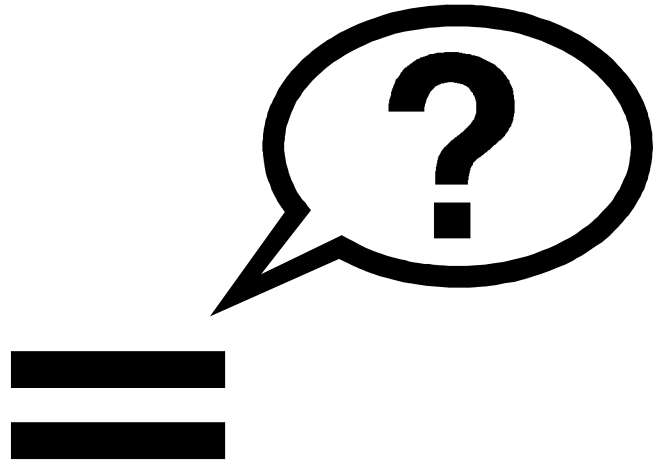
Verifikation



Wozu Verifikation?

- Produktivität im ASIC Entwurf
  - Der Entwurf eines 1–million gate ASIC benötigt ca. 2000 Mann–Tage
  - Quality Assurance benötigt 50% der Zeit/des Geldes
- Produktivity Gap
  - ASICs mit 5 Millionen Gattern werden in kürze üblich
  - Wenn der Entwurfsprozess sich nicht verändert, sind dazu 45 Designer notwendig
  - Weder Man–Power noch Geld sind dafür vorhanden

Wege aus der Krise

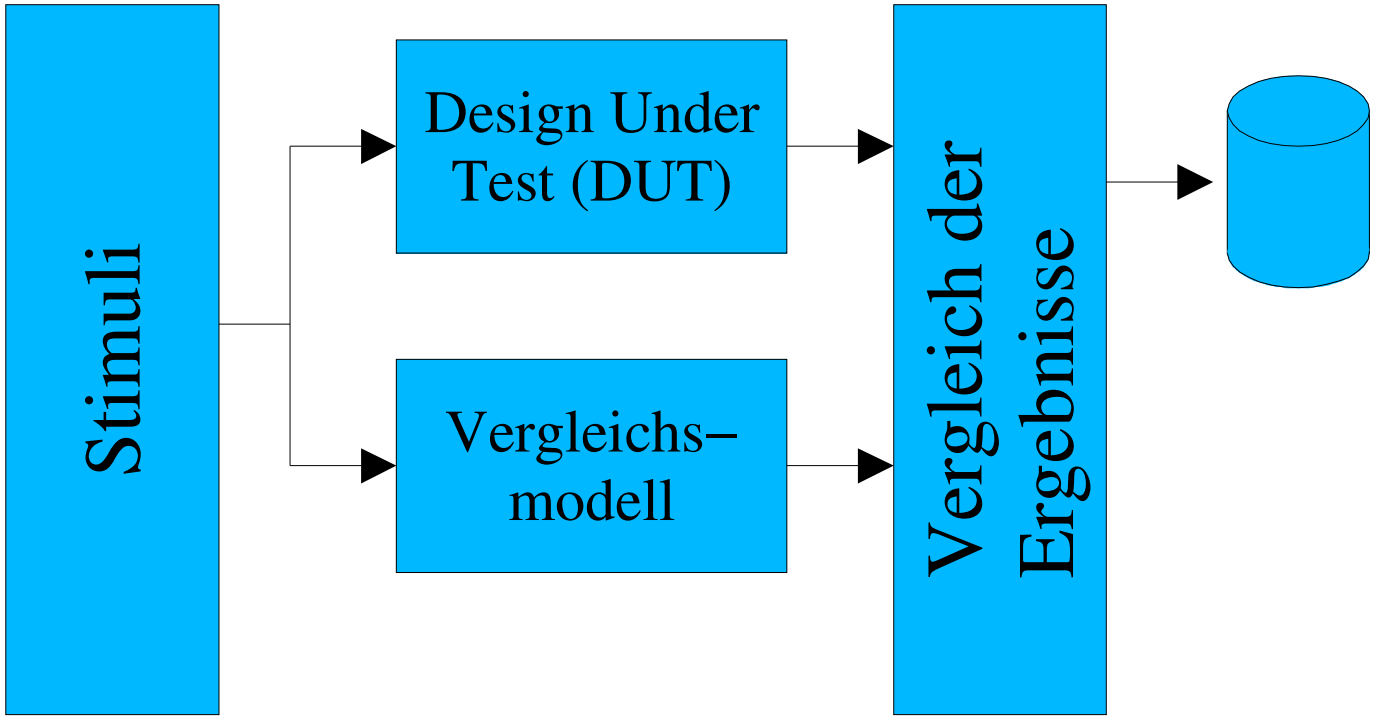
- Wiederverwendung von bereits entworfenen Teilen (*design reuse*)
- Weiterentwicklung der traditionellen Quality Assurance (QA)–Techniken
  - Simulation
  - Testen
- Neue QA–Techniken
  - Formale Verifikation

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

Beispiel: Zähler

```
package count_types is
   subtype bit8 is integer range 0 to 255;
end count_types;

library ieee;
use ieee.std_logic_1164.all;
use work.count_types.all;
entity count is
   port(clk, clk_en : in  std_logic;
        ld, up      : in  std_logic;
        din         : in  bit8;
        qout        : out bit8);
end count;
```

Beispiel: Zähler (2)

```
architecture rtl of count is
   signal val, new_val : bit8;
begin
   process(ld,up,din,val)
   begin
      if ld = '1' then
         new_val <= din;
      elsif up = '1' then
         if val >= 255 then new_val <= 0;
                       else new_val <= val + 1;
         end if;
      else
         if val <= 0 then new_val <= 255;
                     else new_val <= val - 1;
   end if; end if; end process; ...
```

Beispiel: Zähler (3)

```
...
   process(clk)
   begin
      if clk'event and clk = '1' then
         if clk_en = '1' then
            val <= new_val;
         end if;
      end if;
   end process;

   qout <= val;
end rtl;
```

Beispiel: Testbench

```
entity testbench is end;

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.count_types.all;
architecture tb of testbench is

   component count
   port(clk, clk_en : in  std_logic;
        ld, up      : in  std_logic;
        din         : in  bit8;
        qout        : out bit8);
   end component;
```

Beispiel: Testbench (2)

```
   ...
   signal clk, clk_en, ld, up : std_logic := 0;
   signal qout, din : bit8;
begin

   -- instantiate design under test
dut: count
     port map(clk => clk, clk_en => clk_en,
              ld => ld, up => up,
              din => din, qout => qout);

clock_gen: clk <= not clk after 5ns;
...
```

Beispiel: Testbench (3)

```
   ...
   -- provide stimuli and check results
test: process
    -- file containing stimuli and results
    file vector_file : text is in "counter.txt";
    -- a line of the file
    variable l : line;
    variable good_number, good_val : boolean;
    variable space : character;
    variable expected_result : bit8;
    variable clk_en_v, ld_v, up_v : std_logic;
    variable din_v : bit8;
begin
   ...
```

