Hardware Synthese mit

VHDL

Thomas Schanz, Christoph Tenzer

IAAT - Universität Tübingen

September 2004

Vorwort

Dieses Dokument entstand am Rande der wissenschaftlichen Arbeit der Verfasser am IAAT der Universität Tübingen. Es ist an alle diejenigen gerichtet, die die Technik der Hardware-Synthese anhand der Beschreibungsspache VHDL zu erlernen suchen. Das Augenmerk liegt dabei besonders auf einer anschaulichen Darstellung der Grundkonzepte und der Struktur von VHDL (Kapitel 1). Es ist gedacht für den Neueinsteiger in dieses interessante und aktuelle Gebiet der Elektronik. Anhand von Beispielen werden die Strukturen von VHDL erläutert und die wichtigsten Sprachelemente erarbeitet (Kapitel 2). Dieses Dokument ist keine Anleitung zu VHDL! Hierfür verweisen wir auf die Spezialliteratur (z.B. [Lehmann et al. 1994], [Ashenden 2002] usw..).

Der vollständige Weg der Hardwaresynthese beinhaltet neben dem **Design-Entry**, der Schaltungsbeschreibung mit VHDL, auch die **Technologieabbildung** mittels Syntheseprogrammen (Kapitel 4). Am Schluss steht die **Programmirung** der Ziel-Hardware und der Test mit elektronischen Meßgeräten. Aus diesem Grund empfehlen wir begleitend zum Studium dieses Dokuments und der Literatur zu VHDL, auch die Anschaffung des 'picoMAX DIGILAB'-Prototyping-Boards von El Camino (vgl. Kapitel 3) und die Installation der Synthese-Software 'Quartus II' von Altera (siehe. Kapitel 4).

Eine besondere Motivation war es, die Praxistauglichkeit von VHDL aufzuzeigen. VHDL ist nicht eine alltagsferne abstrakte akademische Sprache, sondern ein leistungsfähiges Werkzeug für die Elektronikentwicklung. Kapitel 5 zeigt deshalb zwei einsatzfähige VHDL-Entwürfe: eine Ampelschaltung (z.B.: für den Modellbau oder die Eisenbahnanlage) und einen 10 MHz Frequenzzähler für den Laboreinsatz.

Die Verfasser.

email:
schanz@astro.uni-tuebingen.de
tenzer@astro.uni-tuebingen.de

Hardware Synthese

Ziel der Hardware Synthese ist die möglichst schnelle und kostensparende Synthese elektronischer Hardware in einen FPGA[1] oder ASIC[2]. Die Entwicklung startet auf Ebene A (vgl. Abbildung 1) mit dem sog. Design-Entry und endet auf Ebene D mit dem fertigen Chip. In den Ebenen dazwischen (B/C) nähert sich der Entwurf schrittweise von der abstrakten Beschreibung (mathematischen Beschreibung) der Ebene A bis zur physikalischen Beschreibung und Realisierung auf Ebene D.

- Ebene A heißt Design-Entry. Auf dieser Ebene liegt der Entwurf als abstrakte geometrische oder mathematische Beschreibung vor. Als Design-Entry sind heute Schematics (grafischer Design-Editor zum Schaltplan-Entwurf) oder HDLs[3] üblich. Ziel ist die Simulation und die iterative Verbesserung des Entwurfs innerhalb von Ebene A. Hierzu existieren leistungsstarke Softwaren für Simulation und Verifikation. Alle tieferen Ebenen sind mit erheblichem Zeit- und Geldaufwand verbunden.
- Ebene B heißt Technologie-Abbildung. Hier wird festgelegt, auf welche Ziel-Hardware das Modell umgesetzt wird. An dieser Stelle wird entschieden, ob die Schaltung in einem ASIC, FPGA, CPLD[4] oder vielleicht sogar diskret realisiert werden soll. Im Falle einer FPGA-Realisierung muß die FPGA-Technologie[5] und die FPGA-Familie (und damit i.a. der Hersteller) festgelegt werden. Der Schaltungsentwurf wird dann auf die Primitivzellen der FPGA-Technologie abgebildet.
- Auf Ebene C wird die Plazierung und Verdrahtung der Primitivzellen des FPGA auf dessen Gatter-Array festgelegt. Hierzu muß der FPGA-Typ (Gehäuse-Typ) vorher ausgewählt werden. Die Art der Plazierung der Primitivzellen, ihre Verdrahtung untereinander und nach außen legt in hohem Maße den Platzbedarf des Entwurfs im FPGA und die maximal erreichbare Arbeitsfrequenz fest. Die meisten Synthese-Softwaren bieten auf dieser Ebene die Möglichkeit zur erneuten Simulation (Back Annotated Gate Level Simulation). Die Gate Level Simulation bietet eine zuverlässige Beurteilung über die Leistungsfähigkeit der Technologie-Abbildung. Anders als die funktionelle Simulation von Ebene A berücksichtigt die Gate Level Simulation auch Signallaufzeiten innerhalb des FPGA (Timing Simulation). Erst die Gate-Level-Simulation läßt eine Aussage darüber zu, ob der Entwurf auf der angestrebten Hardware realisierbar ist.
- Ebene D ist die Hardware-Ebene. Hier finden die finalen Tests mit dem bereits fertig synthetisierten Entwurf und dem programmierten FPGA statt. Fehler, die an dieser Stelle entdeckt werden, machen einen vollständigen Durchlauf von Ebene A zu Ebene D erforderlich, sind zeitintensiv und teuer. Der Entwurf sollte soweit wie möglich bereits auf Ebene A verbessert werden.

[1] Field Programable Gate Array
[2] Applied Specifi c Integrated Circuit
[3] Hardware Beschreibungssprache
[4] Complex Programable Logic Device
[5] SRAM, EPROM, EEPROM/FLASHRAM, Antifuse

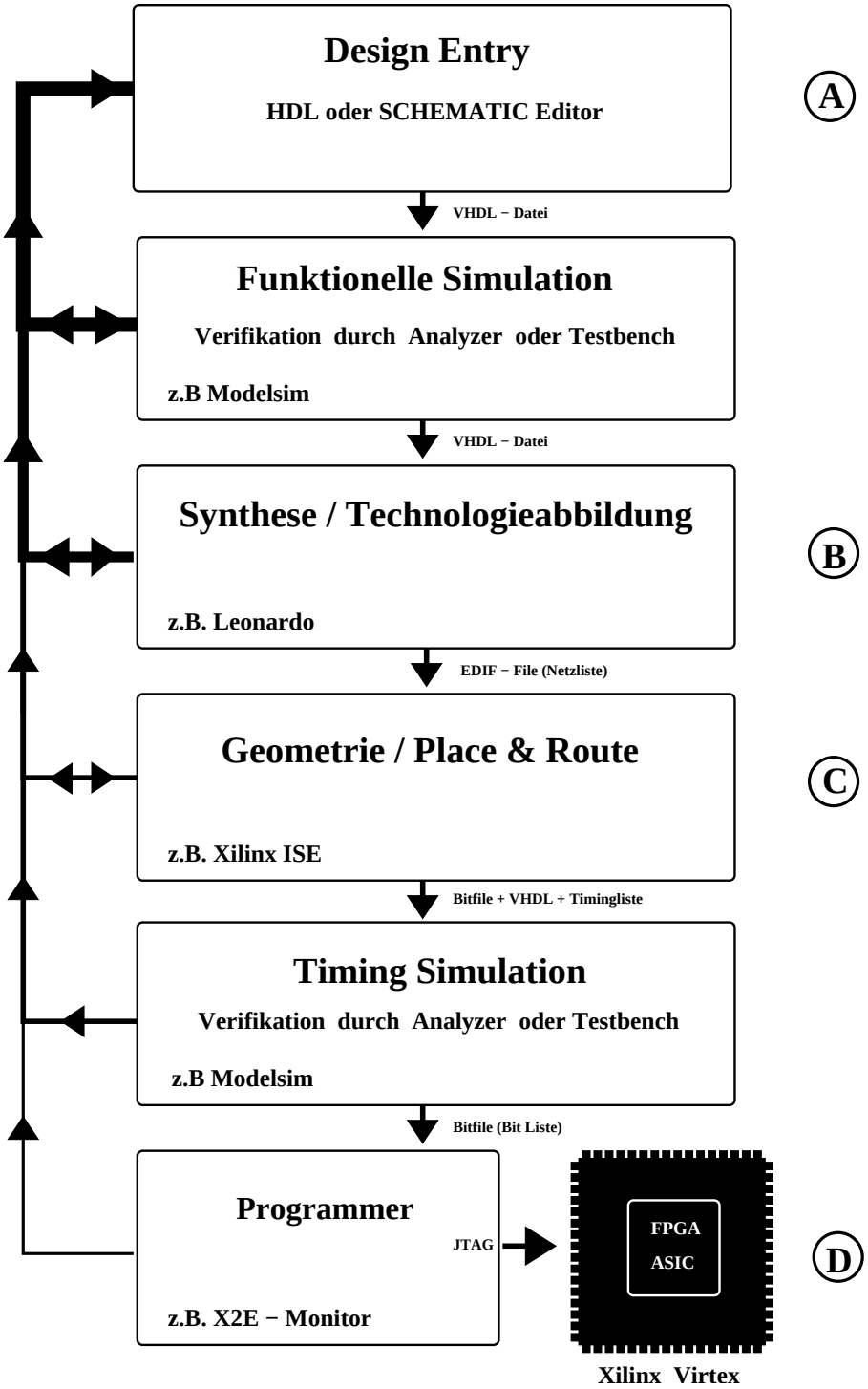


Abbildung 1: *Hardware Synthese: Der Design-Vorgang unter VHDL.*

Inhaltsverzeichnis

Kapitel 1

VHDL Grundlagen

Eine Hardware-Beschreibungssprache (HDL *Hardware-Description-Language*) dient dem Entwurf und dem Test elektronischer Schaltungen, und stellt eine Alternative zum *Schematics* Schaltungsentwurf dar. HDLs können sowohl die Struktur, als auch die Funktion elektronischer Schaltungen abbilden. Die Syntax ähnelt i.a. der höherer Programmiersprachen, enthält aber auch zusätzliche Sprachelemente, z.B. für die Verschaltung von Funktionsblöcken. Während HDLs ursprünglich eher zur Beschreibung und Dokumentation von Schaltungen gedacht waren, haben sie sich rasch in der Entwicklung und Simulation integrierter digitaler Schaltungen (ICs) durchgesetzt. Besonders auf dem Gebiet der frei programmierbaren integrierter Schaltungen wie CPLDs [1] und FPGAs[2], sowie im Umfeld der ASIC [3] Entwicklung sind HDLs heute weit verbreitet. Nahezu jeder Hersteller von CPLDs/FPGAs bietet heute eigene HDL-basierende Design Tools an. Diese bieten in der Regel eine vollständige IDE [4] mit HDL-Editor, Synthese-Compiler, Simulator und Programmierer. Die Verwendung von HDL-Modellen aus Bibliotheken in Verbindung mit leistungsfähigen Simulatoren bietet die Möglichkeit zur kostengünstigen und schnellen Entwicklung komplexer digitaler Schaltungen. Der Entwickler hat zudem die Möglichkeit, Alternativen zu bewerten, ohne jemals einen Prototyp herstellen zu müssen.

Die beiden weit verbreitetesten HDLs sind VERILOG und VHDL [Ashenden 1990, Lehmann et al. 1994]. Die Syntax von VHDL ähnelt sehr der Programmiersprache ADA. Sie enthält alle Elemente einer prozeduralen Programmiersprache, erweitert um Konstrukte für den Schaltungsentwurf. Die Abkürzung VHDL steht für *VHSIC Hardware-Discription-Language*, und VHSIC für *Very-High-Speed-Integrated-Circuits*. VHDL wurde seit 1983 entwickelt und 1987 durch das IEEE[5] standardisiert. Die Sprache hat unlängst begonnen, ihre Mitkonkurrenten (VERILOG und Andere) zu verdrängen und sich als weltweiter Standard zu etablieren. Entwürfe in VHDL sind Herstellerunabhängig und deshalb leicht portierbar. Sie können nicht nur zwischen verschiedenen Rechnerplattformen, sondern auch zwischen Software verschiedener Hersteller ausgetauscht werden. Der Entwickler ist bei der Wahl der Rechnerplattform und der Software somit relativ unabhängig.

1.1 Aufbau

Ein VHDL-Design bietet eine vollständige Beschreibung einer elektronischen Schaltung. Dazu ist es möglich, sowohl die Struktur (Aufbau in Funktionsblöcke und Verdrahtung dieser Blöcke), als auch die Funktion (funktionelle Beschreibung mit Hilfe von Operatoren) einer Schaltung abzubilden. Jedes VHDL-Design ist untergliedert in ENTITY und ARCHITECTURE. Optional kann auch ein CONFIGURATION-Teil existie-

[1] Compact Programable Logic Devices
[2] Field Programable Gate Arrays
[3] Applied Specifi c Integrated Circuits
[4] Integrated Development Environment
[5] IEEE: Institute of Electrical and Electronic Engineers; Standardisierungs-Gremium der USA

ren. In Abbildung 1.1 ist der schematische Aufbau eines VHDL-Designs gezeigt.

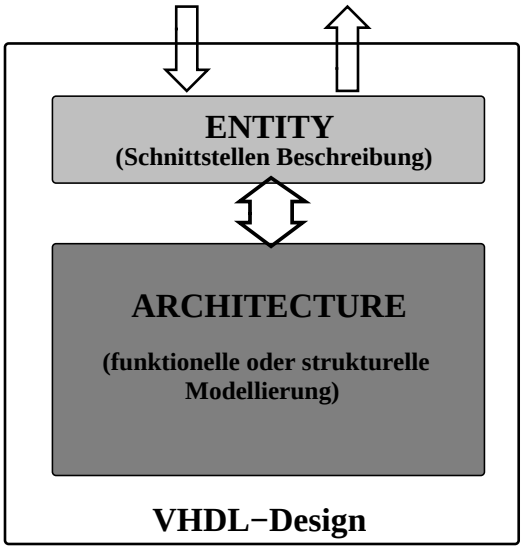


Abbildung 1.1: *Vereinfachte Form eines VHDL-Designs. Die Minimalkonfiguration eines VHDL-Designs besteht aus einer Schnittstellenbeschreibung (*ENTITY*) und der eigentlichen Designeinheit, der* ARCHITECTURE*.*

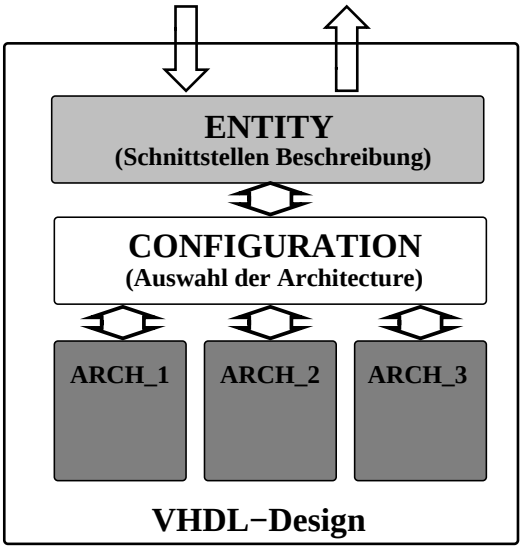


Abbildung 1.2: *Vollständiger Aufbau eines VHDL-Designs. VHDL spezifiziert zusätzlich zum* ENTITY*- und* ARCHITECTURE*-Teil, die* CONFIGURATION*. Die* CONFIGURATION *erlaubt die Auswahl verschiedener* ARCHITECTURE*.*

Die ENTITY definiert die Schnittstelle des Designs. Sie beschreibt alle Ein- und Ausgänge des Designs und spezifiziert die Signaltypen. Die ENTITY ist dabei mit den Anschlüssen eines IC oder dem Stecker einer Platine vergleichbar (siehe hierzu auch Abbildung 1.4).

Die ARCHITECTURE beinhaltet die Beschreibung des Entwurfs, z.B. die funktionelle Aufgabe der Applikation. Im Vergleich mit dem IC, entspricht dies in etwa dem internen Gatteraufbau der integrierten Schaltung. Abbildung 1.2 zeigt die vollständige Spezifikation von VHDL. Sie enthält auch die CONFIGURATION. Mit der CONFIGURATION können vom Entwickler verschiedene ARCHITECTURES ausgewählt und verglichen werden. Alle ARCHITECTURES verwenden in diesem Fall die gleiche ENTITY. In den meisten Fällen ist solch eine Auswahl jedoch nicht erforderlich. Die CONFIGURATION entfällt dann vollständig und es wird die in Abbildung 1.1 gezeigte Form verwendet. Die nachfolgenden Ausführungen beziehen sich deshalb auch ausschließlich auf Designs ohne CONFIGURATION-Teil wie in Abbildung 1.1.

Abbildung 1.3 zeigt das vollständige VHDL-Listing eines D-FlipFlop. In der *Port*-Liste werden alle Schnittstellen vereinbart, mit denen Signale in das FlipFlop hinein oder heraus geführt werden. Als Eingänge sind dies hier das **CLK**-Signal (Clock), das **RESET**-Signal und der **D**-Eingang des FlipFlops. Als einziger Ausgang ist **Q** definiert.

Port-Signale verhalten sich innerhalb von VHDL wie globale Variablen, sie sind in allen Strukturen einer ARCHITECTURE bekannt und können nicht lokal überdefiniert werden. Neben dem *Port* selbst, ist auch für jedes Port-Signal der zugehörige Signal-Typ deklariert. Solche Typen können neben dem sehr häufigen Bit-Typ *std_logic*, auch *Integer*-Typen oder sogar ganze Datenbus-Typen in Form von Vektor-Typen (*std_logic_vector, signed, unsigned*) sein. Im Beispiel des D-FlipFlop definiert der Type *std_logic*

Schnittstelle (entity)

```
ENTITY D-FlipFlop IS
   PORT  (clk     : IN STD_LOGIC;   -- Port Liste
          d       : IN STD_LOGIC;   --
          reset   : IN STD_LOGIC;   --
          q       : OUT STD_LOGIC;  --
          );
END D-FlipFlop ;
```

Architektur (architecture)

```
ARCHITECTURE dff OF D-FlipFlop IS
BEGIN
   d-flipflop: PROCESS (CLK;D;RESET) --
   BEGIN                             --
      IF reset = '1' THEN            -- Reset
         q <= '0';                   -- Bedingung
      ELSE                           --
         IF RISING_EDGE(CLK) THEN    -- CLK Flanke
            q <= d;                  --
         END IF;                     --
      END IF;                        --
   END PROCESS;                      --
END dff;
```

Konfiguration (configuration)

```
CONFIGURATION Konfig OF D-FlipFlop IS
  FOR dff;
  END FOR;
END Konfig ;
```

Abbildung 1.3: *Struktur einer VHDL Beschreibung. Die* CONFIGURATION *ist optional und unnötig, wenn nur eine einzige* ARCHITECTURE *existiert. Die Zuweisung zur* ENTITY *ist in diesem Fall dann eindeutig.*

ein *Port*-Signal mit den beiden logischen Zuständen 0 und 1. Jedem Port-Signal ist zusätzlich noch das Attribut (**IN**/**OUT**) zugeordnet. Es legt fest, ob das Signal als Eingang oder Ausgang verwendet wird. Grundsätzlich gilt, daß **IN**-Ports nur gelesen (nicht getrieben) werden können. Entsprechend können **OUT**-Ports nur getrieben (nicht gelesen) werden.

Die ARCHITECTURE enthält die eigentliche Hardware-Beschreibung des VHDL-Designs. Es werden im wesentlichen zwei Beschreibungsformen unterschieden:

- funktionelle Beschreibung
- strukturale Beschreibung

Elektronische Schaltungen bestehen i.a. aus Bauteilen die auf einer Leiterplatte (PCB[6]) montiert sind.

[6] Printed Circuit Board

Genau diese beiden Aspekte müssen von VHDL beschrieben werden. Sie sind durch die Beschreibungsformen ’funktionell’ und ’struktural’ realisiert.

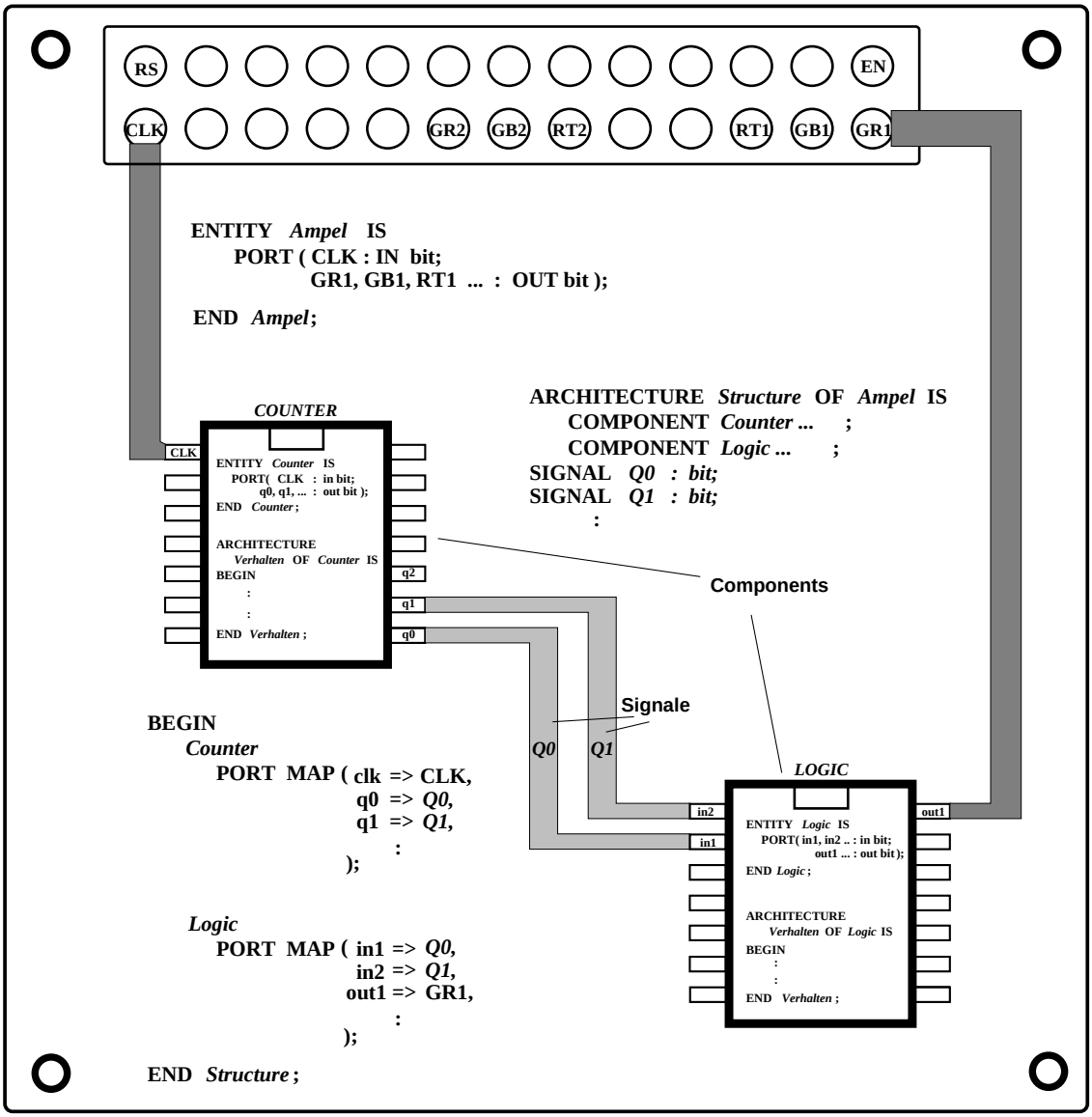


Abbildung 1.4: *Funktionelle und strukturale Modellierung.*

Die funktionelle Beschreibung dient der Modellierung von Bauteilen. Dabei entspricht die ENTITY der Schnittstelle des Bauteils (dem Sockel), die ARCHITECTURE beschreibt stattdessen die Funktion (deshalb funktionelle Beschreibung) des Bauteiles. Die strukturale Beschreibung modelliert die Leiterplatte. Ihre ENTITY entspricht dem Platinenstecker, die ARCHITECTURE beschreibt hier die Verdrahtung der Bauteile.

Abbildung 1.4 fasst das Konzept in einer Darstellung zusammen. Sie zeigt eine fiktive Ampelsteuerung, die aus einer Platine mit Bauteilen (ICs) besteht. Jedes IC ist für sich in einem VHDL-Modell beschrieben (funktionelles VHDL). Aber auch die Platine selbst enthält eine VHDL-Beschreibung (strukturales VHDL) welche die Bauteile miteinander und mit der ENTITY der Platine verbindet. Diese Aufteilung in funktionelle und strukturale Beschreibungen ist nahezu in jedem größeren VHDL-Projekt zu finden. Meistens sind aber beide Aspekte in ein und derselben Entwurfsbeschreibung vermischt.

1.2 Funktionelle Modellierung

Die funktionelle Modellierung beschreibt das Verhalten von Bauteilen, sie wird deshalb auch oft als *Behavioral*-Modell bezeichnet. Abbildung 1.5 zeigt eine solche Beschreibung. Die ENTITY beschreibt die Schnittstelle des Bauteiles, die ARCHITECTURE die Funktion.

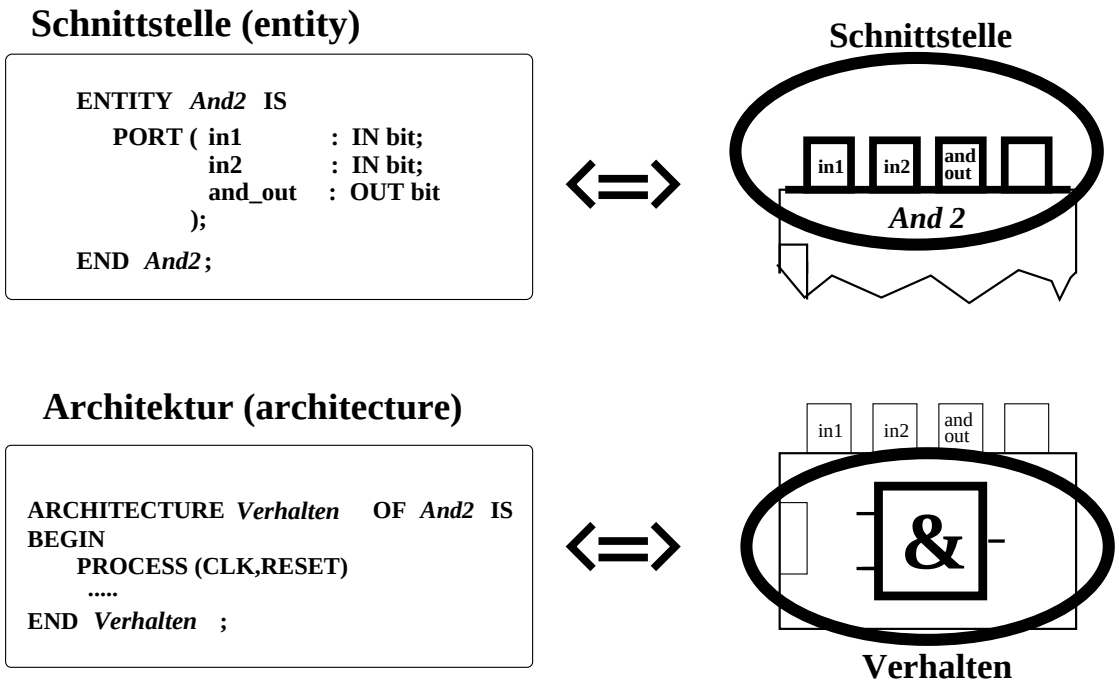


Abbildung 1.5: *Die funktionelle oder Verhaltens- Modellierung von VHDL: Die funktionelle Modellierung erzeugt die Funktion von Bauteilen. Diese können dann in der strukturalen Modellierung zu komplexen Systemen miteinander verbunden werden.*

Die funktionelle Modellierung ermöglicht, die Aufgaben der Schaltung in separate funktionelle Gruppen zu gliedern. Diese können genau wie elektronische Bauteile in getrennten Prozessen oder in verschiedenen Teams entwickelt und getestet werden, bevor sie zum Schluß zu einer Beschreibung zusammengefügt werden. Durch dieses Baukastenprinzip können komplexe Schaltungen leicht aus einfachen Gruppen zusammengefügt werden.

Die wichtigste Struktur für die funktionelle Beschreibung ist der '**Prozess**'. Prozesse arbeiten wie Computerprogramme sequenziell und werden von Signalen gestartet. Eine andere sehr wichtige Struktur sind Ablaufsteuerungen, so genannte FSMs[7]. Auf beide wird im Folgenden näher eingegangen.

[7] Finite State Machine

1.2.1 Prozesse

Prozesse sind das Herz der funktionellen Beschreibung. Sie reagieren auf Signale und arbeiten eine Liste von Kommandos ab. Dabei modifizieren sie andere Signale, die als Ausgabe des Prozesses dienen. Prozesse sind sequenziell arbeitende prozedurale Strukturen, eine VHDL-Beschreibung kann beliebig viele von ihnen enthalten. Prozesse können gleichzeitig starten, wenn sie durch das gleiche Signal ausgelöst werden. Damit wird die parallele Struktur der Elektronik abgebildet.

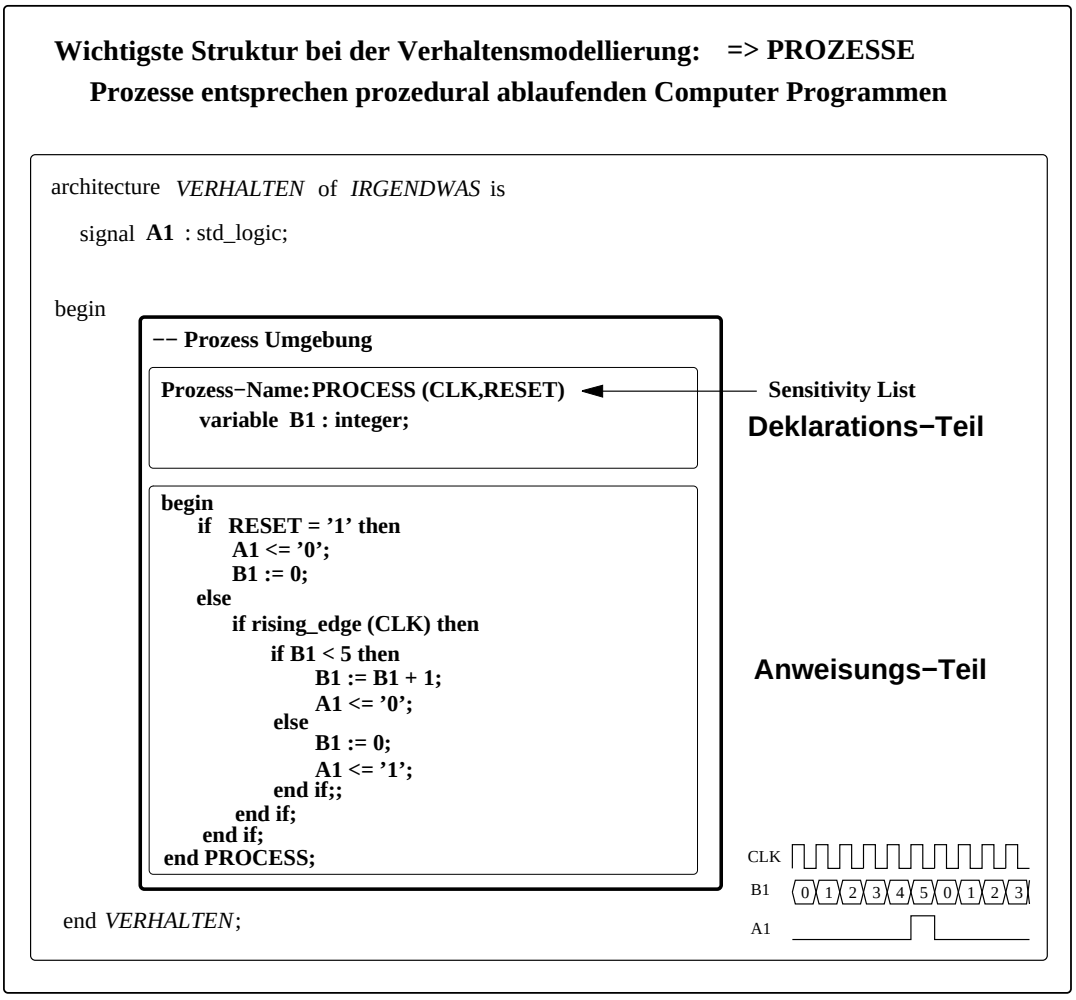


Abbildung 1.6: *Das Beispiel zeigt den Aufbau der Prozess-Umgebung. Auch Prozesse bestehen aus einem Deklarations und einem Anweisungsteil. Die Befehle des Anweisungsteils werden sequenziell abgearbeitet. Der Prozess wird getriggert durch Signale in der Sensitivity-List des Prozess-Kopfes, oder durch eine Wait-Anweisung im Anweisungsteil.*

Die wichtigsten Merkmale von Prozessen sind:

- Prozesse arbeiten sequenziell genau wie Computerprogramme.
- Prozesse bestehen aus einem Deklarations- und einem Anweisungsteil.
- Prozesse werden durch Signale in der '**Sensitivity List**' (im Prozess-Kopf) oder durch eine **Wait**-Anweisung (im Anweisungsteil) gestartet.

- Prozesse können lokale Variablen deklarieren (im Deklarations-Teil).
- Variablen des Prozesses ändern sich im Moment der Zuweisung.
- Prozesse manipulieren Signale (Prozessausgabe).
- Signale des Prozesses ändern sich erst nach Beendigung des Prozesses (Delta-Zyklus).
- Prozesse sind parallele Strukturen, d.h. beliebig viele Prozesse können gleichzeitig starten.

Abbildung 1.6 zeigt den Grundaufbau eines Prozesses. Im Deklarations-Teil steht der Prozess-Kopf mit dem Prozess-Namen und der Sensitivity-List. Die Sensitivity-List ist der Inhalt der runden Klammer hinter dem Wort '***Prozess***'. Sie enthält diejenigen Signale, die den Prozess starten (triggern). Starten bedeutet, daß die Befehle des Anweisungsteiles sequenziell abgearbeitet werden. Jede Änderung eines Signals der Sensitivity-List bewirkt dabei den Start des Anweisungsteils. Im Beispiel von Abbildung 1.6 bedeutet das, daß mit jeder Flankenänderung von 'CLK' oder 'RESET' der Prozess erneut gestartet wird, also zweimal pro 'CLK'-Zyklus. Das 'RESET'-Signal könnte in diesem Fall also auch außerhalb eines 'CLK'-Zyklus (asynchron mit 'CLK') den Prozess starten.
Würde 'RESET' hingegen aus der Sensitivity-List entfernt, so wird der Reset mit der 'CLK'-Signalflanke synchron ausgeführt. Der Zustand des Signals 'RESET' wird dann nur bei jeder 'CLK'-Flanke abgefragt (gesamplet).

Es gibt noch einen zweiten Startmechanismus in Form einer 'Wait'-Anweisung die anstatt der Sensitivity-List verwendet wird. In diesem Fall entfällt die Sensitivity-List vollständig. Der Prozess wartet dann, bis sich die Signale innerhalb der Wait-Bedingung des Anweisungsteils ändern bevor er startet. Beide Mechanismen sind in Abbildung 1.8 einander gegenüber gestellt.

Im Anweisungsteil in Abbildung 1.6 steht ein typisches Konstrukt, um den Reset durchzuführen. Eine solche Reset-Bedingung sollte grundsätzlich immer existieren. Es ist wichtig, daß bei aktivem Reset alle Signale und Variable des Entwurfs auf definierte logische Werte gesetzt werden. Andernfalls kann es zu undefinierten Signal-Zuständen und zu unvorhersagbaren Effekten innerhalb des Entwurfs kommen!
Im Normalbetrieb, wenn 'RESET' nicht aktiv ist, startet die 'else'-Bedingung den Prozess-Rumpf. Der erste Konstrukt dort prüft das 'CLK'-Signal auf eine ansteigende Flanke (if rising_edge(CLK) then). Die anschließende 'if'-Bedingung wird deshalb nur mit jeder ansteigenden Flanke des 'CLK'-Signals gestartet, während der Prozess selbst mit jeder ansteigenden und abfallenden Flanke von 'CLK' getriggert wird. Die 'if'-Bedingung enthält hier einen Zähler bis '5', der mit der internen Variable 'B1' des Prozesses realisiert ist. Das Signal 'A1' dient als Prozess-Ausgang. Es wird in Abhängigkeit der Zählvariable 'B1' verändert. Die Änderung von 'A1' kann maximal einmal pro Prozess-Aufruf erfolgen. Sie wird erst wirksam, wenn der Prozess vollständig abgearbeitet ist. Es hat also keinen Sinn, ein Signal pro Prozessaufruf mehrmals innerhalb des Prozesses zu ändern (z.B in einer Zählschleife), da die Änderung erst nach Beendigung des Prozesses wirksam wird.

Die Anzahl der Befehlszeilen pro Prozess hat einen kritischen Einfluß auf die maximal erreichbare Arbeits-Frequenz des Entwurfs. Werden sehr viele und zeitraubende Operationen in einem Prozess (der mit 'CLK' synchron arbeitet) verwendet, so wird später das Synthese-Programm die maximal erreichbare Arbeits-Frequenz danach festlegen, daß alle Befehle des Prozesses innerhalb eines 'CLK'-Zyklus abgearbeitet sein müssen. Je mehr Befehle also pro 'CLK'-Zyklus in einem Prozess verarbeitet werden müssen, desto niedriger wird die maximal erreichbare Arbeitsfrequenz. Aus diesem Grunde ist es oft ratsam, komplexe Operationen auf mehrere Prozesse zu verteilen, die z.B. von einer Ablaufsteuerung kontrolliert werden können.

In Abbildung 1.6 (unten rechts) ist das zeitliche Verhalten von 'A1' in Relation zum 'CLK'-Signal dargestellt. Der Zähler bewirkt, daß alle fünf 'CLK'-Signale das Signal 'A1' für die Dauer von einem 'CLK'-Zyklus auf logisch Eins geht. Abbildung 1.7 verdeutlicht noch einmal die Parallelität von Prozessen.

