

# Digitale Integrierte Schaltungen

384.086

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



Institut für Computertechnik

Digitale Integrierte Schaltungen

DIS

K1

p. 1

18.01.2013 13:00:57

Die Inhalte der Lehre müssen sich der jeweils gegebenen Situation anpassen. Die Theorie der Schaltnetze und Schaltwerke wird deshalb zunehmend eingeschränkt vorgetragen, um modernen Schaltungsentwurfsverfahren und anderen Themen mehr Raum zu geben. Dieser Prozess ist kontinuierlich. Wer Vorschläge hierzu hat, sollte sie vorbringen.

Aufbauend auf dem Kenntnisstand grundlegender Vorlesungen über digitale Elektronik und Mikroprozessortechnik wird zunächst in Verfahren des systematischen Entwurfs von Schaltnetzen und Schaltwerken eingeführt. Hierfür werden spezielle Themen herausgegriffen, die einerseits zu den Grundkenntnissen auf dem Gebiet der Hardwareentwicklung gezählt werden und andererseits die Technik der Vorgehensweise bei der Entwicklung von Hardwareschaltungen - implizit oder explizit - darlegen.

Innerhalb der Thematik der Schaltnetze wird kurz das QMC-Verfahren angesprochen, das in der Literatur auf breiter Basis behandelt wird. Zu Unrecht! Das Verfahren ist zwar einfach, wird aber in der Praxis garnicht verwendet, da es hohe Rechenzeiten erfordert und hohe Speicherkapazitäten verlangt. Wesentlich effektiver ist das Sharp-Verfahren, das jedoch in der Literatur kaum zu finden ist. Ich gehe kurz darauf ein, da es verdeutlicht, wie über eine geschickte mathematische Herleitung, über die ternäre Logik, die Effizienz von Programmen sowie der Programmieraufwand drastisch reduziert werden kann. Trotzdem, in modernen Entwicklungs-Tools basiert die Optimierung von Schaltnetzen zumeist auf heuristischen Verfahren, deren Ergebnisse im Allgemeinen aber nicht weniger genaue Resultate liefern.

- 1 Einführung
- 2 Technologie und Schaltnetze
- 3 Schaltwerke
  - Automatentheorie der Schaltwerksentwicklung*
  - Stabilitätsbetrachtung*
  - Realisierung von synchronen Schaltwerken über Automatentheorie*
  - Effekte asynchroner Schaltwerke und Hinweise zur Entwicklung*
  - synchroner Schaltwerke*
  - Synthese synchroner Schaltwerke*
  - Berechnung eines Mealy-Automaten und eines Moore-Automaten*
  - Entwurf Hazard-freier Schaltungen*

Der inhaltliche Schwerpunkt dieses "Einführungskapitels" liegt also nicht darin, das Wissen zu erlernen, wie man optimiert - das sollten wir Informatikern überlassen, sondern zu verstehen, dass es außerhalb der Booleschen Algebra weitere mathematische Methoden gibt, mit denen Schaltungen auf elegante Weise entwickelt und optimiert werden können. In den Übungen wird auf das Thema Schaltnetze nicht mehr eingegangen. Im Skript ist das Kapitel dagegen sehr ausführlich behandelt, da es kaum Lehrbücher darüber gibt. In der Vorlesung selbst wird dann genau dargelegt, was prüfungsrelevant ist und was nicht. Aus diesem Grund werden die in der Vorlesung verwendeten Power-Point-Folien elektronisch über das Netz zur Verfügung gestellt.

Das Kapitel Schaltnetze wird abgeschlossen mit einer kurzen Erläuterung über die im Folgenden zugrunde gelegten Technologien, was jedoch mehr Wiederholung aus vorhergehenden Vorlesungen sein soll.

Im Kapitel 3 Schaltwerke werden der wesentliche Unterschied zwischen synchronen und asynchronen Schaltwerken und daran anschließend die Schaltwerksentwicklung über die Modelle der Mealy- und Moore-Automaten erläutert. Ergänzend wird auf ein modifiziertes Automatenmodell eingegangen, womit auf die Vielfalt der Variationsmöglichkeiten der Automatenmodelltechnik hingewiesen werden soll.

Entscheidender Aspekt bei der Hardwareentwicklung ist die Analyse von Hazards und Races, die es im allgemeinen in Schaltungen zu vermeiden gilt. Mit Hilfe der Mealy-Automatentheorie sind sie plausibel darzustellen und zu erläutern und wurden deshalb in das Kapitel Schaltwerke mit aufgenommen.

Abgeschlossen wird das dritte Kapitel Schaltwerke mit Hinweisen, wie die Analyse bzw. Synthese eines Schaltwerkes systematisch aufgebaut werden kann, um schnell und sicher zu einem "vernünftigen" Ergebnis zu gelangen.

Nachdem die Voraussetzungen für die Entwicklung komplexer Schaltwerke geschaffen sind, werden im Kapitel 4 zunächst deren prinzipielle Modularisierung und Aufbau von Mikroprogrammschaltwerken behandelt, bevor mögliche Architekturen besprochen werden.

- 4      **Modularisierung komplexer Schaltwerke**  
*Programmschaltwerke*  
*Mikroprogrammablauf des "Urvaters" 8080A*
- 5      **ASIC Einführung**  
*PLDs, FPGA, Antifuse-FPGA, LCA, ..*  
*Gate Arrays, Standard Cells, Full Custom IC, ..*
- 6      **HDL/VHDL**  
*Anhang*  
*Stichwortverzeichnis*

Kapitel 5 stellt eine Einführung in die ASIC-Entwicklung dar. Ziel ist es, das prinzipielle Verständnis für die automatisierte Schaltungsentwicklung zu schaffen. Erst gegen Ende der Vorlesung über ASIC vorzutragen, erscheint mir aus folgenden Gründen konsequent: Beziehen sich die vorausgehenden Themengebiete allgemein auf die Entwicklung logischer Schaltungen, wird im Kapitel ASIC auf die konkrete Schaltungsentwicklung integrierter Schaltkreise eingegangen. Das Kapitel 5 hebt sich damit deutlich von den vorausgegangenen Kapiteln ab.

Das sechste Kapitel des Skriptums stellt die Entwurfssprache VHDL vor, ein Werkzeug, mit dem heute weltweit digitale ASIC-Designs erfolgen. Es ist eine mächtige Sprache auf hohem abstrakten Level. Das Kapitel ist dementsprechend auch entscheidend für die Prüfung.

Jedem Kapitel des Skriptums wird eine Literaturliste hinzugefügt, die als notwendige Ergänzung anzusehen ist.

Wer Verbesserungen und Ergänzungen vorschlagen möchte, lade ich herzlich dazu ein. Am sinnvollsten ist es dabei, diese an mich elektronisch direkt zu richten:

## Abkürzungen und Sonderzeichen

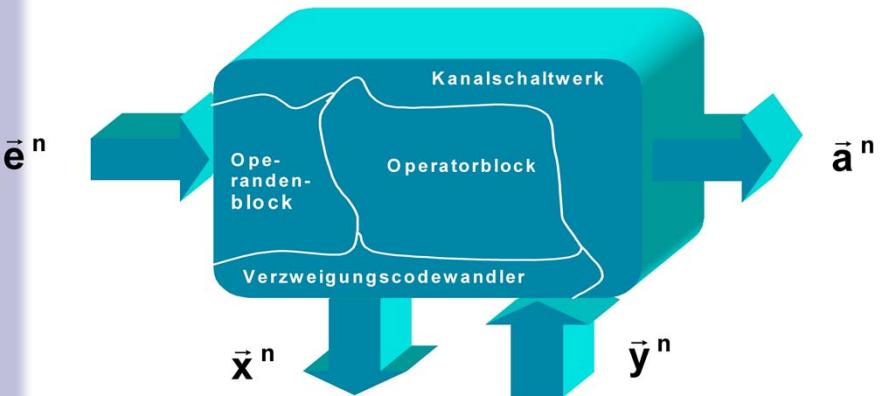
2W	:	Zweiwertlogik (2W: Index)
3W	:	Dreiwertlogik (3W: Index)
AGA	:	Alterable Gate Arrays
ASCII	:	American Standard Code for Information
ASIC	:	Application Specified Integrated Circuit
B	:	binäre Zahlendarstellung
Bin	:	Binärzahlen
BITE	:	Builtin Test Equipment
BIST	:	Builin System Test
BuD	:	Bottomup Design
CISC	:	Complex Instruction Set Computer
Clk	:	Clock
CPLD	:	Complex Programmable Logic Device
CPU	:	Central Processor Unit
CS	:	Chip-select
DEC	:	Decoder (engl.)
DMA	:	Direct Memory Access
DN	:	disjunktive Normalform
EMC	:	Electromagnetic Compatibility
EMV	:	Elektromagnetische Verträglichkeit
EPLD	:	Eraseable Programmable Logic Device
EPROM	:	Erasable Programmable ROM
FF	:	Flipflop
FP	:	Field Programmable Circuits
FPGA	:	Field Programmable Gate Array
FPLA	:	Field Programmable Logic Array
FPLS	:	Field Programmable Logic Sequencer
FPRP	:	Field Programmable ROM Patch
GAL	:	Generic Array Logic
Gl.	:	Gleichung
GND	:	Ground
GTi	:	Gruppentabelle i
HAL	:	Hardware Array Logic
HDL	:	Hardware Description Language
Hex	:	Hexadezimalzahlen
IC	:	Integrated Circuit
IFL	:	Integrated Fuse Logic
I/O	:	Ein-, Ausgabe (In/Out)
KDN	:	kanonische disjunktive Normalform
KI	:	Kernimplikant
KN	:	konjunktive Normalform
KKN	:	kanonische konjunktive Normalform
KV	:	Karnaugh-Veith
LCA	:	Logic Cell Array
ld	:	Logarithmus dualis
log	:	dekadischer Logarithmus
MACH	:	Macro Arrays CMOS High-density
MUX	:	Multiplexer
NB	:	Nebenbedingung
PAL	:	Programmable Array Logic
PC	:	Program Counter
PLA	:	Programmable Logic Array
PLD	:	Programmable Logic Devices
PROM	:	Programmable ROM

Interchange

PT	:	Primterm
PTi	:	Primtermtabelle i
RD	:	Read
RISC	:	Reduced Instruction Set Computer
ROM	:	Read Only Memory
round	:	Anweisung zum Runden einer Zahl
RxD	:	Receive Data
SMD	:	Surface Mounted Devices
TdD	:	Top down Design
TTL	:	Transistor-Transistor-Logic
TxD	:	Transmit Data
$V_+$ , $V_-$	:	positive und negative Versorgungsspannung
VHDL	:	VHSIC Hardware Description Language
VHSIC	:	Very high Speed Integrated Circuit
WR	:	Write

$\not\equiv$	:	Antivalenz
#	:	Sharp-Operator
$\uparrow$	:	Wechsel von der binären in die ternäre Logik
$\downarrow$	:	Wechsel von der ternären in die binäre Logik
$\underline{a}$	:	Vektor a (Ein Vektor wird aus Gründen der Übersicht nicht immer als Vektor gekennzeichnet, nur dort, wo es sinnvoll erscheint. Wir erlauben uns, auch die Schreibweise zu verwenden, bei der der Buchstabe mit einem Pfeil darüber geschrieben wird, die jedoch in Power Point hier auf den Notizseiten nicht verwendbar ist. Die Pfeilschreibweise ist die gebräuchliche, hat aber den Nachteil, dass eine Gleichung schnell unübersichtlich wird, wenn in der Gleichung zusätzlich negierte Größen auftauchen (und wenn das verwendete Textsystem, die Pfeile nicht deutlich genug gegenüber der Negierung hervorhebt).)

Alle weiteren Sonderzeichen werden auf den Folien speziell deklariert, da sie ebenfalls hier nicht darstellbar sind. Wenn es hierzu Fragen, Zweifel oder offene Punkte gibt, wenden Sie sich bitte an uns.



## Kapitel 1: Einführung

Die immer höhere Komplexität von Schaltungen lässt es notwendig werden, von der traditionellen "flachen" Schaltwerksentwicklung abzugehen und sich Methoden anzueignen, die zu einem "schnellen" Ergebnis führen und das System wartbar werden lassen. Es reicht nicht mehr aus, Funktionsblöcke im Rahmen einer Applikationsvorgabe zu definieren, die im einzelnen entwickelt werden und anschließend zusammenzusetzen sind. Zurecht muss man sonst fragen: Sind all die Funktionen im vollen Umfang integriert, die gefordert sind? Habe ich tatsächlich eine optimale Schaltung erhalten? Ist sie für jedermann überschaubar? Ist sie gut zu verifizieren? usw.

In der Praxis erlebt man regelmäßig die gleiche Prozedur: Ein oder mehrere Entwickler vertiefen sich in einen Problembereich, erzielen eine gut funktionierende Schaltung, dokumentieren diese mehr oder weniger ausführlich und glauben, damit etwas geschaffen zu haben, das Bestand hat. Dann übernimmt der nächste Entwickler oder die nächste Gruppe das System und stellt fest, manches hätte man besser machen können, erkennt Fehler und benötigt enorme Zeit, um die Dokumentation überhaupt zu verstehen und im Detail Änderungen vorzunehmen.

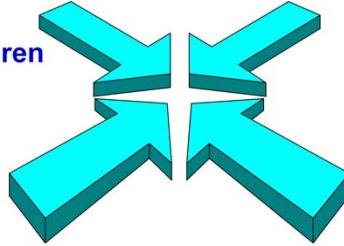
Woran liegt dies im allgemeinen?

Ein zentrales psychologisches Problem ist, das sich prinzipiell nicht vermeiden, sondern nur reduzieren lässt: Arbeitet man sich lange genug in ein Gebiet ein, erhält man ein Gefühl für das System und dokumentiert entsprechend knapp. Das bedeutet aber, man setzt bestimmte Dinge voraus, die man sich eigentlich auch erst erarbeiten musste, bis der "Groschen fiel", nun aber als selbstverständlich voraussetzt. Der "Außenstehende" hat dann kaum noch die Möglichkeit, die Zusammenhänge schnell zu erfassen, um möglichst rasch Details einordnen zu können. Man versteht beispielsweise nicht, warum der andere oft so "begriffsstutzig" ist. Da sich zudem jeder Entwickler rasch in "sein Kind" verliebt, werden unbewusst Scheuklappen gegenüber anderen Konzepten aufgebaut, die nur schwer wieder abzulegen sind.

## Einführung: Begriffe, Definitionen, Übersicht, ...

### Vorgehensweise

- ❖ Problemstellung
- ❖ Synthese
- ❖ Schaltungsentwurfsverfahren
- ❖ Optimierung
- ❖ Entwurfsgrundsätze
- ❖ Designwerkzeuge



Dem Entwickler stellen sich somit immer gleich mehrere Probleme:

- Er muss darauf achten, dass das System "durchsichtig" bleibt.
- Er muss technisch versiert sein, um zu einer "optimalen" Schaltung zu kommen.
- Er muss versuchen, stets offen für andere Lösungen zu sein.
- Er muss auf Test- und Wartbarkeit achten.

All dies verlangt ein hohes Maß an Selbstbeherrschung, Aufmerksamkeit und Disziplin, wozu im Widerspruch steht, dass der Entwickler seiner Phantasie freien Lauf lassen soll, um kreative Ergebnisse zu erzielen. Die Frage ist, wie kann das Problem zumindest weitgehend prinzipiell gelöst werden?

Dass die Thematik nicht trivial und selbstverständlich ist, erkennt man an der Entwicklungsgeschichte der RISC-Prozessoren. Der Neumann-Prozessor wurde konsequent weiterentwickelt, immer mehr Leistungsmerkmale wurden integriert, bis man erkannte, dass in vielen Anwendungsfällen der Programmierer gar nicht mehr bereit ist, die komplexen Befehle so auszunutzen, wie die Entwickler der Mikroprozessoren sich das erhofft hatten. Verwendet werden zumeist die einfachen Befehle, mit denen man, ohne in Büchern nachzuschlagen, umzugehen versteht. Das Ergebnis einer entsprechenden Untersuchung war ein Umdenkprozess, der zum RISC-Prozessor führte (Kapitel 4 des Skriptes). Man entwickelte Prozessoren, die nur wenige Befehle kannten, optimierte dafür nun den Prozessor auf Geschwindigkeit und stellte geeignete Compiler zur Verfügung. Die Erfolge sind bekannt, der RISC-Prozessor hatte einen durchschlagenden Erfolg (allerdings sind heute immer noch in etwa ca. 80 % aller Prozessor CISC-Prozessoren).

Er ist damit nicht das Ergebnis einer konsequenten Weiterentwicklung der konventionellen Prozessorarchitektur, sondern hat seinen Ursprung **in einem neuen Denkansatz**, der auf einem **hohen abstrakten Level** aufsetzt. Man versuchte nicht Schaltungsdetails zu verbessern, sondern fing "von vorne" an und stellte die Fragen: **Was ist eigentlich gewünscht? Was verwendet der**

Was für eine Flasche? Was für ein Programmierer?

# Problem

## Arbeitet man lange genug an einer Thematik :

- Gefühl: alles ganz einfach
- Dokumentation entsprechend kompliziert für andere, da:
  - man setzt viele Dinge voraus
  - beschreibt nur das "Notwendige"
- Folge:
  - "Außenstehende" finden kaum noch Zugang
  - ..
- Da sich jeder Entwickler rasch in sein "Kind" verliebt, werden bewusst Scheuklappen gegenüber anderen Konzepten aufgebaut.
- ..



### Konsequenz:

- **Dokumentation (Spezifikation) lernen + trainieren!**

Ziel für die Hardware-Entwicklung muss es somit sein, stets auf der *höchstmöglichen Abstraktionsebene* zu arbeiten, so dass eine Beschreibung leicht fällt und Variationen einfach werden. Notwendige Voraussetzung ist hierfür

- die Top-down-Methode  
(*welche Funktionen sind wirklich gewünscht*)
  - eine klare Modularisierung  
(*wie fein kann es in einfache Einheiten granularisiert werden*)
- sowie eine eindeutige Schnittstellenbeschreibung  
(*inwieweit ist eine Intrakommunikation zwischen den granularisierten Einheiten einfach zu halten und einfach zu beschreiben*).

# Gegenläufige Probleme

Der Entwickler sieht sich somit gleich mehreren Probleme gegenüber:

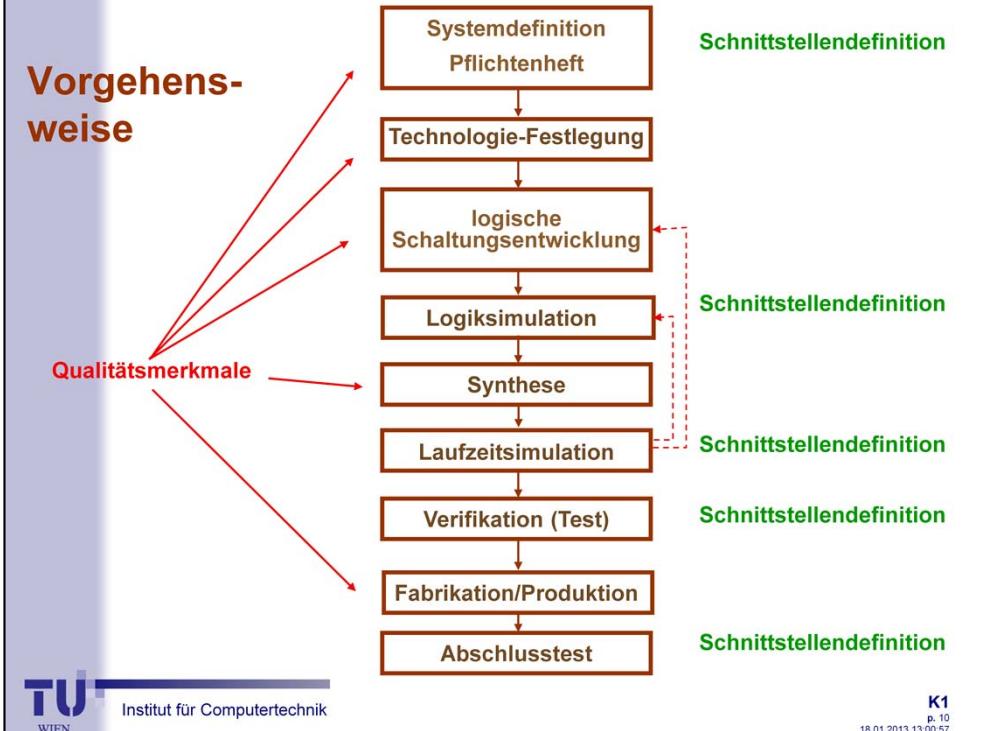
- ❖ Er muss technisch versiert sein, um "optimale" Schaltungen zu schaffen.
- Er muss neu und unverbraucht sein, um keine Scheuklappen zu haben, um nicht vorprogrammiert zu sein.
- ❖ Er muss von seinen Entwürfen überzeugt sein.
- Er muss für andere Lösungen jederzeit offen sein.
- ❖ Er muss darauf achten, dass das System durchsichtig bleibt.
- Er muss es möglichst wenig modularisieren, um es nicht zu groß werden zu lassen.
- ❖ Er muss auf die Test- und Wartbarkeit achten.
- Er darf nicht zuviel Redundanz integrieren.
- ❖ ...



Genau diese Überlegungen sollten als Schwerpunkt der Vorlesung angesehen werden. Die einzelnen Werkzeuge, bestimmte Schaltungsdetails usw. sind einem raschen Wechsel unterworfen und können schon morgen völlig überholt sein. Wie man jedoch an eine Thematik bzw. an eine Aufgabe herangeht, lässt sich auf neue Technologien, neue Systeme übertragen.

An dieser Stelle ist der Begriff **HDL (Hardware Description Language)** zu nennen. Durch die zunehmend hohe Integrationsdichte von Bausteinen wird der Entwickler immer mehr gezwungen, seine Schaltungen mit dem Rechner zu spezifizieren, zu designen, zu simulieren und zu testen. Hierfür verwendet er Tools, die mehr oder weniger die Vorgehensweise vorschreiben. Um diese Tools kompatibel gestalten zu können, werden Hardware-Beschreibungssprachen definiert und standardisiert, was heißt, der Entwickler muss sich mit dieser Thematik intensiv auseinandersetzen, wenn er konkurrenzfähige Produkte erzielen möchte.

Der Entwicklungsablauf integrierter Bausteine im Detail muss allerdings an hauseigene Randbedingungen angepasst erfolgen, und gleichzeitig so weit wie möglich offen sein, um flexibel auf Innovationen reagieren zu können. Das Ergebnis kann nur eine Kompromisslösung darstellen. Welcher Ablauf gewählt wird, hängt von vielen Faktoren ab wie der Anzahl der zu entwickelnden und dann zu produzierenden integrierten Schaltkreise, der einzusetzenden Technologie, der Art der Zusammenarbeit mit dem IC-Hersteller, eigenem in der Firma zur Verfügung stehendem Know-how, Personal- und Rechnerkapazität, Anteil der Eigenleistung usw.



Synthese = Compilierung etc.

Das Bild zeigt die prinzipielle Vorgehensweise, die möglichst auch eingeschlagen werden sollte, wenn nur wenig Rechnerunterstützung zur Verfügung steht. Begonnen wird stets mit der Systemdefinition, die ausführlich sein soll, weitgehend alle Details enthalten und klar und eindeutig in schriftlicher Form niedergelegt sein muss. Dies kann nicht deutlich genug hervorgehoben werden.

1. Durch eine klare schriftliche Formulierung der Aufgabenstellung können Missverständnisse zwischen Auftraggeber und Auftragnehmer weitgehend vermieden werden.
2. Durch die schriftliche Formulierung ist der Entwickler gezwungen, seine Gedanken und Ideen zu ordnen und logisch zu verknüpfen.
3. Die schriftliche Formulierung zwingt bei exakter Vorgehensweise zur klaren Schnittstellendefinition, was die weitere logische Schaltungsentwicklung entscheidend vereinfacht.

Die Systemdefinition sollte in der Praxis deshalb einen großen zeitlichen Raum einnehmen und akribisch durchgeführt werden. Je präziser und ausführlicher man bei dieser Arbeit vorgeht, umso mehr Zeit spart man später.

Hat man die Festlegung der Technologie getroffen, kann man dazu übergehen, die logische Schaltung zu entwickeln. Hierfür stehen vor allem drei Verfahren zur Verfügung:

- die flache Schaltungsentwicklung,
- die mathematische Formulierung logischer Gleichungen und
- die Beschreibung über Zustandsgrafen bzw. Tabellen.

# Schaltungsentwurfsverfahren



## (a) "flache" Schaltungsentwicklung

*schon bei geringer Gatterfunktionsanzahl abzulehnen*

hierarchische Methoden:

## (b) Top down Design

*Funktionsblöcke als "black boxes" definieren*

## (c) Bottom up Design

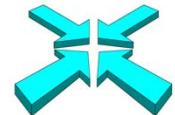
*(von Wurzeln ausgehend), gefährlich, da oft "Überblick vernebelt"*

**für Neuentwicklungen Praxis oft: (b) {+(c)}**

Unter der flachen Schaltungsentwicklung soll verstanden werden, Schaltungen über Symbole und Verbindungsleitungen auf dem Papier oder Bildschirm zu entwerfen. Nicht zu vermeiden ist diese Methode, wenn man sich in der Modellierungsphase befindet, das heißt, Funktionsmodule definiert, die zu einer Einheit zusammengefasst werden. Vermeiden sollte man sie, wenn es sich um schaltungstechnische Details handelt. Dann sollte man die mathematische Formulierung logischer Gleichungen oder die Beschreibung über Zustandsgrafen wählen. Auch die Tabellenform ist nicht günstig, da sie im allgemeinen unübersichtlich ist und damit eine häufige Fehlerquelle darstellt.

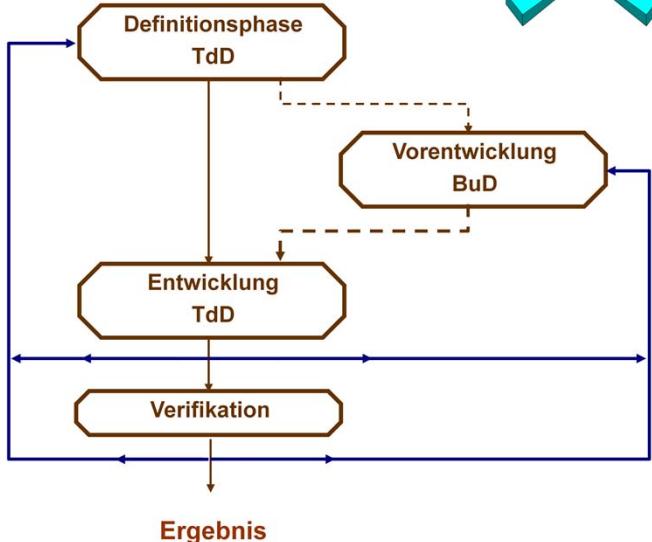
An die logische Schaltungsentwicklung schließt sich die Logiksimulation an sowie die Synthese. Die Synthese für ASICs stellt im allgemeinen kein prinzipielles Problem für den Entwickler dar (wenn nicht harte Laufzeitbedingungen berücksichtigt werden müssen, denn er führt anschließend eine Laufzeitsimulation durch, die ihn in den meisten Fällen auf kritische Pfade hinweist oder sogar Fehler direkt angibt). Bei der Entwicklung von Platinen lassen die Software-Werkzeuge dem Entwickler in der Regel größere Freiheiten, was die Länge und den Querschnitt von Verbindungen, die Verbindungswiege, die Kupferflächenbildung usw. angeht. Verfügt der Entwickler dann nicht über ausreichende Erfahrung, sind oft mehrere Re-Designs notwendig, bis man eine einwandfrei funktionierende Schaltung erhält.

Nach der Laufzeitsimulation ist im allgemeinen die weitere Vorgehensweise weitgehend vorgegeben, auf die der Entwickler weniger Einfluss hat. Doch Schnittstellen müssen weiterhin definiert werden, was in der Praxis auch bezüglich der verwendeten Tools ein zentrales Problem darstellt. Es müssen ja nicht nur die Schnittstellen zwischen den einzelnen Einheiten des Systems definiert, sondern auch die Schnittstellen festgelegt werden, die zwischen den unterschiedlichen Tools Anwendung finden, je nachdem in welchem Entwicklungslevel man sich befindet. Hier ist auch heute eines der großen Problemfelder zu sehen. Tools werden laufend überarbeitet und erhalten so unterschiedliche Versionsnummern. Die Tool-Hersteller stimmen aber selten ihre Revisionszeiten aufeinander ab, was das Handling leicht kompliziert werden lassen kann, wenn man beispielsweise eine neue Version einkauft und diese sich mit einer älteren Generation nicht so recht verträgt, die man aber gerade eingesetzt hat. Verständlich, dass sich deshalb Entwicklungsleiter oft sträuben, mit den Versionsnummern immer auf dem laufenden Stand bleiben zu wollen.



## Top-down-Verfahren

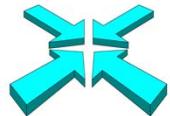
In der  
Praxis  
häufig:



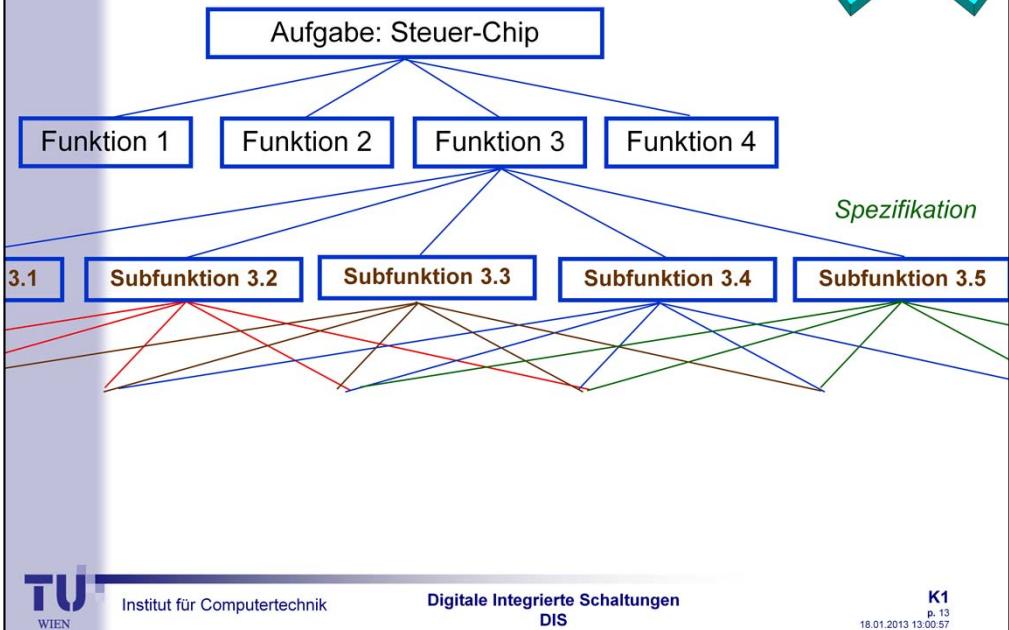
Bezüglich der Qualitätsmerkmale hat sich in der Praxis ein wesentlicher Umdenkmungsprozess vollzogen. Glaubte man früher, dass es ausreichend ist, auf Qualitätsmerkmale erst gegen Ende einer Entwicklung achten zu müssen, steht heute fest, dass diese von Anfang an in die Überlegungen miteinbezogen werden müssen. Auf **Folie 10** weisen sie auf die Level, bei denen sie besonders greifen sollten.

Bezüglich der Schaltungsentwurfsverfahren ist noch hinzuzufügen: Zu unterscheiden sind die "flache Schaltungsentwicklung", das Top down Design (TdD) und das Bottom up Design (BuD). Wie schon erwähnt, ist die "flache Schaltungsentwicklung" weitgehend zu vermeiden, was vor allem auf komplexe Schaltungen und Schaltungen hoher Taktrate zutrifft. Die Wahl zwischen Top down und Bottom up Design ist leicht zu treffen. Entwickelt man Systeme, sollte man zunächst Funktionseinheiten definieren, die man aus der Aufgabenbeschreibung direkt herauslesen kann, das bedeutet, man geht von einem hohen abstrakten Beschreibungs-Level aus und arbeitet sich langsam ins Detail der Einheiten hinein. Das hat den Vorteil, dass man sich zunächst um Detailprobleme nicht zu kümmern braucht, man behält leicht den Überblick, kann relativ spät entscheiden, was in Hardware, was in Software entwickelt wird, und kann Module später leicht austauschen, wenn Optimalere gefunden werden. Und genau das ist die Vorgehensweise beim Top down Design. Das Bottom up Design hat nur dort einen Sinn, wo Machbarkeitsprüfungen im Detail notwendig werden. Eine praxisgerechte Vorgehensweise ist deshalb: Man beginnt mit dem Top-down-Design und wendet es so lange an, bis man auf Details stößt, für die man ad hoc keine Lösung parat hat (Bild oben). Es werden dann evtl. Einzeluntersuchungen notwendig, deren Ergebnisse in die Linie des Top-down-Designs einfließen können.

Zwei Aspekte sind noch nicht angesprochen worden, die zumindest erwähnt werden müssen, bevor auf verschiedene Verfahren der Schaltungsentwicklung im einzelnen eingegangen wird: Optimierungsmöglichkeiten und Designwerkzeug.



## Top-down-Verfahren



Noch eine kurze Bemerkung zum Top-Down-Design. In verschiedenen Bereichen wird dieser Begriff nicht immer gleich verwendet. Aus diesem Grund hier die Definition, wie man ihn im Schaltungsdesign verwendet. Man geht von einer globalen Funktion aus und definiert dann darunterliegende Funktionen, dann wieder darunterliegende Funktionen usw. Zu unterscheiden sind dabei zwei völlig unterschiedliche Hierarchietypen. Zum einen kann man von einer Hierarchie ausgehen, bei der die obere Ebene eine eigenständige Funktion ist, die Ebene darunter die Aufgabe hat, der oberen zu "dienen", also beispielsweise bestimmte Werte zu liefern (siehe beispielsweise das ISO/OSI-Modell, was in FeSy im Folgejahr gelehrt wird).

Es gibt aber auch den Hierarchietyp, bei dem die untere Ebene nur eine detailliertere Darstellung der oberen darstellt, man also von oben nach unten immer weniger abstrahiert (siehe beispielsweise die Hierarchie des SDL, was ebenfalls in FeSy dargelegt wird).

## Optimierung hinsichtlich

- | ➤ Preis  | <u>Konsequenz:</u> |                     |
|--|--------------------|---------------------|
| HW   | ↔                  | SW ?                |
| Entwicklungs-,                                       | ↔                  | Produktionskosten ? |
| Einkaufen  | ↔                  | Lizenzgebühren ?    |
| Systemleistung                                       | ↔                  | Preis ?             |
| ...  |                    |                     |
| ➤ <u>Leistungsanforderungen bzgl.:</u>               |                    |                     |
| Einsatzgebiet (Flugzeugbau, weiße Ware, ..)          |                    |                     |
| Verfügbarkeit ( → SW) ↔ Geschwindigkeit              |                    |                     |
| ➤ <u>Wartung:</u>                                    |                    |                     |
| zukünftig Verbesserungen / Erweiterungen vorgesehen? |                    |                     |



Auf verschiedene Optimierungsmöglichkeiten und -verfahren wird im Skript noch ausführlich eingegangen. Doch sollte man sich von Anfang an darüber im klaren sein, dass eine Optimierung in vielerlei Hinsicht möglich ist. **Die** optimale Schaltung gibt es nicht. Man muss gleich zu Beginn einer Entwicklung (in der Systemdefinitionsphase) festlegen, wo die Schwerpunkte der Optimierung liegen sollen und wie die verschiedenen möglichen Optimierung hierarchisch einzustufen sind. Ist die zu entwickelnde Einheit eine kleine Mikrorechnereinheit, die bestimmten Sicherheitsanforderungen unterworfen werden soll, ist vielleicht möglichst viel in Software und möglichst wenig in Hardware zu entwickeln, was auch sinnvoll sein kann, wenn hohe Stückzahlen dahinter stehen. Kommt es aber auf Reaktionsgeschwindigkeiten an, kann eine Ausführung von Vorteil sein, in der möglichst viele Funktionen in Hardware realisiert werden. Diese Überlegungen müssen in allen Einzelheiten von Anfang an diskutiert werden, wobei Kriterien wie Entwicklungskosten, Produktionskosten, Stückzahl, Verfügbarkeit, Zuverlässigkeit, Reaktionsgeschwindigkeit, Lizenzgebühren, Größe, Leistungsverbrauch usw. eine Rolle spielen.



# Designwerkzeuge

## Geschlossene Systeme

Häufig als Workstations (Units optimal angepasst/aufeinander abgestimmt, jedoch keine Leistungsauswahl für Entwickler möglich)

## Offene Systeme

(z. B. bieten VLSI-Schnittstellen an, um Bibliotheken von Fremdfirmen integrieren zu können)

### "Einzelbausteine"

- QMC
- Sharp
- Mealy-/Moore-Automat
- ..

Der Aspekt der Designwerkzeuge stellt sich im Einzelfall einfacher dar, da der Entwickler im allgemeinen eine bestimmte Entwicklungsumgebung vorfindet beziehungsweise nur eingeschränkte Möglichkeiten hat, schon aus Kostengründen. Prinzipiell ist bezüglich der Rechnerunterstützung zu unterscheiden zwischen geschlossenen und offenen Systemen. Geschlossene Systeme sind heute häufig Workstations, in denen die einzelnen Units optimal aufeinander abgestimmt sind. Leider ist man dabei auf die Einheiten angewiesen, die vorgegeben sind. Der Entwickler hat aber den eindeutigen Vorteil, dass im Fehlerfall oder bei aufkommenden Fragen stets **ein** Ansprechpartner in Frage kommt, nämlich die Firma, von der das System erworben wurde.

Treten in offenen Systemen, also in Systemen, deren Einheiten von verschiedenen Firmen bezogen wurden, Fehler auf, beginnt oft eine gegenseitige Schuldzuweisung der Firmen, wessen Einheit letztendlich den Fehler verursacht: der entscheidende Nachteil offener Systeme. Von Vorteil ist natürlich, dass sich der Entwickler die für ihn günstigsten Einheiten zusammenstellen kann, was gute Kenntnisse im Detail voraussetzt. Auf nähere Einzelheiten wird im Kapitel 5 (ASIC) eingegangen.

# Entwurfsgrundsätze

## Darstellungsmethoden



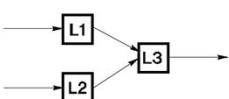
**Didaktisch vorzuziehen:** 1. Strukturelle Darstellung

2. Funktionelle Darstellung

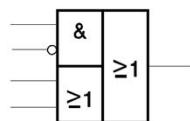
**ABER: Topdown Design für ASICs fordert:**

1. Funktionelle Darstellung

2. Strukturelle Darstellung



*strukturelle  
Darstellung*



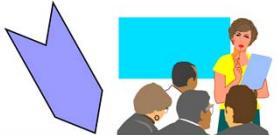
*funktionale  
Darstellung*

*In den Funktionen liegt das Knowhow, was die Firmen ja ungern herausrücken. Sie ziehen also in Datenbüchern Strukturbilder vor, die viel unproblematischer sind.*

Auf noch einen Punkt ist in der Einleitung hinzuweisen: Bei der Beschreibung eines Systems muss zwischen der strukturellen und der funktionalen Darstellung streng unterschieden werden. In einer strukturellen Darstellung werden im allgemeinen nur Knoten (Subeinheiten, Nodes) und der zugehörige Informationsfluss durch Kanten (Edges) wiedergegeben. Komplexe Systeme sind somit relativ einfach beschreibbar. Die funktionale Darstellung beinhaltet dagegen wesentlich mehr Informationen. Bild oben zeigt ein einfaches Beispiel. Aus der strukturellen Darstellung kann nur herausgelesen werden, dass das System aus drei Subeinheiten mit vier Eingängen und einem Ausgang besteht, sowie die Informationsflussrichtung. Aus der funktionalen Darstellung kann dagegen mehr, im vorliegenden Beispiel der boolesche Zusammenhang, ermittelt werden.

Bild oben ist allerdings in der Tat trivial. Doch geht man beispielsweise zur Beschreibung von Mikroprozessoren über, so wird sofort deutlich, dass die strukturelle Beschreibung schon rein aus Komplexitätsgründen einen Sinn hat. In den Datenbüchern von Mikroprozessoren wird diese Technik laufend angewendet und sie ist dem Leser bekannt. Die funktionale Beschreibung erfolgt im allgemeinen nach der strukturellen, um den Leser nicht gleich eingangs mit Details zu überschütten. Diese Vorgehensweise in Datenbüchern ist somit gerechtfertigt. Doch wie steht es beim Entwurf eines ASICs?

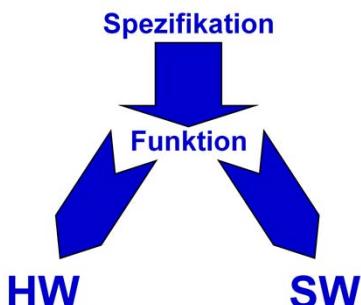
# Darstellungsmethoden - Entwurfsgrundsätze



Funktionelle Darstellung ➔ Strukturelle Darstellung

Denn:

nicht Strukturen sind gefordert,  
sondern Funktionen



Fest steht, dass beim Entwurf eines ASIC's das Topdown-Design angewendet werden soll. Weiterhin werden für den Entwurf zunächst bestimmt keine Strukturen, sondern bestimmte Funktionen gefordert, die Funktionen nämlich, die der Baustein zu erfüllen hat. Das hat klare Konsequenzen. Zunächst muss eine funktionale Beschreibung der gewünschten Einheit erfolgen, aus der dann die strukturelle, eventuell in Hardware und Software getrennt, abgeleitet werden kann (Bild oben). Die Vorgehensweise ist also eindeutig genau umgekehrt wie die oben beschriebene. Oben geht es um die mehr oder weniger sinnvolle didaktische Aufbereitung einer Erläuterung eines Systems oder Bausteins, beim Entwurf muss von den Voraussetzungen ausgegangen werden, und das sind beim ASIC-Entwurf die Funktionen.



# Dokumentation

SW:

- 1. Funktionen
- 2. Strukturen

HW:

- 1. Funktionen
- 2. Strukturen



Gliederung:

- 1. Einführung
- 2. Funktionen
- 3. HW
- 4. SW
- 5. ...

Dazu möchte ich noch eine Bemerkung machen. Findet man in einer Ausarbeitung, beispielsweise in einer Diplomarbeit oder einem Pflichtenheft, eine Gliederung der Kapitel wie folgt vor:

- 1        *Einführung*
- 2        *Hardware*
- 3        *Software*
- 4        *.. ,*

wird daraus eine falsche Vorgehensweise deutlich sichtbar. Bei den beiden angeführten Beispielen einer Diplomarbeit und einem Pflichtenheft kommt es darauf an, den Leser zu überzeugen, warum man zu dieser oder diesen Lösungen beziehungsweise "Strukturen" gekommen ist und nicht zu irgendwelchen anderen Lösungen. Man soll beim Entwurf digitaler Bausteine und der Entwicklung von Software, und davon reden wir hier, nachweisen, dass man nicht "gebastelt", sondern die Topdown-Methode konsequent angewendet hat. Dann gilt aber genau die eben erläuterte Vorgehensweise nach Bild 2. Das bedeutet, dass eine Gliederung der Kapitel dementsprechend wie folgt auszusehen hat:

- 1        *Einführung*
- 2        *Funktionale Beschreibung*
- 3        *Hardware*
- 4        *Software*
- 5        *.. ,*

was im allgemeinen auch für viele Referate und ebenso Dissertationen gilt (es sind allerdings

digitized and evaluated using Kappa statistics.



# Entwurfsgrundsätze

(gelten als Voraussetzung)

1. **Zeitkritische Pfade:** höheres Fan out => Logik duplizieren (parallelisieren), wobei der zeitkritische Pfad von einer Einheit allein und die restlichen Pfade von einer anderen Einheit getrieben werden
2. **Gatter mit vielen Eingängen:** zeitkritische u. zeitunkritische Pfade getrennt zusammenführen (= Gatter in mehreren Stufen aufbauen; bspw. spät kommende Signale am Gatter-Ende zuführen)
3. **Bibliothek:** Intensiv Bibliotheksfunktionen verwenden, da diese i.a. schon zeitoptimiert sind
4. **Möglichst neg. Schaltungen verwenden:** um zusätzliche Inverterstufen einsparen zu können (gilt natürlich bei CMOS nicht und nicht bei allen bipolaren Ausführungen)

Die oben angeführten Entwurfsgrundsätze sind allgemein für digitales Design anzusetzen. Man muss jedoch diese grundsätzlich bezüglich der jeweils verwendeten Technologie durchleuchten. Die oben dargestellten sind zusammengestellt, um einen ersten Eindruck zu gewinnen und ein Gefühl für die Thematik zu gewinnen. Sie werden selbstverständlich in der Vorlesung mehrfach angesprochen.

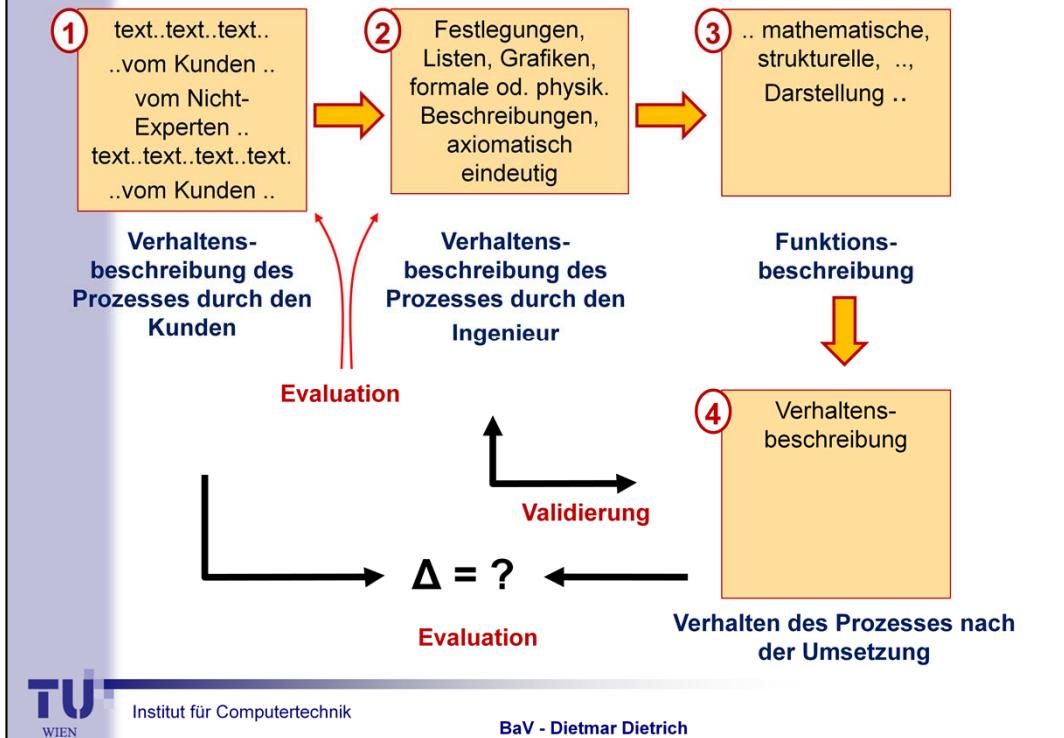


# Entwurfsgrundsätze

(gelten als Voraussetzung)

5. **Asynchrone Schaltwerke vermeiden:** Sie bilden (beim unachtsamen Entwurf) oft die Ursache von **Hazards** (solche Impulse stets über FFs mit hochfrequentem Takt erzeugen: synchr. Schaltw.)
6. **Unterschiedliche Laufzeiten in parallelen Pfaden vermeiden:** bspw. zusätzliche Gatterstufen einführen oder Synchr. über FFs usw.
7. **Takte nicht "vergattern" und dann Einheiten zuführen,** sondern zuerst wieder über FFs nachtakten
8. **Bausteine von Anfang an testbar gestalten:** im Nachhinein oft ohne massive Änderungen nicht mehr möglich
9. **Builtin Test Systems integrieren:** führt zu reduzierten Testvektoren
10. **Interne Busse vermeiden:** möglichst mit Multiplexer arbeiten
11. **Interne Busse nie "frei schwebend" lassen:** Querstrom in den Eingangsstufen muss vermieden werden

## Computerarbeiten (Chip-Design, Simulation, ..)



Die techn. Wissenschaft geht von Modellen aus, also von Annahmen, mit einem Ursache-Wirkungs-Zusammenhang. Oben: 1 nennt man Lastenheft, 2 die Spezifikation.

- (1) Dieser Teil ist zwar evaluierbar (prüfbar), aber nicht beweisbar, also nicht FORMAL verifizierbar. Es ist der Wunsch und die Beschreibung des Kunden.
  - (2) Dieser Teil (2) wird mit Hilfe des Kunden vom Auftragnehmer aus (1) entwickelt und gilt als weitere Rechtsgrundlage z. B. bei der Abnahme. Ideal wäre, wenn er verifiziert werden kann, bevor das Design (3) begonnen wird. Verifizierbarkeit ist die Belegbarkeit. Für (2) kann es unter bestimmten Fällen angewendet werden, wenn es ausreichend formal beschrieben werden kann. In der Praxis verlässt man sich aber allzu häufig auf das Expertenwissen des Auftragsnehmers und verifiziert es nicht formal – schon aus Kostengründen, sondern validiert es nur gegenüber (1).
  - (3) Teil 3 ist das endgültige Design, also die endgültige mathematische und sonstige funktionale, formale Beschreibung des Systems (z. B. in Form von Subfunktionen, logische und Mealy-Automaten-Funktionen, strukturelle Beschreibung, Gleichungen usw.)
  - (4) Der Teil (4) beschreibt das Verhalten der realisierten Schaltung, was natürlich von der Beschreibung gegenüber (2) und vor allem gegenüber (1) abweichen kann. (4) ist wiederum unter bestimmten Umständen gegenüber (2) verifizierbar.

#### (A) Evaluation: Beschreibung, Analyse und Bewertung eines Prozesses/Projektes/...

(B) Validität ist dann die Gültigkeit/Belastbarkeit/ der Annahmen. Unter der Validierung versteht man bei der Chip-Entwicklung den Vergleich zwischen der Spezifikation (2) und seinem Verhalten (4). So versteht man auch darunter die Beweisführung, dass ein System die Anforderungen in der Praxis erfüllt, z. B. Prüfung aller Zeitdurchläufe in einem Chip, ob der Chip den Realtime-Anforderungen entspricht usw.

### Weitere Definitionen:

- (a) Verifikation: Verifizierbarkeit ist die Belegbarkeit einer Behauptung (siehe (2))
  - (b) Formale Verifikation: mit Hilfe von Regeln formal überprüfen
  - (c) Falsifizierbarkeit ist die Widerlegbarkeit.

Die Beschreibung von 3 ist im Allgemeinen wesentlich (!) einfacher als 1, 2 oder 4 (siehe Literatur: Braitenbergsche Vehicles)

# Digitale Integrierte Schaltungen

384.086

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



# Im Wesentlichen Wiederholung!

## Kapitel 2 Schaltnetze

- Vereinfachung nach QMC
- Vereinfachung nach Sharp
- Multilevel Design



Zwei Minimierungsverfahren für Schaltnetze werden im Prinzip kurz angeführt. Zuvor soll kurz die Nomenklatur, die hier verwendet wird, vorgestellt werden.

Zwischen der Booleschen Algebra und der Schaltalgebra ist klar zu unterscheiden. Die Boolesche Algebra, entwickelt von George Boole (*Investigation of the Laws of Thought*, 1854) beschäftigt sich in ihrem Kern mit der Formalisierung von Verknüpfungen von Aussagen, während die Schaltalgebra von Shannon im Jahre 1938 begründet wurde und eine formale Beschreibung von Schaltnetzen und Schaltwerken beinhaltet. Beiden, der Booleschen Algebra sowie der Schaltalgebra, liegt jedoch weitgehend das gleiche Prinzip der formalen Beschreibung zugrunde. Auf die Boolesche Algebra wird im Rahmen dieser Vorlesung nicht tiefergehend eingegangen.

# Definition

**Schaltnetze:** rein logisches Modell  
zeitunabhängig

$$f=f(\neg, \wedge, \vee, \#, \overline{\wedge}, \overline{\vee}, \dots)$$

**Schaltwerke:** logisches, zeitabhängiges Modell

$$f=f(t, \neg, \wedge, \vee, \#, \overline{\wedge}, \overline{\vee}, \dots)$$

# Festlegung - Schaltalgebra

Ein **Ausdruck** im Sinne der Schaltalgebra:  
Verknüpfungen von logischen **Konstanten & Variablen**

Ein **Boolescher Ausdruck** enthält nur:

- Disjunktionen:  $\vee$
- + Konjunktionen:  $\wedge$
- + Negationen:  $\neg$

**Axiome (Postulate) der binären Logik:**

$$0 \vee 0 = 0 \quad \text{Disjunktion}$$

$$0 \vee 1 = 1$$

..

$$0 \wedge 0 = 0 \quad \text{Konjunktion}$$

$$0 \wedge 1 = 0$$

..

$$a = \neg(\neg a) \quad \text{Negation}$$

# Festlegung - Schaltalgebra

Axiome (Postulate):

$$0 \vee 0 = 0 \quad \text{Disjunktion}$$

$$0 \vee 1 = 1$$

..

$$0 \wedge 0 = 0 \quad \text{Konjunktion}$$

$$0 \wedge 1 = 0$$

..

$$a = \neg(\neg a) \quad \text{Negation}$$

Gesetze:

$$a \vee 0 = a$$

$$a \vee 1 = 1$$

..

$$a \wedge 0 = 0$$

$$a \wedge 1 = a$$

..

Systeme bedürfen einer Axiomatik. Darauf aufbauend werden Gesetze entwickelt. Die Axiomatik muss ins sich schlüssig sein, die Gesetze leiten sich per Definition aus den Axiomen ab oder ihre Schlüssigkeit ist über die Axiome zu beweisen.

Unterschied zwischen Axiom und Gesetz? Beispiel Religion: Kirche

# Gesetze

Kommutativgesetz	$a \wedge b \wedge c = c \wedge b \wedge a$
Assoziativgesetz	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$
Distributivgesetz	$(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)$
Gesetz von Morgan	$\dots$
Inversionsgesetz von Shannon (Dualitätsprinzip)	
Absorptionsgesetz	
$\dots$	

Aus den grundlegenden Gesetzen lassen sich direkt die allgemeinen Rechengesetze ableiten. Deren Beweis (im Gegensatz zur z. B. arithmetischen Algebra) kann einfach geführt werden, indem für die beiden Variable a und b die möglichen Werte (0 oder 1) eingesetzt werden. Zu nennen ist das Kommutativgesetz (Vertauschungsgesetz), das Assoziativgesetz (Verbindungsgesetz), das Shannon'sche Inversionsgesetz bis hin zum Reduktionsgesetz.

Es gibt verschiedene Absorptionsgesetze.

Wichtig ist dabei, das Dualitätsprinzip zu erkennen, was z. B. im Distributivgesetz deutlich hervortritt (vgl. hierzu das Distributivgesetz der arithmetischen Algebra).

Diese Gesetze sind Gesetze der Booleschen Algebra.

Darüberhinaus treten in der Praxis noch weitere zweistellige Operationen auf wie beispielsweise NAND, NOR, EXOR, .., für die jedoch zum Teil eigene Gesetze gelten. Auf die Inhibition speziell und daraus abgeleitete Gesetze wird in diesem Kapitel im Rahmen des Sharp-Verfahrens kurz eingegangen.

# Interrupt!



Zwei Minimierungsverfahren für Schaltnetze werden im Prinzip kurz angeführt. Zuvor soll kurz die Nomenklatur, die hier verwendet wird, vorgestellt werden.

Zwischen der Booleschen Algebra und der Schaltalgebra ist klar zu unterscheiden. Die Boolesche Algebra, entwickelt von George Boole (*Investigation of the Laws of Thought*, 1854) beschäftigt sich in ihrem Kern mit der Formalisierung von Verknüpfungen von Aussagen, während die Schaltalgebra von Shannon im Jahre 1938 begründet wurde und eine formale Beschreibung von Schaltnetzen und Schaltwerken beinhaltet. Beiden, der Booleschen Algebra sowie der Schaltalgebra, liegt jedoch weitgehend das gleiche Prinzip der formalen Beschreibung zugrunde. Auf die Boolesche Algebra wird im Rahmen dieser Vorlesung nicht tiefergehend eingegangen.

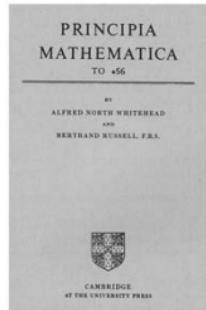
Bertrand Russel: Principia Mathematica 1910



1872 - 1970

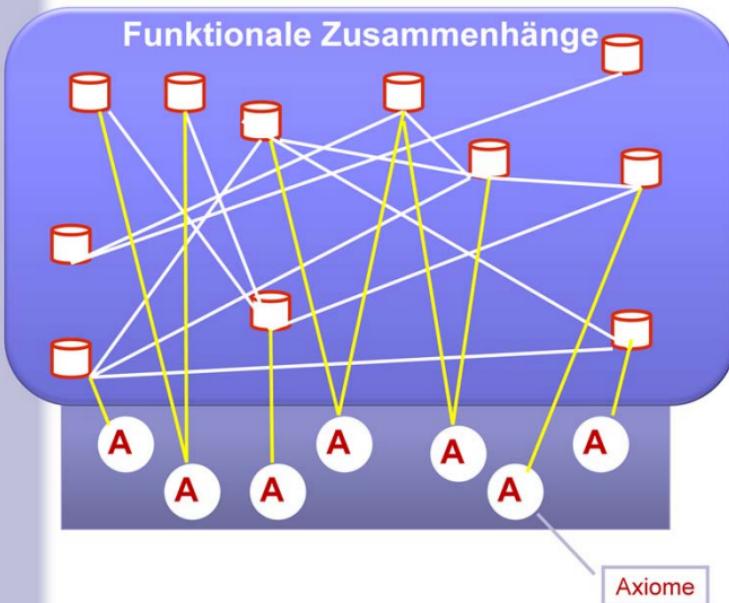
*Was ist das essentielle Problem?  
Ein Apfel ist ein Apfel. Es kommt  
keiner auf die Idee, diesen anders zu  
benennen, ausgenommen in einer  
anderen Sprache. So denkt der  
Techniker und der  
Naturwissenschaftler.*

*Dem Apfel liegt nur EINE Definition  
zugrunde. Es gibt keine >1  
Definitionen.*

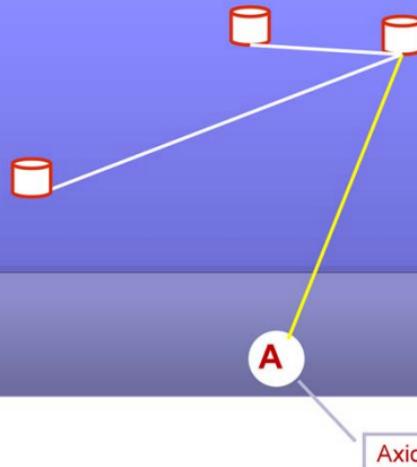


# Regel- werk

# Basis



## Funktionale Zusammenhänge



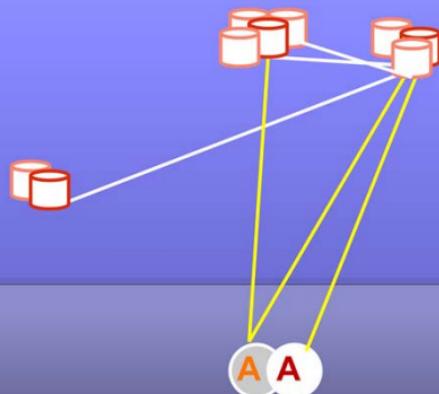
**Regel-  
werk**

**Basis**

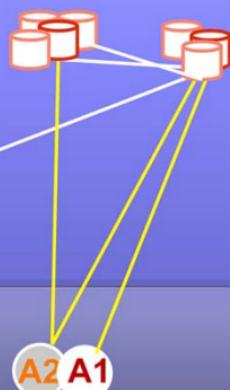
# Regel- werk

# Basis

## Funktionale Zusammenhänge



## Funktionale Zusammenhänge



Regel-  
werk

Basis

A1: Definition 1 von Symbol

A2: Definition 2 von Symbol

A3: ..

} Regelwerk dann nochmöglich?

# m-stellige Operationen

Im folgenden nur Betrachtung von

- ❖ nullstelligen +
- ❖ einstelligen +
- ❖ zweistelligen Operationen

*Drei- und noch höherstellige Operationen können immer auf die zweistellige zurückgeführt werden.*

nullstellig:       $e = x$

---

$f = 0$  Kontradiktion

$f = 1$  Tautologie

**nullstellig:**

e	=	x			
f	=	0	Kontradiktion		
f	=	1	Tautologie		

**m-stellige Operationen****einstellig:**

e	=	0	1			
f	=	0	0	Kontradiktion		
f	=	0	1	Identität		
f	=	1	0	Negation		
f	=	1	1	Tautologie		

**zweistellig:**

e <sub>0</sub>	=	0	1	0	1	
e <sub>1</sub>	=	0	0	1	1	
f	=	0	0	0	0	Kontradiktion
f	=	0	0	0	1	Konjunktion
f	=	0	0	1	0	Inhibition
f	=	0	0	1	1	Identität zu e <sub>1</sub>
f	=	0	1	..	..	..

## Normalform

- Enthält ausschließlich Boolesche Ausdrücke (keine Klammerung)
- Mögliche Funktionsdarstellung  $f_i^n$  durch lexikografische Tabellenanordnung

$e_{n-1} \quad e_{n-2} \quad \dots \quad e_1 \quad e_0 \quad i \quad f_i^n$

Bsp.: UND-Funktion:

0	0	..	0	0	0	$f_o^n$	$f_0^2 = 0$
0	0	..	0	1	1	$f_1^n$	$f_1^2 = 0$
0	0	..	1	0	2	$f_2^n$	$f_2^2 = 0$
0	0	..	1	1	3	$f_3^n$	$f_3^2 = 1$
...							

Weitergehende Erläuterungen lassen sich einfacher formulieren, wenn die sogenannte Normalform eingeführt wird. Per Definition beinhaltet sie nur die Operationen  $\wedge$ ,  $\vee$ ,  $\neg$ . Negiert werden nur die einzelnen Variablen (gemeinsame Variablen in Termen werden nicht ausgeklammert). Zugrunde gelegt ist eine lexikographische Anordnung der Terme einer Funktion in einer Wertetabelle, wie es oben beispielhaft für eine Funktion mit n Eingangsgrößen  $e_{n-1}$  bis  $e_0$  gezeigt wird.

i ist der Index des Funktionssymbols und bezeichnet in der Tabelle die Zeile. Der Exponent des Funktionssymbols  $f_i^n$  gibt die Wertigkeit der Operation wieder (wieviele Eingangsgrößen vorliegen).

Auf der Basis der Funktionssymbole lassen sich disjunktive und konjunktive Normalformen definieren.

$e_{n-1}$	$e_{n-2}$	..	$e_1$	$e_0$	i	$f_i^n$
0	0	..	0	0	0	$f_o^n$
0	0	..	0	1	1	$f_1^n$
0	0	..	1	0	2	$f_2^n$
0	0	..	1	1	3	$f_3^n$
...						

## Disjunktive Normalform (DN)

$$f = f_o \vee f_1 \vee \dots \vee f_i \vee \dots \vee f_k;$$

mit:  $f_i = f_i(e_0, e_1, \dots, e_{p-1}, \wedge, \neg)$

$$k \geq 1; \quad p \geq 0$$

Wenn jeder existierender Term  $f_i$  **verschiedene** Konjunktionsterme beinhaltet.

## Kanonische Disjunktive Normalform (KDN)

- In **jedem** existierenden Konjunktionsterm einer DN treten jeweils **alle** möglichen Variablen auf.
- Bezeichnung der Konjunktionsterme: **Minterme**
- Die Beschreibung einer KDN erfolgt durch ihre Minterme (Wert = 1).

**Beispiel:**  $f = m_3^3 \vee m_4^3 \vee m_6^3 \vee m_7^3$

$$= \bar{e}_2 e_1 e_0 \vee e_2 \bar{e}_1 \bar{e}_0 \vee e_2 e_1 \bar{e}_0 \vee e_2 e_1 e_0.$$


---

i	$e_2$	$e_1$	$e_0$	$m_i^3$	Minterme
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	0	
3	0	1	1	1	$\bar{e}_2 \wedge e_1 \wedge e_0$
4	1	0	0	1	$e_2 \wedge \bar{e}_1 \wedge \bar{e}_0$
5	1	0	1	0	
6	1	1	0	1	$e_2 \wedge e_1 \wedge \bar{e}_0$
7	1	1	1	1	$e_2 \wedge e_1 \wedge e_0$

$e_{n-1}$	$e_{n-2}$	..	$e_1$	$e_0$	$i$	$f_i^n$
0	0	..	0	0	0	$f_o^n$
0	0	..	0	1	1	$f_1^n$
0	0	..	1	0	2	$f_2^n$
0	0	..	1	1	3	$f_3^n$
...						

## Konjunktive Normalform (KN)

$$f = f_0 \wedge f_1 \wedge \dots \wedge f_i \wedge \dots \wedge f_k;$$

mit:  $f_i = f_i(e_0, e_1, \dots, e_{p-1}, \vee, \neg)$

$$k \geq 1; p \geq 0$$

Wenn jeder existierender Term  $f_i$  **verschiedene Disjunktionsterme** beinhaltet.

## Kanonische Konjunktive Normalform (KKN)

- In **jedem** existierenden Disjunktionsterm einer KN treten jeweils **alle** möglichen Variablen auf.
- Bezeichnung der Disjunktionsterme: **Maxterme**
- Die Beschreibung einer KKN erfolgt durch ihre Maxterme (Wert = 0).

**Beispiel:**  $f = M_0^3 M_1^3 M_2^3 M_5^3$

$$= (e_2 \vee e_1 \vee e_0)(e_2 \vee e_1 \vee \bar{e}_0)(e_2 \vee \bar{e}_1 \vee e_0)(\bar{e}_2 \vee e_1 \vee \bar{e}_0).$$

i	$e_2$	$e_1$	$e_0$	$M_i^3$	Maxterme
0	0	0	0	0	$e_2 \vee e_1 \vee e_0$
1	0	0	1	0	$e_2 \vee e_1 \vee \bar{e}_0$
2	0	1	0	0	$e_2 \vee \bar{e}_1 \vee e_0$
3	0	1	1	1	
4	1	0	0	1	
5	1	0	1	0	$\bar{e}_2 \vee e_1 \vee \bar{e}_0$
6	1	1	0	1	
7	1	1	1	1	



Institut für Computertechnik

i ist der Index des Maxterms.

Betrachtet man einen Maxterm und prüft seine Belegungsmöglichkeiten, so erkennt man, dass bei allen außer bei einer Wertekombination der Wert 1 ist ( $\vee$ -Funktion), was zur Wahl des Begriffs führte.

Sind in einer Wertetabelle alle Min- und Maxterme gegeben bzw. definiert, ist die Funktion f eindeutig bestimmt, was in der Praxis oft nicht gegeben ist und was zur Vereinfachung dann beitragen kann.

## Begriffserläuterung

### Minterm $\Leftrightarrow$ Maxterm

**Minterm:** Prüft man einen Minterm auf seine Belegungsmöglichkeit,  
gilt nur bei einer Wertekombination:

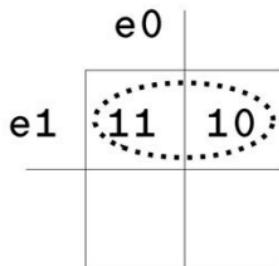
$$m_i^m = 1$$

**Maxterm:** Prüft man einen Maxterm auf seine Belegungsmöglichkeit,  
gilt bei allen außer einer Wertekombination:

$$M_i^m = 1$$

## Primterme

.. sind die **Minterme**, die sich in einer Funktion nicht weiter zusammenfassen (eliminieren) lassen.



$$e_1 e_0 \vee e_1 \bar{e}_0 = e_1,$$

## Beschreibung eines Schaltkreises allgemeiner Form (Schaltnetz: SN)

$$a_{p-1} = a_{p-1}(e_{n-1}, e_{n-2}, \dots, e_0, \wedge, \vee, \neg, \equiv, \not\equiv, \dots)$$



$$a_\beta = a_\beta(e_\alpha, e_{\alpha-1}, \dots, e_0)$$

mit:  $\alpha = 0, 1, 2, \dots, m$

$$a_{\beta-1} = a_{\beta-1}(e_\alpha, e_{\alpha-1}, \dots, e_0)$$

$\beta = 0, 1, 2, \dots, n$

$$\dots = \dots$$

$$a_0 = a_0(e_\alpha, e_{\alpha-1}, \dots, e_0)$$

Schaltnetze stellen die Realisierung logischer Operationen dar, die durch die Schaltalgebra beschrieben werden können. Diese Beschreibung basiert auf einer Abstrahierung der Physik - also einer Modellbildung -, denn es werden Größen und Parameter wie Zeit, Temperatur, mechanische Ausmaße usw. nicht berücksichtigt.

Das bedeutet, eine begrenzte Menge von Ausgangsvariablen bildet eine Funktion einer begrenzten Menge von Eingangsvariablen. Daraus lässt sich auch ablesen, worauf es bei der Beschreibung von Schaltnetzen im wesentlichen ankommt:

1. die **Definition der Eingangsschnittstelle**,
2. die **Definition der Ausgangsschnittstelle und**
3. die **formale Beschreibung der internen Logik**.

Beim **Topdown-Entwurf** wird entsprechend vorgegangen: Zuerst werden die Eingangs- und Ausgangsschnittstellen beschrieben und formal festgelegt, bevor man sich ins Detail einarbeitet (in der Praxis wird dieser Ablauf leider oft nicht eingehalten, was leicht zu Fehlern führen kann und dann Kosten verursacht).

# Vereinfachung (Optimierung) von Schaltnetzen

## Möglich über:

1. Rechenregeln
2. grafische Methoden wie KV, ..
3. numerische Methoden wie QMC, Sharp, ..
4. heuristische Verfahren



Von den grafischen Methoden zur händischen Lösung einfacher Aufgaben hat sich das Verfahren nach Karnaugh-Veith (KV) durchgesetzt. Eines der geeignetsten der numerischen Verfahren ist das Sharp-Verfahren, das relativ einfach in einem Rechner zu implementieren ist. Zu unterscheiden sind prinzipiell die Minimierung einer DN und die Minimierung einer KN.

In Design-Tools zum Entwurf von ASICs (Application Specified Integrated Circuit) werden heutzutage ausschließlich heuristische Verfahren angewendet, die im Bereich Informatik wissenschaftlich ausgiebig behandelt wurden. Für den Bereich der Elektrotechnik sind sie weniger von Interesse.

Die optimale Minimierung im Sinne der Schaltalgebra, bezogen auf Schaltnetze, ist dann erreicht, wenn die Funktion mit minimaler Anzahl von Verknüpfungen vorliegt. Das impliziert, dass eine Minimierung letztendlich immer auf die Minimierung einer einzigen Ausgangsgröße reduziert wird. Moderne Design-Tools versuchen natürlich auch diesen Aspekt zu beherrschen und suchen Optima über mehrere Ausgänge.

## KV-Methode

		e <sub>1</sub>
e <sub>0</sub>	f <sub>3</sub> <sup>2</sup> = 1	f <sub>1</sub> <sup>2</sup> = 0
	f <sub>2</sub> <sup>2</sup> = 1	f <sub>0</sub> <sup>2</sup> = 0

Bei der Vereinfachung über die DN werden die Minterme (Wert = 1) und bei der Vereinfachung über die KN die Maxterme (Wert = 0) zusammengefasst.

.. wenn alle Terme (= Min- oder Maxterme) festgelegt sind,

*sonst: DC-Terme (Don't-care-Terme) berücksichtigen.*

### KV-Methode:

	e <sub>1</sub>	
e <sub>0</sub>	$f_3^2 = 1$	$f_1^2 = 0$
	$f_2^2 = 1$	$f_0^2 = 0$

### Vereinfachung über Formalismus:

$$e_1 e_0 \vee e_1 \bar{e}_0 = e_1,$$

### Beispiel:

	e <sub>1</sub>	
e <sub>0</sub>	1      X      1      1	
	0      X      0      0	

$$f = e_0.$$

Grund, warum es so möglich ist: Eine DN oder eine KN beschreibt jeweils vollständig die entsprechende Funktion. Ob eine Funktion also als DN oder als KN beschrieben wird, ist im Prinzip gleichgültig. Auf die Implementierung und Berechnungszeit im Rechner kann dies natürlich gravierende Bedeutung bezüglich Laufzeiten haben.

## QMC

### Quine-McClusky-Verfahren

- in den 70er Jahren „fast“ in jedem Lehrbuch
- dann keine Bedeutung mehr
- wieder interessant bei Einführung von programmierbaren Bausteinen (PAL, EPLD, .., FPGA)
- heute weitgehend abgelöst durch heuristische Verfahren
- ..
- eignen sich zur Erläuterung von ternärer Logik
- ..
- Ziel: Optimierung einer Ausgangsgröße der Form

$$a = f(e_n, e_{n-1}, \dots, e_0, \wedge, \vee, \neg)$$

Für Lehrbücher der logischen Schaltungsalgebra war es bis in die siebziger Jahre üblich, neben dem Karnaugh-Veith-Diagramm-Verfahren (KV-Verfahren) auch das Quine-McCluskey-Verfahren (QMC-Verfahren) aufzuführen. Durch die zunehmende Komplexität digitaler Bausteine kamen jedoch solcherlei Verfahren bei den meisten Entwicklern digitaler Schaltungen immer weniger zur Anwendung, so dass die Lehrbücher in den folgenden Jahren das QMC-Verfahren oft nicht einmal erwähnten. Die Einführung von Kundenschaltkreisen, PALs, EPLDs usw. hat die Situation wieder geändert. Um möglichst viele logische Funktionen in einen derartigen Baustein hineinpacken zu können, müssen entworfene Schaltungen entsprechend optimiert werden, gleichgültig, ob es sich um Schaltnetze oder sequentielle Schaltwerke handelt, die auf der Basis von Automatenmodellen entworfen werden.

Selbstverständlich sind in CAD-Anlagen Optimierungsprogramme zum Entwurf logischer Schaltungen enthalten, doch sollte der Anwender zumindest prinzipiell über den Aufbau derartiger Einheiten Bescheid wissen.

Da das QMC-Verfahren relativ einfach ist und mit ihm auch eine numerische Methode der Schaltnetzminimierung erläutert werden kann, wird es im folgenden kurz dargelegt, wobei jedoch auf eine vollständige Aus- und tiefergehende Beweisführung verzichtet wird. Das QMC-Verfahren hat nämlich in der Praxis völlig an Bedeutung verloren, da es inzwischen effektivere Verfahren gibt, wie das auch von Quine McCluskey entwickelte Sharp-Verfahren.

## Voraussetzung:

- Disjunktive Normalform (DN)

wenn „per Hand“:

- besser KDN

### Beispiel:

**DN:**  $y = \bar{c}\bar{b}a \vee d\bar{c}\bar{b}a \vee \bar{d}\bar{c}ba \vee dc\bar{b} \vee cb\bar{a} \vee \bar{d}\bar{c},$

**KDN:**  $y = d\bar{c}\bar{b}a \vee \bar{d}\bar{c}\bar{b}a \vee dc\bar{b}a \vee \bar{d}\bar{c}ba \vee$   
 $\vee d\bar{c}ba \vee dc\bar{b}\bar{a} \vee \bar{d}\bar{c}ba \vee \bar{d}\bar{c}ba \vee \bar{d}\bar{c}\bar{b}a$

## QMC basiert auf

- wiederholte Anwendung von:

$$e_1 e_0 \vee e_1 \bar{e}_0 = e_1,$$

- bis Primterme übrig bleiben.

Vorgehensweise „per Hand“:

1.  $DN \Rightarrow$  Binäraquivalente (Dreiwertlogik)
2.  $DN \Rightarrow KDN$
3. Findung aller Primterme
4. Eliminierung der redundanten Primterme

DN:	d	c	b	a
-	0	0	1	
1	1	0	1	
0	0	1	1	
1	1	1	-	
-	1	1	0	
0	0	-	-	

## 1. DN $\Rightarrow$ Binäräquivalente (Dreiwertlogik)

<b>Variable:</b>	$\rightarrow$	1
<b>negierte Variable</b>	$\rightarrow$	0
<b>fehlende Variable</b>	$\rightarrow$	-

$$f = \bar{c}ba \vee dc\bar{b}a \vee \dots$$

## 2. DN (in Form der Binäräquivalente) $\Rightarrow$ KDN



Das QMC-Verfahren (wie auch das Sharp-Verfahren) geht zwecks einer einfacheren Handhabung davon aus, die Ausgangsgleichung in die ternäre Logik (ternären Raum) zu transformieren. In der weiteren Folge wechselt man dann öfters zwischen den binären und ternären Räumen.

Auf das Beispiel von Gleichung Folie 21 angewendet, erhält man danach die Werte von der Tabelle.

Aus der DN wird die KDN gebildet. Das heißt, die DN wird erweitert, indem für jede fehlende Variable eines Terms zwei neue Terme gebildet werden, wobei der fehlenden Variablen einmal der Wert 0 und einmal der Wert 1 zugewiesen wird. Aus -001 (erste Zeile) ergeben sich so die beiden Terme 0001 und 1001. Aus der Sicht der Booleschen Algebra erhält man die Mintermfunktion.

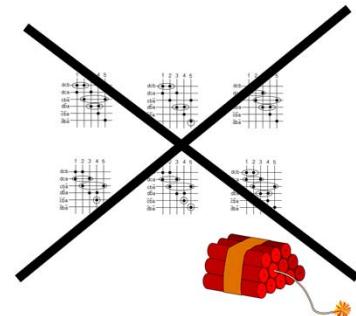
# Sharp-Verfahren

## Nachteil von QMC:

- ❖ hohe Laufzeiten
- ❖ hoher Speicherbedarf
- ❖ ineffiziente Realisierungen

### Beispiele:

- realisierte Tools liefern oft nur eine Lösung
- nicht immer die optimale Lösung
- ..



Die Prämissen für die Entwicklung sind:

- Das Verfahren soll alle Booleschen minimalen Lösungen finden.
- Das Verfahren setzt auf einer disjunktiven Normalform (DN) auf, die beispielsweise über eine Funktionstabelle aus einer Problemstellung heraus gewonnen wird.
- Die Rechenzeit und der Speicherplatzaufwand müssen in einem vertretbaren Rahmen bleiben, damit ein PC die Anwendung eines Verfahrens erlaubt.

Im Gegensatz zu den vorausgegangenen Jahren wird in dieser und den kommenden Vorlesungen dieses Kapitel 2 zugunsten von Kapitel 5 und 6 nicht mehr ausführlich abgehandelt, sondern nahezu auf Null komprimiert. Auf Herleitungen und Beweisführungen wird vollständig verzichtet, da es nun nur noch das Ziel sein soll, ein ungefähres Gefühl für die ternäre Logik und deren Methode zu gewinnen, damit man versteht, welch enorme Möglichkeiten über die einfache Boolesche Algebra hinaus auch für heuristische Verfahren zur Verfügung stehen.

# Sharp-Verfahren

**Definitionen:**

**ON-Vektor:**

**Minterme**

**OFF-Vektor:**

**Maxterme**

**DC-Vektor:**

**Don't care-Terme**

**Dualsystem**



**Ternärsystem**

N r.	a	b	c	Z
0	0	0	0	x
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	x

$$\text{ON} = \begin{vmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{vmatrix}$$

$$\text{OFF} = \begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{vmatrix}$$

$$\text{DC} = \begin{vmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{vmatrix}$$

**Vektor:** *disjunktive Verknüpfung von Termen*

Erfüllt werden diese Prämissen im vollen Umfang durch das im folgenden erläuterte Sharp-Verfahren. Es werden, wie im QMC-Verfahren auch, die Terme der zu minimierenden Funktion in die ternäre Logik transformiert, in dieser Ebene die so gewonnenen Ausdrücke nach speziell hierfür aufgestellten Regeln tabellarisch bearbeitet und letztendlich die Ergebnisse in die binäre Logik zurücktransformiert.

Das eigentliche Sharp-Prinzip stellt eine Minimierungsoperation dar. Wie im vorausgegangenen Abschnitt schon gezeigt, fallen bei der Minimierung einer Schaltungsfunktion jedoch *zwei* Minimierungsaufgaben an:

Erstens sind die Primterme zu finden (= minimale Funktion aller disjunktiven Terme, die sich nicht weiter vereinfachen lassen), und zweitens sind die redundanten Primterme zu eliminieren (= Minimierung auf die absolut notwendige Zahl von Primtermen, die die Ausgangsfunktion voll beschreiben).

Um die Darstellung des QMC-Verfahrens möglichst einfach zu halten und die Beschreibung nicht zu umfangreich werden zu lassen, werden die Don't-Care-Terme im Allgemeinen unterschlagen. Für die Erläuterung des Sharp-Verfahrens könnten sie hier dagegen von Anfang an berücksichtigt werden, da sie sich hier leicht integrieren lassen, was aber nicht konsequent geschehen soll.

Die Vorgehensweise zur Erläuterung des Sharp-Verfahrens ist nun:

(a)

Definition des Sharp-Produktes,

(b)

Darlegung und Anwendung des Sharp-Verfahrens auf eine Boolesche Funktion zur Findung aller Primterme mit Hilfe des Sharp-Produktes und Elimination aller redundanten Terme ebenfalls mit Hilfe des Sharp-Produktes.

# Sharp-Produkt → „Subtraktionsprozess“

**Beispiel:**

$$y = \overline{\overline{b}} \overline{a} \vee \overline{\overline{b}} \overline{c}$$

ON-Terme von y



$$\bar{y} = (b \vee a)(b \vee c)$$

$$= b \vee c a$$



**Definition:** d-Vektor:

$$d(c, b, a) = 1$$

**Es gilt:**

$$F(d) = F(y) + F(\neg y) \Rightarrow F(d) - F(y) = F(\neg y)$$

Stellt man für eine Schaltfunktion eine Wertetabelle auf, so ergibt sich für jede Kombination der Eingangsvariablen genau ein Ausgangswert. Die Ausgangswerte sind den entsprechenden Eingangskombinationen durch eine Boolesche Funktion fest zugeordnet und somit charakteristisch für die Funktion. Sie können die logischen Werte 1 und 0 annehmen. In der Praxis kommt es allerdings häufig vor, dass durch äußere Randbedingungen einige der Eingangskombinationen nicht auftreten oder der sich einstellende Wert irrelevant ist. Für die technische Realisierung ist dann der Wert des Ausgangs bei diesen Eingangskombinationen gleichgültig (engl.: don't care). In der Wertetabelle kennzeichnet man die Ausgangsfunktion derartiger **redundanter** Eingangskombinationen mit einem x, womit eine Wertetabelle demnach letztendlich aus 1-Termen, 0-Termen und x-Termen besteht. Die Menge der 1-Terme (Minterme) stellt den **ON-Vektor**, die Menge der 0-Terme (Maxterme) den **OFF-Vektor** und die Menge der x-Terme den **DC-Vektor** dar. ON-, OFF- und DC-Vektoren zusammen ergeben somit die gleiche Information wie die Wertetabelle beziehungsweise wie die KDN und KKN. Für die Manipulation von Schaltfunktionen sowie für die Notation von Minimierungsalgorithmen bietet die Darstellung in Form von Vektoren jedoch gewisse Vorteile, so dass im weiteren hauptsächlich diese Darstellung Verwendung finden soll.

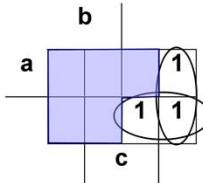
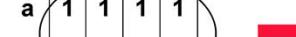
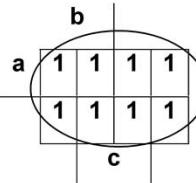
Wird von einem KV-Diagramm ausgegangen, kann die Sharp-Operation als "Subtraktionsprozess" gedeutet werden.

# Sharp-Produkt → „Subtraktionsprozess“

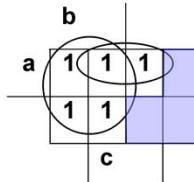
*Es gilt:*

$$F(d) - F(y) = F(\neg y)$$

$$d(c, b, a) = 1 \quad - \quad y = \bar{a}\bar{b} \vee \bar{c}\bar{b}$$



$$= \bar{y} = ca \vee b$$



Sharp-Operation

Die Operation kann man so deuten, als sei von der Funktion

$$d(c, b, a) = 1$$

die Funktion  $F(y)$  "abgezogen" worden.

# Definition: Sharp-Operation

*Angewendet auf zwei Vektoren, bestehend aus Primtermen (PT):*

$$\text{PT}(\underline{\alpha} \# \underline{\beta}) = \text{PT}(\underline{\alpha}) \wedge \overline{\text{PT}(\underline{\beta})}$$

≡ Inhibition

$$\text{mit } \underline{\alpha} = \begin{vmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_k \\ \vdots \\ \alpha_{n-1} \end{vmatrix}; \quad \underline{\beta} = \begin{vmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \\ \vdots \\ \beta_{n-1} \end{vmatrix}$$

Vorausgesetzt werden Vektoren:

- mit einer Komponente oder

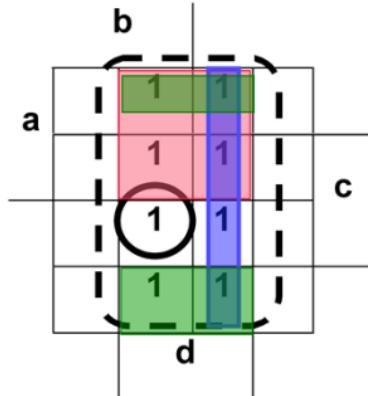
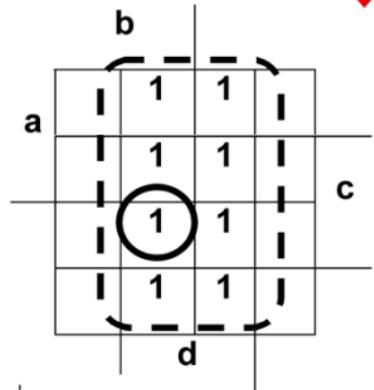
- mit gleicher Anzahl von Komponenten

### Beispiel mit einer Vektor-Komponente:

$$\alpha = d$$

$$\beta = d \ c \ b \ \bar{a}$$

$$\begin{aligned}\Rightarrow \alpha \# \beta &= d (a \vee \bar{b} \vee \bar{c} \vee \bar{d}) \\ &= d \ a \vee d \ \bar{b} \vee d \ \bar{c}.\end{aligned}$$



# Sharp-Produkt Vektor $\leftrightarrow$ Term

## 1. Sharp-Produkt: Term # Vektor: $\alpha \# \underline{\beta}$

$$\omega_0 = \alpha \# \beta_0$$

$$\omega_1 = \omega_0 \# \beta_1 = (\alpha \# \beta_0) \# \beta_1$$

$$\omega_2 = \omega_1 \# \beta_2 = ((\alpha \# \beta_0) \# \beta_1) \# \beta_2$$

:

$$\omega_{m-1} = \omega_{m-2} \# \beta_{m-1} = ((\dots(\alpha \# \beta_0) \# \beta_1) \# \dots \# \beta_{m-1})$$

$$\omega_{m-1} = \alpha \# \underline{\beta},$$

$$\text{mit } \underline{\beta} = \begin{vmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{m-1} \end{vmatrix}$$

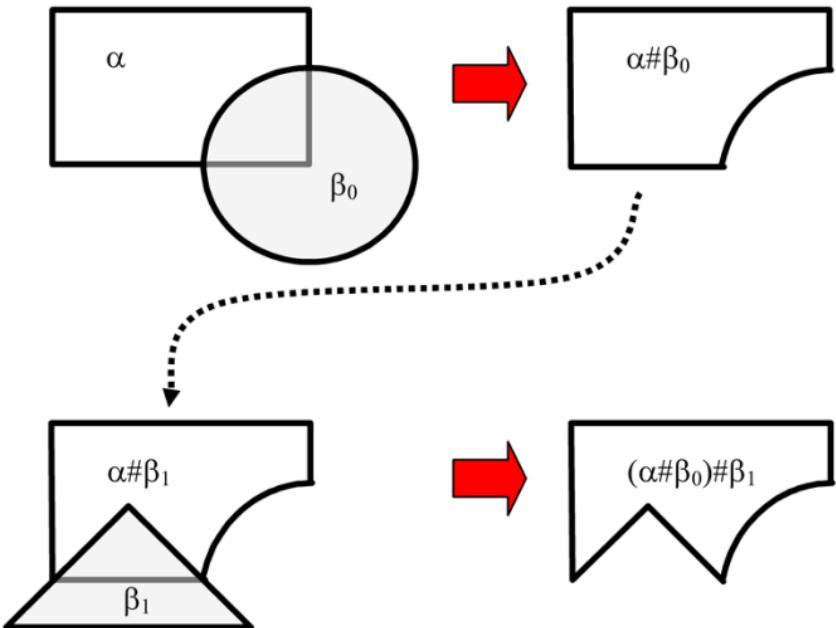
Für die Vereinfachung nach Sharp ist es sinnvoll, zwei Produkte zu definieren:

- (a) Term # Vektor und
- (b) Vektor # Term.

Der Ausdruck kann als sukzessiver Flächensubtraktionsprozess gedeutet werden.

## 1. Sharp-Produkt: Term # Vektor: $\alpha \# \underline{\beta}$

$$\alpha \# \underline{\beta} = ((\dots (\alpha \# \beta_0) \# \beta_1) \# \dots \# \beta_{m-1})$$



# Sharp-Produkt Vektor $\leftrightarrow$ Term

2. Sharp-Produkt: Vektor # Term :  $\underline{\alpha} \# \beta$

definiert wird eine Funktion bei der die Terme  $\alpha_0, \alpha_1, \dots$  jeweils mit  $\beta$  "ver-sharpt" werden:

$$\omega_i = \alpha_i \# \beta$$

das führt zu:

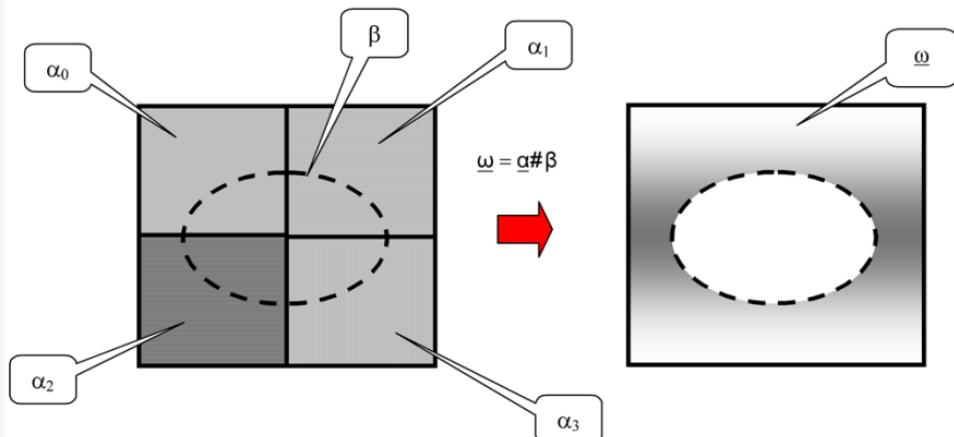
$$\underline{\omega} = (\alpha_0 \# \beta) \cup (\alpha_1 \# \beta) \cup (\alpha_2 \# \beta) \cup (\alpha_3 \# \beta) \cup \dots$$

$\cup$  ist der kommutative Operator für eine Vereinigungsmenge:  
*er soll beinhalten, dass eine OPTIMALE Zusammenfassung der Ausdrücke erfolgt*  
*⇒ hinter "∪" steht also die Bildung ALLER Primterme (!!)* **WIE?**

## Sharp-Produkt: Vektor # Term : $\underline{\alpha} \# \beta$

also:

$$\underline{\omega} = (\underline{\alpha}_0 \# \beta) \cup (\underline{\alpha}_1 \# \beta) \cup (\underline{\alpha}_2 \# \beta) \cup (\underline{\alpha}_3 \# \beta) \cup \dots$$



Die Vereinigung der  $\omega_i$ -Elemente erzeugt dabei  $\underline{\omega}$ , also einen Vektor, dessen einzelne Elemente nicht mit den  $\omega_i$ -Elementen vor der Vereinigungsoperation identisch sein müssen.

# Aber:

Worin liegt der Sinn & Erfolg der Methode?

J. P. Roth und R. M. Karp wiesen in Minimization over Boolean  
Graphs (IBM j. Res. Dev. 6.7-38, 1962) nach, dass eine doppelte  
ver-Sharpung ALLE Primterme (PT) erzeugt!

# Primtermermittlung einer Funktion mittels Sharp (ohne DC-Terme)

Ist:

FPNR

$$d = d(d_0, d_1, \dots, d_{n-1}) = 1 \quad \rightarrow \{ \dots \}$$

gilt nach der Definition der ON, OFF und DC-Vektoren:

$$\begin{aligned} d &= \text{on} \vee \text{off} \\ \text{on} &= d \# \text{off} \\ \text{off} &= d \# \text{on} \end{aligned}$$

denn:

$$\begin{aligned} \text{on} &= d \wedge \overline{\text{off}} = \overline{\text{off}} \\ \text{off} &= d \wedge \overline{\text{on}} = \overline{\text{on}} \end{aligned}$$

## Primtermermittlung einer Funktion mittels Sharp (ohne DC-Terme)

Eine doppelte ver-Sharpung ALLER Primterme (PT) bedeutet:

$$\text{PT}(y) = d \# [d \# \text{on}]$$

Warum kann das richtig sein?

Das bedeutet, eine doppelte Ver-Sharpung führt auf den Ausgangsvektor zurück. Eine doppelte Ver-Sharpung führt aber auch stets über die Operation

Vektor # Term.

$$\text{PT}(y) = d \# [d \# \text{on}]$$

$$\begin{aligned}\text{PT}(Y) &= d \# [d \# \text{on}] \\ &= d \# [d \wedge \neg \text{on}] \\ &= d \# \neg \text{on} \\ &= d \wedge \text{on} \\ &= \text{on}.\end{aligned}$$

1. doppelte Ver-Sharpung führt auf den ON-Vektor zurück
2. führt aber über die Operation: Vektor # Term:

$$\underline{\alpha} \# \beta = (\alpha_0 \# \beta) \cup (\alpha_1 \# \beta) \cup \dots \cup (\alpha_{n-1} \# \beta).$$

Ausgehend von einer DN wird eine Vereinigungsmenge über eine Optimierung gebildet.

# Sharp-Verfahren

Sharp-Verfahren generiert so ALLE möglichen Primterme!

Frage:

*Was ist mit DC-Termen?*

Frage:

Wie kommt man zur minimalen DN?

*= Elimination aller relativen Primterme?*

Nach Roth & Karp gilt:

$$\text{PT}(Y) = d \# [d \# (\text{ON} \cup \text{DC})]$$

*Eine zweimalige Anwendung des Sharp-Operators liefert alle PT!*

DC-Terme werden in diesem Schritt als Minterme angenommen:

+: Sicherstellung für optimale Zusammenfassung  
(Vereinfachung) der ON- und der DC-Termen

-: Zusätzliche "PT" aus reinen DC-Termen sind möglich,  
die im 2. Schritt (PT-Elimination) wieder entfernt werden  
müssen.

Was bedeutet:

$$PT(Y) = d \# [d \# (ON \cup DC)]$$

?

*Ursprungsfunktion:*

$$y = f(on, dc)$$



*Zwischenfunktion:*

$$off = d \# (on \cup dc)$$



*Ergebnisfunktion  
(vollständiger Satz  
aller Primterme) :*

$$PT(y) = d \# off$$

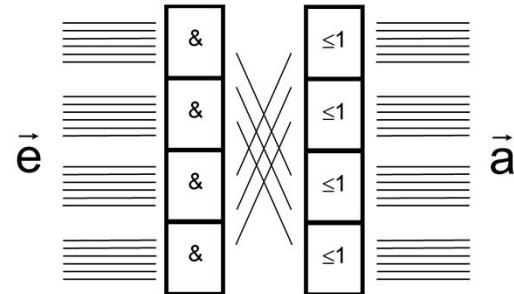
## 2.4 Multilevel-Design

Bisher:

$$a = f(\vec{e})$$

Praxis ist aber zumeist:

$$\vec{a} = f(\vec{e})$$

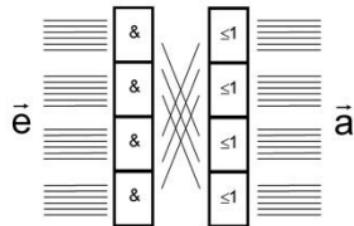


Die bisher dargestellten Methoden haben ausschließlich das Ziel, *eine* Funktion zu minimieren. In der Praxis stellt sich jedoch zumeist die Aufgabe, dass mehrere Ausgangsgrößen von mehreren Eingangsgrößen abhängen.

Ziel einer vollständigen Minimierung muss es deshalb sein, eine Vereinfachung des Gleichungssystems insgesamt zu erreichen. Legt man die Sharp-Methode zugrunde, ist es sinnvoll, die Minimierung der einzelnen Ausgangsfunktionen vorzunehmen und dann in einem zweiten Schritt die erhaltenen Ergebnisse auf eine mögliche Minimierung hin zu überprüfen. Direkt einsichtig ist, dass die Primterme der unterschiedlichen Lösungen miteinander verglichen werden. Zu berücksichtigen ist, dass durch die Minimierung im ersten Schritt für jede einzelne Ausgangsgröße mehrere Lösungen existieren können, was jedoch im folgenden, um die Thematik nicht unnötig zu verkomplizieren, nicht berücksichtigt wird.

## Allgemeines Gleichungssystem

$$\vec{a} = f(\vec{e})$$



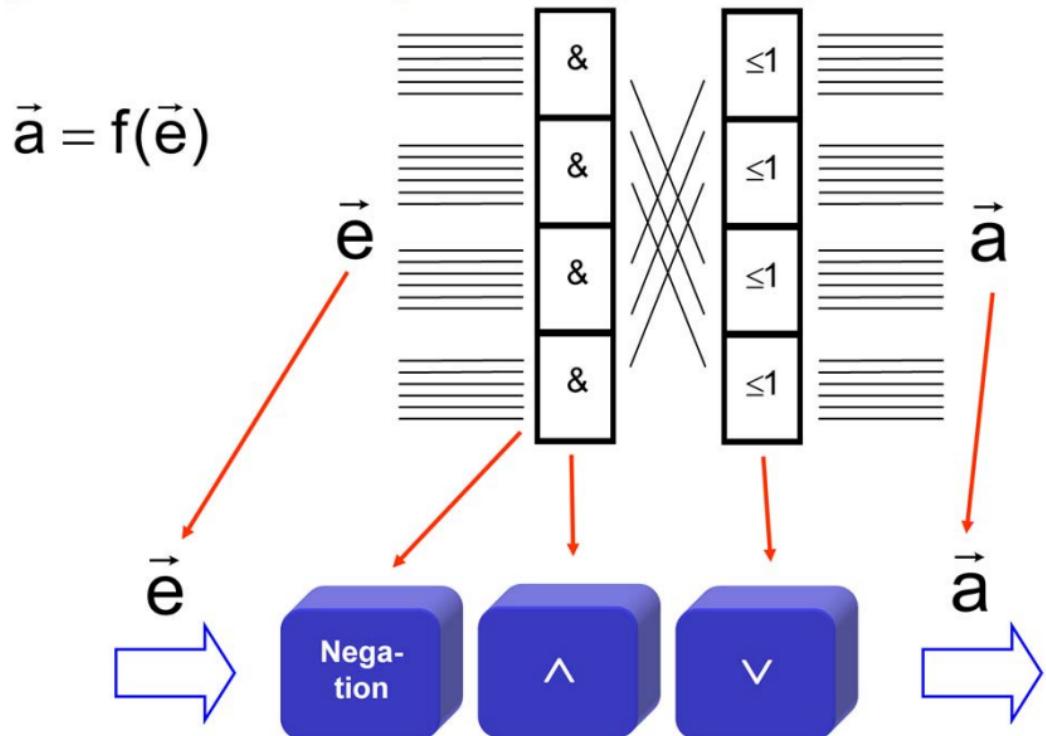
$$a_{q-1} = f(e_{p-1}, e_{p-2}, \dots, e_0)$$

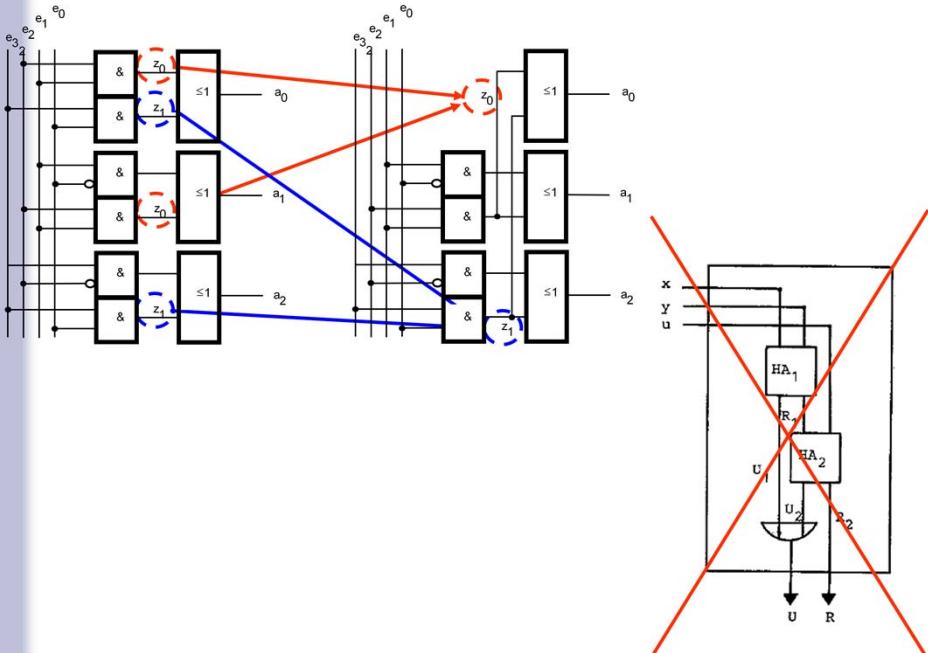
$$a_{q-2} = f(e_{p-1}, e_{p-2}, \dots, e_0)$$

$$\vdots$$

$$a_0 = f(e_{p-1}, e_{p-2}, \dots, e_0)$$

## Ergebnis nach der Optimierung: 3-stufiges SN





Der Vollständigkeit halber sei noch auf die sequentielle Modulbildung verwiesen, um Verknüpfungen einzusparen, was früher ein gängiges Verfahren war, heute jedoch keine Bedeutung mehr besitzt, da im allgemeinen hohe Laufzeitanforderungen gestellt werden. Ein gängiges Beispiel ist der Volladdierer, der in Form von zwei Halbaddierern realisiert werden kann. Vergleicht man solch eine Schaltung mit der einer minimalen DN, so wird deutlich, dass sequentiell modularisierte Schaltungen zwei Nachteile beinhalten:

1. Die Durchlaufzeit der Schaltung ist relativ hoch, und
2. über die verschiedenen Pfade innerhalb der Schaltung ergeben sich unterschiedliche Laufzeiten, was zu Spikes führt,

was für ein ASIC-Design überhaupt nicht in Frage kommt, also heute keinerlei eine Rolle mehr spielt.

Letztendlich sollte auf einen weiteren Entwicklungsaufwand zumindest hingewiesen werden, der allerdings prinzipiell bei der Minimierung von Schaltkreisen zu berücksichtigen ist: Manchen Schaltkreis-Technologien werden spezielle Basisschaltungen zugrunde gelegt, beispielsweise NOR-Schaltkreise oder Multiplexer (Kapitel 5 dieses Skriptes). Dies muss bei der Optimierung selbstverständlich berücksichtigt werden.

# Digitale Integrierte Schaltungen

384.086

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K3-1

p. 1  
18.01.2013 13:12:57

Ziel ist es, die digitale Schaltungsentwicklung im Sinne des Top-Down-Entwurfs rein mathematisch anzugehen. Zugrunde gelegt wird ein Automatenmodell, über das im Prinzip *alle* digitale Schaltungen entwickelt werden können. Die Grenze wird allein durch die Komplexität gebildet. Eine gewisse Problematik bilden asynchrone Schaltwerke, die heute ein wesentliches Forschungsthema bilden. In dieser Vorlesung wird soweit darauf eingegangen, wie sie heute für einen Schaltungsentwurf in der Praxis notwendig sind.

Wie studiert man effizient? .. und man hat doch Spaß daran.

.. man beendet den Master so schnell wie möglich und steigt in eine Diss. ein.

## Wie ist das möglich?

Man baut das Masterstudium inhaltlich Baustein für Baustein aufeinander auf, wobei Teile immer wieder verbessert übernommen werden können.

Bsp.:

Man macht für die "Vertiefung" ein Thema aus, was einem interessiert. Das wählt man auch als Dilpomarbeit und später vielleicht sogar als Dissertation, immer in möglichst leicht abgeänderte Form.

# Kapitel 3

## Schaltwerke

- Einführung
- Speicherelemente
- Synchrone u. asynchrone Schaltwerke
- Mealy- u. Moore-Automaten
- Races und Hazards
- Synthese und Analyse

# Die gute Tante Eugenie

von *Mike W. Shields*

Ein geschätzter Kollege, Herr Dr. F-X. Reid, ist mit einer guten Tante geschlagen, die sich in vorgerücktem Alter befindet, wohlhabend ist und an seniler Demenz leidet. Er erzählt, dass sie auf wenige Reize reagiert; genauer gesagt, reagiert sie nur auf drei Dinge: auf Ghetto-Blaster, mit denen auf der Straße vor ihrer Wohnung ein Höllenlärm veranstaltet wird, auf Video-Scheußlichkeiten und auf Gin, wenn eine Flasche vor ihrer Nase hin und her bewegt wird.

Diese drei Dinge haben in Abhängigkeit von ihrer Laune verschiedene Auswirkungen auf sie. Reid berichtet:

"Es ist fast unmöglich, sie morgens zu wecken. Sie können alles versuchen - halten Sie ihr eine Flasche Gin unter die Nase, spielen Sie "The Ghoul That Ate Guildford" – es ..



Bisher habe ich keine bessere Einführung in die Automatentheorie gefunden als die von *Mike W. Shields*. Ich gebe sie deshalb an dieser Stelle wörtlich wieder.

Zu Ghetto-Blaster: Da ich mich noch geistiger Gesundheit erfreue, hoffe ich inbrünstig, dass diese Dinger veralten und dass der Satz folglich nicht mehr höflich gebraucht wird. In dieser Hoffnung definiere ich heute diesen Terminus, damit er künftigen Generationen erhalten bleibt. Ein Ghetto-Blaster ist ein lautstarkes Objekt, das moral- und geschmacklose Leute mit sich herumtragen, um unschuldigen Bürgern in der Öffentlichkeit Verdruss und Ärger zu bringen.

Zu Scheußlichkeiten: In derselben Stimmung und aus ähnlicher Veranlassung möchte ich Ihnen erklären, dass es sich bei einer Video-Scheußlichkeit um das filmische Äquivalent eines Ghetto-Blasters handelt.

Zu Gin: Ich bin sicher, dass dieses Wort nicht näher erläutert zu werden braucht.

.. es ist vollkommen egal. Sie schnarcht heftig und schläft weiter, manchmal wochenlang. Die Familie war gezwungen, einen "Ghetto-Blaster" zu kaufen - das ist das einzige, das zu ihr durchdringt. Aber sehen Sie, wenn sie erst einmal wach ist, ist sie eine verträgliche alte Dame. Nehmen Sie beispielsweise die "Video-Scheußlichkeiten". Sie würde nie auch nur davon träumen, sich derart unviktorianisch zu verhalten und sich in dieser Stimmung solche Dinge anzuschauen. Aber wenn einer dieser jungen Flegel mit einem "Ghetto-Blaster" an ihrem Fenster vorbeistolziert, dann, du meine Güte! Sie reagiert sehr gereizt und predigt lange über das Thema, dass man wieder dazu übergehen sollte, Ohren abzuschneiden. Ich fürchte, dass es genau das richtige ist, ihr eine Flasche Gin unter die Nase zu halten, wenn sie gut gelaunt ist. Sie fängt dann sofort an, sie herunterzukippen und wird in peinlicher Weise "rührselig".

# Tabelle der Übergänge

Stimmungen	Reize			Reaktionen		
	Ghetto-B	Video	Gin	Ghetto-B	Video	Gin
schlafend	freundlich	schlafend	schlafend	aufwachen	schnarchen	schnarchen
freundlich	gereizt	freundlich	rührselig	predigen	verweigern	trinken
gereizt	gereizt	freundlich	abscheulich	predigen	gackern	trinken
rührselig	abscheulich	rührselig	schlafend	schießen	gackern	zusammenbrechen
abscheulich	abscheulich	rührselig	schlafend	schießen	gackern	zusammenbrechen

nach Shields

Notwendig: ein bisschen Ordnung!



Wie bei den meisten Problemen, die verbal (um nicht zu sagen: langatmig) dargestellt werden, besteht eine gute Strategie darin, die Angaben kohärenter und organisierter zu präsentieren. Die gute Tante Eugenie lässt sich gemäß zusammenfassen.

In der Tabelle entspricht jede Zeile einer ihrer fünf Stimmungen. Der linke Teil der Tabelle gibt an, wie ein Reiz ihre Stimmung ändern kann. Wenn Sie zum Beispiel wissen möchten, was geschieht, wenn ihr während des Schlafens ein Gin angeboten wird, schauen Sie am Schnittpunkt der Zeile "schlafend" mit der Spalte "Gin" nach. Dort finden Sie das Wort "schlafend" - das Angebot bleibt wirkungslos. Wenn Sie andererseits wissen möchten, wie ihre Reaktion aussieht, dann suchen Sie den Schnittpunkt der Zeile "schlafend" mit der Spalte "Gin" im rechten Teil der Tabelle heraus. Sie werden sehen, dass sie in diesem Fall lediglich schnarcht.

Mit der Tabelle ist es auch ganz einfach, die aufeinanderfolgenden Wirkungen einer Reihe von Reizen zu bestimmen. Nehmen wir beispielweise an, dass sie schläft und dass wir nacheinander (i) laute Rockmusik in der Nähe ihres Fensters spielen, (ii) ihr eine Flasche anbieten und (iii) die Rockmusik wiederholen.

Die Auswirkung von (i) besteht darin, dass sie aufwacht und freundlich gelaunt ist. In dieser Stimmung wirkt sich (ii) derart aus, dass sie trinkt und rührselig wird. In der rührseligen Laune reagiert sie auf (iii) insofern, als sie auf uns schießt und ihre Laune abscheulich wird. Bei gegebener Eingabefolge:

Ghetto-B Gin Ghetto-B

und unter der Voraussetzung, dass sie anfangs schlief, ergibt sich die folgende Ausgabefolge:

aufwachen trinken schießen.

## Tabelle der Übergänge ⇒ Übergangstabelle

Stimmungen	Reize			Reaktionen		
	Ghetto-B	Video	Gin	Ghetto-B	Video	Gin
schlafend	freundlich	schlafend	schlafend	aufwachen	schnarchen	schnarchen
freundlich	gereizt	freundlich	rührselig	predigen	verweigern	trinken
gereizt	gereizt	freundlich	abscheulich	predigen	gackern	trinken
rührselig	abscheulich	rührselig	schlafend	schießen	gackern	zusammenbrechen
abscheulich	abscheulich	rührselig	schlafend	schießen	gackern	zusammenbrechen

Reize	Stimmung jetzt	Stimmung nachher	Reaktionen
Video	schlafend	schlafend	schnarchen
Gin	schlafend	schlafend	schnarchen
Ghetto-Bl	schlafend	freundlich	aufwachen

Diese Art von Tabelle ist sehr unübersichtlich. Man kann sie deshalb umzeichnen und kommt auf eine Tabelle mit "Reize", "Stimmung jetzt", "Stimmung nachher" und "Reaktionen", was der Übergangstabelle für Schaltwerke (zeitabhängig) entspricht.

# Übergangs-tabelle

The diagram illustrates the mapping of inputs and states to a transition table. Four arrows point from boxes labeled "Input", "Zustand vorher", "Zustand nachher", and "Output" to the corresponding columns in the table.

**Input:** Video, Gin, Ghetto-Bl

**Zustand vorher:** schlafend, schlafend, schlafend, freundlich, freundlich, freundlich, rührselig, rührselig, rührselig, abscheulich, abscheulich, abscheulich, gereizt, gereizt, gereizt

**Zustand nachher:** schlafend, schlafend, freundlich, gereizt, rührselig, rührselig, abscheulich, schlafend, rührselig, schlafend, gereizt, gereizt, freundlich, abscheulich

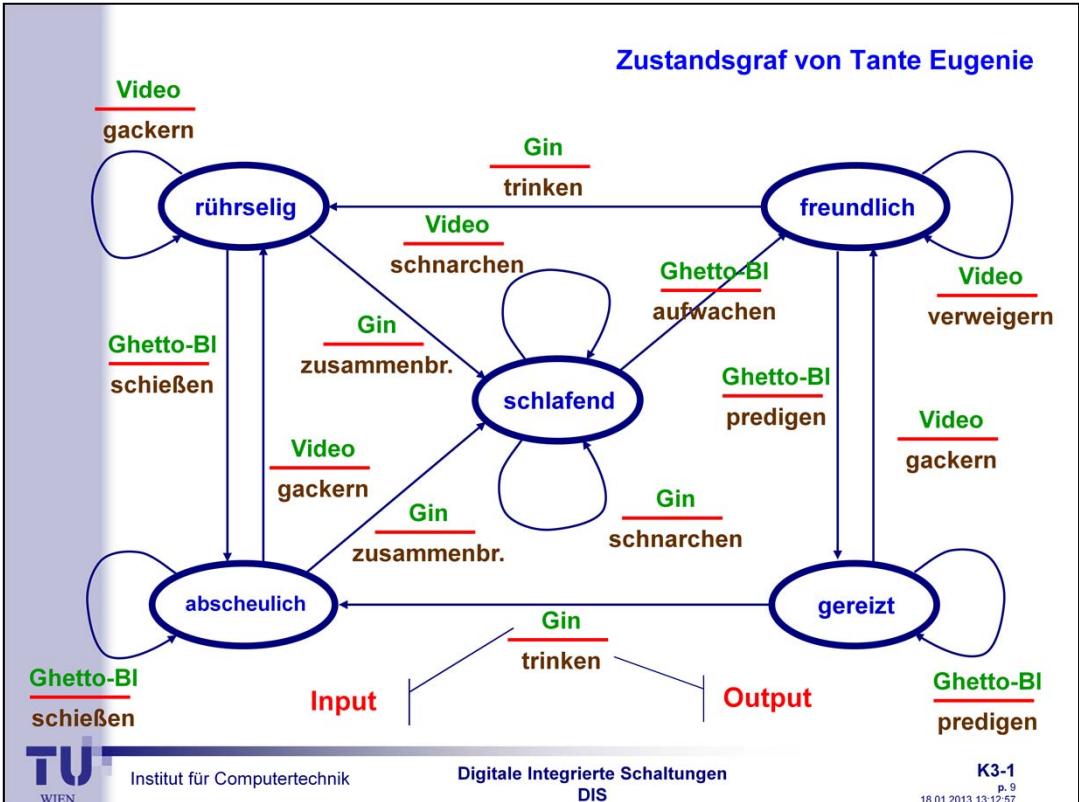
**Output:** schnarchen, schnarchen, aufwachen, verweigern, predigen, trinken, gackern, schießen, zusammenbrechen, schließen, gackern, zusammenbrechen, predigen, gackern, trinken

Reize	Stimmung jetzt	Stimmung nachher	Reaktionen
Video	schlafend	schlafend	schnarchen
Gin	schlafend	schlafend	schnarchen
Ghetto-Bl	schlafend	freundlich	aufwachen
Video	freundlich	freundlich	verweigern
Ghetto-Bl	freundlich	gereizt	predigen
Gin	freundlich	rührselig	trinken
Video	rührselig	rührselig	gackern
Ghetto-Bl	rührselig	abscheulich	schießen
Gin	rührselig	schlafend	zusammenbrechen
Ghetto-Bl	abscheulich	abscheulich	schießen
Video	abscheulich	rührselig	gackern
Gin	abscheulich	schlafend	zusammenbrechen
Ghetto-Bl	gereizt	gereizt	predigen
Video	gereizt	freundlich	gackern
Gin	gereizt	abscheulich	trinken

K3-1  
18.01.2013 13:12:57

Dies ist die vollständige Umsetzung auf eine Übergangstabelle (bei Schaltnetzen spricht man von Wertetabellen).

## Zustandsgraf von Tante Eugenie



Für jede Stimmung existiert ein Kreis, der nach der Stimmung (= Zustand) benannt wird. Zustand Z und Zustand Z' können zwei beliebige Stimmungen darstellen und eine Eingabe e führt von Z nach Z' und erzeugt die Ausgabe a, was durch einen Pfeil von Z nach Z' festgelegt wird, der mit e/a bezeichnet wird.

Wenn wir also die Folgen bestimmen wollen, die ihre Stimmung ändern, müssen wir zusammenhängende Pfeilfolgen suchen, die uns von der ersten zur zweiten Stimmung bringen, wenn wir den Pfeilen folgen. Wenn Sie das Diagramm betrachten, erkennen Sie, dass eine solche markiert ist, und von "schlafend" zu "freundlich" über einen Pfeil, der mit Ghetto-B/ Folge über einen Pfeil von "abscheulich" zu "schlafend" führt, der mit Gin/zusammenbrechen aufwachen markiert ist.

Shields zeigt auf amüsante Weise, wie scheinbar komplizierte dynamische Zusammenhänge auf recht einfache formale Beschreibungen zu reduzieren sind. Genau dies ist entscheidend in der Technik der Schaltnetze und Schaltwerke. Hält man sich an ein derartiges Schema erarbeitet, ist es dann relativ leicht möglich, Maschinenabläufe und genauso Entwicklungsabläufe zu automatisieren. Es versteht sich von selbst, dass der Art der Beschreibung von Shields eine präzise Definition folgen muss, die für die Praxis zu verfeinern ist. Es soll die Fähigkeit der Modellierung hinsichtlich der Schaltungsentwicklung auf der Basis der Automatentheorie so weit vermittelt werden, dass beliebige synchrone und in eingeschränktem Maße auch asynchrone Schaltwerke entworfen werden können.

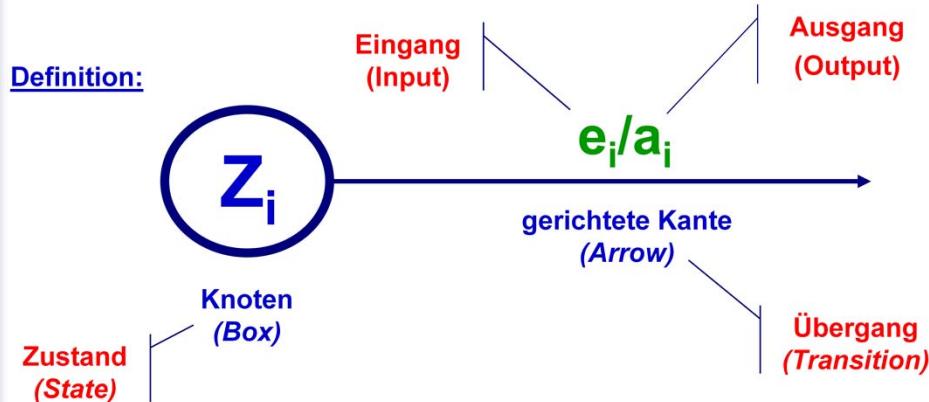
Das Beispiel von Shields vermittelt aber auch, dass der Schritt von der Verbalisierung hin zur Formalisierung ein entscheidender ist. Hier ist der Ingenieur gefordert. Werden hier Fehler versursacht, lässt sich das später nur unter hohem Kostenaufwand reparieren. Deshalb ist es sinnvoll, Verträge zwischen dem Kunden und dem Schaltkreisentwickler prinzipiell nicht auf verbaler Basis zu stellen, sondern prinzipiell auf die formalen Resultate – nur das akzeptiert aus Gründen des Verständnisses oft der Kunde nicht. Zu Ihrer Information: In der Prüfung wird für Sie die Umsetzung vom Verbalen zum Formalen der härteste sein. Um sein Wissen zu prüfen, ist es deshalb wichtig, neue Aufgabe, die man noch nicht kennt, alleine versuchen zu lösen.

Das Beispiel von Shields darf aber nicht dazu verleiten, dass man Automatenmodelle dazuverwenden könnte, das Verhalten des menschlichen Gehirns zu beschreiben. Hierzu sind Modelle von komplexen Systemen notwendig, die sich maßgeblich von der Automatentheorie unterscheiden. Ein wesentliches charakteristisches Unterscheidungsmerkmal komplexer Modelle kann zum Beispiel sein, dass es nicht nur über ein Modell überhaupt beschreibbar ist, sondern vielleicht mehrere Modelle notwendig werden, die sich gar nicht ergänzen müssen, sich vielleicht sogar in Teilspektre widersprechen können. Doch das ist nicht das Thema dieser Vorlesung.

# Digitale Integrierte Schaltungen (DIS)

## Kapitel 3-1 Schaltwerke

# Zustandsgraf



Der Zustandsgraf (State Transition Diagram) synchroner Schaltwerke beinhaltet drei Informationen darüber, in welchem Zustand  $Z_i$  sich das System befindet, über welchen Anreiz  $e_i$  (Eingangsgröße) dieser Zustand verlassen wird und welchen Wert zu dieser Zeit die Ausgangsgröße  $a_i$  einnimmt. Die Darstellung erfolgt in Bubble-Diagramm-Art. Der Zustand (State) wird im Zustandsgrafen in einen Knoten (Box) eingetragen, die im gleichen Zeitraum anstehenden Ein- und Ausgangsgrößen werden an die Kanten (Arrows) geschrieben. Irrelevante Ein- oder Ausgangsgrößen werden einfach weggelassen oder durch ein x ersetzt. Der Slash zwischen  $e_i$  und  $a_i$  sollte stets angeführt werden. Mit dem Clock wechselt dann das System vom Zustand  $Z_i$  in den Folgezustand  $Z_{i+1}$ , was bedeutet, dass synchrone Schaltwerk ist durch die Clock-Vorgabe als stabil anzusehen. Anders dagegen beim asynchronen Schaltwerk: Bei ihm kann jeder Anreiz von außen zu einer Kettenreaktion führen, in deren Verlauf verschiedene Zustände hintereinander eingenommen werden. Man versteht, dass bei der Beschreibung asynchroner Schaltwerke der Begriff Stabilität eine entscheidende Rolle spielen muss, was in einem Zustandsdiagramm auch zum Ausdruck kommt, wie noch gezeigt werden wird.

# Synchronität



## Asynchrone Schaltwerke:

Aufgrund der Aktion von  $e_i$  wird der Zustand  $Z_i$  verlassen bei gleichzeitiger Ausgabe von  $a_i$  und der Zustand  $Z_{i+1}$  wird eingenommen.

## Synchrone Schaltwerke:

Der Taktimpuls  $T_i$  verursacht einen Wechsel von  $Z_i$  nach  $Z_{i+1}$ , wobei  $e_i$  stabil anliegen muss.  $a_i$  ist eine momentane Ausgangsgröße von  $e_i$  und  $Z_i$ .

Zustände:  $Z_i$  mit  $i = 0, 1, 2, \dots$

Kanten:  $e_i$  mit  $i = 0, 1, 2, \dots$

$a_i$  mit  $i = 0, 1, 2, \dots$



Bei Synchronität liegen äquidistante Zeitabschnitte vor, bei Asynchronität kann man keine Aussage über die Dauer der einzelnen Zeitabschnitte treffen.

Die formale Beschreibung von **Schaltnetzen** ist eine starke Abstrahierung physikalischer Gegebenheiten, da die Schaltung zeitunabhängig betrachtet wird. Berücksichtigt man die Zeit, wird von **Schaltwerken** gesprochen, was heißt, dass speichernde Elemente in die Beschreibung mit einfließen. Die speichernde Einheit elementarer Art verzögert einen Impuls beim Durchlaufen einer Strecke. Um nun Informationen über eine beliebige Zeit einzufrieren, reicht dieser Effekt nicht aus. Man legt zwei phasendrehende Transistorstufen zugrunde, deren Ausgänge gegenläufig rückgekoppelt werden, was zum RS-Flip-Flop (RS-FF) führt, der elementaren digitalen Schaltwerkseinheit, auf die alle anderen FF-Arten aufbauen.

Zunächst muss angenommen werden, dass die speichernden Elemente unterschiedliche Zeitverzögerungen hervorrufen. Außerdem muss vorausgesetzt werden, dass Reize von außen zu beliebigen Zeiten eintreffen und entsprechende Aktionen auslösen. Die einzelnen Aktionen im Schaltwerk haben also keine gemeinsamen zeitlichen Aufsetzpunkte, weshalb von

*asynchronen Schaltwerken*

gesprochen wird.

Geht man davon aus, dass die speichernden Einheiten durch einen gemeinsamen Clock so getriggert werden, dass alle zum gleichen Zeitpunkt ihren Zustand wechseln, spricht man von

*synchronen Schaltwerken*.

Da die so definierte Triggerung physikalisch nur angenähert verwirklicht werden kann, wird deutlich, dass synchrone Schaltwerke eine stark abstrahierende Beschreibungsform darstellen, die jedoch der Realität unter bestimmten Bedingungen sehr nahe kommt.

# Definitionen

Betrachtet werden stets 2 unterschiedliche Zustände:

momentaner Zeitpunkt:  
zukünftiger Zeitpunkt:

$t^n$

$t^{n+1}$

Eingangsgröße:

$e_i(t^n)$

Ausgangsgröße:

$a_i(t^n)$

momentane Zustandsgröße:

$Q_i(t^n)$

zukünftige Zustandsgröße:

$Q_i(t^{n+1})$

Achtung:

Zustandsgröße:  $Q_i(t^n)$   
(Codierung)



Zustand:  $Z_i(t^n)$   
(Zustandsname)

*n, n+1 kann entfallen, wenn Eindeutigkeit gegeben ist.*

Die Zeit wird hochgestellt, da in der Schaltungstechnik die Indizes im Allgemeinen für die Nummerierung von Zuständen oder Leitungen vorgesehen werden. Hier ist eine Inkonsistenz zu erkennen. Die Zeiten n und n+1 werden hochgestellt, die Nummerierung der Zustände, die sich ja auch auf Zeiten beziehen, werden aber wieder als Indizes geschrieben.

# Übergangstabelle

$e_{p-1}^n \dots e_1^n e_0^n$	$Q_{r-1}^n \dots Q_1^n Q_0^n$	$Q_{r-1}^{n+1} \dots Q_1^{n+1} Q_0^{n+1}$	$a_{q-1}^n \dots a_1^n a_0^n$
0 .. 0 0			
0 .. 0 1			
0 ..			

**momentane Eingangsgröße      momentane Zustandsgröße      zukünftige Zustandsgröße      momentane Ausgangsgröße**



Für die Entwicklung und die Beschreibung von Schaltwerken sind folgende Hilfsmittel gebräuchlich:

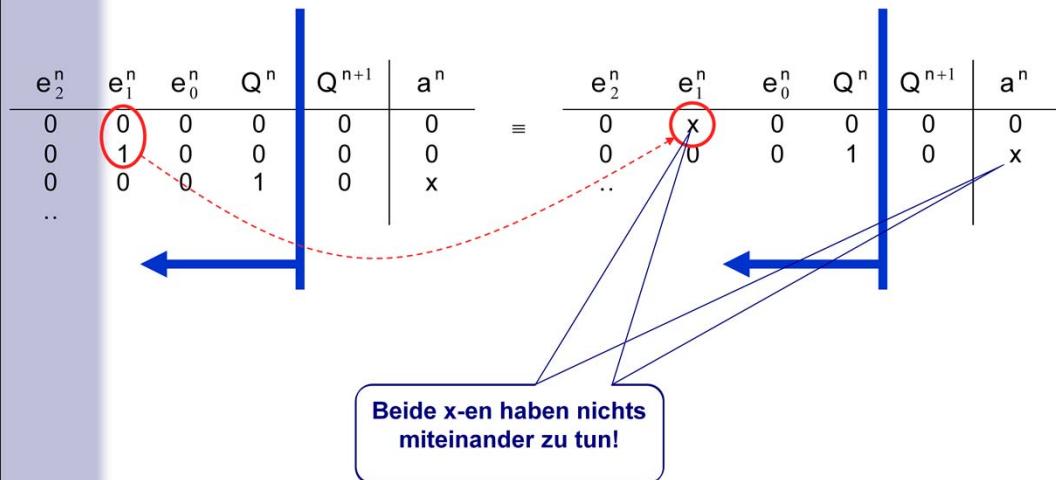
- .. Zustandsgrafen,
- .. Übergangstabellen und
- .. Boolescher Formalismus,

wobei einschränkend hinzugefügt werden muss, dass diese Hilfsmittel nur dazu ausreichen, die logischen Zusammenhänge und Übergänge zu beschreiben (weitere Erläuterungen hierzu folgen noch). Die Beschreibungsform asynchroner und synchroner Schaltwerke kann stark differieren, da beim synchronen System wesentliche Vereinfachungen vorgenommen werden können. Um die Erläuterungen so einfach wie möglich zu gestalten, soll zunächst im wesentlichen nur von synchronen Schaltwerken ausgegangen werden, auf die asynchronen Schaltwerke wird später eingegangen.

Die HDL (Hardware Description Language) sehe ich als hierarchisch übergeordnet an. In HDL-Tools werden Zustandsgrafen, Übergangstabelle und Boolescher Formalismus angewendet; sie stellen somit eine Untermenge der HDL dar.

Ein Zustandsgraf kann direkt in eine Übergangstabelle (State Transition Table) überführt werden, wie sie oben beispielhaft dargestellt wird.

# Übergangstabelle



Sind in einer Tabelle Zeilen bis auf  $e^n$  und  $Q^n$  identisch (also Zeilen der Tabelle links von der blauen senkrechten Linie), kann für die entsprechenden Werte ein x gesetzt werden. Dieses x (don't care) hat in diesem Fall keinerlei schaltungstechnische Relevanz und ist nur eine vereinfachte Darstellungsweise, da man pro x eine Zeile in der Tabelle einspart.

Das x in der rechten Spalte (Ausgang  $a^n$ ) stellt einen DC-Wert (Don't-Care-Wert) dar, was bedeutet, dass es für die Schaltung uninteressant ist, welchen Wert diese Leitung einnehmen wird. Wichtig ist dabei zu realisieren, dass die Leitung selbstverständlich einen Wert einnehmen wird.

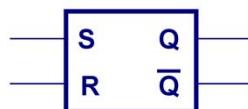
# Aufgabe:

Entwickeln Sie die synchrone Schaltung für die Tabelle.

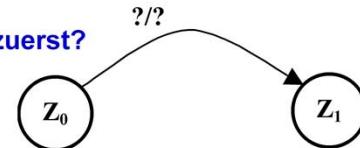
$e^n$	$Z_i^n$	$Q^n$	$Z_i^{n+1}$	$Q^{n+1}$	$a^n$
0	$Z_0$	0	$Z_0$	0	x
1	$Z_0$	0	$Z_1$	1	0
0	$Z_1$	1	$Z_0$	0	0
1	$Z_1$	1	$Z_1$	1	1

x: Don't care!

Basis:

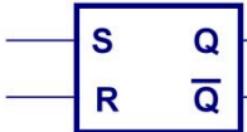


Was zeichnet man zuerst?

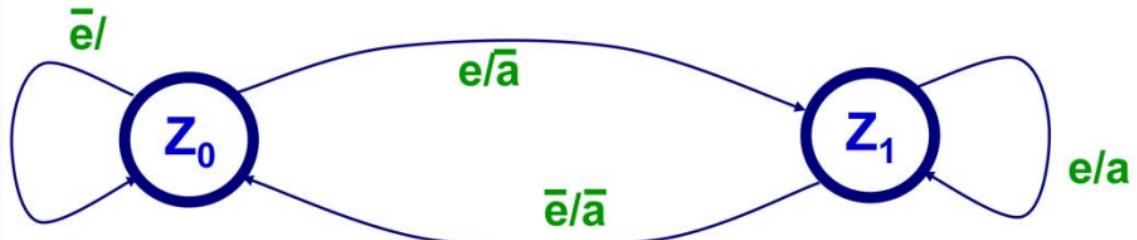


Es ist beispielhaft das Zustandsdiagramm eines einfachen synchronen Schaltwerkes gezeigt. Zwei Zustände können eingenommen werden, wobei im Zustand  $Z_0$  und  $Z_1$  dem gleichzeitig vorliegenden Wert der Eingangsgröße  $e = 0$  die Ausgangsgröße  $a = DC$  (don't care) ist, was bei der Vereinfachung eine wesentliche Rolle spielt, da für diesen Fall  $a = 0$  und  $a = 1$  angenommen werden darf.

$e^n$	$Z_i^n$	$Q^n$	$Z_i^{n+1}$	$Q^{n+1}$	$a^n$
0	$Z_0$	0	$Z_0$	0	x
1	$Z_0$	0	$Z_1$	1	0
0	$Z_1$	1	$Z_0$	0	0
1	$Z_1$	1	$Z_1$	1	1



x: Don't care!



# Speicherelemente

RS-FF

= **asynchrones "Ur-Schaltwerk** =  
"Ur-zelle = kleinste Einheit

asynchrone Schaltwerke

synchrone Schaltwerke

## Beschreibungsmöglichkeiten:

1. flache Schaltung
2. Übergangstabelle
3. Zustandsgraf
4. Charakteristische Gleichung
5. Formale Beschreibung (VHDL, ..)

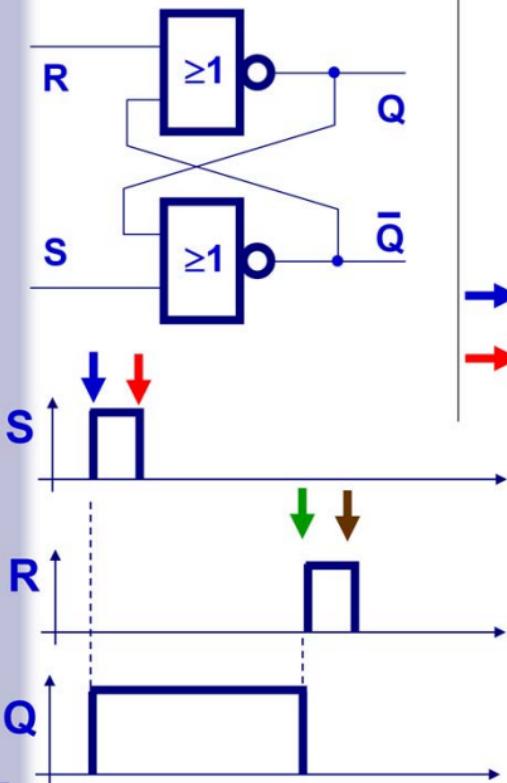
einfache FFs

MS-FFs

Das RS-FF bildet die Urzelle aller anderen FFs.

MS-FF: immer weniger gebräuchlich. In ASICs werden zunehmend einfache FFs integriert.

# RS-FF

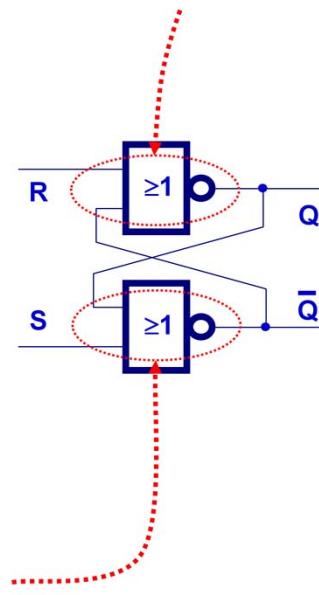


$S^n$	$R^n$	$Z_i^n$	$Q^n$	$Z_i^{n+1}$	$Q^{n+1}$
0	0	$Z_0$	0	$Z_0$	0
0	0	$Z_1$	1	$Z_1$	1
0	1	$Z_0$	0	$Z_0$	0
0	1	$Z_1$	1	$Z_0$	0
1	0	$Z_0$	0	$Z_1$	1
1	0	$Z_1$	1	$Z_1$	1

# RS-FF

$$\overline{R^n \vee \overline{Q}^n} \rightarrow "Q^{n+1}" \rightarrow Q^n$$

S <sup>n</sup>	R <sup>n</sup>	Z <sub>i</sub> <sup>n</sup>	Q <sup>n</sup>	Z <sub>i</sub> <sup>n+1</sup>	Q <sup>n+1</sup>
0	0	Z <sub>0</sub>	0	Z <sub>0</sub>	0
0	0	Z <sub>1</sub>	1	Z <sub>1</sub>	1
0	1	Z <sub>0</sub>	0	Z <sub>0</sub>	0
0	1	Z <sub>1</sub>	1	Z <sub>0</sub>	0
1	0	Z <sub>0</sub>	0	Z <sub>1</sub>	1
1	0	Z <sub>1</sub>	1	Z <sub>1</sub>	1
1	1	-	0	-	-
1	1	-	1	-	-



$$\overline{S^n \vee \overline{Q}^n} \rightarrow \overline{"Q^{n+1}"} \rightarrow \overline{\overline{Q}^n}$$

Die Funktion, wie sie durch die Festlegung der Ausgänge oben gefordert wird, wird gewährleistet, solange

$$R^n \text{ UND } S^n = 1$$

nicht zugelassen wird, was zur obigen Tabelle führt. Die Herleitung der charakteristischen Gleichung kann über die Schaltung selbst erfolgen oder aus der Übergangstabelle ermittelt werden.

