

Einführung in die Technische Informatik Skript

In der Hoffnung, dass es was nützt ...

Christian Kroh

1. Juli 2014, Dresden

Inhaltsverzeichnis

1. Allgemeine Einführung	5
1.1. Qualifikationsziele	5
1.2. Literatur	5
 I. VLSI-Systementwurf	 6
1. Einführung	7
1.1. Themenschwerpunkte	7
1.2. Inhalte der Lehrveranstaltung	8
1.2.1. Inhalte der Vorlesung	8
1.2.2. Inhalte des Praktikums	8
1.3. Klassifikation von ICs	8
1.3.1. Zwei Sichten	8
1.3.2. Anwenderprogrammierbare IC	10
1.3.3. Hardwareprogrammierung	10
1.3.4. Programmiertechnologien	10
1.3.5. Klassifikation Anwenderprogrammierbare IC	13
 2. Schaltkreisentwurf	 14
2.1. Abstraktionsebenen und Sichten	14
2.1.1. Abstraktionsebenen	14
2.1.2. Sichten	14
2.1.3. Y-Diagramm nach Gajski	15
2.2. Entwurfsablauf	16
2.3. Entwurfsstile	17
2.3.1. Full-Custom Entwurf	18
2.3.2. Standardzellenentwurf	19
2.3.3. Maskenprogrammierbare Gate Arrays	20
2.3.4. Anwenderprogrammierbare IC	21
2.4. Entwurfswerkzeuge	24
 3. Automaten	 25
3.1. Automatendarstellung	25
3.1.1. Betrachtungsweisen	25
3.1.2. Automatengraphen	25
3.1.3. SM-Charts	27
3.1.4. GRAFCET & SFC	28
3.2. Automatenkopplung	31
3.2.1. Synchrone Kopplung	32
3.2.2. Asynchrone Kopplung	36
3.3. Initialisierung	38
3.3.1. Reset vs. Power-Up	38
3.3.2. Synchrones Reset	39
3.3.3. Asynchrones Reset	39
3.3.4. Coding Guidelines für Reset	40
 4. Hardwarebeschreibungssprachen - Hardware Description Language (HDL)	 41
4.1. Allgemein	41
4.2. VHDL	41
4.2.1. Geschichte	41
4.2.2. Abstraktionsebenen	41
4.2.3. Grundsätze	42

4.3. Verilog	42
5. Field-programmable Gate-Array (FPGA)	43
5.1. Architektur	43
5.2. Funktionsblöcke	43
5.3. I/O-Zellen	43
5.4. Verdrahtung	43
5.4.1. Topologie	43
5.4.2. Technologie	43
5.5. Speicherelemente	43
5.5.1. LUT-RAM	43
5.5.2. Block-RAM	43
5.6. IP-Cores	43
5.7. Konfigurierbarkeit	43
5.8. Konfigurationsmodi	43
6. Modellierung	44
7. Simulation	45
8. Zeitverhalten	46
9. Test	47
10. Hochgeschwindigkeit	48
11. Verlustleistung	49
 II. Entwurf eingebetteter Systeme	 50
 III. Parallelverarbeitung	 51
 IV. Appendix	 52
1. VLSI-Systementwurf Praktikum	53
1.1. Kurze Beschreibung des Terasic DE0 Board	53
1.2. Aufgabe 1 - Binär-Dekoder	55
1.2.1. Entwurf	55
1.2.2. Auswertung	55
1.3. Aufgabe 2 - Hamming-Distanz	56
1.3.1. Entwurf	56
1.3.2. Auswertung	56
1.4. Aufgabe 3 - Modulo-n-Zähler	57
1.4.1. Entwurf	57
1.4.2. Auswertung	58
1.5. Aufgabe 4 - Entprell-Automat	59
1.5.1. Entwurf	59
1.5.2. Auswertung	60
1.6. Aufgabe 5 - HALLO-Anzeige	61
1.6.1. Entwurf	61
1.6.2. Auswertung	62
1.7. Aufgabe 6 - Stoppuhr	62
1.7.1. Entwurf	63
1.7.2. Auswertung	65
1.8. Anhang	65
1.8.1. 01-Aufgabe Code	65
1.8.2. 02-Aufgabe Code	66
1.8.3. 03-Aufgabe Code	67
1.8.4. 04-Aufgabe Code	68

1.8.5.	05-Aufgabe Code	71
1.8.6.	06-Aufgabe Code	74
2.	Entwurf eingebetteter Systeme: Schaltkreisvalidation	80
2.1.	Programm	80
2.1.1.	Entwurf	80
2.1.2.	Äquivalenzprüfung durch Simulation	81
2.1.3.	Äquivalenzprüfung durch SAT-Solver	82
2.2.	Versuche	83
2.2.1.	1. Versuch	83
2.2.2.	2. Versuch	83
2.2.3.	3. Versuch	83
2.2.4.	4. Versuch	84
2.2.5.	5. Versuch	84
2.2.6.	6. Versuch	84
2.3.	Anhang	85
2.3.1.	Circuit	85
2.3.2.	Simulator	86
2.3.3.	Solver	86
2.3.4.	Parsers	87
2.3.5.	Parser	87
2.3.6.	BENCH	88
2.3.7.	Gates	88
2.3.8.	Gate	89
2.3.9.	Input	90
2.3.10.	DFF	90
2.3.11.	Output	91

1. Allgemeine Einführung

1.1. Qualifikationsziele

Die Studierenden kennen Systemarchitekturen und Modellierungsparadigmen von VLSI-Systemen.

Sie sind in der Lage Beschreibungen von Hardware-Systemen durch Simulation zu verifizieren und mithilfe typischer Werkzeuge in reale Schaltungen umzuwandeln.

Sie können den Ressourcenbedarf, das Zeitverhalten und die Verlustleistung abschätzen oder evaluieren und daraus Entwurfsentscheidungen ableiten.

1.2. Literatur

- F. Kesel und R. Bartholomä: Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs, Oldenbourg Wissenschaftsverlag, ISBN 978-3-486-58976-4.
- H.-D. Wuttke und K. Henke: Schaltsysteme – Eine automatenorientierte Einführung, Pearson Studium, ISBN 3-8273-7035-3.
- H. M. Lipp und J. Becker: Grundlagen der Digitaltechnik, Oldenbourg Wissenschaftsverlag, ISBN 978-3-486-59747-9.

Teil I.

VLSI-Systementwurf

1. Einführung

1.1. Themenschwerpunkte

Verarbeitungsleistung

Im Vordergrund stehen:

- Schnelle Verarbeitung auch einzelner Bits
- Parallelität auf:
 - Bitebene
 - Befehlseben
 - Threadebene
 - Prozess- und Anwendungsebene
- dynamische Rekonfiguration

⇒ Erfüllung der gegebenen Anforderungen

Systemintegration

Im Vordergrund stehen:

- Mehrprozessorsysteme, Mehrkern-, Vielkernprozessorsysteme
- Mehr-Chip- / Einzel-Chip-Lösungen (System-on-a-Chip)
- Parallele Entwicklung von HW und SW (HW-/SW-Codesign)
- System-Prototyping (FPGA-Entwurf)

⇒ Kosteneinsparung, Entwicklungszeiteinsparung (Time-to-Market)

Verlustleistung

Im Vordergrund stehen:

- Verlustleistung im Standby (Akkubetrieb)
- Maximales Abwärmebudget

Kenngrößen:

- Statische und dynamische Verlustleistung
- MIPS pro Watt

⇒ Sowohl für eingebettete Systeme als auch für Server

Korrektheit

Aspekte:

- Verifikation eines Schaltkreises, simulativ / formal
- Profiling und Debugging unter Echtzeitbedingungen (Trace)

⇒ Fehlerfreier Erstentwurf

Fehlertoleranz

Toleranz gegenüber:

- Permanenten Fehlern (zeitunabhängig nach erstem Auftreten)
- Intermitierenden Fehlern (nur unter bestimmten Betriebsbedingungen)
- Transienten Fehlern (aufgrund statischer Störungen)

Fehlererkennung und -korrektur:

- Autonom durch Hardwarearchitektur

- Aus Kombination von HW und SW
- ⇒ Insbesondere wichtig für Sicherheitskritische, hochverfügbare und langlebige zuverlässige Systeme
- ⇒ Steigende Signifikanz mit abnehmenden Strukturgrößen (Integrationsgrad) sowie steigender Transistoranzahl (Moore's Law)

1.2. Inhalte der Lehrveranstaltung

1.2.1. Inhalte der Vorlesung

1. Klassifikation von Schaltkreisen
2. Grundlagen des Schaltkreisentwurfs
3. Automatendarstellung, -kopplung, -vereinfachung
4. Hardwarebeschreibungssprachen
5. Programmierbare Schaltkreise, insbesondere FPGAs - Teil 1
6. Programmierbare Schaltkreise, insbesondere FPGAs - Teil 2
7. Modellierung und Simulation
8. Zeitverhalten und Test
9. Hochgeschwindigkeit und Verlustleistung
10. Anwendungsbeispiele

1.2.2. Inhalte des Praktikums

1. Altera Quartus-Toolchain & Praktikumsboard DE0 mit Cyclone-3
2. Schaltnetze und Schaltwerke
3. Modularisierung
4. "Komplexe" Anwendung: Stoppuhr

1.3. Klassifikation von ICs

1.3.1. Zwei Sichten

Klassifizierung von integrierten Schaltkreisen (ICs) in

- Standardschaltkreis (Standard-IC) und
- applikationspezifische Schaltkreise (Application-Specific IC, **ASIC**)

unter zwei Gesichtspunkten möglich:

- **Herstellungssicht**
- **Entwurfssicht**

Herstellungssicht:

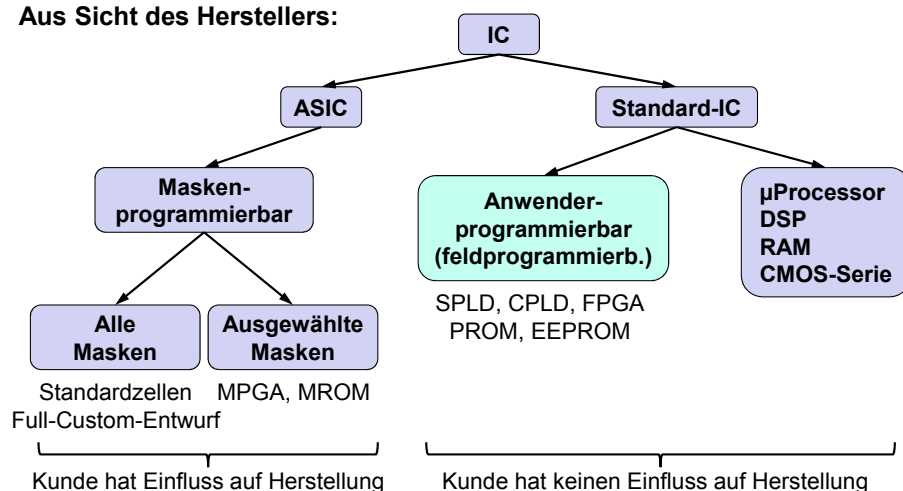
- Standard-IC = große Stückzahl für viele Kunden
- ASIC = für eine/n Kunden/Applikation speziell entwickelter und gefertigter IC mit zugeschnittener Funktionalität

Entwurfssicht:

- Standard-IC = (Hardware-)Funktionalität kann nicht vom Anwender beeinflusst werden
- ASIC = vom Anwender selbst entwickelte (Hardware-)Funktionalität

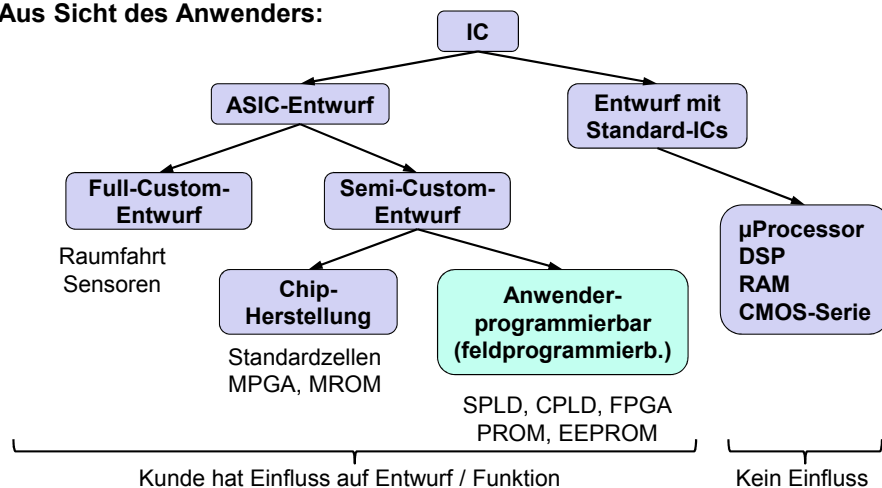
Klassifikation nach Herstellungssicht

Aus Sicht des Herstellers:



Klassifikation nach Entwurfssicht

Aus Sicht des Anwenders:



Abgrenzung der Entwurfsalternativen

Entwurfs-alternative	Transistor-layout	Gatter-position	Verdrah-tung	Funktion
Full-Custom	+	+	+	+
Standardzellen	(+)	+	+	+
MPGA, MROM	-	(+)	+	+
PLD, FPGA, PROM	-	-	(+)*	+
Einzel-ICs	-	-	-	+

* = Einfluss nur indirekt durch Programmierung

1.3.2. Anwenderprogrammierbare IC

Merkmale:

- Field-Programmable \Leftrightarrow feldprogrammierbar
- Vor Ort (im Feld) vom Anwender programmierbar
- Hardware ist fix. Funktionalität kann aber mittels spezieller Konfiguration “programmiert” werden

Anwendung:

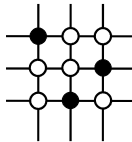
- Anwendungsspezifische IC bei kleinen und mittleren Stückzahlen
- Mehrfach neu programmierbar zwecks Optimierung und Fehlerbehebung, auch während des praktischen Einsatzes
- Einfache Integration eines ganzen Systems auf einem Chip
- Prototyping, HW-/SW-Codesign

1.3.3. Hardwareprogrammierung

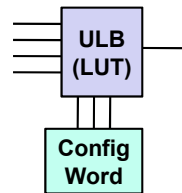
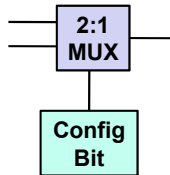
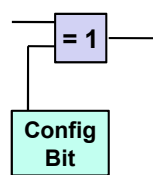
Hardwareprogrammierung

Programmiert (oder auch konfiguriert) werden können:

- Verdrahtung / Verbindung



- Funktionen



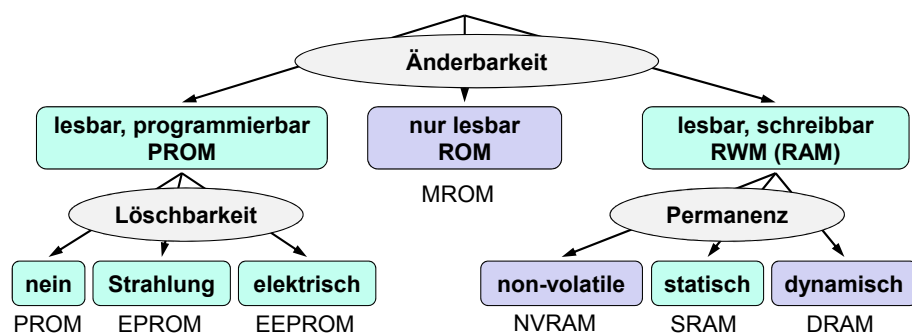
- Speicher: PROM, EPROM, EEPROM (Flash)

1.3.4. Programmiertechnologien

Programmiertechnologien (1)

Klassifikation hinsichtlich Änderbarkeit:

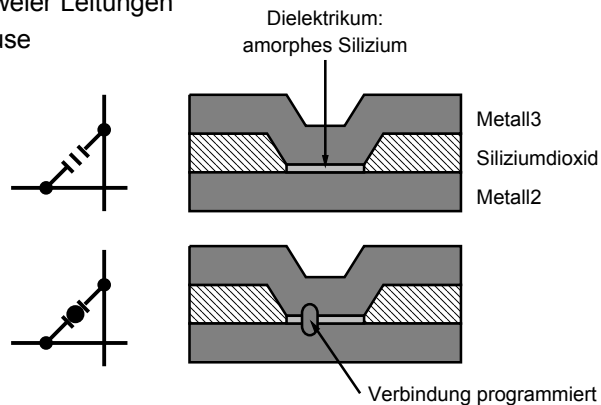
analog Halbleiterspeicher



Programmiertechnologien (2)

Antifuse:

- Programmierung elektrisch, aber nur einmalig
- Schalter oder Verbindung zweier Leitungen
- Beispiel: Metall-Metall-Antifuse

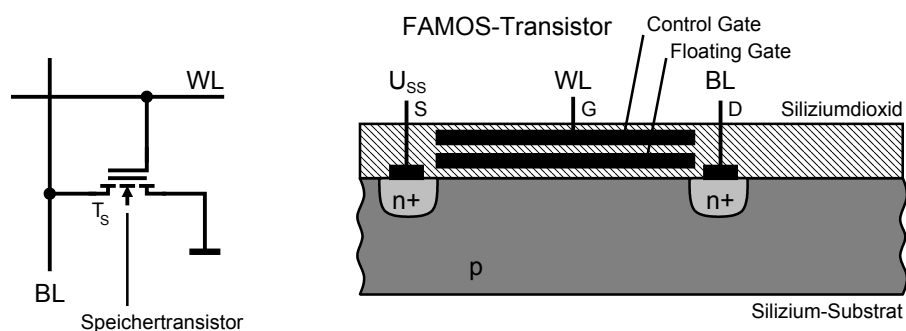


el u. Bartholomä: Entwurf von digitalen Schaltungen
Systemen mit HDLs und FPGAs

Programmiertechnologien (3)

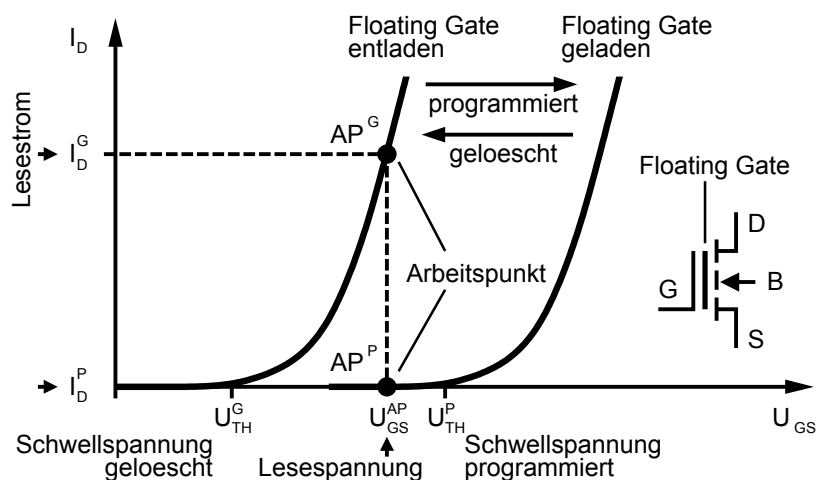
EPROM FAMOS-Transistor: (Floating-Gate Avalanche-injection MOS)

- Ladungsspeicherung (Elektronen) auf dem Floating-Gate.
- Programmierung elektrisch, Löschen durch UV-Bestrahlung (mehrmalig).



Schnittdarstellung planare EPROM-Speicherzelle

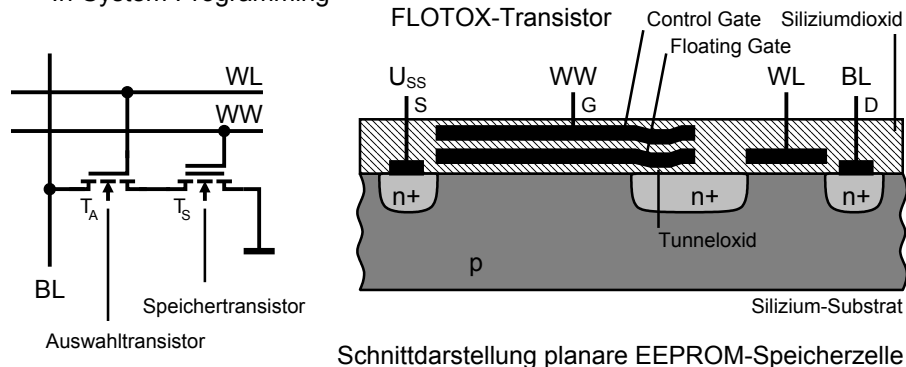
Kennlinie des FAMOS-Transistors:



Programmiertechnologien (4)

EEPROM mit FLOTOX-Transistor: (Floating-Gate Tunneling Oxide)

- Ladungsspeicherung (Elektronen) auf dem Floating-Gate.
- Programmierung elektrisch, Löschen elektrisch über WW (Word Write).
- In-System Programming



Programmiertechnologien (5)

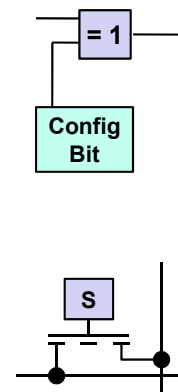
Flash-EEPROM:

- EEPROM mit 1-Transistor-Speicherzelle (FLOTOX-Transistor)
- Nur blockweises Löschen
- NAND-Flash für hohe Speicherdichten
- NOR-Flash für Speicher mit geringen Zugriffszeiten
- Problem: Haltbarkeit, z.B. Intel ETOX-Zelle:
 - Endurance von 10^5 - 10^6 Lösch-/Programmierzyklen.
 - Data Retention von ca. 10 Jahren.

Programmiertechnologien (6)

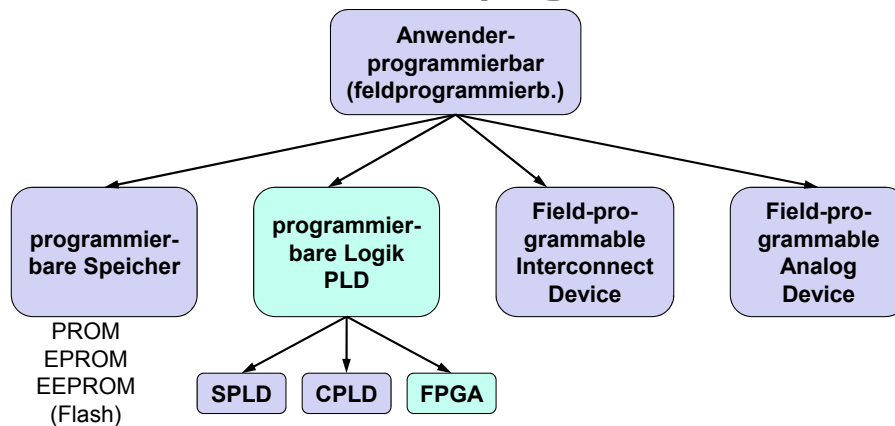
SRAM:

- 4 MOSFET pro Speicherzelle
- 1 Konfigurationsbit pro Speicherzelle zur
 - Funktionsauswahl
 - Ansteuerung eines Pass-Transistors
- Beliebig oft programmierbar, aber Verlust der Information bei Ausfall der Betriebsspannung
- Im Betrieb einfach programmierbar
 - Schreib-/Lesespeicher
 - Dynamische Rekonfiguration



1.3.5. Klassifikation Anwenderprogrammierbare IC

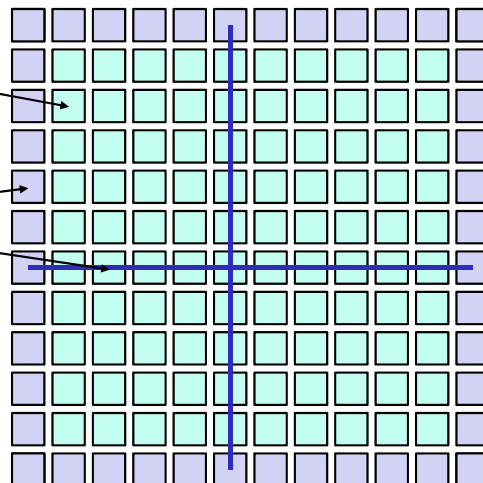
Klassifikation Anwenderprogrammierbare IC



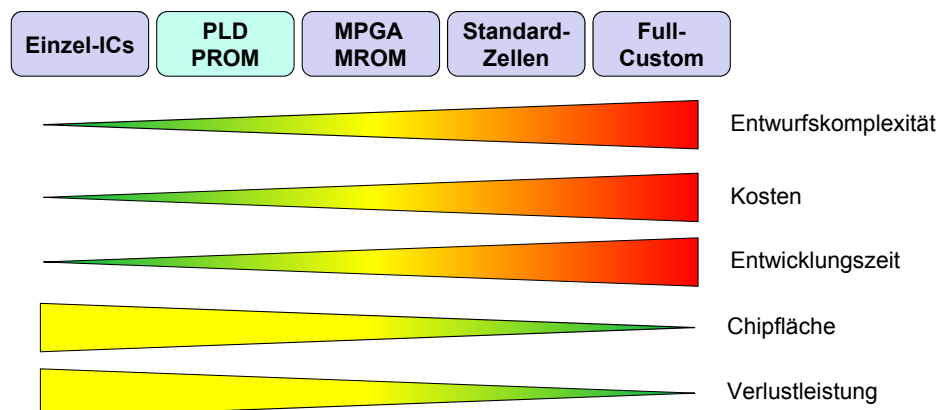
Beispiel: FPGA-Architektur

Grundlegende Bestandteile:

- Funktionsblöcke (FB):
 - angeordnet als Matrix,
 - Multiplexer- oder LUT-basiert.
- I/O-Zellen als spezielle FB.
- Allgemeine lokale Verdrahtung, sowie globale und dedizierte Signalleitungen.
- Spezielle Hard-Makros.



