

# Zusammenfassung Architektur Eingebetteter System

Paul Nykiel

28. Juli 2019

This page is intentionally left blank.

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>4</b>
1.1	Architektur eines Eingebetteten Systems . . . . .	4
1.1.1	Eigenschaften eines Eingebetteten Systems . . . . .	4
1.1.2	Zusätzliche Herausforderungen beim Entwurf . . . . .	5
1.1.3	Entwurfsebenen . . . . .	5
1.2	Hardwarespezifikationssprachen . . . . .	6
1.2.1	Aufbau von VHDL-Beschreibungen . . . . .	7
1.2.2	Beispiel: Multiplexer . . . . .	7
1.3	Configuration . . . . .	8
1.4	VHDL-Simulationssemantik . . . . .	9
1.4.1	Signale Treiben . . . . .	9
1.4.2	Rückkopplungen auflösen . . . . .	9
1.4.3	Verzögerungen modellieren . . . . .	10
<b>2</b>	<b>Processing Elements</b>	<b>11</b>
2.1	Instruction Set Processor (ISP) . . . . .	12
2.1.1	von-Neumann-/Princeton-Architektur . . . . .	12
2.1.2	Befehlszyklus . . . . .	13
2.1.3	Harvard-Architektur . . . . .	14
2.2	Application Specific Instruction Set Processor (ASIP) . . . . .	15
2.3	Application Specific Processor . . . . .	15
2.4	Beispiel: Aufbau eines Schnurlosen DECT-Telefons . . . . .	16
2.5	Wie kommunizieren PEs in heterogenen Systemen? . . . . .	16
2.5.1	Gemeinsame Ressourcen (Speicherkopplung) . . . . .	16
2.5.2	Direkte Verbindung . . . . .	16
2.5.3	Tiled-Architecture . . . . .	17
<b>3</b>	<b>Halbleitertechnologien</b>	<b>18</b>
3.1	Full-Custom-Logic Fabric . . . . .	19
3.2	Semi-Custom (Gate-Array) . . . . .	20
3.3	Programmierbare Schaltungen (Programmable Gate Fabric) . . . . .	21
3.3.1	Programmable Logic Devices (PLDs) . . . . .	22

<b>4</b>	<b>Sensoren und Aktoren</b>	<b>25</b>
4.1	Grundbegriffe der Messtechnik . . . . .	25
4.1.1	DIN-1316: Messen . . . . .	25
4.2	Analog-Digital-Umwandler . . . . .	26
4.2.1	Zeitbasisumsetzer . . . . .	26
4.2.2	Operationsverstärker . . . . .	26
4.2.3	Spannungszeitumsetzer . . . . .	27
4.2.4	Spannungsfrequenzumsetzer . . . . .	28
4.2.5	Stufenumsetzer . . . . .	28
4.3	Digital-Analog-Umsetzer . . . . .	29
4.3.1	Parallelverfahren . . . . .	29
4.3.2	Wägeverfahren . . . . .	29
<b>5</b>	<b>Echtzeitbetriebssysteme</b>	<b>30</b>
5.1	Arten von Echtzeit . . . . .	30
5.2	Aufgaben eines (Echtzeit-)Betriebssystems . . . . .	30
5.3	Aufbau . . . . .	31
5.4	Unterbrechungen . . . . .	31
5.4.1	Folgen von Unterbrechungen . . . . .	31
5.5	Taskverwaltung . . . . .	32
5.5.1	Taskzustände . . . . .	33
5.5.2	Taskscheduler . . . . .	33

# Kapitel 1

## Einführung

Ein eingebettetes System ist in einen technischen Kontext oder Prozess eingebettet.

Im wesentlichen kann ein eingebettetes System als einen Computer, der einen technischen Prozess steuert oder regelt, betrachtet werden.

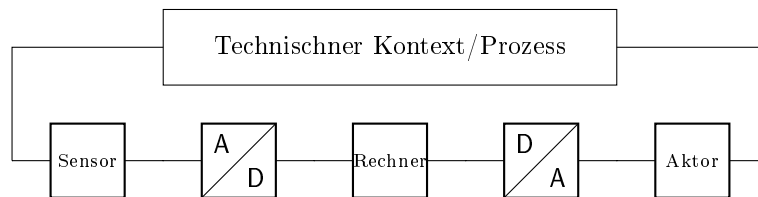


Abbildung 1.1: Aufbau eines Eingebetteten Systems

### 1.1 Architektur eines Eingebetteten Systems

#### 1.1.1 Eigenschaften eines Eingebetteten Systems

- Enge Verzahnung zwischen Hard- und Software
- Strenge funktionale und zeitliche Randbedingungen
- Zusätzlich zum Prozessor wird I/O Hard- und Software benötigt
- Oftmals wird Anwendungsspezifische Hardware benötigt

⇒ Keine „General-Purpose“ Lösung möglich

Zusätzliche Probleme:

- Wenig Platz
- Nur beschränkte Energiekapazität

- System darf nicht warm werden
- Kostengünstig

### 1.1.2 Zusätzliche Herausforderungen beim Entwurf

Die Entwicklung eines eingebetteten Systems ist kein reines Software-Problem, zusätzlich muss beachtet werden:

- Auswahl eines Prozessors, Signalprozessors, Microcontrollers
- Ein-/Ausgabe Konzept&Komponenten
  - Sensoren und Aktoren
  - Kommunikationsschnittstellen
- Speichertechnologien und Anbindung
- Systempartitionierung: Aufteilen der Funktionen der Komponente
- Logik- und Schaltungsentwurf
- Auswahl geeigneter Halbleitertechnologien
- Entwicklung von Treibersoftware
- Wahl eines Laufzeits-/Betriebssystems
- Die eigentliche Softwareentwicklung

⇒ Aufteilung des Entwurfs auf mehrer Entwurfsebene

### 1.1.3 Entwurfsebenen

Verhalten	Syntheseschritt	Entscheidungen	Test
System Specification	Systemsynthese	HW/SW/OS	Modelsimulator / Checker
Behavioural Specification	Verhalten / Architektursynthese	Verarbeitungseinheiten	HW/SW-Simulation
Register-Transfer-Specification	RT-Synthese	Register, Addierer, Mux	HDL-Simulation
Logic-Specification	Logiksynthese	Gatter	Gate-Simulation

Tabelle 1.1: Entwurfsebenen

Grafik

## 1.2 Hardwarespezifikationssprachen

- Verilog
- VHDL (Very High Speed Integrated Circuit Description Language)

