Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 2: Rechnerarchitektur und -organisation

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

04. November 2020

Universität Augsburg
Institut für Informatik

2. Rechnerarchitektur und -organisation

Programmierbarkeit

- Ein Computer ist eine (wenn auch sehr komplexe) elektrische Maschine, die genau tut, was man ihr sagt (eingibt)
- Er ist damit so etwas wie ein komplizierter Kaffeeautomat, aber mit einem **Unterschied**: er ist **programmierbar**

Programm

Ein Programm ist eine vom Computer ausführbare Verarbeitungsvorschrift (Folge von Befehlen / Anweisungen)

Programmierbarkeit

Wenn Programme (austauschbar) gespeichert werden können, und nicht nur eine endliche Anzahl fest verschalteter Programme existiert, ist eine Maschine **programmierbar**

Rechnerarchitektur

Definition 2.1 (Rechnerarchitektur)

- Die Architektur eines Rechners ist festgelegt durch die **Menge seiner Maschinenbefehle** (seinen **Maschinenbefehlssatz**) und deren **Implementierung**.
- Die Implementierung des Maschinenbefehlssatzes ist festgelegt durch die **Rechnerstruktur**, die **Rechnerorganisation** und die **Rechnerrealisierung**.

Definition 2.2 (Maschinenbefehl (vereinfacht))

Ein **Maschinenbefehl** ist eine elementare Operation, die

- ein Rechner ausführen kann, und
- vom Programmierer verwendbar ist

Rechnerarchitektur

Rechnerstruktur

Art der Verknüpfung der verschiedenen Hardwarebausteine eines Rechners (Prozessoren, Speicher, Busse, E/A-Geräte)

Rechnerorganisation

Zeitabhängige Wechselwirkung zwischen den Rechner-Komponenten und ihre Steuerung

Rechnerrealisierung

Logischer Entwurf und physische Ausgestaltung der Rechner-Bausteine

Rechnerarchitektur in dieser Vorlesung

Grundstruktur und -organisation heutiger Rechner **ohne technische Details**:

- abstrakte Sicht
- nur so viele Details, dass plausibel wird, wie es funktioniert

Von-Neumann-Rechner

Theoretisches, aber auch mechanisch umsetzbares Konzept des Aufbaus eines Rechners. Noch heutige Rechner beruhen wesentlich auf diesem Konzept.

Ausblick

Das Thema Rechnerarchitektur füllt ohne Probleme ein Lehrbuch und benötigt zur vollständigen Behandlung eine eigene komplette Vorlesung (genauere Behandlung in der Vorlesung *Systemnahe Informatik*).
Aktuelle Entwicklungen:

- Parallelrechner (im praktischen Einsatz, siehe weiterführende Vorlesungen)
- Quantencomputer (theoretisches Konzept, nur in sehr kleinem Maßstab realisiert)

Vorentwicklungen heutiger Rechner vor 1950

Forschungsergebnisse

- Entwicklung von Rechenvorschriften
- Theoretische Konzepte von Rechenmaschinen
- Konkret konstruierte Rechenmaschinen (für einfache arithmetische Operationen)

Auswahl einiger einflussreicher Forscher

Adam Ries(e), Blaise Pascal, Gottfried Wilhelm Leibniz, Charles Babbage, Konrad Zuse, John von Neumann

Ausblick: Pioniere auf anderen Gebieten

Kurt Gödel, Alan Turing, Noam Chomsky, Donald E. Knuth, Stephen A. Cook, Denis Ritchie, Ken Thompson, Richard Stallman, Tim Berners-Lee, Carl Adam Petri, Edgar F. Codd, E. W. Dijkstra, G. Booch, Tom deMarco, Erich Gamma, S.T. Kleene, Robin Milner, Amir Pnueli, Claude E. Shannon, Marvin L Minsky

Entwicklung von Rechenmaschinen ab ca. 1950

1950er Jahre

- Elektronenröhren / Transistoren als Schaltelemente
- 1000 elementare Operationen/Sekunde
- kein Betriebssystem
- programmierbar (Lochkarten, COBOL, FORTRAN, ALGOL...)

1960er Jahre

- höhere Programmiersprachen (C), Betriebssysteme (Unix)

Entwicklung von Rechenmaschinen ab ca. 1950

1970er Jahre

- Rechner werden schneller, kleiner, billiger, komplexer, mächtiger, benutzerfreundlicher
- Eingabe über Tastatur mit Bildschirm

1980er Jahre

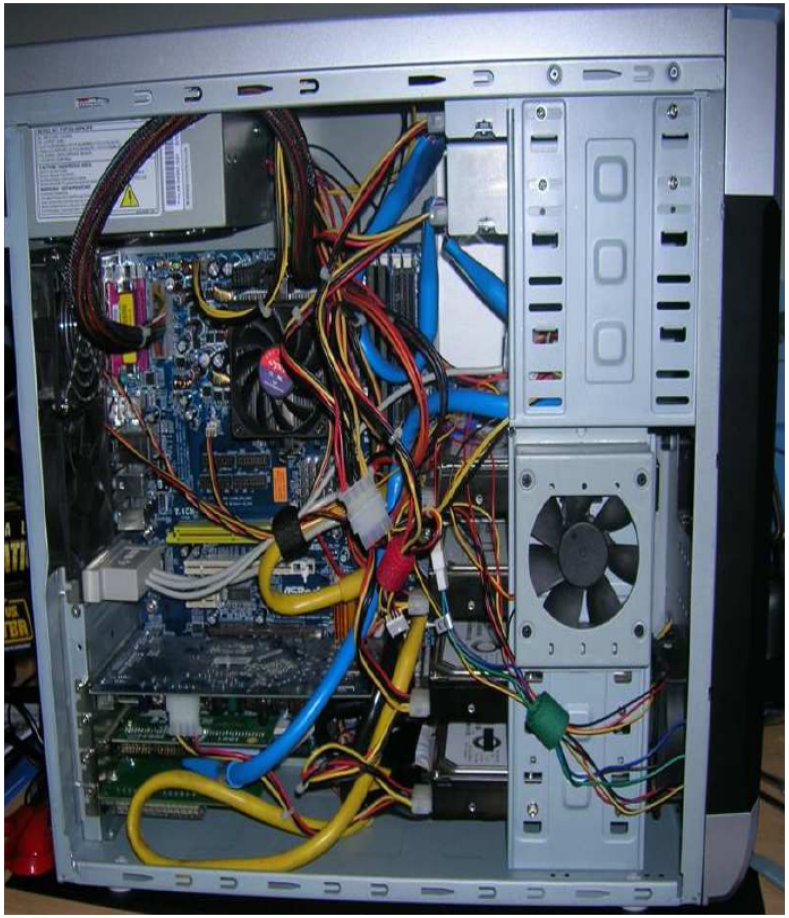
- Halbleiterschaltkreise (hochintegriert, VLSI)
- 2.000.000.000 elementare Operationen/Sekunde
- PCs

Aber ...

Grundlegende Theorien trotzdem nach wie vor gültig – z.B. Algorithmusbegriff (Abstraktion von technischer Realisierung)

Rechner - Innenansicht

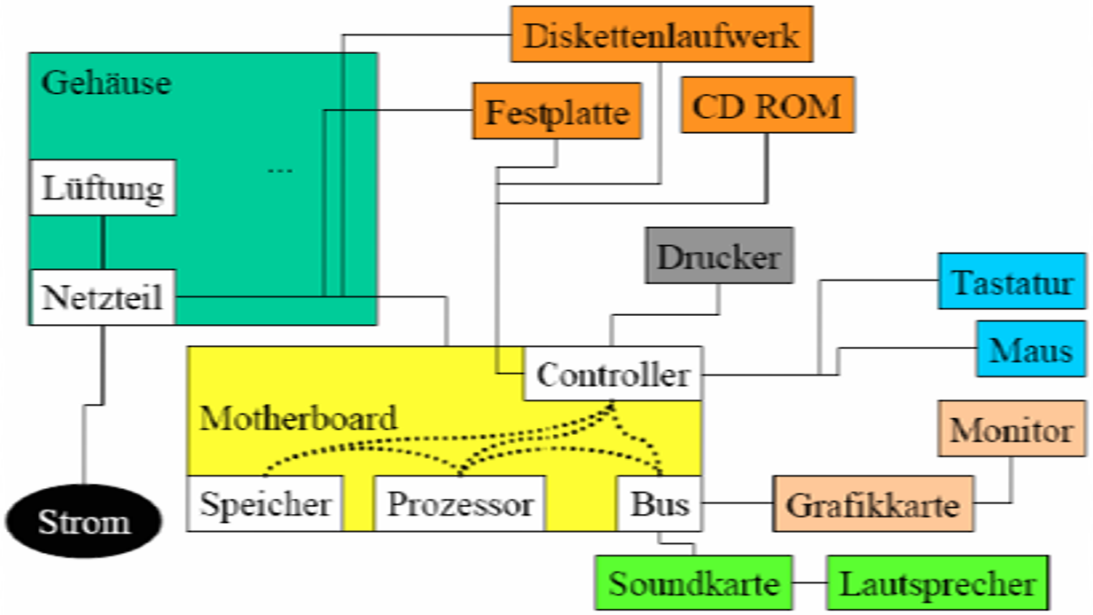
Komplexe Zusammenhänge verstehen durch Abstraktion



(Bild schon etwas veraltet)

Rechner - Logischer Aufbau

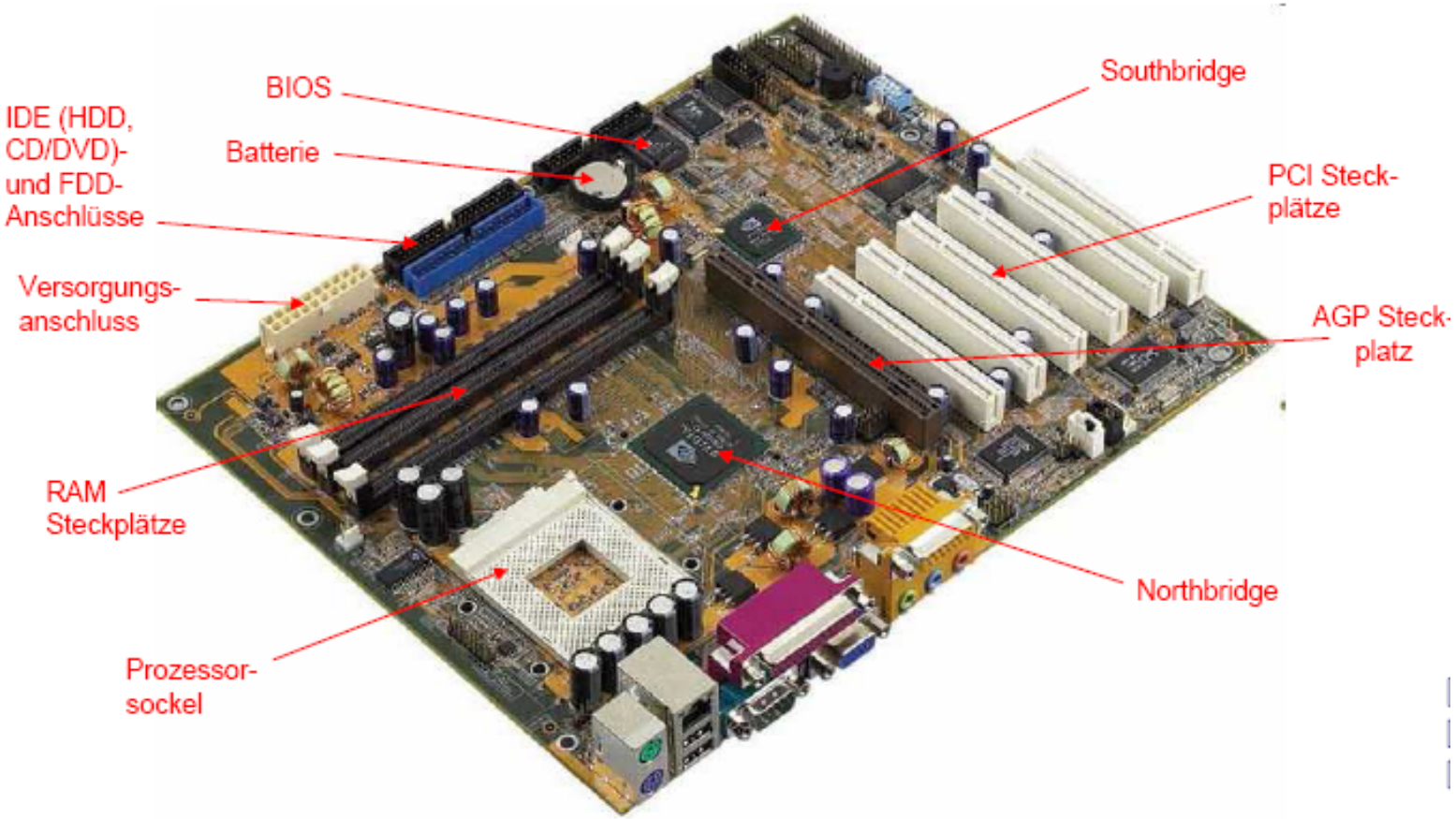
Komplexe Zusammenhänge verstehen durch Abstraktion



(Bild schon etwas veraltet)