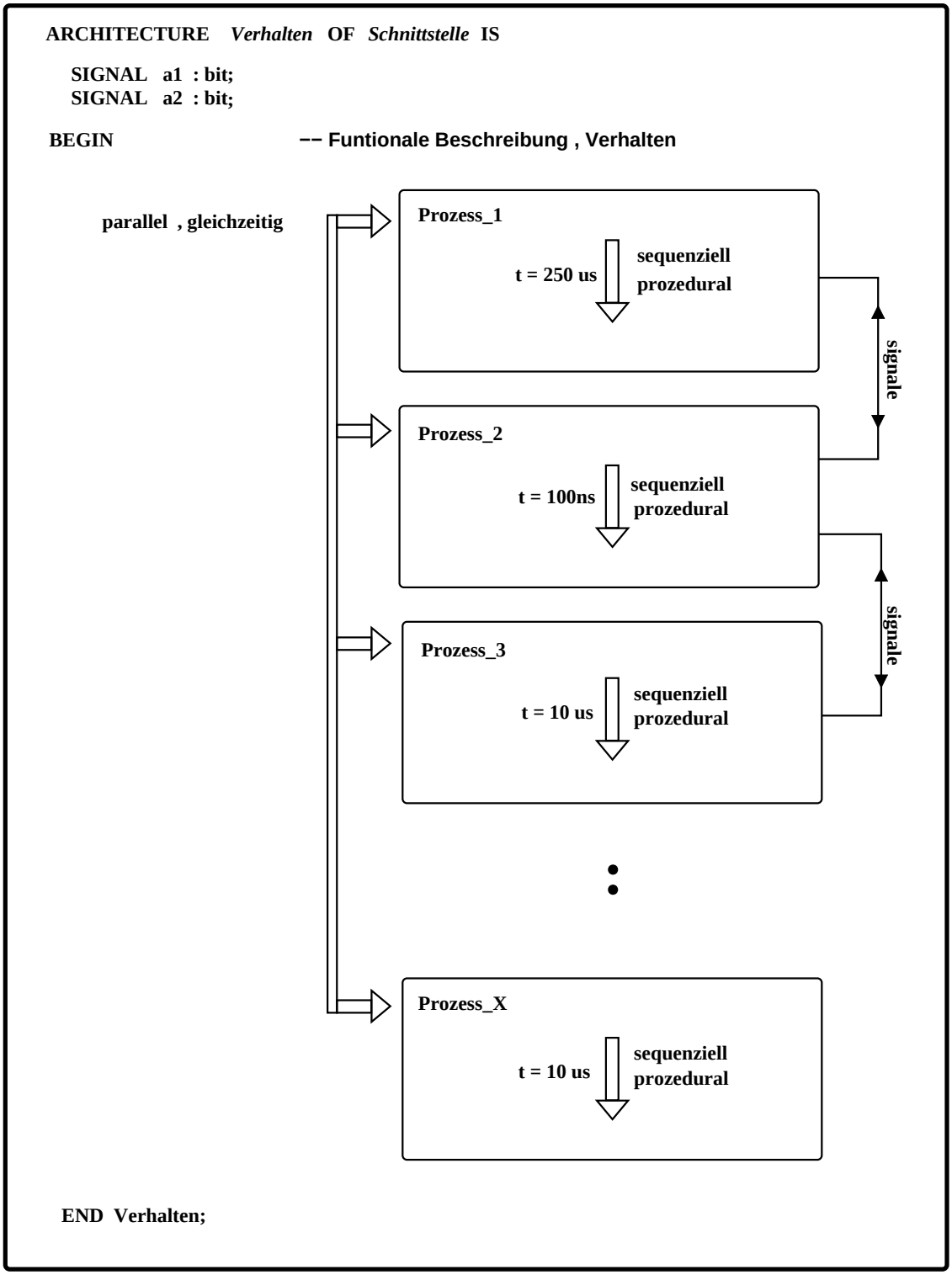


Abbildung 1.7: *Prozess-Aufrufe können grundsätzlich parallel erfolgen. Die Laufzeit der Einzelprozesse hängt dabei erheblich von der Anzahl ihrer Programmzeilen ab, was sich auf die maximale Arbeitsfrequenz bei der Synthese auswirkt. Oft ist eine große Zahl kurzer Prozesse mit hoher Arbeitsfrequenz einer geringen Zahl von Prozessen mit deutlich niederer Frequenz vorzuziehen. Die Synchronisation mehrerer Prozesse, die voneinander abhängig sind, erfordert jedoch oft eine Ablaufsteuerung, vgl. nächster Abschnitt.*

Aufbau von Prozessen:

```
ARCHITECTURE  Verhalten  OF  Schnittstelle  IS        -- Kommentare ....

  SIGNAL  a1 : bit;
  SIGNAL  a2 : bit;

BEGIN

  Prozess_1:  PROCESS (CLK, RESET);                  -- Sensitivity List -> Trigger
                                                     -- Bei jeder Änderung der Signale
      VARIABLE  B1  :  integer;                      -- CLK oder  RESET wird der Prozess
      VARIABLE  B2  :  integer;                      -- gestartet.
      VARIABLE  C1  :  integer;                      -- Variablentypen:  natural, integer,
      VARIABLE  D1  :  real := 1.4;                  -- real ...
  BEGIN
      IF  RESET = '1'  THEN
          a1 <= '0';                                 -- Bei "Reset" alle Signale auf definierten
          a2 <= '0';                                 -- Zustand.
      ELSE
          IF  rising_edge (CLK)  THEN                -- Test auf ansteigende Flanke von CLK
              C1 :=  B1 + B2;                        -- Variablenzuweisung: ":="
              IF  C1 > 8  THEN
                  a1 <= '1';                         -- Signalzuweisung:  "<="
                  a2 <= '0';                         -- Signale sollen  in jedem Zustand
              ELSE                                   -- definiert sein!
                  a1 <= '0';
                  a2 <= '1';
              END IF;
          END IF;
      END IF;
  END  PROCESS  Prozess_1;

  Prozess_2:  PROCESS                                -- Prozess ohne Sensitivity List
      VARIABLE  B1  :  integer;
      VARIABLE  B2  :  natural;
      VARIABLE  C1  :  real := 1.4;
      CONSTANT  P   :  real := 3.1415
  BEGIN
      IF  RESET = '1'  THEN
          :
      ELSE
          IF  rising_edge (CLK)  THEN
              :
          END IF;
      END IF;
      WAIT  ON  CLK, a1;                             -- Prozess mit  "WAIT" - Trigger.
                                                     -- Bei jeder Änderung von CLK oder a1
  END PROCESS  Prozess_2;                            -- wird der Prozess gestartet.#

  :

END  Verhalten;
```

Abbildung 1.8: *Gegenüberstellung der beiden Methoden zum Prozess-Aufruf:* ***'Sensitivity-List'*** *und* ***'Wait-Anweisung'****.*

1.2.2 Endliche Zustandsautomaten

Ein Zustandsautomat FSM[8] ist eine sequenziell arbeitende Logikschaltung die – gesteuert durch ein periodisches Taktsignal – eine Abfolge von Zuständen durchläuft.

FSMs sind nicht nur ein wichtiges Konzept bei der Entwicklung digitaler Applikationen. Ihre Theorie liegt quasi jeder Form von Automation zu Grunde, die in der Technik verwirklicht ist (von der Waschmaschine bis zum Steuerwerk von Mikroprozessoren). In VHDL beruhen FSMs auf Prozessen als zugrundeliegende Struktur. FSMs ermöglichen den Aufbau von Steuerungen womit der Ablauf von Prozessen zeitlich synchronisiert wird. Auf diese Weise lassen sich komplexe Steueraufgaben in überschaubare Prozessabläufe untergliedern. Der Grundaufbau einer FSM ist in Abbildung 1.9 gezeigt.

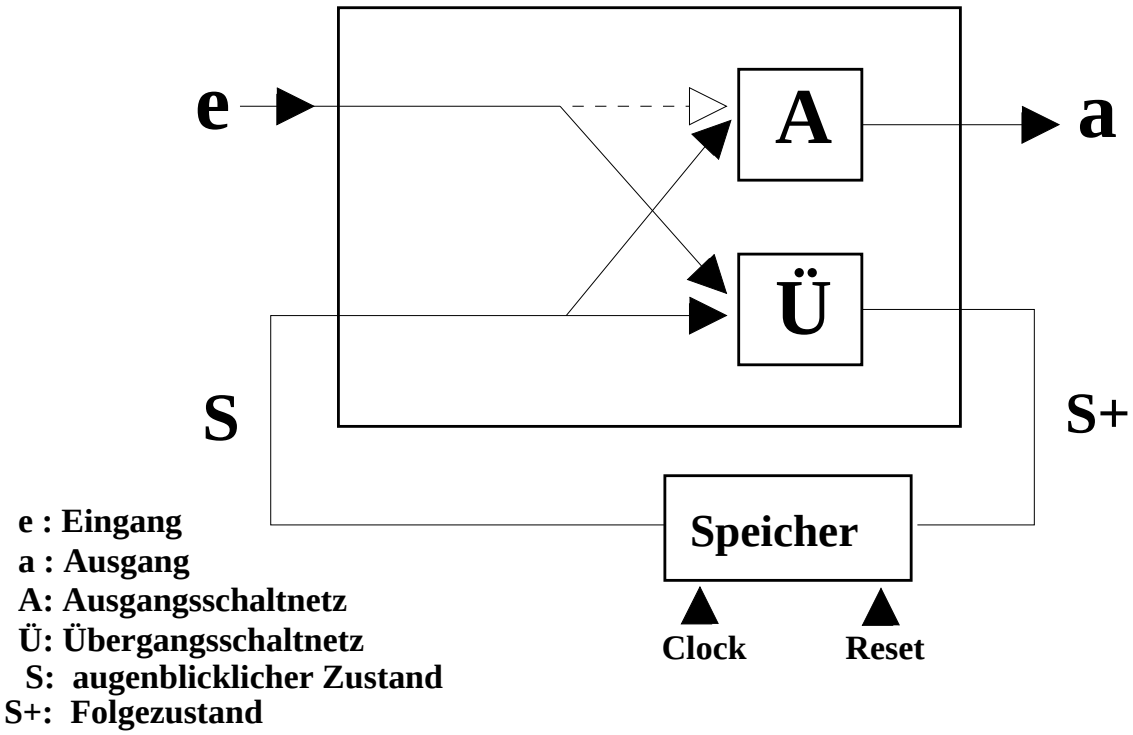


Abbildung 1.9: *Blockschaltbild eines Zustandsautomats (FSM). Die FSM besteht aus einem Übergangsschaltnetz 'Ü' und einem Ausgangschaltnetz 'A'. Das Ausgangsschaltnetz legt den Ausgangswert 'a' in Abhängigkeit vom inneren Zustand 'S' fest. Das Übergangsschaltnetz bestimmt, wann der Automat von 'S' in den Folgezustand 'S+' übergeht.*

Jede FSM besteht im wesentlichen aus einem Übergangsschaltnetz 'Ü' und einem Ausgangsschaltnetz 'A'. Außerdem existiert ein Speicher, in dem der aktuelle Zustand gespeichert ist. Viele, aber nicht alle, FSM besitzen einen Eingang. Die FSM befindet sich immer in einem ihrer internen Zustände. Wann der Zustand des Automaten wechselt (Übergang von 'S' nach 'S+'), wird vom Übergangsschaltnetz festgelegt. Das Übergangsschaltnetz reagiert dabei direkt auf ein Eingangssignal 'e' oder es besitzt eine Uhr oder einen Zähler, der den Übergang auf den neuen Zustand festlegt. Das Ausgangsschaltnetz bestimmt die Ausgabe des Automaten. Es reagiert entweder nur auf den aktuellen inneren Zustand 's' (Moore Automat), oder es reagiert auf 'S' oder 'e' (Mealy Automat – gestrichelt).

Das Listing einer einfachen FSM in VHDL ist in Abbildung 1.10 gezeigt. Um eine FSM zu realisieren wird meistens ein Prozess mit einer 'case'-Anweisung verwendet.

[8] Finite State Machine

```
---
-- Typ Definition
---
type  ZUSTAND_TYP  is  (Z0, Z1, Z2, Z3);

signal  ZUSTAND    : ZUSTAND_TYP;
signal  COUNT      : integer  range 0 to 5;
```

Definition des Typs 'ZUSTAND_TYP' und der möglichen internen Zustände (Z0 bis Z4), die eine Instanz dieses Typs annehmen kann.

Erzeuge das Signal 'ZUSTAND' als Instanz des Typs 'ZUSTAND_TYP'

```
---
-- Übergangsschaltnetz
---
s_machine: process(CLK,RESET);
begin
    if rising_edge (CLK) then   -- ansteigende Flanke
       if RESET = '1' then
          ZUSTAND  <=  Z0;
       else
          case  ZUSTAND  is
             when  Z0  =>
                   ZUSTAND  <=  Z1;
             when  Z1  =>
                   ZUSTAND  <=  Z2;
             when  Z2  =>
                   if  COUNT  =  5  then
                      ZUSTAND  <=  Z3;
                   else
                      ZUSTAND  <=  Z2;
                   end if;
             when  Z3  =>
                   ZUSTAND  <=  Z0;
          end  case;
       end if;
    end if;
end process;
```

Übergangsschaltnetz

Innerhalb des Übergangsschaltnetzes werden die Bedingungen festgelegt, unter denen der Zustandsautomat in den nächsten Zustand wechselt.

Solche Bedingungen können z.B. einfache Zählerabfragen sein, oder es werden Signale bzw. Eingangs−Ports ausgewertet.

Der Zustandsautomat befindet sich zu jedem Zeitpunkt in einem definierten Zustand!

```
---
-- Ausgangsschaltnetz
---

-- Zustandsabhängiger Prozess Start
zustand_0:  process (CLK,RESET);
begin
      if  rising_edge(CLK) then
         if  RESET  = '1' then
            ......  <= '0';
         else
            if  ZUSTAND  =  Z0  then
               ......
               ......
            end if;
         end if;
      end if;
end process;


 -- direkte zustandsabhängige Signal Zuweisung
Output1  <= '1' when (ZUSTAND = Z0) else
            <= '0' when (ZUSTAND = Z1) else
            <= '1' when (ZUSTAND = Z2) else
            <= '0' when (ZUSTAND = Z3);
```

Ausgangsschaltnetz

Aufgabe des Ausgangsschaltnetzes ist die Zuweisung von Ausgangswerten zu den jeweiligen Zuständen. Solche Zuweisungen können einfache Signalzuweisungen aber auch komplizierte Operationen sein, die innerhalb von Prozessen abgewickelt werden. Auch die Zuweisung von Ausgangs−Ports ist möglich.

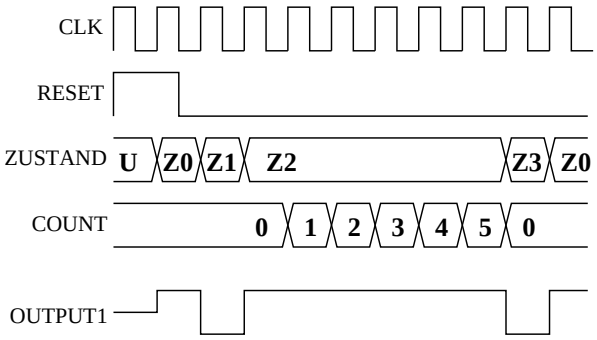


Abbildung 1.10: *Die Realisierung einer FSM in VHDL. Das Übergangsschaltnetz enthält eine 'case'-Anweisung die festlegt, wann die FSM in den nächsten Zustand übergeht (hier rein zählergesteuert). Das Ausgangsschaltnetz besteht in der Regel aus direkten Signalzuweisungen, kann aber auch zustandsabhängige Prozessaufrufe enthalten.*

In Abbildung 1.10 wird zunächst ein eigener Typ '***Zustand_Typ***' mit den Zuständen '**Z0**', '**Z1**', '**Z2**' und '**Z3**' definiert. Dieser Zustandstyp enthält eine Liste aller möglichen Zustände in denen sich die FSM später befinden kann. Nach der Typ-Definition folgt die Signalzuweisung für diesen Typ an das Signal '***ZUSTAND***'. Hierdurch wird eine Instanz des Typs '***ZUSTAND_TYP***' erzeugt.

Das Übergangsschaltnetz besteht aus einem Prozess (s_machine), der mit dem '**CLK**'-Signal getriggert ist. Mit jeder ansteigenden Flanke (rising_edge) von 'CLK' wird in der 'case'-Anweisung der Inhalt des Signals '*ZUSTAND*' getestet (case *ZUSTAND* is). Direkt nach dem Reset ist '*ZUSTAND*' im Zustand 'Z0'. Bereits mit Anstieg der ersten 'CLK'-Flanke nach dem Reset ist die Bedingung 'when Z0' gültig und '*ZUSTAND*' geht in den Zustand 'Z1'. Bei der zweiten 'CLK'-Flanke steht '*ZUSTAND*' nun auf 'Z1' (Bedingung 'when Z1' ist gültig) und geht über auf 'Z2'. Während 'Z2' verhält sich der Ablauf nun anders. Hier wird zunächst die Zählvariable 'COUNT' ausgewertet bevor das Signal '*ZUSTAND*' neu zugewiesen wird. Dazu läuft ein 'CLK'-synchroner Zähler (nicht abgebildet), der mit jeder 'CLK'-Flanke erhöht wird. Sobald 'COUNT=5' erreicht ist, geht '*ZUSTAND*' über auf 'Z3'. Im Zustand 'Z3' wird schließlich mit der nächsten 'CLK'-Flanke zurück auf 'Z0' gewechselt und der FSM Zyklus läuft von neuem.

Das Ausgangsschaltnetz erzeugt die Ausgangs-Signale der FSM (hier Output1). Es kann z.B. aus direkten Signalzuweisungen oder aus weiteren Prozessen bestehen, die auf die inneren Zustände der FSM reagieren. Das im Beispiel von Abbildung 1.10 gezeigte Ausgangsschaltnetz besteht aus dem Prozess 'zustand_0' der nur anläuft, wenn sich '*ZUSTAND*' im Zustand 'Z0' befindet und einer direkten Signalzuweisung des Signals 'OUTPUT1' in Abhängigkeit der Zustände der FSM.

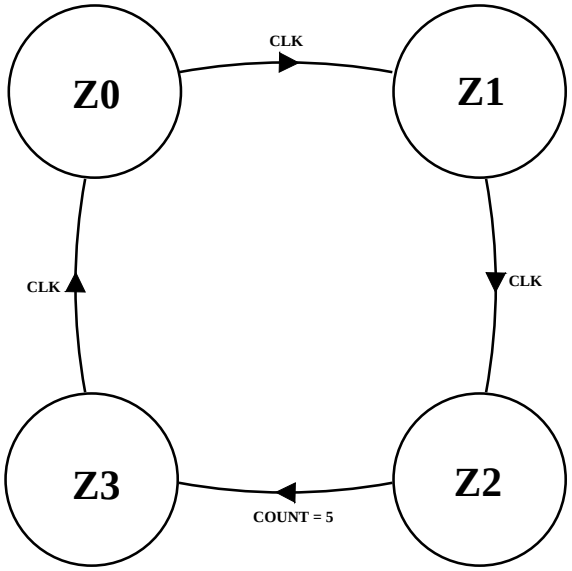


Abbildung 1.11: *Das Zustandsdiagramm der FSM aus Abbildung 1.10.*

Die Änderung der Signale der FSM in Abhängigkeit vom 'CLK'-Signal ist im Diagramm in Abbildung 1.10 (unten) dargestellt. Dort ist auch das Ausgangssignal 'OUTPUT1' des Ausgangsschaltnetzes zu sehen. Abbildung 1.11 zeigt das Zustandsdiagramm der FSM aus Abbildung 1.10. Zustandsdiagramme sind oftmals eine nützliche Hilfe zur Beschreibung der Funktion einer FSM und zu deren Entwurf. Die Zustände der FSM werden darin durch Kreise dargestellt, die Übergänge durch Pfeile die von einem Zustand zum nächsten zeigen.

1.3 Strukturale Modellierung

Die strukturale Modellierung beschreibt die Verdrahtung von Bauteilen oder Baugruppen. Auf diese Weise kann ein VHDL-Entwurf leicht modularisiert werden. Es ist dann einfacher, die funktionellen Gruppen einzeln für sich zu entwickeln und zu testen. Komplexe VHDL-Projekte können damit in einfache Teilschritte gegliedert und z.B. auf mehrere Teams verteilt werden. Abbildung 1.12 zeigt den Aufbau einer strukturalen Beschreibung.

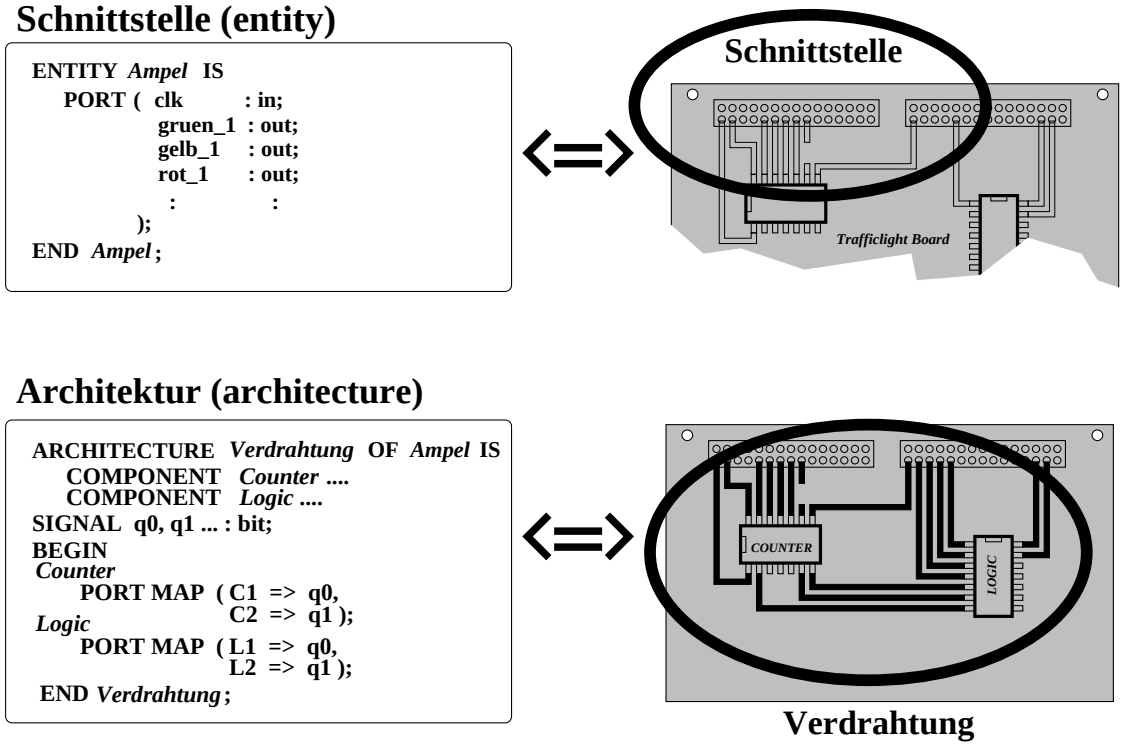


Abbildung 1.12: *Die strukturale Modellierung gestattet die Verdrahtung von Baugruppen. Genauso wie Bauteile auf einer Leiterplatte, können funktionelle VHDL-Modelle auf diese Weise zu komplexen Entwürfen kombiniert werden. Die* ENTITY *der strukturalen Beschreibung entspricht dabei dem Stecker der Leiterplatte. In der* ARCHITECTURE *werden die Leiterbahnen zwischen den Bauteilen gezogen.*

Die ENTITY beschreibt hier die Schnittstelle der Komponentenverdrahtung. Dies entspricht in etwa dem Stecker einer Leiterplatte. In der ARCHITECTURE wird die Verdrahtung der Komponenten (**'Component'**) beschrieben. In einer rein strukturalen Beschreibung gibt es keine funktionellen Strukturen wie z.B. Prozesse. Diese befinden sich dann zumeist in den Components. In vielen Entwürfen kommen jedoch strukturale und funktionelle Konstrukte gemischt vor. Es ist auch oft nicht sinnvoll, beide Modellierungtypen in verschiedene VHDL-Entwürfe zu trennen.

Die ARCHITECTURE einer strukturalen Modellierung beschreibt die Verdrahtung der Components. Dies ist in etwa mit dem Verlegen von Leiterbahnen auf einem PCB vergleichbar. In der ARCHITECTURE werden die ENTITYS der Components miteinander und mit der globalen ENTITY der strukturalen Beschreibung verbunden. Die Verbindungen werden dabei durch 'Signale' im Anweisungsteil der ARCHITECTURE beschrieben.

1.3.1 Components

Das wichtigste Konzept in der strukturalen Modellierung ist das der **'Component'**. Eine Component ist ein vollständiger VHDL-Entwurf mit ENTITY und ARCHITECTURE. Meistens sind Components überwiegend funktionelle Beschreibungen, aus einer vorhergehenden Entwurfsphase, oder aus einer vorgefertigten Bibliothek. Components können natürlich auch ihrerseits strukturale Beschreibungen sein und selbst wieder Components enthalten. Auf diese Weise kann eine vielschichtige Hierarchie von VHDL-Modellen aufgebaut werden wie in Abbildung 1.13 angedeutet.

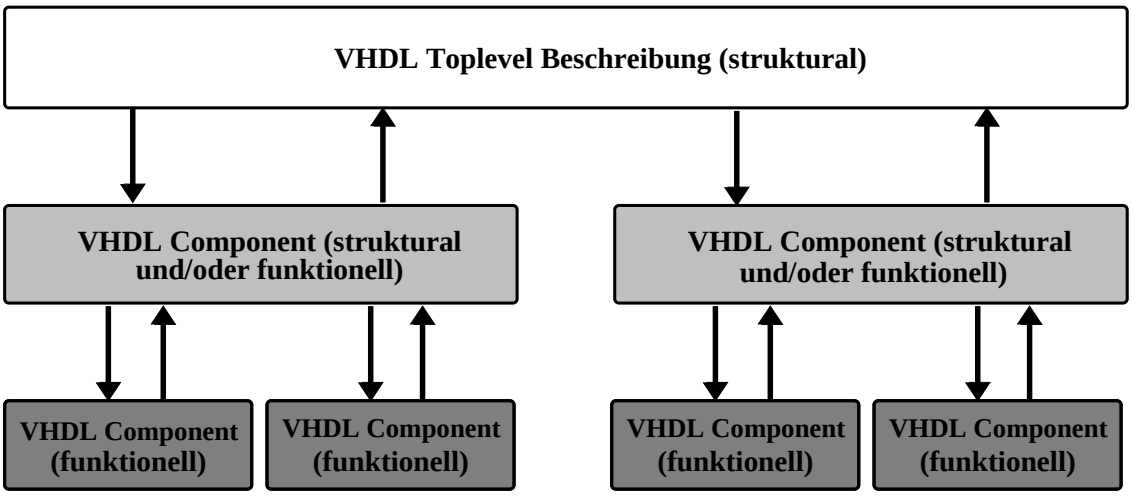


Abbildung 1.13: *Die Hierarchie eines VHDL-Entwurfs. Die unterste Ebene enthält immer funktionelle Entwürfe, während höhere Ebenen zunehmend strukturalen Charakter haben. Der Top-Level Entwurf ist meistens nahezu vollständig struktural.*

Die unterste Hierarchie-Ebene wird in diesem Fall immer funktionell sein, die oberste Ebene vorrangig struktural. Die oberste Ebene wird meist als **'Top Level'** bezeichnet. Während der Synthese ist zu beachten, daß die Hierarchie von unten nach oben synthetisiert/compiliert wird. Eine Component kann in einer höher liegenden ARCHITECTURE erst dann instantiiert werden, wenn sie bereits zuvor synthetisiert/compiliert wurde.

Abbildung 1.14 zeigt ein Beispiel für die ARCHITECTURE einer strukturalen Beschreibung. Zunächst werden die zur Verdrahtung vorgesehenen Components im Deklarations-Teil der ARCHITECTURE mit ihrer jeweils vollständigen ENTITY deklariert. Die hier dargestellten Components *'Zähler'* und *'Logik'* sind der Ampel-Beschreibung aus Abbildung 1.4 entliehen. Der *'Zähler'* generiert darin eine Reihe von Signalsequenzen (Q0, Q1, Q2), die in der *'Logik'*-Component zu den eigentlichen Ampelsignalen (GRUEN_1, GELB_1, ROT_1) kombiniert werden. Die ARCHITECTURE *'Ampel'* dient als Verdrahtungs-Beschreibung der Components.

Zur Verbindung der Entitys der beiden Components werden die internen Signale Q0_INTERN bis Q2_INTERN definiert. Sie dienen im Anweisungsteil als Verbindungsleitungen zwischen dem Ausgang der *'Zähler'*-Component und dem Eingang der *'Logik'-Component*. Die Signalzuweisung geschieht im Anweisungsteil der ARCHITECTURE in einer so genannten **'Port Map'**. Hier werden den Ports von *'Zähler'* und *'Logik'* die internen Signale, sowie die Entity-Ports von *'Ampel'* zugewiesen. Es ist durchaus üblich, für Signale und Ports der strukturalen Beschreibung die gleichen Namen wie in den Entitys der instantiierten Components zu verwenden.

```
entity Ampel is

    port ( CLK        : in std_logic;
           RESET      : in std_logic;
           GRUEN_1    : out std_logic;
           GELB_1     : out std_logic;
           ROT_1      : out std_logic);

end Ampel ;
```

```
architecture Verdrahtung of Ampel is
    component Zähler
        port ( CLK        : in std_logic;
               RESET      : in std_logic;
               Q0         : out std_logic;
               Q1         : out std_logic;
               Q2         : out std_logic );
    end component;

    component Logik
        port ( CLK        : in std_logic;
               RESET      : in std_logic;
               Q0         : in std_logic;
               Q1         : in std_logic;
               Q2         : in std_logic;
               GRUEN1     : out std_logic;
               GELB1      : out std_logic;
               ROT1       : out std_logic );
    end component;

signal Q0_INTERN  : std_logic;
signal Q1_INTERN  : std_logic;
signal Q2_INTERN  : std_logic;

begin
    Ampelzähler: Zähler
        port map    ( CLK     => CLK,
                      RESET   => RESET,
                      Q0      => Q0_INTERN,
                      Q1      => Q1_INTERN,
                      Q2      => Q2_INTERN );
    Ampellogik: Logik
        port map    ( CLK     => CLK,
                      RESET   => RESET,
                      Q0      => Q0_INTERN,
                      Q1      => Q1_INTERN,
                      Q2      => Q2_INTERN,
                      GRUEN1  => GRUNE_1,
                      GELB1   => GELB_1,
                      ROT1    => ROT_1 );

end Verdrahtung;
```

Deklarations−Teil

Anweisungs−Teil

Abbildung 1.14: *Beispiel einer strukturalen Beschreibung. Die im Deklarationsteil der* ARCHITECTURE *deklarierten Components werden im Anweisungsteil mit Hilfe interner Signale (Q0_INTERN bis Q2_INTERN) miteinander und mit der* ENTITY *(CLK, RESET; GRUEN_1, GELB_1, ROT_1) der strukturalen Beschreibung verbunden.*

1.3.2 GENERATE

Mit ***'GENERATE'*** können Components in einer Schleife mehrfach instantiiert werden. Es wäre umständlich, müßte z.B. für einen Zähler die Component 'FlipFlop' N mal von Hand instantiiert werden. Diese Arbeit kann mit der *'GENERATE'*-Anweisung automatisiert werden. *'GENERATE'* arbeitet dabei wie eine 'FOR'-Schleife. Die spezifizierte Component wird so oft instantiiert bis die Abbruchbedingung der Schleife erfüllt ist.

Abbildung 1.16 zeigt den VHDL-Code einer *'GENERATE'*-Umgebung. In der 'FOR'-Schleife wird ein 5 Bit Binärzähler (wenn k=5) generiert, der aus einzelnen FlipFlops aufgebaut ist. Die FlipFlops sind dabei wie in Abbildung 1.15 als eigenständige VHDL-Beschreibung definiert.

```
-- T-FlipFlop
entity FF is
    port ( T        : in std_logic;
           RESET    : in std_logic;
           OUTPUT   : out std_logic);
end FF;

architecture Verhalten of FF is
begin
    flipflop: process (T,RESET) -- sensity List
        variable last_Q  : std_logic;
        variable Q       : std_logic;
    begin
        if RESET = '1' then
            OUTPUT <= '0';
                last_Q := '0';
                    Q := '0';
        else
            if falling_edge (T) then
                if last_Q = '1' then
                    Q := '0';
                    OUTPUT <= '0';
                else
                    Q := '1';
                    OUTPUT <= '1';
                end if;
                last_Q := Q;
            end if;
        end if;
    end process;
end Verhalten;
```

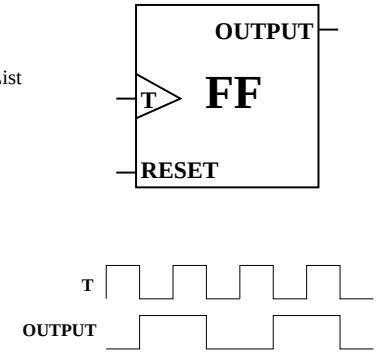
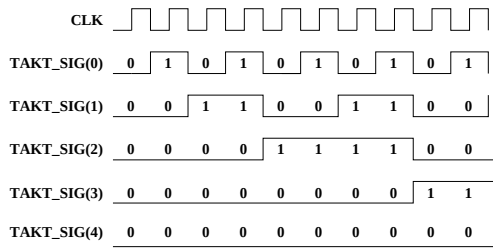


Abbildung 1.15: *Funktionelle Beschreibung eines T-FlipFlop (FF-Component von Abb. 1.16).*

Innerhalb der ARCHITECTURE des Zählers werden die FlipFlops zunächst als Component **'FF'** deklariert. Die Instantiierung erfolgt anschließend im Anweisungsteil der *'GENERATE'*-Umgebung. Dabei ist zu beachten, daß sich in vielen Fällen die erste und die letzte Instantiierung des FF von den mittleren Instantiierungen unterscheidet. (Im hier vorliegenden Fall des Zählers unterscheidet sich im Grunde nur das Eingangs-FlipFlop von den anderen, das Ausgangs-FlipFlop bedarf nicht notwendiger Weise einer Sonderbehandlung.) Das Eingangs-FlipFlop wird direkt vom 'CLK'-Signal getrieben und besitzt keinen Vorgänger. Auch das Ausgangs-FlipFlop erfordert oft eine gesonderte Verdrahtung. Aus diesem Grund teilt sich die *'Generate'*-Umgebung meist in die gezeigten drei Abschnitte.

Mit dem *Generate*-Konstrukt lassen sich auf einfache Weise ganze Kaskaden von Components instantiieren. Dies hat zahlreiche Anwendungen im Bereich von Zählern und Teilern oder im Zusammenhang mit n-Bit breiten Bussen (z.B. Instantiierung von n Comparatoren). Es ist jedoch Vorsicht geboten. In vielen Fällen ist eine funktionelle Beschreibung einfacher und kürzer als die kaskadierte Instantiierung von Components. Der Einsatz ist im Einzelfall abzuwägen.

```
architecture   Beschreibung  of  Zähler  is

   signal  TAKT_SIG          : std_logic_vector (k downto 0);
   signal  CARRY_SIG         : std_logic_vector (3 downto 0);
   signal  RESET_SIG         : std_logic;

component FF
        port ( T            : in std_logic;
               RESET        : in std_logic;
               OUTPUT       : out std_logic
             );

begin

    Teiler: for i in 0 to k generate

        eingang: if i = 0 generate
                 cnt_in: FF
                    port map ( T        =>  CLK,
                               RESET    =>  RESET,
                               OUTPUT   =>  TAKT_SIG(i)
                             );
        end generate;

        innen:   if i > 0 and i < k generate
                 cnt_mitte: FF
                    port map ( T        =>  TAKT_SIG(i-1),
                               RESET    =>  RESET,
                               OUTPUT   =>  TAKT_SIG(i)
                             );
        end generate;

        ausgang: if i = k generate
                 cnt_out: FF
                    port map ( T        =>  TAKT_SIG(k-1),
                               RESET    =>  RESET,
                               OUTPUT   =>  TAKT_SIG(k)
                             );
        end generate;

    end generate;

end  Beschreibung ;
```



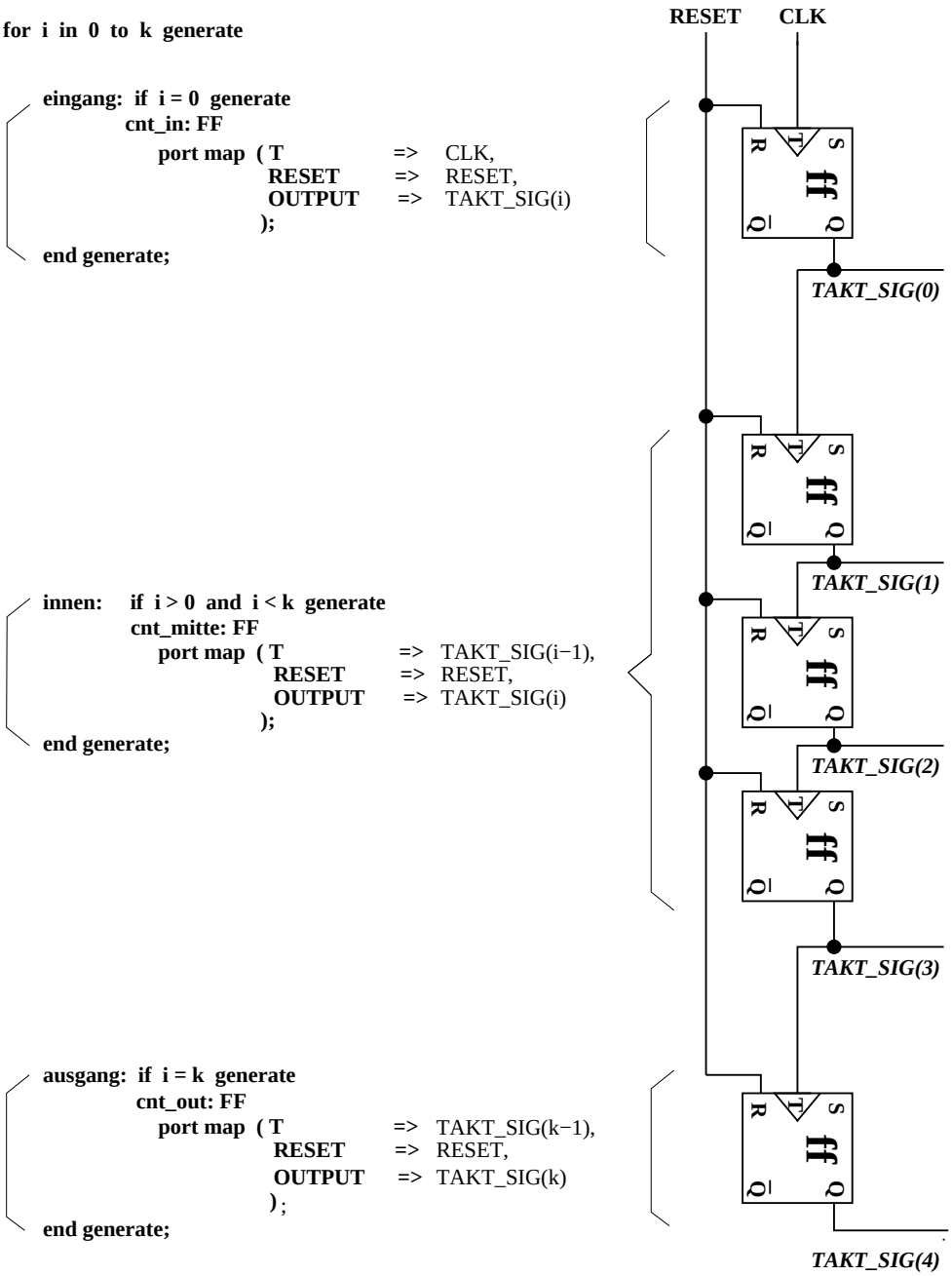


Abbildung 1.16: *Beispiel einer 'Generate'-Umgebung zur Erzeugung eines 5 Bit Zählers (k=5).*

Kapitel 2

Beispiel Ampelsteuerung

Das folgende Kapitel beschäftigt sich mit einer einfachen Ampelsteuerung. Diese dient der weiteren Vertiefung der soeben beschriebenen Techniken von VHDL. Die Ampelsteuerung ist so etwas wie die *Weisse Maus* der Elektronik. Sie erfordert eine einfache Ablaufsteuerung und die Erzeugung der Signal-Sequenzen für die Ampelsignale. Eine typische Aufgabe, wie sie in der Elektronik sehr häufig vorkommt.

