


Anwendungsspezifische (programmierbare) Di- gitallogik und VHDL- Synthese

**Dr. Stefan Bosse
Bremen**

2. Auflage



Anwendungsspezifische (programmierbare) Digitallogik und VHDL-Synthese



1. Vorwort

Dieses Skript stellt eine Vorlesungsbegleitung dar und soll eine Einführung in den Hardware- und Systementwurf mit anwendungsspezifisch konfigurierbarer Digitallogik mittels VHDL-Synthese und deren Anwendungen bieten. Hardware-Synthese ist ein automatischer Prozeß, um aus einer Verhaltens- und Strukturbeschreibung logische Schaltungen und Netzlisten zu erhalten, die direkt technologisch umsetzbar sind. Die verwendete Hardware-Beschreibungssprache sollte dabei unabhängig von der Zieltechnologie sein. Beim Hardware-Entwurf spielen System-On-Chip-Methoden eine wesentliche Rolle.

Zentraler Inhalt ist die Digitallogik und deren Einsatz zur Lösung bestimmter Probleme der Datenverarbeitung. Technologischer Fortschritt ermöglicht komplexe Digitallogiksysteme auf kleinstem Raum, das sog. System-On-Chip-Konzept. Es gibt verschiedene Technologien für verschiedene Zwecke, um anwendungsspezifische Systeme zu implementieren.

Traditionelle Datenverarbeitung ist programmorientiert unter Verwendung generischer Mikroprozessorkonzepte. Im Gegensatz dazu reine anwendungs- und problemorientierte Digitallogik.

► Zu jedem Modul gibt es vertiefende Literatur, die am Ende des Skripts aufgeführt ist.

Modul I ⇒

Einführung und Motivation

[BAT02][PRO96][CIL03]

Modul II ⇒

Einführung in die Digitaltechnik

[CIL03][KOR04]

Modul III ⇒

Programmierbare Logikbausteine

[HER04][CIL03]

Modul IV ⇒

Entwurfs- und Syntheseverfahren

[HER04][REI03][CIL03]

Module V ⇒

Register-Transfer-Logik und Zustandsautomaten

[REI03][CIL03]

Übersicht

2. Einführung und Motivation

2.1. Digitaltechnik

Die Digitaltechnik ist Grundlage der Elektronischen Datenverarbeitung (EDV). Die Applikationen der EDV sind vielfältig:

- Numerik
- Steuerung von Maschinen
- Reaktive Systeme mit Benutzerinteraktion
- Digitale Signalverarbeitung

➔ Die Informationsverarbeitung erfordert eine Mensch-Maschine-Schnittstelle, d.h. die Kodierung von Informationen zu Daten, die eigentliche Datenverarbeitung, und die anschließende Rückgewinnung der Informationen aus Daten.

Abbildung 1: Informationsverarbeitung

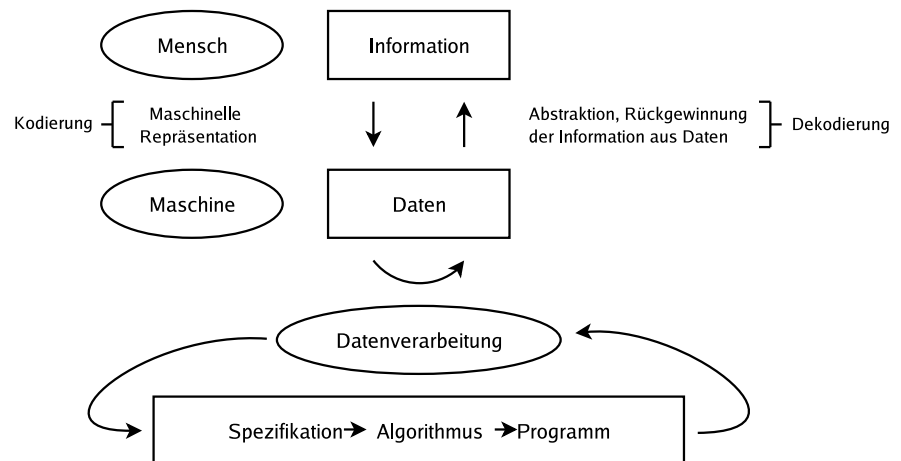


fig001.eps

➤ Die Repräsentation und Kodierung von Informationen erfolgt durch Bitfolgen im Binärzahlensystem. Ein Bit enthält die Informationsmenge bestehend aus zwei Elementen:

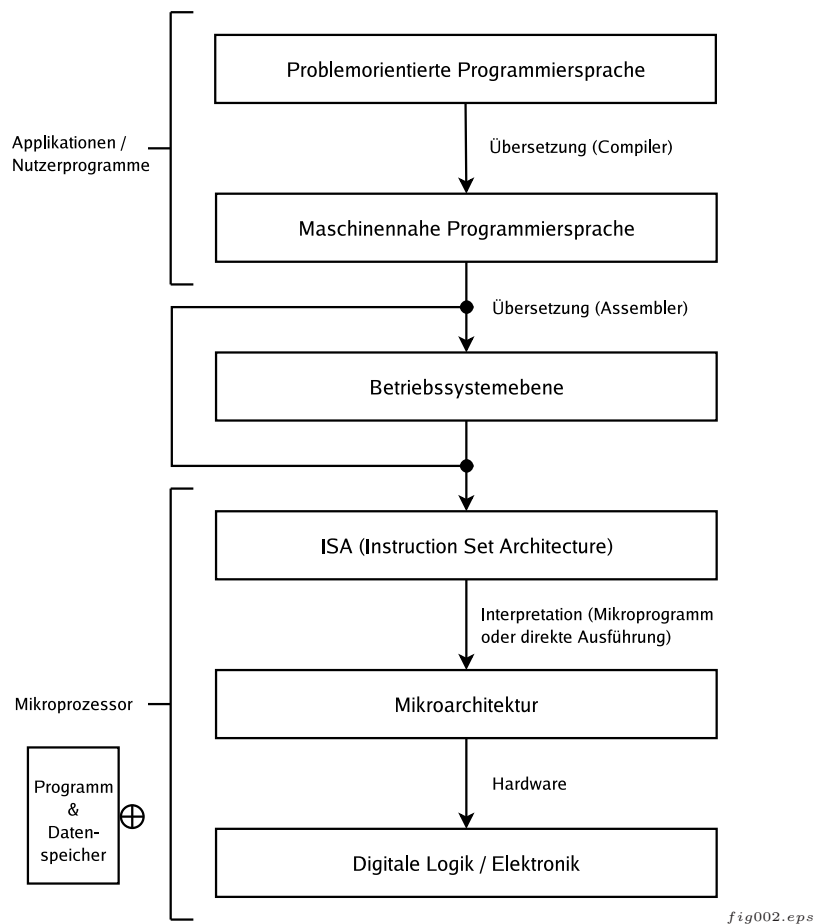
{wahr, falsch}, {hell, dunkel} usw.

Die Kodierung dieser Informationsmenge und technische Umsetzung erfolgt durch Übergang von logischen auf physikalische Größen wie Spannungen oder Ströme:

{1, 0} → {H, L} → {3.3V, 0V}

➤ Ausgangspunkt und Motivation für anwendungs- und problemorientierte Digitallogik liegt in generischer Rechnerarchitektur und herkömmlichen programm-basierten Problemlösungen begründet. Dazu soll das Schichtenmodell einer konventionellen Datenverarbeitungsanlage näher betrachtet werden.

Abbildung 2:
Struktur einer generischen Datenverarbeitungsanlage.



ISA-Ebene.

➤ Die Instruktionsebene des Mikroprozessors kann mit zwei unterschiedlichen Architekturen implementiert werden:

1. direkt mit digitaler Logik realisiert,
2. oder Transformation von komplexen ISA-Maschinenbefehlen in eine Untermenge einfacher Maschinenbefehle (sequenzielle Komposition).
 - ➔ insbesondere bei CISC-Prozessoren (z.B. Intel P4)
 - ➔ RISC-Prozessoren können auf Mikroarchitektur verzichten.

Abstraktion

➤ Die Betriebssystemebene ermöglicht die Abstraktion der Rechnerarchi-

Von-Neumann-
Rechnerarchitektur

tektur, die ISA-Ebene ermöglicht die Abstraktion von der Digitallogikebene.

Ein klassisches Mikroprozessorsystem arbeitet ein Maschinenprogramm (\equiv Ablaufvorschrift) sequenziell ab.

► Zum näheren Verständnis der Programmsteuerung muß der Aufbau und Struktur einer generischen von-Neumann-Anlage näher betrachtet werden.

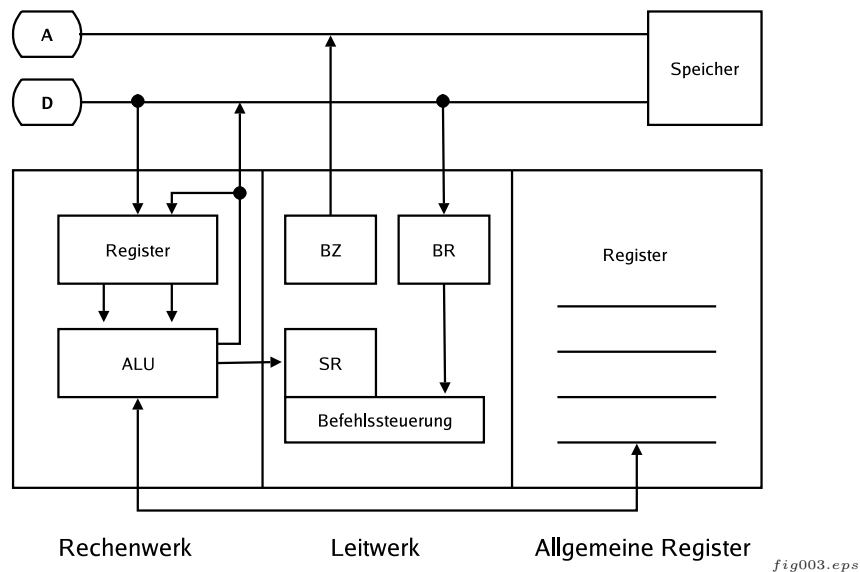


Abbildung 3: von-Neumann-Modell

Die Rechnerarchitektur ist unterteilt in das Rechenwerk, das Leitwerk und ein Satz allgemeiner Register (temporäre Datenspeicher). Das Leitwerk enthält folgende spezielle Register:

BZ: Befehlzähler \Rightarrow Zeigt auf Speicheradresse des nächsten auszuführenden Maschinenbefehls.

BR: Befehlsregister \Rightarrow Enthält Kodierung des aktuell ausgeführten Befehls.

SR: Statusregister \Rightarrow Enthält Informationen über aktuelle oder bereits ausgeführte Operationen, wie Zero- oder Carry-Bits, und beeinflusst die Programmausführung (z.B. bedingte Verzweigung).

Das Rechenwerk, das Leitwerk und der Registersatz sind über einen Daten- und Addressbus mit einem (einzigen) Hauptspeicher verbunden, dessen Speicherzellen über die Adresse ausgewählt wird.

Bussystem

► Ein Bussystem ist eine Gruppe von elektrischen Signalleitungen zur Datenübertragung, und verbindet mehrere Kommunikationsteilnehmer. Bei einem Datenaustausch auf einem Bus ist immer ein Teilnehmer schreibend und ein anderer lesend aktiv.

Phasen der Befehlsausführung

►

1. Befehlsholphase (BH): $\varphi(\text{BZ}) \Rightarrow \text{BR}$
2. Dekodierungsphase (DE): $\text{BR} \Rightarrow \{\text{S1}, \text{S2}, \text{S3}, \dots\}$
3. Operandenholphase (OP): $\varphi \Rightarrow \text{R}$
4. Ausführungsphase (AU): χ
5. Rückschreibephase (RS): $\text{R} \Rightarrow \varphi$
6. Adressierungsphase (AD): $\text{ADDR}(\text{BZ}, \text{SR}, \text{BR}) \rightarrow \text{BZ}$

Dabei bezeichnen φ den Speicher des Rechners, χ eine Aktion und \Rightarrow einen Datentransfer.

► Ein von-Neumann-Rechner ist taktgesteuert, d.h. die Befehlsausführung findet zu zeitlich äquidistanten Zeitpunkten statt.

► Neben der Programmbearbeitung- und Steuerung gibt es Geräte für die Ein- und Ausgabe von Daten. Bei generischen Rechnern sind dies Geräte für die Nutzerinteraktion und allgemeinen Datenübertragung, in der digitalen Signalverarbeitung hauptsächlich Geräte für die Signalwandlung.

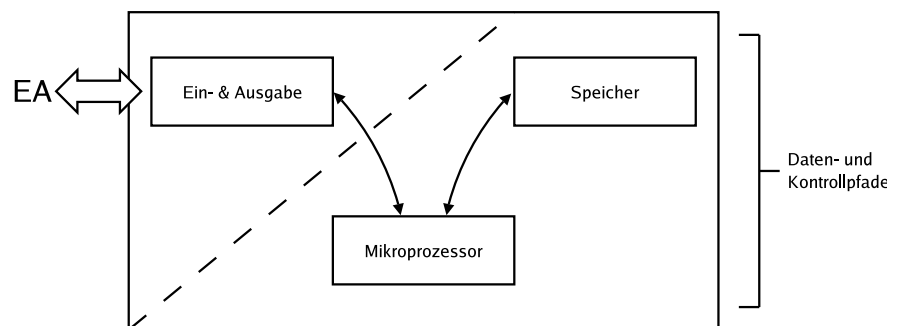


fig005.eps

Abbildung 4: Daten- und Kontrollpfade

Analoge
(physikalische)
Signale

Abbildung 5: Zoom
eines analogen
Signals in Zeit- und
Wertdimension.

2.2. Digitale Signalverarbeitung

Neben Rechnersystemen, die generisch/universell eingesetzt werden können (Desktop-Rechner) gibt es spezialisierte Rechnersysteme für die Signalverarbeitung.

► Analoge Signale sind zeit- und wertkontinuierlich, d.h. man findet in einem beliebig kleinen Intervall $[a,b]$ immer eine Zahl c für die gilt: $a \leq c \leq b$. Ein analoges Signal besitzt aufgrund physikalischer Vorgänge die Eigenschaft keinen exakten und zeitlich konstanten Wert zu besitzen, sondern setzt sich zusammen aus einer Überlagerung mit einem stochastischen Rauschsignal.

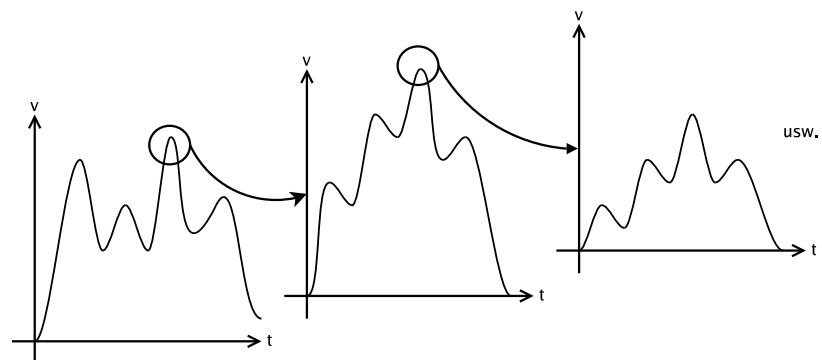
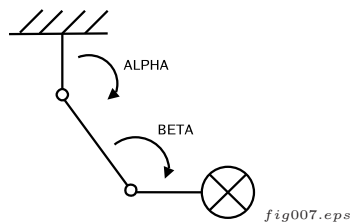


fig006.eps

Beispiele für analoge Signale:

- Spannung, Strom, Druck, Temperatur
- Position eines Robotergelenks:



- Entfernung eines Gegenstandes:

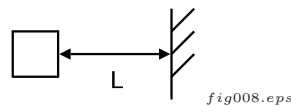
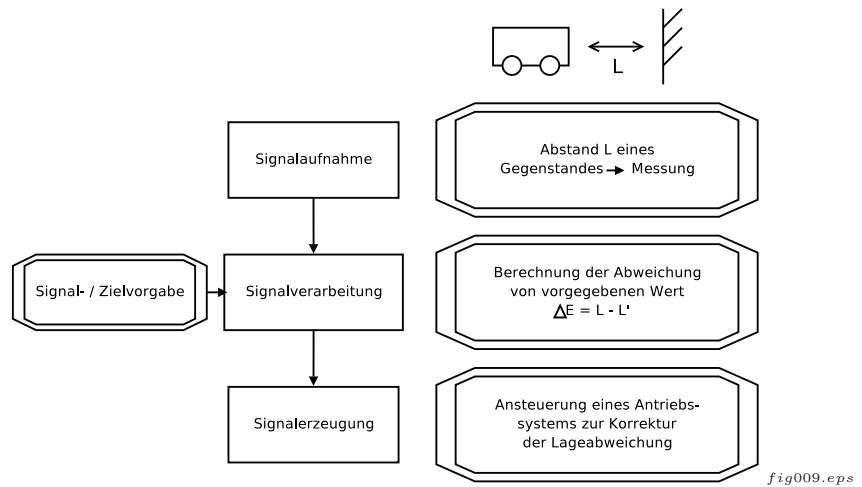


Abbildung 6:
Signalverarbeitung

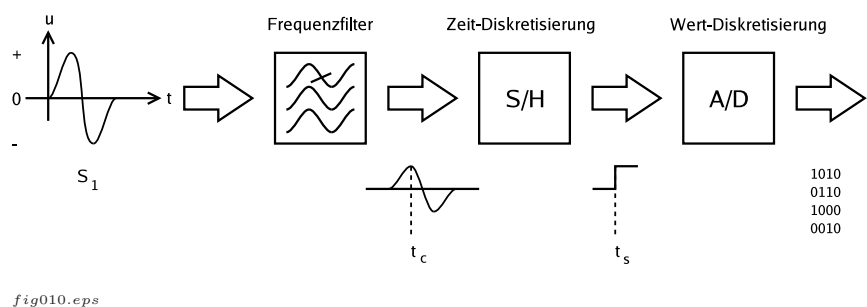


► Die Signalaufnahme und Erzeugung stellt die Ein-/Ausgabeeinheiten dar. Die Signalverarbeitung besteht aus einer Signal- oder Zielvorgabe verknüpft mit einem Algorithmus, der dieses Ziel erreichen soll.

► Die digitale Signalverarbeitung erfordert:

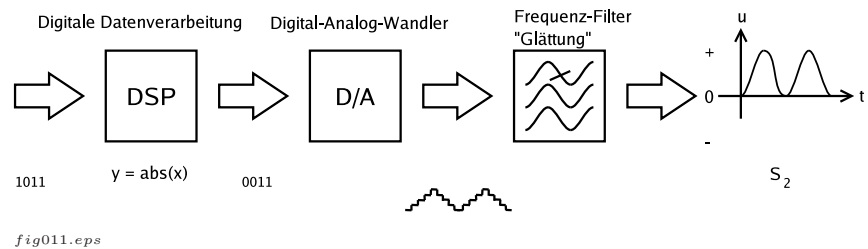
1. Digitalisierung der analogen Eingangssignale (Analog-Digital-Wandler).
2. Erzeugung von analogen Signalen aus digitaler Information (Digital-Analog-Wandler).

Abbildung 7:
Digitalisierung als erste Stufe der Signalverarbeitung



► Die AD-Wandlung setzt i.A. einn Frequenzfilter (Tiefpaß) am Eingang voraus, mit dem der zu erfassende Spektralbereich des Signals begrenzt wird. In der sog. Sample&Hold-Schaltung wird das analoge Signal zeitdiskretisiert, und anschließend mit dem eigentlichen AD-Wandler in einen diskreten i.A. binärkodierten Digitalwert umgesetzt.

Abbildung 8:
Digitale
Signalverarbeitung als
zweite und
Erzeugung analoger
Signale als dritte
Stufe.



► Die DA-Wandlung kann nur ein quasi-analoges Signal (immer noch zeit- und wertdiskret!) erzeugen. Eine "Zeit- und Wertglättung" findet hier ebenfalls unter Verwendung eines Tiefpaß-Frequenzfilters statt.

Anwendungen der
Digitalen
Signalverarbeitung

►

- Digitale Filterung
- Faltungsoperationen
- Korrelationsanalyse
- Zeit \leftrightarrow Frequenztransformationen (FFT)
- Wellenformerzeugung
- Bildverarbeitung
 1. Digitale Filterung
 2. Mustererkennung
 3. 3D-Operationen
- Spracherkennung
- Steuerungs- und Regelungsaufgaben
 1. Positionsregelung
 2. Spannungsregelung
 3. Motorsteuerung (z.B. Drehzahl)
 4. Navigation

2.3. Aufbau eines AD-Wandlers

Ein AD-Wandler ist in drei Stufen unterteilt:

1. Der Sampler führt eine Diskretisierung in der Zeitdimension durch,
2. der Quantisierer führt eine Diskretisierung in der Wertdimension durch, derart, daß ein quantisierter Wert einem Wertintervall $q(n-\Delta) \leq q(n) < q(n+\Delta)$, mit $\Delta/2$ als Auflösung des Quantisierers, entspricht,
3. und einem Kodierer, der das quantisierte Signal in ein Digitalwert kodiert, i.A. Kodierung nach dem Dualzahlensystem oder Gray-Kodierung mit der Eigenschaft, daß aufeinanderfolgende Werte immer nur eine Änderung eines einzigen Bits hervorrufen.

Abbildung 9:
Struktur des
AD-Wandlers

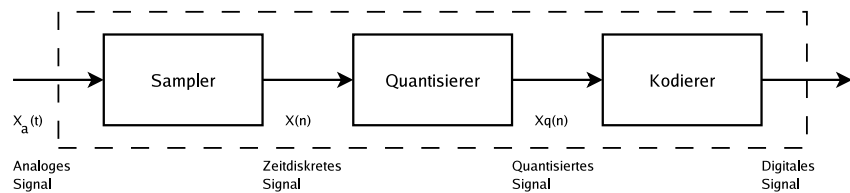


Abbildung 10:
Diskretisierung eines
analogen Signals

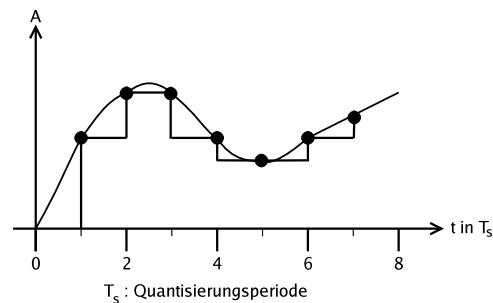
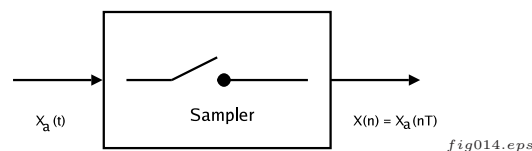


Abbildung 11:
Sampler



► Im allgemeinen wird in der digitalen Signalverarbeitung ein analoges Signal periodisch mit einer Sample-Frequenz f_{Sample} abgetastet. Das Sampling-Theorem besagt, daß das zu digitalisierende Signal nur ein Frequenzspektrum bis zu einer maximalen Frequenz f_{Signal} besitzen darf:

$$f_{\text{Sample}} > 2f_{\text{Signal}} \quad (1)$$

2.4. Daten- und Kontrollfluß in der digitalen Signalverarbeitung

Ein Algorithmus für digitale Signalverarbeitung (Digital Signal Processing DSP) besteht aus drei Komponenten:

1. Signalerfassung - zeitlich diskret
2. Signalverarbeitung
3. Signalerzeugung

Es gibt verschiedene Darstellungsmethoden, um einen Algorithmus symbolisch zu veranschaulichen:

1. Signalflußdiagramm

► Ein Signalflußdiagramm besteht aus folgenden Komponenten:

1. Arithmetische Operationen:

$\oplus \Rightarrow$ Summation

$$y(n) = \sum_i x_i(n)$$

$\blacktriangleright \Rightarrow$ Multiplikation

$$y(n) = x(n) \bullet k$$

2. Verzögerungsglieder für eine Akquisitionsperiode T_s

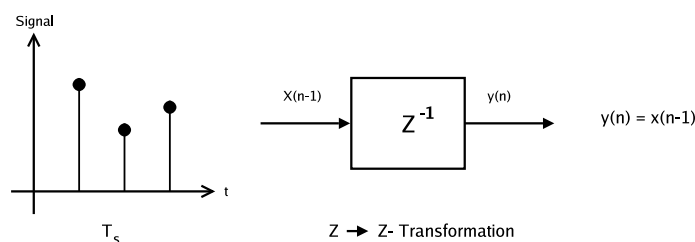


fig015.eps

Z-Transformation

► Signale können dual im Zeit- und Frequenzbereich/Raum beschrieben und manipuliert werden.

Die Z-Transformation ersetzt die Variablen des Signals, eine Zeitgröße, durch eine komplexe Frequenzvariable Z:

$$F(z) = \sum_{n=0}^{\infty} f(n)Z^{-n} \quad (2)$$

↳ Transformation von Signal- in den Spektralbereich

↳ Zeit- und Spektralbereiche sind äquivalent, in jedem ist die vollständige Information über das Signal enthalten

↳ $Z^{-1} \equiv 1/T_s \equiv \Delta n = -1$ mit T_s : Diskretisierungszeit

2. Daten- und Ablaufdiagramme

► Daten- Flußdiagramme beschreiben den Datenfluß zusammen mit einer sequenziellen Ablaufsteuerung.

3. Programmabläufe

► Programmiersprachen beschreiben auf einer abstrakten Ebene textuell einen Algorithmus.

**Beispiel:
Mittelwertfilter**

A. Spezifikation

► Ein einfacher Mittelwertfilter für drei Eingangssignale $\{x_0, x_1, x_2\}$ und einer Gleichrichtungsfunktion $f(x)$.

► Definition des Problems mit mathematischen Formalismus:

$$x = x_0 \bullet c_0 + x_1 \bullet c_1 + x_2 \bullet c_2$$

$$y = f(x)$$

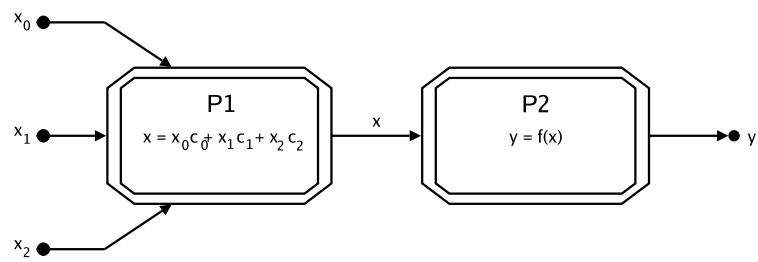
$$f(x) = \begin{cases} x & \text{für } x > 0 \\ 0 & \text{sonst} \end{cases}$$

wobei $\{c_1, c_2, c_3\}$ Konstanten sind.

► Das Datenflußdiagramm zerlegt Aufgabe in Teilprozesse und führt eine Partitionierung durch:

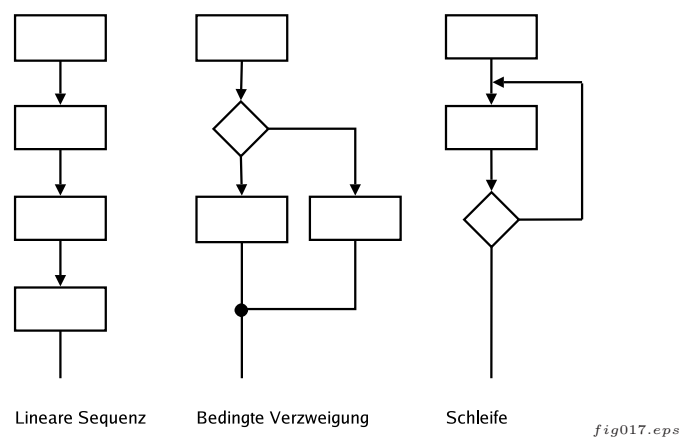
B. Algorithmus

Abbildung 12:
Datenflußdiagramm



Das Flußdiagramm beschreibt den sequenziellen Kontrollfluß, der sich aus dem Algorithmus und der Spezifikation ergibt.

Abbildung 13:
Ablauf-
Flußdiagramm



Man unterscheidet zwischen einer linearen Sequenz, einer bedingten Verzweigung und Schleifen als sequenzielles Implementierungsmittel eines Algorithmus. Aus einem Flußdiagramm lassen sich direkt imperative Pro-

Programm für Mikroprozessor

gramme ableiten. Die Implementierung obiger Problemspezifikation in einer imperativen Programmiersprache könnte wie folgt aussehen:

```
BEGIN
  y <- x0c0;
  y <- y+x1c1;
  y <- y+x2c2;
  IF y < 0 THEN y <- 0;
END;
Alternativ mit Schleife:
BEGIN
  y <- 0;
  FOR i = 0 TO 2
  DO
    y <- y + x(i)c(i);
  DONE;
  IF y < 0 THEN y = 0;
END;
```

► Ein Mikroprozessor besitzt i.A. nur eine arithmetische Logikeinheit und kann zu einem bestimmten Zeitpunkt nur eine arithmetische Operation durchführen. Für die Summation werden daher mindestens 5 sequenzielle Operationen benötigt, unabhängig ob das Programm mit einer Schleife oder linear ausgeführt wird. Zusätzlich wird mindestens eine Operation für die Funktion $f(x)$ benötigt. Wenn ein Mikroprozessor ein Maschinenbefehl pro Taktzyklus ausführen kann, werden hier wenigstens 6 Taktzyklen benötigt. Nicht optimierte Mikroprozessoren benötigen für die Befehlsausführung jeweils 6 Taktzyklen, insgesamt 36 Taktzyklen für diese einfache Mittelwertbildung!

