Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 2 – Rechnerarchitektur und -organisation

Contents

2.1 Grundbegriffe

Programmierbarkeit
Ein Computer ist eine (wenn auch sehr komplexe) elektrische Maschine, die genau tut, was man ihr sagt (eingibt). Diese Eingaben können wir in vielen Art und Weisen vornehmen wie z.B. durch Interaktion über eine Steuerung. Die für uns interessanteste Eingabe hingegen sind Programme. Ein Programm ist eine vom Computer ausführbare Verarbeitungsvorschrift (Folge von Maschinenbefehlen / Anweisungen). Wir definieren somit die Eigenschaft Programmierbarkeit folgendermaßen:

> Definition: 2.0 Programmierbarkeit
>
> Wenn Programme (austauschbar) gespeichert werden können, und nicht nur eine endliche Anzahl fest verschalteter Programme existiert, ist eine Maschine **programmierbar**.

Genauer lässt sich an dieser Stelle noch der Begriff des Maschinenbefehls definieren.

> Definition: 2.2 Maschinenbefehl (vereinfacht)
>
> Ein **Maschinenbefehl** ist eine elementare Operation, die
>
> - ein Rechner ausführen kann, und
> - vom Programmierer verwendbar ist

> Beispiel:
>
> Ein Beispiel für einen Maschinenbefehl wäre dabei die Addition von 2 Werten.

Im weiteren Verlauf werden verschiedene Maschinenbefehle vorgestellt und erklärt (siehe Kapitel 2.5). Auch werden einige Beispiele angegeben, also Programme, die einfache Abfolgen von Maschinenbefehlen beinhalten.

Schlussendlich verstehen wir für einen Rechner diese Eigenschaften als Teil seiner Architektur, welche beschreibt inwiefern der Rechner mit diesen Befehlen programmierbar ist.

> Definition: 2.1 Rechnerarchitektur
>
> - Die Architektur eines Rechner 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**.

Unter den Begriffen Rechnerstruktur, Rechnerorganisation und Rechnerrealisierung versteht man dabei folgendes:

- 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

In diesem Kapitel werden die Grundstruktur und Funktionesweisen heutiger Rechner gelehrt, wohlgemerkt **ohne technische Details**. D.h. nur so viele Details werden präsentiert, dass plausibel wird, wie eine erfolgreiche Funktionsweise ermöglicht wird.
Insbesondere wird die Architektur des *Von-Neumann-Rechners* als Grundkonzept in theoretischer und technischer Sicht verwendet, denn noch heutige Rechner basieren im Wesentlichen 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*).

2.2 Geschichtliches

siehe Foliensatz.

2.3 Von-Neumann-Rechner

Empfunden nach den Vorschlägen und Entwicklungen des Forschers John von Neumann verbindet der *von-Neumann Rechner* verschiedene Konzepte für Rechenmaschinen. Diese Konzepte betreffen die verschiedenen internen Komponenten, welche in nachfolgenden Kapiteln im einzelnen behandelt werden.

Von Neumann Rechner – Komponenten

Der Von-Neumann-Rechner besteht, wie schon angesprochen, aus verschiedenen Komponenten. Jeder dieser Bausteine existiert im Rechner nur einmal und erfüllt einen vordefinierten Zweck:

- 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,...)

Von Neumann Rechner – Übersicht
(nach Burks, Goldstine und von Neumann, 1946)

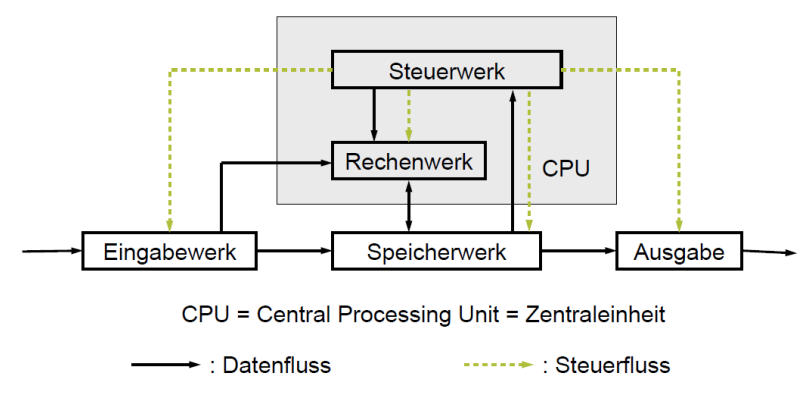


Figure 1: Eine abstrakte Ansicht des Von-Neumann-Rechners. Zu sehen sind die verschiedenen Komponenten und wie sie zusammenspielen.

