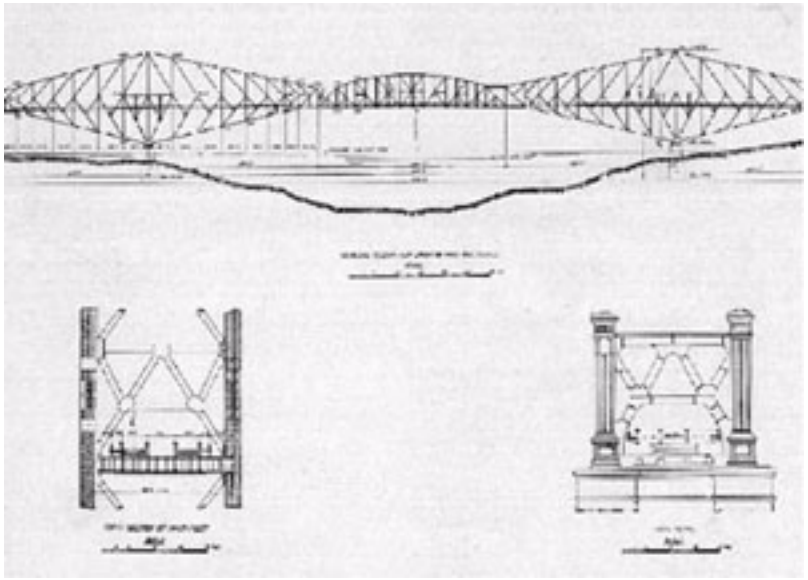
*Seminar “Pleiten, Pech und Pannen der Informatik”*

# Simulation und Testbenchmethodik

Jörg Frauenhoffer

Juli 2002

# Quebec Bridge Disaster, 29. August 1907



Begonnen wurde das Quebec Bridge Project 1887. Man plante eine Brücke über den St. Lorenz-Strom. Pläne wurden gezeichnet, doch weiter passierte die nächsten 13 Jahre nichts.

Im Jahre 1900 übernahm ein Amerikaner, Th. Cooper, die Leitung über das Projekt. Er war schon am Bau mehrer grosser Brücken in Amerika beteiligt gewesen.

# Quebec Bridge Disaster, 29. August 1907

Seine erste Massnahme war die Verlängerung der Spannweite der Brücke, von 1600 auf 1800 Fuss. Das sollte Kosten sparen, da die Stützpfeiler nun nicht mehr im Wasser gebaut werden mussten. Ausserdem würde die Quebec Bridge damit zur grössten Auslegerbrücke der Welt.

Aufgrund des guten Rufes, den Cooper genoss, und da Geld für Tests fehlte, wurde diese Veränderung ohne Beanstandung übernommen.

Als das Geld für den Bau zusammen war, verlangten die Kanadier vorher eine unabhängige Überprüfung der Baupläne. Doch Cooper, ausser sich vor Wut, sagte, man hätte schon genug Zeit verloren. So wurde 1903 mit dem Bau begonnen.

# Quebec Bridge Disaster, 29. August 1907

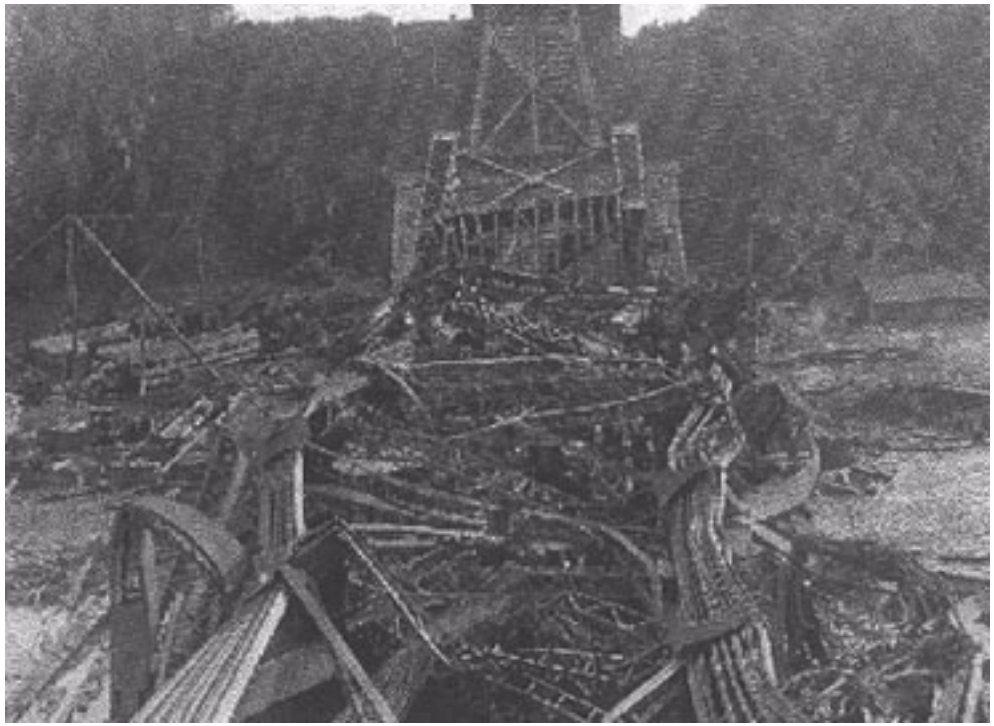
1907 wurden erste Verbiegungen an den Stahlträgern gemeldet. Obwohl Cooper dies anfangs ignorierte, sendete er am 27. August ein Telegramm, man solle die Arbeiten einstellen.

Doch die beauftragte Baufirma vermutete, dass die defekten Träger schon gebogen aus der Fabrik hier ankamen, und man machte sich weiter keine Gedanken.

Den nächsten Tag baute man weiter, ohne dass es Probleme gegeben hätte.

# Quebec Bridge Disaster, 29. August 1907

Am Abend des 29. August - die Feierabend-sirene war schon erklungen - brach die halb-fertige Brücke unter ihrem eigenen Gewicht zusammen. Von den 84 Arbeitern, die sich zu diesem Zeitpunkt noch auf der Brücke befanden, überlebten 11.



# Quebec Bridge Disaster, 29. August 1907

Übertragen auf die Mikrochip-Industrie:

- ein neues Design ist eine neue Herausforderung
- Fehler werden gemacht
- eine unabhängige Überprüfung ist unbedingt notwendig

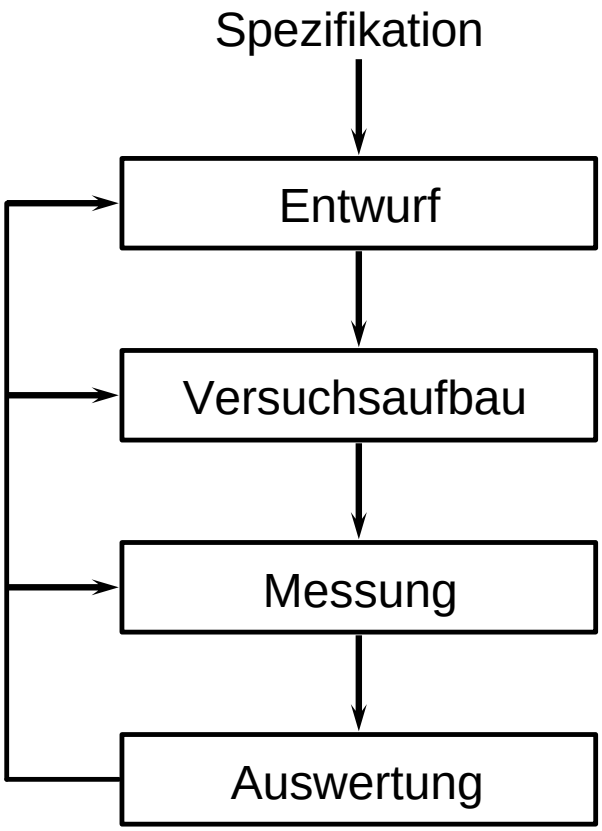
- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Hardware-Design