Motherboard - Ansicht

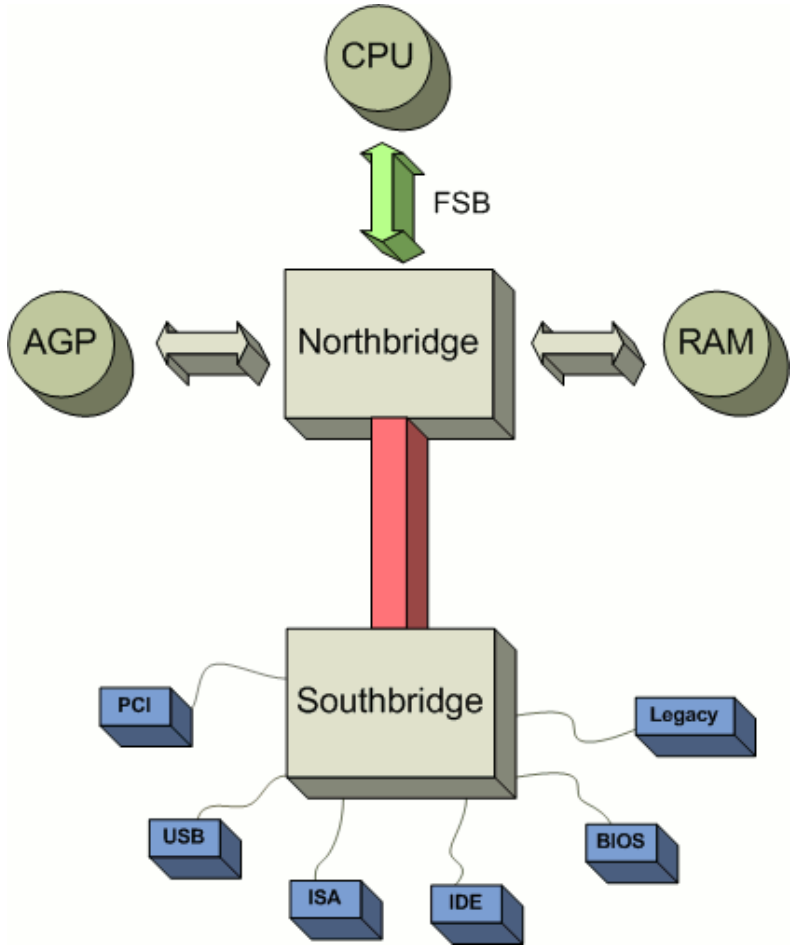
Komplexe Zusammenhänge verstehen durch Abstraktion



(Bild schon etwas veraltet)

Motherboard - logischer Aufbau

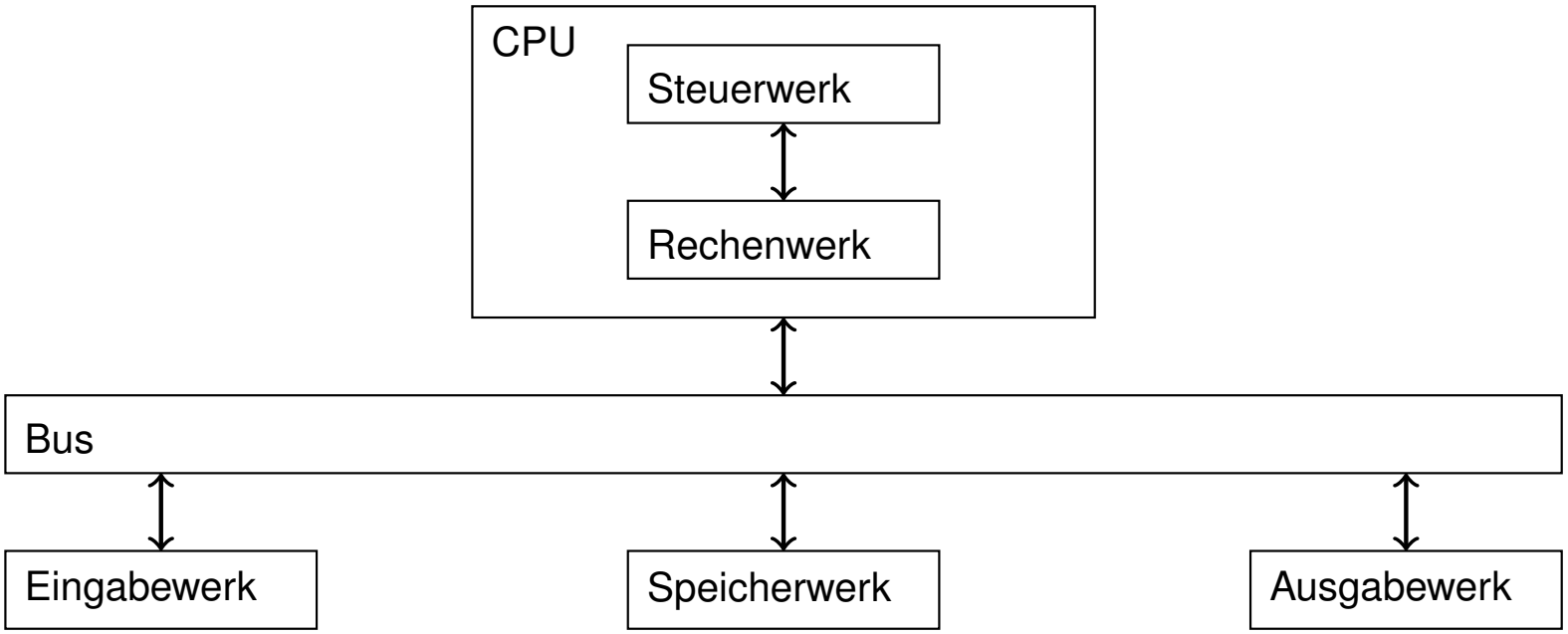
Komplexe Zusammenhänge verstehen durch Abstraktion



(Bild schon etwas veraltet)

Von-Neumann-Rechner - Übersicht

(nach Burks, Goldstine und von Neumann, 1946)



- CPU = Central Processing Unit
- Bus: Datenaustausch zwischen den Komponenten (Steuerwerk, Rechenwerk, Eingabewerk, Ausgabewerk, Speicherwerk)

Von-Neumann-Rechner - Komponenten

Speicherwerk

Dient der Speicherung von Daten **und** Programmen

Rechenwerk

Dient der Ausführung von Maschinenbefehlen

Steuerwerk

Dient der Steuerung des Programmablaufs

Eingabewerk

Dient der Eingabe von Programmen/Daten in das Speicherwerk (von Tastatur, Festplatte, CD, Maus,...)

Ausgabewerk

Dient der Ausgabe von Daten aus dem Speicherwerk (zu Drucker, Monitor, Festplatte, CD,...)

Speichereinheiten

Definition 2.3 (Bit)

Ein **Bit (binary digit)** ist ein Speicherplatz für die kleinstmögliche Informationsmenge zur Unterscheidung zwischen 2 Zuständen: 1 oder 0, Ja oder Nein, Ein oder Aus

Definition 2.4 (Byte)

Ein **Byte (B)** ist eine Gruppierung von acht Bit zur Darstellung eines Zeichens: Es können $2^8 = 256$ Zustände unterschieden werden

Weitere Einheiten

- Kilobyte / Kibibyte: $1\,KB = 10^3\,B \approx 2^{10}\,B = 1\,KiB$
- Megabyte / Mebibyte: $1\,MB = 10^6\,B \approx 2^{20}\,B = 1\,MiB$
- Gigabyte / Gibibyte: $1\,GB = 10^9\,B \approx 2^{30}\,B = 1\,GiB$
- Terabyte / Tebibyte: $1\,TB = 10^{12}\,B \approx 2^{40}\,B = 1\,TiB$

Speicherstruktur

- Die Struktur des Speicherwerks ist **unabhängig** von den zu bearbeitenden Programmen
- Daten und Programme werden im selben Speicher abgelegt

Definition 2.5 (Speicherzellen (SZ))

- Der Speicher ist in gleich große Einheiten unterteilt, in sog. **Speicherzellen (SZ)**
- Die SZ sind mittels eindeutiger Zahlen, sogenannter **Adressen**, durchnummeriert: Direkter Zugriff auf eine SZ über ihre Adresse möglich)

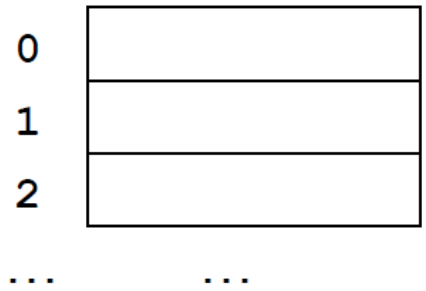


Abbildung: Jede SZ hat eine eindeutige Adresse

Speicherstruktur

- Jede SZ besteht aus einer festen Anzahl `k` von Bits (typisch: $k \in \{8, 16, 32, 64\}$)
- Eine SZ dient zur Speicherung eines Zeichens, einer Zahl oder eines (Programm-)Befehls

Folgerung

Buchstaben, Zahlen und Befehle werden durch Folgen von `0`en und `1`en dargestellt (kodiert) - Details dazu später

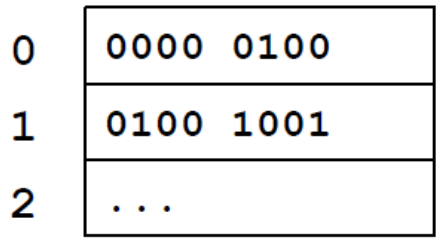


Abbildung: SZ mit 1 Byte Speicherplatz

Realisierung des Speicherwerks

Kriterien für die Realisierung des Speicherwerks:

- schneller Zugriff
- hohe Kapazität
- geringe Kosten

Problemstellung 2.6

Die Kriterien zur Speicherrealisierung widersprechen sich:

- *Große Kapazität + geringe Kosten = Langsamer Zugriff*
- *Schneller Zugriff = hohe Kosten + kleine Kapazität*

Lösung 2.7 (Speicherhierarchie)

*In einem PC gibt es mehrere Arten von Speichern, die abgestimmt zusammenarbeiten (**Speicherhierarchie**)*

Realisierung des Speicherwerks

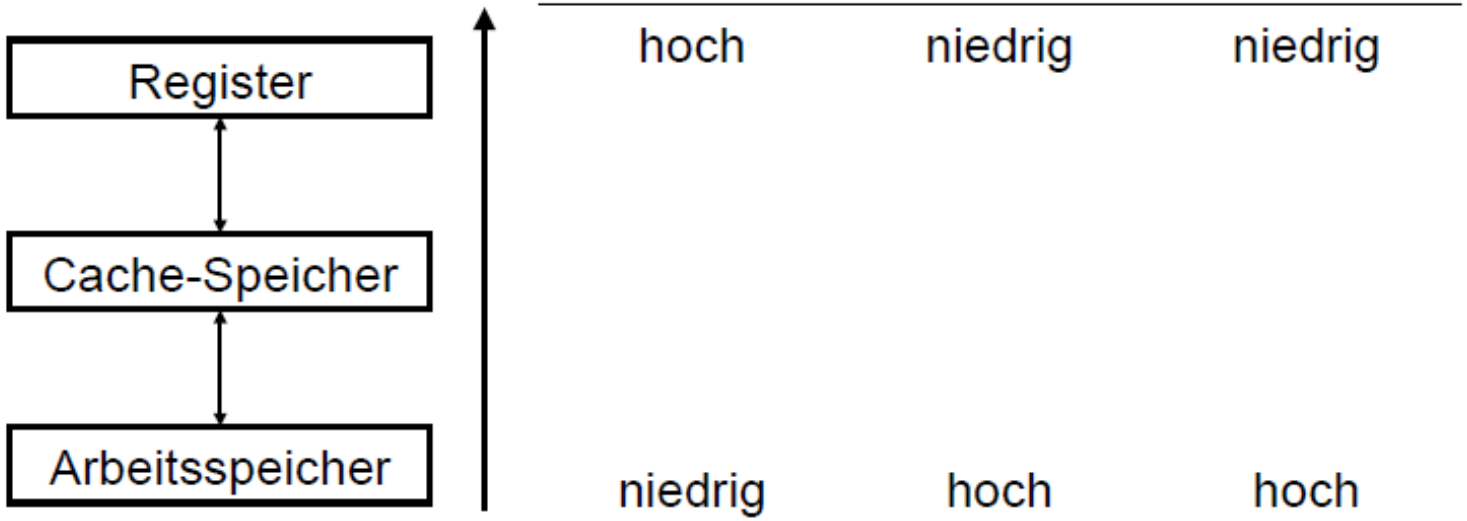


Abbildung: *Übersicht Speicherhierarchie (vereinfacht)*

Komponenten der Speicherhierarchie

Register

Sind unmittelbar der CPU zugeordnet

Cache

- Bindeglied zwischen CPU und Arbeitsspeicher
- Wird für häufig/voraussichtlich benutzte Befehle/Daten verwendet

Arbeitsspeicher

Bezeichnet als RAM (Random Access Memory): enthält Daten / Programme

RAM - Random Access Memory

Definition 2.8 (RAM)

Flüchtiger Speicher (Strom weg = Daten weg) zum Lesen und Schreiben mit wahlfreiem Zugriff auf Daten

- **Nach dem Einschalten**: leerer RAM wird mit Programmen und Daten aus externen permanenten Speichern (z.B. Festplatte) gefüllt
- **Vor dem Ausschalten**: Speicherung der veränderten Daten auf die externen permanenten Speicher
- Technische Realisierungen: statischer RAM (SRAM), dynamischer RAM (DRAM)

DRAM - Dynamic Random Access Memory

Günstiger Speicher für den Einsatz im Arbeitsspeicher

Realisierung eines Bits

Ein Bit wird realisiert durch Kondensator + Transistor:

- Bitwert 1 = Kondensator aufgeladen
- Bitwert 0 = Kondensator ungeladen
- Leseanforderung: Transistor gibt die elektrische Ladung frei

Nachteil

Kondensatoren verlieren Ladung durch Leckströme:

- Auffrischung der Ladung einige 100 Male in der Sekunde
- Während der Auffrischung können keine Daten gelesen werden

SRAM - Static Random Access Memory

Teurer Speicher für den Einsatz im Cache

Realisierung eines Bits

Komplizierte Schaltung (sog. Flipflops) mit 4-6 Transistoren pro Bit

- keine Auffrischung nötig
- 100 mal schneller als DRAM

Nachteil

- benötigt mehr Platz und Energie: wesentlich teurer, weniger Kapzität
- produziert mehr Hitze: muss wesentlich besser gekühlt werden

Externe Speicher

Definition 2.9 (Externe Speicher)

Externe Speicher sind nicht Teil des Speicherwerks, sondern über das Eingabe-/Ausgabewerk mit dem Rechner verbunden

Externe Speicher bieten i.d.R. vergleichsweise langsamen Zugriff, da oft mit mechanisch bewegten Teilen realisiert (z.B. magnetische Festplatte)

Beispiele 2.10 (Externe Speicher)

- ROM
- Magnetische Platten und Bänder
- Flash-Speicher
- Optische Speicher (DVD)

Externe Speicher: ROM - Read Only Memory

Definition 2.11 (ROM)

Nicht-flüchtiger Speicher, der nur gelesen werden kann. Im Computer für das BIOS (Basic Input/Output System) verwendet: Enthält Systemfunktionen für die Initialisierung der Hardware und des Betriebssystems

- Das BIOS wird zum Booten (Hochfahren, Starten) des Rechners verwendet
- Das BIOS wird zunehmend durch das neuere UEFI (Unified Extensible Firmware Interface) abgelöst
- Technische Realisierungen: ROM, PROM, EPROM, EEPROM, Flash-Speicher

2. Rechnerarchitektur und -organisation

Komponenten des Rechenwerks

- Das Rechenwerk besteht aus sog. **Funktionseinheiten**.
- Eine Funktionseinheit realisiert einen Maschinenbefehl.

