LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

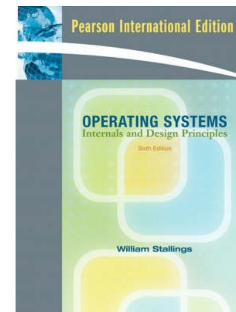
Vorlesung Betriebssysteme

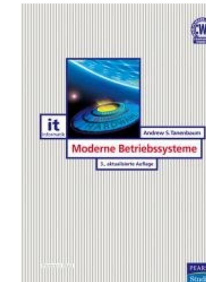
am 28. Oktober 2015

Prof. Dr. Claudia Linnhoff-Popien, Dipl.-Inf. Philipp Marcus

Literatur

Dieser Foliensatz wurde auf Basis folgender Literatur erstellt:





- Claudia Linnhoff-Popien: Skript zur Vorlesung Betriebssysteme im WS15/16
- William Stallings: Operating Systems – Internals and Design Principles, 6. Auflage, Pearson International Edition
- Andrew S. Tanenbaum: Moderne Betriebssysteme, 2. überarbeitete Auflage, Pearson Studium
- Erich Ehses, Lutz Köhler, Petra Riemer, Horst Stenzel, Frank Victor: Betriebssysteme – Ein Lehrbuch mit Übungen zur Systemprogrammierung in Unix/Linux
- Andrew S. Tanenbaum: Computerarchitektur – Strukturen – Konzepte - Grundlagen, 5. Auflage, Person Studium
- David A. Patterson, John L. Hennessy: Rechnerorganisation und – entwuf – Die Hardware/Software-Schnittstelle, 3. Auflage, Spektrum Verlag

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

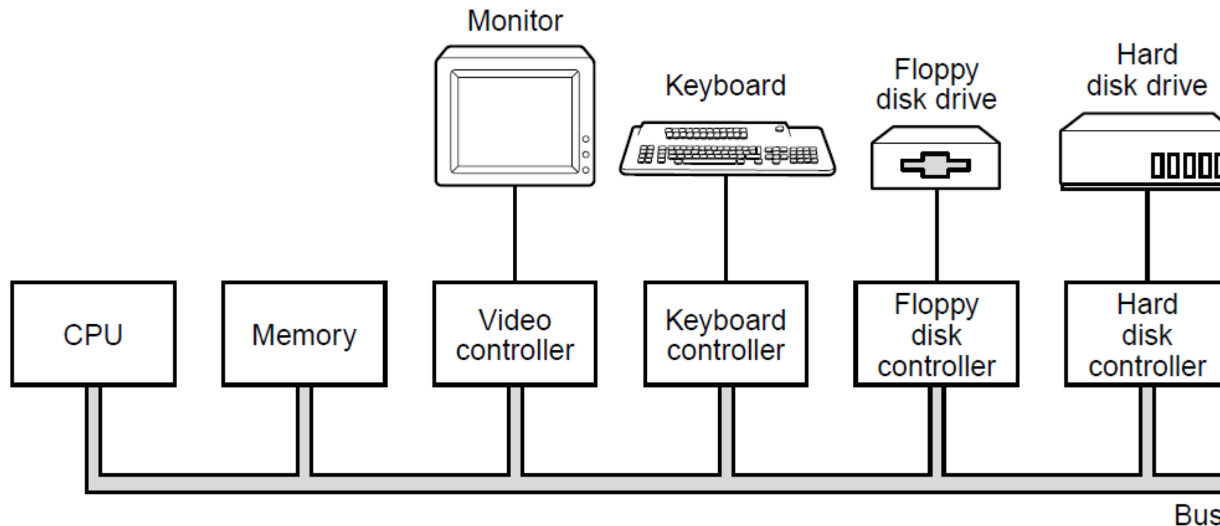
Gliederung

- Einführung
  - Was ist ein Betriebssystem?
  - Die Geschichte von Betriebssystemen
- Die Ebene der physischen Geräte
  - Boolesche Algebra
  - Gatter
- Ebene der Mikroarchitektur
  - Der Befehlszyklus
  - Interrupts
- Die Ebene der Maschinensprache
  - Ein kurzer Überblick zur SPIM-Programmierung
  - Einfache Befehle und ihre Wirkung

EINFÜHRUNG

Geräte in Computern

- Anbindung des Prozessors an die E/A-Geräte:

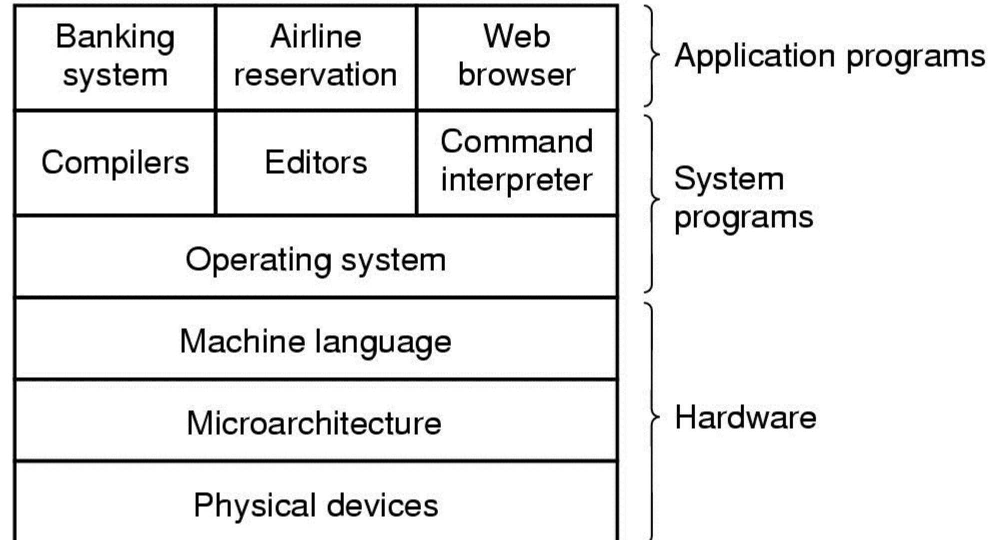


- Die **CPU**, der **Hauptspeicher** und alle **E/A-Module** sind mit dem **Systembus** verbunden
- Die **E/A-Module** bestehen aus Controller und der eigentlichen Hardware (z.B. Laser, rotierende Platten usw.)
- Jedes Gerät ist über einen Controller (E/A-Modul) mit dem Systembus verbunden
- Die Controller nehmen Steuerungsbefehle von der CPU entgegen
- Der Controller bietet Speicherregister zum Empfangen und Ausgeben von Daten

Einführung in Betriebssysteme

- Erstellung von Programmen, welche Geräte benutzen ist sehr komplex

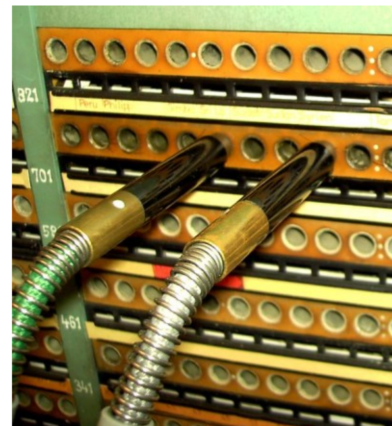
→ Betriebssystem als zusätzliche Softwareschicht. Einfache Schnittstelle zur Hardware.



- Physische Geräte: Integrierte Schaltungen, Drähte, Stromversorgung, Bildschirm u.v.m…
- Mikroarchitektur: Funktionale Einheiten – z.B. Prozessor (CPU)
- Maschinensprache: Hardware-Anweisungen und Assembler
- Betriebssystem: Komplexität verstecken

1. Generation: 1945 – 1955

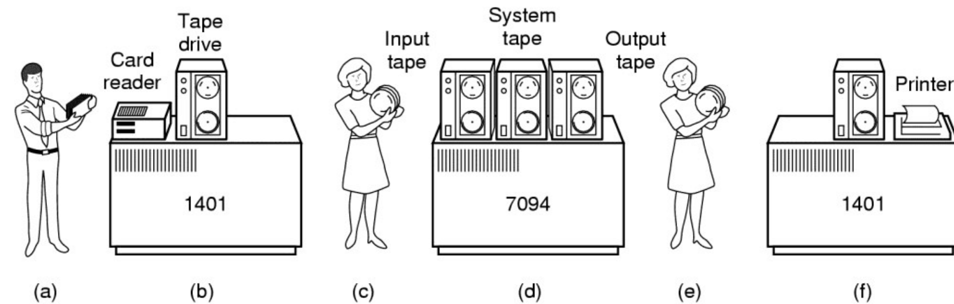
- Etwa gegen Ende des Zweiten Weltkriegs entstanden die ersten Rechner.
- Größen dieser Zeit:
  - Konrad Zuse (1910 bis 1995)
  - John v. Neumann (1903 bis 1957)
- Riesige Apparate mit zehntausenden von Röhren (viele Ausfälle).
- Taktzeiten wurden in Sekunden angegeben
- Programme über Klinkenfelder gesteckt oder in Maschinencode eingegeben



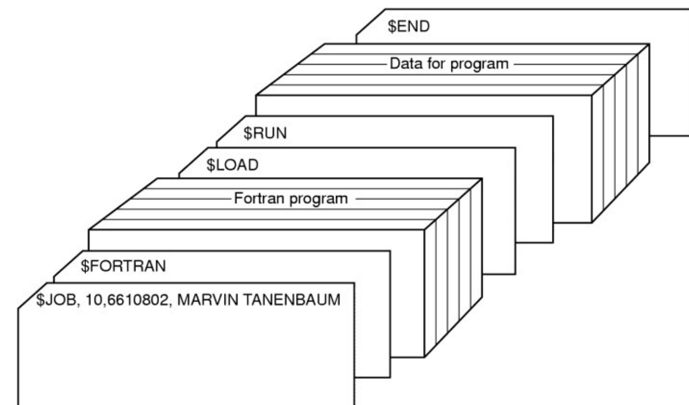
„Zuse-Z4-Totale deutsches-museum“ von Clemens PFEIFFER - CANON PowerShot G7. Lizenziert unter CC BY 2.5 über Wikimedia Commons - https://commons.wikimedia.org/wiki/File:Zuse-Z4-Totale_deutsches-museum.jpg#/media/File:Zuse-Z4-Totale_deutsches-museum.jpg

2. Generation: 1955 bis 1965

- Einführung von Lochkarten statt Klinkenfelder
- Transistoren und Stapelsysteme
- Übergang zum Einspielen von Programmen auf Magnetbänder

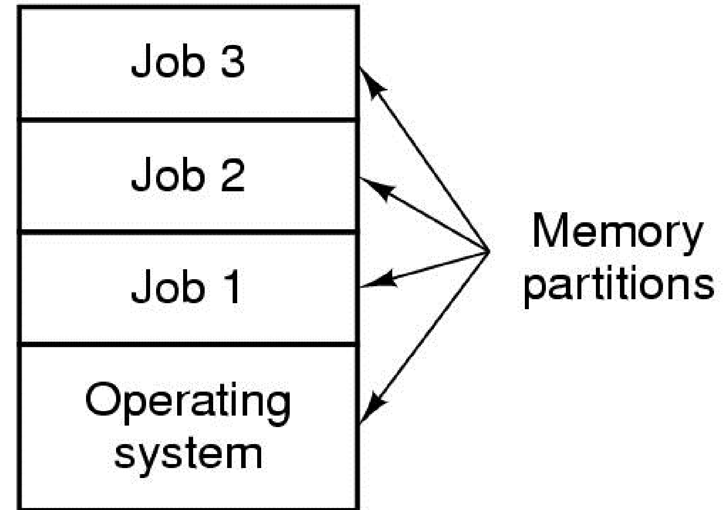


- Programmierung in FORTRAN
- Aufbau des Programmstapels:
  - $JOB-Karte: Laufzeit, Abrechnung, Programmierer
  - $FORTRAN: Compiler laden
  - $LOAD: Übersetztes Prog. Laden
  - $RUN: Programm Ausführen
  - $END: Ende des Jobs
- Problem: Jede E/A unterbrach die CPU



3. Generation: 1965 bis 1980

- IBM Betriebssystem /360 (mehrere Millionen Assembler-Zeilen, tausende Programmierer)
- IBM verfolgte eine Rechnerlinie mit Ausprägung von Privatanwender bis Großrechner
- **Multiprogramming**: Die CPU muss bis zu 90% der Zeit auf E/A warten → Zwischenzeitlich anderes Programm ausführen
- Speicher wurde dazu aufgeteilt
- Spooling: Einlesen von Jobs auf Platte und Ausführen bei freier Kapazität
- **Timesharing**: Pseudo-Parallelität, da die meisten Nutzer kaum Leistung benötigen (Dialogbetrieb)

