wurden.

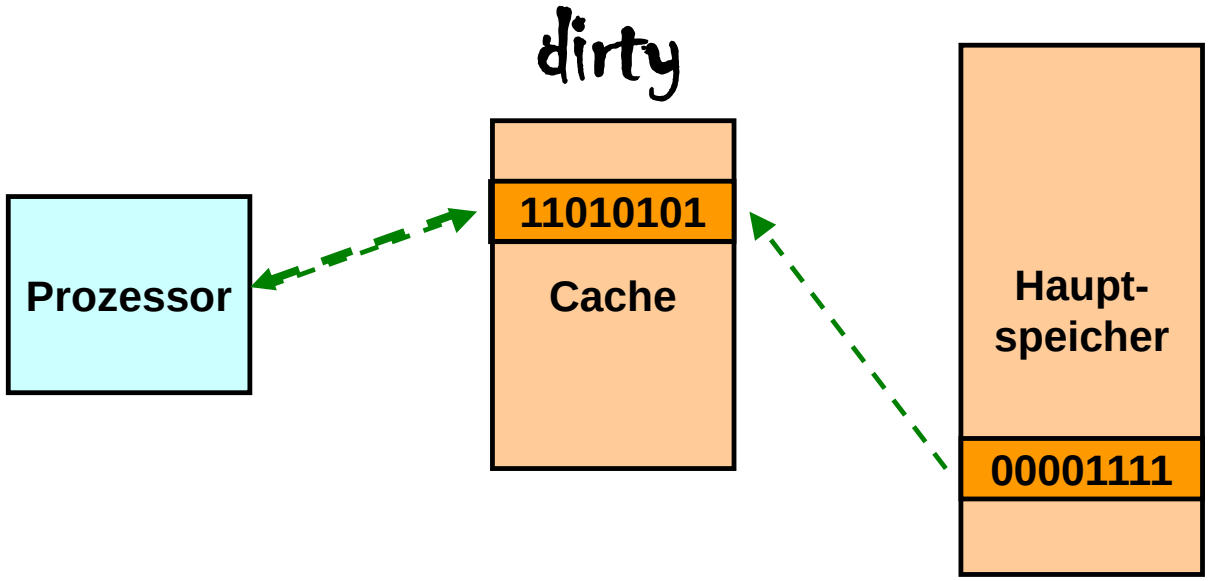
Caches - Lesezugriff

- Befehl zum Lesen eines Datums aus dem Hauptspeicher unter Adresse (*address*)
- CPU überprüft, ob eine Kopie der Hauptspeicherzelle (unter *address*) im Cache abgelegt ist.
  - Falls ja (cache hit)
    - so entnimmt die CPU das Datum aus dem Cache → kein Zugriff auf Hauptspeicher notwendig
    - Die Überprüfung und das eigentliche Lesen aus dem Cache erfolgt in einem Zyklus, ohne einen Wartezyklus einfügen zu müssen.
  - Falls nein (cache miss)
    - so greift die CPU auf den Hauptspeicher zu, lädt zusätzlich die umgebenden Blöcke des Datums in den Cache und lädt das Datum von dort in die CPU



Caches – Lese-/Schreibzugriff

- Was geschieht beim Schreiben?
  - Daten in Cache und Hauptspeicher korrespondieren einander nur solange der Prozessor nicht die Daten im Cache überschreibt

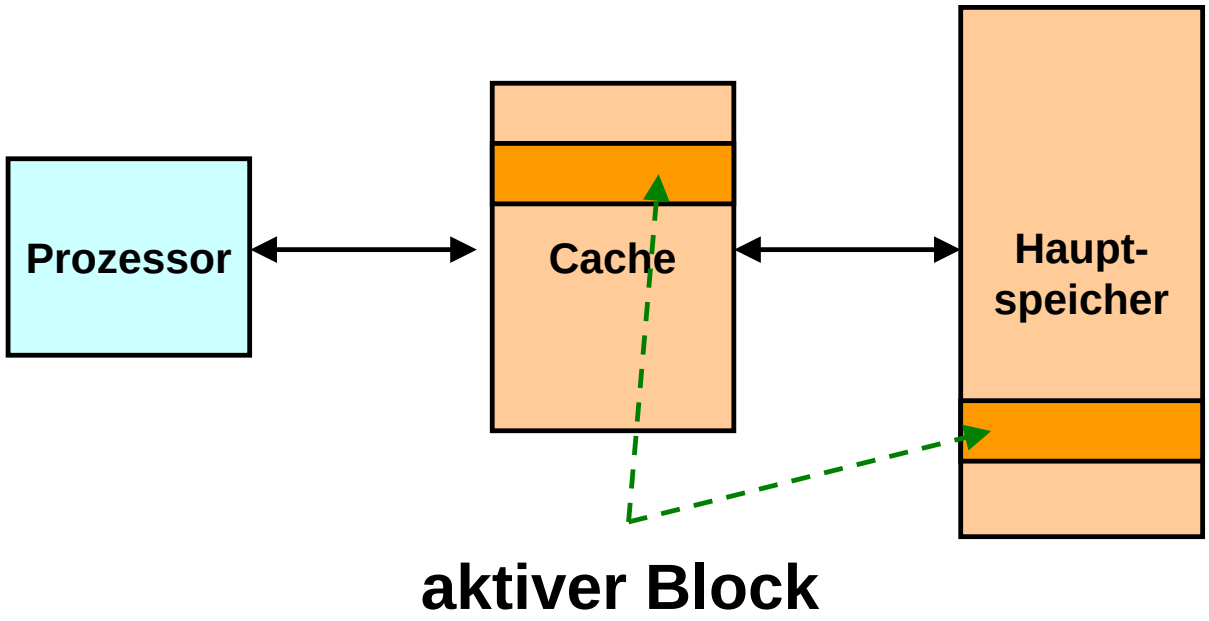


Cache-Schreibstrategien
(des Prozessors)

- Möglichkeiten wenn der Cache die Speicheradresse bereits enthält:
  - Write Back: (Cache hit)
    - Änderungen werden zunächst nur in den Cache geschrieben
      → Setzen eines Dirty-Bits
    - Erst bei einer gezielten Auslagerung der Cachezeile in Hauptspeicher wird diese auch zurück geschrieben. (Konsistenzprobleme bei Multiprozessoren)
  - Write Through: (Cache hit)
    - Jede Änderung wird zugleich in Cache und Hauptspeicher aktualisiert
      → Belastet sehr den Speicherbus
- Möglichkeiten wenn der Cache die Speicheradresse nicht enthält:
  - Write Around: (Cache miss)
    - Änderung wird ohne Cache-Update direkt in den Hauptspeicher geschrieben
  - Write-Allocate: (Cache miss)
    - Block wird aus Hauptspeicher in Cache-Zeile kopiert und dann eine Schreib-Treffer-Operation (Write Back oder Write Through) ausgeführt

Cache-Organisation

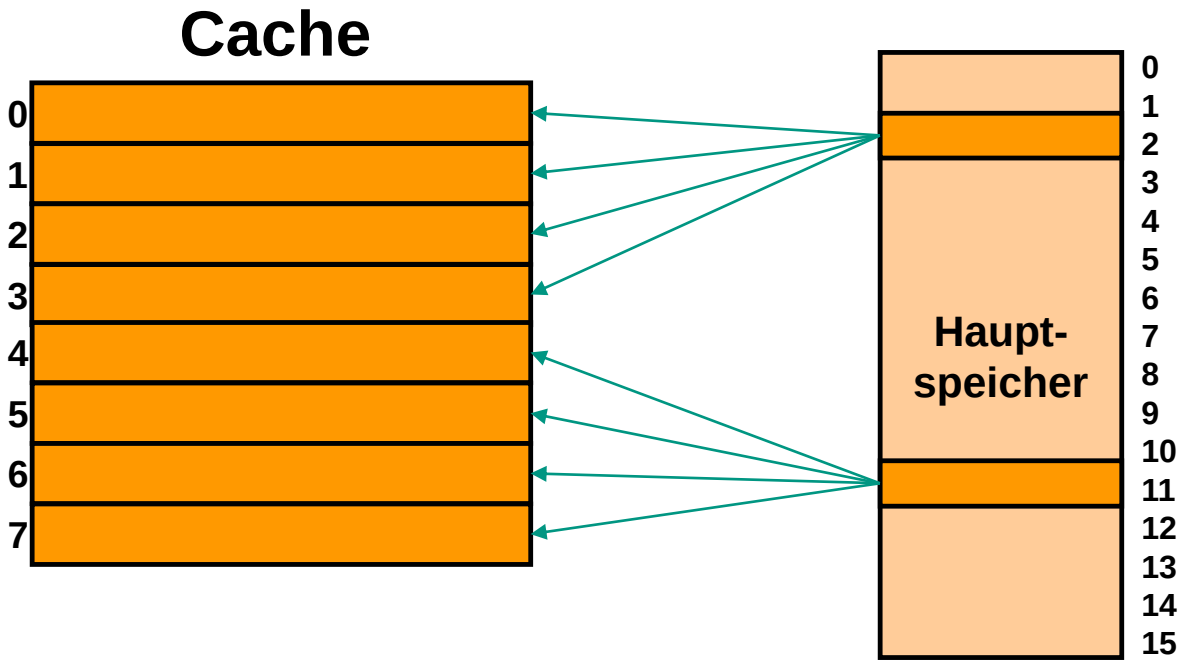
- Antworten auf drei Fragen entscheidend:
  - A: Wo kann ein Block im Cache platziert werden?
  - B: Wie kann festgestellt werden, ob ein Block im Cache ist?
  - C: Welcher Block soll ersetzt werden, falls ein Block gebraucht wird?



Cacheorganisation

A: Wo kann ein Block im Cache platziert werden?

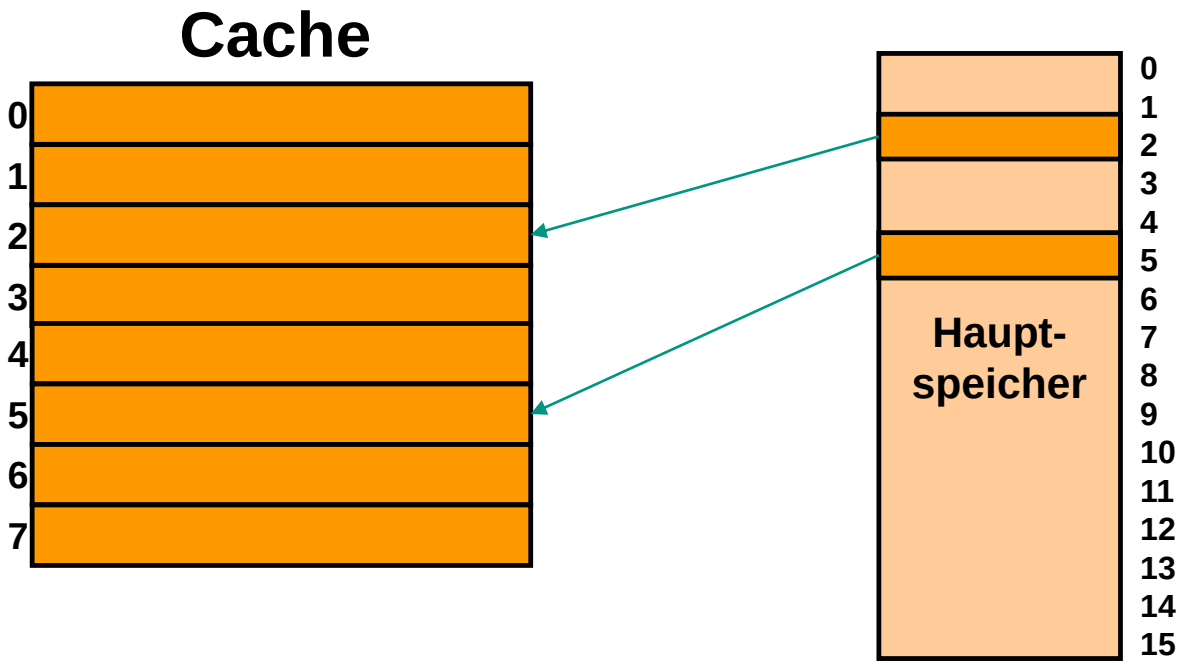
- 1. Verfahren: voll-assoziativ
  - Jeder Block kann an allen Adressen gespeichert werden („wo frei ist“)
  - Nachteil → Beim Suchen muss der Tag jeder einzelnen Cachezeile verglichen werden (hoher Aufwand)



Cacheorganisation

A: Wo kann ein Block im Cache platziert werden?

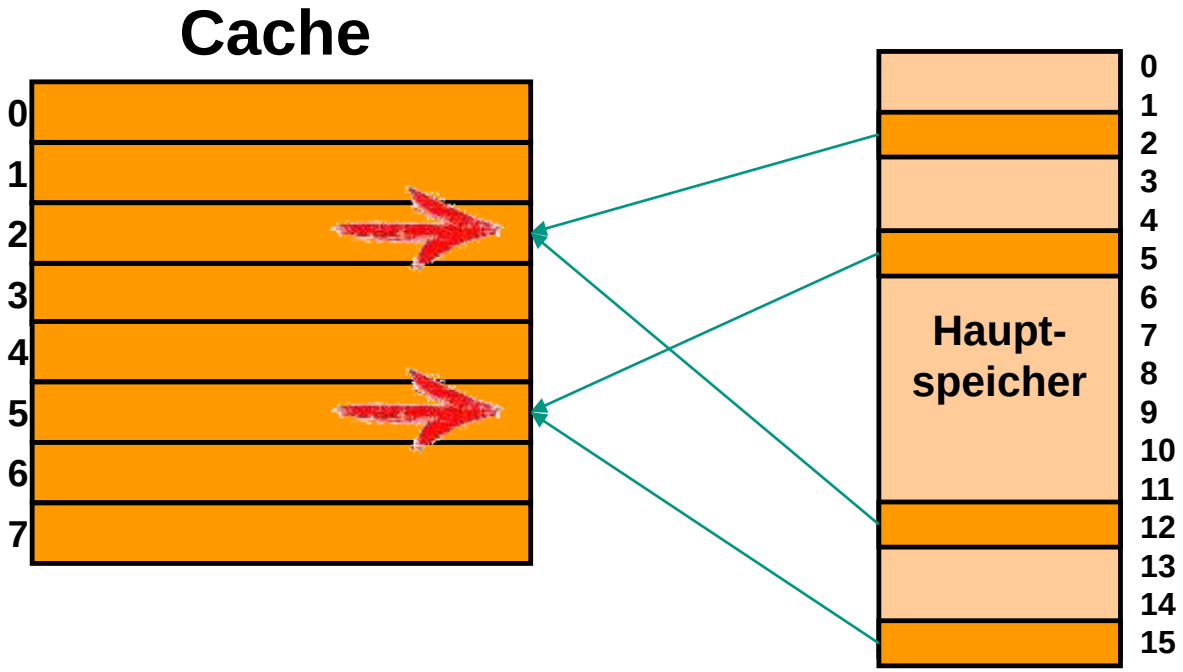
- 2. Verfahren: direkt-abbildend (direct mapped)
  - Jede Datenadresse wird anhand einer Abbildungsvorschrift (z.B. einen Teil der Adresse („Index“) fest auf eine bestimmte Stelle im Cache abgebildet



Cacheorganisation

A: Wo kann ein Block im Cache platziert werden?

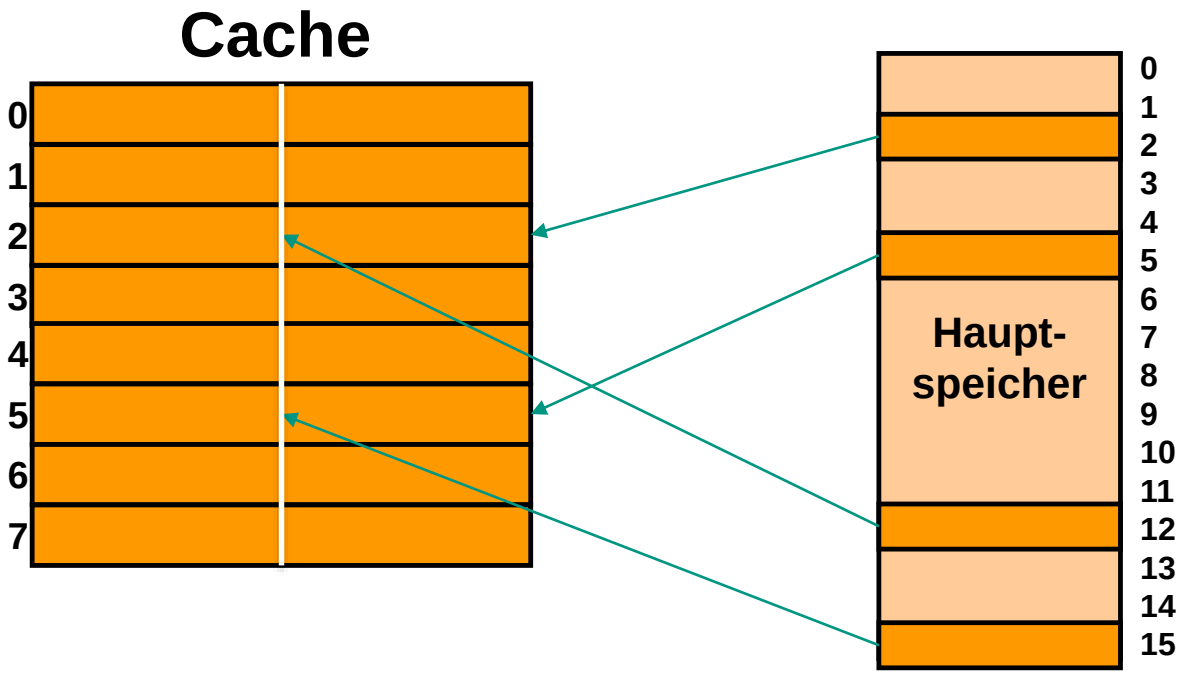
- 2. Verfahren: direkt-abbildend (direct mapped)
  - Jede Datenadresse wird anhand einer Abbildungsvorschrift (z.B. einen Teil der Adresse („Index“) fest auf eine bestimmte Stelle im Cache abgebildet
  - Nachteil → Dopplungen möglich, denn zwei Blöcke können sich gegenseitig verdrängen („gleicher Index“), obwohl Cache ansonsten komplett leer ist



Cacheorganisation

A: Wo kann ein Block im Cache platziert werden?

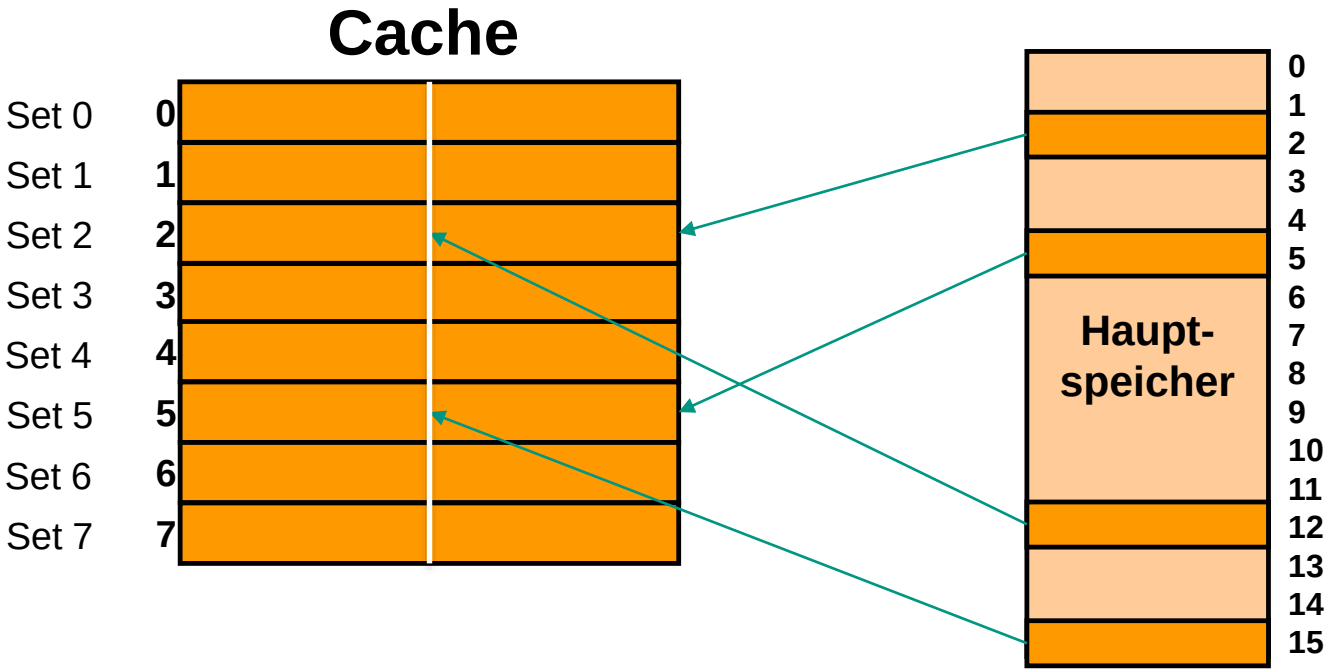
- 3. Verfahren: n-Wege assoziativ
  - Kombination aus vollassoziativen und Direct Mapped Cache



Cacheorganisation

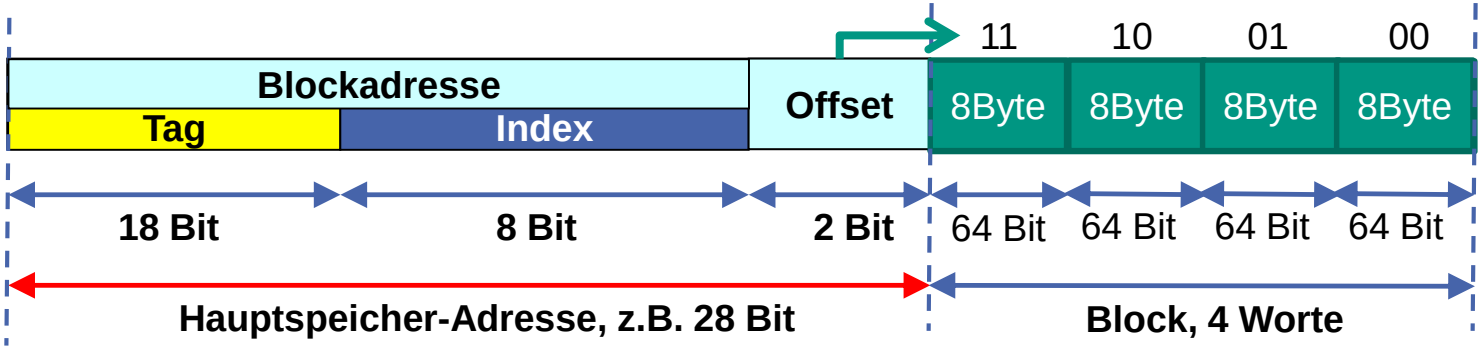
A: Wo kann ein Block im Cache platziert werden?

- 3. Verfahren: n-Wege assoziativ
  - Kombination aus vollassoziativen und Direct Mapped Cache
  - Speicheradresse wird direkt auf 1 aus S Sets im Cache abgebildet,
    - Set besteht aus n Cacheblöcken
    - Vergleich mit allen n Blöcken innerhalb des Sets
  - Bsp.: 2-wege assoziativ, jedem Index stehen 2 Felder zur Verfügung



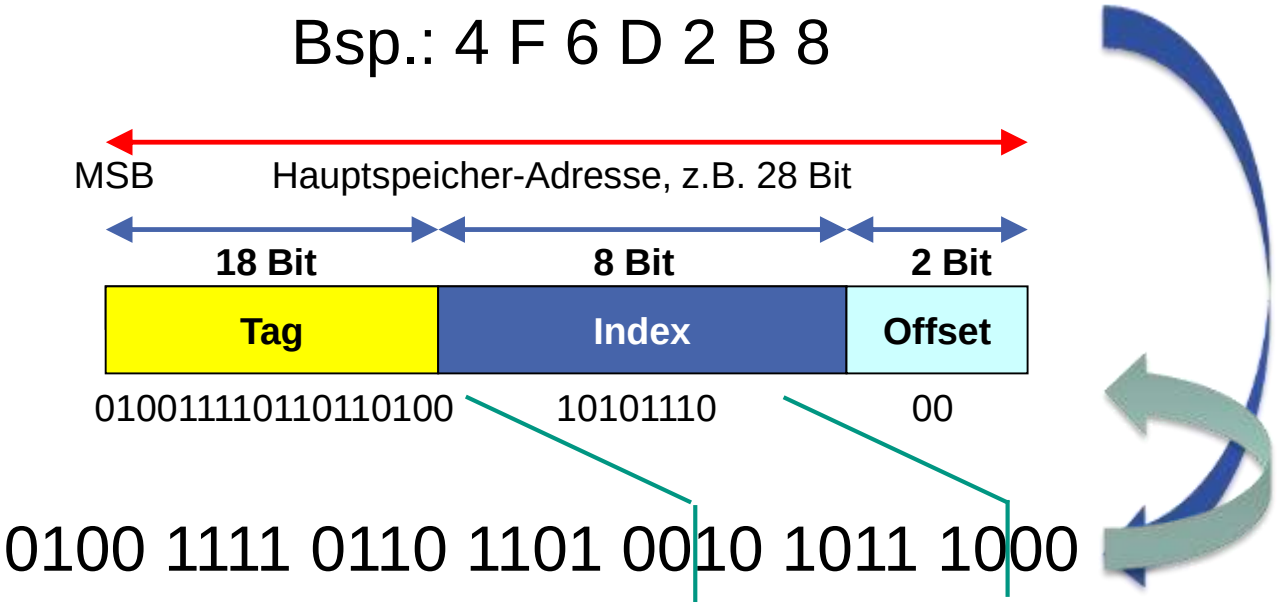
Cache-Organisation

- Antwort auf Frage B: Wie kann festgestellt werden, ob ein Block im Cache ist?
  - Verfahren bei direkt abbildendem Cache
    - Annahme: pro Block im Cache werden k Worte gespeichert, z.B. 4 Worte zu 8 Byte bei 64Bit-Datenbusbreite vom Prozessor zum Cache und Hauptspeicher
    - Der *Offset* ist die Adresse innerhalb eines Blockes, z.B. 2 Bits bei 4 Worten pro Block
    - *Index* gibt die Adresse im Cache an (im Beispiel 256*4 Worte im Cache)
    - *Tag* ist ein Teil der originalen Hauptspeicheradresse, meist der höchstwertige



