# 1 Hardwareentwurf

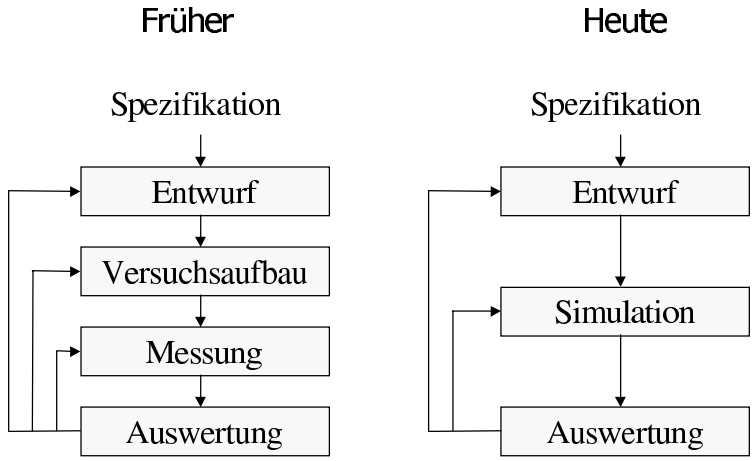
- **1.1** Überblick, Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwaresimulation-/Verifikation
  - Simulation
  - Formale Verifikation
  - Timinganalyse
- **1.4** Hardwaresynthese
- **1.5** Plazierung und Verdrahtung



# Literatur


- Thomas Kropf, Introduction to Formal Hardware Verification, Springer Heidelberg, ISBN 3-540-65445-3
- Franz Rammig, Systematischer Entwurf digitaler Systeme, Teubner Stuttgart, ISBN 3-519-02265-6
- Drechsler und Becker, Graphenbasierte Funktionsdarstellung, B. G. Teubner Stuttgart, ISBN 3-519-02149-8vvorlesung
- Shi-Yu Huang, Kwang-Ting Cheng, Formal Equivalence Checking and Design Debugging Kluwer Academic Publishers, ISBN 0-7923-8184-X

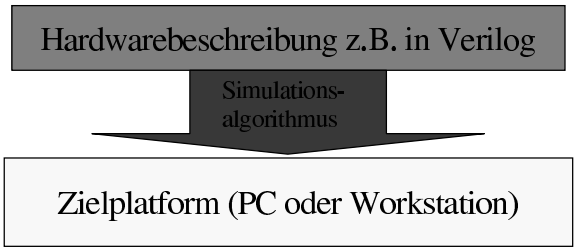
## Motivation






## Simulationstechniken

Aufgabe eines Simulationsalgorithmus:
Abbilden der Hardwarefunktionalität auf eine Zielarchitektur, z.B. von Neumann Rechner




Problem:
hoher Grad an Parallelität muß auf eine sequentielle Maschine „projeziert" werden

## Arten der HW-Simulation

- Analogsimulation (Spice, ...)
  - werte- und zeitkontinuierlich
  - nichtlineare Differentialgleichungen
- Digitalsimulation (VSS, modelsim, ...)
  - werte- und zeitdiskret
  - boolesche Algebra
- Mixed-Mode/Simulatorkopplung (Saber,...)
  - analog/digital-Simulation



## Logiksimulationsalgorithmen

- Streamline Code Simulation
  - Schaltnetze, vollsynchrone Schaltwerke
- Equitemporal Iteration
  - Analogsimulation
- Critical Event Scheduling
  - Schaltungen auf unterschiedlichen Abstraktionsebenen und mit Zeitinformation

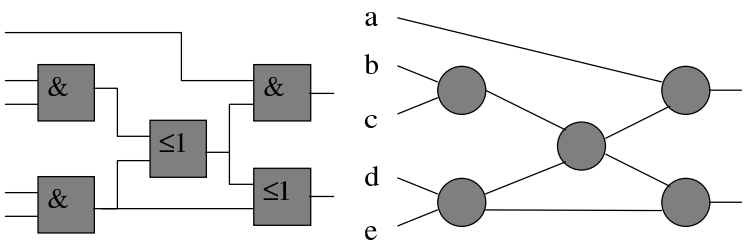
## Streamline Code Simulation

- auch „Compiled Mode"
- es wird direkt ausführbarer Code auf der Zielplattform generiert
- Einschränkungen
  - kombinatorische oder strikt synchrone Schaltungen
  - keine Zeitinformationen



