Entwurf Eingebetteter Systeme

Digitallogik und Programmierbare Logik

Stefan Bosse
University of Bremen
15.1.2018
sbosse@uni-bremen.de

1. Inhalt

1. Inhalt 3
2. Überblick 3
2.1. Grunddaten 4
2.2. Inhalte und Themen 4
2.3. Zentraler Inhalte 5
2.4. Fragestellung 5
2.5. Ziele 6
2.6. Literatur 7
2.7. Software 9
3. Einführung 9
3.1. Informationsverarbeitung 10
3.2. Logik 11
3.3. Generische EDV 12
3.4. ISA Ebenen 13
3.5. Von-Neumann Rechner: Architektur 14
3.6. Von-Neumann Rechner: Programmierung 15
3.7. Speicherhierarchie 16
3.8. Verteilte vs. Parallele Systeme 17
3.9. Harvard Architektur 19
3.10. Parallele Systeme 19
3.11. Echtzeitverarbeitung 23
3.12. Parallele Datenverarbeitung und Prozesse 25
3.13. Limitierungen der von-Neumann Rechnerarchitektur 25
4. Digitale Signalverarbeitung 26
4.1. Digitale Signalverarbeitung 26
4.2. Analog-Digital-Wandler (ADC) 30
4.3. Daten- und Kontrollfluß 35
4.4. Signalflussdiagramm 36

4.5. Simulation von Signalflussdiagrammen . . . 40
4.6. Kodierung von Zahlenwerten . . . 42
4.7. Signalflussdiagramm: Kombinatorische Logik . . . 44
4.8. Signallaufzeit . . . 48
4.9. Signalflussdiagramm: FIR Filter . . . 52
4.10. Signalflussdiagramm: Digitallogik . . . 52
4.11. Sequenzielle Systeme . . . 53
4.12. Hardware- und Softwarentwurf . . . 55
4.13. Parallele Datenverarbeitung . . . 57
4.14. Register Transfer Logik (RTL) . . . 59
5. Digitallogik 61
5.1. Logische Zustände . . . 62
5.2. Schaltungstechnologien . . . 64
5.3. Logikgatter . . . 64
5.4. Logik Simulation . . . 73
5.5. Boolesche Algebra . . . 76
5.6. Technische Logikzustände . . . 87
5.7. Optimierung von Digitallogik . . . 88
5.8. KV Diagramme . . . 89
5.9. QM Verfahren . . . 91
5.10. BDD Verfahren . . . 94
5.11. Hazards . . . 100
6. Kombinatorische Logik 104
6.1. ZIELE . . . 104
6.2. Kombinatorische Logik . . . 104
6.3. Signallaufzeit . . . 105
6.4. Komponenten und Schaltungen . . . 106
6.5. Addierer . . . 106
6.6. Multiplizierer . . . 111
6.7. Logische und Relationale Operationen . . . 113
6.8. Multiplexer und Demultiplexer . . . 113
7. Sequenzielle Logikssysteme 116
7.1. Sequenzielle Logik . . . 117
7.2. Synchrone Logiksysteme . . . 118
7.3. Asynchrone Logiksysteme . . . 118
7.4. Speicherelemente sequenzieller Logik . . . 120
7.5. Sequenzielle Systeme . . . 126
8. Programmierbare Logikbausteine 130
8.1. Programmierbare Logikbausteine . . . 131
8.2. Programmierbare Logikbausteine: ROM und RAM . . . 132
8.3. Programmierbare Logikarrays (PLA) . . . 134
8.4. Komplexe PLD (CPLD) . . . 135
8.5. Xilinx XC9500 CPLD . . . 139
8.6. Feldgatterarrays (FPGA) . . . 142
8.7. Metriken . . . 144
8.8. Xilinx FPGA . . . 146

8.9. Application Specific Integrated Circuit: ASIC 147
9. Register-Transfer Architektur (RTL) 151
9.1. Entwurfs- und Syntheseverfahren 151
9.2. Modellierung 155
9.3. Register-Transfer Ebene 158
9.4. Register-Transfer Ebene: Beispiel und Entwurf 162
9.5. Zusammenfassung 166
10. Hardwarebeschreibungssprachen 166
10.1. Überblick 167
10.2. VHDL 168
10.3. VHDL - Aufbau 169
10.4. VHDL - Komponenten 171
10.5. VHDL - Signale 173
10.6. VHDL - Signaltypen 176
10.7. VHDL - Arraytypen 178
10.8. VHDL - Prozesse 179
10.9. VHDL - Bedingte Ausdrücke und Anweisungen 187
10.10. VHDL - Schleifen 190
10.11. VHDL - Simulation 192
11. Zustandsautomaten 195
11.1. RTL und Zustandsautomaten 195
11.2. Endliche Zustandsautomaten 197
11.3. Der Moore-Automat 199
11.4. Hierarchische RT Systeme 202
12. Synthese 203
12.1. Algorithmen 204
12.2. Register-Transfer Architektur 206
12.3. RTL Synthese von Digitallogikschaltungen 210
12.4. μRTL Synthese von sequenziellen Prozessen 212

2. Überblick

Synopsis

Einführung in den Hardware- und Systementwurf mit anwendungsspezifisch konfigurierbarer Digitallogik mittels VHDL-Synthese und deren Anwendungen.

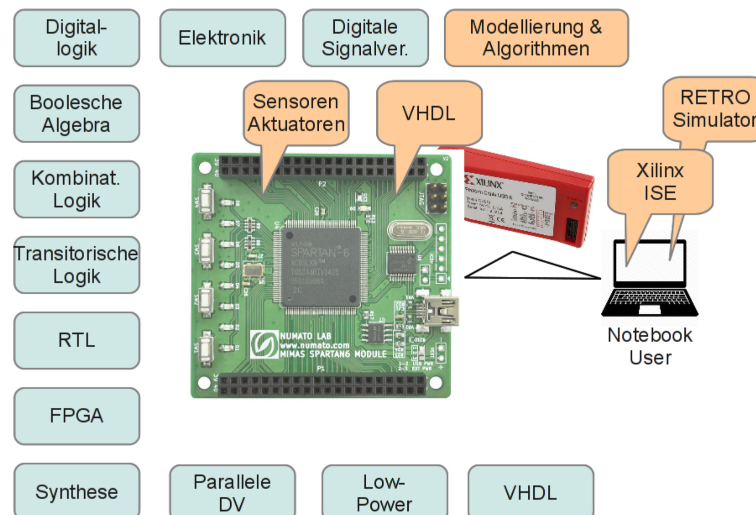
Hardware-Synthese ist ein automatischer Prozess, um aus einer Verhaltens- und Strukturbeschreibung logische Schaltungen und Netzlisten zu erhalten, die direkt technologisch umsetzbar sind.

Die verwendete Hardware-Beschreibungssprache sollte dabei unabhängig von der Zieltechnologie sein. Beim Hardware-Entwurf spielen System-On-Chip Architektur und Modellierungs-Methoden eine wesentliche Rolle

2.1. Grunddaten

VAK	03-ME-712.05
Dozent	PD Stefan Bosse, sbosse@uni-bremen.de
Kategorie	Kurs: Vorlesung und integrierte Übung
Umfang	4 SWS
Art	Master Ergänzung
Profilbereich	KIKR
ECTS	6
Leistung	Übungsblätter, Mündliche Prüfung
Wann	Jedes Winter Semester, Mo. 14-17 Uhr
Wo	Rober Hooke Str. 5, Raum 1.17
Info	http://sun45.informatik.uni-bremen.de

2.2. Inhalte und Themen



2.3. Zentraler Inhalte

➤ **Digitallogik**

❏ Architekturen

- ❏ Modellierung
- ❏ Synthese

- ➤ Einsatz von Digitallogik zur Lösung bestimmter Probleme der Datenverarbeitung
- ➤ Technologischer Fortschritt ermöglicht komplexe Digitallogiksysteme auf kleinsten Raum
- ➤ Auswahlkriterien:
  - ❏ Entwurfsmethoden?
  - ❏ Technologien?

2.4. Fragestellung

- ➤ Wofür? Für welchen Zweck eignen sich anwendungsspezifische Digitallogiksysteme?

  Z.B. für Funktionale Systeme $f(x)$: $x \to y$!
- ➤ Wann?

  Abgrenzung zur klassischen Hardware-Software-Lösung mit generischen μP-Systemen
- ➤ Wie?

  Architekturen der Digitallogik $\to$ Register-Transfer Logik
- ➤ Womit?

  Hardware-Entwurf mit technologieunabhängiger Hardware-Beschreibungssprache VHDL

Wie kommt der Algorithmus auf den Mikrochip?

2.5. Ziele

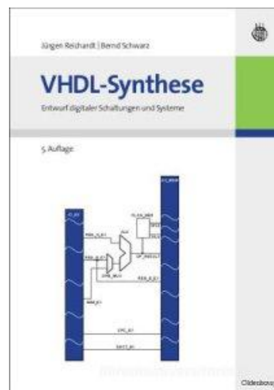
1. Entwurf einfacher Digitallogikschaltungen mittels Boolescher Algebra (bzw. Netzlisten) auf Logikgatterebene $\to$ Praktische Umsetzung
2. Verständnis von kombinatorischer und transitorischer Logik
3. Optimierung von Digitallogikschaltungen (kombinatorisch und transitorisch)

4. Entwurf von Zustandsautomaten aus algorithmischer Beschreibung
5. Verständnis und Entwurf von RTL Architekturen
6. Entwurf einfacher Digitallogikschaltungen mittels Hardwarebeschreibungssprache VHDL → Praktische Umsetzung
7. Verständnis von technologischer Digitallogik und deren Randbedingungen → Praktische Umsetzung

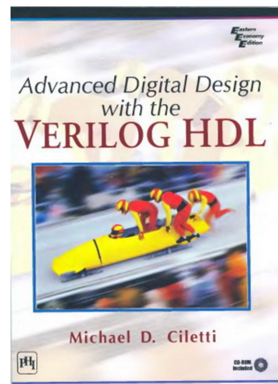
2.6. Literatur

➤ Empfohlene Literatur (neben Vorlesungsskript) zur Vertiefung

VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme, Jürgen Reichardt, Bernd Schwarz, Oldenbourg Wissenschaftsverlag, ISBN 978-3486273847



Advanced Digital Design with the Verilog Hdl (Prentice Hall Xilinx Design Series), Michael D. Ciletti, Prentice Hall, ISBN 978-0130891617

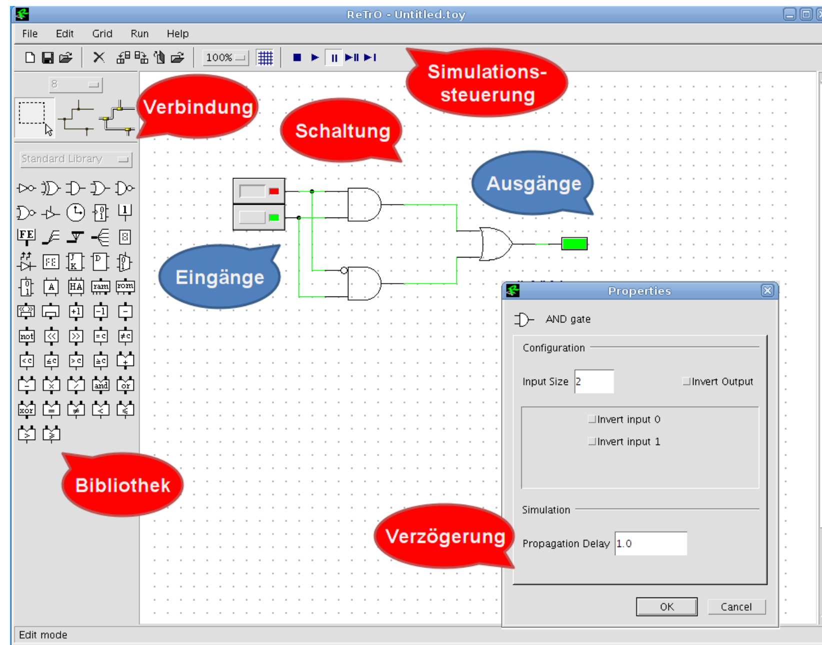


The Scientist & Engineer's Guide to Digital Signal Processing, Steven W. Smith, 1999, California Technical Publishing, ISBN 0-9660176-4-1

2.7. Software

Retro

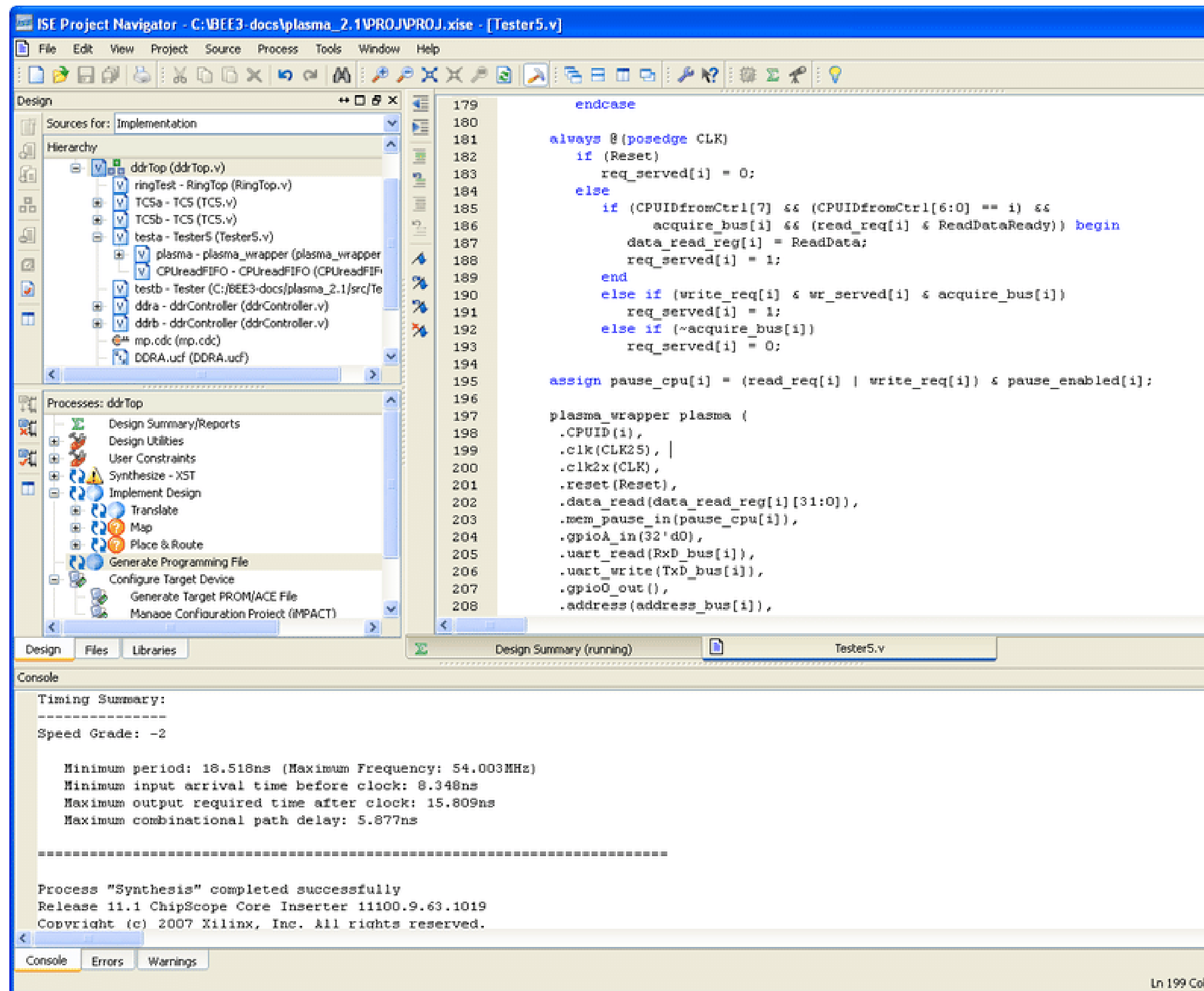
- RTL Simulator
- Einfaches Hardware Modell: Technologische Gatterverzögerung
- Schematischer Entwurf



Xilinx ISE

➤ Gatelevelsynthese (VHDL → FPGA Konf.)

➤ Entwurf mit Hardwarebeschreibungssprache



3. Einführung

3.1. Informationsverarbeitung

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

- Numerik
- Steuerung von Maschinen

- Reaktive Systeme mit Benutzerinteraktion
- Digitale Signalverarbeitung
- Programmgesteuerte Automaten → Generische Mikroprozessoren!
- Kommunikation

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, wie in Abb. 1 gezeigt ist.

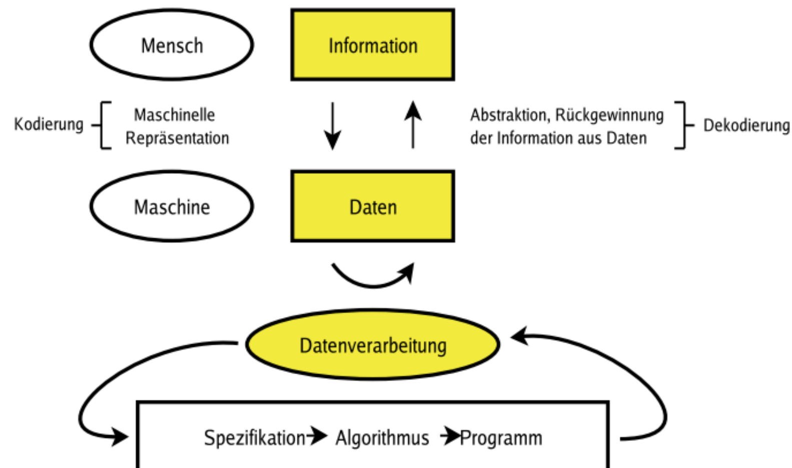


Figure 1. Informations- und Datenverarbeitung

3.2. Logik

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

```
{wahr,falsch}, {hell,dunkel}, {x0,x1} usw.
```

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

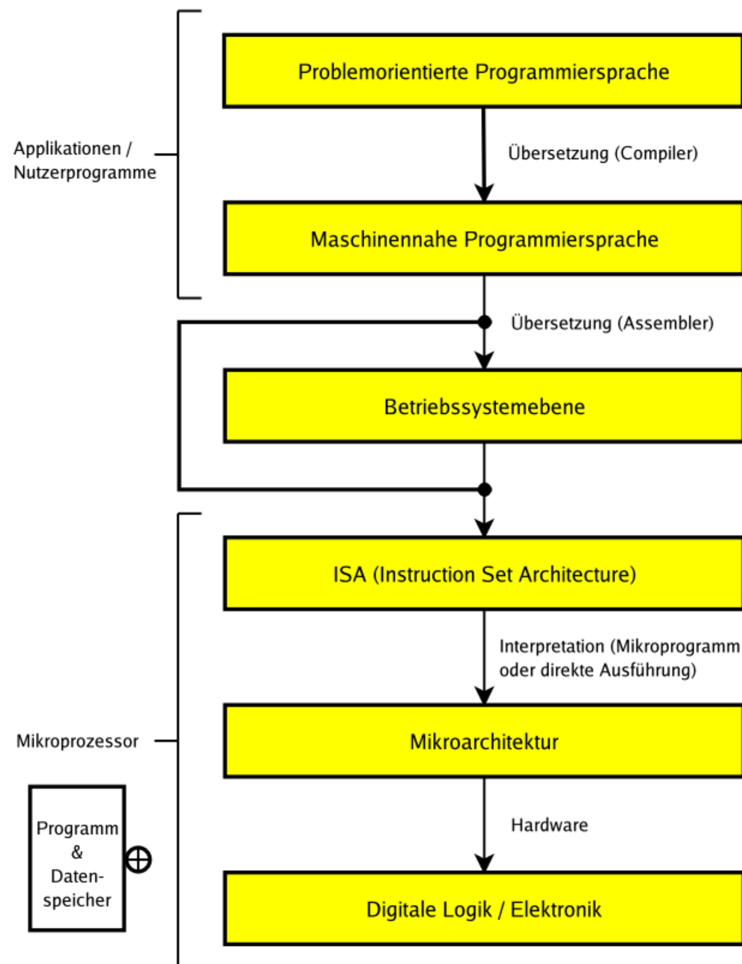
```
{1,0} → {H,L} → {3.3V,0V}
```

- Ausgangspunkt und Motivation für anwendungs- und problemorientierte Digitallogik liegt in generischer Rechnerarchitektur und herkömmlichen programmbasierten Problemlösungen begründet.

Generisches Datenverarbeitungssystem ⇔ Programmsteuerung

- Generische EDV und Programmsteuerung bedeuten Einschränkung bei Optimierungsmöglichkeiten → Die Maschine ist universell!
- Dazu soll das Schichtenmodell einer konventionellen Datenverarbeitungsanlage näher betrachtet werden.

3.3. Generische EDV



3.4. ISA Ebenen

Abstraktion ISA: Instruction Set Architecture

Die Instruktionsebene eines Mikroprozessors kann mit zwei unterschiedlichen Architekturen implementiert werden:

1. Direkt mit digitaler Logik realisiert;

   ➤ RISC Prozessoren

2. Mikroarchitektur: Transformation von komplexen Maschinenbefehlen in eine Untermenge einfacher Maschinenbefehle (sequenzielle Komposition) → Microcode

- ➤ CISC Prozessoren
- ➤ Vorteil: Microcode kann aktualisiert werden (Fehlerbeseitigung!)

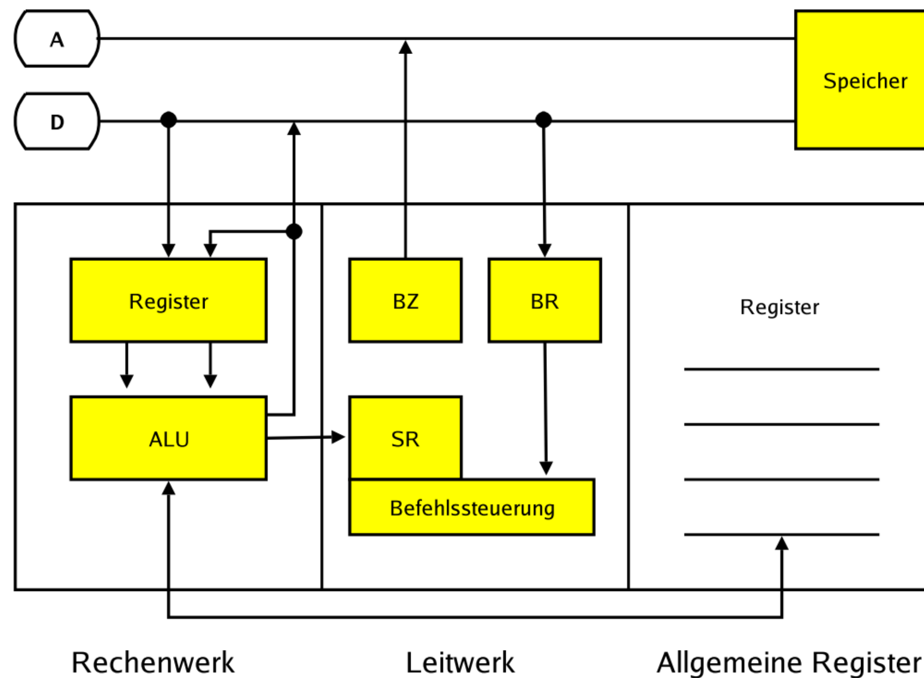
Man unterscheidet:

RISC
: Reduced Instruction Set Computer Architektur
Kleine Instruktionsmenge besteht aus einfachen Operationen meist mit konstanter Instruktionslänge. Können auf ISA Mikroarchitektur verzichten (z.B. SUN Sparc, Motorola/IBM PowerPC, **ARM**)

CISC
: Complex Instruction Set Computer Architektur
Große Instruktionsmenge mit komplexen Operationen und meist variabler Instruktionslänge → ISA Mikroarchitektur bei vielen CISC-Prozessoren (z.B. Intel X86/X64 Pentium usw.);

- ➤ Die Betriebssystemebene ermöglicht die Abstraktion der Rechnerarchitektur, die ISA-Ebene ermöglicht die Abstraktion von der Digitallogikebene.
- ➤ Ein klassisches Mikroprozessorsystem arbeitet ein Maschinenprogramm (→ Ablaufvorschrift) **sequenziell** ab → Limitierung!
- ➤ Die einzelnen Instruktionen eines Maschinenprogramms werden in unterschiedlichen Werken im Mikroprozessor verarbeitet.
- ➤ Zum näheren Verständnis der Programmsteuerung muss der Aufbau und die Architektur einer generischen von-Neumann-Anlage näher betrachtet werden.

3.5. Von-Neumann Rechner: Architektur



Die Rechnerarchitektur ist unterteilt in

➤ das Rechenwerk,

➤ das Leitwerk und ein

➤ Satz allgemeiner Register (temporäre Datenspeicher).

Das Leitwerk enthält folgende spezielle Register:

BZ: Befehlzähler
Zeigt auf Speicheradresse des nächsten auszuführenden Maschinenbefehls.

BR: Befehlsregister
Enthält Kodierung des aktuell ausgeführten Befehls.

SR: Statusregister
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 Adressbus mit einem (einzigen) Hauptspeicher verbunden, dessen Speicherzellen über die Adresse ausgewählt werden.

➤ Der *Hauptspeicher* enthält:

- ❏ Programmbefehle (Code)
- ❏ Programmdaten (Daten → {Eingabe, Ausgabe, Zwischenergebnisse}) → Limitierung

➤ 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. → Limitierung

3.6. Von-Neumann Rechner: Programmierung

➤ Hello World Programm in Hochsprache C:

```
main() { printf("hello, world\n"); }
```

➤ Maschinenprogramm Assembler für den ZX81 (ca. 1981, Zilog Z80)

```
        CALL SPRINT
        DEFM HELLO WORLD.
        DEFB FF
        RET
SPRINT    POP HL
        LD A,(HL)
        INC HL
        PUSH HL
        CP FF
        RET Z
        CALL PRINT
        JR SPRINT
```

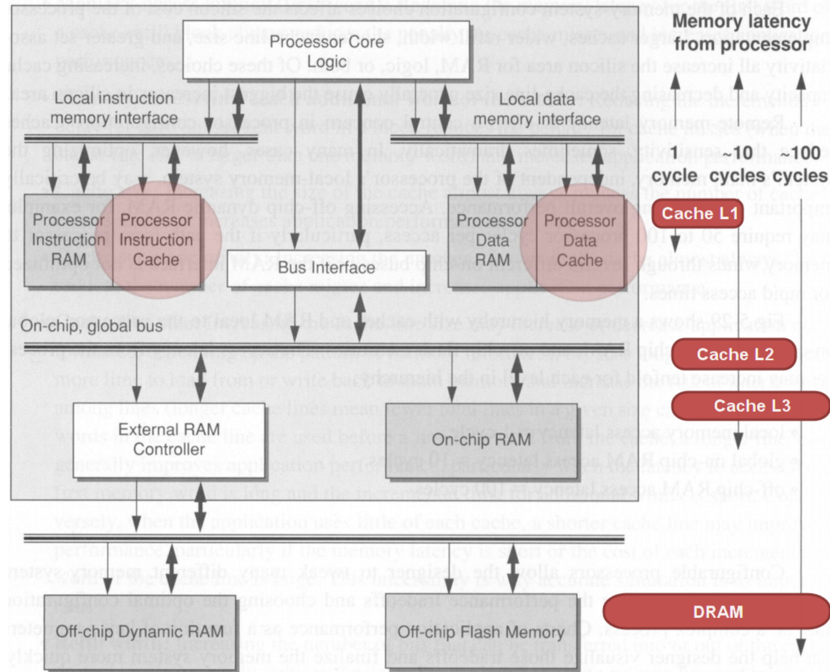
3.7. Speicherhierarchie

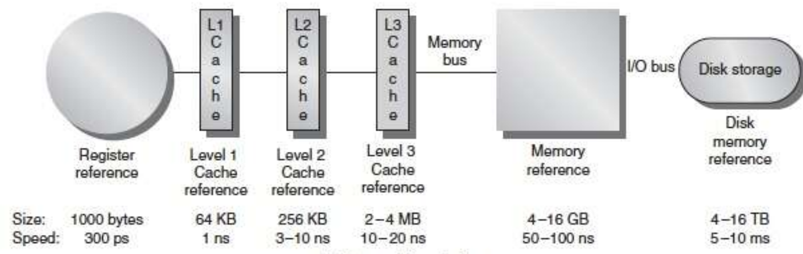
➤ In einem Datenverarbeitungssystem gibt es verschiedene Speichertechnologien und Architekturen

➤ Einzelne Register sind i.A. direkt an Verarbeitungseinheiten gekoppelt, und bieten die kürzeste Zugriffszeit (schnell), bei niedriger Speicherdichte

➤ RAM Blöcke als adressierbare Register Files können eng an die Verarbeitungseinheiten gekoppelt sein (z.B. Cache Speicher), bei mittlerer Zugriffszeit und Speicherdichte, oder als externer Speicher über einen Speicherbus bei hoher Zugriffszeit und Speicherdichte.

Speicherhierarchie eines SoC Designs [ECSOC,Rowen,2004]





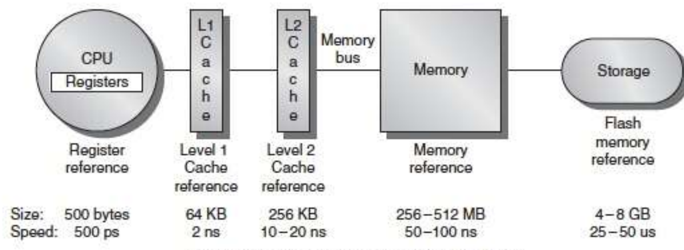


Figure 2. Speicherhirarchie und Cache Speicher [Henessy, 2012]

3.8. Verteilte vs. Parallele Systeme

Verteiltes System

Ein verteiltes System ist eine Sammlung von **lose gekoppelten** Prozessoren oder Computern, die über ein Kommunikationsnetzwerk miteinander verbunden sind (**Multicomputer**).

- **Speichermodell**: Verteilter Speicher → Jeder Prozessor verfügt über privaten Speicher
- **Kommunikation**: Nachrichtenbasiert über Netzwerke
- **Ressourcen**: Nicht direkt geteilt

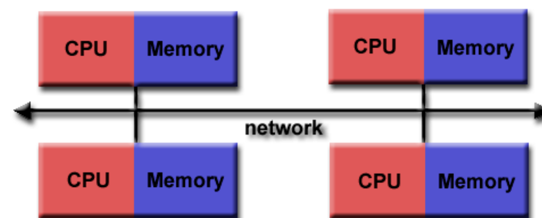
Paralleles System

Ein paralleles System ist eine Sammlung von **stark gekoppelten** Prozessoren (**Multiprozessoren**)

- **Speichermodell**: Gemeinsamer Speicher
- **Kommunikation**: Direkt über elektrische Signale → Switched Network (Kreuzschiene) | Bus → Punkt-zu-Punkt | Punkt-zu-N-Netzwerke
- **Ressourcen**: Gemeinsam genutzt (Bus, Speicher, IO)

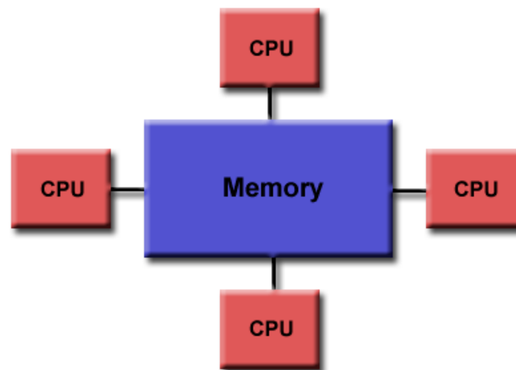
Verteilter Speicher

- Zugriff auf Speicher erfordert Netzwerkkommunikation
- Vorteil: Speicher ist skalierbar mit Anzahl der Prozessoren
- Nachteil: Langsamer Speicherzugriff zwischen Prozessen



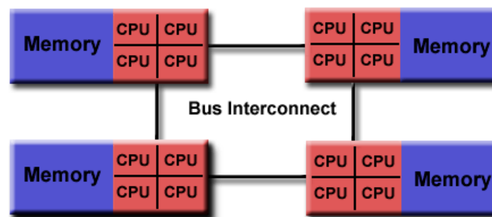
Unified Memory Architecture

- Symmetrisches Multiprocessing (SMP)
- Vorteil: Konstante Zugriffszeit auf Speicher
- Vorteil: Schneller Speicherzugriff zwischen Prozessen



Non Unified Memory Architecture

- Vorteil: Clustering von SMPs
- Nachteil: Ungleiche Zugriffszeiten auf Speicher



[computing.llnl.gov]

3.9. Harvard Architektur

- Der von Daten- und Instruktionen gemeinsam geteilte Hauptspeicher ist der Flaschenhals bei der Programmausführung.
- Aufteilung von Daten und Programmcode in getrennten Speicher beschleunigt die Ausführung von Maschinenanweisungen.
- Das Leitwerk des Prozessors kann direkt mit dem Codespeicher verbunden werden!

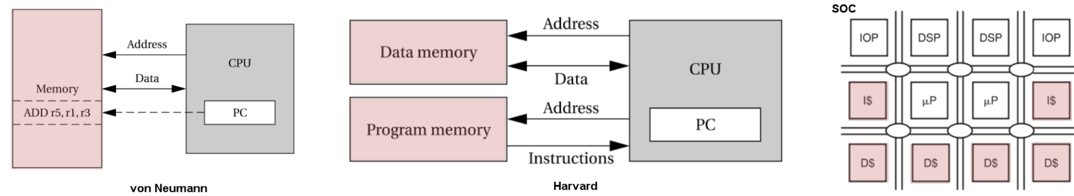


Figure 3. Übergang von zentralem Hauptspeicher zu N-Speicher System [CC,Wolf,2008]

3.10. Parallele Systeme

Definition

- Zerlegung (Partitionierung) eines sequenziellen Algorithmus oder eines Programms in **parallele Tasks** (Prozesse) → **Parallele Komposition**
- Ausführung der Prozesse parallel (nebenläufig und ggfs. konkurrierend) auf mehreren Verarbeitungseinheiten (u. A. generische programmgesteuerte Prozessoren)

Motivation für parallele Datenverarbeitung

- Verkleinerung der Berechnungslatenz

Def. Latenz: Gesamte oder Teilbearbeitungszeit eines Datensatzes

- Erhöhung des Datendurchsatzes

Def. Datendurchsatz: Anzahl der verarbeiteten Datensätze pro Zeiteinheit

- Latenz und Bandbreite sind zunächst unabhängig!
- Pipelining kann die Bandbreite erhöhen (nur Sinnvoll bei Datenströmen)
- Parallele Tasks können die Latenz verringern

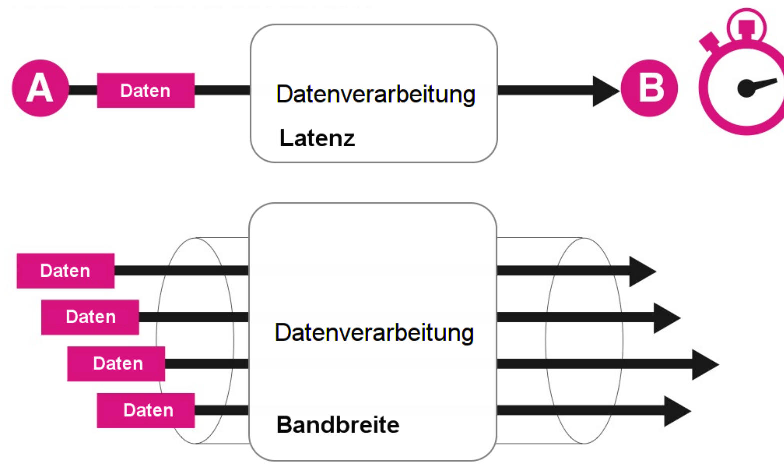


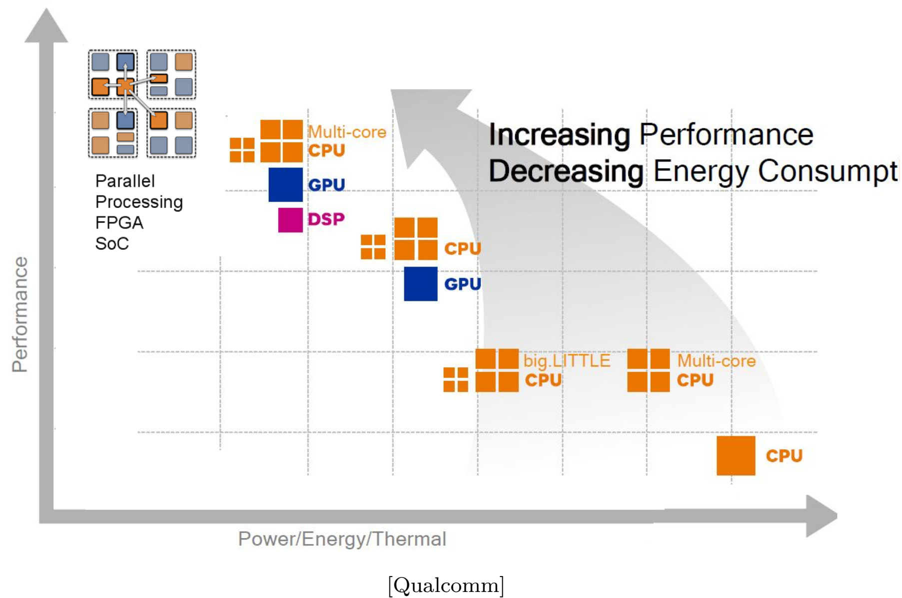
Figure 4. Unterschied Latenz zu Bandbreite

Man unterscheidet: Parallele Rechnerarchitektur und parallele Datenverarbeitung

Motivation für anwendungsspezifische parallele Datenverarbeitung

- Steigerung der Energieeffizienz von mikroelektronischen Systemen:
  - Komplexe generische Einprozessoranlagen besitzen ungünstige Energieeffizienz, aber:
  - Anwendungsspezifizierung und Parallelisierung kann zu verbesserter Energieeffizienz des Gesamtsystems führen!
- Skalierung von parallelen Rechnern auf reine Digitallogiksysteme für anwendungsspezifische Lösungen kann deutliche Reduktion der Hardware-Komplexität und der elektrischen Leistungsaufnahme bedeuten!
  - System-on-Chip Entwurf
  - Leistungsaufnahme eine CMOS Digitalschaltkreises (f: Frequenz, N: Schaltende Transistoren, U: Spannung, C: Technologieparameter):

$$P(f, N, U, C) \sim fNU^2C \tag{1}$$



[Qualcomm]

Figure 5. Erhöhung der Performanz und der Energieeffizienz durch Parallelisierung

Anwendung anwendungsspezifischer paralleler Datenverarbeitung

- Digitale Bildverarbeitung und automatische Bildinterpretation (Vision)
- Datenkompression (Video, mpeg)
- Komplexe Steuerungssysteme mit großer Anzahl von Freiheitsgraden, wie z.B. Positionierungssteuerung von Robotergelenken und Maschinen
- Kommunikation, z. B. nachrichten-basiertes Routing
- Kryptoverfahren
- Parallele numerische Verfahren, wie z. B. Lösung von Differentialgleichungen (Strömungsmechanik, Elektromagnetische Wellenausbreitung, Wetter- und Klimamodelle)

Beispiel Numerik und Digitale Bildverarbeitung

- Welche Probleme und Nachteile gibt es?

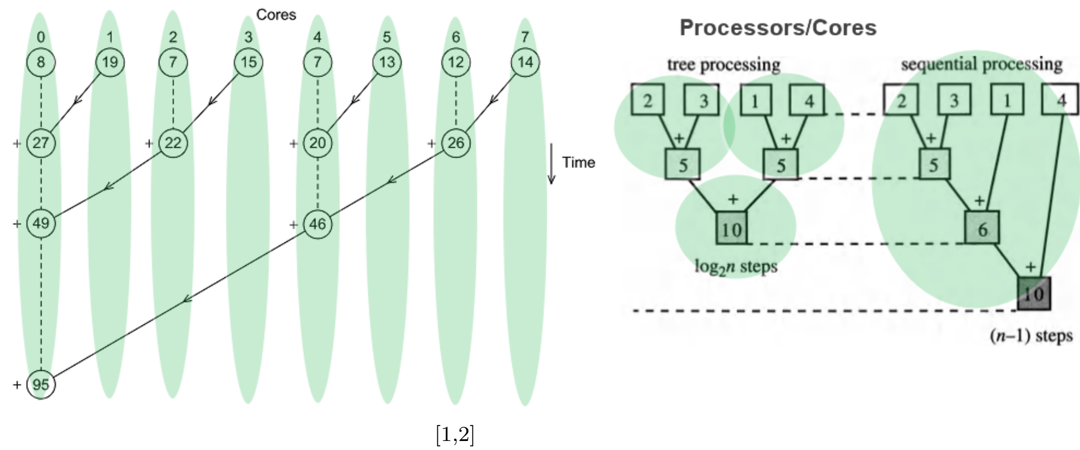
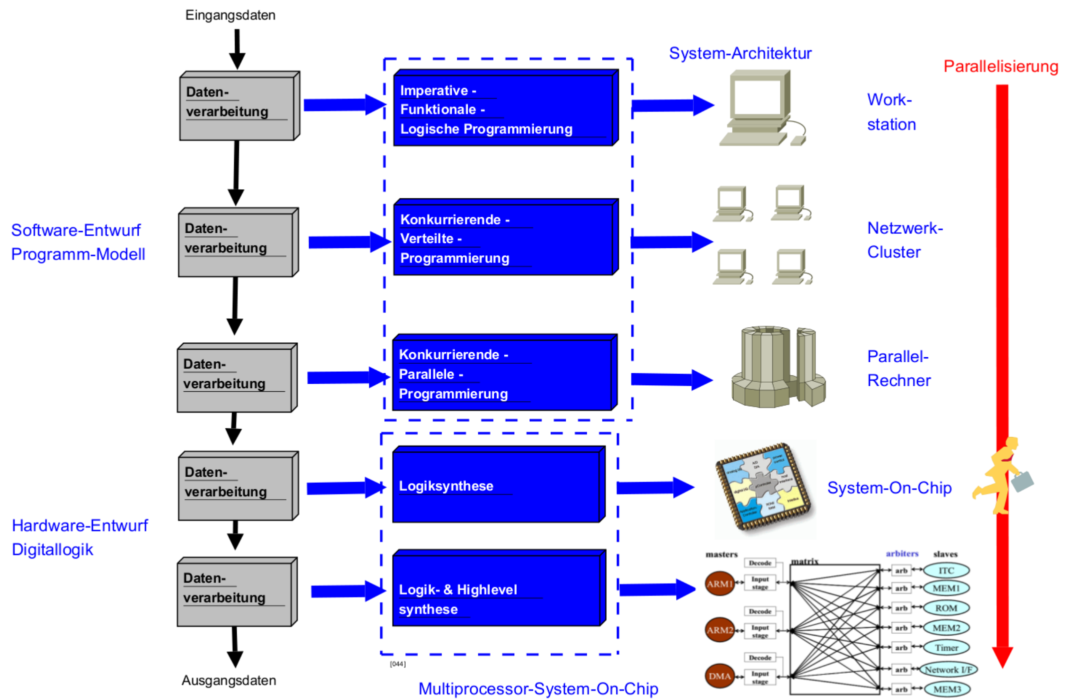


Figure 6. Parallele Berechnung der Summe von Bildpixeln mit Baumstruktur

Ebenen der Datenverarbeitung in Abhängigkeit vom Parallelisierungsgrad



Beispiel Sensorisches Material

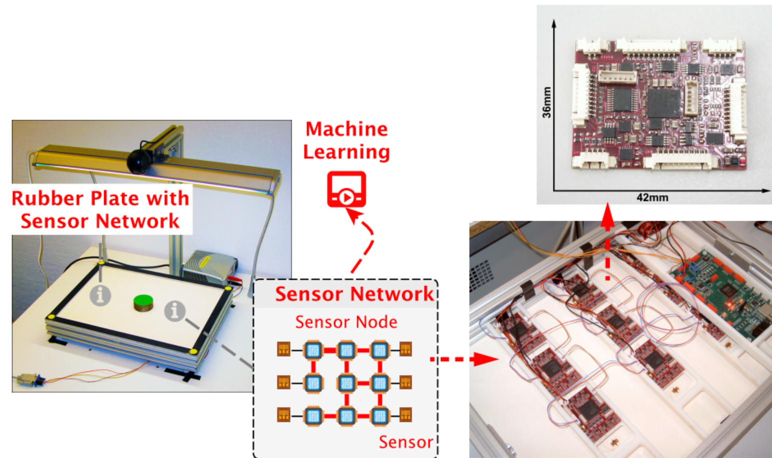


Figure 7. Beispiel eines verteilten und parallelen Systems: Sensornetzwerk (unten rechts) mit Maschen Topologie mit System-on-Chip Sensorknoten (oben rechts) mit massiv paralleler Datenverarbeitung. Sensorik: Dehnungsmessstreifen auf Gummiplatte (links)

3.11. Echtzeitverarbeitung

➤ Echtzeitverarbeitung bedeutet die Ausführung eines Tasks innerhalb eines vorgegebenen Zeitintervalls $[t_0,t_1]$

- ❏ **Soft Realtime**: Zeitüberschreitung bei weicher Echtzeitanforderung $\rightarrow$ Tolerierbar
- ❏ **Hard Realtime**: Zeitüberschreitung bei harter Echtzeitanforderung $\rightarrow$ Systemfehler

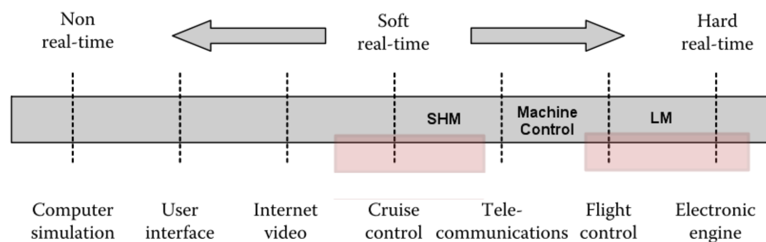


Figure 8. Echtzeitspektrum: Wiche und harte Echtzeitanforderungen bei unterschiedlichen Applikationen [F]

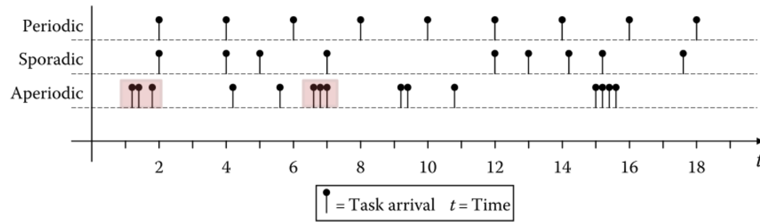


Figure 9. Periodische, sporadische, und aperiodische Tasks [F]

Preemption

➤ Wenn schon nicht alles Tasks gleichzeitig verarbeitet werden können, dann wenigstens die wichtigsten vorrangig ausführen

➤ Das erfordert aber:

- ❏ Festlegung einer Priorität
- ❏ Bei Echtzeitanforderungen die Unterbrechung von unterlegenen (kleiner Priorität) Tasks

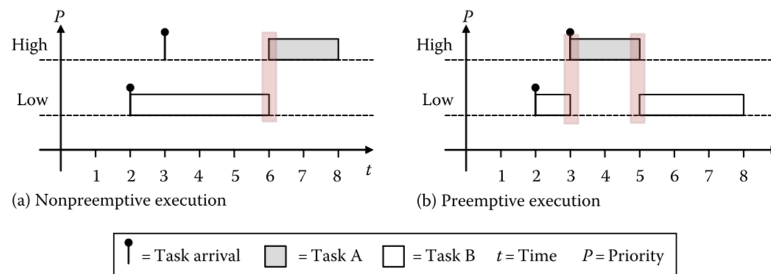


Figure 10. Unterschied preemptive und nicht preemptive Ausführung von Tasks [F]

3.12. Parallele Datenverarbeitung und Prozesse

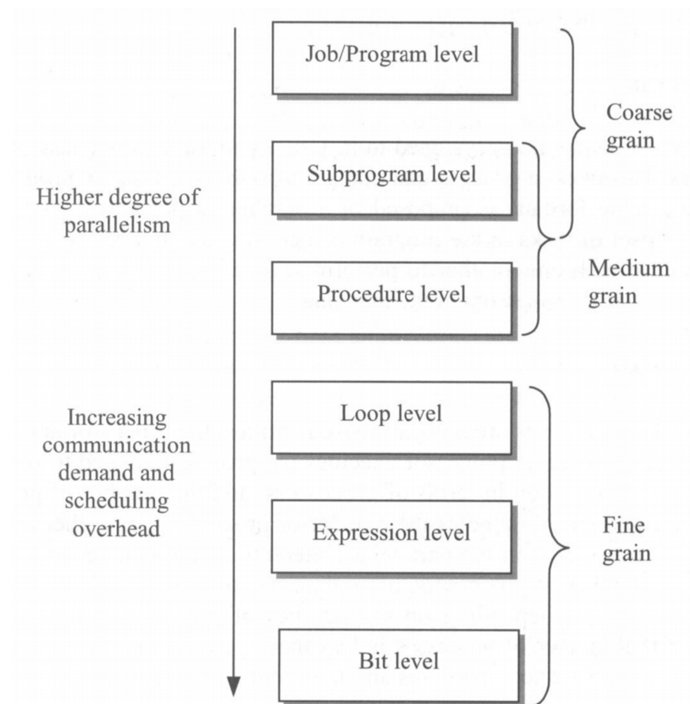


Figure 11. Ebenen der parallelen Datenverarbeitung

3.13. Limitierungen der von-Neumann Rechnerarchitektur

Zusammenfassung

1. Datendurchsatz ist durch geteilten Speicherbus begrenzt - schlechte Skalierung von Multiprozessor- und Multikernsystemen
2. Programm (Befehle) und Daten (Operanden) müssen sequenziell aus dem gemeinsam geteilten Hauptspeicher geladen werden
3. Nur geringe Möglichkeiten der Parallelisierung (Pipeline-Architektur)
4. Befehlsausführung in sechs Phasen erhöht Bearbeitungslatenz
5. Generizität der Programmierbarkeit geht zu Lasten der Performanz (Latenz, Datendurchsatz, Energiebedarf) - keine algorithmische Optimierung möglich

4. Digitale Signalverarbeitung

4.1. Digitale Signalverarbeitung

Digitale Signalprozessoren

- Neben Rechnersystemen, die generisch/universell eingesetzt werden können (Desktop Rechner, Notebook) gibt es spezialisierte Rechnersysteme für die Signalverarbeitung → Digitale Signalprozessoren (DSP)

Analoge (physikalische) Signale

- 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 besitzten, sondern setzt sich zusammen aus einer Überlagerung mit einem stochastischen Rauschsignal.