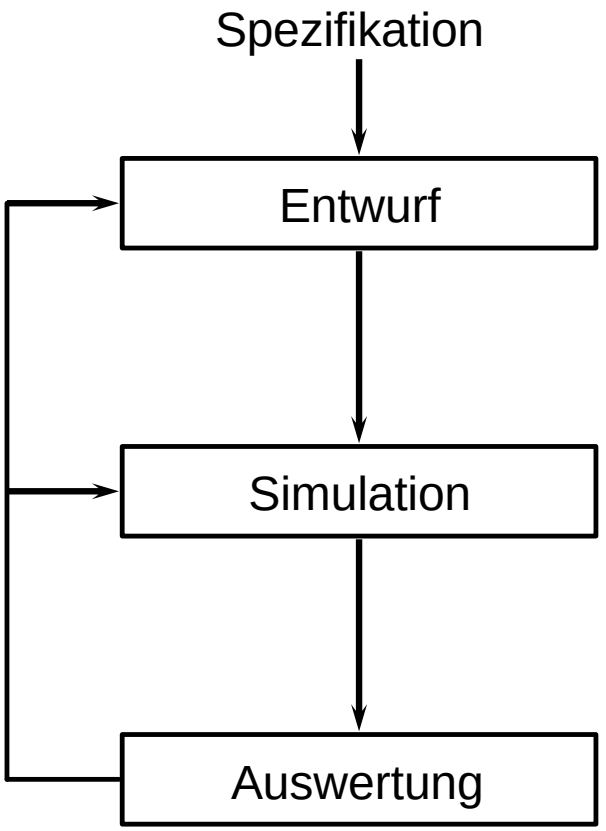
- früher:
  - grafische Editierung von Hardware
  - anschliessendes Umsetzen in reale Hardware zu Testzwecken
  - Fehlerkorrektur aufwendig
  - hohe Entwicklungszeit
- heute:
  - steigende Transistorzahlen pro IC
  - Dokumentation, Simulation rücken in den Vordergrund
  - Beschreibung der Hardware in textueller Form mittels Hardwarebeschreibungssprachen (*HDL*)

# Hardware-Design (Forts.)

früher:




heute:



# Hardware-Design (Forts.)

- Vorteile von HDLs:
  - Wiederverwendbarkeit
  - abstraktes, hohes Niveau (variabel)
  - problem-, nicht maschinenorientiert
- Beispiele für HDLs:
  - Verilog
  - VHDL
  - SystemC

# Hardware-Design (Forts.)

- formale Verifikation
- funktionale Verifikation:
  - kontrolliert, ob ein Design die gewünschte Funktionalität hat
  - mit Hilfe von Simulation
  - kein Beweis!
- Anteil der Verifikation am HW-Designprozess:
  - Infineon: ca. 70%
  - Synopsys: ca. 75%
  - Cisco Systems: ca. 80%

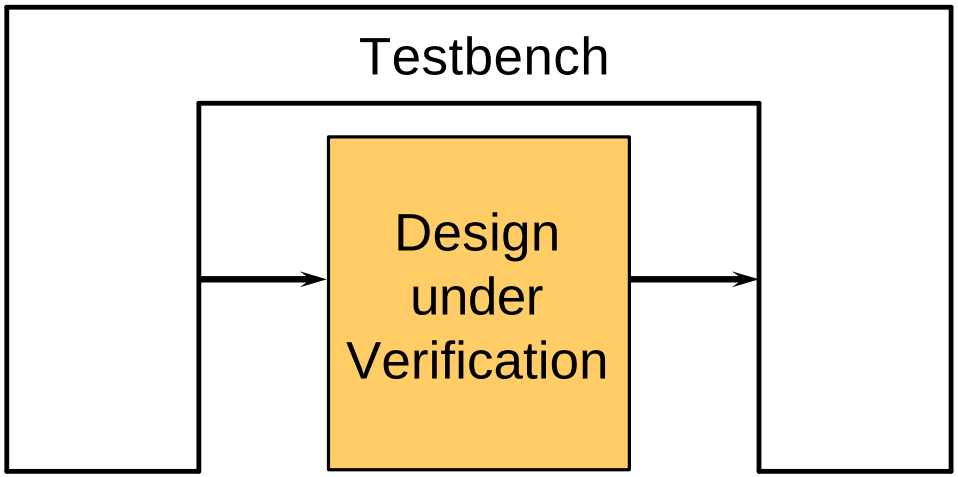
- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

## Simulatoren

- die verbreitetsten und bekanntesten Verifikationswerkzeuge
- ermöglichen Fehlererkennung und -beseitigung vor der Fertigung
- Versuch einer Annäherung an die Realität
- Vereinfachung bzw. Vernachlässigung mancher physischen Gegebenheiten (Abstraktionslevel)
- die zu simulierende Beschreibung des Designs muss aber der Spezifikation entsprechen

# Simulatoren (Forts.)

- keine statischen Werkzeuge
- betten das Design in eine künstliche Umwelt
  - ↳ Testbench:



## Simulatoren (Forts.)

- Testbench muss Inputs zur Verfügung stellen
- Simulator errechnet Outputs anhand der Beschreibung des Designs
- Korrektheit der Outputs muss vom Designer überprüft werden
- Abwesenheit von Fehlern kann nicht gezeigt werden, nur deren Anwesenheit!

- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Simulatorarten

- analog:
  - Timing und Werte bleiben erhalten
  - eher für kleine Schaltkreise geeignet, da aufwendig
  - Bsp.: *spice*
- digital:
  - Verlust von Timing und Werten
  - Simulation auf Basis logischer Funktionen und Werte
- Co-Simulatoren (später...)

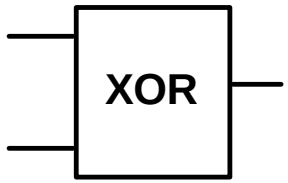
- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Event-Driven Simulation

- Problem: Simulatoren sind immer langsamer als die echte Welt (Transistoren ⟷ Computer)
- Performanz-Verbesserungen sind aber möglich:

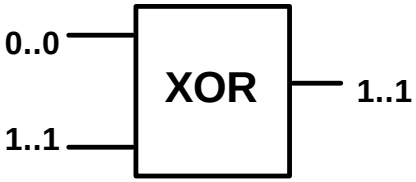
## Wann muss ein Modell ausgeführt werden?

```
XOR_GATE: process (A, B)
begin
   if A = B then
       O <= '0';
   else
       O <= '1';
   end if;
end process XOR_GATE;
```

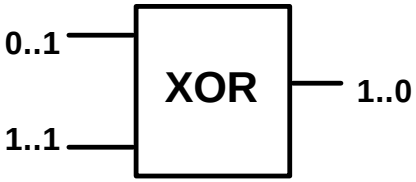


# Event-Driven Simulation (Forts.)

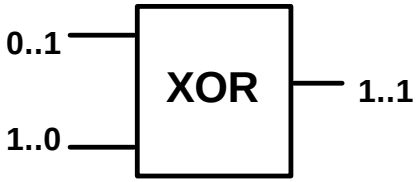
## Betrachte:




(a)



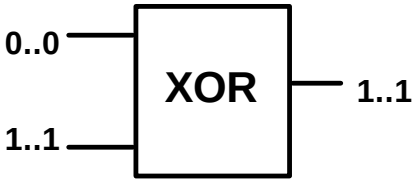

(b)



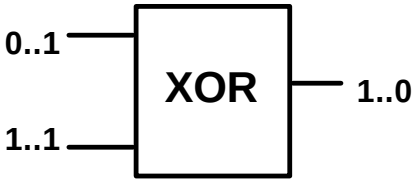

(c)

# Event-Driven Simulation (Forts.)

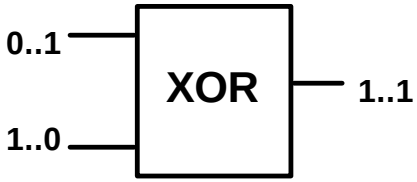
Betrachte:




(a)




(b)



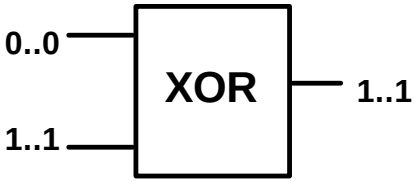

(c)

das Modell muss bei konstanten Eingängen nicht ausgeführt werden, nur wenn sich die Eingänge verändern

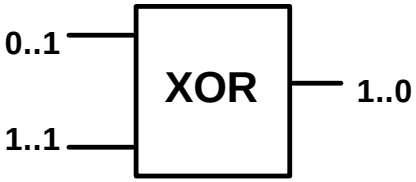
⟶ *event-driven*

# Event-Driven Simulation (Forts.)

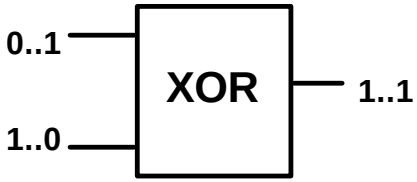
Betrachte:




(a)




(b)