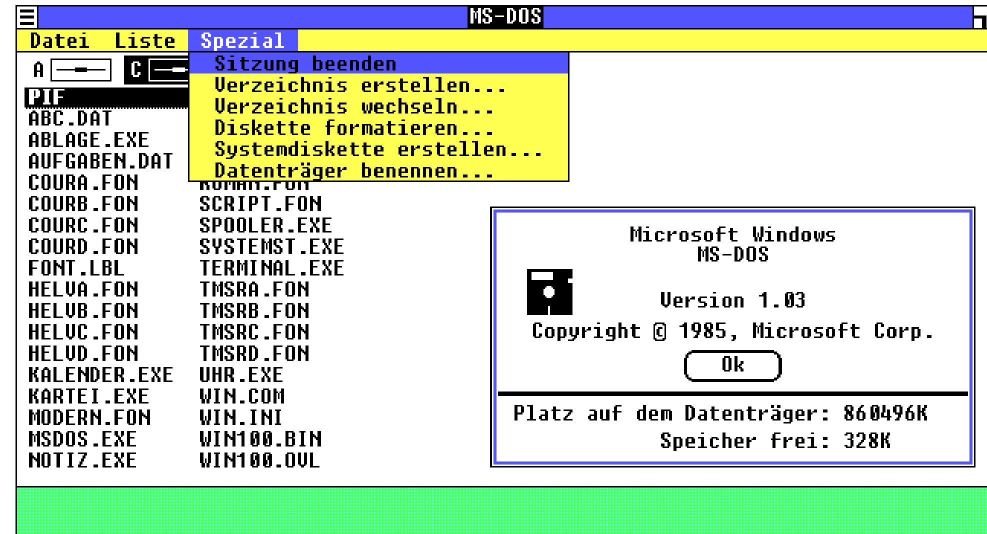


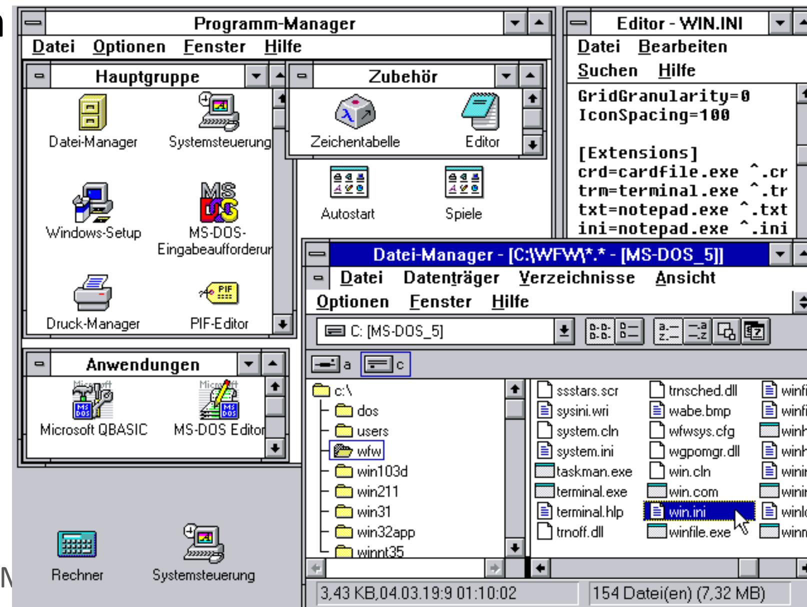
- 1962: Compatible Time Sharing System am MIT
- 1965: Multiplexed Information and Computing Service (MULTICS)
- Uniplexed Information and Computing Service (Unics) war Einbenutzerversion von MULTICS
- Unix war Neuimplementierung in C

4. Generation: seit 1980 (PCs)

- Hohe Rechenleistung zu günstigen Preisen durch LSI (Large Scale Integration) Schaltkreise
- MS-DOS (Microsoft Disk Operating System)
- Erfindung der GUI (bsp. Windows)

- Netzwerk-Betriebssystem:
  - Benutzer weiß um Existenz fremder Rechner und kann sich dort anmelden
- Verteiltes Betriebssystem:
  - Soll dem Benutzer wie ein traditionelles Einprozessorsystem erscheinen, obwohl es (meist) auf räumlich getrennten Rechnern läuft





„Win1.03“ von unbekannt - eigener Bildschirmausdruck (unter VmWare erstellt). Lizenziert unter PD-Schöpfungshöhe über Wikipedia - https://de.wikipedia.org/wiki/Datei:Win1.03.png#/media/File:Win1.03.png

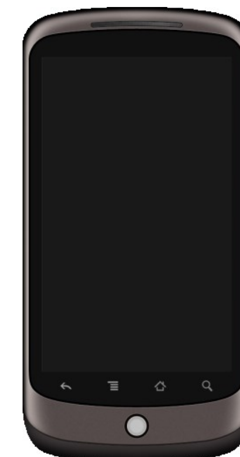
„Wfw“ von unbekannt - eigener Bildschirmausdruck (unter VmWare erstellt). Lizenziert unter PD-Schöpfungshöhe über Wikipedia - https://de.wikipedia.org/wiki/Datei:Wfw.PNG#/media/File:Wfw.PNG

5. Generation: seit ca. 2000 (mobile Betriebssysteme)

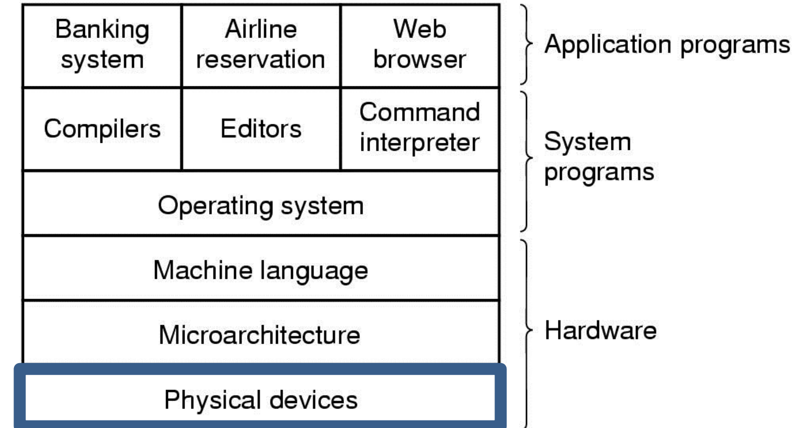
- Als erstes Smartphone wird der Simon Personal Computer von BellSouth und IBM betrachtet (ab 1994). Symbian bis 2006 Marktführer mit 76%. Alternativen waren Windows Mobile, BlackBerry OS und Palm OS.
- Ab der Einführung des iPhones im Jahr 2007 erfolgte Umstieg auf Touch Screens. Es entstanden Android, Palm webOS, und Windows Phone 7.
- Mobile Endgeräte besitzen begrenzte Energiereserven, die geschont werden müssen.
- Begriff der Apps für mobile Anwendungen
- Energiesparfunktionen, Herstellerbindung, Sicherheit, …

„IBM Simon Personal Communicator“ von Bcos47 - http://commons.wikimedia.org/wiki/File:IBM_SImon_in_charging_station.png. Lizenziert unter Gemeinfrei über Wikimedia Commons - https://commons.wikimedia.org/wiki/File:IBM_Simon_Personal_Communicator.png#/media/File:IBM_Simon_Personal_Communicator.png

„IPhone 2G PSD Mock“ von Justin14 - Eigenes Werk. Lizenziert unter CC BY-SA 3.0 über Wikimedia Commons - https://commons.wikimedia.org/wiki/File:IPhone_2G_PSD_Mock.png#/media/File:IPhone_2G_PSD_Mock.png



„Nexus One“ von Zach Vega - Eigenes Werk. Lizenziert unter CC BY-SA 3.0 über Wikimedia Commons - https://commons.wikimedia.org/wiki/File:Nexus_One.png#/media/File:Nexus_One.png



PHYSISCHE GERÄTE

Boolesche Algebra (1)

- George Boole: englischer Mathematiker (1815 – 1864)
- Beschäftigung mit formaler Sicht heutiger digitaler Strukturen

- **Herangehensweise:**
  - Σ_2 = { 0, 1} (Alphabet) wird als B bezeichnet
  - Man betrachte die Variablen a, b $\in$ B und definiert drei Operationen
- Der **OR-Operator** (geschrieben + oder $\vee$) – auch logische Summe bezeichnet:

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Boolesche Algebra (2)

- Der AND-Operator (geschrieben * oder ∧) – auch logisches Produkt bezeichnet:

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

- Der NOT-Operator (geschrieben als $\bar{a}$ oder $\neg$ a) – auch Invertierung genannt:

a	NOT a
0	1
1	0

- → Boolesche Algebra ergibt sich als (B, AND, OR, NOT)

Boolesche Algebra (3)

- Gesetze zur Manipulation logischer Gleichungen:

Kommutativgesetz: $a \vee b = b \vee a$ und $a \wedge b = b \wedge a$

Assoziativgesetz: $(a \vee b) \vee c = a \vee (b \vee c)$ und $(a \wedge b) \wedge c = a \wedge (b \wedge c)$

Distributivgesetz: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ und $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

Identitätsgesetz: $a \vee 0 = a$ und $a \wedge 1 = a$

Null– und Eins–Gesetz: $a \wedge 0 = 0$ und $a \vee 1 = 1$

Komplementärgesetz: $a \vee \overline{a} = 1$ und $a \wedge \overline{a} = 0$

Verschmelzungsgesetz: $(a \vee b) \wedge a = a$ und $(a \wedge b) \vee a = a$

de Morgansche Regeln: $\overline{a \vee b} = \overline{a} \wedge \overline{b}$ und $\overline{a \wedge b} = \overline{a} \vee \overline{b}$

Boolesche Funktionen

- Eine Funktion f: $B^n \rightarrow B$ heißt **n-stellige Boolesche Funktion**
- Im Falle n = 1 ergeben sich 4 mögliche einstellige Funktionen f(x) =0; f(x)=1; f(x)=x und f(x) = ¬ x:

x	0	x	$\overline{x}$	1
0	0	0	1	1
1	0	1	0	1

- Im Falle n = 2 ergeben sich 16 mögliche zweistellige Boolesche Funktionen:

x	y	0	AND	$x\overline{y}$	x	$\overline{x}y$	y	$\not\leftrightarrow$	OR	NOR	$=$	$\overline{y}$	$\overline{\overline{x}y}$	$\overline{x}$	$\overline{x\overline{y}}$	NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}

- Es gibt 2^{2^n} n-stellige Boolesche Funktionen

Die Termdarstellung Boolescher Funktionen

- Die Funktion NAND in Termdarstellung…

x	y	0	AND	$x\bar{y}$	x	$\bar{x}y$	y	$\not\leftrightarrow$	OR	NOR	$=$	$\bar{y}$	$\overline{\bar{x}y}$	$\bar{x}$	$\overline{x\bar{y}}$	NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}

- … lautet:

Distributiv Komplementär

$$\bar{x}.\bar{y} + \bar{x}.y + x.\bar{y} = \bar{x}.(y+\bar{y}) + x.\bar{y} = \bar{x}.1 + x.\bar{y} = \bar{x} + x.\bar{y} = (\bar{x}+x).(\bar{x}+\bar{y}) = (\bar{x}+\bar{y}) = \overline{x.y}$$

Null- und Eins Distributiv Komplementär

Null- und Eins

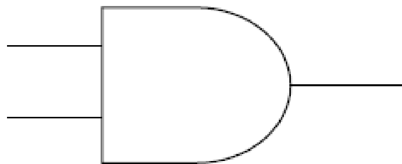
- Die Darstellung für = lautet:

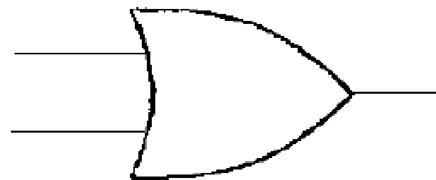
  $x.y + \bar{x}.\bar{y}$

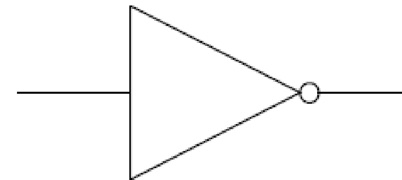
Gatter und Logische Bausteine (1)

- Implementierung der Booleschen Funktionen nun als Gatter
- Schaltungen aus Verknüpfung einfachster Elemente aufbauen
- Gatter: Realisieren Funktionen zweiwertiger Signale
  - 0 – 1 Volt: Binäre 0
  - 1 – 5 Volt: Binäre 1

- Beispiele für Gatter in der IEEE-Darstellung:



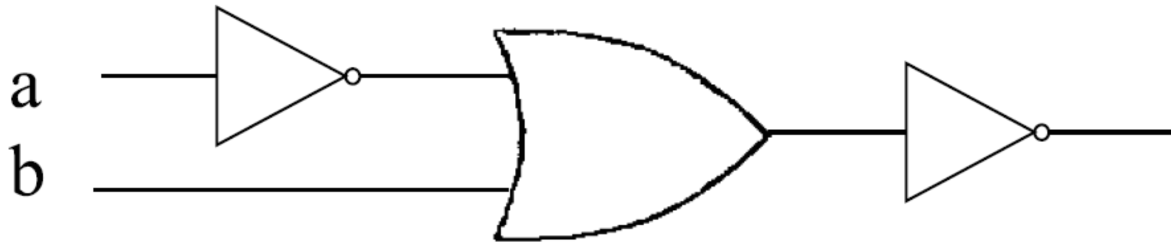




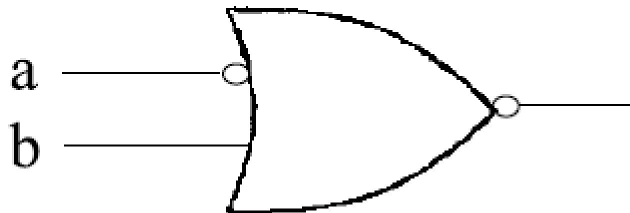
AND OR NOT

Gatter und Logische Bausteine (2)

- Darstellung der Funktion $\overline{\overline{a} + b}$:

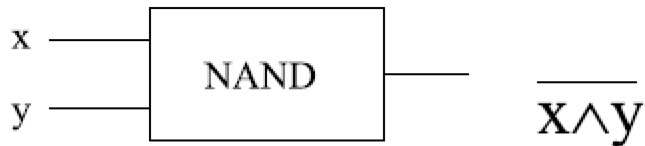


- Vereinfachte Darstellung:

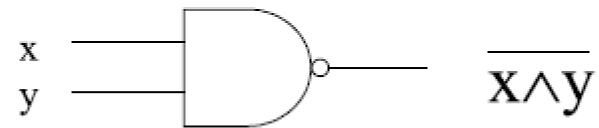


Gatter und Logische Bausteine (3)

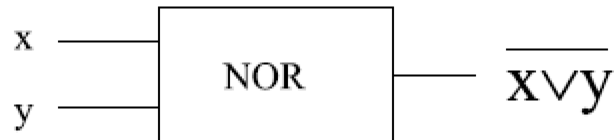
- Betrachte NAND und NOR:



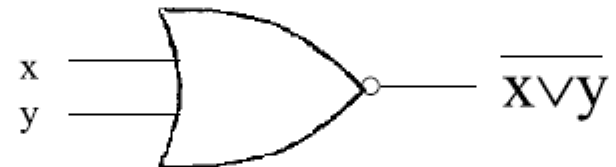
$\overline{x \wedge y}$ steht für



$\overline{x \wedge y}$



$\overline{x \vee y}$ steht für

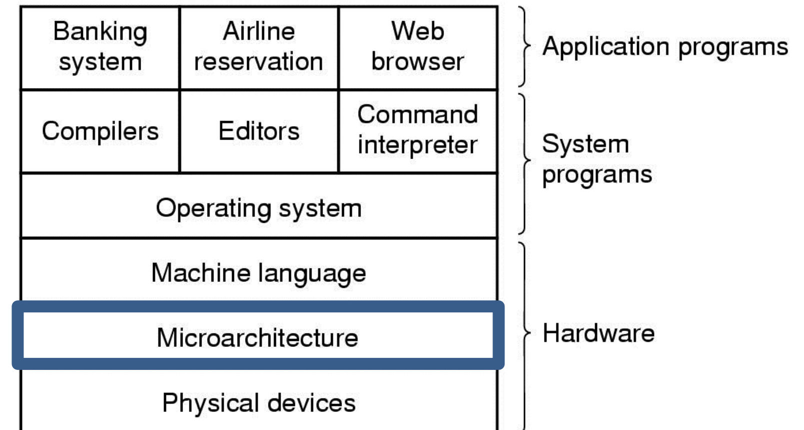


$\overline{x \vee y}$

- Eine Funktion F: $B^n \to B^m$ mit $m,n \in N$ und $n, m \geq 1$ heißt **Schaltfunktion**

→ **Schaltfunktionen werden durch m Boolesche Funktionen dargestellt:**

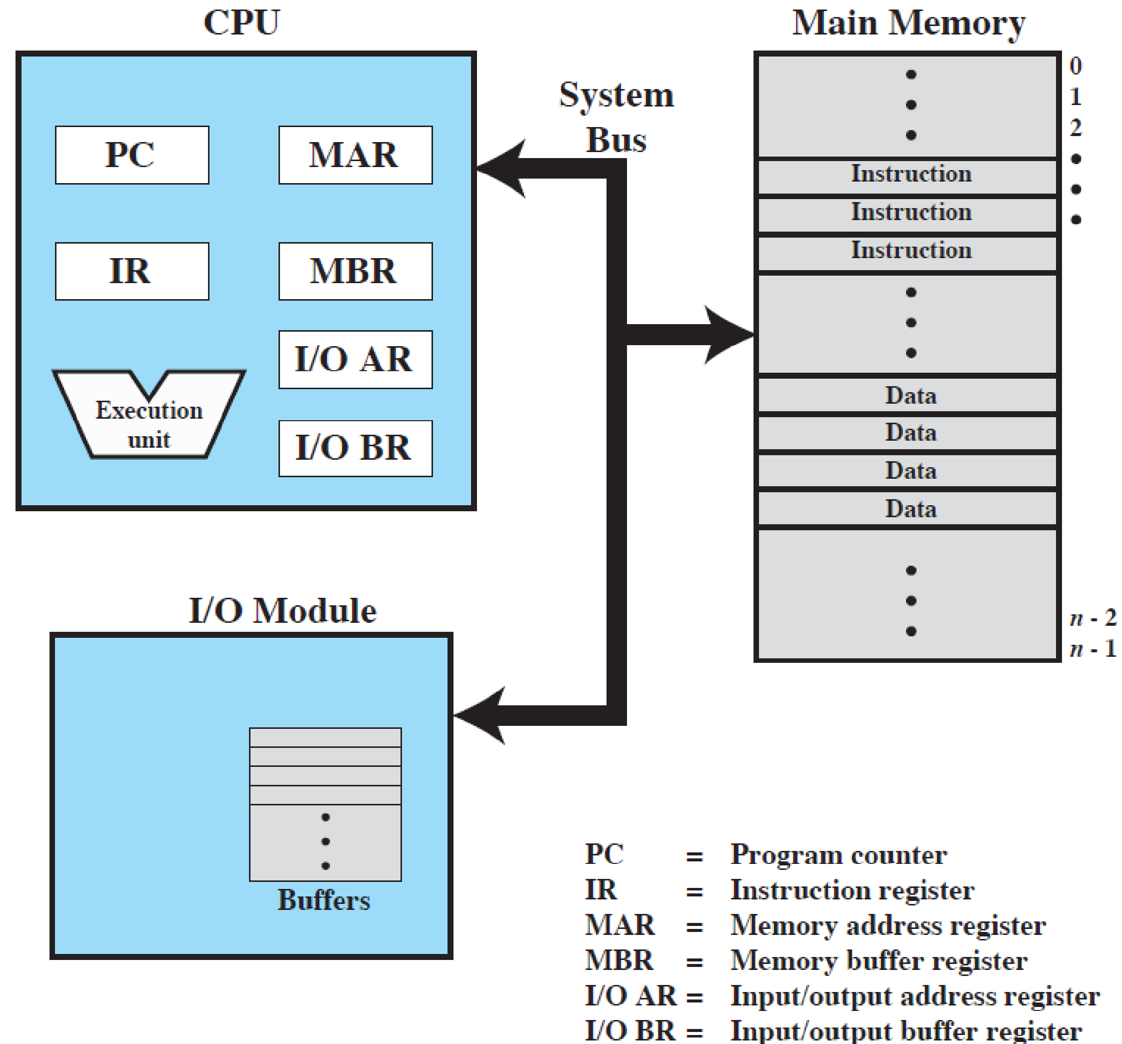
$$F(x_1, ..., x_n) = (f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_m(x_1, ..., x_n))$$



DIE MIKROARCHITEKTUR VON COMPUTERSYSTEMEN

Basiselemente

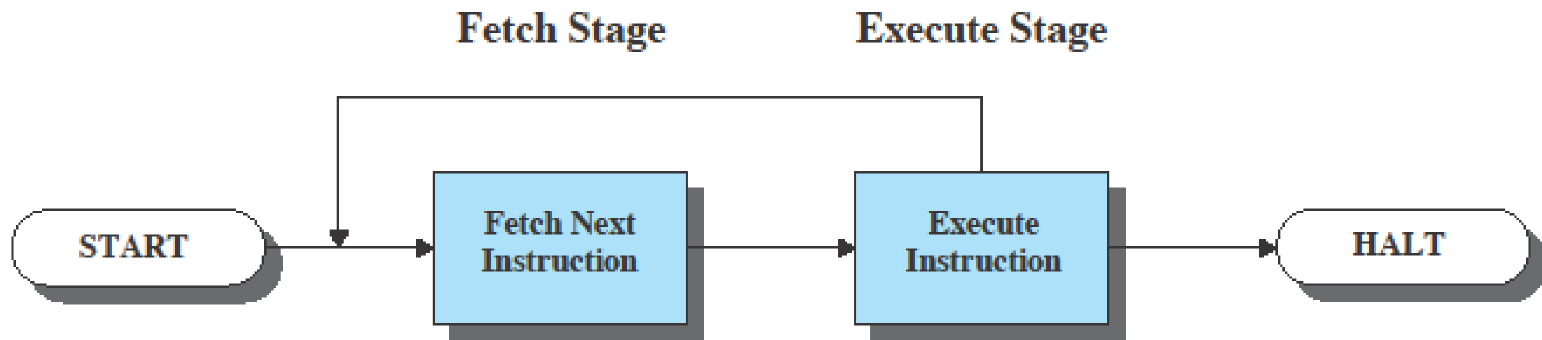
- **AR (address register)** beschreiben die Zieladresse von zu schreibenden/lesenden Daten
- **BR (buffer register)** stellen die Daten bereit oder dienen zum Einlesen von Daten.
- **PC (program counter)** beschreibt Adresse des nächsten Befehls
- **IR (instruction register)**
- **Hauptspeicher:** Satz von fortlaufend nummerierten Speicherzellen
- Inhalt des Hauptspeichers kann als **Befehl** oder **Daten** interpretiert werden



Befehlsausführung (1)

- Befehlsausführung in zwei Schritten:
  - Schritt 1 (Befehlsabruf): Prozessor liest Befehl aus Hauptspeicher
  - Schritt 2 (Befehlsausführung)

- Der Abrufzyklus bzw. Ausführungszyklus der CPU:



→ Maßgeblich ist der Program Counter (PC)

Befehlsausführung (2)

- Program Counter (PC) enthält Adresse des nächsten Befehls
- → Abruf dieses Befehls im ersten Schritt
- → Erhöhung des PC nach Befehlsabruf (falls nicht anders vorgesehen)

- Beispiel: Speicherwort belegt 16 Bit. Befehlsformat:

0001 = Akk. aus dem Speicher laden
0010 = Akk. im Speicher ablegen
0101 = Aus dem Speicher dem Akk. hinzufügen

- 2^4 = 16 mögliche Befehle
  - Prozessor ←→ Speicher: Übertragen von Daten
  - Prozessor ←→ E/A: Daten mit Peripheriegerät austauschen
  - Datenverarbeitung: Arithmetische bzw. logische Operationen durchführen
  - Steuerung: Änderung der Ausführungsreihenfolge (PC)
- 2^{12} = 4096 (4K) Speicherwörter direkt adressierbar

