Digitaltechnik 2

Hochschule Esslingen – Fakultät Informationstechnik

Flandernstrasse 101, 73732 Esslingen

Prof. Dipl.-Ing. Reinhard Keller
F1.457
Tel: 0711-397-4161
Fax: 0711-397-4214
Email:
Reinhard.Keller@hs-esslingen.de
Web:
http://www.hs-esslingen.de/
mitarbeiter/Reinhard_Keller

Prof. Dr.-Ing. Walter Lindermeir
F1.451
Tel: 0711-397-4230
Fax: 0711-397-4214
Email:
Walter.Lindermeir@hs-esslingen.de
Web:
http://www.hs-esslingen.de/
mitarbeiter/Walter_Lindermeir

Prof. Dr.-Ing. Werner Zimmermann
F1.351
Tel: 0711-397-4227
Fax: 0711-397-4214
Email:
Werner.Zimmermann@hs-esslingen
Web:
http://www.hs-esslingen.de/
mitarbeiter/Werner_Zimmermann

Übersicht

Vorlesung

Kap. 1: Einführung

Kap. 2: Schaltwerke und Endliche Automaten

Kap. 3: Aufbau von Halbleiterspeichern und Speicherzugriffe

Kap. 4: Rechnerarchitektur: Aufbau einer CPU

Kap. 5: Rechnerperipherie

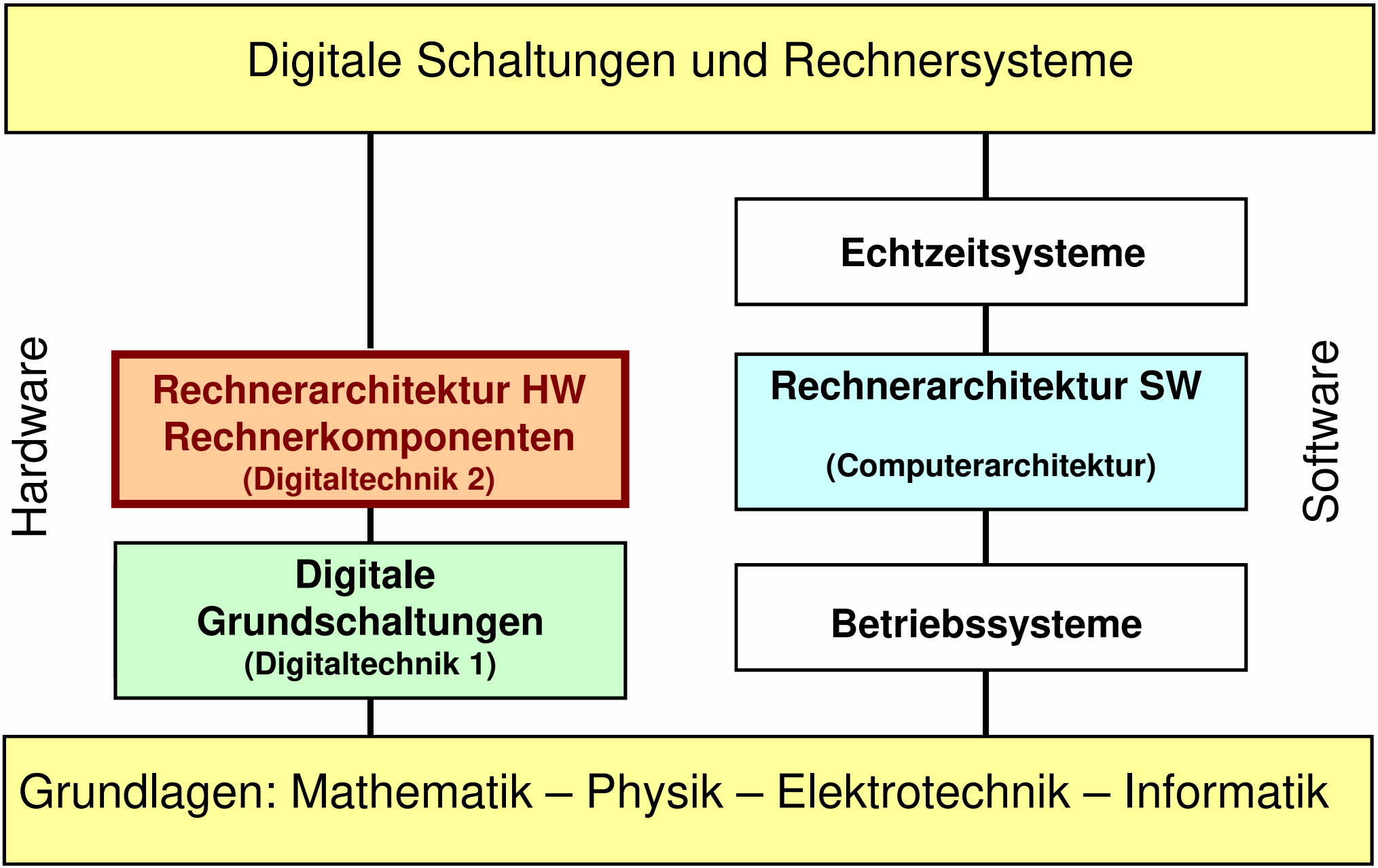
Labor

Labor 1: Endliche Automaten in VHDL

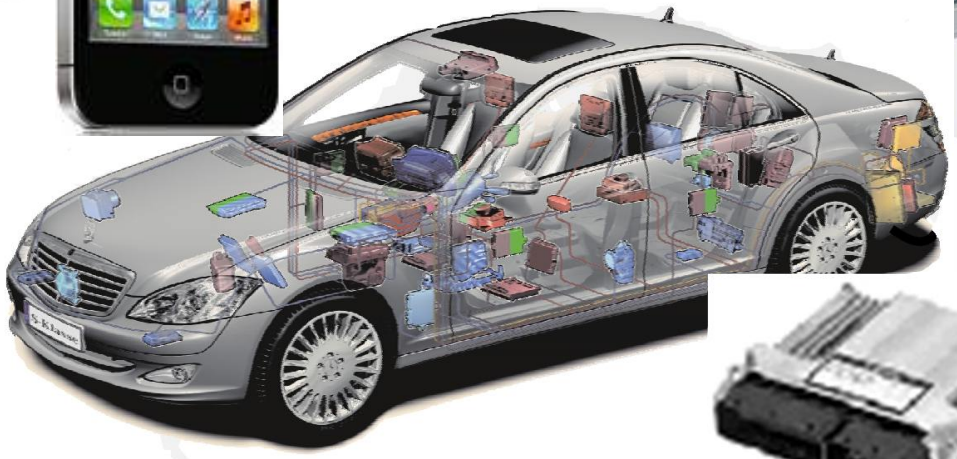
Labor 2: Steuerwerk für Speicherzugriff

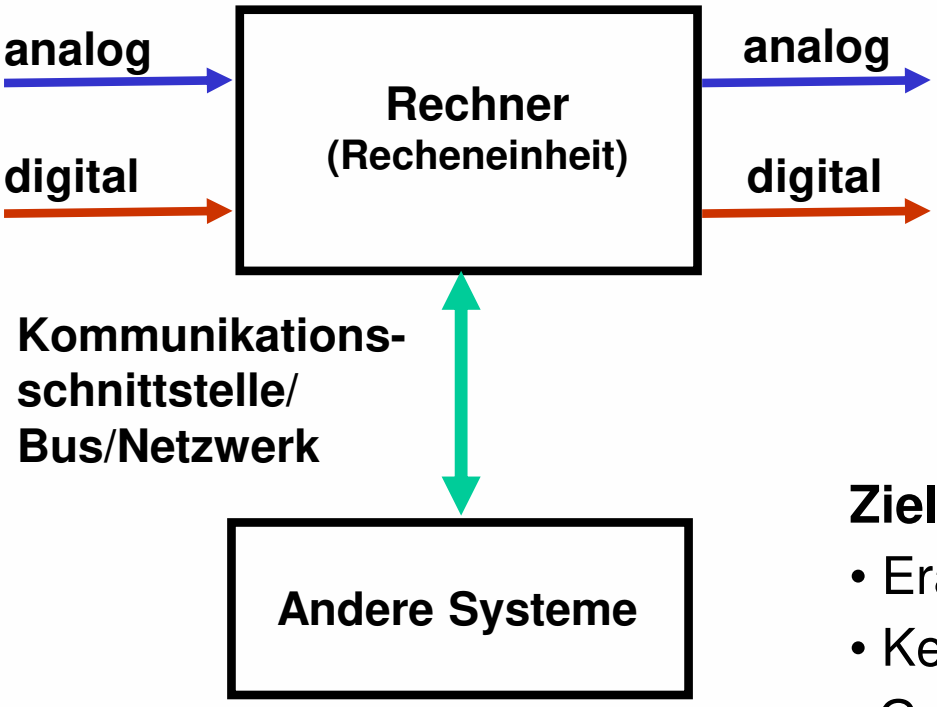
Labor 3: Aufbau einer CPU

Digitaltechnik und Computerarchitektur im Studienmodell



Systemstruktur

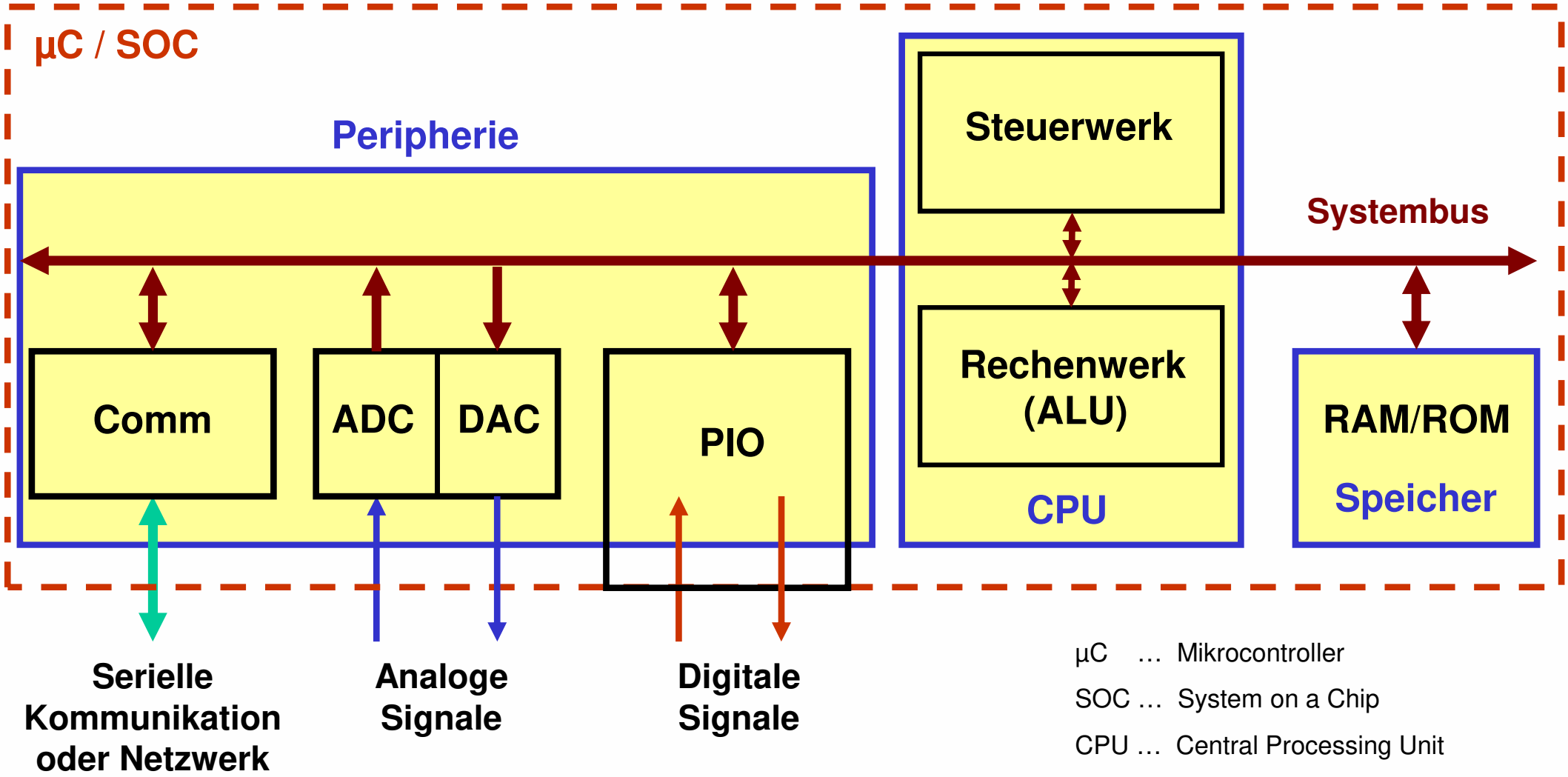




Ziel der Vorlesung:

- Erarbeitung von Grundlagen: Endliche Automaten
- Kennenlernen der Funktionsweise der Grundkomponenten eines Rechners (Architektur der Hardware): CPU, Speicher, Peripherie

Prinzip-Aufbau eines Rechners
(Architektur der Hardware)



µC … Mikrocontroller

SOC … System on a Chip

CPU … Central Processing Unit

Kap. 2: Endliche Automaten (Entwurfsebene)

➔ **Übersicht:**

2.1 Beispiel 2-1: Anlauf-Steuerung

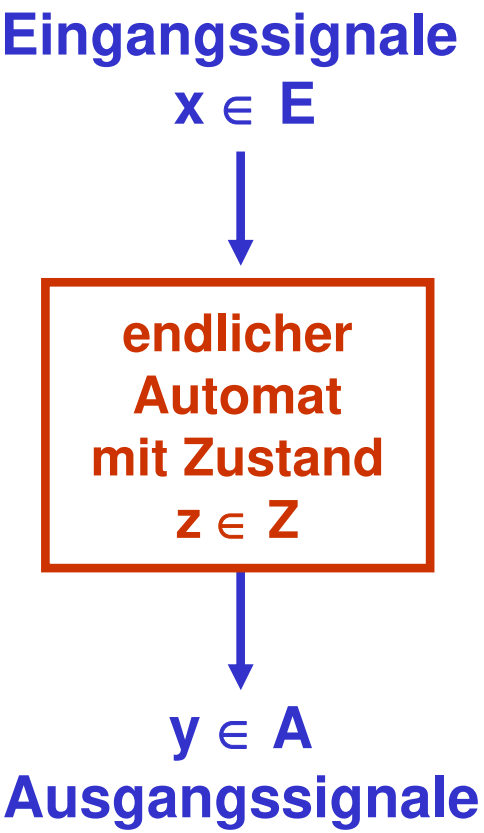
➔ **Beschreibung einer Anwendung als *Endlicher [*1] Automat* (*Finite State Machine FSM*) unabhängig von der Art der Realisierung** (abstraktes Modell)

Viele Realisierungsarten:

- **in Hardware**
- mit Flipflops und Gattern in diskreter Logik
- mit programmierbarer Logik (PLD, CPLD, FPGA)
- mit Hardwarebeschreibungssprachen (VHDL, VERILOG)
- . . .
- **in Software**
- mit C/C++ auf Mikrocontrollern ohne Betriebssystem
- C/C++, Java, … mit Betriebssystem, z.B. OSEK OS
- als SPS-Programm nach IEC 61131
- . . .

Formalisierte Beschreibung:

- Menge der Eingangswerte E und Ausgangswerte A
- Menge der Zustände Z
- Übergangsfunktion $(z)^{(k+1)} = g((z)^{(k)}, (x))$
- Ausgangsfunktionen $(y) = f((z)^{(k)}, (x))$
- Anfangszustand $(z)^{(0)}$



[*1] **Hinweis: Endlicher Automat → Anzahl der Zustände, Ein- und Ausgangswerte endlich.**

2.1 Beispiel 2-1: Anlauf-Steuerung

➔ Abstrakte Beschreibung der Funktion im fehlerfreien Betrieb (ohne Berücksichtigung der Realisierung)

Das Anlaufen einer Maschine soll von einer Anlauf-Steuerung übernommen werden, die sowohl den Antrieb ansteuert als auch den Bediener über eine grüne Kontroll-Leuchte (LED) informiert. Es muss folgender Ablauf eingehalten werden:

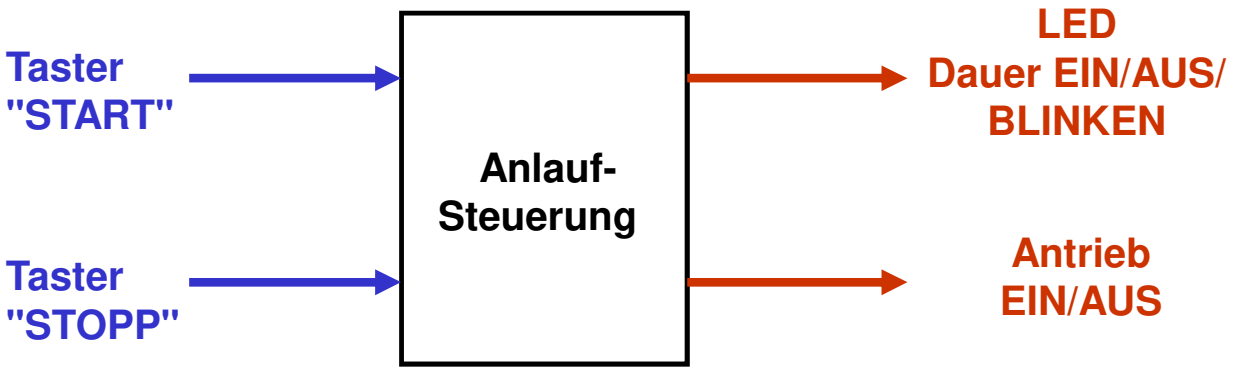
- **Bei stehender Maschine (LED aus) muss der Bediener zunächst kurz den Taster "START" betätigen.**
- **Danach erfolgt eine "Anlauf-Warnung", bei der die LED blinkt.**
- **Nach dem Ablauf der Mindestwarnzeit muss der Bediener den Taster "START" ein zweites Mal betätigen, um den Antrieb einzuschalten (LED ein).**
- **Durch Betätigen des Tasters "STOPP" wird der Antrieb immer sofort ausgeschaltet.**

Zeitbedingungen für Taster „START“:

… muss nach dem ersten Betätigen spätestens nach T_{2max} losgelassen werden

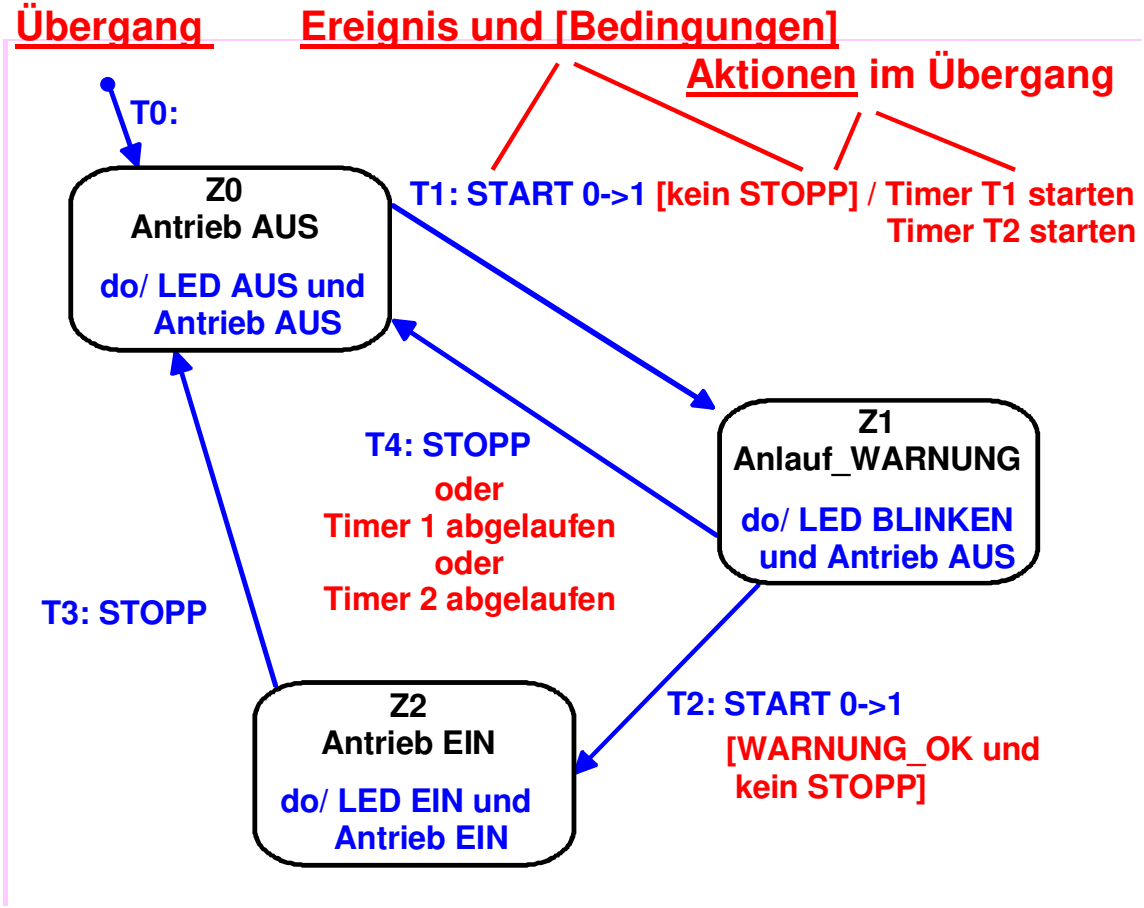
… darf frühestens nach T_{1min} , aber spätestens bis T_{1max} zum zweiten Mal betätigt werden, sonst bleibt der Antrieb abgeschaltet.

Blockschaltbild Antriebssteuerung:



2.1 Beispiel 2-1: Anlauf-Steuerung

➔ **Zustandorientierte Beschreibung des Verhaltens im fehlerfreien Betrieb mittels Zustandsdiagramm in der abstrakten Notation der UML** (Unified Modeling Language)



- ➔ Übergänge (Transitions) werden durch Ereignisse (Events) ausgelöst, Zusatzbedingungen möglich
- ➔ Zeitschranken als Bedingungen möglich

Bedingungen (Guards)

Zeitfenster für zweites Betätigen von START:

WARNUNG_OK: $T_{1min} \leq$ Timer $T_1 \leq T_{1max}$

T1 abgelaufen: Timer $T_1 > T_{1max}$

Maximaldauer für Betätigen von START:

T2 abgelaufen: Timer $T > T_{2max}$

Kein STOPP (in Übergang T1 und T2)

Ereignis "START": Taster START wird betätigt (Flanke, Übergang 0 → 1)

Ereignis "STOPP": Taster STOPP ist betätigt (Statisch, Signal STOPP = 1)

Anmerkungen:

- Anlaufen nur unter strikter Einhaltung des Ablaufs zulässig
- Antrieb AUS, sobald Taster STOPP betätigt wird (keine zusätzliche Bedingung)

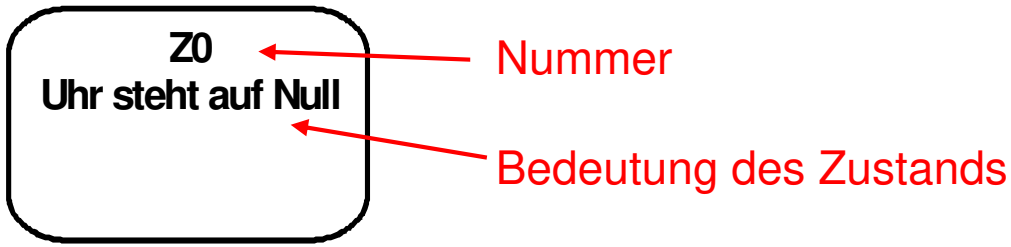
Zustandsorientierte Beschreibung mit einem Automatenmodell:

➔ **Zustände**

Zu jedem beliebigen Zeitpunkt befindet sich ein endlicher Automat in genau einem Zustand (engl. State). Die Zustände sind geeignet zu definieren:

- **Nach dem Eintreten in einen Zustand bleibt der Automat so lange in diesem Zustand, bis ein definiertes Ereignis zu einem Übergang in einen anderen Zustand führt. Der Automat verweilt in einem Zustand stets für eine Zeit t > 0, i.d.R. für mindestens einen Schritt, auch bei idealer Lösung.**
- **Im Diagramm wird Zuständen i.d.R. eine laufende Nummer und vorzugsweise ein charakterisierender Text zugewiesen (siehe Beispiele). Häufig eignet sich eine Beschreibung wie "Warten auf ... ", da in jedem Zustand bis zum Eintreten bestimmter Ereignisse verweilt wird.**
- **Die Anzahl der Zustände eines Systems muss überschaubar gehalten werden. Bei größeren Systemen erfolgt eine hierarchische Strukturierung der Zustände.**

Graphische Darstellung eines Zustandes im Zustandsdiagramm (Notation der UML)



➔ Übergänge: Ereignisse und Bedingungen

Übergänge (Kanten, engl. transitions) in einen anderen Zustand werden stets von Ereignissen (engl. events) ausgelöst und graphisch durch einen Pfeil vom bisherigen in den neuen Zustand repräsentiert.

- Ein Ereignis tritt zu einem bestimmten Zeitpunkt auf, der durch ein bestimmtes Eingangssignal (UML: trigger) bestimmt ist, und kann an zusätzliche Bedingungen geknüpft sein (UML: guards) .
- Bei Hardwareautomaten ist das Ereignis häufig die aktive Flanke eines Taktsignals, die Bedingung eine bestimmte Wertekombination der Eingangssignale. Man spricht dann von einem synchronen Automaten und verzichtet oft auf die explizite Darstellung des Taktsignals.
- Bei Softwareautomaten kann das Taktsignal durch periodisches Aufrufen der Automaten-Funktion simuliert werden. Andere Ereignis-Auslöser können das Klicken der Maus, das Eintreffen eines Hardwaresignals (Interrupt), einer empfangenen Botschaft bei der Datenübertragung usw. sein.

Graphische Darstellung eines Übergangs im Zustandsdiagramm (UML)

Aktionen und Aktivitäten

Erweiterte Form der von Hardwareautomaten bekannten Mealy und Moore-Ausgänge:

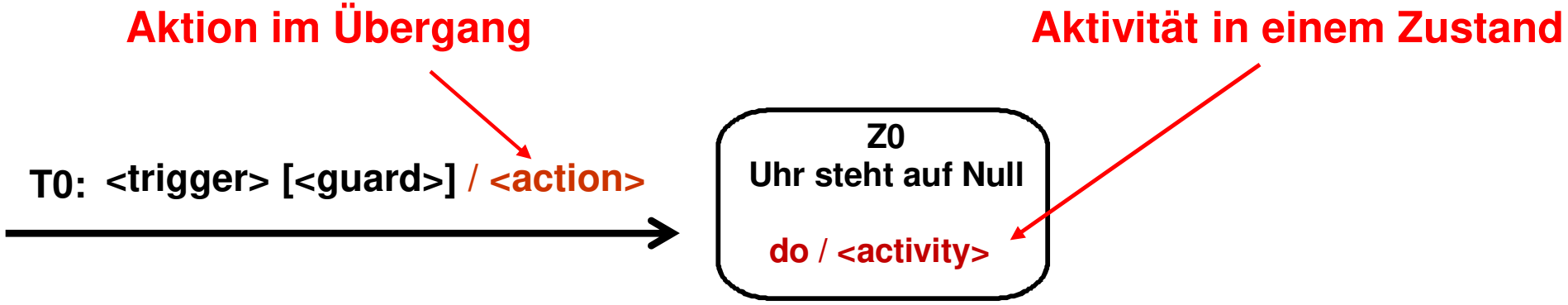
- Eine Aktion ist ein einmaliger Vorgang, der in einem Übergang stattfindet

(Dauer bei idealer Lösung: t = 0, Beispiel: Einschalten eines Antriebs, entspricht in der Hardware einem kurzen Impulssignal)

- Eine Aktivität ist ein andauernder Vorgang, der z.B. in einem bestimmten Zustand mit einer Dauer t > 0 auch bei idealer Lösung ausgeführt wird.

(Beispiel: Einschalten einer LED oder Ausgabe eines Signaltons, entspricht in der Hardware einem Dauersignal)

- Die Darstellung erfolgt gemäß UML nach folgender Konvention:



Vorgehensweise bei der Realisierung eines Automaten in Software:

(1) Gesamte Funktion des Automaten in einem Modul, Übergangs- und Ausgangsfunktion innerhalb einer C-Funktion (C++/Java Methode) implementiert

(2) Zustand (und evtl. Ein- und Ausgangssignale) wird innerhalb des Moduls statisch lokal als Enumerationstyp (Aufzählungstyp) definiert („static enum“)

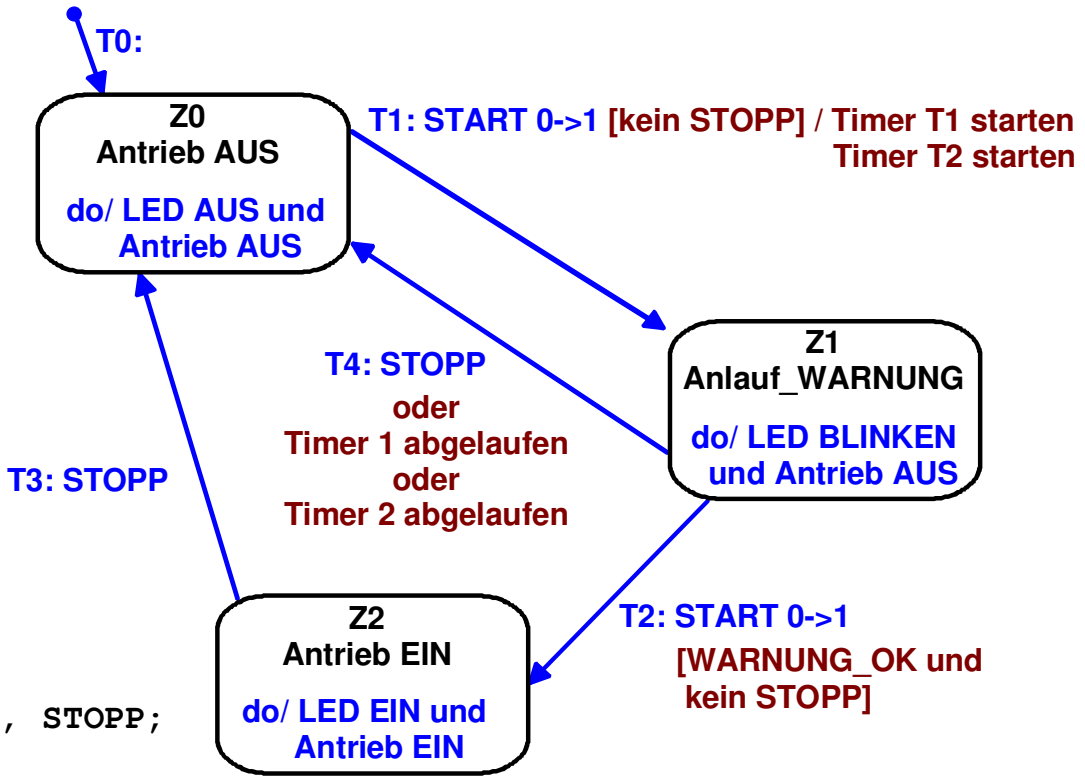
(3) Übergangsfunktion wird durch eine „switch-case“-Anweisung realisiert, die mit jedem Automaten-Takt durchlaufen wird, Übergangsbedingungen mit „if“

(4) Ausgangsfunktion je nach geforderter Funktionalität:
- „Actions“ im entsprechenden case/if-Zweig der Übergangsfunktion, die den zugehörigen Übergang bewirkt
- „Activities“ in einer zweiten „switch-case“-Anweisung, die ebenfalls mit jedem Automaten-Takt durchlaufen werden muss.

(5) Falls Hardware-Signale verarbeitet werden, erfolgt das
- Aktualisieren der Eingangssignale zu Beginn der C-Funktion
- Aktualisieren der Ausgangssignale am Ende der C-Funktion

(6) Takt des Automaten
- Einfachster Fall: C-Funktion in einer Endlosschleife (Takt = Schleifendurchlaufzeit)
- Besser: Takt durch Timer erzeugt, der die C-Funktion zyklisch aufruft
- Oder: Eingangssignal-Ereignis ruft C-Funktion auf (z.B. Maus-Klick)

➔ Beispiel: Anlauf-Steuerung in C




```
// Zustandsautomat für Anlauf-Steuerung
#include <stdio.h>
#include <string.h>

const long  WTime = 100;//ms Takt-Periode
extern void Wait(long WTime);// Taktgeber

// Externe Timer-Funktionen
void StartTimer(int timerNr);
void ResetTimer(int timerNr);
long GetTimer(int TimerNr);
const long T1min=4000, T1max= 10000, T2max=2000;

// Variablen für die Ein- und Ausgangssignale
enum AusgangssignalAntrieb {_AUS_, _EIN_} ANTRIEB;
enum AusgangssignalLed {AUS, BLINKEN, EIN} LED;
enum Eingangssignal {AUS_, FLANKE_0_1, EIN_} START, STOPP;

// Ext. Funktionen zum Schalten der Ausgangssignale ANTRIEB und LED
void SetAusgangAntrieb(AusgangssignalAntrieb antrieb);
void SetAusgangLed(AusgangssignalLed led);

// Ext. Funktion zum Einlesen der Eingangssignale START und STOPP
Eingangssignal GetEingangStart(void);
Eingangssignal GetEingangStopp(void);

//Abkürzungen für Bedingungen
#define WARNUNG_OK   ((T1min <= GetTimer(1)) && (GetTimer(1) <= T1max))
#define T1_EXPIRED   (GetTimer(1) > T1max)
#define T2_EXPIRED   (GetTimer(2) > T2max)
```

2.3 Realisierung eines Automaten in Software

```
void FSM_Anlauf(void) // Zustandsautomat (wird vom Hauptprogramm zyklisch aufgerufen)
{ // Variable für den Zustand mit Anfangszustand Antrieb_AUS
  static enum {Antrieb_AUS, Anlauf_WARNUNG, Antrieb_EIN} zustand = Antrieb_AUS;

  // Aktualisieren der Eingangssignale
  START = GetEingangStart();
  STOPP = GetEingangStopp();
```

Eingangssignale aktualisieren

```
  // Übergangsfunktion mit Zustandswechsel
    switch (zustand)
    {case (Antrieb_AUS):
          if ((START == FLANKE_0_1) && (STOPP == AUS_))
          { zustand = Anlauf_WARNUNG;      // Übergang T1
            StartTimer(1); StartTimer(2); // ACTION zum Übergang T1
          }
          break;
     case (Anlauf_WARNUNG):
           if ((START == FLANKE_0_1) && WARNUNG_OK && (STOPP == AUS_))
             zustand = Antrieb_EIN;        // Übergang T2 (hier ggf. auch ACTION)
          else if ((STOPP == EIN_) || T1_EXPIRED || T2_EXPIRED)
             zustand = Antrieb_AUS;        // Übergang T4 (hier ggf. auch ACTION)
          else if (START == AUS_)
              ResetTimer(2);               // ACTION zum Übergang T5, Zustand unverändert
          break;
     case (Antrieb_EIN):
          if (STOPP == EIN_)
                zustand = Antrieb_AUS;    // Übergang T3 (hier ggf. auch ACTION)
          break;
    }
```

Übergangsfunktion

(Transitions und Actions)

```
// Ausgangsfunktion: Ausführen der "ACTIVITIES"
  switch (zustand)
  {case (Antrieb_AUS):       ANTRIEB = _AUS_;
                             LED = AUS;
                             break;
   case (Anlauf_WARNUNG):    ANTRIEB = _AUS_;
                             LED = BLINKEN;
                             break;
   case (Antrieb_EIN):       ANTRIEB = _EIN_;
                             LED = EIN;
                             break;
  }
```

Ausgangsfunktion

(Activities)

```
  SetAusgangAntrieb(ANTRIEB);
  SetAusgangLed(LED);
```

Ausgangssignale aktualisieren

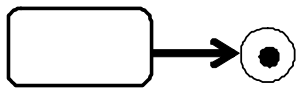
```
}
```

```
// Hauptprogramm
void main(void)
{    while ('TRUE')
     { FSM_Anlauf();
       Wait(WTime);
     }
}
```

Trigger

(Zyklischer Aufruf des Automaten)

2.4 Vergleich Hardware- und Software-Automaten

	Schaltwerk (Hardware-Automat)	**UML-Automat (Software-Automat)**
Auslösendes Ereignis (**Event**, Trigger) • löst Zustandsübergang aus	Aktive Taktflanke	Aufruf der Automaten-Funktion • periodisch (äquivalent zu Taktsignal) ODER • als Folge eines äußeren Ereignisses, z.B. Klicken der Maus, Tastendruck, Eintreffen einer Datenbotschaft, ...
Bedingungen (**Guards**) • muss erfüllt sein, damit Zustandsübergang erfolgt	Bedingungen = logische Verknüpfungen der Eingangssignale	
Ausgänge • Wirkung des Automaten nach außen	Ausgangssignale sind ständig vorhanden: • **Moore-Ausgang**, nur abhängig vom aktuellen Zustand, d.h. gehört zum Zustand und ist während eines Zustandes konstant. • **Mealy-Ausgang**, ist zusätzlich auch abhängig von den aktuellen Eingangssignalen, d.h. kann sich ändern, wenn sich der Zustand ändert oder wenn sich Eingangssignale ändern	Ständige Aktivitäten oder kurze Aktionen: • **Aktivität** wird ständig ausgeführt, d.h. gehört zum Zustand, darf aber trotzdem auch von Eingangssignalen abhängig sein • **Aktion** wird nur bei einem Zustandsübergang ausgeführt, d.h. gehört zum Zustandsübergang und wird einmalig ausgeführt, theoretisch unendlich kurz
Existenz	Funktion, solange Versorgungsspannung eingeschaltet ist	Kann gestartet und beendet werden  Kennzeichnung eines Endzustandes

Entry/ und Exit/ Aktionen

Häufige Situation:

Aufgaben, die grundsätzlich jedesmal einmalig auszuführen sind, wenn ein Zustand betreten oder wenn er verlassen wird.