Beispiel 2.12 (Maschinenbefehle in dieser Vorlesung)

Im Folgenden bezeichnen `A` und `B` Speicheradressen. Diese dienen als Operanden (Argumente) der Maschinenbefehle. Der an einer Adresse `A` gespeicherte Wert heißt deren **Inhalt**.

- `INIT A` : Speichere den Wert `0` an Adresse `A`
- `ADD A,B` : Addiere zum Inhalt an Adresse `A` den Inhalt an Adresse `B`
- `SUB A,B` : Subtrahiere vom Inhalt an Adresse `A` den Inhalt an Adresse `B`
- `DEKREMENT A`: Vermindere den Inhalt an Adresse `A` um `1`
- `DEKREMENT0 A,B`: Falls der Inhalt an Adresse `B` gleich `0` ist, vermindere den Inhalt an Adresse `A` um `1`
- `INKREMENT A`: Erhöhe den Inhalt an Adresse `A` um `1`
- `INKREMENT0 A,B`: Falls der Inhalt an Adresse `B` gleich `0` ist, erhöhe den Inhalt an Adresse `A` um `1`

Komponenten des Rechenwerks

- Das Rechenwerk besteht aus sog. **Funktionseinheiten**.
- Eine Funktionseinheit realisiert einen Maschinenbefehl.

Beispiel 2.13 (Maschinenbefehle in dieser Vorlesung - Fortsetzung)

Im Folgenden bezeichnen `A` und `B` Speicheradressen. Diese dienen als Operanden (Argumente) der Maschinenbefehle. Der an einer Adresse `A` gespeicherte Wert heißt deren **Inhalt**.

- `SPRUNG A` : Gehe zu Adresse `A`
- `SPRUNG0 A,B` : Falls der Inhalt an Adresse `B` gleich `0` ist, gehe zu Adresse `A`
- `RÜCKGABE A` : Gib den Inhalt an Adresse `A` zurück
- `RÜCKGABE0 A,B` : Falls der Inhalt an Adresse `B` gleich `0` ist, gib den Inhalt an Adresse `A` zurück

Maschinenbefehle - Übersicht

- `INIT A` : Speichere den Wert `0` an Adresse `A`
- `ADD A,B` : Addiere zum Inhalt an Adresse `A` den Inhalt an Adresse `B`
- `SUB A,B` : Subtrahiere vom Inhalt an Adresse `A` den Inhalt an Adresse `B`
- `DEKREMENT A`: Vermindere den Inhalt an Adresse `A` um `1`
- `DEKREMENT0 A,B`: Falls der Inhalt an Adresse `B` gleich `0` ist, vermindere den Inhalt an Adresse `A` um `1`
- `INKREMENT A`: Erhöhe den Inhalt an Adresse `A` um `1`
- `INKREMENT0 A,B`: Falls der Inhalt an Adresse `B` gleich `0` ist, erhöhe den Inhalt an Adresse `A` um `1`
- `SPRUNG A` : Gehe zu Adresse `A`
- `SPRUNG0 A,B` : Falls der Inhalt an Adresse `B` gleich `0` ist, gehe zu Adresse `A`
- `RÜCKGABE A` : Gib den Inhalt an Adresse `A` zurück
- `RÜCKGABE0 A,B` : Falls der Inhalt an Adresse `B` gleich `0` ist, gib den Inhalt an Adresse `A` zurück

Realisierung von Funktionseinheiten

Definition 2.14 (Realisierung einer Funktionseinheit)

Mögliche Realisierungen einer Funktionseinheit:

- als fest installierte Schaltung (in sog. RISC-Prozessoren):
  Ein Befehl wird direkt in Steuersignale umgesetzt (platzsparend)
- als ein sog. **Mikroprogramm** (in sog. CISC-Prozessoren)

Definition 2.15 (Mikroprogramm)

Ein **Mikroprogramm** ist eine Folge von Mikrobefehlen, welche wiederum durch fest installierte Schaltungen realisiert sind

Register im Rechenwerk

Für die Eingabe- und Ausgabedaten einer Operation:

Akkumulator-Register

Zu jeder Funktionseinheit gehören ein oder mehrere **Akkumulator-Register (AR)** zur Speicherung von Operanden und Ergebnissen der Operation

Zur Kommunikation mit dem Steuerwerk:

Status-Register

Zu jeder Funktionseinheit gehört ein **Status-Register (SR)** zur Speicherung des Zustands der Funktionseinheit

Register im Rechenwerk

Zur Kommunikation mit dem Speicherwerk:

Adress-Memory-Port (AM)

Enthält (als Dateninhalt) die Adresse der SZ, in die geschrieben bzw. aus der gelesen werden soll

Write-Memory-Port (WM)

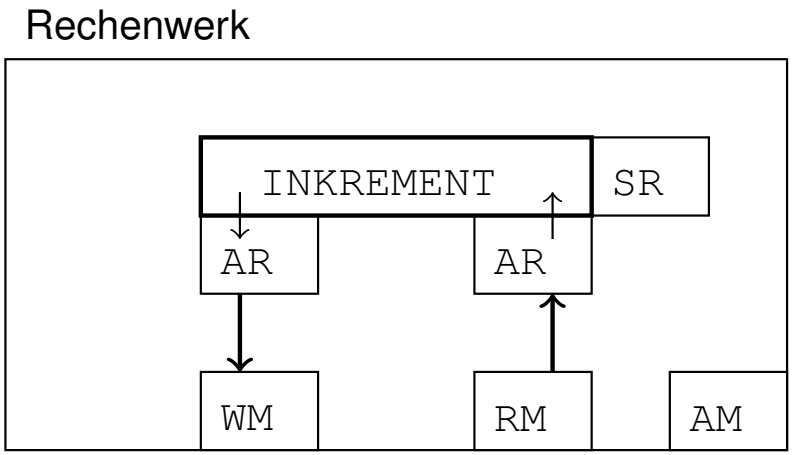
Enthält den zu schreibenden Inhalt für die durch AM adressierte SZ

Read-Memory-Port (RM)

Enthält den gelesenen Inhalt der adressierten SZ

Rechenwerk - Übersicht

Die folgende Grafik zeigt beispielhaft das Rechenwerk mit der Funktionseinheit zum Befehl `INKREMENT A`.



- Adresse `A` wird nach `AM` geschrieben
- Der Inhalt an Adresse in `AM` wird über `RM` nach `AR` geschrieben
- Die Funktionseinheit wird ausgeführt
- Das Ergebnis wird über `AR` nach `WM` geschrieben
- Der Inhalt von `WM` wird nach Adresse in `AM` geschrieben

2. Rechnerarchitektur und -organisation

Was macht das Steuerwerk?

Das Steuerwerk steuert den Rechner:

- Alle anderen Komponenten erhalten ihre Befehle vom Steuerwerk
- Ist zuständig für die Ausführung der in einem Programm niedergelegten Arbeitsvorgänge

EVA-Prinzip

Ein ins Steuerwerk geladener Befehl wird nach dem **EVA-Prinzip** ausgeführt

- (E) Eingabe: Daten einzeln aus dem Speicherwerk (Adresse steht in AM) über RM nach AR laden
- (V) Verarbeitung: Ausführung der Operation durch eine Funktionseinheit
- (A) Ausgabe: Ergebnis-Daten von AR über WM in das Speicherwerk (Adresse steht in AM) schreiben

Register im Steuerwerk

Instruktionsregister (IR)

Enthält den aktuellen Befehl, der gerade auszuführen ist

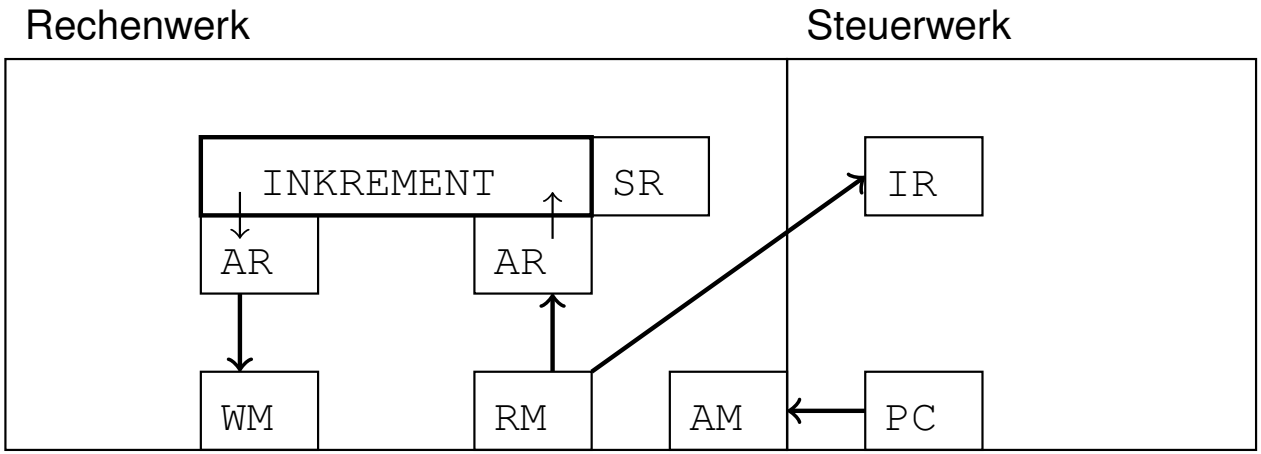
Programm Counter (PC)

Enthält die Adresse des aktuellen Befehls im Speicherwerk. Der PC wird wie folgt aktualisiert:

- Fall 1 - Der Befehl war kein Sprungbefehl:
  PC wird auf die nachfolgende Adresse gesetzt
- Fall 2 - Der Befehl war ein Sprungbefehl zu einer Adresse:
  PC wird auf die Sprungbefehl-Adresse gesetzt

Steuerwerk - Übersicht

Die folgende Grafik zeigt beispielhaft das Steuerwerk und das Rechenwerk mit der Funktionseinheit zum Befehl `INKREMENT A`.



- Adresse in `PC` wird nach `AM` geschrieben
- Der Inhalt an Adresse in `AM` wird über `RM` nach `IR` geschrieben (Befehl holen)
- Der Befehl wird ausgeführt (siehe Rechenwerk)
- Die Adresse in `PC` wird aktualisiert

Fetch/Decode/Execute/Write-Back

Ein Programm wird nach folgendem Zyklus abgearbeitet

1. Holen des nächsten Befehls (FETCH)
   - Befehl gemäß PC adressieren und aus dem Speicherwerk über RM nach IR laden
   - Befehlszähler PC aktualisieren
2. Entschlüsseln des Befehls (DECODE)
   - Erkennen der Befehlsart (Zuordnung zu einer Funktionseinheit)
   - Zerlegung in Bestandteile (Operation und Operanden)
   - (E) Eingabe der Operanden aus dem Speicherwerk
3. Ausführung des Befehls (EXECUTE)
   - (V) Verarbeitung der Daten im Rechenwerk
   - Befehlszähler PC bei Sprung überschreiben
4. Ausgabe des Ergebnisses (WRITE-BACK)
   - (A) Ausgabe des Ergebnisses in das Speicherwerk

Fetch/Decode/Execute/Write-Back

Ein Programm wird nach folgendem Zyklus abgearbeitet

1. Holen des nächsten Befehls (FETCH)
   - Befehl gemäß PC adressieren und aus dem Speicherwerk über RM nach IR laden
   - Befehlszähler PC aktualisieren
2. Entschlüsseln des Befehls (DECODE)
   - Erkennen der Befehlsart (Zuordnung zu einer Funktionseinheit)
   - Zerlegung in Bestandteile (Operation und Operanden)
   - (E) Eingabe der Operanden aus dem Speicherwerk
3. Ausführung des Befehls (EXECUTE)
   - (V) Verarbeitung der Daten im Rechenwerk
   - Befehlszähler PC bei Sprung überschreiben
4. Ausgabe des Ergebnisses (WRITE-BACK)
   - (A) Ausgabe des Ergebnisses in das Speicherwerk

Fetch/Decode/Execute/Write-Back

Ein Programm wird nach folgendem Zyklus abgearbeitet

1. Holen des nächsten Befehls (FETCH)
   - Befehl gemäß PC adressieren und aus dem Speicherwerk über RM nach IR laden
   - Befehlszähler PC aktualisieren
2. Entschlüsseln des Befehls (DECODE)
   - Erkennen der Befehlsart (Zuordnung zu einer Funktionseinheit)
   - Zerlegung in Bestandteile (Operation und Operanden)
   - (E) Eingabe der Operanden aus dem Speicherwerk
3. Ausführung des Befehls (EXECUTE)
   - (V) Verarbeitung der Daten im Rechenwerk
   - Befehlszähler PC bei Sprung überschreiben
4. Ausgabe des Ergebnisses (WRITE-BACK)
   - (A) Ausgabe des Ergebnisses in das Speicherwerk

