Formale Verifikation von Hardware

Wolfgang Günther
Infineon AG
CL DAT TDM VM

guenther@informatik.uni-freiburg.de,
wolfgang.guenther@infineon.com

Inhalt der Vorlesung

- Der Entwurfsprozess
  - Verifikation
- Binary Decision Diagrams
  - Datenstruktur und Algorithmen
- SAT-Solver
- Kombinatorische Verifikation
- Sequentielle Verifikation
  - Algorithmen
  - CTL

Vorbemerkungen

- Zielrichtung der Vorlesung:
  - Breites Wissen im Gebiet der formalen Verifikation
  - neuere Entwicklungen bis (fast) zum aktuellen Stand der Forschung
- Übungen
- Scheinvergabe

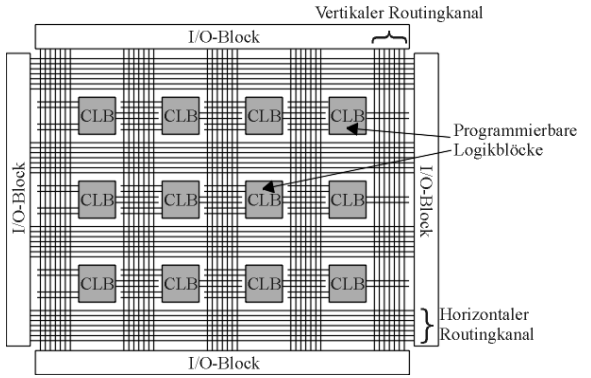
Literatur

- G. Hachtel und F. Somenzi:
  Logic Synthesis and Verification Algorithms.
  Kluwer Academic Publishers, 1996.
- R. Drechsler und B. Becker:
  Graphenbasierte Funktionsdarstellung.
  B.G. Teubner, 1998.
- T. Kropf: Introduction to Formal Hardware Verification. Springer-Verlag, 1999.
- K.L. McMillan: Symbolic Model Checking.
  Kluwer Academic Publishers, 1993.

Klassifikation von integrierten Schaltungen

- Mikroprozessoren
  - universell einsetzbar, durch Software programmierbar
  - Produktion in großen Stückzahlen
- ASICs = Application Specific Integrated Circuits
  - anwendungsspezifische integrierte Schaltungen
  - ganz spezielle Anwendungen, kleine Stückzahlen
- FPGAs = Field-Programmable Gate Arrays
  - Bestehen aus (z.T. programmierbaren) Logikblöcken und programmierbaren Verbindungen
  - Programmierung der Hardware beim Anwender

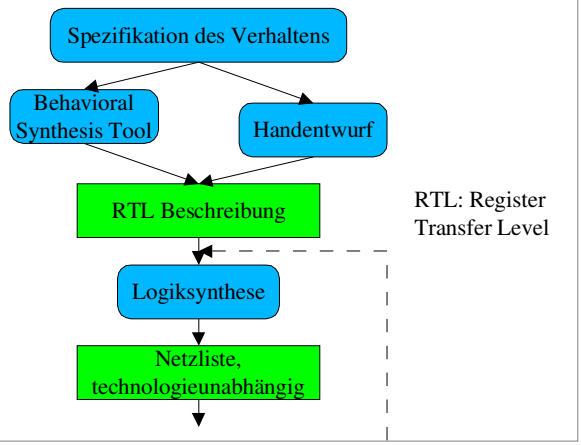
Field Programmable Gate Arrays



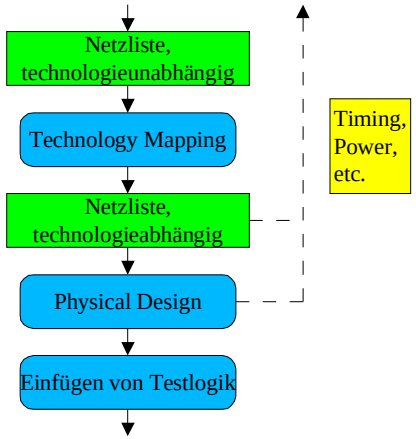
Kostenfaktoren

- Entwurfskosten
  - Kosten, um ausgehend von einer Spezifikation die Fertigungsdaten für die Schaltung zu bekommen
  - Besonders kritisch bei kleinen Stückzahlen
  - Senkung der Kosten: Automatisierung
- Fertigungskosten
  - Kosten, die bei der Fertigung gemäß der Fertigungsdaten entstehen (Maskenherstellungskosten, etc.)
  - Auch beeinflusst durch Ausbeute (Prozentsatz der funktionsfähigen Chips eines Fertigungsprozesses)