Mögliche Realisierung:

„Action“ bei allen Zustandsübergängen in diesen Zustand oder vom Zustand weg.

Aber:

Sehr umständlich, wenn der Zustand sehr viele Zustandsübergänge hat.

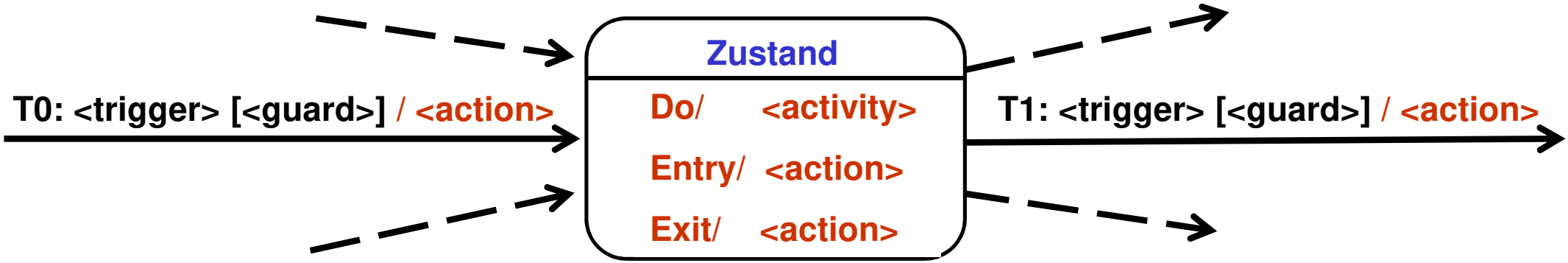
Bessere Möglichkeit bei UML-Zustandsautomaten :

Entry/ Action: **wird einmalig bei jedem Betreten des Zustands ausgeführt**

Exit/ Action: **wird einmalig bei jedem Verlassen des Zustands ausgeführt**

Zusätzlich zu der bekannten

Do/ Activity: **wird ständig ausgeführt, solange der Zustand vorhanden ist**



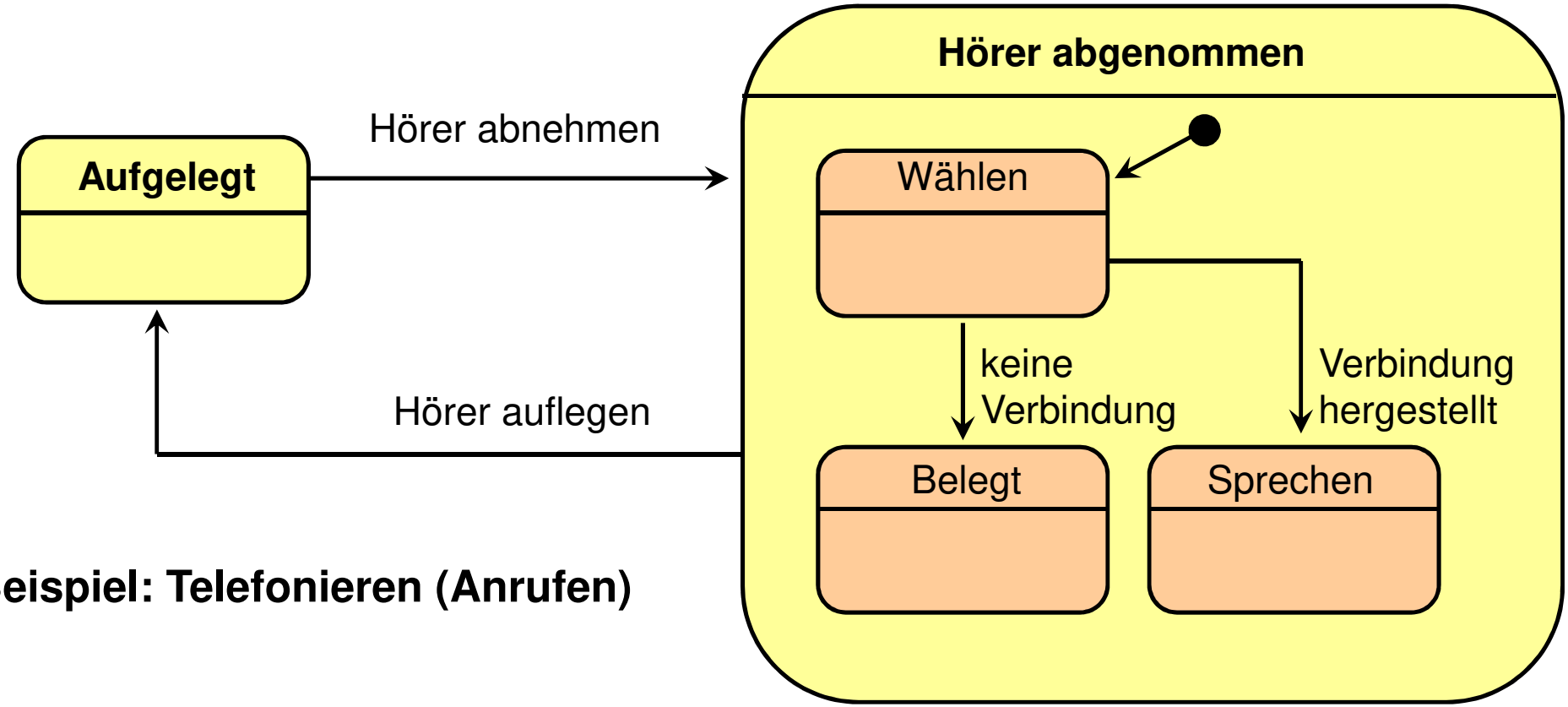
Hierarchische Automaten

Häufige Situation:

Zustandsautomat mit sehr vielen Zuständen

Möglichkeit bei UML-Zustandsautomaten :

Zustände zu Gruppen mit übergeordneter Bedeutung zusammenfassen und Hierarchiestufen (Super- und Subzustände bzw. Ober- und Unterzustände) einführen



Beispiel: Telefonieren (Anrufen)

Ergänzende Literatur zu Kapitel 2:

[1] Borucki, L.: Digitaltechnik, 5. Auflage.
B.G. Teubner, Stuttgart, 2000.
Seite 291 - 363

[2] Jeckle, M. u.a.: UML2 glasklar.
C. Hanser Verlag, München-Wien, 2004.
Seite 267-322

[3] Ashenden, P.: The Designers Guide to VHDL, 2nd Edition, Systems on Silicon,
Morgan Kaufmann, 2002.

[4] Balzert, H.: Lehrbuch der Software-Technik. Spektrum Akad. Verlag, 1996.
Seite 269 - 295

[5] Samek, M.: Practical Statecharts in C/C++. CMP Books, 2002.

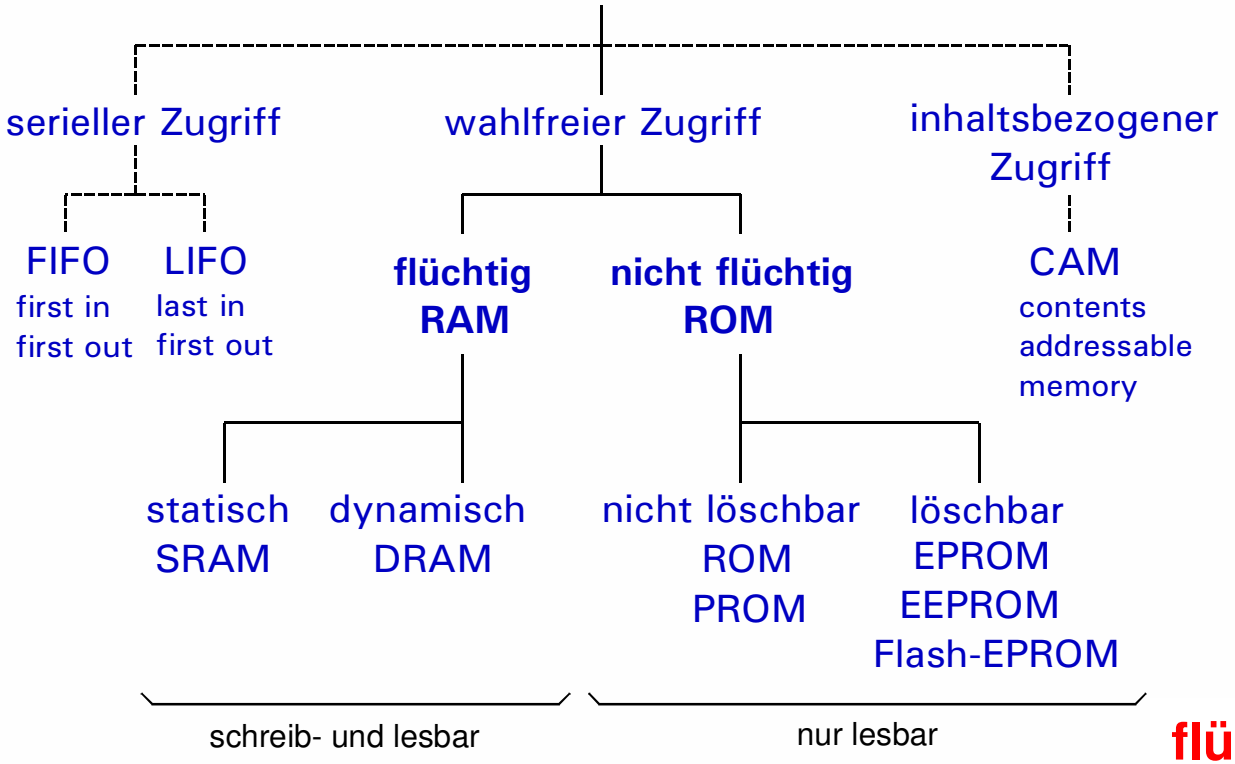
Kap. 3: Halbleiterspeicher

➔ **Übersicht:**

Zweck: Speichern von binären Befehlen oder Daten gesteuert durch CPU

- **Logische Organisation ähnlich einer Tabelle** → **Speicherzellen**
- **Inhalt einer Tabellenzeile** → **Speicherwort** (typ. 8 ... 64 bit)
- **Auswahl einer Tabellenzeile durch die Zeilennummer** → **Adresse, Adressierung**
- **Lesen** (Speicher → CPU) und **Schreiben** (CPU → Speicher) von Speicherworten

Ausführungsformen von Halbleiterspeichern



Wesentliche Begriffe:

Zugriff:
wahlfreier Zugriff über eine Adresse „Random Access"

Haltbarkeit der Daten:
Nicht flüchtiges oder flüchtiges Speichern

ROM (Read Only, Lesespeicher)
für Programmbefehle, Konstanten

RAM (Random Access, Schreib-/ /Lesespeicher) für variable Daten

flüchtig = volatile
nicht flüchtig = non volatile (NV)

Wichtige Ausführungsformen: ROM (Read-Only-Memory)

Nicht flüchtig, kann im Normalbetrieb nur gelesen werden

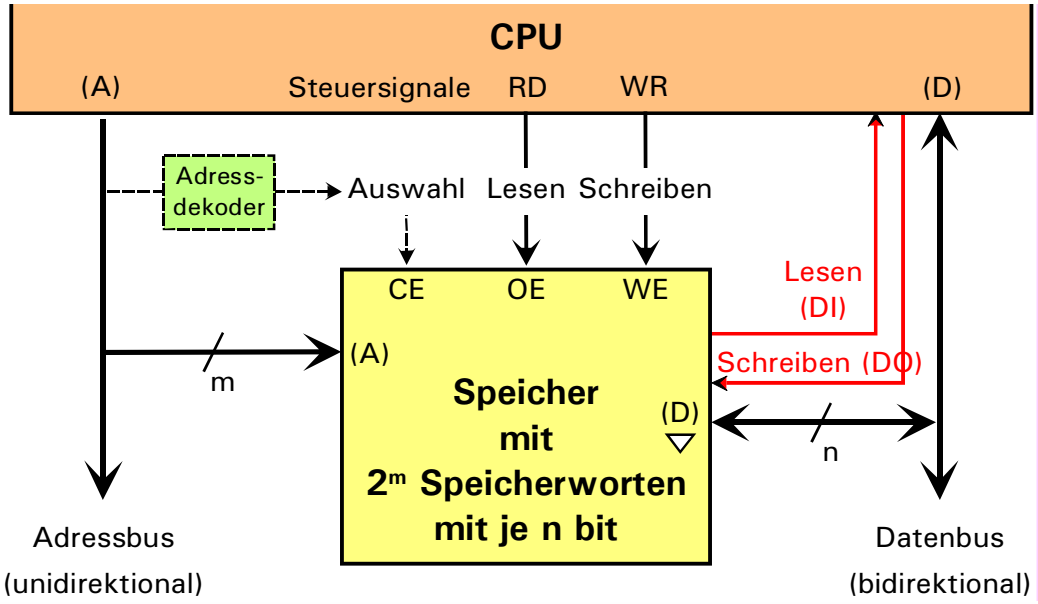
(M)ROM	Maskenprogrammiertes ROM Bei der Herstellung des Bausteins wird der Dateninhalt in der Maske des Bausteins festgeschrieben. Aufwand typ. 1 Transistor je Bit. Anwendung: Mikrocontroller-Programmspeicher für sehr hohe Stückzahlen und hohe Ansprüche, z.B. Safety-Anwendungen
PROM	Programmable ROM Einmalig in einem Programmiergerät programmierbar, z.B. über Durchtrennen einer Verbindung (Fuse bzw. Antifuse-Techniken)
EPROM	Erasable Programmable ROM Mehrfach in einem Programmiergerät programmierbares ROM. Löschen erfolgt mit Hilfe von UV-Licht im ausgebauten Zustand (veraltet).
EEPROM	Electrically Erasable Programmable ROM Mehrfach elektrisch lösch- und programmierbares ROM. Speicherworte werden einzeln gelöscht und programmiert, Schreibdauer im Bereich 1…10ms je Wort.
Flash-ROM	auch Flash-Memory Mehrfach programmierbares ROM, Löschen erfolgt elektrisch in großen Blöcken von z.B. 8 KB. Löschen und Programmieren ist im eingebauten Zustand möglich, aber langsam 100ms … 1s je Block, Lesen in < 100ns je Wort.

Wichtige Ausführungsformen: RAM (Schreib-Lese-Speicher)

Meist flüchtig , d.h. Inhalt geht beim Abschalten der Versorgungsspannung verloren

SRAM	Statisches RAM Jedes Speicherbit ist als Flipflop aufgebaut, d.h. 4…6 Transistoren je Bit. Gespeicherte Information bleibt (solange Spannung anliegt) ohne zusätzliche Maßnahmen erhalten. Anwendung: Schnelle Speicher (Caches), kleine Speicher in Embedded Systems
DRAM	Dynamisches RAM Kondensator (Transistor-Kapazität) bildet den Speicher, nur 1 Transistor je Bit. Information geht ohne zusätzliche Maßnahmen über der Zeit verloren, muss daher periodisch nachgeladen werden (Refresh). Deutlich langsamer als SRAM, problematischer, aber platzsparend und kostengünstiger als SRAM Anwendung: Große Speicherbänke
FRAM MRAM	Ferromagnetisches RAM, Magnetoresistives RAM Speicherung durch Polarisierung eines Magnetfelds (z.B. „Kondensator“ mit ferromagnetischem Dielektrikum), Speicherung nicht-flüchtig (wie ROM), aber zerstörendes Auslesen mit anschließendem „Nachladen“ (wie DRAM), ähnlich schnell wie Standard-SRAM. Neue Technologie, derzeit nur wenige Produkte.

Speichern von Befehlen oder Daten von Programmen in der Regel gesteuert durch einen Mikroprozessor (CPU)



Speicherzugriff durch die CPU über

- **m bit Adressbus**
- **n bit Datenbus (Bidirektional oder DI/DO)**
- **Steuersignale für Lesen und Schreiben**
- **Signal für Auswahl eines Bausteins (Chip Enable), erzeugt durch Adressdekoder**

Aufbau eines Speichers:

- **Ein Speicherwort ist die kleinste adressierbare Einheit.**
- **Speicherwort enthält „Daten“ (Programmbefehle oder Programmdaten)**
- **Speicherwort hat Umfang von 8, 16, 32 oder 64 bit**
- **Adressierung eines Speicherwortes durch eine laufende Nummer (= Adresse)**
- **m bit breite Adresse ermöglicht Auswahl von 2^m Speicherworten**
- **Speicheradresse entspricht Pointer in C/C++-Programmen**
- **Speichergröße ist 2^m x n bit**

Speicherzugriff: Schreib- oder Lesevorgang unter Kontrolle der CPU

Schreibvorgang:	Lesevorgang:
CPU legt Adresse einer Speicherzelle auf Adressbus	
Adressdekoder erkennt Adressbereich des Speicherbausteins und aktiviert CE	
CPU aktiviert Schreibsignal WR	CPU aktiviert Lesesignal RD
CPU legt Daten auf den Datenbus	Speicher legt Daten auf Datenbus
Speicher übernimmt die Daten	CPU übernimmt die Daten
CPU deaktiviert Schreibsignal WR	CPU deaktiviert Lesesignal RD
CPU schaltet Datenausgänge hochohmig	Speicher schaltet Datenausgänge hochohmig

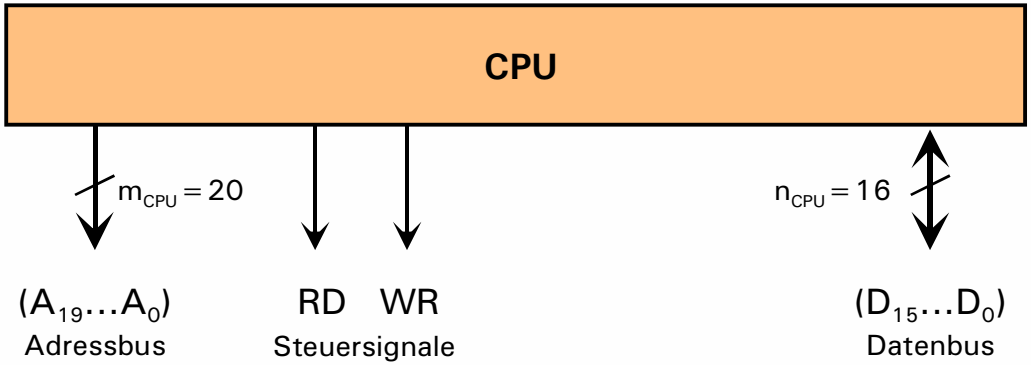
---

CPU legt neue Adresse eines anderen (oder desselben) Bausteins auf Adressbus

Adressdekoder erkennt Adressbereich aktiviert anderes CE (oder lässt dasselbe CE aktiv)

Zyklus beginnt von vorne

➔ Beispielhafter modularer Aufbau eines Speichers (A) Speicherinterface des Mikroprozessors



- **Adressbusbreite der CPU:**
  m_{CPU} = 20 bit
- **Datenbusbreite der CPU:**
  n_{CPU} = 16 bit
- **Steuersignale der CPU:**
  WR Write
  RD Read
- **Kleinste adressierbare Einheit:**
  n_{ADR} = 16 bit
  (hier: 1 Datenwort der CPU, häufig ist die kleinste adressierbare Einheit 8 bit, auch wenn die Datenworte 16 bit oder 32 bit sind.)
- **Adressraum der CPU**
  (maximal möglicher Speicher, „Speichervolumen“)

  $M_{CPU} = 2^{m_{CPU}} \times n_{ADR}$

  $= 2^{20} \times 16$ bit = 1M x 16 bit

  (= 2 MB)

Wichtige Einheiten und Werte:

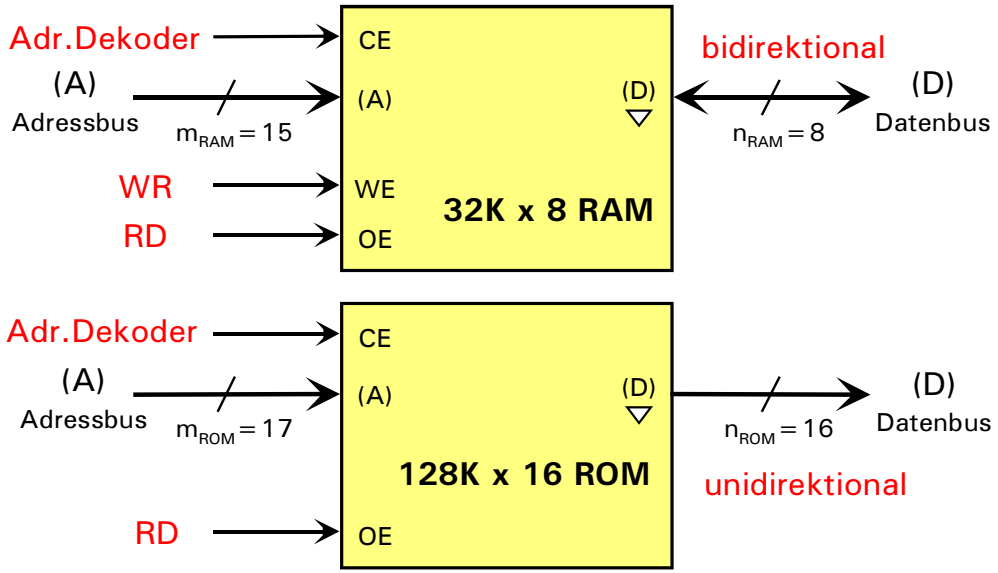
Einheit	Kürzel	Beschreibung
BYTE	B	8 bit-Datenwort
WORD	-	16 bit-Datenwort
DWORD	-	32 bit-Datenwort
QWORD	-	64 bit-Datenwort
Kilo	K	$2^{10} = 1024$
Mega	M	$2^{20} = 2^{10} \cdot 2^{10} = 1.048.576$
Giga	G	$2^{30} \approx 1$ Milliarde
Tera	T	$2^{40} \approx 1$ Billion

➔ Beispielhafter modularer Aufbau eines Speichers (B) Speichersystem, aufgebaut aus mehreren Speicherbausteinen

Gewünschte Daten des Speichersystems:

- 64 KB RAM am Anfang des Adressraumes der CPU (ab Adresse 0 der CPU) **32K x 16**
- 512 KB ROM am Ende des Adressraumes der CPU (bis zur höchsten Adresse) **256K x 16**
- der verbleibende Adressraum soll für Erweiterungen freigehalten werden

Verfügbare Speicherbausteine:



Beim ROM: Kein Schreibsignal WE

RAM 32K x 8
- **Datenwortbreite: n_{RAM} = 8 bit**
- **Adresswortbreite: m_{RAM} = 15 bit**

ROM 128K x 16
- **Datenwortbreite: n_{ROM} = 16 bit**
- **Adresswortbreite: m_{ROM} = 17 bit**

CE .. Chip Enable
WE .. Write Enable OE .. Output Enable

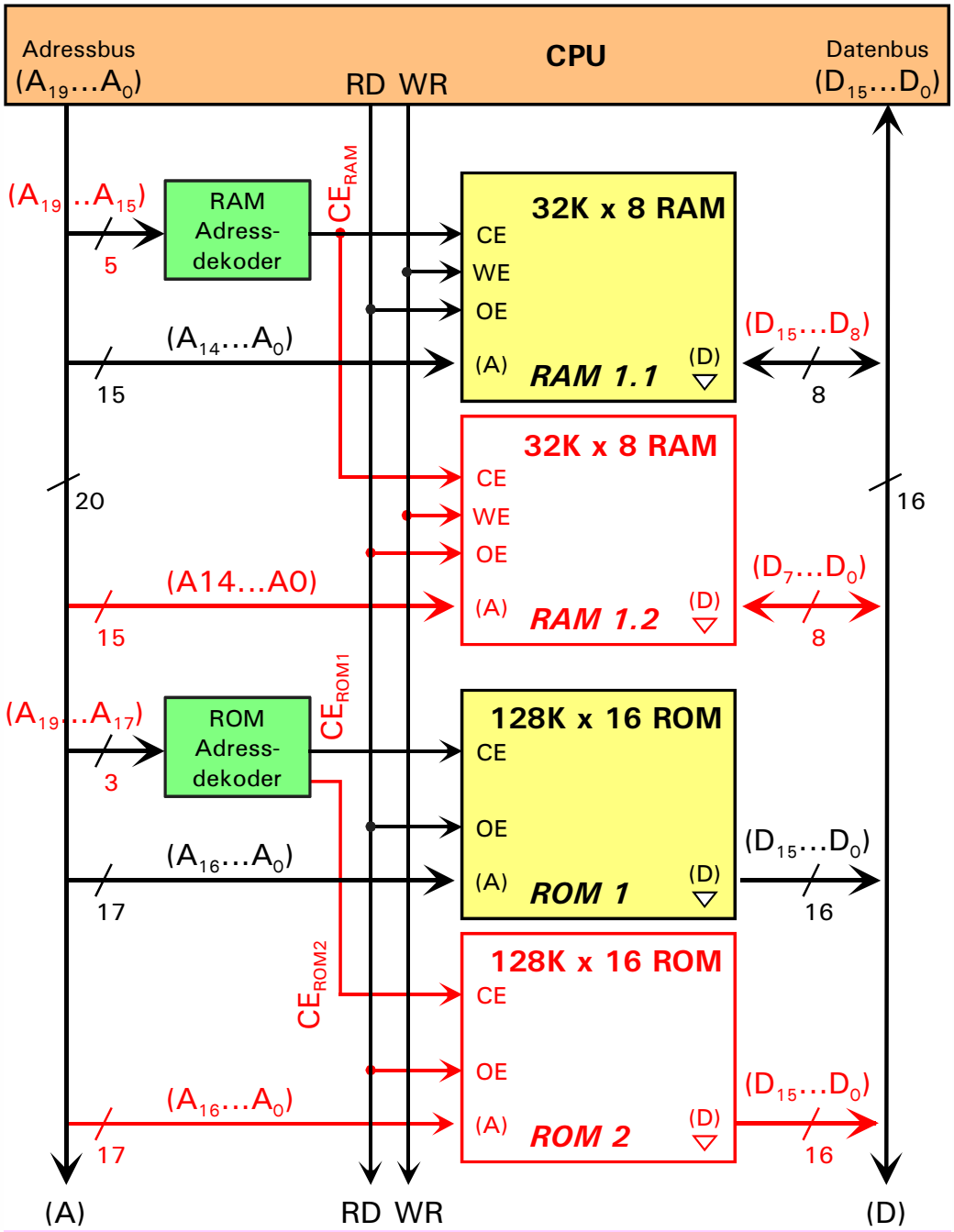
➔ Funktion der Steuersignale CE, OE und WE

CE	OE	WE	Funktion
0	X	X	Speichern – Chip nicht aktiviert, Datenausgänge hochohmig (Z)
1	0	0	Speichern – Chip aktiviert, Datenausgänge hochohmig (Z)
1	0	1	Schreiben – Daten (D) werden unter Adresse (A) gespeichert
1	1	0	Lesen – Unter Adr. (A) gespeichertes Datenwort wird an (D) ausgegeben
1	1	1	Nicht zulässig

Lösungsschritt 1: Festlegen der Adressen des Speichersystems in einer Memory Map (Speicher-Adress-Tabelle) der CPU

Anfangsadresse	Endadresse	Speichertyp
0.0000_H RAM1.1 || RAM1.2	$0.7FFF_H$	RAM 64 KB = 32K x 16 bit 2 Bausteine parallel 1: D15…D8; 2: D7…D0
0.8000_H	$B.FFFF_H$	Frei, reserviert für Erweiterungen
ROM1: $C.0000_H$ ROM2: $E.0000_H$	$D.FFFF_H$ $F.FFFF_H$	ROM 512 KB =256K x 16 bit 2 Bausteine seriell, jeweils D15…D0, aber getrennte Adressbereiche!

3.2 Modularer Aufbau von Speichersystemen



Schritt 2: Blockschaltbild

➔ Aufbau des RAM-Speicherbereichs

Parallelschaltung von 2 Bausteinen, da Datenbusbreite des Bausteins zu klein

1.1: Datenbussignale D_{15} ... D_8

1.2: Datenbussignale D_7 ... D_0

Beide Bausteine müssen gleichzeitig aktiviert werden:

Parallelschaltung von CE, WE und OE

➔ Aufbau des ROM-Speicherbereichs

Serienschaltung von 2 ROM-Bausteinen ergibt die benötigte Speichergröße

1 Baustein = 128Kx16 bit = 256 KB

benötigt: 512 KB

Unterschiedliche CE-Signale

Schritt 3: Adressdekoder - Schaltnetz zur Generierung der CE-Signale gemäß Festlegung in der Memory Map der CPU

(A) RAM: 15 Adressleitungen A14…A0 werden direkt auf den Baustein geführt
restliche Adressleitungen A19…A15 werden im Adressdekoder genutzt

RAM1.1 und 1.2		A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	...	A_0
Anfang	0.0000_H	0	0	0	0	0	0	...	0
Ende	$0.7FFF_H$	0	0	0	0	0	1	...	1

$$CE_{RAM} = \overline{A_{19}} \cdot \overline{A_{18}} \cdot \overline{A_{17}} \cdot \overline{A_{16}} \cdot \overline{A_{15}}$$

(B) ROM: 17 Adressleitungen A16…A0 werden direkt auf die Bausteine geführt
restliche Adressleitungen A19…A17 werden im Adressdekoder genutzt

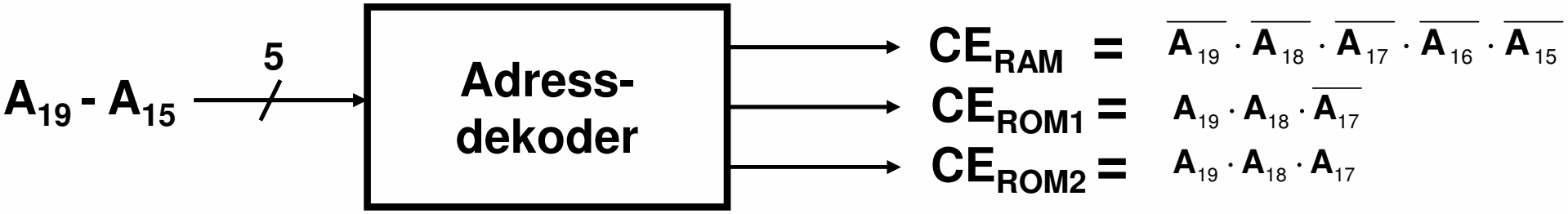
ROM1		A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	...	A_0
Anfang	$C.0000_H$	1	1	0	0	0	0		0
Ende	$D.FFFF_H$	1	1	0	1	1	1		1

$$CE_{ROM1} = A_{19} \cdot A_{18} \cdot \overline{A_{17}}$$

ROM2		A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	...	A_0
Anfang	$E.0000_H$	1	1	1	0	0	0	...	0
Ende	$F.FFFF_H$	1	1	1	1	1	1	...	1

$$CE_{ROM2} = A_{19} \cdot A_{18} \cdot A_{17}$$

➔ Adressdekoder: Schaltnetz zur Generierung der CE-Signale gemäß Festlegung in der Memory Map der CPU



Anmerkungen:

(1) Modularer Aufbau

Häufig werden aus mehreren Speicherbausteinen Speichermodule als Steckkarten aufgebaut. Im Beispiel könnten die Bausteine ROM1 und ROM2 zu einem ROM-Modul zusammengefasst werden, das dann aufgesteckt werden kann. Die Adressdekodierung erfolgt dann in 2 Stufen:

- **Signal für das Modul:** $CE_{ROM-Modul} = A_{19} \cdot A_{18}$

- **Signale für die Bausteine auf dem Modul**

$$CE_{ROM1} = CE_{ROM-Modul} \cdot \overline{A_{17}} \qquad CE_{ROM2} = CE_{ROM-Modul} \cdot A_{17}$$

(2) Vereinfachung durch Don‘t Care:

Wenn der restliche Adressbereich nicht freigehalten werden muss, kann der Adressdekoder oft stark vereinfacht werden

Zusammenfassung: Entwurf Speichersystem

Gegeben: **CPU mit m_{CPU} bit Adressbus und n_{CPU} bit Datenbus,**
Kleinste adressierbare Einheit n_{ADR} bit

→ **Gesamte Anzahl der CPU-Adressen** $N_{CPU} = 2^{m_{CPU}}$

Adressraum der CPU (max. Speichergröße) $N_{CPU} \times n_{ADR}$

Gegeben: **Speicher-IC mit Organisation** $2^{m_{IC}} \times n_{IC}$

m_{IC} … Anzahl der Adresseingänge, n_{IC} … Anzahl der Datenein/ausgänge

Gesucht: **Aufbau eines Speichermoduls der Größe** $N_{Modul} \times n_{ADR} = N^*_{Modul} \times n_{CPU}$

Umrechnung auf die Datenbusbreite der CPU $N^*_{Modul} = N_{Modul}\, n_{ADR} / n_{CPU}$

Anzahl parallel zu schaltender Speicher-ICs $p = n_{CPU} / n_{IC}$

Parallelgruppe. Alle ICs in einer Parallelgruppe werden an dasselbe CE-Signal und an dieselben Adressleitungen, aber an unterschiedliche Datenleitungen angeschlossen.
Eine Parallelgruppe erscheint in der Memory Map als eine einzige Zeile.

Anzahl in Serie zu schaltender Parallelgruppen aus je p Speicher-ICs

$$s = (N_{Modul} \times n_{ADR}) / (2^{m_{IC}} \times n_{IC} \cdot p) = (N^*_{Modul} \times n_{CPU}) / (2^{m_{IC}} \times n_{IC} \cdot p)$$

Alle Parallelgruppen einer Serienschaltung werden an unterschiedliche CE-Signale, aber an dieselben Adress- und Datenleitungen angeschlossen.
Jede Parallelgruppe einer Serienschaltung erscheint in der Memory Map als eigene Zeile.

Zusammenfassung: Entwurf Speichersystem - Fortsetzung

Fall 1: $n_{ADR} = n_{CPU}$

Die Adresseingänge eines Speicherchips werden an die untersten m_{IC} Adressleitungen des Adressbusses angeschlossen.
Die obersten $m_{CPU} – m_{IC}$ Adressleitungen gehen zum Adressdekoder.

