Die Hardwarebeschreibungssprache VHDL

***V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage*

(not programming)

Warum Hardwarebeschreibungssprachen?

- **Veränderung der Industriellen Entwurfsprozesse**
  - **Anstieg der Entwurfskomplexität**
  - **Verkürzte Produktzyklen, Reduzierte Time-To-Market**

→ **Grafische Schaltplaneingabe auf Logikelementebene reicht nicht aus**

  - **Wenig Abstraktionsmöglichkeiten**
  - **Umständliche Konvertierung zwischen Abstraktionsebenen**
  - **Aufwendige Dateneingabe**

Einsatz von Hochsprachen

- Hochsprachen erlauben
  - eindeutigen Beschreibung von Hardware (Spezifikation)
  - Simulation der Spezifikation (Validierung)
  - automatisierte Erzeugung von Hardware (Synthese)

- außerdem
  - Abstraktion von Implementierungsdetails
    - Verschiedene Abstraktionsniveaus möglich
    - technologieunabhängig
    - wiederverwendbar
  - lesbar von Mensch und Maschine
  - Dokumentation des Entwurfs
  - Kommunikation im Projektteam
  - Text basierte Versionierung möglich (GIT, SVN, CVS…)
- Typische Sprachen: VHDL, Verilog, (SystemC)

Historie

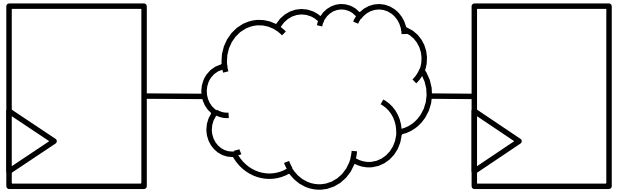
- 1980: Very High Speed Integrated Circuit (VHSIC) Projekt des amerikanischen Verteidigungsministeriums (DoD)
- 1983-85: Entwicklung einer einheitlichen Sprache
  - im Auftrag des DoD
  - Name: VHSIC Hardware Description Language = VHDL
- 1985: Veröffentlichung des ersten VHDL-Standards
- 1987: VHDL wird zum IEEE-Standard (IEEE 1076-1987)
- 1992-heute: Erweiterungen des Sprachumfangs
  - Objective-VHDL
  - mixed-signal-VHDL
  - etc.
- 1993: IEEE-Standard Update (IEEE 1076-1987)
- 2002: Kleineres Update des Standards
- 2008: Erweiterungen des Standards (IEEE 1076-2008)
  - Erweiterungen zur besseren Parametrisierbarkeit

Allgemeines zu VHDL

- **Beschreibungssprache für digitale Schaltungen und Systeme**
- **Sequentielle und parallele Abläufe**
- **Asynchroner und synchroner Entwurf**
  - **Asynchron: Kombinatorische Logik**
  - **Synchron: Sequentielle getaktete Logik**
- **Beschreibungsarten:**
  - **Strukturbeschreibung**
  - **Verhaltensbeschreibung**
    - **Verschiedene Abstrakationsebenen möglich (nicht alle synthesefähig)**
- **Simulation auf allen Abstraktionsebenen**