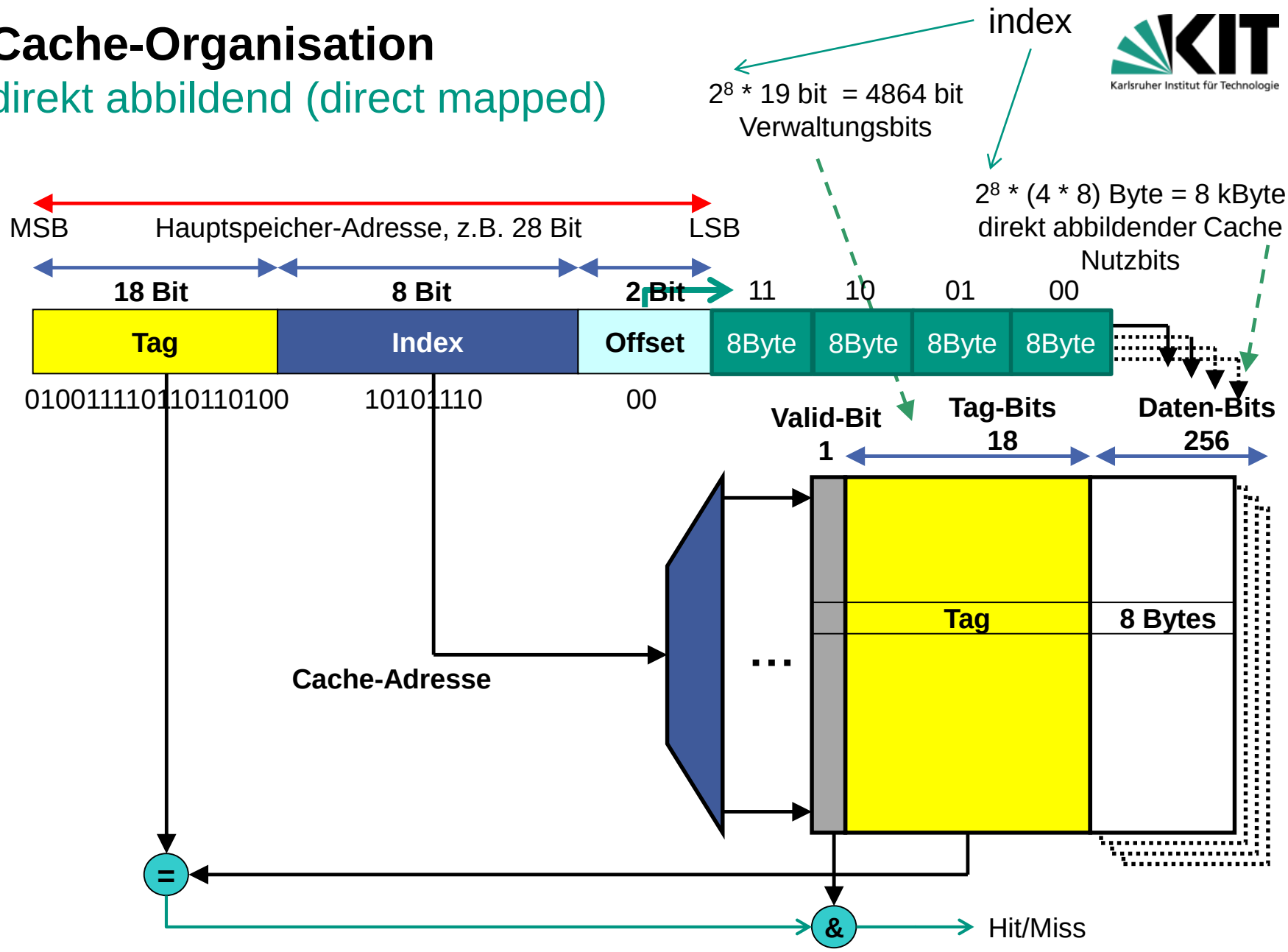
    - 28 Bit Adresse für 64-Bit Worte

Hauptspeicheradresse



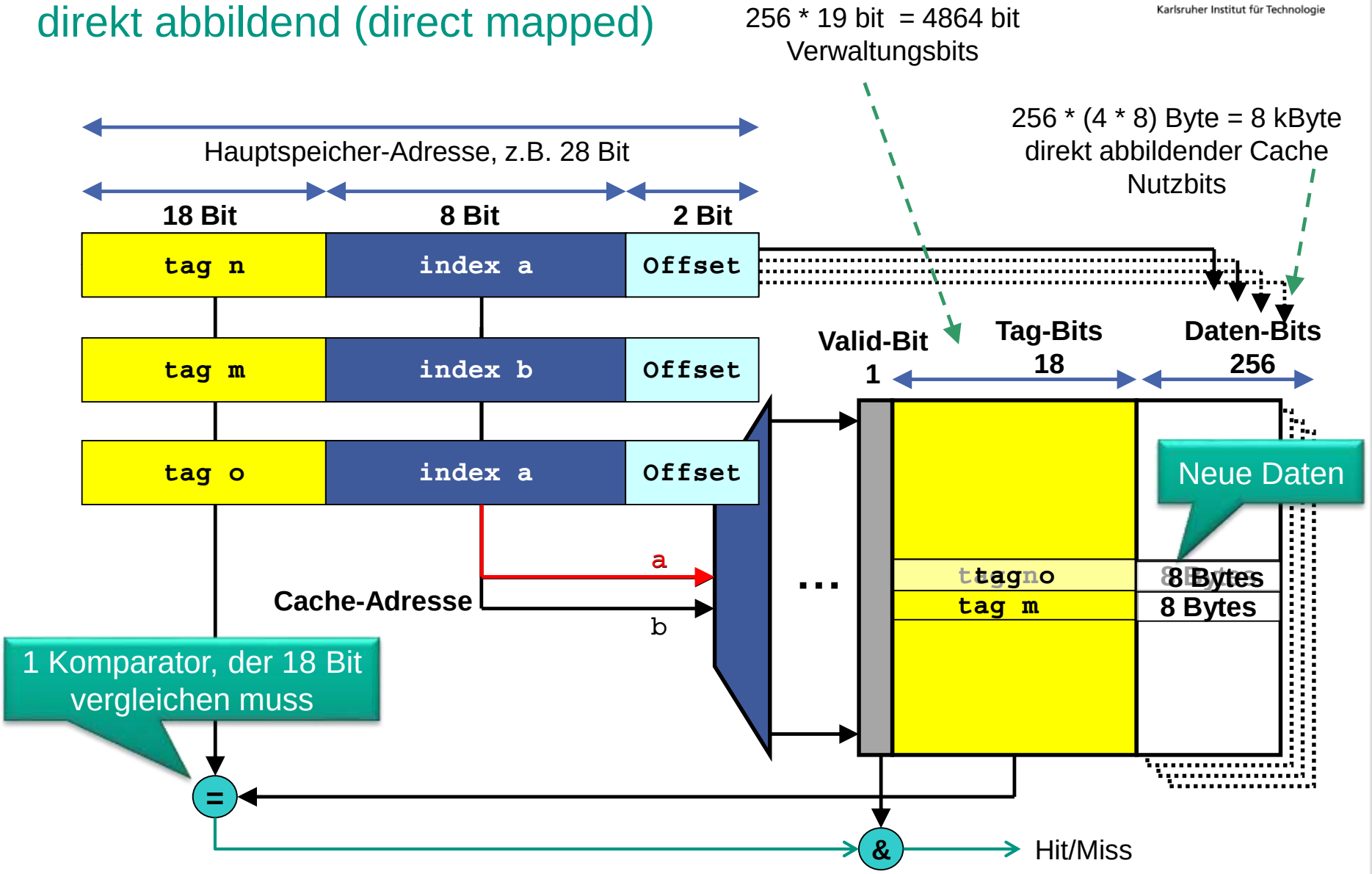
Cache-Organisation

direkt abbildend (direct mapped)

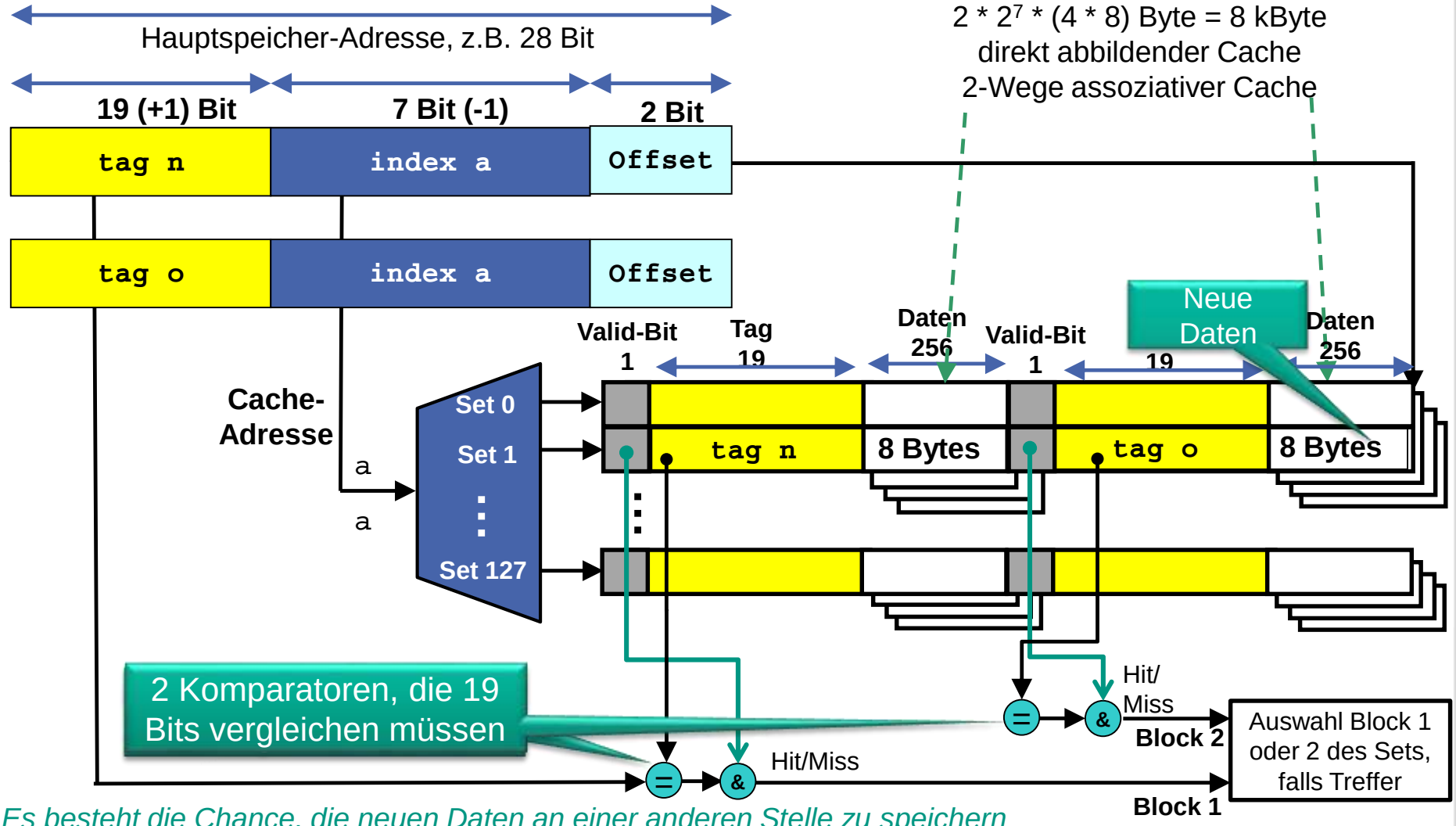


Cache-Organisation

direkt abbildend (direct mapped)



Cacheorganisation
2-Wege assoziativ



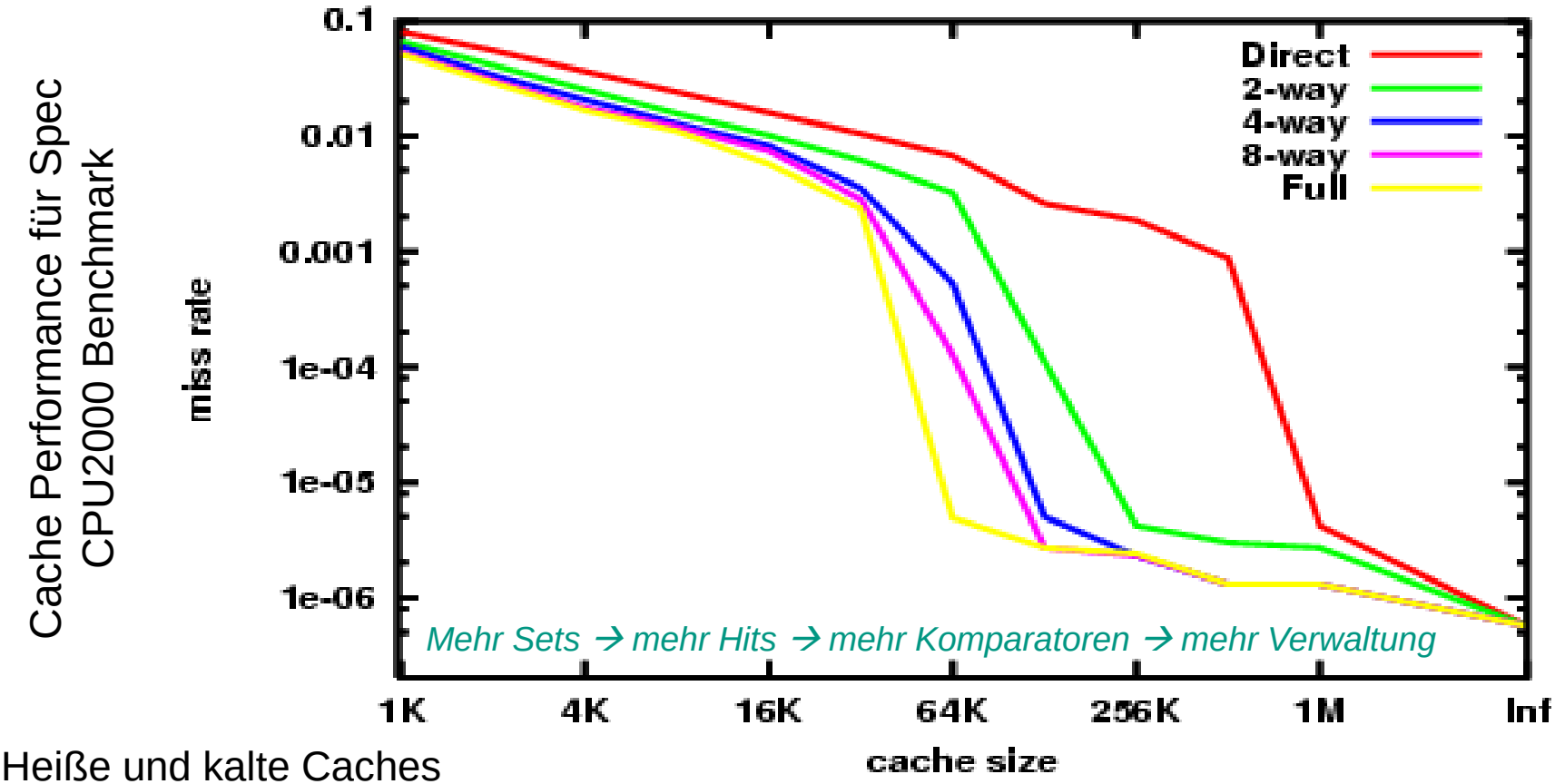
Es besteht die Chance, die neuen Daten an einer anderen Stelle zu speichern

Cache-Organisation - Verdrängungsstrategien
(bei mehrwege-assoziativ)

- LRU-Strategie (Least Recently Used)
  - Der Eintrag, auf den am längsten nicht zugegriffen wurde, wird verdrängt
- LFR-Strategie (Least Frequently Used)
  - Der am seltensten gelesene Eintrag wird verdrängt
- FIFO-Strategie (First In First Out)
  - Der jeweils älteste Eintrag wird verdrängt
- Zufallsstrategie
  - geringster Hardware-Aufwand
- Optimale Ersetzungsstrategie:
  - der Block, dessen Daten in Zukunft am längsten (zeitlich) nicht mehr angesprochen werden, wird ausgelagert
  - Nachteil: diese Information steht meist nicht zur Verfügung

$$t_{averageaccess} = hitrate * t_{hit} + (1 - hitrate) * t_{miss}$$

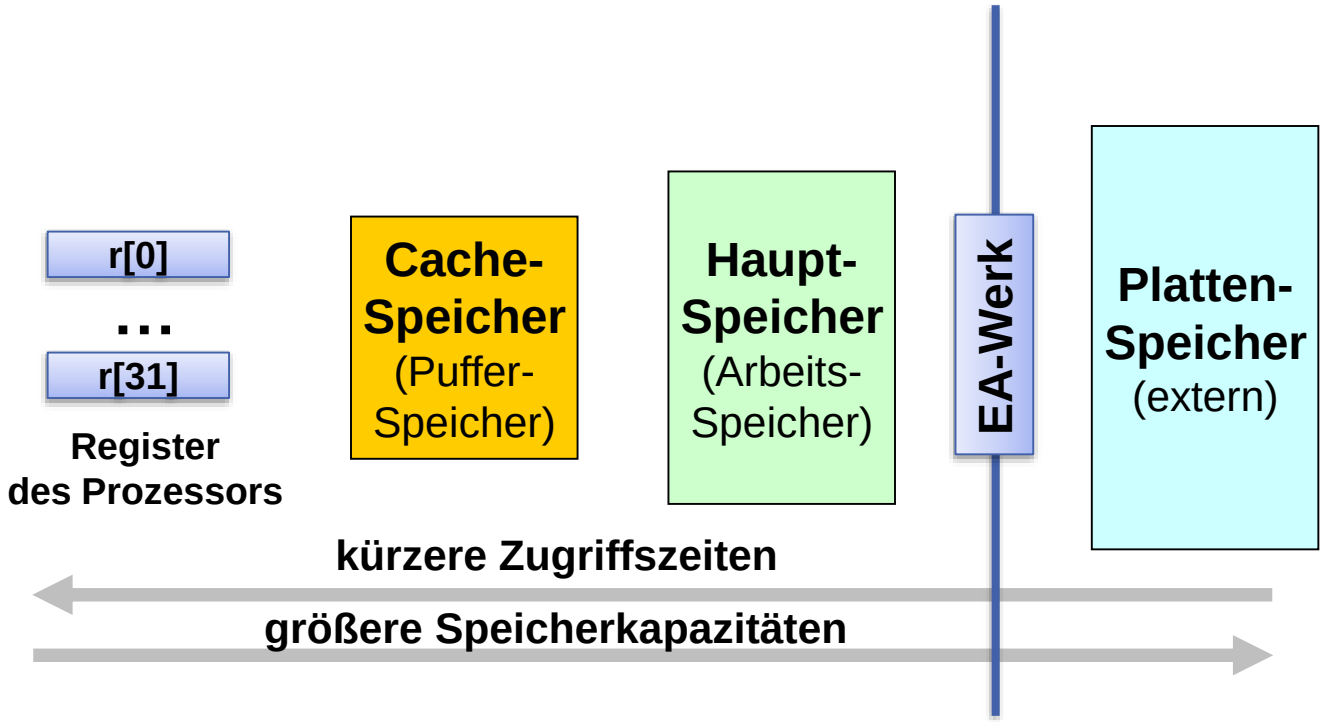
Caches – Missrate Vergleich



- Heiße und kalte Caches
  - Ein Cache ist „heiß“, wenn er optimal arbeitet, also gefüllt ist und nur wenige Cache Misses hat; ist das nicht der Fall, gilt der Cache als „kalt“.
  - Ein Cache ist nach Inbetriebnahme zunächst kalt, da er noch keine Daten enthält und häufig zeitraubend Daten nachladen muss, und wärmt sich dann zunehmend auf, da die zwischengelagerten Daten immer mehr den angeforderten entsprechen und weniger Nachladen erforderlich ist.

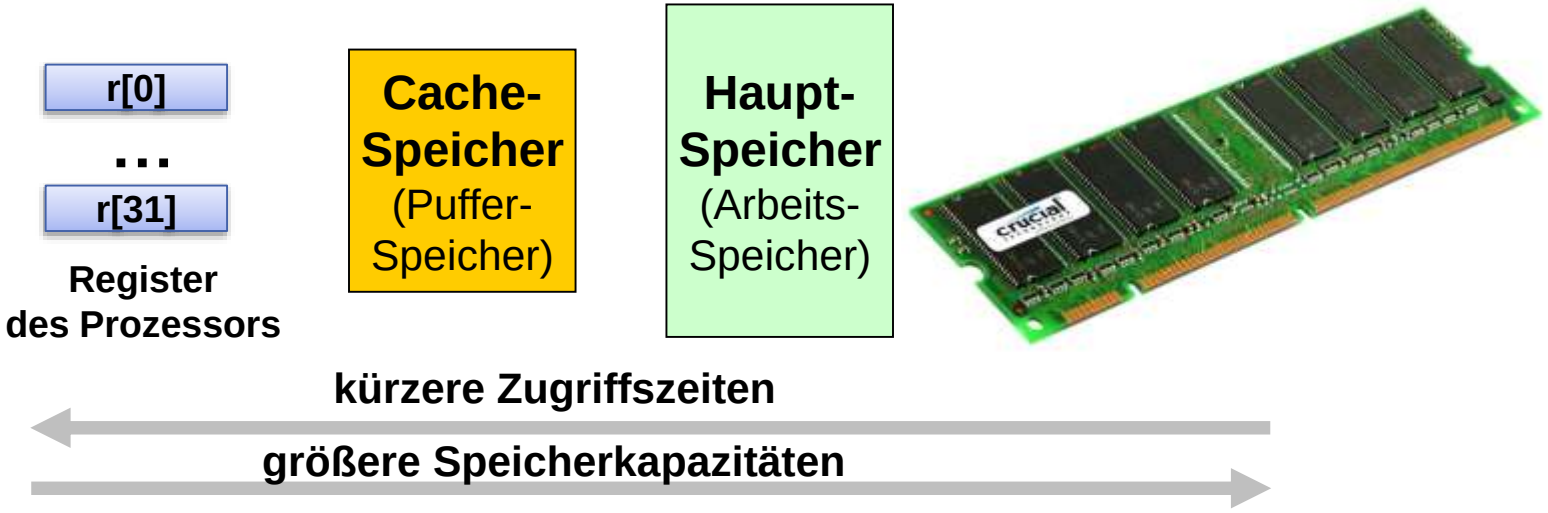
Speicherhierarchie

- Kriterien
  - Kosten
  - Zugriffsgeschwindigkeit
  - Energieverbrauch
- Speicherhierarchie in einem Rechner



Speicherhierarchie

- Kriterien
  - Kosten
  - Zugriffsgeschwindigkeit
  - Energieverbrauch
- Speicherhierarchie in einem Rechner



Elektronische Speicherung – Halbleiterspeicher

Speicherung auf Basis elektronischer Bauelemente

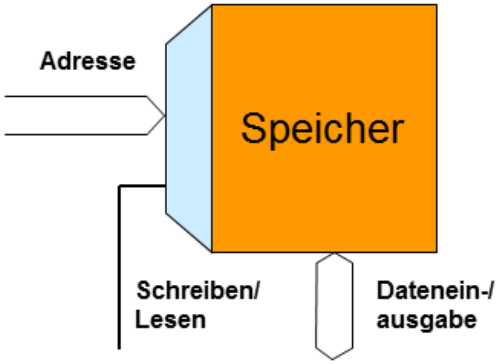
- Einzelne Speichermechanismen können nach der Charakteristik der Datenhaltung unterschieden werden:
  - *flüchtige* Speicher,
    deren Informationen verloren gehen, wenn der Strom abgeschaltet wird
    - DRAM, dynamisches RAM (*dynamic random access memory, periodical refresh*)
    - SRAM (*static random access memory, no refresh*)
  - *permanente* Speicher,
    in denen sich eine einmal gespeicherte oder festverdrahtete Information befindet, die nicht mehr verändert werden kann
    - ROM (*read only memory*)
    - PROM (*programmable read only memory*)
  - *semi-permanente* Speicher,
    die Informationen permanent speichern,
    aber Informationen auch
    verändert werden können.
    - EPROM (*erasable programmable read only memory*)
    - EEPROM (*electrically erasable programmable read only memory*)
    - Flash-EEPROM (USB-Stick)

Externer Speicher

Ein-/Ausgabewerk

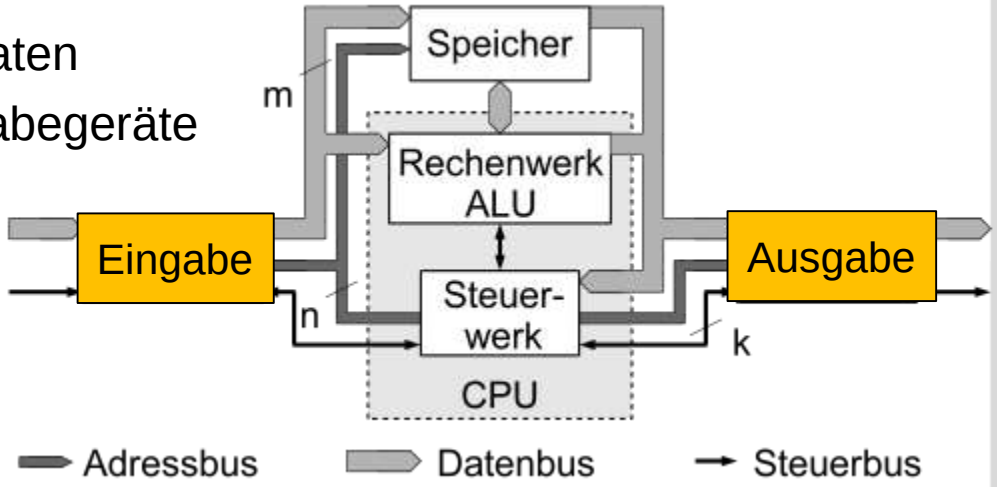
Speicherwerk 1

- Festplattenspeicher (persistent) gehören nicht zum Speicherwerk, sondern zum E/A-Werk
  - Daten werden von der Festplatte zunächst in den Arbeitsspeicher geladen, bevor sie verarbeitet werden können.
  - *Spezialfall: Virtueller Arbeitsspeicher auf externer Festplatte (extrem langsam aber kostengünstig).*



Ein-/Ausgabewerk

- Kommunikation mit dem Benutzer
  - Ein-/Ausgabegeräte (Tastatur, Maus, Bildschirm, Sensor)
  - Peripheriegeräte (z.B. Drucker, Plattenspeicher; Aktuator)
- Eingabe:
  - Übernahme Daten von extern angeschlossenen Geräten (z.B. Sensor)
  - Umsetzung in eine geeignete, normierte Darstellung zur weiteren Verarbeitung
- Ausgabe:
  - Datenformatmäßige und elektrische Aufbereitung der Daten
  - Weiterleitung an externe Ausgabegeräte
  - Eventuell Auswahl des Ausgabegerätes
  - Ansteuern eine Aktors



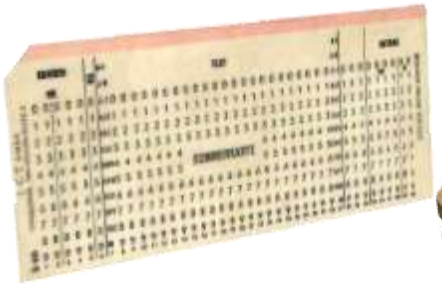
Datenspeicher (fotografisch und mechanisch)

- Fotographische Speicherung
  - Speicher, die durch einen chemischen Bearbeitungsprozess Daten in Form von Lichtbildern (statischen und bewegten Bildern sowie Lichtton) speichern.
  - chemo-optische Speicherform.
  - Speicherung auf Mikrofilm ist immer noch die zur Zeit sicherste Archivierungsmethode
    → Zum Lesen ist nur ein Vergrößerungsgerät notwendig
  - Probleme mit der Dauerhaftigkeit von Formaten und Lesegeräten entfallen, existieren jedoch mit den Datenträgern
  - Beispiele
    - Filme
    - fotografische Papiere
    - Mikrofilme
    - …

- Mechanische Speicherung
  - Es sind physisch Vertiefungen bzw. Erhöhungen im Trägermaterial auf das Speichermedium aufgebracht.
  - Die gefertigten Speichermedien können nur gelesen werden.
  - Mechanischer Lesevorgang
    - analoge Medien
      - Wachswalze
      - LP (Langspielplatte, Vinyl)
    - digitale Medien
      - Lochkarte
      - Lochstreifen
  - optischer Lesevorgang (Laser)
    - nur bei „gepressten“ Medien – per Laser beschriebene Medien siehe „Optische Speicherung“

Datenspeicher (fotografisch und mechanisch)

- Fotographische Speicherung



- Mechanische Speicherung
  - Es sind physisch Vertiefungen bzw. Erhöhungen im Trägermaterial auf das Speichermedium aufgebracht.
  - Die gefertigten Speichermedien können nur gelesen werden.
  - Mechanischer Lesevorgang
    - analoge Medien
      - Wachswalze
      - LP (Langspielplatte, Vinyl)
    - digitale Medien
      - Lochkarte
      - Lochstreifen
  - optischer Lesevorgang (Laser)
    - nur bei „gepressten“ Medien – per Laser beschriebene Medien siehe „Optische Speicherung“

Speicherung (optisch und magneto-optisch)

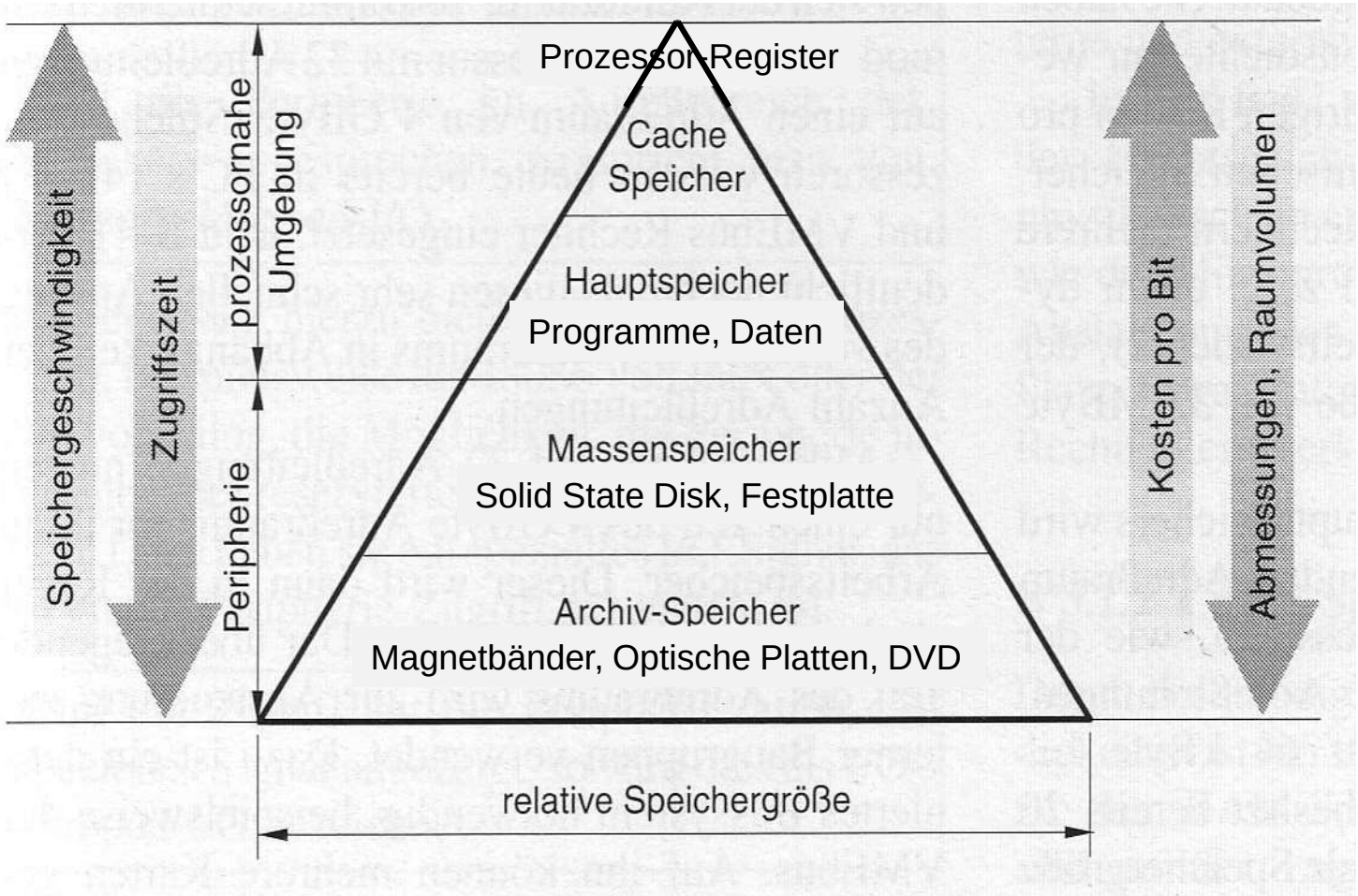
- Zum Lesen und Schreiben der Daten wird ein Laserstrahl verwendet.
  - optische Speicherung nutzt dabei die Reflexions- und Beugungseigenschaften des Speichermediums aus, z. B. bei CDs/DVDs die Reflexionseigenschaften und bei holografischen Speichern die lichtbeugenden Eigenschaften.
  - Die Speicherform ist ausschließlich digital.