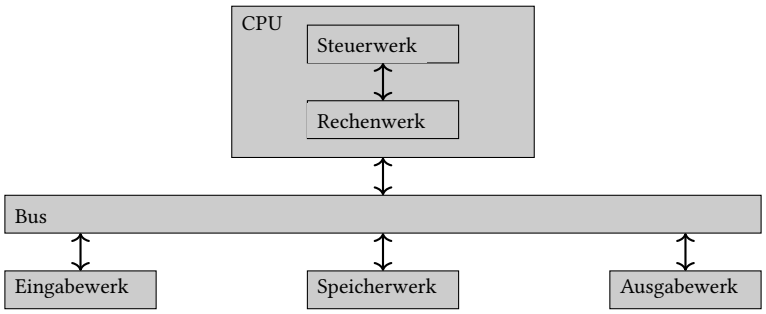


Figure 2: Eine abstraktere Ansicht, wie die einzelnen Komponenten miteinander kommunizieren. In der CPU sind Steuer- und Rechenwerk direkt miteinander verbunden und benötigen keine langen Kommunikationswege. Eingabe-, Speicher- und Ausgabewerk sind über einen *Bus* miteinander verbunden, worüber auch die Kommunikation läuft.

- CPU = Central Processing Unit; führt alle Befehle im Rechner aus.
- Bus: Datenaustausch zwischen den Komponenten (Steuerwerk, Rechenwerk, Eingabewerk, Ausgabewerk, Speicherwerk). Mehr zum Bus im Kapitel 2.8

2.4 Speicherwerk

Das Speicherwerk dient, wie der Name impliziert, dem Speichern von Daten, um Programme auszuführen. Bei der von-Neumann Architektur liegen die Programme und dazugehörigen Daten gemeinsam im Speicherwerk.

2.4.1 Speichereinheiten

Das Speichern in Computern wird realisiert durch Ladezustände von Speicherelementen. Diese können nur die Zustände *geladen* oder *ungeladen* annehmen bzw. 1 oder 0. Man spricht dann von einem *Bit*.

> 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

Mit 8 Bit lassen sich die gängigen Zeichen darstellen. 8 Bit werden daher zusammengefasst als ein *Byte*.

> 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

Speichereinheiten
Moderne Computer besitzen allerdings mehr als nur ein paar Bytes (oftmals über hundert Millionen), weshalb Angaben in „nur Bytes“ sehr unübersichtlich sein können. Deswegen ist oft auch die Sprache von *Kilo-*, *Mega-*, *Giga-* oder *Tera*bytes.

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

2.4.2 Speicherstruktur

Die Struktur bzw. die Art, wie Daten abgelegt werde, ist **unabhängig** von den zu bearbeitenden Programmen. Das bedeutet, dass bspw. eine Textdatei und ein Computerspiel auf die selbe Art und Weise gespeichert werden.
Auch ist zu beachten, dass Daten und Programme im selben Speicher abgelegt werden.

> 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

Jede SZ besteht dabei aus einer festen Anzahl an k Bits und dient der Speicherung eines Zeichens, einer Zahl oder eines (Programm-)Befehls. Typischerweise ist k eine Potenzzahl von 2

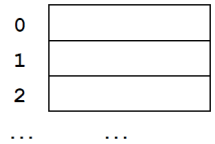


Figure 3: Jede SZ hat eine eindeutige Adresse. Im diesen Bild lauten die Adressen: 0, 1, 2, ...

($k \in \{8, 16, 32, 64, ...\}$). Auch besitzen Speicherzellen eindeutige Adressen, womit sie angesprochen werden können. Diese sind in der Informatik 1 der Einfachheit halber fortlaufend durchnummeriert, ggf. in Kombination mit einem Buchstaben (siehe Kapitel 2.7.2). In „echten"Computern und Rechnern werden Adressen anders dargestellt. Details folgen in der Vorlesung *Systemnahe Informatik*.
Daten werden in Speicherzellen gespeichert. Da Speicherzellen aus k Bits bestehen, bedeutet das auch, dass Buchstaben, Zahlen und Befehle in diesen Speicherzellen gespeichert sind und somit aus Folgen von 0en und 1en dargestellt (kodiert) sind – Details dazu später.

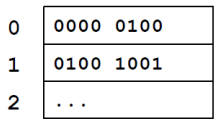


Figure 4: SZ mit 1 Byte Speicherplatz (Erinnerung von oben: 1 Byte entspricht 8 Bit – also 8 aufeinanderfolgende 0en und 1en.)