Gegenüberstellung der Entwurfsoptionen



PLD = Programmierbare Logik (SPLD, CPLD, FPGA)

PROM = Programmierbare Speicher (PROM, EPROM, EEPROM)

2. Schaltkreisentwurf

2.1. Abstraktionsebenen und Sichten

Ebenen des Entwurfs:

- Charakterisierung des jeweiligen Detaillierungsgrades der Beschreibung des Entwurfsgrades
- Abstraktionsgrad von der eigentlichen physikalischen Realisierung
- Abstraktionsniveaus, Hierarchien

Sichten des Entwurfs:

- Betrachtung des Entwurfsgegenstandes aus verschiedenen Richtungen
- Sicht = Eigenschaften die den Entwurfsgegenstand Charakterisieren
- Alle Sichten auf jeder Entwurfsebene \Rightarrow Y-Diagramm / x-Diagramm

2.1.1. Abstraktionsebenen

Systemebene (system level): Systemkonzept des Entwurfsgegenstandes

Algorithmische Ebene (algorithm level): Algorithmische Beschreibung des Entwurfsgegenstandes

Register-Transfer Ebene (register-transfer-level - RTL): Datentransfer und -verarbeitung zwischen Registern

Logikebene (gate level): Beschreibung auf Gatterniveau

Schaltkreisebene: Transistorebene im weiteren Sinne, umfasst:

- Schalterebene
- Schaltungsebene
- Bauelementebene
- Technologieebene

2.1.2. Sichten

Verhaltenssicht: Beschreibung des zeitlichen Verhaltens durch charakterisierende Variablen und deren Werteverläufe über die Zeit

$$\vec{y}(t) = f(\vec{x}(t))$$

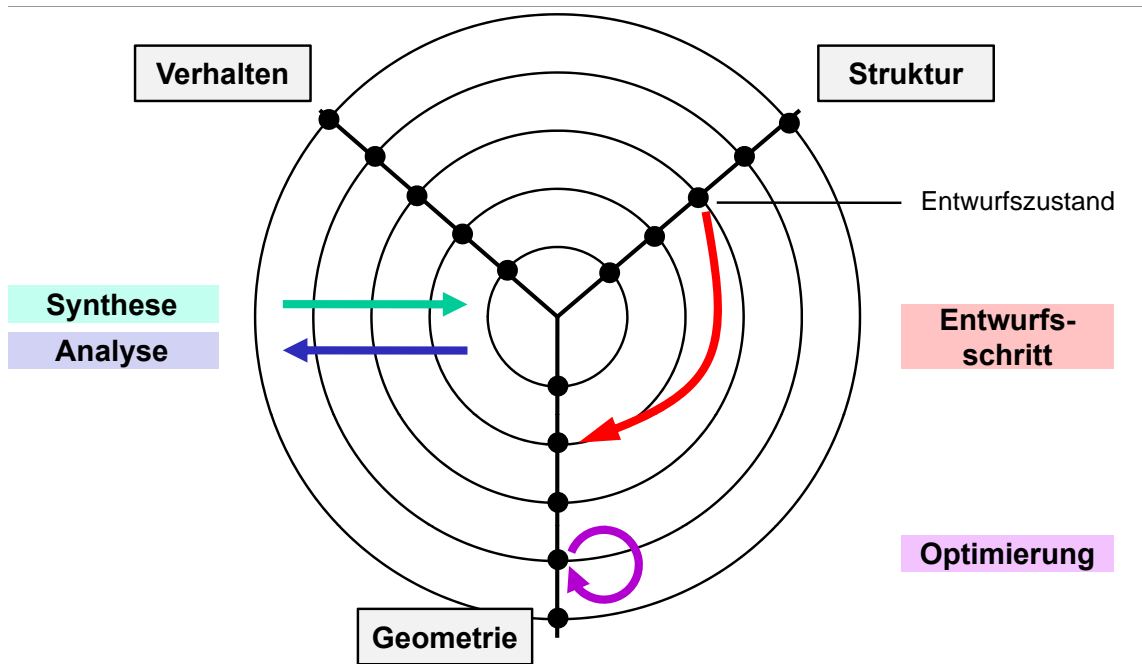
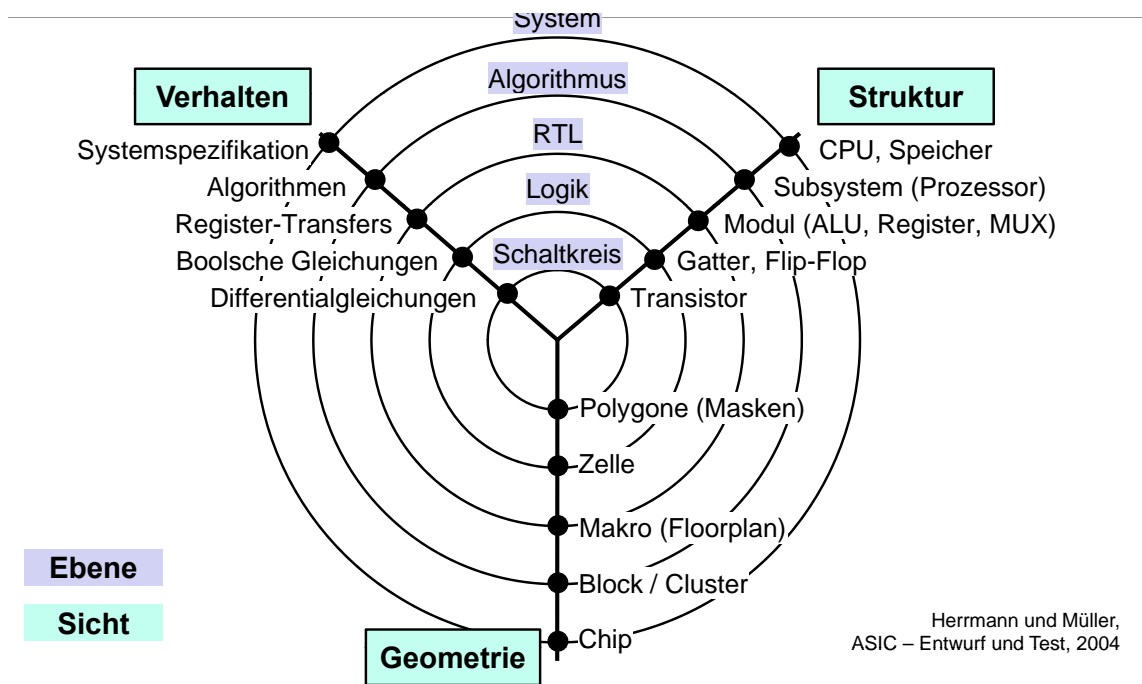
Struktursicht: Spezifizierung eines Objektes durch Subobjekte und deren Verbindungsstrukturen

Geometriesicht: Räumliche Ausdehnung der Subobjekte

Testsicht: Existenz oder Nichtexistenz angenommener struktureller oder funktionaler Defekte (F. J. Rammig, Systematischer Entwurf digitaler Systeme, B.G Teubner Stuttgart 1989)

2.1.3. Y-Diagramm nach Gajski

- Stellt Ebenen und Sichten in Form eines Y-Diagrammes dar
- Ursprünglich nur 3 Ebenen
- Heute: Erweiterung auf 5 Ebenen:
 - Systemebene
 - Algorithmische Ebene
 - Register-Transfer Ebene
 - Logikebene
 - Schaltkreisebene



2.2. Entwurfsablauf

Allgemein: Transformation einer Aufgabenstellung (Pflichtenheft) in einen fertigen Schaltkreis

Top-Down-Strategie:

- Systemebene → Schaltkreisebene
- Vorteil: Parallele Entwicklung auf unteren Ebenen
- Nachteil: Systemspezifikation zu Projektbeginn oft zu ungenau

Bottom-Up-Strategie:

- Analyse vorhandener Komponenten
- Zusammensetzen von neuen Komponenten auf höherer Ebene im Sinne der Aufgabenstellung
- Nachteil: globales Ziel wird nicht immer erreicht

⇒ **Meet-in-the-Middle**

Entwurfsschritt:

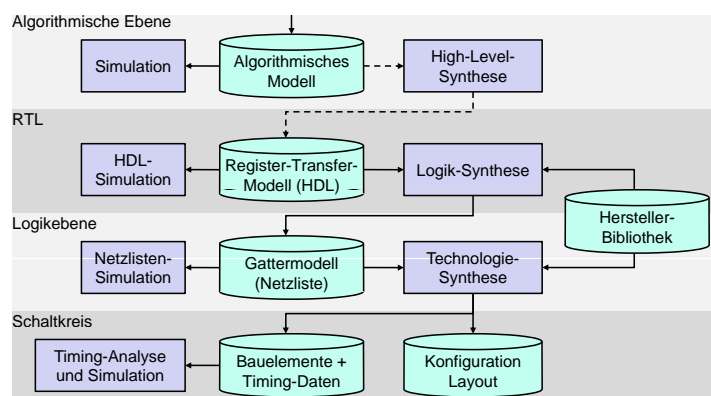
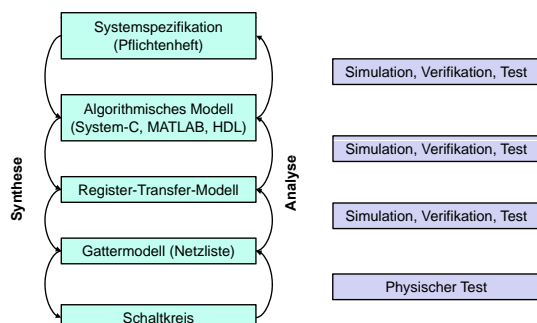
- generierende Aktivität
- überprüfende Aktivität

Syntheseschritt:

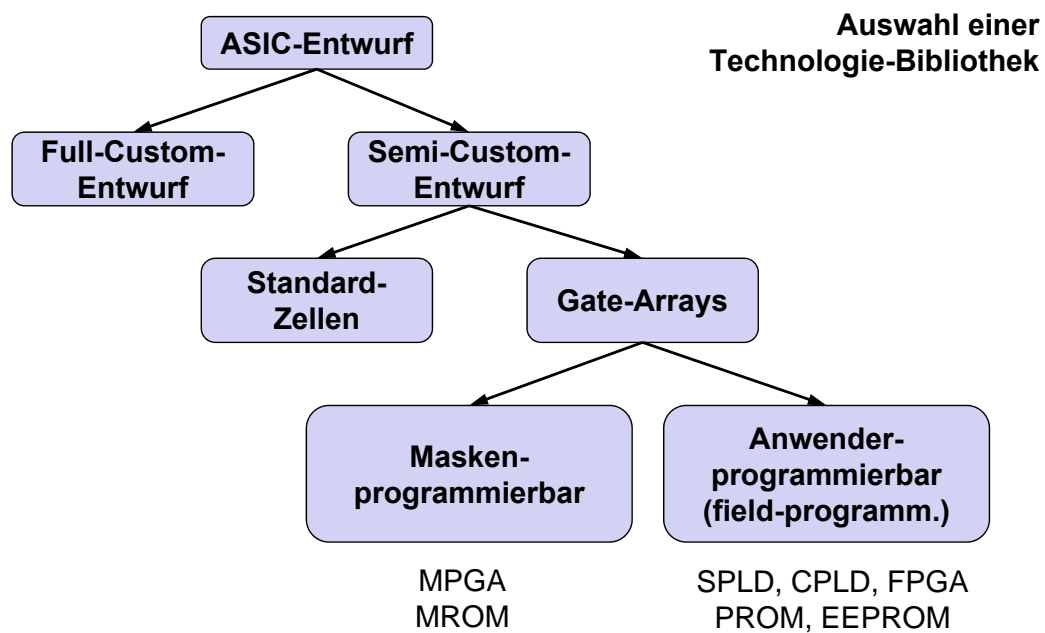
- Abbildung eines Entwurfsschrittes in Richtung auf das Entwurfsziel
- Abstraktionsgrad sinkt, Detailliertheitsgrad steigt
- Einbringung neuer Informationen

Analyseschritt:

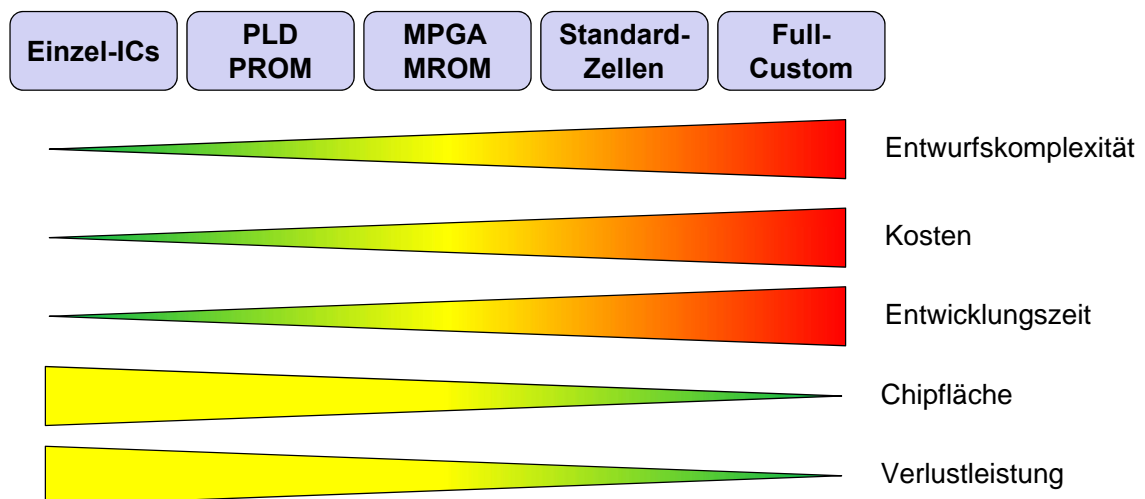
- Abbildung eines Entwurfsschrittes in umgekehrter Richtung zum Syntheseschritt
- Gewinnung abstrakter Informationen durch Zusammenfassen und Generalisieren von Details (Extraktionsprozess)
- Beispiel: Validierung eines Syntheseschrittes



2.3. Entwurststile



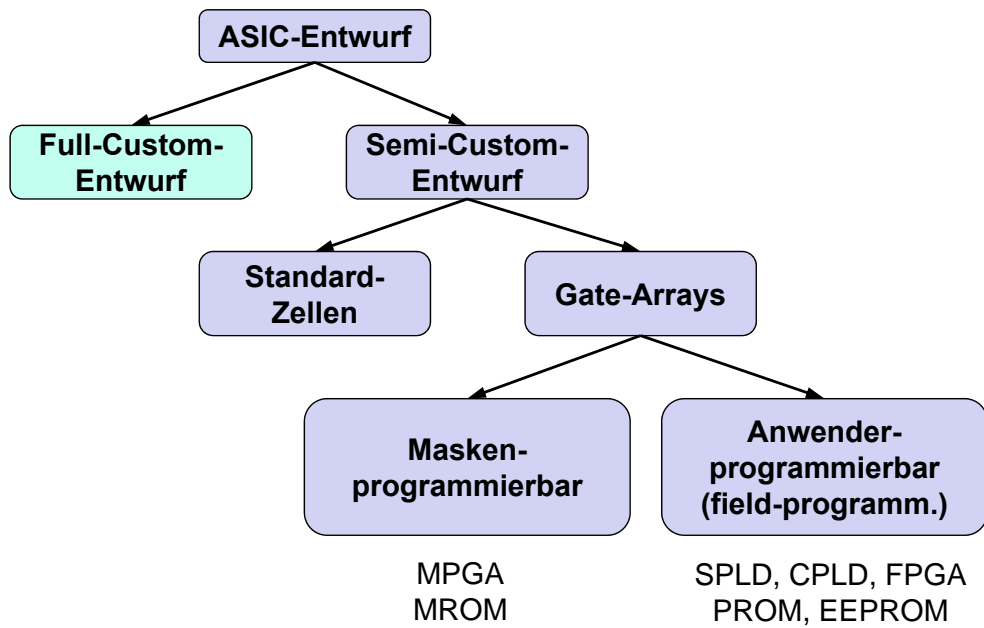
Gegenüberstellung Entwurfsalternativen



PLD = Programmierbare Logik (SPLD, CPLD, FPGA)

PROM = Programmierbare Speicher (PROM, EPROM, EEPROM)

2.3.1. Full-Custom Entwurf



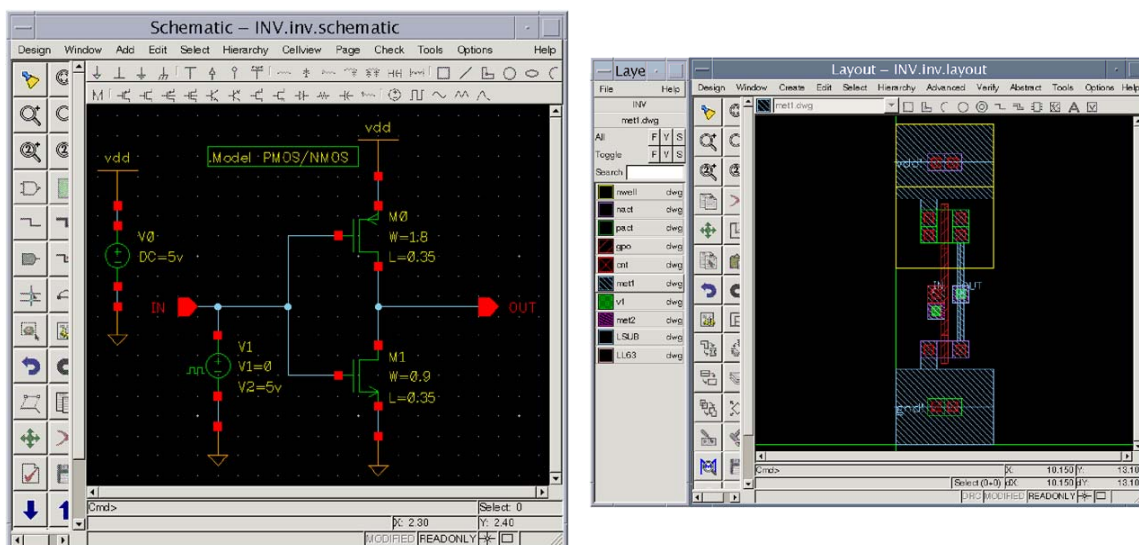
Merkmale:

- Platzierung und Verdrahtung selbst entworfener Transistoren & Gatter
- Auch Mischung von Analog- und Digitaltechnik, z.B. für spezielle I/O-Signaltreiber
- Erfüllung spezieller Anforderungen, z.B. gehärtet gegen Strahlung
- Häufig nur auf kleine Teilschaltungen angewendet
- Hoch qualifizierte Entwicklungsingenieure mit Detailkenntnissen zu den Prozessen erforderlich

Anwendung:

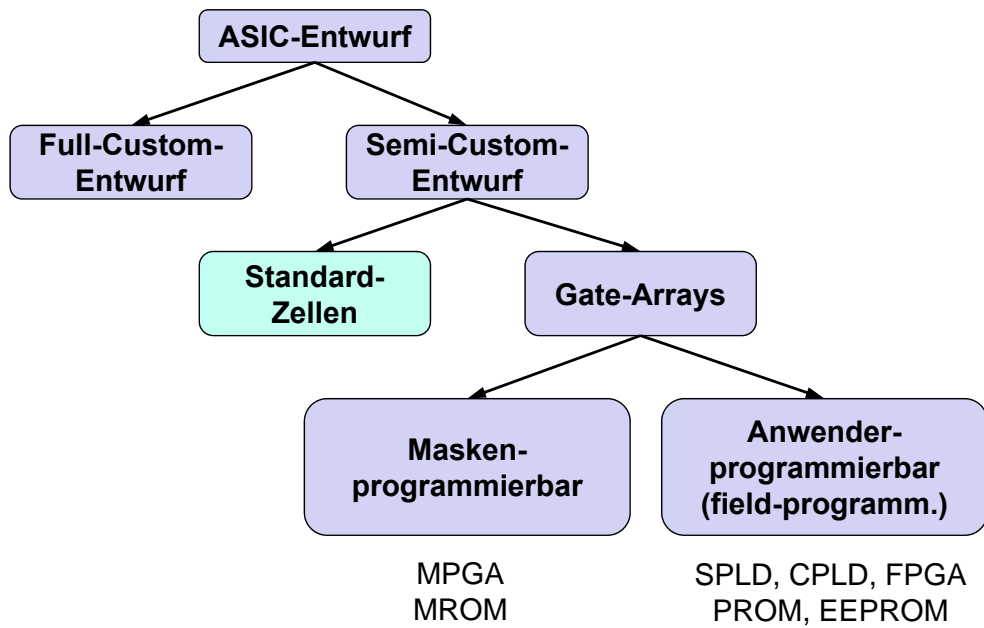
- Sensorik, Mixed-Signal-Schaltungen
- Raumfahrt

Beispiel: Inverter



JEC Huada Electronic Design: ZENI --- Full Custom IC Design Flow Workshop, <http://www.zeni-eda.com>

2.3.2. Standardzellenentwurf



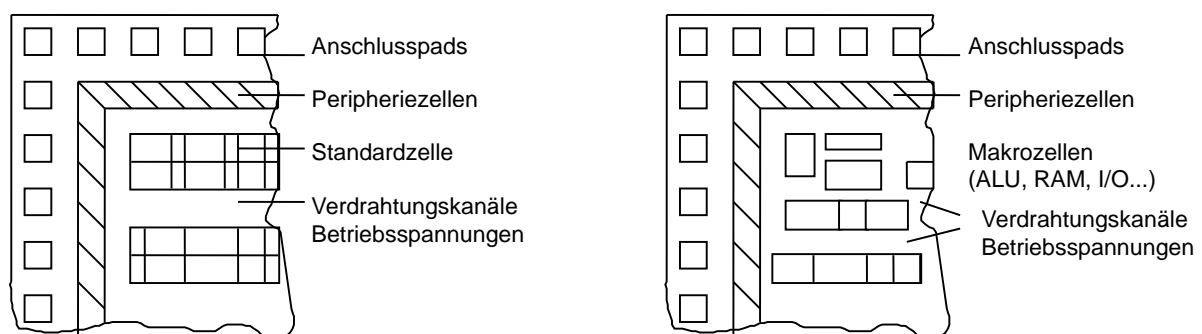
Merkmale:

- Platzierung und Verdrahtung vorgegebener Gatter- oder Makrozellen
- Auswahl einer Technologiebibliothek nach:
 - Fertigungstechnologie, Strukturbreite und Funktion
 - High-Speed, Low-Leakage oder Mischung aus Beidem
- Werkzeuggestützte Platzierung und Verdrahtung
- Makrozellen:
 - Vordefiniert oder per Generator kundenspezifisch erzeugt
 - Bsp: RAM, ALU, I/O-Komponenten