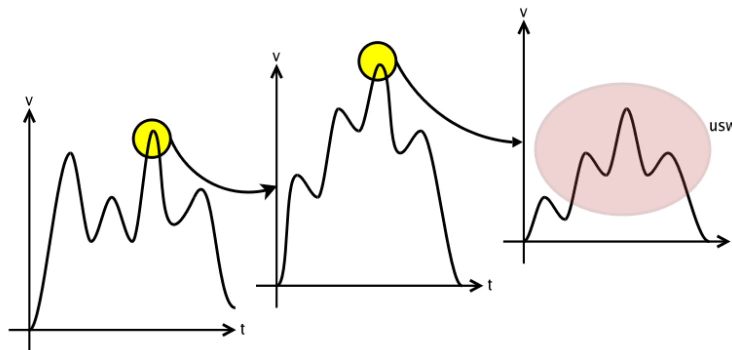
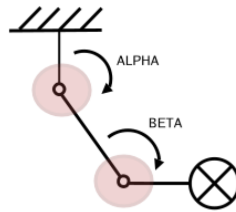


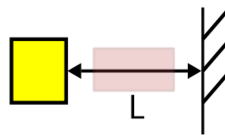
Figure 12. Zoom eines analogen Signals in Zeit- und Wertdimension (vereinfacht)

Beispiele für analoge Signale

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



- Entfernung eines Gegenstandes



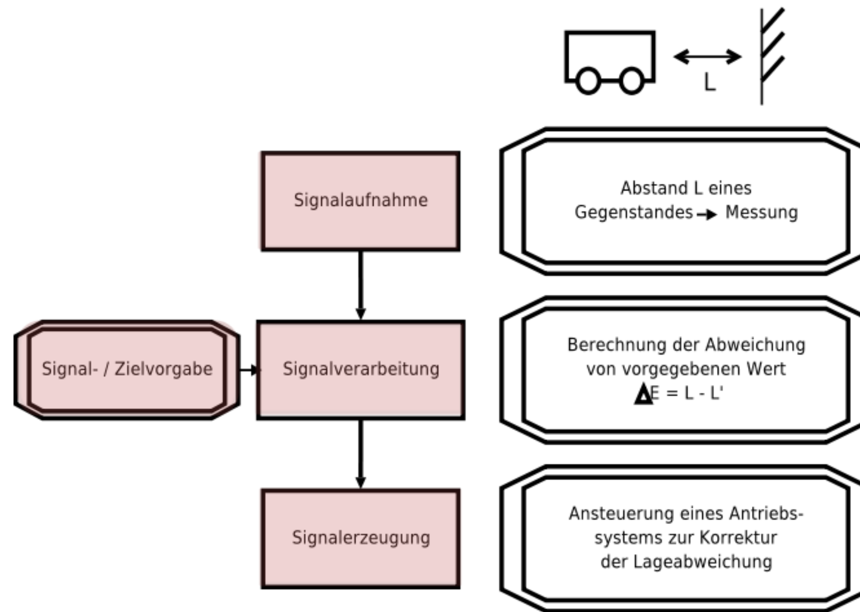
- Lichtintensität und Lichtbilder (Bildverabeitung!)

Signalverarbeitung

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

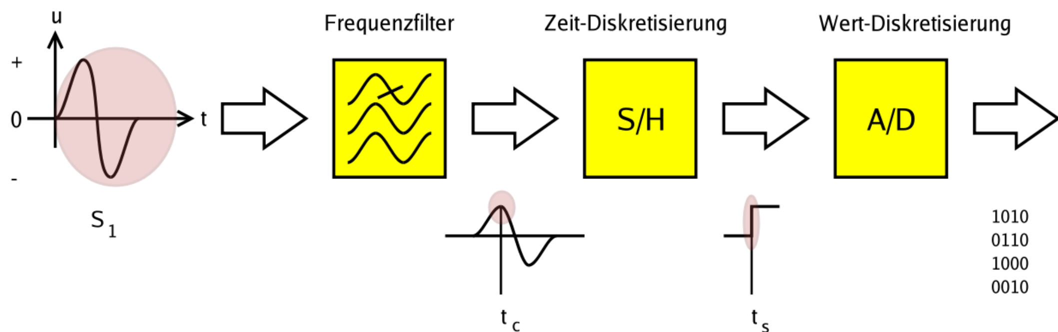
Die digitale Signalverabeitung erfordert:

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



Digitalisierung als erste Stufe der Signalverarbeitung

- Die AD-Wandlung setzt i.A. ein 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.

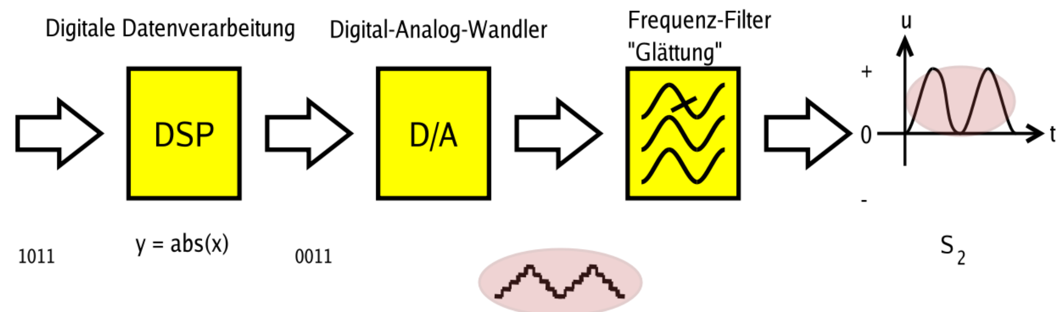


Digitale Signalverarbeitung als zweite Stufe

- Die Eingangsdaten stehen als kontinuierlicher Datenstrom für die weitere Verarbeitung zur Verfügung. Ein Eingangsdatenstrom wird in einen Ausgangsdatenstrom transformiert.

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 (DSP)

- Digitale Filterung
- Faltungsoperationen
- Korrelationsanalyse
- Zeit ⇔ Frequenztransformationen (FFT)
- Wellenformerzeugung
- Bildverarbeitung
    1. Digitale Filterung
    2. Mustererkennung
    3. 3D-Operationen
- Spracherkennung

➤ Steuerungs- und Regelungsaufgaben

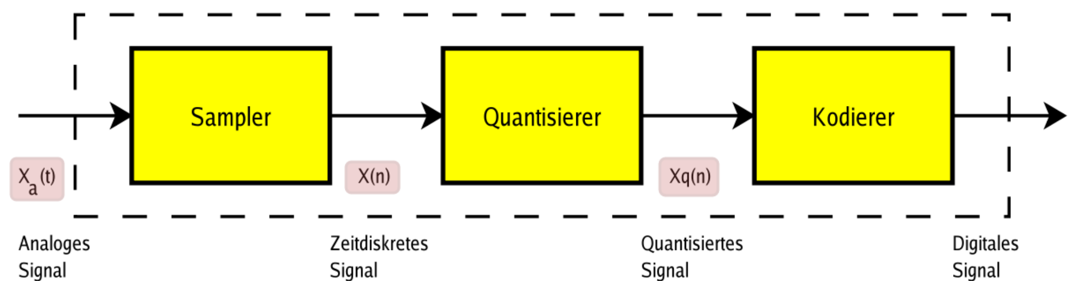
1. Positionsregelung
2. Spannungsregelung
3. Motorsteuerung (z.B. Drehzahl)
4. Navigation

4.2. Analog-Digital-Wandler (ADC)

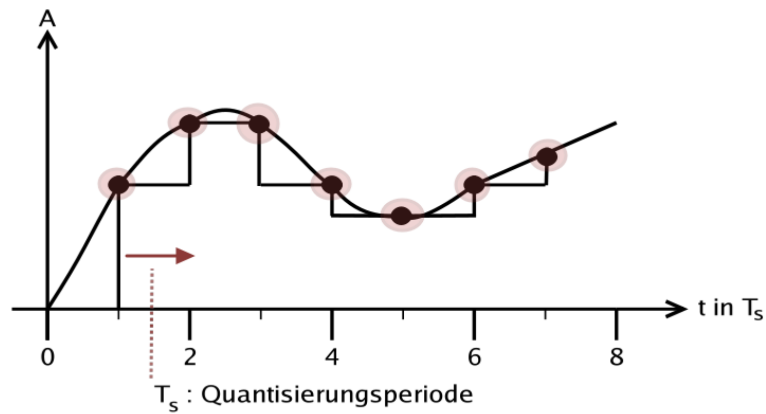
Ein AD-Wandler ist in drei Stufen unterteilt:

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 $q(n\text{-}\Delta) \leq q(n) < q(n+\Delta)$, 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 Graycode-Kodierung mit der Eigenschaft, daß aufeinanderfolgende Werte immer nur eine Änderung eines einzigen Bits hervorrufen.

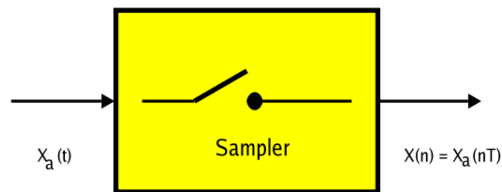
Aufbau eines ADC



Diskretisierung eines analogen Signals



Sampler



Sampling Theorem

Im allgemeinen wird in der digitalen Signalverarbeitung ein analoges Signal periodisch mit einer festen Abtastfrequenz $f_{\mathsf{sample}}=1/T_{\mathsf{sample}}$ abgetastet.

➤ Das Sampling-Theorem besagt, daß das zu digitalisierende Signal nur ein Frequenzspektrum bis zu einer maximalen Frequenz f_{signal} besitzen darf, ansonsten treten Artifakte bei der Signalerfassung auf:

$$f_{sample} > 2f_{signal}$$

Artifakte (Aliasing)

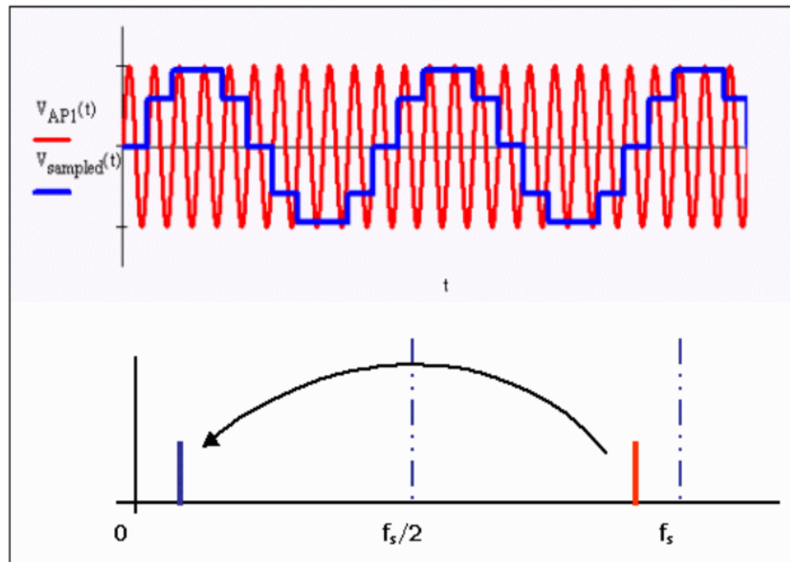


Figure 13. Frequenzen oberhalb $f_{\mathsf{sample}}/2$ werden im Frequenzspektrum gespiegelt - es treten Signalartifkate auf

ADC Verfahren und Architekturen

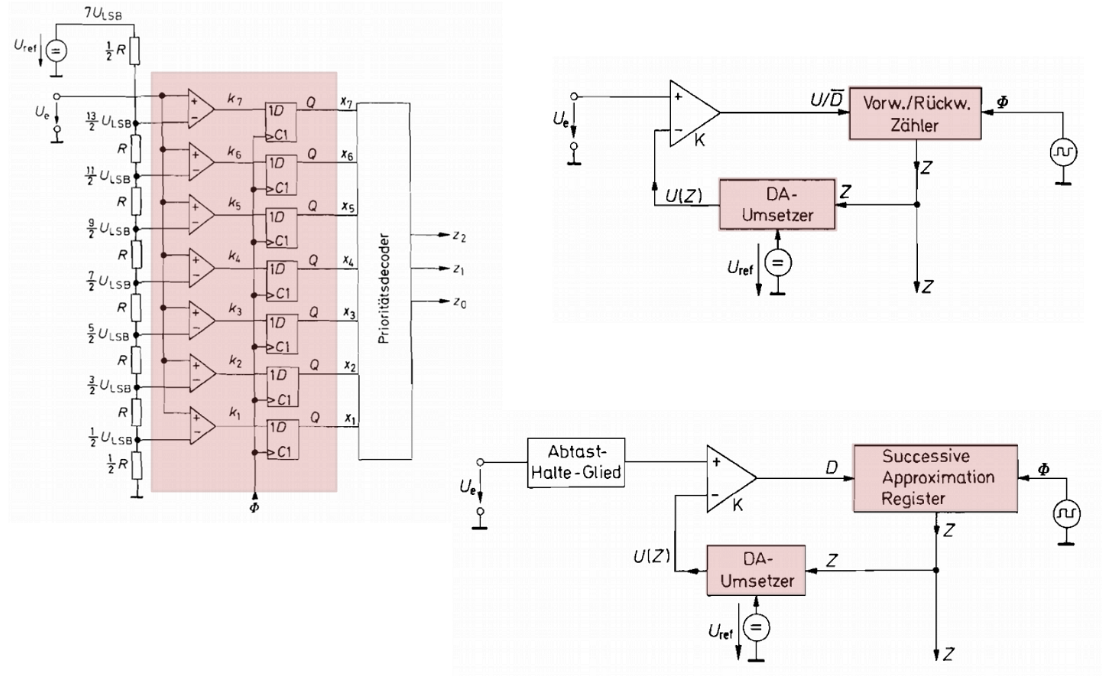
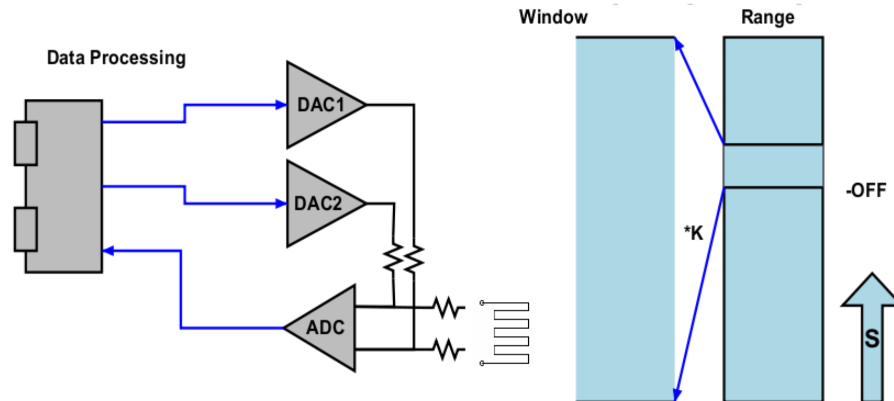


Figure 14. AD Verfahren (Parallel-, Wäge- und Zählverfahren) [HST, Tietze, 2002]

Zooming ADC

- Resistive Sensoren, z.B. Strain-Gauge Sensoren, liefern eine nur kleine relative Änderung ihres Widerstandes in der Größenordung von 1%.
- Annahme: Verwendung eines unkalibrierten und unkompensierten Sensors mit kleinen relativen Messbereich
- Ein Fensterverfahren kann verwendet werden um einen solchen Sensor an das Messsystem anzupassen → hohe Auflösung und Nutzung des vollen Bereichs

$$W(s) = k(s - off)$$



Algorithm 1. (*Autokalibration mit sukzessiver Approximation*)

$sar \leftarrow DIGITALRANGE/2$
$DAC_1 \leftarrow GAIN_0,\ DAC_2 \leftarrow 0$
WHILE $sar <> 0$ **DO**
  IF $ADC > DIGITALRANGE/2$
  THEN $DAC_2 \leftarrow DAC_2 + sar$
  ELSE $DAC_2 \leftarrow DAC_2 - sar$
  END
  $sar \leftarrow shift_{right}(sar, 1)$
END
$off \leftarrow DAC_2 - DAC_1$

4.3. Daten- und Kontrollfluß

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:

Signalflussdiagramm

➤ Ein Signalflussdiagramm ist ein gerichteter Graph mit Knoten (Verabeitunsgelementen) und Kanten (Signalfluss) und beschreibt den Datenfluss explizit und den Kontrollfluss implizit.

➤ Ein Signalflussdiagramm besteht aus folgenden Komponenten:

  ❏ Arithmetische Operationen (Berechnung)

  ❏ Verzögerungsglieder (Digitaler Sample&Hold, Register, Speicher)

4.4. Signalflussdiagramm

1. Arithmetische Operationen:

   ➤ Addition (Summierer)

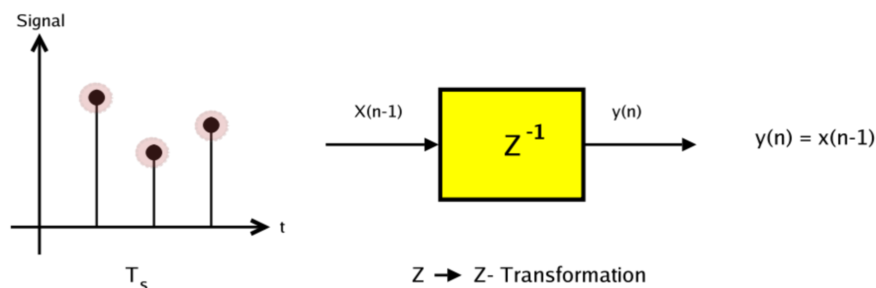
$$y(n) = \sum_i x_i(n)$$

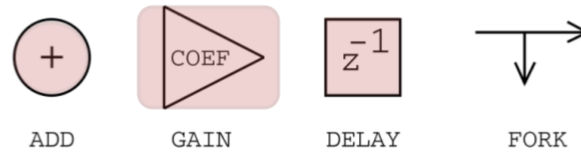
➤ Multiplikation (Verstärker, Skalierer)

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

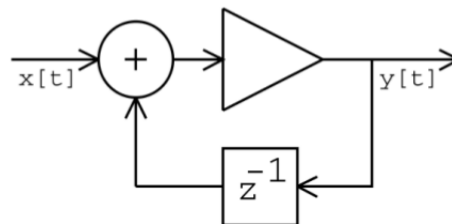
2. Verzögerungsglieder für eine Akquisitionsperiode T_s

$$y(n) = x(n-1)$$





(a) some nodes in a computation graph



(b) an example computation graph

Figure 15. Elemente eines Signalflussgraphen [SODA,Constantinides et al.,2004]

Z-Transformation

- Signale können dual im Zeit- und Frequenzbereich/Raum beschrieben und manipuliert werden.
- Signalverarbeitungssysteme können dual im Zeit- und Frequenzbereich entworfen werden → Frequezraum i.A. beim filternetwurf bevorzugt
- Die Z-Transformation ersetzt die Variablen des Signals, eine Zeitgröße, durch eine komplexe Frequenzvariable Z (Transformation von Signal- in den Spektralbereich):

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

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

$$Z^{-1} \equiv 1/Ts \equiv \Delta n = -1 \text{ mit } T_s : \text{Disketrisierungszeit}$$

Beispiel: Tiefpassfilter

- Näherung einer Mittelwertbildung eines Signals S als *exponentieller Mittelwertbildung* → **Tiefpaß-Filter 1. Ordnung**:

$$\bar{S}(n) = S(n)(1 - b_1) + \bar{S}(n-1)b_1$$

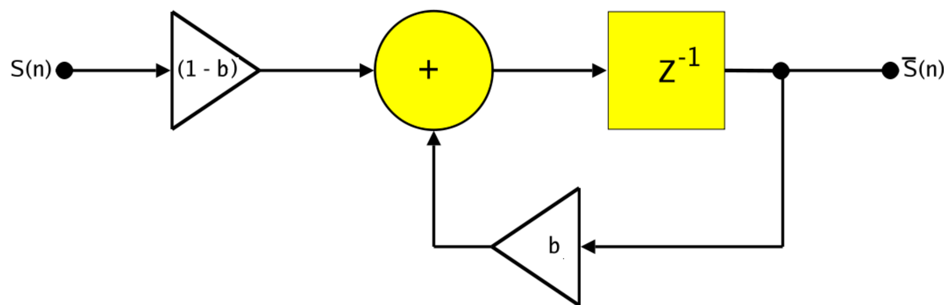


Figure 16. Signalflussdiagramm eines Tiefpaß-Filter 1. Ordnung

Aufgabe

1. Welchen Einfluss hat der Parameter b_1 auf das Übertragungsverhalten?

Annotation

- Bisherige Annahme: Fliesskommaarithmetik mit beliebiger Auflösung und Wertebereich
- In der Realität werden aber Berechnungen in Werteintervallen verarbeitet, z.B. Datentyp *integer[8]* bedeutet ein Werteintervall (Wertemenge) {-128,..,127}
- Signal- und Datenflussgraphen erlauben Annotierungen für und Propagationsanalyse von Datenwortbreiten und Skalierungen, die für den RTL Entwurf genutzt werden können.
- RTL Entwurf vermeidet die Verwendung von Fliesskommaarithmetik (Ressourcenbedarf!), und verwendet stattdessen skalierte Ganzzahl- oder Festpunktarithmetik.
- Daher müssen beim RTL Entwurf die Wortbreiten von Registern und arithmetischen Verarbeitungseinheiten festgelegt werden → Werteberiche von Funktionseinheiten nicht immer unmittelbar ableitbar

➤ Festlegung und Skalierung der Datenwortbreite im Datenpfad findet in Abhängikeit von benötigten Wertebereichen und Operation statt (n: Bitbreite inkl. Vorzeichenbit):

$$+(n, n) \equiv n + 1$$
$$*(n, n) \equiv 2n$$
$$-100, 99, .., 0, .., 100 \equiv n = 8$$

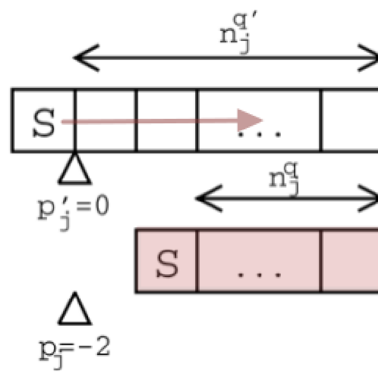


Figure 17. Skalierung und Wortlängenverschiebung durch Bitstellenverschiebung mit Quantisierern [SODA,Constantinides et al.,2004]

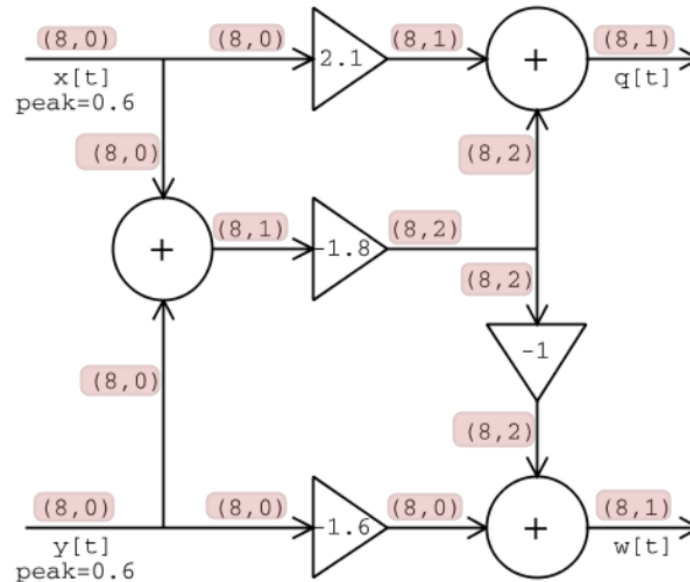


Figure 18. Beispiel einer Annotierung eines Berechnungsgraphens mit Wordlängen-Propagierung (n-bit,scale) [SODA,Constantinides et al.,2004]

4.5. Simulation von Signalflussdiagrammen

SIGFLOW

- Dieser Simulator beschreibt einen Signalflussdiagramm textuel über einer Netzliste
- Alle Berechnungselement können annotiert werden (Datenwortbreite) → Integerarithmetik
- Die Simulation erfolgt ereignisbasiert und schrittweise
- Der zeitliche Verlauf von Signalen kann tabellarisch aufgelistet werden

Diagrammelemente

- **Summierer** `SUM(?width,?monitor)`
- **Verstärker** `MUL(k:number,?width,?monitor)`

- **Verzögerung** `R(?width,?monitor)`
- **Generator** `GEN(options)`
- **Monitor** `MON(options)`

Figure 19. Fensterbasierte GUI vom SIGFLOW Simulator (WEB App)

Beispiel eines Simulationsmodells

```
var C = 0.5
var nodes = {
  // x:GEN({type:'pulse',start:20,end:60,on:100,off:0,bit:8,monitor:true}),
  x:GEN({type:'sin',min:20,max:60,period:10,bit:8,monitor:true}),
  m:MUL(C,8,true),
  s:SUM(10,true),
  r:Z(10,true),
  y:OUT(8,true),
  mon:MON('y(rms)','rms')
}
var net = {
  m:[nodes.x],
  s:[nodes.m,nodes.r],
  r:[nodes.s],
  y:[nodes.s],
  mon:[nodes.y],
}
model={nodes:nodes,net:net}
```

4.6. Kodierung von Zahlenwerten

Ganzzahlkodierung

- Gewichtete Binärzahlenkodierung und Wertebereich

$$z_{(10)} = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} 2^i b_i,\ W = \{-2^{N-1}, .., 0, .., 2^{N-1} - 1\}$$

- Negative Zahlen: Zweierkomplement bilden!

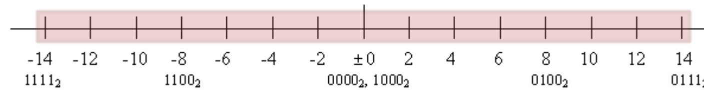
Festpunktkodierung

$$v b_{n-2} b_{n-1} .. b_0 = (-1)^v 2^k + \sum_{i=0}^{N-2} 2^i b_i,\ k < 0$$

- 2^k ist ein Skalierungsfaktor mit $k < 0$ und verschiebt den Dezimalpunkt (Verscheibung von gebrochenen Teil in Ganzzahlteil)

Beispiele:

- N = 4, k = 1:



- N = 4, k = -1:

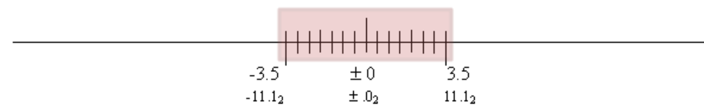
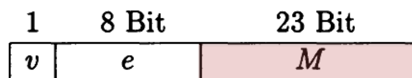


Figure 20. Beispiele für Wertebereiche und Auflösung bei der Festpunktkodierung

Fliesskommakodierung

IEEE-Gleitpunktzahlen: Im IEEE Standard 754-1985 für binäre Gleitpunktzahlen und -arithmetiken werden zwei *Grundformate* spezifiziert:

Einfach langes Format (*single format*): Formatbreite $N = 32$ Bit



Dieses Format entspricht dem Datentyp *single* in MATLAB, der ausschließlich zur Speicherung von einfach genauen Gleitpunktzahlen dient.

Doppelt langes Format (*double format*): Formatbreite $N = 64$ Bit

[5]

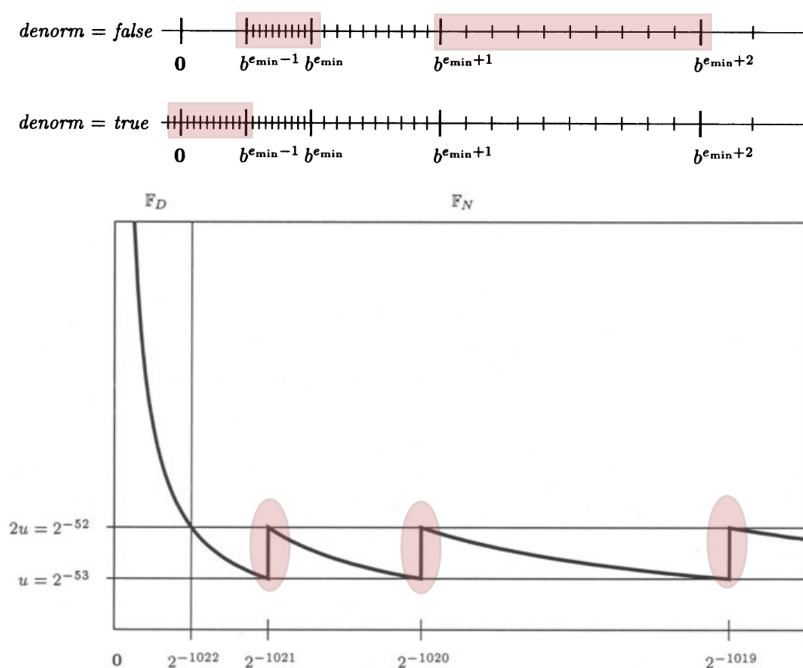


Figure 21. (A) Beispiele für Wertebereiche und Auflösung bei der Fliesspunktkodierung (oben ohne, unten mit Denormalisierung und lückenlosen Wertebereich um Nullpunkt) (B) Relative Abstände von Zahlenwerten in Abhängigkeit vom absoluten Wertebreich [5]

4.7. Signalflussdiagramm: Kombinatorische Logik

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

A. Spezifikation

Definition des Problems mit mathematischen Formalismus:

$$x = x_0 c_0 + x_1 c_1 + x_2 c_2$$
$$y = f(x)$$
$$f(x) = \begin{cases} x, x > 0 \\ 0, x \leqslant 0 \end{cases}$$

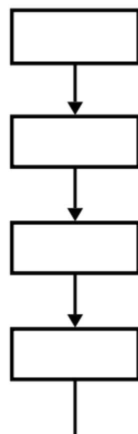
wobei $\{c_0,c_1,c_2\}$ Konstanten sind.

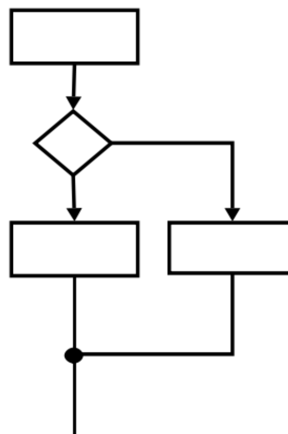
B. Algorithmus

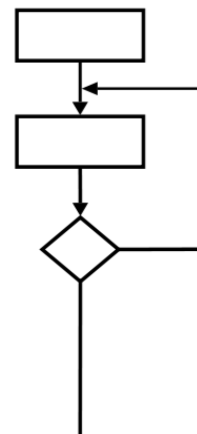
- ➤ Auf algorithmischer Ebene wird das Problem in einzelne atomare Teilschritte zerlegt.
- ➤ Das Datenflußdiagramm zerlegt dabei die Aufgabe in Teilprozesse und führt eine Partitionierung durch, das Kontrollflußdigramm nimmt eine zeitliche Ordnung vor.
- ➤ Das Kontrollflußdiagramm beschreibt den sequenziellen Kontrollfluß, der sich aus dem Algorithmus und der Spezifikation ergibt.
- ➤ Aus einem Flussdiagramm lassen sich direkt imperative Programme ableiten.

Kontrollfluss

- ➤ Einer *linearen Sequenz*,
- ➤ einer *bedingten Verzweigung*, und
- ➤ *Schleifen* als sequenzielles Implementierungsmittel eines Algorithmus.







Programm

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

$y \leftarrow x_0 * c_0$
$y \leftarrow y + x_1 * c_1$
$y \leftarrow y + x_2 * c_2$
IF $y < 0$ **THEN** $y \leftarrow 0$ **END**

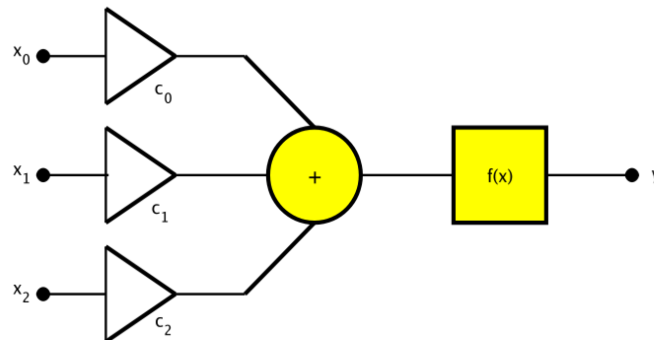
Alternativ mit Schleife (die eine Form lässt sich immer die andere transformieren):

$y \leftarrow 0$;
FOR $i = 0$ **TO** 2 **DO**
  $y \leftarrow y + x(i) * c(i)$;
END
IF $y < 0$ **THEN** $y \leftarrow 0$ **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!

Signalflussdiagramm

- 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

- Kombinatorische Logik besteht nur aus Grundlogikfunktionen wie Und, Oder- und Negierungsverknüpfung, deren Ausgänge nur eine Funktion der aktuell anliegenden Eingangssignale sind
  - Ausnutzung von Nebenläufigkeit von Teilprozessen
  - Parallelisierte Digitallogik

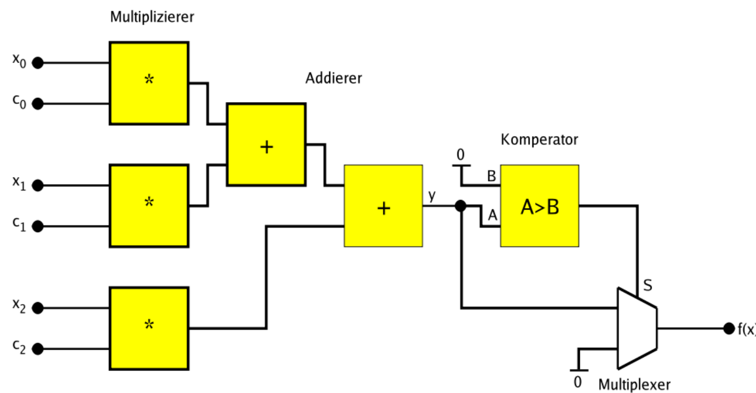


Figure 22. Digitallogikschaltung direkt aus dem Signalflussdiagramm abgeleitet

Hardware Modell

- Mit einer Hardware-Beschreibungssprache wie VHDL kann das Verhalten mi wenigen Anweisungen beschrieben werden.
- VHDL-Beschreibung der kombinatorischen Logik aus dem vorherigen Beispiel:

```
signal x0,x1,x2: bit_vector(7 downto 0);
constant c0,c1,c2: bit_vector(7 downto 0) := X"12";
signal s1,s2,s3: bit_vector(7 downto 0);
signal s,y: bit_vector(7 downto 0);
...
s1 <= x0*c0; s2 <= x1*c1; s3 <= x2*c2;
s <= s1+s2+s3;
y <= s when s > X"00" else X"00";
```

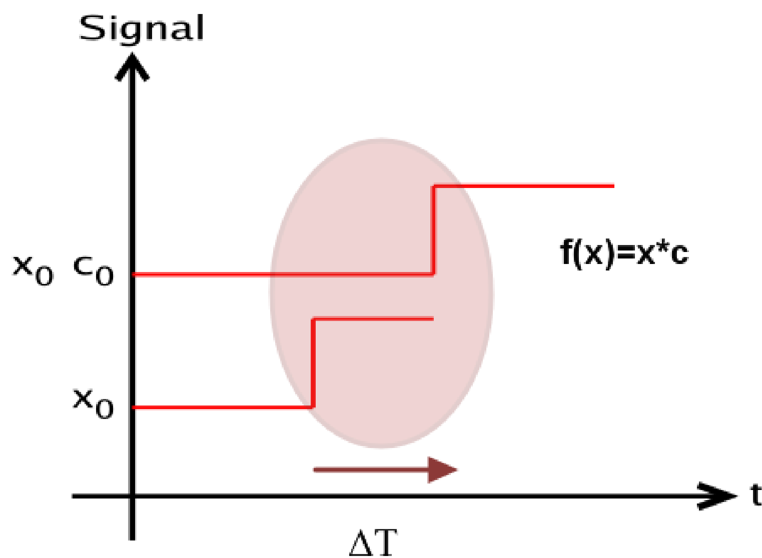
4.8. Signallaufzeit

- 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:

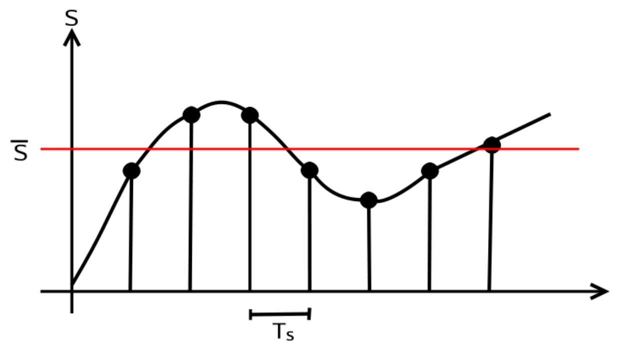
$$T_{signal} = \sum_i T_i^L + \sum_j T_j^E$$

- Signalverzögerung zwischen Ein- und Ausgang einer Digitallogikschaltung:



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

Diskrete Abtastung eines analogen Signals und Mittelwert



➤ Berechnung des Mittelwerts einer Folge diskreter Werte mittels:

$$\bar{S}(N) = \frac{1}{N} \sum_{n=1}^{N \to \infty} S(n)$$

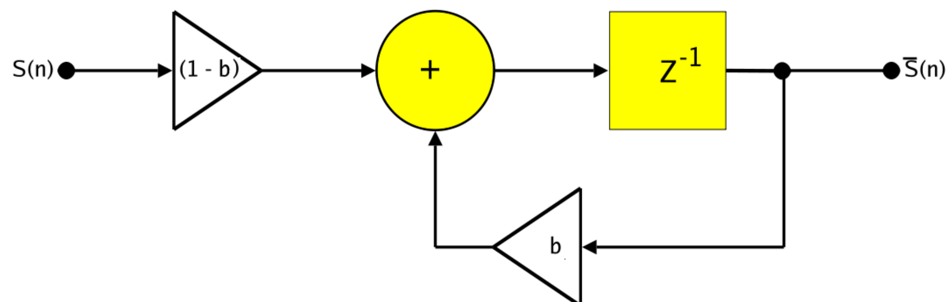
➤ Nicht geschlossen mit einem Signalflußdiagramm darstellbar, da es einen ausgewiesenen Anfangs- und Auswertezustand geben muß (Initialisierung erforderlich)

➤ N ist die Anzahl der Samples und ist variabel!

➤ Näherung: Exponentielle Mittelwertbildung mit einem Tiefpaß-Filter 1. Ordnung (Exponentielle Mittelwertbildung):

$$\bar{S}(n) = S(n)(1-b) + b\bar{S}(n-1)$$

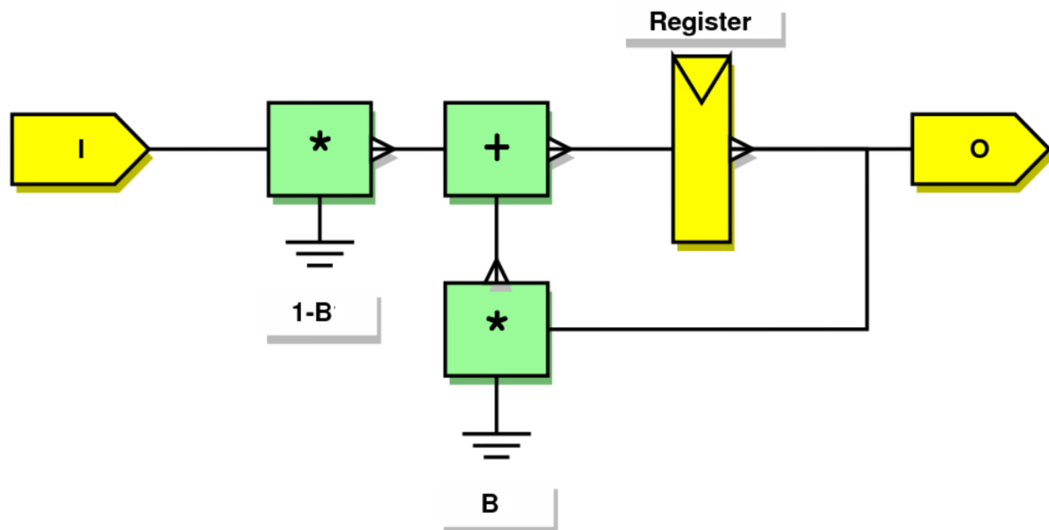
Exponentielle Mittelwertbildung mit rückgekoppelten Filter 1. Ordnung



- 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 im Tiefpaßfilter gewählt wird, desto größer ist der Einfluß von Signalwerten aus der Vergangenheit.