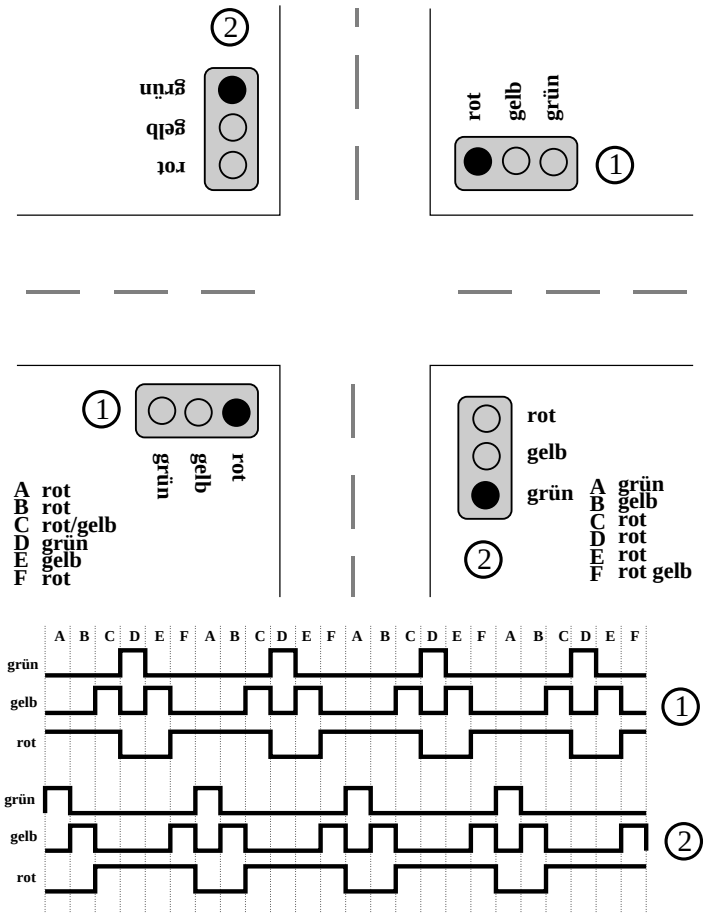


Abbildung 2.1: *Die Ampelanlage einer Verkehrskreuzung. Die diagonal gegenüberliegenden Ampeln sind jeweils zusammengeschaltet. Unten sind die Ampelphasen grafi sch dargestellt. Sie lassen sich in insgesamt 6 Phasen ('A' bis 'F') unterscheiden.*

Abbildung 2.1 zeigt die Ampelkonfiguration einer einfachen Verkehrskreuzung. Sie besteht aus vier Ampeln, von denen jeweils die diagonal gegenüberliegenden Ampeln zusammen geschaltet sind. Das Problem erfordert also nur die Signalerzeugung für zwei Ampeln. In Abbildung 2.1 ist ebenfalls die Signalsequenz[1] für beide Ampeln gezeigt. Diese Sequenzen gilt es nun mit den anschließenden VHDL-Beschreibungen zu erzeugen.

2.1 Ampel Version 1

Als erste Idee könnte man auf den Gedanken verfallen, daß sich die Ampelphasen einfach aus einem CLOCK-Signal[2] erzeugen lassen. Dazu könnte das CLK-Signal mit einem Zähler mehrfach geteilt, und die so erzeugten Signale über logische Gatter miteinander verknüpft werden.

Dies ist in der Tat möglich, wenn die CLK-Signal-Teiler zählstandabhängige Resetsignale erhalten. Abbildung 2.2 zeigt ein einfaches periodisches CLK-Signal, sowie die mit Hilfe eines zweistufigen Zählers erzeugten Steuersignale 'Q1' und 'Q2'. Die Signale 'CLK', 'Q1' und 'Q2' werden dann anschließend durch logische 'AND' und 'OR' miteinander zu den Ausgangssignalen 'GRUEN', 'GELB' und 'ROT' verknüpft.

Die exakte logische Verknüpfung ist in Abbildung 2.3 dargestellt. Die hierzu verwendeten Gatter 'OR' und 'AND' sind in VHDL Bestandteil der Standard-Bibliothek und müssen nicht extra definiert werden.

Das vollständige Listing der VHDL-Beschreibung ist hier anschließend angegeben. Es besteht aus der ENTITY, in der die Ausgangsports der Ampel (GRUEN, GELB, ROT) festgelegt sind, und der ARCHITECTURE, wo das Verhalten der Ampel beschrieben wird. Im Deklarationsteil der ARCHITECTURE werden die Steuersignale der Ampel definiert. Der Anweisungsteil besteht aus einem Prozess und direkten Signalzuweisungen. Durch den Prozess wird die Variation der Steuersignale erzeugt, er besteht aus dem Programm-Zähler der Ampel. Die anschließende direkte Signalzuweisung erzeugt die logische Vernüpfung der Steuersignale zu den Ausgangssignalen der Ampel (wie in Abbildung 2.3 beschrieben), sowie die Zuweisung an die Ausgangsports der ENTITY.

```
-----------------------------
-- Ampelsteuerung,  Version 1
-----------------------------

-- Bibliothek laden
library IEEE;
use IEEE.std_logic_1164.all;  -- Standard Bibliothek



-- Schnittstellen Deklaration
entity AMPEL is
        port (  CLK             : in std_logic;
                RESET           : in std_logic;
                GRUEN_1         : out std_logic;
                GELB_1          : out std_logic;
                ROT_1           : out std_logic;
                GRUEN_2         : out std_logic;
                GELB_2          : out std_logic;
                ROT_2           : out std_logic
              );
end AMPEL;
```

[1] Natürlich handelt es sich hierbei nur um die logische Darstellung der Ampelsequenzen, bei der die absoluten Zeiten für jede Ampelphase ein für eine reale Ampel eher ungünstiges Zeitverhältnis aufweisen

[2] meist als CLK-Signal abgekürzt

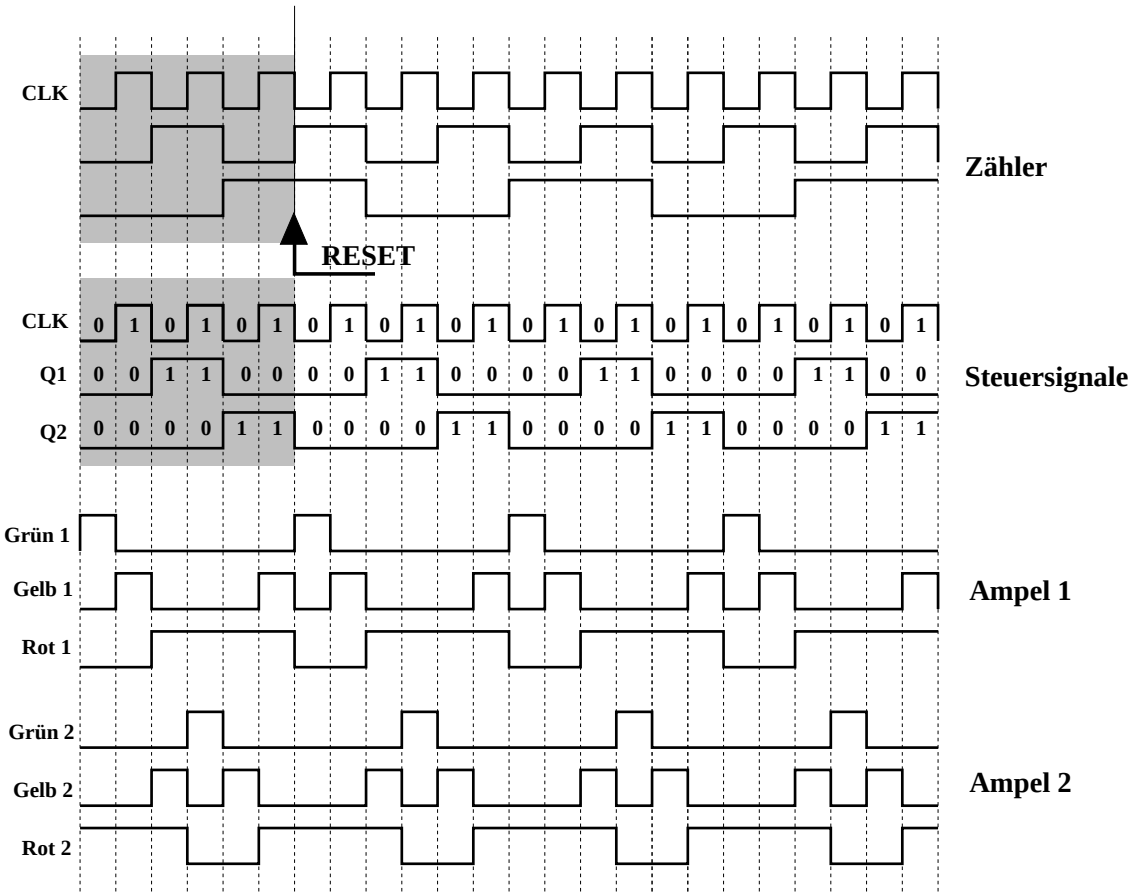


Abbildung 2.2: *Grundlage der Ampelsignale ist ein einfacher Zähler. Es ist möglich, die Ampelsignale aus dem Zählerstand herzuleiten, wenn dieser im richtigen Moment zurückgesetzt wird. Die Reset-Logik sorgt dafür, daß der Grau unterlegte Bereich ständig wiederholt wird.*

Ampel 1:

$$GRUEN\_1 = \overline{CLK} \text{ AND } \overline{Q1} \text{ AND } \overline{Q2}$$

$$GELB\_1 = (CLK \text{ AND } \overline{Q1} \text{ AND } \overline{Q2}) \text{ OR } (CLK \text{ AND } \overline{Q1} \text{ AND } Q2)$$

$$ROT\_1 = Q1 \text{ OR } Q2$$

Ampel 2:

$$GRUEN\_2 = CLK \text{ AND } Q1 \text{ AND } \overline{Q2}$$

$$GELB\_2 = (\overline{CLK} \text{ AND } Q1 \text{ AND } \overline{Q2}) \text{ OR } (\overline{CLK} \text{ AND } \overline{Q1} \text{ AND } Q2)$$

$$ROT\_2 = (\overline{Q1} \text{ AND } \overline{Q2}) \text{ OR } (CLK \text{ AND } \overline{Q1} \text{ AND } Q2) \text{ OR } (\overline{CLK} \text{ AND } Q1 \text{ AND } \overline{Q2})$$

Abbildung 2.3: *Die Ampelsignale können durch einfache logische Verknüpfungen aus dem 'CLK'-Signal und den Steuersignalen 'Q1' und 'Q2' hergeleitet werden.*

```
-- Architekturbeschreibung der Ampel
architecture VERHALTEN of AMPEL is


-- Deklaration der Steuersignale
signal  Q1_SIG          : std_logic;
signal  Q1_BAR_SIG      : std_logic;
signal  Q2_SIG          : std_logic;
signal  Q2_BAR_SIG      : std_logic;
signal  STATE           : integer range 0 to 3;

begin

   -- Ampel Programm Zaehler
   Zaehler: process(CLK)            -- Jede Aenderung von CLK startet den Prozess
   begin
      if RESET = '1' then           -- Alle Signale werden bei Reset auf
         STATE <= 0;                -- einen definierten Wert gesetzt
         Q1_SIG <= '0';
         Q1_BAR_SIG <= '1';
         Q2_SIG <= '0';
         Q2_BAR_SIG <= '1';
      else
         if rising_edge(CLK) then   -- Ausfuehrung bei ansteigender Flanke von CLK
            if STATE < 2 then
               STATE <= STATE + 1;  -- Zaehler
            else
               STATE <= 0;          -- Ruecksetzen bei 2
            end if;
         end if;

         if STATE = 1 then
            Q1_SIG <= '1';          -- Verhalten von Q1 festlegen
            Q1_BAR_SIG <= '0';
         else
            Q1_SIG <= '0';
            Q1_BAR_SIG <= '1';
         end if;

         if STATE > 1 then
            Q2_SIG <= '1';          -- Verhalten von Q2 festlegen
            Q2_BAR_SIG <= '0';
         else
            Q2_SIG <= '0';
            Q2_BAR_SIG <= '1';
         end if;
      end if;
   end process;



   -- Ampel Signal Logik -- Direkte Signalzuweisung --

   -- Zuweisung Ampel 1 --
   GRUEN_1 <= (CLK and Q1_SIG and Q2_BAR_SIG);
   GELB_1  <= (((not CLK) and Q2_BAR_SIG and Q1_SIG) or ((not CLK) and Q1_BAR_SIG and Q
2_SIG));
   ROT_1   <= ((Q1_BAR_SIG and Q2_BAR_SIG) or (Q2_SIG and Q1_BAR_SIG and CLK) or (Q2_BA
R_SIG and Q1_SIG and (not CLK)));

   -- Zuweisung Ampel 2 --
   GRUEN_2 <= ((not CLK) and Q1_BAR_SIG and Q2_BAR_SIG);
   GELB_2  <= ((CLK and Q1_BAR_SIG and Q2_SIG) or (CLK and Q1_BAR_SIG and Q2_BAR_SIG));
   ROT_2   <= (Q1_SIG or Q2_SIG);

end VERHALTEN;
```

Abbildung 2.4 zeigt die Ausgangssignale der Ampelschaltung Version 1 in der Simulation. Im oberen Bereich sind die drei Steuersignale ’CLK’, ’Q1’ und ’Q2’, sowie der Stand des Ampelzählers dargestellt. Im unteren Teil stehen die sechs Ausgangs-Signale ’Gruen_1’, ’Gelb_1’, ’Rot_1’ und ’Gruen_2’, ’Gelb_2’, ’Rot_2’. Der Entwurf beschreibt die Ampel-Signale schon recht gut. Sehr störend wirken sich jedoch die Nadelimpulse (Glitches) in den Ausgangs-Signalen aus. Diese entstehen aus einer Unzulänglichkeit des Entwurfs die sich erst nach der Hardware-Synthese bemerkbar macht.

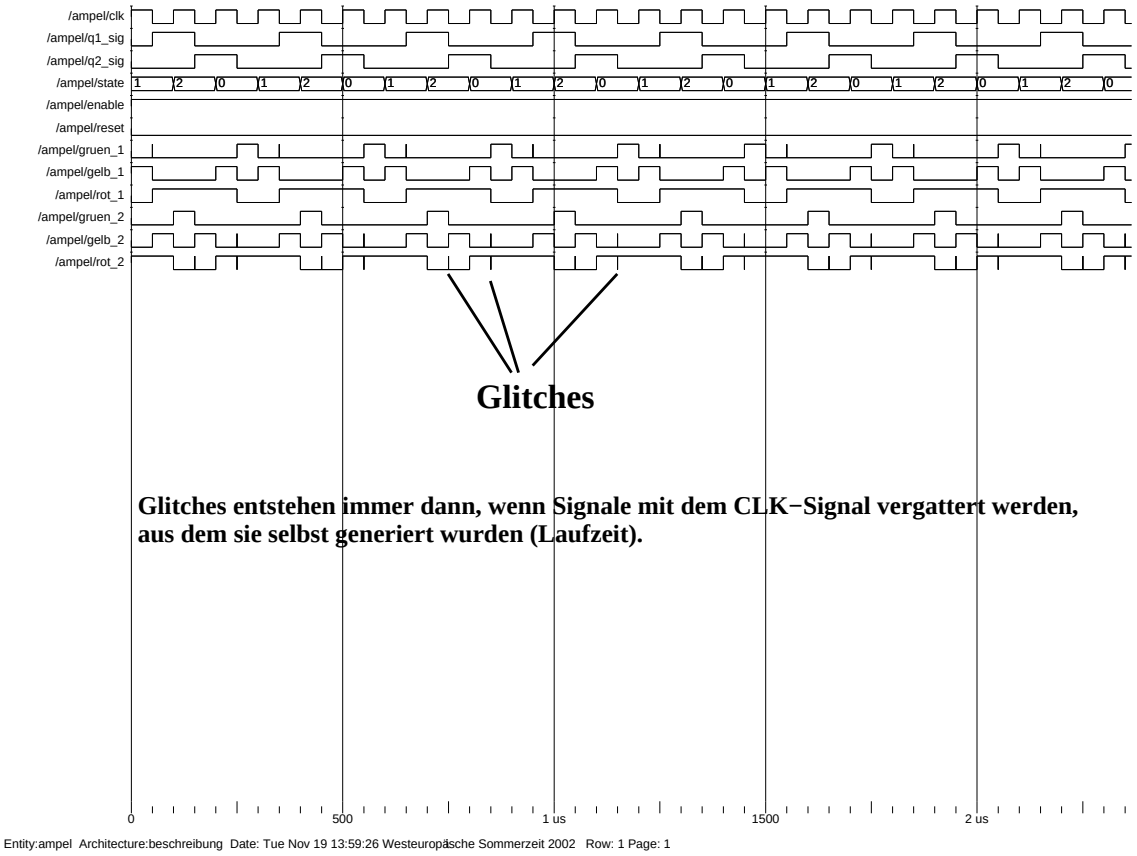


Abbildung 2.4: *Die Ampelsignale bei der Simulation mit einem VHDL-Simulator. Aufgrund von Laufzeitunterschieden kommt es zu störenden Nadelimpulsen (Glitch).*

Obwohl die mathematische Beschreibung in VHDL eine vollkommen korrekte Lösung des Ampelproblems darstellt, ist die physikalische Darstellung in diesem Fall unzulänglich und der Entwurf ist unbrauchbar. Diese Problematik entsteht sehr oft bei der Hardware-Beschreibung mit VHDL. VHDL bietet zunächst eine rein mathematische, also idealisierte Beschreibung der Elektronik. Reale Elektronik ist jedoch zahlreichen physikalischen Rand- und Zwangsbedingungen unterworfen. Zum Beispiel breiten sich reale Signale nur mit Lichtgeschwindigkeit aus, die Makrozellen von FPGAs haben Signallaufzeiten, es existieren parasitäre Kapazitäten die zu endlichen Schaltzeiten führen und so fort. Es liegt im Erfahrungsschatz des Entwicklers, die physikalischen Zwangsbedingungen (constraints) in den Entwurf von vorneherein mit einzubeziehen.

Im vorliegenden Beispiel entsteht das Problem durch die Tatsache, daß die Signale ’Q1’ und ’Q2’ direkt mit dem ’CLK’-Signal logisch verknüpft werden aus dem sie zuvor generiert wurden. Bei der Erzeugung von ’Q1’ und ’Q2’ wurde ein Prozess verwendet, der mit jeder ’CLK’-Flanke erneut startet. Es ist leicht einzusehen, daß dieser Vorgang (Detektion der ’CLK’-Flanke und Abfrage des Zustands von ’STATE’) in Wahrheit nicht unendlich schnell erfolgt. Auf diese Weise entsteht ein Laufzeitunterschied zwischen dem ’CLK’-Signal und den Signalen ’Q1’ und ’Q2’. Die Glitches sind dann die Folge der anschließenden logi-

schen Vernüpfung. Abbildung 2.5 veranschaulicht die Problematik noch einmal grafisch.

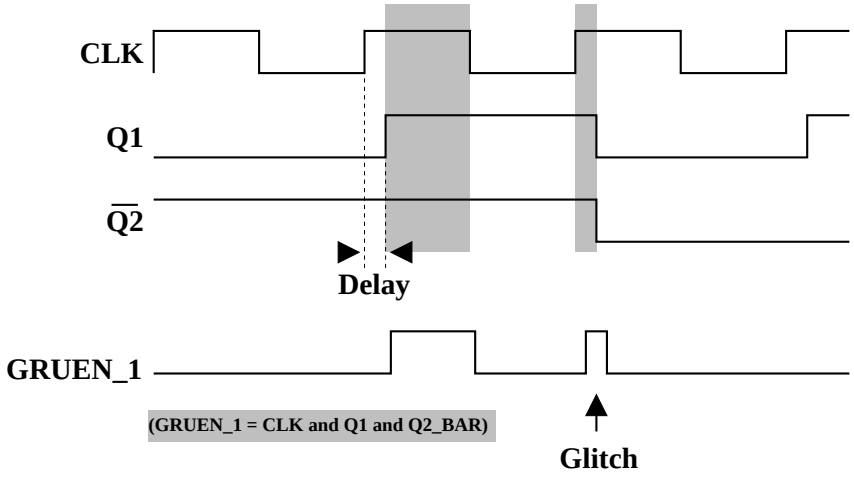


Abbildung 2.5: *Die Steuersignale 'Q1' und 'Q2' sind zum 'CLK'-Signal aufgrund von Gatterlaufzeiten leicht verschoben. Durch die Verknüpfung mit dem 'CLK'-Signal ist die grau unterlegte Bedingung auch an für die Grünphase unbeabsichtigten Stellen (Glitch) erfüllt.*

Eine mögliche erste Abhilfe des Problems wäre es, wenn wir zusätzlich zu 'Q1' und 'Q2' ein weiters Signal 'Q3' aus dem 'CLK'-Signal erzeugen und anschließend 'Q1', 'Q2' und 'Q3' miteinander logisch verknüpfen. Da wir nun alle drei Signale in gleicher Weise aus dem 'CLK'-Signal generiert haben, werden auch alle drei Signale ähnliche Signalverzögerungen aufweisen. Die logische Verknüpfung von 'Q1' bis 'Q3' sollte deshalb schlimmstenfalls zu vernachlässigbar kurzen Glitches führen.

Tatsächlich beschäftigt sich Ampel Version 2 mit einer diesbezüglich verbesserten Version des Ampel Entwurfs 1.

2.2 Ampel Version 2

Ampel Version 2 übernimmt das Grundkonzept von Version 1, jedoch ohne die störenden Glitches in den Ausgangs-Signalen. Die Glitches werden mit zwei Maßnahmen unterdrückt. Erstens wird die Idee von Version 1 aufgegriffen, daß das CLK-Signal nicht selbst logisch verknüpft werden darf. Zweitens werden die Ausgangs-Signale mit dem CLK-Signal gesampelt.

Abbildung 2.6 zeigt das Block-Diagramm der Ampelsteuerung Version 2. In diesem Entwurf werden drei Prozesse im Anweisungsteil der ARCHITECTURE verwendet. Die Prozesse sind über Signale miteinander verbunden, die im Deklarationsteil der ARCHITECTURE deklariert wurden. Der erste Prozess (*Teiler*) teilt das 'CLK'-Signal durch zehn, erzeugt also eine um Faktor zehn tiefere Taktfrequenz (Signal 'TAKT'). Von diesem 'TAKT'-Signal werden im zweiten Prozess (*Zaehler*) die Steuersignale 'Q0', 'Q1' und 'Q2'[3] abgeleitet. Der dritte Prozess (*Logik*) erzeugt wie bei Version 1 die Ausgangs-Signale 'GRUEN_1', 'GELB_1', 'ROT_1' sowie 'GRUEN_2', 'GELB_2, 'ROT_2' durch logische Verknüpfung (vgl. Abb. 2.3 wobei hier 'CLK'='Q0' entspricht). Hierbei werden die Signale diesmal zusätzlich mit dem 'CLK'-Signal gesampelt. Dies bedeutet, daß die Verknüpfung keine statische Verdrahtung darstellt, sondern bei jeder ansteigenden 'CLK'-Flanke von neuem vorgenommen und gespeichert wird. Das Ergebnis wird den Signalen 'GRUEN_1', 'GELB_1', 'ROT_1', sowie 'GRUEN_2', 'GELB_2', 'ROT_2' diesmal

[3] Dies entspricht den Steuersignalen 'Q1', 'Q2' und 'Q3' aus Version 1

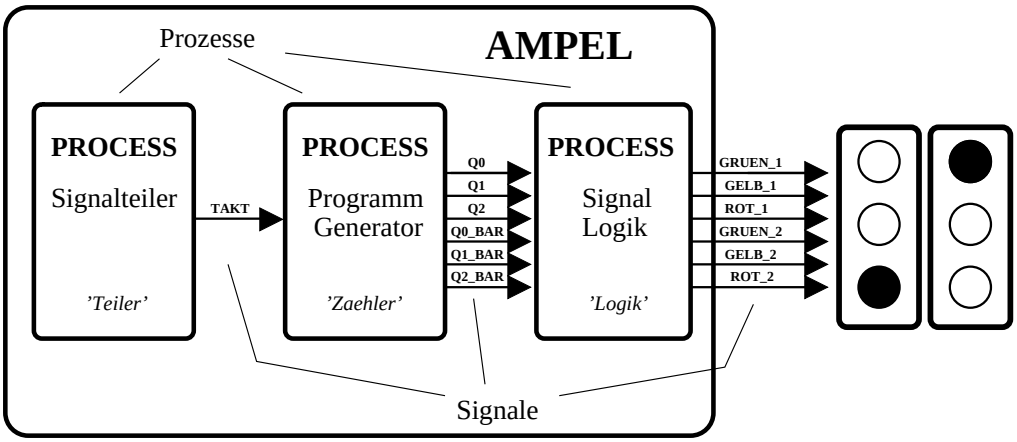


Abbildung 2.6: *Die Ampel besteht hier aus einem funktionellen Entwurf. Die Aufgaben sind in drei separate Prozesse aufgeteilt, die über Signale miteinander verbunden sind.*

über Register zugewiesen. Diese Register sind Kurzzeit-Speicher, die im Augenblick jeder ansteigenden 'CLK'-Flanke neu geschrieben werden. Zwischenzeitlich auftretende Glitches werden somit vollkommen unterdrückt.

Die Abfrage mit steigender 'CLK'-Flanke im Prozess '*Logik*' bietet sich deshalb an, weil die Erzeugung des 'TAKT'-Signales und der Steuersignale 'Q0' bis 'Q2' bei abfallender 'CLK'-Flanke erfolgt ist. Eventuell vorhandene Glitches werden deshalb auch höchstens im Bereich der abfallenden 'CLK'-Flanke auftreten.

Untenstehendes Listing zeigt den VHDL-Code der Ampel Version 2. Die ENTITY ist mit der von Version 1 identisch. Der Anweisungsteil der ARCHITECTURE besteht diesmal aus drei separaten Prozessen, die über Signale miteinander verbunden sind. Der erste Prozess (*Teiler*) besitzt nur das 'CLK'-Signal als Eingang und erzeugt das Signal 'TAKT', das wiederum dem zweiten Prozess (*Zaehler*) als Eingang dient. Der dritte Prozess (*Logik*) verwendet die in Prozess '*Zaehler*' erzeugten Signale 'Q0' bis 'Q2' als Eingang. Sein Ausgang steht in den Registern 'GRUEN_1', 'GELB_1', 'ROT_1', sowie 'GRUEN_2', 'GELB_2', 'ROT_2', die mit den Ausgangs-Ports der ENTITY verbunden sind. Prozess '*Teiler*' wird mit dem 'CLK'-Signal getriggert, Prozess '*Zaehler*' mit dem Signal 'TAKT' aus Prozess '*Teiler*'. Der Ausgangs-Prozess '*Logik*' startet mit jedem 'CLK'-Signal und schreibt bei jeder steigenden Flanke von 'CLK' die Verknüpfung der Steuersignale in die zugewiesenen Register. Die Register sind dauerhaft mit den gleichnamigen Ports der ENTITY verbunden.

```
-----------------------------
-- Ampelsteuerung,  Version 2
-----------------------------

-- Bibliothek laden
library IEEE;
use IEEE.std_logic_1164.all;  -- Standard Bibliothek


-- Schnittstelle deklarieren
entity AMPEL is
        port (  CLK             : in std_logic;
                RESET           : in std_logic;
                GRUEN_1         : out std_logic;
                GELB_1          : out std_logic;
                ROT_1           : out std_logic;
                GRUEN_2         : out std_logic;
```

```
               GELB_2          : out std_logic;
               ROT_2           : out std_logic
            );
end AMPEL;


-- Architektur der Schaltung
architecture VERHALTEN of AMPEL is


-- Deklaration von Signalen
signal  Q0_SIG          : std_logic;    -- Steuersignale
signal  Q0_BAR_SIG      : std_logic;
signal  Q1_SIG          : std_logic;
signal  Q1_BAR_SIG      : std_logic;
signal  Q2_SIG          : std_logic;
signal  Q2_BAR_SIG      : std_logic;
signal  COUNT           : integer range 0 to 10;  -- Zaehler 'Variable'
signal  STATE           : integer range 0 to 6;   -- Zaehler 'Variable'
signal  TAKT            : std_logic;

begin

        -- Prozess - Signalteiler
        Teiler: process(CLK,RESET)   -- Jede Aenderung von CLK startet den Prozess
        begin
           if RESET = '0' then
              COUNT <= 0;
              TAKT <= '0';
           else
              if falling_edge(CLK) then  -- bei jeder fallenden Flanke von CLK ...
                 if COUNT < 10 then      -- Teiler durch 10
                    COUNT <= COUNT + 1;
                    if COUNT < 5 then
                       TAKT <= '1';
                    else
                       TAKT <= '0';
                    end if;
                 else
                    COUNT <= 0;
                 end if;
              end if;
           end if;
        end process;


        -- Prozess - Programm Zaehler
        Zaehler: process(RESET,TAKT)   -- Jede Aenderung von TAKT startet den Prozess
        begin
           if RESET = '0' then
              STATE <= 0;
              Q0_SIG <= '0';
              Q0_BAR_SIG <= '1';
              Q1_SIG <= '0';
              Q1_BAR_SIG <= '1';
              Q2_SIG <= '0';
              Q2_BAR_SIG <= '1';
           else
              if falling_edge(TAKT) then  -- bei jeder fallenden Flanke von TAKT ...
                 if STATE < 5 then        -- Zaehler bis 5
                    STATE <= STATE + 1;
                 else
                    STATE <= 0;
                 end if;
              end if;

              if STATE = 0 or STATE = 2 or STATE = 4 then
```

```
                    Q0_SIG <= '0';
                    Q0_BAR_SIG <= '1';
                 else                              -- Steuersignal Q0
                    Q0_SIG <= '1';
                    Q0_BAR_SIG <= '0';
                 end if;

                 if STATE = 2 or STATE = 3 then
                    Q1_SIG <= '1';
                    Q1_BAR_SIG <= '0';
                 else                              -- Steuersignal Q1
                    Q1_SIG <= '0';
                    Q1_BAR_SIG <= '1';
                 end if;

                 if STATE > 3 then
                    Q2_SIG <= '1';
                    Q2_BAR_SIG <= '0';
                 else                              -- Steuersignal Q2
                    Q2_SIG <= '0';
                    Q2_BAR_SIG <= '1';
                 end if;
              end if;
           end process;


           -- Prozess -  Ampelsignal Logik
           Logik: process(CLK,RESET)         -- Jede Aenderung von CLK startet den Prozess
           begin
              if RESET = '0' then
                 GRUEN_1 <= '1';
                 GELB_1  <= '0';
                 ROT_1   <= '0';
                 GRUEN_2 <= '0';
                 GELB_2  <= '0';
                 ROT_2   <= '1';
              else
                 if rising_edge(CLK) then    -- Register mit jeder steigenden CLK Flanke s
schreiben
                    GRUEN_1 <= (Q0_BAR_SIG and Q1_BAR_SIG and Q2_BAR_SIG);
                    GELB_1  <= ((Q0_SIG and Q1_BAR_SIG and Q2_SIG) or (Q0_SIG and Q1_BAR_S
IG and Q2_BAR_SIG));
                    ROT_1   <= (Q1_SIG or Q2_SIG);
                    GRUEN_2 <= (Q0_SIG and Q1_SIG and Q2_BAR_SIG);
                    GELB_2  <= ((Q0_BAR_SIG and Q2_BAR_SIG and Q1_SIG) or (Q0_BAR_SIG and
Q1_BAR_SIG and Q2_SIG));
                    ROT_2   <= ((Q1_BAR_SIG and Q2_BAR_SIG) or (Q2_SIG and Q1_BAR_SIG and
Q0_SIG) or (Q2_BAR_SIG and Q1_SIG and Q0_BAR_SIG));

                 end if;
              end if;
           end process;

     end VERHALTEN;
```

Abbildung 2.7 zeigt den Screenshot der Ampelsignale bei der Simulation von Version 2. Die Ausgangssignale sind nun frei von Glitches. Abbildung 2.8 zeigt einen vergrößerten Ausschnitt. Deutlich erkennbar ist, wie sich die Signale 'TAKT' und 'Q0' bis 'Q2' mit der fallenden Flanke des 'CLK'-Signals ändern. Die Signale 'GRUEN_1', 'GELB_1', 'ROT_1', sowie 'GRUEN_2', 'GELB_2', 'ROT_2' ändern sich hingegen mit der ansteigenden 'CLK'-Flanke. Durch die zusätzliche Verwendung von Registern, die nur im Moment des 'CLK'-Anstiegs beschrieben werden, können die Glitches von Version 1 vollständig unterdrückt werden.

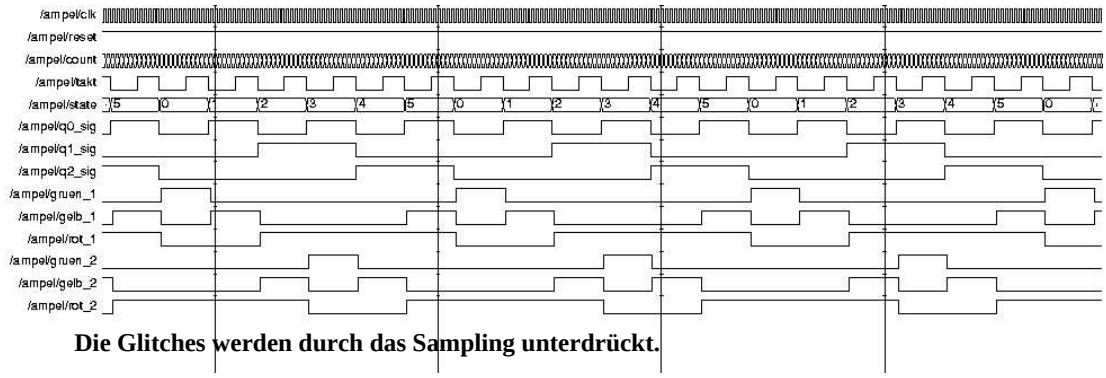


Abbildung 2.7: *Die Abbildung zeigt die Signale des Entwurfs (Version 2) bei der Simulation. Die Glitches können durch das Sampling zuverlässig unterdrückt werden.*

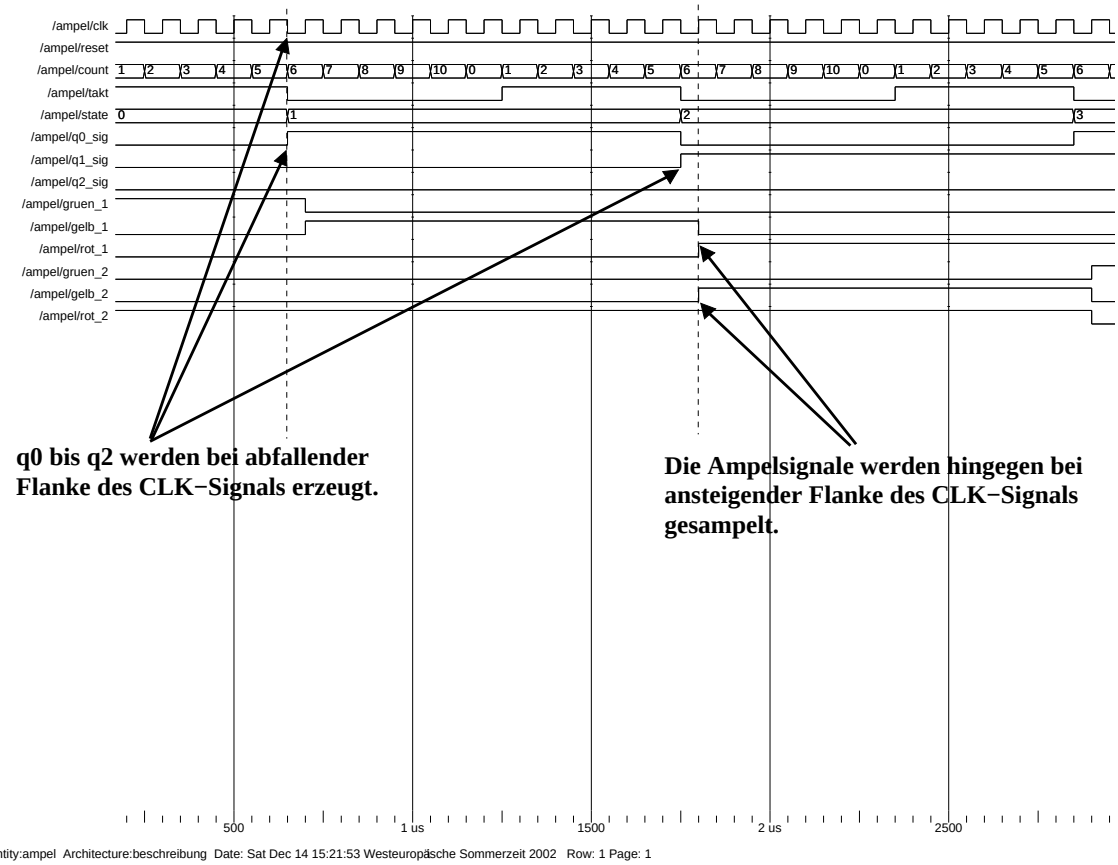


Abbildung 2.8: *Ein weiterer Trick Glitches zu vermeiden, ist das Sampling bei ansteigender 'CLK'-Flanke durchzuführen, während die Signalerzeugung bei abfallender 'CLK'-Flanke geschieht. Laufzeitverschiebungen machen sich so weniger stark bemerkbar.*

2.3 Ampel Version 3

Die Version 3 der Ampelschaltung entspricht logisch exakt der Version 2. Der einzige Unterschied zu Version 2 besteht darin, daß der Entwurf auf mehrere ARCHITECTURES verteilt wird. An diesem Beispiel soll die Verwendung von Components gezeigt werden.

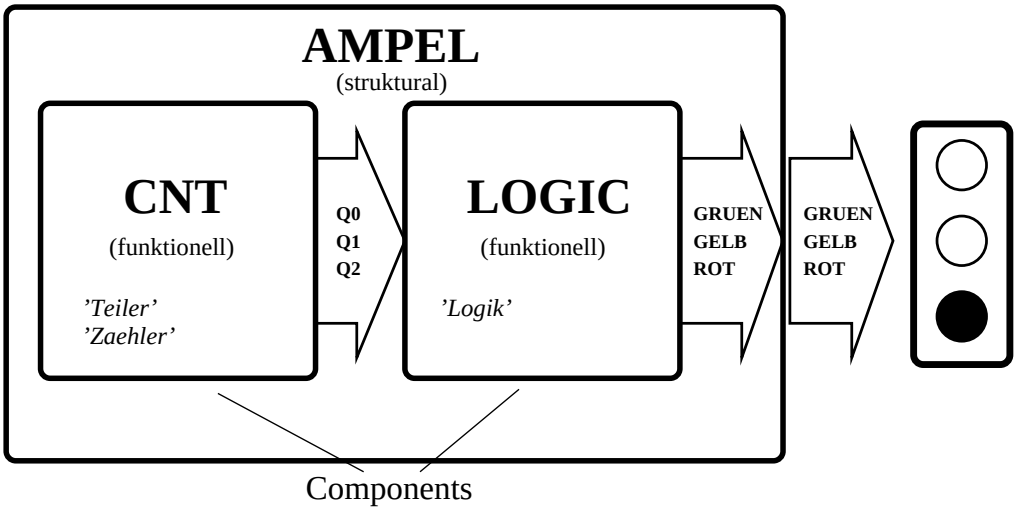


Abbildung 2.9: *Blockdiagramm von Version 3. Die funktionellen Gruppen (Prozesse von Version 2) sind als eigenständige VHDL-Komponenten (Components) ausgeführt. Der Top-Level Entwurf 'AMPEL' ist jetzt weitgehend struktural und dient nur noch der Verdrahtung von 'CNT' und 'LOGIC'.*

Abbildung 2.9 zeigt das Blockdiagramm von Version 3. In Version 2 hatten wir die Ampelschaltung auf die drei Prozesse '*Teiler*', '*Zaehler*' und '*Logik*' der ARCHITECTURE von '*AMPEL*' verteilt. In diesem Beispiel werden wir die Signalerzeugung (Prozess '*Teiler*' und '*Zaehler*') von der Ampellogik (Prozess '*Logik*') trennen, indem wir für beide Aufgaben getrennte ARCHITECTURES verwenden. Die Aufteilung ist in Abbildung 2.9 gezeigt. Die Prozesse '*Teiler*' und '*Zaehler*' werden in der ARCHITECTURE von '*CNT*' zusammengefasst. Der bisherige '*Logik*'-Prozess wird in der ARCHITECTURE von '*LOGIC*' realisiert. Ganz konkret bedeutet dies, daß die Erzeugung der Steuersignale 'Q0' bis 'Q2' und das Sampling der Ausgangs-Signale 'GRUEN_1', 'GELB_1', 'ROT_1', 'GRUEN_2', 'GELB_2', 'ROT_2' nun in zwei vollkommen getrennten Entwürfen (Components) stattfinden wird. Am Schluß werden beide Components dann in der 'Top-Level'-ARCHITECTURE von '*AMPEL*' miteinander verbunden. '*AMPEL*' ist dabei in diesem Fall ein rein strukturaler VHDL-Entwurf, wogegen '*CNT*' und '*LOGIC*' funktionelle VHDL-Entwürfe sind.