Der Entwurfsprozess



Der Entwurfsprozess (2)



Software vs. Hardware

Software

- Sequentielle Ausführung
- Prozeduraufrufe
- Parameterübergabe
- Datentypen

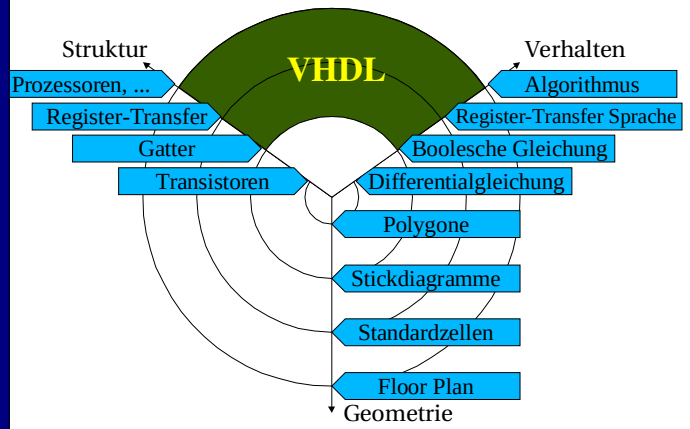
Hardware

- Parallele Ausführung
- Instantiierung von Komponenten
- Verdrahtung
- Datentypen in Beschreibung, nicht in Implementierung

VHDL

- Very High Speed Integrated Circuits Hardware Description Language
- Beschreibung komplexer elektronischer Systeme
  - Austauschbarkeit
  - Wiederverwendbarkeit
- Industriestandard
- Beschreibung auf verschiedenen Abstraktionsebenen

Abstraktionsebenen



VHDL: Wichtige Eigenschaften

- Modellierung auf Verhalten- und auf Strukturebene
- Modellierung von Gleichzeitigkeit
  - Hardware ist inhärent parallel
- Abstrakte Datentypen
- Hochsprachenkonstrukte
- Strenge Typisierung
- Diskrete Event-Simulation

Entity

- Definiert die Schnittstelle nach außen
- Pins können sein
  - in -- Eingang
  - out -- Ausgang
  - inout -- Bidirektional
  - ... und andere
- Ausgänge können nicht gelesen werden, Eingänge können nicht geschrieben werden

Architecture

- beschreibt die Funktion des Bausteins mit Hilfe von
  - Strukturelementen
  - Verhaltensbeschreibung
  - Mischformen sind möglich
- Es kann mehrere Architectures für eine Entity geben

Beispiel: AND-Gatter

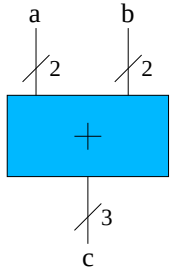
```
entity my_and is
   port(a, b : in  bit;
        c    : out bit);
end my_and;

architecture rtl of my_and is
begin
   c <= a and b;
end rtl;
```

2-Bit-Addierer: Verhalten

```
entity add2 is
   port(a, b : in  integer range 0 to 3;
        c    : out integer range 0 to 6);
end add2;

architecture behave of add2 is
begin
   c <= a + b;
end behave;
```



2-Bit-Addierer: Register-Transfer-Level

```
entity add2 is
   port(a, b : in  bit_vector(1 downto 0);
        c    : out bit_vector(2 downto 0));
end add2;

architecture rtl of add2 is
begin
   c(0) <= a(0) xor b(0);
   c(1) <= a(1) xor b(1) xor (a(0) and b(0));
   c(2) <= (a(1) and b(1)) or
           (a(0) and b(0) and (a(1) or b(1)));
end rtl;
```