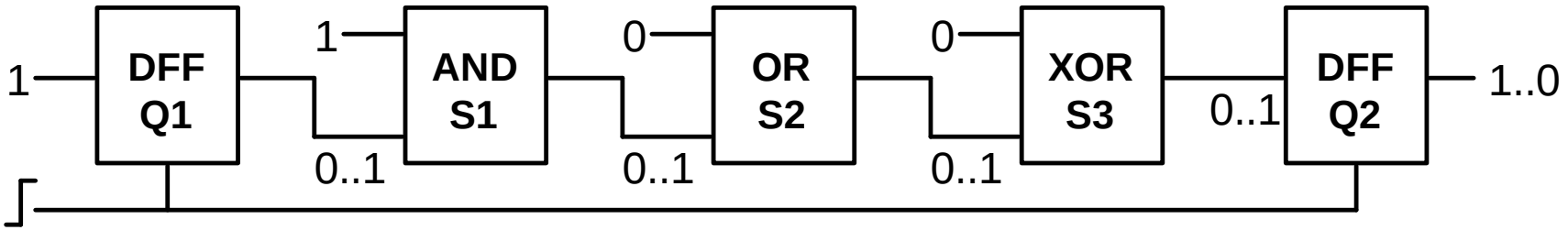

(c)

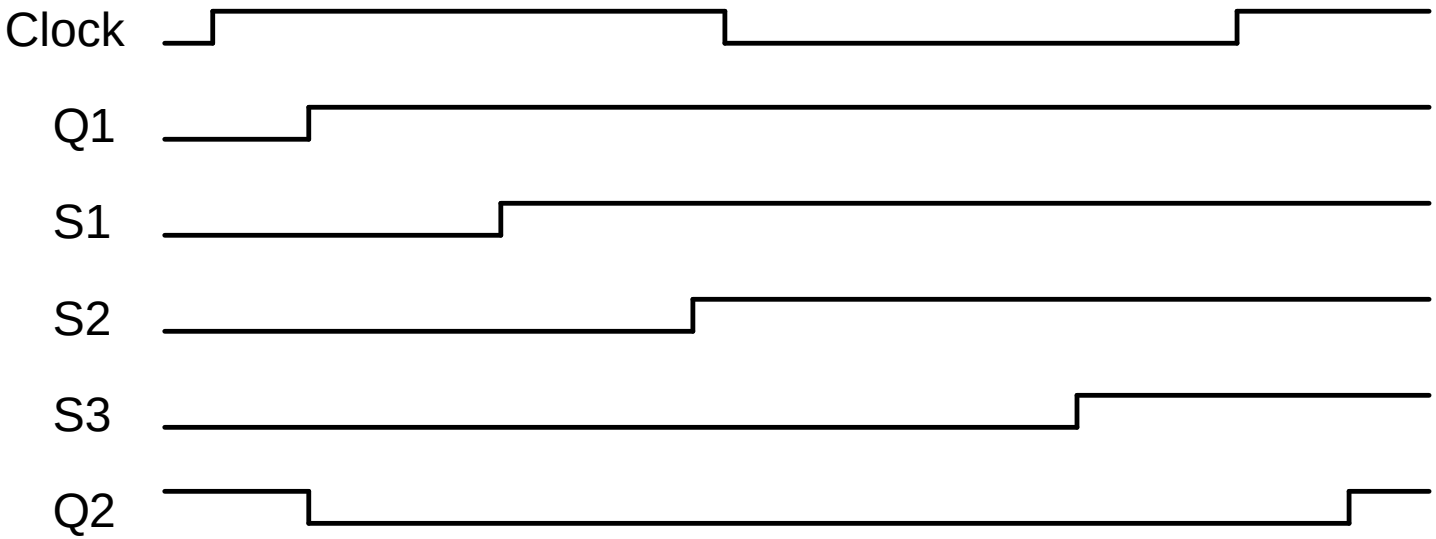
Ausführung des Modells auch im Fall (c):

- Flackern kann auftreten
- Simulator weiss nicht, was ausgeführt wird

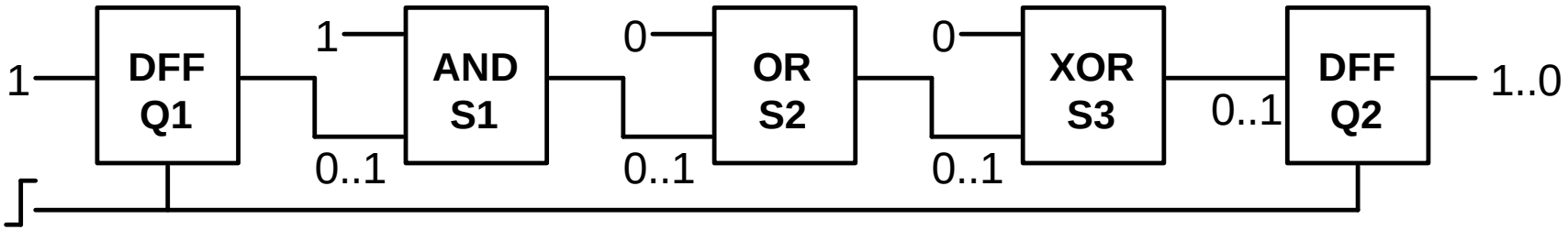
- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Cycle-Based Simulation



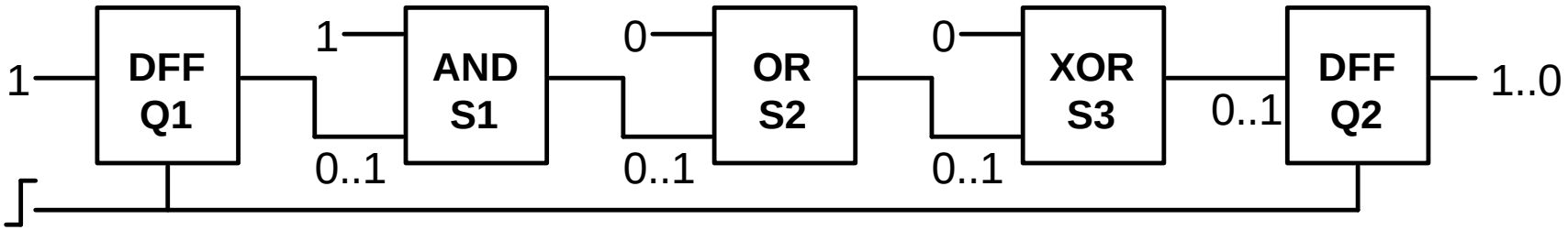



# Cycle-Based Simulation (Forts.)



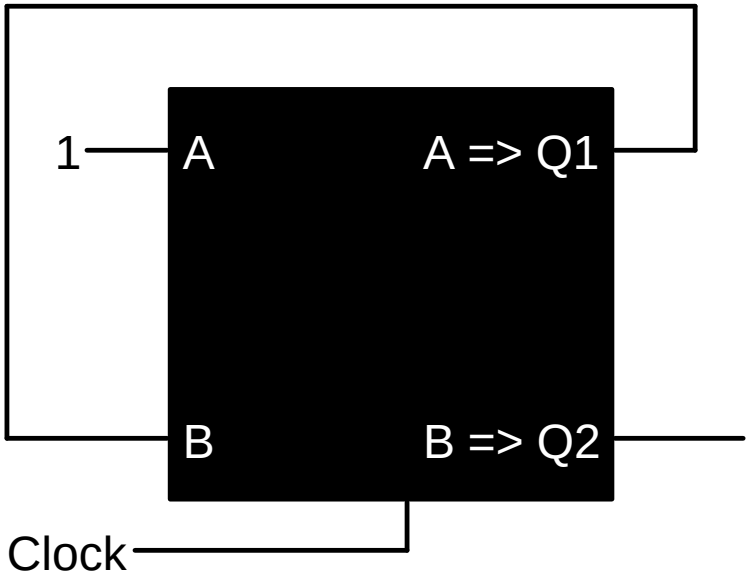

- Event-Driven: 6 Events und 7 Modelle
- Cycle-Based:
  - lediglich die steigenden Flanken der Clock wichtig
  - Simulation basiert nun auf Clock-Zyklen
  - die Beschreibung des Schaltkreises wird bei Kompilation in einen einzigen Ausdruck gewandelt

# Cycle-Based Simulation (Forts.)




```
S1 = Q1 & '1'
S2 = S1 | '0'
S3 = S2 ^ '0'

      ⟶ S3 = Q1
```



# Cycle-Based Simulation (Forts.)

- nur noch 1 Modell und 2 Events
- Verlust der Timing-Information und der Delay-Zeiten
- nur Clock-Flanken signifikant, d.h. nur perfekt synchrone Designs können simuliert werden

- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

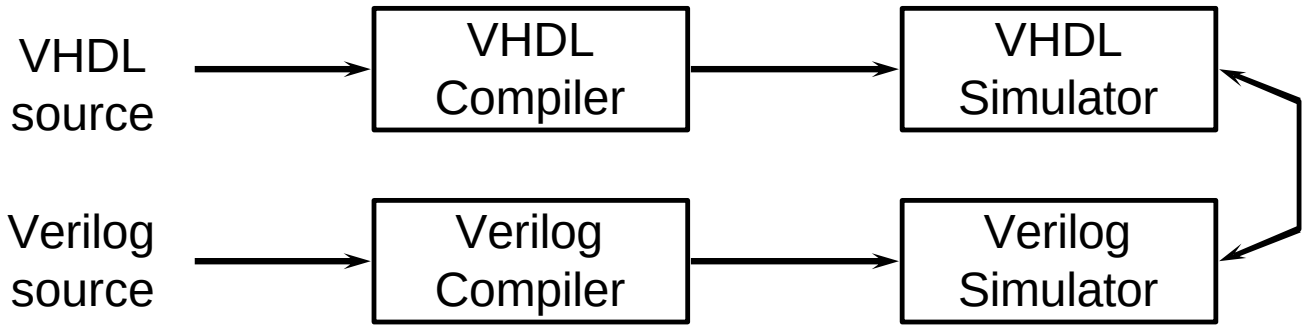
# Co-Simulatoren

- synchrone Teile ⟶ cycle-based
- der Rest ⟶ event-driven
- oder VHDL/Verilog, analog/digital, ...
- Simulation wird schrittweise gleichzeitig abgearbeitet (lock-step)
- der langsamste Simulator bestimmt die Geschwindigkeit der Co-Simulation
- mögl. Verbesserung: *time warp* synchronization

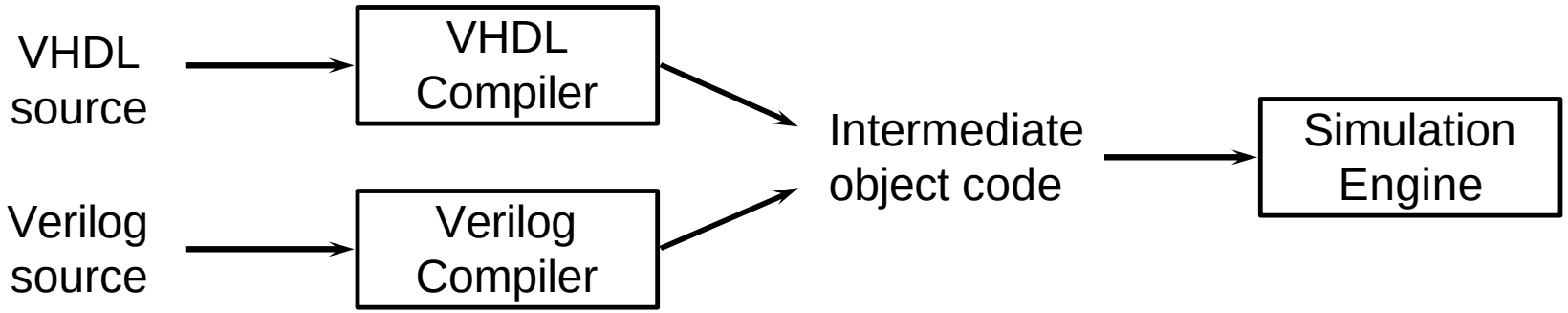
# Co-Simulatoren (Forts.)

- Leistungsminderung durch Kommunikations- und Synchronisationskosten
- Übersetzung von Werten kann zu Problemen führen, Bsp.:
  - Verilog: 128 Zustände ⟶ VHDL: 9 log. Werte
  - analoger Simulator ⟶ digitaler Simulator

# Co-Simulatoren (Forts.)

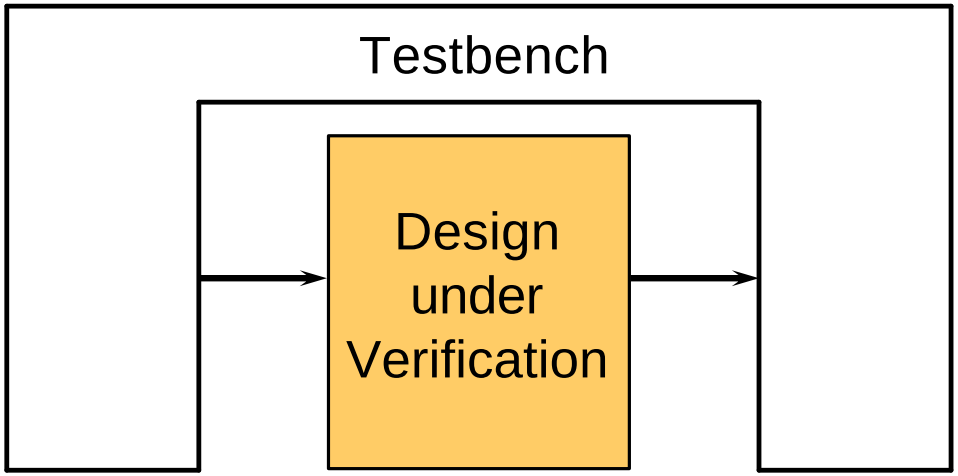



# Vgl. Mixed-Language Simulator:



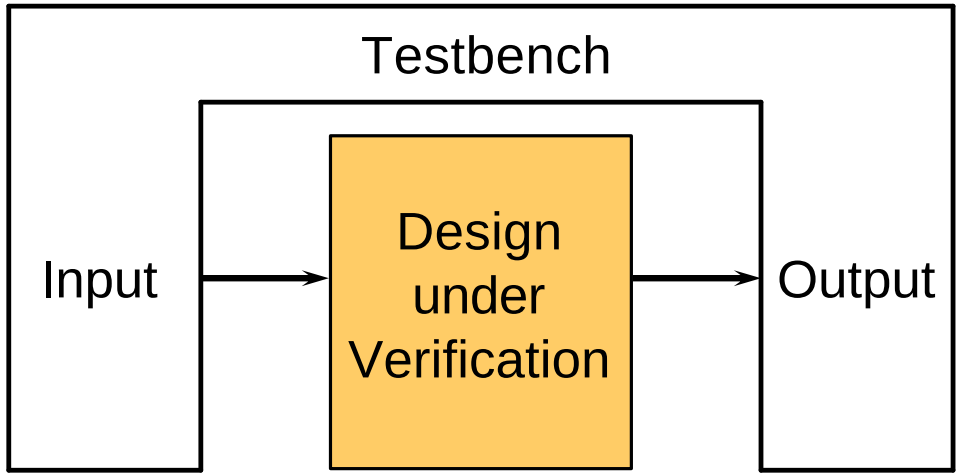
- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Testbenches




- üblicherweise in gleicher Sprache wie Design
- spezielle Verifikations-Sprachen existieren:
  - “VERA” von Synopsys
  - “e” von Verisity
- C / C++ auch mögliche Alternative