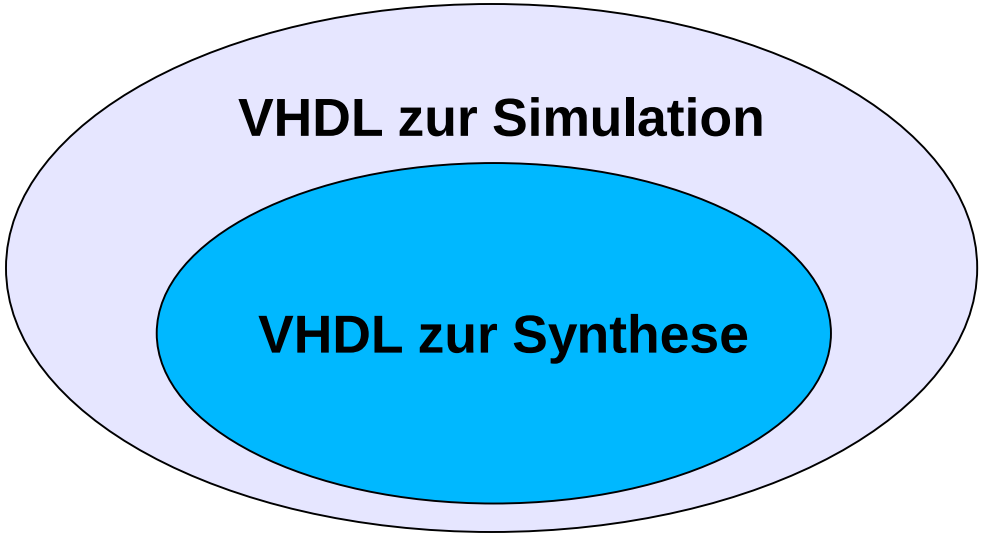
Synthese und Simulation

- **Synthesemodell**
  - **Asynchrone Logik**
  - **“Register-Transfer-Level” (RTL)**
    - **Register mit dazwischenliegender Logik**
  - **Synthetisierbar**



- **Simulationsmodell zusätzlich:**
  - **Modellierung von Zeit**
  - **Datentypen**
  - **Operatoren**
  - **Simulationsausgaben**

- **VHDL-Sprachumfang:**



Entity und Architecture

- Entity (Einheit) definiert die Schnittstellen (Ports) eines Entwurfs
  - Name
  - Mode: in, out
  - Typ: std_logic, std_logic_vector, signed, unsigned, ...
- Architecture beschreibt interne Details einer Entity
  - Verhalten oder interne Struktur
  - Jede Entity muss mindestens eine Architecture haben

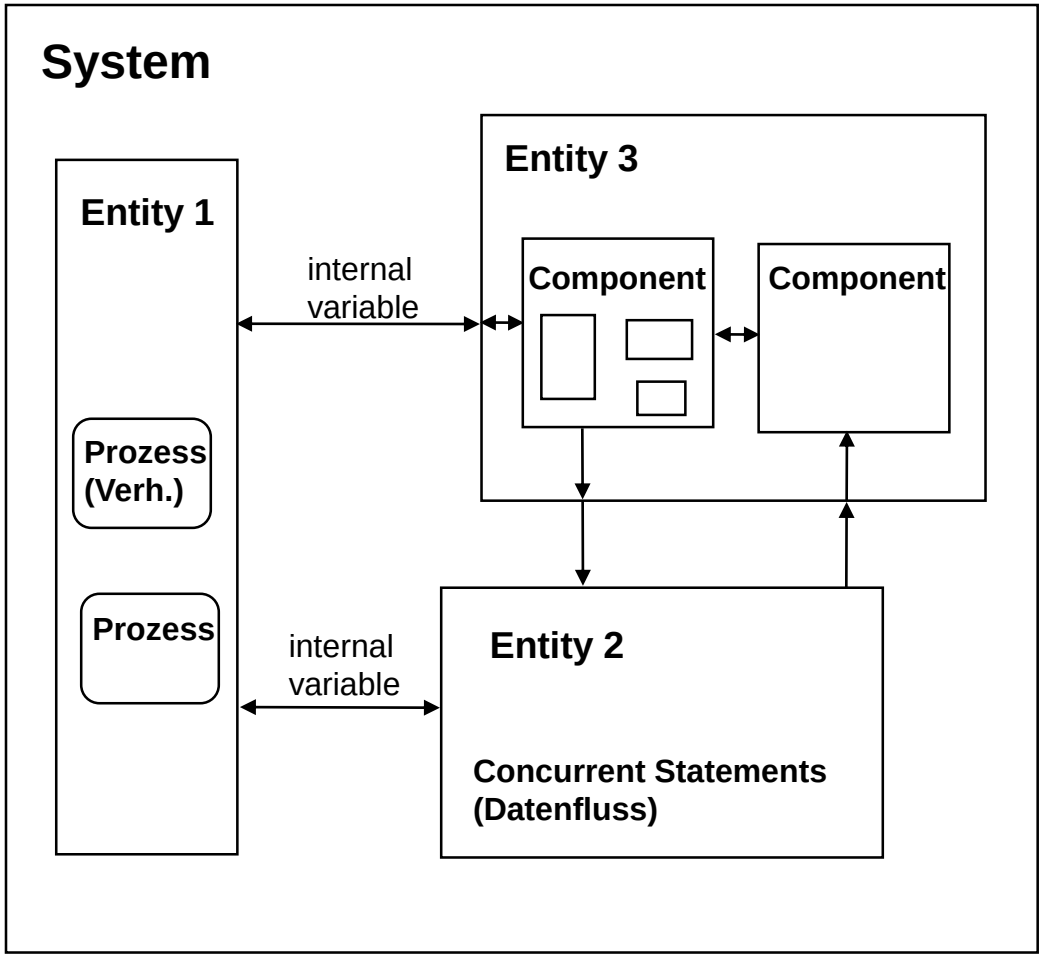
```
entity half_adder is
  port (a, b: in std_logic; s, c: out std_logic);
end;

architecture ARC of half_adder is
begin
-- description of half-adder
end;
```