Magnetische Speicherung

- Die magnetische Speicherung von Informationen erfolgt auf magnetisierbarem Material.
  - Auf Bänder, Karten, Papier oder Platten aufgebracht
  - Magnetische Medien werden mittels eines Lese-Schreib-Kopfes gelesen bzw. beschrieben.
  - Auf dem Medium werden Daten analog, digital oder in beiden Formen gespeichert.
- Typische Speichermedien
  - digitale Medien
    - Magnetband
      (z. B. DLT); DAT und Tonband (digital)
    - Magnetkarte, Magnetstreifen
    - Compact Cassette (Datasette)
    - Trommelspeicher
      - Festplatte (hard disk)
      - Diskette (floppy disk)
    - Wechselplatte z. B. Zip-Diskette
  - analoge Medien
    - Tonband (Musikkassette)
    - Videoband (Videokassette)

Prozessor-Speicher-System

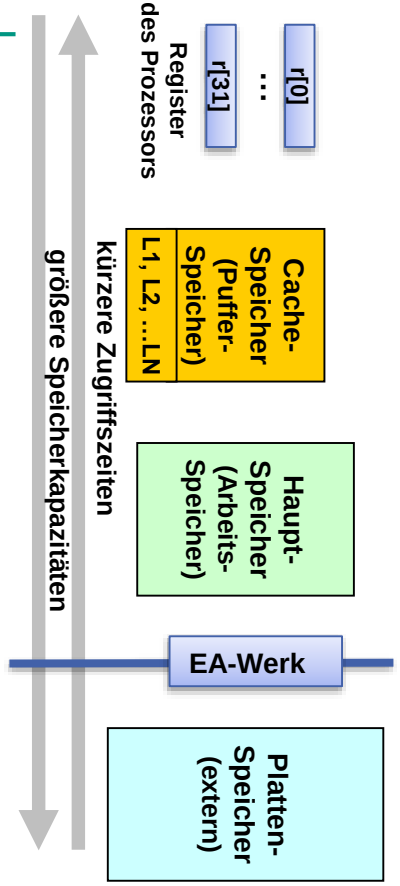
- Hierachische Speicherstruktur in einem Rechnersystem



Hierarchie Prozessor-Speicher-System

Zugriffszeiten auf unterschiedliche Speicher

Speicherart	Zugriffszeit	relativ
Prozessor-Register	100 ps bis 1 ns	0,005
Prozessorcache L1	1 ns bis 5 ns	0,05
Prozessorcache L2	2 ns bis 20 ns	0,1
Prozessorcache L3	30 ns bis 50 ns	0,4
Arbeitsspeicher (DRAM, DDR-SDRAM)	50 ns bis 200 ns	1
Massenspeicher (Festplatte, Solid State Disk)	8 ms bis 10 ms (0,5 ms bis 1 ms)	50
Wechseldatenträger (Magnetband, DVD, USB-Stick)	Sekunden bis Minuten	>1000



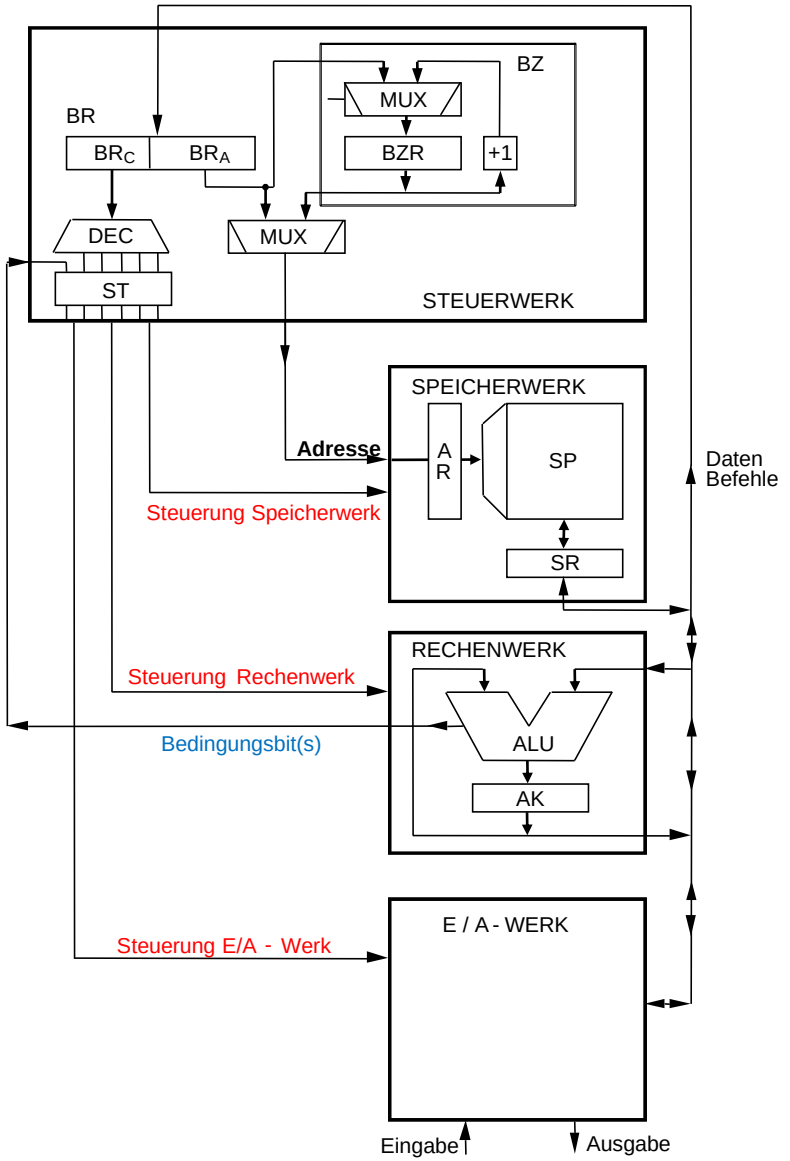
Zusammenfassung von Neumann


Fragen?

Aufbau eines von Neumann Rechners

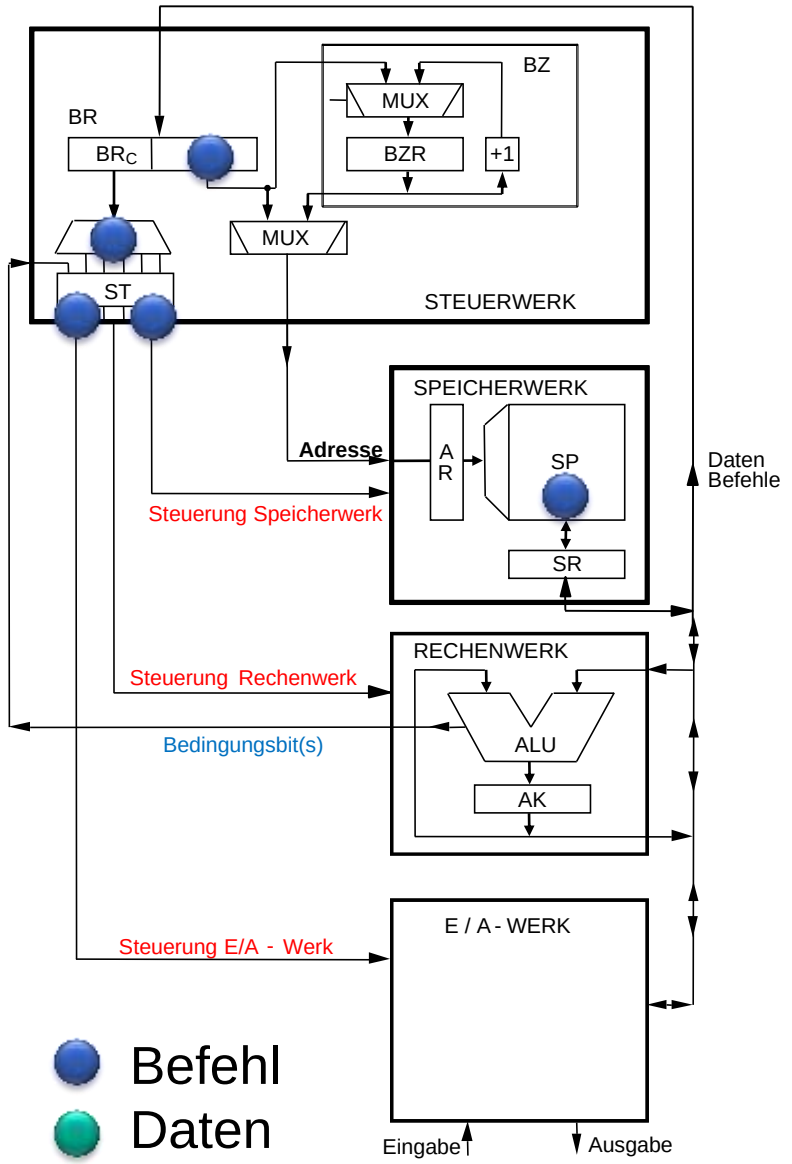
Detaillierte Sichtweise

- Steuerwerk
  - BR: Befehlsregister
    - BRC: Codeteil (Opcode)
    - BRA: Adressteil
  - ST: Register für Steuerbits
  - BZ: Befehlszähler
    - BZR: Befehlszählerregister
- Speicherwerk
  - AR: Adresszähler
  - SP: Speicher
  - SR: Speicherregister
- Rechenwerk
  - ALU: Arithmetische-, logische Einheit
  - AK: Akkumulator (Register)
- E/A-Werk
  - E/A: Eingabe/Ausgabe



Von-Neumann-Zyklus (Arbeitsweise)

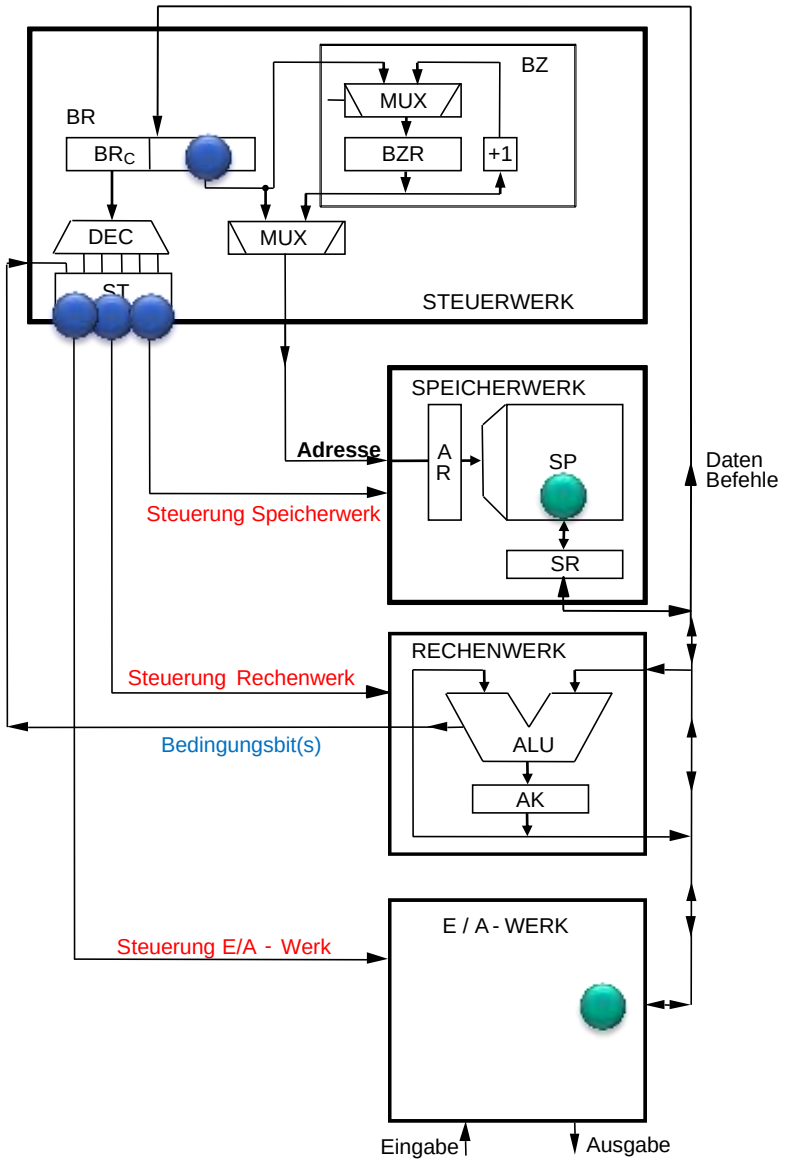
1. FETCH:
   Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
2. DECODE:
   Befehl im Steuerwerk dekodieren
3. FETCH OPERANDS:
   gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
4. EXECUTE:
   Befehl im Rechenwerk ausführen
5. WRITE BACK:
   gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)
6. Weiter bei Schritt 1



Von-Neumann-Zyklus (Arbeitsweise)

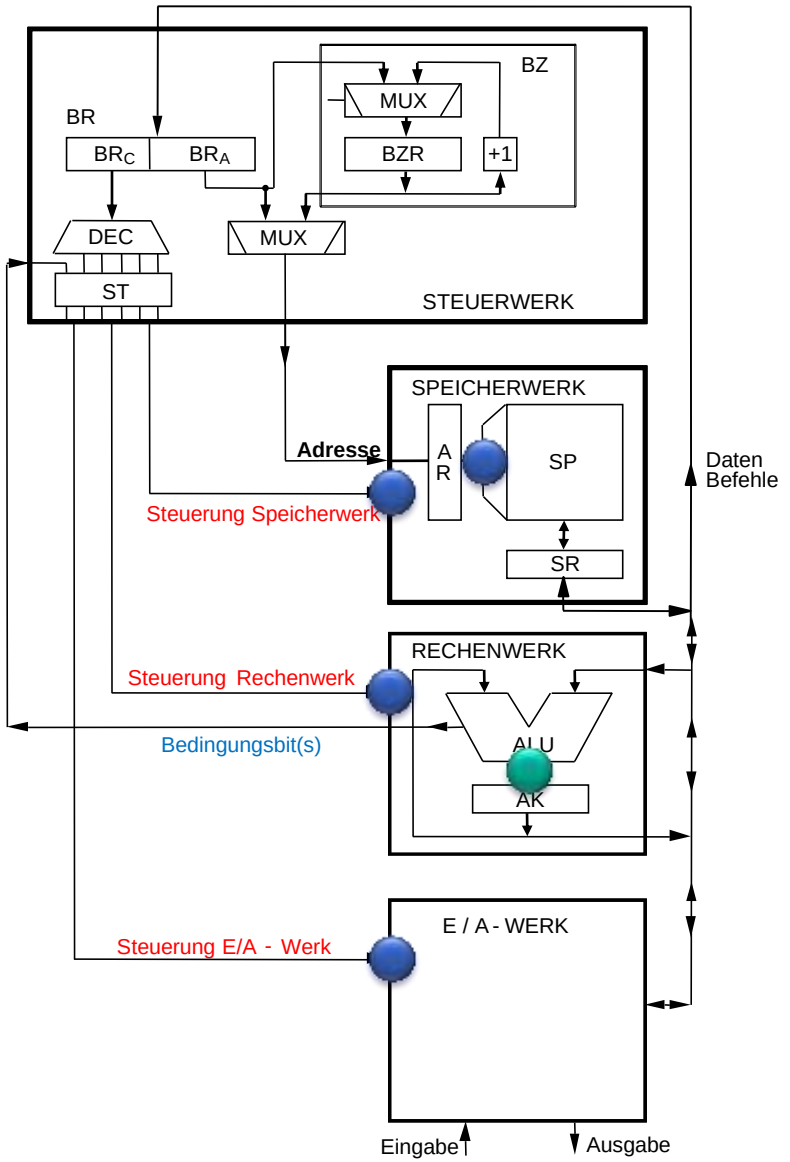
- FETCH:
  Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
- DECODE:
  Befehl im Steuerwerk dekodieren
- FETCH OPERANDS:
  gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
- EXECUTE:
  Befehl im Rechenwerk ausführen,

5. WRITE BACK:
   gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)
6. Weiter bei Schritt 1



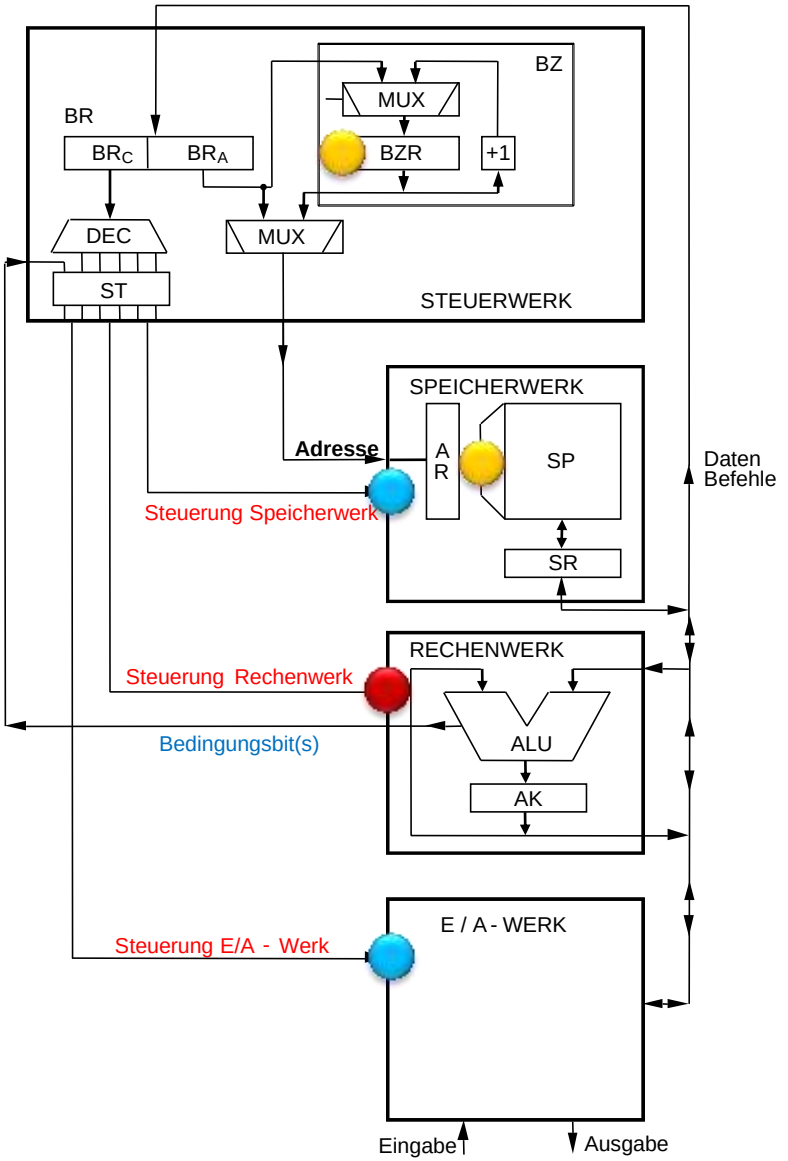
Von-Neumann-Zyklus (Arbeitsweise)

- FETCH:
  Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
- DECODE:
  Befehl im Steuerwerk dekodieren
- FETCH OPERANDS:
  gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
- EXECUTE:
  Befehl im Rechenwerk ausführen
- WRITE BACK:
  gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)
- Weiter bei Schritt 1



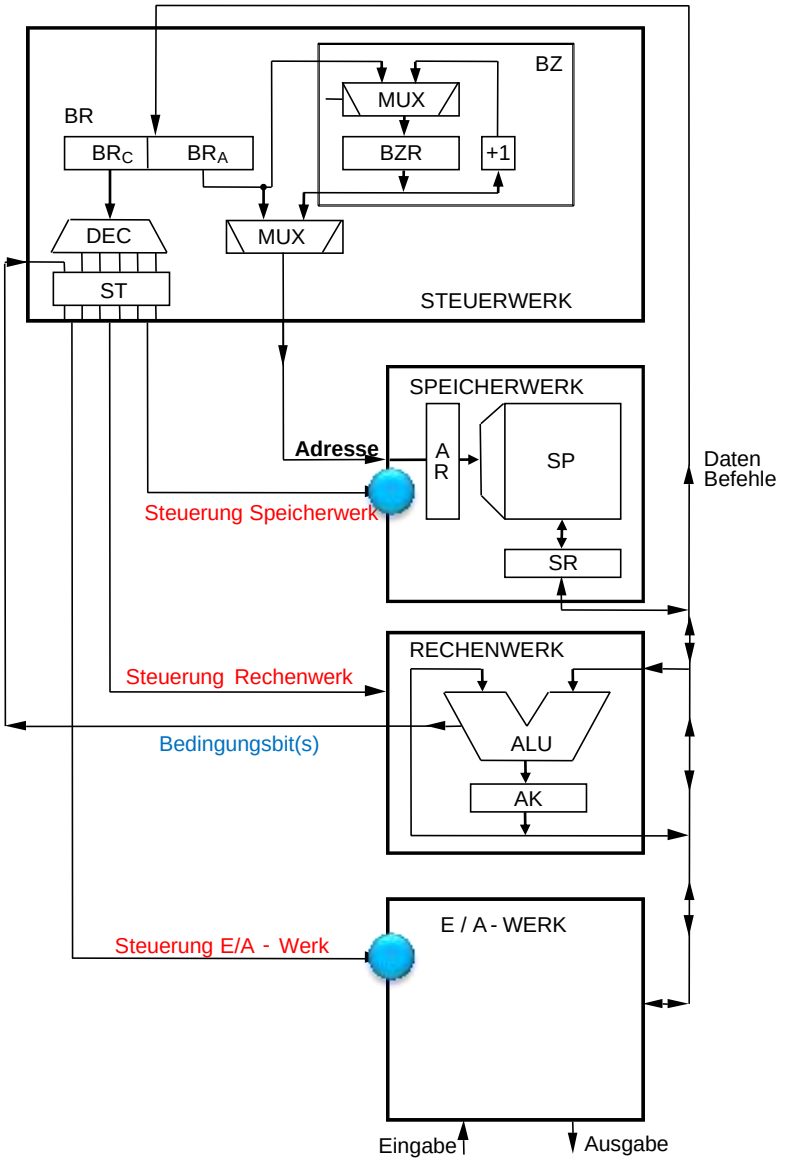
Befehlstypen

- Transportbefehl
- Verarbeitungsbefehle (in der ALU)
- Steuerbefehle



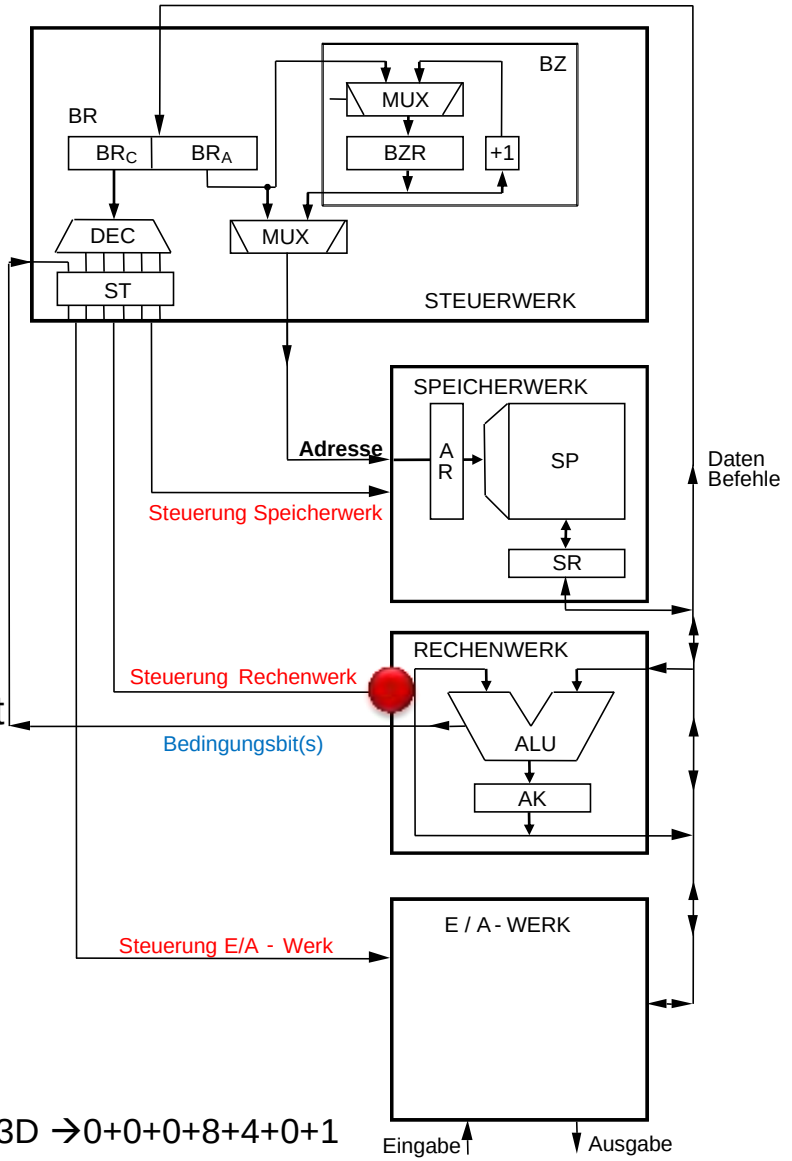
Befehlstypen

- Transportbefehl
  - Externer Datentransfer:
    - Ladebefehl, der Daten aus dem Speicher oder der Peripherie in den Prozessor holt
    - Befehl zum Speichern für die umgekehrte Richtung.
  - Interner Datentransfer (Register-Register)



Befehlstypen

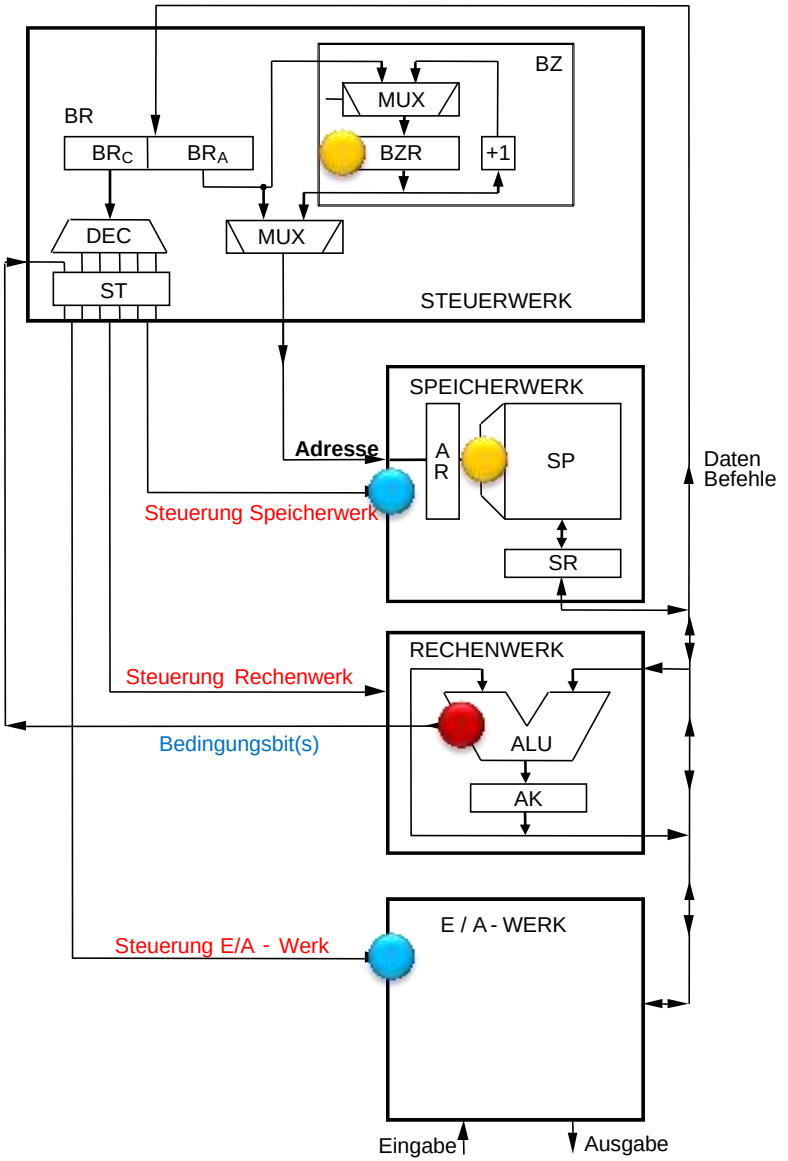
- Verarbeitungsbefehle (in der ALU)
  - Vergleichsbefehle
  - Arithmetische und logische Operationen
    - Beispiel für eine arithmetische Operation ist die Addition
  - Schiebebefehle
    - Der Shiftbefehl zum Schieben von Bits in einem Speicherwort kann ebenfalls als arithmetische Operation aufgefasst werden.
    - Ein Shift nach links verdoppelt den Wert (wenn man von Überträgen absieht).
    - Ein Shift nach rechts halbiert ihn
    - Beispiel: Arithmetische und logische Operationen SHR führt einen Rechts-Shift des Akkumulatorinhalts um i Bits durch.
      - shr 2 , (0110111) = (0001101)