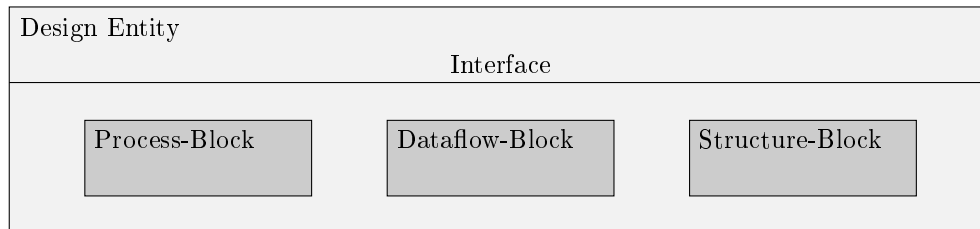


Abbildung 1.2: Aufbau einer Design-Entity

**Process-Block** Sequentiell abgearbeitete Logik:

```
process (clk)
begin
    ...
end
```

**Dataflow-Block** Konkurrent abgearbeitete Logik:

```
begin
    ...
end
```

**Structure-Block** Zusammenschalten weiterer Design-Entitäts:

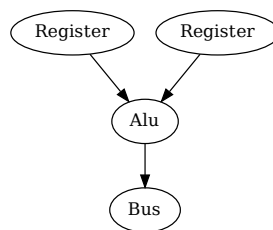


Abbildung 1.3: Structure-Block

### 1.2.1 Aufbau von VHDL-Beschreibungen

- **use**: Import von Bibliotheken
- **entity**: Schnittstellenbeschreibung
- **architecture**: Implementierung der Entity
- **configuration**: **architecture** zu **entity** auswählen

### 1.2.2 Beispiel: Multiplexer

Entity-Deklaration:

```
entity MUX is
    port(a,b,sel: in Bit;
          f: out Bit);
end MUX;
```

Als Process-Block

```
architecture BEHAVIOUR_MUX of MUX is
begin
    process(a,b,sel)
    begin
        if sel = '1' then f <= a;
        else f <= b;
        end process;
    end BEHAVIOUR_MUX;
```

Als Dataflow-Block

```
architecture DATAFLOW_MUX of MUX is
begin
    f <= a when sel = '1' else b;
end DATAFLOW_MUX;
```

alternativ geht auch:

```
architecture DATAFLOW_MUX of MUX is
begin
    f <= (a and sel) or (b and (not sel));
end DATAFLOW_MUX;
```

eine weitere Option:

```
architecture DATAFLOW_MUX of MUX is
    signal nsel, f1, f2 : Bit;
begin
    nsel <= not sel;
```



```

        f1 <= a and sel;
        f2 <= b and nsel;
        f <= f1 or f2;
    end DATAFLOW_MUX;

```

Alternativ: Mit Variablen  
**Als Structure-Block**

```

architecture STRUCTURE of MUX is
    component NOT
        port(i: in Bit; o: out Bit);
    end component;
    component AND
        port(i1, i2: in Bit; o: out Bit);
    end component;
    component OR
        port(i1, i2: in Bit; o: out Bit);
    end component;
    signal nsel, f1, f2: Bit;
begin
    g1: AND port map(a, sel, f1);
    g2: AND port map(b, nsel, f2);
    g3: OR port map(f1, f2, f);
    g4: NOT port map(sel, nsel);
end STRUCTURE;

```

Laut Skript  
 geht das so  
 nicht, sollte  
 aber eigent-  
 lich schon?

## 1.3 Configuration

Rekursive die Architektur für jede Entity auswählen:

```

configuration STRUCTURE_MUX of MUX is
    for STRUCURE
        for g1,g2: AND use entity MYAND(BEHAVIOUR_AND)
            ...
        end for;
    end for;
end STRUCTURE_MUX;

```

Dann muss die oben genutzte Entity und Architektur natürlich noch definiert werden:

```

entity MYAND is
    port(i1, i2: in Bit;
        o: out Bit);
end MYAND;

architecture BEHAVIOUR_MYAND is

```

Was genau  
 passiert da  
 jetzt?

```

        o <= i1 and i2;
    end BEHAVIOUR_MYAND;

```

## 1.4 VHDL-Simulationssemantik

Aufgaben des Simulators:

- Signal treiben/propagieren
- Rückkopplungen auflösen
- Verzögerungen modellieren

### 1.4.1 Signale Treiben

Signale werden durch eine Event-Queue repräsentiert, das heißt nicht die Signale selber, sondern nur Änderungen der Signale (z.b. Flanken) werden gespeichert. Die Event-Queue besteht aus „Transaktionen“ jede Transaktion ist ein Tuple aus der Zeit zu der die Änderung auftritt, und der Änderung selber.

Zum Beispiel:

```

y <= '0' after 0ns, '1' after 10ns, '0' after 20ns;

```

Wird als Event-Queue so dargestellt:

$$\{ \langle 0, '0' \rangle, \langle 10, '1' \rangle, \langle 20, '0' \rangle \}$$

### 1.4.2 Rückkopplungen auflösen

Transaktionen können echt parallel stattfinden (Ereignisse treten asynchron und ggf. gleichzeitig auf).  $\Rightarrow$  Es kann zu Konflikten kommen („Henne-Ei-Problem“), z.B. bei einem zero-delay RS-Latch:

```

1      x <= not (y and lset);
2      y <= not (x and reset);

```

Lösung: Tagged-Event-Queue bzw. Delta-Delay: Jeder Zeitpunkt wird um eine „zweiten Dimension“ ergänzt, Events die direkt nacheinander (z.B. als direkte Folge) auftreten (mit einem Delta-Delay) werden entlang dieser zweiten Dimension geordnet.

t	lset	x	y	reset	Zeile
0	↓	0	1	1	1
$0 + \Delta$	0	↑	1	1	2
$0 + 2\Delta$	0	1	↓	1	1
$0 + 3\Delta$	0	1	0	1	✓
10	↑	1	0	1	1
$10 + \Delta$	1	1	0	1	✓

Tabelle 1.2: Tagged-Event-Queue für den zero-delay RS-Latch

### 1.4.3 Verzögerungen modellieren

Durch Schaltzeiten, Kapazitäten, Laufzeiten etc. kommt es in echten Systemen zu Verzögerungen der Signale. Diese müssen daher auch in VHDL modelliert werden können. Dafür wird zwischen zwei Arten unterschieden:

- Langsames Ansprechverhalten, das heißt kurze Pulse werden nicht durchgelassen
- Verzögerung der Signale

Beispiel (Inverter):

```
out <= reject 10 ns inertial not in after 30ns;
```

Die Verzögerungen werden in VHDL durch `reject` für langsames Ansprechverhalten (hier muss der Puls mindestens 10ns dauern) und `after` für Verzögerungen (hier 30ns) modelliert.

Für reine Laufzeitverzögerungen kann in VHDL auch `transport` genutzt werden, folgende Befehle sind äquivalent:

```
out <= transport in after 30ns;
out <= reject 0ns inertial in after 30ns;
```