Frage bzw. Zusatzbemerkung:

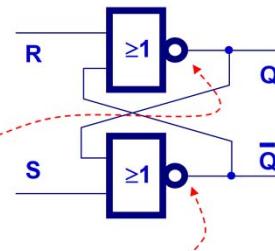
(1) Warum werden in den beiden unteren Zeilen der Tabelle zwar die Reihen S<sup>n</sup>, R<sup>n</sup> und Q<sup>n</sup> mit Werten von 0 und 1 beschrieben, die drei anderen Spalten nicht ausgefüllt?

Diese 3 Werte können in dieser Kombination theoretisch vorkommen, doch da alles andere physikalisch nicht eindeutig beschreibbar ist, macht es keinen Sinn, ein Z, gleichgültig welcher Art zu definieren. Welch nächster Wert eingenommen wird, weiß man so oder so nicht.

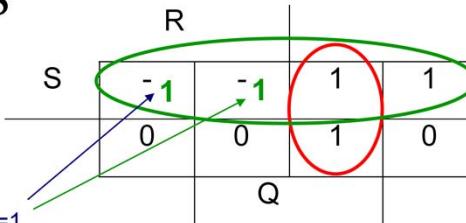
(2) Ein NAND wie in der oberen Darstellung wurde bisher als "zeitloser" Baustein definiert. Hier jedoch spielt die Flankendurchlaufzeit über die interne Transistoren die entscheidende Rolle. Vor dem Transistor ist der Zustand X<sub>n</sub> zu sehen, der über die Kombination der beiden Eingänge zu Q<sub>n+1</sub> führen wird (also nach dem die Flanke den entsprechenden Transistor passiert hat. Da durch den Transistor aber außer der VerUNDung noch die Negation erfolgt, geschehen mehrere Dinge gleichzeitig, die jedoch einzeln im Modell betrachtet werden müssen, um sie mathematisch erfassen zu können: erstens die VerUNDung, zweitens die Negation, drittens die zeitliche Verzögerung (aus S<sup>n</sup> und R<sup>n</sup> wird mit der Negation zusammen ein neg(Q<sup>n+1</sup>), was sich noch nicht eingestellt hat) und viertens ein neg(Q<sup>n</sup>), wenn die Flanke den Transistor passiert hat).

## Herleitung der charakteristischen Gleichung

$$\begin{aligned}
 Q^{n+1} &= \overline{\overline{R^n} \vee \overline{Q^n}} \\
 &= \overline{R^n} \vee (\overline{Q^n} \vee S^n) \\
 &= \overline{R^n}(Q^n \vee S^n) \\
 &= \overline{R^n}Q^n \vee \overline{R^n}S^n
 \end{aligned}$$



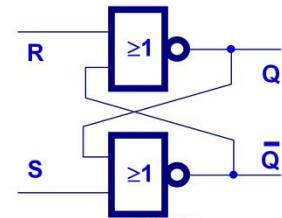
$$Q^{n+1} = \overline{R^n}Q^n \vee S^n$$



mit der Vereinfachung gilt: DC-Felder  $\Rightarrow =1$   
d. h.:  $S = R = 1$ ; dann wäre aber  $Q = \neg Q = 0$   
= Widerspruch: deshalb:  $S = R = 1$ : verbotener Zustand

Auch der Boolesche Formalismus beinhaltet nicht mehr oder weniger Informationen als der Zustandsgraf und die Übergangstabelle. Bei der Beschreibung von FFs führt dies beispielsweise zu den charakteristischen Gleichungen.

	R	-	-	1	1
S	-	-	1	1	
	0	0	1	0	Q



mit DC:

$$Q^{n+1} = \bar{R}^n Q^n \vee S^n$$

ohne DC:

$$Q^{n+1} = \bar{R}^n Q^n \vee \bar{R}^n S^n$$

$S^n$	$R^n$	$S^n \wedge \bar{R}^n$
0	0	0
0	1	0
1	0	1
-	-	-

$$\rightarrow S^n \wedge \bar{R}^n \Rightarrow S^n$$

solange  $S^n = R^n = 1$   
verboten ist!

Das RS-FF wird somit vollständig durch das charakteristische Gleichungssystem beschrieben. Zum gleichen Ergebnis gelangt man, wenn man für die Tabelle das KV-Diagramm aufstellt und so zusammenfasst, dass die Don't-Care-Terme 1 werden. In Datenbüchern findet man im allgemeinen anstatt der vollständigen die vereinfachte Schreibweise von oben.

## Beschreibung des RS-FF

$S^n$	$R^n$	$Z_i^n$	$Q^n$	$Z_i^{n+1}$	$Q^{n+1}$
0	0	$Z_0$	0	$Z_0$	0
0	0	$Z_1$	1	$Z_1$	1
0	1	$Z_0$	0	$Z_0$	0
0	1	$Z_1$	1	$Z_0$	0
1	0	$Z_0$	0	$Z_1$	1
1	0	$Z_1$	1	$Z_1$	1
1	1	-	0	-	-
1	1	-	1	-	-

= **asynchrones Verhalten!**

## reduzierte Gleichungen:

$S^n$	$R^n$	$Q^{n+1}$	$Z_i^{n+1}$
0	0	$Q^n$	$Z_i^n$
0	1	0	$Z_0$
1	0	1	$Z_1$
1	1	-	-

## Charakteristischen Gleichungen:

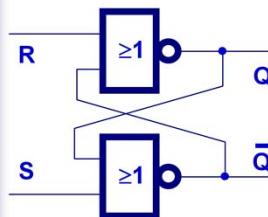
$$Q^{n+1} = S^n \vee \bar{R}^n Q^n$$

mit der Nebenbedingung

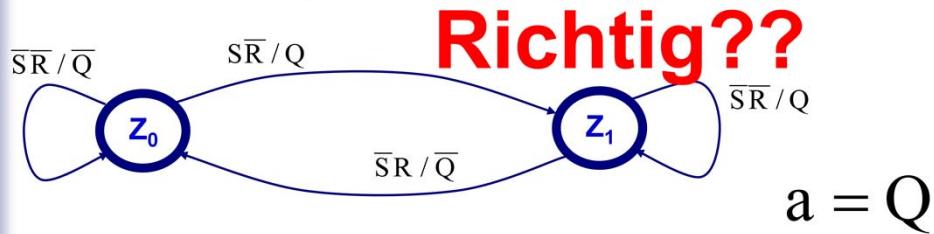
$$S^n \wedge R^n = 0$$

Bezüglich der Herleitung der charakteristischen Gleichungen (bzw. Gleichungssysteme) sei auf die ("klassische") Literatur verwiesen. Im folgenden sind ausschließlich die Gleichungen der häufigsten FF-Typen wiedergegeben, wobei ich darauf hinweisen muss, dass in der Literatur (vor allem in Datenbüchern) nicht nur verschiedene Namen für die gleichen FF-Typen Verwendung finden, sondern auch umgekehrt unter dem gleichen FF-Namen unterschiedliche FF-Typen verstanden werden. Es ist deshalb anzuraten, im konkreten Beispiel stets die charakteristische Gleichung und - wenn mehr als das logische Verhalten beschrieben werden soll - auch das Symbol mitanzugeben.

# RS-FF



$S^n$	$R^n$	$Q^{n+1}$	$Z_i^{n+1}$	
0	0	$Q^n$	$Z_i^n$	speichern
0	1	0	$Z_0$	zurücksetzen
1	0	1	$Z_1$	setzen
1	1	-	-	



## Charakteristische Gleichungen ..

- .. definieren den Typ von FF,
- .. stellen die Basis für die Berechnung von Schaltwerken dar,
- .. spiegeln ausschließlich das logische Verhalten wider.

Wichtige charakteristische Gleichungen von FFs sind:

1. Asynchrones RS-FF:

$$Q^{n+1} = S^n \vee \bar{R}^n Q^n$$

NB :  $S^n \wedge R^n = 0$

2. Taktzustandsgetriggertes RS-FF

$$Q^{n+1} = S^n \vee \bar{R}^n Q^n$$

(transparentes RS-FF)

NB :  $S^n \wedge R^n = 0$

## Charakteristischen Gleichungen (ohne weitere Herleitung):

3. Taktzustandsgetriggertes D-Latch  
(Auffang-FF)

$$Q^{n+1} = D^n$$

4. RS-Master-Slave-FF

$$Q^{n+1} = S^n \vee \bar{R}^n Q^n$$

$$\text{NB: } S^n \wedge R^n = 0$$

5. Positiv einflankengetriggertes D-FF  
(Delay FF)

$$Q^{n+1} = D^n$$

6. JK-Master-Slave-FF

$$Q^{n+1} = (J^n \wedge \bar{Q}^n) \vee (\bar{K}^n \wedge Q^n)$$

## Was enthalten Charakteristische Gleichungen nicht?

- Übergangsverhalten  
(Timing)
  - Wann übernimmt das FF die Eingangsinformation?
  - Wann stellt es sie am Ausgang zur Verfügung?
  - Wann ist es störunempfindlich?
  - ..
- Ist das FF positiv-, negativ flankengetriggert?
- Fan-in, Fan-out?
- Technologie?
- Schwellwert?
- Hysteresis?
- ..

# Automat

*Def. (abgeleitet von Tante Eugenie)*

- (I) *Stimmung*       $\Rightarrow$       *Zustände*       $Q_i$
- (II) *Reize*           $\Rightarrow$       *Eingangsgrößen*       $e_i$
- (III) *Reaktionen*     $\Rightarrow$       *Ausgangsgrößen*       $a_i$
- (IV) *Regel 1*         $\Rightarrow$        $Q_i^n \xrightarrow{e_i^n Q_i^n} Q_i^{n+1}$
- (V) *Regel 2*         $\Rightarrow$        $a_i = f(e_i, Q_i)$

**Maschine: Liste (Q, e, a, α, β)     $\rightarrow$     Quintupel**

## **Nomenklatur:**

$$Q_i^n \xrightarrow{e_i^n Q_i^n} Q_i^{n+1}$$

$Q$  : Vektor

$\overline{Q}_i^n$  : negierte Variable

# Automat

Maschine: Liste  $(\underline{Q}, \underline{e}, \underline{a}, \underline{\alpha}, \underline{\beta})$   Quintupel

$\underline{Q}$  := Vorrat von Zuständen

$\underline{e}$  := Vorrat von Eingangsgrößen

$\underline{a}$  := Vorrat von Ausgangsgrößen

$\underline{\alpha}$  := Regel 1

$\underline{\beta}$  := Regel 2

$\underline{Q}, \underline{e}, \underline{a}$  = finite, nicht leere Mengen

$\underline{\alpha}, \underline{\beta}$  = Funktionen, wobei gilt:

$$Q, e, a, \in (\underline{Q}, \underline{e}, \underline{a})$$

Es wird festgelegt:  $Q_i^n \xrightarrow{e_i^n Q_i^n} Q_i^{n+1}$

$\alpha: e \times Q := Q >$  Zustandsübergangsfunktion

$\beta: e \times Q := a >$  Ausgangsfunktion

mit:  $x:$  kartesisches Produkt

$Q = \{\text{schlafend, freundlich, gereizt, rührselig, abscheulich}\}$

$e = \{\text{Ghetto-Blaster, Video, Gin}\}$

$a = \{\text{aufwachen, schnarchen, predigen, verweigern, trinken, gackern, schießen, zusammenbrechen}\}$

$\alpha(\text{Gin, freundlich}) = \alpha(e, Q) = Q$

$\alpha(..) = ..$

:

$\beta(\text{Gin, schlafend}) = \beta(e, Q) = a$

$\beta(..) = ..$

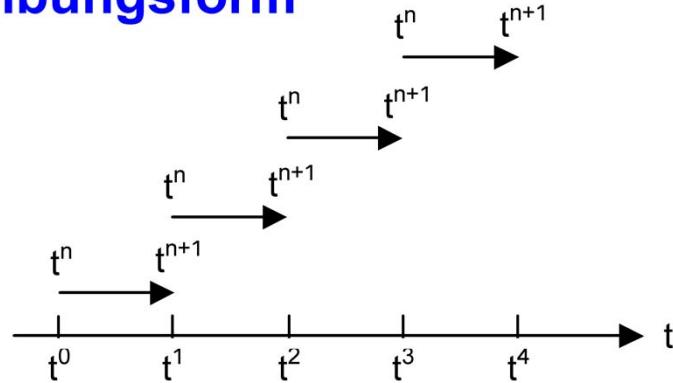
:



Mit diesem Formalismus lässt sich das Beispiel "Tante Eugenie" wie oben beschreiben.

Das Beispiel weist auf den wesentlichen Inhalt der Automatentheorie hin, wie er in der Schaltwerksentwicklung angewendet wird: die *Rekursionsmethode*, die sich in einem Rückkopplungs-Schaltungsmechanismus (*Feedback Logic*) widerspiegelt.

# Beschreibungsform



$$a_i^n = f(e_0^n, e_0^{n-1}, \dots, e_0^0, e_1^n, e_1^{n-1}, \dots, e_2^n, \dots),$$

eine Möglichkeit

Diese Beschreibungsform hat den Nachteil, dass *alle* Vergangenheitswerte gespeichert werden müssen, was nicht zu realisieren wäre. Man verwendet deshalb eine rekursive Methode, die im Folgenden erläutert wird. Die rekursive Methode erlaubt es, nur zwei Zeitpunkte zugrunde zu legen, den momentanen Zeitpunkt  $t(n)$  (present State) und den zukünftigen Zeitpunkt  $t(n+1)$  (next State), der sich einstellen wird – siehe nächste Folie.

*Bemerkung:* Eine theoretisch mögliche Variante wäre, dass zur Zeit  $t=0$  Register eindeutig (unabhängig von ihrem vorherigen Zustand) in einen Zustand versetzt werden, der allein über die Größen  $e_0$  festgelegt wird. Das müsste dann auch mit allen folgenden Größen  $e_1$  und  $e_2$  und .. passieren. Dann hätte man diese Größen jederzeit für das folgende Schaltnetz zur Verfügung, nur das Register würde mit der Zeit stetig zunehmen. Eine Realisierung wäre also Blödsinn.

*"praktische Vorgehensweise:"*

## Beschreibungsmethode in der Schaltungsalgebra

*Mögliche allgemeine Beschreibung eines Schaltwerkes:*

$$a_i^n = f(e_0^n, e_0^{n-1}, \dots, e_0^0, e_1^n, e_1^{n-1}, \dots, e_2^n, \dots)$$

*d. h.: alle Vergangenheitswerte werden berücksichtigt.*

*Deshalb: rekursive Methode  
State Machine*

*dann sind von Interesse nur noch:*

*2 Zeitpunkte:*

$t^n$  und  $t^{n+1}$

momentaner Zeitpunkt  
present State

zukünftiger Zeitpunkt  
next State

## **Notwendigkeit für rekursive Methode**

Zwischenvariable:  $Q_i^n, Q_i^{n+1}$ ,

zur Beschreibung der inneren Zustände der Maschine

**Rekursionsformalismus:**

Übergangsfunktion:

$$Q_i^{n+1} = \alpha_i(e_{p-1}^n, e_{p-2}^n, \dots, e_0^n, Q_{r-1}^n, Q_{r-2}^n, \dots, Q_0^n)$$

= Rückkopplung!

**Feedback Logic**

## Rekursionsformalismus:

### Übergangsfunktion:

$$Q_i^{n+1} = \alpha_i(e_{p-1}^n, e_{p-2}^n, \dots, e_0^n, Q_{r-1}^n, Q_{r-2}^n, \dots, Q_0^n)$$

### Ausgangsfunktion:

$$a_i^n = \beta_i(e_{p-1}^n, e_{p-2}^n, \dots, e_0^n, Q_{r-1}^n, Q_{r-2}^n, \dots, Q_0^n)$$

*oder vektoriell geschrieben:*

$$\underline{Q}^{n+1} = \alpha(\underline{e}^n, \underline{Q}^n) : \text{Übergangsfunktion}$$

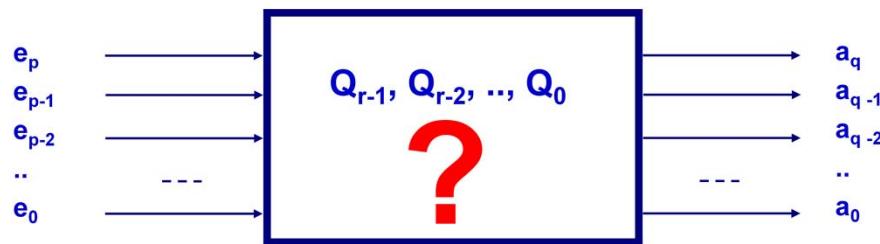
$$\underline{a}^n = \beta(\underline{e}^n, \underline{Q}^n) : \text{Ausgangsfunktion}$$

# Modelle

## Schaltnetz:



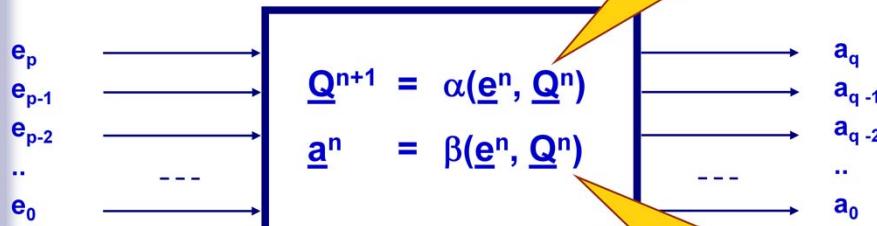
## Schaltwerk:



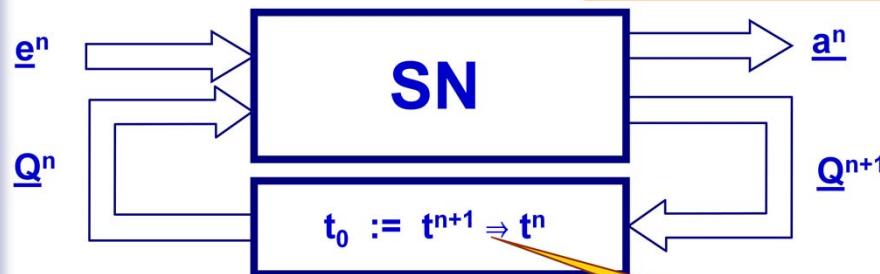
Der Block des Schaltnetzes kann mit Hilfe der Booleschen Algebra beschrieben werden. Wie kann aber im unteren Block die Zeit miteingebunden werden?

# Modell

Blackbox:



Rekursionsgleichung



Ausgangsgleichung

Verzögerung

K3-1

p. 36

18.01.2013 13:12:57



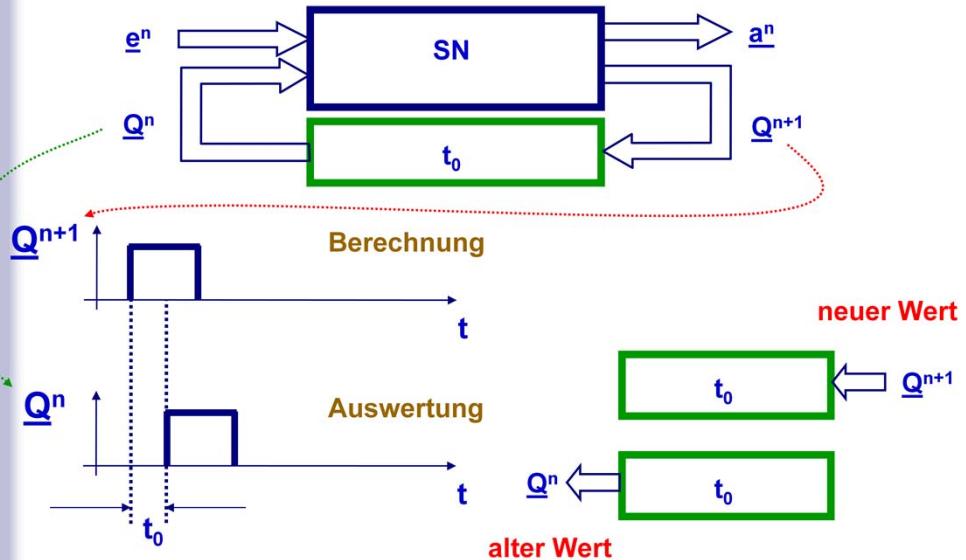
Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

Das Bild oben spiegelt die starke Abstrahierung des Schaltwerksmodells wider. Im Modul SN (Schaltnetz) wird die schaltalgebraische Logik beschrieben, und im Modul  $t_0$  sind die Verzögerungselemente enthalten. Das physikalische System "Schaltwerk" wird demgemäß in eine *zeitunabhängige* Einheit und in eine *zeitabhängige* Einheit abstrahiert und modularisiert. Die formale Beschreibung wird dadurch einfach: Im Schaltnetz können die kombinatorischen Gesetze angewendet werden, ohne auf die zeitlichen Verhältnisse Rücksicht nehmen zu müssen, während die Verzögerungs- bzw. die Speicherelemente im Modul  $t_0$  zusammengefasst werden.

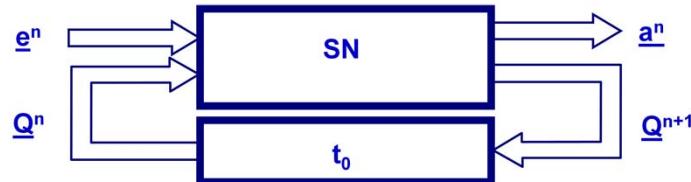
Das obere Modell führt zu 2 unterschiedlichen Ausgangsgrößen ( $Q^{n+1}$  und  $a^n$ ), was im unteren verdeutlicht wird.

## Verständnisschwierigkeiten bzgl.: $Q^{n+1}$



Erfahrungsgemäß führt die Definition von  $Q^{n+1}$  zu Verständnisschwierigkeiten, denn nach der Modellvorstellung steht  $Q^{n+1}$  verzögerungsfrei (Schaltnetz  $f(t)$ ) am Eingang des Moduls  $t_0$  zur Verfügung. Die Signalaktion wird daraufhin verzögert und ist nach der Zeit  $t_0$  als  $Q^n$  verfügbar. Die Betrachtung für die Begriffsdefinition  $Q^n$  und  $Q^{n+1}$  erfolgt somit jeweils zu einem Zeitpunkt nach einer erfolgten Aktion, was oben mit den Begriffen "neuer Wert" für  $Q^{n+1}$  und "alter Wert" für  $Q^n$  verdeutlicht werden soll.

## Verständnisschwierigkeiten bzgl.: $Q^{n+1}$



Nach diesem Modell bildet

$Q^{n+1}$  **verzögerungsfrei**

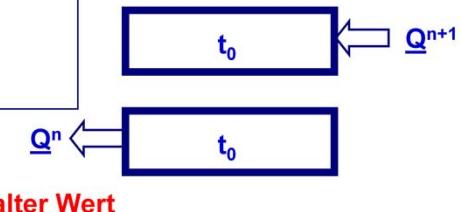
die Ausgangsgröße am oberen Block,

die Eingangsgröße am unteren Block

*und*

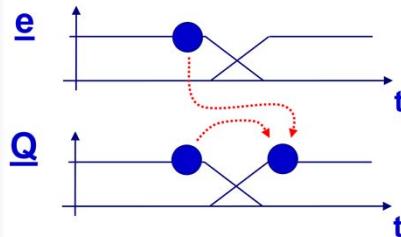
steht dann nach  $t_0$  am Ausgang des unteren Block zur Verfügung.

**neuer Wert**



## Zusammenhang zw. $Q^{n+1}$ und $Q^n$ :

<b>Der zum Zeitpunkt</b>	$t^{n+1}$
<b>auszuwertende Zustandsvektor</b>	$\underline{Q}^{n+1}$
<b>wird aus dem aktuellen Eingangsvektor</b>	$\underline{e}^n$
<b>und dem aktuellen Zustandsvektor</b>	$\underline{Q}^n$
<b>gewonnen.</b>	



In der so dargestellten Erläuterung wird nicht davon gesprochen, wie die Verzögerung erreicht wird. Sie kann ja asynchron erfolgen oder getriggert durch einen gemeinsamen Takt, also synchron. Doch die Interpretation des Modells fällt für rein synchrone Schaltwerke einfacher: Aus der vorhergehenden Darstellung und dem Bild oben ist zu lesen, dass der zum Zeitpunkt  $t^{n+1}$  auszuwertende Zustandsvektor  $\underline{Q}^{n+1}$  aus dem aktuellen Eingangsvektor  $\underline{e}^n$  und dem aktuellen Zustandsvektor  $\underline{Q}^n$  gewonnen wird. Nach einem Ereignis (Event), das bedeutet nach einem Taktwechsel im  $t^0$ -Modul, steht der Wert  $\underline{Q}^{n+1}$  physikalisch zur Verfügung, bis der nächste Taktwechsel erfolgt.

## Zusammenhang zw. $Q^{n+1}$ und $Q^n$ :

Eingangsvektor:	$\underline{e}^n$	=	$(e_{p-1}, e_{p-2}, \dots, e_0)^n$
Ausgangsvektor:	$\underline{a}^n$	=	$(a_{q-1}, a_{q-2}, \dots, a_0)^n$
Zustandsvektor:	$\underline{Q}^n$	=	$(Q_{r-1}, Q_{r-2}, \dots, Q_0)^n$

Übergangsfunktion:	$\underline{Q}^{n+1}$	=	$\alpha(\underline{e}^n, \underline{Q}^n)$	: Regel 1
Ausgangsfunktion:	$\underline{a}^n$	=	$\beta(\underline{e}^n, \underline{Q}^n)$	: Regel 2

## für synchrone & asynchrone Schaltwerke



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K3-1  
p. 40  
18.01.2013 13:12:57

Zusammenfassend soll festgehalten werden: Das Modell kann für asynchrone und synchrone Schaltwerke unterschiedlich interpretiert werden. Die allgemeine formale Beschreibung ist jedoch für beide identisch.

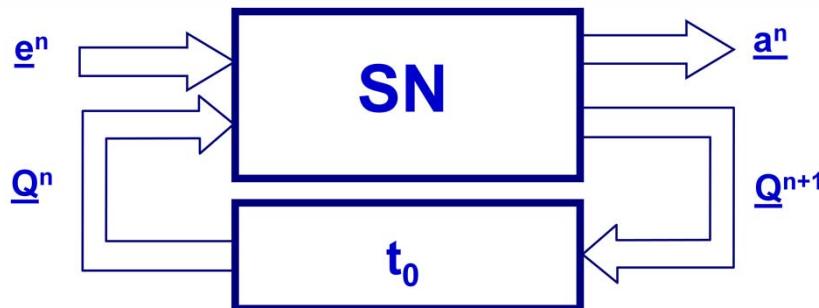
Das Gleichungssystem beschreibt den allgemeinsten Automatentypen, den sogenannten

### Mealy-Automaten

(auch unter dem Begriff 'Huffman Model of a sequential Circuit' bekannt).

# Allgemeinste Darstellung eines Automaten = Mealy-Automat

Eingangsvektor:	$\underline{e}^n = (e_{p-1}, e_{p-2}, \dots, e_0)^n$
Ausgangsvektor:	$\underline{a}^n = (a_{q-1}, a_{q-2}, \dots, a_0)^n$
Zustandsvektor:	$\underline{Q}^n = (Q_{r-1}, Q_{r-2}, \dots, Q_0)^n$
Übergangsfunktion:	$\underline{Q}^{n+1} = \alpha(\underline{e}^n, \underline{Q}^n) : \text{Regel 1}$
Ausgangsfunktion:	$\underline{a}^n = \beta(\underline{e}^n, \underline{Q}^n) : \text{Regel 2}$



Wird das Gleichungssystem grafisch dargestellt, erhält man das Bild oben. Was sagt es aus? Etwas Ungeheuerliches! Man trennt in der Tat damit gedanklich die Logik von der Physik und ist damit in der Lage, zwei Dinge, die aus einer Sichtweise schwer zu erfassen sind, unabhängig von einander zu betrachten und unabhängig von einander, einfach zu berechnen. Dies ist eine Methode, die man in der Technik oft anwendet. Ich komme auf das noch mal in einer anderen Vorlesung, der Fehlertoleranten Systeme (FeTo), zu sprechen.

## Automatentypen

Mealy:

$$Q^{n+1} = \alpha(e^n, Q^n)$$

$$a^n = \beta(e^n, Q^n)$$

Moore:

$$Q^{n+1} = \alpha(e^n, Q^n)$$

$$a^n = \beta(Q^n)$$

Speicherautomat:

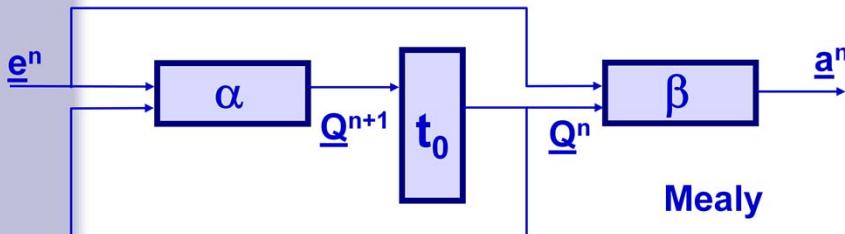
$$Q^{n+1} = \alpha(e^n, Q^n)$$

$$a^n = \beta(Q^{n+1})$$

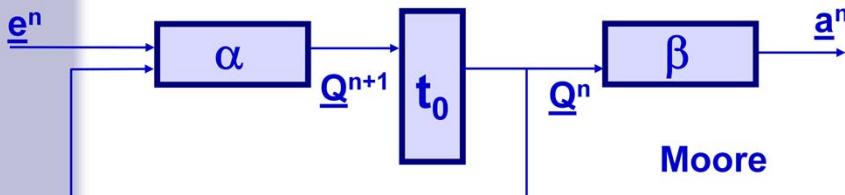
$$= \beta(\alpha(e^n, Q^n))$$

nicht  
realisierbar

## Automatentypen



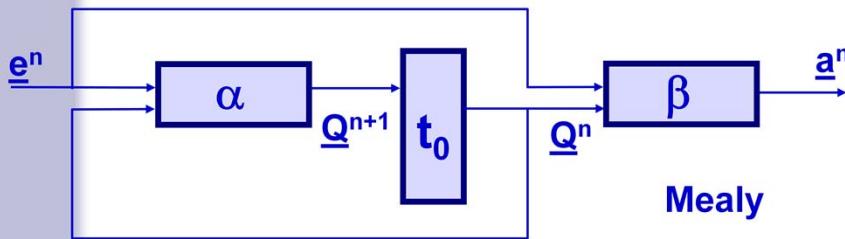
Mealy



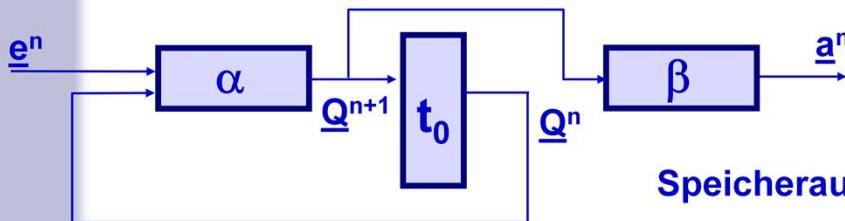
Moore

Dies ist eine differenziertere Darstellung (das Modul "Schaltnetz" ist aufgesplittet in die beiden Funktionseinheiten  $\alpha$  und  $\beta$ ). Eliminiert man die direkte Abhängigkeit des Ausgangsvektors an von den Eingangsvektoren, gelangt man zum Moore-Automaten.

## Automatentypen



Mealy



Speicherautomat

Vorsicht: Begriffswirrwarr:

z. B. Wendt:

Moore  $\Leftrightarrow$  Speicherautomat

Den Speicherautomaten erhält man durch die obige Festlegung.

Den Speicherautomaten findet man in der Praxis wenig. Am häufigsten sind der Mealy- und der Moore-Automat vertreten. Der Mealy-Automat deshalb, weil aufgrund seiner allgemeinen Formalisierung über ihn alle möglichen Schaltwerke zu realisieren sind. Er ist deshalb als "Ur-Automat" anzusehen. Wie noch gezeigt werden soll, sind Schaltungen mit ihm im allgemeinen einfach zu entwickeln, insbesondere wenn es sich um synchrone Schaltwerke handelt.

Es gibt noch weitere Typen, die aber hier und heute kaum noch eine Rolle spielen.

Bezüglich Hardware-Entwicklung nur wichtig:

## Moore & Mealy

### Mealy:

**Vorteil:** alle Schaltungen zu realisieren

**Nachteil:**  $a = f(e, ..)$ :  
direkte Störungsabhängigkeit

### Moore:

**Vorteil:**  $a \neq f(e, ..)$

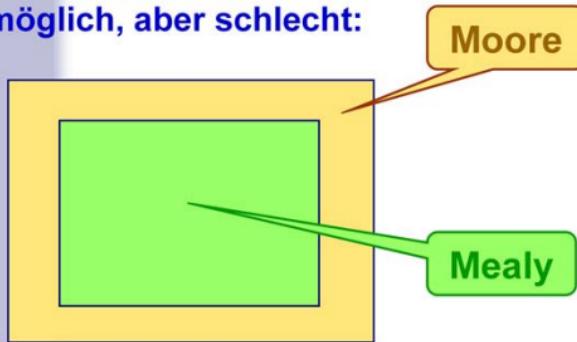
**Nachteil:** - Hardwareaufwand (i. a. höher)  
- Entwurf komplizierter

Der Nachteil des Mealy-Automaten ist jedoch offensichtlich. Durch direkte Abhängigkeit des Ausgangsvektors  $a^n$  von dem Eingangsvektor  $e^n$  pflanzen sich Störungen ungehindert vom Eingang zum Ausgang der Maschine fort, was für viele Schaltungen nicht akzeptiert werden kann. So sollten synchrone Eingangsschaltwerke in ASICs auf keinen Fall auf der Basis des Mealy-Automaten realisiert werden, sondern stets auf der Basis des Moore-Automaten, bei dem Störungen weitgehend durch die FFs abgefangen werden. Das Gleiche gilt für Schaltungen, die an Bussysteme angeschaltet werden, um nur zwei charakteristische Beispiele zu nennen. Geht man von der Menge aller möglichen Zustände einer Schaltung aus, ist klar, dass sie bezüglich einer Schaltwerksentwicklung über den Mealy- oder den Moore-Automaten gleich sein muss. Da aber die funktionale Beziehung beim Moore-Automaten wegfällt, ist selbstverständlich, dass die entsprechende Informationsmenge an anderer Stelle enthalten sein muss, wobei dies nur in inneren Zuständen erfolgen kann. Physikalisch bedeutet dies, dass die Anzahl der FFs in synchronen Schaltwerken im Moore-Automaten (im allgemeinen) höher ist als im vergleichbaren Mealy-Automaten. Auf die Thematik soll noch näher in folgenden Abschnitten eingegangen werden.

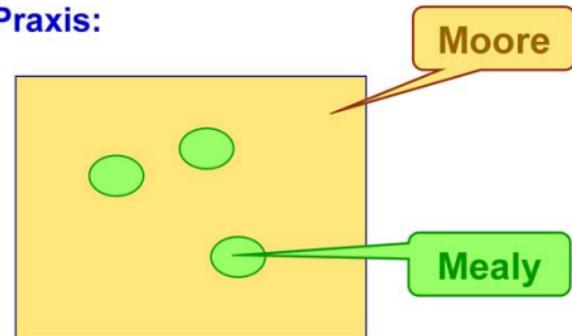
Es sind Schaltungskonfigurationen denkbar, bei denen dies nicht der Fall ist.

# ASICs

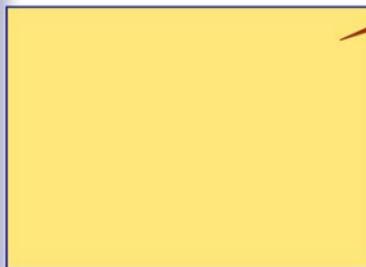
möglich, aber schlecht:



Praxis:



ideal:



**Prinzipiell:**

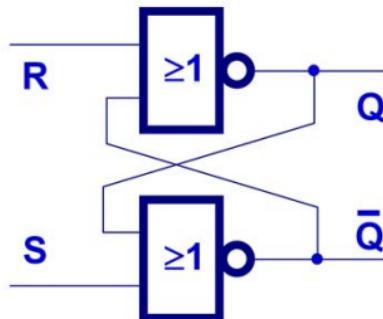
**Damit lassen sich  
ALLE  
digitalen Schaltungen  
realisieren!**

**Warum nicht dann den µC  
damit entwerfen ??**

# Klassifizierung

- synchrone  $\Leftrightarrow$  asynchrone Schaltwerke
- Realisierung mit
  - RS-FFs
  - D-FFs
  - JK-FFs
- (modifizierte Schaltwerke)

Urzelle der sequentiellen Schaltwerke:

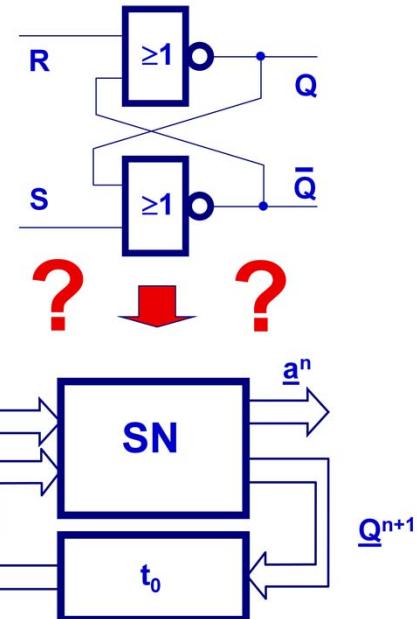


**RS-FF**

Als  
Mealy-Automat?

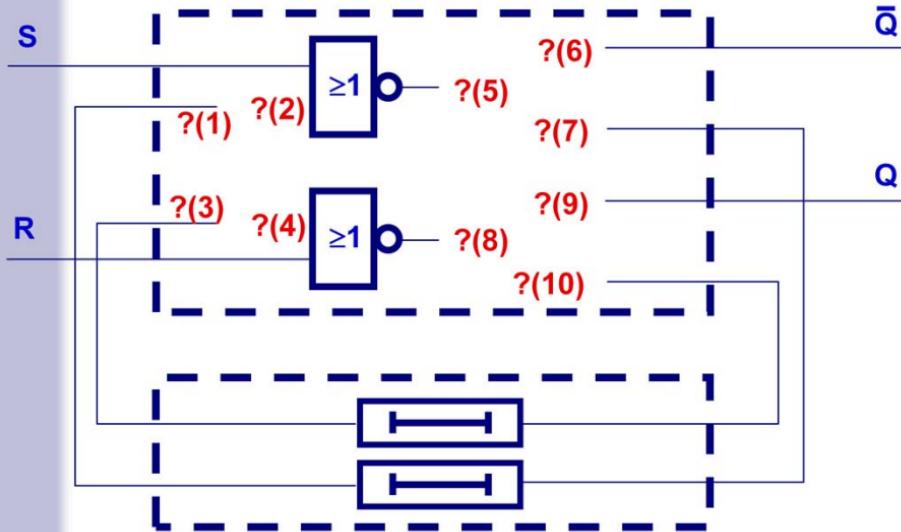
# RS-FF

$S^n$	$R^n$	$Z_i^n$	$Q^n$	$Z_i^{n+1}$	$Q^{n+1}$
0	0	$Z_0$	0	$Z_0$	0
0	0	$Z_1$	1	$Z_1$	1
0	1	$Z_0$	0	$Z_0$	0
0	1	$Z_1$	1	$Z_0$	0
1	0	$Z_0$	0	$Z_1$	1
1	0	$Z_1$	1	$Z_1$	1
1	1	—	0	—	—
1	1	—	1	—	—



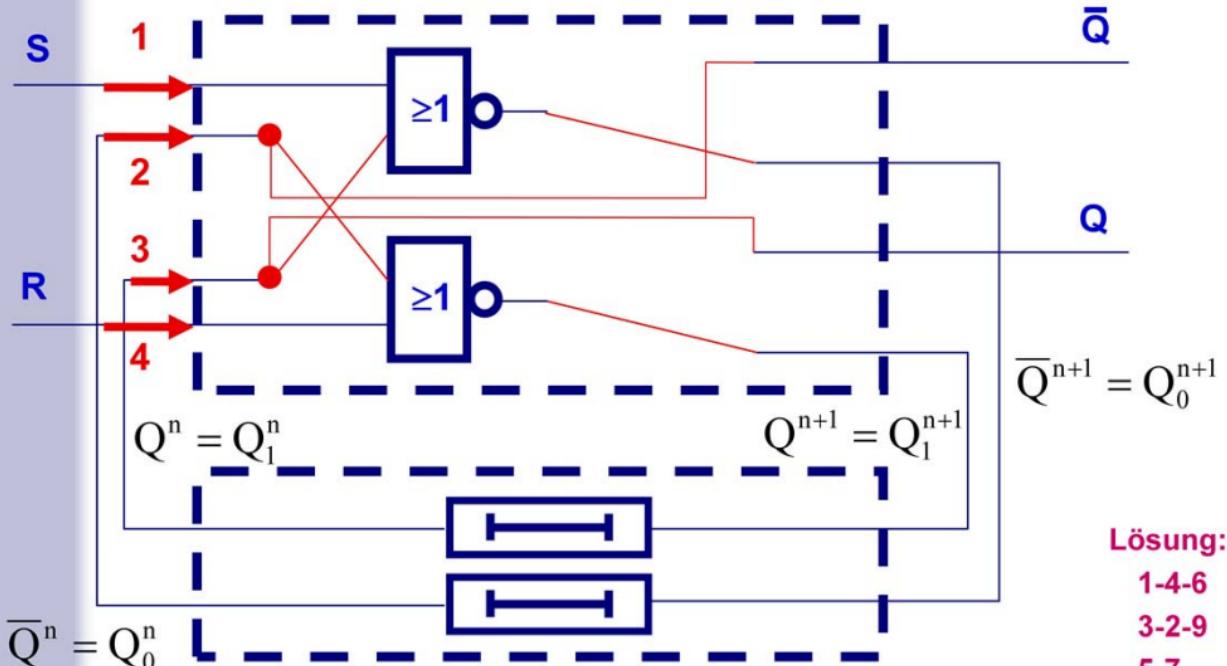
Bisher ist das RS-FF als "einfachste" FF-Architektur aufgeführt. Im folgenden soll anhand dieser Schaltung die Einführung in die Stabilitätsbetrachtung sequentieller Schaltungen erfolgen, wobei man erkennen wird, dass die Funktionsbeschreibung dieses FFs gar nicht so trivial ist. Ausgehend von der Schaltung nach obigen Darstellung und der zugehörigen Übergangstabelle wird deutlich, dass bei einer Modellierung auf der Basis des Mealy-Automaten das zugehörige Schaltnetz 4 Eingangsgrößen und 4 Ausgangsgrößen haben muss, was zur Darstellung auf den folgenden Folien führt, was letztendlich zum Moore-Automaten führt.

## RS-FF als Mealy?



Demgemäß ist eine "vollständigere" Übergangstabelle nur über mindestens 16 Zeilen zu erhalten.

## RS-FF als Mealy



Lösung:  
1-4-6  
3-2-9  
5-7  
8-10

## RS-FF als Mealy: vollständige Ü-Tabelle

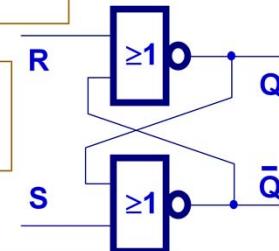
= 4 Eingänge  $\Rightarrow$  16 Zeilen der Übergangstabelle  
(bisher 8 Zeilen)

Eingangsgrößen sind:

- S
- R
- $Q_1(t^n)$
- $Q_0(t^n)$

Zustandsgrößen sind:

- $Q_1 = (Q)$
- $Q_0 = (\neg Q)$



$S^n$	$R^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	i: instabil s: stabil	$Z_i$
0	0	0	0	1	1	i	$Z_0$
0	0	0	1	0	1	s	$Z_1$
0	0	1	0	1	0	s	$Z_2$

Betrachtet man die logischen Zusammenhänge und setzt konkrete Werte ein, wird deutlich, dass bestimmte Zustandsübergänge stabil (s), andere instabil (i).

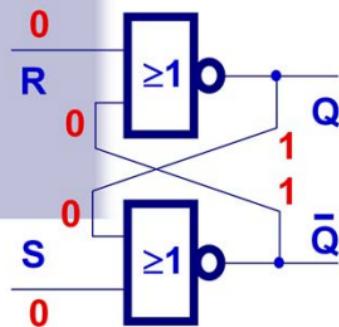
## RS-FF als Mealy: vollständige Ü-Tabelle

$S^n$	$R^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	i: instabil s: stabil	$Z_i$
0	0	0	0	1	1	i	$Z_0$
0	0	0	1	0	1	s	$Z_1$
0	0	1	0	1	0	s	$Z_2$

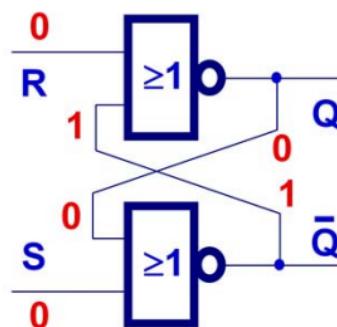
$$Q_1 = (Q)$$

$$Q_0 = (\neg Q)$$

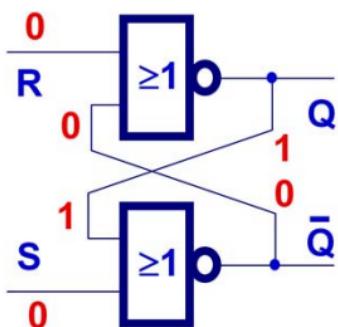
Zeile 1: instabil



Zeile 2: stabil



Zeile 3: stabil



# RS-FF als Mealy: vollständige Ü-Tabelle

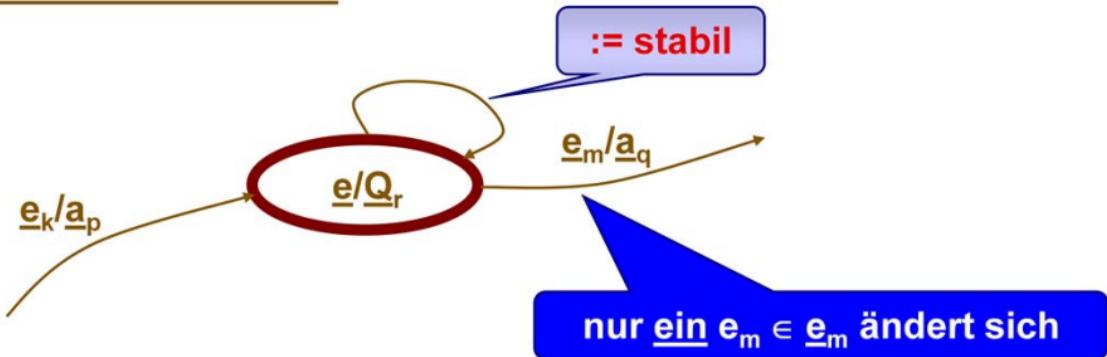
S <sup>n</sup>	R <sup>n</sup>	Q <sub>1</sub> <sup>n</sup>	Q <sub>0</sub> <sup>n</sup>	Q <sub>1</sub> <sup>n+1</sup>	Q <sub>0</sub> <sup>n+1</sup>	i: instabil s: stabil
0	0	0	0	1	1	i
0	0	0	1	0	1	s
0	0	1	0	1	0	s
0	0	1	1	0	0	i
0	1	0	0	0	1	i
0	1	0	1	0	1	s
0	1	1	0	0	0	i
0	1	1	1	0	0	i
1	0	0	0	1	0	i
1	0	0	1	0	0	i
1	0	1	0	1	0	s
1	0	1	1	0	0	i
1	1	0	0	0	0	s
1	1	0	1	0	0	i
1	1	1	0	0	0	i
1	1	1	1	0	0	i

## Stabile Zustände

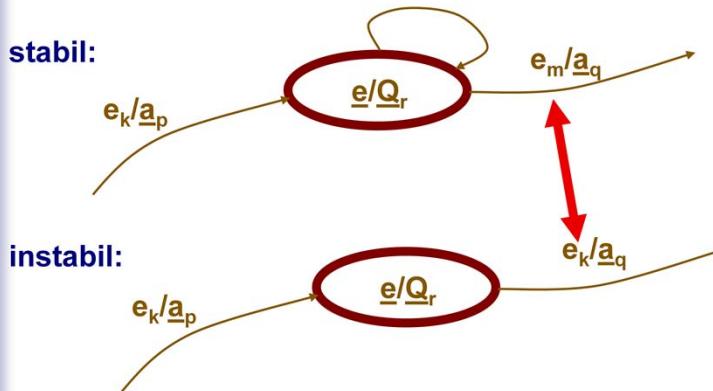
$S^n$	$R^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	i: instabil s: stabil
0	0	0	1	0	1	s
0	0	1	0	1	0	s
0	1	0	1	0	1	s
1	0	1	0	1	0	s
1	1	0	0	0	0	s

## Unterscheidung: stabile & instabile Zustände

- bei bisherigen Zustandsdiagrammen keine Unterscheidung
- bisheriges Zustandsdiagramm gilt prinzipiell nur für synchrone Schaltwerke (= stabil)
- für asynchrone Schaltwerke: Hinzufügung von Stabilitätsinformation



## stabil - instabil



Satz:

Ein Zustand eines Systems wird durch die **Zustände** der systeminternen Speicherelemente & der momentanen **Eingangsvariablen** beschrieben:

Um diese Verhältnisse in einem Zustandsdiagramm deutlicher beschreiben zu können, wird eine Symbolik speziell für asynchrone Schaltwerke eingeführt.

Bild oben zeigt dies für einen Übergang über einen stabilen Knoten, das heißt, durch eine Aktion wechselt das System von einem stabilen Zustand in einen weiteren stabilen Zustand  $Z_p$  und verharrrt darin, bis eine Eingangsgrößenänderung erfolgt. Das Bild darunter zeigt dies für einen Übergang über einen instabilen Knoten: Durch eine Aktion wechselt das System in den Zustand  $Z_p$ , verharrrt aber nicht darin, sondern geht direkt ohne Eingangsänderung in den Folgezustand  $Z_{p+1}$  über. Durch die neue Symbolik wird der Zustand eines Systems also einmal über die systeminternen Speicherelemente (innerer Zustandsvektor) und zum anderen über die momentanen Eingangsgrößen (Eingangsvektor) beschrieben. Dass die Kanten nur mit *einem* Eingangsskalar und nicht mit einem Eingangsvektor beschriftet werden, hat einen wichtigen Grund: Ein absolut zeitgleicher Wertewechsel zweier Eingangsvariablen muss vermieden werden, da dies physikalisch wohl kaum zu realisieren ist. Ließe man einen Vektor zu, würde dies nicht den realen physikalischen Gegebenheiten entsprechen, den die Verzögerungszeiten beider Transistoren werden immer unterschiedlich sein.

Durch die Einbeziehung der Eingangszustandsgrößen in die Knoten nimmt die Anzahl der Knoten stark zu.

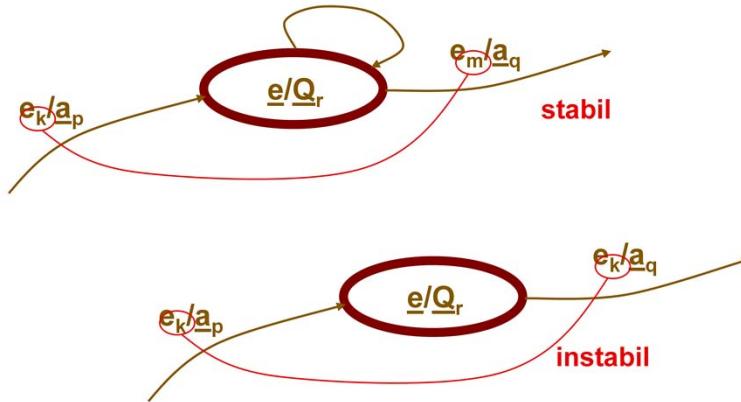
## Physik - Festlegung

- ❖ Aus physikalischen Gründen kann sich immer nur eine Eingangsgröße zu einem Zeitpunkt ändern.
- ❖ Ein stabiler Zustand wird nur durch die Änderung einer Eingangsgröße verlassen.
- ❖ Ein System ist dann beschreibbar, wenn die Eingangsgröße sich dann ändert, wenn sich das System im stabilen Zustand befindet.

# Stabilität asynchroner Schaltwerke

Zustandsänderung durch:

- ❖ Änderung der Eingangsvariablen
- ❖ Änderung der Zustandsvariablen (weil instabiler Zustand!)



Aus diesen Überlegungen heraus kann man für die Beschreibung und den Entwurf asynchroner Schaltwerke schon folgende drei Grundprinzipien festhalten:

1. Zu einem Zeitpunkt kann sich nur *eine* Eingangsgröße ändern.
2. Ein *stabiler* Zustand wird nur durch die Änderung *einer* Eingangsgröße verlassen.
3. Ein System ist beschreibbar, wenn die Eingangsgröße sich nur dann ändert, wenn sich das System im *stabilen* Zustand befindet.

# Stabilität

## *Unterscheidung:*

- ❖ stabiler / instabiler Zustand
- ❖ stabiles / instabiles System

## Zustand:

Ein Zustand ist **stabil**, wenn er nur über die Änderung einer **Eingangsgröße** verlassen wird.

\*

Ein Zustand ist **instabil**, wenn er direkt nach Erreichen sofort wieder verlassen wird, ohne dass sich eine Eingangsgröße verändert.

# Stabilität

## System:

Hat eine Eingangsgrößenänderung

$$\underline{Q}^{n+1} \neq \underline{Q}^n$$

zur Folge

und wechselt das System nach einer **endlichen** Zahl von Zustandsänderungen in einen Zustand

$$\underline{Q}^{n+1} = \underline{Q}^n ,$$

nennt man das System stabil.

*Für den Entwurf asynchroner Schaltwerke gilt:*

**Die Einschwingzeit ist stets zu berücksichtigen!**



Oben ist die exakte Definition der Stabilität für asynchrone Schaltwerke. Sie ist wie folgt zu treffen:

Befindet sich ein System in einem stabilen Zustand  $Z_p$  mit dem Zustandsvektor  $\underline{Q}^n$ , und hat durch die Änderung des Eingangsvektors  $\underline{e}^n$  das System den Wechsel

$$\underline{Q}^{n+1} \neq \underline{Q}^n$$

zur Folge, und wechselt dabei das System nach einer endlichen Zahl von Zustandsänderungen, wobei jeweils

$$\underline{Q}^{n+1} \neq \underline{Q}^n$$

gilt, in einen Zustand

$$\underline{Q}^{n+1} = \underline{Q}^n ,$$

nennt man das System stabil (die Anzahl der durchlaufenen instabilen Zustände kann selbstverständlich Null sein).

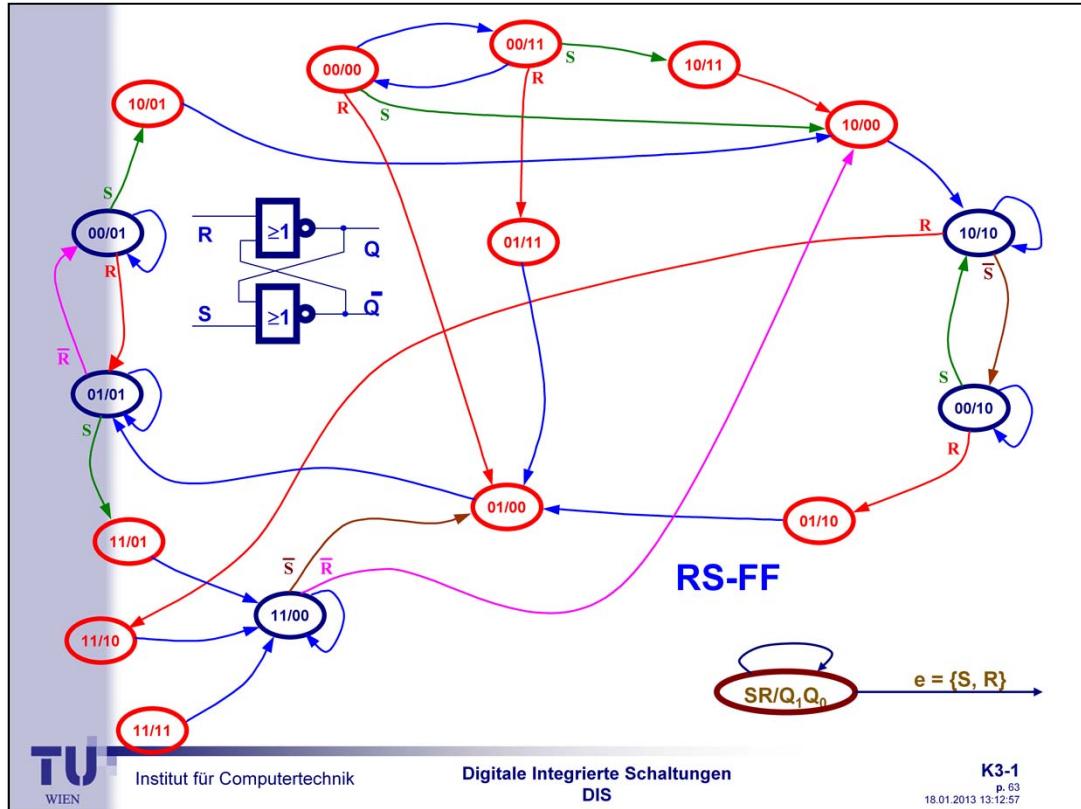
Für einen konkreten Schaltungsentwurf asynchroner Schaltwerke lässt sich daraus direkt ableiten, dass für die Stabilitätsbetrachtung die Einschwingzeit eines Systems eine maßgebliche Rolle spielt. Denn erst nach Ablauf der Einschwingzeit darf sich eine Eingangsgröße wieder ändern, sonst verhält sich das System willkürlich und ist per Definition nicht mehr beschreibbar.

## RS-FF als Mealy: vollständige Ü-Tabelle

$S^n$	$R^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	i: instabil s: stabil
0	0	0	0	1	1	i
0	0	0	1	0	1	s
0	0	1	0	1	0	s
0	0	1	1	0	0	i
0	1	0	0	0	1	i
0	1	0	1	0	1	s
0	1	1	0	0	0	i
0	1	1	1	0	0	i
1	0	0	0	1	0	i
1	0	0	1	0	0	i
1	0	1	0	1	0	s
1	0	1	1	0	0	i
1	1	0	0	0	0	s
1	1	0	1	0	0	i
1	1	1	0	0	0	i
1	1	1	1	0	0	i

Auf diesem Gedankenmodell aufbauend kann ein Zustandsdiagramm des RS-FFs nach dem Bild der nächsten Folie gezeichnet werden. Befindet sich das System beispielsweise im stabilen Zustand  $SR/Q_1Q_0 = 00/01$ , kann es diesen nur verlassen, wenn S oder R den Wert 1 annehmen. Ein instabiler Zustand wird dagegen direkt verlassen, wobei sich die Werte der Eingangsgrößen nicht verändern.

Eine Inkonsistenz ist zu erkennen: Die Übergänge aus den beiden Zuständen 00/00 und 00/11. Sie werden später (Thema: Races, Hazards und Spikes) noch näher analysiert.



Gezeichnet sind die Menge der *totalen Zustände*, das heißt, die Menge aller Eingangszustände und inneren Zustände.

Synchrone Schaltwerke verhalten sich gegenüber asynchronen Schaltwerken bezüglich der Stabilität entscheidend verträglicher. Sie sind nach der obigen Definition stets stabil, da sich die inneren Zustände nur in Abhängigkeit vom Clock ändern.

Hierin liegt der entscheidende Grund, warum komplexe Schaltwerke wie der Mikroprozessor heute als synchrones Schaltwerk realisiert werden. Theoretisch wäre er auch als asynchrones System aufzubauen, allein die Beschreibung fällt dann schwer und ist Thema heutiger Forschungsarbeiten.

Als Konsequenz daraus folgt, dass die Beschreibung des synchronen Schaltwerkes einfach wird.

# Asynchrone Schaltungen



Synchron (= gleichzeitig, gleichlaufend)

## 1. Im physikalischen Sinn:

*Zwei Signale laufen synchron, wenn ihre korrespondierenden Merkmale zum gleichen Zeitpunkt auftreten (Phasengleichheit).*

## 2. Im Sinne der digitalen Codierung:

*Synchronität bedeutet Phasen- und Taktgleichheit und Korrespondenz von Bits.*

## 3. Im Sinne der digitalen Datenübertragung:

*Synchronität ist die periodische Wiederkehr in festen Abständen der Zuteilung von Übertragungskapazität zu einem Kanal auf einem Medium*

## 4. Im Sinne der Datenkommunikation:

*Zwei Kommunikationspartner (Sender und Empfänger) laufen synchron, wenn sie sich im gleichen Stadium eines die Kommunikation regelnden Algorithmus (Protokoll) befinden.*

# Asynchrone Schaltungen



Synchron (= gleichzeitig, gleichlaufend)

## 1. Im physikalischen Sinn:

*Zwei Signale laufen synchron, wenn ihre korrespondierenden Merkmale zum gleichen Zeitpunkt auftreten (Phasengleichheit).*

?

## 2. Im Sinne der digitalen Codierung:

*Synchronität bedeutet Phasen- und Taktgleichheit und Korrespondenz von Bits.*

## 3. Im Sinne der digitalen Datenübertragung:

*Synchronität ist die periodische Wiederkehr in festen Abständen der Zuteilung von Übertragungskapazität zu einem Kanal auf einem Medium*

## 4. Im Sinne der Datenkommunikation:

*Zwei Kommunikationspartner (Sender und Empfänger) laufen synchron, wenn sie sich im gleichen Stadium eines die Kommunikation regelnden Algorithmus (Protokoll) befinden.*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K3-1

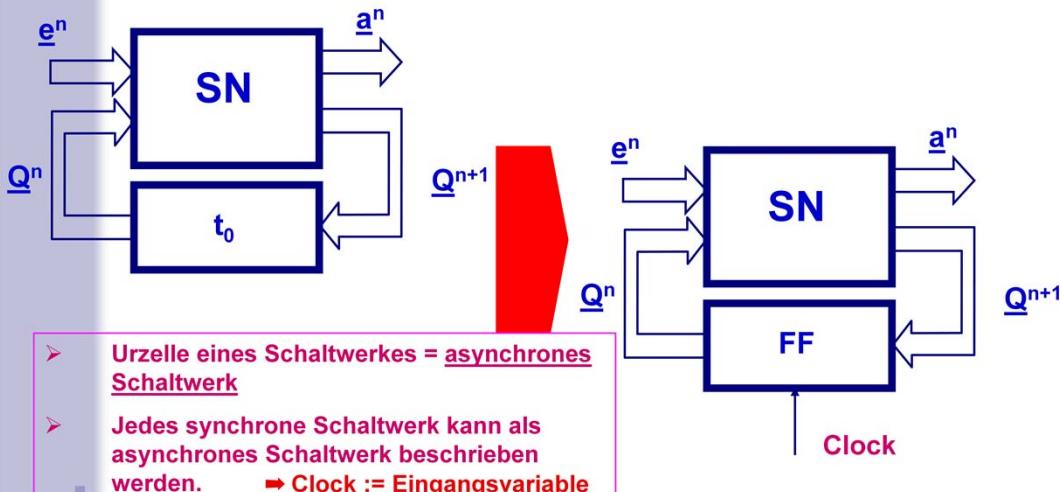
p. 65

18.01.2013 13:12:57

Der Ausdruck "gleicher Zeitpunkt" ist im Grunde falsch. In einer Anwendung hat beispielsweise ein Transputer als Master gearbeitet und synchron Daten verschickt. In den Slaves arbeiteten State-Machines, die sich über die einlaufenden Impulse einsynchronisierten. Alle Systeme liefen synchron, aber mit einem Zeitversatz von einigen Bits hervorgerufen durch die Laufzeit auf den Leitungen. Im Sendemodus der Slaves hat das natürlich nicht funktioniert, der Transputer hat die Daten der Slaves im Asynchronen Modus dann übernommen.

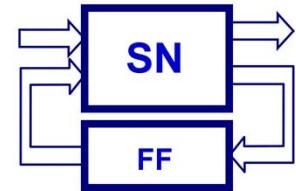
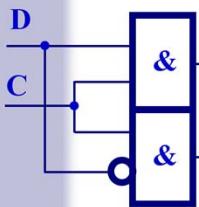
## Synchrone Schaltwerke:

- Synchronisation: Clock!
- innere Zustände ändern sich nur mit Änderung des Clocks!



Legt man den Formalismus des asynchronen Schaltwerkes zugrunde, ist bei einem synchronen Schaltwerk der Clock als Eingangsgröße zu definieren, wobei man im allgemeinen die rechte Darstellung von oben wählt. Da jedoch der Clock in synchronen Schaltwerken selbstverständlich ist, kann man ihn auch weglassen. Für das Übergangsdiagramm gilt entsprechendes, was das Beispiel des taktzustandgetriggerten RS-FFs verdeutlichen soll (nächste Folie).

## Beispiel: Taktzustandsgesteuertes RS-FF synchrone RS-FF



$C^n$	$S^n$	$R^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$		$S^n$	$R^n$	$Q^{n+1}$
0	0	0	0	0	0	0		0	0	$Q^n$
0	0	0	0	1	0	1		0	1	0
..					..	..		..	..	..

Voraussetzung:

$T >> t_{\text{innere Einschwingzeit}}$

## Modell

eines synchronen  
Schaltwerkes

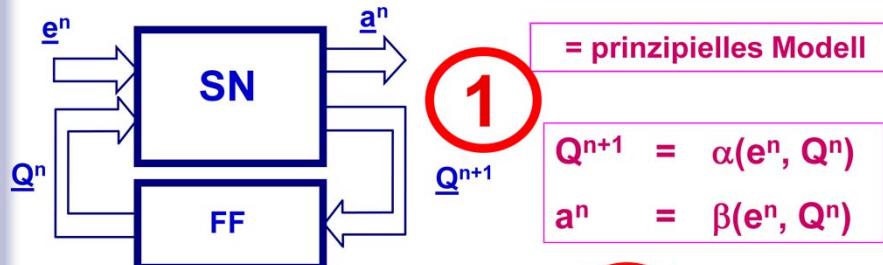
Sieht man das RS-FF als asynchrones Schaltwerk, gelangt man zur Übergangstabelle nach Tab. oben, linke Darstellung. Betrachtet man es als synchrones Schaltwerk, gelangt man zur Übergangstabelle oben rechte Darstellung. Es versteht sich nahezu von selbst, dass diese Art der Modellierung voraussetzt, dass bei einem synchronen Schaltwerk

$$\text{Periodendauer}_{\text{Taktfrequenz}} >> t_{\text{innere Einschwingvorgänge}}$$

gilt. Nur dann kann das Schaltwerk abstrahiert als synchrones Schaltwerk betrachtet werden. Bei einem asynchronen Schaltwerk darf, wie schon erwähnt, die Eingangsänderung erst dann erfolgen, wenn sich das Schaltwerk in einem eingeschwungenen Zustand befindet.

Modell: abstrahierte Beschreibung des physikalischen Systems

## Rechnen mit Automatenmodellen



Charakteristische Gleichungen von FFs:

1. Asynchrones RS-FF:

$$Q^{n+1} = S^n \vee \bar{R}^n Q^n$$

$$\text{NB: } S^n \wedge R^n = 0$$

6. JK-Master-Slave-FF

$$Q^{n+1} = (J^n \wedge \bar{Q}^n) \vee (\bar{K}^n \wedge Q^n)$$

Die Anwendung der Automatentheorie ist nicht schwer, verlangt jedoch etwas Praxis. Hier werden die notwendigen Gleichungen hergeleitet und einige wenige Beispiele vorgeführt. Auf umfangreiche Beispiele wird in der Vorlesung verzichtet. Am Ende von Kapitel 3 werden zusätzlich charakteristische Aufgaben vorgestellt, an Hand denen man ebenfalls die Umsetzung der Theorie in die Praxis herausarbeiten kann.

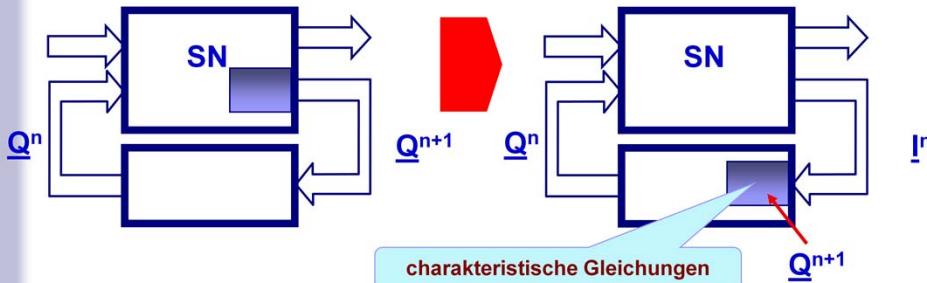
Rechnen mit Automatenmodellen:

Verwendet werden prinzipiell:

1. Modellgleichungen
2. charakteristische Gleichungen

## Wie damit rechnen?

Miteinbeziehung der charakteristischen Gleichungen in die Gleichungen der Automaten



$$I^n = \alpha_2(e^n, Q^n) : \text{Erregerfunktion } \equiv SN$$

$$Q^{n+1} = \alpha_1(I^n, Q^n) : \text{Übergangsfunktion}$$

Die rechte Darstellung des Mealy-Automaten nach obigem Bild impliziert eine noch weitergehende, vereinfachende Möglichkeit der Beschreibung synchroner Schaltwerke. Das D-Latch hat ja die relativ einfache charakteristische Gleichung:

$$Q^{n+1} = D^n .$$

Verwendet man dagegen andere FFs, wie beispielsweise das JK-FF, liegt die Beschaltung der FFs nach dem prinzipiellen Modell des Mealy-Automaten im Schaltnetz, dem "oberen" Block der rechten Darstellung. Ob es in allen Fällen dann noch als synchrones Schaltwerk beschreibbar bleibt, ist zu bezweifeln. Man wählt besser einen anderen Weg.

Die Übergangsfunktion  $Q^{n+1}$  wird in erweiterter Form formuliert, wobei die Komponente  $\alpha_2(e^n, Q^n)$  zur Definition der sogenannten *Erregungsfunktion* (engl.: Irritation Function) führt. Die Übergangsfunktion selbst ist jetzt die charakteristische Gleichung und wandert, bildlich gesehen, in den unteren Block des Mealy-Automaten. Ausgangsgröße des Schaltnetzes ist nun  $I^n$ , was im jeweiligen Anwendungsfall zu berechnen ist.

## Rechnen mit Automatenmodellen

$$\underline{I}^n = \alpha_2(\underline{e}^n, \underline{Q}^n) : \text{Erregerfunktion} \equiv \text{SN}$$

$$\underline{Q}^{n+1} = \alpha_1(\underline{I}^n, \underline{Q}^n) : \text{Übergangsfunktion}$$

beide Gleichungen ineinander eingesetzt:

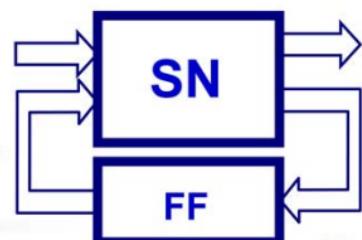
$$\underline{Q}^{n+1} = \alpha_1[\alpha_2(\underline{e}^n, \underline{Q}^n), \underline{Q}^n]$$

bedeutet: 2 x Feedback!

$\alpha_1$  und  $\alpha_2$  zusammengefasst:

$$\underline{Q}^{n+1} = \alpha[\underline{e}^n, \underline{Q}^n]$$

wovon ausgegangen wurde



## Vollständiges Gleichungssystem für synchrone Schaltwerke

Mit:

$$\underline{e}^n = (e_{p-1}, e_{p-2}, \dots, e_0) : \text{Eingangsvektor}$$

$$\underline{a}^n = (a_{q-1}, a_{q-2}, \dots, a_0) : \text{Ausgangsvektor}$$

$$\underline{Q}^n = (Q_{r-1}, Q_{r-2}, \dots, Q_0) : \text{Zustandsvektor}$$

$$\underline{l}^n = (l_{s-1}, l_{s-2}, \dots, l_0) : \text{Erregungsvektor}$$

ergibt sich das Gleichungssystem:

$$\underline{a}^n = \beta(\underline{e}^n, \underline{Q}^n) : \text{Ausgangsfunktion}$$

$$\underline{Q}^{n+1} = \alpha_1(\underline{l}^n, \underline{Q}^n) : \text{Übergangsfunktion} \\ (= \text{charakteristische Gl.})$$

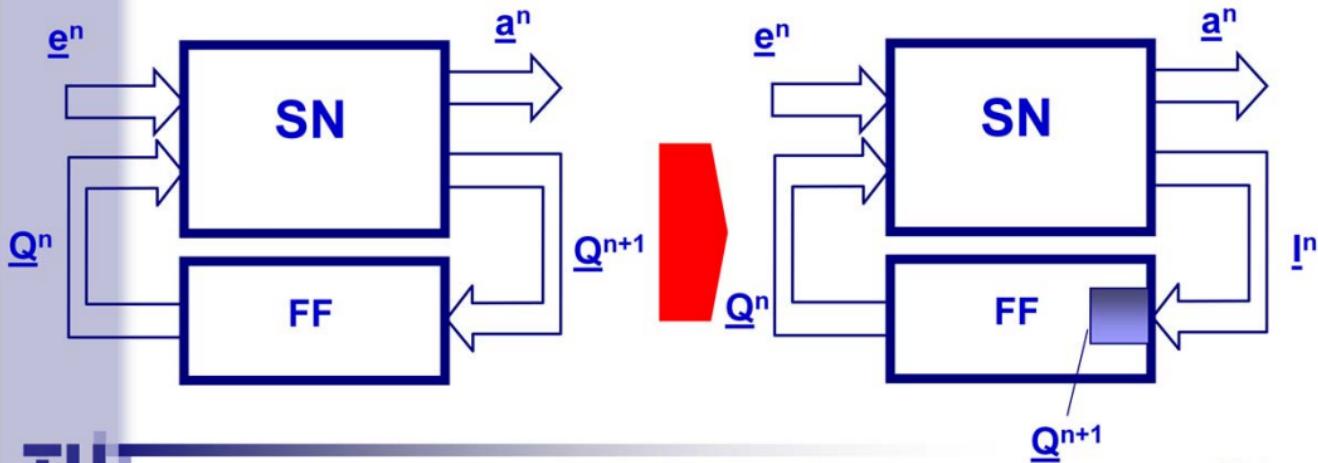
$$\underline{l}^n = \alpha_2(\underline{e}^n, \underline{Q}^n) : \text{Erregungsfunktion}$$

## Vollständiges Gleichungssystem für synchrone Schaltwerke

$$\underline{a}^n = \beta(\underline{e}^n, \underline{Q}^n) : \text{Ausgangsfunktion}$$

$$\underline{Q}^{n+1} = \alpha_1(\underline{I}^n, \underline{Q}^n) : \text{Übergangsfunktion} \\ (= \text{charakteristische GI.})$$

$$\underline{I}^n = \alpha_2(\underline{e}^n, \underline{Q}^n) : \text{Erregungsfunktion}$$



## Automat $\Leftrightarrow$ Charakteristische Gleichung

In  $Q^{n+1}$  ist der

- FF-Anteil (= charakteristische Gleichung) und
- SN-Anteil (= Erregergleichung)

enthalten.

Aus der Gesamtfunktion



$$Q^{n+1} = \alpha(e^n, Q^n)$$

ist deshalb der FF-Anteil (= charakteristische Gleichung) für eine Schaltungssynthese herauszuholen

(bspw. über Koeffizientenvergleich).

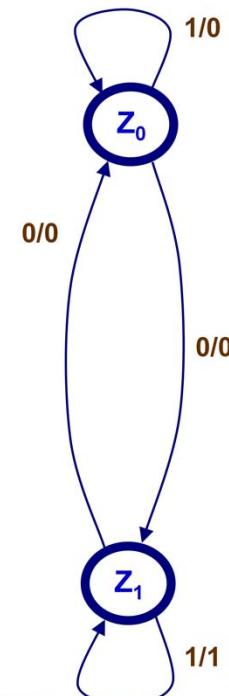
# Praktische Umsetzung

## Beispiel D-FF

$$Q^{n+1} = D^n = I^n : \quad$$

*Übergangsfunktion  
= charakteristische Gl.*

e <sub>n</sub>	Q <sub>n</sub>	Z <sub>n</sub>	Z <sub>n+1</sub>	I <sub>n</sub>	a <sub>n</sub>
0	0	Z <sub>0</sub>	Z <sub>1</sub>	1	0
0	1	Z <sub>1</sub>	Z <sub>0</sub>	0	0
1	0	Z <sub>0</sub>	Z <sub>0</sub>	0	0
1	1	Z <sub>1</sub>	Z <sub>1</sub>	1	1



Dass die Umsetzung des bis jetzt beschriebenen Formalismus in die Praxis sich relativ einfach gestaltet, soll anhand eines Beispiels gezeigt werden. Ein Schaltwerk, beschrieben durch einen Zustandsgrafen, wird beispielhaft auf der Basis dreier synchroner FFs: D-FF, JK-FF und RS-FF, entwickelt.

Die Entwicklung soll ohne die Unterstützung eines Computers erfolgen, da sonst die Gefahr besteht, dass das Prinzip nicht deutlich genug erkennbar wird (zudem ist das Beispiel trivial). Interessieren soll an dieser Stelle auch nicht das exakte Timing und weitergehende Aspekte, von Interesse sei ausschließlich die logische Schaltwerksentwicklung.

Entwicklung auf der Basis von D-FFs

Es liegen nur zwei unterschiedliche Zustände vor, Z<sub>0</sub> und Z<sub>1</sub>, was bedeutet, dass nur ein Speicherelement notwendig wird. Verwendet werden soll das D-FF, das die charakteristische Gleichung

$$Q^{n+1} = D^n$$

beinhaltet. Das bedeutet, die Erregungsvariable ist in diesem Fall identisch der Zustandsgröße Q<sub>n+1</sub> (siehe Gleichung oben). Die Werte sind aus der Übergangstabelle zu ermitteln, die wiederum aus dem Zustandsdiagramm abgeleitet werden können.

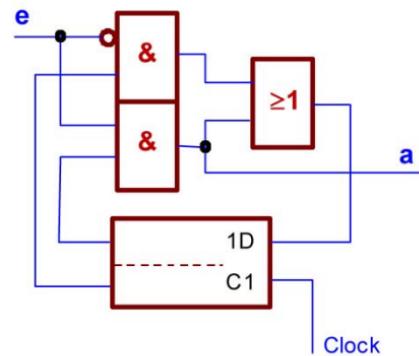
$$Q^{n+1} = D^n = I^n$$

$e^n$	$Q^n$	$Z^n$	$Z^{n+1}$	$I^n$	$a^n$
0	0	$Z_0$	$Z_1$	1	0
0	1	$Z_1$	$Z_0$	0	0
1	0	$Z_0$	$Z_0$	0	0
1	1	$Z_1$	$Z_1$	1	1

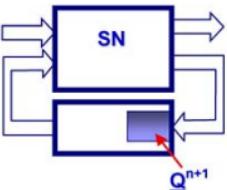


$$I^n = \overline{e^n Q^n} \vee e^n Q^n = D^n$$

$$a^n = e^n Q^n$$



Daraus lässt sich das Ergebnis direkt ablesen, da keine Vereinfachung möglich ist.

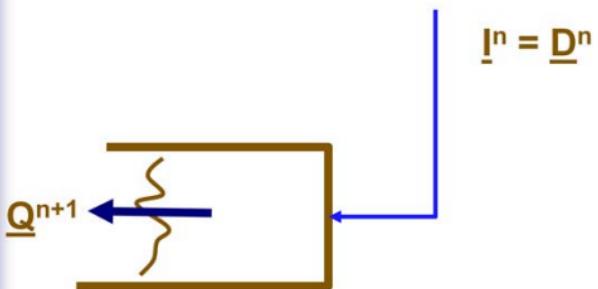


Wie kann man sich das grafisch vorstellen?

charakteristische Gleichung:  $\underline{Q}^{n+1} = \alpha_1(\underline{I}^n, \underline{Q}^n)$

hier:

$$\underline{Q}^{n+1} = \underline{D}^n$$



$$\underline{I}^n = \underline{D}^n$$

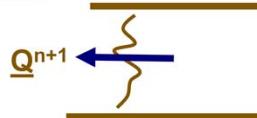
$$\underline{I}^n = \alpha_2(\underline{e}^n, \underline{Q}^n)$$

Tab. aus Zustandsdiagramm



Gleichung

### Beispiel JK-FF



$$\underline{Q}^{n+1} = \alpha_1(\underline{I}^n, \underline{Q}^n)$$

$$\underline{I}^n = \underline{I}^n(\underline{J}, \underline{K})$$

charakteristische Gleichung:

$$Q^{n+1} = \alpha_1(I^n, Q^n)$$

hier:

$$Q^{n+1} = J^n \overline{Q^n} \vee \overline{K^n} Q^n \quad (*)$$

Erregungsfunktion:

$$I^n = \overline{e^n Q^n} \vee e^n Q^n \quad (**)$$

gesucht: Zusammenhang zw.  $\underline{Q}^{n+1}$  und  $I^n$

Lösung: Koeffizientenvergleich zw. (\*) und (\*\*)



$$J^n = \overline{e^n}$$

$$\overline{K^n} = e^n \Rightarrow K^n = \overline{e^n}$$

Entwicklung auf der Basis von JK-FFs

Aus der Tab. der Aufgabenstellung (siehe vorhergehende Folien) gewinnt man entsprechend die Gleichung für die Erregungsfunktion.

## Beispiel JK-FF

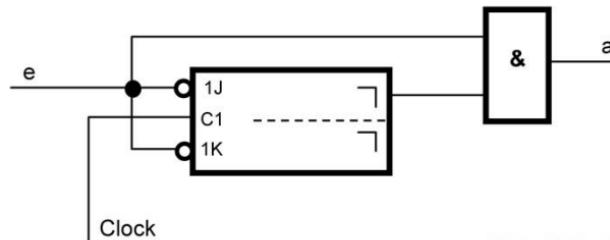
Erregungsfunktion:

$$I^n = \overline{e^n Q^n} \vee e^n Q^n$$

Eingangsfunktion des FF:

$$J^n = \overline{e^n}$$

$$\overline{K^n} = e^n \Rightarrow K^n = \overline{e^n}$$



Wo bleibt die Rückkopplung?

Denn jeder Mealy-Automat beinhaltet eine Rückkopplung!

## Beispiel RS-FF

charakteristische Gleichung:

$$Q^{n+1} = \alpha_1(I^n, Q^n)$$

hier:

$$Q^{n+1} = S^n \vee \overline{R^n} Q^n$$

NB:  $S^n R^n = 0$

### 1. Lösungsmöglichkeit:

Lösung des Gleichungssystems!

Führt oft zu Schwierigkeiten!

Stichwort: Überführung in eine explizite Funktion

Beispiel einer impliziten Funktion:

$$f_1(e_{n-1}, e_{n-2}, \dots, e_0, a_{m-1}, a_{m-2}, \dots, a_0) = f_2(e_{p-1}, e_{p-2}, \dots, e_0)$$



## Entwicklung auf der Basis von RS-FFs

Der Lösungsweg über die Entwicklung der Erregerfunktion und den folgenden Koeffizientenvergleich ist für eine Schaltwerksentwicklung auf der Basis der RS-FFs nur schwer möglich. Das Gleichungssystem des RS-FFs erfordert durch die zusätzliche Nebenbedingung die Lösung der implizit gegebenen Schaltfunktion. In der Praxis schlägt man deshalb im allgemeinen einen einfacheren Weg ein.

## **Beispiel RS-FF**

### **2. Lösungsmöglichkeit:**

**Erweiterung der Tabelle um die Variablen:  $R_i^n, S_i^n$**



$e^n$	$Q^n$	$Z^n$	$Z^{n+1}$	$Q^{n+1}$	$S^n$	$R^n$	$a^n$
0	0	$Z_0$	$Z_1$	1	1	0	0
0	1	$Z_1$	$Z_0$	0	0	1	0
1	0	$Z_0$	$Z_0$	0	0	x	0
1	1	$Z_1$	$Z_1$	1	x	0	1

## **Beispiel RS-FF**

**Wie kommt man dazu?**

**Zugrunde gelegt wird die RS-FF-Tabelle:**

$S^n$	$R^n$	$Z_i^n$	$Q^n$	$Z_i^{n+1}$	$Q^{n+1}$
0	0	$Z_0$	0	$Z_0$	0
0	0	$Z_1$	1	$Z_1$	1
0	1	$Z_0$	0	$Z_0$	0
0	1	$Z_1$	1	$Z_0$	0
1	0	$Z_0$	0	$Z_1$	1
1	0	$Z_1$	1	$Z_1$	1
1	1	-	0	-	-
1	1	-	1	-	-

$S^n$	$R^n$	$Q^n$	$Z_i^{n+1}$
0	0	$Q^n$	$Z_i^n$
0	1	0	$Z_0$
1	0	1	$Z_1$
1	1	-	-

## Beispiel RS-FF

$e^n$	$Q^n$	$Z^n$	$Z^{n+1}$	$Q^{n+1}$	$S^n$	$R^n$	$a^n$
0	0	$Z_0$	$Z_1$	1	1	0	0
0	1	$Z_1$	$Z_0$	0	0	1	0
1	0	$Z_0$	$Z_0$	0	0	x	0
1	1	$Z_1$	$Z_1$	1	x	0	1

**Zusammengefasst:**

$$S^n = \overline{e^n} \underline{Q^n}$$

$$R^n = \overline{e^n} Q^n$$

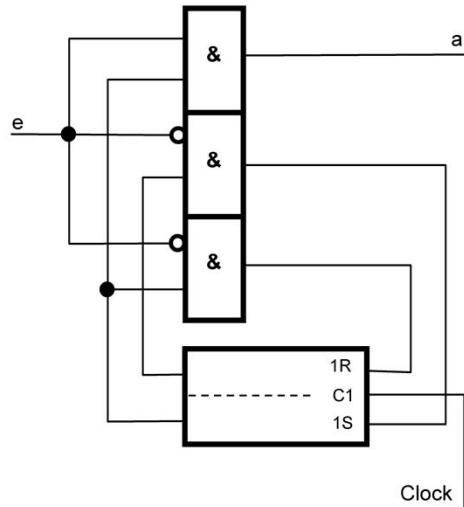
$$a^n = e^n Q^n$$

## Beispiel RS-FF

$$S^n = \overline{e^n Q^n}$$

$$R^n = \overline{e^n Q^n}$$

$$a^n = e^n Q^n$$



Welche der drei Lösungen die geeigneter ist, hängt von der jeweiligen Gesamtschaltung sowie den zu verwendenden FFs ab. Zudem sind hier nur drei Typen von FFs verwendet worden, in der Praxis werden noch andere eingesetzt. Deutlich wird dabei jedoch, dass sich durch eine geschickte Wahl der FFs der Komplexitätsgrad einer Schaltung unter Umständen verringern lässt. Allerdings sollte man nicht umgekehrt zu dem Schluss kommen, dass JK-FFs die einfacheren Lösung garantieren. Im Gegenteil, der "innere" Anteil von JK-FFs ist beträchtlich, warum sie in der Praxis kaum noch verwendet werden.

# Digitale Integrierte Schaltungen

384.086  
Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik  
[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



# Kapitel 3, Teil 2

## Schaltwerke

- Einführung
- Speicherelemente
- Synchrone u. asynchrone Schaltwerke
- Mealy- u. Moore-Automaten
- Races und Hazards
- Synthese und Analyse

Im Folgenden soll ein Beispiel für drei verschiedene FFs durchgerechnet werden, um zu verstehen, wie man die Theorie in die Praxis umsetzen kann.

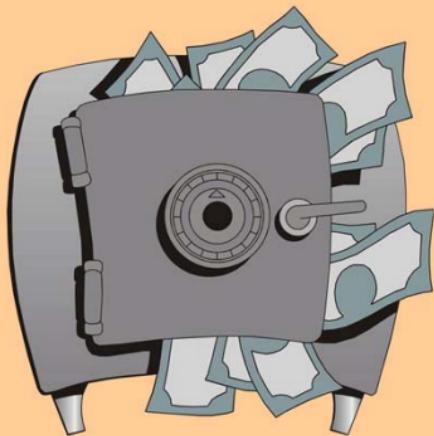
*Beispiel*

## Entwurf eines synchronen Schaltwerkes

### *Zweistelliger Binärzähler mit Übertragsausgang*

#### Spezifikation

- synchroner Zähler
- 2 Modi
- Zählen
- Setzen
- Zählerfolge: 00, 01, 10, 11, 00, 01, ..
- Bei Zählerstand 11 soll während einer Taktperiode für Modus **Zählen** der Übertragsausgang auf 1 gesetzt werden
- Entwurf der Schaltung mit:  
D-FFs, JK-FFs, RS-MS-FFs



*Beispiel*

## Entwurf eines synchronen Schaltwerkes

*Zweistelliger Binärzähler mit Übertragsausgang*

### Spezifikation

- synchroner Zähler
- Modi
- Zählerfolge
- Übertragsausgang
- mit D-, JK-, RS-FFs

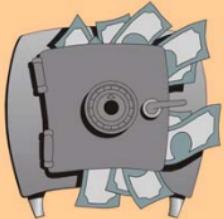
### **WAS IST NICHT SPEZIFIZIERT?**

Wie ist der Modus zu belegen?

Wie wird gesetzt: seriell, parallel, gemixt?

...

**Zu lösen!**



*Nicht interessieren soll:*

*Physik wie*

- *Timing*
- *Technologie*
- ..

**Warum?**

# "Normale" Vorgehensweise

1. Verbalisieren
2. Formalisierung
3. Graf
4. Tabelle / Gleichungsansatz
5. Optimierung
6. Lösung (Gleichungen)



Das Verbalisieren und Formalisieren sind Bestandteile einer Spezifikation. Allerdings sieht man oft noch viel mehr unter dem Begriff der Spezifikation, wobei es allerdings darauf ankommt, in welchem Bereich (ASIC-, Software-Design, Automatisierung usw.). Aus diesem Grund vermeide ich hier diesen Begriff.

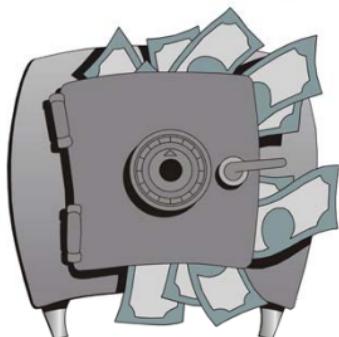
Was man zuerst entwickelt, entweder den Grafen oder die Tabelle (bzw. den Gleichungsansatz), hängt sehr stark von der Aufgabenstellung ab und entscheidet darüber, wie schnell man zur Lösung kommt. Hier muss man sich ein Augenmaß durch Übung erarbeiten.



# Zweistelliger Binärzähler mit Übertragsausgang

Vorgehensweise hier:

1. Verbalisierung
2. Formalisierung
3. Wahl der FFs
4. Charakteristische Gleichung
5. Ermittlung der SN-Tabelle des Automaten
6. Ermittlung der Gleichungen



Aufgabenstellung:

- synchroner Zähler
- Modi: Zählen/Setzen
- Zählerfolge
- ..



1. Verbalisierung ✓

2. Formalisierung:

- Modus-Eingang (Steuereingang)

$M = 0$ : Zählen

$M = 1$ : Zähler setzen

# Zweistelliger Binärzähler mit Übertragsausgang

- synchroner Zähler
- 2 Modi
- Zählen
- Setzen
- Zählerfolge: 00, 01, 10, 11, 00, 01, ...
- bei *Zählen* soll bei Zählerstand 11 der Ausgang C = 1 gesetzt werden
- ..



## 2. Formalisierung

- Moduseingang

- Setzeingänge (Dateneingänge)

- $E_0$  : low Bit

- $E_1$  : high Bit

- Ausgang (Datenausgang)

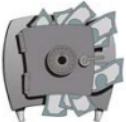
- C : Übertragsausgang stets 0

- nur für  $M = 0 \wedge$  Zählerstand 11 C = 1

- Zustandsvariablen

- $Q_0$  : low Bit

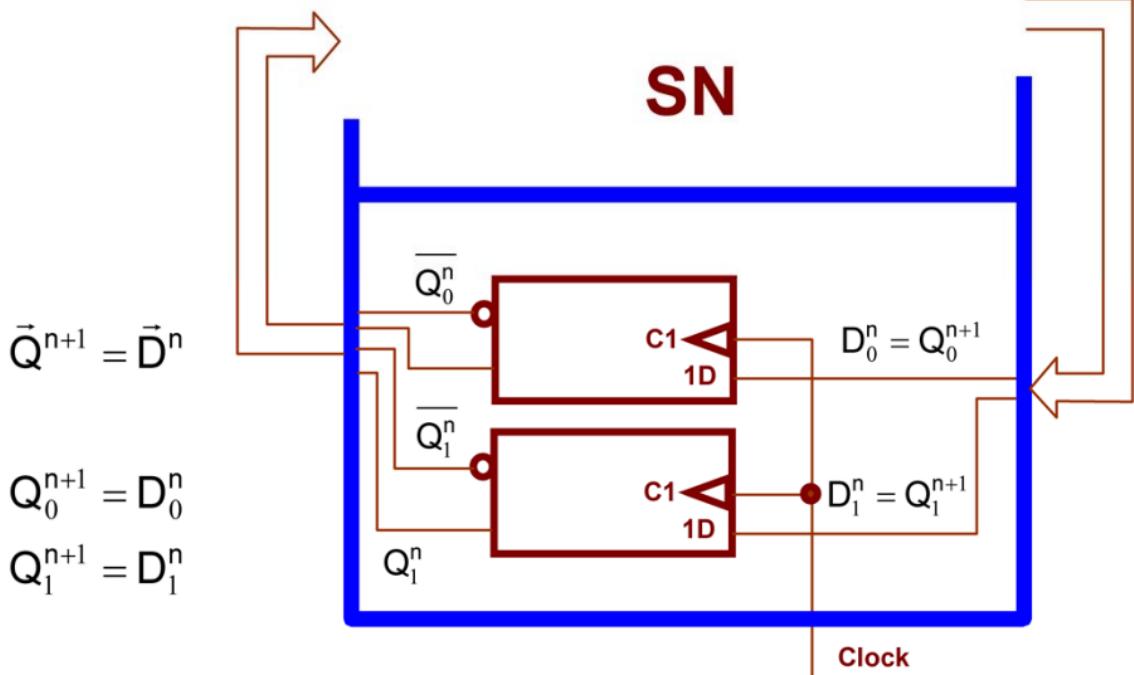
- $Q_1$  : high Bit





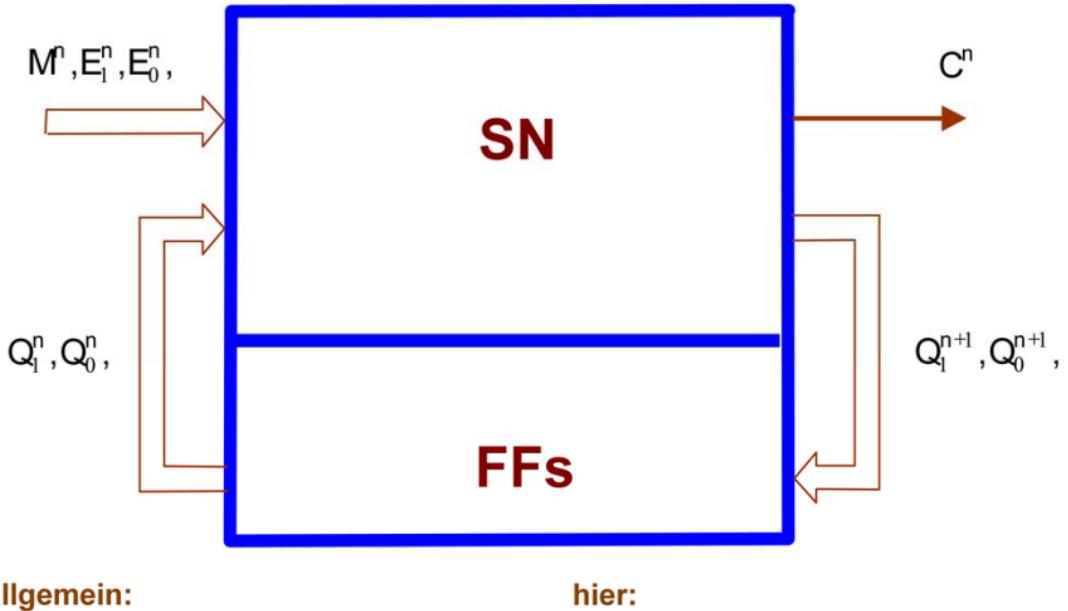
3. Wahl der FFs vorgegeben; 1. Aufgabe: D-FFs  
(einflankengetriggert, z. B. 74LS74)

4. Charakteristische Gleichungen





## 5. Ermittlung der Eingangs- und Ausgangsgrößen



allgemein:

$$\vec{A}^n = \beta(\vec{E}^n, \vec{Q}^n)$$

$$\vec{Q}^{n+1} = \alpha(\vec{E}^n, \vec{Q}^n)$$

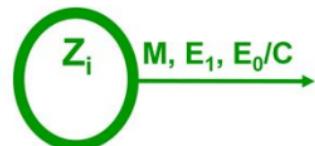
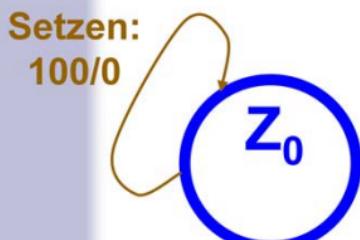


hier:

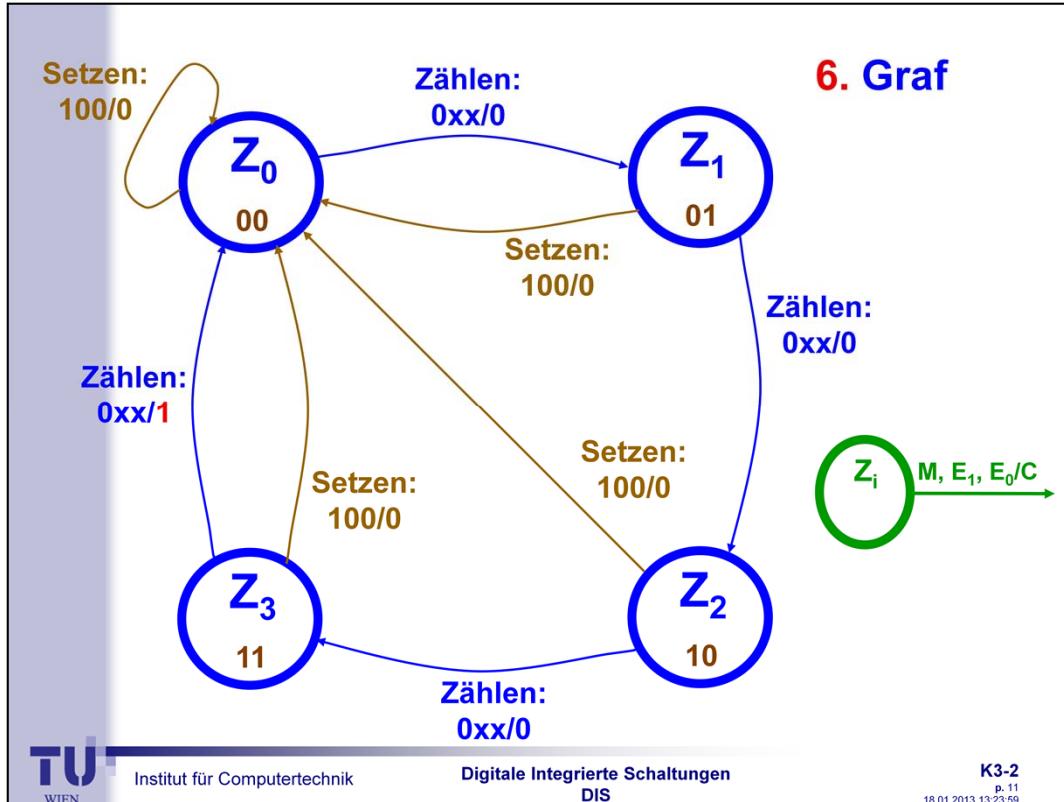
$$C^n = \beta(M^n, E_1^n, E_0^n, Q_1^n, Q_0^n)$$

$$Q^{n+1} = \alpha(M^n, E_1^n, E_0^n, Q_1^n, Q_0^n)$$

## 6. Graf



## 6. Graf



Aufgabe:

- synchroner Zähler
- 2 Modi, Zählen, Setzen
- Zählerfolge: 00, 01, 10, 11, 00, 01, ..
- Bei Zählerstand 11 soll während einer Taktperiode für Modus Zählen der Übertragsausgang auf 1 gesetzt werden.

Der Graf ist hier wegen der Übersichtlichkeit UND weil es nicht notwendig ist, was im folgenden noch gezeigt wird, nicht vollständig gezeichnet. In diesem speziellen Fall ist das Zeichnen des Grafen nämlich gar nicht notwendig. Man kommt viel schneller zum Ziel, wenn man NUR die Tabelle entwirft, die sich hier ganz einfach entwickeln lässt.

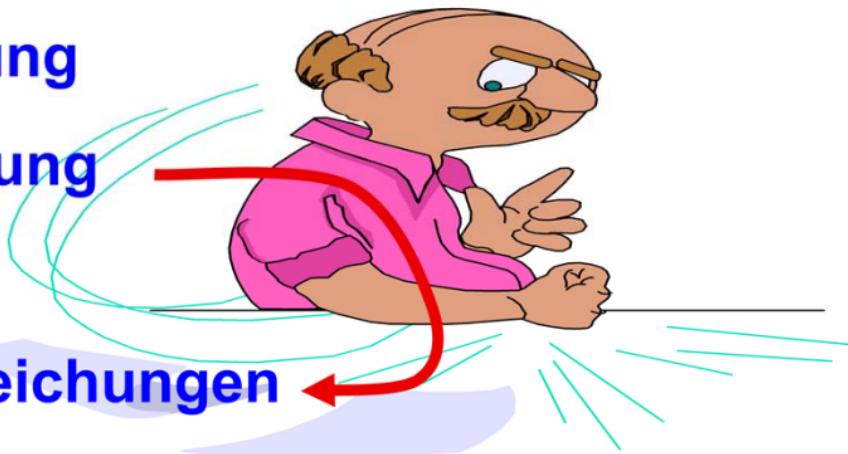
Ist hier ein einfacherer Weg möglich als im  
"Normalfall"?

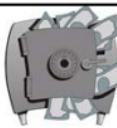
Spezialfall hier 2 voneinander unabhängige Modi:

- Modus  $M = 0$ : Zählen (E-Eingänge interessieren nicht)
- Modus  $M = 1$ : Setzen (Zählerstand interessiert nicht)

# "Normale" Vorgehensweise

1. Verbalisierung
2. Formalisierung
3. Graf
4. Tabelle / Gleichungen
5. Optimierung
6. Lösung (Gleichungen)





# Ermittlung der SN-Tabelle

eigentlich Pkt. 4 in der Vorgehensweise eine Seite vorher

$$C^n = \beta(M^n, E_1^n, E_0^n, Q_1^n, Q_0^n)$$

$$Q^{n+1} = \alpha(M^n, E_1^n, E_0^n, Q_1^n, Q_0^n)$$

5 Eingangsgrößen des Systems:

$$\vec{E}^n = (M^n, E_1^n, E_0^n)$$

$$\vec{Q}^n = (Q_1^n, Q_0^n)$$

3 Ausgangsgrößen des Systems:

$$\vec{A}^n = (C^n)$$

$$\vec{Q}^{n+1} = (Q_1^{n+1}, Q_0^{n+1})$$

Übergangstabelle:

M <sup>n</sup>	E <sub>1</sub> <sup>n</sup>	E <sub>0</sub> <sup>n</sup>	Q <sub>1</sub> <sup>n</sup>	Q <sub>0</sub> <sup>n</sup>	Q <sub>1</sub> <sup>n+1</sup>	Q <sub>0</sub> <sup>n+1</sup>	C <sup>n</sup>

eigentlich Pkt. 5  
in der Vorgehensweise  
2 Seiten vorher

$M^n$	$E_1^n$	$E_0^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	$C^n$
0	x	x	0	0	0	1	0
0	x	x	0	1	1	0	0
0	x	x	1	0	1	1	0
0	x	x	1	1	0	0	1
1	0	0	x	x	0	0	0
1	0	1	x	x	0	1	0
1	1	0	x	x	1	0	0
1	1	1	x	x	1	1	0



Achtung:

x: keine Redundanz

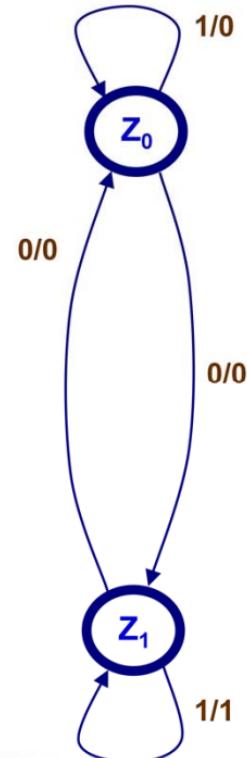
# Praktische Umsetzung

## Beispiel D-FF

$$Q^{n+1} = D^n = I^n : \quad$$

*Übergangsfunktion  
= charakteristische Gl.*

e <sub>n</sub>	Q <sub>n</sub>	Z <sub>n</sub>	Z <sub>n+1</sub>	I <sub>n</sub>	a <sub>n</sub>
0	0	Z <sub>0</sub>	Z <sub>1</sub>	1	0
0	1	Z <sub>1</sub>	Z <sub>0</sub>	0	0
1	0	Z <sub>0</sub>	Z <sub>0</sub>	0	0
1	1	Z <sub>1</sub>	Z <sub>1</sub>	1	1



Die obige Gleichung bedeutet, dass das Ergebnis aus der Tabelle direkt 1:1 übernommen werden kann, denn  $Q^{n+1} = D^n = I^n$ .

# Vereinfachung über SN-Tabelle

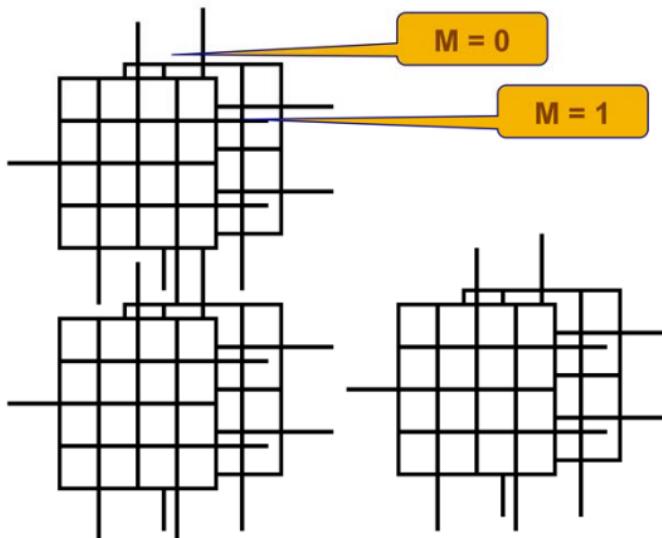
## Punkt 5 bei der Vorgehensweise

= Erstellung der "optimalen" Gleichung

bei

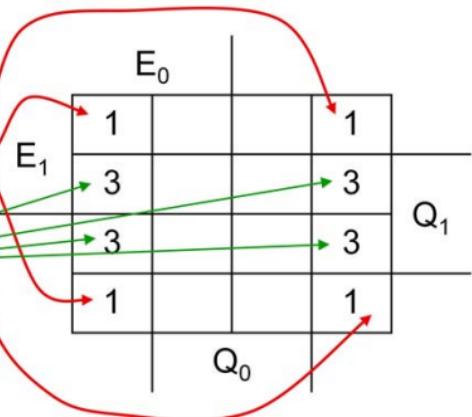
5 Eingangsgrößen:                    2\*4er-Tabelle

und 3 Ausgangsgrößen:                3 \* (2\*4er-Tabelle)

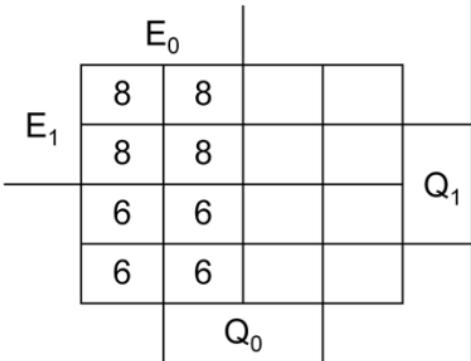


$$Q_0^{n+1} : M = 0$$

M <sup>n</sup>	E <sub>1</sub> <sup>n</sup>	E <sub>0</sub> <sup>n</sup>	Q <sub>1</sub> <sup>n</sup>	Q <sub>0</sub> <sup>n</sup>	Q <sub>1</sub> <sup>n+1</sup>	Q <sub>0</sub> <sup>n+1</sup>	C <sup>n</sup>
0	x	x	0	0	0	1	0
0	x	x	0	1	1	0	0
0	x	x	1	0	1	1	0
0	x	x	1	1	0	0	1
1	0	0	x	x	0	0	0
1	0	1	x	x	0	1	0
1	1	0	x	x	1	0	0
1	1	1	x	x	1	1	0



$$Q_0^{n+1} : M = 1$$



*Einfachere  
Möglichkeit?*

$M^n$	$E_1^n$	$E_0^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	$C^n$
0	x	x	0	0	0	1	0
0	x	x	0	1	1	0	0
0	x	x	1	0	1	1	0
0	x	x	1	1	0	0	1
1	0	0	x	x	0	0	0
1	0	1	x	x	0	1	0
1	1	0	x	x	1	0	0
1	1	1	x	x	1	1	0

$$Q_0^{n+1} : M = 0$$

$$Q_0^{n+1} : M = 1$$

$Q_0$	
$Q_1$	3
	1

$E_0$	
$E_1$	8
	6

$$Q_0^{n+1} = \overline{M^n Q_0^n} \vee M^n E_0^n$$

$$Q_1^{n+1} : M = 0$$

$$Q_1^{n+1} : M = 1$$

$Q_0$	
$Q_1$	3
	2

$E_0$	
$E_1$	8
	7

$$\begin{aligned} Q_1^{n+1} &= \overline{M^n} (Q_1^n \overline{Q_0^n} \vee \overline{Q_1^n} Q_0^n) \vee M^n E_1^n \\ &= \overline{M^n} (Q_1^n \oplus Q_0^n) \vee M^n E_1^n \end{aligned}$$

Antivalenz bzw. Exklusiv-ODER

$M^n$	$E_1^n$	$E_0^n$	$Q_1^n$	$Q_0^n$	$Q_1^{n+1}$	$Q_0^{n+1}$	$C^n$
0	x	x	0	0	0	1	0
0	x	x	0	1	1	0	0
0	x	x	1	0	1	1	0
0	x	x	1	1	0	0	1
1	0	0	x	x	0	0	0
1	0	1	x	x	0	1	0
1	1	0	x	x	1	0	0
1	1	1	x	x	1	1	0

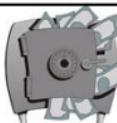
direkt abzulesen:

$$C^n = \overline{M^n} Q_1^n Q_0^n$$

$$Q_0^{n+1} = \overline{M^n} \overline{Q_0^n} \vee M^n E_0^n$$

$$\begin{aligned} Q_1^{n+1} &= \overline{M^n} (Q_1^n \overline{Q_0^n} \vee \overline{Q_1^n} Q_0^n) \vee M^n E_1^n \\ &= \overline{M^n} (Q_1^n \oplus Q_0^n) \vee M^n E_1^n \end{aligned}$$

## Ergebnis für D-FFs



# Das gleiche Beispiel mit JK-FFs

(z. B.: SN 7476)

Charakteristische Gleichung:

$$Q^{n+1} = J^n \overline{Q^n} \vee \overline{K^n} Q^n$$

*Direkte Zuordnung wie beim DD-FF nicht gegeben:*

$$\vec{A}^n = \beta(\vec{E}^n, \vec{Q}^n) := \text{Ausgangsfunktion}$$

$$\vec{Q}^{n+1} = \alpha_1(\vec{I}^n, \vec{Q}^n) := \text{Übergangsfunktion}$$

$$\vec{I}^n = \alpha_2(\vec{E}^n, \vec{Q}^n) := \text{Erregungsfunktion}$$

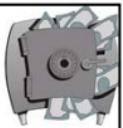
gesucht:

$$\vec{I}^n = \alpha_1(J_1^n, K_1^n, J_0^n, K_0^n, Q_1^n, Q_0^n)$$

gegeben:

$$Q_1^{n+1} = J_1^n \overline{Q_1^n} \vee \overline{K_1^n} Q_1^n$$

$$Q_0^{n+1} = J_0^n \overline{Q_0^n} \vee \overline{K_0^n} Q_0^n$$



KV-Diagramme identisch zur D-FF-Lösung:

$$Q_0^{n+1} = \overline{M^n} Q_0^n \vee M^n E_0^n$$

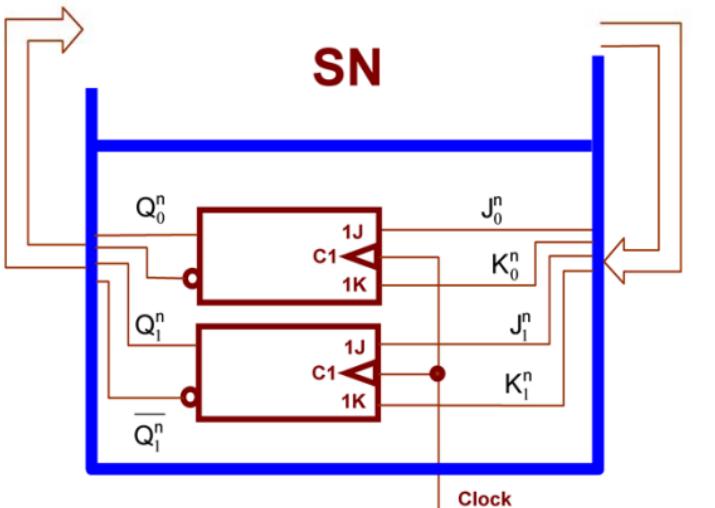
$$\begin{aligned} Q_1^{n+1} &= \overline{M^n} (\overline{Q_1^n} \overline{Q_0^n} \vee \overline{Q_1^n} Q_0^n) \vee M^n E_1^n \\ &= \overline{M^n} (Q_1^n \oplus Q_0^n) \vee M^n E_1^n \end{aligned}$$

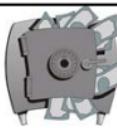
Aufgabe ist,  
dies  
aufzusplitten  
in:

FF-Teil

$$\vec{Q}^{n+1} = \alpha_1(\vec{I}^n, \vec{Q}^n)$$

$$\vec{I}^n = \alpha_2(E^n, \vec{Q}^n)$$





## 1. Weg: Koeffizientenvergleich

$$\begin{aligned} Q_0^{n+1} &= \overline{\overline{M^n}} \overline{Q_0^n} \vee M^n E_0^n \\ &= \overline{\overline{M^n}} \overline{Q_0^n} \vee M^n E_0^n (Q_0^n \vee \overline{Q_0^n}) \\ &= \overline{\overline{M^n}} \overline{Q_0^n} \vee M^n E_0^n Q_0^n \vee M^n E_0^n \overline{Q_0^n} \\ Q_0^{n+1} &= (\overline{\overline{M^n}} \vee M^n E_0^n) \overline{Q_0^n} \vee M^n E_0^n Q_0^n \end{aligned}$$

$J_0^n$                        $\overline{K}_0^n$

Charakteristische GI.:

$$Q_0^{n+1} = J_0^n \overline{Q_0^n} \vee \overline{K}_0^n Q_0^n$$

FF-Teil

$$\begin{aligned} J_0^n &= \overline{\overline{M^n}} \vee M^n E_0^n \\ K_0^n &= \overline{\overline{M^n}} \vee \overline{E_0^n} \end{aligned}$$

SN-Teil

## 1. Weg: Koeffizientenvergleich für $Q_1^{n+1}$ :

$$Q_1^{n+1} = \overline{M^n}(Q_1^n \oplus Q_0^n) \vee M^n E_1^n$$

$$Q_1^{n+1} = \overline{M^n} \overline{Q_1^n} \overline{Q_0^n} \vee \overline{M^n} \overline{Q_1^n} Q_0^n \vee M^n E_1^n$$

$$Q_1^{n+1} = (\overline{M^n} Q_0^n \vee M^n E_1^n) \overline{Q_1^n} \vee (\overline{M^n} Q_0^n \vee M^n E_1^n) Q_1^n$$

$$J_1^n$$

$$\overline{K}_1^n$$

Charakteristische Gl.:

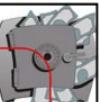
$$Q_0^{n+1} = J_0^n \overline{Q_0^n} \vee \overline{K_0^n} Q_0^n$$

FF-Teil

$$J_1^n = \overline{M^n} Q_0^n \vee M^n E_1^n$$

$$K_1^n = (M^n \vee Q_0^n)(\overline{M^n} \vee \overline{E_1^n})$$

SN-Teil

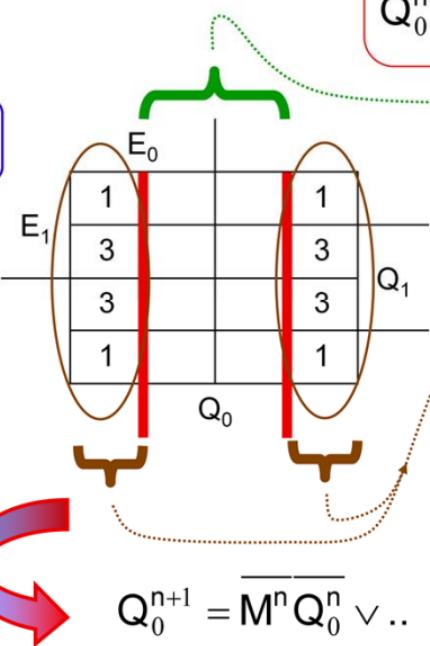


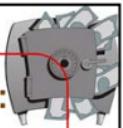
## 2. Weg: aus dem KV-Diagramm

Charakteristische Gl.:

$$Q_0^{n+1} = \overline{J_0^n Q_0^n} \vee \overline{K_0^n Q_0^n}$$

$$Q_0^{n+1} : M^n = 0$$



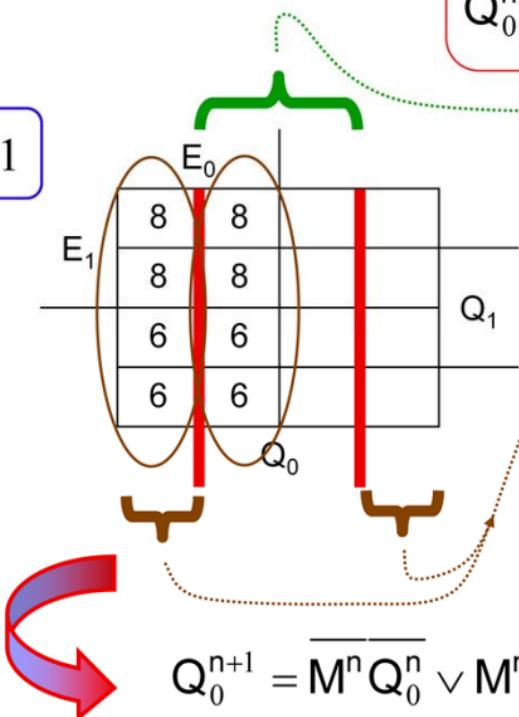


Für  $M^n = 0$  und  $M^n = 1$ :

Charakteristische Gl.:

$$Q_0^{n+1} = \overline{J_0^n Q_0^n} \vee \overline{K_0^n Q_0^n}$$

$Q_0^{n+1} : M^n = 1$



$$Q_0^{n+1} = \overline{M^n Q_0^n} \vee M^n E_0^n \overline{Q_0^n} \vee M^n E_0^n Q_0^n$$



$$Q_0^{n+1} = \overline{M^n Q_0^n} \vee M^n E_0^n \overline{Q_0^n} \vee M^n E_0^n Q_0^n$$

$$Q_0^{n+1} = (\overline{M^n} \vee M^n E_0^n) Q_0^n \vee M^n E_0^n Q_0^n$$

 $J_0^n$  $\overline{K}_0^n$ 

.. und für  $Q_1$  identischer Ablauf ..

**usw.**

## .. mit RS-FFs

Charakteristisches Gleichungssystem:

$$Q^{n+1} = S^n \vee R^n Q^n$$

NB:  $S^n R^n = 0$

gesucht: I-Terme:

$$I^n = \alpha_2(R_2^n, S_2^n, R_1^n, S_1^n, Q_2^n, Q_1^n)$$

Die gleiche Vorgehensweise wie beim D- & JK-FF **NICHT** zu empfehlen!

**Warum?**

Wie kann die NB indirekt mit eingebunden werden?

Das dritte Beispiel hier wäre die Realisierung mit RS-FFs. Oben sind dazu die prinzipiellen Gleichungen aufgeführt, die eingangs entwickelt wurden. Auffallend ist natürlich die NB (Nebenbedingung). Was bedeutet das? Das Gleichungssystem  $Q^{n+1} = \dots$  wird im Allgemeinen verschiedene Lösungen bieten, die dann mit der NB geprüft werden muss, ob sie diese erfüllt. Erfüllt sie die Nebenbedingung nicht, bedeutet es, dass die Lösung nicht brauchbar ist. Das (also die Berechnung über implizite Gleichungen) macht das Ganze umständlich und verlangt einen anderen Weg, der im Folgenden erklärt werden soll.

Aber was sind implizite Gleichungen? Dazu die nächste Folie.

**Notwendig wäre:****Lösung einer impliziten Booleschen Gleichung**

auf eine Vereinfachung soll verzichtet werden

**Triviales Beispiel:**  $(a \vee b)(a \vee \bar{b}) = 1$ **Ansatz:**  $(a \vee b)(a \vee \bar{b}) = c$ **Lösung über Tabelle:**

a	b	c	1	
0	0	0	1	$c \neq 1$
0	1	0	1	$c \neq 1$
1	0	1	1	$c = 1$
1	1	1	1	$c = 1$

$\Rightarrow a = c$

Ein triviales Beispiel einer impliziten Gleichung ist oben dargestellt. Es ist eine Lösung, bei der die gesuchte Variable nicht aufgelöst, also nicht  $a = ..$ , schon da steht, sondern das Ergebnis  $a = ..$  gesucht wird. Wie kann man nun eine implizite Gleichung lösen? Oben wird die Methode über eine Tabelle dargestellt.

## Implizite Gleichungen

Aus der Tabelle zum Lösungsansatz geht hervor:

$$c = a$$

da aber nach:

$$(a \vee b)(a \vee \bar{b}) = 1$$

$$(a \vee b)(a \vee \bar{b}) = c$$

$$c = 1 \quad \text{ist:} \quad a = 1,$$

a	b	c	1	
0	0	0	1	$c \neq 1$
0	1	0	1	$c \neq 1$
1	0	1	1	$c = 1$
1	1	1	1	$c = 1$

was eine eindeutige Lösung darstellt.

## Implizite Gleichungen

$c = a = 1$  kann auch in **DIESEM** simplen Fall direkt über Vereinfachung ermittelt werden:

$$(a \vee b)(a \vee \bar{b}) = 1$$

$$\overline{(a \vee b)(a \vee \bar{b})} = \bar{1}$$

$$\bar{a}\bar{b} \vee \bar{a}b = 0$$

$$\bar{a} = 0$$

$$a = 1$$



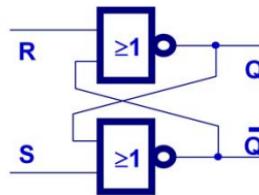
Nicht immer sind die Gleichungen so trivial:

$$a \wedge \bar{a} = 1 \longrightarrow \text{falsch}$$

$$a \wedge b = 0 \longrightarrow \text{Mehrfachlösung}$$

## .. mit RS-FFs

$S^n$	$R^n$	$Q^n$	$Q^{n+1}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

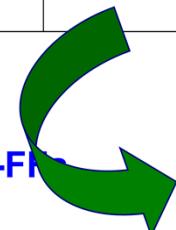


$Q^n \rightarrow Q^{n+1}$	$S^n$	$R^n$
0 $\rightarrow$ 0	0	x
0 $\rightarrow$ 1	1	0
1 $\rightarrow$ 0	0	1
1 $\rightarrow$ 1	x	0

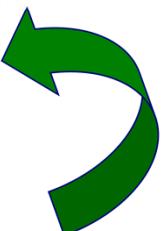
Die linke Tabelle ist die vollständige Tabelle des RS-FFs. Die rechte Tabelle ist eine vereinfachte Darstellung. Das bedeutet, die linke Tabelle kann in die rechte überführt werden und umgekehrt.

$M^n E_1^n E_0^n$	$Q_1^n Q_0^n$	$Q_1^{n+1} Q_0^{n+1}$	$S_1^n R_1^n$	$S_0^n R_0^n$
0 x x	0 0	0 1	0 x	1 0
0 x x	0 1	1 0	1 0	0 1
0 x x	1 0	1 1	x 0	1 0
0 x x	1 1	0 0	0 1	0 1
1 0 0	x x	0 0	0 1	0 1
1 0 1	x x	0 1	0 1	1 0
1 1 0	x x	1 0	1 0	0 1
1 1 1	x x	1 1	1 0	1 0

.. mit RS-FFs



$Q^n \rightarrow Q^{n+1}$	$S^n$	$R^n$
0 → 0	0	x
0 → 1	1	0
1 → 0	0	1
1 → 1	x	0



Die 7 linken Spalten der oberen Tabelle sind ja das Ergebnis der ursprünglichen Aufgabe (siehe erstes dargestellt Beispiel mit D-FFs).

Gezeigt wird hier, wie man das Berechnen über die implizite Gleichung vermeiden kann. Man geht einen Schritt zurück, also man verwendet als erstes nicht das Gleichungssystem, sondern die ursprünglichen Tabellen, über die ja die Gleichungen des RS-FFs entwickelt wurden. Die ursprüngliche Tabelle erweitert man um die vier rechten Spalten S und R, die man sich über die untere Tabelle entwickeln kann, was die grünen Pfeile zeigen sollen.

.. mit  
RS-FFs

$M^n E_1^n E_0^n$	$Q_1^n Q_0^n$	$Q_1^{n+1} Q_0^{n+1}$	$S_1^n R_1^n$	$S_0^n R_0^n$
0 x x	0 0	0 1	0 x	1 0
0 x x	0 1	1 0	1 0	0 1
0 x x	1 0	1 1	x 0	1 0
0 x x	1 1	0 0	0 1	0 1
1 0 0	x x	0 0	0 1	0 1
1 0 1	x x	0 1	0 1	1 0
1 1 0	x x	1 0	1 0	0 1
1 1 1	x x	1 1	1 0	1 0

$$\begin{aligned}
 I_0^n = S_0^n &= \overline{M^n Q_1^n Q_0^n} \vee \overline{M^n Q_1^n \overline{Q_0^n}} \vee M^n \overline{E_1^n} E_0^n \vee M^n E_1^n E_0^n \\
 &= \overline{M^n Q_0^n} \vee M^n E_0^n
 \end{aligned}$$

Über 4 rechten Spalten können dann die endgültigen Gleichungen entwickelt werden.

## .. mit RS-FFs

*vollständiges Ergebnis:*

$$I_0^n = S_0^n = \overline{M^n Q_0^n} \vee M^n E_0^n$$

$$I_1^n = R_0^n = \overline{M^n Q_0^n} \vee M^n \overline{E_0^n}$$

$$I_2^n = S_1^n = \overline{M^n Q_1^n Q_0^n} \vee M^n E_1^n$$

$$I_3^n = R_1^n = \overline{M^n Q_1^n Q_0^n} \vee M^n \overline{E_1^n}$$

*Dieses Verfahren eignet sich für ALLE FFs.*

*Man kann es allerdings im allgemeinen nur beim RS-FF  
als effizient bezeichnen.*



# Asynchrone Schaltungen

***Wir betrachten bzgl. digitaler Schaltungen vor allem:***

## Races

*erwünschte, unerwünschte, critical, noncritical, ..*

## Hazards

*statische, dynamische, kombinatorische, essential H., ..*

## Spikes



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K3-2  
p. 36  
18.01.2013 13:23:59

Races, Hazards und Spikes sind zeitabhängige Effekte in logischen Schaltungen. Sie fallen damit in den Bereich asynchroner Schaltwerke. Es sind Effekte, die zum Teil gewünscht sind, aber auch zur Beeinträchtigung der Schaltungsfunktion bis hin zu ihrer Unbrauchbarkeit führen können. Das Wissen um diese Effekte ist dabei notwendige Voraussetzung für einen Entwickler synchroner und asynchroner Schaltwerke.

Man muss stets daran denken, dass synchrone Schaltwerke eine sehr vereinfachte Modelldarstellung der Realität sind.

### Races

Ausgegangen werden soll nochmals von dem komplexen Zustandsdiagramm des RS-FFs in Kap.3, Teil 1 der Folien. Greift man die Zustandsübergänge

$SR/Q1Q0 = 00/00$  und  $SR/Q1Q0 = 00/11$

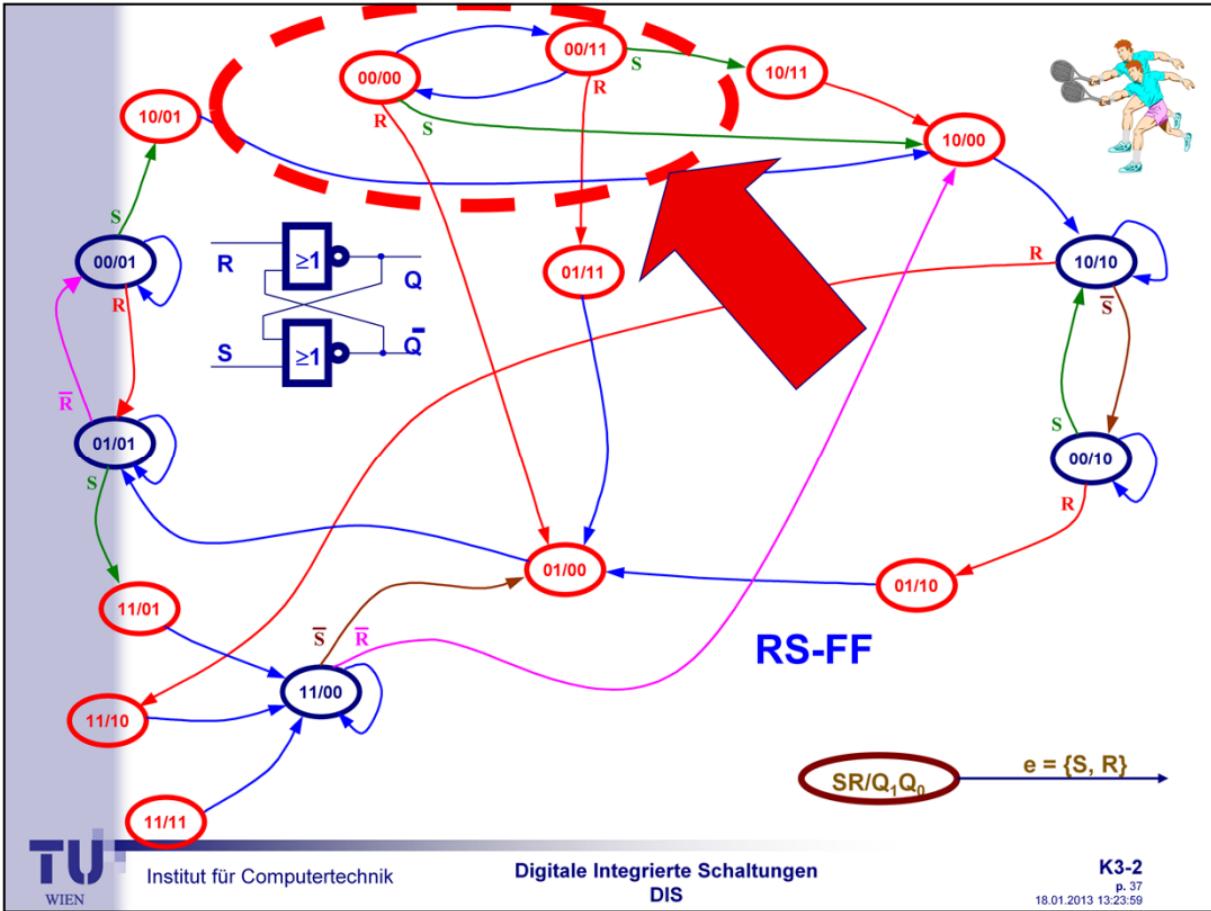
heraus, fallen zwei Dinge auf:

#### 1. Die Zustandsübergänge

$$\begin{array}{ccc} 00/00 & \Rightarrow & 00/11 \\ 00/11 & \Rightarrow & 00/00 \end{array}$$

sind physikalisch kaum möglich, weshalb sie in auf der übernächsten Folie auch gestrichelt dargestellt sind.

Man muss stets daran denken, dass synchrone Schaltwerke eine sehr vereinfachte Modelldarstellung der Realität sind.



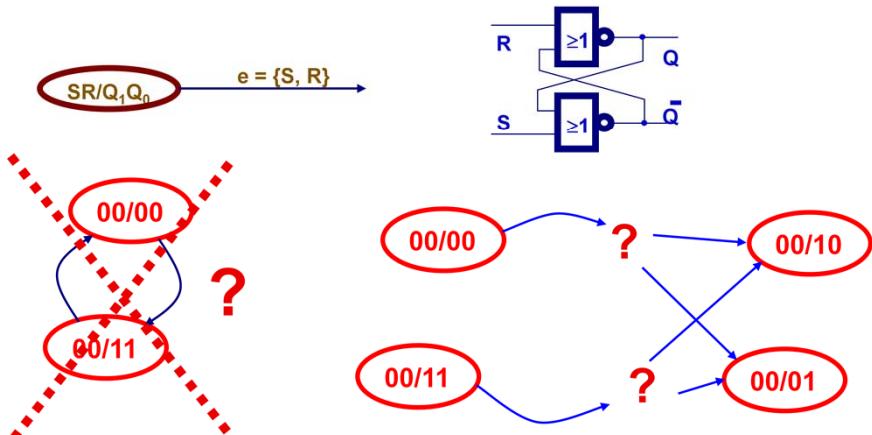
Dieser Teil ist auf der nächsten Folie herausgearbeitet.

# Races

(Laufzeiteffekte)



erwünschte - unerwünschte



Ausgegangen werden soll nochmals von dem komplexen Zustandsdiagramm des RS-FFs in Kap.3, Teil 1 der Folien. Greift man die Zustandsübergänge

$$SR/Q_1Q_0 = 00/00 \text{ und } SR/Q_1Q_0 = 00/11$$

heraus, fallen zwei Dinge auf:

1. Die Zustandsübergänge

$$\begin{array}{lll} 00/00 & \Rightarrow & 00/11 \\ 00/11 & \Rightarrow & 00/00 \end{array}$$

sind physikalisch kaum möglich, weshalb sie in auf der übernächsten Folie auch gestrichelt dargestellt sind.

Man muss stets daran denken, dass synchrone Schaltwerke eine sehr vereinfachte Modelldarstellung der Realität sind.

Vom Zustand 00/00 in den Zustand 00/11 wird sehr kurz verlaufen, dann wird das System in 00/01 oder 00/10 überwechseln, je nachdem welcher Transistor schneller ist.

Es bleibt bei derartigen Übergängen dem Zufall überlassen, ob das System von

$$\begin{array}{lll} 00/00 & \Rightarrow & 00/01 \\ 00/00 & \Rightarrow & 00/10 \\ 00/11 & \Rightarrow & 00/01 \\ 00/11 & \Rightarrow & 00/10 \end{array}$$

springt.