## Streamline Code Simulation II

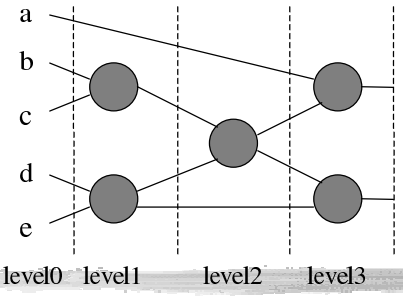
Schaltung wird als gerichteter azyklischer Graph notiert:



## Streamline Code Simulation III

Knoten des azyklischen Graphes werden halbgeordnet (levelizing)

- level($v_i$) = 1+ max { level($v_k$) | es gibt eine Kante von $v_k$ nach $v_i$ }
- Primäreingänge: level($v_{in}$) = 0






## Streamline Code Simulation IV

Eigenschaften des Graphen:

- Knoten auf einer höheren Ebene können Knoten auf niedrigeren Ebenen nicht beeinflussen
- Knoten auf der selben Ebene beeinflussen sich gegenseitig nicht
- Knoten auf niedrigeren Ebenen beeinflussen Knoten auf höheren Ebenen

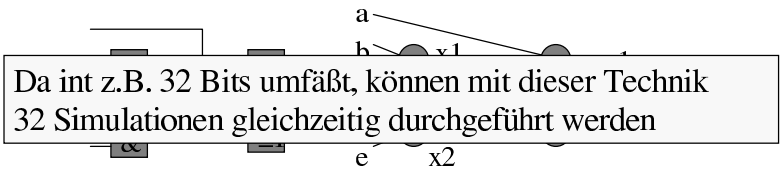
## Streamline Code Simulation V

Codegenerierung:

- Die Level werden nacheinander implementiert
- Die Reihenfolge der Operationen in den Levels ist beliebig
- Die Gatter werden durch Operatoren der Zielmaschine realisiert
- Die Verbindungen (Netze) werden durch Variablen der Zielmaschine implementiert



## Streamline Code Simulation VI




Da int z.B. 32 Bits umfäßt, können mit dieser Technik 32 Simulationen gleichzeitig durchgeführt werden

```
int a, b, c, d, e;
int x1, x2, x3;
int o1, o2;

x1 = b & c;    // level 1
x2 = d & e;    // level 1
x3 = x1 | x2;  // level 2
o1 = a & x3;   // level 3
o2 = x2 & x3;  // level 3
```

## Critical Event Scheduling

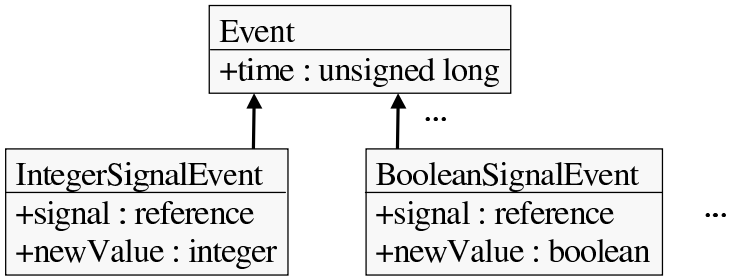
- IDEE:
  - Verwaltung algorithmischer Blöcke durch Threads
  - nur die Systemteile werden neu berechnet, deren Eingaben sich verändert haben (anstehende Ereignisse=Events)
  - globale Datenstruktur: EventQueue
  - Ein spezieller Thread übernimmt die Verwaltung: Scheduler



## Critical Event Scheduling III

Ereignisse

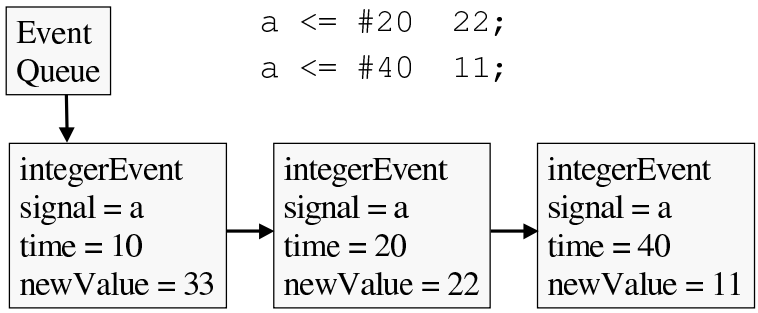
- sind Änderung eines Signalwertes oder werden von Zeitoperatoren (#) generiert
- treten zu einem festen Zeitpunkt auf