0+32+16+0+4+2+1→ 55D → 52D (ohne Übertrag) → 26D → 13D →0+0+0+8+4+0+1

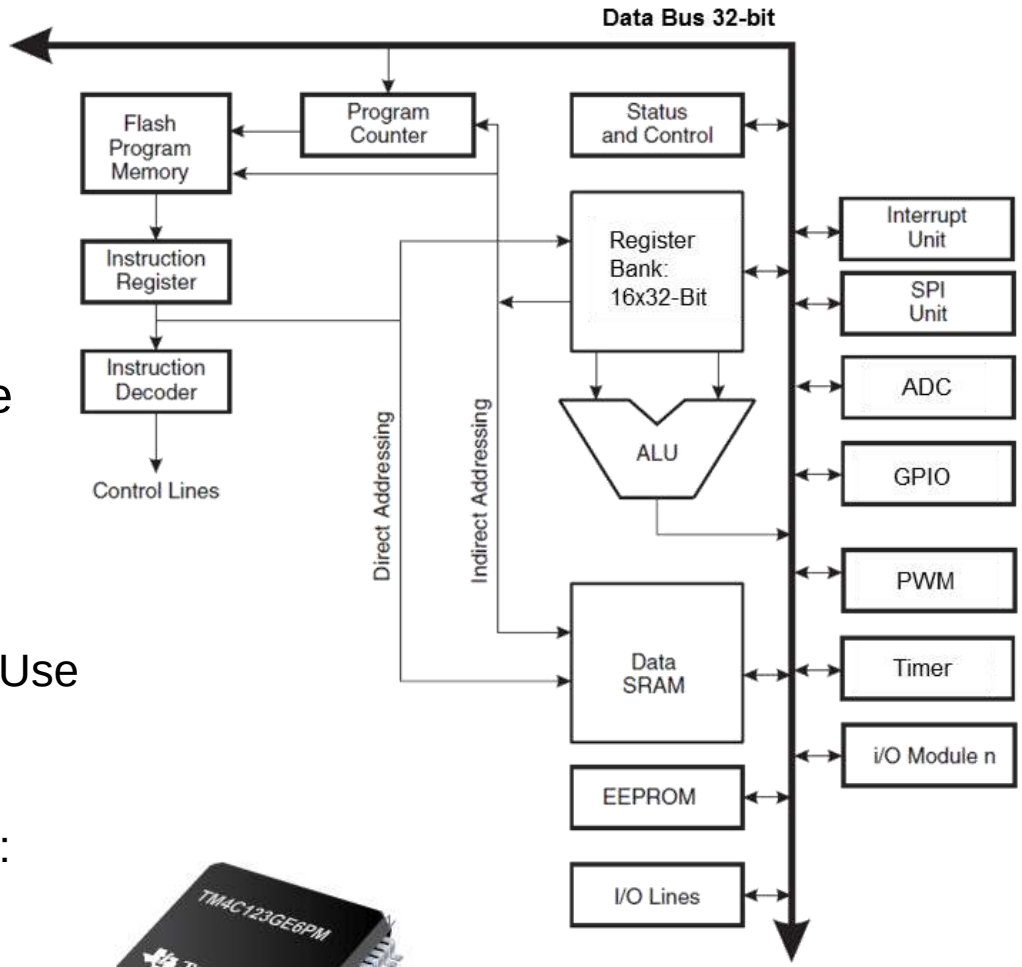
Befehlstypen

- Steuerbefehle
  - unbedingter Sprung
    - goto j
  - bedingter Sprung
    - if <Bedingung> then goto k
    - <Bedingung> ist zum Beispiel a < 0 oder a = g.
  - Vergleich und Überspringen der nächsten Operation (Skip) bei Gleichheit
  - Unterprogrammaufruf
    - call <Unterprogrammname/-adresse>
  - Unterprogrammrücksprung
    - return



Aufbau einer CPU – Beispiel ARM Cortex-M4

- 32-bit Prozessor basiert auf der ARMv7 Architektur
  - veröffentlicht im Jahr 2010
- Speziell entwickelt, um leistungsstarke und low-cost Geräte im Bereich der digitalen Signalsteuerung für eingebettete Systeme zu realisieren
  - Einfacher Aufbau und leichte Programmierbarkeit
  - Angesiedelt im extremen Low-Cost, Low-Power and Ease-of-Use Bereich
- Zahlreiche Quellen:
  - Online verfügbare Dokumente : https://developer.arm.com
  - S. Doku auf Ilias
- Verwendet im PIT Labor

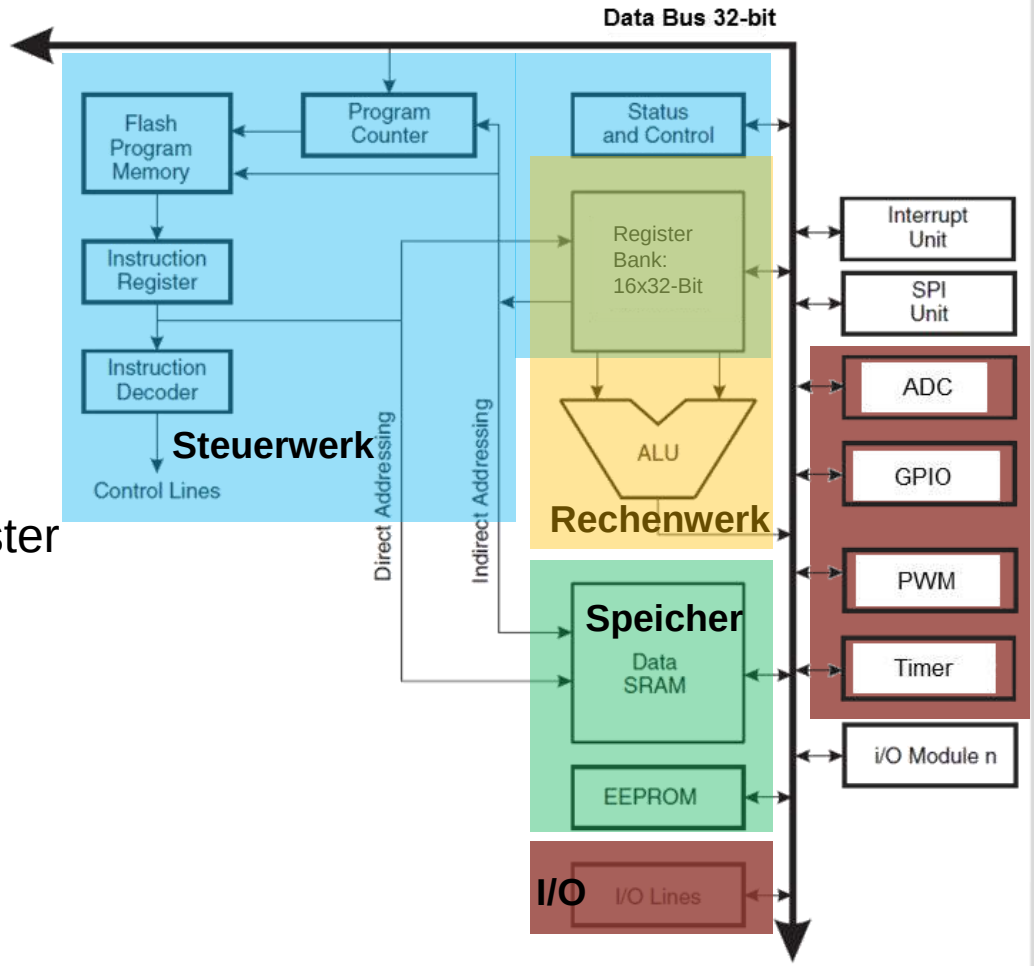


Verallgemeinerte Prozessorarchitektur für Cortex-M4

***TM4C123GE6PM High performance 32-bit ARM® Cortex®-M4F based MCU**

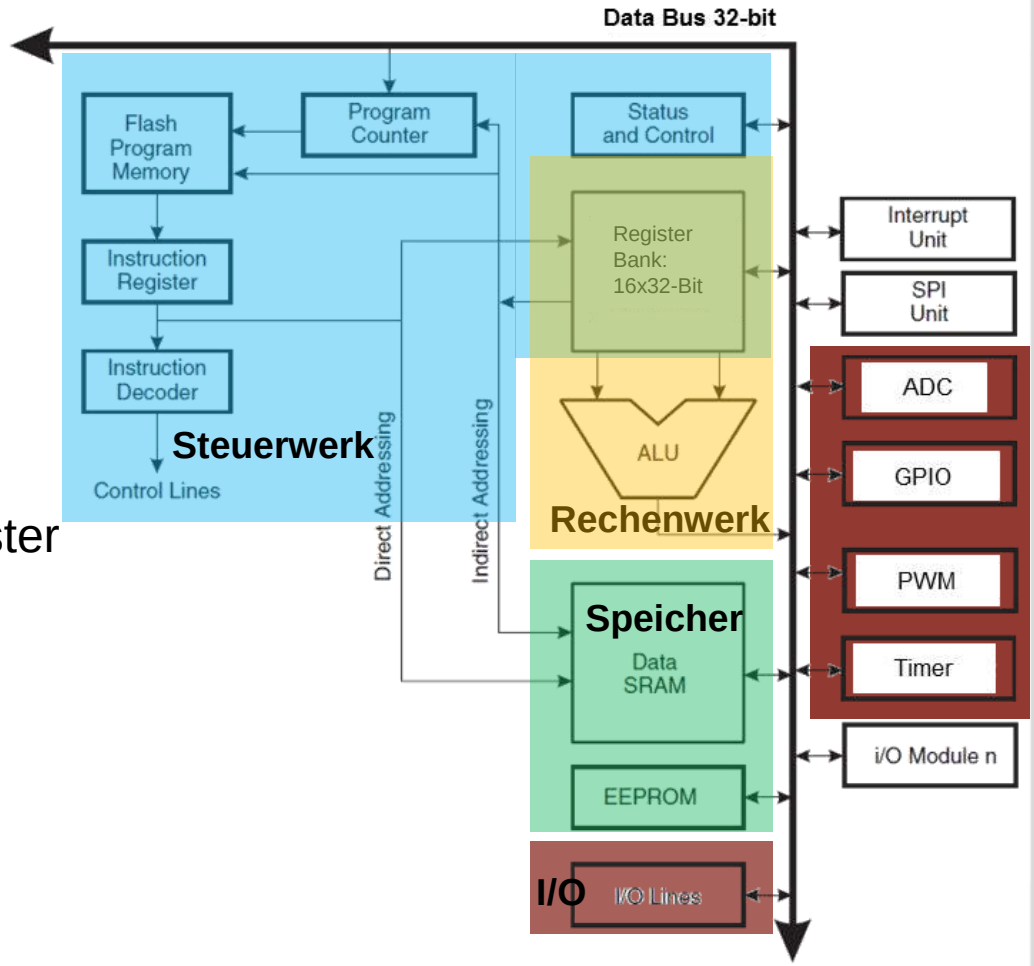
Aufbau einer CPU – Beispiel ARM Cortex-M4

- Harvard Architektur (s. nächstes Kapitel):
  - separate Data-Bus und Instruction-Bus
- 32-Bit RISC Architektur
  - 32 Bit Load/Store
  - 32 Bit Verarbeitungsbreite
  - 16 x32 Bit Register (13 Register sind General-Purpose)
  - 211 Instruktionen
  - 32 Bit Instruktion

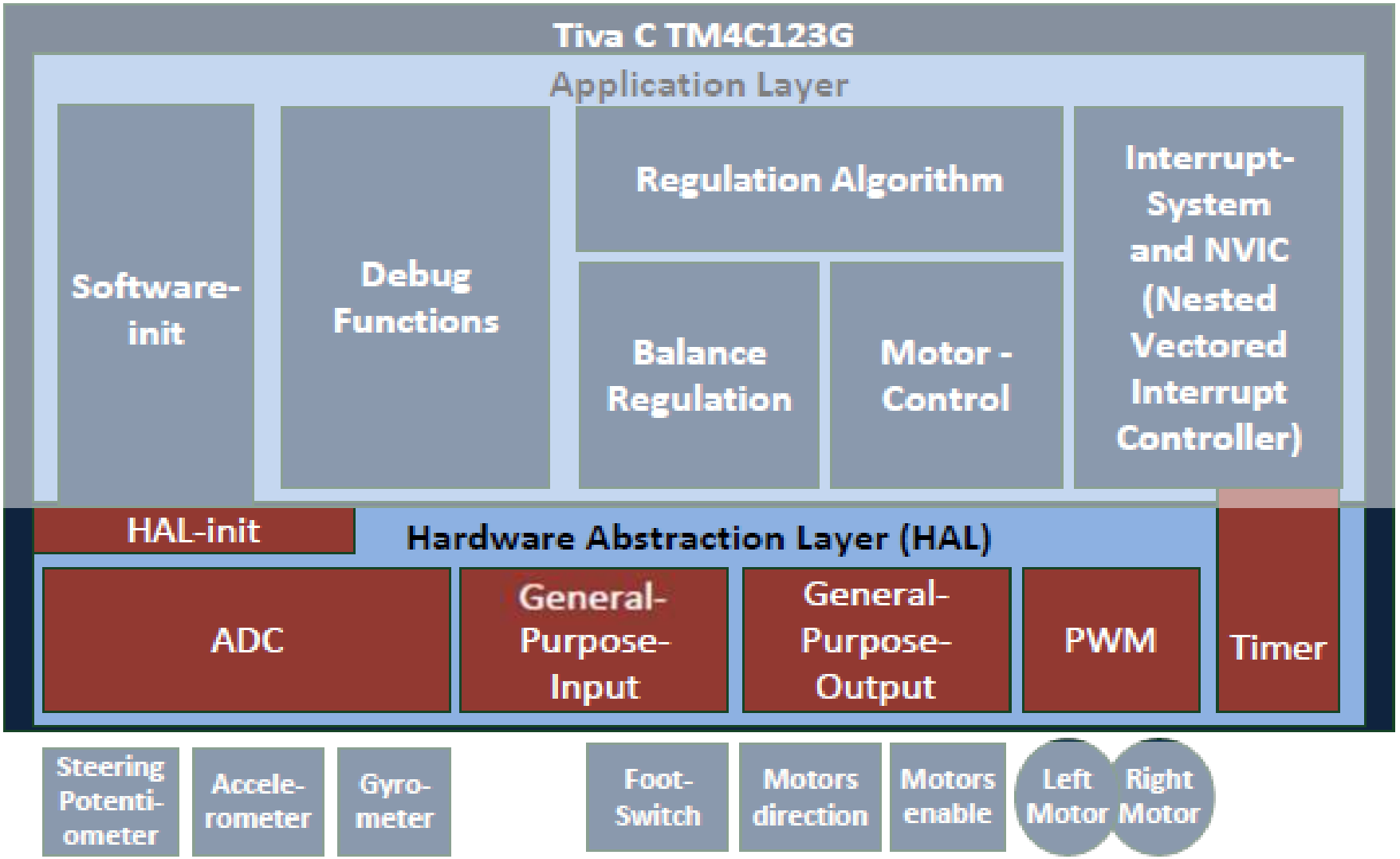


Aufbau einer CPU – Beispiel ARM Cortex-M4

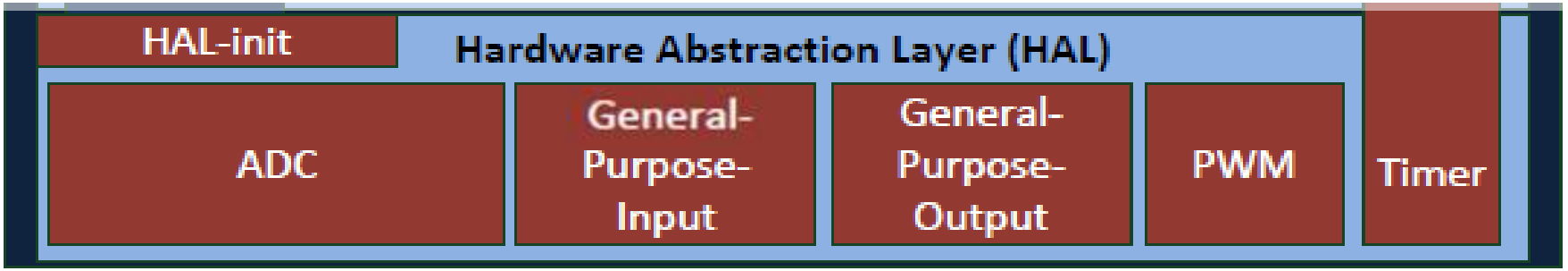
- Harvard Architektur (s. nächstes Kapitel):
  - separate Data-Bus und Instruction-Bus
- 32-Bit RISC Architektur
  - 32 Bit Load/Store
  - 32 Bit Verarbeitungsbreite
  - 16 x32 Bit Register (13 Register sind General-Purpose)
  - 211 Instruktionen
  - 32 Bit Instruktion



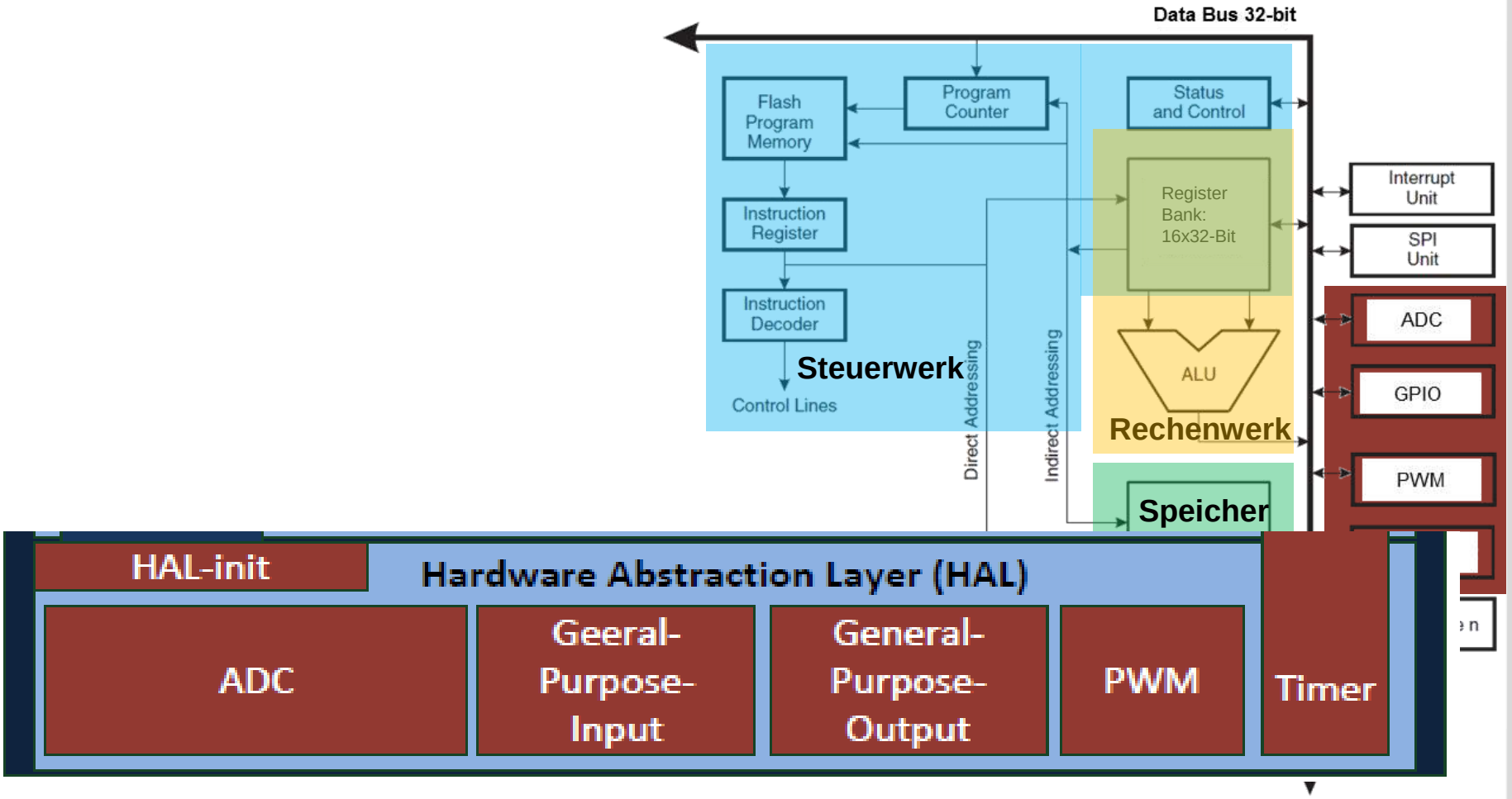
Entwicklungsplattform Praktikum IT



Entwicklungsplattform Praktikum IT



Aufbau einer CPU – Beispiel ARM Cortex-M4

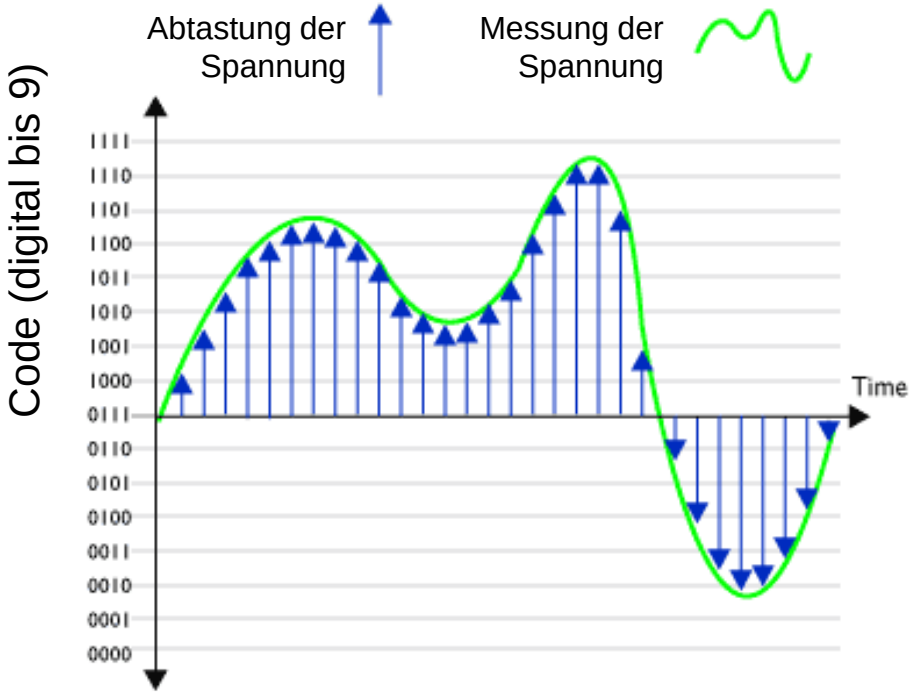


i/O Module

- General Purpose Input/Output (GPIO)
  - Verhalten unabhängig als Eingabe- oder Ausgabe programmierbar
  - Beispiele: Ansteuerung von LEDs, Auslesen von Tastern
- Analog/Digital Wandler (ADC)
  - Umsetzung analoger Eingangssignale in einen digitalen Datenstrom
  - Beispiele: Einlesen der Temperatur, Neigung, …
- Pulsweitenmodulation (PWM)
  - Ändern des Abtastverhältnisses bei konstanter Frequenz
  - Beispiele: Geschwindigkeitsregelung von Motoren

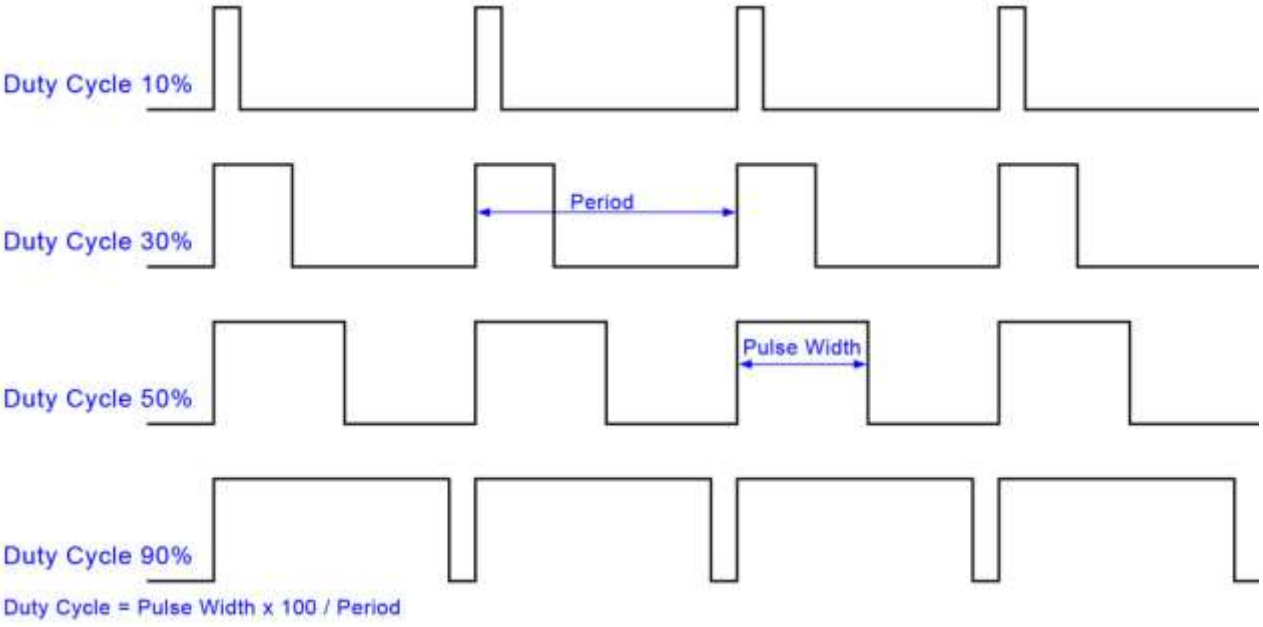
Analog/Digital Wandler (ADC)

- Wandelt analoge Eingangssignale in digitale Werte
- Amplitude der kontinuierlichen analogen Signale werden diskret abgetastet und als digitale Werte abgespeichert.
- Der abtastbare Spannungsbereich hängt von der Versorgungsspannung des Microcontrollers ab



Pulsweitenmodulation (PWM)

- Änderung des Puls-Pause-Verhältnisses bei gleicher Frequenz
  - übertragene Energie hängt von Puls-Pause-Verhältnis ab
- Durch Änderung der Pulsbreite pro Periode kann Drehzahl von Motoren digital geändert werden
  - 0 → 1 Wechsel immer zum Start der Periode
  - 1 → 0 Wechsel wird durch parallel laufenden Zähler ermittelt



Von Neumann Architektur

IT Inhalt

0. Einleitung und Motivation

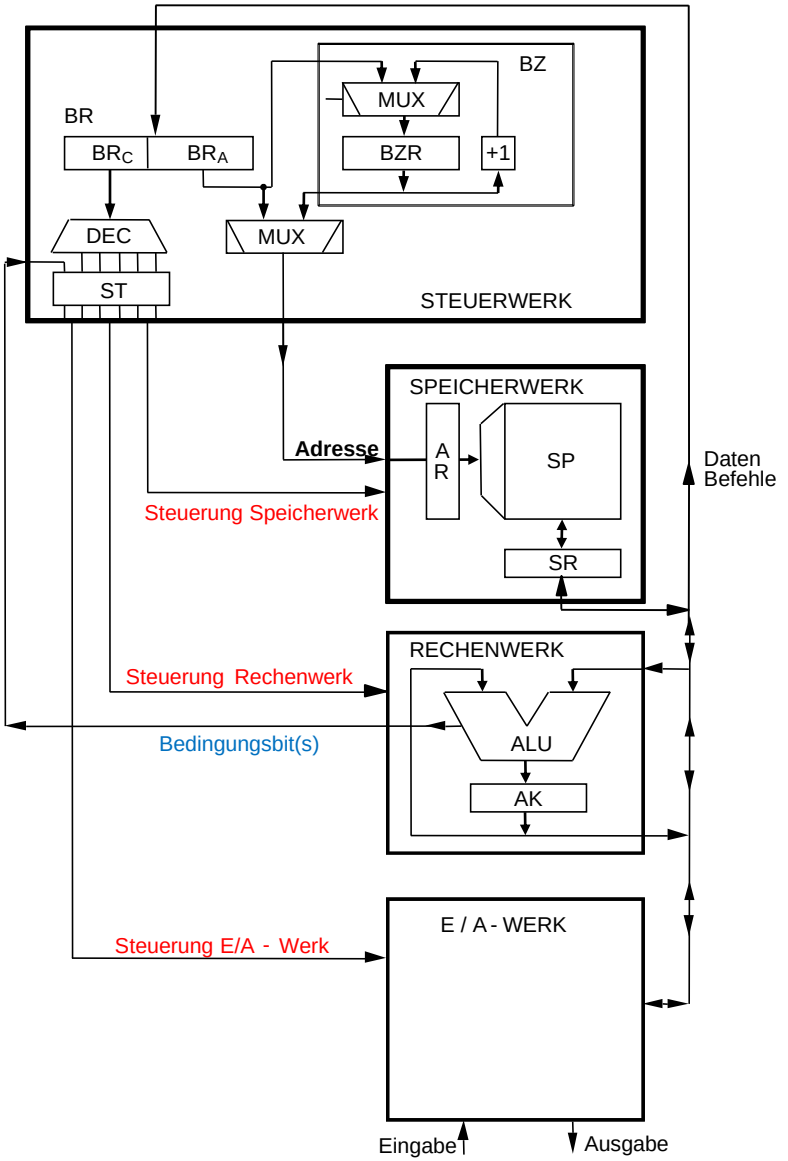
- Organisatorisches
- Begriffe
- Typische Anwendungen

1. Rechnerarchitekturen

- Einführung
- Allgemeiner Aufbau der internen Hardware Architektur
- Instruction Set Architecture (ISA) am Beispiel ARM Cortex M4
- Klassifikation von Rechnerarchitekturen
- Performanzsteigerung
- ausgewählte Beispiele