Fall 2: $n_{ADR} < n_{CPU}$

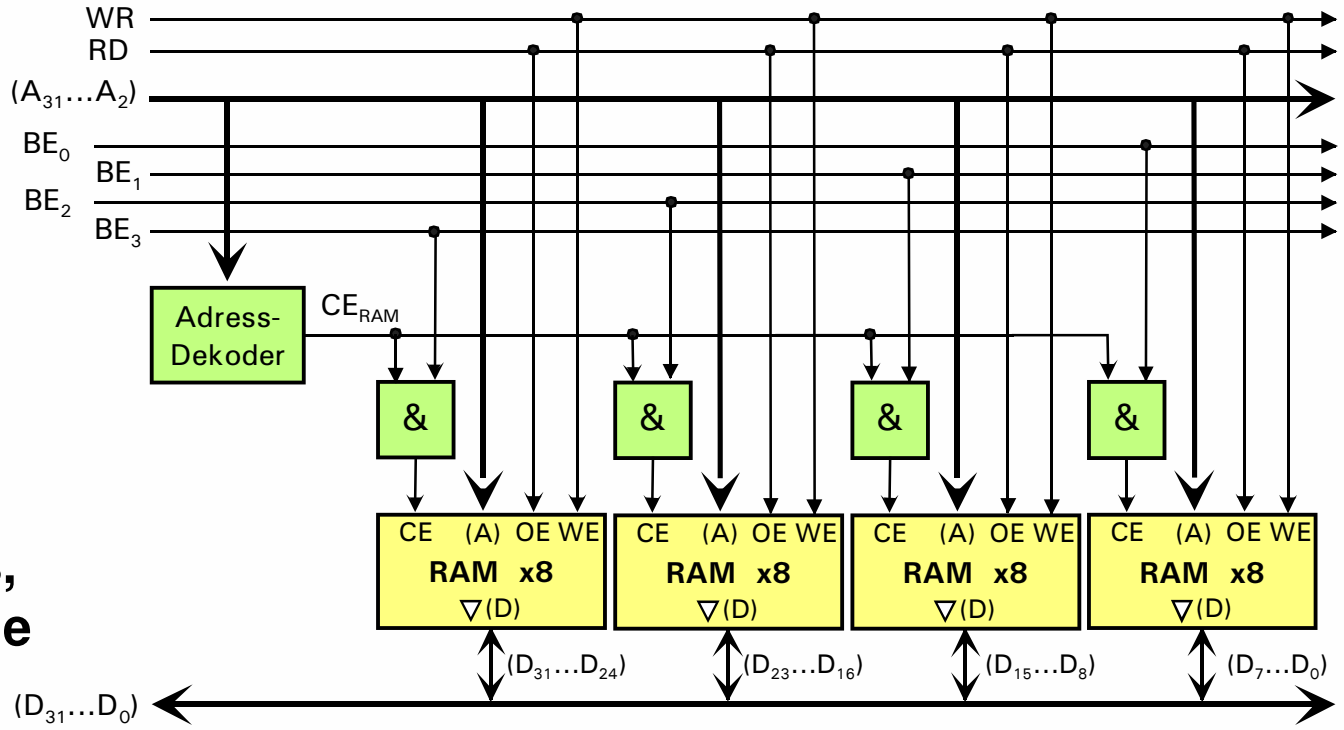
Beispiel

$n_{ADR} = n_{IC} = 8$ bit, $n_{CPU} = 32$ bit

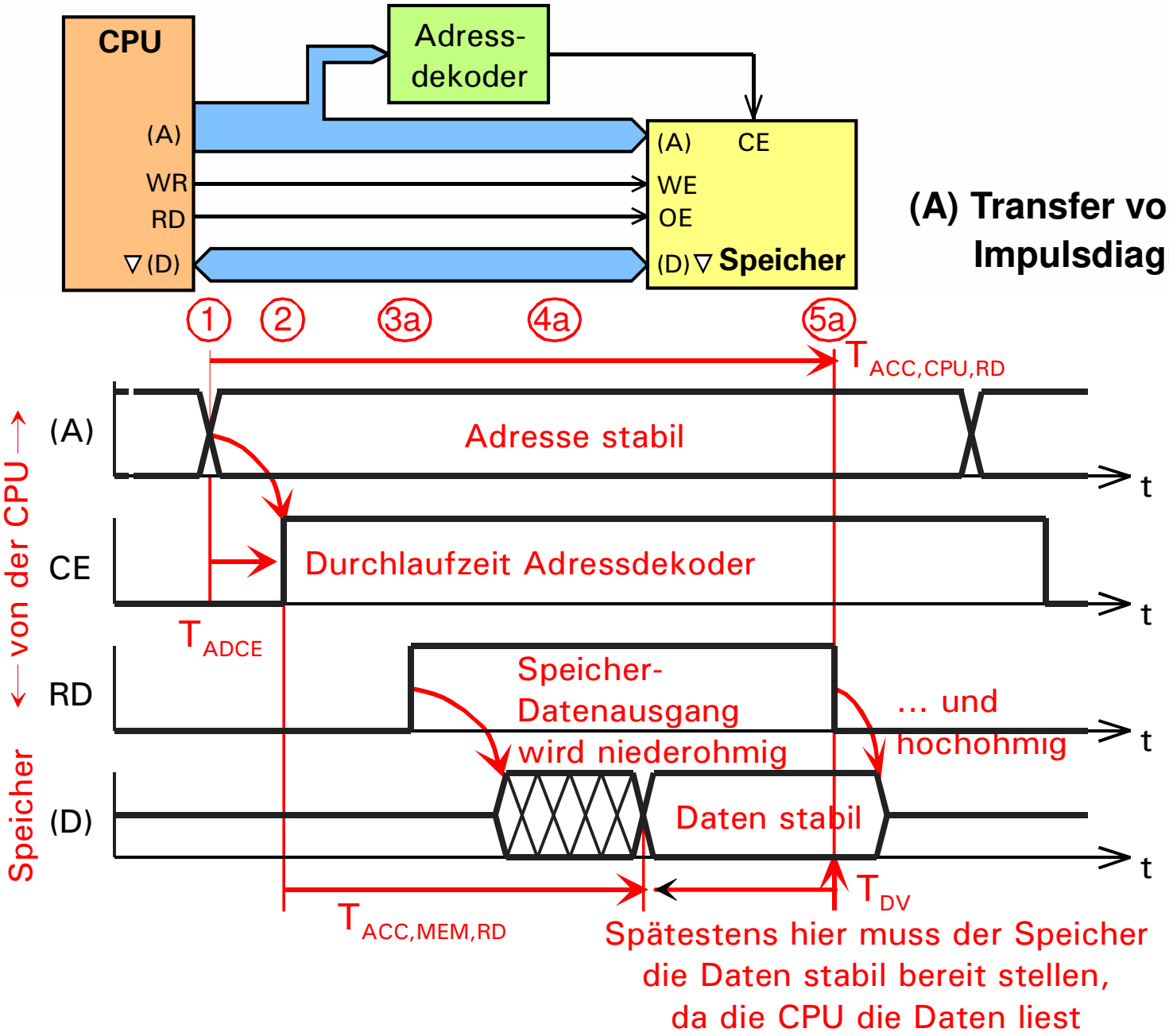
Adressleitungen A_1, A_0 werden nicht verwendet

(A_{31},.., A_2) adressieren Blöcke von 4 Byte

Zusätzlich gibt die CPU 4 Bus Enable Signale (BE_3, .., BE_0) aus, mit denen sie anzeigt, auf welche 1 - 4 Bytes zugegriffen wird.



3.3 Zeitlicher Ablauf von Speicherzugriffen

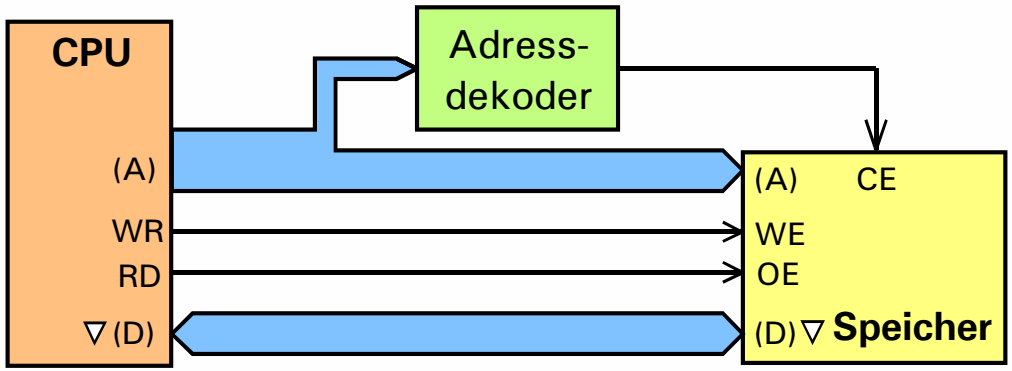


(A) Transfer vom Speicher zur CPU: Impulsdiagramm Lesezugriff

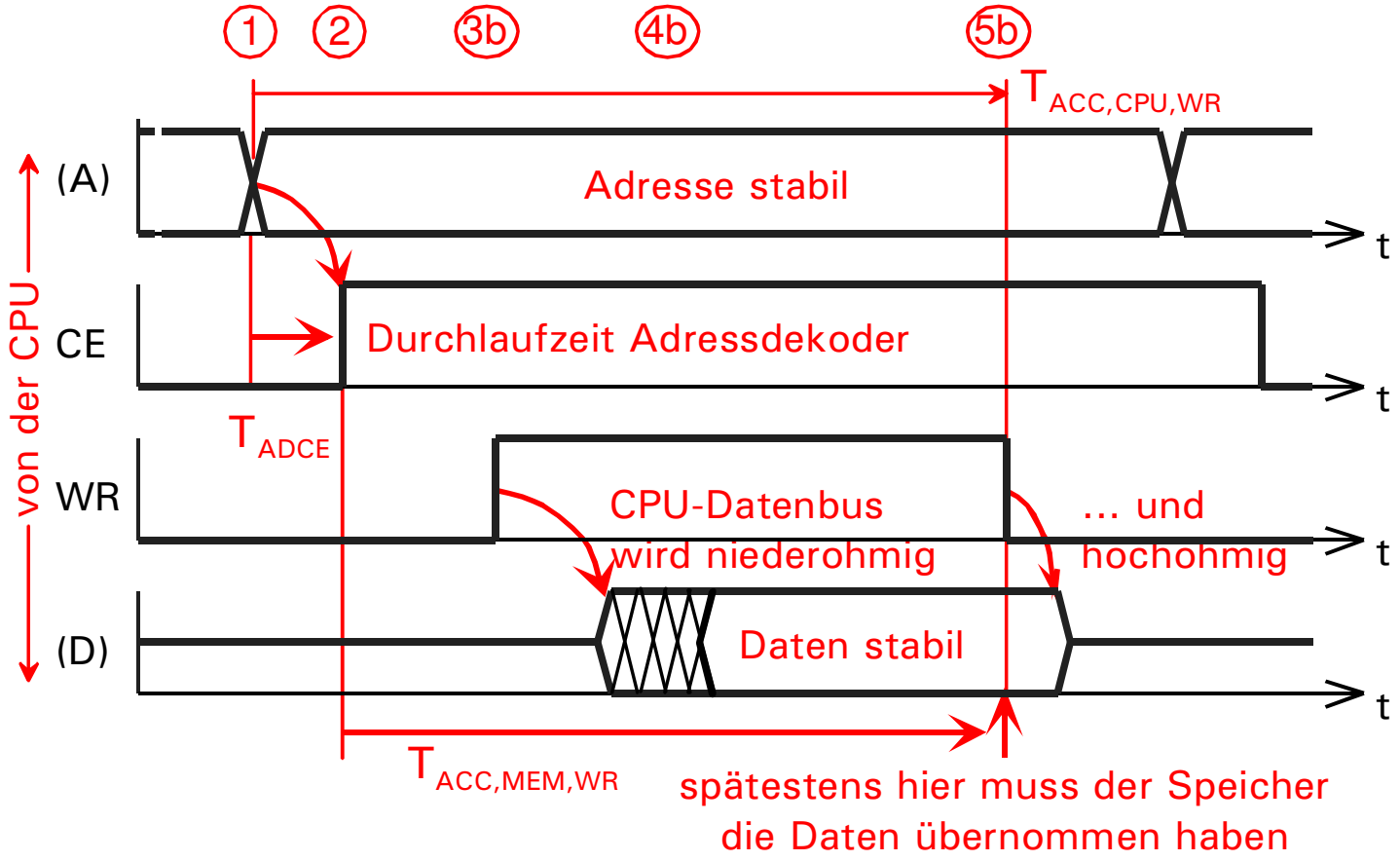
Wichtigste Bedingung für Funktion:

$T_{ADCE} + T_{ACC,MEM,RD} < T_{ACC,CPU,RD} - T_{DV,CPU}$

3.3 Zeitlicher Ablauf von Speicherzugriffen



(B) Transfer von der CPU zum Speicher: Impulsdiagramm Schreibzugriff



Wichtigste Bedingung für Funktion:

$T_{ADCE} + T_{ACC,MEM,WR} < T_{ACC,CPU,WR}$

3.3 Zeitlicher Ablauf von Speicherzugriffen

Der Zugriff erfolgt unter Kontrolle der CPU in folgenden Schritten:

(1) CPU legt neue Adresse an den Adressbus

(2) Angesprochener Baustein erhält CE-Signal über Adressdekoder

(A) für Lesezugriff:

(3A) CPU aktiviert Lesesignal RD

(4A) Speicherbaustein treibt Datenbus und legt die Daten des Speicherworts an

(5A) CPU übernimmt die Daten mit der Deaktivierung des Lesesignals

(B) für Schreibzugriff:

(3B) CPU aktiviert Schreibsignal WR

(4B) CPU treibt Datenbus und legt die Daten für das adressierte Speicherwort an.

(5B) Speicher übernimmt die Daten spätestens mit der Deaktivierung des Schreibsignals

- Der zeitliche Verlauf der Signale RD und WR wird von der CPU bestimmt.

- Weitere Zeitbedingungen wie Setup- und Hold-Time der Speicher müssen eingehalten werden.

3.3 Zeitlicher Ablauf von Speicherzugriffen

Wichtigste Kenngrößen: Lese- und Schreibzugriffszeit der CPU und des Speichers

(A) Lesezugriffszeit der CPU $T_{ACC,\ CPU,\ RD}$:

Zeitdauer zwischen dem Anlegen der neuen Adresse und der Übernahme der Daten (Zeitpunkt: Deaktivierung des RD-Signals). Die Daten müssen aber schon in der Zeit $T_{DV,\ CPU}$ (Data Valid bzw. Data Setup Zeit) vorher stabil gewesen sein.

(B) Schreibzugriffszeit der CPU $T_{ACC,\ CPU,\ WR}$:

Zeitdauer zwischen dem Anlegen der neuen Adresse und der Deaktivierung des WR-Signals (Daten vom Speicher übernommen).

(C) Lesezugriffszeit des Speichers $T_{ACC,\ MEM,\ RD}$:

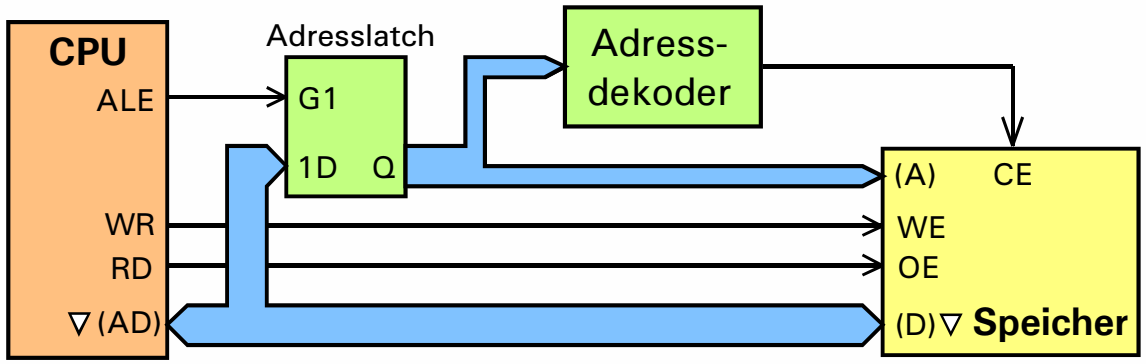
Zeitdauer zwischen der Aktivierung des CE-Signals und der Bereitstellung stabiler Daten auf dem Datenbus durch den Speicherbaustein.

(D) Schreibzugriffszeit des Speichers $T_{ACC,\ MEM,\ WR}$:

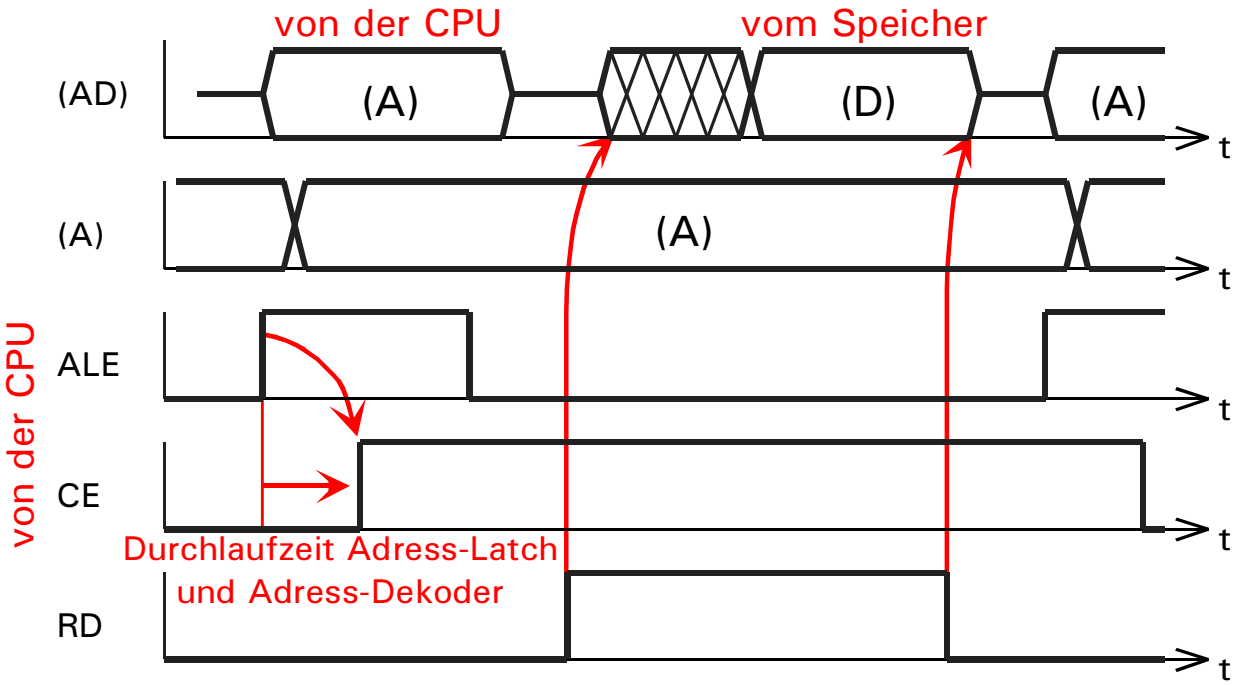
Zeitdauer zwischen der Aktivierung des CE-Signals und der Übernahme der Daten vom Datenbus durch den Speicherbaustein.

Die minimal erforderlichen bzw. maximal zulässigen Zeiten können den Datenblättern der Komponenten entnommen werden. Kann die Ungleichung nicht erfüllt werden, so müssen schnellere Speicher eingesetzt werden. Alternativ bieten CPUs auch eine Verlangsamung der Zugriffe durch Einfügen von Wait-States oder einen zusätzlichen WAIT/READY-Eingang an.

Multiplex-Adress-Daten-Bus



Zeitlicher Ablauf eines Lesezugriffs:



➔ **Aus Kostengründen: Reduzierung der Anzahl von Adress- und Datenleitungen durch Zeitmultiplex**

- **Übertragen von Adressen und Daten zeitlich versetzt zur Einsparung von kostspieligen Leitungen und Pins auf Bausteinen (höherer Zeitbedarf je Transfer)**

➔ **Weitergehende Alternative: Serieller Zugriff auf Speicher**

- **Kostengünstiger, aber wesentlich langsamerer Zugriff auf Speicher durch serielle Adress- und Datenbusse wie z.B. I^2C oder SPI (3 Leitungen, meist synchron-serieller Zugriff)**

Lesezugriff Multiplex-Adress-Daten-Bus:

- Adressen und Daten werden durch CPU im Zeitmultiplex (also nacheinander) auf einem kombinierten Adress- und Datenbus (AD) übertragen

  Nachteil: Einzelner Zugriff wird langsamer (geringere Transferrate)

  Vorteil: Einsparen von Signalleitungen und Anschlüssen (Pins)

- Der Speicher benötigt alle Signale während des gesamten Zugriffs:

  Zwischenspeicher für Adressleitungen (Address Latches) erforderlich

- Es wird ein zusätzliches Signal benötigt, mit dem die Eindeutigkeit bzgl. der anliegenden Daten sichergestellt wird (ALE .. Address Latch Enable).

  (1) CPU aktiviert das Signal ALE und legt Adresse (A) an den Bus (AD).

  (2) Die Adresse wird in das Address-Latch des Speichermoduls übernommen

  (3) Speicher (bei Schreibzugriff: CPU) legt Daten (D) auf den Bus (AD).

  (4) Weiterer Vorgang: siehe Speicherzugriff Seite 3.13 dieses Kapitels.

In der Praxis werden oft Mischformen verwendet, z.B. 16 bit Adress- / 8 bit Datenbus

Höherwertige Adressen (A_{15} - A_8) nicht gemultiplext

Niederwertige Adressen (A_7 - A_0) und Daten (D_7 - D_0) gemultiplext

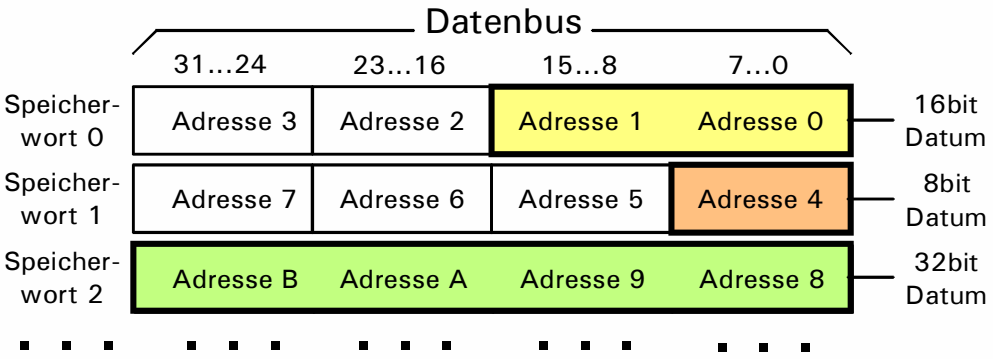
Anhang A: Datenausrichtung

Bei Prozessoren, bei denen die kleinste adressierbare Einheit n_{ADR} kleiner ist als die Datenbusbreite n_{CPU}, beginnen vollständige Speicherworte theoretisch immer bei Adressen, die Vielfache von n_{CPU}/n_{ADR} sind, z.B. bei n_{ADR}=8 bit und n_{CPU}=32 bit bei Adressen, die Vielfache von 4 sind, also 0, 4, 8, 16, …

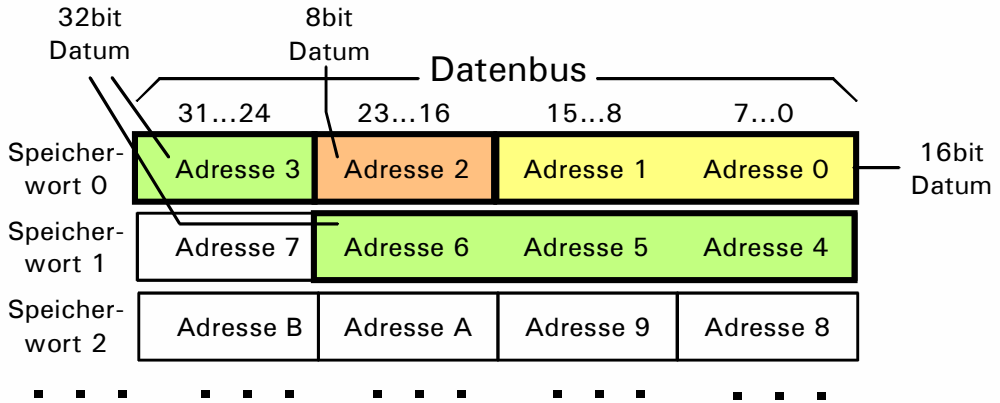
Speichert man in einem solchen Prozessor Daten, die kleiner sind als n_{CPU}, und richtet die gespeicherten Daten streng an den Speicherwortgrenzen aus (**Data Alignment**), so wird Speicherplatz verschwendet.

Günstiger ist es daher, wenn die Daten nicht an Speicherwortgrenzen ausgerichtet, sondern ohne Lücken gespeichert werden.
→ **Vorteil beim Speichern ohne Datenausrichtung: Bessere Speicherausnutzung.**

Beispiel: Speichern eines 16 bit Wertes, eines 8 bit Wertes und eines 32 bit Wertes mit und ohne Datenausrichtung an Speicherwortgrenzen.



Speichern mit Datenausrichtung



Speichern ohne Datenausrichtung

→ **Nachteil beim Speichern ohne Datenausrichtung: Höhere Zugriffsdauer**

Zum Lesen des nicht ausgerichteten 32 bit Wertes sind im Beispiel allerdings zwei Speicherzugriffe notwendig, da die CPU nicht zwei Speicherworte gleichzeitig lesen kann. Die Bussteuereinheiten von CPUs (siehe Kapitel 4), die ohne Datenausrichtung arbeiten, führen solche Zugriffe automatisch aus, ohne dass der Programmierer sich darum kümmern muss.

Der Zugriff auf nicht-ausgerichtete Daten dauert allerdings theoretisch doppelt so lang wie auf ausgerichtete Daten. In der Praxis spielt dies aber eine untergeordnete Rolle, da die meisten CPUs in Verbindung mit Cache-Speichern Daten ohnehin in Blöcken von mehreren Speicherworten lesen.

B.1 ROM-Speicher

- Speicherzelle mit **1 Transistor je Bit**
- Anordnung als (quadratische) **Matrix**
- Aufspaltung der Adresse in **Zeilen- und Spaltenadresse**

Beispiel für einen Zugriff auf Adresse (A)= $FFF001_H$

- Zeilenadresse (A23...A12)=FFF_H → Zeile 4095_D gewählt:

→ X_{4095}=1, alle übrigen X_i=0

→ Zeilentransistoren in Zeile 4095 ein

- Spaltenadresse (A11...A0)=001_H → Spalte 1_D gewählt:

→ Y_1=1, alle übrigen Y_i=0

→ Spaltentransistor T_{y1} ein, übrige Spaltentransistoren aus

➢ Signalpfad von U_0 über $T_{4095,1}$, T_{y1} zum Widerstand R_L

Q=1 → D=1 (wenn CE=1 und OE=1)

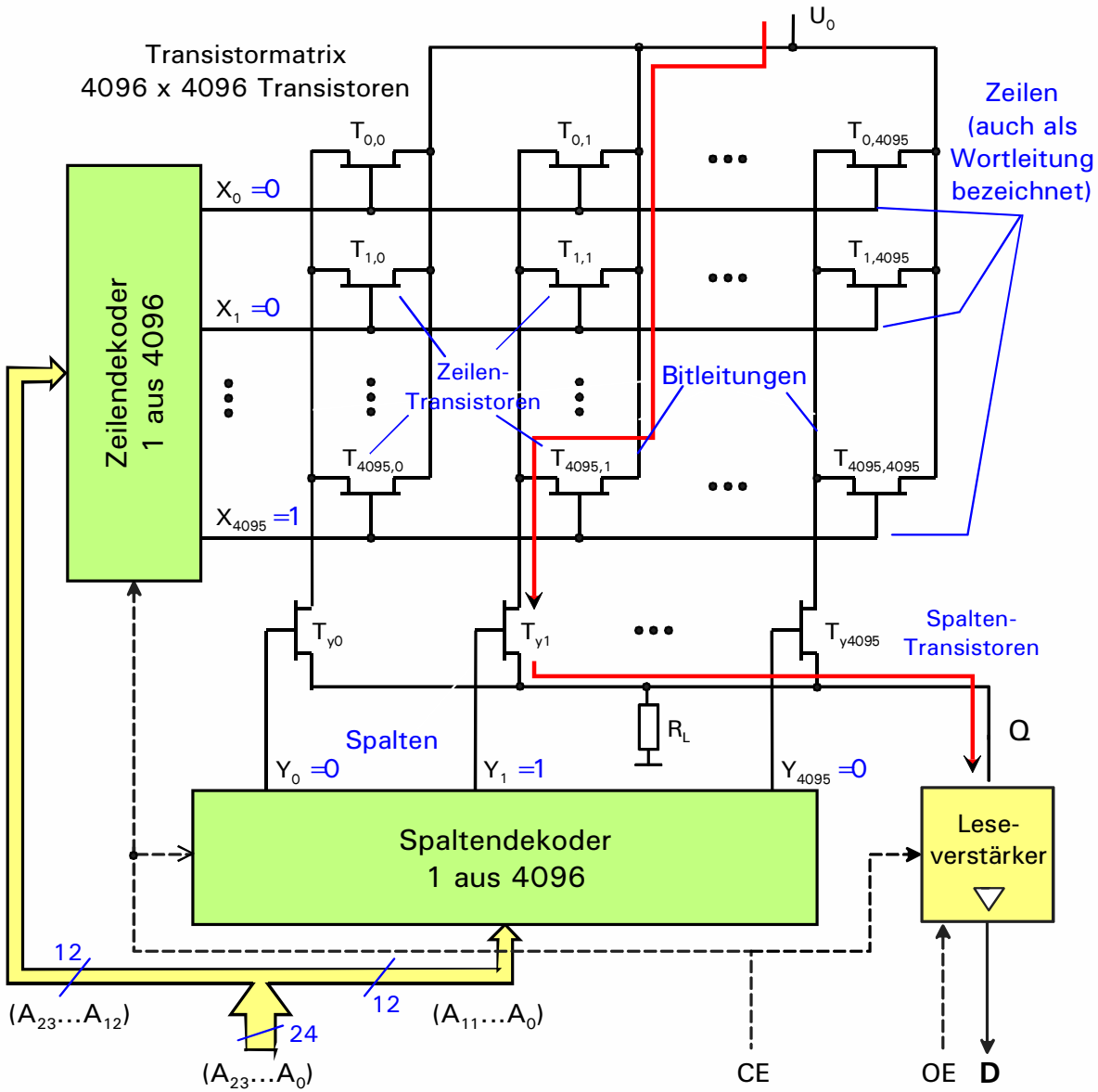
Lesen einer gespeicherten ‚1'

Wenn in dieser Speicherzelle eine ‚0' gespeichert werden soll, wird die Zuleitung zum Transistor T4095,1 nicht angeschlossen bzw. beim Programmiervorgang unterbrochen.

Bei einem … x N Speicher (z.B. 16M x 4) sind die Transistormatrizen und Leseverstärker N mal vorhanden.

Anmerkung: Die Darstellung ist stark vereinfacht, reale Bausteine können komplexer aufgebaut sein.

Bsp.: Ein-Bit-Speicher 16M x 1 bit - MOS-ROM

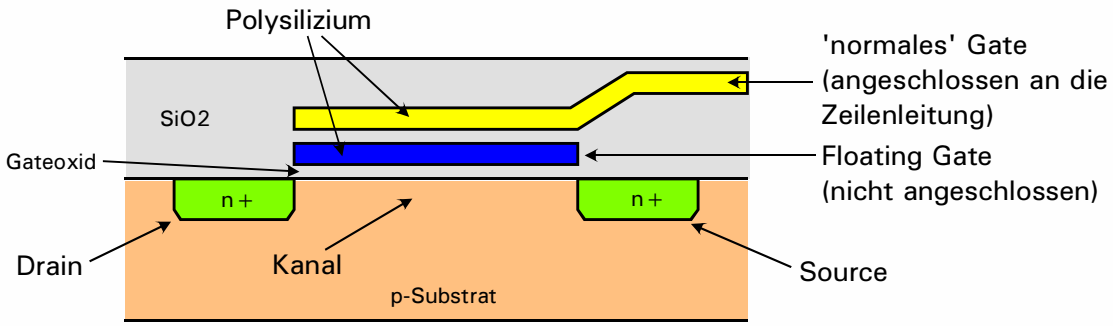


B.2 Lösch- und reprogrammierbare ROMs: EPROM, EEPROM und Flash-ROM

Bei den löschbaren und reprogrammierbaren ROMs werden in der Transistormatrix spezielle Transistoren eingesetzt. Bei diesen Transistoren befindet sich unter dem normalen Gate ein zusätzliches, elektrisch isoliertes Gate, das ***Floating Gate*** :

• **Ungeladenes Floating Gate (Speichern einer '1')**:
Solange sich auf dem Floating Gate keine Ladung befindet, verhält sich der Transistor normal, d.h. sobald die Spannung am Normal-Gate positiv wird leitet der Transistor.

• **Negativ geladenes Floating Gate (Speichern einer '0')**:
Wenn das Floating Gate mit Elektronen stark negativ geladen wird, wird die Schwellspannung des Transistors so groß, dass er auch bei einer positiven Spannung am Normal-Gate sperrt.



• **Laden des Floating-Gate (Programmieren einer '0')**:
Beim Anlegen einer hohen Spannung (Programmierspannung ≈ 12...25V) zwischen G und S bricht die Isolationsschicht durch (Fowler-Nordheim-Tunnel-Effekt oder Hot Electron Injection), es fließt ein Strom über das Floating Gate, der es negativ lädt. Diese Ladung bleibt auch nach Abschalten der Programmierspannung erhalten, da das Gate anschließend wieder völlig isoliert ist. Das Programmieren kann in einem Programmiergerät oder im eingebauten Zustand erfolgen, wenn die Programmierspannung im Chip selbst erzeugt wird, wie bei EEPROMs und Flash-ROMs üblich.

• **Löschen der Ladung auf dem Floating-Gate (Programmieren einer ‚1'):**
Bei den ersten wiederprogrammierbaren Bausteinen, den **EPROMs** (Erasable Programmable ROM, eingeführt ca. 1975) erfolgte das **Löschen, d.h. Entladen des Floating Gates**, durch ultraviolettes Licht in einem Löschgerät im ausgebauten Zustand (Dauer 20..30min). Damit das Licht die Chipoberfläche erreichen kann, müssen diese ICs in ein (sehr teueres) Keramikgehäuse mit Quarzglasfenster verpackt werden.

Bei den um 1985 eingeführten **EEPROMs** (Electrical Erasable Programmable ROM) wird das Floating Gate durch Anlegen eines inversen elektrischen Feldes entladen. Bei EEPROMs kann jedes einzelne Speicherwort einzeln gelöscht und neu programmiert werden. Der Löschvorgang ist relativ langsam und dauert 1…10ms je Speicherwort, während das Lesen in <100ns möglich ist. Da das Erzeugen der Entladespannung und das Anlegen an jede einzelne Speicherzelle sehr aufwendig ist, werden EEPROMs nur mit kleinen Speicherkapazitäten hergestellt, typ. 8…256 KB.

Bei **Flash-ROM**s wird der Aufwand dadurch verringert, dass zwar jedes Speicherwort einzeln programmiert werden kann (Dauer ca. 1…10µs), dass aber nur große Speicherblöcke (typ. 4…64KB) gemeinsam gelöscht werden können (Dauer ca. 1…10s).

Das Laden und Entladen des Floating Gates „stresst" die Halbleiterstruktur stark, daher ist die **Anzahl** der **Lösch- und Programmiervorgänge begrenzt** (typisch 10000 … 1000 000 mal).

B.3 Statische RAM-Speicher (SRAM)

Matrixstruktur wie beim ROM, aber an jedem Kreuzungspunkt ein Flipflop aus 4 Transistoren plus 2 Transistoren als Koppeltransistor (6-Transistor-Speicherzelle) zur Bitleitung

Adressierung:
Über Zeilen- und Spaltenleitung wie beim ROM, neben dem Spaltentransistor sind im adressierten Flipflop auch T_5 und T_6 leitend.

Lesen:
Über T_5 des adressierten Flipflops, die Bitleitung und den leitenden Spaltentransistor $T_{y...}$ liegt **Q am Eingang des Leseverstärkers** und wird von dort auf den Datenausgang D ausgegeben.

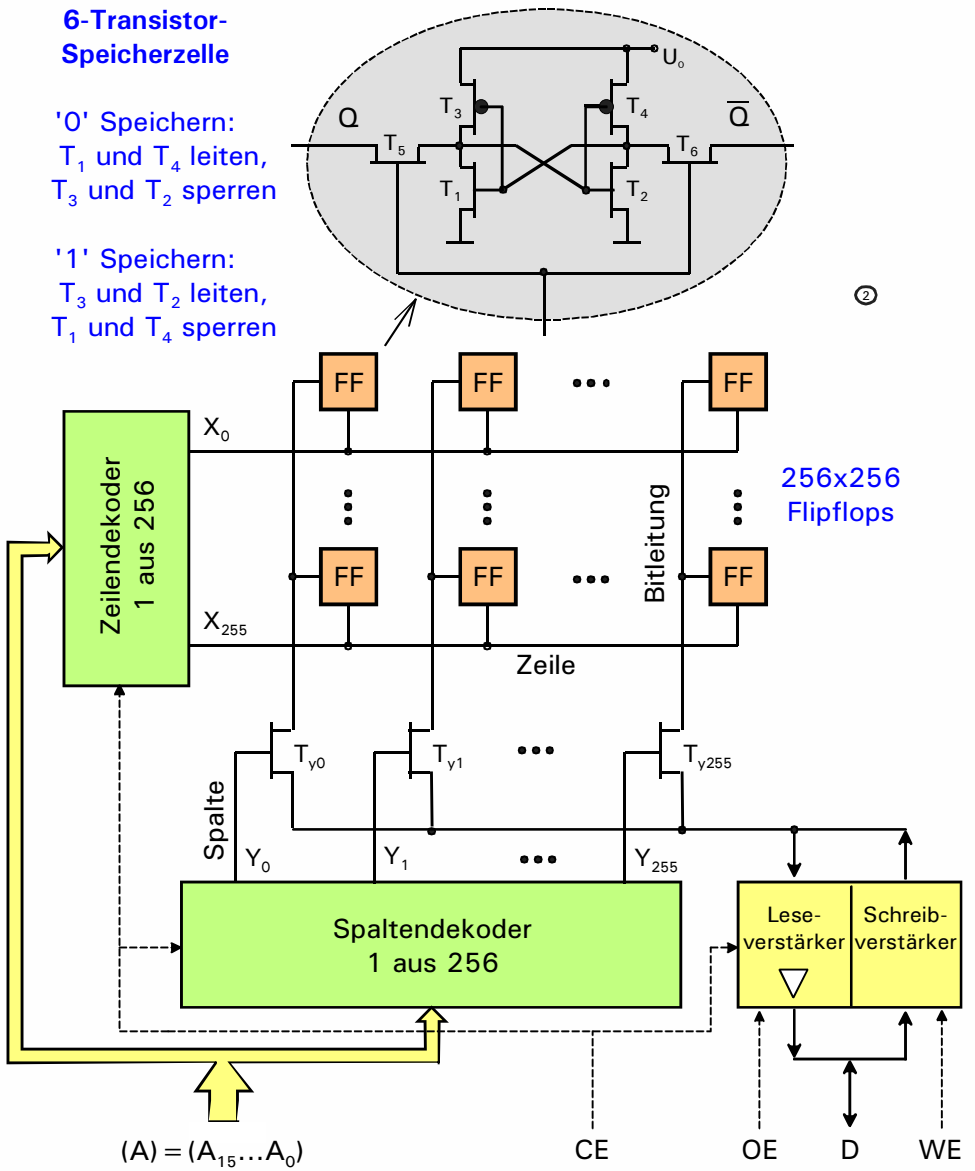
Schreiben:
Der am Dateneingang D liegende Logikwert wird über den Schreibverstärker, die Bitleitung und den leitenden Spaltentransistor $T_{y...}$ sowie über T_5 niederohmig an das Flipflop gelegt. Falls Q=0 ist, wird T_2 gesperrt und T_4 leitend ($\rightarrow$ /Q=1), dadurch wird T_1 leitend, T_3 gesperrt ($\rightarrow$ **Einspeichern von Q=0**). Der Zustand bleibt auch erhalten, wenn T_5 wieder gesperrt wird. Für Q=1 analog.