Fetch/Decode/Execute/Write-Back

Ein Programm wird nach folgendem Zyklus abgearbeitet

1. Holen des nächsten Befehls (FETCH)
   - Befehl gemäß PC adressieren und aus dem Speicherwerk über RM nach IR laden
   - Befehlszähler PC aktualisieren
2. Entschlüsseln des Befehls (DECODE)
   - Erkennen der Befehlsart (Zuordnung zu einer Funktionseinheit)
   - Zerlegung in Bestandteile (Operation und Operanden)
   - (E) Eingabe der Operanden aus dem Speicherwerk
3. Ausführung des Befehls (EXECUTE)
   - (V) Verarbeitung der Daten im Rechenwerk
   - Befehlszähler PC bei Sprung überschreiben
4. Ausgabe des Ergebnisses (WRITE-BACK)
   - (A) Ausgabe des Ergebnisses in das Speicherwerk

Fetch/Decode/Execute/Write-Back

Ein Programm wird nach folgendem Zyklus abgearbeitet

1. Holen des nächsten Befehls (FETCH)
   - Befehl gemäß PC adressieren und aus dem Speicherwerk über RM nach IR laden
   - Befehlszähler PC aktualisieren
2. Entschlüsseln des Befehls (DECODE)
   - Erkennen der Befehlsart (Zuordnung zu einer Funktionseinheit)
   - Zerlegung in Bestandteile (Operation und Operanden)
   - (E) Eingabe der Operanden aus dem Speicherwerk
3. Ausführung des Befehls (EXECUTE)
   - (V) Verarbeitung der Daten im Rechenwerk
   - Befehlszähler PC bei Sprung überschreiben
4. Ausgabe des Ergebnisses (WRITE-BACK)
   - (A) Ausgabe des Ergebnisses in das Speicherwerk

Fetch/Decode/Execute/Write-Back

(Zahlen in `[]`-Klammern bezeichnen Adressen)

Beispiel 2.16 (Abarbeitungszyklus)

Inhalt von PC: `[1000]`
Inhalt von SZ `[1000]`: `INCREMENT [500]`
Inhalt von SZ `[500]`: `17`

1. **Fetch**:
   Hole den Inhalt von SZ `[1000]` über RM nach IR
   Setze Befehlszähler PC auf den nächsten Adresswert
2. **Decode**:
   Erkenne, dass es sich um die Inkrement-Operation für die SZ `[500]` handelt
   (E) Lade Inhalt (`17`) von SZ `[500]` über RM nach AR
3. **Execute**:
   (V) Führe Inkrement-Operation aus (`17+1 = 18`)
4. **Write-Back**:
   (A) Schreibe Ergebnis (`18`) von AR über WM nach SZ `[500]`

Fetch/Decode/Execute/Write-Back

(Zahlen in `[]`-Klammern bezeichnen Adressen)

Beispiel 2.16 (Abarbeitungszyklus)

Inhalt von PC: `[1000]`
Inhalt von SZ `[1000]`: `INCREMENT [500]`
Inhalt von SZ `[500]`: `17`

1. **Fetch**:
   Hole den Inhalt von SZ `[1000]` über RM nach IR
   Setze Befehlszähler PC auf den nächsten Adresswert
2. **Decode**:
   Erkenne, dass es sich um die Inkrement-Operation für die SZ `[500]` handelt
   (E) Lade Inhalt (`17`) von SZ `[500]` über RM nach AR
3. **Execute**:
   (V) Führe Inkrement-Operation aus (`17+1 = 18`)
4. **Write-Back**:
   (A) Schreibe Ergebnis (`18`) von AR über WM nach SZ `[500]`

Fetch/Decode/Execute/Write-Back

(Zahlen in `[]`-Klammern bezeichnen Adressen)

Beispiel 2.16 (Abarbeitungszyklus)

Inhalt von PC: `[1000]`
Inhalt von SZ `[1000]`: `INCREMENT [500]`
Inhalt von SZ `[500]`: `17`

1. **Fetch**:
   Hole den Inhalt von SZ `[1000]` über RM nach IR
   Setze Befehlszähler PC auf den nächsten Adresswert
2. **Decode**:
   Erkenne, dass es sich um die Inkrement-Operation für die SZ `[500]` handelt
   (E) Lade Inhalt (`17`) von SZ `[500]` über RM nach AR
3. **Execute**:
   (V) Führe Inkrement-Operation aus (`17+1 = 18`)
4. **Write-Back**:
   (A) Schreibe Ergebnis (`18`) von AR über WM nach SZ `[500]`

Fetch/Decode/Execute/Write-Back

(Zahlen in `[]`-Klammern bezeichnen Adressen)

Beispiel 2.16 (Abarbeitungszyklus)

Inhalt von PC: `[1000]`
Inhalt von SZ `[1000]`: `INCREMENT [500]`
Inhalt von SZ `[500]`: `17`

1. **Fetch**:
   Hole den Inhalt von SZ `[1000]` über RM nach IR
   Setze Befehlszähler PC auf den nächsten Adresswert
2. **Decode**:
   Erkenne, dass es sich um die Inkrement-Operation für die SZ `[500]` handelt
   (E) Lade Inhalt (`17`) von SZ `[500]` über RM nach AR
3. **Execute**:
   (V) Führe Inkrement-Operation aus (`17+1 = 18`)
4. **Write-Back**:
   (A) Schreibe Ergebnis (`18`) von AR über WM nach SZ `[500]`

Fetch/Decode/Execute/Write-Back

(Zahlen in `[]`-Klammern bezeichnen Adressen)

Beispiel 2.16 (Abarbeitungszyklus)

Inhalt von PC: `[1000]`
Inhalt von SZ `[1000]`: `INCREMENT [500]`
Inhalt von SZ `[500]`: `17`

1. **Fetch**:
   Hole den Inhalt von SZ `[1000]` über RM nach IR
   Setze Befehlszähler PC auf den nächsten Adresswert
2. **Decode**:
   Erkenne, dass es sich um die Inkrement-Operation für die SZ `[500]` handelt
   (E) Lade Inhalt (`17`) von SZ `[500]` über RM nach AR
3. **Execute**:
   (V) Führe Inkrement-Operation aus (`17+1 = 18`)
4. **Write-Back**:
   (A) Schreibe Ergebnis (`18`) von AR über WM nach SZ `[500]`

Steuersignale

Lese- und Schreibvorgänge zwischen Rechenwerk und Speicherwerk werden mit Hilfe von Steuersignalen zwischen Steuerwerk und Speicherwerk koordiniert

Address Strobe (A)

Wird vom Steuerwerk aktiviert, um dem Speicherwerk zu melden, dass eine Adresse aus AM 'gelesen' werden soll

Direction (D)

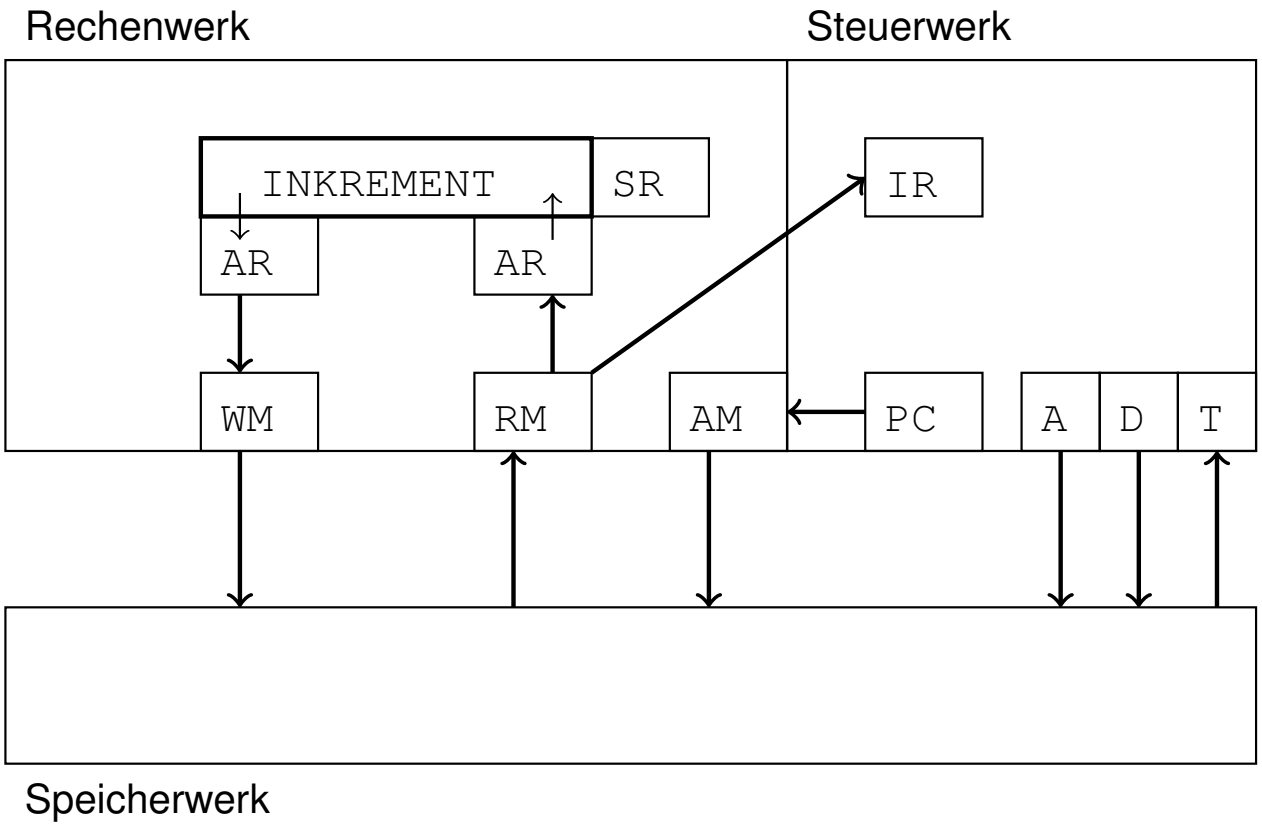
Wird vom Steuerwerk entweder auf 0 oder 1 gesetzt
0: zeigt Speicherwerk an, dass es sich um einen Lesezugriff handelt
1: zeigt Speicherwerk an, dass es sich um einen Schreibzugriff handelt

Data Transfer Acknowledge (T)

Wird vom Speicherwerk zurückgemeldet, nachdem ein Datenzugriff erfolgreich war (nötig, da der Speicher relativ langsam ist)

Kombinierte Gesamtübersicht (1)

Die folgende Grafik zeigt beispielhaft Speicherwerk, Steuerwerk und Rechenwerk in Kombination.



Register und Steuersignale - Übersicht

- `AR`: Akkumulatorregister
- `SR`: Statusregister
- `AM`: Address Memory Port
- `WM`: Write Memory Port
- `RM`: Read Memory Port
- `PC`: Program Counter
- `IR`: Instruction Register
- `A`: Adress Strobe
- `D`: Direction
- `T`: Data transfer Acknowledge

Kommunikation zwischen Speicherwerk und Steuerwerk

Beispiel 2.17 (Schreibvorgang: Wert von `AR` nach SZ `[500]` schreiben)

- Steuerwerk 'schreibt' `[500]` nach `AM`
- Steuerwerk 'schreibt' Wert von `AR` nach `WM`
- Steuerwerk setzt `D` auf Schreiben (`1`)
- Steuerwerk sendet `A`
- Speicherwerk 'liest' Adresse von `AM` (`[500]`)
- Speicherwerk 'liest' Wert von `WM`
- Speicherwerk überschreibt Inhalt der SZ `[500]` mit diesem Wert
- Speicherwerk sendet `T`

Lesevorgänge: Selbst überlegen, siehe auch Übungsblatt

Das Steuerwerk ist getaktet

Taktzyklus

Alle Vorgänge in einem Prozessor laufen **getaktet** ab.
Pro **Takt** werden Fetch/Decode/Execute/Write-Back ausgeführt

Taktfrequenz

Die **Taktfrequenz** ist die Häufigkeit der Takte pro Zeiteinheit:

- Wichtigster Maßstab für die Leistung eines Computers
- **Hertz (Hz)** = Anzahl der Takte pro Sekunde (s)

Zeiteinheiten

- Millisekunde (ms): $1ms = 10^{-3}s$
- Mikrosekunde (μs): $1\mu s = 10^{-6}s$
- Nanosekunde (ns): $1ns = 10^{-9}s$

Takteinheiten

- Kilohertz (KHz): $1KHz = 10^{3}Hz$
- Megahertz (MHz): $1MHz = 10^{6}Hz$
- Gigahertz (GHz): $1GHz = 10^{9}Hz$

Natürliche Grenzen der Schnelligkeit eines Prozessors

Dateneingabe und -ausgabe = Stromfluss auf bestimmten Leitungen

Die Schnelligkeit des Prozessors ist begrenzt durch die Leitungslängen auf dem Prozessorchip

Wie weit kommt ein Elektron in einem Takt auf einem 1GHz Prozessor-Chip?

- Lichtgeschwindigkeit im Vakuum = $300.000 km/s = 0,3 * 10^9 m/s$
- $1 GHz$ entspricht Taktlänge = $10^{-9} sec = 1 ns$: Lichtstrecke = 30 cm/Taktlänge
- Elektronenflussgeschwindigkeit in Kupfer = 2/3 * Lichtgeschwindigkeit: Elektronenstrecke = 20 cm/Taktlänge

Natürliche Grenzen der Schnelligkeit eines Prozessors

Stromfluss erzeugt Wärme.