2-Bit-Addierer: Gatterebene

```
entity add2 is
   port(a, b : in  bit_vector(1 downto 0);
        c    : out bit_vector(2 downto 0));
end add2;

architecture net of add2 is
signal s1, s2, ... : bit;
begin
xor1 : xor_gate port map(a => a(0), b => b(0),
                         c => c(0));
xor2 : xor_gate port map(a => a(1), b => b(1),
                         c => s1);
and1 : and_gate port map(a => a(0), b => b(0),
                         c => s2);
xor3 : xor_gate port map(a => s1,   b => s2,
                         c => c(1));
...
end net;
```


Sequentielle Anweisungen

- In VHDL gibt es die Möglichkeit, Anweisungen parallel oder sequentiell auszuführen (*concurrent statements, sequential statements*)

```
architecture rtl1
          of mux is
begin
   z <= (s and a)
        or
    (not s and b);
end rtl1;
```

```
architecture rtl2
          of mux is
begin
p : process(a, b, s)
  begin
   if (s = '1') then
        z <= a;
   else z <= b;
   end if;
end process p;
end rtl2;
```

Sequential Statements

- Werden in Prozessen und Sub-Programmen (Funktionen, Prozeduren) verwendet
- Dienen oft zur Beschreibung von Kontrollstrukturen
- Es gibt Konstrukte ähnlich wie in Programmiersprachen
- Die Beschreibung muss nicht notwendigerweise synthetisierbar sein

Flipflop

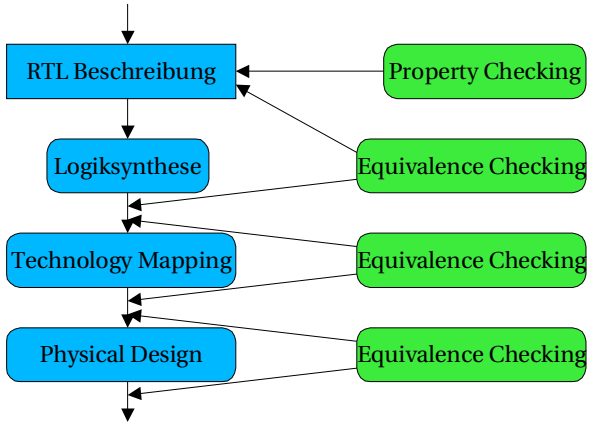
```
entity dff is
   port(clk, d : in  std_logic;
        q      : out std_logic);
end dff;
architecture rtl of dff is
begin
   p : process(clk)
   begin
      if clk'event and clk = '1' then
         q <= d;
      end if;
   end process;
end rtl;
```

Synthese darf Sensitivitätsliste vervollständigen

Flipflop mit asynchronem Reset

```
architecture rtl of dff_ar is
begin
   p : process(clk, reset)
   begin
      if reset = '1' then
         q <= '0';
      elsif clk'event and clk = '1' then
         q <= d;
      end if;
   end process;
end rtl;
```


Formale Verifikation

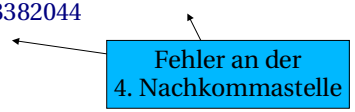


Wozu Verifikation?

- Korrekte, zuverlässige Hardware
  - Einsatz in sicherheitskritischen Anwendungen
- Frühzeitige Erkennung von Entwurfsfehlern
  - doppelte Fertigungskosten
  - Time-to-market
  - Kosten durch Austauschaktionen
  - Schaden für das Image des Unternehmens

Beispiel: Pentium-Bug

- 3. 11. 1994: Erste Berichte in der Intel Newsgroup, dass der Pentium-Prozessor einen Fehler in der Gleitkomma-Division hat
  - 4195835 / 3145727 ergab 1,33373906 und nicht 1,33382044



Beispiel: Pentium-Bug (2)

- 30. 11. 1994: Bericht von Intel
  - Beschreibung Divisionsalgorithmus und Fehler
  - "bei einem normalem Benutzer tritt der Fehler alle 27,000 Jahre auf"
  - kostenloser Austausch "in begründeten Fällen"
- 12. 12. 1994: IBM-Bericht: Annahme über die Gleichverteilung der Inputzahlen nicht legitim
  - → Fehler tritt alle 24 Tage auf