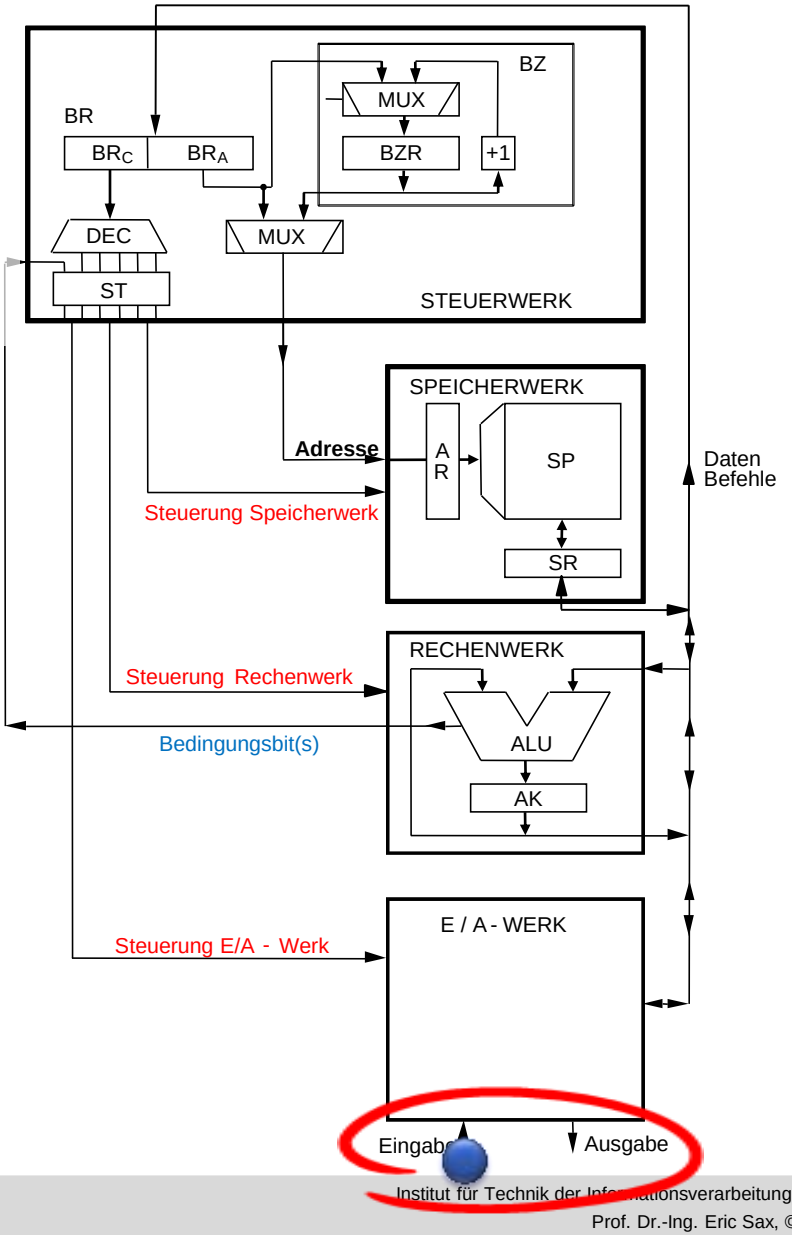
Von-Neumann-Zyklus (Arbeitsweise)

1. FETCH:
   Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
2. DECODE:
   Befehl im Steuerwerk dekodieren
3. FETCH OPERANDS:
   gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
4. EXECUTE:
   Befehl im Rechenwerk ausführen, ggf. Veränderung des Sprungzählers
5. WRITE BACK:
   gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)
6. Weiter bei Schritt 1



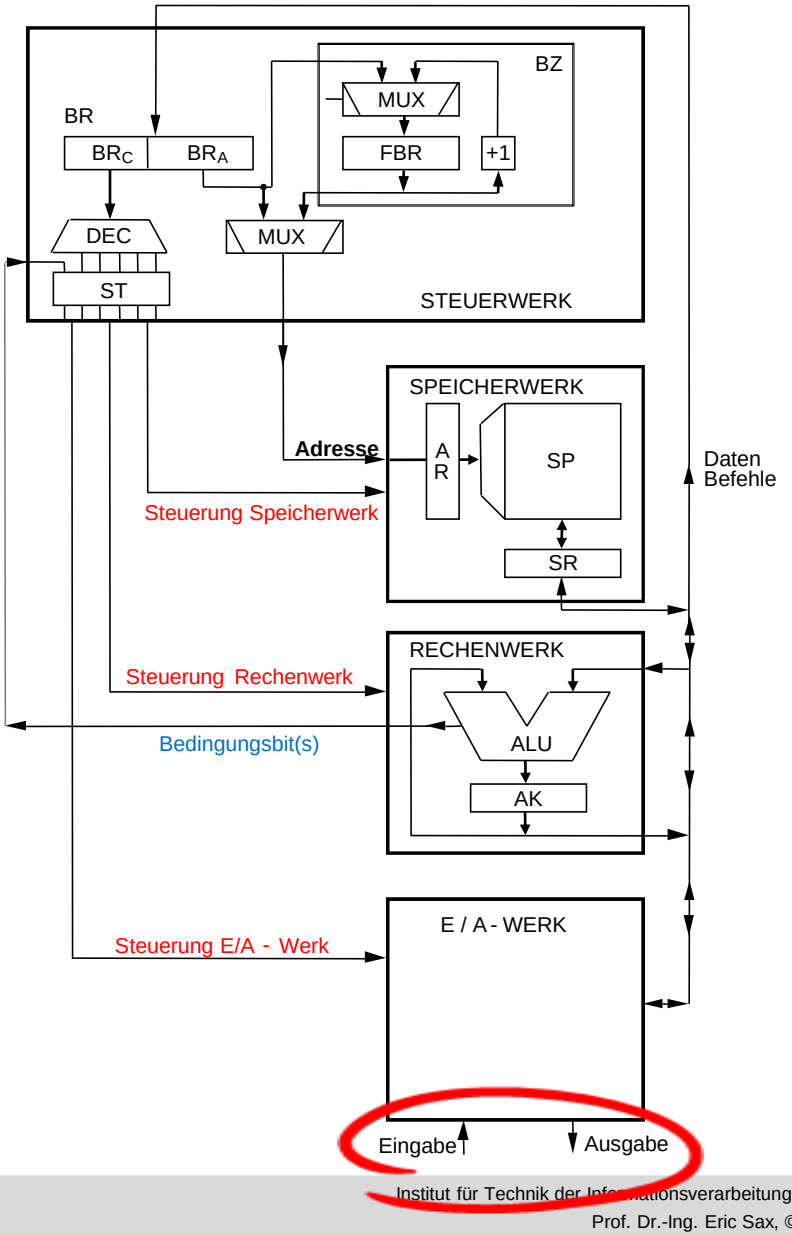
Von-Neumann-Zyklus (Arbeitsweise)

1. FETCH:
   Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
2. DECODE:
   Befehl im Steuerwerk dekodieren
3. FETCH OPERANDS:
   gegebenenfalls Operanden aus dem Speicher oder der Peripherie in den Prozessor holen (lesender Zugriff)
4. EXECUTE:
   Befehl im Rechenwerk ausführen, ggf. Veränderung des Sprungzählers
5. WRITE BACK:
   gegebenenfalls Ergebnis in den Speicher oder die Peripherie schreiben (schreibender Zugriff)
6. Weiter bei Schritt 1



Sicht des Programmierers

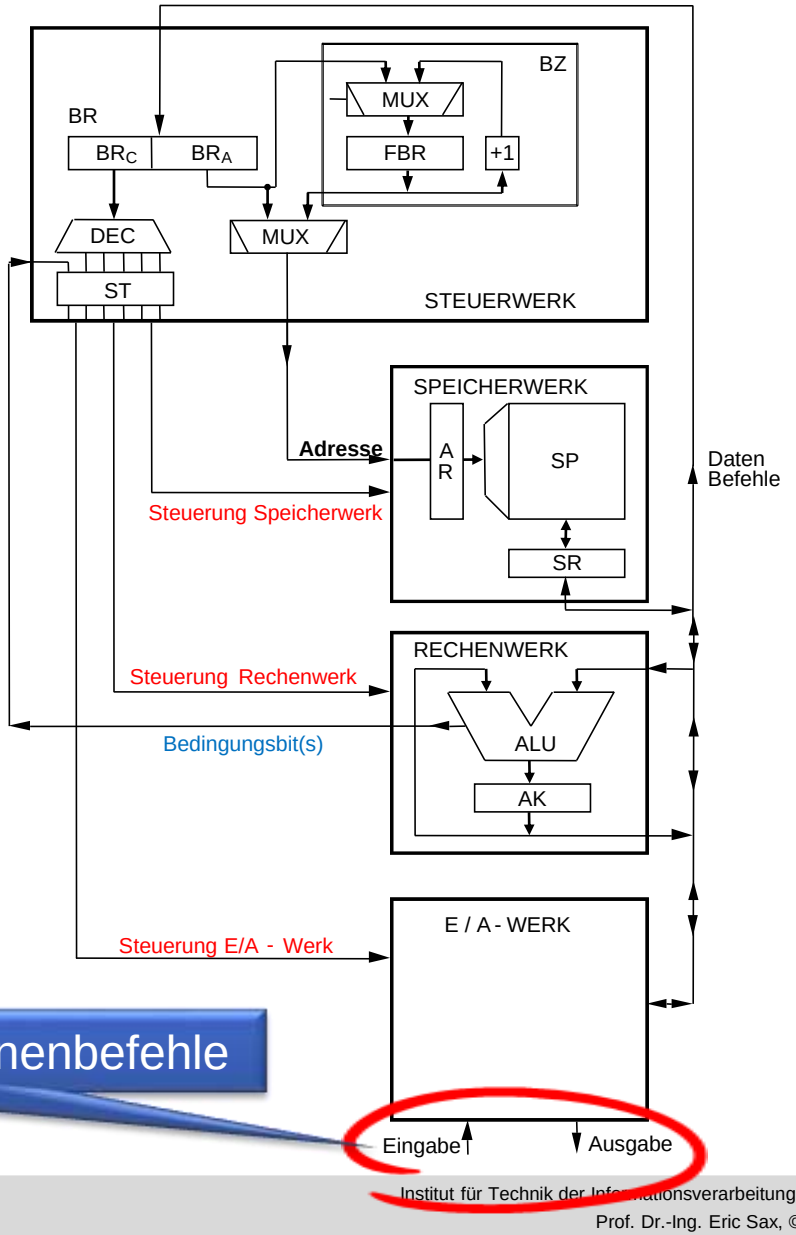
- Sicht des Programmierers auf den Computer hängt ab von der Antwort auf fünf Fragen:
  - Wie werden Daten repräsentiert?
  - Wo können Daten gespeichert werden?
  - Wie kann auf Daten zugegriffen werden?
  - Welche Operationen können mit den Daten ausgeführt werden?
  - Wie werden Befehle codiert?
- Antwort auf diese Fragen definiert die externe Architektur oder Befehlssatz-architektur (*Instruction Set Architecture, ISA)*
  - Häufig wird ISA mit der Prozessor Architektur gleichgesetzt



Sicht des Programmierers

- Interne Architektur (Prozessor Mikroarchitektur) definiert die interne Struktur des Prozessors
  - Verschiedene interne Architekturen können dieselbe externe Architektur aufweisen
- Maschinenbefehlssatz definiert die dem Programmierer sichtbaren Befehle (externe Architektur)

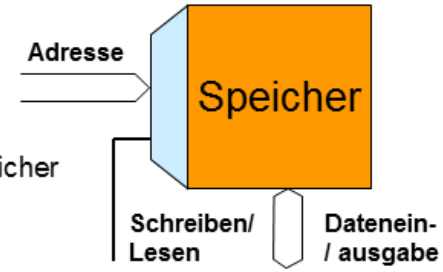
Hochsprachen → Assembler → Maschinenbefehle



Arbeitsspeicher/Hauptspeicher

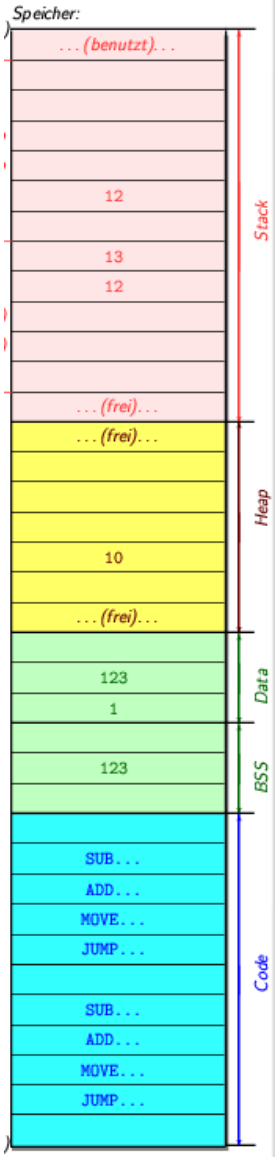
Speicherwerk 1

- Speicherwerk ist frei adressierbarer Schreib/Lese-Speicher
  - nicht für jede Operation muss das Ein-/Ausgabewerk benutzt werden
- Komponenten
  - Hauptspeicher
    - Enthält die aktuell auszuführenden Daten oder Programmteile (flüchtig)
    - Enthält (je nach Architektur) die Firmware (BIOS) (persistent)
  - Cache
    - Schneller Pufferspeicher für bereits einmal geladenen Daten (flüchtig)
  - Register im Rechnerkern
    - Speicherbereich, der direkt mit der eigentlichen Recheneinheit verbunden ist und die unmittelbaren Operanden und Ergebnisse aller Berechnungen aufnimmt (flüchtig)
    - Register sind in der Regel höchstens so groß wie Wortgröße des Prozessorkerns (8, 16, 32, 64 Bit).
    - Die Gesamtheit aller Register bezeichnet man als dessen Registersatz.
    - Verschiedene Register dienen zum Zwischenspeichern von Befehlen, Speicheradressen, Rechenoperanden und in Peripheriebausteinen auch zum Zwischenspeichern von Ein- und Ausgabewerten.
- Breite des Adressvektors ergibt die maximale Zahl der verfügbaren Speicherworte: $2^{<Breite>}$
- Festplattenspeicher (persistent) gehören nicht zum Speicherwerk, sondern zum E/A-Werk
  - Daten werden von der Festplatte zunächst in den Arbeitsspeicher geladen, bevor sie verarbeitet werden können
  - Spezialfall: Virtueller Arbeitsspeicher auf externer Festplatte (extrem langsam aber kostengünstig).



Arbeitsspeicher: Segmente

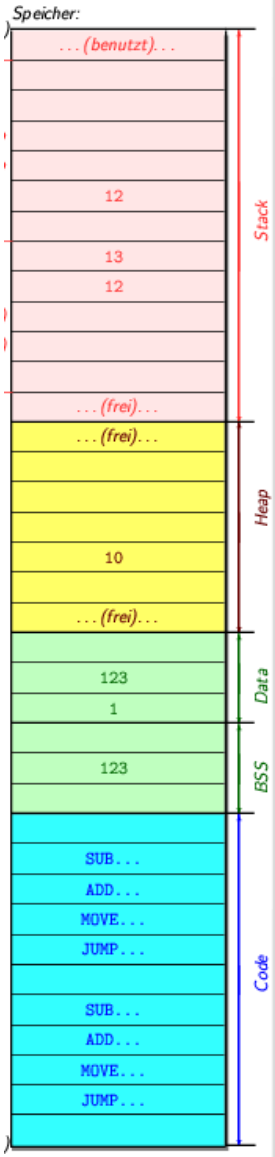
- *Codesegment* (oft auch *Textsegment* genannt) enthält nach dem Laden des Programms den gesamten auszuführenden Programmcode (in Maschinensprache)
  - In diesen Bereich zeigt der program counter.
- Das Datensegment (*data segment*) enthält alle Daten, die während des gesamten Programmlaufs genau einmal existieren (alle globalen Variablen sowie die lokalen statischen), soweit sie bei Programmstart initialisiert werden müssen.
- Das *block storage segment* (BSS) enthält analog alle globalen und lokalen statischen Variablen, die nicht initialisiert werden müssen.
  - *Remark: Die Unterscheidung zwischen initialisierten und nicht initialisierten Variablen wird üblicherweise getroffen, weil initialisierte Daten durch ihre Initialisierungswerte Platz in der Programmdatei belegen, den man sich für nicht initialisierte Variablen sparen kann (es genügt dem Betriebssystem, beim Laden die Gesamtgröße aller BSS-Daten zu kennen).*



http://www.wachtler.de/ck/9_Speichermodell_Prozesses.html

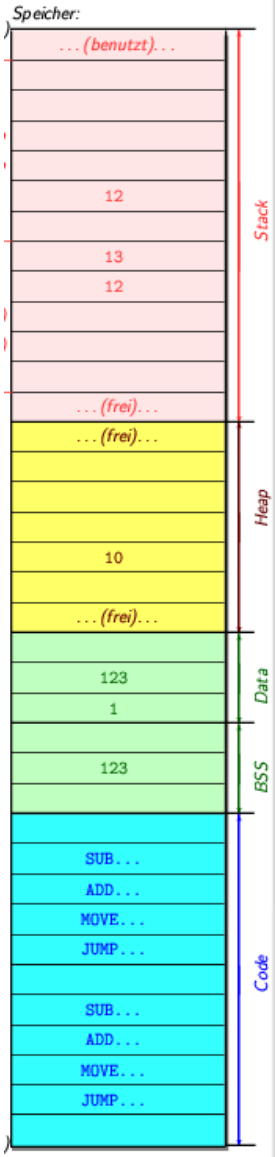
Arbeitsspeicher: Segmente

- Der Stack (stack segment) nimmt Daten auf, die dynamisch zur Laufzeit in einer geordneten Reihenfolge entstehen und genau in der umgekehrten Reihenfolge nicht mehr benötigt werden.
  - Solche Daten sind:
    - alle Argumente an aufgerufene Funktionen
    - deren automatische Variablen (alle lokalen Variablen, die nicht als „static“ deklariert sind)
    - Rücksprungadressen (die Stelle im Programm, an der nach dem Ende einer Funktion fortgefahren werden soll)
    - Registerinhalte, die für die Dauer eines Funktionsaufrufs in Sicherheit gebracht werden sollen
    - teilweise Platz für Rückgabewerte von Funktionen (in C typischerweise für die Rückgabe von nicht elementaren Datentypen, also Feldern oder Strukturen)
    - kurzfristig benötigter Platz für Zwischenergebnisse, beispielsweise bei arithmetischen Ausdrücken
  - Es wird bei Programmstart ein „ausreichend“ großer Bereich im Speicher reserviert (eben der Stack).
  - Der Stack besteht immer aus zwei Teilen: ein zusammenhängender Bereich, der bereits belegt ist (üblicherweise der obere Teil), sowie der gesamte Rest, der noch frei ist.



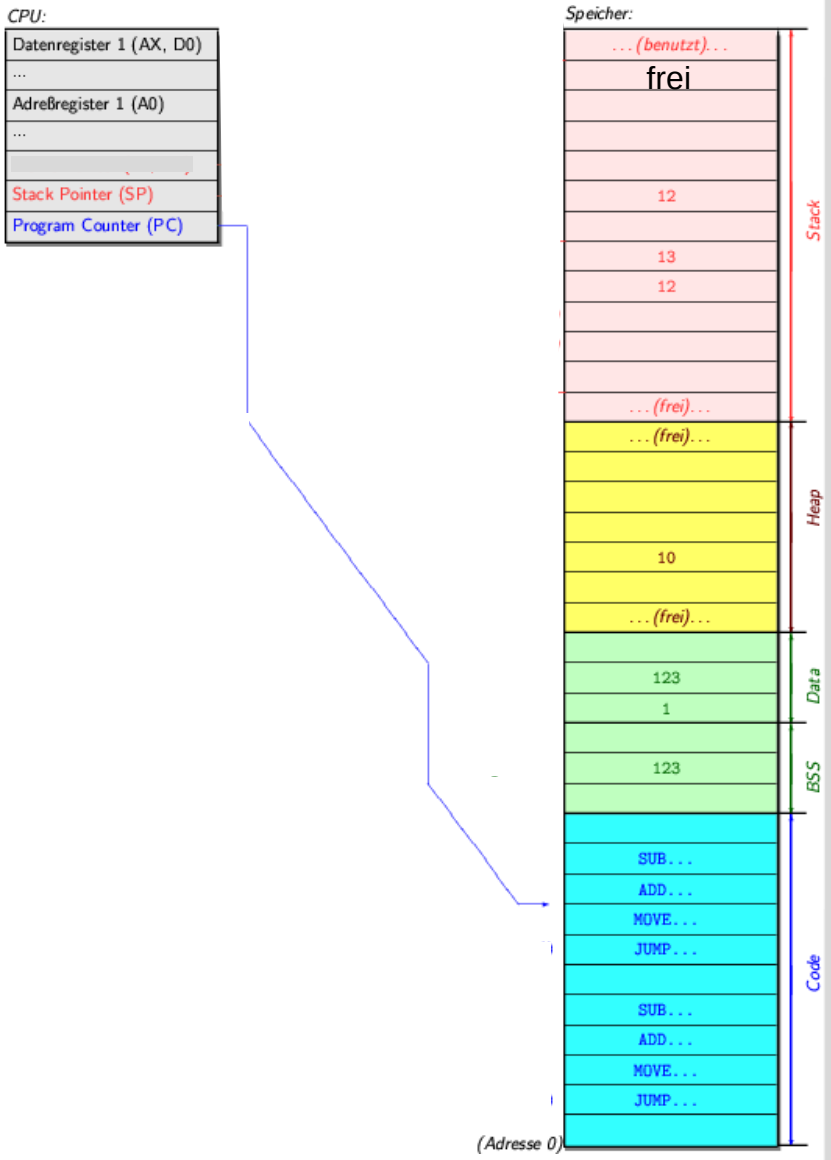
Arbeitsspeicher: Segmente

- Der gesamte, nicht von den genannten Segmenten belegte Speicher, steht zur freien Verfügung des Programms, und heißt Heap.
  - Dieser Heap wird in beliebiger Reihenfolge stückweise reserviert und wieder freigegeben, in C üblicherweise durch malloc() und free().



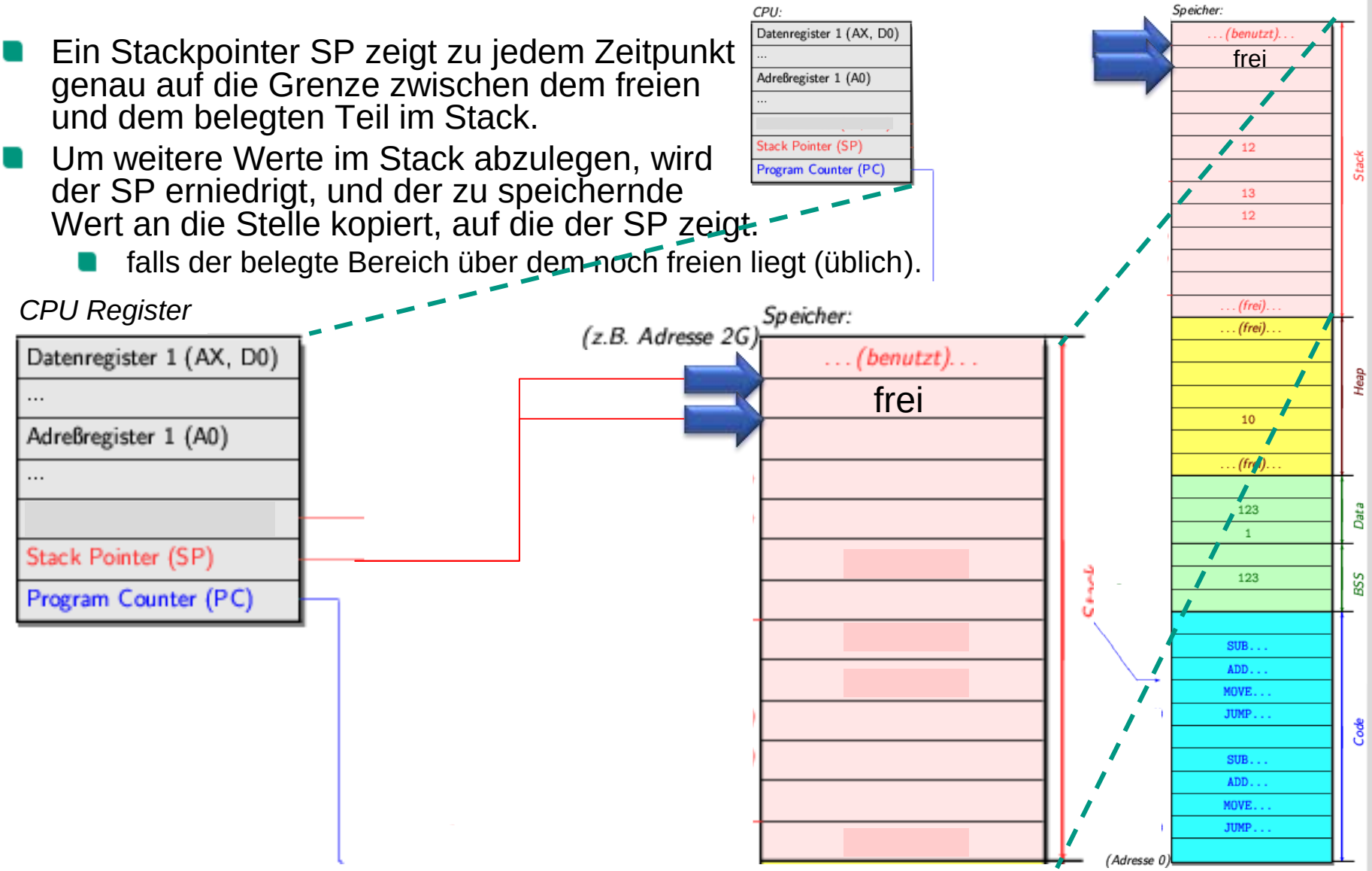
Arbeitsspeicher: Pointer

- Ein Befehlszähler (*Program Counter* / PC) zeigt zu jedem Zeitpunkt auf die aktuelle Programzeile im Hauptspeicher.



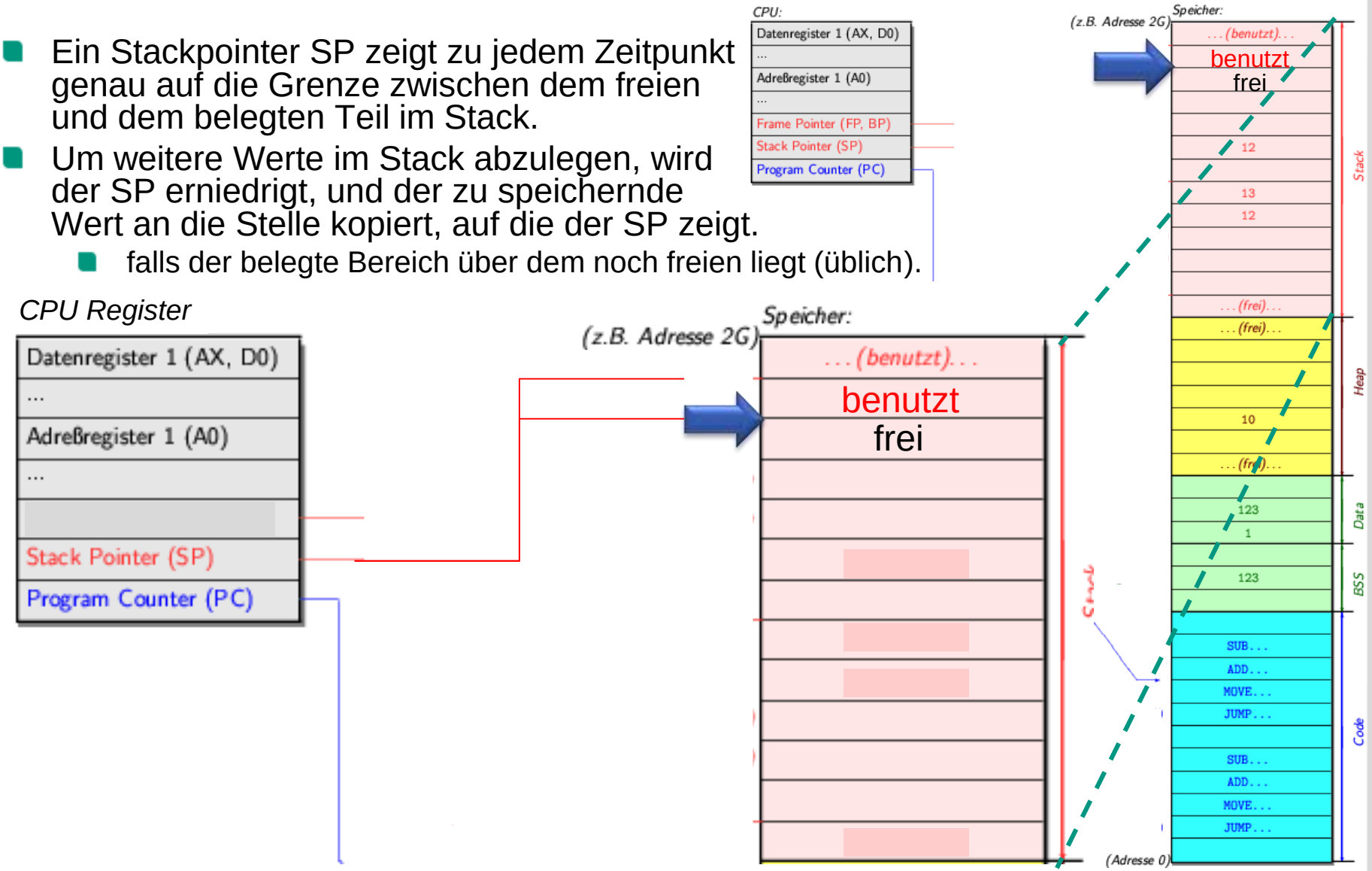
Arbeitsspeicher: Pointer

- Ein Stackpointer SP zeigt zu jedem Zeitpunkt genau auf die Grenze zwischen dem freien und dem belegten Teil im Stack.
- Um weitere Werte im Stack abzulegen, wird der SP erniedrigt, und der zu speichernde Wert an die Stelle kopiert, auf die der SP zeigt.
  - falls der belegte Bereich über dem noch freien liegt (üblich).