2.4.3 Realisierung des Speicherwerks

Zur **Realisierung des Speicherwerks** sollten folgende Anforderungen erfüllt werden:

- schneller Zugriff
- hohe Kapazität
- geringe Kosten

Jedoch ergibt sich dann die Problemstellung, dass alle drei Kriterien nicht gleichzeitig erfüllt werden können:

- Große Kapazität + geringe Kosten = Langsamer Zugriff
- Hohe Kosten + kleine Kapazität = Schneller Zugriff

Als Lösung wurde die **Speicherhierarchie** eingeführt: In einem PC gibt es mehrere Arten von Speichern, die abgestimmt zusammenarbeiten. Man besitzt sowohl schnellen Speicher mit geringer Kapazität, als auch langsamen Speicher mit großer Kapazität.

2.4.4 Komponenten der Speicherhierarchie

Wie aus der Abbildung 5 zu entnehmen, haben sich 3 große Entitäten in der Speicherhierarchie etabliert (ausgenommen von großen persistenten Speichern wie Festplatten).

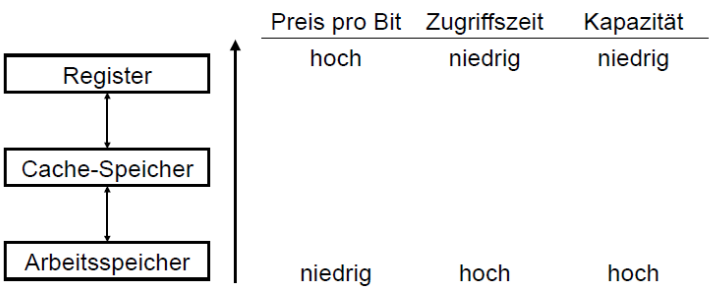


Figure 5: Übersicht Speicherhierarchie (vereinfacht). Es haben sich drei große Entitäten in der Speicherhierarchie etabliert: Register, Cache-Speicher und Arbeitsspeicher.

- Register: Sind unmittelbar der CPU zugeordnet
- Cache: Bindeglied zwischen CPU und Arbeitsspeicher, für häufig/voraussichtlich benutzte Befehle/Daten verwendet.
- Arbeitsspeicher: bezeichnet als RAM (Random Access Memory): enthält Daten / Programme

2.4.4.1 RAM – Random Access Memory

> Definition: 2.8 RAM (Random Access Memory)
>
> 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

Es gibt zwei Arten, RAM technisch zu realisieren: SRAM (statischer RAM) und DRAM (dynamischer RAM). Aufgrund ihrer Eigenschaften werden diese an unterschiedlichen Stellen der Speicherhierarchie eingesetzt.

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

2.4.4.2 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)

- ROM
- Magnetische Platten (bspw. Festplatten) und Bänder
- Flash-Speicher (bspw. SSD, USB-Sticks, ...)
- Optische Speicher (DVD, CD, Blu-Ray Discs, ...)

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 bedeutet, dass der Speicher im ROM nicht verändert und nicht geschrieben, sondern nur gelesen werden kann. Es gibt auch Varianten, die gelöscht und neu beschrieben werden können, z.B. EEPROMs (Electronically Erasable Programmable Read-Only Memory). Dabei muss aber das gesamte ROM gelöscht und dann neu beschrieben werden.

Das erwähnte BIOS wird zum Booten (Hochfahren, Starten) des Rechners verwendet, und liest dafür Daten aus dem ROM.

2.5 Rechenwerk

Das Rechenwerk dient der Ausführung von Maschinenbefehlen und ist ein elementarer Bestandteil des Von-Neumann-Rechners.

2.5.1 Komponenten des Rechenwerks

Das Rechenwerk besteht aus sogenannten **Funktionseinheiten**. Jede davon realisiert einen **Maschinenbefehl**. Diese Befehle setzen elementare Abarbeitungsschritte um, aus denen ein Programm besteht. Diese werden auch als *Assembler-Befehle* bezeichnet. Funktionseinheiten sind als Hardware umgesetzt. Das heißt, es gibt keinen „Code “, der Maschinen-Befehle ausführt. Stattdessen handelt es sich um elektronische Schaltungen, die mit Transistoren u.ä. realisiert sind[1].