Beispiel: Pentium-Bug (3)

- 20. 12. 1994: Austausch des Prozessors auch ohne Begründung
  - 475 Mio. US$ zusätzliche Kosten im 4. Quartal 1994
  - Imageschaden für Intel
- Fehleranalyse ergab:
  - Fehler in einer Look-Up-Table (ROM) zur Bestimmung der korrekten Quotientenbits
  - intensive Verifikationsbemühungen hätten den Fehler aufgedeckt

Ursachen von Fehlern

- Stark wachsende Komplexität der Schaltungen
  - zunehmende Miniaturisierung
  - neue Techniken zur Entwurfsautomatisierung
  - Software zur Entwurfsautomatisierung wird immer komplexer
- Entwurf entspricht nicht der Spezifikation
- Spezifikation ist unvollständig oder fehlerhaft

Verifikation vs. Testen

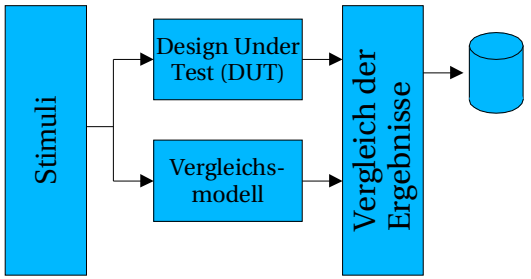
- Verifikation: Fehler im Entwurf
  - Bei einem Fehler sind *alle* gefertigten Chips fehlerhaft
  - Thema dieser Vorlesung
- Testen: Fabrikationsfehler
  - Defekte, die bei der physikalischen Fertigung entstehen (Staubkörner, schlechte Dotierung, etc.)
- Betriebsfehler: Fehler treten erst im Betrieb auf
  - durch Alterung, Verschleiß, etc.

Formen der Verifikation

- Black-Box Verifikation
  - Implementierung ist versteckt
  - Verifikation ist implementierungsunabhängig
- White-Box Verifikation
  - Implementierung ist offen zugänglich
- Grey-Box Verification
  - Implementierung ist versteckt, aber bekannt

Verifikation durch Simulation

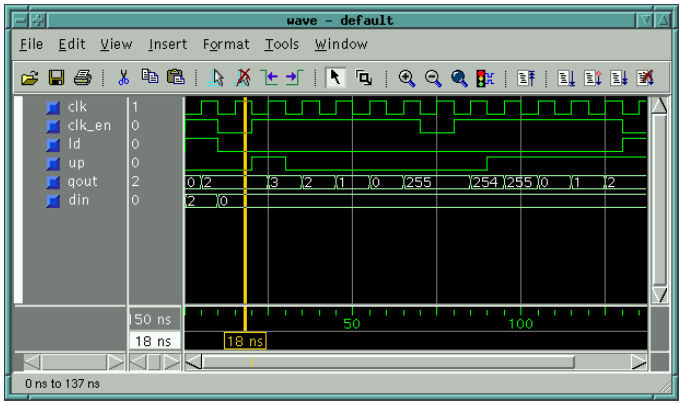
- Schreiben von "Testbenches"



Simulation (2)

- "Testbenches" überprüfen die Funktionalität eines Designs
- "Testbenches" haben oft eine beachtliche Größe
  - "Testbench" hat mehr Sourcecode als Design
  - Analogie zu Software-Entwurf
  - Modularisierung, Code reuse
- Simulationszeit ist beträchtlich

Timing-Diagramm



Verschiedene Ansätze

- Stimulus only
  - "Testbench" enthält Stimuli und DUT
  - Ergebnisse müssen im Simulator überprüft werden
- Full Testbench
  - "Testbench" enthält neben Stimuli und DUT auch die korrekten Ergebnisse und führt einen Vergleich durch
- Simulator-spezifisch
  - spezielle Simulatorbefehle werden verwendet

Test Generation (1)

- Lesen aus Datei
  - Kleine, klar strukturierte "Testbench"
  - Datei kann extern erzeugt werden
    - Z.B. durch ein C-Programm
  - Eingabedatei ist recht unflexibel oder schwer verständlich
  - Fehler in der Eingabedatei werden erst während der Simulation gefunden