Beispiel: Testbench (4)

```
begin
  while not endfile(vector_file) loop
    readline(vector_file, l);

    read(l, clk_en_v, good_val);
    assert good_val
    report "bad clk_en value";

    read(l, ld_v, good_val);
    assert good_val
    report "bad ld value";
    ...
```

```
-- counter.txt
-- clk_en ld up
-- din qout
110 2 2
000 2 2
101 0 3
...
```

Beispiel: Testbench (5)

```
-- counter.txt
-- clk_en ld up
-- din qout
110 2 2
000 2 2
101 0 3
...
```

```
...
read(l, up_v, good_val);
assert good_val
   report "bad up value";

read(l, space);  -- skip space
read(l, din_v, good_val);
assert good_val report "bad din value";

read(l, space);  -- skip space
read(l, expected_result, good_val);
assert good_val report "bad qout value";
...
```

Beispiel: Testbench (6)

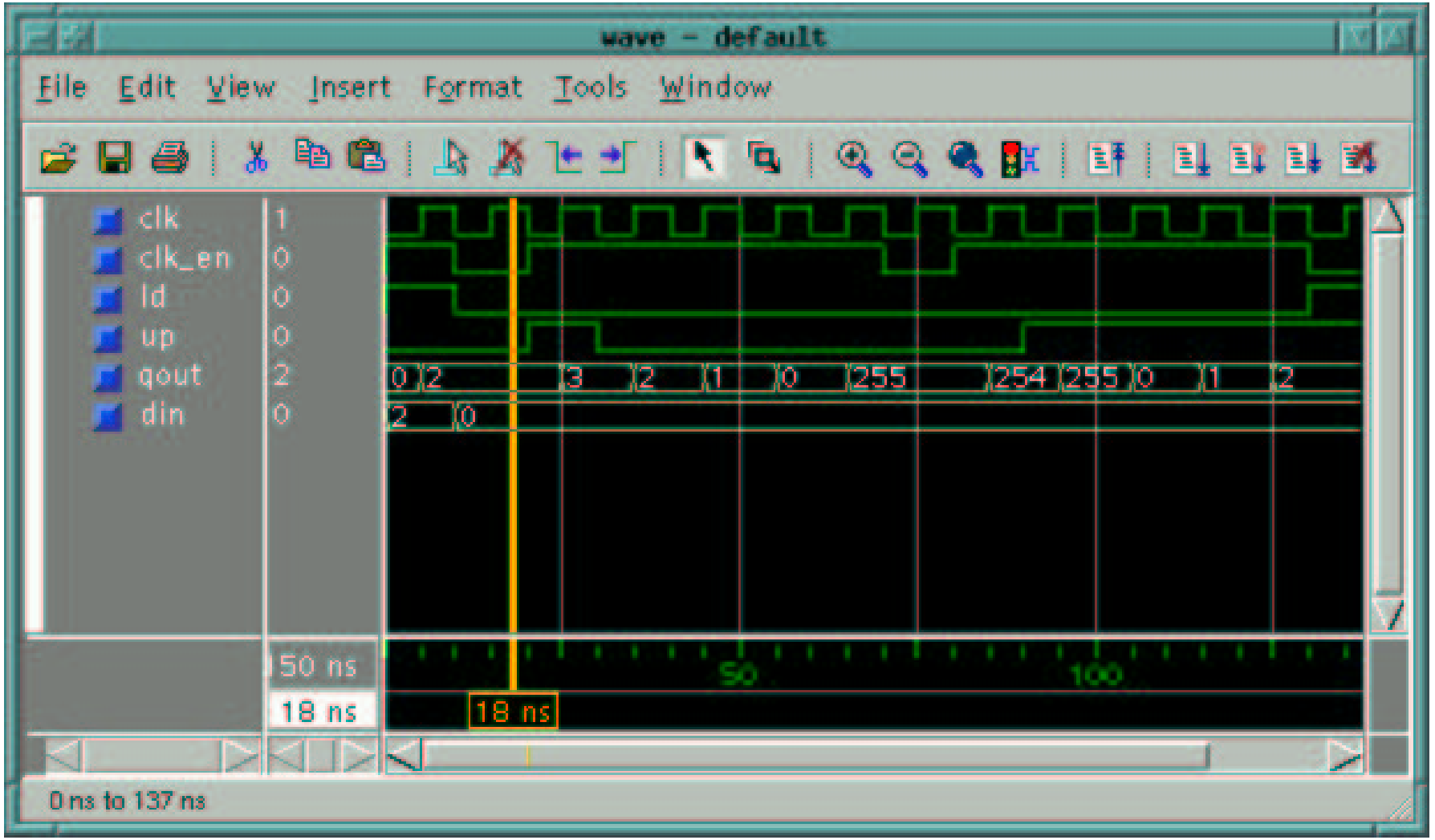
```
      ...
      clk_en <= clk_en_v; ld  <= ld_v;
      up     <= up_v;     din <= din_v;

      -- wait until clk '0' -> '1' -> '0'
      wait until clk='0';

      assert qout = expected_result
         report "wrong result";

    end loop;
    assert false report "test complete";
    wait;
  end process;
end tb;
```

Timing–Diagramm



Verschiedene Testmethoden

- Stimulus only
  - Testbench enthält Stimuli und DUT
  - Ergebnisse müssen im Simulator überprüft werden
- Full Testbench
  - Stimuli, DUT, korrekte Ergebnisse, Vergleich
- Simulator–spezifisch
  - spezielle Simulatorbefehle werden verwendet

Test Generation (1)

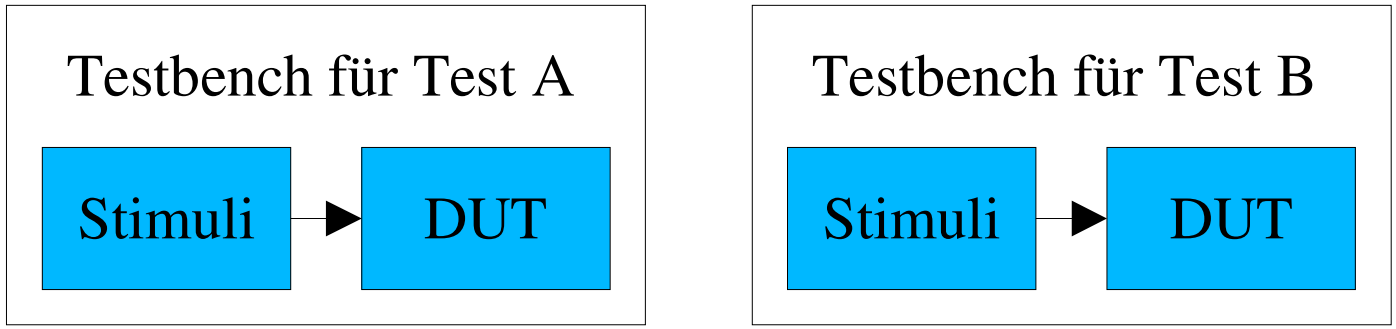
- Lesen aus Datei
  - Kleine, klar strukturierte Testbench
  - Datei kann extern erzeugt werden
    - Z.B. durch ein C–Programm
  - Eingabedatei ist recht unflexibel oder schwer verständlich
  - Fehler in der Eingabedatei werden erst während der Simulation gefunden