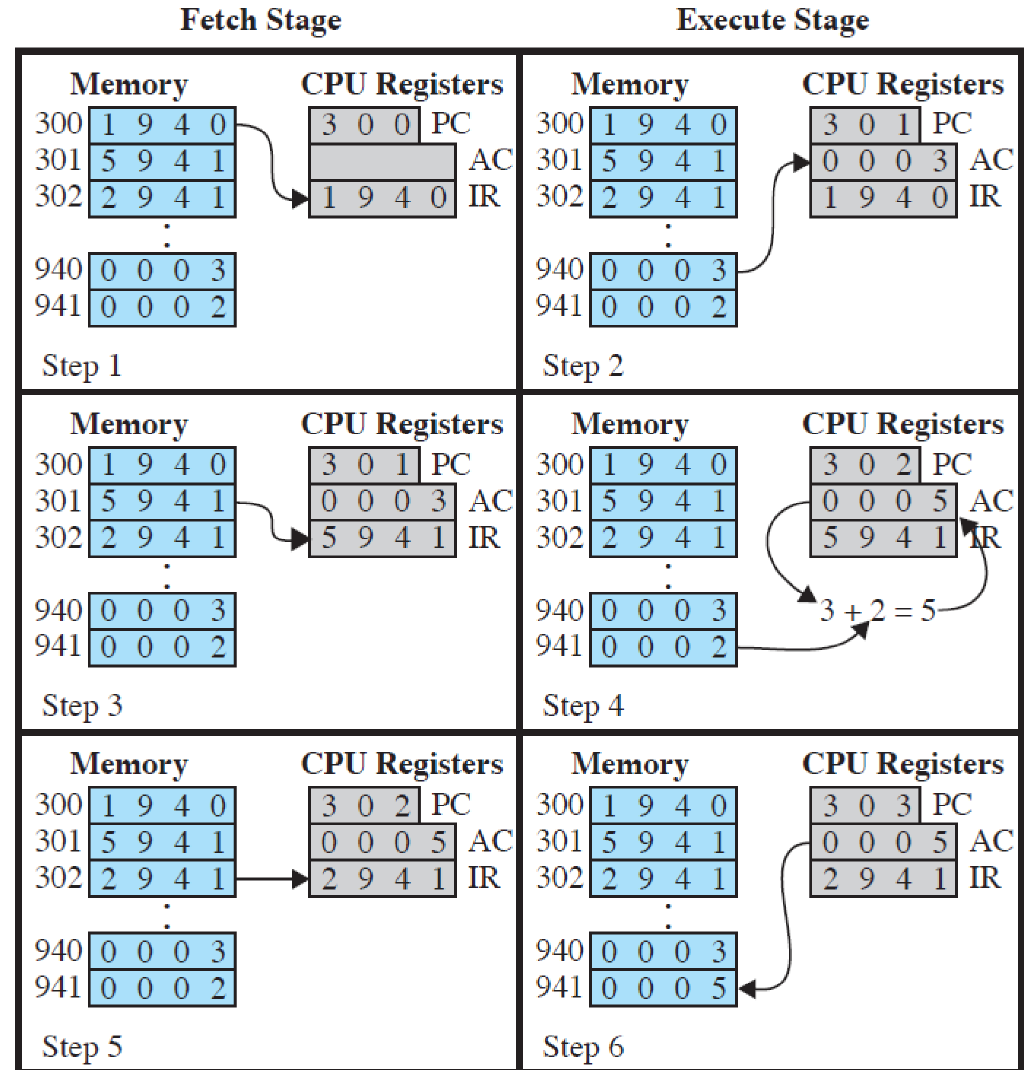
Befehlsausführung (3)

Addiere Inhalt des Speicherworts 940 zu Speicherwort 941. Speichere Ergebnis in Speicherwort 941.

1. Lade ersten Befehl von 300, erhöhe PC
2. Lade Akk. mit Adr. 940
3. Lade Befehl von 301, erhöhe PC
4. Addiere Inhalt von Akk. und 941
5. Lade Befehl von 302, erhöhe PC
6. Speichere Akk. in 941

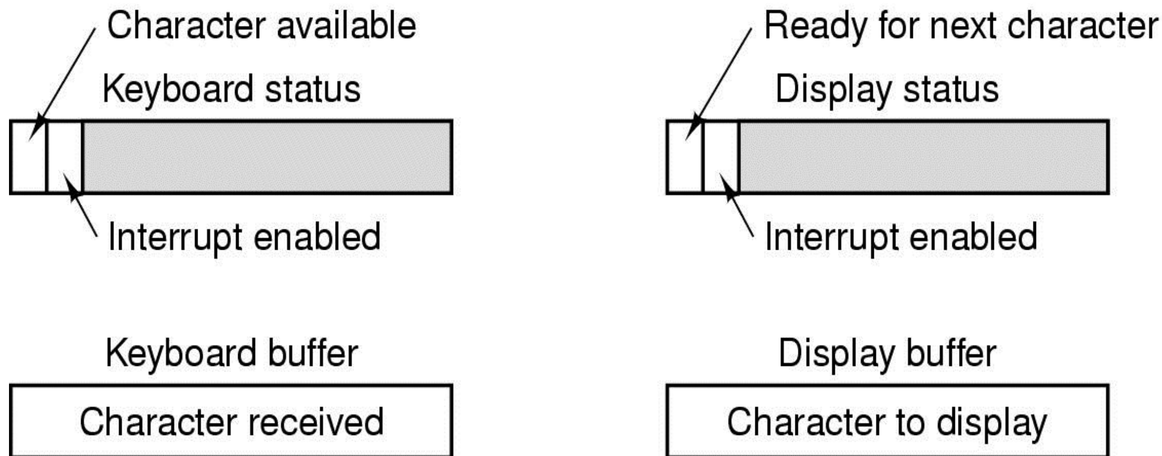
→ Ähnliche Befehlssequenz bei E/A Zugriff möglich

→ Alternativ: DMA (direct memory access). E/A-Modul greift eigenständig auf Speicher zu



Der Zugriff auf E/A-Geräte

- Beispiel: Ein Terminal, bestehend aus Tastatur und Display:



- Am Keyboard eingegebene Zeichen sollen auf Display angezeigt werden

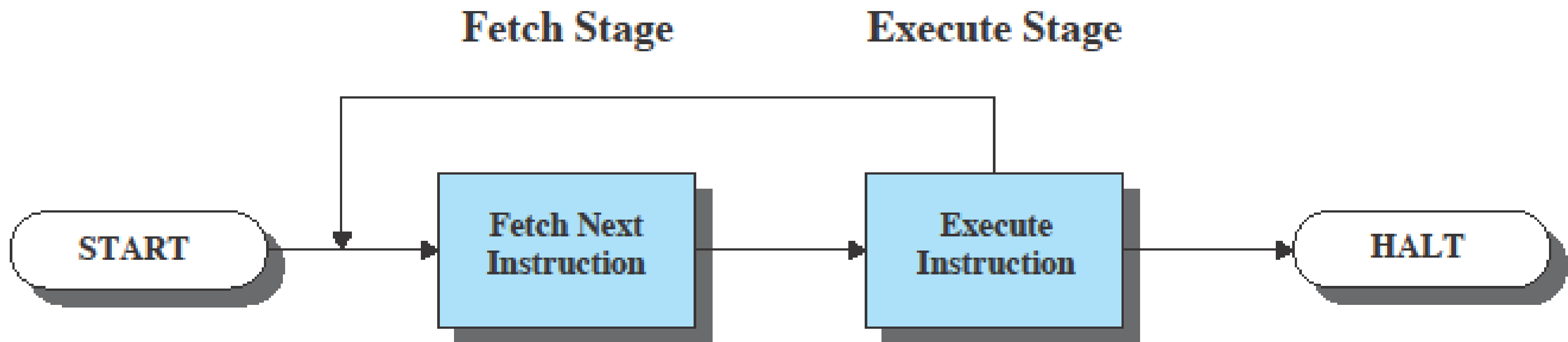
→ Kommunikation nötig

- Jedes Gerät besitzt ein **Statusregister** und ein **Pufferregister**
- **CPU** muss Daten aus dem **Keyboard buffer** in den **Display buffer** kopieren

Frage: Welche Schritte sind dazu nötig?

Übertragen der Datenpuffer

- Wir gehen vorerst weiterhin vom zweistufigen CPU-Zyklus aus:



Zum Übertragen des **Keyboard buffer** in den **Display buffer** sind folgende Schritte nötig:

1. Warten bis `CHARACTER AVAILABE` auf 1 ist
2. Warten bis `READY FOR NEXT CHARACTER` auf 1 ist
3. Kopiere den **Keyboard buffer** in ein Register `R1`
4. Kopiere `R1` in den **Display buffer**
5. Warte bis `READY FOR NEXT CHARACTER` auf 1 und Befehl abgeschlossen ist

Programmiertes Warten (1)

Angenommen, wir schreiben ein Terminal Programm (ähnlich Linux Shell):

```
WHILE True
(1) WHILE CHARACTER AVAILABLE != 1:
            wait
    LOAD KEYBOARD BUFFER to R1   // Controller setzt CHARACTER AVAILABLE auf 0
(2) WHILE READY FOR NEXT CHARACTER  != 1:
            wait
    STORE R1to DISPLAY BUFFER // Controller setzt READY FOR NEXT CHARACTER auf 0
(3) WHILE READY FOR NEXT CHARACTER  != 1:
            wait
```

- Die CPU muss an den Stellen (1), (2), (3) aktiv auf die E/A-Geräte warten
  - (1) bis ein Zeichen eingegeben wurde
  - (2) bis das Display bereit ist
  - (3) bis das Display die Anzeige abgeschlossen hat

→ Programmiertes Warten (busy waiting)

Programmiertes Warten (2)

Frage: Ist programmiertes Warten wirklich ein Problem?

Beispiel zu den Auswirkungen

- CPU mit 1 GHz ~ 10^9 Befehle pro Sekunde
- Festplatte mit 7200 RPM und 4ms Zugriffszeit

→ Festplatte ist 4 Millionen mal langsamer als CPU

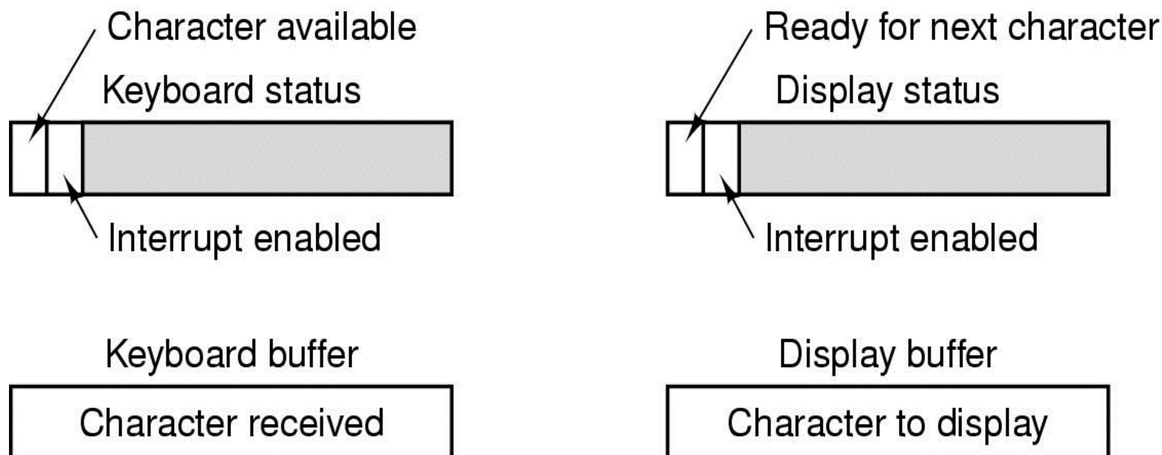
→ Beim Schreiben vergehen im Worstcase 4ms in denen die CPU nur wartet

- **Vorteil:** Abgeschlossene E/A-Operationen werden sofort erkannt (Echtzeit OS)
- **Nachteil:** Die CPU leistet durch die Warteschleifen keine „sinnvolle“ Arbeit

  → Optimal wäre es z.B. einen anderen Prozess in der Zwischenzeit auszuführen

E/A mit Interrupts (1)

- **Idee:** Wenn fertig, schickt der E/A-Controller eine Nachricht an den Prozessor
- **Beispiel aus dem Alltag:**
  - Schicken eines wichtigen Formulars an eine Behörde
  - Permanentes Warten auf Antwort am Briefkasten
  - ODER: Briefträger bitten zu klingeln und in der Zwischenzeit etwas anderes tun
- Ein Programm setzt beim Absetzen der E/A-Operation das **`INTERRUPT ENABLE`**-Flag