Abbildung des Signalflussdiagramms auf Register-Transfer Logik

- Z-Glieder werden mit taktgesteuerten Registern implementiert

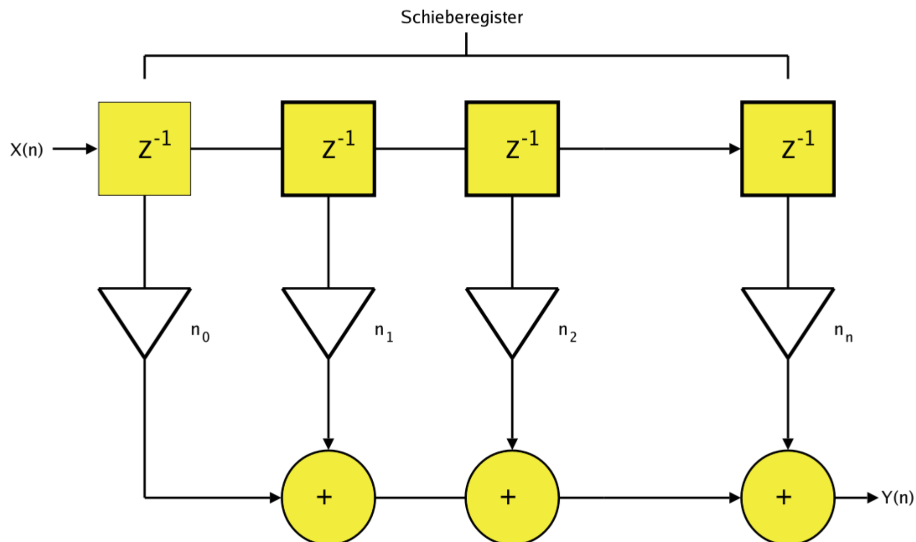


VHDL

```
signal Z:signed(8);
signal X:signed(8);
signal Y:signed(8);
signal S1:signed(8);
signal M1:signed(8); signal M2:signed(8);
...
Zreg: process (CLK)
begin
  if CLK'event and CLK='1' then
    Z <= S1;
  end if;
end process;
M1 <= X * B1;
S1 <= M2 + M1;
M2 <= Z * B;
Y <= Z;
...
```

4.9. Signalflussdiagramm: FIR Filter

FIR-Filter (Finite-Impulse-Response)



➤ Spektraler Filter mit höherer Ordnung ℵ>1 benötigen ℵ

Verzögerungsglieder.

➤ Die Verzögerungsglieder bilden ein Schieberegister.

➤ Schieberegister lassen sich mit generischen Mikroprozessoren und imperativen Programmiersprachen nur aufwendig realisieren, aber in Digitiallogik sehr effizienz umsetzen.

4.10. Signalflussdiagramm: Digitallogik

Aufgabe

1. Erstelle das Signalflussdiagramm eines IR TP 1. Ordnung in DSP DIGFLOW
2. Ermittle für verschiedene Frequenzen das Verhältnis der Größe von Eingangs- und Ausgangssignal (verwende einen rms Monitor)
3. Wiederhole Aufgabe 2. für drei verschiedene *b* Werte
4. Erstelle ein Frequenz-Signalstärke Diagramm (Verhältnis E/A) und bestimme die Grenzfrequenz bei der E/A=0.5 (-3dB)
5. Erstelle das Signalflussdiagramm eines FIR TP 3. Ordnung in DSP DIGFLOW
6. Wiedehole 2-4

4.11. Sequenzielle Systeme

Schieberegister

Man unterscheidet zwei verschiedene Verfahren, Schieberegister zu implementieren:

Ripple Delay

➤ Bei diesem Verfahren wird das Schieberegister linear im Speicher abgelegt.

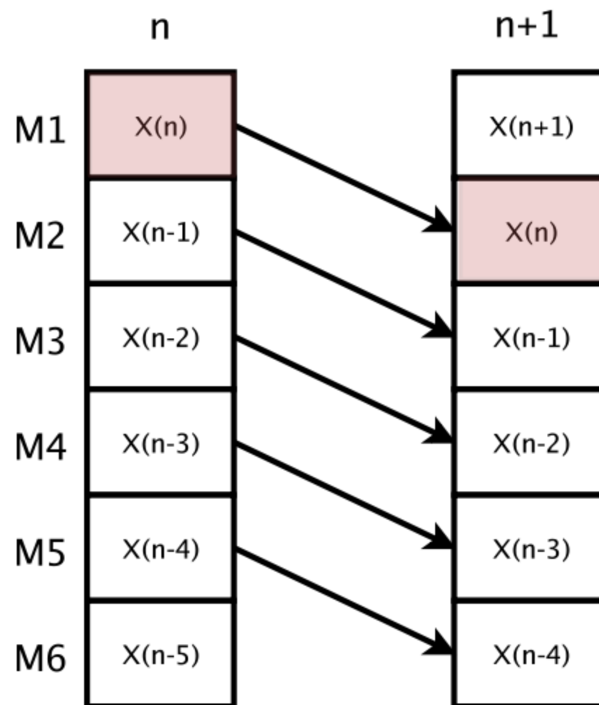
➤ Ein Schiebezyklus erfordert 2(ℵ-1)+1 Speicherzugriffe bei einem ℵ-tiefen Schieberegister, um die Daten zu verschieben.

➤ **Hardware**

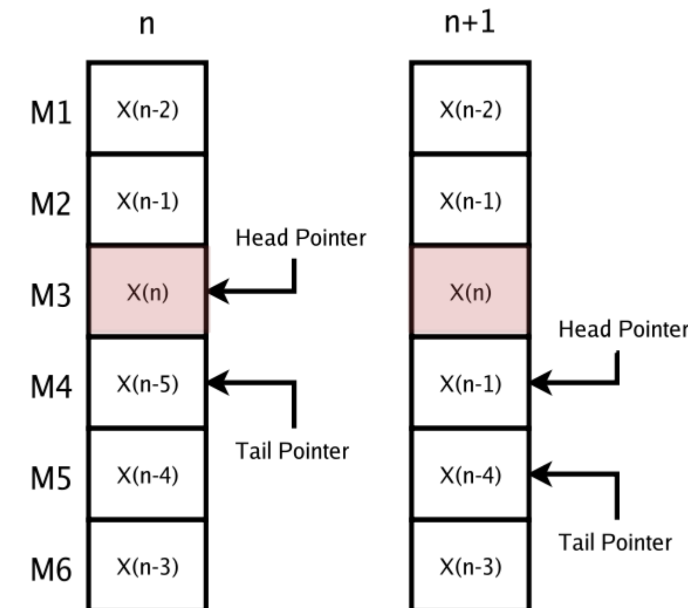
Cyclic Delay

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

Ripple Delay



Cyclic Delay



Schieberegister

- 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.

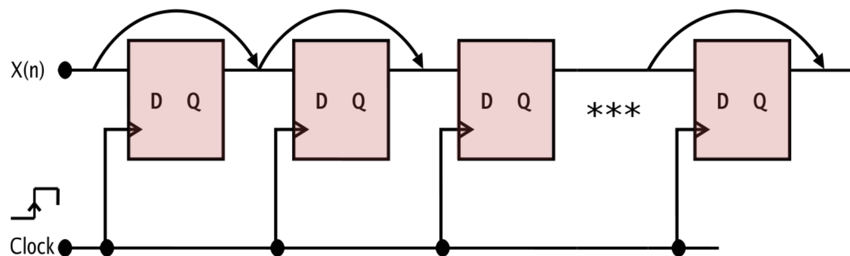


Figure 23. Digitales Schieberegister (Links höchstes, rechts niederwertigstes Bit → rechtsschiebend)

➤ Mit einer Hardwarebeschreibungssprache wie VHDL kann das Verhalten eines ℵ-tiefen und R-bit breiten Schieberegisters mit wenigen Verhaltensanweisungen beschrieben werden.

Example 1. (*VHDL Beschreibung eines Schieberegisters*)

```
signal shift_reg: std_logic_vector(7 downto 0);
...
process()
begin
  if clk'event and clk='1' then
    for i in 7 downto 1 loop
      shift_reg(i) <= shift_reg(i 1);
    end loop;
    shift_reg(0) <= d;
  end if;
end;
y <= shift_reg(7);
```

4.12. Hardware- und Softwarentwurf

Kategorien

Programmgesteuerte generische Maschinen
: Hardware ist Mikroprozessor der für generische Anwendungen ausgelegt ist und programmgesteuert ist. Das Programm ist die Software die die Anwendung beschreibt.

Erweiterbare Prozessoren
: Generische Prozessoren die durch Erweiterungen (i.A. Optimierungen) spezialisiert werden können. Die Anwendung ist teils in Hardware, überwiegend in Software enthalten.

Konfigurierbare Prozessoren
: Hardware-Software-Codesign. Sowohl der Prozesor als auch das Programm sind anwendungsspezifisch.

Register Transfer Logik
: Die Anwendung befindet sich vollständig in Hardware, die nicht mehr programmgesteuert sondern vollstädnig anwendungsspezifisch ist → Reine Digitallogikschaltung und reines Hardwaredesign.

Erweiterbare Mikroprozessoren

- Als Kompromiss-Lösung bieten sich sog. erweiterbare Prozessoren an: μP-Kern ist vorgegeben, kann aber erweitert werden durch
  - Spezielle Befehle, z.B. ℵ×R-Schieberegister;
  - Register (Anzahl und Datenbreite);
  - Funktionen/Operatoren.
- μP-Peripherie kann anwendungsspezifisch entworfen und hinzugefügt werden (Anzahl, Funktion).

Vorteile

- Compiler existieren und sind getestet;
- μP-Kern ist verifiziert;
- Flexbilität durch Softwareentwurf.

Nachteile

- Immer noch generischer Ansatz mit Nachteilen: Peformanz, Energieeffizienz

Konfigurierbare Mikroprozessoren

- Als Kompromisslösung bieten sich weiterhin sog. vollständig konfigurierbare Prozessoren an.
- μP-Kern ist frei definierbar und anwendungsspezifisch:
  - Spezielle und generische Befehle, z.B. ℵ×R-Schieberegister,
  - Allgemeine und spezielle Register (Anzahl und Datenwortbreite);
  - Funktionen/Operatoren (Datenwortbreite, Stelligkeit);
  - Anwednungsspezifische μP-Peripherie (Anzahl, Funktion).

Vorteile

- Flexbilität durch optimale Anpassung des Mikroprozessors an Problemspezifikation

Nachteile

- ➤ Compiler existieren nicht, müssen erst aus Prozessorspezifikation abgeleitet werden
- ➤ μP-Kern ist nicht getestet und verifiziert.
- ➤ Weiterhin eingeschränkte Parallele Datenverarbeitung

4.13. Parallele Datenverarbeitung

Parallelisierung

Partitionierung eines Algorithmus und Programms in mehrere nebenläufige Einheiten (Prozesse), die auf N Prozessoren parallel ausgeführt werden.

- ➤ Anwendungsspezifisch angepasste Architektur (Speicher,..) erfordferlich!

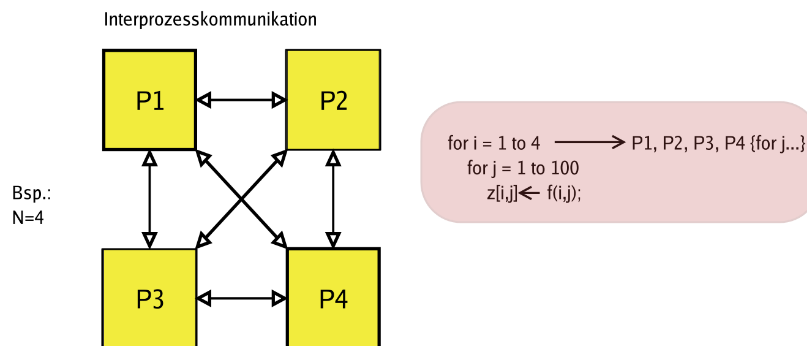


Figure 24. Parallelisierung eines Algorithmus (hier verschachtelte Schleife)

Parallelität durch Multiprozessorsysteme

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 (IPC, Schutz von kritischen Programmbereichen, z.B. gemeinsame genutzte Datenstrukturen, Abhängigkeiten der Instruktionsreihenfolge usw.)

Es gibt zwei verschiedene Hardware/Software-Lösungen für eine N-Prozessor-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 Rechnersysteme 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.

Parallelisierung

Der Programmfluss kann in Daten- und Kontrollfluss zerlegt werden, die jeweils im Daten- und Kontrollpfad eines Datenverarbeitungssystems verarbeitet werden.

Kontrollpfad
: Parallelität auf Prozessebene (Multithreading) mit Interprozesskommunikation → Mittlere bis geringe Beschleunigung, mittlerer Overhead

Datenpfad
: Parallelität auf Dateninstruktionsebene ohne explizite Kommunikation → Hohe Beschleunigung, geringer Overhead

4.14. Register Transfer Logik (RTL)

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

1. Datenfluss → Datenpfade
2. Kontrollfluss → Kontrollpfad → Zustandsautomat

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

Elemente des Datenpfades

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

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

Zustandsautomat

➤ Der Zustandsautomat ist für die Ablaufsteuerung in einem Systemblock zuständig und ist zentraler Bestandteil.

1. Der Zustandsautomat steuert den Datenfluß im Datenpfad;
2. Er ist für die Ein- und Ausgabesteuerung zuständig,
3. Implementierung von Handshake-Protokollen für den Datenaustausch mit anderen Systemblöcken (auch IPC), und
4. Er behandelt Ausnahmesignale und Fehler.

➤ Taktgesteuert findet der Übergang zwischen verschiedenen Zuständen des Systems statt.

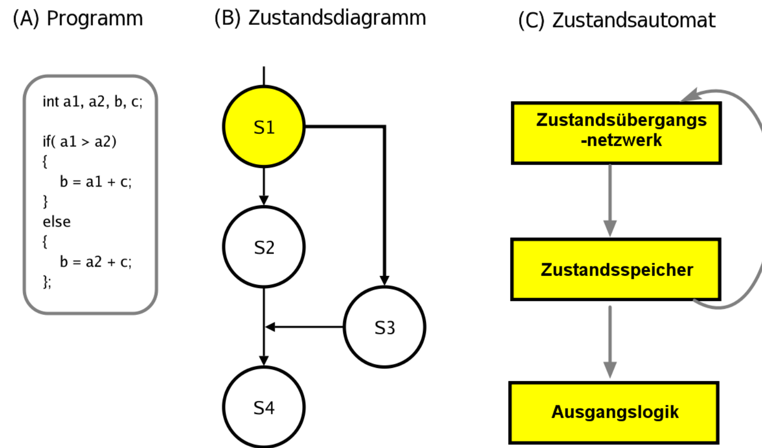


Figure 25. Beispiel Zustandsdiagramm und Ablaufsteuerung

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

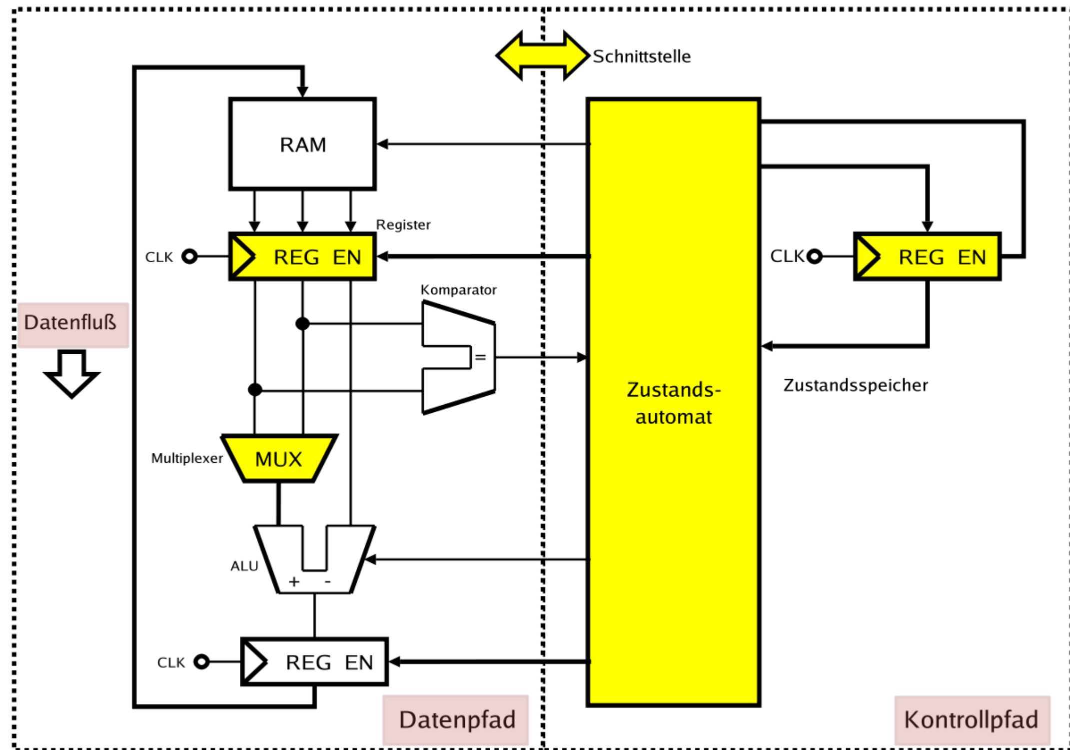


Figure 26. Beispiel RTL (Links) Datenpfad (Rechts) Kontrollpfad

5. Digitallogik

5.1. Logische Zustände

- Logische Variablen besitzen die 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 Spannungsbreiche (Intervalle) verwendet, z.B. für die TTL-Technologie, die mit einer Versorgungsspannung von 5V betrieben wird

0 $\rightarrow$ *Low* $\rightarrow [0, 0.8\,V]$
1 $\rightarrow$ *High* $\rightarrow [3, 5\,V]$

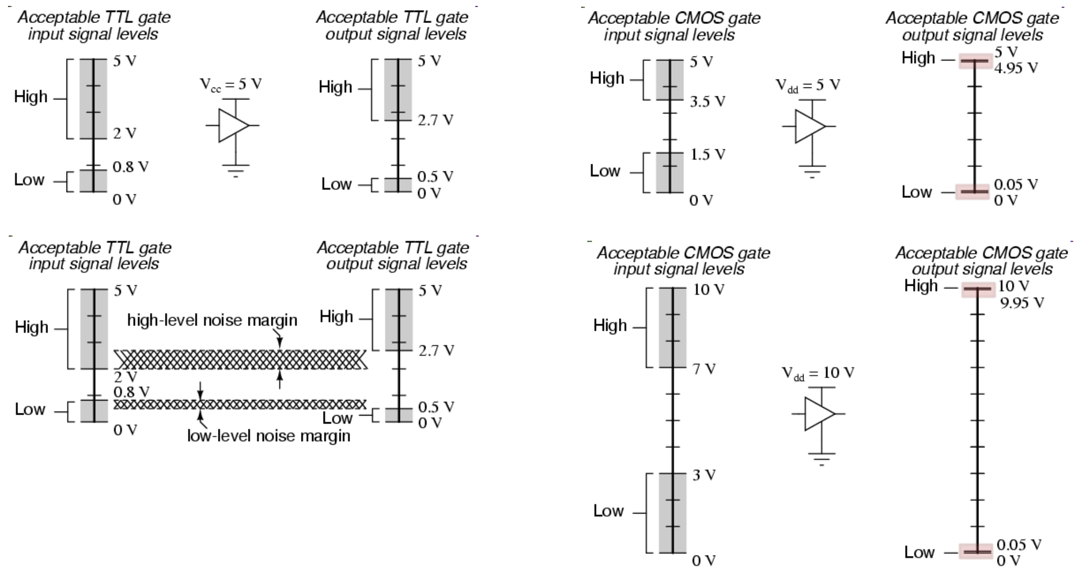
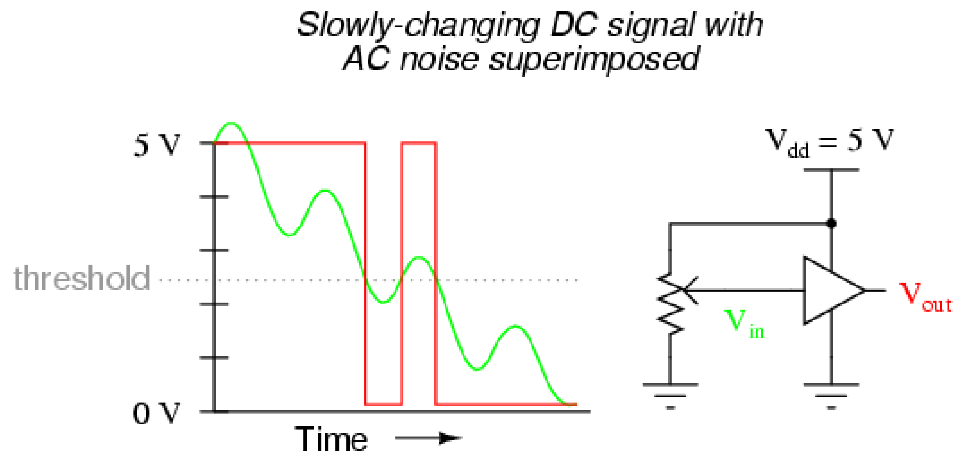


Figure 27. Eingangs- und Ausgangsspannungsbereiche von Digitallogik abhängig von Technologie und Versorgungsspannung (Links) TTL (Rechts) CMOS [1]

- 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.
- Weiterhin können sich Logikpegel nihct beliebig schnell ändern ($0 \rightarrow 1$, $1 \rightarrow 0$), und es gibt immer eine Zeitspanne in der sich ein technisches Logiksignal in einem undefinierten Zustand befindet!



[1]

5.2. Schaltungstechnologien

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

Transistor-Transistor-Logic (TTL)

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, Heute dominierender Technologieprozeß mit folgenden Eigenschaften:

- Spannungsgesteuerte Stromquellen
- Spannungsversorgung: 1-15V
- Geringe statische Verlustleistung, geringe dynamische Verlustleistung bei Schaltaktivität.

➤ Schaltgeschwindigkeiten technologieabhängig, im Bereich 1-10ns

ECL

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

➤ Stromgesteuerte Stromquellen

➤ Spannungsversorgung: NECL → -5V

➤ Hohe Verlustleistung auch ohne Schaltaktivität.

➤ Sehr hohe Schaltgeschwindigkeit ≤ 1ns

5.3. Logikgatter

Logische Grundfunktionen der kombinatorischen Logik

Inverter

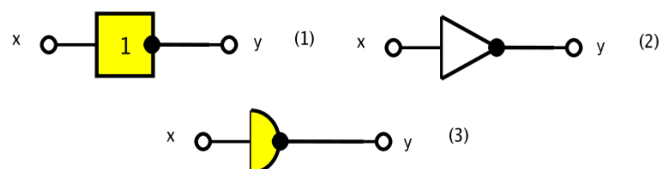
➤ Logische Negierung einer Eingangsvariable x, Boolesche Algebra:

$$y = f(x) = x = \neg x = NOT(x)$$

➤ Negation: Funktionstabelle

x	**y=¬ x**
0	1
1	0

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



Inverter: CMOS Transistorschaltung

CMOS: Complimentary Metal Oxide Substrate Technologie

- Die Transistorschaltung besteht aus einem sog. N-Kanal (unten) und dazu im Verhalten komplementären P-Kanal (oben) **MOS-Feldeffekttransistor**, mit selbstsperrenden 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 der Transistoren, d.h. der Kennlinie eines N-/P-MOSFET-Transistors.
- Vereinfacht kann ein Transistor als steuerbarer Schalter verstanden werden. Jedoch: Ein FET Transistor ist eine spannungsgesteuerte Stromquelle.

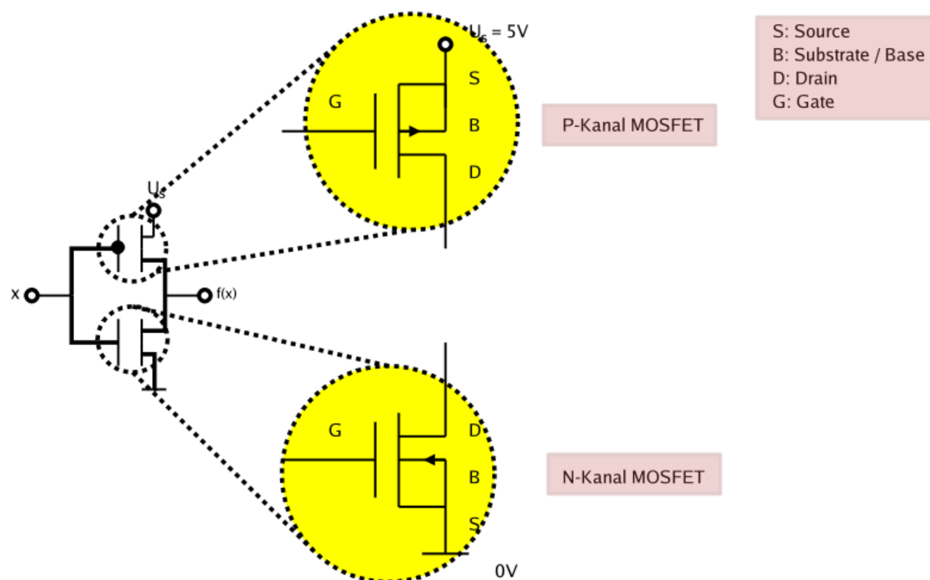


Figure 28. Zwei komplimentäre MOSFET Transistoren bilden einen logischen Inverter

CMOS Transistorschaltung

- 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 Ladungssenke, 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.

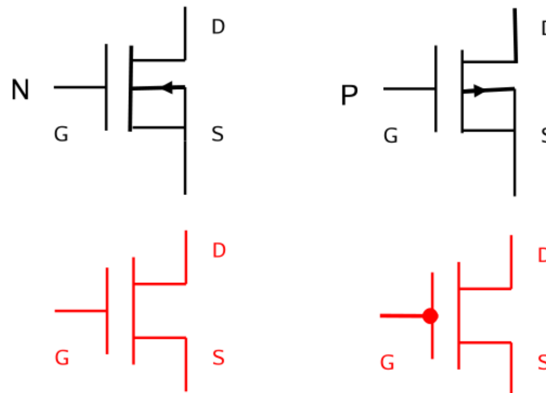
- Ein sog. selbstsperrender Transistor ist dadurch gekennzeichnet, dass bei einer Gate-Source-Spannung UGS=0V kein Drain-Strom fließt, man spricht von einem sperrenden Transistorzustand.

- Der andere Zustand eines Transistors ist der leitende Zustand, bei dem ein elektrischer Strom zwischen Source und Drain-Anschluss IDS fließen kann.

- 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.

Elektronische Schaltsymbole von MOSFET Transistoren



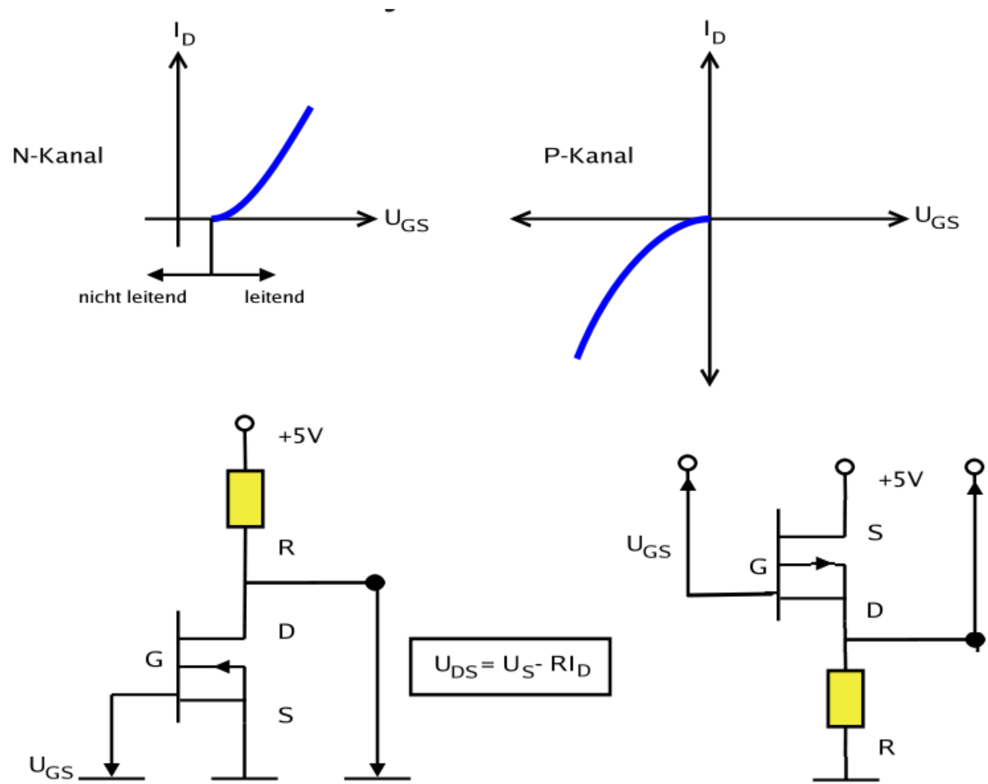


Figure 29. Kennlinie U_{GS}-U_{DS} eines N- und P-Kanal-MOSFET Transistors

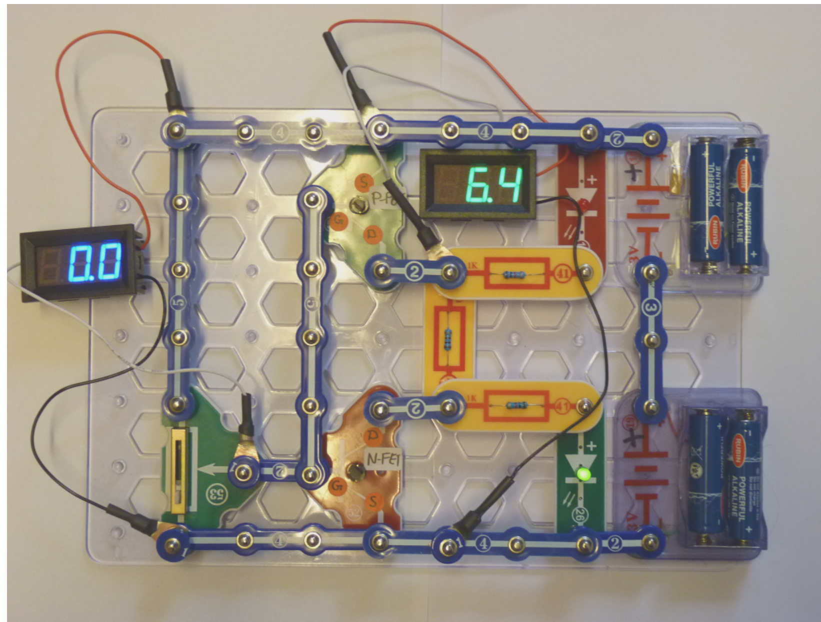


Figure 30. Experimenteller Aufbau einer Inverterschaltung mit zwei MOSFET Transistoren (N- und P-Kanal)

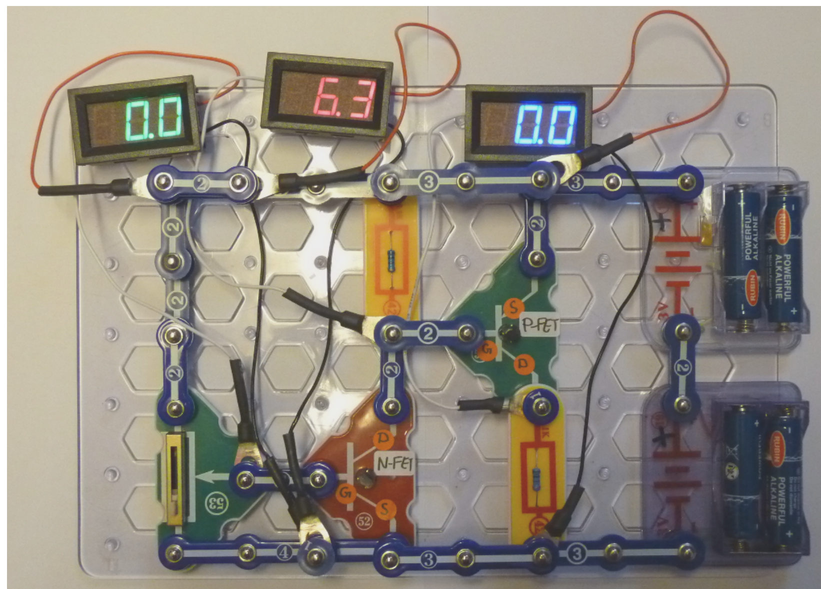


Figure 31. Experimenteller Aufbau einer Doppelinverterschaltung mit zwei MOSFET Transistoren (N- und P-Kanal)

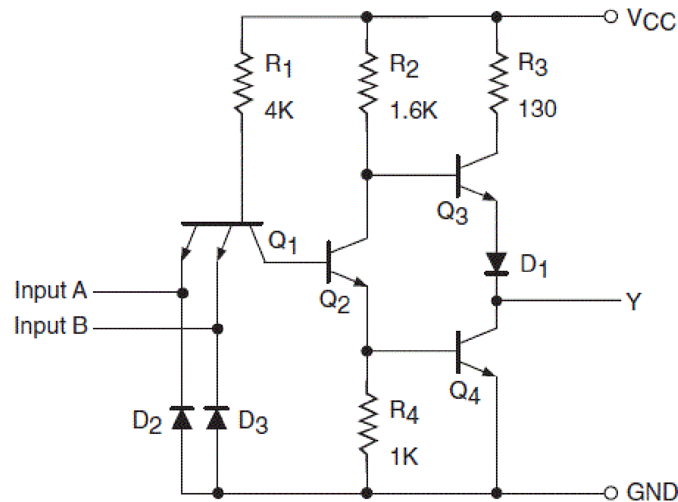


Figure 32. Zum Vergleich: Standard TTL NAND Gatter mit bipolaren Transistoren

Aufgaben

1. Man kann einen Inverter mit zwei Transistoren oder einem Transistor und einem Widerstand aufbauen. Überlege, was elektrotechnisch der Vorteil und der Nachteil wären.
2. Warum sind bipolare Transistoren gegenüber Feldeffekttransistoren in der Digitallogik unterlegen?
3. Welchen Nachteil haben FET Transistoren (physikalisch)
4. Wo kommen die für einen P-Kanal MOSFET benötigten negativen Steuerspannungen her?

Und Gatter

Logische Funktion: Und-Verknüpfung → Konjunktion

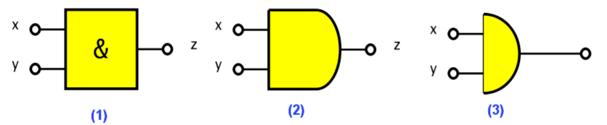
➤ Logische Verknüpfung zweier Eingangsvariablen &arr; 1-Bit Multiplizierer.

$$f(x,y) = x \wedge y = x \bullet y$$

➤ Konjunktion: Funktionstabelle

x	y	z=f(x,y)	¬z
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

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



➤ CMOS Technologie ist grundsätzlich invertierend → Inverter ist Elementarzelle der CMOS Logik!

➤ Daher ist einfachste Implementierung eine Und Gatters ein Nicht-Und Gatter (NAND) → 4 Transistoren

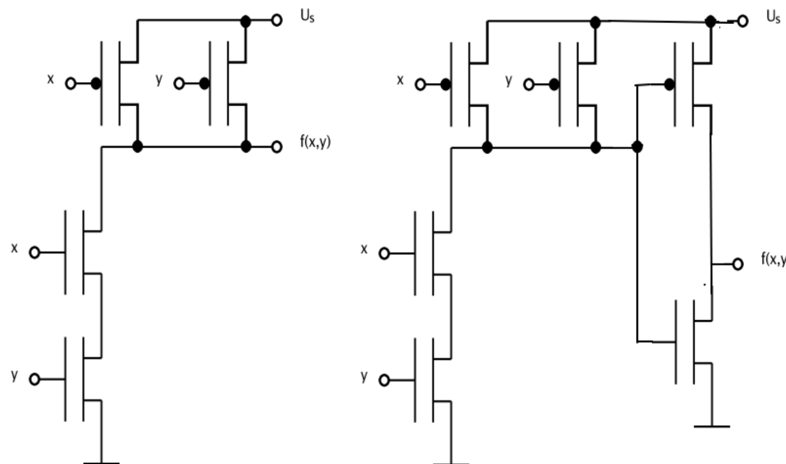


Figure 33. Transistorschaltung eines NAND und AND Gatters

Oder Gatter

Logische Funktion: Oder-Verknüpfung → Disjunktion

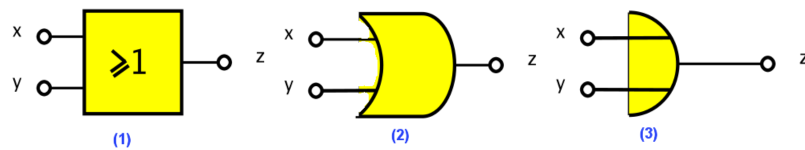
➤ Logische Verknüpfung zweier Eingangsvariablen.

$$f(x, y) = x \vee y = x + y$$

➤ Disjunktion: Funktionstabelle

x	y	z=f(x,y)	¬z
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

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



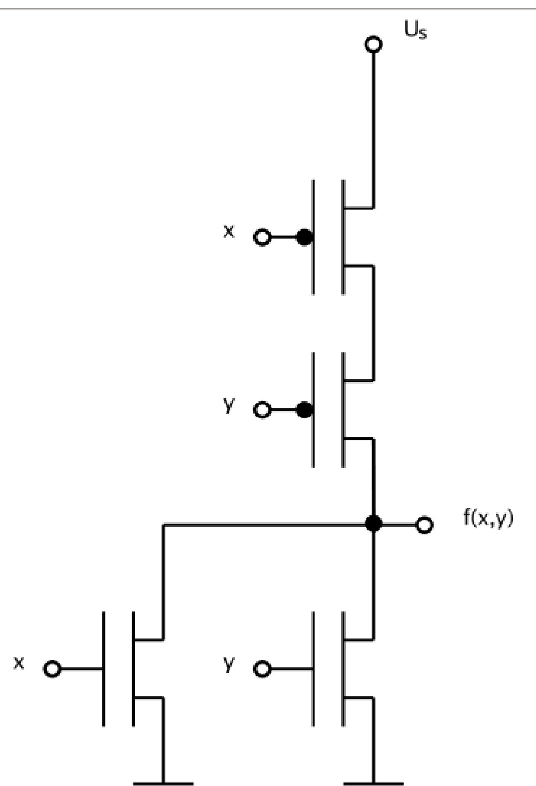


Figure 34. Transistorschaltung eines NOR Gatters

EXOR Gatter

Logische Funktion: Exklusiv-Oder-Verknüpfung → 1-Bit-Addierer!

➤ Logische Verknüpfung zweier Eingangsvariablen.

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

➤ EXOR Funktionstabelle

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

5.4. Logik Simulation

- Eine Logikschaltung besitzt Ein- und Ausgänge, d.h., beschrieben durch Vektoren $\boldsymbol{X}=(x_1,x_2,..,x_n)$ und $\boldsymbol{Y}=(y_1,y_2,..,y_m)$
- Eine Logikschaltung ist eine Netzliste aus elementaren Logikgattern (z.B. aus einer Standardbibliothek)
- Ein Logiksimulator kann ereignisbasiert die Änderung der Ausgangssignale bei einer Änderung der Eingangssignale und allen inneren Signalen berechnen
- Wichtig: Ein technologisches Zeitmodell für die Signalausbreitung ist erforderlich und muss vom Simulator unterstützt werden
  - Jedes elektrische Signal hat eine endliche Ausbreitungs- und Änderungsgeschwindigkeit
  - Jede Transistorschaltung und jedes Gatter fügt Verzögerungen τ_i in einen Signalpfad ein

RETRO Simulator

- RTL Simulator mit Verzögerungszeitmodell

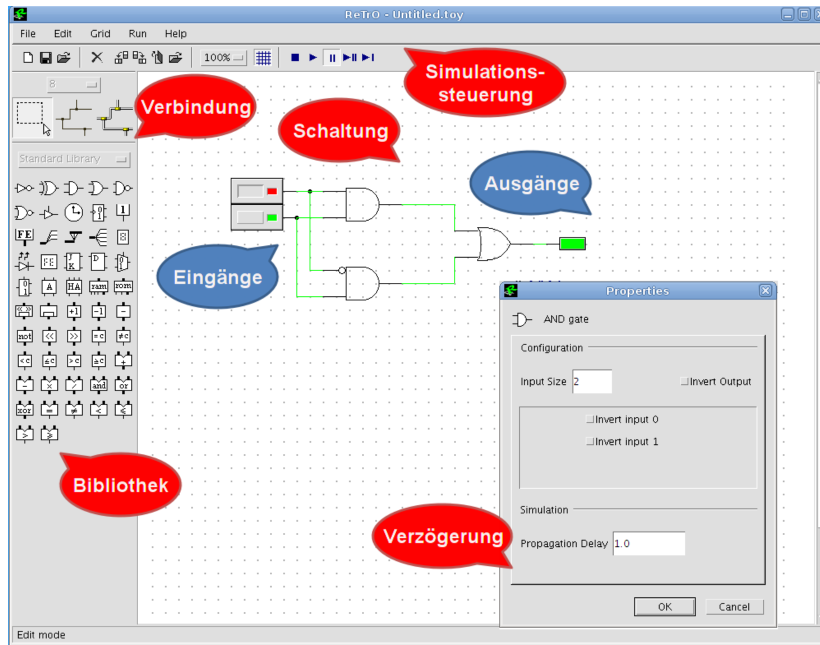


Figure 35. ReTro Simulator und GUI

Logiktest

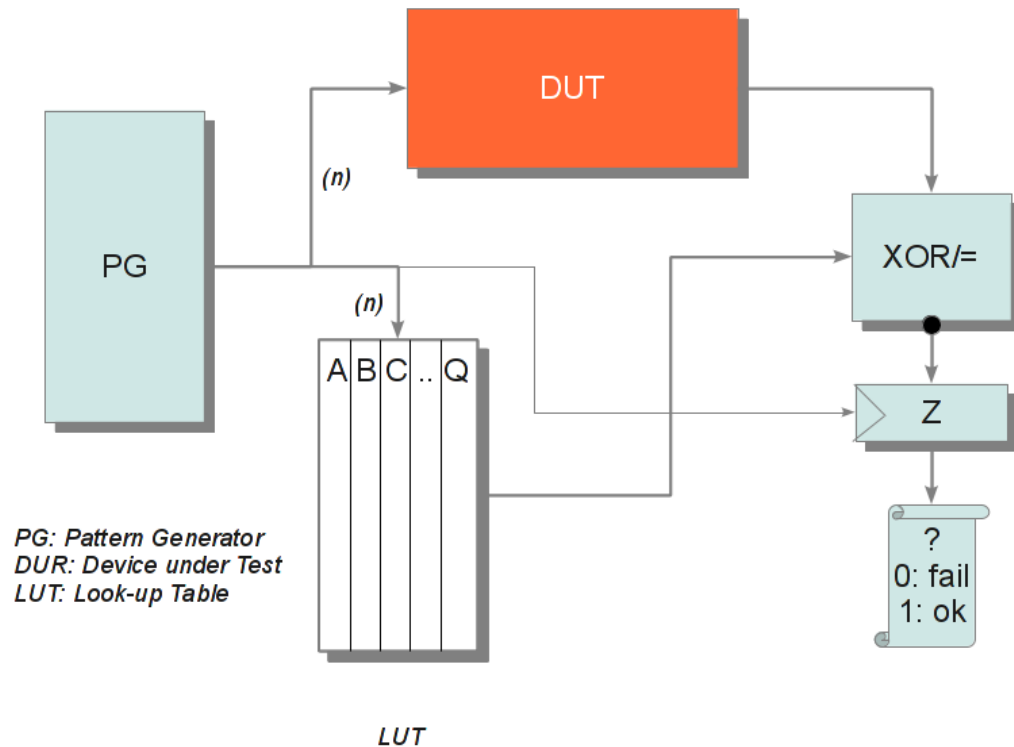


Figure 36. Allgemeiner Aufbau eines automatischen Logiktesters (Äquivalenztest mit Funktionstabelle)

➤ Ein Pulsgenerator zusammen mit einer Look-up Tabelle (LUT) erzeugen die Testmuster und steuert den Test.

  ❏ Ein Pulsgenerator kann auch aus einem Binärzähler (n-Bit) aufgebaut werden

  ❏ Die Look-up Tabelle definiert die Abbildung aller möglichen Muster der n-Bit Eingangssignale auf m-Bit Ausgangssignale → erwartetes Ergebnis

➤ Das Eingabemuster wird gleichzeitig auf die LUT und die zu testende Schaltung (DUT, Dvice under Test) gegeben

➤ Ein Komparator oder bei m=1 ein EXOR Gatter wird verwendet um das Ergebnis zu evaluiren

- ➤ Ein nachfolgendes Register “sampled” das Vergleichsergebnis (Assertion true/false)

  - ❏ Warum ist ein Register erforderlich?

Aufgaben

1. Implementiere die Boolesche Gleichung eines EXOR Gatters in ReTro mit elementaren Logikgattern
2. Füge die Schaltung in eine Testumgebung ein (siehe Abb. 36), um die Wahrheitstabelle testen zu können.

Lösung

5.5. Boolesche Algebra

Definition 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,

$$a = e_1 + e_2 + e_3 + .. + e_n$$

Konjunktion
Und-Verknüpfung von n Eingangswerten zu einem Ausgangswert,

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

Negation
Invertierung eines booleschen Wertes $\neg e$

- ➤ Boolesche Werte besitzen eine Wertmenge {0,1}.