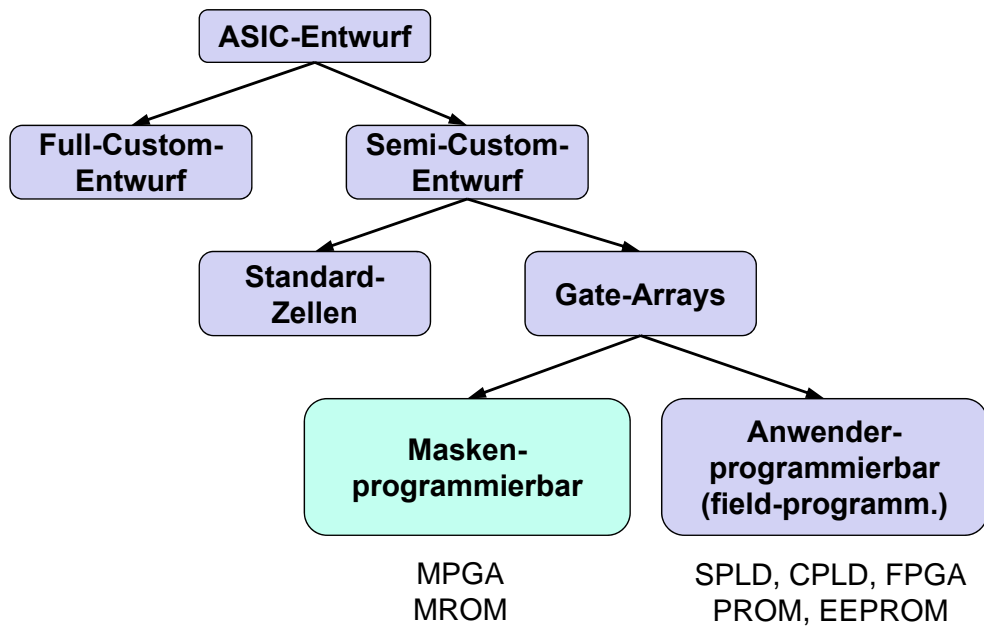
Anwendung:

- Anwendungsspezifische Schaltkreise mit hohen Stückzahlen
- Starke Optimierung bzgl. Chipfläche, Geschwindigkeit und Verlustleistung

Konventionelle Architektur vs. Strukturierte Architektur



2.3.3. Maskenprogrammierbare Gate Arrays



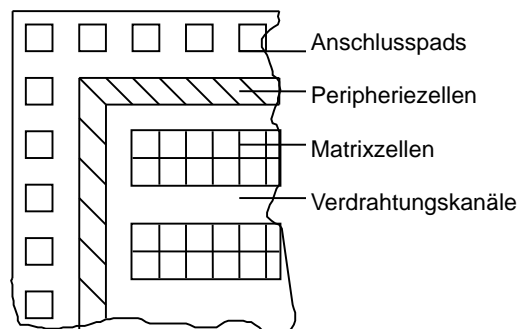
Merkmale:

- Festlegung der Funktion mittels Masken bei der Halbleiterfertigung
- Ausgewählte Masken: teilweise vorgefertigte IC (Master)
- Beispiele:
 - MPGA (Maskprogrammable Gate-Array): auch MGA
 - * Vorgefertigte universelle Gatter/Makros mit fester Anordnung
 - * Verdrahtung kundenspezifisch
 - MROM: ROM dessen Inhalt vom Kunden mit Masken festgelegt wird

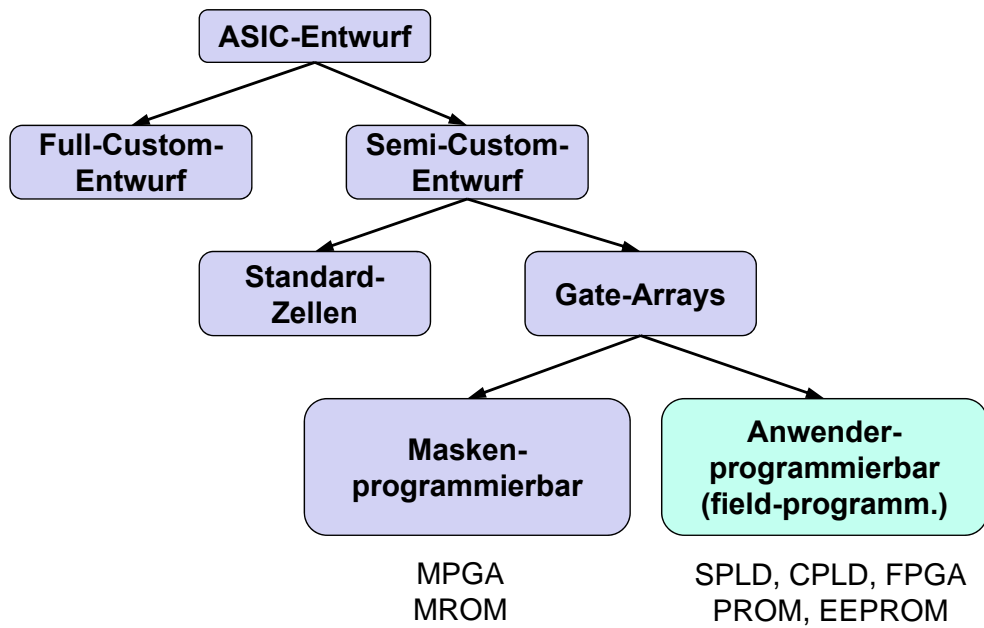
Anwendung:

- Anwendungsspezifische Schaltkreise bei mittleren Stückzahlen
- Kostenoptimiert mit reduzierten Optimierungspotential

MPGA



2.3.4. Anwenderprogrammierbare IC



Merkmale:

- Field-Programmable \Leftrightarrow feldprogrammierbar
- Vor Ort (im Feld) vom Anwender programmierbar
- Hardware ist fix. Funktionalität kann aber mittels spezieller Konfiguration “programmiert” werden

Anwendung:

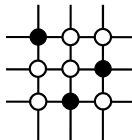
- Anwendungsspezifische IC bei kleinen und mittleren Stückzahlen
- Mehrfach neu programmierbar zwecks Optimierung und Fehlerbehebung, auch während des praktischen Einsatzes
- Einfache Integration eines ganzen Systems auf einem Chip
- Prototyping, HW-/SW-Codesign

Hardwareprogrammierung

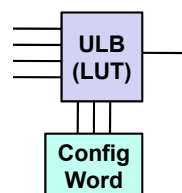
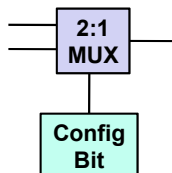
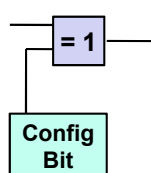
Hardwareprogrammierung

Programmiert (oder auch konfiguriert) werden können:

- Verdrahtung / Verbindung

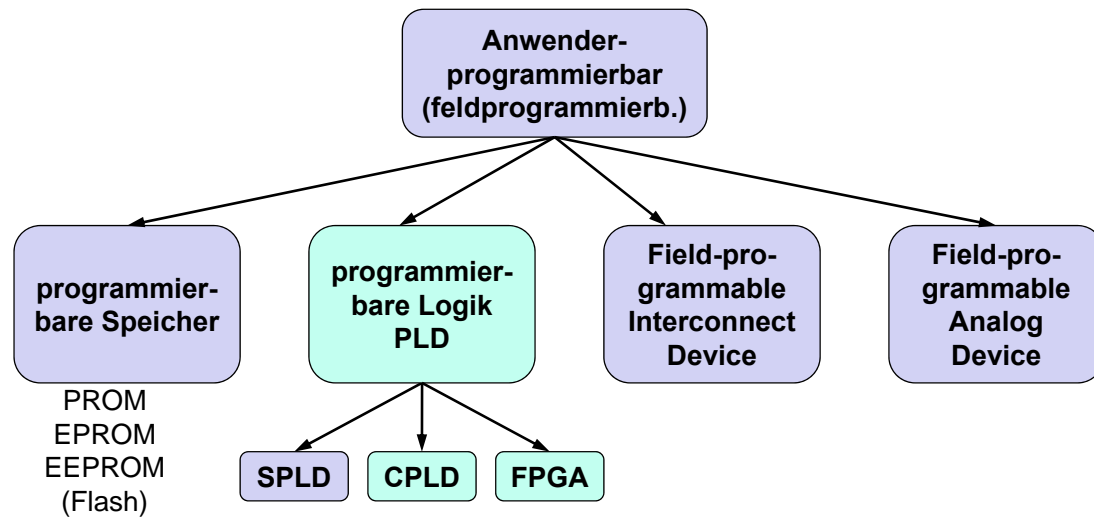


- Funktionen



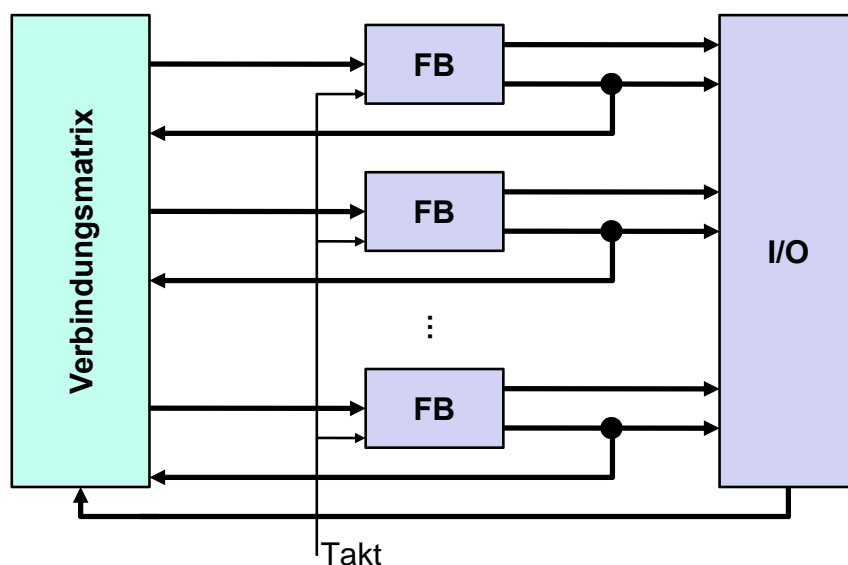
- Speicher: PROM, EPROM, EEPROM (Flash)

Klassifikation Anwenderprogrammierbare IC



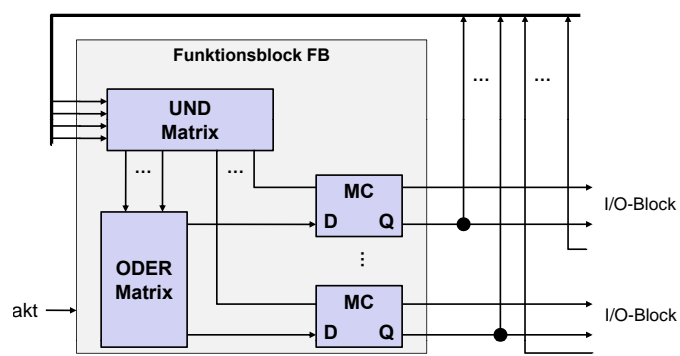
CPLD

Globale Vernetzung einer kleiner Anzahl von Funktionsblöcken (FB)



CPLD-Funktionsblock

Funktionsblock bestehend aus PLA (Und/Oder-Matrix) und Makrozellen (MC)

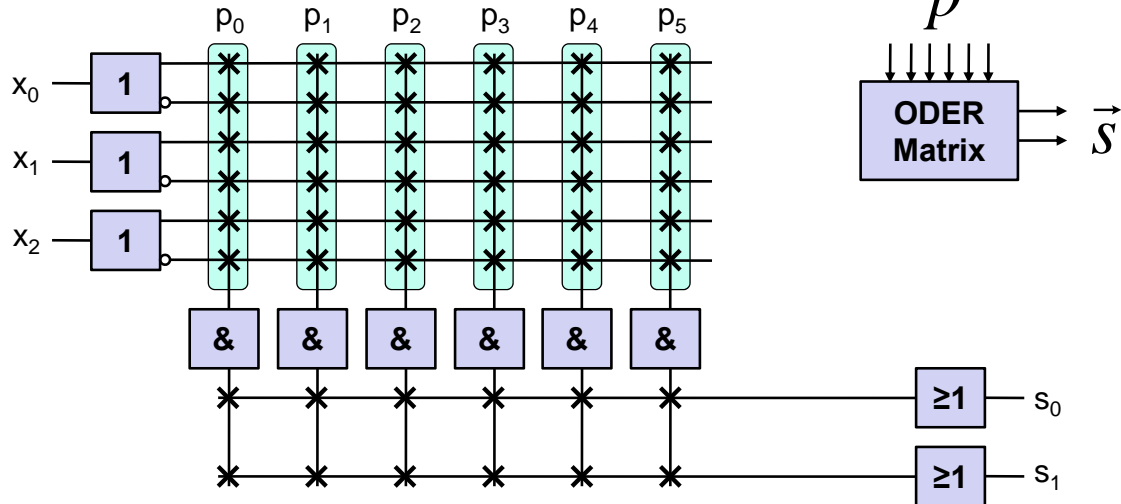


PLA

PLA

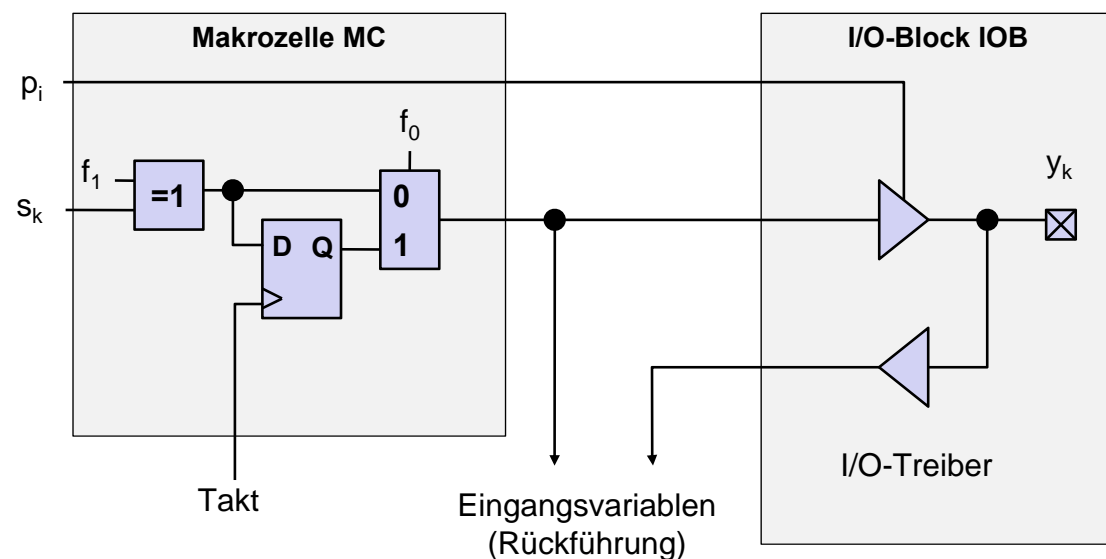
Anzahl Produktterme $< 2^{\text{(Anzahl Eingänge)}}$

Direkte Abbildung der Gleichung in DNF.



Makrozellen + I/O-Block

Verschiedene Betriebsspannungen für digitale Logik (Core) und I/O-Pads



Konfiguration der Makrozelle:

f0	Summenterm s_k
0	nicht negiert
1	negiert

f1	Ausgang y_k
0	kombinatorisch
1	Register

Steuerung des I/O-Blocks:

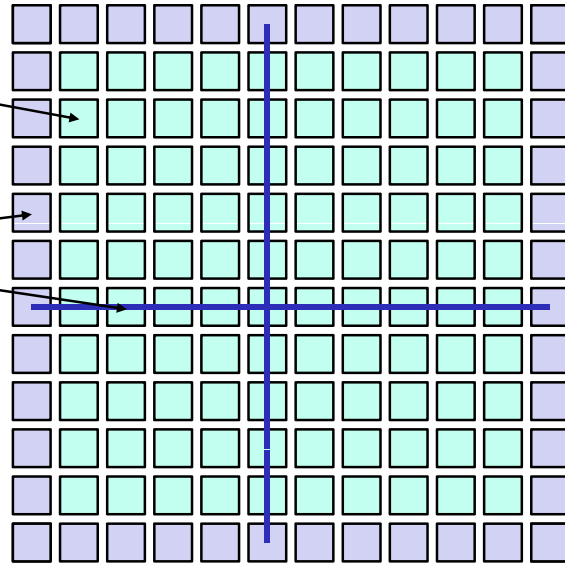
Umschaltung des I/O-Pins zwischen Ein- und Ausgang (Tri-State) zur Laufzeit mittels separatem Produktterm möglich.

FPGA-Architektur

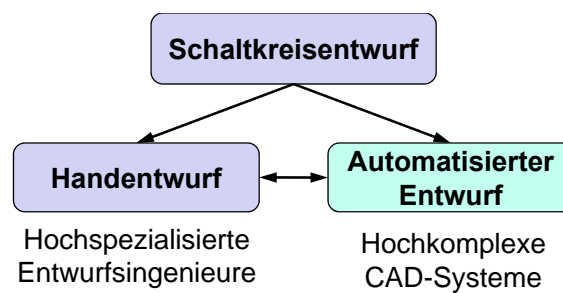
Grundlegende Bestandteile:

- Funktionsblöcke (FB):
 - angeordnet als Matrix,
 - Multiplexer- oder LUT-basiert.
- I/O-Zellen als spezielle FB.
- Allgemeine lokale Verdrahtung, sowie globale und dedizierte Signalleitungen.
- Spezielle Hard-Makros.

Details in einer späteren VL



2.4. Entwurfswerkzeuge



Auswahl CAD-Werkzeuge:

- Cadence
- Xilinx ISE
- Altera Quartus
- Synopsys

3. Automaten

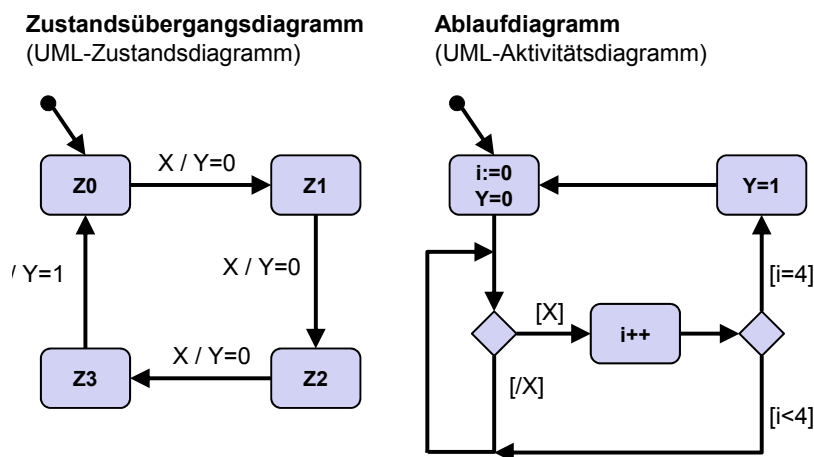
3.1. Automatendarstellung

3.1.1. Betrachtungsweisen

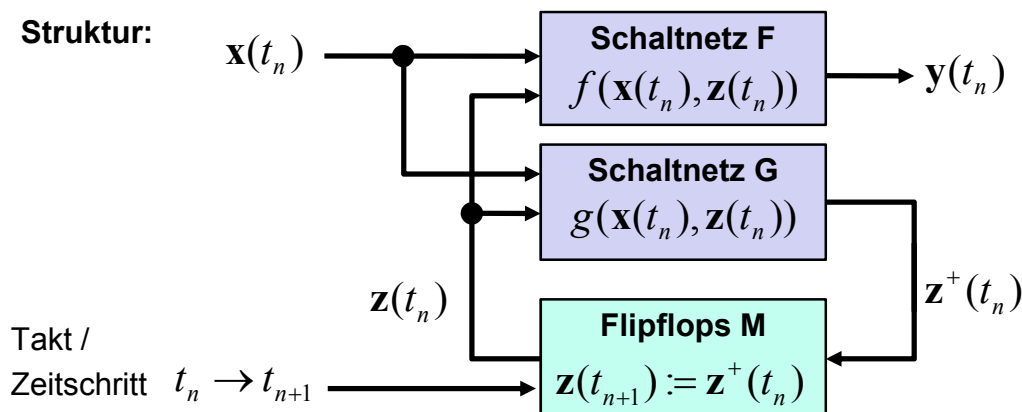
Verschiedene Sichten/Semantik:

- Zustandsübergangsdiagramm (state diagram)
 - Vernetzung von Zuständen
 - Beispiele: UML-Zustandsdiagramm, Automatengraphen, SM Charts, GRAFCET, Sequential Function Charts (SFC)
- Ablaufdiagramm (flow chart)
 - Vernetzung von Prozessen
 - Zustand ergibt sich aus der Verkettung aller Variablenzustände
 - Beispiele: UML-Aktivitätsdiagramm, Programmablaufplan (PAP)

Binärzähler mod 4 mit Trigger X und Übertrag Y



3.1.2. Automatengraphen



Endlicher Automat: Menge der möglichen Eingabezeichen, Ausgabezeichen und inneren Zustände ist endlich ...

Synchron getakteter Automat: Zustandsübergänge aller Speicherglieder erfolgen gleichzeitig, synchron zu einem Taktsignal

Notation

Beschreibung endlicher synchroner Automaten:

Eingabealphabet: $X = \{x_1, \dots, x_l\}$

Ausgabealphabet: $Y = \{y_1, \dots, y_m\}$

Zustandsmenge: $Z = \{z_1, \dots, z_k\}$

Anfangszustand: $z(t_0) \in Z$

Menge der Endzust.: $E \subseteq Z$

Übergangsfunktion: $g : (x_\lambda, z_\kappa) \rightarrow z_r$

Ausgabefunktion: $f : (x_\lambda, z_\kappa) \rightarrow y_\mu$

Automat: $A = (X, Z, Y, z(t_0), E, f, g)$

Weitere Vereinbarungen:

Eingabebezeichen: $x_\lambda = (x_1, \dots, x_l) \in X$

Ausgabebezeichen: $y_\mu = (y_1, \dots, y_m) \in Y$

Zustand: $z_\kappa = (z_1, \dots, z_k) \in Z$

Eingangsvariable: $x_\lambda \in U$

Ausgangsvariable: $y_\mu \in V$

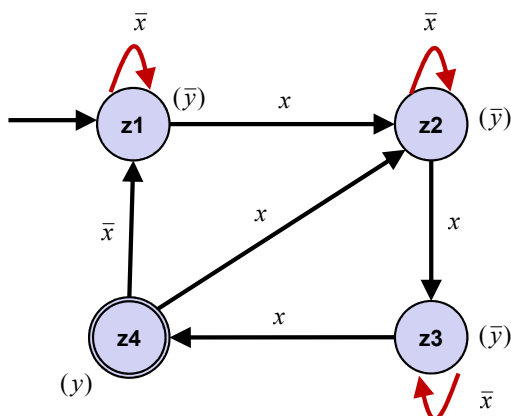
Zustandsvariable: $z_\kappa \in W$

Menge der Eing.-var.: $U = \{x_1, \dots, x_l\}$

Menge der Ausg.-var.: $V = \{y_1, \dots, y_m\}$

Menge der Zust.-var.: $W = \{z_1, \dots, z_k\}$

Grafische Darstellung



$X = \{(x), (\bar{x})\}$

$Y = \{(y), (\bar{y})\}$

$Z = \{z_1, z_2, z_3, z_4\}$

$z(t_0) = z_1$

$E = \{z_4\}$

Was fehlt?

⇒ **Prüfung auf:** Vollständigkeit und Widerspruchsfreiheit

Getaktete Automaten

Theoretische Informatik

- Verarbeitung von Eingabebezeichen sofern vorhanden
- Jedes Zeichen in der Eingabe wird einmalig verarbeitet

Technische Informatik

Taktung des Automaten mit einem Taktsignal → Abtastung der Eingabe

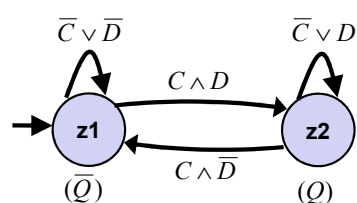
Konsequenzen:

- Zeichen für “keine Eingabe” erforderlich
- Abtastung “derselben Eingabe” in aufeinanderfolgenden Takten möglich

⇒ Korrekte Modellierung des Taktsignals erforderlich

Beispiel - Taktzustandsgesteuertes D-Flip-Flop (Latch)

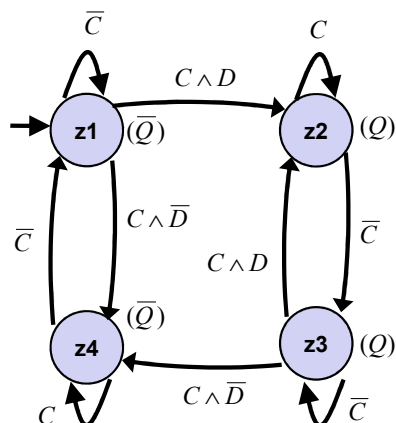
(Taktsignal C als Eingabe)



$X = \{(\bar{C}, \bar{D}), (\bar{C}, D), (C, \bar{D}), (C, D)\}$
 $Y = \{(\bar{Q}), (Q)\}$
 $Z = \{z_1, z_2\}$
 $z(t_0) = z_1$
 $E = \{\}$

Beispiel - Taktflankengesteuertes D-Flip-Flop

(Taktsignal C als Eingabe)

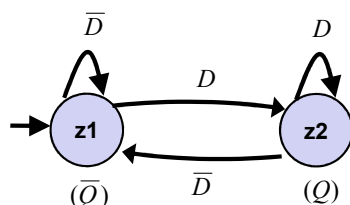


$X = \{(\bar{C}, \bar{D}), (\bar{C}, D), (C, \bar{D}), (C, D)\}$
 $Y = \{(\bar{Q}), (Q)\}$
 $Z = \{z_1, z_2, z_3, z_4\}$
 $z(t_0) = z_1$
 $E = \{\}$

→ Zu kompliziert und auch nicht notwendig!