# Testbenches (Forts.)



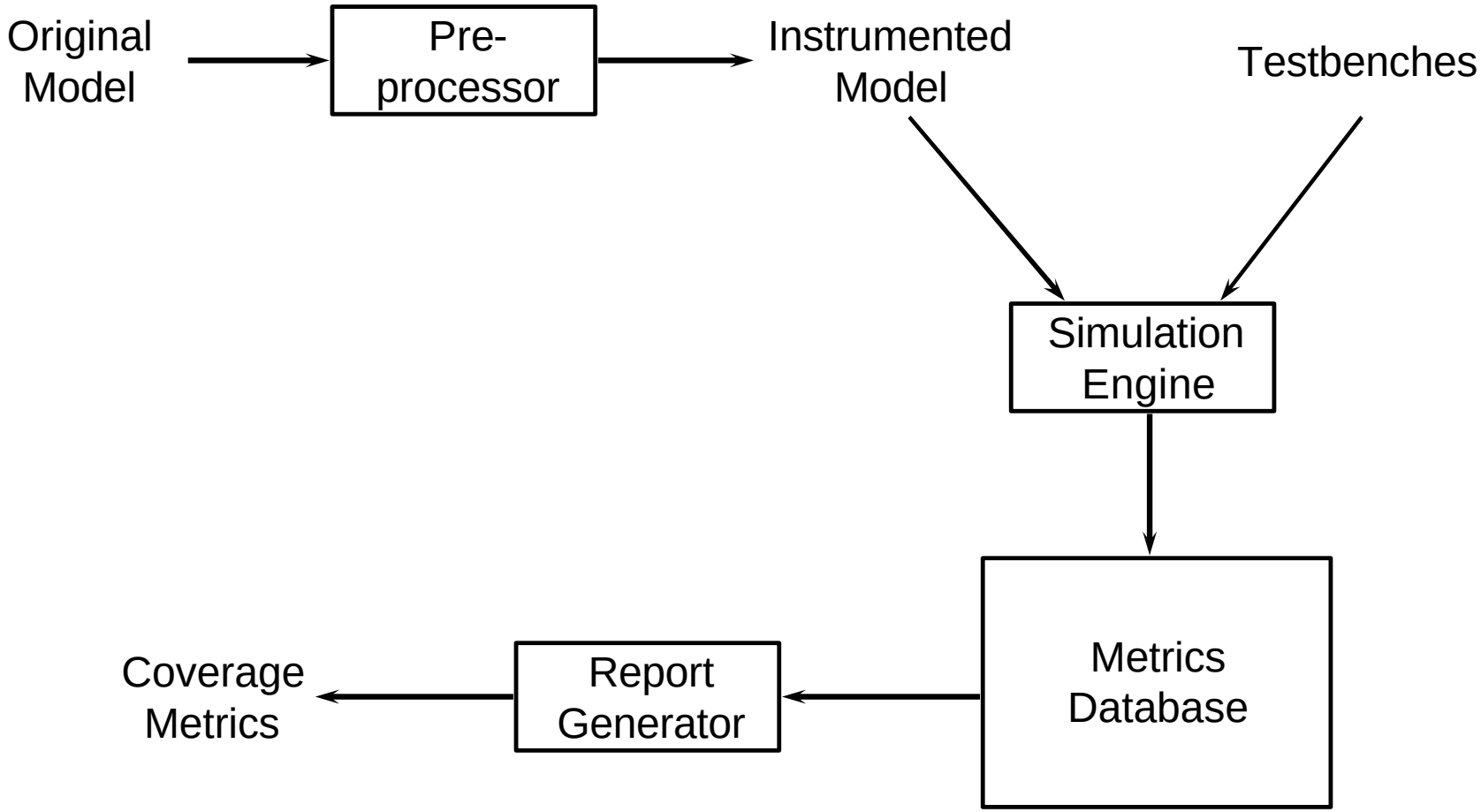

- Wann ist die Verifikation abgeschlossen?
- Woher kommen die Inputs?
- Wie überprüft man die Korrektheit der Outputs?

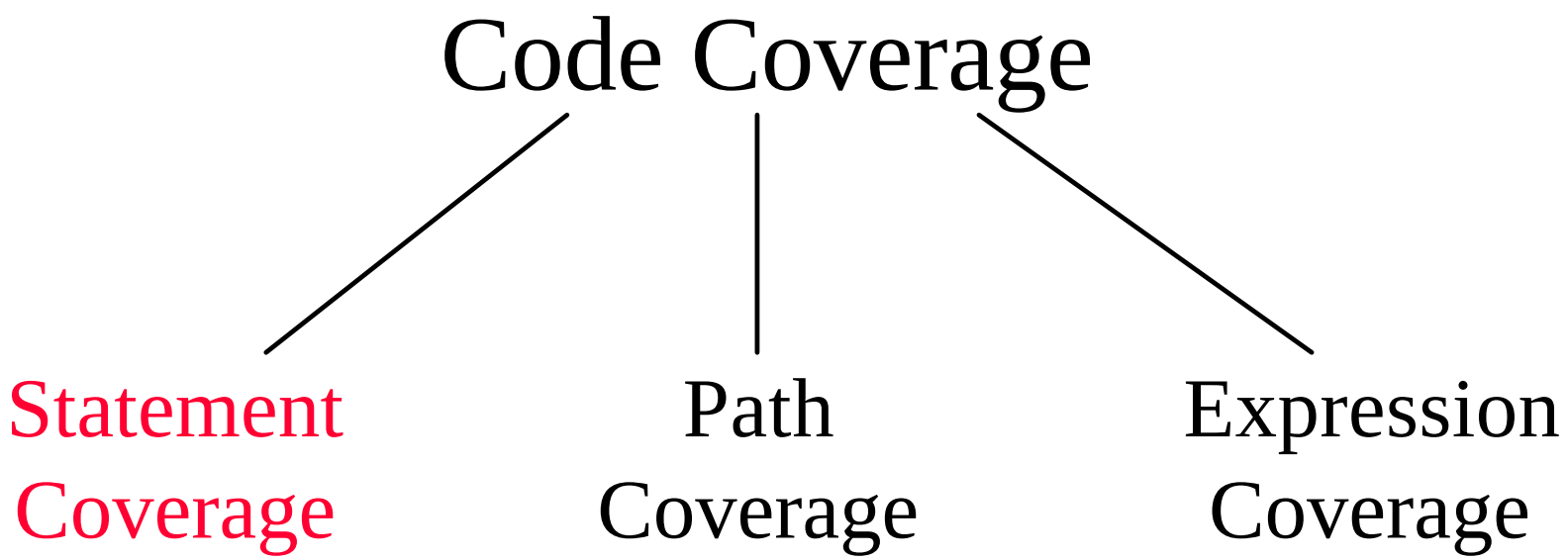
- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Code Coverage

- Source Code des Designs wird mit Checkpoints versehen (*instrumented model*)
- dieses Modell wird simuliert mit allen zur Verfügung stehenden Testbenches
- eine Datenbank sammelt die Spuren aller Simulationen
- ein Bericht wird angefertigt, der verschiedene *Metriken* (=Maß der Abdeckung) berücksichtigt
- normalerweise 100% Abdeckung angestrebt

# Code Coverage (Forts.)




Code Coverage
Statement Coverage
Path Coverage
Expression Coverage

## Statement Coverage

- misst, wieviele Zeilen des Source Codes während der Simulation ausgeführt werden
- normalerweise hilft ein einfaches GUI dem User, die nicht ausgeführten Zeilen zu identifizieren
- Frage: warum wurde dieses Statement (bzw. dieser Block) nicht ausgeführt?
- ⟶Beispiel

# Statement Coverage (Forts.)

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

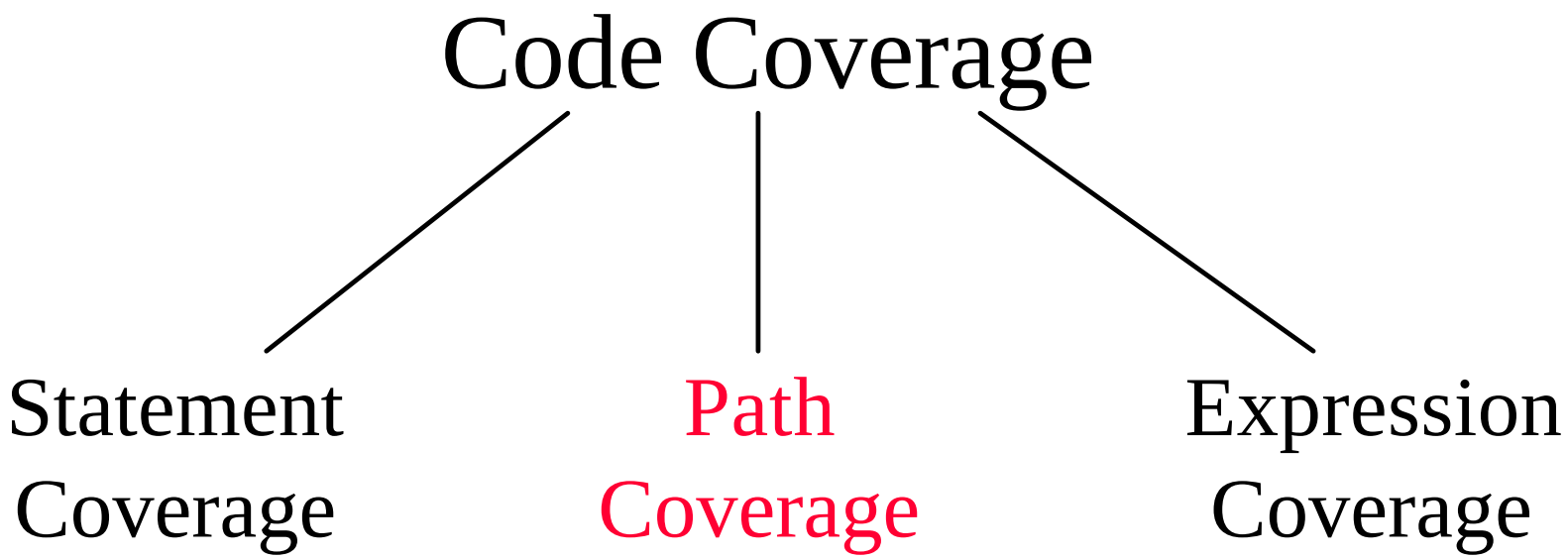
## Statement Coverage (Forts.)