Test Generation (2)

- Verwendung von Verhaltensbeschreibungen
  - Behavioral Model in VHDL
  - Beschreibung des Verhaltens auf einer höheren Abstraktionsebene
  - Simulation des Behavioral Models oft wesentlich effizienter als RTL-Beschreibung

Simulation: Vor- und Nachteile

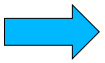
+ System kann auf verschiedenen Abstraktionsebenen modelliert werden
+ Größe des Simulationsmodells wächst linear mit der Systemgröße
– Geringe Abdeckung
– Schwierig, "gute" Eingabevektoren zu finden
– Beispiel: Ein 256 bit RAM Entwurf braucht $2^{256} \simeq 10^{77}$ Simulationsläufe

Die Simulations-Krise

- >50% der Entwurfszeit wird für RTL- und Netzlistensimulation verwendet
- Größere Entwürfe führen zu steigenden Simulationszeiten und gleichzeitig noch geringerer Abdeckung

Systemebene

- Angenommen,
  - Jedes Modul ist mit einer Wahrscheinlichkeit von 95% korrekt
  - Ein System hat 20 Module
- Dann ist das Gesamtsystem mit einer Wahrscheinlichkeit von nur 0.95^{20} = 36% korrekt



Korrektheit der Module ist von hoher Bedeutung

Andere Verifikationstechniken

- Property Checking / Model Checking
  - Beweisen von Eigenschaften auf einer Schaltung
- Formal Linting
  - Ausführen vordefinierter Tests
- Combinational Equivalence Checking
  - Vergleich der kombinatorischen Logik
- Timing Analysis
- Power Estimation

Property Checking

- Eigenschaften für eine Komponente *für alle Eingabevektoren* werden bewiesen
  - Äquivalent zu erschöpfender Simulation, aber effizienter
  - Die Eigenschaft muss (meist) auf ein Zeitfenster endlicher Länge beschränkt sein
- Beispiel: "Schreiben in SDRAM findet mindestens 19 Clock-Zyklen nach dem Lesen aus dem ROM statt"

Property Checking (2)

- Komponenten mit 100K Gattern können in wenigen Minuten bewiesen werden
- Oft werden ungewöhnliche Fehler gefunden, die durch Simulation nicht entdeckt werden

(Mindestens) Ergänzung zur Simulation

Linting

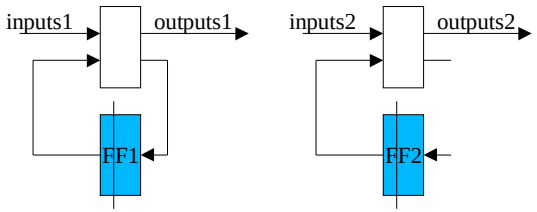
- Vergleichbar mit "lint" für C
- Fehler und fehleranfällige Befehle werden aufgezeigt
  - Nicht passende Typen,
  - Mehrere Treiber für ein Signal,
  - Signale ohne Treiber, etc.
- Kann nicht unterscheiden zwischen "gewollten" und "ungewollten" Anweisungen

Formal Linting

- Vordefinierte Eigenschaften werden formal überprüft
  - Werden automatisiert aus der RTL-Beschreibung extrahiert
- Beispiele:
  - Arrayzugriffe außerhalb der Grenzen
  - Werte außerhalb des gültigen Bereichs
  - 'X' Zuweisungen

Combinational Equivalence Checking (CEC)

- Äquivalenzvergleich zweier Schaltungen
  - kombinatorisch: Flipflops werden aufgebrochen und zu zusätzlichen Ein- und Ausgängen



Equivalence Checking (cont.)

- CEC kann verwendet werden für
  - Vergleich RTL gegen RTL
    - Nach Umschreiben des RTL, etwa in eine andere Sprache
  - Vergleich RTL gegen Netzliste
    - Nach der Synthese, Logik-Optimierung, usw.
  - Vergleich Netzliste gegen Netzliste
    - Austausch der Bibliothek, Routing, clock tree insertion, usw.

Equivalence Checking (cont.)