Der Grund für diesen Effekt ist in den unterschiedlichen Verzögerungszeiten der beiden Transistoren im RS-FF zu suchen. Korrigiert man aufgrund dieser Überlegungen das bisherige komplexe Bild des RS-FFs,

weiter unten nochmals eingegangen wird.



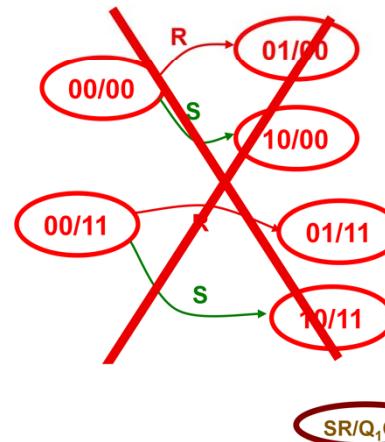
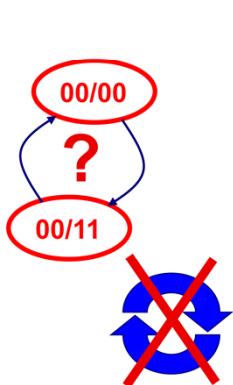
# Erwünschtes "Wettrennen"

(desired Race)

Vorherige Modellbeschreibung unzureichend.

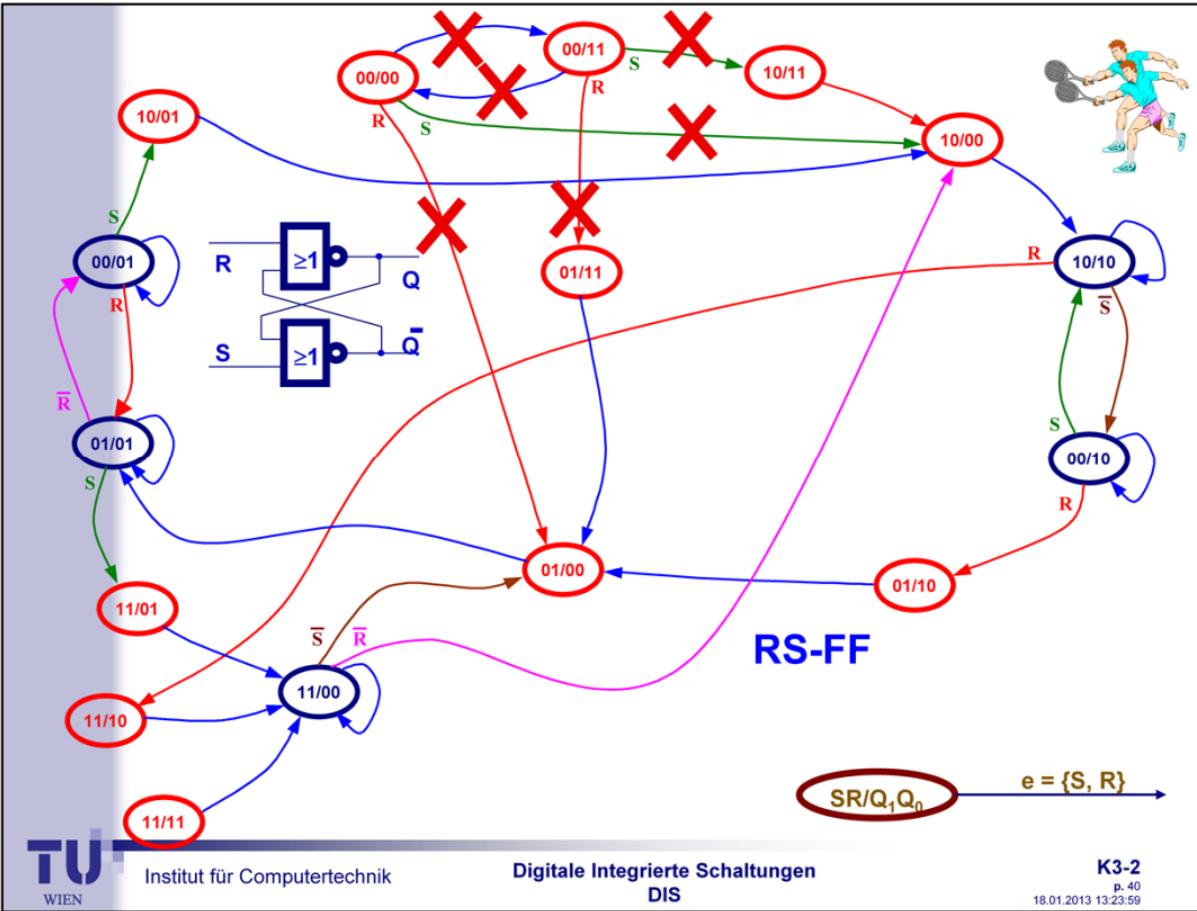
Totzeiten in den Transistoren sind entscheidend:

$$t_0 \leftrightarrow t_1$$

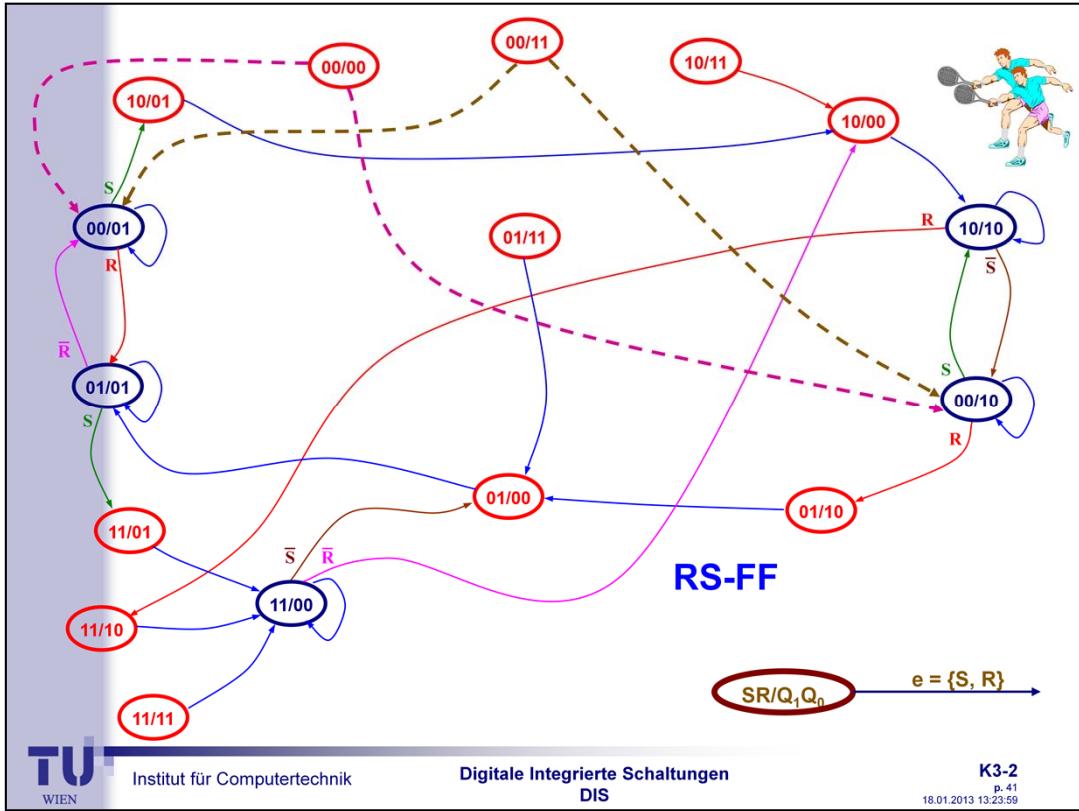


Linkes Bild: eine Kreisbewegung ist physikalisch nicht möglich.

Rechtes Bild: ein Durchlauf über R oder S ist nicht möglich bzw. sollte nicht erfolgen. In einem instabilen Zustand darf ein Eingang nicht geschaltet werden, da sonst das System nicht mehr beschreibbar ist.



Das sind physikalisch nicht mögliche Übergänge.



Im Falle des RS-FFs spricht man von einem erwünschten Race-Effekt. Im Gegensatz dazu gibt es aber auch unerwünschte Race-Effekte. Angenommen sei eine zu entwickelnde Schaltung auf der Basis des Zustandsdiagramms nach Bild 3.28.

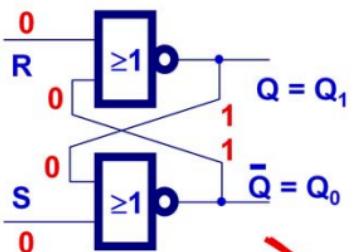
[1] Dargestellt sind wiederum die totalen Zustnde des FFs, also die Menge aller mglichen Kombinationen der Eingangszustnde und der inneren Zustnde.

# RS-Übergangsfolge

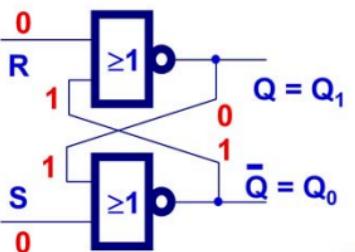
$e = \{S, R\}$



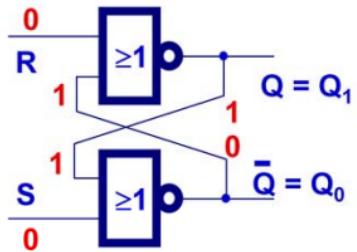
00/00 → 00/11



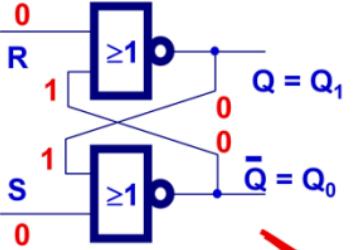
→ 00/01



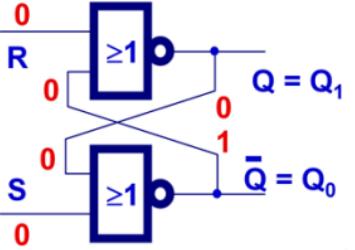
∨ 00/10



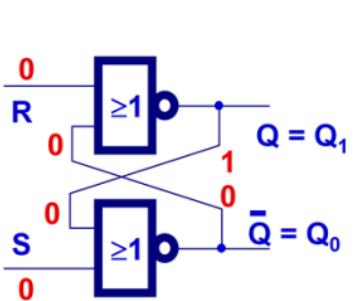
00/11 → 00/00



→ 00/01



∨ 00/10

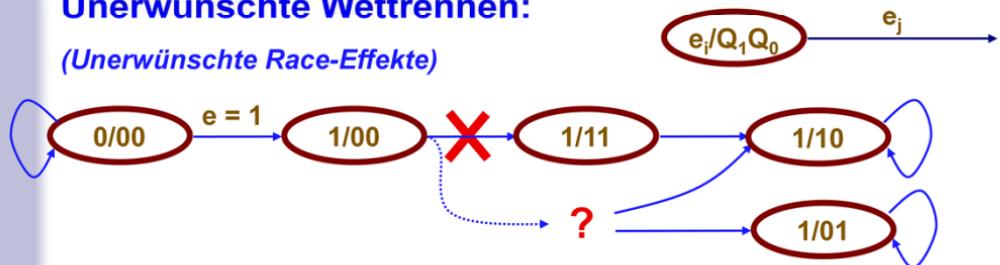


= erwünschtes Wettrennen (desired Race)

*Trotzdem, Problem:*

Durchlauf eines instabilen Zustandes kostet Laufzeit.

**Unerwünschte Wettrennen:**  
*(Unerwünschte Race-Effekte)*



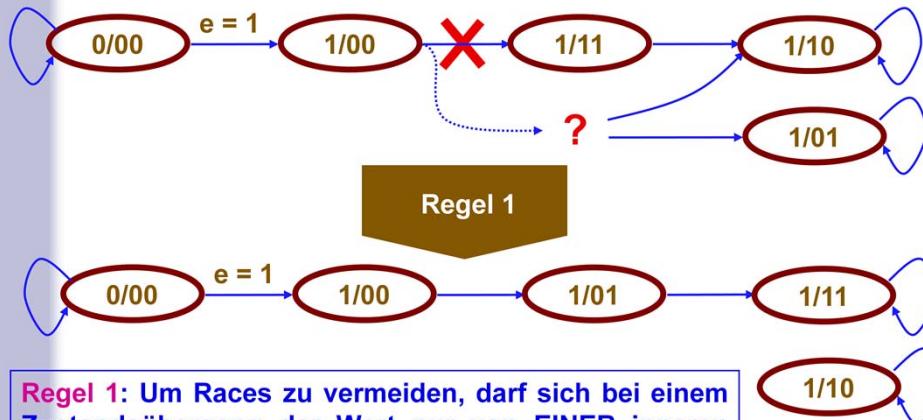
Angenommen sei eine zu entwickelnde Schaltung auf der Basis des Zustandsdiagramms nach Bild oben.

Dargestellt sind wiederum die totalen Zustände des FFs, also die Menge aller möglichen Kombinationen der Eingangszustände und der inneren Zustände.

## Unerwünschte Wettrennen:



Regel 1



**Regel 1:** Um Races zu vermeiden, darf sich bei einem Zustandsübergang der Wert nur von EINER inneren Zustandsvariablen ändern!

Der Zustandsübergang

1/00                     $\Rightarrow$                     1/11

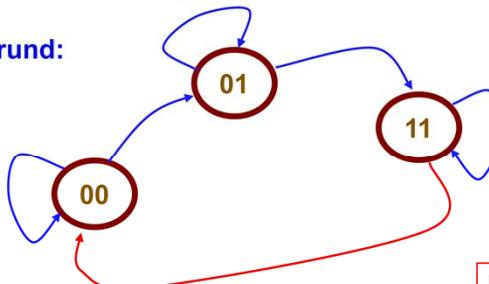
ist physikalisch kaum möglich. Das System wird sich einen anderen Weg (wo das Fragezeichen eingetragen ist) suchen und den Zustand 1/11 umgehen, was aber der gewünschten Funktion nicht entspricht. Gewünscht ist eine Zustandsfolge ausgehend von einem stabilen Zustand, überleitend über zwei instabile Zustände und einmündend wiederum in einen stabilen Zustand. Der unerwünschte Race-Effekt kann demgemäß leicht durch eine Umcodierung behoben werden, wie es Bild unten zeigt. Für die Entwicklung asynchroner Schaltwerke lässt sich daraus eine wichtige Regel ableiten – siehe oben.

Die Voraussetzung hierfür ist die *gerade* Anzahl von Zuständen in einer Zustandsschleife. Bei einer ungeraden Anzahl von Zuständen in einer Zustandsschleife müssen sich mindestens einmal mindestens 2 Stellen des binären Formates ändern.

## Regel 2

**Regel 2:** Bei asynchronen Schaltwerken kann nur eine GERADE Anzahl von Zuständen in EINER Schleife zugelassen werden.

Grund:



RACE-Effekt:

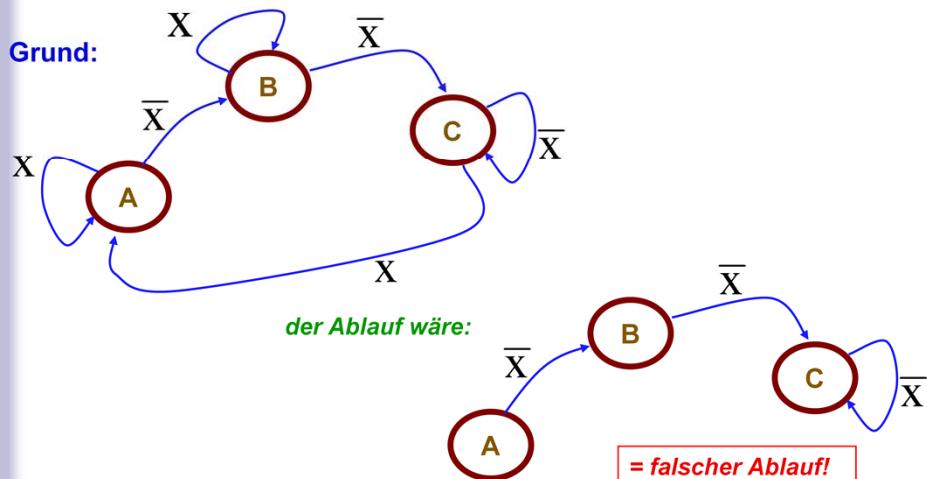
- 2 bits müssen gleichzeitigkippen!
- einschrittigen Code verwenden

Das führt zu Regel 2.

Hat man also eine Zustandskonfiguration nach Bild oben, kann aus den obigen Überlegungen geschlossen werden, dass die Realisierung eines solchen Grafen nicht möglich ist. In diesem Fall müsste ein vierter Zustand hinzugefügt werden, der nicht in jedem Fall stabil sein muss.

## Regel 3

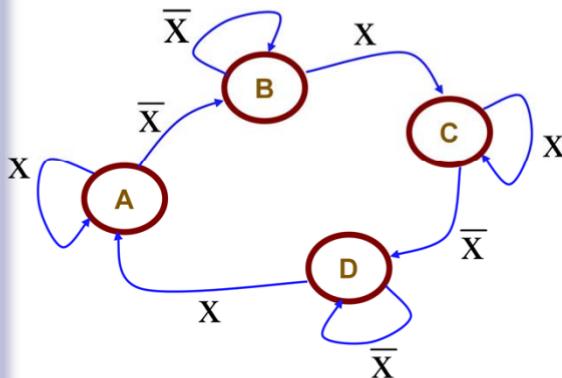
Außerdem ist auf e zu achten:



Die Struktur nach Bild oben beinhaltet noch einen zweiten entscheidenden Fehler. Wechselt das System vom Zustand A in den Zustand B, in dem die Eingangsvariable von  $x$  nach  $\neg x$  gesetzt wird, sollte das System im Zustand B (stabiler Zustand!) verharren. Dies ist aber nur möglich, wenn nach Verlassen von A durch den Wechsel von der Eingang schnell wieder auf  $x$  wechselt, damit ein Haltemechanismus greift. In Bild oben wird dieser Aspekt nochmals herausgearbeitet.

## Regel 3

*.. richtig ist, um die Synchronisation zwischen a und B herzustellen:*



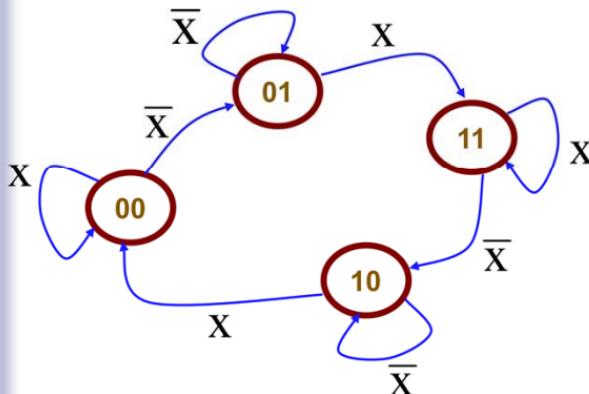
**Regel 3:** Bei asynchronen Schaltwerken ist bei einem Zustandswechsel ein HALTEMECHANISMUS zu beachten.

Eine technisch vernünftige Lösung ist in diesem Fall kaum zu erzielen, außer man führt einen weiteren Knoten ein: Bild oben.

Das führt zur Regel 3.

## Regel 2

*richtig ist, um die Synchronisation zwischen e und B herzustellen:*

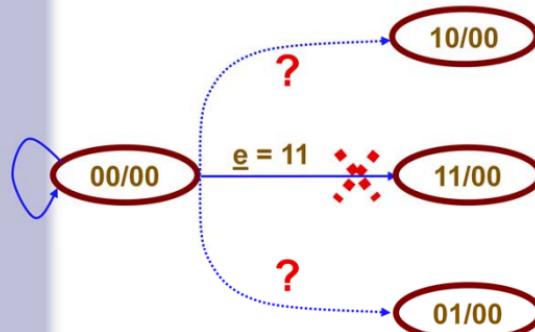


**Regel 2:** Bei asynchronen Schaltwerken kann nur eine GERADE Anzahl von Zuständen in EINER Schleife zugelassen werden.

Verdeutlicht wird dies nochmals durch den richtigen Entwurf nach Bild oben. Hier sind die Anzahl der Zustände gerade, und das System verlässt den Zustand durch Änderung der Eingangsgröße, der gleichzeitig die Haltegröße für den nächsten Zustand darstellt.

## Regel 4

*Entsprechend zu Regel 1 gilt:*



**Regel 4:** Bei asynchronen Schaltwerken darf sich jeweils zwischen 2 stabilen Zuständen nur eine Eingangsvariable ändern.

Zusätzlich kann festgehalten werden, was keiner weiteren Erläuterung bedarf: Was für die inneren Zustandsvariablen zutrifft, gilt auch für die Eingangsvariablen:

Regel 4.

## Races

### Critical Race:

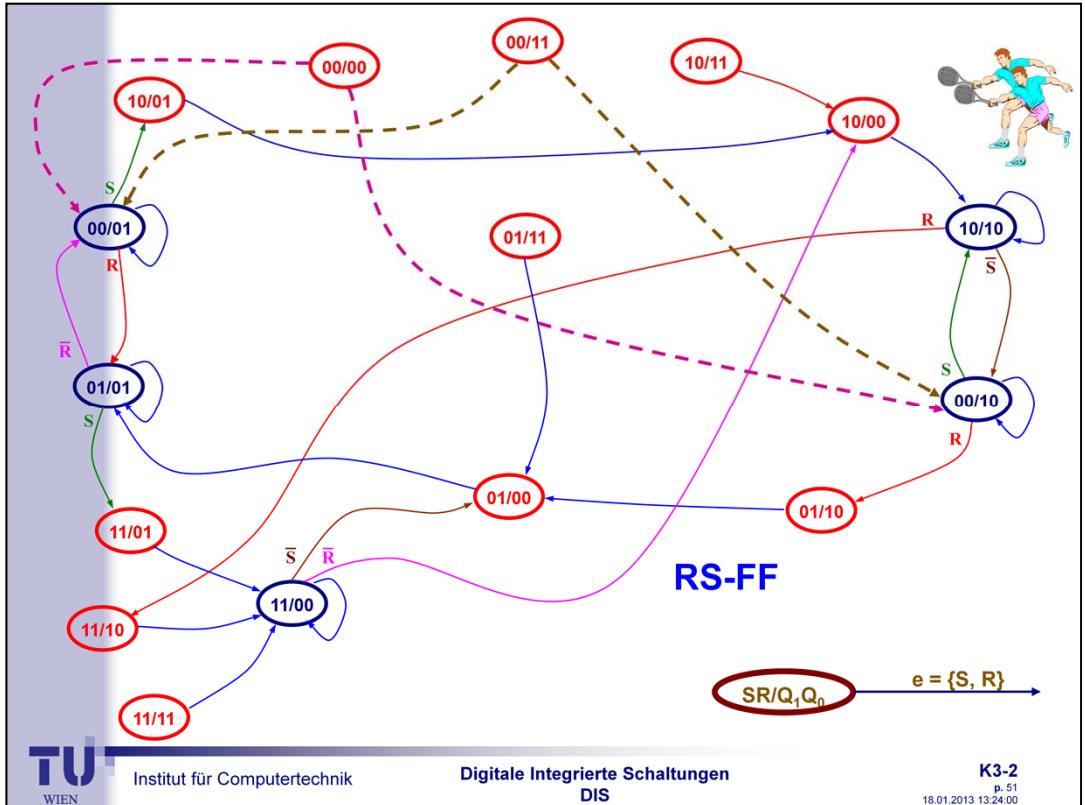
Der erreichte Zustand ist abhängig von der jeweiligen Delay-Time

### Noncritical Race:

Der erreichte Zustand ist unabhängig von der jeweiligen Delay-Time



In der Literatur unterscheidet man zwischen einem "Critical Race" und einem "Non-critical Race". Die bisherigen Feststellungen sind unter dem Sachverhalt des "Critical Race" einzuordnen. Wird bei einem Übergang der Zustand unabhängig von der jeweiligen Delay-Time erreicht, wird von einem "Noncritical Race" gesprochen. Wird dieser bei einer Entwicklung als solcher eindeutig erkannt, muss *keine* Korrektur vorgenommen werden. Doch sollte man bei der Entwicklung prinzipiell versuchen, logische Schaltungen weitgehend unabhängig von der Technologie zu entwerfen, um bei einer Änderung dieser Technologie die Race-Aspekte nicht neu aufrollen zu müssen.



Abschließend soll nochmals kurz auf den komplexen Zustandsgrafen des RS-FFs eingegangen werden. Verwirrend ist die große Menge der Zustände und der Transitionen. Bezüglich der Zustände ist zu berücksichtigen, dass nicht nur die inneren, sondern in diesem Fall auch die äußeren Zustände dargestellt werden, was in synchronen Systemen im allgemeinen nicht notwendig ist. Die Anzahl der gezeigten Transitionen ist sehr hoch, weil die physikalisch möglichen und die nach der entsprechenden Tabelle theoretischen Transitionen wiedergegeben sind.

Zu unterscheiden ist zwischen stabilen und instabilen Zuständen. Von den stabilen Zuständen führen jeweils zwei gerichtete Kanten weg, da zwei Eingangsvariablen definiert sind, aus physikalischen Gründen aber nur *eine* geändert werden darf (die gleichzeitige Aktivierung muss ja ausgeschlossen werden.).

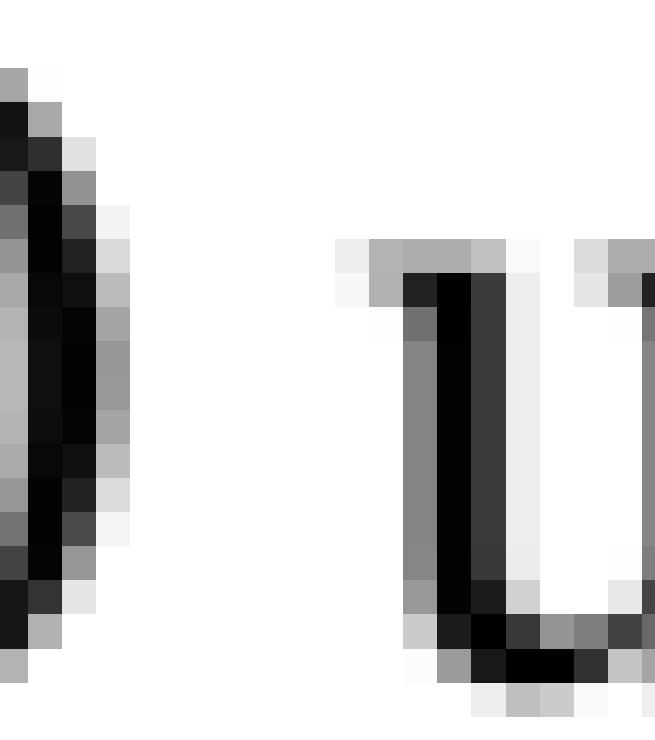
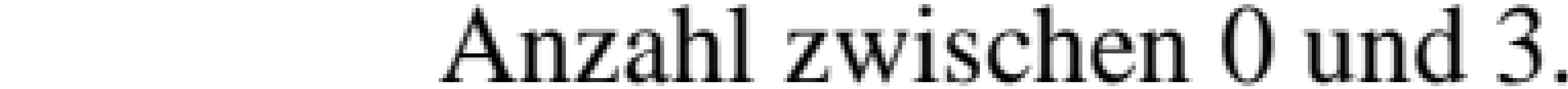
Bei den instabilen Zuständen unterscheidet man zwischen solchen, die nur *einen* weiteren Folgezustand haben, wie beispielsweise der Zustand 01/10 (rechts unten), der direkt in den Zustand 01/00 übergeht, und den Zuständen, die der Race-Problematik unterworfen sind, wie der Zustand 00/00.

Die Übergänge

00/11	→	01/11	durch R,
00/11	→	10/11	durch S,
00/00	→	01/00	durch R,
00/00	→	10/00	durch S,

initiiert, könnten in der Darstellung entfallen, da sie gegen die Regel verstößen. R oder S werden aktiviert, obwohl sich das System in einem instabilen Zustand befindet. Damit stellt sich die Frage, wie beispielsweise der Zustand 00/11 überhaupt zu erreichen ist, da dann keine Kante mehr zu ihm hinführt. Die Antwort ist: Der Einschaltprozess wird durch die Darstellung in Bild oben nicht modelliert. Das bedeutet, jeder in der Tabelle aufgeführte innere und äußere Zustand kann auf physikalischem Wege über den Einschaltprozess erreicht werden, was von vielen Parametern abhängt, die hier nicht näher betrachtet werden sollen.

Letztendlich wird durch dieses Bild deutlich, dass zwischen zwei stabilen Zuständen eine unterschiedliche Anzahl von instabilen Zuständen liegen kann; im vorliegenden Fall eines einfachen RS-FFs variiert die



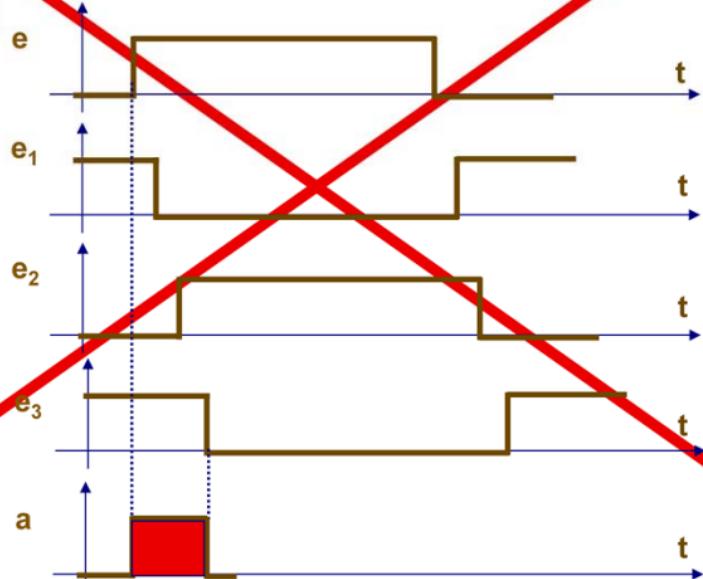
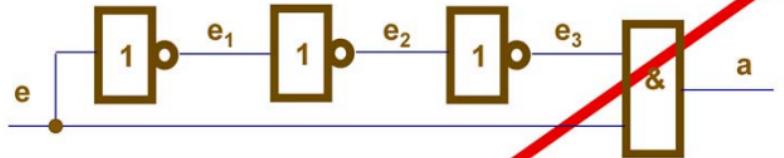
# Hazards

bedeutet:

Gefahr – Zufall ...

(Game of Hazard:  
Glücksspiel)

Nichts  
für  
ASICs!



# Hazards

- Delay technologieabhängig
- mögliche bei Rückkopplungen unerwartete Reaktionen

- Ursachen für Hazards und Races:

**verschieden lange Signalwege**

aber

**Races:**

- mehrere Ausgangsleitungen betroffen  
(siehe RS-FF)

**Hazards:**

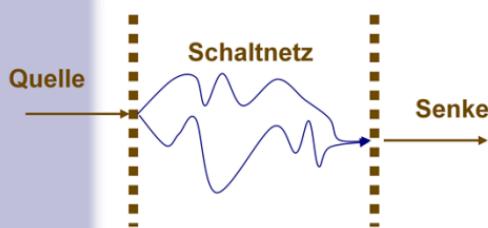
- Quelle und Senke der Signalwege sind identisch

Nichts  
für  
ASICs!

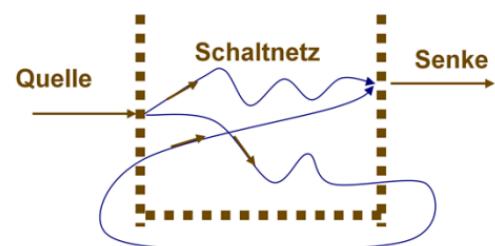
# Hazards

*Unterscheidung zwischen:*

**Kombinatorischer Hazard**



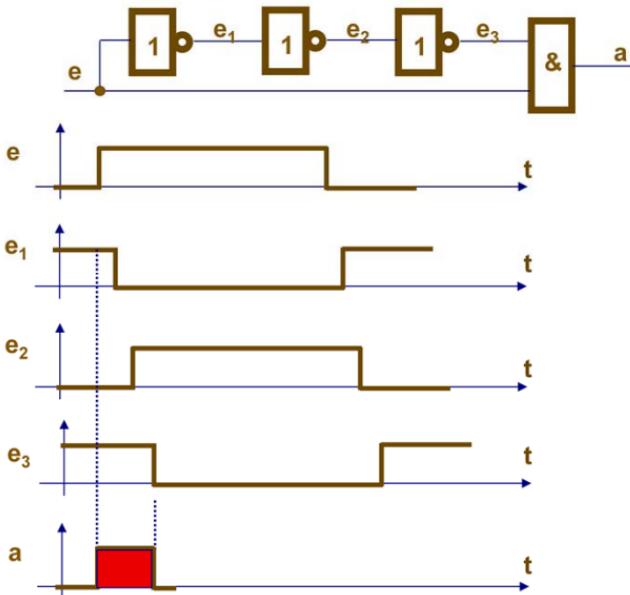
**Essential Hazard  
(Rückkoppung)**



Wie bei Races liegt die Ursache für Hazards in unterschiedlichen Laufzeiten durch verschiedene Signalwege. Während jedoch bei Races jeweils mehrere Ausgangsleitungen betroffen sind, sind bei Hazards die Quelle und die Senke der Signalwege identisch.

# Kombinatorischer Hazard

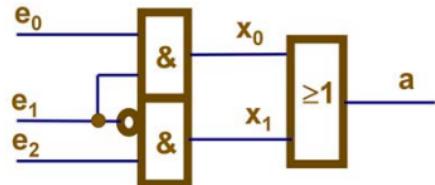
Beispiel 1:



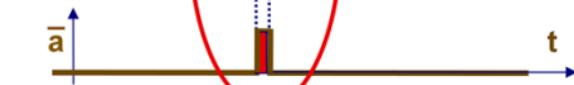
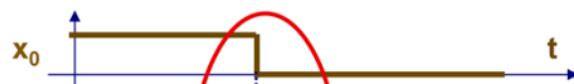
Ein in der Praxis in der "flachen Schaltbrettentwicklung" häufig eingesetztes Verfahren, um Impulse zu generieren, zeigt Bild oben. Die Impulsdauer ist abhängig von der zugrundegelegten Technologie und muss deshalb in der ASIC-Entwicklung prinzipiell vermieden werden.

# Kombinatorischer Hazard

Beispiel 2:



$$a = e_1 e_0 \vee e_2 \bar{e}_1$$



Unterschiedliche Laufzeiten  
können Probleme erzeugen !

Wie kann man das Problem  
beseitigen?

# Kombinatorischer Hazard: Analyse

Ausgangsgleichung:

$$a = e_1 e_0 \vee e_2 \bar{e}_1$$

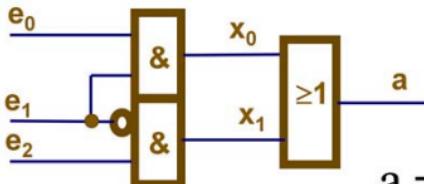
The Karnaugh map shows the function  $a = e_1 e_0 \vee e_2 \bar{e}_1$ . The variables are  $e_2$  (vertical),  $e_1$  (horizontal), and  $e_0$  (depth). The map has four cells: (0,0,0) = 0, (0,1,0) = 1, (1,0,0) = 0, (1,1,0) = 1. Red ovals highlight the minterms 1 and 3. Red arrows point from these ovals to the corresponding rows in the truth table.

$e_2$	$e_1$	$e_0$	$a$
0	0	0	0
0	1	0	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Diesem "erwünschten Hazard" steht der nicht erwünschte gegenüber, der beispielsweise über die Realisierung der Gleichung oben (ungewollt) generiert werden kann. Es stellt sich die Frage, ob er vermeidbar ist. Es gilt: mit Hilfe redundanter Terme ist es grundsätzlich möglich, sofern es sich um synchrone Schaltungen handelt, wobei bezüglich eines Beweises auf die Literatur verwiesen werden muss. Im vorliegenden Fall stellt sich die Lösung recht einfach dar. Betrachtet man das der Schaltung von Bild vorher zugehörige KV-Diagramm (oben), erkennt man, dass durch Hinzufügen des Terms  $e_2 e_0$  der Hazard vermieden werden kann, da dieser Term die "Brücke" zwischen den beiden anderen Termen herstellt. "Brücke" bedeutet hier, dass vom "Umschalten" von einem dieser Terme auf den anderen, dieser dritte Term  $e_2 e_0$  weiterhin eine logische 1 liefern wird.

# Kombinatorischer Hazard: Vermeidung

Problem: wenn  $e_1$  schaltet erfolgt eine Umschaltung von  $x_0$  nach  $x_1$



$$a = e_1 e_0 \vee e_2 \bar{e}_1$$

Lösung:

$$a = e_1 e_0 \vee e_2 \bar{e}_1 \vee e_2 e_0$$

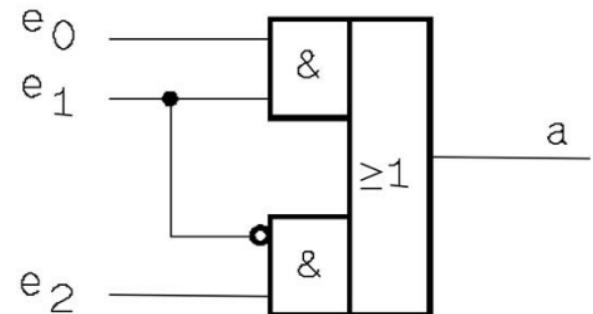
A truth table for the function  $a$  is shown, with inputs  $e_1, e_2, e_0$  and output  $a$ . The table shows the following values:

$e_1$	$e_2$	$e_0$	$a$
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	1

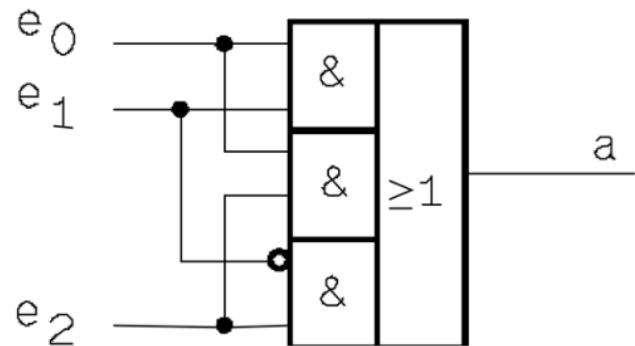
A red oval highlights a hazard at the transition from the state (0, 1, 0) to (1, 0, 1), where the output changes from 1 to 0 despite no single input changing.

# Kombinatorischer Hazard: Vermeidung

$$a = e_1 e_0 \vee e_2 \bar{e}_1$$

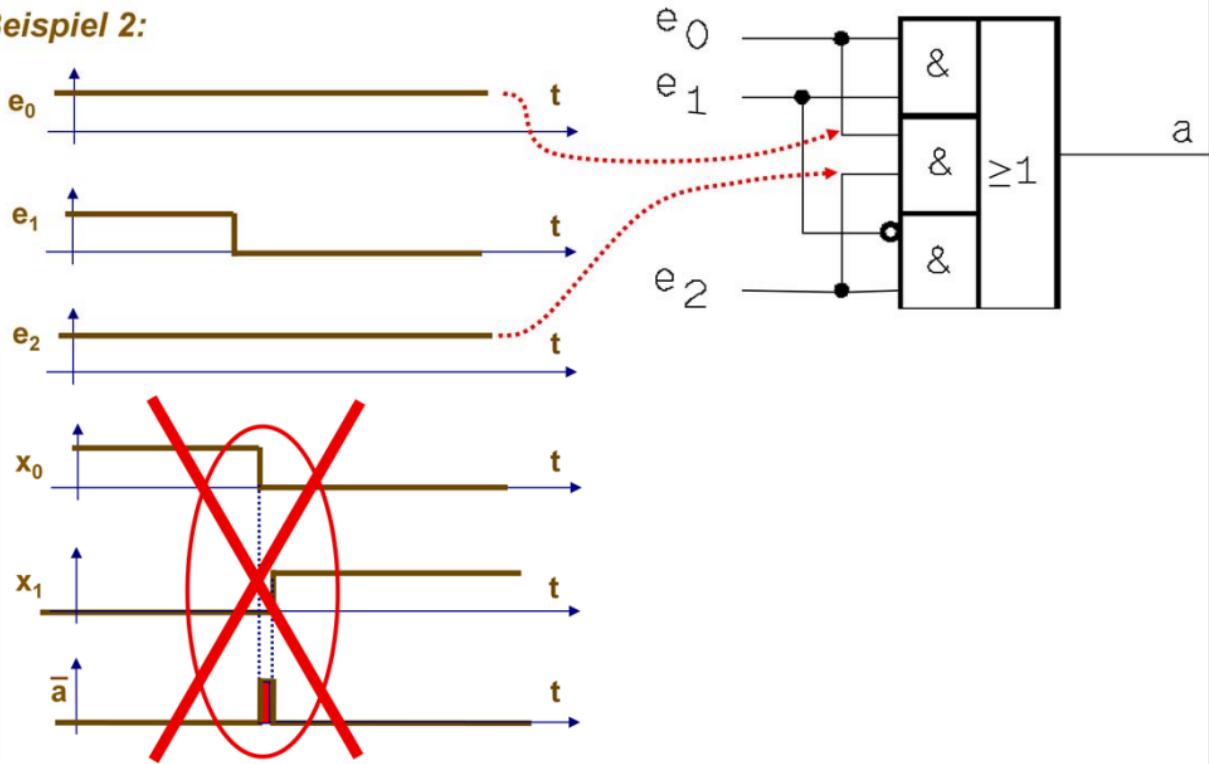


$$a = e_1 e_0 \vee e_2 \bar{e}_1 \vee e_2 e_0$$

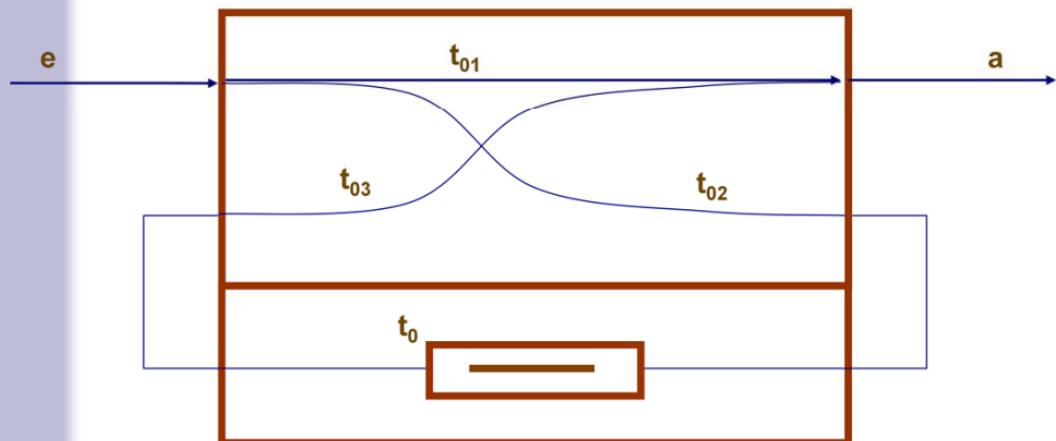
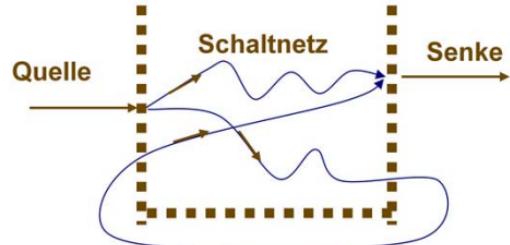


# Kombinatorischer Hazard

Beispiel 2:

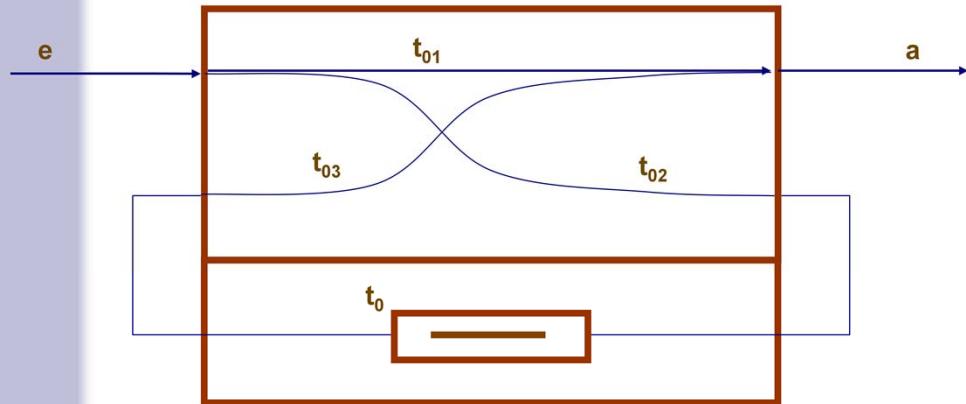


## Essential Hazard



Essential Hazards lassen sich über die Darstellung von Bild oben beschreiben. Nach der Automatentheorie ist das Schaltnetz (oberer Block) unabhängig von der Zeit.

## Essential Hazard



Es muss gelten:

$$t_{01} \ll t_{02} + t_0 + t_{03}$$

sonst Verletzung der Automatentheorie

Auf ein physikalisches System bezogen heißt das, dass stets

$$t_{01} \ll t_{02} + t_0 + t_{03}$$

gelten muss. Ist dies nicht der Fall, versteht es sich nahezu von selbst, dass Effekte (Zustandsübergänge) auftreten können, die nicht erwünscht sind.

# Sonstige Hazards

Weitere übliche Einteilungen:

**statischer Hazard:**

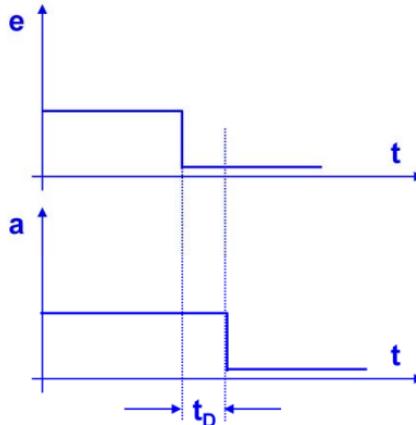
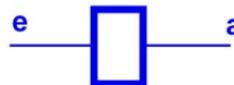


**dynamischer Hazard:**

*(dynamisch überlagerter Übergang)*

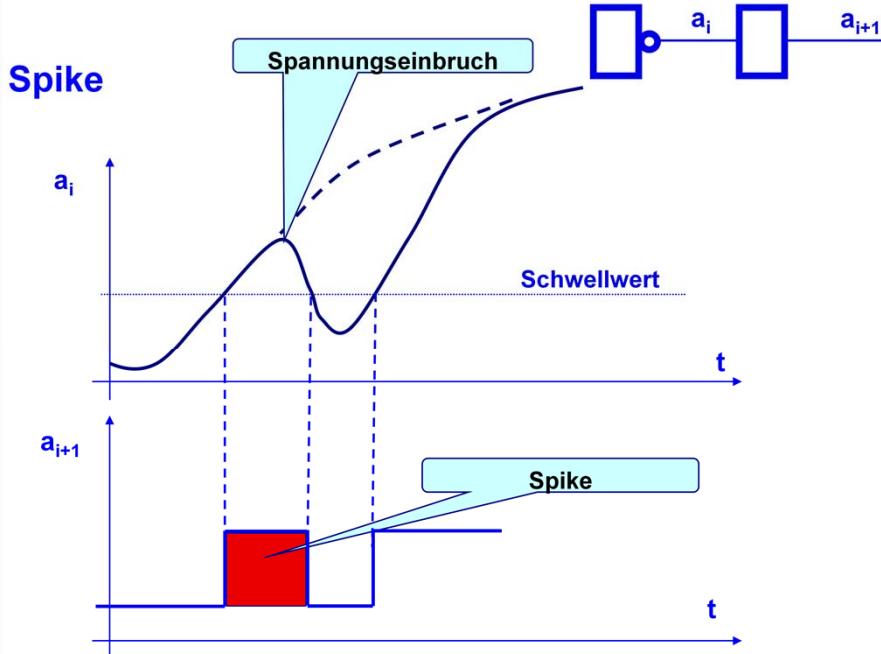


## Spike



*Definition allgemein:*

**Wenn  $e$  sich innerhalb von  $t_D$  nochmals ändert.**



Diese Definition ist eine nicht glücklich gewählte Definition des Begriffes Spike. Meines Erachtens können darunter prinzipiell Effekte wie beispielsweise auch folgender verstanden werden: Zwei Bausteine sind hintereinandergeschaltet. Wechselt der Ausgang  $a_n$  des ersten Bausteins von high auf low, kann der Polaritätswechsel einen erhöhten Stromfluss verursachen, was zu einer erneuten momentanen Spannungserhöhung führt.

Liegt nun die Schaltschwelle des Folgebausteins unglücklich, kann dies an seinem Ausgang einen Spike hervorrufen. Zusatzbemerkung: Die Begriffe Race, Hazard und Spike werden in der Literatur oft widersprüchlich definiert. Entscheidend für den Entwickler ist allein, dass er die unterschiedlich auftretenden Effekte kennt und in der Lage ist, sie zu unterscheiden, um daraus schließen zu können, ob sie "kritisch" sind oder nicht und wie man ihnen begegnen kann.

# Digitale Integrierte Schaltungen

384.086

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



# Kapitel 3, Teil 3

## Schaltwerke

- Einführung
- Speicherelemente
- Synchrone u. asynchrone Schaltwerke
- Mealy- u. Moore-Automaten
- Races und Hazards
- Synthese und Analyse



Die Wahl der richtigen Vorgehensweise bei der Synthese sowie der Analyse von Schaltwerken ist im Allgemeinen mitentscheidend für das Erzielen einer guten Lösung. Je komplexer eine Aufgabenstellung beziehungsweise ein System ist, um so mehr hilft eine *straight forward* Vorgehensweise, eine Lösung innerhalb einer vertretbaren Zeitspanne zu finden. Viele anfangs schwierig erscheinenden Aufgaben lösen sich nahezu von selbst, wenn sie nur ausreichend *stark modularisiert* werden, da sich damit die Einzelschritte vereinfachen. Dabei wird deutlich, dass man sich in vielen Details anfangs nicht festlegen muss, also relativ allgemein bleiben kann (ja sogar sollte, um sich keine Lösungsmöglichkeiten durch zu früh gesetzte Randbedingungen zu verbauen), was gleich zwei Vorteile bietet. Erstens bewirkt eine möglichst späte Festlegung, dass man sich nicht gleich zu Beginn "künstlich" auf wenige Lösungen einschränkt, und zweitens braucht man bei einer iterativen Vorgehensweise (und dazu ist man in der Praxis im Allgemeinen gezwungen) nicht alles nochmals von vorne durchzuarbeiten. Leider zeigt der Alltag in der Lehre, dass eine konsequente Vorgehensweise einen Lernprozess voraussetzt, bevor man die Regeln "freiwillig" einhält. ASIC-Entwicklungs-Tools hingegen verlangen per se zunehmend eine strenge Vorgehensweise, so dass weniger Fehler produziert werden als bei reinen "Papierentwürfen". Die im folgenden beschriebene Vorgehensweise beruht auf der Topdown-Methode. Es ist zu berücksichtigen, dass sie nicht in allen Anwendungsfällen streng eingehalten werden kann, oft ist eine iterative Vorgehensweise unerlässlich.

Hier ist es so ähnlich: Jeder PC-Besitzer muss die bittere Erfahrung machen, dass seine Daten unwiderruflich "zwischendurch" verloren gehen, bis er sich angewöhnt, kontinuierlich und häufig ein Backup seiner Daten durchzuführen.

# Prinzipielle Vorgehensweise bei der Synthese von Schaltwerken

## (1) Verbalisierung der Problemanalyse

*Das Problem ist schriftlich zu formulieren (möglichst frei von Formalismen).*

## (2) Formalisierung der Aussagen

*Die Ein- u. Ausgangsgrößen sind festzulegen, die Randbedingungen des Systems zu formalisieren, kritische Phasen zu analysieren (Werkzeuge: Impuls-Zeitdiagramm, Simulationswerkzeuge, ..).*

## (3) Festlegung der Zustände

*Die möglichen, unterschiedlichen Zustände sind zu bezeichnen, die Ergebnisvektoren festzulegen (keine Codierung der Information ).*

## (4) Entwurf des Zustandsdiagramms



Eine Schaltung zu synthetisieren ist im Allgemeinen einfacher als sie zu analysieren, was sich auch darin äußert, dass sich die Vorgehensweise leicht beschreiben lässt.

## (1) Verbalisierung der Aufgabenstellung

Die Praxis zeigt, dass es wichtig ist, das Problem zunächst *sorgfältig schriftlich* zu formulieren, wodurch der Entwickler gezwungen wird, konkrete Aussagen zu treffen. Mögliche Inkonsistenzen in der Definition und Widersprüche in der Aufgabenstellung werden an dieser Stelle oft schon deutlich (vor allem, wenn es durch eine weitere Person geprüft wird). Zudem kann durch die Rückkopplung mit dem Auftraggeber klargelegt werden, ob die entscheidenden Informationen "angekommen" sind.

Die Verbalisierung sollte möglichst frei von Formalismen sein.

## (2) Formalisierung der Aussagen

Das bisher verbal Formulierte ist nun weitgehend zu formalisieren. Voraussetzung hierfür ist die Bestimmung der Schnittstellen, das bedeutet die Festlegung aller Ein- und Ausgangsgrößen, die Festlegung der Randbedingungen. Erkennbare kritische Phasen sind zu analysieren, wobei Impuls-Zeitdiagramme, Interaktions-Zeitdiagramme usw. dienlich sein können.

## (3) Festlegung der Zustände

Die möglichen inneren Zustände des Systems sind festzulegen, jedoch nicht zu kodieren, da diese Festlegung auf bestimmte Bitkombinationen den Freiheitsgrad einer späteren Optimierung einschränken würde.

## (4) Entwurf des Zustandsdiagramms

Es ist das Zustandsdiagramm zu entwerfen, wobei in die Knoten nur allgemeine Zustandsbezeichnungen einzutragen (*keine* konkreten Bit-Werte) und die Kanten möglichst vollständig mit den Eingangs- und Ausgangsvektoren (*e/a*) zu beschriften sind (da die Eingangs- und Ausgangsvektoren durch die Aufgabenstellung festgelegt werden, können für sie zumeist gleich konkrete Bit-Werte eingetragen werden, wenn sie nicht zu komplex sind).

# Prinzipielle Vorgehensweise bei der Synthese von Schaltwerken

## (5) Elimination redundanter Zustände

*Es kann vorkommen, dass im Schritt (3) u. (4) überflüssige Knoten definiert werden. Äquivalente Knoten, d. h., Knoten, die beim gleichen Eingangsvektor den gleichen Ausgangsvektor generieren u. den selben Folgezustand (Folgeknoten) haben, können eliminiert werden.. (Schritt (3), (4) u. (5) sind als iterative Schritte anzusehen).*

## (6) Codierung der Zustände

*Für w definierte Zustände sind k Zustandsvariablen festzulegen, wobei gilt  $ld(w) \leq k \leq ld(w+1)$ .*

## (7) Festlegung der FFs

*Hier ist die Kreativität und Erfahrung des Entwicklers besonders gefordert, da hiervon oft im entscheidenden Maße der schaltungstechnische Aufwand abhängt.*

## (5) Elimination redundanter Zustände

Zustandsdiagramme beinhalten oft redundante (äquivalente) Zustände (Knoten mit gleichen Ein- und Ausgangsvektoren sowie dem gleichen Folgezustand), die zu eliminieren sind.

## (6) Kodierung der Zustände

Jetzt erst sind die Zustände zu kodieren, das heißt, ihnen werden konkrete Variablen und diesen konkrete Bitwerte zugewiesen. Für w definierte Zustände sind k Zustandsvariablen festzulegen, wobei gilt

$$k = \text{round}[ld(w) + 0,5]$$

bzw.

$$ld(w) \leq k \leq ld(w + 1).$$

## (7) Festlegung der FFs

Sind die FFs durch die Wahl der zu verwendenden Bausteine (Punkt 2: Randbedingungen) nicht schon fest vorgegeben, wird besonders die Kreativität und vor allem Erfahrung des Entwicklers gefordert, da hiervon im entscheidenden Maße der schaltungstechnische Aufwand abhängt.

# Prinzipielle Vorgehensweise bei der Synthese von Schaltwerken

- (8) Zusammenstellen der Übergangstabelle (o. ä.)
- (9) Formulierung des (optimalen) Schaltwerksgleichungssystems
- (10) Ausgabe (Jedec-Datei, Schaltplan, ..)

*Iterative Vorgehensweise ist oft notwendig.*

## Beispiele



### (8) Zusammenstellung der Übergangstabelle

Für den Entwurf kleinerer Automaten ist die Tabellenerstellung unproblematisch, für den Entwurf größerer Automaten oft nur mit starken Vereinfachungen zu realisieren. Um effektiv Schaltungen zu entwickeln, sollte man spätestens zu diesem Zeitpunkt auf geeignete Softwarewerkzeuge zurückgreifen).

### (9) Formulierung des (optimierten) Schaltwerksgleichungssystems

Voraussetzung hierfür sind selbstverständlich effektive Optimierungswerkzeuge, die möglichst nicht nur eine Optimierung bezüglich einer Zustands- oder Ausgangsgröße wahrnehmen, sondern gemäß Kapitel 2 eine mögliche Mehrfachnutzung prüfen.

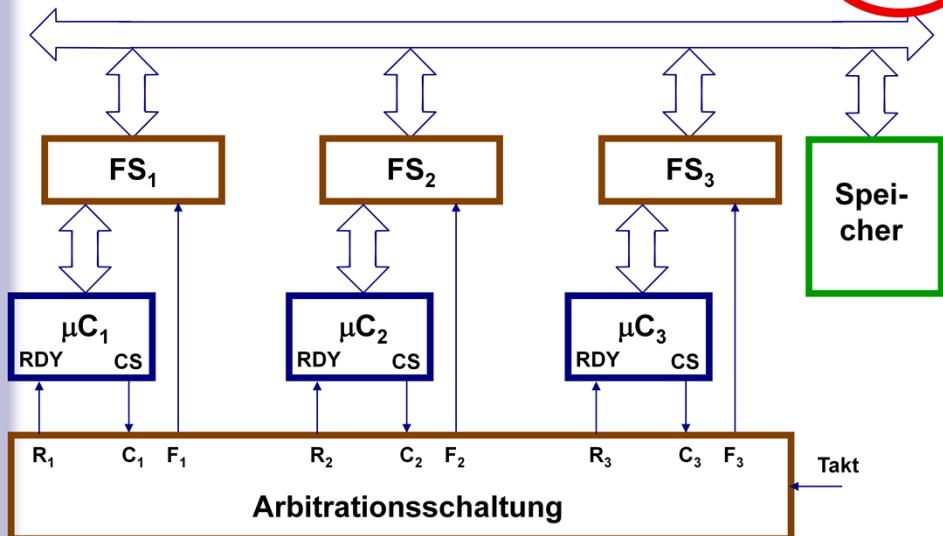
### (10) Ausgabe (Jedec-Datei, Schaltplan, ..)

Die gewünschte Ausgabe ist naturgemäß von der Aufgabenstellung abhängig. Nur eines soll deshalb noch an dieser Stelle erwähnt werden: War es früher eine Selbstverständlichkeit, das Ergebnis als vollständigen Schaltplan vorzulegen, nimmt man heute davon aus verschiedenen Gründen mehr und mehr Abstand. Erstens werden Schaltungsdetails zunehmend formal formuliert. Durch die Umsetzung in eine Grafik gewinnt man in diesem Fall nichts, man verliert nur Zeit (und vielleicht sogar die Übersicht). Zweitens läuft man bei einer iterativen Vorgehensweise, die fast immer notwendig ist, in Gefahr, die Topdown-Methode zu verletzen, was die Ursache von Fehlern sein kann.

**Zusatzbemerkung:** Werden nicht synchrone, sondern asynchrone Schaltwerke entworfen, wird es notwendig, die Liste um weitere Schritte zu ergänzen. So sind genaue Stabilitätsbetrachtungen, die Analyse aller möglichen Zustände usw. notwendig, worauf hier nicht mehr im Einzelnen eingegangen werden kann.

# Berechnung eines sequentiellen Schaltwerkes auf der Basis eines Mealy-Automaten

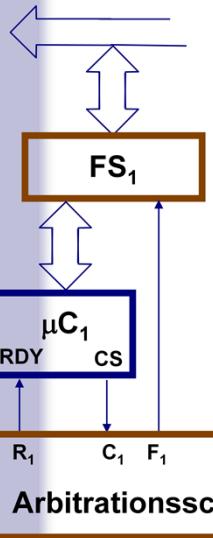
AzÜ wnv



Das Ziel, das im vorliegenden Abschnitt verfolgt wird, ist es, die vorausgegangenen Überlegungen an einem konkreten Beispiel zu verdeutlichen.

## Aufgabenstellung

In einem Multicomputersystem greifen drei Mikrocomputer  $\mu\text{C}_1$ ,  $\mu\text{C}_2$  und  $\mu\text{C}_3$  über einen gemeinsamen Bus auf einen Speicher zu. Damit auf dem Bus keine Kurzschlüsse entstehen, ist zwischen den Computern und dem Bus je eine Freischaltung ( $\text{FS}_1$ ,  $\text{FS}_2$ ,  $\text{FS}_3$ ) vorgesehen, die im wesentlichen aus Three-State-Treibern besteht, deren genaue Schaltung hier jedoch nicht näher interessiert. Die Freischaltungen werden über Signale  $F_1$ ,  $F_2$  und  $F_3$  von einer Arbitrations-Schaltung aktiviert ( $F_i = 1$ ) und geben dann den Zugriff des zugehörigen Mikrocomputers auf den Speicher frei. Dabei muss die Arbitrations-Schaltung sicherstellen, dass immer nur eine Freischaltung zu einem Zeitpunkt aktiv sein darf.



⇒ Multiprozessorsystem

⇒ gemeinsamer Bus

FS: Freischaltung (Steuerleitungen:  $F_1, F_2, F_3$ )

CS: Chip select (Steuerleitungen:  $C_1, C_2, C_3$ )

RDY: Ready (Steuerleitungen:  $R_1, R_2, R_3$ )

## Funktionsablauf:

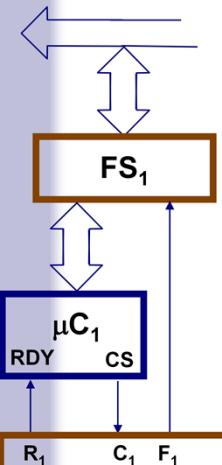
1.  $\mu P_i$  setzt  $C_i$  (CS) aktiv.
2. ArbS setzt  $F_i$ , wenn frei und deaktiviert  $R_i$ , wenn besetzt
3. wenn Zugriff beendet:  $\mu P_i$  deaktiviert CS und ArbS deaktiviert  $F_i$

Der Speicherzugriff eines Mikrocomputers läuft wie folgt ab:

- Der Mikrocomputer, der auf den Speicher zugreifen möchte, setzt seine CS-Leitung aktiv.
- Greift kein anderer Mikroprozessor auf den Speicher zu, erzeugt die Arbitrations-Schaltung das zugehörige Signal zur Freischaltung. Kommuniziert gerade ein anderer Mikrocomputer mit dem Speicher, deaktiviert die Arbitrations-Schaltung das zum anfordernden Computer gehörende RDY-Signal und blockiert somit diesen Mikrocomputer.
- Nach erfolgtem Speicherzugriff deaktiviert der berechtigte Mikrocomputer sein CS-Signal, worauf die Arbitrations-Schaltung das zugehörige Freigabesignal deaktiviert. Wurde zuvor ein anderer Mikrocomputer blockiert, wird anschließend dessen Blockierung aufgehoben und sein Freigabesignal aktiviert, so dass er nun seinen Speicherzugriff ausführen kann.

## Berechnung eines sequentiellen Schaltwerkes auf der Basis eines Mealy-Automaten

AzÜ wnv



### Funktionsablauf:

1.  $\mu P_i$  setzt  $C_i$  (CS) aktiv.
2. ArbS setzt  $F_i$ , wenn frei und deaktiviert  $R_i$ , wenn besetzt
3. wenn Zugriff beendet:  $\mu P_i$  deaktiviert CS und ArbS deaktiviert  $F_i$

### Regeln:

- FIFO-Prinzip
- bei gleichzeitiger Anforderung folgende Prioritäten:  
zuerst  $\mu P_1$ , dann  $\mu P_2$  und dann  $\mu P_3$
- RDY grundsätzlich aktiv

## Arbitrationsschaltung (ArbS)

## Vorgehen?

Es gelten folgende Regeln:

- Grundsätzlich wird die Anforderung des zuerst anfragenden Mikrocomputers auch zuerst bearbeitet. Nur in Fällen, wo gleichzeitige Anforderungen vorliegen, soll  $\mu C_1$  vorrangig vor  $\mu C_2$  und dieser vorrangig vor  $\mu C_3$  bedient werden.
- Die RDY-Signale aller Mikrocomputer sind grundsätzlich aktiv zu halten. Nur wenn eine Blockierung droht, darf eine oder dürfen zwei der Leitungen  $R_1$ ,  $R_2$ ,  $R_3$  deaktiviert (auf 0 gesetzt) werden.

Die Schaltung ist zu entwickeln.

## Lösung

1. Verbalisierung ✓

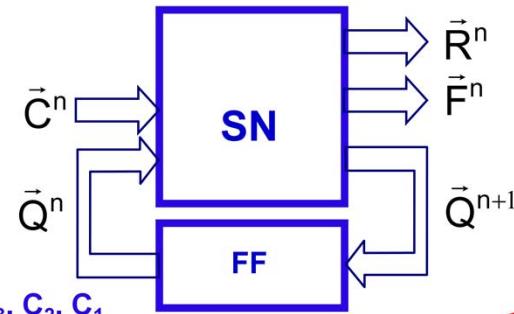
2. Formalisierung

Eingänge:  $C_3, C_2, C_1$

Ausgänge:  $F_3, F_2, F_1, R_3, R_2, R_1$

innere Zustände:  $Z_i$

FF-Ausgänge:  $Q_i$



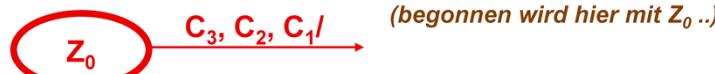
AzÜ wnv

3. Festlegen der Zustände

nicht möglich!

(da Funktionsablauf noch unbekannt)

4. Entwurf des Zustandsdiagramms



## Lösung

Es soll weitgehend die Vorgehensweise des vorausgegangenen allgemeinen Erläuterung eingehalten werden.

(1) *Verbalisierung der Aufgabenstellung:* Sie ist durch die Aufgabenformulierung erfolgt.

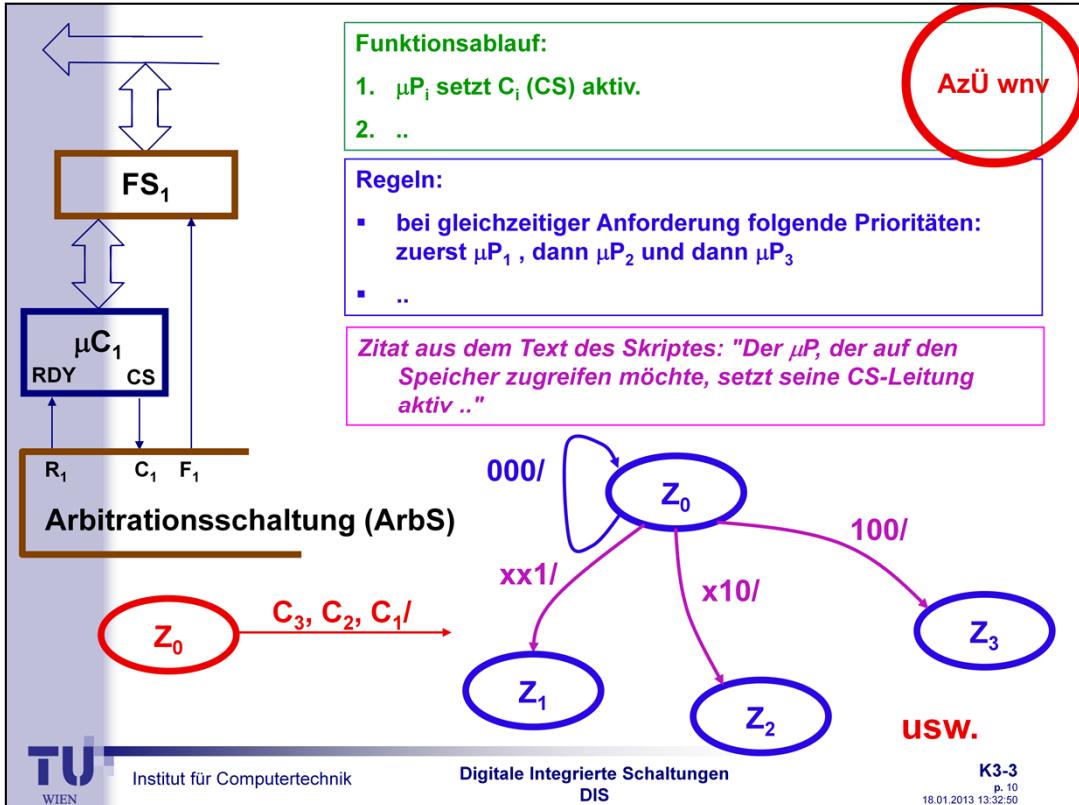
(2) *Formalisierung der Aussagen:* Die Schnittstellen der Arbitrationsschaltung sind:

Eingänge:  $C_3, C_2, C_1$

Ausgänge:  $F_3, F_2, F_1, R_3, R_2, R_1$ .

Das System nimmt unterschiedliche innere Zustände ein, ist also ein sequentielles Schaltwerk, wobei allerdings über die Anzahl der inneren Zustände noch keine Aussage getroffen wird. Sie werden im folgenden mit  $Z_i$  bezeichnet. Die Ausgänge der verwendeten FFs, denen  $Z_i$  zugeordnet werden, werden mit  $Q_i$  beschriftet. Es ergibt sich somit ein Blockschaltbild auf der Basis des Mealy-Automaten nach Bild oben.

(3) *Festlegung der Zustände:* Der Punkt entfällt, da in diesem Stadium in diesem Fall die Anzahl der Zustände nicht angegeben werden kann.



AzÜ wnv

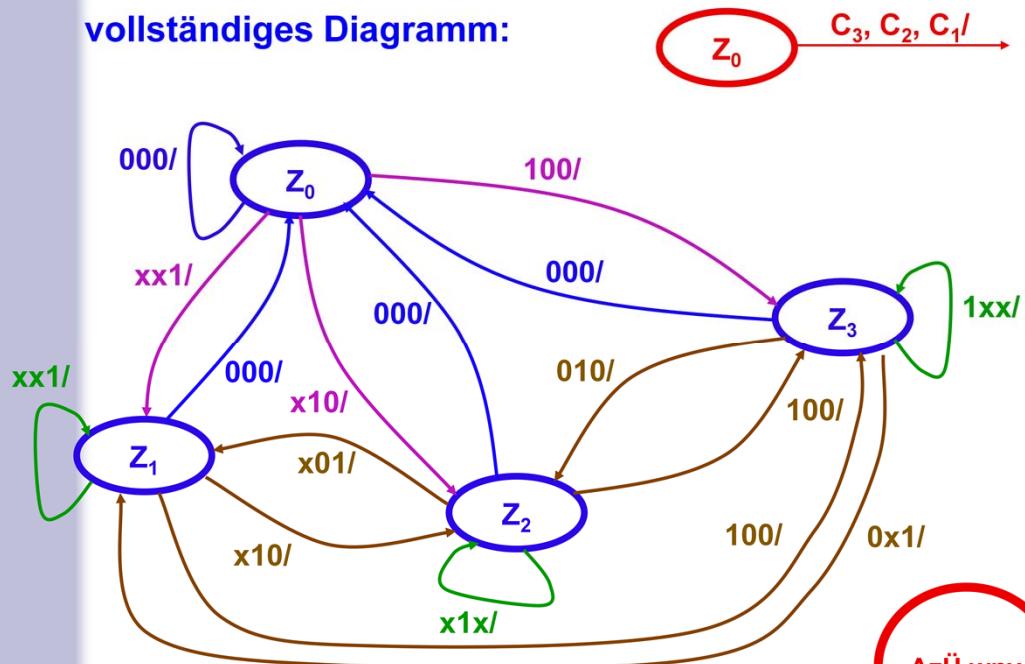
(4) Entwurf des Zustandsdiagramms: Die Aufgabe ist am einfachsten zu lösen, wenn man vom Zustand  $Z_0$  ausgehend sukzessiv die Zustandsübergänge einträgt, wie sie in der Aufgabenstellung gegeben sind. Beginnen wir beispielsweise mit der Interpretation folgender Sätze:

"Der Mikrocomputer, der auf den Speicher zugreifen möchte, setzt seine CS-Leitung aktiv." und  
"Grundsätzlich wird die Anforderung des zuerst anfragenden Mikrocomputers auch zuerst bearbeitet. Nur in Fällen, wo gleichzeitige Anforderungen vorliegen, soll  $\mu C_1$  vorrangig vor  $\mu C_2$  und dieser vorrangig vor  $\mu C_3$  bedient werden."

Danach geht der Zustand  $Z_0$  in sich selbst über, wenn kein Mikrocomputer eine Anforderung stellt, also die Signale  $C_3$ ,  $C_2$  und  $C_1$  deaktiviert sind, was direkt in ein Zustandsdiagramm eingetragen werden kann (Bild oben): Ein Zugriff des ersten Mikrocomputers, vom Zustand  $Z_0$  ausgehend, hat einen Übergang in einen zweiten Zustand  $Z_1$  zur Folge. Die Eingangs-Übergangsbedingung lautet hierfür  $x \ x \ C_1 /$ . Die beiden x bedeuten, dass ein gleichzeitig gewünschter Zugriff auf die Speicher durch  $\mu C_2$  und  $\mu C_3$  gegenüber dem Zugriff durch  $\mu C_1$  nachrangig behandelt wird.

Bei manchen Aufgaben ist es von Vorteil, nun direkt die Ausgangsgrößen mit in die Grafik einzutragen. Im vorliegenden Fall ist davon abzuraten, da die Aufgabe an sich zwar einfach ist, aber die Zusammenhänge dort zunächst wie ein Puzzle zusammengefügt werden müssen. Im ersten Schritt sollte deshalb zunächst die Beschriftung aller Knoten und Übergänge und die Beschriftung der Eingänge erfolgen, und im zweiten Schritt sind die Ausgangsgrößen hinzuzufügen. Man erhält damit gleichzeitig eine Kontrolle über den ersten Schritt.

## vollständiges Diagramm:



In Bild oben ist das Ergebnis dargestellt, in dem allerdings die Ausgangsgrößen aus Gründen der Übersicht noch nicht eingetragen sind.

## 5. Elimination redundanter Zustände

keine vorhanden

## 6. Codierung der Zustände

$$Z_0: Q_1 Q_0 = 00$$

$$Z_1: Q_1 Q_0 = 01$$

$$Z_2: Q_1 Q_0 = 10$$

$$Z_3: Q_1 Q_0 = 11$$

## 7. Festlegung der FFs

..

(5) *Elimination redundanter Zustände:* Redundante Zustände sind nicht vorhanden.

(6) *Kodierung der Zustände:* Welche Kodierung zum günstigsten Ergebnis führt, soll hier nicht untersucht werden (in der Praxis führt man diese Prüfung am einfachsten - wenn die Lösung nicht direkt ersichtlich ist - mit Hilfe des Rechners durch). Der Einfachheit halber werden die Zustände nach dem Dualkode kodiert:

(7) *Festlegung der FFs:* Auch dieser Punkt soll im folgenden nicht von Interesse sein.

## 8. Übergangstabelle

t <sup>n</sup>			t <sup>n</sup>		t <sup>n+1</sup>		t <sup>n</sup>					
C <sub>3</sub>	C <sub>2</sub>	C <sub>1</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>0</sub>	F <sub>3</sub>	F <sub>2</sub>	F <sub>1</sub>	R <sub>3</sub>	R <sub>2</sub>	R <sub>1</sub>
0	0	0	x	x	0	0	0	0	0	1	1	1
0	0	1	0	0	0	1	0	0	1	1	1	1
0	1	0	0	0	1	0	0	1	0	1	1	1
0	1	1	0	0	0	1	0	0	1	1	0	1
1	0	0	0	0	1	1	1	0	0	1	1	1
1	0	1	0	0	0	1	0	0	1	0	1	1
1	1	0	0	0	1	0	0	1	0	0	1	1
1	1	1	0	0	0	1	0	0	1	0	0	1
0	0	1	0	1	0	1	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	0	0	1	1	0	1
0	0	0	1	1	1	1	0	0	0	1	1	1
			1	0	1	0	0	0	1	0	1	1
							1	0	0	1	1	1

## 9. Optimierte Gleichungssystem

usw.

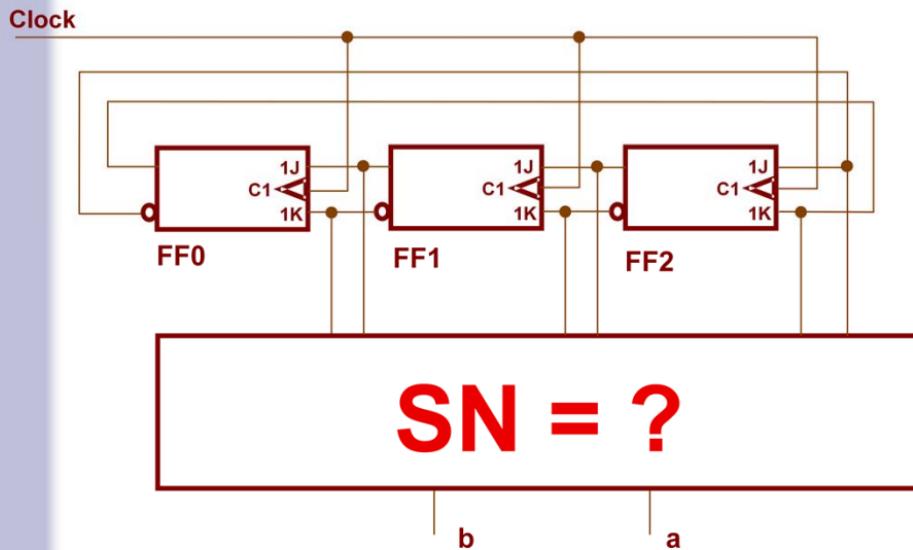
AzÜ wnv



(8) Zusammenstellung der Übergangstabelle: Die Tabelle ist aus dem Zustandsdiagramm und der Aufgabenstellung gemäß den Erläuterungen unter Punkt (4) zu erstellen. Die folgende Tabelle zeigt das Ergebnis.

(9) Formulierung des (optimierten) Schaltwerksgleichungssystems.

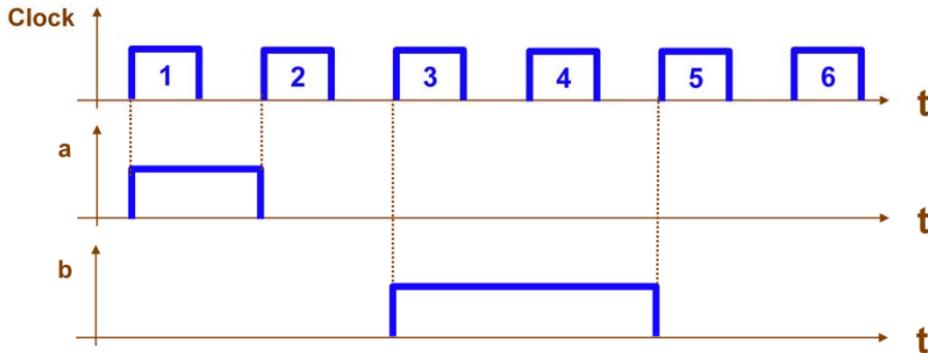
**Beispiel: Schieberegister**  
synchrone Schaltwerk  
Prüfung der TU Karlsruhe



# Aufgabenstellung

## System

- 2 Phasengenerator für einen µP
- 6 Takte



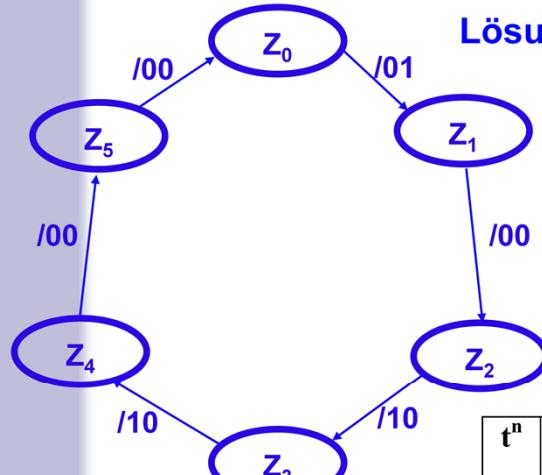
- Zugrundegelegt ein Schieberegister
- das Schaltnetz (SN) ist zu entwickeln

## Berechnen Sie nun SN, also:

- $a = f(??)$
  - $b = f(??)$
- 
- Wieviele FFs benötigt man für ein Schieberegister mit 6 Takten?
  - Wieviele haben wir hier?
  - ..??



## Lösung: "Möbiusband"



.. daraus werden die  
Gleichungen abgeleitet usw. ..

$t^n$	$t^n$			$t^{n+1}$			$t^n$	
$Z_i$	$Q_2$	$Q_1$	$Q_0$	$Q_2$	$Q_1$	$Q_0$	$b$	$a$
$Z_0$	0	0	0	1	0	0	0	1
$Z_1$	1	0	0	1	1	0	0	0
$Z_2$	1	1	0	1	1	1	1	0
$Z_3$	1	1	1	0	1	1	1	0
$Z_4$	0	1	1	0	0	1	0	0
$Z_5$	0	0	1	0	0	0	0	0

Roter Punkt: nicht an die Studenten weitergeben.

Wo bleiben die beiden anderen Zustände:  $Z_6$  und  $Z_7$ ? das sind die Variationen 010 und 101, die einen eigenen Ablauf bilden.

## Beispiel der Berechnung auf der Basis eines Moore-Automaten

Mealy:

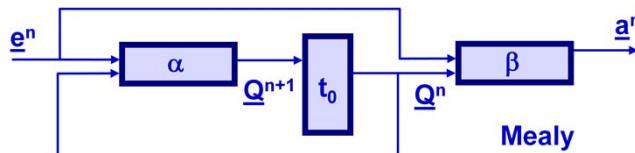
$$Q^{n+1} = \alpha(e^n, Q^n)$$

$$a^n = \beta(e^n, Q^n)$$

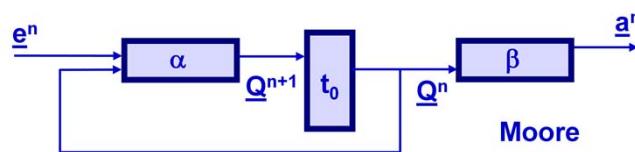
Moore:

$$Q^{n+1} = \alpha(e^n, Q^n)$$

$$a^n = \beta(Q^n)$$



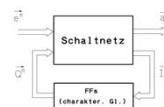
Mealy



Moore

### Berechnung eines sequentiellen Schaltwerkes auf der Basis eines Moore-Automaten (Beispiel)

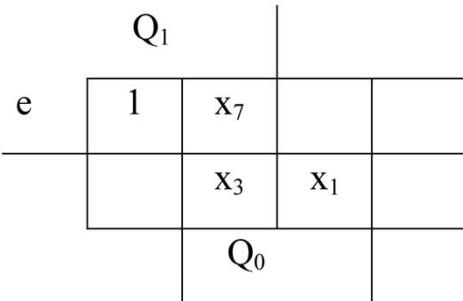
Legt man beliebige Zustandsgrafen zugrunde, führt das im Allgemeinen zum Mealy-Automaten, denn er erlaubt die meisten Freiheitsgrade. Wünscht man dagegen einen Moore-Automaten, der bezüglich der Störproblematik ja günstiger ist, da die Eingangsgröße nicht direkt auf die Ausgangsgröße wirkt, muss man eben gerade auf diesen Freiheitsgrad (die direkte Wirkung des Eingangs auf den Ausgang) verzichten. Man kann diesen Umstand am einfachsten beim Entwurf des Zustandsdiagramms berücksichtigen.



# Mealy $\Leftrightarrow$ Moore

**Fall 1:**

t <sup>n</sup>			t <sup>n+1</sup>		t <sup>n</sup>
e	Z	Q <sub>1</sub>	Q <sub>0</sub>	Z	a
0	0	0	0	0	0
0	1	0	1	x	x <sub>1</sub>
0	2	1	0	3	0
0	3	1	1	x	x <sub>3</sub>
1	0	0	0	1	0
1	1	0	1	2	0
1	2	1	0	3	1
1	3	1	1	x	x <sub>7</sub>



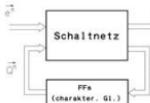
$$a = e Q_1 = \beta(e^n, Q^n)$$



**Mealy**

Sehen wir uns hierzu die Beispiele an.

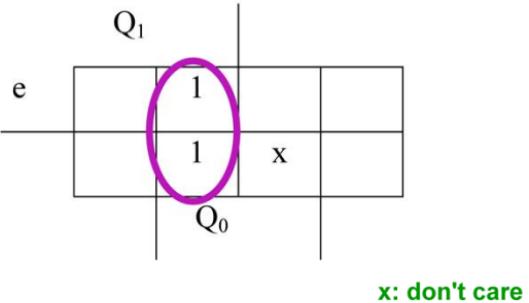
Berechnet man für die Diagramme die jeweilige Ausgangsgröße, erhält man sie hier bspw. in Abhängigkeit der Eingangsgröße.



# Mealy $\Leftrightarrow$ Moore

Fall 2:

t <sup>n</sup>				t <sup>n+1</sup>	t <sup>n</sup>
e	Z	Q <sub>1</sub>	Q <sub>0</sub>	Z	a
0	0	0	0	0	0
0	1	0	1	x	x
0	2	1	0	0	0
0	3	1	1	0	1
1	0	0	0	1	0
1	1	0	1	2	0
1	2	1	0	3	0
1	3	1	1	0	1



$$a = Q_1 Q_0 = \beta(Q^n)$$

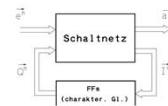
Moore

# Moore

## Entwurfsmethodik

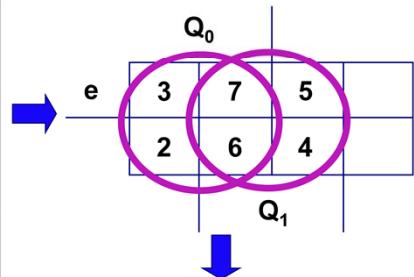
Ziel:  $a \neq f(e)$ !

Ausführung: a muss für e und für  $\bar{e}$  unabhängig sein.



Beispiel:

i	e	Q <sub>1</sub>	Q <sub>0</sub>	Z <sup>n</sup>	Z <sup>n+1</sup>	a
0	0	0	0	0	0	0
1	1	0	0	0	1	0
2	0	0	1	1	2	1
3	1	0	1	1	2	1
4	0	1	0	2	3	1
5	1	1	0	2	3	1
6	0	1	1	3	0	1
7	1	1	1	3	0	1

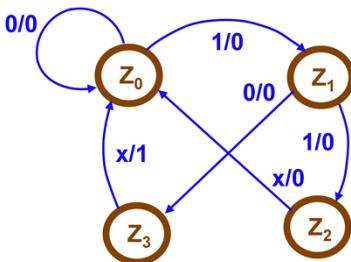
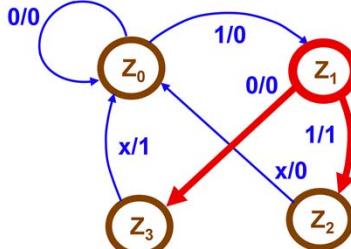
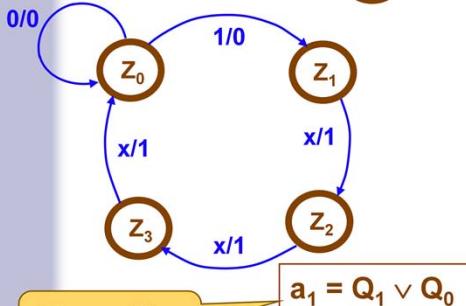
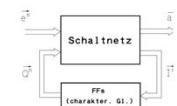


$$a = Q_1 \vee Q_0 = \beta(Q^n)$$

Der Ausgang a muss unabhängig vom Zustand des Eingangs e sein. Wie kann das erreicht werden? Indem beispielsweise in der Spalte Eingang e immer der Wechsel 0 nach 1 direkt hintereinander geschrieben wird, und nun beim Entwurf darauf geachtet wird, dass in der Spalte Ausgang a für diese beiden Zeilen der Wert der gleiche ist, also entweder 0 oder 1.

Diese Möglichkeit ist für die Prüfung in diesem Fach nicht erlaubt, da ja vorgeschrieben wird, die Zeilen der Eingänge fortlaufend 000, 001, 010, ... zu schreiben. Eine prinzipiell einfachere Methode wird auf der nächsten Seite dargestellt: Man sucht die Lösung gleich im Zustandsdiagramm zu finden.

# Analyse



$$a_3 = Q_1 Q_0$$

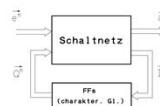
nur hier:

$$a = \beta(e, Q^n)$$

$$a_2 = eQ_0 \vee Q_1 Q_0$$

Im Fall Bild rechts liegt ein Mealy- und in den beiden anderen Fällen ein Moore-Automat vor. Das kann direkt aus den Diagrammen abgelesen werden. Beim Moore-Automaten darf sich eine Eingangsaktion erst nach einem Takt auf den Ausgang auswirken. In Bild rechts verursacht ein Verlassen des Zustandes  $Z_1$  einen Ausgangswert  $a = 1$  oder  $a = 0$ , abhängig davon, ob als nächster Zustandswert  $Z_2$  oder  $Z_3$  angesprungen wird, das heißt, ob der Eingangswert  $e = 1$  oder  $e = 0$  gesetzt ist. Im Fall  $e = 1$  ändert sich jedoch der Ausgangswert:  $a$  von  $0 \rightarrow 1$ , was bedeutet:  $a = (e, Q)$ .

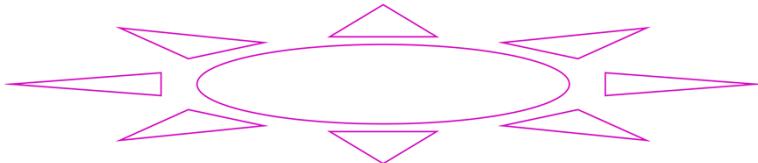
# Synthese



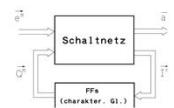
Wie entwickelt man einen Moore-Automaten am einfachsten?

vom Zustandsgrafen ausgehend

dann über Übergangstabelle usw.



Beim Entwurf eines Moore-Automaten ist also darauf zu achten, dass die Zustandsdiagramme entsprechend zu entwerfen sind. Hierzu ein einfaches Beispiel, das nicht vollständig durchgerechnet, sondern nur vom Prinzip her erläutert werden soll.

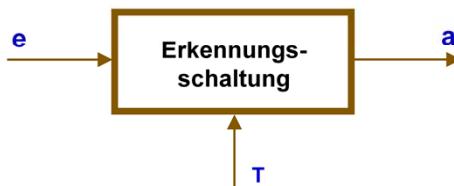
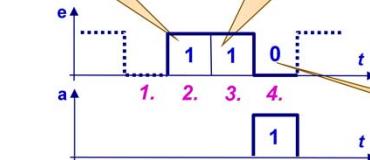


# Beispiel Moore-Automat

1. Bit = 1

2. Bit = 1

3. Bit = 0



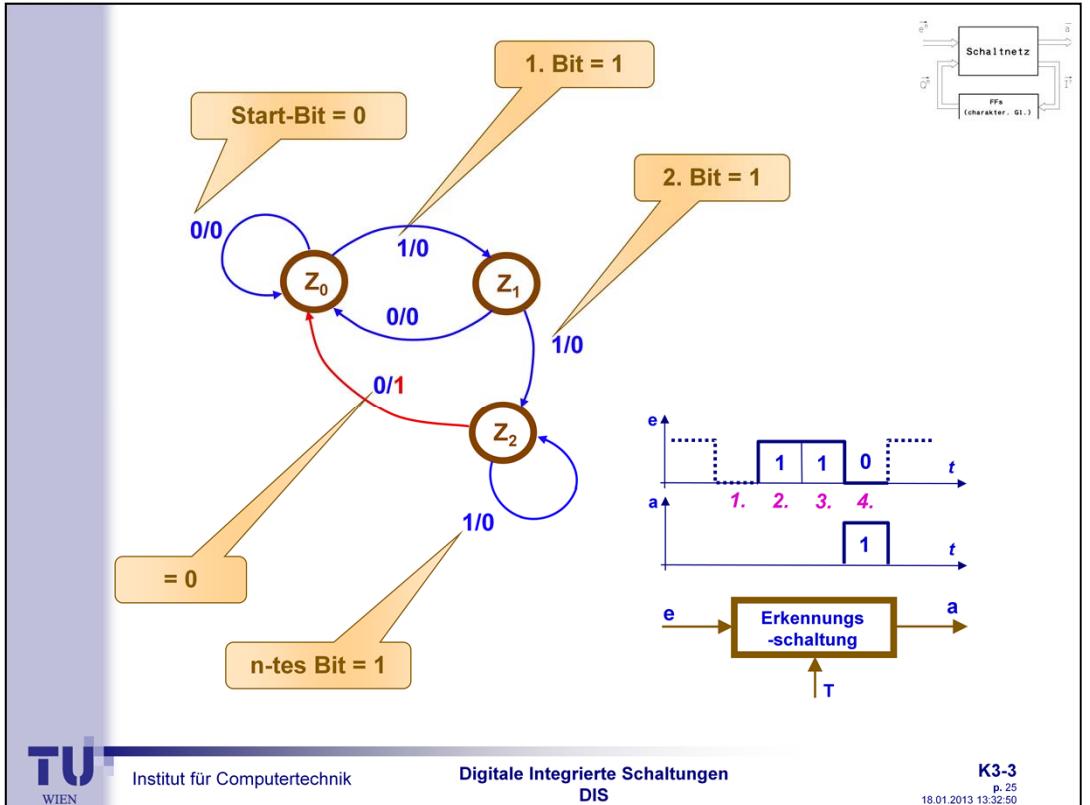
Realisieren Sie die Erkennungsschaltung des asynchronen Datenrahmens als Moore-Automat.

Erkannt werden soll also die Folge 110.

Es wird kein Start-Bit vorausgesetzt: akzeptiert wird also auch die Folge: ..1110

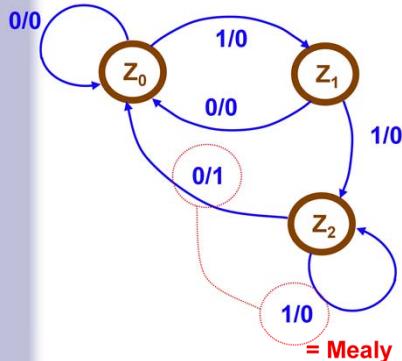
## Aufgabenstellung

Entsprechend dem Bild oben ist eine Schaltung zu entwickeln, die über den Eingang e einen Bitstrom erhält, den sie auf 3-bit-Rahmen hin zu analysieren hat. Die beiden ersten Bits weisen stets den Wert 1, das folgende Stopbit stets den Wert 0 auf. Wird kein Rahmen übertragen, wird davon ausgegangen, dass  $e = 0$  ist. Erkennt die Schaltung einen derartigen Rahmen, ist der Ausgang kurz auf 1 zu setzen ansonsten soll er den Wert 0 haben. Es wird kein Start-Bit=0 vorausgesetzt. Das Start-Bit setzt die Prozedur nur in Gang.

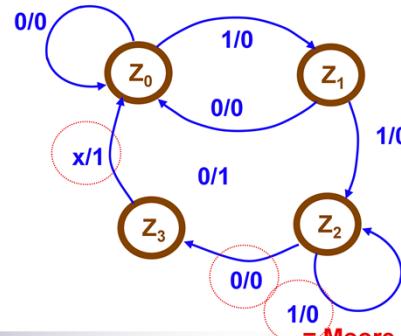
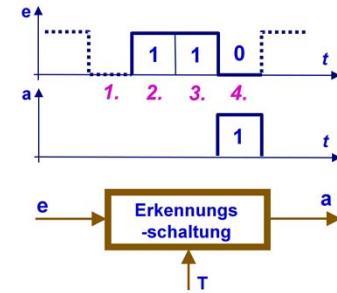


## Lösung 1:

Achtet man nicht darauf, ob die Schaltung auf der Basis eines Mealy- oder Moore-Automaten entwickelt werden soll, gelangt man aufgrund der Aufgabenstellung im einfachsten Fall zu einer Schaltungskonfiguration gemäß dem Zustandsgrafen nach Bild oben. Der Übergang von  $Z_2$  nach  $Z_0$  im Fall  $e/a = 0/1$  zeigt aber, dass der Eingang in diesem Fall ohne eine Zeitverzögerung auf den Ausgang wirkt (oder anders formuliert, der Eingang wirkt sich direkt auf den Ausgang aus, denn zwischen  $Z_2$  und  $Z_0$  wirkt sich der Eingang  $e = 0$  so aus, dass der Ausgang  $a = 1$  gesetzt wird und nicht  $a = 0$  bleibt. Das Ergebnis stellt also den Mealy-Automaten dar.



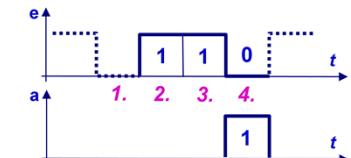
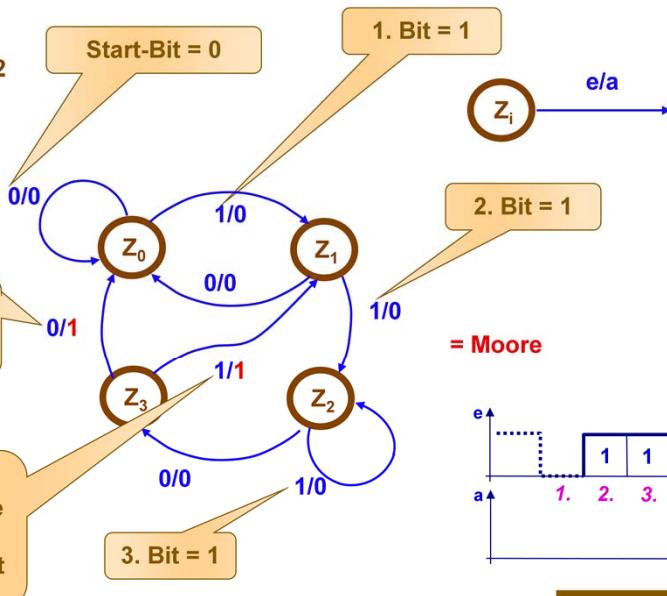
Lösungsvariante 1



Soll die Schaltung jedoch in einem ASIC integriert werden, ist sie als Moore-Automat zu entwerfen. Eine entsprechende Umwandlung ist nicht ohne weiteres möglich, da die Informationsmenge, die die Maschine beinhaltet, die gleiche bleiben muss. Wenn die Eingangsaktion nun nicht direkt auf den Ausgang wirken darf, muss ein innerer Zustand hinzugefügt werden. Die Gleichung  $a = Q_1 Q_0 = \beta(Q)$  kann nun direkt abgeleitet werden.

Das Beispiel lässt deutlich werden, dass die Umwandlung einer Schaltung, die nach dem Mealy-Prinzip entworfen wurde, in eine Schaltung nach dem Moore-Prinzip nicht in jedem Fall einfach sein muss. Oft ist ein erhöhter Schaltungsaufwand notwendig, und - was oft weitreichende Konsequenzen hat - man erhält ein anderes Timing, was der Übergang vom Bild oben zu der Darstellungen unten verdeutlicht.

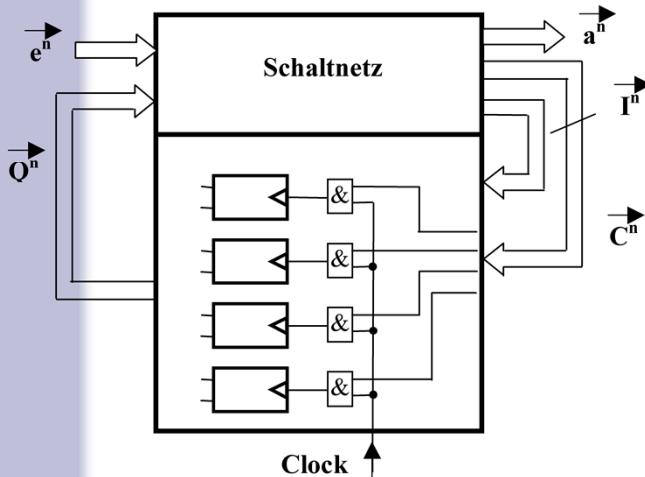
## Lösungsvariante 2



## Lösung 3:

Eine Lösungsmöglichkeit ist dargestellt.

# Modifizierte synchrone Schaltwerke: Taktausblendung



$C^n = (C_{k-1}, C_k, \dots, C_1, C_0)$   
= Steuervektor

Aufgabe: Das FF wird nur dann freigegeben, wenn es schalten soll,  
sonst  $C_i = 0$ !

Unter einem Schaltwerk mit Taktausblendung wird in der Literatur ein Schaltwerk verstanden, bei dem der synchrone Takt nicht direkt an die FFs gelangt. In Abhängigkeit von einem weiteren Schaltnetz  $\gamma$  mit dem Ausgangsvektor  $C^n$  wird über eine Torschaltung der Takt dann ausgeblendet, wenn er nicht relevant ist. Dies soll an einem einfachen Beispiel erläutert werden.

# Taktausblendung

Vorgehensweise:

aus der Übergangstabelle ist zu entnehmen:

wenn FF: 0 → 1 bzw. 1 → 0

dann: C = 1 sonst

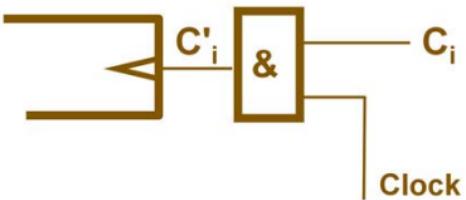
C = 0

→ nun 3 Schaltwerke zu entwerfen:

$$(1) \quad \bar{a}^n = \bar{\beta}(\bar{e}^n, \bar{Q}^n)$$

$$(2) \quad \bar{Q}^{n+1} = \bar{a}(\bar{e}^n, \bar{Q}^n)$$

$$(3) \quad \bar{C}^n = \bar{\gamma}(\bar{e}^n, \bar{Q}^n)$$



## Taktausblendung

$$Q_i^n \rightarrow Q_i^{n+1} \Rightarrow C^n$$



i	e	t <sup>n</sup>			t <sup>n+1</sup>			t <sup>n</sup>				
		Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Z <sub>i</sub>	Z <sub>i+1</sub>	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>0</sub>
1	0	0	0	0	0	2	0	1	0	0	1	0
2	0	0	0	1	1	3	0	1	1	0	1	0
3	0	0	1	0	2	4	1	0	0	1	1	0
4	0	0	1	1	3	0	0	0	0	0	1	1
5	0	1	0	0	4	3	0	1	1	1	1	1
6	0	1	0	1	5	x	x	x	x	x	x	x
7	0	1	1	0	6	x	x	x	x	x	x	x
8	0	1	1	1	7	x	x	x	x	x	x	x
9	1	0	0	0	0	1	0	0	1	0	0	1
10	1	0	0	1	1	3	0	1	1	0	1	0
11	1	0	1	0	2	4	1	0	0	1	1	0
12	1	0	1	1	3	4	1	0	0	1	1	1
13	1	1	0	0	4	3	0	1	1	1	1	1
14	1	1	0	1	5	x	x	x	x	x	x	x
15	1	1	1	0	6	x	x	x	x	x	x	x
16	1	1	1	1	7	x	x	x	x	x	x	x

Die Tabelle ist die Zustandstabelle eines synchronen Schaltwerkes mit  $e$  als Eingangsdatum,  $Q^n$  und  $Q^{n+1}$  sind die Zustandsvektoren, und die Ausgangsleitung ist nicht berücksichtigt. Wechselt das System vom Zustand  $Z_0$  in den Zustand  $Z_2$  (erste Zeile in der Tabelle rechte Seite), verändert sich  $Q_2$  nicht, das bedeutet, der Takt kann in diesem Fall über den Vektor  $C^n$  ausgeblendet werden ( $C_2 = 0$ ). Anders bei  $Q_1$ ,  $Q_1$  wechselt von 0 nach 1,  $C_1$  ist deshalb auf 1 zu setzen.

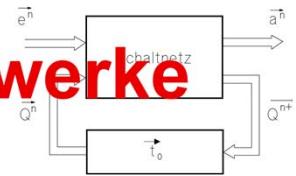
In dieser Art und Weise wird fortgefahrene, bis der vollständige Vektor  $C^n$  vorliegt. Dann kann der Vektor  $Q^{n+1}$  so vereinfacht werden, dass für alle Zustände, in denen  $C_i = 0$  ist, auch  $Q_i = 0$  gesetzt werden kann. Es wird deutlich, dass sich so die Schaltung für  $Q^{n+1}$  vereinfachen lässt. Doch dafür kommt die Schaltung γ und die Torschaltung hinzu. Geht man von der Theorie des Mealy-Automaten aus, dass das Schaltnetz durch die Optimierung prinzipiell die Schaltung ist, die die geringste Anzahl logischer Kombinationen aufweist, muss ein Schaltwerk mit Taktausblendung mehr logische Funktionen enthalten als eine Schaltung nach dem Entwurf des Mealy-Automaten, wie er bisher gezeigt wurde. Warum dann dieser Aufwand?

Wie schon erwähnt wurde, ist die Anzahl der logischen Funktionen des Schaltnetzes im Mealy-Automaten auch abhängig von der Art der verwendeten FFs. So auch hier. Werden FFs eingesetzt, in denen die Torschaltung und vielleicht noch zusätzliche Logik integriert ist, wie beispielsweise in manchen JK-FFs, kann sich selbstverständlich der zusätzliche logische Schaltungsaufwand (also das reine Schaltnetz, oberer Teil des Mealy-Automaten) vereinfachen. Da nun im digitalen ASIC-Entwurf aus Platzgründen stets das einfachste FF verwendet werden sollte, ist das Verfahren der Taktausblendung völlig irrelevant geworden.

Man wird in der Literatur noch ähnliche Verfahren finden, doch sind diese heute zumeist ebenso uninteressant. Das Verfahren der Taktausblendung ist nur beispielhaft deshalb aufgeführt worden, um aufzuzeigen, dass weitere Möglichkeiten der Schaltungsvereinfachung existieren, wenn die Randbedingungen, die für den idealen Mealy-Automaten gelten, geändert werden.

Auf einen Aspekt wurde noch nicht deutlich genug hingewiesen: die Laufzeitprobleme, die durch die Taktausblendung entstehen können, sind gravierend. Ausgangspunkt sei ein stabiler Zustand des Systems. Veranlasst danach der Takt durch einen Flankenwechsel das Register zu schalten, wird konsequenterweise ein neuer Vektor  $C^n$  gebildet, der Ursache dafür sein kann, dass die Taktflanke an FFs wieder weggenommen wird. Die Taktausblendung führt also zur Taktimpulszeitreduzierung. Die Schaltung ist somit auch nicht als synchrones Schaltwerk zu bezeichnen, obwohl ein gemeinsamer Takt zugeführt wird. Für digitale ASICs also völlig ungeeignet.

# Asynchrone Schaltwerke



## Vorteile:

- i. a. geringe Schaltfläche
  - unmittelbare Reaktion auf  $e$   
*schneller als synchr. Schaltwerke !!*
  - Mealy-Automat
- Definition*

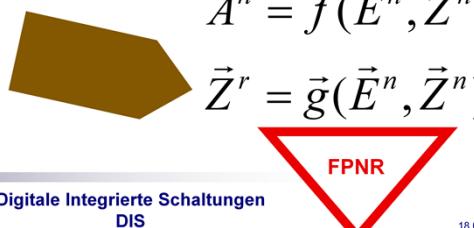
$$\vec{E}^n = (E_1^n, E_2^n, \dots, E_m^n)$$

$$\vec{A}^n = (A_1^n, A_2^n, \dots, A_p^n)$$

$$\vec{Z}^n = (Z_1^n, Z_2^n, \dots, Z_k^n)$$

zur Abgrenzung gegenüber  
synchroner Schaltwerke

*n: momentaner Zeitpunkt*  
*r: zeitlich folgender Zeitpunkt*

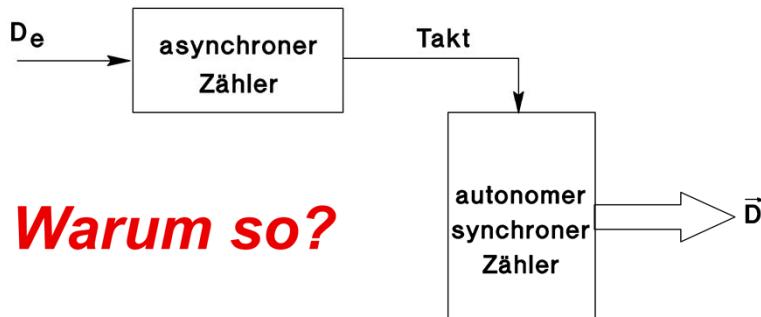
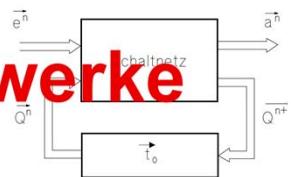


In diesem Abschnitt sollen noch Aspekte kurz angesprochen werden, die sich aus Zeitmangel nicht in die anderen einarbeiten ließen, aber immer wieder in Diskussionen mit Studenten zur Sprache kommen.

## Asynchrone Schaltwerke

Die Synthese asynchroner Schaltwerke wird in der Literatur spärlich behandelt. Selbst in Standardwerken wird die Thematik nur angesprochen oder nur kurz abgetan. Warum das so ist, wurde in diesem Skript schon mehrfach dargelegt: Entwürfe asynchroner Schaltungen sind aufwendiger und schwieriger zu übersehen als Entwürfe synchroner Schaltungen; sie sollten deshalb vermieden werden. Doch wie immer, sind derartige Aussagen mit Vorsicht zu behandeln. Man könnte sich vorstellen, dass die zahlreichen Schritte, die der Entwurf asynchroner Schaltungen verlangt, weitgehendst automatisiert werden. Dann sind die Voraussetzungen besser, und genau dies wird heute in der Forschung mehr und mehr berücksichtigt.

# Asynchrone Schaltwerke



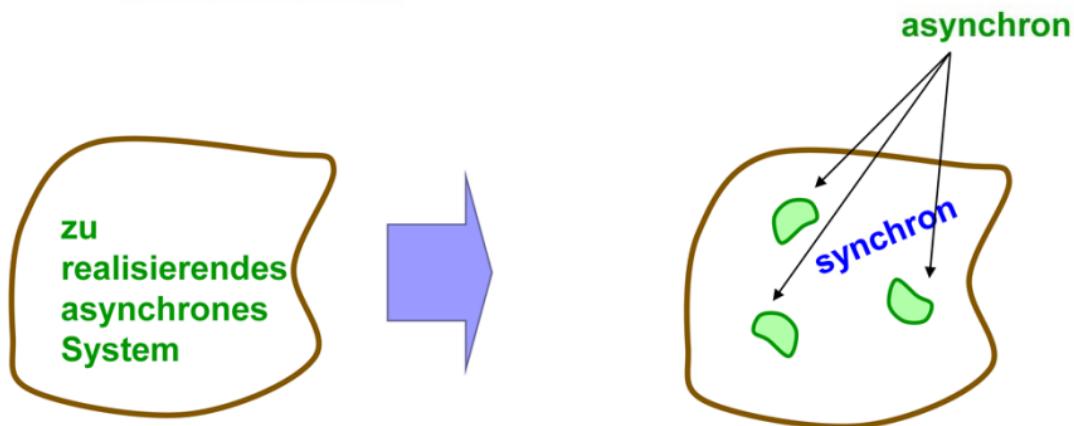
**Warum so?**

Der Entwurf asynchroner Schaltungen wird so mehr und mehr wieder zum wissenschaftlichen Forschungsziel, da asynchrone Schaltkreise im Allgemeinen weniger Schaltungskomponenten aufweisen, und die Schaltgeschwindigkeit nicht durch einen vorgegebenen Takt gebremst wird. Der physikalische Schaltkreis einer asynchronen Schaltung nimmt damit kleinere Dimensionen ein als der funktional gleichwertige synchrone. Denkt man nun vor allem an die Feldebene der Automation, wo zukünftig mehr und mehr Embedded Systems zu integrieren sind, wo man mit Stückzahlen in Größenordnungen von vielen  $10^{12}$  denkt, werden sie einen wichtigen marktwirtschaftlichen Faktor bilden.

Doch synchrone Schaltungen haben asynchrone nie ganz verdrängen können, es gab schon immer Bereiche, in denen sie vorteilhaft einzusetzen waren. Zum Beispiel kommt es häufig vor, Zählimpulse sehr hoher Taktrate zu erfassen. Da in einer synchronen Schaltung die Komponente taktbestimmend ist, die den höchsten Takt benötigt, müsste man Schaltungen dieser Art mit hohen Taktraten versehen, was bestimmte Technologien und entsprechende Maßnahmen voraussetzt. Man kann aber auch kombinierte Systeme realisieren mit asynchronen Zählern hoher Impulseingangsfrequenz und einer synchronen Folgeschaltung niedriger Taktrate. Liegt die Eingangsfrequenz des Datumwechsels von  $D_e$  sehr hoch, kann es wirtschaftlich sein, den ersten Teil des Zählers asynchron auszuführen und den zweiten synchron. Dann muss nicht die ganze Schaltung der hohen Frequenz von  $D_e$  gehorchen; es kann für die zweite Schaltung beispielsweise eine einfachere (preisgünstigere) Technologie verwendet werden.

# Entwurf asynchroner Schaltwerke

- (1) Das Schaltwerk ist so klein wie möglich zu realisieren!
- : möglichst alles synchron!
  - : möglichst wenig asynchron
  - : Modularisierung!



# Entwurf asynchroner Schaltwerke

- 2) Es gilt ansonsten eine entsprechende Vorgehensweise wie bei synchronen Schaltwerken.

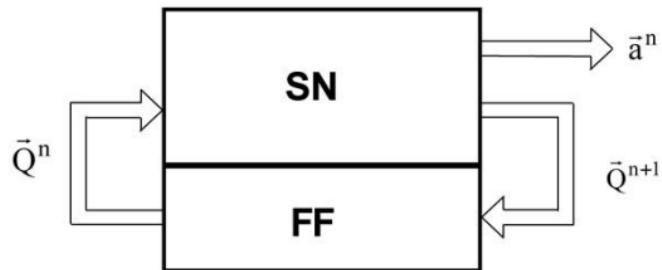
## ABER:

- (3) Der Clock ist eine Variable!
- (4) Die Anzahl der Zustände muss gerade sein.
- (5) Es muss davon ausgegangen werden, dass sich nur eine Eingangsvariable ändert.
- (6) Es darf sich nur eine Zustandsvariable ändern.
- (7) **ALLE** möglichen Übergänge sind zu analysieren (Bsp.: RS-FF).

# Autonome Schaltwerke

$$\vec{Q}^{n+1} = \vec{\alpha}(\vec{Q}^n)$$

$$\vec{a}^n = \vec{\beta}(\vec{Q}^n)$$



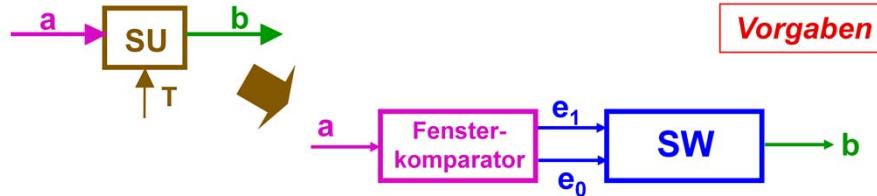
Keine Abhangigkeit von  $\underline{e}^n$

Warnung: nicht

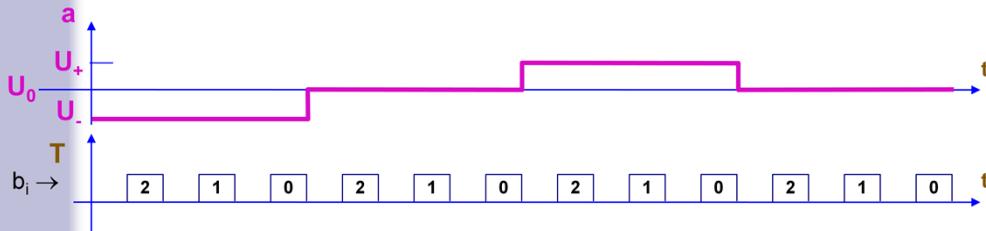
$\underline{e}^n = 0$  schreiben!

*Beispiel: Zahler, ..*

## Beispiel: Elimination redundanter Zustände



$$a = \{U_+, U_0, U_-\} \rightarrow \{e_1, e_0\} = \{01, 11, 10\}$$

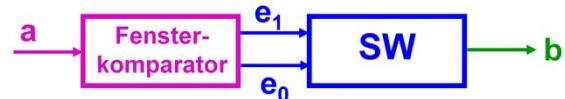


Bei der digitalen Signalübertragung werden oft mehrwertige Leitungssignale verwendet. Ein Signal mit beispielsweise drei unterschiedlichen Spannungswerten  $\{U_+, U_0, U_-\}$  ist Eingangsgröße  $e$  einer Schaltung  $S$  und soll in zwei aufeinanderfolgende binäre Signale  $(e_1, e_0)$  umgeformt und über die Ausgangsleitung  $a$  ausgegeben werden.

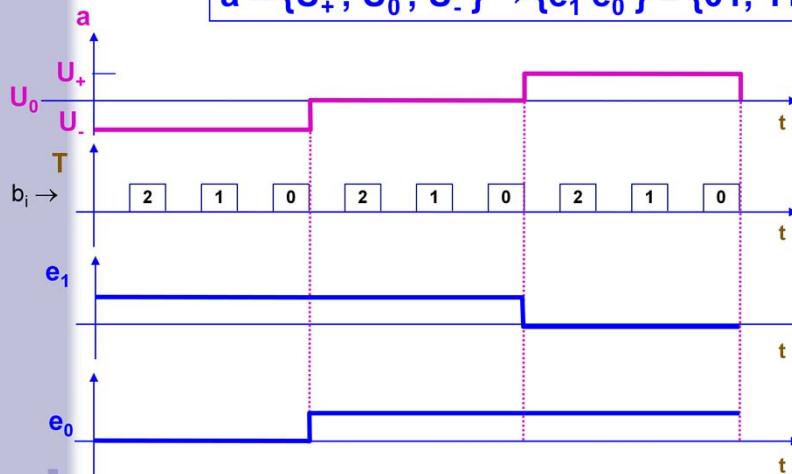
Das Timing-Diagramm zeigt, dass für die Übertragung der Ausgangsinformation noch ein Taktimpuls eingefügt ist, der jedoch hier nicht weiter von Interesse sein und stets den Wert 0 haben soll:  $b = \{b_2, b_1, b_0\} = \{b_2, b_1, 0\}$ . Die Umsetzung der drei analogen Eingangsspannungswerte erfolgt über einen Fensterkomparator, der als Blackbox anzusehen ist und hier ebenfalls nicht von Interesse ist. Seine Ausgänge sind  $e_1$  und  $e_0$ , die wiederum die Eingangswerte des zu entwickelnden Schaltwerkes  $SW$  sind.

## Beispiel: Elimination redundanter Zustände

Vorgaben



$$a = \{U_+, U_0, U_-\} \rightarrow \{e_1 e_0\} = \{01, 11, 10\}$$

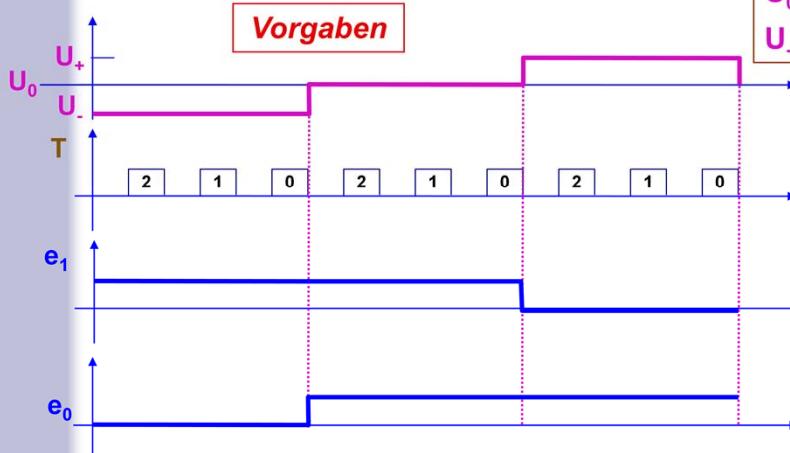
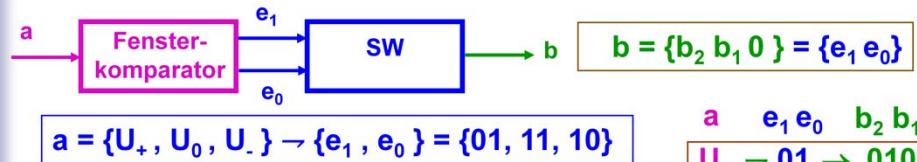


Die Codierung soll folgendermaßen aussehen:

$$\text{für } U_+ \rightarrow \{e_1 e_0\} = 01 \rightarrow b = \{b_2 b_1 0\} = 010,$$

$$\text{für } U_0 \rightarrow \{e_1 e_0\} = 11 \rightarrow b = \{b_2 b_1 0\} = 110,$$

$$\text{für } U_- \rightarrow \{e_1 e_0\} = 10 \rightarrow b = \{b_2 b_1 0\} = 100.$$



$a$	$e_1 e_0$	$b_2 b_1 0$
$U_+$	$01$	$010$
$U_0$	$11$	$110$
$U_-$	$10$	$100$

daraus  
ergeben  
sich 3  
Strände im  
Zustands-  
diagramm

Zeichne den Grafen mit:

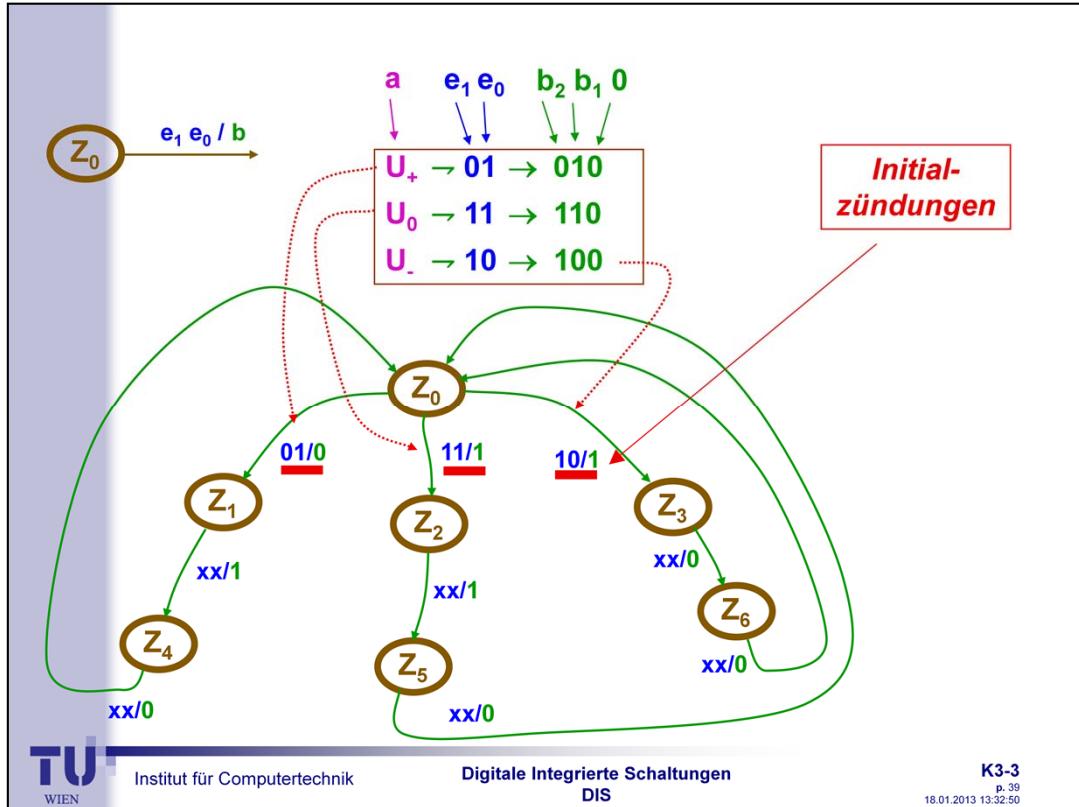


Die Codierung soll folgendermaßen aussehen:

für  $U_+ \rightarrow \{e_1 e_0\} = 01 \rightarrow b = \{b_2 b_1 0\} = 010$ ,

für  $U_0 \rightarrow \{e_1 e_0\} = 11 \rightarrow b = \{b_2 b_1 0\} = 110$ ,

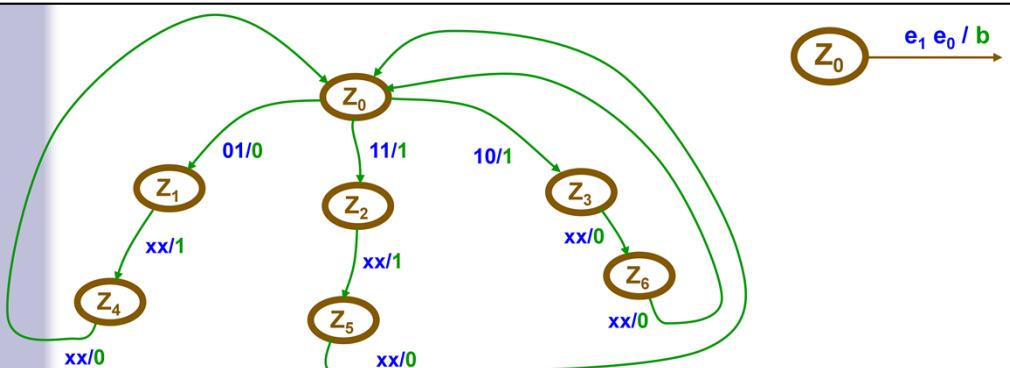
für  $U_- \rightarrow \{e_1 e_0\} = 10 \rightarrow b = \{b_2 b_1 0\} = 100$ .



## Lösung

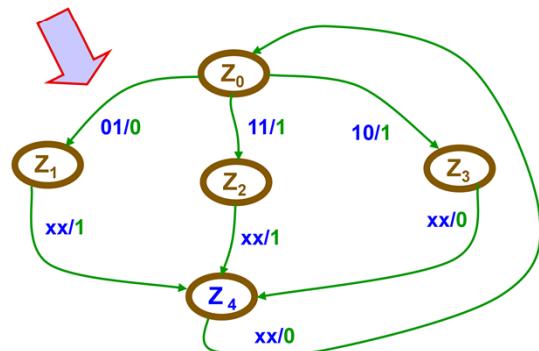
Einen möglichen Lösungsansatz zeigt die Grafik. Aus drei unterschiedlichen Spannungswerten resultieren konsequent drei Pfade. Es ist zu erkennen, dass die drei Kanten, die in  $Z_0$  münden, identische Bezeichnungen tragen, was eine Zusammenfassung der Knoten  $Z_4$ ,  $Z_5$  und  $Z_6$  erlaubt.

Zusatzbemerkung: Auf dieser Diagrammbasis funktionierten die ersten SPS'en (Speicher-Programmierbaren Steuerungen).

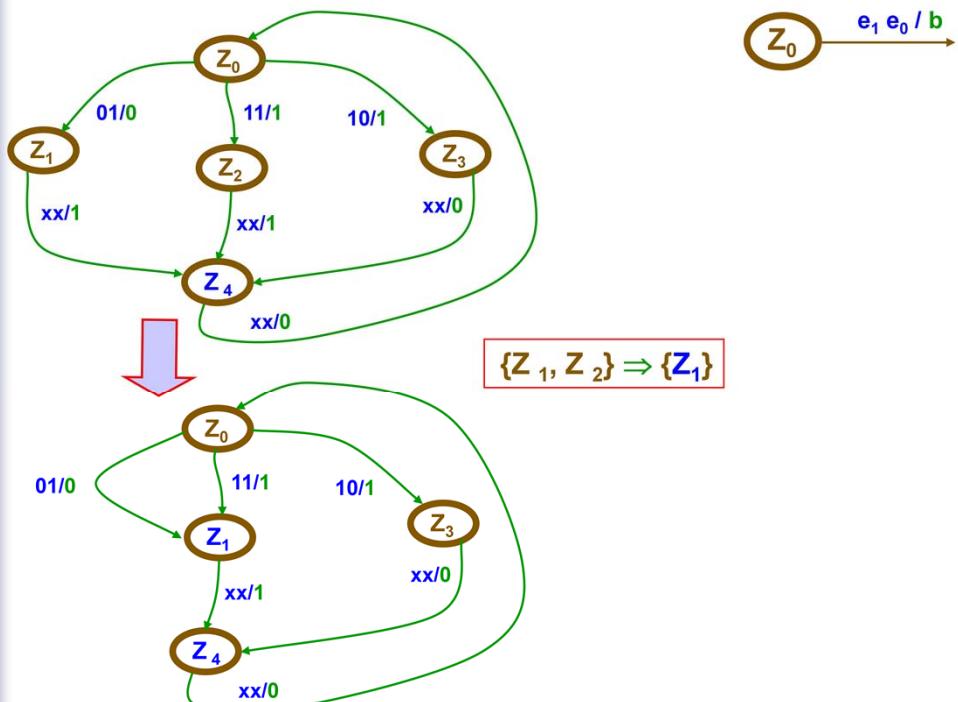


$Z_0 \xrightarrow{e_1 e_0 / b}$

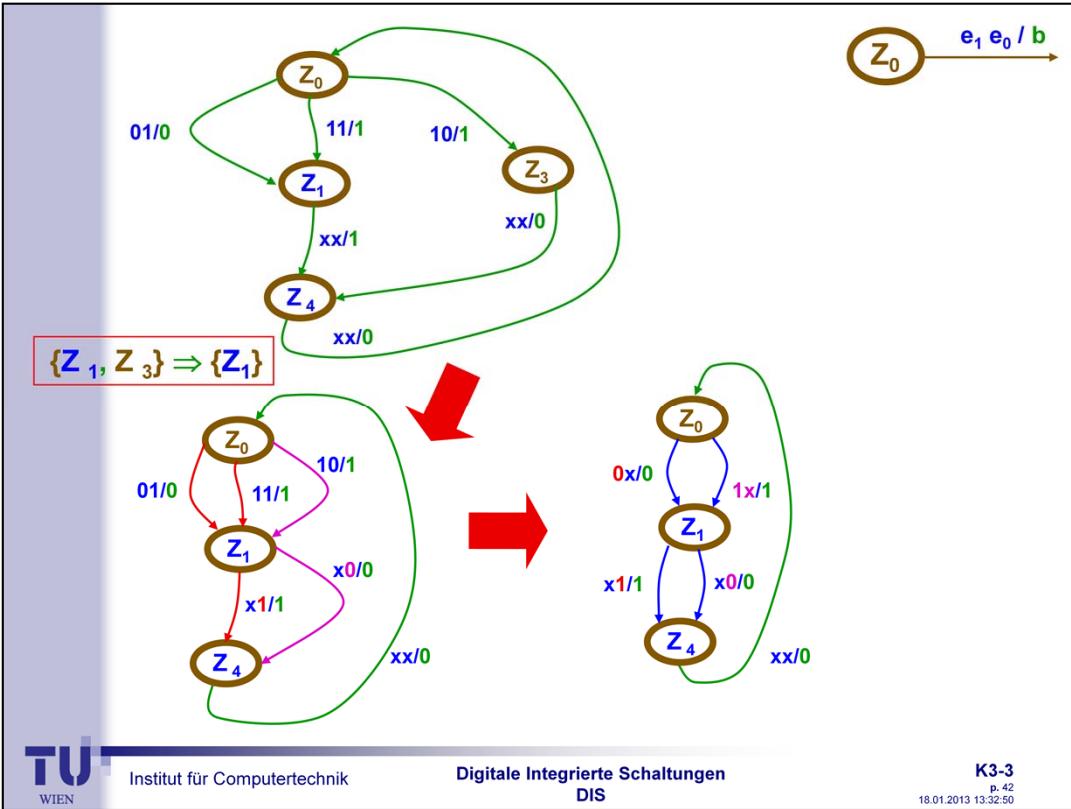
$\{Z_4, Z_5, Z_6\} \Rightarrow \{Z_4\}$



Ebenfalls redundant sind die Knoten  $Z_1$  und  $Z_2$ , da ihre abgehenden Kanten in den gleichen Zielknoten  $Z_4$  münden und diese Kanten wiederum die selbe Beschriftung aufweisen. Man erhält damit einen Grafen mit 4 Knoten.



Im vorliegenden Fall kann man noch einen Schritt weitergehen, wobei im Gegensatz zu bisherigen Beispielen die Anzahl aller möglichen Zustände des Systems nicht verringert wird (man kommt nicht unter die Anzahl von 4). Man kann die Anzahl der inneren Zustände (gespeichert in  $Z_i$ ) verringern, indem man die Menge der äußeren Zustände erhöht. So können in Bild oben die Zustände  $Z_1$  und  $Z_3$  zusammengefasst werden (Reduktion um einen inneren Zustand), es muss dafür aber den Übergängen  $Z_1 \rightarrow Z_4$ :  $xx/1$  und  $Z_3 \rightarrow Z_4$ :  $xx/0$  in Bild unten eine zusätzliche Eingangsinformationsmenge zugewiesen werden, was ..



.., was unten im Bild oben zu den Übergängen  $Z_1 \rightarrow Z_4$ :  $x1/1 \vee x0/0$  führt. Ob der Übergang von vier auf drei innere Zustände zu einer tatsächlichen schaltungstechnischen Vereinfachung führt (hinsichtlich der Reduzierung von Schaltungskomponenten), wäre noch zu untersuchen, doch ist die weitere Schaltwerksentwicklung dieser speziellen Aufgabe hier nicht mehr von Interesse.

Vorsicht: Die Auswertung einer zusätzlichen Information (im Bild unten links) bedeutet, dass das Systemverhalten geändert wird

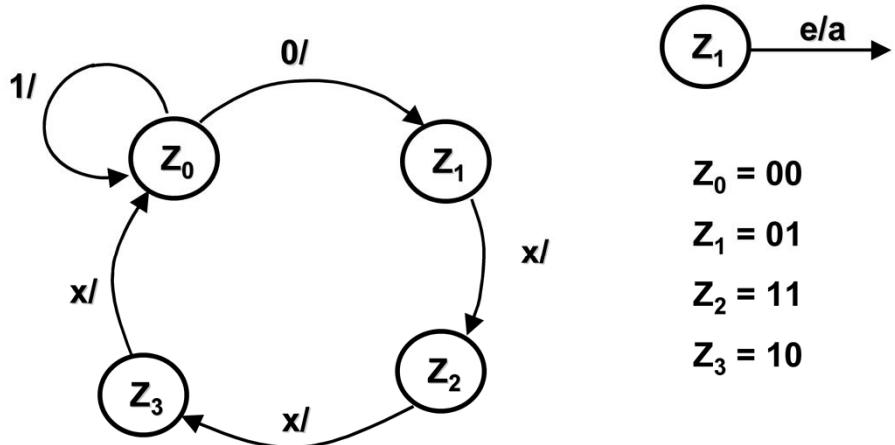
## Beispiel: Elimination redundanter Zustände

# VORSICHT!

Reduzierung von Zuständen bringt nicht immer eine  
Schaltungsoptimierung!

Durch Reduktion von Zuständen kann das Verhalten der  
Schaltung geändert werden – siehe letztes Beispiel!

# Codierungsoptimierung



.. klar: Gray-Code .. wann immer möglich ..