Die Schnelligkeit des Prozessors ist abhängig von Kühlmöglichkeiten

Elektrische Schaltungen erzeugen linear proportional zur Taktfrequenz Wärme (bei gleichbleibender Spannung)

- Ohne ausreichende Kühlung Gefahr des Durchbrennens (in 10 - 30 Sekunden)
- Prozessoren müssen gekühlt werden (Kühlkörper + Lüfter)

Natürliche Grenzen der Schnelligkeit eines Prozessors

Ein Taktzyklus enthält i.d.R. auch Speicherzugriffe (Ein- und Ausgabe von Daten)

Die Schnelligkeit des Prozessors ist abhängig von Speicherzugriffszeiten, Cachegröße und Qualität der sog. Vorhersagelogik

Speicherzugriffszeiten sind i.d.R. langsamer als der Taktzyklus

- Schnelle Speicher mit 1GHz (oder mehr) sind sehr **sehr teure** Speicher: können nur in kleiner Kapazität verbaut werden
- Kleine schnelle Cache-Speicher werden für **voraussichtlich benutzte Daten** verwendet: Bestimmung dieser Daten erfolgt durch eine sog. **Vorhersagelogik**
- Speicherzugriffe werden, wo möglich, parallelisiert

2. Rechnerarchitektur und -organisation

Was passiert bei Start eines Maschinenprogramms?

- Das Programm wird in den Arbeitsspeicher geladen; Die Befehle befinden sich dann in aufeinanderfolgenden Speicherzellen, dem sog. **Programmteil**; Der Programcounter erhält die Adresse des ersten Befehls des Programms
- Dem Programm werden (neben dem **Programmteil**) drei weitere zusammenhängende Speicherbereiche zugeordnet, nämlich
  Datenteil: für globale und statische Variablen (Details dazu in späteren Kapiteln) und für Konstanten
  Stack: für lokale Variablen (aus dem Vorkurs bekannt).
  Heap: für dynamisch zur Laufzeit verwaltete Variablen (Details dazu in späteren Kapiteln).

Dynamisch vs. automatisch verwaltete Variablen

- Bei **automatisch** verwalteten Variablen steht deren Speicherbedarf schon beim Compilieren des Programms fest
- Bei **dynamisch** verwalteten Variablen steht deren Speicherbedarf erst während der Ausführung des Programms fest

Annahmen und Notationen

Vereinfachende Annahmen

- Wir betrachten nur Maschinencode zu einzelnen C-Funktionen, nicht zu kompletten C-Programmen
- Eine Speicherzelle nimmt genau einen Maschinenbefehl, ein Zeichen oder eine Zahl auf.

Notationen für Speicheradressen

- Adressen im Programmteil haben die Form `P`x für eine Nummerierung x
- Adressen im Datenteil haben die Form `D`x für eine Nummerierung x
- Adressen im Stack haben die Form `S`x für eine Nummerierung x
- Adressen im Heap haben die Form `H`x für eine Nummerierung x

Adressen aufeinanderfolgender Speicherzellen sind fortlaufend nummeriert.

Maschinencode zu einer C-Funktion

Adresse	Befehl
`P1`	`INIT S2`
`P2`	`INKREMENT S2`
`P3`	`SPRUNG0 P7,S1`
`P4`	`ADD S2,S2`
`P5`	`DEKREMENT S1`
`P6`	`SPRUNG P3`
`P7`	`RÜCKGABE S2`
`S1`	`2`
`S2`	

- Programm startet mit Befehl an Adresse `P1`
- Programm endet mit Befehl an Adresse `P7`
- Rückgabe des Programms: `4`
- Rückgaben für andere Inhalte an Adresse `S1` bei Aufruf der Funktion?

Beobachtung: Durch (bedingte) Sprungbefehle können in den Programmablauf Wiederholungen eingebaut werden.

Maschinencode zu einer C-Funktion

- Eingabeparameter *n* ist der Speicheradresse `S1` zugeordnet: für *n* wurde hier bei Aufruf 2 übergeben
- Variable *e* ist der Speicheradresse `S2` zugeordnet
- Der `RÜCKGABE`-Befehl entpricht der `return`-Anweisung: damit endet das Maschinenprogramm

Adresse	Befehl
`P1`	`INIT S2`
`P2`	`INKREMENT S2`
`P3`	`SPRUNG0 P7,S1`
`P4`	`ADD S2,S2`
`P5`	`DEKREMENT S1`
`P6`	`SPRUNG P3`
`P7`	`RÜCKGABE S2`
`S1`	`2`
`S2`	

```
int exp2(int n) {
  int e = 0;
  e = e + 1;
  while (n > 0) {
    e = e + e;
    n = n - 1;
  }
  return e;
}
```

Aufruf: `exp2(2)`

Maschinencode zu einer C-Funktion

- *a* ist der Speicheradresse `S1` zugeordnet: Bei Aufruf wird 5 übergeben
- *b* ist der Speicheradresse `S2` zugeordnet: Bei Aufruf wird 5 übergeben
- *c* ist der Speicheradresse `S3` zugeordnet
- Die Konstante 0 ist der Speicheradresse `D1` zugeordnet
- Die Konstante 1 ist der Speicheradresse `D2` zugeordnet

Adresse	Befehl
P1	INIT S3
P2	ADD S3,S1
P3	SUB S3,S2
P4	SPRUNG0 P6,S3
P5	RÜCKGABE D2
P6	RÜCKGABE D1
D1	0
D2	1
S1	5
S2	5
S3	

```
int is_equal(int a, int b)
{
    int c = a - b;
    if (c == 0) {
        return 1;
    }
    return 0;
}
```

Aufruf: `is_equal(5, 5)`

Datenverwaltung im Arbeitsspeicher

Daten einer C-Funktion werden wie folgt verwaltet:

- **Eingabeparameter**: Adresse im Stack, bei Funktionsaufruf initialisiert mit dem übergebenen Wert
- **Lokale Variablen**: Adresse im Stack, werden erst durch Programmanweisungen / Maschinenbefehle initialisiert
- **Konstanten**: Adresse im Datenteil, vorinitialisiert mit vorgegebenem Wert aus C-Funktion

Mehr zu globalen, statischen und dynamisch verwalteten Variablen in späteren Kapiteln

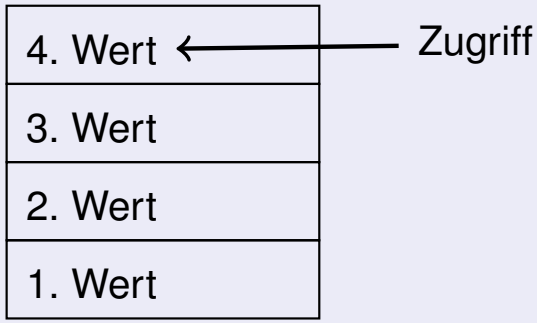
Der Stack

Definition 2.18 (Stack)

Ein **Stack** (**Kellerspeicher**) ist eine zusammenhängende Speicherstruktur mit beschränkten Zugriffsmöglichkeiten

- Es kann immer nur der zuoberst abgespeicherte Wert aus dem Speicher geholt oder gelesen werden
- Neue Werte werden immer zuoberst abgelegt

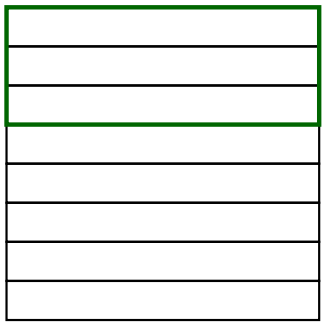
Ein Stack realisiert das sog. **LIFO-Prinzip** (Last in - First out):



Der Stack und lokale Variablen

Verwaltung lokaler Variablen

- Alle lokalen Variablen eines Programms werden im Stack des Programms verwaltet
- Bei Eintritt in einen `{ }`-Anweisungsblock wird zuoberst auf dem Stack ein Bereich für die Aufnahme der zugehörigen lokalen Daten reserviert - ein sog. **Stack Frame**
- Bei Verlassen des Anweisungsblocks wird der Stack Frame wieder freigegeben (kein Zugriff mehr auf die lokalen Daten)
- *Anmerkung: In der praktischen Umsetzung kann auf alle Speicherzellen im aktiven (obersten) Stack Frame zugegriffen werden, nicht nur auf die oberste Speicherzelle*
- *Anmerkung: Der Stack zu einem Programm ist begrenzt (typischerweise 2 MB). Bei zu vielen lokalen Daten kommt es zum Programm-Abbruch wegen **stack overflow**. Der Stack kann beim Compilieren durch eine gcc-Option vergrößert werden.*



Aktiver neuer Stack Frame zu aktuellem Anweisungsblock

inaktiver Teil

Funktionsaufrufe

Bei Aufruf einer Funktion wird

- zur ersten Anweisung im Funktionsrumpf gesprungen
- ein neuer Stack Frame - der **Function Stack Frame** - angelegt zur Aufnahme der lokalen Daten der Funktion

Lokale Daten im Function Stack Frame

Diese Werte werden hintereinander im Stack abgelegt:

- für die Eingabeparameter übergebene Werte
- **Rücksprungadresse**: Adresse des auf den Funktionsaufruf folgenden Befehls im Programm
- lokale Variablen

Das Call-by-Value-Prinzip

Definition 2.19 (Call-by-Value-Prinzip)

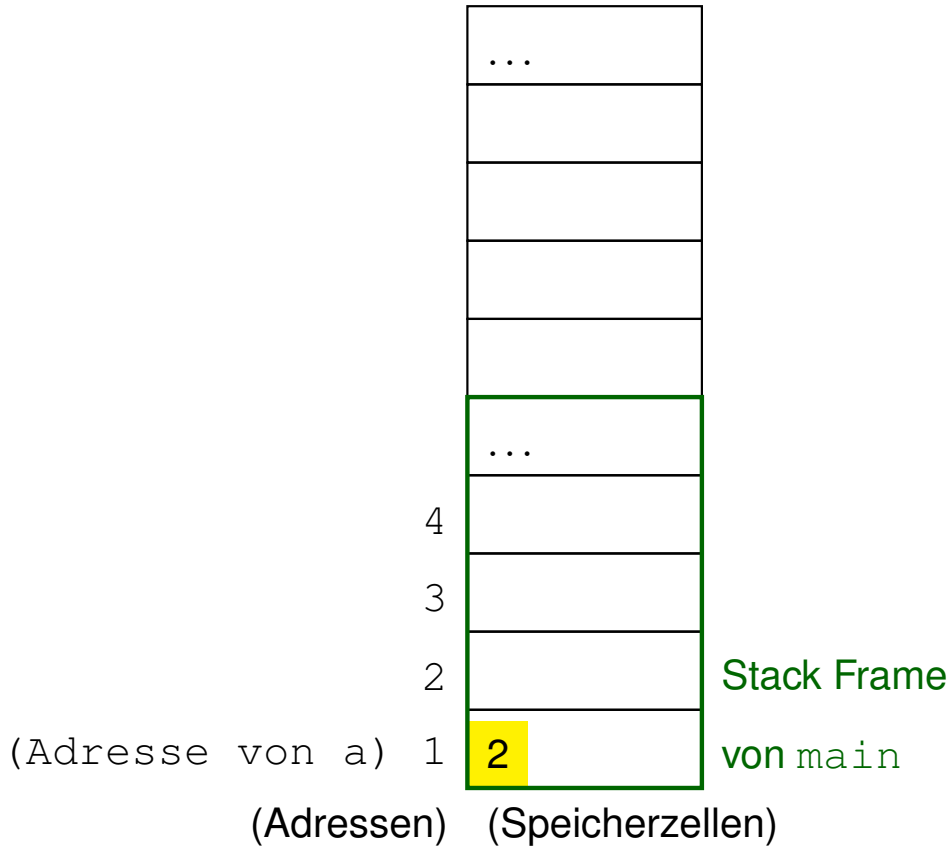
Bei Abarbeitung einer Funktion wird nur mit lokalen Kopien der übergebenen Werte im zugehörigen Stack Frame gerechnet.

- Bei einem Funktionsaufruf werden neue Exemplare (Kopien) für die Eingabeparameter im zugehörigen Stack Frame angelegt.
- Diese Kopien nennt man **lokale Parametervariablen**
- Damit kann der Wert einer übergebenen Variablen durch den Funktionsaufruf **nicht geändert werden**, sondern nur der Wert der zugehörigen Kopie

Beispiel für das Call-by-Value-Prinzip

In der Funktion wird mit Kopien gerechnet.

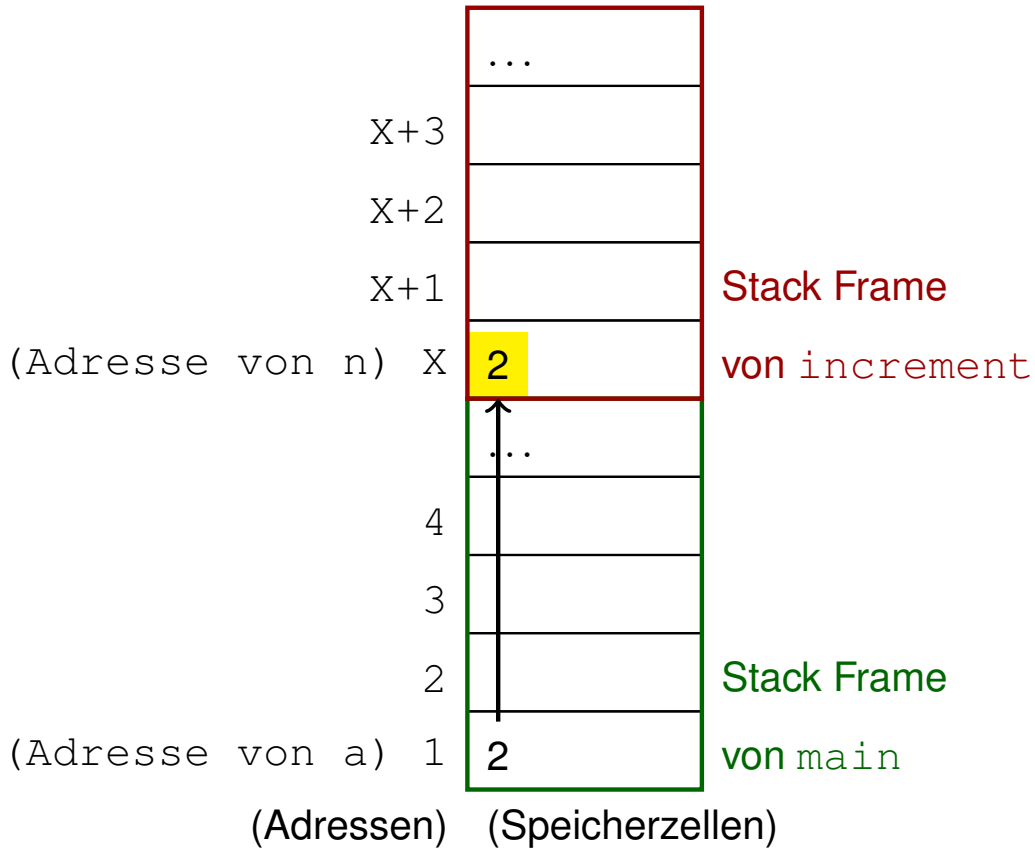
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für das Call-by-Value-Prinzip

In der Funktion wird mit Kopien gerechnet.