wait\_until  
und generic

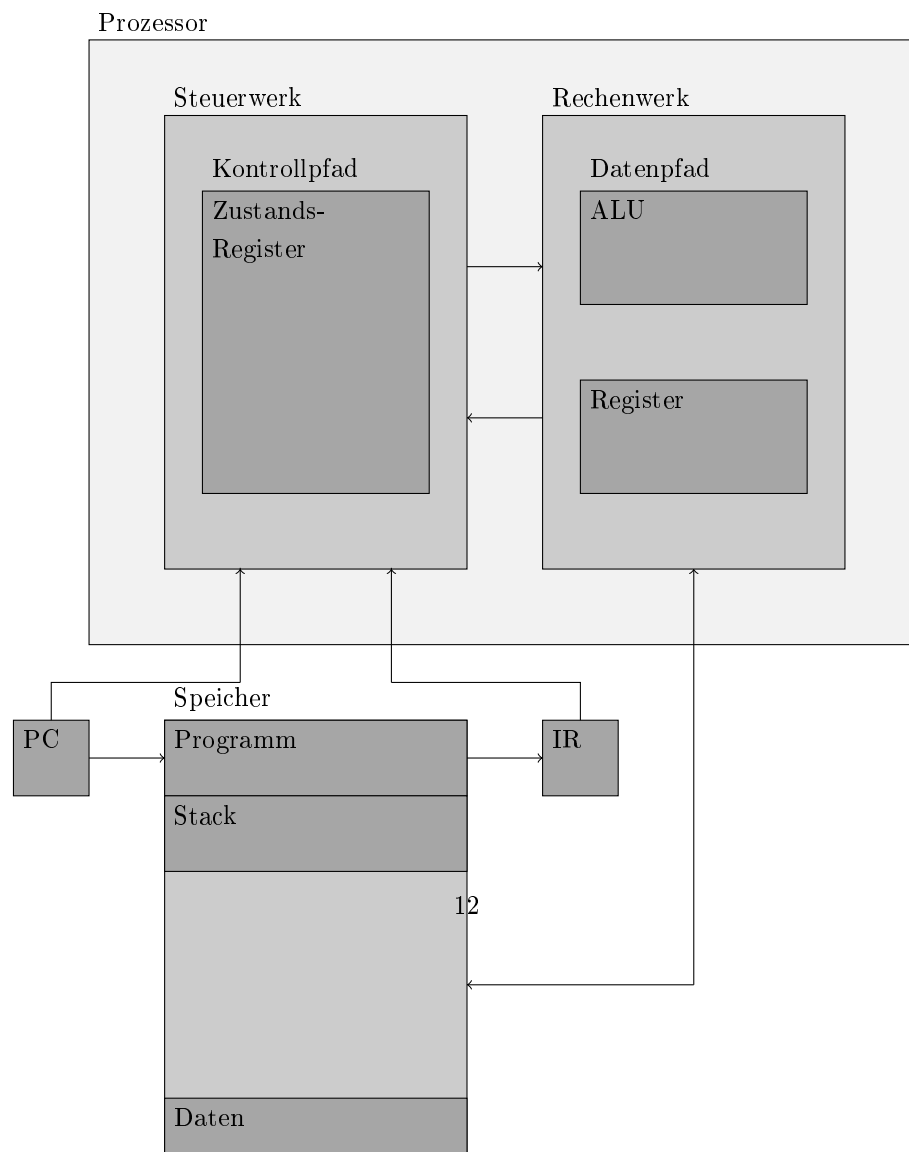


# Kapitel 2

## Processing Elements

### 2.1 Instruction Set Processor (ISP)

#### 2.1.1 von-Neumann-/Princeton-Architektur



### 2.1.2 Befehlszyklus

1. Befehl holen (fetch)
2. Befehl dekodieren (decode)
3. Operanden holen (load)
4. Befehl ausführen (execute)
5. Daten speichern (write back)

Pro	Contra
Analyse einfach Speicher flexibel benutzbar	Auslastung gering von-Neumann Flaschenhals (Daten und Befehle über den selben Bus)

### 2.1.3 Harvard-Architektur

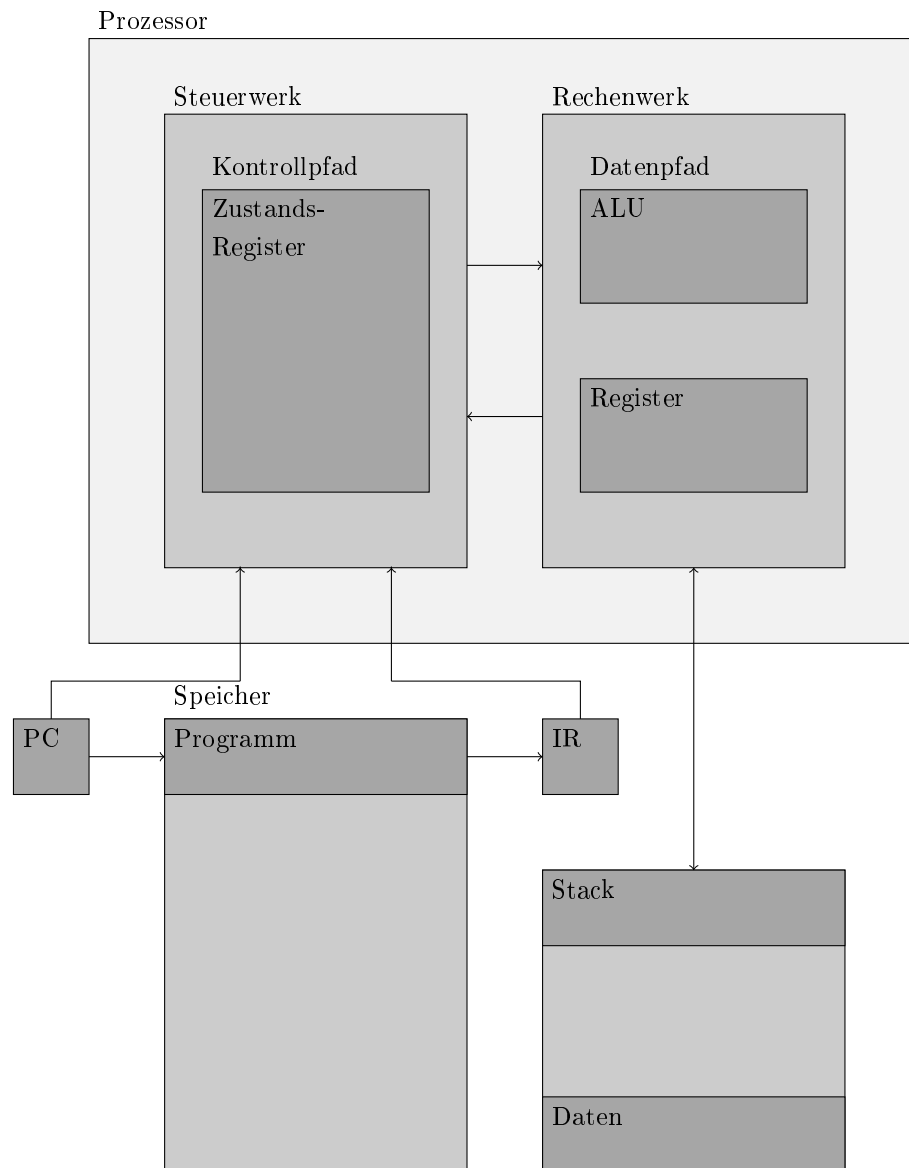


Abbildung 2.2: Aufbau eines Harvard Prozessors

Pro	Contra
Auslastung	Fragmentierter Speicher
Kein Flaschenhals	Analyse schwierig
Schnellere Abarbeitung	Schwierig bei Datenabhängigkeiten

## 2.2 Application Specific Instruction Set Processor (ASIP)

Regulärer ISP wird durch zusätzliche Instruktionen (und damit auch Hardware) für ein bestimmtes Einsatzgebiet optimiert, z.B. durch zusätzlich „Multiply-Accumulate“-Einheit oder auch komplette FFT-Operationen. Beispiel: DSP (Digital Signal Processpr).