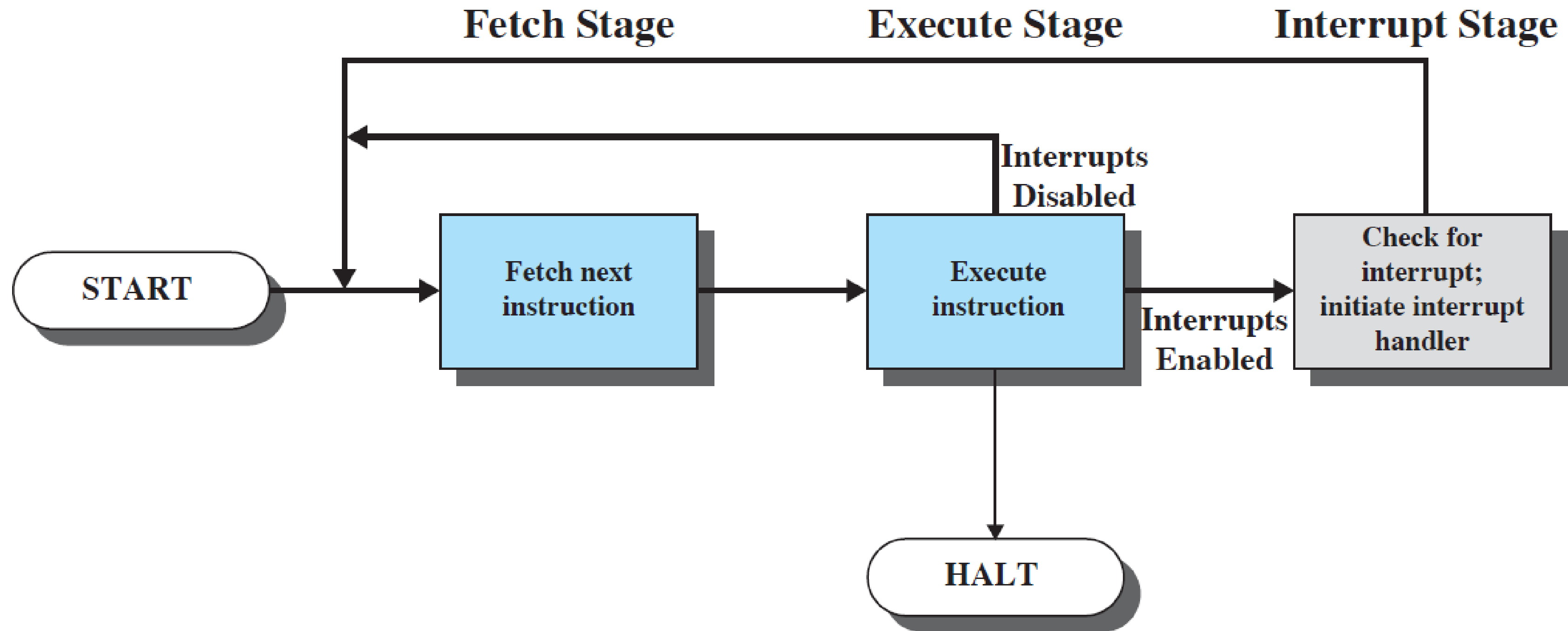


→ Im E/A-Controller : `INTERRUPT ENABLE` **`AND`** `READY FOR NEXT CHARACTER`

→ Absetzen eines Interrupt-Signals auf dem Bus durch den E/A-Controller

E/A mit Interrupts (2)

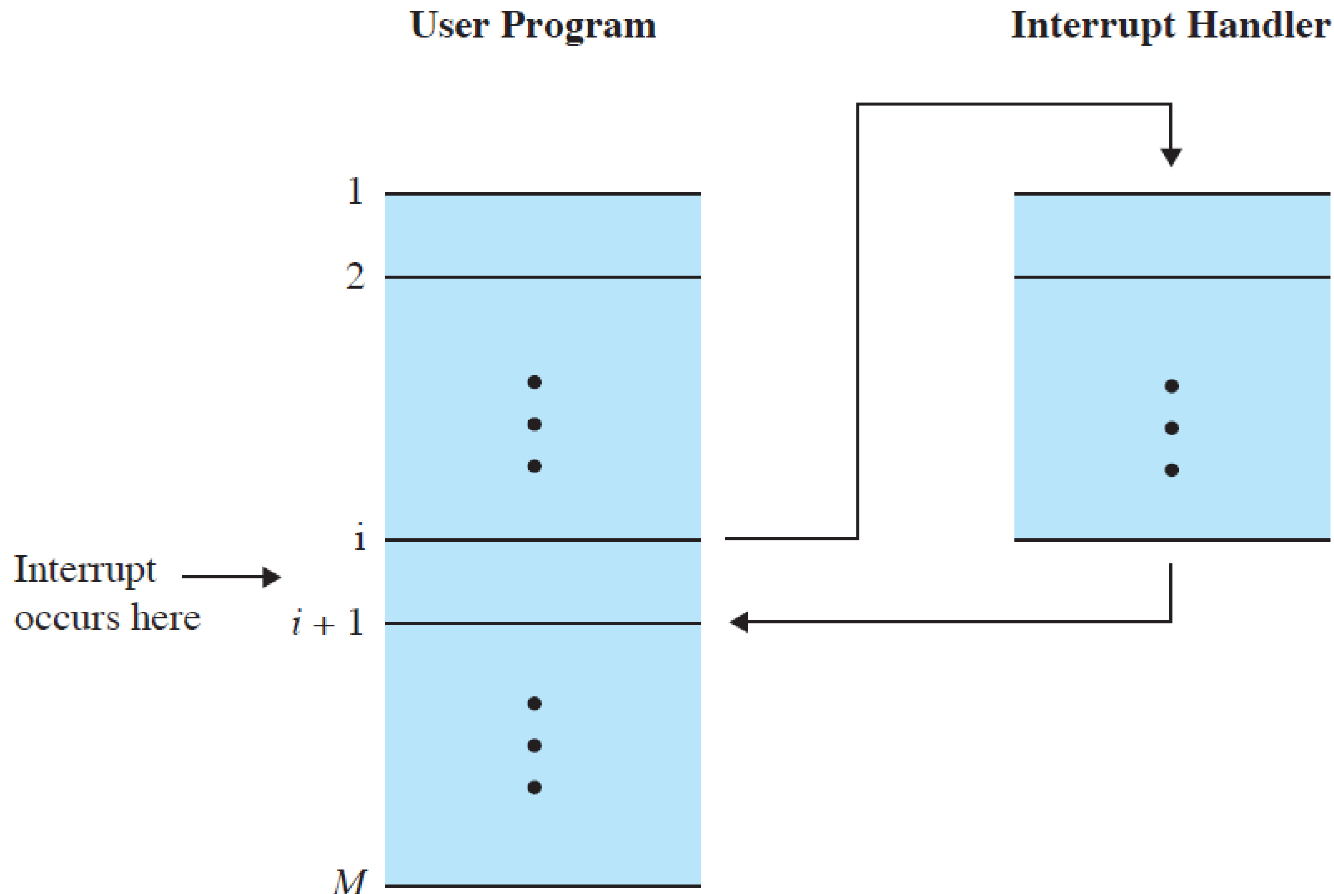
- Bei Interrupt-fähigen Systemen wird der Befehlszyklus im Prozessor angepasst:



- Mit jedem IR ist ein spezieller Interrupt-Handler verknüpft
- CPU springt an die Adresse des Interrupt-Handlers
- Das Betriebssystem kann Systemroutinen, z.B. **WRITE** bereitstellen und blockiert den Nutzerprozess bis alle Worte geschrieben worden sind

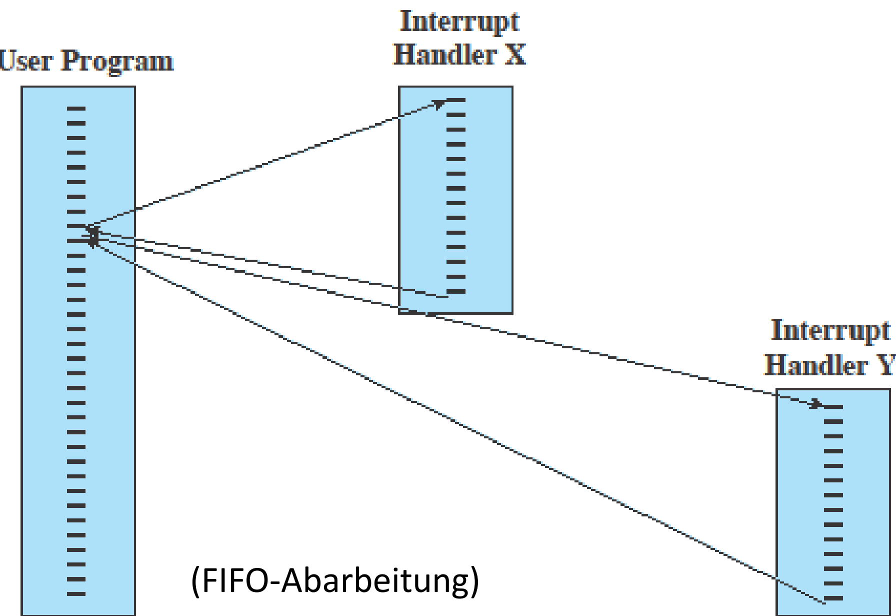
E/A mit Interrupts (3)

- Prozessor führt nun die nächsten Befehle oder ein anderes User Programm aus…
- … bis der E/A-Controller den Interrupt auslöst:

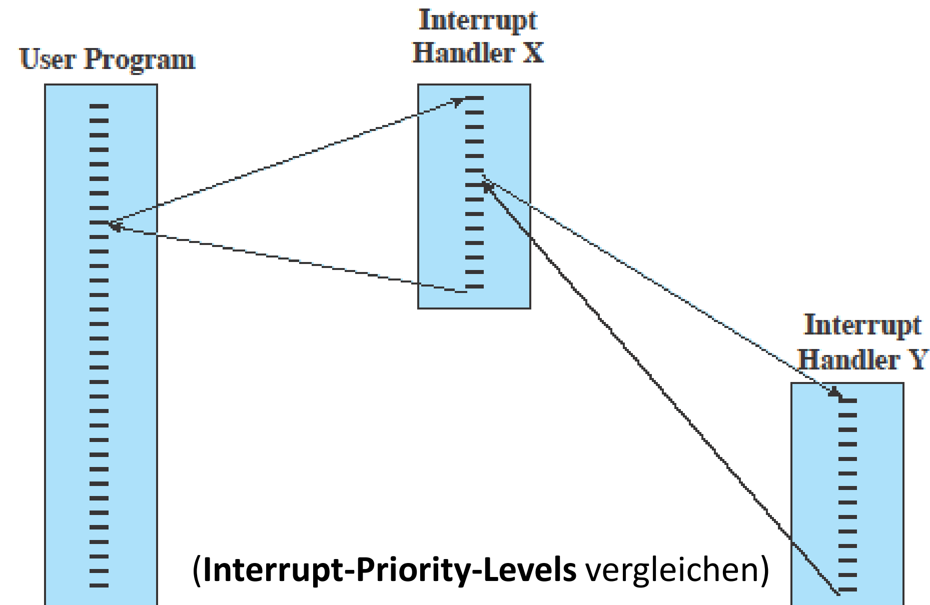


E/A mit Interrupts (4)

- Ein Interrupt besitzt eine ID
- Die Speicheradresse des passenden Interrupt-Handlers wird aus der ID abgeleitet
- Es gibt zwei Möglichkeiten Interrupts abzuarbeiten:



(a) Sequential interrupt processing

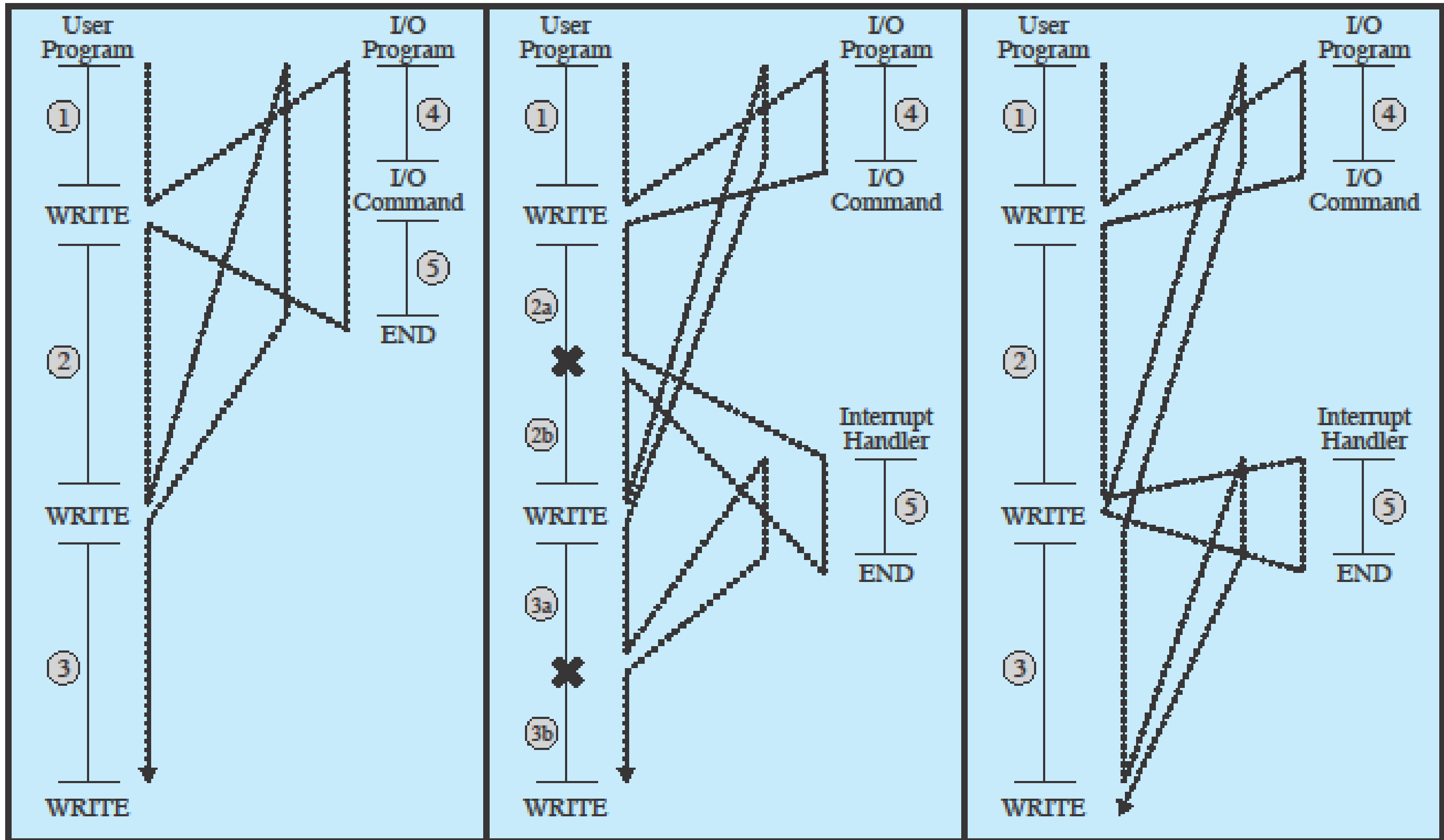


(b) Nested interrupt processing

Sequentielle Abarbeitung (langsam) vs. **Verschachtelte Abarbeitung (teuer)**

E/A mit Interrupts (5)