## Critical Event Scheduling IV

Simulationsalgorithmus speichert eine sortierte Liste der anstehenden Ereignisse

```
a <= #10  33;
a <= #20  22;
a <= #40  11;
```






## Critical Event Scheduling V

Um bei Eintreten von Ereignissen die Simulation weitertreiben zu können müssen die Threads gespeichert werden, die gerade auf Events warten

⇒ Zu jedem Event werden die Threads (Blöcke, Tasks etc.) gespeichert, die gerade sensitiv auf diese Event warten

| Event | |
|---|---|
| +time : unsigned long | // event time |
| +void notify(); | // activate all waiting threads |
| #waiting : List of threads | // threads waiting on this event |

## Critical Event Scheduling VI

```
while (currentTime <= finalTime && !queue.empty()){
  currentEvent = queue.top();
  currentTime = currentEvent.time;
  notify all threads t waiting on currentEvent {
      execute t until next wait/#/@ statement
            // hier werden eventuell neue Ereignisse
            // in die Queue eingeführt
            // beim Erreichen von wait trägt sich der
            // thread in die „Warteliste" des Events
  ein
}}}
```



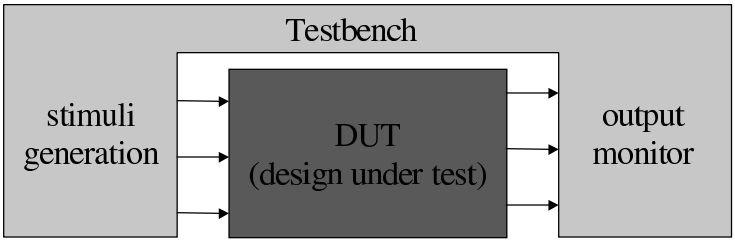
## Critical Event Scheduling VII

Durch diese ereignisbasierte Simulationstechnik ergeben sich zwei Zeitachsen:

- Simulationszeit
- Deltazeit (delta delay)

Deltadelays entstehen durch Abarbeitung mehrerer Ereignisse, die zur gleichen Simulationszeit auftreten

## Verifikation durch Simulation




- Eingangsstimuli erzeugen
- Ausgangssignale beobachten/verifizieren



## Formale Verifikation

- Äquivalenzprüfung
  - sind zwei Schaltungen äquivalent
- Modellprüfung
  - erfüllt eine Schaltung bestimmte (sequentielle) Eigenschaften
- Theorembeweisen
  - interaktive Methode, basierend auf Logik höherer Ordnung

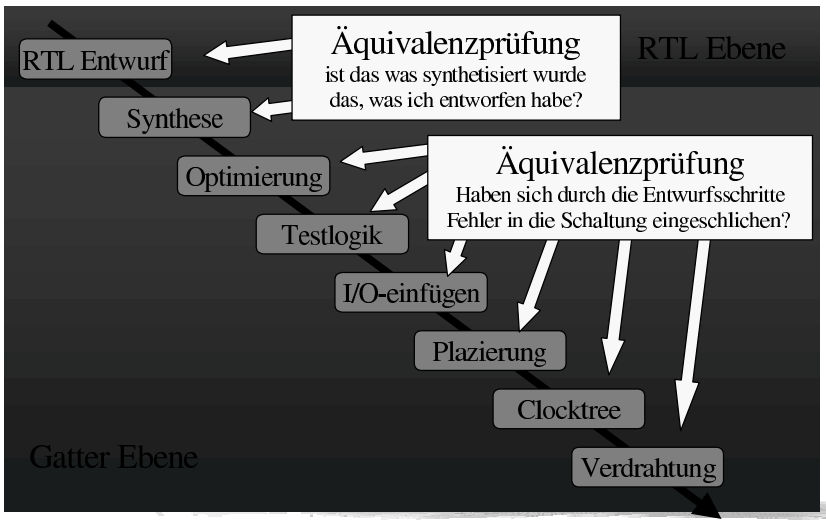
## Was ist Äquivalenzprüfung

Engl. „*equivalence checking*" (EC)

- gegeben: zwei digitale Schaltungen
- gefragt: haben beide die gleiche Funktionalität (keine zeitliches Verhalten)
  - bei kombinatorischen Schaltungen: sind die Ausgänge bei gleichen Eingangsbelegungen gleich?
  - Bei sequentiellen Schaltungen: sind die Ausgänge zu allen Zeitpunkten bei gleichen Eingabefolgen identisch?