Sprachelemente

- `Entity` und `Architecture`
- **Beschreibungsformen**
  - **Verhalten:** `Process` **mit Sequential Statements**
  - **Struktur:** `Component`
  - **Datenfluß: Concurrent Statements**
- **Kommunikation**
- **Datentypen**
- **Operatoren**

Strukturbeschreibung

- Gemischte Beschreibungen
- heterogene Hierarchie
  - Verhalten
  - Datenfluss über interne Variablen
- verschiedene Abstraktionsebenen
- Entities können als Components instantiiert werden



Verhaltensbeschreibung mit Prozessen

- Prozesse sind eine (häufig eingesetzte) Möglichkeit das Verhalten einer `Architecture` zu beschreiben
- Jede `Architecture` kann beliebig viele Prozesse enthalten
- Prozesse arbeiten parallel, jeder einzelne sequenziell
- Prozesse kommunizieren über lokale Signale innerhalb der Architecture
- VHDL-`Process`
  - sequentielle Abarbeitung
  - Aktivierung:
    - Parameter der sensitivity list hier: a und b

```
architecture ARC_1 of half_adder is begin
  process(a, b) begin
    if (a=b) then
       s <= 0;
    else
       s <= 1;
    end if;
    if (a=`1` and b=`1`) then
       c <= 1;
    else
       c <= 0;
    end if;
  end process;
end;
```

Verhaltensbeschreibung mit Concurrent Statements

- Concurrent Statements können parallel zu den Prozessen das interne Verhalten einer `Architecture` beschreiben.
- Als Sensitivitätsliste dienen dabei alle abhängigen Variablen eines Concurrent Statements.
- Somit wird in der `Architecture` kombinatorische Logik erzeugt.
- Direkte Verdrahtung
- Jede Änderung einer abhängigen Variable führt unmittelbar zu einer Neuauswertung des Statements
  - Sensitivität auf a und b (vgl. vorherige Folie) → gleiches Verhalten
- Bedingte (`when`) und selektive (`select`) Signalzuweisungen
- Kein Synchronisierung zur Clock!

```
architecture ARC_2 of half_adder is
begin
  s <= a xor b;
  c <= "1" when (a and b) else "0";
end ARC;
```

Strukturbeschreibung mit Komponenten

- **bereits vorhandene Entities können als Komponenten wiederbenutzt werden (Komponenten-Bibliothek)**
- **je Architecture:**
  - **eine Deklaration**
  - **beliebig viele Instanzen**

```
use work.all; -- benutze gesamte bibliothek

entity half_adder is
    port (a, b: in std_logic; s, c: out std_logic);
end;

architecture ARC_3 of half_adder is
 component AND2
    port (I1, I2:in std_logic; O1: out std_logic);
 end;
 component XOR2
    port (I1, I2:in std_logic; O1: out std_logic);
 end;

begin
  C_1: XOR2 port map (a,b,s);--concurrent statement
  C_2: AND2 port map (O1=>c, I1=>a, I2>=b);
end ARC;
```

Deklaration der verwendeten Sub-Komponenten

Diese **architecture** enthält keinen Prozess, sondern nur eine Struktur-beschreibung. Instanzen **C_1** und **C_2**

Sequential Statements (clk-synchrone Architektur)

- Nur innerhalb von `process` nutzbar
- Bedingungen
  - `IF THEN ELSE`
  - `CASE`
- Schleifen
  - `for loop`
  - `while loop`
- Haltepunkte
  - `wait until CLK'event`
  - `wait until CLK="1"`
  - `bzw. rising_edge(clk)`
- (Funktionen und Prozeduren)
- Nicht synthetisierbar:
  - `wait for 50 ns`

```
architecture ARC of sum is
begin
  process(clk)
    variable i, aux: integer;
  begin
    if(rising_edge(clk) then
      aux := 0;
      if (n/=0) then
        for i in 1 to n loop
          aux := aux + i;
        end loop;
      else
      aux := -1;
        end if;
      result <= aux;
    end if;
  end process;
end;
```