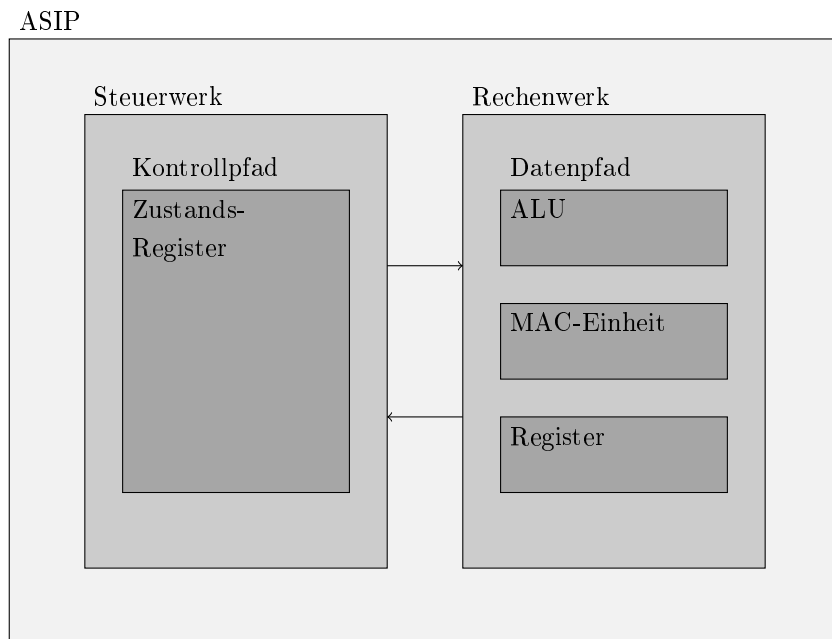


Abbildung 2.3: Aufbau eines ASIP mit zusätzlicher MAC-Einheit

## 2.3 Application Specific Processor

Nicht mehr programmierbar, Prozessor kann nur wenige vorkonfigurierte Befehle ausführen, Steuerung erfolgt über eine spezielle Logik, oftmals in Form einer State-Machine.



ASP

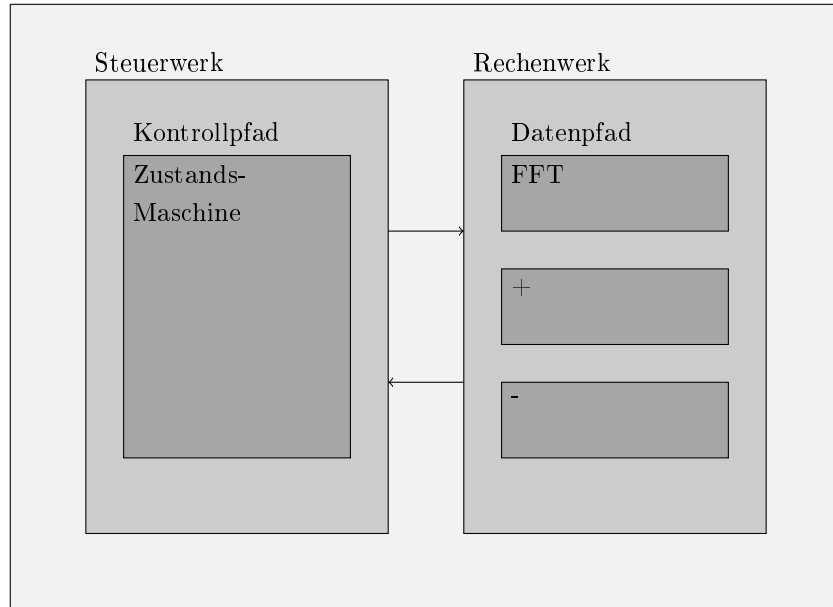


Abbildung 2.4: Aufbau eines ASP, der nur FFT, + und - Operationen durchführt

## 2.4 Beispiel: Aufbau eines Schnurlosen DECT-Telefons

Oftmals wird ein ISP mit zugehöriger RF-Hardware auf einem Chip-Integriert (z.B. in Mobiltelefonen), dann spricht man von einem Sytem-on-a-Chip (SoC).

Grafik

## 2.5 Wie kommunizieren PEs in heterogenen Systemen?

### 2.5.1 Gemeinsame Ressourcen (Speicherkopplung)

Alle PEs können über einen Bus auf eine gemeinsame Ressource (für gewöhnlich Speicher) zugreifen. Für die Synchronisation ist ein „Arbiter“ (Richter) sowie ein Bus-Controller in jedem PE notwendig.

Grafik

### 2.5.2 Direkte Verbindung

Direkte Verbindung zwischen allen PEs die kommunizieren müssen, Arbitrierung ist nicht notwendig, mehrere PEs können gleichzeitig kommunizieren. Es ist wieder ein Controller für jede Verbindung von nöten, zudem muss dieser even-

tuell Daten puffern. Bei  $n$  PEs sind im Worstcase  $\frac{(n-1) \cdot n}{2} \in \mathcal{O}(n^2)$  Verbindung notwendig.

### 2.5.3 Tiled-Architecture

Bei einer Tiled-Architecture oder Network-on-a-Chip (NOC) werden jeweils wenige PEs über eine gemeinsame Resource verbunden. Diese Gruppen an PEs werden dann über ein Gitter aus Leitungen verbunden.



Grafik

## Kapitel 3

# Halbleitertechnologien

Jedes PE, egal ob ISP, ASIP oder ASP muss gefertigt werden. Hierfür muss sich für eine Herstellungsart entschieden werden.



Abbildung 3.1: Optionen für die fertigung von Halbleiterkomponenten

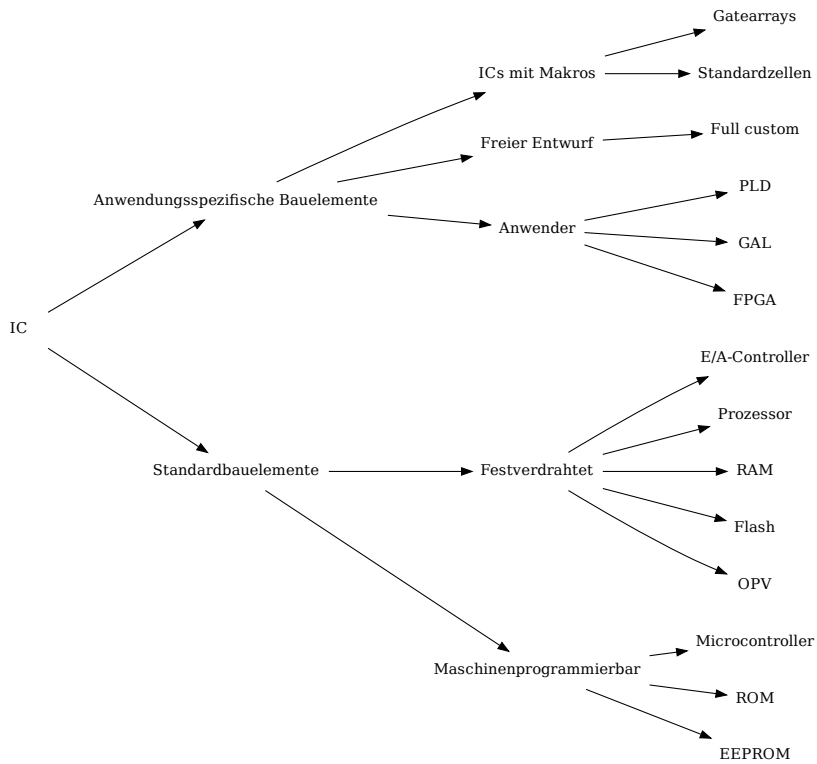


Abbildung 3.2: Überblick über ICs

### 3.1 Full-Custom-Logic Fabric

Sehr teuer, dafür aber maximale Performance. Lohnt sich im Normalfall nur für sehr große Stückzahlen. Beispiele für Full-Custom gefertigte Komponenten:

- PC-Processoren
- GPUs
- ASIPs (z.b. Basisbandprozessoren)
- Microcontroller

Üblicherweise werden Full-Custom-Komponenten in „complementary metal-oxide-semiconductor“ (CMOS) Technik gefertigt.

## 3.2 Semi-Custom (Gate-Array)

Ein Gate-Array besteht aus vielen, nicht verbundenen, Transistorzellen, sogenannten Basismakros. Vom Anwender kann sowohl die Funktion jeder Logikzelle (NAND oder NOR) sowie die Verbindungen zwischen den Logikzellen bestimmt werden. Semi-Custom-Komponenten sind deutlich größer als Full-Custom-Komponenten.

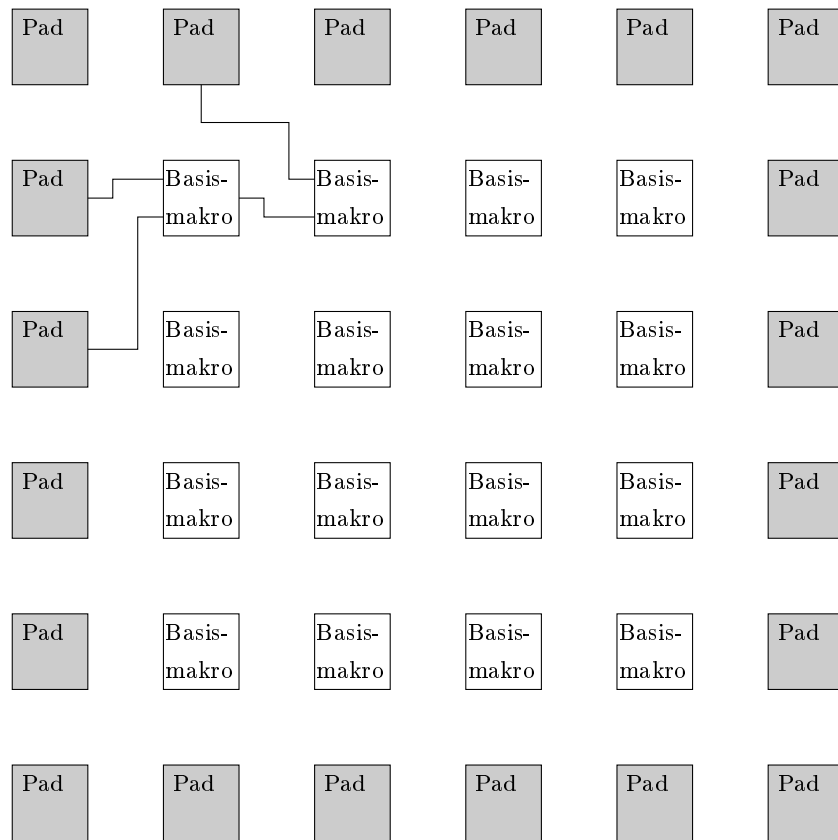


Abbildung 3.3: Aufbau eines Gate-Arrays

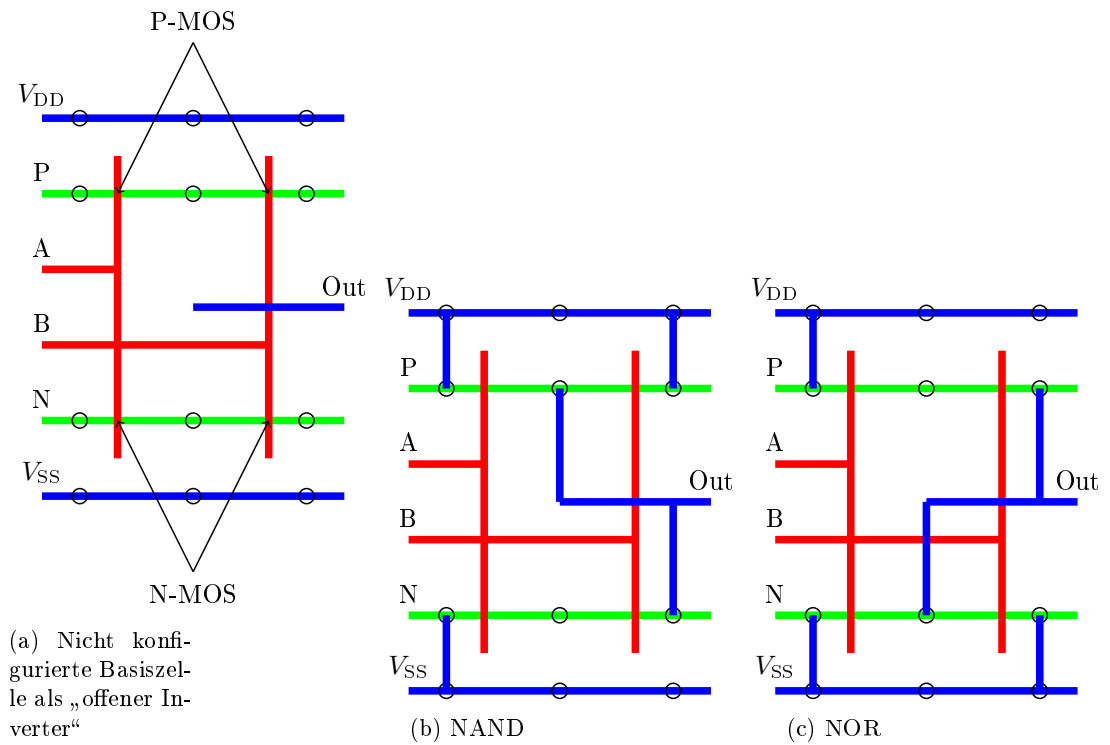


Abbildung 3.4: Aufbau und Nutzung der Transistorzellen/Basismakros

### 3.3 Programmierbare Schaltungen (Programmable Gate Fabric)

Ziel: Logikelemente selbst verbinden  $\Rightarrow$  Schaltelemente

Hinweis: Auch Logikelemente selber können konfigurierbar sein.

