Vorlesungsmitschrieb Rechnerarchitektur
SS 2005, Matthias Bernauer

Rechner

Ein universell (frei programmierbar) einsetzbares Gerät zur automatischen Datenverarbeitung [>>Duden]

-Architektur

extern: Systembestandteile aus Programmierersicht (Befehlssätze),
z.B. Multiplikation intern anders realisiert
intern: interner Aufbau (Organisation und Verbindungen) des Rechners,
z.B. Pipelining extern nicht sichtbar

Hardwareentwurf

1. Wie entwirft man Rechner?

- Einsatzgebiete von Prozessoren
  Früher nur in Rechneranlagen, heute in Haushaltsgeräten (Beleuchtung, Alarm), Telekomunikation (Handy --> geringe Größe & Leistungsverbrauch wichtig), Fahrzeugen (ABS, EPS, Navigation, Airbag, Motorsteuerung), Luft-&Raumfahrt, Medizin, u.v.m.
  --> Cisc-Rechner mit komplexen Befehlen (geringer Speicherbedarf für Programme) wieder interes.
  DSP: Erweiterung zur digitalen Signalverarbeitung, z.B. Für mpeg, jpg

- Rückschläge in der Computerentwicklung: Pathfinder, Challenger Unglück, Pentium Bug
- Neue Aufgabengebiete: Multimedia, Netzwerke, Internet, digitale Signalverarbeitung
- Optimierung des Leistungsverbrauchs:
  1. Logiksynthese: Verfahren zur Reduktion von Schaltvorgängen im Gatternetz
  2. Reduktion der Spannung und Taktrate
  3. Stepspeed
  4. Taktabschaltung bei momentan nicht benötigten Teilen

- Komplexitätszuwachs:
  Verdopplung der Transistor-Dichte alle 18Monate nach Moore's Law
  Neue Architekturkonzepte oftmals erst durch wachsende Anzahl Transistoren je Chip möglich

- Hardwarebeschreibung und -entwurf
  Spezifikation in leb. Sprache -> formelle Spezifikation -> Simulation und Tests -> HighLevelSynthese -> Logiksynthese -> Testvorbereitung (Scanpath) -> Plazierung&Verdrahtung (Place&Route) -> Layout -> Fertigung -> Tests. ständig: formale Verifikation

1. **Entwurfsautomatisierung** (eda – electronic design automation)

   1. Hardwareentwurfsschritte

      - Klassifikation:
        Partitionierung der Systemebene in:

        - Softwarebereich
          auf Mikrocontrollern, digitalen Signalprozessoren, Mikroprozessoren
        - Hardwarebereich
          ASIC – application specific integrated circuit
          FPGA – field programmable gate array (Hardware vom Anwender programmierbar)

- heterogene System
  - embedded systems: Umgebendes System (Mechanik, Hydraulik, Elektronik, analoge HW) -> A/D-Wandler, Sensoren -> Software/Digitale Hardware -> D/A-Wandler, Aktuatoren -> analoge HW
  - system-on-chip: DSP, Mikrocontroller, Speicher, ASIC

- Entwurfsweisen
  top-down: Systemarchitektur -> HardwareRealisierung
  abstrakte Beschreibung des Systems wird sukkzessive bis zur HW verfeinert
  bottom-up: HardwareRealisierung -> Systemarchitektur
  bereits entworfene Komponenten werden zu komplexeren Einheiten zusammengefügt

- **Abstraktionsebenen**
  Warum verschiedene Ebenen und nicht alles auf unterster Ebene?