Abbildung 14:
Signalflußdiagramm

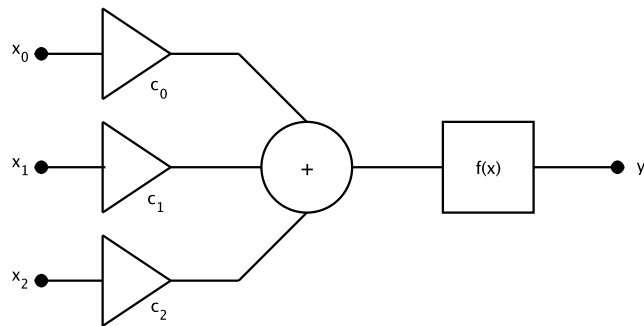


fig020.eps

Das Signalflußdiagramm ist unabhängig von einer Technologie oder Implementierung eines Problems, und beschreibt nur die wesentlichen Funktionen, aber nicht deren Verhalten oder Struktur.

Kombinatorische
Digitallogik

► Man kann nun zeigen, daß eine für dieses Problem angepaßte und spezifizierte Digitallogikschaltung aus dem Signalflußdiagramm direkt abgeleitet werden kann. Es stehen in der Digitaltechnik Systemblöcke für arithmetische und boolesche Operationen zur Verfügung, die durch sog. **kombinatorische Logik** realisiert werden können, d.h. Logik, die sich nur aus Grundlogikfunktionen wie Und, Oder- und Negierungsverknüpfung zusammensetzen, und deren Ausgänge nur einer Funktion der aktuell anliegenden Eingangssignale sind.

- ➔ Ausnutzung von Nebenläufigkeit von Teilprozessen
- ➔ Parallelisierte Digitallogik

Abbildung 15:
Implementierung
durch Digitallogik

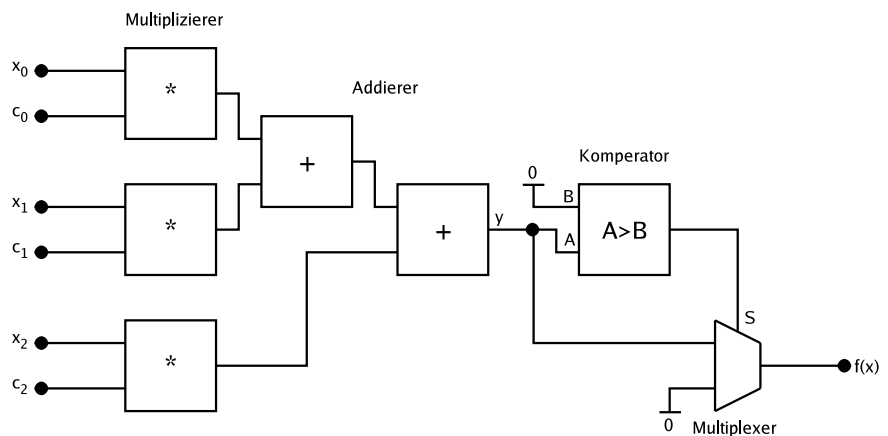


fig021.eps

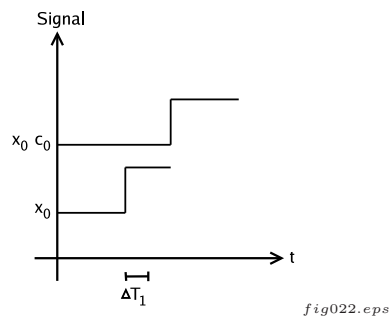
► Die Bearbeitungszeit, d.h. die Zeitdifferenz zwischen dem gültigen Anliegen der Eingangsdaten/signale und dem gültigen Anliegen der Ausgangsdaten/signale, ist aber in der Realität ungleich Null.

► Grund liegt in elektronischen Signalverzögerungen in den einzelnen

Digitallogikbausteinen. Die Signallaufzeiten summieren sich in der Bearbeitungskette auf und setzen sich zusammen aus Signallaufzeit T^L (kein Signal breitet sich schneller als mit Lichtgeschwindigkeit aus) und einer elektronischen Signalverzögerung T^E , die technologisch bedingt ist:

$$T = \sum_i T_i^L + \sum_j T_j^E \quad (3)$$

Abbildung 16:
Signalverzögerung
zwischen Ein- und
Ausgang einer
Digitallogik



2.5. Sequenzielle Systeme

Am Beispiel der Mittelwertbildung eines zeitlich sequenziellen Datenstroms soll die Signalauswertung und deren Umsetzung in Digitallogik betrachtet werden.

Abbildung 17:
Diskrete Abtastung
eines analogen
Signals und
Mittelwert.

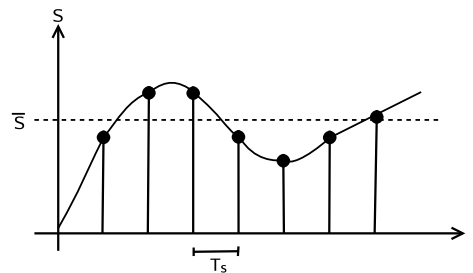


fig023.eps

Berechnung des Mittelwerts einer Folge diskreter Werte mittels:

$$S(N) = \frac{1}{N} \sum_{n=1}^N s(n) \quad (4)$$

➔ Nicht geschlossen mit einem Signalflußdiagramm darstellbar, da es einen ausgewiesenen Anfangs- und Auswertezustand geben muß (Initialisierung erforderlich) - N ist variabel!

➔ Näherung: Exponentielle Mittelwertbildung \equiv Tiefpaß-Filter 1. Ordnung:

$$S(n) = S(n) \cdot (1 - b_1) + s(n-1) \cdot b_1 \quad (5)$$

Abbildung 18:
Exponentielle
Mittelwertbildung
mit rückgekoppelten
Filter 1. Ordnung

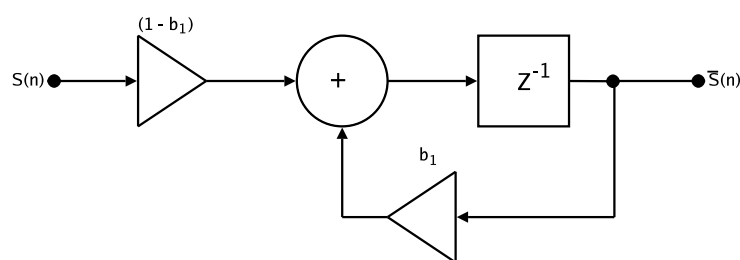


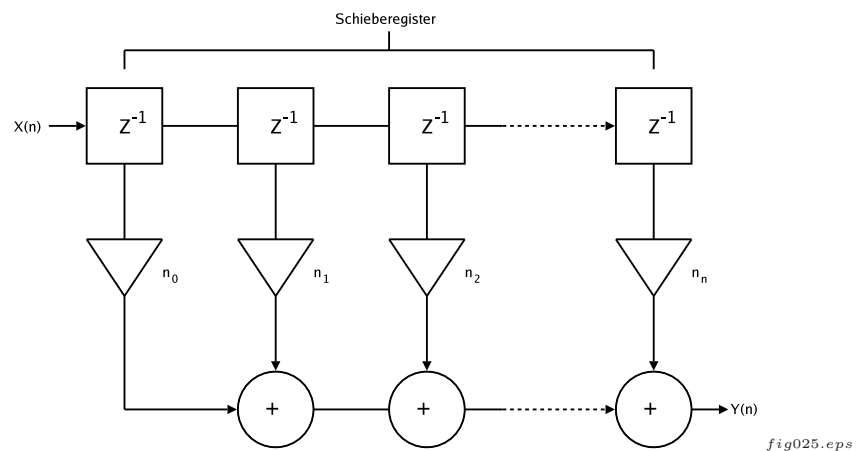
fig024.eps

➤ Sequenzielle Systeme benötigen Datenspeicher, sog. Register, um eine Evaluierung beim aktuellen Zeitpunkt mit retardierten Daten [n-1,n-2,...] zu ermöglichen.

➤ Je größer der Parameter b_1 im Tiefpaßfilter gewählt wird, desto größer ist der Einfluß von Signalwerten aus der Vergangenheit.

➤ Spektraler Filter mit höherer Ordnung $N > 1$ benötigen N Verzögerungsglieder.

Abbildung 19:
FIR-Filter (Finite-
Impulse-Response)

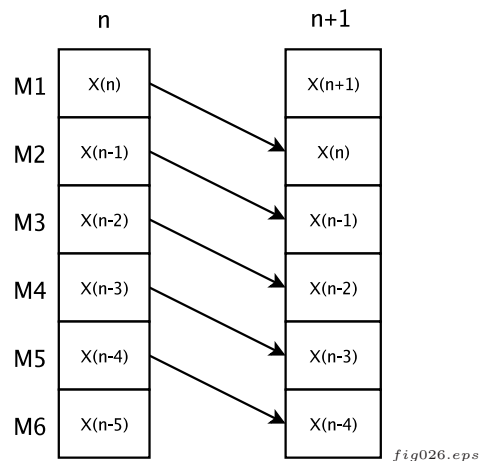


Die N Verzögerungsglieder bilden ein Schieberegister. Schieberegister lassen sich mit generischen Mikroprozessoren und imperativen Programmiersprachen nur aufwendig realisieren.

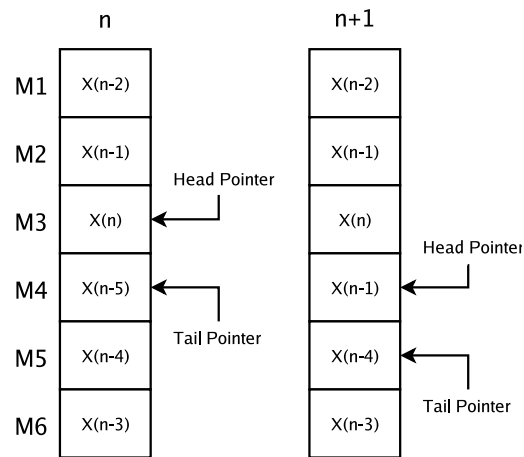
Grund: verallgemeinertes lineares Speichermodell eines von-Neumann-Rechners.

Man unterscheidet zwei verschiedene Verfahren, Schieberegister zu implementieren.

Ripple Delay Bei diesem Verfahren wird das Schieberegister linear im Speicher abgelegt. Ein Schiebezyklus erfordert $2(N-1)+1$ Speicherzugriffe, um die Daten zu verschieben.



Cyclic Delay Bei diesem Verfahren wird das Schieberegister als Ringpuffer mit einem Head- und einem Tail-Zeiger realisiert. Es werden nur noch $2+1$ Speicherzugriffe für eine Verschiebung benötigt.



► Die Realisierung eines Schieberegisters mit Digitallogik ist direkt möglich unter Verwendung von sog. D-Flip-Flop-Speichern. Ein D-Flip-Flop besitzt einen Datenein- und Ausgang. Das Datum D (1 Bit) wird bei einem Taktereignis gespeichert und am Ausgang Q ausgegeben. Nach dem Taktereignis (z.B. Wechsel des Taktsignals $0 \rightarrow 1$) ist das Ausgangssignal unabhängig vom Eingangssignal.

Abbildung 20:
Digitales
Schieberegister

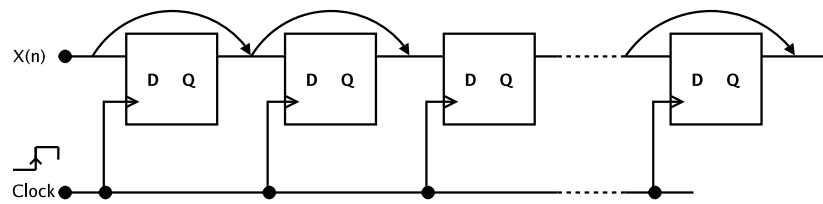


fig028.eps

► Innerhalb eines Taktzykluses kann ein N -tiefes Schieberegister ausgeführt werden. Dabei kann durch Parallelisierung von 1-Bit Schieberegistern beliebige Datenwortbreiten realisiert werden, ohne daß sich die Ausführungszeit ändert!

► Mit einer Hardware-Beschreibungssprache wie VHDL kann das Verhalten eines N -tiefen und R -Bit breiten Schieberegisters mit wenigen Anweisungen beschrieben werden:

VHDL-Beschreibung
eines
Schieberegisters

```

...
if clk'event and clk='1' then
  for i in 7 downto 1 loop
    shift_reg(i) <= shift_reg(i-1);
    shift_reg(0) <= d;
  end loop;
end if;
y <= shift_reg(7);
...

```

2.6. Vergleich generischer μ P-Systeme \leftrightarrow PLD

Mikroprozessor-Systeme erlauben die Partitionierung beim Systementwurf in:

1. Software \Rightarrow Programm orientierter Entwurf, problemspezifisch,
2. Hardware \Rightarrow Digitallogik orientierter Entwurf, generisch, unabhängig vom Problem.

Diese klassische duale Entwurfsmethode hat folgende Nachteile:

- ↳ Datendurchsatz limitiert, beschränkte Möglichkeiten der Parallelisierung von Rechenoperationen.
- ↳ Vielzahl und Breitbandigkeit von Operationen eines generischen Prozessors führt zu
 1. hoher Entwurfs- und Schaltungskomplexität,
 2. hohen elektrischen Leistungsverbrauch.

Der Vorteil eines Programm orientierten Entwurfs liegt in der Flexibilität und die Möglichkeit Fehler nachträglich zu beheben.

► Die Verlustleistung (\equiv Wärmeabgabe) ist eine Funktion der mittleren Anzahl bei einem Taktzyklus schaltenden Transistoren N und der Taktfrequenz f :

$$P \sim f \bar{N} \quad (6)$$

Bei jedem Schaltvorgang eines Logikgatters muß die Ausgangsstufe eines Logikgatters eine (parasitäre) Kapazität auf- oder entladen, entsprechend dem Übergang des Logikpegels $0 \rightarrow 1$ bzw. $1 \rightarrow 0$. Der Strom und somit die aufzuwendende Leistung hängt dann zusätzlich von der Versorgungsspannung der Transistorlogik V_{dd} ab:

$$P \sim f \bar{N} C V_{dd} \quad (7)$$

Reduzierung der Verlustleistung einer Digitallogikschaltung durch:

1. Reduzierung der Taktfrequenz,
2. Reduzierung der pro Taktzyklus im Mittel schaltenden Transistoren (Stilllegung von unbenutzten Teilen in einem synchronen Logiksystem oder Verwendung von asynchroner Digitallogik),
3. Reduzierung der Versorgungsspannung.

Elektrische
Verlustleistung
(CMOS-Technologie)

Parallelisierung

► Verarbeitung von hohen Datenraten- und Mengen durch parallele Datenverarbeitung führt zu niedrigen Latenzzeiten, d.h. die Zeit die benötigt wird, um einen Datensatz zu verarbeiten.

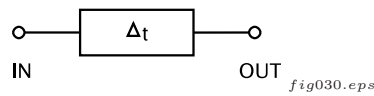
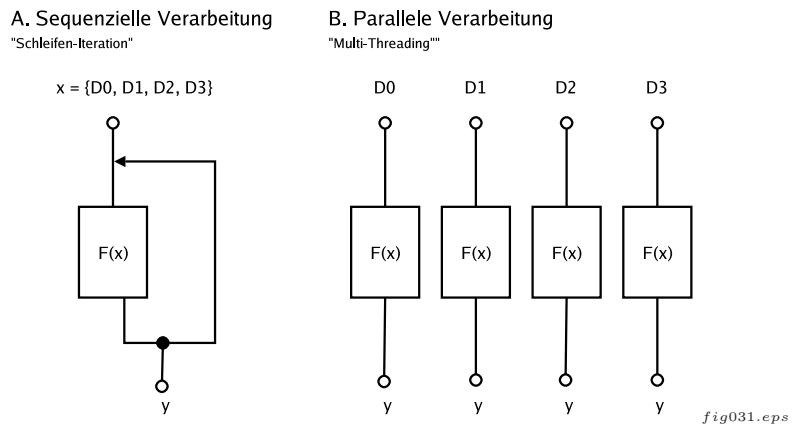


Abbildung 21: Definition der Latenz Δt .

➔ Es findet ein Übergang von der Zeit- auf die Flächendimension statt:

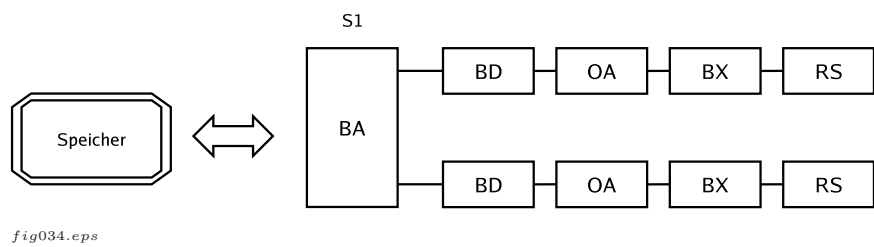
Abbildung 22: Übergang von sequenzieller zu paralleler Datenverarbeitung.



Instruction-Pipelines

➤ Neben Parallelität in anwendungsspezifischer Digitallogik kann Parallelität auch auf der ISA/Mikroarchitektur-Ebene von Mikroprozessorsystemen umgesetzt werden. Dazu wird die Bearbeitungskette eines Mikroprozessors, bestehend aus Befehlsabrufeinheit BA, dem Befehlsdeko-der BD, der Operandenabrufeinheit OA, der Befehlsausführung BX (ALU) und der Rückschreibeinheit RS, parallelisiert: die Instruction-Pipeline.

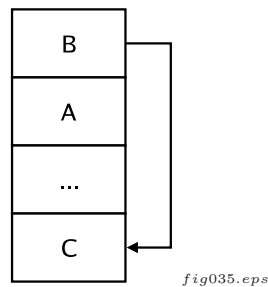
Abbildung 23: Zwei fünfstufige Pipelines mit gemeinsamer Befehlsabrufeinheit.



➤ **Problem:**

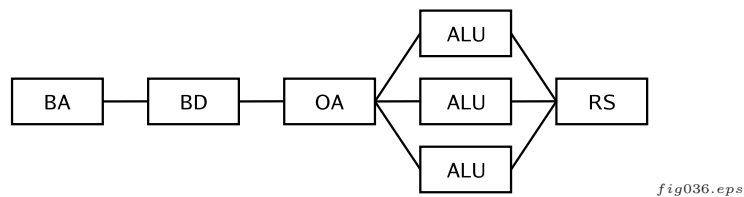
Die einzelnen Pipelines müssen synchronisiert werden und beeinflussen sich gegenseitig, wie nachfolgendes Beispiel einer bedingten Verzweigung verdeutlichen soll. Pipeline P1 enthält die Auswertung einer booleschen Bedingung B, die über einen Sprungbefehl entscheidet, und Pipeline P2 führt einer arithmetische Operation A durch. Die Auswertung von B ergibt, daß A nicht ausgeführt werden darf.

Abbildung 24:
Bedingter
Sprungbefehl (B) und
arithmetische
Operation (A).



- ↳ Beide Befehle sind tief in der Pipeline ausgeführt.
 - ↳ Bedingter Sprung evaluiert zu Sprung nach C
 - ↳ flush (P2)
 - ↳ Ausführung in P2 muß abgebrochen werden
 - ↳ Pipeline P2 muß reinitialisiert werden - Bruch im Programmfluß.
- Synchronisation ist komplex und aufwendig!
➤ Daher häufig Reduzierung der Parallelität, hier durch superskalare Prozessorarchitektur.

Abbildung 25:
Superskalare
Pipeline mit
Parallelität nur in der
Ausführungseinheit
(ALU).



- ↳ Resultat ist deutlich geringerer Entwurfs- und Hardwareaufwand!
- Hardware-Entwurf ist immer Kompromiss zwischen:
1. **Parallelität** ⇔ **Entwurfskomplexität**
 2. **Parallelität** ⇔ **Flexibilität**
 3. **Programmierbarkeit** ⇔ **Effizienz**
- Als Kompromiß-Lösung bieten sich sog. erweiterbare Prozessoren an:
- μ P-Kern ist vorgegeben, kann aber erweitert werden durch {Spezielle Befehle, z.B. $N \times R$ -Schieberegister, Register (Anzahl und Datenbreite), Funktionen/Operatoren}
 - μ P-Peripherie kann anwendungsspezifisch entworfen und hinzugefügt werden (Anzahl, Funktion).
 - Vorteile: Compiler existieren, μ P-Kern ist verifiziert, Flexibilität durch Software-Entwurf.

Hardware-Entwurf

Erweiterbare
Mikroprozessoren

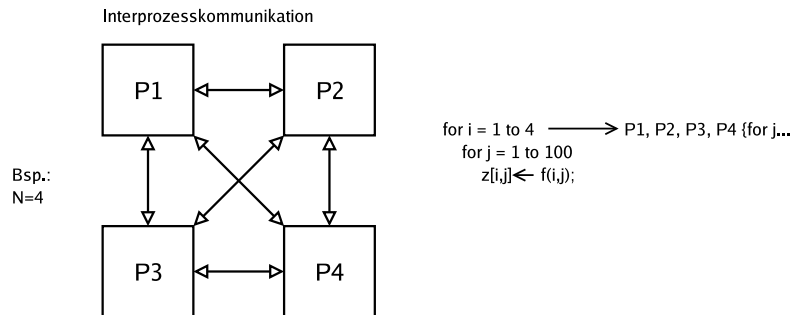
**Konfigurierbare
Mikroprozessoren**

- Nachteile: immer noch generischer Ansatz mit obigen Nachteilen.
- Als Kompromiß-Lösung bieten sich weiterhin sog. vollständig konfigurierbare Prozessoren an:
 - μ P-Kern ist frei definierbar, kann aus Problemspezifikation abgeleitet werden:
{Spezielle und generische Befehle, z.B. $N \times R$ -Schieberegister, allgemeine und spezielle Register (Anzahl und Datenbreite), Funktionen/Operatoren}
 - μ P-Peripherie kann anwendungsspezifisch entworfen und hinzugefügt werden (Anzahl, Funktion).
 - Vorteile: Flexibilität durch optimale Anpassung des Mikroprozessors an Problemspezifikation
 - Nachteile: Compiler existieren nicht, μ P-Kern ist nicht verifiziert.

2.7. Parallelität durch Multi- μ P-Systeme

Partitionierung eines Algorithmus und Programms in mehrere nebeneinanderläufige Einheiten, die auf N Prozessoren parallel ausgeführt werden.

Abbildung 26:
Parallelisierung eines Algorithmus (hier verschachtelte Schleife)



Die einzelnen Prozessoren führen Teilprozesse aus. Ein wichtiger Bestandteil bei der Parallelisierung auf Prozeßebene ist die Interprozeßkommunikation die zwei Funktionen erfüllt:

1. Datenaustausch (Input- und Output-Daten)
2. Synchronisation von einzelnen Prozessoren (Schutz von kritischen Programmbereichen, z.B. gemeinsame genutzte Datenstrukturen, Abhängigkeiten der Instruktionsreihenfolge usw.)

Es gibt zwei verschiedene Hardware/Software-Lösungen für eine N-Processor-Realisierung:

Shared Memory Systems Die einzelnen Mikroprozessoren teilen sich gemeinsamen Hauptspeicher.

- ⊖ Gemeinsamer Speicherbus ist Engpaß
- ⊕ geringer Kommunikationsoverhead der Prozeßsynchronisation, Datenaustausch zwischen einzelnen Mikroprozessoren durch Speicherzeiger.

Distributed Memory Systems Hier werden N unabhängige Rechensysteme mit jeweils eigenen Hauptspeicher über ein Netzwerk miteinander zu einer virtuellen Maschine gekoppelt.

- ⊕ Lokal maximale Rechenleistung und maximaler Datendurchsatz, aber
- ⊖ hoher Kommunikationsoverhead der Prozeßsynchronisation und geringerer Datendurchsatz bei Datenaustausch zwischen einzelnen Knoten.

2.8. Kontroll- und Datenpfade

Jeder generische Mikroprozessor und i.A. jedes anwendungsspezifische Digitallogiksystem läßt sich in zwei funktionale Bereiche aufteilen:

1. Datenfluß \Rightarrow Datenpfade
2. Kontrollfluß \Rightarrow Zustandsautomat

Beiden Bereiche bilden einen sog. Systemblock. Ein Gesamtsystem kann in eine Vielzahl von Systemblöcken partitioniert werden.

► Der Datenpfad enthält:

1. Arithmetische, logische und boolesche Funktionsblöcke (Operatoren),
2. Datenregister zur Speicherung von Daten, z.B. für Zwischenergebnisse von arithmetischen Operationen,
3. Multiplexer zur Steuerung des Datenflusses,
4. Speicherblöcke (RAM) für den Datenaustausch zwischen verschiedenen Systemblöcken:

Operanden \Rightarrow Ergebnisse \Rightarrow RAM

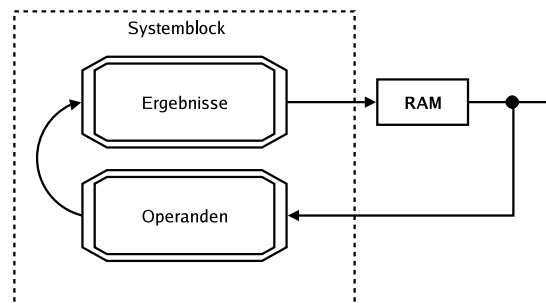


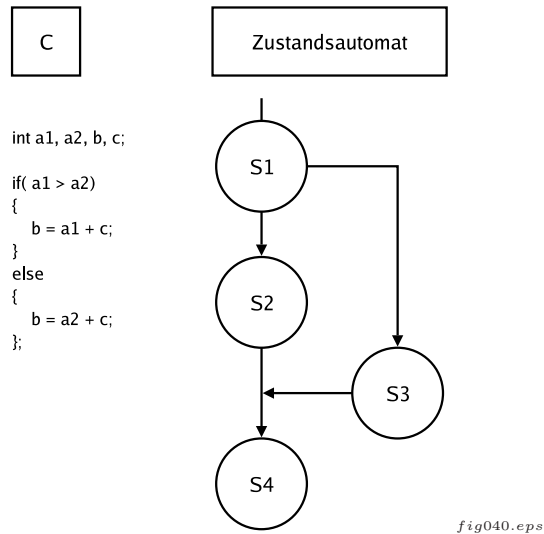
fig036c.eps

► Der Datenpfad belegt größten Anteil von Logikgattern, der Zustandsautomat den kleinsten Anteil.

► Der Zustandsautomat ist für die Ablaufsteuerung in einem Systemblock zuständig und ist zentraler Bestandteil. Taktgesteuert findet der Übergang zwischen verschiedenen Zuständen des Systems statt.

1. Der Zustandsautomat steuert den Datenfluß im Datenpfad,
2. er ist für die Implementierung von Handshake-Protokollen für den Datenaustausch mit anderen Systemblöcken zuständig,
3. und er behandelt Ausnahmesignale und Fehler.

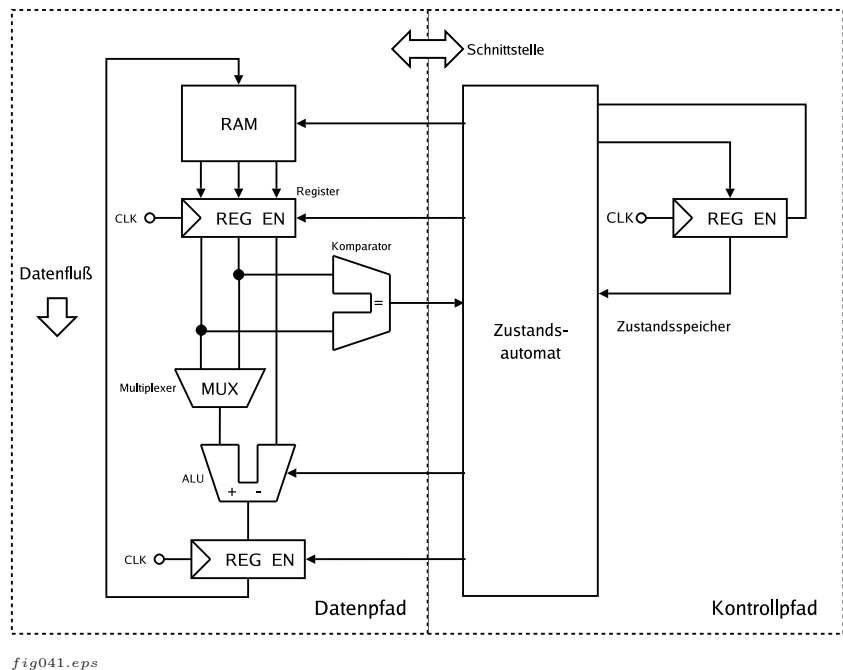
Abbildung 27:
Beispiel
Zustandsdiagramm
und Ablaufsteuerung



Register-Transfer-Logik (RTL)

► Daten- und Kontrollpfade werden mit der sog. Register-Transfer-Logik implementiert. Bei der RTL findet eine schrittweise und taktgesteuerte Bearbeitung des Datenflusses mittels Registern statt.