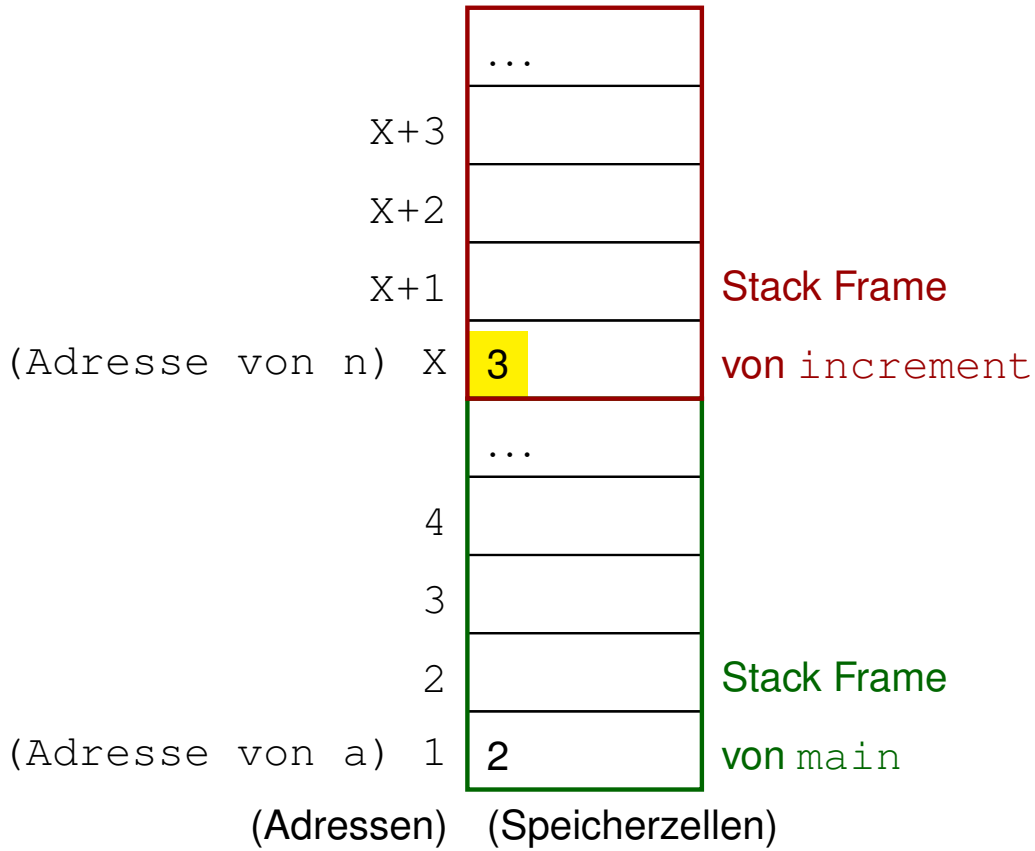
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für das Call-by-Value-Prinzip

In der Funktion wird mit Kopien gerechnet.

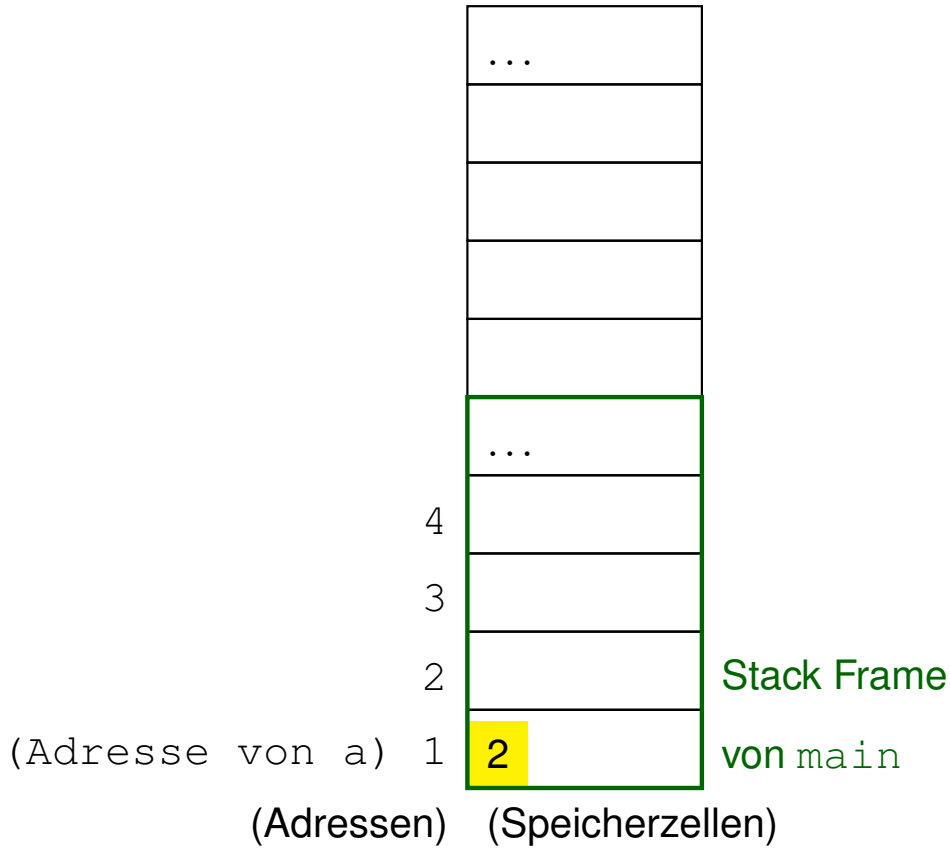
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für das Call-by-Value-Prinzip

In der Funktion wird mit Kopien gerechnet.

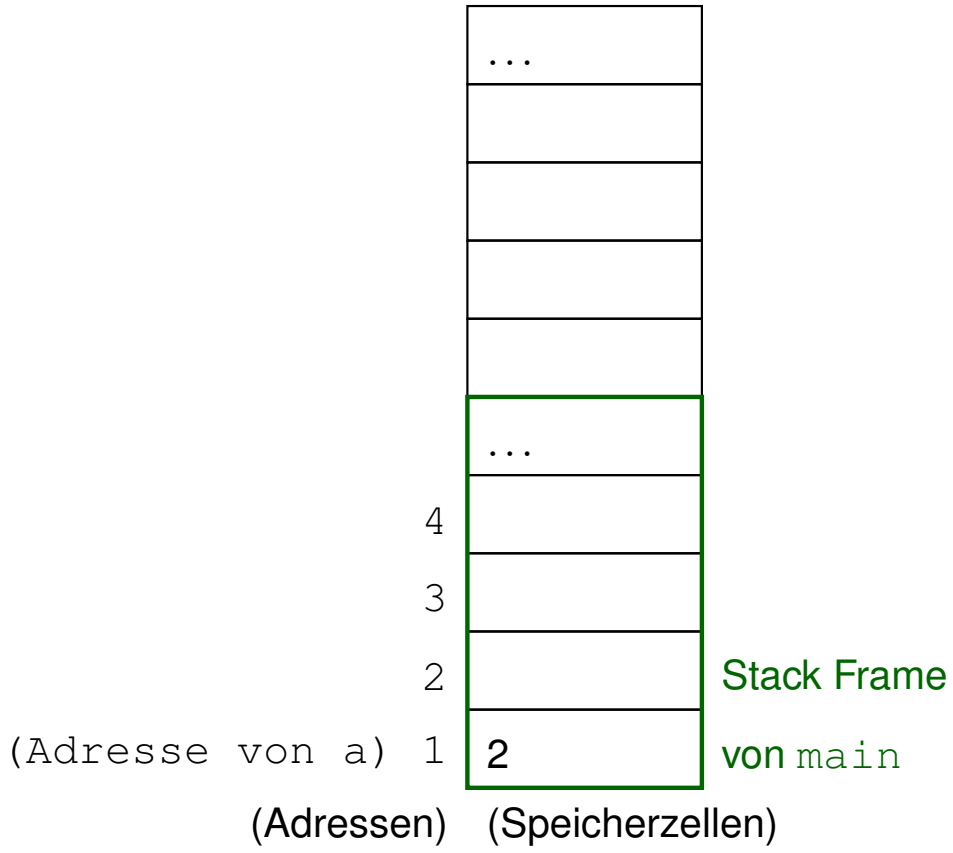
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für Zuweisung von Rückgabewerten

In einer Funktion berechnete Werte können zurückgegeben und an Variablen zugewiesen werden.

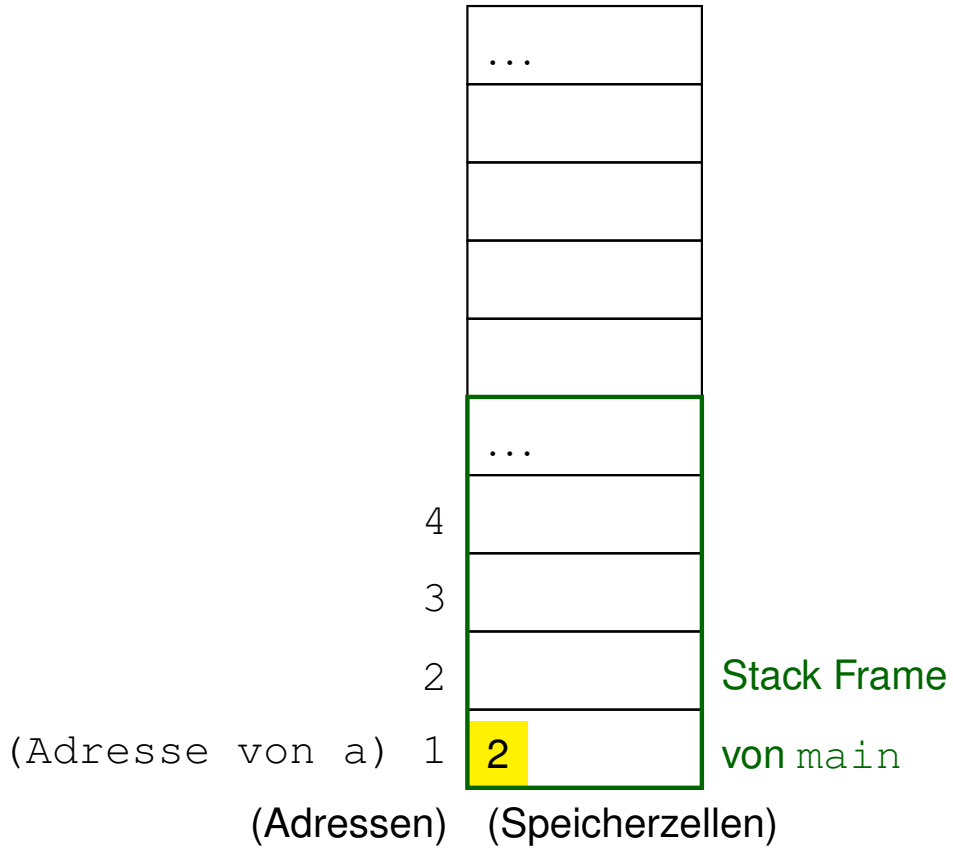
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    a = increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für Zuweisung von Rückgabewerten

In einer Funktion berechnete Werte können zurückgegeben und an Variablen zugewiesen werden.