Die Trennung in Components bringt oft zahlreiche Vorteile. So können die Components jeweils einzeln für sich entwickelt und getestet werden. Komplexe Aufgaben können leicht in überschaubare Gruppen getrennt werden. Der Aufbau von wiederverwendbaren Modul-Bibliotheken wird ermöglicht und die Entwicklung in unabhängigen Teams erleichtert.

Die drei VHDL-Listings von Version 3 werden unten gezeigt. Es sei zunächst die Component 'CNT' betrachtet. Die Component 'CNT' ist ein eigenständiger VHDL-Entwurf, sie erzeugt die Steuersignale 'Q0' bis 'Q2' für die Ampel-Logikeinheit. Dem entsprechend stehen die Signale 'Q0' bis 'Q2' in der ENTITY von 'CNT' als Output-Ports. Der Vorgang der Signalerzeugung ist identisch wie bereits in Version 2 besprochen. Im Prozess '*Teiler*' wird das 'CLK'-Signal durch zehn geteilt. Das Signal 'TAKT' dient anschließend dem Prozess '*Zaehler*' als Input bei der Generierung der Signale 'Q0' bis 'Q2'. Die Signal-Generierung geschieht wie bei Version 2 bei abfallender Flanke von 'CLK' bzw. 'TAKT'.

```
---------------------------------
-- Zaehler, Ampel Signalgenerator
---------------------------------

library IEEE;
use IEEE.std_logic_1164.all;


-- Schnittstelle des Zaehlers
entity CNT is
   port    ( CLK           : in std_logic;
             RESET         : in std_logic;
             Q0            : out std_logic;
             Q0_BAR        : out std_logic;
             Q1            : out std_logic;
             Q1_BAR        : out std_logic;
             Q2            : out std_logic;
             Q2_BAR        : out std_logic
           );
end CNT;


-- Architektur des Zaehlers
architecture VERHALTEN of CNT is

signal STATE    : integer RANGE 0 to 7;
signal COUNT    : integer RANGE 0 to 10;
signal TAKT     : std_logic;


begin

   Teiler: process(CLK,RESET)                   -- sensitivity list
   begin
      if RESET = '0' then
         COUNT <= 0;
         TAKT  <= '0';
      else
         if falling_edge(CLK) then
            if COUNT < 10 then
               COUNT <= COUNT + 1;
               if COUNT < 5 then
                  TAKT <= '1';
               else
                  TAKT <= '0';
               end if;
            else
               COUNT <= 0;
            end if;
         end if;
      end if;
   end process;



   Zaehler: process(RESET,TAKT)   -- sensitivity list
   begin
      if RESET = '0' then
         STATE <= 0;
         Q0 <= '0';
         Q0_BAR <= '1';
         Q1 <= '0';
         Q1_BAR <= '1';
         Q2 <= '0';
```

```
            Q2_BAR <= '1';
         else
            if falling_edge(TAKT) then
               if STATE < 5 then
                  STATE <= STATE + 1;
               else
                  STATE <= 0;
               end if;
            end if;

            if STATE = 0 or STATE = 2 or STATE = 4 then
               Q0 <= '0';
               Q0_BAR <= '1';
            else
               Q0 <= '1';
               Q0_BAR <= '0';
            end if;

            if STATE = 2 or STATE = 3 then
               Q1 <= '1';
               Q1_BAR <= '0';
            else
               Q1 <= '0';
               Q1_BAR <= '1';
            end if;

            if STATE > 3 then
               Q2 <= '1';
               Q2_BAR <= '0';
            else
               Q2 <= '0';
               Q2_BAR <= '1';
            end if;
         end if;
      end process;


end VERHALTEN;
```

Die zweite Component ist 'LOGIC'. In der ENTITY von 'LOGIC' stehen die Steuersignale 'Q0' bis 'Q2' als Input-Ports. Die Ampelsignale 'GRUEN_1', 'GELB_1', 'ROT_1', 'GRUEN_2', 'GELB_2' und 'ROT_2' erscheinen dagegen als Output-Ports. Ansonsten entspricht das Sampling exakt dem Prozess '*Logik*' aus Version 2. Die Steuersignale werden logisch verknüpft und mit jeder ansteigenden 'CLK'-Flanke in Register gespeichert, die direkt mit den Output-Ports verbunden sind.

```
-------------------------------------------------
-- Ampelsteuerung -- Samplen der Ausgangssignale -
-------------------------------------------------

-- Bibliothek laden
library IEEE;
use IEEE.std_logic_1164.all;


-- Schnittstelle deklarieren
entity LOGIC is
   port (  CLK             : in std_logic;
           RESET           : in std_logic;
           Q0              : in std_logic;
           Q0_BAR          : in std_logic;
           Q1              : in std_logic;
           Q1_BAR          : in std_logic;
```

```
            Q2              : in std_logic;
            Q2_BAR          : in std_logic;
            GRUEN_1         : out std_logic;
            GELB_1          : out std_logic;
            ROT_1           : out std_logic;
            GRUEN_2         : out std_logic;
            GELB_2          : out std_logic;
            ROT_2           : out std_logic
         );
end LOGIC;


-- Architektur der Schaltung
architecture VERHALTEN of LOGIC is


begin

   -- Ampel Signal Logik
   Logik: process(CLK,RESET)
   begin
      if RESET = '0' then
         GRUEN_1 <= '1';
         GELB_1  <= '0';
         ROT_1   <= '0';
         GRUEN_2 <= '0';
         GELB_2  <= '0';
         ROT_2   <= '1';
      else
         if rising_edge(CLK) then
            GRUEN_1 <= (Q0_BAR and Q1_BAR and Q2_BAR);
            GELB_1  <= ((Q0 and Q1_BAR and Q2) or (Q0 and Q1_BAR and Q2_BAR));
            ROT_1   <= (Q1 or Q2);
            GRUEN_2 <= (Q0 and Q1 and Q2_BAR);
            GELB_2  <= ((Q0_BAR and Q2_BAR and Q1) or (Q0_BAR and Q1_BAR and Q2));
            ROT_2   <= ((Q1_BAR and Q2_BAR) or (Q2 and Q1_BAR and Q0) or (Q2_BAR and
Q1 and Q0_BAR));
         end if;
      end if;
   end process;

end VERHALTEN;
```

Wir können die Components 'CNT' und 'LOGIC' wie integrierte Schaltungen betrachten (vgl. Abb. 1.4), die als Komponenten einer Ampelsteuerung miteinander verbunden werden müssen. Die Verbindung liefert in der Elektronik eine Leiterplatte, auf der die Verbindungsleitungen zwischen den Komponenten gezogen werden. In voller Analogie entspricht nun der Top-Level VHDL-Entwurf 'AMPEL' der Leiterplatte. 'AMPEL' stellt dabei einen rein strukturalen Entwurf dar. In ihm werden nacheinander die Components deklariert und danach über Signale miteinander und mit der ENTITY des Top-Level verbunden.

Das untenstehende VHDL-Listing 'AMPEL' ist der Top-Level Entwurf der Ampel. Seine ENTITY enthält als Eingang das 'CLK'-Signal und als Ausgang die Ampelsignale 'GRUEN_1', 'GELB_1, 'ROT_1', 'GRUEN_2', 'GELB_2', 'ROT_2'. Die verwendeten Components werden im Anweisungsteil der ARCHITECTURE über die internen Signalen 'Q0_SIG' bis 'Q2_SIG' miteinander verbunden. Die Signale verbinden den Ausgang der Component 'CNT' mit dem Eingang der Component 'LOGIC'. Zudem werden die Ausgangsports von 'LOGIC' mit den ENTITY-Ports des Top-Levels 'AMPEL' verbunden. Ebendies geschieht mit den Eingangsports 'CLK' und 'RESET' der Components. Sie werden direkt mit den ENTITY-Eingangsports 'CLK' und 'RESET' des Top-Level Entwurfs 'AMPEL' verbunden.

```
-------------------------------------------
-- Ampelsteuerung,  Version 3 -- Top Level --
-------------------------------------------

-- Bibliothek laden
library IEEE;
use IEEE.std_logic_1164.all;


-- Schnittstelle deklarieren
entity AMPEL is
   port (  CLK             : in std_logic;
           RESET           : in std_logic;
           GRUEN_1         : out std_logic;
           GELB_1          : out std_logic;
           ROT_1           : out std_logic;
           GRUEN_2         : out std_logic;
           GELB_2          : out std_logic;
           ROT_2           : out std_logic
        );
end AMPEL;



-- Architektur der Schaltung
architecture STRUKTUR of AMPEL is



-- Deklaration des Signal Generators
component CNT
   port (  CLK     : in std_logic;
           RESET   : in std_logic;
           TAKT    : out std_logic;
           Q0      : out std_logic;
           Q0_BAR  : out std_logic;
           Q1      : out std_logic;
           Q1_BAR  : out std_logic;
           Q2      : out std_logic;
           Q2_BAR  : out std_logic
        );
end component;



-- Deklaration der Signal Logik
component LOGIC
   port (  CLK     : in std_logic;
           RESET   : in std_logic;
           TAKT    : in std_logic;
           Q0      : in std_logic;
           Q0_BAR  : in std_logic;
           Q1      : in std_logic;
           Q1_BAR  : in std_logic;
           Q2      : in std_logic;
           Q2_BAR  : in std_logic;
           GRUEN_1 : out std_logic;
           GELB_1  : out std_logic;
           ROT_1   : out std_logic;
           GRUEN_2 : out std_logic;
           GELB_2  : out std_logic;
           ROT_2   : out std_logic
        );
end component;


-- Deklaration von Signalen
```

```
signal  Q0_SIG          : std_logic;
signal  Q0_BAR_SIG      : std_logic;
signal  Q1_SIG          : std_logic;
signal  Q1_BAR_SIG      : std_logic;
signal  Q2_SIG          : std_logic;
signal  Q2_BAR_SIG      : std_logic;
signal  TAKT_SIG        : std_logic;


begin

   -- Signal Verknuepfung



   -- Aufruf des Signal Generators
   AmpelZaehler: CNT
           port map        ( CLK           => CLK,
                             RESET         => RESET,
                             TAKT          => TAKT_SIG,
                             Q0            => Q0_SIG,
                             Q0_BAR        => Q0_BAR_SIG,
                             Q1            => Q1_SIG,
                             Q1_BAR        => Q1_BAR_SIG,
                             Q2            => Q2_SIG,
                             Q2_BAR        => Q2_BAR_SIG
                           );



   -- Aufruf der Signal Logik
   AmpelLogik: LOGIC
           port map        ( CLK           => CLK,
                             RESET         => RESET,
                             TAKT          => TAKT_SIG,
                             Q0            => Q0_SIG,
                             Q0_BAR        => Q0_BAR_SIG,
                             Q1            => Q1_SIG,
                             Q1_BAR        => Q1_BAR_SIG,
                             Q2            => Q2_SIG,
                             Q2_BAR        => Q2_BAR_SIG,
                             GRUEN_1       => GRUEN_1,
                             GELB_1        => GELB_1,
                             ROT_1         => ROT_1,
                              GRUEN_2       => GRUEN_2,
                             GELB_2        => GELB_2,
                             ROT_2         => ROT_2
                           );

end STRUKTUR;
```

Aufgrund der Unterteilung in mehrere VHDL-Entwürfe ist bei der Simulation und Synthese darauf zu achten, daß die Components zuerst compiliert werden müssen bevor die Top-Level Beschreibung übersetzt werden kann.

Abbildung 2.10 zeigt den vollständigen Entwurf noch einmal schematisch. Die Darstellung ist einem Schaltplan in der Elektronik nicht unähnlich.

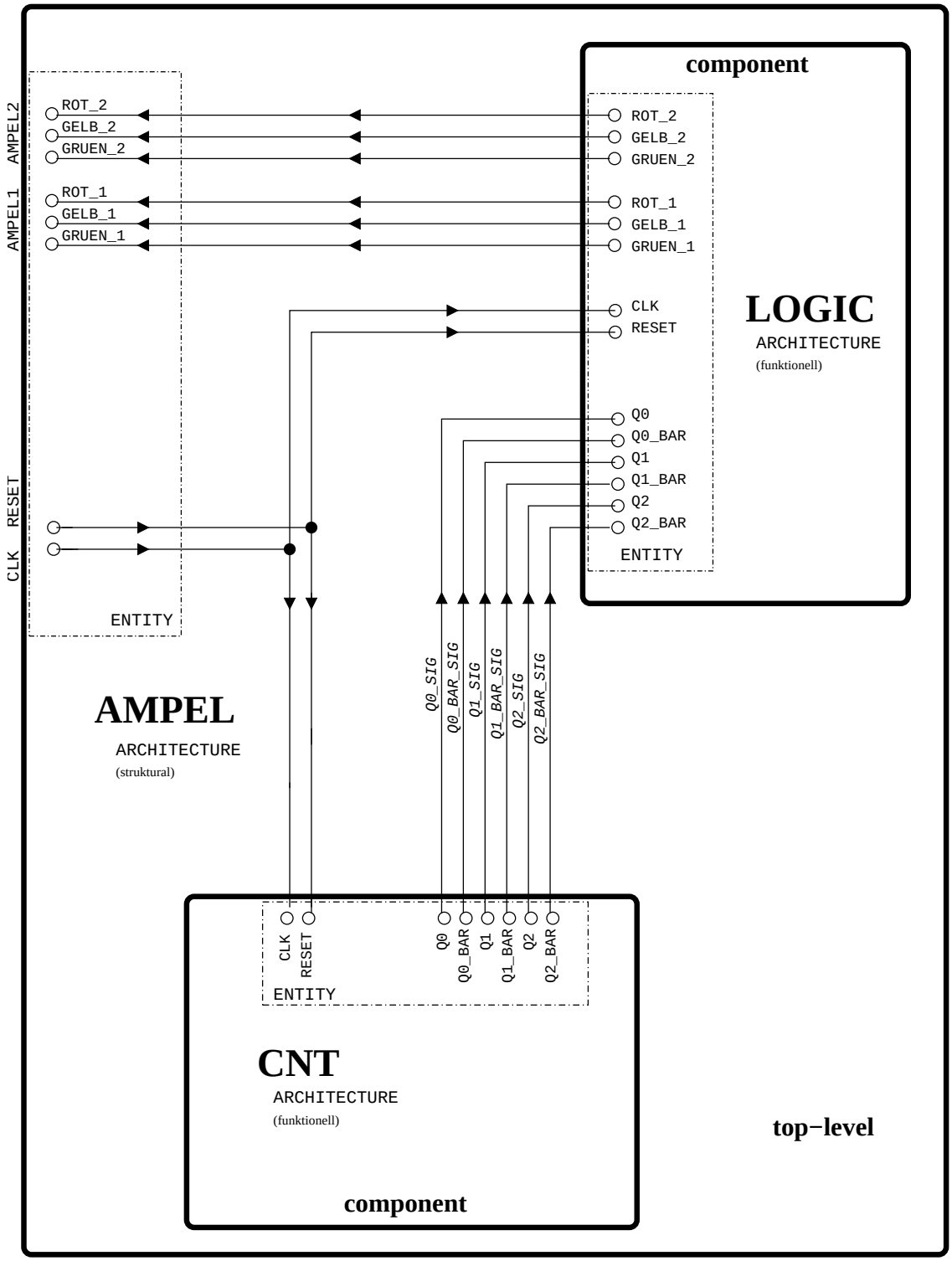


Abbildung 2.10: *Signallaufplan des Entwurfs Version 3. Ganz ähnlich wie auf dem Schaltplan einer Platine sind die Components hier als fertige Baugruppen nur mit ihrer Schnittstelle eingezeichnet. Die Verbindungslinien entsprechen den Signalen des Top-Level VHDL-Entwurfs 'AMPEL'*

2.4 Ampel Version 4

Mit Blick auf Abbildung 2.10 wird noch ein weiterer Vorteil der Verwendung von Components sichtbar. Die Components 'CNT' und 'LOGIC' sind eigenständige Entwürfe und in 'AMPEL' nur über ihre ENTITY sichtbar. Diese Tatsache ermöglicht, daß z.B. die Beschreibung von 'CNT' jederzeit durch eine modifizierte VHDL-Beschreibung ersetzt werden kann, solange nur die ENTITY unverändert bleibt. Dies eröffnet insbesondere die Möglichkeit von Updates, falls sich Beispielsweise die Anforderungen an Gatterverbrauch oder Geschwindigkeit des Entwurfs bei der Synthese verändern.

Ampel Version 4 greift diesen Gedanken auf. Version 4 stimmt in ihrem Aufbau exakt mit Version 3 überein, weshalb die Listings von 'AMPEL' und 'LOGIC' hier auch nicht noch einmal abgedruckt werden. Allerdings wird der Entwurf von 'CNT' ein update erfahren, d.h. wir werden 'CNT' durch einen neuen Entwurf ersetzen, der nur in der ENTITY mit 'CNT' aus Version 3 übereinstimmt.

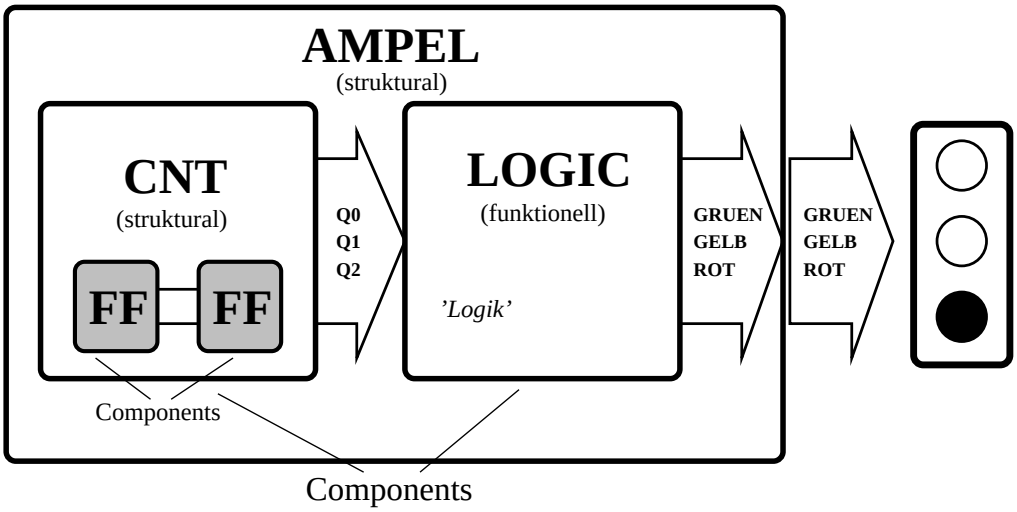


Abbildung 2.11: *Blockdiagramm des Entwurfs Version 4. Die Component 'CNT' ist nun ihrerseits auch aus Components aufgebaut und weitgehend eine strukturale Beschreibung. Solange die Schnittstelle von 'CNT' unverändert bleibt, ist die Funktionalität von 'CNT' weitgehend austauschbar.*

Um den Gedanken der Components weiter zu vertiefen, soll nun auch der Entwurf von 'CNT' aus Components aufgebaut werden. Der Signalgenerator für die Steuersignale 'Q0' bis 'Q2' in 'CNT' kann leicht aus einer Reihe von FlipFlops aufgebaut werden. Abbildung 2.11 zeigt das Blockdiagramm für den Entwurf von Version 4. Der Aufbau ist bis auf den internen Aufbau von 'CNT' mit Version 3 identisch (vgl. Abb. 2.9). Abbildung 2.12 zeigt das Schaltbild für den strukturalen Aufbau von 'CNT'.

Der Zähler kann einfach durch eine Reihe von FlipFlops realisiert werden, wie sie in Abbildung 2.13 dargestellt sind. Die FlipFlops reagieren auf das Signal an ihrem 'TAKT'-Eingang. Springt das Signal von logisch Eins auf Null, so wird das Ausgangssignal 'Q' in den jeweils anderen logischen Zustand geschaltet. Dies entspricht genau einem Teiler um Faktor zwei, d.h. der Ausgang des FlipFlops toggelt genau mit der halben Frequenz wie das Eingangs-Signal 'TAKT'. Mit einer Reihenschaltung mehrerer FlipFlop kann so ein einfacher Zähler aufgebaut werden, wobei die jeweiligen FlipFlop Ausgänge 'Q' die Binärstellen des Zählers sind. Abbildung 2.13 zeigt das Schaltzeichen des FlipFlops und das Verhalten des Ausgangs 'Q' in Relation zum Eingangs-Signal 'T' ('TAKT'). Weiter unten ist die VHDL-Beschreibung des FlipFlops abgedruckt.

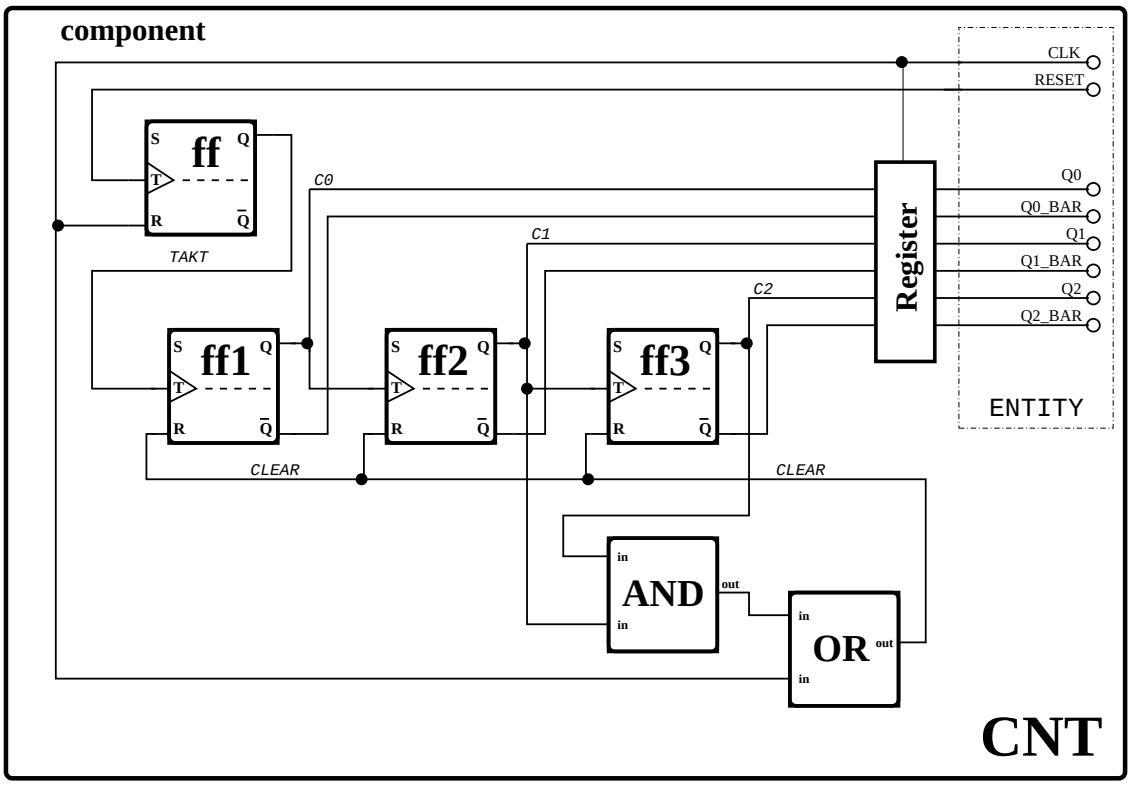


Abbildung 2.12: *Signallaufplan der Component ’CNT’ von Version 4. Der Teiler und der Zähler von ’CNT’ sind als Kette von FlipFlops aufgebaut, die ihrerseits als Components innerhalb von ’CNT’ instantiiert werden. Das ’CLEAR’-Signal wird bei entsprechendem Zählerstand (’C0’ bis ’C2’) durch logische UND- und ODER-Verknüpfung erzeugt und setzt den Zählerstand (’ff1’ bis ’ff3’) zurück auf Null.*

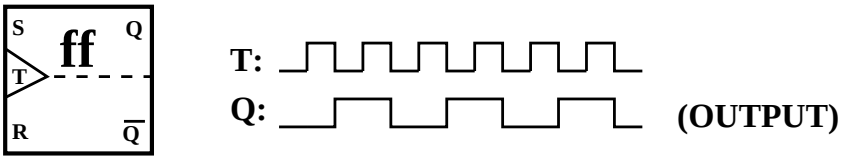


Abbildung 2.13: *Die Component aus der Teiler und Zähler von ’CNT’ aufgebaut sind, ist ein einfaches Toggle-FlipFlop. Jede fallende Flanke am Takteingang ’T’ des FlipFlop schaltet den Ausgang in den jeweils anderen Zustand.*

```
--------------
-- T-FlipFlop
--------------

library IEEE;
use IEEE.std_logic_1164.all;


-- Schnittstelle des Flip Flops
entity FF is
   port    ( T             : in std_logic;
             RESET         : in std_logic;
             OUTPUT        : out std_logic
           );
end FF;


-- Architektur des Flip Flops
architecture VERHALTEN of FF is

begin
   --
   -- Verhaltens Beschreibung des Flip Flops
   --
   flipflop: process(T,RESET)   -- sensitivity list
      variable last_Q  : std_logic;
      variable Q       : std_logic;
   begin
      if RESET = '1' then
         OUTPUT <= '0';
         last_Q := '0';
              Q := '0';
      else
         if falling_edge(T) then
            if last_Q = '1' then
               Q := '0';
               OUTPUT <= '0';
            else
               Q := '1';
               OUTPUT <= '1';
            end if;
            last_Q := Q;
         end if;
      end if;
   end process;

end VERHALTEN;
```

Die strukturale Beschreibung von 'CNT' verwendet nun das obige FlipFlop als Component, die dreimal hintereinander instantiiert wird. Hierdurch entsteht ein einfacher Zähler bis acht. Damit die Steuersignale 'Q0' bis 'Q2' korrekt erzeugt werden, muß der Zähler bei Zählerstand '110' zurückgesetzt und von neuem gestartet werden (vgl. Abbildung 2.14). Hierzu existiert eine Reset-Logik welche den Stand der einzelnen FlipFlop-Ausgänge 'Q' auswertet und damit das Reset-Signal 'CLEAR' erzeugt. Erreicht der Zähler den kritischen Stand '110' (C2 = 1, C1 = 1, C0 = 0), so werden alle FlipFlops gleichzeitig durch 'CLEAR' zurückgesetzt (Zustand '000'). Leider hat das Signal der Reset-Logik (bestehend aus einem AND und einem OR) eine endliche Laufzeit, d.h. 'CLEAR' wird erst einen Augenblick nachdem der Zustand '110' erreicht ist erteilt werden. Diese Verzögerung führt wieder zum Auftreten von Glitches, diesmal auf den Signalen 'C0' bis 'C2'. Abbildung 2.14 verdeutlicht das Auftreten dieser kurzzeitigen Störimpulse. Abhilfe erfolgt hier wie bereits früher bei Version 2. Die Signale 'C0' bis 'C2' werden nicht direkt mit den Ausgangsports 'Q0' bis 'Q2' verbunden, sondern mit dem 'CLK'-Signal in Register gesampelt (vgl. Abb. 2.12).

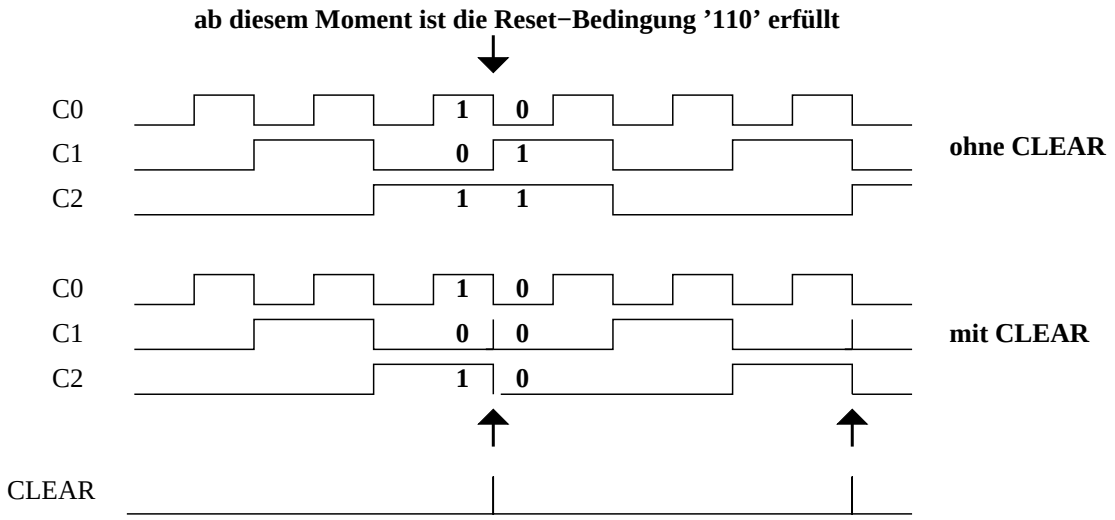


Abbildung 2.14: *Das CLEAR-Signal kann nicht unendlich schnell ermittelt und ausgegeben werden. Es entsteht ein kurzer Moment, bevor das ’CLEAR’-Signal anliegt, in dem der Zähler den inkrementierten Zählerstand anzeigt. Die Laufzeitverzögerung lässt erneut einen Glitch entstehen.*

Das Listing von ’CNT’ ist unten abgedruckt. Die ENTITY stimmt vollkommen mit ’CNT’ von Version 3 überein. Im Deklarationsteil der ARCHITECTURE wird das FlipFlop als Component deklariert. Im Anweisungsteil steht anstatt des Zaehler-Prozesses nun die mehrfache Instantiierung der FlipFlop-Component. Dabei wird zunächst das ’CLK’-Signal durch zwei geteilt was das Signal ’TAKT’ ergibt, das dem Zähler als Eingang dient. Der Zähler selbst besteht aus drei Instanzen der FF-Component, wobei jeder FlipFlop-Eingang mit dem Ausgang des vorhergehenden Zählers verbunden ist. Anschließend werden die FlipFlop-Ausgänge ’C0’ bis ’C2’ Registern zugewiesen (Prozess: ’out_signal’), die dann ihrerseits mit den ENTITY-Ports ’Q0’ bis ’Q2’ von ’CNT’ verbunden sind. Die Signale ’C1’ und ’C2’ dienen zusätzlich der internen Reset-Schaltung als Eingang und erzeugen das Signal ’CLEAR’, mit dem der Zähler auf null zurückgesetzt wird.

```
--------------------------------
-- Zaehler, Ampel Signalgenerator
--------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

-- Schnittstelle des Zaehlers
entity CNT is
   port    ( CLK           : in std_logic;
             RESET         : in std_logic;
             Q0            : out std_logic;
             Q0_BAR        : out std_logic;
             Q1            : out std_logic;
             Q1_BAR        : out std_logic;
             Q2            : out std_logic;
             Q2_BAR        : out std_logic
           );
end CNT;

-- Architektur des Zaehlers
architecture STRUKTUR of CNT is

signal C0           : std_logic;
signal C1           : std_logic;
signal C2           : std_logic;
signal TAKT         : std_logic;
```

```
signal CLEAR        : std_logic;

-- Deklaration des FlipFlops:
component FF
   port ( T      :  in std_logic;
          RESET  :  in std_logic;
          OUTPUT :  out std_logic
        );
end component;

begin
   -- Vorteiler
   FlipFlop_A: FF
            port map ( T      => CLK,
                       RESET  => RESET,
                       OUTPUT => TAKT
                     );


   -- Programm Generator
   FlipFlop_1: FF
            port map ( T      => TAKT,
                       RESET  => CLEAR,
                       OUTPUT => C0
                     );
   FlipFlop_2: FF
            port map ( T      => C0,
                       RESET  => CLEAR,
                       OUTPUT => C1
                     );
   FlipFlop_3: FF
            port map ( T      => C1,
                       RESET  => CLEAR,
                       OUTPUT => C2
                     );


   -- interne Reset Schaltung
   CLEAR <= (C1 and C2) when RESET = '0' else
            '1';


   -- Sample Ausgangssignal, Glitch Unterdrueckung.
   out_signal: process(CLK)
   begin
      if RESET = '1' then
         Q0          <= '0';
         Q0_BAR      <= '1';
         Q1          <= '0';
         Q1_BAR      <= '1';
         Q2          <= '0';
         Q2_BAR      <= '1';
      else
         if rising_edge(CLK) then
            Q0     <= C0;             -- Counter Ausgang mit
            Q0_BAR <= not(C0);        -- Register verbinden.
            Q1     <= C1;
            Q1_BAR <= not(C1);
            Q2     <= C2;
            Q2_BAR <= not(C2);
         end if;
      end if;
   end process;

end STRUKTUR;
```

2.5 Ampel Version 5

Ampel Version 5 ist mit Version 4 identisch. Der einzige Unterschied besteht in der Instantiierung der Components innerhalb von 'CNT'. An dieser Stelle soll die Verwendung der ***'GENERATE'***-Anweisung (vgl. 1.3.2) geübt werden. 'CNT' von Version 4 verwendet drei FlipFlops um den Programm-Zähler zu realisieren. Diese FlipFlops werden als Component deklariert und im Anweisungsteil von 'CNT' einzeln (dreimal) instantiiert.

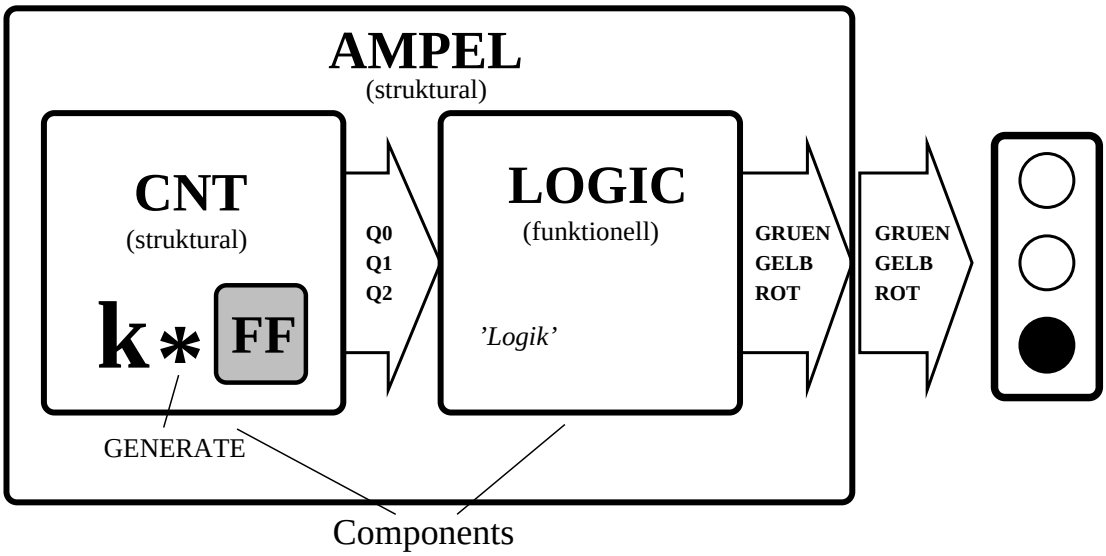


Abbildung 2.15: *Blockdiagramm des Entwurfs von Version 5. Die Component 'CNT' ist bis auf die Instantiierungsmethode der FlipFlops mit Version 4 identisch. In dieser Version werden die FlipFlops in einer 'GENERATE'-Schleife automatisch instantiiert und verschaltet.*

Muß eine Component öfter instantiiert werden, so ist die direkte Instantiierung wie in 'CNT'-Version 4 sehr umständlich, wenn nicht sogar unbrauchbar. Die Instantiierung und Verschaltung von beispielsweise 100 Baugruppen könnte kaum manuell erfolgen. Für diesen Fall ist die *'GENERATE'*-Anweisung gedacht.

Unten stehendes Listing zeigt die VHDL-Beschreibung von 'CNT' aus Version 5. Die Deklaration der FlipFlop-Component in der ARCHITECTURE ist identisch mit Version 4, jedoch wurde bei der Instantiierung der Components im Anweisungsteil nun der *'GENERATE'*-Konstrukt verwendet.

```
---------------------------------
-- Zaehler, Ampel Signalgenerator
---------------------------------
-- mit "generic" und "generate"
library IEEE;
use IEEE.std_logic_1164.all;


-- Schnittstelle des Zaehlers
entity CNT is
   generic ( k             : in positive := 4
           );
```

```
   port    ( CLK           : in std_logic;
             RESET         : in std_logic;
             Q0            : out std_logic;
             Q0_BAR        : out std_logic;
             Q1            : out std_logic;
             Q1_BAR        : out std_logic;
             Q2            : out std_logic;
             Q2_BAR        : out std_logic
           );
end CNT;

-- Architektur des Zaehlers
architecture STRUKTUR of CNT is

signal TAKT   : std_logic_vector (k downto 0);  -- k bit breiter Bus
signal C      : std_logic_vector (2 downto 0);  -- 2 bit breiter Bus
signal CLEAR  : std_logic;

-- Deklaration des Flip Flops:
component FF
   port ( T      :  in std_logic;
          RESET  :  in std_logic;
          OUTPUT :  out std_logic
        );
end component;


begin
   -- Vorteiler
   -- Teiler durch k, generic k ist in ampel.vhd angegeben.

   Teiler: for i in 0 to k generate    -- erzeugt k Teiler

         eingang: if i = 0 generate
                     cnt_in: FF
                         port map ( T      => CLK,
                                    RESET  => RESET,
                                    OUTPUT => TAKT(i)
                                  );
         end generate;

         rest:    if i > 0 generate
                     cnt_rest: FF
                         port map ( T      => TAKT(i-1),
                                    RESET  => RESET,
                                    OUTPUT => TAKT(i)
                                  );
         end generate;

   end generate;



   -- Programm Generator
   Zaehler: for i in 0 to 2 generate

         eingang: if i = 0 generate
                     cnt_in: FF
                         port map ( T      => TAKT(k),
                                    RESET  => CLEAR,
                                    OUTPUT => C(i)
                                  );
         end generate;

         innen:   if i > 0 generate
                     cnt_rest: FF
                         port map ( T      => C(i-1),
```

```
                                    RESET  => CLEAR,
                                    OUTPUT => C(i)
                                  );
        end generate;

    end generate;


    -- interne Reset Schaltung
    CLEAR <= (C(1) and C(2)) when RESET = '0' else
             '1';


    -- Sample Ausgangssignal, Glitches unterdruecken.
    out_signal: process(CLK)
    begin
       if RESET = '1' then
          Q0           <= '0';
          Q0_BAR       <= '1';
          Q1           <= '0';
          Q1_BAR       <= '1';
          Q2           <= '0';
          Q2_BAR       <= '1';
       else
          if rising_edge(CLK) then
             Q0     <= C(0);              -- Counter Ausgang mit
             Q1     <= C(1);              -- Register verbinden.
             Q1_BAR <= not(C(1));
             Q2     <= C(2);
             Q2_BAR <= not(C(2));
          end if;
       end if;
    end process;

end STRUKTUR;
```

Die *'GENERATE'*-Anweisung wird hier an zwei Stellen eingesetzt. Zum einen werden die FlipFlops des Vorteilers mit *'GENERATE'* instantiiert, zum anderen sind es die FlipFlops des Programm Generators. Die Instantiierung des Vorteilers enthält noch eine weitere Neuerung. Die FOR-Schleife der *'GENERATE'*-Anweisung enthält einen variablen Parameter 'k'. Ein solcher Parameter wird in VHDL als ***'GENERIC'*** bezeichnet und ist Bestandteil der ENTITY des Entwurfs. *'GENERIC*'s werden genauso wie Eingangs-Ports behandelt. Ihr Wert kann in der lokalen ENTITY oder im Top-Level des Entwurfs definiert sein und hierarchisch tieferliegenden Entwürfen über deren ENTITY zugewiesen werden (ganz so, als würde es sich um ein Signal handeln).