Abbildung 28:
Beispiel RTL



3. Einführung in die Digitaltechnik

Dieses Modul bietet einen Zugang zur Digitaltechnik über mathematische Boolesche Algebra und technologischer Digitallogik.

3.1. Logische Grundfunktionen

Logische Variablen besitzen Wertemenge $\{0, 1\}$. Die technologische Umsetzung und Implementierung von logischen Zuständen $\{0,1\}$ findet i.A. durch elektronische Schaltungstechnik statt.

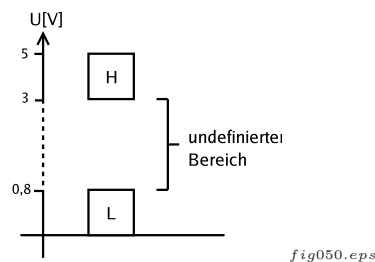
► Logische Funktionen werden mit Funktions- oder Wahrheitstabellen beschrieben, die alle Kombinationen von Logikwerten der Eingangsvariablen auf ein oder mehrere Ausgangswerte abbilden.

► Den logischen Zuständen werden i.A. zwei verschiedene Spannungspegel zugeordnet, deren Werte abhängig von der Schaltungstechnologie sind. Es werden keine festen Spannungswerte sondern Spannungsbereiche (Intervalle) verwendet, z.B. für die TTL-Technologie, die mit einer Versorgungsspannung von 5V betrieben wird:

$0 \rightarrow L \rightarrow 0 \dots 0,8 \text{ V}$

$1 \rightarrow H \rightarrow 3 \dots 5 \text{ V}$

Abbildung 29:
TTL-Logikpegel



► Der Grund von Spannungsintervallen liegt in einem möglichst großen Störabstand begründet, d.h. Immunität gegen Störungen, da digitale Spannungssignale bei der Technologieumsetzung tatsächlich als analoge Signale auftreten, d.h. wert- und zeitkontinuierliche Signale. Ein nicht vermeidbares Phänomen, das Signalrauschen, welches physikalisch bedingt ist, führt immer zu einer Unsicherheit des Spannungspegels von digitalen Signalen.

Schaltungstechnologien

► Es gibt verschiedene Schaltungstechnologien, mit denen Digitallogikschaltungen auf Transistorebene realisiert werden können.

TTL Transistor-Transistor-Logic → bipolare Transistortechnik mit folgenden Eigenschaften:

- Stromgesteuerte Stromquellen
- Spannungsversorgung: 5V
- Moderate Verlustleistung auch ohne Schaltaktivität.
- Schaltgeschwindigkeiten im Bereich von 5ns

CMOS Complementary Metall Oxide Substrate → Feldeffekt-Transistortechnik mit folgenden Eigenschaften:

- Spannungsgesteuerte Stromquellen
- Spannungsversorgung: 1-15V
- Geringe statische Verlustleistung, geringe dynamische Verlustleistung bei Schaltaktivität.
- Schaltgeschwindigkeiten technologieabhängig, im Bereich 1-10ns
- Heute dominierender Technologieprozeß.

ECL Emitter-Coupled-Logic → bipolare Transistortechnik mit folgenden Eigenschaften:

- Stromgesteuerte Stromquellen
- Spannungsversorgung: NECL → -5V
- Hohe Verlustleistung auch ohne Schaltaktivität.
- Sehr hohe Schaltgeschwindigkeit ≤ 1 ns

Logische Funktion:
Negation →
Inverter

► Logische Negierung einer Eingangsvariablen.

$$f(x) = \bar{x} = \neg x \quad (8)$$

x	\bar{x}
0	1
1	0

Tabelle 1: Negation: Funktionstabelle

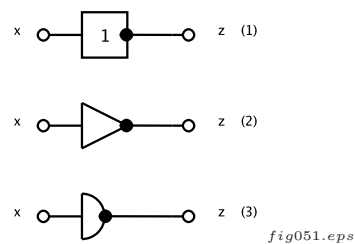


Abbildung 30: Negation: Schaltsymbole, (1) → ISO, (2) → Amerika, (3) → alt

Abbildung 31:
Inverter: CMOS
Transistorschaltung.

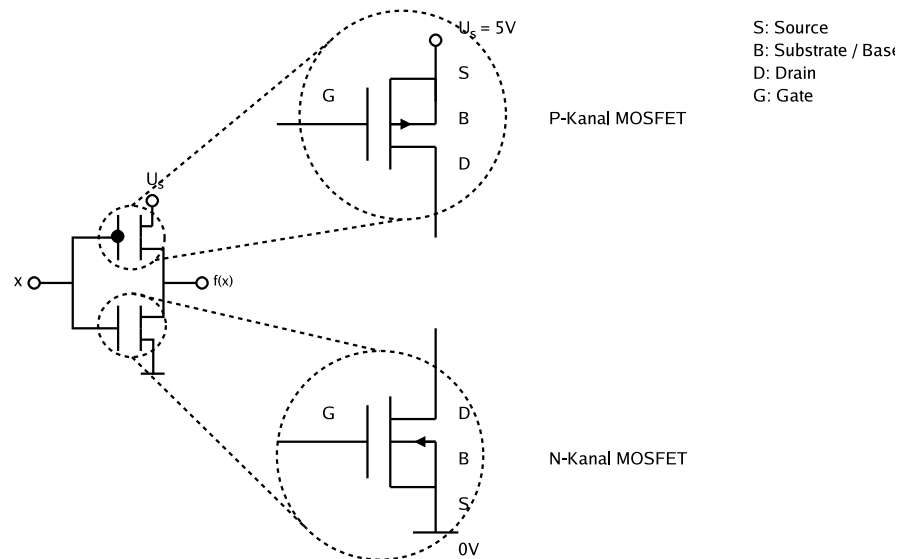


fig052.eps

► Die Transistorschaltung besteht aus einem sog. N-Kanal (unten) und dazu im Verhalten komplementären P-Kanal (oben) MOS-Feldeffekttransistor, mit selbstsperrendem Verhalten. Weitere elektronische Bauelemente sind zur Implementierung im Gegensatz zu der Bipolartransistortechnik nicht erforderlich. Das in der Digitaltechnik gewünschte Schaltverhalten $\{0,1\}$ ergibt sich aus dem analogen Übertragungsverhalten, d.h. der Kennlinie eines N-/P-MOSFET-Transistors.

► Ein FET-Transistor besitzt drei Anschlüsse:

Source S → Dieser Anschluß ist als Ladungslieferant zu verstehen, d.h. der Quelle für elektrische Ladungsträger, den Elektronen (neg.), oder den sog. Löchern (pos.).

Drain D → Gegenüber der Ladungsquelle befindet sich die 'Ladungsenke', über den ein Fluß von Ladungsträgern stattfinden kann.

Gate G → Der Gate-Anschluß beeinflusst den Ladungstransport zwischen Source und Drain-Anschluß, und ermöglicht eine spannungsgesteuerte Stromquelle.

Ein sog. selbstsperrender Transistor ist dadurch gekennzeichnet, daß bei einer Gate-Source-Spannung $U_{GS}=0V$ kein Drain-Strom fließt, man spricht von einem sperrenden Transistorzustand.

Der andere Zustand eines Transistors ist der leitende Zustand, bei dem ein elektrischer Strom zwischen Source und Drain-Anschluß I_{DS} fließen kann.

Abbildung 32:
Kennlinie eines N- und P-Kanal-MOSFET Transistors.

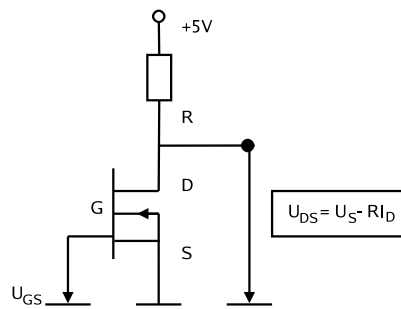
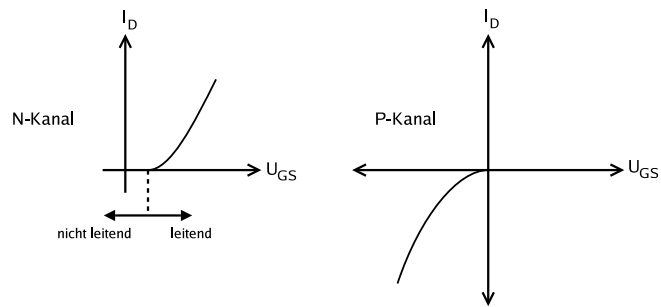


fig053.eps

Neben den drei Anschlüssen {S,G,D} gibt einen sog. Substrat-Anschluß, der mit einem fixen Potential verbunden ist. Aus Gründen der Übersichtlichkeit verwendet man vereinfachte elektronische Transistorsymbole, die im folgenden ausschließlich verwendet werden.

Abbildung 33:
Elektronische Schaltsymbole von MOSFET Transistoren.

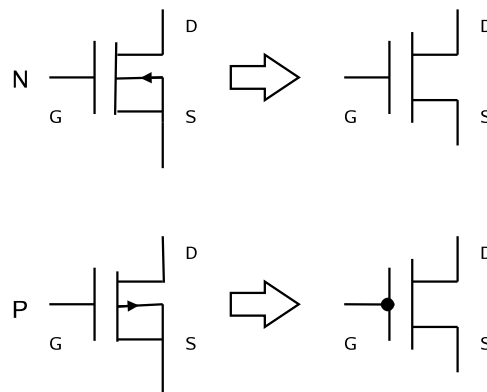


fig054.eps

Logische Funktion:
Und-verknüpfung
→ **Konjunktion**

► Logische Verknüpfung zweier Eingangsvariablen.

$$f(x, y) = x \wedge y = x \bullet y \tag{9}$$

x	y	f(x,y)
0	0	0
1	0	0
0	1	0
1	1	1

Tabelle 2: Konjunktion: Funktionstabelle

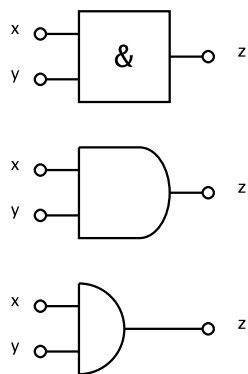


fig055.eps

Abbildung 34: Konjunktion: Schaltsymbole, (1) → ISO, (2) → Amerika, (3) → alt

Abbildung 35:
NAND: CMOS
Transistorschaltung.

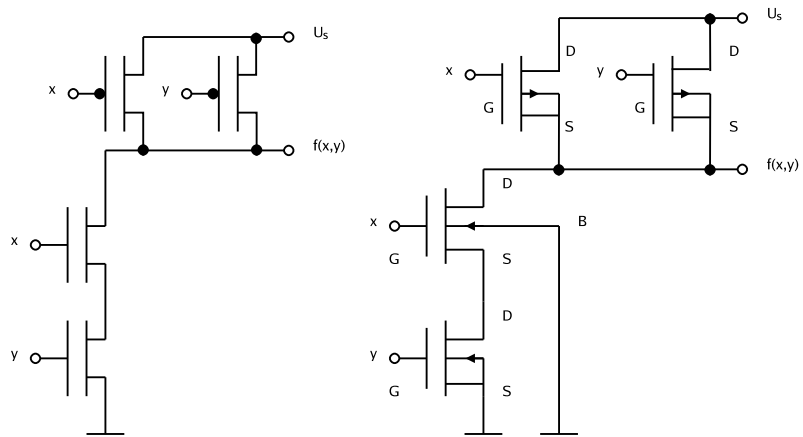


fig056.eps

Logische Funktion:
Oder-Verknüpfung
→ Disjunktion

► Logische Verknüpfung zweier Eingangsvariablen.

$$f(x, y) = x \vee y = x + y \tag{10}$$

x	y	f(x,y)
0	0	0
1	0	1
0	1	1
1	1	1

Tabelle 3: Disjunktion: Funktionstabelle

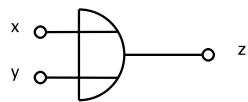
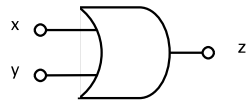
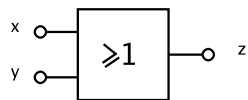


fig057.eps

Abbildung 36: Disjunktion: Schaltsymbole, (1) → ISO, (2) → Amerika, (3) → alt

Abbildung 37:
NOR: CMOS
Transistorschaltung.

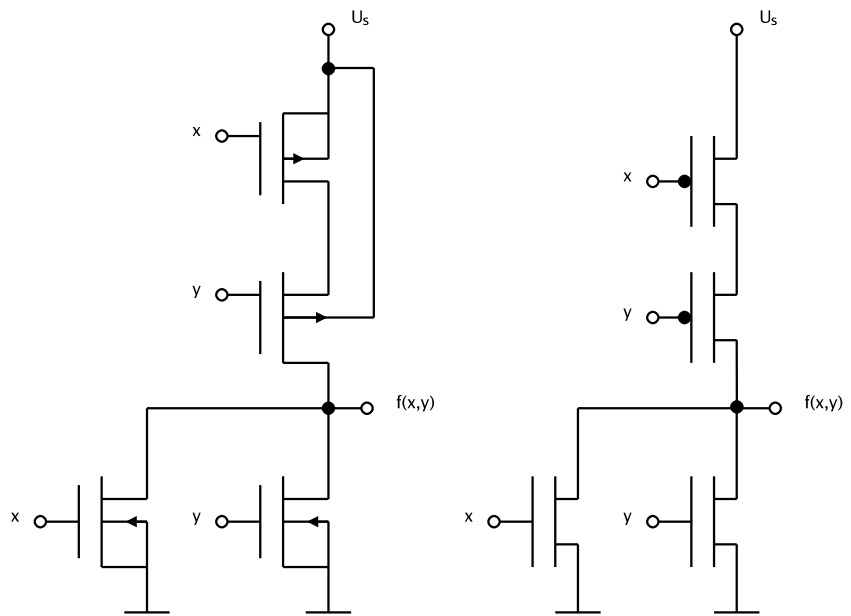


fig058.eps

**Logische Funktion:
Exklusiv-Oder-
verknüpfung**

► Logische Verknüpfung zweier Eingangsvariablen.

$$f(x, y) = (x \wedge \neg y) \vee (\neg x \wedge y) = x \bullet \neg y + \neg x \bullet y = x \oplus y \quad (11)$$

x	y	f(x,y)
0	0	0
1	0	1
0	1	1
1	1	0

Tabelle 4: EXOR: Funktionstabelle

3.2. Boolesche Algebra

Systeme logischer Variablen sind über logische Funktionen verknüpft. Die Funktion einer digitallogischen Schaltung, deren Ausgangswerte nur von den aktuellen Eingangswerten abhängen, wird durch boolesche Algebra beschrieben.

Die Boolesche Algebra besteht aus drei Operationen:

Disjunktion Oder-Verknüpfung von n Eingangswerten zu einem Ausgangswert,

$$a = e_1 + e_2 + e_3 + \dots + e_n$$

Konjunktion Und-Verknüpfung von n Eingangswerten zu einem Ausgangswert,

$$a = e_1 \bullet e_2 \bullet e_3 \bullet \dots \bullet e_n$$

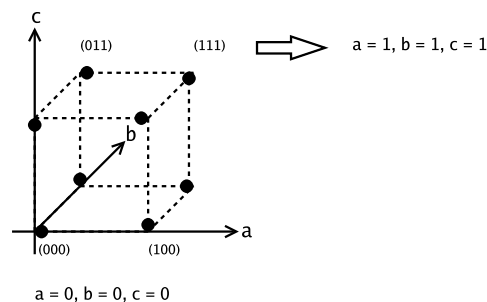
Negation Invertierung eines booleschen Zustandes $\neg e$

- Boolesche Werte besitzen eine Wertemenge $\{0,1\}$.
- Boolesche Funktionen bilden n Variablen (n -dimensionaler Vektor) auf m Ergebniswerte (m -dimensionaler Vektor) ab:

$$f : B^n \rightarrow B^m \quad (12)$$

I.A. ist $m=1$, d.h. Verwendung von skalaren booleschen Funktionen. Die Eingangsvektoren können graphisch bis $n \leq 3$ dargestellt werden, z.B. für $n=3$ und den 3 Eingangsvariablen a, b, c , die jeweils eine Achse eines orthogonalen Koordinatensystems bezeichnen:

Abbildung 38:
Graphische
Darstellung eines
Booleschen Vektors
(a, b, c)



Ein Vektor (a, b, c) entspricht dann genau einem Punkt im 3-dimensionalen booleschen Zustandsraum.

- Boolesche Algebra ist Hilfsmittel beim Entwurf von digitalen Schaltungen. Eine Aufgabenstellung definiert Schaltbedingungen, die in einer Funktions-/Wahrheitstabelle dargestellt werden. Diese Schaltbedingungen können auch durch boolesche Funktionen dargestellt werden, die aus der Funktionstabelle abgeleitet werden.
- Die so gewonnenen Funktionen werden mittels Gesetzen der Booleschen Algebra umgeformt und vereinfacht, so daß eine technische Realisierung mit minimalen Aufwand erfolgen kann.

Normalformen

► Aus Funktionstabellen werden im ersten Entwurfsschritt sog. Normalformen von logischen Funktionen abgeleitet. Man unterscheidet:

Disjunktive Normalform Eine Summe aus Produkttermen (SOP)

Konjunktive Normalform Ein Produkt aus Summentermen (POS)

Disjunktive Normalform SOP

► Jeder Teilterm besteht aus einer Und-Verknüpfung der Eingangsvariablen, die entweder negiert oder nicht negiert im Term auftreten. Alle Teilterme werden Oder-verknüpft und ergeben die boolesche Funktion:

$$f(a_1, a_2, \dots, a_n) = (x_1^1 \bullet x_2^1 \bullet \dots \bullet x_n^1) + (x_1^2 \bullet x_2^2 \bullet \dots \bullet x_n^2) + \dots \quad (13)$$

mit $x_i = \{x_i, \neg x_i\}$.

↳ Ableitung aus Funktionstabelle:

a ₁	a ₂	q
x	x	1 → Teilterm
x	x	0

Tabelle 5: Ableitung Normalform SOP

1. Nur Zeilen in der Funktionstabelle bei denen die Ausgangsvariable logisch 1 ist, führen zu einem Teilterm in der SOP-Normalform.
2. Alle Eingangsvariablen werden Und-verknüpft und negiert, wenn die Eingangsvariable den Wert 0 besitzt.

A	B	Q
0	1	1 → ($\neg A \bullet B$)
1	0	1 → ($A \bullet \neg B$)
0	0	0
1	1	0

Tabelle 6: Beispiel Normalform SOP → $Q = (\neg A \bullet B) + (A \bullet \neg B)$

Die boolesche Funktion Q führt zu folgender Digitallogikschaltung.

**Konjunktive
Normalform POS**

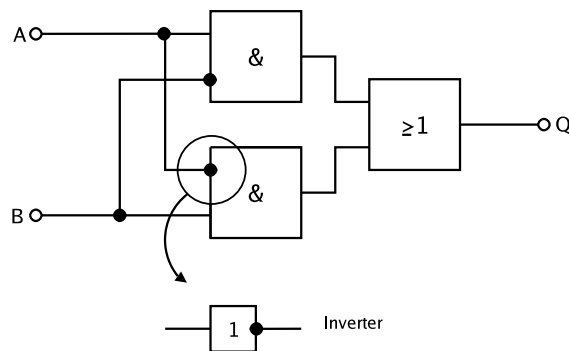


fig061.eps

Abbildung 39: $Q = (\neg A \bullet B) + (A \bullet \neg B)$

► Jeder Teilterm besteht aus einer Oder-Verknüpfung der Eingangsvariablen, die entweder negiert oder nicht negiert im Term auftreten. Alle Teilterme werden Und-verknüpft und ergeben die boolesche Funktion:

$$f(a_1, a_2, \dots, a_n) = (x_1^1 + x_2^1 + \dots + x_n^1) \bullet (x_1^2 + x_2^2 + \dots + x_n^2) \bullet \dots \quad (14)$$

mit $x_i = \{x_i, \neg x_i\}$.

➔ Ableitung aus Funktionstabelle:

a ₁	a ₂	q
x	x	0 → Teilterm
x	x	1

Tabelle 7: Ableitung Normalform POS

1. Nur Zeilen in der Funktionstabelle bei denen die Ausgangsvariable logisch 0 ist, führen zu einem Teilterm in der POS-Normalform.
2. Alle Eingangsvariablen werden Oder-verknüpft und negiert, wenn die Eingangsvariable den Wert 1 besitzt.

A	B	Q
1	1	0 → (¬A+¬B)
0	0	0 → (A+B)
1	0	1
0	1	1

Tabelle 8: Beispiel Normalform POS → $Q = (\neg A + \neg B) \bullet (A + B)$

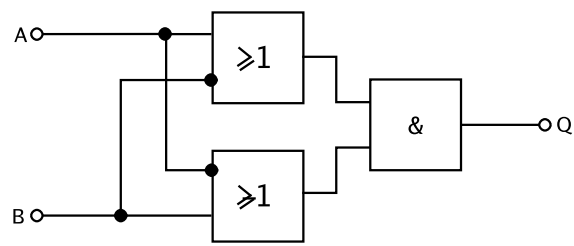


Abbildung 40: $Q = (\neg A + \neg B) \cdot (A + B)$

- ▶ Beide Normalformen sind i.A. noch redundant, d.h. sie können vereinfacht werden.
- ▶ Beide Normalformen können ineinander transformiert werden.
- ▶ Die disjunktive Normalform SOP liefert kurze Gleichungen, wenn die Ausgangsvariable nur in wenigen Fällen logisch 1 ist, und die konjunktive beim Wert 0.

Kommutativ-Gesetze mit 2 Variablen
Assoziativ-Gesetze mit 3 Variablen
Distributiv-Gesetze mit 3 Variablen
Inversions-Gesetze mit 2 Variablen

3.3. Termumformungen

Mittels der booleschen Termumformungen können boolesche Ausdrücke mit folgenden Zielen umgeformt werden :

- Reduzierung der Verknüpfungen,
- Transformation auf eine bestimmte Technologie, z.B. nur Nicht-Oder-Verknüpfungen.

► Die Variablen sind vertauschbar, die Eingänge von Und- bzw. Oder-Gattern können vertauscht werden.

$$\begin{aligned}x \bullet y &= y \bullet x \\ x + y &= y + x\end{aligned}\quad (15)$$

► Reihenfolge der Berechnung ist beliebig, die Zusammenfassung zweier Eingänge von Gattern ist beliebig.

$$\begin{aligned}(x \bullet y) \bullet z &= x \bullet (y \bullet z) = x \bullet y \bullet z \\ (x + y) + z &= x + (y + z) = x + y + z\end{aligned}\quad (16)$$

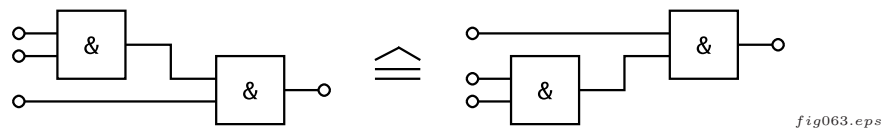


Abbildung 41: Beispiel

► Eine gemeinsame Variable in zwei verknüpften Termen kann ausgeklammert werden.

$$\begin{aligned}(x \bullet y) + (x \bullet z) &= x \bullet (y + z) \\ (x + y) \bullet (x + z) &= x + (y \bullet z)\end{aligned}\quad (17)$$

Die zweite Gleichung kennt keine Analogie in der gewöhnlichen Algebra!

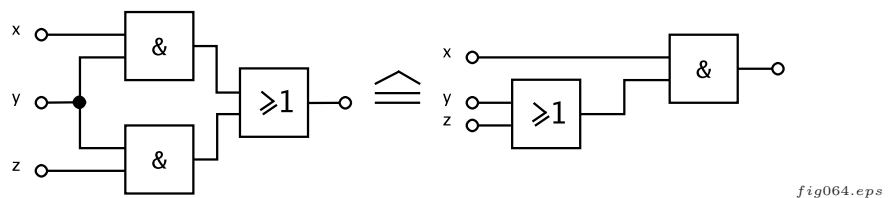


Abbildung 42: Beispiel für eine 3→2 Logikgatterminimierung.

► Transformation von Und- nach Oder-Verknüpfung und umgekehrt; Negierung wandert von Eingängen zum Ausgang. Wichtig für Technologieumsetzung!

$$\begin{aligned}(\bar{x} \bullet \bar{y}) &= \bar{(x + y)} \\ (\bar{x} + \bar{y}) &= \bar{(x \bullet y)}\end{aligned}\tag{18}$$

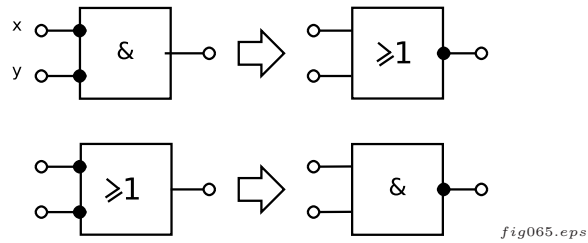


Abbildung 43: De-Morgansche Regeln

Bindungsregeln



1. Negation von Variablen wird stets zuerst evaluiert.
2. Und-Verknüpfung bindet stärker als Oder-Verknüpfung.
3. Alle anderen Verknüpfungen werden von links nach rechts evaluiert.

3.4. Logische und technische Zustände

Die boolesche Algebra kennt nur die zweiwertige Menge $\{0,1\}$ zur Zustandsbeschreibung eines digitalen Wertes. Bei der technologischen Umsetzung können mehrwertige Logikzustände auftreten:

- 0** → Starker logischer Wert False. Ein Signal mit diesem Wert darf mit keinem anderen Signal überlagert bzw. zusammengeschlossen werden (elektrisch: Kurzschluß!).
- 1** → Starker logischer Wert True. Ein Signal mit diesem Wert darf mit keinem anderen Signal überlagert bzw. zusammengeschlossen werden (elektrisch: Kurzschluß!).
- L** → Schwacher logischer Wert False. Schwache Signale können überlagert und mittels einer Auflösungsfunktion einen summierten Wert bilden.
- H** → Schwacher logischer Wert True. Schwache Signale können überlagert und mittels einer Auflösungsfunktion einen summierten Wert bilden.
- Z** → Hochohmiger Zustand. Mehrere Signale können überlagert und einen gemeinsamen Bus bilden. Der Zustand Z entkoppelt den schreibenden (treibenden) Zugriff eines Kommunikationsteilnehmers von einer Signalleitung. Die Signalleitung kann bidirektional verwendet werden. Lesender Zugriff ist aber jederzeit möglich.
- X** → Logischer Zustand "Don't-Care". Bei der Evaluierung von booleschen Funktionen können diese Zustände ignoriert werden, mit der Möglichkeit der Logikoptimierung.

3.5. Systematische Reduktion logischer Funktionen

Ziel der Minimierung: möglichst kleine Anzahl von Logikgatter-Komponenten bei der elektronischen Implementierung bei gleichzeitiger geringer Signallaufzeit, zwei gegenläufige Ziele!

Man unterscheidet vier grundlegenden Verfahren:

1. Minimierung mittels Gesetzen der Booleschen Algebra - nicht systematisch.
2. Karnaugh-Veitch-(KV) Diagramme, beschränkt auf kleine Anzahl von Funktionsvariablen.
3. Quine-McCluskey-Verfahren, systematisch, für mittlere Anzahl von Funktionsvariablen geeignet.
4. Binary-Decision-Diagrams (BDD), häufig in Synthese-Programmen verwendet.

KV-Diagramme

- Die KV-Diagramm-Methode ist anschaulich und intuitiv, und soll als Einstieg verstanden werden.
- Es findet eine Darstellung der Funktionstabelle in Zeilen und Spalten eines Diagramms statt.
- Dabei sind die Diagrammfelder so angeordnet, daß sich bei einem Übergang von einem zu einem anderen Feld immer nur eine Variable ändert.
- Es existieren unterschiedliche Diagramme für DNF (SOP) und KNF (POS) Darstellungen.

Abbildung 44:
KV-Diagramm für 4
Variablen A,B,C,D
unf DNF.

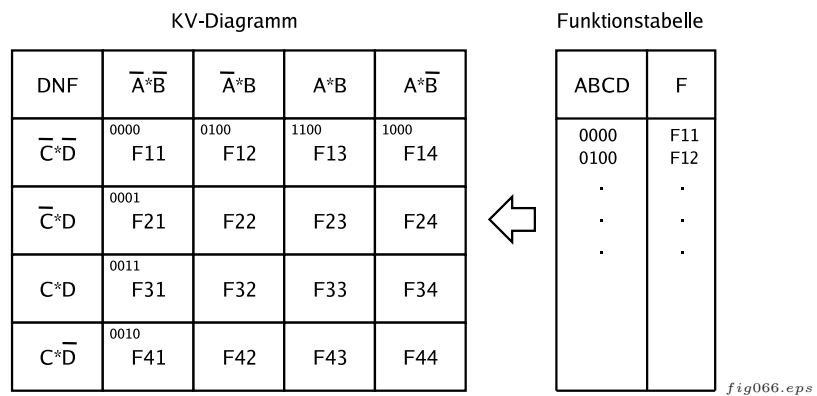
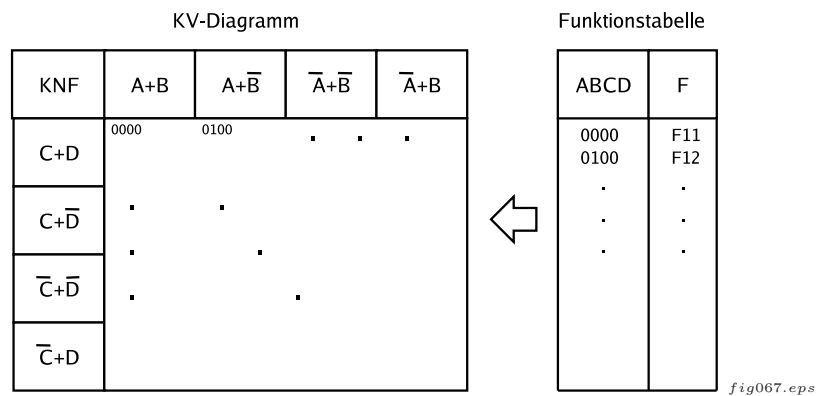


Abbildung 45:
KV-Diagramm für 4
Variablen A,B,C,D
unf KNF.