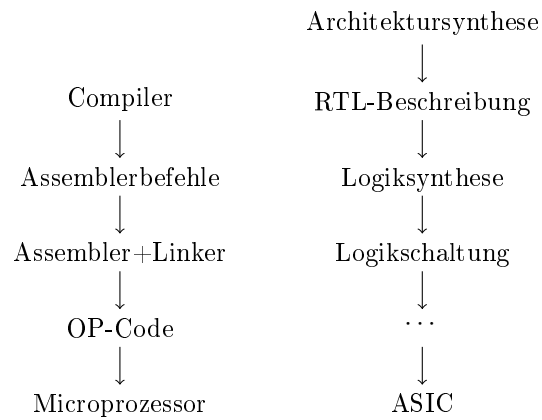


Abbildung 3.5: Vergleich von Hardwarebeschreibung und „normaler“ Software

### 3.3.1 Programmable Logic Devices (PLDs)

- 3 Klassen:
  - SPLD (Simple Programmable Logic Device)
  - CPLD (Complex Programmable Logic Device)
  - FPGA (Field Programmable Gate Array)
- 2 Ebenen:
  - Funktionale Ebene (Logistrukturen)
  - Konfigurationsebenen
- Verbindungen werden über eine SRAM-Zelle geschaltet

Was heißt das jetzt?

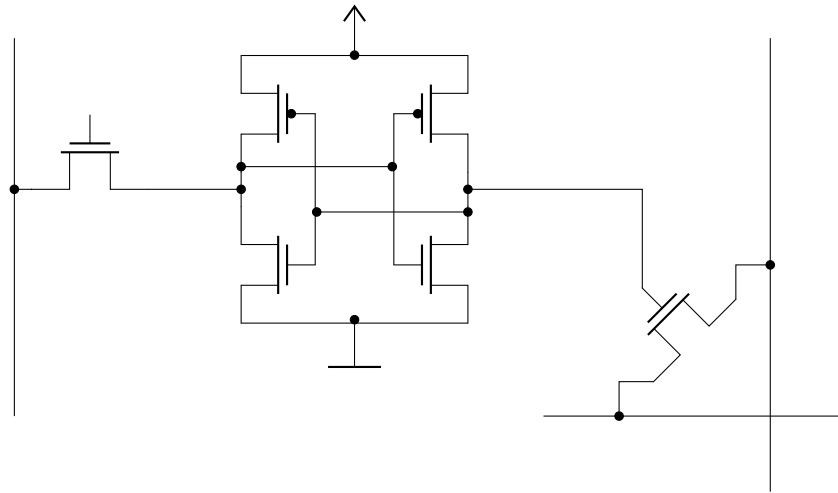


Abbildung 3.6: SRAM-Zelle in einem PLD

- Problem mit SRAM: Speicher ist flüchtig
- Alternativen: EPROM, EEPROM, Flash
- Einmalige Programmierung (Anti-Fuse)

Bild Flash

Grafik

### SPLDs

Mögliche Formen von SPLDs sind:

- PLA: Und und Oder Matrix, beide Programmierbar
- PROM: Und festverdrahtet, Oder Programmierbar (nicht jede Boolesche Funktion abbildbar)
- PAL: Und Programmierbar, Oder fix

Grafik

### CPLDs

Viele Funktionsblöcke, dazwischen „Fast Connect Switch Matrix“.

Grafik

### FPGAs

Beliebige Logikfunktionen darstellbar durch Look-Up-Table (LUT).

Grafik

**Slice (kleinster Baustein)** Ein Slice besteht aus zwei LUTs, zugehöriger Carry-Logik um Slices effizienter nacheinander zu schalten, sowie zwei Flip-Flops.

Grafik



**Makro Cell (bei Xilinx: Configurable Logic Block, CLB)** Eine Makro Cell wird über ein Bus-Netzwerk (vgl. Tiled-Architecture) angeschlossen.



Grafik

# Kapitel 4

## Sensoren und Aktoren

Zur Kommunikation mit dem technischen Prozess sind Sensoren und Aktoren, die eine physikalische Größe in eine Spannung/Strom (bzw. vice versa) umwandeln notwendig. Um die Daten digital zu verarbeiten ist zudem ein Analog-Digital-Umsetzer (ADC) sowie ein Digital-Analog-Umsetzer (DAC) notwendig. Beim Auslegen des Eingebetteten Systems muss entschieden werden, in welchem Punkt die Digital-Analog-Wandlung stattfindet: entweder an den Sensoren/Aktoren, das heißt die Daten werden dann digital übertragen oder an der CPU, das heißt die Signale werden dann analog übertragen.

ADC/DAC bei der CPU	ADC/DAC bei den Sensoren/Aktoren
+ Weniger Bauteile benötigt	+ CPU-Modul weniger komplex
+ Höhere Integration	+ Angepasster ADC/DAC möglich
- Störanfällig	- Kommunikation nicht trivial
- Analog-Prozess in Fertigung	

Tabelle 4.1: Vor und Nachteile für verschiedene Platzierung des ADC/DAC

Es gibt Analoge Sensoren (z.B. Entfernung), sowie digital Sensoren (z.B. Lichtschranke).

### 4.1 Grundbegriffe der Messtechnik

#### 4.1.1 DIN-1316: Messen

**Messen** Ein experimenteller Vorgang, durch den eine physikalische Größe als Vielfaches einer Einheit oder eines Bezugswertes ermittelt wird.

**Messumformer** Ein analoges Eingangssignal wird in ein eindeutig damit zusammenhängendes analoges Ausgangssignal unter der Aufbringung von Energie

Mehr im  
Moodle

umgeformt (Verstärker).

**Messwandler** Messumformer, der die selbe physikalische Größe am Ein- und Ausgang aufweist, und ohne externe Energie auskommt.

**Messumsetzer** Messgerät, das am Eingang und Ausgang verschiedene Signalstrukturen (Analog/Digital) aufweist.

⇒ In ES sind Analog-Digital-Umwandler von besonderer Bedeutung.

## 4.2 Analog-Digital-Umwandler

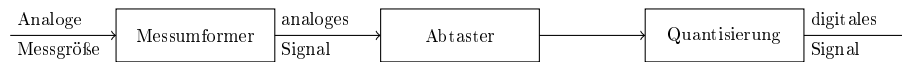


Abbildung 4.1: Aufbau eines ADUs

Es wird nach zwei Arten von Umsetzern unterschieden:

- **Zeitdiskrete-ADUs:** Die Messgröße wird in regelmäßigen Abständen abgetastet und eine Probe der Messgröße genommen. Da die Umsetzung eine gewisse Zeit benötigt muss das analoge Signale für diese Zeit konstant sein.
- **Zeitkontinuierliche-ADUs:** Umsetzer erfordert keine Abtastung, da sie keine Zeit zum Umsetzen benötigen. Der Digitalwert steht also kontinuierlich zur Verfügung (kann aber trotzdem verzögert zum analogen Wert sein).

### 4.2.1 Zeitbasisumsetzer

Die am einfachsten umsetzbare physikalische Größe ist die Zeit. Umsetzung: Vergleich einer Referenzimpulsdauer  $\Delta t$  mit der Periodendauer  $T = \frac{1}{f}$  einer Pulsfolge der Frequenz  $f$  mittels einer Torschaltung (Und-Gatter).