In dieser Vorlesung würde eine Programmierung mithilfe von Assembler-Befehlen den Rahmen sprengen, weshalb erst in *Systemnaher Informatik* mit echten Maschinen-Befehlen gearbeitet wird. Um trotzdem ein Gefühl für Funktionseinheiten und Maschinenbefehle zu bekommen, definieren wir einfache eigene Befehle. Hierzu betrachten wir Funktionseinheiten für Befehle wie Addition oder Subtraktion, Inkrement oder Dekrement (Erhöhen oder Erniedrigen eines Werts um 1), Sprünge und Rückgabe (wie return in C). Wie derartige Maschinenbefehle aussehen könnten, ist auf Seite 17 zu sehen.

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

Im Rechenwerk gibt es eine Vielzahl von reservierten Registern zur Steuerung von Operationen und Kommunikation mit dem Steuer- und Speicherwerk. Mithilfe dieser Register wird eine erfolgreiche Abarbeitung einer Aufgabe ermöglicht.
Man unterteilt diese Register in folgende Kategorien:

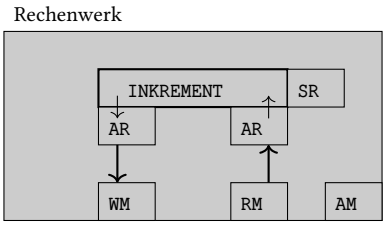
- Für die Eingabe- und Ausgabedaten einer Operation:
  - Akkumulator-Register: Zu jeder Funktionseinheit gehören ein oder mehrere **Akkumulator-Register (AR)** zur Speicherung von Operanden und Ergebnissen der Operation
- Zur Kommunikation mit dem Steuerwerk:
  - Status-Register: Zu jeder Funktionseinheit gehört ein **Status-Register (SR)** zur Speicherung des Zustands der Funktionseinheit
- Zur Kommunikation mit dem Speicherwerk:

[1] Details hierzu in der Master-Vorlesung *Hardware-Entwurf* oder *Prozessorbau*

- 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` (Funktionseinheit, die eine Zahl um 1 erhöht).

Beispiel:



- Es soll ein Wert erhöht werden, der im Speicherwerk an Adresse `A` hinterlegt ist.
- Dazu wird die Adresse `A` nach `AM` (Adress-Memory-Port) geschrieben.
- Der bisherige Wert, der an der Adresse gespeichert ist, die in `AM` hinterlegt ist, wird über `RM` (Read-Memory-Port) nach `AR` (Eingabe-Akkumulator-Register) geschrieben.
- Die Funktionseinheit führt die Operation `INKREMENT` aus.
- Das Ergebnis wird über `AR` (Ausgabe-Akkumulator-Register) nach `WM` (Write-Memory-Port) geschrieben.
- Der Inhalt von `WM` wird im Speicherwerk an der Adresse ersetzt, die in `AM` hinterlegt ist (Adresse `A`).

2.6 Steuerwerk

Das Steuerwerk ist ein weiterer, wichtiger Bestandteil des Von-Neumann-Rechners. Es regelt die Steuerung eines Programmablaufs. Man könnte also sagen, dass *das Steuerwerk dem Rechenwerk sagt, was es zu tun hat.*

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 (vgl. Beispiel 2.5.1: Die ersten drei Schritte)
- (V) Verarbeitung: Ausführung der Operation durch eine Funktionseinheit (vgl. Beispiel 2.5.1: Ausführen von `INKREMENT`)
- (A) Ausgabe: Ergebnis-Daten von AR über WM in das Speicherwerk (Adresse steht in AM) schreiben (vgl. Beispiel 2.5.1: Die letzten zwei Schritte)

2.6.1 Register im Steuerwerk

Auch das Steuerwerk besitzt viele Register, die der Steuerung und dem Ablauf eines Programmes dienen. Man unterscheidet hierbei zwischen folgenden Registerbefehlen:

- 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: Sprungbefehl: Bei einem Sprungbefehl wird der PC mit dem Sprungziel ersetzt (d.h. darin befindet sich die Adresse der Speicherzelle, an die gesprungen wird).
  - Fall 2: Kein Sprungbefehl: Es liegt kein Sprungbefehl vor, weshalb die Befehle „von oben nach unten “ abgearbeitet werden. Der PC zeigt damit automatisch auf den nächsten Befehl.