- In jedem Feld wird der zu den Werten der Eingangsvariablen gehörende Ausganzzustand/wert $F_{ij}=\{0,1\}$ eingetragen.
- Das Diagramm ist zyklisch: der Übergang von rechten Rand zum linken, und von unteren zum oberen ist möglich $\Rightarrow \cong$ Torusoberfläche
- Bei drei Eingangsvariablen reduziert sich das Diagramm um zwei Zeilen.

**Reduzierte
disjunktive
Normalform**

► Folgende Schritte sind zur Ableitung einer minimalen d.h. reduzierten DNF (RDNF) notwendig:

1. Benachbarte Felder $F_{ij}=1$ werden zu Flächen mit 2^N Elementen zusammengefasst. Die größtmöglichen Flächen/Gruppen sollen gebildet werden.
2. Alle Felder müssen in mindestens einer Fläche/Gruppe erfasst werden.
3. Ableitung eines neuen Teilterms der RDNF aus einer Fläche/Gruppe: Produkt aus allen Variablen, die allen Feldern der Gruppe gemeinsam sind.
4. Die Teilterme werden summiert.

Abbildung 46:
Beispiel: KV-DNF

DNF	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{C}\bar{D}$	1	1	0	0
$\bar{C}D$	1	1	0	0
CD	0	0	1	1
$C\bar{D}$	1	0	0	1

fig068.eps

Die reduzierte DNF folgt dann:

$$f'(A, B, C, D) = \bar{A} \cdot \bar{C} + A \cdot C \cdot D + \bar{B} \cdot C \cdot \bar{D} \quad (19)$$

Es ist eine Reduktion von ursprünglich 49 booleschen Operationen und 8 Teiltermen auf 11 Operationen und 3 Teiltermen erfolgt! Nimmt man im Mittel 4 CMOS-Transistoren je boolescher Operation an, ergibt sich eine Verringerung der Transistoren um $196 \rightarrow 44$ und einer Verringerung der Chip-Fläche um den Faktor 2.11!

**Reduzierte
konjunktive
Normalform**

► Folgende Schritte sind zur Ableitung einer minimalen d.h. reduzierten KNF (RKNF) notwendig:

1. Benachbarte Felder $F_{ij}=0$ werden zu Flächen mit 2^N Elementen zusammengefasst. Die größtmöglichen Flächen/Gruppen sollen gebildet werden.
2. Alle Felder müssen in mindestens einer Fläche/Gruppe erfasst werden.
3. Ableitung eines neuen Teilterms der RKNF aus einer Fläche/Gruppe: Summe aus allen Variablen, die allen Feldern der Gruppe gemeinsam sind.
4. Die Teilterme werden multipliziert.

Verfahren nach Quine und McCluskey

► Die QM-Methode ist formaler als das vorherige Diagrammverfahren. Zum Verständnis einige Definitionen:

Minterm → Produkterme (der DNF) werden als Minterme bezeichnet, wenn jede Variable einmal auftritt. Bei der vollständigen DNF ist jeder Produktterm ein Minterm.

$$\text{Beispiel: } Q = A \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot B \cdot D$$

→ Erste Term ist ein Minterm.

Maxterm → Summenterme (der KNF) werden als Maxterme bezeichnet, wenn jede Variable einmal auftritt. Bei der vollständigen KNF ist jeder Summenterm ein Maxterm.

Implikant → Ein Produktterm P heißt Implikant der booleschen Funktion Q, wenn aus $P=1 \Rightarrow Q=1$ folgt. Jeder Produktterm der DNF ist ein Implikant.

Beispiel: sowohl $A \cdot \bar{B} \cdot C \cdot \bar{D}$ als auch $A \cdot B \cdot D$ sind Implikanten von Q.

Primimplikant (Primterm) → Term, der nach Weglassen einer Variablen kein Implikant mehr wäre (kürzester Implikant).

$$\text{Beispiel: } Q = A \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot B \cdot D + \bar{A} \cdot \bar{B} \cdot C \cdot \bar{D}$$

→ Erste Term ist kein PI.

→ Zweiter Term ist ein PI, da $A \cdot B$ oder $B \cdot C$ oder $A \cdot C = 1$ nicht ausreichen, damit $Q=1$ wird.

→ Dritter Term kein PI, da $\bar{B} \cdot C \cdot \bar{D}$ ausreicht, damit $Q=1$ wird ($A=X$)

→ Terme, in denen eine Variable komplementär auftritt, lassen sich verkürzen:

$$A \cdot B \cdot \bar{C} \cdot D + A \cdot B \cdot \bar{C} \cdot \bar{D} = A \cdot B \cdot \bar{C} \cdot (D + \bar{D}) = A \cdot B \cdot \bar{C}$$

Schritt I

► Alle Produkterme werden in einer Tabelle eingetragen: Für jede Variable wird der Wert eingetragen, damit $Q=1$ wird (Selektierte Funktionstabelle!):

A	B	C	D	
1	1	0	1	(1) ⊗
1	1	1	0	(2) ⊗
0	1	1	0	(3) ⊗
1	1	0	0	(4) ⊗
0	1	0	0	(5) ⊗
1	0	0	0	(6) ⊗

Tabelle 9: Beispiel: $Q = A \cdot B \cdot \bar{C} \cdot D + A \cdot B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot C \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot \bar{D}$

Schritt II

► Terme, die sich nur in einer Variable unterscheiden, heißen ähnlich:
Im Beispiel: (1) \Leftrightarrow (4), (2) \Leftrightarrow (3), (2) \Leftrightarrow (4), (3) \Leftrightarrow (5), (4) \Leftrightarrow (5), (4) \Leftrightarrow (6)

A	B	C	D	
1	1	0	X	(14)
X	1	1	0	(23) \otimes
1	1	X	0	(24) \otimes
0	1	X	0	(35) \otimes
X	1	0	0	(45) \otimes
1	X	0	0	(46)

Tabelle 10: Beispiel: Bildung der verkürzten Terme. (nm) bedeutet: Ableitung aus Term (n) und (m)

Schritt III

► Für die neuen verkürzten Terme wird Schritt II wiederholt:

(2435) \Rightarrow X1X0

(2345) \Rightarrow X1X0

↳ Der Vorgang endet, wenn keine 1-komplementären Terme mehr auftreten.

Schritt IV

► Alle Terme, die nicht verkürzt werden konnten, sind Primterme:

(14) \rightarrow $A \bullet B \bullet \bar{C}$

(46) \rightarrow $A \bullet \bar{C} \bullet \bar{D}$

(2435) \rightarrow $B \bullet \bar{D}$

↳ Verkürzte (minimierte) boolesche Funktion:

$$Q' = A \bullet B \bullet \bar{C} + A \bullet \bar{C} \bullet \bar{D} + B \bullet \bar{D} \quad \checkmark$$

► Einzelne Primterme sind möglicherweise noch redundant. Ihre Anzahl läßt sich durch Bestimmung der minimalen Überdeckung noch minimieren.

Schritt V

► Alle Produktterme (nicht minimiert) werden in Zeilen, und alle Primterme in Spalten einer neuen Tabelle eingetragen:

	$AB\bar{C}$	$A\bar{C}\bar{D}$	$B\bar{D}$
$AB\bar{C}D$	X		
$ABC\bar{D}$			X
$\bar{A}BC\bar{D}$			X
$AB\bar{C}\bar{D}$	X	X	X
$\bar{A}B\bar{C}\bar{D}$			X
$A\bar{B}\bar{C}\bar{D}$		X	

Tabelle 11: Beispiel: Suche der Überdeckungen zwischen Produkt- und Primtermen.

Schritt VI

► Alle Überdeckungen werden markiert (X), d.h. wenn ein Primterm vollständig in einem Minterm enthalten ist.

Schritt VII

► Alle Primterme können weggelassen werden, solange in jeder Zeile min-

destens eine Überdeckung vorhanden ist. Im Beispiel ist keine weitere Reduzierung möglich!

Binary-Decision-Diagramm (BDD)-Methode

- ▶ Ein Binary-Decision-Diagramm (BDD) ist ein azyklischer und gerichteter Graph, der einen Algorithmus zur Berechnung einer booleschen Funktion (BF) beschreibt.
- ▶ Jede boolesche Funktion kann als BDD dargestellt werden.
- ▶ Berechnung der dargestellten Funktion für einen gegebenen Eingangsvektor $\{x_1, \dots, x_n\}$ beginnt an der Quelle (Wurzelknoten).
- ▶ Ein BDD besteht aus einem Startknoten (Quelle) und inneren Knoten mit dem Ausgangsgrad 2.
- ▶ Die inneren Knoten sind Variablen der BF zugeordnet.
- ▶ Die beiden ausgehenden Kanten der inneren Knoten entsprechen der booleschen Wertemenge $\{0,1\}$.
- ▶ Am Ende eines Pfades im BDD befindet sich eine Senke mit dem Ausgangsgrad 0.

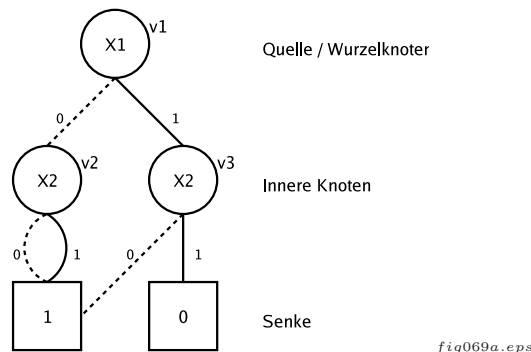


Abbildung 47: Beispiel: $f(x_1, x_2) = \neg x_1 + \neg x_2$

Shanon-Zerlegung

- ▶ Ein Pfad (Quelle \rightarrow Senke) beschreibt die Evaluierung eines Funktionswertes der BF.
- ▶ Erzeugung eines BDD's aus BF schrittweise durch Evaluierung der einzelnen Variablen $\{x_1, \dots, x_n\}$ mit den Werten $\{0,1\}$:

$$f = \neg x_1 f_{|x_1=0} \vee x_1 f_{|x_1=1} \quad (20)$$

Jeder innere Knoten stellt eine neue (Sub-) BF dar:

$$f_{v1} = \neg x_1 + \neg x_2$$

$$f_{v2} = 1$$

$$f_{v3} = \neg x_2$$

- ▶ Die Größe eines BDD's ist maximal $O(2^n/n)$ bei einer BF mit n Variablen.

► **Nachteilige Eigenschaft von BDDs** (wenn n groß ist):
 Die Tatsächliche Größe und Struktur eines BDD's hängt von der Variablenreihenfolge bei der Erzeugung ab! (Ausnahme: symmetrische BF).

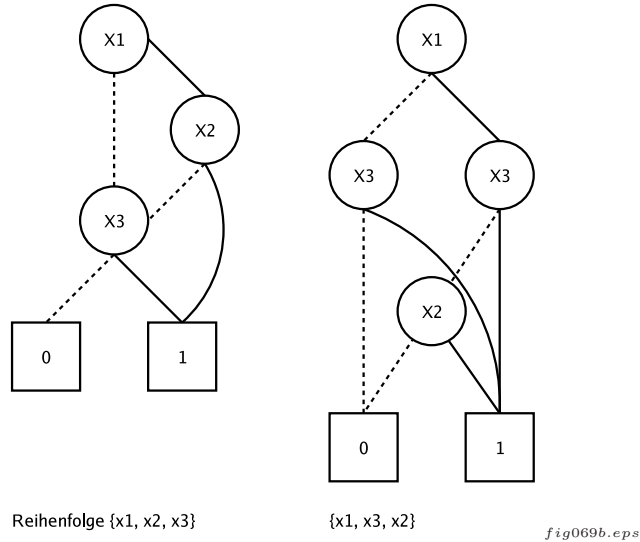


Abbildung 48: Beispiel zweier BDDs erzeugt aus $f(x_1, x_2, x_3) = x_1 \cdot x_2 + x_3$

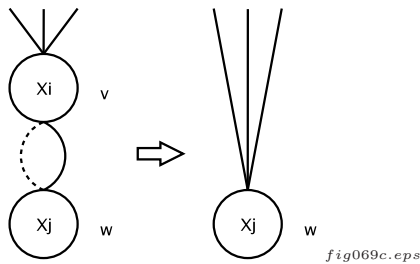
Ordered BDD

Reduktion von BDDs

► Ein OBDD ist ein BDD, in dem auf jeden Pfad alle Variablen höchstens einmal und gemäß einer vorgegebenen Ordnung getestet bzw. evaluiert werden müssen.

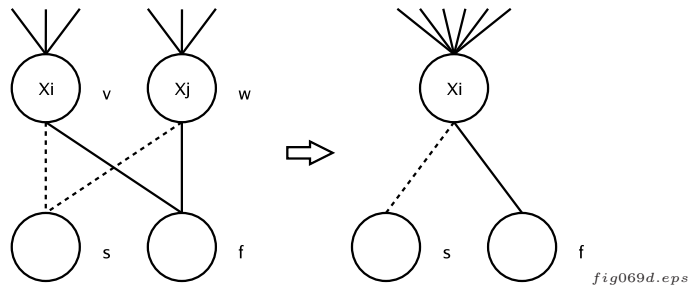
► Die Reduktion eines (O)BDD's hat das Ziel, die Anzahl der Pfade und Variablen zu minimieren, was in einer reduzierten und minimierten BF resultiert.

Deletion-Rule →



Die beiden Kanten eines Knotens $v(x_i)$ verzweigen beide auf den gleichen Nachfolgerknoten $w(x_j)$. Der Knoten v kann entfernt werden, und alle eingehenden Kanten werden auf w umgeleitet.

Merging-Rule →



Zwei Knoten $v(x_i)$ und $w(x_i)$ der gleichen Variable besitzen gleiche 0- und 1-Nachfolger. Die Knoten v und w können zu einem neuen Knoten v' zusammengefaßt werden.

Das folgende Beispiel zeigt eine Reduktion der BF

$$f(x_1, x_2, x_3, x_4) = x_1 \bullet x_2 \bullet x_3 \bullet x_4 + x_1 \bullet \neg x_2 \bullet x_3 + x_1 \bullet x_2 \bullet x_3 \bullet \neg x_4$$

zu

$$f(x_1, x_2, x_3, x_4) = x_1 \bullet x_3$$

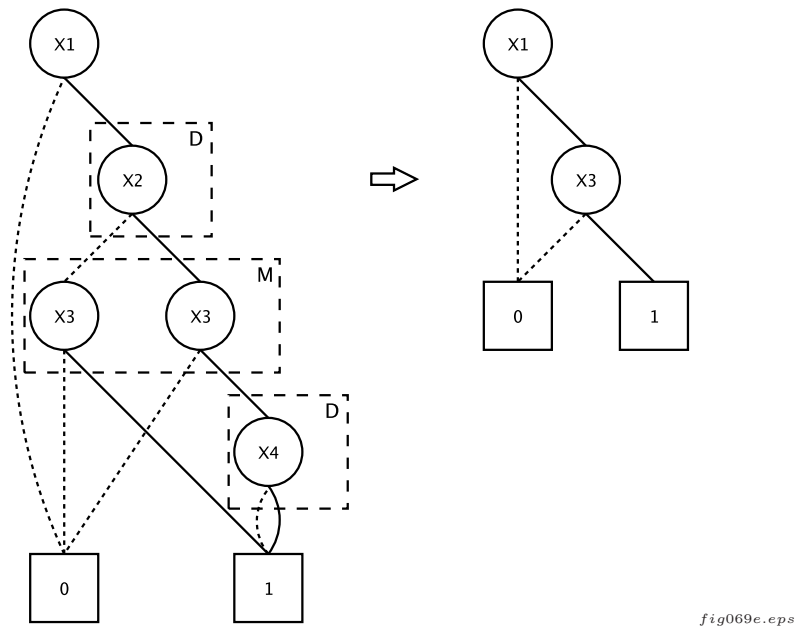


Abbildung 49: Beispiel einer BDD-Reduktion.

Addierer

3.6. Kombinatorische Logiksysteme

Ein Schaltnetz oder kombinatorische Logik ist eine logische Schaltung, deren Ausgangvariable(n) nur von den am Eingang anliegenden Werten, den Eingangsvariablen, abhängt.

➔ Mit boolescher Funktion darstellbar.

Werte aus der Vergangenheit bestimmen den aktuellen Wert der Ausgangsvariable(n) nicht. Im folgenden werden fundamentale Beispiele für Schaltnetze gezeigt.

➤ Ein Halbaddierer besitzt zwei Eingangsvariablen a und b, und zwei Ausgangsvariablen, die Summe und der Übertrag Carry, mit folgender Funktionstabelle:

a	b	c_out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tabelle 12: Halbaddierer.

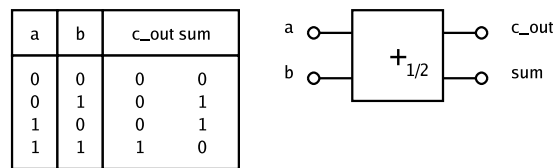


fig070.eps

Abbildung 50: Logisches Blocksymbol eines Halbaddierers.

Boolesche Gleichung:

$$\begin{aligned} \text{sum} &= \neg a \bullet b + a \bullet \neg b = a \oplus b \\ \text{c}_{\text{out}} &= a \bullet b \end{aligned} \tag{21}$$

Abbildung 51: Logikschaltung eines Halbaddierers.

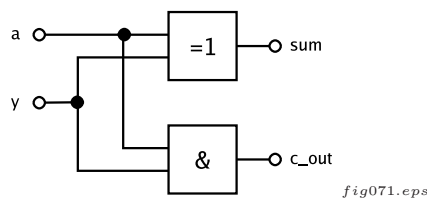


fig071.eps

Erweiterung eines Halbaddierers durch einen Volladdierer, der eine zusätzliche Eingangsvariable, den Übertrag einer weiteren Addiereinheit, enthält:

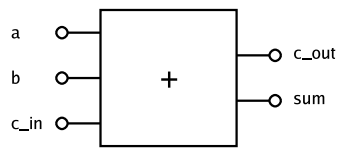


fig072.eps

Abbildung 52: Blockschaltbild eines Volladdierers.

Der Volladdierer kann aus zwei Halbaddierern, die kaskadiert werden, zusammengesetzt werden:

Abbildung 53:
Aufbau eines Volladdierers.

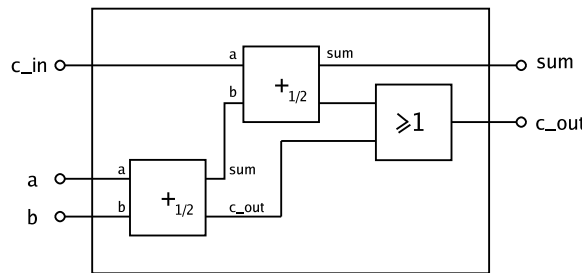


fig073.eps

Man erhält folgende boolesche Funktion für die Ausgangsvariablen Summe und Übertrag:

$$\begin{aligned} \text{sum} &= \neg a \cdot \neg b \cdot c_{in} + \neg a \cdot b \cdot \neg c_{in} + a \cdot \neg b \cdot \neg c_{in} + a \cdot b \cdot c_{in} & (22) \\ \Leftrightarrow \text{sum} &= a \oplus b \oplus c \\ c_{out} &= \neg a \cdot b \cdot c_{in} + a \cdot \neg b \cdot c_{in} + a \cdot b \cdot \neg c_{in} + a \cdot b \cdot c_{in} \end{aligned}$$

N-Bit Addierer

► Ein Addierer zur Addition zweier N Bit breiten Bitvektoren läßt sich mit verschiedenen Architekturen aufbauen, die unterschiedliche Eigenschaften besitzen. Alle Architekturen involvieren Volladdierer.

Ripple-Carry-Addierer

► Bei dieser Architektur werden N Volladdierer kaskadiert, wie in folgender Abbildung gezeigt ist. Dabei findet eine Übertragungssignal-Propagierung vom niederwertigsten zum höchstwertigen Bit statt, d.h. die Berechnung des N-ten Bits erfordert die Ergebnisse der vorherigen N-1 Addierer.

Abbildung 54:
Ripple-Carry-Addierer

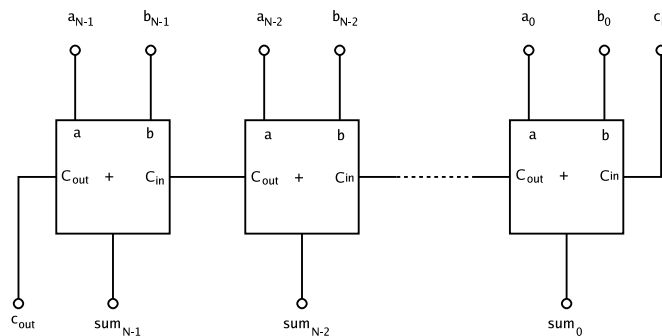


fig074.eps

Die Zeit, die ein Signal vom Eingang eines Logikgatters oder einer damit aufgebauten Logikschaltung benötigt, nennt man Laufzeit. Die Laufzeit bei elektronischen Schaltungen resultiert aus der verwendeten Transistortechnologie, und ist bei CMOS-Technologie in der Zeit begründet, um eine (parasitäre) Kapazität bei einem Logikpegelwechsel umzuladen. Insbesondere die Gate-Source-Kapazität hat Einfluß auf die Schaltzeit des Transistors.

Abbildung 55:
Parasitäre Kapazitäten und Signallaufzeit.

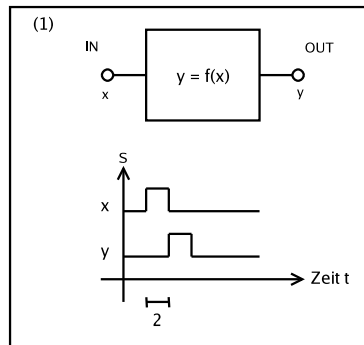


fig075.eps

Jede steuernde Transistorstufe, jede Zuleitung besitzt einen ohmschen (und induktiven) Widerstand, der zusammen mit der technologischen Kapazität C ein RC-Glied bildet.

Abbildung 56:
Ursache der Signalverzögerung durch Signalverhalten eines äquivalenten RC-Gliedes.

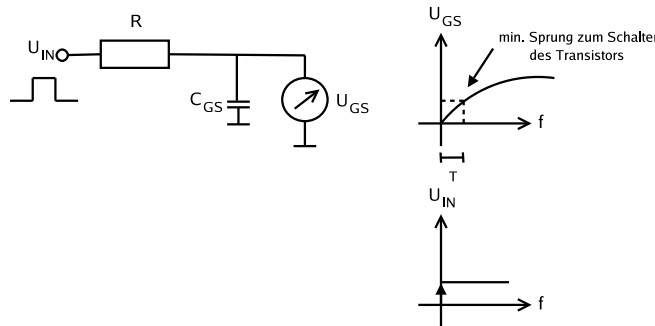


fig076.eps

Der Ripple-Carry-Addierer ist aufgrund der Signallaufzeit langsam, da die gesamte Laufzeit für das höchstwertige Bit

$$T_{\text{tot}} = \sum_{i=0}^{N-1} \Delta T_i \tag{23}$$

beträgt. Während dieser Zeit ist die kombinatorische Logikschaltung metastabil, d.h. die einzelnen Ausgänge des Addierers ändern sich zeitlich mehrfach.

Carry-Lookahead-Addierer

► Der Carry-Lookahead-Addierer besitzt verbesserte Laufzeiteigenschaften, da auf Kaskadierung verzichtet wird, indem die Carry-Signale direkt aus den Eingangsvariablen berechnet werden.

Herleitung:

$$s_i = a_i \oplus b_i \oplus c_i \quad (24)$$

$$\begin{aligned} c_{i+1} &= a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i \\ &= a_i \cdot b_i + (a_i + b_i) \cdot c_i \\ \Rightarrow c_{i+1} &= g_i + p_i \cdot c_i \end{aligned}$$

Dabei ist g der sogenannte generierende Term, und p der sog. Durchlauf-term. Für einen N -Bit-Addierer kann man dann evaluieren:

$$c_1 = g_0 + p_0 \cdot c_{in} \quad (25)$$

$$\begin{aligned} c_2 &= g_1 + p_1 \cdot c_1 \\ &= g_1 + g_0 \cdot p_1 + p_0 \cdot p_1 \cdot c_{in} \end{aligned}$$

$$\begin{aligned} c_3 &= g_2 + p_2 \cdot c_2 \\ &= g_2 + g_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 + p_0 \cdot p_1 \cdot p_2 \cdot c_{in} \end{aligned}$$

$$c_{i+1} = \sum_{j=0}^i \left(g_j \prod_{k=j+1}^i P_k \right) + \prod_{k=0}^i P_k \cdot c_{in} \quad (26)$$

Jedes Carry-Signal als Eingang für einen Volladdierer hängt nur noch von den primären Eingangssignalen ab.

Nachteil: bei großem N werden große Anzahl von Und-Gattern benötigt. Daher wird meistens ein hierarchischer Aufbau mit Teilkomponenten für den Lookahead-Addierer verwendet, mit den Bestandteilen:

1. Mini-Addierer (Ripple-Carry):

$$f : (a, b, c) \rightarrow P, G, S$$

2. Carry-Lookahead-Generator:

$$f : (P, G) \rightarrow C$$

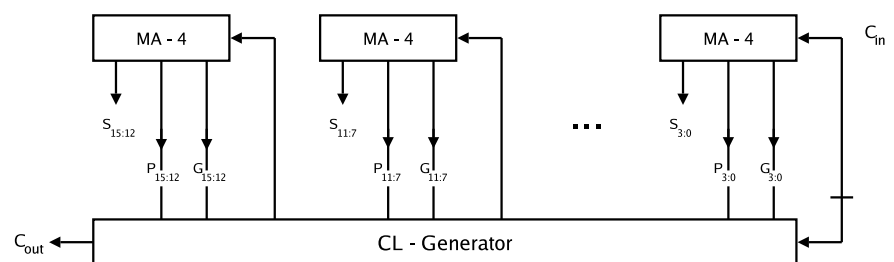


fig077.eps

Abbildung 57: Hierarchischer Carry-Lookahead-Addierer.

Multiplizierer

► Ein 1-Bit Multiplizierer besitzt folgende Funktionstabelle:

a	b	y=a•b
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 13: Multiplizierer

Ein Multiplizierer für die Multiplikation von N-Bit \times M-Bit breiten Zahlen A und B muß zunächst mathematisch behandelt und hergeleitet werden:

$$A = \sum_{i=0}^{n-1} A_i 2^i \quad (27)$$

$$B = \sum_{j=0}^{m-1} B_j 2^j$$

$$A \times B = \sum_{i=0}^{n-1} A_i 2^i \times \sum_{j=0}^{m-1} B_j 2^j = \sum_i \sum_j A_i B_j 2^{i+j}$$

Dieses Produkt hat $m \times n$ Terme. Umformung zu einer Summe mit Laufindex $k=i+j$ führt zu:

$$A \times B = \sum_{k=0}^{m+n-1} P_k 2^k \quad (28)$$

$$P_0 = A_0 \bullet B_0$$

$$P_1 = A_1 \bullet B_0 + A_0 \bullet B_1$$

$$P_2 = A_2 \bullet B_0 + A_1 \bullet B_1 + A_0 \bullet B_2 \text{ usw.}$$

Ein Parallelmultiplizierer setzt sich daher aus Addition und 1-Bit-Multiplizierern zusammen. Ein 4×4 Multiplizierer benötigt:

16 Und-Gatter

8 Volladdierer

4 Halbaddierer

Einführung einer Basiszelle aus 1-Bit-Multiplizierern und Volladdierern ermöglicht Aufbau des Multiplizierers mit einer systolischen Matrixstruktur.

Abbildung 58:
Basiszelle eines Multiplizierers.

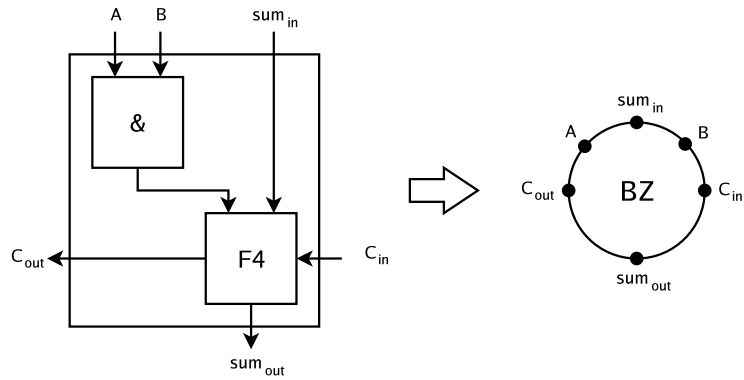


fig078.eps

Abbildung 59:
Matrix-Struktur eines 2×2 Multiplizierers unter Verwendung von Basiszellen.