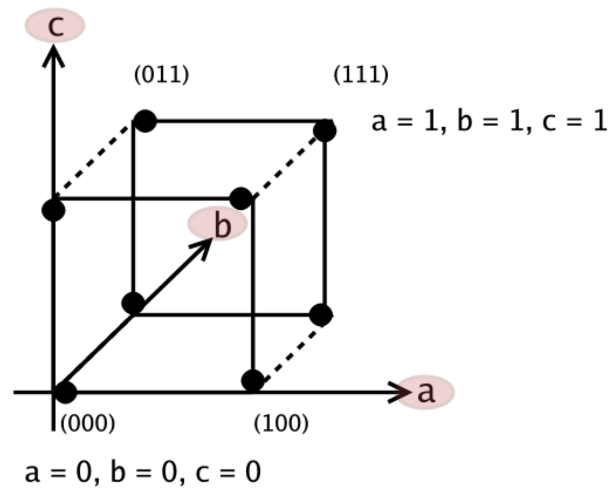
- Boolesche Funktionen bilden n Variablen (n-dimensionaler Vektor) auf m Ergebniswerte (m-dimensionaler Vektor) ab:

$$f : B^n \rightarrow B^m$$

- I.A. ist m=1, d.h. Verwendung von **skalaren booleschen Funktionen**.
- Jede m-dimensionale Boolesche Funktion kann in einen Satz aus m skalaren Funktionen zerlegt werden (ohne weitere Transformation)

$$f(a,b,c) = \begin{pmatrix} a+b+c \\ a+\neg b+c \\ \neg a+b+c \end{pmatrix} \Leftrightarrow$$
$$f_1(a,b,c) = a+b+c$$
$$f_2(a,b,c) = a+\neg b+c$$
$$f_3(a,b,c) = \neg a+b+c$$

- 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. Graphische Darstellung eines Booleschen Vektors (a,b,c):



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

Wahrheits-/Funktionstabellen

- Boolesche Algebra ist Hilfsmittel beim Entwurf von digitalen Schaltungen.
- Eine Aufgabenstellung definiert Schaltbedingungen, die in einer Funktions-/Wahrheitstabelle dargestellt werden.
- Dabei können in einer Tabelle beliebige m-dimensionale Boolesche Funktionen als Verhaltensbeschreibung dargestellt werden, d.h., wie Eingangs- auf Ausgangswerte abgebildet werden.

$\mathbf{switch_1}$	$\mathbf{switch_2}$	$\mathbf{motor_1}$	$\mathbf{light_2}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	0

Normalformen

- 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.
- 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:

$$f(a_1, a_2, a_3, .., a_n) = (x_1^1 \bullet x_2^1 \bullet x_3^1 \bullet .. \bullet x_n^1) + (x_1^2 \bullet x_2^2 \bullet x_3^2 \bullet .. \bullet x_n^2) + ..$$
$$x_i^j = \{a_i, \neg a_i\}$$

➤ Ableitung der SOP Normalform aus Funktionstabelle:

a_i	a_j	**q**	
x	x	1	→ Teilterm
x	x	0	

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.

Beispiel einer SOP Normalform

A	**B**	**Q**	
0	1	1	→ (¬ A • B)
1	0	1	→ (A • ¬ B)
0	0	0	
1	1	0	

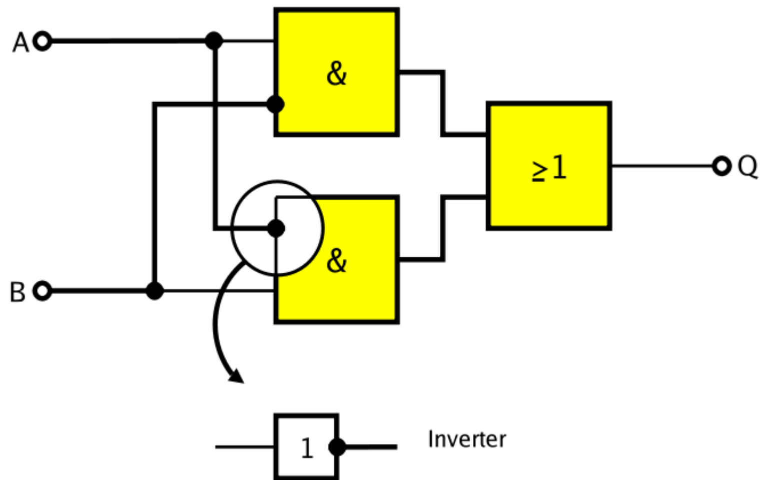
→ Q = (¬A • B) + (A • ¬B)

Abbildung auf Digitallogik

➤ Jede Boolesche Operation (+ *) wird auf Logikgatter abgebildet.

➤ Negation werden auf Inverter abgebildet und ggfs. direkt in einer Verknüpfung integriert

Die boolesche Funktion Q führt zu folgender Digitallogikschaltung:



Konjunktive Normalform

- 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:

$f(a_1, a_2, a_3, .., a_n) = (x_1^1 + x_2^1 + x_3^1 + .. + x_n^1) \bullet (x_1^2 + x_2^2 + x_3^2 + .. + x_n^2) \bullet ..$

$x_i^j = \{a_i, \neg a_i\}$

- Ableitung der SOP Normalform aus Funktionstabelle:

$\mathbf{a_i}$	$\mathbf{a_j}$	**q**	
x	x	0	→ Teilterm
x	x	1	

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.

Beispiel einer POS Normalform

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

→ Q = (¬ A + ¬ B) • (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.

Technisches Beispiel

➤ Eine Schaltmatrix mit drei Eingängen $\boldsymbol{X}$=(A,B,C) und einem Ausgang Y soll bestimmt werden.

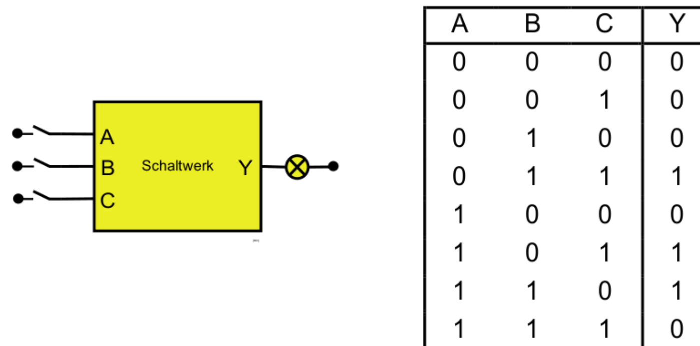
❏ An den Ausgang Y ist eine Lampe angeschlossen.

❏ Die Eingänge werden über Schalter gesteuert:

Taste 1 wird gedrueckt → *A* = *1*
Taste 2 wird gedrueckt → *B* = *1*
Taste 3 wird gedrueckt → *C* = *1*

➤ Immer wenn zwei Tasten gleichzeitig gedrückt werden (X_i=1,X_j=1), soll die Lampe leuchten (Y=1).

➤ Aufgabe: Bestimmung der Funktionstabelle aus obigen Anforderungen und Vereinbarungen



A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figure 37. Strukturelle und Verhaltensbeschreibung einer technischen Schaltung mit einer Funktionstabelle

Aufgabe

1. Leite die Disjunktive Normalform aus der Wahrheitstabelle des technischen Beispiels ab.
2. Leite die Konjunktive Normalform aus der Wahrheitstabelle des technischen Beispiels ab.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Bestimmung der disjunktiven Normalform aus Funktionstabelle.

1. Zeilen der Funktionstabelle, die als Ergebniswert y=1 besitzen, generieren einen Teilterm.
2. Jeder Teilterm setzt sich aus dem Produkt aller Eingangsvariablen a_i derart zusammen dass gilt:

$$x_{i-te\ Variable}^{Term=Zeile} = \begin{cases} a_i & \text{wenn Eingangswert} = 1 \\ \neg a_i & \text{wenn Eingangswert} = 0 \end{cases}$$

3. Alle Teilterme werden summiert

Term 1: $\neg A \bullet B \bullet C$

Term 2: $A \bullet \neg B \bullet C$

Term 3: $A \bullet B \bullet \neg C$

DNF: $Y(A, B, C) = \neg A \bullet B \bullet C + A \bullet \neg B \bullet C + A \bullet B \bullet \neg C$

Bestimmung der konjunktiven Normalform aus Funktionstabelle.

1. Zeilen der Funktionstabelle, die als Ergebniswert y=0 besitzen, generieren einen Teilterm.

2. Jeder Teilterm setzt sich aus der Summe aller Eingangsvariablen a_i derart zusammen dass gilt:

$$x_{i-te\ Variable}^{Term=Zeile} = \begin{cases} a_i & \text{wenn Eingangswert} = 0 \\ \neg a_i & \text{wenn Eingangswert} = 1 \end{cases}$$

3. Alle Teilterme werden summiert

Term 1: $A + B + C$, Term 2: $A + B + \neg C$

Term 3: $A + \neg B + C$, Term 4: $\neg A + B + C$

Term 5: $\neg A + \neg B + C$

DNF: $Y(A, B, C) = A+B+C \bullet A+B+\neg C \bullet A+\neg B+C \bullet \neg A+B+C \bullet \neg A+\neg B+C$

Termumformungen

- Mittels der booleschen Termumformungen können boolesche Ausdrücke mit folgenden Zielen umgeformt werden:
- Reduzierung der Verknüpfungen,
- Verbesserung der Signalaufeigenschaften
- Transformation auf eine bestimmte Technologie, z.B. nur Nicht-Und-Verknüpfungen.

Kommutativ-Gesetze mit 2 Variablen

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

$$x \bullet y = y \bullet x$$
$$x + y = y + x$$

Assoziativgesetze mit 3 Variablen

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

$$(x \bullet y) \bullet z = x \bullet (y \bullet z) = x \bullet y \bullet z$$
$$(x + y) + z = x + (y + z) = x + y + z$$

Distributivgesetze mit 3 Variablen

Eine gemeinsame Variable in zwei verknüpften Termen kann ausgeklammert werden.

$$(x \bullet y) + (x \bullet z) = x \bullet (y + z)$$
$$(x + y) \bullet (x + z) = x + (y \bullet z)$$

➤ Die zweite Gleichung kennt keine Analogie in der gewöhnlichen Algebra!

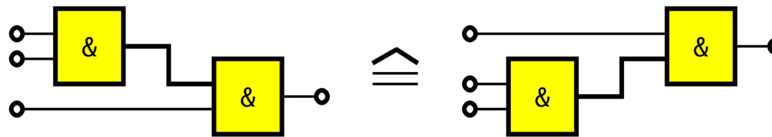


Figure 38. Beispiel der Anwendung des Assoziativgesetzes mit 3 Variablen

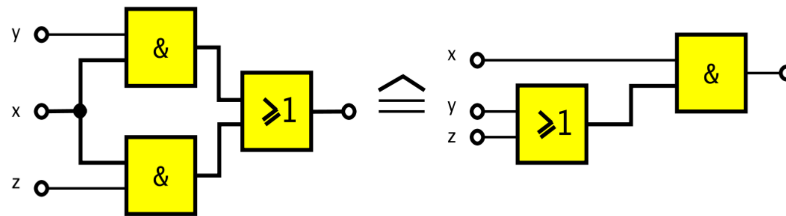


Figure 39. Beispiel für eine 3 → 2 Logikgatterminimierung mit dem Distributivgesetz und 3 Variablen

Inversionsgesetze mit 2 Variablen

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

$$(\neg x \bullet \neg y) = \neg(x + y)$$
$$(\neg x + \neg y) = \neg(x \bullet y)$$

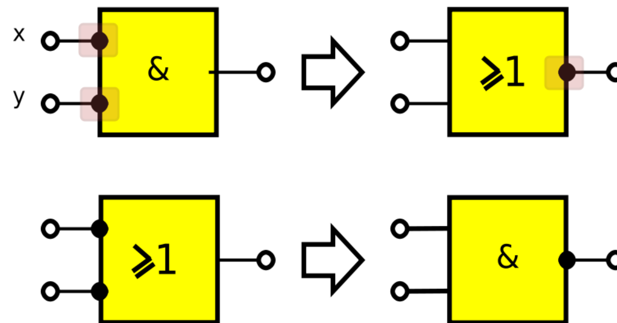


Figure 40. De-Morgansche Regeln und Technologietransformation

Gesetzte der booleschen Algebra für Termumformungen

$$
\begin{array}{cc}
a \bullet 0 = 0 & a \bullet 1 = a \\
a + 0 = a & a + 1 = 1 \\
a \bullet a = a & a + a = a
\end{array}
$$

Bindungsregeln

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

5.6. Technische Logikzustände

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 starken Signal überlagert bzw. zusammengeschlossen werden (elektrisch: Kurzschluss!).

1

Starker logischer Wert *True*. Ein Signal mit diesem Wert darf mit keinem anderen starken Signal überlagert bzw. zusammengeschlossen werden.

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 dieses Zustände ignoriert werden, mit der Möglichkeit der Logikoptimierung.

Logiktypen in VHDL

bit

Zweiwertige Logik {0,1}

std_logic

Mehrwertige Logik {0,1,L,H,Z,X,..}

Example 2. (*Beispiele in VHDL*)

```
signal x: std_logic_vector(3 downto 0);
...
process triStateDriver(we,addr)
begin
  if we =  1  and addr="00" then
    x <= "0000";
  elsif we =  1  and addr="01" then
    x <= "0001";
  else
    x <= "ZZZZ"; -- Tristate BUS State Disconnected
  end if;
end process;
```

5.7. Optimierung von Digitallogik

Ziele

1. Verständnis der Ziele von Optimierungsverfahren in der Digitallogik
2. Verständnis und Anwendung verschiedener systematischer Reduktionsverfahren und ihrer Möglichkeiten und Grenzen
3. Grundkenntnis über moderne graphenbasierte Verfahren

Systematische Reduktion logischer Funktionen

Ziel der Minimierung:

- Möglichst kleine Anzahl von Logikgattern bei der elektronischen Implementierung bei gleichzeitig geringer Signallaufzeit,
  - Gerade bei zwei-stelligen Operationen relevant (und Symmetrie wegen Hazards); n-stellige Operationen werden nicht verbessert!
- 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.

Aufgabe

1. Minimiere die folgenden Booleschen Funktion mit Hilfsmitteln der Booleschen Algebra

$$f(a,b) = (a \bullet b) + (\neg a \bullet b) = ?$$
$$f(a,b,c,d) = a \bullet b \bullet c \bullet \neg b \bullet d = ?$$
$$f(a,b,c) = \neg(\neg a \bullet (\neg b + \neg c)) = ?$$

5.8. 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.
- In jedem Feld wird der zu den Werten der Eingangsvariablen gehörende Ausgangszustand/wert $F_{i,j}$={0,1} eingetragen.
- Das Diagramm ist zyklisch: der Übergang von rechten Rand zum linken, und von unteren zum oberen ist möglich → Torusoberfläche
- Bei drei Eingangsvariablen reduziert sich das Diagramm um zwei Zeilen.

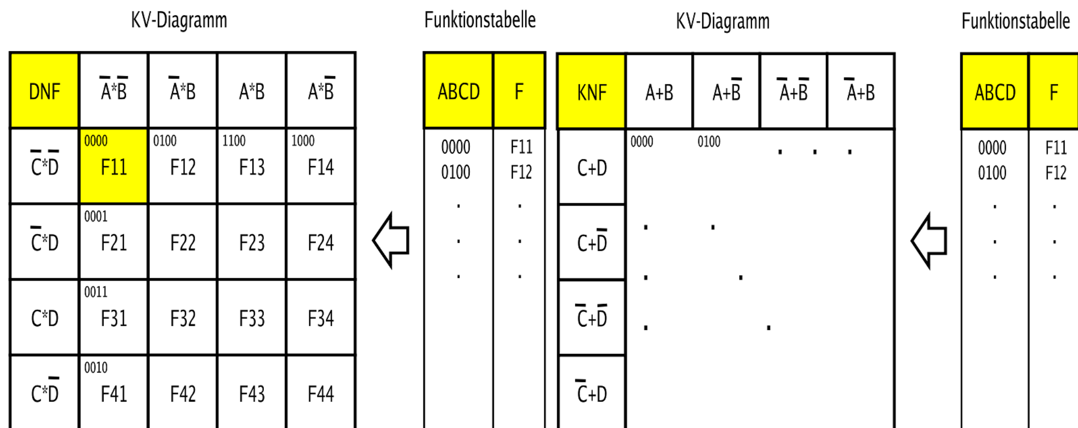


Figure 41. (Links) KV-Diagramm für 4 Variablen A,B,C,D und DNF (Rechts) KV-Diagramm für 4 Variablen A,B,C,D und KNF.

Reduzierte disjunktive Normalform

- Folgende Schritte sind zur Ableitung einer minimalen d.h. reduzierten DNF (RDNF) notwendig:
- Benachbarte Felder $F_{i,j}$=1 werden zu Flächen mit 2N Elementen zusammengefasst. Die größtmöglichen Flächen/Gruppen sollen gebildet werden.
- Alle Felder müssen in mindestens einer Fläche/Gruppe erfasst werden.

- Ableitung eines neuen Teilterms der RDNF aus einer Fläche/Gruppe: Produkt aus allen Variablen, die allen Feldern der Gruppe gemeinsam sind.
- Die Teilterme werden summiert.

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

Figure 42. Beispiel: KV-DNF

- Die reduzierte DNF folgt dann:

$$f(A,\ B,\ C,\ D)\ = \neg A \bullet \neg C + A \bullet C \bullet D + \neg B \bullet C \bullet \neg D$$

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 von 196 → 44, und eine Verringerung der Chipfläche um den Faktor 2.11

Reduzierte konjunktive Normalform

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

1. Benachbarte Felder $F_{i,j}$=0 werden zu Flächen mit 2N 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.

5.9. QM Verfahren

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.

Beispiel: $Q = A \bullet \neg B \bullet C \bullet \neg D + A \bullet B \bullet D$.

➤ Der 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 ⇒ Q=1 folgt. Jeder Produktterm der DNF ist ein Implikant.

Beispiel: sowohl $A \bullet \neg B \bullet C \bullet \neg D$ als auch $A \bullet B \bullet D$ sind Implikanten von Q.

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

Beispiel: Q=A • ¬B • C • ¬D + A • B • C + ¬A • ¬B • C • ¬D

➤ Erste Term ist kein PI.

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

➤ Dritter Term kein PI, da $\neg B \bullet C \bullet \neg D$ ausreicht, damit Q=1 wird (A=X)

Terme, in denen eine Variable komplementär auftreten, lassen sich verkürzen:

$$A \bullet \neg B \bullet C \bullet \neg D + \neg A \bullet \neg B \bullet C \bullet \neg D =$$
$$(A + \neg A) \bullet \neg B \bullet C \bullet \neg D = \neg B \bullet C \bullet \neg D$$

Schritt I

Alle Produktterme werden in einer Tabelle eingetragen. Für jede Variable wird der Wert eingetragen, damit Q=1 wird (Selektierte Funktionstabelle!). Beispiel:

$$Q = A \bullet B \bullet \neg C \bullet D + A \bullet B \bullet C \bullet \neg D + \neg A \bullet B \bullet C \bullet \neg D +$$
$$A \bullet B \bullet \neg C \bullet \neg D + \neg A \bullet B \bullet \neg C \bullet \neg D + A \bullet \neg B \bullet \neg C \bullet \neg D$$

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) ⊗

⊗: Markierung und Auswahl komplementärer Terme

Schritt II

Terme, die sich nur in einer Variable unterscheiden, heißen ähnlich. Bildung der verkürzten Terme (nm) bedeutet: Ableitung aus Term (n) und (m). Im Beispiel sind das: (1) ⇔ (4), (2) ⇔ (3), (2) ⇔ (4), (3) ⇔ (5), (4) ⇔ (5), (4) ⇔ (6)

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

Schritt III

Für die neuen verkürzten Terme wird Schritt II wiederholt. Der Vorgang endet, wenn keine 1-komplementären Terme mehr auftreten.

(2435) ⇒ X1X0
(2345) ⇒ X1X0

Schritt IV

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

(14) → $A \bullet B \bullet \neg C$
(46) → $A \bullet \neg C \bullet \neg D$

Verkürzte (minimierte) boolesche Funktion:

$$Q' = A \bullet B \bullet \neg C + A \bullet \neg C \bullet \neg D + B \bullet \neg D$$

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:

Im Beispiel: Suche der Überdeckungen zwischen Produkt- und Primtermen:

	AB¬C	A¬C¬D	B¬D
AB¬CD	X		
ABC¬D			X
¬ABC¬D			X
AB¬C¬D	X	X	X
¬AB¬C¬D			X
A¬B¬C¬D		X	

Schritt VI

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

Schritt VII

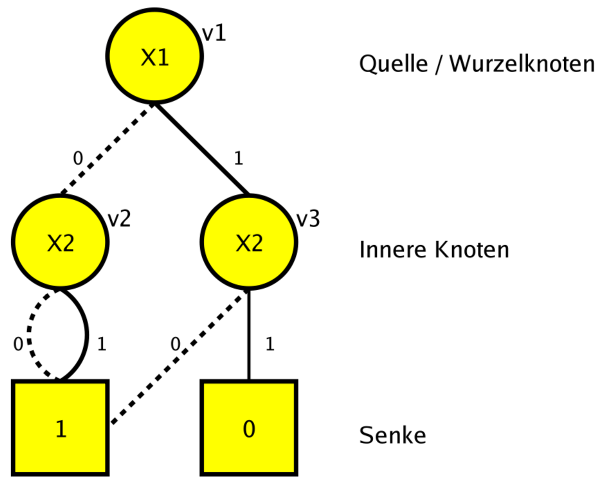
Alle Primterme können weggelassen werden, solange in jeder Zeile mindestens eine Überdeckung vorhanden ist. Im Beispiel ist keine weitere Reduzierung möglich!

5.10. BDD Verfahren

Binary-Decision-Diagram (BDD)

- Ein Binary-Decision-Diagram (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,.., 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.
- Ein Pfad (Quelle $\rightarrow$ Senke) beschreibt die Evaluierung eines Funktionswertes der BF.
- 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).

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



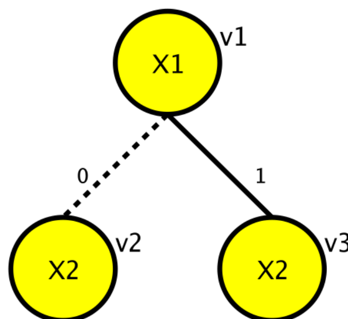
Shanon-Zerlegung

Erzeugung eines BDD's aus BF schrittweise durch Evaluierung der einzelnen Variablen $\{x_1,..,x_2\}$ mit den Werten {0,1}:

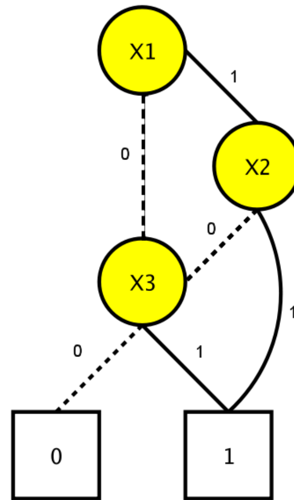
$$f(x_1, x_2, .., x_n) = \neg x_1 f_{|x_1=0} + x_1 f_{|x_1=1}$$

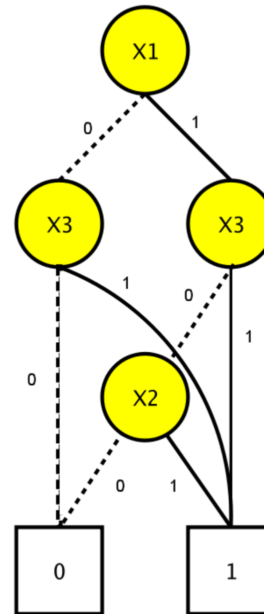
Jeder innere Knoten stellt eine neue (Sub-) BF dar:

$$f_{v1}(x_1, x_2) = \neg x_1 + \neg x_2 = \neg x_1 f_{v2|x_1=0} + x_1 f_{v3|x_1=1}$$
$$f_{v2}(x_2) = 1 + \neg x_2 = 1, f_{v3}(x_2) = 0 + \neg x_2 = \neg x_2$$



- Die Tatsächliche Größe und Struktur eines BDD's hängt von der Variablenreihenfolge bei der Erzeugung ab! (Ausnahme: symmetrische BF).
- Beispiel zweier BDDs erzeugt aus $f(x_1,x_2,x_3)=x_1 \bullet x_2 + x_3$

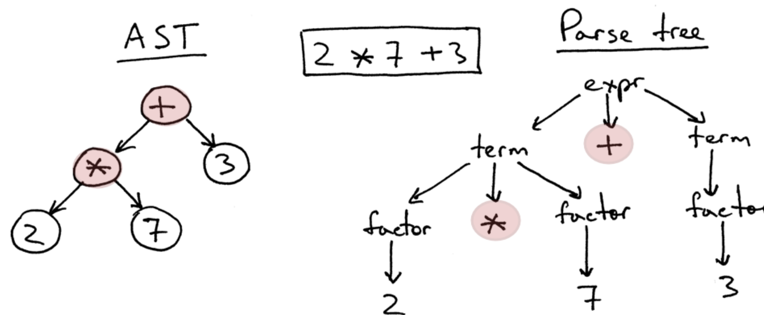




Reihenfolge {x1, x2, x3}   {x1, x3, x2}

Abstrakter Syntaxbaum

- ➤ Alternativ kann die logische Formel (in syntaktischer Notation) zunächst im ersten Schritt durch einen Parser auf einen abstrakten Syntaxbaum (AST) abgebildet werden
  - ❏ Der AST besteht aus binären Operationen (Knoten haben zwei Eingangskanten)
  - ❏ Die Terme werden bereits evaluiert und vereinfacht
- ➤ Im zweiten Schritt wird aus dem AST ein BDD ähnlich der Shanonzerlegung gebildet (aber es wird immer nur ein binärer Term evaluiert)



BDDVIEW

Ordered BDD

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

Reduktion von BDDs

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.

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

Zusammenfassungsregel
: 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 s und f zusammengefasst werden.

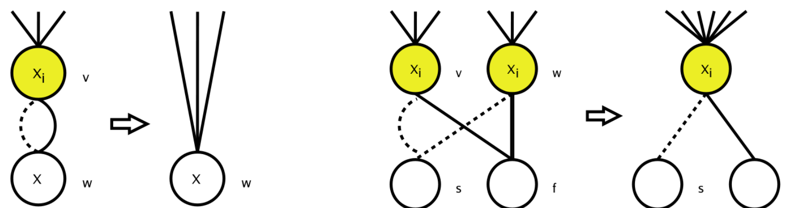
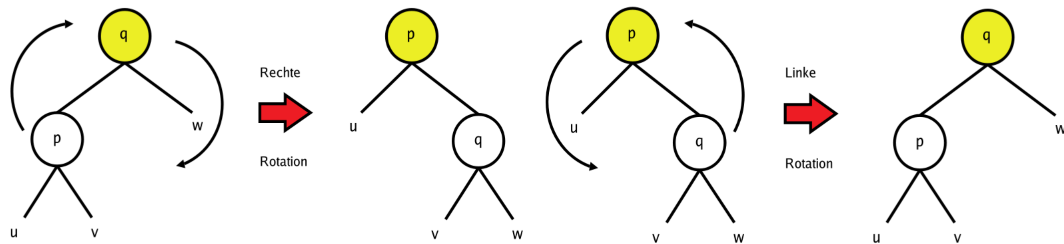


Figure 43. Anwendung der Lösch- und Zusammenfassungsregeln bei einem BDD

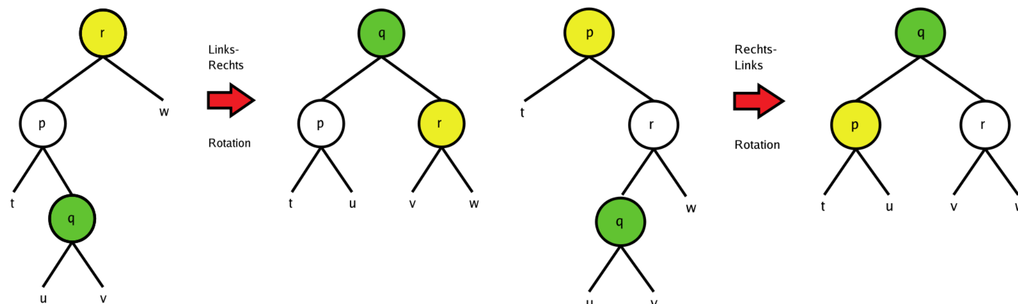
- Für eine möglichst schnelle Suche, optimal $\Theta(\log 2N)$, sollte der Baum balanziert sein, d.h. jeder Knoten hat zwei Nachfolger (außer Endknoten).
- Die Höhe eines Baumes ist dann minimal $H{=}\log 2N$:
- Für die Balanzierung eines Baumes benötigt man zwei Rotationsoperationen:

  1. Rechte Rotation im Uhrzeigersinn von zwei benachbarten Knoten,
  2. und linke Rotation gegen den Uhrzeigersinn.

- Rechte Rotation zweier Knoten q und p. Die weiterführenden Verzweigungen werden entsprechend der Ordnungsrelation umgeordnet (linkes Bild).
- Linke Rotation zweier Knoten q und p. Die weiterführenden Verzweigungen werden entsprechend der Ordnungsrelation umgeordnet (rechtes Bild).



- Kombination aus Links- und Rechtsrotation führt zur Balanzierung eines linkslastigen asymmetrischen Baumes (linkes Bild).
- Kombination aus Rechts- und Linksrotation führt zur Balanzierung eines rechtslastigen asymmetrischen Baumes (rechtes Bild).



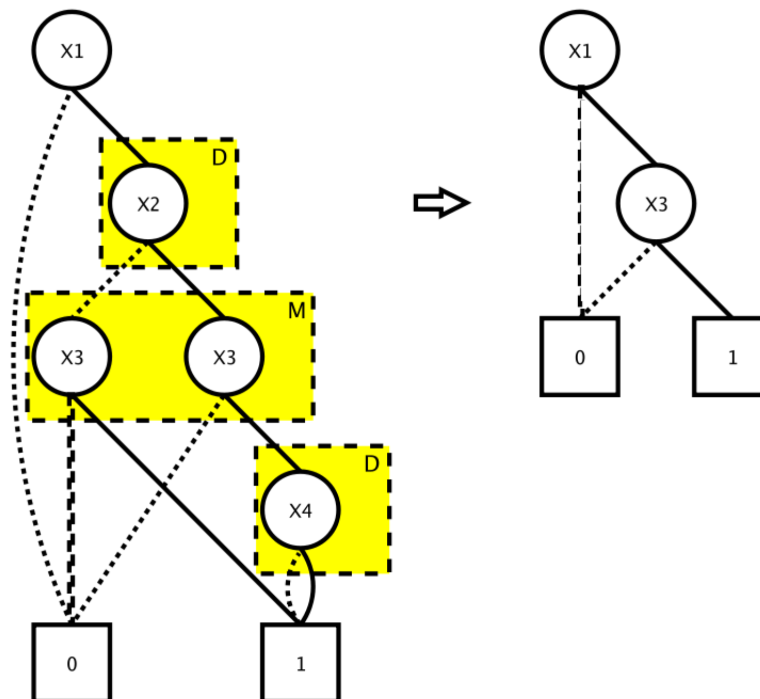
Aufgabe

1. Erstelle von folgender BF einen BDD

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

2. Optimiere den BDD mit den eingeführten BDD Regelen
3. Leite die optimierte BF aus dem BDD ab.

Lösung



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

$$\Rightarrow$$

$$\begin{aligned} f'(x_1, x_3) &= \neg x_1 \bullet 0 + x_1 \bullet (\neg x_3 \bullet 0 + x_3 \bullet 1) \\ &= \neg x_1 \bullet 0 + x_1 \bullet x_3 \\ &= x_1 \bullet x_3 \end{aligned}$$

➤ Die Ableitung einer BF aus einem BDD wendet die inverse Shanon Zerlegung an

- Bei der Ableitung einer BF aus einem BDD beginnt man bei den 0/1 Blättern und aufwärts gehend baut man für jeden Knoten eine BF bis der Wurzelknoten erreicht wurde
- Beispiel: Knoten $v_3(x_3)$ ist $f_3 = \neg\ x_3\ \bullet\ 0 + x_3\ \bullet\ 1 = x_3$

5.11. Hazards

Wenn eine Hazard (Gefahr) in einer digitalen Schaltung auftritt, führt dies zu einer vorübergehenden Änderung der Ausgabe der Schaltung [2].

- D.h., ein Hazard in einer digitalen Schaltung ist eine vorübergehende Störung im idealen Betrieb der Schaltung, die nach einer unbestimmten Zeit von selbst aufgehoben wird.
- Diese Störungen oder Fluktuationen treten auf, wenn unterschiedliche Wege vom Eingang zum Ausgang unterschiedliche Verzögerungen haben, und aufgrund dieser Tatsache Änderungen der Eingangsvariablen den Ausgang nicht sofort ändern, sondern am Ausgang nach einer kleinen Verzögerung erscheinen, die durch die Logikgatter verursacht wird.
- In digitalen Schaltungen gibt es drei verschiedene Arten von Gefahren

  1. Statischer Hazard
  2. Dynamische Hazard
  3. Funktionaler Hazard

Statische Hazards

Formal tritt ein statischer Hazard auf, wenn eine Änderung an einem Eingang dazu führt, dass sich der Ausgang kurzzeitig ändert, bevor er sich auf seinen korrekten Wert stabilisiert. Basierend auf dem korrekten Wert gibt es zwei Arten statischer Gefahren, wie in der folgenden Abbildung dargestellt:

Static-1-Hazard
: Befindet sich der Ausgang aktuell im logischen Zustand 1 und nachdem der Eingang seinen Zustand geändert hat, ändert sich der Ausgang vor dem Setzen auf 1 vorübergehend auf 0, dann handelt es sich um eine Static-1-Gefahr.

Static-0-Hazard
: Befindet sich der Ausgang aktuell im logischen Zustand 0 und nachdem der Eingang seinen Zustand geändert hat, ändert sich der Ausgang vor dem

Setzen auf 0 vorübergehend auf 1, dann handelt es sich um eine Static-0-Gefahr.

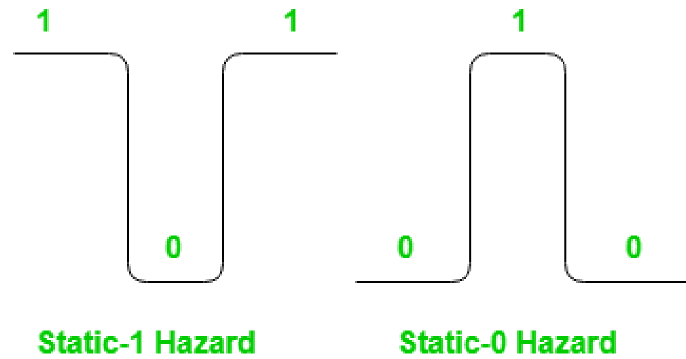


Figure 44. Vergleich von statischen 1- und 0-Hazard [2]

Erkennung statischen Hazards mit K-map

Betrachten wir zuerst die Gefahr von Statik-1. Um eine statische 1-Gefahr für eine digitale Schaltung zu erkennen, werden folgende Schritte ausgeführt:

Schritt 1
Erstelle die BF für den Ausgang der digitalen Schaltung auf, Y(A,B,C,..).

Schritt 2
Erstelle einer K-Map für diese Funktion Y, und notiere alle angrenzenden 1-Werte.

Schritt 3
Wenn es ein Zellenpaar mit 1-Werten gibt, das nicht in derselben Gruppe zu liegen scheint (d.h. ein Prim-Implikant), zeigt dies das Vorhandensein einer statischen 1-Gefahr an. *Jedes dieser Paare ist eine statische Gefahr.*

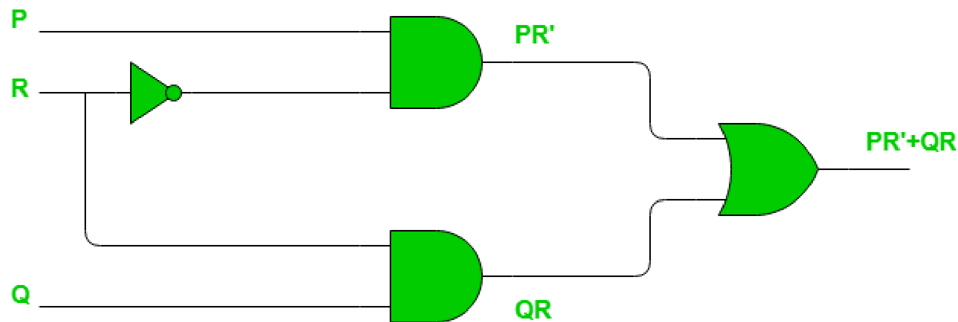


Figure 45. Beispiel Logikschaltung mit Hazards $f(P,Q,R) = Q \bullet R + P \bullet \neg R$

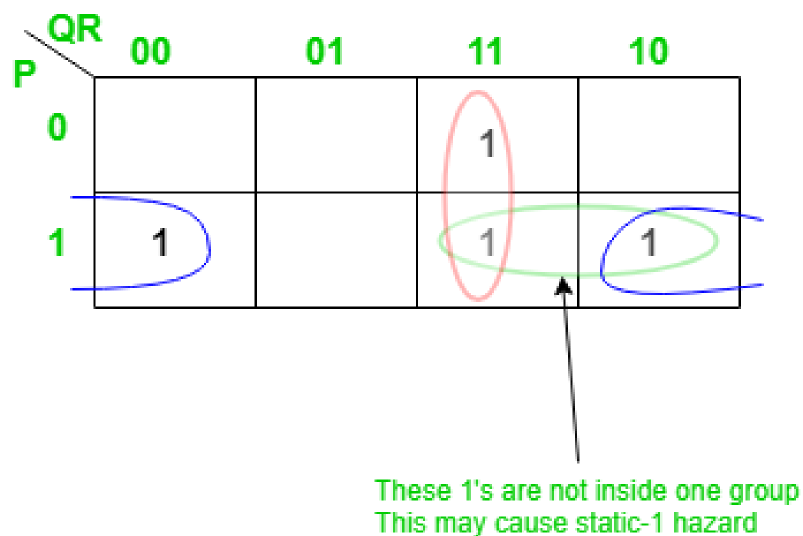


Figure 46. K-Map der Schaltung aus Abb. 45

Das grün umrandete Paar von 1-Werten ist nicht Teil der gleichen Gruppierung / Paarung. Dies führt zu einer statischen Gefahr in diesem Schaltkreis.

Beseitigung von statischen Hazards

- Sobald eine statische 1-Gefahr erkannt wurde, kann sie leicht beseitigt werden, indem der Funktion (Schaltung) weitere Terme (Logikgatter) hinzugefügt werden.
- Der gebräuchlichste Ansatz ist, die fehlende Gruppe in die vorhandene boolesche Funktion einzufügen, da das Hinzufügen dieses Terms die Funktion nicht beeinflusst, die Instabilität jedoch beseitigt.

➤ Da im obigen Beispiel das grün umrandete Paar von 1-Werten die statische-1-Gefahr verursacht, fügen wir dies wie folgt als Hauptimplikant für die vorhandene Funktion hinzu:

$$f(P,Q,R) = Q \bullet R + P \bullet \neg R$$
$$\Rightarrow$$
$$f(P,Q,R)' = Q \bullet R + P \bullet \neg R + P \bullet Q$$

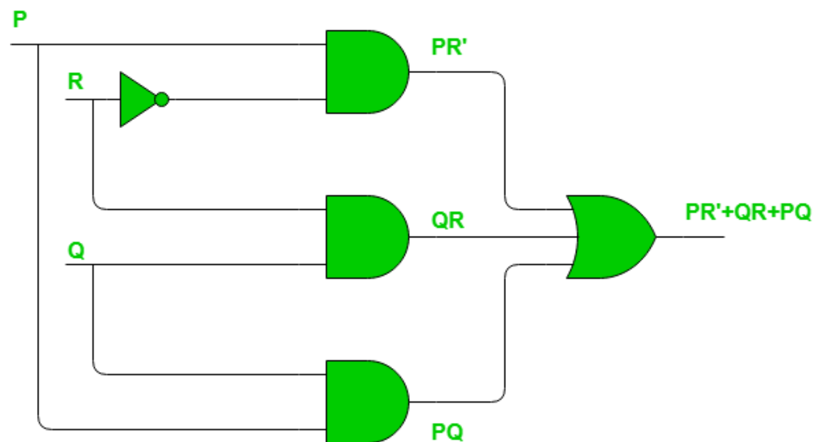


Figure 47. Hazardfreie Variante der Schaltung aus Abb. 45

➤ In ähnlicher Weise müssen für statische-0-Gefahren 0-Werte anstelle von 1-Nummern berücksichtigt werden.

➤ Wenn benachbarte 0-Werte in der K-map nicht in derselben Gruppe zusammengefasst sind, so können diese eine statische-0-Gefahr verursachen.

➤ Die Methode zum Erkennen und Beseitigen der statischen 0-Gefahr ist völlig analog wie die für die statische 1-Gefahr, außer dass anstelle von SOP POS verwendet wird, da es sich in diesem Fall um 0-Werte handelt.

Aufgabe

1. Implementiere die Schaltungen aus Abb. 45 in Retro und weise die Logikhazards durch Testen nach
2. Wo verläuft der längste kombinatorische Pfad in der Schaltung?
3. Implementiere die modifizierte Schaltung aus Abb. 47 in Retro und weise Hazardfreiheit durch Testen exemplarisch nach.

6. Kombinatorische Logik

6.1. ZIELE

- Verständnis vom Aufbau und Funktionsweise von kombinatorischen Logiksystemen als Grundelemente des Datenpfades von RT Systemen
- Verständnis vom Entwurf von arithmetischen Funktionsblöcken mit Basiszellen oder Basisblöcken
- Aufbau und Funktionsweise von Datenpfadselektoren (für RTL)
- Verwendung von Datenpfadselektoren als Programmierbare Digitallogik!

6.2. Kombinatorische Logik

- Kombinatorische Logik besteht nur aus den Logikfunktionen und Gattern:
    - Disjunktion (Oder)
    - Konjunktion (Und)
    - Negation (Inverter)
- Das Verhalten von Kombinatorischer Logik lässt sich vollständig mit Boolescher Algebra beschreiben → Zeit- und zustandsloses Modell

Technische kombinatorische Logik hat aber ein Zeitmodell durch Signallaufzeiten und Verzögerungen!

- Werte aus der Vergangenheit bestimmen den aktuellen Wert der Ausgangsvariable(n) nicht. Im folgenden werden fundamentale Beispiele für Schaltnetze gezeigt.
- D.h., ein Schaltnetz oder kombinatorische Logik ist eine logische Schaltung, deren Ausgangsvariable(n) nur von den am Eingang anliegenden Werten, den Eingangsvariablen, abhängt.

6.3. Signallaufzeit

Die Zeit, die ein Signal vom Eingang eines Logikgatters oder einer damit aufgebauten Logikschaltung bis zum Ausgang 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.

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

➤ Eine Ausgangsstufe eines Logikgatters muss die effektive par. Kapazität umladen. Je mehr Logikgattereingänge auf einen Ausgang geschaltet sind, desto größer die belastende Kapazität, und umso größer die Verzögerungszeit (FANIN/FANOUT).

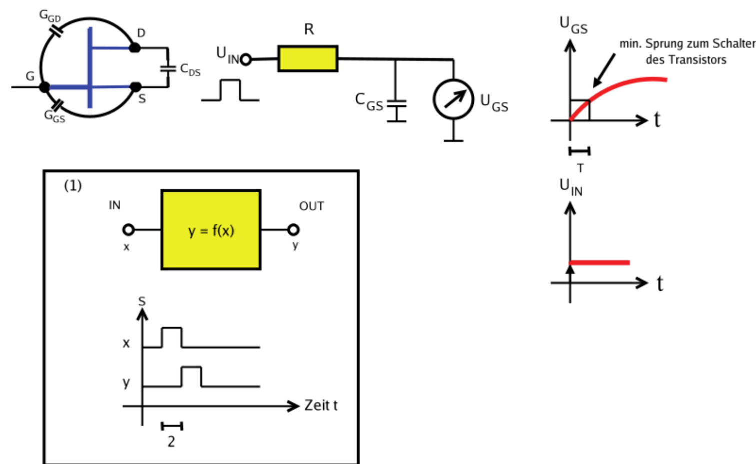


Figure 48. (Links) Parasitäre Kapazitäten und Signallaufzeit (Rechts) Ursache der Signalverzögerung durch Signalverhalten eines äquivalenten RC-Gliedes (rechts).

Längster Kombinatorischer Pfad (LKP)

➤ Es findet eine Akkumulation der einzelnen Signallaufzeiten entlang eines Signalpfades statt:

➤ Die gesamte Laufzeit in einem Pfad ist dann:

$$T_P = \sum_{i=1}^{N(P)} \Delta T_i$$
$$P = e_1, e_2, ..$$

➤ Während dieser Zeit ist die kombinatorische Logikschaltung metastabil,

d.h. die einzelnen Ausgänge können sich zeitlich mehrfach ändern (Hazards).