Beispiel - Taktflankengesteuertes D-Flip-Flop

(Definition: Zustandsübergang nur bei taktflanke)



$X = \{(\bar{D}), (D)\}$
 $Y = \{(\bar{Q}), (Q)\}$
 $Z = \{z_1, z_2\}$
 $z(t_0) = z_1$
 $E = \{\}$

3.1.3. SM-Charts

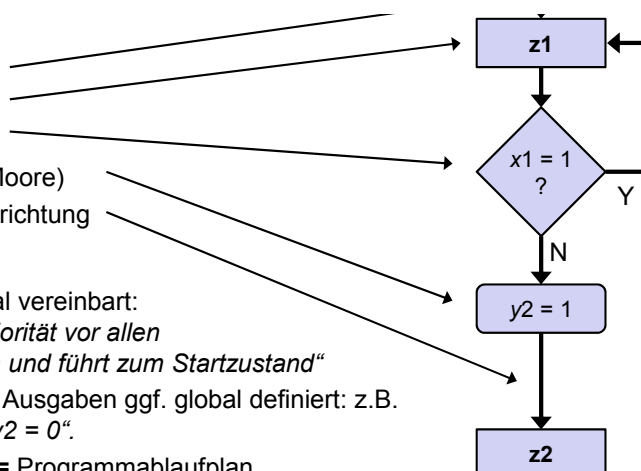
Basiselemente:

- Start
- Zustand
- Verzweigung
- Ausgabe (Mealy/Moore)
- Verbindung / Leserichtung

Weiteres:

- Reset i. Allg. global vereinbart:
z.B. „Reset hat Priorität vor allen anderen Eingaben und führt zum Startzustand“
- Standardwerte für Ausgaben ggf. global definiert: z.B. „Standardmäßig: y2 = 0“.

Achtung: SM Chart != Programmablaufplan

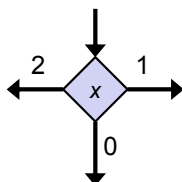
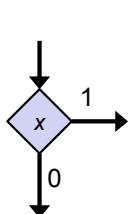


Verzweigungen

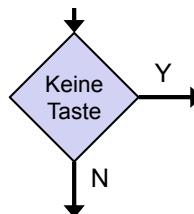
Weitere Merkmale:

- Selbstdefinierte Abkürzungen für komplexe Ausdrücke üblich
- im Allgemeinen nur Prüfung einer Eingangsvariablen (nicht Eingabezeichen) pro Verzweigung

Weitere Beispiele:



Aufzählung aller
Möglichkeiten!



Definition der
Abkürzung im Text.

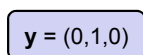
Ausgaben

Ebenso: Selbstdefinierte Abkürzungen für komplexe Ausdrücke üblich

Weitere Beispiele:



$y_2 = 1$

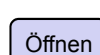


$y_0 = 0$

$y_1 = 1$

$y_2 = 0$

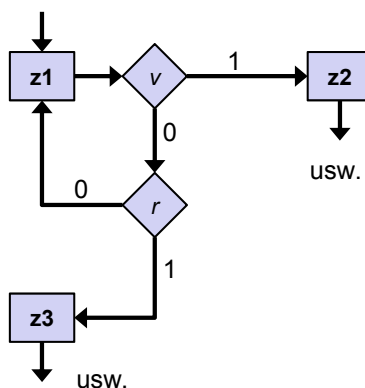
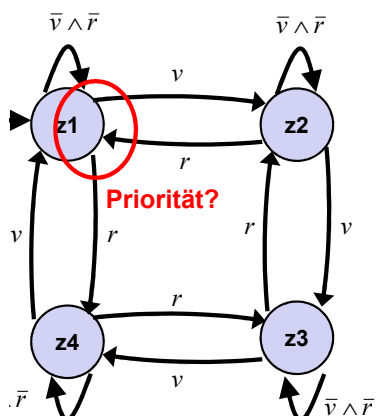
Definition des
Vektors im Text!



Definition der
Abkürzung im Text.

Vorteile gegenüber Automatengraphen

- prinzipiell Vollständig
- präzise Modellierung hierarchischer Verzweigungen



3.1.4. GRAFCET & SFC

Allgemeines:

- Struktur entspricht einem Petri-Netz
- Darstellung von schrittweise ausgeführten Ablaufbeschreibungen

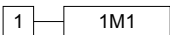
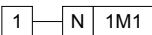
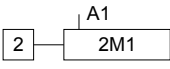
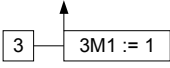
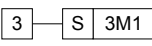

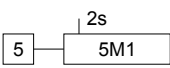
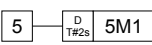
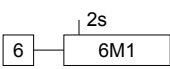
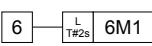

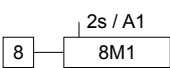
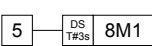
- Anwendung in Automatisierungstechnik und Verfahrenstechnik
- Verwendbar zur Programmierung von Speicherprogrammierbaren Steuerungen (SPS)
- Automatenkopplung nicht vorgesehen; muss durch Eingangs- und Ausgangsvariablen realisiert werden

GRAFCET GRAPhe Functionnel de Commande Etapes/Transitions
SFC sequential function chart

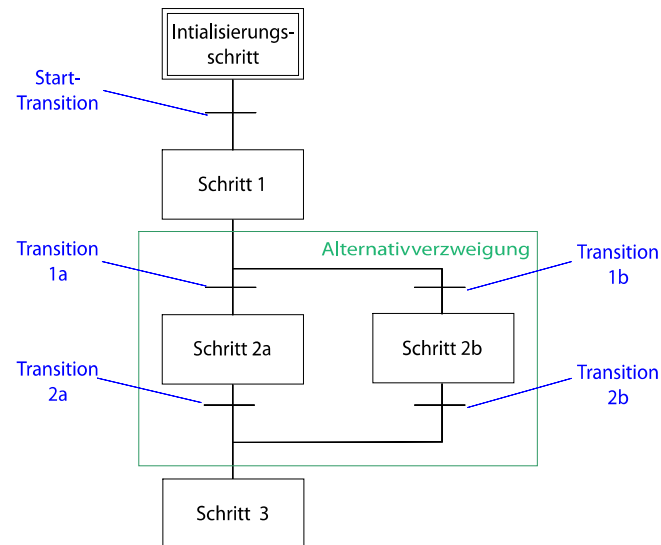
Grundelemente

- Schritt:
 - entspricht einem Zustand
 - mindestens ein Initialisierungsschritt notwendig
- Transition
 - Schaltbedingung für Übergang zwischen zwei Schritten (boolesche Gleichung) = zeitliche Ereignisse
 - Schritte und Transitionen folgen immer aufeinander
 - Leserichtung: Transitionen können nur von oben nach unten durchlaufen werden
- Aktion
 - einem Schritt zugeordnet
 - realisiert die Ausgabe
 - verschiedene Aktionsarten möglich

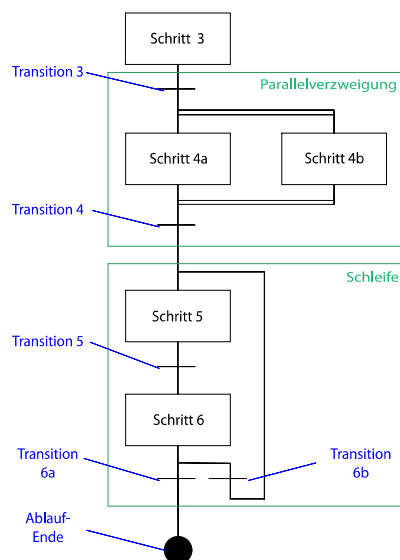
Aktionen

Aktion	GRAFCET	Sequential Function Chart
kontinuierliche Aktion (solange Schritt aktiv ist)		
Aktion mit Zuweisungsbedingung		
Speichernde Aktion		
Rücksetzende Aktion		
verzögerte Aktion		
zeitbegrenzte Aktion		
Aktion für einen Taktzyklus		
kombinierte Aktion		

Kontrollfluss: Verzweigungen



Kontrollfluss: Schleife



Parallelverzweigung:

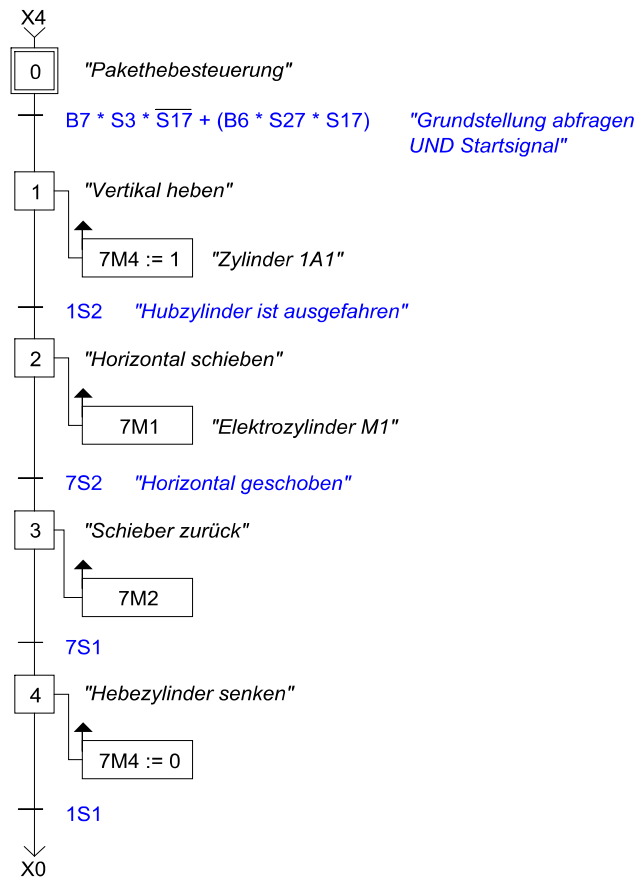
Gemeinsame Transition zum Start bzw. Ende aller Pfade

Vergleich von GRAFCET & SFC

- identische Grundstruktur, Verzweigung und Initialisierung
- Ablauf-Ends in GRAFCET nicht notwendig, dann Sprung zu Initialisierungsschritt
- Schleifen in SFC
 - Vorwärtssprung = Alternativverzweigung
 - Rückwärtssprung = Schleife
- Aktionen s.o.

⇒ GRAFCET ist für den Entwurf konzipiert, SFC für die Implementierung

Beispiel zu GRAFCET



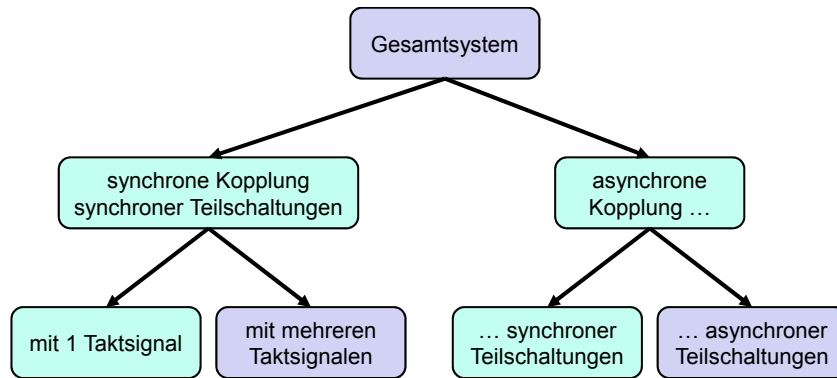
3.2. Automatenkopplung

Zerlegung der Gesamtaufgabe in Teilschaltungen zwecks:

- Nebenläufigkeit (statt sequentieller Abarbeitung)
- Energieeinsparung durch Abschaltung ungenutzter Komponenten (statt Dauerbetrieb der gesamten Schaltung)
- komponentenbasierter, testfreundlicher Entwurf (statt monolithischem Design)
- Integration von Komponenten von Drittanbietern (statt eigenem Entwurf aller Komponenten)

Außerdem: Kommunikation mit der Außenwelt erfordert im Allgemeinen nebenläufig arbeitende I/O-Controller aufgrund asynchron eintreffender Ereignisse (Nachrichten) → Client-Server Architektur

Zeitliche Kopplung



Asynchrone Kopplung synchroner Teilschaltungen

Merkmal: Mehrere (lokale) Taktsignale / Taktdomänen (clock domain)

Vorteile:

- Timing-Analyse sichert Zeitverhalten innerhalb der Domäne
- passende Taktfrequenz innerhalb der Domäne
 - Kombination High-Speed und Low-Power
 - Taktfrequenz dynamisch anpassbar
 - Abschalten einer Domäne für Standby

Nachteil:

Datenaustausch zwischen Taktdomänen erfordern:

- Module mit Single-Bit-Synchronizer, Cross-Clock-FIFOs
- spezielle Timing-Constraints

Synchrone Kopplung mit mehreren Taktsignalen

Merkmal: Mehrere (lokale) Taktsignale / Taktdomänen, zwischen denen aber spezielle Abhängigkeiten bestehen (dependent clocks)

Vorteile:

- Timing-Analyse sichert Zeitverhalten innerhalb der Domäne und auch zwischen den Domänen
- passende feste Taktfrequenz je Domäne
- keine speziellen Synchronisationselemente erforderlich

Nachteil: Taktfrequenzen sind statisch.

3.2.1. Synchrone Kopplung

Kommunikation über:

- Zustände oder
- Ausgaben

Anordnung:

- parallel oder
- seriell

Kommunikation über Zustände

Automat 1:

$$Z'_1 = Z_1 \times Z_2$$

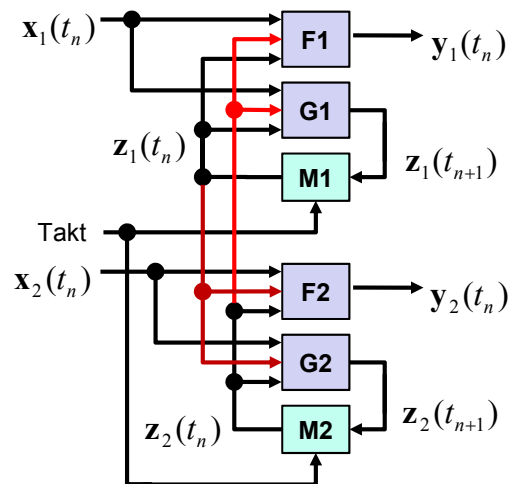
Automat 2:

$$Z'_2 = Z_2 \times Z_1$$

$$\rightarrow Z = Z_2 \times Z_1$$

Anwendung:

Theoretische Informatik,
z.B. Produktautomat



Kommunikation über Ausgaben

Automat 1:

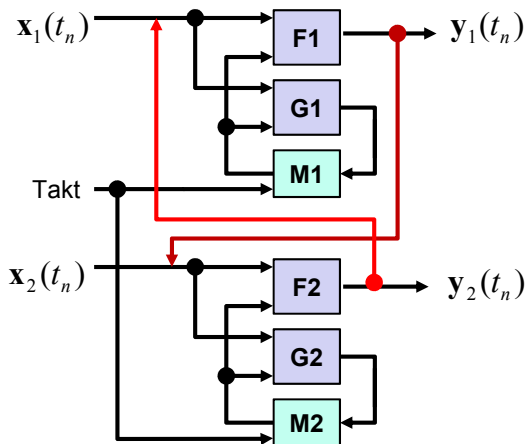
$$X'_1 = X_1 \times Y_2$$

Automat 2:

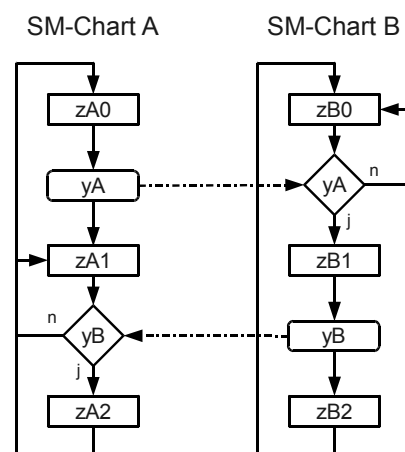
$$X'_2 = X_2 \times Y_1$$

Anwendung:

Technische Informatik,
da aufgrund von Modularisierung
kein Zugriff auf interne Zustände
vorgesehen

**Darstellung im SM-Chart:**

Bsp.: Kommunikation über die
Ausgaben yA und yB



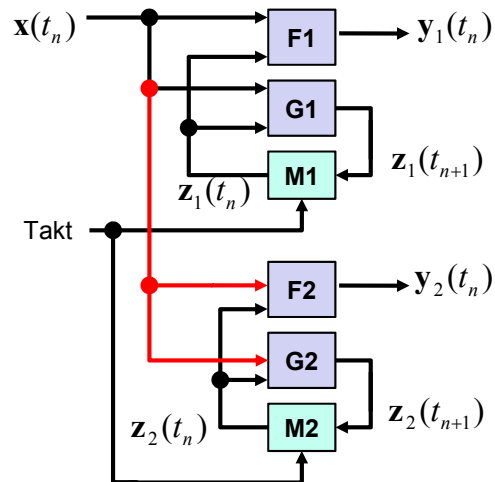
Parallele Anordnung

Automat 1 und 2:

$$X = X_1 = X_2$$

Anwendung:

z.B. Erkennung verschiedener Eingabefolgen



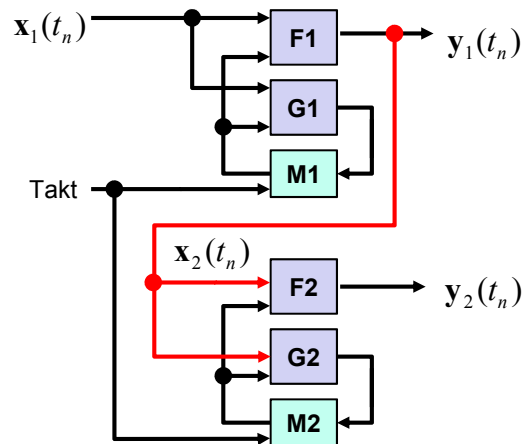
Serielle Anordnung

Automat 2:

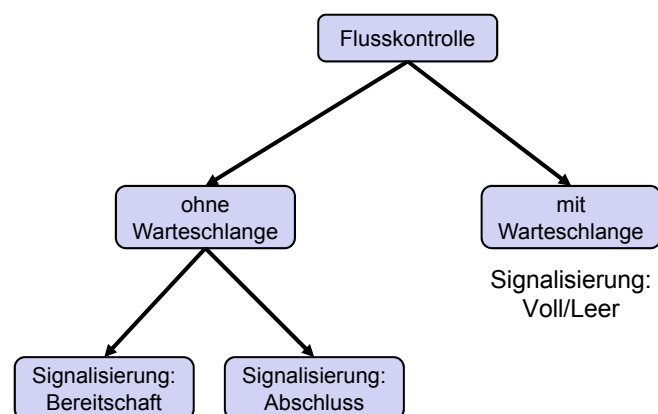
$$X_2 = Y_1$$

Anwendung:

- Verarbeitung in mehreren Teilschritten, Pipeline
- häufig mehrere seriell angeordnete Kopplungen zu verschiedenen Komponenten



Flusskontrolle



Beispiel: Produktautomat

Definition:

$$A = A_1 \times A_2 = (X, Z_1 \times Z_2, Y_1 \times Y_2, (z_1(t_0), z_2(t_0)), E, f, g)$$

Mit $X = X_1 = X_2$, $Z = Z_2 \times Z_1$

$$f((z_1, z_2), x) = (f_1(z_1, x), f_2(z_2, x))$$

$$g((z_1, z_2), x) = (g_1(z_1, x), g_2(z_2, x))$$

→ Kopplung über Zustände, parallele Anordnung

Anwendungsbeispiele:

$E = E_1 \times E_2$: Schnitt zweier Sprachen

$E = Z_1 \times E_2 \cup E_1 \times Z_2$: Vereinigung zweier Sprachen

Beispiel: Datenverarbeitung

Automat 2 besitze 2 Schnittstellen

→ separate Ein- und Ausgabealphabet je Schnittstelle

→ Zusammenfassung der Teilalphabete:

$$X_2 = X_{2a} \times X_{2b}$$

$$Y_2 = Y_{2a} \times Y_{2b}$$

→ Serielle Kopplung über Ausgaben:

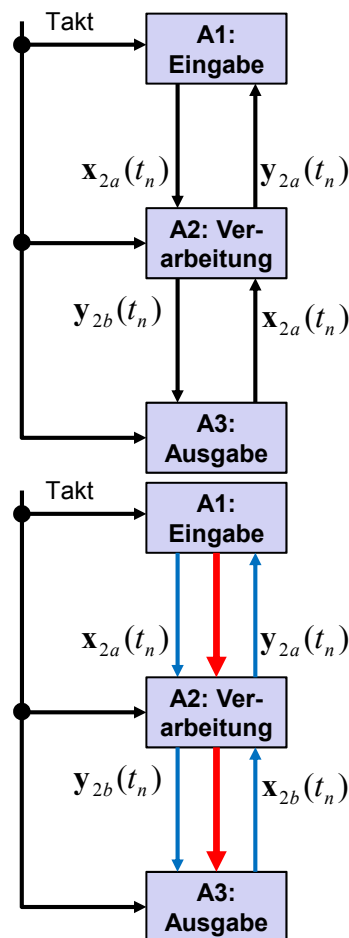
$$X_{2a} = Y_1 \quad X_1 = Y_{2a}$$

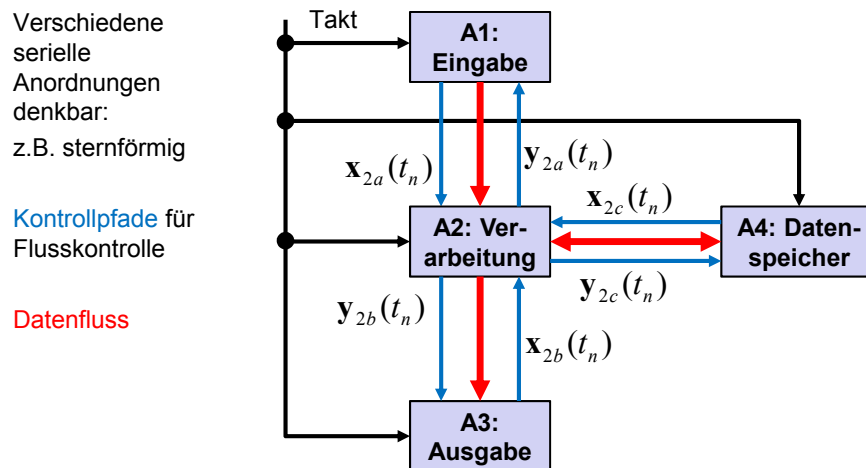
$$X_{2b} = Y_3 \quad X_3 = Y_{2b}$$

Automatenkopplung erfolgt primär über **Kontrollpfade** → Flusskontrolle

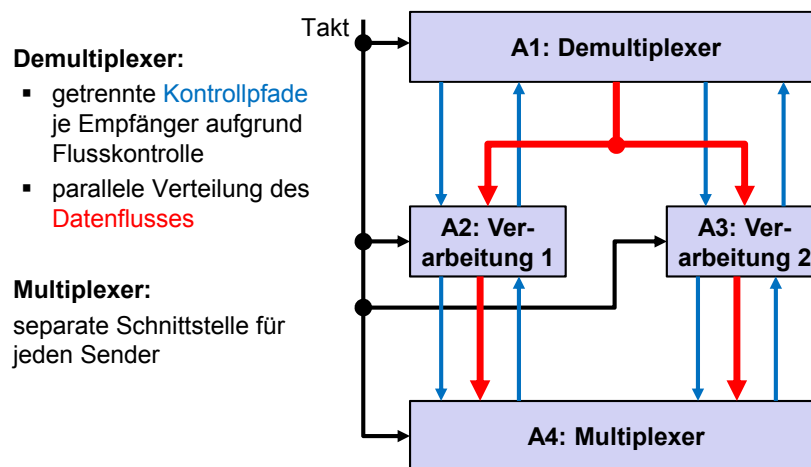
Datenfluss wird indirekt über Kontrollpfade gesteuert:

- Datenfluss nicht Bestandteil der Flusskontrolle.
- Ein- und Ausgabealphabet entsprechen den zu verarbeitenden Daten.