Hat man im Rahmen einer Schaltwerksentwicklung einen Zustandsgrafen entworfen, bieten sich im Allgemeinen unterschiedliche Kodierungsmöglichkeiten für die Zustände an. Nun gibt es keine prinzipiellen Verfahren, über die man generell die Schaltung mit dem geringsten Gatteraufwand gewinnt. Möchte man einen Zähler realisieren, der jeweils nur inkrementiert, ist es plausibel, den Gray-Code zu verwenden, da in diesem Fall ein gleichzeitiger Übergang von mehreren Einsen auf mehrere Nullen vermieden wird. Die freie Wahl der Codierung und damit eine mögliche Vereinfachung durch eine entsprechend geschickte Codierung hinsichtlich der Optimierung ist allerdings damit ausgeschlossen. Unabhängig von der Entwicklung eines Zählers, ist dieses Prinzip im Allgemeinen immer zu empfehlen, solange keine Zustände in Randbedingungen nicht definiert sind (also "übrig bleiben").

Die Ausgangsgrößen sind in diesem Beispiel nicht von Interesse.

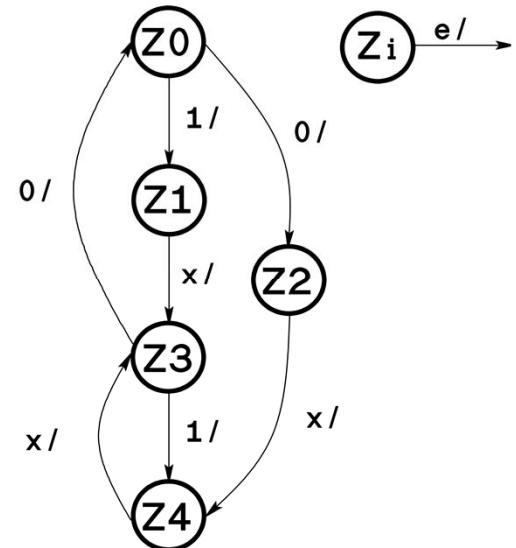
# Codierungsoptimierung

Was passiert mit

$Z_5$ ?

$Z_6$ ?

$Z_7$ ?



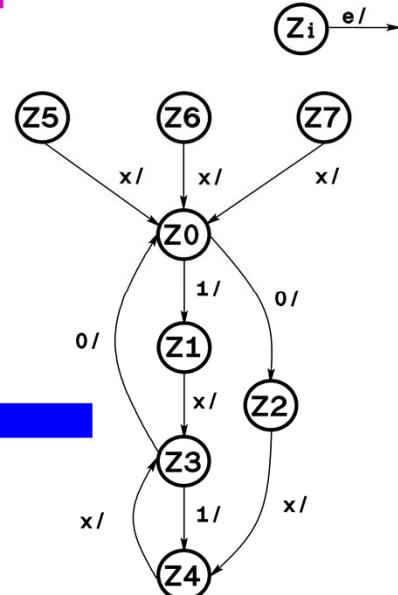
Betrachtet man dagegen ein beliebiges Schaltwerk, ist die Randbedingung einer speziellen Codierung also nicht gefordert, und es kommt nun auf das Geschick des Entwicklers an, die Codierung so zu wählen, dass man zu einem reduzierten Schaltungsaufwand gelangt.

# Möglichkeit 1:

Erzwungener Übergang  
in den  
Ausgangszustand

Nachteil:

- keine Optimierung
- Fehlerverschleierung



*Möglichkeit 1:* Zu einem häufig anzutreffenden Schaltungsentwurfsfehler führt die Überlegung, die redundanten (nicht notwendigen) Zustände in den Ursprungszustand münden zu lassen. Sollte einmal aufgrund eines Fehlverhaltens des Systems einer dieser Zustände anspringen, wird sich das System spätestens nach einem Takt im vorgeschriebenen Zustandszyklus wiederfinden. Das aber ist nicht sinnvoll, da ein derartiger Fehler im Allgemeinen nach außen ein Fehlverhalten des Systems bewirkt und somit auf jeden Fall vermieden werden muss.

# Möglichkeit 2:

## Volloptimierung (ohne Randbedingungen)

gesetzt wird:

i	t <sup>n</sup>				t <sup>n+1</sup>				
	e	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Z <sub>i</sub>	Z <sub>i</sub>	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>
1	0	0	0	0	0	2	0	1	0
2	0	0	0	1	1	3	0	1	1
3	0	0	1	0	2	4	1	0	0
4	0	0	1	1	3	0	0	0	0
5	0	1	0	0	4	3	0	1	1
6	0	1	0	1	5	x	x	x	x
7	0	1	1	0	6	x	x	x	x
8	0	1	1	1	7	x	x	x	x
9	1	0	0	0	0	1	0	0	1
10	1	0	0	1	1	3	0	1	1
11	1	0	1	0	2	4	1	0	0
12	1	0	1	1	3	4	1	0	0
13	1	1	0	0	4	3	0	1	1
14	1	1	0	1	5	x	x	x	x
15	1	1	1	0	6	x	x	x	x
16	1	1	1	1	7	x	x	x	x

$$Z_0 = Q_2 Q_1 Q_0 = 000$$

$$Z_1 = Q_2 Q_1 Q_0 = 001$$

.. = ..

$$Z_7 = Q_2 Q_1 Q_0 = 111$$

.. und dabei alle Zustandskombinationen durchspielen ..

.. kann sehr aufwendig werden ..



*Möglichkeit 2:* Eine Möglichkeit, um zu einer "minimalen Schaltung" zu kommen, besteht darin, alle möglichen Codekombinationen, die zu einer Lösung führen, durchzuspielen, was bei kleinen Schaltungen schnell durchgeführt ist, jedoch bei größeren Schaltwerken relativ aufwendig werden kann.

Legt man für die redundanten Zustände keine Randbedingungen fest, werden deren Übergänge durch die Wahl der Codierung und die Art der Optimierung bestimmt. Wählt man für das Beispiel beispielsweise die Zuordnung:

$$Z_0 := Q_2 Q_1 Q_0 = 000$$

$$Z_1 := Q_2 Q_1 Q_0 = 001$$

..

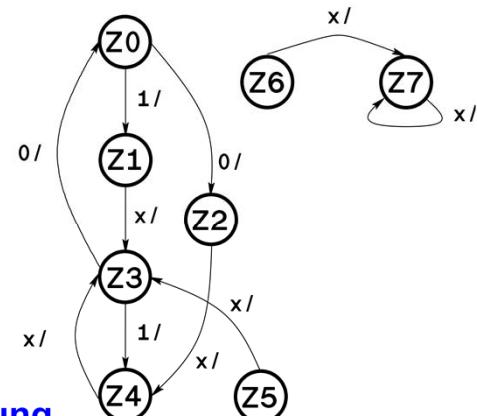
$$Z_7 := Q_2 Q_1 Q_0 = 111$$

erhält man die Übergangstabelle oben und nach der Optimierungsberechnung den Zustandsgrafen der folgenden Darstellung.

# Möglichkeit 2:

## Volloptimierung (ohne Randbedingungen)

Ergebnis:



Nachteil:

- **Z<sub>5</sub>: Fehlerverschleierung**
- **nicht gewünschtes Verhalten im Fehlerfall**



Er beinhaltet den grundsätzlichen Nachteil, dass nicht vorhergesagt werden kann, wie die "übrigen" Zustände, hier also die Zustände Z<sub>5</sub> bis Z<sub>7</sub> über die Optimierung platziert werden, und welchen Einfluss dies auf das Systemverhalten hat. Zum Beispiel könnte hier kritisch angemerkt werden, dass nach einer fehlerhaften Einnahme der Zustände Z<sub>6</sub> oder Z<sub>7</sub> das System nur noch über einen Reset in den gewünschten Ablauf überführt werden kann (der fehlerhafte Übergang in die Zustände Z<sub>6</sub> bzw. Z<sub>7</sub> führt somit in jedem Fall zum "Systemabsturz").

Eine andere Vorgehensweise ist in vielen Fällen effizienter. Gemeint ist die Suche nach einem Kompromiss. Einerseits soll eine schaltungstechnisch möglichst minimale Lösung gefunden werden, andererseits soll ein modifizierter Graf nach Bild oben erarbeitet werden, bei dem nach Einnahme der "verbotenen" Zustände das System nicht "in jedem Fall" abstürzt, sondern Fehlermeldungen absetzt oder ähnliches.

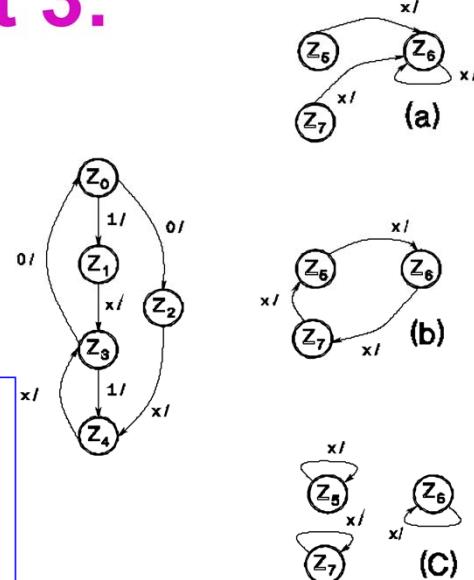
# Möglichkeit 3:

## Sicherheit (Safety)

- **Z<sub>5</sub>, Z<sub>6</sub>, Z<sub>7</sub> werden im Grafen vorgegeben**
- **hier 3 Varianten: a bis c**

### Vorteil:

- **eindeutiges Zustandsverhalten:**
  - (a) ein Endzustand
  - (b) Endlosschleife
  - (c) drei Einzelzustände



*Möglichkeit 3:* In vielen Fällen wird jedoch direkt gewünscht, dass ein Fehler prinzipiell zu einem gezielten Verhalten führt (ein Fehler in einem Rechner, der Bankkonten verwaltet, darf nicht verschleiert werden, sondern muss aufgedeckt werden; der Fehler sollte eruierbar sein). Lösungen können sein, dass fehlerhafte Übergänge über eine spezielle Ausgangsleitung zu einer Fehleranzeige führen. Lösungen können aber auch sein, dass darüberhinaus die Zustandsmaschine in einem bestimmten Zyklus oder sogar in einem bestimmten Zustand verharren bleibt. Beispiele zeigt Bild oben.

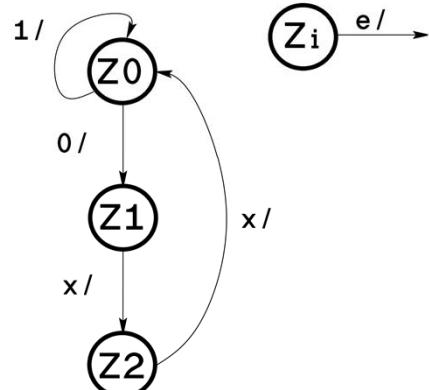
So springt in Bild (a) oben das System in den Zustand Z<sub>6</sub>, wenn fälschlicherweise einer der drei Zustände Z<sub>5</sub> bis Z<sub>7</sub> eingenommen wird. Im Beispiel (b) verharren sie in dem Zyklus Z<sub>5</sub>, Z<sub>6</sub> und Z<sub>7</sub>, und in Bild (c) bleibt das System in dem Zustand, in den es fälschlicherweise übergeht.

Es ist einsichtig, dass in solchen Lösungen nach Möglichkeit 3 die resultierenden Booleschen Ausdrücke komplexer sind als in den Varianten vorher.

# Möglichkeit 4:

## Zustandsverschmelzung

vorgegebener Graf:



Basisüberlegung:

$$Z_0 = Q_1 Q_0 = 00$$

$$Z_1 = Q_1 Q_0 = 01$$

$$Z_2 = Q_1 Q_0 = 11$$

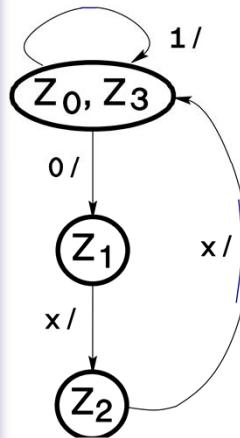
$$Z_3 = Q_1 Q_0 = 10$$

damit wird möglich:

$$Z_0 = Z_3 = Q_1 Q_0 = x0$$

# Möglichkeit 4:

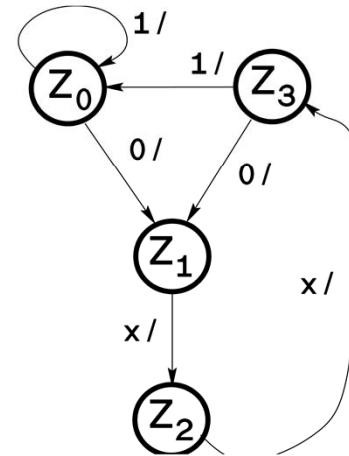
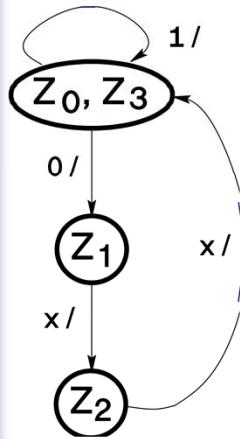
## Zustandsverschmelzung



e	t <sup>n</sup>			t <sup>n+1</sup>		
	Z <sub>i</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Z <sub>i</sub>	Q <sub>1</sub>	Q <sub>0</sub>
0	0	0	0	1	0	1
0	1	0	1	2	1	1
0	2	1	1	0,3	x	0
0	3	1	0	1	0	1
1	0	0	0	0,3	x	0
1	1	0	1	2	1	1
1	2	1	1	0,3	x	0
1	3	1	0	0,3	x	0

# Möglichkeit 4:

## Zustandsverschmelzung



Dann kann man auch formulieren:

$$Z_0 = Z_3 = Q_1 \quad Q_0 = x0,$$

Den daraus berechneten Grafen rechts im Bild weist nach außen hin kein anderes Systemverhalten auf.

## Beispiel: Reduzierung der PIN-Anschlüsse

AzÜ wnv



Es geht nur um die  
Festlegung  
der Adressen über das  
"Mäuseklavier"...

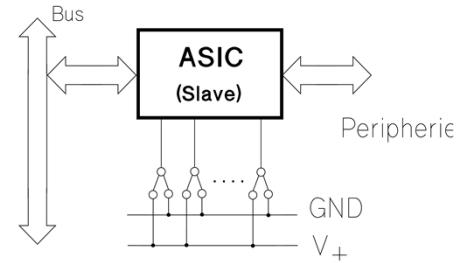
Anzahl Adressen  
 $m = \text{round}(0,49 + \lceil \lg n \rceil)$  für  $n > 1$

Pin-Anzahl

Zweiwert-Logik

Bsp.: für 4 Adressen 2 Pins

für 9 Adressen 4 Pins



$$\text{round}(0,49 + \lceil \lg n \rceil) \equiv \text{ceiling}(\lceil \lg n \rceil)$$



### Reduzierung der Pin-Anschlüsse

Für ein Bussystem ist ein Slave als ASIC zu entwickeln. Zur Identifikation wird jedem Slave eine Adresse zugewiesen, die über Lötbrücken und über die Anschlüsse des ASIC's (Pins) der internen Schaltung des ASIC's mitgeteilt wird. Nachteil dieses Prinzips ist es, dass für  $n$  Adressen  $m$  Anschlüsse am ASIC vorgesehen werden müssen.

Es gilt:  $\lceil \lg x \rceil = \lg x / \lg 2$ .

$\text{ceiling}(y)$ : für  $y$  wird der kleinste Integer-wert gesetzt, der  $\geq y$  ist.

## Reduzierung der PIN-Anschlüsse

$$m = \text{round}(0,49 + \lceil \log_2 n \rceil) \quad \text{für } n > 1$$

bei 8 Pins  $n_{2W} = 2^8 = 256$   
aber:

bei 8 Pins  $n_{3W} = 3^8 = 6561$

bessere Lösung: ternäre Logik

aber NICHT:

00B	$V_-$
01B	$V_0$
10B	$V_+$

.. besser in den **Zeitbereich** ausweichen,

denn Zeit spielt hier keine Rolle (*Initialisierungsphase*)

Für die Adressierung von angenommenen 16 Slaves müssen dementsprechend 4 Pins gespendet werden. Pins kosten jedoch relativ viel Platz und damit bei einem ASIC massiv Geld. Geht man nun von einer Mehrwertlogik aus, beispielsweise von einer Dreiwertlogik, berechnet sich die Anzahl der Pins über die Gleichung oben.

Mit 4 Pins lassen sich somit schon  $n = 4^3 = 64$  anstatt 16 Adressen definieren. Setzt man 8 Pins voraus, ist der Unterschied noch markanter, 6561 anstatt 256. Wechselt man zur Vierwertlogik werden die Verhältnisse noch dramatischer. Doch bleiben wir bei der Dreiwertlogik.

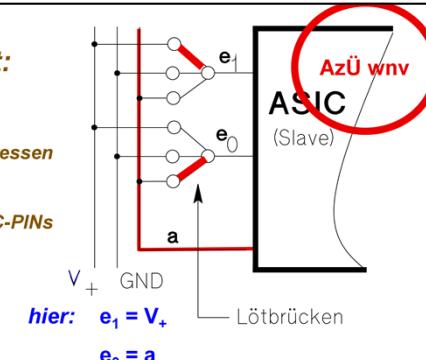
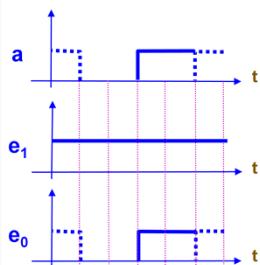
## Eine Lösungsmöglichkeit:

$$n_{i,m} = (2 + i)^{m+i}; m \geq 2, i \geq 0$$

*n: Anzahl der möglichen Adressen*

*i: Anzahl der Ausgänge a*

*m: Anzahl der Adressen-ASIC-PINS*



			e <sub>1</sub>	e <sub>0</sub>
1	t <sub>1</sub>	a wird auf 0 gesetzt		
	t <sub>2</sub>	Abfrage aller Eingänge	1	0
2	t <sub>3</sub>	a wird auf 1 gesetzt		
	t <sub>4</sub>	Abfrage aller Eingänge	1	1

Um drei unterschiedliche Werte pro Zugriff einlesen zu können, muss man jedoch nicht gleich drei unterschiedliche Spannungs- oder Stromwerte definieren (wie beispielsweise  $V_-$ , GND und  $V_+$ ). Man kann auch in den Zeitbereich ausweichen, was bedeutet, man liest den Wert nicht zu einem Zeitpunkt ein, sondern in Folge (wodurch auch eine Vierwert- oder sogar Fünfwertlogik einfacher wird).

Am Adressvorgabeeingang werden zum einen die beiden Potentialwerte GND und  $V_+$  (logische Werte 1 und 0) angeboten, zum anderen auf einen speziellen Ausgang a des ASIC's, an dem definiert 0-1-Potentialwechsel erzeugt werden. Ob der Eingang  $e_i$  an GND, an  $V_+$  oder an a liegt, stellt der ASIC dadurch fest, dass er zweimal abfragt und zwischen den beiden Abfragen die Ausgangsgröße a einmal negiert wird. Ist er an GND oder an  $V_+$  angeschlossen, erhält der Eingang für beide Abfragen die gleiche Information; ist die Lötbrücke an die Leitung a angebunden, wechselt die eingeholte Information im Gleichtakt zu Ausgang a. Über einen Pin kann damit die Informationsmenge einer Dreiwertlogik eingeholt werden, obwohl die Zweiwertlogik (binäre Technik) zugrundegelegt ist.

Entsprechend könnte man nun im Sinne einer Vier- oder Fünfwertlogik verfahren. Wird nämlich nicht nur ein Ausgang a, sondern eine beliebige Anzahl von Ausgängen angenommen, gelangt man zum Formalismus:

$$n_{i,m} = (2 + i)^{m+i}, \quad \text{mit } m \geq 2, i \geq 0 .$$

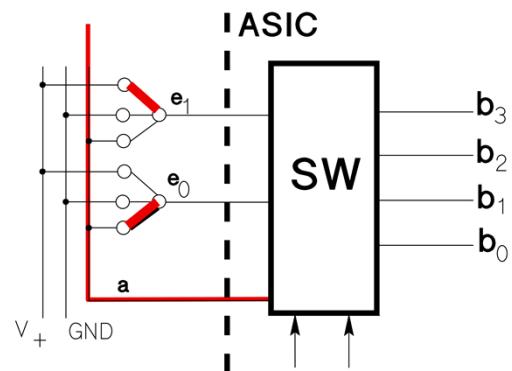
Schon kleine Rechenbeispiele zeigen, dass man für  $i = f(m)$  für konstante Werte  $n_{i,m}$  jeweils ein Maximum erhält. Zu berücksichtigen ist dabei allerdings, dass die Anzahl der Abfragezyklen linear mit i zunimmt und der Schaltungsaufwand sich gleichfalls erhöht, was in der Praxis jedoch im Allgemeinen eine untergeordnete Rolle spielt. Doch sei das hier nicht von Interesse. Im Folgenden soll es hier darum gehen, wie man solche Schaltungen auf einfachste Weise realisiert.

*Beispiel:* Angenommen sei ein Ausgang a, eine Schaltungskonfiguration und ein Timing-Diagramm nach Bild oben. Die Adressenabfrage erfolgt in vier Schritten. Das Modul erkennt, dass an  $e_1$  der logische Wert 1 und an  $e_0$  am Ausgang a anliegt. Das Schaltwerk ist so zu berechnen, dass für den Zeitpunkt  $t_1 > 0$  stets die Adresse  $b = \{b_3 b_2 b_1 b_0\}$  konstant anliegt, bis die Versorgungsspannung abgeschaltet wird (der Reset-Eingang interessiert hier nicht, das Timing in Bild oben soll nur das Prinzip verdeutlichen und muss nicht exakt eingehalten werden).

werden.

## 2-Schritt-Adresserkennung:

		$e_1$	$e_0$
1	$t_1$ a wird auf 0 gesetzt		
	$t_2$ Abfrage aller Eingänge	1	0
2	$t_3$ a wird auf 1 gesetzt		
	$t_4$ Abfrage aller Eingänge	1	1



Prinzipielles  
Verhalten:



Mehrere Lösungen sind möglich

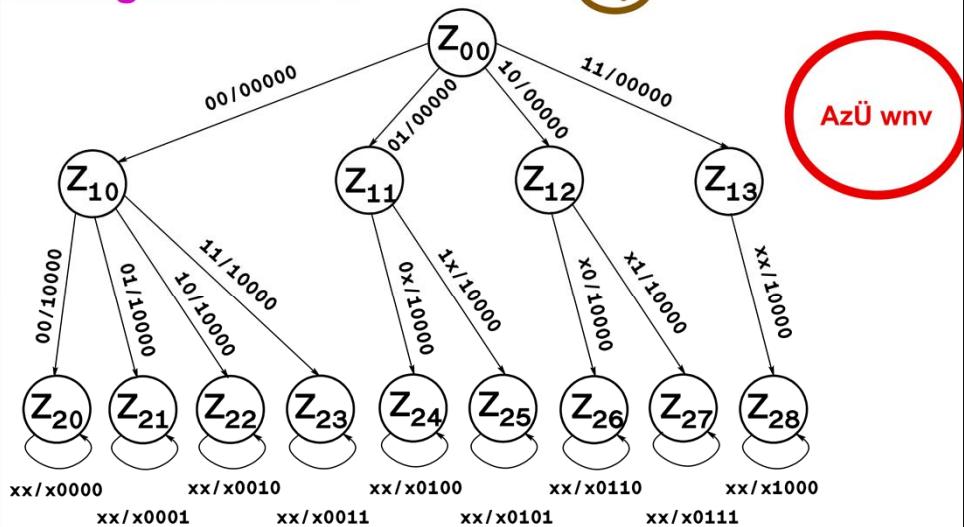
AzÜ wnv

## Lösung

Die Aufgabe ist hinreichend verbalisiert, so dass direkt die Formalisierung (soweit nicht schon durch die Aufgabe selbst vorgenommen) durchgeführt werden kann. Mehrere Lösungen bieten sich an, wobei vom Timing her folgendes Prinzip sinnvoll ist:

Das Setzen des Ausgangs a und die Abfrage des Eingangs e kann im Laufe eines Zustandswechsels  $Z_0 \rightarrow Z_i$  erfolgen. Ebenso kann das folgende Setzen von a = 1 und die nochmalige Abfrage der Eingänge e definiert werden, was dann im Übergang  $Z_i \rightarrow Z_j$  erfolgt.

## Lösungsvariante 1



➤ 9 mögliche Adressen

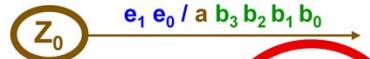
Mehrere Lösungen sind denkbar, von denen 3 angesprochen werden sollen. Prinzipiell soll davon ausgegangen werden, dass der Ausgang a zunächst von  $Z_0 \rightarrow Z_i$  auf 0 und von  $Z_i \rightarrow Z_j$  auf 1 gesetzt wird.

### Lösungsvariante 1

Ausgehend vom Ursprungszustand  $Z_{00}$  ergeben sich bei 2 Eingängen 4 Übergangsmöglichkeiten. Entsprechendes gilt für die Übergänge  $Z_{10} \rightarrow \{Z_{20}, Z_{21}, Z_{22}, Z_{23}\}$ . Ausgehend von den Zuständen  $Z_{11}$  und  $Z_{12}$  sind hier jeweils nur noch 2 Übergänge möglich, denn a war vorher 0 und ist im zweiten Schritt 1. Dann kann für den Übergang von  $Z_{11}$  in den Folgezustand  $Z_j$  die Eingangskombination  $\{e_1 e_0\} = \{00\}$  und  $\{e_1 e_0\} = \{10\}$  nicht mehr auftreten. Das gleiche gilt für den Übergang  $Z_{12}$  nach  $Z_j$ . Die entsprechende Überlegung führt zum Übergang von  $Z_{13} \rightarrow Z_{28}$ .

Diese Lösungsvariante hat zum Vorteil, dass nach einer einmalige Adressfindung das System im Endzustand verharret. Eine weitergehende Änderung über die Eingänge (ob es eine kurzzeitige Störung ist oder sogar ein anhaltende) bewirkt nichts; das System behält die Adressinformation.

## Lösungsvariante 2



$$\{Q_3 \ Q_2 \ Q_1 \ Q_0\} = \underline{b}$$

Ausgänge = Zustände

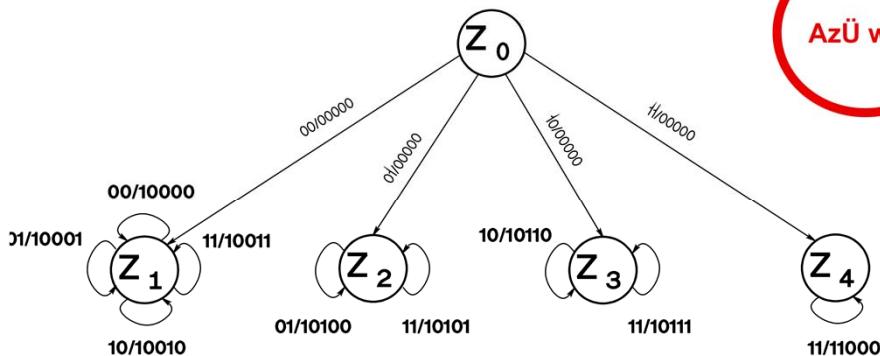
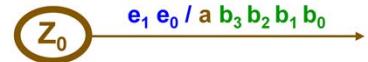
AzÜ wnv

$t^n$			$t^{n+1}$
$e_2 \ e_1$	$Z^v$	$Q_3 \ Q_2 \ Q_1 \ Q_0 = b_3 \ b_2 \ b_1 \ b_0$	$Q_3 \ Q_2 \ Q_1 \ Q_0$
0 0	$Z_{00}$	1 0 0 1 = 1 0 0 1	1 0 1 0
0 1	$Z_{00}$	1 0 0 1 = 1 0 0 1	1 0 1 1
1 0	$Z_{00}$	1 0 0 1 = 1 0 0 1	1 1 0 0
1 1	$Z_{00}$	1 0 0 1 = 1 0 0 1	1 1 0 1
0 0	$Z_{10}$	1 0 1 0 = 1 0 1 0	0 0 0 0
0 1	$Z_{10}$	1 0 1 0 = 1 0 1 0	0 0 0 1
1 0	$Z_{10}$	1 0 1 0 = 1 0 1 0	0 0 1 0
1 1	$Z_{10}$	..	..
0 x	..	..	..

## Lösungsvariante 2

Der Ausgangsvektor  $b$  wird nicht in die Berechnung des Automaten mitaufgenommen, sondern einfach direkt über die FF-Ausgänge  $\{Q_3 \ Q_2 \ Q_1 \ Q_0\}$  abgeleitet. Festgelegt werden könnte beispielsweise eine Zuordnung gemäß Bild oben. Damit fällt eine zusätzliche Ausgangsbeschaltung weg. Ob das in jedem Fall zu einem minimalen Schaltwerk führt, kann nicht generell beantwortet werden und muss von Fall zu Fall untersucht werden. Hier soll es nicht weiter interessieren.

## Lösungsvariante 3



- Vorteil: sehr geringer Aufwand
- schwerwiegender Nachteil: für  $t > 0$  störungsempfindlich

*Lösungsvariante 3:*

Geht man davon aus, dass sich für  $t_1 > 0$  die Eingangswerte  $\underline{e} = \{e_1 e_0\}$  nicht mehr ändern (Störungen also mit einer hohen Wahrscheinlichkeit nicht auftreten), kann auf der Basis der Eliminierung redundanter Zustände (siehe entsprechende Aufgabe vorher) ein Zustandsgraf mit weniger Zuständen entwickelt werden, was die Anzahl der FFs natürlich einschränken hilft.

Zu ermitteln wäre nun noch, ob eine geschicktere Codierung zu einer Lösung mit weniger Schaltungsaufwand führt, was hier jedoch nicht untersucht wird.

Zusätzliche Bemerkungen:

- (1) Welche Lösung die minimale Anzahl logischer Verknüpfungen liefert, ist nur nach Ermittlung der Lösungen zu sagen.
- (2) Für ASIC-Entwicklungen ist die Lösungsvariante 3 nicht so geschickt, da über die Eingänge  $\{e_1 e_0\}$  Störungen eingefangen werden können, was im Allgemeinen zu einer Fehlfunktion des Systems führt. Sind in den Lösungsvarianten 1 und 2 einmal die Zustände  $Z_{20}, Z_{21}, \dots, Z_{28}$  erreicht, haben Störgrößen auf den Leitungen  $\{e_1 e_0\}$  weitgehend keine Wirkung mehr auf das Systemverhalten.

# Digitale Integrierte Schaltungen

384.086

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

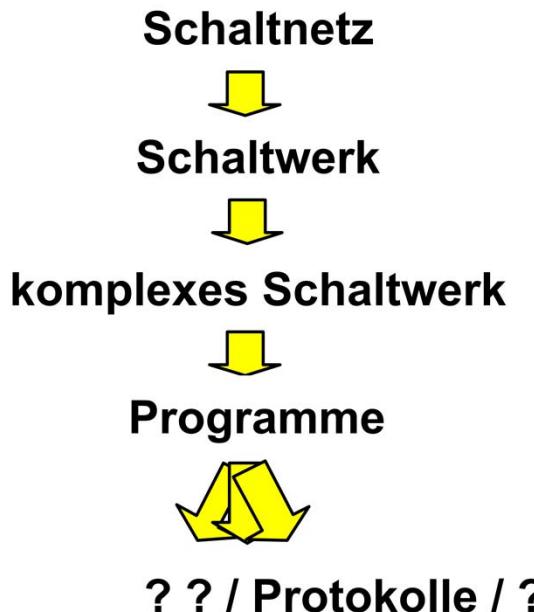
[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



# Kapitel 4

## Schaltwerke hoher Komplexität

setzt auf Schaltnetze und Schaltwerke auf  
schlägt effiziente Strukturen vor



Nachdem Schaltnetze und Schaltwerke behandelt sind, muss konsequenterweise die Thematik der Schaltkreise höherer Komplexität folgen. Selbstverständlich ist es denkbar, von der Black Box ausgehend, jedes System als Mealy- oder Moore-Automaten zu realisieren. Wird jedoch die Anzahl der Ein- und Ausgänge zu hoch (wie beispielsweise bei einem Mikroprozessor), ist der Automat kaum mehr berechenbar, und es muss zu anderen Methoden übergegangen werden.

Von diesem Gedanken ausgehend, kann man daraus schließen, was hierarchisch als nächst höherer Level angesehen werden kann: Programme, die die komplexen Schaltwerke steuern. Dann könnte man als nächst höheren Level z. B. Protokolle sehen, mit denen die komplexen Einheiten untereinander kommunizieren usw.

Jedem von uns ist klar, was Hardware und was Software ist. Doch wie kann man den Grenzbereich definieren? Wo liegen die entscheidende Differenzierungsunterschiede? Wenn die Software aus der Hardware hervorgeht, oder aus einem anderen Blickwinkel her betrachtet, wenn die Software auf die Hardware aufsetzt, muss dies Konsequenzen für die Software haben. Eines ist dabei gewiss, die Hardware setzt der Software Grenzen. Das Thema wird im Kapitel 4 noch mehrfach angesprochen. Es wird in der Automation und vor allem im Bereich der Artificial Intelligence (AI) zunehmend wichtiger. Dort taucht die Frage auf: Wo ist die Schnittstelle zwischen Nervensystem und Psyche. Ein sehr spannendes Gebiet, was direkt mit diesem Thema verknüpft ist.

# Schaltwerke mit hoher Komplexität

*Annahme: mit wenigen Automaten nicht effizient zu realisieren*

## ? Effizienz ?

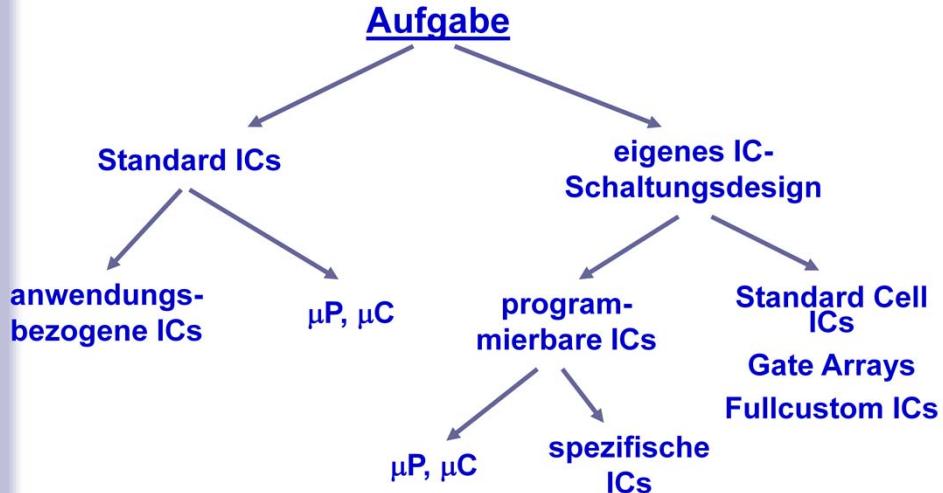
- ❖ **Formalisierung wird**
  - **zu schwierig (unübersichtlich)**
  - **zu komplex ( $Z_n$  mit  $n_{\max}$ : zu groß)**
  - ...
- ❖ **die Physik lässt es nicht zu**
  - **zu lange Leitungen**
  - **FPGA-Problematik**
  - ...



Asynchrone, komplexe Schaltwerke, also Schaltwerke, die auf asynchronen Automaten basieren, sind denkbar, aber mit den heute zur Verfügung stehenden Methoden noch nicht effizient zu realisieren, doch in der Forschung wird fest damit gerechnet, dass es bald soweit sein wird. Die folgenden Betrachtungen gehen deshalb (fast) nur von synchronen Systemen aus, wobei gemeint ist, dass ein System nur einen oder mehrere Systemtakte hat, die Module also nach außen hin synchron wirken. Selbstverständlich ist, dass bestimmte Submodule intern asynchrone Schaltwerke besitzen können. Diese Restriktion ist notwendig, da jede Urzelle eines synchronen FFs ein asynchrones Schaltwerk darstellt, beziehungsweise da bestimmte Schaltungen, wie beispielsweise serielle Hochgeschwindigkeitsschnittstellen, asynchrone Schaltwerke voraussetzen.

Wichtig dabei ist für den Elektrotechniker: synchronen Schaltungen stellen eine Abstraktion asynchroner Schaltungen dar und sind eine vereinfachte Beschreibungsmethode. Synchronen Schaltungen lassen sich immer als asynchrone beschreiben – was auch z. B. bei der Störungssuche oft notwendig wird.

## Schaltwerke mit hoher Komplexität: Lösungsmöglichkeiten



Zwei Möglichkeiten bieten sich an: Man sucht für seine Lösung einen oder mehrere Standardbausteine, beispielsweise entsprechende Mikroprozessoren, oder man entwickelt eigene ICs (die natürlich ebenfalls Mikroprozessoren sein können). Für die Umsetzung der selbst entworfenen Schaltungen kommen wiederum zwei Möglichkeiten in Betracht. Die einfachere Möglichkeit ist die Verwendung programmierbarer Bausteine (PLDs, FPGAs, ..), von denen sich einige auch für die Entwicklung anwendungsspezifischer Mikroprozessoren eignen. Die aufwendigere Variante, die mit höheren Entwicklungskosten verbunden ist, werden entsprechende ASICs auf der Basis eines eigenen Entwurfs produziert (Standard-Cell Arrays, Gate Arrays, Fullcustom ICs, ..). Doch selbst diese zweite Möglichkeit kann, und das war vor ein paar Jahren noch anders, auch für sehr kleine Stückzahlen wirtschaftlich interessant sein. Das Bild hat sich somit für den Entwickler völlig geändert: Er steht nicht mehr nur vor der Alternative, entweder Standard-TTL-Bausteine oder Mikroprozessoren zu verwenden, sondern der Schaltungsdesigner bietet sich heutzutage eine breite Palette von Möglichkeiten. Er muss sich intensiv mit den verschiedenen Methoden und Möglichkeiten der Schaltungsrealisierung auseinandersetzen. Da gleichzeitig die geforderten Schaltungen zunehmend komplexer werden, ist es inzwischen ein Muss, beim Entwurf von Funktionsblöcken auszugehen, also ein klares Top-Down-Design anzugehen.

Im vorliegenden Kapitel soll zunächst die klassische Systemaufteilung komplexerer Schaltungen besprochen werden, die nicht mehr mit einem Automaten zu realisieren sind. Danach werden verschiedene mögliche Realisierungen der dargestellten Submodule (Operationswerk, Steuerwerk, Rechenwerk, ..) vorgestellt und erläutert. Deutlich sollte dabei auch der Übergang zwischen Hard- und Software im Mikroprozessoren hervorgehoben werden, was eines der entscheidenden Ziele dieses Kapitels ist. Die verwendeten Beispiele werden natürlich so klein gehalten, dass sie relativ leicht überschaubar bleiben, also im Grunde oft hier auch als Mealy-Automaten vorstellbar wären.

Im Folgenden wird also zunächst erläutert, wie ein "komplexes Schaltwerk" prinzipiell definiert werden kann, was konsequent zum Basisprinzip des Mikroprozessors führt. Darauf aufbauend wird gezeigt, wie Programmschaltwerke - die Steuereinheiten von Mikroprozessoren - entwickelt und optimiert werden können, die die Basis heutiger Mikroprozessoren darstellen.

Schaltwerke mit hoher Komplexität nur sinnvoll, wenn ..

.. weitgehend SYNCHRON!

(Heute!)

Vorgehensweise: Top-Down-Methode!

1      **Spezifizieren**

*Schnittstellenbeschreibung:*

$\vec{e}, \quad \vec{a}, \quad \vec{s}_{in}, \quad \vec{s}_{out}$

2      **Trennung**

- *Operationswerk*

- *Steuerwerk*

3      **Entwicklung des Operationswerkes**

- *modularisieren nach funktionalen Gesichtspunkten*

- *Schnittstellenbeschreibung:*

$\vec{X}, \quad \vec{Y}$

4      **Entwicklung des Steuerwerkes**

- *Gesamtlösung*

- *Vereinfachung*

# Vorgehensweise im Detail

## Schritt 1:



- allgem. Spezifikation der Funktion
- formale Beschreibung der Funktionalität
- Impulszeitdiagramm (?)
- Zustandsdiagramme (?)
- Tabellen (?)
- ...

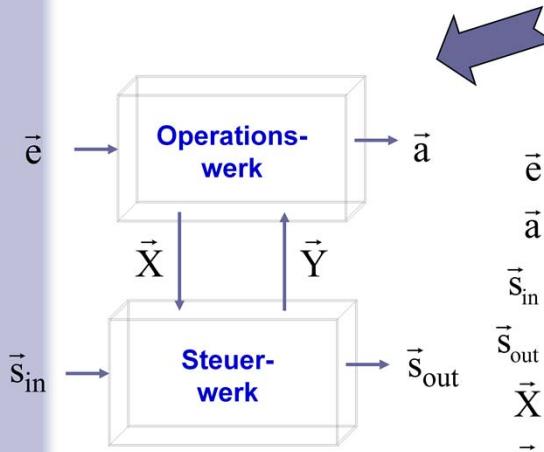
Schaltnetze und Schaltwerke sind als Atome, also als kleinste Zelleinheiten des Systems, aufzufassen. Sie interessieren somit zu Beginn der Schaltungsentwicklung nicht. Ausgegangen wird vielmehr von dem Black-Box-Modell (Bild oben). Das bedeutet, zuerst sind die Eingangs- und Ausgangsschnittstellen sowie die boxinternen Funktionen des zu entwerfenden Systems eindeutig festzulegen und zu beschreiben, was eine klare Aufgabenformulierung voraussetzt. Nach dem Prinzip des Top-Down-Designs ist dann im Folgeschritt das System soweit zu modularisieren, also in weitere Funktionsmodule (Black Boxes) zu untergliedern, bis man diese in akzeptable Schaltnetze und Schaltwerke untergliedert hat.

Ausgegangen wird von der Überlegung, dass ein auf ein komplexes Schaltwerk einwirkender Eingangsvektor  $e^n$  zur Zeit  $t^n$  eine Aktion im komplexen Schaltwerk auslöst. Ebenfalls zur Zeit  $t^n$  wird vom komplexen Schaltwerk ein Ausgangsvektor  $a^n$  gebildet, wobei die Funktion

$$a^n = f^n(e^n)$$

im Allgemeinen nicht mehr zu bestimmen ist (man denke beispielsweise nur an die Komplexität eines Mikroprozessors). Prinzipiell gilt aber wie beim Automaten: Das in das System übernommene Datum  $e^n$  wird im komplexen Schaltwerk manipuliert, und es wird eine Ausgangsgröße  $a^n$  generiert. Nun kann man (funktionell betrachtet) zwei spezifische Aufgaben herauskristallisieren: die Bearbeitung der Daten und die Steuerung des Funktionsablaufs (aus dieser Überlegung haben sich die Begriffe gebildet: Datenpfad und Steuerpfad).

## Schritt 2:



$\vec{e}$ : Nutzdateneingangsvektor

$\vec{a}$ : Nutzdatenausgangsvektor

$\vec{s}_{in}$ : Steureingangsvektor

$\vec{s}_{out}$ : Steuerausgangsvektor

$\vec{X}$ : Meldevektor

$\vec{Y}$ : Steuervektor

Auf der Basis dieser Überlegung sind dann zwei Funktionseinheiten zu definieren: das Operationswerk, in dem das einlaufende Datum manipuliert wird, und das Steuerwerk, das das Operationswerk steuert. Die 'Kommunikation' mit der 'Außenwelt' (Bild oben) passiert über verschiedene Kanäle. Zum einen erhält das Operationswerk Daten von außen ( $\vec{e}$ ) und gibt an diese entsprechende Daten ab ( $\vec{a}$ ), und das Steuerwerk erhält von außen Befehle ( $\vec{s}_{in}$ ) und gibt auch Befehle nach außen weiter ( $\vec{s}_{out}$ ).

Auf einen weiteren Aspekt sei hingewiesen. Die Vektoren  $\vec{e}$ ,  $\vec{a}$ ,  $\vec{s}_{in}$  und  $\vec{s}_{out}$  können jeweils eine einzelne Leitung (ein Vektor mit einer skalaren Größe), aber auch  $n$  Leitungen (mit  $n > 1$ ) repräsentieren. Ist bei allen Größen  $n > 1$ , spricht man von einer parallelen Einheit, ist  $n = 1$ , von einem seriellen System. Wie werden aber nun die gemischten Systeme genannt? Betrachten wir hierzu die bekannte Schnittstellenfestlegung RS232. Sie wird als serielle Schnittstelle bezeichnet, da beim Nutzdateneingangsvektor  $\vec{e}$  und beim Nutzdatenausgangsvektor  $\vec{a}$  nur jeweils eine skalare Größe vorliegt, also  $n = 1$  ist (TxD und RxD). Dass diese Schnittstelle zahlreiche parallele Ein- und Ausgangssteuerleitungen besitzen kann, also die Vektoren  $\vec{s}_{in}$  und  $\vec{s}_{out}$  viele parallele Leitungen repräsentieren können, interessiert nicht, sie wird trotzdem als serielle Schnittstelle angesehen, da man vom Datenpfad ausgeht, nicht vom Steuerpfad. So ist es auch bei vielen anderen Schnittstellen. Im allgemeinen werden nur die Größen  $\vec{e}$  und  $\vec{a}$  zur Begriffsbestimmung herangezogen.

Die Steuersignale, mit denen das Steuerwerk das Operationswerk beeinflusst, werden unter dem Begriff *Steuervektor*  $\vec{y}$  zusammengefasst, während die Informationen, in welchem Zustand sich das Operationswerk befindet, über den *Meldevektor* in das Steuerwerk einfließen.

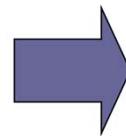
Der im Bild eingezeichnete Clock zur Synchronisation beider Einheiten kann ein Mehrphasen-Clock sein, worauf noch näher eingegangen werden muss.

Streng formal, könnte man hier auch den Begriff des Vektors einführen, doch macht es keinen wirklichen Sinn, denn bei einem Mehrphasen-Clock verlaufen die Impulse phasenverschoben, und das Ziel ist nicht

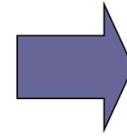
Dateneigenschaften (to Correlation, Standardabweichung, Signifikanzseitenwerte)

# NORM

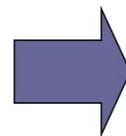
Steuerdaten



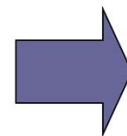
Steuerwerk



Nutzdaten



Operations-  
werk



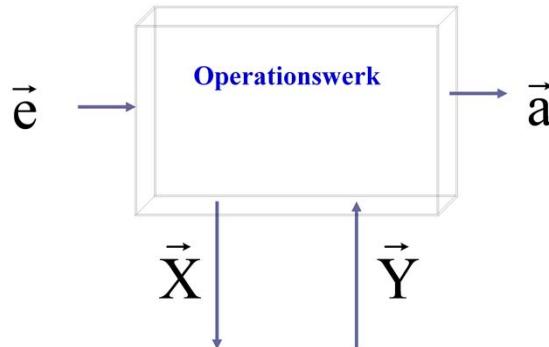
Diese Festlegung ist auch in die internationale Normung der Schaltzeichen eingeflossen, wie das Bild zeigt. Der obere Teil stellt das Steuerwerk, der untere Teil das Operationswerk dar, also umgekehrt wie der Elektrotechniker es im Allgemeinen gewohnt ist (vgl. das Bild mit dem Bild vorher).

Zuse hat die Begriffe *Arbeitswerk* und *Planwerk* gewählt. Später nannte man es dann *Rechenwerk* und *Leitwerk*. Hier sollen die heute in der Mikroprozessortechnik häufigsten Bezeichnungen *Operationswerk* und *Steuerwerk* Verwendung finden.

Der Begriff Kommunikation impliziert auch, dass eine Art Protokoll zwischen der Außenwelt und der Innenwelt des Bausteins ablaufen muss. Wie das Protokoll im Einzelnen auszusehen hat, hängt von der Applikation ab. Hier wird aber eines sehr deutlich, mit welchen Themen man sich konsequenter Weise von der einfachen digitalen Hardware ausgehend - den Schaltnetzen - in Folge zu beschäftigen hat: mit Programmen und Protokollen, wie es schon vorher erwähnt wurde.

Wie ist nun die Vorgehensweise bei der Entwicklung eines komplexeren Schaltwerkes nach dem Bild oben? Da die Methode des Topdown-Entwurfs einzuhalten ist, sind zunächst die Funktionen zu diskutieren, die das System erfüllen soll, und welche Operationen auf die Nutzdaten zu wirken haben. Das heißt, zuerst ist das Operationswerk zu entwickeln und im nächsten Schritt das Steuerwerk.

## Schritt 3:



*Enthält:*

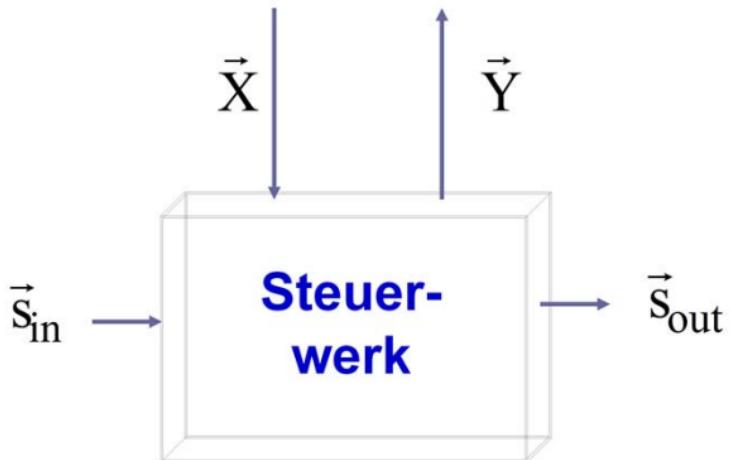
alle Funktionseinheiten zur Verarbeitung der Nutzdaten, also  
Register, Multiplexer, ALUs, ..

*Enthält nicht:*

Ansteuereinheiten der Register, der Multiplexer, ..

Prinzipiell ist es möglich, dem Operationswerk nun wieder einen Automaten zugrunde zu legen, doch dürfte es nur in wenigen Fällen zielführend sein, da er sehr komplex ausfallen würde. Man denke wiederum nur an das Beispiel des Mikroprozessors. Vielmehr nimmt man eine weitere Modularisierung vor und geht über zum

## Schritt 4:



Ansteuereinheit des

- Operationswerkes und
- der Peripherie

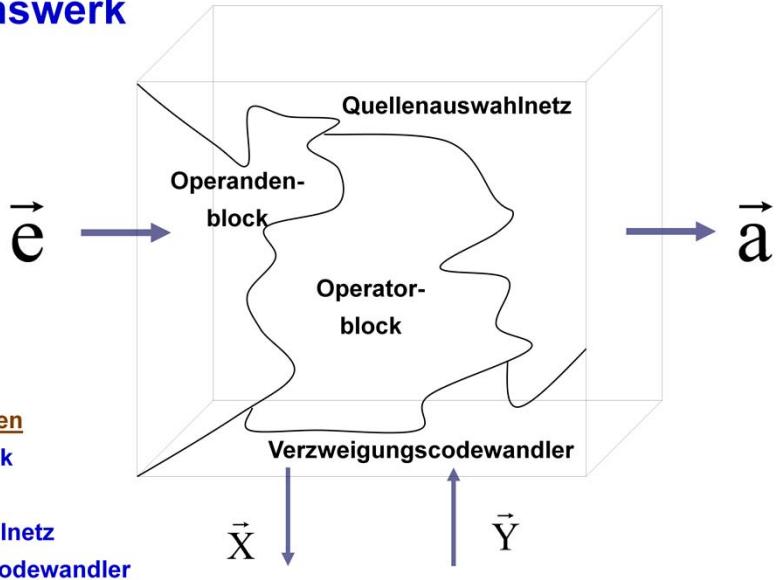
enthält:

- ein oder mehrere Mealy-, Moore-Automaten
- weitere Modularisierung möglich

Realisierung durch:

- logischer Schaltkreisaufbau
- ROM  
( $\mu$ Programmschaltwerk)

# Operationswerk



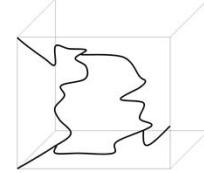
Eine mögliche funktionale Modularisierung des Operationswerkes in vier Submodule zeigt das Bild, die im folgenden kurz erläutert werden sollen.

## Operandenblock

Variable müssen im Allgemeinen - zeitlich gesehen - vor oder nach ihrer Verarbeitung im Operatorblock gespeichert werden. Sind die zu speichernden Datenmengen nicht zu groß, erfolgt dies in den Operandenregistern, sind es zu viele, wie beispielsweise in manchen Mikrocontrollern oder im Transputer, werden ROM- oder RAM-Einheiten integriert. Die Entscheidung, ob die Eingangsdaten tatsächlich im links angeordneten Operandenblock gespeichert werden, wie es in Bild klar und übersichtlich dargestellt ist, weil eine einfache Struktur angenommen wurde, oder ob die Ausgangsdaten zusätzlich in einem weiteren Operandenblock, der weiter rechts im Bild angeordnet wäre, abgelegt werden müssen, oder ob zum Beispiel aufgrund der Komplexität eine interne Busarchitektur gewählt werden muss, bei der alle Daten zu einem beliebigen Zeitpunkt gespeichert werden können, kann nicht generell getroffen werden. In der Praxis hat sich jedoch bewährt, in kleineren Systemen, wie dem im Folgenden vorgestellten, die Daten sofort am Eingang in Register aufzunehmen, während eine Art Busstruktur sich nur für komplexere Systeme (wie beispielsweise) den Mikroprozessoren bewährt. Doch ist diese Aussage sehr pauschal. Denn geht man zu bestimmten Beschleunigungsverfahren des Datendurchsatzes über, so haben weitere, hier noch nicht angesprochene Aspekte Vorrang, und die Situation sieht gänzlich anders aus.

Zusatzbemerkung: Ich verwende extra den Ausdruck "... eine Art Busstruktur.. ", denn wir werden in Kapitel 5 hören, dass *Bussysteme* in Chips selbst sehr unbeliebt sind, da sie unter bestimmten Bedingungen zu Schwingungen neigen können. Deshalb muss man dann zu "ähnlichen" Prinzipien übergehen, die nicht diese Eigenschaften besitzen.

# Operationswerk: Funktionen



## Operandenblock

*Zwischenspeicherung von Informationen (Daten) in Registern, RAMs, ..*

## Operatorblock

*Datenmanipulation in Schaltnetzen, Schaltwerken, ..*

## Quellenauswahlnetz

*Kanalisierung von Daten über Multiplexer, ..*

## Verzweigungscodewandler

*Verdichtung des Steuereingangs- u. Steuerausgangsvektors*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K4-1

b. 13

18.01.2013 13:21:34

## Operatorblock

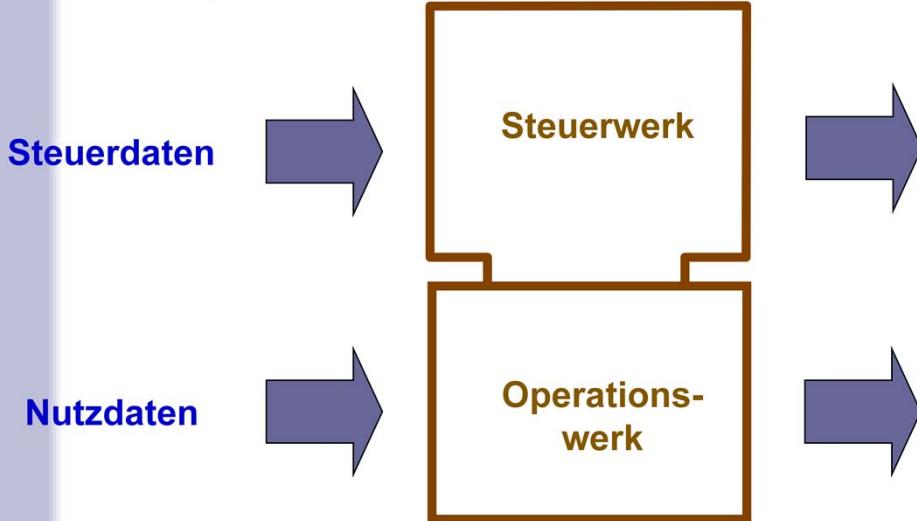
Im Operatorblock werden die Nutzdaten manipuliert. Hier findet somit die eigentliche Informationsverarbeitung statt. Operatoren wirken auf Operanden. Hat man früher die Operatorblöcke hauptsächlich auf der Basis von Automaten realisiert, um Gatter zu sparen, setzt man heute in der Regel nur noch *Schaltnetze* ein, um einen möglichst hohen Datendurchsatz zu erreichen. Beispiele hierfür sind Additions- und Multiplikationseinheiten. Heute wird man kaum noch auf die Idee kommen, Addierer sequentiell zu designen. Selbst den Multiplizierer versucht man möglichst als reines Schaltnetz zu konzipieren, doch reicht dazu nicht immer die Fläche des Chips aus. Das Multiplikationsschaltnetz steht dann eben nicht in voller Datenbreite zur Verfügung, und eine Multiplikation muss in diesem Fall in mehreren Zyklen erfolgen.

## Kanalschaltwerk

Sind Daten (Operanden) in Abhängigkeit von bestimmten Ergebnissen, Zuständen usw. an unterschiedliche Blöcke beziehungsweise Ein- und Ausgänge heranzuführen, müssen sie über logische Gatter, Multiplexer oder Busse kanalisiert werden. In kleineren Einheiten besteht das Kanalschaltwerk vielleicht nur aus wenigen Gattern, und man wird diese Bausteine der Einfachheit halber in die anderen Module einverleiben. Doch in größeren Systemen, wenn das Register Transfer Level Design (RTL Design) beginnt komplex zu werden, kann die Beschreibung mit Hilfe des Kanalschaltwerkes oft deutlich vereinfacht werden. Dies gilt vor allem hinsichtlich des Steuerwerkes, wie man noch sehen wird.

Zusatzbemerkung: Anstelle dem Begriff "Operatorblock" kann auch der synonyme Begriff "Rechenwerk" verwendet werden.

# NORM



## Verzweigungskodewandler

Die Informationen, die zwischen dem Operations- und Steuerwerk transferiert werden, sind oft redundant. Weiterhin wird das Steuerwerk auf dem Chip oft zentral an einem Ort angeordnet. Die Konsequenz daraus sind längere Verbindungen zwischen der Fläche des Operations- und des Steuerwerks. Leitungen auf dem Chip kosten jedoch viel Fläche und bedeuten auch zeitliche Verzögerungen der Signale, sodass es sinnvoll ist, die Informationsredundanz soweit wie möglich zu beseitigen, indem Codewandler integriert werden. In der Literatur findet man Vorschläge, vor dem Steuerwerk wiederum eine Decodierung vorzunehmen. Doch lohnt sich dies im Allgemeinen nicht, wie wir noch sehen werden.

Bevor diese theoretischen Überlegungen in einem Beispiel vertieft werden sollen, kurz ein paar Worte zum Steuerwerk.

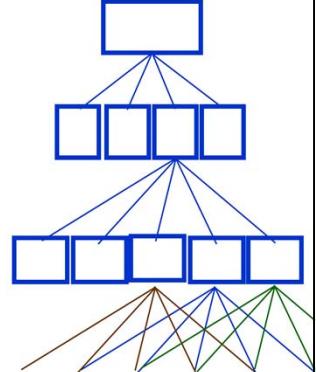
Zusatzbemerkung: Prinzipien wie beim RISC bilden hier eine Ausnahme, da bei ihnen besonders Wert darauf gelegt wird, das Steuerwerk möglichst einfach zu konzipieren, was zum Teil zu einer flächenverteilten Struktur führt. Darauf wird noch näher eingegangen.

Die Aufgabe von Steuerwerken ist auf Eingangsinformationen zu reagieren. Somit können sie im einfachen Fall als Automaten definiert werden, was den entscheidenden Vorteil bietet, dass sie sich immer in einem definierten Zustand befinden. Diese Annahme, dass man einen Automaten zugrunde legen kann, trifft für die meisten Prozessoren zu, die nicht nach dem RISC-Prinzip entwickelt werden. In komplexeren Einheiten findet man dann schon mehrere Automaten integriert (hierarchisch angeordnet oder parallel kommunizierend), die dann durch entsprechende Schaltungen zu erweitern sind.

Die Entwicklung der Automaten war Thema von Kapitel 3. Allerdings sind unterschiedliche Realisierungen möglich, was der zentrale Inhalt von Abschnitt 4.2 *Programmschaltwerke* sein wird.

Zusatzbemerkung: RISC: Reduced Instruction Set Computer (auf ihn wird noch näher eingegangen)

# Topdown- Entwurf



- (1) *Spezifikation*
- (2) *Definition der externen Schnittstellen*
- (3) *Operationswerk und Schnittstellen hin zum Steuerwerk*
- (4) *Steuerwerk*

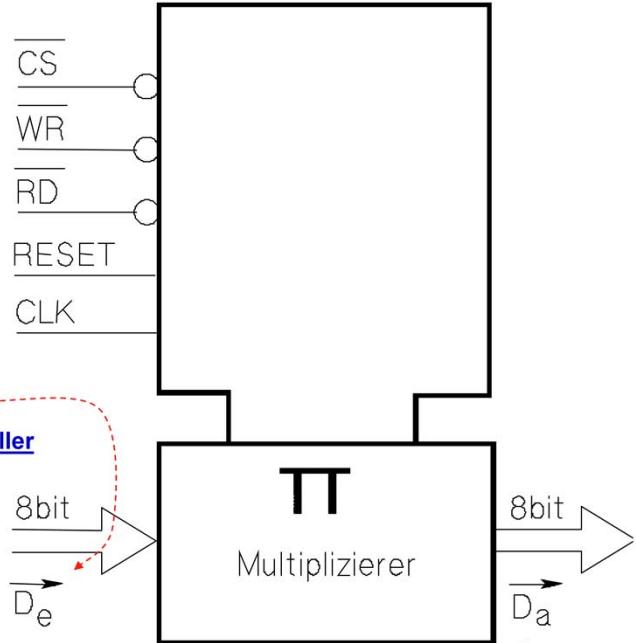
Ein einfaches Beispiel soll die bisherigen Überlegungen verdeutlichen. Entwickelt werden soll ein spezieller Multiplizierer. Die Vorgehensweise soll entsprechend dem Top-Down-Design sein: siehe oben.

Dies entspricht der Vorgehensweise, wie sie in Kapitel 3 für Automaten vorgestellt wurde. Hervorzuheben ist hier nur, dass selbstverständlich das Operationswerk weitgehend durch die Applikation definiert sein muss, bevor das Steuerwerk entwickelt werden kann.

# Beispiel: 8\*8-bit- Multiplizierer

## 1. Spezifikation

- sequentielles Laden von  $D_e$   
(2 x 8 bit)
- interne Zwischenspeicherung aller Daten
- Operation: Multiplikation
- sequentielle Ausgabe von  $D_a$   
(2 x 8 bit)



## (1) Spezifikation

Ausgegangen wird von der Aufgabenstellung, eine 8\*8-Multiplikationseinheit nach Bild oben für einen Mikrocomputer zu entwickeln. Die beiden 8-bit-Datenworte werden sequentiell eingelesen, was ausschließlich über  $\neg CS =$  und  $\neg RD =$  gesteuert wird.  $\neg CS =$  bewirkt die Aktivierung der Unit selbst, über  $\neg RD$  wird das Datum eingelesen. Ist der Lesevorgang abgeschlossen, kann das 16-bit-Ergebnis sequentiell (jeweils 8-bit-weise) ausgelesen werden. Die Besonderheit in diesem Beispiel soll dabei sein, dass das Anlegen der Eingangsdaten nicht exakt erfolgt, dass sich also auch die Steuersignale in engen Grenzen leicht variieren können. Nun soll das allerdings nicht im Detail hier diskutiert werden. Der Einfachheit halber werden deshalb in einem Impulsdiagramm die entsprechenden Zeiten vorgegeben.

# 1. Spezifikation

- (a) **Sequentielle Ein- und Ausgabe: Daten intern speichern**  
→ Operandenblock
- (β) **Multiplikation**  
→ Operatorblock
- (γ) **16-bit-Ergebnis 8-bit-weise ausgeben**  
→ Kanalschaltwerk
- (δ) **Verzweigungscodewandler ??**

Aus der allgemein formulierten Aufgabenstellung heraus können folgende Fakten abgeleitet werden:

- (a) Durch ihre sequentielle Ein- und Ausgabe müssen die Daten intern gespeichert werden.  
Benötigt wird damit mindestens ein Operandenblock.
- (β) In einem Operatorblock müssen die Daten multipliziert werden.
- (γ) Da das 16-bit-Ergebnis 8-bit-weise ausgegeben wird, müssen Daten kanalisiert werden,  
was ein Kanalschaltwerk notwendig werden lässt.
- (δ) Über die Notwendigkeit eines Verzweigungskodewandlers kann noch keine Aussage  
getroffen werden.

## 1. Spezifikation

## 2. Definition der externen Schnittstellen

D<sub>e</sub>: 8-bit-Eingangsdatum: Eingangswort: D<sub>e2</sub>D<sub>e1</sub> ( $2 * 8 \text{ bit} = 16 \text{ bit}$ )

D<sub>a</sub>: 8-bit-Ausgangsdatum: Ausgangswort: D<sub>a2</sub> D<sub>a1</sub> ( $2 * 8 \text{ bit} = 16 \text{ bit}$ )

S<sub>in</sub>: → CS, → WR, → RD, RESET

S<sub>out</sub>: keine



### (2) Definition der externen Schnittstellen

Die externen Schnittstellen lassen sich direkt angeben:

D<sub>e</sub>: 8-bit-Eingangsdatum: Eingangswort: D<sub>e2</sub>D<sub>e1</sub> ( $2 * 8 \text{ bit} = 16 \text{ bit}$ )

D<sub>a</sub>: 8-bit-Ausgangsdatum: Ausgangswort: D<sub>a2</sub> D<sub>a1</sub> ( $2 * 8 \text{ bit} = 16 \text{ bit}$ )

S<sub>in</sub>: → CS, → WR, → RD, RESET

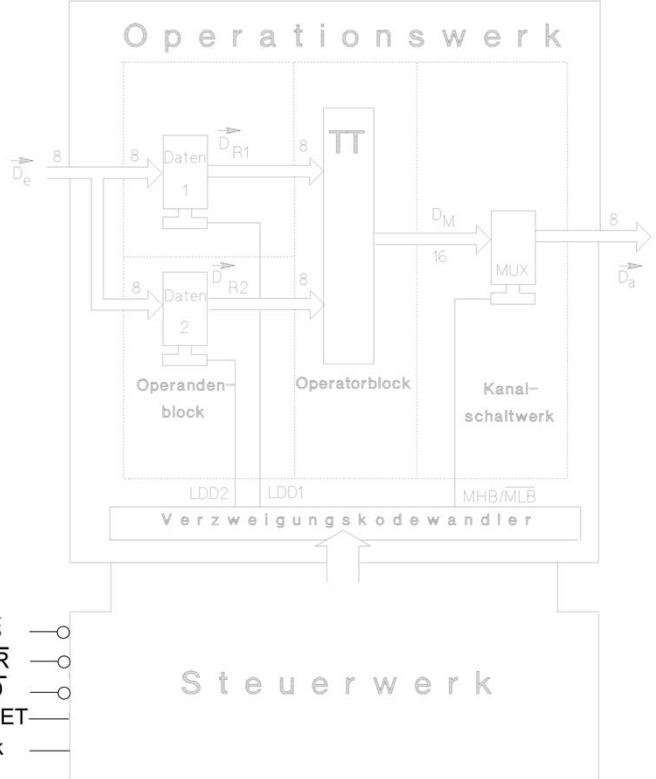
S<sub>out</sub>: keine

Das Signal RESET wird im ASIC-Entwurf schon aus Gründen des "trickfreien" und klaren Entwurfs sowie der Testbarkeit prinzipiell direkt an die FFs geführt und nicht in den logischen Entwurf miteinbezogen. Ebenso der Clock, wie schon in Kapitel 3 dargelegt. Aus diesem Grund werden im Folgenden keine weiteren Worte mehr darüber verloren.

# 1. Spezifikation

# 2. Definition der externen Schnittstellen

# 3. Operationswerk und Schnittstellen hin zum Steuerwerk



Die Schreibweise MHB/ $\neg$ MLB wird hier NUR in der Zeichnung verwendet, um auszudrücken, dass das Signal MHB high-aktiv ist. Ist das gleiche Signal negativ, bedeutet es  $\neg$ MLB.

LDD2: Laden des 1. Datums, LDD1: Laden des 2. Datums, MHB: Multiplexer-high-Byte,  $\neg$ MLB: Multiplexer-low-Byte

### (3) Operationswerk

Wird von der Überlegung ausgegangen, dass Daten unabhängig voneinander sequentiell ein- und ausgelesen werden, steht fest, am Eingang oder Ausgang das jeweilige 16-bit-Datum gespeichert werden muss. Spielt man die verschiedenen Möglichkeiten der Registeranordnungen durch, gelangt man zu dem Ergebnis, dass im vorliegenden Fall eine optimale, registersparende Anordnung dann getroffen wird, wenn zwei 8-bit-Register am Eingang platziert werden. Damit können beide Eingangsdaten sequentiell übernommen und gespeichert werden, und das Ergebnis (ein Datum einer 8\*8-bit-Multiplikation: 16-bit-Datum) kann anschließend, allein abhängig vom  $\neg$ WR-Signal, gesteuert über den Multiplexer, 8-bit-weise sequentiell ausgelesen werden (Kanalschaltwerk oder Quellenauswahlnetz).

Um das Operationswerk zu steuern, werden somit drei Steuerleitungen benötigt: LDD1 und LDD2 zur Ansteuerung der Register und MHB/ $\neg$ MLB, um den Multiplexer zu schalten.

### 3. Operationswerk und Schnittstellen hin zum Steuerwerk

#### Operandenblock (Register)

**Schnittstellen:** LDD2 = 1: erstes Datum wird übernommen  
LDD1 = 1: zweites Datum wird übernommen  
jeweils 8-bit-Eingang und 8-bit-Ausgang

#### Operatorblock (Schaltnetz)

**Schnittstellen:** 2 x 8-bit-Eingang  
1 x 16-bit-Ausgang

#### Quellenauswahlnetz (Schaltnetz)

**Schnittstellen:** MHB = 1 : High-Byte rausschreiben  
= 0 : Low-Byte auslesen  
1 x 16-bit-Eingang; 2 x 8-bit-Ausgang

**Bemerkung:** *Multiplexer-high-Byte:* MHB  $\Leftrightarrow$  MHB / MLB

## Verzweigungscodewandler

Eingangsschnittstelle: LDD1, LDD2, MHB

notwendig (??) (Ausgangsschnittstelle: ?)

- Operandenblock: 2 Register nach Bibliothek
- Operatorblock: Bibliothek oder Eigenentwurf ???

Eigenentwurf:

Eingang: $D_R$										Ausgang				
$D_{R2}$					$D_{R1}$					$D_M$				
$D_{15}$	$D_{14}$	..	$D_9$	$D_8$	$D_7$	$D_6$	..	$D_1$	$D_0$	$D_{M,15}$	$D_{M,14}$	..	$D_{M,1}$	$D_{M,0}$
0	0	..	0	0	0	0	..	0	0	0	0	..	0	0
0	0	..	0	0	0	0	..	0	1	0	0	..	0	1
..														

Offen ist noch, ob ein Verzweigungscodewandler zu integrieren ist. Wird davon ausgegangen, dass zunächst das 1. Datum (High-Byte; über LDD2), dann das 2. Datum (Low-Byte; über LDD1) eingelesen wird, und sich daran die sequentielle Ausgabe anschließt, kann eine Codierung entsprechend der Tabelle auf dem nächsten Slide festgelegt werden. Wird weiterhin davon ausgegangen, dass ein System vorliegt, das nicht exakt taktsynchron arbeitet (wie eingangs formuliert), benötigt man zumindest mehr als vier Schritte. Das bedeutet aber, dass eine Minimierung der Anzahl der Steuerleitungen durch ihre Codierung, das heißt in diesem Fall deren Reduzierung auf zwei Leitungen, auf keinen Fall möglich ist. Mit drei Leitungen sollte man dagegen auskommen, was aber noch genauer zu untersuchen ist. Ein Verzweigungscodewandler kommt somit nicht in Frage.

Ein Eigenentwurf des Multiplizierers ist die eine Möglichkeit, die vollständige Entnahme aus der Bibliothek eine andere. Beim eigenständigen Entwurf des Multiplizierers muss man sich darüber im Klaren sein, dass man es mit 16 Dateneingänge und 16 Datenausgänge zu tun hat (siehe Tabelle oben) und somit der Aufwand groß werden kann. Bevor man sich also damit auseinandersetzt, wie man solche Schaltungen "effizient" entwickelt, sollte man die umfangreichen Bibliotheken in Anspruch nehmen, die heute zur Verfügung stehen.

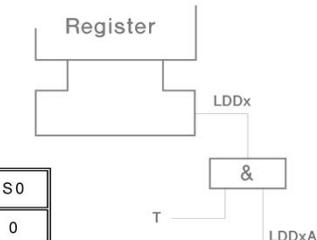
- Operandenblock: 2 Register nach Bibliothek
- Operatorblock: aus Bibliothek und modifizieren
- Quellenauswahlnetz: ??

**Entwurf:**

	L D D 2	L D D 1	M H B / <u>M L B</u>
1 . D a t u m e i n l e s e n	1	0	x
u m s c h a l t e n	0	0	x
2 . D a t u m e i n l e s e n	0	1	x
u m s c h a l t e n	0	0	x
L o w - B y t e a u s l e s e n	0	0	0
H i g h - B y t e a u s l e s e n	0	0	1

	L D D 2	L D D 1	M H B / M L B
1 . D a t u m e i n l e s e n	1	0	x
u m s c h a l t e n	0	0	x
2 . D a t u m e i n l e s e n	0	1	x
u m s c h a l t e n	0	0	x
L o w - B y t e a u s l e s e n	0	0	0
H i g h - B y t e a u s l e s e n	0	0	1

$$L D D x A = L D D x \wedge T :$$



	L D D 2 A	L D D 1 A	M H B / M L B	S 1	S 0
1 . D a t u m e i n l e s e n	1	0	x	0	0
2 . D a t u m e i n l e s e n	0	1	x	0	1
L o w - B y t e a u s l e s e n	0	0	0	1	0
H i g h - B y t e a u s l e s e n	0	0	1	1	1

Somit Reduktion von 3 Steuerleitung auf 2 möglich:

**ABER VERBOTEN!**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

Missachtet man die ASIC-Design-Regeln, kann man den Takt mit Steuerleitungen verknüpfen (theoretisch ist dies für einen diskreten Aufbau möglich, wenn man annehmen darf, dass auch zukünftig keine andere Technologie in Frage kommt). Doch man muss sich dann darüber im Klaren sein, dass dies zu massiven Schwierigkeiten führen kann, da der Entwurf technologieabhängig wird (siehe Kapitel 3, Abschnitt *Schaltwerke mit Taktausblendung*). In diesem Fall bleibt das Eingangssignal der Multiplexer LDDxA (siehe Bild) nicht für die ganze Taktperiode auf logisch 1. Es gilt dann nämlich

$$L D D x A = L D D x \wedge T$$

mit  $T := \text{Clock}$  und  $x = 1, 2$ ,

und die Zeilen *umschalten* in der Tabelle von Tab. oben auf dem Slide können - wenn keine weiteren Verzögerungen zu berücksichtigen sind - entfallen. Für vier Zeilen sind jedoch nun nur noch zwei Steuersignale (sie werden hier mit S1 und S0 bezeichnet) notwendig, was zu einem Verzweigungscodewandler nach Tab. unten im Slide führt.

Doch nochmals: solch ein "Trick" ist für die ASIC-Entwicklung heutzutage ausgeschlossen. Der Takt sollte *nie* Teil der Informationsverarbeitung sein. Die folgenden Betrachtungen gehen also von Tab. oben im Slide aus, wobei die Übergänge noch genauer zu analysieren sind.

# Steuerwerk

## Schnittstellen:

Takt: Clk

Eingang:  $\overline{CS}$ ,  $\overline{RD}$ ,  $\overline{WR}$ , RESET,

Ausgang: LDD2, LDD1, MHB

## Vereinfachung:

$$\overline{RDC} = \overline{RD} \vee \overline{CS}$$

$$\overline{WRC} = \overline{WR} \vee \overline{CS}$$

RESET: geht DIREKT in FFs hinein  
(für weiteren Entwurf somit uninteressant)



## (4) Steuerwerk

Bevor im nächsten Schritt das Zustandsdiagramm entworfen wird, ist zunächst zu prüfen, ob die verschiedenen Steuerleitungen zusammenzufassen sind. Infrage kommen nur  $\neg CS$ ,  $\neg WR$  und  $\neg RD$ . RESET ist ein asynchrones Signal und darf auf keinen Fall mit einem synchronen verknüpft werden. Das heißt, das RESET-Signal führt prinzipiell direkt auf spezielle Rücksetzeingänge der verwendeten FFs. Das Signal muss somit in der weiteren Entwicklung des Entwurfs (synchrone Schaltung!) nicht berücksichtigt werden.

Stellt man keine hohen Laufzeitanforderungen an die einzelnen logischen Komponenten des zu entwickelnden Bausteins (also gemäßigte Taktrate, der bei Mikroprozessoren übliche Read-Write-Zyklus:  $\neg WR$  und  $\neg RD$  darf nur wirksam werden, wenn  $\neg CS$  aktiv ist), kann folgende vereinfachende Zusammenfassung vorgenommen werden:

$$\neg RDC = \neg RD \wedge \neg CS$$

$$\neg WRC = \neg WR \wedge \neg CS$$

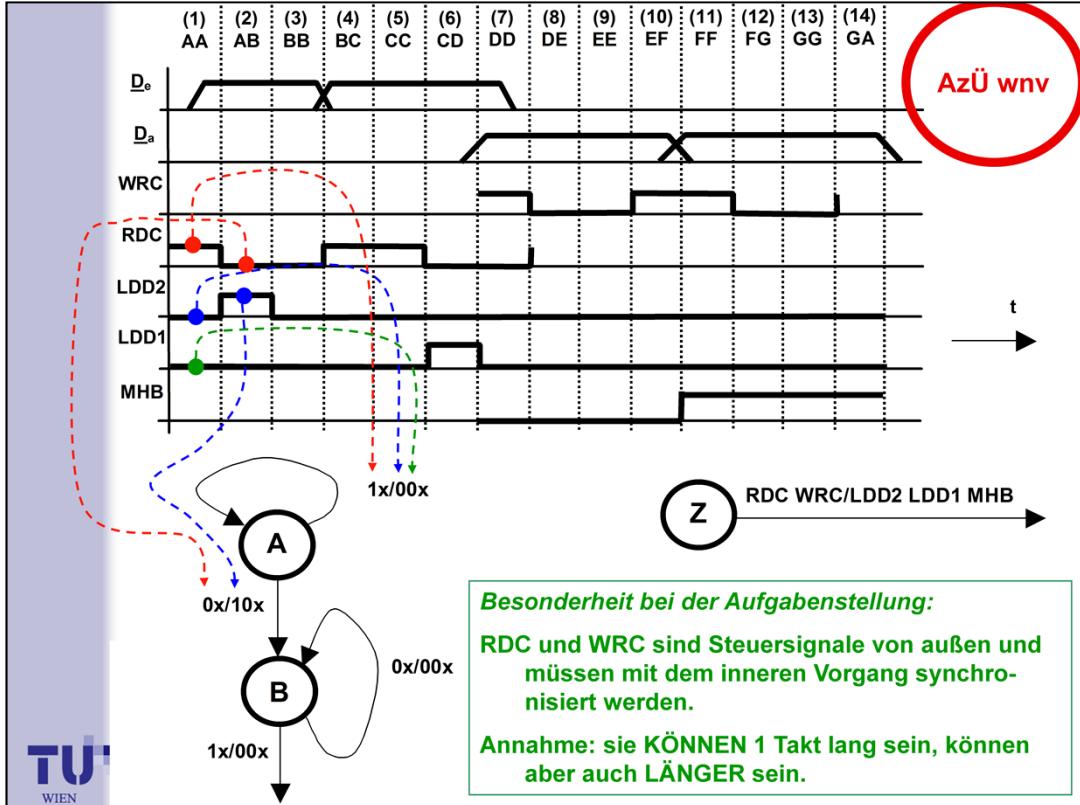
## Weitere Vorgehensweise:

- (1) Spezifikation im Detail + Timing-Diagramm
- (2) Übergangstabelle + Zustandsdiagramm
- (3) Minimierung – Gleichungen als Mealy  
auf (a) Schaltnetzbasis und als (b) ROM

## (1) Spezifikation im Detail + Timing-Diagramm

	Eingang	Funktion	Aktion - Ausgang	Zustandsübergangstyp
1: AA	RDC inaktiv (= 1, da low-aktiv), WRC spielt keine Rolle	Idle-Zustand (warten)	auf Datum $D_e$ warten	$Z_i Z_i$
2: AB	RDC = 0 (aktiv)	$D_e$ liegt stabil an	erstes Datum $D_e$ einlesen: LDD2 (high-Byte)	$Z_i Z_k$
3: BB	eventuell noch RDC = 0	Schleife	Warten auf Wechsel	$Z_i Z_i$
..				

Bevor jedoch nun im Detail die Vorgehensweise der Entwicklung des Timing-Diagramms erläutert werden soll, ein Hinweis zu einer sicheren Vorgehensweise. Um so komplexer ein System wird, um so sinnvoller ist es, sich Dokumentteile bis hin zu Tabellen zu verschaffen, die im ersten Ansatz vielleicht trivial aussehen mögen, aber eine gute Übersicht verschaffen. Es hilft Fehler zu finden, die prinzipiell nicht zu vermeiden sind, vor allem wenn die Komplexität zunimmt. In diesem Sinne ist die folgende Tabelle zu verstehen, in der Zeile für Zeile im Zusammenhang mit dem Timing-Diagramm und in Folge davon mit der Übergangstabelle und dem Zustandsdiagramm entworfen werden kann.

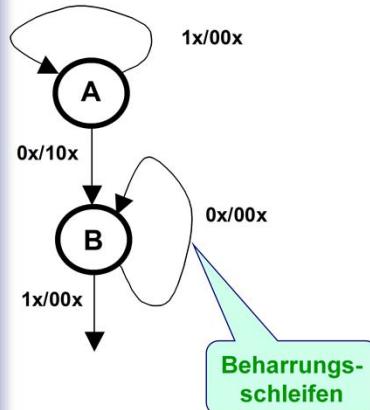


Der nächste Schritt ist nun die Erstellung des Timing-Diagramms. Vorgegeben ist das Timing so, dass die RDC- und WRC-Signale im Allgemeinen einen Takt lang sind, jedoch evtl. sich auch etwas länger hinziehen können, da oft die Technologie oder die Leitungslängen mitberücksichtigt werden müssen). Wenn das erste RDC = 0 (siehe Bild Operationswerk vorne: low-aktiv) ansteht (also im Zustand (2); Zustand (1) soll der Idle-Zustand sein), sollte das Eingangsdatum  $D_e$  schon stabil anliegen. Im Zustand (3) sollte das System dann solange verharren, bis das Signal RDC wieder auf 1 gesetzt wurde. Das kann leicht mit einer Zustandsfolge erreicht werden, wie sie in Bild unten dieses Slides dargestellt ist.

Damit kann für den Zustandsgrafen eine Nomenklatur entworfen werden, wie sie in Bild oben dargestellt ist. RDC und WRC sind die Eingangssignale, LDD2, LDD1 und MHB die Ausgangssignale.

Zusatzbemerkung: Im Bild sind die Steuersignale  $\neg CS$ ,  $\neg WR$  und  $\neg RD$  negiert dargestellt, was bedeuten soll, dass sie low-aktiv sind. Im Zustands- und Timing-Diagramm und in Tabellen verwende ich die nicht negierte Schreibform, da damit leichter umzugehen ist. Für die Schreibweise von MHB gilt entsprechendes. Nur in der Schaltungsdarstellung wird die vollständige übliche Schreibweise (<nicht-negierter Ausdruck 1>/<negierter Ausdruck 2>) verwendet. Ansonsten soll die einfachere Schreibweise MHB genügen.

Vom Idle-Zustand A beginnend, geht das System durch Aktivierung des RDC (= 0) in den Zustand B über und bleibt dort solange, bis RDC wieder auf 1 gesetzt wird. Entscheidend ist dabei, dass das Datum gleich im ersten Takt übernommen wird und der Wert von  $D_e$  im zweiten Takt eigentlich dann schon wieder irrelevant ist.



	<b><math>t^n</math></b>			<b><math>t^{n+1}</math></b>		<b><math>t^n</math></b>		
	RDC	WRC	Z	Z	LDD2	LDD1	MHB	
(1)	1	x	A	A	0	0	x	
(2)	0	x	A	B	1	0	x	
(3)	0	x	B	B	0	0	x	
(4)	..							

**Beharrungs-schleifen**

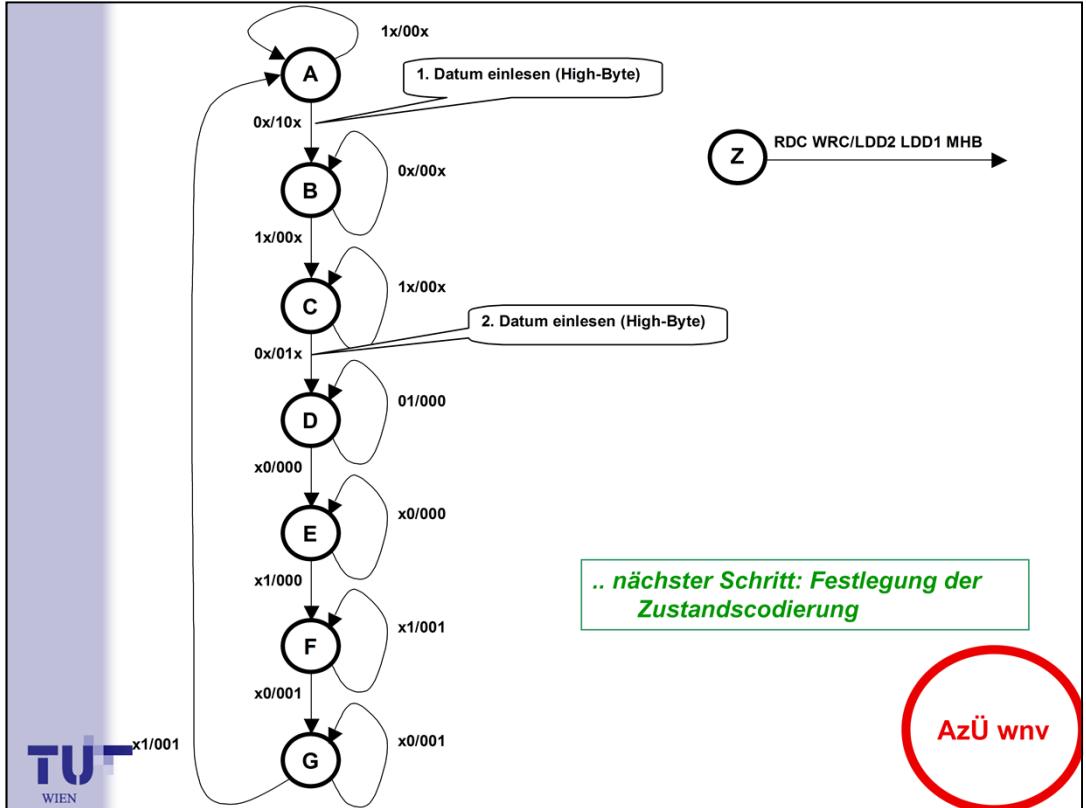
*.. also prinzipiell zu berücksichtigen:*

**RDC und WRC müssen über Beharrungsschleifen synchronisiert werden.**

**Annahme: sie KÖNNEN 1 Takt lang sein, können aber auch LÄNGER sein**

**AzÜ wnv**

K4-1  
p. 27  
18.07.2013 13:21:34



Die entscheidenden Punkte sind:

Zustandsübergang (2): A → B: 1. Datum (High-Byte) wird über LDD2 eingelesen.

Zustandsübergang (6): C → D: 2. Datum (Low-Byte) wird über LDD1 eingelesen.

Zustandsübergang (8): D → E: Nach Zustandsübergang C → D liegt intern das Datum stabil vor und kann nun ausgegeben werden.

Der Übergang (4) B → C und (10) E → F von 0 → 1 ist wichtig, damit eine eindeutige Trennung beim Ein- und Auslesen der Daten vom 1. zum 2. Wort erfolgt.

Die jeweiligen Folgeübergänge (5) C → C und (11) F → F sind integriert, um etwaige Verzögerungen der Eingangsleitungen RDC und WRC auszugleichen. Diese Übergänge können bei kurzen Signalen von RDC und WRC wegfallen, in dem das System diese Zustandsübergänge erst gar nicht einnimmt.

Das lange Halten der Werte von MHB und →MLB müsste nicht sein, trägt aber zur Sicherheit eventuell bei.

	$t^n$						$t^{n+1}$				$t^n$		
	RDC	WRC	Z	$Q_2$	$Q_1$	$Q_0$	Z	$Q_2$	$Q_1$	$Q_0$	LDD2	LDD1	MHB
(1)	1	x	A	0	0	0	A	0	0	0	0	0	x
(2)	0	x	A	0	0	0	B	0	0	1	1	0	x
(3)	0	x	B	0	0	1	B	0	0	1	0	0	x
(4)	1	x	B	0	0	1	C	0	1	0	0	0	x
(5)	1	x	C	0	1	0	C	0	1	0	0	0	x
(6)	0	x	C	0	1	0	D	0	1	1	0	1	x
(7)	0	1	D	0	1	1	D	0	1	1	0	0	0
(8)	x	0	D	0	1	1	E	1	0	0	0	0	0
(9)	x	0	E	1	0	0	E	1	0	0	0	0	0
(10)	x	1	E	1	0	0	F	1	0	1	0	0	0
(11)	x	1	F	1	0	1	F	1	0	1	0	0	1
(12)	x	0	F	1	0	1	G	1	1	0	0	0	1
(13)	x	0	G	1	1	0	G	1	1	0	0	0	1
(14)	x	1	G	1	1	0	A	0	0	0	0	0	1

AzÜ wnv

In der Übergangstabelle und im Zustandsdiagramm sind alle notwendigen Informationen enthalten, um den Entwurf des Steuerwerkes auf der Basis eines Mealy-Automaten zu vervollständigen. Die Frage stellt sich nun, ob das Steuerwerk als

- festverdrahtetes Schaltwerk oder
- ROM-Schaltwerk

realisiert werden soll.

Wird es als festverdrahtetes Schaltwerk entworfen, ist die entsprechende Tabelle mit den Übergangszuständen zu entwerfen. Danach ist ein entsprechender Entwurf mit einem Verzweigungscodewandler durchzuführen, um zu prüfen, welcher der beiden Entwürfe die minimale Anzahl von Gattern beziehungsweise die minimale Anzahl Integrierter Schaltkreise ermöglicht. Die Tabelle zeigt eine mögliche Realisierung mit einer lexikografischen Anordnung der Zustände. Hier könnte vielleicht eine optimalere Lösung noch gefunden werden, doch möchte ich die weiteren Überlegungen auf einen anderen Aspekt lenken.

# Steuerwerk auf Basis logischer Bausteine u. ROM

..weitere Vorgehensweise:

➤ Aufstellen der Gleichungen für einen Mealy-/Moore-Automaten

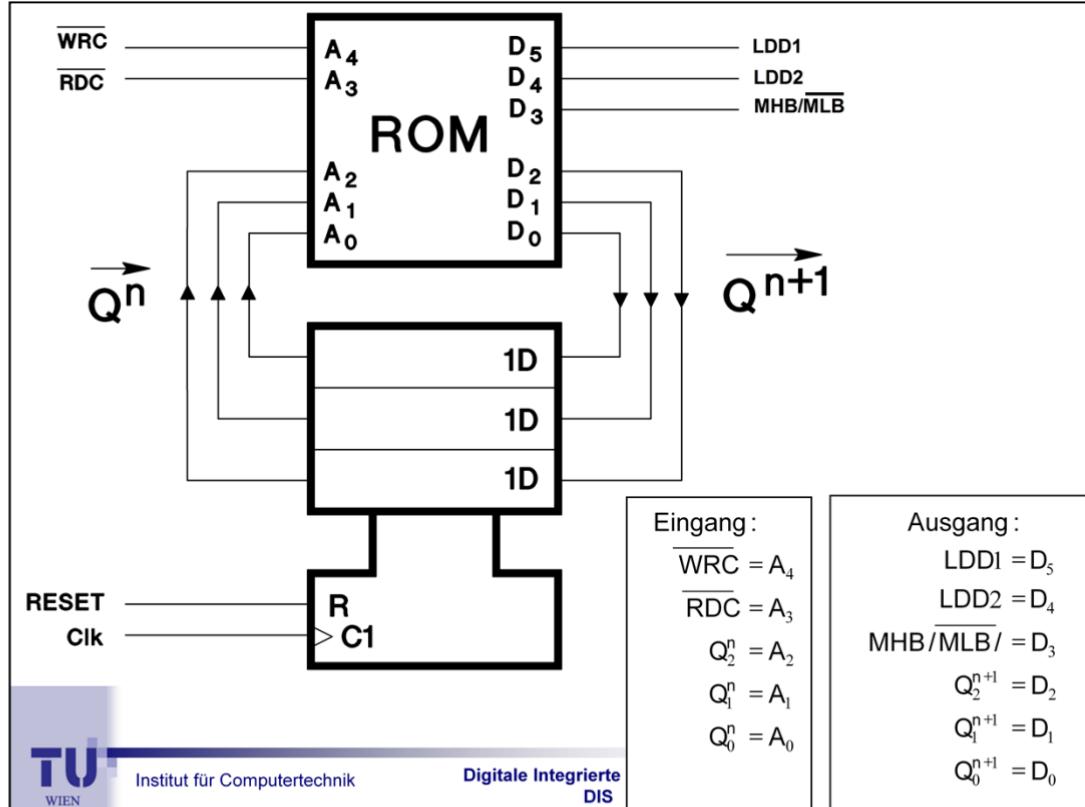
➤ usw.

oder auf ROM-Basis

	t <sup>n</sup>						t <sup>n+1</sup>				t <sup>n</sup>		
	RDC	WRC	Z	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Z	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	LDD1	LDD2	MHB
	A <sub>3</sub>	A <sub>4</sub>	x	A	0	0	0	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>
1	1	1	x	A	0	0	0	0	0	0	0	0	x
2	0	0	x	A	0	0	0	B	0	0	1	1	0
3	..												

Es soll der Entwurf des ROM-Schaltwerkes erläutert werden. Zugrundegelegt wird die Übergangstabelle vom letzten Bild sowie das Schaltbild des Steuerwerkes, das sich aus der Prinzipschaltung des Mealy-Automaten ableitet. Das Schaltnetz wird in diesem Fall über ein ROM realisiert, wodurch die Eingangsgrößen auf die Adressen entsprechend der Tabelle abgebildet werden können.

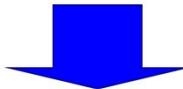
Die Pin-Belegung (blauer Rahmen oben) kann beliebig gewählt werden.



Dann kann die ursprüngliche Tabelle erweitert werden mit den Daten- und Adresszuweisungen aus der Tabelle oben, was zu Tabelle auf dem nächsten Slide führt.

# Steuerwerk auf ROM-Basis

	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
(1)	1	x	0	0	0	0	0	0	0	0	x
(2)	0	x	0	0	0	0	0	1	1	0	x
(3)	0	x	0	0	1	0	0	1	0	0	x



Adressen [Hex]	Daten [Hex]	Adressen [Hex]	Daten [Hex]	Adressen [Hex]	Daten [Hex]	Adressen [Hex]	Daten [Hex]
0	0C	A	1A	10	00	1A	10
1	08	B	18	11	10	1B	*
2	1A	C	28	12	10	1C	28

Doch zunächst noch ein paar Worte zur Erstellung der Programmtabelle. Es ist zu beachten, dass irrelevante Eingänge (mit x bezeichnet) bei der Programmierung berücksichtigt werden müssen, da sie ja im physikalisch vorhandenen Baustein die logischen Werte 0 oder 1 annehmen werden. Zu programmieren ist demnach der Zustand  $x = 0$  und auch der Eingang  $x = 1$ . Für die Ausgangstabelle bedeutet dies beispielsweise, dass der Wert  $D = 00000x$  der Adresse  $A = 1x000$  gleich zweimal zu programmieren ist: einmal unter der Adresse  $A = 10$  [Hex] und einmal unter der Adresse  $A = 18$  [Hex]. Irrelevante Ausgänge interessieren dagegen nicht und können somit mit 0 oder 1 belegt werden. Wegen der einfacheren Durchschaubarkeit werden sie hier mit logisch 0 angenommen, obwohl dies nicht praxisgerecht sein muss.

Ein Hinweis ist noch bezüglich des ROMs zu geben. In der Praxis wird man heute kaum noch in Verlegenheit kommen, ein Steuerwerk diskret aufzubauen. Im Allgemeinen sind sie Teil einer Einheit auf einem Chip, so dass die Größe variiert werden kann. Das bedeutet, jede Zeile des ROMs kann eventuell die Einsparung von Transistoren bedeuten.

EPROMs usw. besitzen im Allgemeinen eine Voreinstellung. Aus Kostengründen ist deshalb der Wert zu programmieren, der die geringste Programmierzeit ("Brennzeit") in Anspruch nimmt. Für ROMs gilt entsprechendes. Es ist der Wert zu wählen, der den geringsten Produktionsaufwand erfordert. In CMOS-Ausführung sind sie im Allgemeinen mit 1 belegt, das bedeutet, irrelevante Ausgänge sollten dann eben nicht auf 0, sondern auf 1 festgelegt werden.

Zusatzbemerkung: Ein ROM ist auch ein Schaltnetz.

# Zunehmende Komplexität des Operationswerkes

- Differenzierung in 4 Module (Operandenblock, Operatorblock, ..)  
nicht mehr ausreichend (Beispiel: µP)  
weitere Modularisierung notwendig
- Hilfsmittel:
  - Top-Down-Design
  - funktionalorientierte, formale Beschreibungssprachen  
(VHDL, ..)

## **Thema im weiteren:**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K4-1

p. 33  
18.01.2013 13:21:34

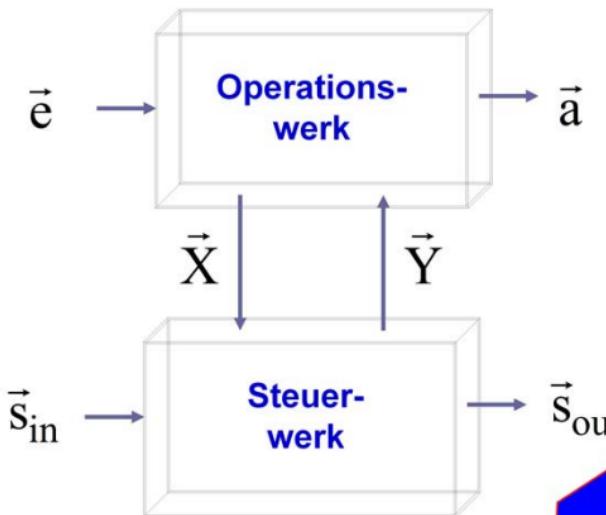
Das Beispiel des vorausgegangenen Abschnitts zeigt, dass die Realisierung eines Programmschaltwerkes im Prinzip einfach ist. Trotzdem ist es notwendig, sich mit dieser Thematik näher zu beschäftigen. Zum einen ist sie durchaus nicht mehr trivial, wenn die Strukturen komplexer werden, und zum anderen kann man das Programm des Programmschaltwerkes als Übergangseinheit, das heißt

### **Schnittstelle zwischen Hard- und Software**

betrachten. Sie bereitet oft Verständnisschwierigkeiten schon prinzipieller Art. Von ihr lassen sich Parallelen ziehen zu Aufgaben im Bereich der AI-Forschung (Artificial Intelligence) und Modelle bionischer Systeme, was jedoch im Rahmen dieser Vorlesung nicht dargelegt werden kann. Was damit jedoch zum Ausdruck gebracht werden soll, ist die Tatsache, dass dieses Wissen nicht nur fundamentales Wissen bezüglich der Computertechnik darstellt, sondern weit mehr zum Verständnis komplexer Computertechnologie bis hin zur AI beiträgt, als man im ersten Moment vielleicht ahnt.

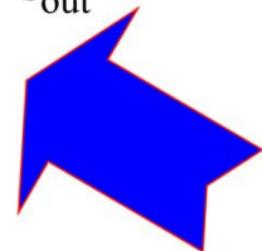
Rembold spricht in /Remb87, S.333/ von den wesentlichen Nachteilen der Von-Neumann-Architektur und schreibt wörtlich: "Das einzige Maschinen-Datenobjekt einer Von-Neumann-Maschine ist letztlich die Bitkette. Informationstypen dieser Bitkette können sein: Adressen, Befehle und Datenwerte bestimmter Daten der verwendeten Programmiersprache wie REAL, INTEGER, RECORD etc. Auf Maschinenebene sind diese Attribute der Datenobjekte, die von einem Programmierer eingesetzt werden, nicht mehr vorhanden. Es existiert somit eine semantische Lücke zwischen den Programmiersprachen und der Von-Neumann-Maschine."

Genau um die Überbrückung dieser Lücke geht es im vorliegenden Abschnitt. Wie ist die Vorgehensweise? In /Patt94/ heißt es auf Seite 338: "The easiest way to understand the microprogram is to break it into pieces that deal with each component of instruction execution, just as we did when we designed the finite state machine."



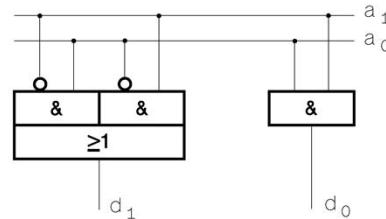
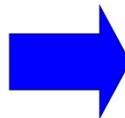
### **Steuerwerk-**

- Historie
- Aufbau
- Optimierung
- Weiterentwicklung



$$d_1 = a_1 \bar{a}_0 \vee \bar{a}_1 a_0$$

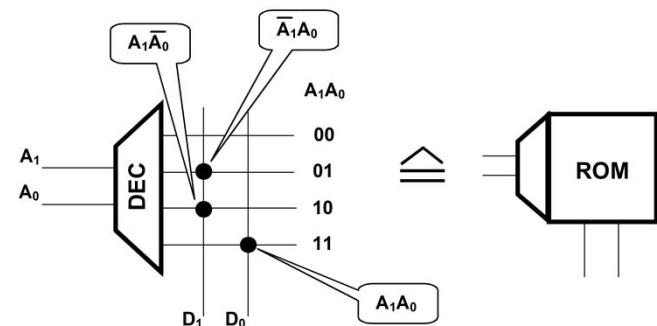
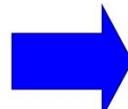
$$d_0 = a_1 a_0$$



*Halbaddierer auf Basis*

*logischer Bausteine  
+ eines ROMs*

$A_1 A_0$	$D_1 D_0$
0 0	0 0
0 1	1 0
1 0	1 0
1 1	0 1

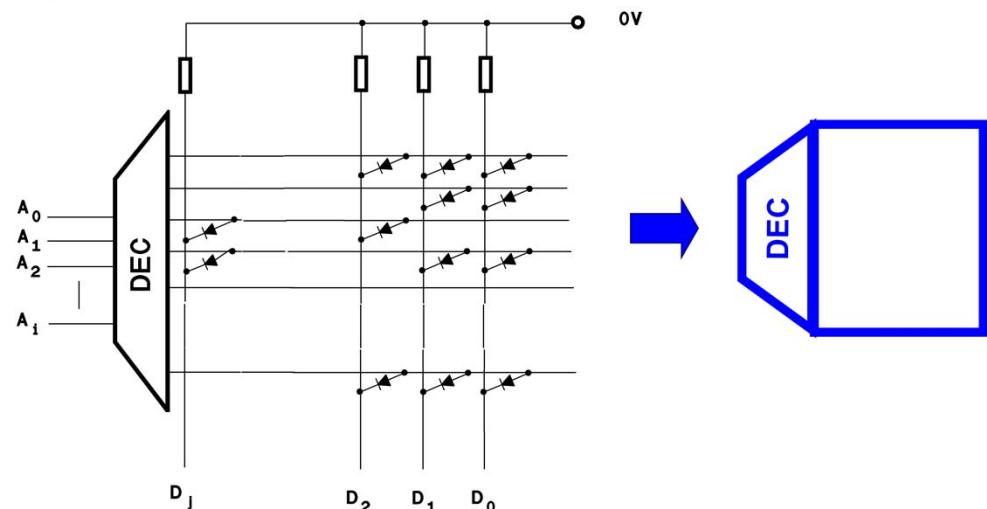


Nochmals zur Erinnerung: Ein Schaltnetz kann über logische Bausteine wie UND und ODER, aber auch über ein ROM realisiert werden. Hierfür das Beispiel oben.

Halbaddierer auf der Basis von logischen Schaltelementen und auf der Basis eines ROMs

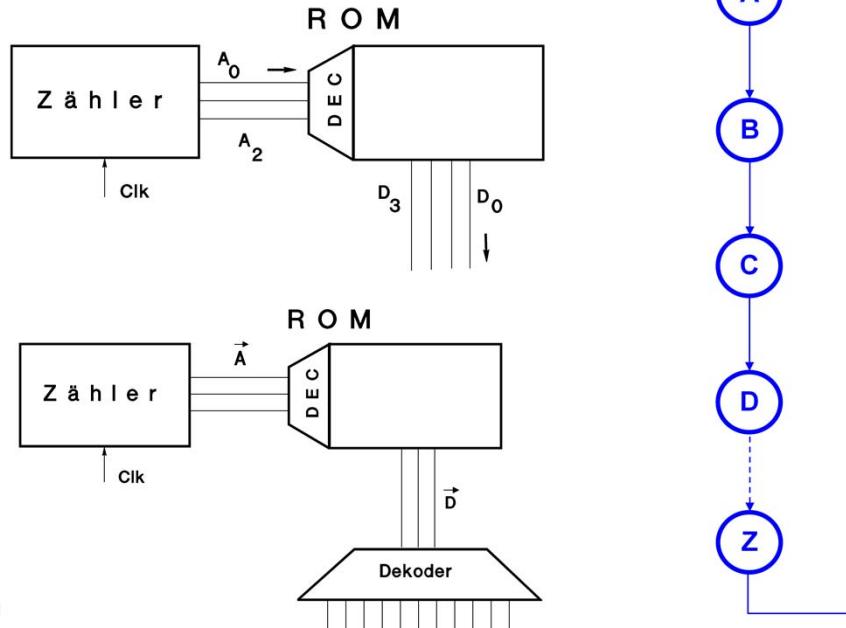
Es stellt das Gleichungssystem des Halbaddierers dar. Das Bild zeigt die Realisierung einmal auf der Basis von UND- und ODER-Gliedern zum anderen auf der Basis einer ROM-Realisierung.

## Eine Realisierungsmöglichkeit des ROMs



Wird ein ROM (man verwendet in der Literatur in diesem Fall dann die Großbuchstaben A (Adressen) und D (Daten)) nach dem Prinzip von Bild oben verwendet, so sind die im Bild des vorhergehenden Beispiels eingezeichneten Punkte in den Linienkreuzungen die Diodenkopplungen. Die Programmierung erfolgt vereinfachend über die Tabelle (siehe Beispiel vorher). Liegt also die Schaltung in Form einer Gleichung vor, ist der Weg über die Tabelle beziehungsweise über einen entsprechenden Algorithmus nicht zu vermeiden, was heute jedoch kein Problem für den Entwickler mehr darstellt, da dies über Rechner automatisiert erfolgen kann. Liegt umgekehrt von Anfang an die Tabelle vor, ist die Implementation der Schaltung auf ein ROM direkt möglich. Eine Umsetzung dann in Boolesche Gleichungen ist nicht mehr notwendig.

## Einfaches und erstes Steuerwerk:



Setzt man nun vor solch ein ROM einen Zähler, wird kontinuierlich die Tabelle abgearbeitet. Dies ist, wie Sie wissen, die einfachste Struktur eines programmierbaren Steuerwerkes, das für z-Schritte den trivialen Zustandsgrafen nach Bild oben aufweist. Für die Verwendung als Programmschaltwerk in einem Mikroprozessor fehlen noch entscheidende Funktionen, um flexibel auf die verschiedenen Möglichkeiten, die bei einem Mikroprozessor (oder aber auch Signalprozessor) benötigt werden. Von dieser Struktur haben aber die SPSen (Speicherprogrammierbare Steuerungen, engl. PLC: Programmable Logic Controller) ihren Namen, der heute allerdings nicht mehr zutrifft und trotzdem so verwendet wird. Heute beinhalten SPSen (= PLCs) als zentrale Einheit nur noch Mikroprozessoren oder sogar IPCs (Industrie-Personalcomputer), also nicht nur einfache "programmierbare" Ablaufsteuerungen. Nebenbei, die eigentliche Übersetzung von SPS ins Englische lautet "Stored-program Controller", eine Bezeichnung, die in der Praxis niemand verwendet.

SPSen werden in der Vorlesung "Feldbusse" angesprochen.

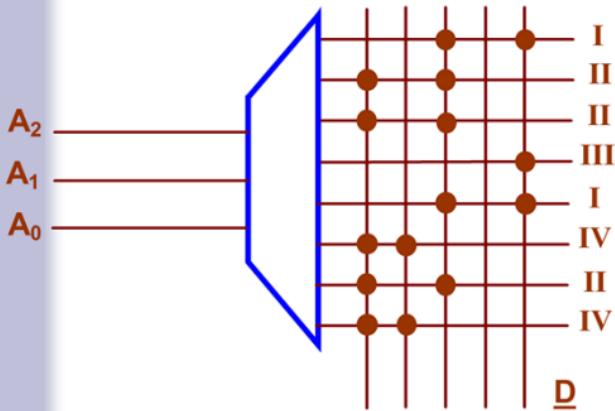
Zusatzbemerkung: Das Verhältnis der Eingänge zu den Ausgängen des Decoders kann tatsächlich so sein; siehe das folgende Beispiel.

### Ausgangsdecoder

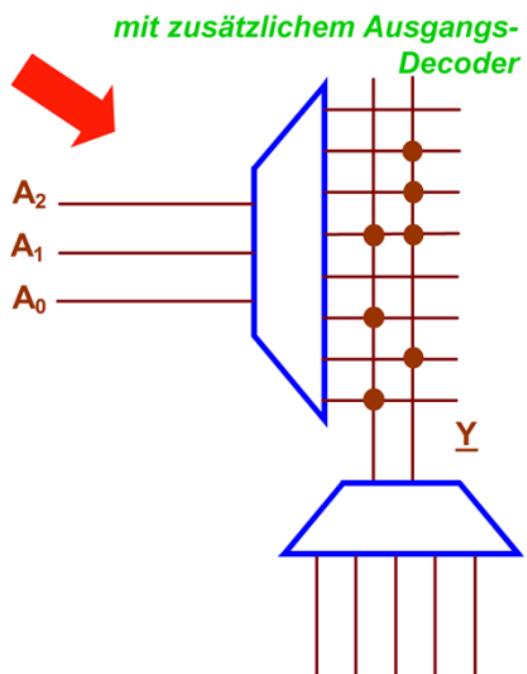
Angenommen sei als Beispiel die Realisierung eines Programmschaltwerkes nach Bild oben, das die Ausgangsgrößen  $D = Y$  gemäß dem Impulsdiagramm der folgenden Slides generiert.

## Beispiel für ein einfaches Steuerwerk

AzÜ wnv



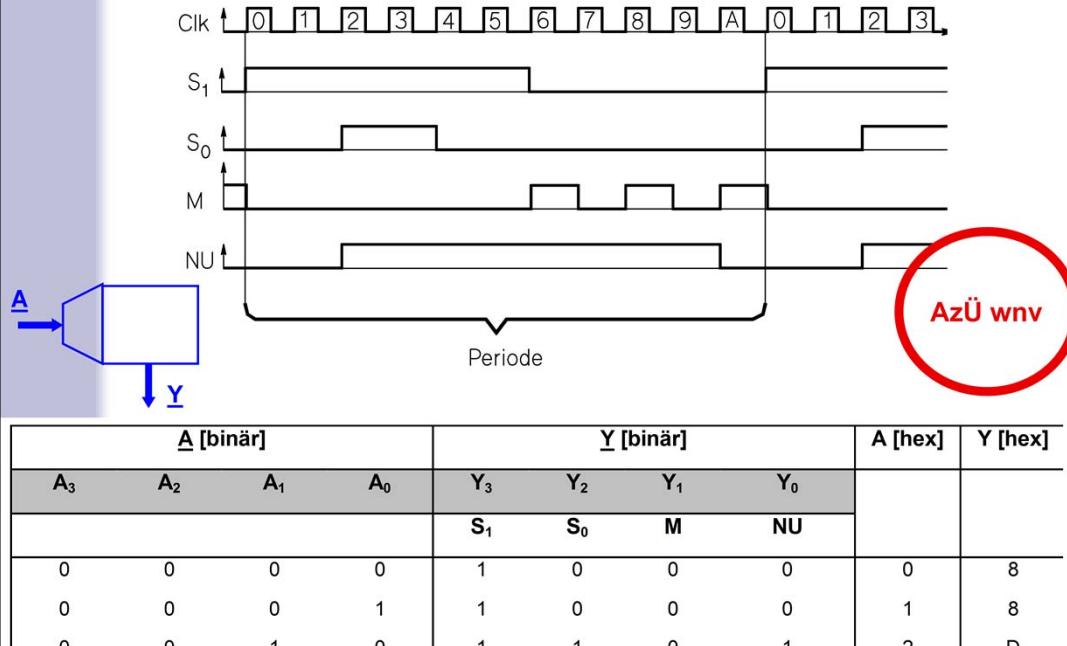
ohne zusätzlichem  
Ausgangs-Decoder



mit zusätzlichem Ausgangs-  
Decoder

$A_2$	$A_1$	$A_0$	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	Nr.	Y <sub>1</sub>	Y <sub>0</sub>
0	0	0	0	0	1	0	1	I	0	0
0	0	1	1	0	1	0	0	II	0	1
0	1	0	1	0	1	0	0	II	0	1
0	1	1	0	0	0	0	1	III	1	1
1	0	0	0	0	1	0	1	I	0	0
1	0	1	1	1	0	0	0	IV	1	0
1	1	0	1	0	1	0	0	II	0	1
1	1	1	1	1	0	0	0	IV	1	0

## Beispiel für ein Steuerwerk für ein zugrunde gelegtes Timing-Diagramm



Clk sei der Clock-Eingang; alle anderen angegebenen Variablen stellen Ausgangsgrößen dar.

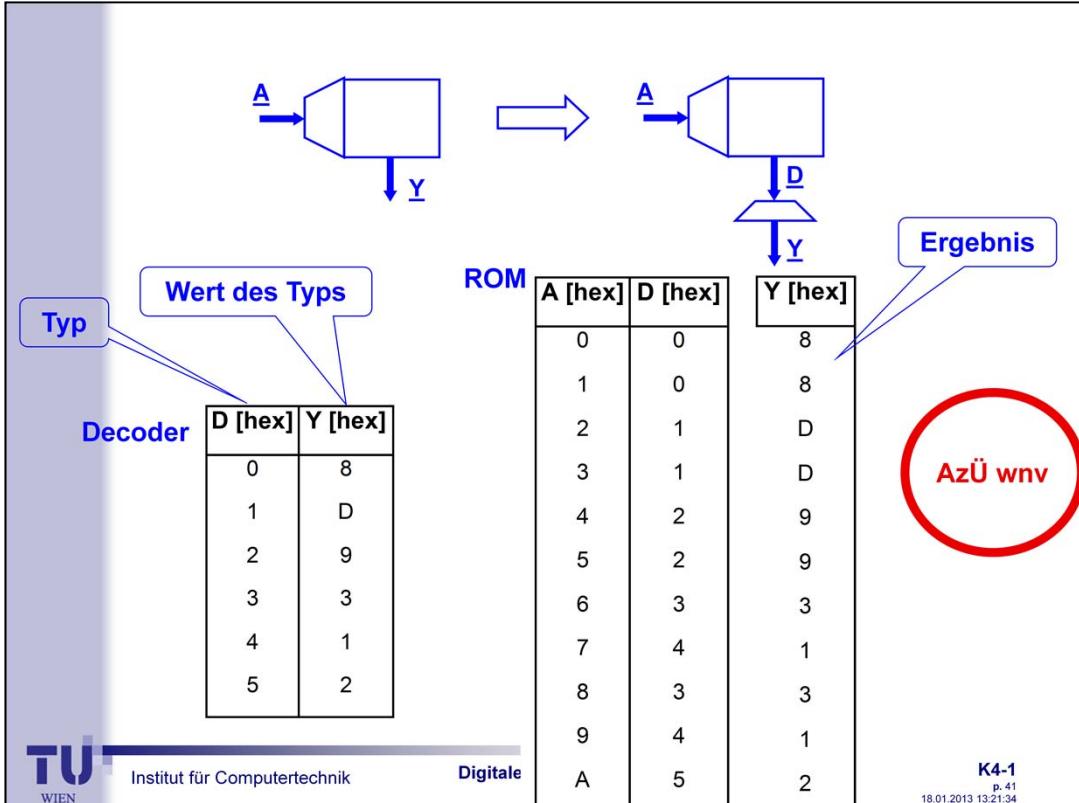
Die Tabelle links kann direkt aus dem Impulsdiagramm gewonnen werden. Wie aus den Hexziffern der letzten Spalte zu entnehmen ist, ergeben sich nur 6 unterschiedliche Typen (= 6 unterschiedliche Werte), was bedeutet, dass die Leitungen D<sub>3</sub> = Y<sub>3</sub> bis D<sub>0</sub> = Y<sub>0</sub> im ROM auf drei reduziert werden können.

Zusatzbemerkung: Zähler inkrementiert oder dekrementiert bedeutet: n := n+1 oder n := n-1

<u>A</u> [binär]				<u>Y</u> [binär]				A [hex]	Y [hex]	Typ
A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	Y <sub>3</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>			
0	0	0	0	1	0	0	0	0	8	0
0	0	0	1	1	0	0	0	1	8	0
0	0	1	0	1	1	0	1	2	D	1
0	0	1	1	1	1	0	1	3	D	1
0	1	0	0	1	0	0	1	4	9	2
0	1	0	1	1	0	0	1	5	9	2
0	1	1	0	0	0	1	1	6	3	3
0	1	1	1	0	0	0	1	7	1	4
1	0	0	0	0	0	1	1	8	3	3
1	0	0	1	0	0	0	1	9	1	4
1	0	1	0	0	0	1	0	A	2	5



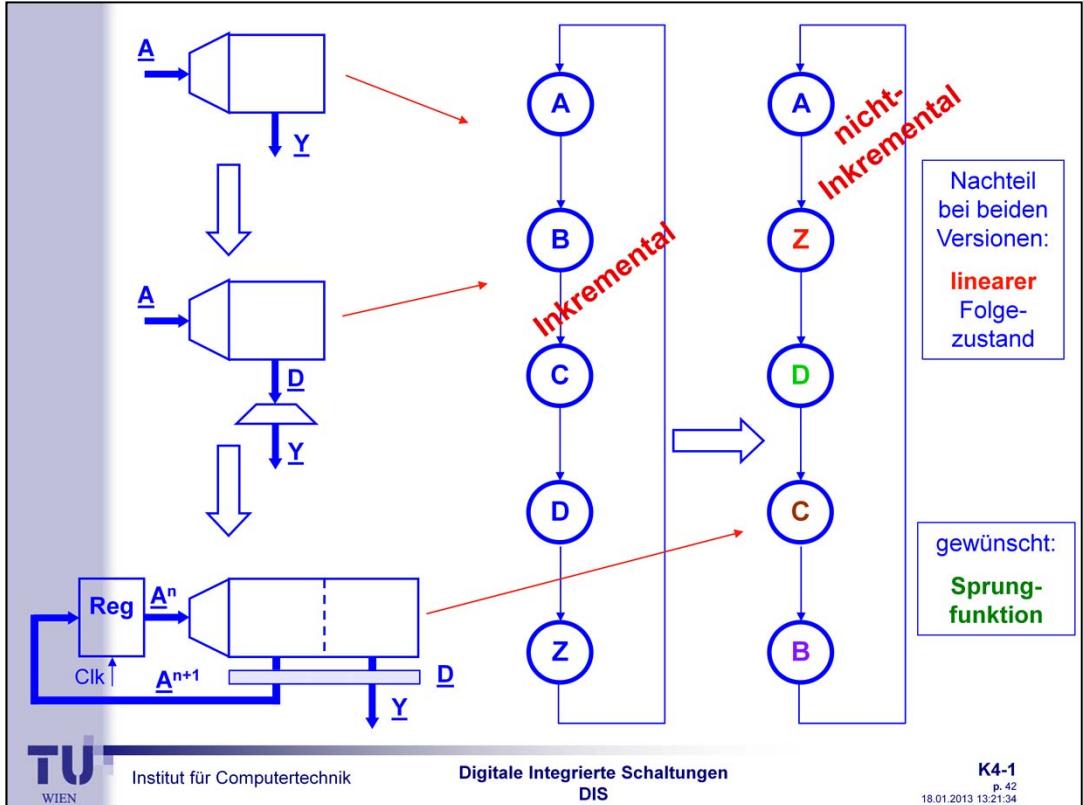
AzÜ wnv



Der hierfür notwendige Decoder ist oben wiedergegeben. Hatte man vorher also eine Ablaufschaltung gemäß Bild ganz oben links, gewinnt man durch die Integration des Decoders eine Darstellung nach Bild rechts. Der Eingangsvektor des Decoders ist mit D und sein Ausgangsvektor mit Y bezeichnet. Die daraus abzuleitende ROM-Tabelle ist unten angegeben.

Man erkennt an diesem Beispiel, dass in diesem Fall eine Decodierung möglich, aber nicht sinnvoll ist, da die Einsparung von 4 Speicherzellen im ROM in keinem Verhältnis steht zu dem Mehraufwand, der durch den zusätzlichen Decoder aufgebracht werden muss. Sind es jedoch nicht nur 4 Tabellenzeilen, sondern 1000 und mehr, und beträgt die Anzahl der Ausgangsleitungen nicht 4, sondern 80 und mehr, wird diese Art von Schaltung wirtschaftlich interessant.

Das Beispiel mit dem Decoder wird an dieser Stelle angeführt, da im Bereich der Speicherprogrammierbaren Steuerungen (SPS) dieser Typ eines Steuerwerkes tatsächlich oft Verwendung fand. Ansonsten stellt das Steuerwerk *eine* von vielen Möglichkeiten der *ROM-Flächenoptimierung* dar, die noch ausführlich behandelt werden. In der folgenden Erläuterung soll dem Decoder in dieser Konstellation zunächst keine Beachtung mehr geschenkt werden.



## Rückkopplung

Ein Zähler der vorgestellten Art kann entweder nur inkrementieren oder dekrementieren. Der Nachteil einer derartigen Steuerung liegt auf der Hand: Für Programmschaltwerke wird im Allgemeinen auch die Möglichkeit der dedizierten Schrittvorgabe gewünscht. Gemeint ist damit, dass im Funktionsablauf auch die *Folgeadressen* programmierbar sein müssen. Dafür muss der Zähler durch ein Register ersetzt werden, und im ROM werden zusätzlich zum Steuerausgang  $Y^n$  auch die Folgeadressen  $A^{n+1}$  gespeichert, was zu einer Rückkopplung führt.

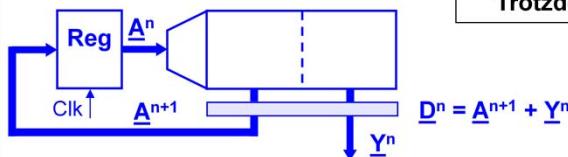
Dies war ein entscheidender Schritt hin zum heutigen Mikroprozessor:

**die Einführung der Rückkopplung in das Steuerwerk.**

Das ROM enthält nun die *Folgeadressmatrix*  $A^{n+1}$  und die *Steuermatrix*  $Y^n$ . Für das ROM ergibt sich somit:

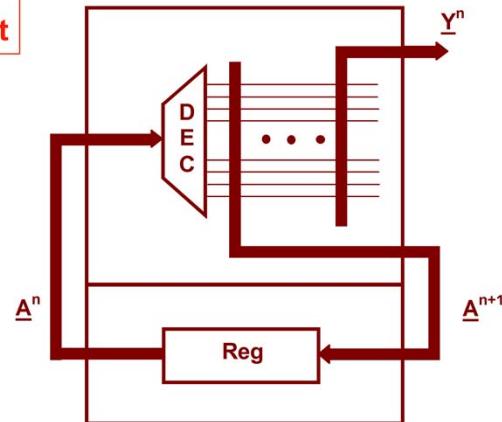
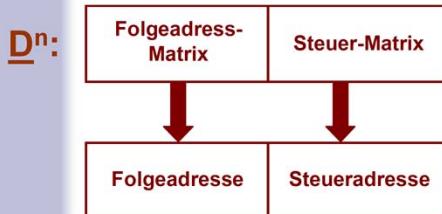
$$D^n = A^{n+1} + Y^n .$$

Zusatzbemerkung: Dies ist keine asynchrone Rückkopplung, da ein Register dazwischen geschaltet ist.



erstmals:

D<sup>n</sup> : Steuerleitungen = Datenwort

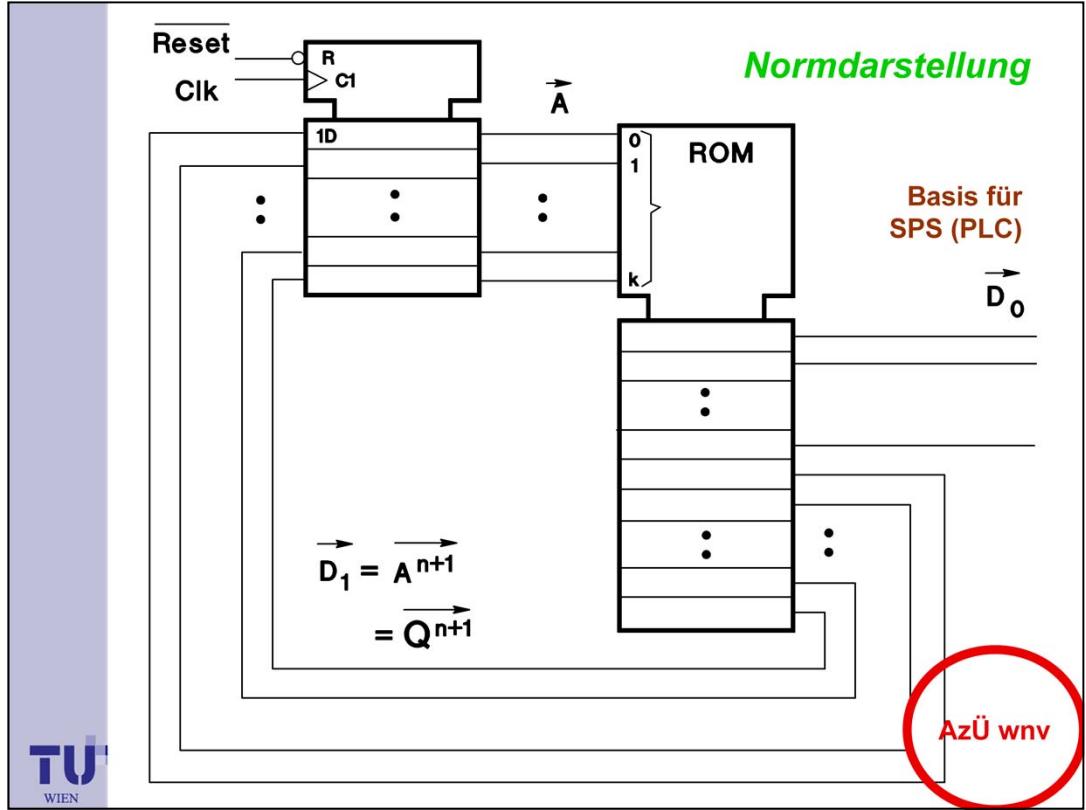


$$D^n = A^{n+1} + Y^n$$

Diese Definition führt zur Definition des des *Mikrobefehls* und kann grafisch entsprechend Bild oben dargestellt werden.

Zusatzbemerkung: Warum die Kombination Folgeadresse und Steuerwort "*Mikrobefehl*" genannt wird, wird noch erläutert.

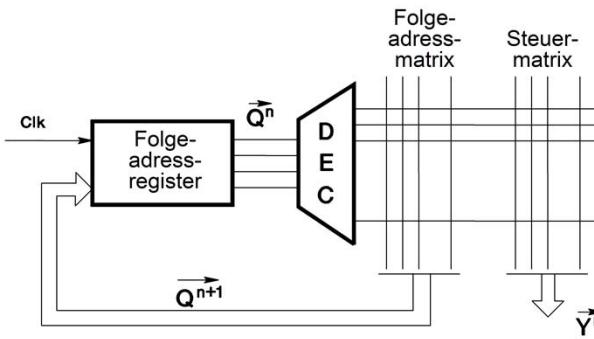
Wird nun Bild ganz oben zum Bild unten umgezeichnet, erkennt man, dass das System den autonomen Mealy-Automaten darstellt. Die Darstellung verdeutlicht, dass ein entscheidendes Leistungsmerkmal für das effiziente Steuerwerk noch fehlt: die Eingangsgröße des Systems zur gezielten Beeinflussung des Steuerwerkes. Doch bevor darauf näher eingegangen wird, soll das System nach Bild oben anhand eines Beispiels detaillierter erläutert werden, da dann die weiteren Beispiele einfacher zu verstehen sind.



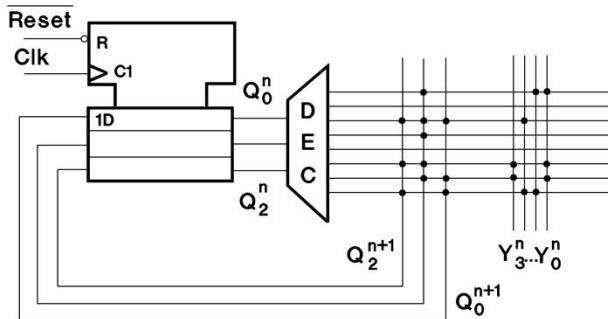
Um den Bezug zur praktischen Realisierung herzustellen, ist das Prinzip zunächst in Normdarstellung wiedergegeben. In der realen Schaltung muss selbstverständlich der Reset berücksichtigt werden, da die Schaltung ohne Reset nicht funktionstüchtig wäre. Denn unabhängig davon, dass ein Steuerablauf stets in einem Anfangszustand  $Z_0$  beginnen sollte (von Ausnahmen wie Lauflichtern einmal abgesehen), kann es ohne Reset vorkommen, dass bei einer Teilprogrammierung des ROM's, und das wird in der Praxis aus wirtschaftlichen Gründen im Allgemeinen der Fall sein, der Funktionsablauf in einem nicht programmierten Adressbereich des ROM's beginnt. Selbstverständlich, dass dann das Steuerwerk nicht den gewünschten Ablauf nimmt!

SPS: Speicherprogrammierbare Steuerung

PLC: Programmable Logic Circuit



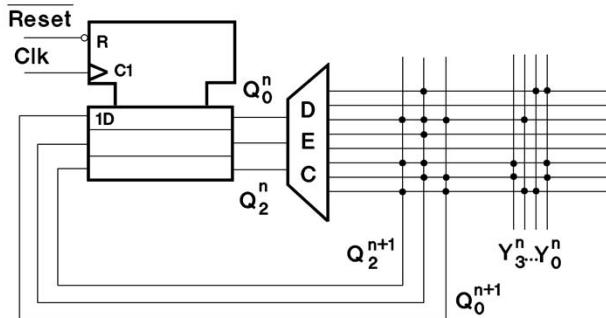
Ablaufsteuerung mit  
programmierbarem  
Folgeschritt



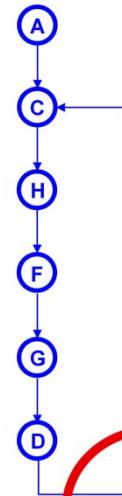
konkretes Beispiel

AzÜ wnv

Im Bild ist die prinzipielle Darstellung etwas detaillierter wiedergegeben, so dass der Schritt zum konkreten Beispiel (vorhergehendes Slide) einfach wird. Die Übergangs- und Programmiertabelle des angenommenen Beispiels zeigt im folgenden Slide.



konkretes Beispiel



AzÜ wnv

K4-1  
p. 46  
18.5.2013 13:21:34

Zustand	Z <sup>n</sup>			Z <sup>n+1</sup>			Y <sup>n</sup>				A [hex]	Y [hex]
	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Y <sub>3</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>		
A	0	0	0	0	1	0	0	0	1	1	0	23
B	0	0	1	x	x	x	x	x	x	x	1	0
C	0	1	0	1	1	1	0	1	0	0	2	74
D	0	1	1	0	1	0	0	0	0	0	3	20
E	1	0	0	x	x	x	x	x	x	x	4	0
F	1	0	1	1	1	0	1	0	0	1	5	69
G	1	1	0	0	1	1	1	0	0	1	6	39
H	1	1	1	1	0	1	0	1	1	0	7	56



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

Der Ablauf beginnt mit dem Zustand  $Z_0$ , einem Zustand, der im weiteren Ablauf nicht mehr eingenommen wird. Die übrige Zustandsfolge ist nun zwar beliebig, doch bezüglich der Ausgangsdaten ist die Wertefolge ebenfalls über ein Programmschaltwerk zu erreichen, wie es vorher gezeigt wurde.

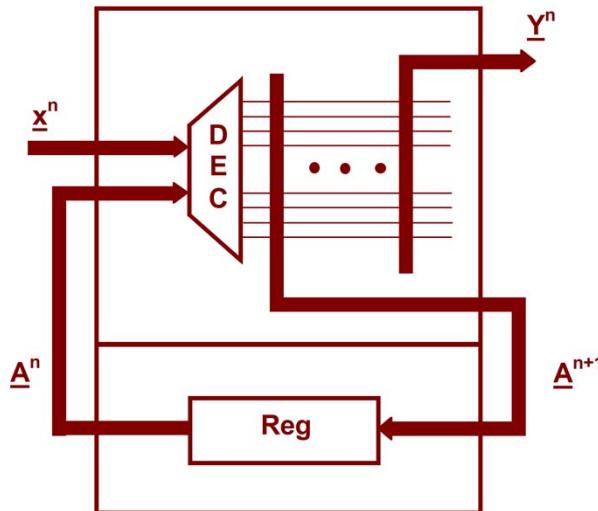
Der Gewinn liegt demnach ausschließlich in der beliebigen Rücksprungadresse. Die Möglichkeit, *unbedingte* Sprünge zuzulassen, reicht also nicht aus. Eine entscheidende Verbesserung der Schaltung wird erst dann erreicht, wenn zusätzlich *bedingte* Sprünge zugelassen werden, das heißt, wenn der interne Ablauf in Abhängigkeit von bestimmten externen Zuständen (Ereignissen) beeinflussbar wird.

Die Zustände B und E sind nicht definiert. Theoretisch müssen sie deshalb auch nicht belegt werden. In der Praxis sollte man sich aber davor hüten. Im Allgemeinen legt man heute ROMs vor allem hinsichtlich des Stromverbrauches aus und der möglichst kurzen Programmierzeit. Da sie bei CMOS z. B. mit "1" vorbelegt sind, sollten deshalb solche Zustände auf diesem Wert belassen werden.

## gewünschte Sprungfunktion

Folgezustand =  
 $f(\text{Eingangsgröße}, \dots)$ :

$$\underline{A}^{n+1} = f(\underline{X}^n, \underline{A}^n)$$



Basis für  
 $\mu$ P-  
Steuerwerk

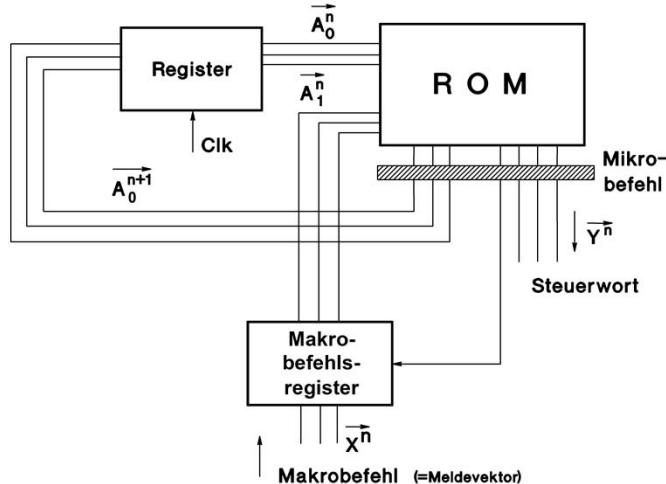
Das Bild zeigt die prinzipielle Darstellung.