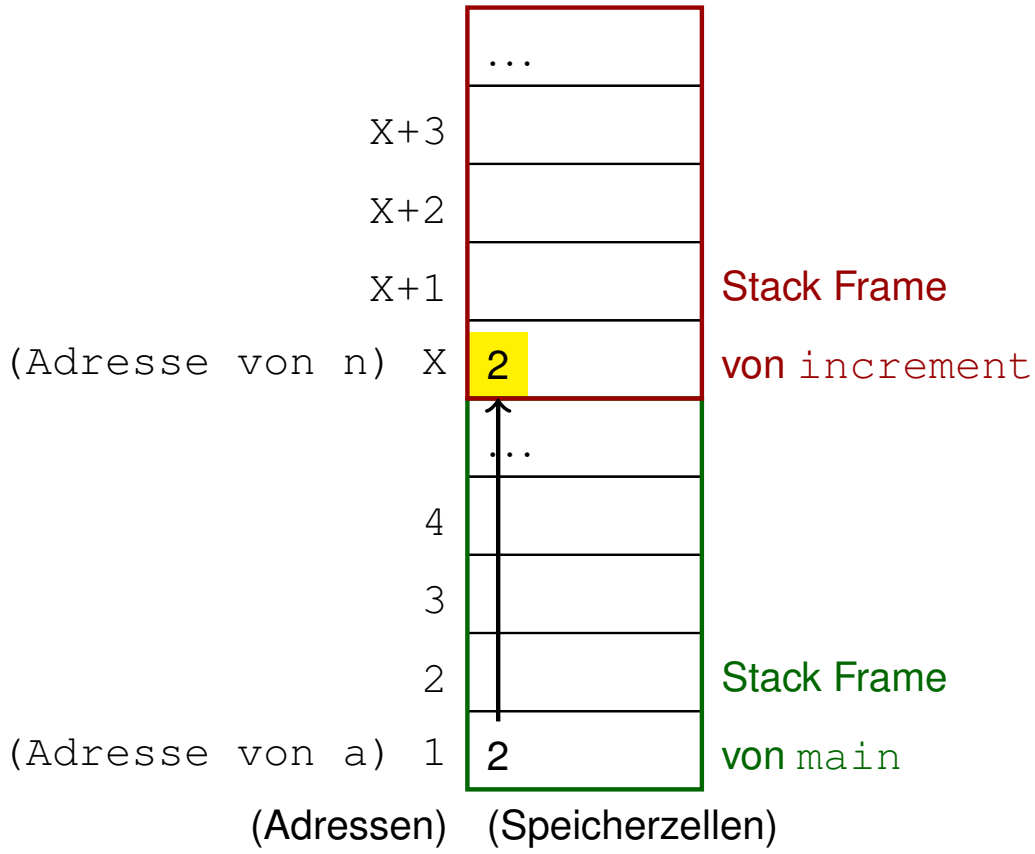
```
1  #include <stdio.h>

3  int increment(int n);

5  int main() {
6    int a = 2;
7    a = increment(a);
8    printf("%i", a);
9    return 0;
10 }

12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für Zuweisung von Rückgabewerten

In einer Funktion berechnete Werte können zurückgegeben und an Variablen zugewiesen werden.

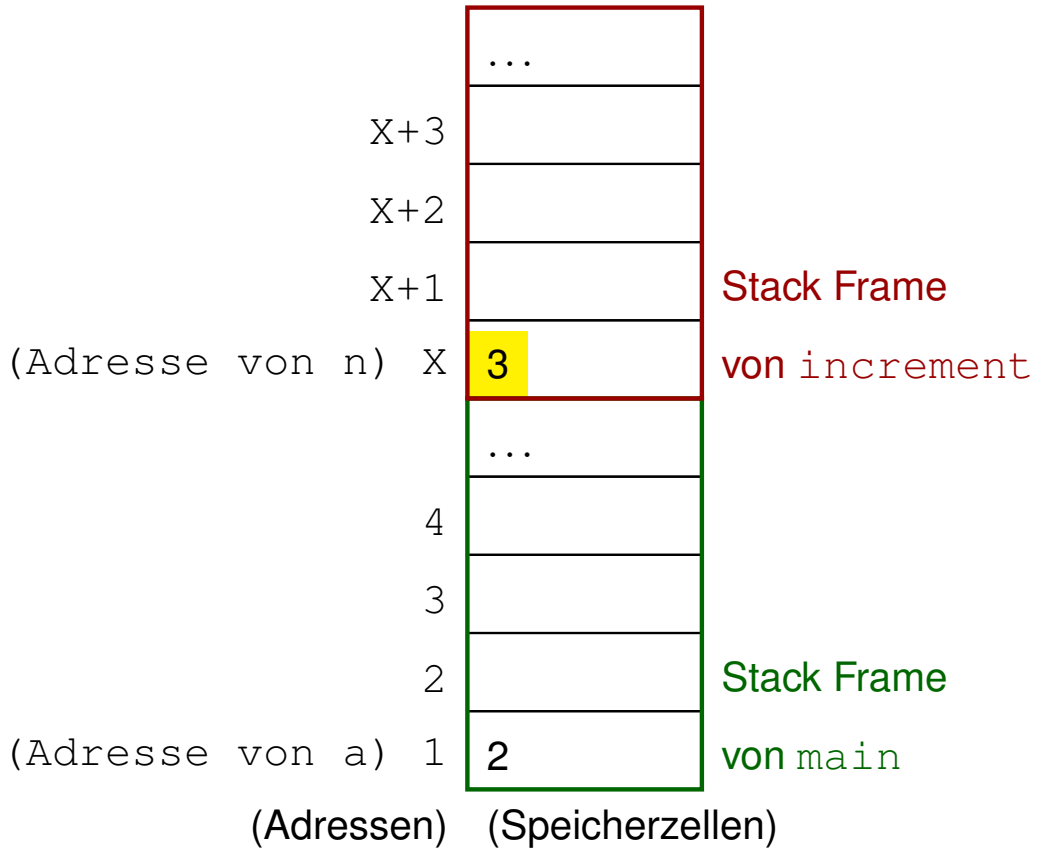
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    a = increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für Zuweisung von Rückgabewerten

In einer Funktion berechnete Werte können zurückgegeben und an Variablen zugewiesen werden.

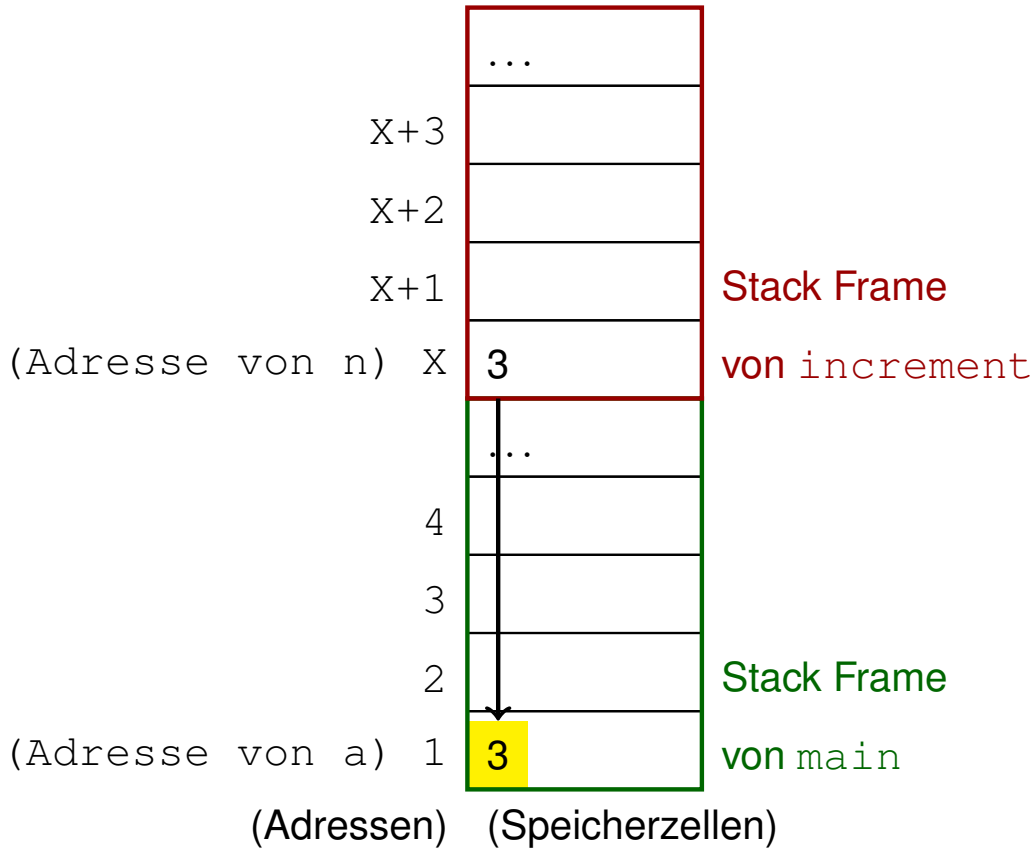
```
1   #include <stdio.h>
2
3   int increment(int n);
4
5   int main() {
6     int a = 2;
7     a = increment(a);
8     printf("%i", a);
9     return 0;
10  }
11
12  int increment(int n) {
13    ++n;
14    return n;
15  }
```



Beispiel für Zuweisung von Rückgabewerten

In einer Funktion berechnete Werte können zurückgegeben und an Variablen zugewiesen werden.

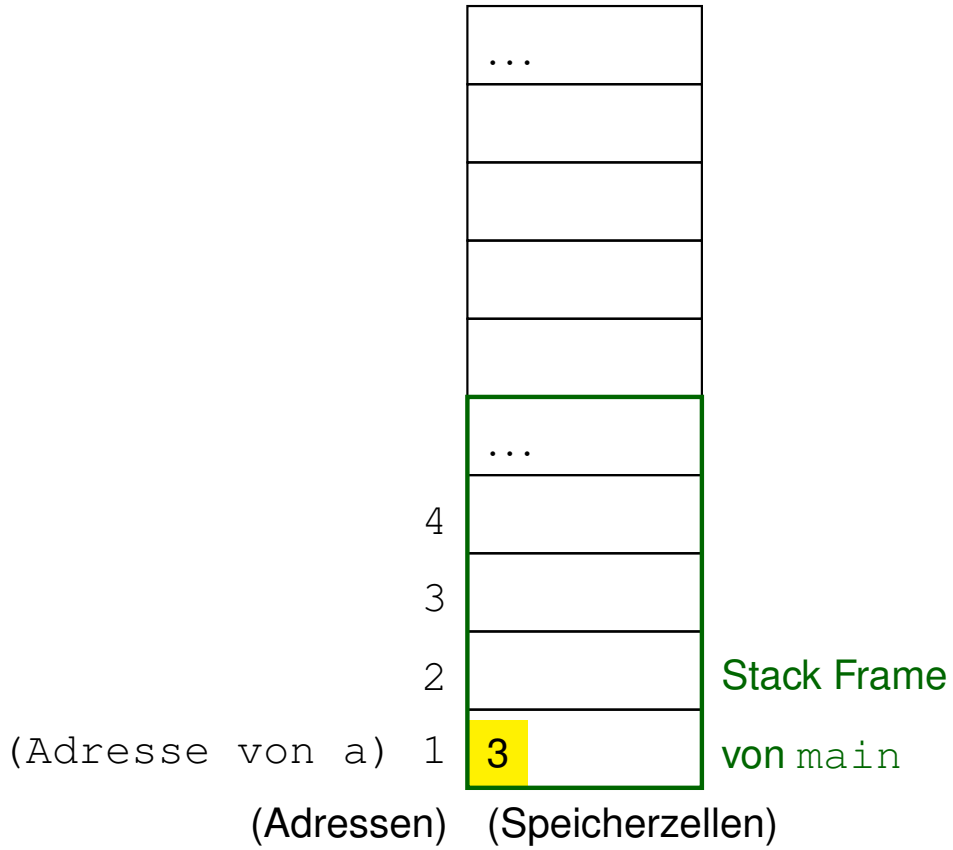
```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    a = increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Beispiel für Zuweisung von Rückgabewerten

In einer Funktion berechnete Werte können zurückgegeben und an Variablen zugewiesen werden.

```
1  #include <stdio.h>
2
3  int increment(int n);
4
5  int main() {
6    int a = 2;
7    a = increment(a);
8    printf("%i", a);
9    return 0;
10 }
11
12 int increment(int n) {
13   ++n;
14   return n;
15 }
```



Wie werden Daten im Rechner übertragen (transportiert)?

Daten, Befehle, Kontroll- und Statussignale werden über elektrische Leitungen zwischen den Rechner-Komponenten übertragen.
Übertragen werden Bitwerte:

- Bitwert 1: Stromfluss (hohe Spannung)
- Bitwert 0: kein Stromfluss (niedrige Spannung)

Möglichkeit 1: Spezielle Übertragungsleitungen zwischen je zwei Komponenten

- hohe Übertragungsgeschwindigkeit
- **ABER: viel zu viele Leitungen**

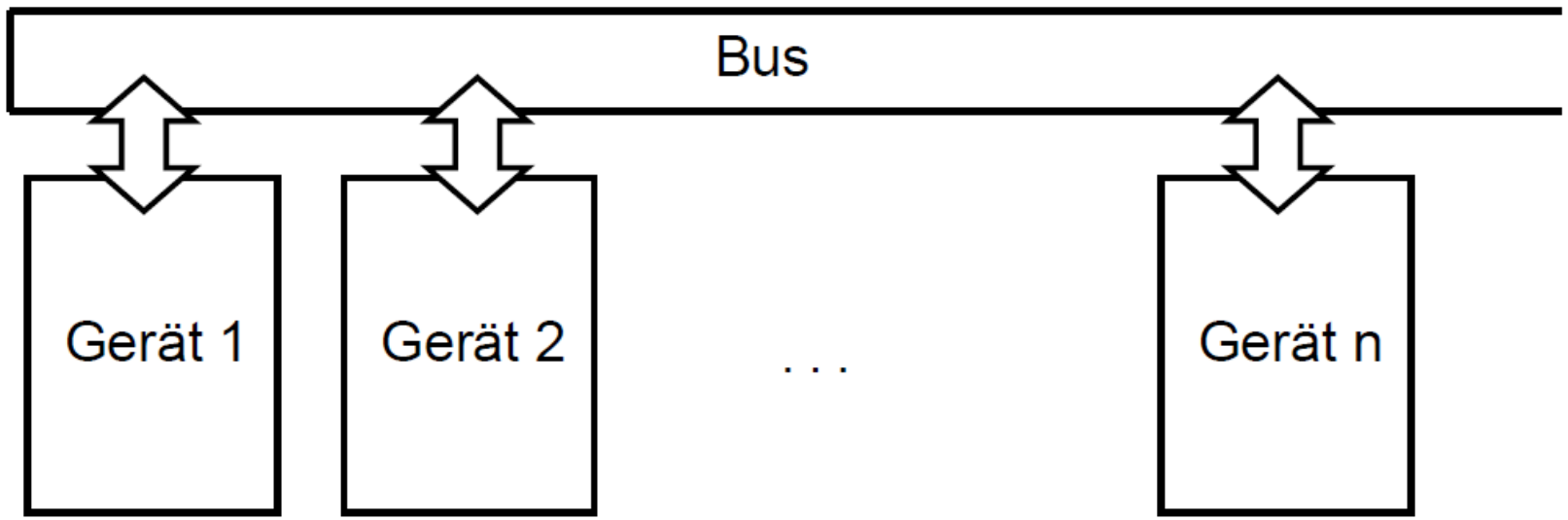
Möglichkeit 2: 'Datensammelwege', sogenannte Busse

- **alle** Komponenten sind an **einen** Bus angeschlossen
- jede Komponente holt sich vom Bus, was für sie bestimmt ist

Übersicht

Definition 2.20 (Bandbreite)

Ein Bus hat eine sog. **(Band-)Breite**. Das ist die Anzahl der parallel übertragbaren Bits (= Anzahl der parallelen Leitungen).