- Der Programmablauf **ohne** und **mit** Interrupts:



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

E/A mit Interrupts (6)

a) Ablauf ohne Interrupts
   - Nutzerprogramm bleibt blockiert, bis E/A-Operation abgeschlossen ist
   - Beispiel mit programmiertem Warten

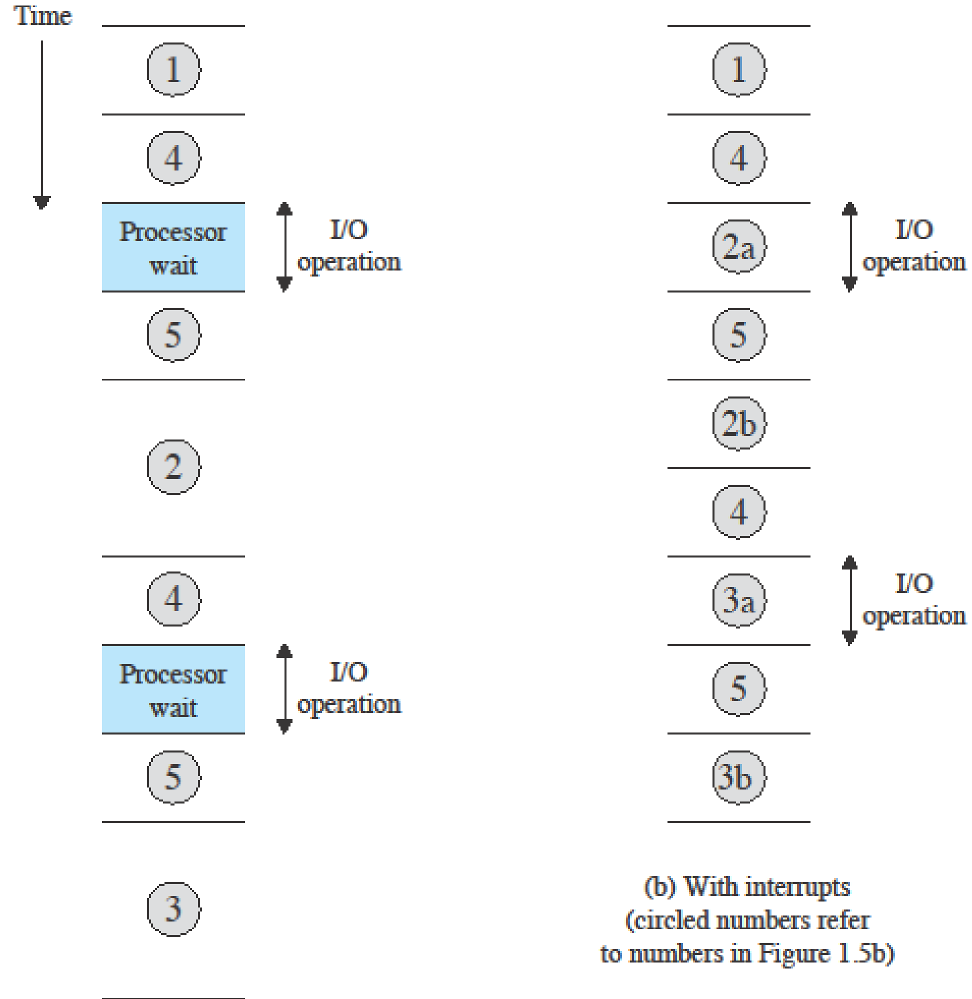
b) Ablauf mit Interrupts für kurze E/A-Operationen
   - Nutzerprogramm kann echt parallel zur E/A-Operation arbeiten
   - Wird erst durch Interrupt und Interrupt-Handler unterbrochen

c) Ablauf mit Interrupts für langwierige E/A-Operationen
   - Die nächste E/A-Operation darf erst ausgelöst werden, wenn die 1. abgeschlossen ist
   - Das Nutzerprogramm muss trotz Interrupts warten, bis die erste E/A-Operation beendet ist
   - In diesem Fall bringen Interrupts weniger Gewinn

E/A mit Interrupts (7)

- Linke Seite:
  - Ohne Interrupts
  - Prozessor wartet bis E/A-Operation abgeschlossen ist
- Rechte Seite:
  - Programmcode 2a kann während der E/A-Operation ausgeführt werden

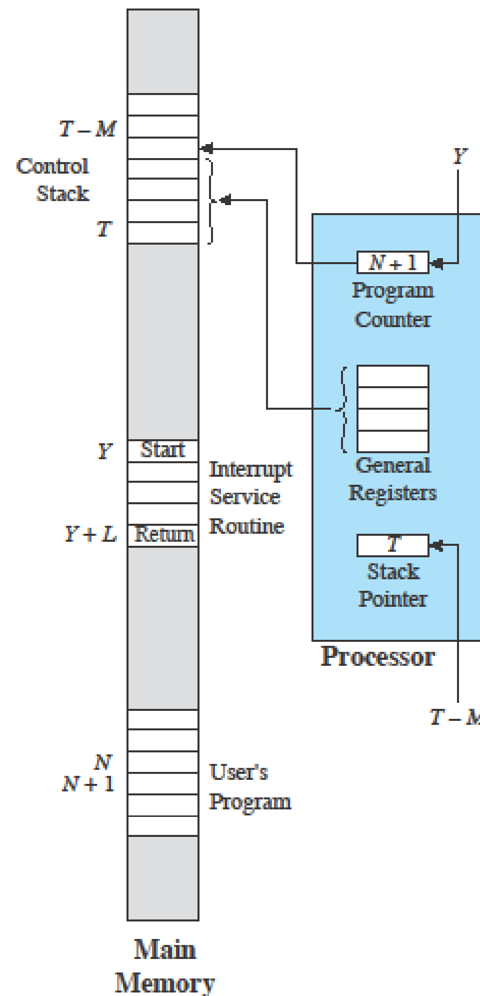
→ Eine Abarbeitung mit Interrupts trägt zur besseren CPU-Auslastung bei



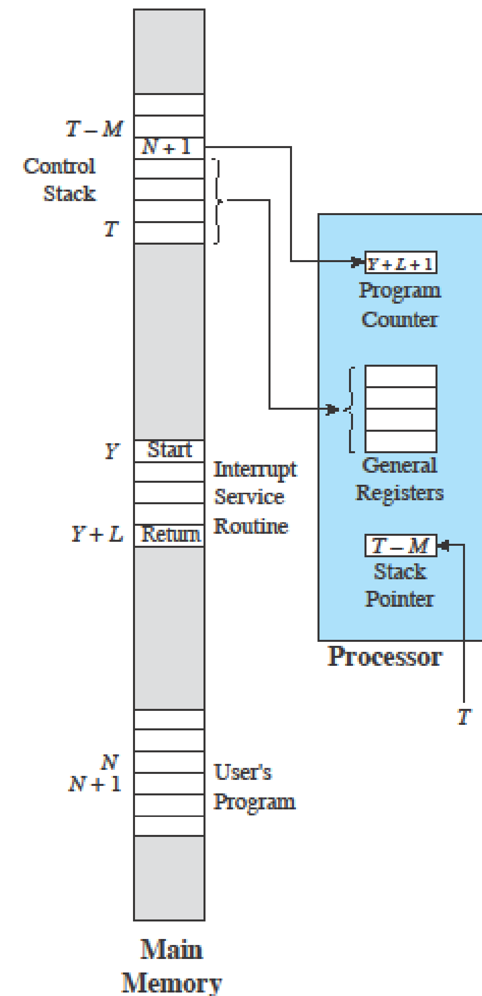
Abarbeitung von Interrupts (1)

- Im Hauptspeicher liegen
  - Controll Stack (OS)
  - Interrupt Handler
  - Nutzerprogramm

- Oft ruft IR-Handler dann OS-Routine auf
- Bevor IR-Handler startet:
  - Sichern aller Daten, so dass Nutzerprogramm später fortgesetzt werden kann
  - Umfasst z.B.:
    - Registerinhalte
    - Program Counter
    - Stack Pointer
    - …



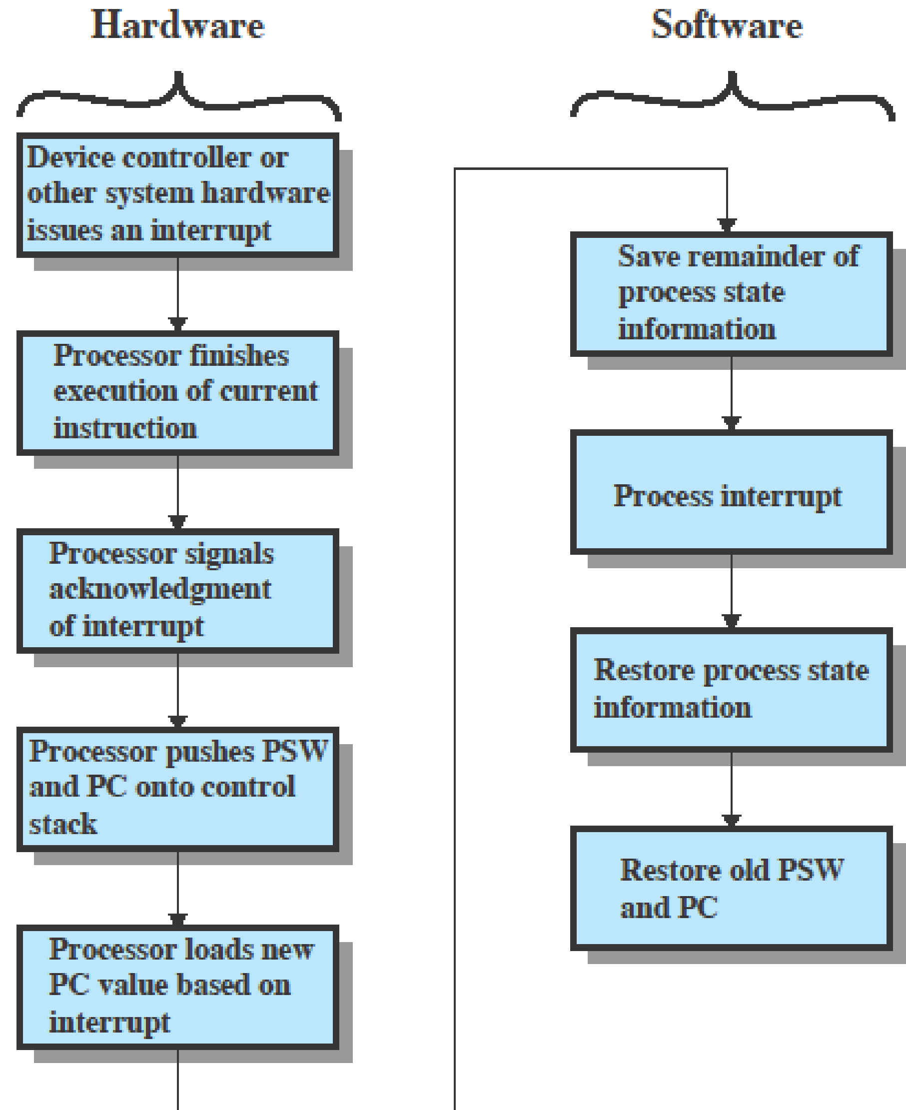
(a) Interrupt occurs after instruction at location N