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

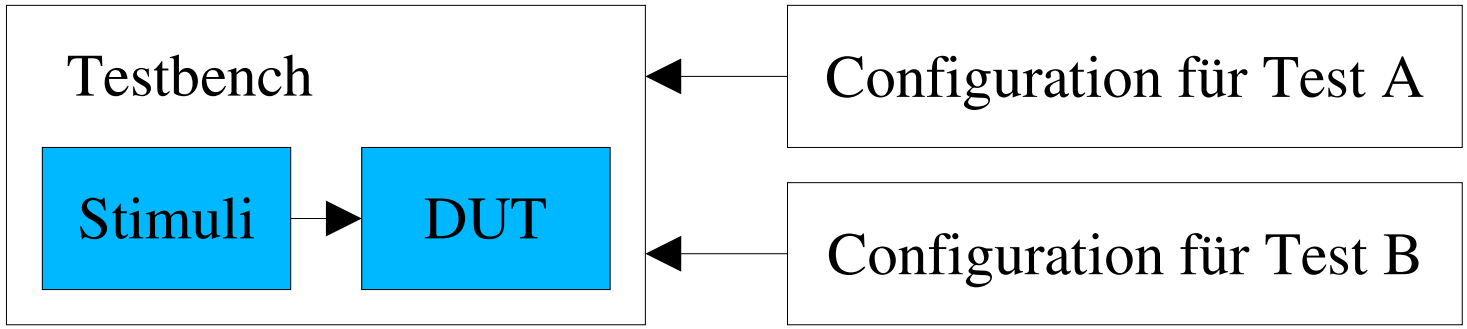
Test Selection (1)

- Verwendung verschiedener Entities
  - Testbench für jeden Test, in VHDL
  - Viele Entities
  - Viel duplizierter Code



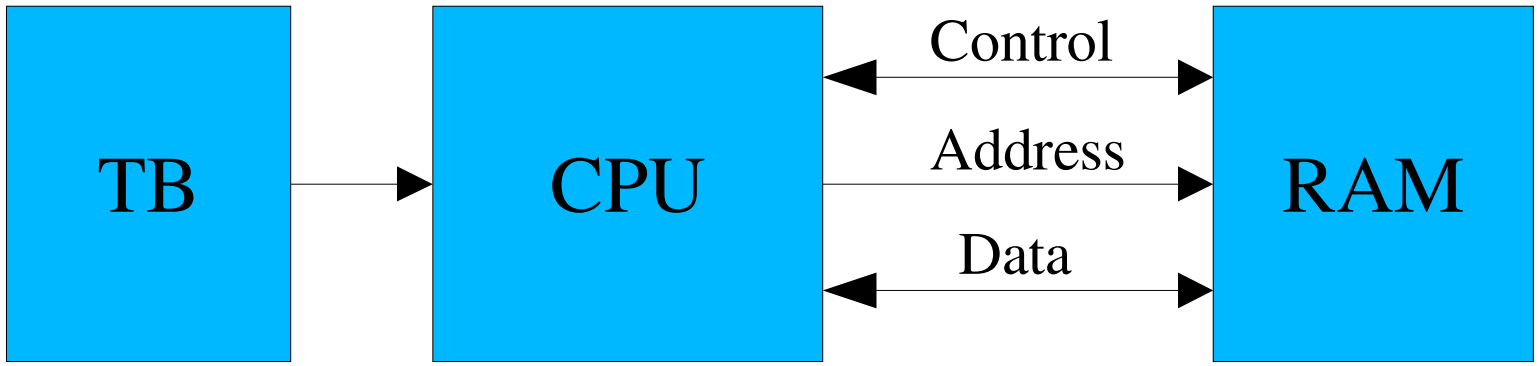
Test Selection (2)

- Verwendung von Konfigurationen
  - Eine Testbench, in VHDL
  - Auswahl der Tests durch Konfigurationen
  - Testbench kann sehr komplex werden