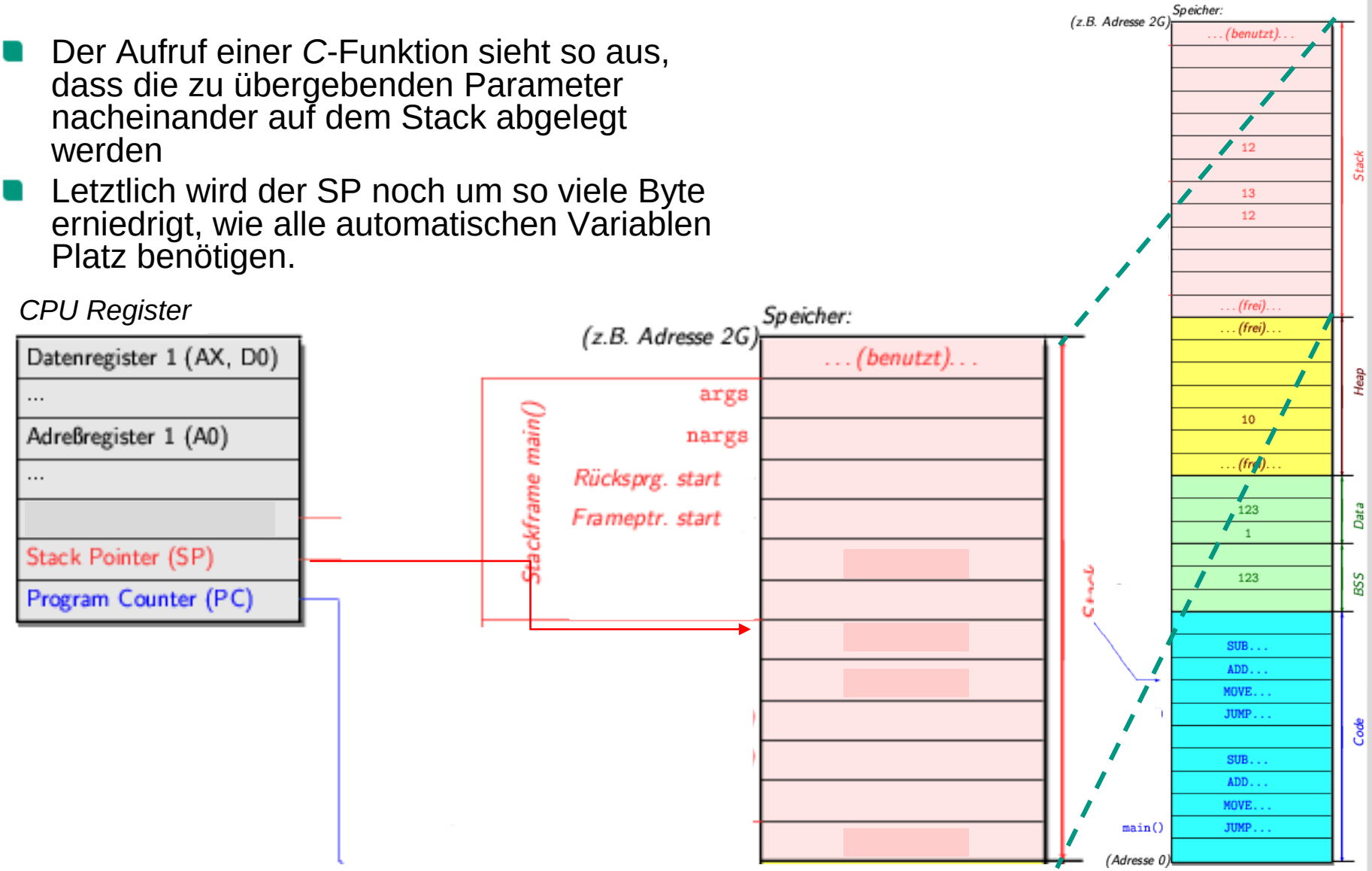
Arbeitsspeicher: Pointer

- Ein Stackpointer SP zeigt zu jedem Zeitpunkt genau auf die Grenze zwischen dem freien und dem belegten Teil im Stack.
- Um weitere Werte im Stack abzulegen, wird der SP erniedrigt, und der zu speichernde Wert an die Stelle kopiert, auf die der SP zeigt.
  - falls der belegte Bereich über dem noch freien liegt (üblich).



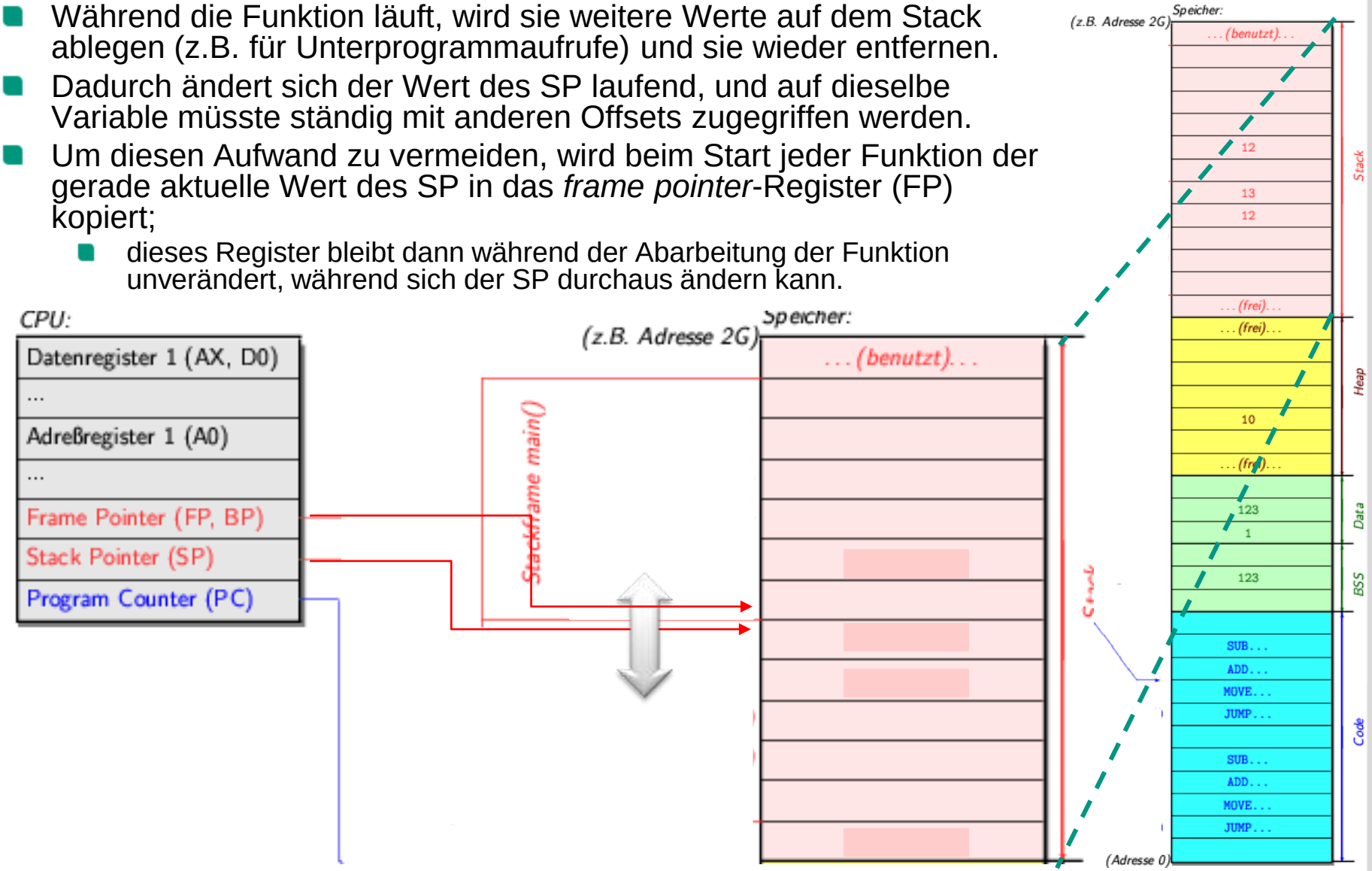
Arbeitsspeicher: Pointer

- Der Aufruf einer *C*-Funktion sieht so aus, dass die zu übergebenden Parameter nacheinander auf dem Stack abgelegt werden
- Letztlich wird der SP noch um so viele Byte erniedrigt, wie alle automatischen Variablen Platz benötigen.



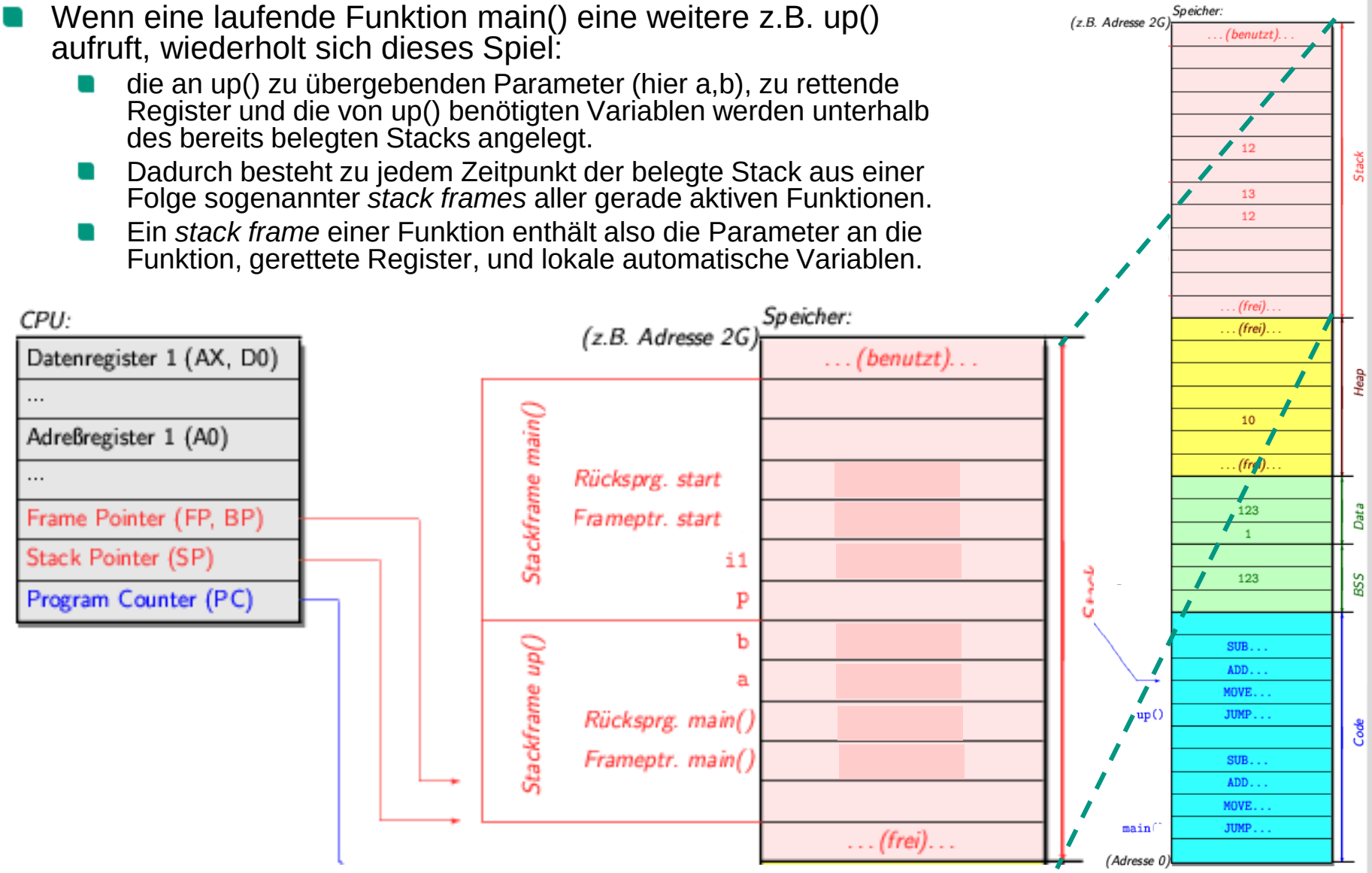
Arbeitsspeicher: Pointer

- Während die Funktion läuft, wird sie weitere Werte auf dem Stack ablegen (z.B. für Unterprogrammaufrufe) und sie wieder entfernen.
- Dadurch ändert sich der Wert des SP laufend, und auf dieselbe Variable müsste ständig mit anderen Offsets zugegriffen werden.
- Um diesen Aufwand zu vermeiden, wird beim Start jeder Funktion der gerade aktuelle Wert des SP in das *frame pointer*-Register (FP) kopiert;
  - dieses Register bleibt dann während der Abarbeitung der Funktion unverändert, während sich der SP durchaus ändern kann.



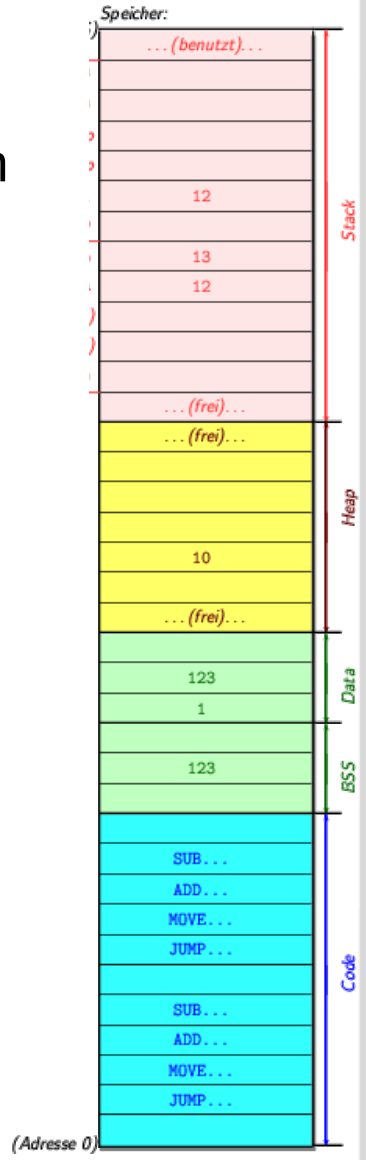
Arbeitsspeicher: Pointer

- Wenn eine laufende Funktion main() eine weitere z.B. up() aufruft, wiederholt sich dieses Spiel:
  - die an up() zu übergebenden Parameter (hier a,b), zu rettende Register und die von up() benötigten Variablen werden unterhalb des bereits belegten Stacks angelegt.
  - Dadurch besteht zu jedem Zeitpunkt der belegte Stack aus einer Folge sogenannter *stack frames* aller gerade aktiven Funktionen.
  - Ein *stack frame* einer Funktion enthält also die Parameter an die Funktion, gerettete Register, und lokale automatische Variablen.



Arbeitsspeicher: Pointer

- Wenn eine aufgerufene Funktion mit ihrem Programmcode beginnt, kann sie nun auf alle ihre automatischen Variablen ebenso wie auf ihre Parameter zugreifen, indem sie zum Wert des SP einen bestimmten Offset addiert (der dem Compiler bereits bekannt ist, weil er die Größen aller Datentypen und damit alle Positionen im stack frame kennt (*SP-relative Adressierung, s. ff.*).



Arbeitsspeicher: Pointer

Quelltext:

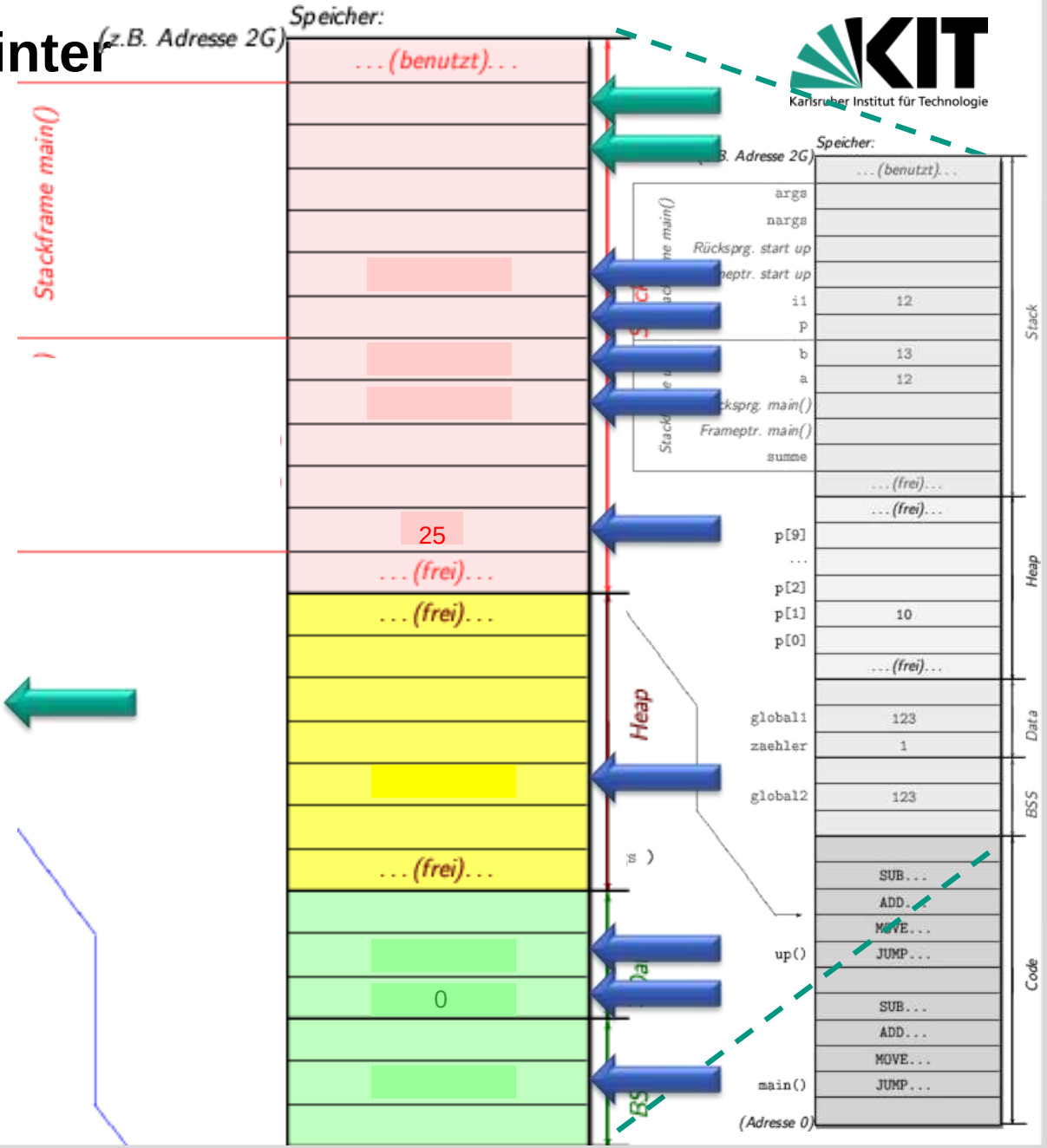
```
int global1 = 123;
int global2;

int up( int a, int b )
{
    int summe;
    static int zaehler = 0;

    summe = a + b;
    zaehler++;
    return summe;
}

int main( int nargs, char** args )
{
    int i1 = 12;
    int *p;

    global2 = global1;
    p = malloc( 10*sizeof(int) );
    p[1] = 10;
    up( i1, 13 );
    return 0;
}
// (C) 2002 Klaus Wachtler
```



Arbeitsspeicher: Pointer

Quelltext:

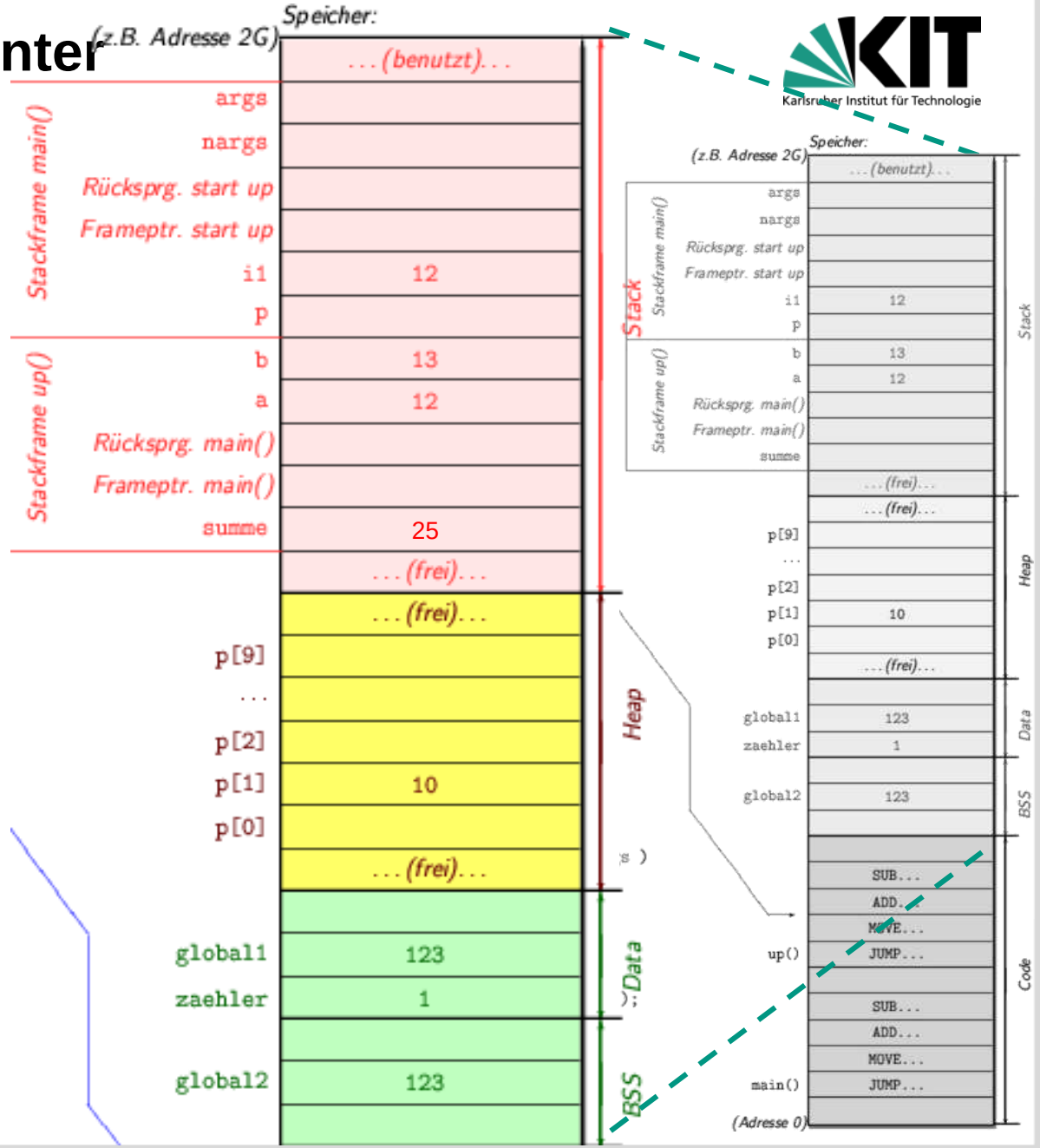
```
int global1 = 123;
int global2;

int up( int a, int b )
{
    int summe;
    static int zaehler = 0;

    summe = a + b;
    zaehler++;
    return summe;
}

int main( int nargs, char** args )
{
    int i1 = 12;
    int *p;

    global1 = global2;
    p = malloc( 10*sizeof(int) );
    p[1] = 10;
    up( i1,  13 );
    return 0;
}
// (C) 2002 Klaus Wachtler
```

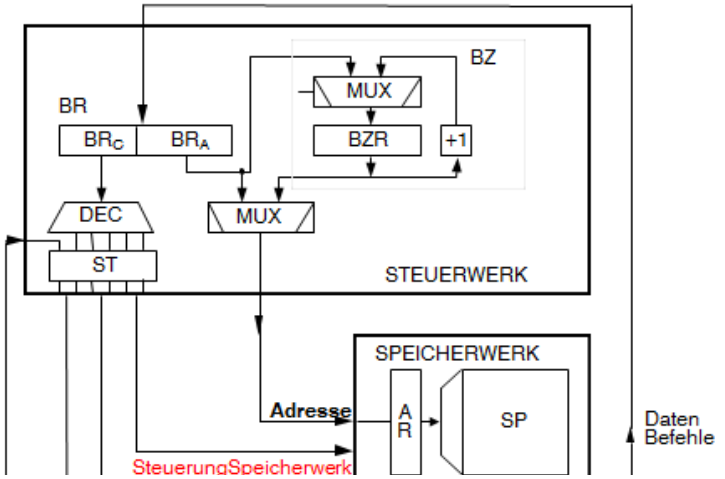


Adressierungsarten

Aufbau eines von Neumann Rechners

Detaillierte Sichtweise

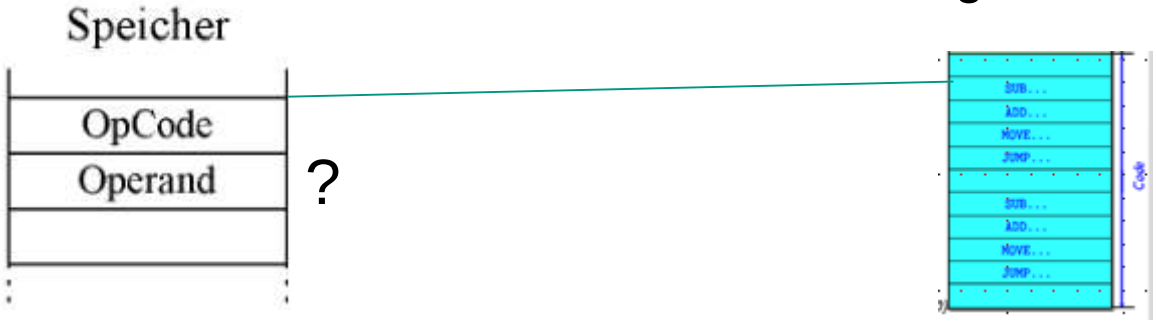
- Steuerwerk
  - BR: Befehlsregister
    - BRC: Codeteil (Opcode)
    - BRA: Adressteil
  - ST: Register für Steuerbits
  - BZ: Befehlszähler
    - BZR: Befehlszählerregister



- *Codesegment* (oft auch *Textsegment* genannt) enthält nach dem Laden des Programms den gesamten auszuführenden Programmcode (in Maschinensprache)
  - In diesen Bereich zeigt der program counter.

Adressierungsarten

- Ein Befehl enthält einen <Opcode> und einen oder mehrere <Operanden>
- Ein Mikroprozessor bietet eine Reihe von Möglichkeiten, die <Operanden> für eine Operation zu bestimmen.
- Diese Möglichkeiten bezeichnet man als Adressierungsarten.



s. zuvor

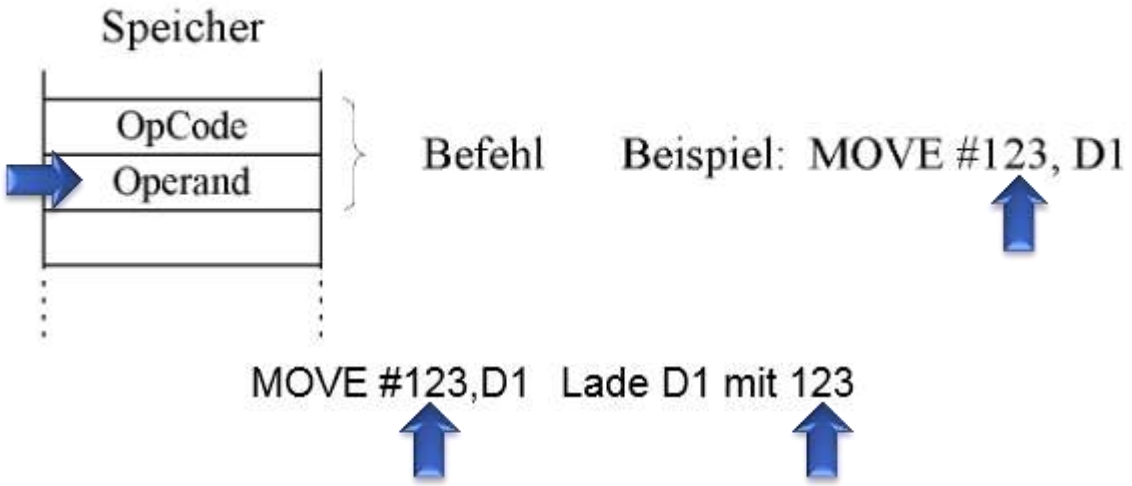
Rem. Im folgenden am Beispiel Intel Befehlssatz

Codesegment (oft auch *Textsegment* genannt) enthält nach dem Laden des Programms den gesamten auszuführenden Programmcode (in Maschinensprache)

- In diesen Bereich zeigt der program counter.

Unmittelbare Adressierung (immediate)

- Bei dieser Adressierungsart ist der Operand eine Konstante, die direkt hinter dem OpCode im Speicher steht.