SRAM-Bausteine sind mit 6 Transistoren je Bit sehr aufwändig, andererseits aber sehr schnell mit Zugriffszeiten im Bereich von einigen Nanosekunden (chip-intern) bis zu einigen zehn Nanosekunden bei separaten ICs.

SRAM werden als On-Chip-Cache-Speicher in großen Mikroprozessoren in Computersystemen (typ. Größe einige KB bis MB) sowie als einziger RAM-Speicher in Mikrocontroller-Systemen (Embedded Systems, typ. Größe 4 … 256 KB) eingesetzt.

Anmerkung: Die Darstellung ist stark vereinfacht, reale Bausteine können komplexer aufgebaut sein.

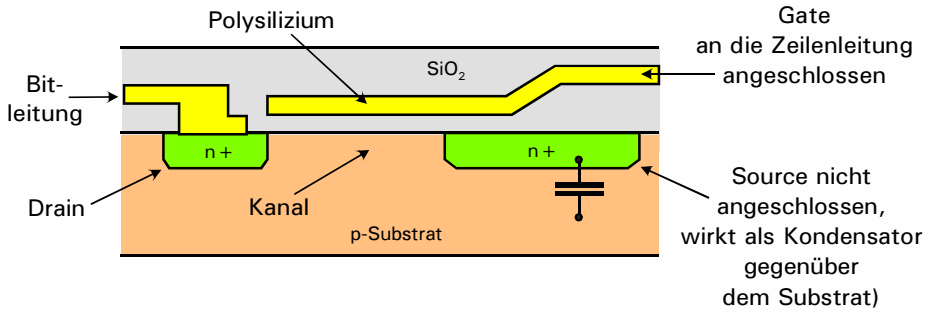
Bsp.: 64Kx1 bit - MOS-Statisches RAM



B.4 Dynamisches RAM (DRAM)

Matrixstruktur wie beim ROM, aber an jedem Kreuzungspunkt ein Transistor mit einer Kapazität als Speicherzelle (**1-Transistor-Zelle**).

Der Source-Anschluss des Transistors wird vergrößert und bildet so einen (kleinen) Kondensator gegenüber dem Substrat, in dem die Information als Ladung gespeichert wird.



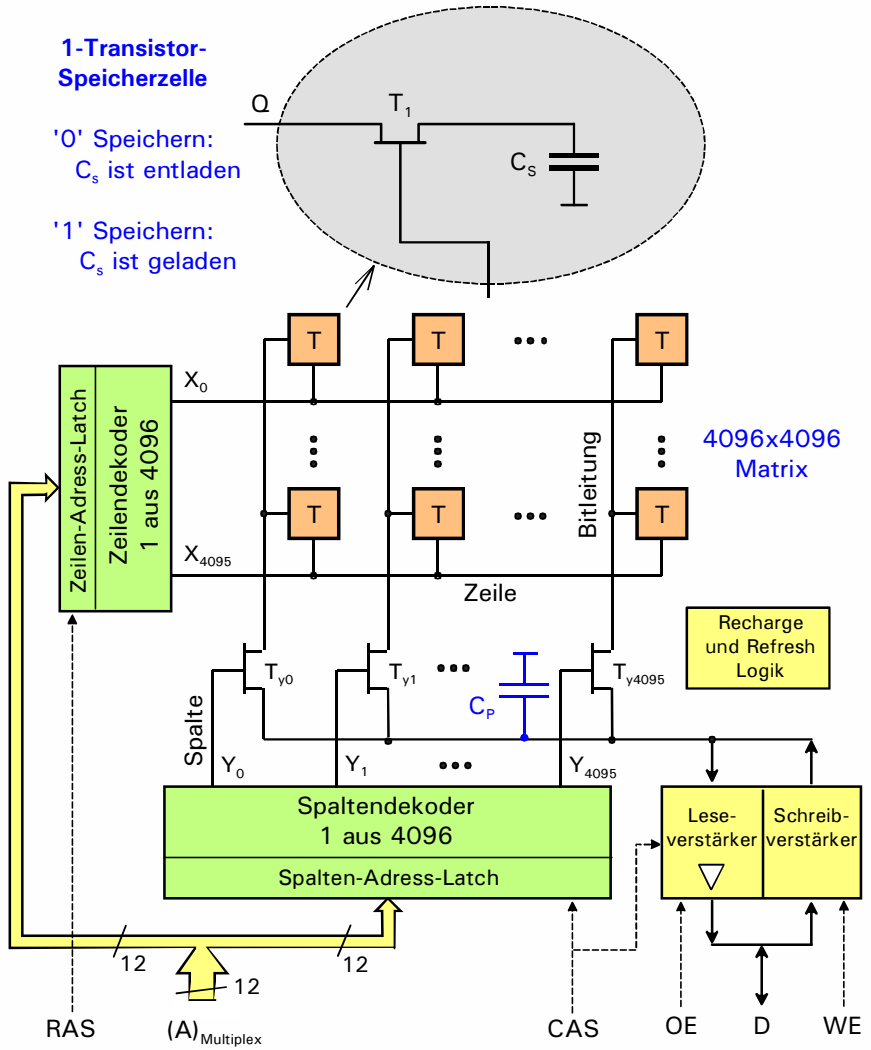
• Beim **Schreiben einer ‚1'** wird die Speicherzelle über die Zeilen- und Spaltenleitungen adressiert und der **Kondensator geladen**, beim **Schreiben einer ‚0'** wird der **Kondensator entladen**.

• **Beim Lesen wird** die Speicherzelle über die Zeilen- und Spaltenleitungen adressiert und die Ladung (Spannung) des Kondensators vom Schreibverstärker ausgewertet.

• Leider ist die Kapazität C_S der Speicherzelle klein im Vergleich zu den unvermeidlichen parasitären Kapazitäten C_P der Bitleitungen und des Schreibverstärkers, so dass **der Speicherkondensator** beim Lesen **entladen** wird (bzw. geladen wird, falls die parasitären Kapazitäten vor dem Lesen geladen waren), d.h. **beim Lesen wird der Inhalt des gelesenen Bits zerstört**. Die Speicherbausteine enthalten **daher** eine **Logik**, **die** den **Wert der gelesenen Speicherzelle nach dem Lesen sofort wieder** in die Zelle **zurückschreibt** und die Kondensatorladung damit erneuert (**Recharge**).

• Die Adressen werden im Multiplex-Verfahren übertragen, um Anschlüsse zu sparen und damit kleinere Gehäuse zu erhalten.

Bsp.: 16Mx1 bit - MOS-Dynamisches RAM



• Selbstverständlich **entladen** sich die Speicherkondensatoren **durch Leckströme** auch dann, wenn nicht auf den Speicher zugegriffen wird. Daher müssen **alle Speicherzellen periodisch nachgeladen** werden (**Refresh**). Bei modernen Bausteinen erfolgt dies durch eine integrierte Logik automatisch.

DRAMs haben sich trotz des Nachteils eines dynamischen Speichers wegen der deutlichen Kostenvorteile als Massenspeicher in größeren Rechnern durchgesetzt:

SRAM: 6 Transistoren je Speicher-Bit - DRAM: 1 Transistor je Speicher-Bit

Der Kostenvorteil verstärkt sich durch ein modifiziertes Speicher-Interface:

Übertragung der Adresse in zwei Hälften im Adress-Multiplex-Verfahren

Vorteile:

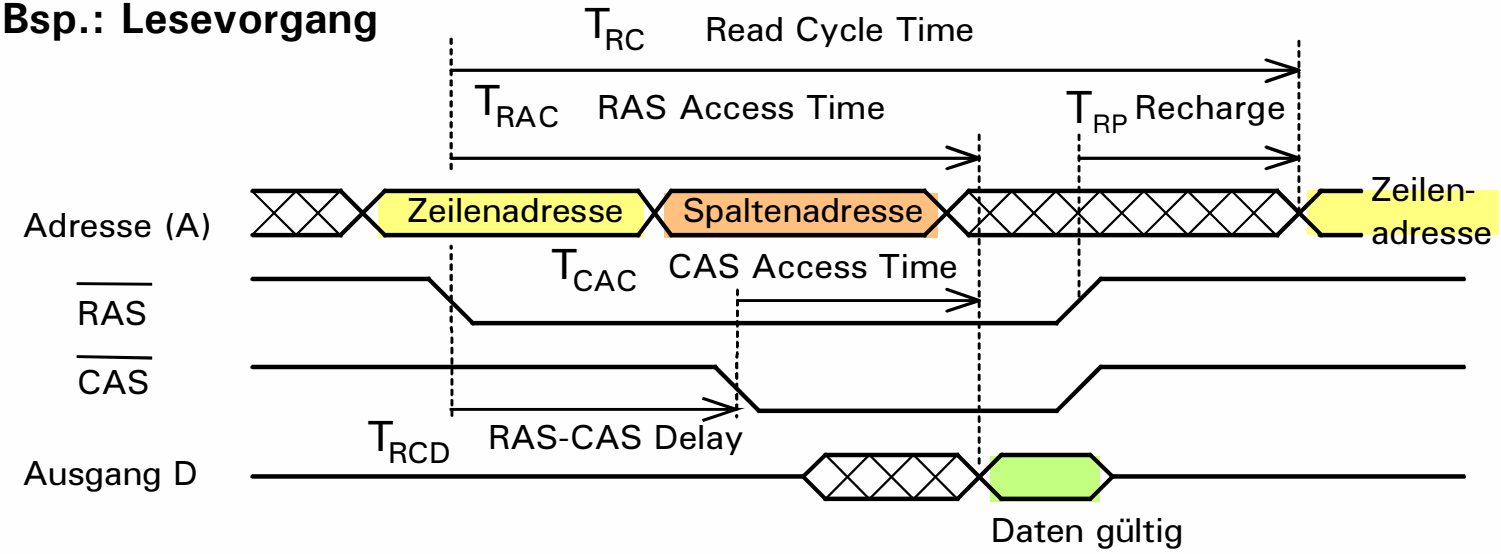
Reduzierung der Anzahl der Adressleitungen (Halbierung) → kleinere Gehäuse

Nachteile:

Ladung der DRAM-Speicherkondensatoren muss zyklisch aufgefrischt werden

→ Zusatzlogik (DRAM-Controller) notwendig, hoher Stromverbrauch.

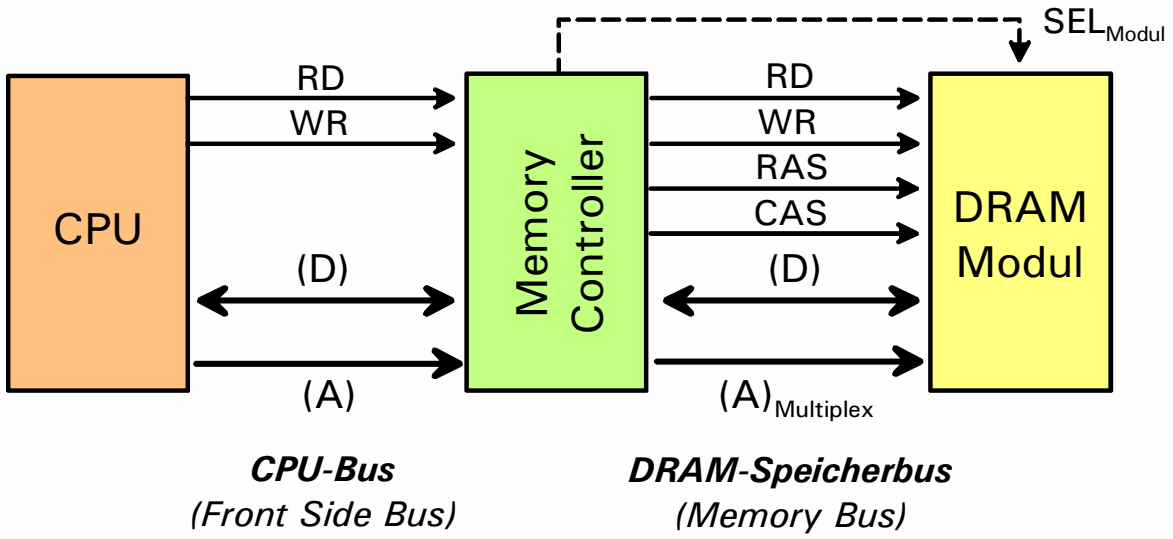
Bsp.: Lesevorgang



Prinzipielle Funktionsweise:

- **Zuerst wird die Zeilenadresse (obere Hälfte der Adresse) angelegt und mit Steuersignal RAS („Row Address Strobe“) angezeigt.**
- **Der Speicher übernimmt die Zeilenadresse in ein internes Address-Latch.**
- **Dann wird die Spaltenadresse (untere Hälfte der Adresse) angelegt und mit Steuersignal CAS („Column Address Strobe“) angezeigt.**
- **Der Speicher übernimmt die Spaltenadresse in ein zweites internes Address-Latch.**
- **Danach werden die Daten wie gehabt transferiert.**

Wegen des abweichenden Speicher-Interfaces wird zwischen CPU und DRAM üblicherweise ein Steuerbaustein (Memory Controller) geschaltet.



Zugriffszeiten:

- Die Verzögerungszeit vom Beginn des Speicherzugriffs (Anlegen Zeilenadresse) bis zum Transfer gültiger Daten liegt im Bereich von T_{RAC} = 60 ns (technologisch bedingt, fortschreitende Miniaturisierung führt zu einer nahezu unveränderten Zeitkonstante $\tau = R \cdot C$, die maßgebend ist für das Auslesen des Speicherkondensators).
- Zudem muss die bei einem Lesezugriff entladene Speicherzelle mit dem Wert des ausgelesenen Bit wieder zurückgeschrieben werden ("Recharge", T_{RP} = 40 ns typisch), da der Speicherkondensator beim Auslesen entladen wird.
- Es ergibt sich somit eine Zykluszeit für den Zugriff $T_{ACC} = T_{RAC} + T_{RP}$ = 100 ns
- Da sich der Speicher-Kondensator mit der Zeit entlädt, muss der gesamte Speicher zusätzlich ständig aufgefrischt werden ("Refresh, typische Zykluszeit 64 ms je Zelle).
- Die in heutigen DRAM-Speichern möglichen Transferraten werden durch Optimierungen erreicht, die die Lokalität von Programmen und Daten ausnützen:

  Statistisch betrachtet werden sehr häufig aufeinanderfolgende Speicherstellen ausgelesen. Das Speicherinterface wird daher für den Transfer ganzer Speicherblöcke ("Burst Modus") optimiert:

  Adresse wird nur beim ersten Zugriff eines Blocks übertragen.

  Beim ersten Zugriff wird der ganze Speicherblock in ein internes SRAM (Cache) übertragen, das bei den folgenden Zugriffen dann sehr schnell ausgelesen wird.

Kapitel 4: CPUs, Mikroprozessoren und Mikrocontroller

4.1 Grundstruktur eines Rechners

Software: Beispiel C++-Programm

```
int j=3;                    ] Daten (global)
void main(void)
{    int i, k;              ] Daten (lokal)
     cin >> i;              ] Befehle  Eingabe
     k = i + j;             ] (Code)   Rechnen
     cout << k;             ]          Ausgabe
}
```

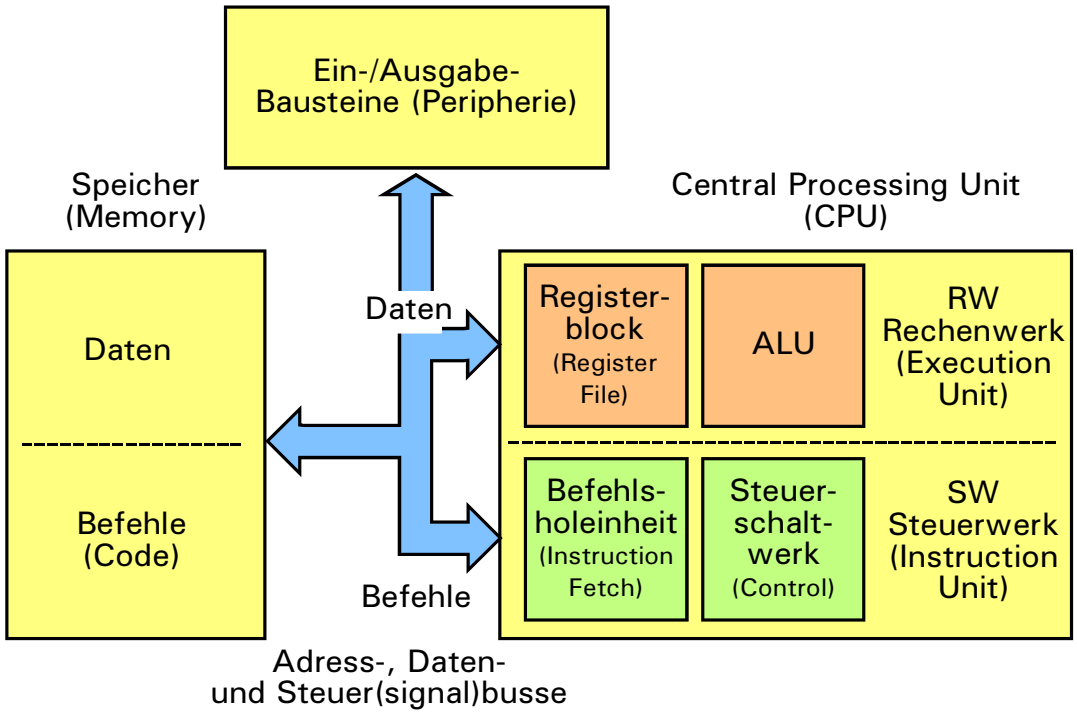
Datenpfad: *Rechenwerk*

Registerblock für Operanden und Ergebnisse (schneller als Speicher) und ALU zur Ausführung arithmetisch-logischer Operationen

Befehlspfad: *Steuerwerk*

Holen und Vorverarbeiten (Dekodieren) von Befehlen (Instruction Fetch Unit) und Schaltwerke(e) zur Erzeugung der Steuersignale für das Rechenwerk

Hardwarestruktur eines Rechners



Mikroprozessor: CPU auf einem Chip, Speicher und Peripherie in separaten Chips allg.Rechner

Mikrocontroller: CPU, Speicher und Ein-Ausgabe-Peripherie auf einem Chip **Embedded System**

Signalprozessor: Rechenwerk optimiert für schnelle Arithmetik **Soundkarte, Modem, Mobiltelefon**

Logische Software/Hardware-Ebenen: Vom Programmcode zur Steuerung der Rechnerhardware

Für Menschen gut verständlich	**Hochsprache**, z.B. C/C++	i = i+1
	⇓	*Übersetzung durch Compiler*
Für Menschen (noch) verständlich Verwendet nur die für den Programmierer sichtbaren CPU-Register	**Machinensprache** (**Assembler**) mit Mnemonics und symbolischen Namen und Adressen	**i → CPU-Register** **CPU-Reg. + 1 → CPU-Reg.** **CPU-Register → i**
	⇓	*Übersetzung durch Assembler*
Für CPU verständlich	**Maschinensprache** binär codiert (Objektcode)	
	⇓ ⇓	*Übersetzung durch Befehlsdekoder im CPU-Steuerwerk*
Verwendet alle CPU-Register	**Mikroprogrammbefehle** (ein oder mehrere Mikrobefehle = Mikroprogramme je Maschinenbefehl)	**Adresse von i → Adressbus** **Datenwort i lesen → CPU-Reg. sp.** **…**
	⇓	*Erzeugung durch Steuerwerk*
Für Gatterlogik verständlich	**Steuersignale für die Hardware** RD, WR, …	**Adressbus-Treiber niederohmig schalten** **Lesesignal aktivieren** **…**

4.2 Programmiermodell: Register, Befehle, Adressierungsarten

Ein C-Compiler/Assembler-Programmierer benötigt folgende Informationen über eine CPU:

- Welche Register gibt es?
- Welche Befehle gibt es?
- Adressierung der Operanden?

Beispiel:

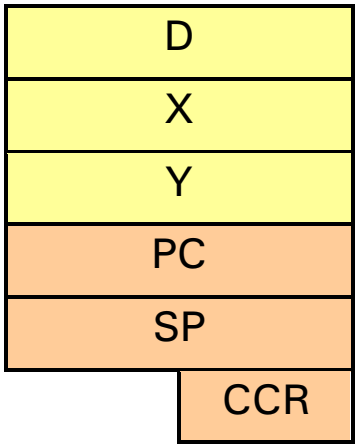
- **Registerblock** CPU-Familie Motorola/Freescale HCS12 (gilt ähnlich auch für andere CPUs)

  Datenwortbreite: 16 bit (***16bit CPU***), 8bit Daten möglich

  Adresswortbreite: 16 bit, Byte-adressierbar, einzelne Modelle mit Adresswortbreite > 16bit

  Bei manchen CPUs sind Daten- und Adressworte unterschiedlich groß. Es vereinfacht aber die Programmierung, z.B. das Rechnen mit Pointern (Adressen), wenn ein Adresswort in ein normales Datenregister passt.

  Die für den Programmierer sichtbaren Register beim HCS12 sind:



16 bit	Akkumulator	Operanden, unterteilt in zwei 8bit Register **A** = $D_{15...8}$, **B** = $D_{7...0}$
16 bit	Indexregister	Operanden und Pointer
16 bit	Indexregister	Operanden und Pointer
16 bit	Programmzähler	Instruction Pointer, Adresse des nächsten Befehls
16 bit	Stackpointer	Adressierung eines speziellen Speicherbereichs (Stack)
8 bit	Condition Code Register	Statusregister, speichert u.a. Carry, Overflow, Zero und Negative Statusbit der ALU

Darüber hinaus besitzt eine CPU sehr viele interne Register, die für den Programmierer nicht direkt sichtbar sind, d.h. die er nicht selbst als Operanden in Befehlen verwenden kann, z.B. das Befehlsregister IR.

Der Speicher ist **Byte-adressierbar**, d.h. jedes Byte im Speicher hat eine eigene Adresse.

Speicherreihenfolge bei Mehrbyte-Daten: **Big Endian** (Most Significant Byte zuerst)

Bei Adressangaben genügt Angabe der Adresse des ersten Bytes (und der Länge)

Bsp.: 16bit Wert $4433 [*1] ab Adresse $0104

Adresse	Inhalt	
$0000	. . .	
. . .	. . .	
$0104	$44	MSByte
$0105	$33	LSByte
. . .	. . .	

Bei anderen Herstellern, z.B. Intel, Infineon, häufig auch **Little Endian**:

Niederwertiges Byte zuerst

[*1] Motorola/Freescale verwendet für hexadezimale Zahlen die Darstellung $. . . statt . . . h, d.h. $4433 = 4433h

- **Maschinenbefehle**

Praktisch alle auf dem Markt erhältlichen CPUs verwenden sehr ähnliche Grundbefehlssätze:

Datentransport	Laden, Speichern und Kopieren zwischen Registern oder Register und Speicher. *Bezeichnungen: Load, Store, Move, Transfer,* ...
Arithmetisch-logische Operationen	Inkrementieren, Dekrementieren, Addieren, Subtrahieren ohne/mit Übertrag, Vorzeichenumkehr von ganzen Zahlen in Betrags- und 2er-Komplementdarstellung (**Integerarithmetik**), Bitweise Invertierung, Und-, Oder-, Exklusiv-Oder-Verknüpfung, Rechts-/Links-Schieben *Bezeichnungen: Add, Subtract, Negate, Not, And, Or, Xor, Shift* Multiplikation und Division von ganzen Zahlen (nur bei großen CPUs mit Hardwaremultiplizierer, bei kleineren CPUs im Mikroprogramm in mehreren Schritten nachgebildet). *Bezeichnungen: Multiply, Divide* Nur bei Hochleistungs-CPUs: Rechnen mit reellen Zahlen (Floating Point, Gleitkomma)
Vergleichs- und Verzweigungsbefehle	Vergleich zweier Werte. *Bezeichnungen: Compare, Test* Bedingte und unbedingte Sprünge. (Bedingte Sprünge werden nur ausgeführt, wenn eine Bedingung erfüllt ist, z.B. das Zero Statusbit gesetzt ist). *Bezeichnungen: Jump, Branch,* ... Unterprogrammaufruf und –rücksprung. *Bezeichnungen: Call, Return*
Steuerbefehle	Einstellen des Betriebszustands, z.B. Rücksetzen der Statusbits

Darstellung von Maschinenbefehlen

- für den Programmier verständlich: **Assembler-Format** Transfer A to B: A → B

Abkürzung des Befehls (Mnemonic)	Operand1 , Operand2

Bsp.: TFR A, B

Die Operanden sind optional, z.B. ist oft einer der beiden Operanden bereits in der Abkürzung des Befehls enthalten, z.B. LDX #$2000: Lade 2000h in das Register X: 2000h → X

Da die ALU maximal zwei Operanden gleichzeitig verarbeiten kann, darf ein Maschinenbefehl **maximal zwei Operanden** haben.

Das Ergebnis einer Operation wird Zieloperand (**Destination Operand**), der andere Quelloperand (**Source Operand**) genannt. Bei Motorola/Freescale steht bei zwei Operanden der Zieloperand hinten (bei Intel steht er vorn).

Um die Befehle kurz zu halten (sh. unten), wird meist **kein separater Ergebnisoperand** angegeben, sondern der **Zieloperand durch** das **Ergebnis überschrieben.** (sogenannte 2-Adress-Maschine)

- Für die CPU müssen der Befehl und die Operanden binär kodiert werden: **Maschinen-Format**

Befehlscode (Opcode)	Adressierungsart	Register, Speicheradresse oder Konstante
bei m Befehlen	bei k Adressierungsarten	bei l Registern min. $\log_2 l$ Bit.
min. $\log_2 m$ Bit	min. $\log_2 k$ Bit	bei n bit Adressen oder Konstanten min. n bit.
	← für jeden Operanden genau ein Mal notwendig →	

Befehle ohne Operanden oder mit Registeroperanden sind kürzer (meist **Ein-Wort-Befehle**) als Befehle mit Speicheroperanden oder Konstanten (**Mehr-Wort-Befehle**).

- **Adressierungsarten**

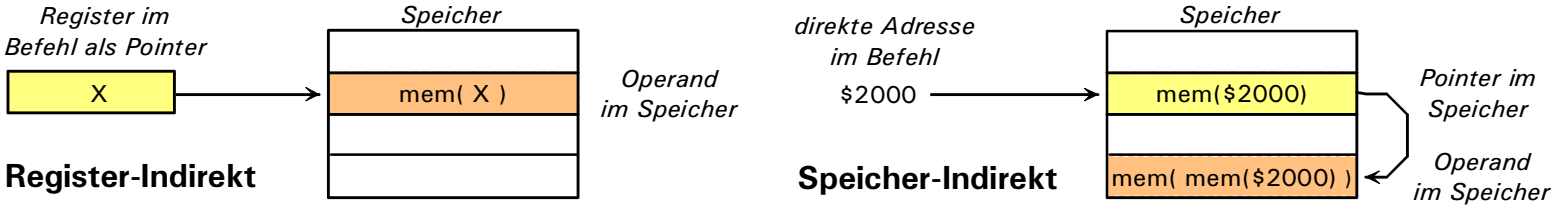
Praktisch alle CPUs haben dieselben Grundadressierungsarten (Bezeichnung bei HCS12 in ()) :

Register	Der Operand steht in einem Register (ist aber nicht schon im Befehlscode enthalten) Bsp.: TFR A, B A → B : A und B sind Register-addressiert Beim HCS12 selten, Register werden meist implizit adressiert, z.B. alternativ zu TFR A,B auch TAB.
Implizit oder **inhärent** (Inherent INH)	Der Operand ist implizit bereits im Befehlscode enthalten und muss nicht separat angeben werden. Bsp.: ABA A + B → A : beide Operanden implizit adressiert LDX #$2000 2000h → X : Zieloperand X implizit adressiert Beim HCS12 wird in der Regel mindestens 1 Operand implizit addressiert.
Unmittelbar (Immediate IMM)	Der Operand ist als Konstante direkt im Befehl enthalten (meist im 2. Befehlswort) Bsp.: LDX #$2000 2000h → X : Konstante 2000h unmittelbar adr. Beim HCS12 werden unmittelbare Operanden mit #... gekennzeichnet.
Direkt (Direct DIR 8bit, Extended EXT 16bit)	Der Operand steht im Speicher, seine Speicheradresse steht direkt im Befehl. Bsp.: LDX $2000 mem(2000h) → X : Zieloperand X implizit adressiert Operand bei Adresse 2000h ist direkt adressiert Beim HCS12 sind 8bit Adressen im Bereich 0 ... 255h (DIR) und 16bit Adressen (EXT) möglich.

Indirekt

Der Operand steht im Speicher, seine Adresse wird wie folgt bestimmt:

- **Register-Indirekt**: Pointer in einem Register, dessen Inhalt die Adresse des Operanden ist. Bsp.: INC mem(X) *1
- **Speicher-Indirekt**: Direkt adressierter Pointer im Speicher, der auf den eigentlichen Operanden zeigt. Bsp.: INC mem($2000) *1



*1 Achtung: Beide Adressierungsarten sind beim HCS12 so nicht möglich, sondern nur als Kombination mit anderen Adressierungsarten als IDX bzw. IDX2, siehe unten.

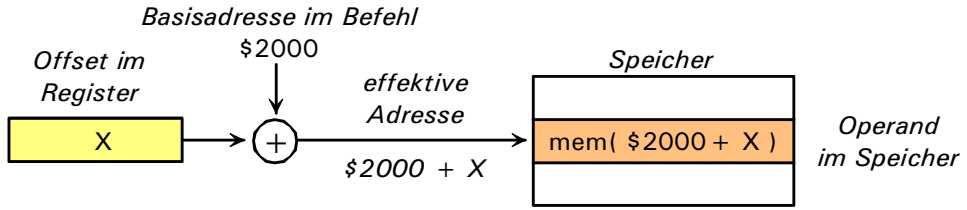
Aus den Grundadressierungsarten entstehen weiterte Adressierungsarten durch Kombination:

Indiziert

(Indexed IDX, IDX1, IDX2)

Der Operand steht im Speicher, seine Adresse wird aus einer direkten (Basis- oder Offset)-Adresse und dem Inhalt eines (Index)-Registers gebildet.

Bsp.: ADDA $2000, X A + mem($2000 + X) → A



Indiziert = Kombination von direkt und Register-indirekt, wobei beim HCS12 die direkte Adresse 5, 9 oder 11bit lang sein darf (Bezeichnungen IDX, IDX1, IDX2) und Varianten existieren, bei denen der Registerinhalt vor oder nach der Ausführung des Befehls inkrementiert oder dekrementiert wird.

(Relative REL)

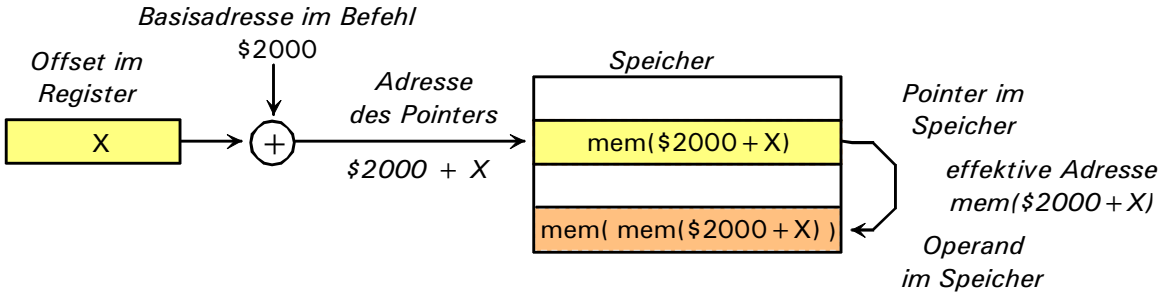
Für Verzweigungsbefehle existiert eine Variante, bei der PC (Program Counter, Instruction Pointer) als Indexregister verwendet wird, d.h. die Verzweigung erfolgt relativ zum aktuellen Programmzählerstand.

Indiziert-Indirekt

(Indexed-Indirect [IDX2])

Der Operand steht im Speicher. Seine Adresse steht in einem Pointer im Speicher. Der Pointer selbst wird indiziert adressiert.

Bsp.: ADDA [$2000, X] A + mem(mem($2000 + X)) → A



Einen Befehlssatz, bei dem jeder Operand mit jeder (sinnvollen) Adressierungsart angesprochen und jedes Register verwendet werden kann, nennt man einen **orthogonalen Befehlsatz**.

Um die binär kodierten Maschinenbefehle kurz zu halten, sind beim **HCS12** - wie bei vielen anderen einfachen Mikroprozessoren - bei den meisten Befehlen nicht alle Register und Adressierungsarten bzw. Kombinationen von Adressierungsarten zulässig, d.h. der **Befehlssatz** ist **nicht orthogonal**.

4.2 Programmiermodell: Register, Befehle, Adressierungsarten

Die direkte Programmierung in Maschinen/Assemblersprache ist mühsam und wird nur noch in Ausnahmefällen (Geschwindigkeits- oder Speicherplatzoptimierung) angewendet. Üblicherweise programmiert man CPUs in höheren Programmiersprachen (C/C++, ...), die dann durch einen **Compiler** übersetzt werden. Die Übersetzung kann relativ schematisch erfolgen:

Globale Variable	Speicherworte im Datenspeicher, in der Regel direkt adressiert
Lokale Variable	Register oder Stack (im Datenspeicher), Stack indirekt adressiert
Konstanten	Unmittelbare Adressierung
Pointer	Indirekte Adressierung
Arrays	Indizierte Adressierung
for, if, while, ...	Vergleichs- und Verzweigungsbefehle wie Compare, Jump, Branch
Unterprogramme	Unterprogrammsprünge und Rücksprünge wie Call, Return

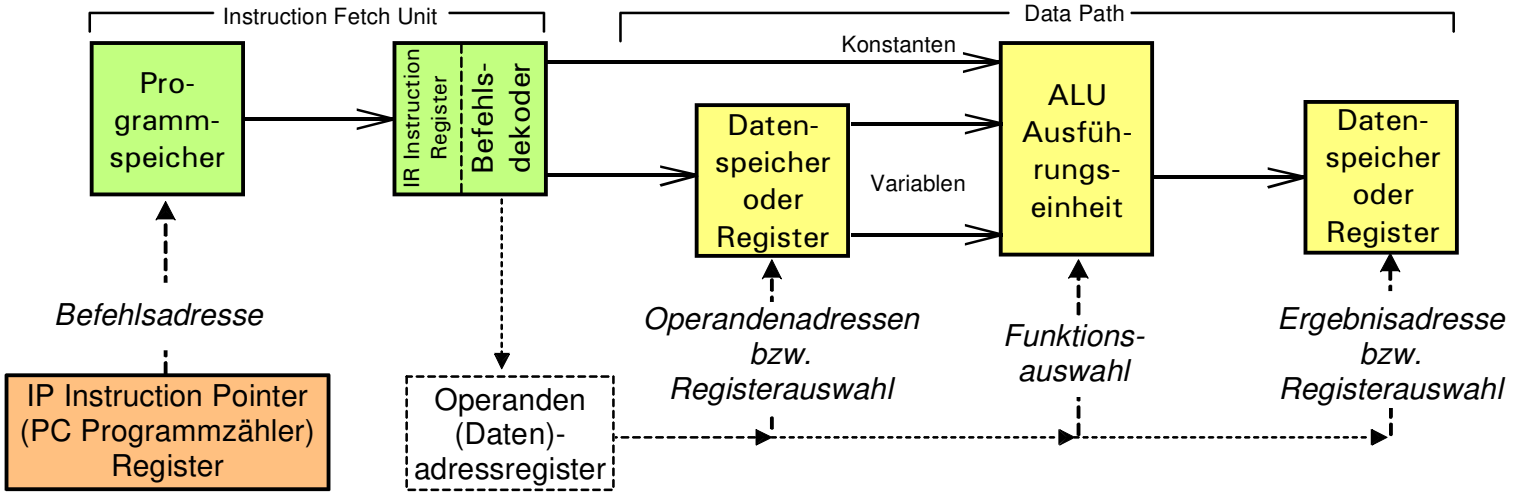
CPU-Familien

Während sich die Mikroarchitektur von CPUs ständig den Fortschritten der Halbleitertechnologie anpasst (siehe Kapitel 4.5), bleibt das Programmiermodell (Befehlssatz mit Registermodell und Adressierungsarten) über viele Jahre konstant bzw. wird aufwärtskompatibel weiterentwickelt. **CPU**s, die **dasselbe Programmiermodell** verwenden, werden als **CPU-Familie** bezeichnet.