Kommunikation in VHDL

- `Signal`
  - globale Kommunikation zwischen Entities, Components und Prozessen
  - dürfen nur einen "Treiber" haben
  - Ein- und Ausgänge von Entities sind automatisch als Signale bekannt
  - *Entspricht einem Draht in einer Schaltung*
  - *Kann einem Register in einer Schaltung entsprechen*
- `Variable`
  - Nur innerhalb von Prozessen
  - Repräsentieren kombinatorische Logik
  - Ziel von Zuweisungen
- `Constant`
  - wie Variable mit konstantem Inhalt
  - *Entspricht VCC oder GND in einer Schaltung*

Bockierende und nicht-blockierende Zuweisungen

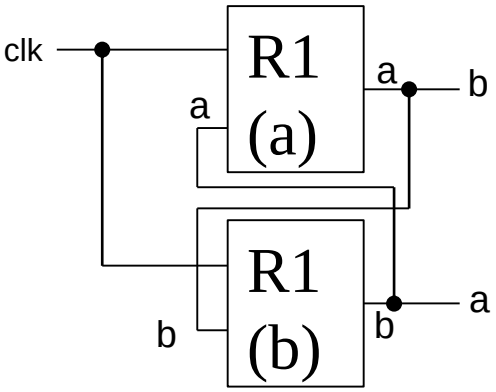
- **Blockierende Zuweisungen (auf Variable) werden sofort ausgeführt.**
- **Sind nur innerhalb eines process-Blocks erlaubt.**
- **Erzeugen kombinatorische Logik, keine Register! a und b sind Prozess interne Variablen**

```
a = 42          a := b;          a = 1337
b = 1337   ⇒    b := a;    ⇒     b = 1337
```

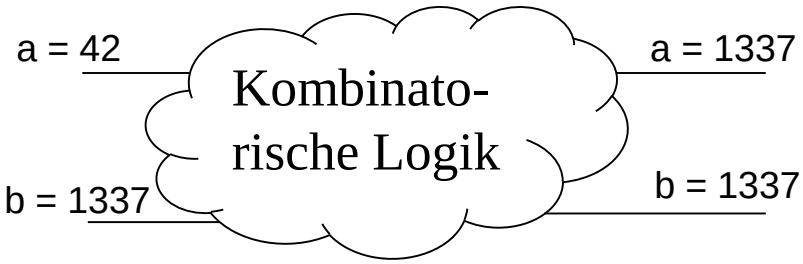
- **Nicht-blockierende Zuweisungen (auf Signale) werden erst am Ende eines Prozesses ausgeführt.**

```
a = 42          a <= b;          a = 1337
b = 1337   ⇒    b <= a;    ⇒     b = 42
```