Unbedingt beachten: durch die Zeitdiskretisierung gibt es eine Ungenauigkeit, es gilt ( $m$  sei die Anzahl der Pulse):

$$\frac{m-1}{f} \leq \Delta t \leq \frac{m+1}{f}$$

Grafik

### 4.2.2 Operationsverstärker

**Nicht-invertierender OPV**

$$A = 1 + \frac{R_N}{R_1}$$

Grafik

### Invertierender OPV

$$A = -\frac{R_N}{R_1}$$

Grafik

### Invertierender Addierer

$$\sum_{k=0}^n \frac{U_n}{R_n} = -\frac{U_N}{R_N}$$

Grafik

### Invertierender Integrator

$$U_a = -\frac{1}{R \cdot C} \int_0^t U_e(\tau) d\tau + U_{a0}$$

Grafik

## 4.2.3 Spannungszeitumsetzer

### Sägezahnumsetzer

Idee: Die Eingangsspannung wird mit einer Sägezahnspannung, die immer bis zu einer Referenzspannung ansteigt verglichen. Es wird die Zeit gemessen, in der die Sägezahnspannung kleiner als die Eingangsspannung ist, die anliegende Spannung ist proportional zu dieser Zeit ( $U_a$  ist die Eingangsspannung,  $U_0$  eine Referenzspannung,  $t_i$  eine Referenzzeit):

$$\Delta t = \frac{U_a}{U_0} \cdot t_i$$

Probleme: Steigung muss möglichst genau sein, d.h. der Sägezahngenerator muss möglichst genau sein.

Blockschaltbild

Timing

### Dual-Slope/Doppel-Integrationsverfahren/Zweirampenverfahren

Idee: Integrator wird in einer gegebenen Zeit mit der Eingangsspannung geladen. Dann wird er mit einer gegebenen (negativen) Referenzspannung ( $-U_{\text{ref}}$ ) entladen und die Zeit bis der Integrator wieder 0 ist bestimmt. Diese Zeit ist proportional zur Spannung. Es gilt ( $t_0$  ist die Startzeit,  $t_1$  die Zeit bei der von Laden zu Entladen gewechselt wird,  $t_2$  die Endzeit,  $U_e$  ist die Eingangssignal,

$U_{\text{ref}}$  die Referenzspannung):

$$\begin{aligned}
 T_1 &= n \cdot T = t_1 - t_0 \\
 U_{t_1} &= -\frac{1}{RC} \cdot \int_{t_0}^{t_1} U_e dt \\
 &= \frac{U_e \cdot t_1 - U_e \cdot t_0}{RC} \\
 &= -U_e \cdot \frac{t_1 - t_0}{RC} \\
 &= -U_e \cdot \frac{T_1}{RC} \\
 T_2 &= m \cdot T = t_2 - t_1 \\
 U_{t_2} &= 0 = U_{t_1} - \left( \frac{1}{RC} \cdot \int_{t_1}^{t_2} -U_{\text{ref}} dt \right) \\
 \Leftrightarrow 0 &= -U_e \frac{T_1}{RC} + U_{\text{ref}} \frac{T_2}{RC} \\
 \Leftrightarrow U_e &= \frac{T_2}{T_1} U_{\text{ref}} \\
 \Leftrightarrow U_e &= \frac{m}{n} U_{\text{ref}}
 \end{aligned}$$

Die Genauigkeit der Messung ist also von keinen Bauteilwerten abhängig, auch die Frequenz ist für die Genauigkeit nicht relevant. Zudem kann durch dieses Verfahren auch die Referenzspannung kalibriert werden (mit  $U_{\text{ein}} = U_{\text{ref}}$ ).

Blockschaltbild

Timing

#### 4.2.4 Spannungsfrequenzumsetzer

Idee: Integrator wird mit Eingangsspannung bis zu gewisser Referenzspannung geladen. Dann wird ein Puls gesendet und der Integrator geleert. Dadurch entsteht eine Sägezahnwelle am Integrator und eine Reihe von Pulsen mit der Frequenz proportional zur Eingangsspannung am Ausgang. Für die Spannung gilt dann also ( $N$  ist die Anzahl an Peaks,  $f$  die Frequenz der Peaks,  $t_m$  die Zeit über die gemessen wird,  $U_m$  die Eingangsspannung,  $U_f$  die Vergleichsspannung):

$$N = f \cdot t_m = \frac{1}{U_f RC} U_m t_m$$

Dadurch wird automatisch über die gemessene Spannung gemittelt, die Messung dauert allerdings auch länger.

Blockschaltbild

Timing

#### 4.2.5 Stufenumsetzer

##### Schablonenumsetzer

Für einen  $n$ -Bit Umsetzer werden  $2^n - 1$  Vergleichsspannungen generiert. Die Messspannung wird mit allen diesen Spannung verglichen, es entsteht ein  $2^n - 1$ -Bit-Signal bei dem alle Bits bis zu der anliegenden Spannung gesetzt sind, durch eine Decode-Logik wird daraus die Spannung bestimmt.

Nachteil: Anzahl der Komparatoren skaliert exponentiell.

Blockschaltbild

$\frac{U_e}{U_{\text{LSB}}}$	$k_7$	$k_6$	$k_5$	$k_4$	$k_3$	$k_2$	$k_1$	Bin
0	0	0	0	0	0	0	0	000
1	0	0	0	0	0	0	1	001
2	0	0	0	0	0	1	1	010
3	0	0	0	0	1	1	1	011
4	0	0	0	1	1	1	1	100
5	0	0	1	1	1	1	1	101
6	0	1	1	1	1	1	1	110
7	1	1	1	1	1	1	1	111

Tabelle 4.2: Decoder Logik für einen 3-bit Schablonenumsetzer

### Kompensationsumsetzer / Wägeverfahren

Idee: Referenzspannung wird mit DAC erzeugt und mit Messspannung verglichen, durch Bisektion wird die Messspannung bestimmt.

Nachteil: Braucht Zeit und DAC.

Blockschaltbild

## 4.3 Digital-Analog-Umsetzer

### 4.3.1 Parallelverfahren

Siehe Schablonenumsetzer,  $2^n - 1$  mögliche Spannungen, die relevante Spannung wird an den Ausgang geschaltet.

Nachteil: skaliert ebenfalls mit exponentiell mit der Auflösung.

Blockschaltbild

### 4.3.2 Wägeverfahren

Idee: Spannungen für jede Stelle der Binärzahl ( $\frac{1}{2}U_{\text{ref}}$ ,  $\frac{1}{4}U_{\text{ref}}$ , ...), die je nach Bit zugeschaltet wird.

Vorteil: Schnell, skaliert linear mit der Auflösung. Nachteil: Widerstände stark unterschiedlicher Größen benötigt.

Blockschaltbild

### Leiternetz

Optimierung des Wägeverfahrens, nur Widerstände zweier Größen benötigt.