Beispiel: De-/Multiplexer



3.2.2. Asynchrone Kopplung

Abtastproblem

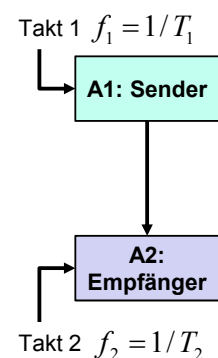
Kommunikation zwischen verschiedenen Taktdomänen aufgrund verschiedener Taktfrequenzen und -phasen komplexer:

- keine feste Zuordnung zwischen zwei Zeitpunkten, allg. Annahme $t_n = n * T_1 \neq m * T_2 = t_m$ mit $n, m \in \mathbb{N}$
- Abtastung von Signalvektoren kann zu Fehlern aufgrund verschiedener Signallaufzeiten führen

Flusskontrolle

Berücksichtigung verschiedener Taktfrequenzen im Protokoll notwendig:

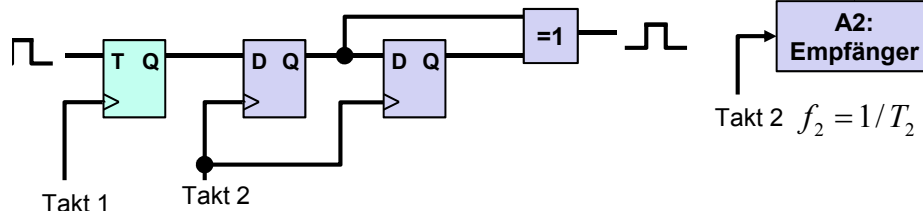
- Mehrfaches Lesen desselben Ausgabezeichens, wenn $f_2 > f_1$
- Verpassen von Ausgabezeichen, wenn $f_2 < f_1$



Übertragung eines einzelnen Bits

Abtastung des gesendeten Signals gemäß
 Abtasttheorem mit $f_2 > 2f_s$
 f_s = max. Änderungsfrequenz des Signals
 → minimale Impulsbreite $t_{p,s} > 2 \cdot T_2$

Lösung: Übertragung von einzelnen (mindestens $t_{p,s}$
 auseinanderliegenden) Impulsen mittels Signalwechseln



Übertragung eines Signalvektors

Mehrere Varianten möglich:

1. Gray-Code
2. Steuerung mittels einzelner, separater Bits
3. Warteschlange mittels Cross-Clock-FIFO

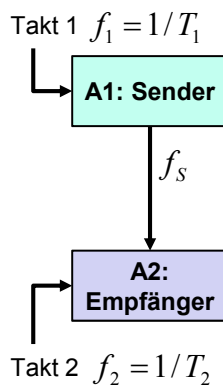
Variante 1: Gray-Code

Bedingungen:

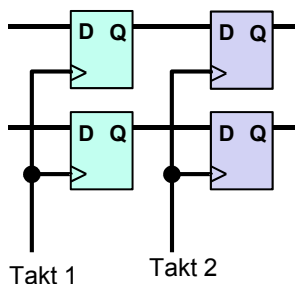
- Hamming-Abstand zwischen zwei aufeinanderfolgenden Ausgabezeichen ist kleiner gleich 1.
- Variation der Signallaufzeit $t_{skew} < T_2$

Erfüllt durch:

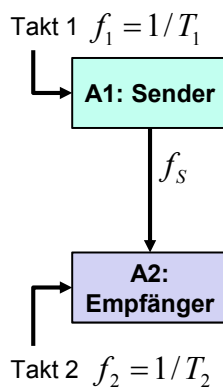
- Aufeinanderfolgende Zeichen des Gray-Codes
 - Ausgabe aus Registern damit t_{skew} klein
- Entweder Abtastung des alten oder des neuen Wertes.



Schaltung:



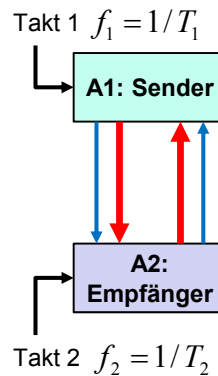
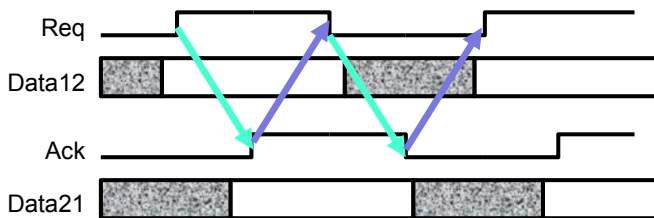
Signalverlauf:



Variante 2: Steuerung mittels einzelner Bits

Prinzip: synchrone Kommunikation mittels

- **Kontrollfluss** bestehend aus jeweils 1 Bit:
 - Sender → Empfänger: Request (Req)
 - Empfänger → Sender: Acknowledge (Ack)
- **Datenbus** bestehend aus mehreren Bits (je Richtung)
- Datenworte bleiben während Übertragung konstant.

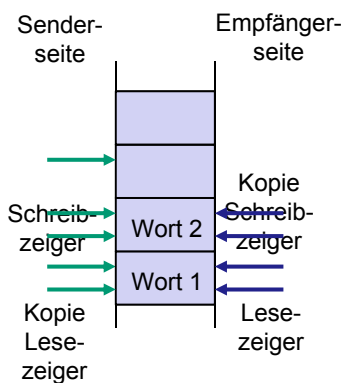
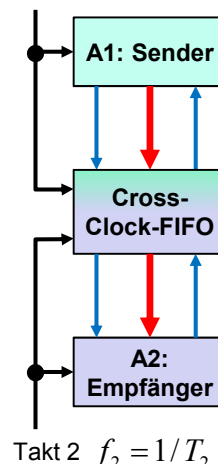


Variante 3: FIFO-Warteschlange

Nachteil Variante 2: Niedrige Datenrate

Lösung: Warteschlange mit Ringspeicher

- Schreiben von 1 Datenwort pro Takt (T_1), solange FIFO nicht voll.
- Lesen von 1 Datenwort pro Takt (T_2), solange FIFO nicht leer.
- Lesefreigabe von geschriebenen Datenwörtern erst nachdem diese vollständig in den Ringspeicher geschrieben wurden.
- Kontinuierliche Datenübertragung bei $f_1 = f_2$ möglich (unabhängig von Phasenlage).



Eigenschaften:

- Vergleich von Zeigern jeweils innerhalb einer Taktdomäne → Zeigerkopien.
- Übertragung der Zeiger (Signalvektoren) mittels Gray-Code.
- Durch Kopie der Zeiger und anschließendem Vergleich → Latenz
→ Aktualisierung der Speicherzelle vor dem Lesen abgeschlossen.
- Gleichzeitiges Schreiben und Lesen (verschiedener) Wörter möglich.

3.3. Initialisierung

3.3.1. Reset vs. Power-Up

Nicht programmierbare Schaltkreise:

- Reset notwendig für Initialisierung der Zustandsregister
- Datenregister können von Automaten initialisiert werden

Programmierbare Schaltkreise:

- Initiale Registerbelegung wird durch Programmierung festgelegt
- Reset-Eingang ist dh. optional

→ **Wiederverwendungsgerechter Entwurf:**

- Reset-Eingang vorsehen
- (Zustands-)Register bei Power-Up und Reset gleichermaßen belegen

3.3.2. Synchrones Reset**Vorteile:**

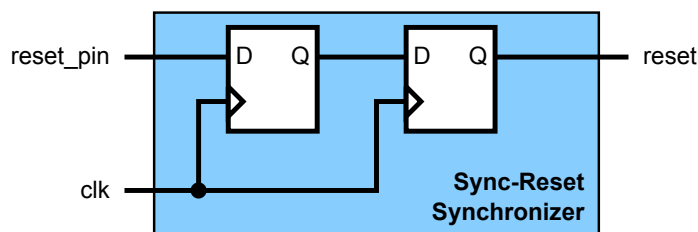
- Synthese immer möglich: Set/Reset als zusätzliche Variable in FF-Ansteuergleichung, sofern nicht explizit vorhanden
- Logik-Zusammenfassung möglich
- Überprüfung in Timing-Analys

Nachteil:

Free-Running Clock benötigt, damit Reset auch ausgelöst wird

Reset-Synchronizer

- Reset-Pin muss synchronisiert werden, damit alle Register in der gleichen Taktperiode zurückgesetzt werden
- Synchronisation mit (min.) 2 FFs, um Metastabilitäten zu vermeiden
- Reset-Pin ist ggf. zu negieren
- Schaltung allg. verwendbar für Synchronisation asynchroner Signale

**3.3.3. Asynchrones Reset****Vorteile:**

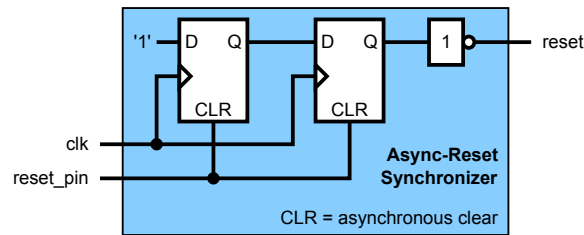
- Keine Free-Running Clock benötigt um Reset auszulösen, u.U. notwendig für Register, die Schaltungsausgänge treiben
- Separater FF-Eingang, damit kein Einfluss auf Timing

Nachteile:

- Implementierung erfordert globale Verdrahtungsressourcen wie Taktsignale
- Timing-Analyse problematisch und häufig standardmäßig ignoriert
- Reset am Register darf nicht in zeitlicher Nähe zur Taktflanke losgelassen werden → sonst Metastabilitäten

Reset-Synchronizer

- Reset muss synchron losgelassen werden, damit alle Register in der gleichen Taktperiode wieder in Betrieb gehen
- Reset-Pin ist ggf. zu negieren
- Reset-Tree sorgt für zusätzliche, notwendige! Verzögerung



3.3.4. Coding Guidelines für Reset

Empfehlungen:

- Synchrones Reset
- Reset-Eingang setzt nur Zustandsregister zurück
→ Fan-Out von Reset-Eingang klein
- Power-Up-Wert automatisch ermitteln lassen. damit identisch zu Reset-Wert

4. Hardwarebeschreibungssprachen - Hardware Description Language (HDL)

4.1. Allgemein

4.2. VHDL

4.2.1. Geschichte

VHDL - Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

1981 Initiiert durch US Verteidigungsministerium um die Wiederverwendung von Hardware in neuen Technologien zu vereinfachen

1983 Intermetrics, IBM and TI wollen eine ADA-basierte HDL entwerfen

1985 Vollendung des VHDL-Core in Version 7.2

1986 US Verteidigungsministerium übergibt alle Rechte an VHDL an IEEE

1987 VHDL wird IEEE-Standard 1076-1987

1987 US Verteidigungsministerium benötigt VHDL-Modelle für alle eingekauften ASICs

1988 VHDL wird ANSI-Standard

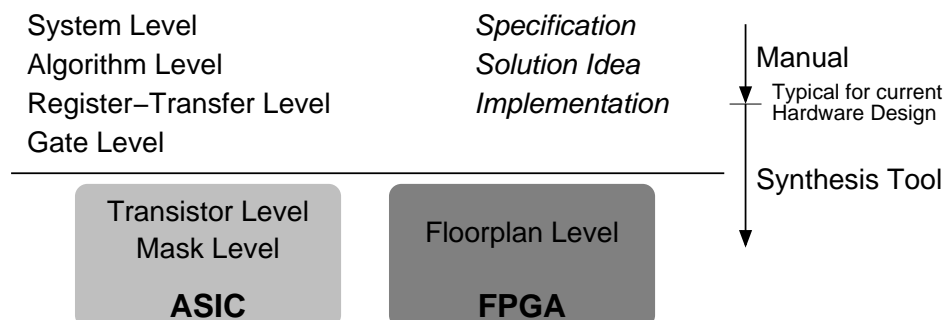
1993 IEEE-Standard 1076-1993 ist immer noch weitv erbreitet

2008 IEEE-Stanard 1076-2008 letzte Hauptversion von VHDL

4.2.2. Abstraktionsebenen

VHDL ist geeignet folgende Ebenen zu beschreiben:

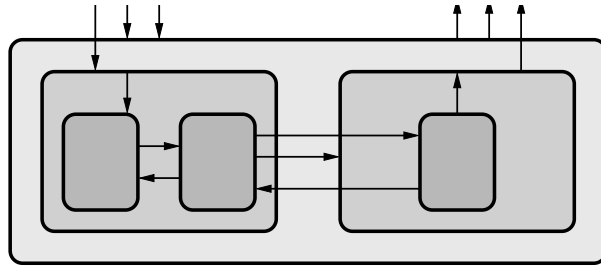
- Systemebene
- Algorithmische Ebene
- Register-Transfer-Ebene (RTL)
- Logikebene (gate level)



4.2.3. Grundsätze

Struktur

Entwürfe werden aus Komponenten zusammengebaut.



- typischerweise: weite, bidirektionale Schnittstellen (Drähte)
- hierarchischer Instanzierungsbaum der Komponenten
- Verdrahtung in oder durch Instanzierungs-Modul

Kohärenz Implementierungen werden aus kohärenten Aussagen gebildet.

- Reihenfolge der Aussagen ist unwichtig
 $p \leq a \text{ xor } b;$
 $s \leq p \text{ xor } c;$
 ist äquivalent zu
 $s \leq p \text{ xor } c;$
 $p \leq a \text{ xor } b;$
- Abhängigkeiten werden durch Signalverbindungen definiert!

4.3. Verilog

5. Field-programmable Gate-Array (FPGA)

5.1. Architektur

5.2. Funktionsblöcke

5.3. I/O-Zellen

5.4. Verdrahtung

5.4.1. Topologie

5.4.2. Technologie

5.5. Speicherelemente

5.5.1. LUT-RAM

5.5.2. Block-RAM

5.6. IP-Cores

5.7. Konfigurierbarkeit

5.8. Konfigurationsmodi

6. Modellierung

7. Simulation

8. Zeitverhalten

9. Test

10. Hochgeschwindigkeit

11. Verlustleistung

Teil II.

Entwurf eingebetteter Systeme

Teil III.

Parallelverarbeitung

Teil IV.

Appendix

1. VLSI-Systementwurf Praktikum

1.1. Kurze Beschreibung des Terasic DE0 Board

Operating voltage for I/O Pins (unless otherwise specified): 3.3-V LVTTTL

2.1 Clock

50 MHz (primary)	50 MHz (secondary)
(G21)	(B12)

2.2 Buttons

The buttons are low-active. Depending on board revision buttons may be debounced. Assume buttons beeing not debounced.

BTN2	BTN1	BTN0
(F1)	(G3)	(H2)

2.3 Switches

The switches are high-active and not debounced.

SW9	SW8	SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
(D2)	(E4)	(E3)	(H7)	(J7)	(G5)	(G4)	(H6)	(H5)	(J6)

2.4 LEDs

Each LED is driven directly by an I/O pin on the Cyclone III FPGA (i.e. LEDs are high-active).

Current strength: 8 mA

Slew rate: 2

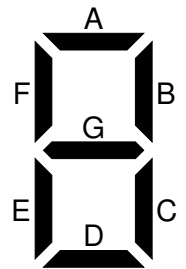
LEDG9	LEDG8	LEDG7	LEDG6	LEDG5	LEDG4	LEDG3	LEDG2	LEDG1	LEDG0
(B1)	(B2)	(C2)	(C1)	(E1)	(F2)	(H1)	(J3)	(J2)	(J1)

2.5 7-Segment Display

Each of the 4 digits consists of 7 segments and a dot. The individual segments are enabled through the LED's cathode (low-active).

Segment Selection (Cathode Control)

	A	B	C	D	E	F	G	Dot
Digit 3	(B18)	(F15)	(A19)	(B19)	(C19)	(D19)	(G15)	(G16)
Digit 2	(D15)	(A16)	(B16)	(E15)	(A17)	(B17)	(F14)	(A18)
Digit 1	(A13)	(B13)	(C13)	(A14)	(B14)	(E14)	(A15)	(B15)
Digit 0	(E11)	(F11)	(H12)	(H13)	(G12)	(F12)	(F13)	(D13)



2.6 PS/2 Connector

Default		Extension	
CLK	DATA	CLK	DATA
(P22)	(P21)	(R21)	(R22)

For using two PS/2 devices simultaneously an extension PS/2 Y-Cable is needed.

2.7 VGA Connector

RGB signals are high-active, sync signals low-active.

Red			
Red[3]	Red[2]	Red[1]	Red[0]
(H21)	(H20)	(H17)	(H19)

Green			
Green[3]	Green[2]	Green[1]	Green[0]
(J21)	(K17)	(J17)	(H22)

Blue			
Blue[3]	Blue[2]	Blue[1]	Blue[0]
(K18)	(J22)	(K21)	(K22)

Sync	
HSync	VSynC
(L21)	(L22)

Aufgabe 1

Implementieren Sie in VHDL einen Dekoder für die Umwandlung einer 4-Bit-Binärzahl in die 7-Segment-Darstellung einer Hexadezimalziffer. Die Position der Segmente a bis g können Sie der Beschreibung des Praktikumsboards entnehmen. Beachten Sie, dass die Ansteuerung der Segmente low-aktiv erfolgt.

Für die Implementierung ist es zweckmäßig statt 8 Einzelsignalen (a bis g sowie Dezimalpunkt) einen 8-Bit-Signalvektor als Ausgangssignal vorzusehen. **Hinweis:** Nutzen Sie `case`- oder `select`-Statements zur Beschreibung des Dekoders.

Für die Eingabe der Binärzahl sind die Schiebeschalter SW3 bis SW0 zu nutzen. Steuern Sie nur das rechte Segment der 4-stelligen Anzeige an.

Hinweis: Die Aufgabe ist als Schaltnetz (ohne Taktsignal) zu lösen.

Überprüfen Sie die korrekte Funktion des Dekoders auf dem Praktikumsboard. Werten Sie die benötigten FPGA-Ressourcen im Praktikumsprotokoll aus.

1.2. Aufgabe 1 - Binär-Dekoder

1.2.1. Entwurf

Input 4-Bit Binärzahl durch Schieberegister SW3 ... SW0

Output 7-Segmente Darstellung einer Hexadezimalziffer (8 Einzelsignale = 7 Segmente + 1 Punkt)

Input				Output								
SW3	SW2	SW1	SW0	HEX	A	B	C	D	E	F	G	DOT
0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	1	1	1	0	0	1	1	1	1	1
0	0	1	0	2	0	0	1	0	0	1	0	1
0	0	1	1	3	0	0	0	0	1	1	0	1
0	1	0	0	4	1	0	0	1	1	0	0	1
0	1	0	1	5	0	1	0	0	1	0	0	1
0	1	1	0	6	0	1	0	0	0	0	0	1
0	1	1	1	7	0	0	0	1	1	1	1	1
1	0	0	0	8	0	0	0	0	0	0	0	1
1	0	0	1	9	0	0	0	0	1	0	0	1
1	0	1	0	A	0	0	0	1	0	0	0	1
1	0	1	1	b	1	1	0	0	0	0	0	1
1	1	0	0	C	0	1	1	0	0	0	1	1
1	1	0	1	d	1	0	0	0	0	1	0	1
1	1	1	0	E	0	1	1	0	0	0	0	1
1	1	1	1	F	0	1	1	1	0	0	0	1

1.8.1 Decoder.vhdl Code

1.2.2. Auswertung

Ressourcenbedarf

- 7 Logik-Elemente

- 12 Pins

Aufgabe 2

Implementieren Sie in VHDL ein Schaltnetz zur Bestimmung der Hamming-Distanz zweier 4-Bit-Worte.

Für die Ein- und Ausgabe sind zu verwenden:

Wort 1	Schiebeschalter SW3 bis SW0
Wort 2	Schiebeschalter SW7 bis SW4
Ergebnis	rechte Ziffer des 7-Segment-Blockes

Überprüfen Sie die korrekte Funktion des Schaltnetzes auf dem Praktikumsboard. Werten Sie die benötigten FPGA-Ressourcen im Praktikumsprotokoll aus.

1.3. Aufgabe 2 - Hamming-Distanz

1.3.1. Entwurf

Input 2 4-Bit Werte

- 1.Wert: 4-Bit Binärzahl durch Schieberegister SW3 ... SW0
- 2.Wert: 4-Bit Binärzahl durch Schieberegister SW7 ... SW4

Output 7-Segmente Darstellung einer Hexadezimalziffer (8 Einzelsignale = 7 Segmente + 1 Punkt)

Ansatz SW3 ... SW0 und SW7 ... SW4 logisch xor verknüpfen und Ergebnis direkt auf 7-Segmente Anzeige mappen

(1.8.2 Hamming.vhdl Code)

1.3.2. Auswertung

Ressourcenbedarf

- 9 Logik-Elemente
- 16 Pins

Aufgabe 3

Entwickeln Sie einen Modulo- n -Zähler, der bei einer Taktung mit 50 MHz einen Impuls pro Sekunde erzeugt. Die Impulslänge soll 1 Taktperiode betragen. Nutzen Sie die so erzeugte Impulsfolge zum periodischen Linksrotieren eines zyklischen 10-Bit-Schieberegisters um eine Bitstelle pro Sekunde.

Hinweise:

- Alle Register sind synchron mit ein- und demselben Taktsignal zu takten.
- Verwenden Sie einen additiven Operator zur Realisierung des Zählers.
- Bei einem Reset ist das Schieberegister mit einer ,1' an der rechten Stelle zu initialisieren.
- Realisieren Sie die Rotation durch eine geeignete Konkatenation.

Für die Ein- und Ausgabe sind zu verwenden:

Takt	50 MHz
Reset	Schiebeschalter SW0
Schieberegister	LED-Zeile

- Bei welchem Zählerstand ist der Zähler zurückzusetzen? Wann muss die Ausgabe des Impulses erfolgen?
- Welcher Funktion entspricht der Impuls aus Sicht des Schieberegisters?
- Implementieren Sie das Schaltwerk in VHDL. Überprüfen Sie die korrekte Funktion des Impulses und des Schieberegisters im Simulator.
- Überprüfen Sie die korrekte Funktion des Schaltwerks auf dem Praktikumsboard. Werten Sie die benötigten FPGA-Ressourcen und die maximale Taktfrequenz im Praktikumsprotokoll aus.

1.4. Aufgabe 3 - Modulo- n -Zähler

1.4.1. Entwurf

- Der Zähler ist nach 50 Millionen Schritten zurückzusetzen (50 MHz Takt entspricht 50 Millionen Taktperioden pro Sekunde)
- Für das Schieberegister ist der Zählerzustand ein Enable-Signal
-

Input

- 50MHz Takt
- Reset (Schiebeschalter SW0)

Output LED-Zeile

Ansatz 2 Komponenten: Schieber und Zähler

Zähler gibt alle 50-Millionen Taktperioden (50MHz Takt ergibt $50 \cdot 10^6$ Taktschritte pro Sekunde) einen Takt lang ein enable-Signal aus. (1.8.3 Zaehler.vhdl-Code)

Schieber beinhaltet den Zähler als Komponente und verschiebt bei dessen enable-Signal die LED-Anzeige um eine Stelle pro Takt. (1.8.3 Schieber.vhdl-Code)

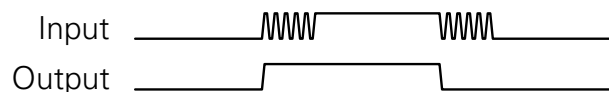
1.4.2. Auswertung

Ressourcenbedarf

- 60 Logik-Elemente
- davon 38 dedizierte Logik-Elemente
- 12 Pins
- maximale Taktfrequenz von 250 MHz

Aufgabe 4

Die Taster des Praktikumsboards sind – abhängig von der Revision des Boards – nicht entprellt. Ihre Prelldauer beträgt bis zu 3 ms.



Waveform eines prellenden Eingangssignals und des entsprechenden entprellten Ausgangssignals.

Entwickeln Sie – unabhängig davon, ob ihr Board entprellte Taster besitzt – einen Automaten, der mit Hilfe eines Zählers nach einer am Eingang vom Taster erkannten Flanke für diese Dauer alle weiteren ignoriert und so ein entsprechend entprelltes Ausgangssignal liefert. Der Zähler ist hierbei mit dem Boardtakt von 50 MHz zu takten. Nutzen Sie diesen Entprellautomaten zum sicheren Ein- und Ausschalten einer LED jeweils durch den Druck desselben Tasters. Nutzen Sie für die Ansteuerung der LED einen zweiten Automaten.