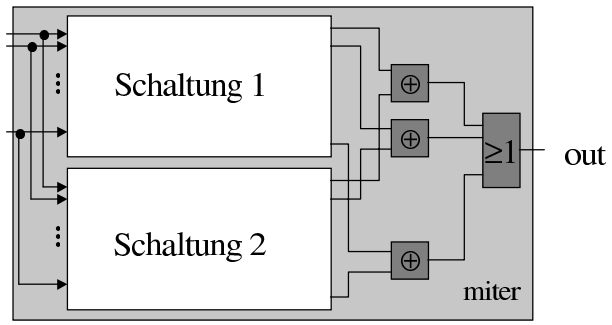


## Designflow mit Äquivalenzprüfung



## Äquivalenzprüfung

- Erzeuge „miter"
- Ist der Ausgang für alle Eingangsbelegungen „false"?






## Äquivalenzprüfung von Schaltnetzen

Vorgehen

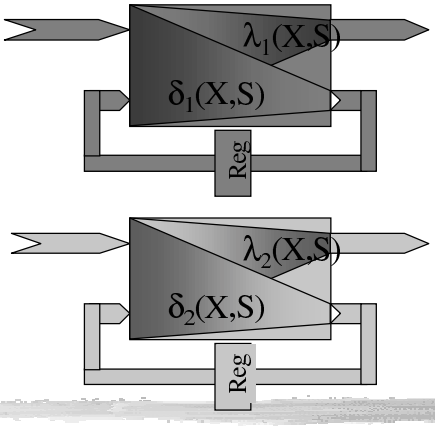
- Transformation der Schaltnetze in eine Normalformdarstellung (z.B. KNF, ROBDDs,...)
- Liegt Äquivalenz der Normalformen vor?

Optimierungen

- Ausnutzen von strukturellen Ähnlichkeiten
- inkrementelle Verfahren

## Äquivalenzprüfung für Schaltwerke

- Schaltungen in Huffman-Normalform

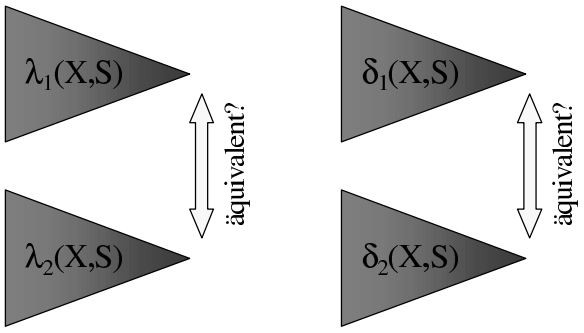





## Äquivalenzprüfung für Schaltwerke II

Gleiche Zustandskodierung und eindeutiger Resetzustand: Reduktion auf Äquivalenzprüfung von Schaltnetzen

Für jeden Ausgang und für jedes FlipFlop:



## Äquivalenzprüfung für Schaltwerke IV

Was ist, wenn man keine gleiche Zustandskodierung hat?

Oder keine gemeinsame Rücksetzleitung?

⇒ Einsatz von Techniken und Methoden der Automatentheorie

- Produktautomaten bilden mit „miter"-Ausgang
- Durchsuchen des Zusandsraumes nach Zuständen, die „true" am Ausgang erzeugen

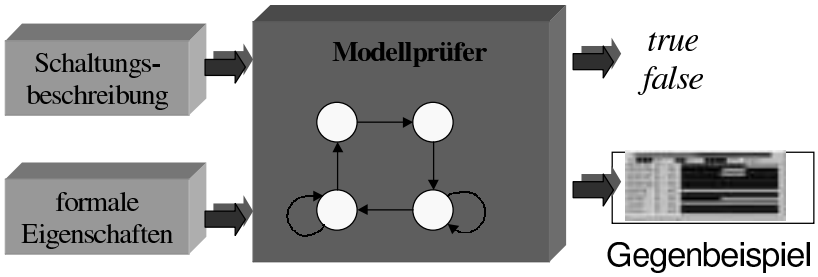


## Modellprüfung

Erfüllt ein Schaltwerk eine gegebene Spezifikation?