Teilbusse

Ein Bus besteht aus Teilbussen:

Datenbus

- Übertragung des Inhalts einer SZ
- Breite = Anzahl der Bits einer SZ

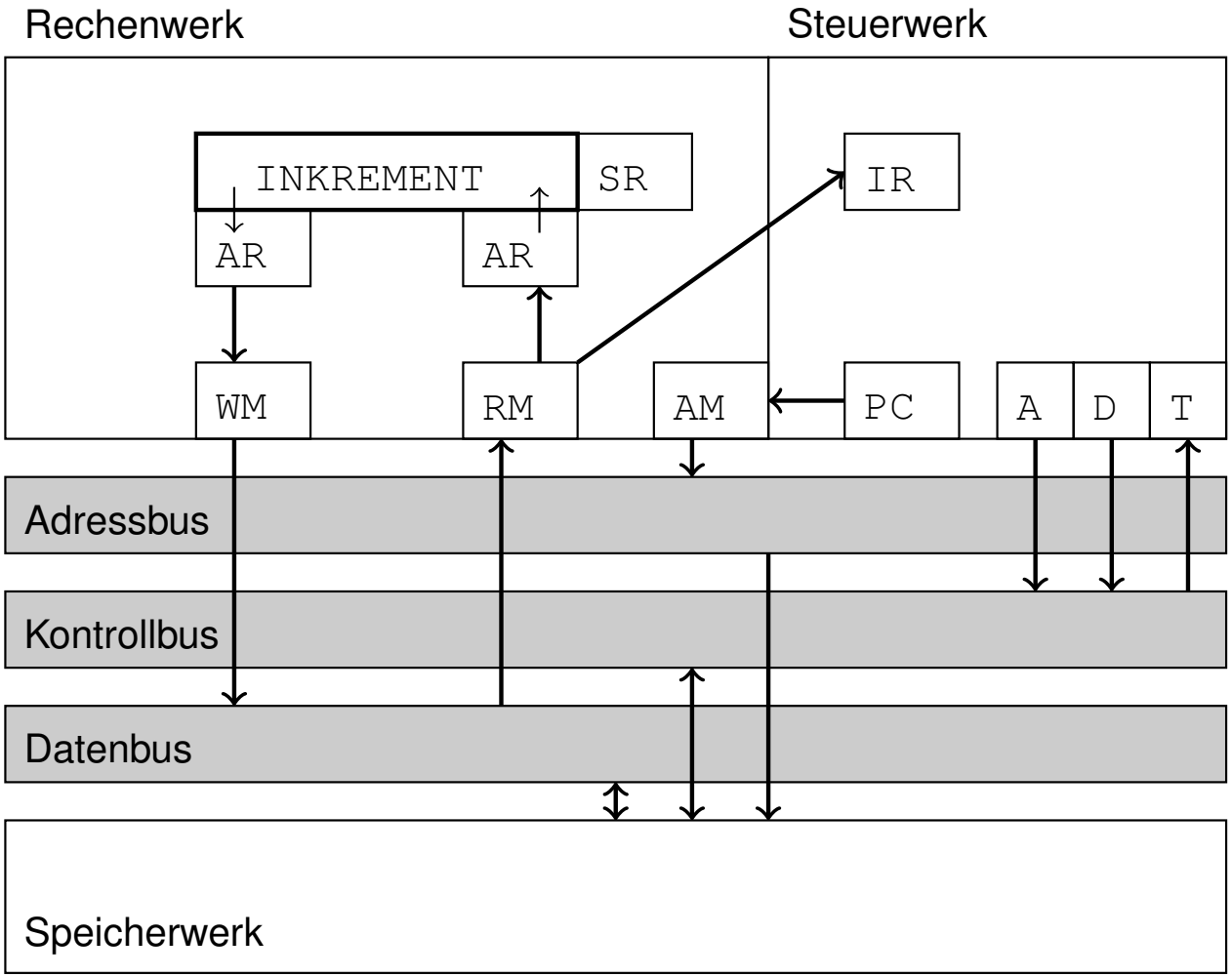
Adressbus

- Identifikation der Quell- und Ziel-Komponente von übertragenen Daten (Codiert als Bitfolge)

Kontrollbus

- Übertragung von Steuersignalen (Codiert als Bitfolge)

Kombinierte Gesamtübersicht (2)



Vorteile

Prinzip des minimalen Hardwareaufwandes

Nichts kann weggelassen werden (jede notwendige Komponente existiert genau einmal)

Prinzip des minimalen Speicheraufwandes

Unabhängig vom Inhalt einer SZ (Daten oder Programmbefehle):

- Einheitliche SZ-Struktur und -Größe
- Einheitlicher SZ-Zugriff

Bestmögliche Nutzung des Speichers unabhängig vom Umfang von Daten und Programmcode

Nachteile

Das Bussystem als **von-Neumann-Flaschenhals**

Durch den Bus ist alles sequentiell (nacheinander) zu transportieren: Steuersignale, Adressen, Daten, Befehle.
Zur Ausführung eines Befehls muss der Bus mehrmals nacheinander benutzt werden:

- Befehle aus Speicherwerk ins Steuerwerk holen
- Operanden aus Speicherwerk ins Rechenwerk holen
- Ergebnis vom Rechenwerk ins Speicherwerk übertragen
- Austausch von Steuersignalen zwischen Steuerwerk und Speicherwerk

Anfälligkeit für Computerviren

Programme können als Daten aufgefasst werden und umgekehrt

Verbesserung: Harvard-Architektur

Harvard-Architektur

Für Programme und Daten gibt es **getrennte Speicher**: Programm-Speicher und Daten-Speicher werden mit getrennten Bussystemen angesteuert.

- Geschwindigkeitsgewinn (zeitgleiche Verarbeitung von Befehlen und Daten)
- Geringe Anfälligkeit für Computerviren
- Größerer Hardwareaufwand: zusätzliche Speicher, Busse

Schnittstellen

- Eine **Schnittstelle** ist eine Konvention für eine Verbindung mit verschiedenen, aber gleichartigen, Bauteilen (z.B. Drucker, Monitore,...) verschiedener Hersteller
- Die Schnittstelle regelt **einheitlich** den Signal- und Datenaustausch bzgl. einer Hardware (HW)- oder Software (SW)-Komponente unabhängig von der Datenverarbeitung innerhalb der Komponente

Treiber

- Ein **Treiber** ist ein Anpassungsprogramm zur Ansteuerung einer SW- oder HW-Komponente (z.B. Druckertreiber, Maustreiber, ...)
- Ein Treiber verbirgt die speziellen Anforderungen eines herstellertypischen Geräts vor den höheren SW-Schichten und erlaubt die Benutzung der Komponente mit einheitlichen Befehlen (z.B. einheitlichen Druckbefehl an Drucker anpassen)
- Treiber müssen installiert werden oder sind im Betriebssystem integriert

Betriebssystem

Das **Betriebssystem** ist eine Software, um die Hardware nutzbar zu machen. Es hat folgende Aufgaben:

- **Prozessverwaltung**: Planung der Abarbeitungsreihenfolge zwischen mehreren Prozessen (Prozess-Scheduling)
- **Speicherverwaltung**: Speicherzuteilung für Prozesse, Zugriffsschutz, Freispeicherverwaltung
- **Dateiverwaltung**: Verzeichnissystem, Zuordnung von externen Speicherbereichen zu Dateien, Zuordnung von Dateiendungen zu Programmen

Prozess

Ein **Prozess** ist ein eigenständig ablaufendes Programm mit eigenem Speicherbereich.

Middleware

Die **Middleware** ist eine Softwareschicht, welche die Heterogenität der einer Anwendung zugrundeliegenden Systemplattform (Hardware, Netzwerke, Betriebssysteme, Programmiersprachen) "maskiert":

- Täuscht eine homogene Systemplattform vor (einheitliche Benutzung verschiedener System-Komponenten in einem Netzwerk)
- Macht heterogenes System programmierbar (vs. Betriebssystem: macht Hardware nutzbar)

Beispiel 2.21 (Middleware)

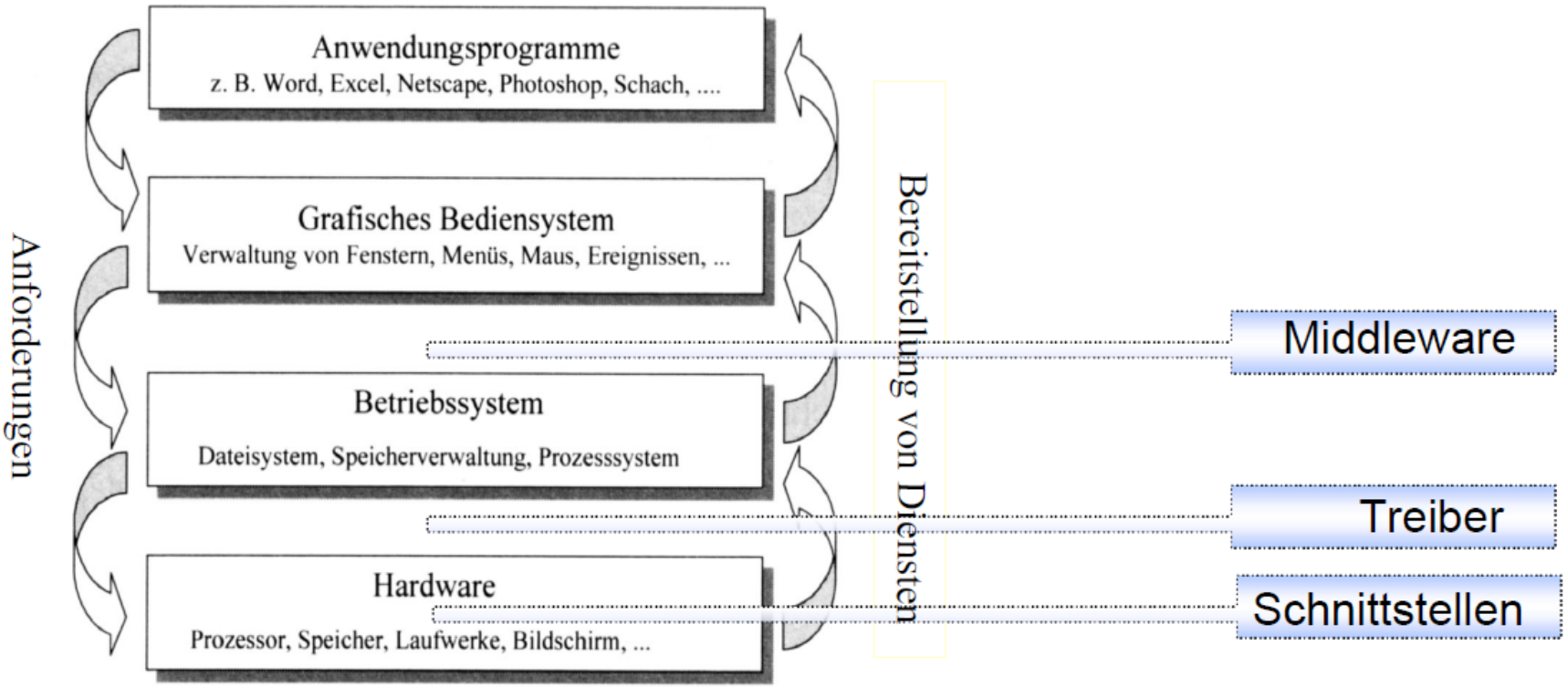
Managementsysteme für verteilte Datenbanken, E-Mail, Remote Login, File Transfer, ...

Grafisches Bediensystem

Ein **grafisches Bediensystem** ist eine Schnittstelle zur Anforderung von Leistungen des Betriebssystem (via Tastatur, Maus, etc.)

- im einfachsten Fall: Kommandozeile
- üblich für PCs: graphisch / fensterorientiert über ein sog. **Graphical User Interface (GUI)**:
  ursprünglich erfunden von Xerox (70er Jahre);
  zuerst aufgegriffen für Macintosh-Rechner (Apple)

Übersicht


Anwendungsprogramme
z. B. Word, Excel, Netscape, Photoshop, Schach, ...
Grafisches Bediensystem
Verwaltung von Fenstern, Menüs, Maus, Ereignissen, ...
Betriebssystem
Dateisystem, Speicherverwaltung, Prozesssystem
Hardware
Prozessor, Speicher, Laufwerke, Bildschirm, ...
Anforderungen
Bereitstellung von Diensten
Middleware
Treiber
Schnittstellen

Lesestoff für die Vor- und Nachbereitung

[Richter, Sandy, Stucky] Der Rechner als System
Kapitel 3.1 Von-Neumann-Rechner

[Gumm, Sommer] Einführung in die Informatik
Kapitel 1.1 Was ist Informatik
Kapitel 1.5 Hardware
Kapitel 1.6 Von der Hardware zum Betriebssystem
Kapitel 5.6 Von den Schaltgliedern zur CPU

Lesestoff für die Vor- und Nachbereitung

[Roland Hellmann] Rechnerarchitektur

Kapitel 1 Einleitung
Kapitel 2 Allgemeiner Aufbau eines Computersystems
Kapitel 3 Performance und Performanceverbesserung
Kapitel 4 Verbreitete Rechnerarchitektur
Kapitel 15 Maschinensprache
Kapitel 16 Steuerwerk
Kapitel 17.1 Mikroprogrammierung - Konzept
Kapitel 21.1 Arten von Speichermedien
Kapitel 21.2 Halbleiter-Speicher
Kapitel 21.3 Statisches und dynamisches RAM