In einem derartigen System wird die *Adresse des ROM's* nicht ausschließlich vom Folgeadressregister vorgegeben, sondern darüber hinaus vom "Makrobefehlsregister". Das Makrobefehlsregister enthält einen "Makrobefehl"  $\underline{X}$ , der, wiederum über das ROM steuerbar, zu bestimmten Zeitpunkten abgerufen werden kann und zusammen mit  $\underline{A}_0$  den Vektor

$$\underline{X}_i = \{\underline{A}_1 \underline{A}_0\}$$

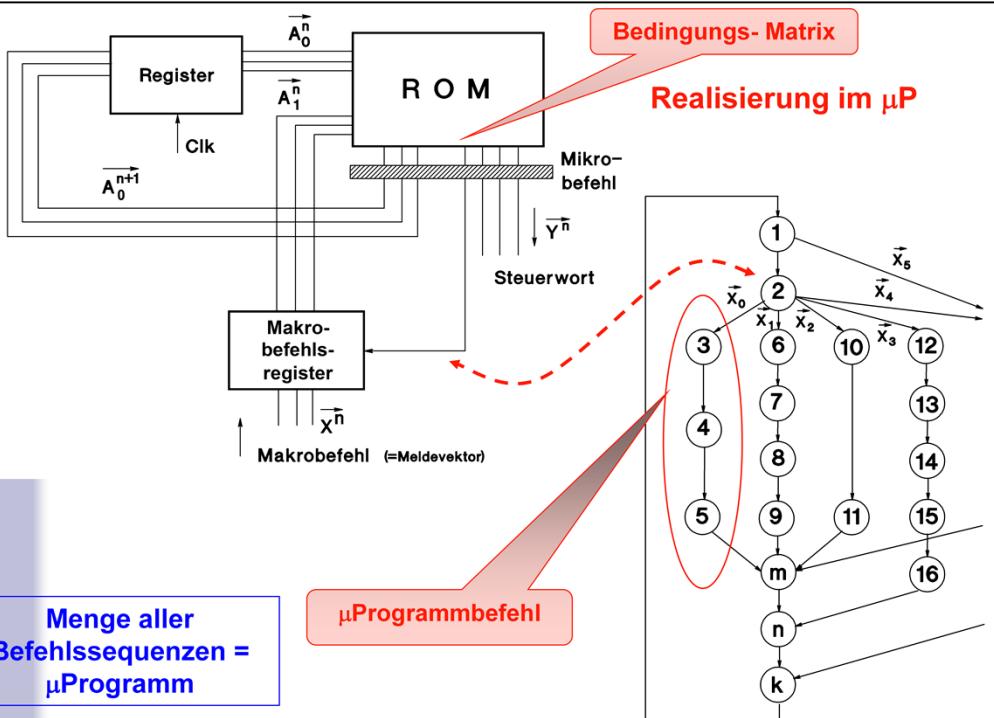
bildet.

# Realisierung im $\mu$ P



Über  $X_i$  können flexibel *Mikroprogrammabläufe* initiiert werden, die komplexere Abläufe im Prozessor steuern wie beispielsweise: Daten vom externen Speicher in die Register des Prozessors laden, gewisse Inhalte anderer Register zu dem eingeholten Wert addieren und das Ergebnis in ein spezielles Zielregister einschreiben.

Diese Architektur eines Programmsteuerwerkes, bei der die Abläufe durch Makrobefehle von außen beeinflussbar sind, führte zum Erfolg der Prozessortechnik, was dann letztendlich auch die Mikroprozessortechnik ermöglichte. Dass diese Thematik und die Konsequenzen hier noch intensiver behandelt werden müssen, wird damit verständlich. Doch zunächst ein triviales Beispiel, das das entscheidende Merkmal herausstellt.

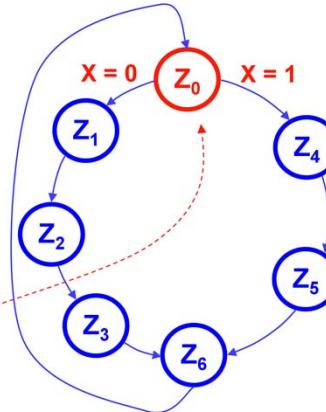


Das in Bild 4.31 gezeigte Beispiel ist einer konkreten Realisierung entnommen, wobei hier Details nicht von Interesse sein sollen. Wichtig ist das Prinzip zu verstehen.

#### Zusatzbemerkungen:

- 1: Dieses ROM hat nichts mit dem prozessorexternen ROM zu tun, in dem das Maschinenprogramm und Applikationsdaten abgelegt sind.
- 2: Die Bedeutung des Begriffs *Mikro* im Wort *Mikroprozessor* wird unterschiedlich interpretiert. Einmal wird der Begriff aus der physikalischen "Kleinheit" des Chips abgeleitet (als die ersten auf den Markt kamen, waren die Mikroprozessoren sehr klein gegenüber der diskret aufgebauten Prozessoren damaliger EDV-Anlagen (EDV: Elektronische Datenverarbeitung)), zum anderen bezieht man sich heute darauf, dass sie Miropogramme enthalten. Dies stimmte bei der "Geburt" des Mikroprozessors natürlich nicht, denn mikroprogrammierte Prozessoren gab es natürlich auch bei diskret aufgebauten Einheiten.

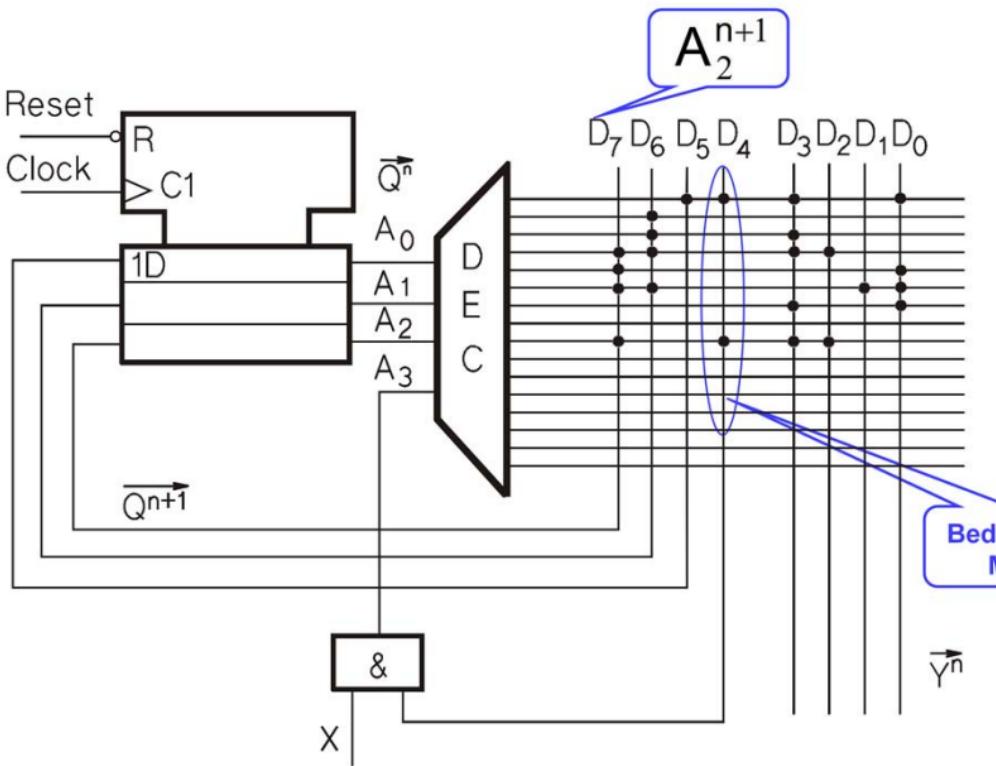
## Einfaches Beispiel



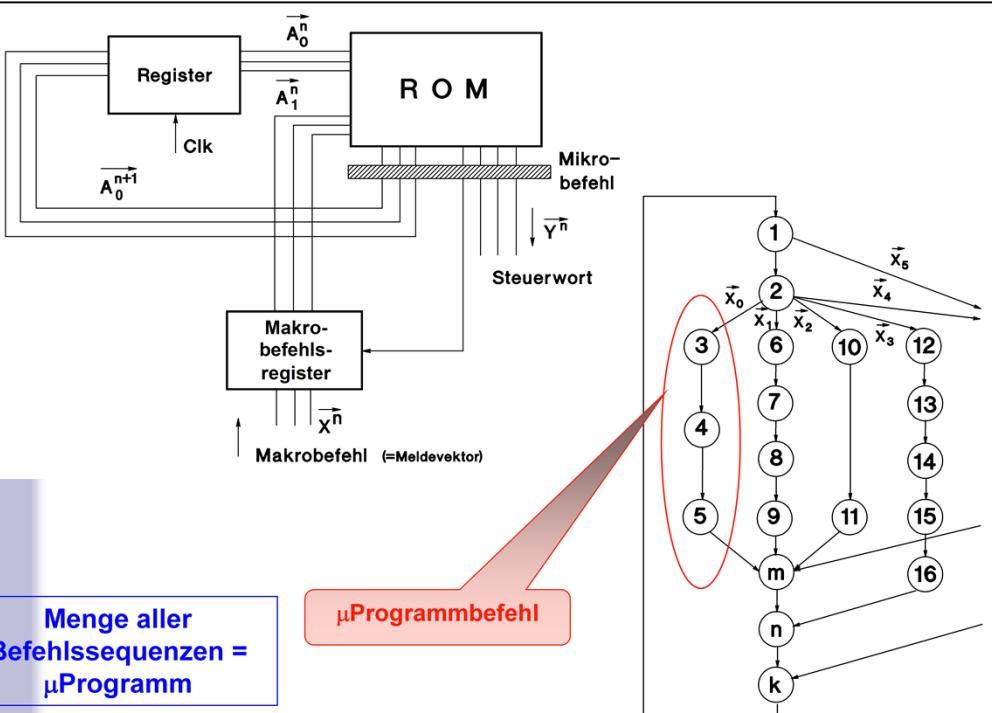
$Z^n$	$X^n$	$Q^n$	$Z^{n+1}$	$Q^{n+1}$	$Y^n$				A [hex]	Y [hex]	
		$A_3 \quad A_2 \quad A_1 \quad A_0$		$D_7 \quad D_6 \quad D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$		
$Z_0$	0	0 0 0	$Z_1$	0 0 1	1	1 0 0	1	0	0	0	39
$Z_0$	1	0 0 0	$Z_4$	1 0 0	1	1 1 0	0	0	0	8	9C
$Z_1$	0	0 0 1	$Z_2$	0 1 0	0	0 0 0	0	0	0	1	40

Im Bild ist ein hypothetisches, triviales Mikroprogrammsteuerwerk gezeigt, das zwei Maschinenbefehle zulässt. Welcher Maschinenbefehl zum Zuge kommt, entscheidet sich im Zustand  $Z_0$ . Wenn dieser eingenommen wird, muss  $D_4$  den Wert 1 annehmen, damit der Eingang X (der die Auswahl des Maschinenbefehls trifft) über  $A_3$  zur Wirkung kommt. Für alle anderen Zustände nimmt  $A_3$  durch den Einfluss von  $D_4$  den Wert 0 ein. Die Werte für das ROM können aus der Tabelle entnommen werden; die beiden rechten Spalten zeigen die zugehörigen Werte in hexadzimaler Form.

<b>Z<sup>n</sup></b>	<b>X<sup>n</sup></b>	<b>Q<sup>n</sup></b>	<b>Z<sup>n+1</sup></b>	<b>Q<sup>n+1</sup></b>	<b>Y<sup>n</sup></b>					<b>A [hex]</b>	<b>Y [hex]</b>
	A <sub>3</sub>	A <sub>2</sub> A <sub>1</sub> A <sub>0</sub>		D <sub>7</sub> D <sub>6</sub> D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>		
Z <sub>0</sub>	0	0 0 0	Z <sub>1</sub>	0 0 1	1	1 0 0 1				0	39
Z <sub>0</sub>	1	0 0 0	Z <sub>4</sub>	1 0 0	1	1 1 0 0				8	9C
Z <sub>1</sub>	0	0 0 1	Z <sub>2</sub>	0 1 0	0	0 0 0 0				1	40
Z <sub>2</sub>	0	0 1 0	Z <sub>3</sub>	0 1 1	0	1 0 0 0				2	68
Z <sub>3</sub>	0	0 1 1	Z <sub>6</sub>	1 1 0	0	1 1 0 0				3	CC
Z <sub>4</sub>	0	1 0 0	Z <sub>5</sub>	1 0 1	0	0 0 0 1				4	A1
Z <sub>5</sub>	0	1 0 1	Z <sub>6</sub>	1 1 0	0	0 0 1 1				5	C3
Z <sub>6</sub>	0	1 1 0	Z <sub>0</sub>	0 0 0	0	1 0 0 1				6	9



$Z^n$	$X^n$	$Q^n$	$Z^{n+1}$	$Q^{n+1}$	$Y^n$				$A$ [hex]	$Y$ [hex]
	$A_3$	$A_2$ $A_1$ $A_0$		$D_7$ $D_6$ $D_5$ $D_4$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$	
$Z_0$	0	0 0 0	$Z_1$	0 0 1	1	1 0 0 1	0	0	1	0 39



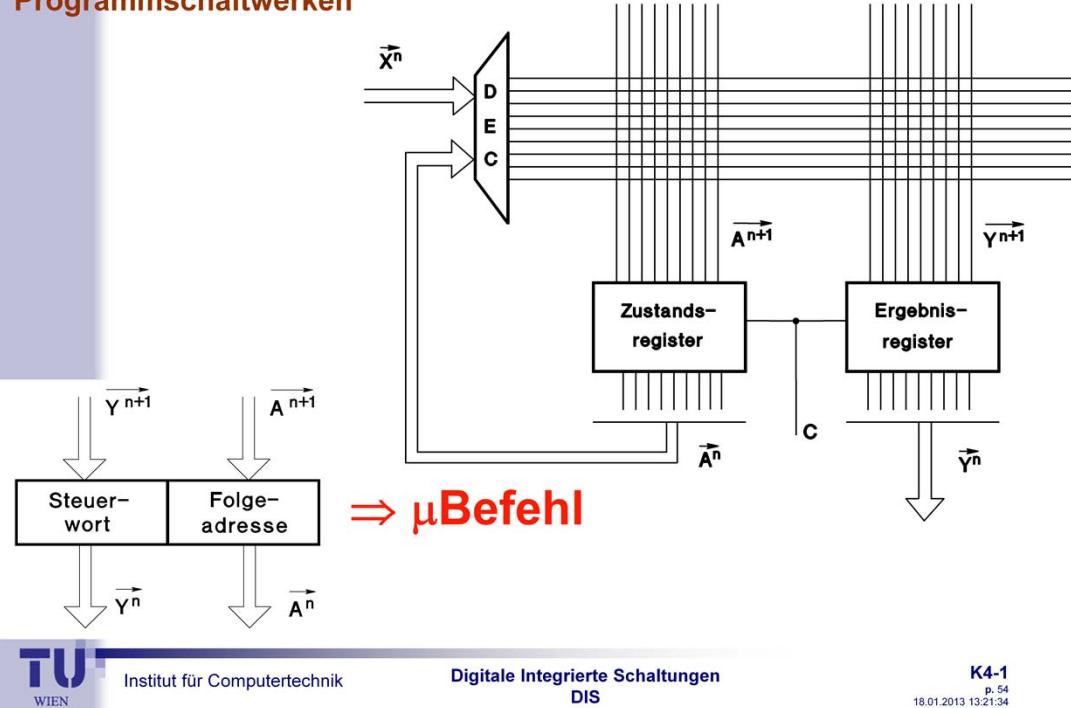
Zurück zu diesem Bild. Der Anwender solcher *mikroprogrammierten* Bausteine braucht sich nicht mehr um die komplexen internen, schaltungstechnisch oft schwer zu übersehenden Abläufe sowie die Behandlung des Timings der Maschine kümmern; diese Arbeit wird ihm abgenommen. Er kann sich auf die für ihn wesentlichere (auf einem abstrakt höheren Level liegende) Aufgaben konzentrieren: die Ablauffolge der *Makrobefehle*, die in der Mikroprozessortechnik **Maschinenbefehle** genannt werden.

In realen Prozessoren ist das Datum  $\underline{X}$  (siehe Bild oben) stark aufgegliedert. Jede dieser Folgen (in Bild rechts unten beispielsweise  $\{3, 4, 5, m, n, k\}$  oder  $\{6, 7, 8, 9, m, n, k\}$  oder ..) repräsentiert einen konkreten Maschinenbefehl, wobei der Vektor  $\underline{X}$  nur eine Komponente des Maschinenbefehles (Operationskodes) ausmacht, der zweite Teil der Komponente wird dem Operationswerk direkt zugeführt, worauf am Ende des Kapitels noch eingegangen wird.

Diese Überlegungen zählen zur Kerndefinition der unter dem Namen von-Neumann-Rechners bekannten Architektur von Prozessoren allgemein. Daten und Programm liegen in einem Speicher vor. Das System ist programmierbar und lässt flexibel Adressierungen zu. Es ist allein ablauffähig.

Zu dem Begriff *Makrobefehl* ist folgendes hinzuzufügen: In der vorliegenden Ausführung wird das Wort synonym zum Wort *Maschinenbefehl* verwendet, um eindeutig vom Mikrobefehl abzugrenzen. Es ist ein Begriff, der in der Literatur oft in diesem Zusammenhang verwendet wird. Trotzdem ist er mit Vorsicht zu verwenden, da er sich in der Assembler-Technik (die heute entscheidend von der Informatik vertreten wird) unter einer völlig anderen Bedeutung durchgesetzt hat. In der Assembler-Technik (also im Bereich der Informatik) spricht man von *Makroprogrammen*, wenn Sequenzen von Maschinenbefehlen über einen Makrobefehl während der Übersetzung eines Programms in das Programm selbst eingefügt werden, um somit Unterprogrammaufrufe vermeiden zu können.

## Realisierungen von Programmschaltwerken



Jedes Steuerwerk eines Mikroprozessors (RISC-Prozessoren seien zunächst ausgeklammert) enthält ein ROM gemäß der prinzipiellen Darstellung nach Bild oben. Die in das ROM programmierten Steuersequenzen, die Liste der Mikrobefehle, also das *Mikroprogramm*, das ursprünglich als *Firmware* (= Summe der Mikrobefehle) definiert wurde, stellt den untersten Sprach-Level in der Sprachhierarchie (siehe folgende Folie) der Rechnertechnik dar (deshalb auch die Begriffsdefinition *Mikroprogramm*).

Dass heute mit dem Begriff Firmware Schindluder getrieben wird, indem es oft für jegliche Software verwendet wird, die in einem ROM gespeichert wird, ist bedauerlich, aber nicht zu ändern.

Von Bild rechts oben ausgehend kann dann Bild links unten abgeleitet werden, das die Definition des Mikrobefehls sehr schön verdeutlicht, wenn man die klassische Bezeichnerreihenfolge eines Maschinenbefehls in der Assemblertechnik berücksichtigt (die zeichnerische Anordnung der beiden Register links und rechts im Bild müssen danach entsprechend vertauscht werden).

Der prinzipielle Verhalten des Steuerwerkes wird dadurch natürlich nicht verändert, doch zeigt das Programmschaltwerk nach Bild oben ein anderes Verhalten als das nach Bild eine Folie vorher, was einem bewusst werden muss. Erstens ist in Bild oben kein Makroregister eingezeichnet. Das bedeutet, die Synchronisation des Vektors  $\underline{X}^n$  mit dem Vektor  $\underline{A}^n$  muss an anderer Stelle geklärt sein. Zweitens wird in Bild oben der Steuerausgangsvektor des ROMs analog zum Folgeadressvektor  $\underline{A}^{n+1}$  mit  $\underline{Y}^{n+1}$  bezeichnet, der erst nach dem **Ergebnisregister** zum  $\underline{Y}^n$  wird. Durch die Einführung des Ergebnisregisters ist dies auch sinnvoll, da der aktuelle Wert, der außen ansteht,  $\underline{Y}^n$  ist, und  $\underline{Y}^{n+1}$  ist der Wert, der sich nach dem nächsten Taktwechsel erst einstellen wird.

Stellt man nun die beiden Darstellungsprinzipien einander gegenüber, wird deutlich, warum im Fall nach Bild vorher der *Mikrobefehl* der Gleichung

$$\underline{D}^n = \underline{A}^{n+1} + \underline{Y}^n$$

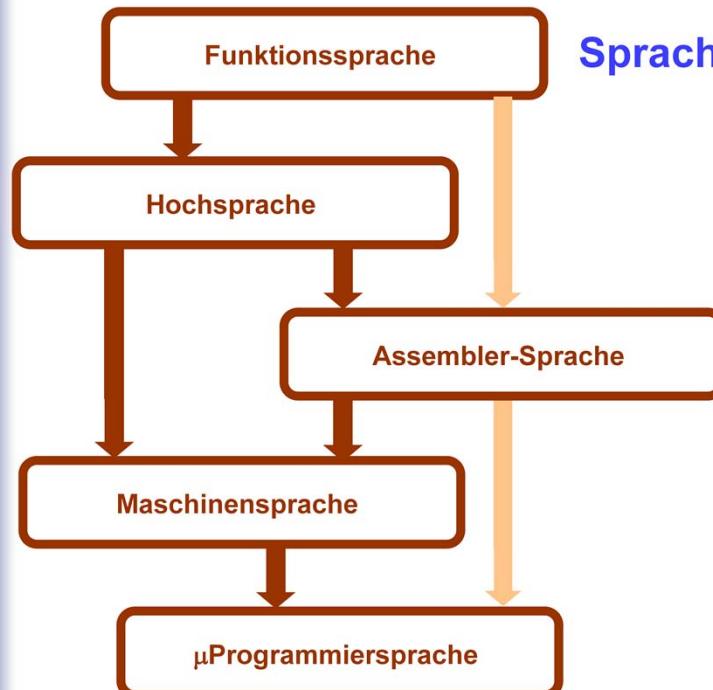
und im Fall nach Bild oben der Gleichung

$$\underline{D}^n = \underline{A}^{n+1} + \underline{Y}^{n+1}$$

gehorchen.

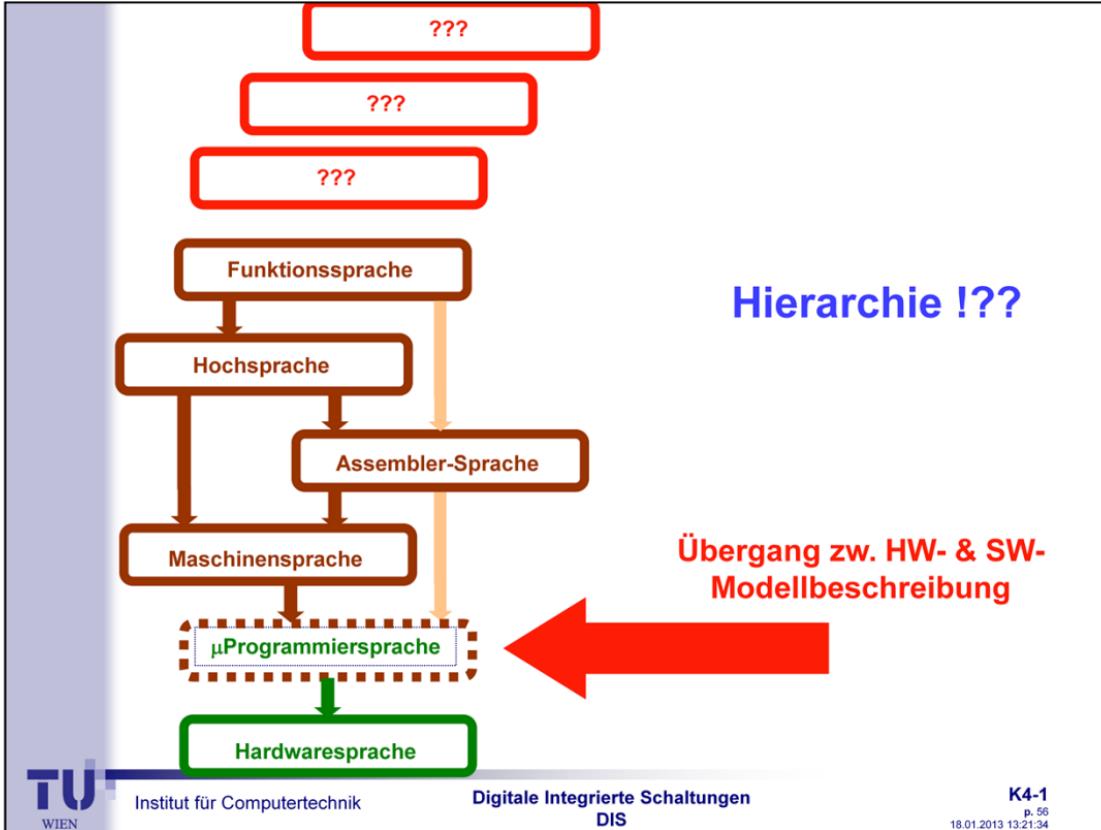
Betrachtet man nun das Bild der vorhergehenden Folie unter dem Aspekt der eingezzeichneten Register, wird erkennbar, dass Bild oben imgrunde unvollständig dargestellt ist.

## Sprachhierarchie



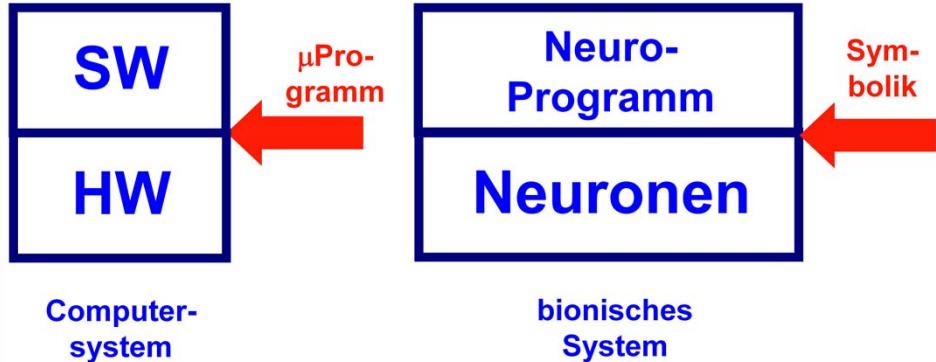
Wie schon erläutert, stellen die Verzweigungen der Mikroprogrammabläufe dar, die über Makrobefehle initiiert werden. Die Makrobefehle stellen somit die zweite Stufe in der Sprachhierarchie dar. Die Makrobefehle sind, wie ebenfalls schon erläutert, die sogenannten *Maschinenbefehle*, die wiederum die unterste Ebene bilden, auf die der Anwender bei einem *mikroprogrammierten* Mikroprozessor Einfluss hat. Da jedoch ein Befehl in Maschinensprache im Grunde eine Bitkombination darstellt und damit schwer lesbar ist, weist man den unterschiedlichen Maschinenbefehlen Bezeichnungen zu, die als die Assembler-Befehle eines Prozessors definiert sind. Die vom Anwender zu erstellende Liste von Assembler-Befehlen ist das Assembler-Programm. Hat es der Anwender (oder eine Maschine) geschrieben, muss es zunächst in das Maschinenprogramm übersetzt werden, bevor es der Prozessor "verstehen" kann. Die Übersetzung läuft im Allgemeinen automatisiert mit einem *Assembler* oder *Cross-Assembler*. Läuft die Übersetzung auf dem Zielrechner selbst, spricht man vom *Assembler*, läuft sie auf einem Fremdrechner, spricht man vom *Cross-Assembler*.

Zusatzbemerkung: Es gab tatsächlich zu "DEC"-Zeiten, bei der man mit einer Art Assembler auch Mikro-Code schreiben konnte.



# Bedeutung des Steuerwerkes

*Das Programm im Steuerwerk gilt als Schnittstelle zwischen Hardware und Software.*



*Will man Steuersysteme (intelligente Systeme) verstehen, muss man ihre Schnittstelle zwischen HW und SW "verstehen".*

Der entscheidende Punkt hier ist hier ein weit über das bisher Gesagte hinausgehender: Es wird mit der Integration eines Steuerwerkes nach Bild oben eine Mikrowelt und eine Makrowelt (deshalb auch die Differenzierung zwischen *Mikrobefehl* und *Makrobefehl*) definiert, und das *Mikroprogramm* wird als

Schnittstelle zwischen Hardware und Software

angesehen, was ein generell entscheidendes Element für das Verständnis von Rechnern auf Basis der Mikroprogrammierung darstellt.

Zusatzbemerkung: Solch eine Schnittstelle zwischen der Neurologie und Psychoanalyse zu definieren, ist eine der großen Herausforderungen in der Wissenschaft heute.

## Bitte denken Sie daran:

Am 24.1.2012 führe ich eine Prüfungsvorbereitung durch. Bereiten Sie sich sehr gut darauf vor. Dort können Sie theoretische Fragen stellen – allerdings werden dort KEINE Aufgaben durchgerechnet oder versucht zu lösen.

Für Aufgaben ist allein  
Herr Pongratz zuständig.



Der entscheidende Punkt hier ist hier ein weit über das bisher Gesagte hinausgehender: Es wird mit der Integration eines Steuerwerkes nach Bild oben eine Mikrowelt und eine Makrowelt (deshalb auch die Differenzierung zwischen *Mikrobefehl* und *Makrobefehl*) definiert, und das *Mikroprogramm* wird als

Schnittstelle zwischen Hardware und Software

angesehen, was ein generell entscheidendes Element für das Verständnis von Rechnern auf Basis der Mikroprogrammierung darstellt.

Zusatzbemerkung: Solch eine Schnittstelle zwischen der Neurologie und Psychoanalyse zu definieren, ist eine der großen Herausforderungen in der Wissenschaft heute.

# Erasmus

Lissabon

Aveiro

Barcelona

Valencia

Léon

...!!...



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K4-1

p. 59

18.01.2013 13:21:04

Der entscheidende Punkt hier ist hier ein weit über das bisher Gesagte hinausgehender: Es wird mit der Integration eines Steuerwerkes nach Bild oben eine Mikrowelt und eine Makrowelt (deshalb auch die Differenzierung zwischen *Mikrobefehl* und *Makrobefehl*) definiert, und das *Mikroprogramm* wird als

Schnittstelle zwischen Hardware und Software

angesehen, was ein generell entscheidendes Element für das Verständnis von Rechnern auf Basis der Mikroprogrammierung darstellt.

Zusatzbemerkung: Solch eine Schnittstelle zwischen der Neurologie und Psychoanalyse zu definieren, ist eine der großen Herausforderungen in der Wissenschaft heute.

# Digitale Integrierte Schaltungen

384.086  
Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

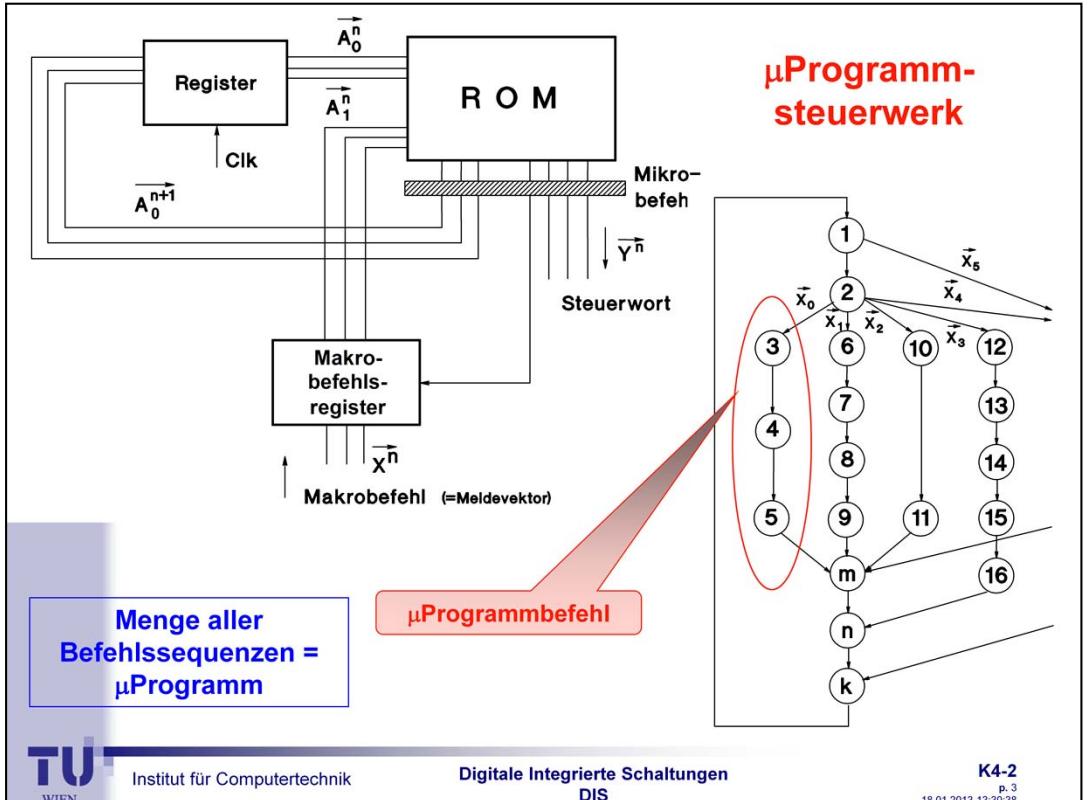
Institut für Computertechnik  
[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



# Kapitel 4

## Teil 2

# Schaltwerke hoher Komplexität



Da dieses Kapitel entscheidend ist, kurz eine Wiederholung mit zusätzlichen Erläuterungen.

Die Definition Macrobefehl hat nichts mit der Definition in der SW-Technik zu tun, wo ebenfalls der Begriff Makrobefehl Verwendung findet, aber unter einer völlig anderen Bedeutung.

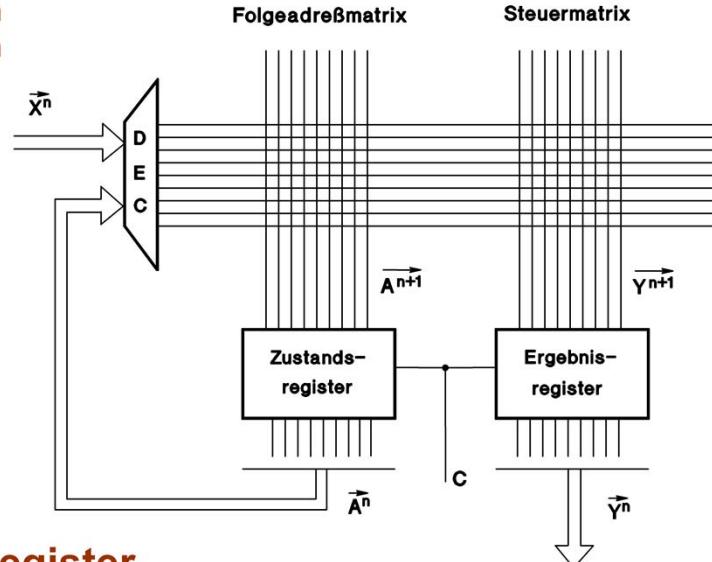
Noch eine Bemerkung zum Zustandsgrafen. Die Anzahl der festgelegten Assemblerbefehle ist gleich der Anzahl der Verzweigungen im Zustandsgraf. Dabei können aus Aufwandsgründen nicht alle denkbaren Mikroprogrammabläufe in einem Mikroprogramm implementiert werden, da die Chip-Fläche eines realen Prozessors begrenzt ist. Funktionen, die seltener verwendet werden, implementiert man nicht, sondern man erwartet, dass sie der Anwender in der höheren Programmiersprache (Assembler- oder Hochsprache) formuliert. Die Programmausführungszeit nimmt dadurch allerdings zu - der Overhead wird größer. Somit ist die Festlegung der unterschiedlichen Maschinenbefehle immer ein Kompromiss in verschiedener Hinsicht und will wohl überlegt sein.

Ein entscheidender Grund für den Overhead ist die Notwendigkeit, dass bei der Bearbeitung eines Maschinenbefehles, also der jeweiligen vertikalen Sequenzfolge im Zustandsgrafen, der Durchlauf von Zuständen erforderlich ist, die einen koordinierten Beginn und Abschluss einer Mikrobefehlssequenz garantieren - und das kostet eben Zeit.

Die Diskrepanz wird deutlich: Je mehr Funktionen man in eine Mikrobefehlssequenz packt, umso "schneller" und somit leistungsfähiger bezüglich einer bestimmten Funktion wird der Prozessor, da der Overhead abnimmt (umso schneller kann der Prozessor die jeweilige Funktion erfüllen), doch umso komplexer wird der Prozessor auch. Und komplexe Strukturen haben schwerwiegende Nachteile. Der erste Nachteil ist offensichtlich: Komplexität eines Programmsteuerwerkes bedeutet, dass Chip-Fläche dafür bereitgestellt werden muss, die für andere Prozessoreinheiten verloren geht, beispielsweise für einen Cache, der wesentlich zur Prozessorbeschleunigung beitragen kann. Komplexität des Programmsteuerwerkes bedeutet aber auch, dass größere Laufzeiten auf Leitungen berücksichtigt werden müssen, was die Taktfrequenz einschränkt. Doch das wohl interessanteste ist, was bei einer Untersuchung /Host87/ in den siebziger Jahren festgestellt wurde, dass "... in der IBM360 zehn der etwa 200

Instruktionen einen Anteil von 80% aller ausgeführten Instruktionen ..., 21 schon 95% und 30 gar 99% ..." darstellten. Das bedeutet, dass viele Instruktionen selten verwendet wurden, entweder weil der Entwickler (Anwender) sie nicht annehmen möchte, oder weil sie definitiv selten notwendig sind. Oder sind sie zu spezifisch? (Ähnliche Widersprüche ergeben sich auch bei der Festlegung von Makrozellen bei programmierbaren Bausteinen, was im Kapitel 5 näher erläutert wird.)

## Realisierungen von Programmschaltwerken

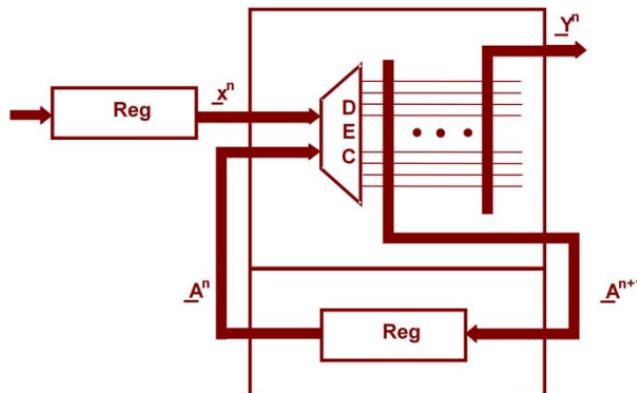


**Wo sind Register  
angebracht?**

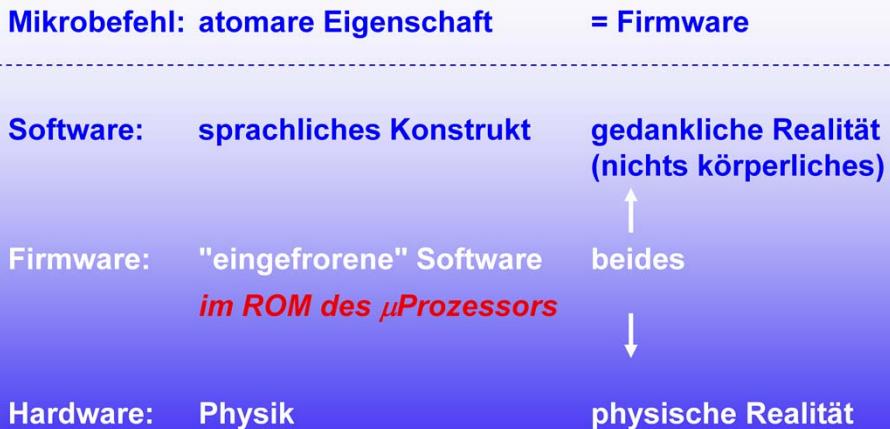
Dass es unter diesen Umständen unsinnig ist, die Funktionsabläufe im Zustandsgrafen zunehmend komplexer zu gestalten, ist einsichtig. Die gegenteilige Meinung jedoch, dass ein derartiges Programmschaltwerk nur noch historisch interessant ist, und allein dem RISC die Zukunft gehört, ist genauso unsinnig. Die Wahrheit liegt, wie so oft, irgendwo dazwischen. Was bedeutet dies für die Mikroprozessorentwicklung? Für den jeweiligen Anwendungsfall ist das Optimum zu suchen, wobei die technischen Einflussfaktoren nicht immer die ausschlaggebenden Faktoren sein können.

Eine weitere Frage ergibt sich: Wo bringt man sinnvoller Weise Register zur Zwischenspeicherung an?

# Mealy-Automatendarstellung



Die Konfiguration von der Darstellung der vorhergehenden Seite, übertragen auf die Darstellungsweise des Mealy-Automaten, ergibt die obige Darstellung. Das bedeutet, der Makrobefehl gelangt zunächst an das Makroregister und wird dann mit dem nächsten Takt erst dem eigentlichen Mealy-Automaten zugeführt.

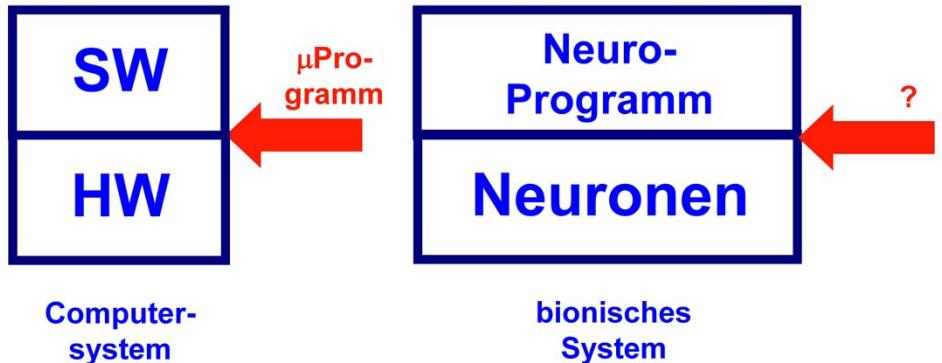


*Vorsicht: Was ist Realität?*

Was ist demnach Software? Nichts anderes als Hardware in einer anderen Beschreibungssprache, in einer anderen Abstraktionssprache. Mit anderen Worten: Die Software kann ohne Hardware nicht "leben". Sie kann nur über eine Hardware (=Körper/Physik) zum Laufen gebracht werden.

# Bedeutung des Steuerwerkes

*Das Programm im Steuerwerk gilt als Schnittstelle zwischen Hardware und Software.*



*Will man Steuersysteme (= intelligente Systeme) verstehen, muss man die Schnittstelle zwischen HW + SW "verstehen".*

.. sonst tappt man im Dunkeln, wie die Neurologie ..bzw. Psychoanalyse.

# **Was ist also Software?**

**Hardware ist Materie und  
was ist SW?**

# Für Naturwissenschaftler: $E = mc^2$

## Information - Wissensbasis - ..

.., das heißt, die Information steckt in der Struktur, in der HW ..

## Für die Naturwissenschaft ist beides dasselbe.

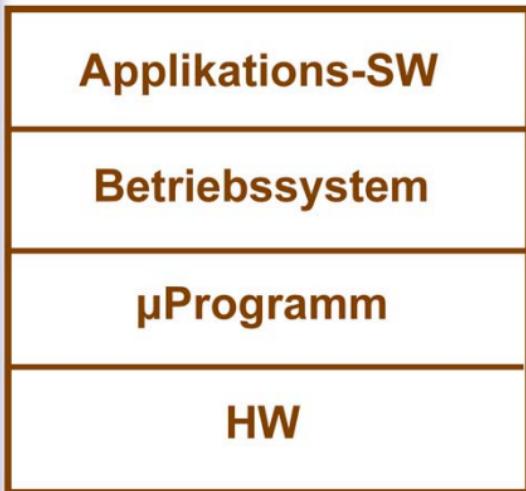
*Die Seelenwanderung ist eine andere  
Geschichte ...*

.. Dualismus zwischen Geist und Seele ..? Freud war einer der ersten, der diesem Spuk ein Ende bereitete ..



Für den Naturwissenschaftler gibt es die Materie = Energie und die Information, die in ihr liegt. Letztendlich gibt es also nur Materie = Energie. Diese Philosophie wird Monismus genannt. HW und SW ist somit dasselbe, nur eine andere Beschreibungsform.

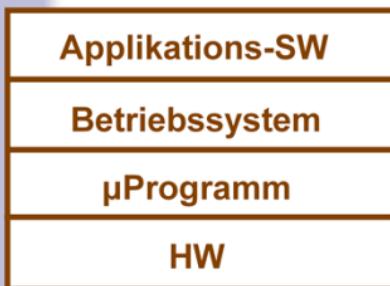
Information ist ein erdachtes Modell von der Veränderung des physikalischen Verhaltens. Für die Naturwissenschaft gibt es also nur Materie = Energie. Aber all diese Begriffe sind Modelle, die die Realität mehr oder weniger gut beschreiben. Also ist auch Hardware ein Modell sowie Software. Und in diesem Sinne ist somit die Software ein Teil der Hardware, die aus der Beschreibung der Hardware herausgenommen wurde.



Wissen, Information

**verschiedene  
Abstraktionslevel,  
aber immer  
dasselbe**

HW  
(= Materie)



# 3 unterschiedliche Betrachtungs- weisen und immer dasselbe

## Stichworte

- **μ (Mikro):** elementare Softwareoperationen  
*(nicht auf die Größe bezogen, sondern auf die hierarchische Stufe bzgl. SW)*
- **Firmware:** gespeicherte μBefehle
- **Definitionen:** DIN 44 300 ff
  - Statement*
  - elementare Anweisung (nicht zerlegbar)*
  - Instruktion*
  - μProgramm (Sequenz von μBefehlen)*
  - :
- **μprogrammierbare μP**
- **μprogrammierte μP**

## *Beispiele aus der Norm*

### Anweisung Statement

Nach den Regeln einer beliebigen Sprache festgelegte syntaktische Einheit, die in gegebenem oder unterstelltem Zusammenhang wie auch im Sinne dieser Sprache eine Arbeitsvorschrift ist.  
Eine Anweisung kann Teile enthalten, die Anweisungen oder Vereinbarungen sind.

### elementare Anweisung

Eine Anweisung, die sich in der benutzten Sprache nicht mehr in Teile zerlegen lässt, die selbst Anweisungen sind.

### Befehl Instruction

Eine elementare Anweisung, die insofern auf eine bestimmte Funktionseinheit bezogen ist, als sie von dieser unmittelbar oder nach Codierung ausgeführt werden kann.

### $\mu$ Programm

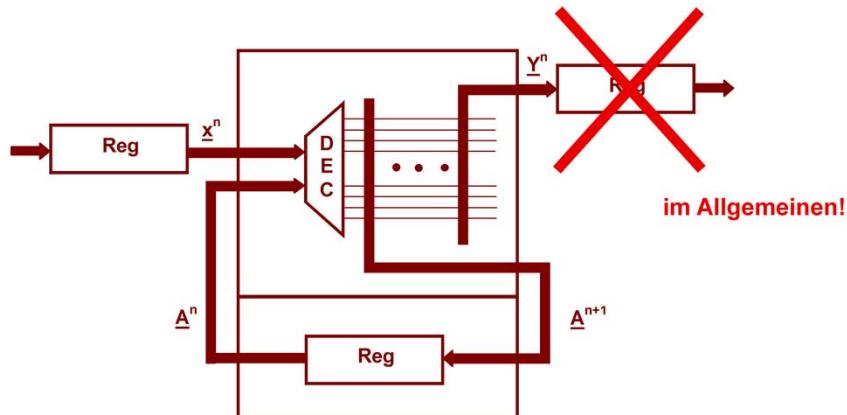
Ein Programm, das zusammen mit Baueinheiten eine Funktionseinheit bildet, deren Aufgabe es ist, einen Befehl einer Maschinensprache auszuführen.

## Idee

- $\mu P$  enthält  $\mu$ Programmspeicher (fast immer in Form eines ROMs)
- $\mu$ Befehl ( $Q^{n+1}$ ,  $Y^n$ ): hohe Datenwortbreite (zum Teil 120 bit und mehr)
- $\mu$ Programm-Datenwortbreite unabhängig von Daten- und Adresswortbreite des  $\mu P$ , und nur von der Anzahl der steuernden Funktionen sowie einer Optimierung abhängig
- $\mu$ Befehle im Allgemeinen unbekannt
- $\mu P$ ,  $\mu C$ ,  $\mu CS$  nicht anwendungsspezifisch
- Anwender ruft über Makrobefehl  $\mu$ Befehl auf (siehe Diagramm)

- $\mu P$ ,  $\mu C$ ,  $\mu CS$ : preisgünstig, flexibel einsetzbar, hohe Stückzahl möglich
- Umsetzung von "abstrakten" Aufgabenstellungen auf Programmebene einfach
- Hardware-Aufwand immer kleiner
- Entwickler braucht sich nicht mehr um einzelne Steuerleitung zu kümmern
- Entwickler braucht sich immer weniger um die HW kümmern  
(Studium "Informatik" löste sich ca. 1970 aus der Elektrotechnik und Mathematik).

# Register-Platzierungsmöglichkeiten



im Allgemeinen!

Selbstverständlich kann man auch ein Register am Ausgang integrieren, was den Vorteil beinhaltet, dass die Ergebnisse des Steuerwerkes konstant am Ausgang anliegen und in diesem Punkt nicht berücksichtigen braucht, wie das Timing der folgenden Schaltung ist.

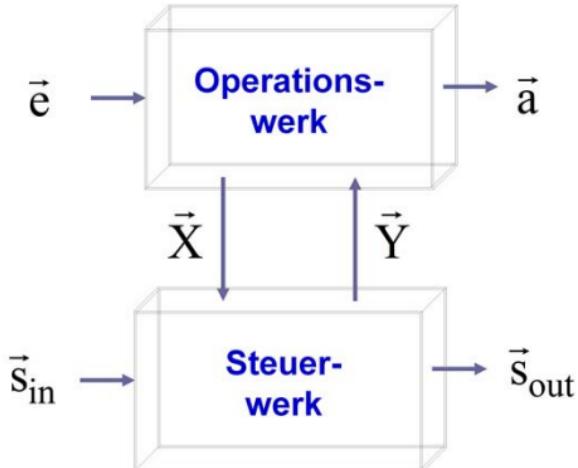
Es ist einleuchtend, dass durch das zusätzliche Ergebnisregister das Steuerwort stets einen Takt später an den Ausgang des Programmschaltwerkes gelangt. Bevor hier ein Datum, das am Eingang der Konfiguration anliegt, am Ausgang zum Tragen kommt, müssen zwei Takte vergehen. Legt man als zentrale Einheit keinen Mealy-, sondern einen Moore-Automaten zugrunde, sind sogar 3 Taktverzögerungen hinzunehmen, was in der Praxis kaum zu akzeptieren ist. Aus diesem Grund verzichtet man im Allgemeinen auf das Ergebnisregister, muss aber dann damit leben, dass das Ausgangssignal des Steuerwerkes (die Ausgangssteuerleitungen des Mikroprozessors) nur zu definierten Zeiten gültig sind.

# Aufgaben des µProgrammsteuerwerkes

- Datenpfad schalten
- ALU, Register, .. steuern
- Prozessor-Steuerausgangsleitungen setzen + rücksetzen
- Statusabfrage
- Interrupt-, Trap-Bearbeitung
- DMA-Bearbeitung
- Speicherverwaltung
- ..

daraus folgt:

- hohe Anzahl von Y- und  $s_{out}$ -Leitungen
- Optimierung der HW erforderlich



# Variationen in der Literatur

## ***Vorsicht***

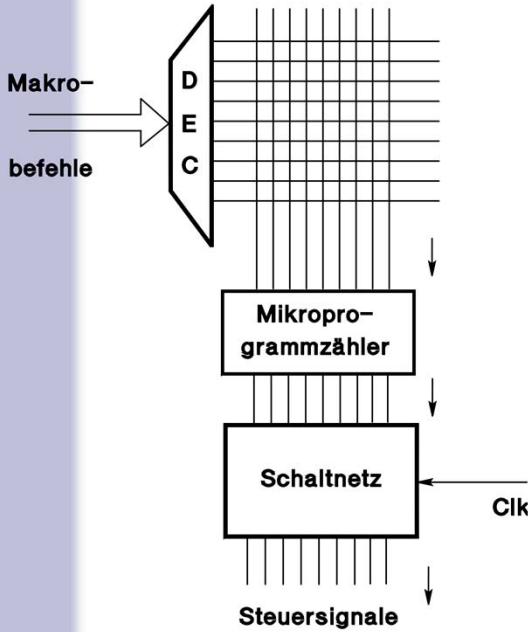
- eine einheitliche Vorstellung gibt es noch nicht
- unterschiedliche Begriffe, Vorstellungen, .. in der Literatur

## ***Beispiele:***

## **Realisierungen**

Es gibt zahlreiche Aspekte, die unter dem Begriff der Realisierung von Programmschaltwerken zu betrachten wären. Drei sollen hier herausgegriffen werden: erstens die Thematik der Mikroprogrammierung an sich, zweitens die Optimierung von Programmschaltwerken und drittens weitergehende Funktionen.

## Mikromaschine nach /Remb94/



- definiert das Schaltnetz zeitabhängig
- definiert den Zähler taktunabhängig
- die Makrobefehle in dieser Darstellung nicht fest anliegend (kein Register vorgeschaltet)
- ..

Das Mikroprogrammsteuerwerk hat, wie ausführlich dargestellt, die Aufgabe der Ansteuerung aller Register, Multiplexer, interner und externer Speicherbausteine, interner und externer Ports usw., kurz die Steuerung aller elektronischen Komponenten der Prozesssteuereinheit eines Computers. Dafür benötigt es interne und externe Statusinformationen, muss Interrupt- und DMA-Ports (DMA: Direct Memory Access) abfragen, liefert Statusinformationen, bedient Handshake-Leitungen usw. Das bedeutet, der Anwender wird durch ein programmiertes Mikroprogrammsteuerwerk (unter anderem) von der Aufgabe weitgehend befreit, sich selbst um das Timing des Prozessors zu kümmern, und kann, aus Sicht der unterschiedlichen Entwicklungsebenen der Programmiersprachen, die hierarchisch angeordnet zu sehen sind, sich den Aufgaben näher widmen, die "höher" angesiedelt sind, die der eigentlichen Anwendung näher liegen. Er muss sich nicht allzu sehr in die Hardware des Prozessors eindenken, was bedeutet, dass *mikroprogrammierte Steuerwerke* (Microprogrammed Control Units) als anwenderfreundlich zu bezeichnen sind.

Welche Architekturprinzipien einem mikroprogrammierten Steuerwerk zugrunde liegen, wird im folgenden Abschnitt behandelt. An dieser Stelle nur soviel: Die Datenwortbreite des Mikrobefehls eines CISC's (Complex Instruction Set Computer) liegt zwischen 50 bit und 150 bit (Beispiel: der Prozessor des IBM Systems /360 Model 50 mit einem 90-bit-Datenwort), kann aber nach /Remb94/ auch Werte von 400 bit erreichen. Es handelt sich somit um komplexere Schaltungen, mit denen es sich zu beschäftigen lohnt. Vorgestellt wurden Mikroprogrammsteuerungen erstmalig im Jahre 1951 von M. V. Wilkes im Jahre 1951 /Wilk51, Haye88/; Mealy (1955) und Moore (1956) entwickelten hierzu dann geeignete Automatenmodelle /Jung92/.

Interessant ist, dass mikroprogrammierte Steuerwerke in der Literatur zum Teil nicht nur unterschiedlich dargestellt, sondern dass auch unterschiedliche Prinzipien hineininterpretiert werden. So spricht /Remb94/ in diesem Zusammenhang auch von der **Mikromaschine** und meint damit eine Maschine, wie sie in Bild oben wiedergegeben ist. Sie setzt sich aus dem *Befehlsdecoder*, dem *Befehlszähler* und dem *Mikroprogrammspeicher* zusammen. Das darin dargestellte Architekturprinzip entspricht *nicht* dem Originalentwurf von M. V. Wilkes aus den beginnenden 50er Jahren und enthält vor allem zwei Aspekte, die mit den bisherigen Erläuterungen in diesem Skript nicht vereinbar sind: Erstens fehlt die Rückkopplung

(Mealy-Prinzip), und zweitens führt in das Schaltnetz der Takt, was impliziert, dass nach der vorliegenden Definition kein Schaltnetz, sondern ein Schaltwerk vorliegt. Dann aber ist der Mikrogrammzähler kein Zähler (bestehend aus FFs), sondern in unserem Sinn ein Schaltnetz.

# RISC & CISC

## **Vorsicht**

- Was ist ein CISC?
- Was ist ein RISC?
- Vorteile – Nachteile?

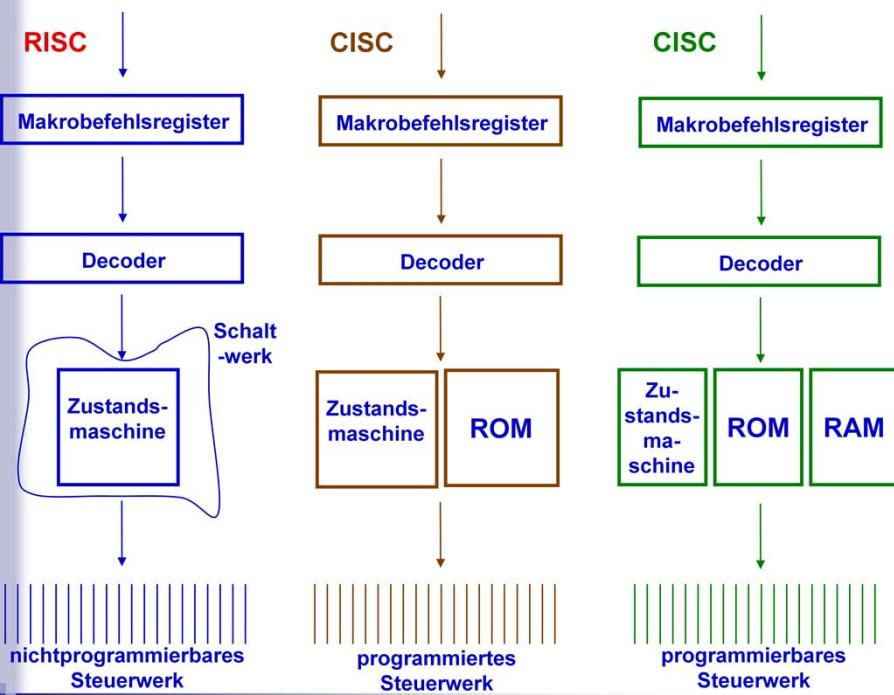
***Steuerwerksprinzipien hierfür:***

# Schaltwerk für einen RISC und einen CISC

- **CISC: Complex Instruction Set Computer**
  - heute alle  $\mu$ programmiert (möglichst hochwertige Maschinenbefehle zur Verfügung stellen)
  - Firmware-Änderung + Variation des Operationswerkes = neuer Baustein
  - ein  $\mu$ Kernel wird oft als Basis für ganze Prozessorfamilien entwickelt
- **RISC: Reduced Instruction Set Computer**
  - war früher wegen des Entwicklungsaufwandes "unmöglich"
  - heute Entwicklung des Steueranteils über bspw. VHDL unproblematisch
- **$\mu$ programmierbares Steuerwerk**  
*(dynamically microprogrammable Control Unit)*
  - Bit-Slice-Prozessoren (SW-Problematik: Assembler-Generierung jeweils neu)

-neuer Baustein: zum Beispiel, um intelligente I/O-Bausteine zu realisieren oder entsprechende Signalprozessoren.

## Gegenüberstellung eines Programmschaltwerkes für:



In einem *nichtmikroprogrammierbaren Steuerwerk* ist kein ROM enthalten und die enthaltene Zustandsmaschine ist hardwareverdrahtet. Das bedeutet, der Befehlsdecodierer des Makrobefehls ist direkt auf ein Schaltwerk geführt, das die Steuerausgangsleitungen liefert, typisch für das Steuerwerk eines *RISC*-Prozessors, auf den noch an anderer Stelle eingegangen wird. (Eine Bemerkung zum Verständnis: Die Steuerung des *RISC* kann auf eine direkte Beeinflussung der Komponenten im Operationswerk reduziert werden, ohne dass eine Zustandsmaschine zugrunde gelegt wird.)

Zur Verdeutlichung ist der Zusammenhang in Bild oben abstrahiert dargestellt. Die mittlere Komponentenfolge zeigt das Programmsteuerwerk des in diesem Kapitel abgeleiteten Programmsteuerwerkes nach dem Prinzip von *Wilke* (*CISC*-Architektur). Links wird die Struktur des Programmsteuerwerkes eines *RISC*-Prozessors gezeigt. Der rechte Teil des Bildes gibt die Struktur des programmierbaren Steuerwerkes wieder, worauf weiter unten noch eingegangen werden soll. Deutlich wird durch diese Darstellung, dass beim Einsatz eines Steuerwerks des *RISC*-Prozessors ein komplexer Registersatz für die Zustandsmaschine eingespart wird, was natürlich für das Timing der Steuerung sehr vorteilhaft ist. Zudem entfällt das ROM, was in Mikroprozessoren oft einen großen Teil der Fläche des Chips ausmacht.

In Literatur wird darauf hingewiesen, dass *RISCs* den Nachteil beinhalten, dass bei einer Änderung der Architektur des Chips das Schaltwerk für das Operationswerk für die Steuerung neu zu redesignen ist. Das stimmt! Doch stellt dies in der Praxis inzwischen kein Problem mehr dar, da ein Schaltwerk dieser Art heute mit entsprechenden Compilern entwickelt wird, mit denen das Schaltwerk auch relativ einfach zu optimieren ist. Vorauszusetzen sind dabei Schaltungsentwurfssprachen wie VHDL (Kapitel 6).

**Zusatzbemerkung:** Zudem ist es Aufgabe der praktischen Übung, dass jeder Student in seiner Gruppe vollständig einen vereinfachten *RISC*-Prozessor mit Steuerwerk entwirft, der in einen programmierbaren Baustein (*ASIC*) zu implementieren ist. Zu empfehlen ist deshalb auf jeden Fall weitergehende Literatur über *RISC*-Prozessoren /Bund88, Bode88, Lieb03/.



CISC-Prozessoren sind heute alle *mikroprogrammiert* und nicht mehr mikroprogrammierbar. Man sagt zwar, mikroprogrammierbare Mikroprozessoren sind "anwenderfreundlich", doch wird mit diesem Begriff in diesem Fall leichtfertig umgegangen. Die Behauptung läuft darauf hinaus, dass durch die Veränderung der Firmware (des Mikroprogramms) "schnell spezielle intelligente" Bausteine geschaffen werden können, ohne dass die Architektur groß zu ändern ist. Dies muss in zweifacher Hinsicht nicht zutreffen. Erstens beschränkt sich ein Designänderungswunsch oder der Wunsch nach einem ähnlichen Baustein selten auf die Änderung des Mikroprogramms, sondern beinhaltet zumeist auch die Änderung des Operationswerkes, zweitens kann gerade die Änderung eines Mikroprogramms eine sehr mühsame Angelegenheit sein, wie man aus den kleinen vorgestellten Beispielen schon ableiten kann, beziehungsweise wie man im nächsten Abschnitt noch sehen wird. Zudem ist die Änderung eines Chip-Designs an sich schon eine kostspielige Sache, wobei weniger die prinzipielle Änderung der Schaltung den hohen Aufwand verursacht, sondern im Allgemeinen das Designen des Testsystems und das erneute Routen und Simulieren des neuen Designs. (Bekanntlich kann die Änderung einer Schaltung oft in wenigen Stunden oder Tagen erfolgen, während das erneute Compilieren mit all seinen Folgen dann die eigentliche Arbeit ausmacht. In Kapitel 5 wird dies näher besprochen.)

Wesentlich häufiger wird in der Praxis heute ein ganz anderer Weg beschritten: Man definiert und entwickelt einen "Basisprozessor", den man dann als "Kernel" anderen Bausteinen zugrunde legt. So kann man beispielsweise heute Prozessoren wie den "Klassiker" 8051 als Library relativ günstig erwerben und ihn in "seinen ASIC" als zentrale Einheit integrieren. Er ist deshalb in der Praxis vielfältig in Chips in den verschiedensten Variationen anzutreffen.

*Mikroprogrammierbare Steuerwerke* (dynamically microprogrammable Control Units) fand man vor allem in *Bit-Slice-Prozessoren* vor, die in den 90er ihre Hochblüte hatten. Bit-Slice-Prozessoren sind Mikroprozessoren, die der Anwender "scheibchenweise" kaufen kann, das heißt beispielsweise 4bit-weise, und die er dann auf die gewünschte Bitbreite kaskadiert

Dass dann die Steuerleitungen weitgehend selbst zu führen sind, ist selbstverständlich. Anwendung fand diese Technik vor allem in Systemen, in denen eine extrem hohe Prozessorverarbeitungsgeschwindigkeit gefordert ist. Das heißt, in diesen Fällen kommt es auf eine optimale Architektur des Operationswerkes genauso wie auf ein optimales Steuerwerk an. Betrachtet man hierzu das Bild der Gegenüberstellung, rechte Seite. Verschiedene Varianten sind denkbar. Man kann sich ein Schaltwerk vorstellen, in das nur ein RAM und kein zusätzliches ROM integriert ist. Das Programm muss dann nach einem Reset zunächst von einer anderen intelligenten Einheit (beispielsweise einem Personal Computer) geladen werden. Man kann sich ebenso eine Variante denken, die ein EPROM und ein hochtaktbares RAM enthält, in das das Programm nach dem Reset zunächst mit einer niedrigeren Taktrate vom EPROM geladen wird. Die volle Leistungsfähigkeit erhält der Bit-Slice-Computer dann, wenn auf das schnell zu taktende RAM umgeschaltet wird. Weitere Varianten sind denkbar.

# CISC- $\mu$ Programmsteuerwerk

- **$\mu$ Programm-ROM**
- **Integration von Funktionen**
- **Optimierung**



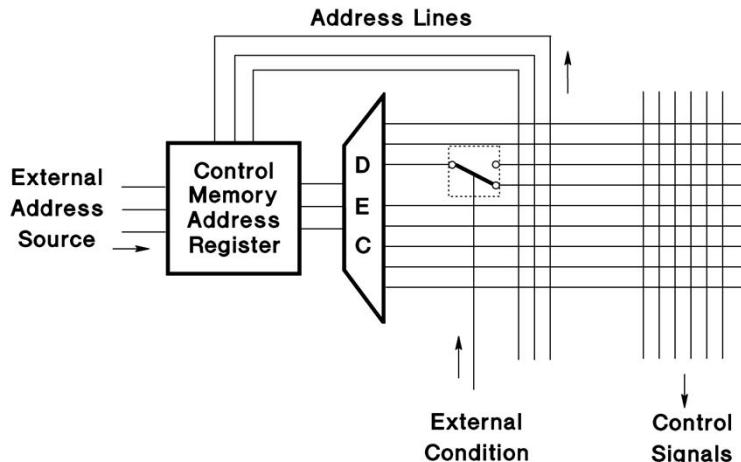
Ein Beispiel für ein Mikroprogrammierbares Steuerwerk stellt der Rechner VAX11/750 dar /Remb94/. Die Maschine hat eine Wortlänge von 80bit, enthält ein Mikro-ROM mit 6 Kbyte und ein Mikro-RAM mit 1 Kbyte. In das Befehlssystem ist der Befehl XFC (Extended Function Code) implementiert, über den der Eintritt in die Mikroprogrammierung möglich wird. Der Erfolg war gering. Das Prinzip hat sich nicht bewährt, nur an Hochschulen und Forschungsinstituten wurde diese Möglichkeit (und auch dort nur in beschränktem Umfang) umfangreicher ausgenutzt. Die Mikroprogrammierbarkeit von CISCs hat damit bis heute, mit Ausnahme der früher eingesetzten Bit-Slice-Computer, einen rein akademischen Wert. Ein entscheidender Punkt mag auch sein, dass nie effiziente Software-Tools für Firmware-Assembler sowie entsprechende Debugging-Tools zur Verfügung gestellt wurden. Der Grund ist relativ offensichtlich: Sie würden sehr teuer werden, da sie entweder mikroprozessorspezifisch oder zu kompliziert und aufwendig in ihrer Anwendung werden würden. Vom Hersteller fertig hergestellte mikroprogrammierbare Mikroprozessoren, die der Anwender noch mikroprogrammieren kann, hat es deshalb nie gegeben.

Bevor im nächsten Abschnitt auf konkrete Optimierungsmaßnahmen von Mikroprogrammschaltwerken eingegangen wird noch einführend allgemeine Bemerkungen hierzu. Wird keine Optimierung des Steuerwerkes bezüglich des ROMs vorgenommen, wird von einer *horizontalen Mikroprogrammsteuerung* gesprochen. Bei ihr wird jede Ausgangssteuерleitung und jede Folgeadressleitung der Folgeadressmatrix des ROMs (im folgenden wird wiederum vereinfachend nur vom ROM gesprochen, obwohl nach den obigen Ausführungen klar ist, dass der Speicher auch ein EPROM, RAM usw. sein kann) direkt zum Zustandsregister beziehungsweise direkt (also ohne weitere Zusammenfassung durch Codierung oder anderer Verfahren) aus dem Programmsteuerwerk herausgeführt, wie es bisher auch dargelegt wurde. Dies hat zwei entscheidende Vorteile: Erstens eine hohe Prozessorleistung erhält man unter anderem dann, wenn die Taktfrequenz des Systems möglichst weit heraufgesetzt werden kann. Das ist aber nur möglich, wenn schaltungstechnisch geringe Laufzeiten zu erzielen sind. Voraussetzung hierfür ist eine Anordnung möglichst weniger Schaltungselemente hintereinander. Zweitens können dann im Operationssteuerwerk viele Register, Multiplexer, Schalter usw. zeitparallel aktiviert werden, was wiederum bedeutet, dass viele Operationen parallel ablaufen können.

Optimierungsteuerwecks-Gitarren-Programm

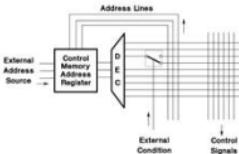
## 1. Idee

Originalentwurf nach Wilkes /Haye 88/



Ziel: Integration von Funktionen und Optimierung

Den vorher schon angesprochenen Originalentwurf von Wilkes gibt das obige Bild wieder, der sich zum heute verwendeten Prinzip nicht wesentlich unterscheidet. Wirkliche Unterschiede sind einmal im Schalter zu sehen, der direkt in die Linienstruktur des ROMs integriert ist, zum andern in der Beeinflussung des Ablaufs. Beim bisher dargestellten ist der Ablauf von der Programmierung der Bedingungsmatrix abhängig, in Bild oben hat eine Änderung der Eingangsgröße X eine direkte Wirkung.



## Programmschaltwerkoptimierung

**Fakt:** Programmwortbreite kann über 120 bit betragen

**Vorteil:** Ziel: Flächeneinsparung auf dem Chip

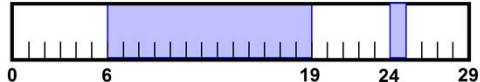
**Vorteil:** Kostenreduktion

**Nachteil:** zumeist größere Laufzeiten, da aufwendige Schaltung

**Nachteil:** klare Strukturierung und Übersicht geht evtl. verloren

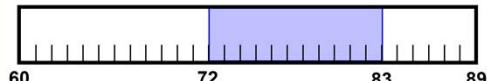
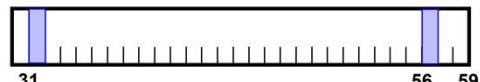
**Beispiele:**

## Horizontale μProgrammsteuerung



### IBM/360 Modell 50

- nicht schattiert: Steuerleitungen
- schattiert: Adressen, Parity-Bits, ungenutzte Bits
- hohe Informationsredundanz
- große Chip-Fläche
- hoch taktendes System
- in Literatur: "Programmierfehler leicht möglich" → Blödsinn

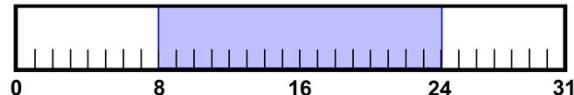


Das Bild oben zeigt einen typischen Vertreter solch einer horizontalen Mikroprogrammsteuerung: das 90-bit-Mikrowort des IBM Systems /360 Modell 50. Der große Nachteil der hohen Speicherknotenzahl wird offensichtlich. Eine derartige (informationsredundante) Programmierung kostet enorm viel Chip-Fläche, und für vielleicht zusätzliche interessante Funktionen des Operationswerkes bleibt dann kein Raum mehr.[1]

Zusatzbemerkung: Nach /Lieb93/ macht beim MC68020 das Mikroprogrammsteuerwerk den größten Teil der Chip-Fläche aus. Bei heutigen Prozessoren ist das Verhältnis nicht mehr ganz so krass, doch ist es sehr ernst zu nehmen.

Dem steht die *vertikale Mikroprogrammsteuerung* gegenüber. Ziel ist es, die Anzahl der Spalten im ROM auf ein Minimum zu reduzieren, was durch Codieren der Signale, durch zusätzlich integrierte Schalter – siehe vorhergehende Bilder – durch die Nano- und Picoprogrammierung usw. erreicht wird. Es sind Verfahren, die Gegenstand des nächsten Abschnittes sind.

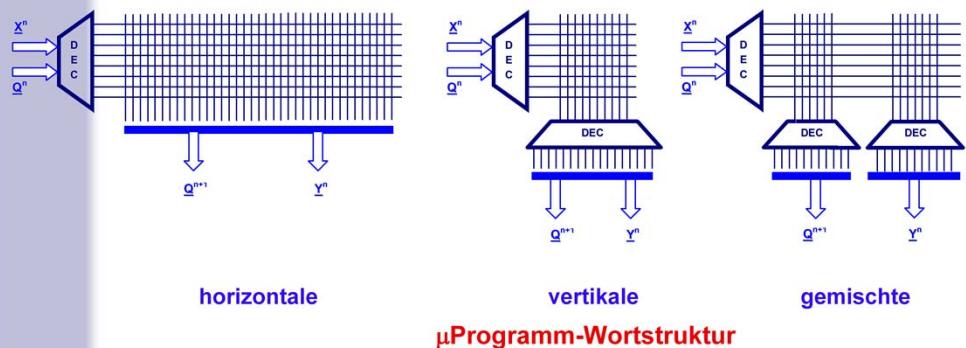
# Vertikale μProgrammsteuerung



## IBM/360 Modell 145

- nicht schattiert: Steuerleitungen
- schattiert: Adressen, Parity-Bits, ungenutzte Bits
- geringe Informationsredundanz
- kleine Chip-Fläche
- niedrig taktentdes System

Ein typischer Vertreter einer vertikalen Maschine ist die IBM System /360 Modell 145. Definiert sind zwei Operandenfelder, aus denen die Steuerleitungen generiert werden. Dazu existiert ein Adressfeld, über das Adressen für den Mikroprogrammspeicher gebildet werden können, und ein Opcode, der das spezielle Maschinenwort kennzeichnet. Das bedeutet, mit jedem Maschinenwort können nur bestimmte Mikroinstruktionen aktiviert werden, die völlige Parallelität ist reduziert auf ein gerade noch akzeptables Maß.



- horizontal:
  - ⇒ jede Ausgangsleitung spezifiziert eine Funktion (Register, MUX, ..)
  - ⇒ hohe Taktfrequenzen möglich, geringe Laufzeiten, wenige Schaltungselemente hintereinander
- vertikal
  - ⇒ geringer Speicherplatz

# Schlussfolgerung

## Horizontale Orientierung

- keine Decodierung des Steuerwortes
- hohe Informationsredundanz
- Fähigkeit viele Aktionen parallel zu initiieren
- Datenwort hat große Bit-Länge (oft größer als 120 bit)

## Vertikale Orientierung

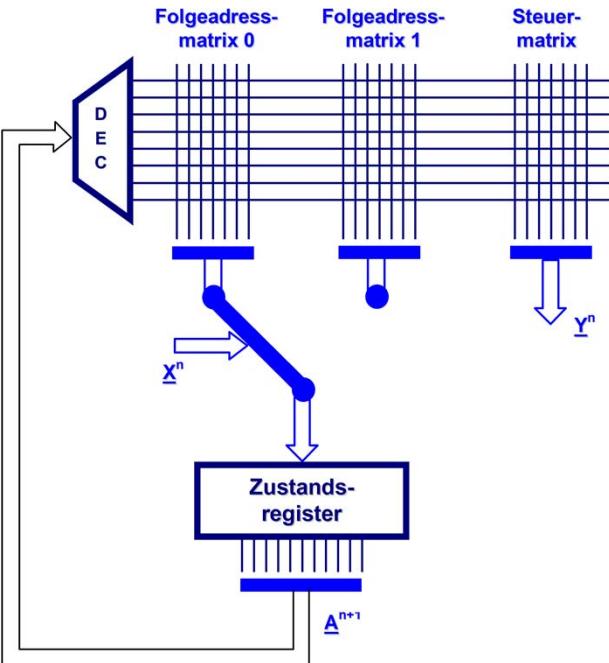
- Serialisierung von Aktionen
- Datenwort kleiner Bitlänge
- hohe Verschlüsselung der Steuerinformation im ROM

Allgemein wird davon ausgegangen, dass alle Mikroprozessoren vor allem eine möglichst hohe Taktrate haben sollten. Dieser Forderung sind zwei Aspekte hinzuzufügen. Erstens finden Mikroprozessoren als Mikrocontroller (Embedded Systems) zunehmend in Geräten, Anlagen, Gebäuden, Kraftfahrzeugen, .. Verwendung, in denen es *nicht* auf eine hohe Taktrate ankommt. Hier spielen Kriterien wie Leistungsverbrauch, Chip-Fläche, Mächtigkeit des Maschinenbefehls (daraus folgt: kleiner externer Speicher) oft eine größere Rolle. Dann aber ist die vertikale Mikroprogrammsteuerung gefragt und nicht die horizontale. Zu denken ist beispielsweise an die vielen 4-bit- und 8-bit-Controller, von denen es marktanteilsmäßig wesentlich mehr gibt als beispielsweise 32-bit- oder sogar 64-bit-Prozessoren.

Zweitens dürfte es selbstverständlich sein, dass Prozessoren heute nach der Topdown-Methode entwickelt werden. Im ersten Schritt wird deshalb im Allgemeinen die Maschinensprache (also auch die Mächtigkeit des Befehlssatzes) definiert, die ein Ergebnis der verlangten Anforderung an den zukünftigen Chip darstellt, im nächsten Schritt wird das Operationswerk spezifiziert, und erst im letzten Schritt erfolgt das Design des Programmsteuerwerkes. Dass man bei dieser Vorgehensweise zwangsläufig zumeist auf eine Mischform zwischen der horizontalen und vertikalen Technik kommt (wenn nicht die eine oder die andere von Anfang an direkt vorgegeben wird), ist verständlich.

## Vorsicht:

Nach /Jung92/ ist ein  
Moore-Automat z. B. ein  
Programmschaltwerk  
entsprechend der  
dargestellten  
Architektur  
?!?

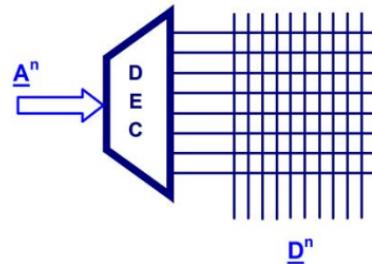


Zum Abschluss sei noch auf einen speziellen Aspekt hingewiesen. In der Literatur wird verschiedentlich bezüglich Programmschaltwerke auf die Unterscheidung zwischen Mealy- und Moore-Automaten hingewiesen. So wird beispielsweise in /Jung92/ von einem Mealy-Automaten gesprochen, wenn eine Struktur nach dem Bild vorliegt, das wir bisher kennen, von einem Moore-Automaten wird dagegen gesprochen, wenn ein *Programmschaltwerk* nach Bild oben gegeben ist. Der Kern der Aussage ist dabei, dass in einer Einheit wie der bisherigen eine Änderung ohne Verzögerung am Ausgang wirksam wird, während die Schaltung nach Bild oben eine Wirkung nur zeigen kann, wenn die Veränderung über das Zustandsregister hinausgegangen ist. Doch dies ist nicht relevant, denn eine Schaltung, wie wir es bisher zugrunde legten, kann nur funktionieren, wenn ein Register davor oder zumindest nachgeschaltet wird. Damit muss aber wiederum ein Takt Verzögerungszeit eingerechnet werden. Die beiden Schaltungen sind deshalb aus dieser Sicht äquivalent, nur dass die Anzahl der Komponenten in Bild oben (in diesem Fall Multiplexer) größer ist als in der bisherigen Darstellung, wo die Daten über die Decoder abgerufen werden - von der Anzahl der Matrixknoten einmal abgesehen.

## Konkrete Beispiele von Optimierungen

## Programmierung hier ohne Nanospeicher

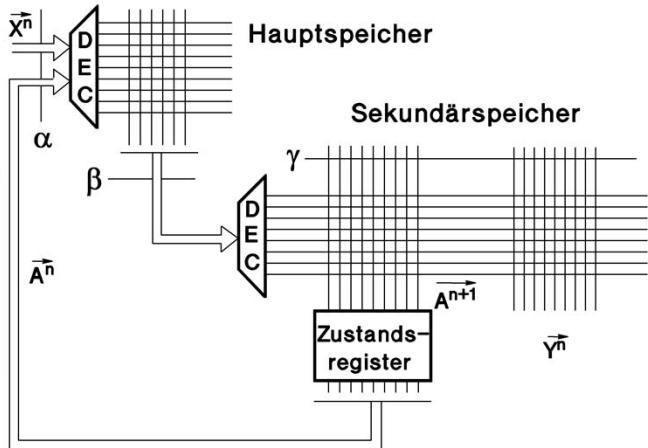
A <sup>n</sup> [hex]	D <sup>n</sup> [hex]
0	28
1	A3
2	0A
3	0A
4	F0
5	28
6	A3
7	E5
8	26
9	A3
A	0A
B	0A
C	E5
D	1A
E	1B
F	28
10	1A
11	1A
12	26
13	A3



Ein einfaches Beispiel soll dies verdeutlichen. Gegeben ist die ROM-Tabelle links oben einer rein horizontalen Konfiguration. In den wiedergegebenen 20 Zeilen sind jedoch nur 8 unterschiedliche Mikrobefehle enthalten, die in einem **Sekundärspeicher** abgelegt werden können

# Sekundärspeicher (Nanospeicher)

**n<sub>a</sub>** = ROM-Datenpunkte vom Ausgangs-ROM (ohne Nanospeicher)  
**n<sub>b</sub>** = ROM-Datenpunkte mit Nano-Speicher



$$\frac{n_a}{n_b} = \frac{2^\alpha \cdot \gamma}{2^\alpha \cdot \beta + 2^\beta \cdot \gamma} \gg 1$$

.. dann ist der Nanospeicher sinnvoll

Unter dem Begriff "Optimierung von Programmschaltwerken" soll im wesentlichen verstanden werden, welche Möglichkeiten bestehen, um von einem ersten Entwurf eines horizontal orientierten Programmschaltwerkes zu einer möglichst vertikalen Lösung zu gelangen, beziehungsweise welche Möglichkeiten einer Mikroprogrammspeicher-Flächenreduktion existieren.

## (a) Sekundärspeicher (Nanoprogrammspeicher)

Nimmt man an, dass sich in einem Mikroprogramm Mikrobefehle laufend wiederholen, bietet es sich an, sie in einem gesonderten Speicher, dem *Sekundärspeicher*, abzulegen und sie entsprechend dem gewünschten Programmablaufwunsch über den *Hauptspeicher* abzurufen. Die Programmierung des Hauptspeichers ist für die Programmbeehlfsfolge verantwortlich, da in ihm nur noch die Nummern der Mikrobefehle sequentiell abgelegt werden. Hierarchisch gesehen steht somit der Hauptspeicher über dem Sekundärspeicher, in dem die einzelnen Mikrobefehle enthalten sind. Da der Hauptspeicher in der "Mikroebene" liegt, wird der Sekundärspeicher auch *Nanospeicher* genannt.

Verhältnis Ausgangs-ROM zu Nano-ROM

Zähler =  $(2^{**\alpha})$  (=Anzahl der ROM-Zeilen) \* gamma( = Ausgangs-ROM)

Nenner = ROM-Matrix 1 + ROM-Matrix 2

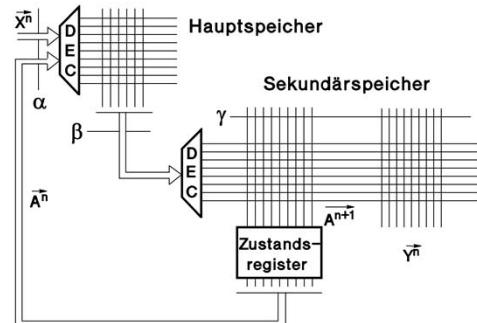
## Sekundärspeicher (Nanospeicher)

### Beispiel

$$\alpha = 10$$

$$\gamma = 80$$

$$\begin{aligned}n_a &= 2^{10} \bullet 80 \text{ bit} \\&= 81.920 \text{ bit} \\&= 10.240 \text{ byte}\end{aligned}$$



mit Nanospeicher:

$\beta = 5$  (= 32 unterschiedliche Nanospeicher-Befehle)

$$\begin{aligned}n_b &= 2^\alpha \bullet \beta + 2^\beta \bullet \gamma = 2^{10} \bullet 5 \text{ bit} + 2^5 \bullet 80 \text{ bit} = \\&= 5.120 \text{ bit} + 2.560 \text{ bit} = 7.680 \text{ bit} \\&= 960 \text{ byte}\end{aligned}$$

$n_a/n_b$  wäre ca. 11:1

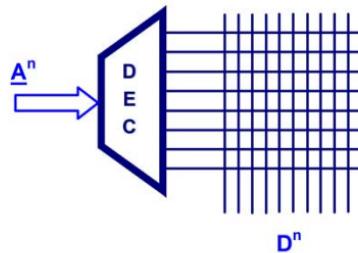
= große Speicherplatzersparnis

Wenn  $n_a$  die Anzahl der Speicherpunkte der rein horizontalen Mikroprogrammierungslösung und  $n_b$  die Speicherpunkte der Lösung mit Sekundärspeicher angibt, ist das Verhältnis  $n_a/n_b$  als Abschätzung anzusehen, ob es sich lohnt eine genaue Berechnung durchzuführen oder nicht. Ist es sogar wesentlich größer als 1, lohnt sich eine Lösung auf der Basis eines Sekundärspeichers auf jeden Fall bezüglich der Chip-Fläche.

$\alpha$ ,  $\beta$  und  $\gamma$  geben jeweils die Anzahl der Leitungen gemäß der Darstellung oben an. Die Berechnung gilt natürlich nur unter der Voraussetzung, dass alle Zeilen des Speichers realisiert werden, was in einem Mikroprozessor nicht der Fall sein muss (nicht benötigte Zeilen des ROMs können in einem Chip unter gewissen Umständen topologisch nicht vorgesehen werden, um auf diese Weise Platz zu sparen).

## Beispiel (Programmierung ohne Nanospeicher)

A <sup>n</sup> [hex]	D <sup>n</sup> [hex]
0	28
1	A3
2	0A
3	0A
4	F0
5	28
6	A3
7	E5
8	26
9	A3
A	0A
B	0A
C	E5
D	1A
E	1B
F	28
10	1A
11	1A
12	26
13	A3



Ein einfaches Beispiel soll dies verdeutlichen. Gegeben ist die ROM-Tabelle links oben einer rein horizontalen Konfiguration. In den wiedergegebenen 20 Zeilen sind jedoch nur 8 unterschiedliche Mikrobefehle enthalten, die in einem *Sekundärspeicher* abgelegt werden können. Die rechte Spalte der Tabelle zeigt, dass dabei 8 Datenausgänge gebraucht werden.

## Beispiel (Programmierung mit Nanospeicher)

The diagram illustrates the reduction of memory space. On the left, a large table labeled "Sekundär-speicher" shows 20 rows of address A<sup>n</sup> [hex] and data D<sup>n</sup> [hex]. In the center, a smaller table labeled "Hauptspeicher" shows 8 rows of address A<sup>n</sup> [hex] and data D<sup>n</sup> [hex]. A double-headed green arrow connects the top 8 rows of the "Sekundär-speicher" table to the "Hauptspeicher" table. To the right, a third table shows rows 8 through 13 removed from the original "Sekundär-speicher" table, resulting in 12 rows of address A<sup>n</sup> [hex] and data D<sup>n</sup> [hex].

A <sup>n</sup> [hex]	D <sup>n</sup> [hex]
0	28
1	A3
2	0A
3	0A
4	F0
5	28
6	A3
7	E5
8	26
9	A3
A	0A
B	0A
C	E5
D	1A
E	1B
F	28
10	1A
11	1A
12	26
13	A3

A <sup>n</sup> [hex]	D <sup>n</sup> [hex]
0	28
1	A3
2	0A
3	F0
4	E5
5	26
6	1A
7	1B

A <sup>n</sup> [hex]	D <sup>n</sup> [hex]
0	0
1	1
2	2
3	2
4	3
5	0
6	1
7	4
8	5
9	1
A	2
B	2
C	4
D	6
E	7
F	0
10	6
11	6
12	5
13	1



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

Der Hauptspeicher (rechts in Tab. 4.11) enthält dann nur noch die Nummern der einzelnen Mikrobefehle (= Adressen des Sekundärspeichers). Die Spalten können so von 8 auf 3 reduziert werden.

Bleibt noch zu klären, wovon es abhängt, ob ein Sekundärspeicher zu integrieren ist oder nicht. Zunächst ist direkt erkennbar, dass der Sekundärspeicher zusätzliche Laufzeiteffekte mit sich bringt. Das Signal muss zweimal während eines Taktzyklus einen Decoder und eine Speicher-Matrix durchlaufen. Zweitens ist durch die Integration eines Sekundärspeichers ein zusätzlicher Aufwand durch einen weiteren Decoder in Kauf zu nehmen. Das bedeutet, die Reduktion von der linken Tabelle auf die beiden anderen Tabellen rechts davon zeigt nicht exakt den Nettogewinn, aber die Größenordnung.

Da es nur 8 unterschiedliche Mikrobefehle gibt ( $\gamma = 8$ ), kann  $\beta$  auf 3 reduziert werden. Geht man nun davon aus, dass die nicht programmierten Zeilen im Mikroprogrammspeicher nicht physikalisch umgesetzt werden müssen, ist die benötigte Kapazität:

13 hex + 1 hex (entspricht 20 Zeilen),

und die nicht benötigte Kapazität (weil dieser Platz nicht programmiert wird):

1F hex - 13 hex (entspricht  $2^5 - 20 = 12$  Zeilen).

Dann gilt nach der anfangs angegebenen Gleichung für  $n_a/n_b \sim 1,3$ .

Man spart im vorliegenden Fall also:

$$n_a - n_b = (2^5 * 8 - 12 * 8) - (2^5 * 3 - 12 * 3 + 2^3 * 8) = 160 - 124 = 36$$

Speicherzellen.

# Schlussfolgerung

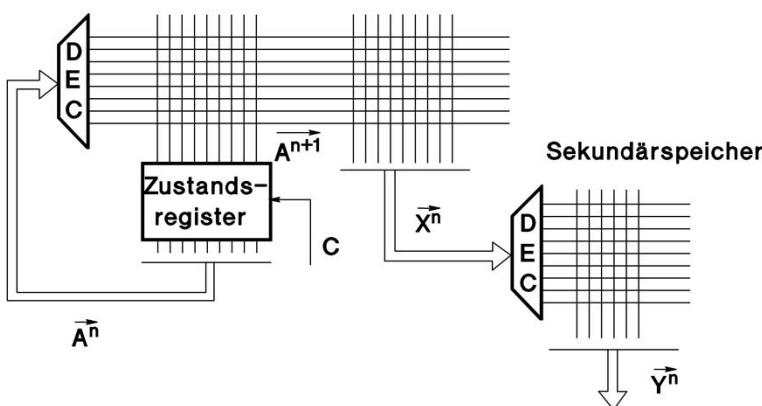
## *Vorteil*

enorme Platzersparnis

## *Nachteil*

- Durchlaufzeiterhöhung (**Speicherzugriffszeit erhöht sich**)
- zusätzlicher Speicher zu integrieren

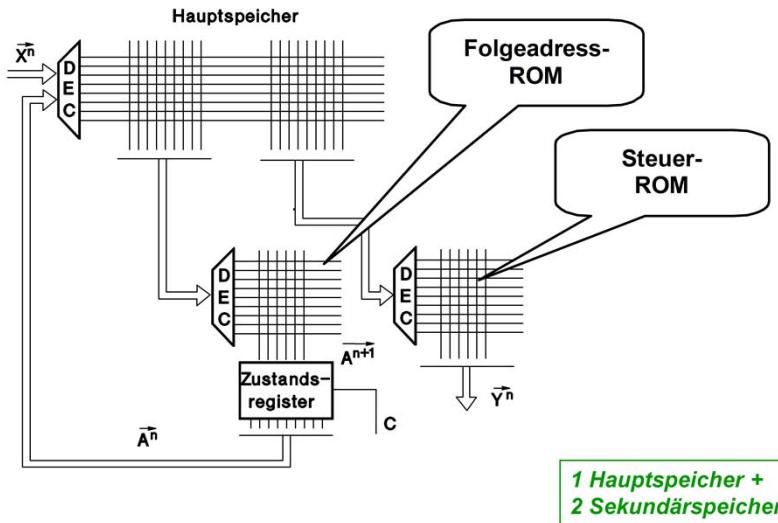
## Variationen



1 Hauptspeicher +  
1 Sekundärspeicher

Das Prinzip des Nanospeichers lässt sich natürlich in verschiedensten Variationen anwenden. Sind in einem Mikrogrammwort beispielsweise die Steuerdaten  $\bar{Y}^n$  häufig gleich, die Zustandsfolgen dagegen stark unterschiedlich, wird sich ein Entwurf nach Bild oben als vorteilhaft erweisen. Umgekehrt gilt natürlich das entsprechende.

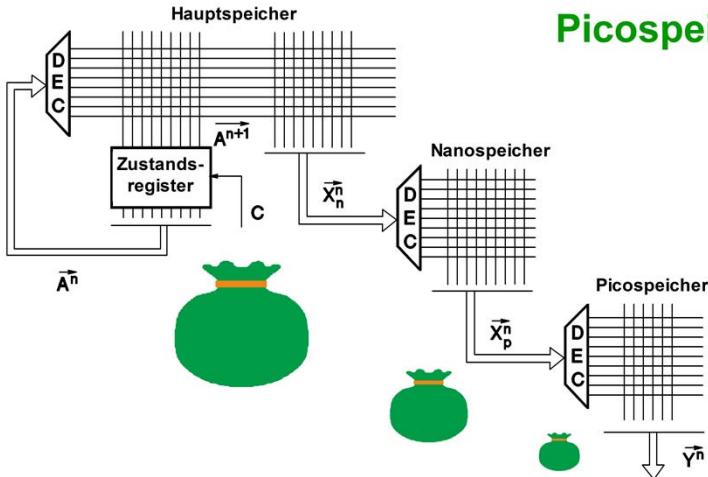
# Variationen



Sind beide Teile des Mikroprogrammwortes häufig gleich, aber in unterschiedlichen Mikroprogrammworten, könnte sich der Entwurf nach Bild oben als vorteilhaft erweisen.

Um die Flächen noch weiter zu optimieren, könnte man noch einen Schritt weiter gehen:  
Einführung des Picoprogrammspeichers

## Picospeicher



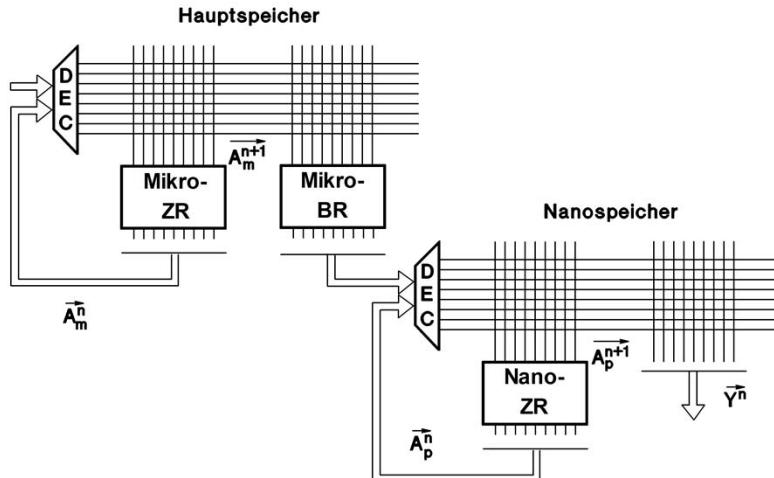
1 Hauptspeicher +  
2 Sekundärspeicher  
(1 Nano- + 1 Pico-Speicher)

### (b) Picoprogrammspeicher

Die Bezeichnung Nanoprogrammspeicher hat man für einen Speicher gewählt, der, hierarchisch gesehen, in der Ebene unter der des Mikroprogrammspeichers liegt. Konsequenterweise führt dies zum Picoprogrammspeicher. Dies klingt zwar faszinierend, doch bei näherer Betrachtung einer solchen Architektur (siehe Bild) wird klar, dass wegen der damit eingehandelten Laufzeiten (drei Decoder plus Speichermatrixen müssen durchlaufen werden) ein derartiges System heute kaum mehr infrage kommt. In der Vergangenheit hat es aber tatsächlich Rechner auf VAX-Basis gegeben, die Picoprogrammspeicher-Architekturen integriert hatten.

In der Literatur taucht zwar zwischendurch auch das Wort Picoprogrammierung auf, doch hat sie keine Bedeutung. Das Gleiche gilt für /Bode80/. Dort wird sogar das Wort Picoprogramm anstelle von Nanoprogramm eingeführt, was keine glückliche Definition sein kann.

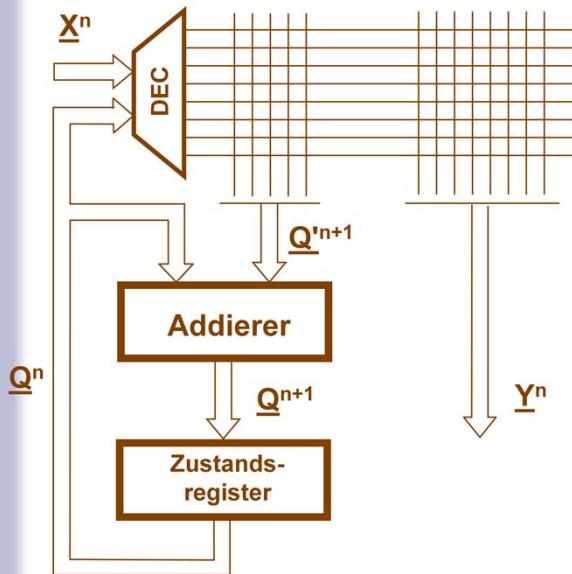
## Mikroprogrammsteuerwerk mit unterlagertem Nanoprogrammsteuerwerk



### (c) Nanoprogrammierung

Im Unterschied durch eine ausschließliche Erweiterung durch einen Nanospiecher wird bei der unabhängigen Nanoprogrammierung ein eigenständiges Nanoprogrammschaltwerk zugrunde gelegt, das dem Mikroprogrammschaltwerk untergeordnet wird. Damit liegen dann 2 Zustandsregister vor und man benötigt zwischen Mikro- und Nanoprogrammschaltwerk noch ein weiteres Register, damit das Adressdatum am Decoder des Nanoprogrammschaltwerkes konstant für den Prozessablauf im Nanoprogrammschaltwerk ansteht. Dies kann Mikroprogrammbefehlsregister genannt werden, man könnte es aber auch als Mikroprogramm-Steuerwerksregister bezeichnen.

# Optimierung durch Relativadressierung



Einsparung von  
ROM-Zellen durch  
Begrenzung der  
Sprungweite

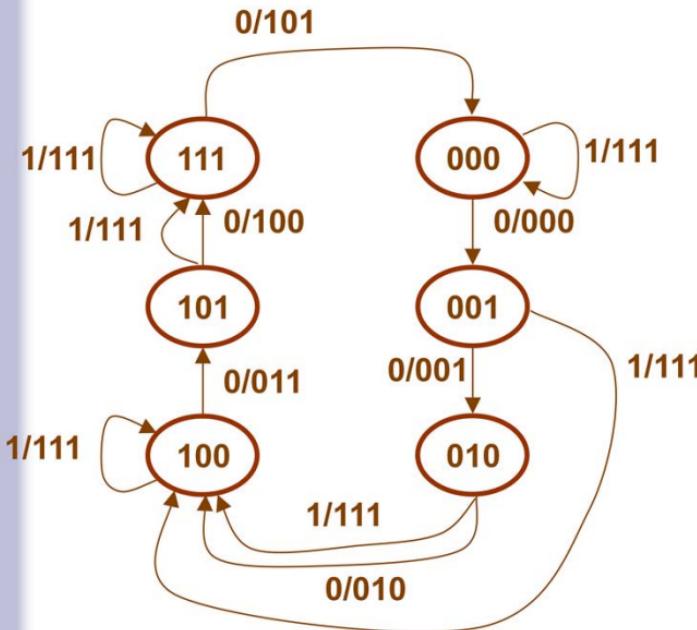
Es gilt:  
 $Q^{n+1} = Q^n + Q^{n+1}$

## (d) Relative Adressierung

Ein Mikroprogrammsteuerwerk, wie es bisher vorgestellt wurde, generiert absolute Adressen. Alle Stellen des Adresswortes werden direkt vollständig vom Mikroprogrammspeicher beziehungsweise durch den Eingangsvektor  $X_n$  des Decoders erzeugt. Wird nun festgelegt, dass der Wert der Folgeadresse  $A^{n+1}$  des Mikroprogrammwortes einen bestimmten Wert nicht unter- beziehungsweise überschreiten darf, kann man sich darauf beschränken, den Abstand zum nächsten  $A^{n+1}$  vorzugeben und nicht mehr  $A^{n+1}$  direkt.  $A^{n+1}$  muss dann von einem Addierwerk generiert werden. Das Bild zeigt das Schaltungsprinzip. Der Addierer generiert die absolute Adresse  $A^{n+1}$  über die relative Adresse  $A^{n+1}$  sowie die momentane Adresse  $A^n$  nach dem Formalismus:

$$A^{n+1} = A^{n+1} + A^n.$$

## Zustandsgraf

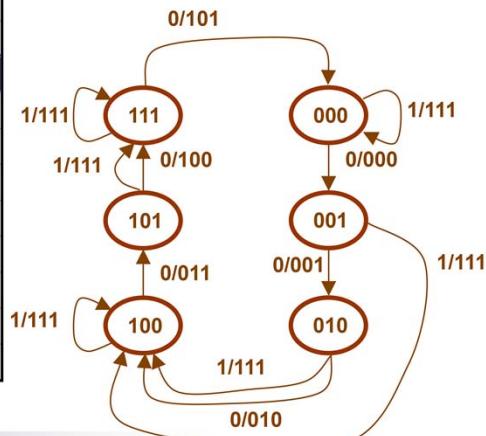


$$\underline{Q} = \{Q_2, Q_1, Q_0\}$$
$$\underline{Y} = \{Y_2, Y_1, Y_0\}$$

Das Bild zeigt ein Beispiel. Die Abstände zwischen den Übergängen sind maximal  $\{11\}_B$ . Das bedeutet, die drei Spalten der Zustandsadressfolgematrix können auf zwei Spalten reduziert werden.

		Z <sup>n</sup>			Z <sup>n+1</sup>				Z <sup>n</sup>			
	X	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q' <sub>1</sub>	Q' <sub>0</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>
0	0	0	0	0	0	0	1	0	1	0	0	0
1	0	0	0	1	0	1	0	0	1	0	0	1
2	0	0	1	0	1	0	0	1	0	0	1	0
3	0	0	1	1	x	x	x	x	x	x	x	x
4	0	1	0	0	1	0	1	0	1	0	1	1
5	0	1	0	1	1	1	1	1	0	1	0	0
6	0	1	1	0	x	x	x	x	x	x	x	x
7	0	1	1	1	0	0	0	0	1	1	0	1
8	1	0	0	0	0	0	0	0	0	1	1	1
9	1	0	0	1	1	0	0	1	1	1	1	1
A	1	0	1	0	1	0	0	1	0	1	1	1
B	1	0	1	1	x	x	x	x	x	x	x	x
C	1	1	0	0	1	0	0	0	0	1	1	1
D	1	1	0	1	1	1	1	1	0	1	1	1
E	1	1	1	0	x	x	x	x	x	x	x	x
F	1	1	1	1	1	1	1	0	0	1	1	1

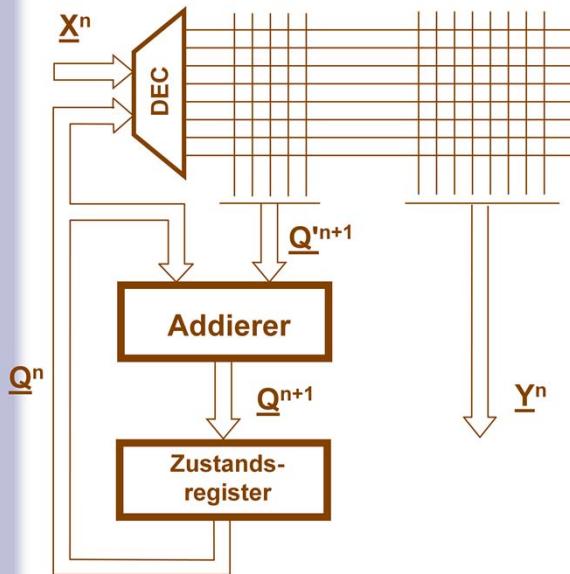
Änderung von nur 2 Stellen



Dargestellt ist die zugehörige Übergangstabelle. Eine Zeile ist darin hervorgehoben. Während die Addition nach der obigen Gleichung in allen anderen Zeilen direkt zum Ergebnis führt, kommt es in der Zeile 7 zu einem Überlauf des Addierwerkes, was berücksichtigt werden muss.

Zu beachten ist bei diesem Verfahren, wie beim Nanoprogrammspeicher, dass man sich mit der Integration eines Addierers zusätzliche Laufzeiten einhandelt.

# Optimierung durch Relativadressierung



Einsparung von  
ROM-Zellen durch  
Begrenzung der  
Sprungweite

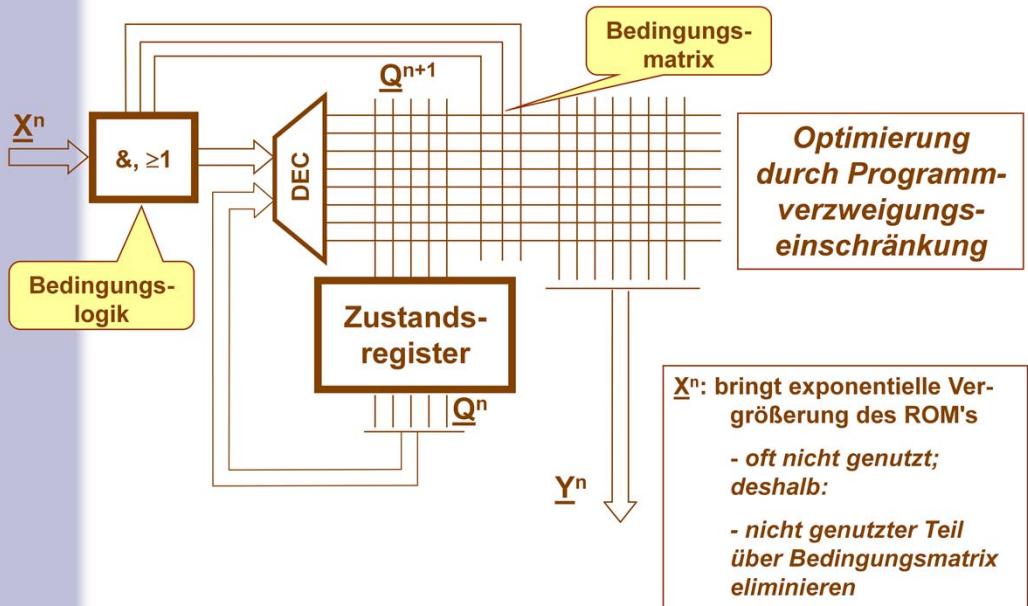
Es gilt:  
 $Q^{n+1} = Q^n + Q^{n+1}$

## (d) Relative Adressierung

Ein Mikroprogrammsteuerwerk, wie es bisher vorgestellt wurde, generiert absolute Adressen. Alle Stellen des Adresswortes werden direkt vollständig vom Mikroprogrammspeicher beziehungsweise durch den Eingangsvektor  $X_n$  des Decoders erzeugt. Wird nun festgelegt, dass der Wert der Folgeadresse  $A^{n+1}$  des Mikroprogrammwortes einen bestimmten Wert nicht unter- beziehungsweise überschreiten darf, kann man sich darauf beschränken, den Abstand zum nächsten  $A^{n+1}$  vorzugeben und nicht mehr  $A^{n+1}$  direkt.  $A^{n+1}$  muss dann von einem Addierwerk generiert werden. Das Bild zeigt das Schaltungsprinzip. Der Addierer generiert die absolute Adresse  $A^{n+1}$  über die relative Adresse  $A^{n+1}$  sowie die momentane Adresse  $A^n$  nach dem Formalismus:

$$A^{n+1} = A^{n+1} + A^n.$$

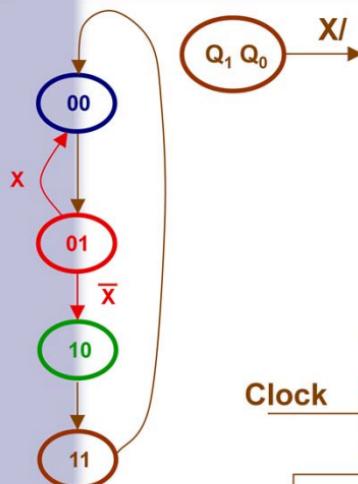
## Bedingte Adressierung



### (e) Bedingte Adressierung

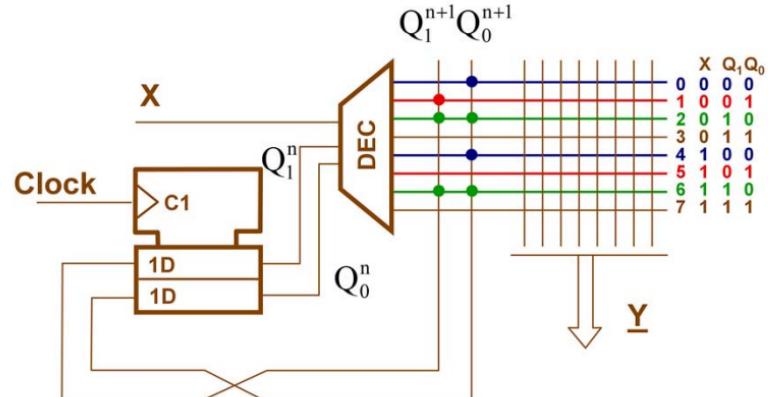
Die bedingte Adressierung ist ein sehr interessantes Verfahren. Es kann zu einer drastischen Speicherplatzoptimierung bei der Mikroprogrammierung führen, ist aber nicht ganz so trivial, wie es im ersten Moment ausschauen mag und hat klare Konsequenzen für das Schaltungsdesign. Mit dem Verfahren kann auch der Unterschied zwischen der synchronen und asynchronen Rückkopplung eines Mealy-Automaten erklärt und die dahinterliegende Problematik eindrucksvoll vor Augen geführt werden.

*angenommener Zustandsgraf:*



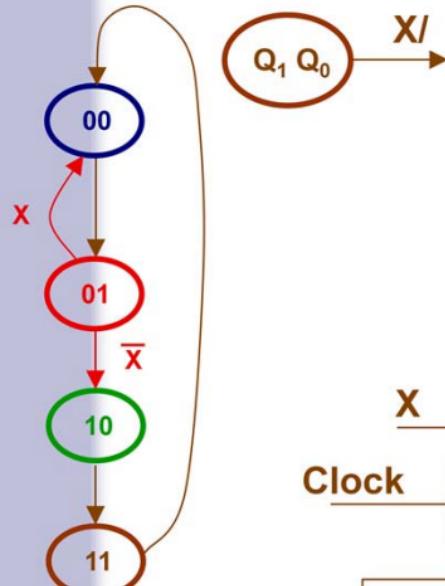
**Beispiel**

*Lösung ohne Optimierung:*

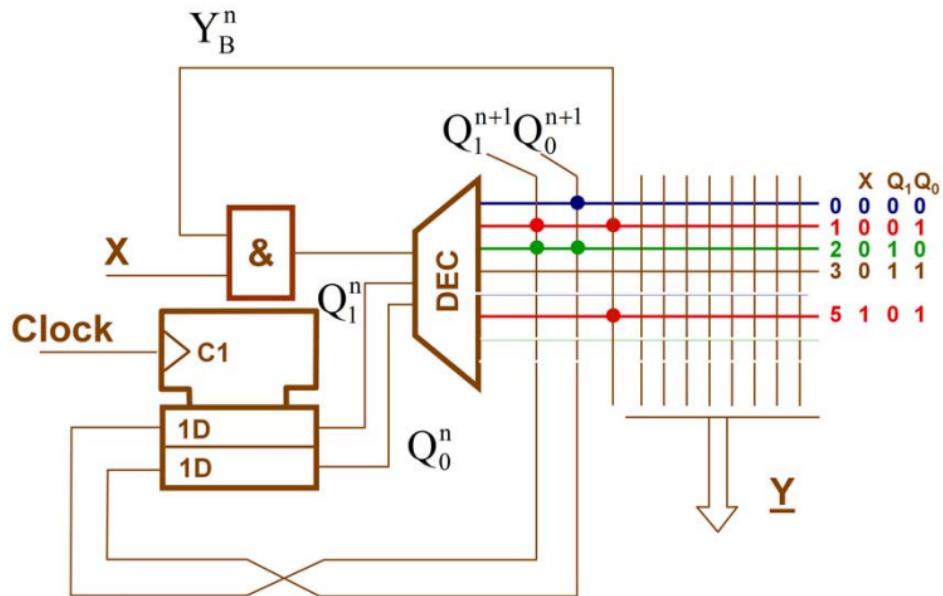


Ausgegangen werden soll von einem einfach verständlichen Beispiel. Gegeben sei ein Zustandsgraf nach Bild oben. Das System wechselt zwischen 4 Zuständen, wobei eine einzige Fallunterscheidung integriert ist. Im Zustand  $Z_1 = Q_1 Q_0 = \{01\}$  entscheidet die Eingangsgröße  $X$ , ob das System zum Zustand  $Z_0 = Q_1 Q_0 = \{00\}$  zurück oder in den Zustand  $Z_2 = Q_1 Q_0 = \{10\}$  übergeht.

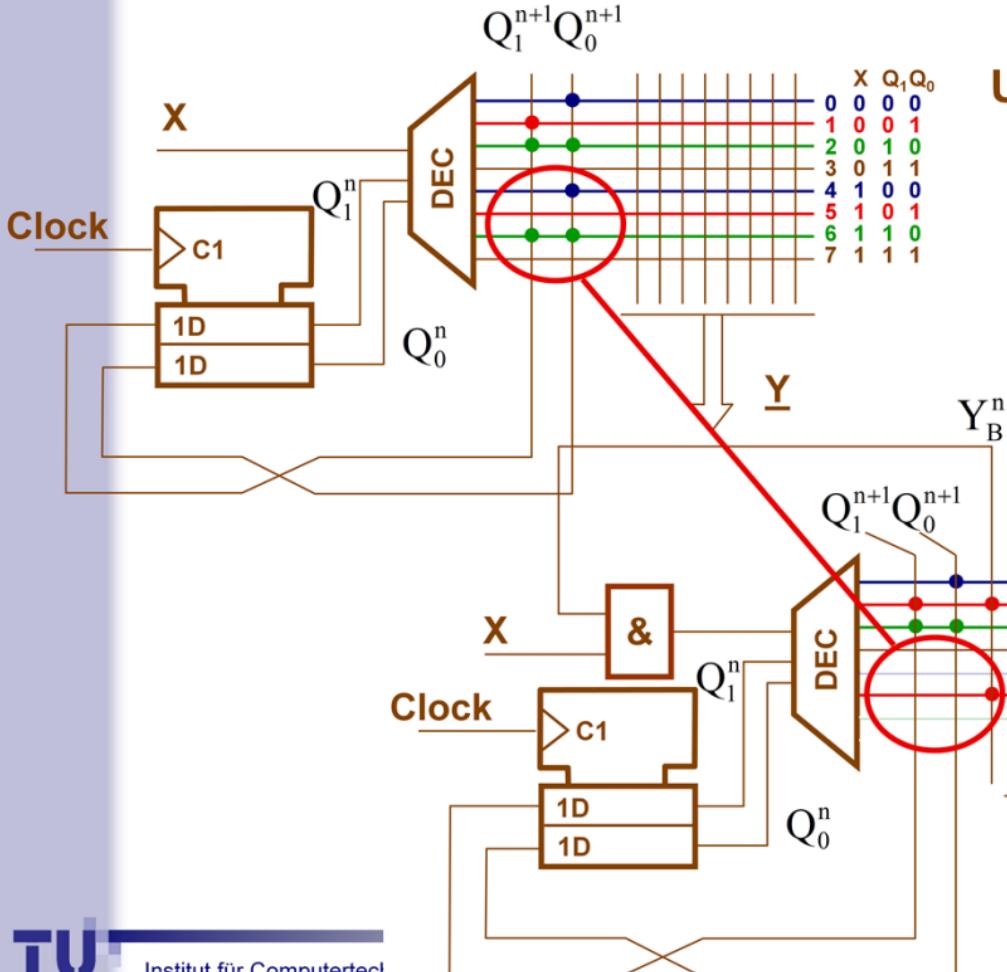
angenommener Zustandsgraf:



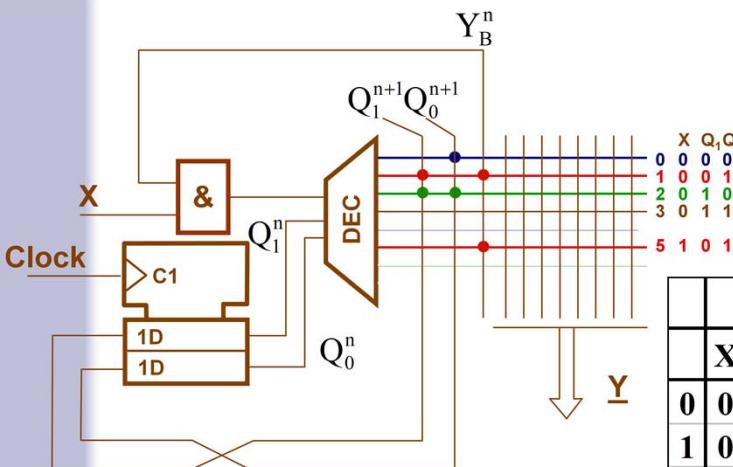
Lösung mit Optimierung



## Unterschied



**Tabelle**



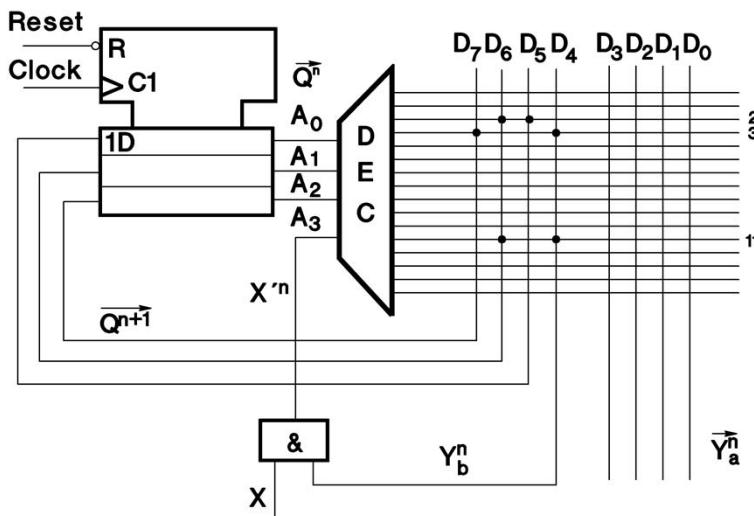
	Z <sup>n</sup>		Z <sup>n+1</sup>		Z <sup>n</sup>	
	X	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Y
0	0	0	0	0	1	0
1	0	0	1	1	0	1
2	0	1	0	1	1	0
3	0	1	1	0	0	0
4	1	0	0	0	1	0
5	1	0	1	0	0	1
6	1	1	0	1	1	0
7	1	1	1	0	0	0

Aus den Bildern ist zu erkennen, dass die Schaltung redundante Informationen enthält (manche Zeilen weisen die gleiche Programmierung auf), es stellt sich nun die Frage, wie diese Redundanzen eliminiert werden können. Die Zeilen 0 und 4 ( $Z_{0,4} = Q_1 Q_0 = \{00\}$ ), 2 und 6 ( $Z_{2,6} = Q_1 Q_0 = \{10\}$ ) sowie 3 und 7 ( $Z_{3,7} = Q_1 Q_0 = \{11\}$ ) müssen identisch programmiert werden, da jeweils beide Eingangswerte, X und  $\neg X$ , vorkommen können. Die Antwort liegt nun auf der Hand. Der obere Teil des Mikroprogrammspeichers darf nur aufgerufen werden, wenn das System sich im Zustand  $Z_1 = Q_1 Q_0 = \{01\}$  befindet, denn nur dann besteht die Möglichkeit, dass auch der Wert X beziehungsweise  $\neg X$  auftreten kann, und der obere Teil des Mikroprogrammspeichers angesprochen wird. Dann aber werden die Zeilen 4, 6 und 7 überflüssig und können sogar physisch weggelassen werden. Dies bringt in einem diskreten Aufbau im Allgemeinen nichts, doch wird die Schaltung in einen Chip integriert, muss nur der Decoder entsprechend realisiert werden, was bei komplexeren Beispielen schnell zu einer drastischen Reduktion der realen, physikalisch vorliegenden Mikroprogrammzeilen führen kann.

Zusatzbemerkung: Es muss also verhindert werden, dass die redundanten Zeilen 4, 6 und 7 angesprungen werden können. Bild oben zeigt eine einfache Lösung. Es wird eine zusätzliche Spalte ( $Y_B$ ) im Mikroprogrammspeicher eingeführt, die so programmiert wird, dass  $X'$  (Eingang des ROM-Decoders) nur dann 1 werden kann, wenn sich das System im Zustand  $Z_1 = \{01\}$  befindet. Die Zeilen 4, 6 und 7 in Tab. 4.13 bleiben dann wirkungslos.

Wichtig zu verstehen ist nun, dass diese Schaltung nicht mit der Schaltung, die einleitend zu diesem Thema vorgestellt wurde, verwechselt werden darf. Dort ist ein Register eingezeichnet, während hier in einem Rückkopplungszweig ein Gatter integriert ist. Das zusätzliche Makroregister in der damaligen Darstellung hat somit nichts mit dem in Bild oben integrierten Gatter zu tun. Die Rückkopplung über ein Register liefert ein synchrones Verhalten, die Rückkopplung über ein Gatter ein asynchrones Verhalten. Dementsprechend stellt die Schaltung nach Bild oben übergeordnet ein synchrones Schaltwerk dar, doch ist ein asynchrones Schaltwerk diesem synchronen unterlagert. In Kapitel 3 wurde dies schon behandelt. Jedes FF enthält als Basiseinheit ein asynchrones RS-FF, was nur laufzeitmäßig berücksichtigt werden muss. Dies soll näher im Folgenden betrachtet werden.

## Momentaufnahme eines Logic-Analyzers



Zugrundegelegt ist eine Schaltung nach Bild oben.

## Momentaufnahme eines Logic-Analyzers

eines µSchaltwerkes mit Bedingungsmatrix ohne dazwischengeschaltetes Register

= asynchroner Automat

Spikes

C	X	Z <sup>n</sup>			Z <sup>n+1</sup>			Y
		Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	
..	..	..	..	..	..	..	..	..
1	0	0	1	0	0	1	1	0
0	0	0	1	0	0	1	1	0
0	0	0	1	1	0	1	1	0
0	0	0	1	1	0	1	1	0
0	0	0	1	1	1	0	0	1
0	0	0	1	1	1	0	0	1
0	0	0	1	1	1	0	0	1
0	1	0	1	1	1	0	0	1
0	1	0	1	1	1	0	0	1
0	1	0	1	1	1	0	0	1
0	1	0	1	1	0	0	0	1
0	1	0	1	1	0	1	0	1
0	1	0	1	1	0	1	0	1
..	..	..	..	..	..	..	..	..

In Bild oben ist das Messergebnis eines Logikanalysators dargestellt, in dem ein Clock-Wechsel (Spalte C:= Clock) von 1 auf 0 festgehalten ist.

Die erste 0 in Spalte C veranlasst, dass der FF-Ausgang Q<sub>0</sub> von 0 auf 1 wechselt. Das bewirkt wiederum eine Änderung des Vektors Q<sup>n+1</sup>, jedoch nicht in einem Schritt, was physikalisch aufgrund der unterschiedlichen Gattereigenschaften kaum möglich ist. Die Reihenfolge der Ereignisse ist zufällig. Im vorliegenden Fall kippt erst das FF<sub>0</sub> (= Q<sub>0</sub>), dann das FF<sub>2</sub> und zuletzt das FF<sub>1</sub>. Der Wert Q<sup>n</sup> = {011} veranlasst aber nicht nur, dass sich der Wert Q<sup>n+1</sup> = {100} einstellt, sondern auch, dass die Leitung Y von 0 nach 1 wechselt. Damit verändert sie auch die Größe X<sup>n</sup>, was eine nochmalige Änderung des Ausgangsvektors Q<sup>n+1</sup> zur Folge hat. Diesmal stellt sich der Wert Q<sup>n+1</sup> = {010} ein.

An diesem praktischen Beispiel ist sehr schön zu erkennen, dass erstens der Wechsel in einzelnen Schritten erfolgt, und zweitens, dass ein Ablauf in der Schaltung nach Bild oben sich in zwei Zyklen abspielt, sobald sich der Wert von X<sup>n</sup> ändert. Dabei stellen sich unterschiedliche Adresswerte des Decoders ein, was nicht nur für die Ausgangswerte von Y<sup>n</sup> zu berücksichtigen ist:

- ❖ Übergänge erfolgen laufzeitverzögert abhängig von einzelnen Transistor- und Leitungscharakteristika
- ❖ deutlich erkennbar: EINSCHWINGVERHALTEN

Daraus folgen Fragen für die Integration einer Bedingungsmatrix:

1. Wie muss die Schaltungsrealisierung erfolgen?
2. Können ausreichend Speicherstellen eingespart werden, damit sich die damit eingekauften Nachteile rentieren?
3. Ist die Forderung erfüllt: Reset-Vorgang:  $\{Q_n, Q_{n-1}, \dots, Q_0\} = \{0, 0, \dots, 0\} \Rightarrow X_0 = 0 \Rightarrow$  erste angesprungene Zeile = Zeile 0

Wichtig:

1. Analyse nicht belegter Zeilen von  $Y_B$  (Bedingungsmatrix)
2. Analyse von Auswirkungen nicht definierter Zwischenzustände von  $Y_A$  (Ausggsm.)

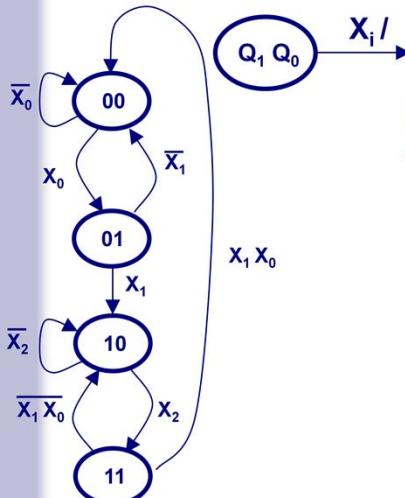
**Die wechselnden Ausgangswerte dürfen keine Wirkung im Operationswerk oder außerhalb des Mikroprozessors haben, solange die Werte nicht stabil sind, also bis beide Zyklen vollständig abgelaufen sind und das System sich wiederum in einem stabilen Zustand befindet.**

Zu beachten ist auch, dass alle existierenden Zeilen, die "zufällig" angesprungen werden, auch entsprechend programmiert sein müssen. Eine falsche Programmierung der  $Y_B$ -Leitung kann zu einer instabilen Oszillation führen, wenn nämlich im asynchronen Zyklus laufend eine Leitung angesprungen wird, wobei einmal der Bedingungsknoten auf 1 steht, ein anderes Mal auf 0.

Noch ein Aspekt darf nicht übersehen werden. In der Schaltung ist kein Reset eingezeichnet. Schaltungen, die mit einem zufälligen Anfangszustand starten dürfen, gibt es in der Praxis in der Tat (Lauflichter oder ähnliche einfache Steuerungen). Im vorliegenden Fall, bei einem Mealy-Automaten auf ROM-Basis Schaltung kann jedoch die Schaltung nicht funktionieren, da beim Einschalten das Zustandsregister zufällig auf eine nichtprogrammierte Zeile verweisen könnte. Es muss also gewährleistet sein, dass zum Augenblick des Reset  $Q^n$ ,  $X$  und eventuell sogar  $Y$  sich in einem definierten Anfangszustand befindet.

## Komplexes Beispiel

angenommener Zustandsgraf:



Möglichkeit 1:  
ohne Optimierung

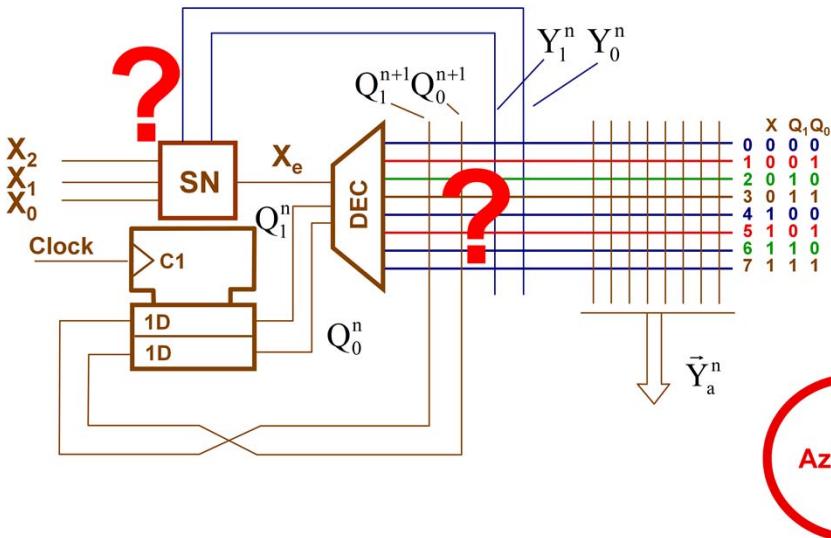
AzÜ wnv

	Z <sup>n</sup>				Z <sup>n+1</sup>		
	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>0</sub>
0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0
2	0	0	0	1	0	1	0
3	0	0	0	1	1	1	0
4	0	0	1	0	0	0	1
5	0	..					
6	..						

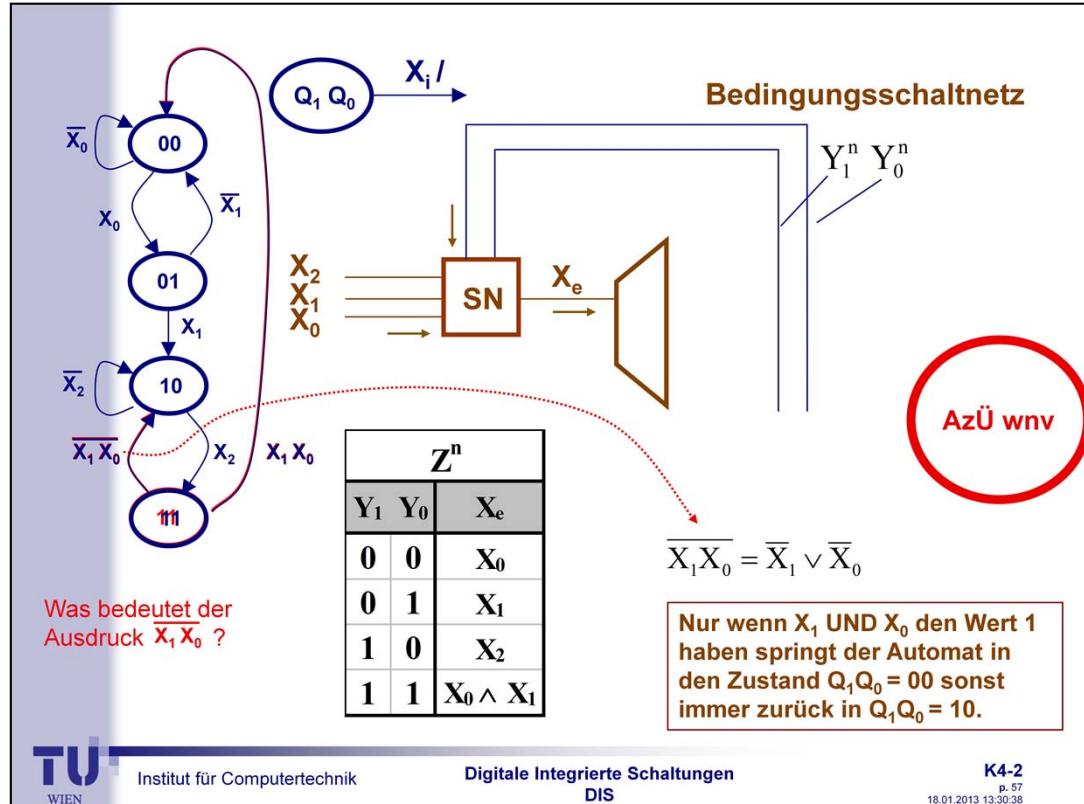
Die großen Einsparungsmöglichkeiten dieses Verfahrens seien an einem weiteren, etwas komplexeren Beispiel vertieft. Zugrunde gelegt sei der Zustandsgraf von Bild oben. Aus den vorhergehenden Überlegungen kann hierzu die Blockschaltung der folgenden Folie entworfen werden, denn es liegen drei Eingangsvariablen vor, die auch in der Kombination von beziehungsweise auftreten. Das bedeutet, anstatt eine Übergangstabelle ohne Optimierung über eine Bedingungsmatrix plus zugehöriger Logik nach Tab. oben zu entwickeln, berücksichtigt man eine Zuordnung nach Tabelle der übernächsten Folie.

Zusatzbemerkung: Beachten Sie, dass es bei diesem Beispiel zwar 3 Eingänge gibt, aber jeweils von einem Knoten nur 2 Kanten abgehen. Daher genügt ein Eingang, das die Funktionen der 3 Eingänge übernimmt.

## Optimierung durch Bedingungsschaltnetz



Daraus lässt sich ablesen, dass eine Bedingungsmatrix mit zwei Spalten notwendig ist. Zu entwerfen sind dabei die Bedingungslogik sowie die Mikroprogrammierung des ROMs. Dass diese Matrix im ROM entscheidend kleiner ausfällt als die Matrix nach der Tabelle der letzten Folie ist offensichtlich, da die Anzahl der Spalten von  $Z^n$  der Übergangstabelle geringer ist, was sich ja exponentiell in der Zeilenzahl niederschlägt (hier 3 anstatt 5 Eingangsvariablen: 8 anstatt 32 Zeilen).



Wie geht man nun bei der Berechnung vor? Im Gegensatz zur Optimierung über den Sekundärspeicher oder die Relativadressierung usw. ist es hier nicht notwendig, von dem nichtoptimierten System auszugehen. Es gilt also zunächst, das Schaltnetz (SN), also die Bedingungslogik zu entwickeln, was im Bild skizziert ist.

Nach der Tabelle oben können sich bezüglich des Eingangsvektors  $\underline{X}$  vier verschiedene Variationen einstellen:

- für  $\underline{Y}_b = \{00\}$  wird  $X_1$  durchgeschaltet,
- für  $\underline{Y}_b = \{01\}$  wird  $X_2$  durchgeschaltet,
- für  $\underline{Y}_b = \{10\}$  wird  $X_3$  durchgeschaltet und
- für  $\underline{Y}_b = \{11\}$  werden  $X_2$  und  $X_1$  durchgeschaltet,

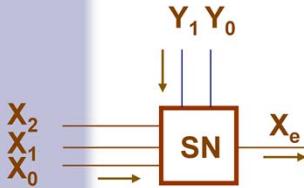
was zu

$$X_e = f(\underline{X}, \underline{Y}_b)$$

führt. Die zugehörige Wertetabelle ist nun Zeile für Zeile zu entwickeln (Tabelle auf der nächsten Folie).