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

Beispiel:

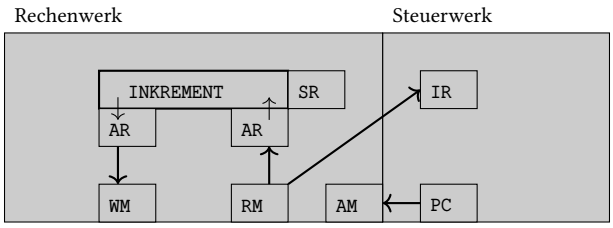


Figure 6: Hier wird nochmal das Beispiel 2.5.1 erweitert. Links ist das bereits bekannte Bild des Rechenwerkes zu sehen. Rechts davon wurde es um das Steuerwerk erweitert.

- 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), AM wird dabei mit der Adresse des Datenwerts überschrieben, der erhöht werden soll
- Die Adresse in `PC` wird aktualisiert

2.6.2 Fetch/Decode/Execute/Write-Back

Das EVA-Prinzip steuert den Ablauf eines einzelnen Befehls. Dafür wird nun der Fetch/Decode/Execute/Write-Back Zyklus betrachtet, nach dem ein Befehl abgearbeitet wird (und in welchem Schritt welcher EVA-Schritt ausgeführt wird). Dabei wird auch berücksichtigt, „woher der Befehl kommt".

1. Holen des nächsten Befehls (FETCH)
   - Befehl gemäß PC adressieren und aus dem Speicherwerk über RM nach IR laden
   - Befehlszähler PC aktualisieren (wie in 2.6.1 beschrieben wurde)
2. Entschlüsseln des Befehls (DECODE)
   - Erkennen der Befehlsart (Zuordnung zu einer Funktionseinheit)
   - Zerlegung in Bestandteile (Operation und Operanden[2]
   - (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

Beispiel:

Beispiel vom Abarbeitungszyklus: (Zahlen in `[]`-Klammern bezeichnen Adressen)
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

[2] bspw. Inkrement: Operation: Erhöhe Wert um 1; Operand: um 1 zu erhöhender Wert

3. **Execute**:
   (V) Führe Inkrement-Operation aus (`17+1 = 18`)
4. **Write-Back**:
   (A) Schreibe Ergebnis (`18`) von AR über WM nach SZ `[500]`

2.6.3 Steuersignale

Lese- und Schreibvorgänge zwischen Rechenwerk und Speicherwerk werden mit Hilfe von Steuersignalen zwischen Steuerwerk und Speicherwerk koordiniert. Man unterscheidet dabei zwischen *Adress Strobe*, *Direction* und *Data Transfer Acknowledge*:

- 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.

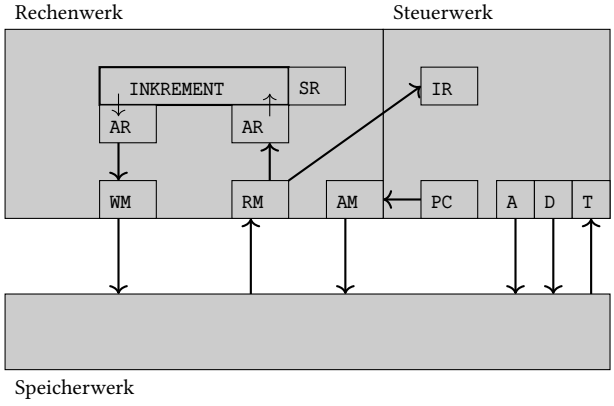


Figure 7: Die bekannte Darstellung vom Rechenwerk und Steuerwerk aus Abbildung 6. Diese wurde nun um das Speicherwerk und die Kommunikation der Module mit dem Speicherwerk ergänzt.

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: 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`

Das Steuerwerk ist getaktet

Alle Vorgänge in einem Prozessor laufen **getaktet** ab.
Moderne Prozessoren überlappen dabei die Phasen Fetch/Decode/Execute/Write-Back, so dass jeden Takt ein Ergebnis zurückgeschrieben wird (Write-Back), der nächste Befehl verarbeitet wird (Execute), ein weiterer Befehl decodiert wird (Decode) und ein neuer geholt wird (Fetch). Details über dieses *Pipelining* folgen in der *Systemnahen Informatik*.