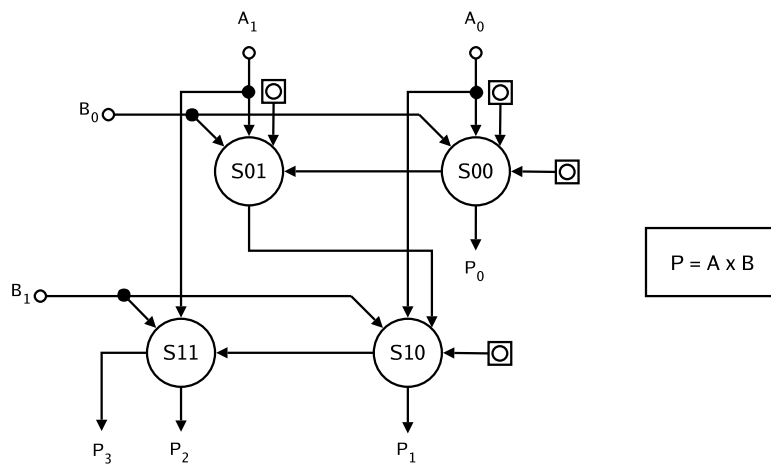


fig079.eps

**Multiplexer
und
Demultiplexer**

- ▶ Multiplexer sind elektronisch gesteuerte Auswahlwähler.
- ▶ Ein Multiplexer legt eines von N Eingangssignalen auf eine Ausgangsleitung. Die Auswahl erfolgt durch eine anliegende Adresse → Datenselektor
- ▶ Umgekehrten Vorgang mit Demultiplexer, der ein Eingangssignal auf N Ausgänge verteilt. Die Auswahl des Ausgangs erfolgt wieder durch Adressierung.

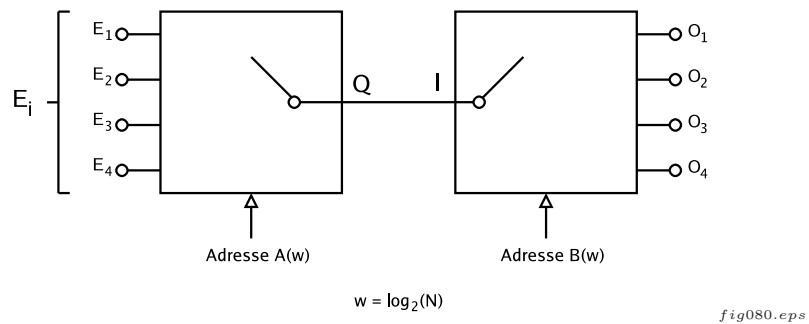


Abbildung 60: Multiplexer und Demultiplexer.

- ▶ Mit Multiplexern lassen sich beliebige logische Funktionen realisieren:

Eingangsvariablen → Adressensignale A_i des Multiplexers

Ausgangsvariable → Ausgangssignal Q des Multiplexers

Logikwerte → gegeben durch Eingangssignale E_i

$$f(A) = S(A, E) \tag{29}$$

$$S \rightarrow \{E_i \text{ wenn } A_i = 1\}$$

Tabelle 14: Beispiel eines Multiplexers als Implementierung einer booleschen Funktion. Die E-Matrix wird zeilenweise Oder-verknüpft.

a	b	q	E_1	E_2	E_3	E_4
0	0	1	1	0	0	0
1	0	1	0	1	0	0
0	1	0	0	0	0	0
1	1	0	0	0	0	0
			1	1	0	0

Abbildung 61:
Beschaltung des Multiplexers aus obigen Beispiel.

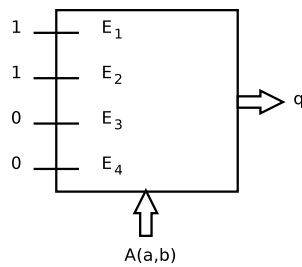


fig081.eps

Boolesche Funktion eines N-Bit Multiplexers

► Gesucht: $f(A,E)$?

Ein 1-Bit-Multiplexer besteht aus einer Und-Verknüpfung mit einem Eingangssignal e und einem Selektorsignal a: $f(e,a) = a \bullet e$:

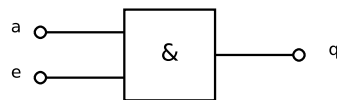


fig082.eps

Abbildung 62: 1-Bit Multiplexer

Ein N-Bit Multiplexer ist aus einer Oder-Verknüpfung einzelner 1-Bit Multipelxer aufgebaut:

Abbildung 63: N-Bit Multiplexer

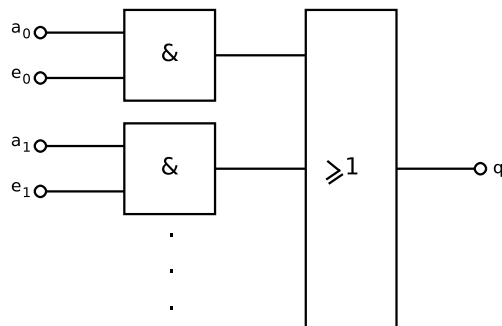


fig083.eps

Die Struktur besteht aus einer Und-Matrix mit N Eingangssignalen mit einer nachfolgenden Oder-Verknüpfung bzw. Oder-Matrix bei M Ausgangssignalen. Diese Und-Oder-Struktur ist charakteristisch für programmierbare Logikbausteine.

Die einzelnen Selektorsignale a_i müssen noch aus den eigentlichen Adresssignalen A dekodiert werden. Dazu wird ein Demultiplexer mit $I=1$ verwendet. Beispiel für $N=2$:

Abbildung 64:
Demultiplexer als
Adresssignaldekodier-
er.

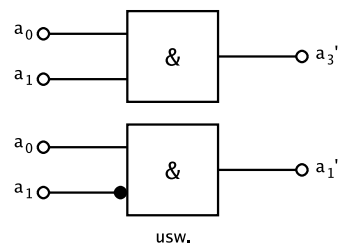


fig084.eps

$$\begin{aligned}
 f(E, A) = & E_0 \cdot \neg A_0 \cdot \neg A_1 & (30) \\
 & + E_1 \cdot A_0 \cdot \neg A_1 \\
 & + E_2 \cdot \neg A_0 \cdot A_1 \\
 & + E_3 \cdot A_0 \cdot A_1
 \end{aligned}$$

3.7. Sequenzielle Logik

Sequenzielle Logik besteht aus:

- Kombinatorischer Logik
- Speicherelementen

Die Ausgangssignale solcher Logik hängen von den aktuellen Eingangssignalen E (Eingangszuständen) und zusätzlich von Signalen (Zuständen) aus der Vergangenheit ab:

Abbildung 65:
Allgemeine Struktur
eines sequenziellen
Logiksystems.

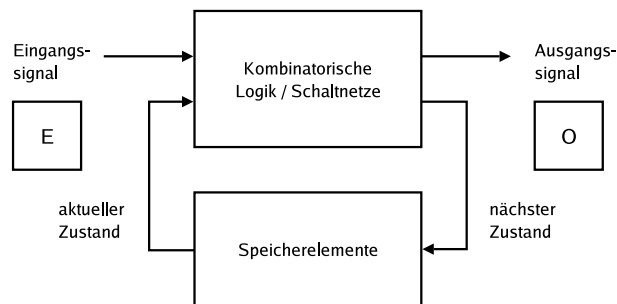


fig085.eps

Das aktuelle Ausgangssignal O bzw. eine logische Funktion davon wird zeitlich verzögert der kombinatorischen Logik durch Rückkopplung wieder zugeführt. Die Speicherelemente werden als Zustandsspeicher des Systems bezeichnet. Man unterscheidet den aktuellen Zustand Z_n des Systems und den nächsten Zustand Z_{n+1} des Systems:

$$\begin{aligned} [O_{n+1}] &= Z_{n+1} = f(Z_n, E_{n+1}) \\ [O_n] &= Z_n = f(Z_{n-1}, E_n) \end{aligned} \quad (31)$$

► Man unterscheidet synchrone und asynchrone sequenzielle Systeme. Synchrone Systeme ändern ihren Zustand nur zu bestimmten Zeitpunkten, asynchrone können ihren Zustand jederzeit (zeitkontinuierlich) ändern. D.h. synchrone Systeme sind taktgesteuert.

► Der (dynamische) Zustandsspeicher eines asynchronen Systems ist implementierbar durch eine zeitliche Verzögerung, im einfachsten Fall eine Leitung der Länge L mit $\Delta t \approx 1/c \cdot L$ (c : Lichtgeschwindigkeit), technologisch und mit größerer Verzögerungszeit zwei kaskadierte Inverter.

► Der Entwurf synchroner Schaltungen ist deutlich einfacher als bei asynchronen. Bei synchronen Systemen übernehmen die Speicherelemente nur zu bestimmten Zeitpunkten oder Intervallen die Daten.

► Die Speicherelemente lassen sich in zwei Gruppen unterteilen:

1. Ungetaktete FLIP-FLOP-Speicher, sog. Latches,
2. Getaktete FLIP-FLOP-Speicher, sog. Register.

Speicherelemente sequenzieller Logik

Latch-Speicher oder RS-Latch

► Aufgebaut mit zwei rückgekoppelten Logikgattern, z.B. Nor-Gattern:

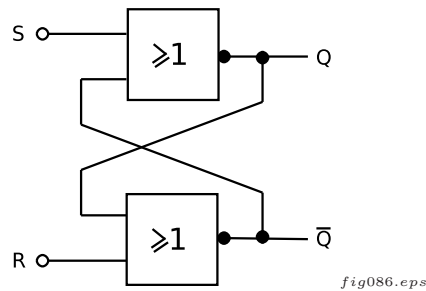


Abbildung 66: NOR-RS-Latch-Speicher

S	R	Q	\bar{Q}
0	0	Q_{n-1}	\bar{Q}_{n-1}
0	1	0	1
1	0	1	0
1	1	0	0

Tabelle 15: S: Set, R: Reset.

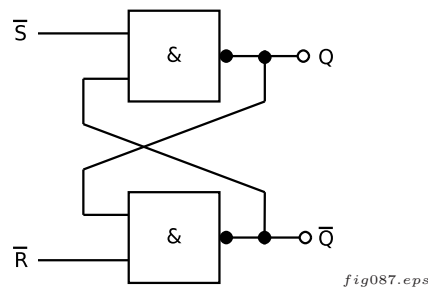


Abbildung 67: RS-Latch alternativ mit NAND-Gattern.

Taktgesteuertes RS-FLIP-FLOP

► Das taktgesteuerte RS-FLIP-FLOP stellt eine Erweiterung des RS-Latch-Speichers mit einem Aktivierungseingang dar, und ist zweistufig aufgebaut. Der RS-Speicher ist nur bei $T=1$ schaltfähig. Ist $T=0$, bleiben die Daten an den Ausgängen unverändert, unabhängig von den Eingangsdaten.

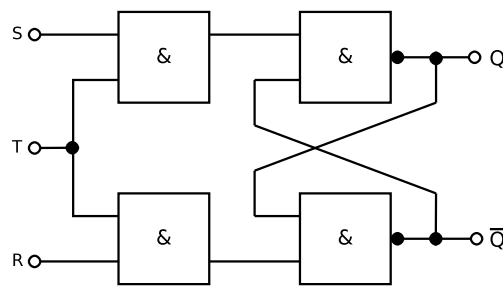


fig088.eps

Abbildung 68: T-RS-FLIP-FLOP.

Taktgesteuertes D-FLIP-FLOP

► Das D-FLIP-FLOP vermeidet durch geeignete Beschaltung am Eingang die irreguläre Eingangskombination $RS=\{00,11\}$.

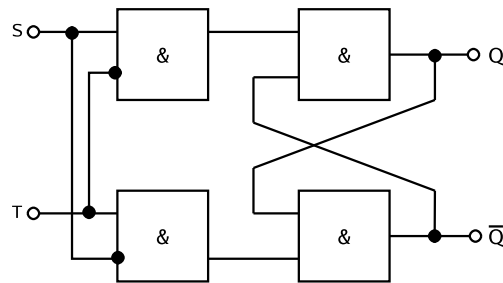


fig089.eps

Abbildung 69: D-FLIP-FLOP

Taktflanken gesteuertes FLIP-FLOP (Register)

► Mit Taktflankensteuerung werden FLIP-FLOPs synchron, d.h. nur zu bestimmten diskreten Zeitpunkten gesteuert. Dadurch wird eine größere Störsicherheit erreicht!

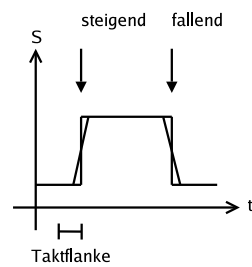


fig090.eps

Abbildung 70: Taktflankensteuerung

Für die Taktflankensteuerung werden Impulsglieder (Hochpass-Filter) benötigt, die nur bei einer auftretenden Änderung des Taktsignals $0 \rightarrow 1$ oder $1 \rightarrow 0$ einen kurzen Impuls erzeugen.

Abbildung 71:
Übertragungsfunktion
von Impulsgliedern,
aufgebaut mit einem
RC-Schaltznetzwerk
und einer Diode.

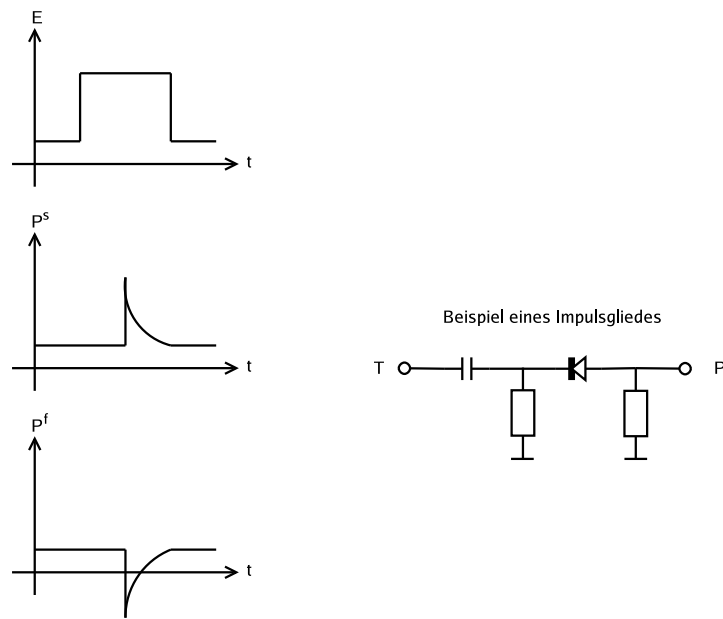


fig091.eps

Das Signal P dient als neuer Aktivierungseingang des RS- oder D-FLIP-FLOP's.

**Taktflanken
gesteuertes D-
und JK-FLIP-
FLOPs**

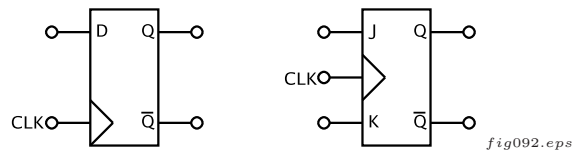


fig092.eps

Abbildung 72: D- und JK-FLIP-FLOPs, die häufig in sequenzieller Logik eingesetzt werden.

D	CLK	Q_{n+1}	\bar{Q}_{n+1}
X	0	Q_n	\bar{Q}_n
X	1	Q_n	\bar{Q}_n
0	↑	0	1
1	↑	1	0

Tabelle 16: D-FLIP-FLOP

Das JK-FLIP-FLOP ist ähnlich dem RS-FF mit logischer Eingangsverknüpfung der J- und K-Eingänge aufgebaut.

J	K	CLK	Q_{n+1}	$\neg Q_{n+1}$
X	X	0	Q_n	$\neg Q_n$
X	X	1	Q_n	$\neg Q_n$
0	0	↑	Q_n	$\neg Q_n$
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	Q_n	$\neg Q_n$

Tabelle 17: JK-FLIP-FLOP

3.8. Beispiele für sequenzielle Systeme

Wichtige sequenzielle Systeme:

Schieberegister → Daten werden taktgesteuert in einem N-Bit breiten Register jeweils um eine Stelle nach links oder rechts verschoben.
Anwendung: Seriell-Parallel-Konverter.

Zähler → Mit taktflankengesteuerten FLIP-FLOPs aufgebaute Zähler.

Zustandsautomaten → Dienen der Implementierung komplexer sequenzieller Systeme mit einer endlichen Zustandsmenge.

Zähler

► Man unterscheidet:

Asynchrone Zähler → Kaskadierung von D-FLIP-FLOPs. Zähler vorwiegend für Binärzahlensystem verwendet. Die Laufzeit der einzelnen Ausgänge des Zählers ist nicht konstant, d.h. das erste Bit hat die kürzeste, und das letzte Bit die längste Laufzeit.

Synchrone Zähler → Realisiert als rückgekoppelter Zustandsautomat. Hier haben die einzelnen Ausgangsbits konstante Laufzeit, d.h. nach einer Verzögerungszeit Δt liegen alle Bits des Zählers als gültiger Wert vor. Die Rückkoppellogik (rein kombinatorisch) bestimmt das Zählverhalten.

Abbildung 73:
Beispiel eines 3-Bit Synchronzählers mit D-FLIP-FLOPs.

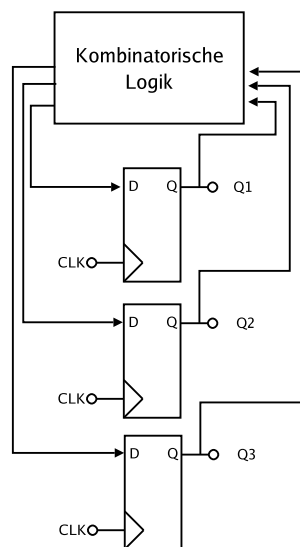


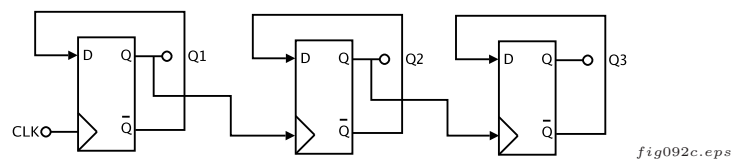
fig092b.eps

VHDL-Beschreibung
des obigen
Synchronzählers.

```
entity syn3count is
  port ( clk: in bit;
        q: out bit_vector(2 downto 0);
  end syn3count;
architecture main of syn3count is
```

```
signal q_int: bit_vector(2 downto 0);
begin
  process (clk)
  begin
    if clk'event and clk='1' then
      q_int <= q_int + "001";
    end if;
  end process;
  q <= q_int;
end main;
```

Abbildung 74:
Beispiel eines 3-Bit
Asynchrnzählers mit
rückgekoppelten
D-FLIP-FLOPs.



4. Programmierbare Logikbausteine

Programmierbare Digitallogik (PDL) dient der technologischen Umsetzung und Implementierung von Schaltnetzen, die mit booleschen Funktionen beschrieben werden können, und Register-Transfer-Logik.

”Programmierbar” ist hier falscher Kontext, da ein Programm eine Ablaufvorschrift darstellt; besser ”anwendungsspezifisch konfigurierbar”.

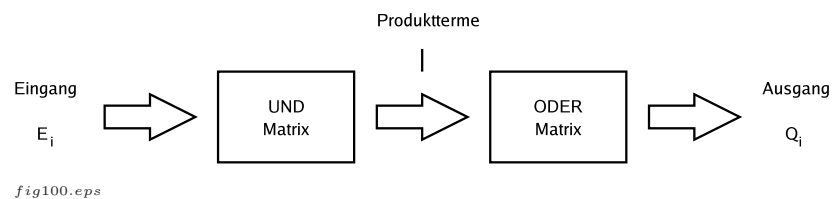
► Ausgangspunkt für PDL-Bausteine (oder engl. Programmable Logic Devices PLD) ist die disjunktive Normalform zur Beschreibung von kombinatorischer Logik (SOP) mit M Ausgangsvariablen bzw. Signalen und N Eingangsvariablen:

$$Q_i = P_{1i} + P_{2i} + \dots + P_{ni} \quad (32)$$

wobei P_{ij} ein Produktterm ist, bei dem die Ausgangsvariable $Q_i=1$ annimmt.

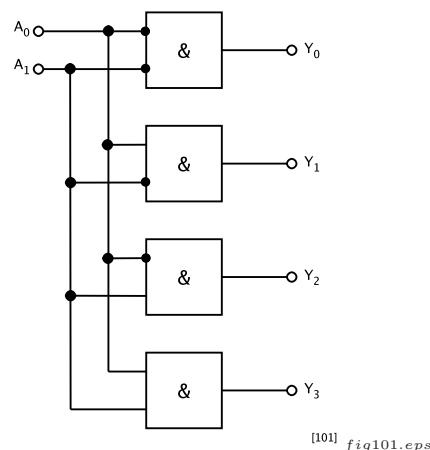
► Eine disjunktive Normalform besteht aus einer Und-Verknüpfungsebene (-matrix), die die Produktterme P_{ij} bildet, und eine Oder-Verknüpfungsebene (-matrix), die die einzelnen Produktterme verbindet.

Abbildung 75:
Technologische Umsetzung der Disjunktiven Normalform.



Ein Adreßdekoder (1-aus-N Dekoder) besteht aus einer Und-Verknüpfungsmatrix.

Abbildung 76:
1-aus-4-Dekoder.



➔ geeignet um Produktterme zu bilden, aber alle Produktterme ergeben

Ausgangswert = 1!

↳ daher wird Produktterm selektive Oder-Matrix benötigt.

4.1. ROM- und RAM-Bausteine als PLD

Ein ROM (Read-Only-Memory) oder RAM (Random Access Memory)-Baustein beinhaltet einen Adreßdekor (Und-Matrix) und eine selektive Oder-Matrix, die durch die Speicherzellen gebildet wird.

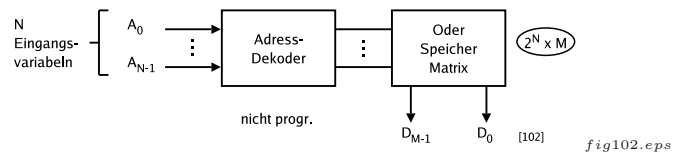


Abbildung 77: ROM/RAM-Architektur.

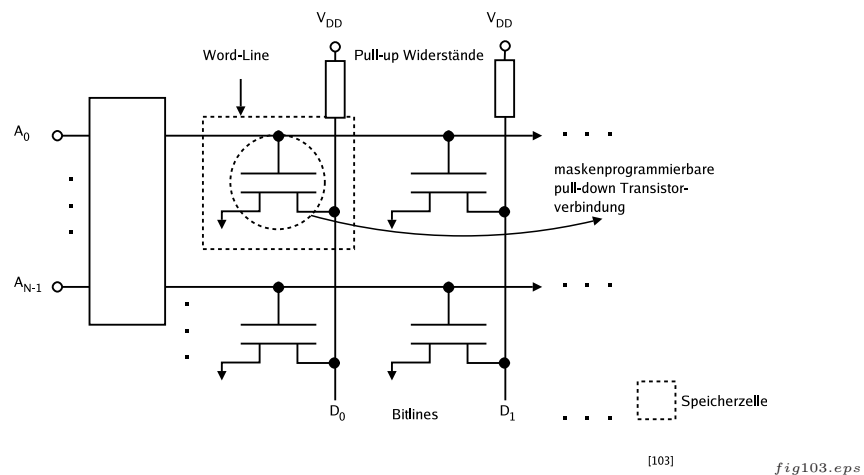
Nachteile bei der Verwendung von ROM/RAM-Logikbausteinen als PLD:

1. Geringe Ausnutzung von Gatterressourcen,
2. Logikminimierung kann daher entfallen, der Adreßdekor=Und-Matrix ist überbestimmt,
3. nur kombinatorische Logik implementierbar; sequenzielle Systeme können wegen fehlender frei verfügbarer Register/FLIP-FLOPs nicht realisiert werden.

Maskenprogrammierbarer
nMOS-ROM-Baustein

Abbildung 78:
Maskenprogrammierbares
nMOS-ROM.

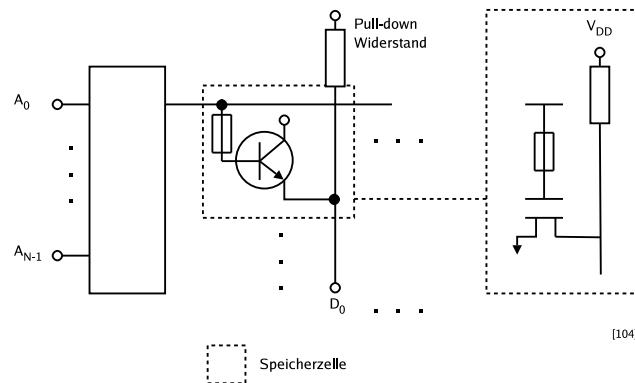
► Das Schaltnetz wird mit maskenprogrammierbaren pull-down Transistorverbindungen realisiert. Die Layoutmaske wird vor der Herstellung des ROM-Bausteins definiert und bestimmt den Fertigungsprozeß.



PROM-Baustein

► Das Schaltnetz wird mit einmalig programmierbaren Sicherungen und pull-up oder pull-down Transistorverbindungen realisiert. Die Layoutmaske ist daher universell und wird vor der Herstellung des ROM-Bausteins definiert. Man unterscheidet Fuse- und Anti-Fuse Technologien. Bei der ersteren wird eine Verbindung durch einen externen elektrischen Strom aufgetrennt, bei der letzteren wird eine Verbindung hergestellt.

Abbildung 79:
Programmable-ROM
(Fuse-Technologie).

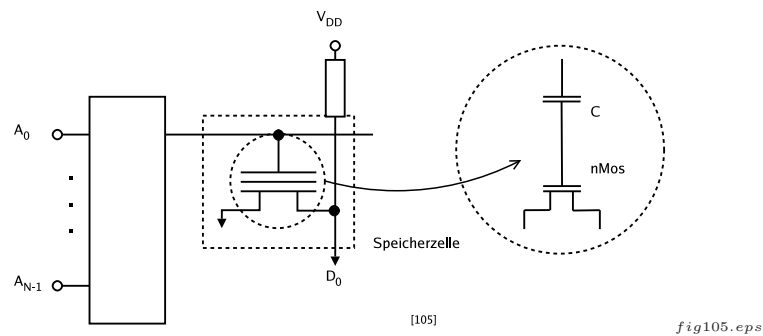


Flash-EEROM-
Baustein

► Das Schaltnetz wird mit mehrfach programmierbaren pull-down MOSFET-Transistorverbindungen realisiert. Bei diesen speziellen MOSFET-Transistoren mit einem sog. "floating gate", einer isolierten Kapazität, bestimmt eine elektrische Ladung das Schaltverhalten des Transistors. Es können also digitale Informationen mittels elektrischer Ladung auf lange Zeit gespeichert werden.

Durch Anlegen einer hohen Spannung in der isolierten Umgebung dieser Kapazität kommt es zu einem Ladungstransfer, wodurch die Kapazität ihre Ladung ändert, was dem Vorgang der Programmierung entspricht. Die Ladungsmenge dieser Kapazität bestimmt dann, ob beim Anlegen einer Gate-Spannung am entsprechenden Transistor dieser leitend wird oder nicht.

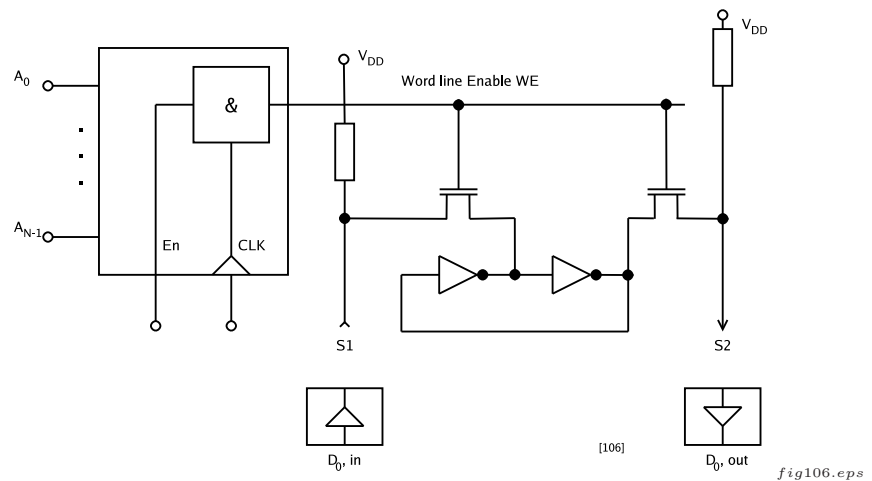
Abbildung 80:
Flash-Electrical-
Erasable-ROM.



SRAM-Baustein

► Schnelle aber flüchtige Programmierung kann mit SRAM-Zellen erfolgen. Eine SRAM-Zelle ist aus zwei rückgekoppelten Invertern (zwei MOSFET Transistoren) und zwei Steuertransistoren für die Datenein- und auskopplung aufgebaut, d.h. insgesamt sechs Transistoren.

Abbildung 81:
SRAM-Speicherblock
und Speicherzelle.