- Modellierung des Schaltwerkes durch endliche Automaten
- Spezifikation der Eigenschaften in temporalen (modalen) Aussagenlogiken

## Modellprüfung






## Temporale Aussagenlogik (LTL)

- atomare Ausdrücke
  - Signale des Designs
  - Boole'sche Funktionen, z.B. (a<b), OneHot(a,b,c)
- Boole'sche Operatoren
  - z.B. $\neg\varphi$, $\varphi\wedge\psi$ , $\varphi\vee\psi$ , ...
- temporale Operatoren $m \in \mathbb{N}$ , $n \in \mathbb{N} \cup \{\infty\}$

  $X_{[m]}\ \varphi$ in genau m Zeitschritten gilt $\varphi$

  $F_{[m,n]}\ \varphi$ im Intervall [m,n] gilt $\varphi$ mindestens einmal

  $G_{[m,n]}\ \varphi$ im Intervall [m,n] gilt immer $\varphi$

  $\varphi\ U_{[m,n]}\ \psi$ $\psi$ wird im Intervall [m,n] wahr und bis dahin gilt immer $\varphi$

## Temporale Aussagenlogik (LTL)

Typische Eigenschaften

- Sicherheit:
  a und b werden nie gleichzeitig wahr
  $G \neg(a \wedge b)$
- Lebendigkeit
  jede Anforderung wird innerhalb von 5 bis 7 bestätigt
  $G\,(req \rightarrow F_{[5,7]} ack)$
- Fairness:
  Innerhalb von 30 Schritten gilt mindestens einmal a
  $G\,F_{[30]} a$



## Semantik von LTL

Definiert über unendlichen Signalfolgen

$Sig = \{ a, b, c, ... \}$ Menge von Bezeichnern für Signale

Signalfolge $\pi$ weist jedem Signal $a \in Sig$ zu jedem Zeitpunkt $t \in N$ einen Boole'schen Wert zu

$\pi: Sig \times N \rightarrow B$

Der Suffix einer Signalfolge ab der Zeit $t$ ist definiert durch

$\pi^t(a,z) = \pi(a,z+t)$

## Signalfolgen

## Semantik von LTL

ist definiert durch die Modellrelation $\models$

$\pi \models a \quad \Leftrightarrow \quad \pi(a,0)=true$

$\pi \models \neg\varphi \quad \Leftrightarrow \quad \pi \not\models \varphi$

$\pi \models \varphi \wedge \psi \quad \Leftrightarrow \quad \pi \models \varphi$ und $\pi \models \psi$

$\pi \models X_{[m]}\, \varphi \Leftrightarrow \quad \pi^m \models \varphi$

$\pi \models F_{[m,n]}\, \varphi \Leftrightarrow \quad \exists\, m \le t \le n\, .\, \pi^t \models \varphi$

$\pi \models G_{[m,n]}\, \varphi \Leftrightarrow \quad \forall\, m \le t \le n\, .\, \pi^t \models \varphi$

$\pi \models \varphi\, U_{[m,n]}\, \psi \Leftrightarrow \quad \exists\, m \le t \le n\, .\, \pi^t \models \psi$ und
$\forall\, k < t\, .\, \pi^k \models \varphi$

## Signalabfolgen und Schaltung

- Die Schaltungsbeschreibung enthält freie Eingäng
  - ➔ beliebig viele Signalabfolgen können aus der Schaltung entstehen
  - ➔ alle Signalabfolgen müssen die LTL-Formel erfüllen
  - ➔ Beweis nicht über den Einzelnen Signalabfolgen, sondern direkt auf dem formalen Modell (endlichen Automaten)



## Theorembeweisen

- Modellierung der Schaltung in Logik Höherer Ordnung
- Modellierung der Eigenschaften in Logik höherer Ordnung
- interaktiver Beweis durch Anwendung von Theoremen
- oft kombiniert mit
  - Heuristiken zur Automation des Beweises
  - automatischen Entscheidungsprozeduren