6.4. Komponenten und Schaltungen

- Die Kombinatorische Logik bildet den RT **Datenpfad**
- Wichtige Komponenten sind u.A.:

  1. Addierer
  2. Multiplizierer
  3. (Dividierer)
  4. Logische Bitoperatoren (trivial)
  5. Relationale Operationen
  6. Multiplexer, Demultipelxer

6.5. Addierer

Halbaddierer

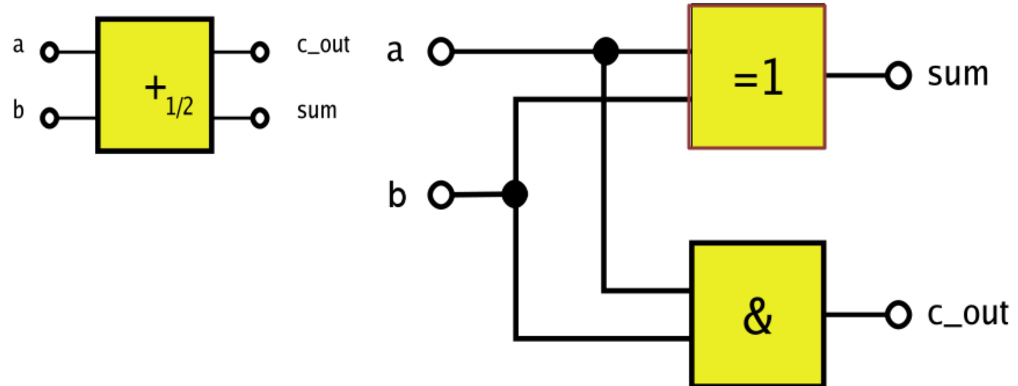
Ein Halbaddierer besitzt zwei Eingangsvariablen a und b, und zwei Ausgangsvariablen, die Summe und der Übertrag Carry, mit folgender Funktionstabelle:

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

Boolesche Gleichung des Halbaddierers

$$sum = \neg a \bullet b + a \bullet \neg b = a \oplus b$$
$$c_{out} = a \bullet b$$

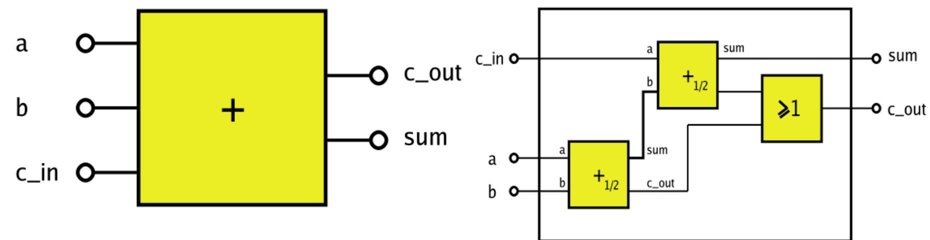
Logikschaltung des Halbaddierers



Volladdierer

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

Logikschaltung des Volladdierers



Boolesche Gleichung des Volladdierers

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

$$
\begin{aligned}
sum &= \neg a \bullet \neg b \bullet c_{in} + \neg a \bullet b \bullet \neg c_{in} + a \bullet \neg b \bullet \neg c_{in} + a \bullet b \bullet c_{in} \\
&= a \oplus b \oplus c \\
c_{out} &= \neg a \bullet b \bullet c_{in} + a \bullet \neg b \bullet c_{in} + a \bullet b \bullet \neg c_{in} + a \bullet b \bullet c_{in}
\end{aligned}
$$

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 Übertragsignal-Propagierung vom niederwertigsten zum höchstwertigen Bit statt, d.h. die Berechnung des N-ten Bits erfordert die Ergebnisse der vorherigen N-1 Addierer.

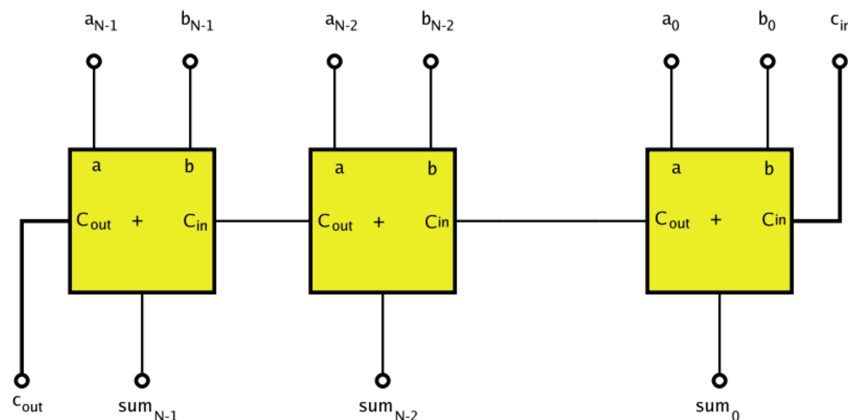


Figure 49. Logikschaltung eines Ripple-Carray Addierers

Aufgabe

1. Implementiere einen 3-Bit Ripple-Carray Addierer in Retro mit elementaren Gattern (Und, Oder, XOR, Not)
2. Teste die Schaltung mit einigen exemplarischen Eingangswerten. Wo befindet sich der längste kombinatorische Pfad und wie groß ist die Verzögerungszeit?
3. Gibt es Hazards?

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

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.

Boolesche Gleichung des Carry-Lookahead Addierers

$$
\begin{aligned}
s_i &= a_i \oplus b_i \oplus c_i \\
c_{i+1} &= a_i \bullet b_i + a_i \bullet c_i + b_i \bullet c_i \\
&= a_i \bullet b_i + (a_i + b_i) \bullet c_i \\
c_{i+1} &= g_i + p_i \bullet ci
\end{aligned}
$$

➤ Dabei ist g der so genannte generierende Term, und p der sog. Durchlaufterm.

➤ Für einen N-Bit-Addierer kann man dann ableiten:

$$
\begin{aligned}
c_1 &= g_0 + p_0 \bullet c_{in} \\
c_2 &= g_1 + p_1 \bullet c_1 \\
&= g_1 + g_0 \bullet p_1 + p_0 \bullet p_1 \bullet c_{in} \\
c_3 &= g_2 + p_2 \bullet c_2 \\
&= g_2 + g_1 \bullet p_2 + g_0 \bullet p_1 \bullet p_2 + p_0 \bullet p_1 \bullet p_2 \bullet c_{in} \\
&\ldots \\
c_{i+1} &= \sum_{j=0}^{i} (g_j \prod_{k=j+1}^{i} P_k) + \prod_{k=0}^{i} P_k \bullet c_{in}
\end{aligned}
$$

➤ 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:
   $f : (P, G) \rightarrow C$

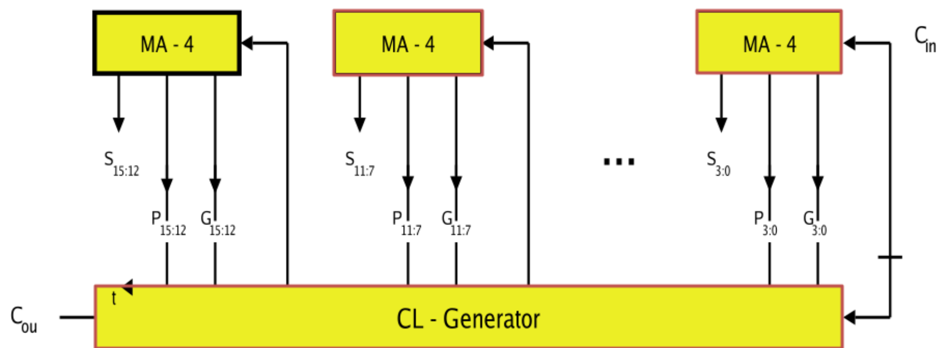


Figure 50. Hybrider Aufbau aus kleinen Ripple-Carry Addierern und Carry-Lookahead Generatoren

6.6. 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

➤ Ein Multiplizierer für die Multiplikation von N-Bit $\times$ M-Bit breiten

Zahlen A und B muss zunächst mathematisch behandelt und hergeleitet werden

❏ Ausgangspunkt ist die gewichtetete Binärsummendarstellung von dezimalen Zahlen

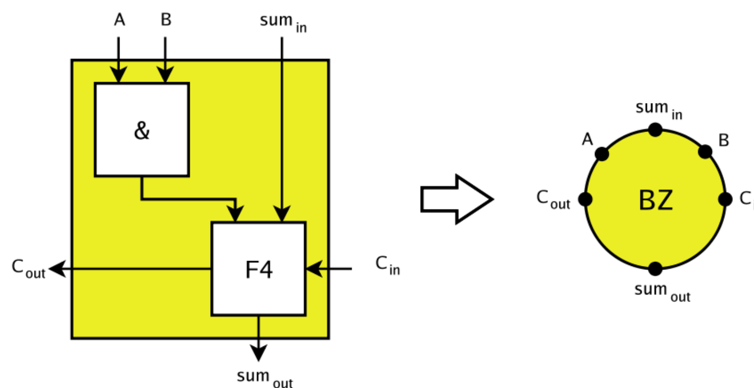
$$A = \sum_{i=0}^{N-1} a_i 2^i, B = \sum_{i=0}^{M-1} b_i 2^i$$
$$A \times B = \sum_{i=0}^{N-1} a_i 2^i \times \sum_{j=0}^{M-1} b_j 2^j = \sum_i \sum_j a_i b_j 2^{i+j}$$

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

$$A \times B = \sum_{k=0}^{M+N-1} P_k 2^k$$
$$P_0 = a_0 \bullet b_0, P_1 = a_1 \bullet b_0 \otimes a_0 \bullet b_1$$
$$P_2 = a_2 \bullet b_0 \otimes a_1 \bullet b_1 \otimes a_0 \bullet b_2, \text{ usw.}$$

Basiszelle eines Multiplizierers

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

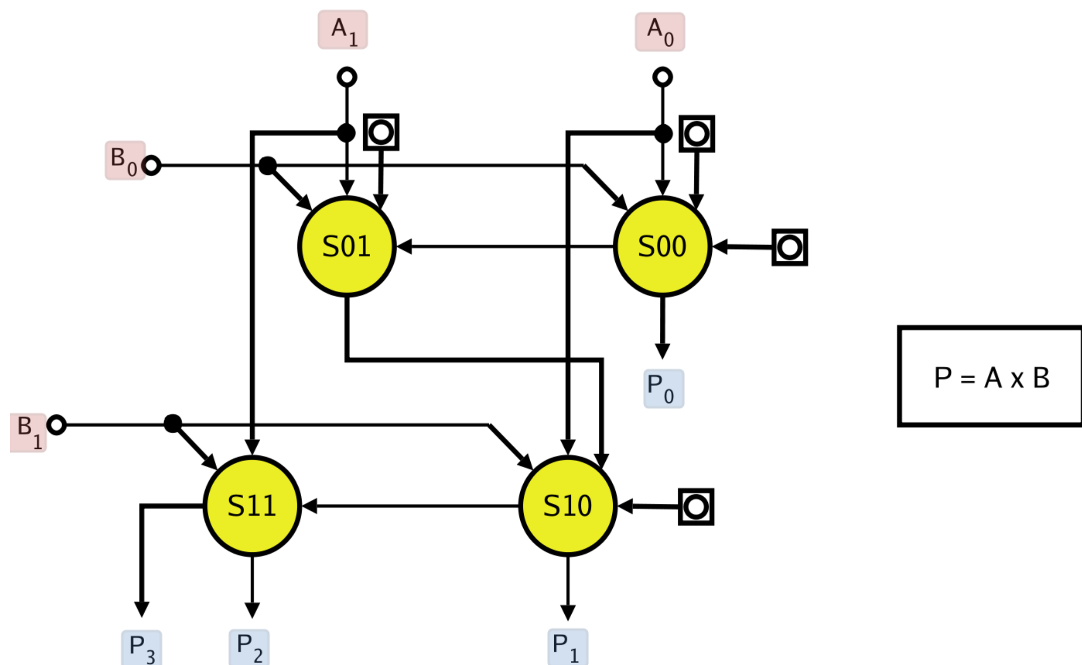


➤ Ein Parallelmultiplizierer setzt sich daher aus Addition und 1-Bit-Multiplizierern zusammen. Ein 4×4 Multiplizierer benötigt dann:

- ❏ 16 Und-Gatter
- ❏ 8 Volladdierer (40 Gatter)
- ❏ 4 Halbaddierer (16 Gatter)

Matrix-Struktur eines Multiplizierers

➤ 2 x 2 Matrix unter Verwendung der Basiszelle



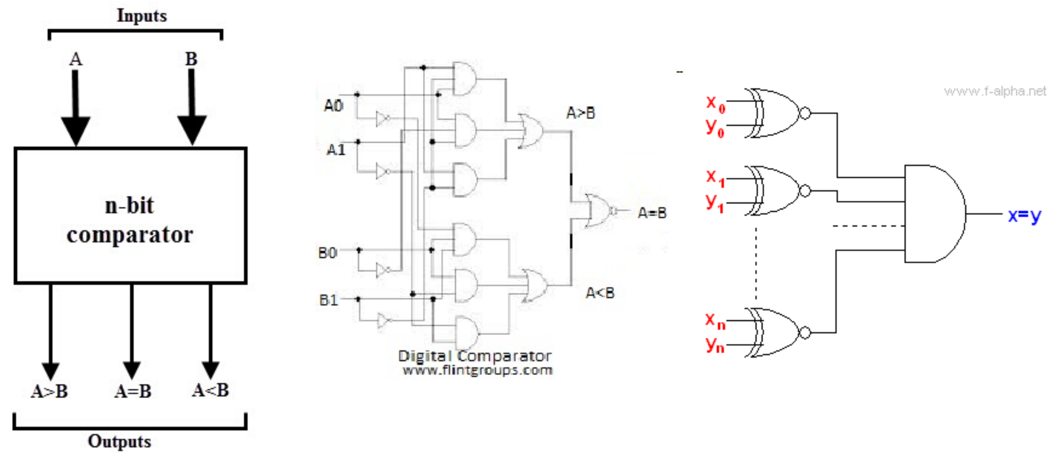
6.7. Logische und Relationale Operationen

➤ N-stellige Bitoperation (Konjunktion, Disjunktion, Negation) lassen sich durch 1-Bit Operationen direkt umsetzen:

$$Y = f(X) = (x_0 \bullet y_0, x_1 \bullet y_1, .., x_n \bullet y_n)$$

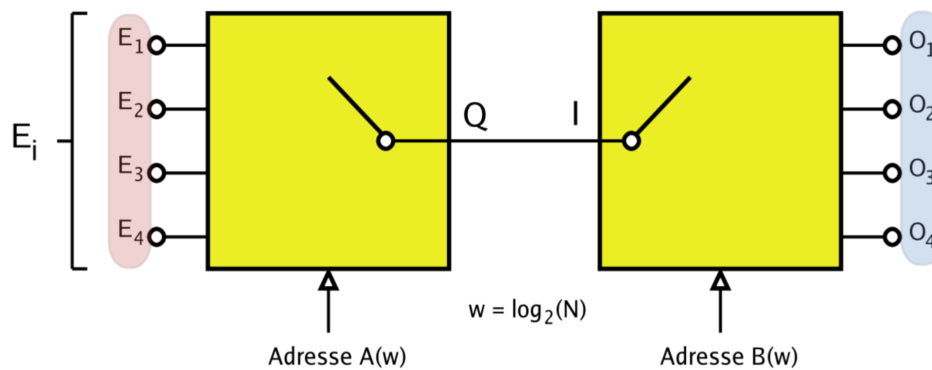
➤ Relationale Operationen:

- ❏ Gleichheit, Kleiner, Größer, usw.



6.8. Multiplexer und Demultiplexer

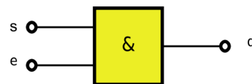
- Multiplexer sind elektronisch gesteuerte Auswahlschalter.
- Ein Multiplexer legt eines von N Eingangssignalen auf eine Ausgangsleitung. Die Auswahl erfolgt durch eine anliegende Adresse → Datenselektor
- Umgekehrten Vorgang mit Demultiplexer, der ein Eingangssignal auf N Ausgänge verteilt. Die Auswahl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



Boolesche Funktion eines 1-Bit Multiplexer

➤ Ein 1-Bit-Multiplexer besteht aus einer Und-Verknüpfung mit einem Eingangssignal e und einem Selektorsignal a:

$$f(e, s) = e \bullet s$$



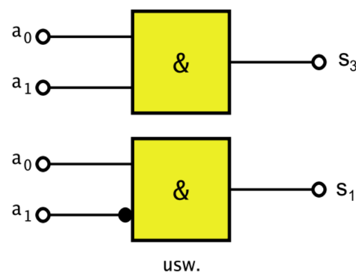
Boolesche Funktion eines N-Bit Multiplexers

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

$$f(e_0, e_1, .., e_i, s_0, s_1, .., s_i) = e_0 \bullet s_0 + e_1 \bullet s_1 + .. + e_i \bullet s_i$$

Adressdekoder

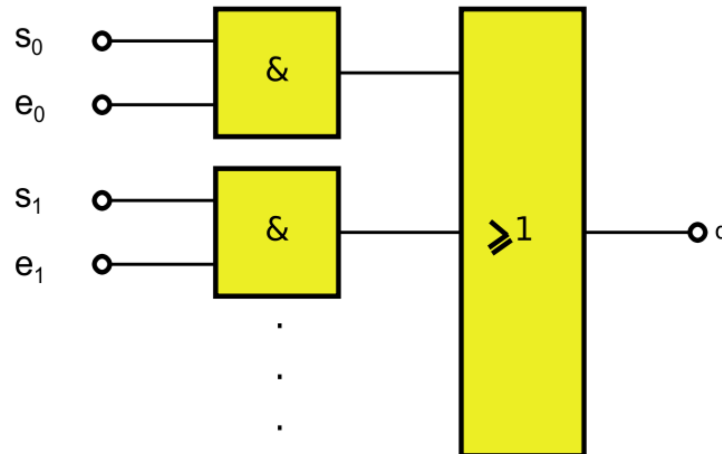
➤ Die einzelnen Selektorsignale ai müssen noch aus den eigentlichen Adresssignalen A dekodiert werden. Dazu wird ein Demultiplxer mit I=1 verwendet. Beispiel für N=2:



Vollständiger BF des Multiplexers

$$f(E, A) = e_0 \bullet \neg a_0 \bullet \neg a_1 + e_1 \bullet a_0 \bullet \neg a_1 +$$
$$e_2 \bullet \neg a_0 \bullet a_1 + e_3 \bullet a_0 \bullet a_1$$

➤ 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.

➤ Mit Multiplexern lassen sich beliebige logische Funktionen realisieren:

Eingangsvariablen der BF
Abbildung auf Adresssignale a_i des Multiplexers

Ausgangsvariable der BF
Ausgangssignal Q des Multiplexers

Logikwerte
Gegeben durch Eingangssignale E

$$f(A) = S(A, E)$$

$$S \rightarrow \begin{cases} e_i, \text{wenn } a_i = 1 \\ x = 0/1, \text{sonst} \end{cases}$$

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

Figure 51. Beispiel eines Multiplexers als Implementierung einer booleschen Funktion. Die *E*-Matrix wird zeilenweise Oder-verknüpft

7. Sequenzielle Logikssysteme

ZIELE

➤ Verständnis der Funktionsweise von zustandsbasierten sequenziellen Logiksystemen

➤ Verständnis und Anwendung der Grundelemente von sequenzieller Digitallogik (Register, FLIP-FLOPs)

➤ Unterscheidung von Daten-, Takt-, und Taktflanken gesteuerten Speichern

7.1. Sequenzielle Logik

➤ Sequenzielle Logik besteht aus:

❏ **Kombinatorischer Logik**

❏ **Speicherelementen**

➤ Die Ausgangssignale solcher Logik hängen von den aktuellen Eingangangssignalen *E* (Eingangszuständen) und zusätzlich von Signalen (Zuständen) aus der Vergangenheit ab.

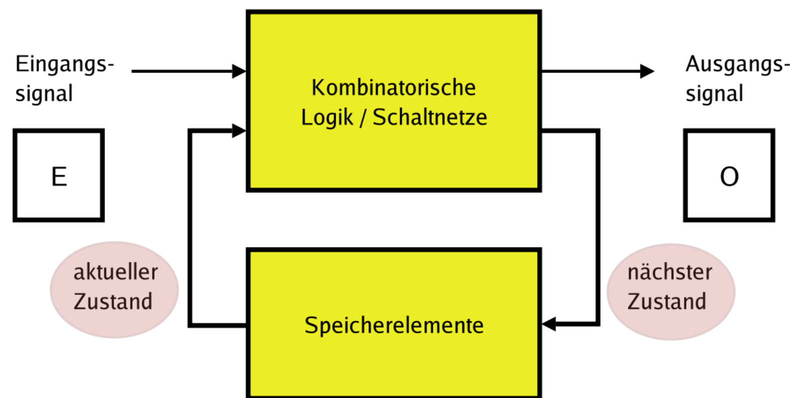


Figure 52. Allgemeine Struktur eines sequenziellen Logiksystems.

- 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:

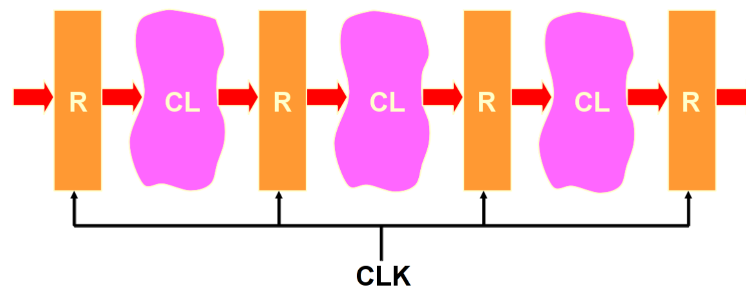
$$Z_{n+1} = f(Z_n, E_{n+1})$$
$$Z_n = f(Z_{n-1}, E_n), O_n = g(Z_n)$$

- Man unterscheidet ***synchrone*** und ***asynchrone*** sequenzielle Systeme.
  - Synchrone Systeme ändern ihren Zustand nur zu bestimmten Zeitpunkten. D.h. synchrone Systeme sind taktgesteuert.
  - Asynchrone können ihren Zustand jederzeit (zeitkontinuierlich) ändern.
- Der Zustandsspeicher eines asynchronen Systems kann mit selbst steuernden Latches, der eines synchronen Systems mit taktgesteuerten FLIP-FLOPs implementiert werden.
- Der Entwurf synchroner Schaltungen ist deutlich einfacher als bei asynchronen. Bei synchronen Systemen übernehmen die Speicherelemente nur zu bestimmten Zeitpunkten oder Intervallen die Daten.

7.2. Synchrone Logiksysteme

- Ein Algorithmus ist eine Sequenz von Berechnungsschritten

- Sequenzielle Datenverarbeitung wird in synchronen Systemen mit einer globalen Zeitreferenz - dem Takt (Clock) - gesteuert.
- Der längste kombinatorische Pfad (die Verzögerung) zwischen zwei Registern bestimmt kürzeste Taktzykluszeit (1/Taktfrequenz)
- Synchrone RT Systeme bestehen aus einer Folge von Registerebenen verbunden durch kombinatorische Logikblöcke



[Jordi Cortadella et al., 2002]

Figure 53. Allgemeine Architektur von synchronen RT Sytstemen

7.3. Asynchrone Logiksysteme

- Asynchrone Logik arbeitet selbst gesteuert ohne globale Zeitreferenz (self-timed, speed-independent, delay-insensitive)
- Längste Laufzeit eines kombinatorischen Digitallogikteils zwischen zwei Registern ist nicht mehr bestimmend für das gesamte System!
- Asynchrones System: Module die über ein Handshake Protokoll (Anfrage REQ - Bestätigung der Verarbeitung ACK) kommunizieren

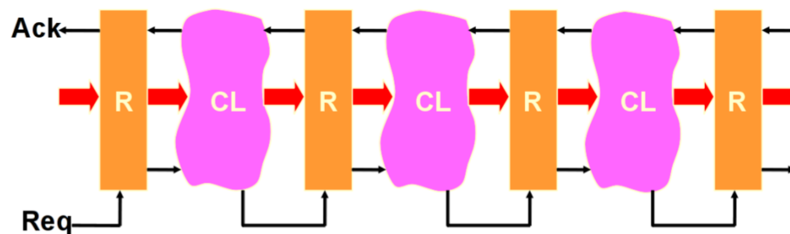


Figure 54. Allgemeine Architektur von asychronen RT Systemen

- Partitionierung eines Datenverarbeitungssystems und der Digitallogik in kleinere synchron arbeitende Module.

➤ Die einzelnen Module besitzen einen eigenen lokalen Takt und sind taktgesteuert. Der Datenaustausch findet zwischen den Modulen asynchron statt (mit REQ-ACK Protokoll): GALS (Global Asynchronous, Local Synchronous)

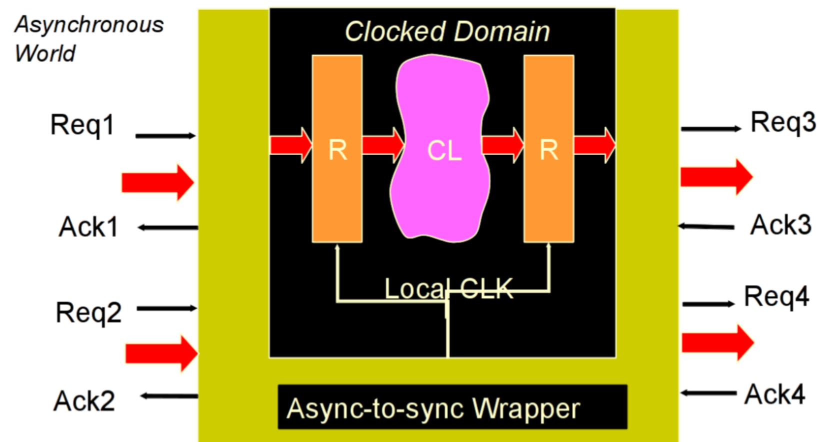


Figure 55. Allgemeiner Architektur und Schnittstelle von GALS (Global Asynchronous, Local Synchronous) Systemen

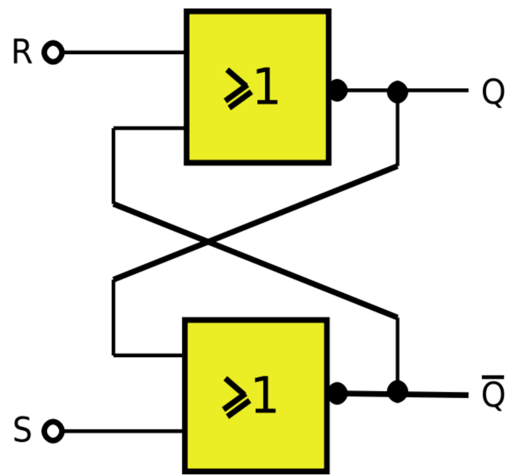
7.4. Speicherelemente sequenzieller Logik

Die Speicherelemente lassen sich in zwei Gruppen unterteilen:

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

Latch-Speicher oder RS-Latch

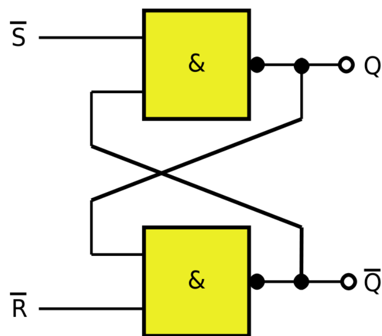
➤ Aufgebaut mit zwei rückgekoppelten invertierenden Logikgattern, z.B. mit NOR-Gattern:



S	R	Q	¬Q
0	0	Q_{n-1}	$\neg Q_{n-1}$
0	1	0	1
1	0	1	0
1	1	0	0

Figure 56. Funktionstabelle des RS-Latches mit Eingängen S: Set und R: Reset und Ausgängen $Q/\neg Q$

➤ RS-Latch alternativ mit NAND-Gattern:



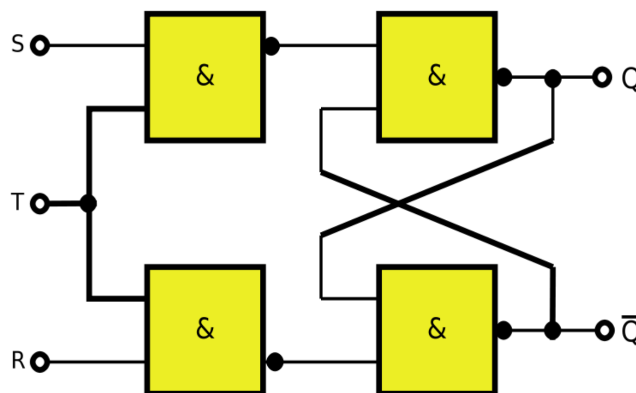
Taktgesteuertes RS-FLIP-FLOP

- 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.

Algorithm 2. (*VHDL Modell eine RS Latches*)

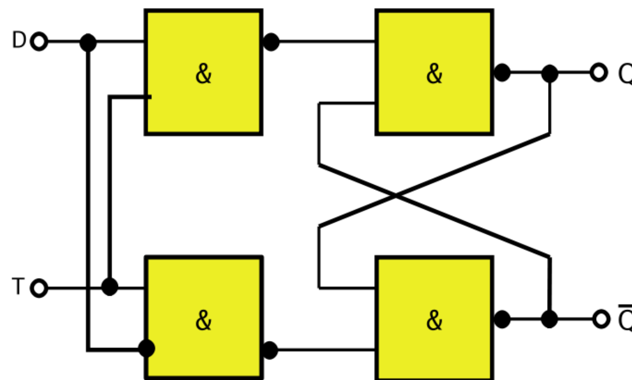
```
entity RSL is
  port { r : in bit, s : in bit, q : out bit, nq : out bit }
architecture main of RSL is
  signal l : bit;
begin
  process rsl(r, s)
  begin
    if r = '1' then
      l <= '0';
    elsif s = '1' then
      l <= '1';
    end if;
  end process;
  q <= l;
  nq <= not l;
end RSL;
```

- T-RS-FLIP-FLOP:



Taktgesteuertes D-FLIP-FLOP

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



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!

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

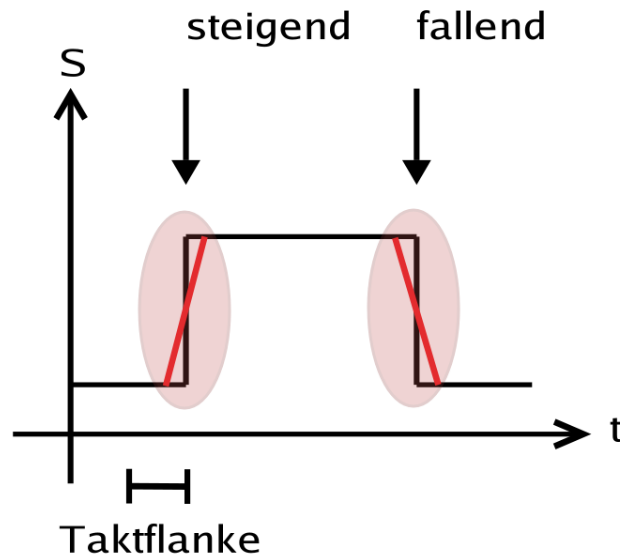
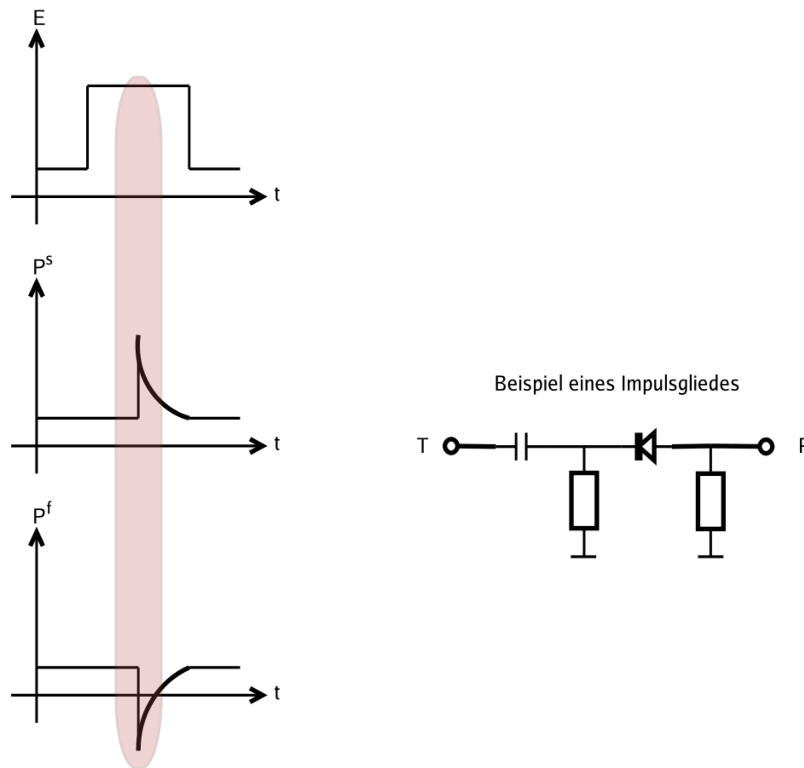


Figure 57. Steigende und fallende Taktflanke als Auslöser für Speicheraktivität (ΔT(Flanke) $\rightarrow \infty$)

- Übertragungsfunktion von Impulsgliedern, aufgebaut mit einem RC-Schaltnetzwerk und einer Diode.
- Das Signal P dient als neuer Aktivierungseingang des RS- oder D-FLIP-FLOP's.



Taktflanken gesteuertes D- und JK-FLIP-FLOP (Register)

➤ D- und JK-FLIP-FLOPs, die häufig in sequenzieller Logik eingesetzt werden.

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

Figure 58. Funktionstabelle eines D-FLIP-FLOP's

Algorithm 3. (*VHDL Modell eine D-FLIP-FLOPs*)

```
entity DFF is
  generic { N : natural := 8 }
  port { wr : in bit_vector(N − 1 downto 0), rd : out bit_vector(N − 1 downto 0),
         clk : in bit, we : in bit }
architecture main of DFF is
  signal r : bit_vector(N − 1 downto 0);
begin
  process dff(clk, we, wr)
  begin
    if clk'event and clk =' 1' then
     if we = '1' then
       r <= wr;
     end if
    end if;
  end process;
  rd <= r;
end DFF;
```

➤ 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	$\neg Q_n$

Figure 59. Funktionstabelle eines JK-FLIP-FLOPs

7.5. 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.

Man unterscheidet:

Asynchronzähler
: 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.

Synchronzähler

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.

Beispiel eines 3-Bit Synchronzählers mit D-FLIP-FLOPs.

Example 3. (*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;
```

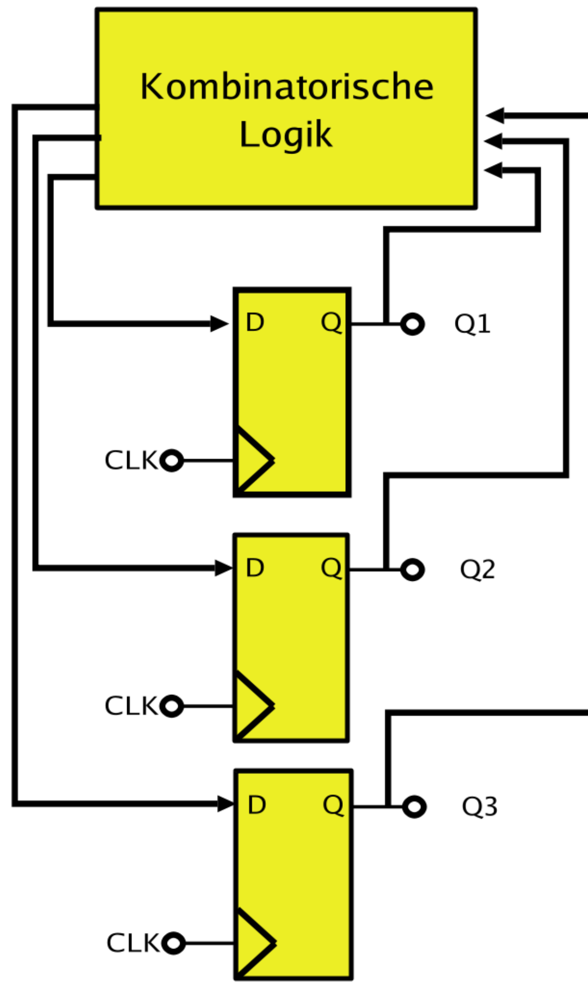


Figure 60. Architektur eines 3-Bit Synchronzählers

Beispiel eines 3-Bit Asynchronzählers mit rückgekoppelten D-FLIP-FLOPs.

Example 4. (*VHDL-Beschreibung eines Asynchronzählers: Nicht implementierbar in FPGAs!*)