Der Vorteiler von Version 4 bestand aus einem einzigen FlipFlop (Teiler durch Zwei). Mit dem *'GENERIC'* 'k' besteht nun die Möglichkeit, den Vorteiler nahezu beliebig zu skalieren, wobei sich das Teilerverhältnis von 'CLK' wie 'k^2' verhält. In der ENTITY von 'CNT' haben wir 'k = 4' festgelegt. Das bedeutet, daß innerhalb des *'GENERATE'*-Konstrukts, die FlipFlop-Component 4 mal instantiiert wird. Das Ausgangssignal 'TAKT(k)' ist deshalb 16 mal so lang, wie das 'CLK'-Signal. Auf diese Weise kann die Länge der Ampelphasen auf ein realistisches Zeitmaß (im Sekundenbereich) angepasst werden, selbst wenn die Basisfrequenz des 'CLK'-Signals im MHz-Bereich liegt.

Der Programm Generator kann ebenfalls unter Verwendung von *'GENERATE'* instantiiert werden. Jedoch ist eine Skalierung über *'GENERICS'* in diesem Fall nicht sinnvoll. Die Anzahl der Instanzen ist hier durch die Logik des Programm-Generators vorgeschrieben (vgl. Kapitel 2.4 und Abb. 2.14) und kann nicht variiert werden.

2.6 Ampel Version 6

Ampel Version 6 verfolgt das Konzept der 'Endlichen Zustandsautomaten' aus Kapitel 1.2.2. Es unterscheidet sich dahingehend deutlich von den bisherigen Entwürfen, weshalb hier wieder alle Teile des VHDL-Listings abgedruckt werden. Die Verwendung von Zustandsautomaten ist vermutlich die leistungsfähigste und flexibelste Realisierung für das Problem der Ampelsteuerung, ist die Ampel doch in Wahrheit nichts anderes als ein zeitgesteuerter Zustandsautomat.

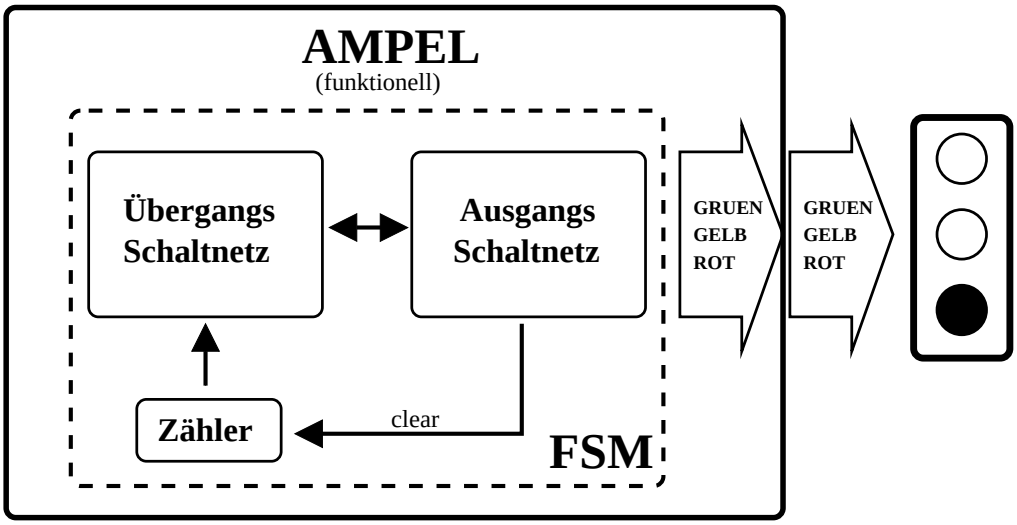


Abbildung 2.16: *Blockdiagramm von Entwurf Version 6. In diesem Entwurf wird die Ampelsteuerung wieder vollständig funktionell beschrieben. Im Übergangsschaltnetz werden die Zustände der Ampel vorangeschaltet, während das Ausgangsschaltnetz in Abhängigkeit des Zustands die Signale der Ampel erzeugt.*

Abbildung 2.16 zeigt das Blockdiagramm der Ampelsteuerung Version 6. Das Übergangsschaltnetz ist zeitgesteuert. Es verwendet das 'CLK'-Signal und einen einfachen Zustandszähler, der mit jeder 'CLK'-Flanke inkrementiert. Hat der Zähler einen bestimmten Zählerstand erreicht, wird er zurückgesetzt und das Übergangsschaltnetz schaltet in den nächsten Zustand. Dieser Vorgang wiederholt sich so oft, bis das Übergangsschaltnetz am Ende wieder den Anfangszustand erreicht hat und der Zyklus von neuem beginnt. Die Zustände des Übergangsschaltnetzes sind in Abbildung 2.17 dargestellt. Für die Ampelsteuerung werden 6 Zustände benötigt, die den Zuständen 'A' bis 'F' aus Abbildung 2.1 entsprechen.

Das Ausgangsschaltnetz reagiert jeweils auf den aktuellen Zustand ('A' bis 'F') des Übergangsschaltnetzes. In Abhängigkeit des aktuellen inneren Zustands werden die Ausgangssignale auf entsprechende Werte geschaltet. Das Ausgangsschaltnetz enthält in diesem Fall auch die Reset-Logik für den Zustands-Zähler und erzeugt das 'CLEAR'-Signal, mit dem der Zähler gelöscht wird.

Das VHDL-Listing (unten) zeigt, wie ein Zustands-Automat in VHDL realisiert werden kann. Für die Zustands-Variable 'SREG0' wird extra im Deklarationsteil der ARCHITECTURE ein Zustands-Typ 'SREG0_TYPE' definiert. 'SREG0' wird anschließend als einzige Instanz dieses Typs angelegt. Der Typ 'SREG0_TYPE' legt alle Zustände fest, welche von den Instanzen dieses Typs angenommen werden können. In diesem Fall sind das die Zustände 'A' bis 'F' der Ampel.

Das Übergangsschaltnetz besteht aus einem Prozess mit einer *'CASE'*-Anweisung. Mit jeder fallenden 'CLK'-Flanke wird 'SREG0' auf seinen aktuellen Zustand geprüft und die entsprechende *'WHEN'* Bedin-

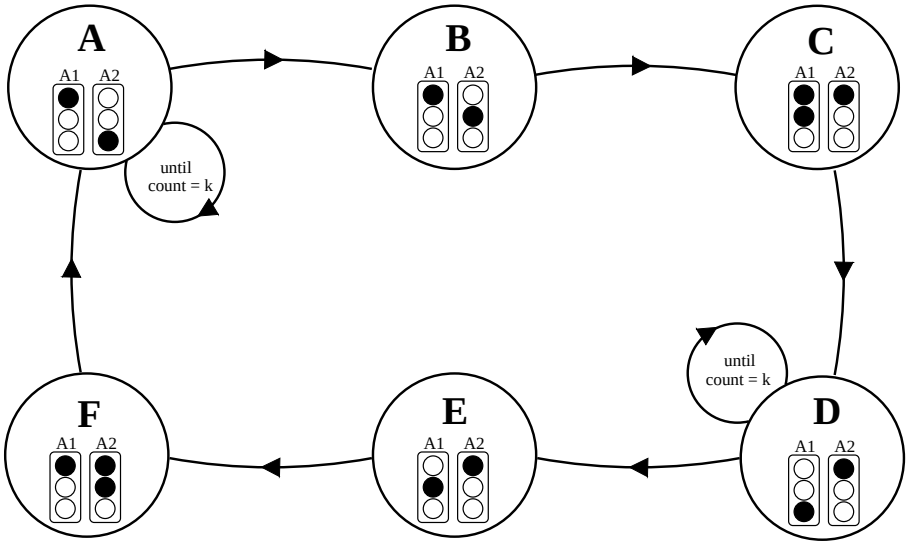


Abbildung 2.17: *Zustandsdiagramm der Ampel Version 6. Die Ampel durchläuft die sechs Zustände 'A' bis 'F'. Die Zustände 'A' und 'D' (Ampelhauptphasen) sind durch einen Zähler auf 'k' 'CLK'-Zyklen verlängert. Alle anderen Zustände sind genau einen 'CLK'-Zyklus lang.*

gung des Prozesses gestartet. Ist 'SREG0' in einem der Zustände 'B', 'C', 'E' oder 'F', so wird sofort auf den nachfolgenden Zustand geschaltet. Dieser wird jeweils mit Beendigung des Prozesses an 'SREG0' zugewiesen. Im Zustand 'A' und 'D' wird mit 'SREG0' zusätzlich auch das Zähler-Signal 'COUNT' ausgewertet. 'COUNT' wird mit jeder fallenden 'CLK'-Flanke inkrementiert. Der Zustandsautomat bleibt deshalb solange im Zustand 'A' bzw. 'D', bis 'COUNT' den Zählstand für die Bedingung 'COUNT = k' erreicht hat. Erst dann wird 'SREG0' in den nachfolgenden Zustand geschaltet. Das Zeitdiagramm des Übergangsschaltnetzes ist in Abbildung 2.18 gezeigt. Die Zustände 'B', 'C', 'E' und 'F' haben die Länge von jeweils einem 'CLK'-Zyklus. Die Zustände 'A' und 'D' sind dagegen 'k' mal ein 'CLK'-Zyklus lang.

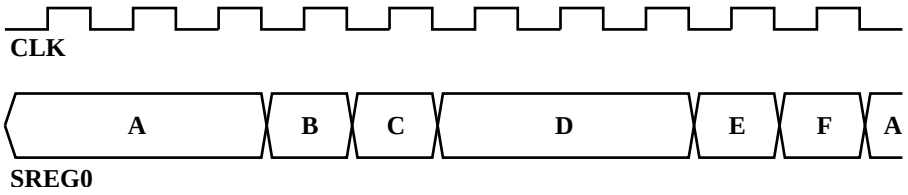


Abbildung 2.18: *Die Ampelhauptphasen (Zustände 'A' und 'D') sind auf 'k' 'CLK'-Zyklen verlängert. Der Parameter 'k' steht als 'GENERIC' in der* ENTITY *und ist ggf. frei programmierbar.*

```
-------------------------------------------
-- Ampelsteuerung, Version 6 -- State Machine
-------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

---
-- entity der Ampel
---
entity AMPEL is
   generic ( k       : positive := 1  -- legt Laenge der Ampelphase fest
           );
```

```
   port    ( CLK     : in std_logic;
             RESET   : in std_logic;
             GRUEN_1 : out std_logic;
             GELB_1  : out std_logic;
             ROT_1   : out std_logic;
             GRUEN_2 : out std_logic;
             GELB_2  : out std_logic;
             ROT_2   : out std_logic
           );
end;

---
-- Architektur der Ampel
---
architecture  STATE_MACHINE of AMPEL is

-- symbolic encoded state machine: SREG0
type sreg0_type is (A, B, C, D, E, F);   -- Ampelzustaende

signal SREG0    : sreg0_type;
signal CLEAR    : std_logic;
signal COUNT    : integer range 0 to 127;

begin
    ---
    -- Zustands Zaehler
    ---
    counter: process (CLK,RESET,CLEAR)
    begin
       if (RESET = '1') or (CLEAR = '1') then
          COUNT <= 0;
       else
          if falling_edge(CLK) then
             COUNT <= COUNT + 1;
          end if;
       end if;
    end process;

    ---
    -- Zustands Generator   -- Uebergangsschaltnetz
    ---
    s_machine: process (CLK)
    begin
       if falling_edge(CLK) then
          if RESET = '1' then
             SREG0 <= A;
          else
             case SREG0 is
                when A =>
                   if COUNT = k then
                      SREG0 <= B;
                   else
                      SREG0 <= A;
                   end if;
                when B =>
                   SREG0 <= C;
                when C =>
                   SREG0 <= D;
                when D =>
                   if COUNT = k then
                      SREG0 <= E;
                   else
                      SREG0 <= D;
                   end if;
                when E =>
                   SREG0 <= F;
                when F =>
```

```
               SREG0 <= A;
            when others =>
               SREG0 <= A;
         end case;
      end if;
   end if;
end process;

---
-- Signal Zustands Zuweisung   -- Ausgangsschaltnetz
---
GRUEN1_assignment:
GRUEN_1 <= '0' when (SREG0 = A) else
           '0' when (SREG0 = B) else
           '0' when (SREG0 = C) else
           '1' when (SREG0 = D) else
           '0' when (SREG0 = E) else
           '0' when (SREG0 = F) else
           '0';

GELB1_assignment:
GELB_1  <= '0' when (SREG0 = A) else
           '0' when (SREG0 = B) else
           '1' when (SREG0 = C) else
           '0' when (SREG0 = D) else
           '1' when (SREG0 = E) else
           '0' when (SREG0 = F) else
           '1';

ROT1_assignment:
ROT_1   <= '1' when (SREG0 = A) else
           '1' when (SREG0 = B) else
           '1' when (SREG0 = C) else
           '0' when (SREG0 = D) else
           '0' when (SREG0 = E) else
           '1' when (SREG0 = F) else
           '0';

GRUEN2_assignment:
GRUEN_2 <= '1' when (SREG0 = A) else
           '0' when (SREG0 = B) else
           '0' when (SREG0 = C) else
           '0' when (SREG0 = D) else
           '0' when (SREG0 = E) else
           '0' when (SREG0 = F) else
           '0';

GELB2_assignment:
GELB_2  <= '0' when (SREG0 = A) else
           '1' when (SREG0 = B) else
           '0' when (SREG0 = C) else
           '0' when (SREG0 = D) else
           '0' when (SREG0 = E) else
           '1' when (SREG0 = F) else
           '1';

ROT2_assignment:
ROT_2   <= '0' when (SREG0 = A) else
           '0' when (SREG0 = B) else
           '1' when (SREG0 = C) else
           '1' when (SREG0 = D) else
           '1' when (SREG0 = E) else
           '1' when (SREG0 = F) else
           '0';


---
```

```
        -- Zaehler Reset
        ---
        CLEAR_assignment:
        CLEAR   <= '0' when (SREG0 = A) else
                   '1' when (SREG0 = B) else
                   '1' when (SREG0 = C) else
                   '0' when (SREG0 = D) else
                   '1' when (SREG0 = E) else
                   '1' when (SREG0 = F) else
                   '1';

end STATE_MACHINE;
```

Die Variable 'k' ist genau wie bei Version 5 als *'GENERIC'* in der ENTITY des Entwurfs definiert. Mit 'k' kann das Längenverhältnis zwischen den Phasen programmiert werden. Der Einfluß von 'k' auf das Tastverhältnis der Ampelsignale wird ebenfalls in Abbildung 2.19 deutlich. Dort sind die Ampelsignale für 'k = 1' und 'k = 2' dargestellt.

Das Ausgangsschaltnetz von Version 6 besteht aus einer Reihe einfacher Signalzuweisungen. Den Ausgangssignalen 'GRUEN_1', 'GELB_1', 'ROT_1', und 'GRUEN_2', 'GELB_2', 'ROT_2' werden dort in Abhängigkeit von 'SREG0' die entsprechenden Werte zugewiesen.

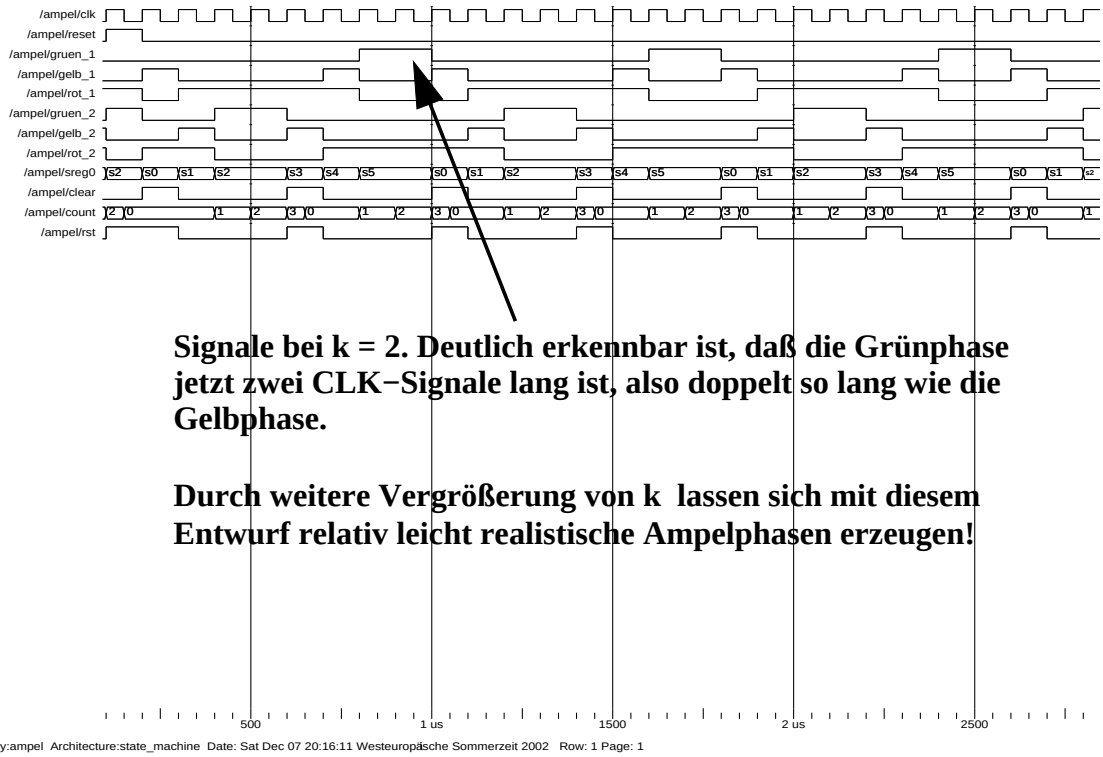


Abbildung 2.19: *Screenplot der Simulation von Version 6 mit 'k = 2'. Deutlich erkennbar ist, daß die Grünphase hier doppelt so lang ist wie bei den bisherigen Entwürfen. Mit Hilfe von 'GENERICS' kann die FSM so aufgebaut werden, daß die Schaltzeiten einer realitätsgetreuen Ampelsteuerung entstehen.*

Kapitel 3

Das DIGILAB

3.1 El Camino picoMAX DIGILAB

Das **picoMAX DIGILAB** ist ein preisgünstiges Prototyping-Board für die ALTERA CLPD's der MAX 3000 und MAX 7000 Familien. Es bietet neben der CPLD-Programmierung auch die Möglichkeit zum experimentellen Betrieb direkt auf dem Board. Hierzu kann optional ein Clock-Oszillator, sowie ein Satz von Buchsenleisten bestückt werden, über den alle Pins des CPLD zugänglich sind. Das DIGILAB besitzt zwei Sockeltypen, somit ist der Einsatz von Bausteinen mit 44- und 84-poligen PLCC Gehäusen möglich. Die Betriebsspannung kann entweder auf 3.3 Volt oder auf 5 Volt eingestellt werden (vgl. die Beschreibung des DIGILAB von El Camino).

Zur Programmierung wird das DIGILAB mit dem mitgelieferten Kabel an die parallele Schnittstelle des Computers und mit dem Steckernetzteil an die Netzversorgung angeschlossen. Die Programmierung erfolgt über das auf dem Board integrierte JTAG-Interface und dauert in der Regel nur wenige Sekunden. Es ist unbedingt darauf zu achten, daß am DIGILAB der für den verwendeten CPLD-Typ korrekte Spannungswert gejumpert ist (vgl. die Anleitung des DIGILAB). Für den anschließenden Betrieb des CPLD auf dem DIGILAB sollte das Programmierkabel entfernt werden. In einigen Fällen zeigten sich im Betrieb Instabilitäten, solange das Programmierkabel noch angeschlossen war. Die erreichbare Betriebsfrequenz richtet sich nach der Komplexität des Entwurfs innerhalb des CPLD. Es ist in der Regel leicht möglich Schaltungen mit Taktfrequenzen bis zu 50 MHz zu realisieren. Weniger komplexe Entwürfe arbeiten auch bei 100 MHz noch einwandfrei. Es empfiehlt sich, für den Clock-Oszillator einen DIL-Sockel vorzusehen. Auf diese Weise kann der Oszillator bei Bedarf gewechselt werden. Von uns wurden vorzugsweise die CPLD-Typen EPM7128AELC84-10 (3.3. Volt) und EPM7128SLC84-10 (5 Volt) bei Clock-Frequenzen zwischen 1 MHz und 50 MHz verwendet. Das DIGILAB, sowie die angegebenen CPLD's sind z.B. bei der Firma CONRAD Elektronik erhältlich.

Tabelle 3.1 gibt einen Überblick der verwendbaren CPLD-Typen von Altera und deren interne Kapazität.

3.2 Altera CPLD

Ein CPLD[1] enthält ein frei programmierbares Logik-Array, vergleichbar einem Gitter mit frei konfigurierbaren Logikbausteinen auf den Gitterplätzen. In gewisser Weise sind CPLD's somit RAM und ROM Bausteinen ähnlich. Bei diesen sind Speicherzellen gitterförmig angeordnet und können über ein Programmiergerät in die logischen Zustände 0 oder 1 versetzt werden. In ROM Bausteine wird die Informationen dauerhaft eingeprägt, bei RAM's ist der Prozess reversibel. Im Grunde stellt der CPLD nun eine Erweiterung dieser Idee dar. Anstatt frei programmierbaren Speicherzellen, besitzt ein CPLD im Prinzip frei

[1] Complex Programable Logic Device

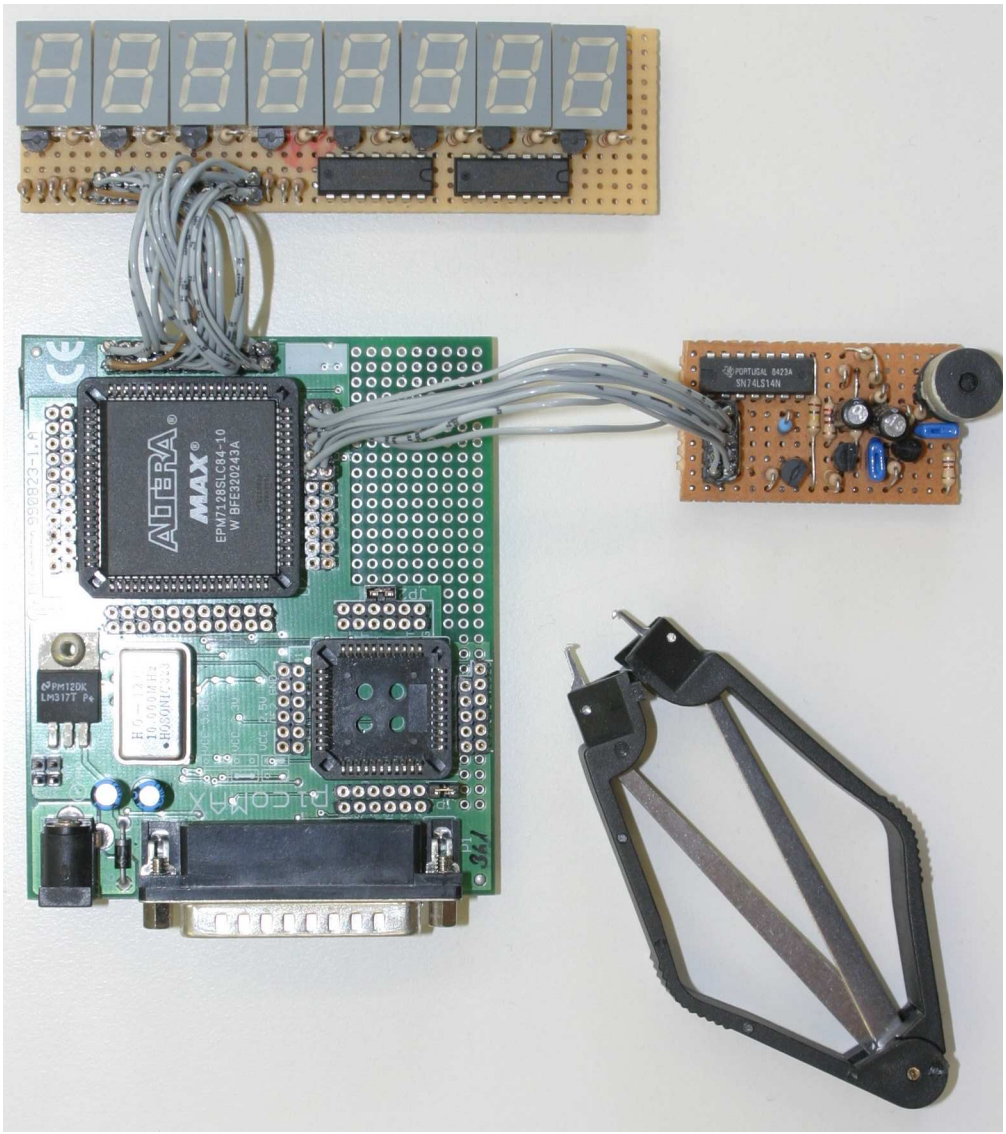


Abbildung 3.1: *Das* DIGILAB *von El Camino: Zusätzlich mit Oszillator und Buchsenleisten bestückt, wird aus dem CPLD-Programmierboard im Handumdrehen ein einfaches Prototyping System. Der CPLD kann direkt auf dem Board betrieben werden. Über Buchsenleisten lassen sich periphere Schaltungen anschließen.*

Familie	Baustein	Gehäuse	User I/O-Pins	Anzahl der Makrozellen
MAX3000	EPM3032	PLCC 44	34	32
MAX3000	EPM3064	PLCC 44	34	64
MAX7000	EPM7032	PLCC 44	36	32
MAX7000	EPM7064	PLCC 44	36	64
MAX7000	EPM7064	PLCC 84	68	64
MAX7000	EPM7096	PLCC 84	64	96
MAX7000	EPM7128	PLCC 84	68	128
MAX7000	EPM7160	PLCC 84	64	160

Tabelle 3.1: *Die für das* DIGILAB *geeigneten CPLD's der MAX3000 und MAX7000 Familie.*

programmierbare Logikschaltungen an den Gitterstellen. Diese können in weiten Bereichen konfiguriert und zu komplexen Logikgebilden kombiniert werden. Auf diese Art lassen sich mittels eines Programmiergerätes auch aufwendige Logikschaltungen innerhalb des CPLD's erzeugen, für die sonst dutzende oder hunderte digitale IC's notwendig wären.

Die kleinste programmierbare Logikzelle der ALTERA MAX Technologie heißt **Makrozelle**. Jede Makrozelle besteht aus ca. 20 logischen Gattern, jeweils 16 Makrozellen sind ihrerseits zu einem Block zusammengefaßt. Die Blöcke sind über ein programmierbares Bussystem miteinander und mit den Anschlüssen des CPLD verbunden.

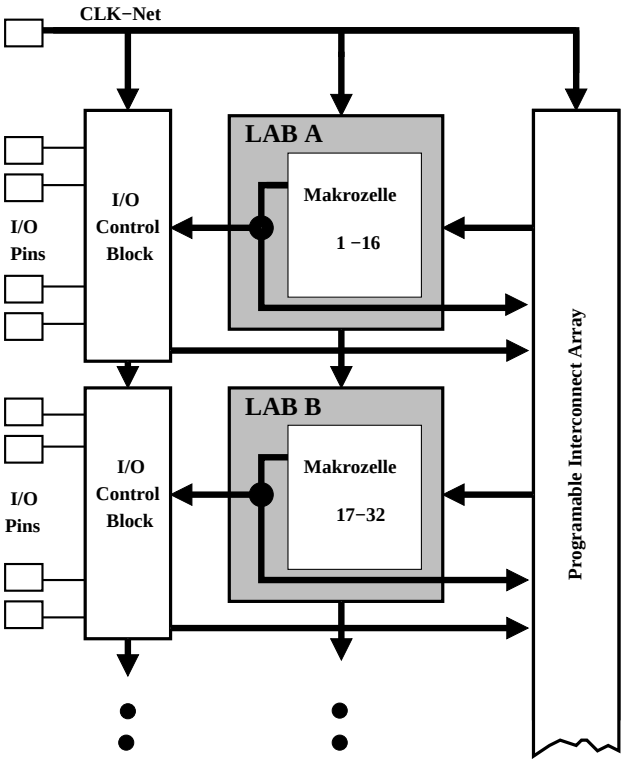


Abbildung 3.2: *Innere Struktur der Altera CPLD's*

Für die Synthese der Hardware ist die Anzahl der belegten Makrozellen entscheidender als die Anzahl der belegten Gatter. Benötigt der Entwurf mehr freie Makrozellen als der CPLD Typ besitzt, so kann

er nicht synthetisiert werden, und das vielleicht obwohl in den bereits verwendeten Makrozellen noch freie Gatter verfügbar sind. Auch der Place&Route-Vorgang kann auf die Anzahl der Makrozellen Einfluß nehmen. Werden beim Place&Route in der Synthese Randbedingungen vorgegeben, so kann sich dies unter Umständen auf die Anzahl der belegten Makrozellen negativ auswirken. In kritischen Fällen empfiehlt es sich manchmal die Vorgabe einer konkreten Pinzuweisung aufzuheben.

Kapitel 4

Altera Quartus II Web Edition Software

Quartus II von Altera ist eine All-in-One-Lösung zur Entwicklung und Synthese digitaler Applikationen für die Altera eigenen CPLD und FPGA Familien. Quartus II ist für den professionellen Einsatz gedacht. Quartus II Web Edition ist eine kostenlose entry-level Version von Quartus II mit den folgenden Merkmalen:

- Design Entry: Schaltplaneingabe (Block und Symbol Editor) oder Hardwarebeschreibungssprachen (VHDL/Verilog/AHDL)
- Synthese: HDL-Compiler und Technologieabbildung für Altera FPGA Technologie
- Fitter: Place & Route Optimierer. Der Fitter versucht den Entwurf auf Platz oder Geschwindigkeit (tradeoff) zu optimieren.
- Assembler: Erzeugung des Bit-Files für den zuvor spezifizierten FPGA Typ der MAX 3000 oder MAX 7000 Familie.
- Programmer: Der Programmer ermöglicht den upload des Bitfiles auf das DIGILAB Entwicklungsboard. Zuvor muß die Pinzuweisung festgelegt werden.
- Simulator: funktionelle-Simulation oder Timing-Simulation des Entwurfs. Zur Erzeugung von Signal-Eingangsstimuli steht ein WaveForm-Editor zur Verfügung.

4.1 Installation

Die Quartus II Web Edition Software kann von der Altera Web Site (www.altera.com) herunter geladen werden. Zum Betrieb ist ein Lizenzfile erforderlich. Die Lizenz für Windows kann auf der Altera Web-Site kostenlos beantragt werden. Zur Registrierung muß entweder die NIC-Nummer der Netzwerk-Karte oder die Hardware ID der C: Festplatte angegeben werden (siehe unten).

4.1.1 Download

- Quartus II Web Edition Software Download: www.altera.com -> Download -> Quartus II Web-Edition -> Single Download File Version (144.7 MByte) -> Registrierung ausfüllen (Company=privat) -> Download starten.
- Quartus II Web Edition Lizenz download: www.altera.com -> Download -> Quartus II Web Edition -> Get a license file -> Registrierung ausfüllen (NIC Number der Netzwerk-Karte eingeben ->

DOS Shell und Befehl: ***ipconfig /all***. Falls keine Netzwerk-Karte vorhanden ist, kann auch die ID der Festplatte verwendet werden -> DOS Shell und Befehl: ***dir /p***) -> Lizenzdaten abschicken -> Das Lizenzfile wird von Altera für gewöhnlich nach wenigen Sekunden an die angegebene email-Adresse zugeschickt. Es können beliebig viele Lizenzen beantragt werden.

4.1.2 Installation

- Zielverzeichnis einrichten, z.B: `C:\altera\quartus2.`
- Quartus II Web Edition Software Installation: Das Quartus II exe Installations file ist selbstextrahierend und kann einfach durch anklicken im Windows Explorer gestartet werden. Danach kann das oben gewählte Zielverzeichnis für die Installation angegeben werden.
- Verzeichnis für Lizenzfile erzeugen, z.B: `C:\altera\licenses.`
- Quartus II Web Edition Lizenz installieren: Die Lizenz in das oben angelegte Verzeichnis kopieren. Anschließend die Quartus II Web Edition Software starten und im Menü ***Tools*** den Punkt ***License Setup ..*** starten. Im Lizenz Setup Manager den Pfad zum Lizenzfile angeben und mit OK bestätigen.

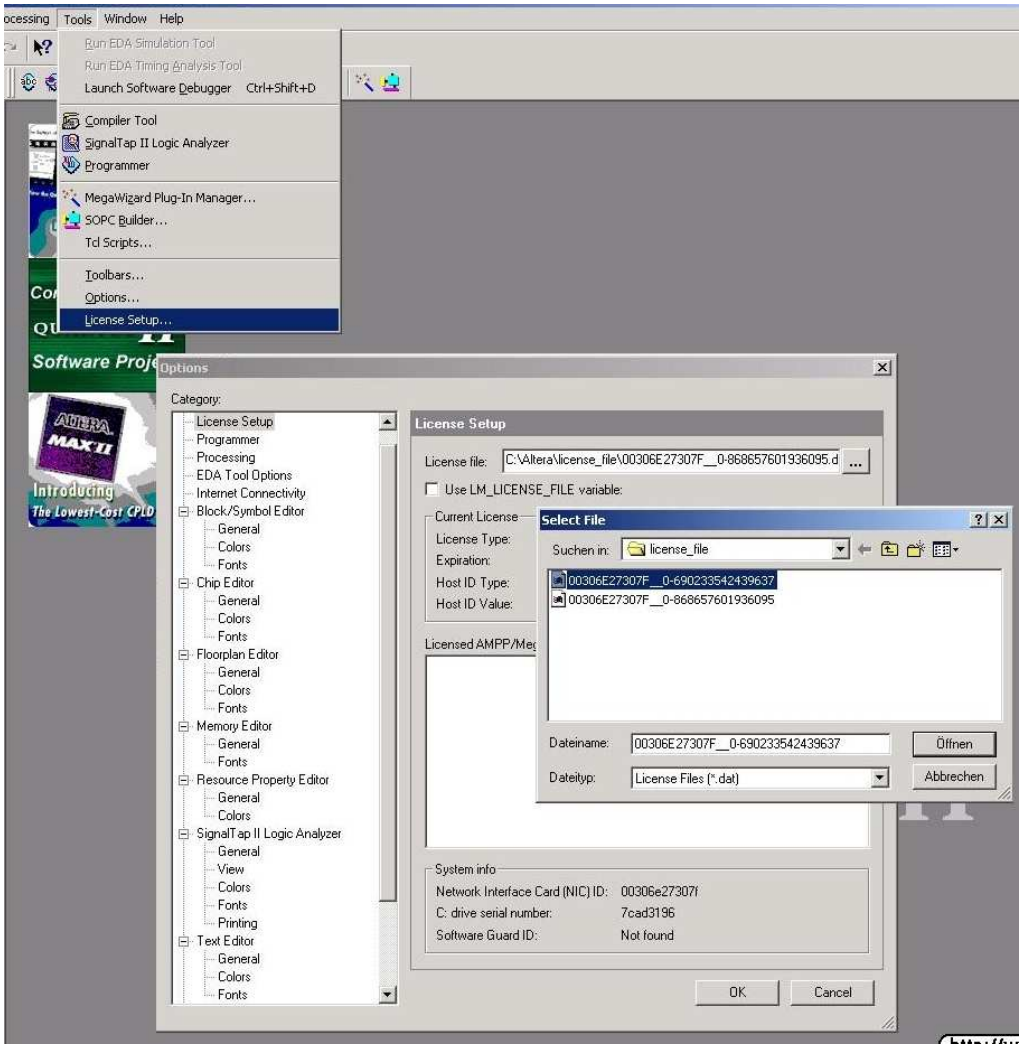


Abbildung 4.1: *Das Lizenzfi le kann einfach in der GUI des Lizenz Browsers eingestellt werden.*

4.2 Software Überblick

Der Aufbau von Quartus II ist in Abbildung 4.2 zu sehen. Auf die Details wird weiter unten eingegangen:

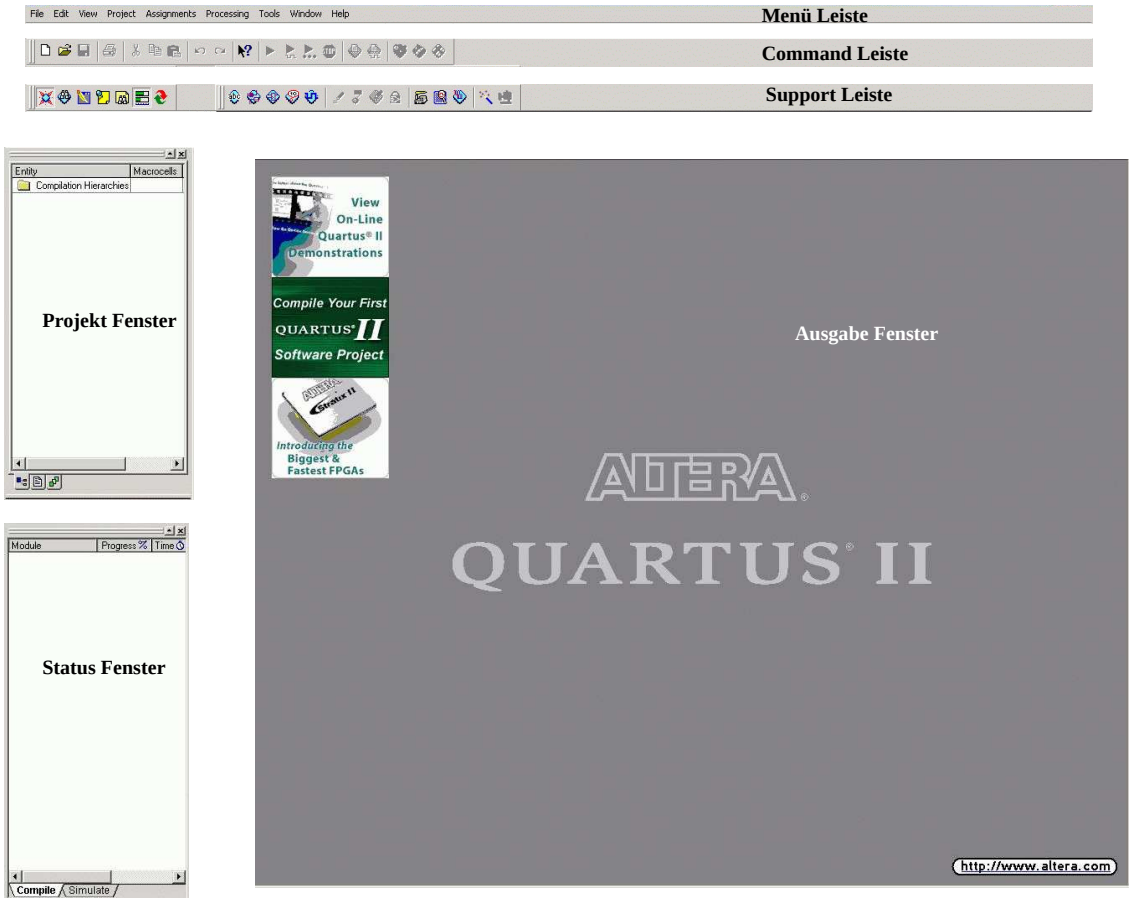


Abbildung 4.2: *Die Oberfläche von Quartus II ist in die dargestellten Bereiche aufgeteilt.*

Der Aufbau von Quartus II besteht im wesentlichen aus den drei Menü-Leisten im oberen Teil und dem Ausgabe-Fenster in der Mitte. Über die Support-Leiste lassen sich eine ganze Reihe zusätzlicher Fenster hinzuschalten, die sich den Platz mit dem Ausgabe-Fenster teilen. Für den laufenden Betrieb hat sich die in Abbildung 4.2 dargestellte Konfiguration bewährt. Der nachfolgende Überblick gibt eine kurze Erläuterung der einzelnen Bereiche:

- Menü Leiste: Die Menüleiste enthält sämtliche Funktionen der Quartus II Software. Ein Teil dieser Funktionen sind in Form von Buttons auf der Oberfläche enthalten, was die Bedienung sehr erleichtert und einem viel Sucherei erspart.
- Command Leiste: Die Command Leiste enthält die wichtigsten Steuer Kommandos für die automatische Synthese und Simulation.
- Support Leiste: Die Support Leiste enthält die Buttons für zusätzliche Anzeigefenster und die wichtigsten Design- und Konfigurations-Editoren.
- Das Ausgabe-Fenster ist der Quartus II Arbeitsplatz. Hier werden Protokoll und Logdateien angezeigt sowie die Editoren für Design und Synthese gestartet.