Blockschaltbild

# Kapitel 5

## Echtzeitbetriebssysteme

### 5.1 Arten von Echtzeit

- **Harte Echtzeit:** Die Aufgabe muss zwingend vor der Deadline abgearbeitet sein
- **Fest Echtzeit:** Wenn die Information nach der Deadline kommen sind sie irrelevant
- **Weiche Echtzeit:** Die Aufgabe sollte meistens vor der Deadline abgearbeitet sein

### 5.2 Aufgaben eines (Echtzeit-)Betriebssystems

Standardaufgaben eines Betriebssystems:

- Taskverwaltung
- Betriebsmittelverwaltung
- Interprozesskommunikation
- Synchronisationsaufgaben
- Schutzmaßnahmen

Ein Echtzeitbetriebssysteme (RTOS) muss zudem folgende Aufgaben erfüllen:

- Rechtzeitigkeit
- Gleichzeitigkeit
- Verfügbarkeit

## 5.3 Aufbau

Ein (Echtzeit-)Betriebssystem ist normalerweise in einem Schichtenmodell aufgebaut.

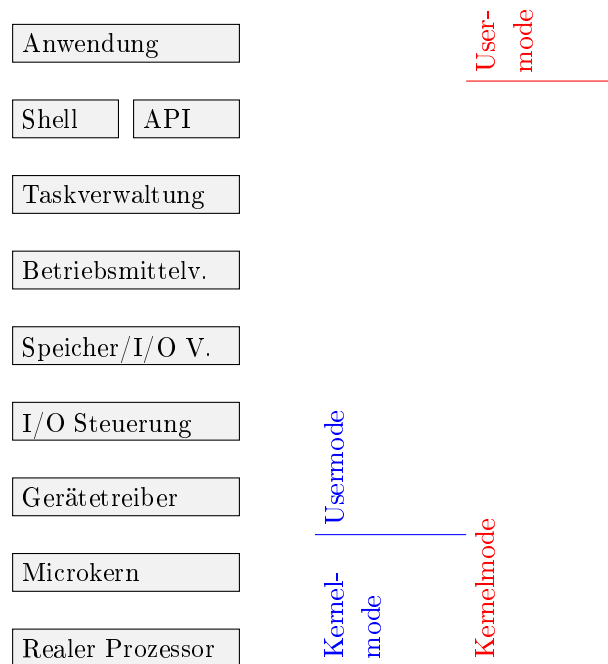


Abbildung 5.1: Beispielhafter Aufbau eines Betriebssystems (Microkernel in blau, Makrokern/Monolithischer Kernel in rot)

## 5.4 Unterbrechungen

### 5.4.1 Folgen von Unterbrechungen

Es gibt nicht unterbrechbare Funktionen („non-reentrant“), z.B. (tmp ist hier global):

```
void swap(int *x, int *y) {  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

Mögliche Lösungen:

Grafik, was auch immer die Aussage?



- tmp lokal definieren
- Interrupts für Funktion deaktivieren (kritisch)
- tmp schützen

Die meisten Funktionen sind unterbrechbar, z.B.:

```
void strcpy(char *dst, const char *src) {  
    while (*dst++ = *src++);  
    *dst = '\0';  
}
```

## 5.5 Taskverwaltung

Ein Task („Rechenprozess“) ist ein ablaufendes Programm zusammen mit Variablen und Betriebsmitteln. Tasks besitzen:

- Aktionsfunktionen („Programm“)
- Zustandsvariablen (Speicher, Register, PC)

---

Grafik

### 5.5.1 Taskzustände

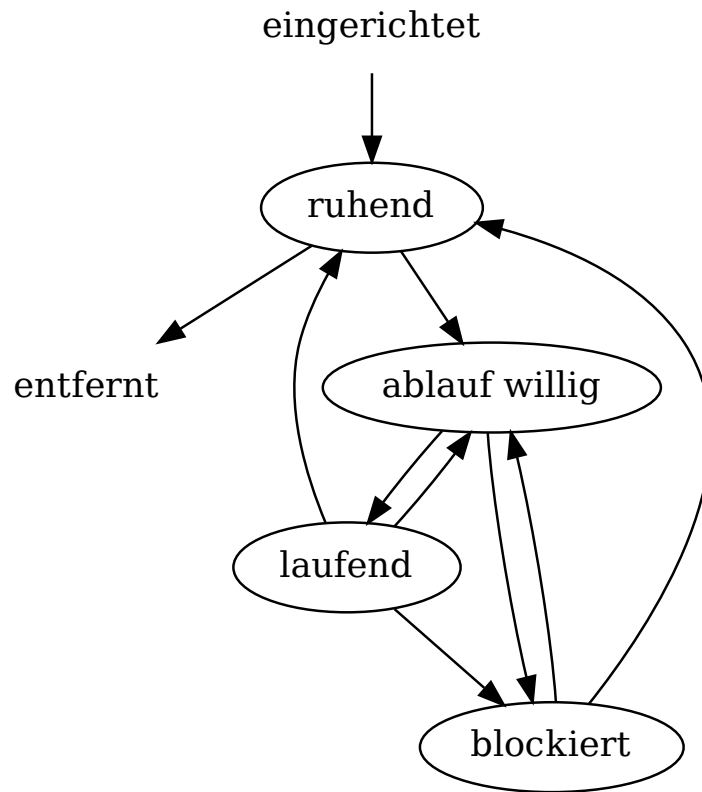


Abbildung 5.2: Zustandsmaschine der Taskübergänge

### 5.5.2 Taskscheduler

**Taskscheduler** Welcher Task soll ausgeführt werden?

**Taskdispatcher** Notwendigen Abläufe um Task zur Ausführung zu bringen  
Sobald mehr als ein Task benötigt wird muss ein Scheduler genutzt werden.

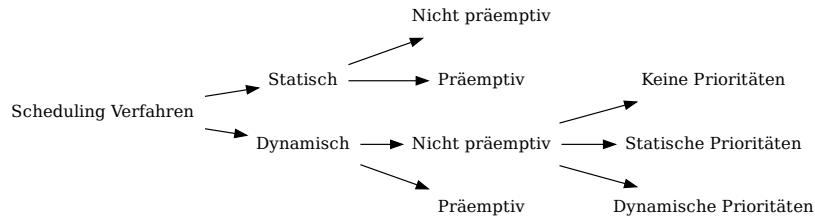


Abbildung 5.3: Übersicht über mögliche Scheduling-Verfahren

### Polled-Loop-Systeme

Der einfachst mögliche RT-Kernel:

```

int main(void) {
    while (true) {
        if (event) {
            event = false;
            threadCall(threadPtr);
        }
    }
}

```

Nachteile: Nur ein thread kann verwaltet werden, bei Burst-Nachrichten (mehrere Nachrichten während der Thread läuft) wird nur eine abgearbeitet.

### Polled-Loop für mehrere Tasks

```

int main(void) {
    while (true) {
        if (event1) {
            event1 = false;
            threadCall(thread1Ptr);
        }
        if (event2) {
            event2 = false;
            threadCall(thread2Ptr);
        }
        if (event3) {
            event3 = false;
            threadCall(thread3Ptr);
        }
    }
}

```

Gleiche Probleme wie oben (Burst). Zusätzlich kann ein Thread alle anderen aushungern.

#### **Phase-/State-Driven Systems**

```
void isr() {
    eventFlag = 1;
}

int main(void) {
    switch(eventFlag) {
        case 1:
            threadCall(thread1Ptr);
            break;
        ...
    }
}
```

Ebenfalls Probleme bei Bursts, Race-Conditions um `eventFlag`.