Für die Ein- und Ausgabe sind zu verwenden:

Takt	50 MHz
Eingabe	Taster BTN2
Ausgabe	LED0

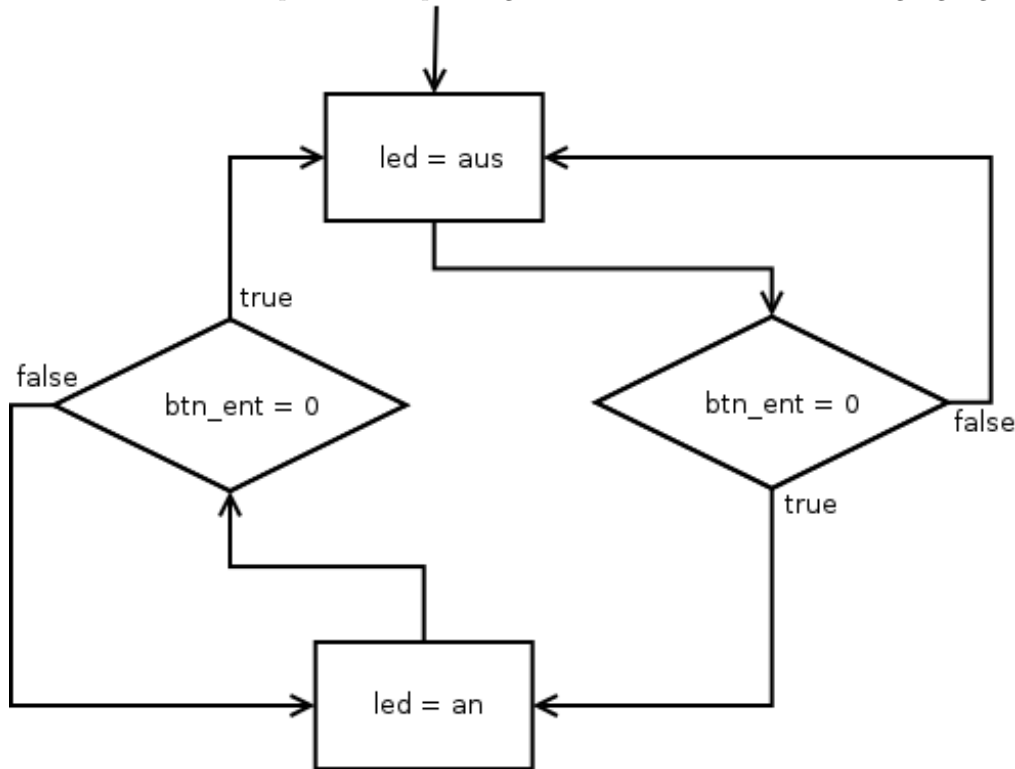
- Erstellen Sie jeweils State-Machine-Charts für den Entprellautomaten und den LED-Automaten. Abfrage und Steuerung des Zählers erfolgt durch den Entprellautomaten mittels geeigneter selbstdefinierter Signale.
- Welcher Typ von Automatenkopplung ist zu verwenden? Über welche Signale erfolgt die Kopplung?
- Implementieren Sie beide Automaten als getrennte VHDL-Module und überprüfen Sie die korrekte Funktion mittels Simulation.
- Implementieren Sie ein Top-Level-Modul welches beide Automaten miteinander koppelt. Überprüfen Sie die korrekte Funktion des Schaltwerks auf dem Praktikumsboard. Werten Sie die benötigten FPGA-Ressourcen und die maximale Taktfrequenz im Praktikumsprotokoll aus.

1.5. Aufgabe 4 - Entprell-Automat

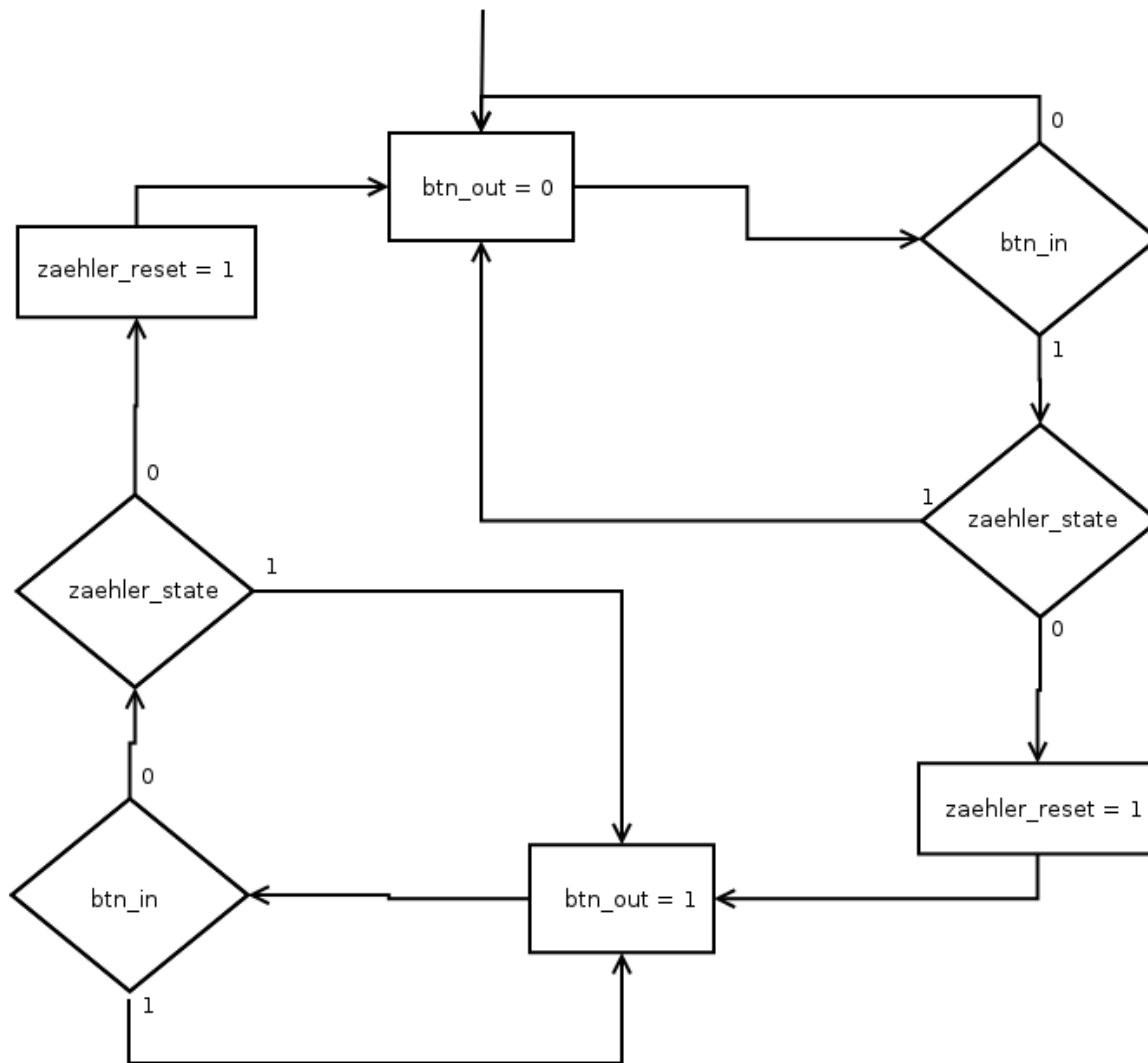
1.5.1. Entwurf

State-Machine-Charts

LED enthält die Komponente Entprellung und verbindet die Ein- und Ausgangssignale. (1.8.4 LED.vhdl-Code)



Entprellung enthält die Komponente Zaehler, der bei der Veränderung des Eingangssignals gestartet wird und für 3ms weitere Änderungen ignoriert. (1.8.4 Entprellung.vhdl-Code)



Zähler implementiert einen Zähler, der durch ein Signal definierte Schritte zählt. Ausgegeben wird der aktuelle Zustand des Zählers. Eingeben ein Reset-Signal. (1.8.4 Zähler.vhdl-Code)

Kopplung

Es wird eine synchrone Automatenkopplung über die Ausgangssignale mit einem Taktsignal verwendet.

1.5.2. Auswertung

Ressourcenbedarf

- 86 Logik-Elemente
- davon 79 dedizierte Logik-Elemente
- 44 Register
- 3 Pins
- maximale Taktfrequenz von 178 MHz

Aufgabe 5

Entwickeln Sie eine Multiplex-Ansteuerung für die 4-stellige 7-Segment-Anzeige des Erweiterungsboards um eine längere Zeichenkette auszugeben. Dabei soll jede halbe Sekunde der Text eine Stelle nach links verschoben werden.

Implementieren sie einen Zähler welcher Impulse mit geeigneter Frequenz ausgibt. Verwenden Sie für den Zähler den 50 MHz-Takt.

Zeigen Sie die Zeichenkette „HALLO“ an, wobei beginnend mit vollständig leerer Anzeige die Zeichenkette von rechts nach links die vier Stellen der Anzeige durchlaufen soll. Ist das letzte Zeichen nach links aus der Anzeige gewandert soll die Zeichenkette erneut ausgegeben werden.

Durch die Betätigung des Resets sollen alle vier 7-Segment-Blöcke gelöscht und danach wieder mit der Anzeige des ersten Zeichens der Zeichenkette in der rechten Stelle begonnen werden

Für die Ein- und Ausgabe sind zu verwenden:

Takt	50 MHz
Reset	Schiebeschalter SW0
Ausgabe	7-Segment-Block

Gliedern sie ihren Systementwurf mindestens in drei Module: Top-Level, Decoder für Textzeichen und Multiplexer für den anzuzeigenden Teil der Zeichenkette.

- Wie viele Bits werden für die Kodierung eines Textzeichens benötigt? Kodieren Sie die Textzeichen mit einem eigenen Code und dokumentieren Sie diesen im Protokoll!
- Implementieren Sie Decoder und Multiplexer als getrennte VHDL-Module und überprüfen Sie die korrekte Funktion mittels Simulation.
- Implementieren Sie ein Top-Level-Modul welches beide Automaten miteinander koppelt. Überprüfen Sie die korrekte Funktion des Schaltwerks auf dem Praktikumsboard. Werten Sie die benötigten FPGA-Ressourcen und die maximale Taktfrequenz im Praktikumsprotokoll aus.

1.6. Aufgabe 5 - HALLO-Anzeige

1.6.1. Entwurf

zu a) Es müssen 5 Zeichen kodiert werden (H, A, L, O, Leerzeichen).

$$\lg 5 = 3$$

Daher werden für eine Binärkodierung mindestens 3 Bits benötigt.

Input			Output								
BIT2	BIT1	BIT0	CHAR	A	B	C	D	E	F	G	DOT
0	0	0		1	1	1	1	1	1	1	1
0	0	1	H	1	0	0	1	0	0	0	1
0	1	0	A	0	0	0	1	0	0	0	1
0	1	1	L	1	1	1	0	0	0	1	1
1	0	0	O	0	0	0	0	0	0	1	1

- Für das Schieberegister ist der Zählerzustand ein Enable-Signal
- (1.8.5 Hallo.vhdl-Code)

1.6.2. Auswertung

Ressourcenbedarf

- 73 Logik-Elemente
- 61 Register
- 34 Pins
- maximale Taktfrequenz von 262 MHz

Aufgabe 6

Implementieren Sie eine auf Zehntelsekunden genaue Stoppuhr mit dem Boardtakt von 50 MHz als Referenz. Das Starten und Anhalten der Stoppuhr soll durch den Druck eines zu entprellenden Tasters ausgelöst werden. Das Rücksetzen der Stoppuhr soll durch das globale Reset erfolgen. Der aktuelle Stand der Stoppuhr ist dezimal auf dem 4-stelligen 7-Segment-Block auszugeben. Dabei sind eine Stelle für die Minutenzählung, zwei für die Sekunden und die verbleibende für die Zehntelsekunden vorzusehen. Verwenden Sie die Dezimalpunkte zur passenden optischen Unterteilung der Anzeige. Die Zeitmessung soll, entsprechend der verfügbaren Stellenzahl, „modulo 10 Minuten“ erfolgen.

Hinweise:

- Nutzen Sie einen BCD-Zähler pro Stelle mit einem entsprechenden Wertebereich.
- Kaskadieren Sie die BCD-Zähler mit Hilfe von Übertragsimpulsen.

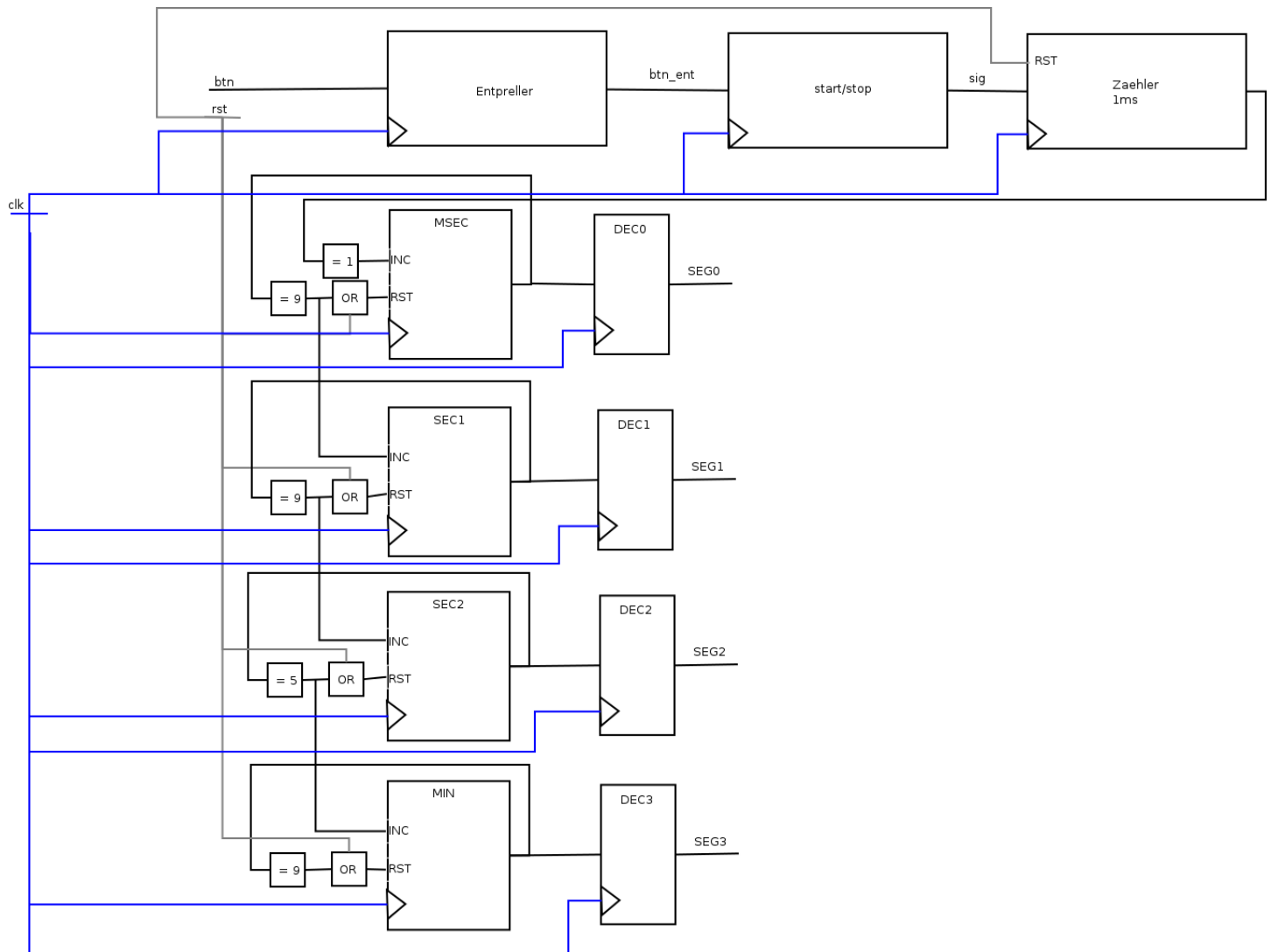
Für die Ein- und Ausgabe sind zu verwenden:

Takt	50 MHz
Reset	Schiebeschalter SW0
Start/Stopp	Taster BTN2
Ausgabe	7-Segment-Anzeige

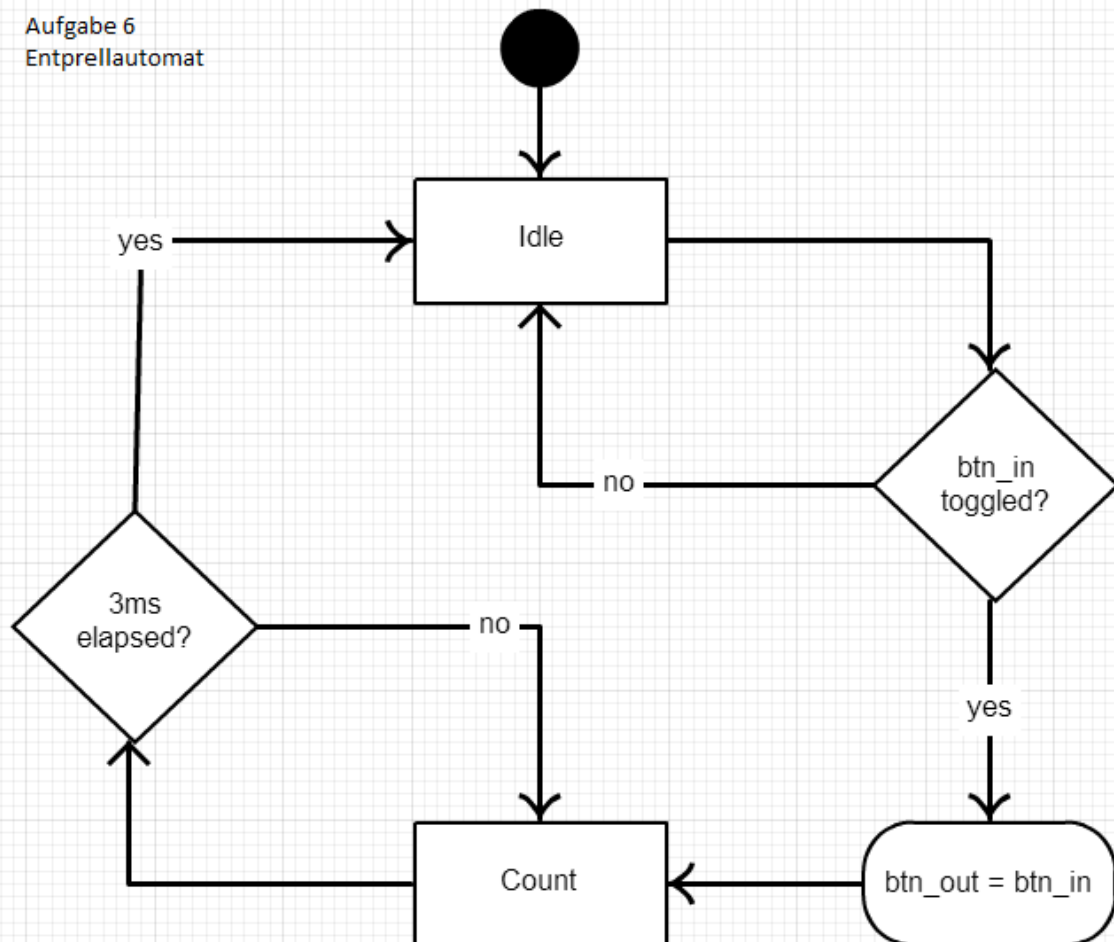
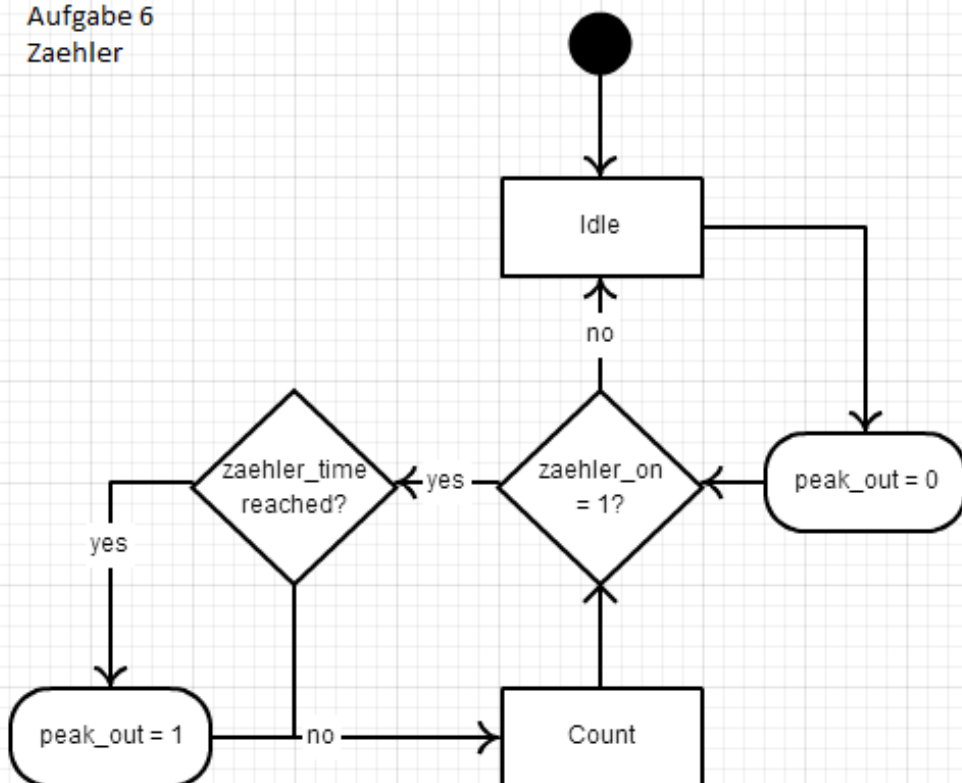
- Überlegen Sie sich eine geeignete Zerlegung des Gesamtsystems in Teilkomponenten und spezifizieren Sie die deren Schnittstellen. Besprechen Sie kurz ihre Lösung mit dem Praktikumsbetreuer.
- Welcher Typ von Automatenkopplung ist zu verwenden? Über welche Signale erfolgt die Kopplung?
- Implementieren Sie die Teilkomponenten in VHDL. Erstellen Sie dazu für jeden Automaten ein State-Machine-Chart. Überprüfen Sie die korrekte Funktion jeder Teilkomponente mittels Simulation.
- Implementieren Sie ein Top-Level-Modul welches alle Komponenten miteinander verbindet. Überprüfen Sie die korrekte Funktion der Schaltung auf dem Praktikumsboard. Werten Sie die benötigten FPGA-Ressourcen und die maximale Taktfrequenz im Praktikumsprotokoll aus.

1.7. Aufgabe 6 - Stoppuhr

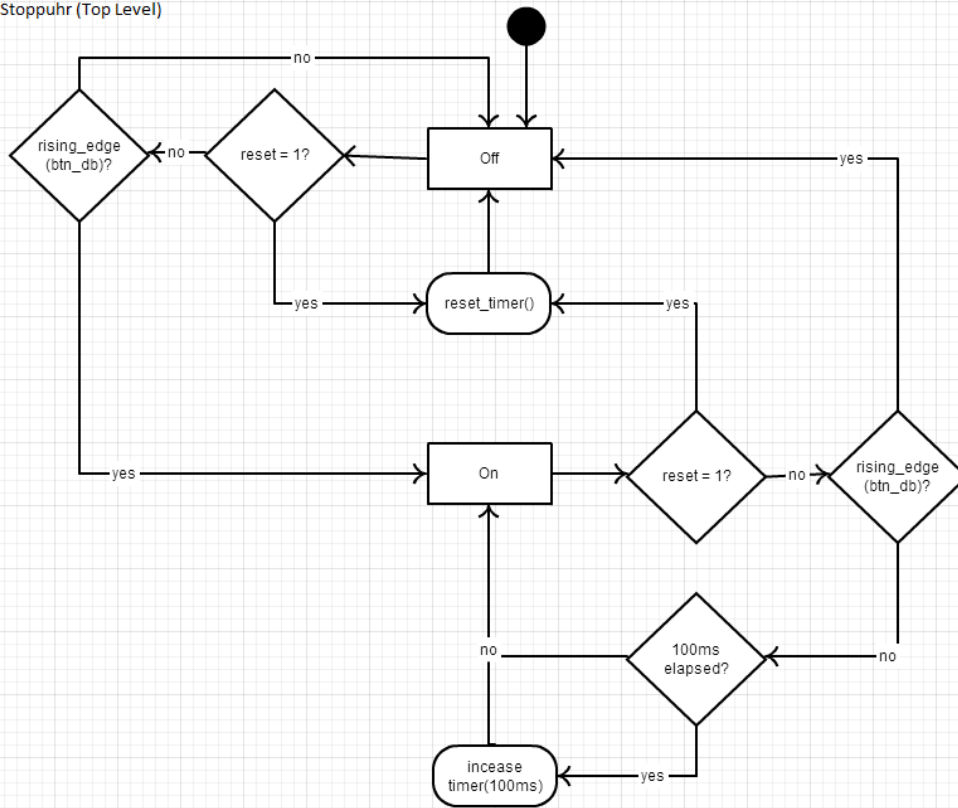
1.7.1. Entwurf



State-Machine-Charts

Aufgabe 6
EntprellautomatAufgabe 6
Zaehler

Aufgabe 6
Stoppuhr (Top Level)



Kopplung

Es wird eine synchrone Automatenkopplung über die Ausgangssignale mit einem Taktsignal verwendet.