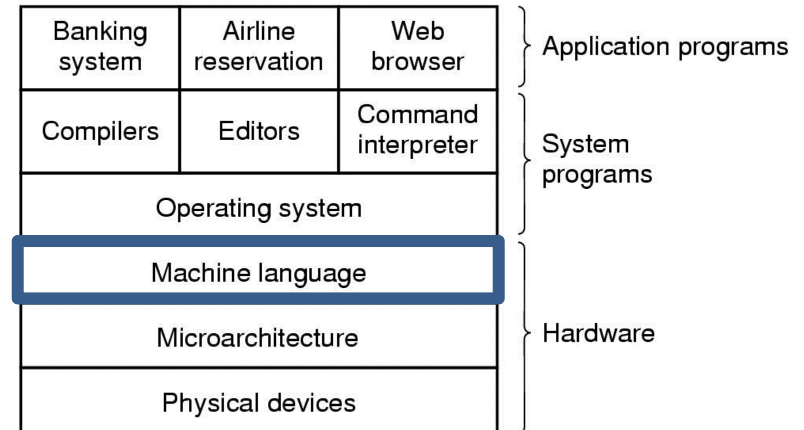


(b) Return from interrupt

Abarbeitung von Interrupts (2)

- Bei einem eintreffenden Interrupt übernimmt die Hardware einen Teil der Arbeit…





DIE MASCHINENSPRACHE AM BEISPIEL VON SPIM

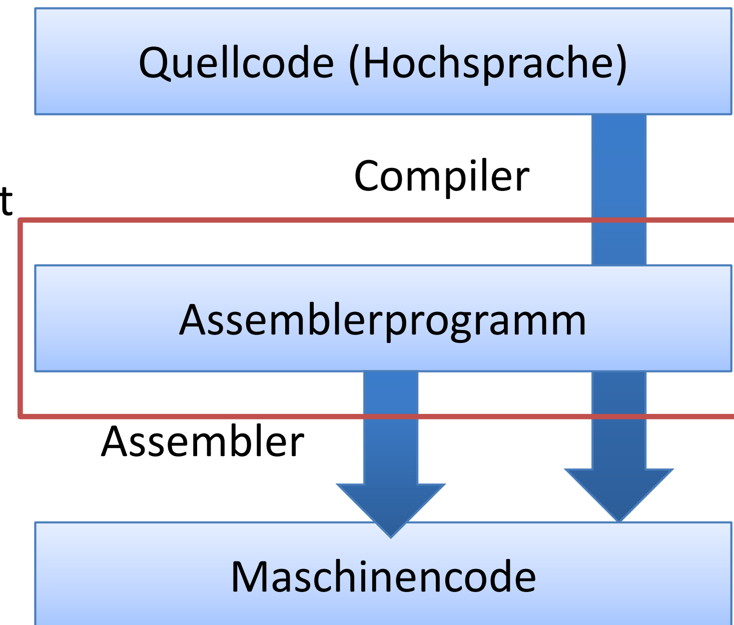
Einleitung & Motivation

Grundproblem:
- Maschinensprache für den Menschen höchst umständlich
- Maschinen können aber nur Maschinensprache (primitive Befehle) verstehen

Lösung:
- Menschen nutzen einen anderen Befehlssatz als Sprache L1 (Bsp.: C++)
- Maschinen verarbeiten eine Übersetzung (Compiler) bzw. Interpretation (Interpreter) von L1, hier als L0 bezeichnet.
- **Compiler**:
  - Vollständige Übersetzung des Programms in L1 zu L0
  - Quellprogramm in L1 wird verworfen
  - Zielprogramm in L0 wird in Speicher geladen und ausgeführt
- **Interpreter**:
  - Jede L1 Anweisung wird analysiert, dekodiert und unmittelbar in L0 ausgeführt

Vom Programm zum Maschinenprogramm

- Vereinfachter Ablauf:
  - Hochsprache (C++, Java,...) wird mittels Compiler in eine Assemblersprache übersetzt
  - Compiler analysiert das Programm und erzeugt Assemblercode
    - Für Menschen verständlicher Maschinen-code
  - Assembler übersetz Assemblercode in Maschinencode
    - Maschinensprache bestehen aus 0 und 1



Fokus nun auf Maschinenebene & Assemblerprogrammierung!

Assemblerprogrammierung

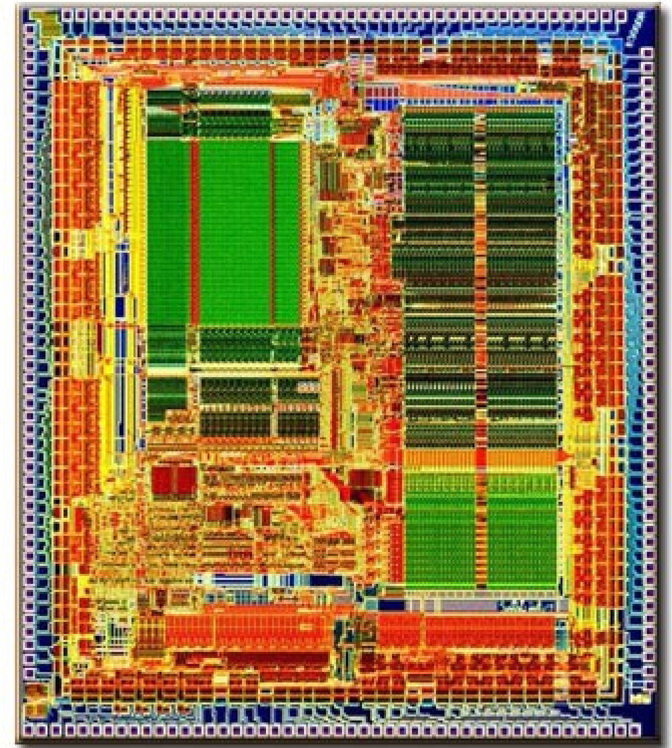
- Assemblersprache:
  - Hardwarenahe Programmiersprache
  - Wird von Assembler direkt in ausführbaren Maschinencode umgewandelt
  - Alle Verarbeitungsmöglichkeiten des Mikrokontrollers werden genutzt
  - Hardwarekomponenten können direkt angesteuert werden
  - Erlauben Namen für Instruktionen, Speicherplätze, Sprungmarken, etc.
  - I.d.R. effizient, geringer Speicherplatzbedarf
  - Anwendung:
    - Gerätetreiber
    - Eingebettete Systeme
    - Echtzeitsysteme
    - Neue Hardware (Keine Bibliotheken vorhanden)
    - Programmierung von Mikroprozessoren (Bsp.: MIPS)

```
lw      $t0,($a0)
add     $t0, $t1,$t2
sw      $t0,($a0)
jr      $ra
```

Beispiel: Assemblercode

MIPS-Architektur

- **MIPS-Architektur** (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages)
  - Mikroprozessor ohne Pipeline-Sperren
  - RISC-Prozessorarchitektur
  - 1981: von John Hennessy entwickelt (Standford-Universität)
  - 1984: Weiterentwicklung durch MIPS Computer Systems Inc., heute MIPS Technologies.
  - 1991: Mit R4000 auf 64 Bit erweitert (ursprünglich 32-Bit)
  - Früher: vor allem in klassischen Workstations und Servern
  - Heute: Haupteinsatzbereich im Bereich Eingebettete Systeme



MIPS R3000

RISC / CISC

Unterscheidung zwischen:

- **RISC** = Reduced Instruction Set Computer
- **CISC** = Complex Instruction Set Computer

Historie:

- Beginn: Mikroprozessoren waren alle RISC Prozessoren.
- Dann: Integration immer weiterer Funktionen
  => Immer komplexere Prozessor Designs
- Analyse zeigte:
  - ca. 95% aller Maschinenbefehle in Programmen sind einfache Befehle
- Also:
  - RISC: Entfernen komplexer Befehle (durch Software realisiert)
    - schnellere Ausführung von Befehlen (keine Interpretation nötig)
  - CISC: Prozessoren haben oft einen RISC Kern
    - Komplexe CISC-Instruktionen werden in Folge von RISC-Instruktionen übersetzt.

CISC vs. RISC

CISC CPUs:

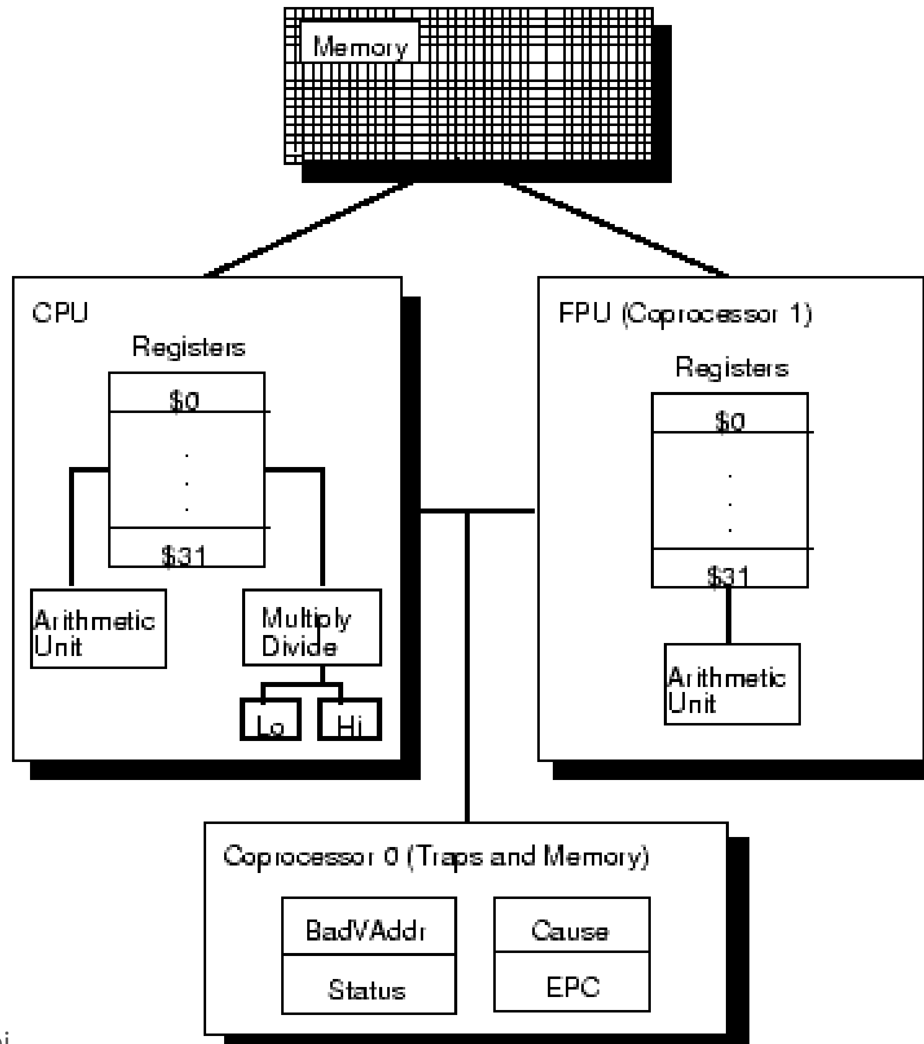
- Motorola 68000, Zilog Z80, Intel x86 Familie
- ab Pentium 486 allerdings mit RISC Kern und vorgeschaltetem Übersetzer in RISC Befehle.

Systeme mit RISC CPU:

- Leistungsstarke eingebettete Systeme (Druckern, Router)
- Workstations
- Supercomputern der Spitzenklasse
- Beispiele:
  - Cisco: Router und Switches bis zur Enterprise-Klasse
  - IBM: Supercomputer, Mittelklasse-Server, Workstations
  - Nintendo: Gamecube, Wii Spielkonsolen
  - Microsoft: Xbox 360 Spielkonsole
  - Motorola: Bordcomputer für PKW und andere Fahrzeuge

Struktur des MIPS Prozessors

- Nicht alle Funktionen sind im Prozessor selbst realisiert, sondern in Coprozessoren ausgelagert.



MIPS-Register (32 Bit breit)

Name	Nummer	Verwendung
$zero	0	Enthält den Wert 0, kann nicht verändert werden.
$at	1	temporäres Assemblerregister. (Nutzung durch Assembler)
$v0	2	Funktionsergebnisse 1 und 2 auch für Zwischenergebnisse
$v1	3	
$a0	4	Argumente 1 bis 4 für den Prozeduraufruf
$a1	5	
$a2	6	
$a3	7	
$t0,…,$t7	8-15	temporäre Variablen 1-8. Können von aufgerufenen Prozeduren verändert werden.