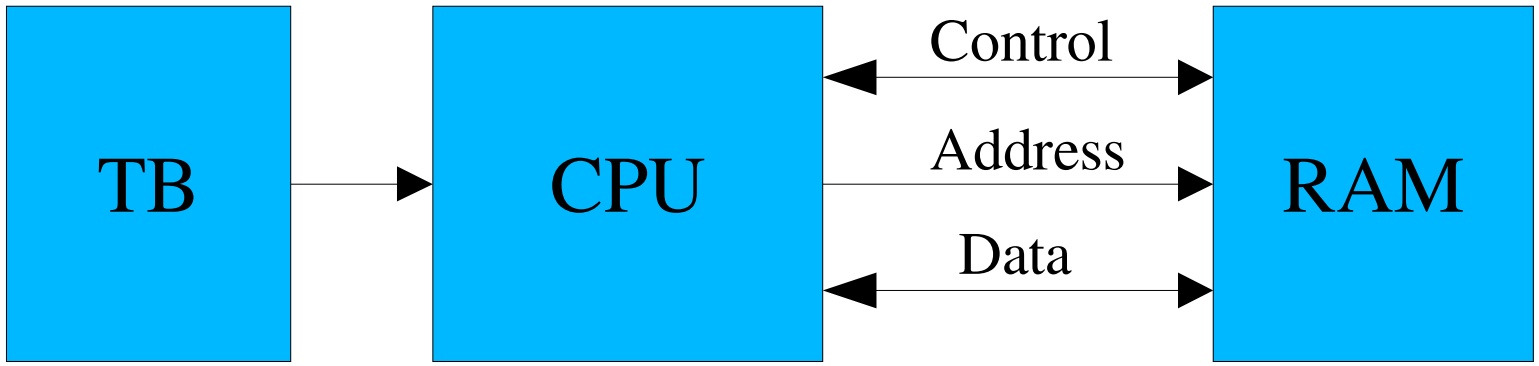
Beispiel: Speicherzugriff

- Simulation von Speicherzugriffen
  - Lesezugriff
  - Schreibzugriff



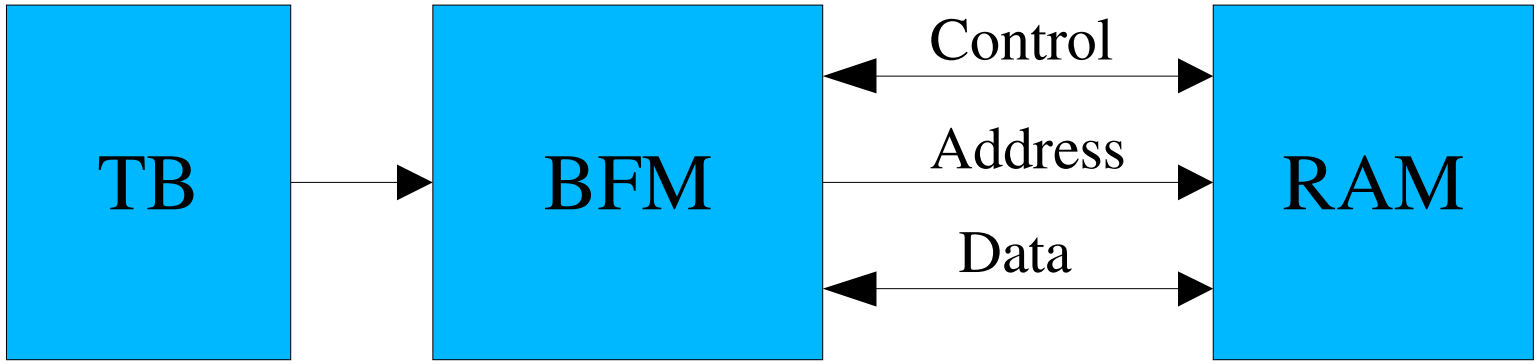
Speicherzugriff (2)

- Nachteil:
  - Simulieren der CPU benötigt viel Rechenzeit
  - Getestet werden soll nur das Verhalten des RAMs



Bus Functional Model

- Verwendung eines Bus Functional Models (BFM) statt der kompletten CPU
  - Einfacheres Modell einer Komponente, das nur den Effekt auf den Bus modelliert



- wesentlich geringere Laufzeiten des Simulators

Strukturierung

- Bekannt aus Software–Design:
  - Datenkapselung
  - Information Hiding
  - Code Reuse
- Übertragen auf Testbench–Entwurf:
  - Keine globalen Variablen/Signale
  - Prozedurale Interfaces

Procedurale Interfaces

```
package i386sx is

subtype add_typ is std_logic_vector(23 downto 0);
subtype dat_typ is std_logic_vector(15 downto 0);

procedure read(raddr: in  add_typ;
               rdata: out dat_typ;
         signal clk  : in  std_logic;
         signal addr : out add_typ;
         signal ads  : out std_logic;
         signal rw   : out std_logic;
         signal ready: in  std_logic;
         signal data : inout std_logic);  ...
```

Procedurale Interfaces (2)

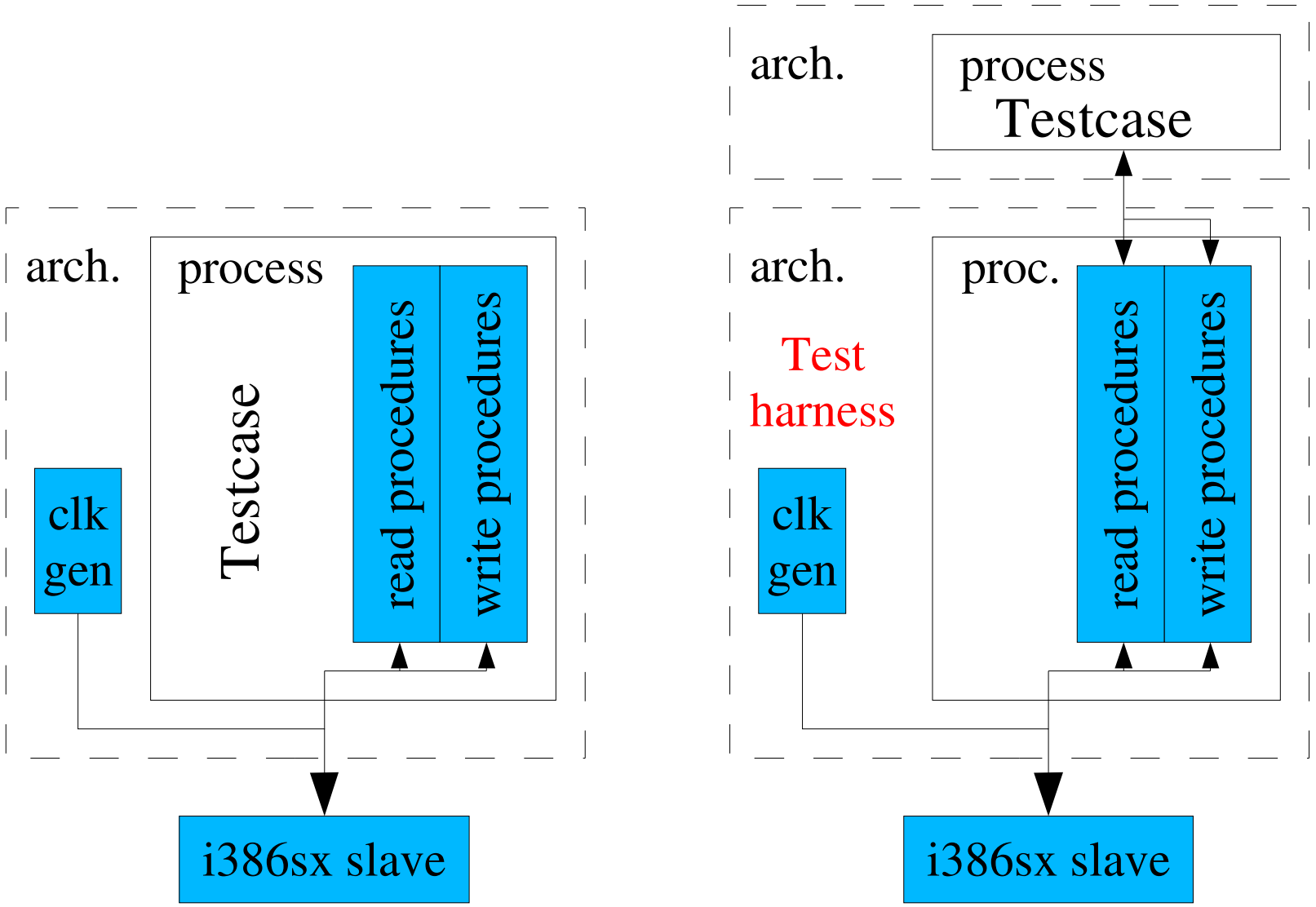
```
use work.i386sx.all;
architecture test of bench is
   signal clk  : std_logic;
   signal addr : add_type;
   ...
begin
dut : design port map(..., clk, addr, ...);

testcase: process
begin
   ...
   read(..., clk, addr, ...);
   ...
```

Prozedurales Interface (3)

- Kapselung der Schnittstelle zu lokalen Daten
- Wiederverwendung
  - Wenn ein anderer Prozessor simuliert werden soll...
- Unabhängig von Implementierung
- Flexibilität
  - Vergleichbar zu Methoden einer Klasse in objektorientierten Programmiersprachen