Signale



Interne Variablen

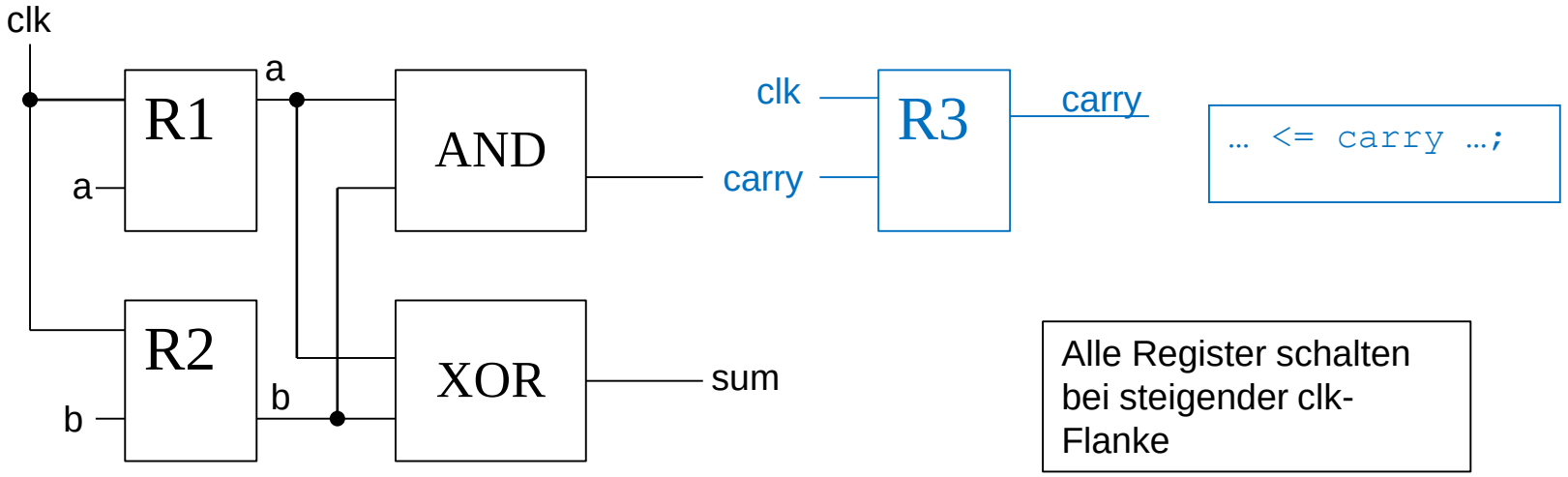


Synthese Beispiel

- Clocksynchroner Prozess `half_adder`
- Zur Erinnerung:

```
s <= a xor b;
c <= a and b;
```

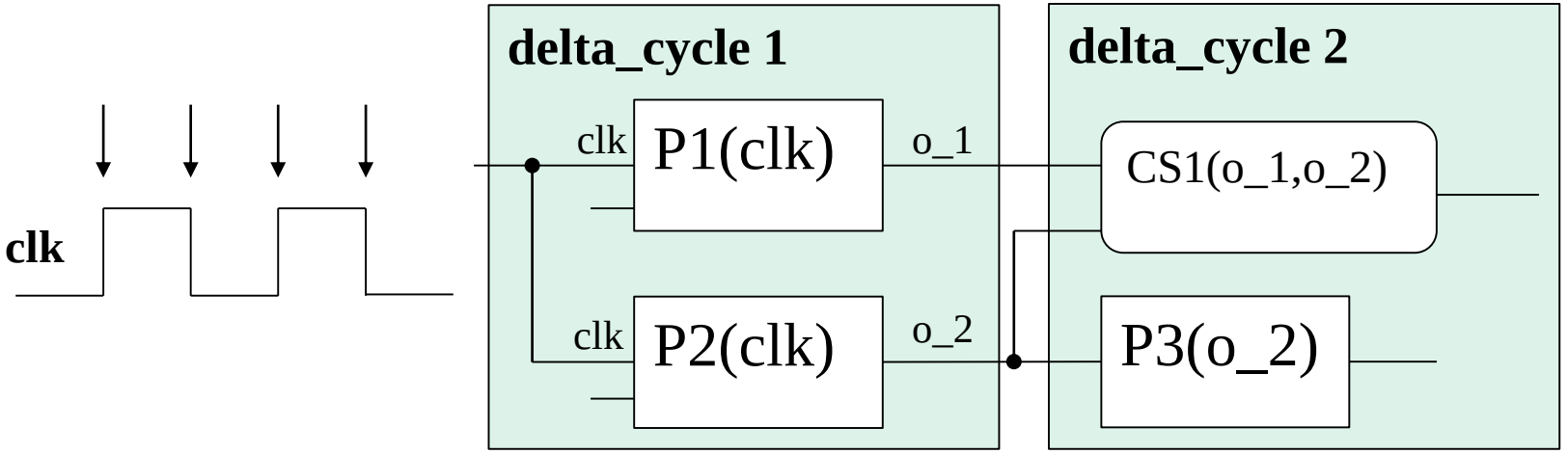
- Bei `rising_edge(clk)` werden die aktuellen Werte von `a` und `b` an den Registern `R1` und `R2` übernommen und ausgewertet.
- In einem weiteren (nebenläufigen) Prozess kann dann z.B. carry als Eingang verwendet werden, was dort wiederum das Register R3 erzeugt.



- Grundsätzlich bedeutet Sensitivität, dass die Register entsprechend der Sensitivitätsliste des Prozesses die Werte übernehmen.