- Projekt-Fenster (optional): Das Projekt-Fenster zeigt und gewährt Zugriff auf sämtliche Projekt-Dateien. Es kann in der Supportleiste an- und ausgeschaltet werden (vgl. Abbildung 4.3).
- Status-Fenster (optional): Das Status-Fenster zeigt den Fortschritt der Synthese oder der Simulation an. Auch das Status-Fenster kann über die Supportleiste an und ausgeschaltet werden (vgl. Abbildung 4.3).

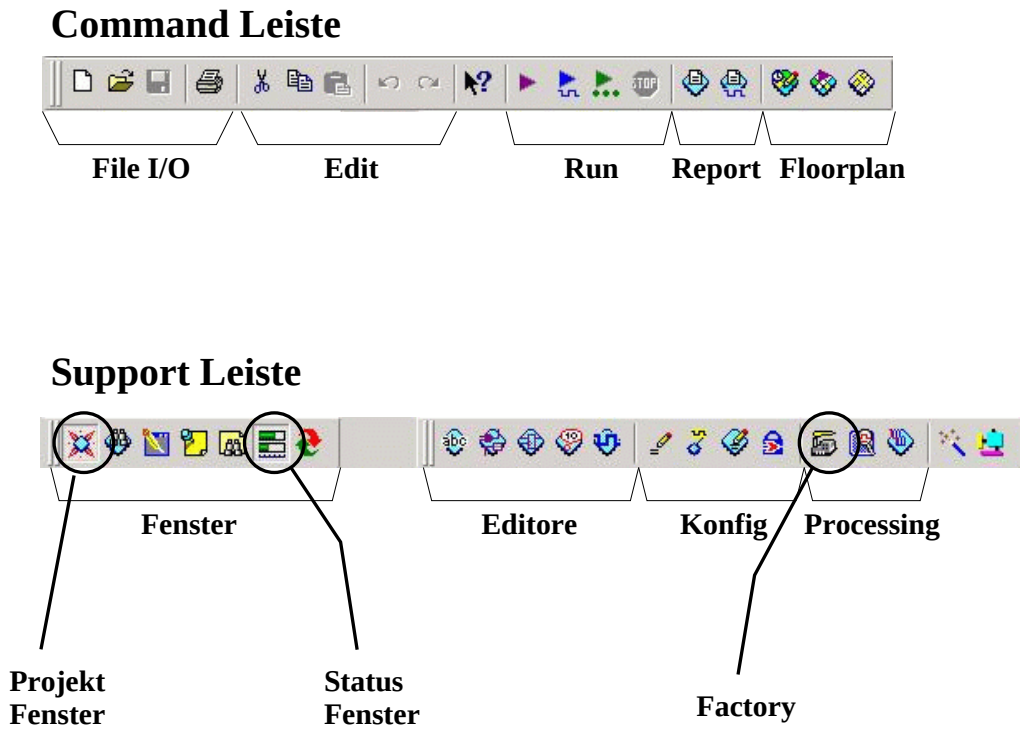


Abbildung 4.3: *Die Funktionsgruppen der beiden Hauptmenüleisten. Besonders hervorgehoben ist der **Factory**-Button. Der Factory-Button startet eine zusätzliche Bedienkonsole im Ausgabe-Fenster die Schrittweise durch den Synthesevorgang begleitet.*

4.3 Mit Quartus II arbeiten

Quartus II Web Edition verfügt über einen sogenannten HDL Design-Entry. Dieser ermöglicht den Entwurf von Schaltungen mittels Hardware-Beschreibungssprachen, wie z.B.: VHDL, Verilog oder AHDL. Für einen schematischen Design-Entry ist vermutlich die MAX+PLUS II Software von Altera die bessere Wahl. Auch für MAX+PLUS II ist eine kostenlose Entry-Level Lizenz erhältlich.

VHDL-Design Entry bedeutet, daß die Schaltung mittels Texteditor als VHDL Modell eingegeben wird, der VHDL Code also die vollständige Beschreibung der Hardware liefert. Quartus II besitzt einen HDL-Texteditor mit Syntax-Highlighting für VHDL. Bevor jedoch die erste Zeile Code eingegeben wird, muß zuerst ein neues Projekt angelegt werden:

4.3.1 Projekt anlegen

Am einfachsten erzeugt man ein neues Projekt mit dem **New-Project-Wizard**. Nach Aufruf (File Menü) öffnet sich ein Fenster, das interaktiv durch die Prozedur leitet. Der New-Project-Wizard erzeugt dabei ein Konfigurations-File im angegebenen Verzeichnis mit der Endung **.quartus**. Es enthält alle Projektinformationen wie z.B: eine Liste der Namen der beteiligten Design Files, Waveform-Files, Memory Initialisation Files, sowie die erforderlichen Konfigurationen für den Compiler, Assembler, Fitter usw.

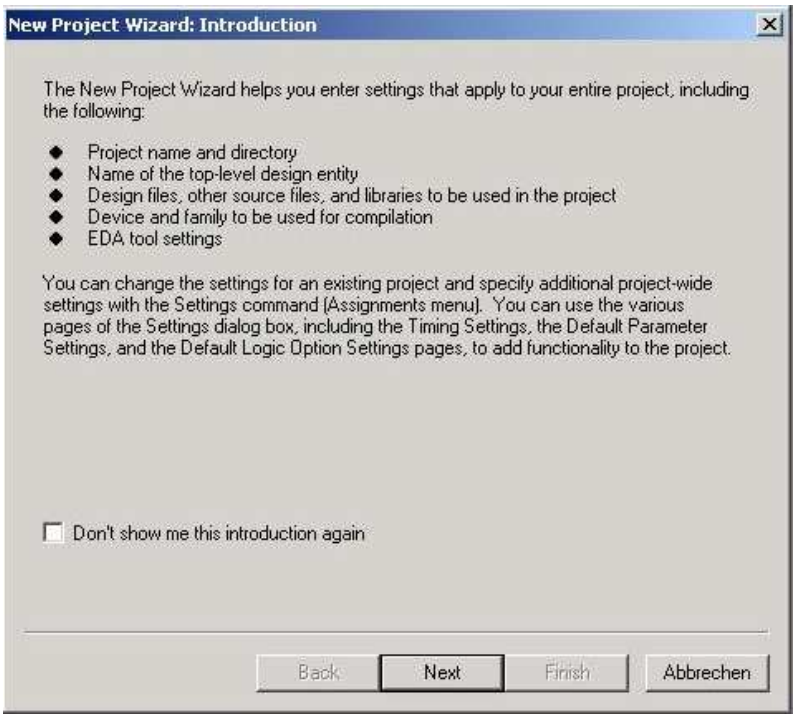


Abbildung 4.4: *Über die GUI des New-Project-Wizard werden alle projektrelevanten Informationen zusammengestellt. Diese sind: Projektname, die beteiligten Source-Dateien, Altera Chip Familie und Chip-Typ. Siehe auch die folgenden Abbildungen 4.5 bis 4.8.*

Sollten noch keine VHDL Source Dateien für das Projekt existieren, können diese mit dem Text-Editor angelegt werden. Das VHDL Syntax Highlighting des Editors reagiert allerdings auf die Dateiendung. Es empfiehlt sich daher VHDL-Dateien mit der Endung **.vhd** zu versehen oder am besten mit der **New**-Funktion der Command-Leiste anzulegen.

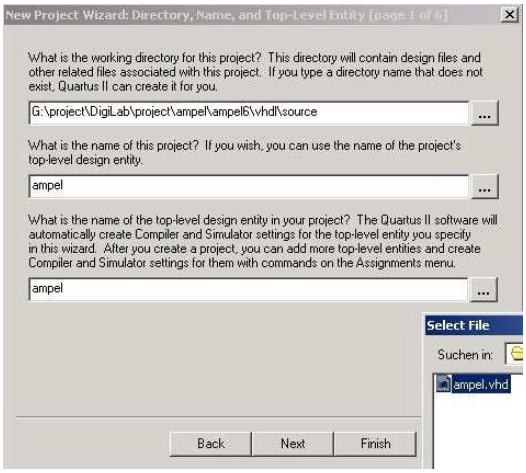


Abbildung 4.5: *Projekt und Entity angeben.*

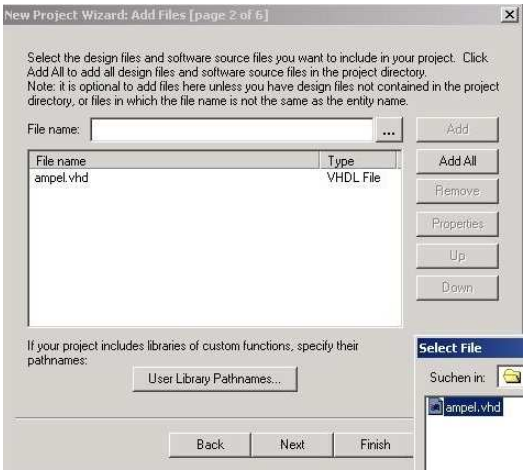


Abbildung 4.6: *VHDL Source Files hinzufügen.*

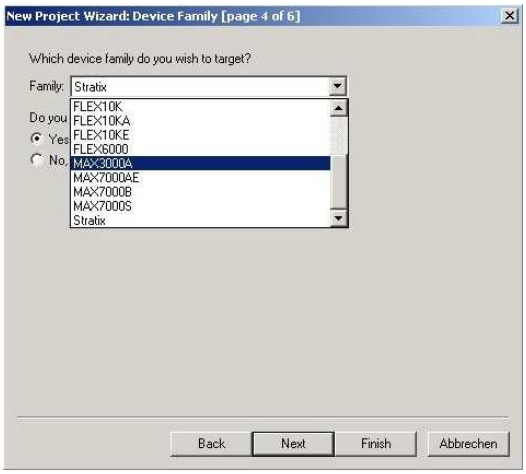


Abbildung 4.7: *Altera FPGA Familie auswählen.*

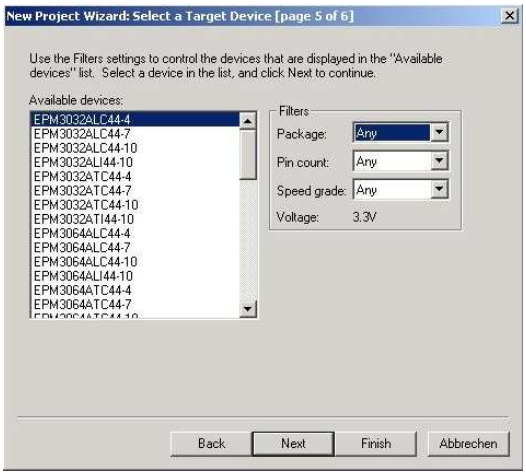


Abbildung 4.8: *FPGA Bauteil auswählen.*

4.3.2 Anlegen einer neuen VHDL Datei

Zum Anlegen einer neuen VHDL Datei am besten den **NEW**-Button aus der Command-Leiste verwenden (oder File Menü -> NEW). Ein Requester erscheint (vgl. Abbildung 4.10) in dem der Datei-Typ (VHDL) ausgewählt werden kann. Der Requester öffnet dann seinerseits automatisch den Text-Editor, deklariert das file als VHDL-Datei (aktiviert Syntax Highlighting) und '*added*' es zum aktuellen Projekt.

Im Kopfbereich des Bildschirms erscheint zusätzlich eine Bedien-Leiste mit den editortypischen Funktionsbuttons, siehe Abbildung 4.9.

Abbildung 4.9:

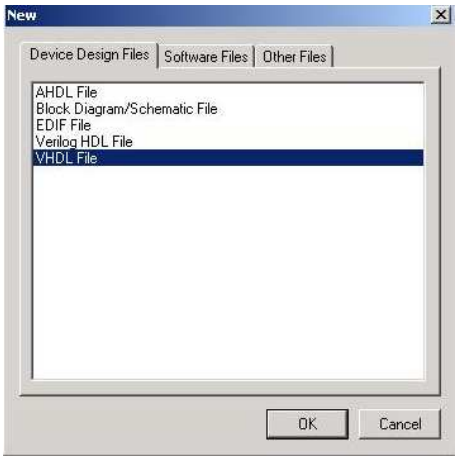


Abbildung 4.10: *Die **New**-File Auswahl wird über den **NEW**-Button der Command-Leiste oder vom File-Menü aus gestartet. Sie ermöglicht das Anlegen neuer Design-Files für Entwurf und Simulation.*

4.3.3 Die Synthese steuern

Der einfachste Weg die Synthese schrittweise zu steuern, ist die Verwendung der Synthese-Factory aus der Support-Leiste (Factory-Button der Support Leiste, siehe Abbildung 4.3). Dies öffnet die Factory GUI auf dem Ausgabe-Fenster worin alle Schritte der Synthese zugänglich sind. Abbildung 4.11 zeigt den Aufbau der GUI. Sie besteht aus fünf Kontrollfeldern für Analyse&Synthese, Fitter, Assembler, Timing-Analyzer und FDA-Netlist-Writer. Jedes dieser Felder ist nach dem gleichen Schema in vier Buttons unterteilt.

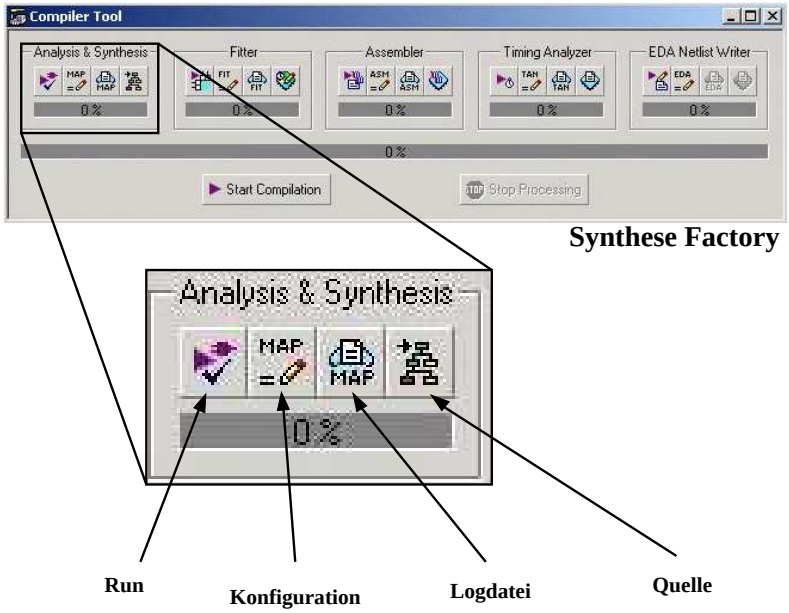


Abbildung 4.11: *Die Factory fasst alle für die Synthese wichtigen Funktionen in einem Panel zusammen. Das Panel erlaubt die Synthese in Einzelschritten über fünf Sub-Panel sowohl als auch die vollständig automatische Synthese ('Start Compilation'-Button).*

Damit kann die Hardware-Synthese in Einzelschritte aufgeteilt werden. Der Fortgang wird jeweils in einem Bargraph angezeigt. Gleichzeitig erscheint im Ausgabe-Fenster ein Message-Fenster, das die aktuellen Logdaten und ggf. Fehlermeldungen anzeigt. Jeder der Einzelschritte der Synthese-Factory kann über den zugehörigen Config-Button in allen Details konfiguriert werden. Die Synthese war dann erfolgreich,

wenn am Schluss alle vier Bereiche (Analyse&Synthese, Fitter, Assembler und Timing Analyzer) ohne Fehler compiliert haben. Abbildung 4.12 zeigt den Screenshot solch einer erfolgreichen Synthese.

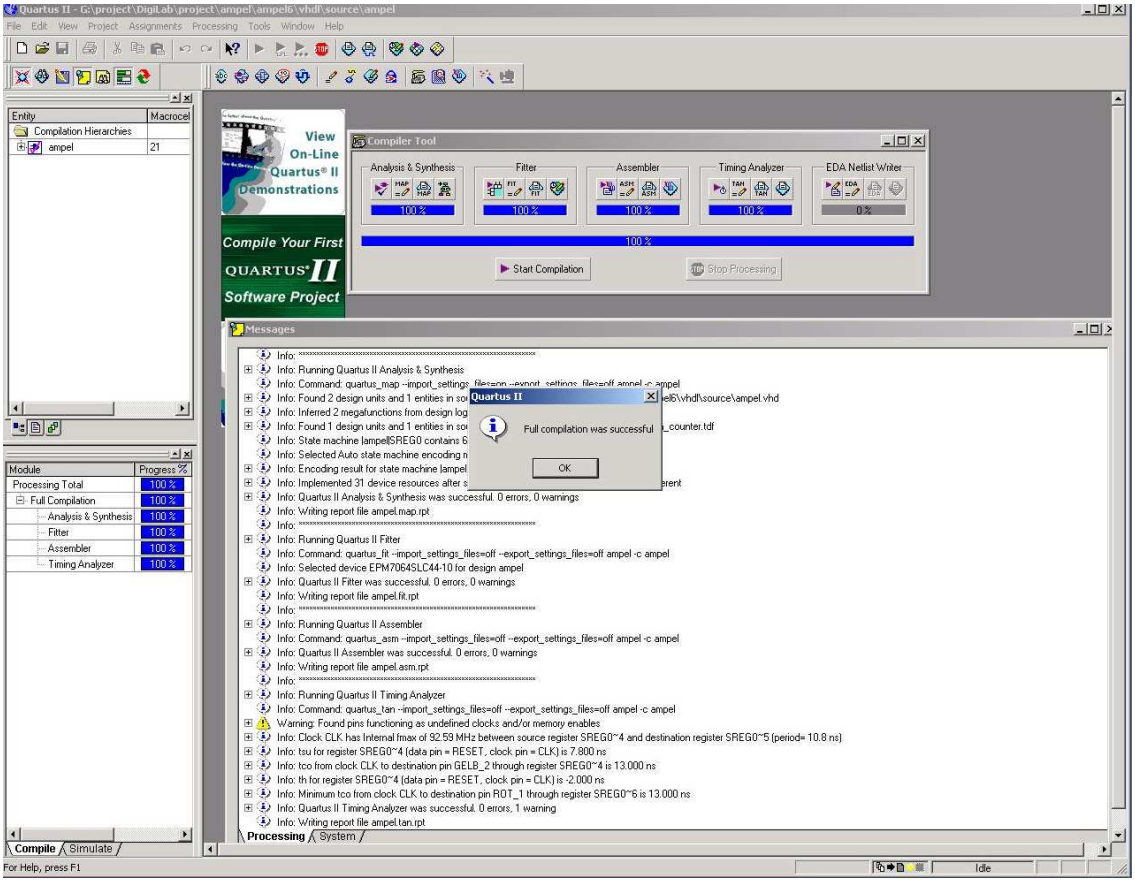


Abbildung 4.12: *Screenshot einer erfolgreichen Hardware-Synthese. Während der Synthese wird der Fortschritt im Bargraph der Synthese-Factory und gleichzeitig im Status Fenster (falls geöffnet) angezeigt. Im Ausgabe-Fenster wird jeder Einzelschritt der Synthese mitprotokolliert. Hier werden auch Fehler angezeigt, z.B: VHDL-Syntax Fehler, Fitter overflow Fehler (wenn das Design zu groß ist), usw ...*

4.3.4 FPGA Pin Zuweisung

Damit der FPGA in einer externen Platine verwendet werden kann, müssen natürlich die Anschlußpins des Bausteins frei konfigurierbar sein. Die Konfiguration wird am einfachsten mit dem Konfigurations-Button des Assembler-Menüs der Synthese-Factory (vgl. Abbildung 4.13) gestartet. Er öffnet das **Settings**-Menü mit dem alle Synthese-Parameter über Folder konfiguriert werden können. Das Settings-Menü ist in Abbildung 4.14 dargestellt. Für die Zuweisung der FPGA-Pins ist lediglich der Sub-Folder unter *Compiler Settings -> Device* von Belang.

Im *Device*-Folders ist bereits die Altera FPGA-Familie eingestellt (hier MAX 7000S), die am Anfang mit dem New-Project-Wizard spezifiziert wurde. Weiter unten werden nun alle zugehörigen FPGAs dieser Familie angezeigt. Hier wird der im DIGILAB (vgl. Kapitel 3) verwendete FPGA ausgewählt. Anschließend wird mit dem Button **ASSIGN PINS** das Pinzuweisungsmenü gestartet (vgl. auch Abbildung 4.15).

Das Zuweisungsmenü (Abbildung 4.15) zeigt alle FPGA-Pins des Ziel-FPGA in einer Liste. Nicht alle

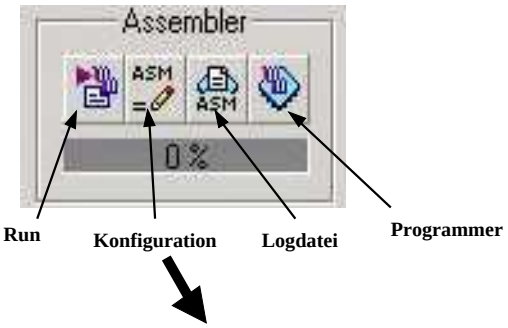


Abbildung 4.13: *Das Assembler-Menü der Synthese-Factory: Der Konfigurations-Button startet die* ***Settings****-GUI (vgl. Abbildung 4.14). Mit dem Programmer-Button ganz rechts wird der Hardware-Programmierer für das* DIGILAB *gestartet. Dieser ermöglicht am Ende den upload der Netzliste auf den angeschlossenen FPGA.*

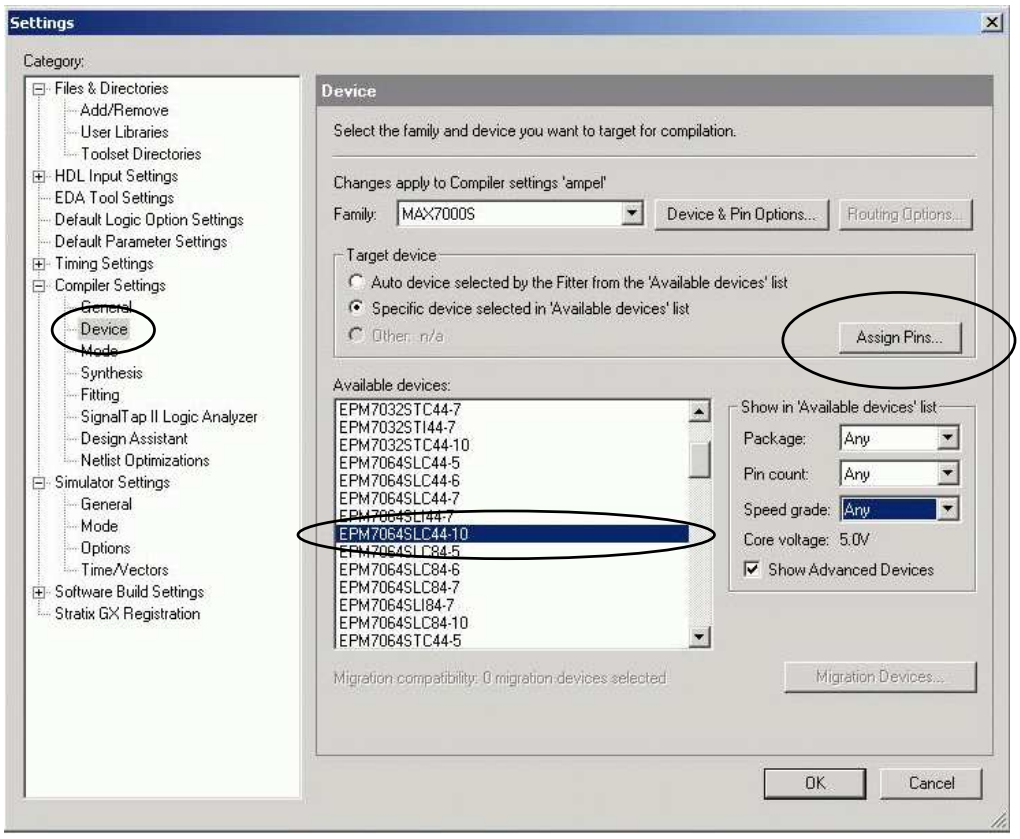


Abbildung 4.14: *Das Settings-Menü erlaubt detailreiche Modifikationen für Synthese und Simulation. Für die Pinzuweisung wird nur der Menüpunkt* ***Device*** *unter* ***Compiler Settings*** *benötigt. Dort wird die FPGA-Familie angezeigt und es muß der zu programmierende FPGA Chip ausgewählt werden. Die Pinzuweisung für diesen Chip wird mit dem Button* ***Assign Pins*** *gestartet.*

Pins sind für den Benutzer frei programmierbar. Einige der Pins sind bereits für Stromversorgung und das JTAG-Interface reserviert. Außerdem ist ein Clock-Signal Eingang vorgesehen. Die übrigen Pins sind frei programmierbar und können von der VHDL-Entity genutzt werden.

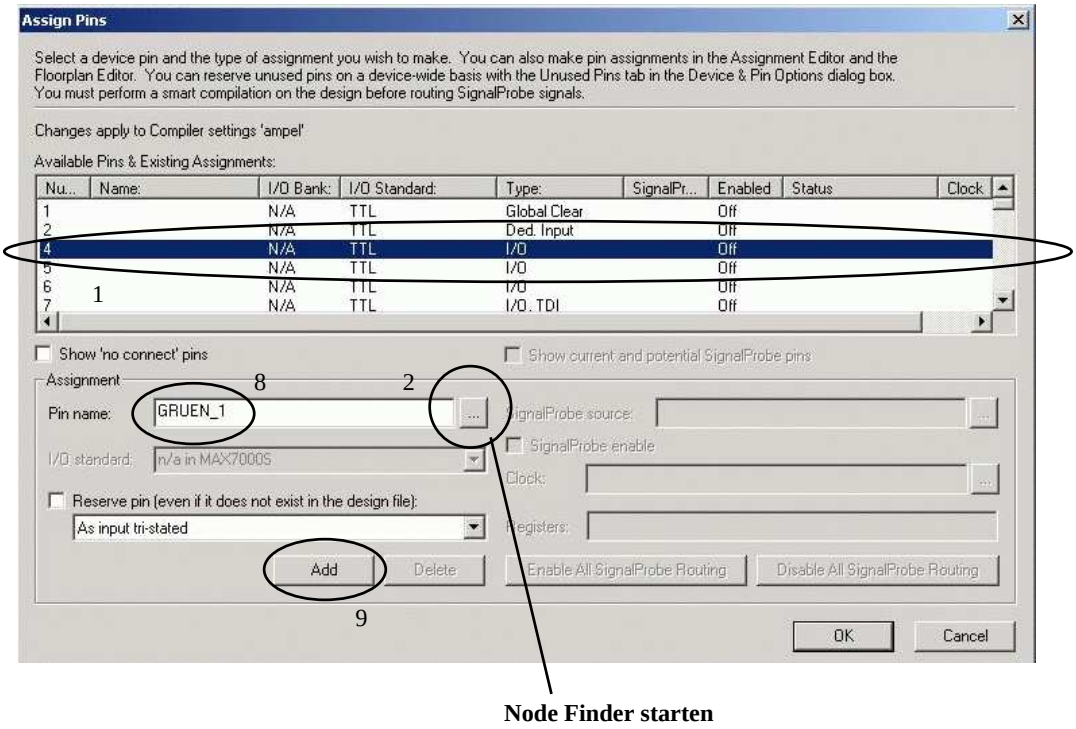


Abbildung 4.15: *Im Zuweisungs-Menü werden alle Pins des FPGAs angezeigt. Mit Hilfe des **Node-Finders** müssen die VHDL-Entity-Signale den freien Pins des FPGA zugewiesen werden. Die Reihenfolge bei der Zuweisung ist in der Abbildung durch Indizes angezeigt (vgl. auch Abb. 4.16).*

Genau diese Zuweisung von VHDL-Entity-Signalen zu den physikalischen Pins des eingesetzten FPGA-Chips ist die Aufgabe des Zuweisungs-Menüs. Um die VHDL-Entity-Signale zu erhalten, muß mit dem Quartus II **Node-Finder** die Entity des VHDL Top-Level Designs gescannt werden. Um eine Zuweisung vorzunehmen, wird im Zuweisungs-Menü der betreffende FPGA-Pin mit der Maus ausgewählt (highlighten) und anschließend der Node-Finder gestartet (siehe Abbildung 4.15 und Abbildung 4.16). Der Node-Finder zeigt zunächst kein Entity-Signal an. Um eine Entity zu scannen muß im oberen Menüfeld (Look in) der Name der Entity eingetragen sein (hier: ampel). Anschließend kann die Entity mit dem **START**-Button des Node-Finders gescannt werden (vgl. Abbildung 4.16). Die Entity-Signale erscheinen jetzt im linken Anzeigefeld (Nodes Found). Das Entity-Signal, das zugewiesen werden soll, wird mit der Maus markiert (highlighten) und mit der Pfeiltaste > des Node-Finders in das rechte Anzeigefeld (Selected Nodes) übertragen. Anschließend wird der Node-Finder mit **OK** verlassen. Das ausgewählte Entity-Signal erscheint jetzt im Zuweisungsmenü unter *Assignment - Pin Name*. Mit dem **ADD**-Button des Zuweisungs-Menüs wird das Signal schließlich dem FPGA-Pin zugewiesen. Der Vorgang muß so oft wiederholt werden, bis alle Entity-Signale des VHDL-Entwurfs physikalischen Pins des FPGA zugewiesen sind.

Zum Schluss muß in der Synthese-Factory der Assembler neu gestartet werden (Abbildung 4.13), damit die Zuweisung in das Ausgangs-Bitfile übernommen wird. Der komplette Zuweisungsvorgang ist in den Abbildungen 4.15 und 4.16 durch Indizes verdeutlicht.

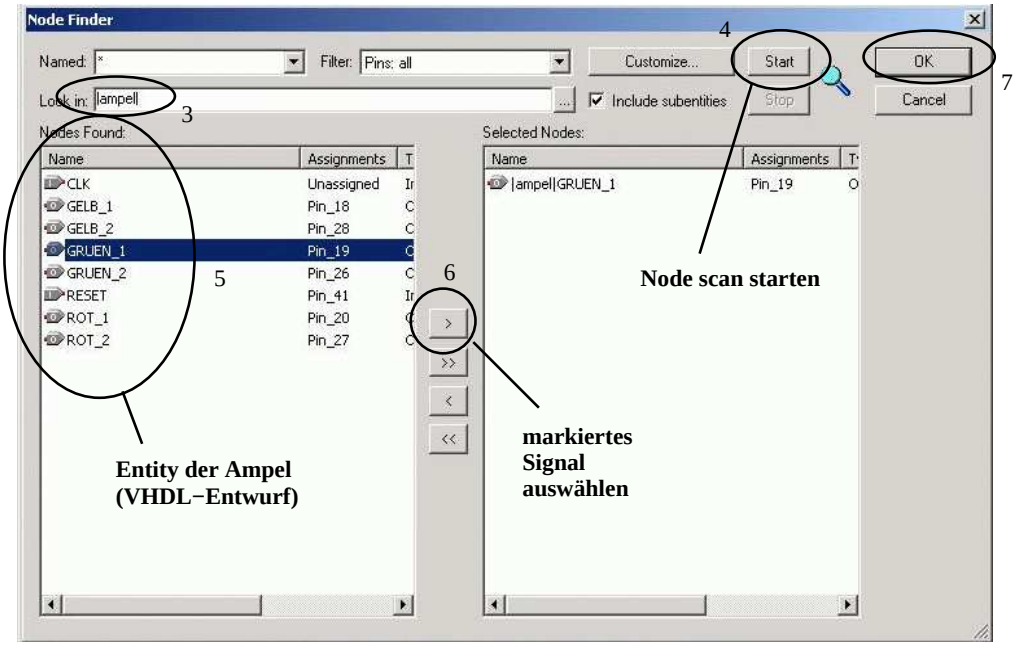


Abbildung 4.16: *Mit dem **Node-Finder** wird die Entity des VHDL-Entwurfs auf unverbundene Signale (Nodes) gescannt. Diese Signale können ausgewählt und an das Zuweisungs-Menü (Abb. 4.15) übergeben werden.*

4.3.5 Das DIGILAB programmieren

Schließlich soll das fertig synthetisierte bitfile auf den FPGA geschrieben werden. Hierzu muß das DIGILAB mit dem beigefügten Kabel an die parallele Schnittstelle des Computers angeschlossen sein. Ebenfalls muß das DIGILAB während dem Schreiben mit Strom versorgt werden. Dann kann der Quartus II **Programmer** gestartet werden. Dies geschieht ebenfalls auf der Konsole der Synthese-Factory im *Assembler*-Segment. Die rechte Taste des Assembler (vgl. Abbildung 4.13) startet die Programmier-Konsole die in Abbildung 4.17 dargestellt ist.

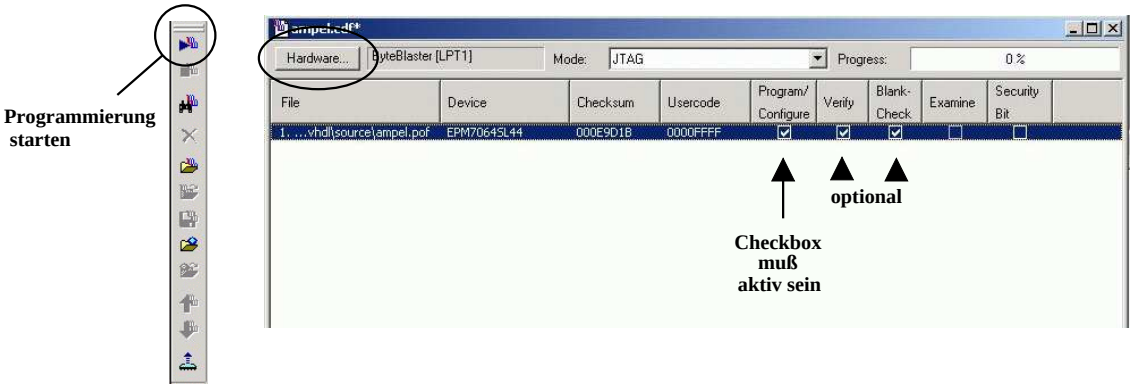


Abbildung 4.17: *Der Programmer besteht lediglich aus einer Konsole mit Check-Boxen und einem Button-Bar der links am Rand des Ausgabe-Fensters erscheint (solange der Programmer läuft).*

Wird der Programmer zum ersten Mal gestartet, so muß zunächst noch die Programmierhardware konfiguriert werden. Hierzu ist einfach der **Hardware**-Button oben auf der Programmier-Konsole zu drücken.

Es erscheint ein Fenster wie in Abbildung 5. Dort muß die Konfiguration für das El-Camino DIGILAB so wie abgebildet eingetragen sein.

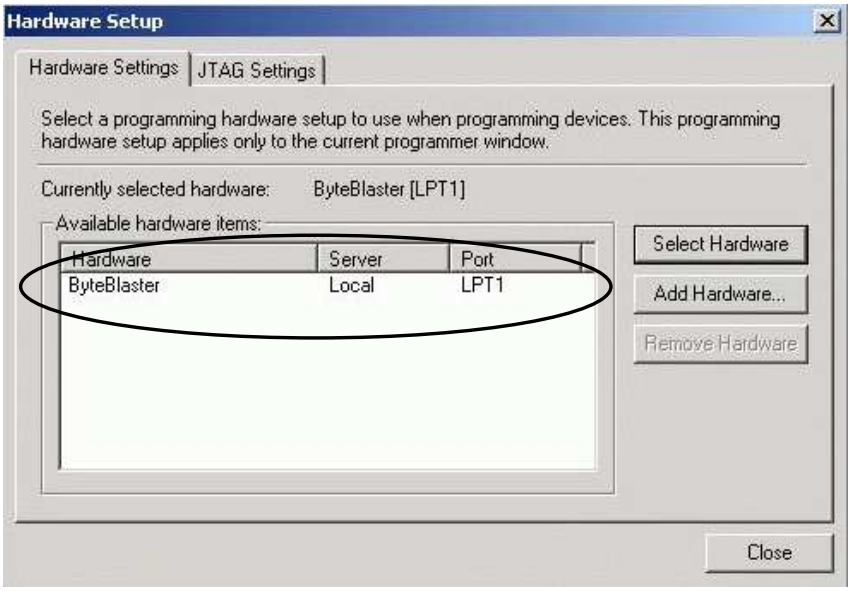


Abbildung 4.18: *Beim allerersten Start des Programmers muß die Programmierhardware eingestellt werden. Hierzu muß das* DIGILAB *zuerst an den Computer angeschlossen und eingeschaltet werden. Mit* ***Select Hardware*** *erscheint das* ***ByteBlaster****-Kabel dann in der Auswahl und kann mit* ***Add Hardware*** *hinzugefügt werden.*

Für die Übertragung des Bitfiles an den FPGA muß im Konsole-Fenster des Programmers (Abbildung 4.17) die Programmier-Hardware ausgewählt werden (highlighten). Außerdem muß dort mindestens die Check-Box **Program/Configure** aktiviert sein.