Test Harness

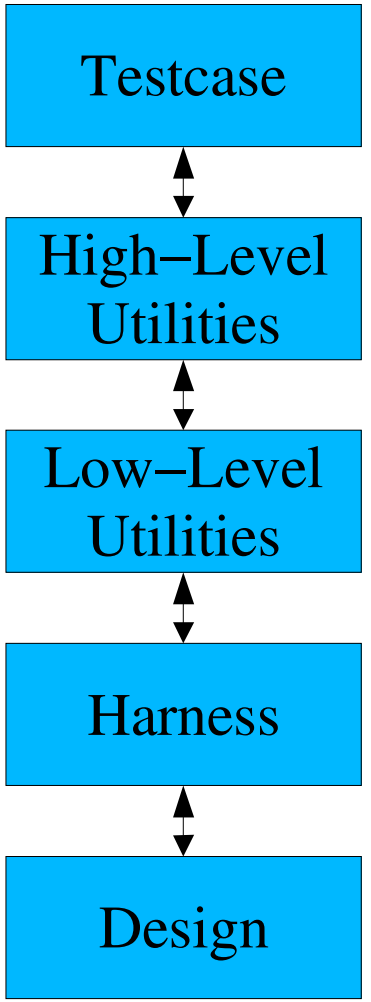


Test Harness (2)

- Gemeinsam nutzbare Funktionalität wird in einer Architecture zusammengefasst (*test harness*)
- Signale zum Design under Verification sind lokal im Test Harness
- Prozeduren müssen verwendet werden, um Stimuli anzulegen

Abstraktionsebenen

- ermöglichen Code Reuse
- verbessern Lesbarkeit
- Zyklische Abhängigkeiten sind möglich, aber nicht sinnvoll



Regressionstests

- Regressionstests sollen überprüfen, ob ein Design nach Änderungen kompatibel zur bereits früher verifizierten Funktionalität ist
  - Oft verursachen Änderungen, die einen Fehler beheben sollen, weitere Fehler
- Testbenches, die einmal erfolgreich gelaufen sind, sollten deshalb zu Regressionstests werden
- Regressionstests werden regelmäßig gestartet (jede Nacht)

Code Coverage

- Ansatz aus dem Software–Entwurf
- Kann während der Simulation protokolliert werden
  - Bei Software ist Instrumentierung notwendig
- Mängel der Testbenches finden
- Kann auch Information über das Design liefern

Coverage Metriken (1)

- Statement Coverage
  - Welche Statements werden ausgeführt?
  - Wenn ein Statement nicht ausgeführt wurde,
    - ist die Testbench nicht gut genug, oder
    - nicht erreichbarer Code ist gefunden und kann entfernt werden, oder
    - der Code soll nicht ausgeführt werden (Resetverhalten aus unerreichbaren Zuständen...)

Coverage Metriken (2)

- Path Coverage
  - Welche Kontrollpfade werden ausgeführt?
  - Nicht alle Kontrollpfade brauchen erreichbar zu sein
  - Exponentielle Anzahl an Pfaden in der Zahl der Kontrollanweisungen
  - 100% Path Coverage zu erreichen ist sehr oft sehr schwierig

Coverage Metriken (3)

- Expression Coverage
  - Welche Werte kann ein Ausdruck annehmen?
  - 100% Expression Coverage zu erreichen ist sehr oft unmöglich

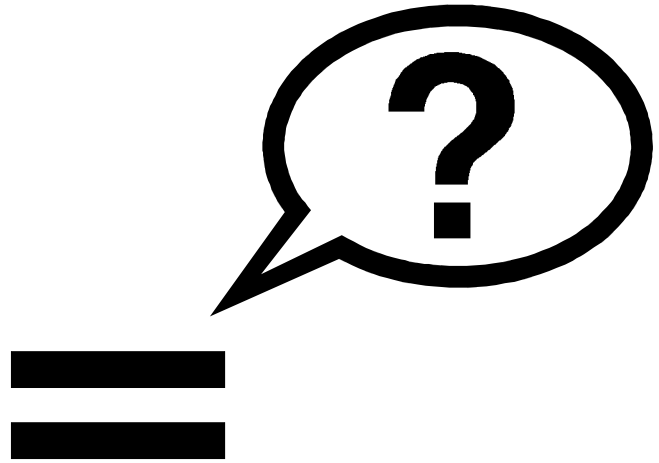
Code Coverage (2)

- Was bedeuten 100% Coverage?
  - ... nicht viel.
- Code coverage gibt an, wie genau der Source–Code ausgeführt wurde
- Code coverage liefert keine Aussage über funktionale Korrektheit
- Code coverage liefert nur eine Aussage, dass man noch nicht mit Testen fertig ist

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

Formale Verifikation

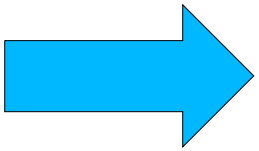


Die Simulations–Krise

- >50% der Entwurfszeit wird für RTL– und Netzlistensimulation verwendet
- Größere Entwürfe führen zu noch größeren 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
  - Beweisen von Eigenschaften auf einer Schaltung
- Formal Linting
  - Ausführen vordefinierter Tests
- Combinational Equivalence Checking
- 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

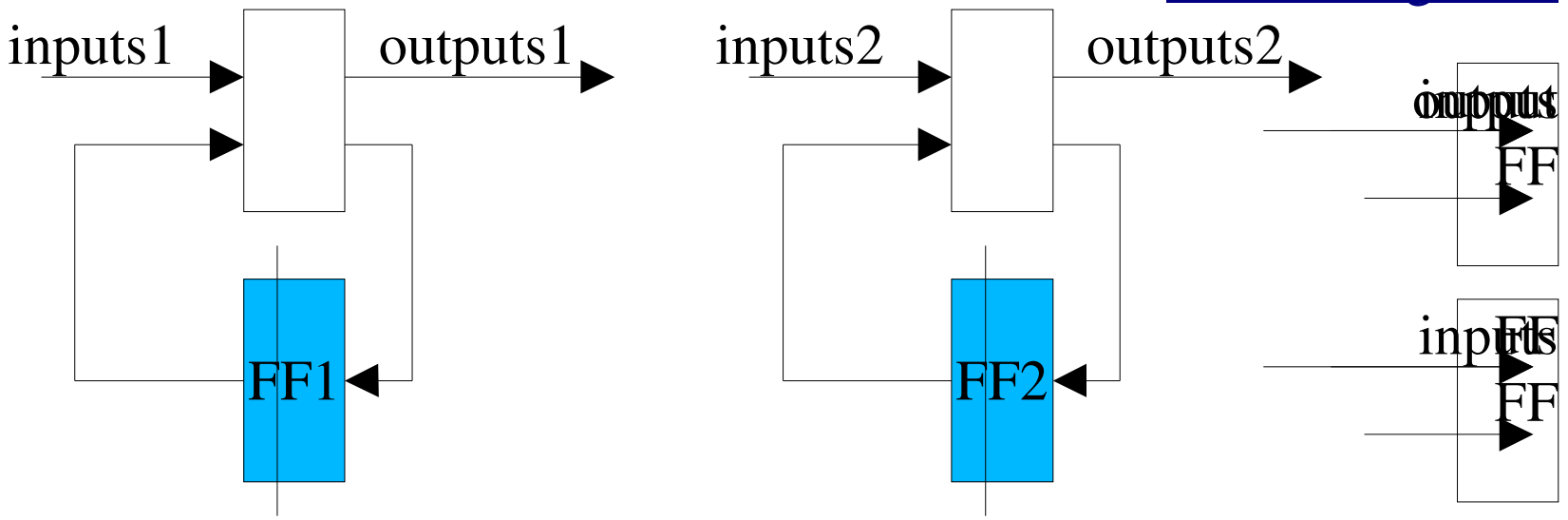
Formal Linting

- 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

Beweisaufgaben:



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

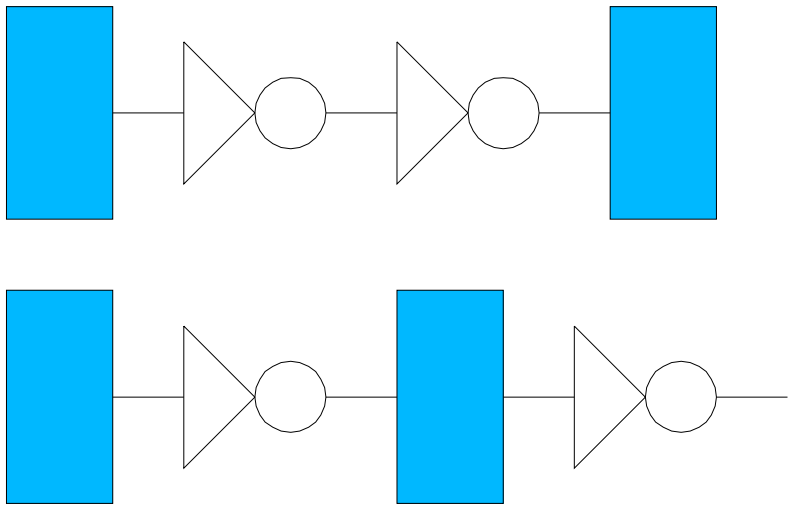
- CEC kann komplette ASICs vergleichen
  - > 4 Millionen Gatter
- Im Gegensatz zu Property Checking ist wenig Wissen über die Schaltung notwendig

Timing Analysis

- Abschätzung des Delays auf dem langsamsten Pfad
- Kann auf verschiedenen Abstraktionsebenen durchgeführt werden

Verbessern des Timings

- Re–Timing



- ... und viele andere Techniken

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?

Reduktion des Energieverbrauchs

- Clock Gating
  - Siehe Kapitel über Arithmetik
- Verbesserung der Kodierung von Zuständen
  - Siehe Kapitel über FSMs
- ... und vieles andere

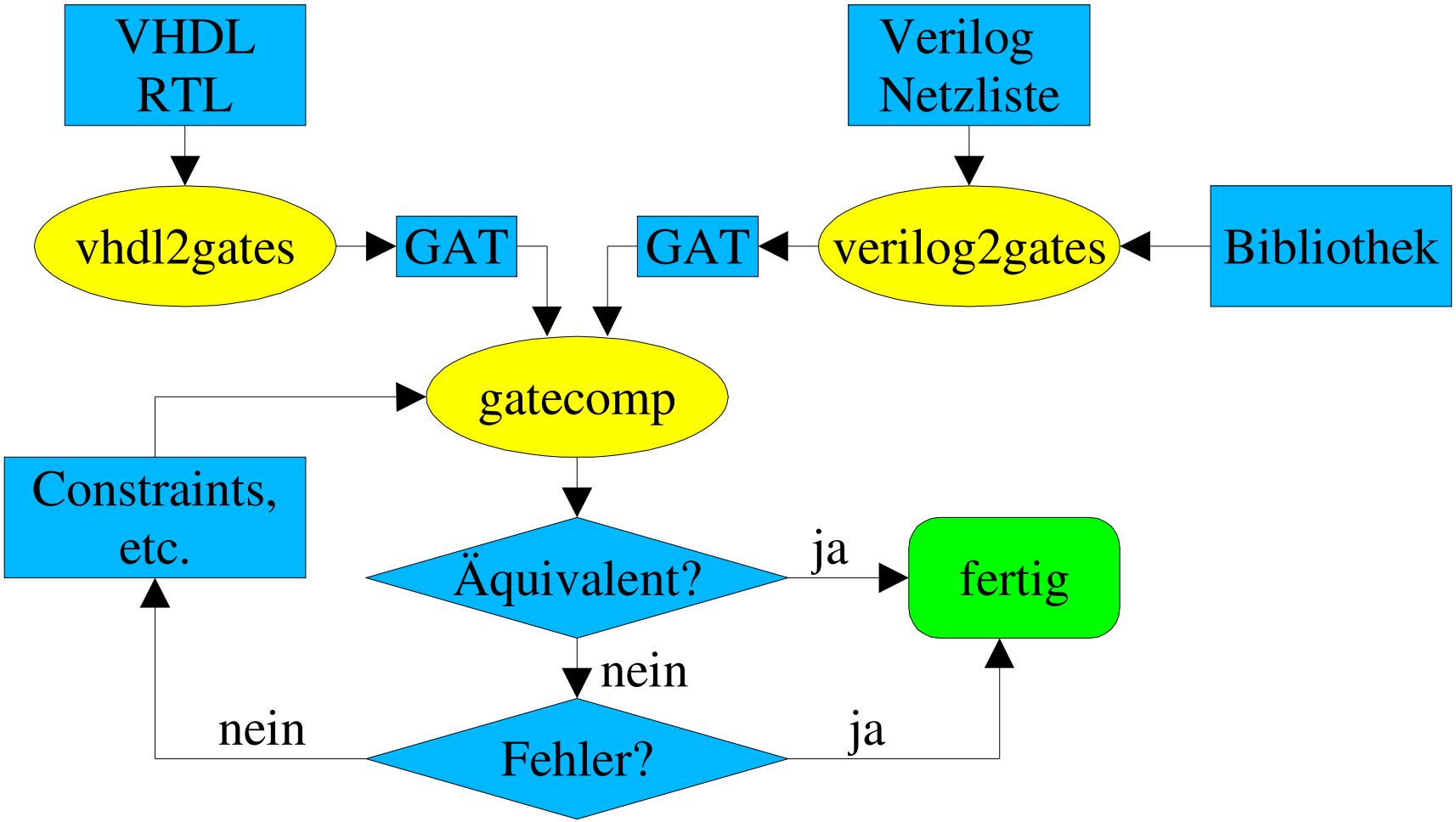
Übersicht

	Funktional	Timing
Implementation	Simulation Property Checking Linting	Static Timing Analysis
Vergleich	Simulation Equivalence Checking	Static Timing Analysis
Produktion	ATPG	At–Speed Test