- 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): $1\text{ms} = 10^{-3}\text{s}$
  Mikrosekunde (µs): $1\mu\text{s} = 10^{-6}\text{s}$
  Nanosekunde (ns): $1\text{ns} = 10^{-9}\text{s}$
- Takteinheiten:
  Kilohertz (KHz): $1\text{KHz} = 10^{3}\text{Hz}$
  Megahertz (MHz): $1\text{MHz} = 10^{6}\text{Hz}$
  Gigahertz (GHz): $1\text{GHz} = 10^{9}\text{Hz}$

2.7 Maschinenprogramme

2.7.1 Was passiert bei Start eines Maschinenprogramms?

Startet ein Maschinenprogramm, wird es anfangs in den Arbeitsspeicher geladen. Die Befehle befinden sich dann in aufeinanderfolgenden Speicherzellen, dem sogenannten **Programmteil**. Gleichzeitig wird der Programm Counter auf die Adresse des ersten Befehls vom Programm gesetzt. Zusätzlich werden auch noch drei weitere, zusammenhängende Speicherbereiche zugeordnet, damit das Programm ausgeführt werden kann:

- **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).

Die eben erwähnten Variablen sind unterteilt in dynamisch und automatisch verwaltete Variablen. Grund dafür sind verschiedene Anwendungsfälle und Situationen, bei denen die beiden genutzt werden:

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

2.7.2 Annahmen und Notationen für diese Vorlesung

1. 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.
2. 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.

> Beispiel:
>
> Ein Beispiel für die Notationen der Speicheradresse befindet sich weiter unten im Beispiel 2.7.3. Dort ist beispielsweise P1 eine Adresse des Programmteiles – und dient der Initialisierung einer Speicherzelle im Stack S2.

2.7.3 Maschinencode zu einer C-Funktion

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

Definition: Maschinenbefehle in der Vorlesung

- `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

Beispiel:

Zu sehen ist ein Beispiel vom Maschinencode (links) und dazugehörigem C-Code (rechts).
Es gilt:

- 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	

```
1  int exp2(int n) {
2    int e = 0;
3    e = e + 1;
4    while (n > 0) {
5      e = e + e;
6      n = n - 1;
7    }
8    return e;
9  }
```

Aufruf: `exp2(2)`

Schritt für Schritt bedeutet das nun:

1. INIT S2: S2 wird mit dem Wert 0 initialisiert. Das ist gleichbedeutend mit der 2. Programmzeile.

2. INKREMENT S2: Der Wert in S2 wird nun um eins erhöht (vgl. Zeile 3)
3. SPRUNG0 P7, S1: Nun wird zuerst der Wert aus S1 genommen und es wird verglichen, ob S1 gleich der 0 ist. Falls ja, wird gesprungen und man geht zum Programmteil P7. Sollte NICHT gesprungen werden, wird der folgende Programmteil ausgeführt (ab P4). Entspricht der while-Schleife in Zeile 4 – die Bedingung ist nur umgekehrt (wenn `n == 0`, überspringe die Schleife).
4. ADD S2, S2 Man addiert den Wert aus S2 mit dem Wert aus S2 (also wird S2 mit sich selber addiert) und der daraus resultierende Wert wird wieder in die Speicherzelle S2 geschrieben (vgl. Codezeile 5)
5. DEKREMENT S1: Nun wird der Wert in S1 um 1 verringert (vgl. Codezeile 6)
6. SPRUNG P3: Es wird nun wieder zur Programmzeile P3 gesprungen. Hier gilt zu beachten: Man springt zu einem Sprungbefehl. Das heißt, man geht zurück zum Code, vergleicht hier Werte und springt gegebenenfalls wieder weg. Das Konzept ist gleichbedeutend mit einer Schleife – genauer: einer while-Schleife, wie sie in Zeile 4 zu sehen ist. Auch dort kommt es zum Vergleich von Wert n. Ist dieser ungleich Null, wird mit Zeile 5 weitergemacht. Ist n jedoch gleich 0, kommt es auch hier „zum Sprung“. Man springt nämlich aus der Schleife heraus und führt die Codezeile nach der Schleife aus, was dem return-Befehl entspricht.
   Durch solche Sprungbefehle bzw. Verknüpfungen von Sprungbefehlen lassen sich Wiederholungen im Programmcode einbauen.
7. RÜCKGABE S2: Analog zum Programmcode wird nun der Wert S2 zurückgegeben, sollte ein Sprung in P3 ausgeführt worden sein.

Beispiel:

- 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	

```
1  int is_equal(int a, int
       b)
2  {
3      int c = a - b;
4      if (c == 0) {
5          return 1;
6      }
7      return 0;
8  }
```

Aufruf: `is_equal(5, 5)`

Schritt für Schritt bedeutet das nun:

1. `INIT S3`: In Zeile 3 im Code steht:

   ```
   int c = a - b;
   ```

   Das bedeutet, es geschehen hier gleich mehrere Dinge auf einmal. Zum einen wird eine Variable `c` erstellt. Dieses bekommt dann nun den Wert `a - b` zugewiesen. Um dies mit Maschinenbefehlen zu realisieren, müssen mehrere Dinge geschehen.
   Zum einen muss ersteinmal die Variable `c` initialisiert werden. Dies geschieht durch den Befehl in Adresse **P1**.