```
entity asyn3count is
  port ( clk: in bit;
         q: out bit_vector(2 downto 0);
end asyn3count;
architecture main of syn3count is
  signal q_int: bit_vector(2 downto 0);
begin
  process (clk)
  begin
    if clk'event and clk=0 then q_int(0) <= not q_int(0); end if;
    if q_int(0)'event and q_int(0)=0 then q_int(1) <= not q_int(1); end if;
    if q_int(1)'event and q_int(1)=0 then q_int(2) <= not q_int(2); end if;
  end process;
  q <= q_int;
end main;
```

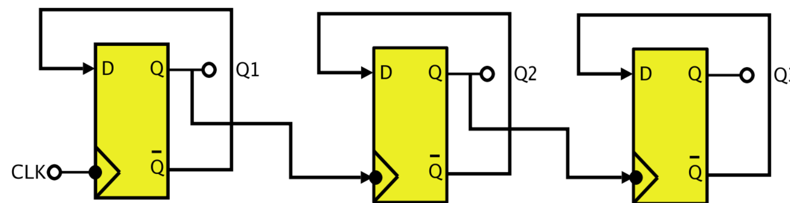


Figure 61. Architektur eines 3-Bit Asynchronzählers

- ➤ In dieser Architektur findet eine Vermischung von Daten- und Taktsignalen statt!
  - ❏ Sollte vermieden werden!
- ➤ Wie beim Ripple-Carry Addierer sidn Laufzeitverzögerungen eines Asynchrinzählers akkumulativ!
- ➤ Das höchste Bit hat die längste Verzögerung

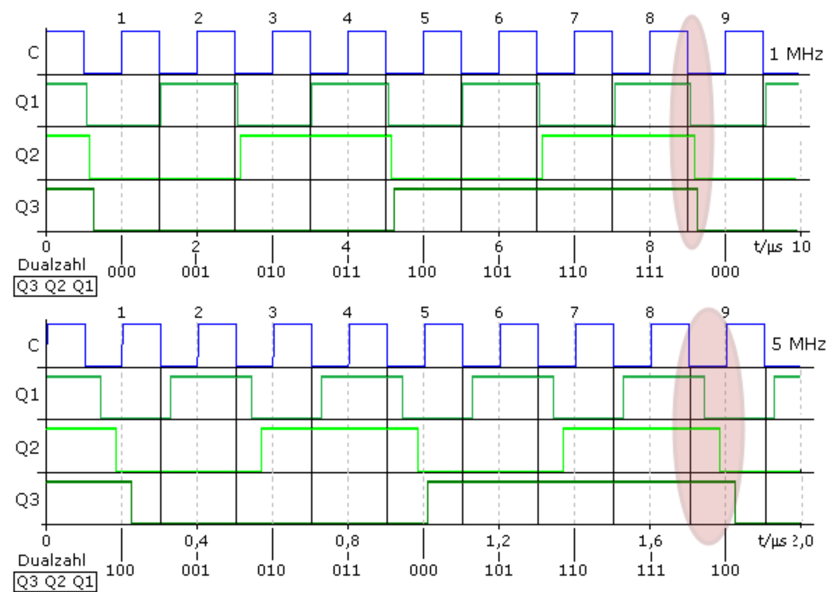


Figure 62. Signaldiagramm eines Asynchronzählers und Signalverzögerungen

8. Programmierbare Logikbausteine

ZIELE

➤ Verständnis der Grundlagen und Architektur von Programmierbaren Digitallogikbausteinen (PDL)

➤ Verständnis der Abbildung von booleschen Funktionen auf einfache PDL Architekturen

➤ Unterscheidung verschiedener Technologien: ROM, PLD, CPLD, FPGA und ASIC

8.1. 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 Terminus, da ein Programm eine Ablaufvorschrift darstellt; besser "anwendungsspezifisch konfigurierbar".

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

$$Q_i = P_{1,i} + P_{2,i} + .. + P_{n,i}$$

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

- Eine disjunktive Normalform besteht aus einer Und-Verknüpfungsebene (-matrix), die die Produktterme $P_{i,j}$ bildet, und eine Oder-Verknüpfungsebene (-matrix), die die einzelnen Produktterme verbindet.
- Ein Adressdekoder (1-aus-N Dekoder) besteht aus einer Und-Verknüpfungsmatrix.
  - Geeignet um Produktterme zu bilden, aber alle Produktterme ergeben Ausgangswert = 1!
  - Daher wird Produktterm-selektive Oder-Matrix benötigt.

Charakteristisch für alle PDL Architekturen und Technologien ist eine logische UND-ODER Matrixstruktur

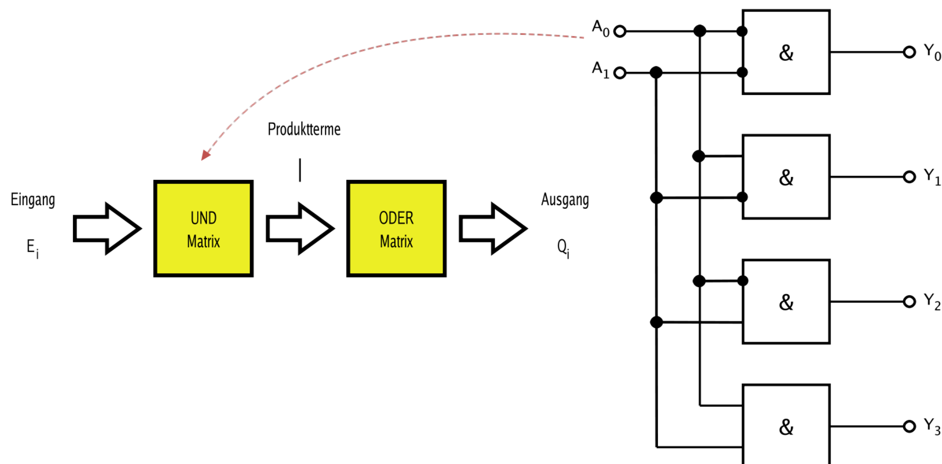
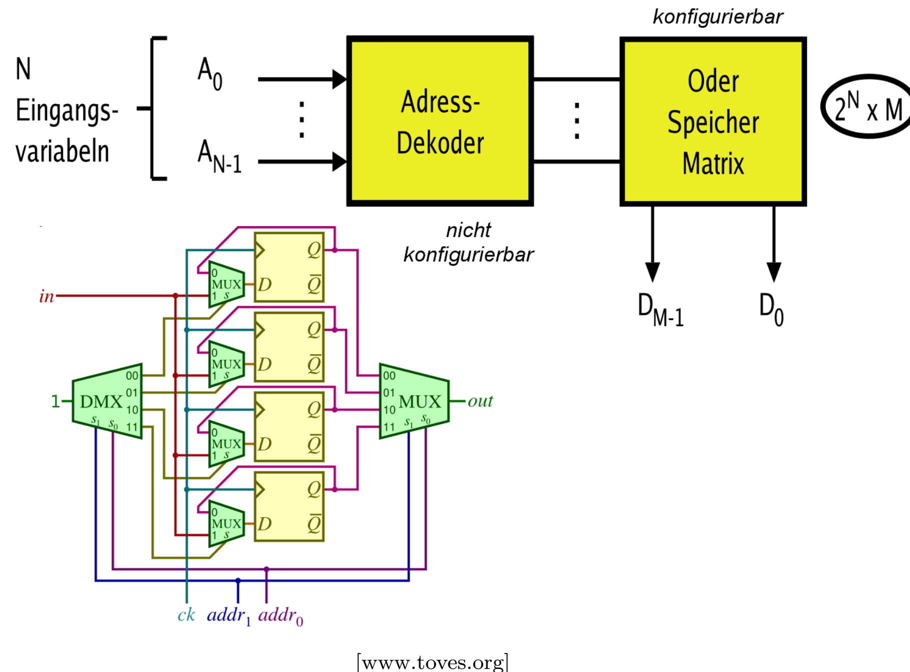


Figure 63. (Links) Technologische Umsetzung der Disjunktiven Normalform (Rechts) 1-aus-4-Dekoder

8.2. Programmierbare Logikbausteine: ROM und RAM

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



[www.toves.org]

Figure 64. Allgemeine ROM/RAM Architektur

Abbildung BF auf ROM/RAM Baustein

➤ Eine beliebige Boolesche Funktion $f(x_1,x_2,..x_n)$: $2^n \rightarrow 2^m$ kann mit einem ROM/RAM Speicher implementiert werden (n Eingangsvariablen).

➤ Die Funktionsvariablen werden den (kodierten) Adresseingängen zugeordnet und es gilt: $a_i = x_i$

➤ Die Zeilen der Funktionstabelle eines kombinatorischen Digitallogiksystems entsprechen den Zellen des Speicherbausteins.

➤ Konfiguration: Die jeweilige Ergebnisspalte der Funktionstabelle wird in den Speicher übertragen.

➤ Die Speicherzellen werden durch die Eingangsvariablen der BF addressiert und das Ergebnis ausgewählt

➤ Es werden 2^n Speicherzellen der Datenwortbreite m Bit benötigt

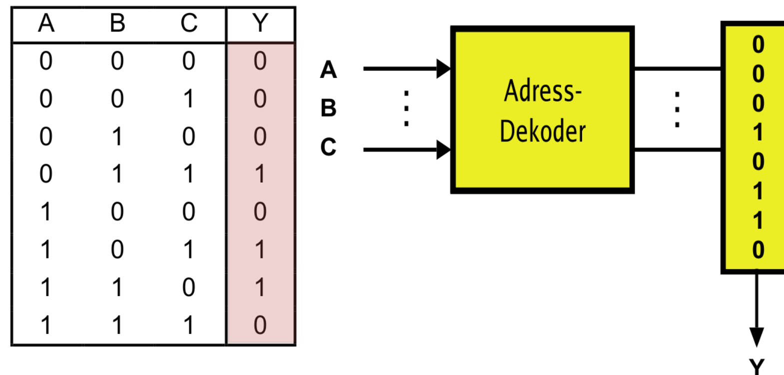


Figure 65. Übertragung der Ausgangsspalte einer Booleschen Funktion auf die Speicherspalte eines RAM/ROM Bausteins

Vorteile bei der Verwendung von ROM/RAM-Logikbausteinen als PDL:

1. Jede beliebige $n \times m$ Funktion läasst sich in einem Speicherbaustein mit 2^n Speicherstellen abbilden
2. Die BF muss (und braucht) nicht vereinfacht werden

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

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

Aufgabe

1. Erstelle die Boolesche Funktion als Funktionstabelle eines 2-Bit Addierers.
   - ➤ Eingänge: x_1,x_2,y_1,y_2
   - ➤ Ausgänge: z_1,z_2,c
2. Implementiere die BF in einem ROM Baustein als PDL in ReTro und teste die Schaltung exemplarisch

8.3. 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 größerer Chip-Fläche
2. und erhöhter Leistungsaufnahme führt.

➤ Spezielle Logikarrays mit programmierbarer UND- sowie ODER-Matrix stellen eine Verbesserung dar.

❑ Durch eine 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:

$$\eta = 1 - \frac{P}{2^N}$$

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

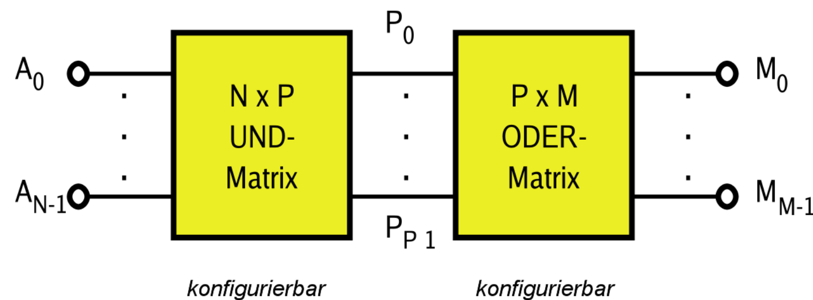


Figure 66. Reduzierte konfigurierbare UND- und ODER-Matrix als PLA.

➤ 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 Chipfläche/Komplexität und Funktionalität gewählt.
- Die UND-ODER-Matrix lässt sich auf eine reine ODER(NOR)-Matrix transformieren, was in technologischer Zweckmäßigkeit begründet sein kann:

$$Y = \neg A \bullet \neg B + \neg C \bullet \neg D$$
$$Y = \neg(A + B) \bullet \neg(C + D)$$

Variante Programmierbare Arraylogik (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.

8.4. 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).

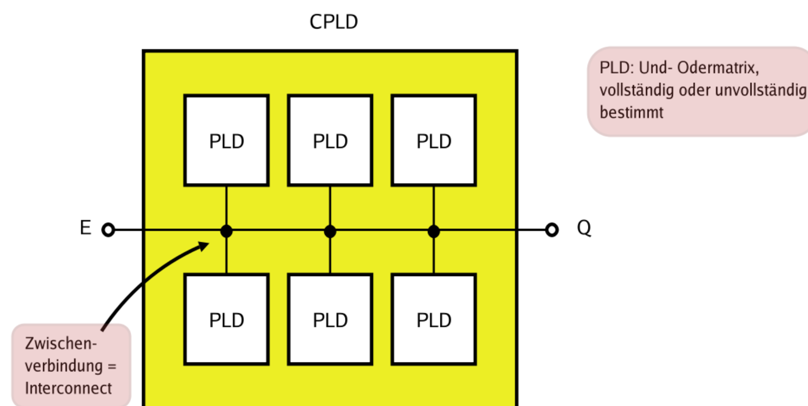
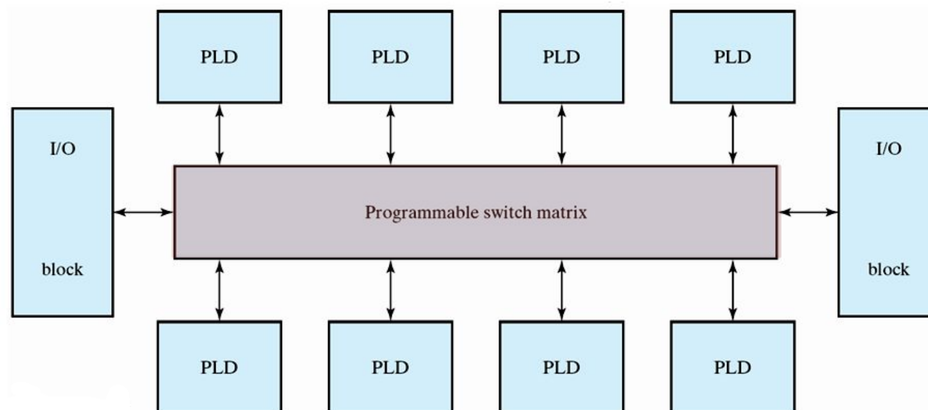


Figure 67. CPLD mit kleineren PLD-Blöcken und Verbindungsstrukturen:

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

Architektur CPLD

1. Konfiguerierbare Logikblöcke (PLD)
2. Konfigurierbare Verbindungungsmatrix (Kreuzmatrixschalter)
3. Konfiguerierbare Ein- und Ausgabeblöcke (IO)



[Hector Allford, slideplayer]

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

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

Feature	E^2CMOS	Flash	SRAM	Antifuse
Reprogrammability	Yes	Yes	Yes	NO
In-System Programmable	Yes	Yes	Yes (Volatile)	NO
Program Time	Fast	Med.	Fast	Slow
Erase Time Fast	Slow	Fast	N/A (OTP)	
Testability	Full	Full	Full	Limited
External Hardware	No	No	EPROM	Pgmr

[Lattice Sem.]

Figure 68. Verschiedene Konfigurationtechnologien im Vergleich und ihre Eigenschaften

IO Blöcke

- 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.

➤ Ausgang mit Puffer (BUFFER, auch mit D-FLIP-FLOP)

Zusammenfassung

Eigenschaften von CPLDs:

1. CPLDs bestehen aus einer Matrixs 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 ein gut vorhersagbares zeitliches Verhalten (LKP, begründet in einem verhältnismäßig einfachen und regulären Aufbau und dem Kreutmatrixschalter (Netzwerktopologie mit konstanter Ausdehnung 1/2).
4. Die Verbindungsmatrix besitzt eine Kreuzstruktur mit eingeschränkten Trassierungsmöglichkeiten.

8.5. Xilinx XC9500 CPLD

Architektur

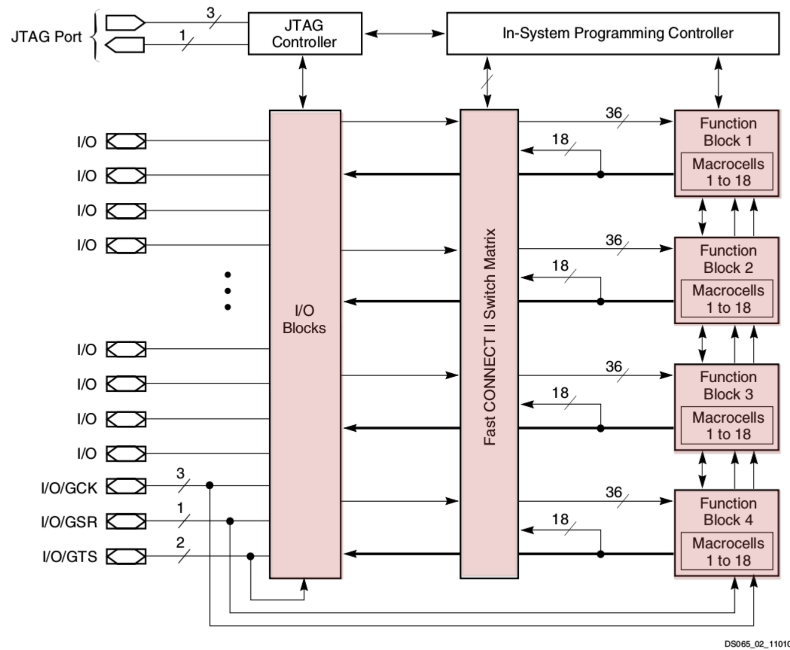


Figure 69. XC9572 Architecture: Function block outputs (indicated by the bold line) drive the I/O blocks directly. [Xilinx]

Architektur Funktionsblock

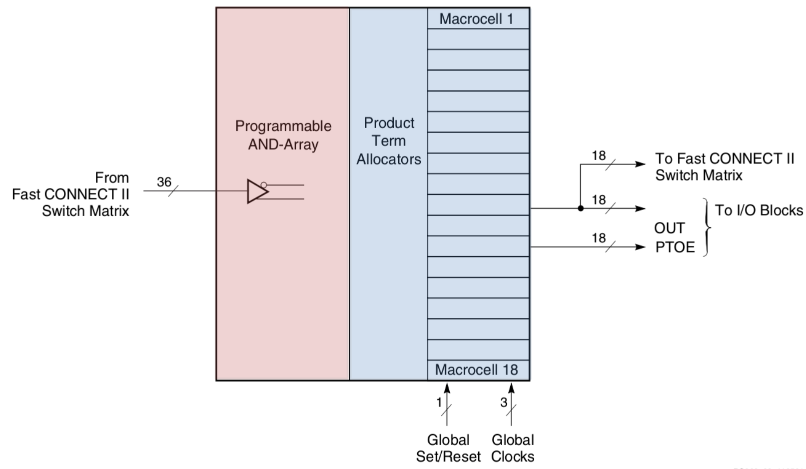


Figure 70. XC9500 Function Block [Xilinx]

Architektur Makrozelle

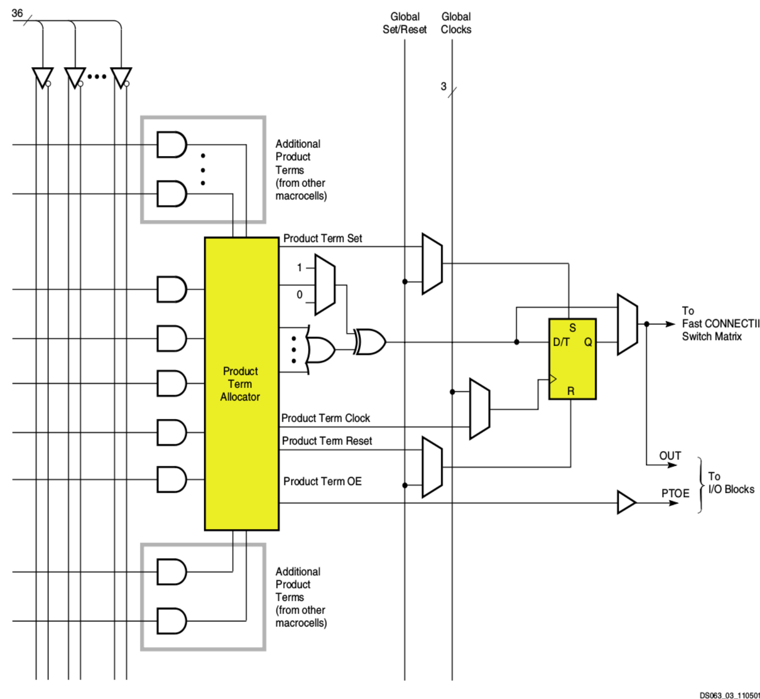


Figure 71. XC9500 Macrocell Within Function Block [Xilinx]

8.6. Feldgatterarrays (FPGA)

➤ 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.

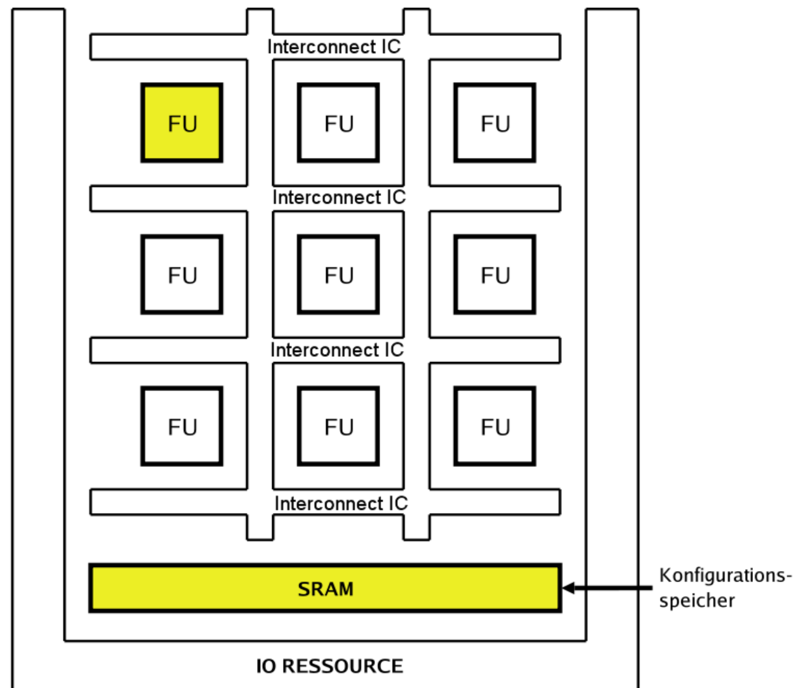


Figure 72. Aufbau eines FPGA mit Funktionseinheiten FU, Verbindungskanälen IC, und IO Blöcken.

➤ 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, LVTTL,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 Verbindungstrassierung. 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 gegenüber EEPROM/Flash:

* 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.

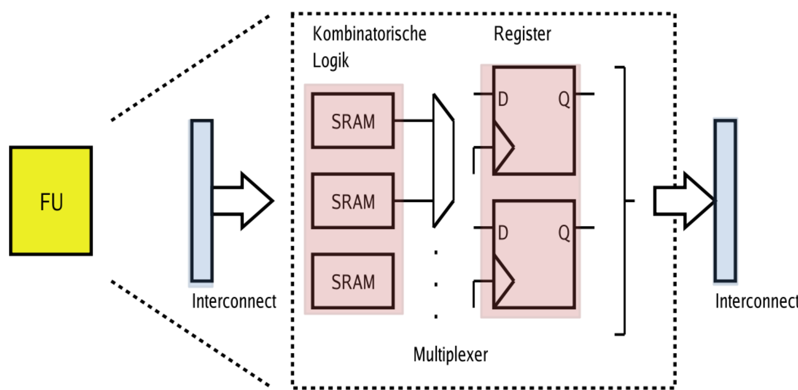


Figure 73. Funktionseinheit eines FPGAs: Kleine SRAM-Blöcke zur Implementierung beliebiger logischer Funktionen.

8.7. Metriken

Gatteräquivalent

➤ Um FPGAs und Implementierung von Digitalsystemen 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 "um-

rechnen", 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.

➤ Bei ASICs ist der Äquivalentwert ein Maß für die Chip-Fläche.

➤ Diese Werte sind aber 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 ⇔ geringer Flächenbedarf;
2. Flexible Konfigurierbarkeit ⇔ kleine Signallaufzeiten.

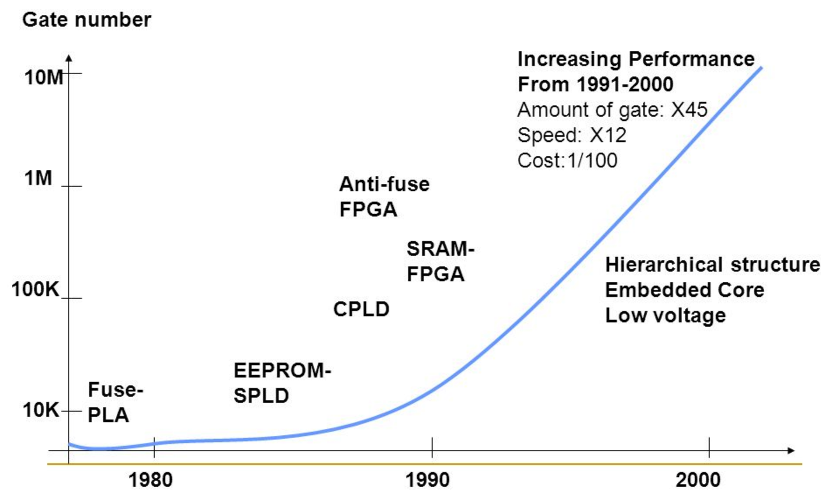


Figure 74. Historische Entwicklung der PLD Technologien und die äquivalente Gatterzahl [Gillian White, slideplayer]

➤ Um einen Eindruck zu vermitteln, mit welchen Logikgatterbedarf man bei bestimmter Funktionalität zu rechnen hat, ist in folgender Tabelle die Gatterzahl von ASICs aus der Sun Microsystems SPARC-Station 1 gezeigt (∼ 1985).

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
	102k

8.8. Xilinx FPGA

- Fein granulierte Funktionsblöcke CLB, RAM Blöcke und IO Blöcke
- Konfigurierbarer und universell einsetzbarer Funktionsblock

Architektur

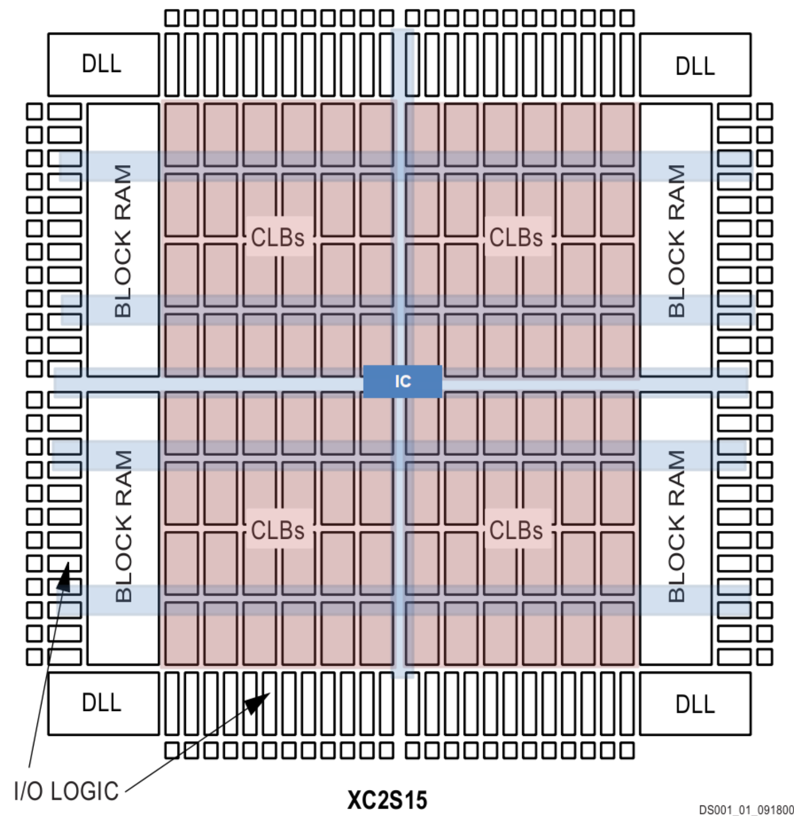


Figure 75. Basic Spartan-II Family FPGA Block Diagram [Xilinx]

IO und Funktionsblockarchitektur

➤ Konfigurierbarer und universell einsetzbarer I/O-Block

➤ Konfigurierbarer und universell einsetzbarer Funktionsblock

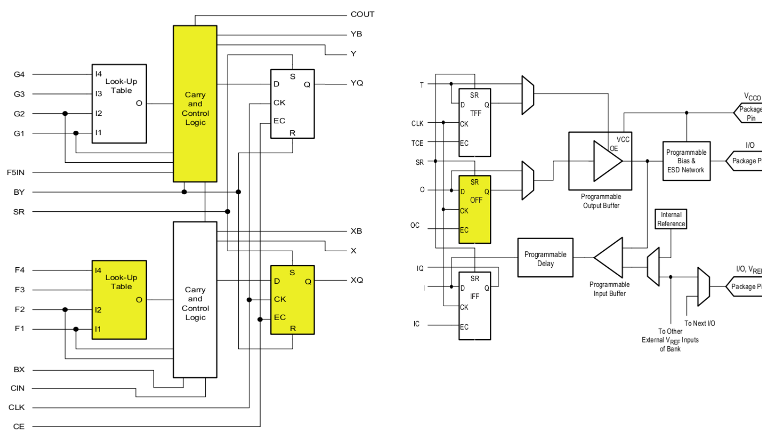


Figure 76. (Links) Spartan-II CLB Slice (two identical slices in each CLB) (Rechts) Spartan-II FPGA Input/Output Block (IOB) [Xilinx]

8.9. 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 anwendungsspezifische 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.

- ➤ Layouts von ASICs sind trotz der Gestaltungsfreiheit wie CPLDs und FPGAs strukturiert, z.B. können Verbindungstrukturen eine Zeilen- oder Spaltenstruktur aufweisen.
- ➤ Man unterscheidet bei der Trassierung von Trassierungskanälen, die zwischen Funktionseinheiten gelegt werden, und solchen, die auch über Funktionseinheiten gelegt werden (Multilayer).

Architekturen

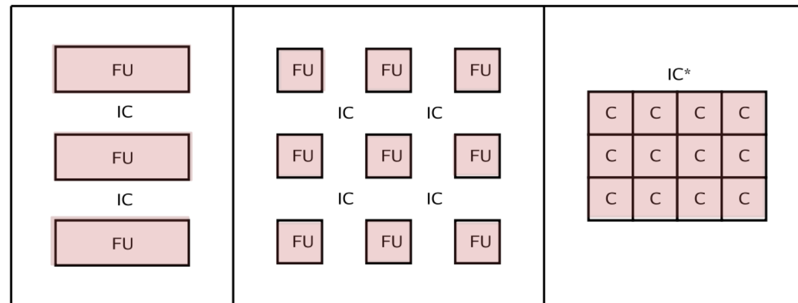


Figure 77. A: Zeilen-oder Spaltenstruktur, B: Blockstruktur, C: Sea-Of-Gates

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.

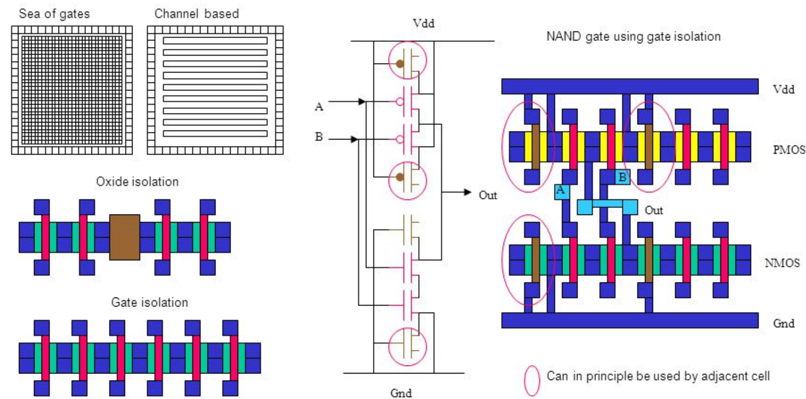


Figure 78. Beispiel eines NAND Gatters in Sea-of-Gate Technologie [Ashton Cockroft, slideplayer]

Herstellung von Mikrochips mit Siliziumtechnologie

[Intel]

9. Register-Transfer Architektur (RTL)

Ziele

- ➤ Verständnis der Funktionsweise von registerbasierter Datenverrabeitung durch Trennung in Daten- und Kontrollpfade
- ➤ Verständnis von Syntheseverfahren auf RT Ebene, Unterscheidung von Struktur- und Verhaltensbeschreibung

- Aufbau und Funktionsweise von Zustandsautomaten

9.1. 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 Hardwarebeschreibungssprachen zur Implementierung und Beschreibung der Algorithmik.
   - Zeitmodell: verfeinertes Kausalitätsmodell.
3. Register-Transfer-Ebene
   - Mikroprogrammebene
   - Aufteilung des Entwurfs von Algorithmen in Daten- und Kontrollfluss bzw. -pfaden.
   - 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.

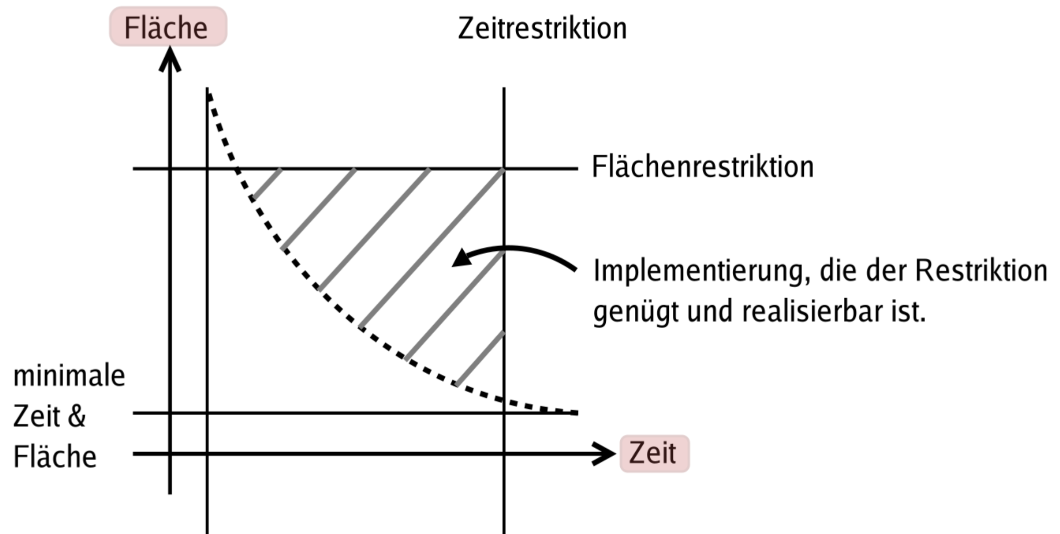


Figure 79. Kriterien für den Systementwurf auf Logikebene

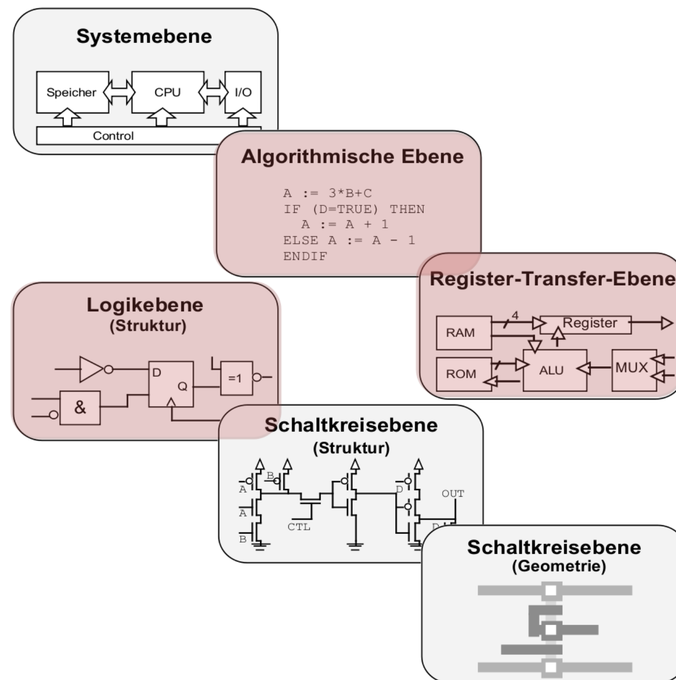


Figure 80. Ebenen beim Schaltkreisentwurf [Lehmann, Wunderm Selz]

Entwurfsablauf

1. Der Systementwurf wird durch Simulation und Verifikation rückgekoppelt.
2. Dem eigentlichen Syntheseprozess schließt sich das so genannte Technologieabbildung an, welches eine Netzliste mit Logikkomponenten auf die Zielhardware abbildet.
3. Simulation kann sowohl auf höheren Ebenen vor der Synthese durch Verhaltensmodellierung, aber auch nach der Synthese auf Logikebene stattfinden.
4. Fehlersuche im Testsystem

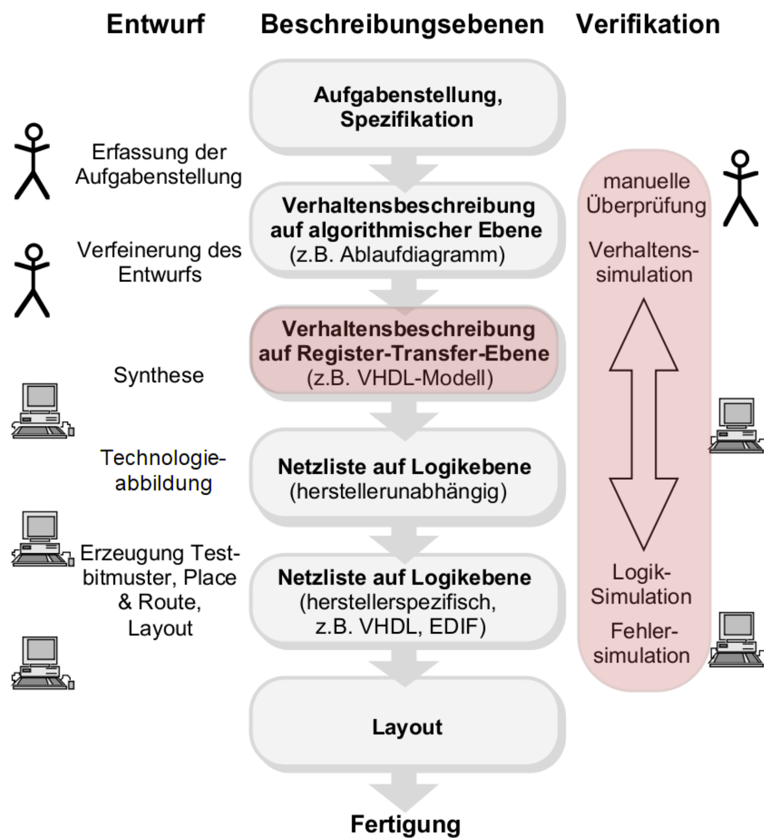


Figure 81. Entwurfsablauf mit VHDL

9.2. Modellierung

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

Strukturelle Beschreibung (SB)

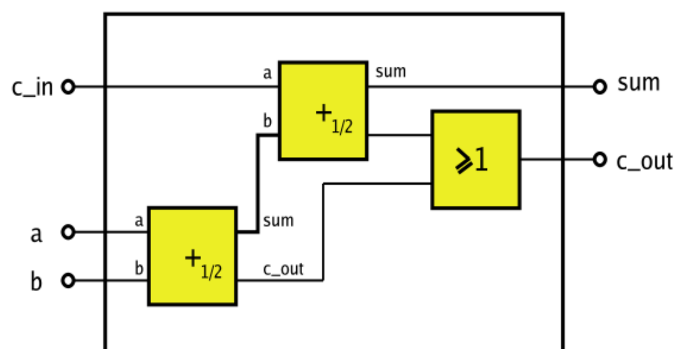
Hier findet die Systembeschreibung mittels einer Netzliste von Komponenten, wie Logikgattern, Registern sowie komplexeren Einheiten wie Addierern oder RAM-Komponenten statt. Die Netzliste bestimmt die Verbindungen der einzelnen Komponenten, vergleichbar mit einem Schaltplan auf Logikebene.

- In der strukturellen Beschreibung gibt es keine Kenntnis über das Verhalten einzelner Komponenten → Black Box Modell (Fehlen der logischen Schaltfunktionen, beschrieben z.B. mit einer Funktionstabelle)
- Kontroll- und Datenpfad sind Strukturbeschreibungen

Beispiel Strukturbeschreibung eines Volladierers

Netzliste

```
comp1:ADDH2(a,b,sum:t1,cout:t2)
comp2:ADDH2(cin,t1,sum,cout:t3)
comp3:AND2(t2,t3,cout)
```



VHDL

```
port(a: in bit, b: in bit,
     cin: bit, sum: out bit,
     cout: out bit) ...
component ADDH2 port (
  a: in bit,  b: in bit,
  cin: in bit, sum: out bit,
  cout: out bit
  );
component AND2 port(...);
signal t1,t2,t3: bit;
comp1: ADDH2 port map (...)
comp2: ADDH2 port map (...)
comp3: AND2 port map (...)
```

Verhaltensbeschreibung (VB)

Hier wird explizit die Abbildung von Eingangs- auf Ausgangssignalen beschrieben, 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 und die daraus folgende Gatterliste die SB!
- VB lässt sich mittels imperativer und funktionaler Sprachkonstrukte ausdrücken.

Example 5. (*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)
```

➤ Neben VB und SB des hardwareverhaltens unterscheidet man noch die Technologiebeschreibung (TB)

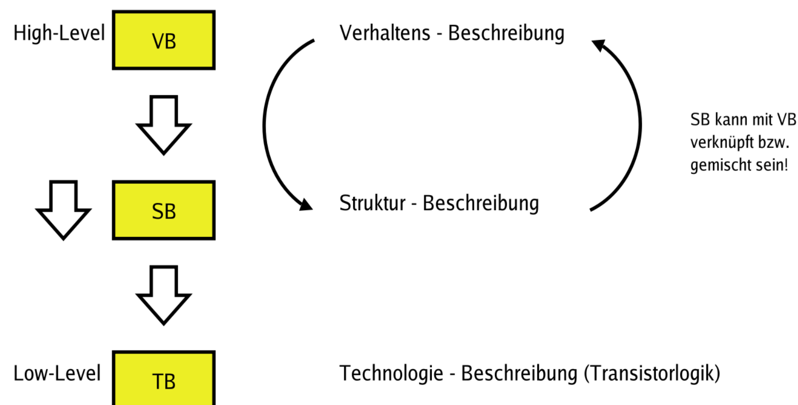


Figure 82. Hierarchische Modellierung mit VB, SB, und TB.

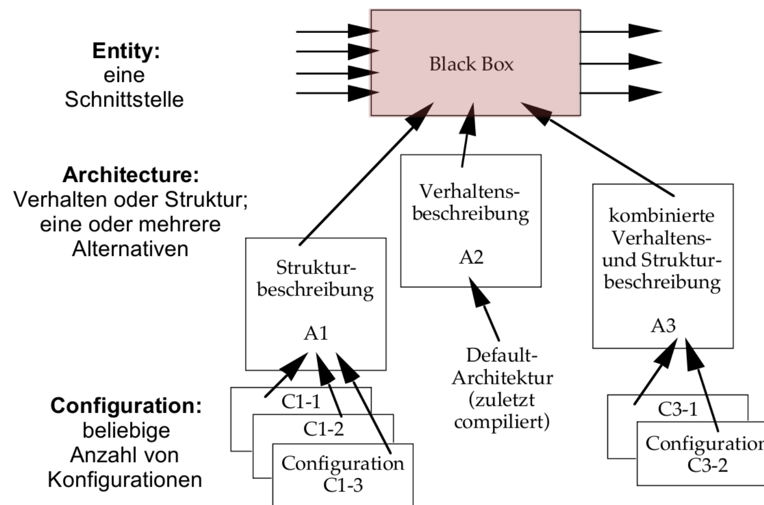


Figure 83. Hierarchische Systemmodellierung von Hardwarekomponenten in VHDL mit Entity, Architecture und Configuration [Lehmann, Wudner, Selz]

9.3. 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

  ❏ Fläche   Hardware   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, logische) und den Transfer der Daten zwischen Registern charakterisiert.

➤ Steuer- und Datenteil ”kommunizieren” über Steuersignale miteinander (bzw. sind verknüpft miteinander).

RT-Implementierung: Aufteilung

1. Der Datenteil besteht aus einer Menge von Modulen: Addierer, Komparatoren, 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 Datenmulitplexer und aktivieren Datenoperationen.
4. Die Bedingungsnetze liefern Ergebnisse von Test-Ausdrücken für daten- oder signalabhängige Verzweigungen der Zustandsübergänge.

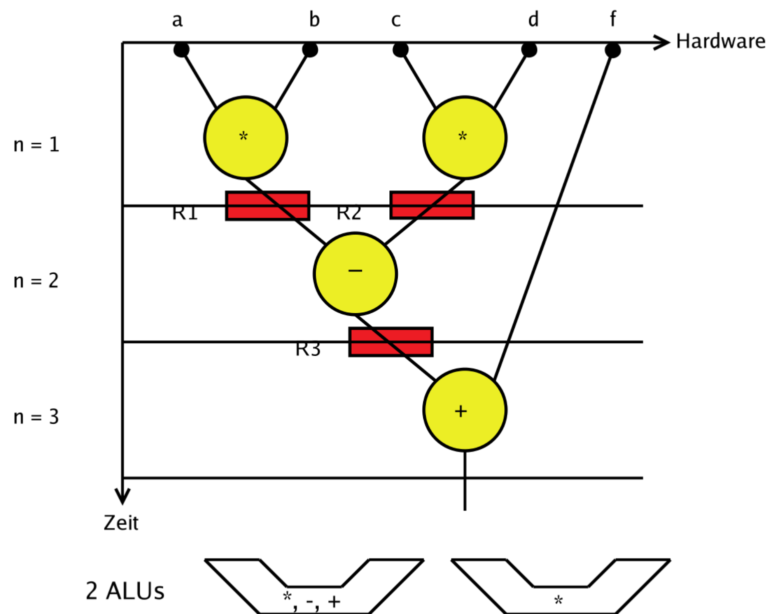


Figure 84. Beispiel (A) Umsetzung einer VB ab-cd+f mit RTL durch Scheduling und Ressourcenallokation. Die Register R zwischen den einzelnen Stufen bilden eine Pipeline.

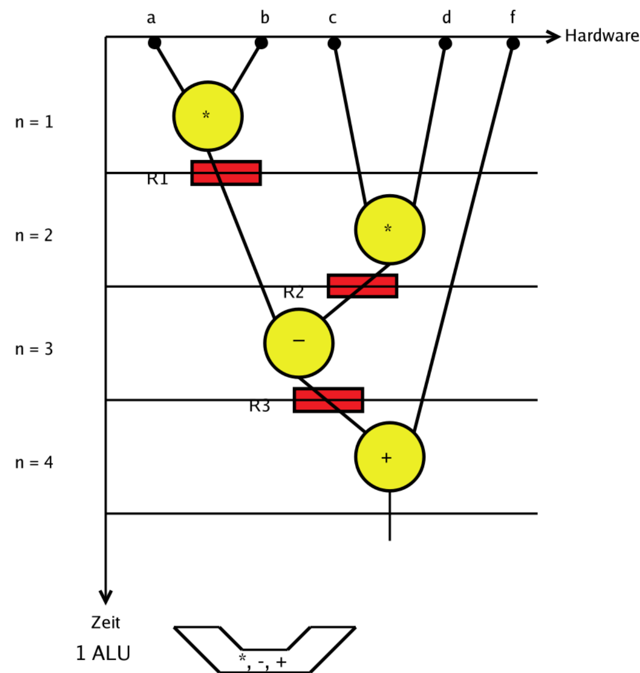


Figure 85. Beispiel (B) Umsetzung einer VB mit RTL durch Scheduling und Ressourcenallokation mit veränderten Parametern.

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 Multiplexer 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 ebenfalls durch Einfügen von Registern zwischen den einzelnen Funktionseinheiten des Datenpfades.

Diese Register erfüllen also folgende Aufgaben:

1. **Barrieren für Metastablilität und Hazards**. 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 Ausgangsdatensignale für eine bestimmte Zeit Δt als metastabil anzusehen, d.h. in diesem Zeitraum können einzelne Signalleitungen mehrfach ihren Logikzustand ändern.

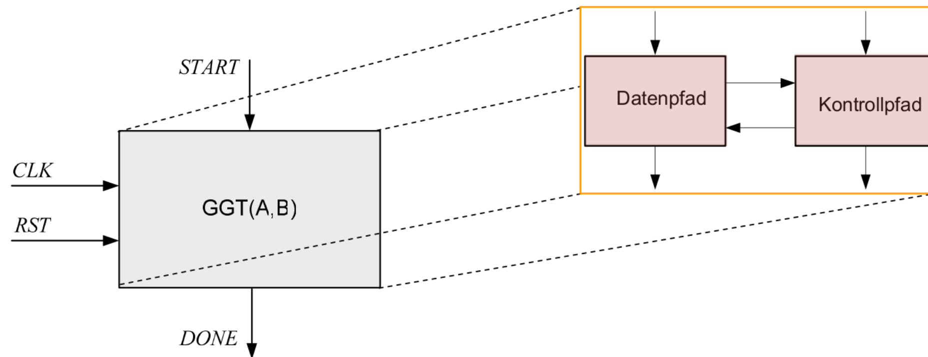
   ➤ Diese Metastabilität würde sich ohne Register (Puffer) durch alle folgenden Funktionseinheiten fortpflanzen, was sich nachteilig auf die Leistungsaufnahme und das elektrische Verhalten der Digitallogik auswirken kann

2. **Datenverarbeitung in einer Pipeline**. Besteht der Datenpfad aus N Funktionseinheiten die durch Registerebenen getrennt sind, so werden wenigstens N Taktzyklen benötigt, bis ein Ergebnis am Ausgang des Datenpfades anliegt, d.h. die Latenz ist $L \geq N T_{\mathsf{clock}}$.

   ➤ 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 $R < L$.

3. **Zwischenspeicher für Ressourcenteilung**. Werden Ressourcen (Funktionseinheiten) in einer Schaltung geteilt, so bedarf es Zwischenspeicher (und Multiplexer).

9.4. Register-Transfer Ebene: Beispiel und Entwurf



Zu entwerfen ist eine Schaltung, die den grössten gemeinsamen Teiler zweier 8-bit unsigned integer Zahlen A und B berechnet.

- Schnittstellensignale: `START, DONE, CLK, RST`
- Ablauf: Bei `START=1` soll die Berechnung beginnen.
- Mit `DONE=1` wird das Ende der Berechnung angezeigt.

Aus: Prof. Dr. Marco Platzner, Grundlagen der Technischen Informatik (GTI) / Digitaltechnik (DT)

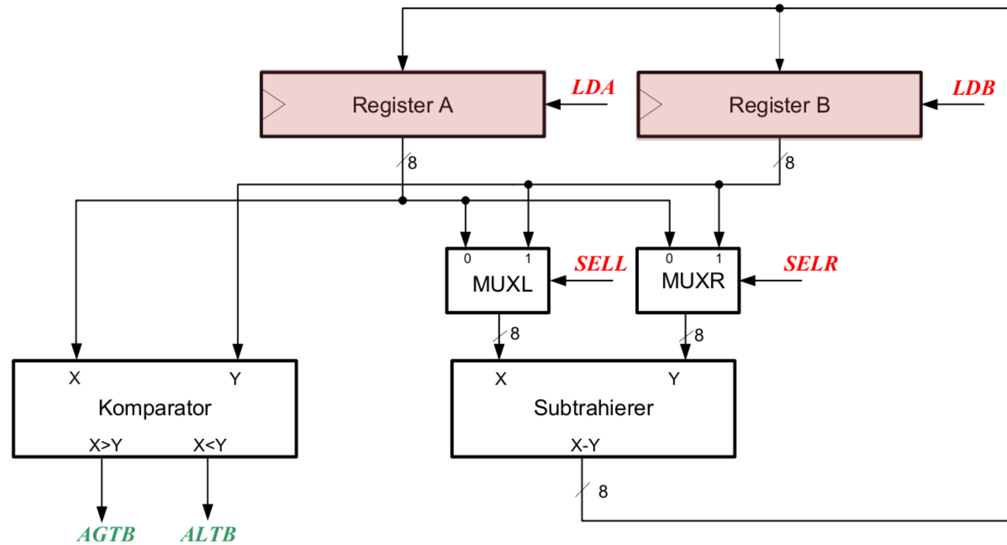
Spezifikation

Algorithm 4. (*GGT(a,b) - Algorithmus*)

$done := FALSE$;
repeat
  if $(a > b)$ **then** $a := a - b$;
  else if $(b > a)$ **then** $b := b - a$;
  else $done := TRUE$;
until $done = TRUE$;

Welche Komponenten benötigen wir im Datenpfad?