1.7.2. Auswertung

Ressourcenbedarf

- 163 Logik-Elemente
- davon 121 dedizierte Logik-Elemente
- 35 Pins
- maximale Taktfrequenz von 225 MHz

1.8. Anhang

1.8.1. 01-Aufgabe Code

Listing 1.1: VHDL-Code Decoder.vhdl

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity Decoder is
6
7     port (
8         sw : in std_logic_vector(3 downto 0);
9         cc : out std_logic_vector(7 downto 0));
10
11 end Decoder;
12

```

```

13 architecture Dec1 of Decoder is
14 begin
15
16 -----
17 -- Outputs: 4 bit breites Wort in Hexadezimale 7-Segment Anzeige
18 -----
19 with sw select
20   cc <= "00000011" when "0000",
21       "10011111" when "0001",
22       "00100101" when "0010",
23       "00001101" when "0011",
24       "10011001" when "0100",
25       "01001001" when "0101",
26       "01000001" when "0110",
27       "00011111" when "0111",
28       "00000001" when "1000",
29       "00001001" when "1001",
30       "00010001" when "1010",
31       "11000001" when "1011",
32       "01100011" when "1100",
33       "10000101" when "1101",
34       "01100001" when "1110",
35       "01110001" when "1111";
36 end dec1;

```

1.8.2. 02-Aufgabe Code

Listing 1.2: VHDL-Code Hamming.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity Hamming is
6
7      port (
8          sw1 : in std_logic_vector(3 downto 0); -- Erstes 4bit Wort
9          sw2 : in std_logic_vector(3 downto 0); -- Zweites 4bit Wort
10         cc : out std_logic_vector(7 downto 0)); -- 7 Segment Ausgabe
11
12 end Hamming;
13
14 architecture ham1 of Hamming is
15     signal xo : std_logic_vector(3 downto 0);
16 begin
17
18     xo <= sw1 xor sw2; -- Jede Stelle nur 1, wenn sich die Woerter an der Stelle unterscheiden
19
20     -- Je nach Anzahl der Einsen im Signal "xo" wird die Ausgabe 0-4 ausgegeben.
21     with xo select
22       cc <= "00000011" when "0000",
23           "10011111" when "0001",
24           "10011111" when "0010",
25           "00100101" when "0011",
26           "10011111" when "0100",
27           "00100101" when "0101",
28           "00100101" when "0110",
29           "00001101" when "0111",
30           "10011111" when "1000",
31           "00100101" when "1001",
32           "00100101" when "1010",
33           "00001101" when "1011",
34           "00100101" when "1100",
35           "00001101" when "1101",
36           "00001101" when "1110",
37           "10011001" when "1111";

```

```
38 end ham1;
```

1.8.3. 03-Aufgabe Code

Listing 1.3: VHDL-Code Schieber.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity Schieber is
7
8      port (
9          clk : in std_logic;
10         rst : in std_logic;
11         ld : out std_logic_vector(9 downto 0)
12     );
13
14
15 end Schieber;
16
17
18 architecture schieb1 of Schieber is
19     component zaehler          -- komponente zaehler in architektur einbinden
20     port (
21         clk : in std_logic;
22         clk_out : out std_logic
23     );
24     end component;
25     signal state : std_logic_vector(9 downto 0) := "0000000001"; -- initialanzeige der led-zeile
26     signal shift : std_logic;
27
28 begin
29     custom_clk : zaehler PORT MAP (clk => clk, clk_out => shift);
30
31     process(clk)
32     begin
33
34         if rising_edge(clk) then
35             if rst = '1' then          -- bei reset zustand der led-zeile zuruecksetzen
36                 state <= "0000000001";
37             elsif shift = '1' then      -- kein reset und signal vom zaehler
38                 state <= state(8 downto 0)&state(9);    -- zustand der linken led, rechts wieder einfuegen
39             end if;
40         end if;
41     end process;
42
43     ld <= state;
44
45 end schieb1;
```

Listing 1.4: VHDL-Code Zaehler.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity Zaehler is
6
7      port (
8          clk : in std_logic;
9          clk_out : out std_logic
10     );
11
12
```

```

13 end Zaehler;
14
15 architecture zael of Zaehler is
16
17     signal counter : unsigned(26 downto 0) := (others => '0'); -- Zaehler mod 50.000.000
18     signal state : std_logic := '1';      -- zaehler-zustand (1 => fertig, 0 => zaehlt)
19 begin
20
21     process(clk, state, counter)
22     begin
23
24         if rising_edge(clk) then
25
26             state <= '0';
27             if counter = to_unsigned(50000000, counter'length) then -- prueft ob counter == 50 mio
28                 counter <= (others => '0');      -- falls true, ist 1 sekunde verstrichen -> counter reset
29                 state <= '1';      -- zaehler-zustand auf fertig setzen
30
31             else
32                 counter <= counter + 1;      -- sonst weiterzaehlen
33             end if;
34
35         end if;
36     end process;
37
38     clk_out <= state;
39 end zael;

```

1.8.4. 04-Aufgabe Code

Listing 1.5: VHDL-Code LED.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity LED is
7
8      port (
9          clk : in std_logic;
10         btn : in std_logic;
11         ld : out std_logic
12     );
13 end LED;
14
15 architecture led1 of LED is
16     -- Komponente Entprellung fuer das btn-Signal wird eingebunden
17     component Entprellung
18     port (
19         clk : in std_logic;
20         btn_in : in std_logic;
21         btn_out : out std_logic
22     );
23 end component;
24
25     signal led_sig : std_logic := '0';
26     signal btn_led : std_logic;
27     signal btn_out : std_logic;
28     signal btn_out_d : std_logic;
29     signal btn_in_d : std_logic := '0';
30     signal btn_in : std_logic := '0';
31
32     begin
33         custom_entpreller : Entprellung PORT MAP (

```

```

34         clk => clk,
35         btn_in => btn_in,
36         btn_out => btn_out);
37 process (btn_led )
38 begin
39     if btn_led = '0' then -- aenderung der led bei uebergang des entprellten btn-signals in den aktiven
        zustand
40         led_sig <= not led_sig;
41     end if;
42 end process;
43
44 -- synchronisierung der signale
45 process (clk)
46 begin
47     if rising_edge(clk) then
48         btn_in <= btn_in_d;
49         btn_in_d <= btn;
50         btn_out_d <= btn_out;
51         btn_led <= btn_out_d;
52     end if;
53 end process;
54
55 ld <= not led_sig;
56 end led1;

```

Listing 1.6: VHDL-Code Entprellung.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6
7  entity Entprellung is
8
9      port (
10         clk : in std_logic;
11         btn_in : in std_logic;
12         btn_out : out std_logic
13     );
14
15 end Entprellung;
16
17 architecture entprell1 of Entprellung is
18
19     signal btn_old : std_logic := '0';
20     signal state : std_logic := '0';
21
22     signal zaehler_state : std_logic := '0';
23     signal zaehler_state_d : std_logic := '0';
24     signal zaehler_reset : std_logic := '0';
25
26
27
28     -- Zaehler-Komponente wird eingebunden
29     component Zaehler
30     port (
31         clk : in std_logic;
32         count_steps : in unsigned(31 downto 0);
33         counter_reset : in std_logic;
34         counter_state : out std_logic
35     );
36 end component;
37
38 begin

```

```

39 custom_zaeher : Zaeher PORT MAP (
40     clk => clk,
41     count_steps => to_unsigned(150000, 32),
42     -- zu zaehlende Schritte, bis deaktivierung des counter_state signals
43     counter_state => zaehler_state,
44     counter_reset => zaehler_reset);
45
46 -- entprellung des eingangsignals btn_in
47 process(clk)
48 begin
49     if rising_edge(clk) then
50         if state = '0' then -- falls ausserhalb der prelldauer
51             if btn_in /= btn_old then -- falls das eingangssignal sich aendert, wird der entpreller gestartet
52                 zaehler_reset <= '1';
53                 btn_old <= btn_in;
54             end if;
55         else
56             -- zaehler_reset soll nur einen takt aktiv sein
57             if zaehler_reset = '1' then
58                 zaehler_reset <= '0';
59             end if;
60         end if;
61     end if;
62 end process;
63
64 -- synchronisierung des zaehler-zustands
65 process (clk)
66 begin
67     if rising_edge(clk) then
68         state <= zaehler_state_d;
69         zaehler_state_d <= zaehler_state;
70     end if;
71 end process;
72
73 -- ausgabe des entprellten signals
74 btn_out <= btn_old;
75 end entprell;

```

Listing 1.7: VHDL-Code Zaeher.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity Zaeher is
7
8      port (
9          clk : in std_logic;
10         count_steps : in unsigned(31 downto 0); -- zu zaehlende taktschritte, bis zur deaktivierung des
            counter_state signals
11         counter_reset : in std_logic;
12         counter_state : out std_logic);
13
14
15 end Zaeher;
16
17 architecture zae1 of Zaeher is
18
19     signal reset : std_logic := '0';
20     signal reset_d : std_logic := '0';
21     signal state : std_logic := '0';
22     signal counter : unsigned(31 downto 0) := (others => '0');
23 begin
24
25     process(clk)

```

```

26   begin
27
28       if rising_edge(clk) then
29           if reset = '1' then -- zuruecksetzen des zaehlers
30               counter <= (others => '0');
31           end if;
32
33           if counter < count_steps then -- zaehler aktiv
34               state <= '1';
35               counter <= counter + 1;
36           else
37               state <= '0';
38           end if;
39       end if;
40   end process;
41
42   -- synchronisierung des reset-signals
43   process(clk)
44   begin
45       if rising_edge(clk) then
46           reset_d <= counter_reset;
47           reset <= reset_d;
48       end if;
49   end process;
50
51
52   counter_state <= state;
53 end zael;

```

1.8.5. 05-Aufgabe Code

Listing 1.8: VHDL-Code hallo.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  -----
7  -- Top Level:
8  -- Verbindet den Multiplexer mit 4 Decodern und legt den Ausgang
9  -- je eines Decoders an eine 7 Segment Anzeige
10 -----
11 entity Hallo is
12
13     port (
14         clk : in std_logic;
15         rst : in std_logic;
16         seg1 : out std_logic_vector(7 downto 0); -----
17         seg2 : out std_logic_vector(7 downto 0); -- 7 Segment
18         seg3 : out std_logic_vector(7 downto 0); -- Ausgaenge
19         seg4 : out std_logic_vector(7 downto 0)); -----
20
21 end Hallo;
22
23 architecture hello of hallo is
24
25     component Multiplex
26     port (
27         clk : in std_logic;
28         rst : in std_logic;
29         led_out : out std_logic_vector(11 downto 0)
30     );
31 end component;
32
33 component Decoder

```

```

34     port (
35         clk : in std_logic;
36         code : in std_logic_vector(2 downto 0);
37         decoded : out std_logic_vector(7 downto 0));
38
39     end component;
40
41     signal dig : std_logic_vector(11 downto 0); -- nimmt 12bit Wort aus dem Multiplexer entgegen
42     signal dig0 : std_logic_vector(2 downto 0); -----
43     signal dig1 : std_logic_vector(2 downto 0); -- 4*3bit die je ein Zeichen aus dem 12bit
44     signal dig2 : std_logic_vector(2 downto 0); -- Wort des Multiplexers abzweigen
45     signal dig3 : std_logic_vector(2 downto 0); -----
46
47 begin
48
49     -----
50     -- Multiplexer Ausgang wird an das Signal "dig" angelegt
51     -- Je ein Signal mit je einem Zeichen wird als Eingang eines Decoders angelegt
52     -----
53     mult : Multiplex PORT MAP( clk => clk, rst => rst, led_out => dig);
54     dec0 : Decoder PORT MAP(clk => clk, code => dig0, decoded => seg4);
55     dec1 : Decoder PORT MAP(clk => clk, code => dig1, decoded => seg3);
56     dec2 : Decoder PORT MAP(clk => clk, code => dig2, decoded => seg2);
57     dec3 : Decoder PORT MAP(clk => clk, code => dig3, decoded => seg1);
58
59     -- abzweigen von je 3bit (ein Zeichen) aus dem Multiplexer Ausgang
60     dig0 <= dig(11 downto 9);
61     dig1 <= dig(8 downto 6);
62     dig2 <= dig(5 downto 3);
63     dig3 <= dig(2 downto 0);
64
65 end hello;

```

Listing 1.9: VHDL-Code Decoder.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity Decoder is
7
8      port (
9          clk : in std_logic;
10         code : in std_logic_vector(2 downto 0);
11         decoded : out std_logic_vector(7 downto 0));
12 end Decoder;
13
14 architecture decoder1 of Decoder is
15
16     signal decoded_out : std_logic_vector(7 downto 0) := (others => '0'); -- Signal, dass an den Ausgang
17                                     "decoded" angelegt wird
18
19 begin
20     -----
21     -- Dekodieren eines 3bit breiten Wortes in ein 8bit
22     -- breites Wort fuer die 7 Segment Anzeige
23     -----
24     process (clk)
25     begin
26         if rising_edge(clk) then
27             case code is
28                 when "000" => decoded_out <= "11111111"; -- _
29                 when "001" => decoded_out <= "10010001"; -- H
30                 when "010" => decoded_out <= "00010001"; -- A

```



```

31         when "011" => decoded_out <= "11100011"; -- L
32         when "100" => decoded_out <= "00000011"; -- 0
33         when others => decoded_out <= "11111111";
34     end case;
35 end if;
36 end process;
37
38 decoded <= decoded_out;
39
40 end decoder1;

```

Listing 1.10: VHDL-Code Multiplex.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity Multiplex is
7
8      port (
9          clk : in std_logic;
10         rst : in std_logic;
11         led_out : out std_logic_vector(11 downto 0) -- 12bit breiter Ausgang (3bit je Zeichen)
12     );
13
14 end Multiplex;
15
16 -- 000: _
17 -- 001: H
18 -- 010: A
19 -- 011: L
20 -- 100: 0
21
22 architecture multi of Multiplex is
23     -- _ _ _ _ H A L L O _ _ _
24     signal tex : std_logic_vector(35 downto 0) := "000000000000001010011011100000000000"; -- kompletter
        Schriftzug der einmal durchlaufen wird
25     signal counter : unsigned(24 downto 0) := (others => '0'); -- Zaehlersignal (mod 25.000.000)
26     signal mul : unsigned(3 downto 0); -- Steuersignal Multiplexer (mod 10) : 9 moegliche 12bit breite
        Teilworte des kompletten Schriftzugs
27
28 begin
29
30     -----
31     -- Inkrementieren des Steuersignals "mul" alle 25.000.000 Takte
32     -----
33     process(clk)
34     begin
35         if rising_edge(clk) then
36             if(rst = '1') then
37                 counter <= (others => '0');
38                 mul <= (others => '0');
39             elsif(counter = "10111110101111000000111111") then
40                 counter <= (others => '0');
41                 mul <= mul + 1;
42                 if(mul = "1000") then
43                     mul <= "0000";
44                 end if;
45             else
46                 counter <= counter + 1;
47             end if;
48         end if;
49     end process;
50
51     -----

```

```

52  -- Je nach Steuersignal "mul" wird ein anderes 12bit breites Teilwort des kompletten Schriftzugs
53      ausgegeben
54  -----
55  with mul select
56      led_out <= tex(35 downto 24) when "0000",
57                tex(32 downto 21) when "0001",
58                tex(29 downto 18) when "0010",
59                tex(26 downto 15) when "0011",
60                tex(23 downto 12) when "0100",
61                tex(20 downto 9)  when "0101",
62                tex(17 downto 6)  when "0110",
63                tex(14 downto 3)  when "0111",
64                tex(11 downto 0)  when "1000",
65                (others => '0') when others;
66  end multi;

```

1.8.6. 06-Aufgabe Code

Listing 1.11: VHDL-Code Stoppuhr.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity Stoppuhr is
7
8      port (
9          clk : in std_logic;
10         rst : in std_logic;
11         onoff : in std_logic;
12         seg1 : out std_logic_vector(7 downto 0);
13         seg2 : out std_logic_vector(7 downto 0);
14         seg3 : out std_logic_vector(7 downto 0);
15         seg4 : out std_logic_vector(7 downto 0));
16
17  end Stoppuhr;
18
19  architecture uhr of Stoppuhr is
20
21      -- Zaehler: gibt jede Zehntelsekunde einen Peak aus
22      component Zaehler
23          port (
24              clk : in std_logic;
25              zaehler_time : in unsigned(31 downto 0);
26              zaehler_on : in std_logic;
27              peak_out : out std_logic
28          );
29      end component;
30
31      -- entprellt das Eingangssignal des (on/off) Buttons
32      component EntprellAutomat
33          port (
34              clk : in std_logic;
35              btn : in std_logic;
36              btnout : out std_logic
37          );
38      end component;
39
40      -- 4 Decoder: Je ein Decoder dekodiert eine Stelle der aktuellen Zeit fuer die 7-Segment Anzeige
41      component Decoder
42          port (
43              clk : in std_logic;
44              code : in std_logic_vector(3 downto 0);
45              decoded : out std_logic_vector(7 downto 0)

```

```

46     );
47 end component;
48
49 -- Steuersignal und Ausgabesignal des Zaehlers
50 signal timer_on, peak : std_logic := '0';
51
52 -- Je ein Signal fuer je eine Stelle der aktuellen Zeit
53 signal min, sec1, sec2, ms : unsigned(3 downto 0) := (others => '0');
54
55 -- Steuersignal fuer die Stoppuhr und Signale fuer den entprellten Button, sowie des alten Signalpegels
    des Buttons
56 signal running, onoff_db, onoff_old : std_logic := '0';
57
58 -- Signale fuer die 7-Segment Anzeige
59 signal segMin, segSec1, segSec2, segMs : std_logic_vector(7 downto 0) := (others => '0');
60
61
62 begin
63
64     zaehl : Zaehler PORT MAP (clk => clk,
65                             zaehler_time => to_unsigned(5000000, 32),
66                             zaehler_on => timer_on,
67                             peak_out => peak);
68
69     prell : EntprellAutomat PORT MAP (
70                                     clk => clk,
71                                     btn => onoff,
72                                     btnout => onoff_db);
73
74     -- Jeder Decoder dekodiert eine Stelle der aktuellen Zeit
75     dec0 : Decoder PORT MAP(clk => clk, code => std_logic_vector(min), decoded => segMin);
76     dec1 : Decoder PORT MAP(clk => clk, code => std_logic_vector(sec1), decoded => segSec1);
77     dec2 : Decoder PORT MAP(clk => clk, code => std_logic_vector(sec2), decoded => segSec2);
78     dec3 : Decoder PORT MAP(clk => clk, code => std_logic_vector(ms), decoded => segMs);
79
80
81 process(clk)
82 begin
83     if rising_edge(clk) then
84         if(rst = '1') then                -- aktiver Reset setzt alles zurueck und stoppt die Uhr
85             running <= '0';
86             timer_on <= '0';
87             min <= (others => '0');
88             sec1 <= (others => '0');
89             sec2 <= (others => '0');
90             ms <= (others => '0');
91         else
92             onoff_old <= onoff_db;
93             if(onoff_db = '1' and onoff_db /= onoff_old) then -----
94                 running <= not running;    -- on/off umschalten wenn btn gedrueckt
95                 timer_on <= not timer_on;   -----
96             end if;
97
98             if(running = '1') then          -- Wenn die Uhr laeuft...
99                 if(peak = '1') then        -- ...und der Zaehler einen Peak ausgibt...
100                     if(ms = to_unsigned(9, 4)) then -----
101                         if(sec2 = to_unsigned(9, 4)) then --
102                             if(sec1 = to_unsigned(5, 4)) then --
103                                 if(min = to_unsigned(9, 4)) then --
104                                     min <= (others => '0'); --
105                                     sec1 <= (others => '0'); --
106                                     sec2 <= (others => '0'); --
107                                     ms <= (others => '0'); --
108                                 else --
109                                     min <= min + 1;    -- ...erhoehe die aktuelle Zeit
110                                     sec1 <= (others => '0'); -- um eine Zehntelsekunde (mod 10 Minuten)

```

```

111         sec2 <= (others => '0'); --
112         ms <= (others => '0'); --
113     end if; --
114     else --
115         sec1 <= sec1 + 1; --
116         sec2 <= (others => '0'); --
117         ms <= (others => '0'); --
118     end if; --
119     else --
120         sec2 <= sec2 + 1; --
121         ms <= (others => '0'); --
122     end if; --
123     else --
124         ms <= ms + 1; --
125     end if; --
126 end if; --
127 end if; -----
128
129 end if;
130 end if;
131 end process;
132
133 seg4 <= segMin and "11111110"; -- Punkt der 7-Segment Anzeige aktivieren
134 seg3 <= segSec1;
135 seg2 <= segSec2 and "11111110"; -- Punkt der 7-Segment Anzeige aktivieren
136 seg1 <= segMS;
137
138 end uhr;

```

Listing 1.12: VHDL-Code Decoder.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity Decoder is
7
8      port (
9          clk : in std_logic;
10         code : in std_logic_vector(3 downto 0);
11         decoded : out std_logic_vector(7 downto 0));
12 end Decoder;
13
14 architecture decoder1 of Decoder is
15
16     signal decoded_out : std_logic_vector(7 downto 0) := (others => '0'); -- Signal, dass an den Ausgang
17                                     "decoded" angelegt wird
18
19     begin
20
21         -----
22         -- Dekodieren eines 4bit breiten Wortes in ein 8bit
23         -- breites Wort fuer die 7 Segment Anzeige
24         -----
25     process (clk)
26     begin
27         if rising_edge(clk) then
28             case code is
29                 when "0000" => decoded_out <= "00000011"; -- 0
30                 when "0001" => decoded_out <= "10011111"; -- 1
31                 when "0010" => decoded_out <= "00100101"; -- 2
32                 when "0011" => decoded_out <= "00001101"; -- 3
33                 when "0100" => decoded_out <= "10011001"; -- 4
34                 when "0101" => decoded_out <= "01001001"; -- 5
35                 when "0110" => decoded_out <= "01000001"; -- 6

```

```

35     when "0111" => decoded_out <= "00011111"; -- 7
36     when "1000" => decoded_out <= "00000001"; -- 8
37     when "1001" => decoded_out <= "00001001"; -- 9
38     when others => decoded_out <= "11111111"; -- error
39   end case;
40   end if;
41 end process;
42
43 decoded <= decoded_out;
44
45 end decoder1;

```

Listing 1.13: VHDL-Code EntprellAutomat.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity EntprellAutomat is
7
8    port (
9      clk : in std_logic;
10     btn : in std_logic;
11     btnout : out std_logic
12    );
13
14 end EntprellAutomat;
15
16 architecture prell of EntprellAutomat is
17
18   component Zaehler
19     port (
20       clk : in std_logic;
21       zaehler_time : in unsigned(31 downto 0);
22       zaehler_on : in std_logic;
23       peak_out : out std_logic
24     );
25   end component;
26
27   type zustaende is (idle, count); -- 2 Zustaende, idle = button betaetigen moeglich, count = 3ms warten
28   -- (bis 150.000 hochzaehlen bei 50Mhz)
29   attribute enum_encoding : string;
30   attribute enum_encoding of zustaende : type is "1 0";
31   signal z_alt, z_neu : zustaende := idle; -- alter und neuer Zustand des Automaten
32   signal btn_s, btn_output : std_logic := '0'; -- Synchronisiertes (Eingangs)Buttonsignal und
33   -- Ausgangssignal des Automaten
34   signal btn_old : std_logic := '0'; -- Speichern des vorherigen "btn_s" Pegels
35   signal btn2 : std_logic := '1'; -- Synchronisierungssignal fuer den Button
36   signal timer_on : std_logic := '0'; -- Steuerung des externen Zaehler Moduls
37   signal peak : std_logic := '0'; -- Ausgabe des externen Zaehler Moduls
38
39 begin
40
41   -- enthaltener Zaehler, der (falls aktiviert) alle 3ms fuer einen Takt eine eins an das Signal "peak"
42   -- ausgibt
43   timer : Zaehler PORT MAP (clk => clk, zaehler_time => to_unsigned(150000,32), zaehler_on => timer_on,
44     peak_out => peak);
45
46   -- Synchronisieren des Eingangssignals (Button) und Speichern des alten "btn_s" Pegels
47   process(clk)
48   begin
49     if rising_edge(clk) then
50       btn2 <= btn;
51       btn_old <= btn_s;
52       btn_s <= not btn2;

```