- kann fraglicher Code überhaupt ausgeführt werden, oder wurde ein Testcase übersehen?
- “toter” Code kann entfernt werden

  ↳ reduziert Grösse, verbessert Maintainability
- er kann aber auch im Modell verbleiben, um Bedingungen, die nicht auftreten dürfen, zu überwachen (*defensive coding*)
- darf dann aber nicht mit die Metrik einfliessen, sondern muss gekennzeichnet werden!

# Statement Coverage (Forts.)

```
case (mode[1:0]) // synopsys full_case
2’b00: ...
2’b10: ...
2’b01: ...
// synopsys translate_off
// coverage off
default: $write(“Case was not really full!\n”);
// coverage on
// synopsys translate_on
endcase
```


Code Coverage
Statement Coverage
Path Coverage
Expression Coverage

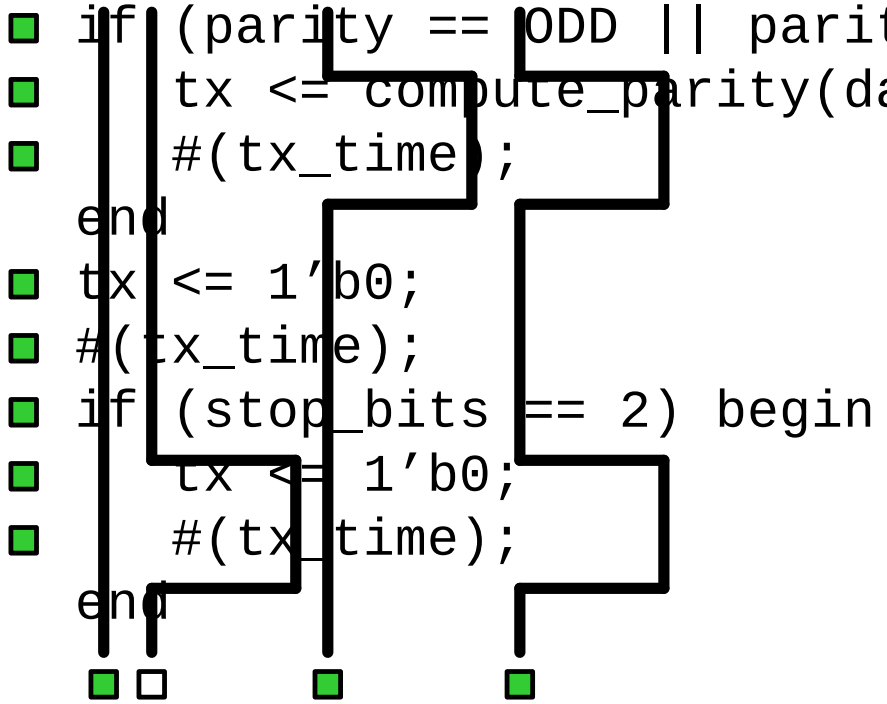
# Path Coverage

- misst alle möglichen Pfade, auf die man eine Sequenz von Befehlen abarbeiten kann:

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

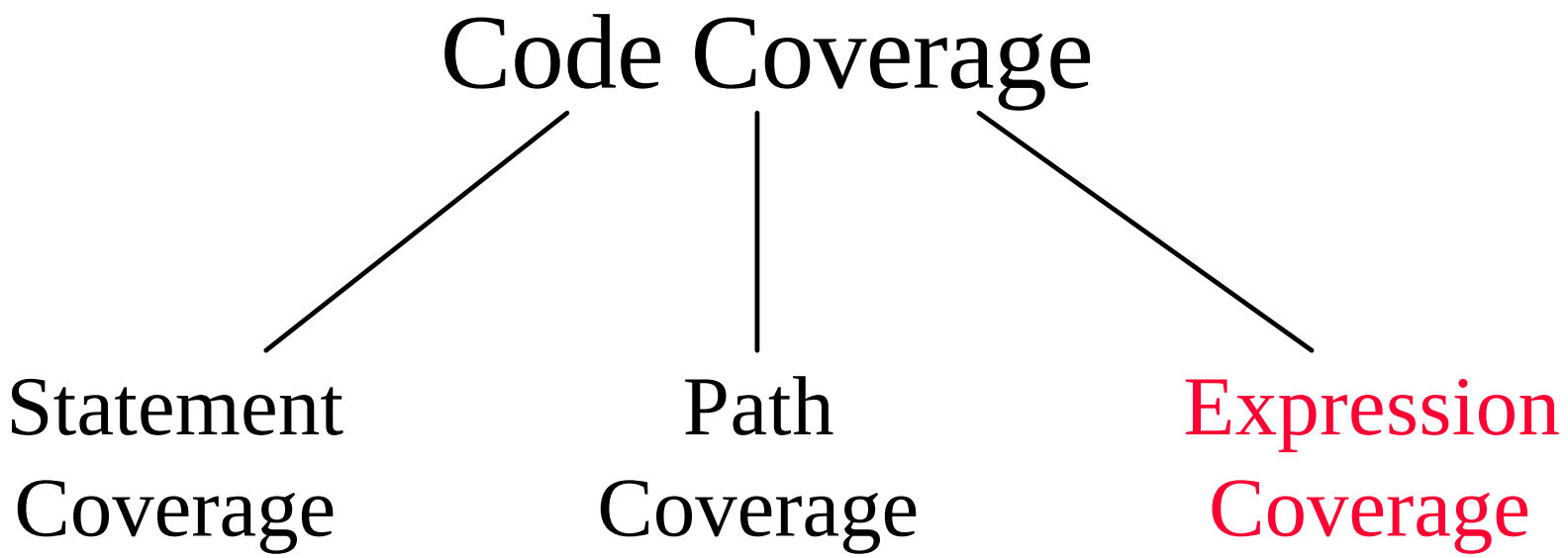
# Path Coverage (Forts.)

- misst alle möglichen Pfade, auf die man eine Sequenz von Befehlen abarbeiten kann:



# Path Coverage (Forts.)

- auch hier: Testcase übersehen? Oder kann die Bedingung gar nicht erfüllt werden?
- die Zahl der Pfade steigt exponentiell mit der Zahl der Kontroll-Bedingungen
- Code Coverage Tools hören auf, Path Coverage zu messen, wenn die Anzahl der Pfade in einem Stück Code zu gross ist
- 100% Path Coverage zu erreichen ist sehr schwierig