  - Design-Gap = [ability to design < available silicon] => daher Entwurfsautomatisierung
    Die Größe von Systemen (HW/SW) übersteigt die Fähigkeiten im Entwurf auf niedrigen Ebenen

  - Time-to-Market
    Neue Produkte müssen in immer kürzeren „Zeitfenstern“ auf den Markt (z.B. Handys)
    - Entwicklungskosten 50% zu hoch --> 5% Gewinnminderung
    - Produktionskosten 80% zu hoch --> 20% Gewinnminderung
    - 6Monate zu spät auf den Markt --> 40% Gewinnminderung

  - ability to verify < ability to design < available silicon ~ #Transistoren/Chip

  - Verifikation-Gap = [ability to verify < available silicon]
    Die Fähigkeit ein großes System zu entwickeln steigt nicht in dem Maße an, wie der Bedarf nach neuen Entwicklungen. Die Fähigkeit zu verifizieren steigt noch langsamer, der ist Verifikationgap also noch größer als der Designgap

- Entwurf großer Systeme
  - Partitionierung in HW- und SW-Blöcke
  - *Frühzeitige* Exploration des Entwurfsraumes durch Kostenfunktionen
  - *automatische* Übersetzung in niedrigere Abstraktionsebenen (HW-/SW-Synthese)
  - Einsatz von *Standardkomponenten* für HW als auch SW
    IP-Cores = vorhandene Entwürfe von Fremdanbietern

  - hirarchisches Arbeiten

- Verringerung von Entwurfszeit und -kosten

  - Finden von Spezifikationsfehlern in frühen Phasen (Vermeidung von turn-arounds) ausführbare Spezifikationen + formale Verifikation
  - rapid prototyping
  - Frühzeitige Abschätzung von kritischen Designparametern wie z.B. Durchsatz, Leistungsaufnahme, Antwortzeiten, Design- und Produktkosten

- AbstraktionsEbenen im Überblick
  - Systemebene — funktionelle Einheiten, die miteinander kommunizieren
  - algorithm. Ebene — Spezifikation der Funktionen einzelner Blöcke mittels Algorithmen in einer HW-Beschreibungssprache
  - RegisterTransfer — Darstellung funktionaler Einheiten durch Datenpfad- und Kontrollpfad (Daten werden von Register zu Register transferiert und dabei ver.)
  - Gatterebene — Es gibt nur noch Boole‘sche Signale, Boole‘sche Gatter & einfache FF
  - Transistorebene — Realisierung Boolescher Elemente durch Transistoren

- Layoutebene — Realisierung von Transistoren durch dotierte Bereiche und isolierende Schichten auf dem IC

Y-Diagramm	Struktur	Verhalten	Geometrie
Systemebene	Prozessoren, Speicher	abstr. Beschreibung	System-Floorplan
algorithm. Ebene	Funktionen, Prozeduren	Algorithmen	
RegisterTransferEb.	ALUs, Multiplexer, Register	endl. Automaten	Modul-Floorplan
Gatterebene	Gatter, FlipFlops	boolsche Gleichungen	Gatter-Floorplan
Transistorebene	Transistoren, Drähte, Kontakte	Spannung, Ströme	Transistornetzliste
Layoutebene	Flächen, Schichten	Flussdichte	Layoutmaske

- Was wird im Y-Diagramm dargestellt?
  Verhalten, Struktur und Geometrie der Abstraktionsebenen

- Systementwurf - Ziel des Entwurfsprozesses
  - Beschreibungen einer hohen Abstraktionsebene in eine niedere Ebene zu transformieren
  - durch Verfeinerung und unter Beibehaltung der Funktionalität (eventuell auch des Zeitverhaltens)
  - Verifikation/Validation der Beschreibung auf den unterschiedlichen Ebenen

- Automatisierung
  - mittels Grafikeditoren
  - Simulation
    - Testbenchgeneratoren um Eingangsdaten für Simulationen zu erzeugen
    - Coverage-Analysatoren
  - Verfifikation
    - linting, z.B. Mindestabstände zwischen Leitungen
    - Äquivalenzprüfung
  - Verhaltens- und Logiksynthese
  - Timinganalyse
  - HW-Test: Automatic Testpattern Generator, Fehlersimulation, Build-in-tests

2. Hardwarebeschreibungssprachen (HDL: hardware description language)
   - präzise Spezifikation
   - Spezifikation ausführbar --> Simulation möglich
   - Automatisierung
   - Austauschformat
   - Dokumentation
   - Beschleunigung

   Was benötigt man zur Schaltkreisbeschreibung?
   Was unterscheidet HDLs von Software-Sprachen?