- Register für a und b
- Komparator: Schaltung zum Vergleichen von a und b: $a > b$, $a < b$, a=b
- Subtrahierer
- Multiplexer an den Subtrahierereingängen zum "Vertauschen" von a und b, damit wir a-b und auch b-a berechnen können

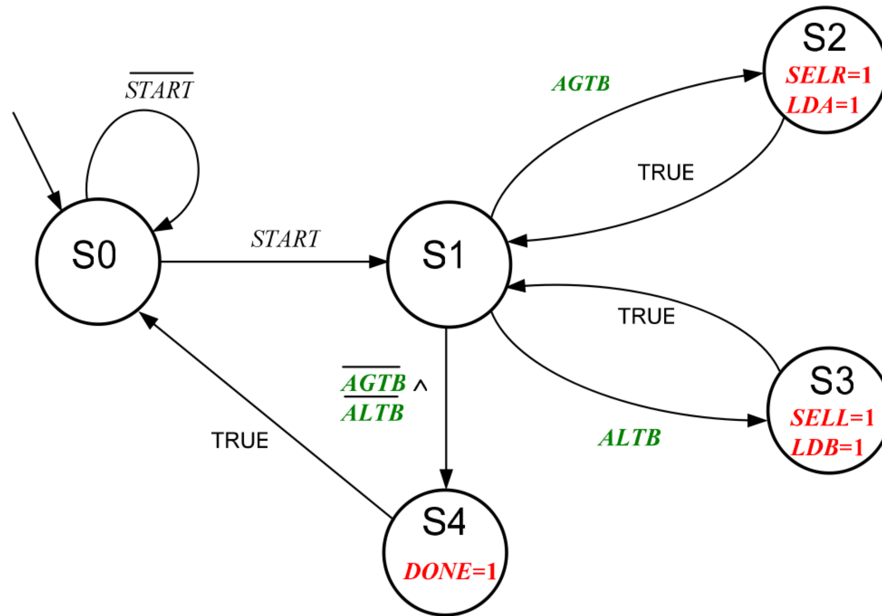


Signale

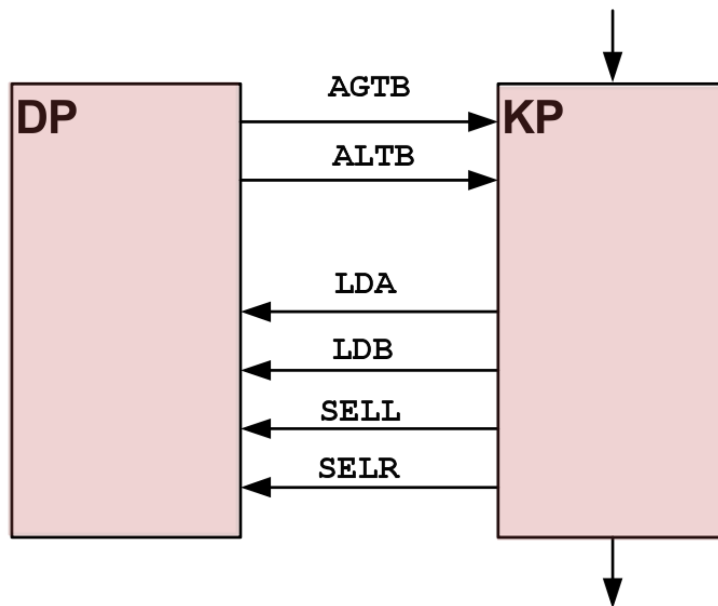
- Steuersignale (Operationen): `LDA` (load register A), `LDB` (load register B), `SELL` (select left MUX), `SELR` (select right MUX)
- Statussignale (Bedingungen): `AGTB` (A greater than B), `ALTB` (A less than B)

Zustandsdiagramm

- Der sequenzielle Ablauf der GGT Berechnung wird in fünf Zustände $\{S_0, S_1, S_2, S_3, S_4\}$ unterteilt.



Daten- und Kontrollpfad



➤ Der Zustandsautomat kann mit Hilfe eines ROM/RAM Baustein und einer Zustandsübergangstabelle implementiert werden.

- ❏ Die Eingabesignale {`S0,S1,S2,ALTB,AGTB,STRT`} dienen als Zeilenselektor der Tabelle. Sie bestehen aus dem binärkodierten Zustandsvektor S und den Statussignalen.
- ❏ Die Ausgangssignale {`S0,S1,S2,DONE,SWAP,LOAD,SELA,SELB`} ergeben binärkodiert die Zeilenwerte.

```
------------------------------------
Input         Output
5 4 3 2 1 0   7 6 5 4 3 2 1 0

A A S S S S   D S L S S S S S
L G T 2 1 0   O W O E E 2 1 0
T T R         N A A L L
B B T         E P D A B
------------------------------------
0 0 0 0 0 0   0 0 0 0 0 0 0 0   00
0 0 0 0 0 1   0 0 1 1 1 0 1 0   3A
0 0 0 0 1 0   1 0 0 0 0 1 0 1   85
0 0 0 0 1 1   0 1 0 1 0 0 1 0   52
0 0 0 1 0 0   0 0 0 0 1 0 1 0   0A
0 0 0 1 0 1   0 0 0 0 0 0 0 0   00
0 0 0 1 1 0   0 0 0 0 0 0 0 0   00
0 0 0 1 1 1   0 0 0 0 0 0 0 0   00
```

```
States:  S0, SL, S1, S2, S3, S4
Input:   STATE(0..2), START, AGTB, ALTB
Output:
STATE(0..2), SELB, SELA, LOAD, SWAP, DONE:=0

S0: 000
    ->SL| START=1
SL: 001
    LOAD:=1, SELA:=1, SELB:=1
    ->S1
S1: 010
    ->S2| AGTB=1
    ->S3| ALTB=1
    ->S4| AGTB=0&ALTB=0
S2: 011
    SWAP:=1, SELA:=1
    ->S1
S3: 100
    SELB:=1
    ->S1
S4: 101
    DONE:=1
    ->S0
```

Aufgabe

1. Implementiere den GGT Algorithmus in RTL unter Verwendung des RETRO Simulators.

   ➤ Der Zustandsautomat wird durch eine Tabelle in einem ROM Baustein ersetzt.

2. Erstelle die Zustandsübergangstabelle und übertrage die Inhalte in den ROM
3. Teste die Schaltung exemplarisch

9.5. Zusammenfassung

Taxonomie der Register-Transfer Architektur

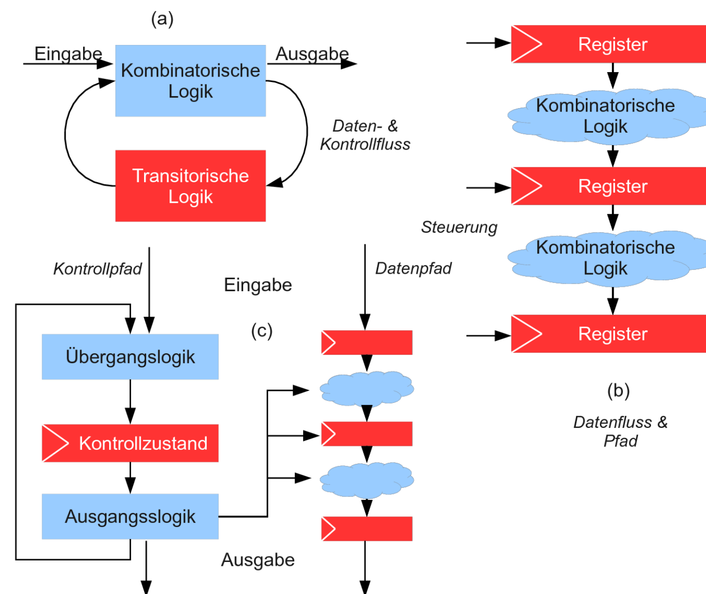


Figure 86. (a) Abstraktes Architekturmodell (b) Datenpfad (c) Kontroll- und Datenpfad

10. Hardwarebeschreibungssprachen

ZIELE

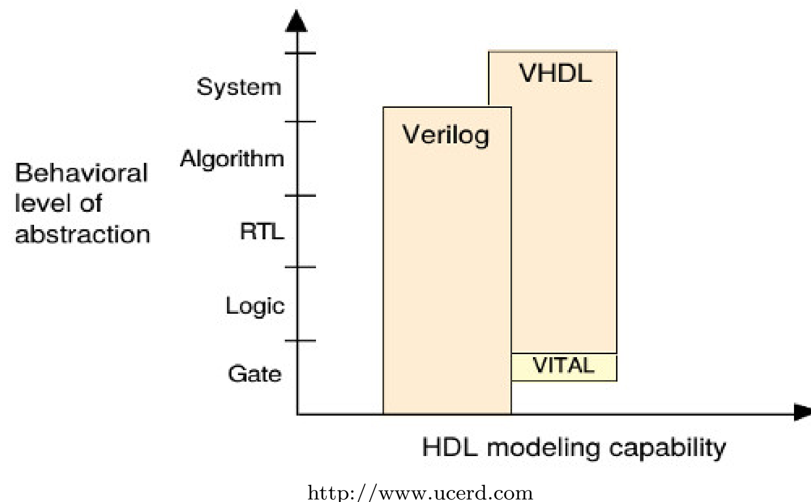
- ➤ Verständnis der Sematik von Sprachen zur Modellierung des Verhaltens von digitalen Schlatkreisen
- ➤ Verständnis und Anwendung der Grundelemente von VHDL
- ➤ Unterscheidung von Struktur- und Verhaltensbeschreibung

10.1. Überblick

Hardwarebeschreibungssprachen (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 prozessororientierten 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.



http://www.ucerd.com

Figure 87. VHDL und Verilog decken einen Großteil der Modellierungsebenen beim Hardwarentwurf ab. VHDL bietet bessere Abstraktionsmöglichkeiten.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(C, SI : in std_logic;
       SO : out std_logic);
end shift;

architecture Behav of shift is
  signal tmp: std_logic_vector(7 downto 0);
begin
process (C)
begin
  if (C'event and C='1') then
    for i in 0 to 6 loop
      tmp(i+1) = tmp(i);
    end loop;
    tmp(0) = SI;
  end if;
end process;
SO = tmp(7);
end Behav;
```

Verilog

```
module shift (C, SI, SO);

input C,SI;
output SO;
reg [7:0] tmp;

  always @(posedge C)
    begin
      tmp = tmp << 1;
      tmp[0] = SI;
  end
  assign SO = tmp[7];
endmodule
```

Figure 88. Syntaktischer und semantischer Vergleich von VHDL und Verilog am Beispiel eines 8-Bit Schieberegsietrs

10.2. VHDL

- VHDL bietet im Gegensatz zu Verilog HDL ein universelles und nutzerdefinierbares **Typensystem**. VHDL ist semantisch streng typisiert.
- VHDL bietet ein
  - universelles [konkrete und abstrakte Datentypen],
  - umfangreiches [`bit, signed, float, string, …`],
  - erweiterbares [`type defintion`],
  - strenges [`type a` ≠ `type b`]

  Typsystem. Strenge Typisierung reduziert Entwurfsfehler, erhöht aber den syntaktischen und semantischen 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;
```

- Eine *Funktion* beschreibt in VHDL eine Abbildungsvorschrift (i.A. kombinatorische Logik) von Eingangs- auf Ausgangssignale! Rein funktional ohne Speicher und sequenziellen Ablauf!
- VHDL ist modulorientiert und hierarchisch strukturiert.

10.3. VHDL - Aufbau

Die Beschreibung einer Digitallogikschaltung, Komponente genannt, benötigt wenig- stens zwei Strukturelemente. Ein Modul (eine VHDL Quelldatei) ist daher unterteilt in:

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

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

Architektur → Architecture
Eine Architektur enthält die Beschreibung der Funktionalität und/oder der Struktur eines Moduls.

- Verhaltenbeschreibung VB, Strukturbeschreibung SB in Form einer Netzliste und Komponentent, die auch verschiedene VHDL-Module verbinden kann.
- Eine Entity bzw. ein Modul kann durch verschiedene Architekturen beschrieben werden. Die Zuordnung einer Architektur zu einer Entity findet in der sog. Konfiguration statt.

- Ein Modul (eine VHDL Quelldatei) ist unterteilt in (Forts.):

Package [optional]

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.

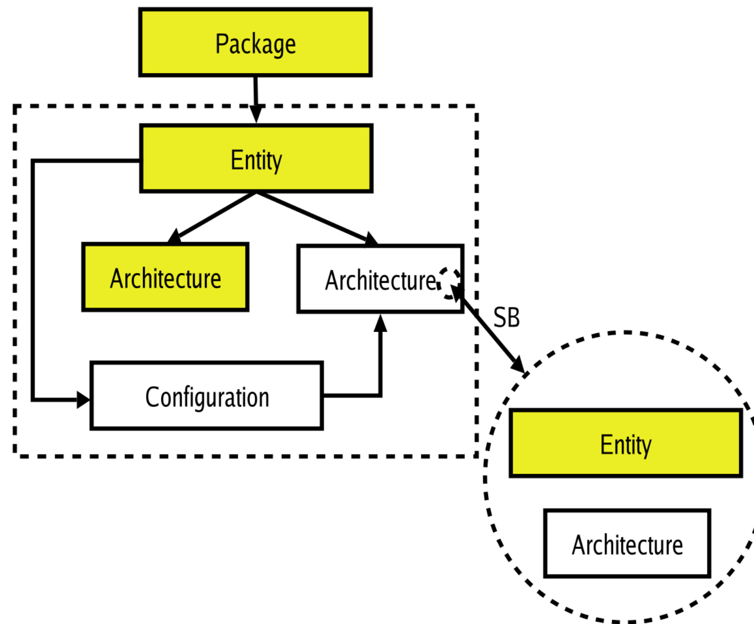
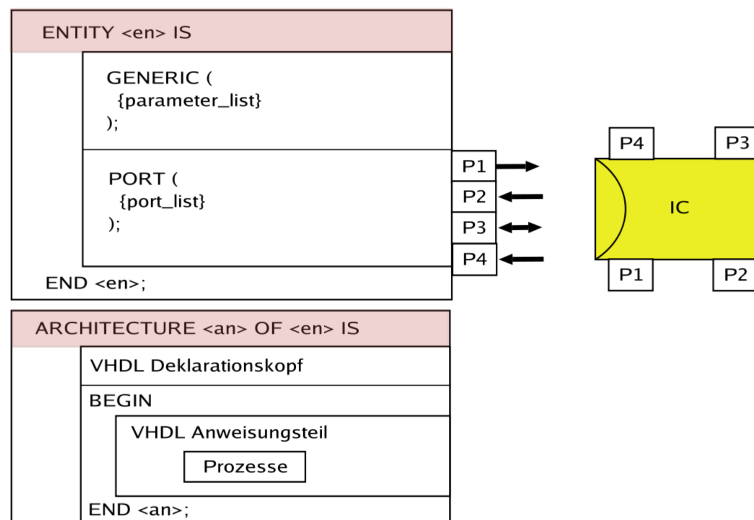


Figure 89. Hierarchisch aufgebaute Struktur in VHDL

Definition eines VHDL Moduls und Analogie zur einer IC-Komponente

➤ Ein VHDL Modul beschreibt die äussere Schnittstelle einer Komponente (Black Box) in der Entity und deren Aufbau oder Funktion (Architecture)



- 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 Modul muß im VHDL Deklarationskopf gemäß ihrer Anschlüsse deklariert werden, und kann im VHDL-Anweisungsteil (Architektur-Körper) eingebunden werden (SB).

10.4. VHDL - Komponenten

- Die strukturelle Modellierung unter- scheidet zwischen der Komponentende- finition, der Verhaltensbeschreibung einer Komponente (Entity), und der Kom- ponenteninstanziierung, d.h. die Einbindung von Komponenten.
- Die einzubindenden Komponenten müssen mit ihrer Port-Schnittstelle (identisch zum Entity-Port) im Dekla- rationsteil der Architektur eingeführt werden.
- Eine Komponente eines bestimmten Typs kann beliebig oft dupliziert im Ar- chitekturkörper eingebunden werden.
- Jede Komponenteninstanz ist unabhängig von jeder anderen, d.h. mit eigener Digitallogik synthetisiert.

Definition 1. (*Komponentendefinition*)

entity M_1 *is*
  port (..)
end M_1;
architecture *A of* M_1 *is*
begin
..
end *A*;

Definition 2. (*Komponenteninstanzierung*)

entity M_2 *is*
  port (..)
end M_2;
architecture *A of* M_2 *is*
  component M_1
    port (..)
  end component;
begin
  $Instance_1$: M_1 **port map** (..);
  $Instance_2$: M_1 **port map** (..);
  ..
end *A*;

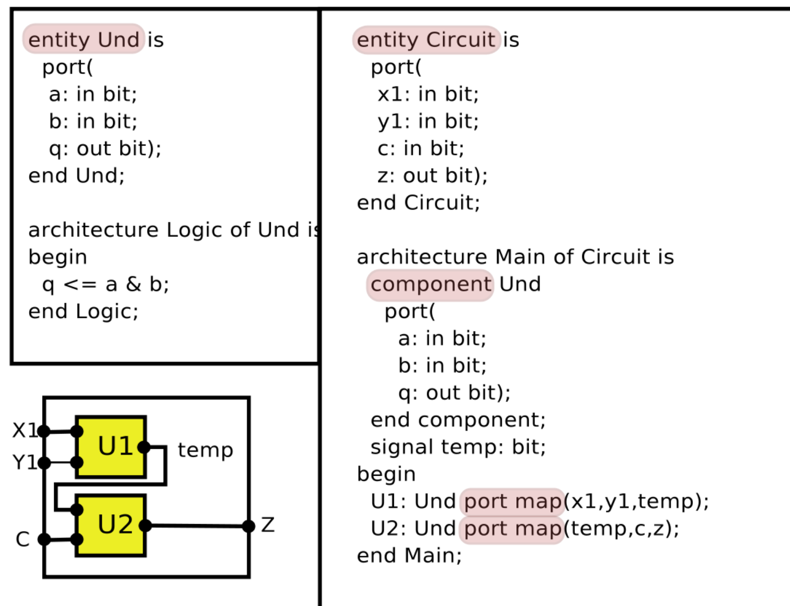


Figure 90. Komponenteninstanzierung in VHDL.

10.5. VHDL - Signale

➤ Signale sind die eigentlichen Datenobjekte in VHDL.

➤ Der einfachste Signaltyp ist ein Bit mit der Wertemenge bit={0,1}.

➤ Ein Signal kann aus einer Bitgruppe == Datenwort mit beliebiger Anzahl von Einzelsignalen bestehen, genannt Vektor.

➤ 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 Architekturdeklarationsteils lokal sichtbar erzeugt werden.

➤ Definition eines globalen Portsignals und Angabe der Signalrichtung.

Definition 3. (*Portdeklaration einer Entity*)

port (
signalname$_1$: *dir*$_1$ *type*$_1$;
signalname$_2$: *dir*$_2$ *type*$_2$;
..
)
typeof *dir* = **in** | **out** | **inout** | **buffer**
typeof *type* = **bit** | **std_logic** | **bit_vector** | ..

Beispiel einer Port-Deklaration:

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

Definition 4. (*Definition eines lokalen (internen) Architektursignals mit der Möglichkeit der Initialisierung*)

signal *signalname* : *type* := *init* ;
signal *signalname* : *type*$_{vector}$ (*a* **downto**|**to** *b*) := *init* ;

- Signalvektoren 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 *b* `downto` *a* verwendet, befindet sich das MSB immer auf der linken Seite einer Binärzahl *MSB* .. *LSB* , wird die Indexrichtung *a* `to` *b* gewählt, auf der rechten Seite *LSB* .. *MSB*.
- Der untere Index *a* kann mit jedem beliebigen natürlichen Zahlenwert gewählt werden. Der Wert 0 mit der *b* `downto` *a* Darstellung ist aber zweckmäßig anzuwenden.
- Beispiel einer Architekturdeklaration mit Signalen.

```
architecture main of en is
  signal incr: bit;
  signal data : signed(7 downto 0);
begin
  incr <= WE and ADDR=X"01" ;
  process (write,data)
    if incr = '1' then
      data <= data + 1;
    else
      data <= data - 1;
    end if;
  end process;
end;
```

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

Definition 5. (*Signalzuweisung*)

signalname <= *expression*;
expression ::=
  [*Konstante*] → *vom gleichen Typ wie das Signal*
    → *Bit* : '0' | '1'
    → *Bitvektor* : "*XX*..." *mit* $X = \{0, 1\}$
  [*Arithmetischer Ausdruck*]
    → *operand* $\{+, -, *, ...\}$ *operand*
    → *Schachtelung* $expr_1(expr_2(expr_3(...)))$
    → *alle Operanden vom gleichen Typ, nur sinnvoll mit Vektoren*
  [*Logischer Ausdruck*]
    → *operand* {*and*, *or*, ...} *operand*
  [*Relationaler Ausdruck*]
    → *operand* {=, <>, ...} *operand*

Nebenläufige Signalzuweisungen, dass sind alle Toplevelanweisungen im Architekturkörper und ausgenommen bedingte Ausrücke in Prozessen, dürfen immer nur ***eine Quelle*** *besitzen, d.h. in obigen Beispiel ist immer nur eine Signalzuweisung erlaubt!*

10.6. VHDL - Signaltypen

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

boolean (V)
: Wertemenge {`true`,`false`}

integer (V)
: {keine spezifische Datenbreite, wird bei der Syntehse bestimmt, i.A. wird aber nur eine Wertemenge von 32/64 Bit unterstützt}

natural (V)
: Wertemenge {natural   integer   0}

positive (V)
: Wertemenge {positive   integer > 0}

bit (V)
: Wertemenge {`0,1`}

bit_vector(natural) (V)
: Definiert ein Array von Bits

character (V)
: Textzeichen, Wertemenge mit ASCII Kodierung

string(natural) (V)
: Definiert ein Array von Textzeichen

std_logic (E)
: Wertemenge {`0,1,H,L,U,Z,-`}, muss über das Paket IEEE.STD_LOGIC_1164.ALL eingebunden werden.

std_logic_vector (E)
: Definiert ein Array von *std_logic* {`0,1,H,L,U,Z,-`}, muss über das Paket `IEEE.STD_LOGIC_1164.ALL` eingebunden werden.

signed, unsigned (E)
: Vorzeichenbehaftete und vorzeichenlose ganze Zahlen mit Binärkodierung, muss über das Paket `IEEE.NUMERIC_BIT.ALL` eingebunden werden.

Aufzählungstyp

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

➤ Abstrakte konstante Aufzählungstypen definieren eine Menge von Symbolen, die noch keine konkrete Kodierung zugeordnet sind.

Definition 6. (*Aufzählungstyp und Signaldefinition eines Aufzählungstyps*)

type *typename* **is** (
  symbol$_1$,
  symbol$_2$,
  ..
);
signal *signalname* : *typename*;

➤ Der Symbolelementname ist ein beliebiger Identifizierername (nutzerdefiniert)

➤ Datentypen werden i.A. im Kopfteil einer Architektur definiert.

➤ Beispiel eines Aufzählungstyps:

```
...
type states is (
  S_start,
  S_loop,
  S_end
);
signal state: states;
signal next_state: states;
begin
  ...
  process fsm()
    case state is
      when S_start => next_state := S_loop;
      when S_loop => ...
    end case;
  end process;
```

Aufgabe

1. Welche Kodierungen könnten die Symbolemente eines Aufzählungstyps in der Digitallogik besitzen?
2. Nenne Vorteile und Nachteile der unterschiedlichen Kodierungen

10.7. VHDL - Arraytypen

➤ 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 in der Verhaltensbeschreibung 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 7. (*Arraytyp, Erzeugung eines Arrayobjekts und Zugriff auf Arrayzellen*)

type *arrayname* **is array**({*range*}) **of** *celltype*;

signal *signalname* : *arrayname*;

signalname(*index*|*range*) := *expression* (*signalname*(*index*|*range*))

➤ Zuerst muss ein Arraytyp definiert werden, dann können Signale definiert werden.

➤ Beispiel für Arrays

```
type ram is array(0 to 255) of bit_vector(7 downto 0);
signal ram1: ram;
signal data: bit_vector(7 downto 0);
... data <= ram1(addr); ...
```

10.8. VHDL - 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 (Prozesse)
2. Nebenläufige Anweisungen

- ➤ Sequenzielle Anweisungen können nur in Prozessen verwendet werden und erzeugen Speicher.
- ➤ Sequenzielle Anweisungen einer Hardwarebeschreibungssprache sind nicht mit Programmsequenzen zu verwechseln.
- ➤ Wenn überhaupt ist eine Sequenz im Sinne einer Laufzeithierarchie und Evaluierungsreihenfolge zu verstehen.
- ➤ Register können nur in Prozessen erzeugt werden.
- ➤ VHDL synthetisiert Register (FLIP-FLOPs, Latches) implizit, d.h. es gibt in VHDL keine Möglichkeit, ein Register explizit zu erzeugen. Die Registerinferenz ergibt sich aus der Verhaltenbeschreibung mit entsprechenden VHDL-Anweisungen.

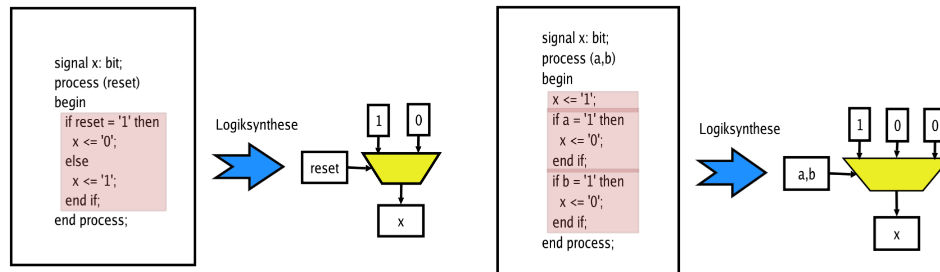
Ein Prozess ist eine Blockkapselung von VHDL Anweisungen und gehört zur VB. Prozesse werden zusammen mit Toplevelanweisungen ausserhalb von Prozessen konkurrierend/nebenläufig ausgeführt.

- ➤ Anweisungen (= Evaluierungsvorschriften) in einem Prozess sind auf gleicher Ebene zueinander nebenläufig zu betrachten.
- ➤ Alle Anweisungen (Signalzuweisungen) in einem Prozess werden zu einem einzigen finalen Ausdruck synthetisiert!

Definition 8. (*Prozess*)

[*name* :] **process** ($sig_1, sig_2, ..$)
  [*Variablendeklaration*]
begin
  $statement_1$;
  $statement_2$;
  ..
end process [*name*];

- ➤ Beispiel Multiplexer-Inferenz mit Signalzuweisung in beiden Zweigen der bedingten Verzweigung (linkes Bild).
- ➤ Beispiel Multiplexer-Inferenz mit Standardsignalzuweisung (Default) und mehreren unvollständigen bedingten Verzweigung (rechtes Bild).



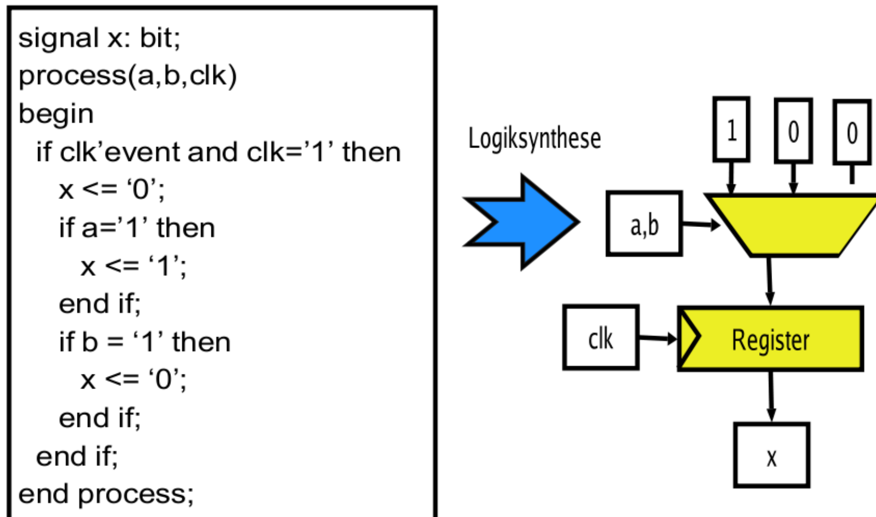
- Unvollständige bedingte Verzweigungen können (und sollen!) Register erzeugen.
- Immer wenn es einen bedingten Fall (Ausdruck) in einem Prozess gibt, der bei einer bestimmten Eingangsvektor kein Ergebnis eines bestimmten Ausgnagssignals ergibt, ist gemäss dem zu grundeliegenden Modell ein Register als Zustandsspeicher erforderlich.
- Dabei muss man zwischen taktflankengesteuerten (dynamischen) und Registern und datengesteuerten (statischen) Latches unterscheiden.
- Eine Signaländerung (Flanke) kann kombinatorisch durch ein *event* Attribut ageleitet werden: `signalname'event`.

Definition 9. (*Registererzeugung in VHDL*)

Statisch, *Latch*, *nicht taktgesteurt* (*Zustand*)
process (*clock*, ..)
  if *clock* = *level* **then**
    register <= *expression*;
  end if;
end process;

Dynamisch, *Register*, *taktgesteuert* (*Ereignis*)
proecss (*clock*, ..)
  if *clock*′**event and** *clock* = *level* **then**
    register <= *expression*;
  end if;
end process;

- Beispiel Registerinferenz mit Datenmultiplexing über ein unvollständige bedingte Verzweigung



- Prozesse modellieren prozedurale Vorgänge.
- Das Modell eines prozeduralen Verhaltens sieht neben Ablaufsteuerung die “scheinbare” Aktivierung und Suspendierung (Blockierung) des Ablaufes eines Prozesses vor.
- Toplevelanweisungen sind im Gegensatz zu Prozessen immer aktiv, worin begründet ist, daß Toplevelanweisungen 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. Aktivierung bei Änderung von Signalen aus der Liste sensitiver Signale im Prozesskopf. Immer wenn sich eines dieser Signale ändert, wird der Prozess aktiviert, d.h. er evaluiert neue Werte der Ausgangssignale.
    2. Wait-Anweisung. Der Prozeß wird aktiviert, wenn die Bedingung einer Wait-Anweisung (ein boolescher Ausdruck) erfüllt ist.

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

Prozesse mit Eingangssignalaktivierung

process (sig_1, sig_2, ..)
begin
  statement$_1$;
  statement$_2$;
  ..
end process;

Prozesse mit Wait Aktivierung

process ()
begin
  statement$_1$;
  statement$_2$;
  ..
  wait on *signal*;
  wait until *expression*;
  wait for *time*;
end process *name*;

Verzögerungsmodelle

- Das zeiltiche Konditional bei der *wait* Anweisung ist *nicht* synthetisierbar, nur simulierbar.

  - Technologische Verzögerung ist i.A. nicht parametrisierbar sondern durch Technologie und Design vorgegeben.

- Es gibt in VHDL verschiedene temporale Verzögerungsmodelle die an die Ausdrucksevaluierung gebunden sind [I]:

Trägheitsverzögerung (Intertial Delay)
: Standard-Verzögerungsmodell: Geeignet für die Modellierung von Verzögerungen durch Geräte mit Trägheit (z.B. Logikgattern). **Impulse, die kürzer als die Verzögerung eines Geräts sind, werden nicht an den Ausgang weitergeleitet**

Transportverzögerung:
: Modellverzögerungen durch Elemente ohne "Trägheit", z. B. Drähte. Keine Trägheit bedeutet alle Eingangsereignisse werden zu Ausgangssignalen übertragen. Ein beliebiger Impuls, nicht zu kurz, wird weitergeleitet.

Delta Verzögerung

Was ist mit Modellen, bei denen keine Propagierungsverzögerungen angegeben sind? Eine unendlich kleine Verzögerung wird automatisch vom Simulator eingefügt (→ 0ns), um die korrekte Reihenfolge der Ereignisse zu erhalten

- Die Delta-Verzögerung ist ein Sonderfall der Verzögerung, die unendlich klein ist
- Signalzuweisungen in Prozessen werden erst am Ende der Aktivierung eines Prozesses sichtbar!
    - D.h.: Innerhalb einer Prozessaktivierung kann ein Signal auf der linken Seite einer Zuweisung nicht unmittelbar wieder in einem Signalausdruck auf der rechten Seite einer Zuweisung verwendet werden!
- Folgende Entwicklung der Signalwerte zweier Signale x und y ergeben sich im folgenden Beispiel:
  t_n: $\{a=1,\ x=1,\ y=1\} \rightarrow t_{n+1}$: $\{a=1,\ x=2,\ y=0\} \rightarrow t_{n+2}$: $\{x=2,\ y=1\}$!

```
signal a,x,y: integer;
process (a,x,y)
begin
  x <= a+1;
  y <= x-1;
  wait for 10ns; -- Nur zur Simulation
end process
```

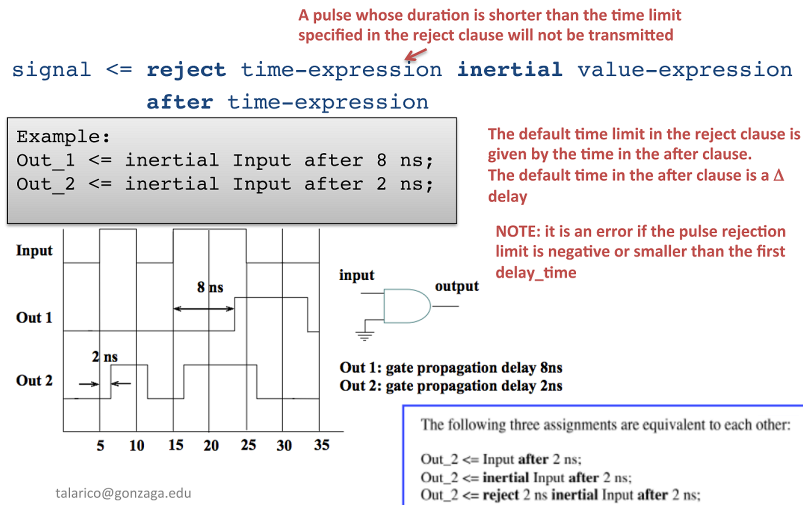


Figure 91. Allgemeine Form des *delay* Operators in Ausdrücken und Beispiele für die Trägheitsverzögerung

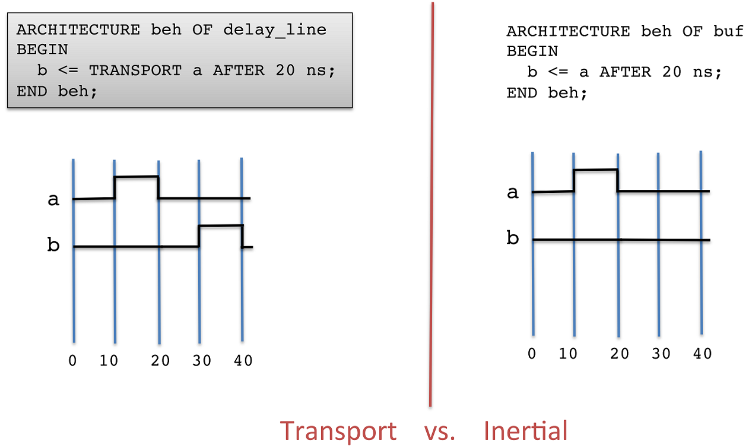


Figure 92. Vergleich der Transport- mit der Trägheitsverzögerung

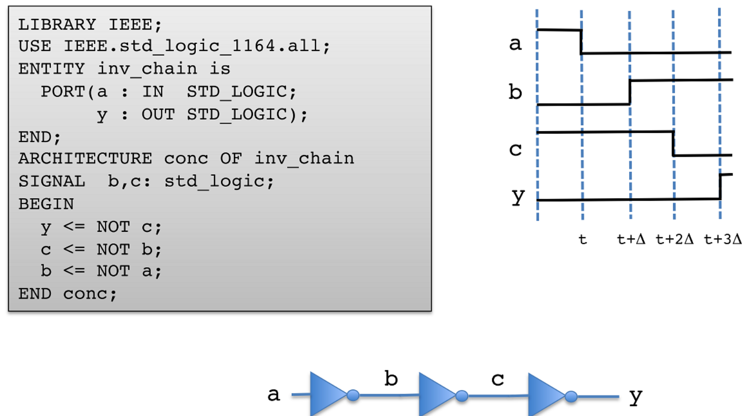


Figure 93. Delta Verzögerung: Implizites Zeitmodell oder nur Abhängikseitsmodell?

Zusammenfassung:

- ➤ Inertial
  - ❏ Für Geräte mit Trägheit (d.h. Bei jedem physischen Gerät)
  - ❏ Impulse, die kürzer als die Verzögerung des modellierten Geräts sind, werden nicht weitergegeben.
  - ❏ VHDL 1993 unterstützt Impulsabstandbreiten
- ➤ Transport
  - ❏ Gewährleistet die Propagierung aller Ereignisse (beliebiger Puls, wird propagiert)
  - ❏ Wird normalerweise zum Modellieren von Elementen verwendet, z.B. ideale Drähte.
- ➤ Delta
  - ❏ Wird automatisch eingefügt, um die funktionale Korrektheit des Codes zu gewährleisten, der keine Verzögerung angibt
  - ❏ Erzwingt die im Code angegebenen Datenabhängigkeiten - Dies ist das Verzögerungsmodell für synthetisierbaren Code

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 Prozessverarbeitung):
- Variablenwerte werden sofort bei der Evaluierung 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.

Definition 10. (*Prozessvariablen*)

process
  variable *vname* : *typ*;
begin
  vname := *expression*;
end process;

- Beispiel für die Verwendung von Variablen in Prozessen zur Wahrung von Auswertereihenfolgen:

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

- Folgende Signaländerung sollte sich ergeben: t_{n}: $\{a{=}1,b{=}1\} \rightarrow \{y{=}61\}$

Aufgaben

1. Prüfe den Vergleich von abhängigen Signal- und Variablezuweisung in Prozessen an einem einfachen Beispiel mit dem Simualtor *ghdl* (und Visualisierung mit *gtkwave*)
2. Unetrusche und vergleiche die verschiedenen VHDL Verzögerungsemodelle in der Simulation

10.9. VHDL - Bedingte Ausdrücke und Anweisungen

Bedingte Ausdrücke

➤ Nur ausserhalb von Prozessen in Toplevelanweisungen

Definition 11. (*Bedingte Ausdrücke*)

signame <= *expression*$_1$ **when** *condition*$_1$ **else**
expression$_2$ **when** *condition*$_2$ **else** ..
expression$_{default}$

Beispiel

```
Y <= '0' when I = "00" else
     '0' when I = "11" else
     '1';
```

Bedingte Anweisung

➤ Nur innerhalb von Prozessen verwendbar

- ❏ Bei der bedingten Verzweigung ist der Else-Zweig optional.
- ❏ Der Ausdruck muss vom Typ boolean sein.
- ❏ Wenn aber die bedingte Anweisung nicht vollständig ist dann wird Speicher inferriert!
- ❏ Bei der Schachtelung muß ein verändertes Schlüsselwort *elsif* verwendet werden.

Definition 12. (*Bedingte Ausdrücke*)

process
  if *condition$_1$* **then**
    statements
  [**elsif** *condition$_2$* **then** *statements*]
  [**else** *statements*]
  end if;
end process;

Beispiel

```
process
  if Y = "00" then
    I <= '0';
  elsif Y = "11" then
    I <= '0';
  else
    I <= '1';
  end if;
end process;
```

- In imperativen Programmiersprachen sind geschachtelte IF-THEN-ELSE Anweisungen und die Mehrfachauswahl isomorph bzw. äquivalent.
- Dies gilt nicht für eine Hardwarebeschreibungssprache. Bedingte Verzweigungen und die folgende Mehrfachauswahl besitzen unterschiedliche Prioritätsstrukur und zeitliches Verhalten:
  - CASE hat für alle Fälle i.A. konstante Signalverzögerungen bzw. Laufzeiten.
  - IF-THEN-ELSE besitzt aufsteigende Laufzeiten von Signalen für die verschiedenen Fälle (mutex).

Mehrfachauswahl

- Mit optionalen Restmengenfall *others*

Definition 13. (*Mehrfachauswahl*)

process
  case *expresssion is*
    when $value_1$ => *statements*
    when $value_2$ => *statements*
    ..
    [**when** *others* => *statements*]
  end case;
end process;

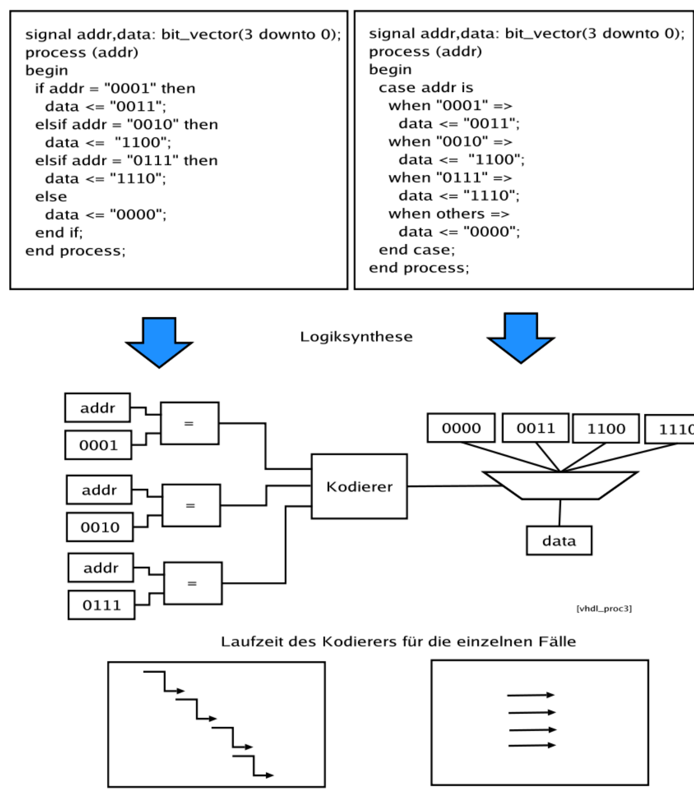


Figure 94. Beispiel für bedingte Verzweigung und Mehrfachauswahl. Die Synthese beider Beschreibungen kann unterschiedliche Ergebnisse liefern (Links: Prioritätskodierer!).

10.10. VHDL - 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 erst zur Lauf- zeit evaluiert. Diese Schleifen benötigen i.A. Zustandsautomaten für die (echte) sequentielle Abarbeitung. Diese Schleifen werden i.A. von Synthesewerkzeugen nicht unterstützt.

➤ Bei statischen Zählschleifen in VHDL muss der Schleifenindex nicht deklariert werden.

➤ Statische Schleifen werden bei der Synthese abgerollt, und die Statements im Schleifenkörper mit den entsprechenden Indexwert repliziert.

Definition 14. (*Zählschleife*)

[*label* :] **for** *index* **in** *range* **loop**
  statements
end loop [*label*] ;

range ::=
  b **downto** *a*
  a **to** *b*

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

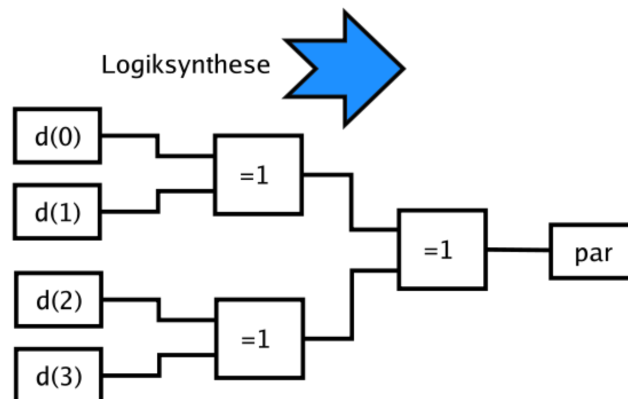


Figure 95. Beispiel statische Zählschleife und Synthese in kombinatorische Logik.

10.11. VHDL - Simulation

➤ Man unterscheidet zwei Klassen von Simulatoren und Simulationen:

- ❏ High-level Verhaltenssimulation ohne Technologiemodell und Synthese
- ❏ Low-level Simulation auf Gatterebene mit Technologiemodell und Synthese einer Gatternetzliste

➤ Eine Simulation besteht aus dem Device under Test (DUT) und einer

Testbesschreibung (Stimuli)

GHDL

- GHDL ist ein Verhaltenssimulator für die VHDL-Sprache.
- Mit GHDL kann VHDL-Code direkt in eine Maschinenprogramm kompiliert und ausgeführt werden.
  - GHDL unterstützt die Versionen IEEE 1076 VHDL (1987, 1993, 2002) und teilweise die letzte Version von 2008 (gut genug, um `fixed_generic_pkg` oder `float_generic_pkg` zu unterstützen).
- Durch die Verwendung eines Codegenerators ist GHDL viel schneller als jeder interpretierende Simulatoren.

Simulationsablauf

1. Erstellung eines DUT Modells in einer separaten Textdatei. Wichtig: Der Einheitsname und der Dateiname ohne Endung müssen identisch sein.
2. Erstellung einer Testbench die folgende Teile enthält:
   - Taktgenerator
   - Stimulusgenerator (Temporale Muster der Eingangssignale)
   - Monitoring
3. Analyse des VHDL Modells

```
> ghdl -a <testbench>.vhdl
```

4. Eloberation des Designs unter Angabe der Testbench Einheit

```
> ghdl -e <testbench>
```

5. Ausführen der Simulatorprogramms unter Angabe der Simulationszeit und der Ausgabedatei für Monitoringsignale (Port der Testbench Einheit)

```
> <testbench> --stop-time=xxus --vcd=<testbench>.vcd
```

6. Darstellung der Signale (Stimuli und Ausgabesignale)

```
gtkwave <testbench>.vcd
```

7. Bei neueren *ghdl* Versionen (ab 0.35?) fällt Eloberation und Ausführung durch den Run Modus zusammen:

```
> ghdl -r <testbench> --stop-time=xxus --vcd=<testbench>.vcd
```

Beispiel eines DUT Modells

```
library ieee ; use ieee.std_logic_1164.all ;
entity shift_reg is
  port ( serial_in , clk , shift_ena : in std_logic ;
         serial_out : out std_logic);
end shift_reg ;

architecture behav of shift_reg is
signal q : std_logic_vector (3 downto 0);
begin
  -- shift register process --
  process ( clk )
  begin
    if rising_edge ( clk ) then
      if shift_ena = '1' then
        q (3 downto 1) <= q (2 downto 0);
        q (0) <= serial_in ;
      end if ;
    end if ;
  end process ;
  serial_out <= q (3);
end behav;
```

Beispiel einer Testbench (1)

```
library ieee; use ieee.std_logic_1164.all;
entity shift_tb is
  port (
    clk_out : out std_logic;
    monitor : out std_logic
  );
end shift_tb ;

architecture behaviour of shift_tb is
  constant clk_period : time := 100 ns;
  component shift_reg
  port ( serial_in , clk , shift_ena : in std_logic ;
         serial_out : out std_logic );
  end component;
  signal clk, serial_in_test, shift_ena_test : std_logic;
begin
 ...
end behaviour;
```

Beispiel einer Testbench (2)

```
  clk_process: process
  begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
  end process;
  dut: shift_reg port map (
    serial_in_test,
    clk,
    shift_ena_test,
    monitor
  );
```

```
-- Stimulus Generator --
stimu_process: process
begin
  serial_in_test <= '1';
  wait for 50 ns;
  serial_in_test <= '0';
  wait for 350 ns;
  serial_in_test <= '1';
  wait for 250 ns;
  serial_in_test <= '0';
end process;
clk_out <= clk;
shift_ena_test <= '1';
```

11. Zustandsautomaten

Ziele

1. Verständnis der Funktionsweise von zustandsbasierten sequenziellen Logiksystemen
2. Verständnis der Funktionsweise und Aufbau von Zustandsautomaten
3. Entwurf von Zustandsautomaten mit VHDL

11.1. RTL und Zustandsautomaten

- 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 Recheneinheiten (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).