Code Coverage
Statement Coverage
Path Coverage
Expression Coverage

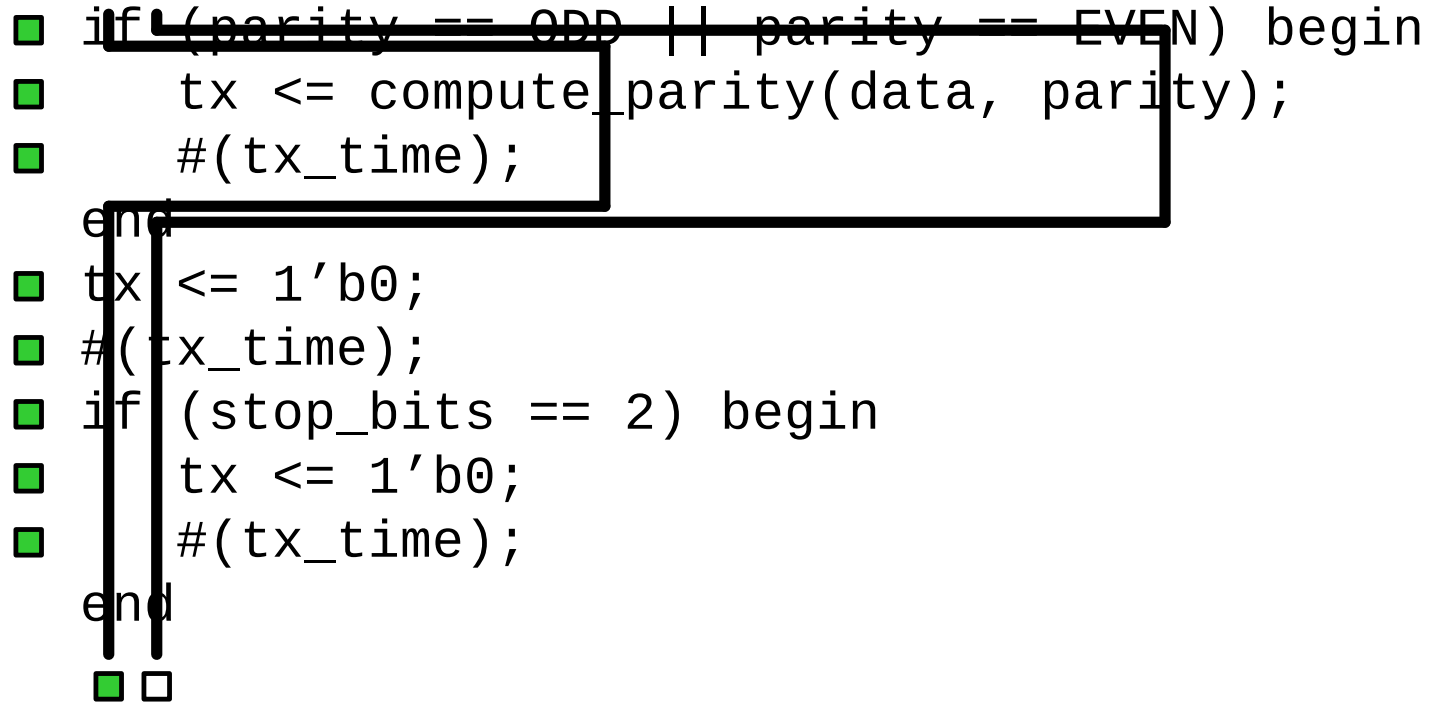
# Expression Coverage

- misst die verschiedenen Weisen, auf die Pfade ausgeführt werden können

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

# Expression Coverage (Forts.)

- misst die verschiedenen Weisen, auf die Pfade ausgeführt werden können



# Expression Coverage (Forts.)

- wieder die Frage: Testcase übersehen? Oder kann die Bedingung einfach nicht auftreten?
- 100% Expression Coverage zu erreichen ist extrem schwierig

# Was bedeutet 100% Abdeckung?

- Code Coverage zeigt nur, wie gründlich der Source Code abgearbeitet wurde
- es sagt aber nichts über die Korrektheit aus
- ein zusätzlicher Indikator für den Fortschritt der Verifikation
- niedrige Abdeckung bedeutet: Verifikation nicht abgeschlossen
- aber hohe Abdeckung bedeutet nicht, dass die Verifikation abgeschlossen ist!

## Was bedeutet 100% Abdeckung? (Forts.)

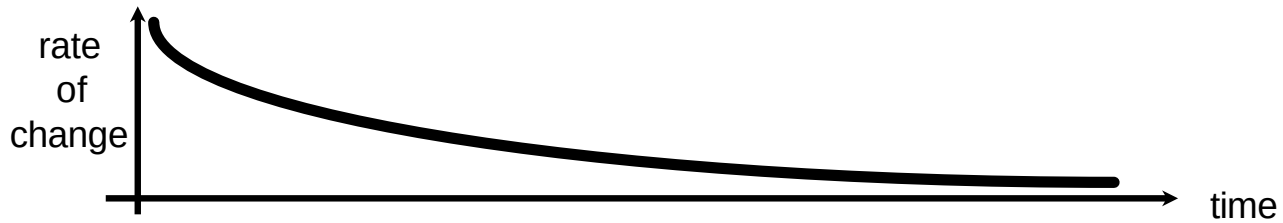
- anderer Aspekt: Benutzung von CC-Tools für *profiling*, d.h. die Zeilen herauszufinden, die am meisten abgearbeitet werden

  ↳ Performanz-Optimierung sollte hier ansetzen

- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Code-Related Metrics

- Code Coverage
- Number of lines of code
  - zur Messung des Aufwands
  - z.B. zum Vergleich von Verification Languages
- Source code changes
  - zur Kontrolle, ob ein Code stabiler läuft



# Quality-Related Metrics

- Number of known outstanding issues
  - was muss noch verändert werden?
  - werden es mehr oder weniger Aufgaben, die zu bewältigen sind?
- Number of bugs found during its service life
  - während des Gebrauchs, nicht bei der Verifikation
  - zu viele Bugs $\longrightarrow$ ein Neudesign wird nötig

- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Generieren von Stimuli

- feste Testmuster (*directed*)
  - eingebaut in die Testbench oder in einem File
  - Erstellung der Testmuster umso aufwendiger, je grösser das Modell
  - Überprüfung einfach, da man zu erwartende Ergebnisse im Vorraus berechnen kann

# Generieren von Stimuli (Forts.)

- zufällige Testmuster (*random*)
  - auch mit Hilfe von Wahrscheinlichkeiten
  - näher an der Realität
  - erwartete Werte müssen während der Simulation berechnet werden
  - dafür wird ein Referenzmodell implementiert

# Generieren von Stimuli (Forts.)

- Testbench Generation Tools
  - benutzen Metriken und erstellen dann Stimuli, um noch nicht abgearbeiteten Code zu erreichen (⟶ *Automatic/Reactive Testbench Generation*)
  - Konzentration auf Code Coverage
  - interessanter Fall: wenn das Tool keine Stimuli erstellen kann, um ein Stück Code zu erreichen

- Hardware-Design (Einleitung)
- Simulatoren
  - Simulatorarten
  - Event-Driven Simulation
  - Cycle-Based Simulation
  - Co-Simulatoren
- Testbenches
  - Code Coverage
  - Weitere Metriken
  - Generieren von Stimuli
  - Bsp.: Genetic Algorithm

# Bsp.: Genetic Algorithm

- Basis: Kodierung von potentiellen Test-Sequenzen als Bit-Matrizen von unterschiedlicher Länge
- anfangs wird eine Anzahl solcher Sequenzen zufällig generiert (⟶ Anfangs-*Population*)
- Ziel des GA: eine Evolution auf dieser Population, um deren *Fitness-Wert* zu steigern
- die Fitness-Funktion misst die Nähe einer Sequenz zum angestrebten Ziel (= best. Block)

# Bsp.: Genetic Algorithm (Forts.)

- das Ziel variiert für jeden Durchlauf des GA
- die Fitness-Funktion:
  - in der Anfangs-Phase:
    - alle Sequenzen werden gleichzeitig betrachtet
    - das Ziel ist es, eine Menge von Sequenzen *S* zu finden, die möglichst viele Statement-Blöcke erreicht/aktiviert
    - $\text{fitness}(S) = \frac{\text{activated\_blocks}(S)}{tot\_blocks}$
    - ⟶ Erkennen von Blöcken, die einfach zu erreichen sind

# Bsp.: Genetic Algorithm (Forts.)

- die Fitness-Funktion:
  - nächster Schritt:
    - der GA visiert einen bestimmten Block *T* an und das Ziel ist nun eine Sequenz zu generieren, die diesen Block abdecken kann
    - eine Sequenz *S* wird genommen, und die Blöcke, die sie erreicht (= *basic blocks*), werden gewichtet aufgrund ihrer Korrelations-Wahrscheinlichkeit mit dem gesuchten Ziel-Block
    - $\text{fitness}(S,T) = \sum_{b \in \text{covered\_bb}(S)} \text{correlation}(b,T)$