- ▶ Der Schreibvorgang setzt $S1=\bar{S}2$ voraus. Wenn die Schreibleitung $WE=1$ aktiviert wird, erfolgt Datenwechsel in der SRAM-Zelle.
- ▶ Der Lesevorgang setzt $S1=S2=1$ voraus, hier aber durch pull-up Widerstände als schwaches Signal ausgeführt, d.h. die Ausgangsstufen der SRAM-Zelle können den Logikwert von $S1$ und $S2$ ändern.
- ▶ Der Aufbau und die Funktionsweise ist kompliziert, aber benötigt weniger Transistoren \rightarrow Chip-Fläche als ein für ein D-FLIP-FLOP erforderlich sind (ca. 20).

4.2. Programmierbare Logikarrays (PLA)

Die UND-Matrix ist bei RAM/ROM-Bausteinen nicht programmierbar bzw. konfigurierbar. Daher schlechte Ausnutzung von Gatter- bzw. Transistorressourcen, was zu:

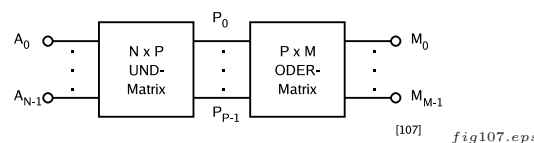
1. nicht notwendig größerer Chip-Fläche,
2. und erhöhter Leistungsaufnahme führt.

Spezielle Logikarrays mit programmierbarer UND- sowie ODER-Matrix stellen eine Verbesserung dar. Durch die zusätzliche konfigurierbare UND-Matrix kann die Matrixgröße reduziert werden, da durch Logikoptimierung bei fast allen Problemen ein Reduktion der Produktterme zu erwarten ist. Unter Ausnutzung der Logikminimierung erreicht man einen deutlich besseren Wirkungsgrad:

$$\eta = 1 - \frac{P}{2^N} \quad (33)$$

mit N als Anzahl der Eingangsvariablen, P Anzahl der Produktterme und folgend M als Anzahl der Ausgangsvariablen.

Abbildung 82:
Reduzierte konfigurierbare UND- und ODER-Matrix als PLA.

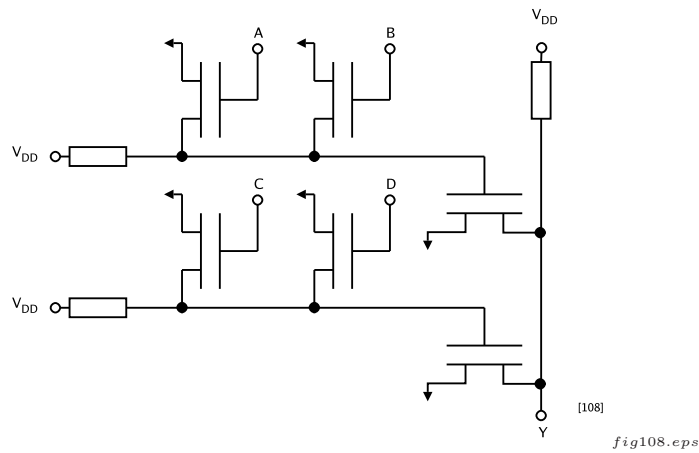


► Ein PLA implementiert daher eine zweistufige Funktion in SOP-Form, nicht notwendiger und sinnvollerweise in vollständiger Normalform. Je weniger Produktterme durch Minimierung die SOP-Funktion bilden, desto kleiner kann P gewählt werden. P wird aber bereits bei der Herstellung eines PLA-Bausteins seitens des Herstellers festgelegt, so daß sich nicht alle Funktionen in einem PLA bestimmter Größe realisieren lassen - P wird immer als Kompromiß zwischen Chip-Fläche/Komplexität und Funktionalität gewählt.

► Die UND-ODER-Matrix läßt sich auf eine reine ODER(NOR)-Matrix transformieren, was in technologischer Zweckmäßigkeit begründet sein kann:

$$\begin{aligned} Y &= \neg A \bullet \neg B + \neg C \bullet \neg D \\ &= \neg(A + B) + \neg(C + D) \end{aligned} \quad (34)$$

Abbildung 83:
Technische
Umsetzung eines
NOR-Gatters mit
sechs Transistoren.



4.3. Programmierbare Array Logik (PAL)

Die zweifache Matrixstruktur UND-ODER wird bei diesen Bausteinen vereinfacht, indem die ODER-Matrix fest verdrahtet wird und nicht programmierbar ist.

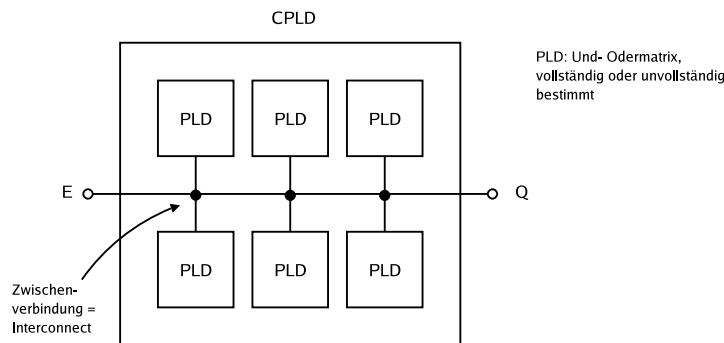
Die ODER-Matrix bildet eine feste Verknüpfung der Produktterme. Die Matrix ist unterbestimmt.

4.4. Komplexe PLD (CPLD)

Bei hoher Anzahl von booleschen Funktionen und hoher Anzahl von Eingangs- und Ausgangsvariablen sind monolithische PLA- oder PAL-Lösungen nicht effizient einsetzbar.

Lösung besteht in der Partitionierung eines gesamten PLD-Bausteins in eine Vielzahl von kleineren PLD-Blöcken (mit konfigurierbarer UND-ODER-Matrix).

Abbildung 84:
CPLD mit kleineren PLD-Blöcken und Verbindungsstrukturen.



Ein PLD-Block kann aus einer Mischung verschiedener Architekturen bestehen, die insbesondere Register in Form von D-FLIP-FLOPs enthalten, um Register-Transfer-Logik umsetzen zu können.

► Der Aufbau und Granularität des PLD-Blocks und deren Anzahl bestimmt die Möglichkeiten, ein gegebenes Logiksystem zu einer Problemstellung in einem CPLD implementieren zu können.

Da meist der PLD-Blockaufbau einfach (fein granularisiert) gehalten ist, werden diese Blöcke auch als Makrozellen bezeichnet.

► Es muß konfigurierbare Verbindungsleitungen zwischen den Makrozellen und zwischen Makrozellen und Ein-/Ausgabeschnittstellen geben. Feine Granularität erfordert eine umfangreiche Verbindungsmatrix im CPLD! Je komplexer ein CPLD aufgebaut und je umfangreicher das zu implementierende Problem ist, desto kritischer wird die Qualität der Verbindungsmatrix und die Granularität der Makrozelle. Häufig sind noch Gatterressourcen vorhanden, können aber nicht mehr verbunden werden.

Zwei Technologien haben sich bei CPLD-Bausteinen für die Konfiguration durchgesetzt:

1. EEPROM-Zellen, die sowohl die Konfiguration der Makrozellen als auch der Verbindungsmatrix dauerhaft speichern, aber mehrmals rekonfiguriert werden können,
2. und Fuse-/Antifuse-Verbindungen, die nur eine einmalige Konfiguration nach dem Herstellungsprozeß zulassen.

Abbildung 85:
Xilinx XC9500 CPLD:
Architektur.

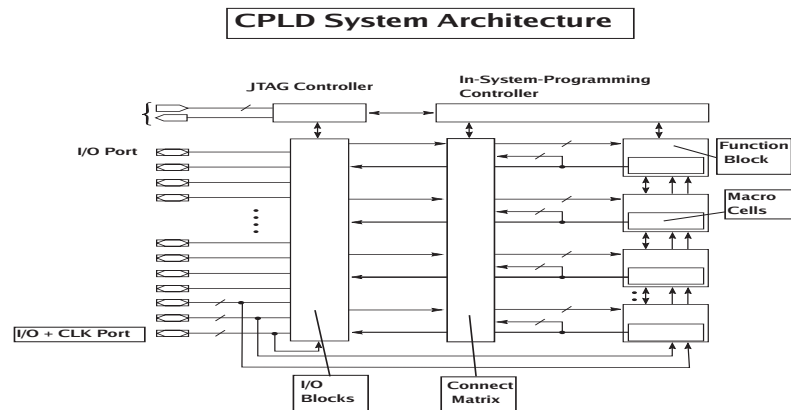


fig110.eps

► Die I/O-Blöcke verbinden die einzelnen Makrozellen mit der technischen Außenwelt. Dabei werden von diesen programmierbaren I/O-Blöcken verschiedene Logikstandards unterstützt:

- TTL bzw. Low Voltage TTL (LVTTL)
- LV CMOS
- LVDS als Beispiel eines differentiellen Signalübertragungsstandards mit zwei Leitungen (nicht direkt unterstützt vom XC9500)
- PCI (3,3V und 5V)
- und viele andere.

Die verschiedenen Signalstandards unterscheiden sich in den Spannungsintervallen, die dem Low- und High-Pegel zugeordnet sind. Weiterhin muß die Signalrichtung wählbar sein:

- Eingang (IN)
- Ausgang (OUT)
- Bidirektional (INOUT) mit Tri-state Ausgangsstufe.

Abbildung 86:
CPLD
Funktionsblock:
Architektur.

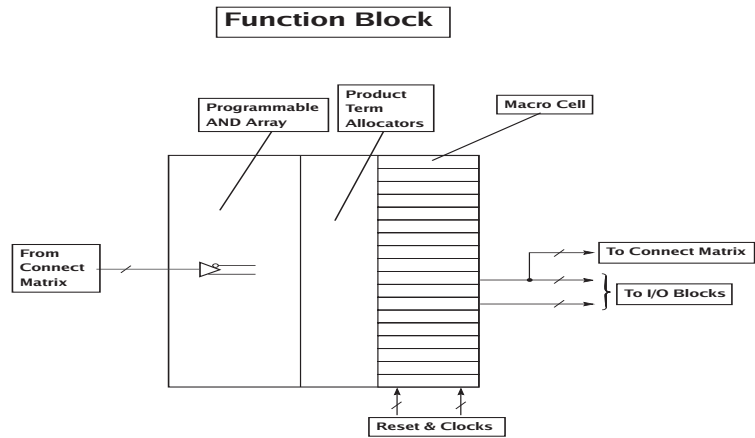


fig111.eps

► Die Funktionsblöcke enthalten bis zu 18 Makrozellen. Jede Makrozelle enthält ein universelles D-RS-FLIP-FLOP. Die verschiedenen Elemente und das FLIP-FLOP werden mittels Multiplexer verschaltetet.

Abbildung 87:
CPLD Makrozelle:
Architektur.

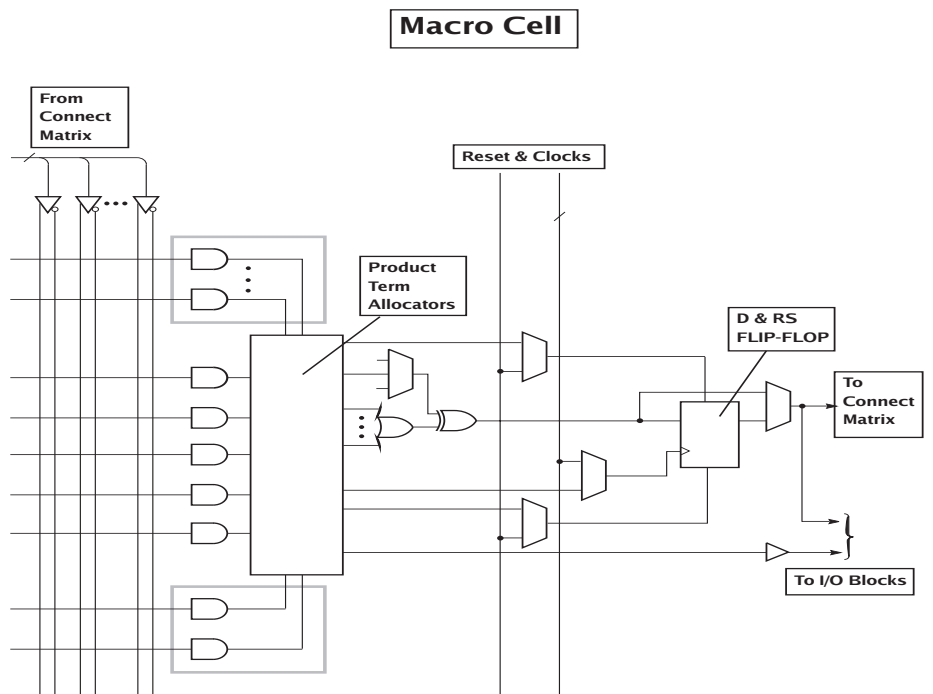


fig111b.eps

4.5. FPGA: Field Programmable Gate Array

Zusammenfassung der Eigenschaften von CPLDs:

1. CPLDs bestehen aus einer Matrix von PAL-Blöcken, die aus kombinatorischer Logik gebildet werden, die SOP-Ausdrücke mit vielen Eingangsvariablen implementieren, ergänzt durch Register. Die Registerdichte ist aber niedrig.
2. CPLDs sind daher gut geeignet für Applikationen mit geringer und mittlerer Gatter-, Funktions- und Registerdichte.
3. CPLDs haben eine gut vorhersagbares zeitlichen Verhalten, begründet in einem verhältnismäßig einfachen und regulären Aufbau.
4. Die Verbindungsmatrix besitzt eine Kreuzstruktur mit eingeschränkten Trassierungsmöglichkeiten.

FPGAs stellen eine Weiterentwicklung der CPLD-Technologie dar. Die Architektur ist komplexer und stärker registerbasiert.

FPGAs sind in Funktionseinheiten unterteilt, die untereinander mit einem flexiblen kanalbasierten Verbindungssystem verbunden sind.

Abbildung 88:
Aufbau eines FPGA mit Funktionseinheiten FU und Verbindungskanälen.

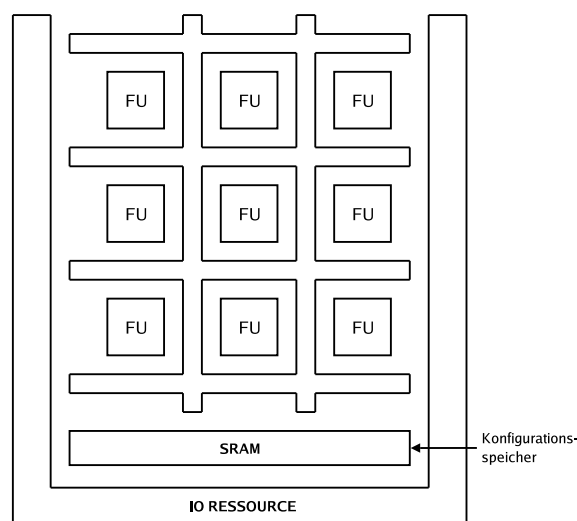


fig112.eps

Die FPGA-Funktionalität wird durch spezielle Komponenten ergänzt:

- Frei verfügbare RAM Blöcke (nicht zur Implementierung logischer Funktionen geeignet)
- vorgefertigte arithmetische und logische Funktionseinheiten:
 - ◇ Schieberegister
 - ◇ Addierer
 - ◇ Multiplizierer
- umfangreiche I/O-Ressourcen, die eine Vielzahl von I/O-Standards wie TTL, LVTTTL, CMOS, LVDS u.v.m. unterstützen,

- bis hin zu partiellen CPU-Kernen (Beispiel Xilinx-Virtex mit bis zu vier PowerPC CPU-Kernen auf dem FPGA-Chip).

► **FPGAs sind für mittlere und hohe Gatter-, Register- und Funktionsdichten und Rapid-Prototyping geeignet.**

Unterschiede zu CPLDs:

- Die Performance (Laufzeiten, max. Taktfrequenz) ist abhängig von der Verbindungsstrassierung. Das zeitliche Verhalten ist nur schwer vorhersagbar, außer durch post-Simulation.
- Die kombinatorischen Funktionen werden nicht mit vorgefertigten PAL-ähnlichen UND-ODER-Matrizen, sondern mit "Look-Up" Tabellen, d.h. mit SRAM, seltener EEPROM-Blöcken aufgebaut.
- Nachteil der SRAM basierten FPGAs:

Die Konfigurationsdaten (sowohl für die Verschaltung der Verbindungsmatrix als auch der Look-Up Tabellen) müssen bei jeder Inbetriebnahme erneut aus einem extern nichtflüchtigen Speicher (EEPROM) geladen werden. Hier sind FPGAs mit EEPROM-Zellen in der Verbindungsmatrix und den Look-Up Tabellen im Vorteil.

Abbildung 89:
Funktionseinheit eines FPGAs: Kleine SRAM-Blöcke zur Implementierung beliebiger logischer Funktionen.

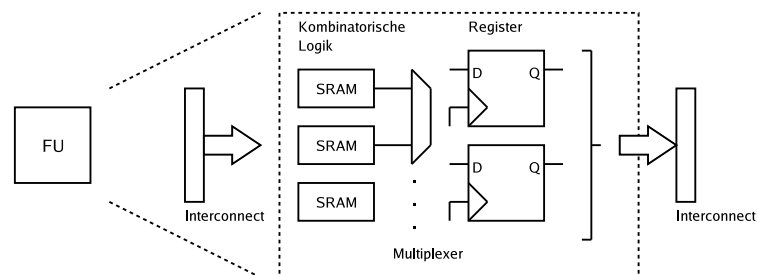


fig113.eps

Äquivalente
Gatterzahl

► Um FPGAs und Implementierung von Digitalssystemen in FPGAs vergleichen zu können, wird der Begriff des Äquivalentgatters verwendet. Ein Äquivalentgatter ist i.A. ein einfaches NAND-Gatter mit zwei Eingängen. Man kann nun einen Hardware-Entwurf in Äquivalentgatter "umrechnen", da sich alle logischen Funktionen auf reine NAND-Gatter transformieren lassen - dies gilt auch für FLIP-FLOPs in Näherung. Aktuelle FPGAs stellen ein Gatteräquivalent von ca. 100k-1M (10M) zur Verfügung. Da aber frei verfügbare Logikfunktionen in einem FPGA nicht direkt als FLIP-FLOPs verwendet werden können, ist die Anzahl frei verfügbarer FLIP-FLOPs getrennt von den möglichen Logikressourcen zu betrachten, anders als bei ASICs.

Um einen Eindruck zu vermitteln, mit welchem Logikgatterbedarf man bei bestimmter Funktionalität zu rechnen hat, ist in folgender Tabelle die Gatterzahl von ASICs aus der Sun Microsystems SPARC-Station 1 gezeigt.

ASIC	Gates
SPARC Integer CPU	20k
SPARC Floating Point CPU	50k
Cache Controller	9k
Memory Management Unit MMU	5k
Data	3k
Direct Memory Access DMA	9k
Video Controller	4k
RAM + Clock Controller	1+1k

Tabelle 18: ASIC Gatterzahl in der SUN SPARC-Station 1.

Diese Werte sind nicht direkt auf die Äquivalentgatterangabe von FPGAs übertragbar, da im Gegensatz zu ASICs die Trassierung der limitierende Faktor sein kann. Die Trassierung der Verbindungsleitungen zwischen den Funktionsblöcken und den Ein- und Ausgängen wird zentraler Bestandteil der Logiksynthese mit widersprechenden Forderungen:

1. möglichst viele Verbindungen \Leftrightarrow geringer Flächenbedarf
2. flexible Konfigurierbarkeit \Leftrightarrow kleine Signallaufzeiten.

Abbildung 90:
Konfigurierbarer und universell einsetzbarer I/O-Block eines Xilinx FPGA's.

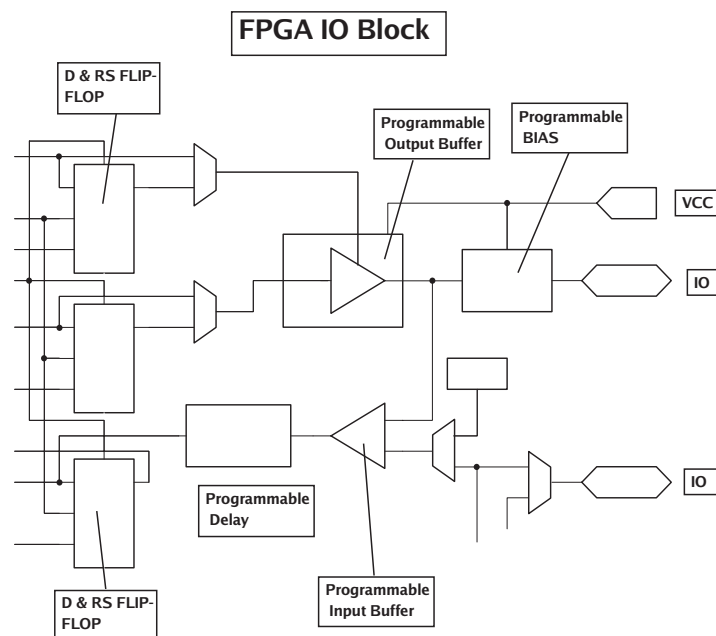


fig113b.eps

Abbildung 91:
Konfigurierbarer und
universell
einsetzbarer
Funktionsblock eines
Xilinx FPGA's.

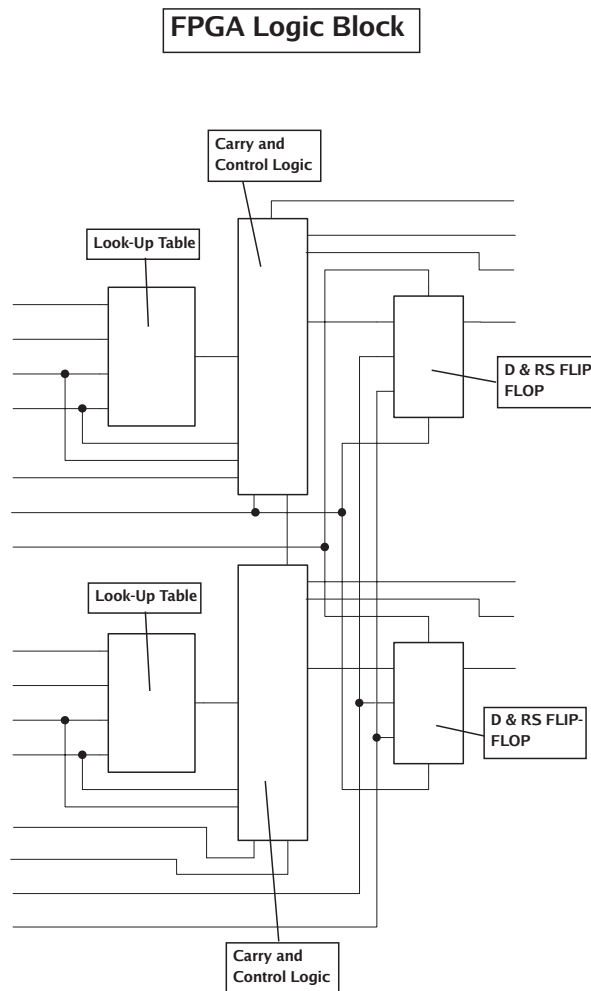


fig113c.eps

4.6. Application Specific Integrated Circuit: ASIC

CPLDs und FPGAs werden nach ihrer Herstellung anwendungsspezifisch konfiguriert.

► CPLDs und FPGAs besitzen eine vom Hersteller vorgegebene Grundstruktur und sind bezüglich maximaler Gatter-, Register- und Trassierungs/Verbindungskapazitäten begrenzt.

► Bei ASICs geht der Systementwurf, der mit einem digitalen Schaltkreis realisiert werden soll, in den Herstellungsprozeß des ICs ein, d.h. es werden am Ende der Logiksynthese die Logikgatter in Transistorzellen und schließlich auf physikalische Layoutebene abgebildet. Für den Fertigungsprozeß werden ähnlich der Leiterplattenherstellung Fertigungsmasken benötigt, die

- den Ort und den Aufbau der einzelnen CMOS-Transistoren bestimmen (mehrschichtiger Aufbau),
- die die Verbindungsleitungen auf den entsprechenden Metall-Layern festlegen und Durchkontaktierungen zwischen einzelnen Layern beschreiben.

Der anwendungsspezielle Systementwurf bestimmt daher maßgeblich den Fertigungsprozeß (eventuell auch die Fertigungstechnologie selbst), und muß für jedes neue Digitallogiksystem erneut entwickelt werden.

► Man unterscheidet:

Full Custom ASICs →

Transistor- und Verbindungsebenen sind frei vom Anwender konfigurierbar und spezifizierbar, setzt aber Wissen der Transistor- und Herstellungstechnologie voraus. Hoher Zeitaufwand und hohes Risiko von Fehlern kennzeichnet diese Entwicklungsmethode.

Motivation für diesen Entwicklungsprozeß:

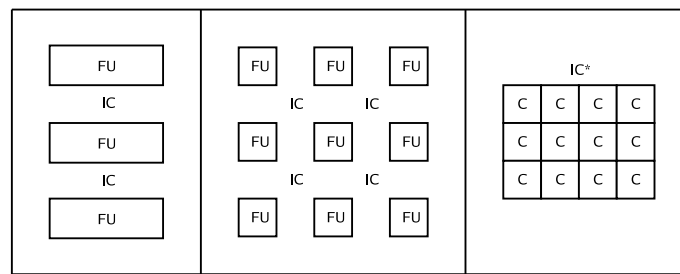
1. Möglichst geringe Chip-Fläche und geringer Energieverbrauch,
2. höchstmögliche Verarbeitungsgeschwindigkeit,
3. hohe Stückzahlen rechtfertigen hohe Entwicklungskosten,
4. Hardware-Entwürfe, die an die Grenzen des technologisch Machbaren gehen.

Standardzellen-ASICs →

Für Grundelemente wie Logikgattern, Register, FLIP-FLOPs, aber auch komplexe Komponenten wie Addierern, Multiplizierer, RAM usw. werden aus einer sog. Standardzellenbibliothek vorgefertigte Transistorblöcke angeboten, die "nur" noch strukturell und schließlich technologisch miteinander verbunden werden müssen.

Auch für die Trassierung existieren vorkonfigurierte Transistorblöcke und Strukturen. Bei dieser Entwurfsmethode werden keine tiefen Kenntnisse über Transistor- und Herstellungstechnologie vorausgesetzt, da diese als Expertenwissen in der Standardzellenbibliothek vorhanden ist.

Abbildung 92: A:
Zeilen-oder
Spaltenstruktur, B:
Blockstruktur, C:
Sea-Of-Gates



A. Zeilen oder Spaltenstruktur

B. Blockstruktur

C. Sea - of - Gates

fig114.eps

Sea-of-Gate- Technologie

► Die Sea-of-Gates-Technologie ist ein Sonderfall im Standardzellen-ASIC Bereich: hier werden die zur Implementierung erforderlichen Transistoren (immer ein n- und ein p-Kanal Transistor gepaart) in einer regulären Matrix angeordnet. Diese Transistormatrix wird dann ohne Verbindungen, d.h. unabhängig von einem Anwendungsentwurf, gefertigt. Diese vorverarbeiteten Chip-Wafer müssen dann nach der initialen Fertigung trassiert werden, d.h. in einem späteren Technologieprozeß werden Metall-Layer und Verbindungsleitungen dem Wafer hinzugefügt. Vorteil: die Basisstruktur kann unabhängig von einem Hardware-Design entwickelt und produziert werden, zu entsprechend niedrigeren Kosten als bei reinen ASIC-Entwürfen. Die Nachbearbeitung kann in entsprechenden Mikrosystem-Laboren vor Ort entsprechend dem anwendungsspezifischen Hardware-Entwurf erfolgen, und stellt keinen nennenswerten Kostenfaktor bei kleinen Stückzahlen dar. Ein Rapid-Prototyping ist mit dieser vorgefertigten Methode möglich, was sonst nur FPGAs vorbehalten ist.

5. Entwurfs- und Syntheseverfahren

Der Entwurfsprozeß gliedert sich in mehrere Stufen:

1. Systemebene →

Aufteilung des Gesamtsystems in Subsysteme. Man spricht von der Partitionierung in kooperierende Prozesse.

- Es liegt noch kein Zeitmodell, sondern nur ein Kausalitätsschema vor (Abhängigkeitsmodell).
- Spezifikation der Funktionalität

2. Algorithmische Ebene →

- Verhaltensbeschreibung
- Festlegung von Datenbreiten, globalen Speichergrößen und Befehlssätzen von Mikroprozessoren.
- Hardware-Software-Co-Design, d.h. Partitionierung des Gesamtentwurfs in einen hardwareabhängigen Teil (Mikroprozessoresysteme) und Software (Assemblercode).
- Einsatz von Hardware-Beschreibungssprachen zur Implementierung und Beschreibung der Algorithmik.
- Zeitmodell: verfeinertes Kausalitätsmodell.

3. Register-Transfer-Ebene →

- Mikroprogrammebene
- Aufteilung des Entwurfs von Algorithmen in Daten- und Kontrollfluß bzw. pfeilen.
- Einführung digitaler Komponenten:
 - Register/Speicher
 - Multiplexer
 - Arithmetische und logische Einheiten
- Zeitmodell: zeitdiskret bei synchronen Systemen, der Kontrollfluß ist takt- und zustandsgesteuert; Deadlines bei asynchronen Systemen.
- Festlegung von Größe und Anzahl von Registern und Verarbeitungswerken.