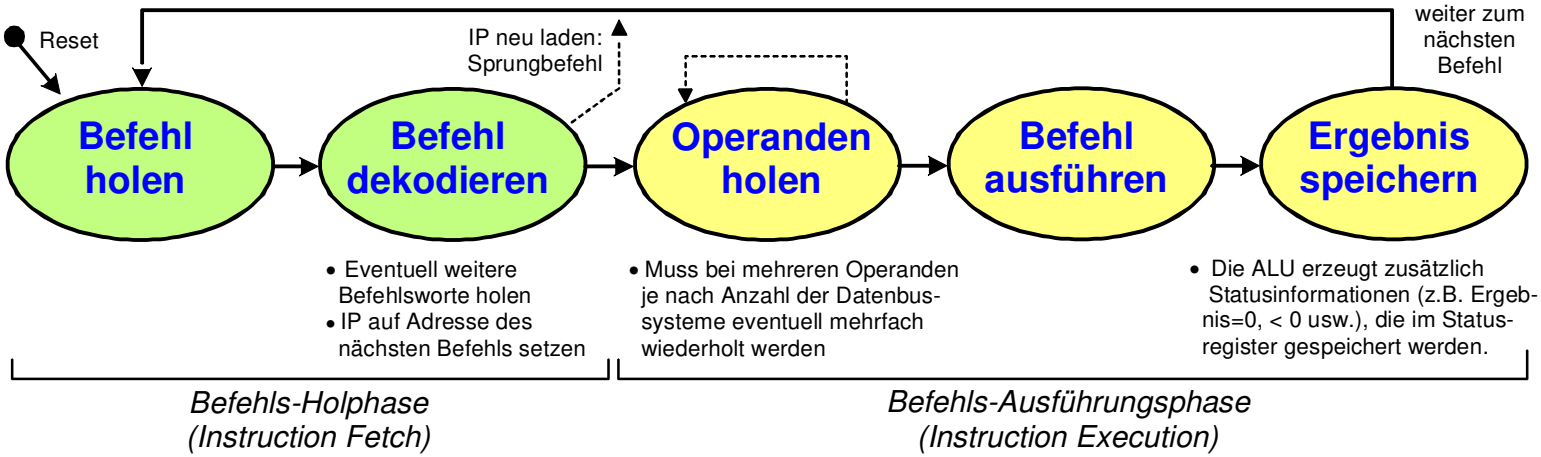
Bsp.: x86-CPUs (Intel Pentium/Core, AMD Athlon) sind aufwärtskompatibel zum 8086 (16bit CPU) von 1979, der von IBM im ersten PC 1981 eingesetzt wurde. Von 1985 (32bit CPU 80386) bis 2005 (Pentium D) hat sich das Programmiermodell praktisch gar nicht geändert und ist mittlerweile aufwärtskompatibel auf 64bit (Intel Core/Core2 i3/i5/i7, AMD64) erweitert worden.

4.3 Mikroarchitektur: Logischer Ablauf bei der Ausführung eines Programms

Beteiligte Hardwareeinheiten



Logischer Ablauf



Erläuterungen

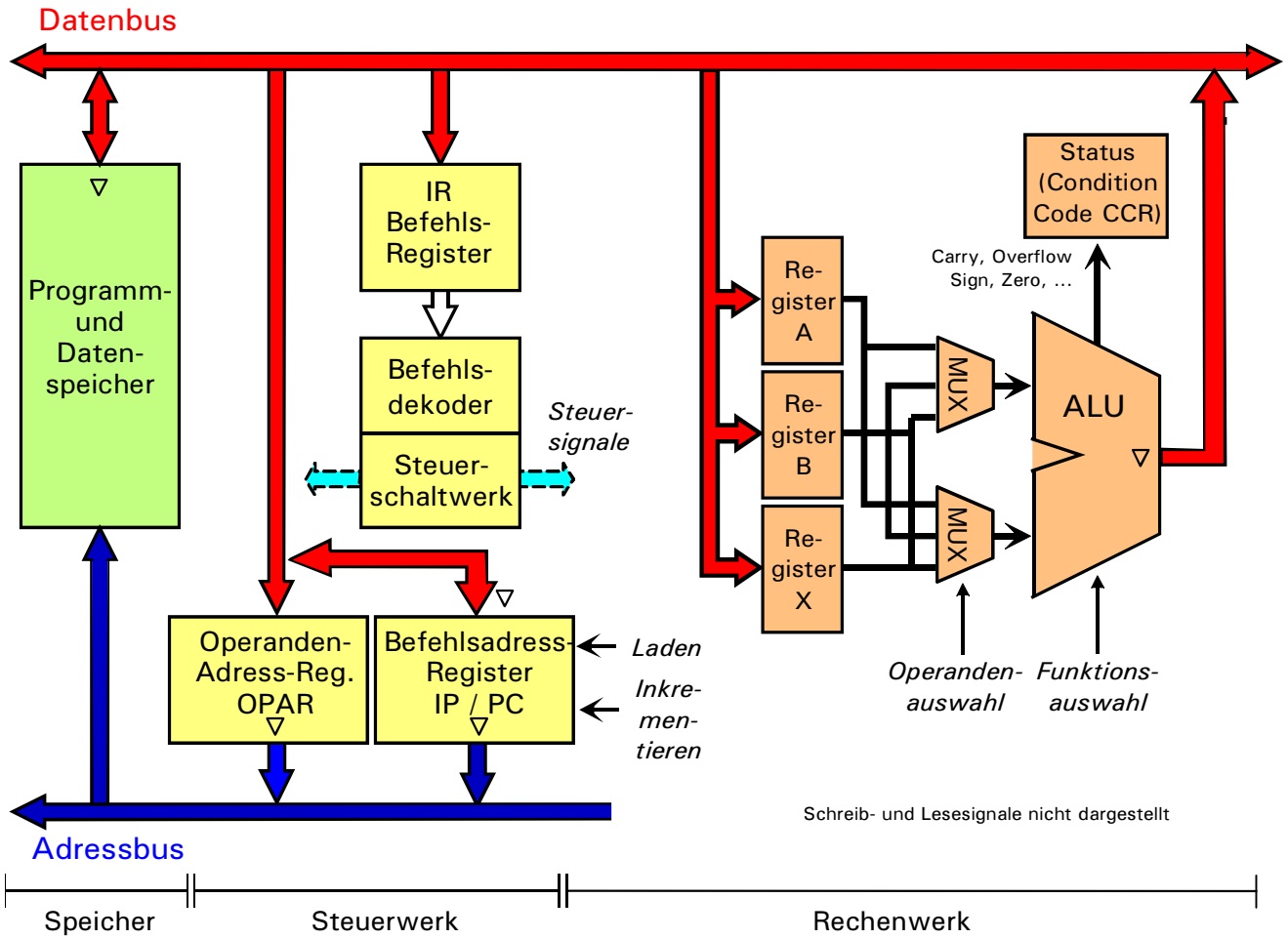
Befehl holen	mem(IP) → IR	**Befehlswort vom Programmspeicher in** die **CPU holen**. Die **Adresse des nächsten Befehls** steht im Steuerwerk **in** einem **Register**, das als **IP Instruction Pointer** (Befehlsadressregister) **oder PC Program Counter** (Programmzähler) bezeichnet wird. Der **Befehl** wird in der CPU **im IR Instruction Register** (Befehlsregister) **gespeichert**. Das Steuerwerk legt die Adresse auf den Adressbus, erzeugt das Lesesignal RD für den Speicher und das Schreibsignal für das IR. **Beim Einschalten** (Reset) **wird IP mit** einem **festen Wert initialisiert**, unter der der allererste Befehl im Programmspeicher stehen muss.
Befehl dekodieren	IP = IP + m (oder IP neu laden bei Sprung)	Das Befehlswort wird dekodiert. Dabei wird erkannt, welcher Befehl auszuführen ist und welche Operanden benötigt werden. **Eventuell** wird dabei erkannt, dass der Befehl aus einem **zweiten (oder weiteren) Befehlswort** besteht, wenn der Befehl nicht in ein einziges Speicherwort passt; dieses Befehlswort wird ebenfalls geholt. Wo es gespeichert wird, hängt vom Inhalt des weiteren Befehlswortes ab. **Häufig enthält** das **zweite Befehlswort** die **Adresse eines Operanden** und **wird in** einem **Operandenadressregister gespeichert**. **IP wird auf** die Adresse des **nächsten Befehl**s **gesetzt**. Bei linearem Programmablauf steht der nächste Befehl bei der nächsten Programmspeicheradresse, dann wird IP einfach um die Anzahl der Befehlsworte inkrementiert (daher statt IP häufig auch PC Program Counter). Bei einem Programmsprung wird IP mit der Adresse des Sprungziels neu geladen.
Operanden holen		Die **Operanden werden geholt** und an die Eingänge der ALU gelegt. Falls die Operanden im Datenspeicher stehen, legt das Steuerwerk die Operandenadresse auf den Adressbus und erzeugt das Lesesignal RD für den Speicher. Falls nur ein Datenbus vorhanden ist, aber mehrere Operanden geholt werden müssen, wird der Vorgang mehrfach wiederholt und die Operanden am Eingang der ALU ggf. zwischengespeichert. Falls die Operanden im Registerblock stehen, erzeugt das Steuerwerk das Auswahlsignal für das/die entsprechende/n Register.

Befehl ausführen		Die **ALU führt** die arithmetisch-logische **Operation** mit den Operanden **aus**. Das Steuerwerk legt dazu die entsprechenden Signale zur Auswahl der Funktion an die ALU.
Ergebnis speichern		Das **Ergebnis der Operation wird gespeichert**. Das Steuerwerk legt die Ergebnisadresse auf den Adressbus, das Ergebnis auf den Datenbus und erzeugt das Schreibsignal WR bzw. erzeugt das Auswahlsignal für das Register, falls das Ergebnis in einem Register gespeichert wird. Neben dem Ergebnis erzeugt die ALU meist auch Statussignale, die anzeigen, ob das Ergebnis 0 oder negativ war usw. Die **Statussignale werden in** einem speziellen Register des Registerblocks, dem **Statusregister gespeichert**.

- Die 5 Schritte können unterschiedlich lang dauern (Bsp.: Lesen eines Operanden aus dem Datenspeicher dauert in der Regel länger, d.h. mehr Takte, als aus einem Register des Registerblocks).
- Bei manchen Befehlen sind nicht alle 5 Teilschritte notwendig (Bsp.: Befehl ohne Operanden oder ohne Ergebnis).
- Im einfachsten Fall erfolgt der dargestellte Ablauf streng sequentiell. Um die Ausführungsgeschwindigkeit zu erhöhen, werden Schritte zeitlich parallel ausgeführt, soweit die Hardware dies zulässt (z.B. Holen von zwei Operanden gleichzeitig oder Holen des nächsten Befehls, während der aktuelle Befehl ausgeführt wird, wenn mehrere Bussysteme zwischen Speicher, Registerblock und ALU vorhanden sind).

➢ Welcher Detailablauf sich bei jedem Befehl ergibt, wird bei der Hardwareentwicklung der CPU festgelegt und als **Mikroarchitektur** bezeichnet. Sie bestimmt die Ausführungszeit von Befehlen, ist aber für den Programmierer nur von untergeordnetem Interesse.

Beispiel einer Mikroarchitektur



Hochsprachenebene:

S = var1 + var2;

Maschinenbefehlsebene:

Programmierer/Compilersicht

mem(100h) + RegA → Reg B

Mikroprogrammebene:

Logische Schritte der Ausführung

[IP++] → IR	*//Befehl holen*
[IP++] → OPAR	*//Op.adresse holen*
[OPAR] → RegX	*//Operand 1 holen*
RegA + RegX → RegB	

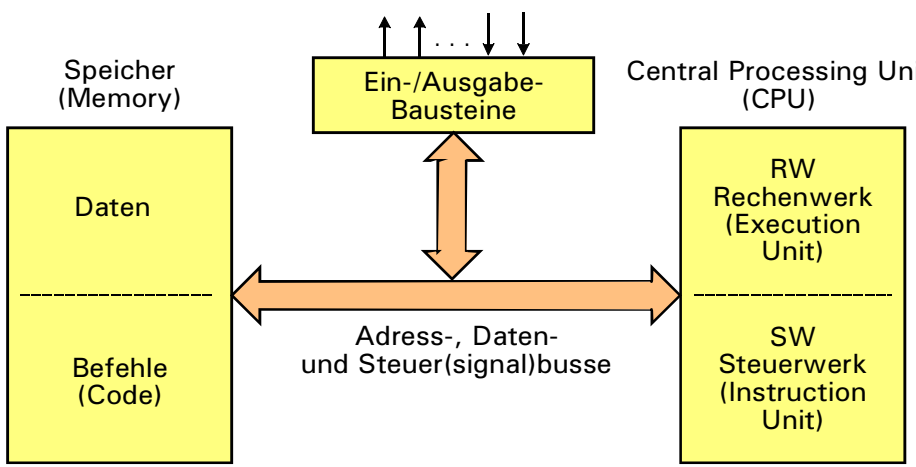
Steuerwerksebene:

Zeitliche Abfolge der Steuersignale

OE_{IP} aktivieren	*//Adr. ausgeben*
RD_{mem} aktivieren	*//Lesen*
WR_{IR} aktivieren	*//Speichern*
INC IP	*//Inkrementieren*
OE_{IP} aktivieren	*//Adr. ausgeben*
RD_{mem} aktivieren	*//Lesen*
WE_{OPAR} aktivieren	*//Speichern*
. . .	

4.4 Ausführungsformen von Rechnerarchitekturen

- **Von Neumann und Harvard-Architektur** (System Memory Architecture)

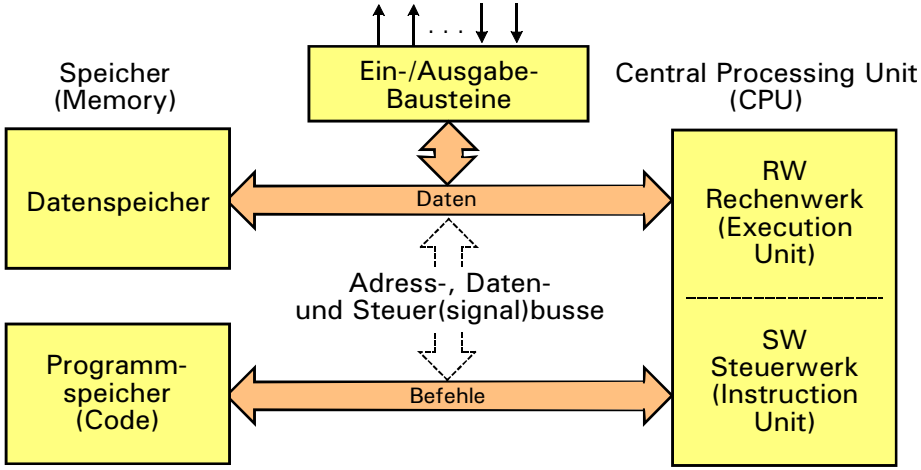


Von-Neumann-Architektur

Befehle und Daten im selben Speicher
Gemeinsames Bussystem und Adressraum

\+ Geringerer Aufwand

\+ Flexible Speicheraufteilung



Harvard-Architektur

Befehle und Daten in getrennten Speichern
Separate Bussysteme und Adressräume

\+ Gleichzeitiger Zugriff auf Code und Daten

schneller

Einfachere CPUs sind meist in **Von-Neumann**-Architektur aufgebaut.

Heutige **Hochleistungs-CPUs** verwenden meist **intern** eine **Harvard**-Architektur (mit getrennten Cache-Speichern für Befehle und Daten) und für den **extern**en Speicher eine **Von-Neumann**-Architektur.

- **CISC und RISC Befehlssätze** (Instruction Set Architecture)

CISC: Complex Instruction Set Computer	**RISC:** Reduced Instruction Set Computer
Optimierungsziel	
Einfach zu programmieren	Einfache, sehr schnelle Hardware
Architektur verwendet	
Mächtige Befehle und Adressierungsarten Bsp.: Datenblock kopieren in einem Befehl Indizierte Adressierung mit Konstante und 2 Indexregistern	**Einfache Befehle und Adressierungsarten**
Folge	
Befehle unterschiedlich lang Operanden in Registern und/oder Speicher Kompliziertes **Steuerwerk mit Mikroprogramm**	Befehle einheitlich 1 Speicherwort lang Operanden und Ergebnisse bei arithmetisch-logischen Operationen nur in Registern Datenzugriff auf den Speicher nur beim Laden oder Speichern von Registerinhalten (Load- and Store-Architektur) **Steuerwerk** mit **festverdrahtem Schaltwerk**.

Heute Mischformen üblich: CISC-Grundarchitektur mit RISC-Untermenge für häufig verwendete Befehle oder RISC-Grundkonzept mit einigen komplexeren CISC-Befehlen.

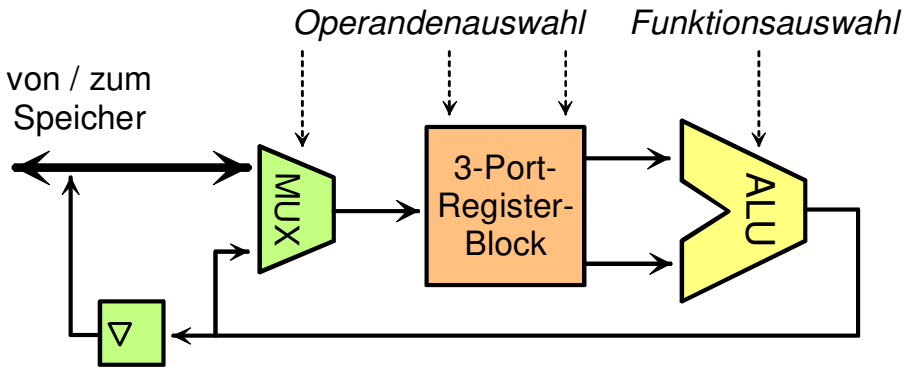
- **2-Adress-CPU, 3-Adress-CPU, Akkumulator- und Stack-Rechenwerke**

Bei der Realisierung des Datenpfads (Verbindung Rechenwerk – Registersatz) gibt es viele Möglichkeiten. Da alle Operanden in einem Befehl spezifiziert werden müssen, wächst die Befehlslänge mit der Anzahl der Operanden. Daher gilt bei praktisch allen CPUs: **Maximal 2 Operanden und 1 Ergebnis je Befehl.** Selbst bei dieser Einschränkung sind noch weitere Vereinfachungen möglich (Beispiel Addition):

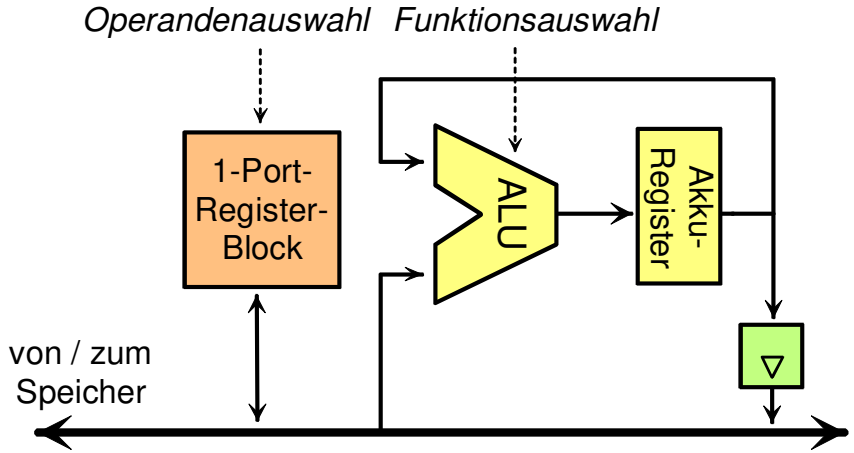
3-Adress-CPU typ. für Hochleistungs-RISC-CPUs	**C = A + B** • Beide Operanden A, B und das Ergebnis C dürfen in drei verschiedenen Registern stehen. • Einer der beiden Operanden kann auch eine Konstante sein. • Seltener: Einer der Operanden oder das Ergebnis darf im Speicher stehen.
2-Adress-CPU weit verbreitet	**A = A + B** • Das Ergebnis überschreibt einen der beiden Operanden. • Rest wie bei der 3-Adress-CPU.
Akkumulator-Maschine veraltet, uP der 1. Generation	**Akku = Akku + B** • Ein Operand und das Ergebnis stehen grundsätzlich immer im selben Register, dem Akkumulator-Register. Der zweite Operand darf in einem beliebigen Register oder im Speicher stehen oder eine Konstante sein.
Stack-Maschine Spezialanwendungen, z.B. x87-FPU	**R0 = R0 + R1** • Statt Registern gibt es einen Stapelspeicher (Stack). Die beiden Operanden werden immer vom Stack heruntergenommen, das Ergebnis immer au dem Stack abgelegt.

Beispiele:

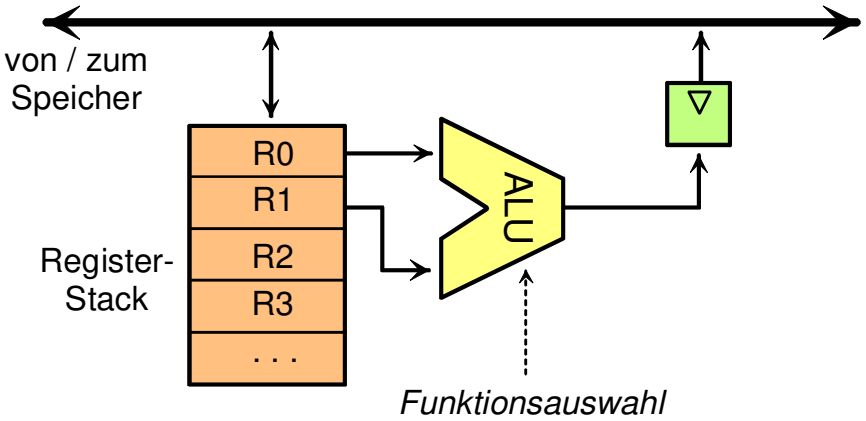
2- oder 3-Adress-CPU



Akkumulator-Maschine



Stack-Maschine



Operationen der Stack-Maschine

- PUSH x: Speichern auf dem Stapel

  …, R2 → R3, R1 → R2, R0 → R1, Bus → R0

- POP x: Lesen vom Stapel

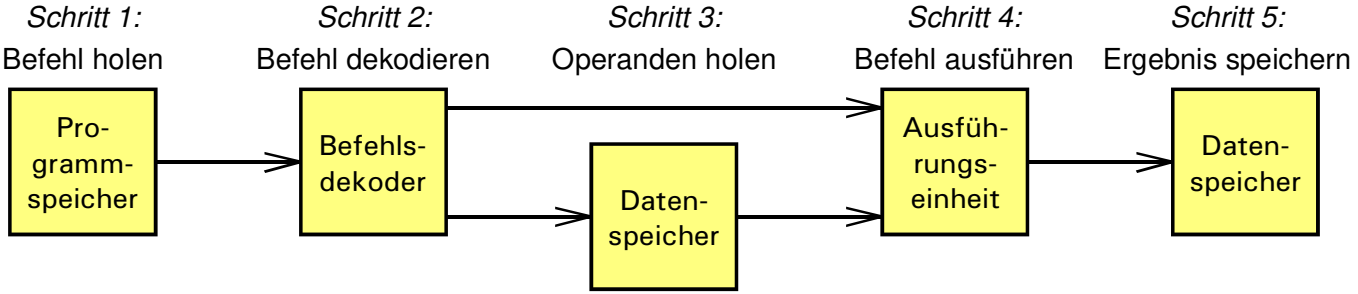
  R0 → Bus, R1 → R0, R2 → R1, R3 → R2, …

- ADD oder andere arithmetisch-logische Operationen

  R0 + R1 → R0, R2 → R1, R3 → R2, …

4.5 Maßnahmen zur Leistungssteigerung

- Schritte zur Ausführung eines Programms und beteiligte Rechnerbaugruppen



Verbesserung der Speicherschnittstelle

- Breiterer Datenbus und höhere Bustaktfrequenz → Probleme: EMV, Signallaufzeiten

  Bsp.: x86-CPUs (Intel Pentium/Core, AMD Athlon) 64bit Datenbus mit bis zu > 200MHz
  CPUs auf Grafikkarten bis zu 256bit Datenbus, interne Datenbusse bis zu 1024bit

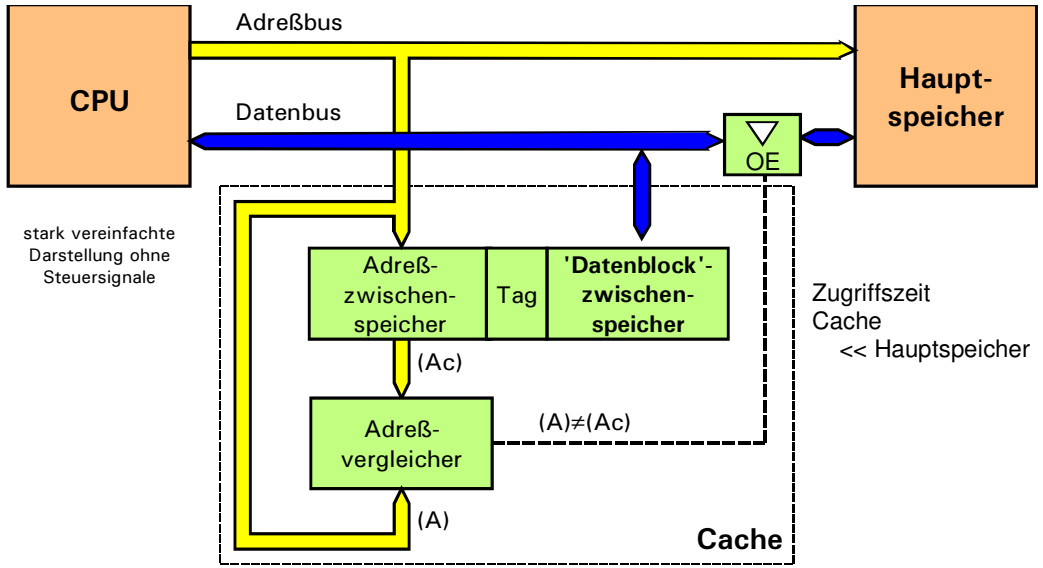
- Kürzere Speicherzugriffszeiten: SRAM statt DRAM, On-Chip-Speicher statt externe Bausteine

  Problem: Da SRAM sehr aufwendig (6 Transistoren/Bit statt 1 Transistor/Bit bei DRAM) sind nur kleine On-Chip-SRAM-Speicher möglich.

  Abhilfe: **Cache-Speicher**
  Schneller kleiner Speicher, in dem immer derjenige Programmcode und diejenigen Daten gespeichert sind, die aktuell benötigt werden. Funktioniert wegen des Prinzips der **Lokalität von Software**:

  - Programmbefehle stehen unmittelbar hintereinander im Speicher
  - Verwendete Befehle und Daten werden oft kurze Zeit später wieder benötigt

Cache: Schneller Zwischenspeicher



Content-Adressable Memory (CAM)

- Erstmals benötigte Speicherworte

  Hauptspeicher → CPU, Cache
  Adresse in Cache-Adressspeicher

  Dabei wird in der Regel gleich ein ganzer Block von Speicherworten gelesen (Cache Line).

- Bei jedem Speicherzugriff prüft der Adressvergleicher, ob Adresse und Speicherwort schon im Cache sind:

→ falls ja: Datum aus Cache → CPU
(Cache Hit)

→ falls nein: wie beim ersten Zugriff
(Cache Miss)

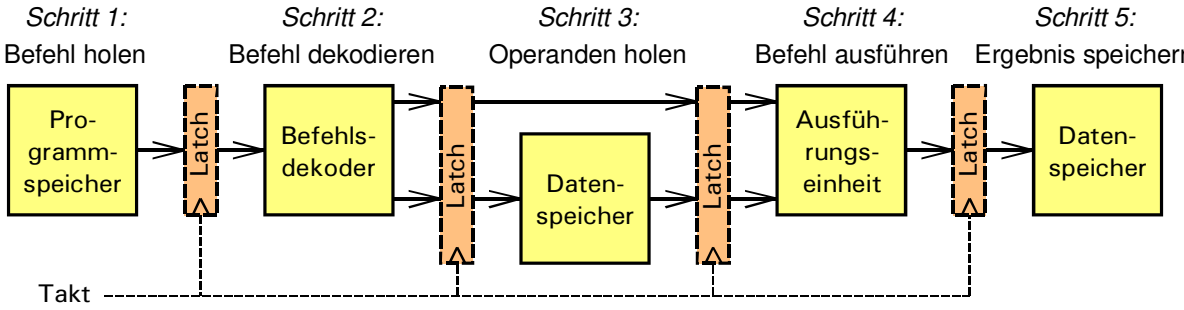
- Verdrängungsstrategie bei vollem Cache: Am längsten nicht benötigten Block löschen (**L**east **R**ecently **U**sed) → Für lesende Speicherzugriffe unkritisch.

→ Für schreibende Speicherzugriffe sicherstellen, dass Cache und Hauptspeicher konsistent bleiben (Cache-Kohärenz):

Write Through Cache: Schreiben immer im Hauptspeicher, Cache beschleunigt nur Lesen

Write Back Cache: Schreiben zunächst nur im Cache. Wenn Block verdrängt wird, Block in den Hauptspeicher zurückschreiben, falls er verändert wurde (Merker: Dirty Tag)

Pipelining: Zeitverschachteltes Ausführen von Befehlen (*Instruction Level Parallelism ILP 1*)



Programmspeicher	Befehlsdekoder	Datenspeicher	Ausführungseinheit	Datenspeicher	↓ t
Befehl 1 holen					1
Befehl 2 holen	Befehl 1 dekodieren				2
Befehl 3 holen	Befehl 2 dekodieren	Operanden 1 holen			3
Befehl 4 holen	Befehl 3 dekodieren	Operanden 2 holen	Befehl 1 ausführen		4
Befehl 5 holen	Befehl 4 dekodieren	Operanden 3 holen	Befehl 2 ausführen	Ergeb. 1 speichern	5
Befehl 6 holen	Befehl 5 dekodieren	Operanden 4 holen	Befehl 3 ausführen	Ergeb. 2 speichern	6
. . .	. . .	. . .	. . .	. . .	...

Bedingung: Alle Befehle verwenden alle Stufen evtl. Leerschritte einfügen

Signallaufzeit aller Stufen gleich bzw. langsamste Stufe bestimmt den Takt.

Latenz: Dauer vom Holen eines Befehls bis zum Speichern seines Ergebnisses hier: 5 Takte

Durchsatz: Anzahl der Befehle je Zeit hier: 1 Befehl je Takt (ohne Pipelining: 1 Befehl je 5 Takte), Einheit MIPS

Vergleich

	ohne Pipeline	*mit n stufiger Pipeline*
Taktperiode	$T_{oP} = \sum_{i=1}^{n} T_i$ mit T_i Durchlaufzeit der Stufen 1 … n	$T_{mP} = \max(T_1, \ldots, T_n) < T_{oP}$ Im Idealfall $T_{mP} = T_{oP} / n$, d.h. n fache Taktfrequenz
Latenz	T_{oP}	$n \cdot T_{mP}$ imIdealfall T_{oP}
Durchsatz	1 Befehl je T_{oP}	1 Befehl je T_{mP} Im Idealfall 1 Befehl je T_{oP} / n

Probleme (Pipeline Hazards)

- Konflikte auf dem Datenbus → Abhilfe: Getrennte Busse für Befehle, Operanden, Ergebnisse

  Bsp. im Takt 5: Befehl 5 holen – Operanden 2 holen – Ergebnis 1 speichern

- Sprungbefehle, insbesondere bedingte Sprungbefehle, bei denen das Sprungziel vom Ergebnis des vorigen Befehls abhängt → Abhilfe: Sprungvorhersage (Branch Prediction)

- Datenabhängigkeiten

  Bsp.: Wenn Ergebnis von Befehl 1 Operand von Befehl 3 ist (Takt 5)

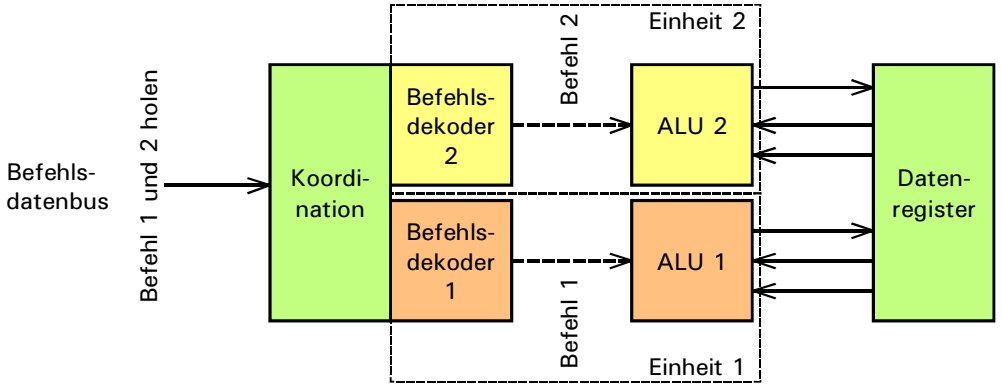
  → Abhilfe: Ergebnis 1 direkt zur Ausführungseinheit führen (Data Forwarding)

  Bsp.: Wenn Ergebnis von Befehl 2 Operand von Befehl 3 ist → Abhilfe: Befehle umsortieren

Superskalare Architektur: Paralleles Ausführen von Befehlen (*ILP2: Multiple Issue*)

- **Gleichzeitige** Bearbeitung mehrerer Befehle in **mehreren** Ausführungseinheiten

  Bsp. mit 2 Ausführungseinheiten:



Vergleich: Theoretisch bei m superskalaren Einheiten Befehlsdurchsatz x m

Probleme:

- **Konflikte auf dem Datenbus** wie beim Pipelining
- **Datenabhängigkeiten** zwischen parallelen Befehlen wie beim Pipelining

  Bsp.: Befehlsfolge $R1 + 1 \rightarrow R2$, $R3 + 2 \rightarrow R4$ → beide Operationen gleichzeitig möglich

  Befehlsfolge $R1 + 1 \rightarrow R2$, $R2 + 2 \rightarrow R4$ → 2.Operation erst nach 1.Operation ausführbar

  → Abhilfe: Reihenfolge der Befehle durch Programmierer, Compiler oder automatisch durch das Steuerwerk umsortieren (Out of Order Execution)

Programmbeispiel für In Order Execution und Out of Order Execution:

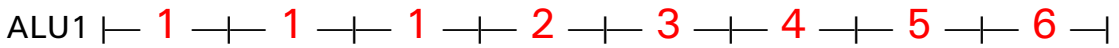
Befehl 1:	[R0]	→	R1
Befehl 2:	R1 + R2	→	R4
Befehl 3:	R5 + R6	→	R5
Befehl 4:	R6 + R3	→	R2
Befehl 5:	R1 + R2	→	R1
Befehl 6:	R2 + R3	→	R3

Annahmen:

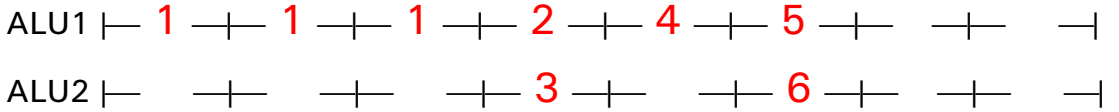
Speicherzugriff im Befehl 1 dauert 3 Takte