Um die Programmierung zu starten, befindet sich am linken Rand des Quartus II Ausgabe-Fensters eine Button-Bar, die beim Start des Programmers dort hinzugefügt wird. Die Button-Bar ist ebenfalls in Abbildung 4.17 dargestellt, auch wenn sie nicht direkt mit dem Konsolefenster des Programmers verbunden ist. Mit Betätigung des **Play**-Buttons startet die Übertragung des Bitfiles an das DIGILAB. Sobald der Bargraph 100% erreicht hat, kann das DIGILAB vom Programmierkabel getrennt werden (vorher stromlos machen!).

Hinweis für den Betrieb:
Wie sich gezeigt hat, laufen manche Programme erst dann einwandfrei, wenn das DIGILAB vom Programmierkabel getrennt ist!

4.4 Simulation mit Quartus II

Anders als ein Computerprogramm kann der Code einer Hardware-Beschreibung mit VHDL nicht einfach auf der Zielhardware gestartet werden, um seine Korrektheit zu verifizieren. Stattdessen ist der Code ja selbst die Hardware! Für eine computerbasierte Entwicklung wird daher ein Simulator benötigt.

Quartus II besitzt die Möglichkeit sowohl zur funktionellen- als auch zur Timing-Simulation des VHDL-Entwurfs. Dies ermöglicht die Vorhersage des logischen Verhaltens und der Signallaufzeiten auf dem angestrebten FPGA, noch bevor dieser programmiert wird. Der Simulator hat dabei die Aufgabe die Eingangs-Signale des Entwurfs mit Stimuli zu versorgen und das Verhalten der Ausgangs-Signale gleichzeitig aufzuzeichnen und darzustellen. Ebenfalls sollte das Verhalten interner Signale, die nicht zur VHDL-Entity zählen, darstellbar sein. Damit dies erreicht wird, besitzt Quartus II einen so genannten Waveform-Editor mit dem die Stimuli der Eingangs-Signale grafisch festgelegt und für die spätere Simulation in einer Waveform-Datei abgespeichert werden.

4.4.1 Eine Simulationsdatei erzeugen

Bevor die Simulation gestartet werden kann, muß eine Waveform-Datei erstellt werden. In dieser Datei werden für alle Eingangs-Signale der VHDL-Entity entweder zeitlich variable Stimuli festgelegt, oder die Signale werden auf konstante Werte (high/low) gesetzt. In der Regel wird zumindest das Clock-Signal (falls die Schaltung getaktet ist, was meistens der Fall sein wird) einen zeitlich variablen Eingang erfordern.

Zur Konfiguration der Waveform-Datei wird zunächst der Waveform-Editor gestartet. Am einfachsten geschieht dies mit dem zugehörigen Button in der Support-Leiste, vergleiche Abbildung 4.19.

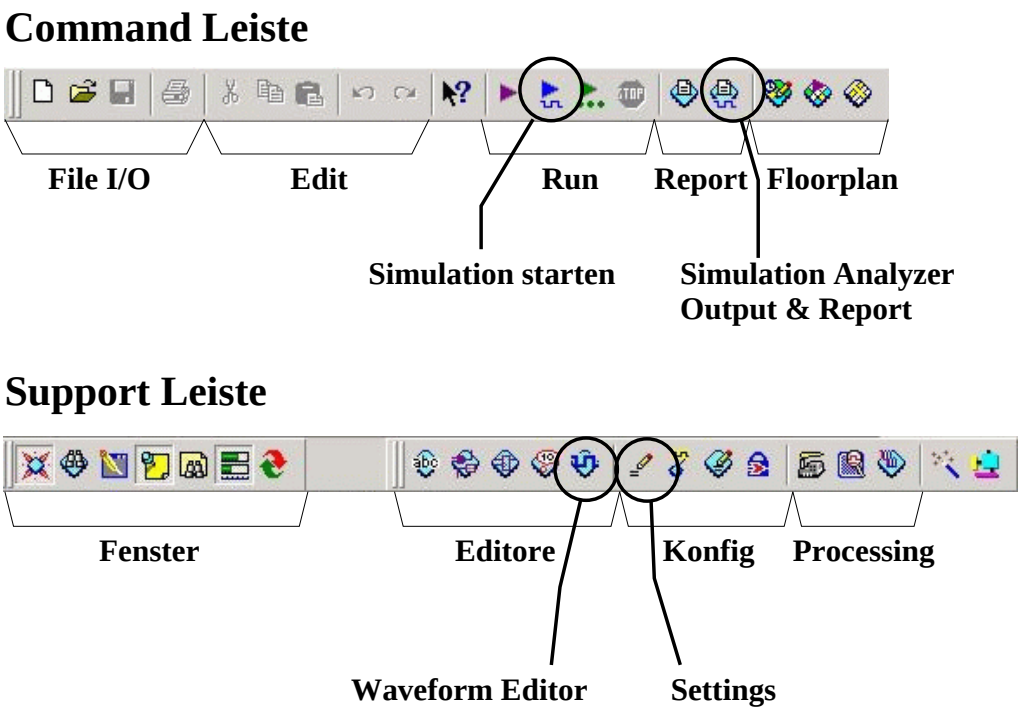


Abbildung 4.19: *Die Buttons für die Simulation befinden sich in der Command- und in der Support-Leiste. Zuerst wird der Waveform-Editor gestartet mit dem die Signale konfiguriert werden, die bei der Simulation überwacht werden sollen.*

Der Waveform-Editor ist ein grafischer Editor, in dem die Signale der VHDL- Beschreibung dargestellt werden. Dabei können sowohl Entity-Signale als auch interne Signale der VHDL-Beschreibung angezeigt werden. Selbst technologiebedingte Signale, die erst während der Technologie-Abbildung erzeugt werden, können dargestellt werden. Eine der ersten Aufgaben ist daher auch die Auswahl der anzuzeigenden Signale. Vorher wird jedoch die System-Laufzeit für die Simulation festgelegt.

Die System-Laufzeit für die Simulation wird im **Edit**-Menü der Menü-Leiste unter **End Time** eingestellt. Es erscheint ein Requester (siehe Abbildung 4.20), mit dem die Gesamtlaufzeit der FPGA-Simulation eingestellt wird. Es handelt sich dabei **nicht** um die Ausführungszeit der Simulation (hängt vom Rechner ab), sondern um die Anzahl der Nanosekunden, die auf dem FPGA simuliert werden sollen! Der noch leere Waveform-Editor ist in Abbildung 4.20 zusammen mit dem Requester zur Zeiteinstellung zu sehen.

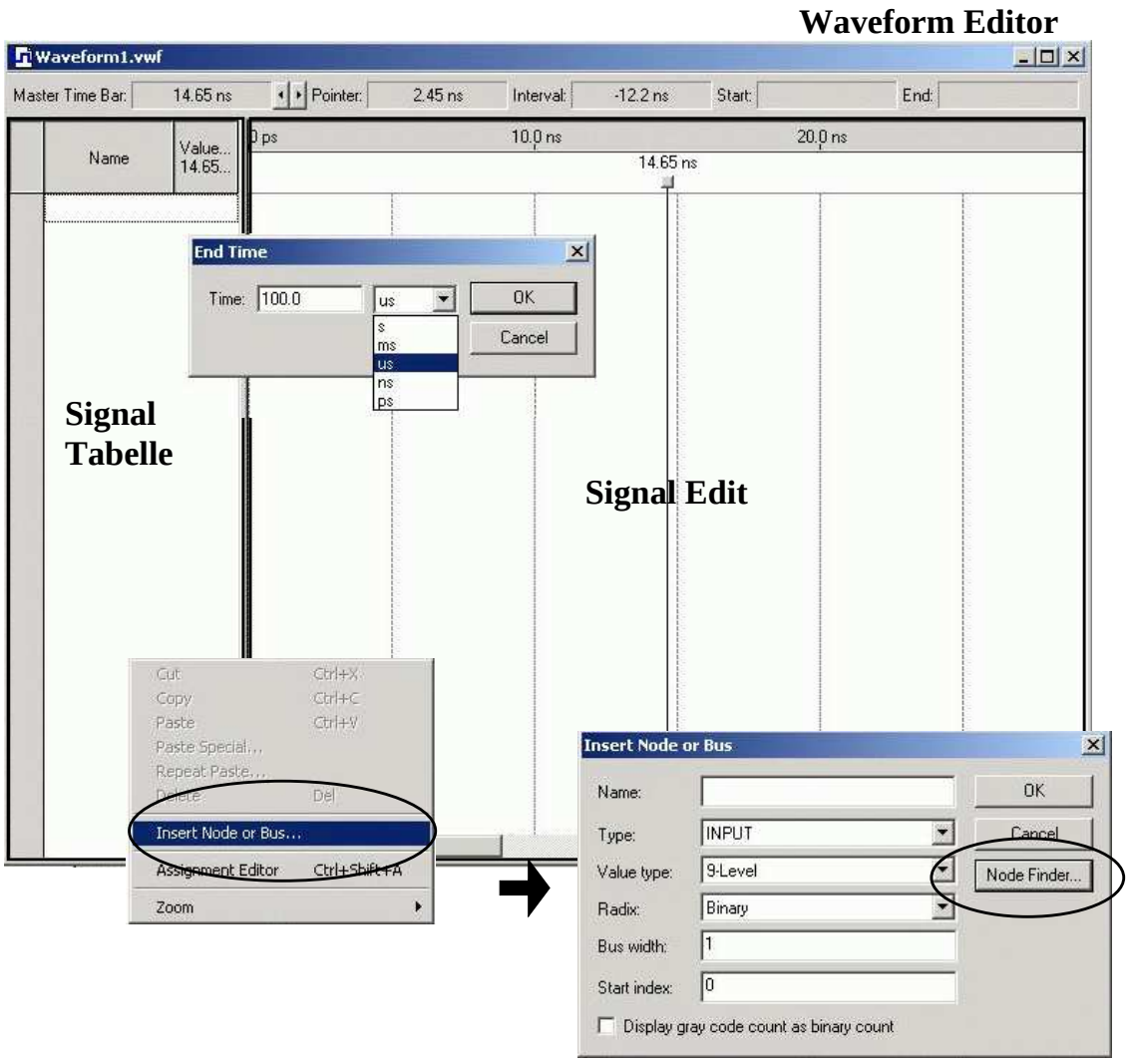


Abbildung 4.20: *Der Waveform-Editor zusammen mit dem **End-Time**-Requester (Menü-Leiste -> Edit -> End Time ...) und dem Signal-Menü (rechte Maustaste im Feld der 'Signal-Tabelle'). Der Waveform-Editor besteht aus dem 'Signal-Tabelle' Bereich (links - noch leer) und dem 'Signal-Edit' Bereich (rechts) in dem die Signale später dargestellt und editiert werden.*

Um Signale für die Anzeige auszuwählen, muß der **Node-Finder** gestartet werden. Dies geschieht am einfachsten mit der rechten Maustaste (Bereich 'Signal-Tabelle' des Waveform-Editors). Es erscheint das in Abbildung 4.20 unten dargestellte Menü. Mit '**Insert Node or Bus**' wird der Node-Finder gestartet, vergleiche hierzu auch Abbildung 4.16 und 4.21.

Mit dem Node-Finder wird der VHDL-Entwurf nach Signalen (Nodes) gescannt (Start-Button im Node-Finder). Unter **Filter** kann das gewünschte Signal-Filter ausgewählt werden. In Abschnitt 4.3.4 hatten wir für die Pinzuweisung nur nach Entity-Signalen gescannt. In der Simulation möchten wir jedoch auch interne Signale des Entwurfs verfolgen. Daher wird der Signal-Filter z.B. auf '**Design-Entry**' gestellt, bevor der Node-Scan gestartet (Start Button) wird. Im linken Fenster 'Nodes Found' werden dann alle Signale des Entwurfs zur Auswahl angezeigt. Abbildung 4.21 zeigt die Signale der Ampelsteuerung nach erfolgreichem scan (linker Teil). Als nächster Schritt werden die interessierenden Signale markiert und mit dem '>' Button auf die rechte Seite 'Selected Nodes' übertragen. Es handelt sich dabei um Signale (interne und Ausgangs-Signale), die in der Simulation überprüft werden sollen und um Eingangs-Signale, die mit Stimuli belegt werden müssen.

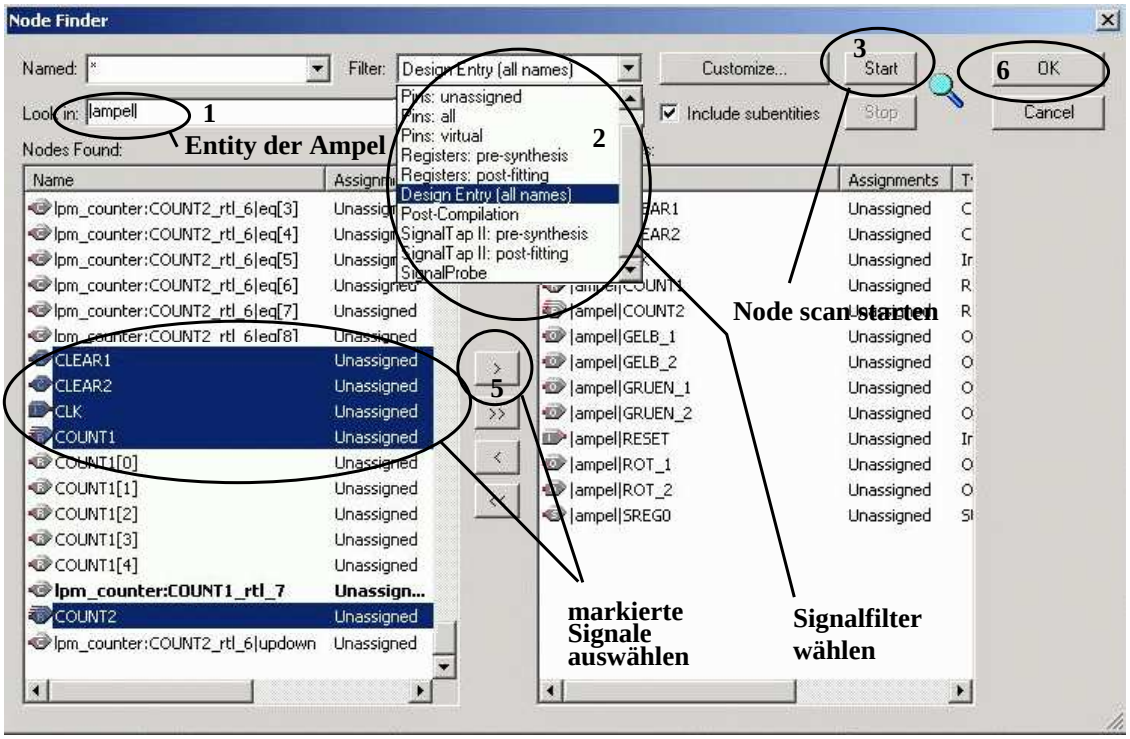


Abbildung 4.21: *Mit dem Node-Finder werden die Signale ausgewählt (Entity und interne), die in der Simulation analysiert werden sollen. Zusätzlich werden die Eingangs-Signale ausgewählt, die mit Stimuli belegt werden sollen. Die Indizes in der Abbildung beziehen sich auf die Reihenfolge der Auswahl.*

Nun wird der Node-Finder mit dem **OK**-Button verlassen und anschließend ebenfalls das Signal-Menü. Damit sind die Signale in den Waveform-Editor übernommen (vgl. Abb. 4.22). Die Reihenfolge der Signale kann im Waveform-Editor sortiert werden. Dazu einfach das betreffende Signal in der Signal-Tabelle mit der Maus anwählen und vertikal verschieben.

Im nächsten Schritt müssen die Signal-Stimuli für die Eingangs-Signale der VHDL-Entity festgelegt werden. Damit die Simulation Sinn hat, muß in der Regel zumindest der Clock-Eingang des Entwurfs mit dem Clock-Generator des Simulators verbunden sein. Statische Signale werden zumeist auf definierte feste logische Werte gesetzt (geforced). Der zeitliche Verlauf von Signalen, die nur selten toggeln[1] (nicht periodisch), wie z.B das Reset-Signal, können mit der Maus abschnittsweise markiert und auf die beabsichtigten logischen Werte geforced werden. Dazu dient auch die Button-Bar des Waveform-Editors, wie in Abbildung 4.24 (links) zu sehen. In Abbildung 4.23 ist die Zuweisung des Clock-Signales mit dem Clock-Generator des Simulators gezeigt. Abbildung 4.24 zeigt, wie das Reset-Signal abschnittsweise auf logisch Null und anschließend (für den Rest der Simulation) auf logisch Eins 'geforced' wird.

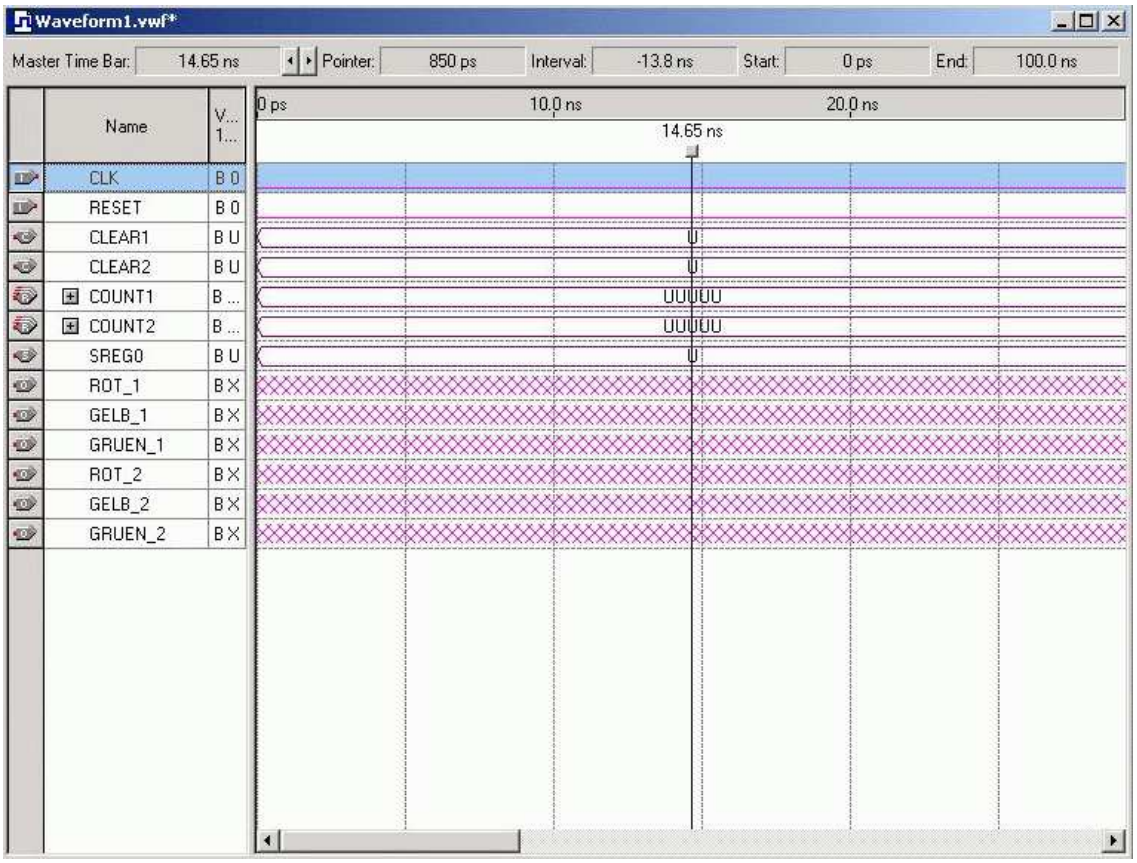


Abbildung 4.22: *Der Waveform-Editor zeigt die für die Simulation ausgewählten Signale.*

Bei längeren Simulationszeiträumen ist die Zoom-Funktion des Waveform-Editors äußerst nützlich. Sie gestattet, den Signalverlauf auf unterschiedlichen Zeitskalen zu konfigurieren. Die Zoom-Funktion wird mit dem Vergrößerungsglas der Button-Bar gestartet. Die Zoom-Funktion liegt auf der linken (hinein) und rechten (heraus) Maustaste, wenn sich die Maus im 'Signal-Edit Bereich' des Waveform-Editors (vgl. Abb. 4.20) befindet. Befindet sich die Maus im 'Signal-Tabelle Bereich', oder ist die Lupenfunktion ausgeschaltet, kann trotzdem mit der rechten Maustaste das Zoom-Menü gestartet werden. Dieses erlaubt auch die Darstellung der Signale des gesamten Simulationszeitraumes ('Fit in Window'-Funktion).

Zum Schluss wird die Waveform-Datei mit dem '**Save as**'-Button der Quartus II Menüleiste (oder File-Menü) für die anschließende Verwendung in der Simulation gespeichert. Damit die Datei als Waveform-

[1] Wechsel zwischen logisch 0 und logisch 1 und vice versa

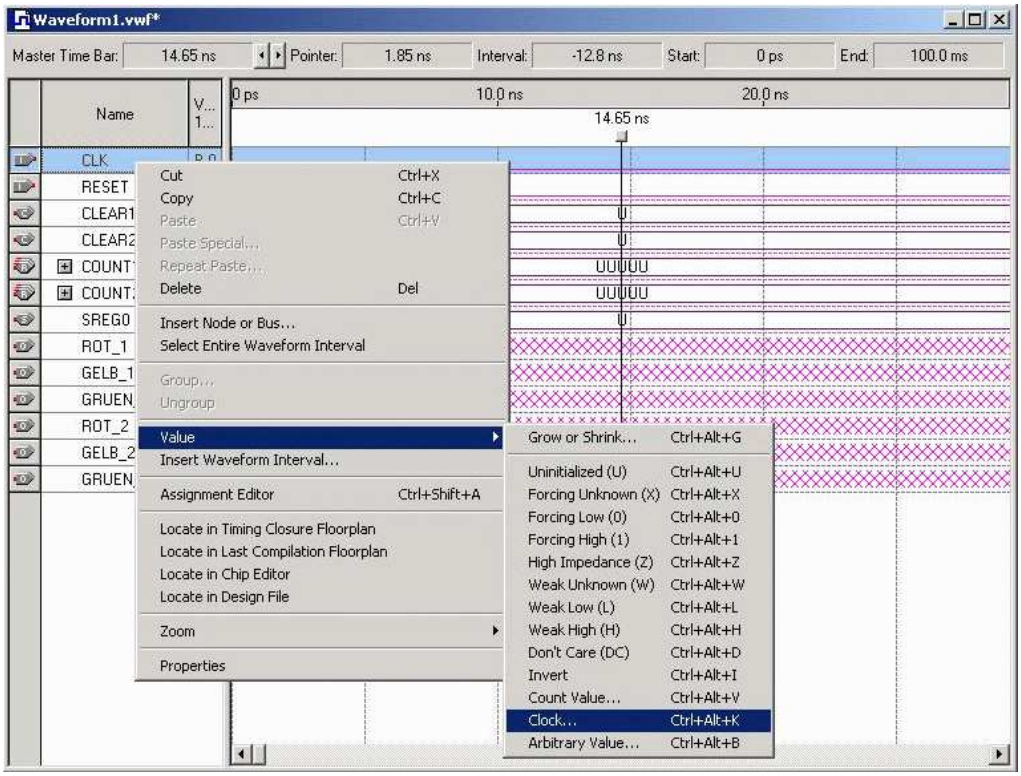


Abbildung 4.23: *Die Eingangssignale des VHDL-Entwurfs (Entity) werden mit Signalquellen (Stimuli) verbunden, oder auf definierte feste Signalwerte gesetzt (geforced).*

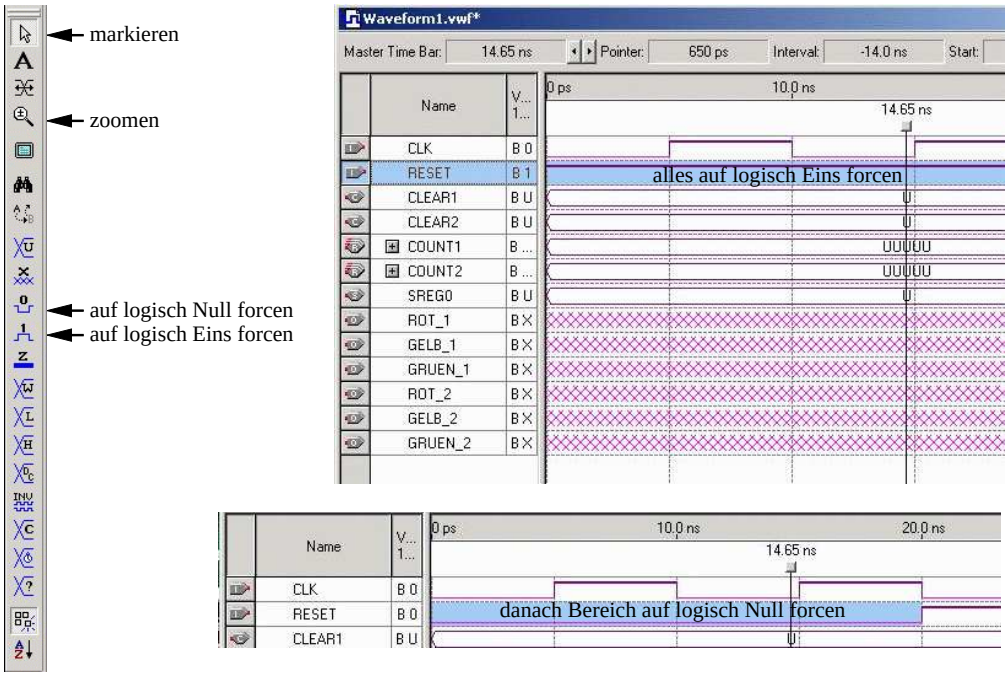


Abbildung 4.24: *Mit dem 'Button-Bar' am linken Bildschirmrand stehen zahlreiche Funktionen für Navigation und zur Einstellung der Signalstimuli zur Verfügung.*

Datei, also als Simulationsinput erkannt wird, muß sie die Endung '**.vwf**' haben. Es können beliebig viele Waveform-Dateien erzeugt werden, um verschiedene Simulationsaspekte in Betracht zu ziehen. Die Waveform-Dateien werden dem Simulator vor dem Start der Simulation übergeben.

4.4.2 Eine Simulation durchführen

Nachdem eine gültige Waveform-Datei existiert (siehe Abschnitt 4.4.1), kann nun die Simulation gestartet werden. Zuerst muß jedoch die Simulation konfiguriert werden.

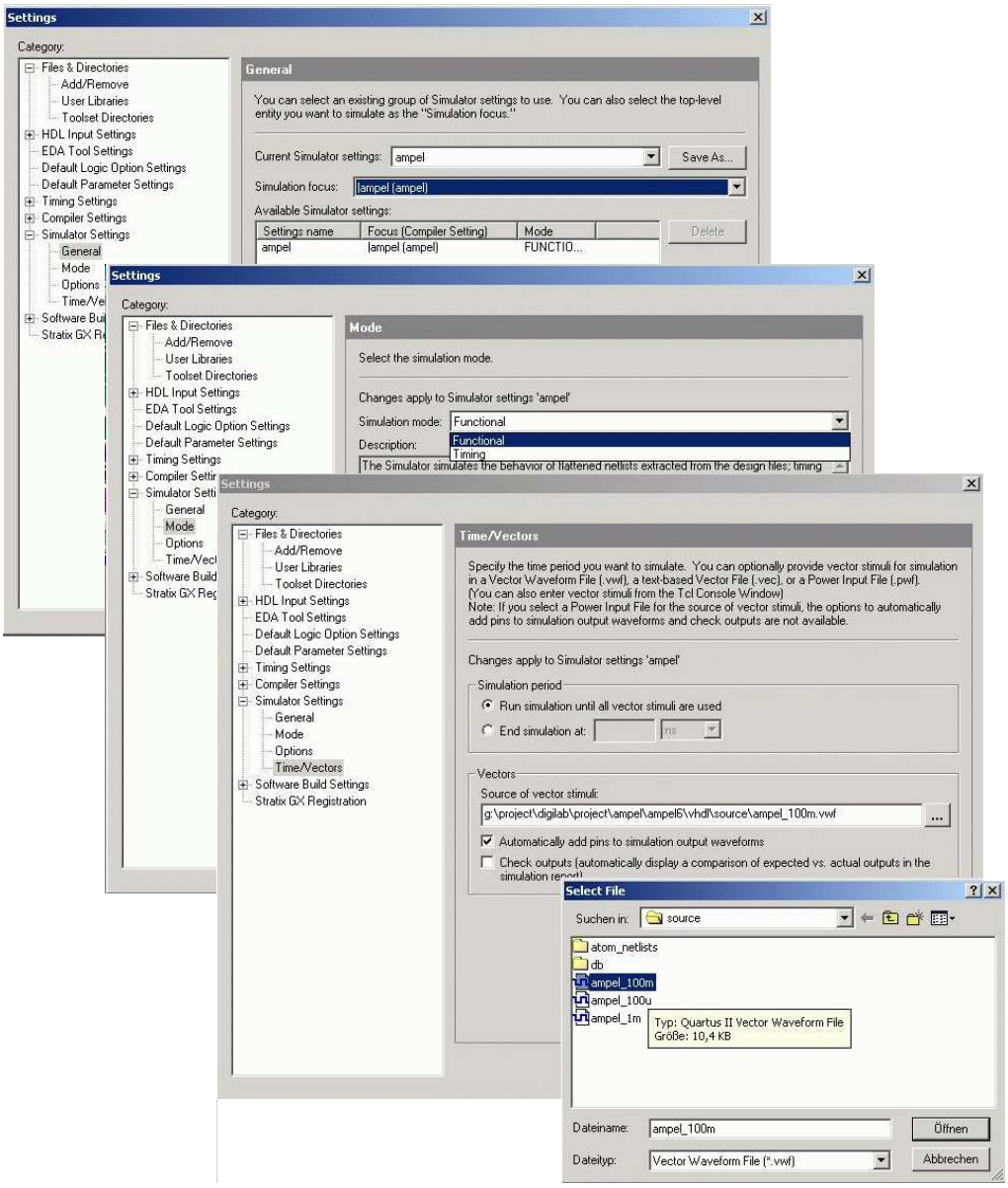


Abbildung 4.25: *Das Settings-Menü. Die Simulation wird unter 'Simulator Settings' konfiguriert. Die Abbildung zeigt die Fenster 'General', 'Mode' und 'Time Vectors'. Damit wird der zu simulierende Entwurf, der Simulations-Typ und die Waveform Datei eingestellt.*

Die Konfiguration der Simulation wird im Settings-Menü eingestellt. Abbildung 4.25 zeigt die für die Simulation relevanten Settings-Menüs. Unter 'General' wird der zu simulierende Entwurf eingestellt (hier

'ampel'). Im Menü 'Mode' kann zwischen einer **Timing**-Simulation und einer **funktionellen**-Simulation ausgewählt werden. Die Timing-Simulation berücksichtigt bereits alle Signal-Laufzeiten, die durch das Place&Route der Synthese hervorgerufen werden und bietet somit eine Laufzeit-Analyse des Entwurfs für den eingestellten FPGA. Die funktionelle Simulation bietet hingegen nur eine Beurteilung der Funktionalität des Entwurfs, ist also eine reine Idealisierung. Im Menü 'TimeVectors' wird zuletzt die Waveform-Datei ausgewählt, mit der simuliert werden soll (vgl. Abschnitt 4.4.1).

Damit ist die Konfiguration abgeschlossen und die Simulation kann gestartet werden. Dies geschieht am einfachsten wieder mit der Button-Bar der Command-Leiste (vgl. Abb. 4.19). Sobald der Button 'Simulation starten' gedrückt wird, erscheint im Quartus II Ausgabe-Fenster das '**Simulation Report**'-Fenster. Dieses enthält nach erfolgreicher Simulation auch den Waveform-Analyzer mit den Signalen der Waveform-Datei. Der Waveform-Analyzer ist in der Bedienung weitgehend identisch mit dem Waveform-Editor (vgl. Abb. 4.26 und Abb. 4.24). Mit Hilfe der Zoom-Funktion ist eine Kontrolle der einzelnen Signale relativ zum Stimuli leicht möglich.

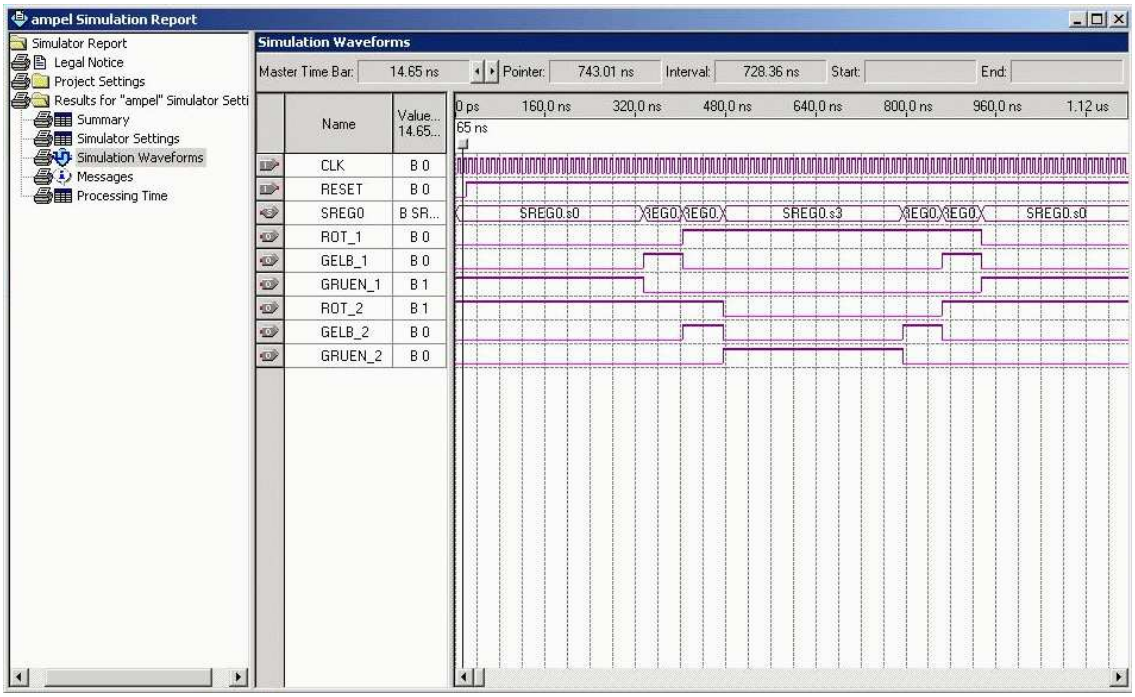


Abbildung 4.26: *Das 'Simulation-Report'-Fenster mit dem Waveform-Analyzer (rechts). Im Waveform-Analyzer werden die Signale aus der Waveform-Datei für den Zeitraum der Simulation angezeigt. Die Bedienung des Waveform-Analyzers ist mit der des Waveform-Editors weitgehend identisch.*

Zusammen mit dem Simulator ist Quartus II ein leistungsfähiges Werkzeug für den einfachen und schnellen Schaltungsentwurf. Der VHDL-Entwurf kann jederzeit mit dem Editor modifiziert werden und in der Regel benötigt die anschließende Synthese und Simulation nur Sekunden. Der Entwurf mit VHDL ist daher um Größenordnungen schneller als andere klassische Entwurfsmethoden.

Kapitel 5

Anwendungsbeispiele

In diesem Kapitel soll anhand von Beispielen gezeigt werden, wie sich die theoretischen VHDL-Grundlagen in praktische Anwendungen umsetzen lassen. Den Einsatzbereichen und der Kreativität des Anwenders werden hierbei von den heute erhältlichen FPGAs, CPLDs und von VHDL selbst kaum Grenzen gesetzt.

5.1 Fußgängerampel

In diesem Beispiel wird die aus den vorhergehenden Kapiteln bekannte Ampelsteuerung aufgegriffen und um eine Fußgängerampel mit Drucktaster ergänzt. Außerdem wird die Dauer der Ampelphasen auf reelle Werte im Sekundenbereich ausgedehnt.

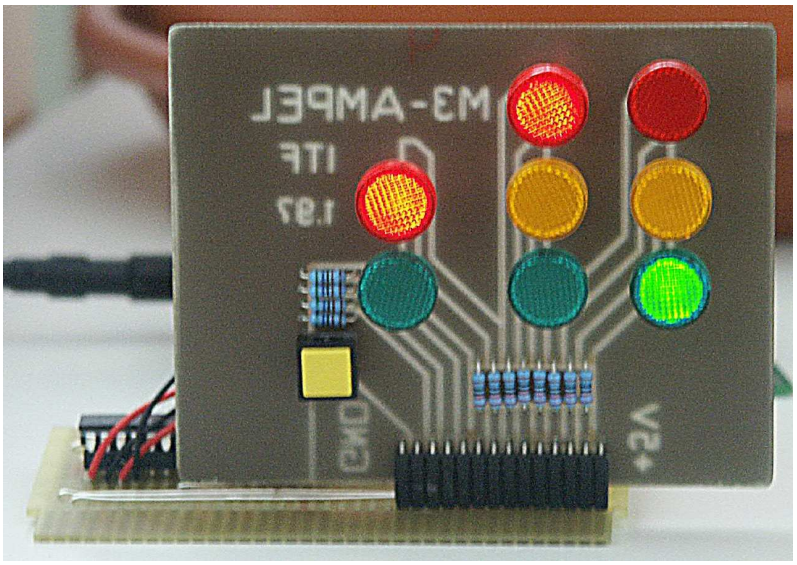


Abbildung 5.1: *Platine mit LEDs in Ampeldesign und Fußgänger-Drucktaster*

Die Platine in Abbildung 5.1 zeigt - symbolisch - die Ampelkonfiguration einer Verkehrskreuzung. Diese besteht aus vier Ampeln für den Kraftfahrzeugverkehr (von denen jeweils die gegenüberliegenden Ampeln zusammengeschaltet sind) und den vier Ampeln für die Fußgängerüberwege. Die Letzteren zeigen alle stets dasselbe Signal. Das Problem erfordert also die Signalerzeugung für drei unterschiedliche Ampeln. Der im Folgenden erläuterte VHDL-Code generiert diese Signale und kann in einen CPLD oder FPGA geschrieben werden. Mit einem 1-MHz-Oszillator getaktet, erzeugt dieser dann an den gewählten Ausgängen die Steuersignale für die Ampelplatine.

Betrachten wir zunächst die Verlängerung der Ampelphasen aus dem Millisekundenbereich auf eine für eine reale Ampel benötigte, mehrere Sekunden andauernde Zeitspanne. Hierzu wird ein Frequenzteiler benötigt, der sich in VHDL einfach durch einen Zähler realisieren lässt. In diesem Fall wird die verwendete CLK-Frequenz um den Faktor 2^{20} verlangsamt. Dieser Teiler wird dann als Component im Hauptprogramm instanziiert.

```
------------------------------
-- Teiler
------------------------------

-- Bibliothek laden
library IEEE;
use IEEE.std_logic_1164.all;


-- Schnittstelle deklarieren
entity teiler is

    port(     CLK                 : in std_logic;
              RESET               : in std_logic;
              CLOCK_OUT           : out std_logic
        );

end teiler;



-- Architektur der Schaltung
architecture behavior of teiler is

  signal COUNT : integer range 0 to 1048575;

begin


 teiler: process (CLK, RESET)
    begin
         if RESET = '1' then
            COUNT <= 0;
            CLOCK_OUT <= '0';
         else
            if CLK = '1' and CLK'event then
               if COUNT < 524288 then
                   CLOCK_OUT <= '0';
               else
                   CLOCK_OUT <= '1';
               end if;
               if COUNT > 1048574 then
                   COUNT <= 0;
               end if;
               COUNT <= COUNT + 1;
             end if;
         end if;
    end process;


end behavior;
```

Die Ampelsteuerung selbst ist, wie die Ampel Version 6 aus den Programmierbeispielen, wieder als FSM realisiert, deren Signalzuweisungen des Ausgangsschaltnetzes abhängig vom aktuellen internen Zustand sind. Einen Überblick über alle Zustände der Ampel und die Abfolge der Ampelphasen gibt Abbildung 5.3. Die tatsächliche Signalfolge der einzelnen Ausgangsleitungen ist in Abbildung 5.4 als Screenshot aus der Simulation zu verfolgen.