4. Logikebene →

- Schaltungsentwurf mit Makrobibliotheken von digitalen Komponenten, abhängig von der Zieltechnologie.
- Boolesche Minimierung kombinatorischer Logik.

➤ Sequenzielle Optimierung im Kontrollfluß (Z.B. Entfernung von nicht erreichbaren Zuständen eines Zustandsautomaten).

➤ Festlegung des zeitlichen Verhaltens (noch diskret): Verzögerungszeiten, Fan-In / FAN-Out, d.h. das zeitliche Verhalten von Logikgattern und Registern wird von der Art und signifikant von der Anzahl an Senken (weitere Logikgatter und Register) bestimmt, die am Ausgang eines Logikgatters angeschlossen sind. Gegebenfalls Einfügen von Pufferstufen in den Signalweg zur Verbesserung des Schaltverhaltens.

5. Schaltkreisebene (Transistorebene) → Logik →

➤ Netzwerke aus Transistoren

➤ Zeitmodell: kontinuierlich, Signal-Wertverlauf: kontinuierlich

6. Schaltkreisebene (Physikalische Layoutebene) →

➤ Konkretes Chip-Layout mit konkreten Abmessungen und Dimensionierung (nur bei ASICs).

➤ Beim Systementwurf gibt es einen zweidimensionalen Entwurfsraum, der aus der Systemlaufzeit (kleinste Periodendauer des Taktsignals bei einem synchronen System) und der Chip-Fläche (Anzahl von Logikgattern und Registern) gebildet wird. Das Ziel beim Systementwurf ist i.A. ein Kompromiss aus Minimierung der Systemlaufzeit und Chip-Fläche.

Abbildung 93:
Kriterien für den
Systementwurf.

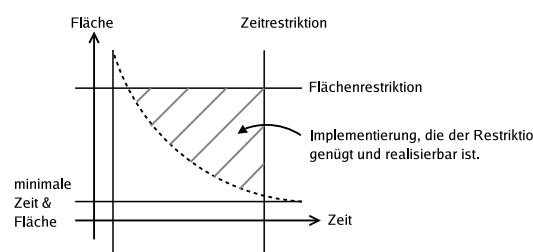
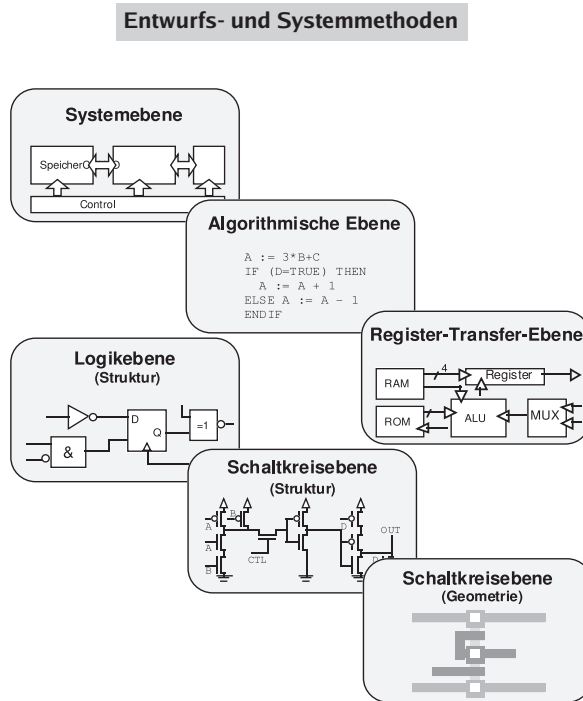


fig120.eps

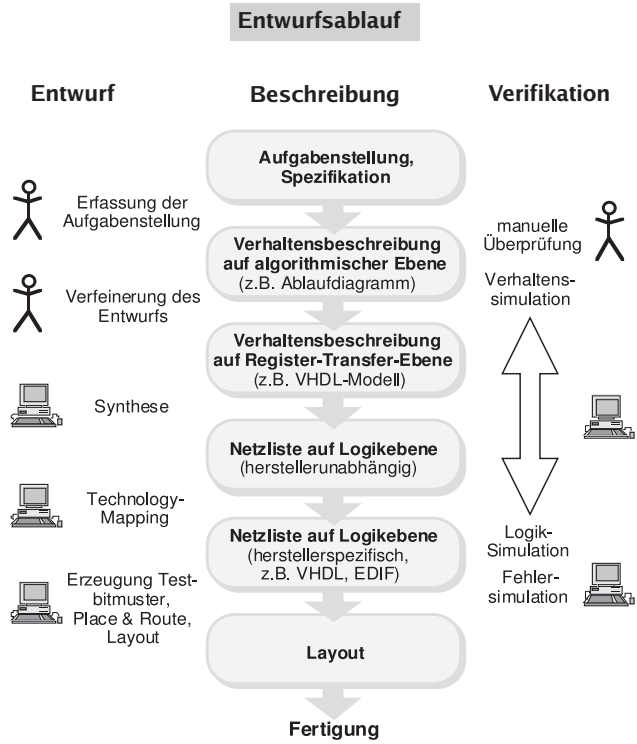
Abbildung 94:
Entwurfs- und Systemmethoden und Unterteilung in verschiedene Abstraktionsebenen.



© G. Lehmann/B. Wunder/M. Selz

fig120a.eps

Abbildung 95: Entwurfsablauf. Der Systementwurf wird durch Simulation und Verifikation rückgekoppelt. Dem eigentlichen Syntheseprozess schließt sich das sogenannte Technologie-Mapping an, welches eine Netzliste mit Logikkomponenten auf die Zielhardware abbildet. Simulation kann sowohl auf höherer Ebene vor der Synthese durch Verhaltensmodellierung, aber auch nach der Synthese auf Logikebene stattfinden.



© G. Lehmann/B. Wunder/M. Selz

fig120b.eps

- Bei der Hardware-Beschreibung unterscheidet man verhaltensbasierte und strukturelle Beschreibung bzw. Spezifikation.

Strukturelle Beschreibung (SB) →

Hier findet die Systembeschreibung mittels einer Netzliste von Komponenten, wie Logikgattern, Registern sowie komplexeren Einheiten wie Addierern oder RAM-Komponenten statt. Die Netzliste bestimmt die Verbindungen der einzelnen Komponenten, vergleichbar mit einem Schaltplan auf Logikebene.

Beispiel:

C:

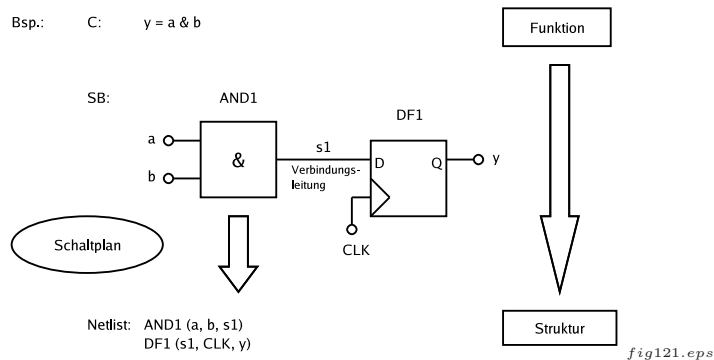
↓

$y = a \& b;$

↓

SB:

↓



Netzliste:

↓

AND1(a, b, s1)

DF1(s1, clk, y)

- In der strukturellen Beschreibung gibt es keine Kenntnis über das Verhalten einzelner Komponenten, wie z.B. logischer Schaltfunktionen, beschrieben z.B. mit einer Funktionstabelle.

Verhaltensbeschreibung (VB) →

Hier wird implizit logisches und zeitliches Verhalten bestimmt, aber nicht die technologische Umsetzung und die Verknüpfung von logischen Einheiten. Z.B. ist eine logische Funktionstabelle die VB, die abgeleitete boolesche Normalform die SB!

VB lässt sich mittels imperativer und funktionaler Sprachkonstrukte ausdrücken:

```
IF <b> THEN s1 ELSE s2;
FOR I = 1 TO 10
DO
S
DONE;
```

Beispiel eines
Übergangs von VB
zur SB.

VB mittels Funktionstabelle:

a	b	q
0	0	1
0	1	1
1	0	1
1	1	0

⇒

```
IF a = 0 AND b = 0 THEN q = 1
ELSE IF a = 0 AND b = 1 THEN q = 1
ELSE IF a = 1 AND b = 0 THEN q = 1
ELSE q = 0;
```

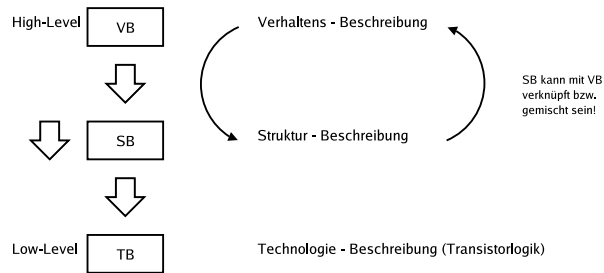
⇒

SB mit der Netzliste:

```
AND(a,b,temp)
NOT(temp,q)
```

‡ Anmerkung: logische Ausdrücke wie (a&b) lassen zwar sofort die logische Basisfunktion erkennen, sind aber aufgrund ihrer Notation eine VB!

Abbildung 96:
Hierarchischer
Aufbau mit VB und
SB.



5.1. Register-Transfer-Ebene

Übergang von algorithmischer Ebene auf RT-Beschreibung durch VB mit einem Daten- und Steuerfluß.

- ▶ Erzeugung einer RTL bedeutet das Abbilden von Daten- und Steuerfluß in zwei Dimensionen: Zeit und Fläche \equiv Hardware \equiv Logik.
- ▶ Der Datenteil bearbeitet die Eingangsdaten eines Systems, so daß die gewünschten Ausgangsdaten erzeugt werden.
- ▶ Der Steuerteil legt die Reihenfolge und Art der Datenoperationen fest.
- ▶ Die Eigenschaften eines RTL-Systems werden durch Operationen (arithmetische, relationale, boolesche, fallsensitive) und den Transfer der Daten zwischen Registern charakterisiert.
- ▶ Steuer- und Datenteil "kommunizieren" über Steuersignale miteinander (bzw. sind verknüpft mit).

RT-Implementierung

- ▶ Aufteilung:

1. Der **Datenteil** besteht aus einer Menge von Modulen: Addierer, Komperatoren, Multiplizierer, weiterhin Speicherelemente (Register, adressierbarer Speicher) und Datenselektoren (Multiplexer).
2. Der **Steuerteil** (Controller) wird in Form einer Tabelle symbolischer Zustandsübergänge beschrieben.
3. Die **Steuernetze** aktivieren oder adressieren Speicher, schalten Datenmultiplexer und aktivieren Datenoperationen.
4. Die **Bedingungsnetze** liefern Ergebnisse von Test-Ausdrücken für daten- oder signalabhängige Verzweigungen der Zustandsübergänge.

Abbildung 97:
Beispiel Umsetzung einer VB mit RTL durch Scheduling und Ressourcenallokation. Die Register R zwischen den einzelnen Stufen bilden eine Pipeline.

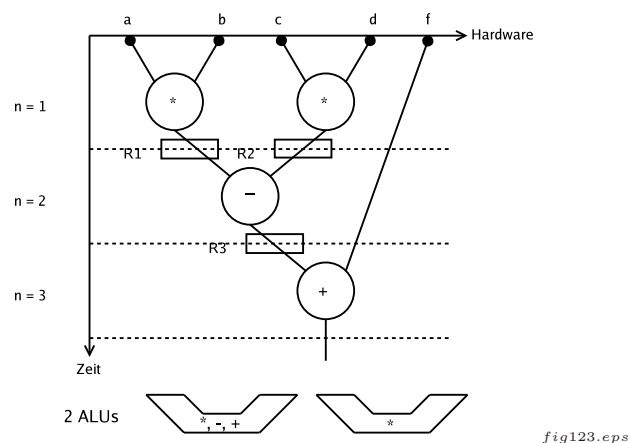


Abbildung 98:
Beispiel Umsetzung
einer VB mit RTL
durch Scheduling und
Ressourcenallokation
mit veränderten
Parametern.

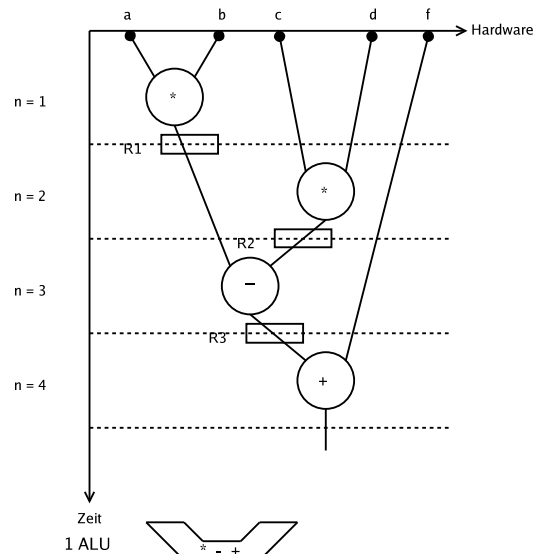


fig124.eps

RTL-Implementierung aus VB bedeutet:

Scheduling → Zuordnung von Operationen zu Zeitschritten. Optimierung: Minimierung der Zeitschritte.

Ressourcen-Allokation → Bestimmung von Typ und Anzahl der erforderlichen Hardware-Ressourcen wie Funktionselemente, Speicher, Busse. Optimierung: Minimierung der Funktionselemente, z.B. durch ALU-Blöcke, die über Multiplexern mit mehreren arithmetischen Operationen verwendet werden können (Ressource-Sharing).

Ressourcen-Zuweisung → Zuordnung von Funktionselementen zu einzelnen Instanzen und Operationen.

→ starke Wechselwirkung zwischen Scheduling und Ressourcen-Allokation.

Pipeline-Architektur

► Eine Pipeline-Architektur ergibt sich durch Einfügen von Registern zwischen den einzelnen Funktionseinheiten des Datenpfades. Diese Register erfüllen zwei Aufgaben:

1. Nur zu definierten und diskreten Zeitpunkten liegen neue Daten am Eingang einer folgenden Funktionseinheit an. Zwischen zwei Zeitpunkten t_0 und t_1 sind die Datensignale stabil. Aufgrund von verschiedenen Laufzeiten einzelner Signale in einer vorherigen Funktionseinheit sind die Ausgangssignale für eine bestimmte Zeit Δt als metastabil anzusehen, d.h. in diesem Zeitraum können einzelne Signalleitungen mehrfach ihren Logikzustand ändern. Diese Metastabilität würde sich ohne Register (Buffer) durch alle folgenden Funktionseinheiten fortpflanzen, was sich nachteilig auf die Leistungsaufnahme und das elektrische Verhalten der Digitallogik auswirken kann.

2. Besteht der Datenpfad aus N Funktionseinheiten, werden wenigstens N Taktzyklen benötigt, bis ein Ergebnis am Ausgang des Datenpfades anliegt, d.h. die Latenz ist $L \geq N$. Da die einzelnen Funktionseinheiten durch Register getrennt sind, können vorherige Funktionseinheiten schon früher neue Daten aufnehmen und verarbeiten, d.h. die sog. Restart-Periode ist $R < L$.

5.2. Hardware-Beschreibungssprachen (HDL)

HDLs führen keinen Übergang von VB zu RTL durch, d.h. die Systembeschreibung unter Verwendung einer HDL erfordert explizite Formulierung der RTL, anders als bei prozessorientierten Programmiersprachen wie C.

- ▶ HDLs ermöglichen sowohl SB als auch VB des Systems, häufig gemischt und hierarchisch.
- ▶ Eine HDL sollte technologieunabhängig sein, d.h. eine VB muß für FPGAs genauso wie für ASICs verwendbar sein, und muß die gleiche Funktionalität ergeben.
- ▶ Die Abstraktionsebene einer HDL ist durch VB höher als reine SB, was kürzere Entwicklungszeiten und eine höhere Entwurfssicherheit bedeutet.
- ▶ Es gibt zwei weit verbreitete HDLs:
 1. Verilog HDL → Gateway Design Automation [1985], Cadence [1989], Standardisierung durch IEEE [1995]
 2. VHSIC (Very High Speed IC) HDL → IBM, Texas Instruments [1982], IEEE [1985,1987]. VHDL ist weit verbreiteter Standard im akademischen und industriellen Umfeld.

VHDL bietet im Gegensatz zu Verilog HDL ein universelles Typensystem. VHDL ist semantisch streng typisiert.

5.3. VHDL

VHDL bietet ein

- universelles [konkrete und abstrakte Datentypen],
- umfangreiches [bit, signed, float, string, ...],
- erweiterbares [type definition],
- strenges [type a \neq type b]

Typsystem. Strenge Typisierung reduziert Entwurfsfehler, erhöht aber den Programmieraufwand.

► VHDL unterstützt statische Funktionen, die mit unterschiedlichen Funktionstypen überladen werden können.

Beispiel einer überladenen Funktion in VHDL.

```
function "+" (std_logic_vector;std_logic_vector)
    return std_logic_vector;
function "+" (std_logic_vector;integer)
    return std_logic_vector;
function "+" (signed;signed)
    return signed;
```

► VHDL ist modulatorientiert und hierarchisch strukturiert.

5.3.1. Aufbau einer VHDL-Entwurfsbeschreibung

Die Beschreibung einer Digitallogikschaltung, Komponente genannt, benötigt wenigstens zwei Strukturelemente. Ein Modul (\equiv eine VHDL Quelldatei) ist daher unterteilt in:

Schnittstellenbeschreibung \rightarrow Entity.

In der Entity wird die extern sichtbare Schnittstelle der zu modellierenden Komponente beschrieben:

- \rightarrow Ein- und Ausgänge (Ports)
- \rightarrow Konstanten
- \rightarrow Funktionen und Prozeduren

Architektur \rightarrow Architecture.

Eine Architektur enthält die Beschreibung der Funktionalität und/oder der Struktur eines Moduls:

- \rightarrow Verhaltenbeschreibung mit Sprachelementen ähnlich den von imperativen Programmiersprachen,

- \rightarrow Strukturbeschreibung SB in Form einer Netzliste, die auch verschiedene VHDL-Module verbinden kann.

\blacktriangleright Eine Entity bzw. ein Modul kann durch verschiedene Architekturen beschrieben werden. Die Zuordnung einer Architektur zu einer Entity findet in der sog. Konfiguration statt.

Package [optional] \rightarrow Ein Paket enthält Anweisungen wie Typ- und Objektdeklarationen sowie statische Funktionen und Prozeduren, die von mehreren Modulen verwendet werden können. Vordefiniert sind bei VHDL die Pakete STANDARD und TEXTIO.

Abbildung 99:
Hierarchisch
aufgebaute Struktur
in VHDL.

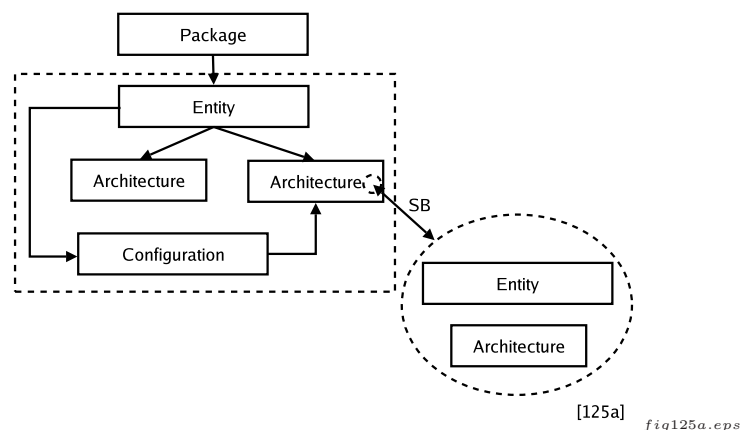
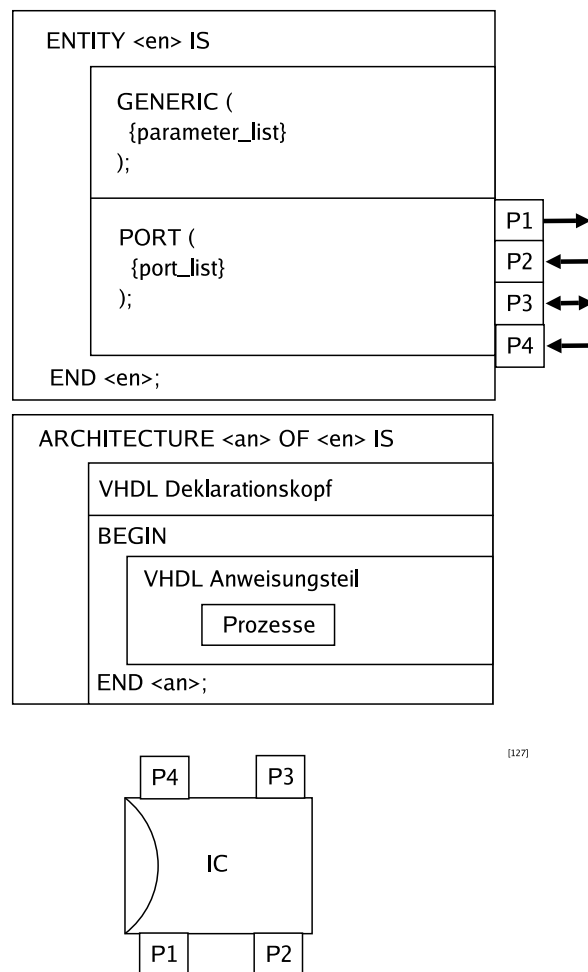


Abbildung 100:
Definition eines
VHDL-Modules und
Analogie zur einer
IC-Komponente.



- Die Port-Schnittstelle legt Typ und Richtung des Anschlüsse einer Entity fest. Die Architektur unterteilt sich in einen Deklarationsteil und einen Körper, der die eigentlichen VHDL-Anweisungen (VB & SB) aufnimmt.
- Wie bei elektrischen Schaltkreisen können verschiedene Module gleichen oder verschiedenen Typs strukturell miteinander verbunden werden.
- Diesen Vorgang der Einbindung bezeichnet man als Komponenteninstanzierung. Eine externe Komponente \equiv Modul muß im VHDL-Deklarationskopf gemäß ihrer Anschlüsse deklariert werden, und kann im VHDL-Anweisungsteil (Architektur-Körper) eingebunden werden (SB).

Abbildung 101:
Komponenteninstanz-
zierung in
VHDL.

```
entity Und is
port(
  a: in bit;
  b: in bit;
  q: out bit);
end Und;

architecture Logic of Und is
begin
  q <= a & b;
end Logic;
```

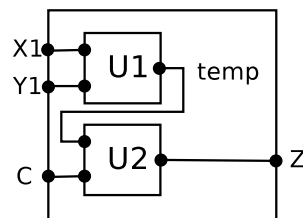


fig128.eps

```
entity Circuit is
port(
  x1: in bit;
  y1: in bit;
  c: in bit;
  z: out bit);
end Circuit;

architecture Main of Circuit is
component Und
port(
  a: in bit;
  b: in bit;
  q: out bit);
end component;
signal temp: bit;
begin
  U1: Und port map(x1,y1,temp);
  U2: Und port map(temp,c,z);
end Main;
```

5.3.2. Signale

- ▶ Signale sind die eigentlichen Datenobjekte in VHDL.
- ▶ Der einfachste Signaltyp ist ein Bit mit der Wertemenge $\text{bit}=\{0,1\}$.
- ▶ Ein Signal kann aus einer Bitgruppe \equiv Datenwort mit beliebiger Anzahl von Einzelsignalen bestehen, genannt Vektor.
- ▶ Ein Signal kann aber auch von einem abstrakten Datentyp sein. Hier ist es dem Synthese-Werkzeug überlassen, eine Synthese auf einen logischen Datentyp durchzuführen (sofern möglich).
- ▶ Synthese von Signalen erfolgt durch
 - reine Verknüpfung bzw. Verbindung zwischen einzelnen Komponenten eines Systems ("wire")
 - durch Erzeugung eines Registers oder Latches.
- ▶ Signale können in der Port-Deklaration global sichtbar oder innerhalb des Architektur-Deklarationsteils lokal sichtbar erzeugt werden.

Definition eines globalen Port-Signals und Angabe der Signalrichtung.

```
<signalname> : <dir> <type>;
<dir> = {in | out | inout | buffer}
```

Beispiel einer Port-Deklaration.

```
port (data : in bit;
      addr : in bit_vector(7 downto 0));
```

Definition eines lokalen Architektur-Signals ohne Angabe der Signalrichtung aber mit Möglichkeit der Initialisierung.

```
signal <signalname> : <type> (<:= <init>);
```

Beispiel einer Architektur-Deklaration.

```
architecture main of en is
  signal write: bit;
begin
  write <= '0';
end;
```

- ▶ Signal-Vektoren der Datenbreite N werden wie Signale definiert. Der Typbezeichner wird um "_vector" ergänzt. Die Angabe des Index-Intervalls kann auf zwei Arten erfolgen, entsprechend der Stelle des höchstwertigsten Bits (MSB). Wird die Indexrichtung $\langle b \text{ downto } a \rangle$ verwendet, befindet sich das MSB immer auf der linken Seite einer Binärzahl $\langle M \bullet \bullet L \rangle$, wird die Indexrichtung $\langle a \text{ to } b \rangle$ gewählt, auf der rechten Seite

**Defintion eines
Signal-Vektors.**

(L••M). Der untere Index a kann mit jedem beliebigen natürlichen Zahlenwert gewählt werden. Der Wert 0 mit der $\langle b \text{ downto } a \rangle$ -Darstellung ist aber zweckmäßig anzuwenden.

```
signal <svname> : <type>_vector( <b> downto <a> );
signal <svname> : <type>_vector( <a> to <b> );
```

**Defintion:
Signalzuweisung.**

► Signalen können Werte zugewiesen werden, ähnlich einer Wertzuweisung einer Variable in imperativen Programmiersprachen. Beim Typ `bit` findet eine Zuweisung eines Logikpegels statt!

```
<lname> <= <expr>;
mit
<expr>:
  [Konstante] → vom gleichen Typ wie <sig>
                → Bit: '0' | '1'
                → Bitvector: "XX•••" mit X = {0,1}
  [Arithmetische Ausdrücke]
                → <OP1> {+,-,*,•••} <OP2>
                → Schachtelung expr1(expr2(expr3(•••)))
                → alle Operanden vom gleichen Typ
                → nur sinnvoll mit Vektoren
  [Logische Ausdrücke]
                → <OP1> {and,or,•••} <OP2>
```

```
architecture demo of en is
  signal a,b: bit_vector(3 downto 0);
begin
  a <= b + "0001";
  Typ bit_vector ←
  a <= b + bit_vector(to_unsigned(1));
  Typ-Konversion für Typ natural ←
  a <= a + b + "000" & '1';
  Bit-Verknüpfung ←
end demo;
```

Beispiele für Signalzuweisungen und Ausdrücke.

► Nebenläufige Signalzuweisungen, daß sind alle Toplevelanweisungen im Architektur-Körper, dürfen immer nur eine Quelle besitzen, d.h. in obigen Beispiel ist immer nur einer Signalzuweisung erlaubt!

**Vordefinierte (V)
und erweiterte (E)
Typen in VHDL**

►

V: BOOLEAN → {true,false}

V: integer → {keine spezifische Datenbreite, i.A. wird aber nur eine Wertemenge von 32/64 Bit unterstützt}

V: natural → {⊆ integer ≥ 0}

V: positive → {⊆ integer > 0}

V: bit → {0,1}

V: bit_vector(natural) → ≡ Array of bits

V: character → {7 bit ASCII}

V: string(natural) → ≡ Array of characters

E: std_logic → {0,1,H,L,U,Z,-}, muß über das Paket IEEE.STD_LOGIC_1164.ALL eingebunden werden.

E: std_logic_vector → ≡ Array of {0,1,H,L,U,Z,-}, muß über das Paket IEEE.STD_LOGIC_1164.ALL eingebunden werden.

E: signed, unsigned → muß über das Paket IEEE.NUMERIC_BIT.ALL eingebunden werden.

Abstrakte konstante Aufzählungstypen und Definition eines Signals dieses abstrakten Typs.

```
type <name> is (
  {ID,}
);
signal <sig> : <typename>;
```

► Abstrakte Aufzählungstypen werden z.B. zur symbolischen Beschreibung der Zustände eines Zustandsautomaten verwendet.

```
type states is (
  S_start,
  S_loop,
  S_end
);
signal state: states;
signal next_state: states;
```

Beispiel eines Aufzählungstyps.

► VHDL unterstützt zur einsortigen Aggregation Arrays. Der Arraytyp kann beliebig sein. Arrays können wie RAM-Blöcke mit adressierbaren Speicherzellen aufgefasst werden. Ob jedoch ein monolithischer RAM-Block während der Synthese erzeugt werden kann, hängt von der Verwendung eines Arrays ab. Wenn nebenläufig z.B. mehrere Arrayzellen gelesen werden, ist die Inferenz eines klassischen RAM-Speicher mit nur einem Leseport nicht mehr möglich! Arrays werden wie generische Typen aufgefasst, anders als in imperativen Programmiersprachen wie C.