   - **Schaltungsbeschreibung**
     Schaltungen bestehen aus Komponenten und Verbindungen

     -> **strukturelle Beschreibung**smögl. notwendig
     Grafiken auf Transistor- oder Gatterebene werden zu komplex für große Schaltungen/Entwürfe

     -> Hirarchiebildung:
     Zusammenfassung von Teilen zu neuen Komponenten, z.B. Gatter, ALU, ...

-> Hierarchische Beschreibung von Systemen mit **entities** und **architectures**
z.B. Entity-Deklaration des Volladdierers

```
entity full_adder is
   port(a, b, carry_in: in Bit;  --input ports
  sum,carry_out: out Bit);     --output ports
end full_adder;
```

Entities definieren Interface einer Teilschaltung
Ports können sein: in (input), out (output), inout (bidirectional)
Outputs können nicht gelesen werden, Inputs nicht beschrieben.

```
architecture structure of full_adder
is component half_adder
  port (in1,in2:in Bit; carry:out Bit; sum:out Bit);
end component;
component or_gate
  port (in1, in2:in Bit; o:out Bit);
end component;
signal x, y, z: Bit;              -- local signals
begin                             -- port map section
    i1: half_adder port map (a, b, x, y);
    i2: half_adder port map (y, carry_in, z, sum);
    i3: or_gate port map (x, z, carry_out);
end structure;
```

Architectures beschreiben Implementierungen von Entities.
Für component half_adder brauchen wir

- eine entity, z.B. **entity** half_adder
  port (in1,in2:**in** Bit; carry:**out** Bit; sum:**out** Bit); **end** half_adder;

- Mindestens eine architecture, die ihrerseits components enthalten kann

Architectures und ihre components können eine Hierarchie beliebiger Tiefe definieren. Pro Entity kann es mehrere architectures geben. Standardmäßig wird die zuletzt analysierte Architecture benutzt. Die Benutzung einer anderen Architecture kann in einer **configuration** vorgeschrieben werden. Architecture-Beschreibungen können Strukturbeschreibungen (wie oben angegeben) oder Verhaltensbeschreibungen sein.

Beispiel einer Architecture mit Verhaltensbeschreibung am Volladdierer:

```
architecture behavior of full_adder is
  begin
    sum <= (a xor b) xor carry_in after 10 Ns;
    carry_out <= (a and b) or (a and carry_in) or (b and carry_in) after 10 Ns;
  end behavior;
```

Paralelle Signalzuweisung mittels <= , Spezifikation von Verzögerungszeiten, vorhandene Operatoren

- Struktur- und Verhaltensbeschreibungen

  - Strukturbeschreibungen benutzen Komponenteninstantiierungen.
  - Verhaltensbeschreibungen spezifizieren das Verhalten ohne die Systemstruktur vorzuschreiben.
  - Mischformen sind möglich.
  - Mischformen sind **nötig** zur Verhaltensbeschr. der Blätter einer strukturellen Hierarchie
  - Strukturelle Hierarchie essentiell für eine kompakte und klare Modellierung großer Hardwaresysteme.

- **Algor. Beschreibung** notwendig, da strukturelle Beschreibungen (20Mio. Gatter) zu komplex. Das Verhalten der Module wird durch imperative Programmiersprachen (Teil der HDL) definiert.

- Besonderheiten von Hardware(-sprachen)
  Zeitbegriff: Funktionen verbrauchen Zeit
  parallele Tasks: Funktionen können parallel arbeiten
  Signale und Ereignisse (events): Kommunikation zwischen Modulen
  mehrwertige Logik (0,1,x,z,...): zweiwertige Logik nicht ausreichend

- VHSIC = very high speed integrated circuit
  VHDL = VHSIC hardware description language
  Erweiterung: VHDL-AMS, enthält auch Modellierung analoger Schaltungen

- Ziele von VHDL
  - Modellierung digitaler Schaltungen
  - Modellierung auf verschiedenen Abstraktionsebenen
  - Technologieunabhängig⇒ Wiederverwendbarkeit (abstrakter) Spezifikationen
  - Standard ⇒ Portabilität (verschiedene Synthese- und Analysetools möglich)
  - Validierung von Designs für verschiedene Abstraktionsebenen
  - Enthält viele Aspekte imperativer Programmiersprachen
    (⇒ wäre prinzipiell auch in der Lage, Software zu beschreiben.)