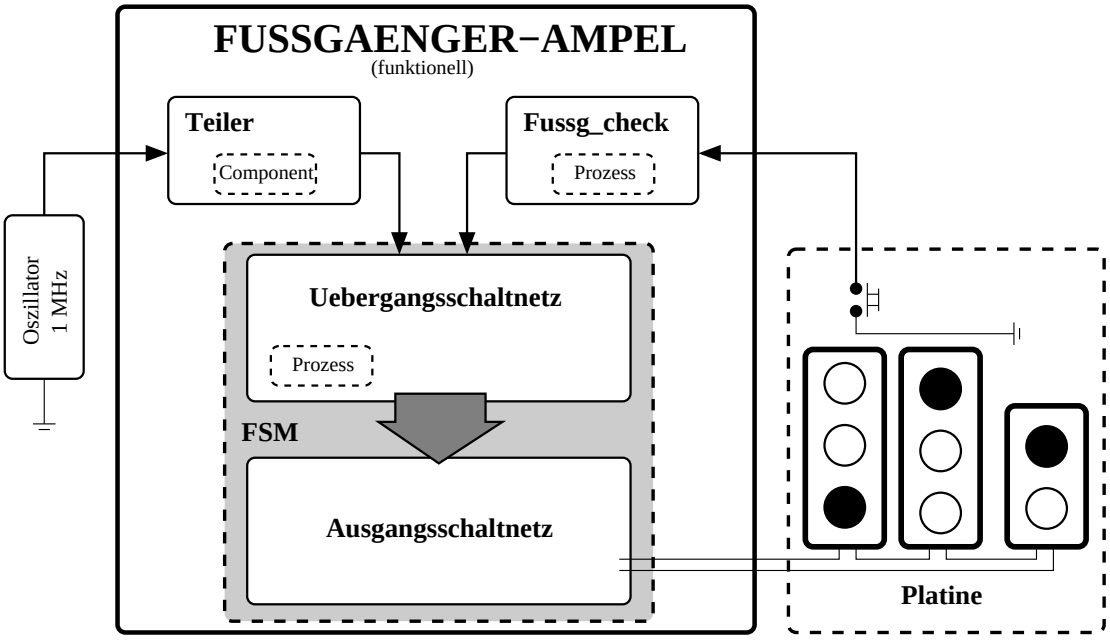


Abbildung 5.2: *Blockschaltbild der Ampelsteuerung mit Ampelplatine.*

```
------------------------------
-- Ampelsteuerung,  Version 7
------------------------------

-- Bibliothek laden
library IEEE;
use IEEE.std_logic_1164.all;

-- Schnittstelle deklarieren
entity ampel_fussg is
  generic (k : positive := 2);          -- Laenge der Gruenphasen
  port ( CLK          : in std_logic;
         RESET        : in std_logic;
         FUSSG_BUTTON : in std_logic;
         GRUEN_1      : out std_logic;
         GELB_1       : out std_logic;
         ROT_1        : out std_logic;
         GRUEN_2      : out std_logic;
         GELB_2       : out std_logic;
         ROT_2        : out std_logic;
         GRUEN_FUSSG  : out std_logic;
         ROT_FUSSG    : out std_logic
       );
end ampel_fussg;
-- Architektur der Schaltung
architecture STATE_MACHINE of ampel_fussg is

 component teiler
  port ( CLK     : in std_logic;
       RESET     : in std_logic;
       CLOCK_OUT : out std_logic
  );
 end component;

  type sreg0_type is (S0,S1,S2,S3,S4,S5,S6,S7,S8);   -- Typ, der die Zustaende der Machine enthaelt
  signal SREG0 : sreg0_type;                         -- Zustandsanzeiger
  signal FUSSG, FUSSG_BUTTON_OLD : std_logic;        -- FUSSG: Flag, wenn Fussgaenger gedrueckt hat
```

```
 signal CLOCK : std_logic;                          -- Ampelphasenumschalter
 signal COUNT1 : integer range 0 to 31;             -- verlaengert Gruenphasen
 signal CLEAR, RST, FUSSG_OFF, FUSSG_RST: std_logic;
begin

 RST <= (RESET or CLEAR);
 FUSSG_RST<=(RESET or FUSSG_OFF);


 -- Teiler fuer sekundenlange Ampelphasen bei 1 Mhz Oszillator
Ampelteiler: teiler
port map ( CLK => CLK,
           RESET => RESET,
           CLOCK_OUT => CLOCK
);

 -- check ob Fussgaenger gedrueckt hat.
 fussg_check: process (CLK, FUSSG_RST)
 begin
  if FUSSG_RST='1' then
   FUSSG<='0';
  else
   if RISING_EDGE(CLK) then
    if FUSSG_BUTTON = '1' and FUSSG_BUTTON_OLD = '0' then
     FUSSG<='1';
    end if;
    FUSSG_BUTTON_OLD<=FUSSG_BUTTON;
   end if;
  end if;
 end process fussg_check;
```

In diesem ersten Teil des VHDL-Codes befinden sich die Instanziierung des Frequenzteilers, sowie der Prozess 'fussg-check'. Dieser Prozess überprüft mit jeder ansteigenden CLK-Flanke, ob der Drucktaster für Fußgänger vom offenen in den geschlossenen, also gedrückten Zustand bewegt wurde. Im Gegensatz zu einem direkt vom Drucktaster getriggerten Prozeß, bei dem FUSSG-BUTTON in der Sensitivity-List enthalten wäre, bietet die hier verwendete Form gleich noch eine Entprellung des Tasters. Ansonsten würde der Prozess durch einen prellenden Taster oder Schalter mehrmals gestartet, was oft zu Komplikationen führt und vermieden werden sollte.

```
 -- Ampelphasenumschalter
 machine: process (RESET, CLOCK)
 begin
  if RESET='1' then                      -- Reset des Automaten
    SREG0<=S0;
  else
   if rising_edge(CLOCK) then            -- Zustandswechsel bei steigender CLOCK-Flanke
    case SREG0 is                        -- Ampel1 Ampel2  Fussg
     when S0 =>                          -- rot    gruen    rot
      if COUNT1=k then                   -- laengere Phase (faktor k)
       SREG0<=S1;
      else
       SREG0<=S0;
      end if;
     when S1 =>                          -- rot    gelb     rot
       SREG0<=S2;
     when S2 =>                          -- rot    rot      rot
       SREG0<=S3;
     when S3 =>                          -- rot/gelb rot    rot
       SREG0<=S4;
     when S4 =>                          -- gruen    rot    rot
      if COUNT1=k then                   -- laengere Phase (faktor k)
       SREG0<=S5;
      else
       SREG0<=S4;
      end if;
     when S5 =>                          -- gelb     rot    rot
```

```
       SREG0<=S6;
     when S6 =>                          -- rot     rot    rot
      if FUSSG='1' then                  -- wenn FUSSG gedrueckt, umschalten nach S7
       SREG0<=S7;                        -- sonst S8
      else
       SREG0<=S8;
      end if;
     when S7 =>                          -- rot     rot   gruen
      if COUNT1=k then                   -- laengere Phase (faktor k)
       SREG0<=S6;
      else
       SREG0<=S7;
      end if;
     when S8 =>                          -- rot   rot/gelb rot
      SREG0<=S0;
     when others =>
      SREG0<=S0;
    end case;
   end if;
 end if;
end process machine;
```

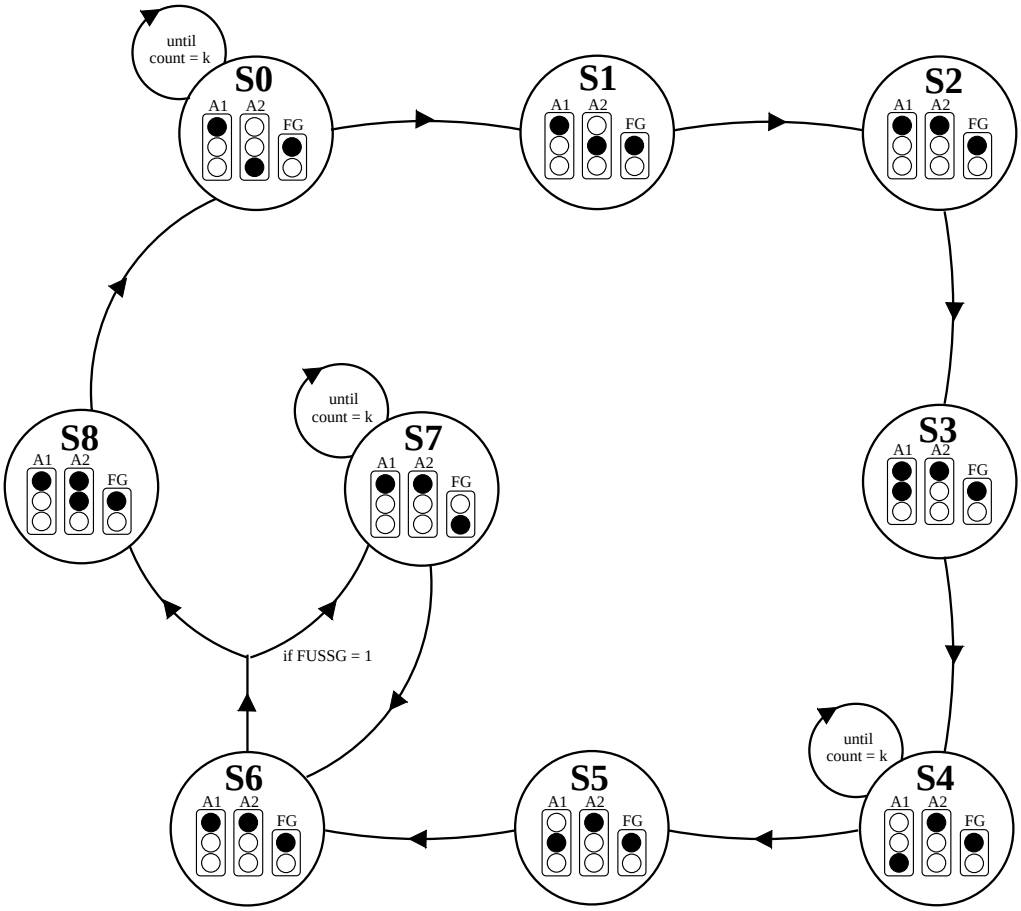


Abbildung 5.3: *Zustandsdiagramm der Ampelsteuerung. Wurde zu irgendeinem Zeitpunkt der Fussgänger-Drucktaster betätigt, so verzweigt die Ampel nach Zustand S6 einmalig zu S7 und wieder zurück.*

```
 -- Prozess verlaengert einzelne Ampelphasen um Faktor k
 counter: process (CLOCK,RST)
 begin
  if RST='1' then                       --Reset des Counters
   COUNT1<=0;
  else
   if rising_edge(CLOCK) then           --Erhoehen des Counters
    COUNT1<=COUNT1+1;                   --bei steigender CLOCK-Flanke
   end if;
  end if;
 end process counter;

-- Ausgansschaltnetz, Signalzuweisungen
GRUEN_1_assignment:
GRUEN_1 <= '0' when (SREG0 = S0) else
           '0' when (SREG0 = S1) else
           '0' when (SREG0 = S2) else
           '0' when (SREG0 = S3) else
           '1' when (SREG0 = S4) else
           '0' when (SREG0 = S5) else
           '0' when (SREG0 = S6) else
           '0' when (SREG0 = S7) else
           '0' when (SREG0 = S8) else
           '0';

GELB_1_assignment:
GELB_1 <= '0' when (SREG0 = S0) else
          '0' when (SREG0 = S1) else
          '0' when (SREG0 = S2) else
          '1' when (SREG0 = S3) else
          '0' when (SREG0 = S4) else
          '1' when (SREG0 = S5) else
          '0' when (SREG0 = S6) else
          '0' when (SREG0 = S7) else
          '0' when (SREG0 = S8) else
          '0';

ROT_1_assignment:
ROT_1 <=  '1' when (SREG0 = S0) else
          '1' when (SREG0 = S1) else
          '1' when (SREG0 = S2) else
          '1' when (SREG0 = S3) else
          '0' when (SREG0 = S4) else
          '0' when (SREG0 = S5) else
          '1' when (SREG0 = S6) else
          '1' when (SREG0 = S7) else
          '1' when (SREG0 = S8) else
          '0';

GRUEN_2_assignment:
GRUEN_2 <='1' when (SREG0 = S0) else
          '0' when (SREG0 = S1) else
          '0' when (SREG0 = S2) else
          '0' when (SREG0 = S3) else
          '0' when (SREG0 = S4) else
          '0' when (SREG0 = S5) else
          '0' when (SREG0 = S6) else
          '0' when (SREG0 = S7) else
          '0' when (SREG0 = S8) else
          '0';

GELB_2_assignment:
GELB_2 <= '0' when (SREG0 = S0) else
          '1' when (SREG0 = S1) else
          '0' when (SREG0 = S2) else
```

```
          '0' when (SREG0 = S3) else
          '0' when (SREG0 = S4) else
          '0' when (SREG0 = S5) else
          '0' when (SREG0 = S6) else
          '0' when (SREG0 = S7) else
          '1' when (SREG0 = S8) else
          '0';

ROT_2_assignment:
ROT_2 <=  '0' when (SREG0 = S0) else
          '0' when (SREG0 = S1) else
          '1' when (SREG0 = S2) else
          '1' when (SREG0 = S3) else
          '1' when (SREG0 = S4) else
          '1' when (SREG0 = S5) else
          '1' when (SREG0 = S6) else
          '1' when (SREG0 = S7) else
          '1' when (SREG0 = S8) else
          '0';

GRUEN_FUSSG_assignment:
GRUEN_FUSSG <= '0' when (SREG0 = S0) else
          '0' when (SREG0 = S1) else
          '0' when (SREG0 = S2) else
          '0' when (SREG0 = S3) else
          '0' when (SREG0 = S4) else
          '0' when (SREG0 = S5) else
          '0' when (SREG0 = S6) else
          '1' when (SREG0 = S7) else
          '0' when (SREG0 = S8) else
          '0';

ROT_FUSSG_assignment:
ROT_FUSSG <= '1' when (SREG0 = S0) else
          '1' when (SREG0 = S1) else
          '1' when (SREG0 = S2) else
          '1' when (SREG0 = S3) else
          '1' when (SREG0 = S4) else
          '1' when (SREG0 = S5) else
          '1' when (SREG0 = S6) else
          '0' when (SREG0 = S7) else
          '1' when (SREG0 = S8) else
          '0';

--Zaehler-Reset zur Verlaengerung einzelner Ampelphasen
CLEAR_assignment:
CLEAR <=  '0' when (SREG0=S0) else
          '1' when (SREG0 = S1) else
          '1' when (SREG0 = S2) else
          '1' when (SREG0 = S3) else
          '0' when (SREG0 = S4) else
          '1' when (SREG0 = S5) else
          '1' when (SREG0 = S6) else
          '0' when (SREG0 = S7) else
          '1' when (SREG0 = S8) else
          '0';

-- Ausschalten des Fussg-Flags wenn Fussgaenger:Gruen
FUSSG_OFF_assignment:
FUSSG_OFF<='1' when (SREG0=S7) else
           '0';

end STATE_MACHINE;
```

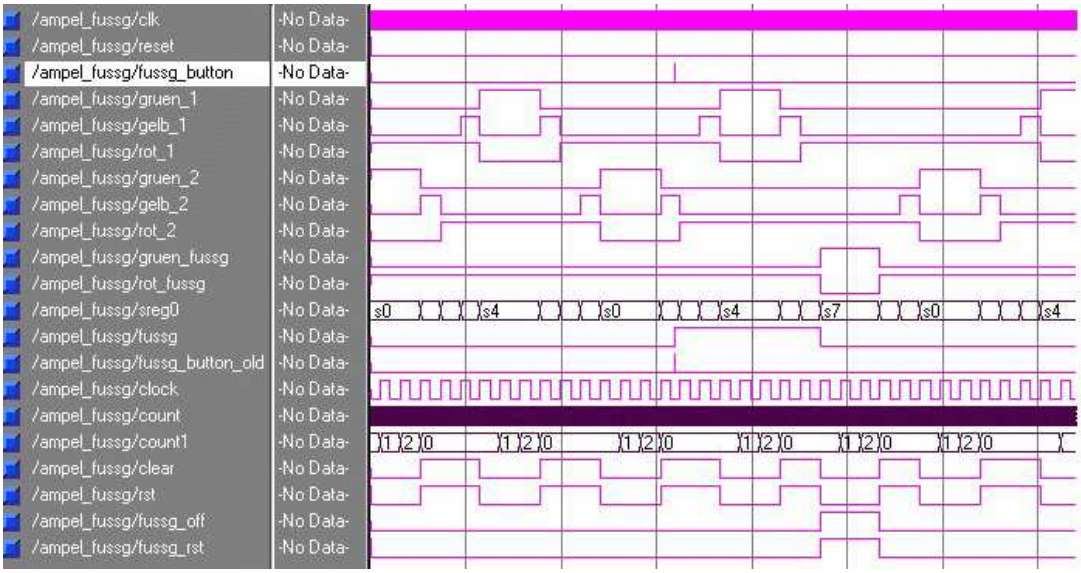


Abbildung 5.4: *Die generierten Steuersignale für die Ampelplatine in der Simulation. Zu erkennen sind die CLK-Frequenz (1 MHz), das millionenfach verlängerte CLOCK-Signal und die davon ausgelösten, aufeinanderfolgenden Ampelphasen. Ausserdem ist der durch das Drücken von FUSSG-BUTTON ausgelöste Fussgängerampel-Mechanismus (ab der Mitte der Simulation) zu verfolgen.*

5.2 Frequenzzähler

Als weiteres Beispiel einer praktischen Anwendung wird der Aufbau eines 10 MHz Frequenzzählers vorgestellt. Der Zähler verdeutlicht, wie praxistauglich die Programmierung von CLPDs via VHDL heute ist. Mit wenigen Seiten VHDL-Code entsteht hier ein alltagstauglicher Zähler, für den früher ein aufwendiger digitaler Schaltungsentwurf mit zahlreichen Einzelkomponenten oder einem Spezial-IC notwendig gewesen wäre. Die Grundschaltung ist dank des eingesetzten CPLD sehr kompakt und umfaßt neben dem CPLD nur einen 10 MHz Clock-Oszillator. Zusammen mit optionalen Vorteilern und Signalverstärkern kann daraus leicht ein eigenständiges Gerät für den Laborbetrieb entstehen.

Mit dem hier gezeigten Frequenzzähler läßt sich die Frequenz eines Signales im Bereich 1 Hz - 20 MHz auf 1 Hz genau messen (abhängig von der Genauigkeit des Oszillators). Der Frequenzzähler ist mit einem 10 MHz-Oszillator getaktet und besitzt eine Torzeit von einer Sekunde. In der Grundschaltung (mit dem CPLD alleine) können nur Signale mit 5 Volt TTL-Pegel gemessen werden.

Der Frequenzzähler ist intern aus einer Kette von sieben einzelnen BCD-Zählern aufgebaut (s. Abb. 5.5), wobei der Übertrag jedes BCD-Zählers mit dem Eingang des nachfolgenden Zählers (nächsthöhere Stelle) verbunden ist. Nach Ablauf der Torzeit wird der aktuelle Wert aller Zähler in ein Register geschrieben und anschließend auf null zurückgesetzt. Die Ausgabe des Registers erfolgt dann auf einem Display mit acht 7-Segmentanzeigen. Die hierfür erforderlichen 8x8 Anschlußleitungen können auf nur insgesamt 16 Anschlüsse reduziert werden, wenn ein Anzeigenmultiplexer verwendet wird. Der Multiplexer schaltet dabei alle Millisekunde die Segmente und den jeweils dort anzuzeigenden Registerwert über einen BCD-to-7-Segment-Decoder auf die Ausgabepins. Die sieben BCD-Zähler sind jeder für sich in Form einer State-Machine (FSM) realisiert, deren interner Zustand den aktuellen Zählerstand repräsentiert. Zur Ausgabe wird dieser Zustand in 4 Output-Bits (BCD) codiert. Als Eingang dient dem ersten Zähler direkt die zu zählende Frequenz, die nachfolgenden Zähler werden jeweils vom Carry-Bit[1] des vorhergehenden Zählers getrieben.

[1] Das Carry-Bit wird beim Übergang des Zählerstandes von neun auf null gesetzt

```
-------------------------------------------
-- Dezimal Zaehler   -- April 2004
-------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity DECCOUNTER is

port    (     CNTIN     : in std_logic;
              RESET     : in std_logic;
              CLEAR     : in std_logic;
              OUTPUT    : out std_logic_vector(3 downto 0);
              CARRY     : out std_logic);
end;


architecture behavior of DECCOUNTER is

-- symbolic encoded state machine: SREG0
type cnt_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9);
signal CNT0    : cnt_type;

begin

    dec_counter: process(CNTIN,RESET)    -- 50 % effektiver als funktioneller Dezimalzaehler

    begin
        if (RESET = '0') or (CLEAR = '1') then
               CNT0  <= S0;
               CARRY <= '0';
        else
            if rising_edge(CNTIN) then
               case CNT0 is
                  when S0 =>
                      CARRY <= '0';
                      CNT0 <= S1;
                  when S1 =>
                      CARRY <= '0';
                      CNT0 <= S2;
                  when S2 =>
                      CARRY <= '0';
                      CNT0 <= S3;
                  when S3 =>
                      CARRY <= '0';
                      CNT0 <= S4;
                  when S4 =>
                      CARRY <= '0';
                      CNT0 <= S5;
                  when S5 =>
                      CARRY <= '0';
                      CNT0 <= S6;
                  when S6 =>
                      CARRY <= '0';
                      CNT0 <= S7;
                  when S7 =>
                      CARRY <= '0';
                      CNT0 <= S8;
                  when S8 =>
                      CARRY <= '0';
                      CNT0 <= S9;
                  when S9 =>
                      CARRY <= '1';
                      CNT0 <= S0;
               end case;
            end if;
         end if;
```

```
     end process;

 OUTPUT <= "0000" when (CNT0 = S0) else    -- nur 4 Bit. Umwandlung auf 8 Bit fuer 7 Segment
           "0001" when (CNT0 = S1) else    -- Anzeige erfolgt im Top-Level im BCD to 7 Segment
           "0010" when (CNT0 = S2) else    -- decoder.
           "0011" when (CNT0 = S3) else
           "0100" when (CNT0 = S4) else
           "0101" when (CNT0 = S5) else
           "0110" when (CNT0 = S6) else
           "0111" when (CNT0 = S7) else
           "1000" when (CNT0 = S8) else
           "1001" when (CNT0 = S9) else
           "0000";

end behavior;
```

Die einzelnen Dezimalzähler werden als Component im Frequenzzähler instantiiert. Obwohl acht 7-Segmentanzeigen vorhanden sind, werden nur sieben BCD-Zähler verwendet - auf der vordersten Anzeige wird nur das Carry-Bit des siebten Zählers, also eins oder null ausgegeben.

```
----------------------------------------------------------------
-- Frequenzzaehler mit Multiplexed Display Driver -- April 2004
----------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity COUNTER is
    generic ( k             : in integer := 6
            );

port    (     CLK       : in std_logic;
              RESET     : in std_logic;
              INPUT     : in std_logic;
              DISPLAY   : out std_logic_vector(7 downto 0);
              ANZEIGE   : out std_logic_vector(7 downto 0)
            );
end;


architecture behavior of COUNTER is

component DECCOUNTER
    port    ( CNTIN             : in std_logic;
              RESET             : in std_logic;
              CLEAR             : in std_logic;
              OUTPUT            : out std_logic_vector(3 downto 0);
              CARRY             : out std_logic
            );
end component;


-- symbolic encoded state machine: SREG0
type sreg0_type is (S0, S1, S2, S3, S4, S5, S6, S7);

signal SREG0     : sreg0_type;
signal COUNT     : integer range 0 to 10000;
signal GCOUNT    : integer range 0 to 10000000;
signal MUX       : std_logic;
signal CARRY     : std_logic_vector(k downto 0);
signal CLEAR     : std_logic;
signal DISNUMB   : std_logic_vector(31 downto 0);
signal STONUMB   : std_logic_vector(31 downto 0);
signal BCDIN     : std_logic_vector(3 downto 0);


begin
```

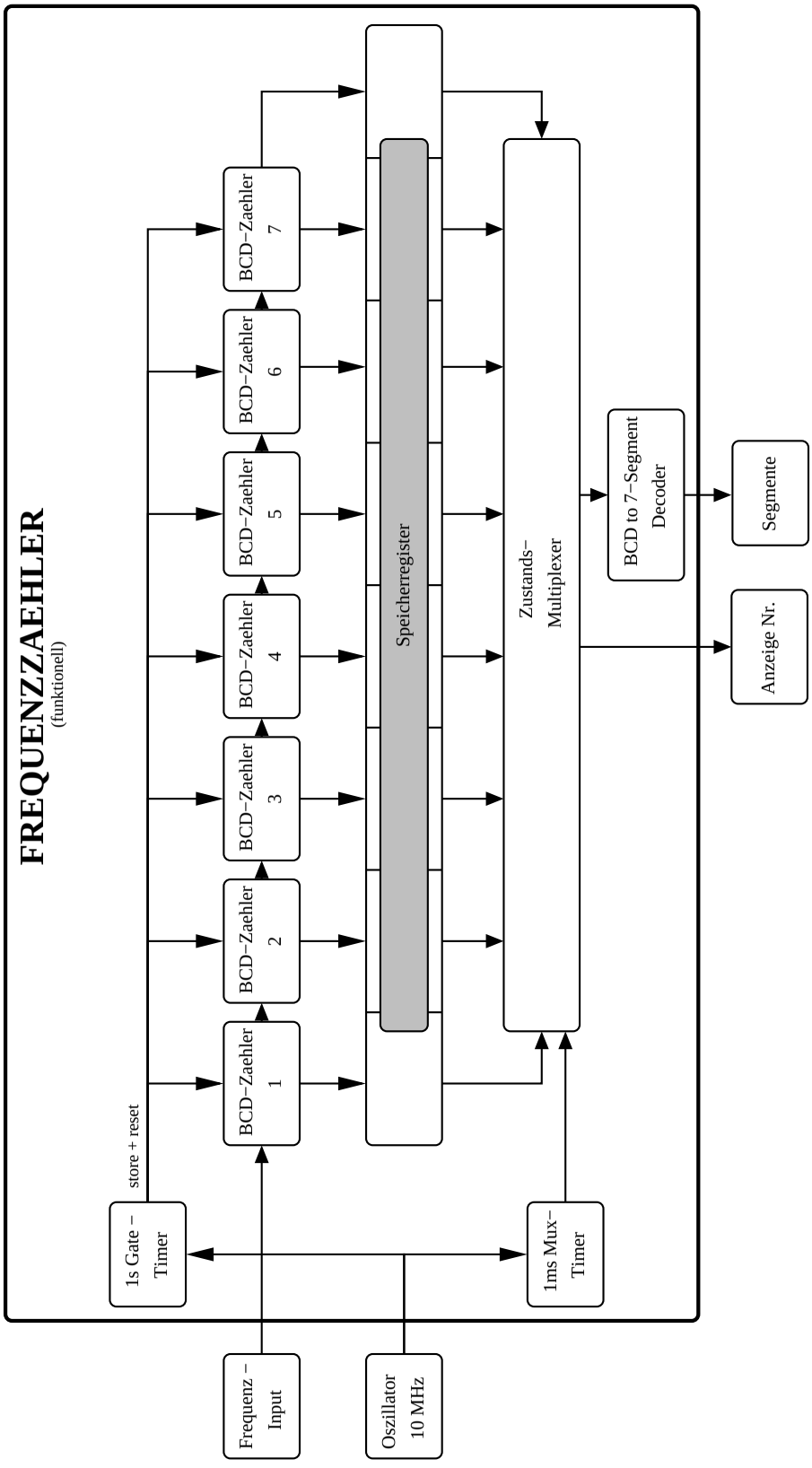


Abbildung 5.5: *Blockschaltbild des Frequenzzählers. Für den Entwurf ist ein ALTERA MAX 7000 EPM7128SLC84 erforderlich.*

```
---
-- 6 BCD - Zaehler
---

counter: for i in 0 to k generate

  first: if i = 0 generate
      cnt0:DECCOUNTER
          port map    (CNTIN         => INPUT,
                       RESET         => RESET,
                       CLEAR         => CLEAR,
                       OUTPUT        => DISNUMB((4*(i+1))-1 downto (4*i)),
                       CARRY         => CARRY(i)
                      );
  end generate;


  rest: if i > 0 generate
       cnti:DECCOUNTER
           port map    (CNTIN         => CARRY(i-1),
                        RESET         => RESET,
                        CLEAR         => CLEAR,
                        OUTPUT        => DISNUMB((4*(i+1))-1 downto (4*i)),
                        CARRY         => CARRY(i)
                        );
       end generate;

end generate;
```

Der Multiplex-Timer reduziert den Takt der CLK-Frequenz von 10 MHz auf 1 kHz für den unten erläuterten Zustands-Multiplexer. Der Gate-Timer sorgt dafür, dass nach Ablauf einer Sekunde der aktuelle Wert der Zähler in die Speicher-Register geschrieben und anschliessend auf null zurückgesetzt wird.

```
---
--  Multiplex Timer
---
muxcounter: process (CLK, RESET)
begin
     if RESET = '0' then
        COUNT <= 0;
        MUX   <= '0';
     else
        if rising_edge(CLK) then
           -- multiplex counter
           if COUNT < 5000 then
              MUX <= '1';
           else
              MUX <= '0';
           end if;
           if COUNT > 9999 then
              COUNT <= 0;
           end if;
           COUNT <= COUNT + 1;     -- besser als down counter und vgl auf Null!
         end if;
     end if;
end process;


---
--  Gate Timer
---
gatecounter: process (CLK, RESET)
begin
   if RESET = '0' then
      GCOUNT   <=  0;
      CLEAR    <= '0';
```

```
      STONUMB  <= (others => '0');
   else
      if rising_edge(CLK) then
         -- multiplex counter
         if GCOUNT < 10000000 then
            CLEAR   <= '0';
         else
            GCOUNT  <= 0;
            CLEAR   <= '1';
         end if;
         if GCOUNT = 10000000 then
            STONUMB <= DISNUMB;
            if CARRY(k) = '1' then
               STONUMB(31 downto 28) <= "0001";
            else
               STONUMB(31 downto 28) <= "0000";
            end if;
         end if;
         GCOUNT <= GCOUNT + 1;
      end if;
   end if;
end process;
```

Der Zustands-Multiplexer ist eine State-Machine, die - vom Multiplex-Timer angetrieben - ihre internen Zustände zyklisch durchläuft. Dadurch werden in Folge die einzelnen Speicher-Register ausgelesen, durch den BCD-Converter geschickt, sowie die zugehörige 7-Segmentanzeige eingeschaltet.

```
---
-- Zustands Multiplexer
---
s_machine: process(CLK,RESET)
     variable last_MUX : std_logic;
begin
    if rising_edge(CLK) then
        if RESET     = '0' then
           SREG0    <= S0;
           last_MUX := '0';
        else
           case SREG0 is
              when S0 =>
                   BCDIN <= STONUMB(3  downto  0);  -- In Register BCDIN speichern mit CLK
                   if MUX = '1' and last_MUX = '0' then
                      SREG0 <= S1;  -- State schaltet mit MUX
                   end if;
                   last_MUX := MUX;
              when S1 =>
                   BCDIN <= STONUMB(7  downto  4);  -- In Register BCDIN speichern mit CLK
                   if MUX = '1' and last_MUX = '0' then
                      SREG0 <= S2;
                   end if;
                   last_MUX := MUX;
              when S2 =>
                   BCDIN <= STONUMB(11 downto  8);  -- In Register BCDIN speichern mit CLK
                   if MUX = '1' and last_MUX = '0' then
                      SREG0 <= S3;
                   end if;
                   last_MUX := MUX;
              when S3 =>
                   BCDIN <= STONUMB(15 downto 12);  -- In Register BCDIN speichern mit CLK
                   if MUX = '1' and last_MUX = '0' then
                      SREG0 <= S4;
                   end if;
                   last_MUX := MUX;
              when S4 =>
                   BCDIN <= STONUMB(19 downto 16);  -- In Register BCDIN speichern mit CLK
                   if MUX = '1' and last_MUX = '0' then
                      SREG0 <= S5;
```

```
                    end if;
                    last_MUX := MUX;
               when S5 =>
                    BCDIN <= STONUMB(23 downto 20);  -- In Register BCDIN speichern mit CLK
                    if MUX = '1' and last_MUX = '0' then
                       SREG0 <= S6;
                    end if;
                    last_MUX := MUX;
               when S6 =>
                    BCDIN <= STONUMB(27 downto 24);  -- In Register BCDIN speichern mit CLK
                    if MUX = '1' and last_MUX = '0' then
                       SREG0 <= S7;
                    end if;
                    last_MUX := MUX;
               when S7 =>
                    BCDIN <= STONUMB(31 downto 28);  -- In Register BCDIN speichern mit CLK
                    if MUX = '1' and last_MUX = '0' then
                       SREG0 <= S0;
                    end if;
                    last_MUX := MUX;
          end case;
       end if;
    end if;
 end process;

---
-- Signal Zuweisung - Multiplex Generator
---
anzeige0_assignment:
ANZEIGE(0) <= '0' when (SREG0 = S0) else
           '1';

anzeige1_assignment:
ANZEIGE(1) <= '0' when (SREG0 = S1) else
           '1';

anzeige2_assignment:
ANZEIGE(2) <= '0' when (SREG0 = S2) else
           '1';

anzeige3_assignment:
ANZEIGE(3) <= '0' when (SREG0 = S3) else
           '1';

anzeige4_assignment:
ANZEIGE(4) <= '0' when (SREG0 = S4) else
           '1';

anzeige5_assignment:
ANZEIGE(5) <= '0' when (SREG0 = S5) else
           '1';

anzeige6_assignment:
ANZEIGE(6) <= '0' when (SREG0 = S6) else
           '1';

anzeige7_assignment:
ANZEIGE(7) <= '0' when (SREG0 = S7) else
           '1';

---
-- BCD to 7 Segment decoder  -- keinen Prozess verwenden, sonst jedesmal Register!!!
---
DISPLAY(7 downto 0) <= "10001000" when (BCDIN = "0000") else  -- Der Dezimalzaehler erzeugt
                       "11011011" when (BCDIN = "0001") else  -- 4 Bit BCD Code anstatt 8 Bit
                       "10100010" when (BCDIN = "0010") else  -- 7 Segment Code!
                       "10010010" when (BCDIN = "0011") else  -- spart Netze da der
```

```
                              "11010001" when (BCDIN = "0100") else  -- Dezimalzaehler 8 mal
                              "10010100" when (BCDIN = "0101") else  -- instanziiert wird !!!
                              "10000100" when (BCDIN = "0110") else
                              "11011010" when (BCDIN = "0111") else
                              "10000000" when (BCDIN = "1000") else
                              "10010000" when (BCDIN = "1001");

end behavior;
```

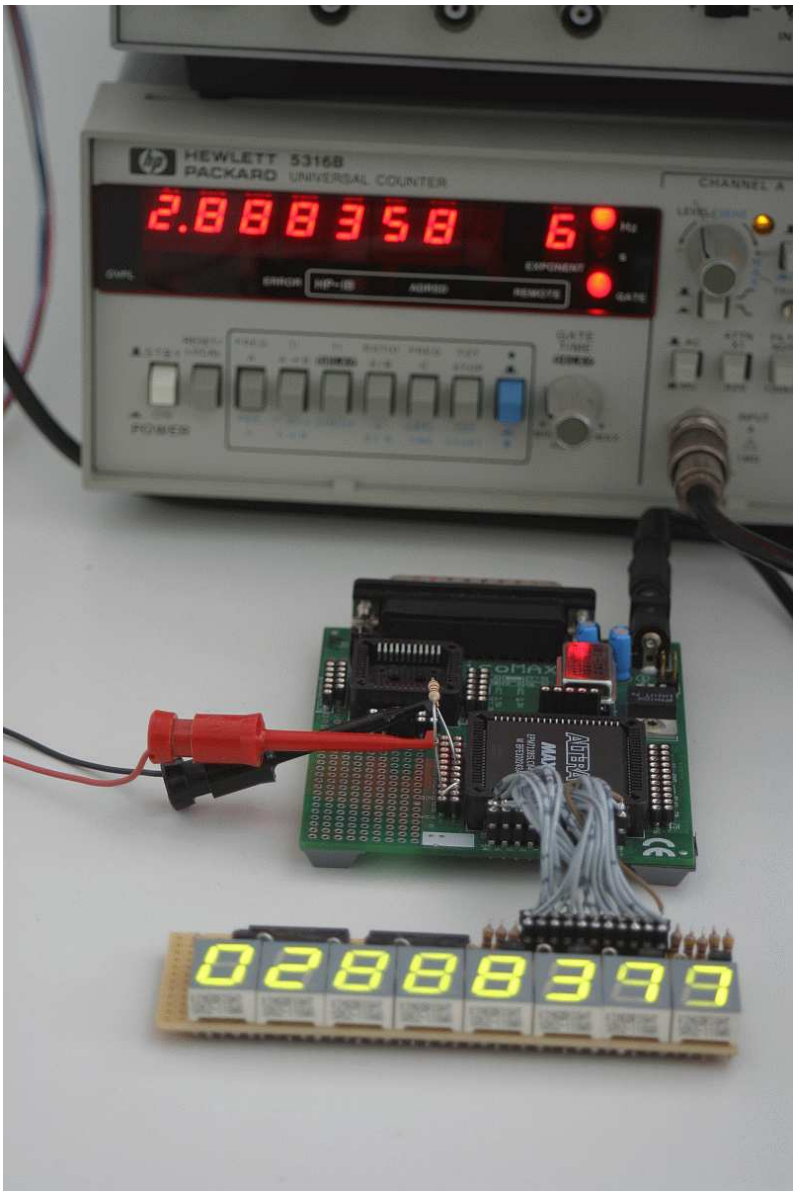


Abbildung 5.6: *Der Frequenzzähler im Betrieb mit einem Funktionsgenerator. Der Generator erzeugt das erforderliche Eingangssignal mit 5 Volt TTL-Pegel. Zur Kontrolle wurde zusätzlich ein kommerzieller Frequenzzähler betrieben.*

Literaturverzeichnis

[Ashenden 1998] Ashenden, P.J., 1998, The Student's Guide to VHDL, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-520-7.

[Ashenden 2002] Ashenden, P.J., 2002, The Designers Guide to VHDL, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-674-2.

[Lehmann et al. 1994] Lehmann, G., Wunder, B., Selz, M., 1994, Schaltungsdesign mit VHDL. Fanzis Verlag, ISBN 3-7723-6163-3, oder im Internet als PDF-Dokument: unter http://tech-www.informatik.uni-hamburg.de/vhdl/ oder http://www.eda.ei.tum.de/forschung/vhdl/

[Altera MAX3000] MAX 3000 CPLD, http://www.altera.com/products/devices/max3k/m3k-index.html

[Altera MAX7000] MAX 7000 CPLD, http://www.altera.com/products/devices/max7k/m7k-index.html Datasheet: http://www.altera.com/literature/ds/m7000.pdf

[El Camino] El Camino DIGILAB picoMAX, http://www.elca.de/

Index