Alle übrigen Befehle dauern 1 Takt

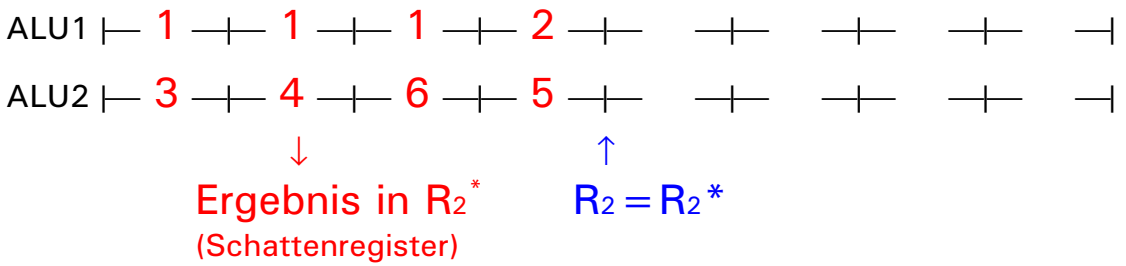
Ablauf mit 1 ALU → **8 Takte**



Ablauf mit 2 ALUs - In-Order-Execution → **6 Takte**



Ablauf mit 2 ALUs - Out-of-Order-Execution → **4 Takte**

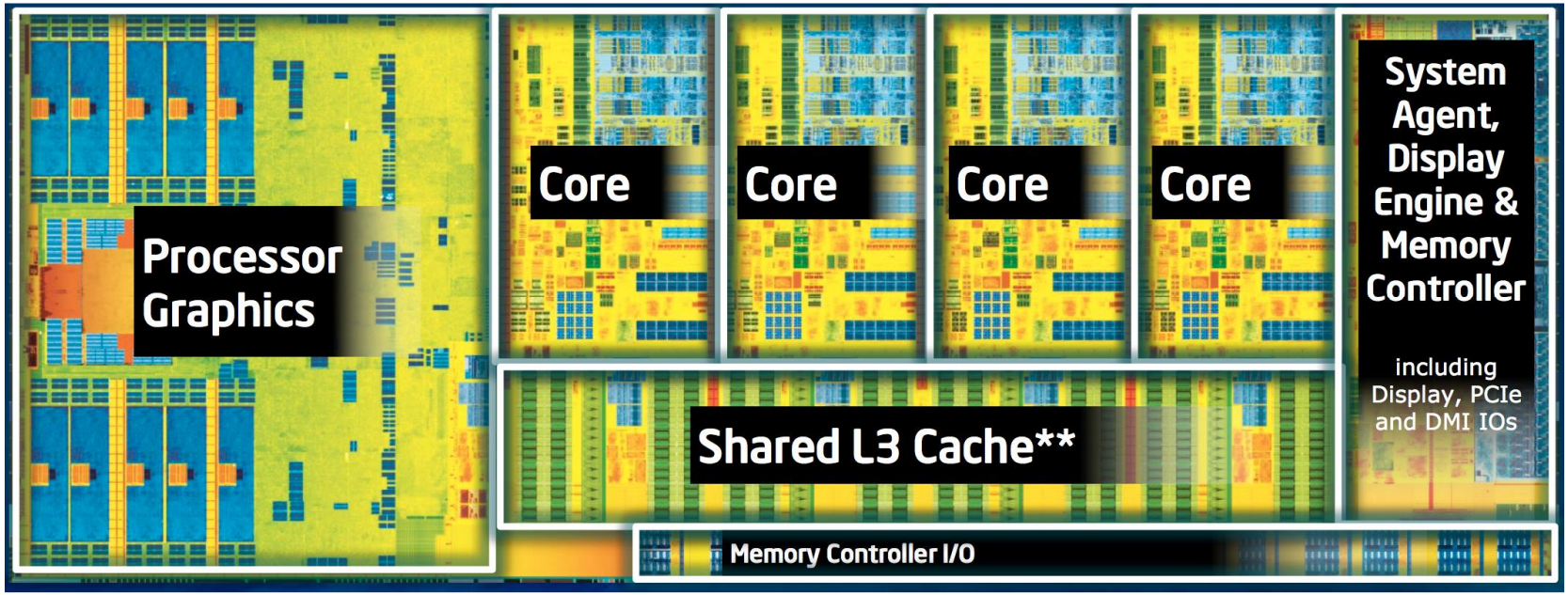


- Statt Umspeichern der Ergebnisse vom Schattenregistern R_X* zum normalen Register R_X → Umschalten der Registerauswahlsignale zwischen R_X* und R_X (**Register Renaming**)
- Superskalare Ausführungseinheiten wegen Aufwand oft nicht symmetrisch, d.h. nicht jede kann alle Befehl ausführen
- Datenabhängigkeiten durch Out-of-Order im Compiler bzw. in der Hardware nur begrenzt auflösbar
- Theoretische Verbesserung um den Faktor m durch m superskalare Einheiten bzw. um den Faktor n durch n Pipelinestufen wird praktisch meist nicht erreicht

Extremfall Multi-Processing:

Mehrere vollständige CPUs parallel (Multi-Core-Architektur, **SMP Symmetric Multi Processing**)

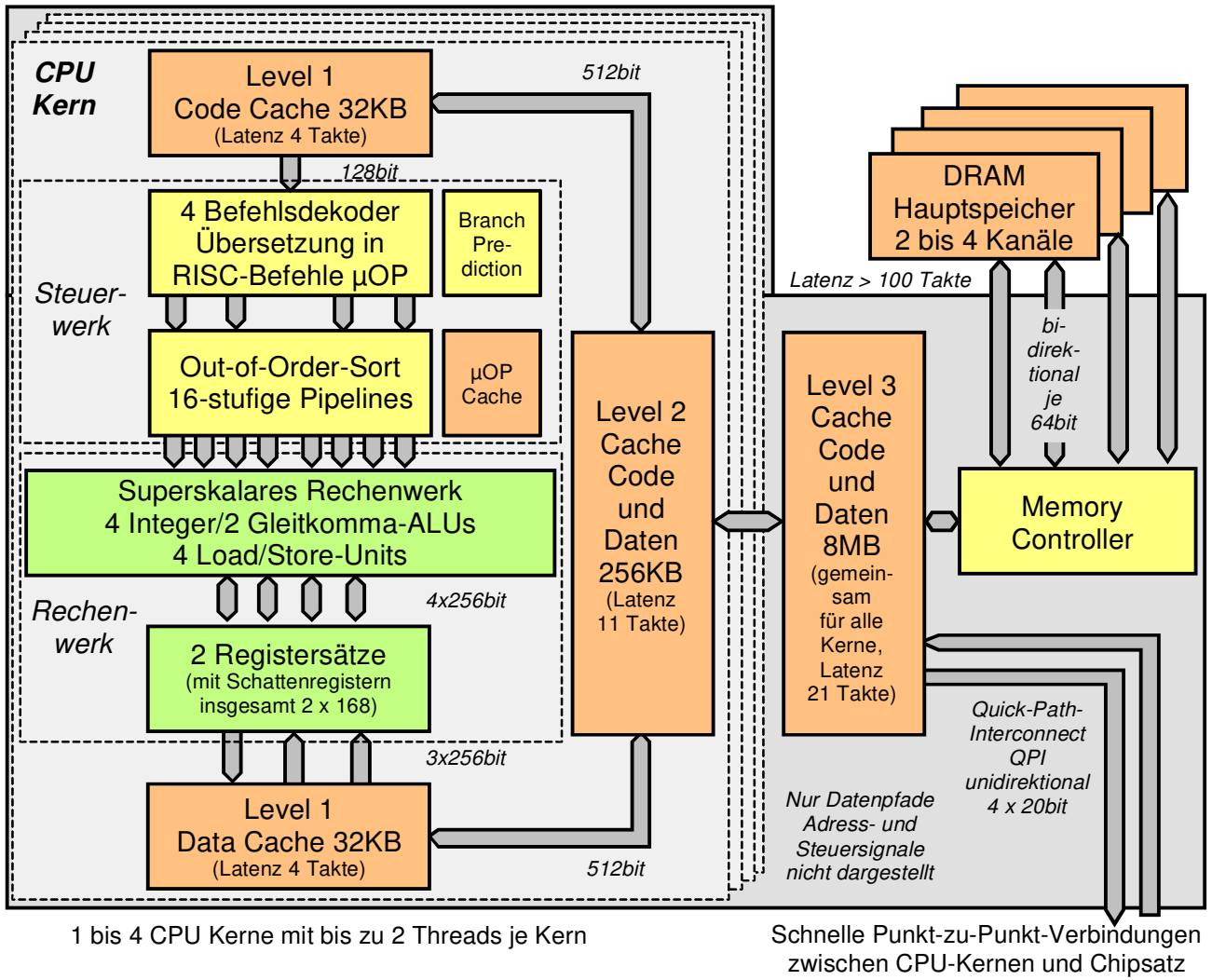
Intel Core i7-4770K
(x86-64 Haswell)
für Desktop-Rechner
(Markteinführung
Sommer 2013)



- 4 CPU Kerne mit je 2 x 32KB Level 1 Cache und je 256KB Level 2 Cache
- Gemeinsamer 8MB Level 3 Cache für alle Kerne
- CPU Takt 3.5 GHz (wenn alle 4 Kerne aktiv sind, Einzelkern bis zu 3.9GHz), Verlustleistung nominal 84W
- Halbleiterprozess mit 22nm Strukturbreite, Chipgröße ca. 180mm²
- Ca. 1,4 Milliarden Transistoren (davon ca. 200 Mio für GPU, ca. 60 Mio je CPU Kern)
- Integrierter Grafik-Prozessor GPU
- Integrierte Spannungsregler für CPU-Kerne
- Gehäuse mit 1150 Pins

Quelle: http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested

Mikroarchitektur x86-64 Core Haswell i7 22nm, Stand 06/2013



1 bis 4 CPU Kerne mit bis zu 2 Threads je Kern

Schnelle Punkt-zu-Punkt-Verbindungen zwischen CPU-Kernen und Chipsatz

Programmiermodell:

- **Von Neumann-CISC** CPU

Tatsächliche Mikroarchitektur:

- **Intern**er Speicher in **Har-vard**-Struktur, **extern**er Speicher in **Von-Neumann**-Struktur
- 80x86-CISC-Befehlssatz wird **intern** in 4 Befehlsde-kodern in **RISC-Befehlssatz** übersetzt
- **Superskalares Rechenwerk** (Multiple Issue mit max. 8 RISC-Befehlen parallel)
- **Mittellange Pipeline** mit ca. 16 Stufen und Sprungvor-hersage
- **Out-of-Order**-Ausführung mit umfangreichem Satz von **Schattenregistern**
- Dynamische **Taktfrequenz-umschaltung**

Definition und Messung der Rechenleistung:

Betrachtung auf der Ebene	*eines Maschinenbefehls*	*eines Programms*
Rechen-/Ausführungszeit T_{Rechen} Wichtig aus Sicht eines einzelnen Anwenders	Zeit vom Beginn der Holphase bis zum Ende der Ausführungsphase	Zeit vom Starten des Programms bis zum Beenden des Programms (gegebenenfalls ohne die Zeit, die das Programm auf Benutzereingaben warten muss).
Durchsatz D Wichtig aus Sicht des Betreibers eines Servers	Anzahl der Befehle, die je Zeit abgearbeitet werden	Anzahl der Aktivitäten, die je Zeit abgearbeitet werden, bei Datenbanksystemen z.B. Transaktionen
	Beim Pipelining und bei superskalaren Einheiten ist: $D \geq \frac{1}{T_{Rechen}}$	

Die Ausführungszeit eines Programms lässt sich abschätzen als

$$T_{Rechen} = \frac{\text{Anzahl der Befehle des Programms} \cdot \text{Durchschnittliche Anzahl der Takte je Befehl}}{\text{Taktrate}}$$

Die Taktrate wird dabei in der Regel durch die Halbleitertechnik vorgegeben, während die Mikroarchitektur die „durchschnittliche Anzahl der Takte je Befehl“ (CPI: Clock cycles Per Instruction) bestimmt.

Kapitel 5: Rechnerperipherie

5.1 Digitale Ein- und Ausgänge

5.2 Analog-Digital-Umsetzer und Digital-Analog-Umsetzer

5.3 Impulssignal-Ein- und Ausgänge

5.4 Serielle Schnittstellen **(Kommunikationsschnittstellen)**

Aus Sicht der CPU erscheinen Peripheriebausteine wie gewöhnliche Speicherzellen oder Register, die gelesen bzw. geschrieben werden können:

- **Memory Mapped Input / Output**: Peripheriebausteine im normalen CPU-Adressraum, Schreiben und Lesen über normale Datentransportbefehle wie MOV (typisch für Freescale HCS12 CPUs).
- **Isolated Input / Output**: Separater Adressraum für die Peripheriebausteine, Schreiben und Lesen mit speziellen Befehlen wie IN, OUT (typisch für Intel x86 CPUs).

Das Lesen der Speicherzellen bzw. Register durch die CPU erfolgt

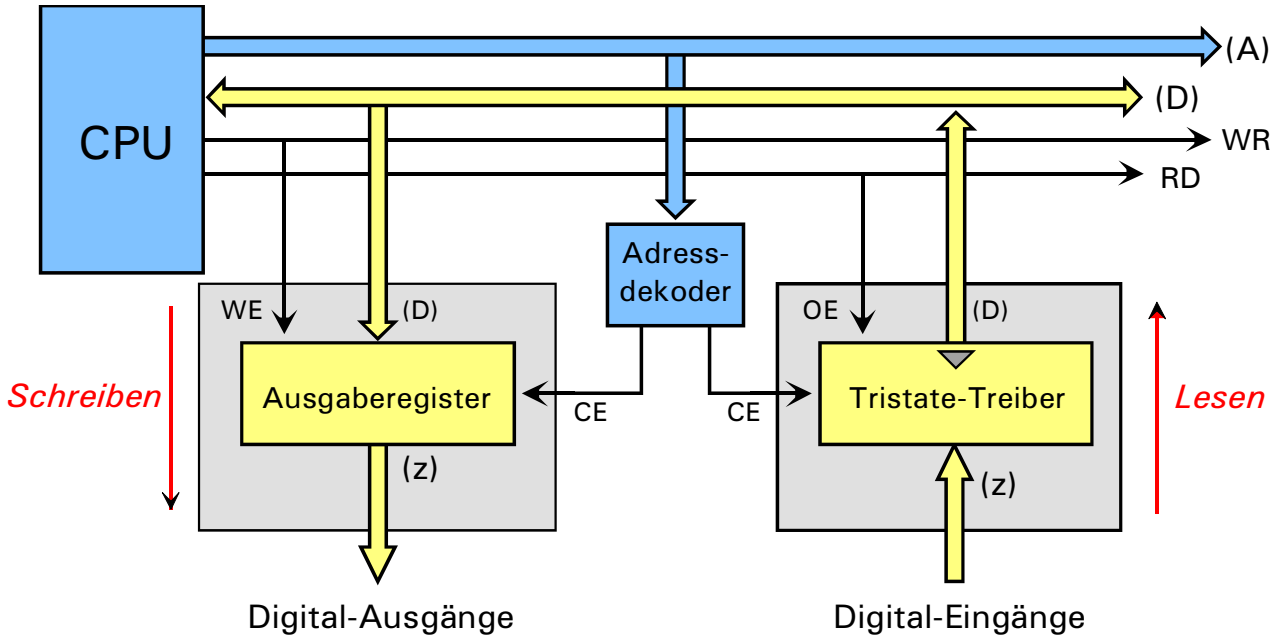
- bei durch Abfragen bei Bedarf oder zyklisch **(Polling)**
- wenn der Peripheriebaustein die CPU durch ein **spezielles Rückmeldesignal** von einem Ereignis informiert, z.B. der Änderung eines Signals **(Interrupt)**

Das Schreiben der Speicherzellen bzw. Register wird von der CPU bei Bedarf ausgeführt.

5.1 Digitale Ein- und Ausgänge

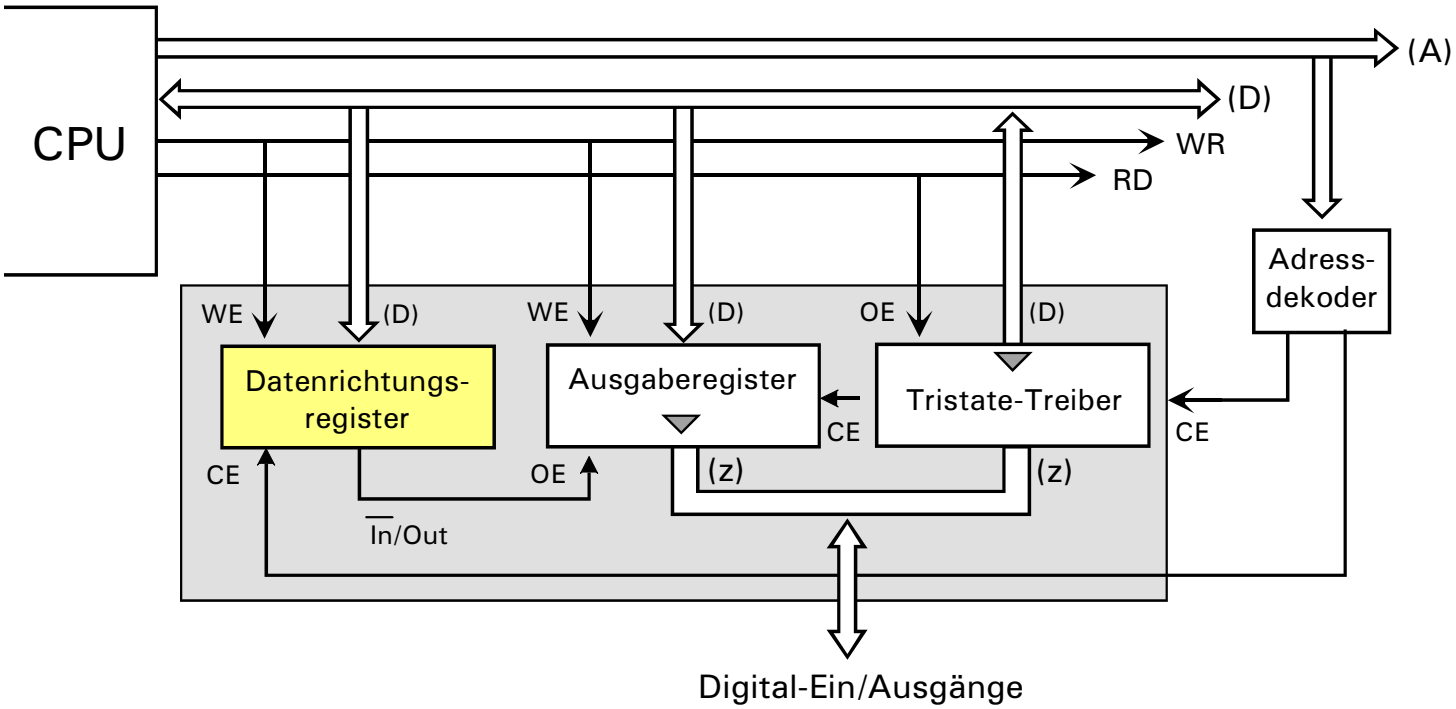
Im einfachsten Fall besteht ein Ein- bzw. Ausgang nur aus einer Leitung (1bit), häufig werden aber mehrere Leitungen (oft 8 oder 16bit) zusammengefasst: **Input Port bzw. Output Port**

- Digitale **Ausgänge** sind aus Sicht der CPU **Register**, in die geschrieben wird. Die geschriebenen Werte erscheinen unmittelbar als Signale an den Ausgängen und bleiben bis zum nächsten Schreibvorgang erhalten.
- Digitale **Eingänge** sind aus Sicht der CPU **Tristate-Treiber**, die ausgelesen werden. Dabei werden die jeweils aktuellen Signale gelesen, eine Speicherung im I/O-Port erfolgt nicht.



5.1 Digitale Ein- und Ausgänge

- Um Anschlüsse zu sparen, werden Ein- und Ausgänge häufig zusammengefasst (**I/O Port**). Ob ein Anschluss als Eingang oder Ausgang arbeitet, kann entweder für jeden Anschluss separat oder für eine Gruppe von Anschlüssen gemeinsam über ein **Datenrichtungs-Register** eingestellt werden:



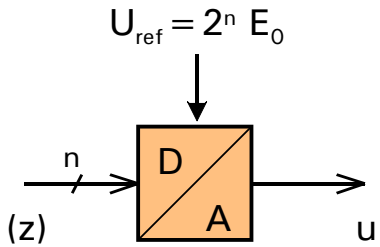
- Auch wenn der Port als Ausgang verwendet wird, kann der ausgegebene Wert bei Bedarf über den Tristate-Treiber zurückgelesen werden.
- Oft ist es zusätzlich möglich, die elektrische Charakteristik des Ein- oder Ausganges über ein weiteres Steuerregister umzuschalten: Gegentakt-(Push-Pull) oder Open-Collector-Ausgang, TTL- oder CMOS-Schaltschwelle, optionaler Pull-Up- oder Pull-Down-Widerstand, High-Side- oder Low-Side-Schalter

5.2 Analog-Digital- und Digital-Analog-Umsetzer

5.2.1 Aufgabenstellung

- Umwandlung einer n bit Dualzahl (z) in eine proportionale Analogspannung u bzw. umgekehrt

Digital → Analog

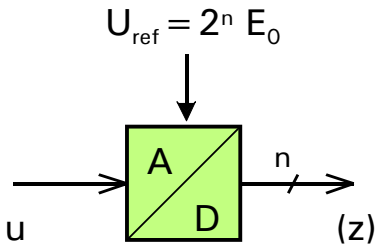


$$u = E_0 \cdot z$$

DAC

Digital to Analog Converter

Analog → Digital

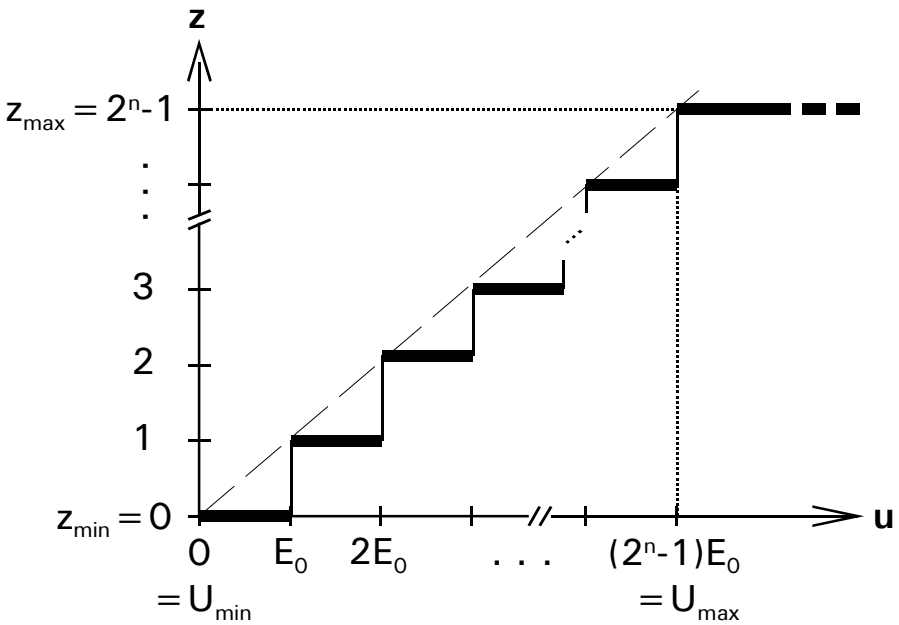


$$z = (int)\ \frac{u}{E_0}$$

ADC

Analog to Digital Converter

Zusammenhang zwischen Analog- und Digitalwert (unipolare Quantisierungskennlinie)

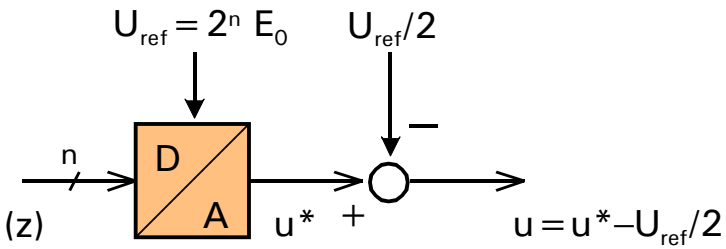


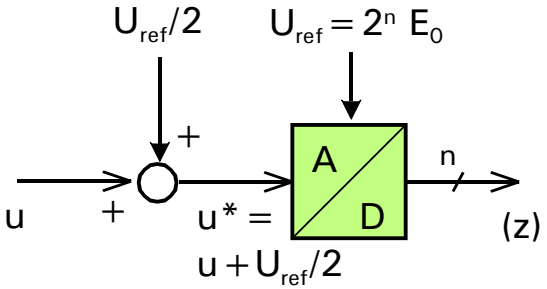
Falls z eine natürliche Zahl (Betragszahl) mit $0 \le z \le 2^n - 1$ ist, gilt: unipolarer Betrieb $U_{min} = 0$

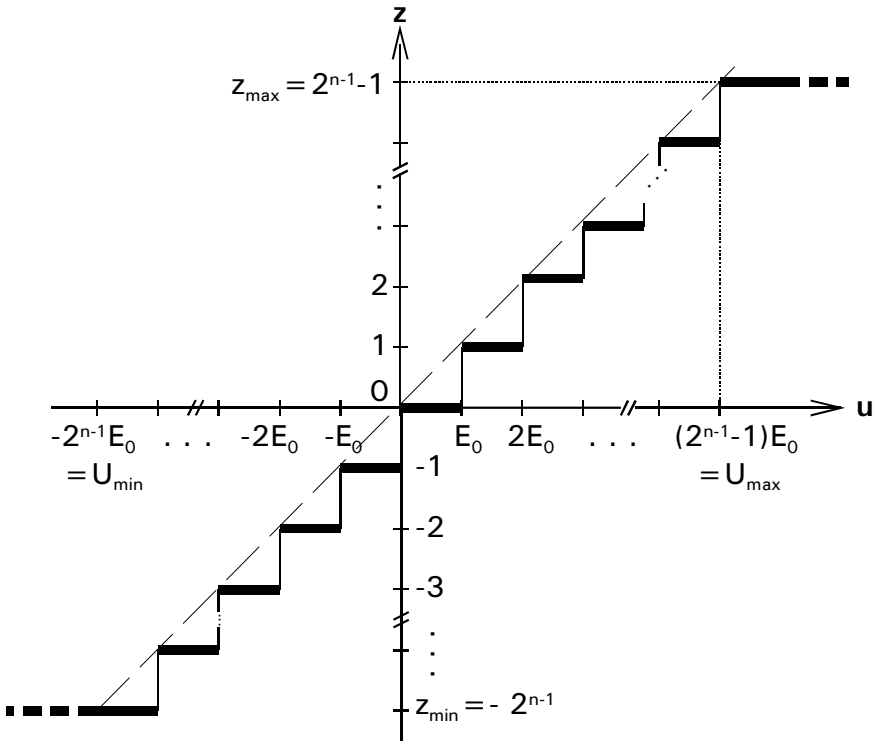
Spannungsbereich: $U_{max} - U_{min} = (2^n - 1)\ E_0$ Schrittspannung (Auflösung): $E_0 = \frac{U_{max} - U_{min}}{2^n - 1}$

- Bipolarer Betrieb

  Die meisten A/D- und D/A-Umsetzer können nur unipolare Signale direkt verarbeiten. Um einen symmetrischen bipolaren Spannungsbereich zu erhalten, addiert bzw. subtrahiert man auf der Analogseite die Spannung $U_{ref}/2 = 2^{n-1} E_0$







Bipolare Quantisierungskennlinie

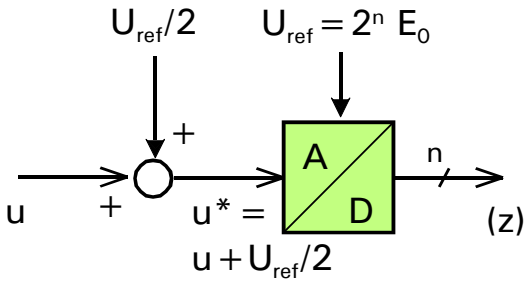
Nun ist z eine ganze Zahl im Bereich $-2^{n-1} \leq z \leq 2^{n-1}-1$

Für den Spannungsbereich und die Schrittspannung ergeben sich dieselben Werte wie beim unipolaren Betrieb.

Achtung:

Im bipolaren Betrieb ergibt sich für die Codierung von (z) nicht der übliche 2er-Komplement-Code, sondern der sogenannte **Dual-Offset-Code**

Beispiel für n = 8: $U_{ref}/2 = 2^{8-1} E_0 = 128\ E_0$



Bipolare Spannung u	Spannung am ADC u* = u + $U_{ref}/2$	Dualzahl (z)	**Dual-Offset-Code**	**2er-Komplement**	Dezimal-wert z
$-128\ E_0$	$0\ E_0$	$0000\ 0000_B$	00_H	80_H	-128_D
...					
$-1\ E_0$	$127\ E_0$	$0111\ 1111_B$	$7F_H$	FF_H	-1_D
0	$128\ E_0$	$1000\ 0000_B$	80_H	00_H	0_D
$+1\ E_0$	$129\ E_0$	$1000\ 0001_B$	81_H	01_H	$+1_D$
...					
$+127\ E_0$	$255\ E_0$	$1111\ 1111_B$	FF_H	$7F_H$	$+128_D$

Da Rechner intern in der Regel mit 2er-Komplement-Werten arbeiten, müssen die Werte bei der Ein- und Ausgabe über A/D- und D/A-Umsetzer umgewandelt werden:

Umwandlung 2er-Komplement ⇔ Dual-Offset-Code: MSB invertieren

Wichtige Kenndaten:

- Spannungsbereich: $U_{max} - U_{min}$

  häufig: $U_{max} = U_{ref}$

  bei unipolaren Wandlern in der Regel $U_{min} = 0$

  bei bipolaren Wandlern ist in der Regel $U_{min} \approx -U_{max}$

- Quantisierungsfehler: absolut $\leq E_0 = \frac{U_{max} - U_{min}}{2^n - 1}$ (Schrittspannung)

  relativ $\frac{E_0}{U_{max} - U_{min}} \cdot 100\% = \frac{1}{2^n - 1} \cdot 100\% \approx \frac{1}{2^n} \cdot 100\%$

- Gesamtfehler: Quantisierungsfehler + Fehler der Referenzspannung
  + sonstige Fehler

- Wandlungsdauer: bei D/A-Umsetzern in der Regel vernachlässigbar

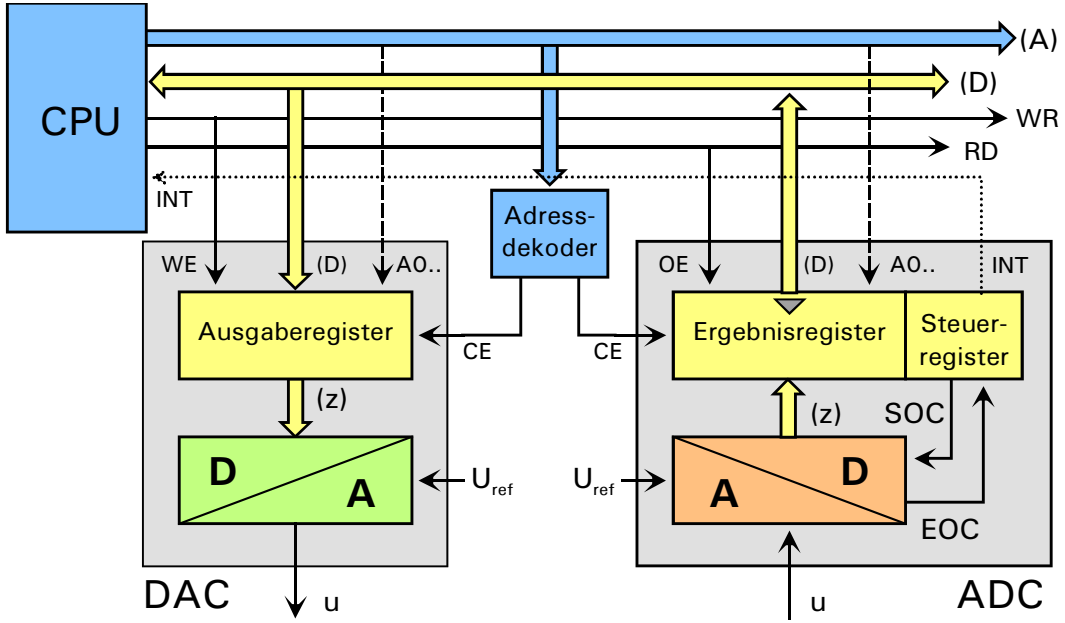
  begrenzt bei A/D-Umsetzern die Abtastfrequenz $f_A < \frac{1}{T_W}$

  die maximale Signalfrequenz f_S die gemessen werden kann,

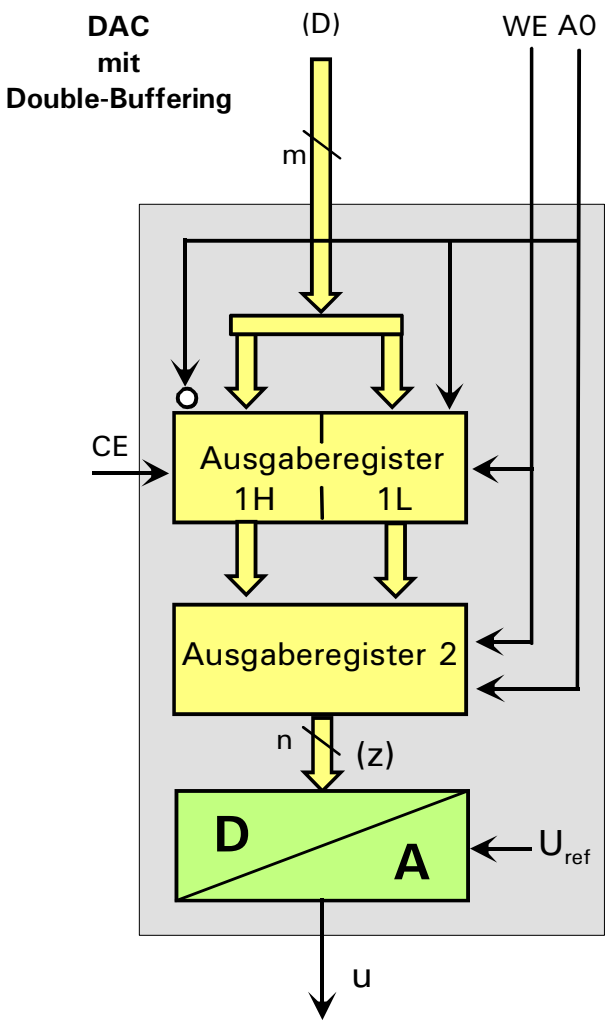
  theoretisch $f_S < f_A / 2$ (Abtasttheorem), praktisch $f_S < f_A / 10$

5.2.2 D/A- und A/D-Umsetzer in Mikrocomputersystemen

Moderne ADC und DAC haben heute ein Bus-Interface, das den direkten Anschluss an den Adress-/Datenbus der CPU mit den üblichen Steuersignalen erlaubt:



- Typ. **Auflösung**en sind **n = 8…12bit** bei einem **Analogsignalbereich** von **0…5V (oder 3,3V)** bzw. **±5V… ±10V**. Bei Umsetzern mit bipolaren Analogsignalen wird meist der **Dual-Offset-Code** verwendet.
- Bei **D/A-Umsetzer**n wird der ins Ausgaberegister geschriebene Wert (z) praktisch innerhalb von $<1\ldots2\mu s$ in das Analogsignal u umgesetzt. Die **CPU wartet nicht und arbeitet** sofort **nach** dem **Schreibvorgang weiter**.
- Bei den oft eingesetzten **A/D-Umsetzern** mit sukzessiver Approximation liegt die **Wandlungszeit im Bereich 5 … 50µs** bei einer Auflösung von 8 … 12bit. Die Wandlung wird durch einen Zugriff auf das Steuerregister ausgelöst (Triggern des Signals **SOC … Start of Conversion**). Nach Ende Wandlung speichert der ADC das Ergebnis im **Ergebnisregister** und kann von der CPU ausgelesen werden. Das Ende der Wandlung erkennt die CPU durch Lesen des Steuerregisters am Signal **EOC … End of Conversion oder** ein dadurch ausgelöstes **Interruptsignal**.



- **Double Buffering**

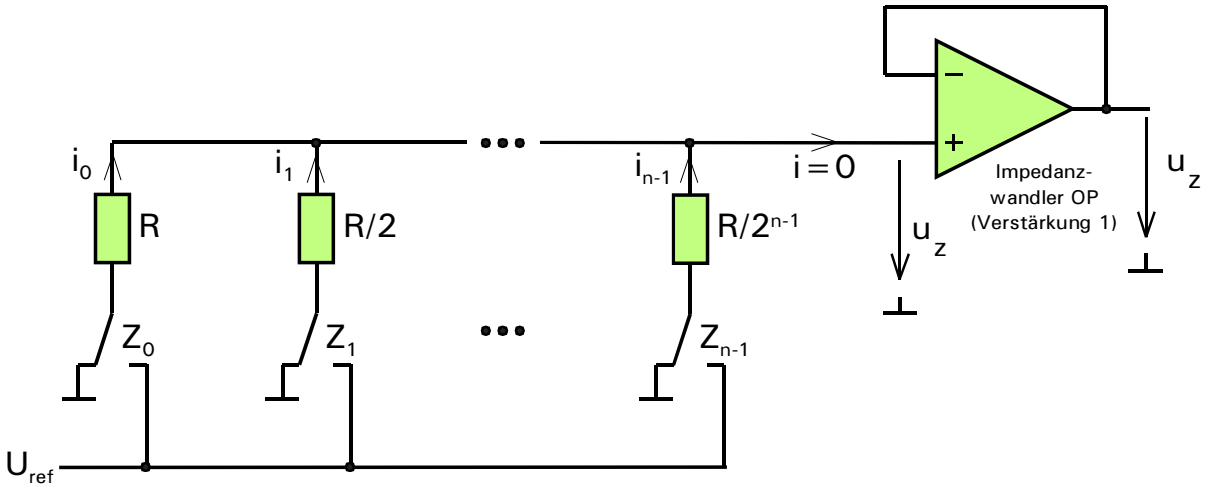
Falls die Datenbusbreite m kleiner ist als die Wortbreite n des DAC (z.B. 12bit DAC an 8bit Datenbus), werden zwei Ausgaberegister hintereinandergeschaltet (Doppelpuffer – **Double Buffering**). Der Digitalwert (z) wird zunächst in zwei (oder mehr) Schreibvorgängen in das direkt an den Datenbus angeschlossene Ausgaberegister 1 geschrieben. Sobald das Datenwort vollständig ausgegeben ist, wird es in einem Schritt an das Ausgaberegister 2 übernommen und liegt damit am D/A-Umsetzer. Auf diese Weise wird vermieden, dass ein falsches Analogsignal ausgegeben wird, solange der neue Ausgabewert noch nicht vollständig ins Ausgaberegister 1 geschrieben wurde.