3. HW-Simulation und Verifikation

- Spezifikation -> Entwurf -> unvollst. Simulation -> Auswertung und ggf. Neuer Entwurf
- Simulationstechniken
  Aufgabe von Simulationsalgorithmen: Abbilden von HW-Funktionalität (Hardwarebeschreibung z.B. in Verilog) auf eine Zielarchitektur (PC oder Workstation, z.B. von Neumann Rechner)
  Problem: hoher Grad an Parallelität muss auf eine sequentielle Maschine „projiziert“ werden

- Arten der HW-Simulation

  Analogsimulation (Spice, ...) Ströme, Spannungen

  - werte- und zeitkontinuierlich

  - nichtlineare Differentialgleichungen

  Digitalsimulation (VSS, Modelsim, ...) nur 0 und 1

  - werte- und zeitdiskret

  - boolesche Algebra oder auf höheren Abstraktionsebene, z.B. Mit Fließkommazahlen

  Mixed-Mode/Simulatorkopplung (Saber,...)

  - analog/digital-Simulation

- Logiksimulationsalgorithmen:
  - Streamline Code Simulation (auch 'Compiled Mode'): Schaltkreise, vollsynchrone Schaltwerke, es wird direkt ausführbarer Code auf der Zielplattform generiert.

    Einschränkungen:

    - kombinatorische oder strikt synchrone Schaltwerke

    - keine Zeitinformationen

    - Simulation bei synchronen Schaltwerken nur „zyklengenau“

    Kombinatorischer Schaltkreis wird als gerichteter azyklischer Graph notiert:

    => Topologische Sortierung: Knoten des azyklischen Graphes werden halbgeordnet (levelizing)

```
int a, b, c, d, e;
int x1, x2, x3;
int o1, o2;
x1 = b & c;   //lev1
x2 = d & e;   //lev1
x3 = x1 | x2; //lev2
o1 = a & x3;  //lev3
o2 = x2 | x3; //lev3
```

Eigenschaften des Graphen:
- Knoten auf einer höheren Ebene können Knoten niedrigerer Ebenen nicht beeinflussen
- Knoten auf derselben Ebene beeinflussen sich gegenseitig nicht
- Knoten auf niedrigeren Ebenen beeinflussen Knoten höherer Ebenen

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

Da int aus z.B. 32 Bits besteht, können mit dieser Technik 32 Simulationen gleichzeitig durchgeführt werden -> wortparallele Simulation mit Beschleunigungsfaktor 32. 32 mögl. Eingangsvektoren:

a = (a0, a1, ..., a31)
b = (b0, b1, ..., b31) => o1 = (o1_0, o1_1, ..., o1_31)
... o2 = (o2_0, o2_1, ..., o2_31)
e = (e0, e1, ..., e31)

Vorteile:
- es wird auf der Ziel-Maschine comilierter Code erzeugt, d.h. Schnelle Abarbeitung
- wortparalleles Arbeiten

Betrachte StreamlineCodeSimulation ohne paralelles Arbeiten:
- das Verfahren ist linear in der Anzahl der Gatter!
- Verbesserung durch die EventDrivenSimulation

- Event Driven Simulation, Critical Event Scheduling:
  Schaltungen auf unterschiedlichen Abstraktionsebenen und mit Zeitinformation
  Die Idee ist es, algorithmische Blöcke durch Threads zu verwalten. Nur die Systemteile werden neu berechnet, deren Eingaben sich verändert haben (anstehende Ereignisse=Events).
  globale Datenstruktur: EventQueue, Scheduler übernimmt die Verwaltung
  - Hier: Spezialfall einer Event Driven Simulation für Schaltkreise
  - Ereignisse/Events sind Änderungen von Signalwerten zu bestimmten Zeitpunkten