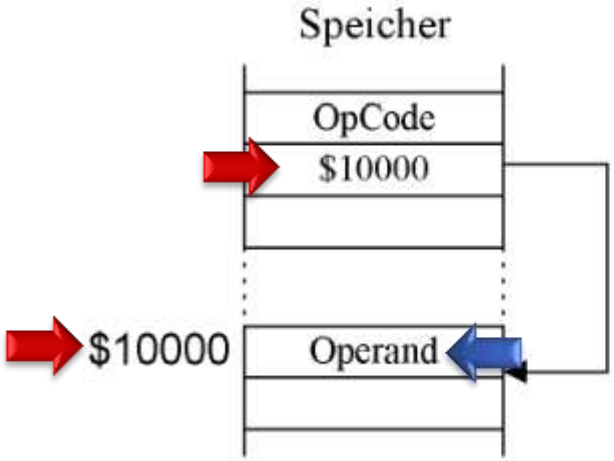
Bemerkung (auch im weiteren):

- *A1 → Adressregister A1*
- *D1 → Datenregister D1*

Quelle ff: FH Augsburg
Fachbereich Elektrotechnik Mikrocomputertechnik
Prof. Dr. Bayer

Absolute Adressierung (absolute)

- Die effektive Adresse des Operanden befindet sich als absolute Adresse direkt im Anschluss an den Opcode im Speicher.
- Bemerkung: *In der Assemblersyntax (68000) steht die Adresse ohne spezielle Kennzeichnung als Zahlenwert direkt hinter dem Opcode.*

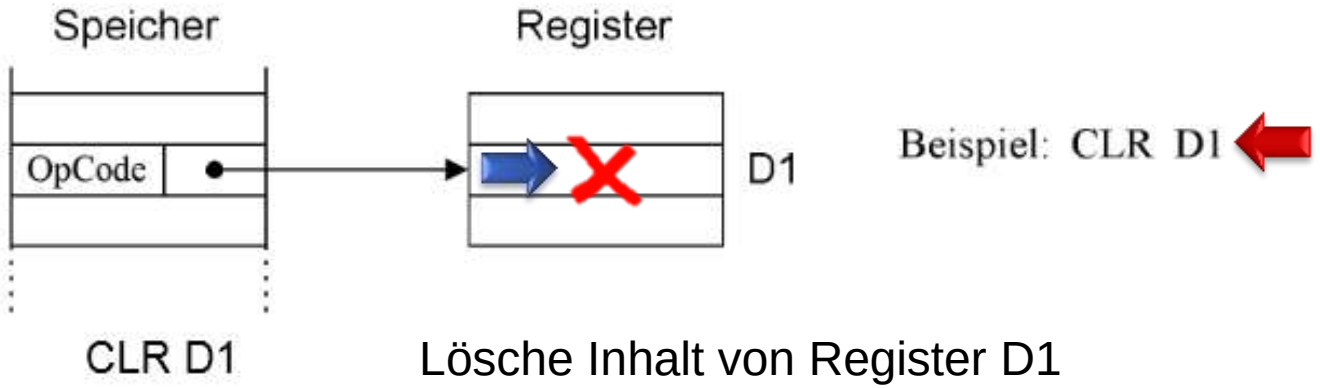


Beispiel: MOVE $10000, D1

Lade D1 mit Inhalt von $10000

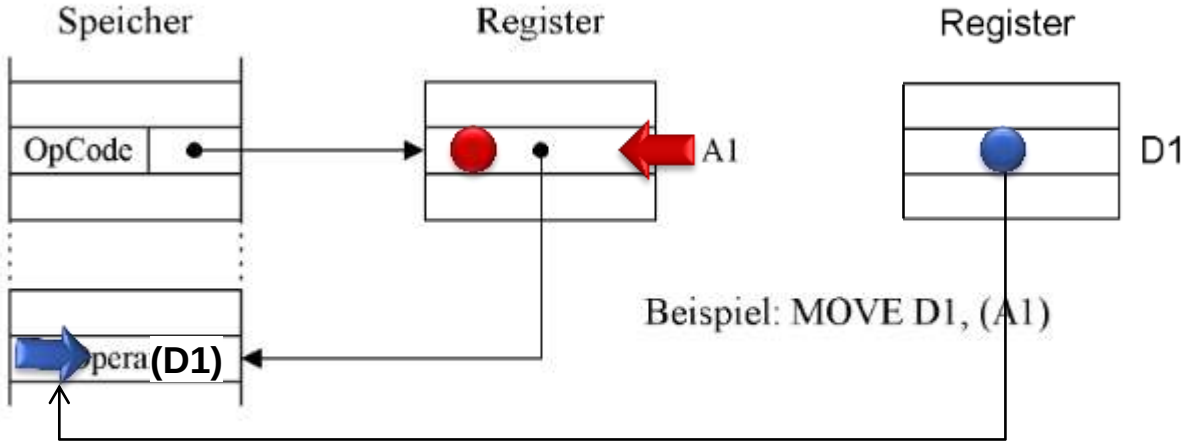
Registeradressierung (register direct)

- Der Operand steht in einem der Prozessorregister.
- Die Adresse steht als kurze Registeradresse im OpCode.
- Als Register können universelle Register wie z.B. Datenregister oder Adressregister, aber auch spezielle Register wie Stackpointer, Statusregister usw. angesprochen werden.



Registerindirekte Adressierung (register indirect)

- Die effektive Adresse steht in einem Register, üblicherweise in einem Adressregister; der Operand steht im Speicher.
- Diese Adressierungsart hat gegenüber der absoluten Adressierung den Vorteil, dass beim Holen des Befehls die Operanden-Adresse nicht gelesen werden muss (PC konstant)
- Sie wird eingesetzt, wenn innerhalb eines Programmteils häufig auf dieselbe Operanden-Adresse zugegriffen wird.

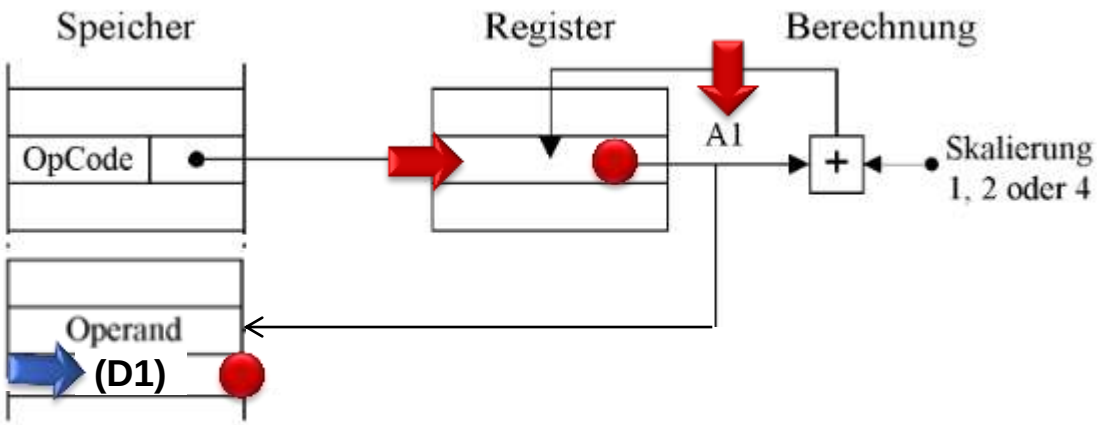


MOVE D1,(A1) Lade die Speicherstelle mit der Adresse in A1 mit dem Registerinhalt von D1

Registerindirekte Adressierung mit Postinkrement (1)

- Bei der Adressierung wird der Inhalt des verwendeten Adressregisters verändert.
- Beim Postinkrement wird nach der Verwendung der Adresse der Inhalt des Registers inkrementiert, wobei das Inkrement abhängig von der Datenbreite des adressierten Operanden die Werte 1, 2 oder 4 annehmen kann.
  - Diese Anpassung der Schrittweite nennt man Skalierung, da das Inkrement mit dem skalaren Faktor 1, 2 oder 4 multipliziert wird.
    - Wird zum Beispiel mit dem Befehl MOVE D1, (A1)+ ein Byte vom Datenregister D1 in den Speicher kopiert, so wird A1 anschließend um den Wert 1 erhöht.
    - Wird mit MOVE D1, (A1)+ ein Langwort, also 4 Byte kopiert, so wird A1 entsprechend um den Wert 4 inkrementiert.
  - Nach der Befehlsausführung zeigt daher das Adressregister immer auf das nächste Datenwort im Speicher.
  - Diese Adressierungsart eignet sich für die Bearbeitung von Datenfeldern in einer Schleife.

Registerindirekte Adressierung mit Postinkrement (2)

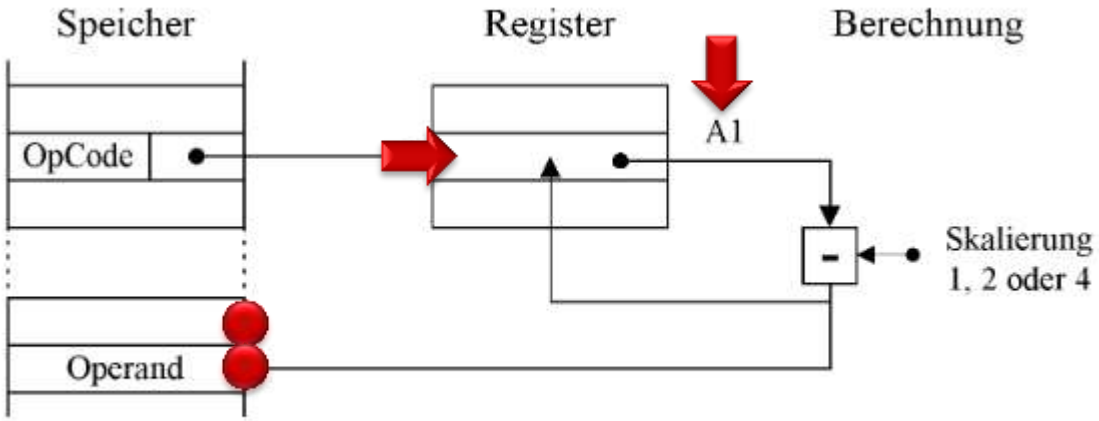


Beispiel: MOVE D1, (A1)+

Lade die Speicherstelle mit der Adresse in A1 mit dem Registerinhalt von D1 (s. zuvor). Erhöhe den PC im Anschluss um 1.

Registerindirekte Adressierung mit Prädekrement

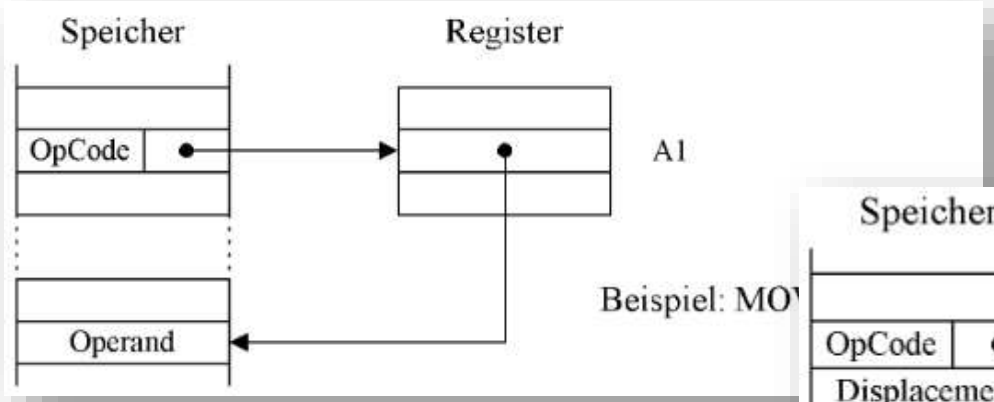
- Das Gegenstück zum Postinkrement ist die Adressierung mit Prädekrement.
- Vor dem Speicherzugriff wird der Inhalt des Adressregisters dekrementiert.
- Diese Adressierungsart wird durch ein Minuszeichen vor dem Adressregister symbolisiert.
- Wird als Adressregister der Stackpointer verwendet, so ersetzen diese beiden Adressierungsarten die Befehle PUSH und POP zur Stackverwaltung



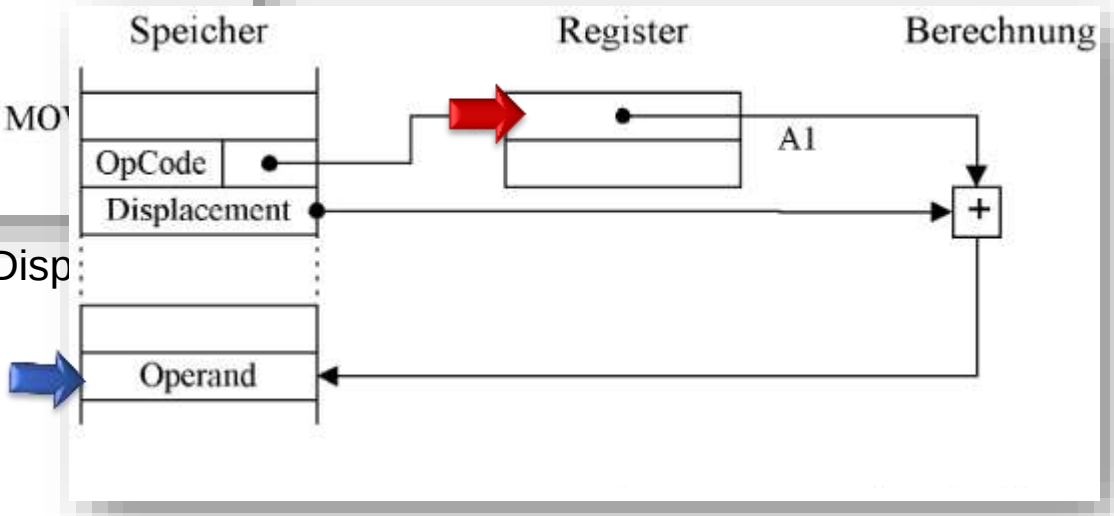
Beispiel: MOVE D1, -(A1)

(Registerindirekte) Adressierung mit Displacement

- Bei der Adressierung mit Displacement wird die effektive Adresse aus dem Inhalt eines Adressregisters und einer konstanten Adreßdistanz (Displacement) berechnet.
- Das Displacement ist vorzeichenbehaftet (2er-Komplement) und erlaubt so eine positive und negative Adressdistanz zur Basisadresse im Adressregister.

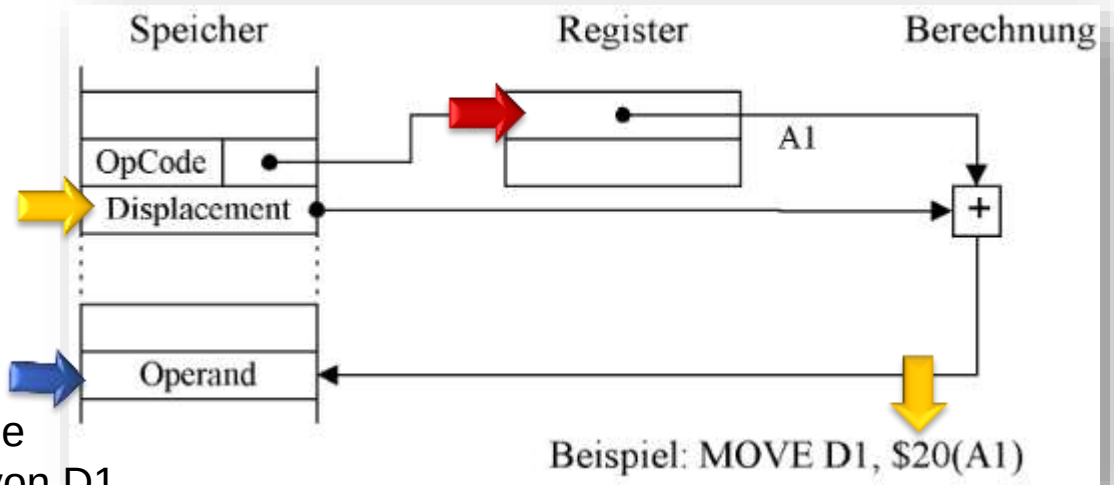


Registerindirekte Adressierung ohne Disp
(s. Folie zuvor)



(Registerindirekte) Adressierung mit Displacement

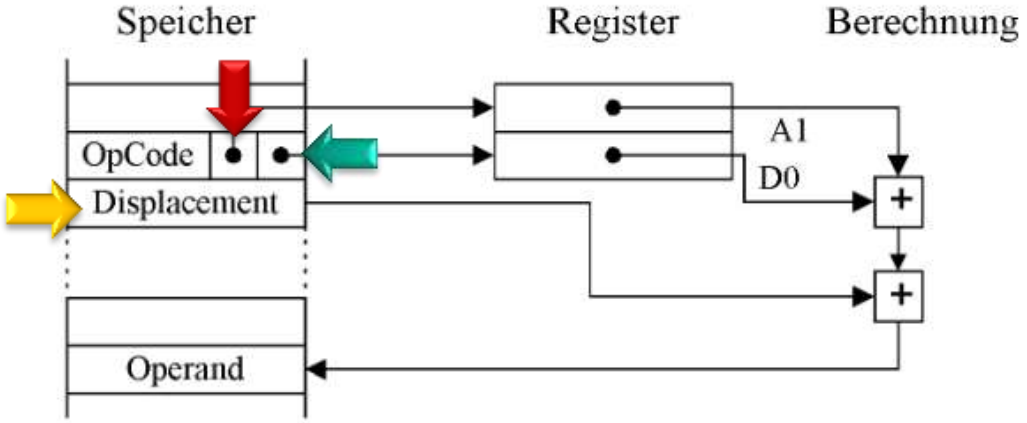
- Diese Methode wird beim Zugriff auf Daten verwendet, die in einer festen Struktur im Speicher vorliegen, wie z.B. Directory-Einträge im Dateiverwaltungssystem oder Register eines Peripheriebausteins.
- Durch Ändern der Basisadresse im Adressregisters wird so auf das gleiche Element im nächsten Datensatz zugegriffen bzw. es kann eine variable IO-Basisadresse berücksichtigt werden.
- In der Assemblersyntax wird das Displacement durch eine Zahl vor dem Adressregister angegeben.



Lade die Speicherstelle mit der Adresse in (A1 + Disp) mit dem Registerinhalt von D1.

Indizierte Adressierung mit Displacement

- Wird neben einer Konstanten noch eine variable Adressdistanz benötigt, die erst zur Laufzeit eines Programms feststeht, so wird die indizierte Adressierung verwendet.
- Die effektive Adresse wird hier aus der Basisadresse in einem Adressregister, einem konstanten und einem variablen Adressversatz aus einem weiteren Register berechnet



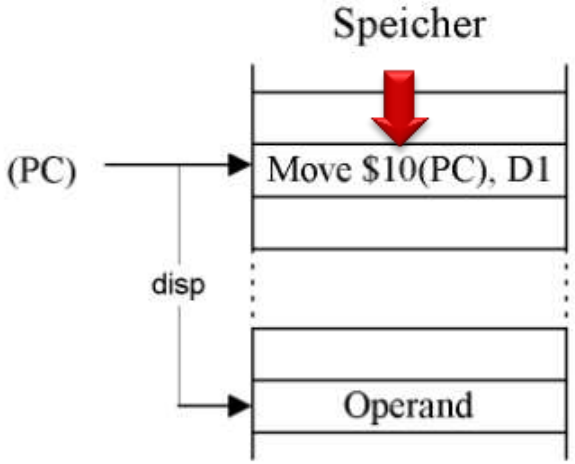
Beispiel: MOVE D1, $5(A1,D0)

Lade die Speicherstelle mit der Adresse
in (A1 + D0+ Disp) mit dem Registerinhalt von D1.

Bemerkung: Motorola 8900 erster Prozessor mit Indizierter Adressierung (1980)

PC-relative Adressierung

- Idee: Verwendung des Befehlszählers (PC program counter) als Adressregisters.
- Die effektive Adresse *des Operanden* wird hier relativ zum aktuellen Befehlszähler mit einem ➡ Adressabstand zum Befehlszähler gebildet.
- Die befehlszählerrelative Adressierung ermöglicht es, ein übersetztes Programm mitsamt Daten und Konstanten im Speicher zu verschieben.
- Der Befehlszähler darf dabei nicht verändert werden, da sonst das Programm nicht mehr korrekt ablaufen würde.



Lade die Speicherstelle mit der Adresse in D1 mit Inhalt von PC + Disp


Fragen?

IT Inhalt

0. Einleitung und Motivation

- Organisatorisches
- Begriffe
- Typische Anwendungen

1. Rechnerarchitekturen

- Einführung
- Allgemeiner Aufbau der internen Hardware Architektur
- Instruction Set Architecture (ISA) am Beispiel ARM Cortex-M4
- Klassifikation von Rechnerarchitekturen
- Performanzsteigerung
- ausgewählte Beispiele

Operationen mit Daten

- Datentransfer
- Arithmetisch
- Floating Point
- Logisch
- Bedingte und unbedingte Verzweigung
- Shift, Rotate
- Bit Manipulation
- Multimedia
- Systemsteuerung
- Co-Prozessorbefehle

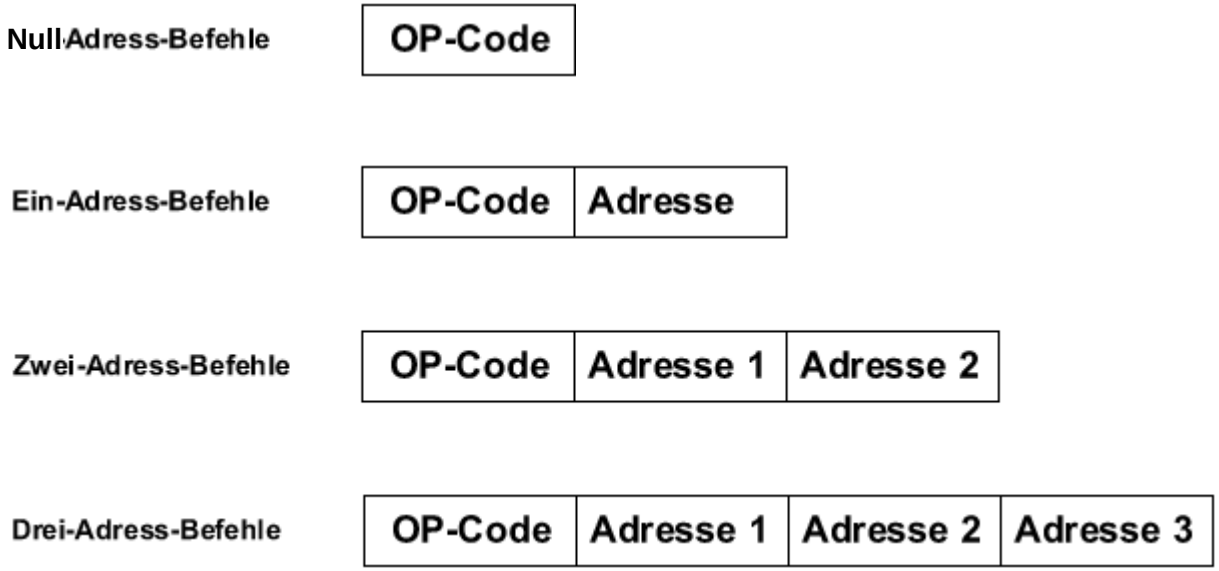
Was mache ich mit den Daten und wie?

Opcode/Daten

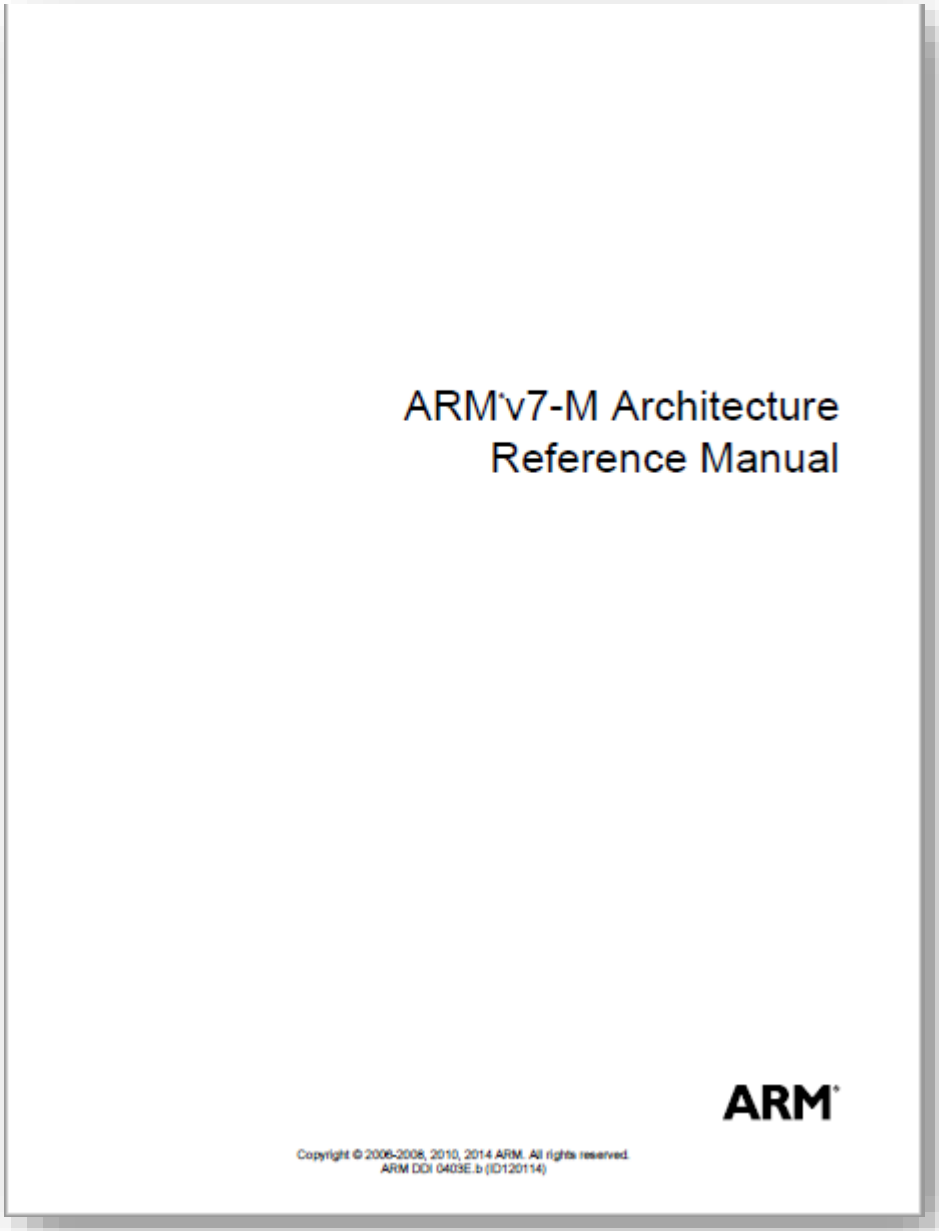
Wo finde ich die Daten?

Befehlsformate

- 0-Adress Befehlsformat: opcode
- 1-Adress Befehlsformat: opcode | Src
- 2-Adress Befehlsformat: opcode | Dest/Src1 | Src2
- 3-Adress Befehlsformat: opcode | Dest | Src1 | Scr2



Reference Manual



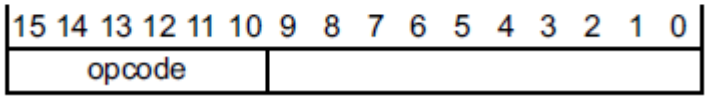
Befehlsformate ARM Cortex-M4

Operation	Description	Assembler	Cycles
Move	Register	`MOV Rd, <op2>`	1
	16-bit immediate	`MOVW Rd, #<imm>`	1
	Immediate into top	`MOVT Rd, #<imm>`	1
	To PC	`MOV PC, Rm`	1 + P
Add	Add	`ADD Rd, Rn, <op2>`	1
	Add to PC	`ADD PC, PC, Rm`	1 + P
	Add with carry	`ADC Rd, Rn, <op2>`	1
	Form address	`ADR Rd, <label>`	1
Hint	Send event	`SEV`	1
	Wait for event	`WFE`	1 + W
	Wait for interrupt	`WFI`	1 + W
	No operation	`NOP`	1
Branch	Conditional	`B<cc> <label>`	1 or 1 + P[c]
	Unconditional	`B <label>`	1 + P
	With link	`BL <label>`	1 + P

Befehlsformate ARM Cortex-M4

16-bit Thumb instruction encoding

The encoding of 16-bit Thumb instructions is:



- Kategorisierung anhand der Bits [15:10]
- Unterschiedliche Opcode-Längen
- Mehrere Codierungen für unterschiedliche Variante Befehls

Add, Move

NOP

Branch

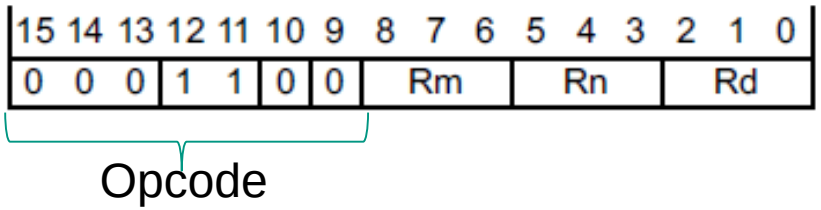
opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-128
010000	*Data processing* on page A5-129
010001	*Special data instructions and branch and exchange* on page A5-130
01001x	Load from Literal Pool, see *LDR (literal)* on page A7-245
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-131
10100x	Generate PC-relative address, see *ADR* on page A7-196
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A7-192
1011xx	*Miscellaneous 16-bit instructions* on page A5-132
11000x	Store multiple registers, see *STM, STMIA, STMEA* on page A7-378
11001x	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A7-239
1101xx	*Conditional branch, and Supervisor Call* on page A5-134
11100x	Unconditional Branch, see *B* on page A7-203

3-Adress Befehlsformat (ARM Cortex-M4)

- opcode | Dest | Src1 | Src2
- Beispiel: ADD(register)
  - fügt einen Registerwert zu einem anderen Registerwert hinzu und schreibt das Ergebnis in das Zielregister
  - Syntax: ADD Rd, Rn, Rm
  - Rd: Zielregister
  - Rn: Quellregister, erster Operand
  - Rm: Quellregister, zweiter Operand
  - <c>: optionaler Bedingungscode

Bedingungscode ermöglicht es, eine Bedingung anhand Flags im Application Program Status Register (APSR) zu testen. Wenn Bedingungstest einer bedingten Anweisung fehlschlägt, wird die Anweisung:
- Nicht ausgeführt
- schreibt keinen Wert in sein Zielregister
- hat keinen Einfluss auf die Flags
- erzeugt keine Ausnahme.

```
ADD<c> <Rd>,<Rn>,<Rm>
```

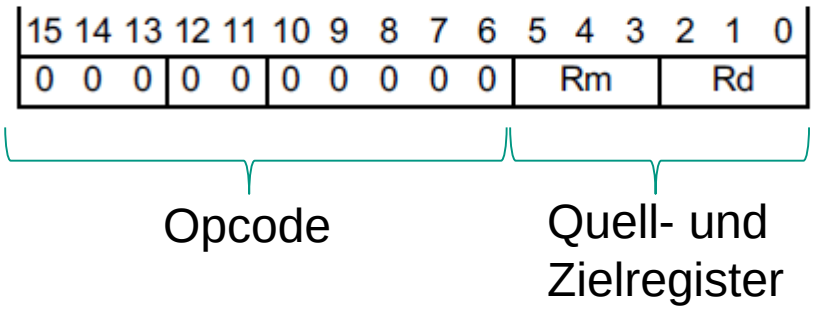


2-Adress Befehlsformat (ARM Cortex-M4)

- opcode | Dst1/Src1 | Src2

- Beispiel: MOV(register) - Kopieren
  - Kopiert einen Wert aus einem Register in das Zielregister.
  - Syntax: MOVS <Rd>,<Rm>
  - <Rd>: Zielregister
  - <Rm>: Quellregister

MOVS <Rd>,<Rm>



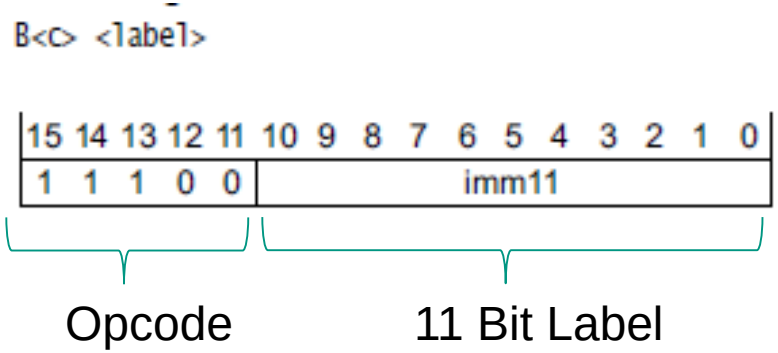
1-Adress Befehlsformat (ARM Cortex-M4)

- opcode | Src1

- Beispiel: B - Branch
  - Verzweigungs-Instruktion
  - Syntax: B<c> <label>
  - <c>: optionaler Bedingungscode
  - <label>: Name des Verzweigungs-Ziels

Beispiel:
B loopA ; Verzweigung zu loopA

Ein Label ist eine Sprungmarke, die ein bestimmtes Ziel im Assembler-Code identifiziert.