Dieselbe Problematik ergibt sich, wenn **mehrere Analogsignale** über verschiedene DAC **zeitsynchron ausgegeben** werden müssen, z.B. bei der Ausgabe der Rot-Gelb-Blau-(RGB)-Farbsignale bei der Ansteuerung von Bildschirmen. Dabei werden die Digitalwerte ebenfalls zunächst in das direkt am Datenbus liegende Ausgaberegister der einzelnen DAC geschrieben und dann für alle DAC gleichzeitig in deren zweites Ausgaberegister übernommen (**Simultaneous Update**). Die Steuerung von Double-Buffering und Simultaneous Update ist in vielen DACs integriert und erfolgt automatisch, wobei der Programmierer dann bei der Ausgabe eine bestimmte Reihenfolge bei der Datenausgabe einhalten muss.

- **Serielle Schnittstelle statt paralleles Bus-Interface:** Um kleinere IC-Gehäuse verwenden zu können, werden inzwischen oft serielle synchrone Schnittstellen eingesetzt. Diese Schnittstellen verwenden meist eine Takt-, eine Sende- und eine Empfangsleitung (**3-Draht-Schnittstelle**), z.B. I²C oder SPI.

5.2.3 Prinzipieller Aufbau von Digital-Analog-Umsetzern

- **D/A-Umsetzer mit binär gewichteten Widerständen**



Digitalwert

$(z) = (z_{n-1}, \ldots, z_\nu, \ldots, z_1, z_0)$

Funktion der Schalter:

$z_\nu = 0 \rightarrow$ Schalter an Masse

$z_\nu = 1 \rightarrow$ Schalter an U_{ref}

für $\nu = 0, 1, \ldots, n-1$

Ohmsches Gesetz: $i_\nu = \dfrac{z_\nu \cdot U_{ref} - u_z}{R/2^\nu}$

Knotenregel: $i = \sum_{\nu=0}^{n-1} i_\nu = 0$

Eingangsstrom bei idealem OP:

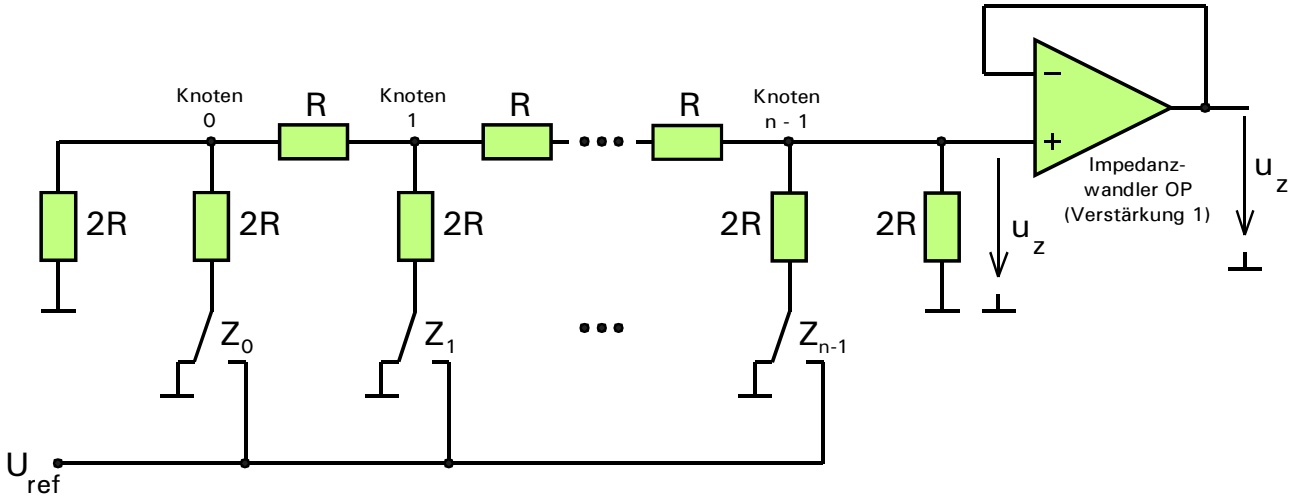
$$i = \sum_{\nu=0}^{n-1} \frac{z_\nu \cdot U_{ref} - u_z}{R/2^\nu} = \frac{U_{ref}}{R} \sum_{\nu=0}^{n-1} z_\nu \cdot 2^\nu - \frac{u_z}{R} \sum_{\nu=0}^{n-1} 2^\nu = 0$$

Daraus folgt mit $\sum_{\nu=0}^{n-1} 2^\nu = 2^n - 1$:

$$\mathbf{u_z} = \frac{U_{ref}}{2^n - 1} \cdot \sum_{\nu=0}^{n-1} z_\nu \, 2^\nu = \mathbf{E_0} \cdot z$$

In IC selten eingesetzt, da sehr großer Widerstandsbereich schlecht integrierbar

- **D/A-Umsetzer mit Kettenleiter (R-2R-Netzwerk)**



Ohne Beweis

(Beweis durch Überlagerungssatz und Maschen- und Knotenregel) :

$$u_z = \frac{U_{ref}}{3 \cdot 2^{n-1}} \cdot \sum_{\nu=0}^{n-1} z_\nu \, 2^\nu$$

$$= \quad E_0 \quad \cdot \quad z$$

nur zwei verschiedene Widerstandswerte, gut integrierbar

Wichtigstes D/A-Umsetzer-Prinzip mit größtem Marktanteil

Wandlungsdauer

Hängt von der Einschwingdauer des OP (und der Umschaltdauer der Analogschalter) ab

Typische Daten

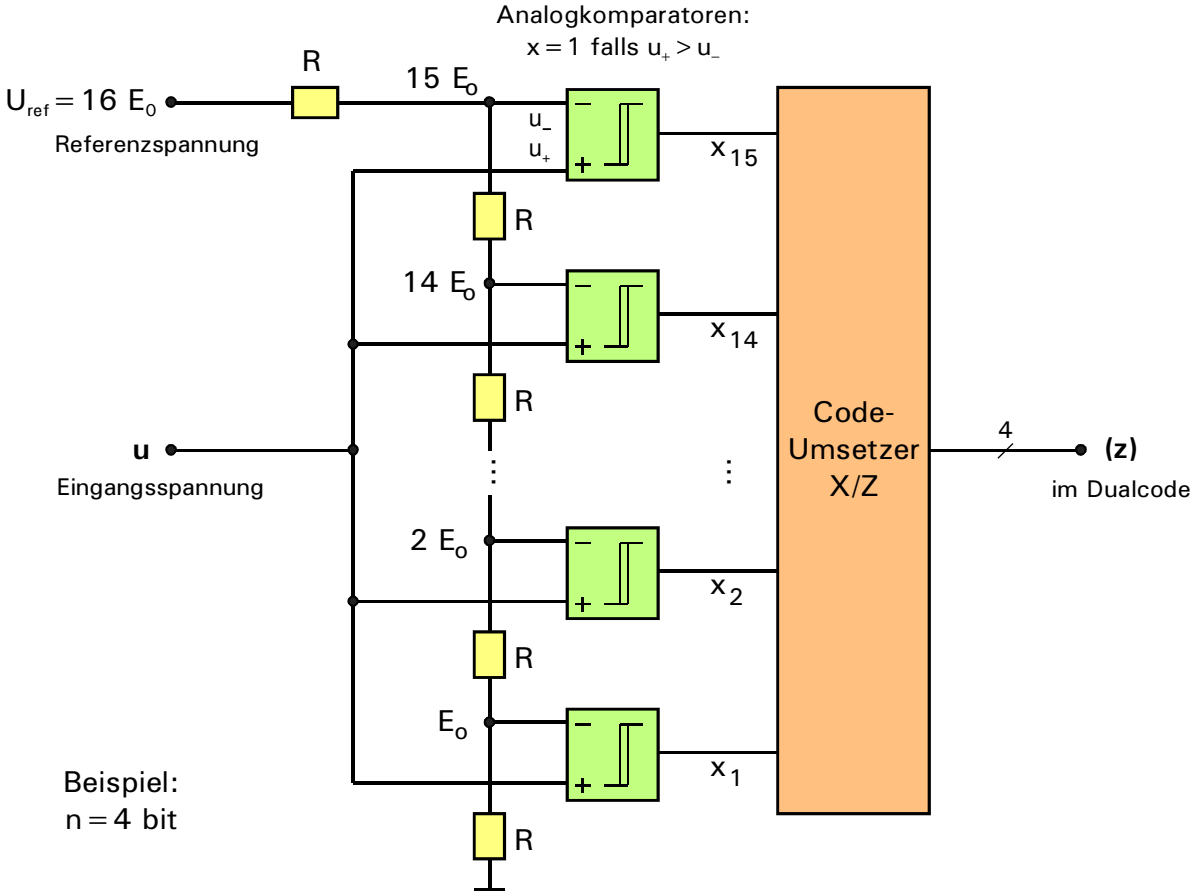
Auflösung: 8...16bit Wandlungsdauer: 0,5 ... 10μs

Multiplizierender D/A-Umsetzer

U_{ref} darf zeitlich veränderlich sein. Der D/A-Umsetzer realisiert dann die Multiplikation eines analogen Signals U_{ref} mit einem digitalen Wert z: $u_z \sim U_{ref} \cdot z$

5.2.4 Prinzipieller Aufbau von Analog-Digital-Umsetzern

- **A/D-Umsetzer mit einem Schritt je Wort: Flash-Umsetzer**



Wandlungsdauer

sehr schnell, Wandlungsdauer = Einschwingzeit eines Komparators (+ Laufzeit Codierer)

Nachteil

sehr großer Aufwand

2^n-1 Analogkomparatoren

Typische Daten

Auflösung: 6 ... 8bit
Wandlungsdauer: 0,1 ... 1μs
auch mehrstufige Ausführungen mit höherer Auflösung möglich

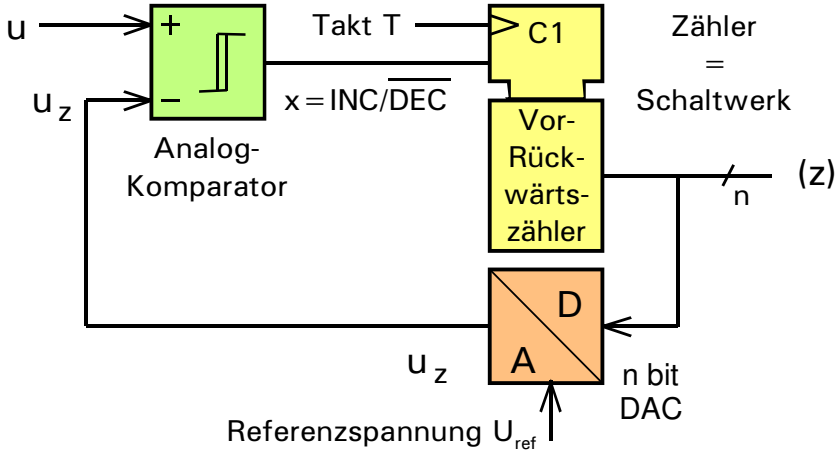
Bsp.: $u = 13{,}5 \cdot E_0$

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Fieberthermometer-Code (x)	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

	3	2	1	0
(z)	1	1	0	1

Dual-Code

- **A/D-Umsetzer mit einem Schritt je Spannungsstufe: Nachlaufumsetzer, Tracking-Umsetzer**



- **Funktionsprinzip**

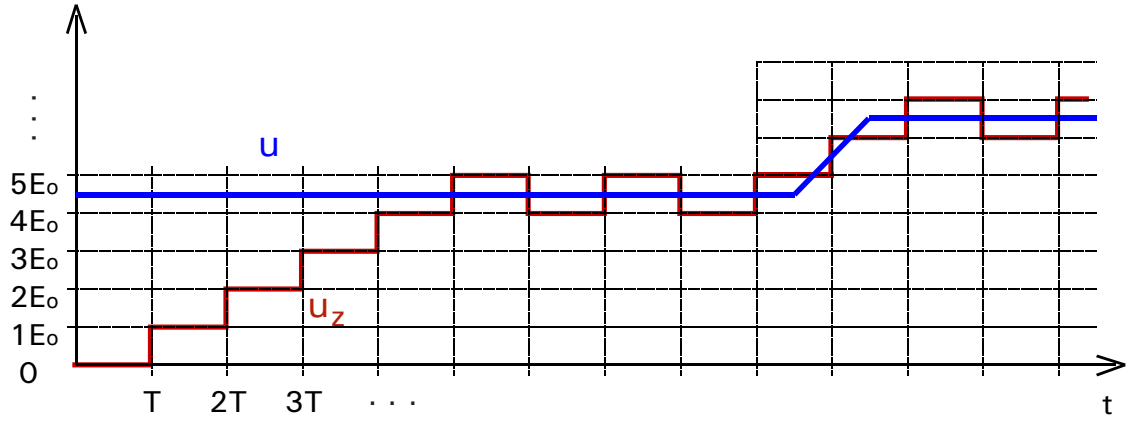
Wenn $u > u_z$: Inkrementiere Zähler um 1

$x = 1 \rightarrow \quad (z)^{(k+1)} = (z)^{(k)} + 1$ max. bis $z_{max} = 2^n - 1$

Wenn $u \leq u_z$: Dekrementiere Zähler um 1

$x = 0 \rightarrow \quad (z)^{(k+1)} = (z)^{(k)} - 1$ min. bis $z_{min} = 0$

- „Stationärer" Zustand: z schwankt um 1bit



Taktperiode T > Einschwingzeit des Komparators und des D/A-Umsetzers (und des Zählers) typ. 500ns…5μs.

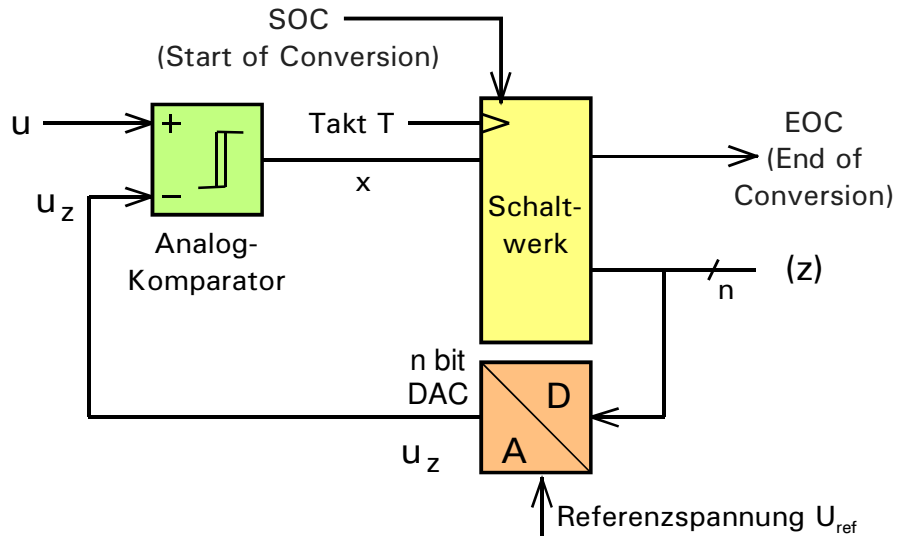
- **Wandlungdauer** (Conversion Time)

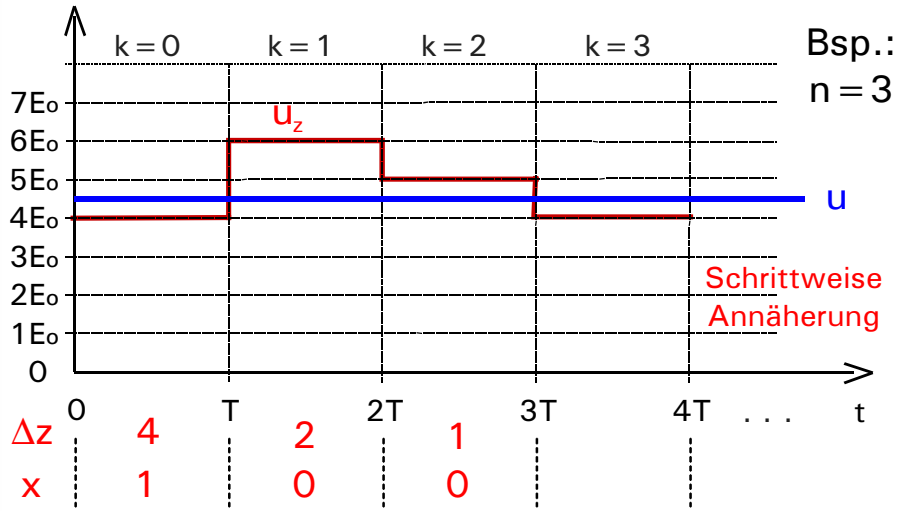
abhängig vom Messwert und vom Startwert des Zählers max. $2^n \cdot T$

Solange sich u nur langsam ändert (um weniger als E_0 je Takt T), ist der Wandler stets eingeschwungen

gut geeignet für stetige Signale, z.B. EKG

- **A/D-Umsetzer mit einem Schritt je Bit: Umsetzer mit sukzessiver Approximation**





- **Funktionsprinzip:**

Anfangszustand k = 0: Start in Bereichsmitte

$$z^{(0)} = \Delta z^{(0)} = 2^{n-1}$$

Schritt k:

Schrittweite halbieren: $\Delta z^{(k)} = \Delta z^{(k-1)}/2$

Wenn $u > u_z$: z um Δz inkrementieren

$$x = 1 \rightarrow \quad (z)^{(k+1)} = (z)^{(k)} + \Delta z^{(k)}$$

Wenn $u \leq u_z$: z um Δz dekrementieren

$$x = 0 \rightarrow \quad (z)^{(k+1)} = (z)^{(k)} - \Delta z^{(k)}$$

Wiederholen bis $\Delta z = 0$ ist, Ende nach n Schritten

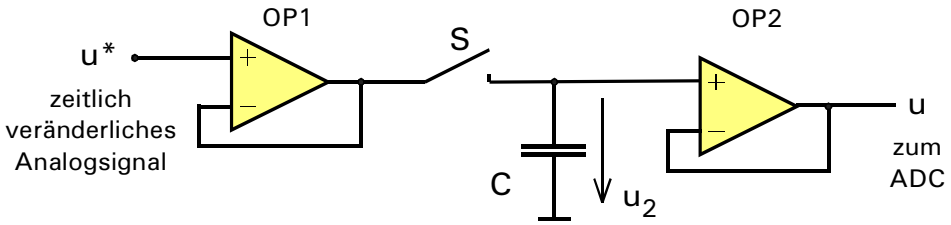
Start der Wandlung durch Signal **SOC**
Ende der Wandlung durch Signal **EOC** angezeigt

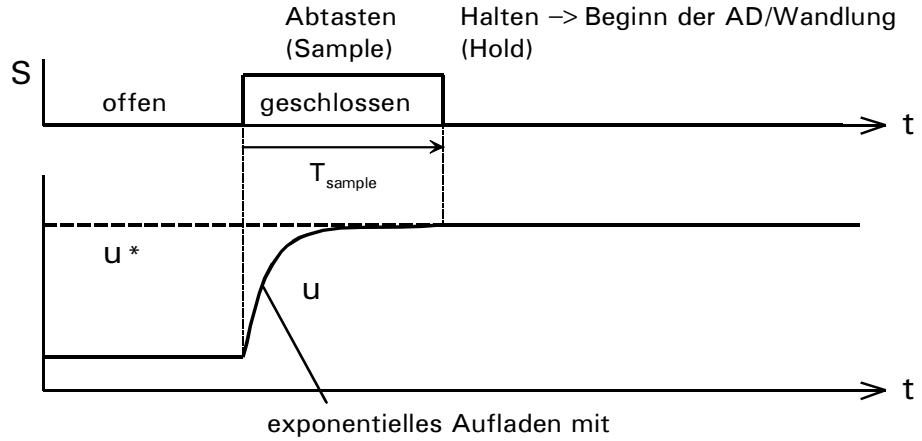
- **Wandlungsdauer** n · T (unabhängig vom Messwert)

Taktperiode T > Einschwingzeit des Komparators und des D/A-Umsetzers (und des Schaltwerks), wie beim Nachlaufwandler

- **Typ. Daten**: 8bit / 1μs ... 16bit / 200μs
- **Am weitesten verbreitetes ADC-Prinzip**

- **Voraussetzung für fehlerfreie Funktion des A/D-Umsetzers mit sukzessiver Approximation:** Eingangsspannung u_x muss während der Wandlung konstant sein. Abhilfe bei zeitlich veränderlichem Eingangssignal u_x: **Abtast-Halteglied (Sample and Hold)**.





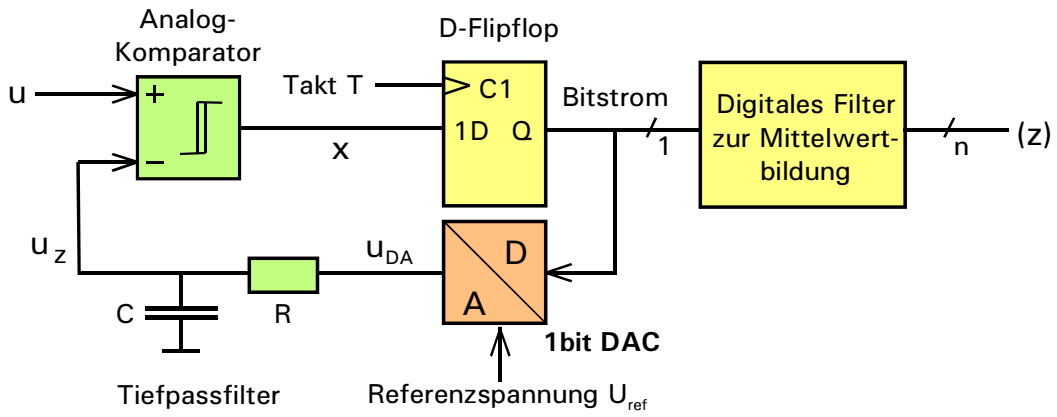
Die Aufladezeit des Kondensators ($\sim \tau$) addiert sich zur Wandlungsdauer.

$T_{sample} = 3...4\ \tau$, typ. $1\mu s$

In der Regel wird das Abtast-Halteglied zusammen mit dem A/D-Umsetzer integriert.

- **Messung mehrerer Analogsignale**
  Da A/D-Umsetzer immer noch relativ aufwendig sind, setzt man statt mehrerer A/D-Umsetzer häufig einen **Analog-Multiplexer** (Umschalter) vor dem ADC ein, wenn mehrere Analogsignale gemessen werden sollen. Die Signale können damit allerdings nur zeitlich hintereinander gemessen werden.

- **Sigma-Delta-Umsetzer**



Wenn $u > u_z$: $x = 1 \longrightarrow u_{DA} = U$ (mit $U > u$) $\longrightarrow u_z$ wird größer

Wenn $u_z > u$: $x = 0 \longrightarrow u_{DA} = 0$ (mit $0 < u$) $\longrightarrow u_z$ wird kleiner

Im eingeschwungenen Zustand schwankt u_z um u. Dabei ist x dann ein Rechtecksignal (*Bitstrom*), dessen Tastverhältnis (relative Dauer der 1 und 0 Zustände) so ist, dass im zeitlichen Mittel gilt $U_z \approx U_{DA} \approx u$

$$U_{DA} = \frac{1}{T} \int_{\tau=0}^{t} u_{DA}(\tau)\, d\tau = U \frac{1}{T} \int_{\tau=0}^{t} x(\tau)\, d\tau = U \cdot X \approx u \longrightarrow X \sim \frac{u}{U}$$

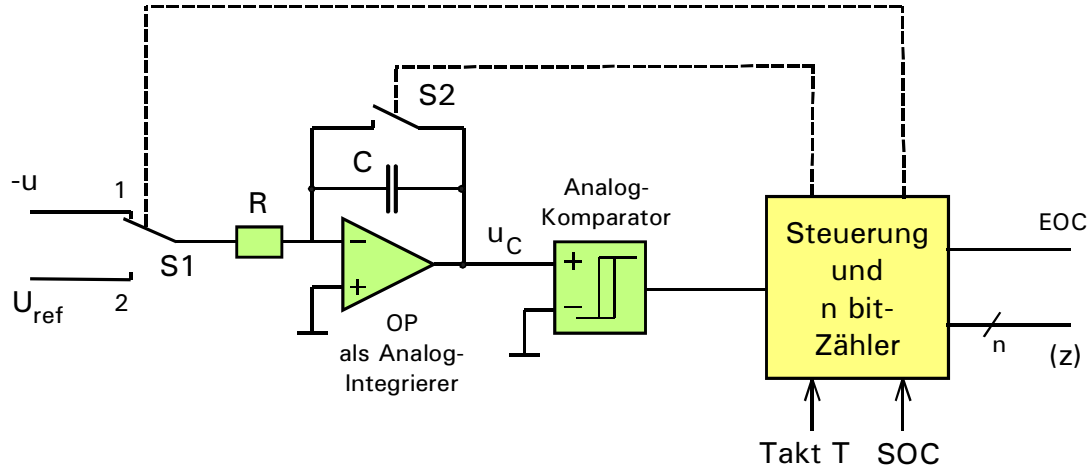
- **Wandlungsdauer**
  Taktperiode T > Einschwingzeit des Komparators und des 1bit-D/A-Umsetzers (und des Zählers) < 100ns

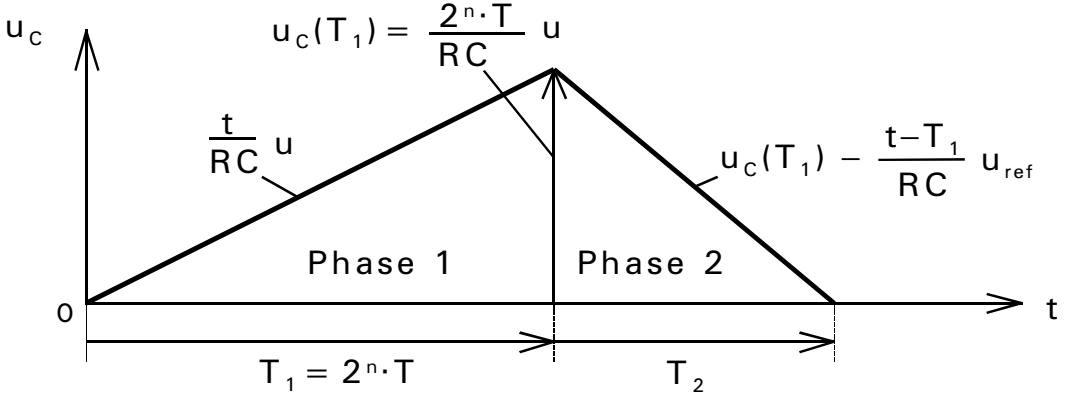
  Für Mittelwertbildung ca. $2^n \cdot T$ (Messverzögerung), verhält sich dynamisch ähnlich wie der Nachlaufwandler → mäßig schnell, hohe Auflösung
- **Typ. Daten**: Auflösung 18 ... 24bit, Wandlungsdauer >100μs
- Häufig eingesetzt in der Audio-Signalverarbeitung.
- Gut integrierbar, da sehr einfacher Analogteil.

Auf der Analogseite wird der arithmetische Mittelwert U_{DA} näherungsweise durch den RC-Tiefpass gebildet (falls RC >> T ist). Auf der Digitalseite wird der arithmetische Mittelwert X durch Aufsummieren der 0 und 1 Impulse im Bitstrom x über einen festen Zeitraum berechnet:

$$X = \frac{1}{T} \int_{\tau=0}^{t} x(\tau)\, d\tau \approx \frac{1}{n} \sum_{k=0}^{n} x(k)$$

- **Dual-Slope-A/D-Umsetzer**





Wandlung in zwei Phasen („Dual Slope")

Anfangszustand: $u_C(0) = 0$, Zählerstand $z = 0$

Phase 1:

S_1 in Stellung 1 → Spannung –u integrieren

u_C steigt zeitlinear an: $u_c(t) = \frac{t}{RC} \cdot u$

Ende der Phase 1, wenn der Zähler den Wert 2^n erreicht hat, d.h. zur Zeit $T_1 = 2^n \cdot T$

Phase 2:

S_1 in Stellung 2 → Spannung U_{ref} integrieren

u_C sinkt zeitlinear: $u_c(t) = u_c(T_1) - \frac{t-T_1}{RC} \cdot U_{ref}$

Ende der Phase 2, wenn wieder $u_C(T_1 + T_2) = 0$ ist.

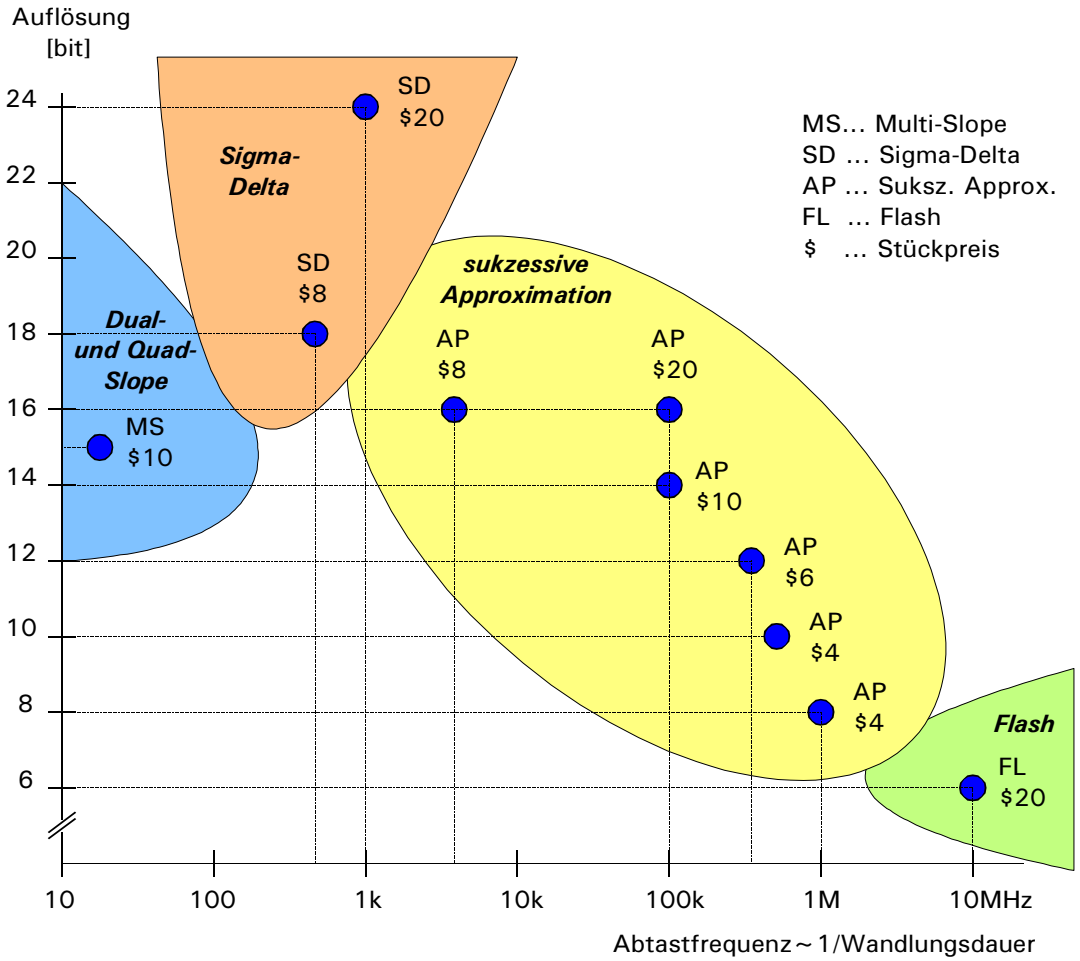
Zählerstand (Ergebnis) am Ende der Phase 2:

$$\mathbf{z = \frac{u}{U_{ref}} \cdot 2^n}$$ unabhängig von RC und T !

- **Wandlungsdauer:** langsam, bestimmt durch analoge Integration, typ. 10...100ms, Auflösung nur durch Auflösung im Analogteil begrenzt, typ. >16 ... 20bit. Einsatz in der Regel in Messgeräten (Digitalvoltmeter).
- Störsignale, z.B. Signalrauschen oder Netzbrummen, werden durch die Analogintegration teilweise ausgefiltert.

- **Stand der Technik bei Analog-Digital-Umsetzern**

Daten ausgewählter Produkte



Typische Genauigkeit = Auflösung – 1 ... 2bit

Aktuelle Trends

- Unterschiedliche Mikroprozessor-Busschnittstellen (8 oder 16bit Parallelbusse z.T. Multiplex, diverse serielle Busse, z.B. I²C, SPI)
- Integration von Zusatzbaugruppen, z.B. Sample & Hold, Referenzspannungsquelle, Multiplexer für Mehrkanalbetrieb
- Reduzierte Leistungsaufnahme für Mobilanwendungen

Wichtige ADC und DAC Hersteller

- Analog Devices (www.analog.com)
- Maxim (www.maxim-ic.com)

5.2 Analog-Digital- und Digital-Analog-Umsetzer

- **Kenngrößen und Fehler bei A/D- und D/A-Umsetzern**

Auflösung
Anzahl der (Bit-)Stellen $n = \log_2$(Anzahl der Quantisierungsstufen). Gelegentlich auch

$$E_0 = \frac{U_{max} - U_{min}}{2^n - 1},$$

d.h. Spannungsschritt, der dem niederwertigsten Bit (LSB) zugeordnet ist.

Quantisierungsfehler 1LSB bzw. $\pm 0{,}5$LSB bzw. $\pm E_0/2$
Systembedingte Abweichung zwischen dem tatsächlichen und dem quantisierten Wert. Bei Sprach- oder Musikübertragung verursacht die Quantisierung das sogenannte 'Quantisierungsrauschen' (Signal/Rauschabstand ca. 6dB/bit).

Quantisierungskennlinie
Zusammenhang zwischen dem wertkontinuierlichen und dem wertdiskreten Signal. In der Regel soll dieser Zusammenhang (quasi-) linear (d.h. Treppenfunktion mit konstanter Stufenhöhe) sein. Ausnahme: Nichtlineare Quantisierung bei PCM-Telefonsystemen mit logarithmischer Kennlinie zur Übertragung kleiner (leiser) Signale mit besserem Signal-Rauschabstand.

Genauigkeit
Abweichung des gemessenen Werts vom wirklichen Messwert. Abweichungen entstehen (unvermeidbar) durch den Quantisierungsfehler sowie (vermeidbar) durch Verstärkungs-, Offset-, Nichtlinearitäts-, Monotonie- bzw. Missing-Code-Fehler.

Offsetfehler
Abweichung des Ausgangssignals vom theoretischen Wert beim Eingangssignal 0.

Verstärkungsfehler
Abweichung des Ausgangssignals vom theoretischen Wert beim maximalen Eingangssignal.

Nichtlinearität
Abweichung zwischen der idealen und der realen Quantisierungskennlinie

Monotonie
Eigenschaft eines D/A-Umsetzers, dass die Ausgangsspannung bei schrittweiser Erhöhung des digitalen Eingangswortes um 1bit stetig ansteiget. D/A-Umsetzer mit großen Linearitätsfehlern sind häufig auch nicht monoton.

Missing-Code-Fehler
Am Ausgang eines A/D-Umsetzers nicht auftretende Codewörter, obwohl die Eingangsspannung den entsprechenden Wert aufweist, z.B. durch nicht-monotone D/A-Umsetzer verursacht.

Einschwingdauer (Settling Time) bei D/A-Umsetzern
Zeit vom Anlegen des digitalen Eingangswortes bis das Ausgangssignal auf $\pm E_0/2$ eingeschwungen ist; liegt (außer bei 1bit-Umsetzern) typ. im Bereich 1 ... 10 μs.