Definition eines Arraytyps, Erzeugung eines Arrayobjekts und Zugriff auf Arrayzellen.

```
type <aname> is array({range}) of <celltype>;
signal <sname>: <aname>;
...
```

<name>(<index>)

```
type ram is array(0 to 255) of bit_vector(7 downto 0);  
signal ram1: ram;  
signal a: bit_vecotr(7 downto 0);  
•••  
a <= ram1(17);
```

Beispiel für Arrays.

5.3.3. Strukturelle Modellierung

► Die strukturelle Modellierung unterscheidet zwischen der Komponentendefinition, der Verhaltensbeschreibung einer Komponente (Entity), und der Komponenteninstanziierung, d.h. die Einbindung von Komponenten. Die einzubindenden Komponenten müssen mit ihrer Port-Schnittstelle (identisch zum Entity-Port) im Deklarationsteil der Architektur eingeführt werden.

► Eine Komponente eines bestimmten Typs kann beliebig oft dupliziert im Architekturkörper eingebunden werden. Jede Komponenteninstanz ist unabhängig von jeder anderen, d.h. mit eigener Digitallogik synthetisiert.

```
architecture <arch> of <entity> is
  component <com>
    port({<signame>: <dir> <type>;});
  end component;
  ...
begin
  <instname1>: <com> port map
    ({<signame> => <sig>,});
  <instname1>: <com> port map
    ({<signame> => <sig>,});
  ...
end <arch>;
```

5.3.4. Verhaltensmodellierung und Prozesse

- ▶ Bei der VB wird das Verhalten einer Komponente durch die Reaktion der Ausgangssignale auf Änderungen der Eingangssignale beschrieben. Bei einer reinen VB findet keine weitere Verzweigung in Unterkomponenten statt.
- ▶ In der VB werden zwei Anweisungsklassen unterschieden:
 1. Sequenzielle Anweisungen
 2. Nebenläufige Anweisungen
- ▶ Sequenzielle Anweisungen können nur in Prozessen verwendet werden. Sequenzielle Anweisungen einer Hardware-Beschreibungssprache sind nicht mit Übergängen von Zustandsautomaten zu verwechseln. Eine Sequenz ist im Sinne einer Laufzeithierarchie zu verstehen.
- ▶ VHDL synthetisiert Register (FLIP-FLOPs, Latches) implizit, d.h. es gibt in VHDL keine Möglichkeit, ein Register explizit zu erzeugen. Die Register-Inferenz ergibt sich aus der Verhaltenbeschreibung mit entsprechenden VHDL-Anweisungen.
- ▶ Register können nur in sog. Prozessen erzeugt werden.
- ▶ Ein Prozeß ist eine Blockkapselung von VHDL-Anweisungen und gehört zur VB. Prozesse werden zusammen mit Toplevelanweisungen außerhalb von Prozessen konkurrierend/nebenläufig ausgeführt.
- ▶ Anweisungen in einem Prozeß sind auf gleicher Ebene zueinander nebenläufig.

Abbildung 102:
Beispiel
Multiplexer-Inferenz
mit Signalzuweisung
in beiden Zweigen der
bedingten
Verzweigung.

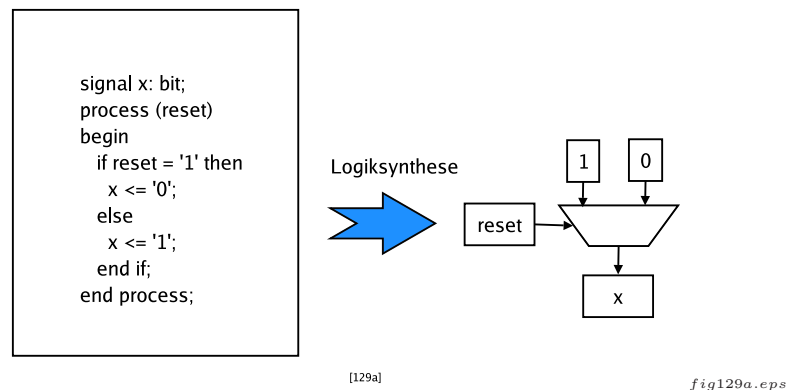
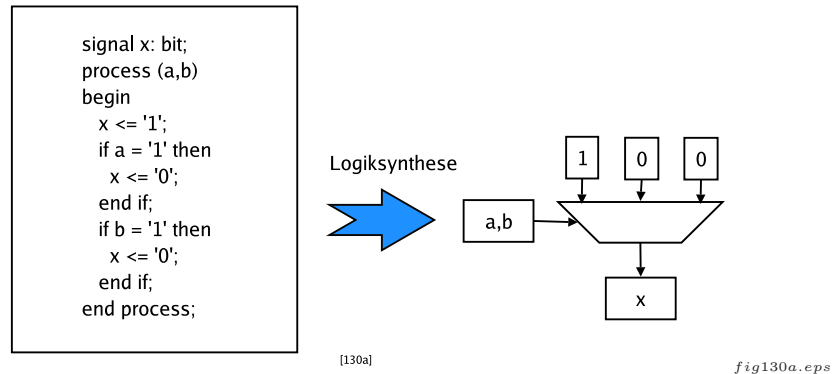


Abbildung 103:
Beispiel
Multiplexer-Inferenz
mit Default-
Signalzuweisung und
unvollständiger
bedingter
Verzweigung.

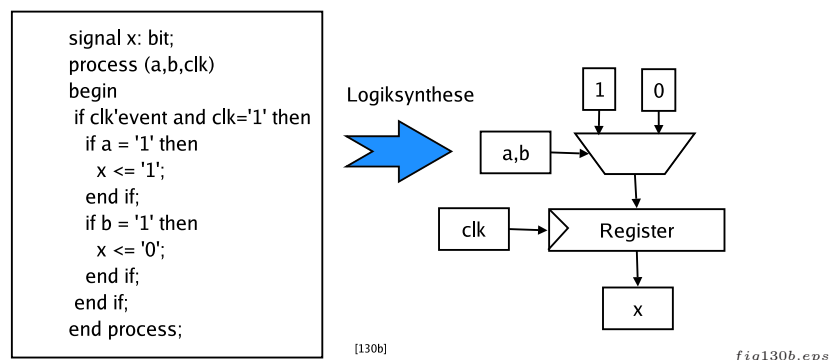


► Unvollständige bedingte Verzweigungen können (und sollen!) Register erzeugen. Immer wenn es einen bedingten Fall in einem Prozeß gibt, der keine Zuweisung eines bestimmten Signals gibt, ist gemäß dem zu grundlegenden Modell ein Register als Zustandsspeicher erforderlich. Dabei muß man zwischen taktflanken gesteuerten (dynamischen) und statischen Registern unterscheiden.

◆ Statisch
if <clk> = <level> then
 <reg> <= <expr>;
end if;
◆ Dynamisch
if <clk>'event and <clk> = <level> then
 <reg> <= <expr>;
end if;

Registererzeugung in VHDL

Abbildung 104:
Beispiel
Register-Inferenz mit
unvollständiger
bedingter
Verzweigung.



Prozeßmodellierung

► Prozesse modellieren prozedurale Vorgänge. Das Modell eines prozeduralen Verhaltens sieht neben Ablaufsteuerung die Aktivierung und Suspendierung des Ablauf eines Prozesses vor.

► Toplevel-Anweisungen sind im Gegensatz zu Prozessen immer aktiv, worin begründet ist, daß Toplevel-Anweisungen keine Register modellieren können, da diese nur unter bestimmten Bedingungen aktiv sind, d.h. Daten für die Speicherung übernehmen können.

► Prozesse werden durch zwei verschiedene Möglichkeiten aktiviert, die sich gegenseitig ausschliessen:

1. Liste sensitiver Signale im Prozeßkopf.

↳ Immer wenn sich eines dieser Signale ändert, wird der Prozeß aktiviert.

2. Wait-Anweisung

↳ Der Prozeß wird aktiviert, wenn die Wait-Anweisung (boolescher Ausdruck) erfüllt ist.

► Prozesse bestehen aus einem Deklarations- und Anweisungsteil (Körper).

Prozeßdefintion: A.)
mit
Empfindlichkeitsliste
zur Aktivierung, B.)
mit wait-Anweisung
und drei
Alternativen.

```
(A)
<<label>:) process ({<sig>,{}}
  <<variabledekl.>)
begin
  { Anweisungen }
end process < <label> );
```

◆

```
(B)
<<label>:) process ()
  <<variabledekl.>)
begin
  { Anweisungen }
  wait <expr>;
end process <<label>;
```

◆

```
wait on <signal>    ⇒ Signal Liste
   | until <expr>  ⇒ Bedingte Ausführung
   | for <time>    ⇒ Zeitverzögerung
```

Variablen

► In Prozessen können lokal sog. Variablen verwendet werden. Eine Variable ist ein Signal, aber mit einem anderen zeitlichen Verhalten als herkömmliche Signale (bezogen auf Prozeßabarbeitung):

↳ Variablenwerte werden sofort bei der Abarbeitung der Anweisungen zugewiesen (während Prozeß noch aktiv ist)

↳ Signalwerte werden erst nach der Abarbeitung des Prozesses nach einem sog. Delta-Zyklus zugewiesen.

► Konsequenz: Ein in einem Prozeß zugewiesener Signalwert kann nicht unmittelbar in folgenden Ausdrücken verwendet werden, im Gegensatz zu Variablen. Die Schreibweise der Variablenzuweisungen unterscheidet sich daher von der von Signalen.

Einsatz von Variablen in Prozessen für temporäre Ausdrücke.

```

signal y: integer;
proces (a,b)
  variable v1,v2: integer;
begin
  v1 := 3 * a + 7 * b;
  v2 := a * b + 5 * v1;
  y <= v1 + 2;
end process;

```

Bedingte Verzweigungen

► In Prozessen können Anweisungsblöcke bedingt nach Auswertung eines Ausdrucks oder durch Mehrfachauswahl ausgeführt (selektiert!) werden. Bei der bedingten Verzweigung ist der Else-Zweig wie bereits beschrieben optional. Der Ausdruck muß vom Typ boolean sein. Jede Anweisung wird mit einem Semikolon abgeschlossen, auch vor dem Else-Zweig (im Gegensatz zu C)!

Bedingte Verzweigung in Prozessen. Bei der Schachtelung muß ein verändertes Schlüsselwort elsif verwendet werden.

```

if <expr> then
  {Anweisungen;}
<else
  {Anweisungen;}
end if;
◆
if <e1> then
elsif <e2> then
...
elsif <en> then
else
end if;

```

Mehrfachauswahl mit optionalen Restmengenfall others .

```

case <expr> is
  {when <val_n> =>
  {Anweisungen;}}
  {when others =>
  {Anweisungen;}}
end case;

```

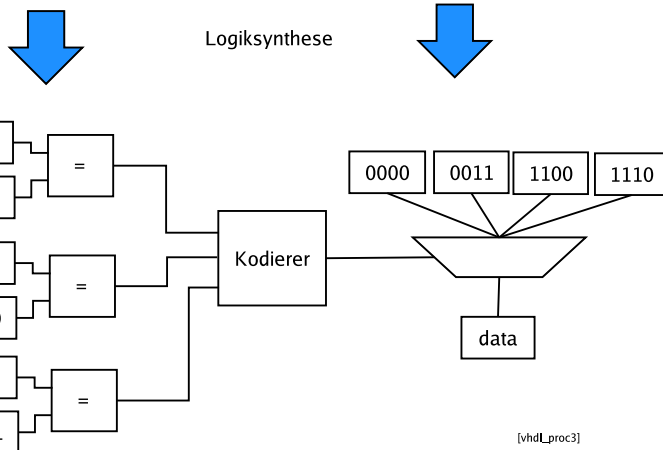
► In imperativen Programmiersprachen sind geschachtelte IF-THEN-ELSE Anweisungen und die Mehrfachauswahl isomorph bzw. äquivalent. Dies gilt nicht für eine Hardware-Beschreibungssprache. Beide Arten der Verzweigungen besitzen unterschiedliche Prioritätsstruktur und zeitliches Verhalten:

CASE → hat für alle Fälle konstante Signalverzögerungen bzw. Laufzeiten.

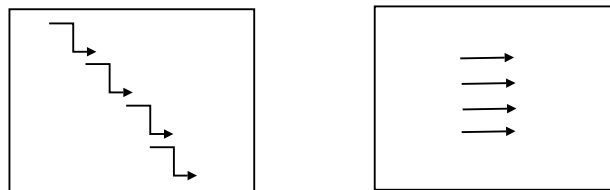
IF-THEN-ELSE → besitzt aufsteigende Laufzeiten von Signalen für die verschiedenen Fälle.

Abbildung 105:
Beispiel für bedingte Verzweigung und Mehrfachauswahl. Die Synthese beider Beschreibungen kann unterschiedliche Ergebnisse liefern.

<pre> signal addr,data: bit_vector(3 downto 0); process (addr) begin if addr = "0001" then data <= "0011"; elsif addr = "0010" then data <= "1100"; elsif addr = "0111" then data <= "1110"; else data <= "0000"; end if; end process; </pre>	<pre> signal addr,data: bit_vector(3 downto 0); process (addr) begin case addr is when "0001" => data <= "0011"; when "0010" => data <= "1100"; when "0111" => data <= "1110"; when others => data <= "0000"; end case; end process; </pre>
---	---



Laufzeit des Kodierers für die einzelnen Fälle



vhdl_proc3.eps

Schleifen

► In VHDL lassen sich ähnlich wie in imperativen Programmiersprachen Anweisungsblöcke mit Schleifen iterativ wiederholen.

↳ Man unterscheidet zwischen

1. statischen Schleifen, die zur Synthesezeit die Blockanweisungen abrollen, d.h. jeder Schleifendurchlauf erzeugt neue Digitallogik, und daher feste Indexgrenzen fordern, beschränkt auf Zählschleifen,
2. dynamische Schleifen, deren Indexgrenzen oder Durchlaufbedingung

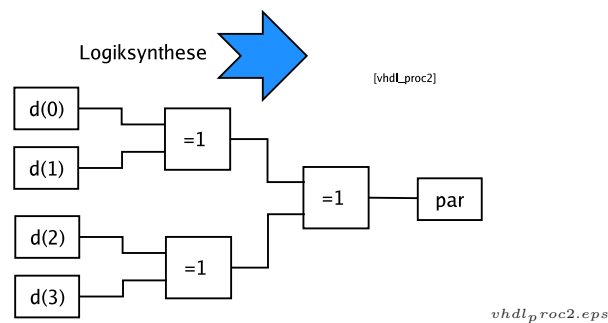
Statische
Zählschleifen in
VHDL. Der
Schleifenindex muß
nicht deklariert
werden.

Abbildung 106:
Beispiel statische
Zählschleife und
Synthese in
kombinatorische
Logik.

erst zur Laufzeit evaluiert. Diese Schleifen benötigen i.A. Zustands-
automaten für die (echte) sequentielle Abarbeitung. Diese Schleifen
werden i.A. von Synthesewerkzeugen nicht unterstützt.

```
<label>: for <index> in <range> loop
  {Anweisungen;}
end loop <label>;
<range>:
<b> downto <a>
<a> to <b>
```

```
signal d: bit_vector(3 downto 0);
process (d)
  variable par: boolean;
begin
  par := false;
  for i in 3 downto 0 loop
    if d(i) = '1' then
      par := not par;
    end if;
  end loop;
end process;
```

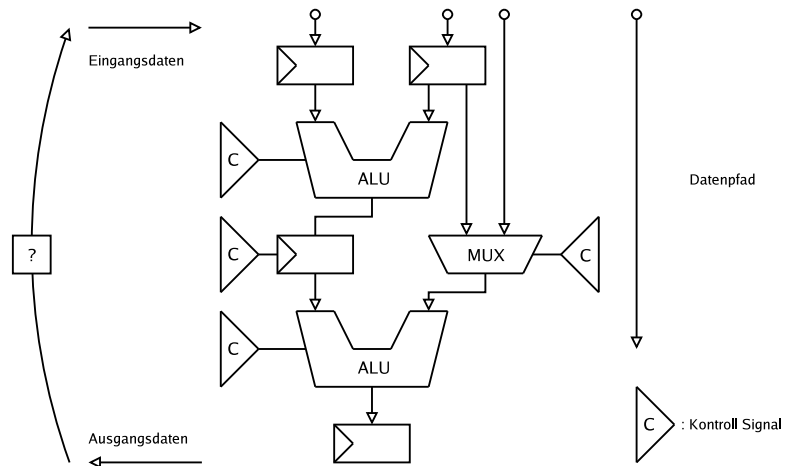


6. RTL und Zustandsautomaten

Zusammenfassung

- Der Bereich von digitalen Systemen reicht von steuerungs- bis zu datenintensiven Systemen.
- Systeme mit Steuerungsschwerpunkt sind rein reaktive Systeme die auf äußere Ereignisse reagieren.
- Systeme mit Datenschwerpunkt benötigen Datenverarbeitung mit hohen Datendurchsatz, wie. z.B. im Bereich der digitalen Signal- und Bildverarbeitung.
- Der Kontrollfluß ist sequenziell aufgebaut, d.h. in mehrere zeitlich getrennte Einzelschritte zerlegt.
- Wie in vorherigen Kapiteln gezeigt wurde, kann ein sequenzielles System in einen Kontroll- und Datenfluß aufgeteilt werden, mit Steuerungs- und Datenpfadeinheiten.
- Datenpfade enthalten
 1. arithmetische und logische Operatoreinheiten (ALUs),
 2. Logik für den Datentransport zwischen einzelnen Datenpfadeinheiten bzw. Stufen,
 3. Register für die Datenzwischenspeicherung und zum Aufbau von Pipelines, die dadurch gekennzeichnet sind, daß zu einem Zeitpunkt t sich mehrere Datensätze in unterschiedlichen Bearbeitungsstufen befinden können (Erhöhung des Datendurchsatzes).

Abbildung 107:
Datenpfade mit RTL



Endliche Zustandsautomaten

- Datenpfade werden mit endlichen Zustandsautomaten gesteuert (FSM: Finit State Machine).
 - ➔ Steuerungseinheit \equiv FSM
 - ➔ Datenpfadeinheiten können zyklisch wechselnde Datensätze, d.h. Datenströme, bearbeiten.

Partitionierte Sequenzielle Maschinen

Abbildung 108:
Allgemeine
Architektur und
Strukturierung einer
Sequenziellen
Maschine.

Zustandsautomat

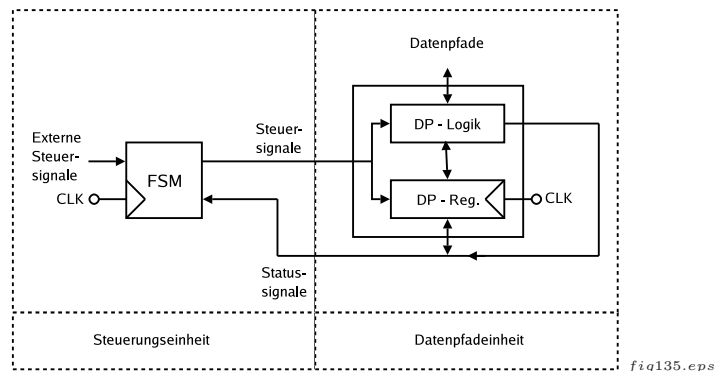
➔ Der Zustandsautomat muß die einzelnen Operationen im Datenpfad steuern und koordinieren:

1. Datentransport (Multiplexer)
2. Datenspeicherung (Register)
3. Operationen auf Daten selektieren (ALU \oplus Multiplexer)

➔ Datenpfade bestehen aus einer Vielzahl von regulären Strukturen (einfach zu optimieren),

➔ Kontrolleinheiten (FSM) bestehen i.A. aus irregulären Strukturen ("zufällig" strukturierte Logik - schwierig/aufwendig zu optimieren).

➤ Partitionierung einer sequenziellen Maschine (z.B. Mikroprozessor) in einen Daten- und Steuerungspfad macht die Architektur deutlicher und strukturierter, und vereinfacht den System/Hardware-Entwurf.



➤ Ein Zustandsautomat ist ein allgemeines sequenzielles System, dessen Reaktionen und Ausgangswerte außer vom aktuellen Zustand auch von den Eingangsgrößen abhängen.

➤ Man unterscheidet im wesentlichen zwei verschiedene Typen von Automaten:

Moore-Automat → Bei diesem Automaten hängen die Ausgangssignale nur vom aktuellen Zustand ab, welcher von Eingangssignalen aus der Vergangenheit und vorherigen Zuständen bestimmt wurde.

Mealy-Automat → Bei diesem Automaten hängen die Ausgangssignale zusätzlich von den Eingangssignalen ab.

Abbildung 109:
Blockschaltbild eines
allgemeinen
Automaten

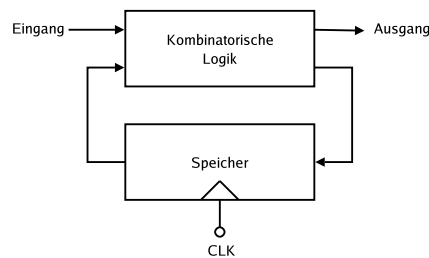


fig136.eps

Der Moore-Automat

- ▶ Der Eingangsvektor $E_t = E_1 \dots E_M$ bezeichnet die Gesamtheit der Eingangsgrößen, den Signalen E_i .
- ▶ Der Zustandsvektor $Z_t = Z_1 \dots Z_N$ bezeichnet den inneren Zustand des Systems. Bezeichnung innerer Zustand durch Umstand daß Signale Z nach außen nicht direkt sichtbar sind.
- ▶ Der Ausgangsvektor $A_t = A_1 \dots A_P$ beschreibt die Gesamtheit der Ausgangsgrößen, die Signale A_i .
- ▶ Index t und $t+1$ bezeichnen aktuellen und nächsten Zustand bzw. Vektorwert.

Abbildung 110:
Architektur des
Moore-Automaten.

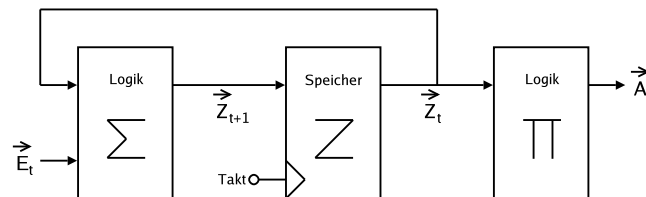


fig137.eps

- ▶ Die kombinatorische Logik Σ bestimmt den nächsten Zustand des Systems:

$$Z_{t+1} = \Sigma(Z_t, E_t) \quad (35)$$

- ▶ Die zweite kombinatorische Logik Π bestimmt den Ausgangsvektor zum Zeitpunkt t :

$$A_t = \Pi(Z_t) \quad (36)$$

- ▶ Der Entwurf des Zustandsautomaten reduziert sich auf den Entwurf der Zustandsübergangslgik Σ , da Π nicht in der Rückführungsschleife liegt.
- ▶ Der Zustandsvektor ist binär kodiert. Jedem Zustand ist ein eindeutiger Binärwert zugeordnet. Es gibt verschiedene Kodierungsmethoden:

Gewichtete Binär-Kodierung → Die Zustände Z_i werden fortlaufend mit einer natürlichen Zahl kodiert:

Zustandskodierung

$Z_1 \rightarrow 1 \rightarrow 0001$
 $Z_2 \rightarrow 2 \rightarrow 0010$
 $Z_3 \rightarrow 3 \rightarrow 0011 \dots$

➔ Mit N D-FLIP-FLOPs lassen sich 2^N Zustände repräsentieren.

One-Hot-Kodierung → Die Zustände Z_i werden jeweils einem eigenen Bit im Binärvektor zugeordnet.

$Z_1 \rightarrow 1 \rightarrow 0001$
 $Z_2 \rightarrow 2 \rightarrow 0010$
 $Z_3 \rightarrow 3 \rightarrow 0100 \dots$

➔ Mit N D-FLIP-FLOPs lassen sich nur N Zustände repräsentieren.

➔ Erwartung: Auswertungen der Zustandsübergänge Σ können i.A. mit einfacheren Schaltnetzen realisiert werden. Jedoch haben aktuelle Untersuchungen gezeigt, daß dieser Vorteil gegenüber gewichteter Binär-Kodierung i.A. nicht gegeben ist!

Gray-Kodierung → Beim Übergang zwischen Zuständen können in der kombinatorischen Logik Σ/Π metastabile Zustände auftreten ("Hazards"), die durch einschrittige Kodierung vermieden werden können, d.h. daß bei aufeinanderfolgenden Zustandsvektoren sich immer nur ein Bit ändert. Weiterhin führt Verringerung der Anzahl der wechselnden Bits im Zustandsvektor zu einer Verringerung der elektrischen Leistungsaufnahme der Digitallogikschaltung. Nur gültig bei aufeinanderfolgenden Zustandsübergängen.

Zustandsübergänge

➤ Die Zustandsübergänge werden mittels Zustandsdiagrammen (State-Transition-Diagramm) dargestellt. Jeder Zustand und jeder Übergang wird getrennt eingezeichnet. Die Bedingung E für einen Zustandswechsel wird am Übergangspfeil vermerkt. Innerhalb des Zustandssymbol wird der Zustand und der damit verknüpfte Ausgangsvektor angegeben.

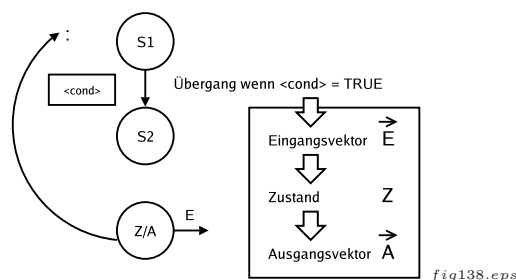


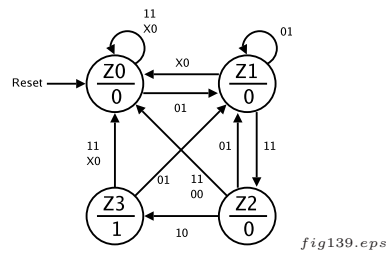
Abbildung 111: Zustandsdiagramme.

Beispiel: Bitfolgenkennung

➤ Es soll eine Bitfolge auf vorgegebene Muster untersucht werden.

Eingangsvektor $E = \{00, 01, 10, 11\}$
 Ausgangsvektor $A = \{0, 1\}$

Abbildung 112:
Zustandsdiagramm
für
Bitfolgenerkennung.



**VHDL-
Implementierung**



1. Partitionierung:

- Zustandsübergang und Speicherung von Z
- Kontrollpfad Σ
- Datenpfad Π

2. Zustandskodierung: abstrakt

```
type states is (Z0,Z1,Z2,Z3);
signal state,state_next: states;
```

3. Aufteilung des Zustandsautomaten in drei Prozesse:

- (a) process state_trans
- (b) process sigma
- (c) process pi

```
state_trans: process(clk,reset,state_next)
begin
  if clk'event and clk='1' then
    if reset='1' then
      state <= Z0;
    else
      state <= state_next;
    end if;
  end if;
end process state_trans;
◆
sigma: process(state,E)
begin
  case state is
    when Z0 =>
      if E="01" then state_next <= Z1;
```

```
        else state_next <= Z0;
        end if;
    when Z1 =>
        if E="11" then state_next <= Z2;
        elsif E="01" then state_next <= Z1;
        else state_next <= Z0;
        end if;
    when Z2 =>
        if E="10" then state_next <= Z3;
        elsif E="01" then state_next <= Z1;
        else state_next <= Z0;
        end if;
    when Z3 =>
        if E="01" then state_next <= Z1;
        else state_next <= Z0;
        end if;
    end case;
end process sigma;
◆
pi: process (state)
begin
    case state is
        when Z3 => A <= '1';
        when others => A <= '0';
    end case;
end process pi;
```

Drei-Prozeß Implementierung des Moore-Zustandsautomaten.

Literatur

- [CIL03] Michael Ciletti
Advanced Digital Design with Verilog HDL
Prentice Hall, 2003
- [REI03] Jürgen Reichardt, Bernd Schwarz
VHDL-Synthese. Entwurf digitaler Schaltungen und Systeme
Oldenbourg, 2003
- [HER04] Göran Herrmann, Dietmar Müller
ASIC - Entwurf und Test
Hanser, 2004
- [KOR04] Ralf Kories, Heint Schmidt-Walter
Taschenbuch der Elektrotechnik. Grundlagen und Elektronik
Harri Deutsch, 2004
- [LEE06] Sunguu Lee
Advanced Digital Logic Design. Using VHDL, State Machines, and Synthesis for FPGAs
Brooks Cole, 2006
- [WAN98] Markus Wannemacher
Das FPGA-Kochbuch
MITP, 1998
- [BAT02] Andrew Bateman, Iain Paterson-Stephens
The DSP-Handbook. Algorithms, Applications and Design Techniques
Prentice Hall, 2002
- [PRO96] John G. Proakis, Dimitris G. Manolakis
Digital Signal Processing
Prentice Hall, 1996
- [ISE98] Rolf Isermann
Digital Control Systems: Fundamentals, Deterministic Control
Springer, 1998
- [KUN01] Martin V. Künzli
Vom Gatter zu VHDL. Eine Einführung in die Digitaltechnik
vdf Hochschulverlag, 2001
- [ROW04] Chris Rowen
Engineering the complex SOC
Prentice Hall, 2004
- [WES04] Neil H. E. Weste, David Harris
Cmos Vlsi Design: A Circuits and Systems Perspective
Addison-Wesley, 2004
- [MEI98] Christoph Meinel, Thorsten Theobald
Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications
Springer, 1998