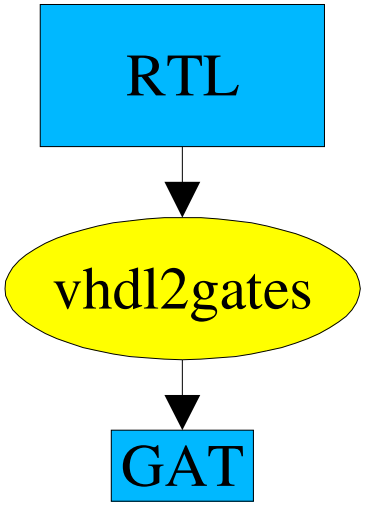
Verifikation bei Infineon: CVE

- Entwickelt bei Siemens CT SE 4
  - Seit 1. April 2002 bei Infineon CL DAT DF
- Umfang von CVE:
  - VHDL–Compiler (vhdl2gates)
  - Verilog–Compiler (verilog2gates, verilogRTL2gates)
  - Equivalence Checker (gatecomp)
  - Property Checker (gateprop)
  - ... und weitere Programme

CVE: Equivalence Checking

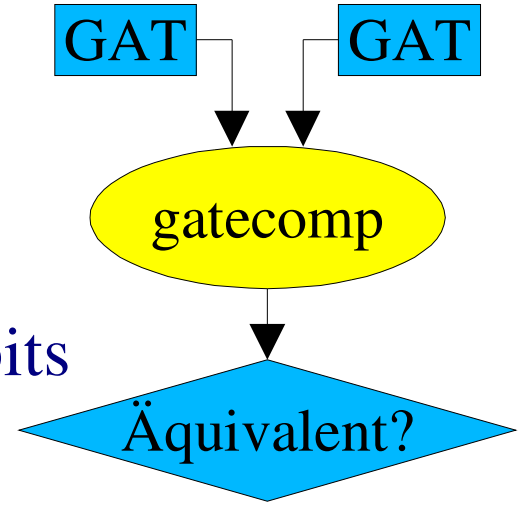


CVE: VHDL/Verilog Compiler



- 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

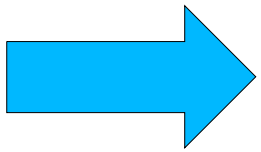
CVE: GateComp



- 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



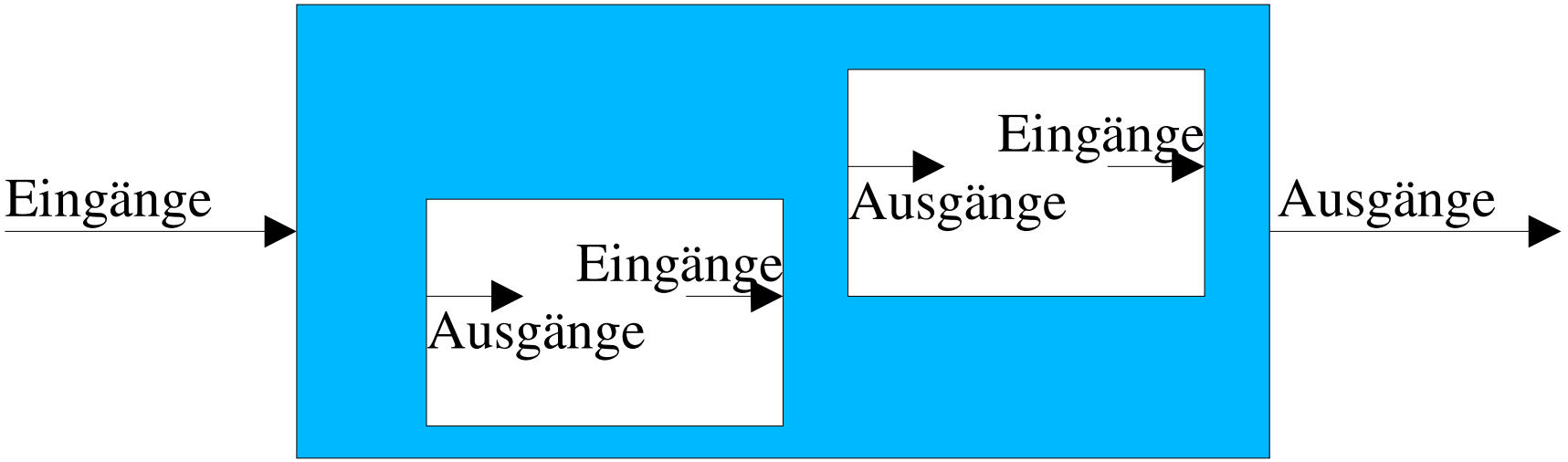
Spezialvorlesung

Black Boxing

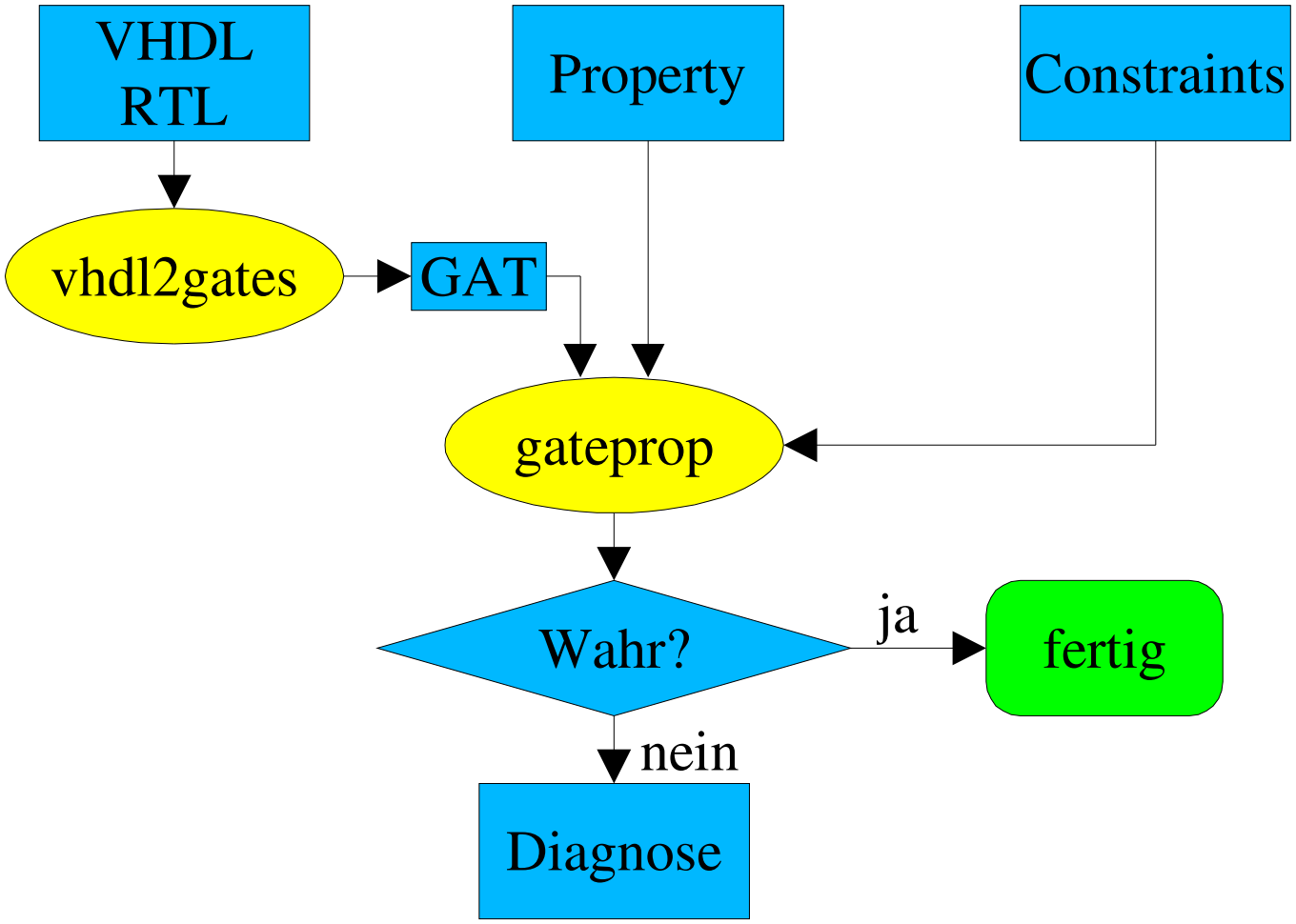
- Bestimmte Teile sollen von der Verifikation ausgeschlossen werden
  - Speicher
  - komplexe Arithmetik
  - unvollständige Implementierung