- Ein Event in der Form (s, v, t), wobei (Signal, Signalwert ∈ {0, 1}, Zeitpunkt ∈ N)

  Die Event-Queue speichert eine nach Zeitpunkten sortierte Liste der anstehenden Ereignisse. Beispiel einer Event-Queue:
- Um bei Eintreten eines Events (s, v, t) die Simulation weitertreiben zu können, muss man wissen, welche Gatter gerade auf das Event warten.
- Das sind alle Gatter, die s als Eingang haben. Diese Gatter warten sensitiv auf Events auf s.
- Zu jedem Signal s gibt es also eine Liste waiting(s) aller Gatter, die s als Eingang haben.
- Ein Event auf s löst evtl. Events auf den Ausgangssignalen der Gatter aus waiting(s) aus.

```
While (time <= endtime AND Eventqueue != empty) {
   time = min({timestamps in Eventqueue})
   while ( Eventqueue != empty AND (s,v,time) = e aus Eventqueue) {
      entferne e aus Eventqueue
      führe e aus, d.h. Weise Wert v an Signal s zu
      if (dadurch Signalwert geändert) {
            trage alle Gatter aus waiting(s) in Gatteraktivierungsliste ein}
   }
   while (Gatteraktivierungsliste != empty) {
      entferne erstes Element g aus Gatteraktivierungsliste
      führe Gatterberechung durch (berechne v(g), Ausgangssignal s(g) mit Verzögerung d(g)
      füge (s(g), v(g), time + d(g)) in Eventqueue ein
   }
}
```

Beobachtungen:

- Event Driven Simulation hat den Vorteil, dass unnötige Gatterauswertungen vermieden werden (Beispiel 2).
- Gatter werden aber evtl. auch mehrmals ausgewertet (Beispiel 1: g5).
- Das ist interessant bei der Analyse von Spikes (siehe Übung).
- Evtl. uninteressant bei Simulation für synchrone Schaltwerke. In dem Fall interessiert man sich nur für die resultierende stabile Belegung.⇒ hier evtl. doch Streamline Code Simulation
- Warum Trennung zwischen Abarbeitung der Eventqueue und Gatteraktivierungsliste in jedem Simulationszyklus?
  1. Gatter können in einem Zyklus mehrfach zur Aktivierungsliste hinzugefügt werden
  2. Es wird erzwungen, dass gerade erst ausgelöster events erst im nächsten Zyklus abgearbeitet werden, also in topologischer Reihenfolge (auch bei Delay 0)
- Event Driven Simulation kann auch mit Gatterverzögerung 0 durchgeführt werden.
- In aufeinanderfolgende Simulationszyklen können dann Events mit gleichen Zeitpunkten abgearbeitet werden.
- Hierbei ergeben sich zwei Zeitachsen: Simulationszeit, Deltazeit (delta delay)

Events der gleichen Simulationszeit, die in unmittelbar aufeinanderfolgenden Simulationzyklen abgearbeitet werden, sind „durch ein delta delay voneinander getrennt“.

- Event Driven Simulation lässt sich verallgemeinern von Schaltkreissimulation auf allg. Szenarien, z.B. Simulation von Hardwarebeschreibungssprachen wie VHDL mit

- Formale Verifikation

  - Äquivalenzprüfung: sind zwei Schaltungen äquivalent
    Was ist Äquivalenzprüfung („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?

    Äquivalenzprüfung von Schaltkreisen: Problemverkleinerung durch Zusammenfassung von strukturell gleichen Teilen: Schaltung 1 + 2

    Vorgehen:
    - Transformation der Schaltkreise in eine Normalformdarstellung (z.B. KNF, BDDs,...)
    - Liegt Äquivalenz der Normalformen vor? z.B. Sind BDDs gleich?
    >> Alternative: Erzeuge miter-Schaltkreis (gelber Bereich)