Wandlungsdauer (Conversion Time) bei A/D-Umsetzern
Zeit vom Beginn einer Wandlung bis zum Vorliegen des digitalen Ausgangswortes, gegebenenfalls inklusive Einschwing-

zeit des Abtast-Halteglieds; umso größer, je höher die Auflösung ist. Typ. für sukzessive Approximation 1 ... 50μs bei 8 ... 12bit, Wandlungsrate $\leq$ 1/Wandlungsdauer [Einheit Samples/sec]

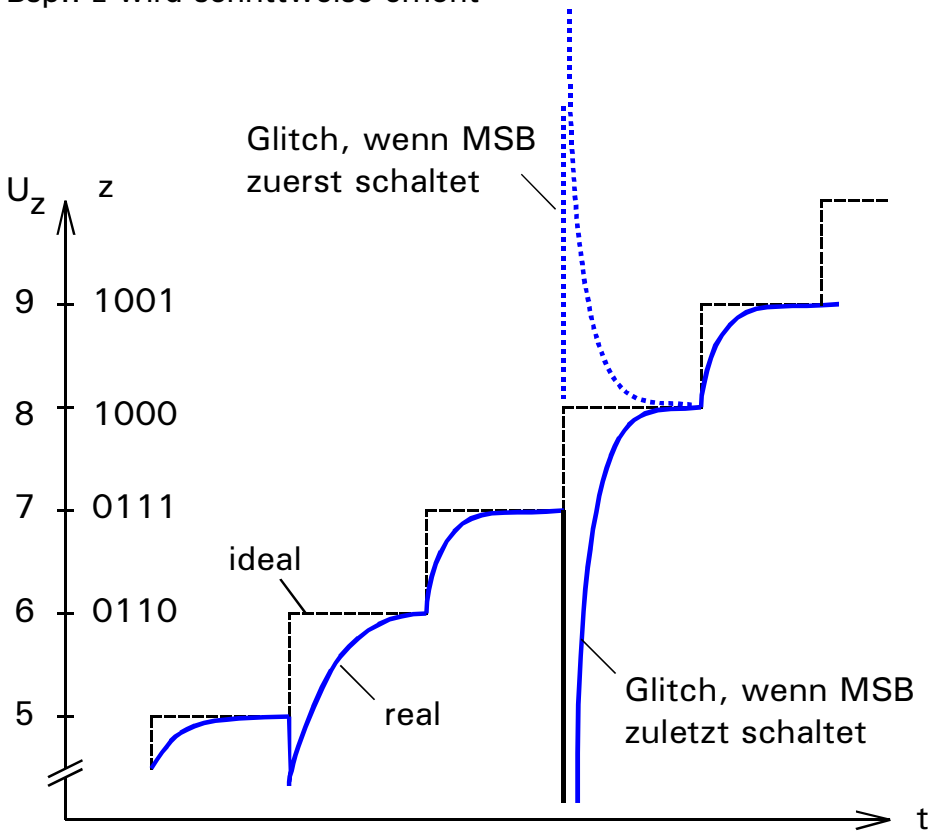
Abtastzeit = 1/Abtastfrequenz (nicht verwechseln mit Abtastdauer!): Zeitabstand zwischen periodischen Abtastungen eines Signals.

Abtastdauer (nicht verwechseln mit Abtastzeit!)
Zeitdauer, während der das Abtast-Halteglied (Sample & Hold) das Eingangssignal abtastet. Innerhalb dieser Zeit muss der Ausgang auf $\pm$0.5LSB eingeschwungen sein. Begrenzt die Größe des Speicherkondensators nach oben.

Spannungsdrift (Droop Rate)
Entladegeschwindigkeit des Speicherkondensators eines Abtast-Halteglieds durch Leckströme während der Haltephase, d.h. während der Wandlungsdauer; sollte $< \pm$0.5LSB sein. Begrenzt die Größe des Speicherkondensators nach unten.

Glitch-Effekt bei D/A-Umsetzern
Wenn sich der Digitalwert (z) ändert, schalten die n Schalter z_ν in der Praxis nicht exakt gleichzeitig. Es kommt also zu ungültigen Zwischenzuständen, die auch in der analogen Spannung als Spannungsspitze oder Spannungseinbruch (engl.: Glitch) zu sehen sind. Der Effekte ist umso größer, je mehr Schalter gleichzeitig umschalten müssen. Der größte Glitch entsteht daher bei den Übergängen (z) = 011...11$_B$ nach 100...00$_B$ und umgekehrt, d.h. wenn das MSB schaltet.

Bsp.: z wird schrittweise erhöht

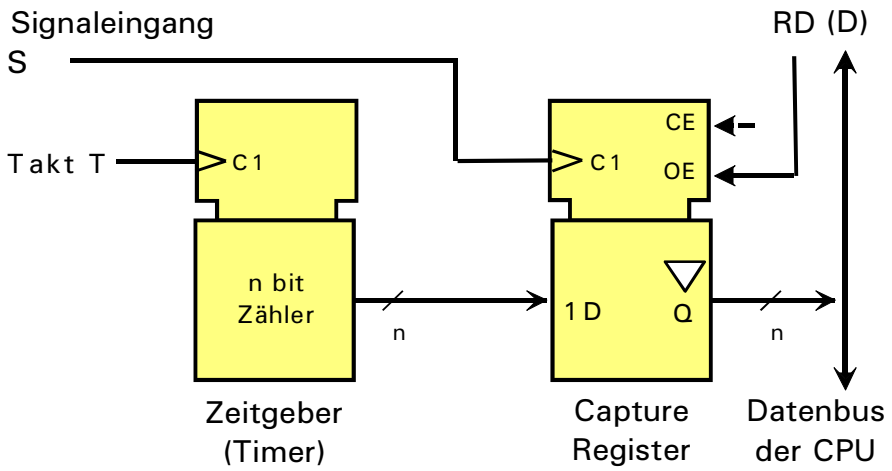


Glitch ist ein nichtlinearer Effekt, er kann durch ein Tiefpassfilter **nicht** beseitigt werden. Abhilfe: 'Deglitching' mit einem Track- und Hold-Glied (Sample und Hold-Glied, das beim Umschalten kurz gesperrt wird.)

5.3 Impulssignal-Ein- und Ausgänge

Bei der Steuerung von Geräten und Anlagen müssen häufig Impulssignale erzeugt oder gemessen werden oder Impulssignale mit vorgegebener Periodendauer und Tastverhältnis generiert werden. Dazu setzt man **Capture-Compare-** und **PWM-Einheiten** ein. Im Kern bestehen diese Baugruppen aus einem Zähler mit festem Takt als Zeitgeber (Timer) sowie weiteren Zählern, Registern und Vergleichern.

Beispiel Capture-Eingang: Messen des Zeitpunkts einer Signalflanke



Typische Anwendung:

Messung des Abstands von Drehzahlimpulsen

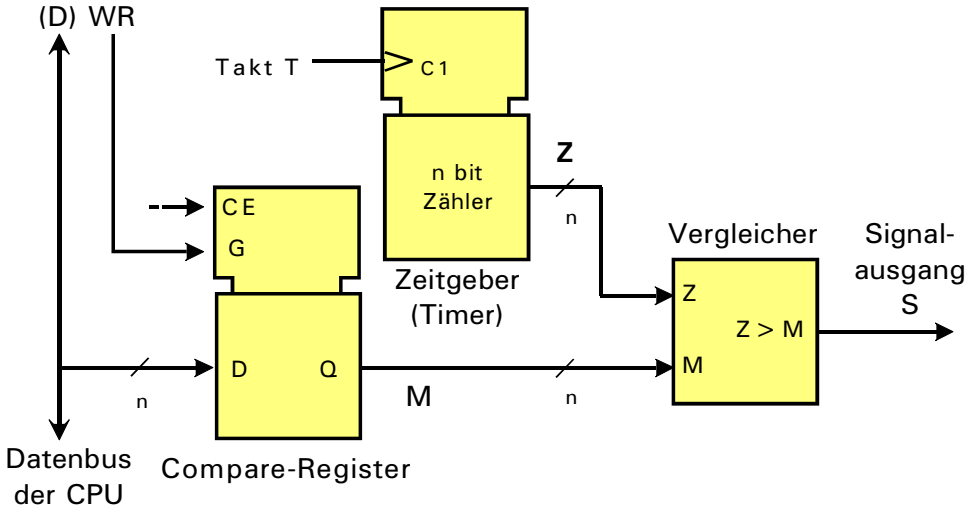
Der Zeitgeber wird beim Einschalten auf einen definierten Anfangswert gesetzt und zählt danach mit konstanter Taktfrequenz.

Bei einer 0 → 1 Flanke am Signaleingang S wird der aktuelle Zählerstand im Capture-Register gespeichert und kann von der CPU gelesen werden.

Üblicherweise sind mehrere Capture-Register vorhanden und die Triggerung des Capture-Registers kann auf positive, negative oder beide Signalflanken eingestellt werden.

Durch Differenzbildung zwischen zwei Messwerten lassen sich Zeitdifferenzen messen.

Beispiel Compare-Ausgang: Zeitlich präzises Erzeugen einer Signalflanke



In das Compare-Register wird von der CPU der gewünschte Zeitpunkt M (Zählerstand) der Signalflanke geschrieben (mit M > Z).

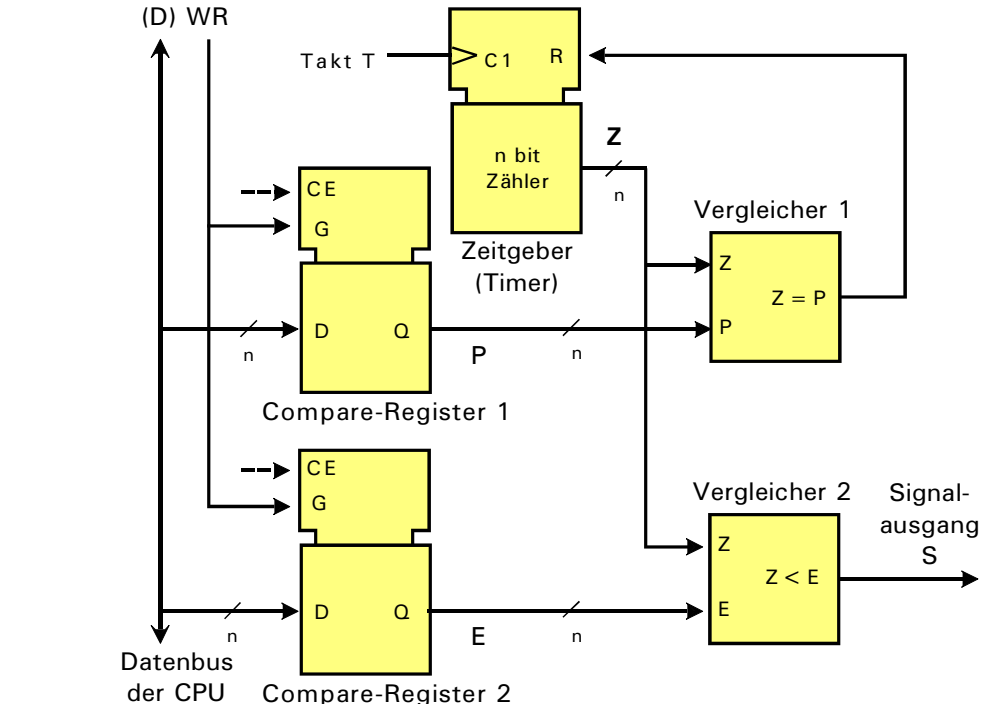
Solange der aktuelle Zählerstand Z < M ist, ist der Vergleicherausgang 0 und damit das Ausgangssignal S = 0.

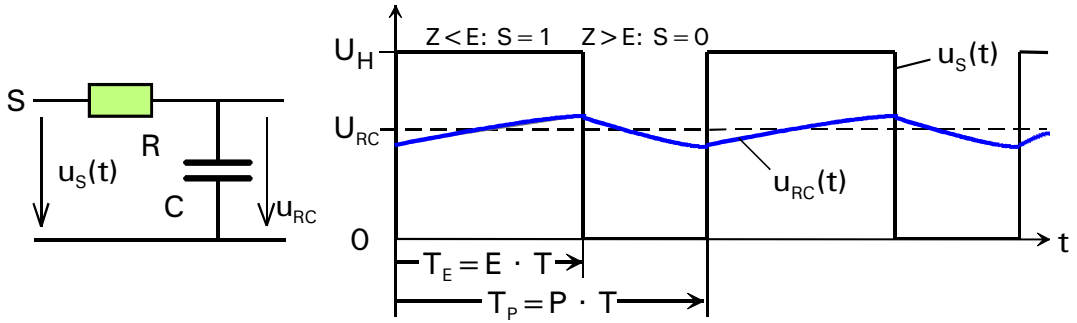
Sobald der Zählerstand Z > M ist, wird der Vergleicherausgang 1 und die gewünschte Signalflanke S = 0 → 1 wird erzeugt.

Üblicherweise sind mehrere Compare-Register und Vergleicher vorhanden, so dass sich sowohl 0 → 1 als auch 1 → 0 Flanken erzeugen und damit komplexe Impulsfolgen generieren lassen.

Typische Anwendung: Erzeugen des Zündsignals für eine Zündkerze

Beispiel Pulsbreiten-moduliertes Impulssignal (PWM Pulse Width Modulation)





1bit DAC, gut integrierbar, Einsatz z.B. in CD-Playern

Die CPU schreibt in die zwei Compare-Register die Periodendauer $T_P = P \cdot T$ und die Einschaltdauer $T_E = E \cdot T$ (bzw. die zugehörigen Zählerstände P und E).

Der Zähler zählt ab Z = 0 an aufwärts. Solange der Zählerstand Z < E ist, liefert der Vergleicher 2 das Signal S = 1.

Wenn der Zählerstand Z > E ist, wird das Signal S = 0, bis beim Stand Z = P der Vergleicher 1 den Zähler auf Z = 0 zurücksetzt. Der Vorgang wiederholt sich periodisch.

Schaltet man hinter den Signalausgang ein RC-Glied mit RC >> P·T, wird aus dem Rechtecksignal $u_S(t)$ eine (annähernd konstante) Analogspannung

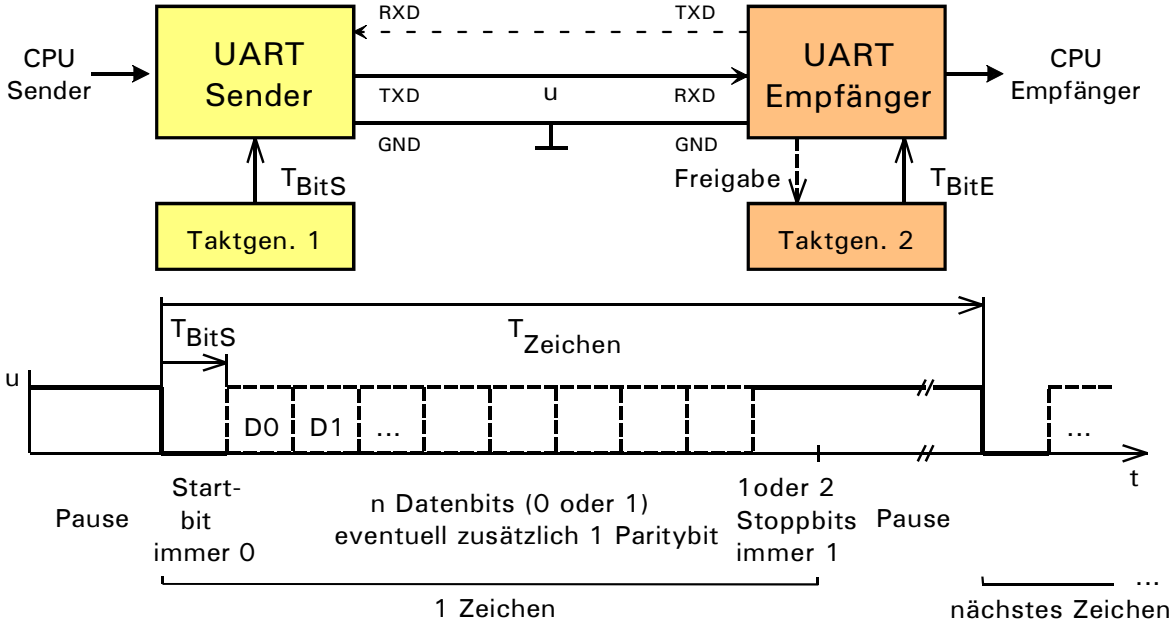
$$U_{RC} \approx \frac{1}{P \cdot T} \int_{t=0}^{P \cdot T} u_S(t)\, dt = U_H \cdot \frac{E}{P}$$

die über E verändert werden kann.

5.4 Serielle Datenübertragung

Die Datenübertragung zwischen Rechnern oder Rechnern und Peripheriegeräten erfolgt bei größeren Entfernungen seriell, um die Anzahl der Übertragungsleitungen möglichst klein zu halten:

- **Asynchrone Datenübertragung**: Start-Stop-Verfahren mit UART



Getrennte (asynchrone) Taktgeneratoren in Sender und Empfänger mit annähernd, aber nicht exakt gleichem Takt $T_{Bit\,S} \approx T_{Bit\,E}$

Leitung im Ruhezustand auf H

Übertragungsbeginn durch L Bit markiert → Freigabe (Start) des Taktgenerators im Empfänger

Übertragung der einzelnen Bits eines Zeichens beginnend beim LSB im Bittakt $T_{Bit\,S}$ des Senders, Abtastung durch den Empfänger im Bittakt $T_{Bit\,E}$

Ende der Übertragung eines Zeichens nach n Datenbit → Stopp des Taktgenerators im Empfänger

Mindestens 1bit Pause (Stoppbit) bis zum Beginn des nächsten Zeichens

Übertragung einzelner ASCII-Zeichen

Anzahl Datenbits n = 5...10, (typ. 8), 1 Startbit, m = 1...2 Stoppbits (typ. 1)

Bitrate $\frac{1}{T_{Bit}}$ *[Bit/s]* Zeichenrate $\frac{1}{T_{Zeichen}} \le \frac{1}{(n+m+1)T_{Bit}}$ *[Zeichen/s]*

5.4 Serielle Datenübertragung

Typische Werte: 8 N 1 = 8 Datenbit, keine (No) Parität, 1 Stoppbit (und 1 Startbit)

Bittakt: 9600, 19200, 38400, 57600, 115200 bit/sec Zeichenrate ≤ 1/10 Bitrate

Gelegentlich wird ein Paritätsbit zur Erkennung einfacher Übertragungsfehler verwendet.

Universal Asynchronous Receiver and Transmitter UART

Üblicherweise sind auf beiden Seiten je eine Sende- und eine Empfangseinheit vorhanden, die zusammen mit der zugehörigen Steuerlogik als **UART** bezeichnet werden. Senden und Empfangen ist in der Regel gleichzeitig möglich (Voll-Duplex-Betrieb).

Die Parallel-Seriell-Umwandlung erfolgt über Schieberegister (Sende- bzw. Empfangsregister), die von der jeweiligen CPU über den Datenbus parallel geschrieben bzw. gelesen werden. Das Senden beginnt automatisch, sobald die CPU ein Zeichen in das Senderegister geschrieben hat. Vorher muss die CPU überprüfen, ob das vorige Zeichen schon vollständig gesendet wurde („Senderegister frei?“). Dazu überprüft die CPU ein Statusbit in einem Steuerregister des UARTs. Den Empfang eines neuen Zeichens kann die CPU durch Prüfen eines weiteren Statusbits des UART erkennen (Polling) oder die CPU erhält ein spezielles Rückmeldesignal vom UART (Empfangs-Interrupt).

Die Steuerung des Sende- und Empfangsvorgangs sowie die Erzeugung des Sende- bzw. Empfangstaktes aus dem Taktsignal der CPU (Bittaktgenerator, anderer Name Baudratengenerator) erfolgt durch ein kleines Schaltwerk. Vor Beginn der Datenübertragung müssen Sender und Empfänger die richtigen Werte für den Bittakt und die Anzahl der Daten- und Stopbits durch Schreiben in die Steuerregister festlegen.

Minimal sind 3 Verbindungsleitungen notwendig, wobei Sendeausgang (Transmit TXD) und Empfangseingang (Receive RXD) der beiden UARTs über Kreuz verbunden werden, zusätzlich ist eine Masseleitung (Ground GND) erforderlich.

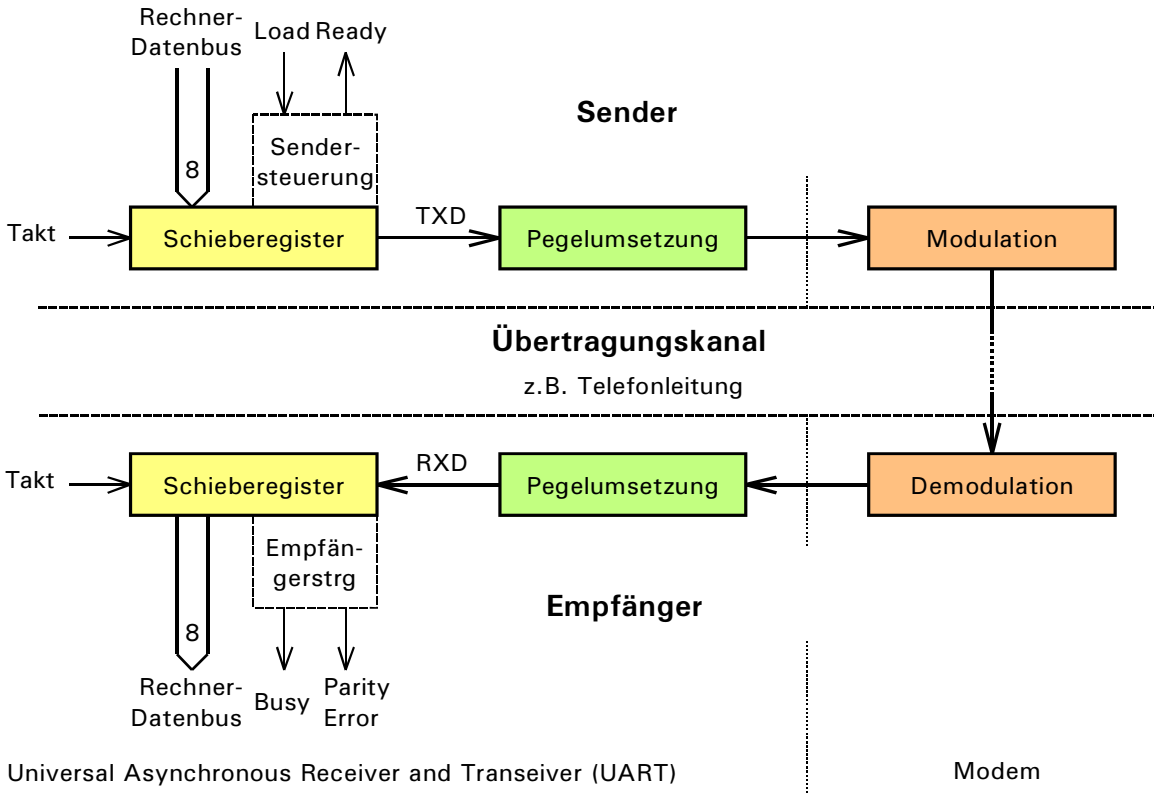
Bei längeren Leitungen überträgt man die Signale auf den Leitungen üblicherweise nicht mit den Logikpegeln des UARTs, sondern verwendet eine Pegelumsetzung mit **negativer Logik (als V24 bzw. RS232C genormt)**:

Sender (Ausgang): '0' ⇔ U ≥ 12V '1' ⇔ U ≤ –12V Empfänger (Eingang): '0' ⇔ U ≥ 3V '1' ⇔ U ≤ – 3V

5.4 Serielle Datenübertragung

Sendet der Sender ein neues Zeichen, bevor die CPU des Empfängers das vorige Zeichen aus dem Empfangsregister ausgelesen hat, wird das vorige Zeichen überschrieben und geht verloren. Die meisten UARTs haben nur einen Pufferspeicher für max. 2 Zeichen. Um einen Datenverlust zu verhindern, können zusätzliche Handshake-Signale (=zusätzliche Leitungen) zur Synchronisation vorgesehen werden. Die wichtigsten dieser Handshake-Signale sind:

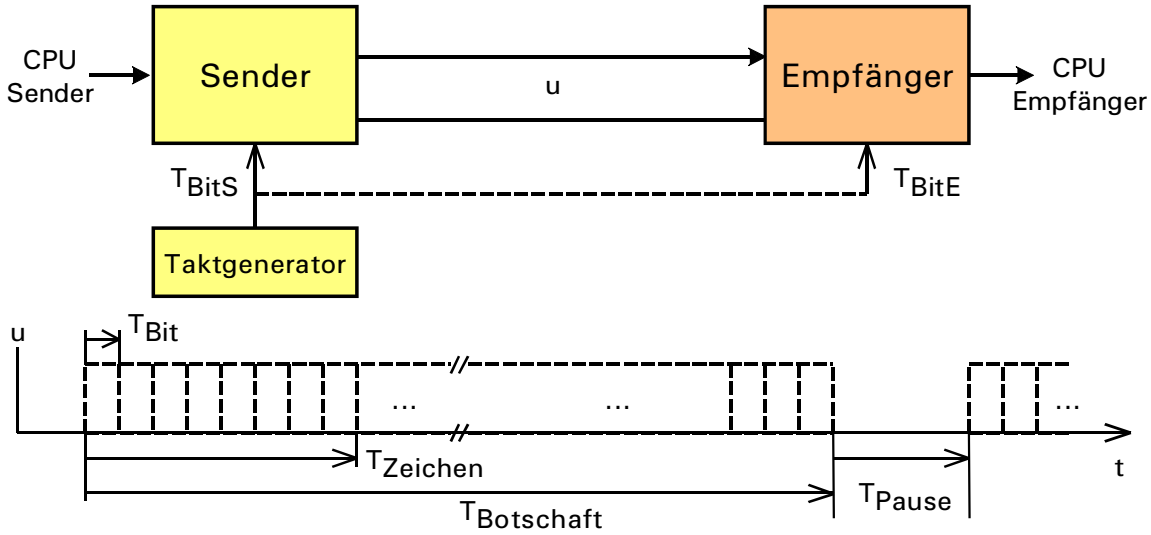
Sendeanforderung:	Request to Send (Sender)	→ verbunden mit → Clear to Send (Empfänger)
Anzeige Empfangsbereitschaft:	Data Terminal Ready (Empfänger)	→ verbunden mit → Data Set Ready (Sender)



Eine längere Übertragungspause oder längere Folgen von 0 oder 1 Bits auf der Leitung wirken wie eine Gleichspannung, die über das Telefonsystem nicht übertragen werden kann (untere Frequenzgrenze ca. 300Hz). Umgekehrt können und dürfen die Oberschwingungen, die sich bei den Rechteckimpulsen des UART ergeben, nicht über die Telefonleitung übertragen werden (obere Frequenzgrenze 3400Hz [*1]). Daher wird das UART-Signal mit einem Modulationsverfahren (Kombination von Amplituden- und Phasenmodulation) moduliert übertragen. Zugehörige Baugruppe: Modem = Modulator + Demodulator.

[*1] Hinweis:
Bei DSL-Leitungen können zwischen Teilnehmer und Vermittlungsstelle auch höhere Frequenzen übertragen werden, wenn die Leitung maximal einige Kilometer lang ist. Eine direkte Übertragung dieser Frequenzen zwischen zwei Teilnehmern ist aber nicht möglich.

- **Synchrone Datenübertragung**: System Peripheral Interface SPI



Da Sender und Empfänger garantiert taktsynchron arbeiten, sind wesentlich längere Botschaften (bei Ethernet z.B. 1500 Byte) und wesentlich höhere Bitraten (> 100 Mbit/s) möglich als bei der asynchronen Datenübertragung.

Damit die hohen Bitraten störsicher übertragen werden, arbeitet man bei größeren Entfernungen mit Differenzsignalen und kleinen Signalpegeln (einige hundert Millivolt) statt massebezogenen Signalen.

Je Übertragungsrichtung sind bei Differenzsignalen zwei Leitungen notwendig. Soll gleichzeitig gesendet und empfangen werden, also vier Leitungen (z.B. Voll-Duplex-Ethernet).

Sender und Empfänger arbeiten mit dem exakt gleichen Bittakt $T_{Bit\ S} = T_{Bit\ E}$. Der Takt wird über eine separate Leitung oder durch ein geeignetes Codierungsverfahren mit übertragen.

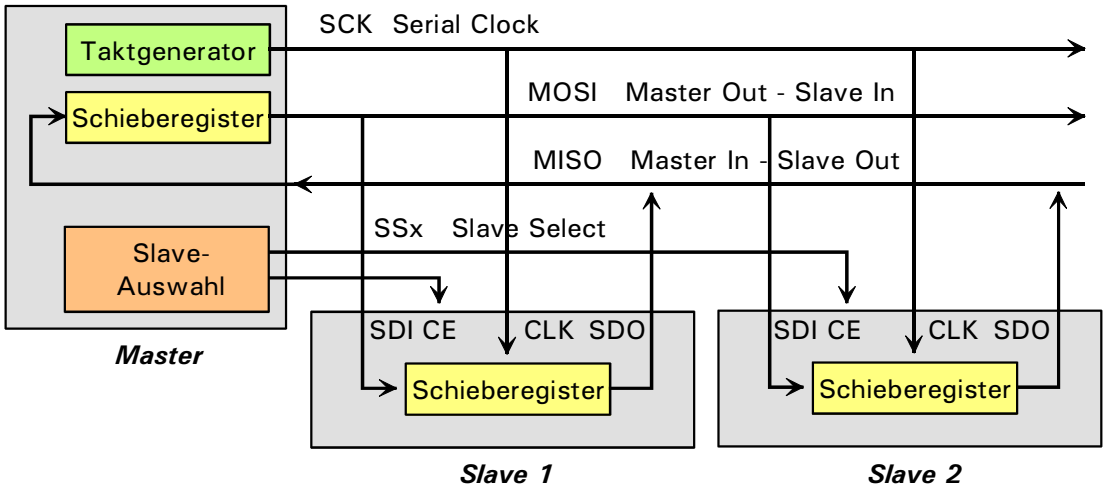
Keine Pause zwischen den Zeichen einer Botschaft (Block), statt 8bit-Zeichen können beliebige Bitfolgen übertragen werden.

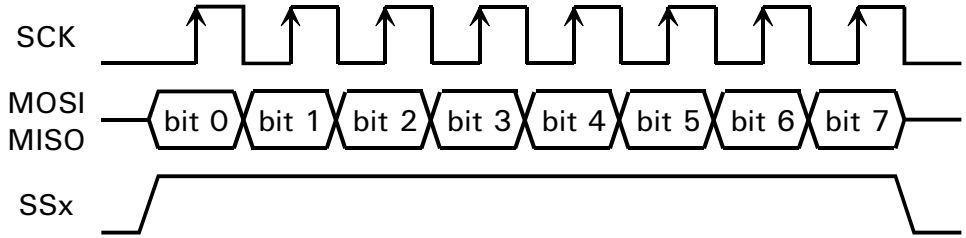
Kennzeichnung des Blockanfangs und -endes durch Übertragung von besonderen Synchronisationszeichens, z.B. ASCII-Zeichen SYN = 16_H

Zur Übertragungssteuerung werden weitere Information und zur Datensicherung zusätzliche Prüfsummen (CRC) übertragen (Botschafts-Header und -Trailer).

5.4 Serielle Datenübertragung

Weit verbreitet ist der **System Peripheral Interface (SPI)** Bus, mit dem Mikrocontroller externe Peripheriebausteine, z.B. DAC, über ein synchrone serielle Verbindung ansprechen können:





Die Länge der Schieberegister und damit die Anzahl der in jeder Richtung übertragenen Bits ist 8. Bit-Reihenfolge (MSB oder LSB zuerst) und aktive Taktflanke (Datenübernahme mit der positiven oder der negativen Taktflanke) sind programmierbar. Der Bittakt ist beliebig bis zu mehreren Mbit/s.

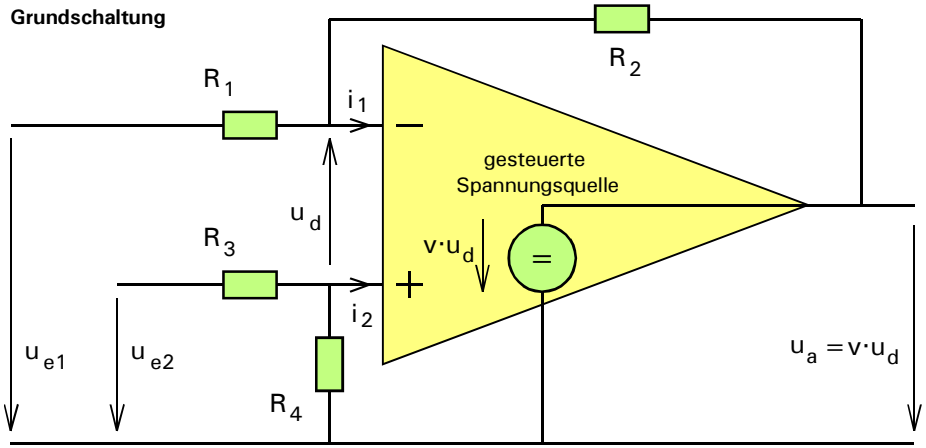
Die Übertragung erfolgt zwischen einem Master und einem Slave gleichzeitig in beiden Richtungen.

Master und Slave schreiben die zu übertragenden Daten parallel in die jeweiligen Schieberegister.

Der Master wählt über eine der Slave-Select-Leitungen SSx genau einen Slave aus und erzeugt das Taktsignal SCK für die Übertragung.

Mit jedem Takt wird ein Bit aus dem Schieberegister des Masters in das Schieberegister des Slave und gleichzeitig ein Bit aus dem Schieberegister des Slave in das Schieberegister des Masters übertragen.

Anhang: Einfache Operationsverstärkerschaltungen



Annahme für die Berechnung: Idealer OP im linearen Betrieb

d.h. a) OP-Eingangswiderstände $\rightarrow \infty$

$\rightarrow$ **OP-Eingangsströme** $\mathbf{i_1 = i_2 = 0}$

b) OP-Verstärkung $v \rightarrow \infty$

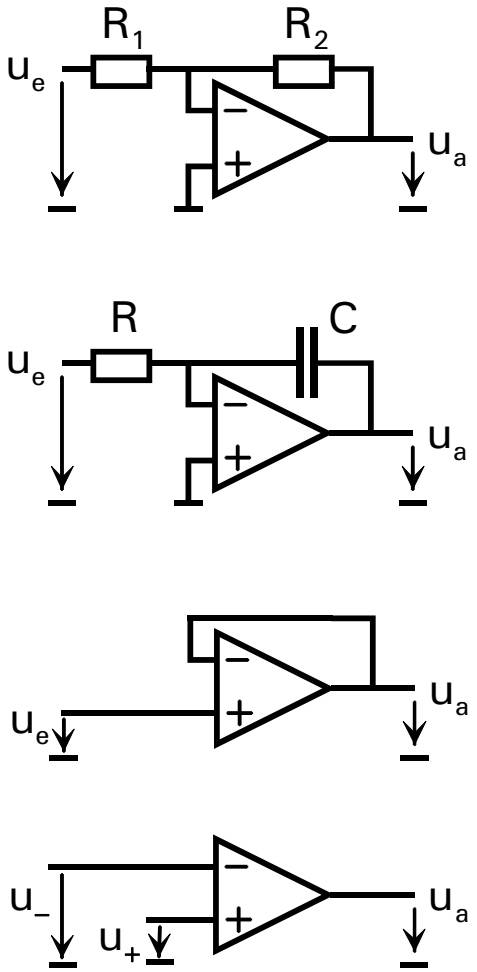
$\rightarrow$ **Differenzeingangsspannung** $\mathbf{u_d = 0}$

(d.h. gleiche Spannung am + Eingang und am − Eingang)

Damit unter Anwendung des Überlagerungssatzes für u_{e1}, u_{e2} und u_a:

$$u_d = \frac{R_4}{R_3+R_4} \cdot u_{e2} - \left[\frac{R_2}{R_1+R_2} \cdot u_{e1} + \frac{R_1}{R_1+R_2} \cdot u_a \right] = 0$$

$$\rightarrow \quad u_a = \frac{R_1+R_2}{R_1} \cdot \frac{R_4}{R_3+R_4} \cdot u_{e2} - \frac{R_2}{R_1} \cdot u_{e1}$$



Invertierender Verstärker

$$u_a(t) = -\frac{R_2}{R_1}\, u_e(t)$$

$$\frac{U_a(s)}{U_e(s)} = -\frac{R_2}{R_1}$$

Invertierender Integrierer

$$u_a(t) = -\frac{1}{RC} \int u_e(t)\, dt$$

$$\frac{U_a(s)}{U_e(s)} = -\frac{1}{sRC}$$

Impedanzwandler

$$u_a(t) = u_e(t)$$

$$\frac{U_a(s)}{U_e(s)} = 1$$

Analog-Komparator

(nicht-linearer Betrieb ohne Gegenkopplung)

$u_+ > u_- \rightarrow u_a = +1$

$u_+ < u_- \rightarrow u_a = 0$ (oder −1)