Simulation und delta cycles

- Die Simulation einer beschriebenen Schaltung basiert auf **delta cycles**.
- Jede Änderung eines Schaltungseingangs (z.B. clk) ist ein **Event zu einer bestimmten Zeit**.
- Alle **Prozesse und concurrent statements** die auf diesen Eingang sensitiv sind werden in eine Liste geschrieben und **parallel abgeabreitet**.
- Das geschieht in $\Delta t = 0s$ und repräsentiert einen delta cycle.
- Danach werden alle Prozesse und concurrent statements in eine Liste geschrieben, die sensitiv auf geänderte Ausgänge des vorherigen cycles sind und im nächsten **delta cycle** abgearbeitet.
- Dies wird so lange wiederholt, bis die Liste leer bleibt, anschließend springt die Simulationszeit weiter.



Synchroner und asynchroner reset

- **Ein asynchroner reset wird unmittelbar in der Schaltung wirksam.**
- **Kritisch: Undefinierter Zustand, falls der reset zu nah an der nächsten steigenden clock-Flanke ausgelöst wird:**

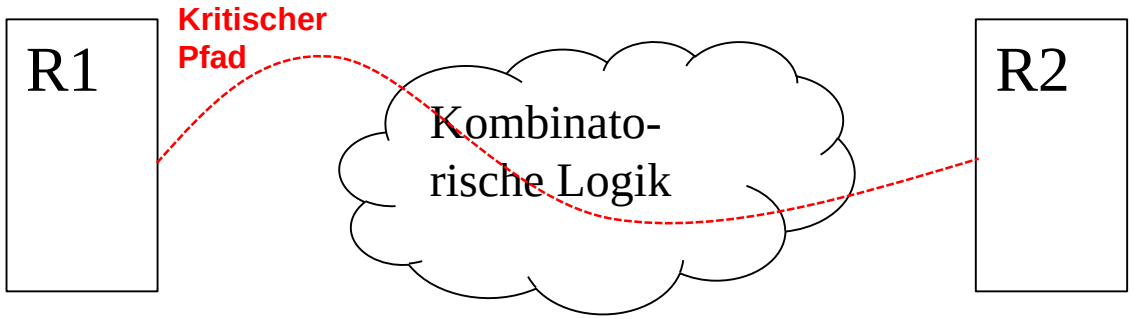
```
process(clk, ncl)
 begin
  if(ncl = '0')then
    -- do reset
  elsif(rising_edge(clk)) then
    -- do regular operation
  end if; end process;
```

- **Ein synchroner reset wird erst zur nächsten steigenden clock-Flanke wirksam.**

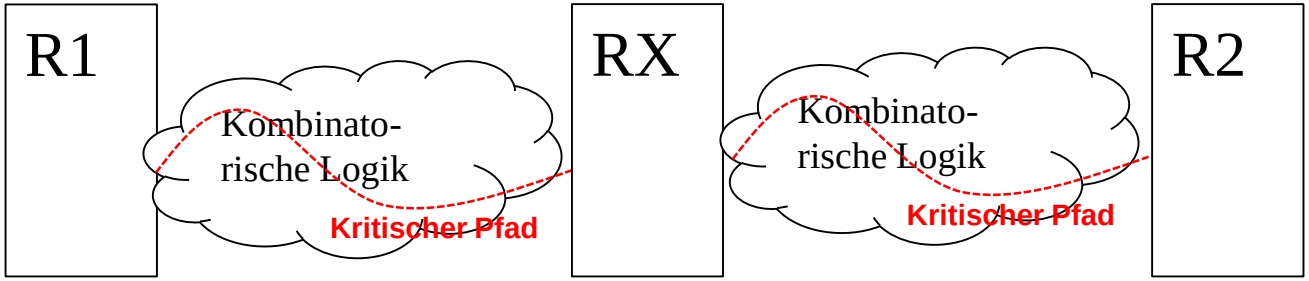
```
process(clk)
 begin
  if(rising_edge(clk) then
    if(reset) then
       -- do reset
    else
       -- do regular operation
    end if;
  end if;
end process;
```

Simulationszeit und kritischer Pfad

- **Der kritische Pfad ist der längste Pfad zwischen zwei Registerstufen in einer clock-Domäne**