- CEC kann **nicht** verwendet werden,
  - wenn sich die Kodierung der Zustände ändert oder vom Synthese-Tool (beliebig) festgelegt wurde
  - wenn sich die Zahl der erreichbaren Zustände ändert
  - nicht alle Abstraktionsebenen werden unterstützt
    - C/C++, Transistor-Ebene

Einsatz von CEC

- CEC kann komplette ASICs vergleichen
  - > 5 Millionen Gatter
- Im Gegensatz zu Property Checking ist wenig Wissen über die Schaltung notwendig

Timing Analysis

- Abschätzung des Delays auf dem langsamsten Pfad
- Kann auf verschiedenen Abstraktionsebenen durchgeführt werden
  - Je nach Abstraktionsebene gibt es unterschiedlich genaue "Timing-Modelle"
    - Kapazitive Lasten der Gatter können erst nach dem Technology Mapping bestimmt werden
    - Verzögerungen der Leitungsbahnen können erst nach dem Routing berechnet werden

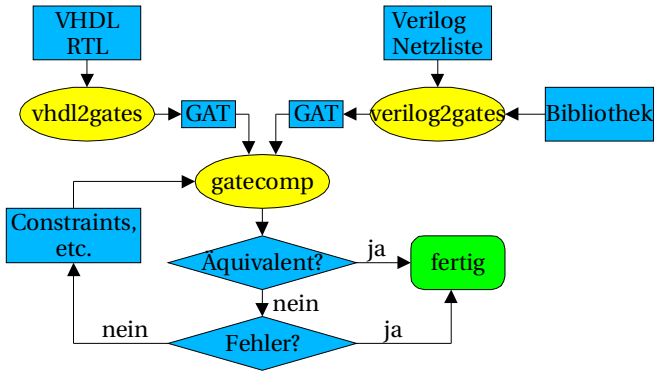
Power Estimation

- Verschiedene Fragestellungen:
  - Energieverbrauch insgesamt
    - Wie lange hält die Batterie / Wie viel Wärme entsteht?
  - Leistung maximal
    - Maximalleistung der Batterie?
  - Energieverbrauch lokal
    - Gibt es Teile des Chips, die besonders warm werden?

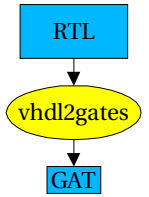
Beispiel: Formale Verifikation bei Infineon

- CVE (Circuit Verification Environment)
- Umfang von CVE:
  - VHDL-Compiler (vhdl2gates)
  - Verilog-Compiler (verilog2gates, verilogRTL2gates)
  - Equivalence Checker (gatecomp)
  - Property Checker (gateprop)
  - ... und weitere Programme

CVE: Equivalence Checking

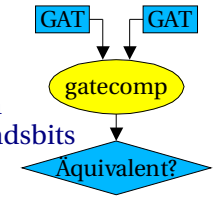


CVE: VHDL/Verilog Compiler



- Übersetzen von VHDL/Verilog in eine Netzliste (Logiksynthese)
  - Einfache Grundgatter
- Eigenschaften:
  - Logik braucht nicht optimiert zu werden
  - Sollte Simulations-Semantik folgen
  - Warnungen bei "problematischen" Anweisungen

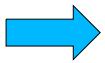
CVE: GateComp



- Matching
  - Finden von korrespondierenden Ein-/Ausgängen und von Zustandsbits der beiden Schaltungen
  - Namen werden durch Synthese zum Teil verändert
  - Namen sind oft nicht eindeutig
  - Verschiedene Verfahren sind implementiert
- Vergleich von Ausgängen und Übergangsfunktionen

CVE: GateComp (2)

- Methoden zum Vergleich der Funktionen:
  - Binary Decision Diagrams (BDDs)
  - SAT (Erfüllbarkeit von Formeln in konjunktiver Normalform)
  - Test-Pattern Generation (ATPG)
  - Finden von äquivalenten Punkten in den beiden Schaltungen