    Liefert der miter für alle Eingangsbelegungen 0, sind die Schaltungen äquivalent

    -> es wurde eine Übersetzung in ein Erfüllbarkeitsproblem vorgenommen.
    Optimierungen: BDDs, SAT-Solver, Ausnutzen struktureller Ähnlichkeiten, inkrementelle Verfahr.
    Sat-Solver: Algorithmus zur Lösung von Erfüllbarkeitsproblemen (in KNF gegeben)

    Äquivalenzprüfung von Schaltwerken:
    Was ist, wenn man keine gleiche Zustandskodierung oder gemeinsamen Startzustand hat?
    ⇒ Einsatz von Techniken und Methoden der Automatentheorie
    - Produktautomaten bilden mit miter-Ausgang
    - Durchsuchen des Zustandsraumes nach erreichbaren Zuständen, die „Unterschied“ am Ausgang erzeugen
    -Symbolische Methoden mit BDDs

    Def.: $M = (I, O, S, s_0, \delta, \lambda)$ ist Mealy-Automat gdw.:
    - I endlich, nichtleere Menge von Eingabesymbolen ist
    - O endlich, nichtleere Menge von Ausgabesymbolen ist
    - S endlich, nichtleere Menge von Zuständen ist
    - s_0 aus S Anfangszustand ist
    - δ: SxI -> S Übergangsfunktion ist
    - λ: SxI -> O Ausgabefunktion ist

    Berechnung eines Mealy-Automaten:
    Sei $w=(w^0,w^1,...)$ eine (un)endl. Folge von Eingabesymbolen
    $M = (I, O, S, s_0, \delta, \lambda)$ berechnet eine (un)endl. Folge von Ausgabesymbolen $u=(u^0,u^1,...)$ zu w gdw. Es eine Folge von Zuständen $s=(s^0,s^1,...)$ gibt, so dass $s^0 = s$; $\delta(s^i,w^i) = s^{i+1}$; $\lambda(s^i,w^i) = u^i$ für alle i >= 0

    Def. Automatenäquivalenz:
    Seien $M^A = (I, O, S^A, s_0^A, \delta^A, \lambda^A)$ und $M^B = (I, O, S^B, s_0^B, \delta^B, \lambda^B)$ Mealy-Automaten.
    M^A und M^B heißen äquivalent gdw. M^A und M^B zu jeder Eingabefolge die identische Ausgabefolge berechnen. Prinzipiell kann die Äqu. der Automaten folgendermaßen entschieden werden:
    - Berechene zu M^A und M^B die minimalen Automaten $M^{A,min}$ und $M^{B,min}$

- M^A und M^B sind äquivalent, wenn $M^{A,min}$ und $M^{B,min}$ bis auf Isomorphie identisch sind

Aber: Heutige Verfahren zur Automatenminimierung arbeiten auf den Zustandsgraph von M^A und M^B. Zustandsgrafen sequentieller Schaltungen können sehr groß werden, z.B. n FlipFlops führen zu 2^n Zuständen. -> Alernativverfahren nutzen, bei denen eine explizite Repräsentation des Zustandsgrafen vermieden werden kann (symbolische Darstellung von Zustandsmengen mit BDDs)

Produktautomaten M^{Prod} von M^A und M^B = (I, {0,1}, S^A x S^B, ($s_0{}^A$,$s_0{}^B$), δ^{Prod}, λ^{Prod}) mit

$\delta^{Prod}(s_A,s_B,w) = (\delta^A(s_A,w), \delta^B(s_B,w))$ für alle s_A aus S_A, s_B aus S_B, w aus I

$\lambda^{Prod}(s_A,s_B,w) = 1$ gdw. $\lambda^A(s_A,w)=\lambda^B(s_B,w)$ für alle s_A aus S_A, s_B aus S_B, w aus I

M^A und M^B sind genau dann nicht äquivalent wenn es eine Eingabefolge $w=(w^0,w^1,..., w^z)$ gibt, so dass

- die zugehörige Ausgabefolgen von M^A und M^B verschieden sind
- der Produktautomat eine Folge != (1,...,1) ausgibt (k Einsen)

D.h. das Äquivalenzproblem von M^A und M^B entspricht einem Erreichbarkeits-problem im Produktautom.