2. `ADD S3, S1`: Es ist mit unseren Maschinenbefehl nicht möglich, zwei Werte voneinander abzuziehen und gleichzeitig einer anderen Speicheradresse zuzuweisen. Deswegen die Zeile 3 im Code weiter aufgeteilt werden.
   Zuerst weißt man der Variable `c` den Wert von `a` zu. Dies geschieht mit dem Additionsbefehl. Man addiert den Wert aus `S1` (also den Wert in `a`) mit dem Wert aus `S3` und speichert diesen Wert auch in `S3`. Da `S3` vorhin erst initialisiert wurde, ist in `S3` der Wert `0` gespeichert.

3. `SUB S3, S2`: Hier wird nun der letzte Befehl für Zeile 3 ausgeführt. Man subtrahiert hier nun `a` um `b`.

4. `SPRUNG0 P6, S3`: Mit dieser Zeile wird die `if`-Bedingung überprüft. Sollte der Wert in `S3` gleich `0` sein, dann wird zur Adresse `P6` gesprungen. Sollte der Wert `!= 0` sein, dann geht man zur nächsten, darauf folgenden Adresse (also zu von `P4` nach `P5`)

5. `RÜCKGABE D2`: Wird nun nicht gesprungen (also `S3 != 0`), dann gibt man den Wert aus Speicherzelle `D2` zurück, also den Wert 1 (vgl. Zeile 5).

6. `RÜCKGABE D1`: Wird doch gesprungen (also `S3 == 0`), dann gibt man den Wert aus Speicherzelle `D1` zurück, also Wert 0 (vgl. Zeile 7).

Datenverwaltung im Arbeitsspeicher
Der Arbeitsspeicher dient auch der Speicherung von Daten, Programmteilen, etc. Diese lassen sich wie folgt aufgliedern:

- **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

Details dazu folgen in späteren Kapiteln, wenn wir diese Konstrukte in C betrachten.

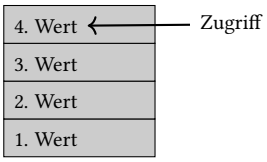
2.7.4 Der Stack

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):



Beispiel:

Das Konzept vom Stack kann anfangs sehr verwirrend sein. Hierzu lässt sich gut das folgende Beispiel zur besseren Veranschaulichung beschreiben:
Der Stack lässt sich wie eine alte Keksdose veranschaulichen. Man kann die Kekse aufeinander in die Dose hineinlegen und beliebig stapeln. Will man jedoch einen Keks herausnehmen, kann man nicht irgendeinen beliebigen Keks essen. Man kommt immer nur an den obersten Keks heran.
Das bedeutet, dass der zuletzt reingelegte Keks auch als erstes gegessen wird – und hat somit eine Last in - First out Keksdose.

Der Stack und lokale Variablen
Der Stack spielt eine wichtige Rolle für lokale Variablen – also Variablen, die während dem Ausführen von Programmen erstellt wurden. Es werden nämlich alle lokale Variablen eines Programmes im Stack verwaltet.

Betritt man einen Anweisungsblock, wird zuerst „oben“ auf dem Stack ein Bereich für die Aufnahme der zugehörigen lokalen Daten reserviert – ein sogenannter **Stack Frame**. Bei Verlassen des Anweisungsblocks wird der Stack Frame wieder freigegeben und man besitzt somit keinen 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

> Beispiel:
>
> Ein solcher Stack Frame lässt sich wieder mit Keksen veranschaulichen. Man kann sich die Stack Frames nämlich als breite Aufbewahrungsbox vorstellen. Man hat viele verschiedene Boxen (viele verschiedene Stack Frames), die man in einer Schublade aufbewahrt. Die Schublade ist allerdings sehr eng, weshalb man immer nur die vorderste Keksbox herausholen kann. Hält man nun aber eine Box in der Hand, kann man problemlos in den Keksen darin herumwühlen und sich einen beliebigen Keks herausholen.

Figure 8: Eine abstrakte Darstellung des Stacks und dessen StackFrame. Man kann nur auf das oberste Stackframe zugreifen, darin jedoch beliebig auf Variablen zugreifen.