MIPS-Register 16-31

Name	Nummer	Verwendung
$s0,…, $s7	16 … 23	langlebige Variablen 1-8. Dürfen von aufgerufenen Prozeduren nicht verändert werden.
$t8,$t9	24,25	temporäre Variablen 9 und 10. Können von aufgerufenen Prozeduren verändert werden.
$k0, k1	26,27	Kernel-Register 1 und 2. Reserviert für Betriebssystem, wird bei Unterbrechungen verwendet.
$gp	28	Zeiger auf Datensegment
$sp	29	Stackpointer Zeigt auf das erste freie Element des Stacks.
$fp	30	Framepointer, Zeiger auf den Prozedurrahmen
$ra	31	Return Adresse

Adressierung

- Adressierung: byteweise!
- Wort mit Adresse n => Nächstes Wort: Adresse n+4

Adresse	...	0xA8	0xA9	0xAA	0xAB	0xAC	0xAD	0xAE	0xAF	
Bytegrenze	...	byte 168	byte 169	byte 170	byte 171	byte 172	byte 173	byte 174	byte 175	
Wortgrenze	...	word 42				word 43				...

Load and Store Architektur

- SPIM hat eine Load-Store Architektur
  - Daten müssen erst aus dem Hauptspeicher in Register geladen werden (load), bevor sie verarbeitet werden können.
  - Ergebnisse müssen aus Registern wieder in den Hauptspeicher geschrieben werden (store).
- Es gibt keine Befehle, die Daten direkt aus dem Hauptspeicher verarbeiten.

Trennung von Programm und Daten

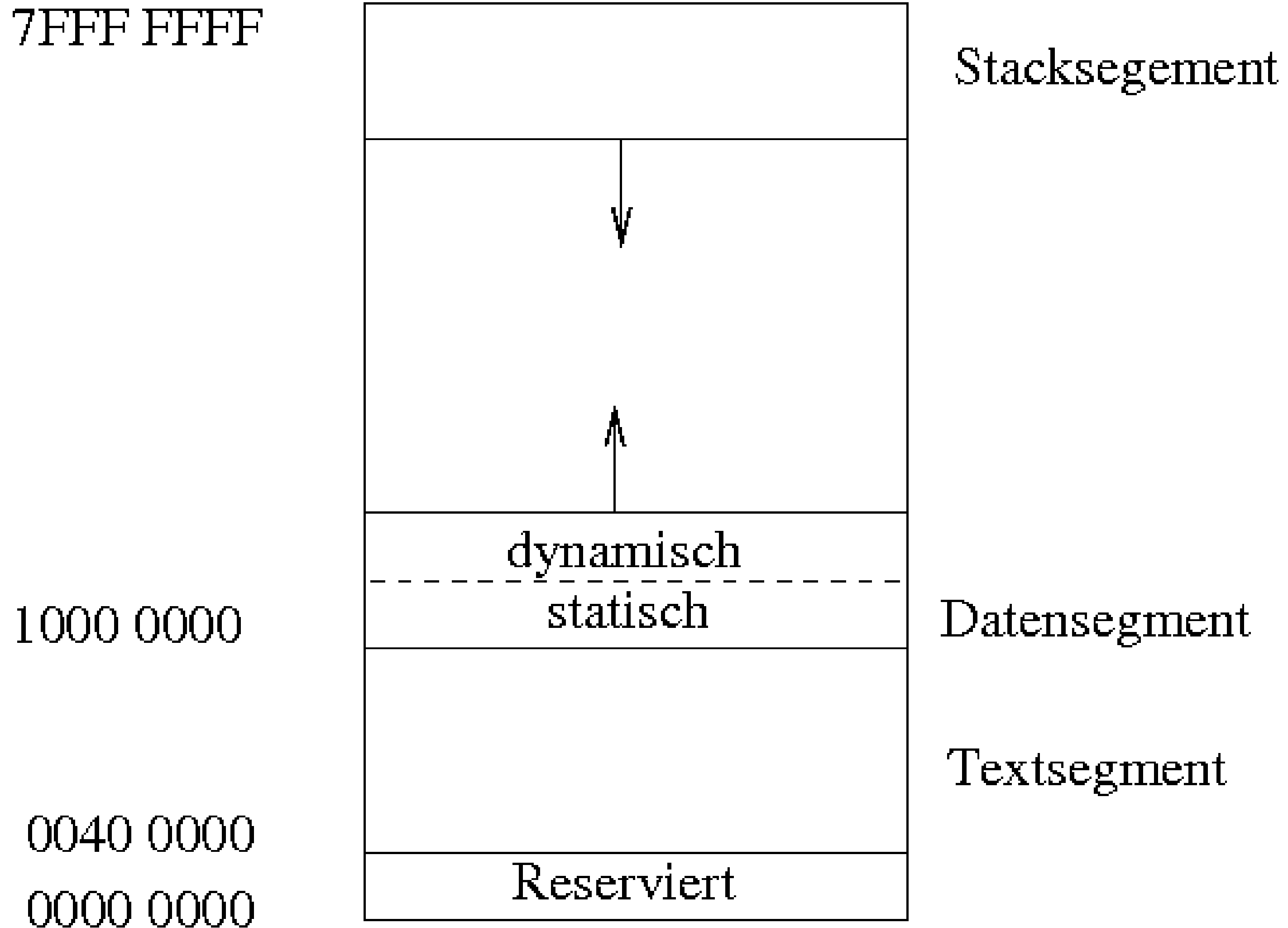
Grundprinzip (Von-Neumann):

- Gemeinsamer Speicher für Daten und Programme

- SPIM teilt den Hauptspeicher in **Segmente,** um Konflikte zu vermeiden:
  - **Datensegment**
    - Speicherplatz für Programmdaten (Konstanten, Variablen, Zeichenketten, ...)
  - **Textsegment**
    - Speicherplatz für das **Programm**.
  - **Stacksegment**
    - Speicherplatz für den Stack.

- Es gibt auch noch jeweils ein Text- und Datensegment für das Betriebssystem:
  - Unterscheidung zwischen User- und Kernel- Text/Data Segement

Speicherlayout



Assemblerprogramm, Beispiel

- Das Programm berechnet den Umfang des Dreiecks mit den Kanten x, y, z

```
            .data                      ← Datensegment
x:          .word 12
y:          .word 14
z:          .word 5
U:          .word 0
                                        ← Textsegment
                                        ← Kommentarzeichen
            .text
main:       lw $t0, x            # $t0 := x
            lw $t1, y            # $t1 := y
            lw $t2, z            # $t2 := z
            add $t0, $t0, $t1    # $t0 := x+y
            add $t0, $t0, $t2    # $t0 := x+y+z
            sw $t0, U            # U := x+y+z
            li $v0, 10           # EXIT
            syscall
```

Erklärung

- Direktiven:
  - .data (.text):
    - Kennzeichnet den Start des Datensegments (Textsegments)
  - .word:
    - sorgt für Reservierung von Speicherplatz
    - hier für die Variablen x,y,z,U. Jeweils ein Wort (32 Bit) wird reserviert.
    - Inhalt wird mit den Zahlen 12, 14, 5 und 0 initialisiert.

- (Pseudo-) Befehle:
  - **`lw $t0,x`** lädt den Inhalt von x in das Register $t0. (SPIM realisiert Load-Store Architektur)
  - **`add $t0,$t0,$t1`** addiert den Inhalt von $t0 zu $t1 und speichert das Resultat wieder in $t0.
  - **`sw $t0,U`** speichert den Inhalt von $t0 in den Speicherplatz, der U zugewiesen ist.
  - **`li $v0,10`** und syscall halten das Programm an.

Integer Daten anlegen

- Mit der Direktive

  `.word Wert1 Wert2` …

  werden Folgen von 32-Bit Integern angelegt (z.B. nützlich zur Speicherung von Feldern...)

- Beispiel:

  `x: .word 256 0x100`

  – reserviert im Speicher 2*32 Bit und schreibt in beide den Wert 256 hinein (0x... bedeutet hexadezimal).
  – **`x`** ist eine *Marke*. Man kann damit auf den ersten Wert zugreifen.
  – Mit **`x+4`** kann man auf den **zweiten** Wert zugreifen.

Aufbau einer Assembler-Befehlszeile

```
<Marke>: <Befehl> <Arg 1> <Arg 2> <Arg 3> #<Kommentar>
```

Oder mit Kommas

```
<Marke>: <Befehl> <Arg 1>,<Arg 2>,<Arg 3> #<Kommentar>
```

In der Regel 1 – 3 Argumente:

- Fast alle arithm. Befehle 3: 1 Ziel + 2 Quellen
- Befehle für Datenübertragung zw. Prozessor und HS: 2 Arg
- Treten i*n folgender Reihenfolge auf:*
    1.) Register des Hauptprozessors, zuerst das Zielregister,
    2.) Register des Coprozessors,
    3.) Adressen, Werte oder Marken

Notation der Befehle

Befehl	Argumente	Wirkung	Erläuterung
div	Rd,Rs1,Rs2	RD=INT(Rs1/Rs2)	divide
li	Rd,Imm	Rd=Imm	Load Immediate

- Erläuterungen:
  - Rd = destination register (Zielregister)
  - Rs1 = source register (Quellregister)
  - Imm = irgendeine Zahl

- Beispiele:

```
div $t0,$t1,$t2
```

dividiere den Inhalt von $t1 durch den Inhalt von $t2 und speichere das Ergebnis ins Zielregister $t0.

Ladebefehl und Adressierung

Befehl	Argumente	Wirkung	Erläuterung
lw	Rd, Adr	RD=MEM[Adr]	Load word

- **`(Rs)`**: Der Wert steht im Hauptspeicher an der Adresse, die im Register **`Rs`** steht (Register-indirekt)
- **`label`** oder **`label+Konstante`**: Der Wert steht im Hauptspeicher an der Stelle, die für **`label`** reserviert wurde, bzw. nachdem Konstante dazu addiert wurde (direkt).
- **`label(Rs)`** oder **`label+Konstante(Rs)`**: Der Wert steht im Hauptspeicher an der Stelle, die für **`label`** reserviert wurde **+** **`Konstante + Inhalt`** von Register **`Rs`** (indexiert).

Ladebefehl und Adressierung

Befehl	Argumente	Wirkung	Erläuterung
la	Rd, Label	RD=Adr(Label)	Load address

- `la` lädt die **Adresse** auf die das Label **label** zeigt in das Zielregister Rd.
- Zum Vergleich: `lw` lädt die **Daten** aus der angegeben Adresse `Adr` in das Zielregister `Rd`.

Beispiel für Adressierung

```
        .data
var:    .word 20, 4, 22, 25, 7
        .text
main:   lw $t1, var                  # $t1 enthaelt "20"
                                     # (direkte Adr.)
        la $t1, var                  # Adr. von "20" in $t1
```

Speicherbefehle

- speichern Registerinhalte zurück in den Hauptspeicher.

Befehl	Argumente	Wirkung	Erläuterung
sw	Rs,Adr	MEM[Adr]:=Rs	store word
sb	Rs,Adr	MEM[Adr]:=Rs MOD 256	store byte (die letzten 8 Bit)
sh	Rs,Adr	MEM[Adr]:=Rs MOD 2^{16}	store halfword(die letzten 16 Bit)
sd	Rs,Adr	sw Rs,Adr sw Rd+1,Adr+4	Store double word

Register-Transfer Befehle

- **move**: Kopieren zwischen Registern.
- **li**: Direktes laden des Wertes in ein Register.
- **lui**: Lädt den Wert in die oberen 16 Bits des Registers (und macht die unteren 16 Bits zu 0).

Befehl	Argumente	Wirkung	Erläuterung
move	Rd,Rs	Rd :=Rs	move
li	Rd, Imm	Rd := Imm	load immediate
lui	Rd,Imm	Rd := Imm * 2^{16}	Load upper immediate

Addition und Subtraktion

- Ein Überlauf (Overflow) bewirkt den Aufruf eines Exception Handlers (ähnlich catch in Java).
- Es gibt auch arithmetische Befehle, die Überläufe ignorieren.

- **Weitere arithmetische Befehle:**
  - **div, mult** (in Versionen mit und ohne overflow, sowie mit und ohne Vorzeichen)
  - **neg** (Zahl negieren), **abs** (Absolutbetrag), **rem** (Rest)

Befehl	Argumente	Wirkung	Erläuterung
add	Rd,Rs1,Rs2	Rd := Rs1+Rs2	addition (mit overflow)
addi	Rd,Rs1,Imm	Rd := Rs1+Imm	addition immediate (mit overflow)
sub	Rd,Rs1,Rs2	Rd := Rs1-Rs2	subtract(mit overflow)

VIELEN DANK