```

49     end if;
50 end process;
51
52 -- Uebernehmen und berechnen des neuen Zustandes
53 process(clk)
54 begin
55     if rising_edge(clk) then
56         z_alt <= z_neu;
57         case z_alt is
58             when idle => if(btn_s /= btn_old) then -- Wechseln in "count" und starten des Zaehlers, sobald
                           sich btn_s aendert
59                 btn_output <= btn_s;
60                 z_neu <= count;
61                 timer_on <= '1';
62             end if;
63             when count => if(peak = '1') then -- Wechseln zurueck in "idle", sobald der Zaehler eine eins an
                           "peak" anlegt (3ms vergangen)
64                 z_neu <= idle;
65                 timer_on <= '0';
66             end if;
67         end case;
68     end if;
69 end process;
70
71 btnout <= btn_output;
72 end prell;

```

Listing 1.14: VHDL-Code Zaehler.vhdl

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity Zaehler is
7
8      port (
9          clk : in std_logic;
10         zaehler_time : in unsigned(31 downto 0); -- steuert wieviele Takte der Zaehler zaehlen soll, bis er
              einen peak ausgeben soll. (Zaehler zaehlt mod zaehler_time)
11         zaehler_on : in std_logic; -- Steuersignal, bei 1 laeuft der Zaehler, bei 0 wird der Zaehler gestoppt
              und zurueckgesetzt
12         peak_out : out std_logic -- gibt fuer einen Takt eine eins aus, sobald der durch "zaehler_time"
              angelegte Wert erreicht ist
13     );
14
15 end Zaehler;
16
17 architecture timer of Zaehler is
18
19     signal counter : unsigned(31 downto 0) := (others => '0'); -- Zaehlsignal
20     signal peak : std_logic := '0'; -- Signal, dass an den Ausgang "peak_out" gegeben wird
21
22 begin
23
24     -----
25     -- Liegt "zaehler_on" auf eins, wird das Signal "counter" inkrementiert.
26     -- Wird der durch "zaehler_time" angegebene Maximalwert erreicht wird fuer einen Takt
27     -- eine 1 an das Signal "peak" ausgegeben und "counter" auf 0 zurueckgesetzt
28     -- Liegt "zaehler_on" auf null, wird "counter" auf 0 gesetzt und nicht hochgezahlt.
29     -----
30
31 process(clk)
32 begin
33     if rising_edge(clk) then
34         peak <= '0';
35         if(zaehler_on = '1') then

```

```
35         if(counter = zaehler_time - 1) then
36             counter <= (others => '0');
37             peak <= '1';
38         else
39             counter <= counter + 1;
40         end if;
41     else
42         counter <= (others => '0');
43     end if;
44 end if;
45 end process;
46
47 peak_out <= peak;
48
49 end timer;
```

2. Entwurf eingebetteter Systeme: Schaltkreisvalidation

2.1. Programm

2.1.1. Entwurf

Circuit

Das Grundgerüst des Programms bildet die **Circuit**-Klasse, die einen Schaltkreis und seine Funktionalitäten implementiert. Die Eingangsbelegung des Schaltkreises wird durch eine Zustandsvariable **unsigned int state** repräsentiert, die einen Wert zwischen 0 und 2^N annehmen kann, wobei N die Anzahl der Eingänge ist.

Die Eingänge (**Input**-Objekte), Gatter (**Gate**-Objekte) und Ausgänge (**Output**-Objekte) werden in jeweils eigenen Maps abgelegt, deren Keys vom Typ **std::string** die Bezeichner der jeweiligen Objekte sind. Alle Gatter sind als Klassen implementiert, die von der abstrakten Klasse **Gate** erben. Bei manchen Gattern - wie AND, Or, usw. - können beliebig viele Eingänge definiert werden. Ein Gattereingang ist ein Zeiger, auf entweder den Wert eines **Input**-Objektes oder auf den Ausgang eines anderen **Gate**-Objektes. Die Ein- und Ausgänge der Gatter sind vom Typ **bool**.

Bei der Simulation eines Schaltkreises wird über den **state** der **Circuit**-Klasse iteriert und dessen Wert in Binärdarstellung auf die Eingänge abgebildet. Nach jeder neuen Eingangsbelegung müssen die Gatter durchlaufen werden, bis das neue Signal die Ausgänge erreicht.

Parser

Zum Parsen wird die Klasse **Parser** als Grundgerüst zur Verfügung gestellt, wobei die genauere Implementierung für unterschiedliche Benchmarks in von dieser Klasse erbenenden Klassen verschoben wurde. Für das Benchmark BENCH ist eine solche gleichnamige Klasse vorhanden.

Ein Parser kann Schaltkreise aus Dateien lesen und diese anschließend als **Circuit**-Objekte zurückgeben.

Der Parser für das BENCH-Format geht davon aus, dass in der einzulesenden Datei zuerst alle Eingänge (INPUT), dann die Ausgänge (OUTPUT), danach die ggf. vorhandenen FlipFlops (DFF) und zuletzt die restlichen Gatter definiert werden.

Um einen Schaltkreis aus einer Datei zu Parsen, müssen dem Programm die Argumente
-p BENCHMARK DATEI mitgeteilt werden.

Simulator

Die Klasse **Simulator** enthält eine Liste von geparsen **Circuit**-Objekten, die simuliert werden können. Dazu wird wieder über den Zustand der Schaltkreise iteriert.

Um zum Beispiel zwei Schaltkreise aus Dateien zu parsen und zu Simulieren müssen folgende Argumente übergeben werden:
-p BENCHMARK1 DATEI1 -p BENCHMARK2 DATEI2 -s

2.1.2. Äquivalenzprüfung durch Simulation

Voraussetzungen

- Schaltkreise müssen gleichviele Eingänge bzw. Ausgänge besitzen
- falls in einem Schaltkreis ein Eingang bzw. Ausgang vorkommt, muss ein gleichnamiger Eingang bzw. Ausgang auch in den anderen Schaltkreisen vorkommen
- es können beliebig viele Schaltkreise verglichen werden, falls einer nicht äquivalent mit einem anderen ist, wird **false** zurückgegeben
- ist kein Schaltkreis definiert, wird **false** zurückgegeben
- ist nur ein Schaltkreis definiert, wird **true** zurückgegeben

Vorgehen

1. über mögliche Zustände (2^N mit N ... Anzahl der Eingänge) iterieren
 - a) durch Schaltkreise iterieren
 - i. Zustand auf Eingänge abbilden
 - ii. Schaltkreis traversieren und Ausgangsbelegung ermitteln
 - b) Belegungen mit denen des vorangegangenen Schaltkreises Vergleichen
 - c) Bei unterschiedlicher Belegung wird **false** zurückgegeben
 - d) Sonst wird die Iteration über die Zustände fortgesetzt
2. wurde über alle 2^N Zustände iteriert und keine Varianz der Ausgangsbelegung bei den Schaltkreisen festgestellt, wird **true** zurückgegeben

Implementierung

Für alle Circuits werden die möglichen Inputs berechnet und die erhaltenen Outputs miteinander verglichen. Sollte dabei ein Circuit enthalten sein, der abweicht, wird false zurückgegeben (Circuits sind nicht äquivalent).

2.1.3. Äquivalenzprüfung durch SAT-Solver

Voraussetzungen

- Schaltkreise müssen gleichviele Eingänge bzw. Ausgänge besitzen
- falls in einem Schaltkreis ein Eingang bzw. Ausgang vorkommt, muss ein gleichnamiger Eingang bzw. Ausgang auch in den anderen Schaltkreisen vorkommen
- es können beliebig viele Schaltkreise verglichen werden, falls einer nicht äquivalent mit einem anderen ist, wird **false** zurückgegeben
- ist kein Schaltkreis definiert, wird **false** zurückgegeben
- ist nur ein Schaltkreis definiert, wird **true** zurückgegeben

Vorgehen

1. es wird über alle Schaltkreise iteriert
 - a) die KNFs aller Gatter werden ermittelt
 - b) falls der Schaltkreis nicht der erste ist, werden seine Eingänge mit denen des ersten Schaltkreises verbunden d.h. die Identität wird durch eine KNF dargestellt
 - c) falls der Schaltkreis nicht der erste ist, werden seine Ausgänge XOR mit denen des ersten Schaltkreises verbunden d.h. als KNF dargestellt
2. alle Ausgänge der XOR-Verbindungen werden OR verküpft, d.h. es wird eine Klausel gebildet, die alle diese Ausgänge enthält

lässt sich eine dieser Variablen auf **true** abbilden (die These ist SATISFIABLE), dann sind die definierten Schaltkreise nicht äquivalent
3. gesamte KNF wird in Datei geschrieben und der SATsolver gestartet

Implementierung

Der verwendete SAT-Solver ist miniSAT in der Version 1.14 als Binary. Die generierten KNF-Klauseln werden in eine Datei mit dem Namen cnf geschrieben und anschließend vom SAT-Solver gelesen. Die jeweiligen Gatter-Klassen sind so implementiert, dass sie ihre Funktion als KNF darstellen können. Die Klasse **Solver** enthält einen Zähler für die Benennung der KNF-Variablen.

2.2. Versuche

2.2.1. 1. Versuch

BENCHMARKS	tests/01-test/s344.bench	tests/01-test/s349.bench
PARSER	BENCH	BENCH
INPUTS	9	9
DFFs	15	15
OUTPUTS	11	11
GATTER	160	161
ZEIT IN SEKUNDEN	0.105102	0.103989
ÄQUIVALENZCHECK DURCH SAT-SOLVING	äquivalent	
ZEIT IN SEKUNDEN	0.196012	
ÄQUIVALENZCHECK DURCH SIMMULIERUNG	nicht ausgeführt	
ZEIT IN SEKUNDEN	ewig	

Zu lange Ausführungszeiten für den Simulator.

2.2.2. 2. Versuch

BENCHMARKS	tests/02-test/s298.bench	tests/02-test/s298.bench
PARSER	BENCH	BENCH
INPUTS	3	3
DFFs	14	14
OUTPUTS	6	6
GATTER	119	119
ZEIT IN SEKUNDEN	0.079251	0.05012
ÄQUIVALENZCHECK DURCH SAT-SOLVING	äquivalent	
ZEIT IN SEKUNDEN	0.096006	
ÄQUIVALENZCHECK DURCH SIMMULIERUNG	äquivalent	
ZEIT IN SEKUNDEN	2034.31	

2.2.3. 3. Versuch

BENCHMARKS	tests/03-test/s298.bench	tests/03-test/s298a.bench
PARSER	BENCH	BENCH
INPUTS	3	3
DFFs	14	14
OUTPUTS	6	6
GATTER	119	119
ZEIT IN SEKUNDEN	0.048399	0.047988
ÄQUIVALENZCHECK DURCH SAT-SOLVING	nicht äquivalent	
ZEIT IN SEKUNDEN	0.100006	
ÄQUIVALENZCHECK DURCH SIMMULIERUNG	nicht äquivalent	
ZEIT IN SEKUNDEN	0.016245	

2.2.4. 4. Versuch

BENCHMARKS	tests/04-test/c17.bench	tests/04-test/c17a.bench
PARSER	BENCH	BENCH
INPUTS	5	5
DFFs	0	0
OUTPUTS	2	2
GATTER	6	6
ZEIT IN SEKUNDEN	0.002447	0.00174
ÄQUIVALENZCHECK DURCH SAT-SOLVING	nicht äquivalent	
ZEIT IN SEKUNDEN	0.004	
ÄQUIVALENZCHECK DURCH SIMMULIERUNG	nicht äquivalent	
ZEIT IN SEKUNDEN	0.000122	

2.2.5. 5. Versuch

BENCHMARKS	tests/05-test/c17.bench	tests/05-test/c17a.bench
PARSER	BENCH	BENCH
INPUTS	5	5
DFFs	0	0
OUTPUTS	2	2
GATTER	6	7
ZEIT IN SEKUNDEN	0.002533	0.001905
ÄQUIVALENZCHECK DURCH SAT-SOLVING	äquivalent	
ZEIT IN SEKUNDEN	0	
ÄQUIVALENZCHECK DURCH SIMMULIERUNG	äquivalent	
ZEIT IN SEKUNDEN	0.001855	

Veränderung: ein NAND-Gatter durch ein AND und ein NOT ersetzt. Ergebnis korrekt.

2.2.6. 6. Versuch

BENCHMARKS	tests/06-test/c6288.bench	tests/05-test/c6288a.bench
PARSER	BENCH	BENCH
INPUTS	32	32
DFFs	0	0
OUTPUTS	32	32
GATTER	2416	2416
ZEIT IN SEKUNDEN	8.64887	8.4408
ÄQUIVALENZCHECK DURCH SAT-SOLVING	nicht äquivalent	
ZEIT IN SEKUNDEN	17.3411	
ÄQUIVALENZCHECK DURCH SIMMULIERUNG	nicht äquivalent	
ZEIT IN SEKUNDEN	6.16808	

Veränderung: 2 Gatter geändert.

2.3. Anhang

2.3.1. Circuit

Listing 2.1: Circuit.h

```

1  #ifndef circuit_h
2  #define circuit_h
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <cmath>
8  #include <map>
9  #include <vector>
10 #include <string>
11 #include <set>
12
13 class Circuit;
14
15
16 #include "../gates/Gates.h"
17 typedef boost::shared_ptr<Circuit> CircuitPtr;
18
19
20
21 class Circuit{
22
23     private:
24         // unsigned int gateCount;
25         std::string name;
26         std::string description;
27         gateMap gates;
28         inputMap inputs;
29         outputMap outputs;
30
31         unsigned int state;
32         unsigned int depth;
33
34     public:
35
36         Circuit();
37         void simulateAllInputs();
38         void simulateCircuit();
39
40         void printInputs();
41         void printOutputs();
42
43         void resetGates();
44         void resetOutputs();
45
46         void addInput(std::string name, InputPtr input);
47         void addOutput(std::string name, OutputPtr output);
48         void addGate(std::string name, GatePtr gate);
49
50         bool isInput(std::string name);
51         bool isOutput(std::string name);
52         bool isGate(std::string name);
53
54         GatePtr getGate(std::string name) { return this->gates[name]; };
55         InputPtr getInput(std::string name) { return this->inputs[name]; };
56         OutputPtr getOutput(std::string name) { return this->outputs[name]; };
57
58         unsigned int getGatesCount() { return this->gates.size(); };
59         unsigned int getInputsCount() { return this->inputs.size(); };

```

```

60     unsigned int getFFInputsCount();
61     unsigned int getOutputsCount() { return this->outputs.size(); };
62
63     unsigned int getState() { return this->state; };
64     bool setNextState();
65     void resetState();
66
67     std::set<std::string> getInputKeys();
68     std::set<std::string> getOutputKeys();
69     std::set<std::string> getGateKeys();
70
71     void setName(std::string name) { this->name = name; }
72     void calculateDepth();
73
74
75
76 };
77 #endif

```

2.3.2. Simulator

Listing 2.2: Simulator.h

```

1  #ifndef simulator_h
2  #define simulator_h
3
4
5  #include "../parser/Parsers.h"
6  #include "Circuit.h"
7
8  typedef std::map<std::string, CircuitPtr> circuitMap;
9
10 class Simulator{
11
12     protected:
13         circuitMap circuits;
14         unsigned int globalGateCount;
15
16
17     public:
18
19         Simulator() { this->globalGateCount = 1; };
20         void parseCircuit(std::string parser, std::string filePath);
21         void simulateCircuits();
22         virtual bool checkEquivalenceOfCircuits();
23         unsigned int getCircuitCount() { return this->circuits.size(); };
24
25         unsigned int getGlobalGateCount() { return this->globalGateCount; };
26         circuitMap getCircuits() { return this->circuits; };
27         void importCircuits(circuitMap circuits, unsigned int globalGateCount) { this->circuits = circuits;
28             this->globalGateCount = globalGateCount; };
29 };
30 #endif

```

2.3.3. Solver

Listing 2.3: Solver.h

```

1  #ifndef solver_h
2  #define solver_h
3
4  #include "Simulator.h"
5  #include "../parser/Parsers.h"
6  #include "Circuit.h"
7

```

```

8
9 class Solver : public Simulator{
10
11
12     public:
13
14         Solver() : Simulator() { };
15         bool checkEquivalenceOfCircuits();
16         void writeCNF(std::string cnf);
17         std::string getCNF();
18     };
19 #endif

```

2.3.4. Parsers

Listing 2.4: Parsers.h

```

1 #ifndef parsers_h
2 #define parsers_h
3
4 class Parser;
5 #include "Parser.h"
6
7
8 #include "BENCH.h"
9 #include "VERILOG.h"
10
11
12 #endif

```

2.3.5. Parser

Listing 2.5: Parser.h

```

1 #ifndef parser_h
2 #define parser_h
3
4 #include <map>
5 #include <string>
6 #include <boost/regex.hpp>
7 #include "../main/Circuit.h"
8 #include "../gates/Gates.h"
9 class Parser;
10
11 template<typename T> Parser * createParserInstance() { return new T; }
12 typedef std::map<std::string, Parser*(*)()> parserType;
13
14 class Parser
15 {
16
17     protected:
18         std::string parseInput;
19         Circuit *circuit;
20         unsigned int gateCounter;
21
22     public:
23         Parser() { this->gateCounter = 0; };
24
25         Parser* getParser(std::string parser);
26         void parseCircuit(std::string filePath, unsigned int globalGateCount);
27
28         void readFile(const char * path);
29
30         std::string parseComments() { return ""; };
31         void setParseInput(std::string parseInput);

```

```

32     GatePtr getGateFromString(std::string gateName, std::string name);
33
34     virtual void parseInputs();
35     virtual void parseFFs();
36     virtual void parseOutputs();
37     virtual void parseGates() {};
38
39     virtual boost::regex getInputRegex() { return (boost::regex (".")); };
40     virtual boost::regex getFFRegex() { return (boost::regex (".")); };
41     virtual boost::regex getOutputRegex() { return (boost::regex (".")); };
42     virtual boost::regex getGateRegex() { return (boost::regex (".")); };
43     virtual boost::regex getCommentRegex() { return (boost::regex (".")); };
44
45     CircuitPtr getCircuit() { return CircuitPtr(this->circuit); };
46
47     unsigned int getGatesCount() { return this->gateCounter;};
48
49 };
50 #endif

```

2.3.6. BENCH

Listing 2.6: BENCH.h

```

1  #ifndef bench_h
2  #define bench_h
3
4  #include "Parser.h"
5
6  class BENCH : public Parser
7  {
8      public:
9          BENCH() { this->gateCounter = 0; }
10         boost::regex getInputRegex();
11         boost::regex getFFRegex();
12         boost::regex getOutputRegex();
13         boost::regex getGateRegex();
14         boost::regex getCommentRegex();
15
16         void parseGates();
17
18     };
19 #endif

```

2.3.7. Gates

Listing 2.7: Gates.h

```

1  #ifndef gates_h
2  #define gates_h
3  #include <boost/shared_ptr.hpp>
4  #include <boost/make_shared.hpp>
5
6  #include "Gate.h"
7  typedef boost::shared_ptr<Gate> GatePtr;
8  typedef std::map<std::string, GatePtr> gateMap;
9
10 #include "Input.h"
11 typedef boost::shared_ptr<Input> InputPtr;
12 typedef std::map<std::string, InputPtr> inputMap;
13
14
15 #include "Output.h"
16 typedef boost::shared_ptr<Output> OutputPtr;
17 typedef std::map<std::string, OutputPtr> outputMap;

```



```

18
19 #include "AND.h"
20 #include "OR.h"
21 #include "NAND.h"
22 #include "NOR.h"
23 #include "NOT.h"
24 #include "BUFF.h"
25 #include "XOR.h"
26 #include "XNOR.h"
27 #include "DFF.h"
28
29
30 template<typename T> Gate * createGateInstance() { return new T; }
31 typedef std::map<std::string, Gate*(*)()> gateType;
32
33 #endif

```

2.3.8. Gate

Listing 2.8: Gate.h

```

1 #ifndef gate_h
2 #define gate_h
3 class Gate;
4
5 #include <deque>
6 #include <string>
7 #include <sstream>
8
9
10 class Gate
11 {
12
13     protected:
14         std::deque<bool*> inputs;
15         std::vector<unsigned int> inputKeys;
16         unsigned int outputKey;
17
18         bool temp_output;
19
20         std::string name;
21
22     public:
23         bool output;
24
25         Gate() : inputs(2), inputKeys(2) { };
26         Gate(unsigned int in) : inputs(in), inputKeys(in) {};
27         Gate(std::string name) : inputs(2), inputKeys(2) { this->name = name; };
28
29         void setInput(unsigned int input, bool* outputsRef, unsigned int inputKey)
30         {
31             if(this->inputs.size()-1<input){
32                 this->inputs.resize(input + 1);
33                 this->inputKeys.resize(input + 1);
34             }
35             this->inputs[input] = outputsRef;
36             this->inputKeys[input] = inputKey;
37         };
38
39         bool getInput(unsigned int input) { return *this->inputs[input]; };
40
41         void calculateOutput() { this->temp_output = this->gateOutput(); };
42         void setOutput() { this->output = this->temp_output; };
43         bool getOutput() { return this->output; };
44         void resetOutput() { this->output = false; this->temp_output = false;};
45

```

```

46     bool* getOutputRef() { return &this->output; };
47
48     virtual bool gateOutput() { return false; };
49
50     void setOutputKey(unsigned int outputKey) { this->outputKey = outputKey; }
51     unsigned int getOutputKey() { return this->outputKey; }
52
53     virtual std::string getCNF() { return ""; }
54
55     virtual unsigned int getNumberOfCNFClauses() { return 0; }
56
57     virtual std::string getGateType() { return ""; };
58 };
59 #endif

```

2.3.9. Input

Listing 2.9: Input.h

```

1  #ifndef input_h
2  #define input_h
3
4
5  #include "Gate.h"
6
7
8  class Input
9  {
10
11     protected:
12         std::string name;
13         bool *value;
14         bool innerValue;
15         unsigned int outputKey;
16
17     public:
18         Input() { this->innerValue = false; this->value = &this->innerValue; }
19         Input(std::string name) {this->name = name; this->innerValue = false; this->value = &this->innerValue; }
20         void setInput(bool* outputsRef) { this->value = outputsRef; };
21         void setInput(bool input) {this->innerValue=input; this->value = &this->innerValue; };
22         bool* getOutputRef() { return this->value; };
23         bool getOutput() { return *this->value; };
24
25         void setOutputKey(unsigned int outputKey) { this->outputKey = outputKey; }
26         unsigned int getOutputKey() { return this->outputKey; }
27
28         virtual std::string getGateType() { return "Input"; };
29 };
30 #endif

```

2.3.10. DFF

Listing 2.10: DFF.h

```

1  #ifndef dff_h
2  #define dff_h
3
4
5  #include "Gate.h"
6
7
8  class DFF : public Input
9  {
10     public:
11         DFF() : Input() {};

```

```

12     DFF(std::string name) : Input(name) {};
13
14     std::string getGateType() { return "DFF"; };
15 };
16 #endif

```

2.3.11. Output

Listing 2.11: Output.h

```

1  #ifndef output_h
2  #define output_h
3
4
5
6
7  class Output
8  {
9
10     private:
11         std::string name;
12         bool *value;
13         unsigned int outputKey;
14
15     public:
16         Output() {}
17         Output(std::string name) {this->name = name; }
18         void setInput(bool* outputsRef, unsigned int outputKey) { this->value = outputsRef; this->outputKey =
            outputKey; };
19         bool* getOutputRef() { return this->value; };
20         bool getOutput() { return *this->value; };
21
22         void setOutputKey(unsigned int outputKey) { this->outputKey = outputKey; }
23         unsigned int getOutputKey() { return this->outputKey; }
24     };
25 #endif

```