Funktionsaufrufe und der Stack

Wird nun eine Funktion aufgerufen, geschehen folgende Dinge:

1. Es wird zur ersten Anweisung im Funktionsrumpf gesprungen
2. Ein neuer Stack Frame – der **Function Stack Frame** – wird angelegt; zur Aufnahme der lokalen Daten der Funktion

Das lässt sich auch nochmal mithilfe der Grafik links veranschaulichen. Lokale Daten im Function Stack Frame werden hintereinander im Stack abgelegt. Zu diesen lokalen Daten zählen:

- Für die Eingabeparameter übergebene Werte
- **Rücksprungadresse**: Adresse des auf den Funktionsaufruf folgenden Befehls im Programm (ist die Funktion abgearbeitet worden, nutzt man die Rücksprungadresse, um zum nächsten Befehl zu springen, der bearbeitet werden soll. Somit wird verhindert, dass eine Funktion fertig ist und „gar nichts mehr“ passiert.)
- lokale Variablen

2.7.5 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: Beispiel für das Call-by-Value Prinzip

Das Beispiel wird durch eine Abfolge von Folien realisiert und lässt sich im Skript nicht so schön darstellen wie im Foliensatz. Schauen Sie es sich daher im Foliensatz an.

Die Kernessenz hierbei ist, dass bei einem Funktionsaufruf die übergebenen Parameter (also bspw. Zahlenwerte) als Kopie übergeben werden. Somit arbeitet man nicht „mit den echen Werten“, sondern auf einer Kopie.

2.8 Buskonzept

Busse dienen, ähnlich wie auf einer Straße, dem Transport. Es werden allerdings keine Menschen, Tiere und Koffer von A nach B befördert, sondern Daten.

2.8.1 Wie werden Daten im Rechner übertragen (transportiert)?

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

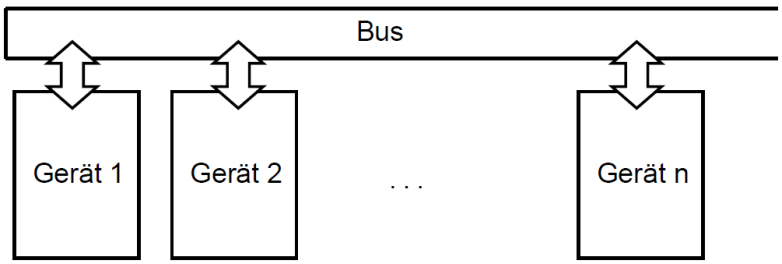
- Bitwert 1: Stromfluss (hohe Spannung)
- Bitwert 0: kein Stromfluss (niedrige Spannung)

Hierbei existieren zwei Möglichkeiten, wie man Daten übertragen kann:

1. Möglichkeit 1: Spezielle Übertragungsleitungen zwischen je zwei Komponenten des Von-Neumann-Rechners
    - hohe Übertragungsgeschwindigkeit
    - **ABER: viel zu viele Leitungen**
2. 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)

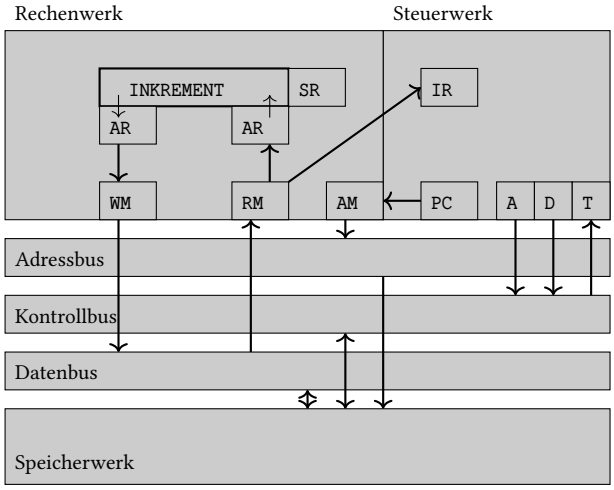


Figure 9: Nun sind alle Komponenten des Von-Neumann-Rechners bekannt und erklärt worden. Somit ergibt sich diese Gesamtübersicht.

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

2.9 Bewertung des von-Neumann-Rechners

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

Wie ist es heute?

In modernen Computern werden beide Konzepte vereint: In Ebenen der Speicherhierarchie, die sehr nah an der CPU sind, wird die Harvard-Architektur realisiert, in den weiter entfernten Ebenen die von-Neumann-Architektur.

2.10 Bedienung des Rechners

Siehe Foliensatz

2.11 Literatur

Siehe Foliensatz