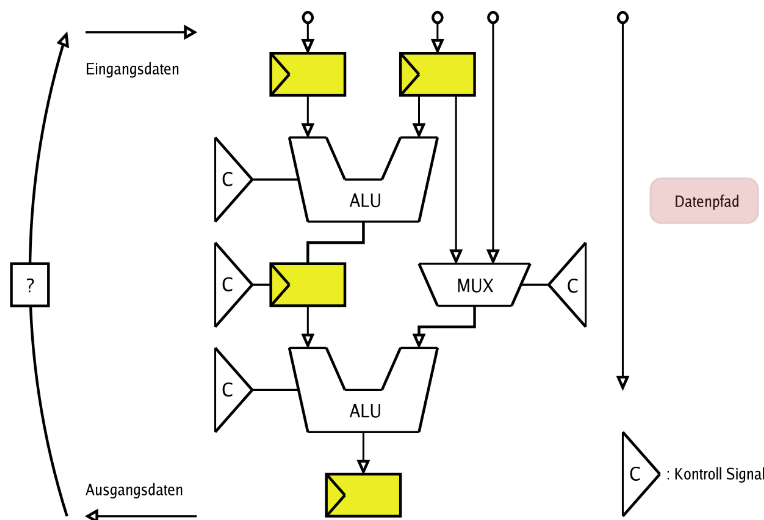


Figure 96. Datenpfade mit RTL

11.2. Endliche Zustandsautomaten

➤ Datenpfade werden mit endlichen Zustandsautomaten gesteuert (FSM: Finite State Machine).

➤ Steuerungseinheit ⇔ FSM

➤ Datenpfadeinheiten können zyklisch wechselnde Datensätze, d.h. Datenströme, bearbeiten.

➤ Der Zustandsautomat muss die einzelnen Operationen im Datenpfad steuern und koordinieren:

1. Datentransport (Multiplexer)

2. Datenspeicherung (Register)
3. Operationen und Daten selektieren (ALU + 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).

Partitionierte Sequenzielle Maschinen

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

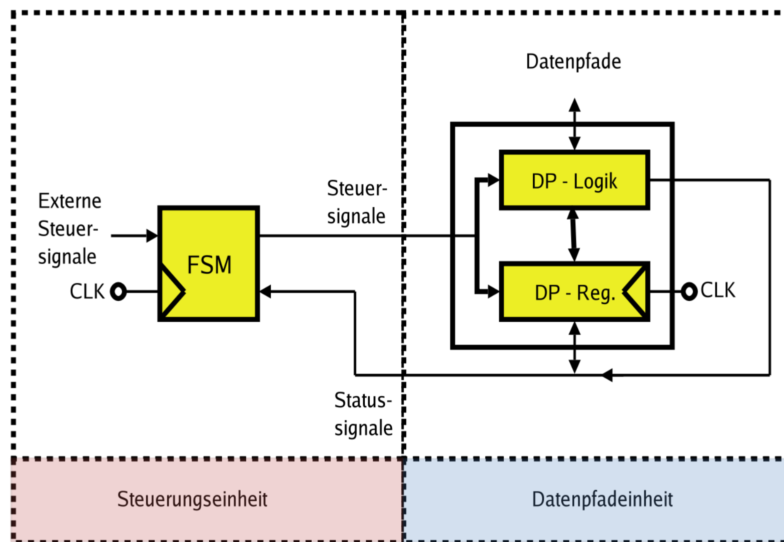


Figure 97. Allgemeine Architektur und Strukturierung einer Sequenziellen Maschine.

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.

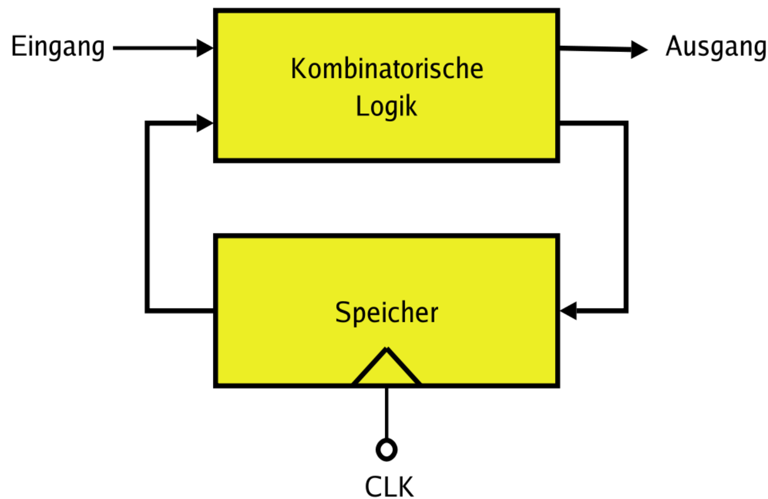


Figure 98. Blockschaltbild eines allgemeinen Automaten (ohne Datenpfad des RTL Systems!)

11.3. Der Moore-Automat

- Der Eingangsvektor $\boldsymbol{E}_{\mathsf{t}} = E_1$ .. E_{M} bezeichnet die Gesamtheit der Eingangsgrößen, den Signalen E_{i}.
- Der Zustandsvektor $\boldsymbol{Z}_{\mathsf{t}} = Z_1$ .. Z_{N} bezeichnet den inneren Zustand des Systems. Bezeichnung innerer Zustände durch Umstand dass Signale $\boldsymbol{Z}$ nach außen nicht direkt sichtbar sind.
- Index t und t+1 bezeichnen aktuellen und nächsten Zustand bzw. Vektorwert.

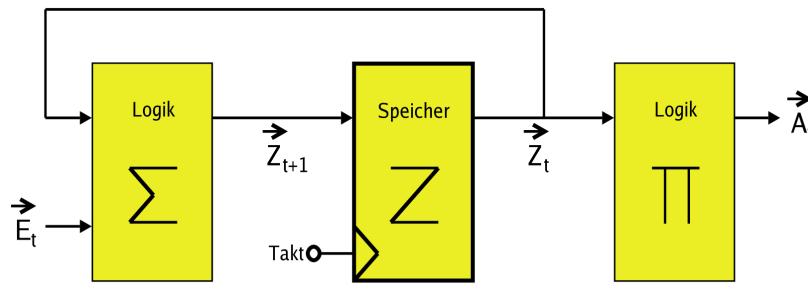


Figure 99. Architektur des Moore-Automaten

Zustandsübergänge

- Die Zustandsübergänge werden mittels Zustandsdiagrammen (State-Transition-Diagram) 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.

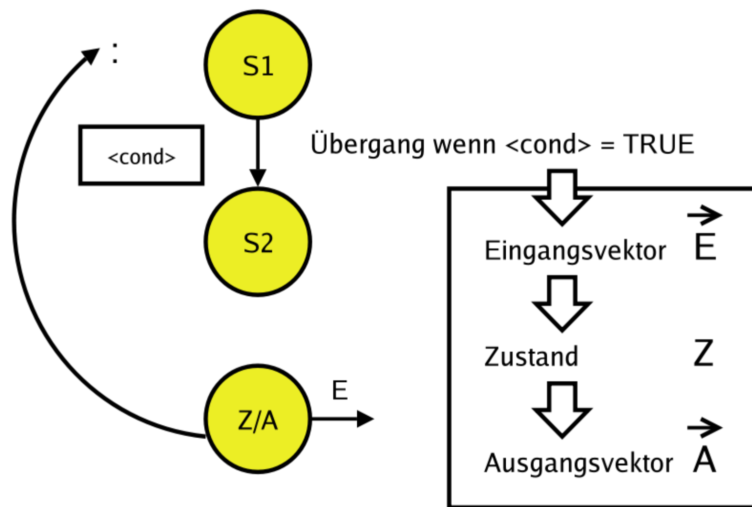


Figure 100. Zustandsdiagramme

Beispiel: Bitfolgenerkennung

- ➤ Es soll eine Bitfolge auf vorgegebene Muster untersucht werden.
- ➤ Problem-Definition
  - ❏ Eingangsvektor E={00,01,10,11}
  - ❏ Ausgangsvektor A={0,1} → Impulsfolge detektiert?
  - ❏ Z(Σ) ← Folge von Eingangsvektoren (01,11,10)
  - ❏ Zustandsvektor Z={Z0,Z1,Z2,Z3}

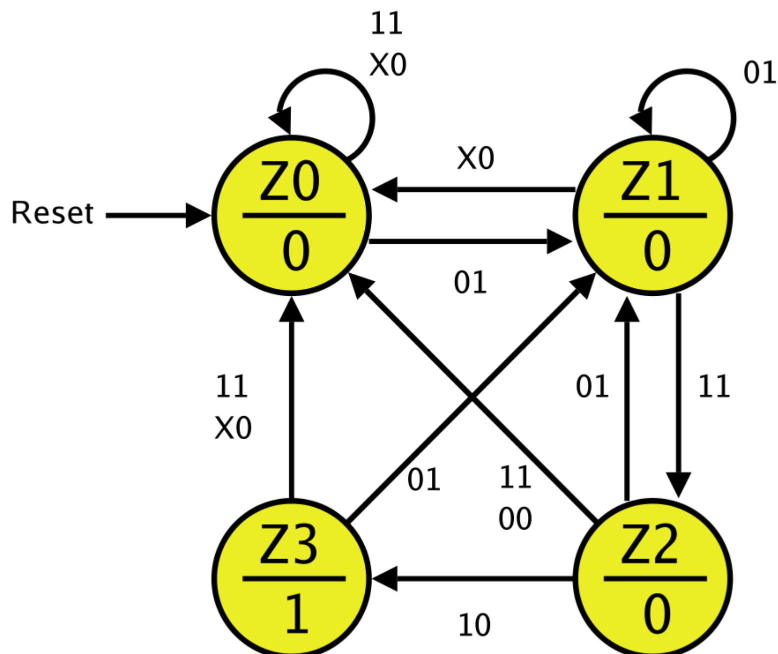


Figure 101. Zustandsdiagramm für Bitfolgenerkennung.

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*

Zustandsspeicher Z

```
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;
```

Ausgangslogik

```
pi: process (state)
begin
  case state is
    when Z3 => A <= 1 ;
    when others => A <= 0 ;
  end case;
end process pi;
```

Schaltnetzwerk

```
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
      else state_next <= Z0; end if;
    when Z2 =>
      if E="10" then state_next <= Z3
      elsif E="01" then state_next
      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;
```

11.4. Hierarchische RT Systeme

- Mehrere RTL Systeme (d.h. FSMs und Datenpfade) können gekoppelt werden
- Eine Sub-FSM implementiert dabei eine prozedurale Funktion die von einer anderen FSM “aufgerufen” (also gestartet) werden kann
- Kommunikation zwischen FSMs via Handshakesignalen:
    - Request `REQ` FSM1 → FSM2
    - Acknowledge `ACK` FSM2 → FSM1
    - Busy `BUSY` FSM2 → FSM1
    - Daten `RX/TX` FSM1 ↔ FSM2
- Wenn FSM1 die andere FSM2 startet wird FSM1 i.A. blockiert (angehalten)

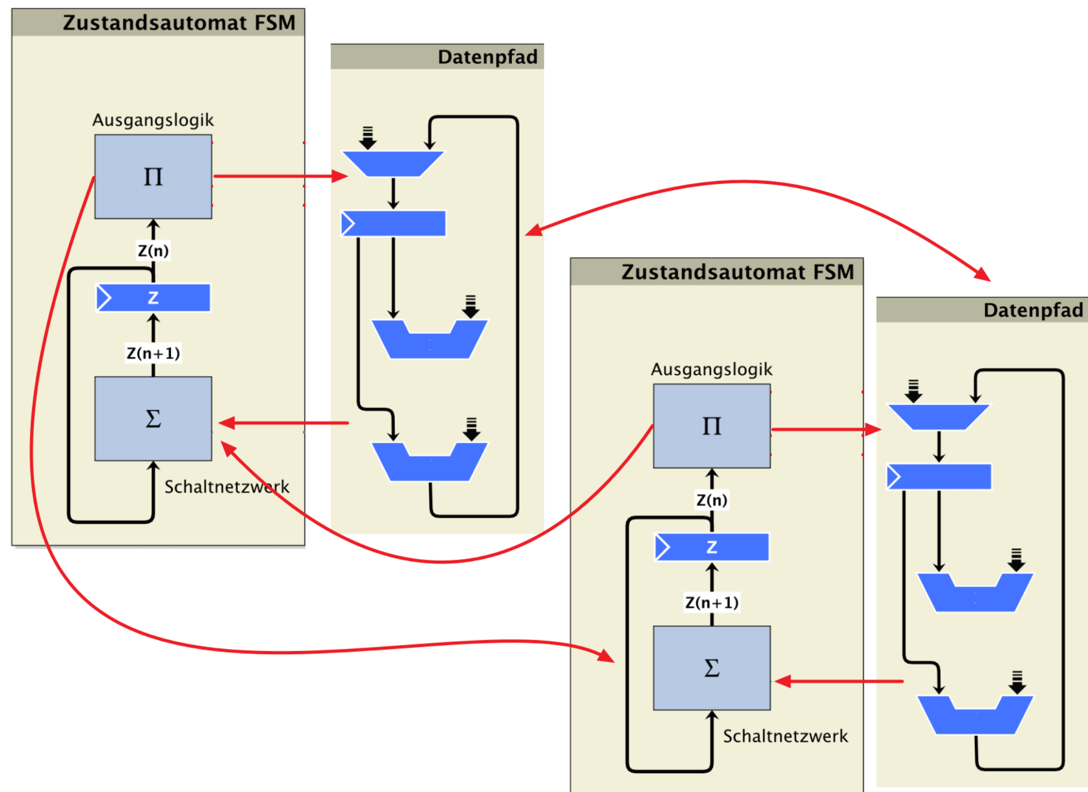


Figure 102. zwei gekoppelte RTL Systeme mit Kommunikation der FSMs und Datenpfade

12. Synthese

Abbildung von sequenziellen Prozessen auf die Register-Transfer Architektur und Digitallogik

12.1. Algorithmen

Frage: Wie kommt der Algorithmus auf den Mikrochip?

Es gibt verschiedene Beschreibungs- und Modellierungsebenen für die Abbildung von Algorithmen auf Digitallogik und der Entwicklung von Digitallogikschaltungen:

Algorithmisches Verhalten (C): Hohe Abstraktion
: Mit Mitteln einer imperativen Programmiersprache findet die Datenverarbeitung mit komplexen Berechnungen und Kontrollanweisungen schrittweise statt.
Komposition mit Anweisungen und Funktionen.

Architektur und Struktur (RTL): Mittlere Abstraktion
: Die Datenverarbeitung wird durch eine Menge von Funktionseinheiten auf Blockebene beschrieben, die miteinander verbunden sind und Daten austauschen (Kommunikation und Transaktion).
Komposition mit Blöcken: Funktionseinheiten können wieder aus weiteren Funktionseinheiten zusammengesetzt sein.

Hardware Verhalten (VHDL) : Niedrige Abstraktion
: Das Verhalten von Digitallogik wird mit Hardware-Beschreibungssprachen modelliert. Dabei wird das Verhalten der Digitallogik oder eines Teils davon beschrieben wie Eingangssignale einer solchen Komponente auf Ausgangssignale abgebildet werden (funktionale Abbildungsvorschrift von Signalen).

Beim Entwurfsprozess findet Partitionierung und Komposition der Datenverarbeitung statt:

- **Algorithmisches Verhalten.** Module, Objekte, Funktionen, Prozesse, Kommunikation, Datentypen, Variablen
- **Architektur und Struktur.** Komponenten, Verbindungen, Kommunikation und Protokolle, Daten- und Kontrollpfade, Zustandsautomaten, Aktivierungs- und Ablaufdiagramme
- **Hardware Verhalten.** Module (Packages), Komponenten, Funktionen, Datentypen, Prozesse, Signale

Die Register-Transfer Ebene (RTL) ist wesentliche Datenverarbeitungsarchitektur, um Algorithmik systematisch auf Digitallogik abbilden zu können. Sie erlaubt die zustandsgesteuerte und schrittweise Berechnung.

- RTL kennt verschiedene Abstraktionsebenen:

    I. Zustandsdiagramme (Verhalten, nur Kontrollfluss)

    II. Zeit-Operationen / Zeit-Fläche Diagramme (Verhalten und Struktur, Daten- und Kontrollfluss)

III. Kombinatorische Schaltungsblöcke und transitorische Register (Struktur, Blöcke und Verbindungen)

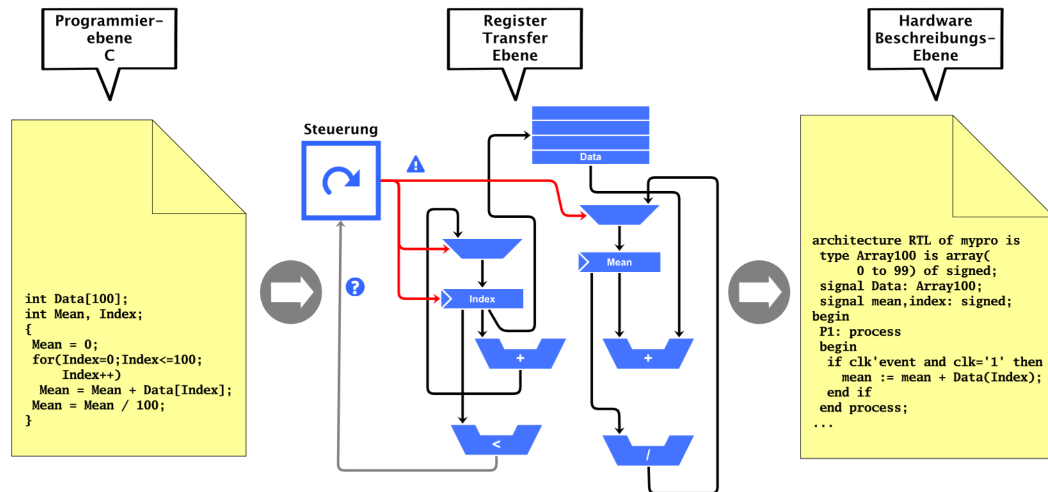


Figure 103. Verschiedene Abstraktionsebenen der Datenverarbeitung

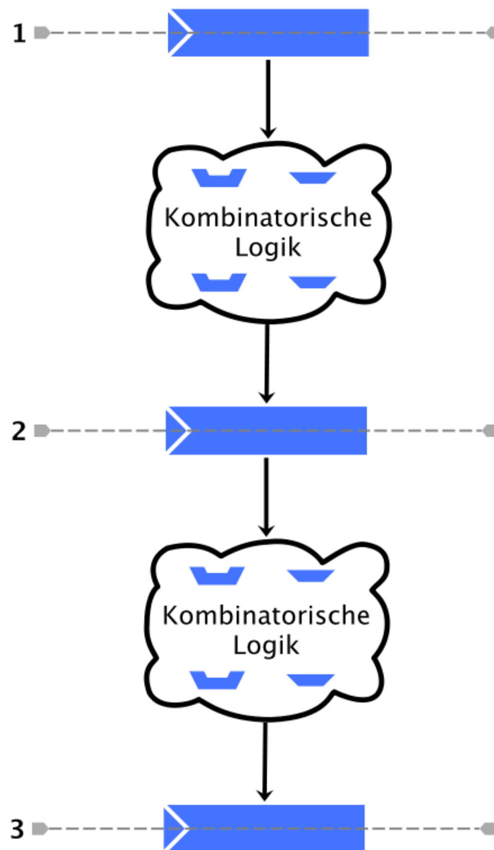
12.2. Register-Transfer Architektur

➤ Die Register-Transfer Architektur ist die Rechnerarchitektur der anwendungsspezifischen Datenverarbeitung und dient zur Implementierung von Algorithmen

Register Register dienen der Speicherung von Zwischen- und Endergebnissen von Berechnungen. Digitallogik: transitorische Logik.

Berechnungen Arithmetische, relationale, und boolesche Berechnungen sind wesentlicher Bestandteil der Datenverarbeitung.

Datenpfad Berechnungen und Register werden im Datenpfad zusammengefasst. Dabei sind die Berechnungseinheiten (Digitallogik: kombinatorische Logik) zwischen Registerebenen eingebettet.



➤ Register-Transfer Architektur bedeutet die Partitionierung der Datenverarbeitung in einen Daten- und Kontrollfluss.

Datenfluss

Der Datenfluss beschreibt den Fluss von Daten durch Verarbeitungs- und Speichereinheiten (Register). Die Verarbeitungseinheiten sind Knoten eines gerichteten Graphs, die Kanten beschrei- ben den Datenfluss und bilden die Datenpfade. Die Verarbeitungseinheiten müssen aktiviert werden.

Kontrollfluss

Der Kontrollfluss beschreibt die temporale schrittweise Verarbeitung von Daten im Datenpfad durch zustandsbasierte selektive Aktivierung von Verarbeitungseinheiten. Der Kontrollfluss kann durch Zustandsübergangsdiagramme beschrieben werden.

Programmfluss

Der Programmfluss setzt sich kombiniert aus Daten- und Kontrollfluss

zusammen.

➤ Programmanweisungen werden Zuständen S_1,S_2,.. zugeordnet.

➤ Einfache Anweisungen (Berechnungen) werden jeweils einem Zustand, komplexe Anweisungen i. A. mehreren Unterzuständen zugeordnet.

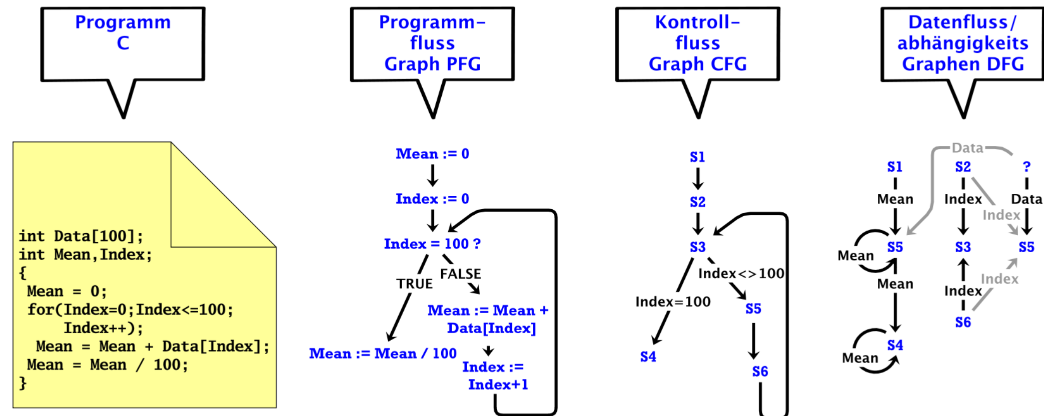


Figure 104. Programm-, Kontroll-, und Datenflussgraphen

Kontrollpfad und Datenpfad kommunizieren über Steuer- und Bedingungsleitungen!

Kontrollpfad

➤ Zustandsbasierte zeitliche Steuerung des Datenpfades

➤ Zustandsautomat

➤ Komponenten:

i. Zustandsregister Z

ii. Eingangs-/Schaltnetzwerk Σ

iii. Ausgangsnetzwerk Π

➤ Verhalten des Zustandsautomaten wird durch Zustandsübergangsdiagramm/graphen beschrieben.

➤ Ein Zustand ist gekennzeichnet durch eine Aktion (Aktivierung von Datenpfadelementen) und Zustandsübergängen

➤ Zustandsübergänge können bedingt (boolesche Bed.) oder unbedingt (immer wahr) sein.

Steuerung: Aktivierung Register (Speicherung), Selektion von Datenpfaden und Operationen

Datenpfad

➤ Führt die eigentliche Verarbeitung von Daten durch.

➤ Komponenten:

i. Funktionseinheiten: arithmetische, relationale, boolesche

ii. Datenpfadselektoren: Multiplexer/Join, Demultiplexer/Split

iii. Register (Daten)

➤ Der Datenpfad kann Parallelität aufweisen (mehrere parallele Teilpfade).

➤ Register trennen kombinatorische Bereiche (die eigentlichen Berechnungen) voneinander (Partitionierung des Datenpfades).

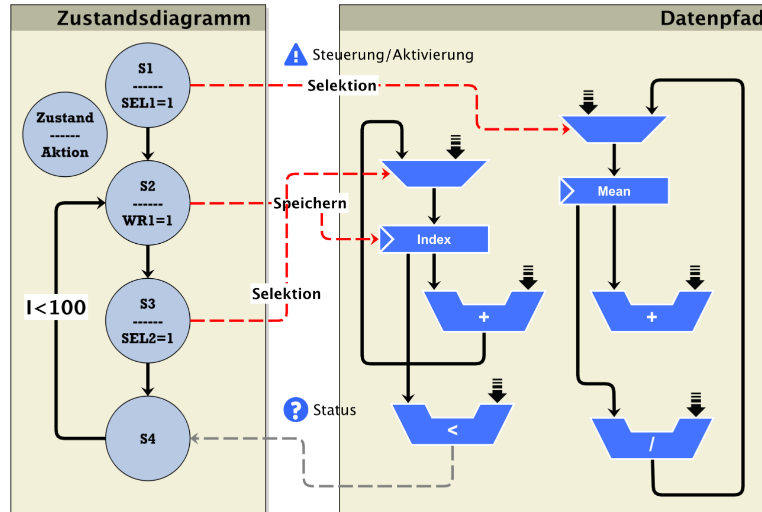


Figure 105. Zustandsbasierte Aktivierung im Datenpfad: (Links) Verhaltensbeschreibung des KP mit Zustandsdiagramm (Rechts) Architektur des Datenpfades

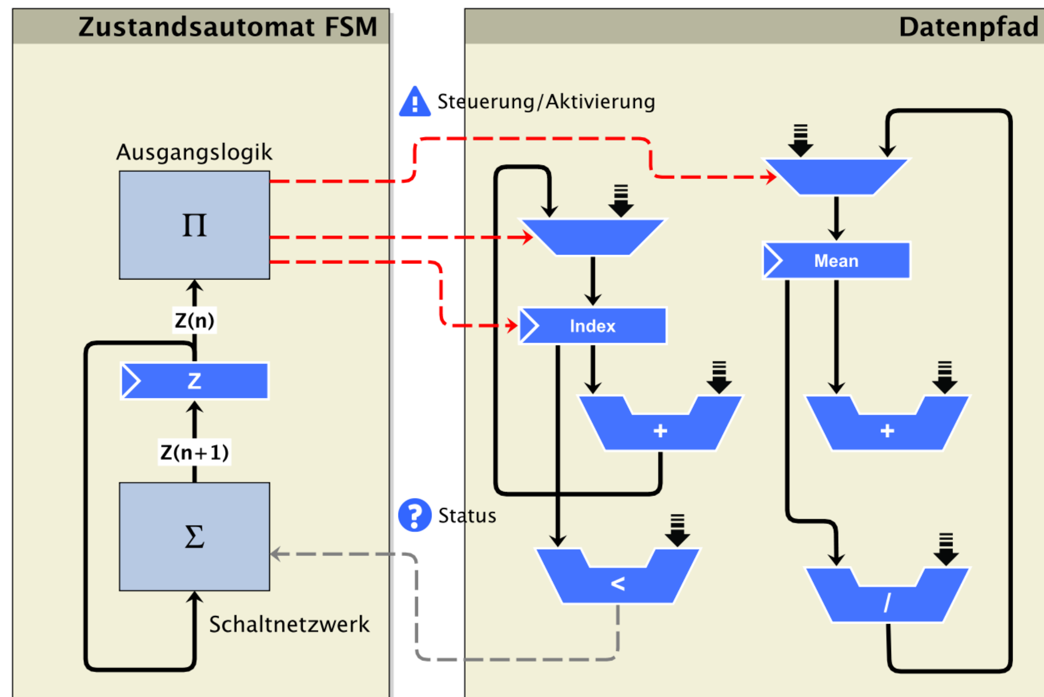


Figure 106. Zustandsautomat (Kontrollpfad) und Kommunikation mit Datenpfad: (Links) Architektur des KP mit Zustandsautomat (Rechts) Architektur des Datenpfades

12.3. RTL Synthese von Digitallogikschaltungen

➤ RTL-Modellierung entspricht einer Spezifikation als gerichteter Graph, deren Knoten Schaltungsblöcke und die Kanten Verbindungsleitungen darstellen.

➤ Die RTL-Spezifikation wird dann in eine Hardwareverhaltensbeschreibung (VHDL,Verilog) umgesetzt:

1. per Hand,
2. automatisch als Synthese-Verfahren.

➤ Obwohl alle Algorithmen mit dieser Methode modelliert und spezifiziert werden können, steigt die Komplexität überlinear (exponentiell) mit der Algorithmus- und Systemkomplexität.

➤ Diese stark ansteigende Entwurfskomplexität resultiert in einem

entsprechenden Anstieg an Entwicklungszeit- und Ressourcen.

➤ Zudem steigt die Fehleranfälligkeit gleichzeitig mit einer reduzierten Möglichkeit, diese Fehler überhaupt feststellen und testen zu können.

➤ Die RTL-Ebene bietet inhärente Parallelität im Datenpfad, die aber vom Entwickler explizit formuliert werden muss.

➤ Parallelität bedeut aber auch Synchronisation - diese muss ebenfalls explizit modelliert werden und ist nicht Bestandteil des Entwurfsmodells.

➤ Datenpfadparallelität (im Gegensatz zur Kontollpfadparallelität) kommt aber **ohne weitere Synchronisation** aus;

  ❏ Nur Datenabhängigkeit und

  ❏ Geteilte Ressourcen sind zu **berücksichtigen**.

RTL-Implementierung aus algorithmischer Beschreibung

➤ Die RTL-Ebene setzt weiterhin explizite Modellierung des zeitlichen Ablaufs und aller Ressourcen voraus → Diskretisierung des Kontrollpfades → Scheduling und Allokation.

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

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) → Datenpfad.

➤ Erzeugung einer RTL bedeutet das Abbilden von Daten- und Kontrollfluss in zwei Dimensionen: Zeit FSM und Fläche Hardware Logik.

Ressourcen-Zuweisung
: Konkrete Belegung von kombinatorischen und transitorischen Elementen

I. Zuordnung von Funktionselementen zu einzelnen Instanzen und Operationen → Datenpfad.

II. Abbildung auf eine Hardware-Verhaltensbeschreibung

➤ starke Wechselwirkung zwischen Scheduling und Ressourcenallokation

➤ RTL-Scheduling- und Allokation kann per Hand, aber auch regel- und modellbasiert mit automatischer High-Level-Synthese erzeugt werden.

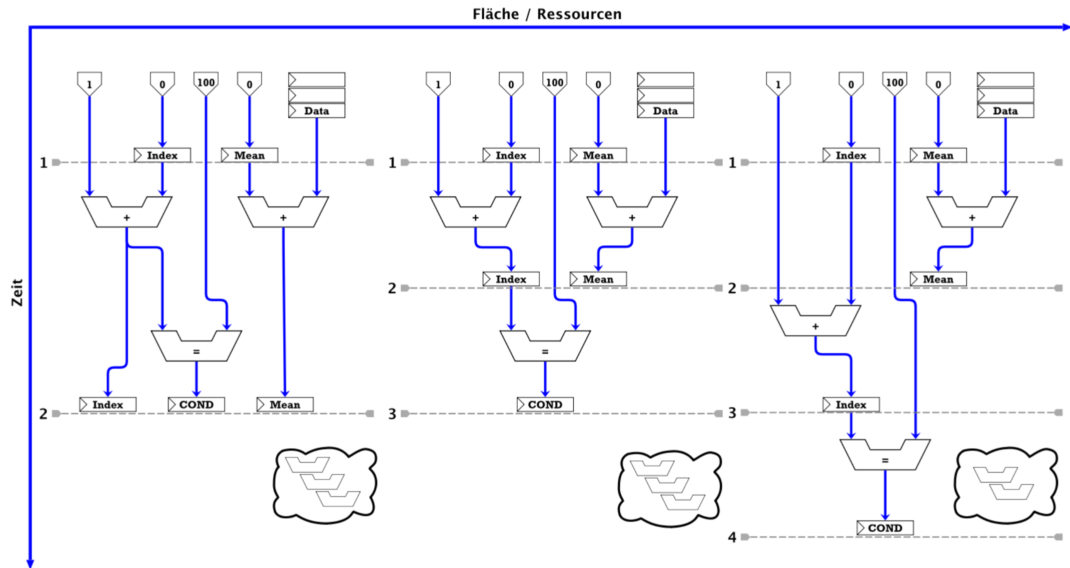


Figure 107. Zeitliche Partitionierung (Scheduling) und Ressourcenallokation mit Zeit-Operationen Diagrammen

Einfachstes Scheduling- und Allokationsmodell: Grundregeln

Regel I

Es gibt nur Register als Datenspeicher mit beliebiger Datenwortbreite.

Regel II

Keine Ressourcenteilung: jedes Register R was definiert wurde wird auch als Hardware-Block ω implementiert.

Regel III

Jede elementare Anweisung κ wird auf einen Zustand φ des Kontrollpfades Γ=(Φ,E) als gerichteter Graph bzw. der Zustandsmenge $\Phi=\{\varphi_1,\varphi_2,..\}$ des RTL-FSM abgebildet.

Regel IV

Der Datenpfad Δ wird in Abhängigkeit der Zustände Φ zerlegt.

Regel V

Keine Ressourcenteilung: jeder funktionale Operator (arithmetisch, relational und Boolesch) π wird mit eigenen Hardware-Blöcken ω implementiert.

Regel VI

Komplexe Anweisungen K (z.B. Schleifen) müssen in eine Menge $\{\kappa\}$ von elementaren Anweisungen mit Transformationsregeln zerlegt werden.

Regel VII

Zu jedem Zustand φ gehört ein Folgezustand $\varphi+$, der bedingt oder unbedingt definiert sein kann.

μ**RTL**

- μRTL ist eine symbolische Beschreibungssprache mit Mikrooperationen die als Zwischenrepräsentation dient (Programmiersprache $\rightarrow$ μRTL $\rightarrow$ Γ, Δ)
- RTL Beschreibung ist eine lineare Liste von Anweisungen - kein Graph!

12.4. μRTL Synthese von sequenziellen Prozessen

Operation	**Beschreibung**
$r \leftarrow \epsilon$	Register Datentransfer und Berechnung
i_1 ; i_2 ; ..	In mehreren Zeitschritten sequenziell ausgeführte Anweisungen (; leitet nächsten Zeitschritt ein)
i_1 , i_2 , .. ;	In einem Zeitschritt parallel ausgeführte Anweisungen (Paralleler Block)
branch s	Unbedingte Verzweigung zu einer anderen Anweisung *s*
branch cond:s	Bedingte Verzweigung zu einer anderen Anweisung *s* wenn *cond=true*
s:	Zuordnung eines Labels (Zustand) zu einer Anweisung

Programmanweisung (C)	**μRTL**
$x = \epsilon_1;$	s_i: $x \leftarrow \epsilon_1;$
$y = \epsilon_2;$	$s_\text{i+1}$: $y \leftarrow \epsilon_2;$
$x = \epsilon_1;$	s_i: $x \leftarrow \epsilon_1,$
$y = \epsilon_2;$	$y \leftarrow \epsilon_2;$

Table 1. Abbildung von sequenziellen Programmanweisungen auf μRTL

C

```
int Data[100];
int Mean, Index;
{
 Mean = 0;
 for(Index = 0; Index < 100; Index++)
  Mean = Mean + Data[Index];
 Mean = Mean / 100;
}
```

µRTL

```
s1: Index ← 0, Mean ← 0;
s2: branch (Index=100):s6;
s3: Mean ← Mean + Data[Index];
s4: Index ← Index + 1;
s5: branch s2;
s6: Mean ← Mean / 100;
```

Figure 108. μRTL Code abgeleitet aus C Programmbeispiel

- Ein sequenzieller Prozess P besteht aus einer Reihe von Anweisungen {i1,i2,..} die der Reihe nach ausgeführt werden.
- Der Prozess kann Register R={r_1,r_2,..} als Datenspeicher nutzen.
- Es gibt elementare Anweisungen die Berechnungen durchführen und Daten speichern (Registertransfer im Datenpfad):

```
r := expression(r1,r2,...,+,-,*,...)s
```

- Es gibt komplexe Anweisungen die den Programmfluss steuern, wie z.B. bedingte Verzweigungen (*if*) und Schleifen (*while*, *for*).
- Komplexe Anweisungen müssen in elementare Anweisungen und einfache Kontrollflussverzweigungen (*branch*) zerlegt werden.
- Elementaren Anweisungen wird ein Zustand s_i zugeordnet, komplexe spalten in multiple Zustände $s_\text{i,j}$ auf.

Programmanweisung (C)	**μRTL**

if ($cond_1$)
   I_1;
else if ($cond_2$)
   I_2;
-next-
	$s_{i,1}$: *branch* $cond_1$: $s_{i,4}$;
	$s_{i,2}$: *branch* $cond_2$: $s_{i,6}$;
	...
	$s_{i,3}$: *branch* $s_{i,x}$;
	$s_{i,4}$: I_1;
	$s_{i,5}$: *branch* $s_{i,x}$;
	$s_{i,6}$: I_2;
	...
	$s_{i,x}$: -next-

Table 2. Abbildung von sequenziellen Programmanweisungen auf μRTL (2): Bedingte Verzweigung

Programmanweisung (C)	**μRTL**

switch (e) {
   case v_1: I_1; break;
   case v_2: I_2; break;
   default: I_0; break;
}
	$s_{i,1}$: branch (e=v_1):$s_{i,c1}$;
	$s_{i,2}$: branch (e=v_2):$s_{i,c2}$;
	..
	$s_{i,n0}$: I_0;
	$s_{i,n1}$: branch $s_{i,x}$;
	$s_{i,c1}$: I_1;
	$s_{i,c11}$: branch $s_{i,x}$;
	$s_{i,c2}$: I_2;
	$s_{i,c21}$: branch $s_{i,x}$;
	..
	$s_{i,x}$: -next-

Table 3. Abbildung von sequenziellen Programmanweisungen auf μRTL (3): Mehrfachauswahl

Programmanweisung (C)	**μRTL**
while (*cond*) I_1; -next-	$s_{i,1}$: *branch not cond*: $s_{i,x}$; $s_{i,2}$: I_1; $s_{i,3}$: *branch* $s_{i,1}$; Optm.: $s_{i,3}$: *branch cond*:$s_{i,2}$; $s_{i,x}$: -next-
for (i=*start*; $i \leq$*start*+*count*;i++) I_1; -next-	$s_{i,1}$: $i \leftarrow$ *start*; $s_{i,2}$: *branch* ($i \geq$*start*+*count*): $s_{i,x}$; $s_{i,3}$: I_1; $s_{i,4}$: $i \rightarrow i + 1$; $s_{i,5}$: *branch* $s_{i,2}$; Optm.: $s_{i,5}$: *branch* (i<*start*+*count*):$s_{i,3}$; $s_{i,x}$: -next-

Table 4. Abbildung von sequenziellen Programmanweisungen auf μRTL (4): Schleifen

Hardware-Modell

➤ Der Datenpfad Δ wird aufgespalten in einen reinen funktionalen Berechnungsteil (Abbildung mit kombinatorischer Logik, Ausgangslogik Π des FSM) und einen datenspeichernden Berechnungsteil (Abbildung mit transitorischer Logik/Registern).

➤ Der Kontrollpfad Γ wird unterteilt in ein kombinatorisches Zustandsübergangsnetzwerk Σ, welches den Folgezustand des Zustandsautomaten berechnet, und ein Zustandsregister Z.

➤ Die Ausgangslogik Π ist bereits im funktionalen Berechnungsteils des Datenpfades enthalten.

➤ Jeder Teil wird auf Hardware-Beschreibungsebene mit einem eigenen Hardware Prozess H implementiert:

- ❏ `process state_transition`
- ❏ `process control_path`
- ❏ `process data_path`
- ❏ `process data_trans`

VHDL Partitionierung der Datenverarbeitung eines sequenziellen Prozesses P mit vier Hardware Prozessen H.

```
architecture RTL of P is
  type states is (
    S_i1, S_i2_1,S_i2_2,..);
  signal s,s_next: states;
  signal r,d,..: <datatype>;
begin
  state_transition: process()
    if clk'event then
      s ⇐ s_next
    end if
  end process;
  control_path: process()
    case s is
      when S_i1 ⇒ s_next ⇐ ...
      when S_i2 ⇒ s_next ⇐ ...
      ..
    end case;
  end process;
```

```
  data_path: process()
    case s is
      when S_i1 ⇒ d ⇐ ϵ
      when S_i2_1 ⇒ null
      ..
    end case;
  end process;

  data_trans: process()
    if clk'event then
      case s is
        when S_i1 ⇒ null;
        when S_i2_1 ⇒ r ⇐ ϵ
        ..
      end case;
    end if;
  end process;
end architecture;
```

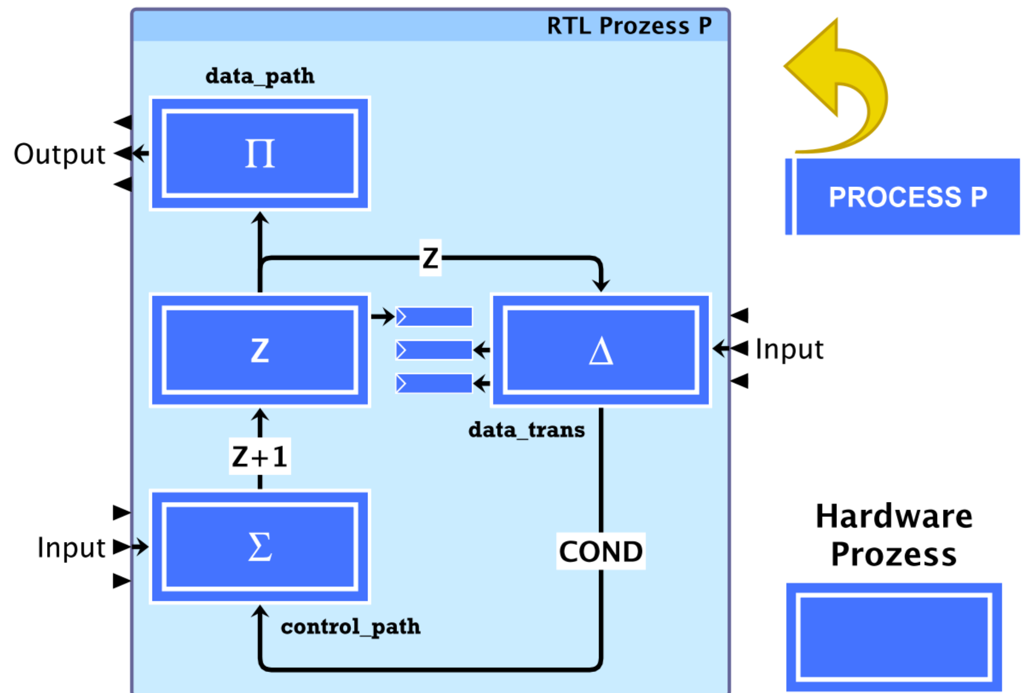


Figure 109. Vier-Prozess RTL Hardware-Implementierung eines sequenziellen Prozesses

Berechnungen im Datenpfad mit Register Datentransfer

- Berechnungen werden im Datenpfad direkt abgebildet und mit kombinatorischer Logik durchgeführt.
- Ergebnis von Berechnungen werden Registern zugewiesen.
- Daher Berechnungen im transitorischen Datenpfadprozess *data_trans*.
- Möglichkeit der Bindung von Datenpfadoperationen in einem Zeitabschritt (auch bei Mehrfachauswahl und ausdrucksevaluierung), sofern Datenabhängigkeit dieses zulässt.
- Wichtig: Zeitmodell einer transitorischen Datenspeicherung (Latenz):
  - Rechte Seite einer Signalzuweisung evaluiert mit alten Wert eines Registers!

❏ Erst nach Evaluierung der rechten Seite der Zuweisung findet Datentransfer an Register statt!

Algorithm 5. (*Parallel Berechnung in Datenpfad*)

```
-- μRTL --
s1 : r1 ← r2 + r3;
s2 : r1 ← r2 + r3, r2 ← r1 + r3;
-- VHDL --
signal r1, r2.r3 : < datatype >;
...
data_trans : process()
  if clk'event then
    case s is
      when s1 ⇒
        r1 ⇐ r2 + r3;
      when s2 ⇒
        r1 ⇐ r2 + r3; r2 ⇐ r1 + r3;
    ..
    end case;
  end if;
end process;
```

Bedingte Verzweigungen im Kontrollfluss

➤ Verzweigungen im Kontrollfluss werden durch Auswertung von Ausdrücken im Datenpfad ausgeführt (ausgenommen unbedingte Verzweigungen).

➤ Die relationalen oder booleschen Ausdrücke können entweder im kombinatorischen Datenpfadprozess *data_path* oder direkt im Übergangsnetzwerkprozess *control_path* eingebettet werden (benötigt keine weiteren Zwischensignale).

Algorithm 6. (*Bedingte Verzweigungen im Kontrollfluss*)

```
-- μRTL --
s1 : branch (r2 = r3) : s3;
s2 : −next−
-- VHDL --
...
signal r1, r2.r3 : < datatype >;
...
control_path : process()
  case s is
    when s1 ⇒
     if r2 = r3 then s_next ⇐ s3;
     else s_next ⇐ s2; end if;
    ..
  end case;
end process;
```