-> gibt es einen Zustand (s^{*A},s^{*B}) im Produktautomaten und ein Eingabesymbol w* aus I, so dass
$\lambda^{Prod}(s^{*A},s^{*B},w^*) = 0$ und (s^{*A},s^{*B}) vom Startzustand ($s_0{}^A$,$s_0{}^B$) aus durch irgendeine Eingabefolgeerreichbar ist?

- Berechene Produktautomten

- Bestimme die Menge aller Zustände
$S^{bad}=\{(s^{*A},s^{*B})|$ es gibt ein w* aus I, s.d. $\lambda^{Prod}(s^{*A},s^{*B},w^*)=0\}$

- Bestimme die Menge aller vom Startzustand ($s_0{}^A$,$s_0{}^B$) aus erreichbaren Zustände $S^{rechable}$ (funktioniert da M^{prod} endlich!)

- Ist S^{bad} geschnitten $S^{reachable}$ leer (d.h. dann auch äquivalent) oder nicht?

! Das Verfahren lässt sich BDD basiert ohne explizites Darstellen von Zustandsmengen durchführen !

- Modellprüfung: erfüllt eine Schaltung bestimmte (sequentielle) Eigenschaften

Was ist eine Modellprüfung? Die Frage, ob ein Schaltwerk eine gegebene Spezifikation erfüllt?

- Modellierung des Schaltwerkes durch endliche Automaten
- Spezifikation der Eigenschaften in temporalen Aussagenlogiken
- Model Checking: Fixpunktiteration, Traversierung des Zustandsraumes, BDDs [vollständig]
- Bounded Model Checking: SAT-Solver zur Lösung
[unvollst.: nur zum finden von Fehlern – kein Korrektheitsbeweis!

>> Temporale Aussagenlogiken (CTL*, CTL, LTL)

Formeln in temporalen Aussagelogiken beschreiben gewünschte Eigenschaften des Designs.

Formeln werden rekursiv aufgebaut auf Basis von

- atomaren Formeln: Signale des Designs, z.B.: y_i bedeutet: Ausgang y_i hat aktuell

den Wert 1

- Booleschen Operatoren, z.B. $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, z.B.: $y_i \wedge y_j$

- temporalen Operatoren: machen Aussagen über einen einzelnen Berechnungs**pfad** des Berechnungsbaumes des zugehörigen Schaltwerkes
  z.B. Gϕ oder always: Formel ϕ gilt in **jedem** Zustand des Pfades.
  z.B. Fϕ oder sometimes: Formel ϕ gilt in **einem** Zustand des Pfades.
  z.B. Xϕ oder next: Formel ϕ gilt für den Pfad beginnend mit dem nächsten, d.h. zweiten Zustand des Pfades
  z.B. ϕUψ oder until: Formel ϕ solange bis Formel ψ gilt

  Diese vier Operatoren machen Aussagen über Berechnungspfade

Aussagenlogiken mit temporalen Operatoren
- machen Aussagen über Eigenschaften von **Zuständen** auf der Basis von Pfadformeln ϕ:
- Aϕ oder for all paths: Formel ϕ gilt auf **allen möglichen** Berechnungspfaden, die in dem Zustand beginnen.
- Eϕ oder there exists a path: Formel ϕ gilt auf **einem** Berechnungspfad, der in dem Zustand beginnt.

Jede Zustandsformel kann im folgenden Sinne als Pfadformel interpretiert werden: Wenn die Zustandsformel ϕ in einem Zustand gilt, dann gilt sie für jeden Pfad, in dessen Anfangszustand die Zustandsfolgel ϕ gilt

- Theorembeweisen: interaktive Methode, basierend auf Logik höherer Ordnung (Logik höherer Ordnung unentscheidbar)
  - Modellierung der Schaltung/Eigenschaften in Logik höherer Ordnung
  - interaktiver Beweis durch Anwendung von Theoremen
  - oft kombiniert mit Heuristiken zur Automation des Beweises oder automatischen Entscheidungsprozeduren