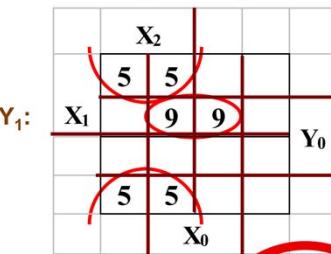
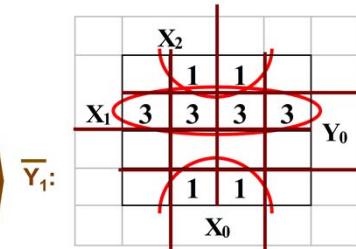
$\neg(X_1 X_0) = (\neg X_1) \vee (\neg X_0) :=$  wenn  $X_1$  oder  $X_0$  den Wert 0 einnimmt, ist  $X_e=0$ ; nur bei  $X_1 X_0=11$  springt der Automat in den Zustand 00.

## Bedingungsschaltnetz



Z <sup>n</sup>		X <sub>e</sub>
Y <sub>1</sub>	Y <sub>0</sub>	X <sub>e</sub>
0	0	X <sub>0</sub>
0	1	X <sub>1</sub>
1	0	X <sub>2</sub>
1	1	X <sub>0</sub> $\wedge$ X <sub>1</sub>
..	..	..

Z <sup>n</sup>					
X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	Y <sub>1</sub>	Y <sub>0</sub>	X <sub>e</sub>
*	*	0	0	0	0
*	*	1	0	0	1
*	0	*	0	1	0
*	1	*	0	1	1
0	*	*	1	0	0
1	*	*	1	0	1
*	0	0	1	1	0
*	0	1	1	1	0
*	1	0	1	1	0
*	1	1	1	1	1
..	..	..	..	..	0



AzÜ wnv

K4-2

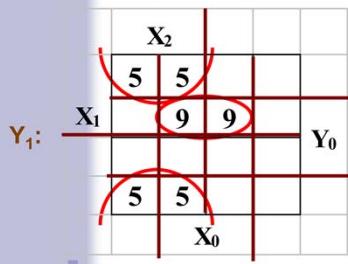
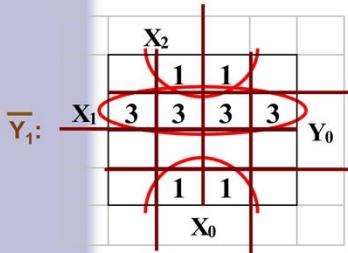
p. 58

18.07.2013 13:30:38

Bemerkung: Jeder Stern bedeutet die Verdopplung einer Zeile (die Werte sind irrelevant, das Ergebnis wird durch ihre Werte nicht beeinflusst).

Zunächst spielen die Eingänge X<sub>2</sub> und X<sub>1</sub> keine Rolle, Y<sub>1</sub> und Y<sub>0</sub> haben jeweils den Wert 0, und der Wert X<sub>e</sub> richtet sich direkt nach dem Wert von X<sub>0</sub> usw. Zu beachten ist, dass die Sternchen in den ersten beiden Spalten der Tabelle oben nur eine vereinfachte Schreibweise bedeuten. Das heißt, dass im Grunde für jedes Sternchen die Anzahl der jeweiligen Zeile zu verdoppeln ist (also für das Sternchen in der Zeile jeweils eine 0 und in der folgenden Zeile eine 1 zu schreiben wäre. Für eine Zeile mit zwei Sternchen wären also vier Zeilen zu schreiben)

## Bedingungsschaltnetz



$$X'' = (X_1 Y_0 \vee X_0 \bar{Y}_0) \bar{Y}_1 \\ = X_1 \bar{Y}_1 Y_0 \vee X_0 \bar{Y}_1 \bar{Y}_0$$

$$X_e = X'_e \vee X''_e$$

$$= X_1 \bar{Y}_1 Y_0 \vee X_0 \bar{Y}_1 \bar{Y}_0 \vee X_2 Y_1 \bar{Y}_0 \vee X_1 X_0 Y_1 \bar{Y}_0$$

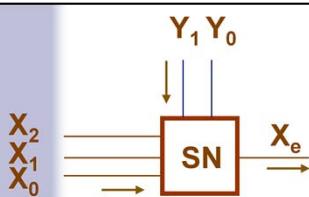
$$X'_e = (X_2 \bar{Y}_0 \vee X_1 X_0 Y_0) Y_1 \\ = X_2 Y_1 \bar{Y}_0 \vee X_1 X_0 Y_1 \bar{Y}_0$$

AzÜ wnv

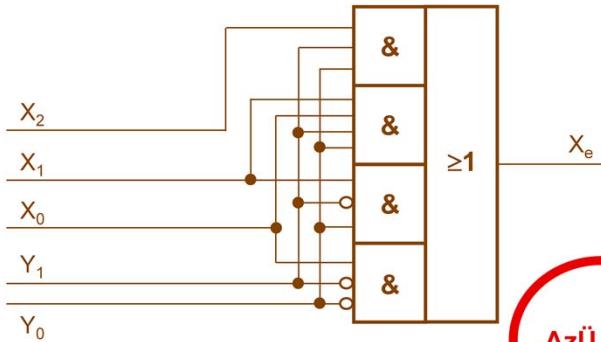
K4-2  
p. 59  
18.01.2013 13:30:38

Die Wertetabelle kann nun vereinfacht werden, hier beispielsweise über das KV-Diagramm.

## Bedingungsschaltnetz

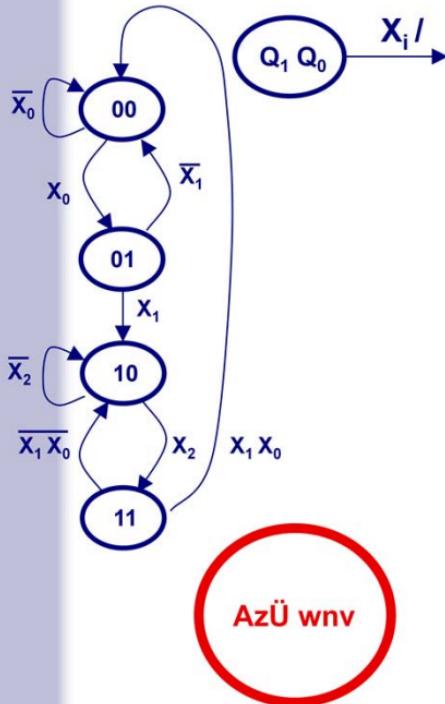


$$\begin{aligned}X_e &= X'_e \vee X''_e \\&= X_1 \bar{Y}_1 Y_0 \vee X_0 \bar{Y}_1 \bar{Y}_0 \vee_1 X_2 Y_1 \bar{Y}_0 \vee X_1 X_0 Y_1 \bar{Y}_0\end{aligned}$$



AzÜ wnv

K4-2  
p. 60  
18.01.2013 13:30:38



$Z^n$		$X_e$
$Y_1$	$Y_0$	
0	0	$X_0$
0	1	$X_1$
1	0	$X_2$
1	1	$X_0 \wedge X_1$

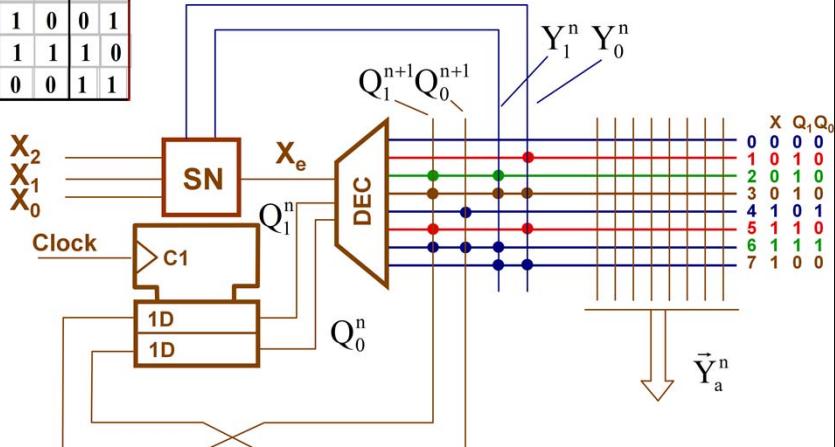
Bedingungsmatrix

	$Z^n$						$Z^{n+1}$	$Z^n$		
	$X_2$	$X_1$	$X_0$	$X_e$	$Q_1$	$Q_0$	$Q_1$	$Q_0$	$Y_1$	$Y_0$
0	*	*	0	0	0	0	0	0	0	0
1	*	0	*	0	0	1	0	0	0	1
2	0	*	*	0	1	0	1	0	1	0
3	*	0	0	0	1	1	1	0	1	1
4	*	*	1	1	0	0	0	1	0	0
5	*	1	*	1	0	1	1	0	0	1
6	1	*	*	1	1	0	1	1	1	0
8	*	1	1	1	1	1	0	0	1	1

Im nächsten Schritt erfolgt die Entwicklung der ROM-Matrix, was zunächst die Erstellung der Übergangsmatrix für das Mikroprogrammschaltwerk bedeutet.

	Z <sup>n</sup>			Z <sup>n+1</sup>			Z <sup>n</sup>			
	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	X <sub>e</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Q <sub>1</sub>	Q <sub>0</sub>	Y <sub>1</sub>	Y <sub>0</sub>
0	*	*	0	0	0	0	0	0	0	0
1	*	0	*	0	0	1	0	0	0	1
2	0	*	*	0	1	0	1	0	1	0
3	*	0	0	0	1	1	1	0	1	1
4	*	*	1	1	0	0	0	1	0	0
5	*	1	*	1	0	1	1	0	0	1
6	1	*	*	1	1	0	1	1	1	0
8	*	1	1	1	1	1	0	0	1	1

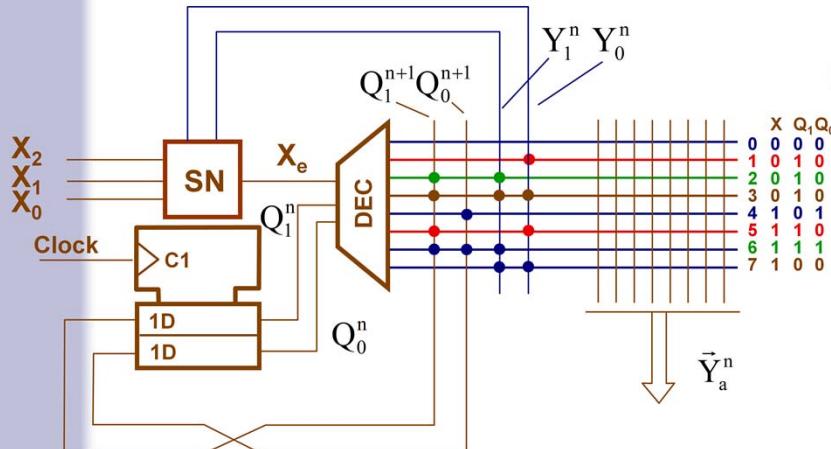
## Optimierung durch Bedingungsschaltnetz



Die Eingangsgrößen des Automaten sind  $X_e$  und  $Q^n$ , die Ausgangsgrößen sind die Zustandsfolgeadresse  $Q^{n+1}$  sowie der Bedingungsmatrixvektor  $\underline{Y}$ . Beim Aufstellen der Übergangsmatrix Tabelle oben geht man wie folgt vor: Zunächst überträgt man alle Spalten  $X_i = \{X_2, X_1, X_0, X_e\}$ . Dann hierzu die Werte von  $Q^n$  und letztendlich die von  $Q^n$  in aufsteigender Form in Unterstützung des Zustandsdiagramms. Als letztes übernimmt man die Werte für  $Q^{n+1}$ .

Aus dem Diagramm lässt sich ablesen, dass im Zustand  $Q^n = \{00\}$  die Größe  $X_0$  darüber entscheidet, ob der Wert  $Q^n = \{00\}$  oder  $Q^n = \{01\}$  angesprungen wird. Nach der Tabelle oben wird damit  $\underline{Y} = \{00\}$ . Im Zustand  $Q^n = \{01\}$  ist die entscheidende Größe  $X_1$ , usw.

Damit kann nun die endgültige Schaltung erarbeitet werden. Bleibt noch die Frage offen, wie groß nun die Einsparung im vorliegenden Fall ist.



## Einsparung

### **ohne Optimierung:**

### **5 Adresseingänge und 2 bit Wortbreite:**

$$2^5 * 2 = 64 \text{ Speicherzellen}$$

**mit Optimierung:**

$$2^3 * 4 = 32 \text{ Speicherzellen}$$

ohne Beachtung von  $\underline{Y}_a$ !



AzÜ wnv

K4-2

p. 63

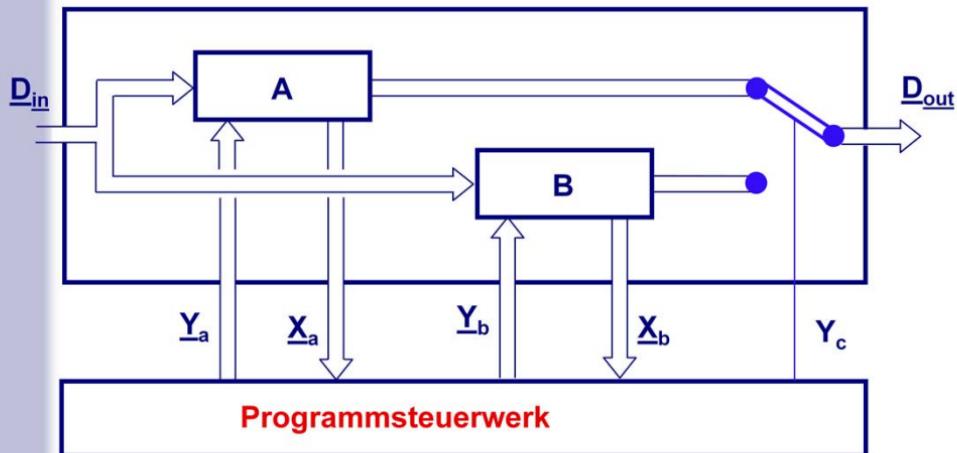
Ohne Optimierung liegt ein System mit 5 Adresseingängen für das Mikroprogrammschaltwerk vor mit einer Wortbreite des Ausgangs  $Q^{n+1}$  von 2 bit (der  $Y_a$ -Vektor sei hier nicht mitgezählt, was das Ergebnis noch zusätzlich entscheidend zu Gunsten der Optimierung ausfallen lassen würde). Es ergeben sich somit für  $Q_{n+1} \cdot 2^5 * 2 \text{ bit} = 64 \text{ bit}$  (entspricht 64 Speicherzellen). Mit der Optimierung haben wir nur 3 Eingänge, jedoch 4 Spalten, also  $2^3 * 4 \text{ bit} = 32 \text{ bit}$  (entspricht 32 Speicherzellen), was eine Einsparung von 50% bedeutet. Zu berücksichtigen ist darüberhinaus, dass der Aufwand der Bedingungslogik sowie der kleinere Decoder ebenfalls nicht berücksichtigt wurde.

Wo liegt hier die Einsparung? Normalerweise müsste man hier bei 5 Eingängen 32 Zeilen haben. Über die Bedingungsmatrix ergeben sich aber nur 8 Zeilen + 2 zusätzliche Spalten. Siehe nächste Folie. Hier geht es also darum, dass man keinen Teil eines "oberen" Teil eines Speichers einspart, sondern um die Reduzierung von Zeilen prinzipiell.

## Beispiel: 2 parallele, arithmetische Operationswerke

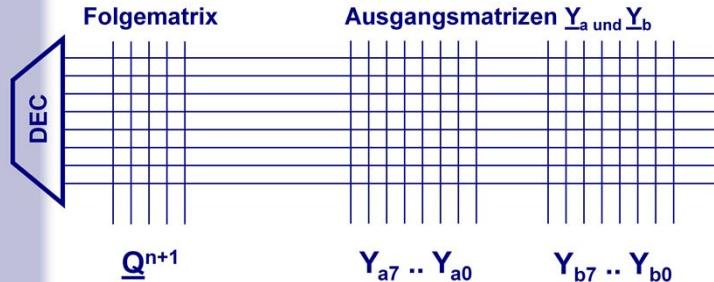
Aktionen laufen gegenläufig: entweder A oder B

Das Programmsteuerwerk ist zu entwerfen .

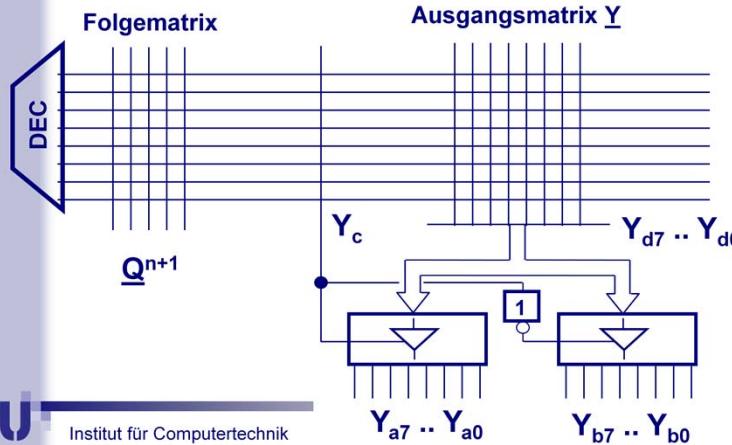


### (f) Mikrobefehlmehrfachcodierung

Erfolgen in einem Operationswerk Aktionen, wie beispielsweise in einem System nach Bild oben, gegenläufig, beinhaltet ein Programmspeicher einfacher Ausführung (siehe folgende Folie oben) zuviel Redundanz, da nur die Steuerleitungen  $Y_a$  ODER  $Y_b$  aktiviert werden müssen.

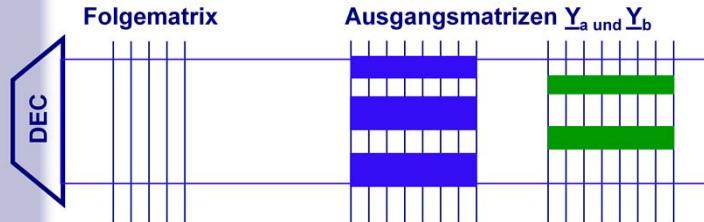


Entwurf ohne Optimierung



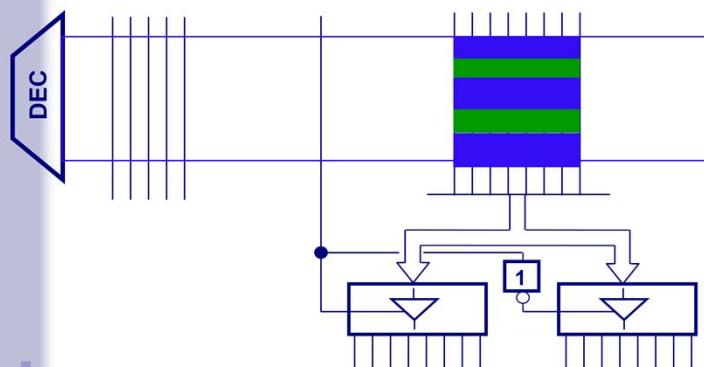
Entwurf mit Optimierung

Eine Vereinfachung stellt der Entwurf nach Bild unten dar. Abhängig von der Programmierung der Steuerleitung  $Y_c$ , wird zwischen den beiden Ausgangsstufen ausgewählt. Verständlich, dass die Realisierung nicht ganz so einfach ist. Möglich wird sie überhaupt nur dann, wenn man es schafft, zwei Basiszustandsfolgen zu finden, die identisch sind. Die Nachfolgeschaltung kann ein Decoder sein, ein Speicher oder in manchen Fällen sogar nur ein Multiplexer oder eine Ausgangstreiberstufe.



Entwurf ohne Optimierung

## Ausnutzung des Speicherraumes



Entwurf mit Optimierung

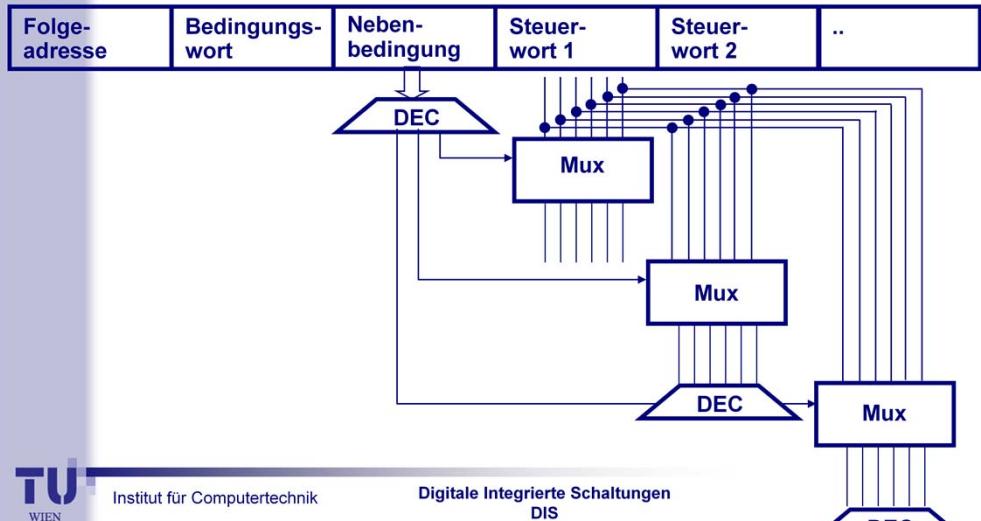
Die Prinzipielle Einsparung im Idealfall zeigt Bild oben.

In der Literatur findet man für die Bezeichnung *Mikrobefehlmehrachtfachcodierung* oft auch *Mikroprogrammspeicher mit umschaltbaren Ausgangsdecodern*.

## Optimierung durch Mikrobefehl-Mehrachcodierung

wenn Aktionen gegenläufig

Beispiel:



Ein reales System könnte somit eine Schaltung nach Bild oben aufweisen. Ob das Prinzip überhaupt verwendet werden soll, muss im jeweiligen Fall individuell entschieden werden, da es schnell zu unüberschaubaren Programmen führen kann (vergleiche in der Softwaretechnik die Anweisung "goto").



## Literatur

Die mit \* gekennzeichnete Literatur wird zur Vertiefung des Studiums besonders empfohlen.

- /Bode88/ Bode, A.: (Hrsg.) RISC-Architekturen; BI Wissenschaftsverlag, Mannheim, Wien, Zürich; 1988
- /Bund88/ Bundschuh, B; Sokolowsky, P.: Rechnerstrukturen und Rechnerarchitekturen; Friedr. Vieweg & Sohn; 1988
- /Haye88/ Hayes, J. P.: Computer Architecture and Organization, McGraw-Hill Publishing Company; 2<sup>nd</sup> Edition; 1988
- /Host87/ Hoster, P.; Manstetten, D.; Pelzer, H.: RISC - Reduced Instruction Set Computer, Konzepte und Realisierungen, Dr. Alfred Hüthig Verlag Heidelberg, 1987
- /Jung92/ Jungmann, D.; Stange, H.: Einführung in die Rechnerarchitektur; Carl Hanser Verlag München Wien, 1992
- /Lieb03/\* Liebig, H.: Rechnerorganisation, Die Prinzipien; Springer-Verlag; 3. Auflage, 2003
- /Märt01/ Märtin, Chr.; Rechnerarchitekturen, CPUs, Systeme, Software-Schnittstellen; Fachbuchverlag Leipzig im Carl Hanser Verlag, 2001
- /Ober03/ Oberschelp, W.; Vossen, G.: Rechneraufbau und Rechnerstrukturen, Oldenbourg, 9. Aufl. 2003
- /Patt94/ Patterson, D. A.; Hennessy, J. L.: Computer Organization and Design, the Hardware / Software Interface; Morgan Kaufmann Publishers, San Mateo, California, 1994 (siehe auch deutsche Ausgabe /Henn94/)
- /Remb87/ Rembold, U.: Einführung in die Informatik für Naturwissenschaftler und Ingenieure; Hanser Verlag, 1987
- /Remb94/ Rembold, U.; Levi, P.: Realzeitsysteme zur Prozeßautomatisierung; Carl Hanser Verlag München Wien, 1994
- /Tan90/\* Tanenbaum, A. S.: Structured Computer Organization; Prentice-Hall International Editions; 3<sup>rd</sup> Edition; 1990
- /Tab95/ Tabak, D.: Advanced Microprocessors; McGraw-Hill, Inc.; 1995
- /Wal80/ Waldschmidt, K.: Schaltungen der Datenverarbeitung; Teubner, B. G., 1980
- /Wilk51/ Wilkes, M. V.: The Best Way to Design an Automatic Calculating Machine; Rept. Manchester University Computer Inaugural Conf., pp. 16-18, 1951 [Reprinted in E. E. Swartzlander (ed.): Computer Design Development: Principia Papers, pp. 266-270, Hayden, Rochelle Park, N. J., 1976]
- /Zuse93/ Zuse, K.; Der Computer - Mein Lebenswerk; Berlin Heidelberg New York Tokio, Springer Verlag 1993
- /Märt03/ Märtin, Ch.: Einführung in die Rechnerarchitektur; Fachbuchverlag Leipzig, 2003

# Digitale Integrierte Schaltungen

384.086

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



# **Schaltwerke hoher Komplexität**

## **Kapitel 4**

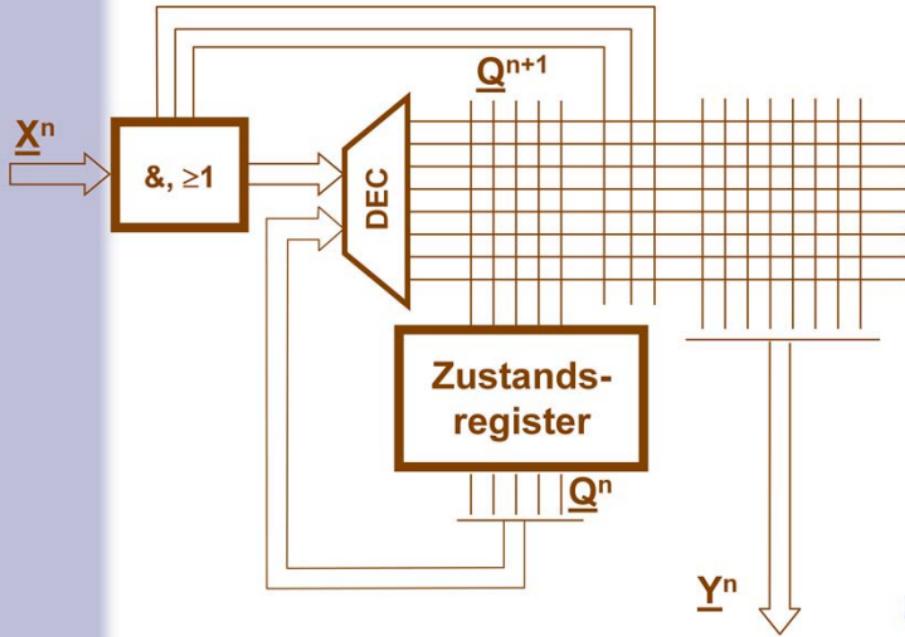
**Teil 3**

**Vollständiges Steuerwerk eines Mikroprozessors**

**+**

**Unterbrechungs-Prinzipien des Steuerwerkes des  
Mikroprozessors**

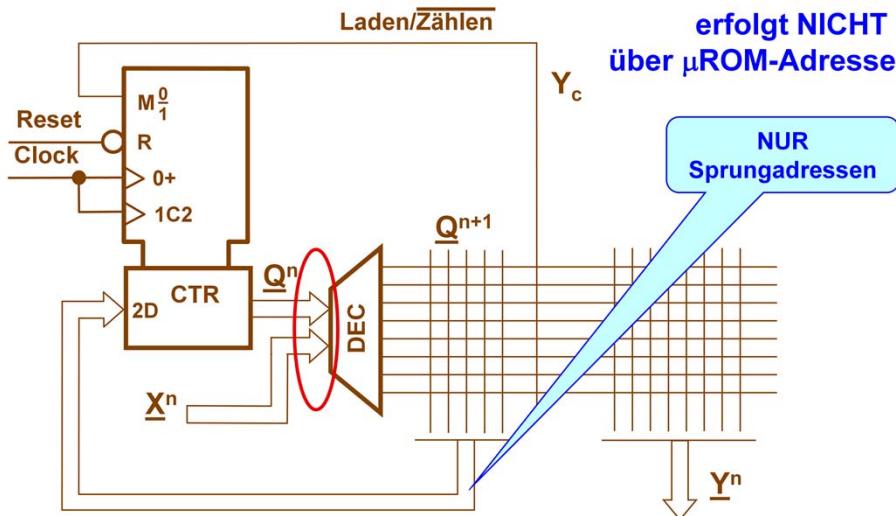
# $\mu$ Programmschaltwerk eines Prozessors



prinzipieller und  
einfachster Art

Wie sehen reale  
 $\mu$ Programmschaltwerke  
aus?

## Inkrementieren + Dekrementieren

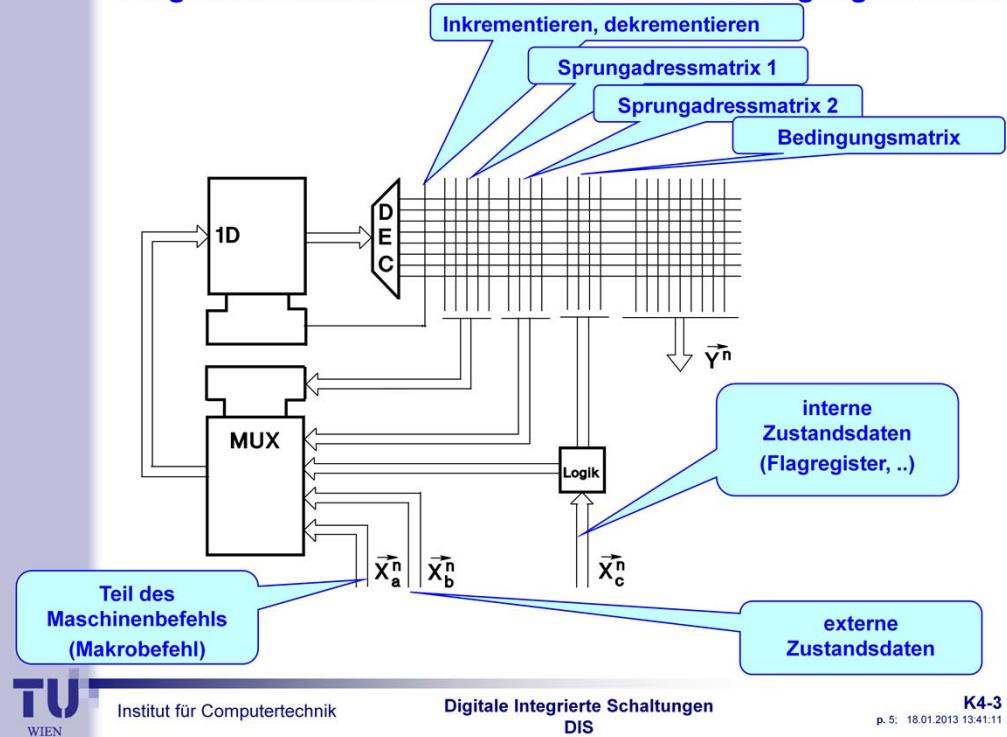


- Inkrementieren + Dekrementieren erfolgt über die niedrigerwertigen Adressbits
- Einsprünge über die oberen Adressbits

### (h) Integration eines Mikroprogramm-Counters

Die Integration eines Mikroprogramm-Counters versteht sich eigentlich von selbst und wurde bisher der Einfachheit halber ignoriert. Wenn für Mikrobefehle wie *Inkrementieren* oder *Dekrementieren* stets die Zustandsfolgeadresse ermittelt und abgespeichert werden müsste, wäre das sehr speicherplatzfressend. Entsprechend dem Verfahren der Bedingungsmatrix kann dagegen eine Lade-, –Zählen-Matrix integriert werden, wenn gleichzeitig das Zustandsfolgeaddressregister durch einen ladbaren Zähler ersetzt wird. Im diesem Fall müssen nur Sprungadressen programmiert werden. Da die Anzahl der Einsprungadressen dabei im Allgemeinen gering sein dürfte, wird darüberhinaus eine zusätzliche Nanoprogrammierung von Vorteil sein, wenn die Signallaufzeiten dies erlauben. Das Ergebnis dieser Überlegung führt zur nächsten Folie.

# Programmsteuerwerk mit unterschiedlichen Eingangsvektoren



Achtung: Register genau umgekehrt gezeichnet wie im Bild vorher.  $Y_c$  geht noch immer in den Ansteuer teil.

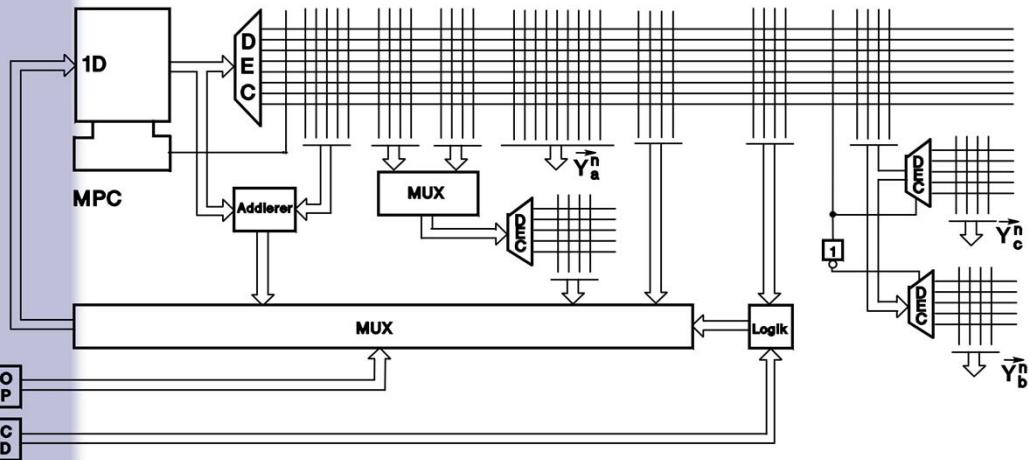
## Programmschaltwerksvariationen

Im Folgenden soll nur kurz gezeigt werden, wie sich all die besprochenen Verfahren zusammenführen lassen und mit vielen weiteren eine breite Palette von Möglichkeiten eröffnen, die in der Praxis zahlreiche Anwendungen finden.

### (a) Differenzierung zwischen unterschiedlichen Eingangsvektoren

Um die Flexibilität zu steigern, bieten sich ohne Frage weitere Möglichkeiten an, die nicht alle angeführt werden können. Nur einige sollen exemplarisch vorgestellt werden. In Bild oben wird vor den Zähler ein Multiplexer angeordnet, über den verschiedene Eingangsvektoren auf den Zähler geschaltet werden können. Unterschieden wird im Allgemeinen zwischen Datenteilen  $X_a$  des Maschinenbefehls (die restlichen Daten des Maschinenbefehls werden den Multiplexern und Registern des Operationswerkes direkt zuge-führt) und internen und externen Zustandsinformationen, die einmal direkt als Daten für den Vektor  $Q^{n+1}$  dienen ( $X_b$ ) und mit Daten der Bedingungsmatrix verknüpft werden ( $X_c$ ). Angesteuert werden kann der Multiplexer durch Daten des Mikroprogrammschaltwerkes.

## Komplexes Beispiel



CD: Condition Date

DEC: Decoder

MPC: Micro Program Counter

OP: Operation Date

### (b) Vorgehensweise beim Entwurf

Selbstverständlich lassen sich nun die unterschiedlichsten Variationen von Mikroprogrammschaltwerken ausmalen. Reale Systeme haben oft 100-bit-Mikroprogrammworte, und können sehr komplex ausfallen. Es ist deshalb für den Entwurf wichtig, schrittweise vorzugehen. Das bedeutet, zunächst den gewünschten Maschinenbefehlssatz vorzugeben, dann die prinzipielle Architektur zu entwerfen und zuletzt die Optimierungen vorzunehmen. Ein Aufbau auf der Basis verschiedener unterschiedlicher Optimierungsverfahren, wie er in Bild oben beispielhaft wiedergegeben wird, ist nur effizient, wenn von der nicht optimierten Schaltung ausgegangen wird. Es ist auch selbstverständlich, dass außer dem klaren Top-Down-Entwurf auch nur eine sehr differenzierte Modularisierung zielführend sein kann. Ohne Modularisierung in solch komplexen Systemen vorzugehen, führt schnell zu unüberschaubaren Einheiten.

# *Wiederholung 1:*

## **Wesentliche Schritte zum elementaren Prozessor (nach Neumann)**

- I. Entwicklung des Mealy-Automaten
- II. Trennung zwischen Operationswerk und Steuerwerk
- III. Aufteilung des Operationswerkes in
  - Operandenblock (Register)
  - Operatorblock (arithmetische, boolesche, .. Verknüpfungen)
  - Quellenauswahlnetz (boolesche Verknüpfungen, MUX, ..)
  - Verzweigungscodewandler (Reduktion der Variablen)
- IV. Programm gespeichert in ROM (= programmiertes Schrittschaltwerk)
- V. Folgeadressen als Folgeadressmatrix ebenfalls im ROM
- VI. Programmadresse wird mitbestimmt durch X
- VII. ..



Vorteil des Aufbaus:

## **Wiederholung 2:**



- i. Atomare Eigenschaft des Systems kann sehr komplex sein.
- ii. Anwender stößt das System über das Makrosystem an.
- iii. Anwender braucht sich nicht mit schaltungstechnischen Problemen aufzuhalten (EMV, ..)
- iv. Leichte Handhabung komplexer schaltungstechnischer Abläufe.
- v. Es sind mächtige Maschinenbefehle (= Makrobefehle) möglich.
- vi. Durch leichte Variation ist hohe Flexibilität garantiert (ROM-Änderung = Verhaltensänderung, ..).
- vii. Variabler Einsatz ermöglicht hohe Stückzahl bei der Produktion.

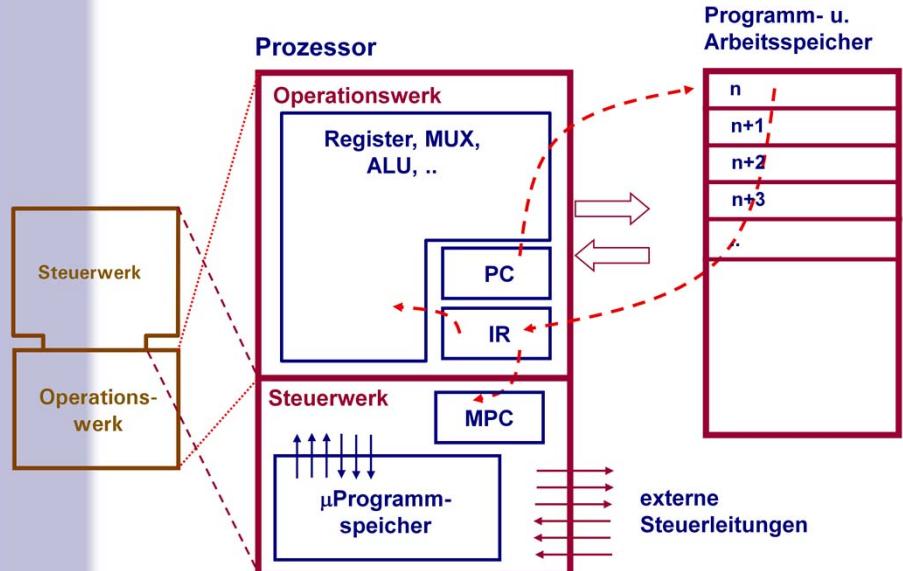
# Systemablauf bei einem $\mu$ P:

PC (Program Counter: Register/Zähler)  
*gibt die Adresse des folgenden MB (Maschinenbefehls) im externen Speicher vor.*

MB (Maschinenbefehl)  
*wird vom externen Speicher ins IR (Instruction Register) übernommen*

Datum des IR initiiert den MPC ( $\mu$ Programm-Counter)  
*Start des  $\mu$ Programmablaufs*  
:  
*PC wird inkrementiert oder neu geladen*

# Prinzip des Mikroprozessors (Basics)



Heutzutage sind Mikroprozessoren und Mikrocontroller mikroprogrammiert, wenn nicht eine RISC-Architektur zugrunde liegt. Der Ablauf ist im Allgemeinen wie folgt: Im Program Counter (PC) steht die Speicheradresse des nächsten Maschinenbefehls des mikroprozessorexternen Speichers (kann aber im Controller sein). Dieser Befehl wandert in das Instruction Register, das wie der Program Counter (PC) als Teil des Operationswerkes des Mikroprozessors angesehen wird. Das Datum des Instruction Registers wird decodiert und steuert zum Teil direkt Multiplexer und Register im Operationswerk, wirkt darüberhinaus als  $X_i$ -Vektor (in Bild oben: OP) auf das Mikroprogrammsteuerwerk und veranlasst dort entsprechende Mikroprogrammabläufe.

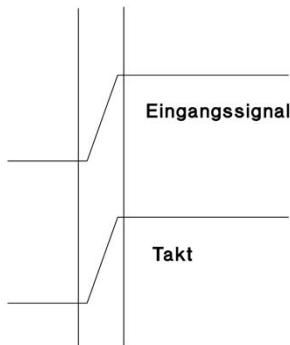
Weitere Eingänge des Mikroprogrammschaltwerkes sind das "interne" Zustandsdatum, das vom Operationswerk kommt (Folie vorher: CD), sowie Informationen, die über entsprechende Steuerleitungen von außen in den Prozessor gelangen. Das CD, das vom Operationswerk zum Steuerwerk transferiert wird, kann einmal von operationswerkinternen Überlauf-Flags, Zero-Flags usw. herrühren, oder es können Informationen von in das Operationswerk eingelangten Daten sein.

Für den Designer ergeben sich somit zwei kritische Schnittstellen: die von extern einlangenden asynchronen Steuerinformationen und die Ansteuerung externer Speicher. Von außen eintreffende Steuerinformationen müssen synchronisiert werden, was im Allgemein kein Problem bereitet, doch zwischendurch nicht ganz unproblematisch sein kann (siehe Kapitel 3). Der Designer muss sich bewusst sein, dass er es mit asynchronen Signalen zu tun hat, die er nicht einfach mit synchronen "mischen" darf. Der Entwurf muss entsprechend ausgeführt werden. Im allgemeinen differenziert er zwischen dem synchronen und asynchronen Teil und gestaltet den asynchronen auf Kosten des synchronen möglichst klein (mit nur wenigen Zuständen), da er sehr viel aufwendiger im Entwurf ist.

Zusatzbemerkung: Es soll wiederum die Definition zugrunde gelegt werden, dass ein Mikrocontroller einen Mikroprozessor plus einen Speicher plus Ein- und Ausgabeeinheiten auf *einem* Chip enthält.

Es bedeuten: PC: Program Counter, IR: Instruction Register, MPC: Micro Program Counter.

# Synchronisationsunsicherheit



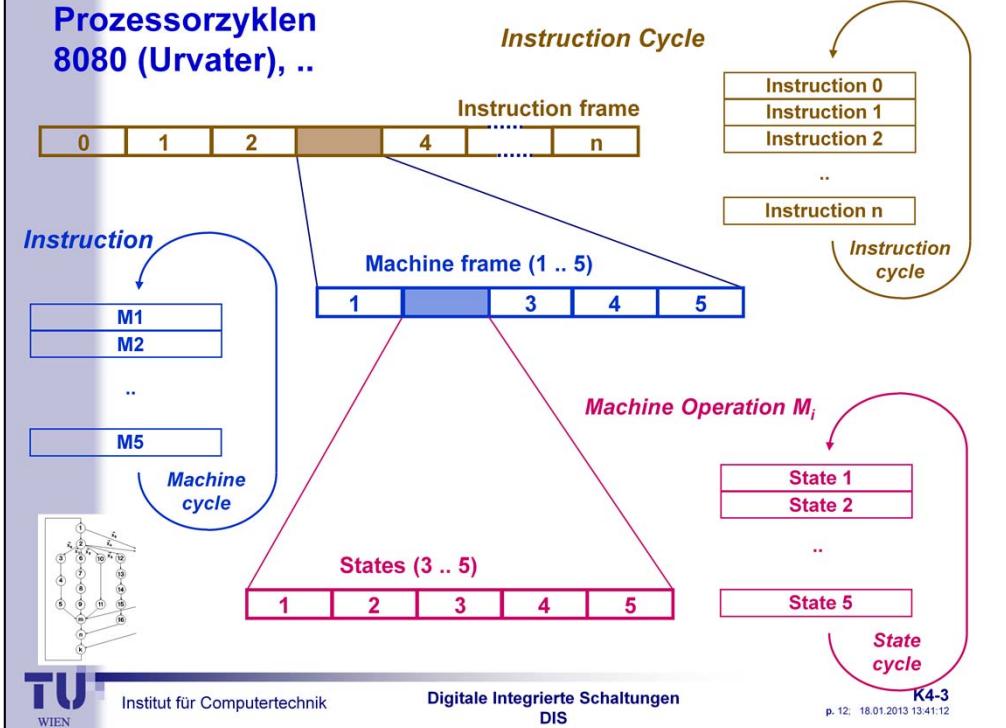
Wie ist das  
Problem  
lösbar?

Es gibt aber noch weitere Probleme in diesem Zusammenhang. Da heute Prozessoren höher und höher getaktet werden, hat es sich zum Beispiel als schwierig herausgestellt, entsprechend "schnell", das heißt in einem Taktzyklus rechtzeitig zu reagieren. Die Thematik wird in Kapitel 5 noch näher beleuchtet werden. Nur so viel zum Verständnis: Ein synchroner ASIC-Entwurf schreibt vor, dass jedes externe, asynchrone Eingangssignal über ein synchrones FF zu synchronisieren ist. Liegen die Signale lang an, und kann eine Verzögerung über mehrere Takte hinweg in Kauf genommen werden, spielt dieser Aspekt keine Rolle. Man denke an den Aufbau einer USART 8051A (Universal Synchronous/Asynchronous Receiver/Transmitter). Bei ihm wird eine mindestens 16-fache Grundtaktung vorgenommen, was die sichere Erfassung des Eingangssignals garantiert.

Anders liegen jedoch die Verhältnisse, wenn man mit jedem Takt direkt erkennen will, ob ein Signal eingetroffen ist. Dann kann die einlaufende Flanke in die Übergangsphase des FFs fallen, und man kann nicht mit Sicherheit sagen, ob der Eingangsimpuls noch erfasst werden konnte oder nicht (Bild oben). Will man derartige Synchronisationszeiten erreichen, benötigt man aufwendigere Synchronisationsschaltungen (beispielsweise zwei parallele, zeitlich versetzte Synchronisationseinheiten), was bei vielen Eingangssignalen wiederum Siliziumfläche kostet.

Die zweite "kritische" Schnittstelle eines Prozessors oder Controllers ist die zur externen Speichereinheit oder sogar zu externen Ein- und Ausgabebausteinen hin. Wirtschaftlich ungünstig kann es sein, wenn zu einem Prozessor nur ganz bestimmte Bausteine mit einem bestimmten Timing passen. Das sind Fragen der Verfügbarkeit und der Kosten. Es gibt deshalb schon Hersteller, die dazu übergegangen sind, sogenannte "intelligente" Speicherschnittstellen zu integrieren. Sie meinen damit "programmierbare" Speicherschnittstellen, also Speicherschnittstellen, die den gewünschten Speicherbausteinen angepasst werden können (ohne Rückwirkung auf das Mikroprogrammsteuerwerk). Wenn man sich nun die beiden vorgehenden Folien vor Augen führt, erkennt man, dass dies nicht ganz so einfach zu lösen ist, wenn der Aufwand nicht zu groß werden soll.

# Prozessorzyklen 8080 (Urvater), ..



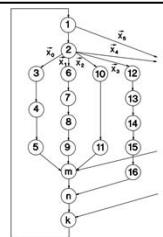
Steuerwerke von Mikroprozessoren sind im Allgemeinen schon deshalb im Entwurf aufwendiger, da vielfältige Anforderungen an das Timing zu stellen sind. Darüberhinaus wird bei heute erhältlichen Mikroprozessoren und Mikrocontrollern das "Innenleben", vor allem des Steuerwerkes, selten offen gelegt, und die Architektur eines Prozessors, und vor allem seines Steuerwerkes, bleibt letztendlich das Geheimnis des Herstellers. Es ist sein Knowhow und damit sein Kapital. In der Literatur sind die Informationen darüber entsprechend spärlich.

Ein Baustein, über den dagegen viel publiziert und dessen Architektur in der Literatur gut beschrieben wurde, ist der Prozessor 8080A. Ihn kann man auch als den "Urvater" der Prozessoren bezeichnen, denn er ist der erste, der eine weite Verbreitung gefunden hat. Anhand des 8080A sollen deshalb im folgenden einige kurze Erläuterungen erfolgen, die die "Basics" heutiger Prozessoren darstellen.

Wie bei den meisten mikroprogrammierten Prozessoren, und nur von denen soll in diesem Abschnitt die Rede sein, wird von einer dreifachen Hierarchie von Prozessorzyklen ausgegangen (Bild oben). Die oberste Ebene bildet der *Befehlszyklus* (Instruction Cycle), der das *Maschinenprogramm* darstellt.

Maschinenprogramm (Machine Frame): Befehlszyklus (Instruction Cycle)

## Die 3 Unterbrechungs-Prinzipien (8080, ..)



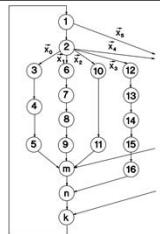
Das heißt, der Prozessor liest von einem Speicher (Bild oben)  $n$  Maschinenbefehle (assemblierte Assembler-Befehle oder direkt aus einer Hochsprache compilierter Maschinenbefehle) ein. Teile dieser Informationen dienen zur direkten Ansteuerung von Registern und Multiplexern im Operationswerk, der Rest wird dem Steuerwerk des Mikroprozessors zugeführt. Dieser "Rest" steht somit als OP (Operation Code) im Steuerwerk an und generiert über den *Micro Program Counter* (MPC) die einzelnen *Maschinenzyklen* (Machine Cycles). Ein Maschinenzyklus ist die Ausführung *eines Befehls* und besteht aus  $i$  *Machine Operations*:  $M_1, M_2, \dots, M_i$ ;  $i = 1$ : minimal,  $i = 5$ : maximal). Ein Machine Operation  $M_i$  wird über ein *State Cycle* gebildet, der im Zustand 1 ( $T_1$ ) beginnt und durchläuft nacheinander verschiedene *Zustände* (States), bis er wieder im Zustand 1 ankommt.

Zusatzbemerkung:  $n$  ist die Anzahl der Befehlswort (Instructions, Statements) eines in Maschinensprache formulierten Programms.

Die vollständige Bearbeitung eines Maschinenbefehls (Instruction) stellt einen Maschinenzyklus (Machine Cycle) dar, der in Bild rechts oben in Zustandsspalten durchläuft. Ein in Bild rechts oben einmal senkrecht durchlaufener Durchgang ist der State Cycle (wobei man in der Literatur im Allgemeinen nur von States spricht).

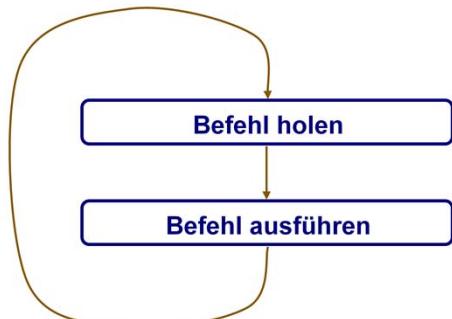
Dieser Ablauf kann in verschiedener Weise unterbrochen werden. Sind an den Prozessor langsame Speicher angeschlossen, kann der Speicher bei einem MEMR (Memory Read) oder MEMW (Memory Write) mit einem  $\neg$ READY (= Anforderung) reagieren, was den Prozessor veranlasst zu warten und über WAIT (Antwort) nach außen signalisiert. Selbstverständlich ist die  $\neg$ READY-Anforderung nicht nur Speichern vorbehalten, auch für beliebige andere Bausteine kann diese "Zeitverzögerung" implementiert werden.

## Die 3 Unterbrechungs-Prinzipien (8080, ..)



Eine weitere Möglichkeit, den Taktzyklus zu unterbrechen, besteht darin, eine Anforderung über HOLD einzuleiten, die eine DMA-Anforderung (Direct Memory Access) darstellt. Die Antwort ist in diesem Fall HLDA (HOLD-Acknowledge). Eine dritte Möglichkeit bietet der Softwarebefehl HALT, wobei der Prozessor in diesem Fall mit einem HLTA (HALT-Acknowledge) antwortet. Man könnte nun annehmen, dass alle drei Unterbrechungsmöglichkeiten im Steuerwerk die gleiche oder zumindest eine ähnliche Abfolge verursachen. Berücksichtigt man jedoch, dass alle drei unterschiedliche Ursachen und Aufgaben haben, wird klar, dass man sie unterschiedlich handhaben muss.

## Ablauf



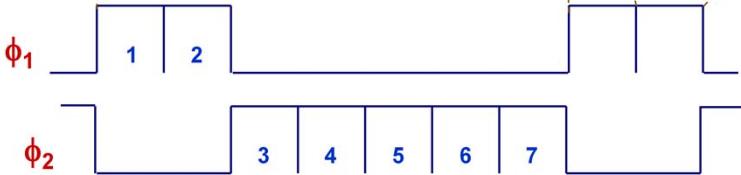
Fetch-Phase

Execute-Phase

Taktaufbau

Taktperiode des Prozessors

Taktperiode  
des Quarzes

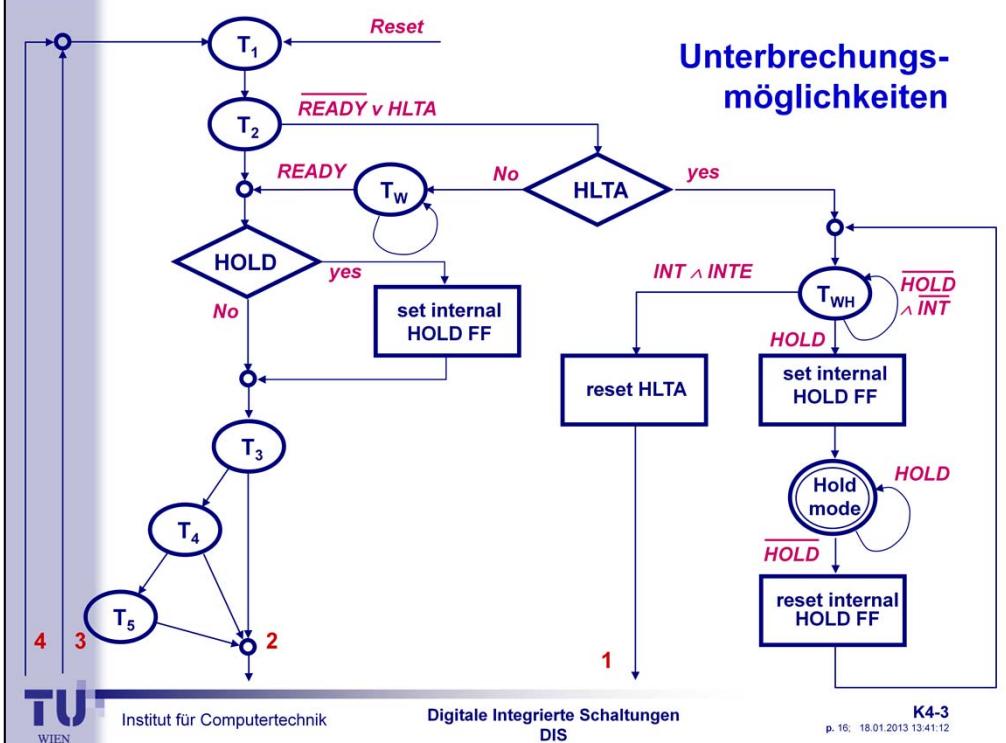


Bevor dies aber im Detail erläutert wird, kurz noch zwei prinzipielle Erläuterungen zum Verständnis: Gemäß der Darstellung in Bild oben rechts beginnt jeder *Machine Cycle* mit einer *Fetch-Phase*, das heißt, in der *Machine Operation M<sub>1</sub>* wird ein *Maschinenbefehl* eingeholt und ausgeführt. Der Machine Cycle kann insgesamt 5 Machine Operations ( $M_1$  bis  $M_5$ ) umfassen. Die *Execute-Phase* variiert dabei stark von Maschinenbefehl zu Maschinenbefehl.

Dem 8080A ist ein 2-Phasentakt mit insgesamt 9 Subtakteinheiten zugrunde gelegt (Bild unten), um die prozessorinterne Laufzeiten zwischen Operations- und Steuerwerk besser beherrschen zu können, ein Verfahren. Die beiden zueinander verschobenen Takte werden mit  $\phi_1$  und  $\phi_2$  bezeichnet und sind unsymmetrisch zueinander. Die Unsymmetrie spiegelt die prozessorinterne Laufzeitproblematik wider.

Zusatzbemerkung: Die Bezeichnungen *M1-Zyklus* und *Fetch-Zyklus* werden deshalb auch synonym verwendet.

## Unterbrechungsmöglichkeiten

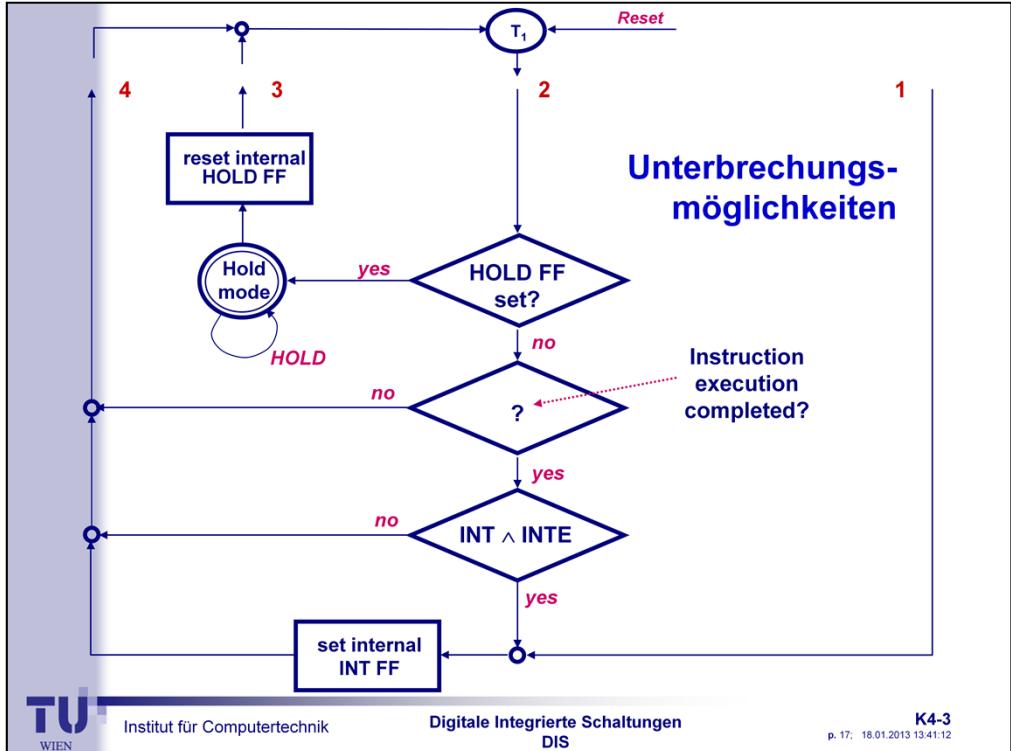


Nun zum Zustandsgrafen des 8080A-Steuerwerkes selbst. In Bild 4.84 ist die prinzipielle Darstellung wiedergegeben, unabhängig davon, welche Befehle abgearbeitet werden. Die zentrale Thematik stellen hier die drei genannten verschiedenen Unterbrechungsmöglichkeiten dar.

Nach dem RESET wird der Zustand 1 (State 1:  $T_1$ ) eingenommen. Damit legt der Prozessor über das Daten-Port ein Datum nach außen, das als Steuerwort zu interpretieren ist (beim 8080A wird der Datenbus und ein Teil des Steuerbus gemultiplext, um Pins zu sparen).

In  $T_2$  wird abgefragt, ob eine  $\neg \text{READY}$ -Anforderung anliegt. Das ist zu diesem Zeitpunkt notwendig, da der Prozessor bei minimal 3 und maximal 5 States zum Ende von  $T_2$  stets Daten über den Datenbus einlesen (beispielsweise für den Fetch-Zyklus) oder ausgeben möchte.

Langsame Speicher- oder I/O-Einheiten können dann entsprechend durch Setzen von  $\neg \text{READY}$  reagieren. Der Prozessor gelangt dadurch in den Zustand  $T_W$ , der die Anzahl der States pro State Cycle um einen ganzzahligen Wert verlängert, bis  $\neg \text{READY}$  auf READY gesetzt wird.



Nach der Abfrage auf  $\neg$ READY, beziehungsweise nach  $T_W$ , fragt der Prozessor den HOLD-Eingang ab, ob ein DMA angefordert wird. Im Gegensatz zur  $\neg$ READY-Anforderung hält der Prozessor jedoch nicht in  $T_2$  an, sondern setzt nur das interne HOLD-FF, was entsprechend nach außen über eine Steuerleitung signalisiert wird. Vom Timing her wird dies so organisiert, dass der Adress- und Datenbus beim READ-Modus schon während  $T_3$  und im WRITE-Modus nach  $T_3$  gefloatet wird, um einen Cycle-Stealing-Prozess zu ermöglichen. In  $T_4$  und  $T_5$  laufen prinzipiell prozessorinterne Prozesse ab. Nach  $T_5$  wird das HOLD-FF geprüft, ob die DMA-Anforderung immer noch besteht. Ist dies der Fall, bleibt das System im HOLD-Modus (wiederum eine ganzzahlige Verlängerung der Taktzeiten). Wurde die DMA-Anforderung in der Zwischenzeit zurückgenommen, wird das interne HOLD-FF zurückgesetzt. Der Cycle-Stealing-Prozess ist zu erreichen, indem die HOLD-Anforderung mit HOLD vor  $T_2$  gesetzt, aber nach  $T_3$  wieder zurückgenommen wird.

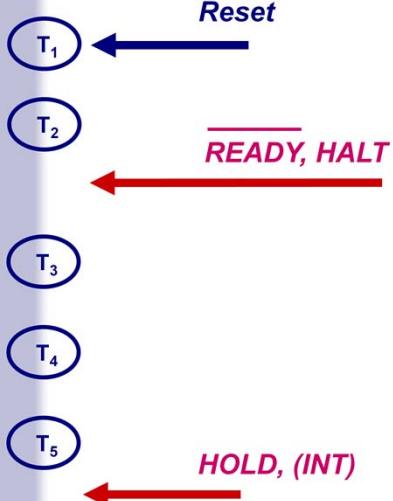
Der HALT-Befehl wurde eingeführt, um den Prozessor in einen "Ruhezustand" zu versetzen, wenn keine anderen Aktivitäten mehr anstehen. Wie aus dem Zustandsdiagramm zu ersehen ist, kann der Befehl erst in  $T_3$  zur Wirkung kommen, was bewirkt, dass in  $T_2$  des folgenden State Cycle ein gesetztes HLTA abgefragt werden kann. Der Prozessor läuft damit in eine Zustandsschleife, die er nur durch einen Interrupt wieder verlassen kann, wenn vorher allerdings auch das INTE (Interrupt-Enable-FF) gesetzt wurde. Wiederum ist  $T_{WH}$  ein ganzzahlig Vielfaches des Taktes.

Da während des  $T_{WH}$  auch ein DMA durch HOLD angefordert werden kann, kann der Prozessor damit zwar  $T_{WH}$  verlassen und in den HOLD-Modus wechseln, nach dessen Verlassen fällt er aber automatisch wieder in  $T_{WH}$  zurück.

Aus dem Zustandsdiagramm nach Bild oben kann man weiterhin ablesen, dass ein Interrupt nur nach der vollständigen Bearbeitung eines Befehls angenommen wird. Denn das interne Interrupt-FF wird erst dann gesetzt, wenn ein Interrupt (INT) anliegt und das Interrupt-Enable-FF (INTE) gesetzt ist sowie der vorausgegangene Befehl vollständig bearbeitet wurde (Instruction Execution Completed).

Das Beispiel zeigt, dass die Beschreibung und Darstellung des Machine-Cycle-Zustandsgrafen schon eines relativ einfachen Programmschaltwerkes schwierig wird. Um das Verhalten des Programmschaltwerkes des 8080A vollständig zu verstehen, wären weitere Detaildarstellungen, Timing-Diagramme usw. notwendig, worauf jedoch verzichtet werden soll.

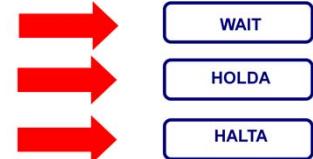
## Unterbrechungsmöglichkeiten



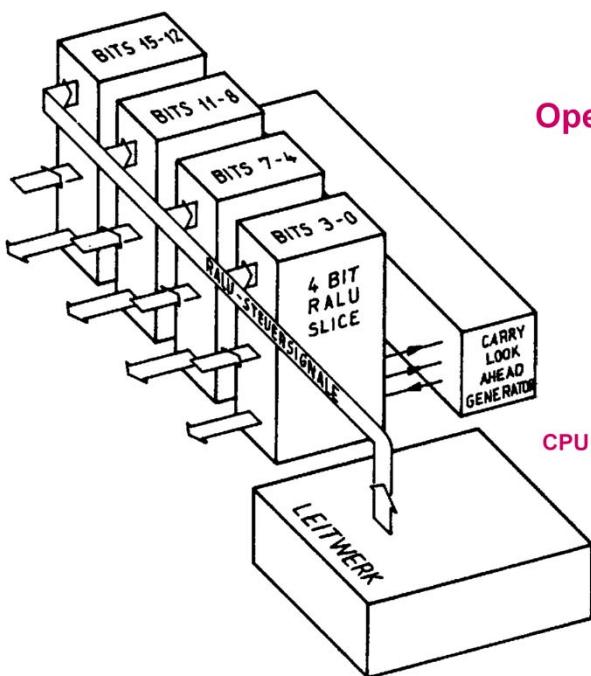
Anforderung



Reaktion

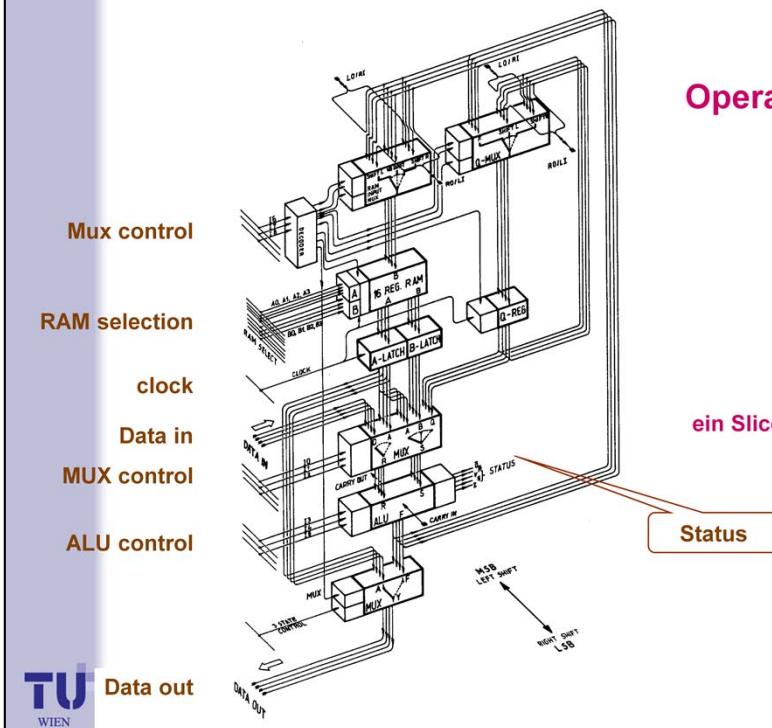


## Aufbau des Operationswerkes

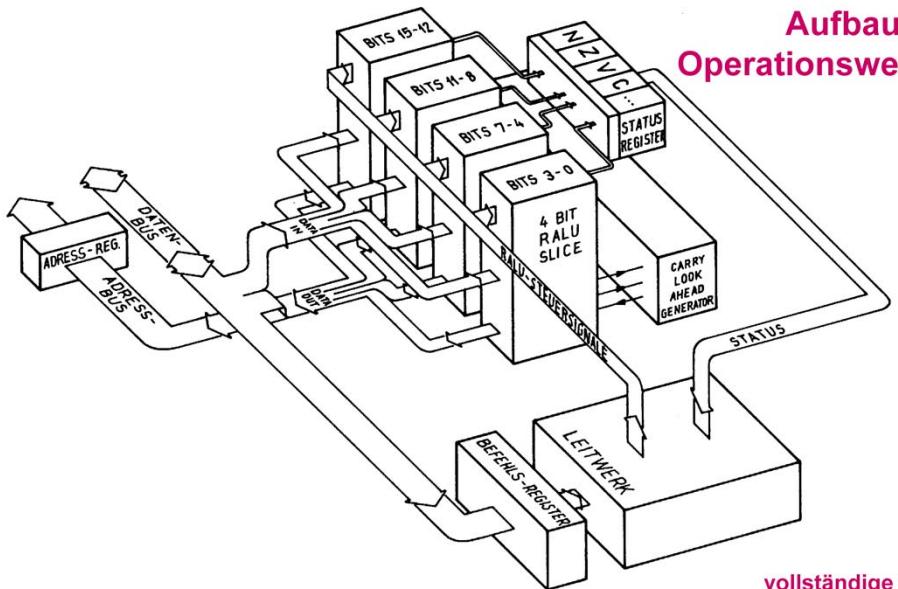


CPU aus 4-bit-Elementen

# Aufbau des Operationswerkes



## Aufbau des Operationswerkes



# Digitale Integrierte Schaltungen

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-1

b. 1  
18.01.2013 13:45:01

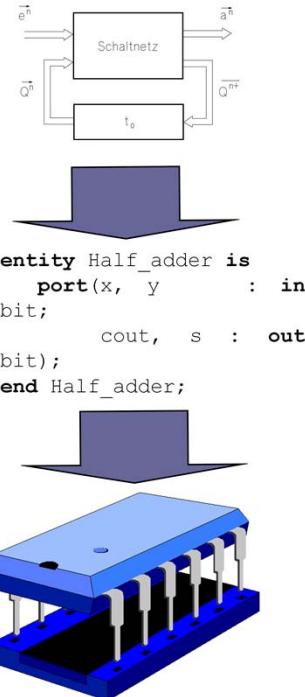
Die Kapitel eins bis vier behandeln Themen, die in vorausgegangenen Vorlesungen bisher nur oberflächlich oder gar nicht angesprochen wurden, die aber für die Hardwaretechnik wichtig sind. Im Folgenden soll nun eine Einführung in die Technik des ASIC-Schaltungsdesigns erfolgen. Die daraus gewonnenen Kenntnisse sind auch als Basis für Fächer zu verstehen, die die einzelnen Themen aufgreifen und vertiefen. Dies gilt vor allem für BIST (Built-in-Test-Systeme) und Boundary Scan, was z. B. in der Vorlesung *Fehlertolerante Systeme* näher bearbeitet wird, das gilt aber vor allem auch für die Sprache VHDL, der weitergehende Unterrichtseinheiten gewidmet sind.

# Kapitel 5

## Teil 5-1

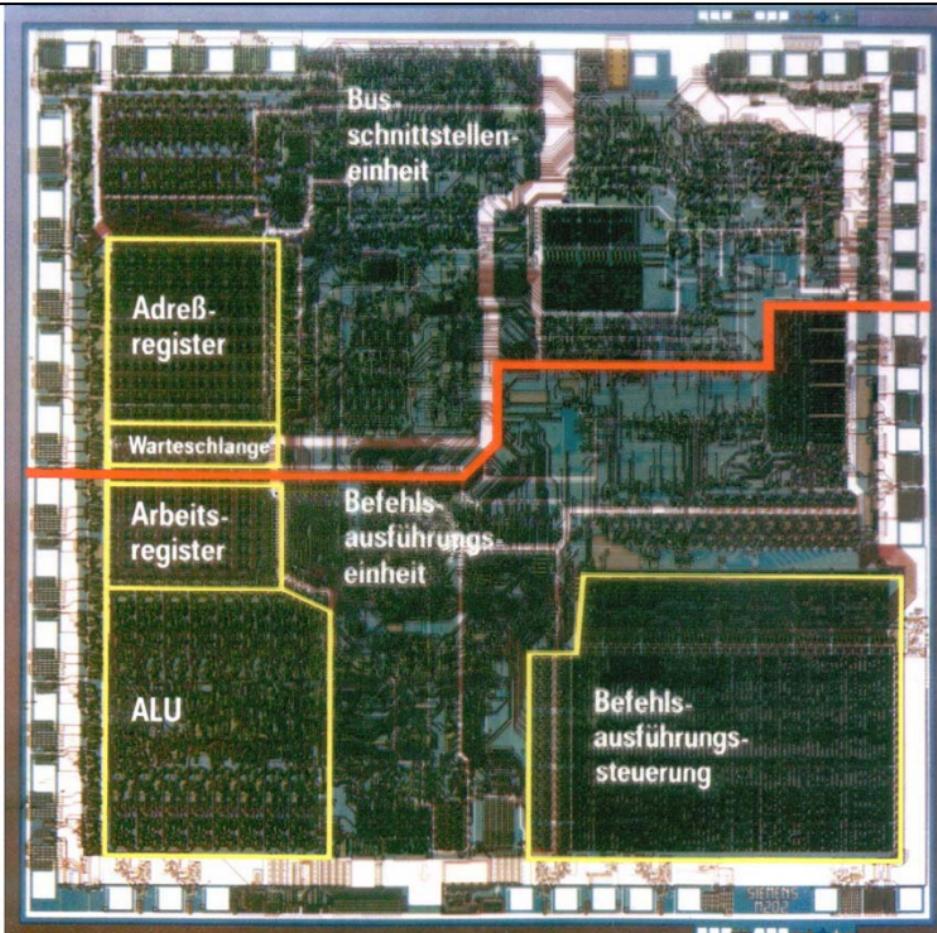
# ASIC Design

- Einführung & Prinzipielles
- Programmierbare Bausteine

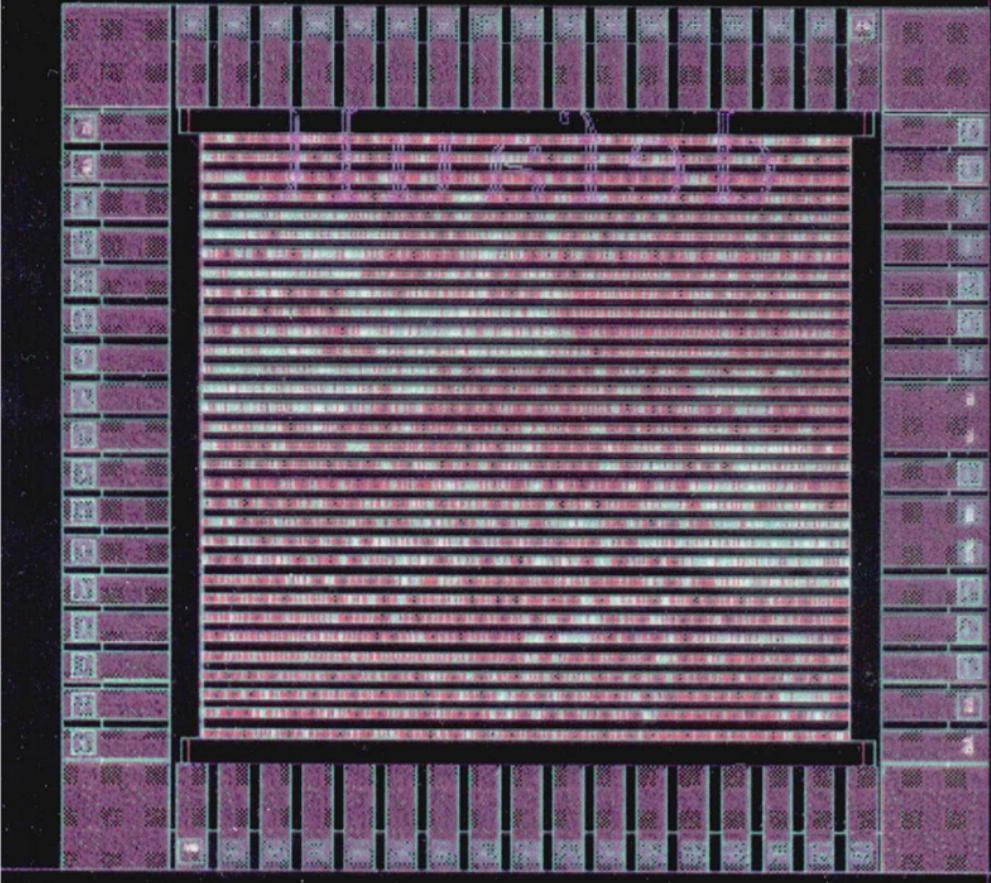


Die Bedeutung des Begriffs **ASIC** (Application Specified Integrated Circuit) wird in der Literatur, und dort vor allem in den nichtwissenschaftlichen Fachzeitschriften, immer noch unterschiedlich gehandhabt. Hier soll die allgemeinste Definition gelten, und prinzipiell sollen all die integrierten Bausteine gemeint sein, die nicht zu den Standardbausteinen zählen.

**μP**



# ASIC



Zitat 1943 von Thomas Watson, chairman of IBM:

*I think there is a world market for maybe five computers.*

Zitat 1977 von Ken Olson, President von Digital Equipment:

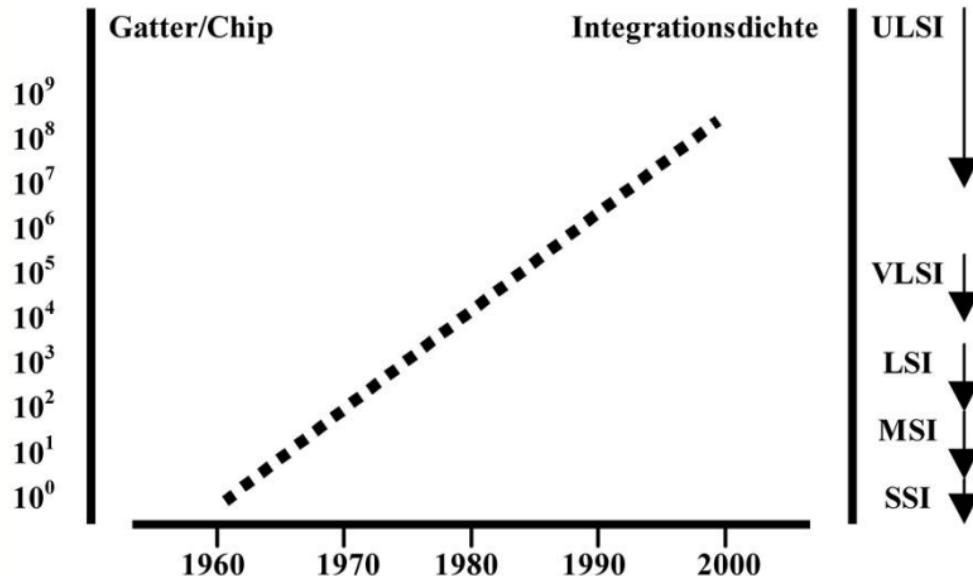
*There is no reason for any individual to have a computer in his home.*

Bzgl. Embedded Systems, Fieldbussystems, .. behauptete ich:

*Aufgrund der jetzigen Situation ist zu erkennen, dass sich in allen Bereichen in den nächsten Jahren sich Gewaltiges bewegen wird.*

*Die Anzahl der Sensoren, Aktoren und Stellsysteme werden ähnliche Zuwachsrraten annehmen wie im Kraftfahrzeugbereich die Elektronik, getrieben durch die Kraftfahrzeugtechnik selbst, aber auch durch den Konkurrenzkampf in der Industrie, im Bankengeschäft, in der Braunen Ware usw.*

# Scale Integration



**SSI:** Small Scale Integration

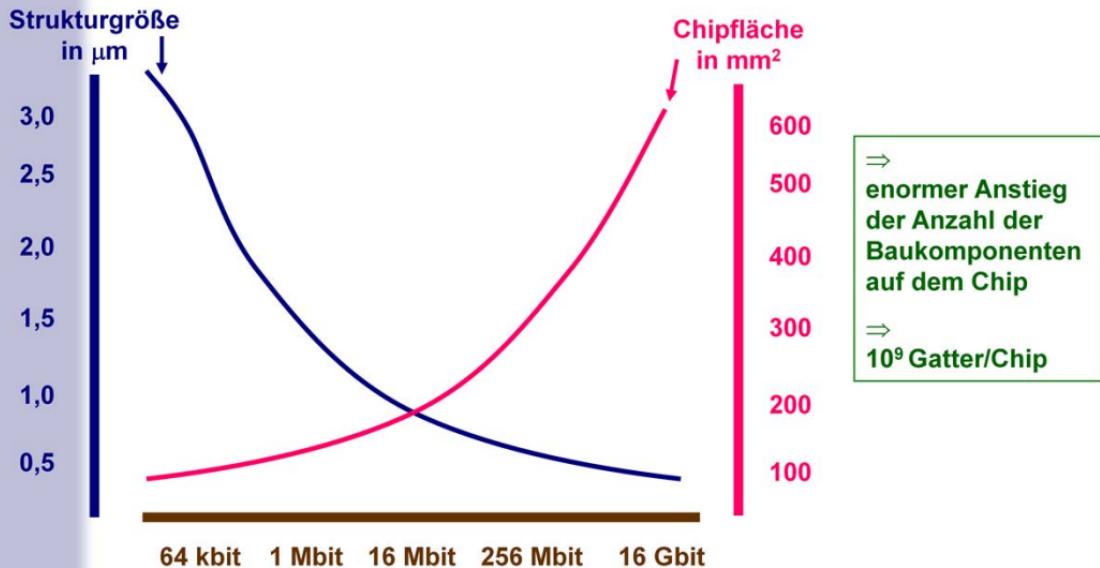
**MSI:** Medium SI

**LSI:** Large SI

**VLSI:** Very LSI

**ULSI:** Ultra LSI

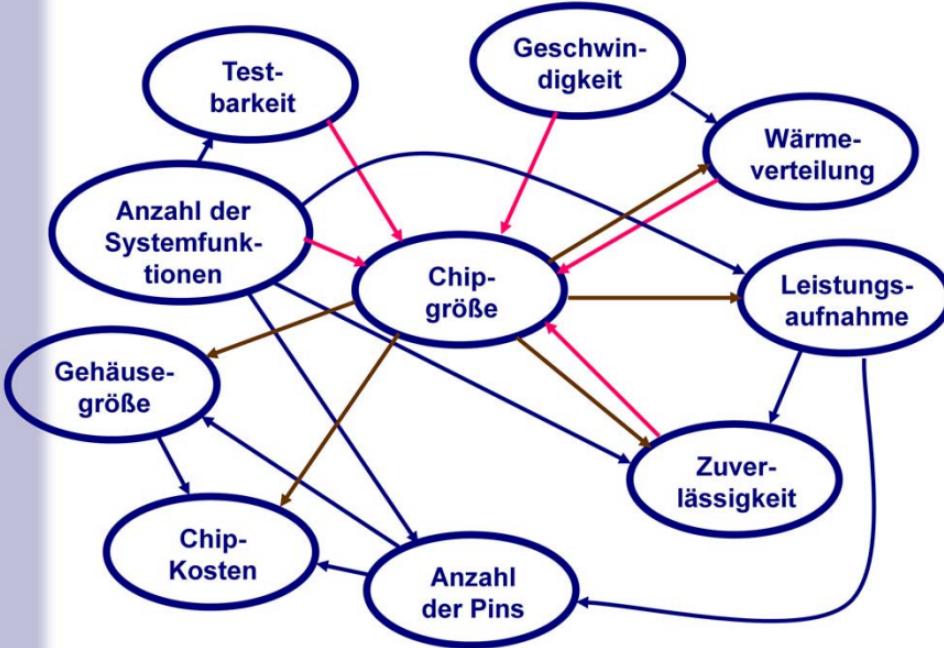
# Chipfläche $\Leftrightarrow$ Strukturgröße



# Vorteile

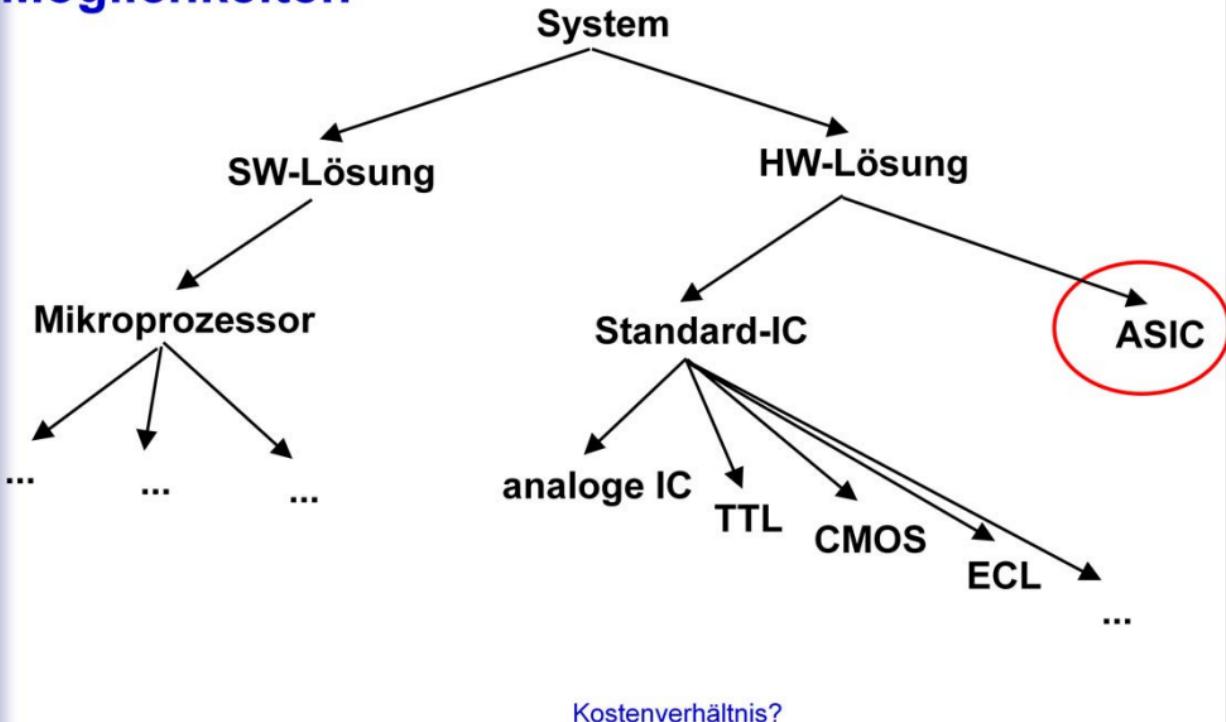
- **hohe Integrationsdichte**
  - *geringer Flächenbedarf*
  - *geringe Leistungsaufnahme*
  - *hohe Störfestigkeit*
  - *hohe Taktfrequenz*
  - *hohe Systemzuverlässigkeit*
- **"einfache" Integration von Intelligenz**
  - *hohe Testbarkeit*
  - *automatische Testbarkeit*
  - *geringe Produktionskosten*
  - *geringe Wartungskosten*

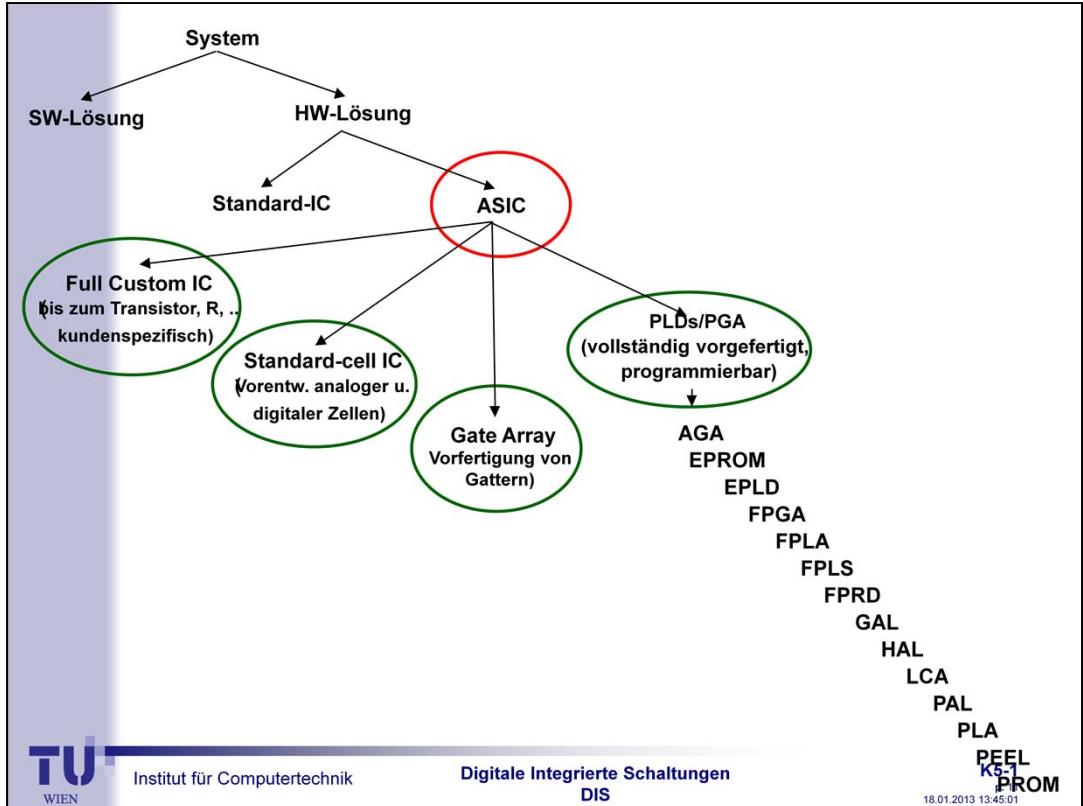
# Notwendigkeit für kleinere Strukturen



Das sind natürlich nicht alle Abhängigkeiten, nur in etwa die Wichtigsten.

# Möglichkeiten





Die Motivation, von diskreten Bauelementen wie Transistoren, Widerständen usw. zu diskreten logischen Schaltungen wie NAND, NOR usw. überzugehen, war, den Entwickler möglichst von elementaren Aufgaben des diskreten Schaltungsentwurfs zu befreien, was in letzter Konsequenz auch zum Mikroprozessor führte. Das bedeutet, der Entwurf wird auf einer höheren abstrakten Ebene durchgeführt, der Entwickler kann sich mehr auf die eigentliche Arbeit konzentrieren, nämlich auf den Entwurf der Funktionen, die gewünscht sind (die „Elektrotechnik“ wird ihm „funktionsfähig“ zur Verfügung gestellt [1]).

Mikroprozessoren waren jedoch anfangs teuer und im Grunde auch „langsam“. Selbstverständlich suchte man andere Lösungen. Eine Antwort waren die 1979 vorgestellten PLAs (Programmable Logic Arrays) der Firma Signetics. Beispielsweise über die KV-Methode entwickelte Schaltungen ließen sich darauf direkt abbilden, was auch bezüglich der Laufzeiten zu optimalen Schaltungen führte. Als weiteren Generationsschritt kann man die über UV-Licht löschen programmierbaren Bausteine bezeichnen, dann die elektrisch löschen programmierbaren UND- und ODER-Matrizen, sondern fügte auch zusätzlich FFs hinzu, so dass direkt Mealy- und Moore-Automaten realisiert werden konnten. Des Weiteren integrierte man spezielle logische Schaltungen wie Addierer oder Ausgangstreiber und Tristate-Buffer, so dass die Bausteine auch direkt in häufig angewendeten Schaltungen wie Prozessoren integriert werden konnten. Die Anzahl der Einzelkomponenten nahm dabei von Jahr zu Jahr drastisch zu, so dass die Aufgabe nun zunehmend darin besteht, die optimale Zellstruktur sowie den entsprechenden optimalen Compiler zu finden. Bei großen Bausteinen ist es nämlich nicht mehr sinnvoll, dass der Entwickler selbst festlegt, welcher Teil der Schaltung wo im programmierbaren PLD plaziert wird, dies kann ein Compiler effizienter lösen, da hier die verschiedensten Aufgaben zu bearbeiten sind, angefangen bei der Optimierung von Laufzeiten bis hin zur optimalen Wärmeverteilung im Baustein. Die Konsequenz wird deutlich: Mit PLDs lassen sich relativ schnell und unkompliziert digitale Schaltungen entwerfen. Darüber hinaus weisen sie im allgemeinen geringe Laufzeiten auf (aufgrund beispielsweise der Matrixstruktur oder der DN-Struktur), können vielfältig eingesetzt werden, erschweren einen unerlaubten Nachbau usw.

[1] Der Zusammenhang ist überspitzt dargestellt, um das Wesentliche deutlich werden zu lassen.

# ASIC

*Application specific integrated Circuit*

## ○ Semicustom ICs

*halbkundenspezifische Bausteine*

◆ PLDs

*(vorgefertigt) PROMs, FPGAs, ..*

◆ (Gate Arrays)

*- vorgefertigt werden Gatterzeilen; Wafer können daher in großer Stückzahl vorgefertigt werden; erst die letzte Maskenebene wird kundenspezifisch gefertigt; geringe Kosten + Entw.- + Produktionszeiten*

*- relativ viel Verdrahtungsfläche notwendig*

## ○ Custom ICs

*kunden- und anwendungsspezifisch*

◆ Standard-cell ICs

◆ Full Custom ICs



Differenziert wird oft zwischen folgenden zwei Gruppen:

- p Semicustom ICs und
- p Custom ICs,

wobei aus den folgenden Erläuterungen erkennbar werden dürfte, dass sie nicht einfach begründbar ist. Fest steht, Semicustom ICs sind mit wenig (PLDs) bis mittlerem Aufwand zu entwickeln, Custom ICs ausschließlich mit einem hohen. Eine feinere Klassifizierung ist:

- |  |  |
|--|--|
| p Semicustom ICs   | p Custom ICs   |
| <ul style="list-style-type: none"><li>• PLDs</li><li>• Gate Arrays</li></ul> | <ul style="list-style-type: none"><li>• Standard-cell ICs</li><li>• Full Custom ICs.</li></ul> |

Zu den PLDs sollen zunächst alle direkt programmierbaren Bausteine zählen, angefangen bei den PROMs (Programmable ROMs) bis hin zu den FPGAs (Field Programmable Gate Arrays), von denen, mit Ausnahme der PROMs, im folgenden die Rede sein wird. Die Gruppe der Gate Arrays (vorgefertigt werden Gatter), der Standard-cells (vorgefertigt werden analoge oder digitale Zellen) und der Full Custom Circuits (volle Entwicklung bis hinunter auf die Ebene der Transistoren, Widerstände usw.) wird dagegen nicht so ausführlich behandelt, da die Thematik sonst zu umfangreich würde. Wichtig ist dagegen zu wissen, wie solche Bausteine auf der logischen Ebene entworfen werden und was zu den Entwurfsprogrammiersprachen führt (beispielsweise zu VHDL). Dies im einzelnen in der Vorlesung zu behandeln, ist für einen Elektrotechniker im allgemeinen zu trocken und deshalb ineffizient. Entwurfsprogrammiersprachen sollen aus diesem Grund vor allem den zusätzlich angebotenen Übungen vorbehalten sein.

Sieht man von den PLDs einmal ab, spielt für ASICs die Testthematik, und hier wiederum der Built-in-Test, die zentrale Rolle beim Baustein-Design. Dies ist der Grund, warum der entsprechende Abschnitt von großer Bedeutung ist. Das Boundary Scan (oder ähnliche Verfahren) hat sich auf breiter Basis durchgesetzt, weshalb diesem Verfahren ein eigener Abschnitt gewidmet ist. Es sei noch darauf hingewiesen, dass über ASICs gerade in der letzten Zeit zahlreiche Bücher neu erschienen und auch Sonderhefte in Fachzeitschriftenreihen herausgegeben wurden.

*Zusatzbemerkung: Die spätere vorgenommene Differenzierung zwischen CPLDs und FPGAs soll hier noch keine Rolle spielen.*

# ASIC

*Application specific integrated Circuit*

- **Semicustom ICs**

*halbkundenspezifische Bausteine*

- ◆ **PLDs**
- ◆ **(Gate Arrays)**

- **Custom ICs**

*kunden- und anwendungsspezifisch*

- ◆ **Standard-cell ICs**

*- in einer Bibliothek vorentworfene Funktionseinheiten mit allen physikalischen und elektronischen Parametern*

*- kurze Entwurfszeiten; Packungsdichte höher als bei Gate Arrays*

- ◆ **Full Custom ICs**

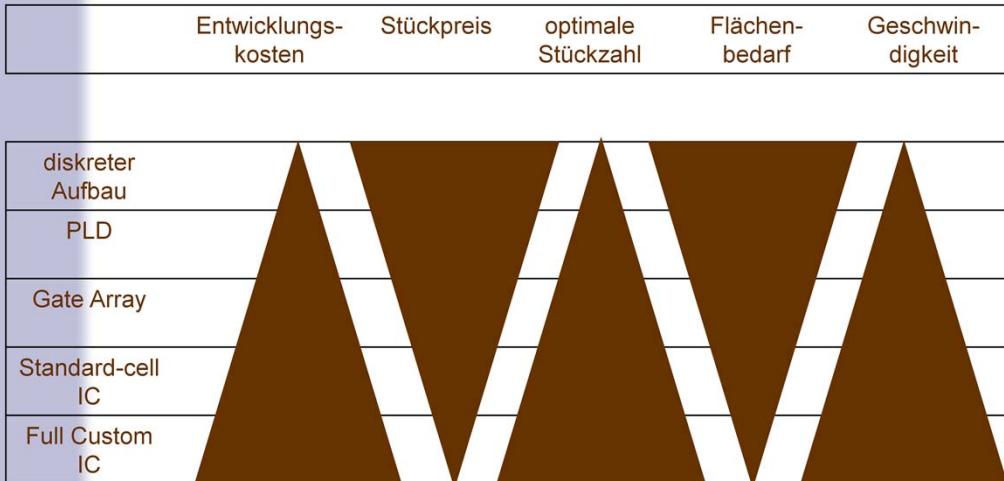
*- Entwurf bis "hinunter" zum Transistor*

*- "teuerste" Methode*



Eine Systemlösung verlangt die Entscheidung für eine Hardware. Dabei kann man sich für eine Lösung entscheiden, die einen hohen Hardwareaufwand erfordert, dafür aber eine relativ einfache Software voraussetzt (hardwareorientierte Lösung), oder, umgekehrt, einen geringen Hardwareaufwand, dafür aber ein umfangreiches Softwarepaket erforderlich werden lässt (softwareorientierte Lösung). Die extrem softwareorientierte Lösung wäre beispielsweise die Verwendung eines möglichst einfachen Hardwareprozessors (im allgemeinen ein Mikroprozessor). Ob dieser ein Standardbaustein ist oder speziell dafür entworfen wird, muss hier nicht diskutiert werden. Bei einer stark hardwareorientierten Lösung hat man sich bisher vor allem für Standardbausteine entschieden, da diese am preisgünstigsten waren. Dieses Bild hat sich zunehmend gewandelt. Erstens sind programmierbare Bausteine seit langem ebenfalls preisgünstig zu haben. Zweitens wird das Arbeiten mit Standard-cell ICs immer einfacher und auch für kleine Stückzahlen ( $> 500$ ) kostengünstiger. Drittens will man verhindern, dass Schaltungen einfach zu kopieren sind. Sie sollen andererseits aber ohne großen Aufwand veränderbar sein (was bei Standard ICs ganz gewiss nicht der Fall ist, dafür aber für viele PLDs zutrifft). Viertens sollen die Schaltungen einfach zu warten sein usw. Standard ICs werden deshalb heute großenteils durch Standard-cell ICs und PLDs ersetzt. Allerdings darf nun nicht der Umkehrschluss gezogen werden, dass die Zeit den Markt der Standard ICs überholt hat. Spezielle oder komplexere Bausteine, deren eigene Entwicklung oder Produktion noch immer zu teuer ist, werden auch weiterhin günstiger als Hardware als in Form eines Bibliotheksobjektes zu haben sein. Trotzdem ist der Trend spürbar, zunehmend Bausteine nicht nur in Hardware, sondern auch als Software- Bibliotheksobjekte anzubieten, die dann direkt in eigene ASIC-Entwürfe einfließen können.

# Vergleich



In Bild oben sind verschiedene Kriterien einander global gegenübergestellt, die nicht immer im einzelnen stimmig sind, doch ein Gefühl dafür vermitteln, wo die verschiedenen Technologien pauschal eingeordnet werden können. Nicht berücksichtigt sind die Dünn- und Dickschichttechnik, die SMD-Technik (Surface Mounted Devices = oberflächenmontierbare Bauelemente) usw.

Mit der Zeile „diskreter Aufbau“ ist der Aufbau einer Schaltung auf der Basis von Full Custom ICs gemeint, jedoch nicht als ein einziger, integrierter Baustein. Der „diskrete Aufbau“ bildet das Pendant zum Full Custom IC. Beim „diskreten Aufbau“ fielen bisher die geringsten Entwicklungskosten an, aber die höchsten Stückkosten, was wiederum bedeutet, dass die Stückzahl nur klein sein kann, der Flächenbedarf dafür aber sehr groß ist, woraus auch hohe Laufzeiten resultieren. Die Ebene "Diskreter Aufbau" ist besonders gekennzeichnet, da sich das Bild inzwischen gewandelt hat. Im allgemeinen ist heute der Aufbau mit PLDs preisgünstiger geworden, da die dafür notwendigen Werkzeuge auch für PCs zur Verfügung stehen.

## Weitere Technologien

### Dünnschichttechnik:

*Hybridtechnik (bspw. durch Aufdampfen oder Aufstäuben)*

### Dickschichttechnik:

*Hybridtechnik (auf der Basis von Dickfilmen) nachträgliche Bearbeitung möglich*

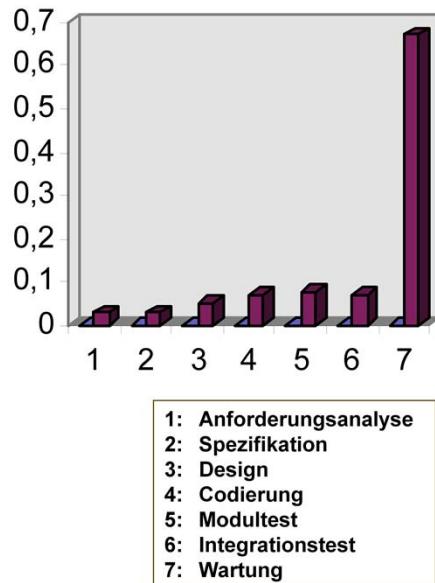
### SMD-Technik:

*Surface Mounted Devices*

**hier nicht von Interesse, da reine Produktionsprozesse**

## Kosten

Die Wartung wird über den Life Cycle am teuersten.



7: Wartung



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-1  
p. 16  
18.01.2013 13:45:01

Kapitel 4 behandelt den Entwurf komplexer Schaltungen. ASICs sind im allgemeinen komplexe Schaltungen, doch kommt bei dieser Technik für den Designer, der diskrete Schaltungen entwirft, ein erschwerendes Moment hinzu: Ist die Schaltung in einem Baustein einmal implementiert, stehen ihm im allgemeinen nicht mehr schaltungsinterne Messpunkte zur Verfügung. ASIC-Entwicklung bedeutet also nicht nur, eine entworfene Schaltung zu implementieren. Es heißt, sie muss so entworfen werden, dass sie mit größtmöglicher Sicherheit einwandfrei funktioniert und auch getestet werden kann. Die zusätzlichen Testschaltungen lassen aber das Design schnell noch komplexer werden, so dass verständlich wird, dass ohne "vernünftige" Werkzeuge heute nicht mehr gearbeitet werden kann. Will man sich also in die ASIC-Entwicklung ernsthaft einarbeiten, ist es am effizientesten, wenn man ein konkretes ASIC-Design entwickelt (Learning by doing) und dabei die theoretischen Grundkenntnisse anwendet.

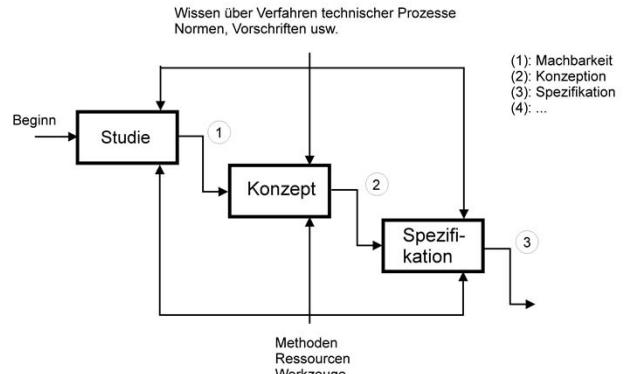
Wichtig, wie schon mehrfach erwähnt, ist die Vorgehensweise, möglichst nach der Topdown-Methode. Man spezifiziert die gewünschten Funktionen und arbeitet sich, von diesem Entwurf ausgehend, zu Lösungen vor, deren Hardware-Softwareaufteilung im allgemeinen von vorneherein nicht feststehen kann und auch nicht sollte. Forschungsziel der ASIC-Tool-Entwickler sollte deshalb sein, dass der Entwickler zukünftig ein Werkzeug für digitale Schaltungen in die Hände bekommt, durch das er sich zunehmend von Transistoren, Gattern, ja von der Hardware ganz lösen kann. Er sollte ausschließlich Funktionen spezifizieren, und das Tool sollte ihm dann vorschlagen, was in Hardware und was in Software zu entwickeln ist. Wir sind noch weit von diesem Ziel entfernt, doch lässt dieser Ausblick deutlich werden, wie wir heute schon vorzugehen haben.

Die gewählte Vorgehensweise ist so weit wie möglich zu modularisieren, zu spezifizieren und zu dokumentieren. Die Problematik der Softwarekrise zeigt uns die Notwendigkeit hierzu. Was in der Welt des ASIC-Entwurfs nicht passieren darf, ist in der Welt des Softwareentwurfs zur Tatsache geworden: Die Kostenverteilung im Software Life Cycle hat dramatische Ausmaße angenommen. Für die Wartung von Software rechnet man bis zu 70% der Gesamtkosten und mehr. Beim ASIC-Entwurf legt man deshalb die Tools so an, dass die gewonnenen Daten pro Arbeitsschritt weitgehendst auf ihre Richtigkeit und Konsistenz geprüft werden. Erst dann ist ein weiterer Arbeitsschritt akzeptabel. Auch wurde der Begriff **99%-Testbarkeit** in diesem Zusammenhang ein stehender Begriff: Bevor ein ASIC in Produktion geht, muss er, soweit möglich, getestet worden sein. Dazu gehört, dass jedes FF beim Test mindestens einmal in jedem Zustand stabil gewesen sein muss, und dazu gehört auch, dass alle Schaltnetze vollständig geprüft wurden bis auf die Multiplexer, die zwischen „Test“ und „Betrieb“ umschalten.

## → Forward Engineering →

Analyse		Entwurf	Lösungs-realisierung
Anforderungs-definition	Funktions-definition	Systementwurf	Schaltungs-realisierung

## Entwurfsmethoden



7:

Wartung



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-1

p. 17  
18.01.2013 13:45:01

Legt man das Forward Engineering zugrunde, reicht die grobe Differenzierung, wie sie in Bild oben links dargelegt ist, nicht aus. Anforderungen, funktionale Spezifikation usw. sind lediglich Topics, die weiter zu gliedern sind. Hier sei beispielsweise auf das Wasserfallmodell (Waterfall Model) oder auf ähnliche Modelle verwiesen, wie sie beispielsweise in /Klö95/ behandelt werden.

Wird dabei auch eine ausführliche Dokumentation angelegt, worauf in Kapitel 1 schon verwiesen wurde, ist nicht nur die Wartbarkeit hinsichtlich eines kostengünstigen Redesigns garantiert, sondern auch die Wiederverwendbarkeit in weiteren Projekten, indem entsprechende Bibliotheksobjekte archiviert werden, die, von verschiedenen Entwurfsphasen ausgehend, einfach zu modifizieren sind. Allein die laufend verbesserten ASIC-Technologien lassen dies notwendig werden, da ein für eine spezielle Technologie „eingefrorenes“ Objekt schon für eine nachfolgende Technologie unbrauchbar sein kann. Ebenso sind „Basteleien“ unverantwortlich, die sich vielleicht zufällig in der momentanen Simulation als ausreichend ergeben. Ein bekanntes Beispiel hierfür ist die Integration asynchroner Rückkopplungen in synchronen Schaltungen (siehe hierzu auch Kapitel 3). Zitat einer Fachzeitschrift:

„Just say NO to asynchronous Design: Synchronous designs are safer than asynchronous designs, more predictable, easier to simulate and to debug. Asynchronous design methods may **ruin your project, your career and your health**, but some designers still insist on creating that seemingly simple, fast little asynchronous circuit.“

Twenty years ago, TTL-MSI circuits made synchronous design attractive and affordable, fifteen years ago, synchronous microprocessors took over many hardware designs; more recently, synchronous State Machines have become very popular, but some designers still feel the itch to play asynchronous tricks.

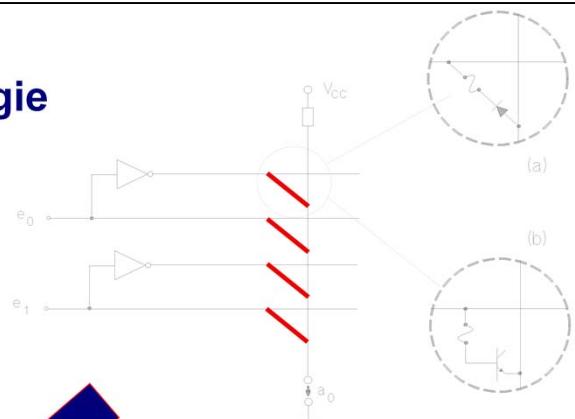
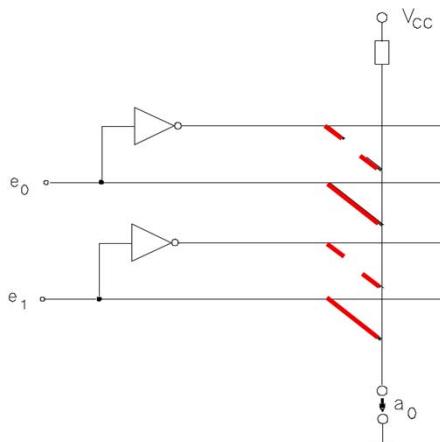
The recent popularity of ASICs has created a new flurry of asynchronous designs in a specially treacherous environment: Gate Arrays and Programmable Gate Arrays are being customized at the gate level, and may tempt the designer to develop bad asynchronous habits, especially dangerous since it is very difficult to inspect internal nodes, and impossible to calm them down with capacitive load, the Band-Aid of simpler technologies.

## **4 entscheidende Technologien bei programmierbaren Bausteinen:**

- 1. Fuse-Technologie**
- 2. Antifuse-Technologie**
- 3. Floating-Gate-Technologie**
- 4. SRAM-Technologie**

# 1. Fuse-Technologie

- ❖ älteste Technologie
- ❖ Dioden / Transistoren
- ❖ Programmierung über Stromstöße



Realisierung der Funktion:

$$a_0 = e_1 \text{ UND } e_0$$

benötigt  
relativ viel  
Chip-Fläche  
-  
irreversible  
Prog-  
mierung

ale Integrierte Schaltungen  
DIS

K5-1

p. 19

18.01.2013 13:45:01

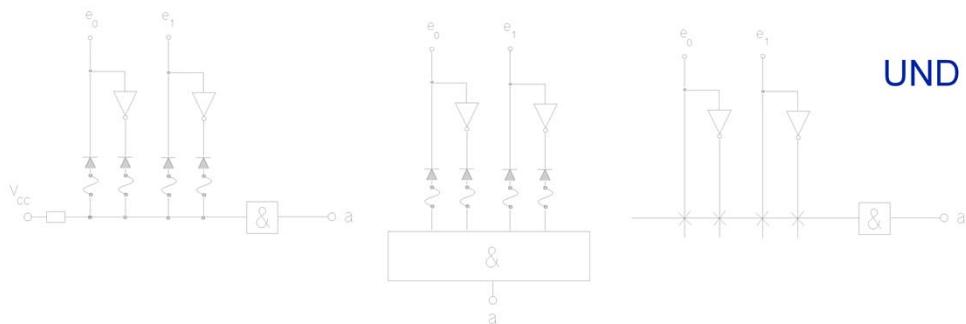
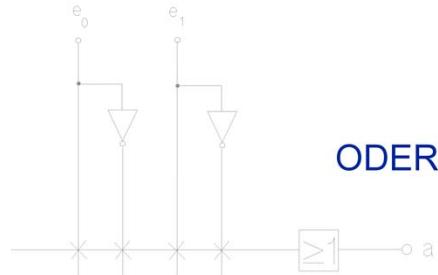
Die **Fuse-Technologie** ist die älteste, bei der Zuleitungen von Dioden oder Transistoren, die nur wenige Mikrometer stark sind, als Verbindungspunkte verwendet werden, die durch Stromstöße derart zerstört werden können, dass sie keine elektrische Verbindung mehr zwischen den Polen aufweisen. Die Stromstöße werden durch zeitlich definierte Überspannungen erzeugt. Da derartige Vorgänge irreversibel sind, sind die Bausteine nur einmal programmierbar. Der Vorteil dieser Technologie ist allerdings auch, dass die Bausteine ihre „Speicherfähigkeit“ mit einer großen Wahrscheinlichkeit behalten, ihre Zuverlässigkeit somit als hoch eingestuft werden kann.

## ODER-Funktion

Um die Funktion  $a = e_1 \text{ ODER } e_0$  entsprechend oben zu realisieren, ist eine Programmierung wie in Bild unten notwendig. In der Praxis wird dies jedoch einfacher dargestellt, um die meist umfangreichen Matrizen relativ einfach und übersichtlich darstellen zu können.

# 1. Fuse-Technologie

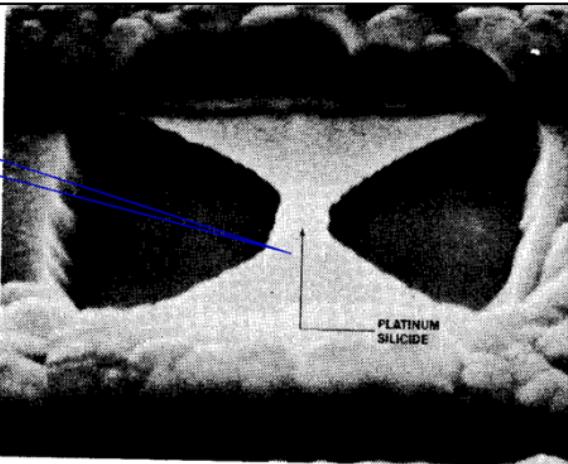
andere Darstellungsformen



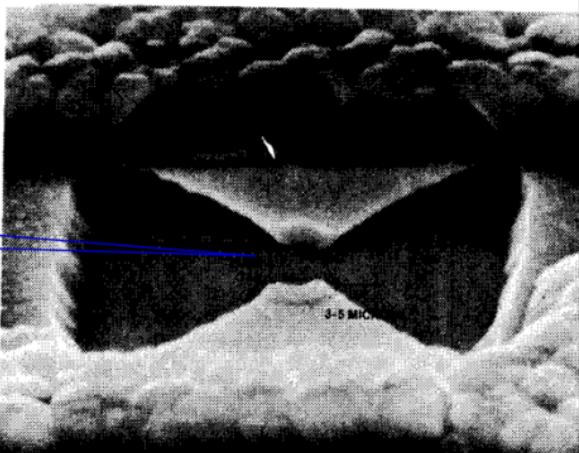
Es werden unterschiedliche Darstellungsformen der gleichen Schaltung verwendet, wobei im allgemeinen die rechte Form oben und unten verwendet wird.

Die Fuse-Technologie tritt noch in weiteren Variationen auf. So beinhaltet die als *laterale* Implementation zu bezeichnende Technologie den großen Nachteil, dass sie viel Chipfläche benötigt. Demgegenüber gibt es Hersteller, die zwei Dioden, die gegeneinander geschaltet sind, vertikal anordnen und somit eine höhere Speicherdichte erreichen. Bei der Programmierung wird eine der beiden Dioden „weggebrannt“.

unprogrammiert

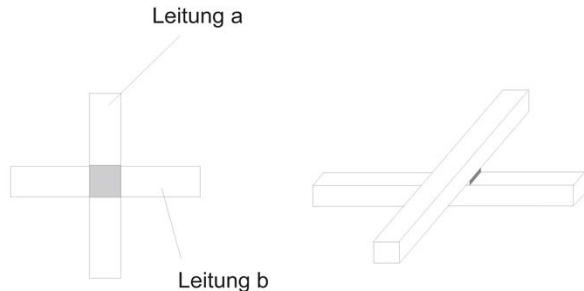


programmiert:  
offen



## 2. Antifuse-Technologie

- 2 Leiterbahnen über einer hauchdünnen Isolierschicht getrennt
- Bahnen sind Metall
- Kreuzungspunkt: Si, ..
- Programmierung ebenfalls irreversibel
- sehr hohe Speichererdichte

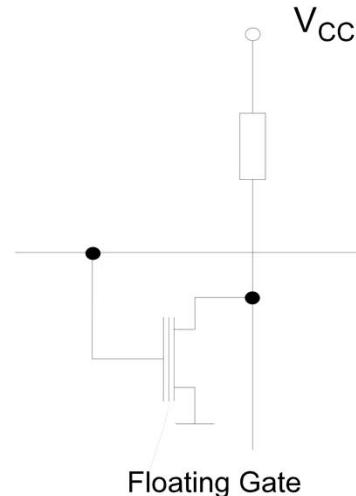


Der Fuse-Technologie verwandt ist die **Antifuse-Technologie**. Zwei Leiterbahnen werden durch eine hauchdünne Isolierschicht (beispielsweise eine Oxid-Nitrid-Oxid-Schicht: ONO) elektrisch voneinander getrennt. Sie kann durch eine Überspannung durchbrochen werden, was nicht immer ganz einfach gelingt. Aus diesem Grund ist es bei derartigen Bausteinen sinnvoll, dem Programmervorgang einen vollständigen Testvorgang anzuschließen. Selbstverständlich ist, dass die Programmervorgänge auch bei dieser Technologie irreversibel sind.

Die Leitungen bei der Antifuse-Technologie werden matrixförmig angeordnet, wobei die Matrix durch sogenannte Pass-Transistoren in Segmenten angeordnet wird, was eine einfache Adressierung erlaubt. Vorteil der Antifuse-Technologie gegenüber der Fuse-Technologie ist die zu erzielende höhere Speicherelementendichte.

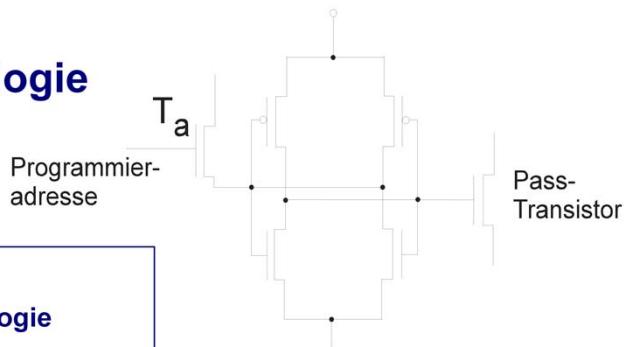
### 3. Floating-Gate-Technologie

- mehrmals programmierbar  
*da steuerbarer MOS-Transistor mit Floating Gate*
- Variante 1:  
EPLD (erasable PLD)  
*löschbar über ultraviolettes Licht*
- Variante 2:  
EEPLD (electrical EPLD)



Der schwerwiegende Nachteil der Fuse-Technologie, die nur einmalige Programmierung, wird bei der **Floating-Gate-Technologie** vermieden. Die ersten Bausteine dieser Technologie, die auf den Markt kamen, waren die durch ultraviolettes Licht löschenbaren **EPLDs**: Erasable PLDs; später kamen die **EEPLDs**: Electrical EPLDs hinzu. Die programmierbare Zelle enthält einen MOS-Transistor, der über eine Steuerelektrode (Gate) schaltbar ist. Vor der Steuerelektrode ist ein *Floating Gate* angeordnet, das im geladenen Zustand eine Anhebung der Schwellspannung des Transistors bewirkt (Verschiebung der  $i_D$ - $U_{GS}$ -Kennlinie), so dass ein Schalten des Transistors im üblichen Arbeitsbereich verhindert wird. Bei den EPLDs kann nun durch ultraviolettes Licht die Ladung wieder abgebaut werden, was ein entsprechendes Sichtfenster im IC und dessen Ausbau voraussetzt.

## 4. SRAM-Technologie

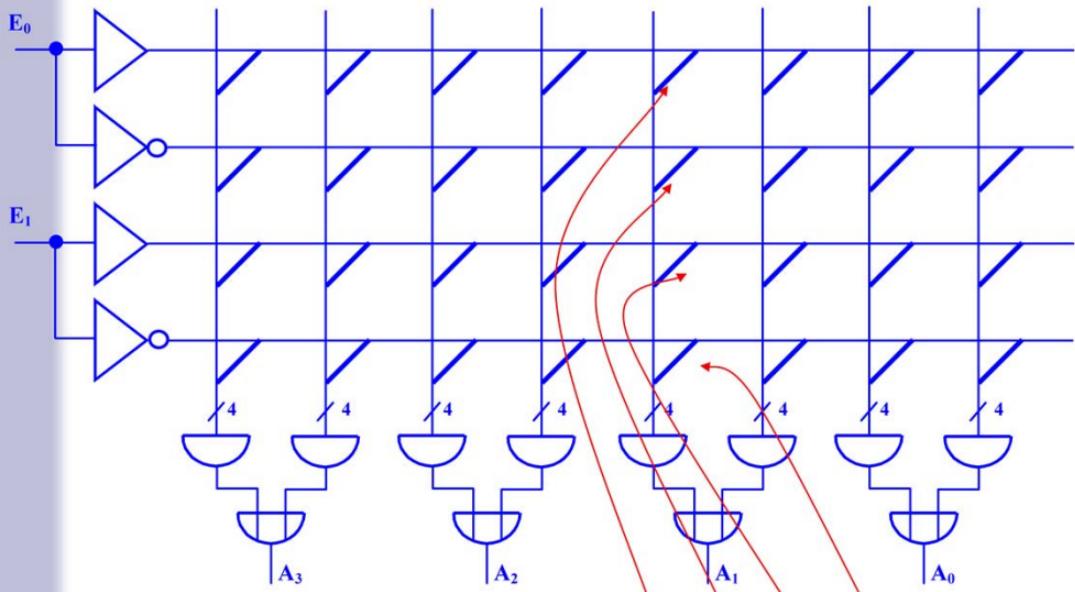


- **Static RAM**
- **aufwendigste Technologie  
(6 Transistoren)**
- **Basis: CMOS-Speicherzellen**
- **keine "echte" Programmierung**
- **Ladung über State Machine**
- **$T_a$ : Programmieradresse**
- **Ausgang: Pass-Transistor**
- **sehr flexibel**

Sehr flexibel einsetzbar sind Bausteine, die auf der **SRAM-Technologie** (Static RAM) basieren. Zugrunde gelegt sind statische CMOS-Speicherstellen, die somit flüchtig sind. Die „Programmierung“ muss also nach Abschaltung der Versorgungsspannung jeweils neu vorgenommen werden und erfolgt üblicherweise über ROMs und eine entsprechende State Machine. Der Aufwand einer Zelle ist relativ groß. Eine Zelle besteht im allgemeinen aus sechs Transistoren. Das zentrale Element ist ein FF, das durch zwei rückgekoppelte Inverter gebildet wird. Der Transistor  $T_a$  dient zum Setzen und Rücksetzen der Zelle und verfügt deshalb über eine Adress- und eine Datenleitung. Der Ausgang der Zelle ist der Pass-Transistor, der von der Zelle angesteuert wird.

Die SRAM-Technologie ist deshalb so bedeutend geworden, da heute relativ komplexe Bausteine realisiert werden können, in denen man „eben mal schnell“ seine Schaltung bis hin zu Mikroprozessoren testen kann. Lösch- und Ladevorgang erfolgen in Zeitspannen, die nicht viel größer als die Compilier-Durchlaufzeiten selbst sind.

# Grundidee der ersten Bausteine



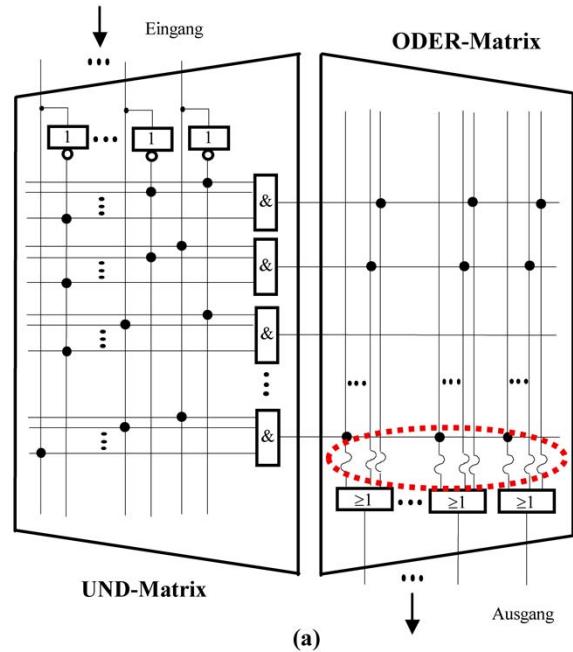
= direkte Umsetzung der  
Minterm-Maxterm-Struktur }

$$A_i = (E_0 \wedge \bar{E}_0 \wedge E_1 \wedge \bar{E}_1)$$

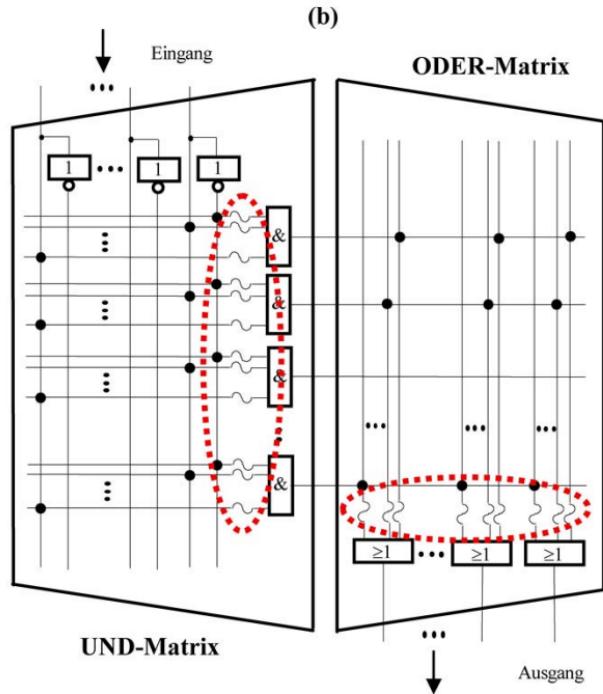
# EPROM

Vergleiche:  
Schaltnetz nach  
Minimierung:

Umsetzung der  
Minterm-Maxterm-  
Struktur (DN)

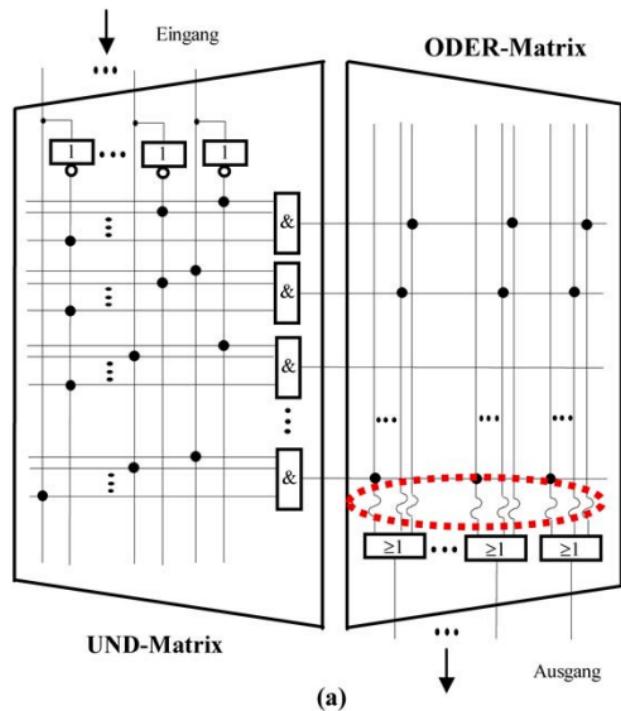
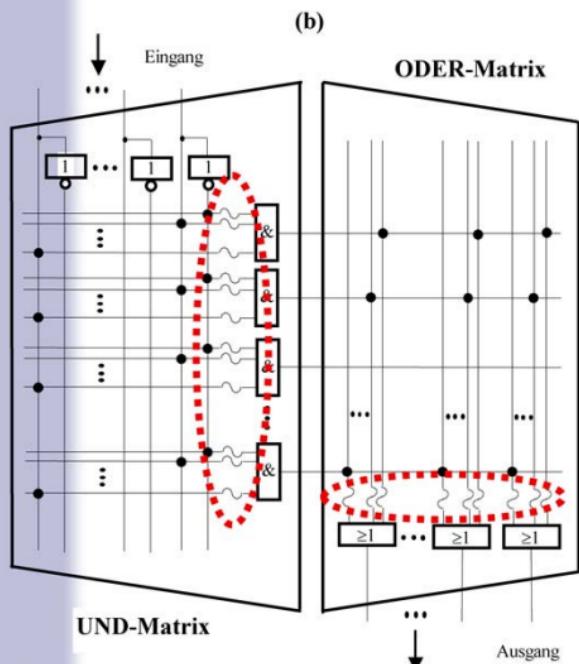


# PLD (Programmable Logic Device)

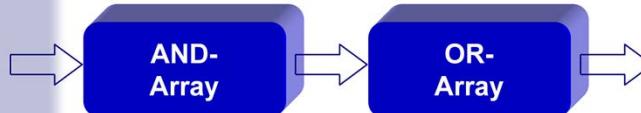


Es ist der PLA (Programmable Logic Array). Wie die Darstellungen zeigen, liegt der Idee der ersten PLDs die DN-Struktur zugrunde, die auch als Decoder-Coder-Einheit dargestellt werden kann, was Bild oben vermitteln soll.

# PLD + EPROM



## 1. Generation

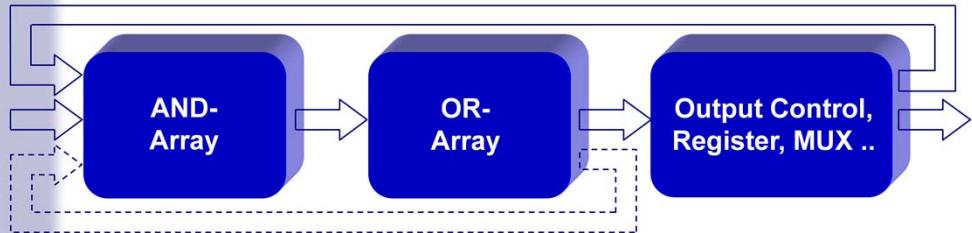


# Historie der PLDs

## Zwischenschritt



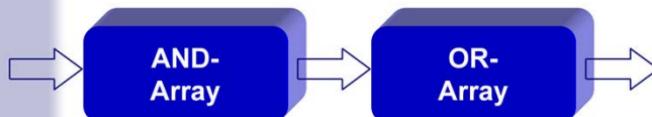
## 2. Generation



Der Weg der immer komplexer werdenden Strukturen der programmierbaren Bausteine ist damit einfach nachvollziehbar. Von der UND-ODER-Struktur ausgehend, fügte man an den Ausgang *Output Control Register* sowie Multiplexer an und integrierte Rückkopplungen. Man erreichte damit die bekannte Mealy-Struktur nach Kapitel 3. Die untere Rückkopplung der unteren Darstellung ist dabei eine asynchrone Rückkopplung und somit mit Vorsicht zu verwenden (siehe hierzu Kapitel 3).

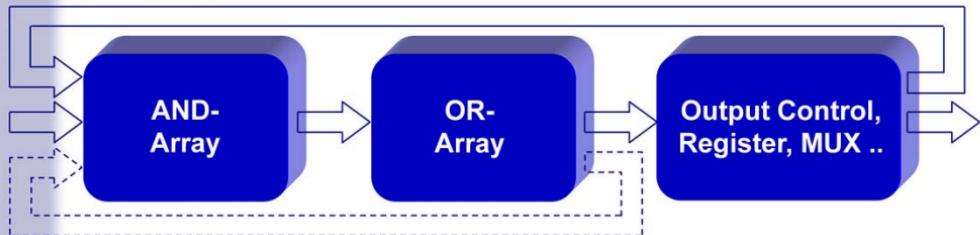
Um nun dem Anwender einen möglichst effizienten Baustein für die diversen Schaltungen bieten zu können, werden zahlreiche Variationen hergestellt, auf die im nächsten Abschnitt noch näher eingegangen wird. Doch wie ging es nach der Entwicklung der ersten Bausteine dieser Art weiter? Die Matrizen beliebig zu vergrößern hatte keinen Sinn, da die Beschreibung einer Schaltung auf der Basis eines Mealy-Automaten sich zwar für gewisse Steuerwerke wie das des klassischen Mikroprozessors eignet, doch beispielsweise beim Operationswerk des Mikroprozessors im allgemeinen versagt. So werden beim Operationswerk eines Mikroprozessors viele Register verlangt, dafür aber relativ wenige Schaltnetzfunktionen, wenn man von Multiplexern absieht.

## 1. Generation



# Historie der PLDs

## 2. Generation



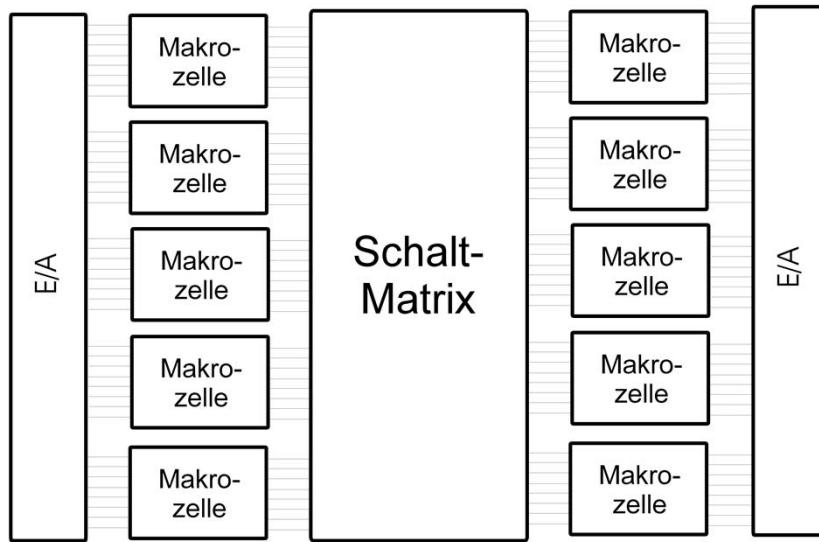
= Mealy- / Moore-Automat

heute: zahlreiche Variationen, auch mit Matrix mit  
Mealy-Zellen: LCA

2. Generation: die zweite Rückkopplung ist natürlich für den Mealy-Automaten nicht notwendig

LCA: Logic Cell Array

# Ein modernes Prinzip



**Ziel:** hoher Ausnutzungsgrad = genutzte Fläche/Gesamtfläche



Will man also ein Steuerwerk auf der Basis eines Mealy-Automaten bauen, ist nicht jede PAL-Architektur zu empfehlen, da ansonsten der *Ausnutzungsgrad* des Bausteins denkbar schlecht ist.

Das bedeutet, man muss Strukturen finden, die möglichst vielen Schaltungen optimal angepasst sind. So entwickelte man Bausteine, in denen zentral eine Matrix angeordnet ist, die mit Makrozellen umgeben ist, und um die wiederum Ein- und Ausgabebausteine angeordnet wurden. Die Makrozellen enthalten oft nur wenige Bausteine, wie beispielsweise ein, zwei oder drei FFs, ein paar Multiplexer und UND- und ODER-Glieder. Sie können aber auch aus kleineren Produkttermen (UND-ODER-Matrizen) bestehen, an die über Multiplexer Register angeschlossen sind. Oder es ist zentral wiederum eine Schaltmatrix integriert, um sie herum aber größere Blöcke, die jeweils ein UND-ODER-Array, eine Ein-, Ausgabeeinheit sowie bestimmte Makrozellen beinhalten. Gemeinsam haben sie, dass ihnen Makrozellen zugrunde gelegt werden, die entsprechend programmiert und verknüpft werden können.