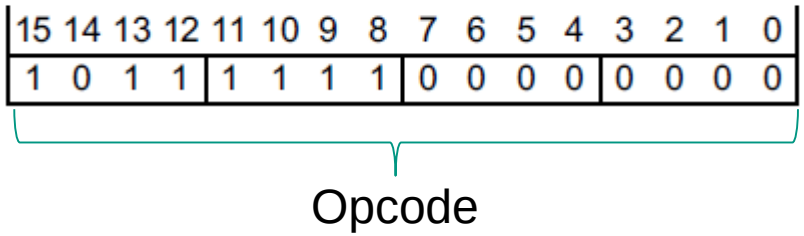


0-Adress Befehlsformat (ARM Cortex-M4)

- Opcode

- 0-Adress Befehlsformat: nur Befehlscode
- Beispiel: NOP – No Operation
  - Wird z.B. verwendet:
    - um Wartezyklen beim Hardware-Zugriff zu realisieren.
    - NOP zum Auffüllen, z.B. um den folgenden Befehl auf einer 64-Bit-Grenze zu platzieren
  - Operation:
    - NOP tut nichts
    - Prozessor kann Befehl aus Pipeline entfernen, bevor er die Ausführungsphase erreicht

NOP<c>




Fragen?

Backup

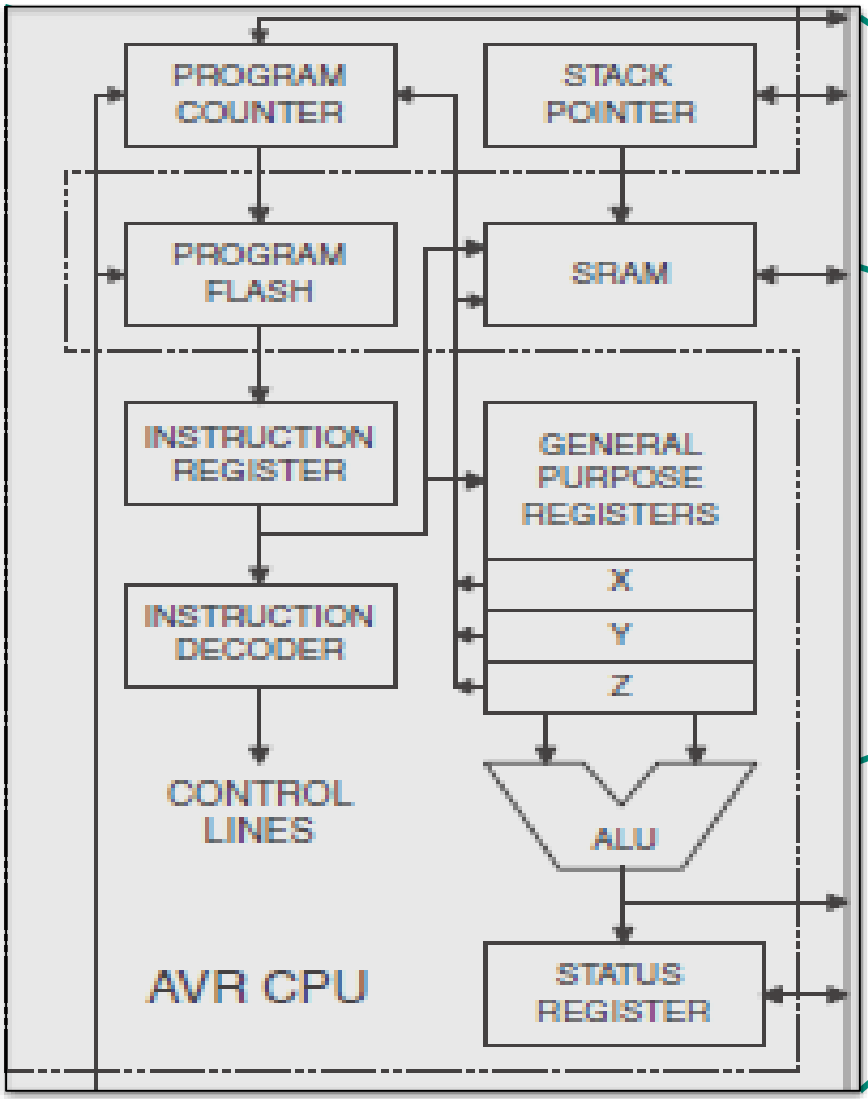
Datenrepräsentation Datenformate

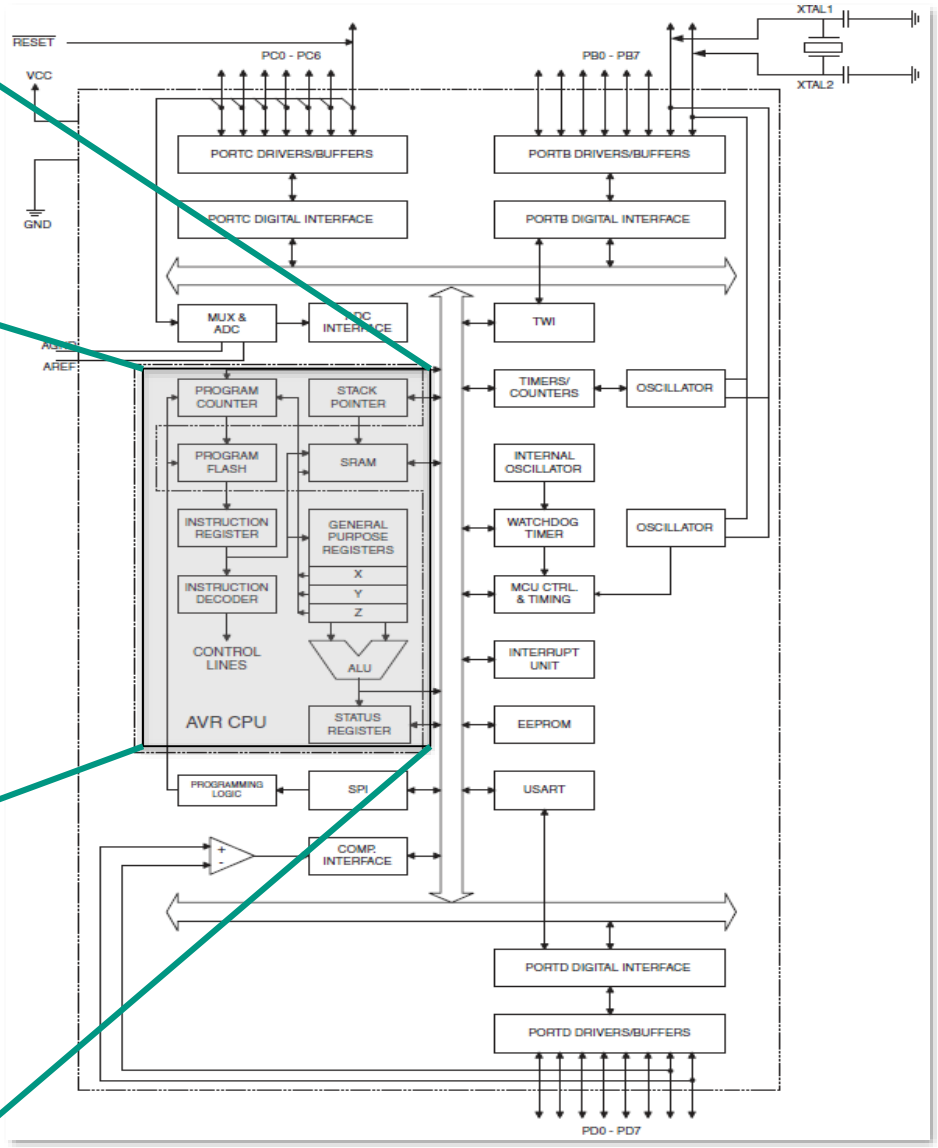
- ISA unterstützt verschiedene Datenformate
  - Integer
  - Gepackte und ungepackte BCD (Binary Coded Decimal) Zahlen
  - ASCII Character
  - Fließkommazahlen
- Es wurden für MMX (Multimedia Extensions) 4 neue Datenformate geschaffen:
  - PackedByte
  - PackedWord
  - PackedDoubleWord
  - QuadWord

64bit Breite

- Damit ist es möglich, bis zu 64 Bit große Integer-Datenpakete auf einmal zu bearbeiten.
- *Bemerkung (s. zuvor): Es gibt zwei Arten Byte-Adressen in einem Wort zu ordnen:*
  - *big-endian: most significant byte first*
  - *little-endian: least significant byte first*
- *Bemerkung: Im Prinzip ist ein QuadWord nur ein 64-Bit-Feld, das man auch DoubleLongInt hätte nennen können*

ATMEL AVR8 Microcontroller





Befehlsformate AVR8

Arithmetic	Bit & Others	Transfer	Jump	Branch	Call
ADD Rd, Rr	BSET s	MOV Rd, Rr	RJMP δP12	CPSE Rd, Rr	RCALL δP12
ADC Rd, Rr	BCLR s	MOVW Rd+1:Rd, Rr+1:Rr	IJMP		ICALL
ADIW Rd+1:Rd, K6	SBI IO5, b		EIJMP	SBRC Rr, b	EICALL
	CBI IO5, b	IN Rd, IO6	JMP P22	SBRS Rr, b	CALL P22
SUB Rd, Rr	BST Rd, b	OUT IO6, Rr			
SUBI Rd, K8	BLD Rd, b			SBIC IO5, b	RET
SBC Rd, Rr		PUSH Rr		SBIS IO5, b	RETI
SBCI Rd, K8	NOP	POP Rr			
SBIW Rd+1:Rd, K6	BREAK			BRBC s, δP7	
	SLEEP	LDI Rd, K8		BRBS s, δP7	
INC Rd	WDR	LDS Rd, D16			
DEC Rd					
		LD Rd, X			
AND Rd, Rr		LD Rd, -X			
ANDI Rd, K8		LD Rd, X+			
OR Rd, Rr					
ORI Rd, K8		LDD Rd, Y+δD6			
EOR Rd, Rr		LD Rd, -Y			
		LD Rd, Y+			
COM Rd		LDD Rd, Z+δD6			
NEG Rd		LD Rd, -Z			
CP Rd, Rr		LD Rd, Z+			
CPC Rd, Rr					
CPI Rd, K8		STS D16, Rr			
SWAP Rd					
		ST X, Rr			
LSR Rd		ST -X, Rr			
ROR Rd		ST X+, Rr			
ASR Rd					
		STD Y+δD6, Rr			
MUL Rd, Rr		ST -Y, Rr			
MULS Rd, Rr		ST Y+, Rr			
MULSU Rd, Rr					
FMUL Rd, Rr		STD Z+δD6, Rr			
FMULS Rd, Rr		ST -Z, Rr			
FMULSU Rd, Rr		ST Z+, Rr			
		LPM			
		LPM Rd, Z			
		LPM Rd, Z+			

Befehlsformate AVR8

Arithmetic	Bit & Others	Transfer	Jump	Branch	Call
ADD Rd, Rr	BSET s	MOV Rd, Rr	RJMP δP12	CPSE Rd, Rr	RCALL δP12
ADC Rd, Rr	BCLR s	MOVW Rd+1:Rd, Rr+1:Rr	IJMP		ICALL
ADIW Rd+1:Rd, K6	SBI IO5, b		EIJMP	SBRC Rr, b	EICALL
	CBI IO5, b	IN Rd, IO6	JMP P22	SBRS Rr, b	CALL P22
SUB Rd, Rr	BST Rd, b	OUT IO6, Rr			
SUBI Rd, K8	BLD Rd, b			SBIC IO5, b	RET
SBC Rd, Rr		PUSH Rr		SBIS IO5, b	RETI
SBCI Rd, K8	NOP	POP Rr			
SBIW Rd+1:Rd, K6	BREAK			BRBC s, δP7	
	SLEEP	LDI Rd, K8		BRBS s, δP7	
	WDR	LDS Rd, D16			
		LD Rd, X			
		LD Rd, -X			
		LD Rd, X+			
		LDD Rd, Y+δD6			
		LD Rd, -Y			
		LD Rd, Y+			
		LDD Rd, Z+δD6			
		LD Rd, -Z			
		LD Rd, Z+			
		STS D16, Rr			
		ST X, Rr			
		ST -X, Rr			
		ST X+, Rr			
		STD Y+δD6, Rr			
		ST -Y, Rr			
		ST Y+, Rr			
		STD Z+δD6, Rr			
		ST -Z, Rr			
		ST Z+, Rr			
		LPM			
		LPM Rd, Z			
		LPM Rd, Z+			

Atmel AVR instruction set overview

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOP
0	0	0	0	0	0	0	1	D	D	D	D	R	R	R	R	MOVW Rd,Rr Move register pair
0	0	0	0	0	0	1	0	d	d	d	d	r	r	r	r	MULS Rd,Rr
0	0	0	0	0	0	1	1	0	d	d	d	0	r	r	r	MULSU Rd,Rr
0	0	0	0	0	0	1	1	0	d	d	d	1	r	r	r	FMUL Rd,Rr
0	0	0	0	0	0	1	1	1	d	d	d	u	r	r	r	FMULS(U) Rd,Rr
0	0	0	cy	0	1	r	d	d	d	d	d	r	r	r	r	CPC/CP Rd,Rr
0	0	0	cy	1	0	r	d	d	d	d	d	r	r	r	r	SBC/SUB Rd,Rr
0	0	0	cy	1	1	r	d	d	d	d	d	r	r	r	r	ADC/ADD Rd,Rr ROL/LSL Rd (ADC/ADD with Rd=Rr)
0	0	0	1	0	0	r	d	d	d	d	d	r	r	r	r	CPSE Rd,Rr
0	0	1	0	0	0	r	d	d	d	d	d	r	r	r	r	TST Rd,Rr
0	0	1	0	0	1	r	d	d	d	d	d	r	r	r	r	EOR Rd,Rr
0	0	1	0	1	0	r	d	d	d	d	d	r	r	r	r	OR Rd,Rr
0	0	1	0	1	1	r	d	d	d	d	d	r	r	r	r	MOV Rd,Rr
0	0	1	1	K	K	K	K	d	d	d	d	K	K	K	K	CPI Rd,K
0	1	0	cy	K	K	K	K	d	d	d	d	K	K	K	K	SBCI/SUBI Rd,K
0	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K	ORI Rd,K SBR Rd,K

Befehlsformate AVR8

Arithmetic	Bit & Others	Transfer	Jump	Branch	Call
ADD Rd, Rr	BSET s	MOV Rd, Rr	RJMP δP12	CPSE Rd, Rr	RCALL δP12
ADC Rd, Rr	BCLR s	MOVW Rd+1:Rd, Rr+1:Rr	IJMP		ICALL
ADIW Rd+1:Rd, K6	SBI IO5, b				
SUB Rd, Rr					
SUBI Rd, K8					
SBC Rd, Rr					
SBCI Rd, K8					
SBIW Rd+1:Rd, K6					

Instruction encoding

Bit assignments:

- rrrr = Source register
- rrr = Source register (R16-R31)
- rr = Source register (R16-R23)
- RRRR = Source register pair (R0:R1 ... R30:R31)
- ddddd = Destination register
- dddd = Destination register (R16-R31)
- ddd = Destination register (R16-R23)
- DDDD = Destination register pair (R0:R1 ... R30:R31)
- pp = Register pair, W, X, Y or Z
- y = Y/Z register pair bit (0=Z, 1=Y)
- u = FMUL(S(U)) signed with 0=signed or 1=unsigned
- s = Store/load bit (0=load, 1=store)
- c = Call/jump (0=jump, 1=call)
- cy = without carry (0=with carry 1=without carry)
- aaaaaa = I/O space address
- aaaaa = I/O space address (first 32 only)
- bbb = Bit number

Atmel AVR instruction set overview

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOP
0	0	0	0	0	0	0	1	DDDD				RRRR				MOVW Rd,Rr Move register pair
0	0	0	0	0	0	1	0	dddd				rrrr				MULS Rd,Rr
0	0	0	0	0	0	1	1	0	ddd			0	rrr			MULSU Rd,Rr
0	0	0	0	0	0	1	1	0	ddd			1	rrr			FMUL Rd,Rr
0	0	0	0	0	0	1	1	1	ddd			u	rrr			FMULS(U) Rd,Rr
0	0	0	cy	0	1	r		ddddd				rrrr				CPC/CP Rd,Rr
0	0	0	cy	1	0	r		ddddd				rrrr				SBC/SUB Rd,Rr
0	0	0	cy	1	1	r		ddddd				rrrr				ADC/ADD Rd,Rr ROL/LSL Rd (ADC/ADD with Rd=Rr)
0	0	0	1	0	0	r		ddddd				rrrr				CPSE Rd,Rr
0	0	1	0	0	0	r		ddddd				rrrr				TST Rd,Rr
0	0	1	0	0	1	r		ddddd				rrrr				EOR Rd,Rr
0	0	1	0	1	0	r		ddddd				rrrr				OR Rd,Rr
0	0	1	0	1	1	r		ddddd				rrrr				MOV Rd,Rr
0	0	1	1	KKKK				dddd				KKKK				CPI Rd,K
0	1	0	cy	KKKK				dddd				KKKK				SBCI/SUBI Rd,K
0	1	1	0	KKKK				dddd				KKKK				ORI Rd,K SBR Rd,K

LPM Rd, Z+

Befehlsformate AVR8

Arithmetic	Bit & Others	Transfer	Jump	Branch	Call
ADD Rd, Rr	BSET s	MOV Rd, Rr	RJMP δP12	CPSE Rd, Rr	RCALL δP12
ADC Rd, Rr	BCLR s	MOVW Rd+1:Rd, Rr+1:Rr	IJMP		ICALL
ADIW Rd+1:Rd, K6					
SUB Rd, Rr					
SUBI Rd, K8					
SBC Rd, Rr					
SBCI Rd, K8					
SBIW Rd+1:Rd, K6					

Instruction encoding

Bit assignments:

- rrrr = Source register
- rrr = Source register (R16-R31)
- rr = Source register (R16-R23)
- RRRR = Source register pair (R0:R1 ... R30:R31)
- ddddd = Destination register
- dddd = Destination register (R16-R31)
- ddd = Destination register (R16-R23)
- DDDD = Destination register pair (R0:R1 ... R30:R3
- pp = Register pair, W, X, Y or Z
- y = Y/Z register pair bit (0=Z, 1=Y)
- u = FMUL(S(U)) signed with 0=signed or 1=unsigne
- s = Store/load bit (0=load, 1=store)
- c = Call/jump (0=jump, 1=call)
- cy = With carry (0=without carry 1=with carry)
- e = Extend indirect jump/call address with EIND (0=
- q = Extend program memory address with RAMPZ (
- aaaaaa = I/O space address
- aaaaa = I/O space address (first 32 only)
- bbb = Bit number

Atmel AVR instruction set overview

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOP
0	0	0	0	0	0	0	1	DDDD				RRRR				MOVW Rd,Rr Move register pair
0	0	0	0	0	0	1	0	dddd				rrrr				MULS Rd,Rr
0	0	0	0	0	0	1	1	0	ddd			0	rrr			MULSU Rd,Rr
0	0	0	0	0	0	1	1	0	ddd			1	rrr			FMUL Rd,Rr
0	0	0	0	0	0	1	1	1	ddd			u	rrr			FMULS(U) Rd,Rr
0	0	0	cy	0	1	r		ddddd				rrrr				CPC/CP Rd,Rr
0	0	0	cy	1	0	r		ddddd				rrrr				SBC/SUB Rd,Rr
0	0	0	cy	1	1	r		ddddd				rrrr				ADC/ADD Rd,Rr ROL/LSL Rd (ADC/ADD with Rd=Rr)
0	0	0	1	0	0	r		ddddd				rrrr				CPSE Rd,Rr
0	0	1	0	0	0	r		ddddd				rrrr				TST Rd,Rr
0	0	1	0	0	1	r		ddddd				rrrr				EOR Rd,Rr
0	0	1	0	1	0	r		ddddd				rrrr				OR Rd,Rr
0	0	1	0	1	1	r		ddddd				rrrr				MOV Rd,Rr
0	0	1	1	KKKK				dddd				KKKK				CPI Rd,K
0	1	0	cy	KKKK				dddd				KKKK				SBCI/SUBI Rd,K
0	1	1	0	KKKK				dddd				KKKK				ORI Rd,K SBR Rd,K

Adierbefehl (AVR8)

ADD – ADDIERE REGISTER OHNE CARRY-FLAG

Syntax:	Funktion:
ADD *Rd, Rr*	*Rd* ← *Rd* + *Rr*

Beschreibung:
Der Inhalt des Registers **Rr** wird zum Inhalt des Registers **Rd** addiert. Das Ergebnis der Addition steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert.
*(zulässig für **Rd, Rr**: r0 bis r31)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	1	Befehlszeit:	250 ns	125 ns

ADC – ADDIERE REGISTER MIT CARRY-FLAG

Syntax:	Funktion:
ADC *Rd, Rr*	*Rd* ← *Rd* + *Rr* + *C*

Beschreibung:
Der Inhalt des Registers **Rr** und das C-Flag (Carry-Flag) des Statusregisters werden zum Inhalt des Registers **Rd** addiert. Das Ergebnis der Addition steht im Register **Rd**. Der Inhalt des Registers **Rr** bleibt unverändert.
*(zulässig für **Rd, Rr**: r0 bis r31)*

beeinflusste Flags:	Taktzyklen:	AVR	bei 4 MHz:	bei 8 MHz:
H S V N Z C	1	Befehlszeit:	250 ns	125 ns

2-Adress Befehlsformat (AVR8)

- opcode | Dest/Src1 | Src2

ADC – Add with Carry

Description:

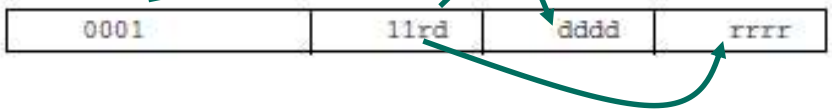
Adds two registers and the contents of the C Flag and place

Operation:

(i) Rd ← Rd + Rr + C

	Syntax:	Operands:
(i)	ADC Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31

16-bit Opcode:



Instruction encoding

Bit assignments:

- rrrr = Source register
- rrr = Source register (R16-R31)
- rr = Source register (R16-R23)
- RRRR = Source register pair (R0:R1 ... R30:R31)
- ddddd = Destination register
- dddd = Destination register (R16-R31)
- ddd = Destination register (R16-R23)
- DDDD = Destination register pair (R0:R1 ... R30:R31)
- pp = Register pair, W, X, Y or Z
- y = Y/Z register pair bit (0=Z, 1=Y)
- u = FMUL(S(U)) signed with 0=signed or 1=unsigned
- s = Store/load bit (0=load, 1=store)
- c = Call/jump (0=jump, 1=call)
- cy = With carry (0=without carry 1=with carry)
- e = Extend indirect jump/call address with EIND (0=0:Z, 1=
- q = Extend program memory address with RAMPZ (0=0:Z,
- aaaaaa = I/O space address
- aaaaa = I/O space address (first 32 only)
- bbb = Bit number

2^5 = 32bit

2-Adress Befehlsformat (AVR8)

- opcode | Dest/Src1 | Src2

AVR Instruction Set

ADD – Add without Carry

Description:

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

1-Adress Befehlsformat (AVR8)

Karlsruher Institut für Technologie

AVR Instruction Set

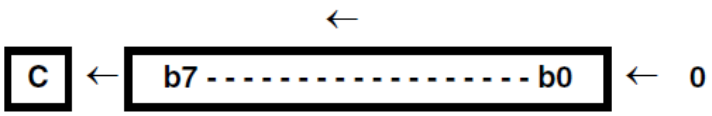
LSL – Logical Shift Left

Description:

Shifts all bits in Rd one place to the left. Bit 0 is cleared. Bit 7 is loaded into the C Flag of the SREG. This operation effectively multiplies signed and unsigned values by two.

Operation:

(i)



	Syntax:	Operands:	Program Counter:
(i)	LSL Rd	$0 \le d \le 31$	PC ← PC + 1

16-bit Opcode: (see ADD Rd,Rd)

0000	11dd	dddd	dddd

B

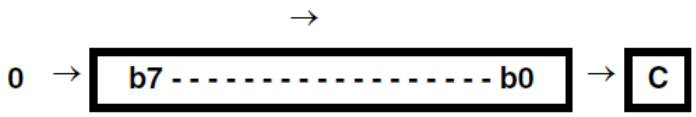
1-Adress Befehlsformat (AVR8)

LSR – Logical Shift Right

Description:

Shifts all bits in Rd one place to the right. Bit 7 is cleared. Bit 0 is loaded into the C Flag of the SREG. This operation effectively divides an unsigned value by two. The C Flag can be used to round the result.

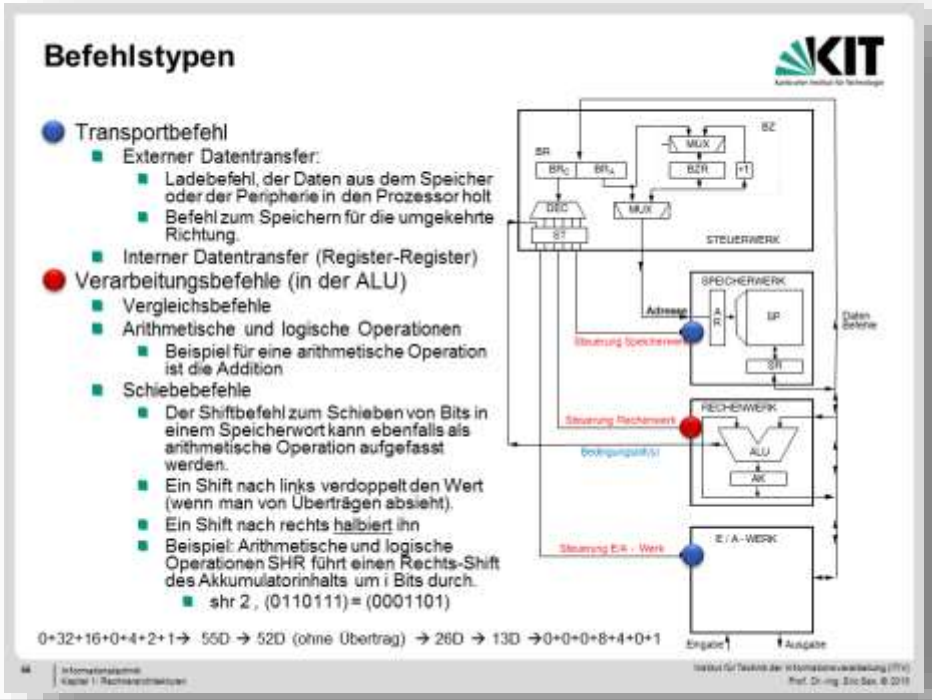
Operation:



	Syntax:	Operands:
(i)	LSR Rd	$0 \le d \le 31$

16-bit Opcode:

1001	010d	dddd	0110



0-Adress Befehlsformat (AVR8)

- 0-Adress Befehlsformat: nur Befehlscode
- Wird z.B. verwendet, um Wartezyklen beim Hardware-Zugriff zu realisieren.

NOP – No Operation

Description:

This instruction performs a single cycle No Operation.

Operation:

(i) No

	Syntax:	Operands:	Program Counter:
(i)	NOP	None	PC ← PC + 1

16-bit Opcode:

0000	0000	0000	0000