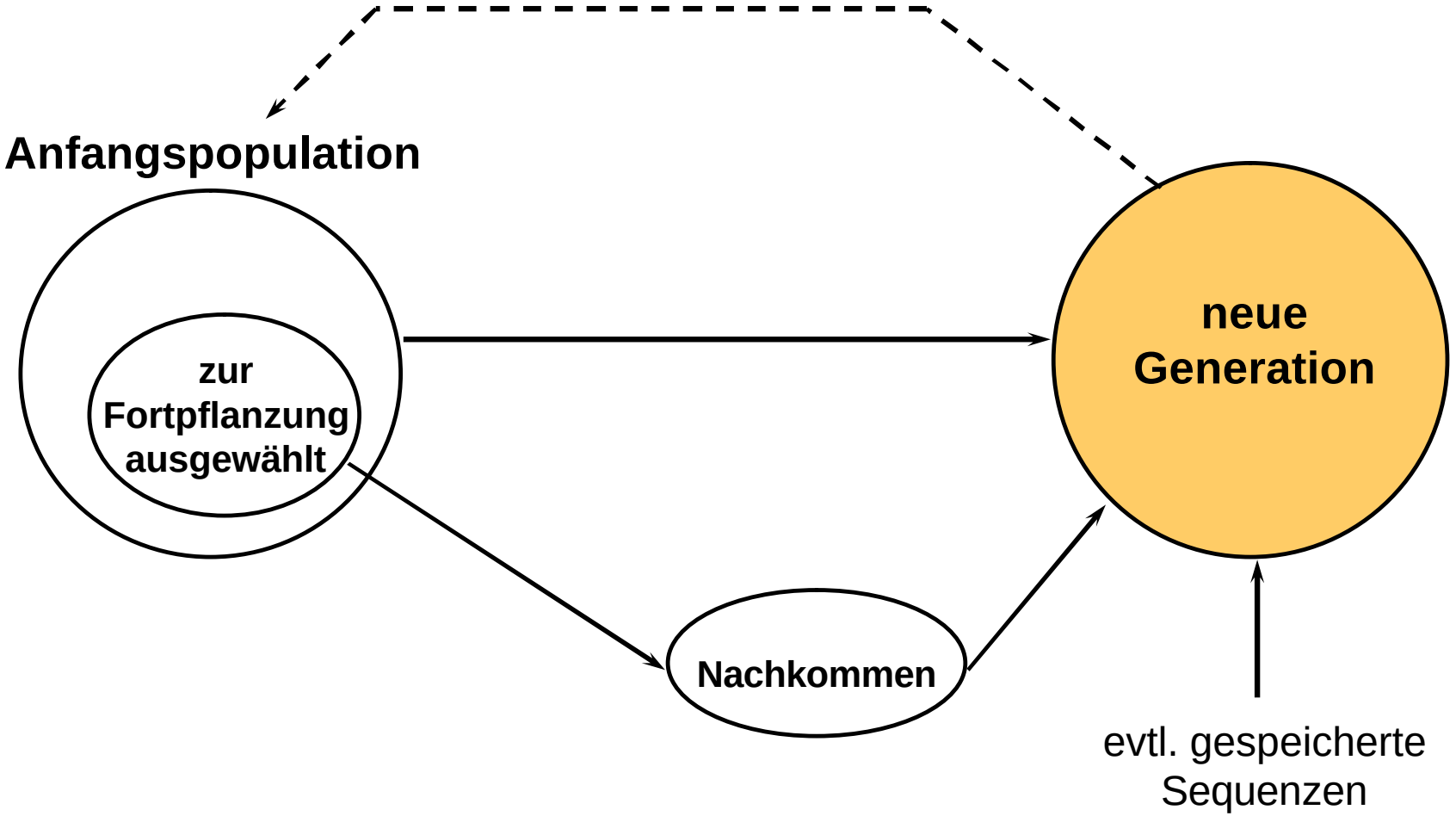
# Bsp.: Genetic Algorithm (Forts.)

- während eines jeden Durchlaufs werden auch Sequenzen erkannt, die für spätere Durchläufe nützlich sein können
- diese werden gespeichert und später in die Anfangs-Population des entsprechenden Durchlaufs eingefügt (nicht mehr als 5%)

# Bsp.: Genetic Algorithm (Forts.)

- Züchten einer neuen Generation durch:
  - *Mutation*: die originale Sequenz wird gekürzt, verlängert, oder Bits werden vertauscht
  - *Mating*: zwei Sequenzen werden benutzt zur "Fortpflanzung"
    - entweder: Anfang von S1, Ende von S2
    - oder: Spalten der Matrizen S1, S2 werden gemischt
  - Auswahl auf Grund der Fitness-Werte

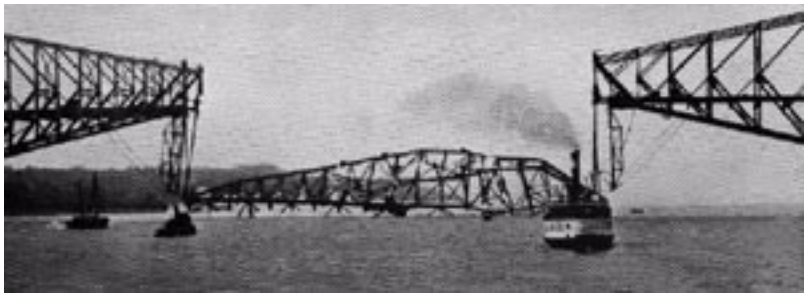
# Bsp.: Genetic Algorithm (Forts.)



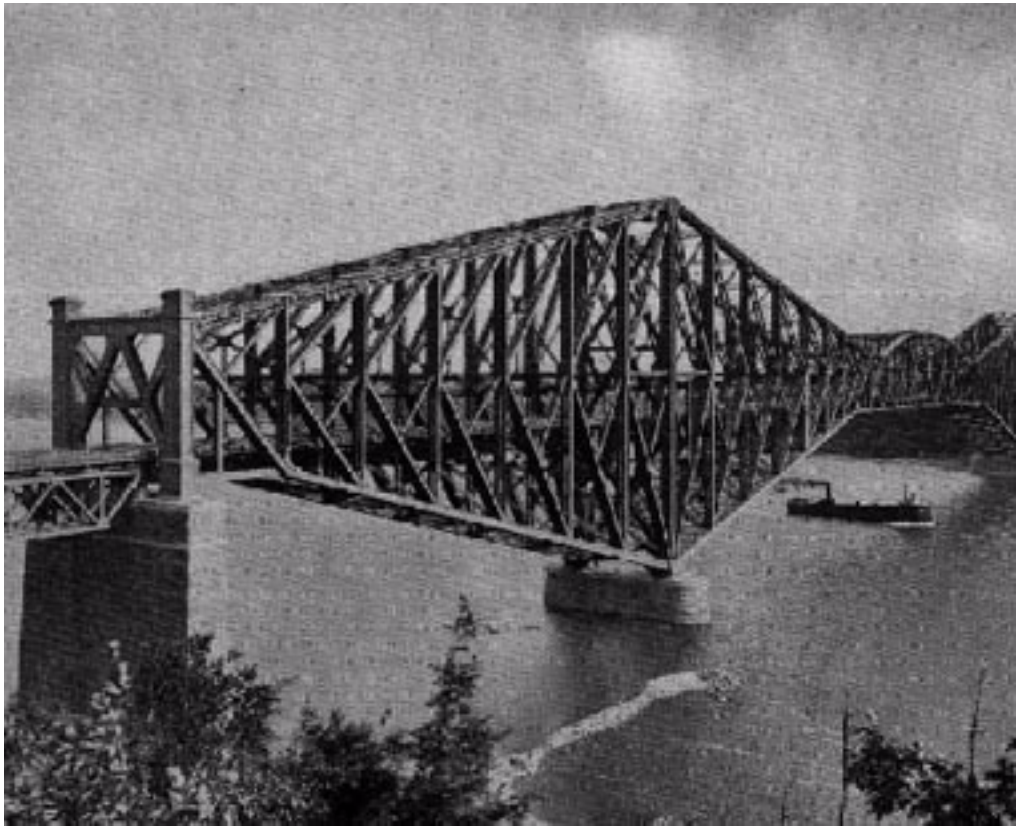
# Epilog

Die Brücke über den St.Lorenz-Strom wurde später erneut gebaut. Dieses Mal mit entsprechenden Verstärkungen.

Ironischerweise kam es beim Bau 1916 erneut zu einem Unfall, als der vorgefertigte Mittelteil der Brücke an seinen Platz gehievt werden sollte, es gab 11 Tote.



# Epilog



Die Brücke wurde 1918 endgültig fertiggestellt. Sie wird heute noch genutzt und ist immer noch die grösste Auslegerbrücke der Welt.

# Quellen und Literaturhinweise


- “Writing Testbenches”, Janick Bergeron. Kluwer Academic Publishers. 2000. [http://www.janick.bergeron.com]
- “Hardwaremodellierung”, Christian Siemers. Hauser. 1999.
- Skript zur Vorlesung “Hardwareverifikation”, Jürgen Ruf und Thomas Kropf. 2001. [http://www-ti.informatik.uni-tuebingen.de/~ruf/vorlesung]
- “Automatic Validation of Protocol Interfaces Described in VHDL”, F. Corno, M.S. Reorda, G. Squillero. 2000. [http://www.cad.polito.it/pap/db/evotel2000a.pdf]
- http://www.synopsys.com/products/vera
- http://www.verisity.com
- “The First Quebec Bridge Disaster - A Case Study” [http://www.civeng.carleton.ca/ECL/reports/ECL270/Introduction.html]