Der Nachteil dieser Bausteine wird ebenfalls sichtbar: Ist die zu entwickelnde Schaltung nicht optimal an den verwendeten PLD anzupassen, kann der Ausnutzungsgrad rasch auf einen niedrigen Wert sinken. Nicht selten kommt es dann vor, dass dieser in Größenordnungen von 0,4 liegt.

Neben diesem Typ nach Bild oben entstanden die Familien der *Programmierbaren Gate-Arrays*.

# Differenzierter

PLDs, basierend auf Makrozellen			Programmierbare Gate Arrays		
programmierbar: UND fest: ODER	programmierbar: UND + ODER	CPLDs mehrere Arrays		kanalstrukturierte Gate Arrays	Sea of Gates
PAL, GAL, EPLD, ..	PLA, PLS, ..	MAX, MACH, PLUS LOGIC, ..	LCA	FPGA	ERA
Produktterme, zahlreiche Variationen, typisch auch für Mealy- Anwendungen		SRAM	Antifuse- Technologie, nichtflüchtig	SRAMs auf NANDS aufbauend	

Typische Vertreter der Gate Arrays sind die Bausteine auf der Basis von *Sea of Gates*, Bausteine mit konfigurierbaren Logikblöcken und Gate Arrays mit Kanalstruktur. Es wird im einzelnen darauf noch eingegangen.

Programmierbare Gate Arrays besitzen als Basiselemente einfachste Zellen wie NANDs (beispielsweise beim ERA-Typ), die entsprechend zu programmieren sind. In der Literatur wird als Nachteil oft angegeben, dass durch die Granularität der Zellen der Compiler-Aufwand im Vergleich zu den auf Makrozellen basierenden PLDs sehr hoch ist. Dies ist logisch, da eine feine Granularität eine hohe Flexibilität erlaubt, die wiederum mit einem erhöhten Entwicklungsaufwand bezahlt werden muss. Da man aber heute entsprechende Maschinen zum Compilieren zur Verfügung hat (jeder etwas bessere PC reicht heute dazu aus), ist diese Negativwertung nicht mehr stichhaltig. Dass der Ausnutzungsgrad durch die feine Granularität sehr hoch ist, versteht sich von selbst.

# Beispiele:

- **PAL:** Programmable Array Logic
  - UND-Matrix: programmierbar
  - ODER-Matrix: fest
- **FPLA:** Field Programmable Logic Array
  - UND-Matrix: programmierbar
  - ODER-Matrix: programmierbar
- **PROM:** Programmable Read Only Memory
  - UND-Matrix: fest: Adressdecoder
  - ODER-Matrix: programmierbar
- **HAL:** Hardware Array Logic
  - programmierbare PALs
- **EPLD:** Eraseable Programmable Logic Device
  - PAL, nur mit Floating-Gate-Technologie  
(Fuse programming: EPROM-Zellen)
  - mit UV-Licht löschen

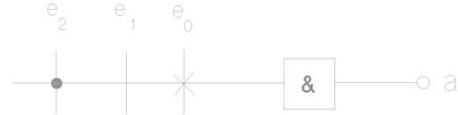
# Beispiele:

- AGA: Alterable Gate Arrays
  - RAM-Zellen + Logik + Steuerlogik
- IFL: Integrated Fuse Logic
  - Fuse Link (Schmelzpfade)
  - FPGA, FPLA, FPLS, FPRP
- FPGA: Field Programmable Gate Array
  - UND-Matrix: programmierbar
  - Gate Array: programmierbar
- FPLS: Field Programmable Logic Srquencer
  - FPGA mit programmierbaren Registerfunktionen
- FPRP: Field Programmable ROM Patch
  - UND-Matrix: fest (Adressdecoder)
  - ODER-Matrix: programmierbar

# Beispiele:

- LCA: Logic Cell Array
  - CMOS static RAM cells
  - Inhalt muss jeweils geladen werden
- GAL: Generic Array Logic
  - PAL, aber wiederprogrammierbar, elektrisch lösbar
- PLA: Programmable Logic Array
  - 1979 (der erste)
- MAX:
  - NAND-NAND structure
  - jeder Funktion sind 3 Produktterme zugewiesen
- MACH: Macro Arrays CMOS High-density
  - PAL-Block, Schaltermatrix

# Definitionen



$$A = e_2 \wedge e_0$$

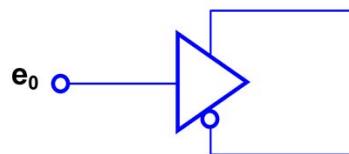
## Product Lines: UND-Eingänge

- **3 Verbindungstypen:**
  - Punkt: hard wire connection
  - nichts: blown fuse (keine Verbindung)
  - Kreuz: intact fuse (Verbindung)
- **Texas Instruments:**

*“When all the fuses are blown on a product line, the output of the AND gate will always be a logic 1. This has the effect of locking up the output of the OR gate to a logic level 1.”*

Die verschiedenen Architekturen sind eng verknüpft mit ihrer Geschichte, was kurz darzulegen ist. Doch zuvor noch ein paar Worte zu den *Basic Symbologies*, wie sie beispielsweise in den Texas-Datenbüchern zu finden sind. Wie schon oben gezeigt, werden die Eingangsleitungen der UND-Glieder (= *Product Lines*) als eine Leitung gezeichnet. Man unterscheidet drei verschiedene Verbindungstypen: ein Punkt symbolisiert eine feste Verbindung (dot: *hard Wire Connection*), kein Zeichen keine Verbindung (*blown Fuse*), und ein X bedeutet eine intakte Verbindung (*intact Fuse*). Dabei ist zu beachten: „*When all the fuses are blown on a product line, the output of the AND gate will always be a logic 1. This has the effect of locking up the output of the OR gate to a logic level 1.*“

# Definitionen



## Definition in der Digitaltechnik:

- oben: nicht negierter Ausgang
- unten: negierter Ausgang

**Vorsicht:**

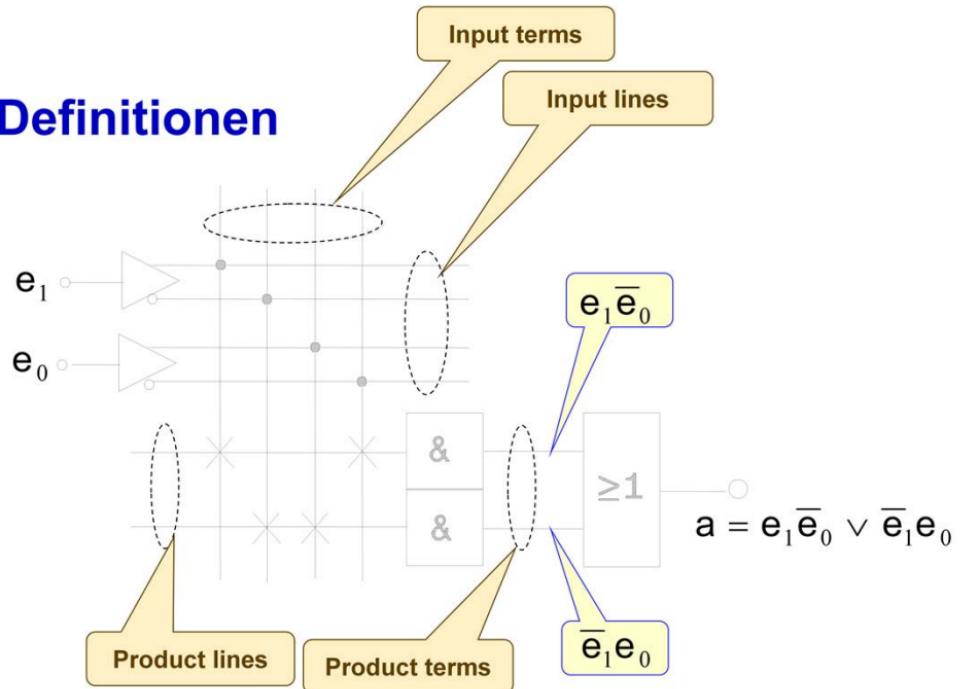
*in der Analogtechnik eine andere Definition*

Das Eingangsverstärkersymbol besitzt zwei Ausgänge, einen nichtinvertierenden und einen invertierenden. Dieses Symbol ist entliehen aus der analogen Technik, wo es eine entsprechende Bedeutung hat. Während in der digitalen Technik stets einfache Verhältnisse vorliegen, ist es in der Analogtechnik doch etwas komplizierter.

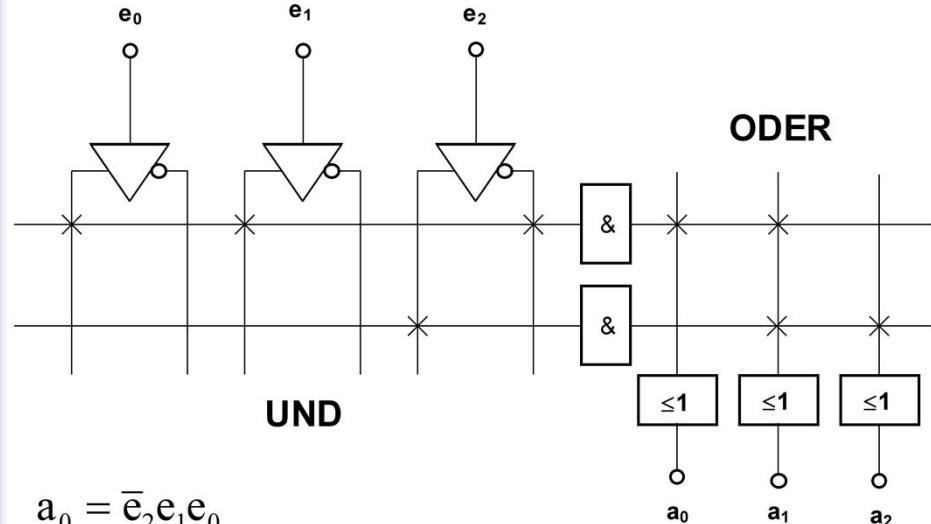
In der Digitaltechnik nimmt der invertierte Ausgang den logischen Wert "0" an, wenn der nichtinvertierende den logischen Wert "1" besitzt und umgekehrt. Bezüglich der Spannungspotentiale gilt Entsprechendes.

In der Analogtechnik sind nicht nur zwei Potentialzustände definiert, sondern unendlich viele, so dass hier die Definition erweitert werden muss. Der negierte Ausgang liefert den entgegengesetzten Spannungswert, von einem Mittelwert zwischen den beiden Ausgängen ausgehend. Arbeitet man beispielsweise zwischen +15 V und -15 V, liegt der Mittelwert bei 0 V. Liefert dann der nichtnegierte Ausgang +3 V, steht am negierten Ausgang -3 V. Arbeitet man dagegen zwischen +15 V und -5 V, liegt der Mittelwert bei +5 V, bei einem Ausgangshub von 3 V, die Ausgangswerte liegen bei +8 V und +2 V. Eingangsseitig wird dann im allgemeinen das Differenzsignal gemessen, um Störungen eliminieren zu können (siehe Vorlesung *Feldbusssysteme*).

# Definitionen



Weiterhin sind definiert: *Input Terms*, die *Product Terms* sowie die *Sum of Products*. Diese Definitionen spiegeln direkt die ursprüngliche Idee der programmierbaren Bausteine wider (zu denen auch das ROM zählt). Sie beruht auf der Minterm-Maxterm-Struktur einer DN (siehe hierzu auch Kapitel 2 des Skriptes).



$$a_0 = \bar{e}_2 e_1 e_0$$

$$a_1 = \bar{e}_2 e_1 e_0 \vee e_2$$

$$a_2 = e_2$$

### Beispiel PAL

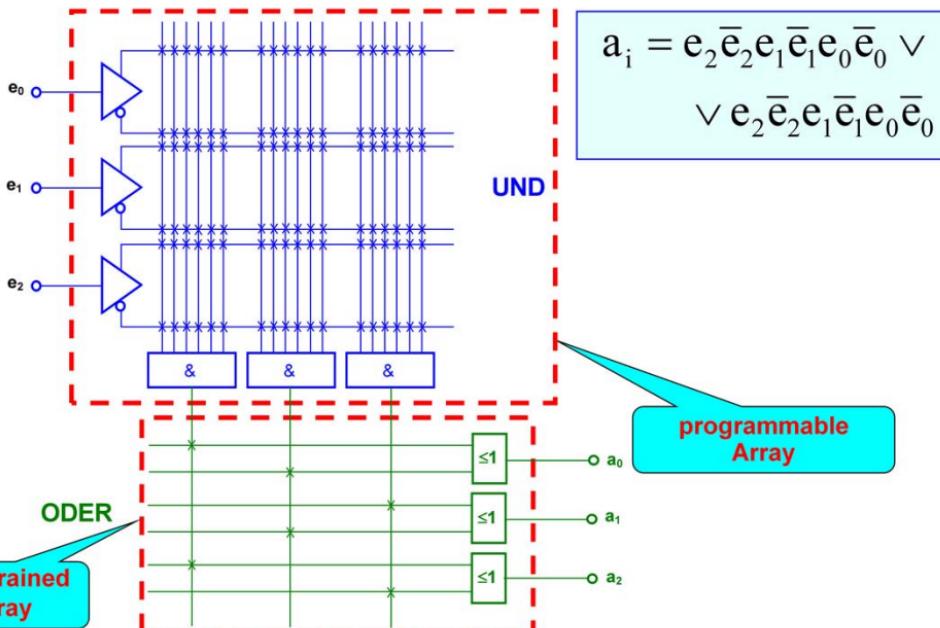
als erster programmierbarer  
Logikbaustein

## PAL und PLA

PAL (Programmable Array Logic) ist ein eingetragenes Warenzeichen der Firma *Monolithic Memories* und somit ein *Brand Name*. Den PAL kann man als ersten frei programmierbaren Baustein bezeichnen, wenn man einmal vom ROM absieht. Wie die vorhergehenden Darstellungen schon vermitteln, liegt der Idee der ersten PLDs die DN-Struktur zugrunde, die auch als Decoder-Coder-Einheit dargestellt werden kann, was auch die folgenden Bilder vermitteln sollen.

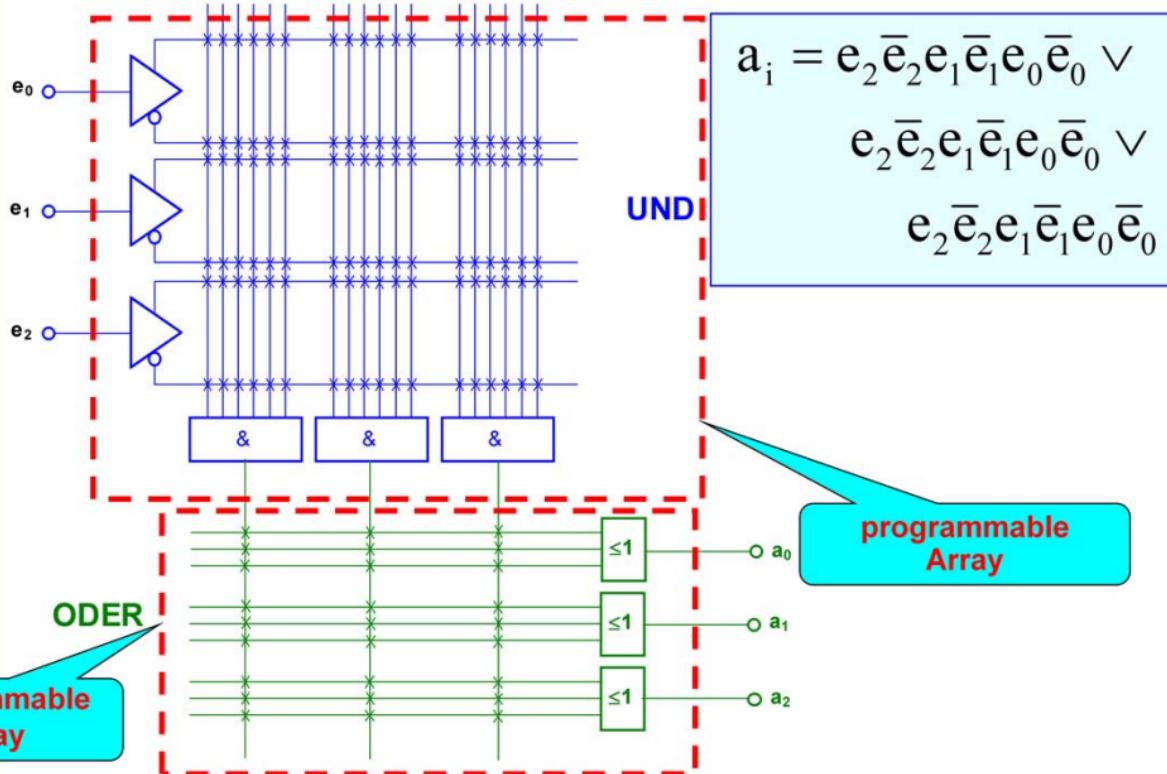
1. realisierter Baustein:

## PAL: Programmable Array Logic

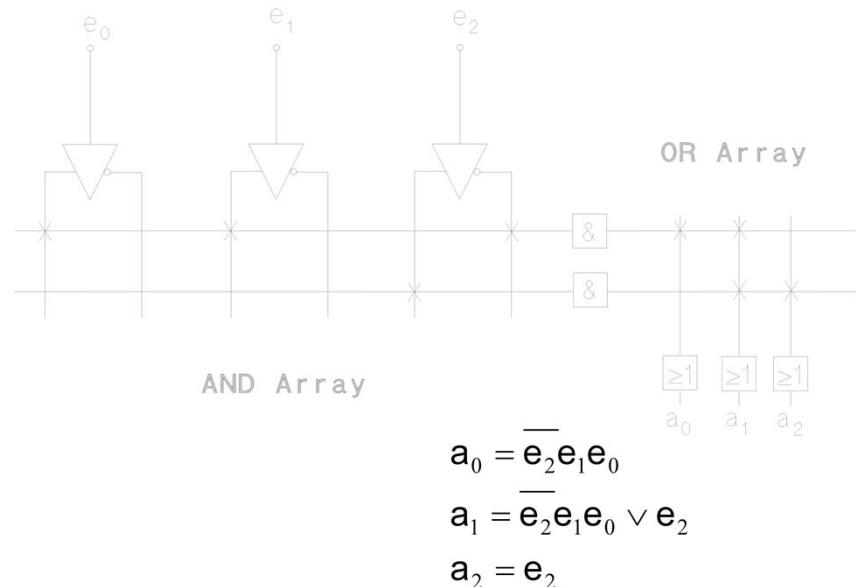


In dieser und der folgenden Folie sind zum besseren Verständnis die beiden Prinzipien von PAL und PLA schaltungstechnisch vereinfacht wiedergegeben. Man erkennt direkt, dass der PAL die einfache Variante darstellt, aber auch später als das ROM auf den Markt kommen musste.

# PLA: Programmable Logical Array



# PLA: Programmable Logical Array



Oben ist die typische Darstellungsweise eines PLAs aus Datenbüchern wiedergegeben. Wie erkennbar, liegt eine  $6 * 2$ -UND-Matrix und eine  $2 * 3$ -ODER-Matrix vor. Die Definition der Bezeichnung [1] lautet jedoch, da in der UND-Matrix die negierten Linien nicht gezählt werden:

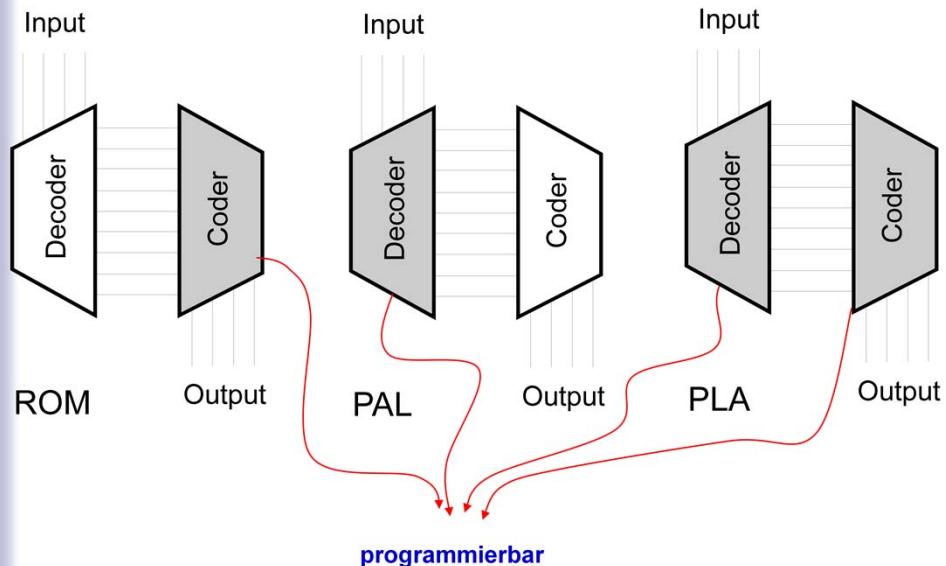
$3 * 2 * 3$ -PLD .

Ist die ODER-Matrix nicht programmierbar, handelt es sich also um ein PAL, lautet die Definition  $3 * (2 * 3)$ -PLD.

Auf weitere Einzelheiten der Programmierung soll hier nicht eingegangen werden, diese können Datenbüchern der Firmen Texas Instruments, Advanced Micro Devices (AMD), Lattice Semiconductor usw. entnommen werden. Eingegangen werden soll dagegen noch auf verschiedene Variationen dieser Bausteintypen. Die Darstellungen im folgenden sind Datenbüchern wie den Firmen AMD oder Texas Instruments entnommen

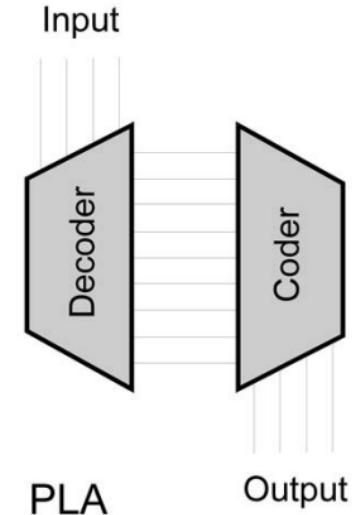
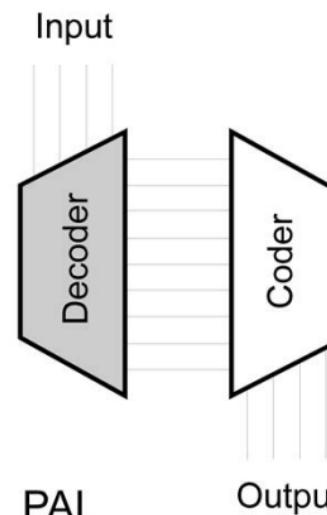
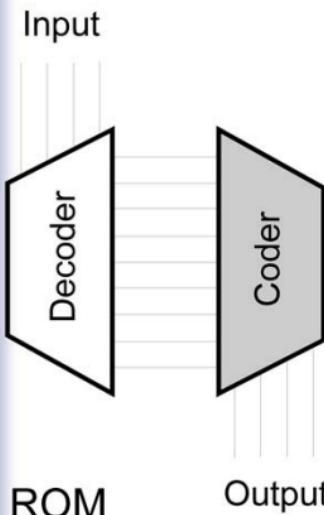
[1] Vorsicht: Diese Definitionen werden in Datenbüchern zum Teil unterschiedlich gehandhabt.

## Programmierbarkeit vom ROM, PAL und PLA



Im Bild oben werden die drei unterschiedlichen Typen bezüglich ihrer Programmierbarkeit gegenübergestellt. Im Unterschied zum PLA (Programmable Logic Array) ist beim PAL nur der Decoder programmierbar, und im Coder werden die Produktterme über ODER-Gatter zusammengefasst; beim PLA dagegen sind Decoder und Coder programmierbar, was selbstverständlich eine höhere Flexibilität voraussetzt, jedoch auch leistungsfähigere Compiler (wie schon erwähnt).

# Programmierbarkeit vom ROM, PAL und PLA



- **einfach zu handhaben**
- **Tools preisgünstig**

- **mehr Freiheitsgrade**
- **hohe Flexibilität**
- **Tools teurer**

## 2. Generation

**GAL**  
Generic Array Logic

- CMOS
- elektrisch löschbar
- 100 - 1000 erase- and write-Zyklen
- Datenhaltung 20 Jahre
- Built-in Security Circuit (Copy Protection)

- elektrisch löschbar
- ..

**EPLD**  
electrical PLD



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

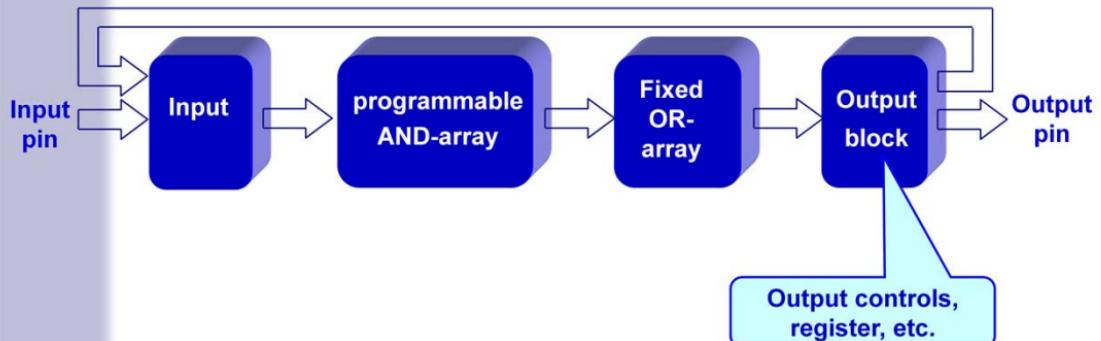
K5-1  
p. 45  
18.01.2013 13:45:01

Nach der Einführung der PALs kam konsequenterweise der Schritt, Bausteine zu entwickeln, die mehrfach programmierbar waren. Die ersten, die diese Forderung erfüllten, waren die EPLDs (Erasable PLDs) und die GALs (Generic Array Logic). In EPLDs wird anstelle von Fuses eine Floating-Gate-Technologie (entsprechend der EPROM-Technologie) verwendet. Durch UV-Licht (ca. 20 Minuten) kann nun die gesamte Programmierung rückgängig gemacht werden. Der Nachteil liegt auf der Hand: die Bausteine oder zumindest die Baugruppe müssen hierfür im allgemeinen ausgebaut werden.

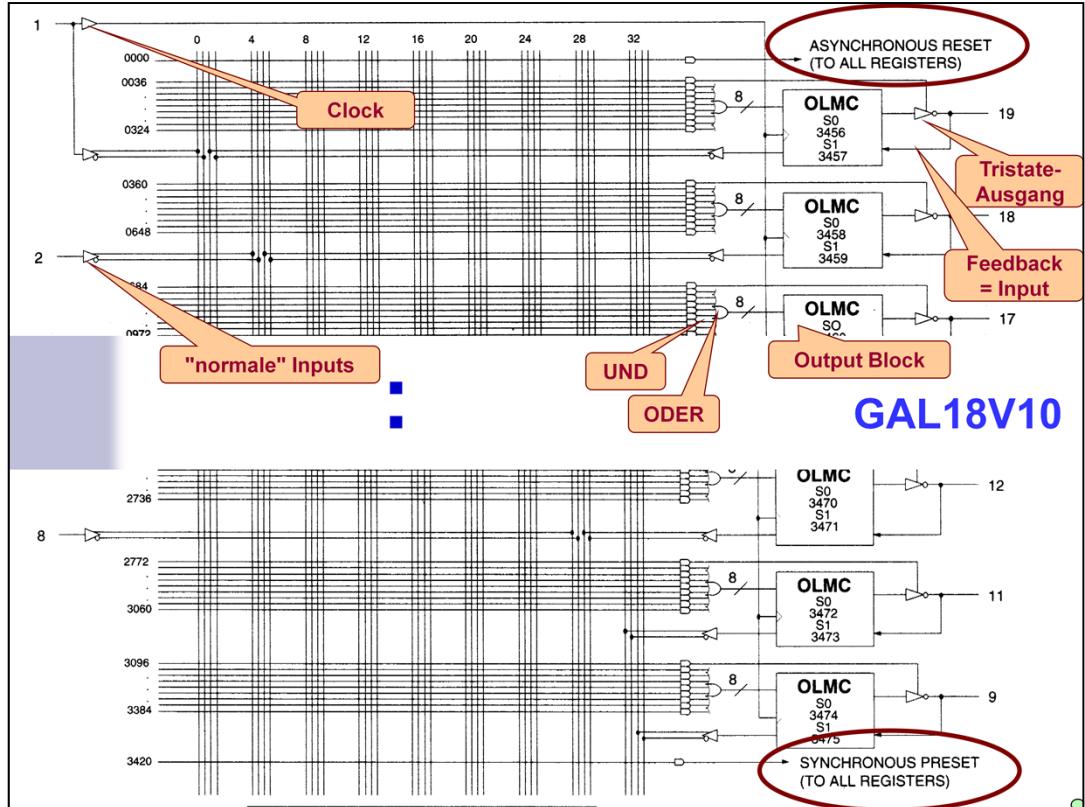
Im GAL der Firma Lattice (sie nennen es auch EEPLD: Electrically EPLD) setzt man nun für die Programmierung die Technik der EEPROMs ein, um sie elektrisch löschen zu können. Zu erhalten sind CMOS-Ausführungen für den Low-Power- oder für den High-Speed-Bereich. Interessant ist auch der integrierte *Built-in Security Circuit* für ein Copy Protection.

Wichtig ist zu wissen, dass die Anzahl der Löschzyklen sich auf die Lebensdauer der Bausteine auswirkt. Dabei wird nicht die Anzahl der möglichen Programmierungen reduziert, sondern die Haltbarkeitsdauer der Bausteine. Bei hundert und mehr Programmierzyklen verkürzt sich die Zeitspanne, die der Baustein die Daten "garantiert" hält, schnell um die Hälfte, die bei GALs üblicherweise 20 Jahre beträgt.

## Prinzipielle Struktur

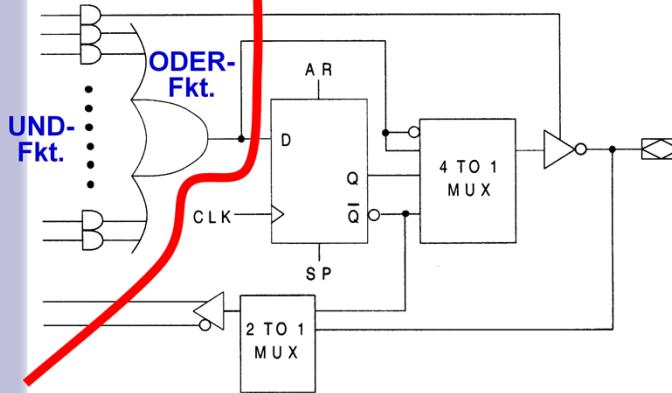


Die internen Strukturen von EPLDs, GALs und PALs sind gleich (siehe Bild oben), nur die Anzahl der Varianten von EPLDs und GALs ist gegenüber der der PALs bescheiden. Die beiden bekanntesten (und auch ältesten) GAL-Typen sind der 16V8 und 20V8, auf die hier kurz eingegangen werden soll.



GAL18V10

Vergleicht man die prinzipielle Darstellung der Ansicht vorher mit der Schaltung des Bausteins 18V10 oben, so ist zu erkennen, dass er keine Input-Latches aufweist, sondern nur ein programmierbares AND Array, ein nichtprogrammierbares OR Array, eine programmierbare Rückkopplung sowie die Ausgangszellen (OLMC: Output Logic Macro Cell). Diese OLMCs ermöglichen eine enorme Flexibilität, da sie unterschiedliche Schaltungsprinzipien durch ihre Programmierung zulassen. Die wahre Leistungsfähigkeit des Bausteins erkennt man also erst, wenn man sich näher mit der OLMC beschäftigt.



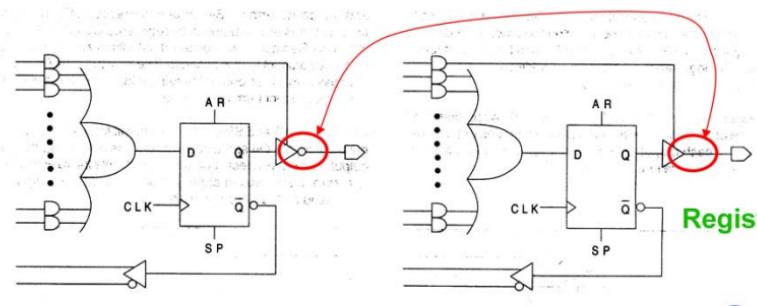
## OLMC: Output Logic Macrocell

- programmierbare Datenpfade
- Rückkopplung
- nur 1 Register
- asynchroner Reset
- synchroner Preset

Der 18V10 enthält zehn unabhängig konfigurierbare OLMCs. Auf Seiten der Eingänge der Makrozelle sind wiederum die Produktterme angeordnet. Intern enthält die Makrozelle vor allem Multiplexer und ein D-Register, über deren Programmierung man verschiedene Ausgangskonfigurationen festlegen kann. So kann festgelegt werden, wie der Ausgang der Makrozelle beschaltet wird, ob der Ausgang der Produktterme direkt auf den Ausgang wirken soll, welche Ausgänge des D-FFs auf den Ausgang geschaltet werden soll usw..

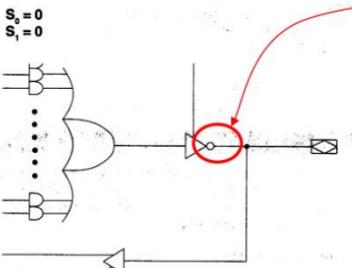
Der *Feedback Multiplexer* legt fest, aus welcher Quelle die Rückführung der OLMC in die Matrix bezogen werden soll. Der *Output Multiplexer* hat die Aufgabe, das D-Flip-Flop in den Ausgangsweg einzukoppeln oder es auszublenden.

Weitere derartige Details sind jedoch den Datenbüchern zu entnehmen. Dass heutige Bausteine dieser Art auch über asynchrone Set- und Reset-Funktionen verfügen, versteht sich von selbst.



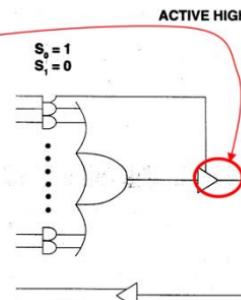
Registered mode

ACTIVE LOW



ACTIVE LOW

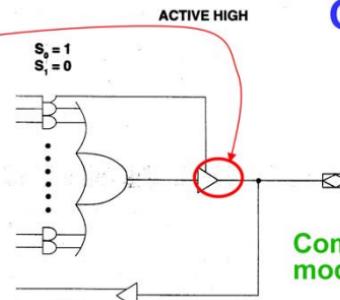
$$\begin{matrix} S_0 = 0 \\ S_1 = 0 \end{matrix}$$



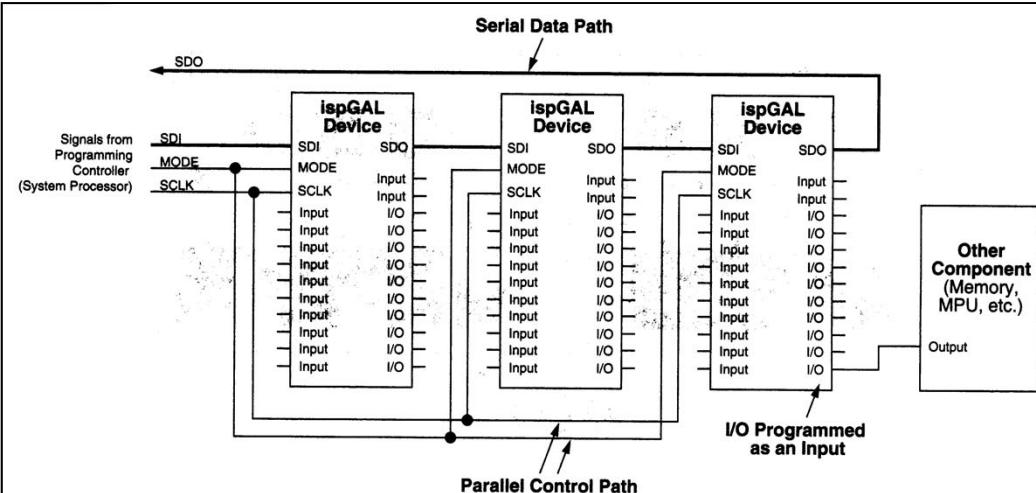
Combinatorial mode

ACTIVE HIGH

$$\begin{matrix} S_0 = 1 \\ S_1 = 0 \end{matrix}$$



GAL18V10  
modi



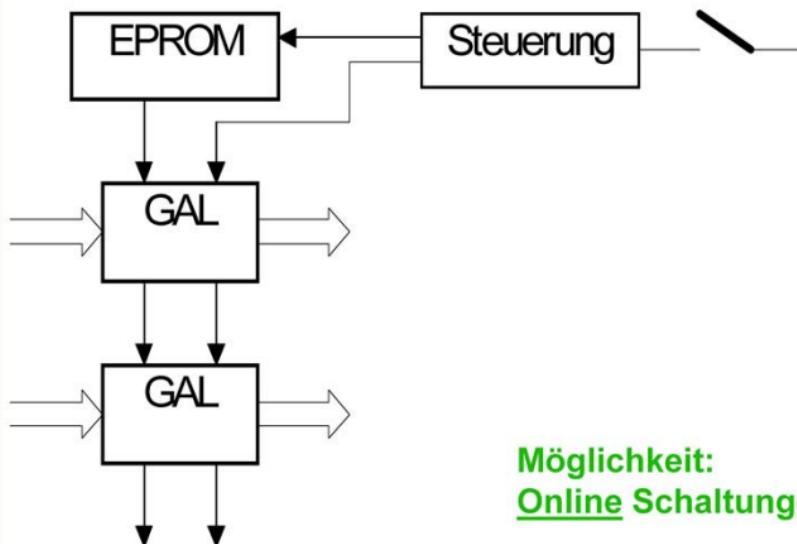
### Serial Daisy Chain

- während der Programmierung keine Unterbrechung möglich
- einzelne Bausteine nicht einzeln programmierbar

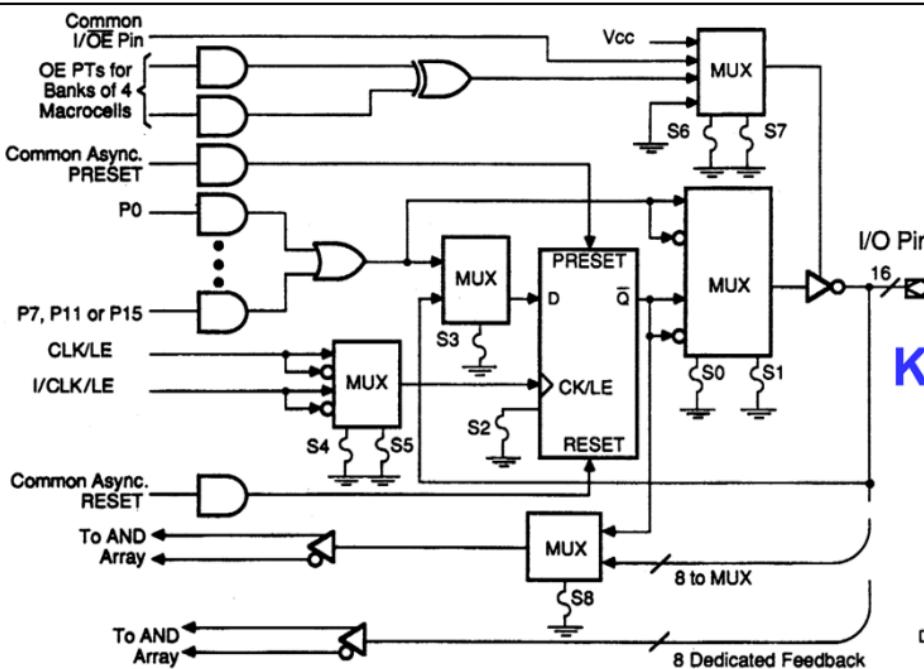
Die Programmierung von GALs erfolgt heutzutage über flexible Programm-Tools, die unterschiedliche Eingabemöglichkeiten bis hin zur Minimierung zulassen. Die Ausgabe kann über das JEDEC-Format erfolgen, auf das man sich schon bei einfacheren PAL-Bausteinen geeinigt hat. Der Aufbau dieses Formates ist einfach gehalten. Die zu übertragenden Daten sind in einer Matrix angeordnet, deren jeweilige Zeilen zunächst einen Kennbuchstaben, dann die Adresse, gefolgt von den eigentlichen Nutzdaten, enthalten.

GALs wurden vor allem entwickelt, um Schaltungen auch on-board zu verändern. Dabei kommt es nun darauf an, die Ladevorgänge möglichst effizient zu gestalten. Die Firmen bieten hierzu Methoden wie das Hintereinanderschalten von GALs an, wie es Bild oben zeigt. Damit sind dann auch Lösungen denkbar, denen on-line Schaltungen verschiedener Art geladen werden können, um Typenvariationen von Geräten, die über Schalter einstellbar sind, zu erhalten. Die Anzahl der unterschiedlichen Hardwaretypen lässt sich somit oft drastisch senken.

# Gesteuerte Schaltungsvariation



Möglichkeit:  
Online Schaltungen laden!!??



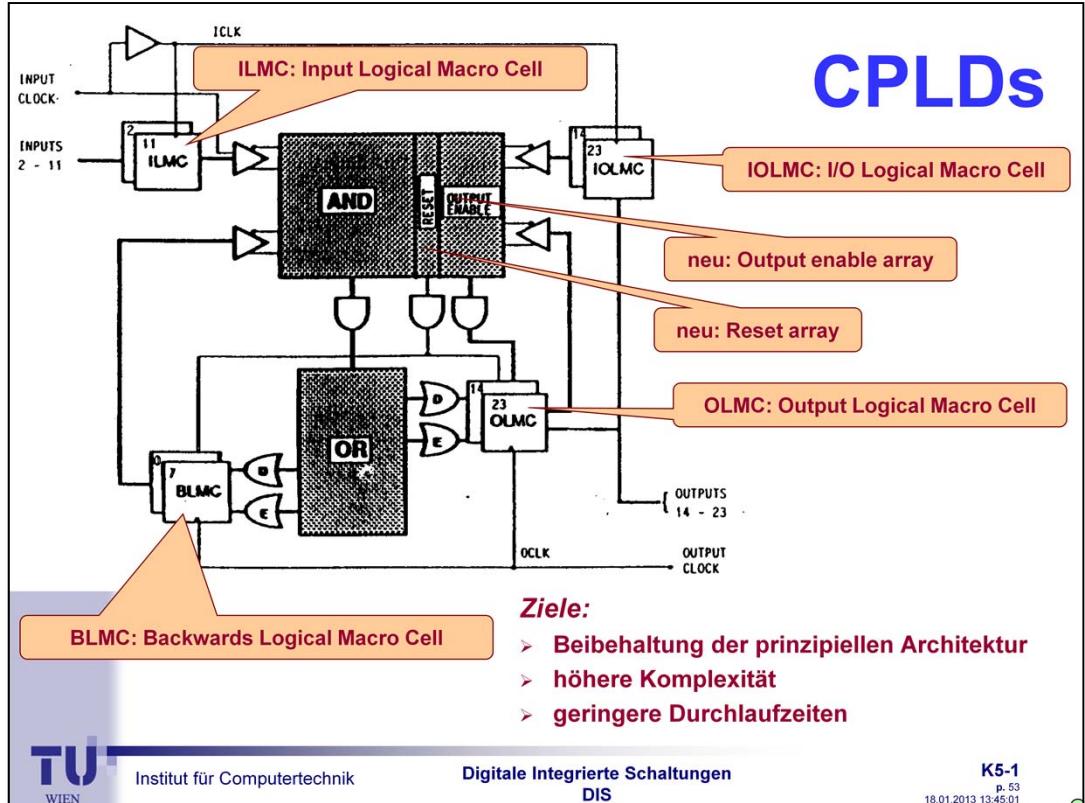
## Komplexität nimmt zu

DF006181

trotzdem:

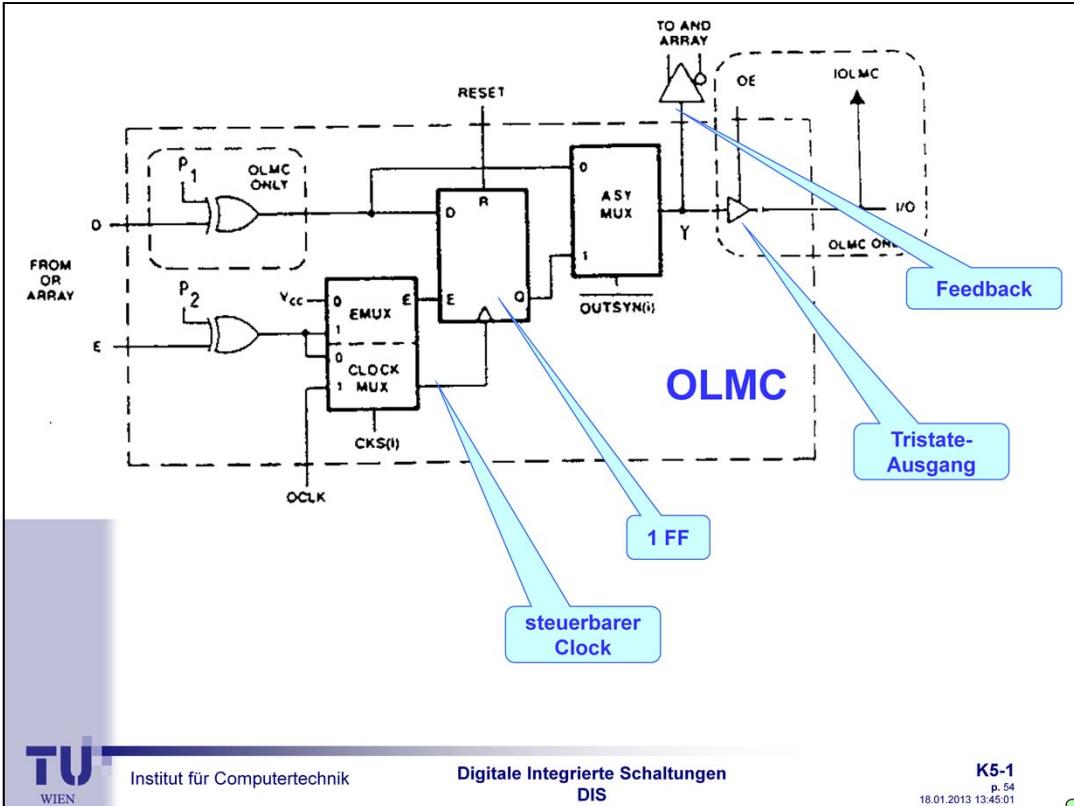
in einer I/O-Zelle nur 1 Register  
aber mehr und mehr  
kombinatorische Variationen

# CPLDs



## CPLD: Complex Programmable Logic Devices

Bei der Folgegeneration von GAL-Bausteinen legte man zunächst Wert darauf, die schon sehr kurzen Laufzeiten auf 10 ns und weniger zu drücken, aber auch auf eine höhere Komplexität und somit flexibleren Einsatz dieser Bausteine. Das Bild oben zeigt den typischen Aufbau derartiger Bausteine. Zentral sind die programmierbaren AND- und OR-Arrays angeordnet. Jedem Eingang ist ein *Input Logical Macro Cell* (ILMC) vorangeschaltet. Für die Art der Rückkopplung, zur Bildung von Automaten, sind die *Backwards Logical Macro Cells* (BLMC) zuständig. Die Ausgangssignale der *Output Logical Macro Cells* (OLMC) werden zum einen an die Ausgangspins des GAL6001 geführt, können aber über die *Input Output Logical Macro Cells* (IOLMC) wieder auf das And-Array zurückgekoppelt werden. Eine Direktanschaltung der OLMC-Signale auf das AND-Array ist gleichfalls gegeben, was den Vorteil bietet, dass bei Ansteuerung der Steuerleitung durch *Output Enable* Ausgänge auch als Eingänge verwendet werden können.



Das Bild zeigt den Aufbau einer Makrozelle des Typs OLMC. Sie besteht zentral aus einem D-Register, das über einen Enable-, Reset- und Clock-Eingang verfügt. Die Steuerung wird von den verschiedenen Multiplexern übernommen. Der D- beziehungsweise E-Eingang der Zelle erhält seine Daten vom OR-Array, wobei er über  $P_1$  invertierbar ist. Der Baustein lässt zudem zu, dass Zellen als reines Schaltnetz, D-FF, RS-FF oder als JK-FF konfiguriert (programmiert) werden.

# Problem

## WUNSCH:

<i>hohe Flexibilität</i>	$\leftrightarrow$	<i>kleine Fläche</i>
<i>applikations-unspezifisch</i>	$\leftrightarrow$	<i>applikationsspezifisch</i>
<i>kleinste Granularität</i>	$\leftrightarrow$	<i>Durchschaubarkeit</i>
<i>wenige FFs in einer Zelle</i>	$\leftrightarrow$	<i>Zellen hoher Performance</i>

## KONSEQUENZEN:

<i>Datenpfade zentrale Bedeutung</i>	$\leftrightarrow$	<i>Vielfalt explodiert (vgl. Risc-Proz.)</i>
<i>Entwurf immer komplexer</i>	$\leftrightarrow$	<i>Spezialisierung auf wenige</i>

Durch die GALs zeichnete es sich schon ab: der entscheidende Unterschied zwischen den ersten programmierbaren und wieder löschenbarem Bausteinen und der Nachfolgegeneration ist die zunehmende Komplexität der Bausteine, die sich auch in den verschiedenen, neu kreierten Namen widerspiegelt: CPLDs (Complex PLD), FPLA (Field PLA), Multi-PAL-Architektur, MAPL (Multi Array Programmable Logic) usw. Die Schwierigkeit, die sich bei allen Bausteinen ergibt, ist die Notwendigkeit, sie einerseits so zu designen, dass sie für einen breiten Einsatz vorteilhaft sind, also möglichst allgemein spezifiziert werden, andererseits aber bei der konkreten Schaltung einen möglichst flächenoptimierten Schaltungsentwurf zulassen sollen.

Hinzu kommt das Problem der UND-ODER-Strukturen, die Produktterm-Makrozellen-Zuordnung effizient zu lösen. Im konventionellen PAL, GAL, .. sind die Produktterme festen ODER-Funktionen zugeordnet. Das bedeutet, für jeden Ausgang müssen die vorausgehenden Schaltungen so oft dupliziert werden, wie sie benötigt werden. Ein "Mischen" der Funktionen ist nicht möglich. Die Flexibilität des Einsatzes ist eingeschränkt. Der Overhead ist hoch.

Ziel hochintegrierter, programmierbarer Bausteine muss es also sein, dem globalen Routen auf dem Baustein mehr Beachtung zu schenken, also eine flexiblere Zuordnung zwischen den Makrozellen zu erreichen. Exakt dies drücken die Strukturen der Bausteine aus, die zu den Familien höherer Komplexität gehören.

## Firmen versuchten ideales Prinzip zu finden

### Main stream:

- ❖ hohe Zellenanzahl
- ❖ in 1 Zelle nur 1 – 2 FFs
- ❖ hochwertige Tools (nichts mehr "per Hand")
- ❖ hoher Abstraktions-Level (HW immer unbekannter; immer mehr auf Symbolebene)



Es bleiben jedoch die Widersprüche: einerseits einen Baustein zu haben, der allgemein einsetzbar ist und möglichst viele konventionelle Standardbausteine ersetzen kann, andererseits aber höchste anwendungsspezifische Anforderungen erfüllt, dann wiederum einen Baustein zu haben mit möglichst vielen Makrozellen, aber gleichzeitig auch mit möglichst effizienten, flexiblen Kopplungsmöglichkeiten zwischen den Makrozellen usw. Diese Widersprüche versuchten die unterschiedlichen Firmen jeweils auf ihre Weise zu lösen, wobei sich jedoch bei allen Konzepten bestimmte Gemeinsamkeiten ergaben.

- p Man behielt die Makrostruktur mit *kleinen* Zellen bei. Die Granulation zu vergrößern, also die Anzahl der FFs auf größer als 4 zu erhöhen, hat bisher keine Firma als vorteilhaft angesehen (normal ist die FF-Anzahl gleich 1).
- p Es muss zwischen "internen" (Feedback-Zellen) und I/O-Makrozellen unterschieden werden, was nicht nur technologiebedingt, sondern vor allem funktional sinnvoll ist.
- p Die Verbindungsbahnen nehmen mehr und mehr an Bedeutung zu; sie verlangen besondere "Bussysteme". Beim Routing der Schaltung müssen die Laufzeiten in den Optimierungsalgorithmen berücksichtigt werden, eine prinzipielle Vernachlässigung ist nicht mehr möglich.
- p In CPLDs werden einfache Strukturen als Blöcke definiert. Die Laufzeiten differieren, wenn eine unterschiedliche Anzahl von kombinatorischen Termen und Multiplexern in Datenpfade eingebunden wird. Trotzdem wird versucht, den Vorteil von CPLDs beizubehalten, die Laufzeiten fest vorgeben zu können (sie nicht vom Routing abhängig werden zu lassen). Dass dies nur annähernd sein kann, da letztendlich das Routing doch einen Unsicherheitsfaktor darstellt, und so das Resultat bei einem Baustein besser, bei einem anderen schlechter ausfällt, ist selbstverständlich, was bei der Wahl der Bausteine berücksichtigt werden muss. Programmierbare Bausteine hoher Komplexität stellen immer einen Kompromiss unterschiedlicher Randbedingungen dar.

# Generationen

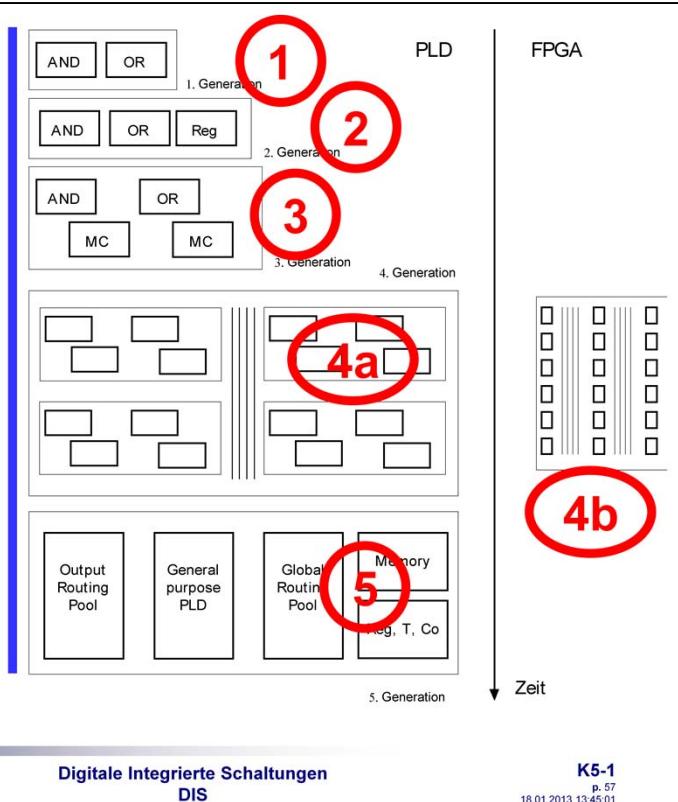
**CPLDs:**  
Complex Programmable  
Logic Device

**FPGAs:** feld- programmier-  
bare Gate Arrays

CPLD  $\leftrightarrow$  FPGA

Co: Counter

MC: Macro Cell



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-1

p. 57  
18.01.2013 13:45:01

Zeichnet man sich also die Prinzipien über die Zeitachse auf, gelangt man zu einer Darstellung nach Bild oben. Die Schritte auf der linken Seite des Bildes sind verhältnismäßig einfach nachzuvollziehen. Der Schritt zum Baustein auf der rechten Seite wird verständlich, wenn man bedenkt, dass beim Entwurf der zu implementierenden Schaltungen immer mehr zu formalen Methoden übergegangen wird, eine Grafikeingabe aufgrund der Komplexität der Schaltungen zunehmend ineffizient wird. Wenn aber Schaltungen nur noch über Tools entwickelt, optimiert und geroutet werden, sollte die Granularität der Zellen möglichst fein sein. Während jedoch die Bausteine der linken Seite EPLDs darstellen, handelt es sich bei jenen der rechten Seite um Bausteine vom Typ SRAM, auf die im folgenden noch eingegangen wird. Auf eines kann jetzt schon hingewiesen werden: Bei den Bausteinen des rechten Typs sind die Laufzeiten im jeweiligen Fall zuerst zu simulieren, da die Granulation sehr gering und der Datenpfad völlig vom Routing abhängig ist.

Das Blockschaltbild in Bild links unten stellt die neueste Generation von PLDs dar. Zu den bisherigen Arrays, Makrozellen und Verbindungsstrukturen kommen nun noch Grundbausteine der Mikroprozessoren wie RAM, Zähler, Timer usw. Das Design eines Prozessors kann vollständig in einem Baustein vorgenommen werden, wenn die Größe des programmierbaren Bausteines ausreichend ist.

# Generationen

1. Generation: AND, OR
2. Generation: AND, OR, Register
3. Generation: CPLDs
4. Generation: "high" CPLDs
5. Generation: "systems"

*Im Folgenden:*

**3. bis 5. Generation + LCAs**

# Differenzierter

PLDs, basierend auf Makrozellen			Programmierbare Gate Arrays		
programmierbar: UND fest: ODER	programmierbar: UND + ODER	CPLDs mehrere Arrays		kanalstrukturierte Gate Arrays	Sea of Gates
PAL, GAL, EPLD, ..	PLA, PLS, ..	MAX, MACH, PLUS LOGIC, ..	LCA	FPGA	ERA
Produktterme, zahlreiche Variationen, typisch auch für Mealy- Anwendungen			SRAM	Antifuse- Technologie, nichtflüchtig	SRAMs auf NANDS aufbauend

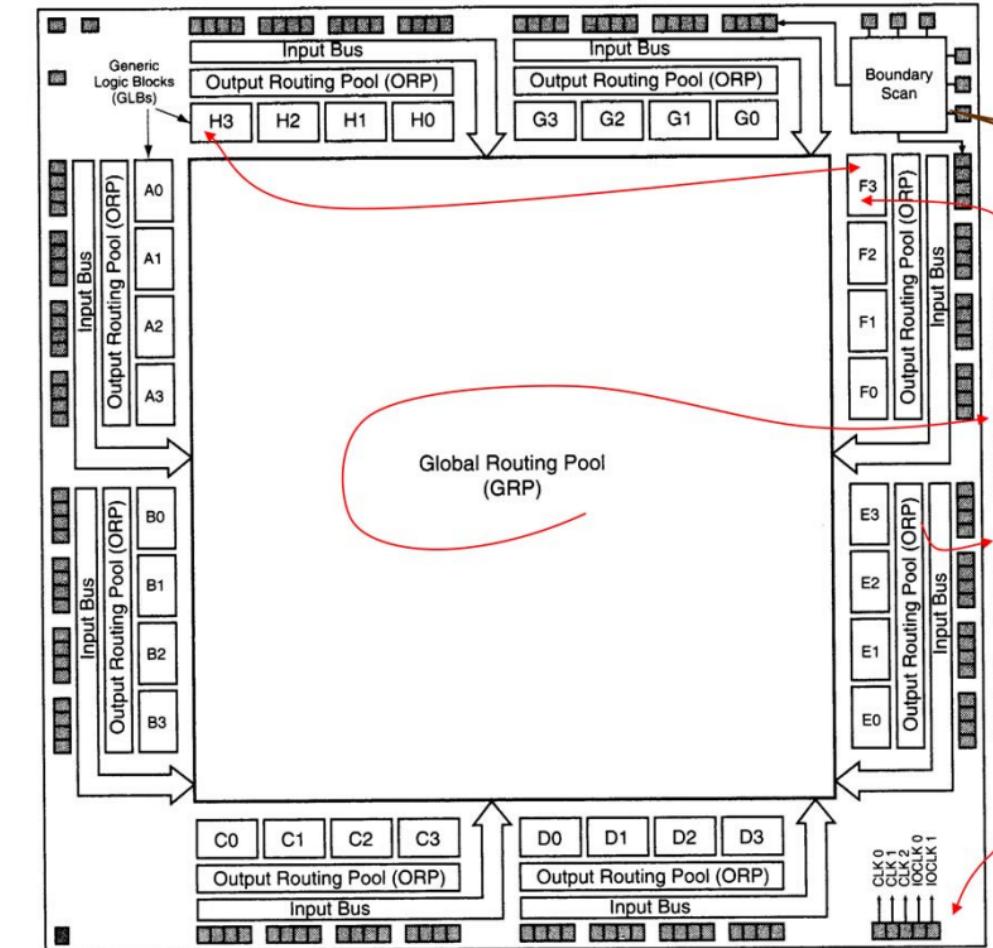


Man muss unterschieden zwischen LCA, FPGA und Sea of Gates.

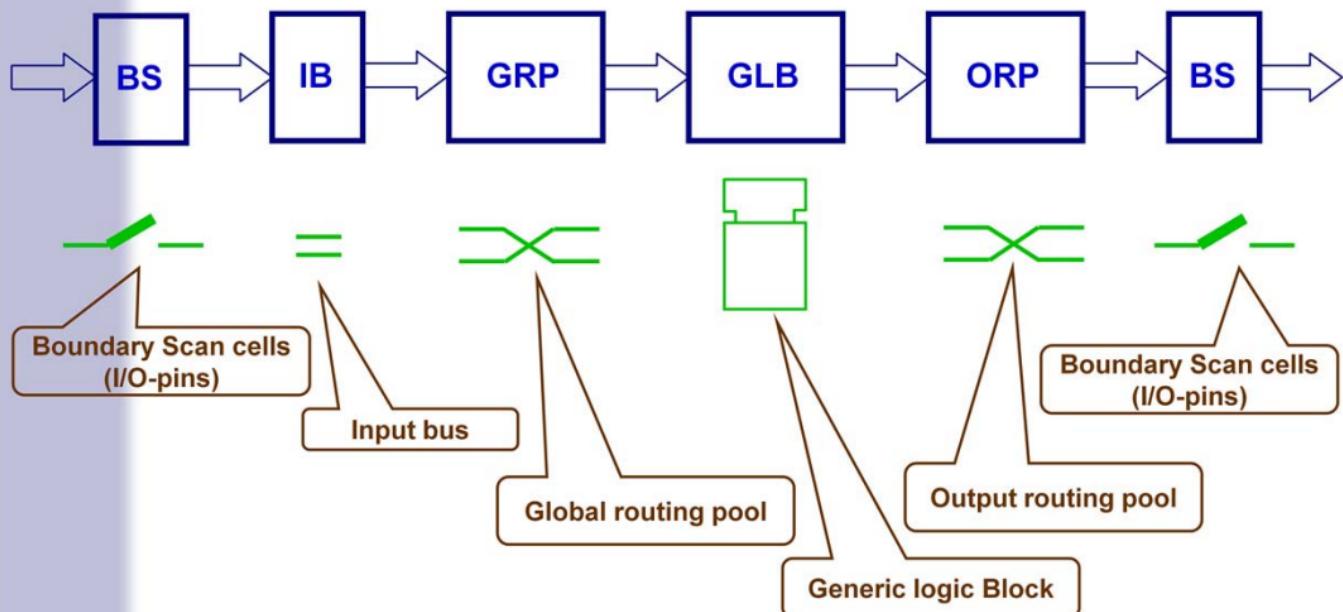
Typische Vertreter der Familien der *Programmierbaren Gate-Arrays* sind d!“=

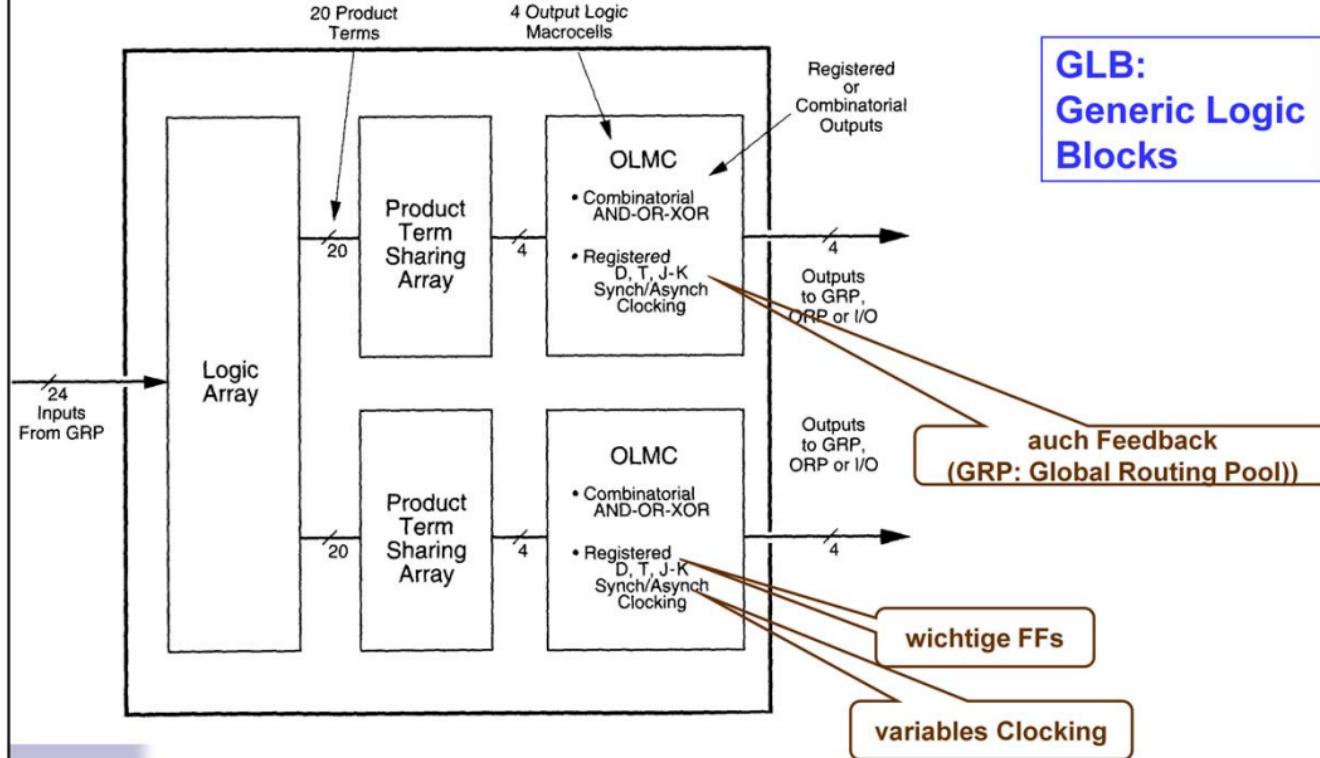
ie Bausteine auf der Basis von *Sea of Gates*, Bausteine mit konfigurierbaren Logikblöcken und Gate Arrays mit Kanalstruktur. Es wird im einzelnen darauf noch eingegangen.

Programmierbare Gate Arrays besitzen als Basiselemente einfachste Zellen wie NANDs (beispielsweise beim ERA-Typ), die entsprechend zu programmieren sind. In der Literatur wird als Nachteil oft angegeben, dass durch die Granularität der Zellen der Compiler-Aufwand im Vergleich zu den auf Makrozellen basierenden PLDs sehr hoch ist. Dies ist logisch, da eine feine Granularität eine hohe Flexibilität erlaubt, die wiederum mit einem erhöhten Entwicklungsaufwand bezahlt werden muss. Da man aber heute entsprechende Maschinen zum Compilieren zur Verfügung hat (jeder etwas bessere PC reicht heute dazu aus), ist diese Negativwertung nicht mehr stichhaltig. Dass der Ausnutzungsgrad durch die feine Granularität sehr hoch ist, versteht sich von selbst.



## Lattice Semiconductor





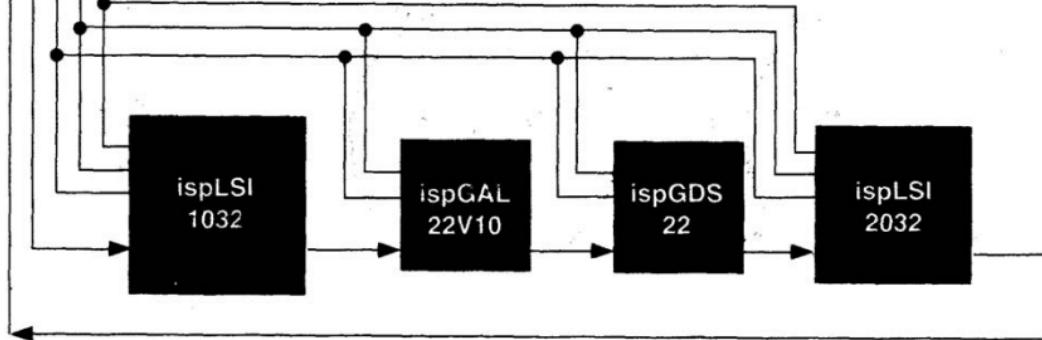
## Family

	ispLSI 1000E Family	ispLSI 1000E Family	ispLSI 1000E Family	ispLSI 1000E Family
Density of PLD Gates	2.000 – 8.000	1.000 – 6.000	8.000 – 14.000	25.000
Speed (MHz, max)	125 – 60	154 – 80	100 - 50	70 - 50
Speed (ns)	7,5 – 20	5,5 – 15	10 – 20	15 – 20
Macrocells	64 – 192	32 – 128	192 - 320	192
Registers	96 – 288	32 – 128	384 – 480	416
Memory (Bits)	-	-	-	4608
Inputs & I/Os	36 – 110	34 – 136	128 – 192	159
Pins/Package	44-68-, 84-PLCC 84-, 133-CPGA, ..	44-, 84-PLCC, ..	160-, 208-, 240 MQFP, ..	208 MQFP

SDO  
SDI  
MODE  
SCLK  
ispEN

} 5-wire ISP Programming Interface

Ein Schaltungsvorschlag der Firma Lattice

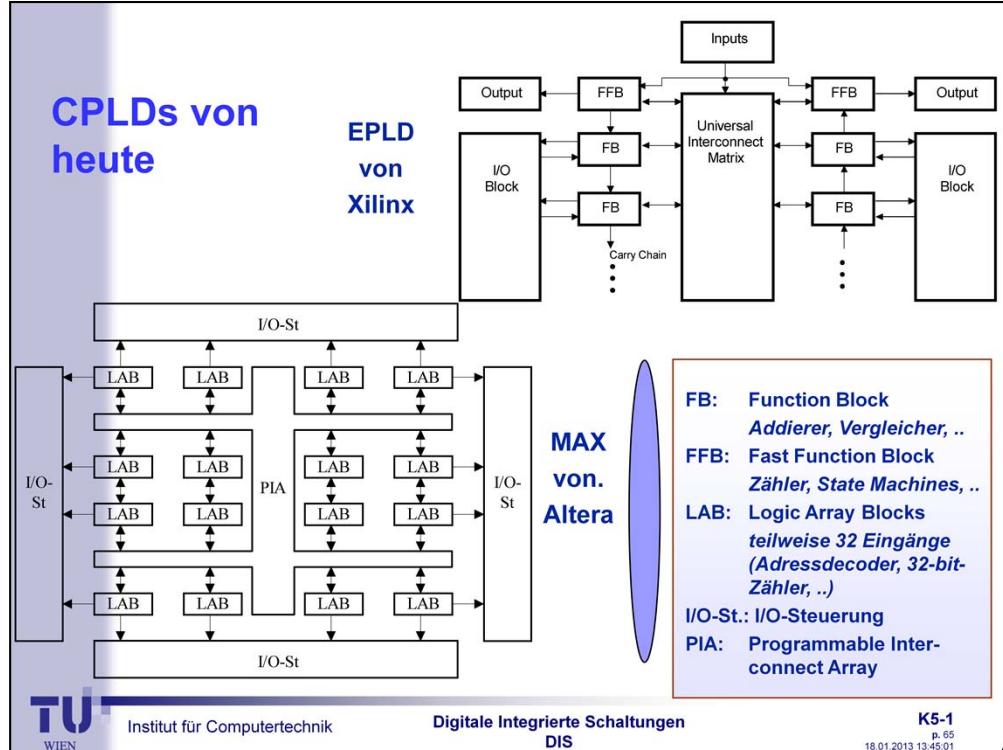


Was fällt auf?

Vorgeschlagen werden Bausteine, deren Namen und Symbole nichts mehr sagen.

Vielfältig! Undurchsichtig! Lange Einarbeitungszeit! ...

## CPLDs von heute



Baustein rechts oben: XC73108 (FB: Function Block, FFB: Fast Function Block)

Baustein links unten: Blockschaltbild der Bausteine der MAX-7000-Familie (I/O-St.: I/O-Steuerung, LAB: Logic Array Block, PIA: Programmable Interconnect Array)

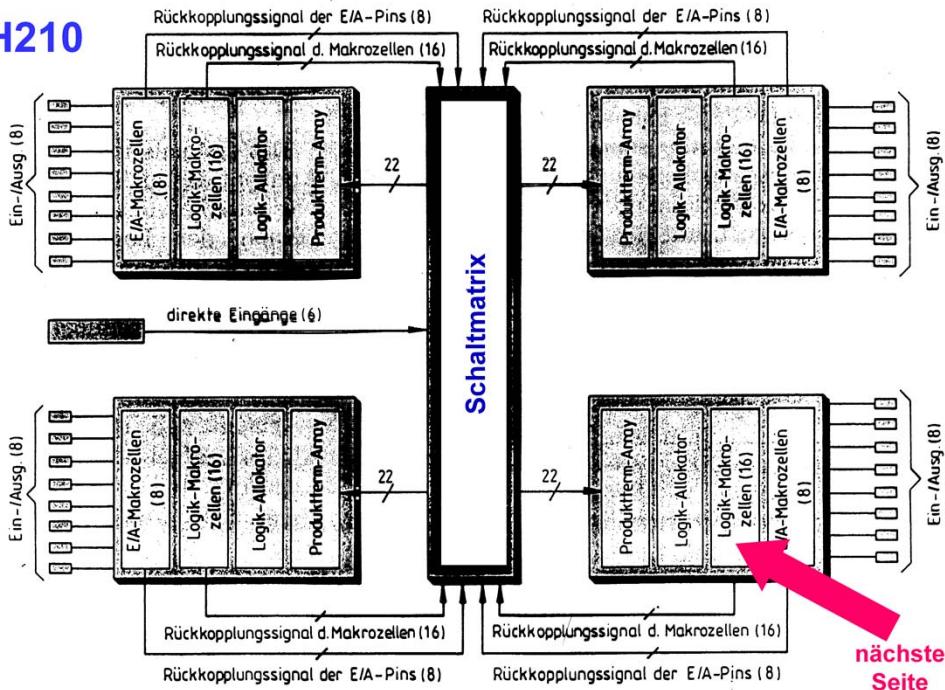
Drei interessante Blockschaltbilder von CPLDs sollen noch kurz beispielhaft angeführt werden, das des XC73108 (EPLD) der Firma Xilinx (Bild rechts oben), das der MAX-7000-Familie der Altera und das des Bausteins MACH210 der Firma AMD. Da oft Zählimpulse hoher Frequenz gezählt werden müssen, unterscheidet man beim XC73108 zwischen normalen Function Blocks und Fast Function Blocks, die auch jeweils getrennte Eingänge haben. Damit sind Pin-to-Pin-Verzögerungszeiten von unter 12 ns realisierbar. Die FFBs können mit einer maximalen Zählfrequenz von 80 MHz als ladbare Zähler oder als Zustandsmaschine anderer Art betrieben werden. Die FBs enthalten Addierer mit Lookahead-Carry-Charakteristik und Vergleicher. Nach Angaben des Herstellers lassen sich damit z. B. 55-MHz-synchrone Zustandsmaschinen und 18-bit-Akkumulatoren realisieren.

Die Blockschaltung der Bausteine der MAX-7000-Familie zeigt deutlich die Symmetrie des Bausteins sowie die im Grunde genommen geringe Anzahl der *Logic Array Blocks* (LAB). Die LABs derartiger Bausteine verfügen dabei jedoch oft über 32 Eingänge, was beispielsweise für Adressdecoder enorme Geschwindigkeitsvorteile bringt. Besitzt eine Makrozelle, wie bei FPGAs anzutreffen, nur 4 Inputs, werden für einen 24-bit-Adressdecoder gleich dreistufige Logikschaltungen notwendig, was erhöhte Laufzeiten mit sich bringt gegenüber einer Zelle mit 32 Inputs, die mit einer Stufe auskommt. Nicht verwunderlich also, dass mit einem Baustein der Familie MAX-7000 36-bit-Zählerketten zu realisieren sind, die vollsynchrone mit ca. 78 MHz laufen, wie es der Hersteller angibt.

Die Anzahl der Produktterme pro Makrozelle ist beim vorliegenden Baustein 5, wobei viel Wert auf eine flexible Verwendung gelegt wird: Rückkopplungsmöglichkeiten, gesonderte Taktzuführung, asynchrone Setz- und Rücksetzfunktionen, zugeführt über Produktterme, was allerdings die Taktrate beschränkt, oder global zugeführt über gesonderte Pins. Das FF jeder Makrozelle kann als D-, T-, JK oder auch als RS-FF konfiguriert werden. Programmierbar ist auch die Geschwindigkeits-Stromaufnahmefunktion. Der Designer kann also über die Programmierung der Makrozelle darüber bestimmen, ob diese wenig Leistung verbrauchen oder einer hohen Taktrate genügen soll.

# Leistungsmerkmale

- **2 unterschiedliche Typen von "intelligent Blocks":  
Fast Function Blocks und Normal Function Blocks**
- Pin-to-Pin-Verzögerung min. 12 ns
- max. Zählfrequenz: 80 MHz
- einfache Realisierung von Automaten
- enthalten Addierer mit Lookahead-Carry-Generator
- beispielhafte Realisierungen: 55-MHz-synchrone  
Zustandsmaschinen oder 18-bit-Akkumulatoren



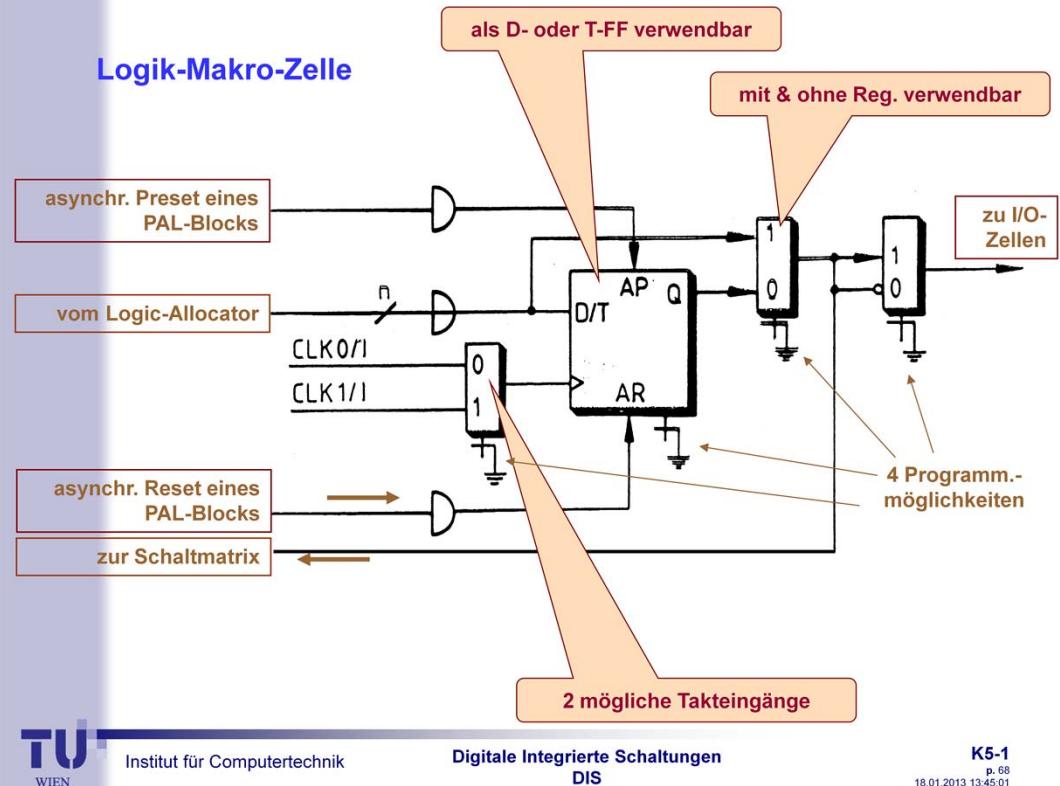
Der vierte und letzte Baustein, der hier noch angeführt werden soll, ist der MACH210, der schon seit Jahren Nachfolger hat, den MACH220 usw., die sich im wesentlichen nur darin unterscheiden, dass sie noch mehr Zellen der unterschiedlichen Sorten enthalten. Er wird als Brücke zwischen den FPGAs und den "schnellen" PALs gehandelt, da er zum einen eine recht komplexe Struktur zeigt, zum anderen sehr gute, garantierte Durchlaufzeiten.

Das Bild zeigt das Blockschaltbild. Auffallend sind die 4 PAL-Blöcke, die um eine Schaltmatrix angeordnet sind. Diese Blöcke bestehen jeweils aus logischen Arrays und unterschiedlichen Macrocells. Die Schaltmatrix ist direkt über Eingänge zugänglich, verbindet aber im wesentlichen die vier großen Blöcke. Die Laufzeit in der Schaltmatrix wird mit maximal 2 ns angegeben, was einen recht niedrigen Wert darstellt.

Die Schaltmatrix trifft zunächst auf ein Product-Term-Array (UND-Verbindungen). Dem schließt sich direkt ein Logic Allocator an. Dieser übernimmt die Daten der 64 Produktterme und ordnet sie 16 Makrozellen zu, wobei nur eine vernachlässigbare zusätzliche Verzögerungszeit auftritt. In PALs fehlt im allgemeinen dieser Logic Allocator, sie verfügen im allgemeinen über eine feste Zuordnung zwischen den Produktterms und den Makrozellen sowie den Ausgangszellen, was nach Angaben des Herstellers eine ungünstige Ausnutzung der Siliziumfläche mit sich bringen kann. Da die Tools von AMD ein automatisches 100%-Routing des Bausteins über ein entsprechendes Tool voraussetzen, bedeutet dabei der Allocator keinen zusätzlichen Aufwand für den Designer.

Dem Logic Allocator sind unterschiedliche Zelltypen nachgeschaltet: zwei verschiedene Macrocells: die Output und die Buried Cells, die jeweils ein FF sowie Logik enthalten, und die einfachen I/O Cells ohne FFs, aber mit Tristate-Treiber.

## Logik-Makro-Zelle



T-FF: Toggle-FF: bei jeder positiven Flanke toggelt das FF

Die Output Cells (die keine direkte Verbindung zu den IC-Pins haben) können als Latch (D-Latch), als Register (T-FF) oder rein kombinatorisch programmiert werden. Sie enthalten Rückkopplungsmöglichkeiten über das jeweilige FF oder am FF vorbei (asynchrone Verschaltungsmöglichkeit). Das FF kann über zwei unterschiedliche Clocks angesteuert werden. Fast selbstverständlich sind die asynchronen Reset- und Preset-Leitungen. Die Eingangslogik kann als active-low oder als active-high programmiert werden.

Die I/O Cells enthalten im wesentlichen einen Tristate-Buffer und einen Multiplexer (4:1). Vorteil der Entkopplung der Output Macrocells von den Output Cells ist die zweifache Rückkopplungsmöglichkeit auf die Switch Matrix (einmal von der Output Macrocell aus, zum anderen von der Output Cell aus). Beim Aufbau von Schieberegister und Zähler bietet diese Art der Verschaltung bezüglich der Flächenoptimierung Vorteile.

## MACH:

- ❖ mehrere Arrays
- ❖ PAL-Blöcke über eine Schaltmatrix verbunden
- ❖ Schaltmatrix mit direkten Eingängen + Rückkopplungseingängen
- ❖ I/O- & Logik-Makro-Zellen
- ❖ Schaltmatrix-Laufzeitverzögerung: 1 .. 2 ns
- ❖ Gesamtdurchlaufszeit: 15 ns
- ❖ max: 50 MHz
- ❖ Tristate-Buffer

## Advertising

.. family at 15 ns. MAX devices could range from 25 ns to 52 ns. When using more than 16 product terms, MACH operates at 30 ns while MAX devices could range from 37 ns to 102 ns. The MACH family's performance is predictable in any application. MAX products can only be determined after several iterations of simulation costing you delayed time to market and money .."

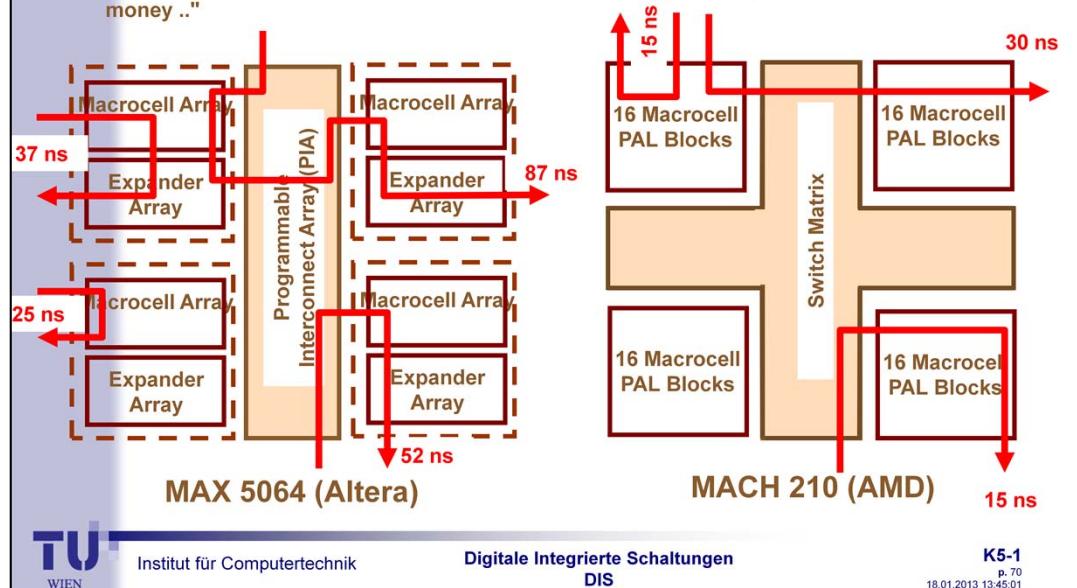


Bild oben ist einem Marketing-Papier von AMD entnommen. Hierzu heißt es: "System designers are speed, cost and time-to-market driven. The MACH family provides a 66% to 300% speed improvement over competing technologies such as Altera's MAX™ devices (depending on the application). In typical applications where fewer than 16 product terms are used, you can count on the MACH family at 15 ns. MAX devices could range from 25 ns to 52 ns. When using more than 16 product terms, MACH operates at 30 ns while MAX devices could range from 37 ns to 102 ns. The MACH family's performance is predictable in any application. MAX products can only be determined after several iterations of simulation costing you delayed time to market and money." Note: Numbers on the block diagram are best case numbers, and vary depending on the path. Following are ranges: (1) 25 to 40; (2) 37 to 52; (3) 87 to 102; (4) 52 to 67."

Derartige "Töne" waren in früheren Zeiten nicht üblich. Man hob die Vorteile seiner Bausteine heraus, vermeid aber die Diskriminierung anderer Firmen. Daraus lässt sich erkennen, wie hart der IC-Markt geworden ist, man erkennt aber auch, dass alle Angaben der Hersteller in Flyers nicht sehr viel wert sind. Der Designer ist gezwungen, vor allem bei zunehmender Komplexität der Bausteine, die Daten nicht nur exakt nachzuprüfen, sondern sie auf Lücken hin zu untersuchen.

- CPLDs lassen sich nicht mehr "per Hand" designen
- Metastabilität: MTBF normal > 5 h  
spezielle Bausteine erreichen: < 20.000 Jahre
- CPLDs zeichnen sich aus durch "feste" Durchlaufzeiten
- bei komplexen Schaltungen: vorsicht vor zweimaligem "Bausteindurchlauf"
  
- anfangs: ca. 800 Gatteräquiv. + 1,2 mm + 2 Metallebenen
  - 1997: ca. 50.000 GÄ - 0,25 mm - 5 ME
  - 1999: ca. 1.000.000 GÄ - 0,25 mm - 5 ME
  - 2000: ca. 2.000.000 GÄ - 0,15 mm - 7 ME
  - 2005: ca. 50.000.000 GÄ - 0,075 mm
  - 2010: ca. 250.000.000 GÄ
- neue Bausteine enthalten Multi-Core-Technologie
- beispielsweise.: Produkt-Terme + RAM-Blöcke + MUX-Block

### Zusätzliche Bemerkungen zu CPLDs:

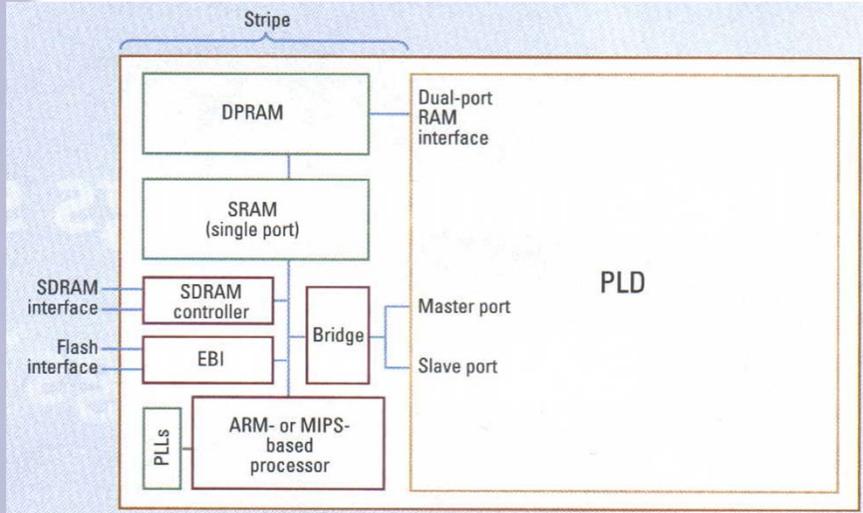
- CPLDs lassen sich generell nicht mehr "per Hand" designen und programmieren. Entscheidend ist also bei der Auswahl, welche Werkzeuge für die Entwicklung zur Verfügung gestellt werden können.
- **Metastabilität** [1] von taktflankengesteuerten FFs ist prinzipiell nicht zu vermeiden. Man kann jedoch das Problem verlagern. So kann man beispielsweise die Wahrscheinlichkeit - die in diesem Fall allgemein mit dem MTBF-Wert [2] gemessen wird - so reduzieren, dass sich die Metastabilität in einem Master-Slave-FF über den Master hinaus kaum fortpflanzt. Zu erreichen ist dies, indem schaltungstechnische sowie technologische Maßnahmen (Veränderung bestimmter Parameter) vorgenommen werden. Philips bietet beispielsweise einen PLD an, den ABT22V10, der einen MTBF-Wert von 4,1 Stunden anstatt 43.000 Jahren gegenüber dem 22V10 aufweist. In Kauf genommen werden muss dabei allerdings eine Verzögerungszeit, deren Wert nicht garantiert werden kann (für Absolutzeitmessungen sehr hoher Auflösung also nicht geeignet).
- CPLDs werden oft als asynchrone Bausteine für digitale Schaltungen angeboten. Man kann in diesem Fall nur wiederholt betonen, von diesen Schaltungen Abstand zu nehmen, da die Kosten der Wartung solcher Schaltungen nicht zu übersehen sind.
- Die angegebenen Durchlaufzeiten in CPLDs sind im allgemeinen Worst-Case-Angaben, die wiederum auf einer *bestimmten* Durchlaufzahl von Termen basieren. Wird diese Zahl von Termen, die funktional hintereinandergeschaltet sind, überschritten, kommt es zu einem zweiten Gesamtdurchlauf des Bausteins, was die totale Durchlaufzeit verdoppeln kann.
- Die Switch-Matrizen verschiedener Bausteine können unterschiedliche Aufbauweisen haben. So gibt es aufwendige Strukturen, die nach der Kreuzschienenmethode funktionieren und damit eine große Anzahl von Kombinationsmöglichkeiten erlauben. Andere bauen auf dem Prinzip der Multiplexer auf.

[1] Bei einem asynchron einlaufenden Signal eines taktflankengetriggerten FFs kann nicht garantiert werden, dass die Setup und die Hold Time eingehalten werden. Dann muss das FF nicht nach der Clock-to-Output Time in den definierten Zustand übergehen, sondern kann länger in einem einschwingenden Zustand "hängen" bleiben, bis es wieder in einen stabilen Zustand übergeht.

[2] MTBF: Mean Time Between Failures ist die Zuverlässigkeitgröße einer empirisch zu ermittelnden mittleren Zeit zwischen zwei aufeinanderfolgenden Fehlverhalten eines Systems. Für elektronische Komponenten kann die Exponentialverteilung angenommen werden

## Typischer "CPLD" von heute

Altera



oder ist das ein Controller?

- aus Embedded Systems im November 2000;
- WICHTIG: innen ist ein PLD und kein CPLD; das Ganze kann man aber als CPLD bezeichnen.

*Perspektiven:*

- Um ein Maß für die Größenordnung heutiger CPLDs zu erhalten: Im Rahmen der MACH-Familie werden heute schon programmierbare Bausteine mit bis zu 256 Makrozellen angeboten.

# Digitale Integrierte Schaltungen

Fach: Schaltungstechnik

*Eine Einführung in komplexe Schaltwerke und ASIC-Design*

Dietmar Dietrich

ICT

Institut für Computertechnik

[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



TU Archiv / Karl Mayrstorfer, Fassade um 1860  
(Gebäudevertrag 119)



Institut für Computertechnik

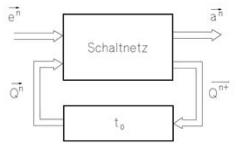
Digitale Integrierte Schaltungen  
DIS

K5-2  
p. 1 25.09.2011 17:48:35

# Teil 5-2

# ASIC

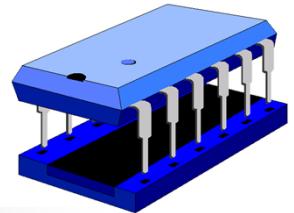
# Design



```
entity Half_adder is
  port(x, y : in bit;
       cout, s : out bit);
end Half_adder;
```



Programmierbare Bausteine



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-2

p. 2 25.09.2011 17:48:35

# Generationen

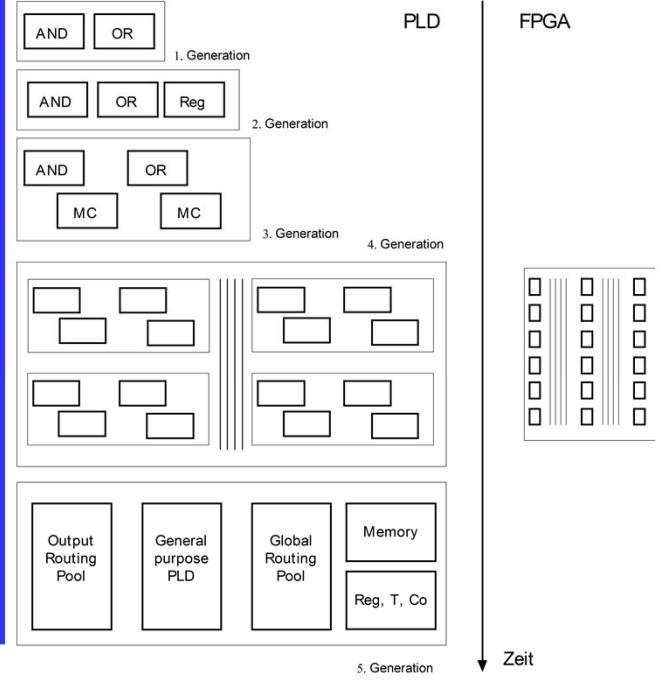
**CPLDs:**  
**Complex Programmable  
Logic Device**

**FPGAs: feld-programmier-  
bare Gate Arrays**

**CPLD** ↔ **FPGA**

Co: Counter

MC: Macro Cell



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

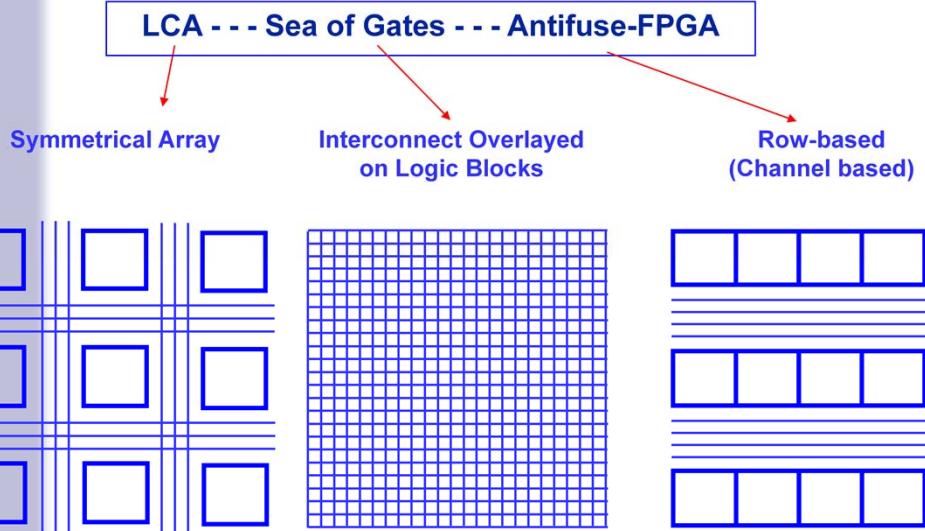
K5-2

p. 3 25.09.2011 17:48:35

## FPGA

Time to Market und Rapid Prototyping sind Notwendigkeiten, die wesentlich dazu beitragen, dass viel in Bausteine investiert wird, die ein rasches Entwickeln und Produzieren elektronischer Komponenten zulassen. Neben den CPLDs bot sich noch ein anderer Weg an, der eine weitere Steigerung der Gatteranzahl pro Chips versprach, die FPGAs (Field-Programmable Gate Arrays). Der Name drückt die Zielrichtung schon aus: FPGAs bilden die gedankliche Brücke zwischen CPLDs und Gate Arrays. Mit zunehmender Performance der Entwicklungs-Tools konnte das wesentliche Problem gelöst werden: Es wurde möglich, PC-Software anzubieten, die auch kompliziertere Routing-Algorithmen zuließ, die notwendig sind, um für Bausteine dieser Art das entsprechende Layout zu generieren. Der eindeutige Vorteil der CPLDs wie die klare symmetrische Struktur, die exakte Laufzeitbestimmungen erlaubt, wird damit zum Teil aufgehoben.

# FPGA (programmierbare Gate-arrays)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

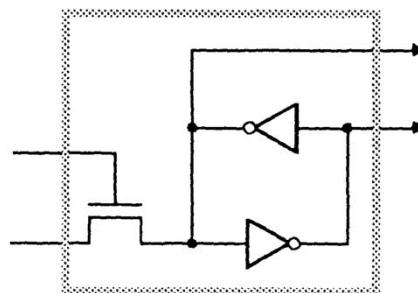
K5-2  
p. 4 25.09.2011 17:48:35

Bezüglich der Struktur lassen sich heute im wesentlichen drei unterschiedliche Typen unterscheiden (Bild oben): symmetrical Arrays, Row-based Arrays und Sea-of-Gates. Hinter den unterschiedlichen Strukturen stehen unterschiedliche Firmenphilosophien. Bezogen auf die Technologie haben sich vor allem zwei Prinzipien durchgesetzt: das Antifuse- und das SRAM-Prinzip. Bei der Antifuse-Technologie, die von Actel patentiert wurde, besitzen die Logikzellen eine sehr einfache Struktur und der Baustein eine extrem kleine Granularität. Sie sind, wie der Name schon verrät, nicht mehr reprogrammierbar. Der SRAM-Typ ist technologisch aufwendiger, verliert seine Informationen bei Abschaltung der Versorgungsspannung, muss dementsprechend jeweils nach der Zuschaltung der Versorgungsspannung mit den schaltungstechnischen Daten neu geladen werden. Die Möglichkeit jedoch, die Schaltung laufend verändern zu können, kann verständlicherweise die Entwicklungs- und Produktionszeit dramatisch reduzieren und macht darüberhinaus auch das Verändern von Schaltungen "on the Board" beziehungsweise im System möglich. Welche neuen Wege hier verfolgt werden, soll am Schluss dieses Abschnittes noch besprochen werden.

# RAM-FPGA → LCA

- ❖ 1985 eingeführt
- ❖ Basis: RAM-Zelle
- ❖ Symmetrical Array Structure

➤ einfachste RAM-Zelle



Institut für Computertechnik

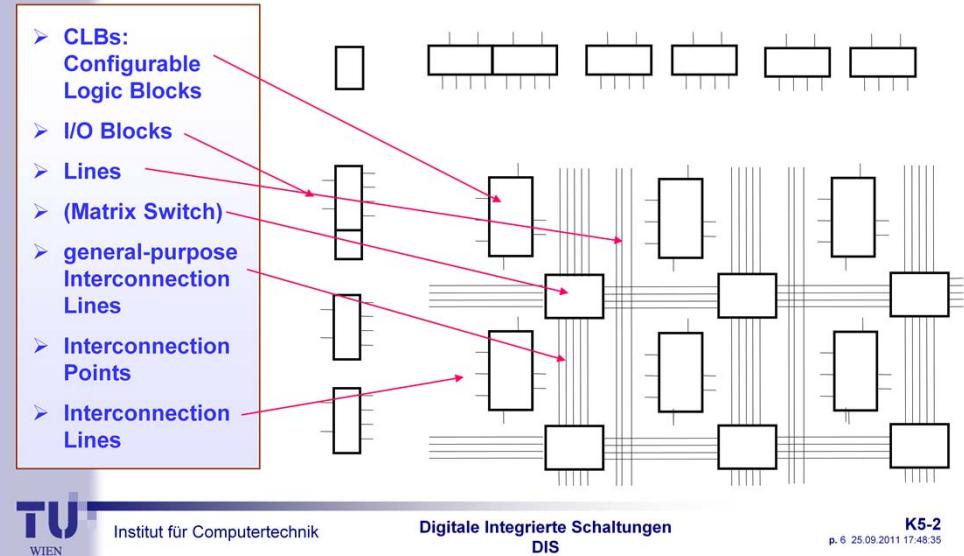
Digitale Integrierte Schaltungen  
DIS

K5-2

p. 5 25.09.2011 17:48:35

FPGA auf RAM-Basis hat Xilinx 1985 eingeführt und Logic Cell Arrays genannt, ebenfalls auf CMOS-Basis.  
Als Basiselement liegt eine einfache RAM-Zelle zugrunde.

# RAM-FPGA ➔ LCA



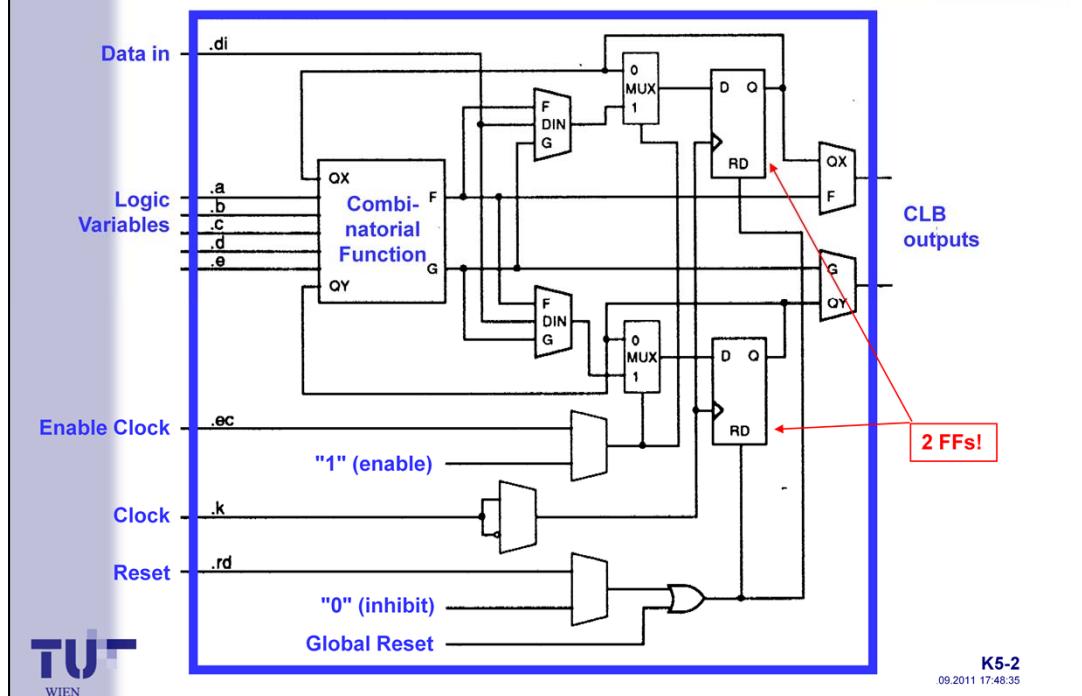
Nach der Erläuterung im Bild vorher hat sie eine symmetrical Array Structure, wie sie oben gezeigt wird.

- Programmierbar sind 3 Elemente:
- die CLBs (Configurable Logic Blocks),
- die I/O Blocks und
- die Verbindungsleitungen,

die in verschiedenen Arten vorliegen:

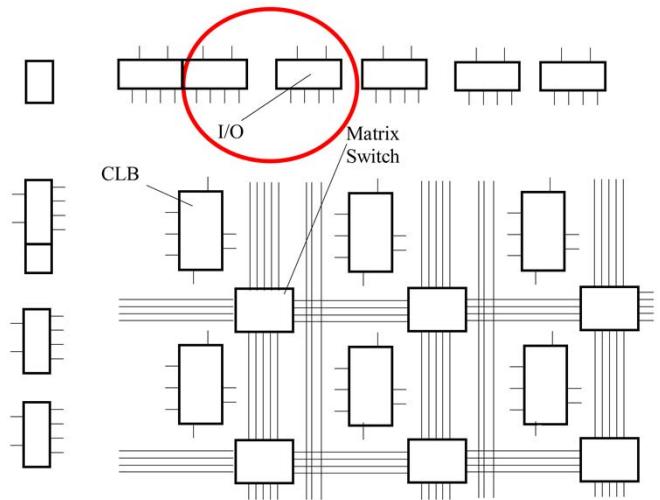
- Interconnection Lines,
- general-purpose Interconnection Lines
- horizontal long Lines und
- die vertical long Lines.

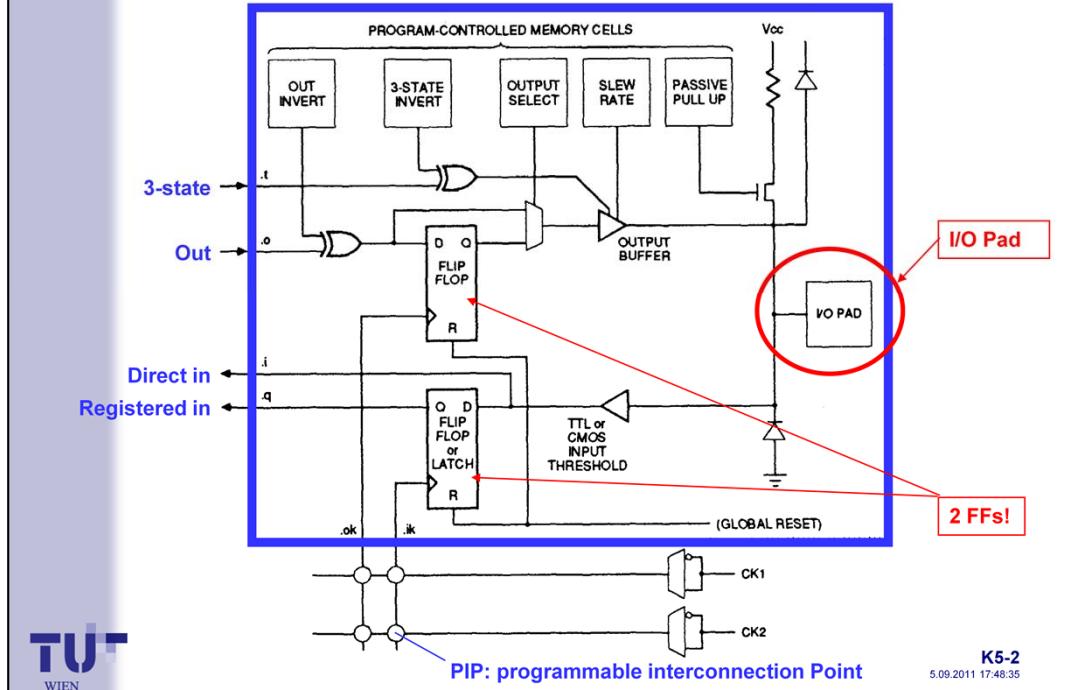
*Interconnection Lines:* An die Blöcke führen kurze Leitungen, die über *Interconnection Points* mit allen anderen vorbeiführenden gekoppelt werden können. Die *general-purpose Interconnection Lines* sind zwischen Matrizen-Schaltzellen (*Switching Matrix*) angeordnet. Um geringe Laufzeiten über größere Strecken zu erhalten, sind die *horizontal long Lines* (1 pro Reihe) und die *vertical long Lines* (2 pro Reihe). Darüber hinaus gibt es eine *global Line*, die über einen ausreichend starken Buffer getrieben wird, um für alle Speicherelemente einer Reihe als Takteingang zu dienen.



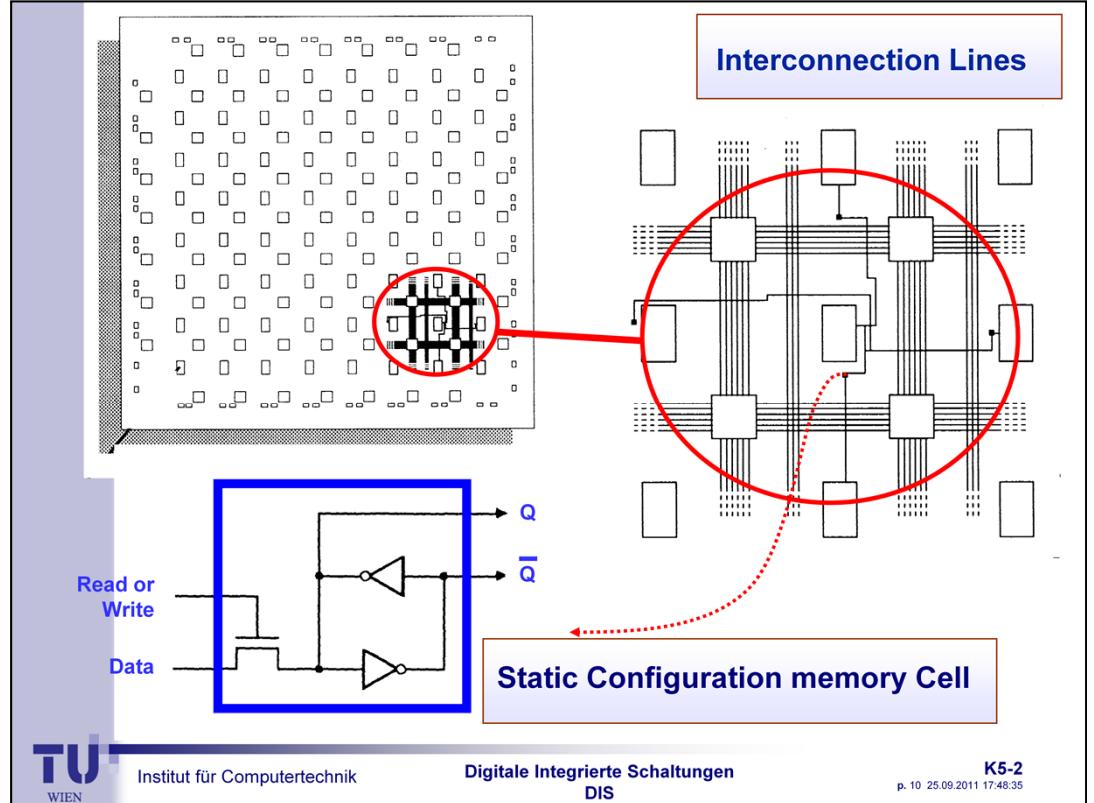
Der wesentliche Unterschied zwischen den Bausteinen der LCA-Familie liegt in der Art der CLBs und I/O-Blocks sowie der Anzahl der Blöcke und Lines. Für die XC4000-Serie gilt:  $8 \times 8 = 64$  bis  $30 \times 30 = 900$  CLBs und 64 bis 240 I/O-Blocks, wobei der kleinste Baustein (64 CLBs und 64 I/O-Blocks) 2000 Gatteräquivalenten und der größte (900 CLBs und 240 I/O-Blocks) 20.000 Gatteräquivalenten entspricht. Charakteristisch ist vor allem, dass nur wenige Register im CLB enthalten sind, im XC2000 nur eines, im XC3000 und XC4000 zwei, dafür aber einige Multiplexer und vor allem eine programmierbare kombinatorische Logik, mit der jegliche Funktion mit 4 Variablen (XC2000) oder 2 Funktionen mit jeweils 3 Variablen realisierbar sind. Die Propagation Delay (sich fortpflanzende Verzögerung) ist dabei unabhängig von der Funktion.

# RAM-FPGA → LCA





Die I/O-Blocks sind noch einfacher gehalten, trotzdem können alle Zellen als Ein- und als Ausgang eingesetzt werden, und der Ausgangs-Buffer ist ein Tristate-Treiber. Die Daten können eingangseitig über ein Latch gehalten werden, ausgangsseitig hat man darauf verzichtet. Beim XC4000 sind die Pull-up- und Pull-down-Widerstände bemerkenswert, die weitergehende Busschaltungen zulassen. Selbst die Spannungs-Anstiegsgeschwindigkeit (Slew rate) kann variiert werden.



General-Purpose Interconnect wird in horizontaler Ebene über 4 Metallverbindungen, in der horizontalen über 5 Metallverbindungen gebildet.

Wie bei fast allen programmierbaren Bausteinen liegt eine der Schwierigkeiten bei der richtigen Wahl einer effizienten Verbindung zwischen den einzelnen Blöcken [1].

Die Bausteinfamilien XC3000, XC4000, .. sind nicht nur als rein schaltungstechnische Verbesserungen zu sehen, sondern die CLBs und I/O-Blocks sind jeweils so aufeinander abgestimmt (auch die Familien unter sich), dass bei der jeweiligen Applikation eine effiziente Bausteinwahl bezüglich des Preisleistungsverhältnisses möglich wird.

Da diese Thematik speziell erst in Vertiefungsfächer der Computertechnik behandelt wird, soll es an dieser Stelle nur erwähnt werden: In der XC4000-Serie wurde die Boundary Scan Logic (IEEE1149.1) mit drei zusätzlichen Pins integriert. Damit wird auch für diesen Baustein das "Bed of Nails" überflüssig, was für Chips dieser Größenordnung selbstverständlich sein sollte.

[1] Die zweite große Schwierigkeit ist die richtige Wahl der Granularität, und die dritte Schwierigkeit besteht darin, das ausgewogene Verhältnis der Komponenten in den Blöcken zu finden. Ziel ist es, stets im Mittel einen möglichst hohen Ausnutzungsgrad der Chips zu erreichen.

# LCA

- ❖ horizontal long Lines
- ❖ vertical long Lines
- ❖ global Lines (leistungsstarke Buffer)
- ❖ Anordnung für XC4000:
  - ❖ 8\*8 bis 30\*30 CLBs bei 64 bis 240 I/O Blocks
  - ❖ 64 CLBs + 64 I/O Blocks = 2.000 Gatteräquivalente,  
900 CLBs + 240 I/O Blocks = 20.000 Gatteräquivalente

typisch seit 2000:

- ❖ 150.000 Logikzellen (Spartan-II-Chip von Xilinx, ca. 10 US\$)



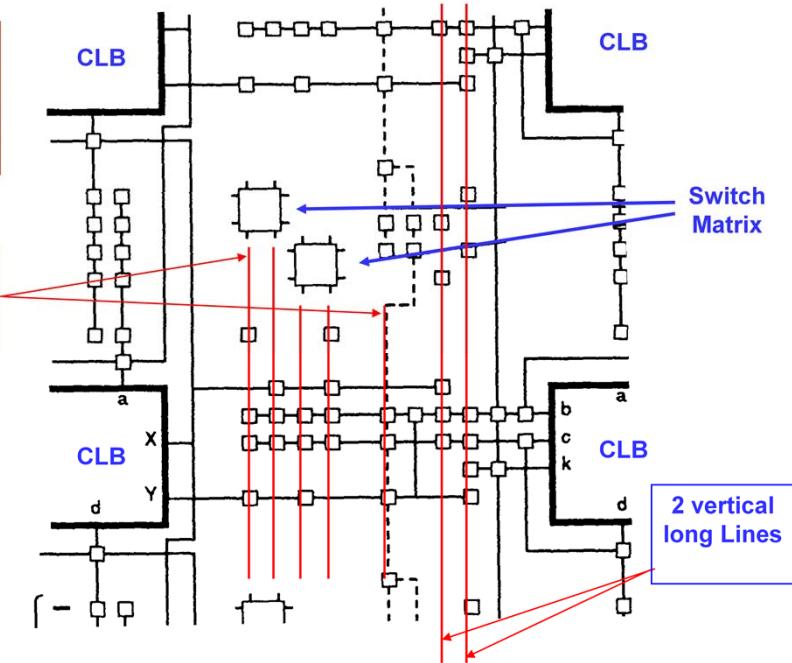
Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-2  
p. 11 25.09.2011 17:48:35

## Routing and Switch Matrix Connection

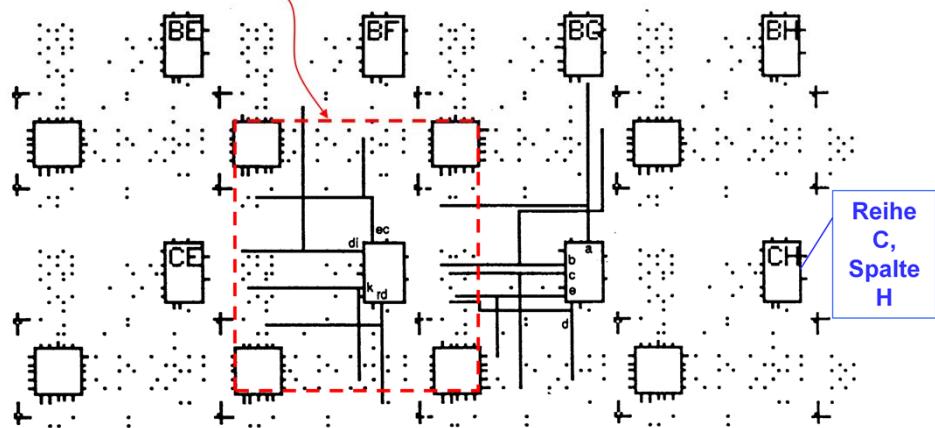
5 vertical general propose  
Interconnect between  
Switch Matrices



## Interconnect Access

### Treiberproblematik:

- Interconnect innerhalb eines Blockbereiches
- Interconnect zwischen benachbarte Blockbereiche
- Interconnect über Switching Matrices
- Interconnect über long Lines

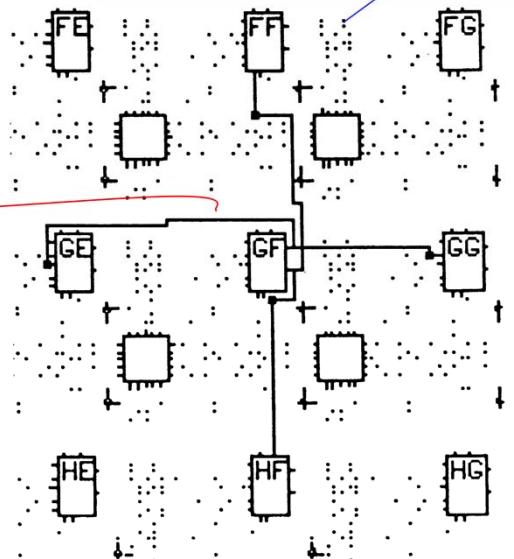


## Interconnect between adjacent CLBs

PIP: programmable Interconnect Points

### Treiberproblematik:

- Interconnect innerhalb eines Blockbereiches
- Interconnect zwischen benachbarte Blockbereiche
- Interconnect über Switching Matrices
- Interconnect über long Lines

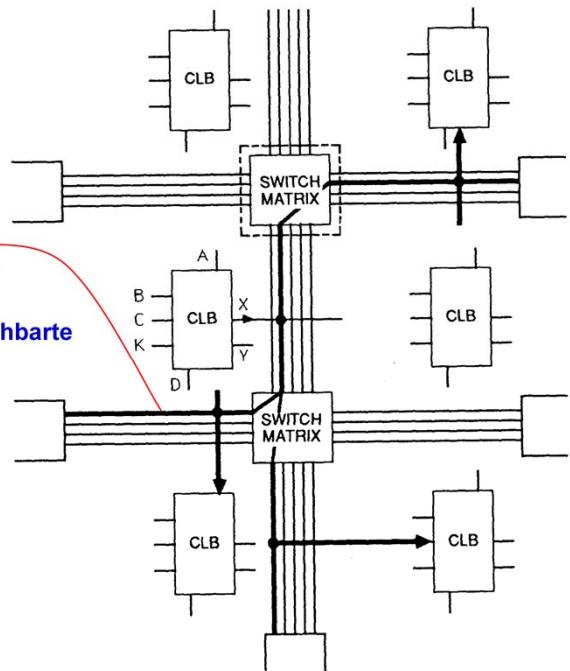


Eine typische Verbindung von einer CLB ausgehend auf 2 weitere CLBs zeigt Bild oben.

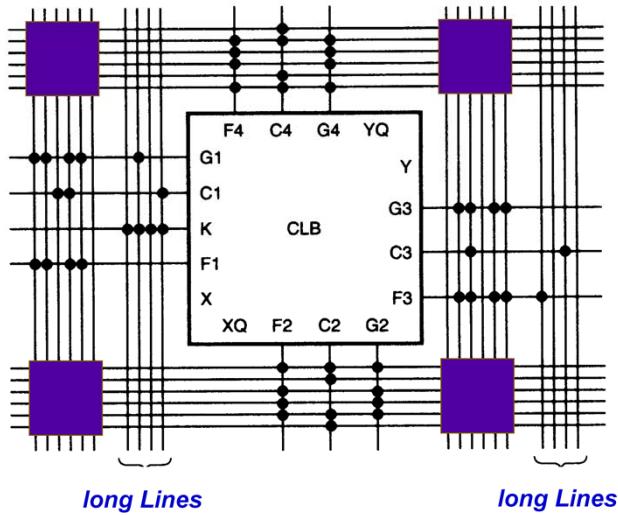
## Switch Matrix

### Treiberproblematik:

- Interconnect innerhalb eines Blockbereiches
- Interconnect zwischen benachbarte Blockbereiche
- Interconnect über Switching Matrices
- Interconnect über long Lines



## Switch Matrix



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

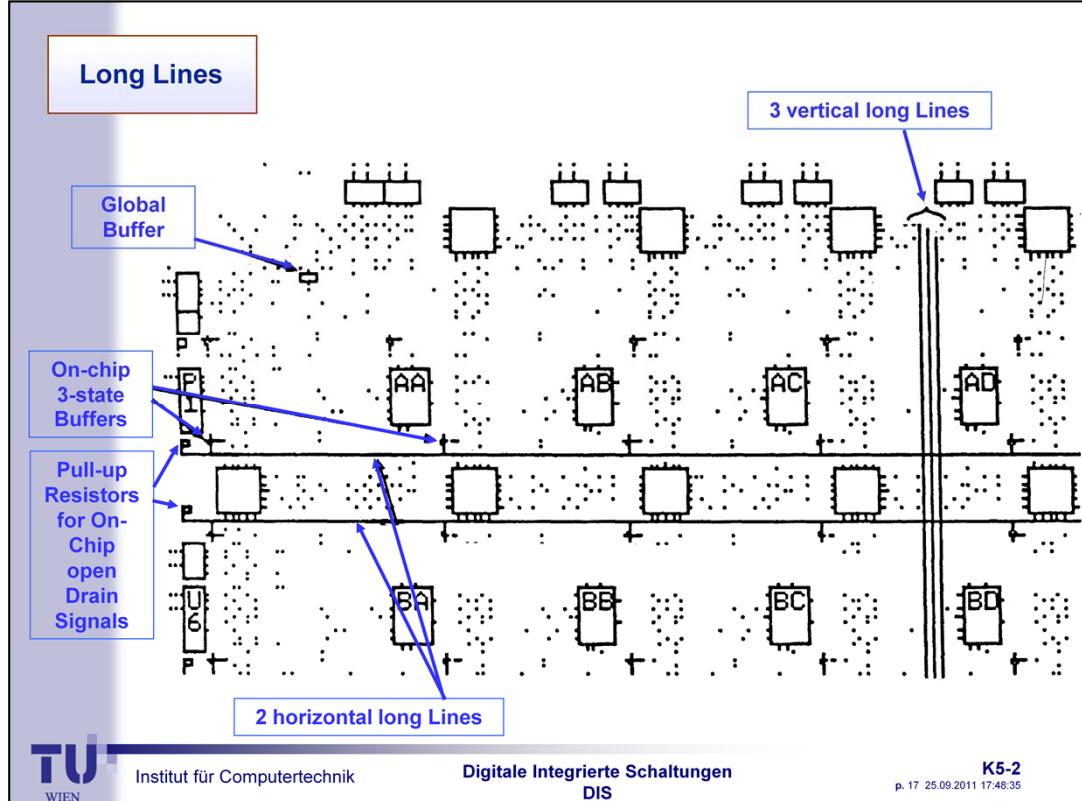
K5-2

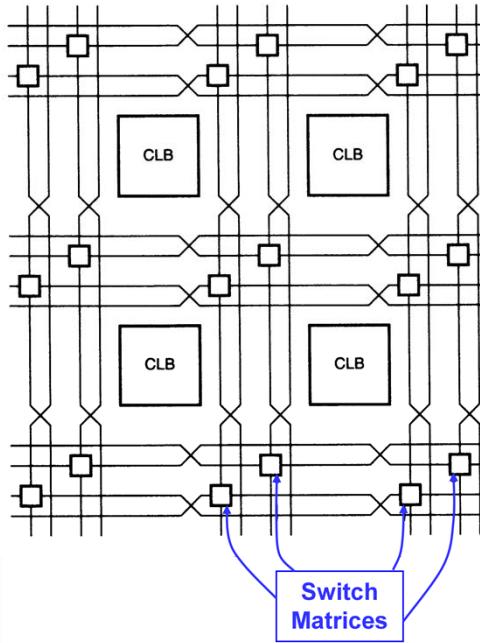
p. 16 25.09.2011 17:48:35

Hier sind jeweils 4 long Lines abgebildet, auf der anderen Darstellung nur 3, abhängig vom Bausteintyp.

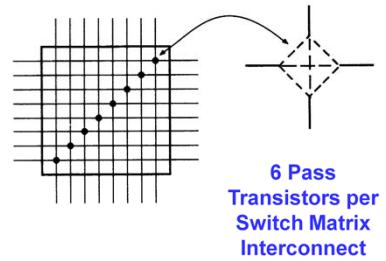
Ein entscheidender Bottle-Neck bei der Kopplung der CLBs liegt in den Verbindmöglichkeiten über die verschiedenen Lines. Hier hat man nun bei der XC4000-Familie den Aufwand drastisch gesteigert. Die Anzahl der general-purpose Interconnections Lines, das heißt, die Verbindung zwischen den Switch Matrices, wurde erhöht und horizontal wie vertikal auf 8 festgelegt; sie werden nun auch *single-length interconnect Lines* genannt.

## Long Lines





**LCA mit Double-length Lines**

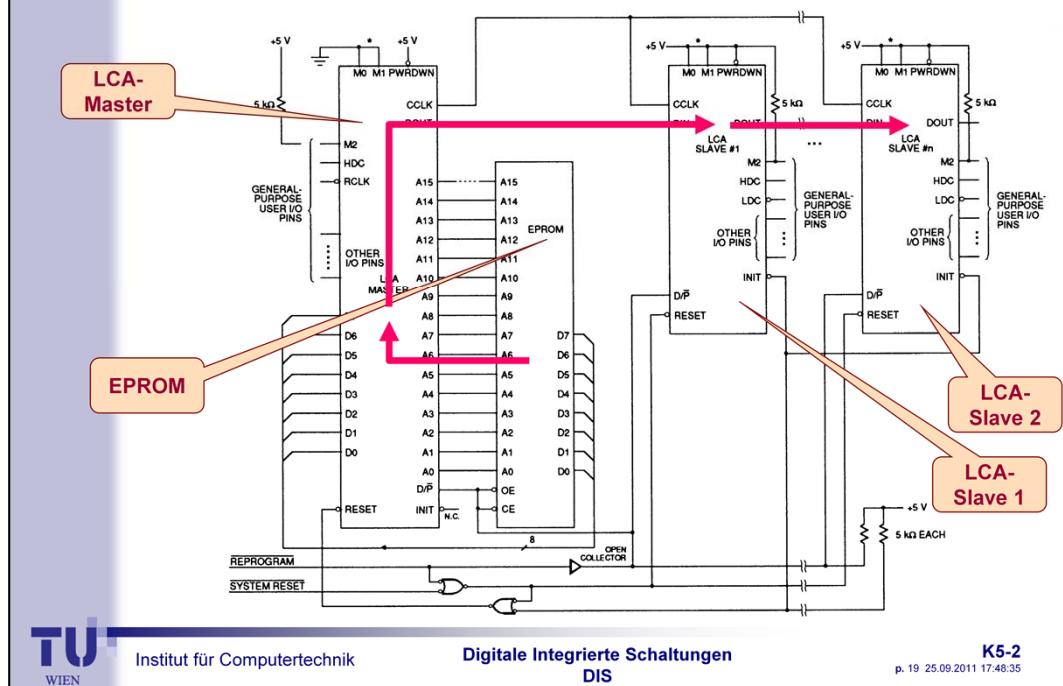


6 Pass  
Transistors per  
Switch Matrix  
Interconnect  
Point

Neben den single-length Lines gibt es nun noch double-length Lines. Zwischen den CLBs sind jeweils 4 von ihnen angeordnet. Sie wurden aufgrund der Erkenntnis eingeführt, dass es relativ viele Verbindungen über ein CLB hinaus gibt, die zu keinen zusätzlichen Laufzeitverzögerungen führen dürfen. Diese Leitungen werden deshalb jeweils am nächsten Baustein vorbeigeschleift.

Die Verbindungsmöglichkeiten in den Switch Matrices wurden vermindert, wodurch die Routing-Möglichkeit pro Interconnection Point damit reduziert ist, und es sind nun mehr nur noch 6 Pass Transistors pro Interconnection Point integriert, die über eine Configuration Memory Cell programmiert werden können. Damit lassen sich die 6 Wege entsprechend Bild oben schalten (die gestrichelten Linien zeigen die Wege). Bei den Familien XC2000 und XC3000 konnten von 8 Interconnection Points jeweils 5 Wege gleichzeitig geschaltet werden.

## Master Mode Configuration with Daisy chained Slave Mode Devices



Institut für Computertechnik

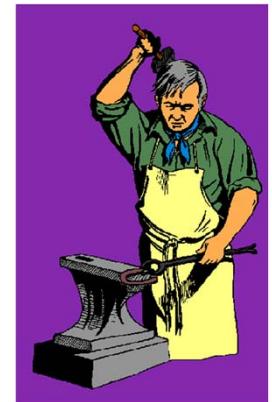
**Digitale Integrierte Schaltungen**  
**DIS**

K5-2

p. 19 25.09.2011 17:48:35

Das LCA muss beim Einschalten der Versorgungsspannung jeweils mit den schaltungstechnischen Daten geladen werden. Dies erfolgt normalerweise über ein Speicherbaustein im Parallel- oder Serial-Mode.

- ❖ **Schaltungseingabe**
- ❖ **Übersetzung des Entwurfs (Compilierung)**
- ❖ **funktionelle Simulation**
- ❖ **Placement- und Routing-Funktion**
- ❖ **Zeitsimulation**
- ❖ **Erstellung der Ladedatei**



Wie schon erwähnt, ist für die Entwicklung solcher Bausteine ein umfangreiches Entwicklungssystem notwendig, was folgende Komponenten enthalten muss:

- Schaltungseingabe,
- Übersetzung des Entwurfs (Compilierung),
- funktionelle Simulation,
- Placement- und Routing-Funktion,
- Zeitsimulation und
- Erstellung der Ladedatei.

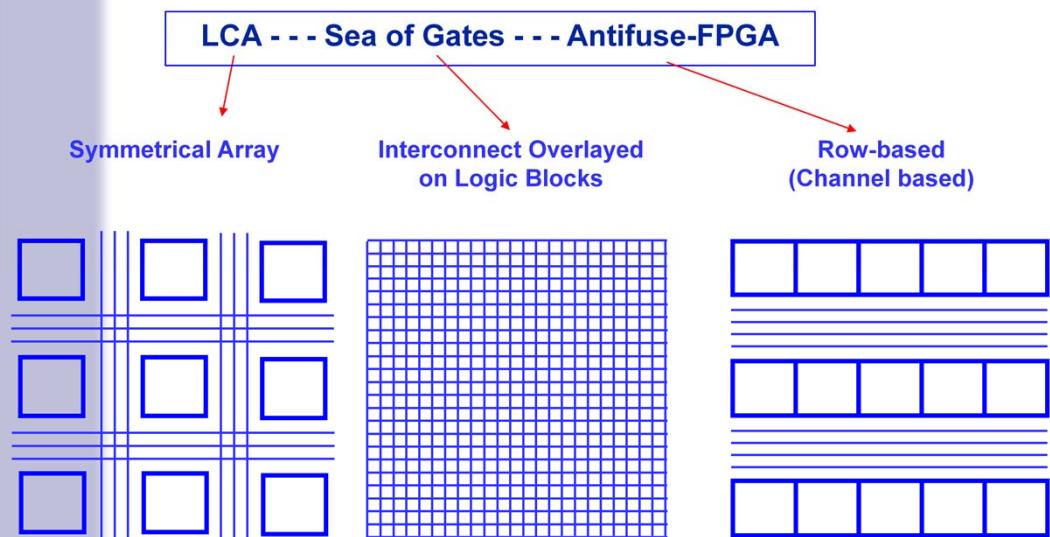
Die Firmen bieten zwar geschlossene Entwicklungssysteme an, die diese einzelnen Einheiten abdecken, doch da man mehr und mehr dazu übergeht, den Entwurf in VHDL zu schreiben, ist auch auf die Kompatibilität derartiger Programmsysteme zu achten.

Der kritische Part des Entwurfs sind das Placement und Routen. Das Programm verfährt heutzutage allgemein nach heuristischen Optimierungsalgorithmen, die man über Optionseingaben, "Constraints-Dateien", Reihenfolge von Netzeingaben usw. beeinflussen kann. Einerseits schränken diese Vorgaben die Freiheitsgrade des Optimierungsprogramms ein (reduzieren also damit die Chance, zu einer evtl. minimalen Chip-Fläche zu gelangen), können aber andererseits helfen, kritische Stellen zu entschärfen.

Die heuristischen Algorithmen haben noch einen anderen Effekt: Jede Compilierung des Programms kann ein anderes Ergebnis liefern. Es gibt deshalb bei manchen Programmen von vornherein die Option, dass mehrere Durchläufe getätig werden sollen, um nachher zwischen den besten entscheiden zu können.

Die hier genannten Angaben sind wegen des Aufwandes natürlich nicht vollständig. Sie sollen nur einen Eindruck vermitteln, wie komplex die Bausteine gegenüber den einfachen PALs geworden sind.