Black Boxing (2)

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

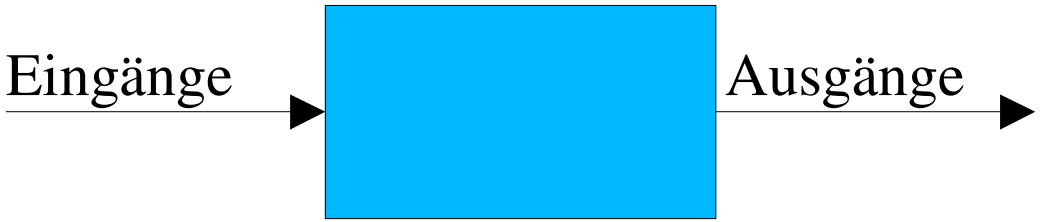


CVE: Property Checking



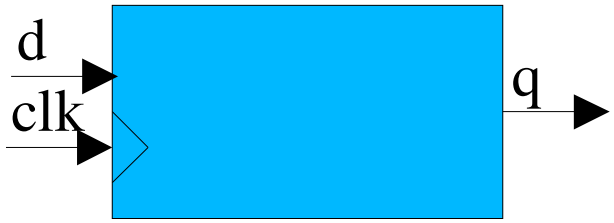
Eigenschaften

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



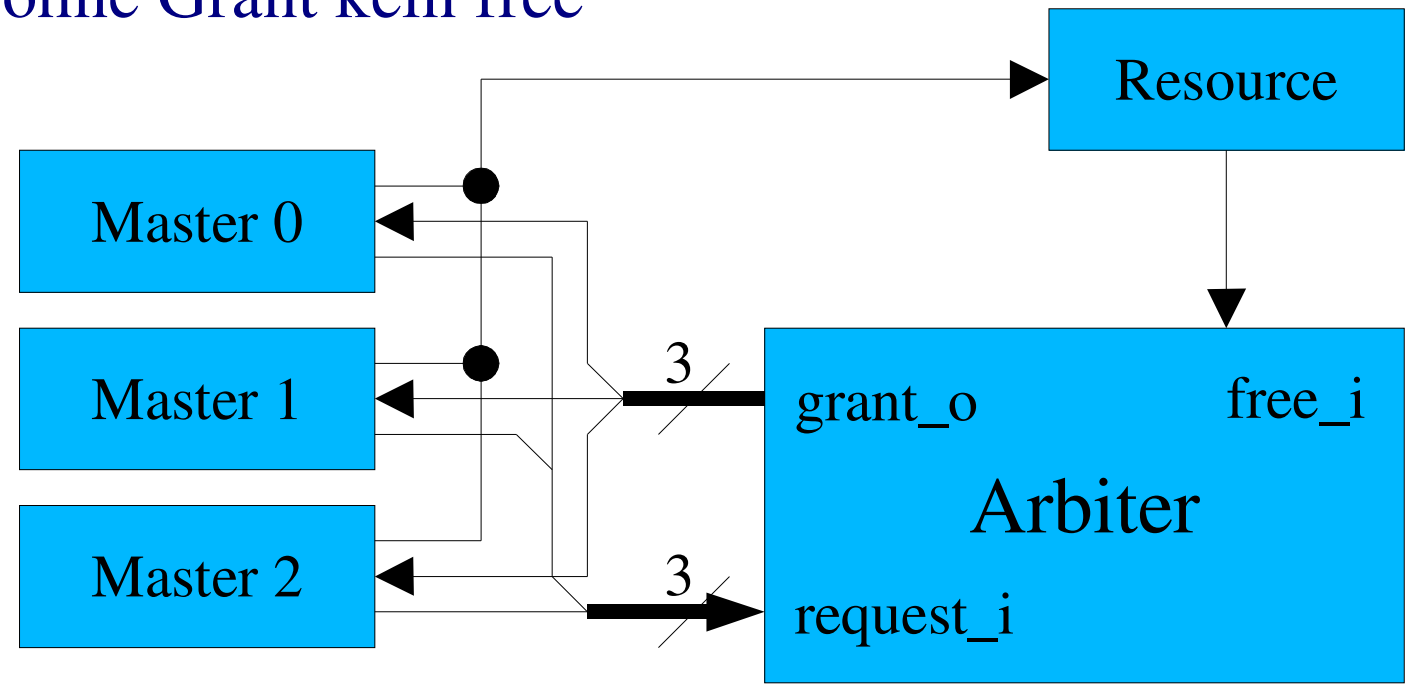
Eigenschaften eines Flipflops

- 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

Fehlerlokalisation

- Wenn ein Fehler gefunden wurde:
  - Wo liegt die Ursache?
- Beim *Equivalence Checking* wird meist ein Gegenbeispiel geliefert
  - Kann >50 Zustandsbits / Eingänge enthalten
- Beim *Property Checking* erstreckt sich das Gegenbeispiel über verschiedene Zeitschritte
  - Anzeige im Timing–Diagramm nicht immer ausreichend

Fehlerlokalisation (2)

- Wünschenswert wäre ein Hinweis auf die verursachende Zeile im RTL Code
  - schwierig
  - Aktuelles Forschungsgebiet

Zusammenfassung (1)

- Grundlagen von VHDL
  - Datentypen
  - Concurrent Statements
  - Sequential Statements
  - Subprograms (Funktionen, Prozeduren)
  - Strukturierung großer Projekte
    - Libraries
    - Packages

Zusammenfassung (2)

- Modellierung von FSMs
- Arithmetische Schaltungen
  - Addierer
  - Multiplizierer
  - Barrel Shifter
  - Clock Gating

Zusammenfassung (3)

- Verifikation
  - Simulation
  - Equivalence Checking
  - (Formal) Linting
  - Property Checking
  - Timing Analyse
  - Power Estimation