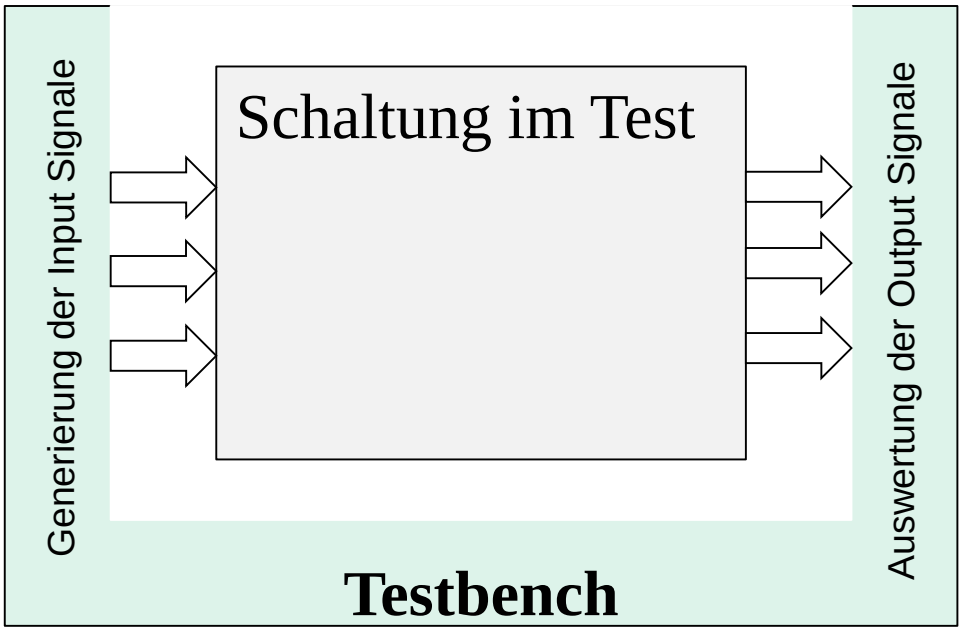
- **Ein langer kritischer Pfad senkt die maximal mögliche clk-Frequenz**
- **Aufteilung des kritischen Pfades (hier: 1ne zusätzliche Registerstufe)**
  - **Doppelte Anzahl an nötigen clk-Takte zur Berechnung**
  - **Aber: Steigerung der Performanz für die gesamte restliche Schaltung**



- **Die Ermittlung des kritischen Pfades folgt der Simulation in einer Logiksynthese (Gatternetzliste + Technologiebibliothek)**

Testbench

- **Eine Testbench simuliert die Schaltungumgebung**
  - **Eingänge werden generiert (z.B. clock und weitere Signale)**
  - **Ausgänge werden evaluiert (mittels assert Statements)**
- **Clock Events aus der Testbench starten dann z.B. alle 50ns einen delta cycle.**



Spezielle Simulationskonstrukte

- **Modellierung von Verzögerungen**
  - **in Prozessen**

```
process
  -- some sequential statements
  wait for 20 ns;
  -- some other sequential statements
end process;
```

  - **in Zuweisungen**

```
result <= `1` after 10 ns;
```

- **Funktionsgeneratoren**

```
wave <= `0`, `1` after 5ns, `0` after ...
```

- **Überwachungsfunktionen**

```
assert ((NOW - LastEventOnCLK) <= HOLD_TIME)
  report "hold time too short!"
  severity WARNING;
```

Datentypen

- **Skalare**
  - **Aufzählungen (z.B. std_logic, boolean)**
  - **Integer**
  - **Gleitkomma**
  - **Physikalisch (Zeit, Spannung, etc.)**
- **Feldtypen**
  - **array (z.B. std_logic_vector)**
  - **Record-Typen**
- **Datei**

Nicht alle Typen sind synthetisierbar

Operatoren

- **Logische Operatoren**
  - `and, or, nand, nor, xor, not`
- **Vergleichsoperatoren**
  - `=, /=` (ungleich), `<, >, <=, >=`
- **Additionsoperatoren**
  - `+, -, &` (Konkatenation)
- **Multiplikationsoperatoren**
  - `*, /, mod` (Modulo), `rem` (Remainder)
- **Sonstige**
  - `abs` (Betrag), `**` (Potenz)

Nicht alle Operanden sind für Synthese geeignet