## Timinganalyse

- Wie schnell Propagieren sich Signalwechsel zwischen den Eingängen, den Ausgängen und den Registern
  - ➔ Bestimmung der Taktrate



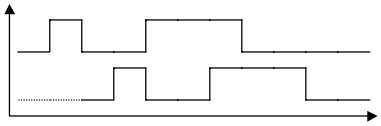
## Zeitmodelle für Gatter

- Totzeit
  - das Eingangssignal erscheint zeitverschoben am Ausgang
- Totzeiten für steigende und fallende Taktflanken
  - steigende und fallende Taktflanken werden unterschiedlich verzögert
- träge Totzeit
  - kurze Impulse werden nicht an den Ausgang propagiert

## Zeitmodelle für Gatter

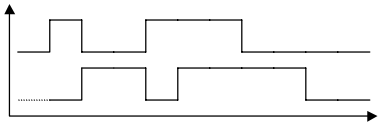
- Totzeit

$\delta_t=2$



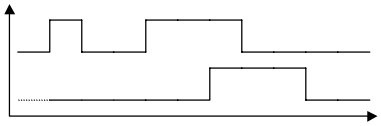
- Totzeiten für steigende und fallende Taktflanken

$\delta_\uparrow=1, \delta_\downarrow=2$



- träge Totzeit

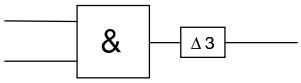
$\delta_{min}=2, \delta_t=2$



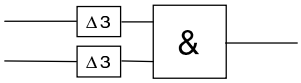


## Einfaches Totzeitmodell

- Jedes Gatter arbeitet ohne Verzögerung
- Jedem Gatter ist ein externes Verzögerungsglied zugeordnet




äquivalent zu

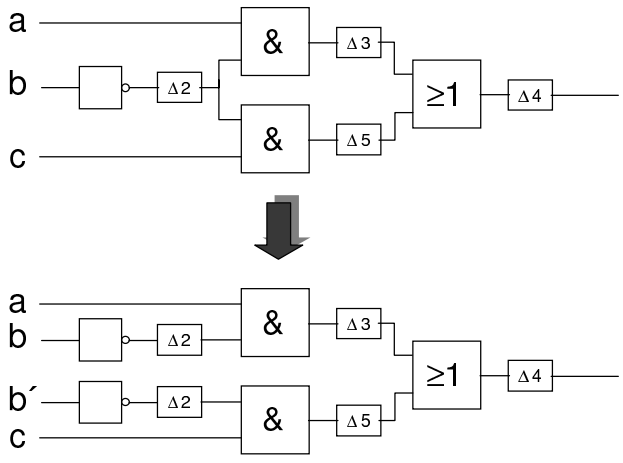


## Timinganalyse

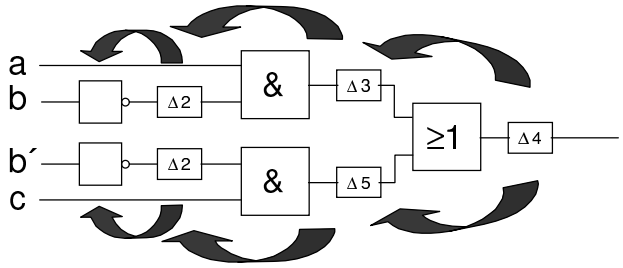
- duplizieren von mehrfach benutzen Schaltungsteilen und Eingängen
- verschieben der Verzögerungszeiten zu den Eingängen der Gatter
- addieren von Verzögerungszeiten auf einer Leitung

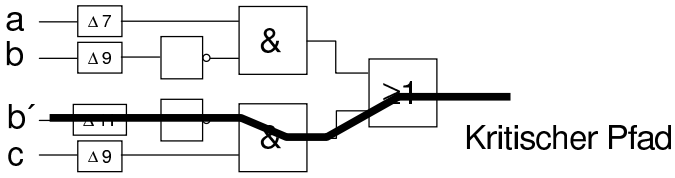


## Duplizieren von Schaltungsteilen



## Addieren der Verzögerungszeiten






## Sensibilisierbarkeit

Ist der längste Pfad auch sensibilisierbar?

D.h. gibt es ein Eingangssignalwechsel, der sich über den längsten Pfad zum Ausgang durchsetzt?

→D-Kalkül