# FPGA (programmierbare Gate-arrays)

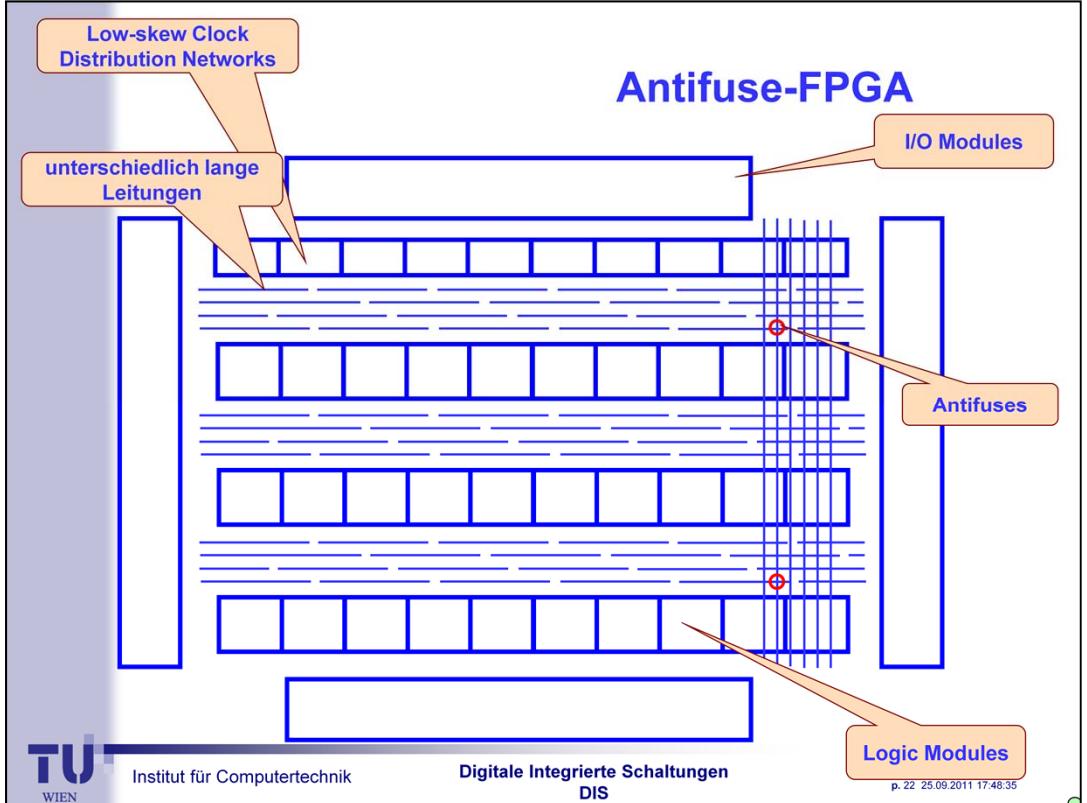


Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-2

p. 21 25.09.2011 17:48:35

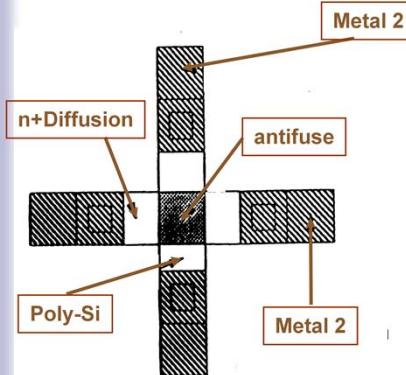


Entwickelt wurde die Technologie von Actel. Verwendet wird sie jedoch heute noch von anderen Firmen wie Crosspoint Solutions und QuickLogic, auf deren Realisierung jedoch nicht näher eingegangen werden soll, da sie sich nicht wesentlich von dem Prinzip von Actel unterscheiden.

Entsprechend der Darstellung auf der letzten Folie verwendet Actel eine Row-based-Struktur, um die I/O-Module angeordnet sind (Bild oben). Die horizontal eingezeichneten, parallelen Leitungen sind unterschiedlich segmentiert, um einen möglichst hohen Auslastungsgrad zu erhalten. An den Kreuzungspunkten zweier Bahnen sind die Antifuse-Programmierzellen angeordnet. Die Art der Logikmodule hängt von der Platzierung des Moduls in der Reihe und von dem Chiptyp ab.

.....

Low-skew Clock: die Laufzeitunterschiede des Clocksignals zwischen verschiedenen mit dem Clock betriebenen Teilen wird minimiert. Bei moderner Technik muss der Skew unterhalb von wenigen 10 ps liegen. Es ist eine echte Herausforderung für den Entwickler, entsprechende Clock-trees zu gestalten.



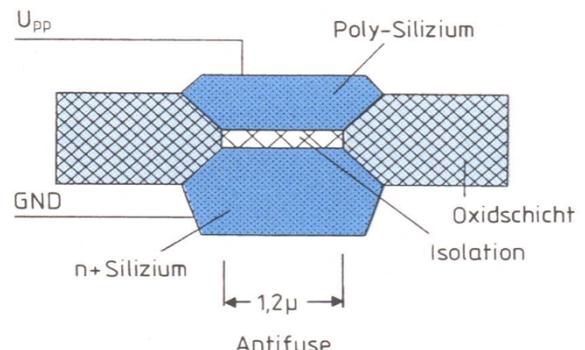
- Fuse-Techn. (z. B. PAL) vor der Programmierung leitend
- Antifuse: nichtleitend
- Isolation wird weggebrannt, Si-Schichten wachsen zusammen

## Antifuse-Technologie

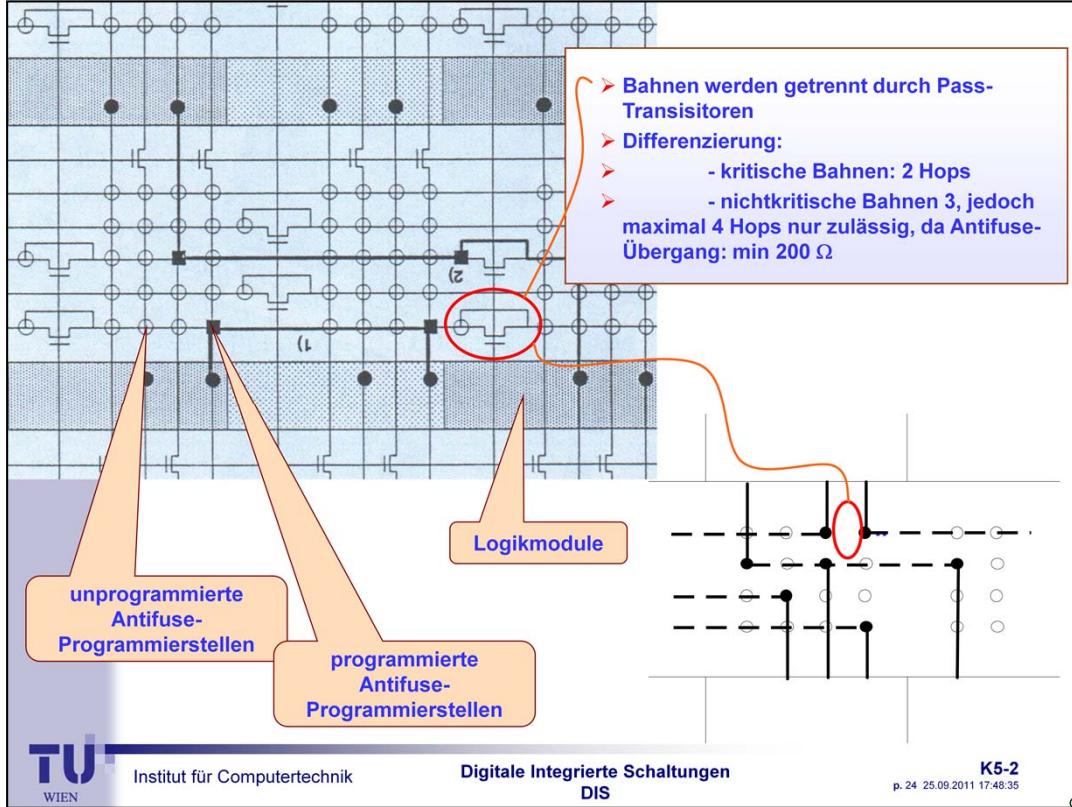
<b>Programmierspannung <math>U_{PP}</math>:</b>	21 V
<b>Programmierzeit:</b>	< 5 ms
<b>Programmierstrom:</b>	< 10 mA
<b>Übergangswiderstand programmiert:</b>	200 – 500 $\Omega$
<b>Übergangswiderstand unprogrammiert:</b>	> 100 M $\Omega$
<b>Antifuse-Geometrie:</b>	$1,2 \times 1,6 \mu\text{m}^2$



Institut für Computertechnik



Die Antifuse-Zelle selbst besteht im wesentlichen aus drei Schichten: die Basisschicht sowie die obere Schicht sind Silizium, die mittlere ein Dielektrikum. Die Programmierung erfolgt über eine relativ hohe Programmierspannung von etwa 18 V sowie einen Einbrennstrom von ca. 5 mA, der ausreicht, das Dielektrikum zum Schmelzen zu bringen und einen dauerhaften Kontakt herzustellen. Das Silizium ist an die Metalleiter angeschlossen, so dass die vollständige Antifuse-Einheit einen maximalen Wert von 200 bis 500 m $\Omega$  bildet.

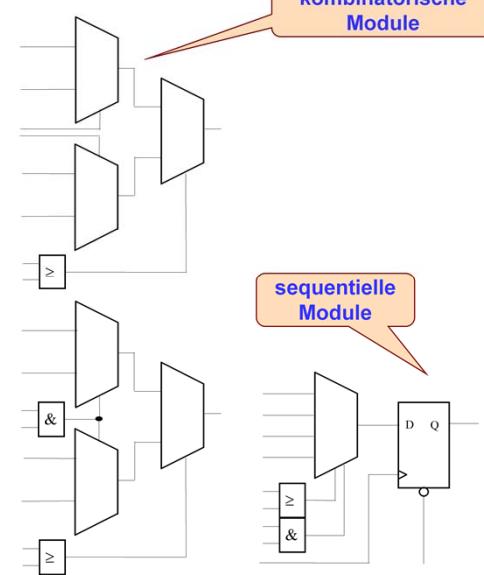


Da die Signalwege durch das Routing effizient festgelegt werden, sind die Laufzeiten über die Signalpfade natürlich nicht vorher bestimmbar. Dafür steht eine BITE [1] (Built-in-Test Equipment) mit entsprechender Testsoftware zur Verfügung. Um die Laufzeiten kurz zu halten und effizient festzulegen, hat Actel verschiedene Regeln für das Routing festgelegt. Kritische Pfade dürfen nur 2 (gebrannte) Antifuse-Zellen (Hops) enthalten, die meisten 3, das Maximum sind 4 Hops.

[1] BITE bedeutet, dass im Chip selbst, Testschaltungen integriert sind, die mit dem Test-Equipment außerhalb der Schaltung eine Testeinheit bilden. Ein BITE kann auch für Online-Tests konzipiert sein.

# Antifuse-FPGA

- kleine Granularität
- Module für kombinatorische und Module für sequentielle Schaltungen
- FFs auch mit mehreren kombinatorischen Modulen zu realisieren
- kleinste einfache Zellen (vor allem MUX, auch logische Funktionen werden mit MUX realisiert)
- größere haben ein FF
- keine feste Laufzeiten
- komplexe Tools
- optimal heute: VHDL



Institut für Computertechnik

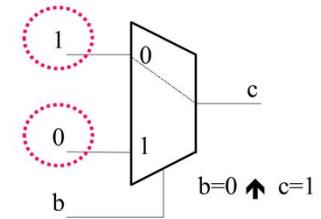
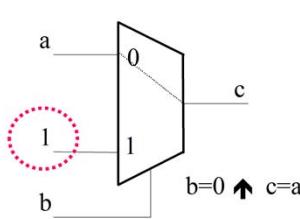
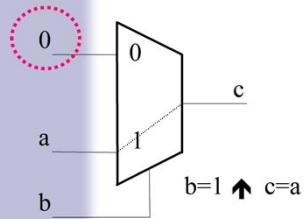
Digitale Integrierte Schaltungen  
DIS

K5-2

p. 25 25.09.2011 17:48:35

Der einfachste Chip hat nur das einfachste Modul, das auf drei steuerbaren Multiplexer basiert (die mittleren beiden Darstellungen), mit denen alle elementaren logischen Funktionen realisiert werden können, die auf der nächsten Folie gezeigt werden. Komplexere Chips enthalten auch schon vorgefertigte FFs, wie es die rechte Darstellung zeigt.

## Verwendete MUX-Technik



$$c = a \wedge b$$

0	a	b	c
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1

$$c = a \vee b$$

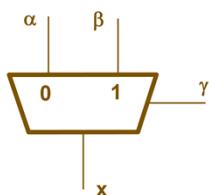
1	a	b	c
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$c = \neg b$$

b	c
0	1
1	0

Entwerfen Sie die Schaltung mit ausschließlich zweitorigen Mux-Bausteinen:

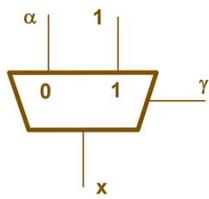
**zweitoriger Mux-Baustein:**



$$f = \bar{d}\bar{c}\bar{a} \vee d\bar{c}a \vee \bar{c}ba \vee \bar{d}ca$$



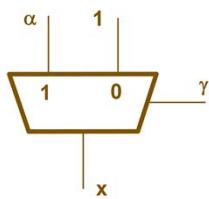
ODER



$\gamma$	$\alpha$	$x$
0	0	0
0	1	1
1	0	1
1	1	1

$$x = \alpha \vee \gamma$$

ODER

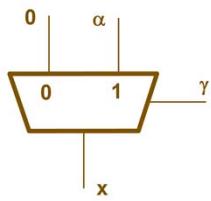


$\gamma$	$\alpha$	$x$
0	0	1
0	1	1
1	0	0
1	1	1

$$x = \alpha \vee \bar{\gamma}$$



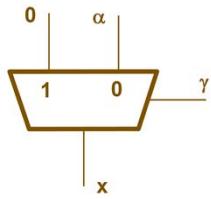
UND



$\gamma$	$\alpha$	$x$
0	0	0
0	1	0
1	0	0
1	1	1

$$x = \gamma \alpha$$

UND

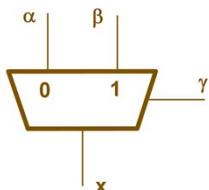


$\gamma$	$\alpha$	$x$
0	0	0
0	1	1
1	0	0
1	1	0

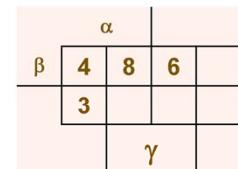
$$x = \bar{\gamma} \alpha$$



$$x = f(\gamma, \beta, \alpha)$$



$\gamma$	$\beta$	$\alpha$	$x$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



$$x = \bar{\gamma} \alpha \vee \gamma \beta$$

Steuereingang für Schalter

Ausgangsgleichung:

$$f = \bar{d}\bar{c}\bar{a} \vee d\bar{c}a \vee \bar{c}ba \vee \bar{d}ca$$

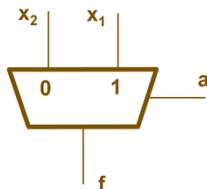
$$f = x_1 a \vee x_2 \bar{a}$$

$$\text{mit } x_1 = d\bar{c} \vee \bar{c}b \vee \bar{d}c$$

$$\text{und } x_2 = \bar{d}\bar{c}$$

}

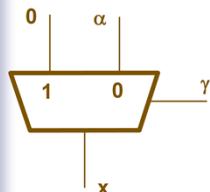
Herausgreifen einer Variablen  
hier z. B. a bzw.  $\neg a$   
:= Schalterwirkung



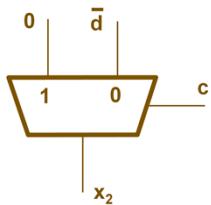
Entwicklung der  
LUT (Look-up Table) für  $x_2$   
hier: UND

$$x_2 = \bar{d} \bar{c}$$

$$x = \bar{\gamma} \alpha$$



$$x_2 = \bar{d} \bar{c}$$



- Wahl der Variablen frei wählbar

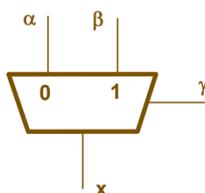
$$x_1 = d\bar{c} \vee \bar{c}b \vee \bar{d}c$$

Reduktion des Ausdrucks  
auf 2 Terme über eine  
ODER-LUT

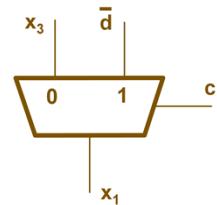
Lösung Schritt 3

$$x_1 = (d \vee b)\bar{c} \vee \bar{d}c = x_3\bar{c} \vee \bar{d}c$$

$$x = \bar{\gamma} \alpha \vee \gamma \beta$$



$$x_1 = x_3\bar{c} \vee \bar{d}c$$

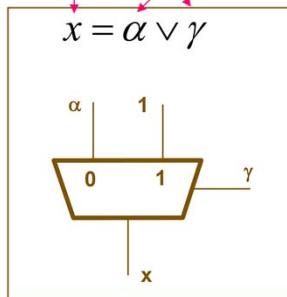


Aus:

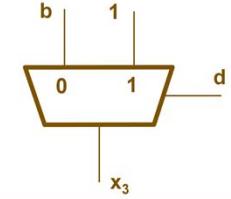
$$x_1 = (d \vee b)\bar{c} \vee \bar{d}c = x_3\bar{c} \vee \bar{d}c$$



$$x_3 = d \vee b$$



$$x_3 = d \vee b$$



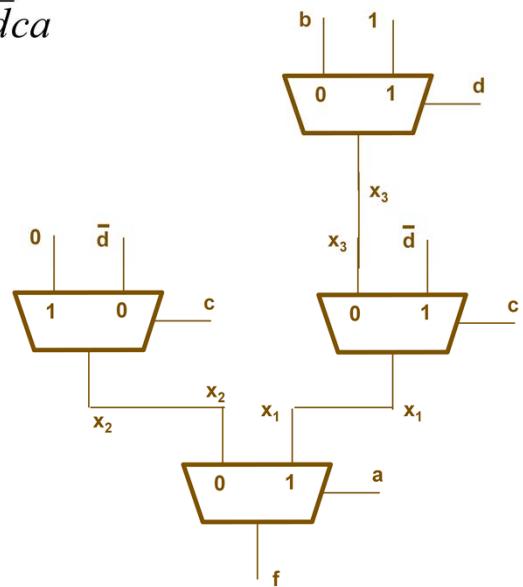
$$f = \bar{d}\bar{c}\bar{a} \vee d\bar{c}a \vee \bar{c}ba \vee \bar{d}ca$$

$$f = x_1a \vee x_2\bar{a}$$

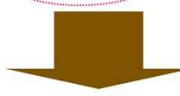
mit  $x_1 = x_3\bar{c} \vee \bar{d}c$

und  $x_2 = \bar{d}\bar{c}$

und  $x_3 = d \vee b$



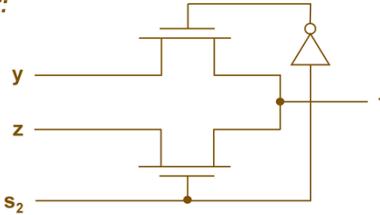
$$f = (\bar{s}_1 w \vee s_1 x)(\bar{s}_2 y \vee s_2 z)(\bar{s}_3 \vee \bar{s}_4)(s_3 \vee s_4)$$



nach ACTEL:

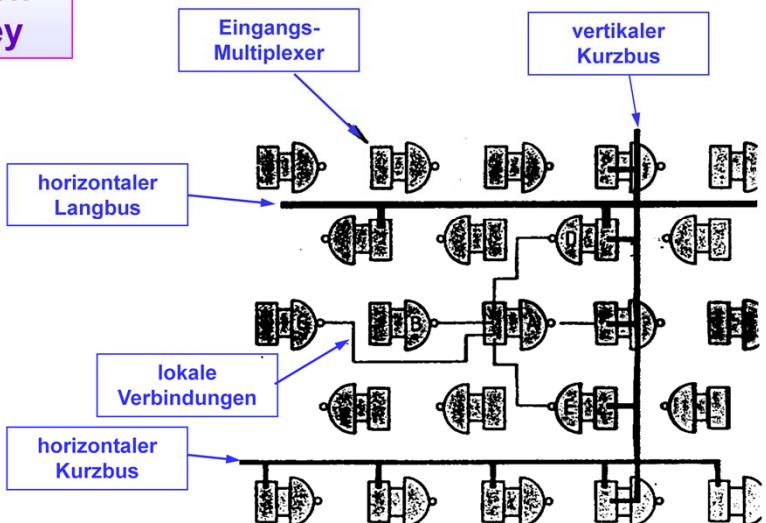
702 logische Funktionen zu realisieren

Mux-Architektur:



# Sea of Gates (Plessey)

## ERA von Plessey



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-2

p. 37 25.09.2011 17:48:35

Nach Vorstellung zweier unterschiedlicher FPGAs sollen noch ein paar Statements abgegeben werden, deren detailliertere Erläuterung zu viel Aufwand bedeuten würde, die jedoch andererseits so wichtig sind, dass sie hier vorgebracht werden sollen.

Bisher nur kurz angedeutet wurden die Bausteine auf der Basis der Sea-of-Gates-Architektur. Entwickler des Produktes - genannt ERA - ist Plessey. Sämtliche Zellen sind identisch. Beim einfachsten Baustein kann die Zelle entweder als einfaches NAND mit zwei Eingängen oder als Latch verwendet werden. Jede Zelle ist symmetrisch mit vier anderen verbunden. Auch sind sie verbindungsmäßig so optimiert, dass sich einfach zwei Zellen zu Master-Slave-FFs zusammenschließen lassen. Damit lassen sich in idealer Weise beispielsweise Automaten aufbauen. Wie im LCA sind auch hier lange und kurze Busse vorgesehen. Auch werden die Konfigurationen von speziellen RAM-Bits gesteuert. Das Laden der Daten erfolgt entsprechend der LCAs über einen speziell angeschlossenen Speicherbaustein (EPROM, ROM, ..) oder direkt über einen Mikroprozessor.

# Sea of Gates (Plessey)

- ❖ jede Zelle: einfaches NAND oder Latch
- ❖ jede Zelle symmetrisch mit vier anderen Zellen verbunden
- ❖ zwei Zellen: 1 Master-Slave-FF
- ❖ Speicherung über RAM-Zellen (ähnlich wie beim LCA  $\Leftrightarrow$  ROM)
- ❖ ideal für Automaten



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-2  
p. 38 25.09.2011 17:48:35

# Bemerkungen zu FPGAs

- ❖ FPGAs: ideal, auch wenn nur ein Teil benötigt wird
- ❖ Kopierschutz (bei LCA nur eingeschränkt)
- ❖ zur Erhöhung der Leistung:
  - nun bis zu 4 FFs
  - bis zu 4 Look-up Tables
  - Carry-Generatoren
  - Treiber für hohe Lasten
  - Pull-up-, Pull-down-Widerstände
- ❖ Tools werden immer denen des Full-Custom-Designs ähnlicher bzw. nur noch VHDL, Verilog o. ä.
- ❖ Hardware-Makro ↔ Software-Makro



- "Was macht es aus, wenn nur vielleicht 10% der Zelle genutzt werden? Hauptsache die Kosten stummen."
- Look-up-Table (LUT): vollständige Wahrheits- oder Übergangstabelle wird abgelegt; in LCA/FPGAs: die LUT-Eingangsvariablen steuern einen Multiplexer und bilden damit die Adresse zum Auslesen der SRAM-Programmierzellen; die LUT hat nur einen Ausgang (Look-up Table: Multiplexer-Einheit)

---

## Ergänzende Bemerkungen zu FPGAs und dem entsprechenden Design

- Um die Entwicklungskosten möglichst stark einzuschränken, setzt man heute meist komplexe Bausteine ein, auch wenn sie nur zu 5 oder 10% genutzt werden. Damit werden Schaltungen im allgemeinen einfach. Das Nachbauen der Print-Platten muss trotzdem verhindert werden. FPGAs reduzieren schon die Möglichkeit einer einfachen Kopie. Enthält der FPGA darüberhinaus noch einen richtigen Kopierschutz, wird ein Nachbau langwieriger und vor allem teuer.
- Der Aufwand eines Schaltungsdesigns wird oft drastisch erhöht, wenn in der Schaltung ein (für eine rein synchrone Schaltung) oder zwei Clocks (für das Operations- und das Steuerwerk eines Mikroprozessors beispielsweise getrennt) gefordert werden, oder wenn ein gemeinsame Reset-Leitung zugrunde gelegt werden soll, und dies nicht in der Struktur des programmierbaren Baustein berücksichtigt ist. Dann muss nämlich bezüglich der Treiber und Verzögerungen ein relativ großer Aufwand betrieben werden, da bei hohen Taktraten eine symmetrische Baumstruktur der Taktverteilung wie auch der Reset-Leitung eine Grundregel darstellt.
- Ähnliche Überlegungen gelten für Fast-Carry-Logikeinheiten, Schieberegister- oder RAM-Strukturen (um Off-Chip-Lösungen wegen der Verzögerungszeiten zu vermeiden), Statusregisterseinheiten usw.
- Diese Überlegungen gehen bei manchen Firmen so weit, dass sie ihre Blocks intern 4-bit-weise orientieren, das heißt, intern in einem Block nicht nur vier FFs anordnen, sondern auch 4 Look-upTables, entsprechende Carry-Generatoren usw.
- Immer interessanter werden Bausteine, die direkt zum Treiben bestimmter Lasten verwendbar sind. Hier zählen nicht nur die Größe des Ausgangsstromes, sondern genauso, ob die Ansteiggeschwindigkeit des Ausgangsstromes reduziert werden kann, Tristate-Treiber zur Verfügung stehen, Pull-up- oder Pull-down-Widerstände im Baustein integriert sind usw.

# Einsatz von FPGAs



## HW mit Mikroprozessoren

### Vorgehensweise

1. Spezifikation des Systems
2. Entwurf (und Simulation) der Software
3. Entwicklung der Mikroprozessorschaltung
4. Entwicklung der peripheren Schaltung
5. Glue Logic  $\Leftrightarrow$  FPGA

- iterativer Prozess -

**Glue Logic kann in Grenzen bis zum Schluss  
geändert werden.**



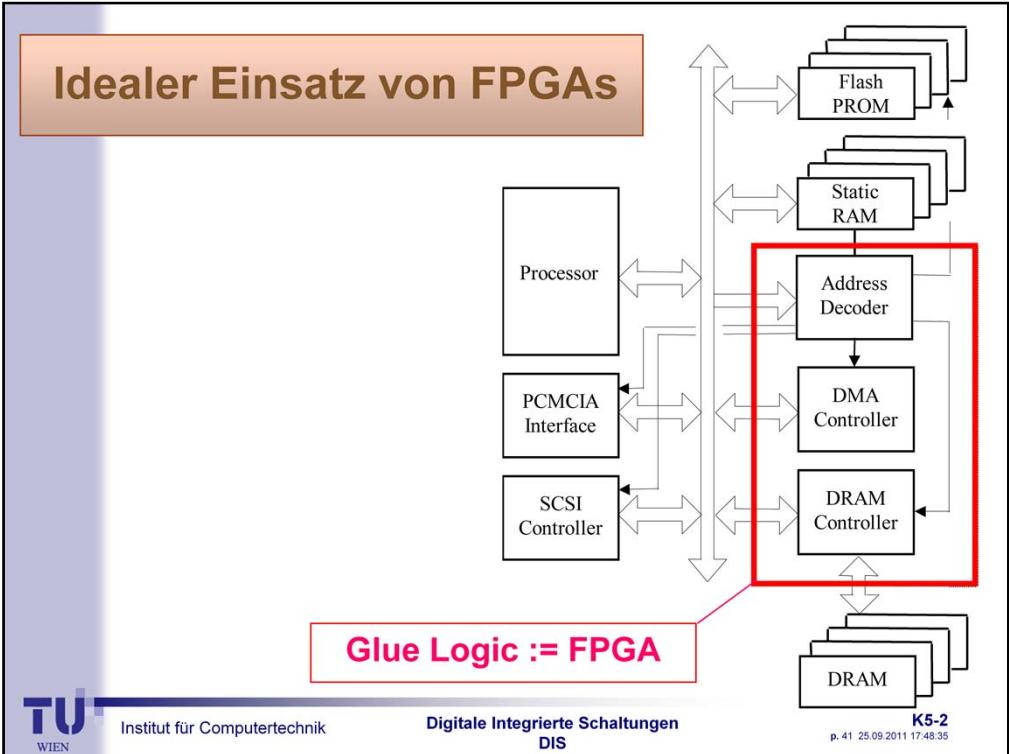
Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K5-2  
p. 40 25.09.2011 17:48:35

- Das Entwickeln und Testen von FPGAs erfolgt mehr und mehr gemäß den Prinzipien von Mikroprozessoren. So gibt es nicht nur logische Simulatoren und Laufzeitsimulatoren, sondern ebenfalls, was man von der Mikrocomputertechnik sehr gut kennt, In-Circuit-Debugger, die ein effizientes Fehlersuchen in der Schaltung ermöglichen.
- In der Literatur werden die Begriffe "Hardwaremakro" und "Softwaremakro" verwendet. Diese werden, bezogen auf unterschiedliche Produkte, unterschiedlich definiert. Bei Actel- Produkten bedeutet ein Hardwaremakro zwei gekoppelte kombinatorische Blocks, die sich nach außen wie ein sequentieller Block verhalten. Software Makros sind beliebig zusammengesetzte Blocks beispielsweise zu einer ALU, einem Automaten usw. Bei Xilinx bedeuten Hardwaremakros Beschreibungen von beliebigen Hardware-Designs, die bis zum Placement und Routing spezielle Informationen enthalten, während die Bedeutung von Softwaremakros der von Actel entspricht.
- *Partitioning* verwendet Xilinx für das Aufteilen in verschiedene I/O Blocks und CLBs, was noch nicht zum Placement zählt (Partitioning ist somit eine Umstrukturierung der Eingabedarstellung in eine I/O-Block- und CLB-gerechte Darstellung).

## Idealer Einsatz von FPGAs



### Typische Anwendung eines FPGAs

Ein typisches Anwendungsfeld von PALs, PALs war bisher vor allem die Glue Logic. Die Einbeziehung von DMA Controller, USARTs, Interrupt Controller, vor allem, wenn ihnen eine größere Funktionalität zugedacht war, war immer mit zusätzlichen Chips verbunden. Betrachtet man dagegen heutige Layouts, fällt auf, dass nur noch wenige Chips auf einem Board sind, fast die gesamte Fläche wird für die Leitungsführung genutzt (vor wenigen Jahren war dies exakt umgekehrt, fast die meiste Fläche wurde von den Chips aufgebraucht). Der Grund ist einfach: Wenn sehr hoch getaktet werden soll, wenn die Produktions- und Wartungskosten möglichst günstig sein sollen, muss ein hoher Integrationsgrad erreicht werden. Exakt das wird mit FPGAs erreicht. Nicht nur die Glue Logic, sondern auch möglichst viele andere Einheiten eines Computers werden in den FPGA gepackt, bis hin zu Dual-Port RAMs. Dass der Baustein damit eine große Pin-Zahl benötigt, ist verständlich. Auf der anderen Seite können damit aber viele Hardwarefunktionen wiederum im Baustein selbst abgespielt werden, was hohe Taktraten ermöglicht.

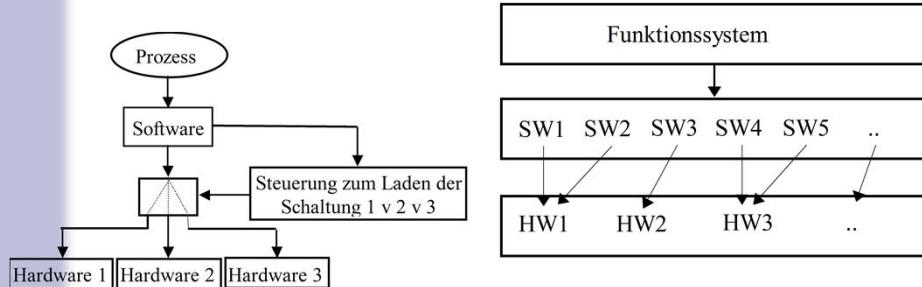
Bei der Entwicklung von Hardwaresystemen mit integrierten Prozessoren wird man wie folgt vorgehen:

1. Spezifikation des Systems,
2. Entwurf (und Simulation) der Software,
3. Entwicklung der Mikroprozessorschaltung,
4. Entwicklung der peripheren Schaltung und zuletzt die
5. Glue Logic,

für die sich FPGAs schon aus schaltungstechnischer Sicht in idealer Weise eignen, da sie aus vielen einzelnen Zellen besteht, die die anderen Chips miteinander verbinden. Doch auch in anderer Hinsicht eignen sich gerade dafür FPGAs: Da die Entwicklungsschritte in der Praxis nicht streng aufeinander folgen, sondern sich überlappen und auch iterativ bearbeitet werden, ist die Glue Logic der Schaltungsteil, der am wahrscheinlichsten Änderungen unterworfen ist. Hat man dann einen FPGA, ist die Änderung noch relativ spät möglich, bei den nachträglich programmierbaren sogar noch nach der Produktion (!).

**Basis:**

- Ladevorgang nicht mehr im s-Bereich, sondern im ns-Bereich
- Teilbereiche ladbar
- sehr hohe Gatterzahl



In der Werbung und den Datenblätter wird bei Wiederprogrammierbarkeit von FPGAs auf den großen Vorteil der reduzierten Entwicklungs- und Produktionszeiten hingewiesen. Das ist eindeutig richtig, doch es ergeben sich darüberhinaus noch andere weitaus interessantere Aspekte, die völlig neue Wege eröffnen.

Der Entwurf eines Prozessors wirft stets die Frage nach der Optimierung des Systems auf: Flexibilität auf der einen Seite, Verarbeitungsgeschwindigkeit auf der anderen Seite lassen sich kaum optimal unter einen Hut bringen. Entweder erhält man einen Baustein, der relativ langsam die unterschiedlichsten Aufgaben bewältigen kann, oder man produziert eine schnelle Maschine, die jedoch nur für spezifische Aufgaben ausgelegt ist. Mikroprozessoren der flexiblen Kategorie sind für den Einsatz in PCs geeignet, Prozessoren der anderen Kategorie sind beispielsweise die Signal- oder Grafikprozessoren. In diesen Bereichen wird sich auch in den nächsten Jahren wenig bewegen, abgesehen davon, dass die Einheiten immer leistungsfähiger werden. Denkt man nun aber an kleinere Prozesse, wie Steuerungen jeglicher Art, von der Industrieautomatisierung angefangen bis hin zur weißen und braunen Ware, bei denen die Stückzahlen der Prozessoren enorme Werte darstellen, jedoch aufgrund der unterschiedlichen Anforderungen die unterschiedlichsten Typen verlangen, macht es Sinn, über neue Wege nachzudenken.

Legt man folgende Fakten zugrunde:

- Das Rekonfigurieren von FPGAs kostet bei nur etwas älteren Typen noch einige Sekunden, die neueren schaffen diese Leistung im **ms-Bereich**. Man kann deshalb direkt von einer dynamischen Rekonfiguration sprechen, vor allem, da man erwartet, dass diese Zeiten sich weit unter 1 ms drücken lassen (Ziel ist der untere ns-Bereich).
- Die Anzahl der Gatteräquivalente ist inzwischen so angestiegen (> 225.000 Gates: XH3-Familie von Xilinx), dass sich kleinere Prozessoren oder zumindest größere Teile eines Prozessors in einen Baustein integrieren lassen.
- Bei neuen FPGAs wird eine partielle Rekonfigurationsmöglichkeit angestrebt, was bedeutet, dass Teile einer Schaltung on-line neu konfiguriert werden, während andere Teile der Schaltung voll aktiv sind.

Stehen vom Prozess her also unterschiedliche Aufgaben an, so kann man je nach Bedarf die entsprechende Hardware laden (was nach "oben" wie ein entsprechender Software-Funktionsaufruf aussehen muss), um dann die entsprechend hohe Arbeitsgeschwindigkeit für das System zu erreichen. Für die Entwicklung derartiger Systeme muss das heißen: Beim Entwurf des Systems ist streng auf das Top-down Design zu achten. Zunächst entwerfe ich das System funktional, dann breche ich in die einzelnen Softwaremodule herunter und überlege, wie dazu entsprechende Hardwaremodule aussehen könnte. Der Optimierungsfähigkeit des Entwurfssystems wird man sehr viel abverlangen, denn es werden vielfältige Möglichkeiten existieren:

- Eine Software-Unit ist vielleicht vollständig effizient in eine Hardware-Unit umzusetzen.
- Eine Software-Unit kann vielleicht effizient in verschiedene Software- und Hardware-Units umgesetzt

werden.

# Zukünftige Aspekte



Völlig neue Wege denkbar:

- im Kfz fällt ein mechanisches System aus:  
*eine andere Hardware wird online neu geladen*
- Datenübertragungswege verändern ihre Güte:  
*die HW der Codierung wird online geändert*
- Verschlüsselung von Daten:  
*für jeden Schlüssel wird die Hardware entsprechend entwickelt*
- ein Prozessor übernimmt eine andere Aufgabe:  
*und stellt sich dafür aus einer Bibliothek die optimale HW zusammen*



Da aber zwischen diesen drei prinzipiellen Lösungen alle anderen Varianten möglich sind, die Randbedingungen aber vielfältiger Natur sein werden (Chip-Fläche, Anzahl der Bausteine, Kosten, Anzahl der Leitungen, EMV, Arbeitsgeschwindigkeit usw.) und die Wissenschaft und Industrie uns kaum eine allumfassende Lösung für diese Problemstellung liefern werden, muss die Problematik von jedem Entwicklungingenieur erfasst und verarbeitet werden. Denn eines ist gewiss, es lassen sich damit äußerst effiziente und preisgünstige Systeme designen, die die anderen, weniger durchdachten Systeme "überrunden" werden.

Den Gedanken kann man aber noch weiter spinnen. Man denke an massive Parallelrechner oder neuronale Netze. Hier sind zahlreiche Einzelprozessoren aktiv, die sich laufend auf neue Prozesse einstellen müssen. Die Problematik bestand bisher darin, die Schaltung zu finden, die einerseits hohe Arbeitsgeschwindigkeiten verspricht, andererseits aber für die verschiedensten Anwendungen einsetzbar ist - exakt das Problem, das eingangs geschildert wurde. Bei neuronalen Netzen ist man beispielsweise schon den Weg gegangen, die einzelnen Neuronen jeweils durch ein Fuzzy-System zu parametrisieren, das heißt, den gegebenen Umständen anzupassen. Derartige Lösungen werden jedoch bisher fast nur auf Softwarebasis erzielt oder nur im beschränkten Umfang verwendet.

Noch ein Gedanke: Der Weg der vollständigen Informationserfassung und -verteilung über die verschiedenen Netze bis hin zu den Feldbusssystemen, die Sensoren und Aktoren miteinander vernetzen, fordert mehr und mehr den Selbsttest der Systeme. Es reicht in einem Kfz nicht mehr, dass das Rücklicht über einen elektronischen Schalter gesteuert wird, sondern man möchte jederzeit während des Fahrens wissen, ob die Leuchteinheit funktionsfähig ist oder nicht. Das wird heute im allgemeinen so gelöst, dass der Strom des Treibers gemessen wird, der charakteristische Werte behalten muss. Zudem wird in die Treiber ein BITE (Built-in Test Equipment) integriert, das laufend die Funktionsfähigkeit des Treibers prüft und signalisiert. Wenn aber nun im ms-Bereich Schaltungen rekonfigurierbar sind, können sie on-line laufend in eine einfach testbare Schaltung umgewandelt und geprüft werden, ohne dass der äußere Prozess davon etwas bemerken muss.

Mit der neuen FPGA-Technologie eröffnen sich völlig neue Wege. Es ist daran zu denken, dass on-line, also während des Prozesslaufs, neue Schaltungen oder partielle Teile davon vom System entworfen, compiliert und dann geladen werden. Man stelle sich wiederum ein Kfz vor. Größere Kraftfahrzeuge verfügen für das Navigationssystem schon über leistungsfähige PCs. Fällt nun im Kraftfahrzeug ein System oder ein Teil einer Systemeinheit aus, ist es denkbar, dass der PC die Aufgabe übernimmt, die neue Situation zu analysieren und zu prüfen, ob es nicht möglich ist, bestimmte elektronische Schaltungen so abzuändern, dass wiederum ein halbwegs vernünftiger Prozessablauf erreichbar ist.

"Wir werden in ein paar Jahren für nur 10 Dollar einen Systemchip bauen können, der einen 386er-Mikroprozessor samt einem 4Mbyte-DRAM sowie den erforderlichen Eingangs- und Ausgangsbaustein integriert. Das entspricht einem PC des Jahres 1990. Aber wir wissen noch nicht, was für ein Gerät wir damit auf den Markt bringen sollen."



*Der Satz wurde im Dezember 1996 auf der IEDM (International Electron Devices Meeting) gesagt.*

.....

Das ermöglicht völlig neue Testkonzepte für alle Bereiche der Elektronik, und in diesen Bereich wird zur Zeit sehr viel Geld investiert, da sich bei effizienter Testung während und vor allem nach der Produktion enorme Dienstleistungskosten einsparen lassen, ein wesentlicher Teil der Wartungskosten.

Oder man denke an die Verschlüsselung. Beim DES-Verfahren (Data Encryption Standard) müssen beispielsweise Algorithmen abgearbeitet werden, wobei bestimmte Schaltungseinheiten nur über das Wissen von Teilschlüssel verfügen müssen. Wird beim Schaltungsentwurf der Teilschlüssel konkret berücksichtigt (muss man also nicht alle möglichen Schlüsselkombinationen berücksichtigen, sondern nur einen (!)), vereinfacht sich die Schaltung natürlich wesentlich. Da bietet es sich an, einen Baustein zu verwenden, bei dem dann eine neue Schaltung geladen wird, wenn dem Algorithmus ein neuer Teilschlüssel zugrunde liegt.

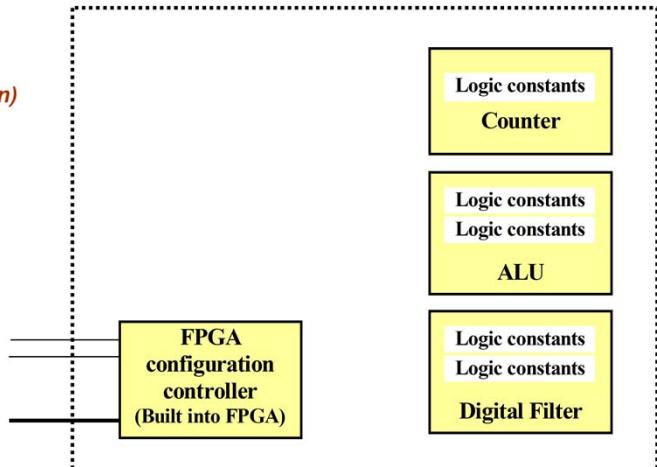
Schon 1995 wurde schon der erste Prozessor mit einem variablen Befehlssatz vorgestellt: DISC (Dynamic Instruction Set Computer). Er ist in der Lage, sich bei bestimmten Anforderungen aus einer Bibliothek einen neuen Konfigurationssatz bestimmter Schaltungsteile zu holen, um damit effizienter die spezielle Aufgabe lösen zu können.

**RC:**

*die gerade laufende Anwendung (Applikation) passt die Computerarchitektur dynamisch den momentan Anforderungen an.*

*erste Ideen: 1992:  
Algotorix*

*Ziel:  
Rekonfigurationszeit  
heute: 200 ms  
Ziel: min. 5 ms*



In /Emb97/ wird eine Software *QuickChange* vorgestellt, die es erlaubt, In-system Reconfigurations of FPGAs (IRF) vorzunehmen. Als wesentliche Anwendungsgebiete werden konsequenterweise DSPs (Digitale Signalprozessoren) sowie die Kommunikationstechnik genannt. Ziel ist es zunächst, Platz und Pins des Chip freizumachen, um andere anfallende Aufgaben lösen zu können, also nicht zuviel gleichzeitig in einen Baustein packen zu müssen. Kontrolliert wird dies über einen FPGA Configuration Controller, der selbst im Chip integriert ist. Bild oben zeigt ein Beispiel, bei dem eine dynamische Rekonfiguration möglich ist. Die normalerweise als Parameter integrierten Größen werden hier als Konstante gehandelt. Eine Parameterveränderung erwirkt somit eine Rekonfiguration der Teilschaltung.

/Emb97/      Embedded Systems Programming Europe: Reconfigurable FPGA Design; September 1997  
(Autor unbekannt)

## ISOC: Integrated System on one Chip

- Reduces interface overhead
- Permits greater device utilization
- Frees internal routing resources
- 10 Mbyte/s transfer rate



Bringt man diese Überlegung zusammen mit den Vorstellungen der ladbaren Hardware auf einen Punkt, sieht dies also in Zukunft in einem Chip (**ISOC: Integrated System on one Chip**), wird deutlich, welche Performance man von zukünftigen Chips bei welchen Preisen erwarten darf. Chips dieser Größenordnung können dann aber in viele preisgünstige, handelsübliche Systeme integriert werden, was vielleicht spüren lässt, welcher Aufbruch in der Informationstechnologie zu erwarten ist.

# Komplexe Schaltwerke und ASIC-Entwicklung

Dietmar Dietrich  
Institut für Computertechnik  
[dietrich@ict.tuwien.ac.at](mailto:dietrich@ict.tuwien.ac.at)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1  
p. 1  
18.01.2013 14:03:23

Die Komplexität digitaler Systeme hat dramatisch zugenommen und wird das noch weiter tun. Sehr deutlich wird dies am Generationswechsel der programmierbaren Bausteine, angefangen vom PAL bis zu den heutigen "Mega-Bausteinen", wobei, wie erläutert wurde, schon darüber nachgedacht wird, wie man unter bestimmten Umständen verschiedene Designs online laden kann, um die Hardware optimal den verschiedenen Applikationen anzupassen.

Doch wie so oft spielt nicht nur *ein* Faktor die entscheidende Rolle, damit sich Entscheidendes bewegt. Im Falle der Hardwareentwicklung können die folgenden Aspekte aufgezählt werden, die dazu führten, dass enorme Anstrengungen für die Entwicklung einer effizienten Entwurfssprache unternommen wurden:

- Komplexität
- Time to Market
- höhere Produktivität
- Notwendigkeit der Kostensenkung aufgrund der enormen Konkurrenz

Dass dadurch von der traditionellen 'Paper-and-Pencil'- und 'Capture-and-Simulate'-Methode Abschied genommen werden muss, wird verständlich:

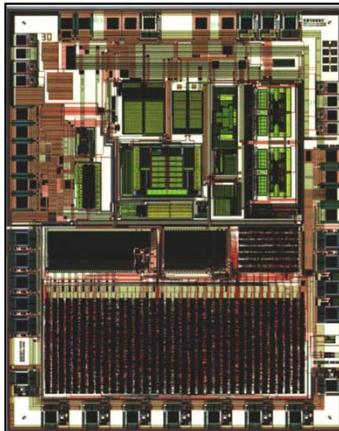
- Es muss auf einem höheren Abstraktions-Level aufgesetzt werden.

*Nach der Topdown-Methode muss eine weitgehende Abstrahierung vorgenommen werden, um das Problem möglichst zu vereinfachen.*

- Wissen, das nicht notwendig ist, muss ausgegrenzt werden.

*Alles, was nicht zur Lösung beiträgt, ist zu eliminieren.*

# 6 VHDL



- Ziel, Motivation
- Historie
- Entwurfsebenen
- Prinzip - Aufbau

## VHSIC Hardware Description Language

- VHSIC: Very high Speed Integrated Circuit

Teil 1: allgemeiner Überblick

Teil 2: im Detail



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p.2  
18.01.2013 14:03:23

- Abläufe müssen so automatisiert werden, dass Fehler weitgehend vermieden werden.  
*Fehler können nicht vermieden werden. Man kann jedoch Abläufe so automatisieren und Regeln unterwerfen, dass möglichst viele Fehler offensichtlich werden.*
- Abläufe müssen so automatisiert werden, dass Wissen nicht mehrfach (redundant) eingebracht werden muss.  
*Wissen mehrfach eingebracht ist nicht nur eine überflüssige Fehlerquelle, sondern verursacht zusätzlichen Arbeitsaufwand. Dies zielt darauf ab, dass die verschiedenen Tool-Units, die ein Frame Work beinhaltet, miteinander kommunizieren müssen, aber auch darauf, dass solchen Tools umfangreiche Datenbanken zugrunde gelegt werden müssen. Zum dritten bedeutet es auch, dass Mehrfachentwicklungen möglichst vermieden werden sollen.*

Hardwareentwicklung heutiger Zeit verlangt 'describe-and-synthesize'-Methoden, die weitgehend automatisiert werden können. Damit stehen wir am Anfang einer Entwicklung, die erst begonnen hat und sich noch sehr lange fortsetzen wird. VHDL (VHDL Hardware Description Language) ist eine standardisierte Hardware-Beschreibungssprache, die sich als hocheffizient erwiesen hat und in der Zwischenzeit auch in anderen Bereichen Anwendung findet. Dass ihr, wie allen anderen lebenden Sprachen, immer neue Elemente hinzugefügt werden, ist verständlich.

Ziel dieses Abschnitts kann es nun nicht sein, VHDL umfassend darzulegen. Der Abschnitt soll ausschließlich einen kurzen Überblick geben, um welche Art von Werkzeug es sich handelt und wie man damit umgeht. Um mit VHDL designen zu können, bedarf es eines tiefergehenden Studiums und vor allem Programmierpraktikums. Denn wie jede Programmiersprache ist sie am effizientesten über die Praxis erlernbar.

- Ashenden, P. J.: The Designer's Guide to VHDL; Morgan Kaufmann Publishers, Inc., 1996**  
sehr ausführlich, viele Beispiele, 688 Seiten  
Will sich jemand intensiv mit VHDL beschäftigen: empfehlenswert!
- Bhasker, J.: A VHDL Primer, Prentice Hall; Englewood Cliffs, NJ 07632; 1992**  
gute Einführung, die späteren Kapitel weniger von Interesse
- Bhatnagar, H.: Advanced ASIC Chip Synthesis; Kluwer Academic Publishers, 1999**
- Jerraya, A. A.; Ding, H.; Kission, P.; Rahmouni, M.: Behavioral Synthesis and Component Reuse with VHDL; Kluwer Academic Publishers; 1997**  
leicht geschrieben, viele prinzipielle Überlegungen, die als Einführung nicht immer wichtig sind
- Kurup, P.; Abbasi, Th.: Logic Synthesis using SYNOPSYS; Kluwer Academic Publishers; 1995**  
viele Beispiele, behandelt viele prinzipiellen Dinge
- Lipsett, R.; Schaefer, C.; Ussery, C.: VHDL: Hardware Description and Design; Kluwer Academic Publishers; 1989**  
eines der ersten Bücher, heute interessant als Ergänzung zu lesen
- Perry, D. L.: VHDL; McGraw-Hill Inc.; 1991**  
einfache Einführung, viele Beispiele, empfehlenswert
- Siemers, Chr.: Hardwaremodellierung, Einführung in Simulation und Synthese von Hardware; Hanser, 2001**  
gute Einführung, einfach zu lesen, empfehlenswert
- Smith, M J. S.: Application-Specific Integrated Circuits, Addison-Wesley, 1997**  
sehr ausführlich bzgl. ASICs, steigt bei VHDL aber sofort tief ein

## Nachteile traditioneller Entwurfsmethoden

- Detailkenntnisse werden verlangt ↗ niedriger Abstraktionslevel
- wenig Formalismus ↗ viele Fehlerquellen
- Dokumentation schwierig ↗ hohe Wartungskosten
- Redesign aufwendig ↗ hohe Redesignkosten
- Schnittstellendefinition zwischen Teams schwierig ↗ sequentielles Design
- verleiten zur Bottom-up-Methode (Piecemeal Design) ↗ nicht effizientes Design
- Bibliotheken schwierig ↗ laufende Neuentwicklungen



**Time to Market - hohe Entstehungskosten -  
hohe Redesign-Kosten**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1  
p. 4  
18.01.2013 14:03:23

Gehen wir von den oben getroffenen globalen Aussagen zu den konkreten Aspekten über, welche die traditionellen Designverfahren unwirtschaftlich machen.

- Aufgrund des niedrigen Abstraktionslevel werden viele Detailkenntnisse verlangt.
- Die klassischen Verfahren basieren auf keinem strengen Formalismus, bilden also Fehlerquellen und sind auch nicht so einfach zu automatisieren.
- Die Dokumentation hängt stark von der Konsequenz und Fähigkeit des Designers ab und wird mehr und mehr die Ursache für Wartungskosten, die in manchen Projekten schon den größten Teil der Gesamtkosten ausmachen.
- Schlechte Dokumentation und fehlender Formalismus sind dann aber auch die Ursache für hohe Redesignkosten.
- Komplexe Bausteine können nicht mehr von einer Gruppe und schon gar nicht von einer Person realisiert werden. Somit müssen Schnittstellen geschaffen werden, die besonders dann schwierig zu realisieren sind, wenn sie unterschiedliche Projektaufgaben beinhalten wie reines Hardware Design, Test-Tool-Erstellung, Software-Anbindung usw. Schnittstellen eines streng formalisierten Hardwaredesigns lassen sich relativ einfach und klar aus dem entworfenen Formalismus ableiten.
- Die traditionellen Entwurfsprinzipien verleiten zu Bottom-up- oder Middle-out-Methoden, wo es gar nicht notwendig ist. Das Ergebnis ist dann ein "Piecemeal Design", das nicht besser ist als der Spaghetti-Code in der Softwaretechnologie.
- Die Traditionellen Methoden führen nicht zu Bibliotheken. Ein Entwurf zwingt den Designer zum wiederholten Entwurf. Time-to-Market, Zuverlässigkeit der Schaltung usw. zwingen jedoch heute dazu, möglichst auf Bibliotheken aufzubauen

*Zusatzbemerkung: Die Komplexität digitaler Schaltungen nimmt drastisch zu. Wäre man nicht in der Lage, die Fehlerhäufigkeit mindestens in gleichem Maße zu reduzieren, würde die Qualität dieser Schaltungen drastisch sinken. Ziel von modernen Hardwaredesign-Methoden muss es also sein, die Entwicklungsqualität "überproportional" zur Komplexität zu steigern.*

# Designmethoden

- **klassisches Entwurfsverfahren**
  - Bottom-up
  - Top-Down
- **Designsprachen**
  - PAL-Assembler
  - zahlreiche firmenspezifische Varianten
  - HDL
  - VHDL



Diesen Nachteilen wirkt eine HDL (Hardware Description Language), die auf strengen Formalismen basiert, entgegen, was sich letztendlich auch in kleinen Designs wirtschaftlich positiv auswirkt.

Wie müssen nun aus den aufgezählten, nachteiligen Aspekten die Charakteristika einer HDL aussehen? Wie muss eine Gliederung oder Modularisierung von Design-Werkzeugen aussehen, die den hohen Anforderungen standhält, insbesondere unter der Annahme, dass man heute schon weiß, dass das Ende dieser Entwicklung bei weitem noch nicht erreicht ist?

Mit Hardwarebeschreibungssprachen (HDLs), wie z. B. VHDL können Verhalten und Struktur beschrieben werden. HDLs sind nicht geeignet um Geometrie zu beschreiben, dazu werden eigene Layout-Tools verwendet. Der wichtigste Einsatzbereich von HDLs liegt auf der algorithmischen Ebene, der RT-Ebene und der Logikebene. Es ist aber auch möglich, sie auf der Systemebene zu verwenden, und es gibt Ansätze für den Einsatz auf Schaltkreisebene.

VHDL und Verilog sind die zwei wichtigsten Hardwarebeschreibungssprachen. Beide Sprachen haben weltweit in etwa die gleiche Verbreitung. Während VHDL in Europa dominierend ist, wird in der amerikanischen Industrie überwiegend Verilog eingesetzt. Hier soll daher im folgenden nur von VHDL die Rede sein.

# Beschreibungsmethoden

- **logische Schaltbilder**
  - **Boolesche Gleichungen**
  - **Tabellen**
    - Zustandstabelle
    - Übergangstabelle
  - **Zustandsmaschinen**
  - **Modularisierung nach dem Prinzip komplexer Schaltwerke**
  - **Beschreibungssprachen**
- in Zukunft zusätzlich:**
- **funktionale Beschreibungsmethoden**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 6

18.01.2013 14:03:23

# Beschreibungsmethode: Componentware

- **Keyword:** Visuelle Sprache, visuelle Programmierung  
*GUI: Graphic User Interface*
  - **Idee:** Lego-Bausteinkastenprinzip
  - **Basis:** objektorientierte Sprachen
  - **Bausteine:** Icons mit Pfeilen (Ein-, Ausgänge) =  
Vaustein
  - **Vorgehensweise:** Anordnung von  
Vausteinen in einem  
definierten Bereich
  - **Festlegung der Kommunikationsbeziehungen**  
(= Visuagramm)
- siehe SDL in Vorlesung: Feldbussysteme

Funktionsebene



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 7  
18.01.2013 14:03:23

- Increasing Design Size and Complexity
- hoher Abstraktionslevel
  - ☞ Black-Box-Prinzip
  - ☞ Konzentration auf die Funktion
- Shrinking Time-to-Market Window
- Shorter Design Cycles
- Arbeiten in kleinen Teams
  - ☞ Concurrent Engineering
- höhere Qualität
- Voraussetzung bei Systemen hoher Komplexität

## Fehlerhäufigkeit (fiktive Zahlen)

Annahme: Wahrscheinlichkeit, dass ein Atomkraftwerk hochgeht:

1 pro 100.000 Jahre

- in Deutschland existieren 50:
  - ☞ alle **2000** Jahre eines!
- in Frankreich existieren 250:
  - ☞ alle **400** Jahre eines!
- in Europa existieren 1000:
  - ☞ alle **100** Jahre eines!
- weltweit existieren 5000:
  - ☞ alle **20** Jahre eines!



Institut für Computertechnik

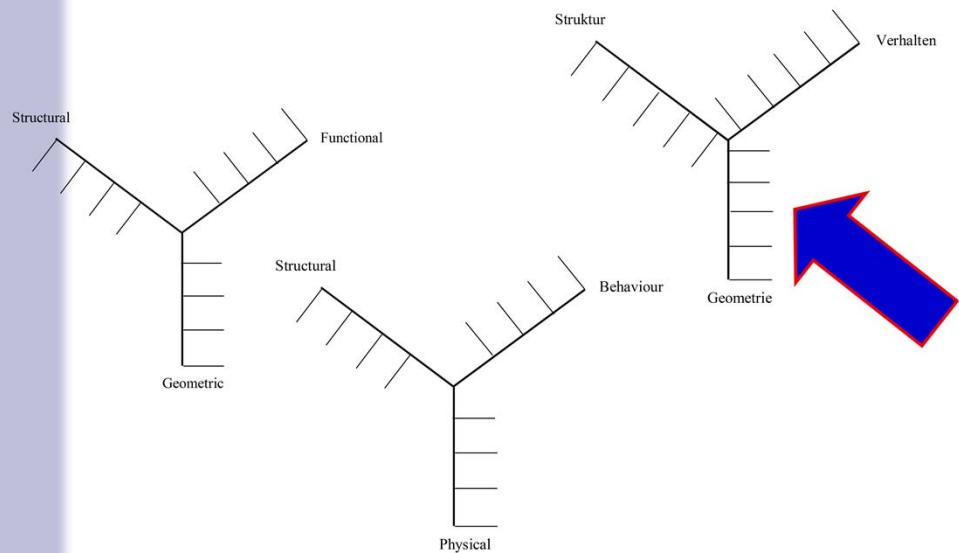
Digitale Integrierte Schaltungen  
DIS

K6-1

p. 9  
18.01.2013 14:03:23

Und wann war Tschernobyl? 1986! Also, 2006 wird's vielleicht wieder!

## Betrachtungsweisen: Y-Charts



Institut für Computertechnik

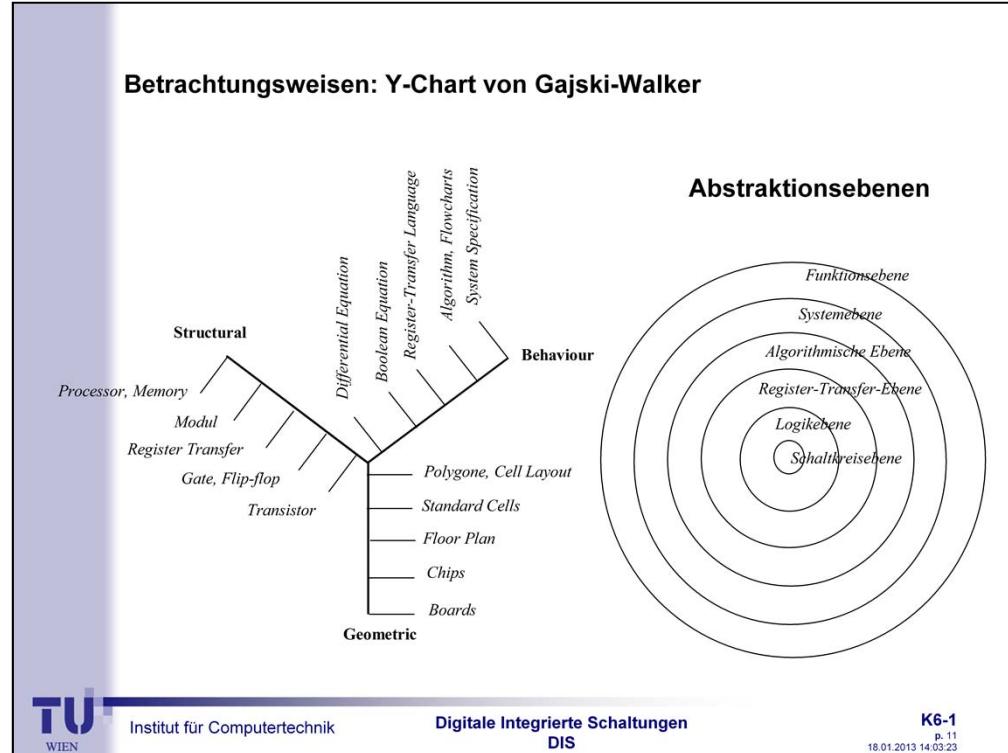
Digitale Integrierte Schaltungen  
DIS

K6-1

p. 10  
18.01.2013 14:03:23

Eine Abstraktion verlangt die Reduktion auf das Wesentliche und somit eine eingeschränkte Betrachtungsweise. In der Literatur findet man in diesem Zusammenhang verschiedene Ansätze. Das sogenannte Y-Diagramm, dargestellt in Bild oben, ist eine der möglichen Darstellungsweisen.

## Betrachtungsweisen: Y-Chart von Gajski-Walker



In der Abbildung sind die Abstraktionsebenen als konzentrische Kreise dargestellt. Folgende Ebenen sind bei elektronischen Systemen üblich:

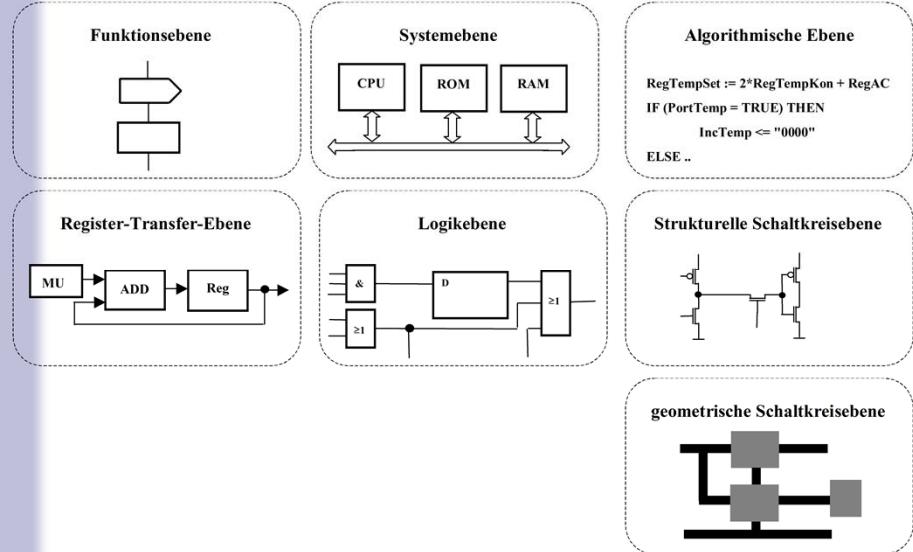
- Systemebene
- Algorithmische Ebene
- Register-Transfer-Ebene (RTL: Register Transfer Level)
- Logikebene
- Schaltkreisebene

Auf jeder dieser Abstraktionsebenen kann das System beschrieben werden im:

- Verhaltensbereich
- Strukturbereich
- Geometriebereich

Das vorgestellte Y-Diagramm stellt eine Vereinfachung der realen Verhältnisse dar. Es ist nicht immer möglich, eine Beschreibung eindeutig zu einer bestimmten Ebene oder zu einem bestimmten Bereich (Verhalten, Struktur oder Geometrie) zuzuordnen. Zudem kann die Beschreibung eines komplexen, elektronischen Systems auf verschiedenen Ebenen und in verschiedenen Bereichen stattfindet. So können zum Beispiel Teile eines Systems auf der Algorithmischen Ebene und der Rest auf der RT-Ebene beschrieben werden.

# Abstraktionsebenen



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1  
p. 12  
18.01.2013 14:03:23

Diese Bereiche sind in der Abbildung als radiale Linien dargestellt. In Bild oben sind einige Beispiele für die unterschiedlichen Sichtweisen und Abstraktionsebenen veranschaulicht.

## Applikations-Funktionsebene

Funktionsebene



- **definiert: Funktionen, unabhängig von HW oder SW**
- **von Interesse: vor allem Abläufe, Zustände, Kommunikationsdaten,**  
..
- **in VHDL noch nicht definiert**
- **Beispiele für Sprachen in dieser Ebene: SDL (Specification Description Language)**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

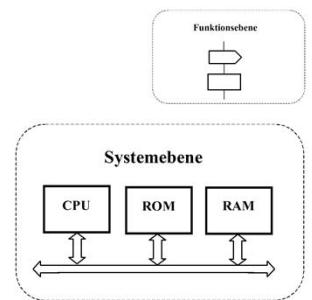
K6-1

p. 13

18.01.2013 14:03:23

Die Applikationsebene wird im Allgemeinen nicht zu den Design-Ebenen gezählt und somit ist sie auch nicht in VHDL spezifiziert.

# Systemebene



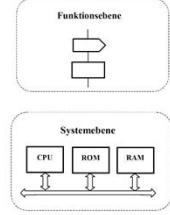
- ❖ **Beschreibung der Systemeinheiten (System Units)**  
*CPU, RAM, ROM, I/O, ..*
- ❖ **Bussysteme, Steuerleitungen**
- ❖ **funktionale Beschreibung der Units**  
*Befehlssatz, Protokolle, Zeitverhalten, ..*
- ❖ **evtl. Bausteinkomponenten-Gliederung**  
*Baugruppen, ICs, diskrete Bausteine, ..*



Die Systemebene weist den höchsten Abstraktionsgrad auf. Hier werden die grundlegenden Eigenschaften eines Systems festgelegt. Im Verhaltensbereich wird zumeist natürliche Sprache verwendet, um eine Schaltung zu spezifizieren. Im Strukturbereich wird das elektronische System durch die Verbindung von typischen Komponenten, wie Prozessoren, Speicher, Controller, Datenbusse u. a. beschrieben.

## Algorithmische Ebene

automatische  
Synthese  
bedingt  
möglich!



- **Beschreibung von nebenläufigen Prozessen**  
*Prozeduren, Funktionen, ..*
- **Blöcke kommunizieren über Signale miteinander**
- **Verhaltensbeschreibung**  
*Beschreibung des Verhaltens über Variablen und Operatoren*
- **kein Bezug zu HW-Positionierungen von HW-Komponenten**
- **keine zeitlichen Definitionen**  
*Takt, Funktions- und System-Delays, ..*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

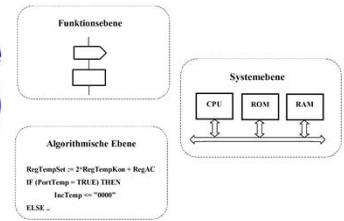
K6-1

p. 15

18.01.2013 14:03:23

Auf der algoritmischen Ebene erfolgt die Beschreibung einer Schaltung noch immer, ohne auf deren spezielle Implementierung Bezug zu nehmen. Zudem werden auf dieser Ebene ebenfalls keine zeitlichen Details spezifiziert. Zum Beispiel ist hier nicht von Interesse, ob eine bestimmte Operation in einem Takt ausgeführt werden kann, oder ob Pipelining notwendig ist, um eine geforderte Taktfrequenz zu erreichen. Trotzdem gibt es bereits Compiler, die eine automatische Synthese (die Umsetzung von einer logisch höheren auf eine niedrigere Abstraktionsebene) von der algorithmischen Ebene aus ermöglichen. Allerdings ist deren Anwendungsbereich noch eingeschränkt. In Zukunft wird diese Art von Compilern immer mehr an Bedeutung gewinnen.

# Register-Transfer-Ebene RTL (Register Transfer Level)



## Prinzipiell:

- Beschreibung über Operatoren
  - Addition, Multiplexen, ..*
- Beschreibung des Datentransfers zu differenzieren sind:
  - strukturelle Beschreibung
    - Addierer, Multiplexer, Register, Codierer, ..*
  - Verhaltensbeschreibung
    - Automaten*
  - Integration von Takt- und Rücksetzsignalen
  - Zeitflankendefinition

Register-Transfer-Ebene

MU → ADD → Reg

Erste Ebene der möglichen automatischen Synthese!



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

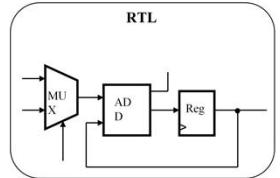
p. 16

18.01.2013 14:03:23

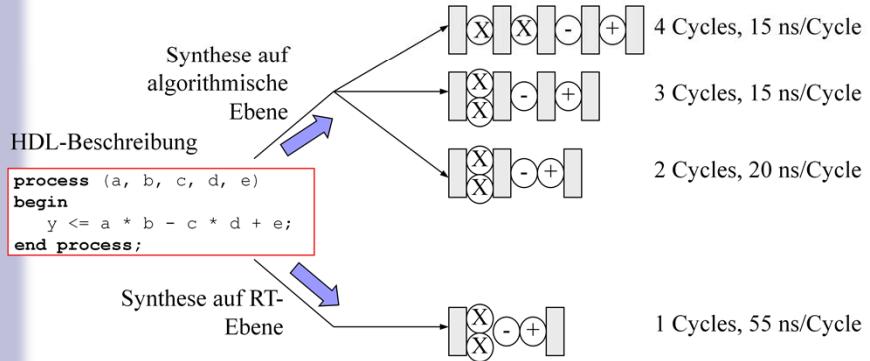
Auf der RT-Ebene (RTL: Register Transfer Level) wird die Funktion einer Schaltung durch Operationen und den Transfer von Daten zwischen Registern beschrieben. In der Literatur wird ein Design, das auf RT-Ebene formuliert ist, oft als "synthesierbar" bezeichnet, weil die automatische Synthese von dieser Ebene zur Zeit die vorherrschende Art der Synthese ist.

Algorithmische Ebene  
Verhalten

```
process (a, b, c, d, e)
begin
  y <= a * b - c * d + e;
end process;
```



## Algorithmische $\leftrightarrow$ RT-Ebene (RTL)



In Bild oben ist der wesentliche Unterschied zwischen der Synthese auf RT-Ebene und auf algorithmischer Ebene dargestellt. Die grau hinterlegten Rechtecke symbolisieren dabei Register. In der RT-Ebene ist durch die Beschreibung festgelegt, welche Operationen in einem Taktzyklus erfolgen. Auf der Algorithmischen Ebene ist diese Information hingegen nicht vorhanden, und der Compiler hat daher die Möglichkeit denselben Algorithmus auf verschiedene Arten zu implementieren. Das ist ein großer Vorteil, da dadurch verschiedene Lösungsvarianten frühzeitig im Designprozess evaluiert werden können. Es hängt von den an das System gestellten Anforderungen ab, welche Lösung die optimale ist.

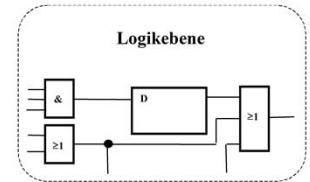
*Zusatzbemerkung: Der Ausdruck im Kästchen links oben sagt also noch nichts über die verschiedenen Entwurfsmöglichkeiten aus, wie welche Register angeordnet werden. Überträgt man die algorithmische Darstellung auf das rot gekennzeichnete Kästchen darunter, bieten sich verschiedene Realisierungsmöglichkeiten an:*

*Algorithmischen Ebene (Behaviour): Das ganze kann sich im Pipeline-Verfahren abspielen. Jeder Zyklus braucht dabei die angegebenen Zeiten und man kommt in der Gesamtzeit evtl. höher wie beim Entwurf direkt über die RTL-Ebene. Man muss jedoch beachten, dass man hier Daten parallel verarbeitet.*

*Die Zeitbetrachtungen sind hier noch offen. Deshalb sind mehrere Möglichkeiten denkbar: Lösungen in 4, in 3, in 2, in .. Takte.*

*RTL: Die Hardware legt hier klar fest, dass hier 2mal multipliziert, einmal subtrahiert und einmal addiert wird. Die Schaltung wird entsprechend gelöst. Das Ganze muss sich in einem Takt abspielen. In der RTL-Ebene macht sich die Zeit damit gravierend bemerkbar.*

# Logikebene



- ❖ **Beschreibung über logische Elemente und deren Eigenschaften**  
*Verzögerungszeiten, Tristate-Zustand, ..  
also: AND, OR, .., FFs, .., MUX, ..*
- ❖ **hier Bereitstellung von Bibliotheken**
- ❖ **zur Erstellung der Gatternetzliste:**  
*Schematic Entry (graphische Eingabewerkzeuge)*
- ❖ **für Behaviour: Boolesche Gleichungen**
- ❖ **Übergänge zwischen Behaviour - Structural - Geometric:**  
*i. a. Syntheseprogramme*



Das Ergebnis der RTL-Synthese ist eine optimierte Netzliste auf Logikebene. Hier werden Grundelemente, wie NAND-, NOR-, XOR-Gatter und Flip-Flops zusammengeschalten. Diese Bauelemente werden einer Bibliothek entnommen, die auf eine bestimmte Technologie ausgelegt sein muss. Die Logikebene ist somit technologieabhängig. In der Verhaltenssicht werden vor allem Boolesche Gleichungen und Funktionstabellen verwendet.

## Beispiel

```
while (X>Y) loop
```

```
  if COUNT=1 then
```

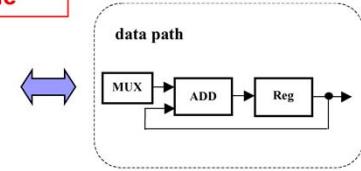
```
    X = X +2;
```

```
    Y = 1;
```

```
  else
```

```
  ..
```

### RTL-Ebene



### Logikebene

```
process (CLK, Q2, ..)
```

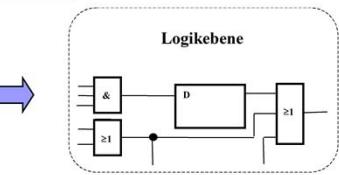
```
begin
```

```
  if rising_edge(CLK) then
```

```
    Q1 <= IN1
```

```
    Q2 <= T2 or T1
```

```
  ..
```



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

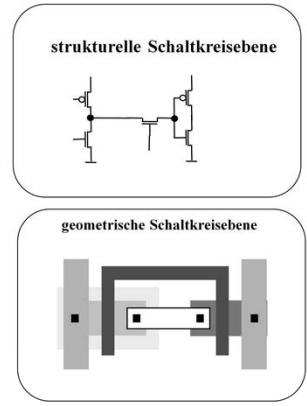
K6-1

p.19

18.01.2013 14:03:23

RTL: Register-Transfer-Ebene

# Strukturelle + geometrische Schaltkreisebene



- **strukturelle SchEbene:** Netzliste mit Transistoren, Kapazitäten, ..
- **geometrische SchEbene:** Polygondarstellung (= unterschiedliche Dotierungen, ..)
- **Verhaltens-SchEbene:**  
Differentialgleichungen
- **zeitkontinuierliche Signaldarstellung**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

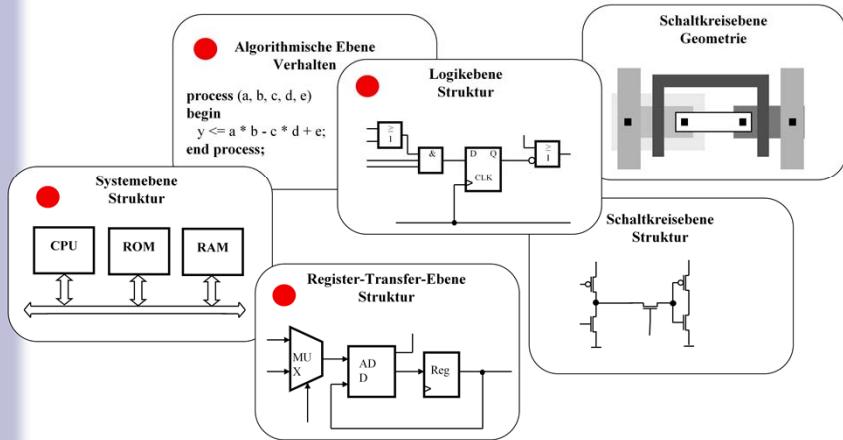
p. 20

18.01.2013 14:03:23

Die Schaltkreisebene weist den niedrigsten Abstraktionsgrad auf. In der strukturalen Sicht wird das elektronische System durch die Zusammenschaltung von Transistoren, Kapazitäten und Widerständen dargestellt. Im Verhaltensbereich werden auf dieser Ebene Differentialgleichungen verwendet. Im Unterschied zu den anderen Abstraktionsebenen können die Signale auf dieser Ebene beliebige Werte annehmen, sie sind daher zeit- und wertkontinuierlich. Die geometrische Sicht definiert die für die Fertigung des Chips notwendigen Masken.

Diese Thematik ist somit nicht Inhalt dieser Vorlesung. Der Teil wird nur der Vollständigkeit halber gebracht.

*Im Überblick:*  
**Abstraktionsebenen**



Die Systemebene sehen wir hier nur als Level, in der eine CPU usw. beschrieben werden. Einen höheren Abstraktions-Level soll hier nicht betrachtet werden.

# VHDL:

## VHSIC Hardware Description Language

**VHSIC:**

**Very high Speed Integrated Circuit**

Teil 1: allgemeiner Überblick

**Teil 2: im Detail**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 22

18.01.2013 14:03:23

Der Name ist NICHT Very high Description.Language, wie viele meinen.

# Historie (1)

1981: Finanzierung des VHSIC-Programms durch DoD (Workshops in Woods Hole, Massachusetts: Requirement Definitions)

Ziel: Dokumentationssprache digitaler Komponenten

- Senkung der Wartungskosten / Nachbesserungskosten  
(betrugen bisher die Hälfte der Gesamtkosten!)
- schnellere und einfachere Erfassung komplexer Systeme
- Austauschbarkeit von Modulen zwischen Projektgruppen  
(Entwicklung von HW-Bibliotheken)

1983: Intermetrics, Texas Instruments, IBM unter DoD Contract

Ziel: Entwicklung von digitalen Design Programs  
(in Anlehnung an ADA)

*ADA: nach Augusta Ada Byron benannt*

*DoD: Department of Defense*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 23

18.01.2013 14:03:23

Dieser Abschnitt soll eine Einführung in VHDL darstellen und nur einige der wichtigsten Konzepte von VHDL beschreiben. Der Schwerpunkt wird dabei auf Konzepte gelegt, die von "normalen" Programmiersprachen, wie PASCAL oder C, abweichen.

## Historie (2)

- 1985: Fertigstellung der ersten Version
- 1986: Übergabe an IEEE Standard
- 1986: Bildung der VASG: VHDL Analysis and Standardization Group (Beteiligung der CAE-Industrie)
- 1987: IEEE Standard 1076
- 1988: First synthesized Chip by IBM

Heute auch ANSI-Standard!

Beschreibt nur Syntax und Semantik,  
nicht Vorgehensweise oder spezifische Anwendungsprofile  
oder ..

CAE: Computer Aided Engineering

ANSI: American National Standards Institute



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 24

18.01.2013 14:03:23

Bemerkenswert: erst nach dem der Standard fertig war, wurde ein Jahr später damit der erste Chip designed! Nicht umgekehrt!

- VHDL: schon mehrere Revisionen in IEEE  
→ im wesentlichen Erweiterungen
- inzwischen weltweite Aktivitäten (Europa: ESPRIT)
- Konkurrenzmodelle:
  - Verilog HDL (USA)
  - UDL/1 (Japan)
- Ziel: Definition eines allgemeinen Standards für das Design elektronischer Systeme
  - also nicht nur für die Synthese digitaler Chips!

# VHDL-Prinzipien

- **Design-Methoden**
  - Top-down
  - (Bottom-up)
  - (Middle-out)
- **Modellierungen**
  - **strukturelle Modelle**
  - **Algorithmen**
  - **Datenfluss**
  - **synchrone Modelle**
  - **asynchrone Modelle**
- **Hardware-Aufbauten**
  - Chip Design
  - Modul-Design
  - PCB (Printed Circuit Board)
- digital
- analog



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 26

18.01.2013 14:03:23

# Features

- Design
- Synthese
- Dokumentation (*Na ja!*)
- Simulation
- Austauschbarkeit
- Testen (Strong Typing)
  - Compile Time Checking
  - User-definierte Simulationswerte
    - Signale
    - Variablen
    - Konstanten
    - Attribute
  - ...



Institut für Computertechnik

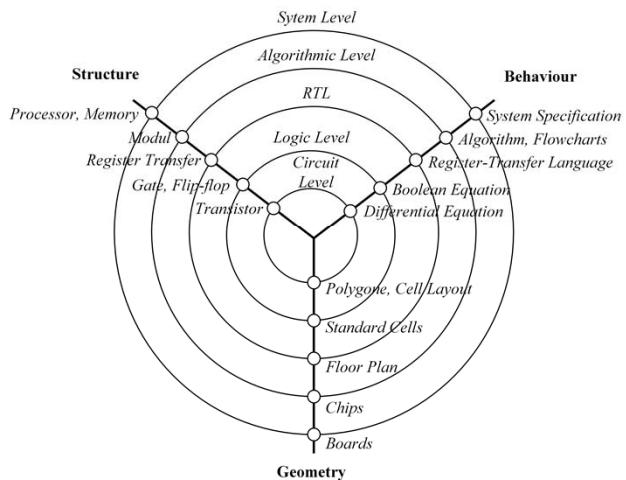
Digitale Integrierte Schaltungen  
DIS

K6-1

p. 27

18.01.2013 14:03:23

# Betrachtungsweisen: Y-Charts



Institut für Computertechnik

Digitale Integrierte Schaltungen

DIS

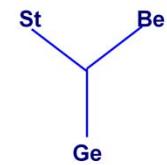
Betrachtet werden sollen hier im wesentlichen nur der Structure- und der Behaviour-Teil.

# VHDL- Abstraktions- ebenen

## Architecture Abstraction Level:

Behavioral: *Performance Specification*

Structural: *Logical Interconnection of CPUs, Memories, Buses, ..*



## Algorithmic Abstraction Level:

Behavioral: *Sequential Behaviours*

Structural: *Physical Interconnection of Chips and Modules*

## Functional Abstraction Level:

Behavioral: *Concurrent operations, RT, States*

Structural: *Physical Interconnection of Functions (ALU, Registers, ..)*

## Logical Abstraction Level:

Behavioral: *Boolean Equations*

Structural: *Physical Interconnection of Gates, Latches, ..*

## Circuit Abstraction Level:

Behavioral: *Differential Equations*

Structural: *Transistors, Capacitors, Resistors*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

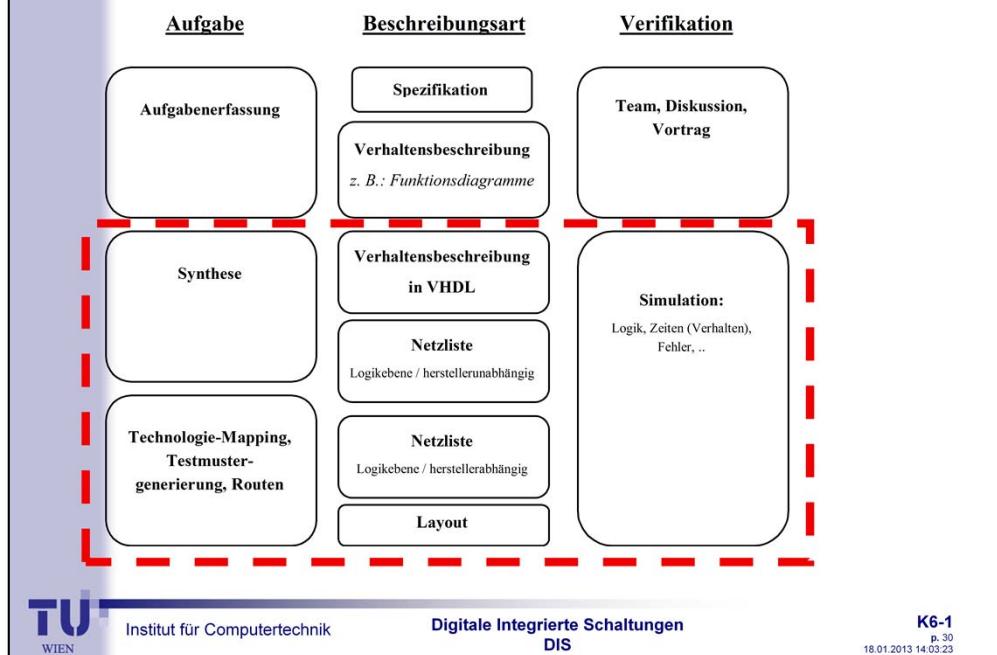
K6-1

p. 29

18.01.2013 14:03:23

Gegenüber der Folie Y-Chart: hier die konkreten Bezeichnungen der Ebenen für VHDL.

# Prinzipieller Design-Ablauf



# Tools

**Frame Work  
Design Management**

- Versionstypen
- Beziehungen
- Sicherheit

**Capture Design**

- Grafik
- Text

**Analyse**

- Syntax
- Semantik

**Simulator + Debugger**

- Logik
- Zeitverhältnisse
- Stimuli-Units

**Synthese**

- RTL-Beschreibung
- Gate-Level-Beschreibung
- Level-Umsetzung*
- Ergebnis: Netzliste*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

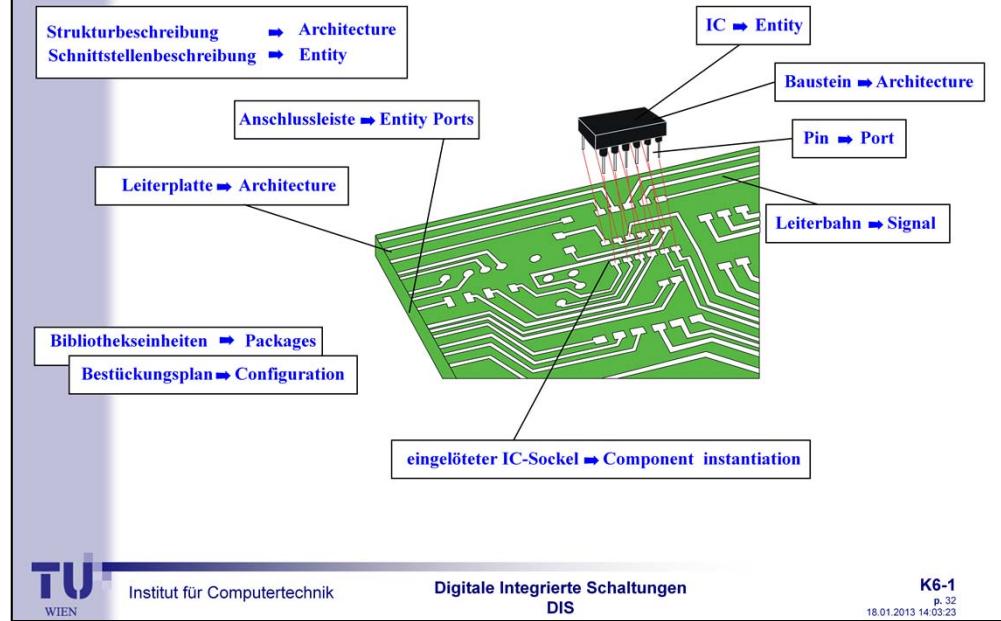
K6-1

p.31

18.01.2013 14:03:23

Capture: Datenerfassung

# Analogien in der Sprache (1)



Diese Analogien sind nicht immer richtig und gut!

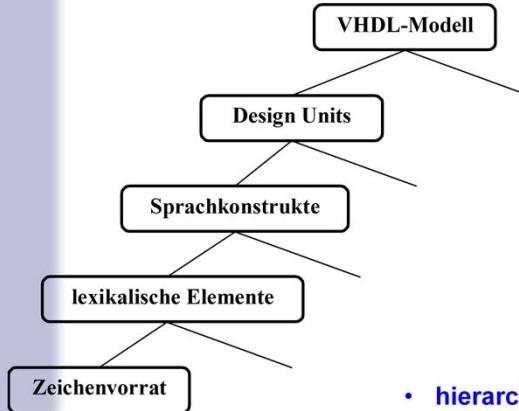
Behaviour + Structure = Beschreibung der Funktion!

Entity = IC (wenn die Pins gemeint sind); Entity ist die Schnittstellenbeschreibung, gleichzeitig aber der IC.

Architecture ist die Leiterplatte, gleichzeitig der Baustein: wird als Black Box betrachtet

Component instantiation: verdrahtet (Verbinden der Komponenten)

# Sprache (2)



- **hierarchisches Modell**
- **Kommentare werden nicht ausgewertet,**  
*beginnen mit ;,--“*



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p.33

18.01.2013 14:03:23

Was ist der Vorteil eines hierarchischen Modells?

## Zeichenvorrat

- **Character Declaration**
  - *<nahezu vollständige ASCII-Tabelle mit vielen Sonderzeichen*
  - *i. a. (I) case **insensitive** (*input01 = Input01*)*

## Lexikalische Elemente

- **reservierte Worte** (*ABS, ARCHITECTURE, BUS, CONSTANT,..*)
- **Bezeichner** (*Identifier: Namen für Funktionen, Objekttypen, ..*)
- **Größen** (*numerische Größen, Zeichengrößen, Zeichenketten, Bit-String-Größen*)
- **Trenn-, Begrenzungszeichen** (*Einzelzeichen, zusammengesetzte Zeichen*)
- **Kommentare**

**insensitive:** **unempfindlich:** kleine Schreibfehler wirken sich nicht gleich groß aus; man hat mehr Flexibilität; aber es besteht die Gefahr zum Schludern!

## Sprachkonstrukte (*lexikalische Elemente mit syntaktischer Bedeutung*)

- **Primitiven** (*Operanden + Operatoren*)
- **Befehle** (*Anweisungen*)
- **syntaktische Rahmen** (*Funktionen, Prozeduren; enden mit „end“*)

## Design Units

## VHDL Modell

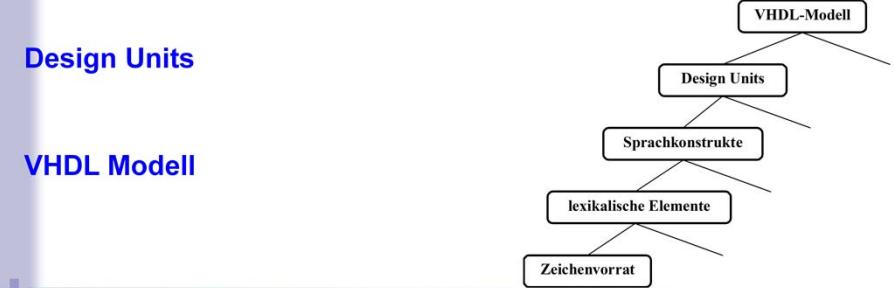


Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

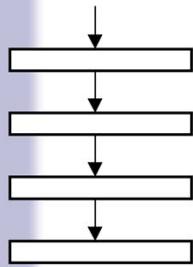
K6-1

p. 35  
18.01.2013 14:03:23

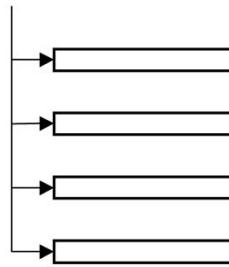


# VHDL Statements

Sequential



Concurrent



- charakteristisch für VHDL
- concurrent üblich: synchrone Schaltung!



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

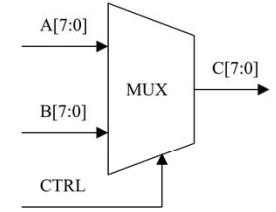
p. 36

18.01.2013 14:03:23

# Einfaches Beispiel: Multiplexer

```
entity MUX is
    port (A, B : in bit_vector(7 downto 0);
          CTRL : in bit;
          C     : out bit_vector(7 downto 0)
        );
end MUX;

architecture MUX_behavioural of MUX is
begin
    C <= A when CTRL='0' else
        B;
end MUX_behavioural;
```



algorithmische Ebene



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 37

18.01.2013 14:03:23

Als Einführung soll eine einfache kombinatorische Schaltung, ein Multiplexer, dienen. Bild oben zeigt das Blockdiagramm der Schaltung. Dieser Multiplexer besitzt zwei 8-bit-Dateneingänge, einen Kontrolleingang und einen 8-bit-Datenausgang.

Der im Bild gezeigte Multiplexer stellt ein Schaltnetz dar, das je nach Zieltechnologie aus UND- und ODER- Gattern oder aus anderen Grundelementen aufgebaut sein kann. In VHDL besteht die Möglichkeit, den Multiplexer auf einer höheren Abstraktionsebene (z. B. RTL) zu beschreiben und die Aufgabe, eine optimale Anordnung von logischen Gattern zu bestimmen, der logischen Synthese zu überlassen. Auf Register-Transfer-Ebene reicht für die Beschreibung der Schaltung eine einzige nebenläufige Anweisung (Concurrent Statement) aus. Darin ist auch die Stärke von VHDL zu sehen: Es wird eine große Anzahl von Highlevel-Konstrukten unterstützt, die es erlauben, komplexes Verhalten einfach zu beschreiben.

Der oben dargestellte Sourcecode zeigt eine von mehreren Möglichkeiten, den Multiplexer in VHDL zu beschreiben.

Man sieht, dass die VHDL-Beschreibung aus zwei Teilen besteht:

- Eine Entity Declaration, die die Ein- und Ausgänge definiert.
- Eine Architecture Declaration, die die Funktion der Schaltung beschreibt. Dazu wird eine einzige nebenläufige Anweisung (Concurrent Statement) verwendet.

Die Entity Deklaration beginnt mit dem Schlüsselwort **entity**, gefolgt von einem eindeutigen Namen (MUX). Außerdem enthält sie ein Port-Statement, in dem alle Ein- und Ausgänge der Entity definiert werden. Es werden hier 4 Ports definiert: A, B, C und CTRL. Jedem der 4 Ports wird eine Richtung (**in**, **out** oder  **inout**) und ein Typ (in diesem Fall entweder **bit\_vector(7 downto 0)** oder **bit**) zugewiesen. VHDL bietet hierbei eine große Anzahl von verschiedenen Datentypen, auf die im Folgenden noch näher eingegangen wird. Um die Erklärung einfach zu halten, werden zunächst nur die Typen **bit** und **bit\_vector** verwendet.

Der zweite Teil des VHDL Sourcecodes ist die Architecture Declaration. Für jede Entity Declaration muss es zumindest eine dazugehörige Architecture Declaration geben. Die erste Zeile der Architecture Declaration beginnt mit dem Schlüsselwort **architecture**, gefolgt vom Namen der Architecture (behavioural). Nach dem Schlüsselwort **of** wird der Name der dazugehörigen Entity angegeben, in diesen Fall MUX. Zwischen den Schlüsselwörtern **begin** und **end** befindet sich die Beschreibung der Funktionalität des Multiplexers.

Es gibt mehrere Möglichkeiten, um Schaltnetze in VHDL zu beschreiben. Die Methode, die im Beispiel verwendet wird, wird als bedingte Zuweisung (Conditional Statement) bezeichnet. Dem Ausgang C wird der Wert von A zugewiesen, falls CTRL auf logisch 0 gesetzt ist, anderenfalls der Wert von B. Eine einzige nebenläufige Anweisung dieser Art stellt die einfachste Form einer VHDL Architecture dar. Es stehen eine Menge verschiedener Typen von nebenläufigen Anweisungen zur Verfügung, die es gestatten, sehr komplexe Architectures zu formulieren.

*Zusatzbemerkung: Mehrere nebenläufige Anweisungen werden gleichzeitig ausgeführt. Die Reihenfolge von nebenläufigen Anweisungen im Source Code spielt daher keine Rolle.*

*Um ein System zu beschreiben, sind im Allgemeinen viele unterschiedliche Architectures notwendig, also bspw. MUX1, MUX2, ..*

## Deklaration eines Objektes durch

- Bezeichner (Identifier) und

(erstes Zeichen muss ein Buchstabe sein; "\_" ist nach der ersten Version verboten)

- Datentyp

## Beispiel:

```
type bit    is ('0', '1')
```



In VHDL stehen die folgenden Objektklassen zur Verfügung:

- Konstanten kann nur einmal ein Wert zugewiesen werden, der dann bis zum Programmende konstant bleibt.
- Variablen können während des Programmlaufs beliebig oft neue Werte zugewiesen werden.
- Signale werden verwendet, um zwischen Prozessen zu kommunizieren oder um die Untereinheiten eines Designs zu verbinden

Bei der Deklaration eines Objekts muss ihm ein Datentyp (z. B. binär, festgelegt durch eine Anweisung oder vorbestimmt) und ein Bezeichner (bspw. "bit") zugewiesen werden. Der Datentyp legt die Wertemenge (binär, boolean, ..) und die für das Objekt erlaubten Operationen fest. Über den Bezeichner kann auf das Objekt zugegriffen werden.

*Zusatzbemerkung: Die neuen VHDL-Standards erlauben die Möglichkeit, jedes gewünschte Zeichen einzubinden (bis auf den Bezeichner "/", durch das dies markiert wird).*

- **Vorschrift ist eine strenge Typisierung.**
- **Einem Typ ist eine Menge von Werten zugewiesen.**
- **Einem Typ ist eine vordefinierten Menge von Operatoren zugeordnet.**
- **Verschiedene Typen sind inkompatibel zueinander.**
- **Alle abgeleitete Typen desselben Basistyps sind zueinander kompatibel.**

In VHDL stehen die folgenden Objektklassen zur Verfügung:

- Konstanten kann nur einmal ein Wert zugewiesen werden, der dann bis zum Programmende konstant bleibt.
- Variablen können während des Programmlaufs beliebig oft neue Werte zugewiesen werden.
- Signale werden verwendet, um zwischen Prozessen zu kommunizieren oder um die Untereinheiten eines Designs zu verbinden

Bei der Deklaration eines Objekts muss ihm ein Datentyp und ein Bezeichner zugewiesen werden. Der Datentyp legt die Wertemenge und die für das Objekt erlaubten Operationen fest. Über den Bezeichner kann auf das Objekt zugegriffen werden.

Die verschiedenen in VHDL unterstützten Datentypen und die Objektdeklarationen werden in den folgenden Kapiteln beschrieben.

# VHDL-Datentypen (2 von 4)

```
-- Aufzählungs-Typen
type boolean      is (false, true);
type bit          is ('0', '1');

-- ganzzahlige Typen
type augenzahl    is range 1 to 6;
type word_length  is range 31 downto 0;

--Fliesskomma-Typen
type float         is range 1.0 to 10.0;
type interval      is range -5.00 to +5.00
```



## Aufzähltypen

Objekte dieses Typs können nur bestimmte Werte annehmen. Die endliche Anzahl von möglichen Werten wird in der Typendeklaration festgelegt. Der folgende Sourcecode zeigt einige Beispiele von Aufzähltypen, die bereits vordefiniert sind und in jedem VHDL-Modell eingesetzt werden können:

```
type char is ( ... ); -- (VHDL'87: 128 Zeichen; VHDL'93: 256 Zeichen)
type severity_level is (note, warning, error, failure)
```

## Ganzzahlige Typen

Ganzzahlige Typen werden durch die Angabe einer ganzzahligen Ober- und Untergrenze des möglichen Wertebereichs deklariert. Das obige Codefragment zeigt zwei Beispiele. Der Typ integer ist bereits vordefiniert und umfasst einen systemabhängigen Wertebereich.

## Fliesskommotypen

Fliesskommotypen werden entsprechend den ganzzahligen Typen deklariert. Der einzige Unterschied ist die Ober- und Untergrenze des Bereichs. Das obige Codefragment zeigt zwei Beispiele.

Der Typ real ist bereits vordefiniert und umfasst einen systemabhängigen Wertebereich.

### Zusatzbemerkungen:

- kompatibel: übergeordneter Typ kann einem untergeordneten zugewiesen werden: Integer kann einem Typ zugewiesen werden, der zur Menge Integer gehört
- Wortlänge: hier z. B. 32 bit

Leider gibt es keine Compiler-Anweisung, wie eine Floating-Point-Operation in eine Schaltung umgewandelt wird. Man kann Ausdrücke wie „Float“ verwenden, muss dann aber die Anweisung selbst in Einzelschritte entwickeln, wie die entsprechende Schaltung auszusehen hat. Das gilt z. B. für die Division und zum Teil die Multiplikation in entsprechender Weise.

```
-- Physikalischer Typ
type time is range ... -- systemabhängiger Bereich
units fs;               -- Basiseinheit: fs
    ps = 1000 fs;        -- abgeleitete Einheiten
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min =   60 sec;
    hr =    60 min;
end units;
```



## Physikalische Typen

Ein physikalischer Typ enthält Werte, die eine physikalische Größe, wie z. B. Zeit, Länge, Spannung, Strom etc. repräsentieren. Der vordefinierte Typ time ist ein Beispiel für einen physikalischen Typ.

*Zusatzbemerkung: sec und ms: schlecht, aber Standard*

```
-- abgeleitete Typen (vordefiniert):  
subtype natural is integer range 0 to highest_integer;  
subtype positive is integer range 1 to highest_integer;
```

```
-- zusammengesetzte Datentypen  
type byte_t is array (7 downto 0) of bit; -- constraint array  
type int_vec_t is array (positive range <>) of bit;  
-- unconstraint array
```



## Abgeleitete Typen

Man kann von bereits deklarierten Typen weitere Typen, sogenannte Subtypes, ableiten, die z. B. im Wertebereich eingeschränkt sind. Abgeleitete Typen bieten den Vorteil, dass Objekte mit dem selben Basistyp mit den Operatoren des Basistyps verknüpft werden können.

## Zusammengesetzte Datentypen

Ähnlich wie bei Programmiersprachen (Pascal, C, ...) gibt es in VHDL zwei Arten von zusammengesetzten Datentypen, zum einen records und zum anderen arrays. An dieser Stelle soll nur gezeigt werden, wie Vektortypen (eindimensionale Arrays) definiert und verwendet werden. Das folgenden Codefragment zeigt zwei unterschiedliche Möglichkeiten Vektoren zu definieren.

Im ersten Fall handelt es sich um einen beschränkten Vektortyp (Constraint Array), im zweiten Fall um einen unbeschränkten Vektortyp (unconstraint Array), bei dem nur ein Wertebereich für den Index angegeben wird. Bei unbeschränkten Vektortypen wird der konkrete Indexbereich erst bei der Objektdeklaration festgelegt.

Beispiele für vordefinierte Vektortypen sind:

```
type string is array (positive range <>) of character;  
type bit_vector is array (natural range <>) of bit;
```

## Signal

- „Hardware orientiert“
- besitzt eine Geschichte (vergangener Wert, momentaner Wert, zukünftiger Wert)
- Kommunikation zwischen Modulen, Prozessen

## Variable

- „Software orientiert“
- nur lokal innerhalb eines Prozesses, einer Prozedur oder Funktion

## Constant

- unveränderlicher Wert



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 44  
18.01.2013 14:03:23

## Objektdeklaration

Neben Konstanten und Variablen, deren Deklaration und Verwendung ähnlich wie in höheren Programmiersprachen ist, gibt es in VHDL auch noch Signale. Im Gegensatz zu Variablen wird bei Signalen der zeitliche Verlauf gespeichert, so dass auch auf Werte in der Vergangenheit zugegriffen werden kann. Außerdem ist es möglich, für ein Signal einen Wert in der Zukunft vorzusehen. Beispielsweise weist der Befehl

```
a <= '0' after 10 ns;
```

dem Signal a den Wert ‘0’ in 10 ns, vom momentanen Zeitpunkt ab gerechnet, zu. Die Ports einer Entity werden innerhalb der dazugehörigen Architecture wie Signale behandelt.

*Zusatzbemerkung:*

*HW orientiert: kommt aus der Gedankenwelt der HW*

*SW orientiert: kommt aus der Gedankenwelt der SW*

# Objektdeklaration

R /  $\overline{W}$

```
constant address : bit_vector(0 to 7):="10001100";
constant delay   : time := 5 ns;
variable counter : bit_vector(7 downto 0):= "00000000";
signal D        : bit_vector(7 downto 0):= "00000000";
signal RWB      : bit;
```

```
...
Counter := "10101010";
D<= "10101010";
RWB<='1' after 10 ns;
...
```



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

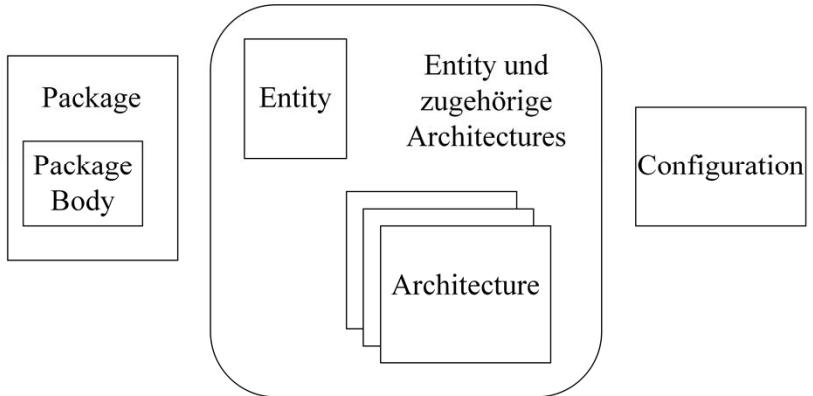
K6-1  
p. 45  
18.01.2013 14:03:23

Der obige Sourcecode zeigt einige Beispiele für die Deklaration von Konstanten, Variablen und Signalen.

*Zusatzbemerkung:*

- Wird eine Variable mit einem Wert versehen, bedeutet dies die Initialisierung.
- 7 downto 0) := "000000": die Anzahl der Nullen definiert die Wortbreite.

# VHDL Design Units

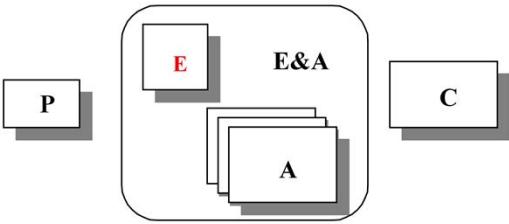


Ein komplettes VHDL-Modell besteht aus mehreren Teilen (Design Units): Einer Schnittstellenbeschreibung, der Entity (einer oder mehrerer Verhaltens- oder Strukturbeschreibungen), den Architectures, und gegebenenfalls mehreren Konfigurationen (Configurations), die die Verbindung zwischen Entity und Architecture herstellen. Werden bestimmte Objekttypen, Funktionen oder Prozeduren von mehreren Modellen benötigt, so werden sie üblicherweise in unabhängigen Einheiten, den sogenannten Packages, abgelegt.

Oben sieht man, dass zu einer Entity mehrere Architectures definiert sein können. Die Configuration wählt davon eine Architecture aus und legt auch für eventuelle Untereinheiten fest, welche Entity-Architecture-Paare verwendet werden (siehe Configuration). Entity, Architecture und Configuration können von den in einem Package definierten Objekten Gebrauch machen.

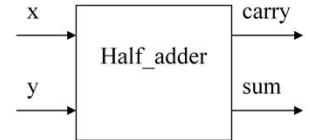
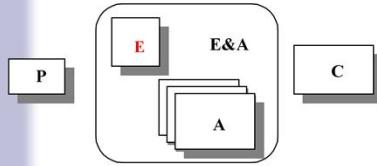
Zusatzbemerkung: Die Entity beschreibt also die Schnittstelle zur Außenwelt durch die Festlegung der PORTs (Klemmen) nach "außen". Die Architecture beschreibt das elektrische Verhalten und wird der Entity zugeordnet.

# Entity



- enthält die gemeinsame Information für die zugehörigen Architectures
- definiert die externe Schnittstelle eines Modells:
  - Ports
  - Generic Parameters

Die Entity definiert die externe Sicht (Blackbox-Betrachtung) eines Modells: Ports und Parameter (Generics) sowie statische Überprüfung der Parameter Werte (z. B. Wertebereich) und dynamische Überprüfung der Ports (z. B. Setup oder Hold Time Verification). Im Folgenden werden zwei Beispiele, ein Halbaddierer und ein 8-fach-DFF, vorgestellt.



## Halbaddierer

```
entity Half_adder is
  port(x, y      : in bit;
       sum, carry : out bit);
end Half_adder;
```

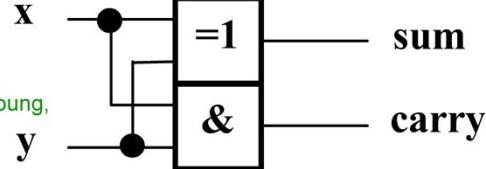
Die Entity mit dem Name Half\_adder hat zwei Eingänge, x und y (sie haben den Modus **in**), und zwei Ausgänge, s und cout (sie haben den Modus **out**). Neben **in** und **out** ist auch noch der Modus **inout** wichtig. Er wird für bidirektionale Ports verwendet. Der Typ bit ist in VHDL vordefiniert und kann nur die Werte '0' oder '1' annehmen.

# Entity Declaration

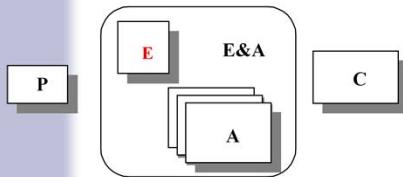
The entity declaration specifies the name of the entity and lists the set of interface ports:

```
entity Half_adder is
    port (x, y      : in  bit;
          sum, carry : out bit);
end Half_adder;
```

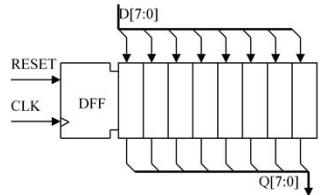
-- Dies ist die Entity-Deklaration,  
-- Name + Ports – doch keine Beschreibung,  
-- der inneren Funktionen,  
-- bit: enumeration type for '0' and '1',  
....



- Aufzählung (enumeration) der Ein- und Ausgänge nur in Bit-Form.



## 8-fach-DFF



```

entity dff8 is
  generic (delay      : time := 4 ns );
  port    (clk, reset : in bit;
           D          : in bit_vector(7 downto 0);
           Q          : out bit_vector(7 downto 0));
end dff8;
  
```

In der Entity dff8 wird zusätzlich ein Parameter (Generic) delay vom Typ time definiert. Mit Hilfe von Generics lassen sich beispielsweise die Verzögerungs- oder Setup-Zeiten eines Modells von außen an das Modell übergeben.

An den vorangegangenen Beispielen sieht man, dass in der Entity keine Information über den internen Aufbau eines VHDL-Modells enthalten ist.

### Zusatzbemerkung:

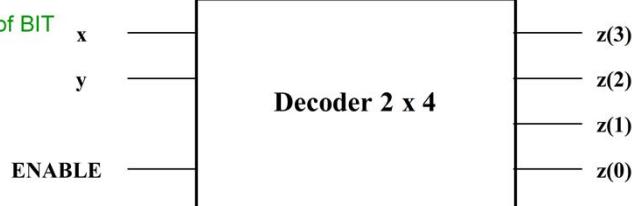
*Die Angabe "generic 4 ns" zur Synthese nicht angegeben werden, nur wenn z. B. solch ein System ohne genaue technologische Angabe vernünftig simulieren soll (dann kann es angegeben werden). Es steht hier nicht, wo das delay greift, das muss in der architecture deklariert werden.*

# Entity Declaration

The Entity Declaration specifies the name of the Entity and lists the set of interface ports:

```
entity DECODER2x4 is
  port (x, y, ENABLE: in BIT; z: out BIT_VECTOR(0 to 3);
end DECODER2x4;
```

-- BIT\_VECTOR: predefined  
-- unconstrained array type of BIT  
....



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p.51

18.01.2013 14:03:23

# Architecture Body

Eine entity ist spezifiziert durch den architecture body:

- ❖ ein Set von interconnected Components  
*oder*
- ❖ ein Set von concurrent assignment Statements  
*(beschreibt einen Datenfluss)*  
*oder*
- ❖ ein Set von sequential assignment Statements  
*(beschreibt ein Verhalten)*  
*oder*
- ❖ eine Kombination



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

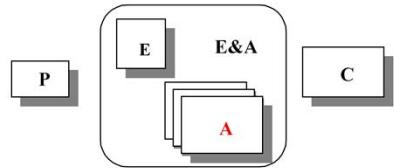
K6-1

p. 52

18.01.2013 14:03:23

Die Architecture führt zur Beschreibung der Funktionalität eines Modells. Für eine Entity können mehrere Architectures definiert werden, wobei über die Configuration dann jeweils eine Architecture ausgewählt wird. Jede dieser Architectures stellt eine alternative Sichtweise der Hardware dar. Das können z. B. Beschreibungen auf unterschiedlichen Abstraktionsebenen oder verschiedene Entwurfsvarianten sein.

# Architecture



- Jede Architecture stellt eine alternative Sicht der HW dar.
- Nichts in der Architecture ist extern sichtbar.
- Eine Architecture kann beschrieben werden durch:
  - Struktur (3)
  - Verhalten
    - nebenläufige Anweisungen (concurrent Statements) (1)
    - sequentielle Anweisungen (sequential Statements) (2)
    - eine Kombination von beidem



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 53

18.01.2013 14:03:23

Im vorgestellten Y-Diagramm werden drei Sichten unterschieden. Die Sprache VHDL ermöglicht eine Beschreibung in der strukturalen Sicht und in der Verhaltenssicht. Es wird bei einem VHDL-Modell daher immer zwischen Verhaltensmodellierung (Behavioral Modelling) und Strukturmodellierung (Structural Modelling) unterschieden. Die eindeutige Einordnung eines VHDL-Modells in eine der beiden Modellierungsarten ist nicht immer möglich, da VHDL die Verwendung beider Beschreibungsarten innerhalb eines Modells gestattet.

*Zusatzbemerkung: die E/A-Leitungen werden in der entity definiert.*

# Behavioural Description



```
architecture Behavioural_par of Half_adder is
begin
    carry <= x AND y;
    sum  <= x XOR y;
end Behavioural_par;
```

## (1) nebenläufige Anweisungen (concurrent Statements)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1  
p. 54  
18.01.2013 14:03:23

In der Verhaltenssichtweise werden zwei prinzipielle Beschreibungsmittel unterschieden:

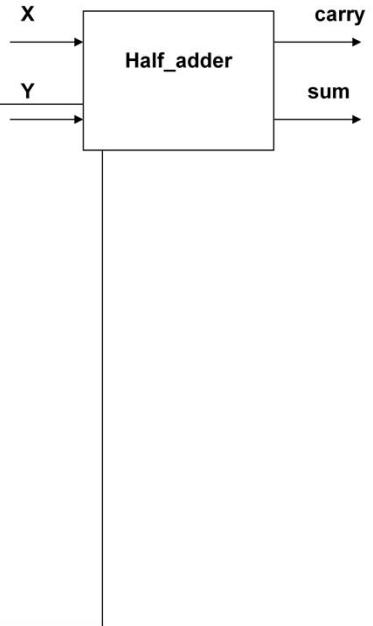
- sequentielle Anweisungen (sequential statements)
- nebenläufige Anweisungen (concurrent statements)

*Nebenläufige Anweisungen:*

Im Gegensatz zu den gängigen Programmiersprachen mit ihren sequentiellen Konstrukten verfügt VHDL zusätzlich noch über nebenläufige Anweisungen, die es erlauben, parallel ablaufende Operationen zu beschreiben. Damit wird es möglich, die spezifischen Eigenschaften von Hardware (parallel arbeitende Funktionseinheiten) abzubilden.

Der obige Sourcecode zeigt die Modellierung des Halbaddierers mit Hilfe von nebenläufigen Signalzuweisungen.

# Behavioural Description



## (2) sequentielle Anweisungen (prozedurale Beschreibung)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 55

18.01.2013 14:03:23

### Sequentielle Anweisungen:

In sequentiellen oder auch prozeduralen Beschreibungen werden Konstrukte wie Verzweigungen (IF-ELSIF-ELSE), Schleifen (LOOP) oder Unterprogrammaufrufe verwendet. Die einzelnen Anweisungen werden nacheinander (sequentiell) abgearbeitet. Die Beschreibung ähnelt den Quelltexten höherer Programmiersprachen. Sequentielle Anweisungen können nur innerhalb eines Prozesses oder innerhalb von Funktionen und Prozeduren stehen. Der Prozess selbst ist jedoch eine nebenläufige Anweisung.

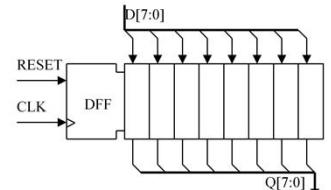
Die obige Architecture zeigt eine sequentielle Modellierung des Halbaddierers.

Die Architecture Behavioural\_seq enthält einen Prozess mit dem Namen **combi**. Auf das Schlüsselwort **process** folgt in runden Klammern eine Liste von sensitiven Signalen (Sensitivity List). Der Prozess wird einmalig bei der Initialisierung durchlaufen und zu einem späteren Zeitpunkt erst wieder aktiv, wenn sich eines der Signale in der Sensitivity List ändert. Ändert sich also eines der Signale x oder y, dann wird der Prozess **combi** ausgeführt und den Signalen s und cout in Abhängigkeit von x und y ein neuer Wert zugewiesen.

Wenn kombinatorische Hardware beschrieben werden soll, muss man besonders auf die folgenden Dinge achten:

- Alle gelesenen Signale müssen in der Sensitivity List stehen. Würde z. B. das Signal x in der Sensitivity List fehlen, dann würden s und cout bei Änderung von x den alten Wert beibehalten, und die beschriebene Hardware würde daher speichernde Elemente enthalten.
- Allen output-Signalen muss ein Wert zugewiesen werden, egal welcher Zweig ausgeführt wird.

# Behavioural Description



```
architecture Behavioural of dff8 is
begin
    synch:process(clk, reset)
    begin
        if reset = '1' then
            q <= "00000000";
        elsif clk'event and clk = '1' then
            q <= d after delay;
        end if;
    end process;
end Behavioural;
```

## sequentielle Anweisungen (prozedurale Beschreibung)



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 56

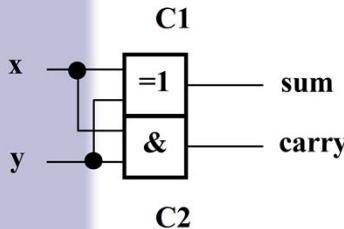
18.01.2013 14:03:23

Der obige Sourcecode zeigt die Architecture des 8-fach-D-Flip-Flops. In dieser Architecture stehen nicht alle gelesenen Signale in der Sensitivity List (das Signal d fehlt), daher wird die resultierende Hardware speichernde Elemente (Flip-Flops) enthalten.

Sobald der Eingang reset gesetzt wird, wird das 8-fach-Flip-Flop gelöscht. Ist reset hingegen nicht gesetzt, dann werden mit der steigenden Taktflanke die Daten von d auf q übernommen. Um eine steigende Flanke am Takeeingang zu detektieren, wird das Attribut event des Signals clk abgefragt. Attribute können allgemein dazu verwendet werden, um bestimmte Eigenschaften von Objekten oder Typen abzufragen. Attribute werden verwendet, indem man den Objekt- oder Typennamen gefolgt von einem Tick (') und dem Attributnamen angibt. Der Ausdruck clk'event liefert true, falls beim Signal clk während des aktuellen Simulationszyklus ein Zustandswechsel auftritt, sonst false. Diese Abfrage ist deshalb notwendig, da der Prozess synch auch dann ausgeführt wird, wenn das Signal reset von '1' auf '0' wechselt.

# Architecture Body

(3) as a set of interconnected components:



```
architecture HALF_AD_STRUCTURE of HALF_ADDER
is
-- declarative part:
component XOR2
    port (a, b: in BIT; c: out BIT);
end component;
component AND2
    port (d, e: in BIT; f: out BIT);
end component;
-- statement part (instantiation):
begin
    C1: XOR2 port map (x, y, sum);
    C2: AND2 port map (x, y, carry);
end HALF_AD_STRUCTURE;
-- name of architecture body: HALF_AD_STRUCTURE
-- entity declaration: HALF_ADDER
```



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 57

18.01.2013 14:03:23

Instantiation: „Zustandseinheit“

Folgende Verbindung zw. a, b und x, y sind möglich:

1. positional association (siehe oben)

2. named association: a=>x, ..

declarative part: describing the units in a common way

instantiation part: referring to the concrete circuit which is to design

# Architecture Body

**architecture DEC of DECODER2x4 is**

-- declarative part:

**component INV**

port (a: in BIT; b: out BIT);

**end component;**

**component NAND3**

port (d, e, f: in BIT; z: out BIT);

**end component;**

**signal s1, s0: BIT;**

-- statement part (instantiation):

**begin**

I1: INV port map (x, s1);

I0: INV port map (y, s0);

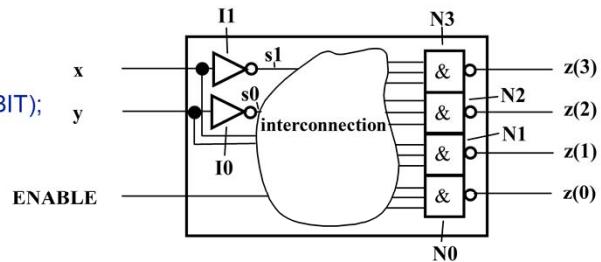
N3: NAND3 port map (x, y, ENABLE, z(3));

N2: NAND3 port map (x, s1, ENABLE, z(2));

N1: NAND3 port map (y, s0, ENABLE, z(1));

N0: NAND3 port map (s1, s0, ENABLE, z(0));

**end DEC;**



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 58

18.01.2013 14:03:23

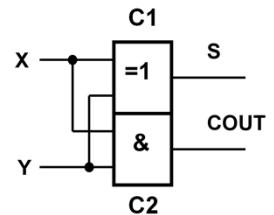
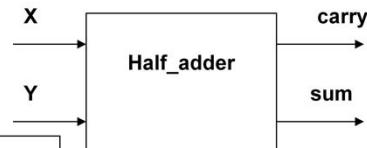
definition of the internal signals

# Structural Description

```

architecture Structure of Half_adder is
component and_gate
    generic (and_blk_delay:time:=4 ns);
    port (a, b      : in bit;
          c        : out bit );
end component;
component xor_gate
    generic (xor_blk_delay:time:=4 ns);
    port (a, b      : in bit;
          c        : out bit );
end component;
begin
    C1:xor_gate generic map(xor_blk_delay=>3 ns)
        port map( a=>x, b=>y, c=>s);
        -- named association
    C2:and_gate generic map(4 ns)
        port map( x, y, cout);
        -- positional association
end Structure;

```



Bei der strukturalen Modellierung wird das Verhalten eines Modells durch die Verbindung von Unterkomponenten dargestellt. Die Eigenschaften der Unterkomponenten werden in unabhängigen VHDL-Modellen beschrieben, die compiliert in Modellbibliotheken (Libraries) zur Verfügung stehen. Die strukturelle Modellierung ermöglicht komplexe Systeme in einzelne Untermodule zu zerlegen und diese getrennt voneinander zu implementieren.

Zunächst werden die verwendeten Komponenten deklariert (in diesem Fall ein AND- und ein XOR-Gatter). Dabei werden neben den Ports auch die zu übergebenden Parameter aufgeführt. Die Komponentendeklaration ist somit ein Abbild der Entity des einzusetzenden Modells. Nach dem Schlüsselwort begin werden die Komponenten instanziert. Dabei erhält jede Komponente einen Referenznamen (C1 und C2). Es existieren verschiedene Möglichkeiten, die Ports und Parameter einer Komponente zu jenen der Instanz zuzuordnen:

- Positional Association: Die Zuordnung ist durch die Position festgelegt.
- Named Association: Die einzelnen Elemente können mit dem Zuweisungssymbol „=>“ direkt angesprochen werden, die Reihenfolge ist beliebig.

Für dieses Beispiel ist es noch notwendig, festzulegen, welche Entity-Architecture-Paare für die Instanzen C1 und C2 verwendet werden. Diese Festlegung kann mit Hilfe der im nächsten Abschnitt beschriebenen Configuration erfolgen.

*Zusatzbemerkung: C1, C2, .. sind eindeutig und zählen alle Komponenten auf. Wenn also mehrere „and\_gate“ existieren, ist damit ebenfalls die Unterscheidung möglich.,*

# Wait Statement



```
counter: process
  variable count_val : integer:=0;
begin
  wait until clk='1';
  count_val:=(count_val+1)mod 256;
  if count_val=255 then
    count_out<='1';
  else
    count_out<='0';
  end if;
end process;
```

<b>wait on A, B;</b>	-- warte bis sich A oder B ändert
<b>wait until clk='1';</b>	-- warte bis Bedingung erfüllt
<b>wait for 10 ns;</b>	-- warte 10 ns
<b>wait;</b>	-- warte für immer



wait until 1, also bis die 0-1-Flanke kommt

Das obige Beispiel zeigt außerdem die Verwendung von Variablen in VHDL. Die Deklaration von Variablen darf nur im Deklarationsteil von Prozessen, Funktionen oder Prozeduren stehen. Um zwischen Prozessen oder Modulen zu kommunizieren, können daher nur Signale verwendet werden. Man sieht auch, dass in VHDL zwischen einer Zuweisung an eine Variable (`:=`) und an ein Signal (`<=`) unterschieden wird. ..

```

architecture ... of ... is

signal count_out : bit;

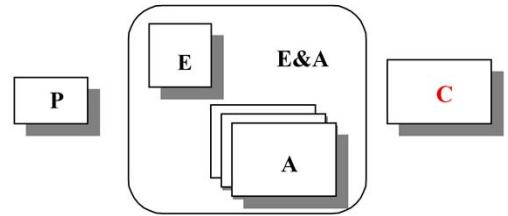
begin
:
:
counter:process
variable count_val : integer:=0;
begin
  wait until clk='1';
  count_val:=(count_val+1)      mod
256;
  if count_val=255 then
    count_out<='1';
  else
    count_out<='0';
  end if;
end process;
:
:
end ...;
```

**Möglichkeit, um  
Prozesse zu aktivieren  
und zu stoppen**



Das obige Beispiel zeigt eine andere Möglichkeit, um Prozesse zu aktivieren und zu stoppen: die WAIT Anweisung.

WAIT-Anweisungen dürfen nur in Prozessen ohne Sensitivity List auftreten. Als Argument für eine WAIT-Anweisung können ein oder mehrere Signale, Bedingungen oder Zeitangaben verwendet werden. Ein Wait ohne jegliches Argument bedeutet „warte für immer“ und beendet somit die Ausführung eines Prozesses. Prozesse können auch mehrere WAIT-Anweisungen enthalten.



# Configuration

```
configuration Half_adder_conf of Half_adder is
    for Structure
        for C1:xor_gate use entity work.xor_gate(xor_bhv);
        end for;
        for C2:and_gate use entity work.and_gate(and_bhv);
        end for;
    end for;
end Half_adder_conf;
```



Configuration: Auswahl der Architektur und der Submodule

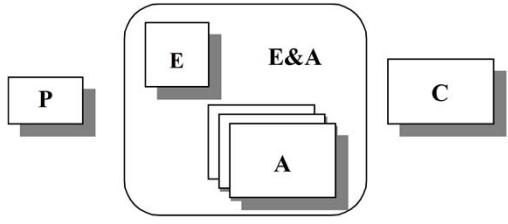
Die Configuration legt fest, welche der beschriebenen Architekturvarianten einer bestimmten Entity zugeordnet ist, und welche Entity-Architecture-Paare für die Submodule verwendet werden. Es können hier außerdem bestimmte Parameter für tiefere logische Ebenen zugewiesen werden.

Eine Konfiguration für den Halbaddierer sieht beispielsweise wie in Bild oben aus.

Die Konfiguration beginnt mit der Vereinbarung des Namens (Half\_adder\_conf). Danach folgt der Name der Entity, die konfiguriert wird (Half\_adder), und die Angabe, auf welche Architecture (Structure) sich die Konfiguration bezieht. In den **for**-Klauseln werden die zu konfigurernden Instanzen und deren Komponentennamen angegeben (z. B. C1:xor\_gate). In der folgenden **use**-Klausel wird das Entity-Architecture-Paar an die Instanz gebunden. So wird zum Beispiel in der ersten **use**-Klausel festgelegt, dass für die Instanz C1 der Komponente xor\_gate die Entity xor\_gate und die Architecture xor\_bhv aus der Bibliothek work verwendet werden

*Zusatzbemerkung: Die Bibliothek work hat eine besondere Bedeutung. In dieser Bibliothek wird die Library Unit eingefügt, die bei der Analyse einer Design Unit entsteht.*

# Package



```
package example_pack is
    type mvl_val is ('u', '0', '1', 'd', '*');
    type mvl_array is array (integer range <>) of mvl_val;
    constant delay : time;
end example_pack;
```

```
package body example_pack is
    constant delay : time := 2 ns;
end example_pack;
```



## Package Declaration and Package Body

Anweisungen wie Typ- oder Objektdeklarationen und die Beschreibung von Prozeduren und Funktionen, die in mehreren Design Units gebraucht werden, können in einem Package zusammengefasst werden. Ein Package ist also eine Bibliothek, die vordefinierte sowie frei wählbare Funktionen und Datentypen enthält. Der obige Sourcecode zeigt ein Beispiel für ein Package.

Es fällt auf, dass das Package aus zwei Teilen besteht:

- Die Package Declaration definiert die Schnittstelle zum Package. Sie enthält die Deklaration von Typen, Konstanten, Funktionen, usw., die anderen Design Units zugänglich gemacht werden können.
- Der Package Body ist optional und enthält die versteckten Details eines Packages, wie z. B. die Werte von Konstantendeklarationen oder Funktionskörpern.

Der Zusammenhang von Package Declaration und Package Body wird durch den identischen Namen hergestellt. Die Aufteilung in Package Declaration und Package Body bietet den Vorteil, dass bei einer Änderung im Package Body nicht das gesamte Design neu compiliert werden muss.

*Zusatzbemerkung: <> bedeutet: ist nicht angegeben*

# Library und Use Statement

```
library my_lib
use my_lib.example_package.all

entity example is ...
```

```
library my_lib
use my_lib.example_package.delay

architecture example_arch of example is ...
```



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 64  
18.01.2013 14:03:23

Auf die in einem Package enthaltenen Definitionen kann von anderen Design Units aus zugegriffen werden, indem die **library**- und die **use**-Anweisung verwendet werden. Angenommen, das obige Package wird in eine Bibliothek mit dem Namen `my_lib` kompiliert, dann kann die entity `example` das Package verwenden, wie es im oberen Kasten formuliert worden ist.

Mit der **library**-Anweisung wird zunächst die verwendete Bibliothek (`my_lib`) der Design Unit bekanntgegeben; der Name `my_lib` kann daher in der Beschreibung verwendet werden. Darauf folgt eine **use**-Anweisung, die alle Deklarationen im Package `my_lib` in die entity `example` importiert. Die **library**- und **use**-Anweisung beziehen sich immer auf die nachfolgende Design Unit.

Es ist auch möglich, nur ausgewählte Deklarationen von einer Package Deklaration in eine andere Design Unit zu importieren: siehe untenen Kasten im Bild oben.

# Std\_logic\_1164

```
type std_ulogic is ('U',      -- uninitialized
                     'X',      -- forcing unknown
                     '0',      -- forcing 0
                     '1',      -- forcing 1
                     'Z',      -- high impedance
                     'W',      -- weak unknown
                     'L',      -- weak 0
                     'H',      -- weak 1
                     '-',      -- don't care
```

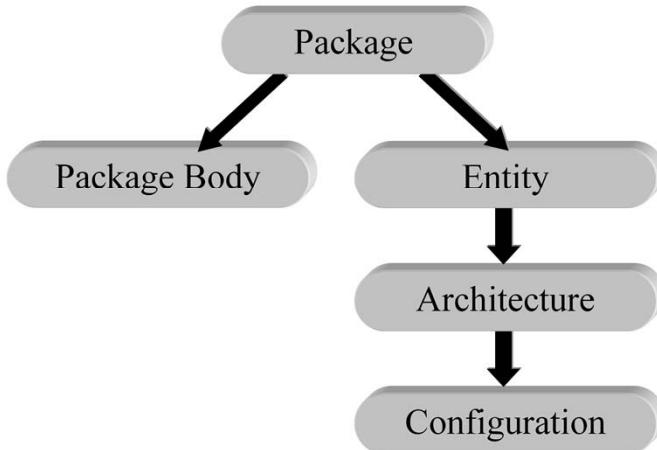


Ein sehr wichtiges Package ist das Package std\_logic\_1164 in der Bibliothek IEEE. In diesem Package werden die Typen std\_ulogic und std\_ulogic\_vector definiert. Diese Typen haben 9 mögliche Werte und ermöglichen es daher digitale Schaltungen genauer zu modellieren, als dies mit den Typen bit und bit\_vector möglich ist. Der obige Sourcecode zeigt die Definition des Typs std\_ulogic.

*Zusatzbemerkung: Es gibt Typen: resolved und unresolved. Resolved: über eine Tabelle können dem Bus, auf den mehrere Treiber zugreifen können, bei Standard-Logik 9 verschiedene Zustände zugewiesen werden. Auf "unresolved"-Leitungen kann nur ein Treiber zugreifen.*

1. Da VHDL ursprünglich für die Systemsimulation entwickelt worden ist, und die Forschung erst nachträglich erkannt hat, dass man die Sprache ja auch für die Synthese verwenden kann, bestand das Problem, elektrische Systeme möglichst genau zu Beschreiben. Daher musste eine Möglichkeit geschaffen werden, mehrere Treiber auf einer Leitung mit allen möglichen Auswirkungen behandeln zu können. Dies wird mit Hilfe von „Resolution Functions“ erreicht, die man zu beliebigen VHDL-Datentypen erstellen kann. Im Wesentlichen handelt es sich dabei nur um eine Übergangstabelle in der aufgezeichnet ist, auf welchen Wert ein Signal gesetzt werden muss, wenn zwei Treiber auf dieses Signal gleichzeitig aktiv sind. Eine „Resolution Function“ hat damit zwei Eingänge und einen Ausgang – bei mehr als zwei Treibern wird sie mehrmals rekursiv angewendet.
2. Ein Datentyp der „resolved“ ist, besitzt somit eine „Resolution Function“. Umgekehrt, wenn ein Datentyp „unresolved“ ist, so ist im Sourcecode keine „Resolution Function“ für das Signal angegeben worden.
3. Der Datentyp std\_logic (ohne u!) wird daher für Signale verwendet, die elektrischen Leitungen in einem System entsprechen. Nur damit sind Bussystem simulierbar
4. Der Datentyp std\_ulogic ist „unresolved“ und kann daher nicht für Leitungen mit mehr als einem Treiber verwendet werden. Im Gegenteil, setzt man mehr als eine Zuweisung auf solch ein Signal drauf (mehr als ein Treiber), ist das ein Syntaxfehler, und das System ist nicht übersetzbare oder simulierbar. „Unresolved“-Datentypen sind daher mit viel mehr Einschränkungen behaftet als „Resolved“-Datentypen und werden daher bei Designs angewendet, bei dem extrem strikte Randbedingungen existieren – zum Beispiel beim Design von ASICs. Dort sind mehrfache Treiber auf den Leitungen sowieso verboten, daher ist es in diesem Bereich sogar sehr sinnvoll nur „Unresolved“-Datentypen zu verwenden, da damit formale Designfehler bereits vom Synthesetool zurückgeworfen werden.

# Abhängigkeiten der Design Units



## *Abhängigkeiten beim Compilieren*

Bild oben zeigt die Abhängigkeiten beim Compilieren der Design Units. So müssen zum Beispiel nach einer Änderung in einem Package alle zugehörigen Design Units neu compiliert werden. Eine Änderung im Package Body erfordert dagegen nur das erneute Compilieren dieser Design Unit.

# Um die ersten kleinen Programme zu schreiben, fehlen noch:

- **Configuration Declaration**
  - selection of the architecture bodies
  - to bind components
  - ..
- **Package Declaration**
  - to store a set of common declarations
- **Model Analysis**
  - to validate a unit
- **Simulation**
  - to validate a unit



Institut für Computertechnik

Digitale Integrierte Schaltungen  
DIS

K6-1

p. 67

18.01.2013 14:03:23