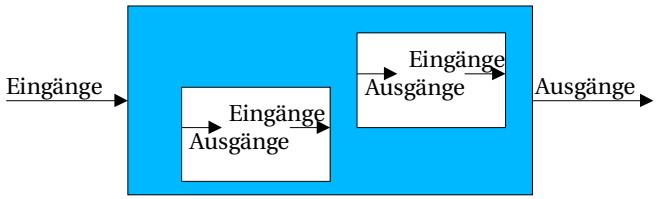
Inhalt dieser Vorlesung

Black Boxing

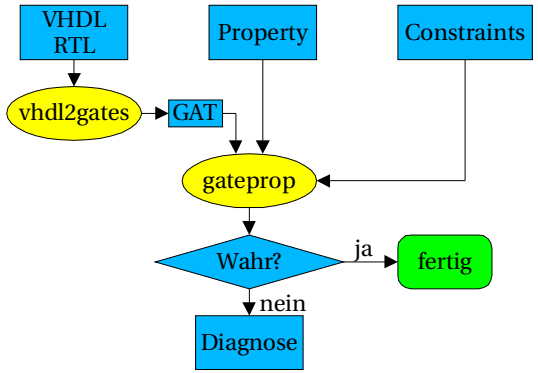
- Bestimmte Teile sollen von der Verifikation ausgeschlossen werden
  - Speicher
  - komplexe Arithmetik
  - unvollständige Implementierung

Black Boxing (2)

- Auswirkung:
  - Eingänge der "Black Box" werden zu Ausgängen
  - Ausgänge der "Black Box" werden zu Eingängen



CVE: Property Checking

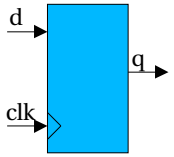


Eigenschaften

- Beschreiben eine Spezifikation
- Ein System kann durch mehrere Eigenschaften beschrieben werden (Übersichtlichkeit)
- Randbedingungen müssen berücksichtigt werden
- Vollständige Beschreibung ist wünschenswert

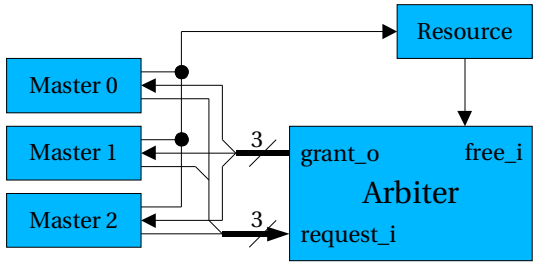
Eigenschaften eines Flipflops

- Der Wert von d steht einen Takt später am Ausgang q
- Randbedingung: d und clk schalten nicht gleichzeitig



Eigenschaften eines Arbiters

- Randbedingungen:
  - auf einen Grant folgt genau ein free
  - ohne Grant kein free



Eigenschaften des Arbiters

- Höchstens ein Grant
  - grant_o(2) + grant_o(1) + grant_o(0) <= 1
- Ohne Grant gibt es keinen Request
  - grant_o(0) → *prev*(request_i(0));
- Kein Grant solange die Resource beschäftigt ist
- Jeder Request führt zu einem Grant
- ...

Bemerkungen

- Eigenschaften können sich auf verschiedene Zeitpunkte beziehen
- Reset-Verhalten muss oft getrennt bewiesen werden
  - Ein Request führt nicht zu einem Grant, wenn der Reset gezogen wurde
- Randbedingungen sind getrennt zu beweisen

Erreichbarkeit

- Problem: Nicht alle Zustände sind (sequentiell) erreichbar
  - `signal s : integer range 0 to 5;`
  - s wird durch 3 Bits dargestellt
  - Die Werte 6 und 7 können nicht auftreten
    - Illegal, aber bei fehlerhaften Schaltungen möglich...
  - Je nach Umgebung können auch andere Werte nicht auftreten
    - Etwa, wenn der Wert 4 nie zugewiesen wird

Erreichbarkeit (2)

- Gegenbeispiele brauchen nicht erreichbar sein
  - Die Schaltungen verhalten sich in nicht erreichbaren Zuständen unterschiedlich (*Equivalence Checking*)
  - In einem nicht erreichbaren Zustand tritt ein Fehler auf (*Property Checking*)
- Eine Erreichbarkeitsanalyse ist schwierig (hohe Komplexität) und deshalb in der Regel bei großen Schaltungen nicht möglich
  - Approximationstechniken